Entwurf von Informationssystemen

Cocktaildatenbank

Jannis Kranz [redacted]
Julian Tietje [redacted]
Tobias van Treeck [redacted]

9. Oktober 2013

Inhaltsverzeichnis

Kapitel 1

Einleitung

Im Rahmen der Veranstaltung *Entwurf von Informationssystemen*, wird im Folgenden ein Informationssystem einer Cocktail-Bar beschrieben. Mit Hilfe von *USE* werden in *OCL* ein Klassendiagramm sowie Pre- und Postconditions und zusätzlich Invarianten definiert. Mit der Programmiersprache *SOIL* können Anfragen an das System gestellt oder konkrete Instanzen unseres Modells erzeugt werden. Diese können mittels USE auf ihre Korrektheit bezüglich des in OCL definierten Modells überprüft und in einem Objektdiagramm dargestellt werden.

Kapitel 2

Systembeschreibung

Bei unserer Cocktaildatenbank handelt es sich um ein Informationssystem für die Gastronomie, speziell für den Einsatz in Cocktailbars. Mit dem System ist es möglich, Angebote und Menükarten zu organisieren, Bestellungen entgegen zu nehmen, zu stornieren, Abrechnungen zu machen, Lagerbestände zu verwalten und einzusehen was mit den vorhanden Zutaten noch zubereitet werden kann. Obwohl wir bei der Entwicklung dieses Systems den Hauptaugenmerk auf den Einsatz für Cocktails gelegt haben, ist das System sehr generisch aufgebaut und dadurch sehr universell einsetzbar. Es könnte somit auch Problemlos für andere gastronomische Produkte wie z.B. Speisen verwendet werden. Das System ist also nicht nur auf Cocktailbars beschränkt sondern könnte auch in Restaurants oder Bistros Anwendung finden.

Der Gegenstand in dem sich in unserem System alles dreht ist der Artikel (also bspw. ein Cocktail). Dieser setzt sich aus Zutaten, die in Essbares und Trinkbares unterteilt sind, und sogenannten Extras zusammen. Zu den Extras gehören Hilfsmittel zur Zubereitung, wie Limettenpressen, Crushed-Ice, Gläser und sonstige Dekorationen. Lebensmittel die hauptsächlich der Dekoration dienen, wie z.B. eine einzelne Limettenscheibe oder ein Zuckerrand, werden ebenfalls als Extras geführt. Jeder Artikel ist in eine von drei Komplexitäts- oder Schwierigkeitsstufen eingeteilt, was gewährleisten soll, dass jeder Barkeeper/Koch genau die Artikel zugeteilt bekommt, die seinen Fähigkeiten am ehesten entsprechen.

Jeder Artikel wird der Kundschaft dann als ein konkretes Angebot offeriert, welche in Menükarten organisiert werden können. Über diese Angebote können dann Artikel bestellt werden. Eine Bestellung kann dabei fünf verschiedene Zustände wie z.B. „bestellt“, „serviert“ oder „bezahlt“ annehmen. Dies ermöglicht den Angestellten eine sehr gute Übersicht über die Bestellungen und verhindert bspw., dass ein Artikel doppelt abgerechnet wird. Jede Bestellung wird dabei von unserem System einzeln erfasst und erst beim finalen Bezahlvorgang auf einer gemeinsamen Rechnung zusammenfasst und aufsummiert.

Kapitel 3

Klassendiagramm

3.1 Klassendiagramm

Unser Informationssystem repräsentiert eine Cocktail-Bar, dessen Klassendiagramm in Abbildung 3.1 zu sehen ist. Die Klasse *Item* (*class* Item, Kapitel 3.2.6) symbolisiert einen Artikel in unserer Bar und bildet die Kernklasse unseres Systems. Ein Artikel kann mehrere Formen annehmen: Er kann zum einen ein Cocktail (*class* Cocktail, Kapitel 3.2.13) werden, muss dies aber nicht zwangsläufig. Er könnte auch ein einfaches Getränk sein oder auch eine Speise. Egal was der Artikel nun ist, er besteht immer aus Zutaten, die als Klasse *Ingredient* (*class* Ingredient, Kapitel 3.2.15) dargestellt werden. Diese Zutaten können entweder etwas Essbares (*class* Food, Kapitel 3.2.16) oder etwas Trinkbares (*class* Liquid, Kapitel 3.2.17) sein. Trinkbare Zutaten (Flüssigkeiten) sind zudem in alkoholische (*class* Alcohol, Kapitel 3.2.19) und nicht nicht-alkoholische (*class* Juice, Kapitel 3.2.18) Flüssigkeiten unterteilt. Alkoholische Flüssigkeiten können dabei noch einer Art oder Gruppe von Spirituosen zugeordnet werden (*association* Spirit, Kapitel 3.3.9). Bacardi gehört beispielsweise zur Gruppe Rum.

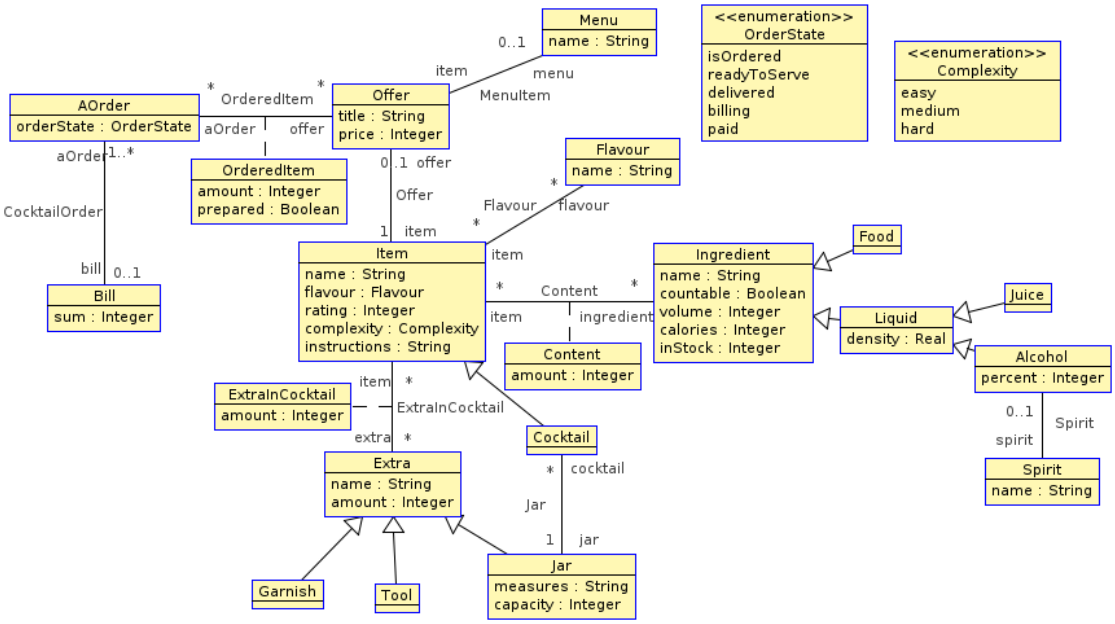


Abbildung 3.1: Das Klassendiagramm unseres Informationssystems

Je nach dem aus wie vielen Zutaten ein Artikel besteht hat er eine andere Komlexitäts- oder Schwierigkeitsstufe, die in der Aufzählungsklasse *Complexity* (*enumeration class* Complexity, Kapitel 3.2.22) festgehalten wird. Die Komplexität eines Artikels hängt jedoch nicht

ausschließlich von der Anzahl seiner Zutaten ab, sondern hat noch andere Einflüsse. Ein Artikel kann außerdem verschiedene Geschmäcker annehmen, die in der Klasse *Flavour* (*class* Flavour, Kapitel 3.2.7) festgehalten werden. Hierbei ist man nicht auf eine Geschmacksrichtung beschränkt, sondern kann auch mehrere angeben, wie beispielsweise süß und sauer.

Dekorationen für (*class* Garnish, Kapitel 3.2.10) einen Artikel (Cocktail), wie eine Limettenscheibe oder ein Zuckerrand, zählen nicht zu den Zutaten. Diese werden als Extras in der gleichnamigen Klasse geführt (*class* Extra, Kapitel 3.2.9). Neben solcher Dekorationen fasst die Klasse *Extra* auch noch Werkzeuge (*class* Tool, Kapitel 3.2.11) die beim Zubereiten eines Artikels benötigt werden und Gefäße (*class* Jar, Kapitel 3.2.12) in denen Artikel serviert werden.

Um Artikel bestellen zu können, werden diese als konkrete Angebote (*class* Offer, Kapitel 3.2.4) offeriert, die auf einer Menükarte (*class* Menu, Kapitel 3.2.5) geführt sind. Alle Bestellungen werden von der Klasse *Aorder* (*class* AOrder, Kapitel 3.2.2) repräsentiert und haben fünf verschiedene Bestell-Stati (wie z.B. „serviert“ oder „bezahlt“) die in der Aufzählungsklasse *OrderState* (*enumeration class* OrderState, Kapitel 3.2.21) definiert sind. Um Bestellungen schlussendlich zu bezahlen, dient die Klasse *Bill* (*class* Bill, Kapitel 3.2.1), welche eine oder mehrere Bestellungen zusammenfasst und in Rechnung stellt.

3.2 Klassen

Im Folgenden haben wir alle Klassen mit ihren Attributen, Operationen, Vor- und Nachbedingungen und Invarianten dokumentiert.

3.2.1 *class* Bill

Die Klasse *Bill* repräsentiert eine Rechnung von einer (oder mehrere) Bestellung(en) (*class* AOrder, Kapitel 3.2.2). Das Attribut *sum* beinhaltet dabei die Summe, die am Ende explizit berechnet werden muss, sich also nicht permanent automatisch erhöht.

Attribute

Name	Typ	Beschreibung
sum	Integer	Die Summe einer Rechnung.

Operationen

getBill()

Berechnet die Summe aller zugehörigen Bestellungen.

Vor- & Nachbedingungen

Name	Typ	Beschreibung
allDelivered	pre	Überprüft ob alle Bestellungen dieser Rechnung bereits ausgeliefert wurden.

Deklaration:

```
1  getBill():Tuple(list:Bag(Tuple(name:String, description:String, ↙
       ↳count:Integer, sum:Integer)), wholeSum:Integer)
2          begin
3                  declare tempSet:Set(Tuple(name:String, description:String, ↙
                       ↳count:Integer, sum:Integer)), ↙
                       ↳resultSet:Bag(Tuple(name:String, description:String, ↙
                       ↳count:Integer, sum:Integer)), ↙
                       ↳waste:Bag(Tuple(name:String, description:String, ↙
                       ↳count:Integer, sum:Integer));
4                  self.sum := ↙
                       ↳self.aOrder.collect(a:AOrder|a.billOrder())->sum();
5                  for o in self.aOrder do
6                          o.orderState := #billing;
7                          tempSet := o.orderedItem->collectNested($oI : ↙
                               ↳OrderedItem|
8                                  Tuple{name=$oI.offer.title, description = ↙
                                       ↳$oI.offer.item.name, count=$oI.amount,
9                                          sum=$oI.amount * ↙
                                               ↳$oI.offer.price})->asSet();
10                         if not tempSet->oclIsUndefined() then
11                             if resultSet->oclIsUndefined() then
12                                 resultSet := tempSet->asBag();
13                             else
```

```
14                                      resultSet := ↙
                                            ↳resultSet->union(tempSet->asBag());
15                                  end;
16                              end
17                  end;
18                  result := Tuple{list=resultSet, wholeSum=self.sum};
19          end
```

Quellcode 3.1: Quellcode von Bill::getBill()

payBill()

Bezahlt eine zuvor erstellte Rechnung.

Vor- & Nachbedingungen

Name	Typ	Beschreibung
allBilled	pre	Überprüft ob alle Bestellungen dieser Rechnung bereits in Rechnung gestellt wurden.

Deklaration:

```
1 payBill()
2         begin
3                 for o in self.aOrder do
4                         o.orderState := #paid;
5                 end
6         end
7         pre allBilled:
8                 self.aOrder.forAll(a|a.orderState = #billing)
```

Quellcode 3.2: Quellcode von Bill::payBill()

addOrder(order:AOrder)

Fügt dieser Rechnung eine Bestellung hinzu.

Vor- & Nachbedingungen

Name	Typ	Beschreibung
hasBeenDelivered	pre	Überprüft ob diese Bestellungen bereits bestellt wurde.

Deklaration:

```
1 addOrder(order:AOrder)
2         begin
3                 insert (order,self) into BilledOrder;
4         end
5         pre hasBeenDelivered:
6                 order.orderState = #delivered
```

Quellcode 3.3: Quellcode von Bill::addOrder()

removeOrder(order:AOrder)

Löscht eine Bestellung aus dieser Rechnung.

Vor- & Nachbedingungen

Name	Typ	Beschreibung
isOnBill	pre	Überprüft ob diese Bestellung auf dieser Rechnung ist.
isOffBill	post	Überprüft ob diese Bestellungen von dieser Rechnung entfernt wurde.

Deklaration:

```
1 removeOrder(order:AOrder)
2         begin
3                 delete (order,self) from BilledOrder;
4         end
5         pre isOnBill:
6                 self.aOrder->includes(order)
7         post isOffBill:
8                 self.aOrder->excludes(order)
```

Quellcode 3.4: Quellcode von Bill::removeOrder()

Invarianten

sumNotNegative

Stellt sicher, dass die Summe niemals negativ wird.

Deklaration:

```
1 inv sumNotNegative:
2         self.sum >= 1
```

Quellcode 3.5: Deklaration der Invariante Bill::sumNotNegative

sumIsWholeSum

Stellt sicher, dass die Summe auf der Rechnung auch die Summe der einzelnen Bestellungen ist.

Deklaration:

```
1 inv sumIsWholeSum:
2                 self.sum >= self.aOrder.orderedItem->collectNested($e : ↙
                      ↳OrderedItem | $e.amount * $e.offer.price)->sum()
```

Quellcode 3.6: Deklaration der Invariante Bill::sumIsWholeSum

---

3.2.2 *class* AOrder

Die Klasse *AOrder* repräsentiert eine Bestellung, die aus einem oder mehreren Angeboten bestehen kann. Eine Bestellung kann sich in einem von drei Status (*enumeration class* OrderState, Kapitel 3.2.21) befinden, der jeweils im Attribut *orderState* angegeben wird.

Attribute

Name	Typ	Beschreibung
orderState	OrderState	Der Status (*enumeration class* OrderState, Kapitel 3.2.21) der Bestellung.

Operationen

placeOrder(of:Offer, amnt:Integer)

Fügt eine konkretes Angebot *of* in der Anzahl *amnt* dieser Bestellung hinzu.

Vor- & Nachbedingungen

Name	Typ	Beschreibung
allIngredientsInStock	pre	Überprüft ob vor einer Bestellung auch alle benötigten Zutaten vorrätig sind.
isOrdered	post	Überprüft ob der Status der aktuellen Bestellung nun auf *Bestellt* gesetzt ist.

Deklaration:

```
1  placeOrder(of:Offer, amnt:Integer)
2          begin
3                  declare a:OrderedItem;
4                  if (self.offer->includes(of)) then
5                          a := self.orderedItem.any(o|of = o.offer);
6                          a.amount := a.amount + amnt;
7                  else
8                          a := new OrderedItem between (of, self);
9                          a.amount := amnt;
10                 end;
11                 of.item.decreaseStock(amnt);
12                 self.orderState := #isOrdered;
13         end
14         pre allIngredientsInStock:
15                 of.countAvailOffers()>=amnt
16         post isOrdered:
17                 self.orderState = #isOrdered
```

Quellcode 3.7: Quellcode von AOrder::placeOrder()

getItems():Bag(Item)()

Berechnet alle *Items* die zu dieser Order gehören und gibt diese als *Bag* zurück.

Vor- & Nachbedingungen

Name	Typ	Beschreibung
offerIncluded	pre	Überprüft ob die Offer, die entfernt werden soll auch vorhanden ist.

Deklaration:

```
1 getItems():Bag(Item) =
2         self.orderedItem.collect(offer.item);
```

Quellcode 3.8: Quellcode von AOrder::getItems()

pendingOrders():Set(AOrder)

Sucht alle Bestellungen heraus, die Bestellt, aber noch nicht serviert wurden.

Vor- & Nachbedingungen

Name	Typ	Beschreibung
-	-	-

Deklaration:

```
1 pendingOrders():Set(AOrder)=
2         AOrder.allInstances()->select(a | a.orderState = #isOrdered);
```

Quellcode 3.9: Quellcode von AOrder::pendingOrders()

getAllIngredients():Set(Tuple(ingred:Ingredient,requiredAll:Integer))

Berechnet alle Zutaten, die zur dieser Bestellung gehören und gibt diese als Zutat-Anzahl Tupelliste wieder zurück.

Vor- & Nachbedingungen

Name	Typ	Beschreibung
-	-	-

Deklaration:

```
1 getAllIngredients():Set(Tuple(ingred:Ingredient, requiredAll:Integer))=
2         self.orderedItem->collect($e : OrderedItem ↙
              ↳|($e.offer.item.getAllIngredients($e.amount)))->
3                 iterate(t; acc:Set(Tuple(ingred:Ingredient, ↙
                      ↳requiredAll:Integer))=Set{} |
4                 if acc->exists(t1 | t1.ingred = t.ingred) then
5                         let existingT = acc->any(t1 | t1.ingred = ↙
                              ↳t.ingred) in
6                                 acc->excluding(existingT)->including( ↙
                                      ↳Tuple{ingred=existingT.ingred, ↙
                                      ↳requiredAll = existingT.requiredAll + ↙
                                      ↳t.required})
7                 else
8                         acc->including(Tuple{ingred=t.ingred, ↙
                              ↳requiredAll=t.required})
9                 endif
10                )
```

Quellcode 3.10: Quellcode von AOrder::getAllIngredients()

billOrder():Bag(Integer)

Berechnet die Rechnung.

Vor- & Nachbedingungen

Name	Typ	Beschreibung
-	-	-

Deklaration:

```
billOrder():Bag(Integer)=
        self.orderedItem.collect($e : OrderedItem | $e.amount * ↙
            ↳$e.offer.price)
```

Quellcode 3.11: Quellcode von AOrder::billOrder()

prepareOrders()

Markiert alle Bestellungs als servierfertig.

Vor- & Nachbedingungen

Name	Typ	Beschreibung
-	-	-

Deklaration:

```
prepareOrders()
        begin
                declare orders:Set(AOrder);
                orders := self.pendingOrdersSoil();
                        self.prepareOrder(ord.orderedItem);
                        ord.orderState := #readyToServe;
                end;
        end
```

Quellcode 3.12: Quellcode von AOrder::prepareOrders()

prepareOrder()

Markiert ein einzelnen Artikel einer Bestellung als bestellt.

Vor- & Nachbedingungen

Name	Typ	Beschreibung
-	-	-

Deklaration:

```
prepareOrder(oItems:Set(OrderedItem))
        begin
                for oIt in oItems do
                        if (not oIt.prepared) then
                                oIt.prepared := true;
                        end;
                end;
        end
```

Quellcode 3.13: Quellcode von AOrder::prepareOrder()

getItems():Bag(Item)

Ermittelt alle Artikel dieses Angebots.

Vor- & Nachbedingungen

Name	Typ	Beschreibung
-	-	-

Deklaration:

```
getItems():Bag(Item) =
        self.orderedItem.collect(offer.item);
```

Quellcode 3.14: Quellcode von AOrder::geItems()

removeOffer(of:Offer, amnt:Integer)

Ermittelt alle Artikel dieses Angebots.

Vor- & Nachbedingungen

Name	Typ	Beschreibung
enoughOrderedToBeremoved	pre	Überprüft ob mehr Bestellungen vorhanden sind, als entfernt werden müssen.
offerIncluded	pre	Überprüft ob die zu entfernende Bestellung Teil dieser Bestellunge ist.

Deklaration:

```
removeOffer(of:Offer, amnt:Integer)
        begin
                declare set1:OrderedItem;
                set1 := self.orderedItem.any(o|of = o.offer);
                set1.amount := set1.amount - amnt;
                if(set1.amount < 1) then
                        delete (of, self) from OrderedItem;
                end;
                of.item.increaseStock(amnt);
        end
        pre enoughOrderedToBeremoved:
                self.orderedItem.select(of.title = ↙
                    ↳offer.title).exists(amount >= amnt)
        pre offerIncluded:
                self.offer->includes(of)
```

Quellcode 3.15: Quellcode von AOrder::removeOffer()

removeOrder()

Ermittelt alle Artikel dieses Angebots.

Vor- & Nachbedingungen

Name	Typ	Beschreibung
hasNotBeenPrepared	pre	Überprüft ob die Bestellung noch nicht zubereitet wurde.

Deklaration:

```
removeOrder()
        begin
                for of in self.offer do
                        delete (of, self) from OrderedItem;
                end
        end
        pre hasNotBeenPrepared:
                self.orderState <> #readyToServe;
```

Quellcode 3.16: Quellcode von AOrder::removeOrder()

complain(off:Offer, cnt:Integer)

Falls man sich über eine Bestellung beschweren will, bekommt man die gleiche nocheinmal Gratis.

Vor- & Nachbedingungen

Name	Typ	Beschreibung
hasBeenOrderedCntTimes	pre	Stellt sicher, dass die Artikel über die man sich beschwert auch von einem bestellt wurden.

Deklaration:

```
complain(off:Offer, cnt:Integer)
        begin
                declare minOffer:Offer;
                minOffer := new Offer('minOffer');
                minOffer.init('Complaint Item', 0, off.item);
                self.placeOrder(minOffer, cnt);
                self.orderState := #isOrdered;
        end
        pre hasBeenOrderedCntTimes:
                self.orderedItem.select(o :OrderedItem| off.title = ↙
                    ↪o.offer.title).exists(amount >= cnt)
```

Quellcode 3.17: Quellcode von AOrder::complain()

Invarianten

orderStateReadyToServeImpliesAllPrepared

Wenn eine Bestellung servierfertig ist, müssen alle ihre einzelnen Artikel zubereitet sein.

Deklaration:

```
inv orderStateReadyToServeImpliesAllPrepared:
        self.orderState=#readyToServe implies self.orderedItem.forAll(o | ↙
            ↪o.prepared = true)
```

Quellcode 3.18: Deklaration der Invariante AOrder::orderStateReadyToServeImpliesAllPrepared

orderStatePaidImpExOfBill

Wenn eine Bestellung als bezahlt markiert wurde, muss sie über eine Rechnung verfügen.

Deklaration:

```
inv orderStatePaidImpExOfBill:
        self.orderState=#paid implies self.bill->size() > 0
```

Quellcode 3.19: Deklaration der Invariante AOrder::orderStatePaidImpExOfBill

exOfBillImpOrderStatePaid

Wenn eine Bestellung über eine Rechnung verfügt muss ihr Bestellstatus „bezahlt“ oder „in Rechnung“ sein.

Deklaration:

```
inv exOfBillImpOrderStatePaid:
        (self.bill->size() > 0 implies self.orderState=#paid) or ↙
            ↪(self.bill->size() > 0 implies self.orderState=#billing)
```

Quellcode 3.20: Deklaration der Invariante AOrder::exOfBillImpOrderStatePaid

orderStateImpSizeOfItems

Wenn die Bestellung einen Bestellstatus hat, muss sie mindestens ein Angebot beinhalten.

Deklaration:

```
1 inv orderStateImpSizeOfItems:
2         (self.orderState = #delivered implies self.orderedItem->size() > ↙
              ↘0) or
3         (self.orderState = #paid implies self.orderedItem->size() > 0) or
4         (self.orderState = #billing implies self.orderedItem->size() > 0) or
5         (self.orderState = #readyToServe implies self.orderedItem->size() ↙
              ↘> 0)or
6         (self.orderState = #isOrdered implies self.orderedItem->size() > 0)
```

Quellcode 3.21: Deklaration der Invariante AOrder::orderStateImpSizeOfItems

3.2.3 *association class* OrderedItem

Bei der Klasse *OrderedItem* handelt es sich um eine Assoziationsklasse zwischen einer Bestellung (*class* AOrder, Kapitel 3.2.2) und einem Angebot (*class* Offer, Kapitel 3.2.4), die die Häufigkeit des Vorkommens (konkret die Menge) von einem Angebot angibt.

Attribute

Name	Typ	Beschreibung
amount	Integer	Die Menge eines Angebots in dieser Bestellung.
prepared	Boolean	Signalisiert, ob eine Bestellung fertig zubereitet ist.

Operationen

setPrepared()

Setzt das Attribut *prepared* auf *true*.

Vor- & Nachbedingungen

-

Deklaration:

```
1 setPrepared()
2         begin
3                 self.prepared := true;
4         end
```

Quellcode 3.22: Quellcode von OrderedItem::setPrepared()

Invarianten

amountNotNegative

Stellt sicher, dass die Anzahl (*amount*) niemals negativ wird.

Deklaration:

```
1 inv amountNotNegative:
2         self.amount >= 0
```

Quellcode 3.23: Deklaration der Invariante OrderedItem::amountNotNegative

3.2.4 *class* Offer

Die Klasse *Offer* stellt ein Angebot für ein Artikel (*class* Item, Kapitel 3.2.6) dar.

Attribute

Name	Typ	Beschreibung
title	String	Name des Angebots.
price	Integer	Preis des Angebots.

Operationen

init()

Der Konstruktor der Klasse.

Vor- & Nachbedingungen

Name	Typ	Beschreibung
-	-	-

Deklaration:

```
1 init(title:String,nPrice:Integer, it:Item)
2         begin
3                 self.title:=title;
4                 self.price:=nPrice;
5                 insert (it, self) into Offer;
6         end
```

Quellcode 3.24: Quellcode von Offer::init()

countInStockFactor()

Der Konstruktor der Klasse.

Vor- & Nachbedingungen

-

Deklaration:

```
1 countInStockFactor():Set(Tuple(ingred:Ingredient, requiredAll:Integer, ↙
    ↳inStock:Integer, available:Real))=
2         self.item.getAllIngredients(1)->collectNested($e : Tuple ↙
              ↳(ingred:Ingredient, required:Integer, inStock:Integer, ↙
              ↳available:Real) | Tuple{ingred $e.ingred, ↙
              ↳requiredAll=$e.required, inStock=$e.ingred.inStock, ↙
              ↳available= $e.ingred.inStock / $e.required} )->asSet()
```

Quellcode 3.25: Quellcode von Offer::countInStockFactor()

getIngredientTuples():Set(Tuple(ingred:Ingredient,required:Integer))

Ermittelt alle Zutaten & Mengen und gibt diese als Liste von Tupeln zurück.

Vor- & Nachbedingungen

Name	Typ	Beschreibung
-	-	-

Deklaration:

```
1 getIngredientTuples():Set(Tuple(ingred:Ingredient,required:Integer))=
2         self.item.getAllIngredients(1);
```

Quellcode 3.26: Quellcode von Offer::getIngredientTuples()

isAvail():Boolean

Überprüft ob noch genug Zutaten vorhanden sind um das aktuelle Angebot zubereiten zu können.

Vor- & Nachbedingungen

Name	Typ	Beschreibung
-	-	-

Deklaration:

```
1 isAvail():Boolean =
2         self.item.content.forAll(i, c | i.ingredient.inStock >= c.amount)
```

Quellcode 3.27: Quellcode von Offer::isAvail()

countAvailOffers():Real

Ermittelt, wieviele Angebote mit den im Lager befindlichen Zutaten noch hergestellt werden können.

Vor- & Nachbedingungen

Name	Typ	Beschreibung
-	-	-

Deklaration:

```
1 countAvailOffers():Real=
2         self.countInStockFactor()->collectNested($e : ↙
              ↳Tuple(ingred:Ingredient, requiredAll:Integer, ↙
              ↳inStock:Integer, available:Real) | $e.available)->min()
```

Quellcode 3.28: Quellcode von Offer::countAvailOffers()

Invarianten

priceNotNegative

Stellt sicher, dass der Preis dieses Angebots größer 0 ist.

Deklaration:

```
1 inv priceNotNegative:
2         self.price >= 0
```

Quellcode 3.29: Deklaration der Invariante Offer::priceNotNegative

---

3.2.5 *class* Menu

Die Klasse *Menu* repräsentiert ein Menü, auf dem sich alle oder bestimmte Angebote (*class* Offer, Kapitel 3.2.4) wiederfinden.

Attribute

Name	Typ	Beschreibung
name	String	Der Name des Menüs.

Operationen

init()

Der Konstruktor dieser Klasse

Vor- & Nachbedingungen

-

Deklaration:

```
1 init(title:String,nPrice:Integer)
2         begin
3                 self.name:=title;
4         end
```

Quellcode 3.30: Quellcode von Menu::init()

addOffer()

Fügt diesem Menü ein Angebot hinzu.

Vor- & Nachbedingungen

-

Deklaration:

```
1 addOffer(off:Offer)
2         begin
3                 insert (off, self) into MenuItem;
4         end
```

Quellcode 3.31: Quellcode von Menu::addOffer()

getAvailOffers():Set(Offer)

Ermittelt alle Verfügbaren Angebote, davon ausgehend für welche Angebote noch genügend
Zutaten vorhanden sind.

Vor- & Nachbedingungen

-

Deklaration:

```
1 getAvailOffers():Set(Offer) =
```

```
2         Offer.allInstances()->select( o | o.item.content->forAll(i, c | ↙
              ↳i.ingredient.inStock >= c.amount))
```

Quellcode 3.32: Quellcode von Menu::getAvailOffers()

Invarianten

Diese Klasse verfügt über keine Invarianten.

3.2.6 *class* Item

Die Klasse *Item* ist die Kernklasse unseres Informationssystems. Sie stellt ein Artikel in unserer Bar da, in den meisten Fällen also ein Cocktail (*class* Cocktail, Kapitel 3.2.13). Sie kann jedoch auch ein Essen sein.

Attribute

Name	Typ	Beschreibung
name	String	Der Name des Artikels.
flavour	Flavour	Die Geschmacksrichtung (*class* Flavour, Kapitel 3.2.7) des Artikels.
rating	Integer	Die Bewertung des Artikels.
complexity	Complexity	Die Komplexität (*enumeration class* Complexity, Kapitel 3.2.22) des Artikels, welche angibt, wie schwer die Zubereitung ist.
instructions	String	Das Rezept für die Zubereitung.

Operationen

init()

Der Konstruktor der Klasse.

Vor- & Nachbedingungen

-

Deklaration:

```
1 init(aName:String, flav:Flavour, rate:Integer,complex:Complexity, ↙
    ↳instructs:String)
2         begin
3                 self.name:=aName;
4                 self.flavour:=flav;
5                 self.rating:=rate;
6                 self.complexity:=complex;
7                 self.instructions:=instructs;
8         end
```

Quellcode 3.33: Quellcode von Item:init()

addIngredient(ingred:Ingredient, amount:Integer)

Fügt eine Zutat zu diesem Artikel hinzu und initiiert dazwischen die Assoziationsklasse *Content*.

Vor- & Nachbedingungen

-

Deklaration:

```
1 addIngredient(ingred:Ingredient, amount:Integer)
```

```
2         begin
3                 declare c : Content;
4                 c:= new Content between (self, ingred);
5                 c.amount:= amount;
6         end
```

Quellcode 3.34: Quellcode von Item:addIngredient()

addExtra(extr:Extra)

Fügt dieser Klasse ein Extra hinzu.

Vor- & Nachbedingungen

-

Deklaration:

```
1 addExtra(extr:Extra)
2         begin
3                 insert (self, extr) into ExtraInCocktail;
4         end
```

Quellcode 3.35: Quellcode von Item:addExtra()

increaseStock(amnt:Integer)

Erhöht den Vorrat für diesen Artikel um die angegeben Größe.

Vor- & Nachbedingungen

-

Deklaration:

```
1 increaseStock(amnt:Integer)
2         begin
3                 for theW in self.getAllIngredients(amnt) do
4                     theW.ingred.increaseStock(theW.required)
5                 end
6         end
```

Quellcode 3.36: Quellcode von Item:increaseStock()

decreaseStock(amnt:Integer)

Verringert den Vorrat für diesen Artikel um die angegeben Größe.

Vor- & Nachbedingungen

-

Deklaration:

```
1 decreaseStock(amnt:Integer)
2         begin
3                 for theW in self.getAllIngredients(amnt) do
4                     theW.ingred.decreaseStock(theW.required)
5                 end
6         end
```

Quellcode 3.37: Quellcode von Item:decreaseStock()

getAllIngredients(amount:Integer):Set(Tuple(ingred:Ingredient,required:Integer))=

Ermittelt alle Zutaten und Mengen, die zu diesem Artikel gehören.

Vor- & Nachbedingungen

-

Deklaration:

```
1 getAllIngredients(amount:Integer):Set(Tuple(ingred:Ingredient, ↙
    ↳required:Integer))=
2         self.content->collect($c : Content | Tuple{ingred = $c.ingredient, ↙
              ↳required = amount * $c.amount})->asSet()
```

Quellcode 3.38: Quellcode von Item:getAllIngredients()

Invarianten

ratingNotNegative

Stellt sicher, dass die Bewertung immer größer als Null ist.

Deklaration:

```
1 inv ratingNotNegative:
2         self.rating >= 0
```

Quellcode 3.39: Deklaration der Invariante Item::ratingNotNegative

nameIsUnique

Stellt sicher, dass der Name des Artikels eindeutig ist und nicht mehrfach vorkommt.

Deklaration:

```
1 inv nameIsUnique:
2         Item.allInstances->isUnique(i|i.name)
```

Quellcode 3.40: Deklaration der Invariante Item::nameIsUnique

ItemIngredientsBiggerNull

Stellt sicher das ein Item mindestens eine Zutat zugewiesen bekommt.

Deklaration:

```
1 inv ItemIngredientsBiggerNull:
2         self.content->collect(i|i)->size()>1
```

Quellcode 3.41: Deklaration der Invariante Item::ItemIngredientsBiggerNull

---

3.2.7 *class* Flavour

Die Klasse *Flavour* repräsentiert die Geschmacksrichtung eines Artikels (*class* Item, Kapitel 3.2.6).

Attribute

Name	Typ	Beschreibung
name	String	Die Geschmacksrichtung.

Operationen

Diese Klasse verfügt über keine Operationen.

Invarianten

Diese Klasse verfügt über keine Invarianten.

3.2.8 *association class* ExtraInCocktail

Bei der Klasse *OrderedItem* handelt es sich um eine Assoziationsklasse zwischen einem Artikel (*class* Item, Kapitel 3.2.6) und einem Extra (*class* Extra, Kapitel 3.2.9), die angibt, wie oft ein Extras im jeweiligen Artikel vorkommt.

Attribute

Name	Typ	Beschreibung
amount	Integer	Die Anzahl eines Extras in diesem Artikel.

Operationen

Da es sich hier um eine Assoziationsklasse handelt, haben wir auf Operationen verzichtet.

Invarianten

amountNotNegative

Stellt sicher, dass die Anzahl (*amount*) niemals negativ wird.

Deklaration:

```
1 inv amountNotNegative:
2         self.amount >= 0
```

Quellcode 3.42: Deklaration der Invariante ExtraInCocktail::amountNotNegative

3.2.9 *class* Extra

Die Klasse *Extra* repräsentiert alles was an Extras zu einem Artikel gehören kann oder für die Herstellung benötigt wird und dabei keine Zutaten sind. Zu ihr gehören bspw. Gefäße (*class* Jar, Kapitel 3.2.12) oder Werkzeuge (*class* Tool, Kapitel 3.2.11).

Attribute

Name	Typ	Beschreibung
name	String	Der Name des Extras.
amount	Integer	Die Anzahl des Extras.

Operationen

init()

Der Konstruktor der Klasse.

Vor- & Nachbedingungen

-

Deklaration:

```
1 init(aName:String, amnt:Integer)
2         begin
3                 self.name:=aName;
4                 self.amount:=amnt;
5         end
```

Quellcode 3.43: Quellcode von Extra::init()

Invarianten

amountNotNegative

Stellt sicher, dass die Anzahl des Extras nicht negativ wird.

Deklaration:

```
1 inv amountNotNegative:
2         self.amount >= 0
```

Quellcode 3.44: Deklaration der Invariante Extra::amountNotNegative

3.2.10 *class* Garnish

↳ `extends` *class* Extra

Die Klasse *Garnish* erbt von der Klasse *Extra* (*class* Extra, Kapitel 3.2.9) und stellt eine Garnierung für einen Artikel (*class* Item, Kapitel 3.2.6) dar. Also bspw. eine Verfeinerung eines Artikels die aber zu gering/nebensächlich ist um als eigenständige Zutat (*class* Ingredient, Kapitel 3.2.15) durchzugehen.

Attribute

Diese Unterklasse besitzt keine eigenen Attribute, erbt jedoch alle ihrer Oberklasse (*class* Extra, Kapitel 3.2.9).

Operationen

Diese Unterklasse besitzt keine eigenen Operationen, erbt jedoch alle ihrer Oberklasse (*class* Extra, Kapitel 3.2.9).

Invarianten

Diese Unterklasse besitzt keine eigenen Invarianten, erbt jedoch alle ihrer Oberklasse (*class* Extra, Kapitel 3.2.9).

3.2.11 *class* Tool

↳ `extends` *class* Extra

Die Klasse *Tool* erbt von der Klasse *Extra* (*class* Extra, Kapitel 3.2.9) und repräsentiert ein Werkzeug oder Hilfsmittel, das benötigt wird um einen Artikel (*class* Item, Kapitel 3.2.6) zuzubereiten.

Attribute

Diese Unterklasse besitzt keine eigenen Attribute, erbt jedoch alle ihrer Oberklasse (*class* Extra, Kapitel 3.2.9).

Operationen

Diese Unterklasse besitzt keine eigenen Operationen, erbt jedoch alle ihrer Oberklasse (*class* Extra, Kapitel 3.2.9).

Invarianten

Diese Unterklasse besitzt keine eigenen Invarianten, erbt jedoch alle ihrer Oberklasse (*class* Extra, Kapitel 3.2.9).

3.2.12 *class* Jar

↳ `extends` *class* Extra

Die Klasse *Jar* erbt von der Klasse *Extra* (*class* Extra, Kapitel 3.2.9) und stellt ein Behältnis oder Gefäß für einen Artikel (*class* Item, Kapitel 3.2.6) dar.

Attribute

Name	Typ	Beschreibung
measures	String	Die Ausmaße des Gefäßes.
capacity	Integer	Die Kapazität des Gefäßes.

Desweiteren erbt diese Klasse alle Attribute ihrere Oberklasse (*class* Extra, Kapitel 3.2.9).

Operationen

init()

Der Konstruktor der Klasse.

Vor- & Nachbedingungen

-

Deklaration:

```
1 init(aName:String, amnt:Integer, cap:Integer, meas:String)
2         begin
3                 self.name:=aName;
4                 self.amount:=amnt;
5                 self.capacity:=cap;
6                 self.measures:=meas;
7         end
```

Quellcode 3.45: Quellcode von Jar::init()

Desweiteren erbt diese Klasse alle Operationen ihrer Oberklasse (*class* Extra, Kapitel 3.2.9)

Invarianten

capacityNotNegative

Stellt sicher, dass die Kapazität eines Gefäßes größer Null ist.

Deklaration:

```
1 inv capacityNotNegative:
2         self.capacity >= 0
```

Desweiteren erbt diese Klasse alle Invarianten ihrer Oberklasse (*class* Extra, Kapitel 3.2.9).

3.2.13 *class* Cocktail

↳ `extends` *class* Item

Die Klasse *Cocktail* erbt von einem Artikel (*class* Item, Kapitel 3.2.6) und repräsentiert ein Cocktail.

Attribute

Diese Unterklasse besitzt keine eigenen Attribute, erbt jedoch alle ihrer Oberklasse (*class* Item, Kapitel 3.2.6).

Operationen

init()

Diese Operation fügt diesem Cocktail ein Gefäß (*class* Jar, Kapitel 3.2.12) hinzu.

Vor- & Nachbedingungen

-

Deklaration:

```
addGlass (jarr:Jar)
        begin
                insert (self, jarr) into Jar;
                insert (self, jarr) into ExtraInCocktail;
        end
```

Quellcode 3.46: Quellcode von Cocktail::addGlass()

Desweiteren erbt diese Klasse alle Operationen ihrer Oberklasse (*class* Item, Kapitel 3.2.6).

Invarianten

capacityMeetsAmount

Diese Invariante überprüft, ob der Cocktail auch in sein Gefäß hinein passt.

Deklaration:

```
inv capacityMeetsAmount:
        self.content.amount->sum() < self.jar.capacity
```

Quellcode 3.47: Deklaration der Invariante Cocktail::capacityMeetsAmount

jarIsInOwnExtras

Diese Invariante überprüft, das Gefäß auch in der Menge der eigenen Extras vorkommt.

Deklaration:

```
inv jarIsInOwnExtras:
        self.extra->includes(self.jar)
```

Quellcode 3.48: Deklaration der Invariante Cocktail::jarIsInOwnExtras

properComplexity

Diese Invariante definiert, dass bei zwei oder mehr Zutaten die Schwierigkeit/Komplexität des Artikels nicht mehr *leicht* sein kann.

Deklaration:

```
1 inv properComplexity:
2         self.content->collect(i|i)->size()>2 implies self.complexity<>#easy
```

Quellcode 3.49: Deklaration der Invariante Cocktail::properComplexity

Desweiteren erbt diese Klasse alle Invarianten ihrer Oberklasse (*class* Item, Kapitel 3.2.6).

3.2.14 *association class* Content

Bei der Klasse *Content* handelt es sich um eine Assoziationsklasse zwischen enem Artikel (*class* Item, Kapitel 3.2.6) und seinen Zutaten (*class* Ingredient, Kapitel 3.2.15), also zwischen einem Artikel und seiner Zutaten. Sie gibt an, wie oft eine Zutat in einem Artikel auftritt.

Attribute

Name	Typ	Beschreibung
amount	Integer	Die Anzahl der Zutat in einem Artikel.

Operationen

Da es sich hier um eine Assoziationsklasse handelt, haben wir auf Operationen verzichtet.

Invarianten

Da es sich hier um eine Assoziationsklasse handelt, haben wir auf Invarianten verzichtet.

3.2.15 *class* Ingredient

Die Klasse *Ingredient* repräsentiert ein Zutat für einen Artikel (*class* Item, Kapitel 3.2.6). Je nachdem wie von ihr geerbt wird kann es sich dabei um Esswaren (*class* Food, Kapitel 3.2.16) sowie um Flüssigkeiten/Getränke (*class* Liquid, Kapitel 3.2.17) handeln.

Attribute

Name	Typ	Beschreibung
name	String	Der Name der Zutat.
countable	Flavour	Countable gibt an ob die Zutat abzählbar ist oder ob nicht.
volume	Integer	Das Volumen/die Ausdehnung der Zutat.
calories	Integer	Der Brennwert der Zutat in Kalorien.
inStock	String	Der Lagerbestand der Zutat, also wieviel noch verfügbar ist.

Operationen

init()

Der Konstruktor der Klasse.

Vor- & Nachbedingungen

Name	Typ	Beschreibung
positiveValues	pre	Überprüft, ob die Eingabe für *stock*, *cals* und *vol* positiv sind.

Deklaration:

```
1  init(aName:String, counta:Boolean, vol:Integer, cals:Integer, stock:Integer)
2          begin
3                  self.name:=aName;
4                  self.countable:=counta;
5                  self.volume:=vol;
6                  self.calories:=cals;
7                  self.inStock:=stock;
8          end
9          pre positiveValues:
10                 stock >= 0 and cals >= 0 and vol >= 0
```

Quellcode 3.50: Quellcode von Ingredient::init()

decreaseStock(amount:Integer)

Veringert den Lagerbestand um *amount* Einheiten.

Vor- & Nachbedingungen

Name	Typ	Beschreibung
enoughInStock	pre	Überprüft, ob überhaupt genug Einheiten im Lager sind um sie um *amount* zu verringern.

Deklaration:

```
1 decreaseStock(amount:Integer)
2         begin
3                 self.inStock := self.inStock - amount;
4         end
5         pre enoughInStock:
6                 self.inStock >= amount;
```

Quellcode 3.51: Quellcode von Ingredient::decreaseStock()

increaseStock(amount:Integer)

Erhöht den Lagerbestand um *amount* Einheiten.

Vor- & Nachbedingungen

Name	Typ	Beschreibung
-	-	-

Deklaration:

```
1 increaseStock(amount:Integer)
2         begin
3                 self.inStock := self.inStock + amount;
4         end
```

Quellcode 3.52: Quellcode von Ingredient::increaseStock()

Invarianten

volumeNotNegative

Stellt sicher, dass das Volumen niemals Negativ wird.

Deklaration:

```
1 inv volumeNotNegative:
2         self.volume >= 0
```

Quellcode 3.53: Deklaration der Invariante Ingredient::volumeNotNegative

caloriesNotNegative

Stellt sicher, dass der Brennwert niemals negativ wird.

Deklaration:

```
1 inv caloriesNotNegative:
2         self.calories >= 0
```

Quellcode 3.54: Deklaration der Invariante Ingredient::caloriesNotNegative

inStockNotNegative

Stellt sicher, dass der Lagerbestand niemals negativ wird.

Deklaration:

```
1 inv inStockNotNegative:
2         self.inStock >= 0
```

Quellcode 3.55: Deklaration der Invariante Ingredient::inStockNotNegative

3.2.16 *class* Food

↳ `extends` *class* Ingredient

Die Klasse *Food* erbt von einer Zutat (*class* Ingredient, Kapitel 3.2.15) und macht sie zu einer Essware.

Attribute

Diese Unterklasse besitzt keine eigenen Attribute, erbt jedoch alle ihrer Oberklasse (*class* Ingredient, Kapitel 3.2.15).

Operationen

Diese Unterklasse besitzt keine eigenen Operationen, erbt jedoch alle ihrer Oberklasse (*class* Ingredient, Kapitel 3.2.15).

Invarianten

Diese Unterklasse besitzt keine eigenen Invarianten, erbt jedoch alle ihrer Oberklasse (*class* Ingredient, Kapitel 3.2.15).

3.2.17 *class* Liquid

↳ `extends` *class* Ingredient

Die Klasse *Liquid* erbt von einer Zutat (*class* Ingredient, Kapitel 3.2.15) und macht sie zu einer Flüssigkeit bzw. zu einem Getränk.

Attribute

Name	Typ	Beschreibung
densitiy	Real	Die Dichte der Flüssigkeit.

Desweiteren erbt diese Klasse alle Attribute ihrer Oberklasse (*class* Ingredient, Kapitel 3.2.15).

Operationen

init()

Der Konstruktor der Klasse.

Vor- & Nachbedingungen

Name	Typ	Beschreibung
positiveValues	pre	Überprüft, ob die Eingabe für *stock*, *cals* und *vol* positiv sind.

Deklaration:

```
1  init(aName:String, counta:Boolean, vol:Integer, cals:Integer, ↙
       ↳stock:Integer, dense:Real)
2          begin
3                  self.name:=aName;
4                  self.countable:=counta;
5                  self.volume:=vol;
6                  self.calories:=cals;
7                  self.inStock:=stock;
8                  self.density:=dense;
9          end
10         pre positiveValues:
11                 stock >= 0 and vol >= 0 and cals >= 0 and dense >= 0
```

Quellcode 3.56: Quellcode von Liquid::init()

Desweiteren erbt diese Klasse alle Operationen ihrer Oberklasse (*class* Ingredient, Kapitel 3.2.15).

Invarianten

Diese Unterklasse besitzt keine eigenen Invarianten, erbt jedoch alle ihrer Oberklasse (*class* Ingredient, Kapitel 3.2.15).

3.2.18 *class* Juice

↳ `extends` *class* Liquid
↳ `extends` *class* Ingredient

Die Klasse *Juice* erbt von einer Flüssigkeit (*class* Liquid, Kapitel 3.2.17) und macht sie zu einem Saft oder allgemein gesagt zu einer nicht alkoholischen Flüssigkeit.

Attribute

Diese Unterklasse besitzt keine eigenen Attribute, erbt jedoch alle ihrer Oberklasse (*class* Liquid, Kapitel 3.2.17).

Operationen

Diese Unterklasse besitzt keine eigenen Operationen, erbt jedoch alle ihrer Oberklasse (*class* Liquid, Kapitel 3.2.17).

Invarianten

Diese Unterklasse besitzt keine eigenen Invarianten, erbt jedoch alle ihrer Oberklasse (*class* Liquid, Kapitel 3.2.17).

3.2.19 *class* Alcohol

↳ `extends` *class* Liquid
↳ `extends` *class* Ingredient

Die Klasse *Alcohol* erbt von einer Flüssigkeit (*class* Liquid, Kapitel 3.2.17) und macht sie zu einem alkoholischen Getränk. Zu welcher Art von Spirituose dieses Getränk gehört kann in der Klasse *Spirit* (*association* Spirit, Kapitel 3.2.17) definiert werden.

Attribute

Name	Typ	Beschreibung
percent	Real	Der Alkoholgehalt dieser Flüssigkeit in Volumenprozenten.

Desweiteren erbt diese Klasse alle Attribute ihrer Oberklasse (*class* Ingredient, Kapitel 3.2.15).

Operationen

init()

Der Konstruktor der Klasse.

Vor- & Nachbedingungen

Name	Typ	Beschreibung
positiveValues	pre	Überprüft, ob die Eingabe für *stock*, *cals*, *vol*, *dense* und *perc* positiv sind.

Deklaration:

```
1  init(aName:String, counta:Boolean, vol:Integer, cals:Integer, ↙
       ↳stock:Integer, dense:Real, perc:Integer, spir:Spirit)
2          begin
3                  self.name:=aName;
4                  self.countable:=counta;
5                  self.volume:=vol;
6                  self.calories:=cals;
7                  self.inStock:=stock;
8                  self.density:=dense;
9                  self.percent:=perc;
10                 insert (spir, self) into Spirit;
11         end
12         pre positiveValues:
13                 stock >= 0 and vol >= 0 and cals >= 0 and dense >= 0 and ↙
                       ↳perc > 0
```

Quellcode 3.57: Quellcode von Alcohol::init()

Desweiteren erbt diese Klasse alle Operationen ihrer Oberklasse (*class* Liquid, Kapitel 3.2.17).

Invarianten

capacityNotNegative

Stellt sicher, dass der Alkoholgehalt dieser Flüssigkeit größer Null ist.

Deklaration:

```
1 inv percentNotNegative:
2         self.percent > 0
```

Quellcode 3.58: Deklaration der Invariante Alcohol::percentNotNegative()

Desweiteren erbt diese Klasse alle Invarianten ihrer Oberklasse (*class* Liquid, Kapitel 3.2.17).

3.2.20 *class* Spirit

Die Klasse *Spirit* repräsentiert eine Art von Spirituosen in der dann ein alkoholisches Getränk (*class* Alcohol, Kapitel 3.2.19) kategorisiert werden kann.

Attribute

Name	Typ	Beschreibung
name	String	Die Art der Spirituose.

Operationen

Diese Klasse verfügt über keine Operationen.

Invarianten

Diese Klasse verfügt über keine Invarianten.

3.2.21 *enumeration class* OrderState

Bei der Klasse *OrderState* handelt es sich um eine Aufzählungsklasse. Sie gibt den Status einer Bestellung (*class* AOrder, Kapitel 3.2.2) an.

Attribute

Name	Typ	Beschreibung
isOrdered	-	Bestellung wurde Bestellt.
readyToServer	-	Bestellung ist fertig zum servieren.
delivered	-	Bestellung wurde serviert.
billing	-	Bestellung wurde in Rechnung gestellt.
paid	-	Bestellung wurde bezahlt.

Operationen

Da es sich hier um eine Aufzählungsklasse handelt, gibt es hier keine Operationen.

Invarianten

Da es sich hier um eine Aufzählungsklasse handelt, gibt es hier keine keine Invarianten.

3.2.22 *enumeration class* Complexity

Bei der Klasse *Complexity* handelt es sich um eine Aufzählungsklasse. Sie gibt die Schwierigkeit/Komplexität einer Zubereitung für einen Artikel (*class* Item, Kapitel 3.2.6) an.

Attribute

Name	Typ	Beschreibung
easy	-	Die Zubereitung des jeweiligen Artikels ist leicht und erfordert keine Vorkenntnisse.
medium	-	Die Zubereitung des jeweiligen Artikels ist mittelschwer und nur für Fortgeschrittene mit Vorkenntnissen gedacht.
hard	-	Die Zubereitung des jeweiligen Artikels ist schwer und nur für Experten mit vielen Vorkenntnissen gedacht.

Operationen

Da es sich hier um eine Aufzählungsklasse handelt, gibt es hier keine Operationen.

Invarianten

Da es sich hier um eine Aufzählungsklasse handelt, gibt es hier keine keine Invarianten.

3.3 Assoziationen

Im folgenden haben wir alle Assoziationen mit ihren Multiplizitäten und Assoziationsklassen dokumentiert.

3.3.1 *association* CocktailOrder

`associate` *class* Bill
`associate` *class* AOrder

Die Assoziation *CocktailOrder* verknüpft eine Rechnung (*class* Bill, Kapitel 3.2.1) mit einer Bestellung (*class* AOrder, Kapitel 3.2.2).

Multiplizitäten

Die Multiplizitäten sind so gewählt, dass eine Rechnung mindestens eine bis beliebig viele (`1..*`) Bestellungen fassen kann und eine Bestellung zu keiner oder einer Rechnung gehört (`0..1`).

Deklaration

```
1 association CocktailOrder between
2         AOrder [1..*]
3         Bill   [0..1]
4 end
```

Quellcode 3.59: Deklaration der Assoziation CocktailOrder

Assoziationsklasse

Diese Assoziation verfügt über keine Assoziationsklasse.

3.3.2 *association* OrderedItem

`associate` *class* Bill
`associate` *class* AOrder

Die Assoziation *OrderedItem* verknüpft eine Bestellung (*class* AOrder, Kapitel 3.2.2) mit einem Angebot (*class* Offer, Kapitel 3.2.4).

Multiplizitäten

Die Multiplizitäten sind so gewählt, dass eine Bestellung beliebig viele Angebote haben kann (`0..*`) und andersherum ein Angebot auch beliebig viele Bestellungen (`0..*`).

Deklaration

```
1 associationclass OrderedItem between
2         Offer   [*]
3         AOrder  [*]
4 end
```

Quellcode 3.60: Deklaration der Assoziation OrderedItem

Assoziationsklasse

Diese Assoziation wird durch eine Assoziationsklasse ergänzt, die mit dem Attribut *amount* angibt wie oft ein Angebot in einer Bestellung vorkommt. Mehr dazu kann man unter *association class* OrderedItem im Kapitel 3.2.3 erfahren.

3.3.3 *association* Offer

`associate` *class* Offer
`associate` *class* Item

Die Assoziation *Offer* verknüpft ein Angebot (*class* Offer, Kapitel 3.2.4) mit einem Artikel (*class* Item, Kapitel 3.2.6).

Multiplizitäten

Die Multiplizitäten sind so gewählt, dass ein Angebot genau ein Artikel haben muss (`1`) und ein Artikel ein oder gar kein Angebot haben kann (`0..1`).

Deklaration

```
1 association Offer between
2         Item  [1]    role item
3         Offer [0..1] role offer
4 end
```

Quellcode 3.61: Deklaration der Assoziation Offer

Assoziationsklasse

Diese Assoziation verfügt über keine Assoziationsklasse.

3.3.4 *association* MenuItem

`associate` *class* Offer
`associate` *class* Menu

Die Assoziation *MenuItem* verknüpft ein Angebot (*class* Offer, Kapitel 3.2.4) mit einem Menü (*class* Menu, Kapitel 3.2.5).

Multiplizitäten

Die Multiplizitäten sind so gewählt, dass ein Menü mindestens ein bis beliebig viele (`1..*`) Angebote fassen kann und ein Angebot zu einem oder keinem Menü gehört (`0..1`).

Deklaration

```
1 association MenuItem between
2         Offer [1..*] role item
3         Menu  [0..1] role menu
4 end
```

Quellcode 3.62: Deklaration der Assoziation MenuItem

Assoziationsklasse

Diese Assoziation verfügt über keine Assoziationsklasse.

3.3.5 *association* ExtraInCocktail

`associate` *class* Item
`associate` *class* Extra

Die Assoziation *ExtraInCocktail* ordnet einem Artikel (*class* Item, Kapitel 3.2.6) ein Extra (*class* Extra, Kapitel 3.2.9) zu.

Multiplizitäten

Die Multiplizitäten sind so gewählt, dass ein Artikel beliebig viele Extras haben kann (`0..*`) und andersherum ein Extra auch beliebig viele Artikel (`0..*`).

Deklaration

```
1 associationclass ExtraInCocktail between
2         Item   [*]
3         Extra  [*]
4 end
```

Quellcode 3.63: Deklaration der Assoziation ExtraInCocktail

Assoziationsklasse

Diese Assoziation wird durch eine Assoziationsklasse ergänzt, die mit dem Attribut *amount* angibt wie oft ein Extra in einem Artikel vorkommt. Mehr dazu kann man unter *association class* ExtraInCocktail im Kapitel 3.2.8 erfahren.

3.3.6 *association* Jar

`associate` *class* Cocktail
`associate` *class* Jar

Die Assoziation *Jar* ordnet einem Cocktail (*class* Cocktail, Kapitel 3.2.13) mit ein Gefäß zu (*class* Jar, Kapitel 3.2.12).

Multiplizitäten

Die Multiplizitäten sind so gewählt, dass ein Cocktail genau ein Gefäß haben kann (`1`) und eine Art Gefäß zu beliebig vielen Cocktails gehören kann (`0..1`).

Deklaration

```
1 association Jar between
2         Cocktail [*]
3         Jar      [1]
4 end
```

Quellcode 3.64: Deklaration der Assoziation Jar

Assoziationsklasse

Diese Assoziation verfügt über keine Assoziationsklasse.

3.3.7 *association* Flavour

`associate` *class* Item
`associate` *class* Flavour

Die Assoziation *Flavour* ordnet einem Artikel (*class* Item, Kapitel 3.2.6) einem Geschmack zu (*class* Flavour, Kapitel 3.2.7).

Multiplizitäten

Die Multiplizitäten sind so gewählt, dass ein Artikel beliebig viele Geschmäcker haben kann (`0..*`) und andersherum ein Geschmack auch in beliebig vielen Artikeln vorkommen kann (`0..*`).

Deklaration

```
1 association Flavour between
2         Item     [0..*]
3         Flavour  [0..*]
4 end
```

Quellcode 3.65: Deklaration der Assoziation Flavour

Assoziationsklasse

Diese Assoziation verfügt über keine Assoziationsklasse.

3.3.8 *association* Content

`associate` *class* Item
`associate` *class* Ingredient

Die Assoziation *Content* ordnet einem Artikel (*class* Item, Kapitel 3.2.6) seine Zutaten (*class* Ingredient, Kapitel 3.2.15) zu.

Multiplizitäten

Die Multiplizitäten sind so gewählt, dass ein Artikel beliebig viele Zutaten haben kann (`0..*`) und andersherum eine Zutat auch zu beliebig vielen Artikeln gehören kann (`0..*`).

Deklaration

```
1 associationclass Content between
2         Item       [*]
3         Ingredient [*]
4 end
```

Quellcode 3.66: Deklaration der Assoziation Content

Assoziationsklasse

Diese Assoziation wird durch eine Assoziationsklasse ergänzt, die mit dem Attribut *amount* angibt wie oft eine Zutat in einem Artikel vorkommt. Mehr dazu kann man unter *association class* Content im Kapitel 3.2.14 erfahren.

3.3.9 *association* Spirit

`associate` *class* Alcohol
`associate` *association* Spirit

Die Assoziation *Spirit* ordnet einem konkreten alkoholischen Getränk (*class* Alcohol, Kapitel 3.2.19) eine Klasse von Spiritousen (*association* Spirit, Kapitel 3.3.9) zu.

Multiplizitäten

Die Multiplizitäten sind so gewählt, dass ein alkoholisches Getränk zu einer oder keiner Klasse von Spirituosen gehören kann `0..1`) und eine Spiritousenklasse beliebig viele alkoholische Getränke haben kann.

Deklaration

```
1 association Spirit between
2         Spirit  [1] role spirit
3         Alcohol [0..1] role alcohol
4 end
```

Quellcode 3.67: Deklaration der Assoziation Spirit

Assoziationsklasse

Diese Assoziation verfügt über keine Assoziationsklasse.

Kapitel 4

Szenarien

Im Folgenden finden sich die von uns gewählten Szenarien, die als Testfälle unser System überprüfen und zeigen ob das System korrekt reagiert, wieder. Die Szenarien wurden in Gruppen eingeteilt, welche jeweils einem thematisch zusammengehörenden Anwendungsfall entsprechen. Diese sind dann weiter unterteilt in jeweils positive und negative Abläufe, in denen kleinere Veränderungen am Szenario jeweils zu einem positiven oder negativen Ergebnis führen.

Die Scripte befinden sich alle unter *Diagramme/Szenarien*

4.1 Setup

4.1.1 Korrektes Setup

Hier testen wir zunächst unser Setup in dem sich bereits einige Objekte befinden und für die weitere Prüfung in den folgenden Szenarien genutzt werden und ein bereits laufendes System simulieren sollen.
Damit dies als Ausgangspunkt dienen kann stellen wir hier zunächst dar, dass alle Invarianten und Multiplizitäten korrekt eingehalten werden.

```
1 open Helper_ModelBase.soil
```

Quellcode 4.1: Skript (Sz0_Base_Model.cmd)

Diagramm

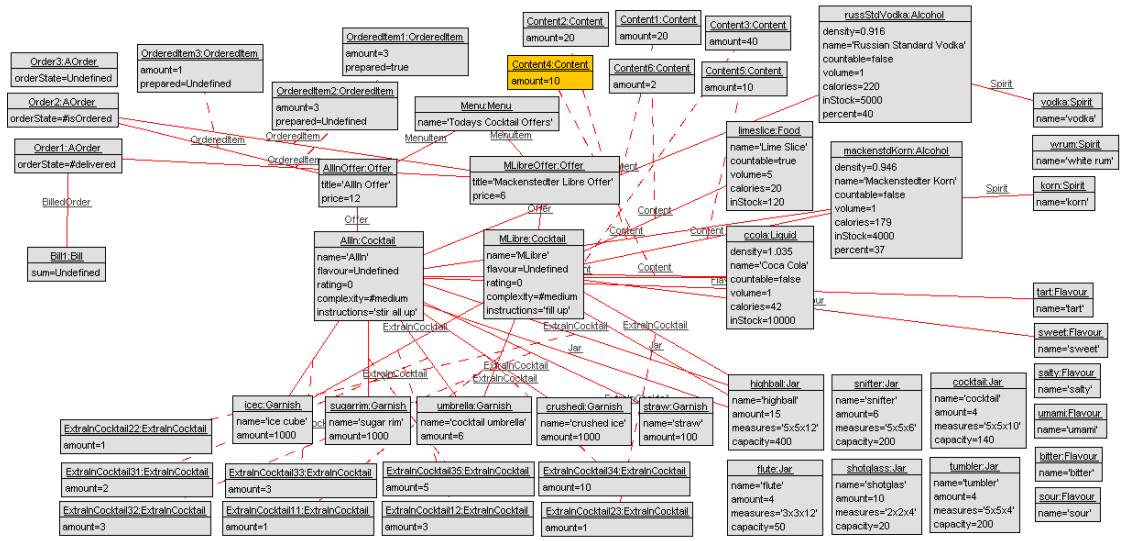


Abbildung 4.1: Die Objekte des Setupscripts

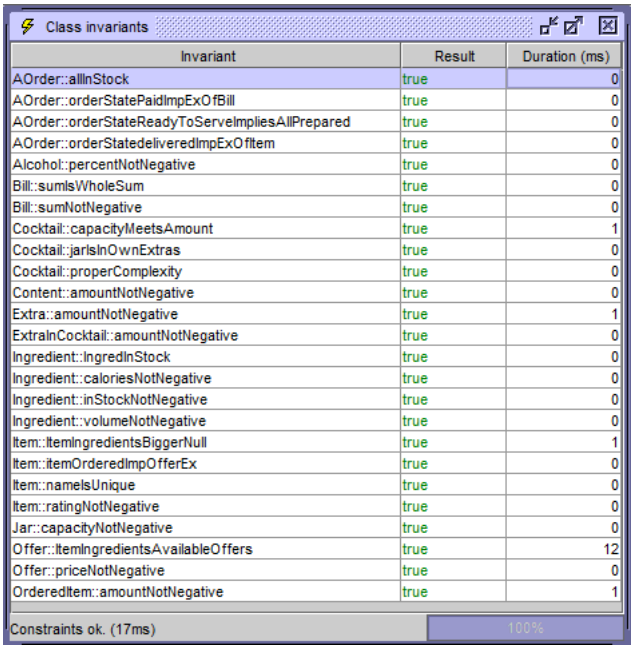


Abbildung 4.2: Alle Invarianten positiv nach dem Setup

4.2 Bestellen

4.2.1 Positiv

Hier wird eine neue Bestellung aufgegeben, zwei "Mackenstedter Libre" und sechs "AllIn" bestellt. Anschließend werden die Cocktails gefertigt und von der Bedienung ausgeliefert. Zum Schluss wird die Rechnung für diese Bestellung angefordert und bezahlt.
Alles läuft korrekt und sollte bei keinen Bedingungen Probleme verursachen.

```
1 open Helper_ModelBase.soil
2
3 -- Neue Bestellung und einzelne Items aufnehmen
4 !create ord1:AOrder
5 !ord1.placeOrder(MLibreOffer, 2)
6 !ord1.placeOrder(AllInOffer, 6)
7
8 -- Barpersonal fragt zu fertigende Bestellungen ab,
9 -- bereitet diese zu und ist bereit zum Servieren
10 !ord1.prepareOrders()
11
12 -- Servierfertige Bestellungen werden von Bedienung abgefragt
13 !b := ord1.readyToServeOrders()
14
15 -- Bestellung ausgeliefert
16 !ord1.setDelivered()
17
18 -- in Rechnung stellen
19 !create b1:Bill
20 !b1.addOrder(ord1)
21
22 -- Rechnung ausgeben
23 !b1.getBill()
24
25 -- bezahlen
26 !b1.payBill()
```

Quellcode 4.2: Skript (Sz1_Bestellen_Pos.cmd)

Diagramm

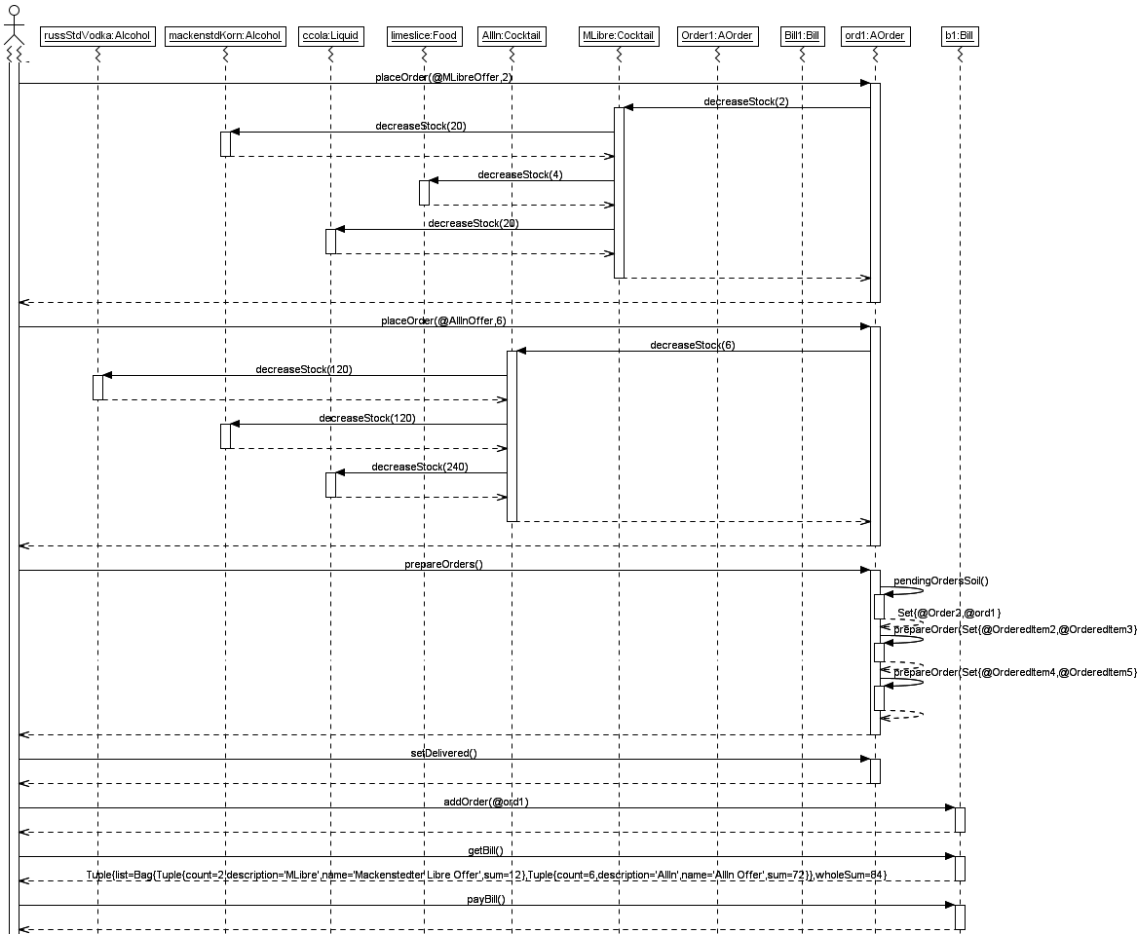


Abbildung 4.3: Erfolgreicher Bestellablauf

Class invariants

Invariant	Result	Duration (ms)
AOrder::allInStock	true	0
AOrder::orderStatePaidImpExOfBill	true	1
AOrder::orderStateReadyToServeImpliesAllPrepared	true	0
AOrder::orderStatedeliveredImpExOfItem	true	0
Alcohol::percentNotNegative	true	0
Bill::sumIsWholeSum	true	1
Bill::sumNotNegative	true	0
Cocktail::capacityMeetsAmount	true	0
Cocktail::jarIsInOwnExtras	true	0
Cocktail::properComplexity	true	0
Content::amountNotNegative	true	0
Extra::amountNotNegative	true	1
ExtraInCocktail::amountNotNegative	true	0
Ingredient::IngredInStock	true	0
Ingredient::caloriesNotNegative	true	0
Ingredient::inStockNotNegative	true	0
Ingredient::volumeNotNegative	true	0
Item::ItemIngredientsBiggerNull	true	0
Item::itemOrderedImpOfferEx	true	0
Item::nameIsUnique	true	1
Item::ratingNotNegative	true	0
Jar::capacityNotNegative	true	0
Offer::ItemIngredientsAvailableOffers	true	15
Offer::priceNotNegative	true	0
OrderedItem::amountNotNegative	true	1

Constraints ok. (24ms) 100%

Abbildung 4.4: Invarianten erfüllt

4.2.2 Negativ: Nicht genügend Vodka für eine Bestellungsannahme

Hierbei wird ein ebenso gültiger Bestellablauf wie im vorigen Fall durchgeführt. Das es sich hierbei um einen negativen Testfall handelt liegt an der vierten Zeile des Skripts. Hier wird der Wert des auf Lager befindlichen Vodkas auf Null gesetzt, so dass die Bestellung in Zeile Acht nicht mehr ausgeführt werden kann. Bei dieser Operation verhindert die Precondition AOrder::allIngreddientsInStock korrekter Weise, dass eine Anzahl an Cocktails bestellt werden kann für die nicht genug Zutaten vorhanden sind.

```
1 open Helper_ModelBase.soil
2
3 -- kein Vodka mehr da, AllIn Offer nicht bestellbar
4 !russStdVodka.inStock:=0
5
6 !create ord1:AOrder
7 !ord1.placeOrder(MLibreOffer, 2)
8 !ord1.placeOrder(AllInOffer, 6)
9
10 -- Barpersonal fragt zu fertigende Bestellungen ab
11 !a := ord1.pendingOrders()
12
13 -- Bestellung wurde zubereitet und ist bereit zum Servieren
14 !ord1.prepareOrders()
15
16 -- Servierfertige Bestellungen werden von Bedienung abgefragt
17 !b := ord1.readyToServeOrders()
18
19 -- Bestellung ausgeliefert
20 !ord1.setDelivered()
21
22 -- in Rechnung stellen
23 !create b1:Bill
24 !b1.addOrder(ord1)
25
26 -- Rechnung ausgeben
27 !b1.getBill()
28
29 -- bezahlen
30 !b1.payBill()
```

Quellcode 4.3: Skript (Sz1_Bestellen_Neg_Zu_Wenig_Vodka.cmd)

Diagramm

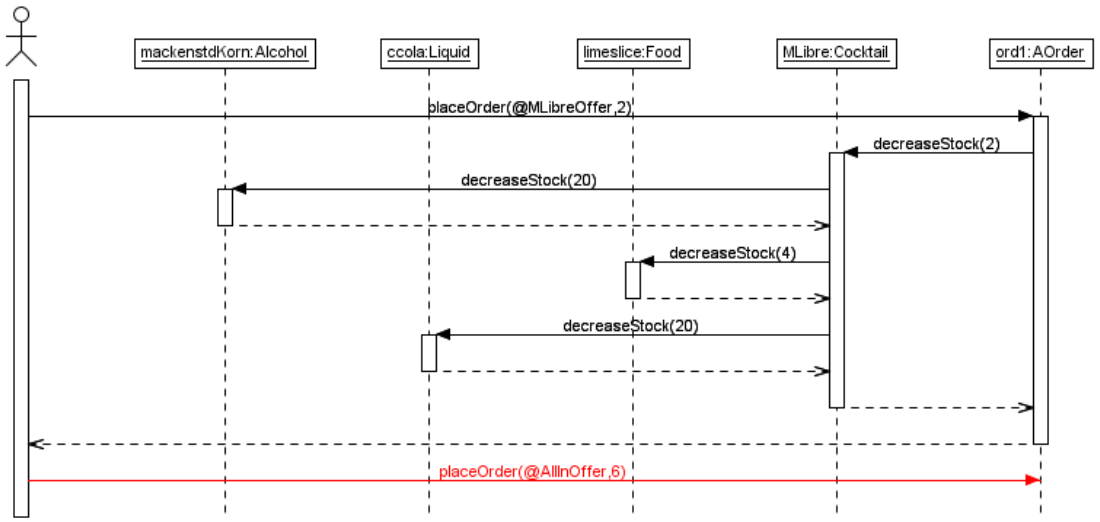


Abbildung 4.5: Precondition AOrder::allIngredientsInStock verletzt

4.2.3 Negativ: Nicht fertige Bestellungen werden ausgeliefert

Die Precondition AOrder::hasBeenPrepared verhindert, dass eine Bestellung ausgeliefert wird die noch nicht Vorbereitet wurde. Für das Vorbereiten ist der Aufruf von prepareOrders() notwendig.

Quellcode 4.4: Skript (Sz1_Bestellen_Neg_Nicht_Fertige_Werden_Ausgeliefert.cmd)

Diagramm

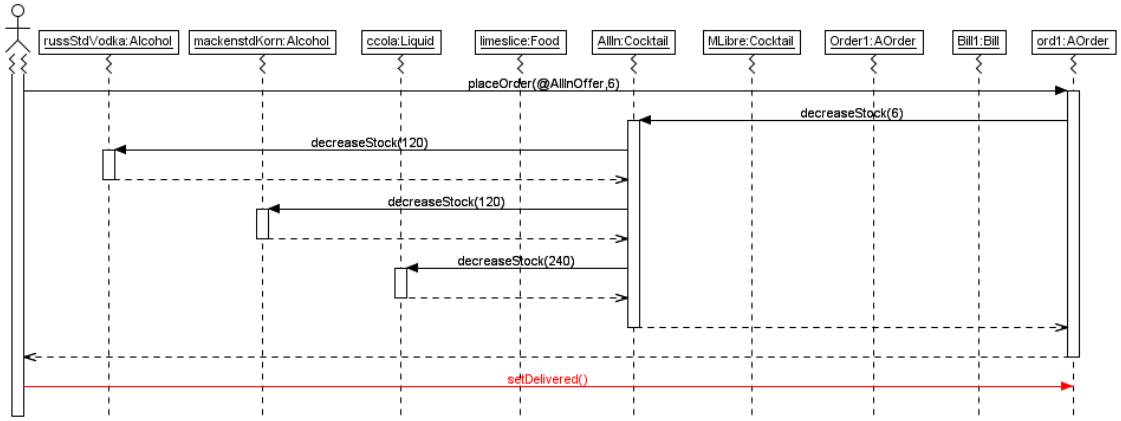


Abbildung 4.6: Precondition AOrder::hasBeenPrepared verletzt

4.3 Teilbestellungen

4.3.1 Positiv: Teilbestellung stornieren

In diesem Szenario wird ein Teil der Bestellung storniert. Dies ist ein regulärer Ablauf und führt zu keiner Verletzung von Bedingungen, da sechs "AllIn" bestellt wurden und vier wieder abbestellt werden, was auch möglich sein soll.

```
1 open Helper_ModelBase.soil
2
3 !create ord1:AOrder
4 !ord1.placeOrder(MLibreOffer, 2)
5 !ord1.placeOrder(AllInOffer, 6)
6
7 -- 4 Bestellungen entfernen
8 !ord1.removeOffer(AllInOffer, 4)
9
10 -- Barpersonal fragt zu fertigende Bestellungen ab
11 !a := ord1.pendingOrders()
```

Quellcode 4.5: Skript (Sz1_Bestellen_Pos.cmd)

Diagramm

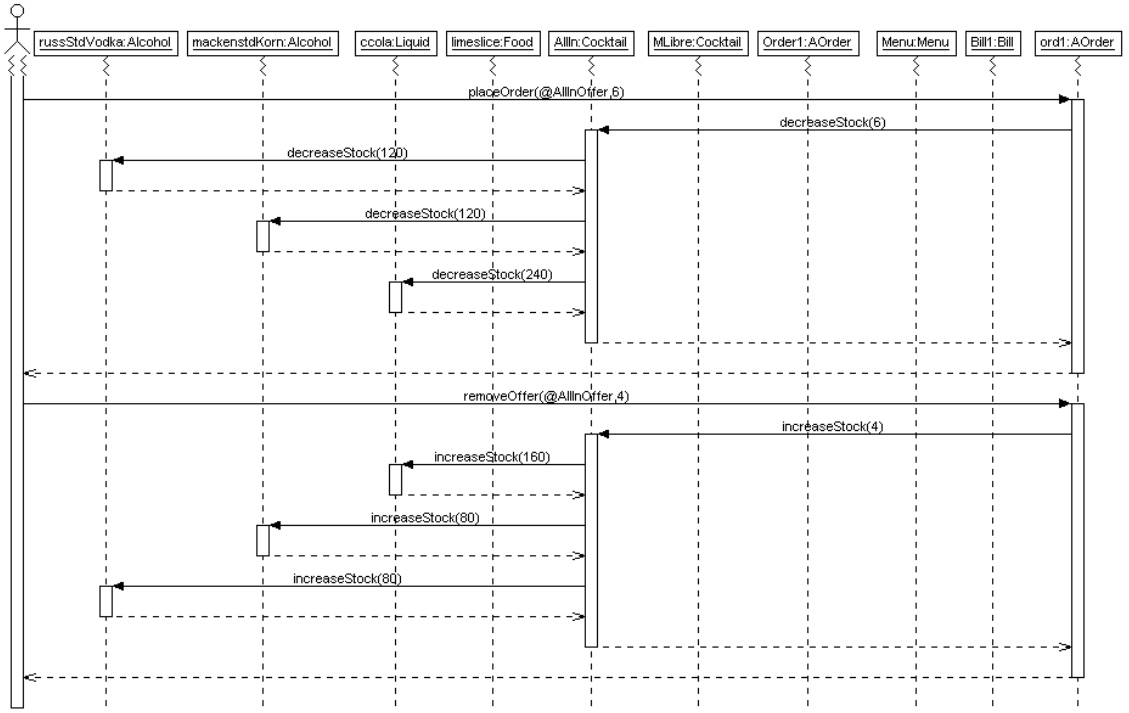


Abbildung 4.7: Erfolgreiche Teilstornierung

Nach dem Entfernen von vier AllIn Offers sind korrekterweise noch zwei Teil der Bestellung:

```
1 use> ?ord1.offer
```

```
2 -> Set{@AllInOffer,@MLibreOffer} : Set(Offer)
3 use> ?ord1.orderedItem
4 -> Set{@OrderedItem4,@OrderedItem5} : Set(OrderedItem)
5 use> ?ord1.orderedItem.amount
6 -> Bag{2,2} : Bag(Integer)
```

Quellcode 4.6: Abfrage der Objekte nach erfolgreichem Entfernen.

4.3.2 Negativ: Nicht bestellte Teilbestellung stornieren

Hier wird versucht eine Stornierung durchzuführen mit Artikeln die nicht bestellt wurden, mittels einer Precondition wird dies unterbunden.

```
1 open Helper_ModelBase.soil
2
3 -- neuer Cocktail
4 -- Vodka Libre
5 open Helper_Neue_Offer_VLibreOffer.cmd
6
7
8 !create ord1:AOrder
9 !ord1.placeOrder(MLibreOffer, 2)
10 !ord1.placeOrder(AllInOffer, 6)
11
12 -- FEHLER! Nicht bestellte Bestellung kann nicht entfernt werden
13 !ord1.removeOffer(VLibreOffer, 2)
14
15 -- Barpersonal fragt zu fertigende Bestellungen ab
16 !a := ord1.pendingOrders()
```

Quellcode 4.7: Skript (Sz2_Teilbestellung_Neg_Nicht_Bestellt.cmd)

Diagramm

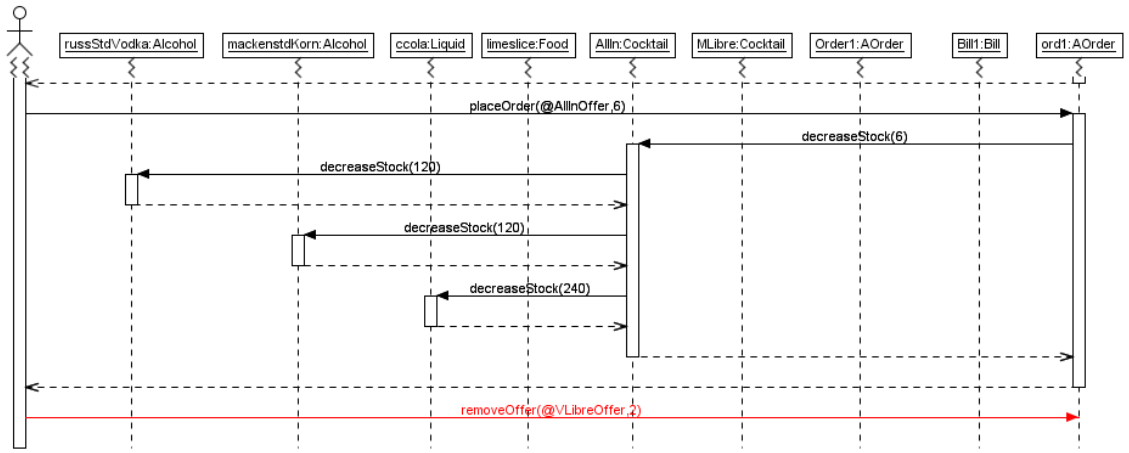


Abbildung 4.8: Precondition AOrder::offerIncluded verletzt

Nach dem gescheiterten Entfernen sind die zuvor bestellten Offer korrekterweise noch Teil der Bestellung:

```
1 use> ?ord1.offer
2 -> Set{@AllInOffer,@MLibreOffer} : Set(Offer)
```

Quellcode 4.8: Abfrage der Objekte nach gescheitertem Entfernen.

4.3.3 Negativ: Zu viele Offers stornieren

Hier wird versucht eine Stornierung durchzuführen bei der die Anzahl der Stornierungen die Anzahl der bestellten Artikel übersteigt. Dies wird durch die Precondition AOrder::enoughOrderedToBeremoved verhindert.

```
1  open Helper_ModelBase.soil
2
3  -- neuer Cocktail
4  -- Vodka Libre
5  open Helper_Neue_Offer_VLibreOffer.cmd
6
7
8  !create ord1:AOrder
9  !ord1.placeOrder(MLibreOffer, 2)
10 !ord1.placeOrder(AllInOffer, 6)
11
12 -- FEHLER! Zu viel koennen nicht storniert werden
13 !ord1.removeOffer(AllInOffer, 8)
14
15 -- Barpersonal fragt zu fertigende Bestellungen ab
16 !a := ord1.pendingOrders()
```

Quellcode 4.9: Skript (Sz2_Teilbestellung_Stornieren_Neg_Zu_Viel_Storniert_.cmd)

Diagramm

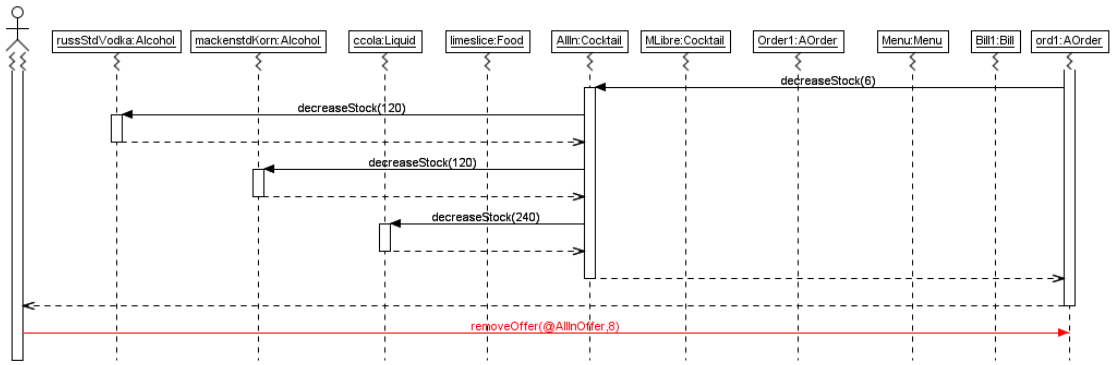


Abbildung 4.9: Precondition AOrder::enoughOrderedToBeremoved verletzt

Nach dem gescheiterten Entfernen von zu vielen Offers sind die zuvor bestellten Offer korrekterweise noch Teil der Bestellung und in entsprechender Anzahl vorhanden:

```
1 use> ?ord1.offer
2 -> Set{@AllInOffer,@MLibreOffer} : Set(Offer)
3 use> ?ord1.orderedItem
4 -> Set{@OrderedItem4,@OrderedItem5} : Set(OrderedItem)
5 use> ?ord1.orderedItem.amount
6 -> Bag{2,6} : Bag(Integer)
```

Quellcode 4.10: Abfrage der Objekte nach gescheitertem Entfernen.

4.4 Rechnung ausgeben

4.4.1 Positiv: Rechnungsliste ausgeben

.

```
1 open Helper_ModelBase.soil
2
3 !create ord1:AOrder
4 !ord1.placeOrder(MLibreOffer, 2)
5 !ord1.placeOrder(AllInOffer, 6)
6
7 -- Barpersonal fragt zu fertigende Bestellungen ab
8 !a := ord1.pendingOrders()
9
10 -- Bestellung wurde zubereitet und ist bereit zum Servieren
11 !ord1.prepareOrders()
12
13 -- Servierfertige Bestellungen werden von Bedienung abgefragt
14 !b := ord1.readyToServeOrders()
15
16 -- Bestellung ausgeliefert
17 !ord1.setDelivered()
18
19 -- in Rechnung stellen
20 !create b1:Bill
21 !b1.addOrder(ord1)
22
23 -- Rechnungsliste ausgeben
24 !b1.getBill()
```

Quellcode 4.11: Skript (Sz3_Rechnungsliste_Pos.cmd)

Diagramm

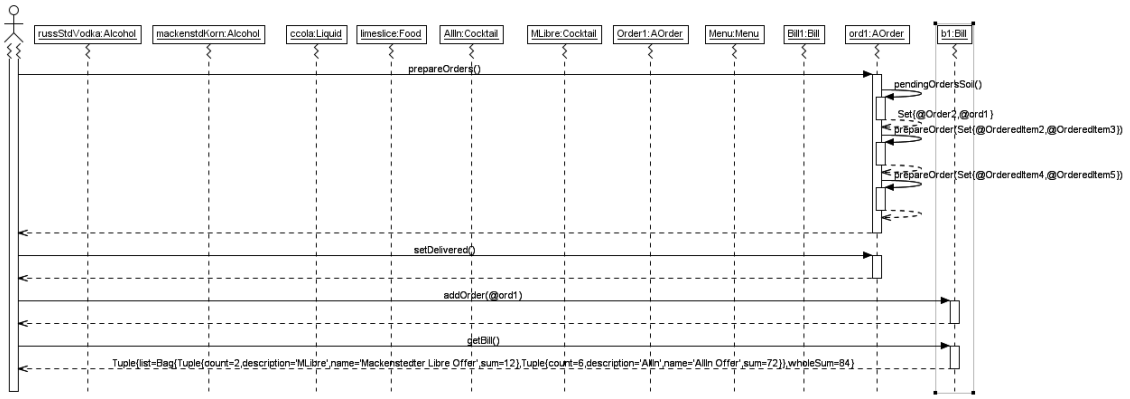


Abbildung 4.10: Rechnungsliste ausgeben

Nach dem erfolgreichen Bestellen und ausliefern lässt sich eine Rechnung mit den geordert Items der zugeordneten Bestellungen ausgeben und im Ergebnistupel ist zusätzlich die Gesamtsumme enthalten:

```
1 use> !a := b1.getBill()
2 use> ?a
3 -> Tuple{list=Bag{Tuple{count=2, description='MLibre', name='Mackenstedter␣↙
     ↪Libre␣Offer',sum=12}, Tuple{count=6, description='AllIn', name='AllIn
```

```
4 Offer',sum=72}}, wholeSum=84} : Tuple(list:Bag(Tuple(count:Integer, ↙
    ↳description:String, name:String, sum:Integer)), wholeSum:Integer)
5 use>
```

Quellcode 4.12: Resultierende Rechnungsliste nach der Bestellung.

4.5 Beschwerden

4.5.1 Positiv: Bemängeln einer Offer

Hier handelt es sich um das Szenario in dem eine Offer vom Gast bemängelt wird. Dies stößt an, dass eine neue Offer zu der Bestellung hinzugefügt wird und anschließend dem Gast kostenlos serviert wird. .

```
1  open Helper_ModelBase.soil
2
3  !create ord1:AOrder
4  !ord1.placeOrder(MLibreOffer, 2)
5  !ord1.placeOrder(AllInOffer, 6)
6
7  -- Barpersonal fragt zu fertigende Bestellungen ab
8  !a := ord1.pendingOrders()
9
10 -- Bestellung wurde zubereitet und ist bereit zum Servieren
11 !ord1.prepareOrders()
12
13 -- Servierfertige Bestellungen werden von Bedienung abgefragt
14 !b := ord1.readyToServeOrders()
15
16 -- Bestellung ausgeliefert
17 !ord1.setDelivered()
18
19 -- Bemaengeln
20 !ord1.complain(MLibreOffer, 1)
21
22 -- Bestellung erneut zubereiten
23 !ord1.prepareOrders()
24
25 -- Bestellung erneut ausgeliefert
26 !ord1.setDelivered()
27
28 -- in Rechnung stellen
29 !create b1:Bill
30 !b1.addOrder(ord1)
31
32 -- Rechnung ausgeben
33 !b1.getBill()
34
35 -- bezahlen
36 !b1.payBill()
```

Quellcode 4.13: Skript (Sz4_Bemaengeln_Pos.cmd)

Diagramm

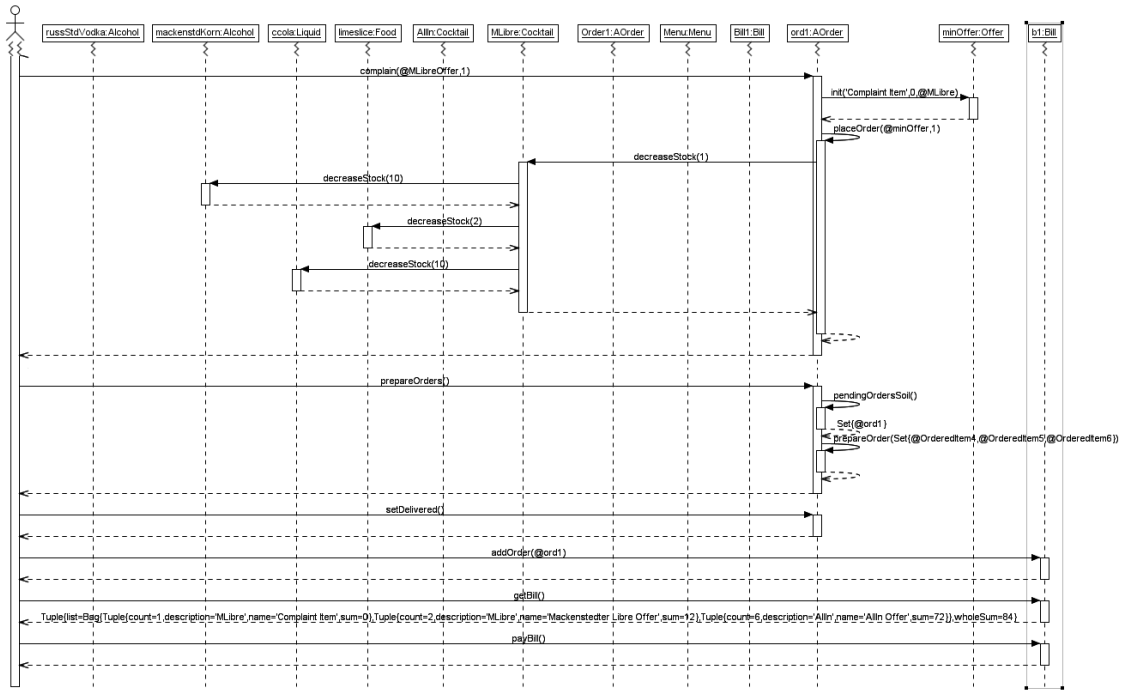


Abbildung 4.11: Bemängeln und anschließendes neu liefern einer Offer

Nach erfolgreicher Neubestellung der bemängelten Offer taucht sie für 0 Euro auf der Rechnung auf:

```
use> !a:= b1.getBill()
use> ?a
-> Tuple{list=Bag{Tuple{count=1,description='MLibre',name='Complaint␣↙
    ↳Item',sum=0},Tuple{count=2,description='MLibre',name='Mackenstedter␣Lib
re␣Offer',sum=12},Tuple{count=6,description='AllIn',name='AllIn␣↙
    ↳Offer',sum=72}},wholeSum=84} : ↙
    ↳Tuple(list:Bag(Tuple(count:Integer,descriptio
n:String,name:String,sum:Integer)),wholeSum:Integer)
```

Quellcode 4.14: Abfrage der Objekte nach gescheitertem Entfernen.

4.5.2 Negativ: Bemängeln einer nicht bestellten Offer

Hier handelt es sich um ein Szenario wie das vorige, mit dem Unterschied dass der Gast versucht sich über eine Order zu beschweren die er nicht bestellt hat. Folgerichtig wird dies vom System unterbunden .

```
1  open Helper_ModelBase.soil
2
3  !create ord1:AOrder
4  !ord1.placeOrder(AllInOffer, 6)
5
6  -- Barpersonal fragt zu fertigende Bestellungen ab
7  !a := ord1.pendingOrders()
8
9  -- Bestellung wurde zubereitet und ist bereit zum Servieren
10 !ord1.prepareOrders()
11
12 -- Servierfertige Bestellungen werden von Bedienung abgefragt
13 !b := ord1.readyToServeOrders()
14
15 -- Bestellung ausgeliefert
16 !ord1.setDelivered()
17
18 -- FEHLER - Bemaengeln aber nicht bestellt
19 !ord1.complain(MLibreOffer, 1)
```

Quellcode 4.15: Skript (Sz4_Item_Bemaengeln_Neg_Nicht_Bestellt.cmd)

Diagramm

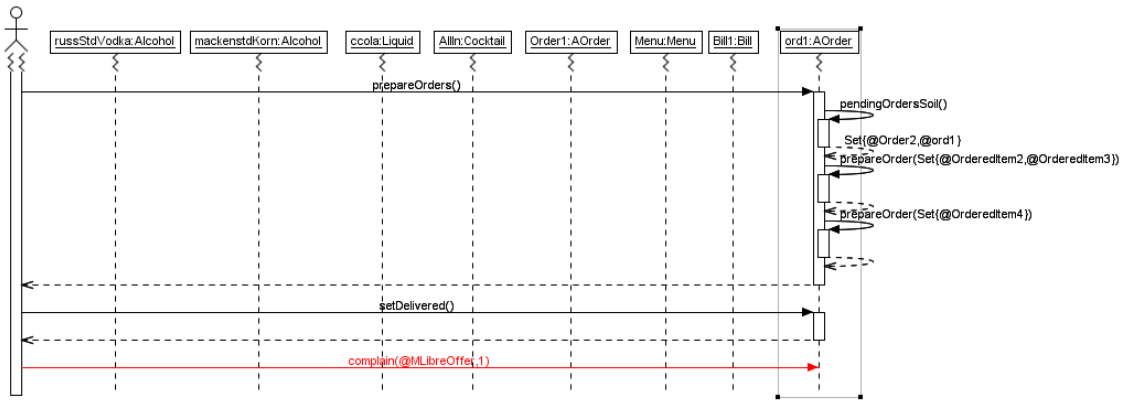


Abbildung 4.12: Precondition AOrder:: ist verletzt

Kapitel 5

Evolution

In diesem Kapitel wird darauf eingegangen, wie sich unser Model im Zeitverlauf verändert hat und welche äußeren Einflüsse hierzu geführt haben. Neben reichlicher Überlegung, half uns USE, Fehler in Struktur und Multiplizitäten zu finden. Darüber hinaus stellte sich der KodKod Validator als hilfreiches Tool dar. Unsere Verwendung und den daraus gezogenen Nutzen stellen wir im Kapitel 7 Modelvalidation ausführlicher vor.

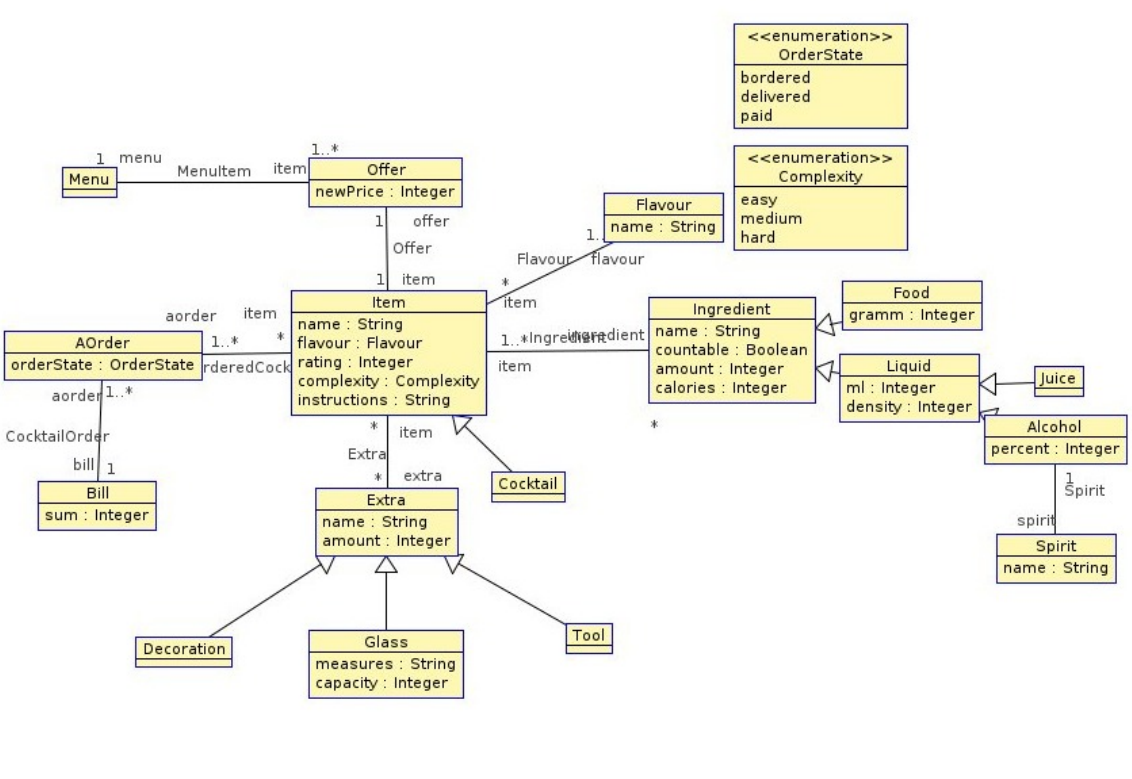


Abbildung 5.1: Klassen Diagramm in einem frühen Statium

Die Evolution lässt sich in verschiedene Teile gliedern. Am Anfang stand die Entwicklung des Klassendiagramm im Vordergrund. Es wurden Klassen, Attribute und Multiplizitäten definiert.

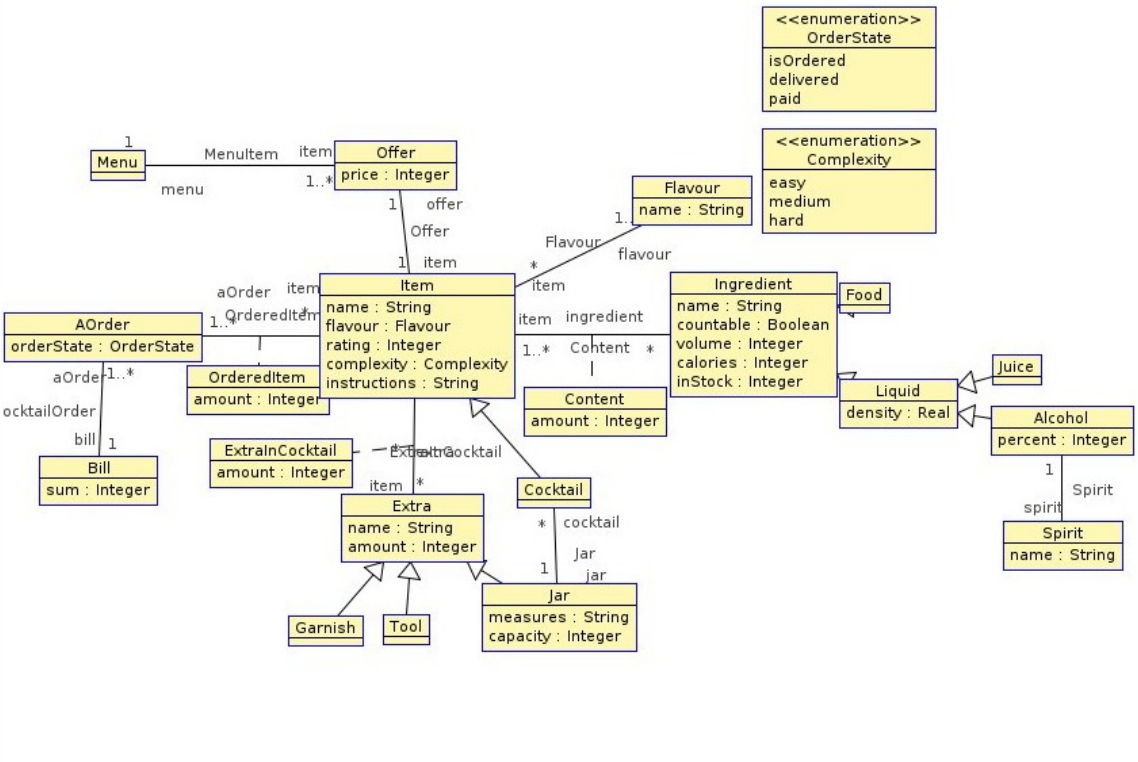


Abbildung 5.2: Klassen Diagramm vor Structure Check

Anschließend wurden die ersten Objektdiagramme erzeugt und erste Invarianten definiert. Wie im Abschnitt *Structure Check* (Structure Check, Kapitel 7.2) beschrieben, hat mit den ersten Objektdiagrammen der *Structure Check* einige Fehler in den Multiplizitäten aufgedeckt.

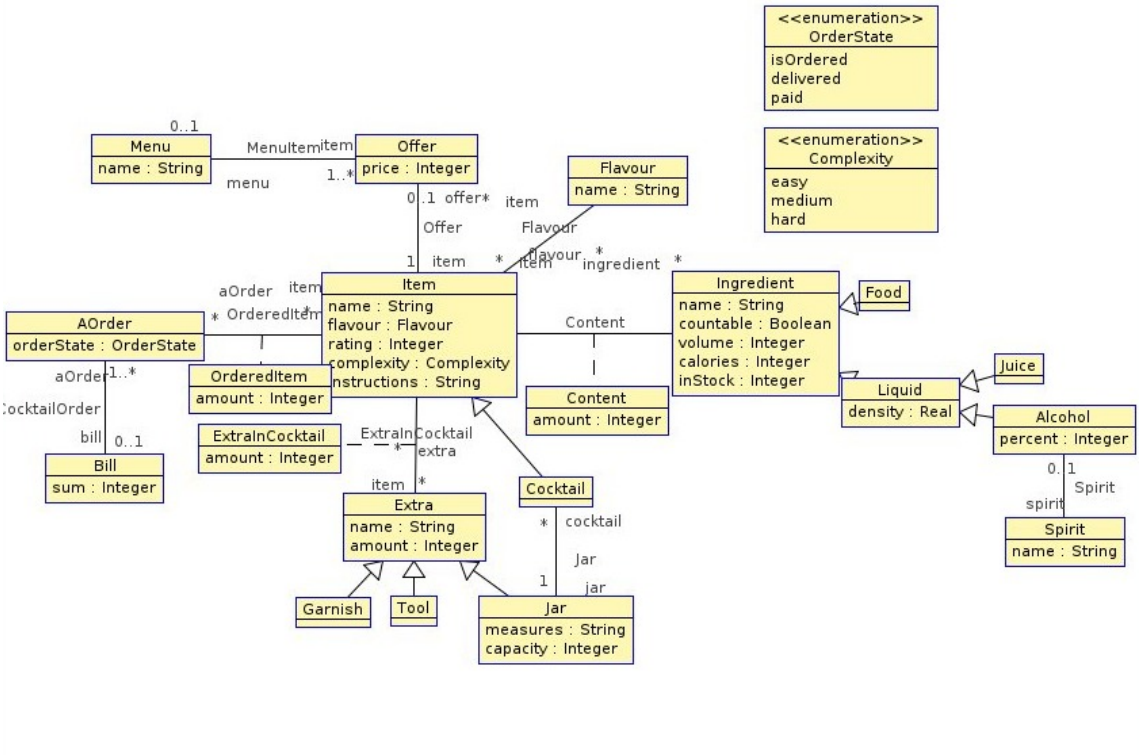
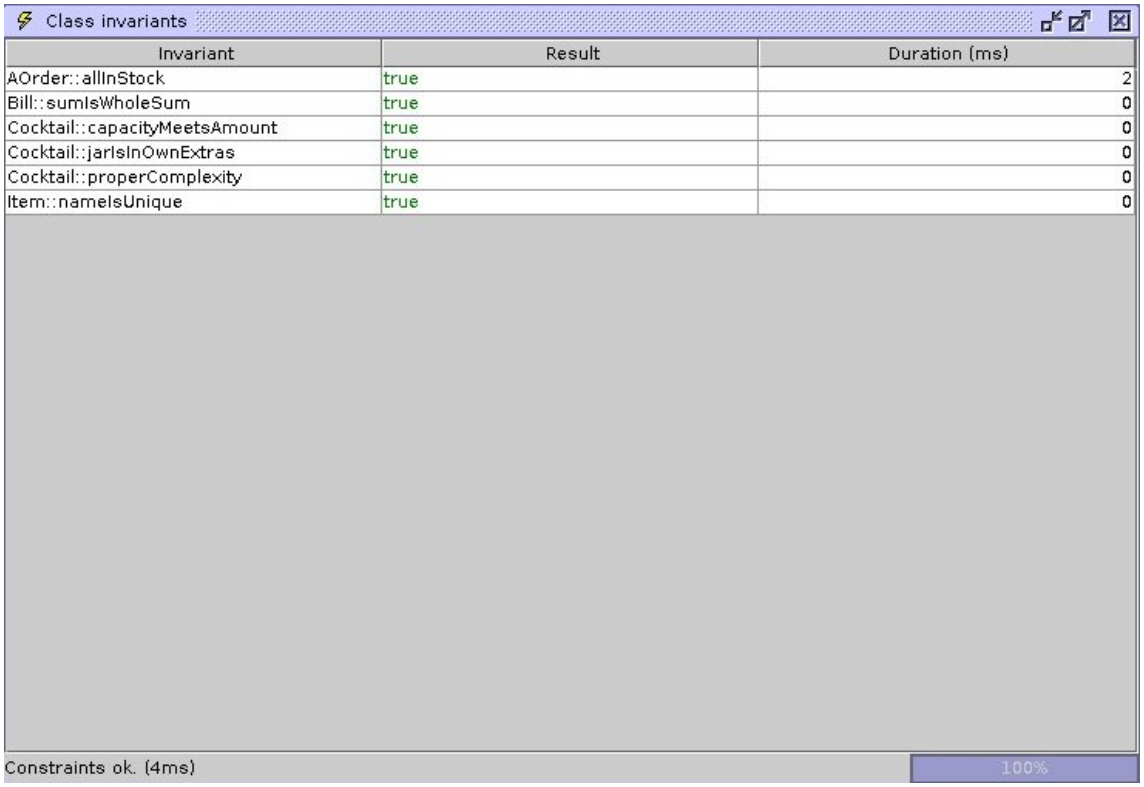


Abbildung 5.3: Klassen Diagramm nach Structure Check

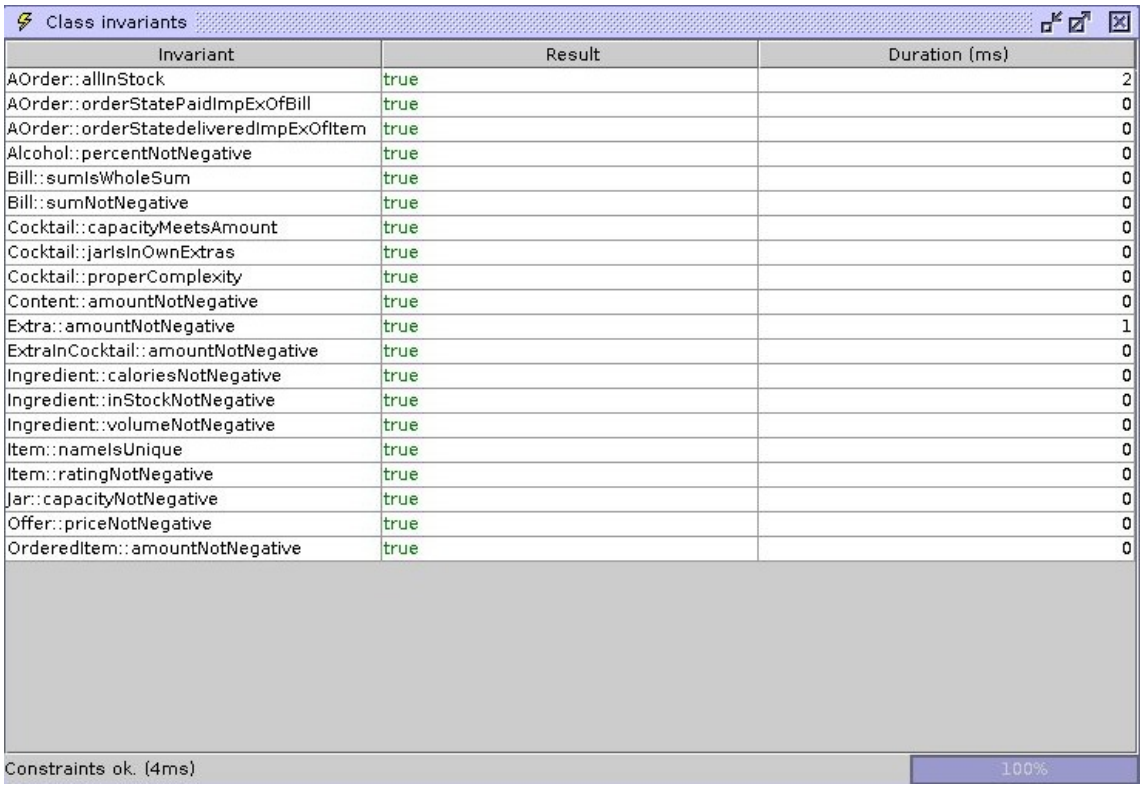
Nachdem die ersten Anfrage- und Manipulations-Funktionen implementiert und der ModelValidator eingesetzt wurde, hat sich die Anzahl der Invarianten verdreifacht.


Class invariants

Invariant	Result	Duration (ms)
AOrder::allInStock	true	2
Bill::sumIsWholeSum	true	0
Cocktail::capacityMeetsAmount	true	0
Cocktail::jarIsInOwnExtras	true	0
Cocktail::properComplexity	true	0
Item::nameIsUnique	true	0

Constraints ok. (4ms) 100%

Abbildung 5.4: Invarianten in Revision 17


Class invariants

Invariant	Result	Duration (ms)
AOrder::allInStock	true	2
AOrder::orderStatePaidImpExOfBill	true	0
AOrder::orderStatedeliveredImpExOfItem	true	0
Alcohol::percentNotNegative	true	0
Bill::sumIsWholeSum	true	0
Bill::sumNotNegative	true	0
Cocktail::capacityMeetsAmount	true	0
Cocktail::jarIsInOwnExtras	true	0
Cocktail::properComplexity	true	0
Content::amountNotNegative	true	0
Extra::amountNotNegative	true	1
ExtraInCocktail::amountNotNegative	true	0
Ingredient::caloriesNotNegative	true	0
Ingredient::inStockNotNegative	true	0
Ingredient::volumeNotNegative	true	0
Item::nameIsUnique	true	0
Item::ratingNotNegative	true	0
Jar::capacityNotNegative	true	0
Offer::priceNotNegative	true	0
OrderedItem::amountNotNegative	true	0

Constraints ok. (4ms) 100%

Abbildung 5.5: Invarianten in Revision 19

Als Anwendungsbeispiele für das System entwickelt waren, um fehlende Funktionen für die Sequenzdiagramme zu ermitteln, schien das System fast fertig. Die folgende Implementierung der fehlenden Funktionen und Invarianten war durch die für uns zuvor unbekannte Programmiersprache SOIL mit der USE/OCL-Kombination am aufwändigsten.

Class invariants

Invariant	Result	Duration (ms)
AOrder::allInStock	true	0
AOrder::exOfBillImpOrderStatePaid	true	1
AOrder::orderStateImpSizeOfItems	true	0
AOrder::orderStatePaidImpExOfBill	true	0
AOrder::orderStateReadyToServeImplie...	true	0
Alcohol::percentNotNegative	true	0
Bill::sumIsWholeSum	true	0
Bill::sumNotNegative	true	0
Cocktail::capacityMeetsAmount	true	0
Cocktail::jarIsInOwnExtras	true	0
Cocktail::properComplexity	true	0
Content::amountNotNegative	true	1
Extra::amountNotNegative	true	0
ExtraInCocktail::amountNotNegative	true	0
Ingredient::caloriesNotNegative	true	0
Ingredient::inStockNotNegative	true	0
Ingredient::volumeNotNegative	true	0
Item::ItemIngredientsBiggerNull	true	0
Item::itemOrderedImpOfferEx	true	0
Item::nameIsUnique	true	0
Item::ratingNotNegative	true	0
Jar::capacityNotNegative	true	0
Offer::priceNotNegative	true	1
OrderedItem::amountNotNegative	true	0

Constraints ok. (3ms) 100%

Abbildung 5.6: Invarianten in Revision 47

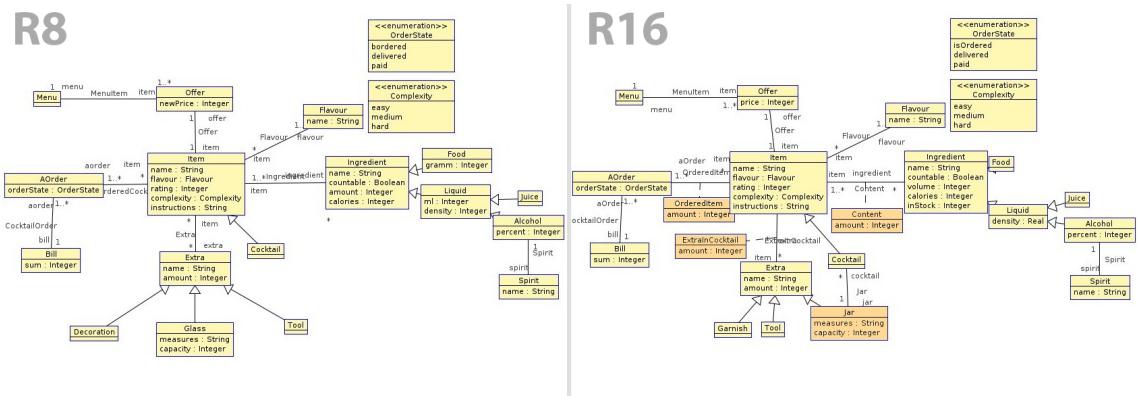


Abbildung 5.7: Vergleich der Revisionen 8 und 16

Ein relativ deutlicher Unterschied lässt sich in den in 5.7 gegenüber gestellten Revisionen erkennen. Eine der größten Änderungen umfasst das Hinzufügen der Assoziationsklassen. Die Notwendigkeit hierfür wurde uns am deutlichsten bei der Assoziation zwischen Item (3.2.6) und Ingredient (3.2.15). Hier ist es für eine Zutatenliste die zu einem Item gehört notwendig, die Menge der jeweiligen Zutat festzuhalten, dies lässt sich am einfachsten in einem Attribut einer Assoziationsklasse realisieren.
Unser erster Versuch sah vor, die Menge in Ingredient(3.2.15) selber im Attribut *amount* zu speichern (siehe Revision 8 in 5.7). Mit dem Nachteil dass jede Ingredient (3.2.15) für jeden Cocktail (3.2.13) neu erzeugt werden müsste, dies hätte zu Redundanzen geführt und entsprach darüber hinaus nicht unseren Vorstellungen. Jede Ingredient (3.2.15) sollte für genau eine Zutat stehen und die verschiedenen Cocktails (3.2.13) auf diese eine Klasse verlinken, da sich in ihr allgemeine Informationen befinden. Die konkreten Werte finden

sich dann in der anschließend eingeführten Assoziationsklasse.
Die gleichen Sachverhalte fanden sich in den Beziehungen in denen weitere Assoziationsklassen von uns eingeführt wurden.
Im Laufe des Semesters wurde kurzzeitig die Idee angeregt unser System neben Cocktails auch auf weiteres Essbares auszuweiten. Hieraus resultierte das Cocktail (3.2.13) aus der Mitte des Diagramms gezogen wurde und dem generischeren Item (3.2.6) weichte. Da wir uns im weiteren Verlauf aber dennoch wieder auf Cocktails beschränkten ist dies unsere einzige erbende Klasse und stand im Zentrum unserer Entwicklungen.
Eine weitere Besonderheit, die uns erst während des Arbeitens mit dem System auffiel ist die Sonderstellung des Jar (3.2.12) bei einem Cocktail (3.2.13). Wir hatten uns als zu formulierende Bedingung vorgenommen, die Menge aller Zutaten auf das maximale Fassungsvermögen des jeweiligen Jar (3.2.12) zu begrenzen. Im ersten Entwurf (siehe Revision 8 5.7) wäre hierzu notwendig gewesen in den Extra (3.2.9) eines Cocktail (3.2.13) nach einem Glas zu suchen. Da es sich aber um ein besonderes Extra (3.2.9) bei einem Cocktail (3.2.13) handelt (da es geradezu immer und genau einmal vorhanden ist), sollte es nicht einfach nur Teil der Menge der Extra (3.2.9) sein, sondern wurde von uns über eine weitere Assoziation Jar (3.3.6) direkt mit ihrem Cocktail (3.2.13) verbunden. Nun war die Invariante Cocktail::capacityMeetsAmount deutlich einfacher und verständlicher zu formulieren.

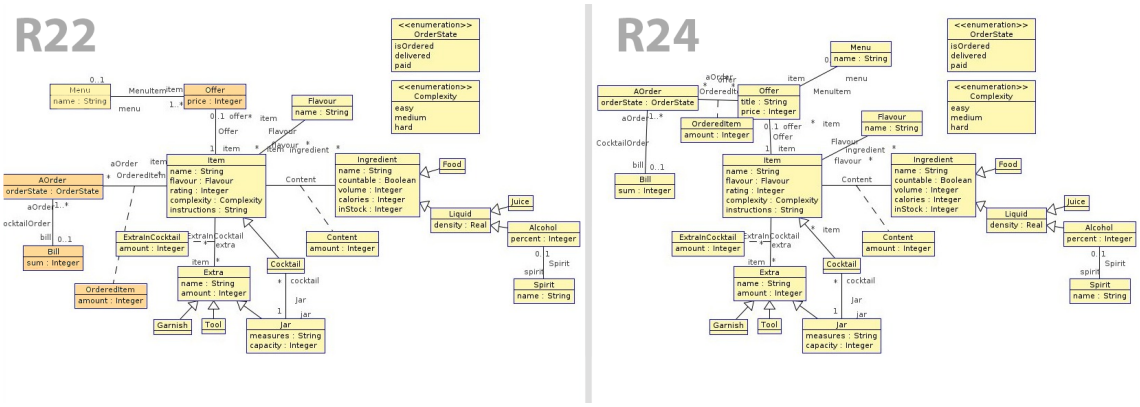


Abbildung 5.8: Vergleich der Revisionen 22 und 24

Ein Vergleich der Revisionen in 5.8 zeigt, dass wir eine größere Umorganisation im Bereich der Bill vorgenommen haben und diese nun eine Menge von Offer (3.2.4) enthält und nicht Item. Hier sind uns Probleme aufgefallen als wir versuchten die gesamte Summe der Rechnung zu berechnen. Es war uns bei der vorherigen Situation nicht möglich festzustellen, welches Angebot für die Bestellung des Item (3.2.6) verantwortlich ist, da ein Item mit verschiedenen Offer (3.2.4) assoziiert sein kann. Dies veranlasste uns dazu die Offer (3.2.4) direkt mit der Bestellung (3.2.2) zu verbinden, so dass eine Bestellung aus Angeboten besteht und somit auch für jedes Item (3.2.6) ein Angebot bestehen muss.
Dies waren einige der Bereiche, in denen wir größere Veränderungen an unserem Modell vorgenommen haben.

Kapitel 6

Anfragen an das System

Im Folgenden erläutern wir unserer Anfrageoperationen (*query operations*) und führen mit jeder eine Anfrage an unser System aus. Operationen die von anderen Operationen als Hilfsoperationen verwendet werden, erläutern wir dabei nicht noch einmal extra, sondern nur in ihrer aufrufenden Funktion. Desweiteren ist davon auszugehen, dass wir es hier mit einem korrekten Systemzustand zutun haben und somit alle Invarianten und Multiplizitäten erfüllt sind.

6.1 Alle Zutaten einer Bestellung anzeigen

Die Operation `getAllIngredients()` aus der Klasse *AOrder* (*class* AOrder, Kapitel 3.2.2) ermittelt alle Zutaten einer Bestellung und gibt diese als Zutat-Menge Tupel zurück.

```
1 getAllIngredients():Set(Tuple(ingred:Ingredient,requiredAll:Integer))=
2         self.orderedItem->collect($e : OrderedItem | ↙
              ↳($e.offer.item.getAllIngredients($e.amount)))->iterate(t; ↙
              ↳acc:Set(Tuple(ingred:Ingredient,requiredAll:Integer))=Set{} |
3                 if acc->exists(t1|t1.ingred=t.ingred) then
4                         let existingT = acc->any(t1|t1.ingred=t.ingred) in
5                         acc->excluding(existingT)->including( ↙
                              ↳Tuple{ingred=existingT.ingred, requiredAll = ↙
                              ↳existingT.requiredAll + t.required})
6                 else
7                         acc->including(Tuple{ingred=t.ingred, ↙
                              ↳requiredAll=t.required})
8                 endif
9                 )
```

Quellcode 6.1: Quellcode von getAllIngredients()

Ein mögliches Ergebnis kann wie folgt aussehen:

```
1 Set{ Tuple{ingred=@ccola,requiredAll=30},
2      Tuple{ingred=@limeslice,requiredAll=6},
3      Tuple{ingred=@mackenstdKorn,requiredAll=30} } : ↙
           ↳Set(Tuple(ingred:Ingredient,requiredAll:Integer))
```

Quellcode 6.2: Ergebnis von getAllIngredients()

6.2 Alle verfügbaren Bestellungen anzeigen

Die Operation `getMenu()` aus der Klasse *Menu* (*class* Menu, Kapitel 3.2.5) sucht alle anhand des Lagerbestand zubereitbaren Angebote heraus und gibt diese dann als Quadrupel, bestehend aus Name, Zutatenliste, Anzahl der noch zubereitbaren Artikel und Preis, zurück. Die Operation baut auf der Operation `getAvailOffers()` auf, welche erst einmal alle Angebote heraussucht die mit dem aktuellen Lagerbestand noch zubereitet werden können.

```
1 getMenu():Set(Tuple( name:String, ingredients:Set(Ingredient), ↙
    ↳available:Real, price:Integer))
2         =self.getAvailOffers()->collectNested($e : Offer| ↙
              ↳Tuple{name=$e.title, ↙
              ↳ingredients=$e.item.getAllIngredientsNames(), ↙
              ↳available=$e.countAvailOffers(), price=$e.price} )->asSet()
```

Quellcode 6.3: Quellcode von getMenu()

Ein mögliches Ergebnis kann wie folgt aussehen:

```
1 Set{ Tuple{available=60.0, ingredients=Set{@ccola, @limeslice, ↙
    ↳@mackenstdKorn}, name='Mackenstedter Libre Offer', price=6},
2   Tuple{available=200.0, ingredients=Set{@ccola, @mackenstdKorn, ↙
        ↳@russStdVodka}, name='AllIn Offer', price=12}} : ↙
        ↳Set(Tuple(available:Real, ingredients:Set(Ingredient), ↙
        ↳name:String, price:Integer))
```

Quellcode 6.4: Ergebnis von getMenu()

6.3 Rechnung erstellen und anzeigen

Die Operation `getBill()` aus der Klasse *Bill* (*class* Bill, Kapitel 3.2.1) erzeugt eine Rechnung.

```
1  getBill():Tuple(list:Bag(Tuple(name:String, description:String, ↙
     ↳count:Integer, sum:Integer)), wholeSum:Integer)
2          begin
3                  declare tempSet:Set(Tuple(name:String, description:String, ↙
                       ↳count:Integer, sum:Integer)), ↙
                       ↳resultSet:Bag(Tuple(name:String, description:String, ↙
                       ↳count:Integer, sum:Integer)), ↙
                       ↳waste:Bag(Tuple(name:String, description:String, ↙
                       ↳count:Integer, sum:Integer));
4                  self.sum := ↙
                       ↳self.aOrder.collect(a:AOrder|a.billOrder())->sum();
5                  for o in self.aOrder do
6                          o.orderState := #billing;
7                          tempSet := o.orderedItem->collectNested($oI : ↙
                               ↳OrderedItem|
8                                   Tuple{name=$oI.offer.title, description = ↙
                                       ↳$oI.offer.item.name, count=$oI.amount,
9                                           sum=$oI.amount*$oI.offer.price})->asSet();
10                         if not tempSet->oclIsUndefined() then
11                             if resultSet->oclIsUndefined() then
12                                 resultSet := tempSet->asBag();
13                             else
```

```
14                                          resultSet := ↙
                                                ↳resultSet->union(tempSet->asBag());
15                                      end;
16                                  end
17                      end;
18                      result := Tuple{list=resultSet, wholeSum=self.sum};
19          end
```

Quellcode 6.5: Quellcode von getBill()

Ein mögliches Ergebnis kann wie folgt aussehen:

```
1 Tuple{list=Bag{Tuple{count=3, description='MLibre', name='Mackenstedter␣↙
    ↳Libre␣Offer', sum=18}}, wholeSum=18} : ↙
    ↳Tuple(list:Bag(Tuple(count:Integer, description:String, name:String, ↙
    ↳sum:Integer)), wholeSum:Integer)
```

Quellcode 6.6: Ergebnis von getBill()

6.4 Alle ausstehenden Bestellungen anzeigen

Die Operation `pendingOrders()` aus der Klasse *AOrder* (*class* AOrder, Kapitel 3.2.2) sucht alle ausstehenden Bestellungen heraus, also Bestellungen mit *OrderState = #isOrdered*.

```
1 pendingOrders():Set(AOrder)=
2         AOrder.allInstances()->select(a | a.orderState = #isOrdered);
```

Quellcode 6.7: Quellcode von pendingOrders()

Ein mögliches Ergebnis kann wie folgt aussehen:

```
1 Set{@Order1} : Set(AOrder)
```

Quellcode 6.8: Ergebnis von pendingOrders()

6.5 Alle servierfertigen Bestellungen anzeigen

Die Operation `readyToServeOrders()` aus der Klasse *AOrder* (*class* AOrder, Kapitel 3.2.2) sucht alle servierfertigen Bestellungen heraus, also Bestellungen mit *OrderState = #readyToServe*.

```
1 readyToServeOrders():Set(AOrder)=
2         AOrder.allInstances()->select(orderState = #readyToServe);
```

Quellcode 6.9: Quellcode von readyToServeOrders()

Ein mögliches Ergebnis kann wie folgt aussehen:

```
1 Set{@Order2} : Set(AOrder)
```

Quellcode 6.10: Ergebnis von readyToServeOrders()

Kapitel 7

Modelvalidation

7.1 Motivation

Die Möglichkeit in *USE* und *OCL* einen Weltausschnitt zu beschreiben geht über das einfache definieren von Klassen hinaus. Es gibt Invarianten, Pre- und Postconditions und zusätzlich die Möglichkeit durch das Erstellen von Objekten ein Weltausschnitt zu instanziieren. USE stellt in der Standard-Installation zwei Möglichkeiten zur Verfügung einen Objektzustand gegen das Modell zu validieren. Der *Structure Check* und die Möglichkeiten Invarianten zu prüfen. Eine dritte Möglichkeit ist der Validator *KodKod*, der erst als Plugin installiert werden muss.

7.2 Structure Check

Der Structure Check ermöglicht das Validieren eines existierenden Objektzustands gegen das Modell. Anders als bei den Invarianten werden beim *Structure Check* Verletzungen der definierten Multiplizitäten angezeigt, wenn der Objektzustand diese verletzt.

7.2.1 Methode

Es werden von Hand gültige Objektzustände erstellt und diese gegen das Modell geprüft

7.2.2 Ergebnisse

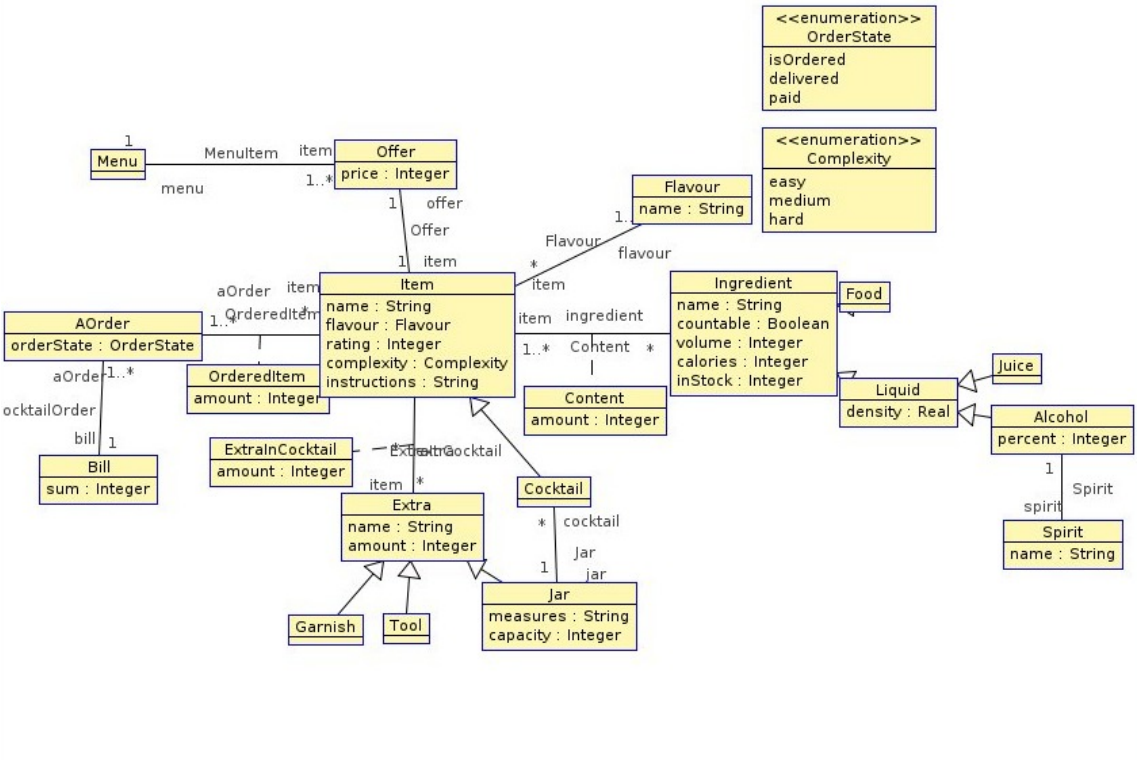


Abbildung 7.1: Klassen Diagramm vor Structure Check

Nachdem der erste Weltausschnitt in Form eines Objektzustand erstellt wurde, deckte der *Structure Check* einige Fehler in den Multiplizitäten auf. So war es z.B. nur möglich ein *Item* (*class* Item, Kapitel 3.2.6) zu erstellen, wenn man auch gleichzeitig ein zugehöriges *Offer* (*class* Offer, Kapitel 3.2.4) und *Menu* (*class* Menu, Kapitel 3.2.5) erstellt und diese dem *Item* (*class* Item, Kapitel 3.2.6) zuweist. Ebenso mussten zu einem *Item* auch immer eine *AOrder* (*class* AOrder, Kapitel 3.2.2) und eine zugehörige Rechnung (*class* Bill, Kapitel 3.2.1) existieren. Hier hat sich gezeigt das viel zu häufig 1:1 Multiplizitäten bzw. solche mit einer Kardinalität < 1 (von größer als eins) angewendet wurden. Durch das Erzeugen von Objektzuständen, die einen gültigen Systemzustand widerspiegelten, wurden diese und ähnliche Fehler durch den *Structure Check* aufgedeckt. Das zwingende Fordern eines zugehörigen Objektes führt dazu, dass viele temporäre Weltausschnitte ungültig werden. Ob diese Zustände dann in der Praxis gültig sein müssen, ist wiederum eine nicht zu verallgemeinernde Design- und Implementierungsfrage.

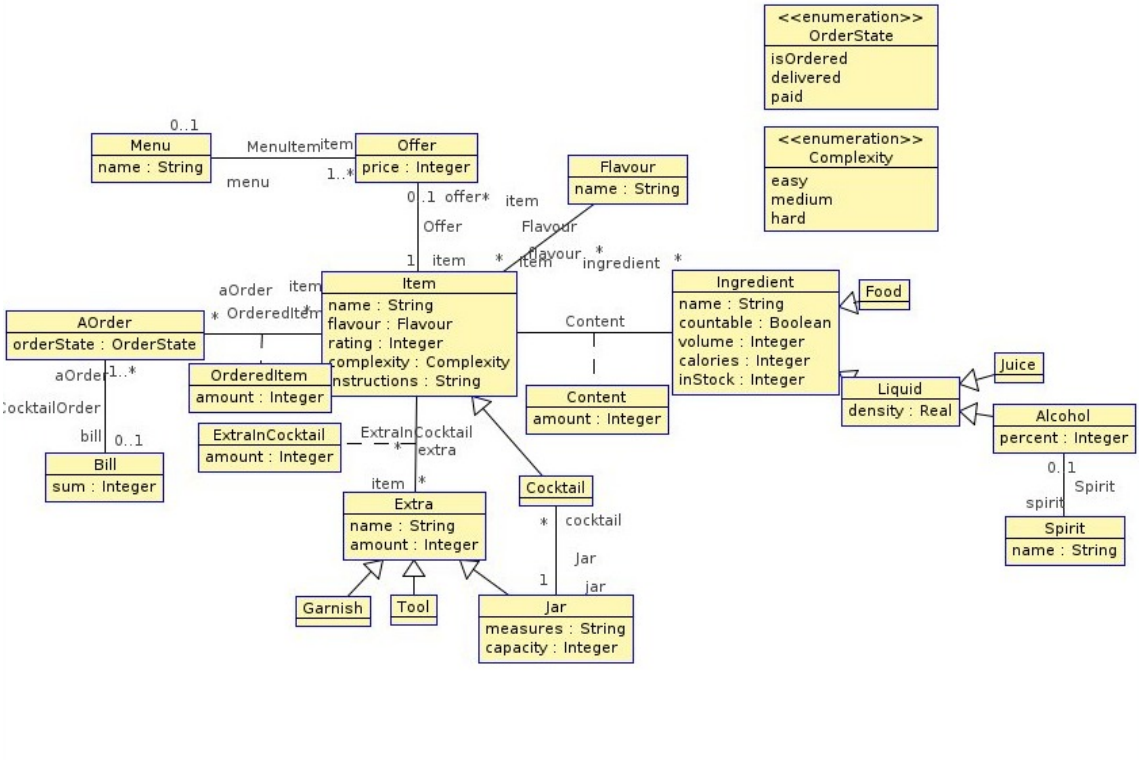


Abbildung 7.2: Klassen Diagramm nach Structure Check

7.3 KodKod

Der Validator KodKod wird als Plugin installiert und integriert sich vollständig in die USE Umgebung. Anders als beim *Structure Check* und dem Prüfen von Invarianten gegen ein existierendes Modell wird mit KodKod irgendein gültiger Objektzustand generiert der die Spezifikation erfüllt.

7.3.1 Methode

Der von KodKod generierte Objektzustand wird "von Hand" analysiert und entschieden ob der Objektzustand gültig sein soll oder nicht. Ist dies nicht der Fall, muss die OCL Spezifikation angepasst werden. Da gültige Objektzustände algorithmisch durch KodKod mit einem SAT-Solver berechnet werden, sind die Ergebnisse deterministisch, also für gleiche Eingabegrößen ebenfalls gleich. Um das Verhalten zu beeinflussen ist es möglich lokale und globale Mini- und Maxima anzugeben. Die globalen Maxima beschränken sich auf *Real*, *Integer* und *String* Werte. Die Lokalen beschränken sich auf das jeweilige Attribut einer Klasse unabhängig davon von welchem Typ dieses ist. Diese Grenzen werden, wenn es Invarianten fordern, überschritten und ein Objektzustand erzeugt der laut KodKod "SATISFIABLE" ist. Die Konfigurationsdatei erlaubt ebenfalls das Festlegen von "steps", die die Schrittgröße, für das Probieren von Werten der einzelnen Datentypen für den SAT-Solver festlegt. Dies geschieht wieder global und nicht für einzelne Attribute einer Klasse. Durch diese Limitierungen ist es erforderlich, KodKod durch zusätzliche Invarianten in die richtige Richtung zu lenken. Soll sichergestellt sein, dass ein bestimmter Zustand nicht erreicht werden kann, wird eine Invariante definiert die genau den nicht gewollten Zustand fordert. Wenn es dann keine Objekt-Instanziierung die diesen Zustand erfüllt, gibt es eine Invariante die diese Forderung verhindert und das Modell ist gegen diesen Zustand gesichert (z.B. das Ausliefern eines Cocktails ohne Zutaten). Ein Problem bei der Steuerung

durch Invarianten ist, dass nicht jeder OCL Ausdruck in SAT Logik übersetzt werden kann. Teilweise erhält man gute Fehlermeldungen, welche Ausdrücke nicht unterstützt werden.

```
1 ERROR InvariantTransformator: Cannot transform invariant ↙
    ↳'Offer::ItemIngredientsAvailableOffers'. Iterate not supported
```

Das gilt allerdings nicht für alle Operationen.

```
1 ERROR InvariantTransformator: Cannot transform invariant ↙
    ↳'Offer::ItemIngredientsAvailableOffers'. null
```

Hierdurch haben sich weitere Limitierungen ergeben. Mehr Invarianten und abhängige Attribute sind die "schlimmsten" Eingabegrößen, wenn es um den Aufwand und somit um die Dauer der Validierung geht.

7.3.2 Validierung

Durch die angesprochenen Limitierungen und den hohen Aufwand wurde das Modell Stück für Stück validiert, indem am Anfang von jeder Klasse genau ein Objekt erzeugt wurde und iterativ für jede Klasse die Anzahl der Objekte erhöht und dann auf ungültige Zustände analysiert wurde. Auf Grund des exponentiell steigenden Aufwands und dem erhöhten zeitlichen Aufwand wird unser Modell nicht an allen Stellen bis zur Erschöpfung validiert.

Beispiel

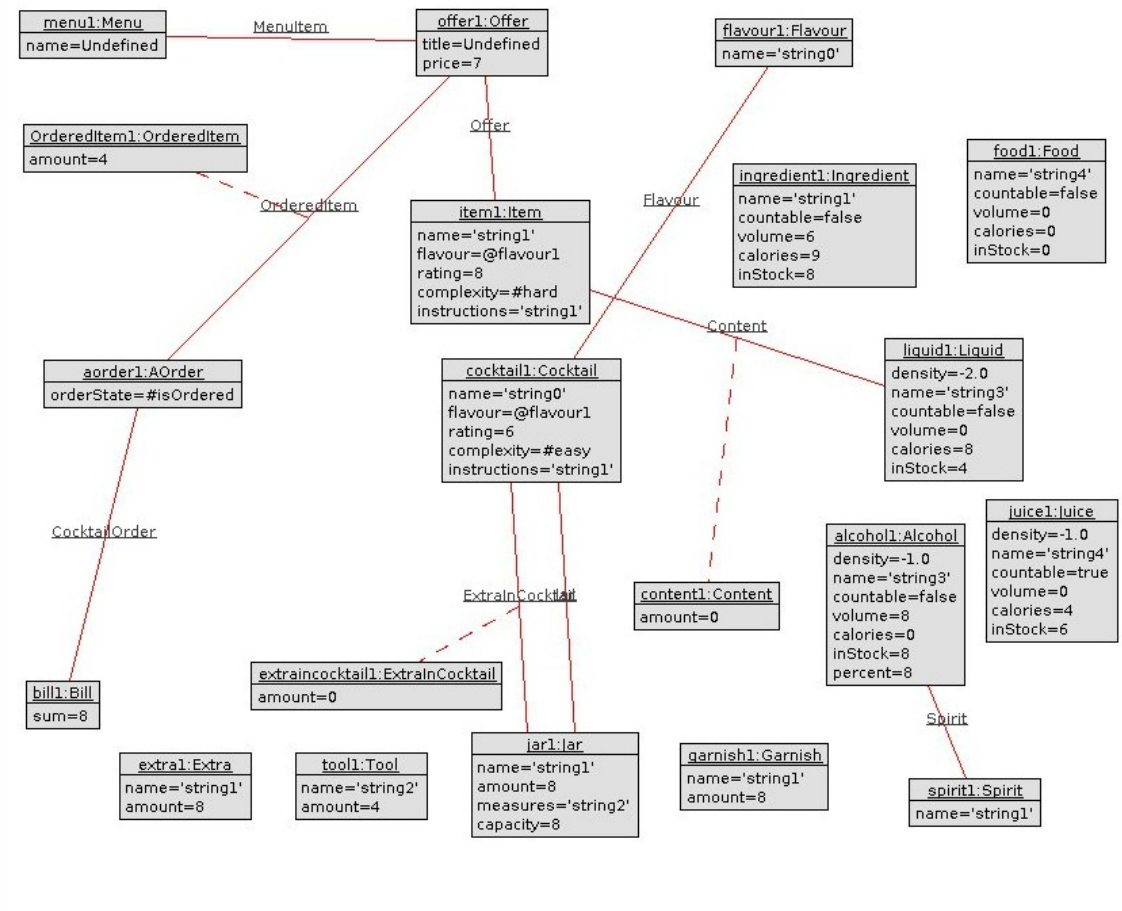


Abbildung 7.3: KodKod Validierung: Ausgangszustand

In Abbildung 7.3 ist die Ausgangssituation dargestellt. Gibt es für jede Klasse ein instanziiertes Objekt, liefert KodKod einen Zustand der gültig ist. Wird die Anzahl der Objekte für

die Klasse *AOrder* (*class* AOrder, Kapitel 3.2.2) erhöht, zeigt sich, dass die erforderlichen Verbindungen immer noch vorhanden sind, jedoch eine Rechnung für eine Order existiert, die keine Bestellungen in ihrer *Order* hat (Eine Rechnung ohne Bestellungen).

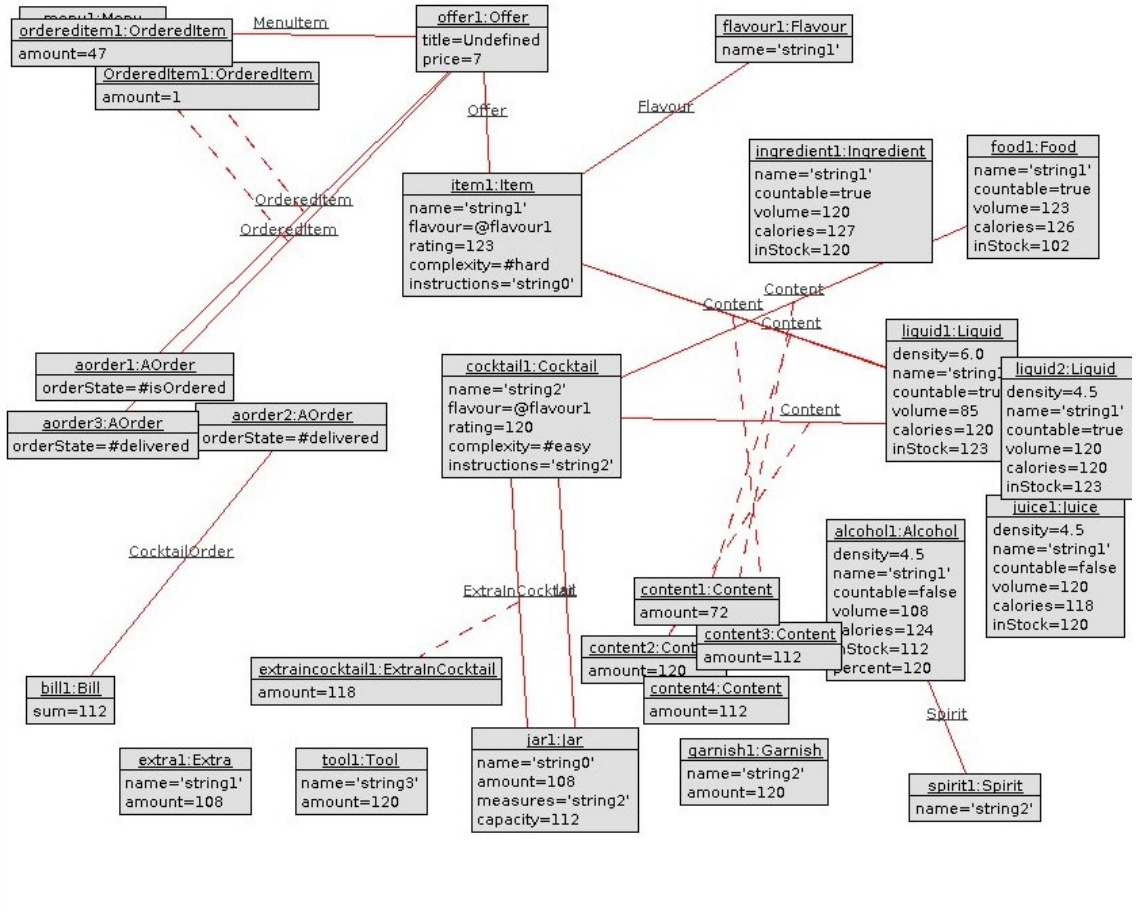


Abbildung 7.4: KodKod Validierung: fehlende Invarianten

Es zeigt sich, dass die Forderung für ein zugehöriges "Partner-Objekt" für genau ein Objekt einer Klasse ausreichend ist. Sobald aber zwei existieren, kann eine Lücke wie in Abbildung 7.4 entstehen. Hier muss gefordert werden, dass jede *AOrder*-Instanz für die eine *Bill* (*class* Bill, Kapitel 3.2.1) existiert, auch eine *Offer* (*class* Offer, Kapitel 3.2.4) zugeordnet sein muss.

Ein anders Beispiel stellt die Invariante *SumOfBillIsHoleSum* in der Klasse *Bill* dar, die fordert dass die Summe der Rechnung gleich der Summe aller Bestellungen ist. Beim Validieren findet KodKod einen einfachen erfüllenden Zustand. Dies ist genau dann der Fall wenn ein *Item* (*class* Item, Kapitel 3.2.6) bestellt wurde, aber genau 0 mal. Dann gilt diese Invariante für alle nur erdenklichen Zustände, da die Summe aller Items gleich 0 ist. Gibt man KodKod in der Konfigurationsdatei einen Minimalwert von 1 für dieses Attribut mit, wird dieses vernachlässigt und das Modell trotzdem als "SATISFIABLE" angezeigt.

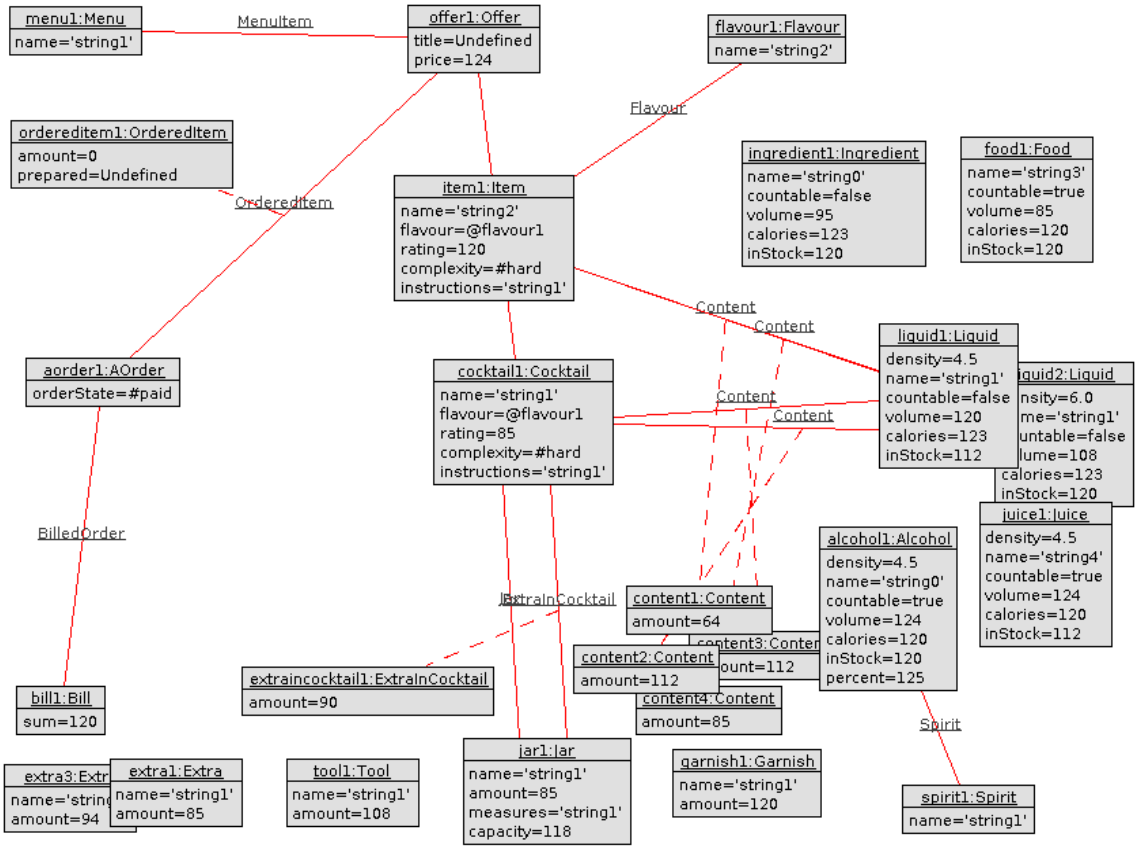


Abbildung 7.5: KodKod Validierung: Objektzustand der die Spezifikation erfüllt

An dieser Stelle ist es wie im Abschnitt Methode beschrieben nötig mit Invarianten die gewünschten Werte zu erzwingen. Bei großen Größen wird das Modell irgendwann als "SATISFIABLE" angezeigt, obwohl beim Prüfen der Invarianten in der USE GUI diese nicht vollständig erfüllt sind, obwohl diese in SAT Logik übersetzt werden können. Hier wieder am Beispiel *SumOfBillIsHoleSum*.

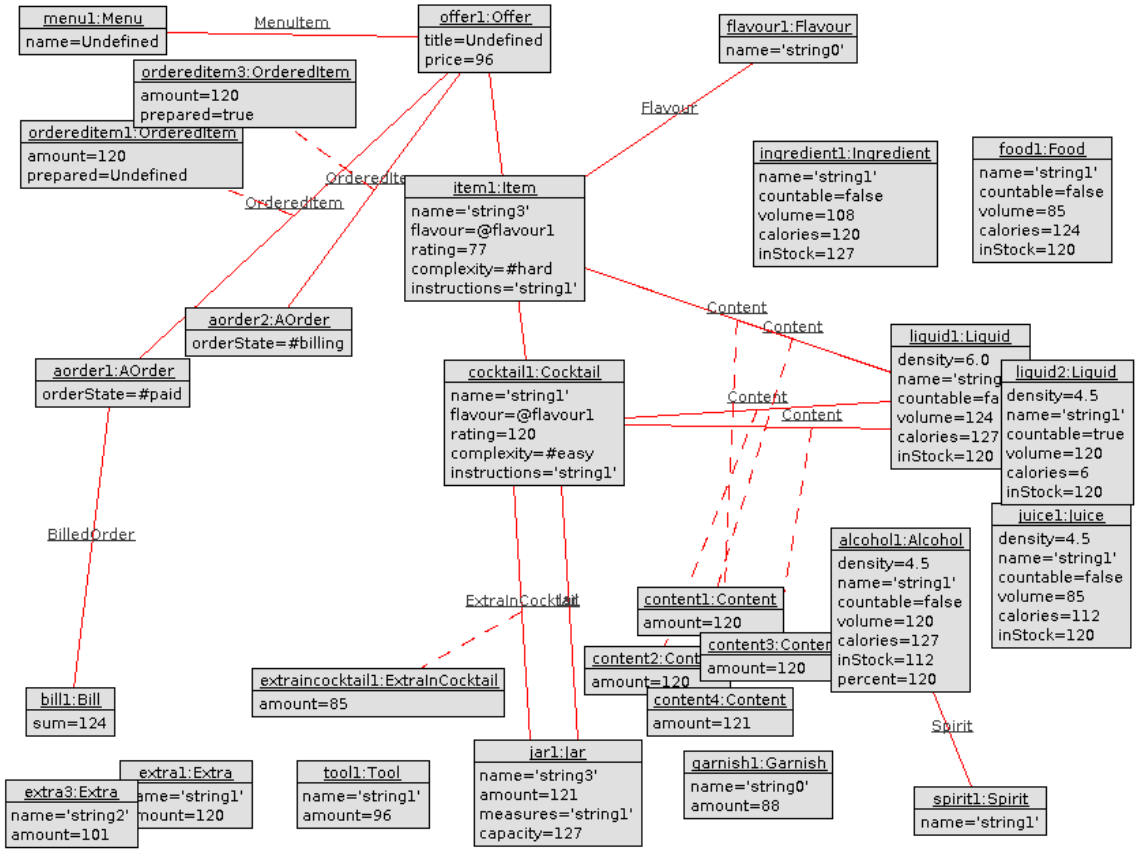


Abbildung 7.6: KodKod Validierung: Objektzustand der die Spezifikation nicht erfüllt

7.3.3 Ausblick

Wie bereits erwähnt, wurde im Rahmen dieser Arbeit keine erschöpfende Validierung gefordert. Durch die angesprochenen Limitierungen und den zeitlichen Aufwand ist es durchaus möglich eine tiefer gehende Validierung durchzuführen. Beim Validieren wurden nie, von mehr als drei Klassen, mehr als ein Objekt erzeugt. Die einstellbaren Grenzen in der Konfigurationsdatei von KodKod wurden wie im Abschnitt Validierung beschriebenen nur noch marginal benutzt und verändert. Nicht alle Invarianten können in SAT-Logik übersetzt werden. So mussten Invarianten teilweise angepasst oder Workarounds entwickelt werden, die ähnliche Bedingungen fordern oder für dieses Szenario ausreichende Bedingungen garantieren. Die benutzte USE Version 3.0.6 ist inzwischen veraltet, von einem Wechsel wurde uns allerdings abgeraten, da sich viele Eigenschaften geändert haben. Die Perspektive weitere Validierungen durchzuführen ist also durchaus vorhanden.

Kapitel 8

Fazit

8.1 Arbeit

Unser Modell ist im Rahmen eines Lernprozesses entstanden und beschreibt einen kleinen Ausschnitt eines vollwertigen Systems. So sind nicht alle Begebenheiten so scharf formuliert das sie praxistauglich sind. Für unser Modell konnten erfolgreich Invarianten, Abfrage- und Manipulations-Funktionen implementiert werden. Die Invarianten konnten so scharf formuliert werden, dass ein durch ein SAT-Solver generierter Zustand, einen gültigen Zustand gemäß unserer Vorstellungen ergeben hat. Hierfür mussten auch sehr triviale Invarianten definiert werden. In Rahmen der Sequenzdiagramme konnten sinnvolle Anwendungsfälle implementiert werden.

8.2 USE - UML-based Specification Environment

Zu Beginn unserer Ausarbeitung standen wir vor Begriffen wie OCL, USE und SOIL. Diese kannten wir bisher nur aus der Theorie und hatten sie zuvor in der Praxis noch nicht angewendet. Während der Arbeit mit der USE-Umgebung wurden unsere zunächst verhaltenen Vorstellungen aber positiv übertroffen. Das Zusammenspiel von Klassen- und Objektdiagrammen in einer IDE stellten wir uns zunächst ein wenig trocken vor, durch die tatsächliche praktische Relevanz und die nützlichen Validierungsmöglichkeiten stellte sich die Arbeit damit aber letztendlich als sehr nützlich heraus.

Mit Programmiererfahrung, mathematischem Mengenverständnis, und Kenntnissen in einer Datenbank-Abfragesprache verfügten wir bzgl. dieser Veranstaltung schon über reichlich Vorwissen; dass OCL eine reine Abfragesprache, OCL in SOIL aufgerufen werden kann, aber nicht umgekehrt und dass ausschließlich mit SOIL Objekte manipuliert werden können, wurde jedoch erst Schritt für Schritt erkenntlich. So waren Nachfragen beim Tutor teilweise übereilt, da uns am Anfang die Grundlagen fehlten. Andererseits war uns bspw. die Syntax oft nicht sofort klar, so dass wir durch einfache Fehler hierbei relativ viel Zeit verloren haben. Dass das Benutzen einer lokalen Variable nach einer for-schleife das Beenden jener mit “end;” statt “end” erfordert, war eines der Probleme, welches uns Zeit gekostet hat. Hier hätte uns ein kleines Kompendium, welches diese Informationen zusammenfasst bereitstellt, sicherlich weiter geholfen. Waren diese Schwierigkeiten aber beseitigt, konnten wir einiges mit USE anstellen. Mit den gewonnenen Erkenntnissen, aber auch gerade durch das Erarbeiten, hat die Arbeit mit USE und dem eingesetzten Technik-Mix einiges an Spaß bereitet. Die Möglichkeit OCL-Code für die Generierung von Programmcode zu benutzen, gekoppelt mit den Modellierungs- und Validierungsmöglichkeiten, hat unser Interesse in diesem Gebiet geweckt und motiviert, sich auch weiter mit USE zu befassen.

Abbildungsverzeichnis

Quellcodeverzeichnis

Literaturverzeichnis

[1] AG-Datenbanksysteme: *USE - A UML based Specification Environment*. Universität Bremen, 2007.

[2] Anika Bischoffs, Frank Hilken, Sebastian Langer und Danny Zitzmann: *Ausarbeitung zum Thema Schulungsverwaltung*. Entwurf von Informationssystemen, Universität Bremen, 2012.

[3] Bulgac, Vasile: *Green Alcohol Cocktail*, 2013. http://www.sxc.hu/photo/1422791.

[4] Büttner, Fabian: *Reusing OCL in the Definition of Imperative Languages*. 2011.

[5] Carmen Avila, Yoonsik Cheon: *OCL 2.0 QUICK REFERENCE*. University of Texas at El Paso, 2008.

[6] Detmers, Michael: *Hausarbeit zum Thema Storage*. Entwurf von Informationssystemen, Universität Bremen, 2009.

[7] Edvin Pehlivanovic, Roman Asendorf: *Hausarbeit zum Thema Ligaverwaltung*. Entwurf von Informationssystemen, Universität Bremen, 2007.

[8] Kai Michael Poppe, Bastian Kammann: *Hausarbeit zum Thema CarRentalAgency*. Entwurf von Informationssystemen, Universität Bremen, 2008.

[9] Lars Hamann, Fabian Büttner: *SOIL*, 2013. http://sourceforge.net/apps/mediawiki/useocl/index.php?title=SOIL.