Semesterarbeit „Drinks Dispense Machine“

Fallstudie der Vorlesung Software-Modellierung

WS08/09

vorgelegt von

Daniel Kuhn
Computer-Science and Media
dk047@hdm-stuttgart.de
Matrikelnummer: 20486

Inhaltsverzeichnis

Abbildungsverzeichnis

Abstract

Die vorliegende Semesterarbeit beschäftigt sich mit der Modellierung einer Drinks Dispense Machine, die nach der Aufgabenstellung vom 16.12.2008 bearbeitet und erstellt wurde. Das erste Kapitel diskutiert die allgemeinen Modelltheorie und setzt den Modellbegriff in Zusammenhang mit der Softwaremodellierung. Der zweite Teil beschreibt das Übersichtsmodell gemäß der gegebenen textuellen Beschreibung des Getränkeautomaten und zeigt damit dessen Struktur und Verhalten. Diese Struktur und dieses Verhalten wird danach mit der Z-Spezifikation des Automaten in einigen Punkten exemplarisch verglichen und Gemeinsamkeiten und Unterschiede zu dem im zweiten Teil vorgestellten UML-Modell beschrieben. Der vierten Teil zeigt den Einsatz von OCL-Constraints des UML-Modells und zeigt, wie mithilfe von OCL Constraints ein Z-Modell zu einem Z-Profil verallgemeinert werden kann. Das letzte Kapitel beschäftigt sich mit der Metamodellierung. Zudem wird in diesem Teil der Zusammenhang zwischen UML-Activities, BPMN, EPK und EEPK verdeutlicht.

1. Aufgabendefinition

Aufgabe der Semesterarbeit ist die Erstellung eines UML-Modells eines Getränkeautomaten aus der folgenden Aufgabe entnommen aus SHEPPARD, Deri: An Introduction to Formal Specification with Z and VDM. McGraw-Hill Publishing Co., 1995.:

„This final case study examines the specification of drinks dispensing machine, the control panel for which in [Abbildung 1]. Coins are inserted to (at least) the value of the drink and the drink chosen by pressing an appropriate combination of the selection buttons. The drink is then served by pressing button D. Unlike tea and coffee, chocolate is served only white and sweet (commercial chocolate powder contains whitener and sweetener). If for some reason the drink cannot be dispensed the customer can obtain a full refund by pressing button R. The machine accepts any coin that is legal tender in the United Kingdom an the correct change is issued (if necessary) once a drink has been dispensed.

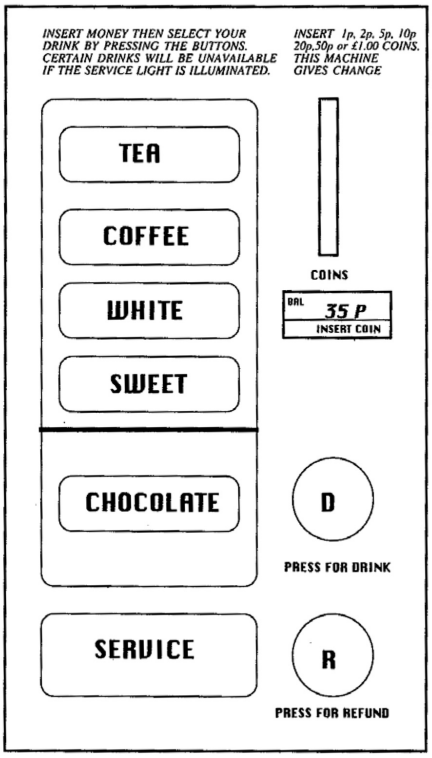


Abbildung 1: Drinks Dispense Machine Control Panel

The machine has a number of displays which are useful to the customer an to those who maintain it. The service light is illuminated if the machine needs to be serviced. Servicing will be required when ingredients for certain drinks are exhausted. However, the machine is fully functional for drinks that do not need those ingredients. The only occasion when the machine is effectively out of use (apart from initialization) is when the ingredients for all the drinks have been exhausted. The two other displays are the coin display and the report display. The coin display shows the current value (in pence) of the coins inserted into the machine. This is a simple LED display with the word BAL (for balance) and the letter P permanently part of the display. Immediately below this is he report display which is used to give instructions to the customers. On the real machine this would be designed in some eye-catching way.“

2. Allgemeinen Modelltheorie

Um das Übersichtsmodell und die Bestandteile gemäß der allgemeinen Modelltheorie von Martin Glinz einzuordnen, wird zuerst der allgemeine Begriff „Modell“ genauer beleuchtet. Dazu werden neben den Vorlesungsfolien von Martin Glinz [Glinz 2005] weitere Vorlesungsfolien von Prof. Dr. Uwe Kastens [Kastens 2001] herangezogen.

2.1. Lexikalischer Modellbegriff

Modell [italien., zu lat. modulus „Maß, Maßstab“], allg. Muster, Vorbild, Entwurf.
▷ Mensch (auch Tier), der (das) als Vorbild für künstler. Studien oder Kunstwerke dient („sitzt“).
▷ in der *Bildhauerei* meist in verkleinerter Form ausgeführter Entwurf einer Plastik oder Tonarbeit, die in Bronze gegossen werden soll. – ↑Architekturmodell.
▷ in der *Modebranche* Bez. für 1. ein nur einmal oder in eng begrenzter Anzahl hergestelltes Kleidungsstück *(M.kleid)*; 2. die Vorlage für eine Vervielfältigung; 3. svw. Mannequin.
▷ im *Sprachgebrauch verschiedener Wiss.* (Philosophie, Naturwiss., Soziologie, Psychologie, Wirtschaftswiss., Politikwiss., Kybernetik u. a.) ein Objekt materieller oder ideeller (Gedanken-M.) Natur, das von einem Subjekt auf der Grundlage einer Struktur-, Funktions- oder Verhaltensanalogie für ein anderes Objekt *(Original)* eingesetzt und genutzt wird, um Aufgaben zu lösen, deren Durchführung unmittelbar am Original selbst nicht möglich bzw. zu aufwendig ist (z. B. Flugzeug-M. im Windkanal). Die **Modellmethode** vollzieht sich in vier Schritten: 1. Auswahl (Herstellung) eines dem [geplanten] Original entsprechenden M.; 2. Bearbeitung des M., um neue Informationen über das M. zu gewinnen (**Modellversuch;** ↑Ähnlichkeitsgesetze); 3. Schluß auf Informationen über das Original (meist Analogieschluß); ggf. 4. Durchführung der Aufgabe am Original. Infolge der Relationen zw. Subjekt, Original und M. **(Modellsystem)** ist ein M. einsetzbar u. a. zur Gewinnung neuer Informationen über das Original (z. B. Atom-M.), zur Demonstration und Erklärung (z. B. Planetarium), zur Optimierung des Originals (z. B. Netzplan), zur Überprüfung einer Hypothese oder einer techn. Konstruktion (z. B. Laborversuch). – Abweichend von diesem M.begriff versteht die *mathemat. Logik* unter M. eine Interpretation eines Axiomensystems, bei der alle Axiome dieses Systems wahre Aussagen darstellen. Diese **Modelltheorie** liefert grundlegende Verfahren zur Behandlung von Fragen der Vollständigkeit, Widerspruchsfreiheit und Definierbarkeit.

Wissenschaften einschließlich Informatik

mathematische Logik

Abbildung 2: Modellbegriff im allgemeinen Lexikon [Kastens 2001, Folie 115a]

Modell (allgemeiner Begriff)

Teilgebiet: Modellierung
model (in general)

Während wir in den Formalwissenschaften wie Mathematik oder Physik einen präzisen Gebrauch des Wortes „Modell" (→ *Gegenstandsraum*) vorfinden, wird das Modell-Denken in den Sozialwissenschaften weitgehend durch einen vagen Gebrauch des Ausdrucks „Modell" gekennzeichnet. Folgende Begriffe, die sich in ihrer Intention oft stark unterscheiden, dürften die gebräuchlichsten Verwendungsweisen sein:

1. *Modell in der mathematischen Logik*
2. Modell als Bezeichnung für Theorien schlechthin
3. Modell als Resultat der Abbildung der Wirklichkeit.

Weitere Klassifizierungskriterien (→ *Klassifizierung*[2]) lassen sich nach dem Zweck, der mit den einzelnen Modellen verfolgt wird angeben (siehe Abb. S. 512).

Modell als Theorie schlechthin (2) findet sich häufig im verbalen Sprachgebrauch der Sozialwissenschaften. Insbesondere jene Teilklassen von Theorien, die mathematisiert, quantifiziert bzw. formalisiert sind, werden allgemein als Modell bezeichnet. Beispiele sind Preismodell, Rentenmodell.

Modelle als Abbild der Realität (3) stellen eine umfangreiche, sehr heterogene Klasse dar. Hierbei bilden die Beschreibungen ohne Verwendung einer Sprache, meist auf ein handliches Maß verkleinerten Nachbildungen eines vorgestellten Originals, die bekannteste Art von Modellen. Diese werden, wie z.B. der Globus, auch als ikonische oder materiale Modelle bezeichnet. *Stübel*

Teilgebiet: Logik
model

Es gibt zwei unterschiedliche Definitionen für Modelle der mathematischen *Logik*:

a) Eine Struktur Σ heißt Modell einer Formelmenge X, wenn jede Formel aus X in Σ gültig ist.
b) Das Paar (I,ζ), bestehend aus einer Interpretation I und einer *Belegung* ζ, heißt Modell einer Formelmenge X, wenn jede Formel aus X bei I und ζ wahr ist.

Für Mengen X von Aussagen, also Formeln ohne freie Variablen, sind beide Definitionen gleichwertig, da dann die Belegung keine Rolle spielt.

Die Modelltheorie beschäftigt sich mit gegenseitigen Beziehungen zwischen Aussagen formalisierter Theorien und mathematischen Strukturen, in denen die Aussagen gelten. *Müller; Stübel*

Modell, abstrakt symbolisches

Teilgebiet: Modellierung
abstract symbolic model

Eine vor allem in der Betriebswirtschaft sehr verbreitete Klasse von Modellen bilden die abstrakt symbolischen Abbilder eines Realitätskomplexes. Dabei kann es sich sowohl um rein verbale Reproduktionen eines Systems handeln als auch um ein künstliches Sprachsystem, das durch zunächst inhaltsleere symbolische Zeichen und syntaktische (→ *Syntax von Programmiersprachen*) Regeln gekennzeichnet ist. *Stübel*

Abbildung 3: Modellbegriff im Lexikon der Informatik [Kastens 2001, Folie 115b]

Die Abbildung 2 zeigt einen Auszug der Definition des Begriffs „Modell" aus „Meyers Neues Lexikon, in zehn Bänden, Meyers Lexikonverlag, 1993". Abbildung 3 zeigt die Definition des „Modell" aus dem „Lexikon der Informatik und Datenverarbeitung" (H-J. Schneider, 3. Aufl. Oldenbourg Verlag 1991). Beide Abbildungen wurden aus den Folien der Vorlesung „Modellierung" von Prof. Dr. Uwe Kastens der Universität Paderborn entnommen.

Anhand der Vorlesungsfolien von Prof. Dr. Uwe Kastens wurden im Diplomstudiengang „Digitale Medien“ an der Berufsakademie Mosbach die Vorlesung „Mathematik für Softwareentwickler“ von Prof. Dr. Arnulf Mester gelehrt.

Ein Modell ist also neben den für die Informatik unzutreffenden Bedeutungen, „ein Objekt materieller oder ideeller (Gedanken-M.) Natur, das von einem Subjekt auf der Grundlage einer Struktur-, Funktions- oder Verhaltensanalogie für ein anderes Objekt (Original) eingesetzt und genutzt wird, um Aufgaben zu lösen, deren Durchführung unmittelbar am Original selbst nicht möglich bzw. zu aufwendig ist […].“ [Kastens 2001, Folie 115a] Neben dem Modellbegriff beschreibt das Lexikon die Modellmethode in vier Schritten, wobei der Punkt 4 nicht direkt für die Informatik zutreffen ist, da das Original (Softwareprojekt) meist nur eine Abbildung oder gedankliche Vorstellung ist (materieller oder ideeller Natur):

1. Auswahl eines dem geplanten Modell entsprechenden Modells
2. Bearbeitung des Modells um neue Informationen über das Modell zu gewinnen
3. Schluss aus Informationen über das Original
4. (ggf. Durchführung der Aufgabe am Original)

Die Definition aus Abbildung 3 gibt für den Begriff „Modell“ drei gebräuchliche Verwendungsweisen an:

1. Modell in der mathematischen Logik
2. Modell als Bezeichnung für Theorien schlechthin
3. Modell als Resultat der Abbildung der Wirklichkeit

2.2. Allgemeiner Modellbegriff

Ein Modell kann ein nach [Glinz 2005, Folie 7] und [Kastens 2001, Folie 115] wie folgt eingeordnet werden:

- Abbild eines vorhandenen Originals (z.B. Schiffsmodell, Eisenbahnmodell)
- Vorbild für ein herzustellendes Original (Gebäude in kleinem Maßstab; Vorbild in der Kunst)
- konkretes oder abstraktes (gedankliches) Modell (Schiffsmodell, Rentenmodell)
- konkretes oder abstraktes Original (Schiff oder Bevölkerungsentwicklung)

Davon abweichende Bedeutungen des Modellbegriffs sind das Fotomodell, das Automodell (z.B. eine Typenreihe) und der Modellbegriff in der Logik (Modell des Axiomensystems: Eine Struktur S ist ein Modell der Formeln F, wenn für alle F für S gelten)

2.3. Bedeutung in der Informatik

In der Informatik versteht man den Begriff Modell allerdings als abstraktes Abbild oder Vorbild zu abstrakten oder konkreten Originalen. Allerdings ist hier die Ähnlichkeit, beziehungsweise die genaue Übereinstimmung zwischen Original und Modell, nicht immer das Ziel. Meist wird das Modell bewusst vom Original abstrahiert oder vom Original vereinfacht.

In der Softwareentwicklung versuchen Modelle meist eine Abbildung der Realität zu sein. Auf unser Beispiel bezogen ist die formale Beschreibung des Getränkeautomaten das Original. Das Modell zu diesem Original hat dabei die Anforderung dieses möglichst genau abzubilden.

Im Firmenumfeld bildet Software jedoch meist Geschäftsprozesse ab, um diese zu vereinfachen oder zu automatisieren. Ziel dabei ist es dem Nutzer die Arbeit zu erleichtern oder Arbeit abzunehmen und eventuell einen Zusatznutzen zu bieten (z.B. statistische Auswertungen, aktuelle Verkaufszahlen, ...). Im Falle des Getränkeautomaten ist der Geschäftsprozess das Verkaufen von Getränken.

Ein Modell ist bei Software in den meisten Fällen nötig, um entweder vorhandene Software zu dokumentieren (zwecks Erweiterbarkeit und/oder Optimierung) oder um eine noch nicht existierende Software zu beschreiben. Der Vorteil eines Modells bei Software ist die Möglichkeit zur Aufteilung der in Arbeitspakete bei der Entwicklung, die innerhalb von Teams und gegebenenfalls Teamübergreifend teilbar und handhabbar sind.

Mit Beschreibungssprachen wie UML ist es möglich Software und deren Geschäftsprozesse zu modellieren und darzustellen. Allerdings schreibt die UML dem Softwareentwickler bei der Modellierung nicht vor ob die Abbildung adäquat oder abstrakt modelliert werden soll. Je nach Anwendungszweck ist es demnach sinnvoll abstrakt oder detailliert zu modellieren. Beispielsweise sind für eine Komponenten-Ansicht die Detailprozesse unwichtig, man würde eher das Zusammenspiel zwischen den Komponenten betrachten. Eine Detailansicht kann dann für jeden Teilaspekt einer Komponente adäquat modelliert werden, die dann als Arbeitspaket gegebenenfalls zur Entwicklung freigegeben wird.

Softwareentwicklung wird abgesehen von den planbaren Eigenschaften aber auch fast immer dazu verwendet um eine gemeinsame Vorstellung zwischen dem Kunden und der Entwickler zu schaffen. Denn meist haben Kunden wenig Vorstellung von Softwareentwicklung oder im schlechtesten Falle keine Vorstellung davon was sie überhaupt von ihrer Software erwarten.

Der Softwaremodellierungsprozess hilft zum einen die Anforderungen an die Software klar zu dokumentieren, die Vorstellungen beziehungsweise die verschiedenen Sichten auf die Anforderungen zu vereinheitlichen und dem Kunden (Wissensträger) im besten Falle alle Informationen über seine Vorstellungen zu entlocken, bevor mit der Entwicklung begonnen wird. Allerdings ist Softwareentwicklung ein iterativer Prozess. In der Regel werden die Modelle, die zu Beginn des Projektes erstellt werden meist während der Entwicklung weiter angepasst und verfeinert. Denn gerade bei der aktuellen Wirtschaftslage werden viele Projekte gestrichen oder gekürzt. Für die Softwareentwicklung heisst dies dann meist, dass Anforderungen, die für die Kernfunktion nicht zwingen notwendig sind in einem späteren Folgeprojekt umgesetzt werden. Daher kann es passieren, dass geplante Modelle gekürzt oder neu angepasst werden müssen.

Zusammengefasst sind die Vorteile von Modellen in der Informatik beziehungsweise speziell in der Softwareentwicklung folgende:

- Dokumentation durch eine Abbildung von vorhandener Software zum Zweck der Optimierung oder Erweiterbarkeit
- Softwaremodellierung als Vorbild einer zu entwickelnden Software (unter Anderem aus Gründen der Planbarkeit, Realisierbarkeit und Aufteilung in Arbeitspakete)
- Softwaremodellierung, um die Komplexität eines Abbilds oder Vorbilds darzustellen und zu dokumentieren
- Softwaremodellierung um eine gemeinsame Basis zwischen dem Wissensträgers, dem Softwarearchitekten und des Entwicklers zu etablieren.
- Softwaremodellierung als iterativer Prozess um die Komplexität zu erfassen und sich ändernden oder neuen Anforderungen anzupassen

2.4. Bezug auf die Übungsaufgabe „Drinks Dispense Machine“

Im Falle der Fallstudie des Getränkeautomaten ist das zugrunde liegende Modell in erster Linie ein sprachliches Modell (siehe Kapitel 1), welches den Getränkeautomaten beschreibt. Das grafische Modell der Bedienoberfläche des Getränkeautomaten ist dabei eher zur Verdeutlichung des sprachlichen Modells zu sehen. Ein weiteres Modell, das unter Umständen in die endgültige Modellierung eingeht, ist die Vorstellung des Softwareentwicklers. Denn gerade bei einem Getränkeautomaten hat man neben dem vorliegenden sprachlichen Modell ein gedankliches Modell von Getränkeautomaten, die man bisher benutzt hat - Insbesondere deshalb, da das in der Aufgabe vorliegende sprachliche Modell einige Prozesse nicht genau und vollständig beschreibt, sodass Fragen offen bleiben. Bei einem echten Projekt würde man dies allerdings mit dem Kunden (Wissensträger) genauer besprechen.

3. Übersichtsmodell

In diesem Kapitel wird gemäß der Aufgabendefinition ein Übersichtsmodell in UML erstellt. Dazu werden zuerst die wichtigen Elemente der textuelle Beschreibung stichpunktartig zusammengefasst:

- Geldeinwurf (mit mindestens dem Gesamtwert des Getränks)
- Gültige Münzen sind alle Münzen in Großbritannien
- Auswahl des Getränks durch Drücken einer Kombination von Buttons auf der Bedienoberfläche
- Tee, Kaffee können mit Weißer und Zucker kombiniert werden (je eine Menge)
- Schokolade wird implizit mit Weißer und Zucker angeboten
- Wenn das Getränk nicht ausgeschenkt werden kann, bekommt der Kunde sein Rückgeld durch drücken des R-Buttons
- Wenn zu viel Geld eingeworfen und das Getränk ausgeschenkt wurde bekommt der Kunde den Differenzbetrag als Rückgeld
- Der Automat hat verschiedene Anzeigen
  - Service Licht: wird angeschaltet wenn der Automat Service benötigt
    - Zutaten für Getränke sind leer
  - Out of Order: wenn keine Zutaten für ein Getränk vorhanden ist
  - Münzwert-Display (zeigt den Wert der eingeworfenen Münzen an)
  - Anzeigen-Display (gibt dem Kunden Anweisungen und Hinweise)
- Getränke können so lange ausgeschenkt werden wie noch Zutaten vorhanden sind
- Buttons:
  - Kaffee, Tee, Schokolade
  - Weißer, Zucker
  - D-Button
  - R-Button

3.1. Strukturmodellierung: Klassendiagramm

„Ein Klassendiagramm beschreibt die Objekttypen im System und die verschiedenen Arten von statischen Beziehungen zwischen ihnen. Außerdem zeigen Klassendiagramme die Eigenschaften und Operationen einer Klasse und sagen aus, welche Einschränkungen für Objektbeziehungen gelten.“ [Fowler 2004]

Das Klassendiagramm in Abbildung 4 zeigt die statische Gesamtsicht auf den Getränkeautomaten. Um das Diagramm im folgenden besser beschreiben zu können wird es in drei Bereiche gegliedert:

- **Basisklassen** bestehend aus *Getränkeautomat*, *Bedienpanel*, *Buttons*, *Service-Lampe*, *Display* und *Geldeinwurf* (siehe Abbildung 5)
- Klassen die den **Bezahlprozess** abbilden bestehend aus *Muenzlager, Muenze* und *Britische_Muenzen* (siehe Abbildung 28)
- Klassen die den **Inhalt** und die **Bestellung** beschreiben bestehend aus *Inhaltslager, Inhaltslager_Getraenk, Inhaltslager_Zutat, Inhalt, Getraenk, Zutat, Getraenke_Typ* und *Zutaten_Typ* (siehe Abbildung 30)

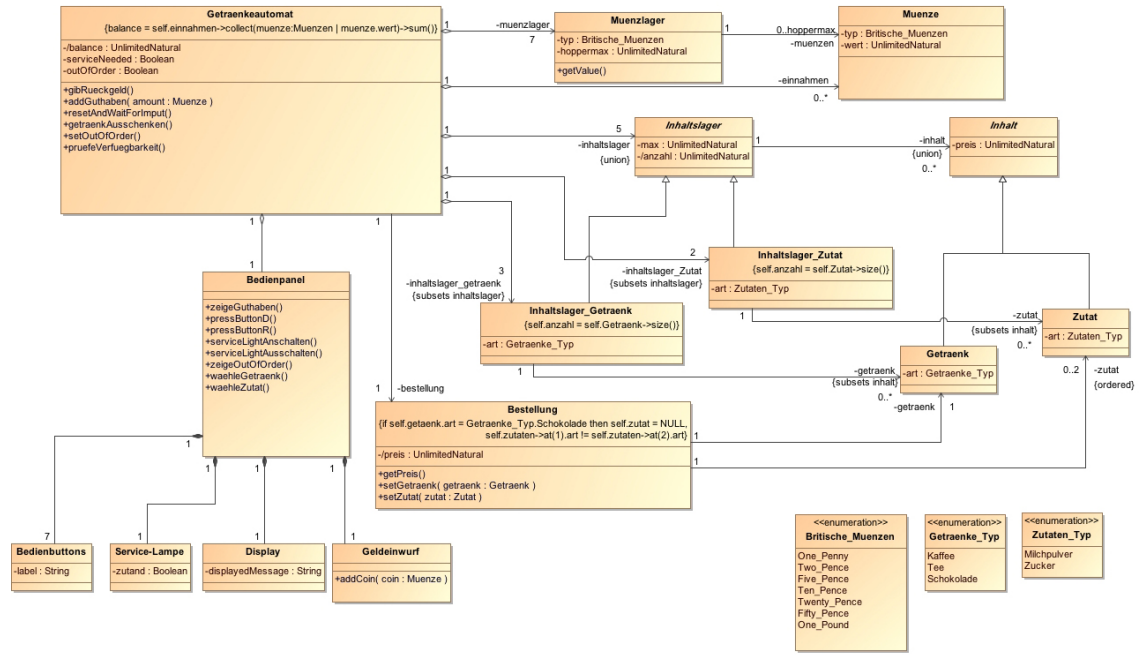


Abbildung 4: Klassendiagramm des Getränkeautomaten (Gesamt)

3.1.1. Basisklassen

Die *Getraenkeautomat* Klasse ist in diesem Modell die zentrale Klasse. Sie steht mit allen anderen Klassen direkt oder indirekt in Beziehung und besteht aus Klassen, die für den Bezahlprozess, den Inhalt und die Interaktion mit dem Kunden verantwortlich sind. Im diesem Unterkapitel werden die Klassen und Attribute, die den Bezahlprozess und den Inhalt des Automaten abbilden allerdings nicht näher ausgeführt. Diese beiden Teilaspekte werden in den folgenden Abschnitten näher beschrieben.

3.1.1.1. Klasse Getränkeautomat

Die Klasse Getränkeautomat enthält die Attribute:

- *balance (UnlimitedNatural)* ist ein abgeleitetes Attribut (derived attribute) und errechnet sich aus der Summe der Einnahmen. Der Typ *UnlimitedNatural* umfasst den Wertebereich 0 bis unendlich.
- *serviceNeeded (Boolean)* gibt an ob eine Zutat oder ein Getränk leer ist
- *outOfOrder (Boolean)* ist *true* wenn kein Getränk mehr ausgeschenkt werden kann
- *muenzlager (Muenzhopper)* beschreibt die Anzahl der vorhandenen Münzen im Münzlager (siehe 3.1.2.)
- *einnahmen (Muenze)* ist eine Sammlung der von Kunden eingeworfenen Münzen (siehe 3.1.2.)
- *inhaltslager (Inhaltslager)* gibt den Füllstand der einzelnen Inhalte an (siehe 3.1.3)
- *bestellung (Bestellung)* repräsentiert die Benutzereingaben nach dem Drücken des Button D

Methoden der Klasse Getränkeautomat:

- *gibRueckgeld()* weist den Automaten an die Münzen, die der Kunden eingeworfen hat zurückzugeben.
- *addGuthaben(amout:Muenze)* wird ausgelöst wenn ein Kunde Geld einwirft. Die Münze selbst wird zu den Einnahmen hinzugefügt

- *resetAndWaitForInput()* ist die Methode, die ausgelöst wird, wenn der Kunde den Reset Knopf drückt. Die Methode ruft dann die gibRueckgeld() Methode auf und bewirkt somit, dass das Attribut balance den Wert 0 erhält. Zudem wird der Zustand des Bedienpanels zurückgesetzt und die aktuelle Auswahl sowie die Bestellung verworfen.
- *getraenkAusschenken()* weist den Automaten an die Bestellung auszuschenken

3.1.1.2. Klasse Bedienpanel

Die Klasse *Bedienpanel* stellt die Methoden für das physische Interface bereit. Es besteht aus sieben Bedienbuttons, einer *Service Lampe*, einem *Display* und einem *Geldeinwurf*. Da die Klassen vom Bedienpanel existenzabhängig sind, sind sie als Komposition realisiert. Das Bedienpanel selbst steht in einer 1 : 1 Beziehung zum Getränkeautomat und ist Teil dieses.

Die Klasse *Bedienpanel* enthält selbst keine Attribute, sondern delegiert die Methodenaufrufe an den Getränkeautomat falls nötig weiter.

Die Methoden von Bedienpanel:

- *displayGuthaben()* zeigt das Guthaben auf dem Display an (wird aufgerufen von *addGuthaben(..), resetAndWaitforInput()*
- *pressXYZ()* sind die Methoden, die beim Drücken eines Buttons aufgerufen werden. („XYZ“ ist in dieser Beschreibung der Platzhalter für Button D, Button R, Tea, Coffee, White, Sweet, Chocolate)
- *serviceLightAnschalten/Ausschalten()* schaltet das Service Licht an oder ab
- *displayOutOfOrder()* zeigt „Out of Order“ im Display an (ist nicht in der texutellen Beschreibung spezifiziert)

3.1.1.3. Diskussion

Es gibt viele verschiedene Ansätze wie man die Klasse des Getränkeautomaten gestalten kann. Man könnte beispielsweise auf die Klasse des *Bedienpanels* verzichten und die Methoden direkt in den Getränkeautomaten übernehmen. Es ist allerdings übersichtlicher wenn alle Methoden, die eine Interaktion des Benutzers verarbeiten und Informationen darstellen in einer Art View-Klasse zusammengefasst werden. Auch fraglich ist ob man die Buttons, die Service-Lampe das Display oder de Geldeinwurf noch einmal separat aufführen sollte. Die Klassen wurden hier zur Vollständigkeit modelliert.

Alternativ zum Datentyp *UnlimitedNatural* kann man auch Integer wählen und diesen über ein OCL Contraint einschränken, sodass der Wert nicht < 0 werden darf. *Unlimited-Natural* hat diese Einschränkung schon implizit.

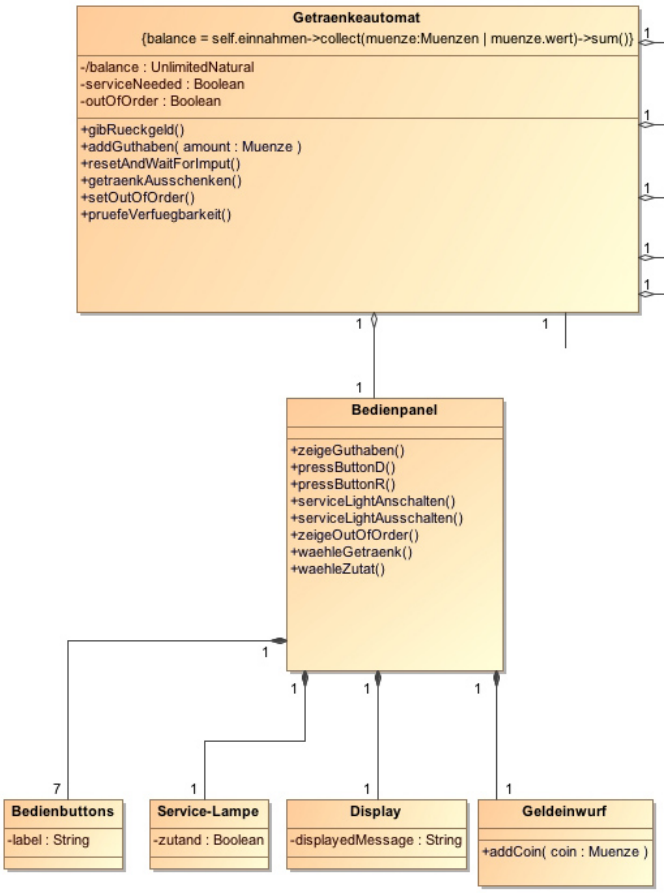


Abbildung 5: Klassendiagramm des Getränkeautomaten (Basisklassen)

3.1.2. Bezahlprozess

Der Vorgang des Bezahlprozesses, beziehungsweise die Speicherung der Münzen, der Münzwerte und der Einnahmen kann in zwei Klassen abgebildet werden.

3.1.2.1. Muenzlager

Die Klasse *Muenzlager* entspricht den physischen Münzcontainern eines Getränkeautomaten. Hier in dieser Fallstudie gibt es 7 Instanzen dieser Klasse, da es für jeden in Großbritannien vorhandenen Münztyp eine Muenzlager gibt.

Jedes Muenzlager hat daher zwei Attribute:

- Das Attribut *typ* (*Britische_Muenze*) gibt an um welchen Münztyp es sich handelt

– Das Attribut *muenzen* beinhaltet die Instanzen eines Typs von Münzen. Die Anzahl der Münzen, die das Münzlager beinhalten kann, wird hier über die Multiplizität „*0..hoppermax*“ angegeben, wobei *hoppermax* in diesem Fall eine statische Variable sein könnte.

3.1.2.2. Muenze

Die Klasse *Muenze* repräsentiert eine phyische Münze die in den Automat eingeworfen werden kann. Sie enthält den Typ der Münze (vom Typ *Britische_Muenzen*) und den Wert (in der kleinsten Währungseinheit: Pence) dieser.

Neben der Beziehung zu *Muenzlager* enthält die Klasse *Getränkeautomat* noch eine weitere Beziehung zu *Muenze*. Wie bereits in Abschnitt 3.1.1.1. beschrieben, werden die Münzen, die der Kunde einwirft, separat gesammelt und nicht direkt in die *Muenzlager* eingeordnet.

3.1.2.3. Britische_Muenzen

Gemäß der textuellen Beschreibung des Automaten sind nur Münzen aus Großbritannien gültig. Diese Werte der Münzen werden daher in dieser Enumeration gehalten. Eine Enumeration wurde auch deswegen gewählt, um eine gewisse Erweiterbarkeit des Automaten sicher zu stellen. Ändern sich die Münzen (falls Großbritannien sich entscheiden würde die Landeswährung auf Euro umzustellen) kann einfach die Enumeration ausgetauscht werden.

3.1.2.4. Diskussion

Die hier gezeigte Modellierung ist nur ein Weg, wie man die Münzhaltung des Getränkeautomaten hätte umsetzen können. Wie zu Beginn erwähnt macht die texutelle Beschreibung an manchen Stellen keine genauen Angabe über den Ablauf. Daher sind an dieser Stelle zum Einen die Vorstellung des Modellierers und zum Anderen die Umsetzung des Getränkeautomaten in Z eingeflossen.

Eine alternative Modellierung der Problematik könnte so aussehen, dass es der Getränkeautomat sieben fest definierte Muenzlager enthält, die nur textuell den Wert der beinhalteten Münzen angibt (zum Beispiel OnePennyLager:Collection, …). Des Weiteren wäre denkbar, dass die Einnahmen nicht in einer separaten Collection gehalten werden, sondern direkt in die Münzlager eingeordnet werden. Der Wert der eingewor-

fenen Münzen müsste dann in einem separaten Attribut im Getränkeautomat gespeichert werden. Zudem müsste die Geldrückgabe dann aus den Muenzlagern erfolgen, wodurch die Implementierung der Methode *gibRueckgeld()* komplexer werden würde da die entsprechende Wertigkeit von Münzen aus den Münzlagern herausgesucht werden müsste.

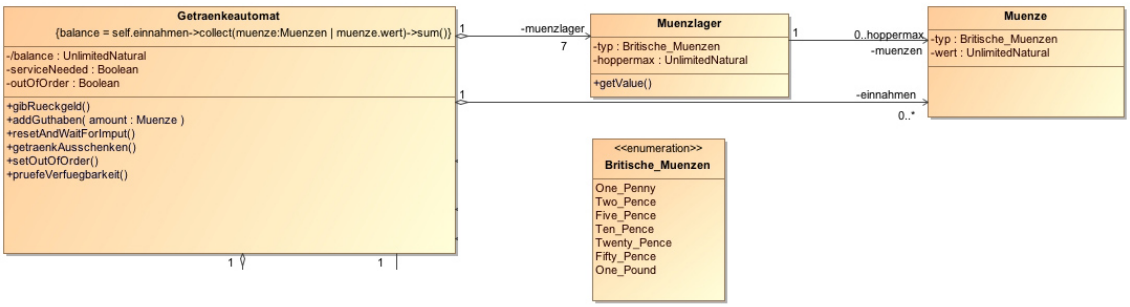


Abbildung 6: Klassendiagramm des Getränkeautomaten (Bezahlprozess)

3.1.3. Inhalt und Bestellung

Die Inhalte des Getränkeautomaten werden in den Klassen Inhaltslager und Inhalt mit deren Unterklassen abgebildet.

3.1.3.1. Getraenke_Typ und Zutaten_Typ

Laut der textuellen Beschreibung gibt es fünf Inhaltstypen, davon die drei Getränke Kaffee, Tee und heiße Schokolade sowie die Zutaten Milchpulver und Zucker. Diese werden in zwei Enumerations *Getaenke_Typ* und *Zutaten_Typ* abgebildet.

3.1.3.2. Inhalt, Getraenk und Zutat

Die abstrakte Klasse *Inhalt* enthält das Attribut *preis*, welches den Preis der Inhalte (Getränk oder Zutat) in der kleinsten Einheit (pence) angibt. Die beiden Unterklassen *Zutat* und *Getraenk* haben jeweils ein Attribut *art* vom Typ *Getraenke_Typ* bezieungsweise *Zutaten_Typ*

3.1.3.3. Inhaltslager, Inhaltslager_Getraenk und Inhaltslager_Zutat

Da die Anzahl der Getränke im Automaten abgebildet werden soll, hat der Getränkeautomat für jeden Getränke- und Zutatentyp jeweils ein Inhaltslager. Um die Inhaltslager der Getränke und Zutaten zu unterscheiden, hat die abstrakte Klasse *Inhaltslager* zwei Unterklassen *Inhaltslager_Getraenk* und *Inhaltslager_Zutat*. Beide Klassen erben die Attribute *max:UnlimitedNatural,* welche den maximalen Füllstand angibt, *anzahl:UnlimitedNatural* für den aktuellen Füllstand und das abgeleitete Attribut *inhalt,* das alle Instanzen des entsprechenden Inhaltstyps (*Getraenk* oder *Zutat*) enthält.

Dass das Attribut *inhalt* des Tee-Inhaltslagers auch Instanzen von *Getraenk* mit dem Attributwert *art = Tee* beinhaltet kann, wird bei der Modellierung mittels einer derived Union und einem subset dieser ausgedrückt. Das Attribut *inhalt* der abstrakten Klasse Inhaltslager stellt die gesamte Menge dar. Das Property *{subsets inhalt}* der Klassen *Inhaltslager_Getraenk* und *Inhaltslager_Zutat* gibt an, dass dieses Attribut eine Teilmenge des *inhalt*-Attributs ist.

Der aktuelle Füllstand jeder Instanz eines Inhaltslagers (*Getraenk* oder *Zutat*) wird über das OCL Constraint *{self.anzahl = self.Getraenk->size()}* spezifiziert. Die Anzahl der vorhandenen Getränke und Zutaten entspricht der Anzahl der vorhandenen Instanzen der Klassen Zutat und Getränk. Diese Zusammenhänge werden in Abbildung 7 anhand eines Objektdiagramms noch einmal verdeutlicht.

Die Instanz von *Inhaltslager_Getraenk* ist hier im Beispiel das Tee-Inhaltslager mit der maximalen Stückzahl 200 und dem aktuellen Füllstand von 2. Der Füllstand von 2 ergibt sich aus der Anzahl der Inhalte im Attribut *getraenk*, die aktuell zwei Instanzen von Getraenk (Tee) enthält. Das Attribut *inhalt* des Klassendiagramms wird nicht befüllt, da es sich um ein abgeleitetes Attribut (engl. derived) handelt.

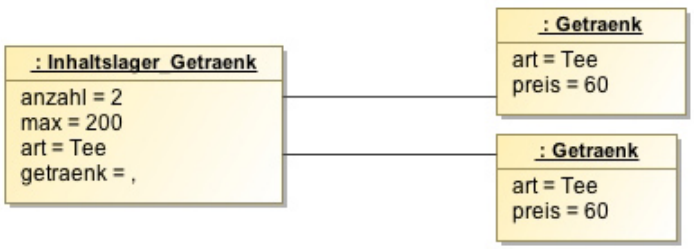


Abbildung 7: Objektdiagramm eines Beispiel Tee-Inhaltslagers

Analog zu Abbildung 7 verhalten sich auch die Inhaltslager der verbleibenden Getraenke und Zutaten.

3.1.3.4. Bestellung

Die Klasse *Bestellung* repräsentiert eine physische Bestellung am Getränkeautomaten. Wenn der Kunde seine Auswahl getroffen hat und den Button D drückt, wird die Auswahl der Bestellung hinzugefügt. Die Bestellung hat zwingend ein Getränk (Attribut *getraenk*) und 0 bis 2 Zutaten (Attribut *zutat*). Allerdings gibt es laut der textuellen Beschreibung die Einschränkung, dass wenn Schokolade gewählt wurde, keine Zutaten ausgewählt werden können. Dies wird über die Multiplizität 0..2 und das OCL Contraint *if self.getaenk.art = Getraenke_Typ.Schokolade then self.zutat = NULL ausgdrückt.*

3.1.3.5. Diskussion

In diesem Klassendiagramm wurden die *Inhaltslager* und die *Inhalte* als abstrakte Oberklassen modelliert und generalisieren jeweils getrennte Klassen für Getränke und Zutaten, die ihre Art neben den Klassennamen über die Werte der Enumerations *Getraenke_Typ* und *Zutaten_Typ* definieren. Diese Modellierung wurde gewählt um explizit zwischen Zutat und Getränk unterscheiden zu können.

Es gibt mehrere Alternativen, wie man dieses Konstrukt hätte anders modellieren können. Man hätte beispielsweise komplett auf die explizite Unterscheidung zwischen Getränk und Zutat verzichten können. Dies würde die Vererbungsbeziehung von *Inhaltslager_Getraenk* und *Inhaltslager_Zutat* und die Beziehung zwischen *Inhalt*, *Getraenk* und *Zutat* hinfällig machen. Zumdem würde es das Klassendiagramm sehr vereinfachen: Man könnte man auf die derived Union, das subsetted Propertiy sowie auf den OCL Constraint verzichen, der die Anzahl genau festlegt. Die Enumerationen Getraenke_Typ und Zutaten_Typ könnten dann zusammengefasst werden. Die Abbil8ung zeigt schematisch, wie die Modellierung des Inhalts und der Bestellung aussehen könnte.

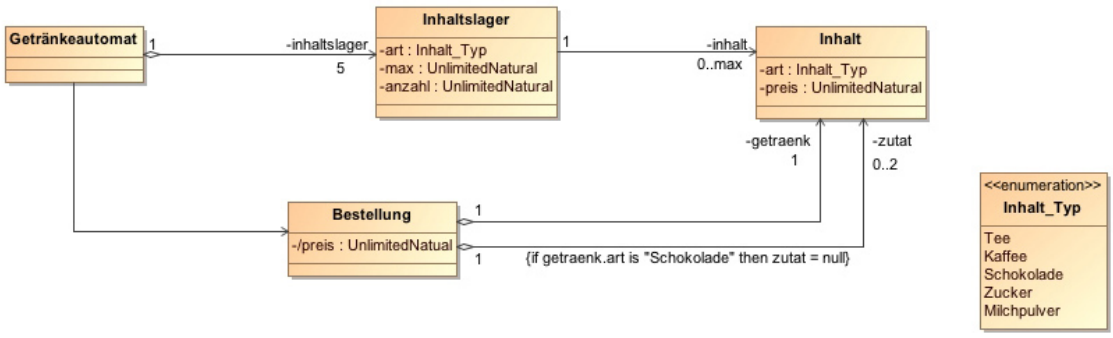


Abbildung 8: Alternatives Klassendiagramm des Inhalts und der Bestellung

Eine weitere Alternative zum beschriebenen Klassendiagramms wäre die Getränke und die Zutaten nicht in Enumerationen, sondern als Unterklassen von *Getraenk* oder *Inhalt* zu modellieren. Zudem könnte man auch die 5 *Inhaltslager* dann auch direkt als Tee-Inhaltslager, Kaffe-Inhaltslager, … angeben.

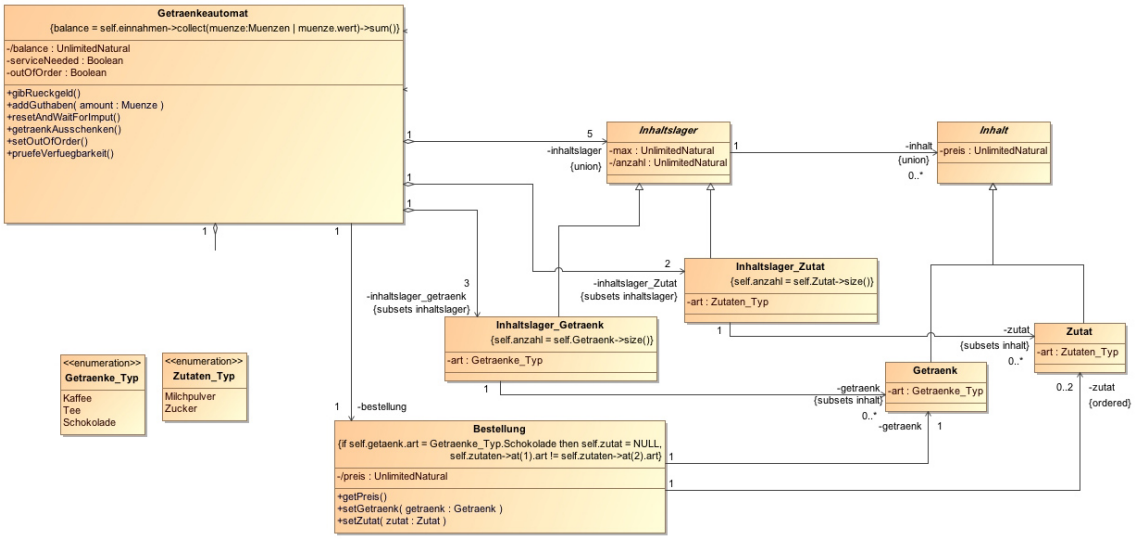


Abbildung 9: Klassendiagramm des Getränkeautomaten (Inhalt und Bestellung)

3.2. Verhaltensmodellierung: Aktivitätsdiagramm

Ein Aktivitätsdiagramm beschreibt die Ablaufmöglichkeiten eines Systems mit Hilfe von Aktivitäten. „Eine Aktivität ist ein Zustand mit einer internen Aktion und einer oder mehreren ausgehenden Transitionen, die automatisch dem Abschluss der internen Aktion folgen. Eine Aktivität ist ein einzelner Schnitt in einem Ablauf“ [Oesterreich 2001].

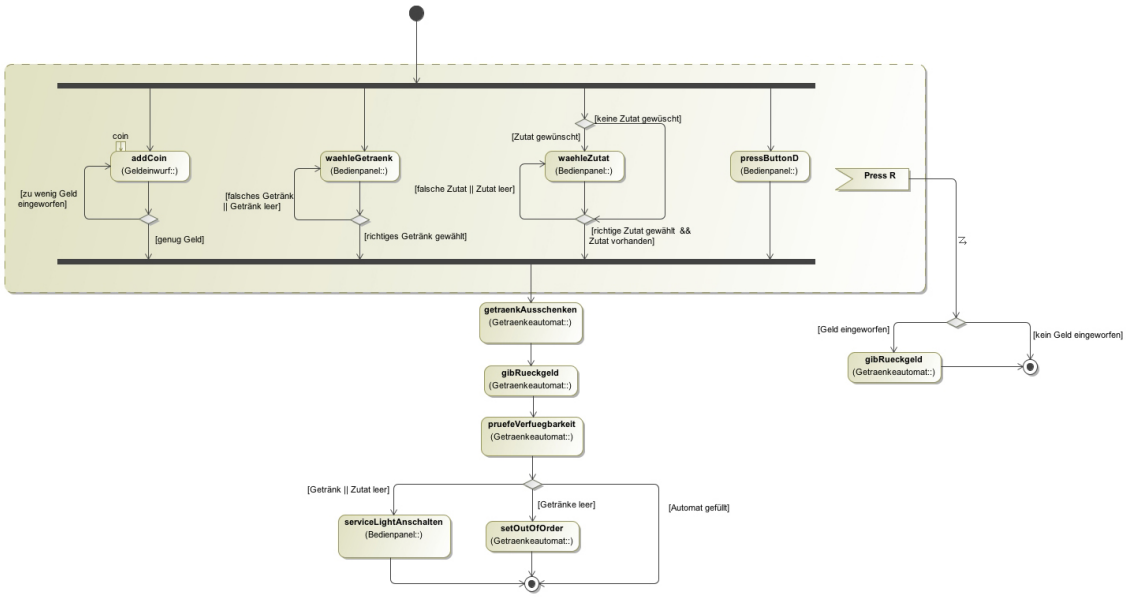


Abbildung 10: Aktivitätsdiagramm des Getränkeautomaten

3.2.1. Ablauf des Bestellvorgangs

Die Beschreibung des Getränkeautomaten beschreibt den Ablauf einer Bestellung nur indirekt und lässt einige Fragen offen. Beispielsweise wird nicht beschrieben in welcher Reihenfolge der Ablauf der Getränkewahl, der Zutatenwahl und des Geldeinwurf erfolgen muss. Es wäre denkbar, dass zuerst das Getränk und dann die Zutat gewählt werden muss bevor man Geld einwerfen kann. Beim Abbruch des Vorgangs durch Drücken des R-Buttons würden alle Eingaben wieder zurückgesetzt. Allerdings wäre bei diesem Vorgang ein Umwählen des Getränks nicht möglich. Andere Getränkeautomaten schreiben dem Kunden wiederum die Reihenfolge der Auswahl nicht vor. Ein Kunde kann beispielsweise zuerst das Geld einwerfen, dann die Getränke wählen und

anschließend die Zutaten. Entscheidet er sich nun für ein anderes Getränk oder ist der Vorrat des Getränks erschöpft kann er es umwählen. Hat er zuvor zu wenig Geld eingeworfen muss er nachzahlen.

Das erste Beispiel mit einer festgelegten Reihenfolge vereinfacht die Modellierung des Aktivitätsdiagramms stark. Auf eine Aktivität folgt die nächste. Parallele Abläufe sind nicht möglich. Die zweite Möglichkeit ist allerdings komplizierter und lässt mehrere Modellierungen zu. Eine Herangehensweisen wäre den Ablauf in allen Möglichkeiten auszuführen. Bei den vier parallelen Aktionen „Geld einwerfen“, „Getränk wählen“, „Zutat wählen“ und „Button D drücken“ ergeben sich somit $2^4 = 16$ (|POW(A1, A2, A3, A4)|) verschiedene Möglichkeiten, die in einem Aktivitätsdiagramm nur sehr unübersichtlich darzustellen sind. Eine andere Modellierungsweise wurde im folgenden Diagramm gewählt. Die parallelen Abläufe werden hier über Fork und Join in einer interruptable Region dargestellt (siehe Abbildung 11).

Die Aktionen Geld einwerfen, Getränk wählen, Zutat wählen und Button D drücken können dadurch parallel in beliebiger Reihenfolge ablaufen. Der Kontrollfluss läuft erst dann weiter, wenn die vier Abläufe ausgeführt wurden. Erst wenn genügend Geld eingeworfen, das gewünschte Getränk und die gewünschte (optionale) Zutat gewählt wurde, wird der Bestellvorgang abgeschlossen und der Ausschank des Getränks kann erfolgen.

Allerdings bildet die Modellierung mit Fork und Join den Abbruch durch Drücken des R-Buttons nicht ab. Daher wird der gesamte Vorgang in einer Interruptable Region modelliert, die durch das Event „Button R drücken“ unterbrochen werden kann. Das Drücken des R-Buttons bewirkt somit, dass das der Ablauf der Getränkeauswahl unterbrochen wird und sofern nötig das Rückgeld ausgegeben wird.

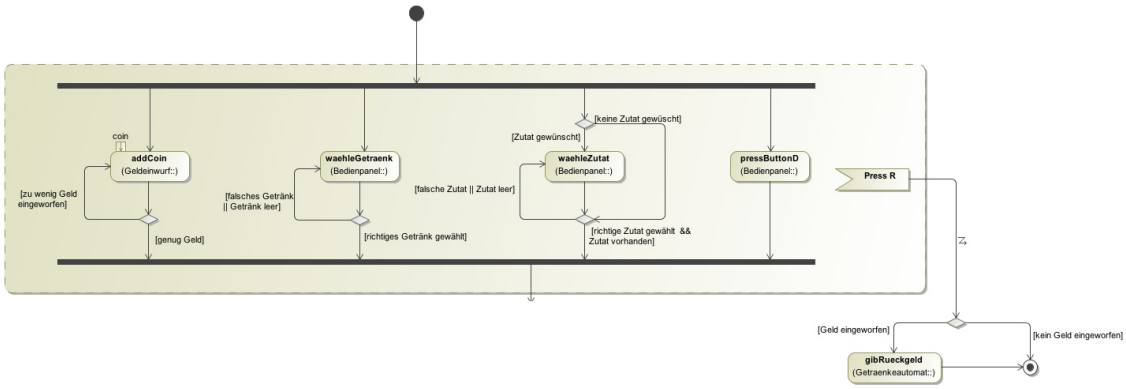


Abbildung 11: Aktivitätsdiagramm des Getränkeautomaten (Bestellvorgang)

3.2.2. Getränkeausgabe

Wurde der Bestellvorgang abgeschlossen, muss zuerst die Verfügbarkeit geprüft werden. Sind Getränk und Zutat vorhanden wird das Getränk ausgegeben und eventuell zu viel eingeworfenes Geld zurückgegeben. Anschließend wird wiederum die Verfügbarkeit geprüft, um gegebenenfalls das Service Licht anzuschalten wenn der Vorrat eines Getränks und/oder einer Zutat erschöpft ist, oder den Getränkeautomat zu deaktivieren wenn der Vorrat der Getränke aufgebraucht ist und somit kein Ausschank mehr möglich ist. An dieser Stelle trifft die texutelle Beschreibung des Automaten wiederum keine genaue Aussage darüber ob und wie der Kunde außer dem Service Licht benachrichtigt wird welche Zutat oder welches Getränk nicht mehr vorhandenen ist. Zudem lässt sich anhand der Beschreibung nicht sagen wann auf Verfügbarkeit des Getränks und/oder der Zutat geprüft wird. Bei der aktuellen Modellierung wird der Füllstand direkt bei Auswahl des Getränks und der Zutat der geprüft und der Benutzer eventuell dementsprechend benachrichtigt. Eine Alternative dazu könnte die Prüfung nach Abschluss des Bestellvorgangs sein. Dem Kunden müsste dann allerdings die Option eingeräumt werden, die Auswahl wiederum anzupassen, oder den Bestellvorgang abzubrechen, was an dieser Stelle aber nicht sehr benutzerfreundlich wäre.

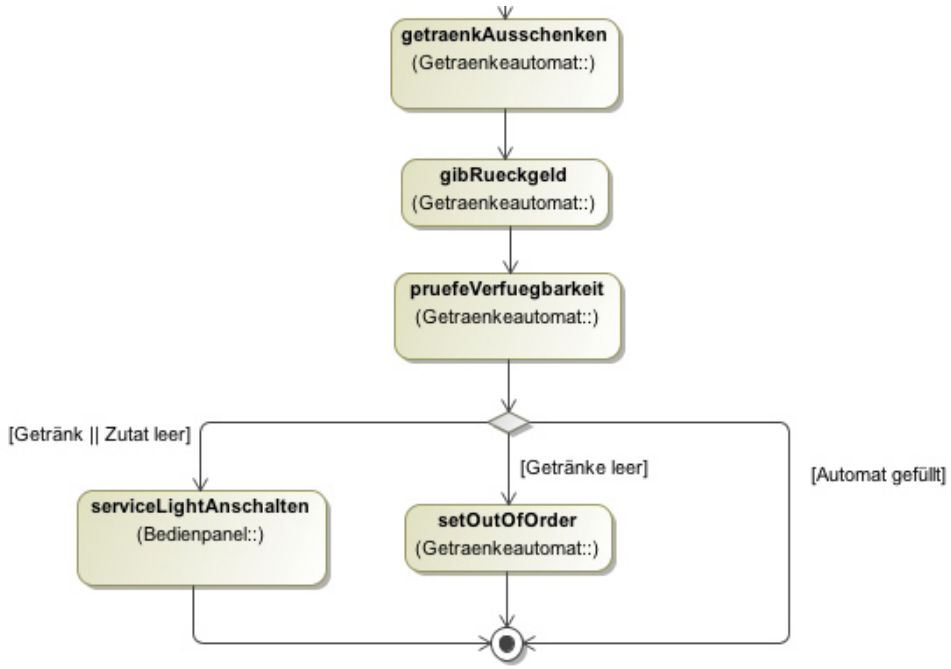


Abbildung 12: Aktivitätsdiagramm des Getränkeautomaten (Getränkeausgabe)

3.2.3. Gesamtsicht

Da das Aktivitätsdiagramm aus Abbildung 13 nur die Bestellung eines Getränks abbildet, wird im folgenden nochmals eine Gesamtsicht auf den Getränkeautomat gezeigt, die den „Lebenszyklus“ des Getränkeautomaten abbildet und das Aktivitäts-diagramm der Bestellung ergänzt. Insbesondere die Prüfung ob der Automat Betriebs-bereit oder Out of Order ist (nach dem Einschalten des Automaten und nach der Benutzung) wird in dieser Ansicht nochmals genau aufgeführt).

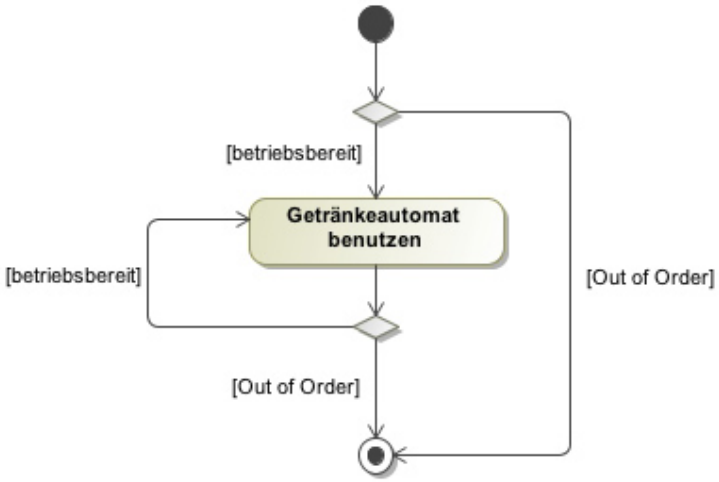


Abbildung 13: Aktivitätsdiagramm des Getränkeautomaten (Gesamtsicht)

3.3. Verhaltensmodellierung: Zustandsdiagramm

„Ein Zustandsdiagramm zeigt eine Folge von Zuständen, die ein Objekt im Laufe seines Lebens einnehmen kann und aufgrund welcher Stimuli Zustandsänderungen stattfinden. Ein Zustandsdiagramm beschreibt eine hypothetische Maschine (Endlicher Automat), die sich zu jedem Zeitpunkt in einer Menge endlicher Zustände befindet." [Oesterreich 2001, S. 303]

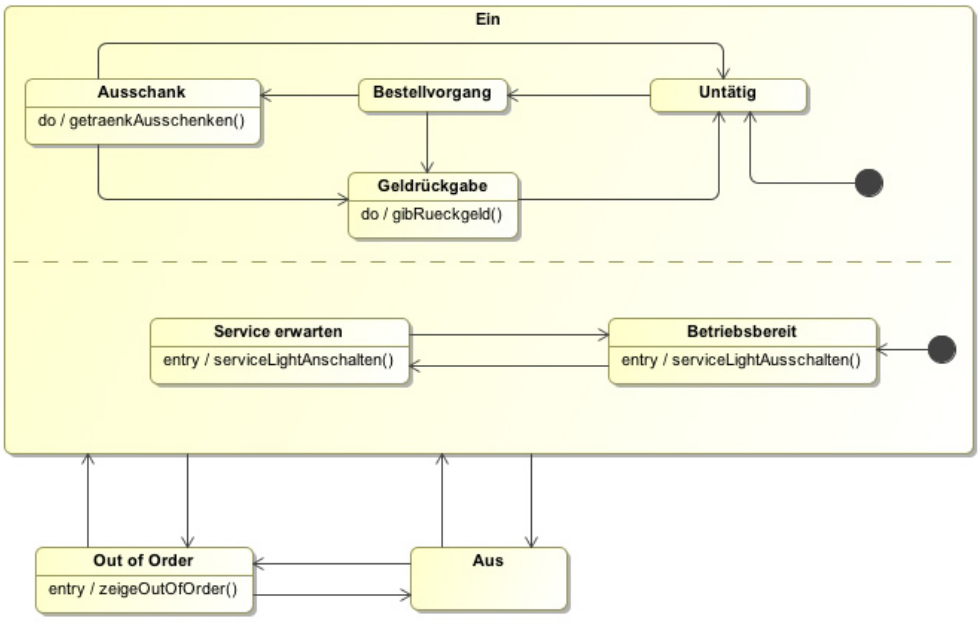


Abbildung 14: Zustandsdiagramm des Getränkeautomaten

Abbildung 14 zeigt das Zustandsdiagramm des Getränkeautomaten, welches die beiden Grundzustände *Aus, Ein* und *Out of Order* abbildet. Der Zustand *Aus* ist denkbar beim Transport, der Wartung oder der vorübergehenden Stilllegung des Automaten. Der Zustand *Out of Order* bildet den Automaten in einem Zwischenzustand ab, indem zwar keine Getränke mehr ausgeschenkt werden können, der Automat allerdings auch nicht ausgeschaltet ist. Der Zustand *Ein* beschreibt den Automaten im Betrieb und beinhaltet die parallelen Unterzustände des Bestellmechanismus und des Service Mechanismus (nach Oesterreich 2001, S. 308). Da der Zustand „Service erwarten" und Betriebsbereit immer parallel zu dem Bestellvorgang ablaufen kann, können die Zustände im Diagramm mittels eines „Orthogonal State" abgebildet werden.

3.3.1. Zustandsverlauf des Automaten

Beim Einschalten des Automaten befindet sich dieser zuallererst im Zustand *Untätig*. Parallel dazu befindet er sich im Zustand *Betriebsbereit*. Wenn mindestens ein Getränk oder mindestens eine Zutat nicht mehr vorhanden ist, wird der Zustand von *Betriebsbereit* auf *Service erwarten* gewechselt, das Service Licht angeschaltet und eventuell der Kunde über das Report Display benachrichtigt. Kann kein Getränk mehr ausgeschenkt werden, da der Vorrat an Getränken erschöpft ist, wird in den Zustand *Out Of Order* gewechselt. Von der Begrifflichkeit würde man den Zustand *Out of Order* als Unterzustand von Ein sehen. Allerdings können parallel keine Bestellungen mehr durchgeführt werden. Daher wird der Zustand als Zwischenzustand von *Aus* und *Ein* modelliert.

Der Bestellmechanismus selbst hat vier Zustände: *Untätig*, *Bestellvorgang*, *Ausschank* und *Geldrückgabe*. Im Zustand *untätig* wartet der Automat auf die Eingabe des Kunden. Drück ein Kunde einen Button oder wirft er Geld ein, wechselt der Automat in den *Bestellvorgang*. Der Bestellvorgang wurde in diesem Diagramm in einem Zustand modelliert. Es wäre allerdings denkbar diesem Zustand in weitere Unterzustände aufzugliedern (siehe Abschnitt 3.3.2.). Ist der Bestellvorgang abgeschlossen wird in den Zustand *Ausschank* gewechselt und das Getränk ausgegeben. Anschließend wird dem Kunden, wenn nötig, das Rückgeld gegeben (Zustand *Geldrückgabe*). Nach Abschluss des Ausschanks oder der Geldrückgabe wird dann wieder in den *Untätig* Zustand gewechselt.

3.3.2. Unterzustände eines Bestellvorgangs

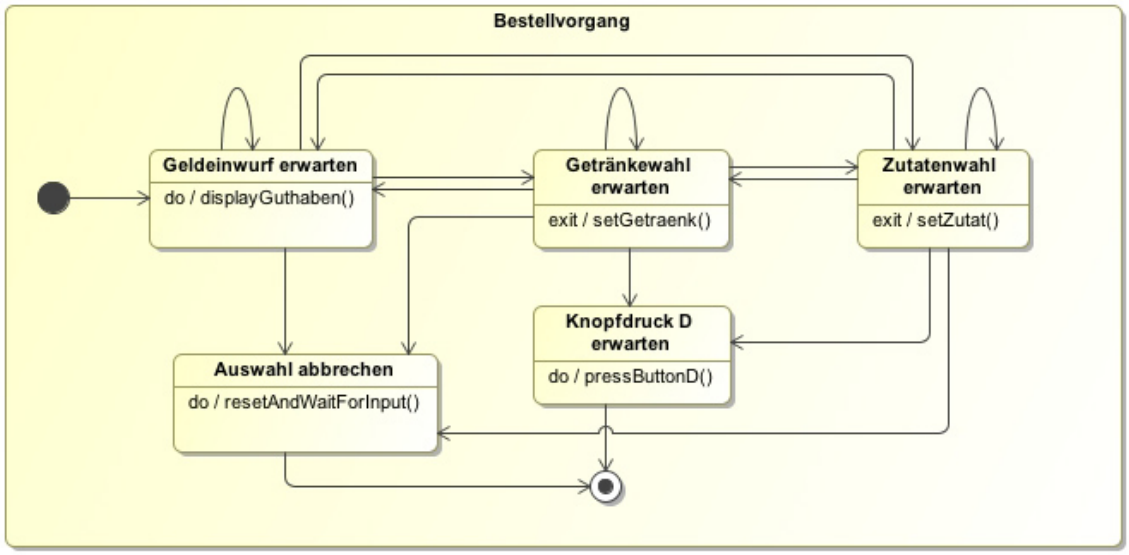


Abbildung 15: Unterzustände eines Bestellvorgangs

Der Zustand eines Bestellvorgang lässt sich anhand der Abbildung 15 detaillierter beschreiben. Die Zustände *Geldeinwurf erwarten*, *Getränkewahl erwarten* und *Zutatenwahl erwarten* sind von der Reihenfolge unabhängig, da der Kunde nicht gezwungen wird zuerst das Geld einzuwerfen oder die Getränke auszuwählen. Zwischen ihnen kann daher in beide Richtungen hin und her gewechselt werden. Für den weiteren Ablauf gibt es zwei Möglichkeiten: entweder der Kunde entscheidet sich die Auswahl zu bestellen, indem er den Knopf D drückt (zustand *Knopfdruck D erwarten*) oder er bricht die Auswahl (beispielsweise durch Drücken des R-Buttons) ab (Zustand *Auswahl abbrechen*).

3.3.3. Alternatives Zustandsdiagramm

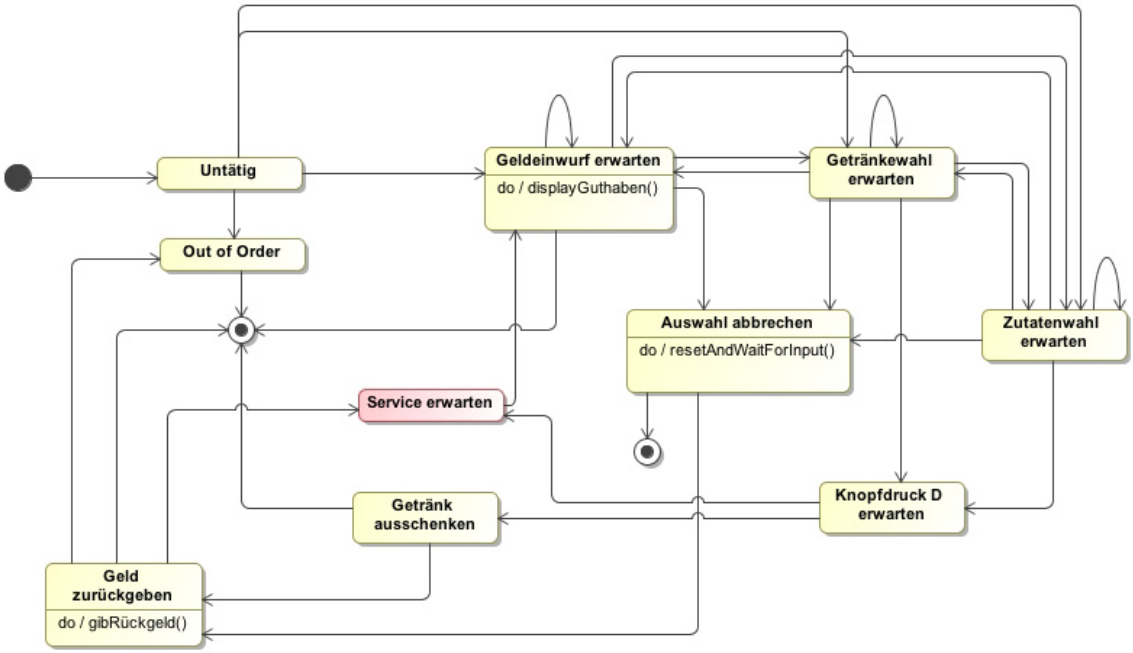


Abbildung 16: Alternatives Zustandsdiagramm (mit Fehlern)

Abbildung 16 zeigt ein alternatives Zustandsdiagramm, dass nur die Zustände eines eingeschalteten Automaten abbildet. Die Modellierung beinhaltet zudem nicht die Parallelität der Zustände *Betriebsbereit*, *Service erwarten* (und *Out of Order*), sondern zeigt den Zusammenhang linear. Allerdings fällt hier auf, dass der Zustand *Service erwarten*, der eigentlich laut der Beschreibung parallel ablaufen soll, den Automaten von seiner Betriebsbereitschaft abhalten würde. Der Automat würde sich so lange im Zustand befinden bis dieser wieder befüllt wurde. Dies wäre beim *Out Of Order* Zustand ein gewünschter Effekt, da kein Getränk mehr ausgegeben werden kann, jedoch nicht beim Service erwarten. Dies war der Grund dafür die Abläufe in einem Orthogonal State (siehe Abbildung 14) darzustellen.

4. Vergleich zur Spezifikation in Z

In diesem Kapitel wird das in Kapitel 3. vorgestellte Modell des Getränkeautomaten mit der vorliegenden Modellierung in Z verglichen. Z ist eine Notation zur formalen Spezifikation von Softwaresystemen und basiert auf der Zermelo-Fraenkel-Mengenlehre und der Prädikatenlogik erster Stufe. Ein Schema besteht dabei aus einer Anzahl typisierter Variablen und Bedingungen, welche an die Belegungen der Variablen gestellt werden.

Da ein vollständiger Vergleich im Rahmen dieser Semesterarbeit nicht möglich ist, werden nur einige vorgegebenen Z-Spezifikationdetails verglichen.

4.1. Vergleich der Datentypen

4.1.1. Selection_Buttons

Die Z-Modellierung des Getränkeautomaten beschreibt die Auswahl Buttons mittels eines Sets wie folgt:

$$Selection\_buttons ::= TEA \mid COFFEE \mid WHITE \mid SWEET \mid CHOCOLATE$$

In der UML Modellierung aus Abschnitt 3.1.1.2. wurden die Buttons exemplarisch beschrieben. Die Klasse *Getraenkeautomat* assoziiert dabei die Klasse *Bedienbuttons* mit der Kardinalität von sieben. An dieser Stelle wurde zwischen den Buttons Tee, Kaffee, Schokolade, Milchpulver, Zucker, D oder R nicht explizit unterschieden. Die Modellierung im vorgestellten UML Diagramm weicht damit von der vorliegenden Z-Modellierung ab.

Um das UML-Diagramm an Z anzugleichen sind mehrere Modellierungen möglich: eine 1:1 Umsetzung und eine Umsetzung die konsistent zur bestehende UML Modellierung ist.

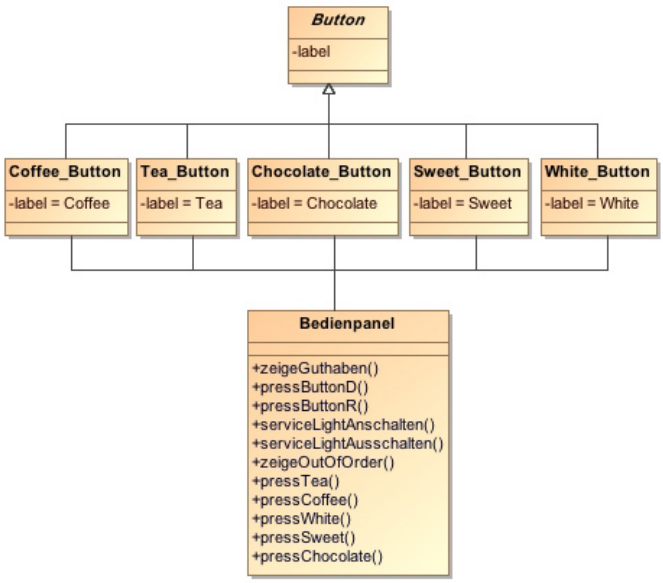


Abbildung 17: UML-Klassendiagramm der Selection Buttons (1)

Für die direkte Umsetzung in UML könnten fünf Klassen (*Tea_Button*, *Coffee_button*, …) modelliert werden, die von einer abstrakten Klasse *Button* erben (vgl. Abbildung 17). Andererseits könnte man nur auch nur eine Klasse *Button* erstellen, die ihr Label aus der einer Enumeration mit den Literalen *Tea*, *Coffee*, *Chocolate*, *Sweet* und *White* bezieht.

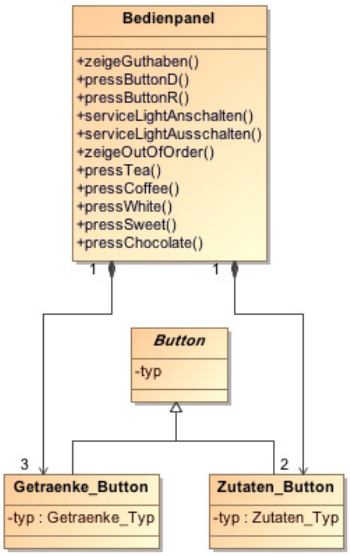


Abbildung 18: UML-Klassendiagramm der Selection Buttons (2)

Um das bestehende Klassendiagramm anzupassen um der Z-Modellierung des Getränkeautomaten bestmöglich zu entsprechen, könnte man die beiden bestehenden Enumerationen *Getraenke_Typ* und *Zutaten_Typ* verwenden. In der UML-Modellierung wurde zwischen Getränken und Zutaten unterschieden. Um das Modell nun konsistent

zu halten, könnte man auch bei den Buttons zwischen Getränke-Buttons und Zutaten-Buttons unterscheiden. Beide Button Klassen würden dann wieder von einer abstrakten Oberklasse Button erben (siehe Abbildung 18).

4.1.2. Ingredient

Die Z-Modellierung beschreibt die Zutaten, die der Automat enthalten muss um die Getränke auszuschenken, im Set *Ingredient:*

$$Ingredient ::= Milk\_powder \mid Chocolate\_powder \mid Tea\_bag \mid Coffee\_granules \mid Sugar \mid Water$$

Bei den Zutaten unterscheidet sich die Z-Modellierung von der UML-Modellierung aus Kapitel 3. In Z werden die Zutaten (entspricht im UML Diagramm aus Abbildung 4 im übertragenen Sinne der abstrakten Klasse *Inhalt*) granularer aufgeführt. Zudem ist Wasser als Inhaltstyp vorhanden, welches in der UML Modellierung nicht berücksichtigt wurde. Zudem wird bei der Z-Modellierung, wie schon bei den Buttons, nicht zwischen Getränke und Zutaten unterschieden.

Als direkte UML Umsetzung des Ingredient Sets sind wiederum mehrere Möglichkeiten denkbar. Die Inhalte könnten einerseits durch einzelne Unterklassen einer abstrakten Oberklassse Zutat (oder Inhalt) modelliert werden, oder analog zu den Selection Buttons als Klasse Zutat (oder Inhalt), die ihren Typ aus einer Enumeration mit den Ingredients als Literalen bezieht.

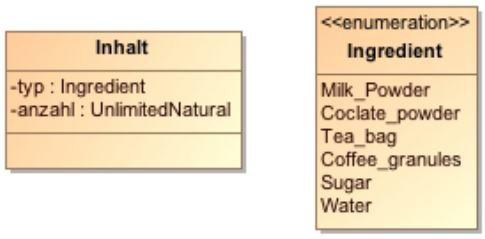


Abbildung 19: UML Klassendiagramm - Ingredient

4.1.3. Message

Notation der Messages des Getränkeautomaten in Z:

$$Message ::= this\ drink\ is\ unavailable \mid insert\ more\ money\ and\ select\ drink\ again \mid correct\ change\ unavailable \mid no\ cups\ in\ machine \mid try\ another\ coin \mid not\ in\ use \mid insert\ coin$$

Die Messages, die der Getränkeautomat im Reportdisplay ausgeben kann, wurden durch die textuelle Beschreibung nicht genauer spezifiziert. Diese wurde daher in der UML Modellierung des Automaten auch nicht berücksichtigt. Würde man diese jedoch modellieren, würde man diese in der Regel als statische Variablen z.B. in der Klasse *Display* (siehe Abbildung 4) realisieren.

4.1.4. Onoff

Das Onoff Set in der Z-Modellierung gibt die Zustände der Service Lampe an.

$$Onoff ::= on \mid off$$

In der UML Modellierung wird dieser Zustand in der Klasse Service-Lampe durch die boolsche Variable *zustand* beschrieben.

4.1.5. Coin

Das Z-set *Coin* beinhaltet alle Münzen, die in Großbritannien erhältlich sind. Zusätzlich enthält dieser einen Typ für alle anderen Münzen, die nicht akzeptiert werden können.

$$Coin ::= One\_Penny \mid Two\_Pence \mid Five\_Pence \mid Ten\_Pence \mid Twenty\_Pence \mid Fifty\_Pence \mid One\_Pound \mid Unacceptable\_coin$$

Die UML-Modellierung des Getränkeautomaten ähnelt an dieser Stelle sehr stark der Modellierung in Z. Die Klasse *Muenze* repräsentiert dabei eine physische Münze und könnte mit dem Set *Coin* verglichen werden. Einziger Unterschied ist der Inhalt „Unacceptable_coin“, der im UML Diagramm nicht abgebildet wird. Wird eine Münze nicht akzeptiert, so wird sie vom Automaten wieder ausgegeben und nicht im Münzlager

gespeichert. Daher wird auch kein Datentyp für eine nicht akzeptierte Münze in UML gebraucht. Der Umstand einer nicht akzeptierten Münze würde sich nur im UML-Aktivitätsdiagramm zeigen (siehe Abbildung 20).

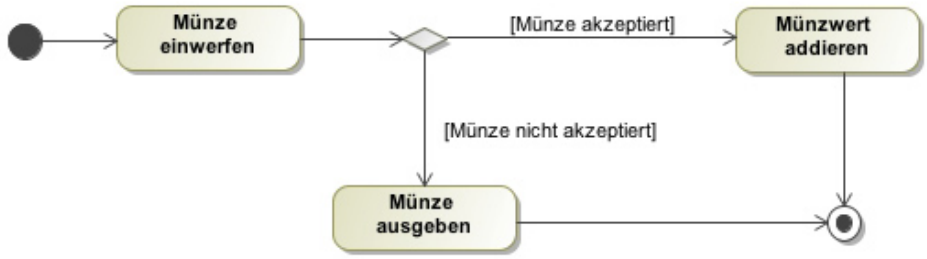


Abbildung 20: UML-Aktivitätsdiagramm eines Münzeinwurfs (Münzannahme)

4.2. Sets

4.2.1. British_Coin

Das Set *British_Coin* beschreibt nur Münzen, die in Großbritannien erhältlich sind.

$$British\_Coin == \{c : Coin \mid c \neq Unacceptable\_coin\}$$

Die Münzwerte beziehungsweise der Typ der Münze wird im UML-Klassendiagramm durch die Enumeration *Britische_Muenze* bestimmt, die in Z dem Set *Britische_Muenze* entspricht.

4.2.2. Drink

Das Drink-Set enthält alle möglichen Kombinationen von Getränken und Zutaten die über die Knöpfe ausgewählt werden können. Normalerweise würde dies der Potenzmenge von TEA, COFFEE, CHOCOLATE, WHITE und SWEET (*selected_buttons*) entsprechen, allerdings legt die Beschreibung des Getränkeautomaten fest, dass CHOCLATE nicht mit SWEET und WHITE kombiniert werden kann, da es diese schon implizit enthält.

$$\begin{aligned} Drink == \{&\{TEA\}, \{COFFEE\}, \{CHOCOLATE\}, \{TEA, WHITE\}, \\ &\{TEA, SWEET\}, \{TEA, WHITE, SWEET\}, \{COFFEE, WHITE\}, \\ &\{COFFEE, SWEET\}, \{COFFEE, WHITE\ SWEET\}\} \end{aligned}$$

In Klassendiagramm entspricht dieses Set indirekt der Klasse Bestellung, die ein Getränk und 0 bis 2 Zutaten enthalten kann. Das Klassendiagramm bildet nicht direkt ab welche Kombinationsmöglichkeiten der Benutzer beim Drücken der Knöpfe hat. Lediglich über den OCL Constraint *{if self.getaenk.art = Getraenke_Typ.Schokolade then self.zutat = NULL}* wird beschrieben, dass eine heiße Schokolade keine Zutaten enthalten darf. Theoretisch könnte man diesen Umstand auch im Aktivitätsdiagramm darstellen, allerdings beschreibt das Aktivitätsdiagramm im Kapitel 3.2. das Zusammenspiel zwischen Getränkewahl und Zutatenwahl hierfür nicht detailliert genug. Abbildung 21 zeigt daher exemplarisch die Getränke- und Zutatenauswahl anhand eines Aktivitätsdiagramms (ohne die Berücksichtigung der freien Abfolge zwischen Getränkewahl, Zutatenwahl, Geldeinwurf).

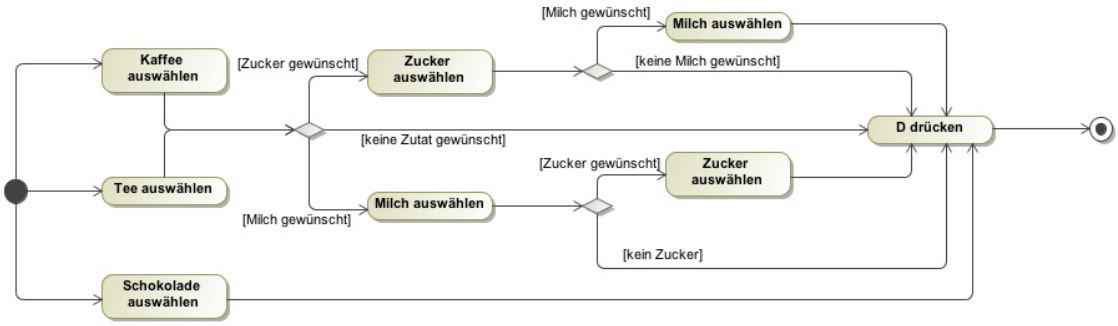


Abbildung 21: Aktivitätsdiagramm der Getränkewahl (exemplarisch)

4.2.3. List_of_Ingredients

Das Set *List_of_Ingredients* entspricht den Zutaten aller möglichen Kombinationen die für die Zubereitung eines Drinks benötigt werden.

$$List\_of\_ingredients == \{\{Tea\_bag, Water\}, \{Coffee\_granules, Water\}, \{Chocolate\_powder, Water\}, \{Tea\_bag, Water, Milk\_powder\}, \{Tea\_bag, Water, Sugar\}, \{Tea\_bag, Water, Sugar, Milk\_powder\}, \{Coffee\_granules, Water, Milk\_powder\}, \{Coffee\_granules, Water, Sugar\}, \{Coffee\_granules, Water, Sugar, Milk\_powder\}\}$$

Die List_of_Ingredients entspricht im UML-Klassendiagramm direkt keiner Klasse, sondern wird durch die Kombination von Bestellung und des Inhaltslagers ausgedrückt. Das Inhaltslager enthält alle Instanzen eines Getränks oder einer Zutat. Die Bestellung enthält das Getränk und die Zutaten, die für eine Bestellung des Kunden ausgegeben werden.

Die Anzahl der Inhalte und der Getränke wird über OCL Contraints zu den Inhalten ausgedrückt. Das UML Klassendiagramm in Kapitel 3. modelliert aber nicht die möglichen Kombinationen der Zutaten.

4.3. Relations/Functions

4.3.1. Receipe

Das Set Ingredient ist für jedes Getränk einzigartig. Jedes Getränk hat ein anderes Rezept und damit eine bestimmte Kombination von Zutaten. Die Rezepte werden in Z als totale Injektion vom Datentyp Drinks zu dem Set List_of_ingredients definiert. Totale Injektion daher, da jeder Drink auf eine Kombination der List_of_ingredients abgebildet wird.

$$Recipe : Drink \rightarrowtail List\_of\_ingredients$$

$$\begin{aligned} Recipe = \{&\{TEA\} \mapsto \{Tea\_bag, Water\} \\ &\{COFFEE\} \mapsto \{Coffee\_granules, Water\} \\ &\{CHOCOLATE\} \mapsto \{Chocolate\_powder, Water\} \\ &\{TEA, WHITE\} \mapsto \{Tea\_bag, Water, Milk\_powder\} \\ &\{TEA, SWEET\} \mapsto \{Tea\_bag, Water, Sugar\} \\ &\{TEA, WHITE, SWEET\} \mapsto \{Tea\_bag, Water, Sugar, Milk\_powder\} \\ &\{COFFEE, WHITE\} \mapsto \{Coffee\_granules, Water, Milk\_powder\} \\ &\{COFFEE, SWEET\} \mapsto \{Coffee\_granules, Water, Sugar\} \\ &\{COFFEE, WHITE, SWEET\} \mapsto \{Coffee\_granules, Water, Sugar, Milk\_powder\}\} \end{aligned}$$

In der Z-Modellierung werden die Zutaten und die Zutatenkombinationen (Rezepte) aufgeführt. In der UML wird dieser Zusammenhang durch die Klasse Bestellung im Klassendiagramm und im Aktivitätsdiagramm dargestellt.

4.3.2. Worth

Die Worth-Funktion bildet einen Münztyp auf einen Wert ab. Jede britische Münze bekommt damit den Wert in der kleinsten Wärungseinheit den sie textuell beschreibt.

$$Worth : British\_Coin \rightarrowtail \mathbb{N}$$

$$\begin{aligned} Worth = \{&One\_Penny \mapsto 1, Two\_Pence \mapsto 2, Five\_Pence \mapsto 5, Ten\_Pence \mapsto 10, \\ &Twenty\_Pence \mapsto 20, Fifty\_Pence \mapsto 50, One\_Pound \mapsto 100\} \end{aligned}$$

In der UML Modellierung des Getränkeautomaten wird ähnlich verfahren. Die Enumeration *Britisch_Coin* enhält die Namen der Münze (One_Penny, Two_Pence, …), welche in der Klasse Muenze mit einem Wert kombiniert wird.

4.3.3. HopperMax

HopperMax gibt die maximale Anzahl des Füllstandes eines Münz-Hoppers jedes Münztypen an. In Z ist Hoppermax eine Abbildung von British_Coin auf eine natürliche Zahl.

$$HopperMax : British\_Coin \rightarrow \mathbb{N}_1$$
$$HopperMax = \{One\_Penny \mapsto .., \ldots, One\_Pound \mapsto ..\}$$

Die Klasse *Muenzlager* im UML-Klassendiagramm entspricht einem Münz-Hopper im Z. Die maximale Anzahl der Instanzen von *Muenze,* die das Attribut *muenzen* enthalten kann, wird durch die Multiplizität der Assoziation *0..hoppermax* ausgedrückt. *Hoppermax* ist dabei ein Attribut der Klasse *Muenzlager*. Die Anzahl der Münzlager wird durch die Anzahl der verschiedenen Münztypen (*Britische_Muenze / British_Coin*) bestimmt.

4.3.4. StockMax

StockMax beschreibt analog zu HopperMax die maximale Anzahl der Zutaten im Automaten und bildet das Set Ingredient auf eine natürliche Zahl ab.

$$StockMax : Ingredient \rightarrow \mathbb{N}_1$$
$$StockMax = \{Tea\_bag \mapsto .., \ldots, Water \mapsto \infty\}$$

Stockmax entspricht im UML-Klassendiagramm dem Attribut *max* der abstrakten Oberklasse *Inhaltslager* (beziehungsweise *Inhaltslager_Getraenk* und *Inhaltslager_Zutat*)

4.3.5. Value

Die Funktion Value berechnet den Wert einer Sammlung von Muenzen.

$$Value : \text{bag } British\_Coin \rightarrow \mathbb{N}$$

$$Value \, [\![\,]\!] = 0$$
$$\forall c : British\_Coin ; n : \mathbb{N} \bullet Value \, [\![c \mapsto n]\!] = Worth \, c * n$$
$$\forall b_1, b_2 : \text{bag } British\_Coin \bullet Value \, (b_1 \uplus b_2) = Value \, b_1 + Value \, b_2$$

Da es im Getränkeautomat UML-Modell zwei Assoziationen mit der Klasse *Muenze* gibt (den Inhalt des Behälters mit dem eingeworfenen Geld und die Behälter der Muenzvorräte), entspricht die Z Value Funktion der Funktion *getValue()* auf *Muenzlager* oder dem derived Attribute *balance* des Getränkeautomaten.

Für die Funktion *getValue()* gelten daher folgende Conditions:

```
context Muenzlager :: getValue():UnlimitedNatural
post result = muenzen->size()
```

Für das abgeleitete Attribut *balance* gilt folgende Invatiant:

```
context Getraenkeautomat inv:
self.balance = einnahmen->size()
```

5. OCL / Z-Profile

5.1. Einsatz der Object Constraint Language (OCL)

Im Kapitel 3. wurde bereits ansatzweise die Verwendung der OCL-Contraints im Klassendiagramm des Getränkeautomaten beschrieben. An dieser Stelle werden nun alle Constraints noch einmal ausführlich beschrieben.

5.2. Invariants

5.2.1. Balance Attribut der Klasse Getraenkeautomat

```
context Getraenkeautomat inv balance
balance = self.einnahmen->collect(muenze : Muenzen | muenze.wert)-
>sum()
```

Die Klasse Getraenkeautomat enthält ein abgeleitetes Attribut balance, das den Gesamtwert aller eingeworfenen Münzen eines Bestellvorgangs darstellt. Der Wert muss sich daher aus der Summe der Wertigkeit der eingeworfenen Münzen (Collection einnahmen) zusammensetzen. In OCL können mit der collect-Operation alle Instanzen oder Attribute einer Collection gesammelt werden. Der Rückgabewert der Operation ist eine bag. Darauf kann nun die sum-Operation angewendet werden kann, die die Werte aufsummiert. Das Ergebnis wird dem Attribut balance zugewiesen.

5.2.2. Anzahl der Inhalte des Inhaltslager

```
context Inhaltslager_Zutat inv inhalt
self.anzahl = self.zutat->size()

context Inhaltslager_Getraenk inv inhalt
self.anzahl = self.Getraenk->size()
```

Das vererbte Attribut *anzahl* der Klassen *Inhaltslager_Zutat* und *Inhaltslager_Getraenk* wird über OCL-Constraints definiert. Das Attribut soll dabei der Anzahl aller Instanzen der entsprechenden Unterklasse von *Inhalt* entsprechen. Dazu wird auf den Collections zutat oder getraenk die size() Operation aufgerufen und der Wert dem Attribut zugewiesen.

5.2.3. Zutatenbeschränkung der Schokolade-Bestellung

```
context Bestellung inv art
if self.getaenk.art = Getraenke_Typ.Schokolade
then self.zutat = NULL
```

Laut der textuellen Beschreibung dürfen keine Zutaten ausgewählt werden, wenn als Getränk Schokolade gewählt wurde. Dazu wird per OCL-Constraint das Attribut zutat auf NULL gesetzt wenn als Getränkeart Schokolade gewählt wurde.

5.2.4. Eindeutige Zuaten

```
context Bestellung inv uniqueZutat
self.zutaten->at(1).art != self.zutaten->at(2).art
```

Jede Zutat darf kann nur einmal ausgewählt werden. Es dürfen beispielsweise keine zwei „Portionen“ Milch bestellt werden. Daher werden über den OCL-Constraint die Elemente der geordnete Collection *zutaten,* die maximal zwei Elemente enthalten kann, miteinander verglichen. Diese dürfen entsprechend der Beschreibung nicht von der gleichen Art sein.

5.3. Z-Modell zum Z-Profil verallgemeinern

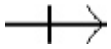
Viele der Z-Modellierungen lassen sich in UML immer ähnlich abbilden, daher könnte man Teile des Z-Modells zu einem Z-Profil zusammenfassen. Beispielsweise lassen sich die verschiedenen Z-Funktionen (Partiell, Total, Bijektiv, …) in UML als Assoziation darstellen. Die verschiedene Arten der Z-Funktionen unterscheiden sich nur in den Multiplizitäten der Assoziationen.

5.3.1. Funktionstypen

5.3.1.1. Beispiel partielle Funktion

Eine partielle Funktion von der Menge X in die Menge Y ist eine rechtseindeutige Relation, das heißt eine Relation, in der jedem Element er Menge X höchstens ein Element der Menge Y zugeordnet wird.

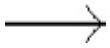
Formal: $\forall x \in X \; \exists y \in Y : f(X) = Y$

Z-Symbol: 

5.3.1.2. Beispiel totale Funktion

Eine totale Funktion von der Menge X in die Menge Y gibt an, dass es für jedes Element der Menge X mindestens ein Element der Menge Y gibt.

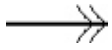
Formal: $\exists x \in X \; \exists y \in Y : f(X) = Y$

Z-Symbol: 

5.3.1.3. Beispiel surjektive Funktion

Eine surjektive Funktion bedeutet, dass jedes Element der Zielmenge mindestens einmal als Funktionswert angenommen wird, also mindestens ein Urbild hat. Eine surjektive Funktion ist damit rechtstotal.

Formal: $\forall y \in Y \; \exists x \in X : f(x) = y$

Z-Symbol: 

5.3.1.4. Beispiel injektive Funktion

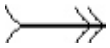
Eine Injektive Funktion bedeutet, dass jedes Element der Zielmenge höchstens einmal als Funktionswert angenommen wird. Es werden also keine zwei verschiedene Elemente der Definitionsmenge auf ein und dasselbe Element der Zielmenge abgebildet. Eine injektive Funktion ist damit linkseindeutig

Formal: $\forall y \in Y : (\exists ! x \in X : f(x) = y \vee \neg(\exists x \in X : f(x) = y))$

Z-Symbol: 

5.3.1.5. Beispiel Bijektive Funktion

Eine Funktion ist bijektiv, wenn sie verschiedene Elemente ihres Definitionsbereichs auf verschiedene Elemente der Zielmenge abbildet (injektiv) und wenn zusätzlich jedes Element der Zielmenge als Funktionswert auftritt (surjektiv). Eine bijektive Funktion hat daher immer eine Umkehrfunktion und ist invertierbar.

Z-Symbol: 

5.3.2. Funktionstypen in UML

Die verschiedenen Eigenschaften von Funktionen lassen sich in Multiplizitäten von Assoziationen ausdrücken. Diese werden in Form einer Tabelle den entsprechenden Z-Funktionen zugeordnet. Die Umsetzung in ein UML Diagramm erfolgt anschließend exemplarisch.

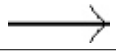
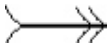
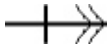
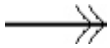
	Symbol	x	y
Partial Function		0..1	0..*
Total Function		1	0..*
Bijective Function		1	1
Partial Injecition		0..1	0..1
Total Injection		1	0..1
Partial Surjection		0..1	1..*
Total Surjection		1	1..*

Tabelle 1: Multiplizitäten der UML Assoziation anhand der Z-Funktionen

Abbildung 22: UML-Klassendiagramm exemplarisch anhand der Z-Funktionen

Das UML Klassendiagramm zeigt exemplarisch die Assoziation zwischen der Klasse X und der Klasse Y. Die Multiplizitäten sind entsprechend der Tabelle für x und y einzusetzen.

Diese verschiedenen Funktionstypen kann man nun in einem Z UML-Profil abbilden, welches später einem Modellierer bei der Umsetzung eines UML Diagramm aus einem in Z spezifizierten Model hilft und die Modellierung vereinfacht. Dazu wird die UML Metaclass *Association* der M2 Ebene um Stereotypen von *Association* der Funktionstypen (PartialFunction, TotalFunction, BijectiveFunction, …) auf der M1 Ebene erweitert. Abbildung 23 zeigt exemplarisch wie diese Beziehung für PartitialFunction und TotalFunction aussehen könnte. Die Trennung von M1 und M2 ist nur zur Verdeutlichung dargestellt und gehört normalerweise nicht zur Modellierung eines Profils.

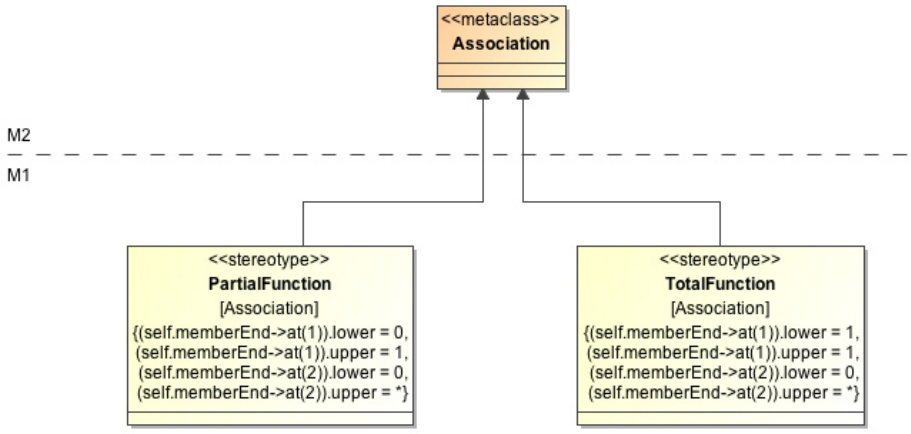


Abbildung 23: Beispiel eines Z UML-Profils für Assozationen

Die Multiplizitäten, die für die Funktionstypen im letzten Abschnitt hergeleitet wurden, müssen nun für die jeweiligen Stereotypen über OCL-Constraints vordefiniert werden. Um mit OCL jedoch die richtigen Attribute zu setzen muss zuerst anhand der UML Infrastructure (in den Abbildungen 24 und 25) hergeleitet werden, wie die Multiplizitäten der Assoziationen im UML Standard-Profil umgesetzt sind: Die Klasse *Assoziation* assoziiert die Klasse *Property* mit dem geordneten (*ordered*) Attribut *memberEnd* und der Multiplizität 2..*. Dieses Attribut stellt die beiden Beziehungsenden (mindestens 2) einer Assoziation mit einem *Property* dar. Jedes *Property* erbt wiederum von der K*lasse MuliplicityE*lement und hat dadurch die beiden Attribute *lower* und *upper*, welche die unter und obere Grenze einer Multiplizität angeben. So entspricht beispielsweise ein *Property* mit *lower* = 1 und *upper* = 2 einer Beziehung von „*1..2*“.

Da eine *Association* zwei geordnete Attribute *memberEnd* hat, kann mit OCL über die *at*-Funktion auf die jeweiligen Instanzen zugegriffen werden. Diese ist laut der OCL-Spezifikation [OCL] wie folgt definiert und zeigt dass der Index der Funktion mit 1 startet.

Die UML Infrastructure Specification macht zu Association folgende Anmerkungen. Darunter auch eine Beschreibung für *Member End*

Associations

- memberEnd : Property [2..*]
  Each end represents participation of instances of the classifier connected to the end in links of the association. This is an ordered association. Subsets Namespace::member.
- ownedEnd : Property [*]
  The ends that are owned by the association itself. This is an ordered association. Subsets Association::memberEnd, Classifier::feature, and Namespace::ownedMember.
- / endType: Type [1..*]
  References the classifiers that are used as types of the ends of the association.
- navigableOwnedEnd : Property [*]
  The navigable ends that are owned by the association itself. Subsets Association.ownedEnd.

Die OCL-Constraints der Stereotypen von Association auf M1 sehen damit exemplarisch für eine partielle Funktion wie folgt aus:

```
{(self.memberEnd->at(1)).lower = 0}
{(self.memberEnd->at(1)).upper = 1}

{(self.memberEnd->at(2)).lower = 0}
{(self.memberEnd->at(2)).upper = *}
```

Die Constraints für alle anderen Funktionen werden analog zu den Multiplizitäten der Tabelle 1 gebildet werden.

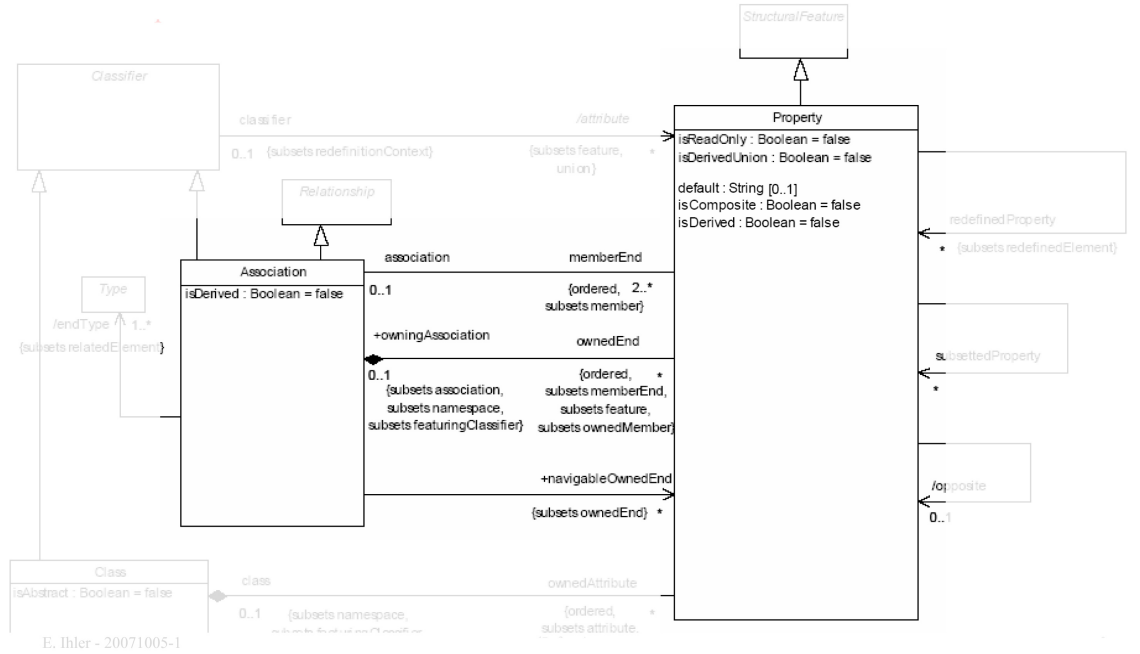


Abbildung 24: InfrastructureLibrary::Core::Basic:: - Class Diagram (S.6)*

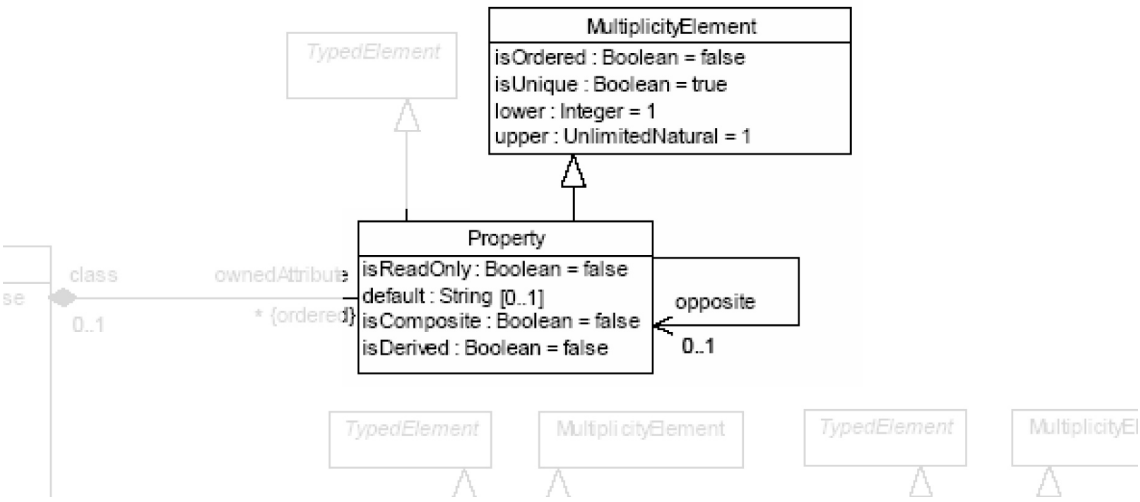


Abbildung 25: InfrastructureLibrary::Core::Contructs:: - Class Diagram (S.8)*

6. Metamodellierung

6.1. Unterschiede zwischen UML-Profilen und MOF-Metamodellierung

Bei der Metamodellierung gibt es 4 Modellierungsebenen: Die oberste Ebene, M3, beschreibt die Sprache für Metamodelle. Das schwergewichtige Meta-Object Framework (MOF) ist ein Beispiel für ein Meta-Metamodell. Die M2 Ebene definiert eine Sprache für spezielle Metamodelle, wie beispielsweise das UML-Metamodel. Die M1 Ebene repräsentiert das Modell selbst. Unsere Modellierung des Getränkeautomaten befindet sich auf M1. M0 ist der Bereich der individuellen Objekte und Instanzen zur Laufzeit – sprich die realen Objekte.

UML Profile erweitern das bestehende Metamodell auf M1 Ebene, sehen aber von der Handhabung so aus als wären sie auf M2 Ebene realisiert. Ein UML-Profil könnte daher auch als „Erweiterung“ beschrieben werden, im Gegensatz zu richtigen Metamodellierung mit dem Meta-Object Framework. Mit dem MOF lassen sich eigene Metamodelle zu UML erstellen oder bestehende „hart“ abändern. Diese Modellierung befindet sich auf M2 und bietet mehr Möglichkeiten als ein UML-Profil. Die Folgende Abbildung zeigt wie ein UML-Profil modelliert werden würden.

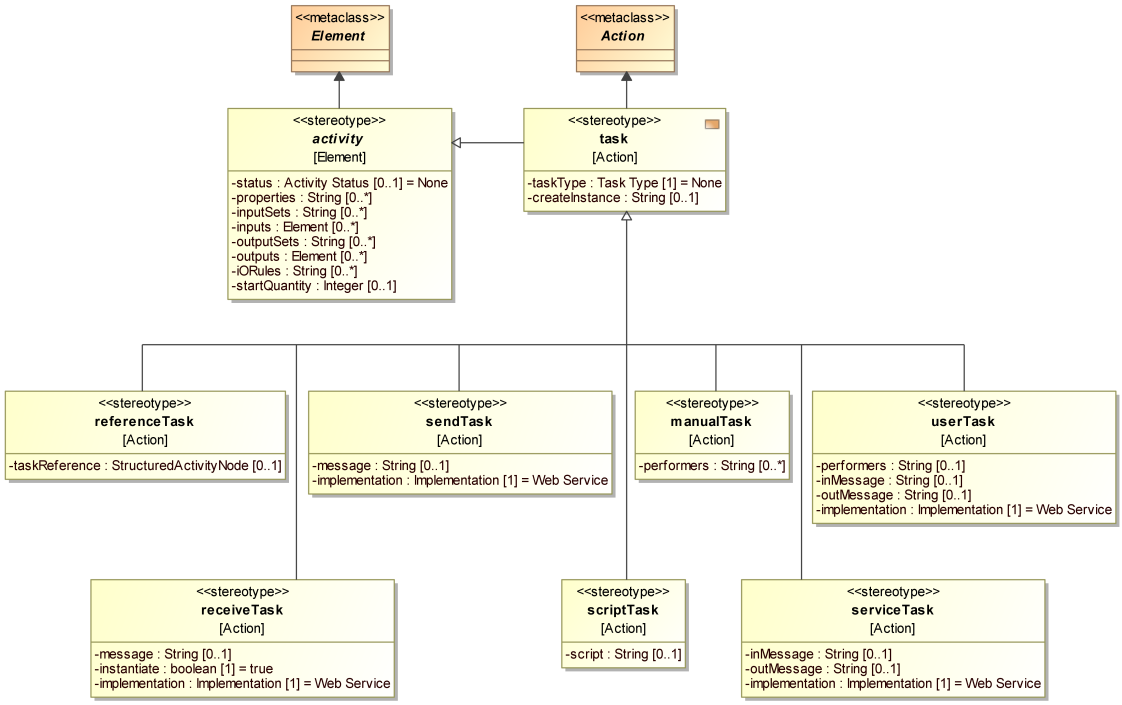


Abbildung 26: Schematische Darstellung: UML-Profile

6.2. Zusammenhang UML-Activities, BPMN, EPK und EEPK

6.2.1. BPMN

Die Business Process Modeling Notation (BPMN) ist eine grafische Notation um Geschäftsprozesse oder Workflows zu beschreiben und darzustellen. Ziel von BPMN ist es Geschäftsprozesse für technische Anwender und nicht-technische Anwender zugleich zu beschreiben.

6.2.2. EPK

Die ereignisgesteuerte Prozesskette (EPK) ist ein Modell zur Darstellung von Geschäftsprozessen. EPK stellt Arbeitsprozesse in einer Modellierungssprache grafisch mit Syntaxregeln dar.

6.2.3. eEPK

Die erweiterte ereignisgesteuerte Prozesskette (eEPK) stellt die erweiterte Form der EPK dar. Die in der EPK dargestellten logischen Abläufe eines Geschäftsprozesses werden anhand der eEPK um die Elemente der Organisations-, Daten und Leistungsmodellierung erweitert. So kann beispielsweise jede Funktion zusätzlich mit einem Informationsobjekt verbunden werden, aus dem Informationen geladen oder in das Informationen gespeichert werden.

6.2.4. UML Activities

UML Aktivitäten (Activities) sind Modellelement in der UML. Sie beschreiben das Verhalten eines Systems. Eine Aktivität besteht aus Aktionen, die einer konkreten Aufgabe entsprechen, die abgearbeitet werden muss um eine Funktion zu erreichen. Eine Aktivität kann sein:

- ein Teil eines Geschäftsprozesses
- ein gesamter Geschäftsprozess

6.3. Gemeinsamkeiten und Unterschiede

Grundsätzlich werden alle vier aufgeführten Beschreibungssprachen dazu verwendet um Geschäftsprozesse zu beschreiben. Dabei steht der Fokus auf der leichten Verständlichkeit. Auch Nicht-Entwickler (z.B. Wissensträger) sollen in der Lage sein ihre Geschäftsprozesse und Abläufe grafisch darzustellen. Um die Gemeinsamkeiten und Unterschiede zu verdeutlichen, wird im Folgenden ein simpler Geschäftsprozess eines Bestellvorgangs in BPMN, EPK, eEPK und UML (Aktivitiätsdiagramm) modelliert.

6.3.1. BPMN

BPMN spezifiiziert Tasks, die mit UML-Aktionen verglichen werden können. Die Tasks im Beispielprozess der Bestellung sind Bestellung erfassen, liefern und nachbestellen. Die Fallunterscheidung ob bestellte Artikel vorhanden oder nicht vorhanden sind, wird in BMPN über ein Gateway (OR) mit anschließenden Intermedia Event Nodes abgebildet. Diese Kombination ist im UML-Aktivitätsdiagramm vergleichbar mit einer Decision Node und den entsprechenden Guard-Conditions. Activity End und Activity Final entsprechen in BPMN einer Start Event Node bzw. einer End Event Node. Input Objekte oder entstehende Artefakte werden in BPMN mit Data-Objects dargestellt. Je nach Art der Assoziation zu einem Task sind diese Input oder Output. Diese Data-Objects sind in einem UML Aktivitätsdiagramm vergleichbar mit einer Object-Node.

Die Unterschiede zur UML zeigen sich zum einen in der Visualisierung des Prozesses und in der Granularität der Modellierung, wie beispielsweise bei Gateways. BPMN unterscheidet unter anderem XOR, OR, AND Verzweigungen. Weitere Unterschiede zeigen sich nicht direkt in der Modellierung des Beispielprozesses. Schaut man sich allerdings komplexere Beispiele an, wird deutlich, dass BPMN eine feinere Modellierung zulässt als ein UML-Aktivitätsdiagramm. BPMN umfasst unter anderem verschieden Flow Objekte (Task, Subprocesses), verschieden Gateways (AND, OR, XOR, Event-based), verschiedene Events (Start, End, Intermediate, Start-Message, Intermediate-Timer, End-Exception), verschiedene Flows (Sequence Flows, Conditional Flows, Default Flows) und Swimmlanes.

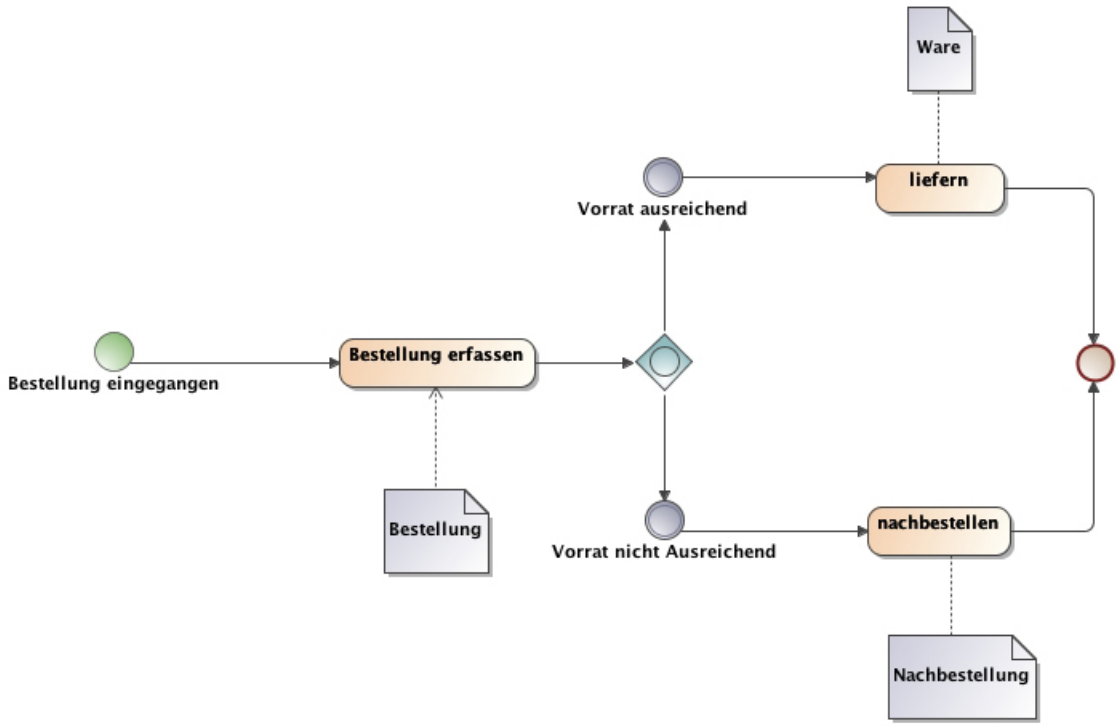


Abbildung 27:BPMN Modell einer Bestellung

6.3.2. EPK

Im EPK-Modell werden Ereignisse in 6-eckigen roten Boxen dargestellt, Funktionen des Geschäftsprozesses werden in abgerundeten grünen Boxen beschrieben. Fallunterscheidungen werden durch Konnektoren dargestellt, die wie bei BPMN AND, OR und XOR den Steuerfluss entsprechend weitergeben. Im Vergleich zu BPMN umfasst EPK jedoch keine Input- und Output-Objekte, keine erweiterten Ereignisse und keine Swimmlanes bzw. Akteure.

Betrachtet man das EKP Modell im Vergleich zu einem UML Aktivitätsdiagramm (Abbildung 30) oder zu einem BPMN Modell (Abbildung 27) fällt auf, dass die beide Diagramme sich in der visuellen Form sehr ähnlich sind. Die Funktionen in EPK korrespondieren im Beispiel direkt mit den UML-Aktivitäten. Im Gegensatz zu den EPK-Ereignissen. Diese können auf verschiedene Weise in die UML überführt werden. Bei Verzweigungen können diese als Guard-Condtition in UML umgesetzt werden, wie hier im Beispiel der Ereignisse „Vorrat ausreichend“ und „Vorrat nicht ausreichend“.

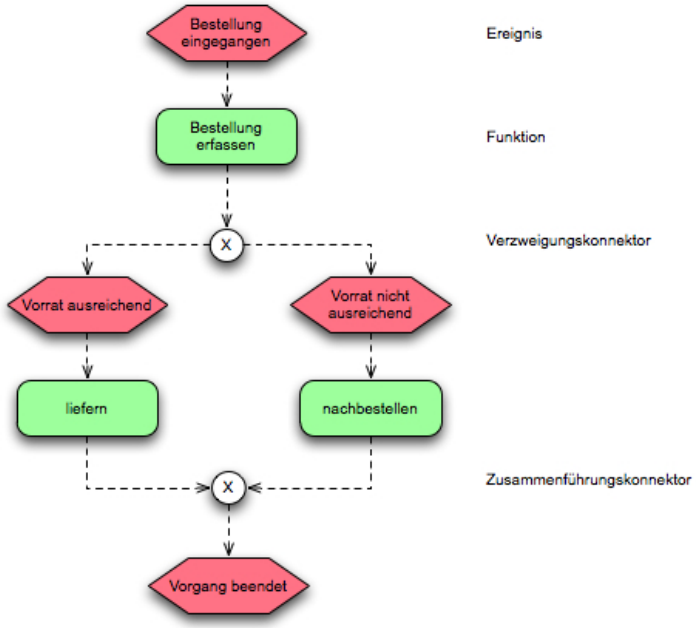


Abbildung 28: EPK Modell einer Bestellung

6.3.3. eEPK

Die erweiterte EPK Notation ergänzt EPK um Informationsobjekte, Datenobjekte und Organisationseinheiten. Damit lassen sich zu den Abläufen der Geschäftsprozesse die beteiligten Akteure und die entstehenden oder einwirkenden Datenobjekte (Artefakte) modellieren. Der Beispielprozess einer Bestellung in eEPK könnte wie folgt modelliert werden (siehe Abbildung 29). Die Ereignisse, Funktionen und Konnektoren bleiben unverändert. Hinzugekommen sind in diesem Modell die Akteuere (Gelb) die mit einer Funktion verbunden sind. Diese Akteuere (Kunde, Mitarbeiter) interagieren mit der Funktion: Ein Kunde gibt die Bestellung auf, ein Mitarbeiter nimmt die Bestellung entgegen.

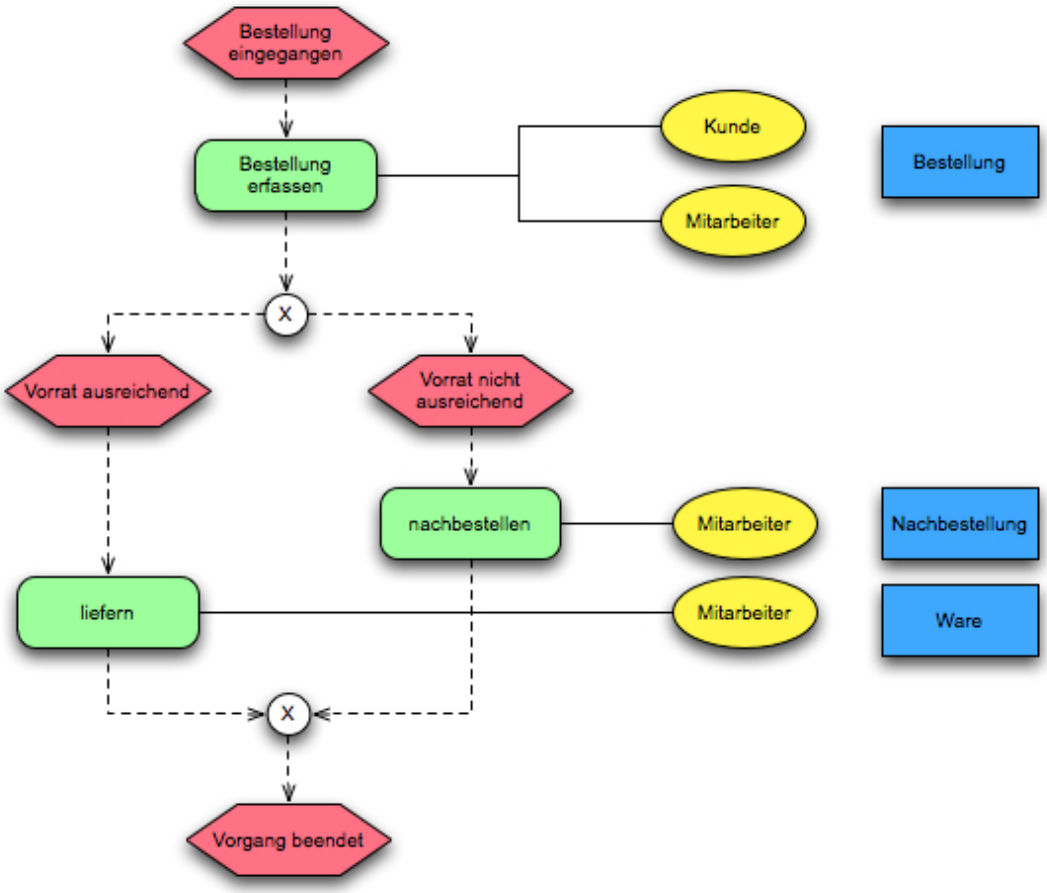


Abbildung 29: eEPK-Modell einer Bestellung

6.3.4. UML

Der gleiche Beispielprozess kann in UML auf mehrere Arten umgesetzt werden, die in den Abbildungen 30 und 31 dargestellt werden. Das erste Aktivitätsdiagramm bildet die Aufgaben als UML-Aktionen ab, die Fallunterscheidung ob der Vorrat ausreichen oder nicht ausreichend ist, wird als Guard-Condition realisiert. Die Input und Output-Objekte werden hier als Object-Node dargestellt und referenzieren Klassen (z.B. aus einem Klassendiagramm).

Der gleiche Geschäftsprozess lässt sich in UML auch anders gestalten (siehe Abbildung 29), wobei diese Umsetzung eher untypisch ist. Im Vergleich zum ersten Modell werden die EPK Events in diesem Aktivitätsdiagramm in UML Event Actions umgesetzt. Die Entscheidungen ob nachbestellt oder geliefert wird, werden hier mittels Forks und Joins realisiert: Wird das Event „Vorraus ausreichend“ getriggert, geht der Kontrollfluss zu liefern über.

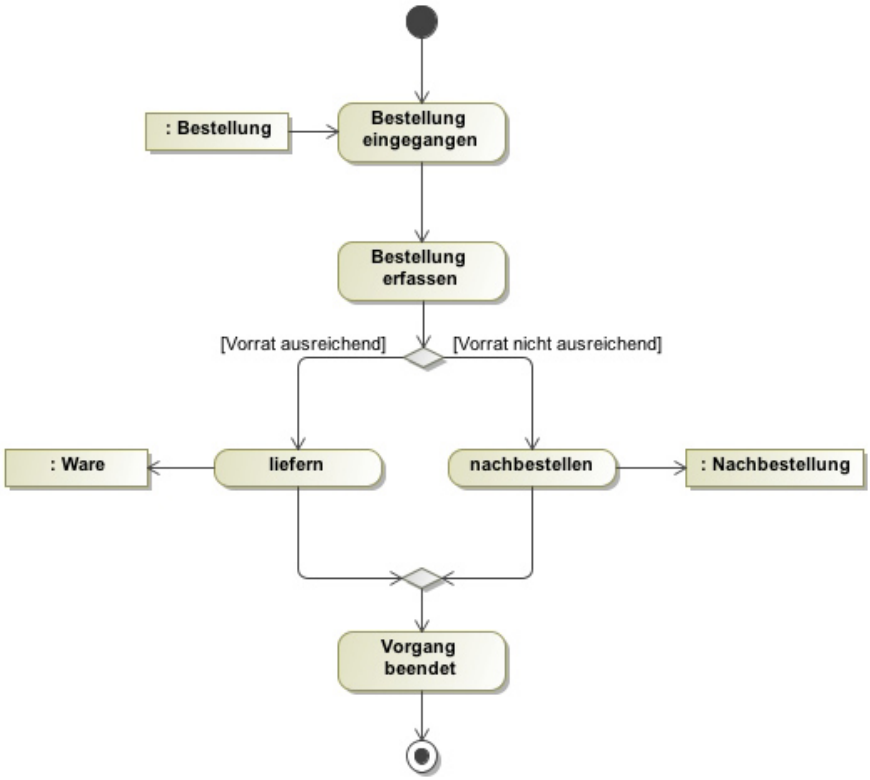


Abbildung 30: UML-Aktivitätsdiagramm einer Bestellung (2)

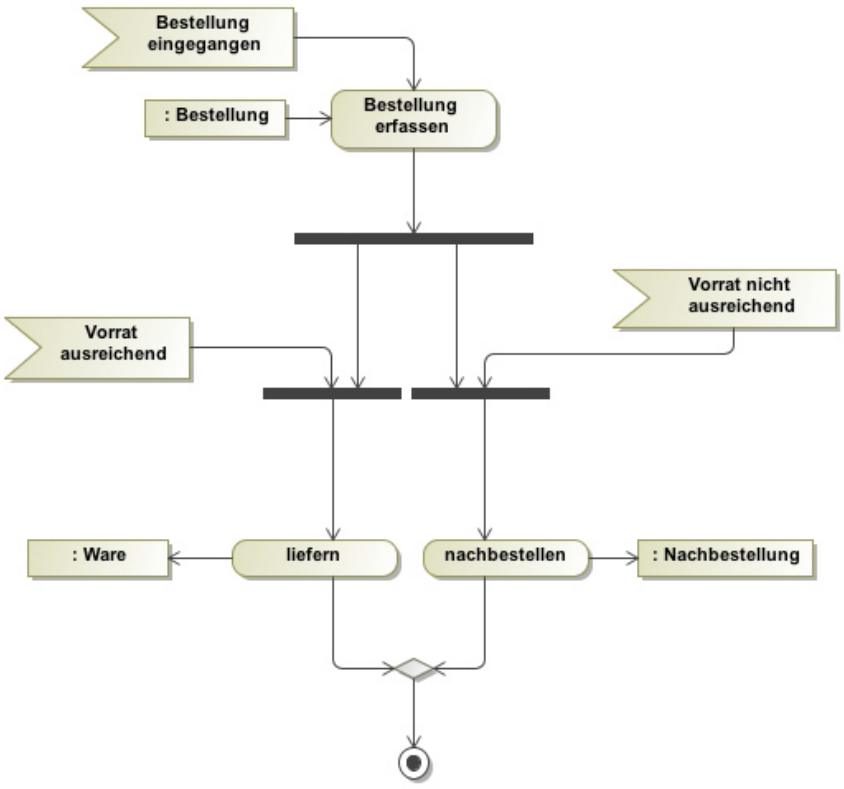


Abbildung 31: UML-Aktivitätsdiagramm einer Bestellung (1)

6.3.5. Vergleich

Die folgende Tabelle zeigt zur Vervollständigung einen Vergleich der verschiedenen Begrifflichkeiten in den unterschiedlichen Business-Modelling Modellen. Sie verdeutlicht nochmals, dass alle der vier Prozessmodell Darstellungen die gleichen Elemente – zwar mit einer anderen grafischen Umsetzung und anderen Bezeichnungen – verwenden.

UML	**EPK**	**EEPK**	**BPMN**
Aktion	Funktion	Funktion	Task
Aktion / Guard-Condition	Ereignis	Ereignis	Intermedia Event
Desicion- / Merge-Node	Konnektor (AND, OR, XOR, Verzweigung)	Konnektor (AND, OR, XOR, Verzweigung)	Gateway (AND, OR, XOR, Event)
Object Node	-	Informationsobjekt/ Datenobjekt	Data-Objects

Zusammenfassung

Diese Arbeit beschäftigte sich mit der Softwaremodellierung in UML und der Modellierung eines Getränkeautomaten anhand einer textuellen Beschreibung.

Zu beginn wurde der Modellbegriffs in der Informatik – insbesondere der Softwareentwicklung – diskutiert. Diese Beschreibung wurde im dritten Kapitel anhand eines Klassendiagramms, eines Aktivitätsdiagramms und eines Zustandsdiagramms dargestellt. Dabei wurden mögliche Alternativen aufgezeigt, die verdeutlichen, dass es in der Softwaremodellierung - mit dem Fokus auf die UML - verschiedene Möglichkeiten gibt, um den Getränkeautomaten und dessen Struktur und Verhalten abzubilden.

Das vierte Kapitel zeigte den Vergleich vom UML-Modell des Automaten dessen Modellierung in Z. Dazu wurden exemplarisch Teile aus der Z-Modellierung in den in UML entsprechenden Alternativen dargestellt. In Kapitel fünf wurde der Einsatz von OCL-Constraints des UML-Modells aufgezeigt, beschrieben und Alternativen diskutiert. Zudem zeigt das Kapitel anhand des Beispiels der Z-Funktionstypen, wie mithilfe von OCL-Constraints ein Z-Profil vom allgemeinen Z-Modell erstellt werden kann.

Das letzte Kapitel beschäftigt sich mit der Metamodellierung und verdeutlicht den Unterschied zwischen UML-Profilen und MOF-Metamodelierung. Des weiteren zeigt das Kapitel den Zusammenhang zwischen den verschiedenen Business-Process-Modelling Darstellungen BPMN, EPK, EEPK und UML-Activities.

Quellenverzeichnis

[Kastens 2001] Prof. Dr. Uwe Kastens 2001
Folien - Vorlesung Modellierung WS 2001/2002
Universität Paderborn
http://ag-kastens.uni-paderborn.de/lehre/material/model2001/folien/inhalt.html

[Glinz 2005] Martin Glinz
Folien Vorlesung Informatik für Ökonomen II:
Modellierung von Informatiksystemen WS 2005/06
Universität Zürich
http://www.ifi.uzh.ch/rerg/courses/ws0506/inf_oec_ii/

[Fowler 2004] Martin Fowler
UML Konzentriert, 3. Auflage
Addison Wesley, ISBN: 3-8273-2126-3

[Oesterreich 2001] Dipl.-Ing. Bernd Oesterreich
Objektorientierte Softwareentwicklung
Oldenbourg Verlag, ISBN: 3-486-255732-8

[BPMN08] Business Process Modeling Notation V1.1
2008, OMG
http://www.bpmn.org/Documents/BPMN%201-1%20Specification.pdf

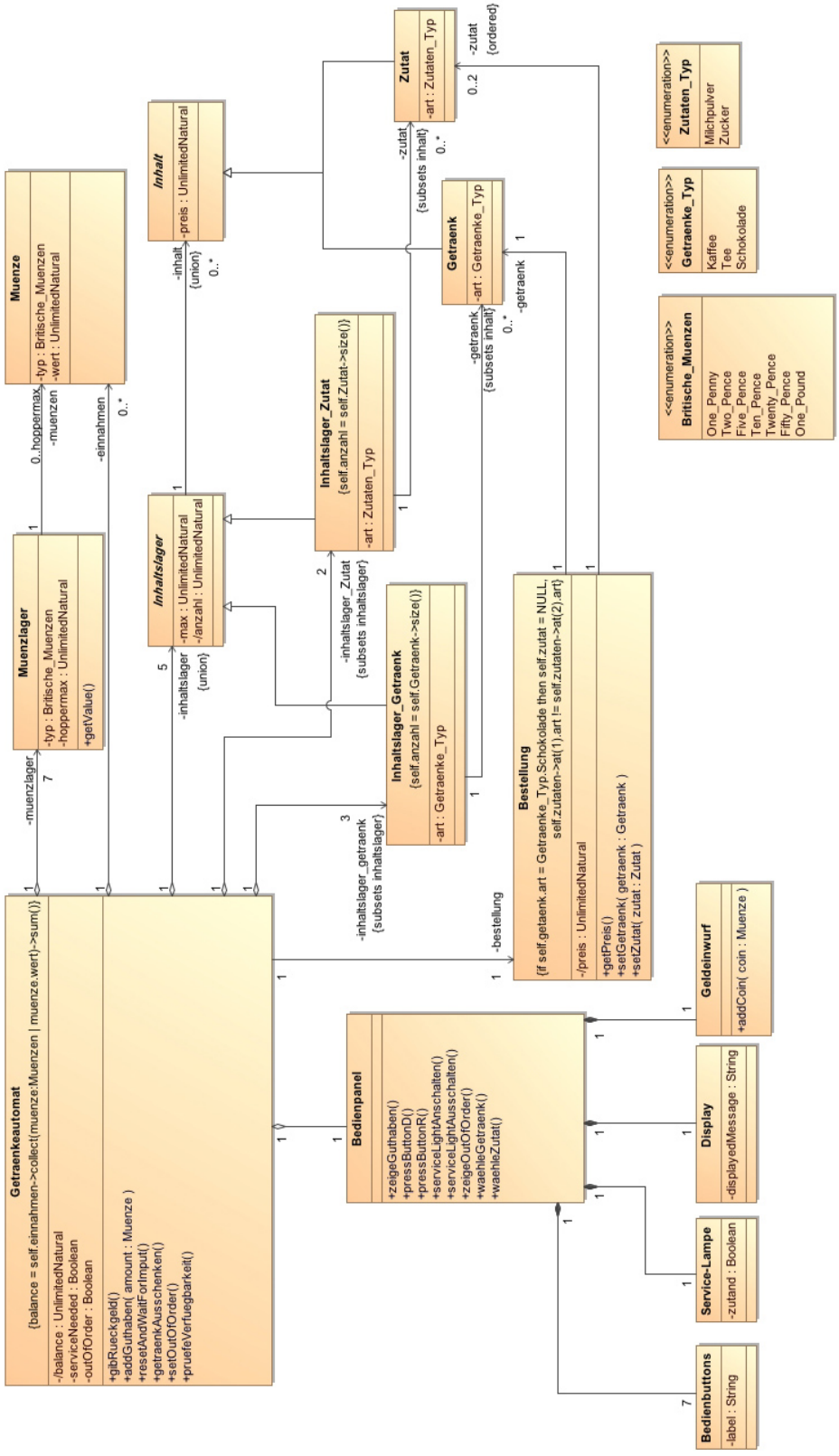
[BPMN] Process Modeling Notations and Workflow Patterns
Stephan A. White, IBM Corp. United States
www.bpmn.org/Documents/Notations%20and%20Workflow%20Patterns.pdf

[UMLIF] UML Infrastructure Library

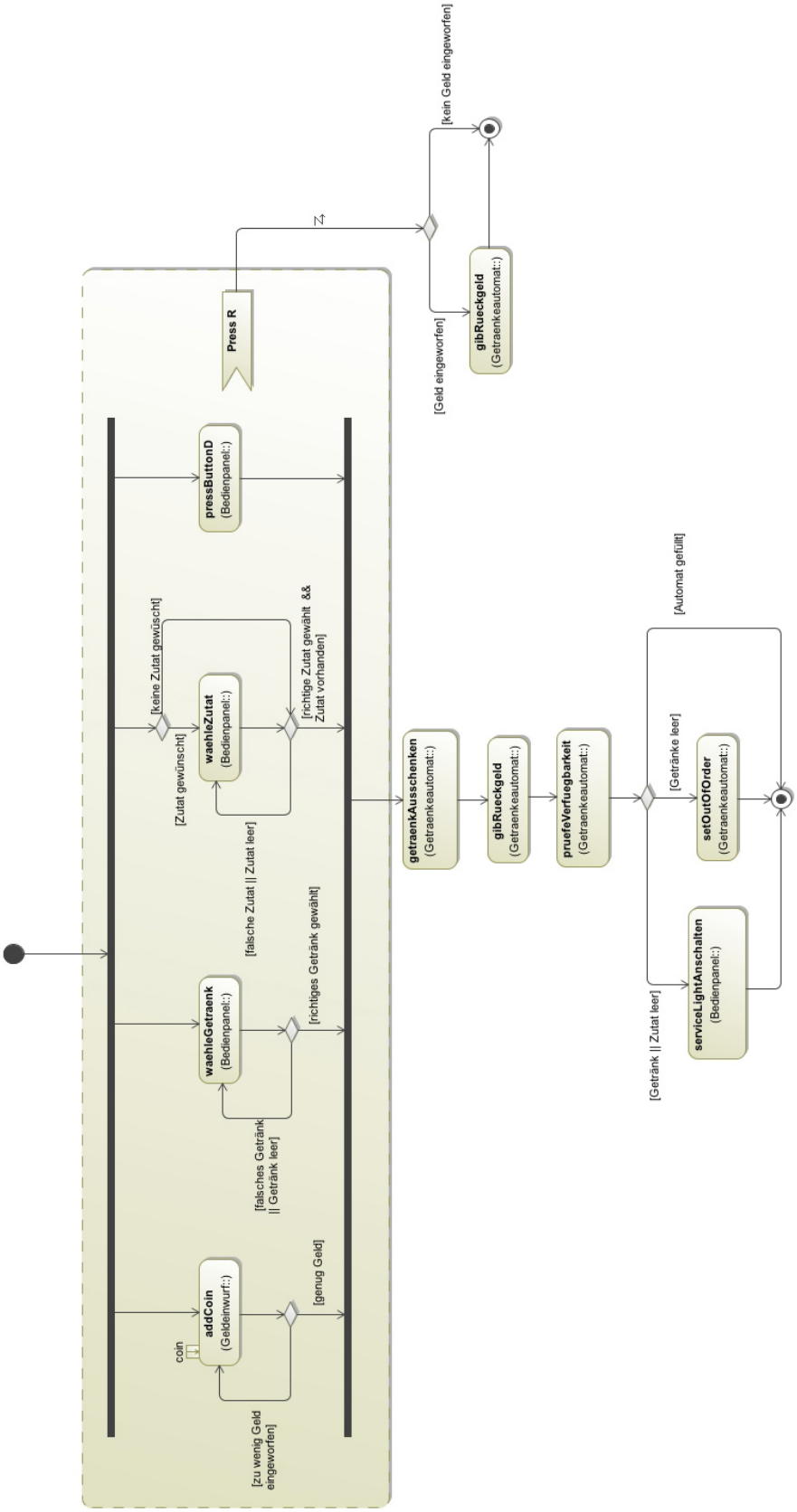
[OCL] Object Constraint Language Formal Specification
2001, OMG

Anhang

Klassendiagramm



Aktivitätsdiagramm



Zustandsdiagramm