Friedrich Steimann, Daniela Keller, Sebastian Küpper

Modul 63613

Moderne Programmiertechniken und -methoden

Lektionen 1–7

Leseprobe

Fakultät für
Mathematik und Informatik

FernUniversität in Hagen

Inhaltsverzeichnis

Vorwort

Was ist modern? Das, was gerade angesagt ist? Dann müsste diese Vorlesung jedes Semester neu geschrieben werden. Was dann?

Manches Wissen der Informatik hat eine erschreckend kurze Halbwertszeit. Das gilt auch für den Bereich der Softwareentwicklung und der Programmiersysteme: Die Programmiersprache der Wahl schien bis vor wenigen Jahren JAVA zu sein, heute rücken deren Derivate wie C#, SCALA oder KOTLIN stärker in den Fokus und wer nicht anders kann, verwendet JAVASCRIPT. Aspektorientierte Programmierung schickte sich noch vor 10 Jahren an, die Softwareentwicklung zu revolutionieren, heute ist es bereits auf dem Abstellgleis der großen Hoffnungen gelandet. Was also gehört in eine Vorlesung, die das Attribut „modern" trägt?

Meine Antwort darauf heißt: Wissen, das man vor Jahren noch nicht hatte, das Sie aber voraussichtlich auch noch nutzen können, wenn Sie Ihr Studium abgeschlossen haben und wenn Sie dann (wieder) mitten in einem Programmierprojekt stecken. Ihr hier erworbenes Wissen entspricht dann sicher nicht dem aktuellen Hype, aber es hat hoffentlich noch eine gewisse Aktualität (während es sich mit dem Hype von heute vielleicht längst erledigt hat). Um konkreter zu werden: Das Wissen dieses Kurses sollte eine Halbwertszeit von zehn Jahren haben, d. h., die Hälfte dessen, das Sie heute lernen, sollte in zehn Jahren noch gültig und verwertbar sein.

Genderisierung

Da Friedrich Steimann in seinen Kurstexten eine persönliche Ansprache pflegt, ist dieser Kurs im generischen Neutrum verfasst. Das bedeutet, dass im Text bei Ansprachen, die in der deutschen Sprache ein grammatikalisches Geschlecht besitzen, die Grundform, aber nicht mit dem standardsprachlich üblichen Maskulin, sondern dem Neutrum verwendet wird Eine Beispielsformulierung wäre „das Leser des Kurstextes". Ich wähle diese Form, um alle Geschlechter gleich zu behandeln und gleichzeitig die Geschlechtlichkeit in den vielen Kontexten, in denen sie sachlich keine Rolle spielt, bewusst *nicht* zu betonen.

App zum Kurs

Für diesen Kurs, mit Ausnahme der neuen Kurseinheit 3 gibt es eine **Papier/Digital-Brücke** in Form einer experimentellen Android-App, mit deren Hilfe Sie sich das Nachschlagen von Begriffen im Index ersparen und die Links im Rand direkt verfolgen können. Außerdem liest die App den Kurs seitenweise vor und tut dabei allerlei Dinge, die das Vorleseergebnis verbessern und die der Barrierefreiheit dienen. Probieren Sie sie aus! Kurseinheit 3 wird stattdessen von einem Vorlesungsvideo begleitet.

Übersicht

Interfacebasierte Programmierung

Der Kurs beginnt mit einem etwas eigenwilligen Thema, nämlich der sog. *interfacebasierten Programmierung*. Diese propagiert die Verwendung von Interfaces (als Typen wie in JAVA oder C#) anstelle von Klassen bei der Typisierung von Variablen (also in Variablendeklarationen). Das Konzept und die Verwendung von Interfaces sind immer wiederkehrende Themen in den folgenden Kurseinheiten; die Einführung der interfacebasierten Programmierung gleich zu Anfang scheint daher gerechtfertigt, selbst wenn es sich bei ihr ausdrücklich nicht um ein Standardthema handelt.

Design by contract

Interfaces à la JAVA und C# sind unvollständig. Was ihnen fehlt, ist eine (formale) Beschreibung dessen, was die Einhaltung des Interfaces über die rein syntaktischen Methodensignaturen hinaus verlangt, gewissermaßen eine Semantik der Methoden oder, anders gesagt, eine genaue, damit verbundene Verhaltensspezifikation. *Design by contract* ist ein besonders griffig formuliertes Prinzip, dieses Defizit auszugleichen: Ihm zufolge wird über ein Interface ein Vertrag geschlossen, nach dem beiden Seiten — gewissermaßen über Kreuz — gegenseitige Verpflichtungen und Nutzen zugeschrieben werden. Die Einhaltung dieses Vertrages kann über sog. *Zusicherungen* zur Laufzeit und — in ausgewählten Fällen — auch statisch, also zur Übersetzungszeit, geprüft werden. All dies ist Gegenstand der zweiten Kurseinheit.

Model Checking

Die Korrketheit einer Software-Lösung relativ zu einer Spezifikation zu überprüfen geschieht üblicherweise auf zwei verschiedene Weisen: Testen oder Verifikation. In vergangenen Fassungen dieses Kurses wurde das Unit-Testen eingehend betrachtet, doch das Testen hat einen entscheidenden Nachteil: Mit Tests kann ein Programm nur auf Fehler überprüft werden, aber die Korrektheit nicht positiv festgestellt werden. Die Verifikation hingegen stellt die Korrektheit einer Lösung mathematisch sicher. Allerdings ist Verifikation dennoch keine Universallösung, da das Verifikationsproblem im Allgemeinen unentscheidbar ist. Wir werden uns daher auf eine etablierte Verifikationstechnik konzentrieren, die allerdings Testen leider nicht in jedem Anwendungsfall ersetzen kann: Das Model Checking. Die grundlegende Idee ist, dass das Software-Designer die Anforderungen an die Software mittels spezieller Logiken spezifiziert und die intendierte Lösung mit einem abstrakten Modell gestaltet, so dass es anschließend das Modell gegen die logischen Formeln verifizieren kann.

Entwurfsmuster

Die vierte Kurseinheit widmet sich dann der Idee der *Entwurfsmuster*. Entwurfsmuster bieten zunächst einen Katalog von Standardlösungen für häufig wiederkeh-

rende Entwurfsprobleme; sie bilden aber ganz nebenbei auch ein Vokabular, das es Softwareentwicklern erlaubt, sich kurz und prägnant über Software zu unterhalten, und zwar ohne sich in Details zu verlieren, nämlich unter Verweis auf bekannte Konzepte (ganz so, wie sich Drehbuchautoren und Produzenten in Robert Altmans „The Player" über Filme unterhalten).

Refactoring

Auch wenn es aus Managementsicht bedauerlich sein mag: Die Programmierung ist ein evolutionärer Prozess, bei dem einmal getroffene Entscheidungen ständig auf die Probe gestellt und gegebenenfalls wieder geändert werden müssen. Die Änderung von Code ist aber in der Regel eine verzweigte und entsprechend verzwickte Angelegenheit: Nur selten ist es mit einer Änderung an einer Stelle getan. Das führt dann regelmäßig zu der Erkenntnis, dass Software unter Änderung verdirbt. Dem sollen sog. *Refactorings*, standardisierte und teilweise auch automatisierte Änderungen von Code, die dessen Bedeutung nicht verändern, entgegenwirken. Ja noch viel mehr: Mit Hilfe von Refactorings soll sich der Entwurf (und damit die Qualität) existierender Software durch gezielte Änderungen nachträglich verbessern lassen.

Metaprogrammierung

Als Abschluss der Programmiertechniken wird noch ein anderes Thema aufgegriffen, das — in Zeiten immer heterogener Systemlandschaften — immer mehr an Bedeutung gewinnt: die *Metaprogrammierung*. Unter Metaprogrammierung versteht man zunächst die Erstellung von Programmen, die (andere) Programme erzeugen, lesen oder ändern; aber auch die Fähigkeit, zur Laufzeit Information über ihren eigenen Aufbau preiszugeben, kann für Programme wichtig sein, z. B. wenn Programme, die unabhängig voneinander entwickelt wurden, zusammenarbeiten sollen. Besonders interessant wird die Metaprogrammierung dann, wenn ein Programm sich selbst zu verändern in der Lage ist. Dies ist in gewisser Weise bei der *aspektorientierten Programmierung* der Fall.

Agile Softwareentwicklung

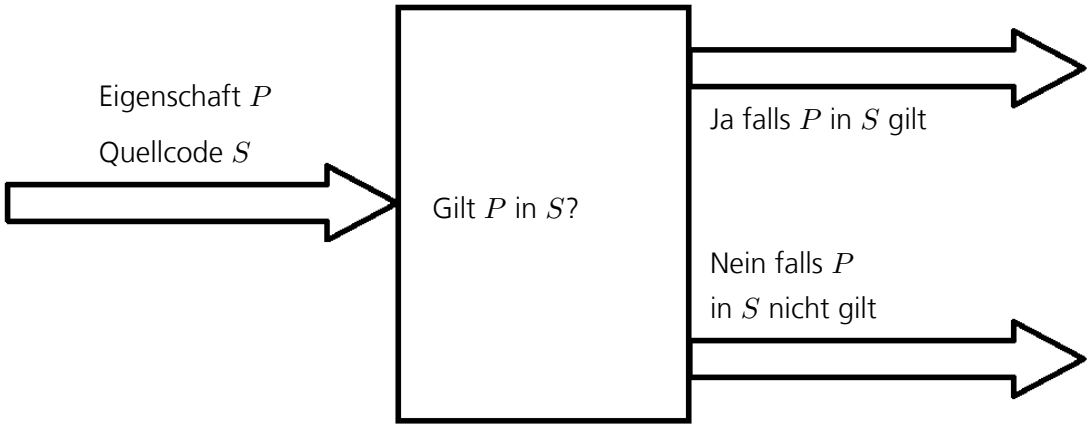
Unit-Tests und Refactorings sind zwei Programmiertechniken, die im Rahmen einer bestimmten Programmiermethode größere Bekanntheit erlangt haben, nämlich des *Extreme Programming*. Extreme Programming vereint eine Vielzahl relativ unorthodoxer Herangehensweisen zu einem Gesamtkunstwerk, dessen Praxistauglichkeit in der Vergangenheit viel diskutiert wurde. Ohne dass heute ein abschließendes Ergebnis vorläge, kann das Extreme Programming als Wegbereiter von „agilen Methoden" oder Prozessen wie Scrum oder Kanban verstanden werden und damit von einer immer stärker werdenden Abkehr von schwergewichtigen Softwareentwicklungsansätzen, die, zumindest aus Sicht der Programmierer, zu einer Art Befreiung von der Bürokratie hergebrachter Prozessmodelle geführt hat. Auch wenn die agile Softwareentwicklung längst nicht auf alle Projekte und Organisationen passt, so lässt sich doch sicher die eine oder andere Anregung daraus mitnehmen.

Kurseinheit 3: Verifikation mit CTL-Model Checking

Eine zentrale Fragestellung in der Entwicklung von Software-Systemen ist im Regelfall die Korrektheit des Systems. Bei sicherheitskritischen Systemen, beispielsweise der Steuersoftware einer Rakete oder eines Atomkraftwerks ist es unbedingt notwendig, dass die Software genau das gewünschte Verhalten zeigt. Anderenfalls können Menschenleben gefährdet oder große wirtschaftliche Schäden hervorgerufen werden. Aber selbst bei Software, die im Fall einer Fehlfunktion nicht unmittelbar zu fatalen Konsequenzen führt, ist die Frage der Korrektheit spätestens bei der Vergütung der Entwicklungstätigkeit nicht unerheblich.

Um die Korrektheit eines Software-Systems zu überprüfen, gibt es zwei grundsätzliche Ansätze: Software kann entweder verifiziert oder getestet werden. Ziel der Verifikation ist es, einen absoluten Beweis für die Korrektheit zu erbringen, wohingegen beim Testen vor allem das Identifizieren von Fehlern im Vordergrund steht. In diesem Kurs soll das Testen als Technik zur Überprüfung der Korrektheit eines Software-Systems allerdings nicht in den Fokus gerückt werden. Testverfahren werden ausführlich in den Veranstaltungen des Lehrgebiets Softwaretechnik diskutiert. Stattdessen konzentrieren wir uns im vorliegenden Kurs ausschließlich auf Verifikationstechniken.

Idealerweise stellt man sich als Software-Entwickler Verifikation von Software wahrscheinlich ungefähr wie folgt vor: Gegeben sei ein Software-System, im einfachsten Fall unmittelbar in Form des Quelltextes, sowie eine formale Beschreibung der gewünschten Eigenschaft des Systems, beispielsweise in Form von Vor- und Nachbedingungen oder in Form eines logischen Ausdrucks. Das Entwickler gibt beide Eingaben in eine bereits vorweg implementierte Blackbox, drückt einen Knopf mit der Aufschrift „verifizieren“, wartet eine Weile und erhält am Ende entweder die Ausgabe „ja“ oder „nein“ – je nachdem ob die Eigenschaft erfüllt ist oder nicht. Exemplarisch sei diese Idealvorstellung in der nachfolgenden Abbildung dargestellt.



Dass diese Idealvorstellung in der Praxis leider nicht uneingeschränkt erreicht werden kann, folgt aus der Unentscheidbarkeit bereits einfacher Eigenschaften wie Terminierung. In den Kursen der Theoretischen Informatik werden zahlreiche Probleme vorgestellt, die ebenfalls unentscheidbar sind.

Hinweise zur Literatur Die Inhalte dieser Kurseinheit basieren auf verschiedenen Quellen, einzelne Themenbereiche folgen allerdings jeweils einer Hauptquelle. Der Abschnitt zu Transitionssystemen und Verhaltensäquivalenzen folgt in weiten Teilen den Darstellungen in den Kursmaterialien „Modellierung nebenläufiger Systeme".

Besonderer Dank gilt Barbara König für die Genehmigung der stellenweise engen Orientierung an ihren Kursmaterialien.

Strukturhinweis In Hinsicht auf Umfang und Schwierigkeitsgrad entspricht diese Kurseinheit zweien statt nur einer Kurseinheit. Sie ersetzt komplett die Kurseinheit zum Testen, statt aber noch eine zweite Kurseinheit zu ersetzen, wurden zwei andere Kurseinheiten umfassend gekürzt. Berücksichtigen Sie dies bitte auch bei Ihrer Prüfungsvorbereitung, da die vorliegende Kurseinheit demnach auch in der Gewichtung bei der Abschlussprüfung ein doppeltes Gewicht besitzt.

3.1 Transitionssysteme

Verifikation via Model Checking geschieht nicht direkt am Quellcode, sondern auf einer Abstraktion des Programms. Unser Ziel ist es, Model Checking mit CTL einzuführen. Hierzu benötigen wir das Modell der Transitionssysteme.

3.1.1 Transitionssysteme: Definition und Beispiele

In der Theoretischen Informatik haben Sie bereits nichtdeterministische endliche Automaten kennengelernt, die zur Beschreibung von regulären Sprachen dienen. Ganz ähnlich zu nichtdeterministischen Automaten sind Transitionssysteme definiert:

Definition 3.1 (Transitionssystem)**.** *Ein Transitionssystem ist ein Dreitupel $(Z, A, \rightarrow)$, wobei*

- *Z eine Menge von Zuständen ist*
- *A eine Menge von Aktionen ist und*
- *$\rightarrow \subseteq Z \times A \times Z$ die Menge der Transitionen ist.*

Wir schreiben $z \xrightarrow{a} z'$, falls $(z, a, z') \in \rightarrow$ gilt und schreiben $z_1 \xrightarrow{a_1 a_2 \ldots a_n} z_{n+1}$ falls es Zustände $z_2, z_3, ..., z_n$ gibt, so dass $z_1 \xrightarrow{a_1} z_2 \xrightarrow{a_2} z_3 ... \xrightarrow{a_n} z_{n+1}$ gilt. Es gilt mit dieser Konvention $z \xrightarrow{\epsilon} z$ für alle Zustände $z \in Z$. Wenn uns der erreichte Zustand nicht interessiert, schreiben wir zudem für alle $w \in A^*$ kurz $z \xrightarrow{w}$ falls es einen Zustand z' gibt, so dass $z \xrightarrow{w} z'$ und anderenfalls, wenn es einen solchen Zustand z' nicht gibt, $z \not\xrightarrow{w}$.

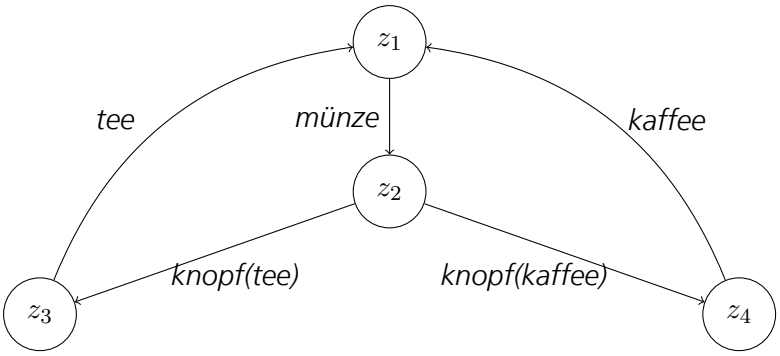
Transitionssysteme unterscheiden sich von nichtdeterministischen Automaten in zweierlei Hinsicht: Es gibt keine Start- oder Endzustände und die Menge der Zustände und Alphabetsymbole ist nicht zwingend endlich. Dementsprechend werden wir sehen, dass der Begriff der Sprache eines Transitionssystems sich ein wenig von der eines nichtdeterministischen Automaten unterscheidet. Allerdings halten wir an dieser Stelle bereits fest, dass endliche Transitionssysteme, d.h. solche, bei denen die Zustandsmenge und das Alphabet endlich sind, als Modellierungswerkzeug die Ausdrucksmächtigkeit im Vergleich zu turingmächtigen Systemen einschränken. Für endliche Transitionssysteme umgehen wir also die Problematik der Unentscheidbarkeit bereits auf Modellierungsebene, können im Gegenzug aber nicht alle Systeme mit (endlichen) Transitionssystemen modellieren.

Ein klassisches Beispiel, mit dem Transitionssysteme illustriert werden, ist eine Maschine für Heißgetränke, die dem Kunden die Wahl aus den Getränken Tee und Kaffee lässt:

Beispiel 3.2. *Wir können einen Automaten für Heißgetränke wie folgt als Transitionssystem modellieren:*

$$(\{z_1, z_2, z_3, z_4\}, \{kaffee, münze, tee, taste(kaffee), taste(tee)\}, \rightarrow),$$

wobei wir die Überführungsfunktion $\rightarrow$ wie folgt grafisch repräsentieren.



Diese grafische Notation für Transitionssysteme werden wir in Beispielen häufig verwenden, da sie, jedenfalls bei hinreichend kleinen Transitionssystemen, wesentlich suggestiver ist, als eine Darstellung als Formel.

Ähnlich wie bei Automaten gibt es auch im Falle von Transitionssystemen eine deterministische Variante.

Definition 3.3 (Deterministisches Transitionssystem)**.** *Ein Transitionssystem* $(Z, A, \rightarrow)$ *ist deterministisch, falls für alle* $z \in Z$*,* $a \in A$ *gilt, dass* $z \xrightarrow{a} z_1$ *und* $z \xrightarrow{a} z_2$ *impliziert, dass* $z_1 = z_2$*.*

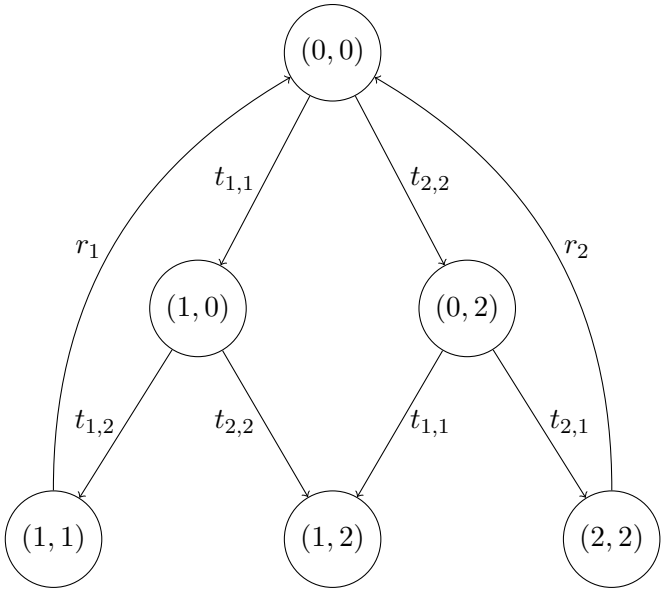
Dementsprechend ist unser zuvor besprochenes Beispiel eines Kaffee- und Tee-Automaten deterministisch. Beachten Sie, dass, anders als bei deterministischen Automaten, nicht gefordert wird, dass für jedes Paar von Zustand und Aktion ein Nachfolgezustand existiert.

Wir wollen nun einen ersten Blick auf ein Beispiel werfen, in dem ein Transitionssystem zur Modellierung eines Protokolls und zur Identifikation eines Fehlers in diesem Protokoll dient. Das Problem der Verklemmung (engl. Deadlock) ist ein typischer Softwarefehler in einem verteilten System. Wenn mehrere Prozesse unabhängig voneinander auf eine geteilte Ressource zugreifen können sollen, kann dieser Fehler auftreten, wie anhand des Bildes der dinierenden Philosophen (engl. dining philosophers) illustriert wird. Was viele Menschen über Philosophen nämlich nicht wissen[1], ist der Umstand, dass Philosophen zum Essen immerzu zwei Gabeln benötigen – das motorische Geschick wird durch das angestrengte Denken offenbar in Mitleidenschaft gezogen. Nun kommt es aber gelegentlich vor, dass Philosophen sich zu einem gemeinsamen Abendessen verabreden, um über gewichtige Probleme zu debattieren. Zu diesem Zwecke bietet sich natürlich ein runder Tisch in einem Restaurant an, der den maximalen Abstand zwischen Paaren von Philosophen minimiert. Leider wissen aber insbesondere viele Gastronomen nichts von den Essgewohnheiten von Philosophen, so dass es vorkommen kann, dass für jedes Philosophen nur eine einzige Gabel vorgesehen ist, die sich dann auf der linken Seite ihres Tellers befindet. Zum Glück für die Philosophen bedeutet das an einem runden Tisch allerdings, dass jedes Philosoph zwei Gabeln neben seinem Teller liegen hat. In der beschriebenen Situation kann ein Philosoph also durchaus essen, indem es zunächst nach der Gabel links neben seinem Teller greift, anschließend die Gabel rechts von seinem Teller aufnimmt, mit seinen beiden Esswerkzeugen eine Weile isst und anschließend seine Gabeln wieder auf dem Tisch platziert.

Im Laufe des Abends können die Philosophen sich nun in zwei Zuständen befinden, sie können hungrig sein oder in Diskussionslaune sein. Wenn ein Philosoph in Diskussionslaune ist, diskutiert es und verwendet hierzu keine Gabeln. Wenn es aber hungrig ist, versucht es wie oben beschrieben zwei Gabeln aufzunehmen und zu essen. Gelingt dem Philosophen nicht, nach der Aufnahme der linken Gabel auch die rechte Gabel vom Tisch aufzunehmen, weil sie gerade in Verwendung ist, wartet das Philosoph geduldig, bis die zweite Gabel wieder verfügbar ist und nimmt sie dann auf, um seinen Magen zu füllen. Die Philosophen können unabhängig voneinander hungrig werden und Gabeln aufnehmen. Wir

[1] Dem Vernehmen nach hängt das damit zusammen, dass diese Aussage in der Realität gar nicht der Wahrheit entspräche.

wollen nun diese Situation als Transitionssystem modellieren, um zu untersuchen, ob das beschriebene System eine Verklemmung besitzt, ob also passieren kann, dass Philosophen wechselseitig aufeinander warten. Angenommen wir betrachten die Situation im einfachsten Fall mit nur zwei Philosophen, dann können wir die Situation wie folgt modellieren: Zustände sind Paare $(g_1, g_2) \in \{0, 1, 2\} \times \{0, 1, 2\}$ mit der folgenden Bedeutung: g_1 zeigt an, ob die Gabel des ersten Philosophen gerade auf dem Tisch liegt (dann ist $g_1 = 0$), vom ersten Philosophen gehalten wird (dann ist $g_1 = 1$) oder vom zweiten Philosophen gehalten wird (dann ist $g_1 = 2$). Analog zeigt g_2 den Status der Gabel des zweiten Philosophen an. Es gibt also insgesamt $3^2 = 9$ Zustände. Weiterhin gibt es die Aktionen $t_{i,j}$ und r_i mit $i, j \in \{1, 2\}$. Die Aktion $t_{i,j}$ bedeutet, dass das i-te Philosoph die j-te Gabel aufnimmt (t für engl. take) und r_i bedeutet, dass das i-te Philosoph beide Gabeln zurücklegt (r für engl. return). Betrachten wir nun den zugehörigen Graphen, der die Transitionsfunktion anzeigt, so sehen wir, dass von der Startsituation $(0, 0)$ aus die Situation $(1, 2)$ erreichbar ist, von der aus keine weitere Aktion durchgeführt werden kann, es gibt also einen Deadlock im Transitionssystem.



Diese Situation entspricht dem ungünstigen Szenario, dass das erste Philosoph hungrig geworden ist und seine Gabel aufnimmt. Noch bevor es aber nach rechts zur zweiten Gabel greifen kann, wird auch der zweite Philosoph hungrig und greift nach seiner Gabel. Nun warten beiden Philosphen wechselseitig aufeinander und leiden ohne Eingreifen von außen beliebig lange Hunger.

Analog lässt sich das Szenario auch für eine größere Zahl an Philosophen modellieren, in jedem Fall können aber Verklemmungen auftreten. Im betrachteten Beispiel gibt es verschiedene Lösungsmöglichkeiten für das Verklemmungsproblem:

- Eines der Philosophen nimmt immer zuerst die rechte Gabel auf, mindestens ein anderes erst die linke.

- Allgemeiner: Man definiert eine Prioritätenfolge auf den Gabeln, die für alle Philosophen gilt und verlangt, dass die Gabeln immer nur in dieser Reihenfolge aufgenommen werden können.
- Ein Philosoph muss stets beide Gabeln auf einmal aufnehmen.
- Philosophen können nicht beliebig lange mit einer Gabel in der Hand warten, sondern müssen nach einer Weile die Gabel wieder ablegen.

Wie eingangs erwähnt, ist das Bild der dinierenden Philosophen ein abstraktes Modell für Schwierigkeiten, die in einem Netzwerk mit geteilten Ressourcen auftreten können und die Lösungsstrategien entsprechen tatsächlichen Lösungsstrategien für Verklemmungen in nebenläufigen Systemen. Die ersten beiden Punkte entsprechen der Definition einer Prioritätenfolge auf geteilten Ressourcen, der dritte Punkt der Einführung größerer atomarer Aktionen und der vierte Lösungsvorschlag entspricht einem Timeout. Unser Ziel soll es sein, derartige Probleme bereits am Modell zu erkennen und somit in der Implementation des Systems von vorneherein zu vermeiden.

3.1.2 Verhaltensäquivalenzen für Transitionssysteme

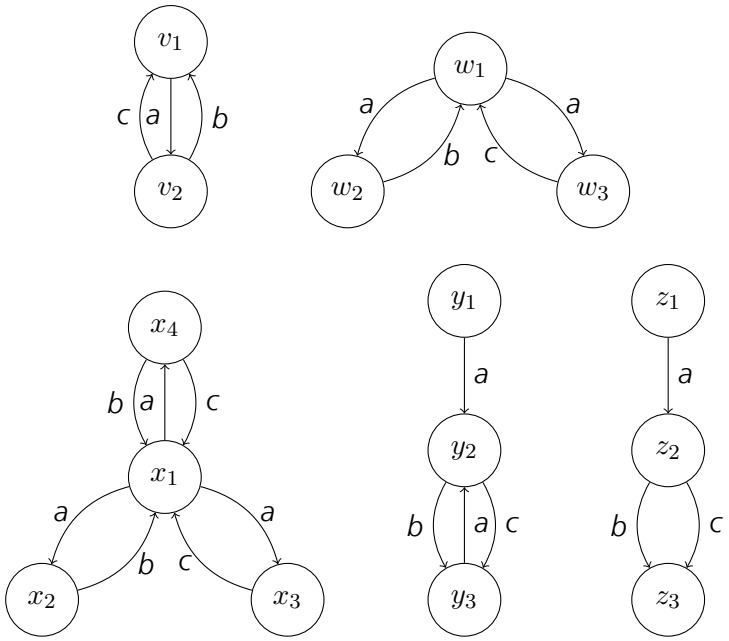
Zur Analyse von Transitionssystemen müssen wir uns zunächst darüber Gedanken machen, was das Verhalten eines Transitionssystems charakterisieren soll. Wir beginnen damit, verschiedene Ansätze, das Verhalten eines Zustandes zu definieren und zu vergleichen. Sie kennen aus der theoretischen Informatik bereits den Begriff der Sprachäquivalenz für nichtdeterministische Automaten, der sich nahezu unmittelbar auf Transitionssysteme übertragen lässt. Allerdings werden wir uns mit verschiedenen Varianten der Sprachäquivalenz beschäftigen, denn wenn wir uns einmal unsere Eingangsbeispiele zu Transitionssystemen in Erinnerung rufen, werden wir Transitionssysteme vor allem dazu verwenden, beliebig lang laufende Systeme zu modellieren. Daher kann es sinnvoll sein, nicht nur endliche Abläufe zu betrachten.

Definition 3.4 (Sprachäquivalenz)**.** *Es sei $(Z, A, \rightarrow)$ ein Transitionssystem und $z \in Z$ ein Zustand des Transitionssystems. Wir definieren nun drei verschiedene Sprachen des Zustandes z und verwenden die Schreibweisen A^* für alle endlichen Sequenzen über A, A^ω für alle unendlichen Sequenzen über A und A^∞ für alle (unendlichen oder endlichen) Sequenzen über A:*

- $S(z) = \{w \in A^* \mid \exists z' \in Z : z \xrightarrow{w} z'\}$
- $S^\omega(z) = \{a_1 a_2 a_3 ... \in A^\omega \mid \exists z_1, z_2, z_3, ... : z \xrightarrow{a_1} z_1 \xrightarrow{a_2} z_2 \xrightarrow{a_3} z_3 ...\}$
- $S^\infty(z) = S(z) \cup S^\omega(z)$

Zwei Zustände $z, z' \in Z$ sind S-sprachäquivalent (S^ω-sprachäquivalent, S^∞-sprachäquivalent), wenn die jeweiligen S-Sprachen (S^ω-Sprachen, S^∞-Sprachen) gleich sind, in Zeichen: $S(z) = S(z')$ ($S^\omega(z) = S^\omega(z')$, $S^\infty(z) = S^\infty(z')$).

Aufgabe 1. *Für diese und zwei weitere Selbsttestaufgaben verwenden wir die folgenden fünf Transitionssysteme:*



1. *Sind die Zustände v_1 und w_1 sprachäquivalent? Geben Sie hierzu die jeweils generierte Sprache $S(v_1)$ und $S(w_1)$ an.*
2. *Sind die Zustände w_1 und x_1 sprachäquivalent? Geben Sie hierzu die jeweils generierte Sprache $S(w_1)$ und $S(x_1)$ an.*
3. *Sind die Zustände v_1 und z_1 sprachäquivalent? Geben Sie hierzu die jeweils generierte Sprache $S(v_1)$ und $S(z_1)$ an.*
4. *Sind die Zustände v_1 und y_1 sprachäquivalent? Geben Sie hierzu die jeweils generierte Sprache $S(v_1)$ und $S(y_1)$ an.*

Lösung für Aufgabe 1: Wir geben zunächst die Sprachen der gefragten Zustände an:

$$\begin{aligned} &S(v_1) = S(w_1) = S(x_1) = S(y_1) \\ =&\{w \in \{a,b,c\}^* \mid \forall 1 \leq i \leq |w| : i \mod 2 = 1 \Leftrightarrow w[i] = a\} \end{aligned}$$

Alternativ kann man diese Sprache auch als regulären Ausdruck aufschreiben:
$L((a(b|c))^*(a|\epsilon))$.
Weniger formell ist es die Menge aller Wörter, die an ungeraden Stellen ein a und an geraden Stellen ein b oder ein c besitzen. Die Sprache $S(z_1) = \{\epsilon, a, ab, ac\}$ ist hingegen endlich. Damit kann man unmittelbar die Lösung der Aufgabenteile einsehen:

1. Ja.

2. Ja.
3. Nein.
4. Ja.

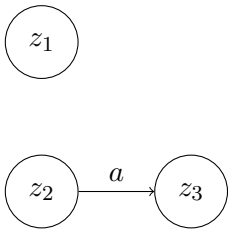
Es ist offensichtlich, dass zwei Zustände S^∞-sprachäqivalent sind, genau dann wenn sie sowohl S-sprachäquivalent, als auch S^ω-sprachäquivalent sind: Ist $S(x) = S(x')$ und $S^\omega(x) = S^\omega(x')$, so ist auch $S(x) \cup S^\omega(x) = S(x') \cup S^\omega(x')$. Umgekehrt beachten wir, dass $S(x)$ und $S^\omega(x)$ disjunkt sind für alle x, da $S(x)$ nur endliche Sequenzen enthält und S^ω nur unendliche Sequenzen enthält. Also gilt, wenn $S^\infty(x) = S^\infty(x')$ ist:

$$\begin{aligned} S(x) &= (S^\omega(x) \cup S(x)) \cap A^* = S^\infty(x) \cap A^* \\ &= S^\infty(x') \cap A^* = (S^\omega(x') \cup S(x')) \cap A^* = S(x') \end{aligned}$$

und analog

$$\begin{aligned} S^\omega(x) &= (S^\omega(x) \cup S(x)) \cap A^\omega = S^\infty(x) \cap A^\omega \\ &= S^\infty(x') \cap A^\omega = (S^\omega(x') \cup S(x')) \cap A^\omega = S^\omega(x'). \end{aligned}$$

Ein wenig schwieriger einzusehen ist, dass S-Sprachäquivalenz nicht S^ω-Sprachäquivalenz impliziert und umgekehrt, dass S^ω-Sprachäquivalenz nicht S-Sprachäquivalenz impliziert. Ein Beispiel zu finden für zwei Zustände, die S^ω-sprachäquivalent sind aber nicht S-sprachäquivalent sind, ist allerdings unproblematisch, wenn man sich vor Augen führt, dass alle Paare von Zuständen, die keine unendlichen Wörter akzeptieren, unmittelbar als S^ω-sprachäquivalent eingesehen werden können. Das einfachste Beispiel für diesen Umstand wären also die beiden folgenden Zustände z_1, z_2:



Von z_1 aus sind keine Aktionen möglich, aber von z_2 aus kann man eine a-Aktion durchführen, also gilt $S(z_1) = \{\epsilon\} \neq \{\epsilon, a\} = S(z_2)$. Allerdings kann weder von z_1 aus noch von z_2 aus eine unendlich lange Folge von Transitionen durchgeführt werden, also gilt trivial $S^\omega(z_1) = \emptyset = S^\omega(z_2)$.

Für die umgekehrte Richtung, zwei Zustände, die S-sprachäquivalent, aber nicht S^ω-sprachäquivalent sind, benötigen wir unendlich viele Zustände.

Satz 3.5. *In einem Transitionssystem mit endlich vielen Zuständen gilt: Wenn zwei Zustände z und z' S-sprachäquivalent sind, dann sind sie auch S^ω-sprachäquivalent.*

Beweis. Der Grund hierfür ist einfach ersichtlich, wenn man sich vor Augen führt, dass jedes endliche Präfix eines unendlichen Wortes ein endliches Wort ist. Damit zwei Zustände $z, z' \in Z$ existieren können, die nicht S^ω-sprachäquivalent sind, muss einer der

beiden Zustände, o.B.d.A. z einen unendlichen Pfad besitzen, dessen Wort w vom anderen Zustand z' aus nicht eingelesen werden kann. Wenn die beiden Zustände zusätzlich S-sprachäquivalent sind, müssen aber alle endlichen Präfixe des Wortes w von z' aus akzeptiert werden. Wir konstruieren nun einen unendlichen w-Pfad $z_0, z_1, z_2...$ von z' aus.

Es gilt $z_0 = z$. Die (einelementige) Sequenz $S_0 = z_0$ hat die Eigenschaft, dass es für jedes endliche Präfix w' von w einen w'-Pfad gibt, der mit $S_0 = z_0$ beginnt.

Sei nun eine Sequenz S_i der Länge $i \in \mathbb{N}$ gegeben, die die Eigenschaft hat, dass jedes endliche Präfix w' von w entweder von einem Präfix von S_i akzeptiert wird, oder (wenn w' länger als i ist) S_i Präfix eines Pfades ist, der w' akzeptiert. Wir wollen nun einen Zustand z_{i+1} wählen, so dass die Sequenz $S_i z_{i+1}$ diese Eigenschaft behält.

Hierzu verwenden wir das unendliche Taubenschlag- oder Schubfachprinzip, das besagt: Wenn man eine unendliche Zahl an Elementen (hier: Präfixe des Wortes w) auf endlich viele Äquivalenzklassen verteilen möchte (hier: Zustände), dann muss es mindestens eine unendlich große Äquivalenzklasse geben.

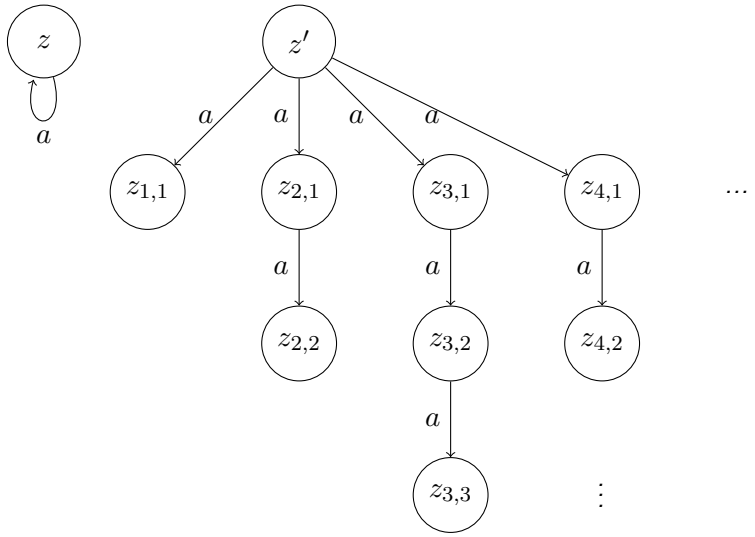
Wir betrachten nun also die Menge aller Präfixe w' von w, die mehr als i Zeichen enthalten, sowie deren zugehörigen Sequenzen $S_{w'}$, so dass S_i ein Präfix von $S_{w'}$ ist und $S_{w'}$ das Wort w' akzeptiert. In jeder dieser Sequenzen gibt es einen Zustand, der durch das Einlesen des $i+1$-ten Zeichens von w' erreicht wird. Mindestens einer dieser Zustände, nennen wir ihn $\hat{z}$, wird nach obigem unendlichen Taubenschlagprinzip unendlich oft erreicht. Wir setzen $z_{i+1} = \hat{z}$.

Wir müssen nun zeigen, dass die so erhaltene Sequenz $S_i z_{i+1}$ die in der Induktionsvoraussetzung gegebene Bedingung auch tatsächlich erfüllt. Sei hierzu ein beliebiges Präfix $\overline{w}$ von w gegeben. Wenn $\overline{w}$ maximal die Länge $i+1$ hat, ist nichts zu zeigen. Ist $\overline{w}$ hingegen länger als $i+1$, so ist zu zeigen, dass es eine Fortsetzung von $S_i z_{i+1}$ gibt, die $\overline{w}$ akzeptiert. Da $S_i z_{i+1}$ Präfix unendlich vieler Sequenzen $S_{w'}$ ist, die ein Präfix w' von w mit mehr als i Zeichen akzeptieren, gibt es ein Präfix $\tilde{w}$, das länger ist als $\overline{w}$, so dass $S_i z_{i+1}$ ein Präfix von $S_{\tilde{w}}$ ist und $S_{\tilde{w}}$ das Wort $\tilde{w}$ akzeptiert. Da $\overline{w}$ ein Präfix von $\tilde{w}$ ist, muss das Präfix der Länge $|w'| + 1 > i + 1$ von $S_{\tilde{w}}$ das Wort w' akzeptieren.

Wir haben also eine unendliche Sequenz von Zuständen konstruiert, die jedes Präfix von w akzeptiert, also akzeptiert die Sequenz insgesamt auch w. □

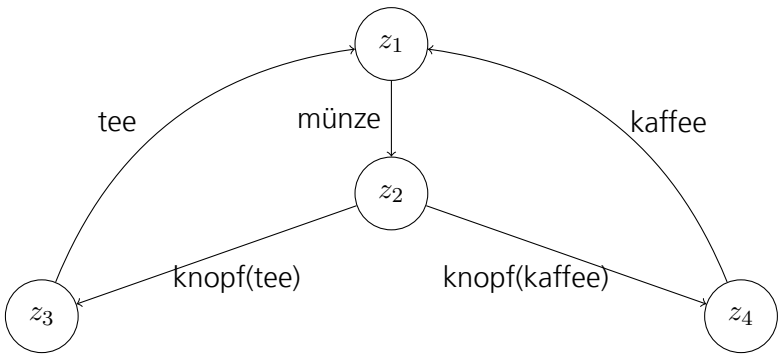
Dieser Beweis gibt uns auch bereits einen Anhaltspunkt, wie ein Beispiel aussehen kann, bei dem S^ω-Sprachäquivalenz nicht S-Sprachäquivalenz impliziert.

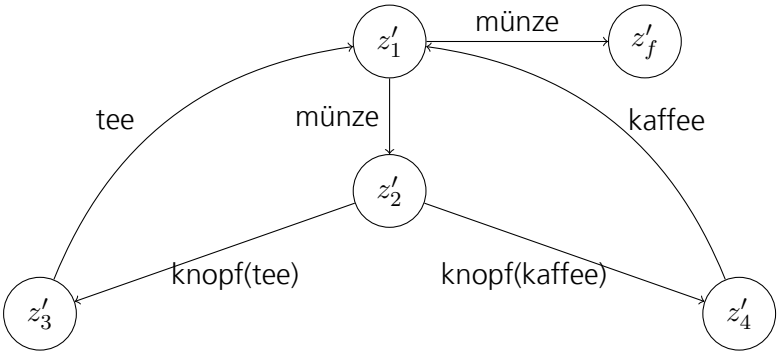
Beispiel 3.6. *Wir benötigen einen Zustand z, der einen unendlichen Pfad für das unendliche Wort w besitzt und einen weiteren Zustand z', der auf Pfaden, die ab einem bestimmten Index disjunkt sind, für jede endliche Wortlänge n das Präfix der Länge n von w akzeptieren. Im einfachsten Fall wird das unendliche Wort unmittelbar durch eine Schleife gebildet und das Transitionssystem kann beispielsweise wie folgt aussehen:*



Es gibt also für jedes $n \in \mathbb{N}$ eine eigene Sequenz von Zuständen $z_{n,1}z_{n,2}...z_{n,n}$, so dass $z'z_{n,1}z_{n,2}...z_{n,n}$ genau das Wort a^n akzeptiert. $S(z) = \{a^n \mid n \in \mathbb{N}_0\} = S(z')$, aber $S^\omega(z) = \{a^\omega\} \neq \emptyset = S^\omega(z')$.

Wir haben bislang Adaptionen des Begriffs Sprachäquivalenz in Anlehnung an die Automatentheorie kennen gelernt. Allerdings reicht es uns bei der Analyse von Software-Systemen in aller Regel nicht, nur zu überprüfen, welche Aktionen *möglich* sind, manchmal ist es, im Falle von nichtdeterministischen Systemen, gleichsam interessant, zu untersuchen, welche Aktionen *unmöglich* sind. Zu diesem Zweck betrachten wir zunächst einmal das folgende motivierende Beispiel, das unser einführendes Beispiel eines Heißgetränkeautomaten wieder aufgreift. Neben dem ursprünglichen Beispiel (oben) betrachten wir noch eine alternative Modellierung (unten), bei der die Möglichkeit modelliert wird, dass der Getränkeautomat in einen Fehlerzustand übergeht.





Die Sprachen der Zustände z_1 und z_1' sind gleich: Beide Graphen sind isomorph mit Ausnahme des zusätzlichen Zustandes z_f' und des zugehörigen münze-Übergangs im rechten Automaten. Allerdings ermöglicht dieser keine zusätzlichen Transitionsfolgen, da vom Zustand z_1' aus bereits ein münze-Übergang existiert. Allerdings verhalten sich beide Automaten aus Sicht eines Benutzers durchaus unterschiedlich. Während im ursprünglichen Transitionssystem sichergestellt ist, dass auf jeden Münzeinwurf irgendwann die Auswahl aus Kaffee und Tee erfolgen kann, kann das rechte Transitionssystem nach Münzeinwurf blockieren. Selbst ungeachtet des Umstandes, dass man auf diese Weise keine Heißgetränke mehr erhalten kann eine unangenehme Situation.

Nach dieser Überlegung wollen wir nun eine weitere Verhaltensäquivalenz für Transitionssysteme kennen lernen, die Verweigerungsäquivalenz (engl. failures equivalence).

Definition 3.7 (Verweigerungsäquivalenz)**.** *Es sei $(Z, A, \to)$ ein Transitionssystem und $z \in Z$ ein beliebiger Zustand, dann bezeichnen wir die folgende Menge als die Menge von Verweigerungspaaren:*

$$V(z) = \{(w, A') \in A^* \times \mathcal{P}(A) \mid \exists z' \in Z : z \xrightarrow{w} z' \wedge \ \forall \ \ a \in A' : z' \not\xrightarrow{a}\}$$

Wir nennen zwei Zustände z, z' verweigerungsäquivalent oder V-äquivalent, falls $V(z) = V(z')$ gilt.

Die Verweigerungspaare eines Zustandes z sind also die Paare aller Wörter w und Aktionen $A' \subseteq A$, so dass man von z aus das Wort w einlesen kann und in einem Zustand z' landen kann, so dass man anschließend keine Transition mit einem Label a ausführen kann. Die Menge $V(z)$ ist oft unendlich, weil A^* unendlich ist. Es ist wichtig zu beachten, dass im Falle eines nichtdeterministischen Transitionssystems die Menge $V(z)$ maximale Paare (w, A') und (w, A'') enthalten kann – maximal bedeutet hier, dass es kein $\overline{A}' \supseteq A'$ gibt mit $(w, \overline{A}') \in V(z)$ – die verschieden sind. Der Grund hierfür ist, dass mit dem gleichen Wort w verschiedene Zustände erreicht werden können, die jeweils potentiell verschiedene Eingaben verweigern.

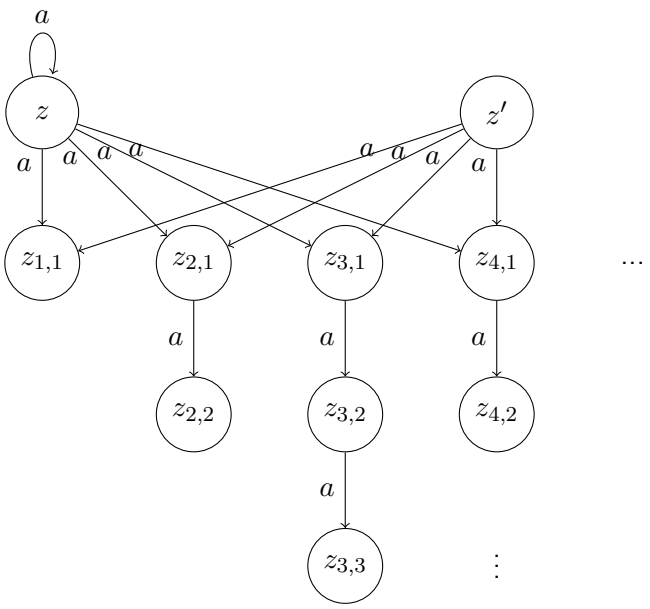
Betrachten wir unser obiges Beispiel, so können wir beobachten, dass die Zustände z_1 und z_1' nicht verweigerungsäquivalent sind, denn es gilt $(\text{münze}, A) \in V(z_1')$ aber $(\text{münze}, A) \notin V(z_1)$

Wir haben damit auch bereits ein Beispiel kennen gelernt, das zeigt, dass S^∞-Äquivalenz (und damit auch S^ω- und S-Äquivalenz) nicht Verweigerungsäquivalenz implizieren. Umgekehrt gilt allerdings, dass V-Äquivalenz S-Äquivalenz impliziert.

Satz 3.8. *Es sei $(Z, A, \to)$ ein Transitionssystem und z, z' V-äquivalent. Dann sind z, z' auch S-äquivalent.*

Beweis. Wir zeigen dies per Widerspruchsbeweis und nehmen an $V(z) = V(z')$, aber $S(z) \neq S(z')$. O.B.d.A nehmen wir an es existiert ein Wort $w \in S(z) \setminus S(z')$, dann gilt $(w, \emptyset) \in V(z) \setminus V(z')$, da es einen Zustand gibt, der von z aus mit w erreicht wird. Ganz sicher ist es möglich, von dort aus gar keine Eingabe zu verweigern, es gilt also $(w, \emptyset) \in V(z)$. Allerdings gibt es keinen Zustand, der mit w von z' aus erreicht wird, also ist $V(z') \cap \{(w, A') \mid A' \subseteq A\} = \emptyset$ nach Definition von $V(z')$. Das ist aber ein Widerspruch zu der Annahme $V(z) = V(z')$. □

Verweigerungsäquivalenz impliziert allerdings nicht S^ω-Äquivalenz (und demnach auch nicht S^∞-Äquivalenz), wie man anhand einer leichten Modifikation eines vorherigen Beispiels erkennen kann:



Die Zustände z und z' sind verweigerungsäquivalent, da sie für jedes $n \geq 1$ nach Eingabe des Wortes a^n das Zeichen a verweigern können, aber nichts für das Wort ϵ verweigern können. Sie sind aber nicht S^ω-äquivalent, weil z ein unendliches Wort akzeptiert, z' aber nicht.

Aufgabe 2. *Wir setzen die Aufgabe 1 fort und vergleichen die gleichen Paare wie in Beispiel 1 auf Verweigerungsäquivalenz.*

1. *Sind die Zustände v_1 und w_1 verweigerungsäquivalent? Geben Sie bei nicht verweigerungsäquivalenten Paaren ein Paar (w, A') an, das ein Verweigerungspaar des einen aber nicht des anderen Zustandes ist.*

2. *Sind die Zustände w_1 und x_1 verweigerungsäquivalent? Geben Sie bei nicht verweigerungsäquivalenten Paaren ein Paar (w, A') an, das ein Verweigerungspaar des einen aber nicht des anderen Zustandes ist.*

3. *Sind die Zustände v_1 und z_1 verweigerungsäquivalent? Geben Sie bei nicht verweigerungsäquivalenten Paaren ein Paar (w, A') an, das ein Verweigerungspaar des einen aber nicht des anderen Zustandes ist.*

4. *Sind die Zustände v_1 und y_1 verweigerungsäquivalent? Geben Sie bei nicht verweigerungsäquivalenten Paaren ein Paar (w, A') an, das ein Verweigerungspaar des einen aber nicht des anderen Zustandes ist.*

Lösung für Aufgabe 2:

1. Nein, $(a, \{b\})$ kann von w_1 aus verweigert werden (man macht einen Schritt von w_1 nach w_3), aber nicht von v_1 aus.

2. Ja.

3. Nein. Das folgt bereits daraus, dass die Zustände nicht sprachäquivalent sind. Von Zustand z_1 aus kann das Paar $(ab, \{a\})$ verweigert werden, nicht jedoch von v_1 aus.

4. Ja.

Wir wollen nun eine Verhaltensäquivalenz kennen lernen, die alle bisher vorgestellten Verhaltensäquivalenzen subsummiert und damit ein vollständigeres Bild der Nutzererfahrung eines Transitionssystems ergibt: Bisimulation. Wir werden zudem sehen, dass Bisimulation sich auch algorithmisch als noch bedeutend angenehmer erweist als S-Sprachäquivalenz. Im Fall von nichtdeterministischen endlichen Automaten – und damit ganz analog auch bei Transitionssystemen – ist die S-Sprachäquivalenz zu entscheiden PSPACE-vollständig, wohingegen wir ein Polynomzeitverfahren kennen lernen werden, um zu entscheiden, ob zwei Zustände bisimular sind.

Definition 3.9 (Bisimulation)**.** *Es sei $(Z, A, \to)$ ein Transitionssystem und $R \subseteq Z \times Z$ eine Relation. Dann nennen wir R eine Bisimulationsrelation oder kurz eine Bisimulation, falls die folgenden Eigenschaften für alle $(z_1, z_2) \in R$ gelten:*

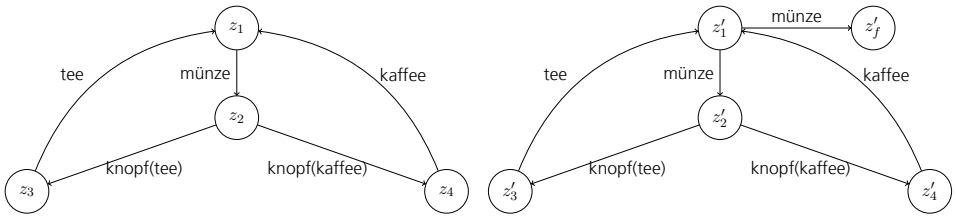
1. *Falls es ein $a \in A$ und ein $z_1' \in Z$ gibt, so dass $z_1 \xrightarrow{a} z_1'$, dann gibt es ein z_2' so dass $z_2 \xrightarrow{a} z_2'$ und $(z_1', z_2') \in R$*

2. *Falls es ein $a \in A$ und ein $z_2' \in Z$ gibt, so dass $z_2 \xrightarrow{a} z_2'$, dann gibt es ein z_1' so dass $z_1 \xrightarrow{a} z_1'$ und $(z_1', z_2') \in R$*

Wenn es eine Bisimulationsrelation R gibt, die zwei Zustände z_1, z_2 in Relation setzt, also $(z_1, z_2) \in R$, dann nennen wir z_1 und z_2 bisimular, in Zeichen $z_1 \sim z_2$. Die Relation $\sim$ nennen wir die Bisimularität.

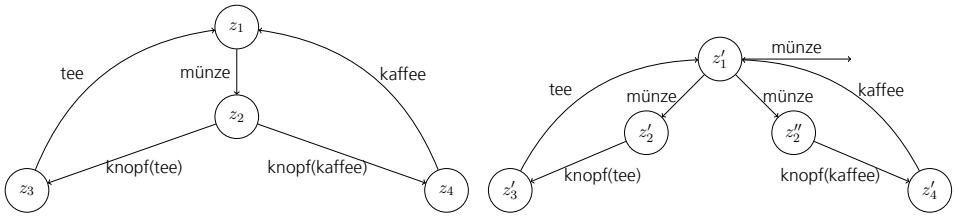
Hinweis: „Bisimulation“ heißt in der englischsprachigen Literatur ebenfalls „bisimulation“, allerdings verwendet man im Englischen statt des Begriffs „bisimular“ den Begriff „bisimilar“.

Alle Beispiele, die wir bisher betrachtet haben, die nicht sprachäquivalent oder verweigerungsäquivalent waren, sind ebenfalls nicht bisimular. Betrachten wir noch einmal unser Beispiel eines Heißgetränkeautomaten, so können wir feststellen, dass die Zustände z_1 und z_1' nicht bisimular sind:



Eine Bisimulationsrelation, die (z_1, z_1') enthält, muss wegen Bedingung 2 auch (z_2, z_f') enthalten, denn z_1' kann einen münze-Schritt zu z_f' ausführen und der einzige Zustand, der mit einer münze-Transition von z_1 aus erreichbar ist, ist der Zustand z_2. Allerdings kann (z_2, z_f') in keiner Bisimulationsrelation liegen, da z_2 (beispielsweise) eine knopf(tee)-Transition ermöglicht, z_f' aber nicht.

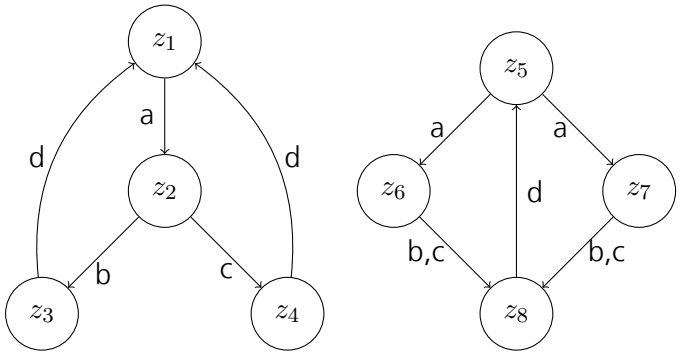
Das klassische Beispiel mit dem Bisimulation motiviert wird, ist aber das folgende:



Auch in diesem Fall sind die Zustände z_1 und z_1' nicht bisimular, denn damit eine Bisimulationsrelation z_1 und z_1' in Relation setzen kann, muss nach Bedingung 2 auch z_2 mit z_2' in Relation gesetzt werden, da von z_1' aus eine münze-Transition zu z_2' möglich ist und der einzige Zustand, der von z_1 aus mit einer münze-Transition erreicht werden kann z_2 ist. Nun führt Bedingung 1 zum Widerspruch, denn von z_2 aus ist es möglich, eine taste(kaffee)-Transition durchzuführen, es gibt aber keinen Zustand, der mit einer taste(kaffee)-Transition von z_2' aus erreichbar ist.

Dieses Beispiel zeigt eine weitere Hinsicht, in der Sprachäquivalenz nicht hinreichend ist, um die Nutzererfahrung eines Software-Systems abzubilden. Das linke Modell ermöglicht dem Nutzer nach Einwurf einer Münze die Wahl, die Kaffee-Taste zu drücken um einen Kaffe zu erhalten, oder die Tee-Taste zu drücken, um einen Tee zu erhalten. In dem rechten Modell hingegen wird die Wahl, ob das Nutzer die eine oder die andere Taste wird drücken können, bereits bei der Eingabe der Münze nichtdeterministisch getroffen. Je nach konkreter Umsetzung des Nichtdeterminismus bedeutet das, dass das Nutzer bereits zum Zeitpunkt des Einwurfs der Münze entscheiden muss, welches Getränk es bestellen möchte, oder gar, dass das Nutzer gar keinen Einfluss darauf hat, welches der beiden Getränke es erhält. In jedem Fall wird die Nutzererfahrung im rechten Modell signifikant anders ausfallen als im linken Modell, obwohl die Zustände z_1 und z_1' sprachäquivalent sind.

Wir betrachten nun ein nicht-triviales Beispiel für zwei Zustände, die bisimular sind:



Die Zustände z_1 und z_5 sind bisimular, wie man anhand der folgenden Bisimulationsrelation erkennen kann:

$$R = \{(z_1, z_5), (z_2, z_6), (z_2, z_7), (z_3, z_8), (z_4, z_8)\}$$

Zu Übungszwecken überprüfen wir diese Relation darauf, ob sie tatsächlich eine Bisimulationsrelation ist. Hierfür müssen wir die beiden Bedingungen, die wir an Bisimulationsrelationen gestellt haben, für jedes Paar in der Relation überprüfen.

(z_1, z_5) Von Zustand z_1 aus ist nur eine a-Transition möglich, mit der der Zustand z_2 erreicht wird. Von z_5 aus kann man beispielsweise den a-Nachfolger z_6 wählen und sieht, dass $(z_2,\ z_6) \in R$ gilt. Umgekehrt kann von z_5 aus nur eine a-Transaktion durchgeführt werden, es können allerdings zwei mögliche Nachfolgezustände gefunden werden: z_6 und z_7. Von z_1 aus können wir in beiden Fällen nur z_2 auswählen, das ist allerdings kein Problem, denn sowohl $(z_2, z_6) \in R$, als auch $(z_2, z_7) \in R$.

(z_2, z_6) Von z_2 aus ist eine b-Transition möglich, die zu z_3 führt. Von z_6 aus gibt es ebenfalls nur eine b-Transaktion, die zu z_8 führt und $(z_3, z_8) \in R$. Weiterhin ist von z_2 aus eine c-Transaktion möglich, die zu z_4 führt und wiederum ist z_8 der einzige c-Nachfolger von z_6. Es gilt $(z_4, z_8) \in R$. Umgekehrt führt eine b-Transaktion von z_6 aus zu z_8 und von z_2 aus gibt es nur eine mögliche b-Transaktion, die zu z_3 führt. Es gilt $(z_3, z_8) \in R$. Schließlich führt die c-Transaktion von z_6 aus zu z_8, wohingegen von z_2 aus nur eine c-Transaktion möglich ist, die zu z_4 führt. Es gilt $(z_4, z_8) \in R$ wie gefordert.

(z_2, z_7) Analog zum vorherigen Fall.

(z_3, z_8) Von Zustand z_3 aus gibt es nur eine Transition: Eine d-Transition zu z_1. Von z_8 aus gibt es ebenfalls eine d-Transition, die zu z_5 führt und $(z_1, z_5) \in R$. Umgekehrt gibt es von z_8 aus nur eine d-Transition. Diese führt zu z_5. Von z_3 aus gibt es ebenfalls eine d-Transition, die zu z_1 führt und $(z_1, z_5) \in R$.

(z_4, z_8) Analog zum vorherigen Fall.

Wir haben also für alle Paare in der Relation die Bedingungen an eine Bisimulation überprüft und sind daher sicher, dass R tatsächlich eine Bisimulation ist.

Aufgabe 3. *Wir setzen die Aufgabe 1 fort und vergleichen die gleichen Paare wie in Aufgabe 1 auf Bisimularität.*

1. *Sind die Zustände v_1 und w_1 bisimular? Geben Sie bei bisimularen Paaren eine Bisimulationsrelation an, die (v_1, w_1) enthält. Anderenfalls beweisen Sie, dass die beiden Zustände nicht bisimular sein können.*

2. *Sind die Zustände w_1 und x_1 bisimular? Geben Sie bei bisimularen Paaren eine Bisimulationsrelation an, die (w_1, wx_1) enthält. Anderenfalls beweisen Sie, dass die beiden Zustände nicht bisimular sein können.*

3. *Sind die Zustände v_1 und z_1 bisimular? Geben Sie bei bisimularen Paaren eine Bisimulationsrelation an, die (v_1, z_1) enthält. Anderenfalls beweisen Sie, dass die beiden Zustände nicht bisimular sein können.*

4. *Sind die Zustände v_1 und y_1 bisimular? Geben Sie bei bisimularen Paaren eine Bisimulationsrelation an, die (v_1, y_1) enthält. Anderenfalls beweisen Sie, dass die beiden Zustände nicht bisimular sein können.*

Lösung für Aufgabe 3:

1. Nein. Angenommen R sei eine Bisimulationsrelation, die (v_1, w_1) enthält. Dann gilt: Da $v_1 \xrightarrow{a} v_2$ muss es einen Zustand w geben, so dass $w_1 \xrightarrow{a} w$ und $(v_2, w) \in R$. Wir können hierzu aus zwei Zuständen wählen: $w = w_2$ oder $w = w_3$. Falls $w = w_2$, dann kann aber kein Antwortschritt auf den Schritt $v_2 \xrightarrow{c} v_1$ gegeben werden. Umgekehrt, wenn $w = w_3$, dann kann kein Antwortschritt auf den Schritt $v_2 \xrightarrow{b} v_1$ gegeben werden. Da die beiden Zustände nicht verweigerungsäquivalent sind und Bisimulation Verweigerungsäquivalenz impliziert, wäre dies als alternative Argumentation ebenfalls hinreichend.

2. Nein. Angenommen R sei eine Bisimulationsrelation, die (w_1, x_1) enthält. Dann gilt: Da $x_1 \xrightarrow{a} x_4$ muss es einen Zustand w geben, so dass $w_1 \xrightarrow{a} w$ und $(w, x_4) \in R$. Wir können hierzu aus zwei Zuständen wählen: $w = w_2$ oder $w = w_3$. Falls $w = w_2$, dann kann aber kein Antwortschritt auf den Schritt $x_4 \xrightarrow{c} x_1$ gegeben werden. Umgekehrt, wenn $w = w_3$, dann kann kein Antwortschritt auf den Schritt $x_4 \xrightarrow{b} x_1$ gegeben werden.

3. Nein, angenommen $(v_1, z_1) \in R$, dann muss wegen $v_1 \xrightarrow{a} v_2$ auch $(v_2, z_2) \in R$ (da z_2 der einzige mögliche Antwort-Zustand ist). Dann wiederum muss, beispielsweise wegen $v_2 \xrightarrow{b} v_1$, auch $(v_1, z_3) \in R$ gelten (wiederum ist z_3 der Zustand der beim einzigen möglichen Antwortschritt erreicht wird). Nun gibt es aber eine Transition $v_1 \xrightarrow{a} v_2$, auf die von z_3 aus nicht geantwortet werden kann. Da die beiden Zustände nicht sprachäquivalent sind und Bisimulation Sprachäquivalenz impliziert, wäre dies als alternative Argumentation ebenfalls hinreichend.

4. Ja. Das kann man anhand der Bisimulationsrelation

$$R = \{(v_1, y_1), (v_2, y_2), (v_1, y_3)\}$$

einsehen.

Kurseinheit 5: Refactoring

So wie sich in der Software bestimmte Muster ständig wiederholen, so gibt es auch im Prozess der Programmierung selbst bestimmte stereotype Tätigkeiten, die immer wieder und dabei mit nur leichten Variationen ausgeführt werden müssen. Dies betrifft insbesondere die Änderung von Code, bei der selbst so einfache Dinge wie das Umbenennen einer Klasse eine Masse von manuellen Änderungen nach sich ziehen, die eigentlich vollkommen schematisch und damit automatisch erfolgen können sollten.

Selbsttestaufgabe 5.1

Was muss alles geändert werden, wenn sich der Bezeichner einer Klasse ändert?

Refaktorisieren und Softwarefäulnis

Der zum Teil erhebliche Aufwand, den selbst solch kleine Änderungen nach sich ziehen, führt allzu häufig dazu, dass nur die unvermeidlichen, weil die Funktionalität betreffenden Änderungen im Code durchgeführt werden, während die Modifikationen, die lediglich dem Erhalt von Struktur, Wartbarkeit und Lesbarkeit des Codes dienen würden, ausbleiben. Der Code verkommt damit in dem Maße, in dem er geändert wird; man spricht hier auch von *Softwarefäulnis*. Das sog. **Refaktorisieren** wirkt der Softwarefäulnis entgegen.

> **Refaktorisierung: eine Änderung der internen Softwarestruktur mit dem Ziel, sie verständlicher und einfacher änderbar zu machen, ohne das beobachtbare Verhalten zu ändern**
>
> übersetzt aus [29]

zum Begriff der Refaktorisierung

Da die mit der Refaktorisierung verbundenen Änderungen auf ein Umbauen des Programms abzielen, ohne dessen Bedeutung zu verändern, heißen sie englisch Refactorings. Dabei ist das Wort „Refactoring“ eine Neuschöpfung, die durch die Analogie erklärbar ist, dass ein Programm ein aus einer Menge von Faktoren bestehendes Produkt, eben faktorisiert ist. Diese Faktorisierung offenbart eine innere Struktur,

Link

die ohne sie (die Faktorisierung) nicht so leicht sichtbar wäre. Eine Änderung der Faktorisierung könnte man demnach Refaktorisierung nennen.[35]

Mit der systematischen Befassung und der Katalogisierung möglicher Refaktorisierungen hat der Begriff der Refaktorisierung eine Mehrdeutigkeit erfahren: Er bezeichnet nicht nur den Vorgang oder das Ergebnis einer entsprechenden Tätigkeit (an sich schon eine Mehrdeutigkeit), sondern auch das Muster, nach dem die Tätigkeit erfolgt bzw. dem das Ergebnis gleicht. Während man im Deutschen noch eine sprachliche Unterscheidung treffen kann (das „Refaktorisieren" für die Tätigkeit vs. „die Refaktorisierung" für das Ergebnis oder das Muster), gelingt dies im Englischen nicht: Dort heißt es in allen Fällen schlicht „refactoring". Dazu kommt, dass mit Refaktorisierung bzw. Refactoring in zunehmendem Maße auch die Werkzeuge, die eine solche automatisch durchführen, benannt werden. Diese sollten aber zur besseren Unterscheidung Refaktorisierungswerkzeuge bzw. Refactoring tools genannt werden.

5.1 Einordnung

Allgemein dient das Refaktorisieren der Verbesserung des Designs einer Software, *nachdem* diese geschrieben wurde. Dieses Verhalten ist (war) eigentlich verpönt (das Design hat schließlich nach gängigen Vorstellungen von einem geregelten Softwareentwicklungsprozess vor der Implementierung zu erfolgen), die Notwendigkeit ergibt sich aber faktisch aus der Praxis — sie ist schlichtweg Realität. Inzwischen ist man dazu übergegangen, dies anzuerkennen; nicht zuletzt führen die Änderungen der Anforderungen während der Entwicklungsphase (sog. *Requirements drift*, als Daumenregel ca. 1% der Anforderungen pro Monat) dazu, dass man sich mit dem Thema auseinandersetzen muss. Die Methode des *Extreme Programming* (Abschnitt 7.1) verzichtet sogar vollständig auf ein A-priori-Design und setzt vollständig auf das Refaktorisieren.

Refactorings wurden zuerst in der SMALLTALK-Programmierung eingesetzt. Die erste größere wissenschaftliche Abhandlung ist die Dissertation von Opdyke [28]. Das erste bekannte Werkzeug war der sog. *Refactoring-Browser* für SMALLTALK. Da Refactoring die Semantik von Programmiersprachen berührt, ist es — unter theoretischen Gesichtspunkten — eines der härteren Probleme des Software Engineering. So ist insbesondere der Nachweis, dass eine Refaktorisierung das Verhalten des refaktorisierten Programms nicht ändert, zumindest für den allgemeinen Fall immens schwer.

5.1.1 Katalogisierung

Ähnlich wie die Entwurfsmuster aus Kurseinheit 4 lassen sich Refaktorisierungen standardisieren, indem man Schemen in Form strukturierter Beschreibungen vorgibt und diese mit

[35] Bei der im Deutschen ebenfalls üblichen Übersetzung „Refaktorierung" handelt es sich um eine sinnberaubende Verstümmelung, die hier bewusst nicht verwendet wird.

Namen versieht. Diese Namen werden dann zu Synonymen häufig recht komplexer Tätigkeiten und damit Bestandteil des Vokabulars von Programmierern .

Die Beschreibung eines Refactorings geht in der Regel von einem Design aus, das (aus welchem Grund auch immer) verbesserungswürdig erscheint. Es stellt diesem Design ein alternatives gegenüber, das das Ziel des Refactorings darstellt. Die notwendige Transformation des Quellcodes vom alten in das neue Design erfolgt dann mittels einer Reihe von kleinen Schritten, die in der Summe die erwünschte Änderung bewirken.

Die wohl auch heute noch umfassendste Katalogisierung von Refactorings ist die von Martin Fowler in seinem gleichnamigen Buch [29]. Eine Erweiterung dieses Katalogs wird von ihm selbst gepflegt; eine Neuauflage des Buchs mit den entsprechenden Aktualisierungen ist jedoch nicht geplant. Das Thema ist dennoch für die Programmierpraxis unverändert wichtig: Während von der Weiterentwicklung von Programmiersprachen in den letzten Jahren nur noch Produktivitätssteigerungen ausgingen, können zuverlässige Programmierwerkzeuge die Arbeit erheblich vereinfachen, indem sie stereotype Änderungen wie eben das Refaktorisieren von Code automatisieren. Damit diese Werkzeuge jedoch auch eingesetzt werden, ist es notwendig, dass sie über die Grenzen der sie jeweils anbietenden Entwicklungsumgebungen hinaus bekannt sind und in den Softwareentwicklungsprozess auch gedanklich einbezogen werden (so wie das bei den Entwurfsmustern aus Kurseinheit 4 längst der Fall ist).

große Refactorings

Die katalogisierte Beschreibung von Refactorings erfasst fast ausschließlich kleine Refaktorisierungen, also solche, die sich als eine überschaubare Menge von Einzelschritten durchführen lassen und deren Erfolg sich leicht überprüfen lässt. Für die Praxis genauso relevant ist aber die Definition und Umsetzung von sog. **großen Refactorings**, also Refactorings, die nicht nur einzelne Programmelemente betreffen, sondern größere Zusammenhänge bis hin zur gesamten Architektur eines Systems. Von einer formalen Vorgehensbeschreibung (oder gar von einer Werkzeugunterstützung) ist man hier aber noch weit entfernt; stattdessen verlegt man sich darauf, große Refactorings als eine Aneinanderreihung kleiner Refactorings zu betrachten. Dagegen spricht jedoch die hohe Fehlerquote, die entsteht, wenn eine große Transformation jedes Mal wieder neu in viele kleine zerlegt werden soll, die dann jeweils einzeln durchgeführt werden müssen. Das wird insbesondere dann schwierig, wenn dabei Zwischenprodukte entstehen, die das große Ziel aus den Augen verlieren lassen.

5.1.2 Refaktorisierungen als Algorithmen

Wenn man sich die Beschreibungen von Refaktorisierungen anschaut, fällt auf, dass sie einem (verbal spezifizierten) Algorithmus gleichen: Ausgehend von einem vorgefundenen Design, der Eingabe, wird durch eine Reihe von Schritten ein Zieldesign, die Ausgabe, erzeugt. Dabei kann das Refactoring während seiner Durchführung weitere Eingaben (von dem Benutzer) erwarten oder bereits vor seinem Start mit Parametern versorgt werden. Das besondere an Refactorings ist lediglich, dass es sich bei Ein- und Ausgabe um Programme handelt.

Ein Refaktorisierungswerkzeug ist damit ein Programm, das Programme verarbeitet, also ein *Metaprogramm* (s. Kurseinheit 6).

Vor- und Nachbedingungen von Refactorings

Refaktorisierungen können damit genau wie andere Programme spezifiziert werden. Insbesondere hat jedes Refactoring eine Menge von *Vorbedingungen*, die erfüllt sein müssen, damit das Refactoring anwendbar ist, und eine Menge von *Nachbedingungen*, die erfüllt sein müssen, wenn die Refaktorisierung abgeschlossen ist. Vor- und Nachbedingungen sind insbesondere auch dann interessant, wenn man mehrere Refactorings hintereinander ausführen will, wobei jedes folgende Refactoring eine oder mehrere Nachbedingungen des Vorgängers als Vorbedingungen zur Voraussetzung hat. Allerdings ist der Beweis der Korrektheit von Refactorings, genau wie der Beweis der Korrektheit von Algorithmen, alles andere als einfach.

automatisches Testen der Korrektheit

Eine besondere Nachbedingung jedes Refactorings ist definitionsgemäß, dass es die Bedeutung des Programms nicht ändert. Neben der trivialen Bedingung, dass sich das Programm nach dem Refactoring weiter problemlos übersetzen lassen muss (zumindest wenn dies zuvor der Fall war), gehört auch dazu, dass verfügbare Unit-Tests äquivalente Ergebnisse liefern, in der Regel also weiter erfolgreich durchlaufen, oder aber, dass die verifizierten Eigenschaften des Ursprungsdesigns auch im Produkt weiterhin vorliegen. Unabhängig von der Verfügbarkeit von Unit-Tests eröffnet die Anwendung von Refactorings zusätzlich die Möglichkeit des sog. *Back-to-back-Testens*: Man kann einfach die Ausgaben des Programms vor und nach der Refaktorisierung gegeneinander vergleichen. Voraussetzung hierfür ist allerdings, dass man über ein Testframework verfügt, das solche Tests (inkl. der automatischen Generierung der dafür notwendigen Eingaben) automatisch durchführt. Der Grundsatz des Testens, dass man damit nur Fehler finden kann und keine Fehlerfreiheit nachweisen, behält aber auch hier Gültigkeit.

5.1.3 Refactoring to patterns

Zweck eines Refactorings ist es, ein vorgefundenes Design in ein angestrebtes zu überführen. Angestrebtes objektorientiertes Design ist Ihnen ja schon begegnet, und zwar in Form von *Entwurfsmustern* (Kurseinheit 4). Was läge also näher, als spezielle Refactorings vorzusehen, die die Verwendung eines Entwurfsmusters zum Ziel haben?

Einer katalogartigen Fassung solcher Refaktorisierungen steht vor allem im Wege, dass die Ausgangslage, also ein Design, das kein Entwurfsmuster verwendet, nahezu beliebig ist. Wenn aber die Form der Eingabe unbekannt ist, ist es schwer, ein allgemeines Verfahren anzugeben, wie sie in die Ausgabe überführt werden kann. Stattdessen kann man nur von der gewünschten Ausgabe (die ja feststeht) rückwärts ausgehend angeben, welche der katalogisierten Refactorings dorthin führen können, und die Auswahl der für die jeweils benötigten Transformationen geeigneten Refactorings dem Entwickler überlassen. Die dabei auftretenden Probleme entsprechen doch im Wesentlichen denen der *großen Refactorings*: Das Refactoring muss jedes mal neu als eine Folge von kleineren Einzel-Refactorings geplant werden und damit hängt der Erfolg ganz wesentlich vom Geschick des Programmierers ab.

Etwas anderes ist es jedoch, wenn ein Entwurfsmuster (genauer: seine Anwendung) in ein anderes überführt werden soll: Hier sind sowohl Ausgangs- als auch Zielsituation bekannt. Ein Beispiel für ein solches Refactoring werden wir in Abschnitt 5.1.5 ausführlicher durchgehen. Im Allgemeinen dienen jedoch verschiedene Entwurfsmuster verschiedenen Zwecken und zwei Entwurfsmuster, die auf dieselbe Problemstellung angewendet dasselbe Programmverhalten bewirken, sind relativ selten, so dass man auch eher selten von einem Entwurfsmuster in ein anderes refaktorisieren wird.

5.1.4 Werkzeugunterstützung

Die oben erwähnte algorithmische Fassung von Refactorings legt nahe, dass sich Refactorings automatisieren lassen. Allerdings ist es bei fast allen Refactorings (wie schon beim eingangs erwähnten einfachen Beispiel des Umbenennens einer Klasse) nicht mit einem globalen, textuellen Suchen und Ersetzen getan — praktisch immer erfordern die Änderungen eine Interpretation des Programmtextes. Sie benötigen in der Regel den *abstrakten Syntaxbaum* (engl. abstract syntax tree, AST) des Programms.

Nun erlauben moderne IDEs mit ihren APIs in der Regel den Zugriff auf den abstrakten Syntaxbaum eines Programms, so dass die erforderlichen Programmanalysen und -transformationen von einem Refaktorisierungswerkzeug, das auf diesen APIs aufsetzt, direkt auf den benötigten Datenstrukturen durchgeführt werden können. Allerdings sind derartige APIs zum einen nicht sonderlich stabil, zum anderen unterliegen auch die Programmiersprachen, auf deren ASTs die Refactorings ausgeführt werden sollen, ständigen Änderungen, die nicht nur eine Anpassung von Editor und Compiler, sondern auch der Refactorings erfordern. So führte beispielsweise der Übergang von JAVA 2 zu JAVA 5 (mit seinen generischen Typen) dazu, dass viele Refactorings nicht mehr funktionieren und deshalb komplett überarbeitet werden müssten (was aber aufgrund des erheblichen Aufwands nicht immer passiert). Dazu kommt, dass gleiche Refactorings selbst für syntaktisch stark ähnliche Programmiersprachen (wie etwa JAVA und C#) heute nicht einfach übernommen werden können, da die Implementierungen zu sehr von den konkreten ASTs der jeweiligen Sprache abhängen. Höhere Abstraktionsstufen, die eine allgemeine Formulierung von Refactorings erlauben, sind jedoch noch nicht gefunden (oder zumindest noch nicht etabliert) — eine allgemeine, einheitliche Refactoring-Schnittstelle von IDEs ist (noch) nicht in Sicht.

So kommt es, dass sich heutige IDEs auch darin stark unterscheiden, ob und welche Refactorings sie anbieten. Alleinstehenden Refaktorisierungswerkzeugen scheint, eben aufgrund ihrer mangelnden Integration, in der Praxis keine besondere Bedeutung mehr zuzukommen, selbst wenn ihnen noch am ehesten die Lösung der obengenannten technischen Probleme zuzutrauen wäre.

Zum Schluss noch eine Bemerkung zur Qualität: Trotz des beträchtlichen Potentials zur Automatisierung des Tests von Refaktorisierungswerkzeugen (s. o.) weisen die heute verfügbaren Implementierungen zum Teil erhebliche Mängel auf. So führen selbst einfache Refaktorisierungen in der Praxis schon zu Syntax- oder Typfehlern, so dass das wichtigste Merkmal

heutiger Refaktorisierungswerkzeuge ist, ob die Undo-Funktion zuverlässig funktioniert. Daraus lassen sich zwei Dinge ableiten: Die korrekte Fassung eines Refactorings, einschließlich seiner Vorbedingungen, für alle möglichen Verwendungen ist viel komplexer, als es die oft informellen Beschreibungen glauben machen wollen, und die Verfügbarkeit von zuverlässigen, automatisierten Refaktorisierungen ist weit weniger entscheidend für den Erfolg einer IDE, als man vielleicht erwarten würde. Tatsächlich ergeben Umfragen unter Entwicklerinnen und die Rückmeldungen aus Fehlerdatenbanken, dass verfügbare Refaktorisierungswerkzeuge längst nicht im erwarteten Umfang genutzt werden.

5.1.5 Ein Beispiel

Nach den eher theoretischen Betrachtungen der vorigen Abschnitte wollen wir uns nun etwas ausführlicher mit einem konkreten Beispiel befassen, und zwar einem, das Ihnen aus Abschnitt 4.2 bereits bekannt ist: Vererbung durch Delegation ersetzen. Das dazugehörige Refactoring nennt sich denn auch genau so, auf englisch **Replace Inheritance with Delegation**.

In [29] wird dieses Refactoring kurz und knapp wie folgt charakterisiert:

> *A subclass uses only part of a superclasses interface or does not want to inherit data.*
>
> *Create a field for the superclass, adjust methods to delegate to the superclass, and remove the subclassing.*

Es folgt eine etwas ausführlichere Motivation und Beschreibung der Schritte, die jedoch nicht über den Inhalt des obigen hinausgeht. Die zur Durchführung des Refactorings konkret notwendigen Schritte werden in einem Abschnitt „Mechanics“ wie folgt ausgeführt:

> - *Create a field in the subclass that refers to an instance of the superclass. Initialize this field to* `this`.
> - *Change each method defined in the subclass to use the delegate field. Compile and test after changing each method.*
> - *Remove the subclass declaration and replace the delegate assignment with an assignment to a new object.*
> - *For each superclass method used by a client, add a simple delegating method.*
> - *Compile and test.*

Vorbedingungen

Wir erkennen im ersten Satz die *Vorbedingung* (wenn auch nicht unbedingt als solche formuliert): Eine Klasse ist Subklasse einer anderen und erbt von ihr Dinge, die sie selbst nicht braucht. Der erste Teil der Vorbedingung ist ein hartes Ausschlusskriterium: Wenn eine Klasse keine Subklasse ist, lässt sich das Refactoring nicht anwenden. Der zweite Teil ist etwas weicher: Wenn sie keine Dinge erbt, die sie nicht braucht, ist das Refactoring zwar immer noch anwendbar, bringt aber nicht den beschriebenen Nutzen.

Schritte und Nachbedingungen

Der zweite Satz stellt eine Kombination von grob zusammengefassten Handlungsanweisungen und *Nachbedingung* dar: Nachdem man das Erforderliche getan hat, kann die Vererbungsbeziehung gelöst werden, die damit nicht mehr besteht. Dass das Programm weiter funktioniert wie bisher, ist im Begriff des Refactorings implizit (wenn sich die Bedeutung ändern würde, wäre es kein Refactoring) und braucht daher nicht extra erwähnt zu werden.

Dem aufmerksamen Leser von Kurseinheit 4 wird nicht entgangen sein, dass das Refactoring *Delegation* einzuführen vorgibt, aber nur *Forwarding* einführt: Für eine echte Delegation fehlt ja der Verweis (das Feld) von der ehemaligen Superklasse zurück zur Klasse. Das bedeutet, dass wenn es vorher offen rekursive Aufrufe in der Superklasse gab, dass dann die refaktorisierte Klasse nach dem Refactoring davon nicht mehr erreicht wird. Es folgt, dass entweder die Vorbedingungen oder die Handlungsanweisungen unvollständig sind.[36] Eine genauere Untersuchung des Refactorings ist also angebracht.

5.1.5.1 Vorbedingungen

Eine triviale Vorbedingung des Refactorings, die in [29] unerwähnt bleibt, ist die, dass die Superklasse nicht abstrakt sein darf, da sonst keine Instanz gebildet werden kann, an die „delegiert" (eigentlich: geforwardet) wird. Zudem müssen alle Konstruktoren der Superklasse, die benötigt werden, um eine brauchbare Instanz zu erhalten, an die delegiert werden kann, zugreifbar sein. Sie dürfen also in JAVA insbesondere nicht `private` oder `protected` deklariert sein. Welche Konstruktoren (neben dem Default-Konstruktor, der in JAVA bei Vererbung automatisch mit aufgerufen wird) man braucht, lässt sich aus den in den Konstruktoren der Klasse vorhandenen Super-Aufrufen ablesen.

Eine dritte, schon weniger offensichtliche Vorbedingung ist die, dass das Programm keine Zuweisungen von Instanzen der zu refaktorisierenden Klasse an Variablen vom Typ der Superklasse oder dessen Supertypen enthalten darf, denn mit der Elimination der Vererbungsbeziehung geht auch die Aufgabe der Subtypenbeziehung einher. Diese ist aber zumindest in Sprachen mit *nominalem Typsystem* (worunter fast alle heute gebräuchlichen Sprachen mit statischer Typprüfung fallen; vgl. Kurs 01814) Voraussetzung für die Zulässigkeit einer Zuweisung. Diese Vorbedingung lässt sich etwas abschwächen, wenn man bereit ist, Subtypenbeziehungen zu Supertypen der Superklasse durch neue Extends- bzw. Implements-Klauseln wiederherzustellen, wobei man sich bei ersteren natürlich wieder Vererbung einkauft, die man ja gerade eliminieren wollte. Wenn aber Zuweisungen an den Typ der Superklasse im Programm vorhanden sind, lässt sich das Refactoring einfach nicht anwenden.

Die vierte Vorbedingung hatten wir bereits erwähnt: Da es sich bei der durch das Refactoring eingeführten „Delegation" lediglich um schnödes Forwarding handelt, dürfen Instanzen der zu refaktorisierenden Klasse nicht von ihrer Superklasse aus offen rekursiv, also über

[36] Wie schon im Kontext von *Design by contract* (in Kurseinheit 2) erwähnt, kann man ein fehlerhaftes Programm dadurch korrigieren, dass man seine Vorbedingungen verschärft. Wenn die Vorbedingungen nicht eingehalten werden, braucht das Programm auch nichts (Sinnvolles) zu tun.

this, aufgerufen werden. Eine Quelle solch offen rekursiver Aufrufe wurde dabei jedoch schon ausgeschlossen: Da die Zuweisungskompatibilität der Klasse mit ihrer ehemaligen Superklasse aufgehoben wird, können Instanzen der Klasse die ihrer Superklasse nicht mehr ersetzen. Das Vorkommen entsprechender Zuweisungen im Programm schließt die dritte Vorbedingung bereits aus. Es bleiben jedoch geforwardete Aufrufe der Klasse selbst an die ehemalige Superklasse, die nun, wie in Abbildung 4.1 zu sehen, nicht mehr die Methoden der Klasse aufrufen können, weil this nun eine Instanz der ehemaligen Superklasse bezeichnet. Solche Aufrufe müssen also ebenfalls ausgeschlossen werden, wenn keine Änderung des Programmverhaltens provoziert werden soll.

Eine fünfte Vorbedingung ergibt sich aus dem Umstand, dass in JAVA nur Methodenaufrufe weitergeleitet werden können: Greifen Klientinnen der Klasse auf von der Superklasse geerbte Felder zu, so würden diese Zugriffe nach dem Refactoring ins Leere gehen. Das ließe sich allerdings vermeiden, wenn Feldzugriffe konsequent über Zugriffsmethoden (Accessoren) erfolgen würden, die dann weitergeleitet werden könnten. In EIFFEL ist das übrigens immer der Fall, da dort Feldzugriffe transparent, also für die Zugreifer nicht sichtbar, immer über Accessoren erfolgen; in C# lässt es sich über sog. *Properties* ebenfalls einrichten, ohne die Klienten anfassen zu müssen. Man beachte, dass es nicht ausreicht, die betreffenden Felder in der Klasse einfach zu wiederholen (neu zu deklarieren): Die ehemals geerbten Methoden, an die jetzt weitergeleitet wird, hätten nur Zugriff auf ihre eigenen Versionen dieser Felder und eine Synchronisation der Inhalte ist aus technischen Gründen nicht möglich.

Eine sechste Vorbedingung ergibt sich aus den Sichtbarkeitsregeln der jeweils verwendeten Programmiersprache: Wenn beispielsweise in JAVA eine Methode mit dem Access modifier protected geerbt wird und vererbende und erbende Klasse nicht im selben Paket sind, dann ist diese Methode nach dem Refactoring nicht mehr zugreifbar und kann auch nicht per Forwarding aufgerufen werden.

Eine siebte Vorbedingung ergibt sich daraus, dass bestimmte Subtypbeziehungen aufrecht erhalten werden müssen, selbst wenn keine sie verlangenden Zuweisungen in einem Programm vorhanden sind: So dürfen beispielsweise keine entsprechenden expliziten Typtests (in JAVA per instanceof oder getClass()) vorkommen und die Ableitung von sog. Unchecked exceptions wie Error und RuntimeException darf nicht durch eine von Throwable ersetzt werden, damit der Compiler keine fehlenden Exception handler bzw. Throws-Klauseln moniert.

Eine letzte Vorbedingung schließlich ist so unoffensichtlich, dass vermutlich nur die allerwenigsten durch bloßes Überlegen auf sie kommen. Wenn auf den Instanzen einer Subklasse synchronisierte Methoden aufgerufen werden, die teilweise von einer Superklasse geerbt werden, und sich diese Aufrufe nach dem Refactoring auf zwei verschiedene Instanzen verteilen, klappt die Synchronisation u. U. nicht mehr (da jetzt zwei anstelle eines Monitors herangezogen werden).

Wie man sieht, sind die Voraussetzungen für die Anwendung selbst eines scheinbar so simplen Refactorings wie des hier beschriebenen alles andere als banal. Dabei ist man noch gut bedient, wenn der Compiler die Verletzung einer Vorbedingung entdeckt, weil sich nämlich

das trotzdem refaktorisierte Programm nicht mehr übersetzen lässt; weit schlimmer ist es, wenn die Verletzung — oder gar die Existenz! — einer Vorbedingung unentdeckt bleibt. Da aber der formale Beweis, dass die für ein Refactoring genannten Vorbedingungen vollständig sind und ausreichen, eine korrekte, bedeutungserhaltende Refaktorisierung durchzuführen, alles andere als auf der Straße liegt, ist man bei allen Refaktorisierungen, und seien sie noch so gut dokumentiert, darauf angewiesen, ihren Erfolg per Testen oder Verifikation zu überprüfen. Nicht gerade schmeichelhaft für eine Disziplin, die der Softwareentwicklung eine bedeutende Produktivitätssteigerung bescheren will.

5.1.5.2 Durchführung

Sind die Vorbedingungen erfüllt, kann das Refactoring durchgeführt werden. Doch auch dabei ergibt sich ein nichttriviales Problem: Es ist nämlich gar nicht automatisch klar, welche geerbten Felder und Methoden einer Klasse nicht gebraucht werden. Der Wunsch allein, bestimmte Elemente loszuwerden, reicht dazu nicht aus — es muss auch sichergestellt werden, dass diese Elemente nicht von Klienten oder Subklassen der Klasse benötigt werden.

Nun kann man sich auf den Standpunkt stellen, der Compiler wird einem schon sagen, welche Elemente erhalten bleiben müssen — Zugriffe auf nicht mehr vorhandene Programmelemente führen schließlich zu einem Übersetzungsfehler. Aber dieser Ansatz ist der des Trial and error: Man entfernt einfach die Elemente, die man gern loswerden würde, und wenn darunter eines zu viel war, bekommt man dies rückgemeldet. Für die Praxis ist das jedoch wenig befriedigend: Zum einen werden in den typischen Anwendungsfällen dieses Refactorings sehr viele Elemente geerbt (50 und mehr bei Ableitung von Frameworkklassen wie denen des AWT oder SWING) und zum anderen kann sich dabei herausstellen, dass das Refactoring insgesamt keine gute Idee war (nicht den gewünschten Effekt hat), man es also gern komplett wieder rückgängig machen möchte. Dann aber ist schon viel geändert und die Wiederherstellung des Ausgangszustands entsprechend aufwendig. Nicht zuletzt ist es, wenn kein entsprechendes Refaktorisierungswerkzeug vorhanden ist, kaum zumutbar, erst alle Methodenweiterleitungen einzuführen, um diese dann einzeln wieder zu entfernen. Macht man es aber andersherum, beginnt man also ohne eine Methodenweiterleitung und führt diese wie vom Compiler gefordert ein, kann es sein, dass man zunächst so viele Fehlermeldungen bekommt, dass der Überblick verlorengeht.

Was man stattdessen für eine zielgerichtete Anwendung des Refactorings bräuchte, ist eine Programmanalyse, die vorab feststellt, welche Elemente von der zu refaktorisierenden Klasse verlangt werden. Eine solche Analyse wird im Prinzip vom Compiler durchgeführt, während er die Referenzen auflöst — ein entsprechendes Refaktorisierungswerkzeug kann sich diese zunutze machen, wenn es Zugriff auf den AST des Programms hat. Tatsächlich verfügt die Implementierung des REPLACE INHERITANCE WITH DELEGATION Refactorings in INTELLIJ IDEA über eine solche Analyse; leider ist sie nicht vollständig, so dass das Programm nach der Durchführung des Refactorings so manches Mal nicht mehr kompiliert. Das zeigt einmal mehr, wie wenig entwickelt das Thema Refaktorisierungswerkzeuge heute noch ist.

5.1.5.3 Nachbedingungen

Nach einer erfolgreichen Durchführung des Refactorings sollten neben der allgemeinen Nachbedingung für Refactorings, dass das Programm hinterher immer noch dasselbe tut, auch die speziellen Ziele erreicht sein. Das heißt konkret, dass die refaktorisierte Klasse nicht mehr von ihrer ehemaligen Superklasse erbt, sondern stattdessen ein Feld besitzt, mittels dessen ihre Instanzen auf jeweils eine Instanz der ehemaligen Superklasse verweisen, an die sie weiterdelegieren können. Dieses Feld muss bei jeder möglichen Form der Instanziierung der refaktorisierten Klasse automatisch mit einer Instanz der ehemaligen Superklasse versorgt werden. Weiterhin enthält die Klasse für mindestens all die Methoden, die ehemals geerbt wurden und die vom Programm gebraucht werden, entsprechende Weiterleitungsmethoden an die ehemalige Superklasse. Konstruktoren der Superklasse, die zuvor per `super` aufgerufen wurden, werden jetzt zur Erzeugung der Instanz, an die delegiert wird, aufgerufen. Zuletzt ist die Klasse weiterhin Subklasse von Superklassen ihrer ehemaligen Superklasse, nämlich dann, wenn Zuweisungen im Programm eine entsprechende Zuweisungskompatibilität verlangen, und sie implementiert alle Interfaces, die das Programm (wiederum per Existenz entsprechender Zuweisungen) verlangt. Man beachte, dass durch die verlangten Interfaceimplementierungen u. U. auch Methoden in das *Klasseninterface* aufgenommen und weitergeleitet werden müssen, die von keinem Klienten jemals aufgerufen werden; in diesem Fall ist jedoch das entsprechende Interface zu hinterfragen (und ggf. zu refaktorisieren).

Eine allgemeine, in den Kontext anderer Refactorings eingebundene Beschreibung von REPLACE INHERITANCE WITH DELEGATION finden Sie zusammen mit einem Anwendungsbeispiel in Abschnitt 5.2.4.8.

5.2 Eine Auswahl von Refactorings

Leider gibt es für längst nicht alle bekannten Refactorings heute schon Implementierungen, die den Ablauf automatisieren. Wie das vorangegangene ausführliche Beispiel klargemacht haben sollte, liegt das nicht am mangelnden Interesse der Programmierer an diesen Refactorings — vielmehr zeigt erst der Versuch, ein Refactoring in einem Werkzeug umzusetzen, auf, was alles berücksichtigt werden muss und worin die eigentlichen Schwierigkeiten dabei liegen. Die meiste Änderungsarbeit muss daher immer noch von Hand gemacht werden. Dennoch haben auch Refactorings ohne Implementierung einen Wert für sich: Sie geben nämlich — in strukturierter Form — praktische Beispiele dafür, wie schlechter Code aussieht, wie guter Code aussieht und wie man vom einen zum anderen gelangt. In diesem Sinne trifft die nachfolgende Vorstellung einzelner Refactorings auch eine Auswahl auf Basis dessen, was eine „gute" objektorientierte Idiomatik (also eine Art, sich in einem Programm objektorientiert auszudrücken) darstellt.

Ähnlich wie Entwurfsmuster lassen sich auch Refactorings nach ihrem Inhalt klassifizieren. Im Folgenden werden Beispiele aus den folgenden Kategorien besprochen:

- Bedingungen vereinfachen
- Lesbarkeit verbessern
- Daten organisieren
- Generalisierung einsetzen
- Methoden organisieren

Die Darstellung orientiert sich dabei an [29].

5.2.1 Bedingungen vereinfachen

Die Kontrolllogik eines Programms verursacht häufig einen großen Teil seiner Komplexität. Dabei muss nach Fred Brooks zwischen der natürlichen oder **essentiellen Komplexität** (engl. essential complexity), die in der Natur des Problems begründet ist, und der **künstlichen Komplexität** (engl. accidental complexity), die durch eine nichtideale Umsetzung entsteht, unterschieden werden. Letztere kann reduziert werden, indem man die Bedingungen, die die Kontrolllogik eines Programms ausmachen, vereinfacht.

5.2.1.1 Verschachtelte Bedingungen durch Wächter ersetzen (Replace Nested Conditional with Guard Clauses)

Wer hat das Problem noch nicht selbst erlebt: Eine zunächst einfache Fallunterscheidung muss immer mehr Spezialfälle berücksichtigen, so dass immer neue Else-if-Zweige hinzugefügt und bestehende Bedingungen mit Und-, Oder, und/oder Nicht-Ausdrücken verfeinert werden müssen. Am Ende steht man vor einem unüberschaubaren Wust aus Zweigen, der sich trotz bester Einrückungspraxis nicht mehr nachvollziehen lässt und in dem Fehler zu beheben wie eine Sisyphusarbeit anmutet: Kaum passt die eine Bedingung, stimmt es an einer anderen Stelle nicht mehr. Das folgende Beispiel mag in dieser Hinsicht als harmlos angesehen werden:

```
744 double getPayAmount() {
745     double result;
746     if (_isDead) result = deadAmount();
747     else {
748         if (_isSeparated) result = separatedAmount();
749         else {
750             if (_isRetired) result = retiredAmount();
751             else result = normalPayAmount();
752         };
753     }
754     return result;
755 };
```

Eine einfache, aber recht effektive Art, dieses Problem zu lösen, ist, die Else- und Else-if-Teile aufzulösen und stattdessen nur noch Ifs zu verwenden. Deren Bedingungen müssen natürlich komplexer sein, da sie die beim Else implizite Negation des vorangegangenen Ifs wiederholen müssen. Auf der anderen Seite ist so für jeden Zweig die Bedingung, die für

dessen Ausführung erfüllt sein muss, unmittelbar dem Zweig zugeordnet (und muss nicht umständlich und fehleranfällig aus dem gesamten Verzweigungskomplex zusammengesucht werden). Da die Bedingung die Anweisungen gewissermaßen (vor der Ausführung) schützt, spricht man auch von einem Guard bzw. von *Guarded commands* (so genannt von Dijkstra). Untereinander hingeschrieben entsprechen die Guarded commands den Einträgen in einer Wahrheitstabelle, wobei im Ergebnisteil natürlich beliebige Anweisungen (Commands; anstelle von Wahrheitswerten) eingetragen sein können und all die Zeilen, in denen keine Anweisung steht, weggelassen werden. Eine solche Wahrheitstabelle (durch Einfügung von Don't cares weiter vereinfacht) ist die nachfolgende:

_isDead	_isSeparated	_isRetired	Anweisungen
true	don't care	don't care	result = deadAmount()
false	true	don't care	result = separatedAmount()
false	false	true	result = retiredAmount()
false	false	false	result = normalPayAmount()

Diese Tabelle lässt sich in folgenden Code übersetzen:

```
if (_isDead) result = deadAmount();
if (! _isDead && _isSeparated) result = separatedAmount();
if (! _isDead && !_isSeparated && _isRetired) result =                retiredAmount();
if (! _isDead && !_isSeparated && !_isRetired) result =
        normalPayAmount();
```

Im gegebenen Beispiel lässt sich zusätzlich ausnutzen, dass die Ausführung einer Methode jederzeit durch Rückgabe (return) abgebrochen werden kann. Zwar lässt sich diskutieren, ob ein Return inmitten einer Methode (oder an mehreren Stellen einer Methode) noch der strukturierten Programmierung entspricht (nach der jedes Konstrukt genau einen Eingang und einen Ausgang haben sollte), man wird aber wohl zustimmen, dass die Lesbarkeit hier keinen Schaden nimmt.

```
double getPayAmount() {
    if (_isDead) return deadAmount();
    if (_isSeparated) return separatedAmount();
    if (_isRetired) return retiredAmount();
    return normalPayAmount();
};
```

5.2.1.2 Bedingung zerlegen (DECOMPOSE CONDITIONAL)

Manchmal ist schon *eine* Bedingung so kompliziert, dass die Lesbarkeit darunter leidet. Wenn dann noch dazukommt, dass die bedingten Aktionen nicht selbsterklärend sind, kann dieses Refactoring helfen. Aus

```
if (date.before (SUMMER_START) || date.after(SUMMER_END))
    charge = quantity * _winterRate + _winterServiceCharge;
else
    charge = quantity * _summerRate;
```

wird dann

```
770 if (notSummer(date))
771     charge = winterCharge(quantity);
772 else
773     charge = summerCharge (quantity);

774 private boolean notSummer(Date date) {
775     return date.before(SUMMER_START) || date.after(SUMMER_END);
776 }

777 private double summerCharge(int quantity) {
778     return quantity * summerRate;
779 }

780 private double winterCharge(int quantity) {
781     return quantity * _winterRate + _winterServiceCharge;
782 }
```

Die Namen der eingefügten Methoden haben den Charakter ausführbarer Kommentare. Das Refactoring löst damit auf einfache Weise das Problem, dass man bei eingefügten Kommentaren oft nicht so genau weiß, auf welchen Teil des Quellcodes sie sich beziehen, ein Problem, das man nur durch längliche Formulierungen oder Wiederholung von Teilen des Quellcodes im Kommentar lösen kann.

Man könnte argumentieren, dass die bessere Lesbarkeit teuer, nämlich um den Preis der verlangsamten Ausführung erkauft wird. Ein geschickter Compiler wird jedoch versuchen, die Methodenaufrufe zu inlinen (also an Ort und Stelle — in der Zeile — einzufügen), so dass keine Verschlechterung des Laufzeitverhaltens entsteht. Außerdem hilft die durchgeführte Fragmentierung, doppelten Code — der von manchen als die Wurzel allen Übels angesehen wird — zu vermeiden: Wenn an anderer Stelle dieselbe Bedingung oder Aktion noch einmal benötigt werden sollte, kann man direkt darauf zurückgreifen. Ist das jedoch nicht der Fall, wird die Klasse allerdings mit Methoden überfrachtet, deren Nutzen außerhalb des Kontextes der Bedingung nicht ersichtlich ist, was die Lesbarkeit in gewisser Weise verschlechtert. Eine Lösung könnte hier die Möglichkeit der Definition von lokalen Methoden (Methoden, die innerhalb von Methoden deklariert sind) bieten; diese besteht in JAVA aber leider nicht.

Dieses Refactoring ist übrigens eine konkrete Anwendung des Methode-extrahieren-Refactorings, das in Abschnitt 5.2.5.1 behandelt wird.

5.2.1.3 Bedingung durch Polymorphismus ersetzen (REPLACE CONDITIONAL WITH POLYMORPHISM)

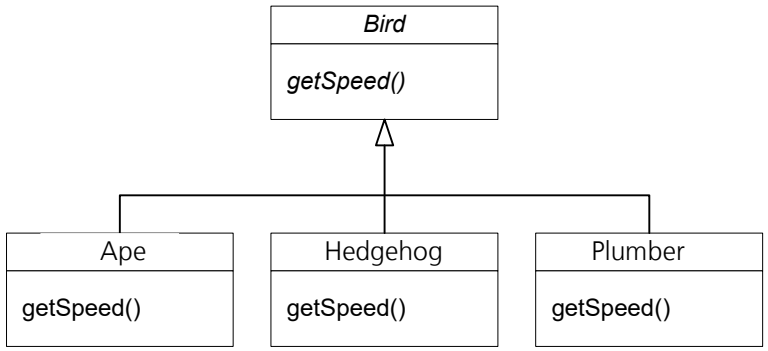
Nicht selten hängt eine Fallunterscheidung am Typ eines Objekts. Im Folgenden (nicht ganz ernstgemeinten) Beispiel ist das der Fall:

```
783 double getSpeed() {
784   switch (_type) {
785     case APE:
786       return getBaseSpeed();
787     case HEDGEHOG:
```

```
788     return getSpeedOfSound();
789   case PLUMBER:
790     return (runButtonPressed()) ? 2*getBaseSpeed() : getBaseSpeed();
791   }
792   throw new RuntimeException ("Should be unreachable");
793 }
```

Die Variable _type soll hier offensichtlich nur einen von drei verschiedenen Werten annehmen können: APE, HEDGEHOG oder PLUMBER. In Abhängigkeit vom jeweiligen Wert, der eine Tierart repräsentiert, wird der Wert für eine Variable speed berechnet und zurückgegeben. Da es sich bei der Variable _type um ein Attribut (Feld) des Objektes handelt, das die Methode getSpeed() implementiert (dessen Speed also berechnet werden soll), ist davon auszugehen, dass es sich bei dem Objekt um ein Tier handelt, das eben von einem der drei genannten Arten (Typen) sein kann.

In der objektorientierten Programmierung würde man genau diesen Umstand, nämlich dass es drei verschiedene Arten von Spielfiguren geben soll, dadurch ausdrücken, dass man eine abstrakte Klasse GameCharacter vorsieht und von ihr drei konkrete Klassen, Ape, Hedgehog und Plumber, ableitet, wie im nachfolgenden UML-Diagramm dargestellt. Die abstrakte Methode getSpeed(), die in der gemeinsamen Superklasse deklariert ist, wird dann in den drei Unterklassen konkret überschrieben. Die explizite Verzweigung der Switch-Anweisung wird damit durch die implizite Verzweigung des *dynamischen Bindens* (der technischen Umsetzung der Polymorphie) ersetzt.



Refaktorisiert sieht die obige Methode dann so aus:

```
794 abstract class GameCharacter {
795     abstract double getSpeed();
796 }

797 class Ape extends GameCharacter {
798     double getSpeed() {
799         return getBaseSpeed();
800     }
801 }

802 class Hedgehog extends GameCharacter {
803     double getSpeed() {
804         return getSpeedOfSound();
805     }
```

```
806 }

807 class Plumber extends GameCharacter {
808     double getSpeed() {
809         return (runButtonPressed()) ? 2*getBaseSpeed() : getBaseSpeed();
810     }
811 }
```

Die refaktorisierte Version hat gleich mehrere Vorteile:

1. Mit der Anzahl der zu unterscheidenden Fälle (Typen) steigt die Länge der Switch-Anweisungen. Einzelne Methoden können dadurch sehr lang werden, was in der objektorientierten Programmierung einigermaßen verpönt ist (vgl. Kurs 01814). Bei der polymorphismusbasierten Lösung wird hingegen jeder Fall einzeln (in einer anderen Klasse) abgehandelt und die Methoden bleiben entsprechend kurz.

2. Häufig bleibt es nicht bei *einer* Fallunterscheidung nach dem Typ: Switch-Anweisungen der obigen Art, die große Redundanzen enthalten, durchziehen dann den Code. Dies ist insbesondere dann ein Problem, wenn sich die Zahl der zu unterscheidenden Arten (Typen) irgendwann einmal ändert: Dann müssen alle Switch-Anweisungen nachgepflegt werden — wehe dem, das eine vergisst! Bei der polymorphismusbasierten Lösung hingegen wird beim Einfügen eines neuen Typs (repräsentiert durch eine neue konkrete Klasse) vom Compiler erzwungen, dass alle in der Superklasse abstrakt deklarierten Methoden auch implementiert werden und die Fallunterscheidung damit stets vollständig getroffen wird.
3. Nicht zuletzt ergibt sich aus dem Polymorphismusansatz eine bessere Erweiterbarkeit: Während bei einer Erweiterung des Aufzählungstyps und entsprechend der Switch-Anweisungen bereits bestehender Quellcode (auf den man unter Umständen gar keinen oder nur eingeschränkten Zugriff hat) geändert werden muss, kann man im anderen Fall einfach eine neue Klasse hinzufügen — im günstigsten Fall muss an bereits bestehendem Code überhaupt nichts geändert werden.

Selbsttestaufgabe 5.2

Unter welchen Voraussetzungen muss am Code nichts geändert werden?

Dem gegenüber steht aber auch ein durchaus gravierender Nachteil:

- Da die einzelnen Fälle und damit die Fallunterscheidung als Ganzes auf mehrere Klassen verteilt ist, ist es schwierig, sich einen Überblick über alle Alternativen (die ja gegebenenfalls noch nicht einmal vollständig bekannt sind; siehe Punkt 3 oben) zu verschaffen: Man muss schon alle Klassen nebeneinanderlegen, um zu sehen, worin sich die verschiedenen Fälle unterscheiden. Damit einhergehend (und nicht selten als Hauptnachteil der objektorientierten Programmierung bezeichnet) ist der Umstand, dass das Programm bei der schrittweisen Ausführung (beim Debuggen)

wild hin und her springt und man schnell den Überblick darüber verliert, wo man gerade ist (und wie man dort hingekommen ist; vgl. Kurs 01814).

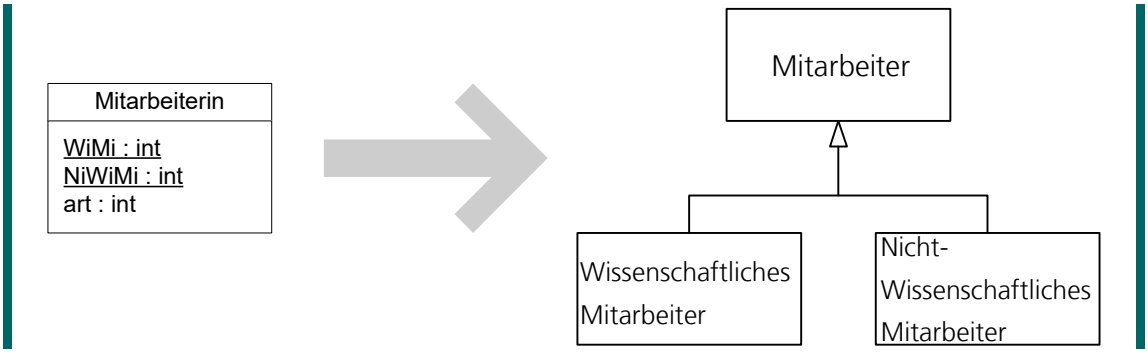
Zusammenfassung

Es ergibt sich also, dass die Fallunterscheidungen jeweils anders gruppiert werden: Bei einer (expliziten) Verzweigung stehen alle Fälle an einer Stelle, aber mehrere Fallunterscheidungen mit den gleichen Alternativen über den Code verstreut; bei der Polymorphie stehen die Fälle an verschiedenen Stellen (Klassen), aber die sonst verstreuten Fallunterscheidungen stehen (nach Alternativen geordnet) zusammen (in Klassen, die die Alternativen repräsentieren).

konsequente Anwendung in SMALLTALK

Das Prinzip, den Polymorphismus an die Stelle der Bedingung bzw. der bedingten Verzweigung zu setzen, wurde übrigens in der Programmiersprache SMALLTALK auf die Spitze getrieben: Da dort alles, also auch die Wahrheitswerte `true` und `false`, ein Objekt ist, kann man auf die If-Anweisung ganz verzichten, einfach indem man eine abstrakte Klasse `Boolean` mit zwei konkreten Subklassen `True` und `False` vorsieht, die jeweils verschiedene Implementierungen für die (in `Boolean` abstrakt deklarierten) Methoden `ifTrue` und `ifFalse` vorsehen (wobei der vom Ergebnis abhängig auszuführende Code als Parameter an die jeweilige Methode übergeben werden muss).

Anmerkung: Stark mit diesem Refactoring verwandt ist „Typcode durch Subtypen ersetzen“ (REPLACE TYPE CODE WITH SUBCLASSES):



5.2.1.4 Einführung eines Nullobjekts (INTRODUCE NULL OBJECT)

Gewissermaßen ein Spezialfall des vorgenannten Refactorings stellt die Einführung eines Nullobjekts dar. Häufig ist es zulässig (entspricht es der Realität), dass ein Objekt als Attributwert anstelle eines anderen eben auch kein Objekt haben kann. So kann es beispielsweise sein, dass jemand nicht verheiratet ist (und damit eben kein Ehepartner hat), dass ein Kunde kein Konto hat etc. In solchen Fällen muss vor einem Zugriff auf das Attributobjekt geprüft werden, ob es überhaupt vorhanden ist oder ob stattdessen ein Nullwert vorliegt. Der Nullwert wäre dann entsprechend anders zu behandeln.

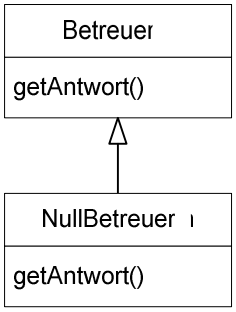
Das nachfolgende Codefragment zeigt ein Beispiel aus einem gedachten Kursbetreuungssystem:

```
812 Betreuer betreuer = kurs.betreuer;
```

```
813 if (betreuer == null)
814     antwort = "leider keines da, das eine Antwort wüsste";
815 else
816     antwort = betreuer.getAntwort();
```

Wesentlich an dem Beispiel ist, dass für den Fall des Vorliegens eines Nullwertes eine wirkliche Alternative vorgesehen ist und nicht nur ein für den Normalfall gedachter Codeblock übersprungen wird.

Diese Fallunterscheidung lässt sich nun eliminieren (korrekter: durch Polymorphismus ersetzen), indem man eine spezielle Klasse NullBetreuerin einführt, die angibt, was zu tun ist, wenn zu einem Kurs keine Betreuerin (die Nullbetreuerin) vorliegt. Das dazugehörige Klassendiagramm kann so aussehen:



Anmerkung: Es könnte aber auch eine abstrakte Klasse Betreuer mit zwei oder mehr konkreten Subklassen, darunter EchtesBetreuer und NullBetreuer vorgesehen werden. Da im gegebenen Fall die Klasse NullBetreuer ein Singleton ist (nur genau eine Instanz hat; s. Hinweis unten), ist dies vielleicht etwas übertrieben.

Das obige Codefragment würde damit auf das folgende zusammenschrumpfen:

```
817 Betreuer betreuer = kurs.betreuer;
818 antwort = betreuer.getAntwort();
```

Dafür wäre dann noch das folgende zu implementieren:

```
819 class NullBetreuer  extends Betreuer {
820     String getAntwort() {
821         return  "leider keines da, das eine Antwort wüsste";
822     }
823 }
```

Diese Klasse würde dann ggf. noch um zusätzliche Methoden zur Behandlung von weiteren Spezialfällen bei nicht vorhandenen Betreuern ergänzt. Im Gegensatz zu den möglichen Problemen des allgemeineren Refactorings oben (Bedingung durch Polymorphismus ersetzen) hat dies nicht den Nachteil, dass der Code weit verstreut wird: Es gibt nur den einen Sonderfall (den Nullwert) und die Behandlung dessen wird an einer Stelle konzentriert.

Hinweis

Da es im Allgemeinen nicht sinnvoll sein wird, mehrere Instanzen einer solchen Nullklasse zuzulassen, wird man diese in Form des *Singleton Pattern* realisieren

(siehe [31]). Da NullBetreuer zudem eng an Betreuer gebunden ist (und außerhalb des Kontexts von Betreuerin wohl kaum Verwendung finden wird), kann man sich überlegen, ob man ersteres nicht als innere Klasse implementieren will. Die äußere Klasse (im gegebenen Fall Betreuer) würde dann nur noch eine Konstante (das Nullobjekt) öffentlich machen.

5.2.1.5 Zusicherung einfügen (INTRODUCE ASSERTION)

Das *Design by contract* verlangt, dass jede Methode die Voraussetzungen für ihr Gelingen mittels Vorbedingung explizit macht. Das sog. *defensive Programmieren* verlangt dagegen, dass mögliche Zustände (Eingaben etc.), mit denen eine Methode nicht zurechtkommen würde, in dieser Methode geprüft werden und bei Vorliegen entsprechend (fehlertolerant) reagiert wird[37]. In der Praxis wird jedoch häufig weder das eine noch das andere gemacht.

Wann immer man findet, dass eine bestimmte, im Code implizite Bedingung Voraussetzung für die fehlerfreie Ausführung des Codes (einer Codesequenz) ist, sollte man diese Bedingung explizit machen, wenn schon nicht durch Vorbedingungen à la Design by contract oder Guards, dann durch das Einstreuen von allgemeinen *Zusicherungen* (Asserts). Der nachfolgende Code, der nur funktioniert, wenn ein Angestelltes eine Spesenbeschränkung oder das Projekt, dem es primär zugeordnet ist, eine solche hat, enthält eine solche implizite Bedingung (im Kommentar ausgedrückt):

```
824 class Employee {
825     private static final double NULL_EXPENSE = -1.0;
826     private double _expenseLimit = NULL_EXPENSE;
827     private Project _primaryProject;

828     double getExpenseLimit() {
829         // should have either expense limit or primary project
830         return (_expenseLimit != NULL_EXPENSE) ?
831             _expenseLimit :
832             _primaryProject.getMemberExpenseLimit();
833     }
834 }
```

Er wird zu

```
835     double getExpenseLimit() {
836         assert (_expenseLimit != NULL_EXPENSE
837                 || _primaryProject != null);
838         return (_expenseLimit != NULL_EXPENSE) ?
839             _expenseLimit :
840             _primaryProject.getMemberExpenseLimit();
841     }
```

[37] was jedoch nicht immer möglich ist; s. Abschnitt 2.4