Storage

Hausarbeit im Rahmen des Kurses

Entwurf von Informationssystemen
bei Prof. Dr. Martin Gogolla

im Sommersemester 2009

Michael Detmers
2144612
mdetmers@tzi.de

7. Juli 2009

Inhaltsverzeichnis

1 Vorwort

1.1 Einleitung

Mit dieser Hausarbeit soll ein Informationssystem mit Hilfe von UML[1] in USE[2] beispielhaft umgesetzt werden. Mit UML lassen sich Modelle mit Klassen, Attributen und Operationen strukturieren, wobei OCL[3] bei der weiteren Definition von Regeln in Form von Invarianten und speziellen Bedingungen behilflich ist.

1.2 Zur Dateistruktur

1.2.1 Welches Layout verwenden?

Wenn es zu einem Zustand eine ähnlich benannte Datei gibt, die nur am Ende mit einem „*.layout.olt“ versehen ist, ist darunter das entsprechende Layout gespeichert.

Ist keine solche Datei vorhanden, zum Beispiel bei den Testfällen, findet sich ein Layout, dass man mit minimalem Aufwand anpassen kann, unter dem Basiszustand.
Beispiel: „tc0*_deleteStored.command“ gehört zum Layout „storage.deleteStored.layout.olt“.

1.2.2 Was sind das für Dateiendungen?

- ***.command** ist immer eine Form von Operationsimplementierung. Sei es ein Test oder die grundlegende Implementierung einer Funktion.
  `Command`-Dateien können auch Zustände laden und/oder andere `Command`- Dateien aufrufen und damit die Ausführung von Tests gestatten.

  Die Dateiendung wurde so benannt, weil man über viele E-mail-Provider keine Archive mit `cmd`-Dateien verschicken kann.

1 Unified Modeling Language
2 UML-based Specification Environment ist erhältlich unter
`http://www.db.informatik.uni-bremen.de/projects/USE/`
oder
`http://sourceforge.net/projects/useocl/`
3 Object Constraint Language

- ***.state** ist immer ein Zustand, der erzeugt werden kann. Jeder Zustand setzt den vorher geladenen zurück.
  Diese Dateiendung habe ich gewählt, um mich besser im Dateichaos zurechtzufinden.

1.2.3 Das „Who is Who“ der Dateien

. . . findet sich im Dateiverzeichnis auf Seite 88.

1.3 Verwendete USE-Version

Version 2.5.0 mit installiertem Hotfix

2 Das Klassendiagramm

2.1 Systembeschreibung

Das für diese Hausarbeit ausgearbeitete Modell hat einen praktischen Hintergrund und beschreibt eine Lagerverwaltung für Elektroartikel.
Dieses Lager besteht aus vielen Lagerorten, welche jeweils eine bestimmte Anzahl von einem Artikel beinhalten. Dieser Artikel gehört in eine bestimme Kategorie, zum Beispiel eine Energiesparlampe in die Kategorie Leuchtmittel, ist von einem bestimmten Hersteller produziert worden und wird von mindestens einem Lieferanten angeboten. Des Weiteren soll jeder Artikel durch Tags beschrieben werden.

Jeder Hersteller vergibt seinen Produkten Artikelnummern und jeder Lieferant ordnet jedem Artikel Bestellnummern und einen Preis zu. Verwaltungstechnisch erhält jeder Artikel im Lager zusätzlich seine eigene Produktnummer.

Schlussendlich soll jedes Produkt und Lager das Datum der letzten Änderung beinhalten. Das Verwaltungssystem sollte über die üblichen Funktionen verfügen, also Produkte finden, ändern, löschen und erstellen, Lagerbestände verschieben, ändern und löschen können, et cetera.

2.2 Das Diagramm

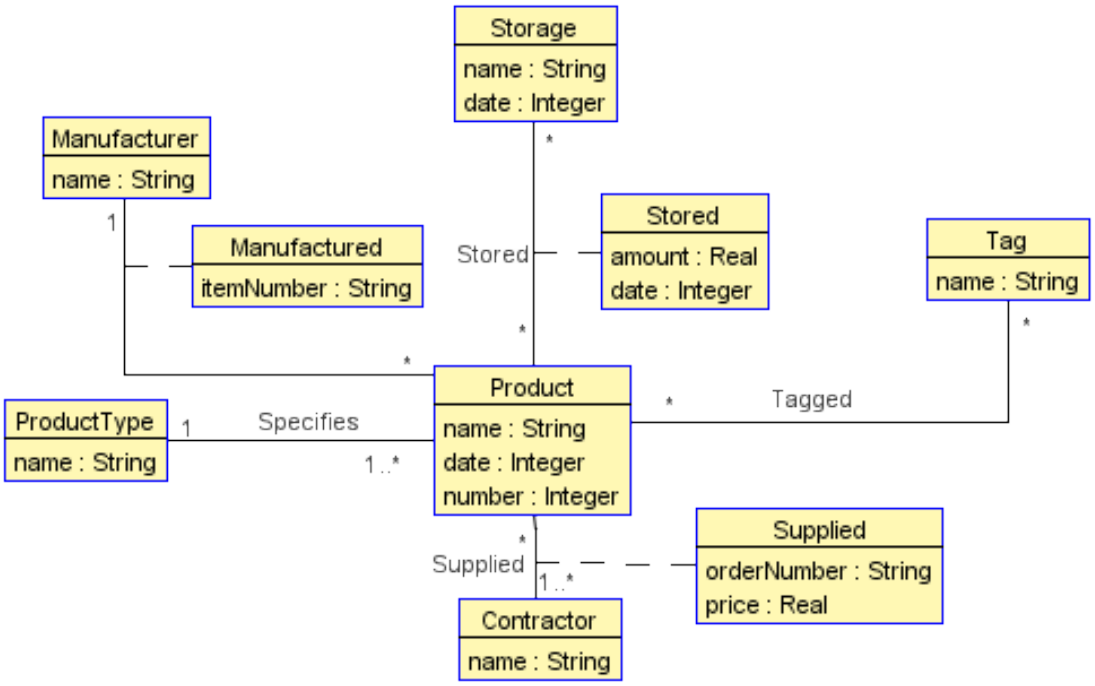

Abbildung 2.1: Das Klassendiagramm in USE. Nur Multiplizitäten und Attribute werden angezeigt.

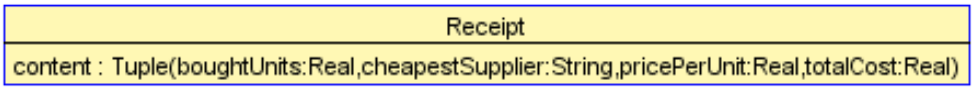

Abbildung 2.2: Die Hilfsklasse „Receipt“ wurde für die Operation *restock()* eingeführt.

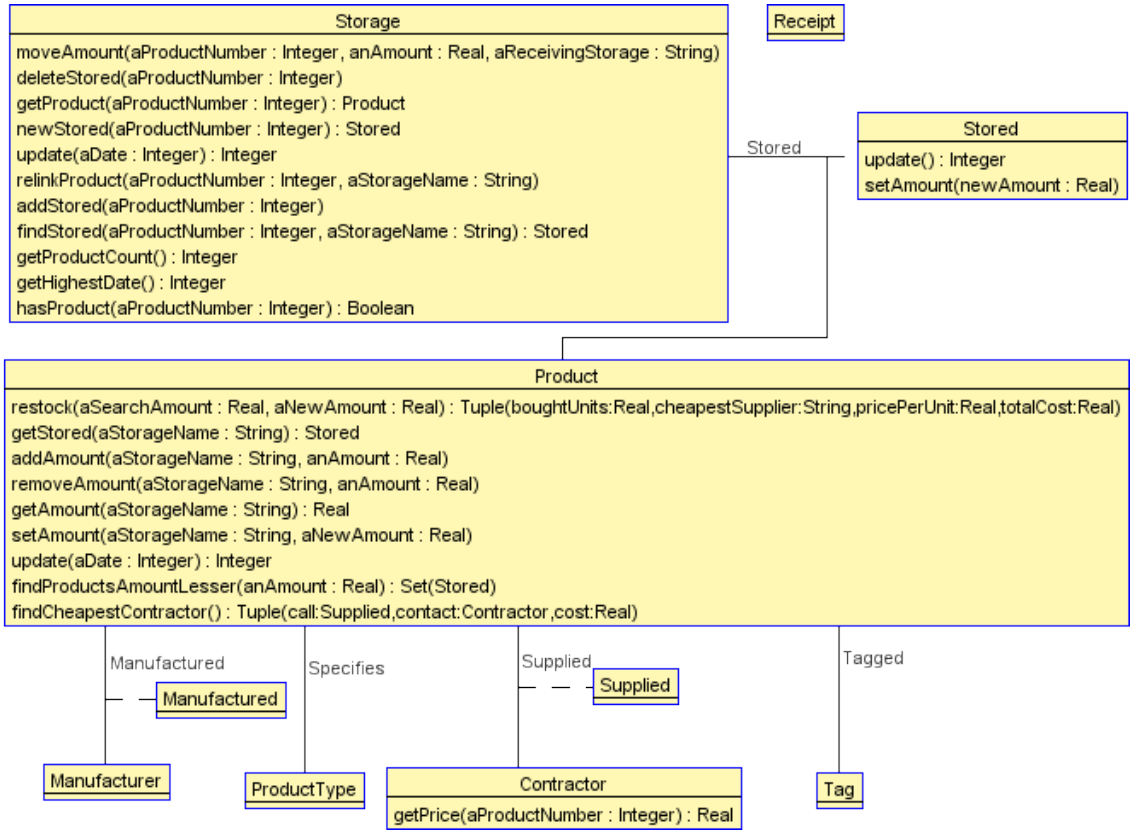

Abbildung 2.3: Das Klassendiagramm in USE. Nur die Operationen werden werden angezeigt.

2.2.1 Klassen, Attribute und Operationen

Anmerkung zu den *date*-Attributen:
Da es in USE keinen Typ für Datumsangaben gibt, nehme ich mir hier die Freiheit, das mi einer `Integer`-Darstellung zu simulieren. Dabei soll das Datum rückwerts als Ganzzahl gespeichert werden. Also wird aus der Darstellung $Tag.Monat.Jahr$ die neue Datumseingabe $JahrMonatTag$, aus dem 12.03.2005 wird damit 20050312.
In den aufgeführten Szenarien und Zuständen wird der Einfachheit halber nur ein niedriger `Integer`-Wert benutzt, der, da es keine Möglichkeit gibt, das heutige Datum zu bestimmen, bei Aktualisierung inkrementiert wird.

Die wichtigsten Operationen dieser Hausarbeit sind *deleteStored()*, *moveAmount()* und *restock()*. Es wurde darauf verzichtet, alle denkbaren Operationen zu deklarieren und/oder zu implementieren. Einige Hilfsfunktionen wurden jedoch eingeführt, um der besseren Anschauung in den produzierten Sequenzdiagrammen oder der bequemeren Operationsab-

wicklung zu dienen.

Es folgen die Klassen und Assoziationsklassen alphabetisch sortiert. Innerhalb dieser sind Attribute und Operationen in Anzeigereihenfolge aufgeführt. Operationen sind durch ein Klammerpaar gekennzeichnet, Invarianten aufgrund ihres Bool'schen Charakters mit einem Fragezeichen.

Contractor

... beschreibt den Lieferanten, beziehungsweise die Bezugsquelle. Diese Klasse ließe sich eventuell noch durch Adressdaten oder ähnliches erweitern.

name gibt die Bezeichnung des Lieferanten als `String` an. Da es jede Bezugsquelle nur einmal gibt, muss auch diese Bezeichnung eindeutig sein und darf damit nicht wieder als Lieferatenname benutzt werden.

Manufactured

... ist die Assoziationsklasse, die den Hersteller mit seinem Erzeugnis verknüpft. Dabei hat jedes Produkt genau einen Hersteller und jeder Hersteller beliebig viele Produkte.

itemNumber gibt dabei die vom Hersteller bestimmte Artikelnummer als `String` an, da Buchstaben und Sonderzeichen als Artikelnummer zu erwarten sind. Des Weiteren darf jeder Hersteller dieselbe Artikelnummer nur einmal verwenden.

Manufacturer

... beschreibt den Hersteller.

name ist ein `String`, der den Namen des Herstellers angibt. Ein Herstellername muss selbstverständlich einzigartig sein.

Product

... beschreibt ein Produkt. Dieses lässt sich durch den Produktnamen, die verknüpften Tags, den Hersteller, die Lieferanten, den Produkttypen sowie der internen Produktnummer beschreiben. Auch wenn zum Beispiel zwei Dimmschalter an sich identisch sind, den selben Namen tragen, die selben Tags tragen und von denselben Lieferanten angeboten werden, muss ein zweites *Product*-Objekt angelegt werden, wenn die Hersteller oder deren Artiklenummern sich unterscheiden.

name gibt die Bezeichnung des Produkts als `String` an.

date gibt das Datum der letzten Änderung an diesem Produkt als `Integer` an.[1]

number beschreibt die Produktnummer als `Integer`, die intern mitgezählt werden soll. Jedes Produkt soll demnach seine eigene Nummer erhalten und daran eindeutig identifizierbar sein.

restock() ist eine der drei für diese Hausarbeit relevanten Operationen. Aufgabe dieser Funktion ist, alle Lagerbestände des aufrufenden Produkts daraufhin zu untersuchen, ob die gelagerte Menge kleiner ist, als die übergebene, erste `Real` und die Treffer, bei denen das der Fall ist, auf die zweite angegebene `Real` aufzustocken. Zusätzlich soll der Preis für das Lageraufüllen ausgehend vom günstigsten Lieferanten ermittelt werden und als dieser zusammen mit der gekauften Stückzahl, dem Einzelpreis und dem günstigsten Lieferanten als `Tuple` ausgegeben werden.[2]

getStored() ist eine Hilfsfunktion, die unter Angabe des gewünschten Lagernamens die verbindende Assoziationsklasse *Stored* zurückliefert.

addAmount() ist eine Hilfsfunktion, die die zu einem mitgeliefertem Lagernamen gehörige Lagermenge um die ebenfalls übergebene `Real` zu erhöhen.

1 Siehe dazu 2.2.1 auf Seite 5
2 Siehe auch im Dateiverzeichnis Seite 88

removeAmount() ist eine Hilfsfunktion, die die zu einem mitgeliefertem Lagernamen gehörige Lagermenge um die ebenfalls übergebene `Real` zu verringern.

getAmount() ist eine Hilfsfunktion, die zum übergebenen Lagernamen die Lagermenge ausgibt.

setAmount() ist eine Hilfsfunktion, die die zum übergebenen Lagernamen gehörige Lagermenge auf den ebenfalls übergebenen `Real`-Wert zu setzen.[1]

update() soll das Produktdatum aktualisieren, sollte das übergebene Datum größer sein, als das aktuelle. Diese Funktion ist so angelegt, da die hier verwendeten Testfälle auch nur mit Testdaten laufen und kein „Heute“ bestimmt ist.[2]

findProductsAmountLesser() ist eine Hilfsfunktion, die zu diesem Produkt gehörige Lagerbestände, die kleiner sind als der übergebene Parameter, als `Set` von *Stored* zurückgibt.

findCheapestContractor() ist eine Hilfsfunktion, die die Lieferassoziation *Supplied*, den darin enthaltenden Preis als `Real` und den dazugehörigen Lieferanten *Contractor* als Tupel zurückgibt. Es wird, wie der Funktionsname erkennen lässt, die Lieferverbindung mit dem günstigsten Preis gesucht.

ProductType

... beschreibt die Produktkategorie. So könnte der Produkttyp „Leuchtmittel“ Produkte wie eine normale Glühbirne und eine Energiesparlampe kategoriesieren.

name ist die `String`-Beschreibung der Produktkategorie. Diese sollte eindeutig sein und nicht noch einmal vorkommen.

1 Siehe auch im Dateiverzeichnis Seite 88
2 Siehe dazu 2.2.1 auf Seite 5

Receipt

... ist eine für die Operation *restock()* eingeführte Klasse, die Rechnungsdaten angibt und nur aufgrund von Funktionalität und Veranschaulichung beibehalten wird.

content ist ein `Tuple`, dass die Stückzahl, die Gesamtkosten, die Stückkosten als `Real` und den günstigsten Lieferantennamen als `String` abspeichert.

Storage

... beschreibt ein Lager, beziehungsweise einen Lagerort. Auch diese Klasse könnte eventuell um Adressdaten oder Zusatzbeschreibungen ergänzt werden.

name ist die eindeutige Lagerbezeichnung als `String`.

date gibt den letzten Stand des Lagers an.[1]

moveAmount() ist eine der drei für diese Hausarbeit relevanten Hauptfunktionen. Sie soll bestimmte Mengen eines bestimmten Produkts aus dem aufrufendem Lager in ein angegebenes verschieben. Bei diesem Prozess gibt es vier Fälle: (1) Ein Teil der Lagermenge wird verschoben. (2) Ein Lagerbestand wird gelöscht und der Inhalt einem bestehenden hinzugefügt. (3) Ein Teil der Lagermenge wird verschoben und eine neue Lagerbeziehung muss erstellt werden. (4) Die gesamte Lagermenge wird beim aufrufenden Lager gelöscht und beim Ziellager neu erstellt. Alternativ zum letzten Punkt könnte auch die gesamte Lagerbeziehung einfach auf das Ziellager übertragen werden, was leider aufgrund der verwendeten Assoziationsklasse in USE bisher nicht möglich ist.[2]

deleteStored() ist eine Hauptfunktion, die eine Lagerbeziehung aus dem aufrufendem Lager löschen soll. nach dem Löschen muss gegebenenfalls das Lagerdatum auf den

1 Siehe dazu 2.2.1 auf Seite 5
2 Siehe auch im Dateiverzeichnis Seite 88

neuen, letzten Stand gesetzt werden, da wir den alten soeben gelöscht haben können.[1]

getProduct() ist eine Hilfsfunktion, die mit Hilfe der übergebenen Produktnummer das gelagerte *Product* zurückgibt.

newStored() ist eine Hilfsfunktion und soll eine neue Lagerbeziehung zu dem Produkt mit der angegebenen Nummer herstellen.

update() soll das Lagerdatum aktualisieren, sollte das übergebene Datum größer sein, als das aktuelle. Diese Funktion ist so angelegt, da die hier verwendeten Testfälle auch nur mit Testdaten laufen und kein „Heute“ bestimmt ist.[2]

relinkProduct() ist eine Hilfsfunktion, die bei der *moveAmount()*-Operation ursprünglich eingesetzt werden sollte. Diese Herangehensweise ist in den entsprechenden Implementierungen auskommentiert, da sie nicht unterstützt wird, soll aber der Vollständigkeit halber aufgeführt bleiben.

addStored() ist eine Hilfsfunktion und erstellt eine Lagerbeziehung zwischen dem aufrufenden Lager und dem Produkt mit der übergebenen Nummer.

findStored() ist eine Hilfsfunktion, die zwei Parameter, eine Produktnummer und einen Lagernamen, entgegennimmt und dazu die gehörige Lagerbeziehung *Stored* ausgibt.

getProductCount() ist eine Hilfsfunktion, die zählt wieviele Produkte mit dem aufrufenden Lager verbunden sind.

getHighestDate() ist eine Hilfsfunktion und liefert das höchste, verfügbar Datum als `Integer`, sollte das Lager noch Lagerbeziehungen unterhalten. Lagert das

1 Siehe auch im Dateiverzeichnis Seite 88
2 Siehe dazu 2.2.1 auf Seite 5

Lager keine Produkte mehr, gibt diese Funktion das Lagerdatum zurück.

hasProduct() ist eine Hilfsfunktion, die eine Produktnummer erwartet und eine `Boolean` ausgibt. Sollte das aufrufende Lager ein Produkt verlinken, welches die Produktnummer aufweist, gibt die Funktion `true` zurück, sonst `false`.

Stored

... beschreibt die Lagerverbindung zwischen einem Produkt und einem Lager. Ein Lager kann beliebig viele Produkte lagern. Ein Produkt kann in beliebig vielen Lagern eingelagert sein.

amount speichert die Lagermenge als `Real`. Dies ist ein Kompromiss, da zum Beispiel Kabelmengen durchaus meterweise angegeben werden, es allerdings keinen Sinn ergibt, 12,5 Glühbirnen zu lagern. Denkbar wäre selbstverständlich auch `Integer`, aber in diesem Fall müsste man Kabelmengen in Centimeter angeben. So ist es möglich überall „native“ Einheiten anzugeben.

date gibt die letzte Änderung an.[1]

update() ist eine Hilfsfunktion, die aufgrund der hier verwendeten Datumsverarbeitung das eigene Datum einfach inkrementiert.[2]

setAmount() ist eine Hilfsfunktion, die die Lagermenge auf den angegebenen Paramterwert setzt. [3]

1 Siehe dazu 2.2.1 auf Seite 5
2 Siehe auch im Dateiverzeichnis Seite 88
3 Siehe auch im Dateiverzeichnis Seite 88

Supplied

... beschreibt das Lieferverhältnis zwischen einem Produkt und einem Lieferanten. Dabei muss ein Produkt mindestens einen Lieferanten haben. Ein Lieferant hingegen kann beliebig viele Produkte anbieten.
Diese Assoziationsklasse köntnem an eventuell um bestimmte Sodnerangebote oder Mengenrabatte ergänzen.

oderNumber ist die vom Lieferanten bestimmte Bestellnummer als `String`, da diese Nummer auch Sonderzeichen und Buchstaben enthalten kann. Des Weiteren darf jede Bestellummer bei einem Lieferanten nur einmal vorkommen.

price speichert den vom Lieferanten veranschlagten Preis für das verlinkte Produkt als `Real`.

Tag

... ist eine Beschreibung oder eine Kategorie, die beliebig vielen Produkten zugewiesen werden kann. So kann man sich bei späteren Suchvorgängen Produkte mit bestimmten Eigenschaften herraussuchen, seien diese Produktattribute, wie zum Beispiel „Farbe Weiß“, oder Bewertungen wie „gute Qualität“.
Eine Typuntescheidung verschiedener Tags, sowie eventuelle Anwendungsregeln wären hier eine sinnvolle Erweiterung.

name ist die `String`-Beschreibung der Eigenschaft.

2.2.2 Assoziationen

Specifies

... ist die Verbindung zwischen einer Produktkategorie und einem Produkt. Ein Produkt gehört immer nur einem Produkttypen an, aber ein Produkttyp beschreibt beliebig viele Produkte.

Tagged

... verbindet beliebig viele Tags mit beliebig vielen Produkten.

2.2.3 Invarianten

Contractor

```
1 inv contractorIsDefined :  name.isDefined
```

contractorIsDefined? Der Name eines Lieferanten muss definiert sein.

```
1 inv contractorIsUnique :  Contractor.allInstances -> forAll(self2 |
    self <> self2 implies self.name <> self2.name)
```

contractorisUnique? Es darf jeden Lieferanten nur einmal geben, das heißt, dass wenn zwei *Contractor*-Objekte unterschiedlich sind, auch ihre Namen unterschiedlich sein müssen.

Manufactured

```
1 inv itemNumberIsDefined :  itemNumber.isDefined
```

itemNumberIsDefined? Die Artikelnummer muss definiert sein.

```
1 inv itemNumberIsUniqueForManufacturer : self.manufacturer.manufactured
    -> forAll(self2 | self <> self2 implies self.itemNumber <> self2.
    itemNumber)
```

itemNumberIsUniqueForManufacturer? Jeder Hersteller darf eine Artikelnummer nur einmal für ein Produkt verwenden, das heißt, dass jedes Produkt eines Herstellers, das verschieden von einem anderen Produkt des gleichen Hersetllers ist, auch eine unterschiedliche Artikelnummer hat.

Manufacturer

```
1 inv manufacturerIsDefined :  name.isDefined
```

manufacturerIsDefined? Der Name des Herstellers muss definiert sein.

```
1 inv manufacturerIsUnique :  Manufacturer.allInstances -> forAll(self2
    | self <> self2 implies self.name <> self2.name)
```

manufacturerIsUnique? Es darf jeden Hersteller nur einmal geben. Sind also zwei *Manufacturer*-Objekte unterschiedlich, so müssen auch die deren Namen unterschiedlich sein.

Product

```
1 inv productIsDefined : name.isDefined and date.isDefined and number.
    isDefined
```

productIsDefined? Name, Datum und Produktnummer müssen definiert sein.

```
1 inv productIsUnique : Product.allInstances -> forAll(self2 |
2     if (self <> self2) then (self.manufactured.itemNumber = self2.
          manufactured.itemNumber) implies (self.manufacturer <> self2.
          manufacturer)
3     else true endif)
```

productIsUnique? Damit ein Produkt einzigartig ist, dürfen nicht die Artikelnummern und die Hersteller übereinstimmen. Wenn also bei der Überprüfung zwei unterschiedliche *Product*-Objekte gefunden werden, so setzt eine gleiche Artikelnummer voraus, dass die Hersteller der Produkte unterschiedlich sind.[1]

```
1 inv productNumberIsUnique : Product.allInstances -> forAll(self2 |
    self.number <> self2.number implies self <> self2)
```

productNumberIsUnique? Sind die Produktnummern unterschiedlich, muss auch das Produkt ein unterschiedliches sein. Damit ist jede Produktnummer einzigartig.

```
1 inv hasLatestDate : self.stored.date -> asSequence -> select(s | s <=
    self.date) -> size() = self.stored.date -> size()
```

hasLatestDate? Wir stellen sicher, dass das Produktdatum das aktuellste ist. Dafür bilden wir zwei Mengen: Die erste Menge beinhaltet alle Daten der produktzugehörigen Lagerverbindungen, die kleiner gleich dem Produktdatum sind. Die

1 Eigentlich wird dies implizit mit der Invariante *itemNumberIsUniqueForManufacturer?* überprüft. Demnach gilt diese Invariante als Formsache.

zweite Menge enthält alle verfügbaren Daten der produktzugehörigen Lagerverbindungen. Es werden die Elemente beider Mengen gezählt. Sollte das momentane Produktdatum tatsächlich das aktuellste sein, so müsste jedes der zu erst ausgewählten Daten kleiner oder gleich sein. Demnach müssten nun beide Mengen gleich groß sein, dann muss das Produkt das aktuellste Datum tragen.

ProductType

```
1 inv productTypeIsDefined :  name.isDefined
```

productTypeIsDefined? Der Name des Produkttyps muss definiert sein.

```
1 inv productTypeIsUnique :  ProductType.allInstances -> forAll(self2 |
    self <> self2 implies self.name <> self2.name)
```

productTypeIsUnique? Es sollte jeden Produkttypen nur einmal geben. Das heißt, wenn zwei *ProductType*-Objekte unterschiedlich sind, müssen auch deren Namen unterschiedlich sein.

Storage

```
1 inv storageIsDefined :  name.isDefined and date.isDefined
```

storageIsDefined? Name und Datum müssen definiert sein.

```
1 inv storageIsUnique :  Storage.allInstances -> forAll(self2 | self <>
    self2 implies self.name <> self2.name)
```

storageIsUnique? Jedes Lager soll einzigartig sein. Wenn also zwei *Storage*-Objekte unterschiedlich sind, müssen auch deren Namen unterschiedlich sein.

```
1 inv storagesDateIsConsistent :  (Storage.allInstances -> forAll(s |
2     if self.product->size() = 0 then true else
3     (Storage.allInstances -> forAll(s | self.stored.date -> exists (d
          | d = self.date)) ) endif)
4     )
```

storagesDateIsConsistent? Es muss entwerder mindestens eine Lagerverbindung geben, die dasselbe Datum wie das Lager aufweist, doer das Lager msus leer sein. Es wird also geprüft, ob das Lager keine Produkte hat, ist dies der Fall wird `true` ausgewertet, sonst wird geprüft, ob die Menge an Daten der verknüpften Lagerverbindungen das Lagerdatum enthält.

Stored

```
1 inv storedIsDefined :  amount.isDefined and date.isDefined
```

storedIsDefined? Sowohl das Datum als auch die Stückzahl müssen definiert sein.

```
1 inv amountIsPositive :  amount > 0
```

amountIsPositive? Die Stückzahl soll größer Null sein. Negative Lagerbestände sind nicht sinnvoll und wenn keine produkte in dem Lager gelagert werden, also die Stückzahl Null beträgt, soll es auch keine Lagerverbindung geben.

Supplied

```
1 inv suppliedIsDefined : price.isDefined and orderNumber.isDefined
```

suppliedIsDefined? Preis und Bestellnummer müssen definiert sein.

```
1 inv orderNumberIsUniqueForContractor : self.contractor.supplied ->
    forAll(self2 | self <> self2 implies self.orderNumber <> self2.
    orderNumber)
```

orderNumberIsUniqueForContractor? Jeder Lieferant darf eine Bestellnummer nur einmal verwenden. Demnach müssen alle Lieferverbindungen eines Lieferanten die unterschiedlich sind, auch unterschiedliche Bestellnummern aufweisen.

Tag

```
1 inv tagIsDefined :  name.isDefined
```

nameIsDefined? Der Name eines Tags muss definiert sein.

```
1 inv tagIsUnique : Tag.allInstances -> forAll(self2 | self <> self2
    implies self.name <> self2.name)
```

tagIsUnique? Der Name eines Tags muss einzigartig sein. Das heißt, dass wenn zwei *Tag*-Objekte unterschiedlich sind, auch deren Namen unterschiedlich sein müssen.

2.2.4 Konditionen

Auflistung der Operationen in alphabetischer Reihenfolge:

deleteStored()

```
1 pre parametersAreDefined : aProductNumber.isDefined
```

pre parametersAreDefined? Alle Parameter müssen definiert sein.

```
1 pre storedExists : stored -> exists( s | s.product.number =
    aProductNumber)
```

pre storedExists? Das Produkt, das durch die übergebene Produktnummer angegeben wurde, muss eine Lagerverbindung zum aufrufendem Lager aufweisen. Ist die Produktnummer ungültig, schlägt diese Prekondition ebenfalls fehl. Für diese Überprüfung wird die Menge der mit dem aufrufenden Lager verknüpften Lagerverbindungen nach der Existenz einer Verbindung zu einem Produkt mit der angegebenen Produktnummer untersucht.

```
1 post dateIsConsistent : if (self.stored -> size() > 0) then (self.
    stored.date -> asSequence -> last() = self.date) else (self.date =
    self.date@pre) endif
```

post dateIsConsistent? Das Datum des Lagers soll nach der Operation das höchste verfügbare sein. Dafür wird zunächst geprüft, ob dieses Lager weitere Lagerverbindungen besitzt. Ist dies nicht der Fall, so muss das Lagerdatum dasselbe wie vor der Operationsausführung sein. hat das Lager aber noch Lagerverbindungen, so werden deren Daten zu einer Sequenz geformt und das letzte,

also höchste, Datum dieser Sequenz müsste nun dem neuen Lagerdatum entsprechen.

```
1 post storedIsDeleted : if (self.stored -> size() > 0) then (self.
    product -> forAll(p | p.number <> aProductNumber)) else (true)
    endif
```

post storedIsDeleted? Es wird nachgeprüft, ob das die angegebene Lagerverbindung tatsächlich gelöscht wurde. Ist die Menge verknüpfter Lagerverbindungen nun kleiner gleich Null, so muss das vorher vorhandene Lager gelöscht worden sein. Sind noch Lagerverbindungen verfügbar, so muss jede dieser Verbindungen ein Produkt referenzieren, dessen Produktnummer unterschiedlich zu der als Parameter übergebenen ist.

findCheapestContractor()

```
1 post foundCheapest : self.supplied->forAll(s | s <> result.call
    implies result.cost < s.price)
```

post foundCheapest? Hier wird noch einmal überprüft, ob wir tatsächlich die günstigste Liefermöglichkeit ausfindig machen konnten. Dafür wird jede Liefermöglichkeit *Supplied* mit der herausgefundenen, günstigsten verglichen. Sind die Zwei Unterschiedlich, so muss der resultierende Preis kleiner sein, als der verglichene.

moveAmount()

```
1 pre parametersAreDefined : aProductNumber.isDefined and anAmount.
    isDefined and aReceivingStorage.isDefined
```

pre parametersAreDefined? Alle Parameter müssen definiert sein.

```
1 pre movingAmountIsPositive : anAmount > 0
```

pre movingAmountIsPositive? Die zu verschiebene Stückzahl muss größer als Null sein. Ist sie negativ, müsste man eigentlich die Funktion von dem Ziellager aus mit positiver Stückzahl ausführen, ist die Stückzahl jedoch Null, gibt es nichts für die Operation zu tun.

```
1 pre storedExists : stored -> exists(s | s.product.number =
    aProductNumber)
```

pre storedExists? Das Produkt, von dem wir eine Anzahl transferieren wollen, muss auch im aufrufenden Lager verfügbar sein. Deswegen prüfen wir, ob eine Lagerverbindung dieses Lagers auf ein Produkt mit der gewünschten Nummer zeigt. Damit stellen wir sicher, dass es sowohl das Produkt, als auch die Lagerverbindung gibt.

```
1 pre amountAvailable : stored -> any(s | s.product.number =
    aProductNumber).amount >= anAmount
```

pre amountAvailable? Wir müssen über die Stückzahl die wir verschieben wollen auch verfügen können. Also muss die Lagerverbindung dieses Lagers zum Produkt mehr oder genausoviel enthalten, wie wir transferieren wollen.

```
1 pre receivingStorageExists : Storage.allInstances -> exists(s | s.name
    = aReceivingStorage)
```

pre receivingStorageExists? Das Ziellager muss existieren. Dafür wird geprüft, ob in der Gesamtmenge an Lagern eines existiert, dessen Name mit dem übergebenen übereinstimmt.

```
1 pre receivingStorageIsNotSending : self.name <> aReceivingStorage
```

pre receivingStorageIsNotSending? Das Ziellager sollte nicht das sendende Lager sein, sonst wäre der Transfer von Artikeln sinnlos. Dazu wird überprüft, ob der Name des sendenden Lagers verschieden mit dem des empfangenen ist.

```
1 post noEmptySendingStorage : if stored -> any(s | s.product.number =
    aProductNumber).amount@pre > anAmount
2       then (stored->any(s | s.product.number = aProductNumber).
            amount@pre - anAmount) > 0
3       else not stored->exists(s |  s.product.number = aProductNumber)
            endif
```

post noEmptySendingStorage? Da keine Lagerverbindung exisiteren soll, deren Stückzahl Null ist, darf so ein Lager nach dem Verschoeben von Artikeln nicht existieren. Es müsste gelöscht worden sein. Dafür wird geprüft, ob die alte Stückzah größer als die zu verschiebene war. Ist dies der Fall, muss die alte Stückzahl weniger der Transfermenge immernoch größer als Null sein. Andernfalls darf die Lagerverbindung zum angebenen Produkt nicht mehr in der Lagerverbingungsmenge des sendenden Lagers existieren.

```
1 post finallyStored :
2       let storedHere : Stored = Stored.allInstances -> any(s | s.
          product.number = aProductNumber and s.storage = self) in
3       let storedThere : Stored = Stored.allInstances -> any(s | s.
          storage.name = aReceivingStorage and s.product.number =
          aProductNumber) in
4       if storedThere.amount@pre.isUndefined and storedHere.amount.
          isDefined then
5          storedThere.amount = anAmount and
6          storedHere.amount =  storedHere.amount@pre - anAmount
7          else
8          if storedThere.amount@pre.isDefined and storedHere.amount.
             isUndefined then
9             storedThere.amount = storedThere.amount@pre + anAmount
10            else
11            if storedThere.amount@pre.isUndefined and storedHere.
                amount.isUndefined then
12               storedThere.amount = anAmount
13               else
14               storedThere.amount = storedThere.amount@pre +
                   anAmount and
15               storedHere.amount =  storedHere.amount@pre -
                   anAmount
16            endif
17         endif
18      endif
```

post finallyStored? Hier soll geprüft werden, ob die Menge tatsächlich verschoben worden ist. Dazu müssen wir je zwei Stückzahlen betrachten: Die Sendezahl und die Empfängerzahl. Und diese können je nach Zeitpunkt und Situation definiert sein oder nicht. Wenn also die Empfänderzahl vor der Operationsausführung nicht definiert war, die Senderzahl danach aber schon, so muss aktuell die Empfängerzahl gleich der Transfermenge sein und die Senderzahl die alte Senderzahl abzüglich der Transfermenge. Ist aber die Empfängerzahl vorher definiert und die Senderzahl nachher nicht, so muss nur die Empfängerzahl die alte Empfängerzahl plus die Transfermenge sein. Ist die Empfängerzahl

allerdings vorher undefiniert und die Senderzahl nachher auch, so muss nur die Empfängerzahl genausogroß wie die Transfermenge sein. Ist die Empfängerzahl davor und die Senderzahl danach definiert, so muss die Empfängerzahl die alte Zahl zuzüglich der Transfermenge und die Senderzahl die alte Zahl abzüglich der Transfermenge sein.

restock()

```
1 pre parametersAreDefined : aSearchAmount.isDefined and aNewAmount.
    isDefined
```

pre parametersAreDefined? Alle Parameter müssen definiert sein.

```
1 pre amountsArePositive : (aSearchAmount >= 0) and (aNewAmount > 0)
```

pre amountsArePositive? Die Stückzahl, nach der gesucht werden muss, muss größer gleich Null sein und die Auffüllzahl einfach größer als Null.

```
1 pre amountsAreBalanced : aSearchAmount < aNewAmount
```

pre amountAreBalanced? Die Stückzahl, nach der gesucht weden soll, muss kleiner als die Auffüllmenge sein.

```
1 post amountRestocked : self.stored -> collect(s | s.amount@pre) -> sum
    () + result.boughtUnits = self.stored.amount -> sum()
```

post amountRestocked? Die Stückzahlen müssen tatsächlich aufgefüllt worden sein. Das heißt, die jetzt verfügbaren Stückzahlen zusammengezählt sollten nun genausogroß sein, wie die resultierende Zahl der alten Stückzahlen addiert mit den aufgefüllten Stückzahlen des Operationsergenisses.

3 Objektdiagramme

Da sich einige Invarianten vom Aufbau, Zweck und Testweise so ähneln, wird nur eine Invariante repräsenativ für ihre ähnlichen Verwandten aufgeführt. Zudem wurden nur Invarianten getestet, die auch einigermaßen interessant sind.

3.1 Ein gültiges Objektdiagramm

3.1.1 Das Objektdiagramm

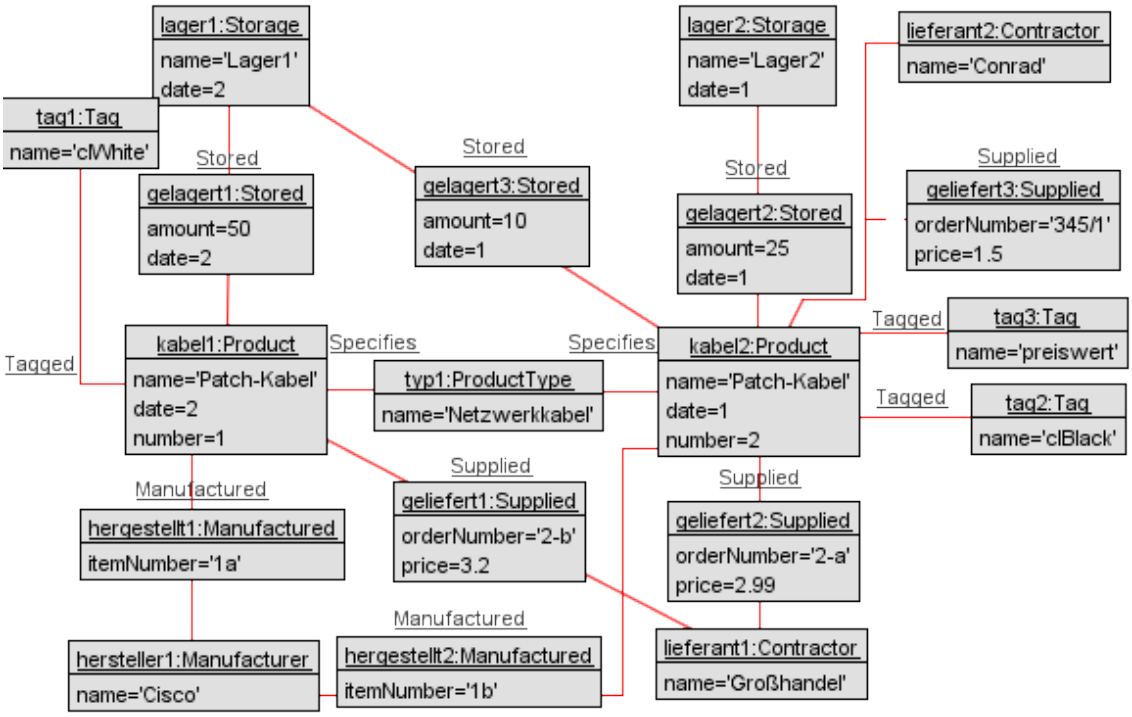

Abbildung 3.1: Ein Beispiel für ein gültiges Objektdiagramm. (Siehe dazu das Dateiverzeichnis auf Seite 88)

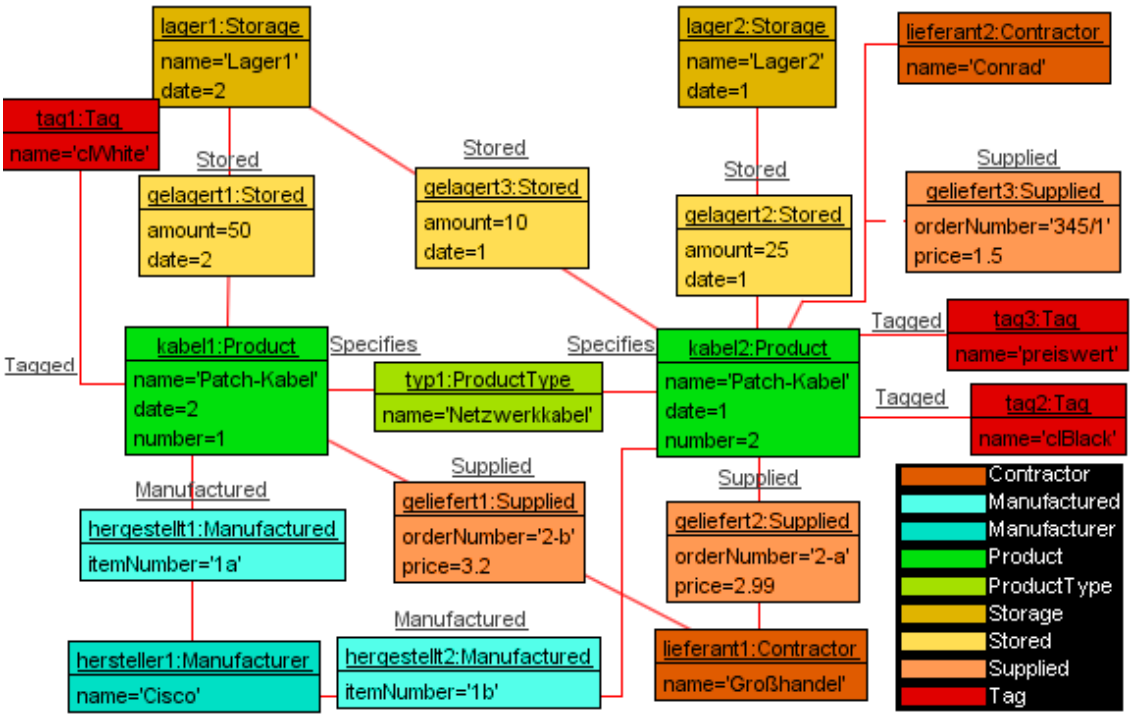

Abbildung 3.2: Abbildung 3.1 zur besseren Übersicht eingefärbt. Rechts unten befindet sich die Legende.

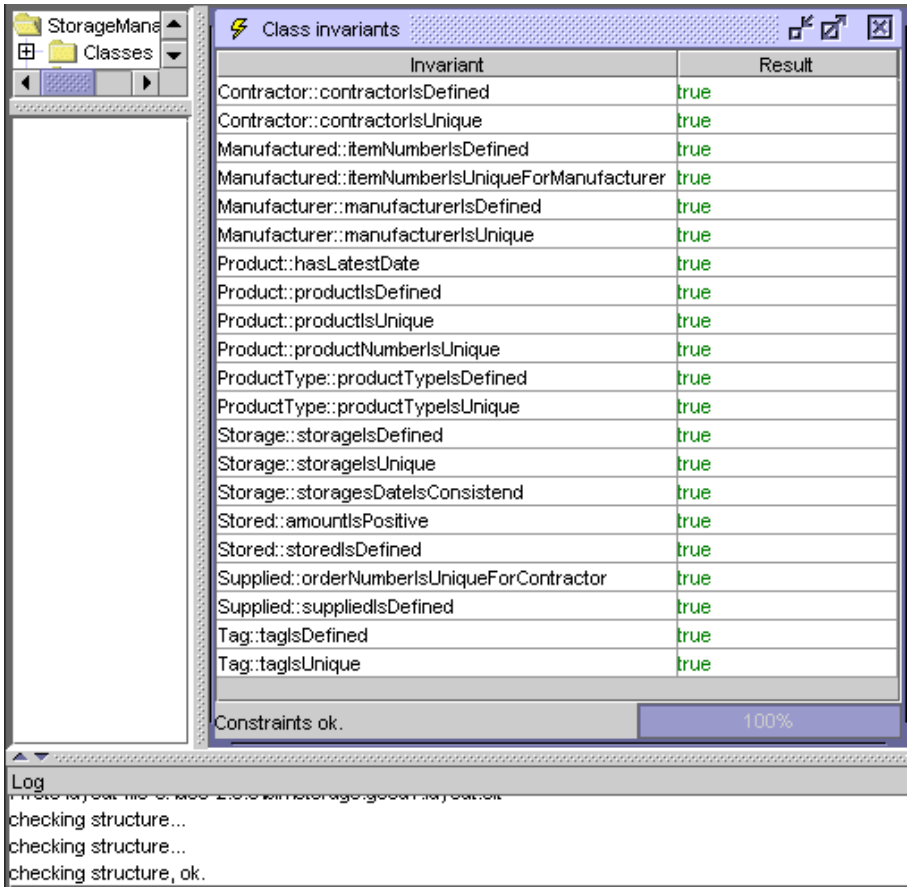

Abbildung 3.3: Die Auswertung der Invarianten für das Objektdiagramm aus Abbildung 3.1 zeigt einen gütligen Zustand an. (Dass in vielen Invarianten-Screenshots „storagesDateIsConsistend“ anstatt „storagesDateIsConsistent“ steht ist mir ledier erst sehr spät aufgefallen. Ich bitte diesen kleinen Fehler zu entschuldigen.)

3.1.2 Zur Anordnung der Objekte

Das Objektdiagramm aus Abbildung 3.1 soll als Beispiel eines gültigen Zustandes dienen. Natürlich ist das Lagersystem erst dann richtig nützlich, wenn es sehr viele Produkte, Tags, Lieferanten und Hersteller gibt, doch lässt sich dann das Objektdiagramm nicht mehr übersichtlich darstellen. Tatsächlich mussten schon hier die Assoziationsklassen auf die Assoziationskanten verschoben werden, um das ganze Diagramm übersichtlich zu halten. Zusätzlich habe ich die verschiedenen Klassen in der Abbildung 3.2 eingefärbt, was es vereinfacht, die Objekte auseinanderzuhalten.

3.1.3 Zum Inhalt des Diagramms

In dem Diagramm gibt es zwei Lager, zwei Produkte, zwei Lieferanten, einen Hersteller, drei Tags und einen Produkttypen. Die Produkte sind soweit identisch und unterscheiden sich nur darin, dass sie vom Hersteller mit einer unterschiedlichen Artikelnummer versehen sind und sich die lagerinterne Produktnummer unterscheidet. Des Weiteren ist das zweite Produkt in beiden Lagern vorhanden und wird von zwei Lieferanten zu unterschiedlichen Preisen angeboten.[1]

3.2 Ungültige Objektdiagramme

Das Objektdiagramm aus Abbildung 3.1 soll hier in leicht veränderter Form als Anschauung dienen und die deklarierten Invarianten überprüfen.

3.2.1 Doppelte Namen

[2]

1 Seite 88
2 Seite 88

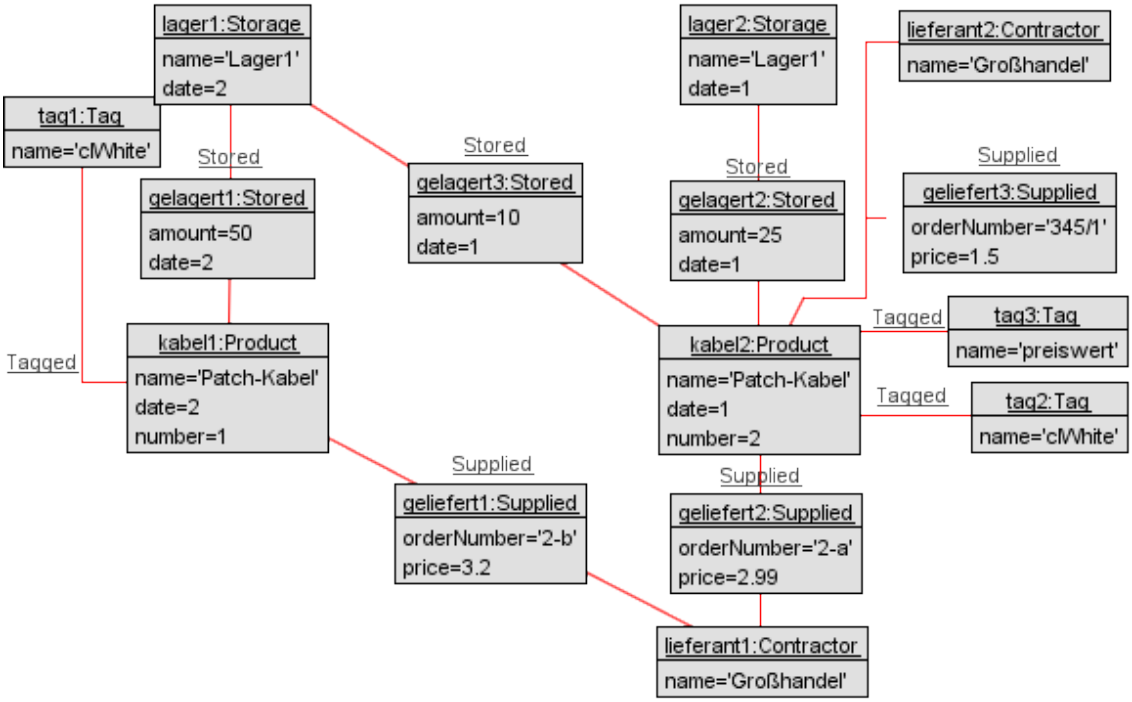

Abbildung 3.4: Lieferanten, Lager und zwei Tags haben die selben Namen erhalten, was dieses Objektdiagramm ungültig macht. Produkttyp und Hersteller wurden für die Darstellung ausgeblendet.

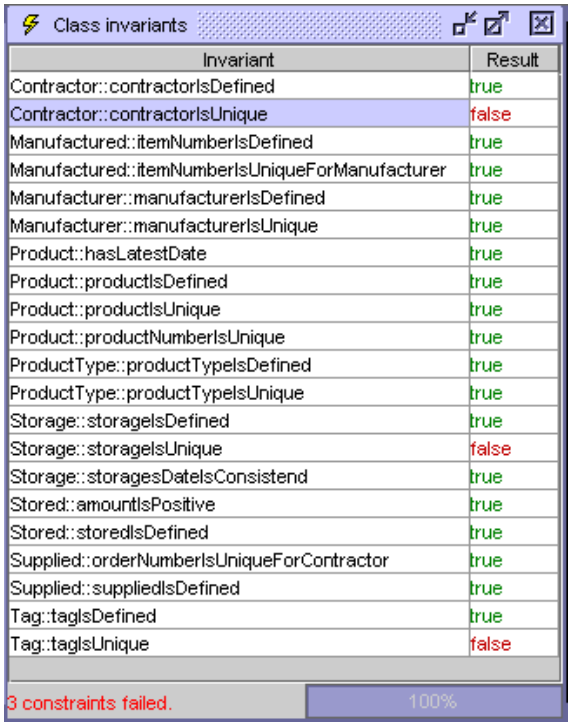

Abbildung 3.5: Die Auswertung der Invarianten für das Objektdiagramm aus Abbildung 3.4 zeigt Verstöße gegen die drei Invarianten *contractorIsUnique?*(Seite 13), *storageIsUnique?* (Seite 15) und *tagIsUnique?* (Seite 17). Damit ist dieser Zustand ungültig.

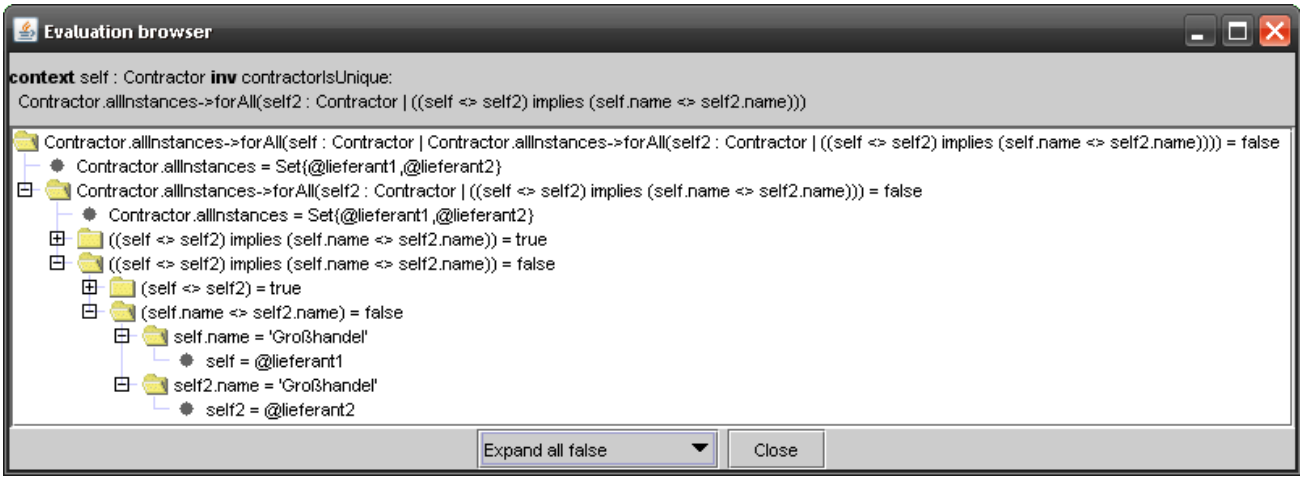

Abbildung 3.6: Der Evaluationsbrowser für die Invariante *contractorIsUnique?* steht exemplarisch für die anderen scheiternden in Abbildung 3.4. Obwohl die zwei geprüften Objekte verschieden sind, sind ihre Namen identisch. Die Invariante schlägt fehl.

3.2.2 Identische Bestellnummern bei einem Lieferanten

1

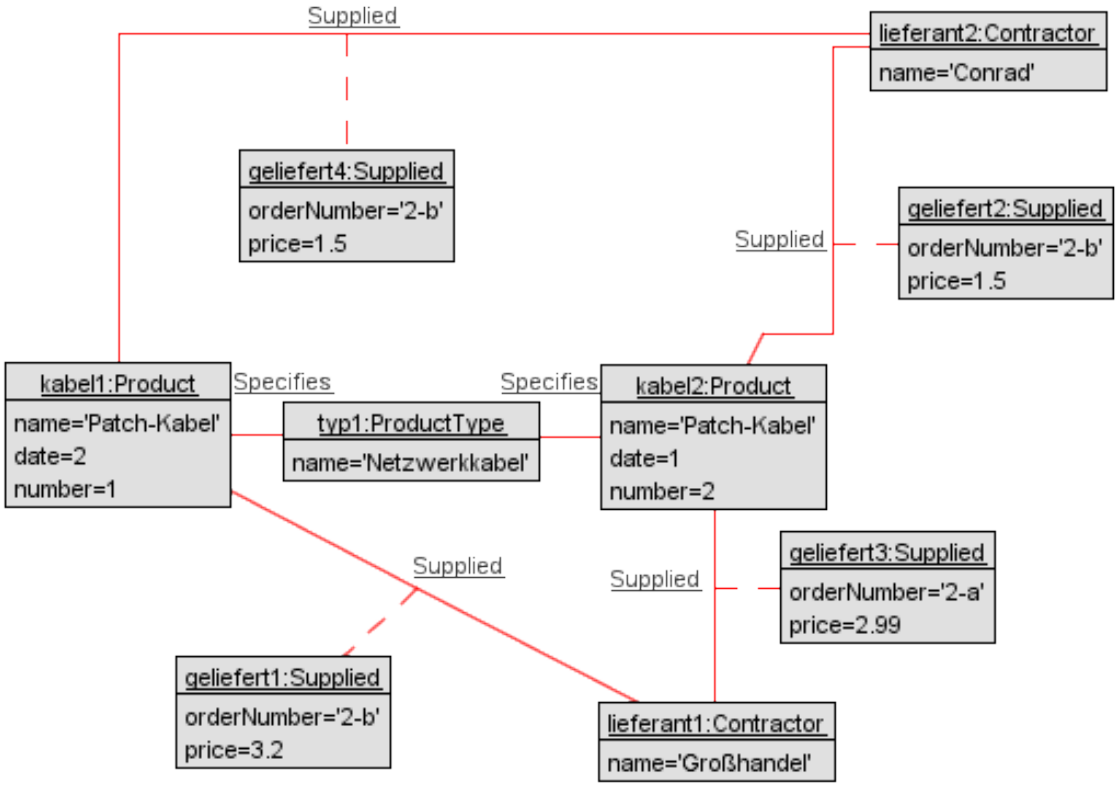

Abbildung 3.7: In diesem Beispiel benutzt der Lieferant „Conrad“ für beide Produkte dieselbe Bestellnummer. Dass der Lieferant „Großhandel“ ebenfalls einmal dieselbe Bestellnummer benutzt, ist allerdings erlaubt. Für die Darstellung wurde alles bis auf die Produkte, der Produkttyp und die Lieferanten ausgeblendet.

1 Seite 88

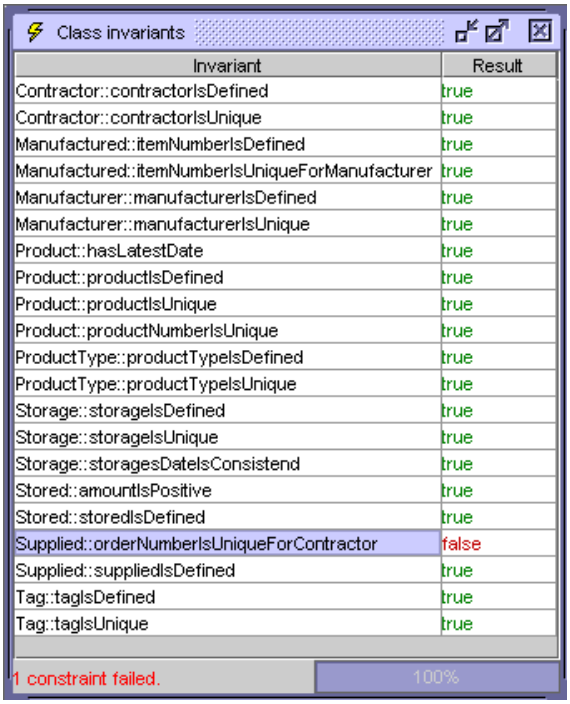

Abbildung 3.8: Die Auswertung der Invarianten für das Objektdiagramm aus Abbildung 3.7 zeigt Verstöße gegen die Invariante *orderNumberIsUniqueForContractor?*(Seite 16). Damit ist dieser Zustand ungültig.

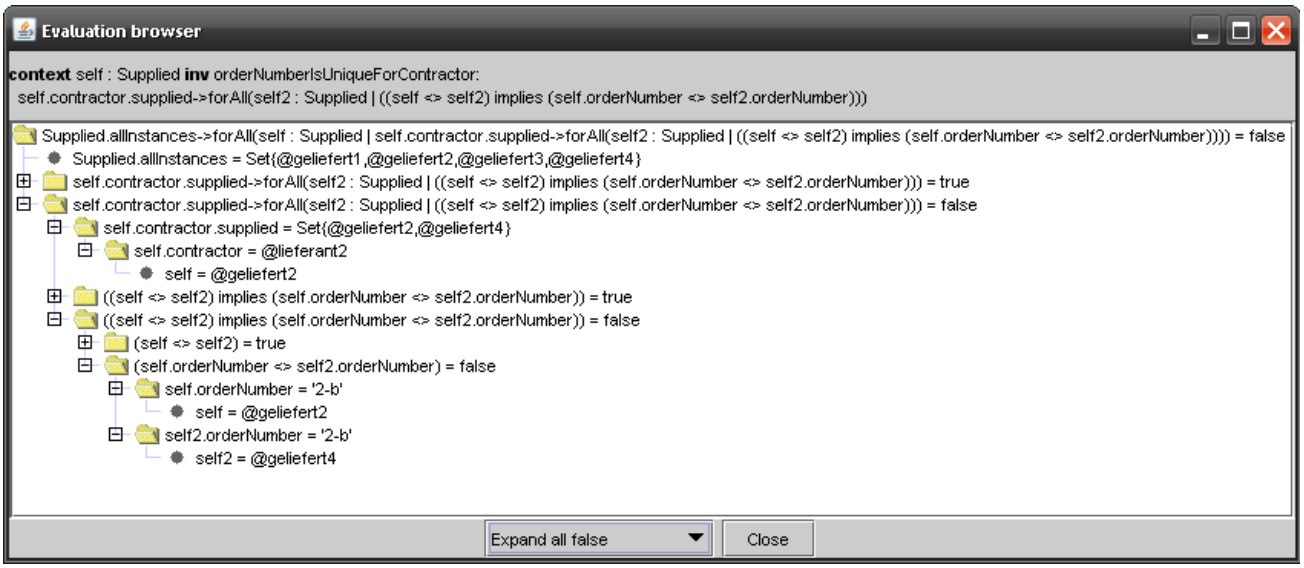

Abbildung 3.9: Der Evaluationsbrowser für die Invariante *orderNumberIsUniqueForContractor?*.

3.2.3 Produktdatum ist nicht konsistent

1

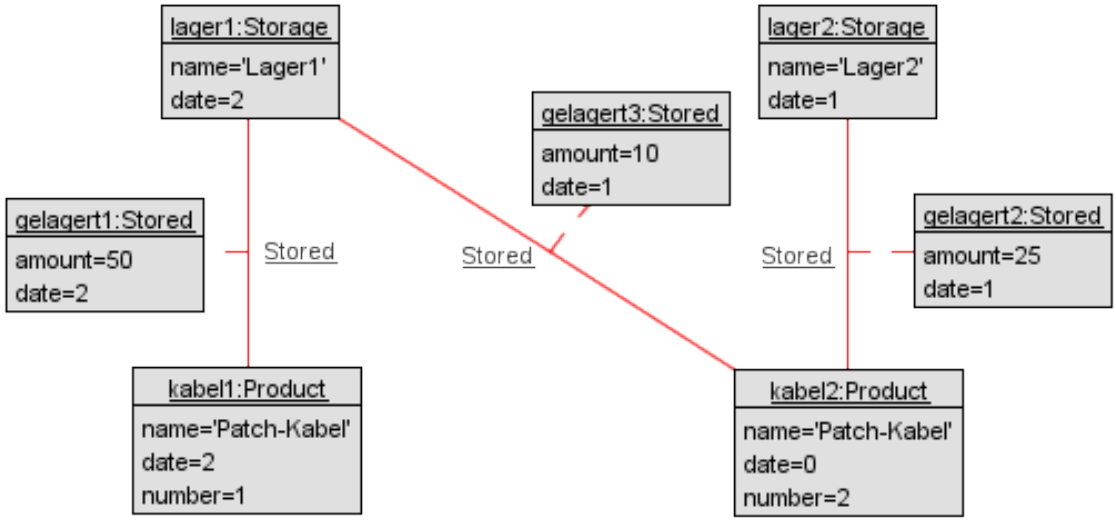

Abbildung 3.10: Hier wurde das Datum vom zweiten *Product*, „kabel2“, auf 0 gesetzt. Damit sind alle mit dem Produkt zusammenhängende Daten größer als das eigene. Datumskonsistenz ist nicht gegeben, weswegen dieser Zustand ungültig ist. Zur besseren Übersicht werden nur Lager, Produkte und Lagerbeziehungen angezeigt.

1 Seite 88

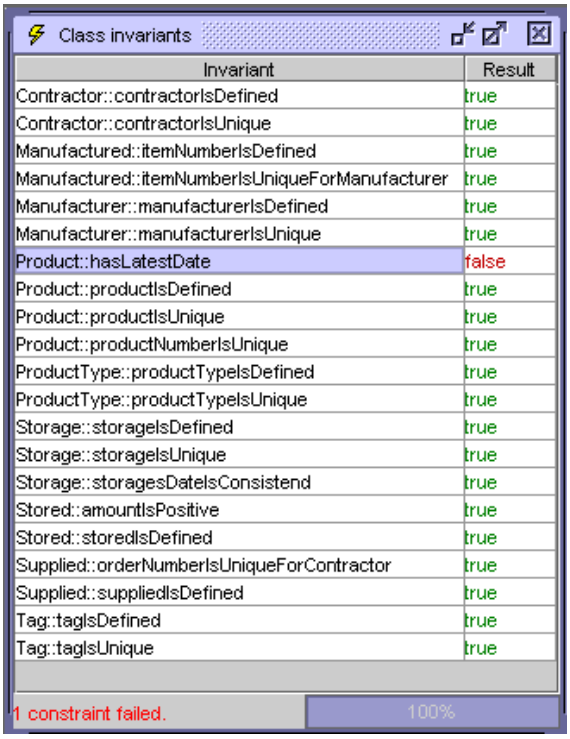

Abbildung 3.11: Die Auswertung der Invarianten für das Objektdiagramm aus Abbildung 3.10 zeigt Verstöße gegen die Invariante *hasLatestDate?*(Seite 14). Damit ist dieser Zustand ungültig.

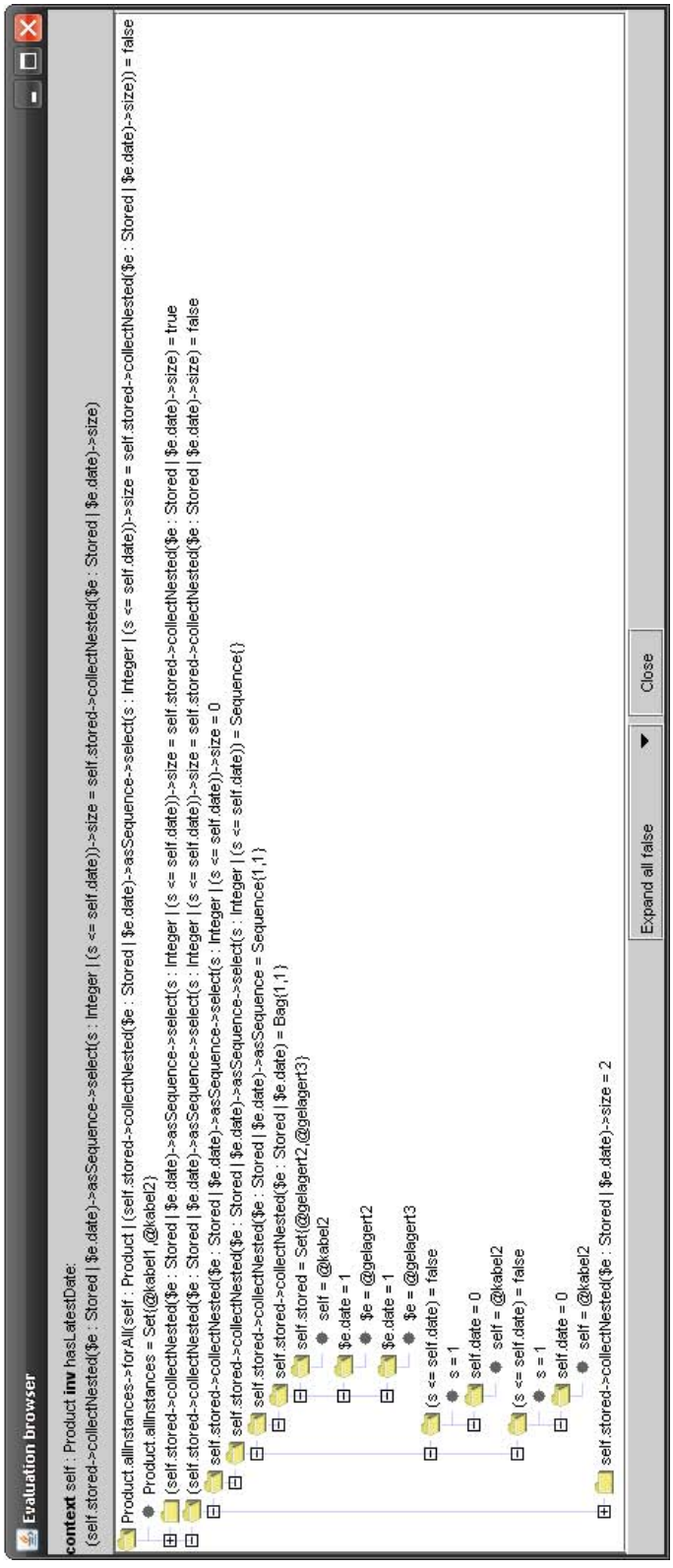

Abbildung 3.12: Der Evaluationsbrowser für die Invariante *hasLatestDate?*.

3.2.4 Produktdatum ist nicht konsistent II

1

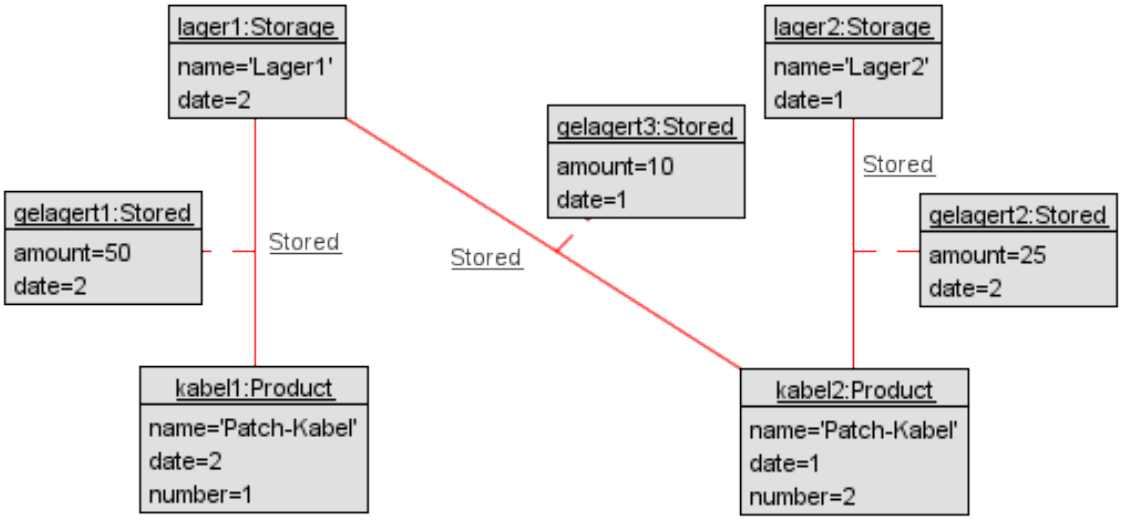

Abbildung 3.13: Hier wurde das Datum der Lagerverbindung „gelagert2“ auf 2 gesetzt. Nun hat das dazugehörige Produkt „kabel2“ nicht mehr das höchste Datum und das Lager verfügt auch nicht mehr über eine Lagerverbindung mit korrespondierendem Datum. Der Zustand ist ungültig. Zur besseren Übersicht werden nur Lager, Produkte und Lagerbeziehungen angezeigt.

1 Seite 88

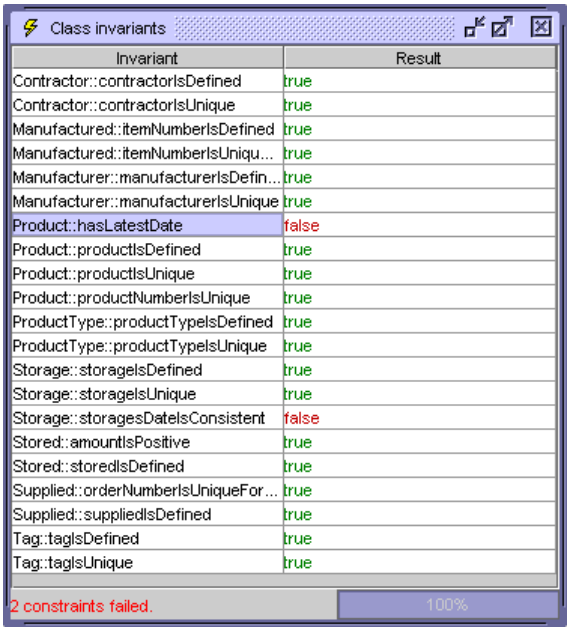

Abbildung 3.14: Die Auswertung der Invarianten für das Objektdiagramm aus Abbildung 3.13 zeigt Verstöße gegen die Invarianten *hasLatestDate?*(Seite 14) und *storagesDateIsConsistent?*(Seite 16). Damit ist dieser Zustand ungültig.

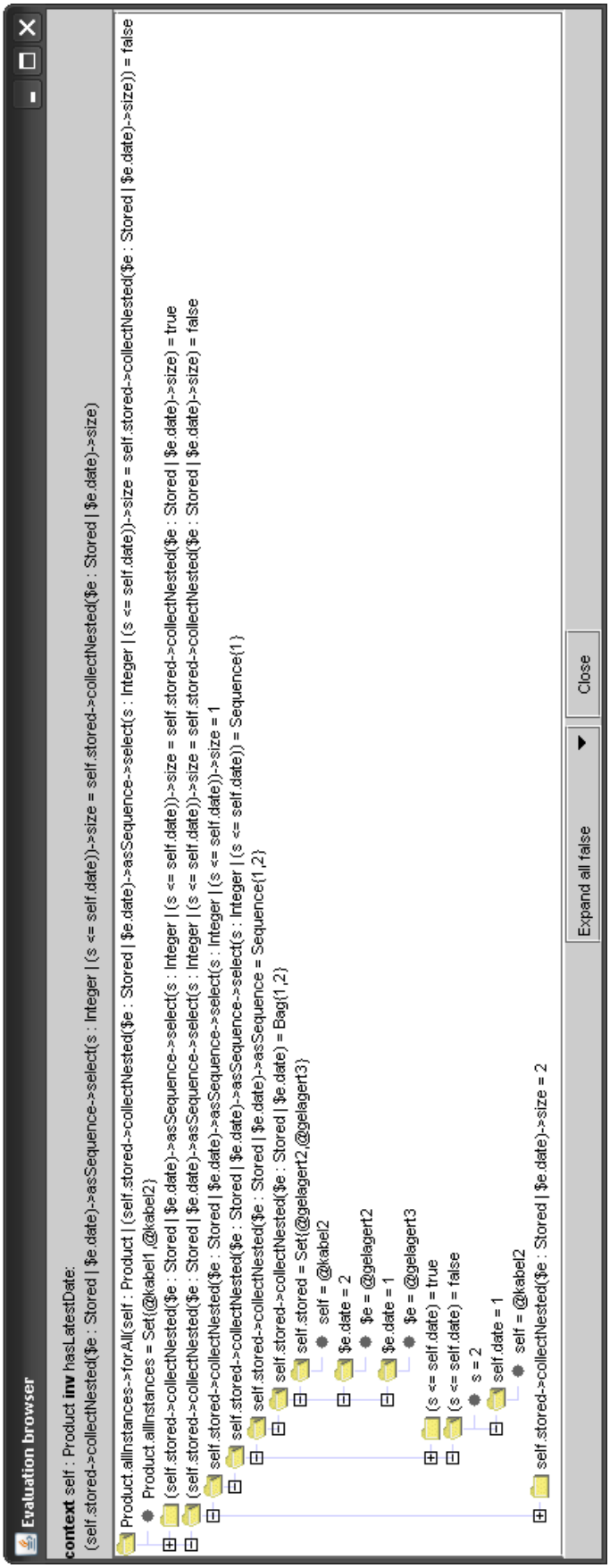

Abbildung 3.15: Der Evaluationsbrowser für die Invariante *hasLatestDate?*.

3.2.5 Lagerdatum ist inkonsistent

1

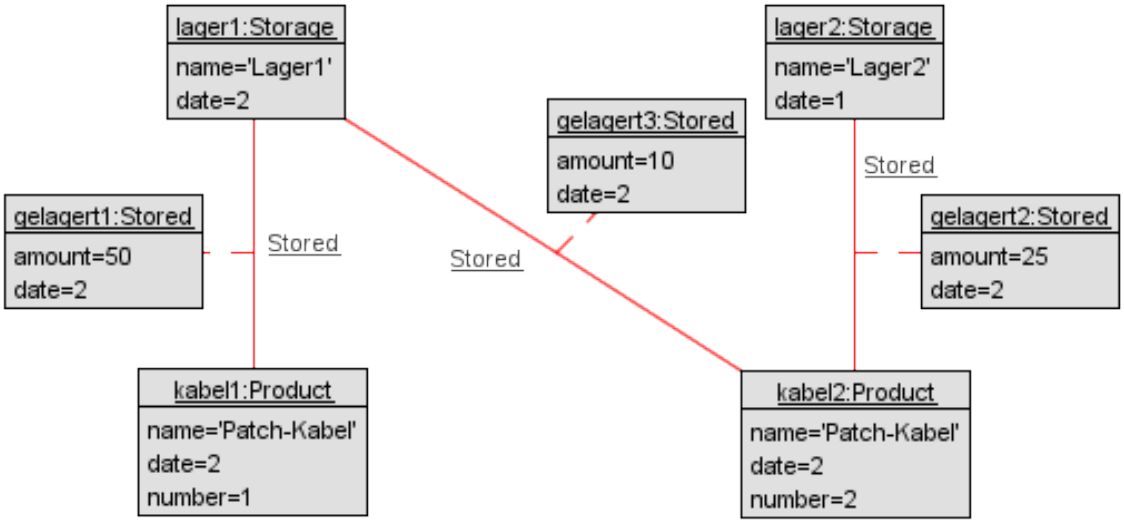

Abbildung 3.16: Die Daten von „kabel2“ und den dazugehörigen Lagerverbindungen wurde auf 2 gesetzt. Allerdings ist dieses Produkt das einzige in „lager2“, weswegen das dortige Datum, wertigkeit 1, nicht mehr das aktuellste ist. Zur besseren Übersicht werden nur Lager, Produkte und Lagerbeziehungen angezeigt.

1 Seite 88

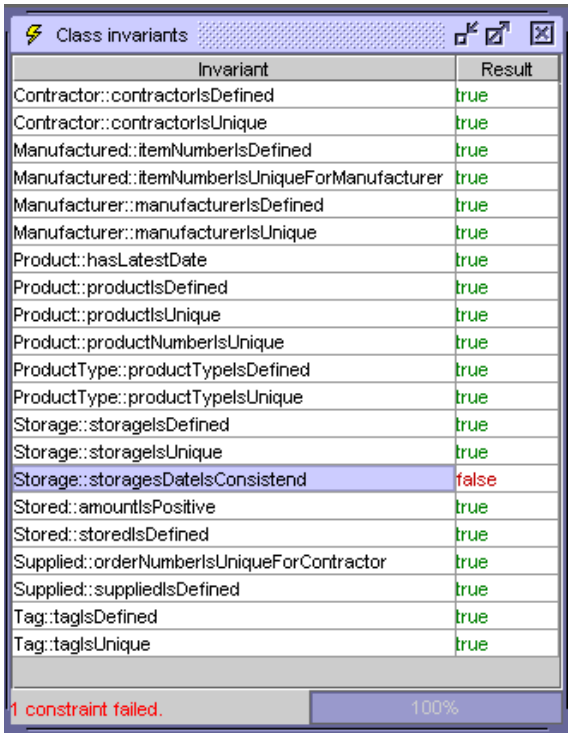

Abbildung 3.17: Die Auswertung der Invarianten für das Objektdiagramm aus Abbildung 3.16 zeigt Verstöße gegen die Invariante *storagesDateIsConsistent?*(Seite 16). Damit ist dieser Zustand ungültig.

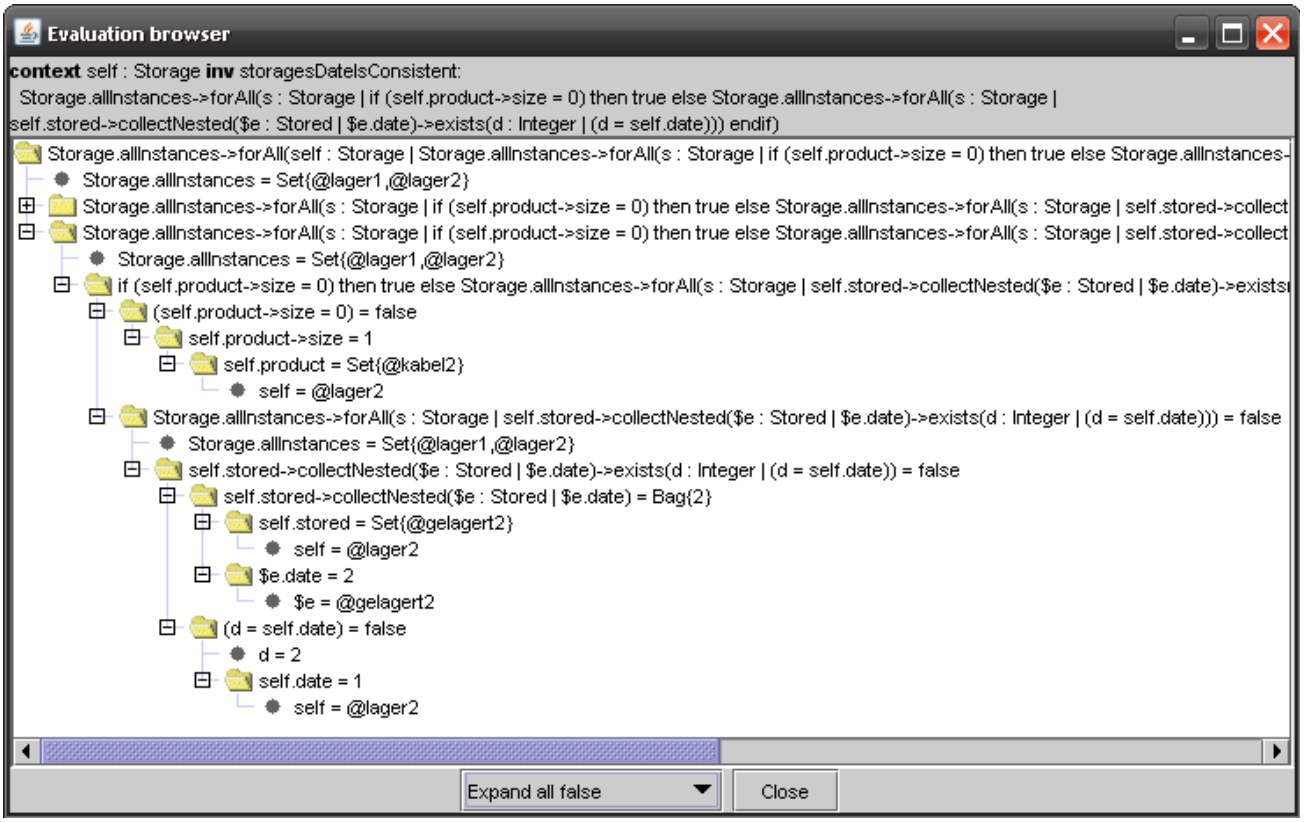

Abbildung 3.18: Der Evaluationsbrowser für die Invariante *storagesDateIsConsistent?*.

3.2.6 Zwei Produkte sind identisch

1

1 Seite 88

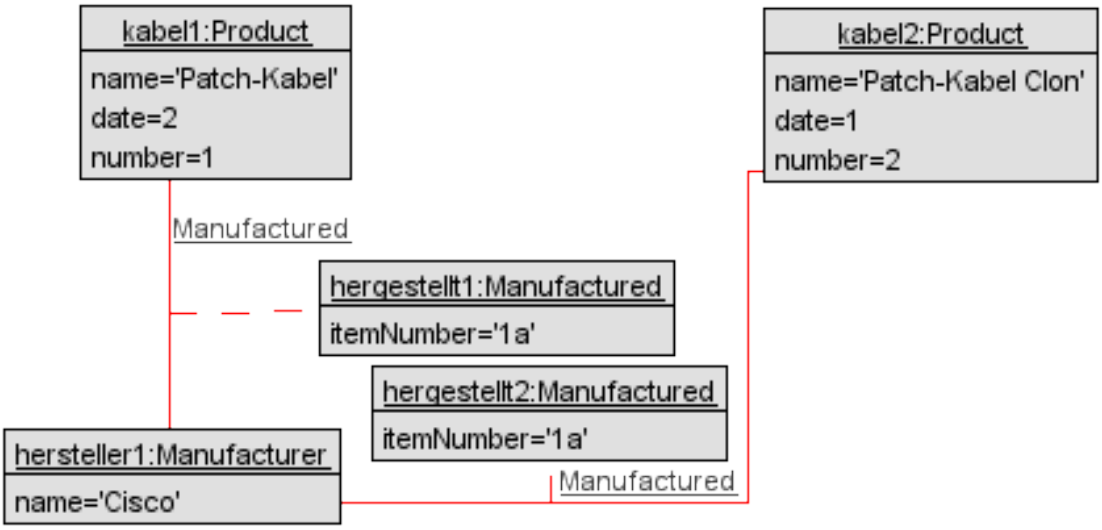

Abbildung 3.19: Artikelnummer und Hersteller vom *Product* „kabel2“ sind identisch mit denen vom *Product* „kabel1“. Aufgrund dessen sind diese zwei Produkte identisch und der Zustand ungültig. Zusätzlich is es nicht erlaubt, dass der Hersteller zweimal dieselbe Artikelnummer verlinkt. Zu besseren Übersicht werden nur die Produkte und der Hersteller angezeigt.

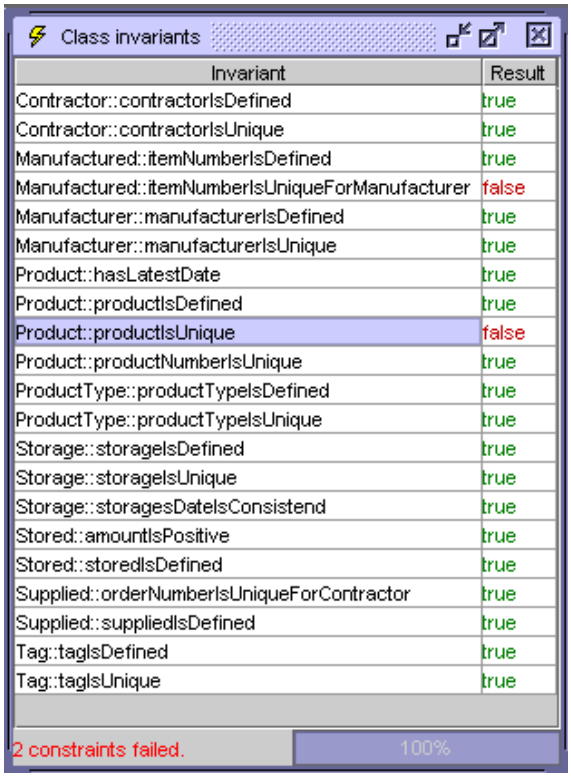

Abbildung 3.20: Die Auswertung der Invarianten für das Objektdiagramm aus Abbildung 3.19 zeigt Verstöße gegen die Invarianten *itemNumberIsUniqueForManufacturer?*(Seite 13) und *productIsUnique?*(Seite 14). Damit ist dieser Zustand ungültig.

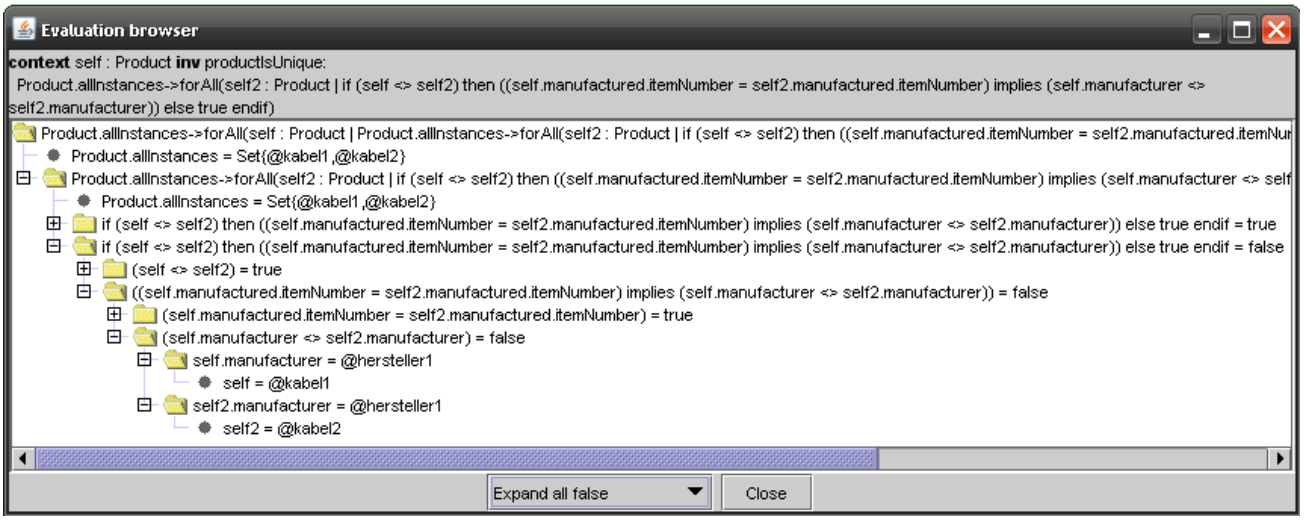

Abbildung 3.21: Der Evaluationsbrowser für die Invariante *productIsUnique?*.

4 Sequenzdiagramme

In diesem Kapitel sollen die drei Hauptfunktionen getestet werden. Für jede Funktion wird ein Testzustand eingeführt, auf dem anschließend die Operationen ausgeführt werden. Für einige Testfälle wird dieser Zustand zuvor oder im Nachhinein ein wenig verändert, um andere Resultate zu erzielen oder Konditionen zu verletzen. Andere Testfälle gehen progressiv vor und verändern den Zustand mit jedem Schritt.
Die Kommandozeilenausgaben zeigen immer den Ausschnitt, in dem der jeweilige Testzustand bereits eingeladen ist.
Im Allgemeinen wird versucht, sich auf die wichtigen/interessanten Diagramme und Ausschnitte zu beschränken.

4.1 deleteStored()

In jedem Testfall dieser Serie werden die Operationen direkt auf dem Testzustand ausgeführt.

Die Deklaration:

```
1 -- When an AssociationClass Stored is to be deleted, one should
    enforce date consistency! "Stored" will be deleted if it is
    connected to the calling storage and the specified product.
2  deleteStored(aProductNumber : Integer)
3  -- Parameters have to be defined.
4  pre parametersAreDefined : aProductNumber.isDefined
5  -- Product is actually connected to this storage.
6  pre storedExists : stored -> exists( s | s.product.number =
       aProductNumber)
7  -- After deletion the storage which before featured the product have
       to get a new date. Preferably the next highest, i.e. latest date
       available. If there are any other products stored in the
       respective storage, that is.
8  post dateIsConsistent : if (self.stored -> size() > 0) then (self.
       stored.date -> asSequence -> last() = self.date) else (self.date
       = self.date@pre) endif
9  post storedIsDeleted : if (self.stored -> size() > 0) then (self.
       product -> forAll(p | p.number <> aProductNumber)) else (true)
       endif
```

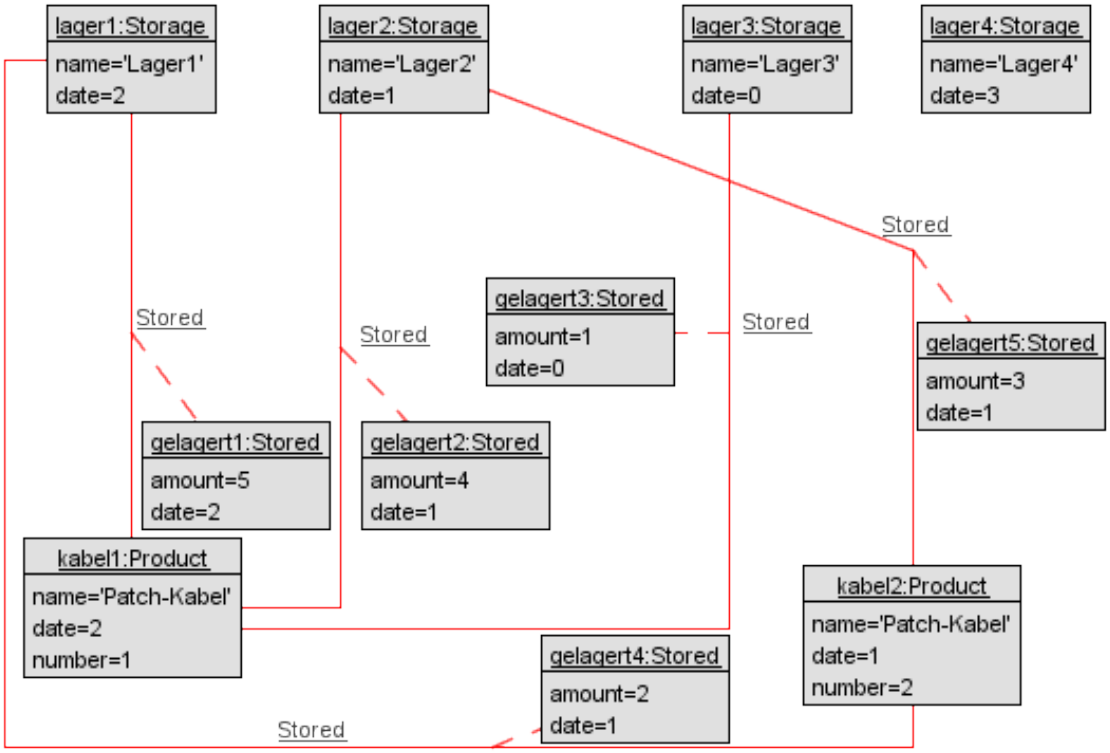

Abbildung 4.1: Der gültige Zustand für die Operation *deleteStored()*, die im folgendem getestet werden soll. Es werden nur die Produkte, die Lager und die Lagerverbindungen angezeigt.(Die Datei dazu findet sich auf Seite 88)

4.1.1 TC01

In diesem Test soll aus dem Zustand aus Abbildung 4.1 die Lagerverbindung zwischen „lager1“ und „kabel1“ gelöscht und die Daten korrigiert werden. [1]

1 Siehe auch im Dateiverzeichnis Seite 88

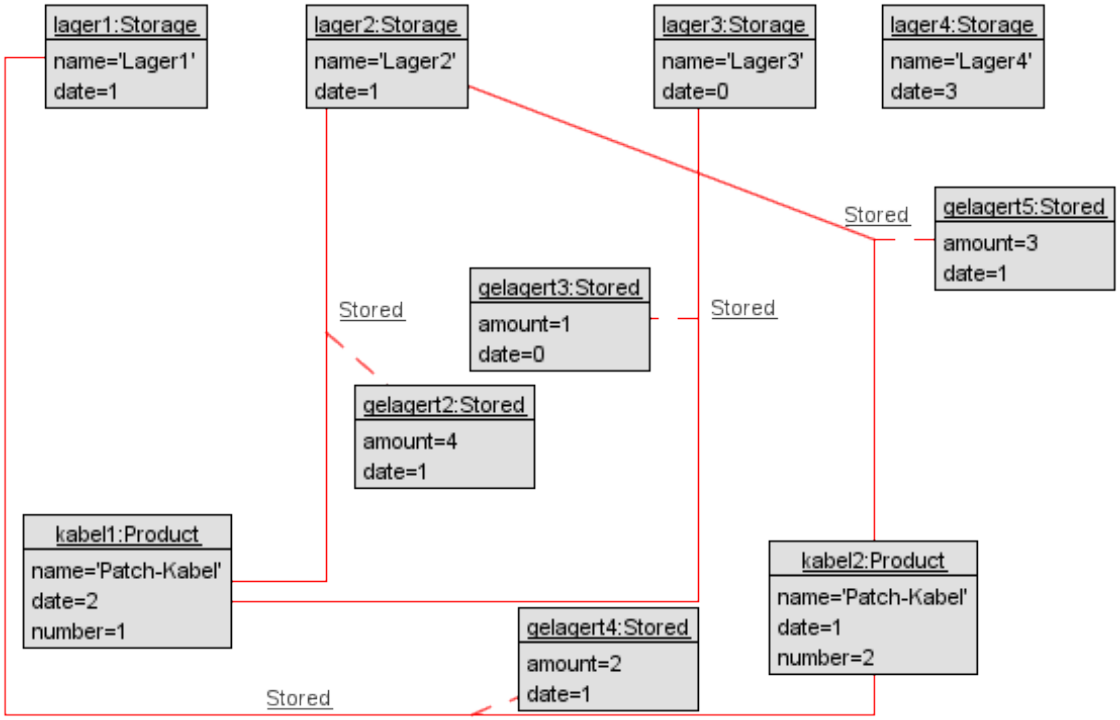

Abbildung 4.2: Wie man erkennt, wurde das Datum von „lager1“ wie geplant auf 1 gesetzt und die gewünschte Lagerverbindung gelöscht.(Alle Invarianten sind ebenfalls erfüllt.)

Die Kommandozeilenausgabe

```
1  storage.deleteStored.state>
2  tc01_deleteStored.command> !openter lager1 deleteStored(1)
3  precondition `parametersAreDefined' is true
4  precondition `storedExists' is true
5  tc01_deleteStored.command>
6  tc01_deleteStored.command>       open storage.deleteStored.command
7  storage.deleteStored.command> --Storage::deleteStoredProduct(
       aProductNumber : Integer)
8  storage.deleteStored.command>
9  storage.deleteStored.command>    !openter self hasProduct(
       aProductNumber)
10 storage.deleteStored.command>    !opexit self.hasProduct(aProductNumber
       )
11 storage.deleteStored.command>    !destroy self.findStored(
       aProductNumber, self.name)
12 storage.deleteStored.command>    !openter self getHighestDate()
13 storage.deleteStored.command>    !opexit self.getHighestDate()
```

```
storage.deleteStored.command>  !set self.date := if (self.stored->
    size()>0) then (self.stored.date -> asSequence -> last()) else (
    self.date) endif
storage.deleteStored.command>
storage.deleteStored.command>
tc01_deleteStored.command>
tc01_deleteStored.command> !opexit
postcondition `dateIsConsistent' is true
postcondition `storedIsDeleted' is true
tc01_deleteStored.command>
```

Sequenzdiagramm

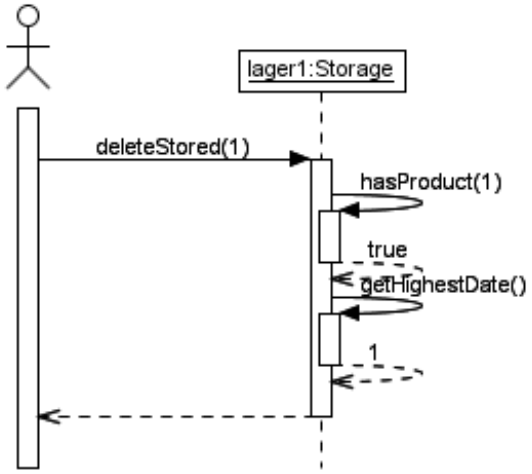

Abbildung 4.3: Das Sequenzdiagramm zu TC01. Die Funtkion sollte die Existenz des Zielprodukts überprüfen und es dann gegebenenfalls löschen. Abschließend soll die Operation das Datum des Lagers auf das nächst höchste Datum der noch verfügbaren Lagerverbindungen setzen.

4.1.2 TC02

In diesem Test löschen wir die Lagerverbindung zwischen „kabel2“ und „lager2“. Das Lagerdatum bleibt 1. [1]

1 Siehe auch im Dateiverzeichnis Seite 89

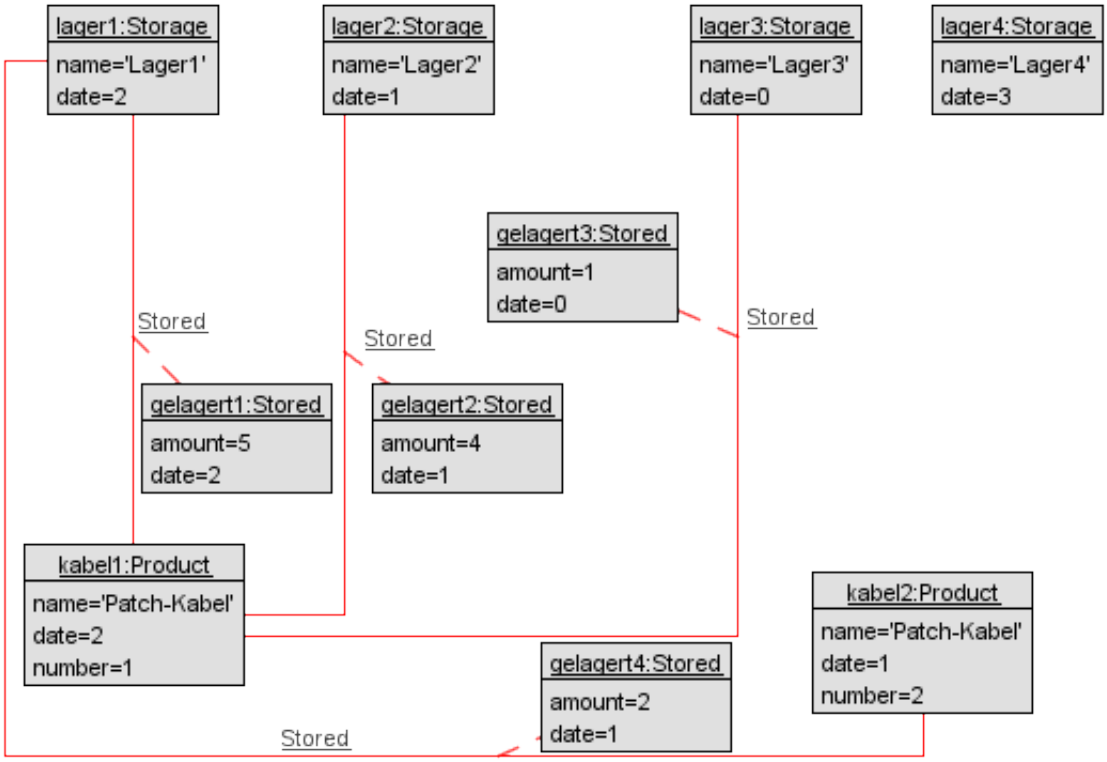

Abbildung 4.4: Die korrekte Verbindung wurde gelöscht, Datum und Invarianten sind in Ordnung.

Die Kommandozeilenausgabe

```
1  storage.deleteStored.state>
2  tc02_deleteStored.command> !openter lager2 deleteStored(2)
3  precondition `parametersAreDefined' is true
4  precondition `storedExists' is true
5  tc02_deleteStored.command>
6  tc02_deleteStored.command>       open storage.deleteStored.command
7  storage.deleteStored.command> --Storage::deleteStoredProduct(
       aProductNumber : Integer)
8  storage.deleteStored.command>
9  storage.deleteStored.command>    !openter self hasProduct(
       aProductNumber)
10 storage.deleteStored.command>    !opexit self.hasProduct(aProductNumber
       )
11 storage.deleteStored.command>    !destroy self.findStored(
       aProductNumber, self.name)
12 storage.deleteStored.command>    !openter self getHighestDate()
13 storage.deleteStored.command>    !opexit self.getHighestDate()
```

```
storage.deleteStored.command>   !set self.date := if (self.stored->
    size()>0) then (self.stored.date -> asSequence -> last()) else (
    self.date) endif
storage.deleteStored.command>
storage.deleteStored.command>
tc02_deleteStored.command>
tc02_deleteStored.command> !opexit
postcondition `dateIsConsistent' is true
postcondition `storedIsDeleted' is true
tc02_deleteStored.command>
```

4.1.3 TC03

„kabel1“ wird von „lager3“ gelöscht. Das Lager ist daraufhin leer, aber Datum und Invarianten sind korrekt. [1]

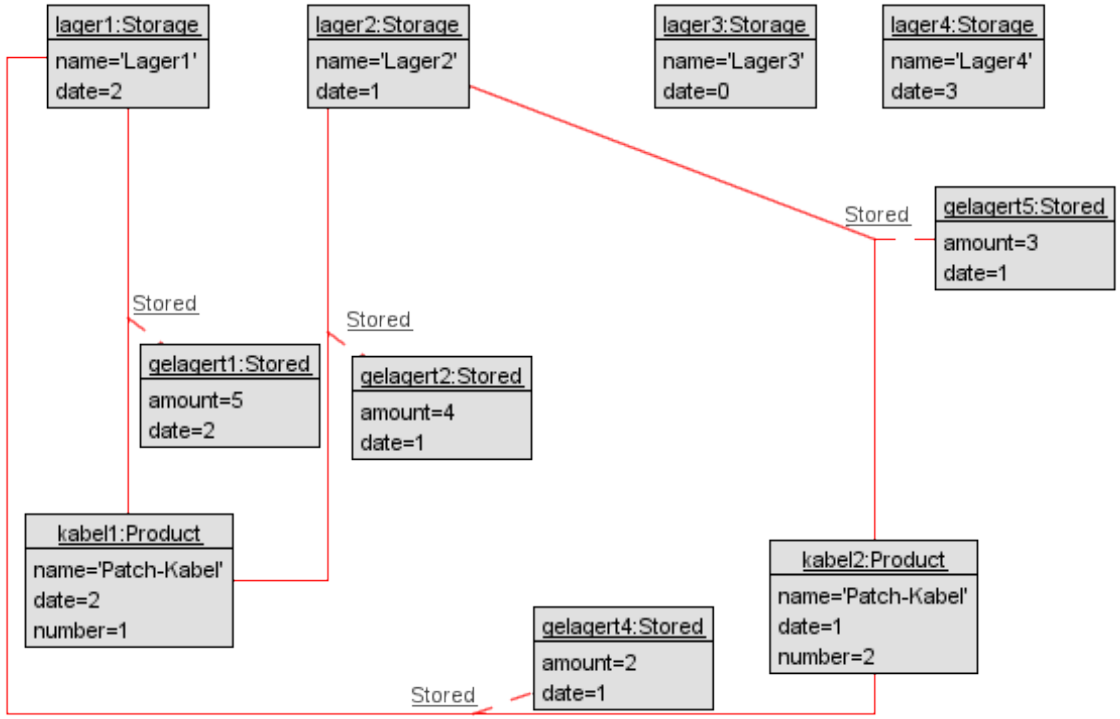

Abbildung 4.5: Die korrekte Verbindung wurde gelöscht, Datum und Invarianten sind in Ordnung.

Die Kommandozeilenausgabe

1 Siehe auch im Dateiverzeichnis Seite 89

```
1  storage.deleteStored.state>
2  tc03_deleteStored.command> !openter lager3 deleteStored(1)
3  precondition `parametersAreDefined' is true
4  precondition `storedExists' is true
5  tc03_deleteStored.command>
6  tc03_deleteStored.command>       open storage.deleteStored.command
7  storage.deleteStored.command> --Storage::deleteStoredProduct(
       aProductNumber : Integer)
8  storage.deleteStored.command>
9  storage.deleteStored.command>    !openter self hasProduct(
       aProductNumber)
10 storage.deleteStored.command>    !opexit self.hasProduct(aProductNumber
       )
11 storage.deleteStored.command>    !destroy self.findStored(
       aProductNumber, self.name)
12 storage.deleteStored.command>    !openter self getHighestDate()
13 storage.deleteStored.command>    !opexit self.getHighestDate()
14 storage.deleteStored.command>    !set self.date := if (self.stored->
       size()>0) then (self.stored.date -> asSequence -> last()) else (
       self.date) endif
15 storage.deleteStored.command>
16 storage.deleteStored.command>
17 tc03_deleteStored.command>
18 tc03_deleteStored.command> !opexit
19 postcondition `dateIsConsistent' is true
20 postcondition `storedIsDeleted' is true
21 tc03_deleteStored.command>
```

Sequenzdiagramm

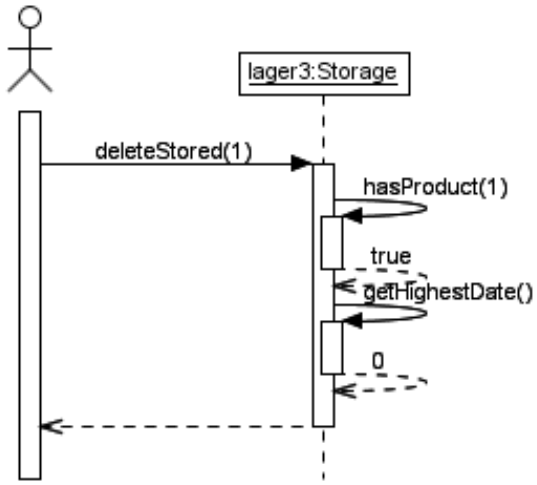

Abbildung 4.6: Das Sequenzdiagramm zu TC03. Es wird das aktuelle Lagerdatum zurückgegeben, da kein weitere, verknüpfte Lagerverbindung mehr gefunden werden kann.

4.1.4 TC04

In diesem Test wird versucht ein Produkt aus einem leeren Lager zu löschen. Die Kondition *pre storedExists?*[1] schlägt wie erwartet fehl. [2]

1 Siehe 19

2 Siehe auch im Dateiverzeichnis Seite 89

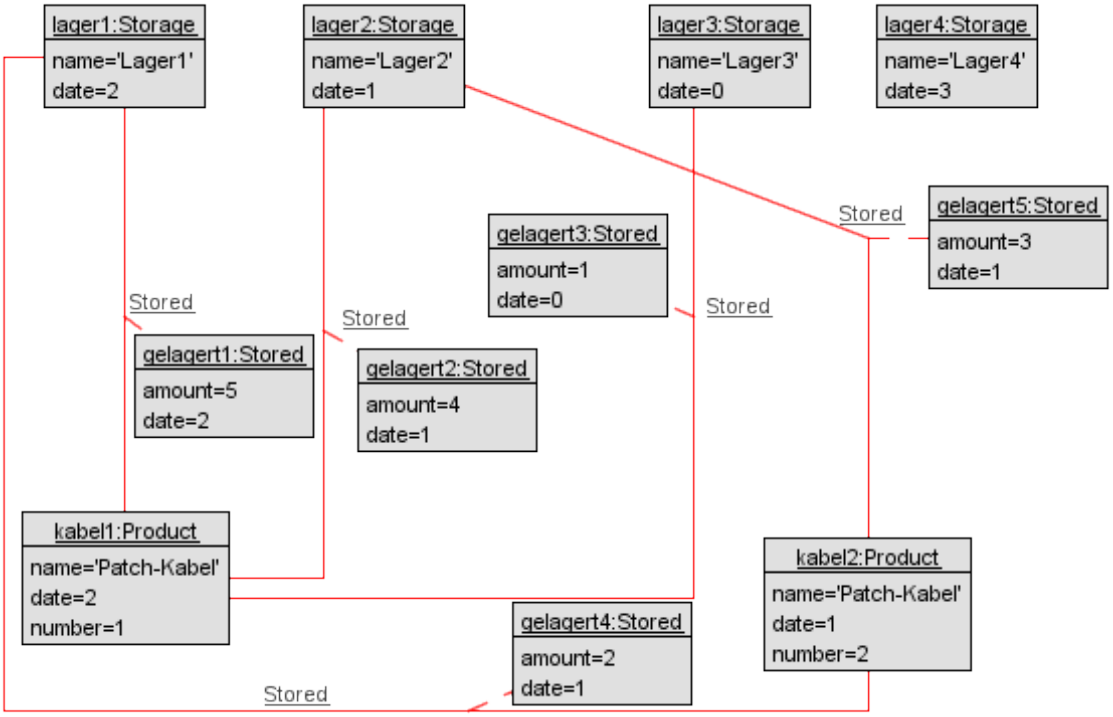

Abbildung 4.7: Es hat sich gegenüber dem zsutand aus Abbildung 4.1 nichts verändert.

Die Kommandozeilenausgabe

```
1  tc04_deleteStored.command> !openter lager4 deleteStored(1)
2  precondition `parametersAreDefined' is true
3  precondition `storedExists' is false
4  tc04_deleteStored.command>
5  tc04_deleteStored.command>       open storage.deleteStored.command
6  storage.deleteStored.command> --Storage::deleteStoredProduct(
       aProductNumber : Integer)
7  storage.deleteStored.command>
8  storage.deleteStored.command>    !openter self hasProduct(
       aProductNumber)
9  null:1:8: Undefined variable `self'.
10 storage.deleteStored.command>    !opexit self.hasProduct(aProductNumber
       )
11 null:1:7: Undefined variable `self'.
12 storage.deleteStored.command>    !destroy self.findStored(
       aProductNumber, self.name)
13 null:1:8: Undefined variable `self'.
14 storage.deleteStored.command>    !openter self getHighestDate()
15 null:1:8: Undefined variable `self'.
```

```
storage.deleteStored.command>    !opexit self.getHighestDate()
null:1:7: Undefined variable `self'.
storage.deleteStored.command>    !set self.date := if (self.stored->
    size()>0) then (self.stored.date -> asSequence -> last()) else (
    self.date) endif
null:1:4: Undefined variable `self'.
storage.deleteStored.command>
storage.deleteStored.command>
tc04_deleteStored.command>
tc04_deleteStored.command> !opexit
Error: Call stack is empty.
tc04_deleteStored.command>
```

Sequenzdiagramm

Abbildung 4.8: Das Sequenzdiagramm zu TC04 zeigt das fehlschlagen einer Prekondition.

4.1.5 TC05

Es wird versucht ein imaginäres Produkt zu löschen, die angegebene Produktnummer 404 exisitert nicht. Demnach schlägt die Kondition *pre storedExists?*[1] planmäßig fehl.[2]

Die Kommandozeilenausgabe

```
tc05_deleteStored.command> !openter lager1 deleteStored(404)
precondition `parametersAreDefined' is true
precondition `storedExists' is false
tc05_deleteStored.command>
tc05_deleteStored.command>       open storage.deleteStored.command
storage.deleteStored.command> --Storage::deleteStoredProduct(
    aProductNumber : Integer)
storage.deleteStored.command>
```

1 Siehe 19
2 Siehe auch im Dateiverzeichnis Seite 89

```
storage.deleteStored.command>   !openter self hasProduct(
    aProductNumber)
null:1:8: Undefined variable `self'.
storage.deleteStored.command>   !opexit self.hasProduct(aProductNumber
    )
null:1:7: Undefined variable `self'.
storage.deleteStored.command>   !destroy self.findStored(
    aProductNumber, self.name)
null:1:8: Undefined variable `self'.
storage.deleteStored.command>   !openter self getHighestDate()
null:1:8: Undefined variable `self'.
storage.deleteStored.command>   !opexit self.getHighestDate()
null:1:7: Undefined variable `self'.
storage.deleteStored.command>   !set self.date := if (self.stored->
    size()>0) then (self.stored.date -> asSequence -> last()) else (
    self.date) endif
null:1:4: Undefined variable `self'.
storage.deleteStored.command>
storage.deleteStored.command>
tc05_deleteStored.command>
tc05_deleteStored.command> !opexit
Error: Call stack is empty.
tc05_deleteStored.command>
```

Sequenzdiagramm

Abbildung 4.9: Das Sequenzdiagramm zu TC05 zeigt das fehlschlagen einer Prekondition.

4.1.6 TC06

Hier wird zwar ein gültiges Produkt gelöscht, doch wird kurz vor Operationsabschluß das Lagerdatum wieder auf 2 gesetzt. Folglich schlagen die Invariante *storagesDateIsConsis-*

tent?[1] und die Kondition *post dateIsConsistent?*[2] fehl.[3]

Die Kommandozeilenausgabe

```
1  tc06_deleteStored.command> !openter lager2 deleteStored(2)
2  precondition `parametersAreDefined' is true
3  precondition `storedExists' is true
4  tc06_deleteStored.command>
5  tc06_deleteStored.command>       open storage.deleteStored.command
6  storage.deleteStored.command> --Storage::deleteStoredProduct(
       aProductNumber : Integer)
7  storage.deleteStored.command>
8  storage.deleteStored.command>    !openter self hasProduct(
       aProductNumber)
9  storage.deleteStored.command>    !opexit self.hasProduct(aProductNumber
       )
10 storage.deleteStored.command>    !destroy self.findStored(
       aProductNumber,self.name)
11 storage.deleteStored.command>    !openter self getHighestDate()
12 storage.deleteStored.command>    !opexit self.getHighestDate()
13 storage.deleteStored.command>    !set self.date := if (self.stored->
       size()>0) then (self.stored.date -> asSequence -> last()) else (
       self.date) endif
14 storage.deleteStored.command>
15 storage.deleteStored.command>
16 tc06_deleteStored.command>       -- Thou shalt not corrupt thy date,
       but uphold the law of thy date's holy consistency!
17 tc06_deleteStored.command>       !set lager2.date := 2
18 tc06_deleteStored.command>
19 tc06_deleteStored.command> !opexit
20 postcondition `dateIsConsistent' is false
21 evaluation results:
22   self : Storage = @lager2
23   self.stored : Set(Stored) = Set{@gelagert2}
24   self.stored->size : Integer = 1
25   0 : Integer = 0
26   (self.stored->size > 0) : Boolean = true
27   self : Storage = @lager2
28   self.stored : Set(Stored) = Set{@gelagert2}
29   $e : Stored = @gelagert2
30   $e.date : Integer = 1
```

1 Seite 16
2 Seite 17
3 Siehe auch im Dateiverzeichnis Seite 89

```
31   self.stored->collectNested($e : Stored | $e.date) : Bag(Integer) =
       Bag{1}
32   self.stored->collectNested($e : Stored | $e.date)->asSequence :
       Sequence(Integer) = Sequence{1}
33   self.stored->collectNested($e : Stored | $e.date)->asSequence->last
       : Integer = 1
34   self : Storage = @lager2
35   self.date : Integer = 2
36   (self.stored->collectNested($e : Stored | $e.date)->asSequence->last
       = self.date) : Boolean = false
37   if (self.stored->size > 0) then (self.stored->collectNested($e :
       Stored | $e.date)->asSequence->last = self.date) else (self.date
       = self.date@pre) e
38 ndif : Boolean = false
39 postcondition `storedIsDeleted' is true
40 tc06_deleteStored.command>
```

Sequenzdiagramm

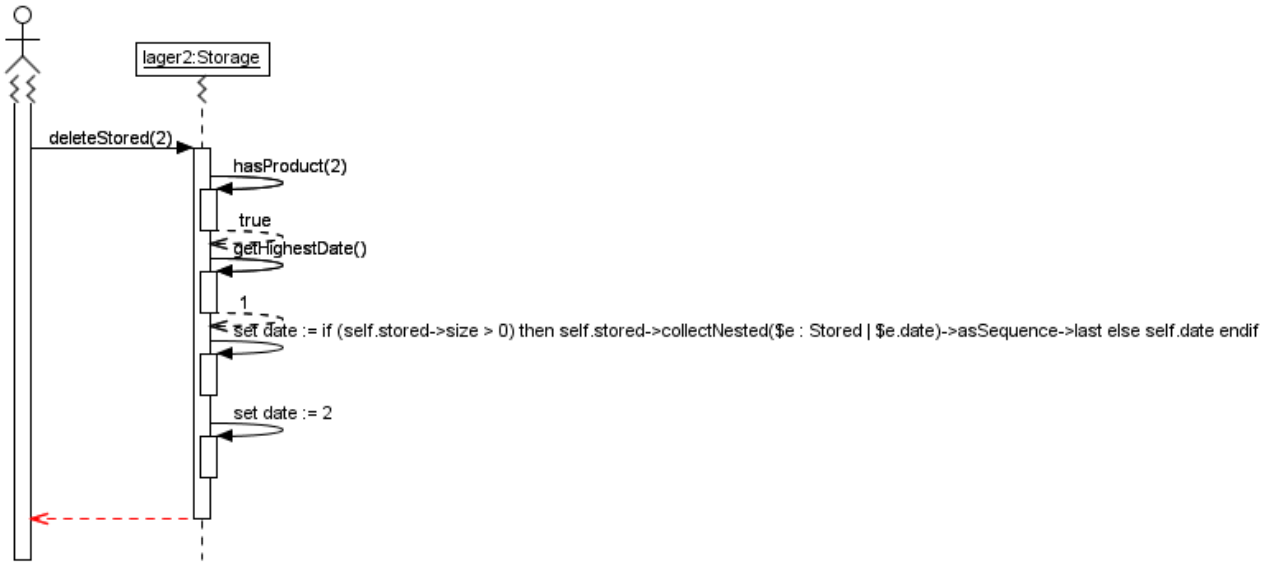

Abbildung 4.10: Das Sequenzdiagramm zu TC06 zeigt das fehlschlagen einer Postkondition aufgrund der vorherigen Manipulation des Datums von „lager2“.

4.1.7 TC07

Auch hier wird eine gültige Lagerverbindung zerstört, doch bauen wir diese vor Operationsabschluß wieder auf. Dadurch schlägt die Kondition *post storedIsDeleted?*[1] fehl.[2]

Die Kommandozeilenausgabe

```
1  tc07_deleteStored.command> !openter lager3 deleteStored(1)
2  precondition `parametersAreDefined' is true
3  precondition `storedExists' is true
4  tc07_deleteStored.command>
5  tc07_deleteStored.command>       open storage.deleteStored.command
6  storage.deleteStored.command> --Storage::deleteStoredProduct(
       aProductNumber : Integer)
7  storage.deleteStored.command>
8  storage.deleteStored.command>    !openter self hasProduct(
       aProductNumber)
9  storage.deleteStored.command>    !opexit self.hasProduct(aProductNumber
       )
10 storage.deleteStored.command>    !destroy self.findStored(
       aProductNumber,self.name)
11 storage.deleteStored.command>    !openter self getHighestDate()
12 storage.deleteStored.command>    !opexit self.getHighestDate()
13 storage.deleteStored.command>    !set self.date := if (self.stored->
       size()>0) then (self.stored.date -> asSequence -> last()) else (
       self.date) endif
14 storage.deleteStored.command>
15 storage.deleteStored.command>
16 tc07_deleteStored.command>       -- Behold as we thwart deleteStored's
       plans to delete a product's storage from lager3!
17 tc07_deleteStored.command>       !create gelagert6 : Stored between (
       kabel1, lager3)
18 tc07_deleteStored.command>       !set gelagert6.amount := 1
19 tc07_deleteStored.command>       !set gelagert6.date := 0
20 tc07_deleteStored.command>
21 tc07_deleteStored.command> !opexit
22 postcondition `dateIsConsistent' is true
23 postcondition `storedIsDeleted' is false
24 evaluation results:
25   self : Storage = @lager3
26   self.stored : Set(Stored) = Set{@gelagert6}
27   self.stored->size : Integer = 1
28   0 : Integer = 0
```

1 Seite 18
2 Siehe auch im Dateiverzeichnis Seite 89

```
29   (self.stored->size > 0) : Boolean = true
30   self : Storage = @lager3
31   self.product : Set(Product) = Set{@kabel1}
32   p : Product = @kabel1
33   p.number : Integer = 1
34   aProductNumber : Integer = 1
35   (p.number <> aProductNumber) : Boolean = false
36   self.product->forAll(p : Product | (p.number <> aProductNumber)) :
         Boolean = false
37   if (self.stored->size > 0) then self.product->forAll(p : Product | (
         p.number <> aProductNumber)) else true endif : Boolean = false
38 tc07_deleteStored.command>
```

Sequenzdiagramm

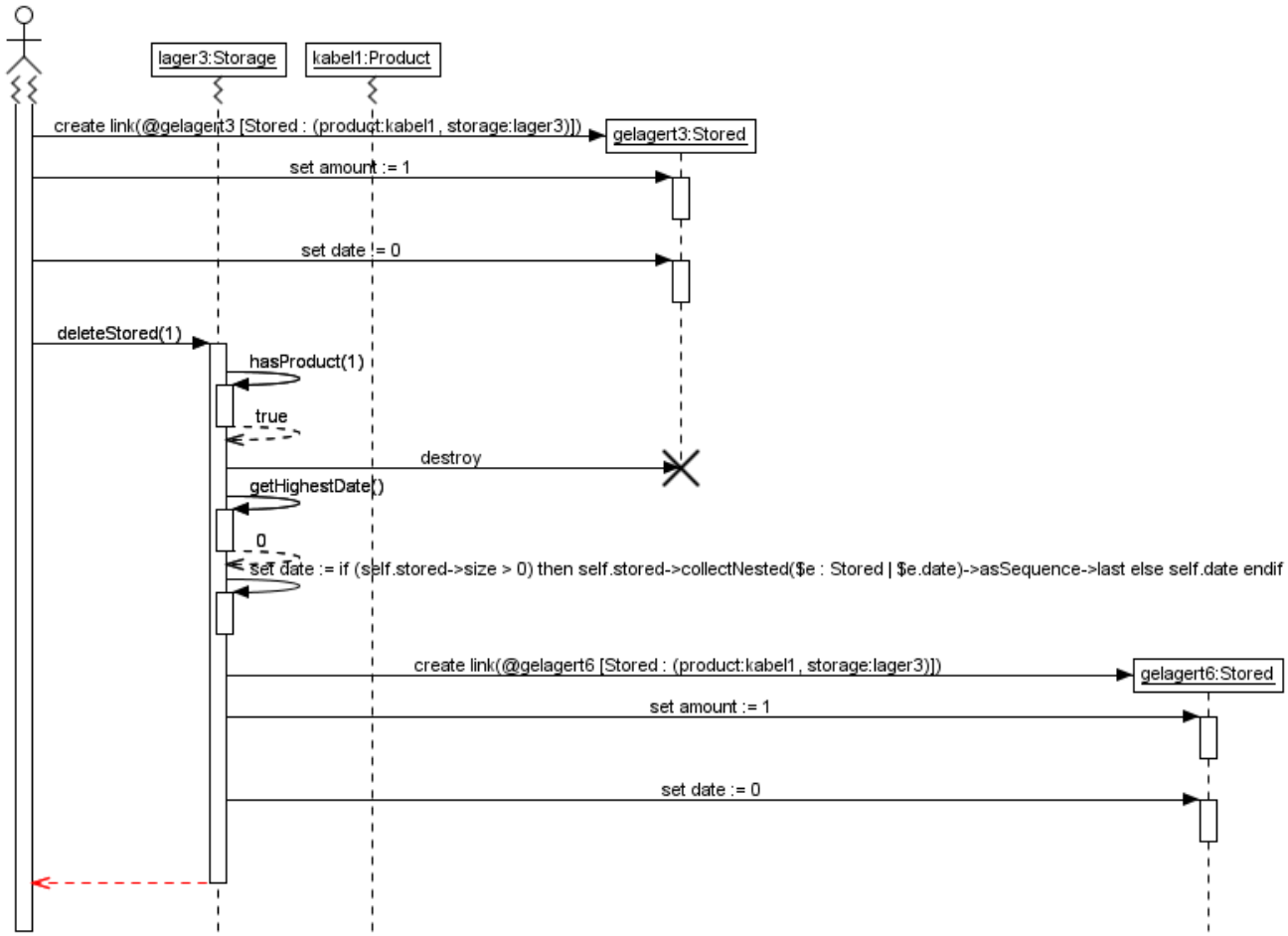

Abbildung 4.11: Das Sequenzdiagramm zu TC07 zeigt das fehlschlagen einer Postkondition aufgrund der Wiederherstellung einer Lagerverbindung zwischen „kabel1“ und „lager3“.

4.2 moveAmount()

Es ist zu bedenken, dass *moveAmount()*[1] aufgrund seiner vielen Fallunterscheidungen nicht so generisch aufgerufen werden kann, wie die anderen Operationen. Deswegen wird jeweils der gewünschte Fall angewendet.

Die Deklaration:

```
1 -- Moves the specified amount from the calling storage to the one set
    as a parameter. (The user has searched for a product and enters its
    number as identification.)
2  moveAmount(aProductNumber : Integer, anAmount : Real,
       aReceivingStorage : String)
3    -- Every parameter must be defined. No exceptions.
4  pre parametersAreDefined : aProductNumber.isDefined and anAmount.
       isDefined and aReceivingStorage.isDefined
5  -- Movement of negative amounts of products is stricly forbidden.
6  pre movingAmountIsPositive : anAmount > 0
7  -- The product must exist in this storage, duh.
8  pre storedExists : stored -> exists(s | s.product.number =
       aProductNumber)
9  -- There has to be enough that can be moved to begin with.
10 pre amountAvailable : stored -> any(s | s.product.number =
       aProductNumber).amount >= anAmount
11 -- There's got to be a place you can send your stuff to, shouldn't
       there? And this place should be different from the one you want
       to send things from.
12 pre receivingStorageExists : Storage.allInstances -> exists(s | s.
       name = aReceivingStorage)
13 -- The receiving Storage shouldn't be the sending Storage. That'd be
       stupid.
14 pre receivingStorageIsNotSending : self.name <> aReceivingStorage
15 -- If you've send everything from storage A to B, there shouldn't be
       a Stored associationclass in A belonging to the moved product.
16 post noEmptySendingStorage : if stored -> any(s | s.product.number =
       aProductNumber).amount@pre > anAmount
17      then (stored->any(s | s.product.number = aProductNumber).
           amount@pre - anAmount) > 0
18      else not stored->exists(s |  s.product.number = aProductNumber)
           endif
19 -- The receiving storage should have the sent amount stored.
20 post finallyStored :
21       let storedHere : Stored = Stored.allInstances -> any(s | s.
```

1 Seite 18

```
        product.number = aProductNumber and s.storage = self) in
22      let storedThere : Stored = Stored.allInstances -> any(s | s.
        storage.name = aReceivingStorage and s.product.number =
        aProductNumber) in
23      if storedThere.amount@pre.isUndefined and storedHere.amount.
        isDefined then
24      storedThere.amount = anAmount and
25      storedHere.amount =  storedHere.amount@pre - anAmount
26      else
27      if storedThere.amount@pre.isDefined and storedHere.amount.
        isUndefined then
28       storedThere.amount = storedThere.amount@pre + anAmount
29       else
30       if storedThere.amount@pre.isUndefined and storedHere.
         amount.isUndefined then
31        storedThere.amount = anAmount
32        else
33        storedThere.amount = storedThere.amount@pre +
          anAmount and
34        storedHere.amount =  storedHere.amount@pre -
          anAmount
35       endif
36      endif
37      endif
```

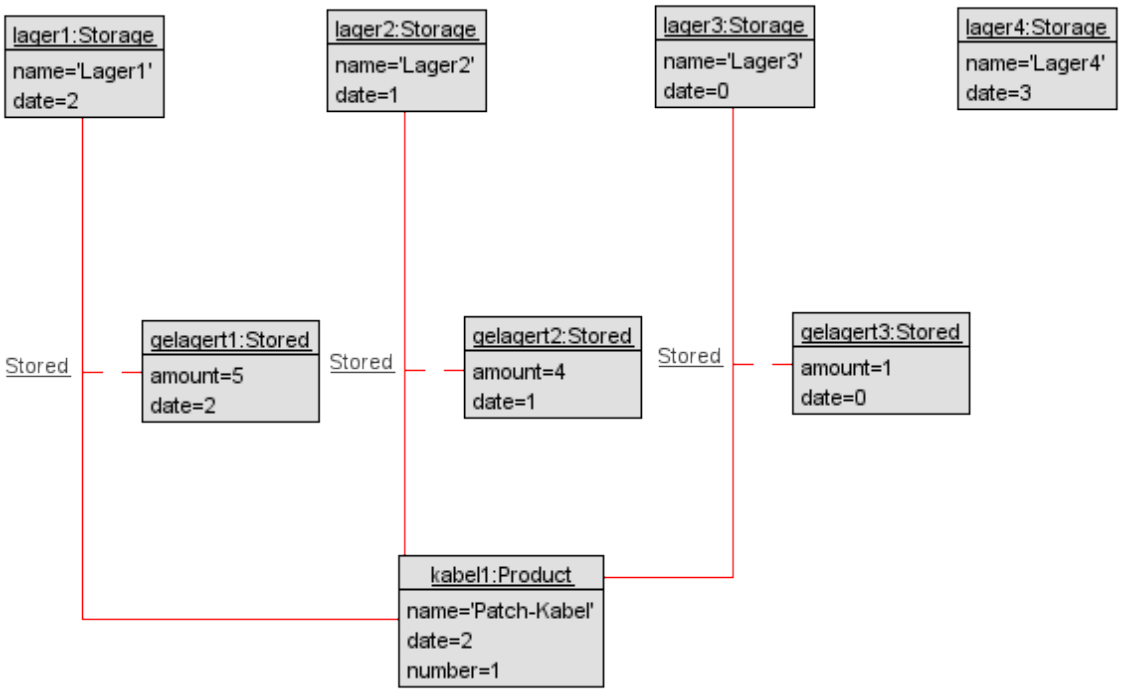

Abbildung 4.12: Der gültige Zustand für die Operation *moveAmount()*, die im folgendem getestet werden soll. Es werden nur die Produkte, die Lager und die Lagerverbindungen angezeigt.(Die Datei dazu findet sich auf Seite 88)

4.2.1 TC08

Es sollen 2 Einheiten von „kabel1“ in „lager3“ verschoben werden, was auch erfolgreich geschieht. [1]

Die Kommandozeilenausgabe

(Zeilenumbrüche leicht umformatiert)

```
1 tc08_moveAmount.command> !openter lager2 moveAmount (1,2,'Lager3')
2 precondition `parametersAreDefined' is true
3 precondition `movingAmountIsPositive' is true
4 precondition `storedExists' is true
5 precondition `amountAvailable' is true
6 precondition `receivingStorageExists' is true
7 precondition `receivingStorageIsNotSending' is true
8 tc08_moveAmount.command>
```

1 Siehe auch im Dateiverzeichnis Seite 89

```
9  tc08_moveAmount.command>        --Product number and storage are
       searched. let's assume we did that here..
10 tc08_moveAmount.command>        !openter self getProduct(
       aProductNumber)
11 tc08_moveAmount.command>        !opexit self.getProduct(aProductNumber
       )
12 tc08_moveAmount.command>
13 tc08_moveAmount.command>        !openter self.getProduct(
       aProductNumber) getAmount(self.name)
14 tc08_moveAmount.command>        !opexit self.getAmount(aStorageName)
15 tc08_moveAmount.command>
16 tc08_moveAmount.command>        !openter self.getProduct(
       aProductNumber) removeAmount(self.name, anAmount)
17 tc08_moveAmount.command>                !set self.stored->any(s|s.
       storage.name=aStorageName).amount := self.stored->any(s|s.storage.
       name=aStorageName).amount-anAmount
18 tc08_moveAmount.command>        !opexit
19 tc08_moveAmount.command>
20 tc08_moveAmount.command>        !openter self.getProduct(
       aProductNumber) getAmount(aReceivingStorage)
21 tc08_moveAmount.command>        !opexit lager2.findStored(1,
       aStorageName).amount
22 tc08_moveAmount.command>
23 tc08_moveAmount.command>        !openter self.getProduct(
       aProductNumber) addAmount(aReceivingStorage, anAmount)
24 tc08_moveAmount.command>                !set Product.allInstances ->
       any(p | p.number = 1).stored -> any(s | s.storage.name =
       aStorageName).amount := Product.allInstances -> any(p | p.number =
       1).stored -> any(s | s.storage.name = aStorageName).amount +
       anAmount
25 tc08_moveAmount.command>        !opexit
26 tc08_moveAmount.command>
27 tc08_moveAmount.command>        -- We're updating the Stored-date
       first by increasing it by one. If that new date is bigger than the
       one available in the product attributes, the latter will be updated
       . Finally, the storage's date will be set to the Stored one if the
       former is lesser than the latter.
28
29 tc08_moveAmount.command>        !openter self.findStored(
       aProductNumber, aReceivingStorage) update()
30 tc08_moveAmount.command>                open stored.update.command
31 stored.update.command>
32 stored.update.command>  !set self.date := self.date + 1
33 stored.update.command>  !openter self.product update(self.date)
34 stored.update.command>          !set self.date := if self.date >=
       aDate then self.date else aDate endif
35 stored.update.command>  !opexit self.date
```

```
36 stored.update.command>  !openter self.storage update(self.date)
37 stored.update.command>          !set self.date := if self.date >=
       aDate then self.date else aDate endif
38 stored.update.command>  !opexit self.date
39 stored.update.command>
40 tc08_moveAmount.command>        !opexit self.update()-1
41 tc08_moveAmount.command>
42 tc08_moveAmount.command> !opexit
43 postcondition `noEmptySendingStorage' is true
44 postcondition `finallyStored' is true
45 tc08_moveAmount.command>
```

Sequenzdiagramm

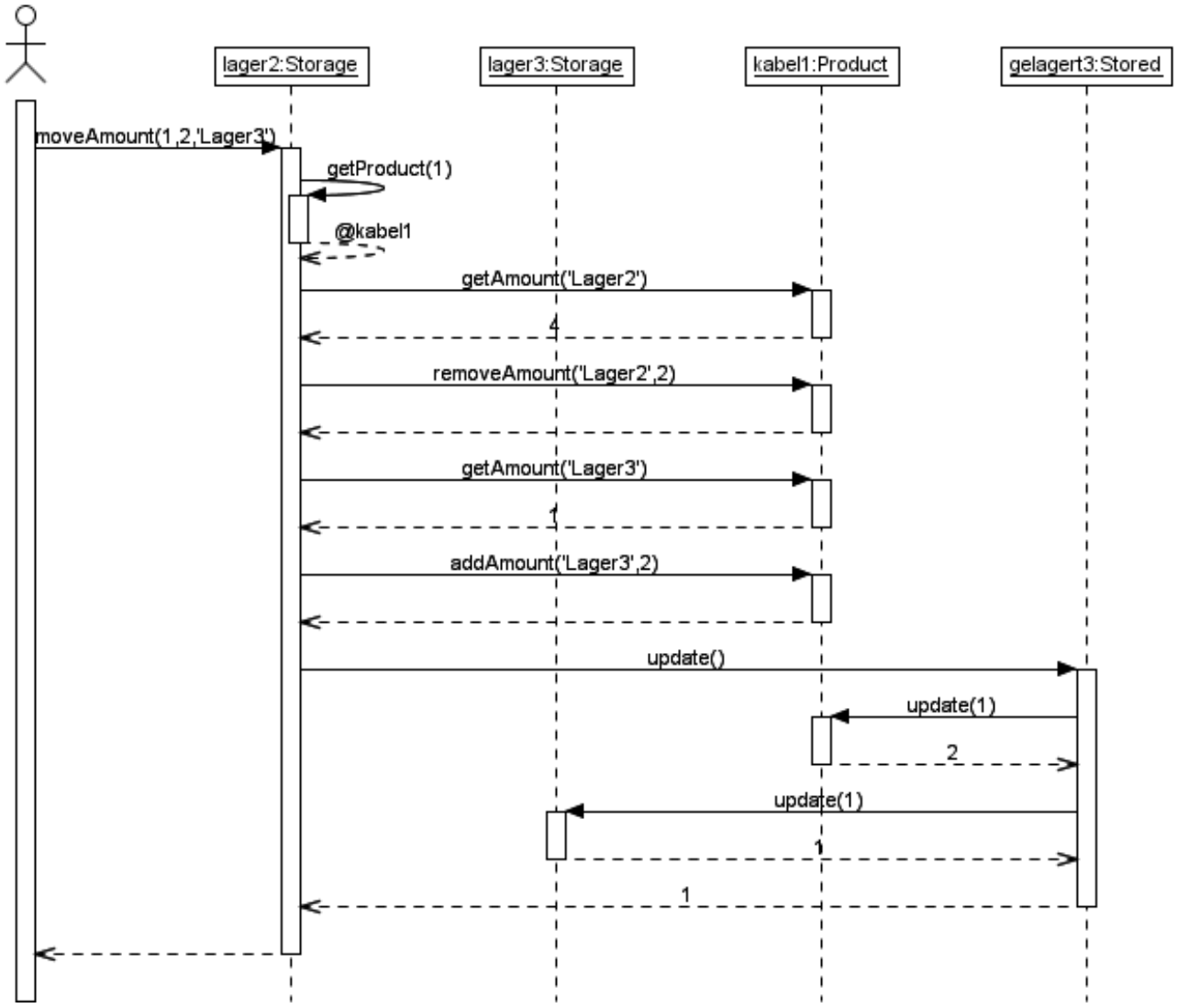

Abbildung 4.13: Das Sequenzdiagramm zu TC08 zeigt, wie das Produkt zunächst abgefragt, die Einheiten abgezogen und hinzugefügt und dann das Empfängerlager auf den neuesten Datumsstand gebracht wird. (Die Aktualisierung des Senderlagers ist zur Visulalisierung herausgekürzt worden.)

4.2.2 TC09

Hier machen wir dort weiter, wo wir im Testfall TC08 aufgehört haben, zuerst durchlaufen wir diesen und verschieben dann aus „lager1“ alle 5 Einheiten nach „lager4“, welches vorher

leer war. Da wir leider nicht die Verknüpfungsdaten der Lagerverbindung neu verlegen können um die Lagerverbindung einfach von „ 'lager1' nach 'lager5' zu schieben“, müssen wir zuerst die sendende Lagerverbindung löschen und dann eine empfangende erstellen. In diesem besonderen Fall muss ebenfalls darauf geachtet werden, dass die sendende Lagerverbindung nicht auf den neuesten Datumsstand gebracht werden kann, da sie nach der Operation nicht mehr existiert.[1]

Ein Sequenzdiagramm hierzu habe ich leider nicht in eine papierlesbaren Form bringen können.[2]

Die Kommandozeilenausgabe

(Zeilenumbrüche leicht umformatiert)

```
1  tc09_moveAmount.command> !openter lager1 moveAmount (1,5,'Lager4')
2  precondition `parametersAreDefined' is true
3  precondition `movingAmountIsPositive' is true
4  precondition `storedExists' is true
5  precondition `amountAvailable' is true
6  precondition `receivingStorageExists' is true
7  precondition `receivingStorageIsNotSending' is true
8  tc09_moveAmount.command>
9  tc09_moveAmount.command>          --Unfortunately, Stored is not an
       association but an association class. Therefore the following lines
        won't work and have been commented out:
10 tc09_moveAmount.command>          --!openter self relinkProduct(
       aProductNumber,aReceivingStorage)
11 tc09_moveAmount.command>          !openter self findStored(
       aProductNumber,self.name)
12 tc09_moveAmount.command>          !opexit self.findStored(aProductNumber
       ,self.name)
13 tc09_moveAmount.command>          --!delete (self.findProduct(
       aProductNumber),self) from self.findStored(aProductNumber, self.
       name)
14 tc09_moveAmount.command>          --!insert (Product.allInstances->any(p
        | p.number = aProductNumber),Storage.allInstances->any(s | s.name
       = aStorageName)) into self.findStored(aProductNumber, self.name)
15 tc09_moveAmount.command>          --!opexit
16 tc09_moveAmount.command>
17 tc09_moveAmount.command>          !openter Product.allInstances -> any(p
        | p.number = aProductNumber) getAmount(self.name)
18 tc09_moveAmount.command>          !opexit self.getAmount(aStorageName)
```

1 Siehe auch im Dateiverzeichnis Seite 89

2 Dabei habe ich festgestellt, dass es bei vielen Objekten sehr schwer sein kann, eines zu selektieren. Auch wenn dieses von keinem anderen Objekt überdeckt wird.

```
19 tc09_moveAmount.command>
20 tc09_moveAmount.command>          !openter self deleteStored(
       aProductNumber)
21 precondition `parametersAreDefined' is true
22 precondition `storedExists' is true
23 tc09_moveAmount.command>                    open storage.deleteStored.
       command
24 storage.deleteStored.command> --Storage::deleteStoredProduct(
       aProductNumber : Integer)
25 storage.deleteStored.command>
26 storage.deleteStored.command>    !openter self hasProduct(
       aProductNumber)
27 storage.deleteStored.command>    !opexit self.hasProduct(aProductNumber
       )
28 storage.deleteStored.command>    !destroy self.findStored(
       aProductNumber, self.name)
29 storage.deleteStored.command>    !openter self getHighestDate()
30 storage.deleteStored.command>    !opexit self.getHighestDate()
31 storage.deleteStored.command>    !set self.date := if (self.stored->
       size()>0) then (self.stored.date -> asSequence -> last()) else (
       self.date) endif
32 storage.deleteStored.command>
33 storage.deleteStored.command>
34 tc09_moveAmount.command>          !opexit
35 postcondition `dateIsConsistent' is true
36 postcondition `storedIsDeleted' is true
37 tc09_moveAmount.command>
38 tc09_moveAmount.command>          !openter Product.allInstances -> any(p
       | p.number = aProductNumber) getAmount(aReceivingStorage)
39 tc09_moveAmount.command>          !opexit self.getAmount(aStorageName)
40 tc09_moveAmount.command>
41 tc09_moveAmount.command>          !openter Storage.allInstances -> any(s
       | s.name = aReceivingStorage) addStored(aProductNumber)
42 tc09_moveAmount.command>                    !create gelagert4 : Stored
       between (kabel1, lager4)
43 tc09_moveAmount.command>                    !set gelagert4.amount := 0
44 tc09_moveAmount.command>                    !set gelagert4.date := lager4.
       date
45 tc09_moveAmount.command>          !opexit
46 tc09_moveAmount.command>
47 tc09_moveAmount.command>          !openter Product.allInstances -> any(p
       | p.number = aProductNumber) addAmount(aReceivingStorage, anAmount
       )
48 tc09_moveAmount.command>                    !set Product.allInstances ->
       any(p | p.number = 1).stored -> any(s | s.storage.name =
       aStorageName).amount := Product.allInstances -> any(p | p.number =
       1).stored -> any(s | s.storage.name = aStorageName).amount +
```

```
    anAmount
49 tc09_moveAmount.command>          !opexit
50 tc09_moveAmount.command>
51 tc09_moveAmount.command>          -- We're updating the Stored-date
    first by increasing it by one. If that new date is bigger than the
    one available in the product attributes, the latter will be updated
    . Finally, the storage's date will be set to the Stored one if the
    former is lesser than the latter.
52
53 tc09_moveAmount.command>          !openter self.findStored(
    aProductNumber, aReceivingStorage) update()
54 tc09_moveAmount.command>                  open stored.update.command
55 stored.update.command>
56 stored.update.command>  !set self.date := self.date + 1
57 stored.update.command>  !openter self.product update(self.date)
58 stored.update.command>          !set self.date := if self.date >=
    aDate then self.date else aDate endif
59 stored.update.command>  !opexit self.date
60 stored.update.command>  !openter self.storage update(self.date)
61 stored.update.command>          !set self.date := if self.date >=
    aDate then self.date else aDate endif
62 stored.update.command>  !opexit self.date
63 stored.update.command>
64 tc09_moveAmount.command>          !opexit self.update()-1
65 tc09_moveAmount.command>
66 tc09_moveAmount.command>          -- There is no Stored at the sending
    storage which could be updated.
67 tc09_moveAmount.command>          --!openter self.findStored(
    aProductNumber, self.name) update()
68 tc09_moveAmount.command>          --      open stored.update.command
69 tc09_moveAmount.command>          --!opexit self.update()-1
70 tc09_moveAmount.command>
71 tc09_moveAmount.command> !opexit
72 postcondition `noEmptySendingStorage' is true
73 postcondition `finallyStored' is true
74 tc09_moveAmount.command>
```

4.2.3 TC10

Wir wiederholen den Test TC09, doch ohne die sendende Lagerverbindung zu löschen, sodass die Kondition *post noEmptySendingStorage?*[1] und damit auch die *post finallyStored?*[2]fehlschlägt. [3]

1 Seite 20
2 Seite 20
3 Siehe auch im Dateiverzeichnis Seite 89

Die Kommandozeilenausgabe

(Zeilenumbrüche leicht umformatiert)

```
1  tc10_moveAmount.command>          -- Uups, guess we just forgot to
       delete the sending Stored...
2  tc10_moveAmount.command>          --!openter self deleteStored(
       aProductNumber)
3  tc10_moveAmount.command>          --       open storage.deleteStored.
       command
4  tc10_moveAmount.command>          --!opexit
5  tc10_moveAmount.command>
6  tc10_moveAmount.command>          !openter Product.allInstances -> any(p
       | p.number = aProductNumber) getAmount(aReceivingStorage)
7  tc10_moveAmount.command>          !opexit self.getAmount(aStorageName)
8  tc10_moveAmount.command>
9  tc10_moveAmount.command>          !openter Storage.allInstances -> any(s
       | s.name = aReceivingStorage) addStored(aProductNumber)
10 tc10_moveAmount.command>                    !create gelagert4 : Stored
       between (kabel1, lager4)
11 tc10_moveAmount.command>                    !set gelagert4.amount := 0
12 tc10_moveAmount.command>                    !set gelagert4.date := lager4.
       date
13 tc10_moveAmount.command>          !opexit
14 tc10_moveAmount.command>
15 tc10_moveAmount.command>          !openter Product.allInstances -> any(p
       | p.number = aProductNumber) addAmount(aReceivingStorage, anAmount
       )
16 tc10_moveAmount.command>                    !set Product.allInstances ->
       any(p | p.number = 1).stored -> any(s | s.storage.name =
       aStorageName).amount := Product.allInstances -> any(p | p.number =
       1).stored -> any(s | s.storage.name = aStorageName).amount +
       anAmount
17 tc10_moveAmount.command>          !opexit
18 tc10_moveAmount.command>
19 tc10_moveAmount.command>          -- We're updating the Stored-date
       first by increasing it by one. If that new date is bigger than the
       one available in the product attributes, the latter will be updated
       . Finally, the storage's date will be set to the Stored one if the
       former is lesser than the latter.
20
21 tc10_moveAmount.command>          !openter self.findStored(
       aProductNumber, aReceivingStorage) update()
22 tc10_moveAmount.command>                    open stored.update.command
23 stored.update.command>
24 stored.update.command>  !set self.date := self.date + 1
25 stored.update.command>  !openter self.product update(self.date)
```

```
stored.update.command>          !set self.date := if self.date >=
    aDate then self.date else aDate endif
stored.update.command>  !opexit self.date
stored.update.command>  !openter self.storage update(self.date)
stored.update.command>          !set self.date := if self.date >=
    aDate then self.date else aDate endif
stored.update.command>  !opexit self.date
stored.update.command>
tc10_moveAmount.command>        !opexit self.update()-1
tc10_moveAmount.command>
tc10_moveAmount.command>        -- There is no Stored at the sending
    storage which could be updated.
tc10_moveAmount.command>        --!openter self.findStored(
    aProductNumber, self.name) update()
tc10_moveAmount.command>        --      open stored.update.command
tc10_moveAmount.command>        --!opexit self.update()-1
tc10_moveAmount.command>
tc10_moveAmount.command> !opexit
postcondition `noEmptySendingStorage' is false
evaluation results:
  self : Storage = @lager1
  self.stored : Set(Stored) = Set{@gelagert1}
  s : Stored = @gelagert1
  s.product : Product = @kabel1
  s.product.number : Integer = 1
  aProductNumber : Integer = 1
  (s.product.number = aProductNumber) : Boolean = true
  self : Storage = @lager1
  self.stored : Set(Stored) = Set{@gelagert1}
  s : Stored = @gelagert1
  s.product : Product = @kabel1
  s.product.number : Integer = 1
  aProductNumber : Integer = 1
  (s.product.number = aProductNumber) : Boolean = true
  self.stored->any(s : Stored | (s.product.number = aProductNumber)) :
      Stored = @gelagert1
  self.stored->any(s : Stored | (s.product.number = aProductNumber)).
     amount@pre : Integer = 5
  anAmount : Integer = 5
  (self.stored->any(s : Stored | (s.product.number = aProductNumber)).
     amount@pre > anAmount) : Boolean = false
  self : Storage = @lager1
  self.stored : Set(Stored) = Set{@gelagert1}
  s : Stored = @gelagert1
  s.product : Product = @kabel1
  s.product.number : Integer = 1
  aProductNumber : Integer = 1
```

```
66    (s.product.number = aProductNumber) : Boolean = true
67    self.stored->exists(s : Stored | (s.product.number = aProductNumber)
          ) : Boolean = true
68    not self.stored->exists(s : Stored | (s.product.number =
          aProductNumber)) : Boolean = false
69    if (self.stored->any(s : Stored | (s.product.number = aProductNumber
          )).amount@pre > anAmount) then ((self.stored->any(s : Stored | (s
          .product.number = aProductNumber)).amount@pre - anAmount) > 0)
          else not self.stored->exists(s : Stored | (s.product.number =
          aProductNumber)) endif : Boolean = false
70
71  postcondition `finallyStored' is false
72  evaluation results:
73    Stored.allInstances : Set(Stored) = Set{@gelagert1,@gelagert2,
          @gelagert3,@gelagert4}
74    s : Stored = @gelagert1
75    s.product : Product = @kabel1
76    s.product.number : Integer = 1
77    aProductNumber : Integer = 1
78    (s.product.number = aProductNumber) : Boolean = true
79    s : Stored = @gelagert1
80    s.storage : Storage = @lager1
81    self : Storage = @lager1
82    (s.storage = self) : Boolean = true
83    ((s.product.number = aProductNumber) and (s.storage = self)) :
          Boolean = true
84    s : Stored = @gelagert2
85    s.product : Product = @kabel1
86    s.product.number : Integer = 1
87    aProductNumber : Integer = 1
88    (s.product.number = aProductNumber) : Boolean = true
89    s : Stored = @gelagert2
90    s.storage : Storage = @lager2
91    self : Storage = @lager1
92    (s.storage = self) : Boolean = false
93    ((s.product.number = aProductNumber) and (s.storage = self)) :
          Boolean = false
94    s : Stored = @gelagert3
95    s.product : Product = @kabel1
96    s.product.number : Integer = 1
97    aProductNumber : Integer = 1
98    (s.product.number = aProductNumber) : Boolean = true
99    s : Stored = @gelagert3
100   s.storage : Storage = @lager3
101   self : Storage = @lager1
102   (s.storage = self) : Boolean = false
```

```
103  ((s.product.number = aProductNumber) and (s.storage = self)) :
         Boolean = false
104  s : Stored = @gelagert4
105  s.product : Product = @kabel1
106  s.product.number : Integer = 1
107  aProductNumber : Integer = 1
108  (s.product.number = aProductNumber) : Boolean = true
109  s : Stored = @gelagert4
110  s.storage : Storage = @lager4
111  self : Storage = @lager1
112  (s.storage = self) : Boolean = false
113  ((s.product.number = aProductNumber) and (s.storage = self)) :
         Boolean = false
114  Stored.allInstances : Set(Stored) = Set{@gelagert1,@gelagert2,
         @gelagert3,@gelagert4}
115  s : Stored = @gelagert1
116  s.product : Product = @kabel1
117  s.product.number : Integer = 1
118  aProductNumber : Integer = 1
119  (s.product.number = aProductNumber) : Boolean = true
120  s : Stored = @gelagert1
121  s.storage : Storage = @lager1
122  self : Storage = @lager1
123  (s.storage = self) : Boolean = true
124  ((s.product.number = aProductNumber) and (s.storage = self)) :
         Boolean = true
125  s : Stored = @gelagert2
126  s.product : Product = @kabel1
127  s.product.number : Integer = 1
128  aProductNumber : Integer = 1
129  (s.product.number = aProductNumber) : Boolean = true
130  s : Stored = @gelagert2
131  s.storage : Storage = @lager2
132  self : Storage = @lager1
133  (s.storage = self) : Boolean = false
134  ((s.product.number = aProductNumber) and (s.storage = self)) :
         Boolean = false
135  s : Stored = @gelagert3
136  s.product : Product = @kabel1
137  s.product.number : Integer = 1
138  aProductNumber : Integer = 1
139  (s.product.number = aProductNumber) : Boolean = true
140  s : Stored = @gelagert3
141  s.storage : Storage = @lager3
142  self : Storage = @lager1
143  (s.storage = self) : Boolean = false
```

```
144 ((s.product.number = aProductNumber) and (s.storage = self)) :
        Boolean = false
145 s : Stored = @gelagert4
146 s.product : Product = @kabel1
147 s.product.number : Integer = 1
148 aProductNumber : Integer = 1
149 (s.product.number = aProductNumber) : Boolean = true
150 s : Stored = @gelagert4
151 s.storage : Storage = @lager4
152 self : Storage = @lager1
153 (s.storage = self) : Boolean = false
154 ((s.product.number = aProductNumber) and (s.storage = self)) :
        Boolean = false
155 Stored.allInstances->any(s : Stored | ((s.product.number =
        aProductNumber) and (s.storage = self))) : Stored = @gelagert1
156 Stored.allInstances : Set(Stored) = Set{@gelagert1,@gelagert2,
        @gelagert3,@gelagert4}
157 s : Stored = @gelagert1
158 s.storage : Storage = @lager1
159 s.storage.name : String = 'Lager1'
160 aReceivingStorage : String = 'Lager4'
161 (s.storage.name = aReceivingStorage) : Boolean = false
162 s : Stored = @gelagert1
163 s.product : Product = @kabel1
164 s.product.number : Integer = 1
165 aProductNumber : Integer = 1
166 (s.product.number = aProductNumber) : Boolean = true
167 ((s.storage.name = aReceivingStorage) and (s.product.number =
        aProductNumber)) : Boolean = false
168 s : Stored = @gelagert2
169 s.storage : Storage = @lager2
170 s.storage.name : String = 'Lager2'
171 aReceivingStorage : String = 'Lager4'
172 (s.storage.name = aReceivingStorage) : Boolean = false
173 s : Stored = @gelagert2
174 s.product : Product = @kabel1
175 s.product.number : Integer = 1
176 aProductNumber : Integer = 1
177 (s.product.number = aProductNumber) : Boolean = true
178 ((s.storage.name = aReceivingStorage) and (s.product.number =
        aProductNumber)) : Boolean = false
179 s : Stored = @gelagert3
180 s.storage : Storage = @lager3
181 s.storage.name : String = 'Lager3'
182 aReceivingStorage : String = 'Lager4'
183 (s.storage.name = aReceivingStorage) : Boolean = false
184 s : Stored = @gelagert3
```

```
185  s.product : Product = @kabel1
186  s.product.number : Integer = 1
187  aProductNumber : Integer = 1
188  (s.product.number = aProductNumber) : Boolean = true
189  ((s.storage.name = aReceivingStorage) and (s.product.number =
         aProductNumber)) : Boolean = false
190  s : Stored = @gelagert4
191  s.storage : Storage = @lager4
192  s.storage.name : String = 'Lager4'
193  aReceivingStorage : String = 'Lager4'
194  (s.storage.name = aReceivingStorage) : Boolean = true
195  s : Stored = @gelagert4
196  s.product : Product = @kabel1
197  s.product.number : Integer = 1
198  aProductNumber : Integer = 1
199  (s.product.number = aProductNumber) : Boolean = true
200  ((s.storage.name = aReceivingStorage) and (s.product.number =
         aProductNumber)) : Boolean = true
201  Stored.allInstances : Set(Stored) = Set{@gelagert1,@gelagert2,
         @gelagert3,@gelagert4}
202  s : Stored = @gelagert1
203  s.storage : Storage = @lager1
204  s.storage.name : String = 'Lager1'
205  aReceivingStorage : String = 'Lager4'
206  (s.storage.name = aReceivingStorage) : Boolean = false
207  s : Stored = @gelagert1
208  s.product : Product = @kabel1
209  s.product.number : Integer = 1
210  aProductNumber : Integer = 1
211  (s.product.number = aProductNumber) : Boolean = true
212  ((s.storage.name = aReceivingStorage) and (s.product.number =
         aProductNumber)) : Boolean = false
213  s : Stored = @gelagert2
214  s.storage : Storage = @lager2
215  s.storage.name : String = 'Lager2'
216  aReceivingStorage : String = 'Lager4'
217  (s.storage.name = aReceivingStorage) : Boolean = false
218  s : Stored = @gelagert2
219  s.product : Product = @kabel1
220  s.product.number : Integer = 1
221  aProductNumber : Integer = 1
222  (s.product.number = aProductNumber) : Boolean = true
223  ((s.storage.name = aReceivingStorage) and (s.product.number =
         aProductNumber)) : Boolean = false
224  s : Stored = @gelagert3
225  s.storage : Storage = @lager3
226  s.storage.name : String = 'Lager3'
```

```
227 aReceivingStorage : String = 'Lager4'
228 (s.storage.name = aReceivingStorage) : Boolean = false
229 s : Stored = @gelagert3
230 s.product : Product = @kabel1
231 s.product.number : Integer = 1
232 aProductNumber : Integer = 1
233 (s.product.number = aProductNumber) : Boolean = true
234 ((s.storage.name = aReceivingStorage) and (s.product.number =
        aProductNumber)) : Boolean = false
235 s : Stored = @gelagert4
236 s.storage : Storage = @lager4
237 s.storage.name : String = 'Lager4'
238 aReceivingStorage : String = 'Lager4'
239 (s.storage.name = aReceivingStorage) : Boolean = true
240 s : Stored = @gelagert4
241 s.product : Product = @kabel1
242 s.product.number : Integer = 1
243 aProductNumber : Integer = 1
244 (s.product.number = aProductNumber) : Boolean = true
245 ((s.storage.name = aReceivingStorage) and (s.product.number =
        aProductNumber)) : Boolean = true
246 Stored.allInstances->any(s : Stored | ((s.storage.name =
        aReceivingStorage) and (s.product.number = aProductNumber))) :
        Stored = @gelagert4
247 storedThere : Stored = @gelagert4
248 storedThere.amount@pre : Real = Undefined
249 storedThere.amount@pre.isUndefined : Boolean = true
250 storedHere : Stored = @gelagert1
251 storedHere.amount : Integer = 5
252 storedHere.amount.isDefined : Boolean = true
253 (storedThere.amount@pre.isUndefined and storedHere.amount.isDefined)
        : Boolean = true
254 storedThere : Stored = @gelagert4
255 storedThere.amount : Integer = 5
256 anAmount : Integer = 5
257 (storedThere.amount = anAmount) : Boolean = true
258 storedHere : Stored = @gelagert1
259 storedHere.amount : Integer = 5
260 storedHere : Stored = @gelagert1
261 storedHere.amount@pre : Integer = 5
262 anAmount : Integer = 5
263 (storedHere.amount@pre - anAmount) : Integer = 0
264 (storedHere.amount = (storedHere.amount@pre - anAmount)) : Boolean =
        false
265 ((storedThere.amount = anAmount) and (storedHere.amount = (
        storedHere.amount@pre - anAmount))) : Boolean = false
```

```
266   if (storedThere.amount@pre.isUndefined and storedHere.amount.
          isDefined) then ((storedThere.amount = anAmount) and (storedHere.
          amount = (storedHere.amount@pre - anAmount))) else if (
          storedThere.amount@pre.isDefined and storedHere.amount.
          isUndefined) then (storedThere.amount = (storedThere.amount@pre +
           anAmount)) else if (storedThere.amount@pre.isUndefined and
          storedHere.amount.isUndefined) then (storedThere.amount =
          anAmount) else ((storedThere.amount = (storedThere.amount@pre +
          anAmount)) and (storedHere.amount = (storedHere.amount@pre -
          anAmount))) endif endif endif : Boolean = false
267   let storedThere : Stored = Stored.allInstances->any(s : Stored | ((s
          .storage.name = aReceivingStorage) and (s.product.number =
          aProductNumber))) in
268 if (storedThere.amount@pre.isUndefined and storedHere.amount.isDefined
      ) then ((storedThere.amount = anAmount) and (storedHere.amount = (
      storedHere.amount@pre - anAmount))) else if (storedThere.amount@pre
      .isDefined and storedHere.amount.isUndefined) then (storedThere.
      amount = (storedThere.amount@pre+ anAmount)) else if (storedThere.
      amount@pre.isUndefined and storedHere.amount.isUndefined) then (
      storedThere.amount = anAmount) else ((storedThere.amount = (
      storedThere.amount@pre + anAmount)) and (storedHere.amount = (
      storedHere.amount@pre - anAmount))) endif endif endif : Boolean =
      false
269   let storedHere : Stored = Stored.allInstances->any(s : Stored | ((s.
          product.number = aProductNumber) and (s.storage = self))) in
270   let storedThere : Stored = Stored.allInstances->any(s : Stored | ((s
          .storage.name = aReceivingStorage) and (s.product.number =
          aProductNumber))) in
271   if (storedThere.amount@pre.isUndefined and storedHere.amount.
          isDefined) then ((storedThere.amount = anAmount) and (storedHere.
          amount = (storedHere.amount@pre - anAmount))) else if (
          storedThere.amount@pre.isDefined and storedHere.amount.
          isUndefined) then (storedThere.amount = (storedThere.amount@pre +
           anAmount)) else if(storedThere.amount@pre.isUndefined and
          storedHere.amount.isUndefined) then (storedThere.amount =
          anAmount) else ((storedThere.amount = (storedThere.amount@pre +
          anAmount)) and (storedHere.amount = (storedHere.amount@pre -
          anAmount))) endif endif endif : Boolean = false
272 tc10_moveAmount.command>
```

4.2.4 TC11

Wir wiederholen TC09 abermals, doch dieses Mal versuchen wir alles von „lager1“ in sich selber zu verschieben. Die Kondition *pre receivingStorageIsNotSending?*[1] schlägt wie er-

1 Seite 19

wartet fehl.[1] Da die Operation aber trotzdem nicht abgebrochen wird und der Objektname per Hand eingefügt werden musste, schlagen nun auch noch zwei Invaranten fehl:

1. *hasLatestDate?*[2], da die neue, fälschlicherweise eingeführte Lagerverbindung ein aktuelleres Datum trägt.
2. *amountIsPositive?*, da die neue Lagerverbindung als Standardstückzahl 0 beträgt.

Die Kommandozeilenausgabe

(Zeilenumbrüche leicht umformatiert)

```
1  tc11_moveAmount.command> !openter lager1 moveAmount (1,5,'Lager1')
2  precondition `parametersAreDefined' is true
3  precondition `movingAmountIsPositive' is true
4  precondition `storedExists' is true
5  precondition `amountAvailable' is true
6  precondition `receivingStorageExists' is true
7  precondition `receivingStorageIsNotSending' is false
8  tc11_moveAmount.command>
9  tc11_moveAmount.command>          --Unfortunately, Stored is not an
       association but an association class. Therefore the following lines
       won't work and have been commented out:
10 tc11_moveAmount.command>          --!openter self relinkProduct(
       aProductNumber,aReceivingStorage)
11 tc11_moveAmount.command>          !openter self findStored(
       aProductNumber,self.name)
12 null:1:8: Undefined variable `self'.
13 tc11_moveAmount.command>          !opexit self.findStored(aProductNumber
       ,self.name)
14 null:1:7: Undefined variable `self'.
15 tc11_moveAmount.command>          --!delete (self.findProduct(
       aProductNumber),self) from self.findStored(aProductNumber, self.
       name)
16 tc11_moveAmount.command>          --!insert (Product.allInstances->any(p
       | p.number = aProductNumber),Storage.allInstances->any(s | s.name
       = aStorageName)) into self.findStored(aProductNumber, self.name)
17 tc11_moveAmount.command>          --!opexit
18 tc11_moveAmount.command>
19 tc11_moveAmount.command>          !openter Product.allInstances -> any(p
       | p.number = aProductNumber) getAmount(self.name)
20 null:1:51: Undefined variable `aProductNumber'.
```

1 Siehe auch im Dateiverzeichnis Seite 89
2 Seite 14

```
21 tc11_moveAmount.command>          !opexit self.getAmount(aStorageName)
22 null:1:7: Undefined variable `self'.
23 tc11_moveAmount.command>
24 tc11_moveAmount.command>          !openter self deleteStored(
       aProductNumber)
25 null:1:8: Undefined variable `self'.
26 tc11_moveAmount.command>                    open storage.deleteStored.
       command
27 storage.deleteStored.command> --Storage::deleteStoredProduct(
       aProductNumber : Integer)
28 storage.deleteStored.command>
29 storage.deleteStored.command>    !openter self hasProduct(
       aProductNumber)
30 null:1:8: Undefined variable `self'.
31 storage.deleteStored.command>    !opexit self.hasProduct(aProductNumber
       )
32 null:1:7: Undefined variable `self'.
33 storage.deleteStored.command>    !destroy self.findStored(
       aProductNumber,self.name)
34 null:1:8: Undefined variable `self'.
35 storage.deleteStored.command>    !openter self getHighestDate()
36 null:1:8: Undefined variable `self'.
37 storage.deleteStored.command>    !opexit self.getHighestDate()
38 null:1:7: Undefined variable `self'.
39 storage.deleteStored.command>    !set self.date := if (self.stored->
       size()>0) then (self.stored.date -> asSequence -> last()) else (
       self.date) endif
40 null:1:4: Undefined variable `self'.
41 storage.deleteStored.command>
42 storage.deleteStored.command>
43 tc11_moveAmount.command>          !opexit
44 Error: Call stack is empty.
45 tc11_moveAmount.command>
46 tc11_moveAmount.command>          !openter Product.allInstances -> any(p
       | p.number = aProductNumber) getAmount(aReceivingStorage)
47 null:1:51: Undefined variable `aProductNumber'.
48 tc11_moveAmount.command>          !opexit self.getAmount(aStorageName)
49 null:1:7: Undefined variable `self'.
50 tc11_moveAmount.command>
51 tc11_moveAmount.command>          !openter Storage.allInstances -> any(s
       | s.name = aReceivingStorage) addStored(aProductNumber)
52 null:1:49: Undefined variable `aReceivingStorage'.
53 tc11_moveAmount.command>                    !create gelagert4 : Stored
       between (kabel1, lager4)
54 tc11_moveAmount.command>                    !set gelagert4.amount := 0
55 tc11_moveAmount.command>                    !set gelagert4.date := lager4.
       date
```

```
56 tc11_moveAmount.command>          !opexit
57 Error: Call stack is empty.
58 tc11_moveAmount.command>
59 tc11_moveAmount.command>          !openter Product.allInstances -> any(p
       | p.number = aProductNumber) addAmount(aReceivingStorage, anAmount
       )
60 null:1:51: Undefined variable `aProductNumber'.
61 tc11_moveAmount.command>                    !set Product.allInstances ->
       any(p | p.number = 1).stored -> any(s | s.storage.name =
       aStorageName).amount := Product.allInstances -> any(p | p.number =
       1).stored -> any(s | s.storage.name = aStorageName).amount +
       anAmount
62 null:1:85: Undefined variable `aStorageName'.
63 tc11_moveAmount.command>          !opexit
64 Error: Call stack is empty.
65 tc11_moveAmount.command>
66 tc11_moveAmount.command>          -- We're updating the Stored-date
       first by increasing it by one. If that new date is bigger than the
       one available in the product attributes, the latter will be updated
       . Finally, the storage's date will be set to the Stored one if the
       former is lesser than the latter.
67
68 tc11_moveAmount.command>          !openter self.findStored(
       aProductNumber, aReceivingStorage) update()
69 null:1:8: Undefined variable `self'.
70 tc11_moveAmount.command>                    open stored.update.command
71 stored.update.command>
72 stored.update.command>  !set self.date := self.date + 1
73 null:1:4: Undefined variable `self'.
74 stored.update.command>  !openter self.product update(self.date)
75 null:1:8: Undefined variable `self'.
76 stored.update.command>          !set self.date := if self.date >=
       aDate then self.date else aDate endif
77 null:1:4: Undefined variable `self'.
78 stored.update.command>  !opexit self.date
79 null:1:7: Undefined variable `self'.
80 stored.update.command>  !openter self.storage update(self.date)
81 null:1:8: Undefined variable `self'.
82 stored.update.command>          !set self.date := if self.date >=
       aDate then self.date else aDate endif
83 null:1:4: Undefined variable `self'.
84 stored.update.command>  !opexit self.date
85 null:1:7: Undefined variable `self'.
86 stored.update.command>
87 tc11_moveAmount.command>          !opexit self.update()-1
88 null:1:7: Undefined variable `self'.
89 tc11_moveAmount.command>
```

```
tc11_moveAmount.command>          -- There is no Stored at the sending
    storage which could be updated.
tc11_moveAmount.command>          --!openter self.findStored(
    aProductNumber, self.name) update()
tc11_moveAmount.command>          --      open stored.update.command
tc11_moveAmount.command>          --!opexit self.update()-1
tc11_moveAmount.command>
tc11_moveAmount.command> !opexit
Error: Call stack is empty.
tc11_moveAmount.command>
```

4.3 restock()

In den zwei nun folgenden Testfällen muss ich eine Schleife manuell abarbeiten. Es werden alle *Stored*-Objekte, die eine Stückzahl kleiner als die gesuchte aufweisen, in einer Menge gesammelt. Daraus wird dann das erste Element wieder aufgefüllt. Solange bis die Menge leer ist.

Die Deklaration:

```
-- If the stored amount of a product is below the specified amount, it
    is restocked up to the second specified amount by using the
    cheapest available contractor. This operation should produce a
    receipt which lists the amounts, total prices and the ceapest
    contractor.
 restock(aSearchAmount : Real, aNewAmount : Real):Tuple(boughtUnits:
    Real, totalCost:Real, pricePerUnit:Real, cheapestSupplier:String)
 -- Parameters have to be defined.
 pre parametersAreDefined : aSearchAmount.isDefined and aNewAmount.
    isDefined
 -- The specified amounts are supposed to be positive.
 pre amountsArePositive : (aSearchAmount >= 0) and (aNewAmount > 0)
 -- The amount to be searched for should be smaller than the one we
    aim to restock with.
 pre amountsAreBalanced : aSearchAmount < aNewAmount
 -- After restocking, the desired amount should be available.
 post amountRestocked : self.stored -> collect(s | s.amount@pre) ->
    sum() + result.boughtUnits = self.stored.amount -> sum()
```

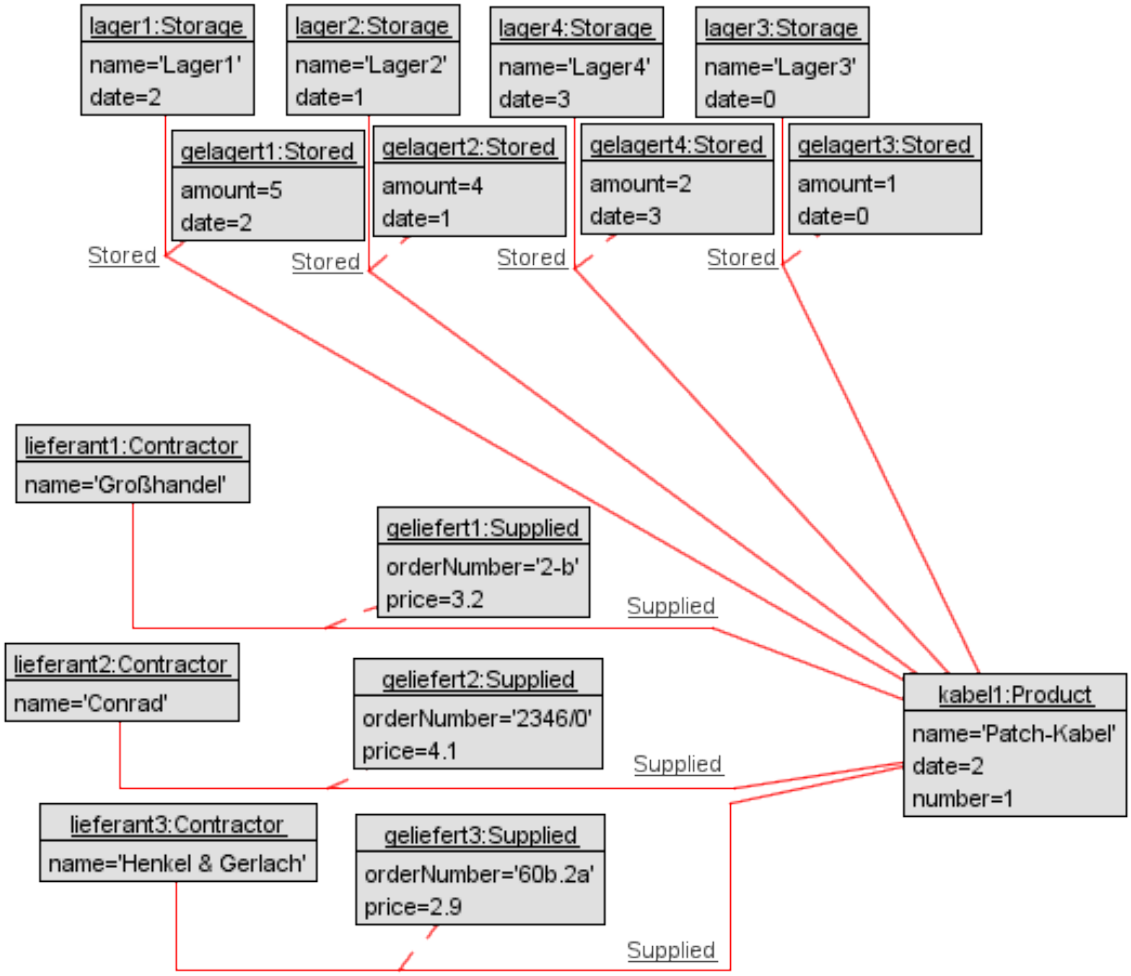

Abbildung 4.14: Der gültige Zustand für die Operation *restock()*, die im folgendem getestet werden soll. Es werden nur die Produkte, die Lager, Lieferanten und die entsprechenden Verbindungen angezeigt. (Die Datei dazu findet sich auf Seite 88)

4.3.1 TC12

„gelagert2“, „gelagert3“ und „gelagert4“ sollen die Stückzahl auf 6 anheben. Darüber soll eine Rechnung geschrieben werden, die den günstigsten Lieferanten berücksichtigt, die gekaufte Stückzahl, den Einzelpreis und den Gesamtpreis angibt. In diesem Testfall hat im Nachhinein nur noch „gelagert1“ 5 Einheiten, während alle anderen auf 6 gesetzt wurden.[1]

1 Siehe auch im Dateiverzeichnis Seite 89

Die Kommandozeilenausgabe

(Zeilenumbrüche leicht umformatiert)

```
1  tc12_restock.command> !openter kabel1 restock(4,6)
2  precondition `parametersAreDefined' is true
3  precondition `amountsArePositive' is true
4  precondition `amountsAreBalanced' is true
5  tc12_restock.command>
6  tc12_restock.command>    open storage.restock.command
7  storage.restock.command> !openter self findCheapestContractor()
8  storage.restock.command>         !opexit self.findCheapestContractor()
9  postcondition `foundCheapest' is true
10 storage.restock.command>
11 storage.restock.command>         !openter self findProductsAmountLesser
       (aSearchAmount)
12 storage.restock.command>         !opexit self.findProductsAmountLesser(
       anAmount)
13 storage.restock.command>
14 storage.restock.command>         !create quittung01 : Receipt
15 storage.restock.command>         !set quittung01.content := Tuple{
       boughtUnits = (self.findProductsAmountLesser(aSearchAmount).amount
       ->size()*aNewAmount) - self.findProductsAmountLesser(aSearchAmount)
       .amount->sum(), totalCost = ((self.findProductsAmountLesser(
       aSearchAmount).amount->size()*aNewAmount)- self.
       findProductsAmountLesser(aSearchAmount).amount->sum())*self.
       findCheapestContractor().cost, pricePerUnit = self.
       findCheapestContractor().cost, cheapestSupplier = self.
       findCheapestContractor().contact.name}storage.restock.command>
16 tc12_restock.command>
17 tc12_restock.command>    !openter (self.findProductsAmountLesser(
       aSearchAmount)->asSequence->first()) setAmount(aNewAmount)
18 tc12_restock.command>            open stored.setAmount.command
19 stored.setAmount.command> -- stored.setAmount(Real)
20 stored.setAmount.command>
21 stored.setAmount.command>         !set self.amount := newAmount
22 stored.setAmount.command>         -- We're updating the Stored-date
       first by increasing it by one. If that new date is bigger than the
       one available in the product attributes, the latter will be updated
       . Finally, the storage's date will be set to the Stored one if the
       former is lesser than the latter.
23
24 stored.setAmount.command>         !openter self update()
25 stored.setAmount.command>                 open stored.update.command
26 stored.update.command>
27 stored.update.command>  !set self.date := self.date + 1
28 stored.update.command>  !openter self.product update(self.date)
```

```
stored.update.command>          !set self.date := if self.date >=
    aDate then self.date else aDate endif
stored.update.command>  !opexit self.date
stored.update.command>  !openter self.storage update(self.date)
stored.update.command>          !set self.date := if self.date >=
    aDate then self.date else aDate endif
stored.update.command>  !opexit self.date
stored.update.command>
stored.setAmount.command>       !opexit self.update()-1
stored.setAmount.command>
tc12_restock.command>   !opexit
tc12_restock.command>
tc12_restock.command>   !openter (self.findProductsAmountLesser(
    aSearchAmount)->asSequence->first()) setAmount(aNewAmount)
tc12_restock.command>           open stored.setAmount.command
stored.setAmount.command> -- stored.setAmount(Real)
stored.setAmount.command>
stored.setAmount.command>       !set self.amount := newAmount
stored.setAmount.command>       -- We're updating the Stored-date
    first by increasing it by one. If that new date is bigger than the
    one available in the product attributes, the latter will be updated
    . Finally, the storage's date will be set to the Stored one if the
    former is lesser than the latter.

stored.setAmount.command>       !openter self update()
stored.setAmount.command>               open stored.update.command
stored.update.command>
stored.update.command>  !set self.date := self.date + 1
stored.update.command>  !openter self.product update(self.date)
stored.update.command>          !set self.date := if self.date >=
    aDate then self.date else aDate endif
stored.update.command>  !opexit self.date
stored.update.command>  !openter self.storage update(self.date)
stored.update.command>          !set self.date := if self.date >=
    aDate then self.date else aDate endif
stored.update.command>  !opexit self.date
stored.update.command>
stored.setAmount.command>       !opexit self.update()-1
stored.setAmount.command>
tc12_restock.command>   !opexit
tc12_restock.command>
tc12_restock.command>   !openter (self.findProductsAmountLesser(
    aSearchAmount)->asSequence->first()) setAmount(aNewAmount)
tc12_restock.command>           open stored.setAmount.command
stored.setAmount.command> -- stored.setAmount(Real)
stored.setAmount.command>
stored.setAmount.command>       !set self.amount := newAmount
```

```
66 stored.setAmount.command>        -- We're updating the Stored-date
       first by increasing it by one. If that new date is bigger than the
       one available in the product attributes, the latter will be updated
       . Finally, the storage's date will be set to the Stored one if the
       former is lesser than the latter.
67
68 stored.setAmount.command>        !openter self update()
69 stored.setAmount.command>                open stored.update.command
70 stored.update.command>
71 stored.update.command>  !set self.date := self.date + 1
72 stored.update.command>  !openter self.product update(self.date)
73 stored.update.command>          !set self.date := if self.date >=
       aDate then self.date else aDate endif
74 stored.update.command>  !opexit self.date
75 stored.update.command>  !openter self.storage update(self.date)
76 stored.update.command>          !set self.date := if self.date >=
       aDate then self.date else aDate endif
77 stored.update.command>  !opexit self.date
78 stored.update.command>
79 stored.setAmount.command>        !opexit self.update()-1
80 stored.setAmount.command>
81 tc12_restock.command>    !opexit
82 tc12_restock.command>
83 tc12_restock.command>
84 tc12_restock.command> !opexit quittung01.content
85 postcondition `amountRestocked' is true
86 tc12_restock.command>
```

Sequenzdiagramm

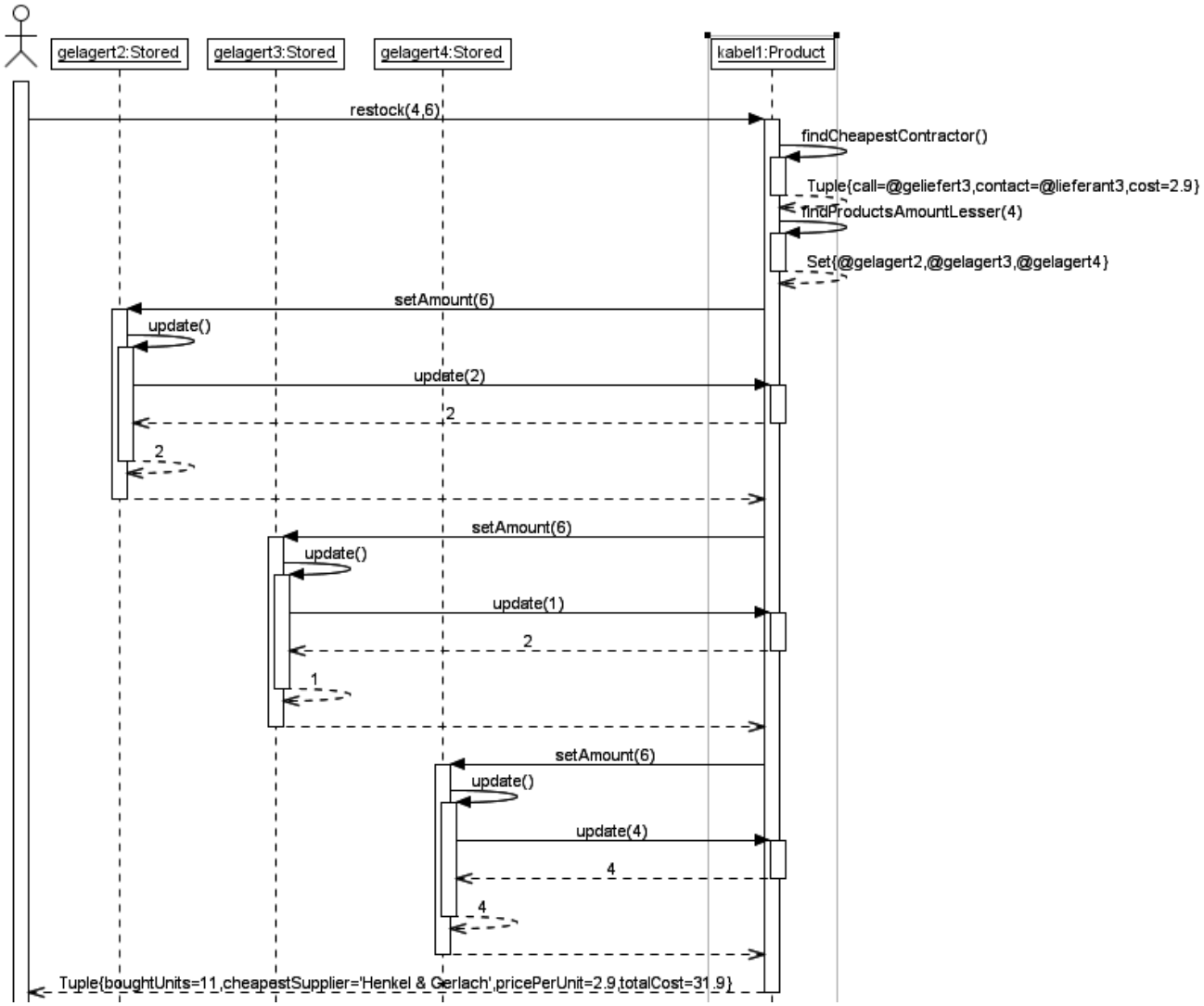

Abbildung 4.15: Das Sequenzdiagramm zu TC12 zeigt, wie zunächst die erforderlichen Daten für die Rechnung erhoben werden und dann die Lagerverbindungen nacheinander bearbeitet werden.)

4.3.2 TC13

Wir bauen auf dem Zustand von TC12 auf und füllen alle Lagerverbindungen die kleiner/-gleich 6 waren auf 7. Dazu setzen wir noch den Lieferpreis für „lieferanten1“ auf 1. Nun ist das der günstigste Anbieter. Das Ergebnis ist recht einfach nachzuvollziehen, deshalb

nur das Hilfsobjekt als Ausgabe: [1]

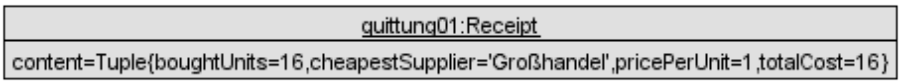

Abbildung 4.16: Dieser Ausschnitt zeigt ein Objekt der Hilfsklasse *Receipt*. Es zeigt das aktuelle, zufriedenstellende Ergebnis.

4.3.3 TC14

In diesem Fall wiederholen wir TC11, versäumen es aber ungeschickterweise die Lagerverbindungen tatsächlich aufzufüllen. *post amountRestocked?*[2] schlägt fehl. [3]

Die Kommandozeilenausgabe

(Zeilenumbrüche leicht umformatiert)

```
1  tc14_restock.command> !openter kabel1 restock(4,6)
2  precondition `parametersAreDefined' is true
3  precondition `amountsArePositive' is true
4  precondition `amountsAreBalanced' is true
5  tc14_restock.command>
6  tc14_restock.command>    open storage.restock.command
7  storage.restock.command> !openter self findCheapestContractor()
8  storage.restock.command>         !opexit self.findCheapestContractor()
9  postcondition `foundCheapest' is true
10 storage.restock.command>
11 storage.restock.command>         !openter self findProductsAmountLesser
       (aSearchAmount)
12 storage.restock.command>         !opexit self.findProductsAmountLesser(
       anAmount)
13 storage.restock.command>
14 storage.restock.command>         !create quittung01 : Receipt
15 storage.restock.command>         !set quittung01.content := Tuple{
       boughtUnits = (self.findProductsAmountLesser(aSearchAmount).amount
       ->size()*aNewAmount) - self.findProductsAmountLesser(aSearchAmount)
       .amount->sum(), totalCost = ((self.findProductsAmountLesser(
       aSearchAmount).amount->size()*aNewAmount)- self.
       findProductsAmountLesser(aSearchAmount).amount->sum())*self.
       findCheapestContractor().cost, pricePerUnit = self.
       findCheapestContractor().cost, cheapestSupplier = self.
       findCheapestContractor().contact.name}
```

1 Siehe auch im Dateiverzeichnis Seite 89
2 Seite 21
3 Siehe auch im Dateiverzeichnis Seite 89

```
16 storage.restock.command>
17 tc14_restock.command>
18 tc14_restock.command>   -- We're not going to actually update the
       stock. We just bill for nothing.
19 tc14_restock.command>   --!openter (self.findProductsAmountLesser(
       aSearchAmount)->asSequence->first()) setAmount(aNewAmount)
20 tc14_restock.command>   --       open stored.setAmount.command
21 tc14_restock.command>   --!opexit
22 tc14_restock.command>
23 tc14_restock.command>   --!openter (self.findProductsAmountLesser(
       aSearchAmount)->asSequence->first()) setAmount(aNewAmount)
24 tc14_restock.command>   --       open stored.setAmount.command
25 tc14_restock.command>   --!opexit
26 tc14_restock.command>
27 tc14_restock.command>   --!openter (self.findProductsAmountLesser(
       aSearchAmount)->asSequence->first()) setAmount(aNewAmount)
28 tc14_restock.command>   --       open stored.setAmount.command
29 tc14_restock.command>   --!opexit
30 tc14_restock.command>
31 tc14_restock.command>
32 tc14_restock.command> !opexit quittung01.content
33 postcondition `amountRestocked' is false
34 evaluation results:
35   self : Product = @kabel1
36   self.stored : Set(Stored) = Set{@gelagert1,@gelagert2,@gelagert3,
         @gelagert4}
37   s : Stored = @gelagert1
38   s.amount@pre : Integer = 5
39   s : Stored = @gelagert2
40   s.amount@pre : Integer = 4
41   s : Stored = @gelagert3
42   s.amount@pre : Integer = 1
43   s : Stored = @gelagert4
44   s.amount@pre : Integer = 2
45   self.stored->collect(s : Stored | s.amount@pre) : Bag(Integer) = Bag
         {1,2,4,5}
46   self.stored->collect(s : Stored | s.amount@pre)->sum : Integer = 12
47   result : Tuple(boughtUnits:Real,cheapestSupplier:String,pricePerUnit
         :Real,totalCost:Real) = Tuple{boughtUnits=11,cheapestSupplier='
         Henkel & Gerlach',pricePerUnit=2.9,totalCost=31.9}
48   result.boughtUnits : Integer = 11
49   (self.stored->collect(s : Stored | s.amount@pre)->sum + result.
         boughtUnits) : Integer = 23
50   self : Product = @kabel1
51   self.stored : Set(Stored) = Set{@gelagert1,@gelagert2,@gelagert3,
         @gelagert4}
52   $e : Stored = @gelagert1
```

```
53  $e.amount : Integer = 5
54  $e : Stored = @gelagert2
55  $e.amount : Integer = 4
56  $e : Stored = @gelagert3
57  $e.amount : Integer = 1
58  $e : Stored = @gelagert4
59  $e.amount : Integer = 2
60  self.stored->collectNested($e : Stored | $e.amount) : Bag(Integer) =
        Bag{1,2,4,5}
61  self.stored->collectNested($e : Stored | $e.amount)->sum : Integer =
        12
62  ((self.stored->collect(s : Stored | s.amount@pre)->sum + result.
      boughtUnits) = self.stored->collectNested($e : Stored | $e.amount
      )->sum) : Boolean = false
63 tc14_restock.command>
```

Sequenzdiagramm

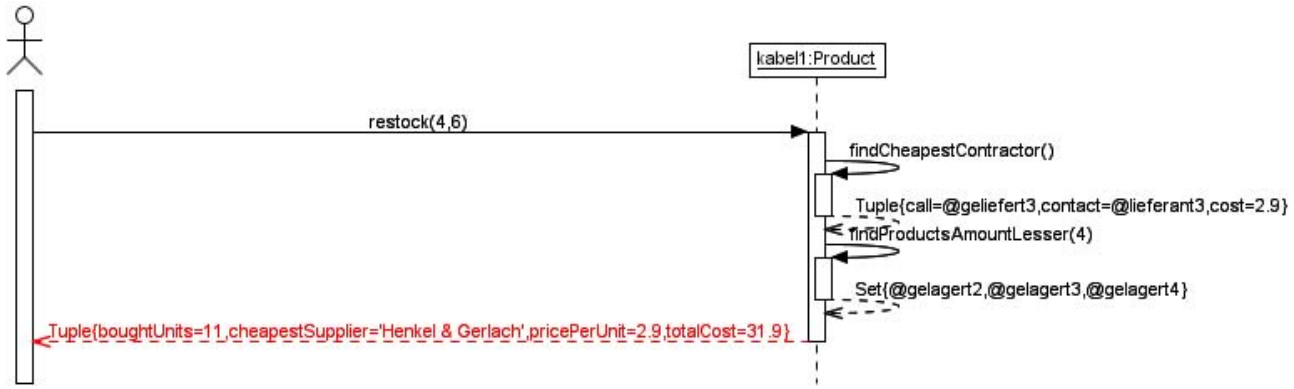

Abbildung 4.17: Das Sequenzdiagramm zu TC14 zeigt, dass die Daten für die auszugebende Rechnung korrekt gesammelt weden, doch die einzige Postkondition *post amountRestocked?* (Seite 21) fehl schlägt. Es wurden keine Stückzahlen aufgefüllt.

5 Verzeichnisse

Abkürzungen

UML Unified Modeling Language

OCL Object Constraint Language

USE UML-based Specification Environment

Dateiverzeichis

1. **storage.bad1.state** ist der auf Seite 26 aufgeführte Zustand.
2. **storage.bad2.state** ist der auf Seite 28 aufgeführte Zustand.
3. **storage.bad3.state** ist der auf Seite 30 aufgeführte Zustand.
4. **storage.bad4.state** ist der auf Seite 33 aufgeführte Zustand.
5. **storage.bad5.state** ist der auf Seite 36 aufgeführte Zustand.
6. **storage.bad6.state** ist der auf Seite 39 aufgeführte Zustand.
7. **storage.deleteStored.command** ist die auf Seite 9 und 17 besprochene Operation. Die dazugehörige Testserie findet sich ab Seite 42.
8. **storage.deleteStored.state** ist der auf Seite 43 aufgeführte Zustand.
9. **storage.good1.state** ist der auf Seite 22 aufgeführte Zustand.
10. **storage.good2.state** ist in dieser Hausarbeit nicht weiter verarbeitet worden.
11. **storage.moveAmount.command** ist die auf Seite 9 und 18 besprochene Operation. Die dazugehörige Testserie findet sich ab Seite 57.
12. **storage.moveAmount.state** ist der auf Seite 59 aufgeführte Zustand.
13. **storage.product.setAmount.command** ist die auf Seite 8 besprochene Operation.
14. **storage.restock.command** ist die auf Seite 7 und 21 besprochene Operation. Die dazugehörige Testserie findet sich ab Seite 77.
15. **storage.restock.state** ist der auf Seite 78 aufgeführte Zustand.
16. **Storage.use** ist die ab Seite 3 vorgestellte Hauptdatei.
17. **storage.setAmount.command** ist die auf Seite 11 besprochene Operation.
18. **storage.update.command** ist die auf Seite 11 besprochene Operation.
19. **tc01_deleteStored.command** ist der Test auf Seite 43.

Abbildungsverzeichnis

Danksagung

Vielen Dank geht an Matthias Pospiech, der die für diese Hausarbeit erforderliche LaTeX-Grundlage online verfügbar gemacht hat.[1]

1 Zu finden unter: `http://www.matthiaspospiech.de/latex/vorlagen/allgemein/`