Universität Bremen
Fachbereich 3: Mathematik/Informatik

Ausarbeitung zum Thema

Schulungsverwaltung

Im Rahmen der Lehrveranstaltung

Entwurf von Informationssystemen

Anika Bischoffs (anika@tzi.de)
Frank Hilken (fhilken@tzi.de)
Sebastian Langer (sevox@tzi.de)
Danny Zitzmann (danny@tzi.de)

Sommersemester 2012

bei

Prof. Dr. Martin Gogolla
M.Sc. Lars Hamann

Eingereicht am:
31.08.2012

Inhaltsverzeichnis

1. Einleitung

Diese Ausarbeitung wurde im Rahmen der Lehrveranstaltung “Entwurf von Informationssystemen“ im Sommersemester 2012 angefertigt. Das Ziel ist die Entwicklung eines Anwendungssystems, in diesem konkreten Fall eine Schulungsverwaltung, mit Hilfe des UML-Formalismus. Die Modellierung erfolgt in USE (UML Based Specification Environment)[Bre07], einem UML (Unified Modeling Language)[OMG11] und OCL (Object Constraint Language)[OMG12] Werkzeug, welches von der Arbeitsgruppe Datenbanksysteme der Universität Bremen entwickelt und bereitgestellt wurde.

Die Struktur des entworfenen Systems wird hierbei durch UML Klassen und ihre Beziehungen untereinander modelliert. Die jeweiligen Klassen werden durch die Definition von Attributen und Methoden gemäß des UML Standards ausgeprägt. Zur Prüfung der Korrektheit der Struktur und des Systementwurfs kommt OCL zum Einsatz, welches die Definition und Prüfung von Invarianten, die in bestimmten Systemzuständen gelten müssen, sowie die Bildung von Vor- und Nachbedingungen für die entwickelten Operationen, erlaubt.

Nach einer allgemeinen Systembeschreibung zu Beginn wird zunächst das Modell näher erläutert, d.h. alle Klasse, Attribute, Assoziationen, Operationen und Invarianten der definierten Modellelemente werden im Detail beschrieben. Anschließend folgt die Definition und Durchführung von Testfällen. Hierbei wird die Strategie verfolgt ein global gültiges Objektdiagramm zu erzeugen, welches ausreichend umfangreich ist und von jedem Test eingebunden werden kann. Die einzelnen Tests werden nach dem Einbinden des großen Modells gezielt bestimmte Eigenschaften verletzen und prüfen, ob diese hinreichend durch die definierten Invarianten abgesichert wurden.

Abgerundet wird dieses Dokument durch einige ausgewählte Sequenzdiagramme, die das Verhalten bestimmter Systemkomponenten besser veranschaulichen. Zudem werden noch einige Anfragen an das System formuliert, die zeigen sollen, dass die Logik, die sich hinter dem modellierten System verbirgt, korrekt ist und das System den Anforderungen, die in seinem Einsatzbereich gelten, gerecht werden kann.

Der Entwurf und die Durchführung der Tests des Systems wurden mit der USE Version 3.0.3 vorgenommen. Alle Beispiele, Beschreibungen und Listings beziehen ihre Gültigkeit auf die genannte Version und kamen unter dieser erfolgreich zum Einsatz. Auszüge aus einzelnen Listings werden nur an Stellen gezeigt, wo dieses als besonders sinnvoll erachtet wurde. Das vollständige Listing des Systems befindet sich im Anhang dieses Dokuments und dient als Basis für die hier vorliegende Beschreibung der einzelnen Komponenten.

2. Systembeschreibung

Für die Bearbeitung der Semesterarbeit wurde das zweite von den Veranstaltern vorgeschlagene System gewählt. Es gilt eine Verwaltungssoftware für ein Unternehemen zu entwerfen, welches diverse Schulungen für bestimmte Software- und Hardwareprodukte anbietet. Im Folgenden werden die einzelnen Elemente dieses Systems sowie deren Eigenschaften beschrieben.

Im Mittelpunkt des Systems stehen die angebotenen Schulungen des Unternehemens. Diese besitzen eine intern eindeutige ID, sodass es keine zwei Schulungen mit derselben Kennung geben kann. Jede angebotene Schulung besitzt zudem einen Titel sowie eine Beschreibung, die Aufschluss über die Inhalte der angebotenen Veranstaltung gibt. Um das Kursangebot leichter durchsuchen und kategorisieren zu können, wird es eine Möglichkeit geben die Kurse anhand von Kategorien zu katalogisieren.

Um eine Schulung für die Teilnehmer anzubieten wird für diese eine Session erstellt, für die sich Teilnehmer anmelden können. Um mehr Teilnehmer zeitgleich zu bedienen, können mehrere Sessions erstellt werden.

Sollte ein Kurs genügend hohe Vorannmeldungen von potentiellen Teilnehmern haben, soll es möglich sein einen Termin für dessen Durchführung anzusetzen und die Schulung durchzuführen. Hierfür wird neben der eigentlichen Schulungsverwaltung, die der Kern des Systems ist, auch die Möglichkeit geboten Räume, Dozenten, Catering, Rechnungen und Dokumente anzulegen und zu verwalten, die für eine bestimmte Schulung benötigt werden.

Gibt es nicht genug Anmeldungen für eine Session, können zeitgleich stattfindende Sessions zusammengelegt oder einzelne Sessions abgesagt werden.

Für die einzelnen Schulungsangebote müssen jeweils ein oder mehrere Dozenten, die die Qualifikation besitzen die jeweilige Schulung durchführen zu können, im System vorhanden sein, andernfalls kann keine Schulung zu diesem speziellen Thema angeboten werden. Da es sich bei den angebotenen Schulungen um Einführungen in bestimmte Software- und Hardwareprodukte handeln soll, bietet das System auch die Möglichkeit diese zu verwalten und in Beziehung mit bestimmten Schulungen zu setzen.

Jeder Kurs kann zudem Kursteilnehmer besitzen. Diese registrieren sich zunächst für die angebotene Schulung. Die Anmeldung kann dann akzeptiert oder abgelehnt werden. Sobald ein Teilnehmer für den Kurs angenommen wurde und die Termine und

Räume fest stehen, kann eine Rechnung erstellt werden. Neben der Möglichkeit private Teilnehmer aufzunehmen und zu schulen, wird auch der Fall abgedeckt, dass die Teilnehmer Mitarbeiter einer Firma sind und von dieser auf das Fortbildungsseminar geschickt wurden. Das System bietet daher die Möglichkeit einzelne Teilnehmer einer Firma zuzuordnen.

Wird eine Rechnung nicht bezahlt, kann der Teilnehmer für einen Kurs auch wieder abgelehnt werden. über den Teilnehmer selbst werden alle nötigen Informationen gespeichert. Nach Abschluss des Kurses und dem Bezahlen der Rechnung erhält der Teilnehmer ein Zertifikat.

3. Modell

Das Klassendiagramm in Abbildung 3.1 auf der nächsten Seite zeigt die Struktur der entwickelten Schulungsverwaltung.

Den Kern des Systems bildet die Klasse `Training`, die als Vorlage für Schulungen dient. Um daraus ein Schulungsangebot zu erstellen gibt es die Klasse `TrainingSession`. Im System gibt es zu den Tools jeweils Trainings, welche dann als Schulungen angeboten werden. Die einzelnen, konkreten Schulungen werden dann als `TrainingSession` repräsentiert. Jedes `Training` kann einer Kategorie, dargestellt durch `Category`, zugeordnet werden, welche selbst wieder Teil einer Kategorie sein kann, so lässt sich eine beliebige Hierarchy von Kategorien und Unterkategorien aufbauen. Jedes `Training` kann mit Tools und Dokumenten versehen sein, die für das Training benötigt werden. Die Tools zerfallen dabei in Soft- und Hardware.

Der `Instructor`, der an einem angesetzten `Appointment` die Schulung durchführen soll, muss über eine hinreichende Qualifikation für die zu lehrenden Tools verfügen. Jede konkrete `TrainingSession` verfügt über `Appointments`, welche jeweils mit einem `Room` und `Catering` verbunden sein können. Ein `Instructor` steht über die Assoziationsklasse `Teaches` mit einem `Appointment` in Verbindung. Die Teilnehmer (`Participant`) stehen über die Assoziationsklasse `Registration` direkt mit der `TrainingSession` in Verbindung. Die Assoziationsklasse `Registration` steht mit der Klasse `Bill` in Beziehung, um Rechnungen für die Teilnehmer zu speichern.

Für die Personen ist das *Role-Model* implementiert worden. Die Klassen `Participant` und `Instructor` ergeben sich jeweils aus der Klasse `Person`, in der die Basisdaten einer jeden Person im System erfasst werden. `Participant` kann zudem noch eine Beziehung zu einer `Company` haben, für der er oder sie arbeitet.

Neben den genannten Systemklassen wurden auch die beiden Klassen `Date` und `Period` aufgenommen, die aus einer gemeinsamen Entwicklung aller Gruppen und auf Basis von [All83] im Rahmen der Lehrveranstaltung hervorgegangen sind.

Auf den nachfolgenden Seiten werden die einzelnen Enumerations und Klassen des Modells mit ihren Attributen, Operationen und Invarianten im Detail beschrieben.

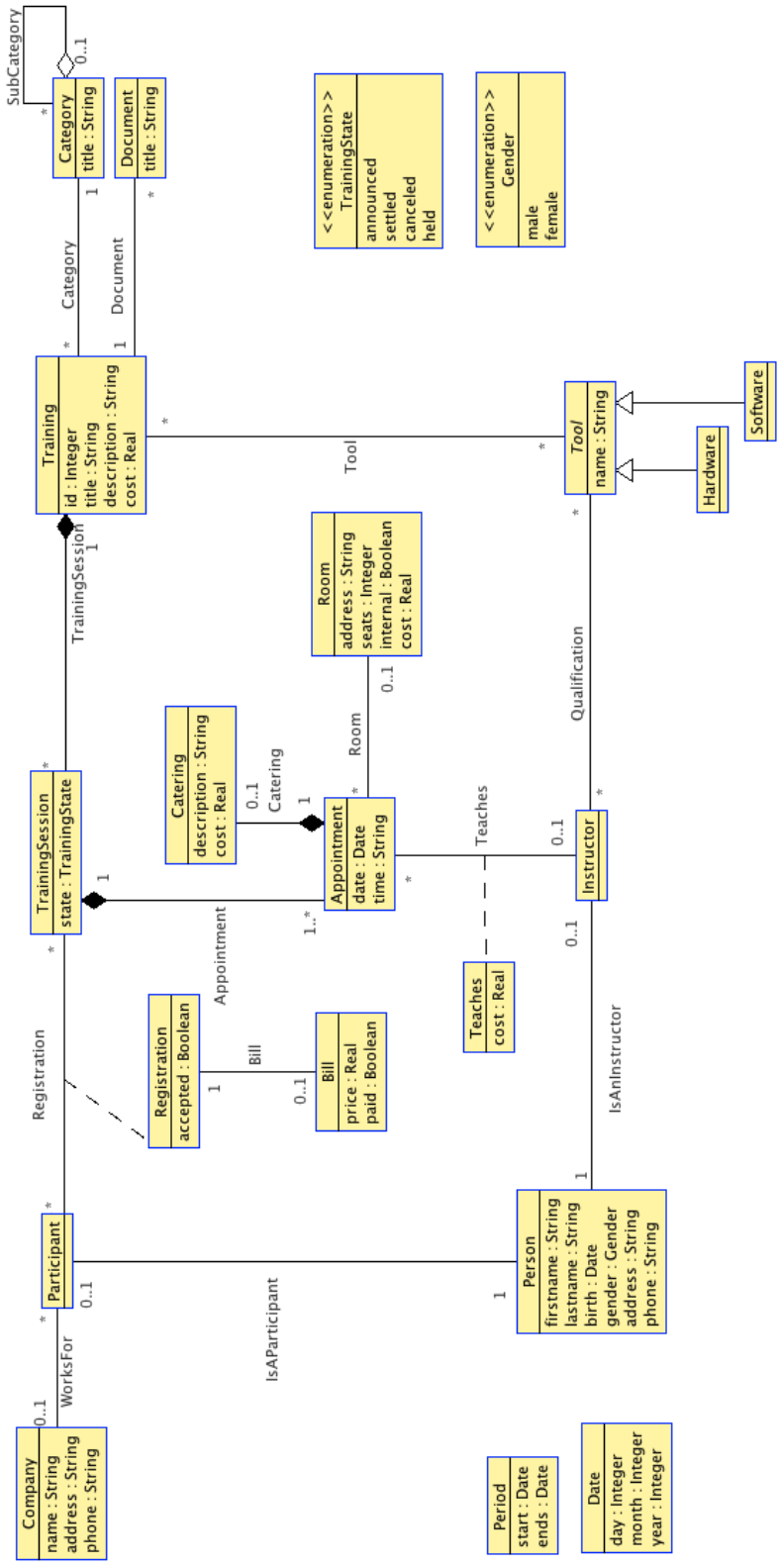

Abbildung 3.1.: Klassendiagramm der Schulungsverwaltung

3.1. Enumerations

Während der Realisierung des Klassendiagramm wurden zwei Enumerations (`Gender`, `TrainingState`) definiert, die in den nachfolgenden Abschnitten im Detail beschrieben werden und in verschiedenen Klassen des erstellten Systems zum Einsatz kommen.

3.1.1. Gender

Die Enumeration `Gender` wird ausschließlich von der Oberklasse `Person` genutzt von der sich die Klassen `Participant` und `Instructor` ableiten lassen. `Gender` wurde eingeführt, um allen im System vertretenen Personen ein eindeutiges Geschlecht zuweisen zu können. Die Enumeration besteht aus den folgenden zwei Feldern:

- `male` - Kann genutzt werden, um eine Person als männlich zu definieren.
- `female` - Kann genutzt werden, um eine Person als weiblich zu definieren.

3.1.2. TrainingState

Die Enumeration `TrainingState` wird ausschließlich von `TrainingSession` genutzt, nicht jedoch von der Oberklasse `Training`, da nur konkrete Trainingsinstanzen einen Status haben können. Traingsvorlagen haben keinen Status. `TrainingState` wurde eingeführt, um allen im System vertretenen Trainingsinstanzen einen eindeutigen Zustand zuweisen zu können. Die Enumeration besteht aus den folgenden vier Feldern:

- `announced` - Kann genutzt werden, um auszudrücken, dass eine Schulung (Instanz von `TrainingSession`) sich im Status *angekündigt* befindet.
- `settled` - Kann genutzt werden, um zu beschreiben, dass eine Schulung zu einem bestimmten Termin festgesetzt wurde und damit über den Status `announced` hinaus ist.
- `canceled` - Wenn eine Schulung abgesagt wurde, kann dieser Wert genutzt werden, um sie als abgebrochen zu kennzeichnen.
- `held` - Wurde eine Schulung wie geplant durchgeführt und ist nun abgeschlossen, sollte ihr der Wert `held` zugewiesen werden.

3.2. Klassen und Attribute

3.2.1. Person

Klassenbeschreibung:

Die Klasse `Person` dient zur Erfassung und Verwaltung von Personen im Schulungssystem. Sie enthält dabei alle benötigten personenspezifischen Daten und dient als Ausgangspunkt zur Erzeugen von Klasseninstanzen des Typs `Instructor` und `Participant`, die auf die hier hinterlegten Attribute verweisen können.

Attribute:

firstname	Speichert den Vornamen einer Person im System. Der Datentyp dieses Klassenattributs ist ein `String`.
lastname	Erfasst den Nachnamen einer im System registrierten Person. Der Datentyp dieses Klassenattributs ist ebenfalls ein `String`.
birth	Repräsentiert das Geburtsdatum der erfassten Person. Hierbei handelt es sich um eine Klasseninstanz vom Typ `Date`.
gender	Dient zum Erfassen des Geschlechts einer Person. Hierbei handelt es sich um eine definierte Enumeration `Gender`.
address	Ermöglicht das Hinterlegen der Anschrift einer Person im System. Der Datentyp dieses Klassenattributs ist ein `String`.
phone	Ermöglicht das Hinterlegen der Telefonnummer einer Person im System. Der Datentyp dieses Klassenattributs ist ebenfalls ein `String`.

3.2.2. Participant

Klassenbeschreibung:

Die Klasse `Participant` dient zur Repräsentation von Schulungsteilnehmer im System. Sie selbst verfügt über keinerlei Attribute, steht aber in Beziehung zur Klasse `Person`, in der alle Basisattribute einer Person gespeichert werden können. Instanzen vom Typ `Participant` werden im System über eine Operation der Klasse `Person` erzeugt. Siehe dazu die Operationsbeschreibung unter 3.4.2 auf Seite 26.

Attribute:

Diese Klasse verfügt über keine Attribute.

3.2.3. Instructor

Klassenbeschreibung:

Die Klasse `Instructor` dient zur Repräsentation von Dozenten im System, die befähigt sind Schulungen durchzuführen. Sie selbst verfügt über keinerlei Attribute, steht aber wie die Klasse `Participant` in Beziehung zur Klasse `Person`, in der alle Basisattribute einer Person gespeichert werden können. Instanzen vom Typ `Instructor` werden im System über eine Operation der Klasse `Person` erzeugt. Siehe dazu die Operationsbeschreibung unter 3.4.3 auf Seite 26.

Attribute:

Diese Klasse verfügt über keine Attribute.

3.2.4. Company

Klassenbeschreibung:

`Company` ist eine Klasse im entwickelten System, die dazu dient Firmen zu erfassen. Dies wird benötigt, wenn man Schulungsteilnehmer einer Firma zuordnen möchte, im Falle dessen, dass sie von dieser Firma zu einem Fortbildungsseminar geschickt worden sind. Die dann relevanten Kontaktdaten sind nicht die Anschriften der einzelnen Personen, sondern die der Firmen, für die diese Personen arbeiten.

Attribute:

name	Dient zur Erfassung des Namens einer Firma. Bei dem Datentyp dieses Attributs handelt es sich um einen `String`.
address	Repräsentiert die Anschrift der erfassten Firma. Auch dieses Attribut ist ein `String`.
phone	Erfasst die Telefonummer der registrierten Firma. Bei dem Datentyp dieses Klassenattributs handelt es sich ebenfalls um einen `String`.

3.2.5. Category

Klassenbeschreibung:

Die Klasse `Category` wurde eingeführt, um Schulungen im System bestimmten Kategorien, die selber wieder Teil einer Oberkategorie sein können, zuzuweisen. Das Katalogisieren und Suchen nach Schulungen bestimmter Themenbereiche soll so erleichtert werden.

Kategorien werden in Bäumen verwaltet und haben eine übergeordnete und beliebig viele untergordnete Kategorien. Dabei können mehrere Kategorienbäume gleichzeitig existieren. Jede Kategorie, die keine übergeordnete Kategorie besitzt, eine Wurzelkategorie. Diese Definition ist relevant für den Katalog in Abschnitt 6.9 auf Seite 95.

Attribute:

title Dieses Klassenattribut dient zur Speicherung des Kategorienamens. Bei dem Datentyp dieses Attributs handelt es sich um einen `String`.

3.2.6. Training

Klassenbeschreibung:

Die Klasse `Training` dient zur Repräsentation eines Schulungsthemas. Diese Klasse dient als Basis zum Erzeugen von `TrainingSession` Instanzen. Die Idee ist, dass jedes Schulungsthema in einem `Training` erfasst wird und sich von einem solchen vordefinierten `Training` beliebig viele Instanzen der Klasse `TrainingSession`, die die konkret angesetzten Schulungen repräsentieren, erzeugen lassen.

Attribute:

id Eine eindeutige ID unter der sich das `Training` im System finden lassen kann. Es darf niemals zwei oder mehr Trainings geben, die über dieselbe ID verfügen. Der Datentyp dieses Attributs ist ein `Integer`.

title Dient zum Speichern des Titels einer Schulung. Der Datentyp dieses Klassenattributs ist ein `String`.

description Enthält eine detaillierte Beschreibung der angebotenen Schulung. Dieses Klassenattribut ist ebenfalls vom Typ `String`.

cost Gibt die Kosten an, die für einen Schulungsteilnehmer entstehen würden, wenn er an dieser Schulung teilnehmen würde. Der Datentyp dieses Klassenattributs ist ein `Real`.

3.2.7. TrainingSession

Klassenbeschreibung:

Die Klasse `TrainingSession` dient der Repräsentation einer konkret angesetzten und durchzuführenden Schulung im System. Die `TrainingSession` kann sich dabei in den Zuständen angekündigt, festgesetzt, abgesagt oder durchgeführt befinden. Siehe dazu auch 3.1.2 auf Seite 13.

Attribute:

state Dieses Klassenattribut gibt den Zustand an, in dem sich eine angesetzte Schulung befindet. Hierbei handelt es sich um die Enumeration `TrainingState`.

3.2.8. Appointment

Klassenbeschreibung:

Die Klasse `Appointment` steht in einer direkten Beziehung zu einer Instanz der Klasse `TrainingState` und dient zum Erfassen des Datums und der Uhrzeit, an dem eine Schulung durchgeführt werden soll.

Attribute:

date Gibt das Datum an, für das der Schulungstermin angesetzt worden ist. Der Typ dieses Klassenattributs ist eine Instanz der Klasse `Date`.

time Speichert die konkrete Uhrzeit, zu der die Schulung starten soll. Beim Typ dieses Attributs handelt es sich um einen `String`.

3.2.9. Room

Klassenbeschreibung:

Die Klasse `Room` dient zur Raumverwaltung. Jeder Raum wird im System durch eine Instanz dieser Klasse repräsentiert. Erfasst werden alle benötigten Informationen, wie etwa die Adresse, Sitzplätze und die Kosten, die bei einer Mietung des Raums entstehen. Die Klasse `Room` steht in Beziehung zu einer Instanz der Klasse `Appointment` und beschreibt somit, in welchem Raum eine angesetzte Schulung durchgeführt wird.

Attribute:

address Gibt die Adresse an, unter der sich der Raum finden lässt. Beim Datentyp dieses Klassenattributs handelt es sich um einen `String`.

seats Dieses Attribut enthält die maximale Anzahl an Sitzplätzen in einem Raum. Das Klassenattribut ist vom Typ `Integer`.

internal Gibt an, ob es sich bei dem Raum um einen Raum handelt, der der Schulungsfirma gehört, oder um einen externen Raum, der extra angemietet werden muss. Der Datentyp dieses Klassenattributs ist ein `Boolean`.

cost Sollte es sich nicht um einen internen Raum handeln, können hier die Mietkosten für den Raum erfasst werden. Das Klassenattribut ist vom Typ `Real`.

3.2.10. Catering

Klassenbeschreibung:

Zu jedem guten Schulungsseminar gehört auch die Bewirtung der Schulungsteilnehmer. Die Klasse `Catering` repräsentiert eine solche Verpflegung der Schulungsteilnehmer. Sie steht in direkter Beziehung zu einer Instanz der Klasse `Appointment`.

Attribute:

description Dieses Klassenattribut dient zur Erfassung einer detaillierten Beschreibung des `Caterings`. Der Datentyp ist ein `String`.

cost Gibt die Kosten an, die durch das `Catering` entstehen. Der Datentyp dieses Klassenattributs ist ein `Real`.

3.2.11. Tool

Klassenbeschreibung:

Die Klasse `Tool` stellt eine abstrakte Oberklasse dar, von der die beiden Klassen `Software` und `Hardware` erben. Alle Attribute dieser Klasse stehen den erbenden Klassen zur Verfügung. `Tool` wird benötigt, um Tools, die für ein Schulungsseminar erforderlich sind, im System erfassen zu können.

Attribute:

name Dieses Klassenattribut dient zur Erfassung des jeweiligen Toolnamens. Es ist vom Datentyp `String`.

3.2.12. Software

Klassenbeschreibung:

Bei der Klasse `Software` handelt es sich um eine Kindklasse von `Tool`. Sie verfügt über keine eigenen Attribute, sondern greift lediglich auf die Attribute ihrer Elternklasse zu. Die Klasse `Software` dient zur Repräsentation von Softwareprodukten im System, die für bestimmte Seminare benötigt werden.

Attribute:

Diese Klasse verfügt über keine Attribute, kann aber auf das Attribute der Oberklasse `Tool` zugreifen.

3.2.13. Hardware

Klassenbeschreibung:

Bei der Klasse `Hardware` handelt es sich um eine Kindklasse von `Tool`. Sie verfügt über keine eigenen Attribute, sondern greift lediglich auf die Attribute ihrer Elternklasse zu. Die Klasse `Hardware` dient zur Repräsentation von Hardwareprodukten im System, die für bestimmte Seminare benötigt werden.

Attribute:

Diese Klasse verfügt über keine Attribute, kann aber auf das Attribute der Oberklasse `Tool` zugreifen.

3.2.14. Document

Klassenbeschreibung:

Die Klasse `Document` steht in direkter Beziehung zur Klasse `Training` und dient zur Erfassung von Dokumenten, die im Rahmen eines Schulungsseminars benötigt und gegebenenfalls an die Teilnehmer ausgehändigt werden. Jede Instanz dieser Klasse repräsentiert genau ein Dokument.

Attribute:

title Dieses Klassenattribut dient zur Erfassung des Dokumentnamens. Es ist vom Typ `String`.

3.2.15. Registration

Klassenbeschreibung:

Bei der Klasse `Registration` handelt es sich um eine Assoziationsklasse, zwischen den Klassen `Participant` und `TrainingSession`. Sie gibt an, dass ein Teilnehmer für eine Schulung registriert ist und verwaltet, ob dieser auch für das Seminar akzeptiert wurde.

Attribute:

accepted Dieses Klassenattribut erfasst, ob der jeweilige Teilnehmer bereits für das anstehende Schulungsseminar akzeptiert wurde oder nicht. Der Datentyp dieses Klassenattributs ist ein `Boolean`.

3.2.16. Bill

Klassenbeschreibung:

Die Klasse `Bill` dient zur Repräsentation von Rechnungen im System und steht in direkter Verbindung mit der Assoziationsklasse `Registration`. Rechnungen werden üblicherweise den Schulungsteilnehmern ausgestellt, deren Status auf `accepted` wechselt, wenn die Rechnung bezahlt wurde.

Attribute:

price In diesem Klassenattribut werden die in Rechnung gestellten Kosten erfasst. Der Datentyp ist ein `Real`.

paid Hierbei handelt es sich um ein boolsches Flag, welches auf wahr gesetzt wird, wenn der Teilnehmer seine Rechnung bezahlt hat.

3.2.17. Teaches

Klassenbeschreibung:

Bei der Klasse `Teaches` handelt es sich um eine Assoziationsklasse, die bei einem Link zwischen den beiden Klassen `Instructor` und `Appointment` zum Einsatz kommt. Sie gibt an, dass ein Dozent für einen Schulungstermin zum Einsatz kommen soll und verwaltet dabei die entstehenden Kosten, also das Gehalt, das der Dozent für die Durchführung eines bestimmten Schulungsseminars bekommt.

Attribute:

cost Gibt die Kosten an, die durch den Einsatz des jeweiligen Dozenten entstehen. Der Datentyp dieses Klassenattributs ist ein `Real`.

3.2.18. Date

Klassenbeschreibung:

Bei der Klasse `Date` handelt es sich um eine gemeinsam im Rahmen der Lehrveranstaltung entwickelten Klasse, die aus dem Bedürfnis heraus entstanden ist, Datumsangaben im System einführen und verwalten zu können, da es einen entsprechenden Datumstypen nicht vordefiniert gab.

Attribute:

day	Dieses Klassenattribut dient zur Erfassung des genauen Tages. Der Datentyp ist ein `Integer`.
month	In diesem Attribut wird der jeweilige Monat gespeichert. Als Datentyp für dieses Klassenattribut kommt ebenfalls ein `Integer` zum Einsatz.
year	Speichert das zu einem Datum zugehörige Jahr. Der Datentyp dieses Klassenattributs ist ein `Integer`.

3.2.19. Period

Klassenbeschreibung:

Die Klasse `Period` stellt eine Art Ergänzung der Klasse `Date` dar und dient der Erfassung von Zeiträumen. Sie ist ebenfalls ein Produkt der gemeinsamen Entwicklung im Rahmen der Lehrveranstaltung. Die Klasse kommt im entworfenen Schulungssystem nicht direkt zum Einsatz, sollte der Vollständigkeit halber aber dennoch erwähnt und beschrieben werden.

Attribute:

start	Dieses Klassenattribut gibt den Startpunkt einer zeitlichen Periode an. Der Datentyp ist eine Instanz der Klasse `Date`.
ends	Dieses Klassenattribut gibt den Endpunkt einer zeitlichen Periode an. Der Datentyp ist eine Instanz der Klasse `Date`.

3.3. Assoziationen, Kompositionen und Aggregationen

3.3.1. IsAnInstructor

Bei der Beziehung `IsAnInstructor` handelt es sich um eine Assoziation zwischen den Klassen `Person` und `Instructor`. Eine Person kann dabei maximal ein `Instructor` sein, muss es aber nicht.

3.3.2. IsAParticipant

Bei der Beziehung `IsAParticipant` handelt es sich um eine Assoziation zwischen den Klassen `Person` und `Participant`. Eine Person kann dabei maximal ein `Participant` sein, muss es aber nicht.

3.3.3. WorksFor

Bei der Beziehung `WorksFor` handelt es sich um eine Assoziation zwischen den beiden Klassen `Participant` und `Company`. Personen können dabei maximal mit einer `Company` assoziiert sein, müssen es aber nicht. Natürlich kann ein Teilnehmer theoretisch für mehrere Firmen arbeiten, allerdings muss dieser Fall hier nicht abgedeckt werden, da es unwahrscheinlich ist, dass ein Teilnehmer zeitgleich von seinen beiden Firmen zu demselben Fortbildungsseminar geschickt wird.

3.3.4. Registration

Die Beziehung `Registration` ist mit der Assoziationsklasse des gleichen Namens verknüpft. Eine Beziehung `Registration` kann immer zwischen den Klassen `Participant` und `TrainingSession` entstehen, wobei ein `Participant` mit beliebig vielen Instanzen von `TrainingSession` verbunden sein kann und umgekehrt.

3.3.5. Bill

Bei `Bill` handelt es sich um eine Assoziation zwischen der Klasse `Bill` und der Assoziationsklasse `Registration`. Jede Rechnung kann dabei mit genau einer Teilnehmeranmeldung verbunden sein. `Registration` kann wiederum mit maximal einer Rechnung verknüpft sein, muss es aber nicht.

3.3.6. Category

Bei der Beziehung `Category` handelt es sich um eine Assoziation zwischen den beiden Klassen `Training` und `Category`. Ein `Training` ist dabei immer mit exakt einer Instanz von `Category` verbunden. Jede `Category` kann jedoch mit beliebig vielen Instanzen von `Training` verbunden sein.

3.3.7. SubCategory

Bei der Beziehung `SubCategory` handelt es sich um eine Aggregation zwischen zwei Instanzen der Klasse `Category`. Eine Oberkategorie hat dabei beliebig viele Unterkategorien, kann selbst aber immer nur maximal eine Oberkategorie haben, muss dies aber nicht.

3.3.8. TrainingSession

Bei der Beziehung `TrainingSession` handelt es sich um eine Komposition zwischen den beiden Klassen `Training` und `TrainingSession`. Es wurde die Komposition gewählt, da es sich hierbei um eine Teil-Ganze-Beziehung handelt. Die `TrainingSession` ist dabei immer fest mit einem `Training` verbunden. Umgekehrt kann ein `Training` eine Beziehung zu beliebig vielen Instanzen von `TrainingSession` besitzen.

3.3.9. Appointment

Bei der Beziehung `Appointment` handelt es sich um eine Komposition zwischen den beiden Klassen `Appointment` und `TrainingSession`. Es wurde die Komposition gewählt, da es sich hierbei um eine Teil-Ganze-Beziehung handelt. Jede `TrainingSession` ist dabei immer fest mit einem oder mehreren Instanzen von `Appointment` verbunden. Umgekehrt kann ein `Appointment` immer nur in Beziehung mit einer Instanz von `TrainingSession` stehen.

3.3.10. Room

Bei der Beziehung `Room` handelt es sich um eine Assoziation zwischen den beiden Klassen `Room` und `Appointment`. Ein `Appointment` kann dabei maximal mit einer Instanz von `Room` verbunden sein. Jeder `Room` kann jedoch mit beliebig vielen Instanzen von `Appointment` verbunden sein.

3.3.11. Catering

Bei der Beziehung `Catering` handelt es sich um eine Komposition zwischen den beiden Klassen `Catering` und `Appointment`. Es wurde die Komposition gewählt, da es sich hierbei um eine Teil-Ganze-Beziehung handelt. Jedes `Appointment` ist dabei maximal mit einer Instanzen von `Catering` verbunden. Umgekehrt ist ein `Catering` immer mit genau einer Instanz von `Appointment` verbunden.

3.3.12. Document

Bei der Beziehung `Document` handelt es sich um eine Assoziation zwischen den beiden Klassen `Document` und `Training`. Ein `Document` ist dabei immer mit genau einer Instanz von `Training` verbunden. Jedes `Training` kann jedoch mit beliebig vielen Instanzen von `Document` verbunden sein.

3.3.13. Teaches

Die Beziehung `Teaches` ist mit der Assoziationsklasse des gleichen Namens verknüpft. Sie verbindet die beiden Klassen `Instructor` und `Appointment`. Ein `Instructor` kann dabei mit beliebig vielen `Appointment` Instanzen verbunden sein. Umgekehrt hat ein `Appointment` maximal einen `Instructor` besitzen.

3.3.14. Qualification

Bei der Beziehung `Qualification` handelt es sich um eine Assoziation zwischen den beiden Klassen `Instructor` und `Tool`. Ein `Tool` kann dabei mit beliebig vielen Instanzen von `Instructor` verbunden sein und umgekehrt.

3.3.15. Tool

Bei der Beziehung `Tool` handelt es sich um eine Assoziation zwischen den beiden Klassen `Training` und `Tool`. Ein `Training` kann dabei mit beliebig vielen Instanzen von `Tool` verbunden sein und umgekehrt.

3.4. Operationen

3.4.1. Person::init()

Die Funktion `init` wird genutzt, um die Klassenvariablen `firstname`, `lastname`, `birth`, `gender`, `address` und `phone` mit gültigen Werten zu belegen.

Parameter:

aFirstname	Beinhaltet den Vornamen der Person. Der Datentyp ist ein `String`.
aLastname	Enthält den Nachnamen der Person. Der Datentyp ist ein `String`.
aBirth	Gibt das Geburtsdatum der Person an. Als Datentyp dient die Klasse `Date`.
aGender	Dient zur Erfassung des Geschlechts der Person. Als Datentyp dient die Enumeration `Gender`.
anAddress	Beinhaltet den primären Wohnsitz der Person. Der Datentyp ist ein `String`.
aPhone	Repräsentiert die Telefonnummer der Person. Der Datentyp ist ein `String`.

Rückgabewert:

Diese Funktion liefert keinen Rückgabewert.

Preconditions:

freshInstance	Beim Aufruf der Funktion muss es sich bei dem Objekt, auf dem sie aufgerufen wird, um eine frisch erzeugte Instanz handeln, d.h. die betroffenen Klassenvariablen müssen `undefined` sein.

Postconditions:

firstnameDefined	Der in der Klassenvariable `firstname` gespeicherte Vorname der Person entspricht dem Namen, der als Parameter `aFirstname` übergeben wurde.
lastnameDefined	Der Nachname, gespeichert im Attribut `lastname`, hat den Wert aus dem Parameter `aLastname` zugewiesen bekommen.
birthDefined	Das Geburtsdatum der Person, angegeben als Parameter `aBirth`, wurde in der Klassenvariablen `birth` gespeichert.
genderDefined	Das Geschlecht der Person, welches in der Klassenvariable `gender` gespeichert wird, entspricht dem Wert des Parameters `aGender`.
addressDefined	Die im Attribut `address` gespeicherte Adresse der Person entspricht der Adresse, die als Parameter `anAddress` übergeben wurde.
phoneDefined	Die Klassenvariable `phone` besitzt den als Parameter übergebenen Wert `aPhone`.

3.4.2. Person::becomeParticipant()

Die Funktion `becomeParticipant` wird genutzt, um einen neuen Schulungsteilnehmer im System auf Basis der in `Person` erfassten Daten zu erzeugen.

Parameter:

Diese Funktion enthält keine Parameter.

Rückgabewert:

Der Rückgabewert der Funktion ist das `Participant`-Objekt, welches die Rolle des Schulungsteilnehmers für die Person im System repräsentiert.

Preconditions:

isNoParticipant	Beim Aufruf dieser Funktion darf die erfasste Person noch nicht in einen Schulungsteilnehmer umgewandelt worden sein.

Postconditions:

isParticipant	Nach dem Aufruf dieser Funktion muss die erfasste Person erfolgreich in einen Schulungsteilnehmer umgewandelt worden sein.
participantIsNew	Mit dieser Postcondition wird sichergestellt, dass das `Participant`-Objekt für die Person während der Operation neu erstellt wurde. *Diese Postcondition ist auskommentiert, da USE zum Zeitpunkt dieser Ausarbeitung dieses OCL-Feature noch nicht unterstützt.*

3.4.3. Person::becomeInstructor()

Die Funktion `becomeInstructor` wird genutzt, um einen neuen Dozenten im System auf Basis der in `Person` erfassten Daten zu erzeugen.

Parameter:

Diese Funktion enthält keine Parameter.

Rückgabewert:

Der Rückgabewert der Funktion ist das `Instructor`-Objekt, welches die Rolle des Dozenten für die Person im System repräsentiert.

Preconditions:

isNoInstructor	Beim Aufruf dieser Funktion darf die erfasste Person noch nicht in einen Dozenten umgewandelt worden sein.

Postconditions:

isInstructor	Nach dem Aufruf dieser Funktion muss die erfasste Person erfolgreich in einen Dozenten umgewandelt worden sein.
instructorIsNew	Mit dieser Postcondition wird sichergestellt, dass das `Instructor`-Objekt für die Person während der Operation neu erstellt wurde. *Diese Postcondition ist auskommentiert, da USE zum Zeitpunkt dieser Ausarbeitung dieses OCL-Feature noch nicht unterstützt.*

3.4.4. Participant::registerForTrainingSession()

Die Funktion `registerForTrainingSesseion` wird genutzt, um einen potentiellen Schulungsteilnehmer einer konkreten Schulung zuzuordnen.

Parameter:

aTrainingSession	Dieser Parameter gibt eine konkrete Instanz der Klasse `TrainingSession` an, mit der der Teilnehmer verbunden werden soll.

Rückgabewert:

Der Rückgabewert der Funktion ist das `Registration`-Objekt, welches die Anmeldung eines Teilnehmers an einer Schulung repräsentiert.

Preconditions:

isNotRegistered	Beim Aufruf dieser Funktion darf die erfasste Person noch nicht mit der als Parameter übergebenen Schulungsinstanz verbunden sein. Eine verknüpfung des Teilnehmers mit anderen Schulung ist natürlich dennoch möglich.

Postconditions:

isRegistered	Nach dem Aufruf dieser Funktion wurde der Teilnehmer erfolgreich mit der als Parameter übergebenen Schulungsinstanz verbunden.
numIncreased	Die Anzahl aller Teilnehmer, die an der angegebenen Schlungsinstanz teilnehmen, wurde um eine Person erhöht.
registrationIsNew	Mit dieser Postcondition wird sichergestellt, dass das `Registration`-Objekt für die Anmeldung während der Operation neu erstellt wurde. *Diese Postcondition ist auskommentiert, da USE zum Zeitpunkt dieser Ausarbeitung dieses OCL-Feature noch nicht unterstützt.*
isNotAccepted	Der Teilnehmer ist nach dem Aufruf dieser Funktion zwar für eine Schulung registriert, in dieser aber noch nicht angenommen. Hierfür müsste er zunächst die gestellte Rechnung bezahlen.

3.4.5. Instructor::addQualification()

Die Funktion `addQualification` wird genutzt, um einem Dozenten eine Qualifikation für ein `Tool` zuzuordnen, was ihm dann ermöglicht Schulungen für dieses Tool durchführen zu dürfen.

Parameter:

t	Dieser Parameter gibt eine konkrete Instanz der Klasse `Tool` an, für welche der Dozent die Qualifikation, um es zu lehren, besitzt.

Rückgabewert:

Diese Funktion liefert keinen Rückgabewert.

Preconditions:

isNotQualified	Beim Aufruf dieser Funktion darf der Dozent noch nicht mit dem als Parameter übergebenen Tool im Sinne einer Qualifikation verbunden sein. Eine verknüpfung des Dozenten mit anderen Tools, die er ebenfalls lehren darf, ist natürlich dennoch möglich.

Postconditions:

isQualified	Nach dem Aufruf dieser Funktion wurde die Qualifizierung des Dozenten für das als Parameter übergebenen Tool erfolgreich im System vermerkt.
numIncreased	Die Anzahl aller Tools, die der Dozent lehren darf, wurde um ein weiteres Tool erhöht.

3.4.6. Company::init()

Die Funktion `init` wird genutzt, um die Klassenvariablen `name`, `address` und `phone` mit gültigen Werten zu belegen.

Parameter:

aName	Beinhaltet den Namen, welcher der Firma zugewiesen werden soll. Der Datentyp ist ein `String`.
anAddress	Beinhaltet die Kontaktadresse des Firmenstandortes oder des Standortes einer Zweigstelle der Firma. Der Datentyp ist ein `String`.
aPhone	Repräsentiert die Telefonnummer der Firma, unter der das Schlungsunternehmen sie erreicht kann. Der Datentyp ist ein `String`.

Rückgabewert:

Diese Funktion liefert keinen Rückgabewert.

Preconditions:

freshInstance	Beim Aufruf der Funktion muss es sich bei dem Objekt, auf dem sie aufgerufen wird, um eine frisch erzeugte Instanz handeln, d.h. die betroffenen Klassenvariablen müssen `undefined` sein.

Postconditions:

nameDefined	Der in der Klassenvariable `name` gespeicherte Name der Firma entspricht dem Namen, der als Parameter `aName` übergeben wurde.
addressDefined	Die in der Klassenvariable `address` gespeicherte Adresse des Firmensitzes oder der Zweigstelle entspricht der Adresse, die als Parameter `anAddress` übergeben wurde.
phoneDefined	Die Klassenvariable `phone` besitzt den als Parameter übergebenen Wert `aPhone`.

3.4.7. Company::addEmployee()

Die Funktion `addEmployee` wird genutzt, um Schulungsteilnehmer (`Participant`) mit einer Firma zu assoziieren, bei die der Teilnehmer arbeitet, und welche (möglicherweise) die Schulung angeordnet hat.

Parameter:

p Eine Instanz der Klasse `Participant`, die den Teilnehmer repräsentiert, der mit dieser Firma verbunden werden soll.

Rückgabewert:

Diese Funktion liefert keinen Rückgabewert.

Preconditions:

isNoEmployee Beim Aufruf dieser Funktion darf der als Parameter übergebene Teilnehmer noch nicht als Angestellter dieser Firma registriert sein.

Postconditions:

isEmployee Nach dem Aufruf der dieser Funktion muss der als Parameter übergebene Schlungsteilnehmer erfolgreich als Angestellter der Firma registriert sein.

numIncreased Die Anzahl der mit dieser Firma verbundenen Mitarbeiter ist um einen Mitarbeiter angestiegen.

3.4.8. Category::init()

Die Funktion `init` wird genutzt, um die Klassenvariable `title`, die den Namen einer Kategorie enthält, mit einem gültigen Wert zu belegen.

Parameter:

aTitle Gibt den Namen an, den die Kategorie tragen soll. Der Datentyp dieses Parameters ist ein `String`.

Rückgabewert:

Diese Funktion liefert keinen Rückgabewert.

Preconditions:

freshInstance Beim Aufruf der Funktion muss es sich bei dem Objekt, auf dem sie aufgerufen wird, um eine frisch erzeugte Instanz handeln, d.h. die betroffenen Klassenvariablen müssen `undefined` sein.

Postconditions:

titleDefined Der in dem Attribut `title` gespeicherte Name der Kategorie entspricht dem Namen, der als Parameter `aTitle` übergeben wurde.

3.4.9. Category::mkFlatOrderedTree()

Die Funktion `mkFlatOrderedTree`, definiert in Quellcode 3.1, macht aus der Baumstruktur der Kategorien eine flache Sequenz, die zusätzlich, auf den einzelnen Gliederungsebenen, alphabetisch nach den Titeln der Kategorien geordnet ist.

Hierzu wird eine Tiefensuche auf den Baum angewandt und jede Ebene alphabetisch sortiert.

Parameter:

cats Die Wurzelkategorien, dessen Baum in eine flache, geordnete Struktur umgewandelt werden soll als Sequenz von `Category`-Objekten.

Rückgabewert:

Geordnete Sequenz der Kategorien von den übergebenen Kategoriebäumen.

Preconditions:

Diese Funktion besitzt keine Preconditions.

Postconditions:

Diese Funktion besitzt keine Postconditions.

```
1 mkFlatOrderedTree( cats : Sequence(Category) ) : Sequence(Category)=
2   cats→sortedBy( c | c.title )→iterate( c ; ret : Sequence(Category)
      = Sequence{} |
3     if c.child→isEmpty() then
4       ret→append( c )
5     else
6       ret→append( c )→union( c.mkFlatOrderedTree(
          c.child→asSequence() ))
7     endif )
```

Quellcode 3.1: Implementierung der Funktion `mkFlatOrderedTree`

3.4.10. Category::mkBreadcrumb()

Die Funktion `mkBreadcrumb` erzeugt eine Sequenz aus Kategorien, welche von der Wurzelkategorie bis zur Kategorie reicht, auf der diese Funktion aufgerufen wurde. Dadurch entsteht eine so genannte „Brotkrümelnavigation“, in der alle Schritte von der Wurzel bis zum Ziel enthalten sind.

Parameter:

Diese Funktion enthält keine Parameter.

Rückgabewert:

Alle Kategorien auf dem Pfad von der Wurzelkategorie bis zu dem Element, auf dem die Funktion aufgerufen wurde.

Preconditions:

Diese Funktion besitzt keine Preconditions.

Postconditions:

Diese Funktion besitzt keine Postconditions.

3.4.11. Training::init()

Die Funktion `init` wird genutzt, um die Klassenvariablen `id`, `title`, `description` und `cost` mit gültigen Werten zu belegen und das Objekt somit korrekt zu initialisieren.

Parameter:

anId	Gibt eine eindeutige ID an, mit der das Training im System identifiziert werden kann. Der Datentyp dieses Parameters ist `Integer`.
aTitle	Dieser Parameter gibt einen Titel für das Training an. Der Datentyp dieses Parameters ist `String`.
aDescription	Gibt eine Beschreibung an, die diesem Training zugeordnet werden soll. Der Datentyp dieses Parameters ist `String`.
aCost	Dieser Parameter definiert den Preis, den das Training pro Teilnehmer kosten soll. Der Datentyp dieses Parameters ist `Real`.

Rückgabewert:

Diese Funktion liefert keinen Rückgabewert.

Preconditions:

freshInstance	Beim Aufruf der Funktion muss es sich bei dem Objekt, auf dem sie aufgerufen wird, um eine frisch erzeugte Instanz handeln, d.h. die betroffenen Klassenvariablen müssen `undefined` sein.

Postconditions:

idDefined	Die in der Klassenvariable `id` gespeicherte eindeutige ID der Schulung entspricht der ID, die als Parameter `anId` übergeben wurde.
titleDefined	Der Titel der Schulung, gespeichert in der Klassenvariable `title` hat den Wert aus dem Parameter `aTitle` zugewiesen bekommen.
descriptionDefined	Der Beschreibungstext der Schulung, angegeben als Parameter `aDescription`, wurde in der Klassenvariablen `description` gespeichert.
costDefined	Der Preis der Schulung, welche in der Klassenvariable `cost` gespeichert wird, entspricht dem Wert des Parameters `aCost`.

3.4.12. Training::addTrainingSession()

Diese wird genutzt, um eine neue Instanz von der Klasse `TrainingSession` zu erzeugen, die eine konkrete Schulung auf Basis der in diesem Training hinterlegten Daten repräsentiert.

Parameter:

Diese Funktion enthält keine Parameter.

Rückgabewert:

Als Rückgabewert liefert die Funktion eine Instanz von `TrainingSession`.

Preconditions:

Diese Funktion besitzt keine Preconditions.

Postconditions:

sessionIsNew	Mit dieser Postcondition wird sichergestellt, dass das `TrainingSession`-Objekt für die Person während der Operation neu erstellt wurde. *Diese Postcondition ist auskommentiert, da USE zum Zeitpunkt dieser Ausarbeitung dieses OCL-Feature noch nicht unterstützt.*
sessionIsLinked	Die neu erzeugte Instanz von `TrainingSession` muss mit dieser Vorlage vom Typ `Training` verlinkt und somit in Beziehung gebracht worden sein.

3.4.13. TrainingSession::init()

Die Funktion `init` wird genutzt, um die Klassenvariablen `state` mit gültigen Werten zu belegen und das Objekt somit korrekt zu initialisieren.

Parameter:

Diese Funktion enthält keine Parameter.

Rückgabewert:

Diese Funktion liefert keinen Rückgabewert.

Preconditions:

freshInstance	Beim Aufruf der Funktion muss es sich bei dem Objekt, auf dem sie aufgerufen wird, um eine frisch erzeugte Instanz handeln, d.h. die betroffenen Klassenvariablen müssen `undefined` sein.

Postconditions:

stateDefined	Nach dem Aufruf dieser Funktion muss sich die Instanz von `TrainingSession` im Zustand `announced` befinden.

3.4.14. TrainingSession::addAppointment()

Die Funktion `addAppointment` wird genutzt, um einer Instanz von `TrainingSession` ein Datum und eine Uhrzeit hinzuzufügen, gekapselt in einer Instanz von `Appointment`, an der die Schulung abgehalten werden soll.

Parameter:

aDate	Dieser Parameter gibt das Datum an, welches in dem zu erzeugenden `Appointment` gespeichert werden soll. Der Datentyp ist von der Klasse `Date`
aTime	Wird genutzt, um die Uhrzeit anzugeben, an der die Schulung startet. Der Datentyp ist ein `String`.

Rückgabewert:

Die Funktion liefert eine Instanz der Klasse `Appointment` zurück, die ein Datum und eine Uhrzeit für diese Instanz von `TrainingSession` enthält.

Preconditions:

Diese Funktion verfügt über keine Preconditions.

Postconditions:

appointmentIsNew	Mit dieser Postcondition wird sichergestellt, dass das `Appointment`-Objekt für die Person während der Operation neu erstellt wurde. *Diese Postcondition ist auskommentiert, da USE zum Zeitpunkt dieser Ausarbeitung dieses OCL-Feature noch nicht unterstützt.*
appointmentIsLinked	Nach dem Aufruf dieser Funktion muss die neue Instanz der Klasse `Appointment` mit dieser Instanz von `TrainingSession` verbunden sein.
valuesCorrect	Die als Parameter übergebenen Angaben zum Datum und zu der Uhrzeit müssen in der neuen Instanz der Klasse `Appointment` gespeichert worden sein.

3.4.15. TrainingSession::confirmRegistration()

Die Funktion `confirmRegistration` ist nötig, um um einen Teilnehmer (Instanz von `Participant`) vom Status nicht akzeptiert auf den Status akzeptiert hochzustufen. Dies darf jedoch nur möglich sein, wenn der Teilnehmer sich zuvor für die `TrainingSession` registriert hat und noch genügend Plätze im Schulungsraum vorhanden sind.

Parameter:

aParticipant	Dieser Parameter enthält die Instanz des Teilnehmers, der für diese Schulung akzeptiert werden soll.

Rückgabewert:

Diese Funktion liefert keinen Rückgabewert.

Preconditions:

participantRegistered	Die als Parameter übergebenen Instanz der Klasse `Participant` muss in der Menge der Teilnehmer, die sich für diese Schulung registriert haben, vorkommen.
isNotAccepted	Der Teilnehmer darf zu diesem Zeitpunkt noch nicht für die Schulung akzeptiert worden sein.
roomsHaveEnoughSeats	Der Raum, in dem die Schulung stattfindet, muss über genügend freie Sitze verfügen, um die Registrierung erfolgreich durchführen zu können.

Postconditions:

isAccepted	Der Schulungsteilnehmer, der als Parameter dieser Funktion übergeben wurde, muss nun in der Menge der akzeptierten Personen, die an dieser Schulung teilnehmen dürfen, enthalten sein.

3.4.16. TrainingSession::settleTrainingSession()

Die Funktion `settleTrainingSession` ändert den Status einer `TrainingSession` von `announced` zu `settled`, was bedeutet, dass die Schulung nun angesetzt ist und durchgeführt werden kann.

Parameter:

Diese Funktion enthält keine Parameter.

Rückgabewert:

Diese Funktion liefert keinen Rückgabewert.

Preconditions:

stateAnnounced	Die Schulung muss sich bisher im Zustand `announced` befunden haben.
hasParticipants	Die Menge der Personen, die an dieser Schulung teilnehmen wollen und dafür akzeptiert worden sind, darf nicht leer sein.

Postconditions:

stateSettled	Die Schulung muss sich nach dem Aufruf dieser Funktion im Zustand `settled` befinden.
billsExist	Für alle registrierten Teilnehmer, die an dieser Schulung teilnehmen werden, muss eine Rechnung gestellt worden sein.

3.4.17. TrainingSession::cancelTrainingSession()

Die Funktion `cancelTrainingSession` ändert den Status von `TrainingSession` von `settled` zu `canceled`, was bedeutet, dass die angesetzte Schulung abgesagt wurde.

Parameter:

Diese Funktion enthält keine Parameter.

Rückgabewert:

Diese Funktion liefert keinen Rückgabewert.

Preconditions:

isNotCanceled	Die abzusagende Schulung darf sich nicht bereits im Zustand `canceled` befinden.
isNotHeld	Die abzusagende Schulung darf sich nicht im Zustand `held`, welcher angibt, dass die Schulung bereits gehalten wurde, befinden.

Postconditions:

stateCanceled	Die Schulung muss sich nach dem Aufruf dieser Funktion im Zustand `canceled` befinden.
noAcceptedParticipants	Die Menge der für diese Schulung registrierten Teilnehmer muss leer sein.
roomsFree	Nach dem Aufruf dieser Funktion dürfen mit dieser Schulung keinerlei Raumreservierungen mehr verbunden sein.
instructorsFree	Ebenso dürfen keine Dozenten mehr für das Abhalten dieser Schulung eingeplant sein.

3.4.18. TrainingSession::finishTrainingSession()

Die Funktion `finishTrainingSession` ändert den Status von `TrainingSession` von `settled` zu `held`, was bedeutet, dass die angesetzte Schulung erfolgreich und wie geplant durchgeführt werden konnte.

Parameter:

Diese Funktion enthält keine Parameter.

Rückgabewert:

Diese Funktion liefert keinen Rückgabewert.

Preconditions:

isSettled	Die durchgeführte Schulung muss sich zuvor im Zustand `settled` befunden.

Postconditions:

isHeld Nach dem Aufruf dieser Funktion muss die Schulung in den Zustand `held` gewechselt haben.

3.4.19. Appointment::init()

Die Funktion `init` wird genutzt, um die Klassenvariablen `date` und `time` mit gültigen Werten zu belegen.

Parameter:

aDate Dieser Parameter gibt das Datum an, welches in dem zu erzeugenden `Appointment` gespeichert werden soll. Der Datentyp ist von der Klasse `Date`

aTime Wird genutzt, um die Uhrzeit anzugeben. Der Datentyp ist ein `String`.

Rückgabewert:

Diese Funktion liefert keinen Rückgabewert.

Preconditions:

freshInstance Beim Aufruf der Funktion muss es sich bei dem Objekt, auf dem sie aufgerufen wird, um eine frisch erzeugte Instanz handeln, d.h. die betroffenen Klassenvariablen müssen `undefined` sein.

Postconditions:

dateDefined Nach dem Aufruf dieser Funktion muss der Inhalt der Klassenvariable `date` dem Inhalt des Parameters `aDate` entsprechen.

timeDefined Der Inhalt der Klassenvariable `time` muss, nachdem die Funktion aufgerufen worden ist, dem Inhalt des Parameters `aTime` entsprechen.

3.4.20. Appointment::assignRoom()

Die Funktion `assignRoom` wird benötigt, um eine Instanz der Klasse `Appointment` mit einem Raum zu verbinden, in dem die Schulung, die mit einer Instanz der Klasse `Appointment` verbunden ist, stattfinden soll.

Parameter:

r Der Raum, der mit diesem `Appointment` assoziiert werden soll. Der Datentyp dieses Parameters ist `Room`.

Rückgabewert:

Diese Funktion liefert keinen Rückgabewert.

Preconditions:

hasNoRoom Das `Appointment` darf zum Zeitpunkt dieses Funktionsaufrufs mit noch keinem Raum verknüpft sein.

Postconditions:

roomAssigned Nach Abschluss der Operation wurde dem `Appointment` ein Raum zugewiesen.

3.4.21. Appointment::assignInstructor()

Die Funktion `assignInstructor` wird benötigt, um eine Instanz von `Appointment` mit einem Dozenten zu verbinden, der die Schulung, die mit einer Instanz der Klasse `Appointment` verbunden ist, abhalten soll.

Parameter:

i Der Dozent, der mit diesem `Appointment` assoziiert werden soll. Der Datentyp dieses Parameters ist `Instructor`.

Rückgabewert:

Diese Funktion liefert keinen Rückgabewert.

Preconditions:

hasNoInstructor Das `Appointment` darf zum Zeitpunkt dieses Funktionsaufrufs mit noch keinem Dozenten verknüpft sein.

Postconditions:

instructorAssigned Nach dem Aufruf dieser Operation wurde dem `Appointment` ein Dozent zugewiesen.

3.4.22. Appointment::addCatering()

Die Funktion `addCatering` wird benötigt, um eine Instanz der Klasse `Appointment` mit einer neu erzeugten Instanz der Klasse `Catering` in Verbindung zu bringen.

Parameter:

Diese Funktion enthält keine Parameter.

Rückgabewert:

Die Funktion liefert eine neue Instanz einer Klasse vom Typ `Catering` zurück.

Preconditions:

hasNoCatering	Das `Appointment` darf zum Zeitpunkt dieses Funktionsaufrufs mit noch keiner Instanz der Klasse `Catering` verknüpft sein.

Postconditions:

cateringIsNew	Mit dieser Postcondition wird sichergestellt, dass das `Catering`-Objekt für die Person während der Operation neu erstellt wurde. *Diese Postcondition ist auskommentiert, da USE zum Zeitpunkt dieser Ausarbeitung dieses OCL-Feature noch nicht unterstützt.*
cateringIsLinked	Nach Abschluss der Operation wurde die neu erzeugte Instanz der Klasse `Catering` dem `Appointment` zugewiesen.

3.4.23. Room::init()

Die Funktion `init` wird genutzt, um die Klassenvariablen `address`, `seats`, `internal` und `cost` mit gültigen Werten zu belegen.

Parameter:

anAddress	Die Adresse, unter der sich `Room` befindet. Der Datentyp dieses Parameters ist `String`.
numSeats	Die Anzahl der Sitze, die in dem Raum verfügbar sind. Der Datentyp dieses Parameters ist `Integer`.
isInternal	Dieser Parameter gibt an, ob der Raum intern ist oder extern angemietet werden muss. Der Datentyp dieses Parameters ist `Boolean`.
aCost	Falls es sich um einen externen Raum handelt, der angemietet werden muss, beschreibt dieser Parameter die Kosten, die dadurch anfallen. Der Datentyp dieses Parameters ist `Real`.

Rückgabewert:

Diese Funktion liefert keinen Rückgabewert.

Preconditions:

freshInstance	Beim Aufruf der Funktion muss es sich bei dem Objekt, auf dem sie aufgerufen wird, um eine frisch erzeugte Instanz handeln, d.h. die betroffenen Klassenvariablen müssen `undefined` sein.

Postconditions:

addressDefined	Die Klassenvariable `address` wurde auf den Wert des Parameters `anAddress` gesetzt.
seatsDefined	Der Wert der Variablen `seats` entspricht dem Wert des Parameters `numSeats`.
internalDefined	Das Flag `internal` muss entsprechend dem Parameter `isInternal` gesetzt worden sein.
costDefined	Der Wert `cost` muss die Kosten enthalten, die durch den Parameter `aCost` angegeben wurden.

3.4.24. Catering::init()

Die Funktion `init` wird genutzt, um die Klassenvariablen `description` und `cost` mit gültigen Werten zu belegen.

Parameter:

aDescription	Gibt eine Beschreibung des `Catering` an. Der Datentyp dieses Parameters ist ein *String*.
aCost	Gibt die Kosten für das `Catering` an. Der Datentyp dieses Parameters ist ein *Real*.

Rückgabewert:

Diese Funktion liefert keinen Rückgabewert.

Preconditions:

freshInstance	Beim Aufruf der Funktion muss es sich bei dem Objekt, auf dem sie aufgerufen wird, um eine frisch erzeugte Instanz handeln, d.h. die betroffenen Klassenvariablen müssen `undefined` sein.

Postconditions:

descriptionDefined	Die Klassenvariable `description` wurde auf den Wert des Parameters `aDescription` gesetzt.
costDefined	Der Wert `cost` muss die Kosten enthalten, die durch den Parameter `aCost` angegeben wurden.

3.4.25. Software::init()

Die Funktion `init` wird genutzt, um die Klassenvariable `name` mit einem gültigen Werte zu belegen. Dies bedeutet, dass der Software, ein Name zugeordnet wird.

Parameter:

aName Der Name, der dieser Instanz von `Software` zugewiesen werden soll. Der Datentyp dieses Parameters ist ein `String`.

Rückgabewert:

Diese Funktion liefert keinen Rückgabewert.

Preconditions:

freshInstance Beim Aufruf der Funktion muss es sich bei dem Objekt, auf dem sie aufgerufen wird, um eine frisch erzeugte Instanz handeln, d.h. die betroffenen Klassenvariablen müssen `undefined` sein.

Postconditions:

nameDefined Der Wert `name` muss den Softwaretitle enthalten, der durch den Parameter `aName` angegeben wurden.

3.4.26. Hardware::init()

Die Funktion `init` wird genutzt, um die Klassenvariable `name` mit einem gültigen Werte zu belegen. Dies bedeutet, dass der Hardware, ein Name zugeordnet wird.

Parameter:

aName Der Name, der dieser Instanz von `Hardware` zugewiesen werden soll. Der Datentyp dieses Parameters ist ein `String`.

Rückgabewert:

Diese Funktion liefert keinen Rückgabewert.

Preconditions:

freshInstance Beim Aufruf der Funktion muss es sich bei dem Objekt, auf dem sie aufgerufen wird, um eine frisch erzeugte Instanz handeln, d.h. die betroffenen Klassenvariablen müssen `undefined` sein.

Postconditions:

nameDefined Der Wert `name` muss den Hardwarenamen enthalten, der durch den Parameter `aName` angegeben wurden.

3.4.27. Document::init()

Die Funktion `init` wird genutzt, um die Klassenvariable `title` mit einem gültigen Werte zu belegen. Dies bedeutet, dass das Dokument, welches in einer Schulung benutzt werden soll, mit einem Title versehen wird.

Parameter:

aTitle Der Titel, der dieser Instanz von `Document` zugewiesen werden soll. Der Datentyp dieses Parameters ist ein `String`.

Rückgabewert:

Diese Funktion liefert keinen Rückgabewert.

Preconditions:

freshInstance Beim Aufruf der Funktion muss es sich bei dem Objekt, auf dem sie aufgerufen wird, um eine frisch erzeugte Instanz handeln, d.h. die betroffenen Klassenvariablen müssen `undefined` sein.

Postconditions:

titleDefined Der Wert `title` muss den Dokumentnamen enthalten, der durch den Parameter `aTitle` angegeben wurden.

3.4.28. Registration::init()

Die Funktion `init` wird aufgerufen, wenn eine neue Instanz von `Registration` erzeugt wurde und sorgt dafür, dass die Klassenvariable mit einem gültigen Wert versehen wird.

Parameter:

Diese Funktion enthält keine Parameter.

Rückgabewert:

Diese Funktion liefert keinen Rückgabewert.

Preconditions:

freshInstance Beim Aufruf der Funktion muss es sich bei dem Objekt, auf dem sie aufgerufen wird, um eine frisch erzeugte Instanz handeln, d.h. die betroffenen Klassenvariablen müssen `undefined` sein.

Postconditions:

acceptedDefined	Nach dem Aufruf dieser Funktion muss der Status von `Registration` auf `not accepted` gesetzt sein, da dies der initialen Bedeutung einer Instanz von `Registration` entspricht.

3.4.29. Bill::init()

Die Funktion `init` wird aufgerufen, wenn eine neue Instanz von `Bill` erzeugt wurde und sorgt dafür, dass die Klassenvariablen mit gültigen Werten versehen werden. Der in Rechnung zu stellende Betrag schwankt je nachdem, ob die Schulung ein `Catering` enthielt oder nicht. Die Funktion setzen voraus, dass es ein `Appointment` und somit auch einen Dozenten, eine `TrainingSession` und eine Raumzuweisung gibt.

Parameter:

Diese Funktion enthält keine Parameter.

Rückgabewert:

Diese Funktion liefert keinen Rückgabewert.

Preconditions:

freshInstance	Beim Aufruf der Funktion muss es sich bei dem Objekt, auf dem sie aufgerufen wird, um eine frisch erzeugte Instanz handeln, d.h. die betroffenen Klassenvariablen müssen `undefined` sein.

Postconditions:

priceDefined	Der Werte der Klassenvariable `price` wurde auf den Preis der gesamten Schulung gesetzt. Dieser ergibt sich aus den Kosten für das Seminar, dem zugehörigen Raum, dem Dozenten und gegebenenfalls dem Catering, falls dieses für die Schulung angeboten und zur Verfügung gestellt worden ist.
paidDefined	Die Rechnung gilt zu diesem Zeitpunkt noch als nicht bezahlt.

3.4.30. Bill::updatePrice()

Die Funktion `updatePrice` wird aufgerufen, wenn sich der Preis für ein Seminar noch einmal geändert haben sollte. Eine Änderung des Preises ist allerdings nur möglich, wenn die Rechnung noch nicht bezahlt worden ist. Die Funktion setzt voraus, dass es ein `Appointment` und somit auch einen Dozenten, eine `TrainingSession` und eine Raumzuweisung gibt.

Parameter:

Diese Funktion enthält keine Parameter.

Rückgabewert:

Diese Funktion liefert keinen Rückgabewert.

Preconditions:

isNotPaid	Beim Aufruf der Funktion muss sichergestellt sein, dass die Rechnung noch nicht bezahlt worden ist.

Postconditions:

priceCorrect	Der Preis wurde auf einen neuen und korrekten Wert gesetzt. Er setzt sich dabei aus denselben Faktoren zusammen wie unter 3.4.29 auf der vorherigen Seite beschrieben.

3.4.31. Bill::pay()

Die Funktion `pay` wird genutzt, um den Status einer Rechnung so zu verändern, dass diese im System als bezahlt vermerkt wird.

Parameter:

Diese Funktion enthält keine Parameter.

Rückgabewert:

Diese Funktion liefert keinen Rückgabewert.

Preconditions:

isNotPaid	Der Status der Rechnung, auf der die Methode `pay` aufgerufen wird, darf nicht bereits auf `paid` gesetzt sein.

Postconditions:

isPaid	Der Status der Rechnung muss nach dem Funktionsaufruf auf `paid` gesetzt sein.

3.4.32. Teaches::init()

Die Funktion `init` wird aufgerufen, wenn eine neue Instanz von `Teaches` erzeugt wurde und sorgt dafür, dass die Klassenvariable mit einem gültigen Wert versehen wird.

Parameter:

aCost Gibt die Kosten an, die entstehen, wenn das Schulungsunternehmen den Dozenten mit der Veranstaltung beauftragt. Der Datentyp des Parameters ist `Real`.

Rückgabewert:

Diese Funktion liefert keinen Rückgabewert.

Preconditions:

freshInstance Beim Aufruf der Funktion muss es sich bei dem Objekt, auf dem sie aufgerufen wird, um eine frisch erzeugte Instanz handeln, d.h. die betroffenen Klassenvariablen müssen `undefined` sein.

Postconditions:

costDefined Der Preis, der in der Klassenvariablen `cost` gespeichert wurde, muss dem Wert aus dem Parameter `aCost` entsprechen.

3.4.33. Date::equals()

Die Funktion `equals` kann genutzt werden, um zu überprüfen, ob ein Datum der Klasse `Date` mit einem anderen Datum dieser Klasse übereinstimmt.

Parameter:

d Ein Datum, welches mit der aktuellen Instanz dieser Klasse verglichen werden soll. Der Parameter ist eine Klasseninstanz vom Typ `Date`.

Rückgabewert:

Die Funktion liefert einen boolschen Wert zurück, also entweder wahr oder falsch.

Preconditions:

Diese Funktion besitzt keine Preconditions.

Postconditions:

Diese Funktion besitzt keine Postconditions.

3.4.34. Date::after()

Die Funktion `after` kann genutzt werden, um zu überprüfen, ob ein Datum der Klasse `Date` zeitlich gesehen nach einem anderen Datum dieser Klasse liegt.

Parameter:

d Ein Datum, welches mit der aktuellen Instanz dieser Klasse verglichen werden soll. Der Parameter ist eine Klasseninstanz vom Typ `Date`.

Rückgabewert:

Die Funktion liefert einen boolschen Wert zurück, also entweder wahr oder falsch.

Preconditions:

Diese Funktion besitzt keine Preconditions.

Postconditions:

Diese Funktion besitzt keine Postconditions.

3.4.35. Date::before()

Die Funktion `before` kann genutzt werden, um zu überprüfen, ob ein Datum der Klasse `Date` zeitlich gesehen vor einem anderen Datum dieser Klasse liegt.

Parameter:

d Ein Datum, welches mit der aktuellen Instanz dieser Klasse verglichen werden soll. Der Parameter ist eine Klasseninstanz vom Typ `Date`.

Rückgabewert:

Die Funktion liefert einen boolschen Wert zurück, also entweder wahr oder falsch.

Preconditions:

Diese Funktion besitzt keine Preconditions.

Postconditions:

Diese Funktion besitzt keine Postconditions.

3.4.36. Date::liesBetween()

Die Funktion `liesBetween` kann genutzt werden, um zu überprüfen, ob ein Datum der Klasse `Date` zeitlich gesehen zwischen zwei anderen Daten dieser Klasse, die als Parameter angegeben werden, liegt.

Parameter:

start	Das Startdatum, dass die Untergrenze des Zeitraums darstellt. Bei diesem Parameter handelt es sich um eine Klasse vom Typ `Date`.
ending	Das Enddatum, dass die Obergrenze des Zeitraums darstellt. Auch bei diesem Parameter handelt es sich um eine Klasse vom Typ `Date`.

Rückgabewert:

Die Funktion liefert einen boolschen Wert zurück, also entweder wahr oder falsch.

Preconditions:

Diese Funktion besitzt keine Preconditions.

Postconditions:

Diese Funktion besitzt keine Postconditions.

3.4.37. Date::toString()

Die Funktion `toString` wandelt das in dieser Instanz gespeicherte Datum in einen `String` um.

Parameter:

Diese Funktion enthält keine Parameter.

Rückgabewert:

Die Funktion liefert einen `String` zurück, der das aktuelle Datum, welches von dieser Klasseninstanz gehalten wird, in als String repräsentiert.

Preconditions:

Diese Funktion besitzt keine Preconditions.

Postconditions:

Diese Funktion besitzt keine Postconditions.

3.4.38. Date::liesIn()

Die Funktion `liesIn` kann genutzt werden, um zu überprüfen, ob ein Datum der Klasse `Date` zeitlich gesehen in einer als Parameter angegebenen Periode (Instanz der Klasse `Period`) liegt.

Parameter:

p Eine Instanz der Klasse `Period`, die genutzt werden soll, um zu prüfen, ob das Datum in eben dieser Periode liegt.

Rückgabewert:

Die Funktion liefert einen boolschen Wert zurück, also entweder wahr oder falsch.

Preconditions:

Diese Funktion besitzt keine Preconditions.

Postconditions:

Diese Funktion besitzt keine Postconditions.

3.4.39. Period::equals()

Die Funktion `equals` vergleicht eine Instanz der Klasse `Period` mit einer anderen Instanz dieser Klasse und bestimmt, ob beide Perioden identisch sind, d.h. ob der Start- und Endpunkt bei beiden Instanzen gleich ist.

Parameter:

p Eine Instanz der Klasse `Period`, die genutzt werden soll, um sie mit der aktuellen Instanz zu vergleichen.

Rückgabewert:

Die Funktion liefert einen boolschen Wert zurück, also entweder wahr oder falsch.

Preconditions:

Diese Funktion besitzt keine Preconditions.

Postconditions:

Diese Funktion besitzt keine Postconditions.

3.4.40. Period::before()

Die Funktion `before` vergleicht eine Instanz der Klasse `Period` mit einer anderen Instanz dieser Klasse und bestimmt, ob die Periode, auf der diese Funktion aufgerufen wird, endet, bevor die als Parameter übergebenen Periode startet.

Parameter:

p Eine Instanz der Klasse `Period`, die genutzt werden soll, um sie mit der aktuellen Instanz zu vergleichen.

Rückgabewert:

Die Funktion liefert einen boolschen Wert zurück, also entweder wahr oder falsch.

Preconditions:

Diese Funktion besitzt keine Preconditions.

Postconditions:

Diese Funktion besitzt keine Postconditions.

3.4.41. Period::after()

Die Funktion `after` vergleicht eine Instanz der Klasse `Period` mit einer anderen Instanz dieser Klasse und bestimmt, ob die Periode, auf der diese Funktion aufgerufen wird, startet, nachdem die als Parameter übergebenen Periode endet.

Parameter:

p Eine Instanz der Klasse `Period`, die genutzt werden soll, um sie mit der aktuellen Instanz zu vergleichen.

Rückgabewert:

Die Funktion liefert einen boolschen Wert zurück, also entweder wahr oder falsch.

Preconditions:

Diese Funktion besitzt keine Preconditions.

Postconditions:

Diese Funktion besitzt keine Postconditions.

3.4.42. Period::during()

Die Funktion `during` vergleicht eine Instanz der Klasse `Period` mit einer anderen Instanz dieser Klasse und bestimmt, ob die Periode, auf der diese Funktion aufgerufen wird, innerhalb der als Parameter übergebenen Periode liegt.

Parameter:

p Eine Instanz der Klasse `Period`, die genutzt werden soll, um sie mit der aktuellen Instanz zu vergleichen.

Rückgabewert:

Die Funktion liefert einen boolschen Wert zurück, also entweder wahr oder falsch.

Preconditions:

Diese Funktion besitzt keine Preconditions.

Postconditions:

Diese Funktion besitzt keine Postconditions.

3.4.43. Period::contains()

Die Funktion `contains` vergleicht eine Instanz der Klasse `Period` mit einer anderen Instanz dieser Klasse und bestimmt, ob die Periode, die als Parameter übergeben wurde, innerhalb der Periode, auf der die Funktion aufgerufen wird, liegt.

Parameter:

p Eine Instanz der Klasse `Period`, die genutzt werden soll, um sie mit der aktuellen Instanz zu vergleichen.

Rückgabewert:

Die Funktion liefert einen boolschen Wert zurück, also entweder wahr oder falsch.

Preconditions:

Diese Funktion besitzt keine Preconditions.

Postconditions:

Diese Funktion besitzt keine Postconditions.

3.4.44. Period::overlaps()

Die Funktion `overlaps` vergleicht eine Instanz der Klasse `Period` mit einer anderen Instanz dieser Klasse und bestimmt, ob die Periode, auf der diese Funktion aufgerufen wird, mit der als Parameter übergebenen Periode überlappt. Dies würde bedeuten, dass der Start der als Parameter übergebenen Periode innerhalb der vorhandenen Instanz liegt, das Ende jedoch nicht mehr.

Parameter:

p Eine Instanz der Klasse `Period`, die genutzt werden soll, um sie mit der aktuellen Instanz zu vergleichen.

Rückgabewert:

Die Funktion liefert einen boolschen Wert zurück, also entweder wahr oder falsch.

Preconditions:

Diese Funktion besitzt keine Preconditions.

Postconditions:

Diese Funktion besitzt keine Postconditions.

3.4.45. Period::overlapped_by()

Die Funktion `overlapped_by` vergleicht eine Instanz der Klasse `Period` mit einer anderen Instanz dieser Klasse und bestimmt, ob die Periode, die als Parameter übergeben wurde, mit der Periode, die durch diese Instanz beschrieben wird, überlappt. Dies würde bedeuten, dass der Start der vorhandenen Instanz innerhalb der als Parameter übergebenen Methode liegt, das Ende jedoch nicht mehr.

Parameter:

p Eine Instanz der Klasse `Period`, die genutzt werden soll, um sie mit der aktuellen Instanz zu vergleichen.

Rückgabewert:

Die Funktion liefert einen boolschen Wert zurück, also entweder wahr oder falsch.

Preconditions:

Diese Funktion besitzt keine Preconditions.

Postconditions:

Diese Funktion besitzt keine Postconditions.

3.4.46. Period::meets()

Die Funktion `meets` vergleicht eine Instanz der Klasse `Period` mit einer anderen Instanz dieser Klasse und bestimmt, ob die Periode, die als Parameter übergeben wurde, auf die Periode, die durch diese Instanz beschrieben wird, trifft. Dies würde bedeuten, dass das Ende der vorhandenen Instanz mit dem Start der als Parameter übergebenen Methode übereinstimmt.

Parameter:

p Eine Instanz der Klasse `Period`, die genutzt werden soll, um sie mit der aktuellen Instanz zu vergleichen.

Rückgabewert:

Die Funktion liefert einen boolschen Wert zurück, also entweder wahr oder falsch.

Preconditions:

Diese Funktion besitzt keine Preconditions.

Postconditions:

Diese Funktion besitzt keine Postconditions.

3.4.47. Period::met_by()

Die Funktion `met_by` vergleicht eine Instanz der Klasse `Period` mit einer anderen Instanz dieser Klasse und bestimmt, ob die Periode, die von dieser Instanz repräsentiert wird, auf die Periode, die als Parameter übergeben wurde, trifft. Dies würde bedeuten, dass der Start der vorhandenen Instanz mit dem Ende der als Parameter übergebenen Methode übereinstimmt.

Parameter:

p Eine Instanz der Klasse `Period`, die genutzt werden soll, um sie mit der aktuellen Instanz zu vergleichen.

Rückgabewert:

Die Funktion liefert einen boolschen Wert zurück, also entweder wahr oder falsch.

Preconditions:

Diese Funktion besitzt keine Preconditions.

Postconditions:

Diese Funktion besitzt keine Postconditions.

3.4.48. Period::starts()

Die Funktion `starts` vergleicht eine Instanz der Klasse `Period` mit einer anderen Instanz dieser Klasse und bestimmt, ob die Periode, die von dieser Instanz repräsentiert wird, der Startpunkt der als Parameter übergebenen Periode ist. Dies würde bedeuten, dass der Start der vorhandenen Instanz mit dem Start der als Parameter übergebenen Methode übereinstimmt, die vorhandene Instanz jedoch vor der als Parameter übergebenen Instanz endet.

Parameter:

p Eine Instanz der Klasse `Period`, die genutzt werden soll, um sie mit der aktuellen Instanz zu vergleichen.

Rückgabewert:

Die Funktion liefert einen boolschen Wert zurück, also entweder wahr oder falsch.

Preconditions:

Diese Funktion besitzt keine Preconditions.

Postconditions:

Diese Funktion besitzt keine Postconditions.

3.4.49. Period::started_by()

Die Funktion `started_by` vergleicht eine Instanz der Klasse `Period` mit einer anderen Instanz dieser Klasse und bestimmt, ob die Periode, die als Parameter übergeben wurde, der Startpunkt der von dieser Instanz repräsentierten Periode ist. Dies würde bedeuten, dass der Start der vorhandenen Instanz mit dem Start der als Parameter übergebenen Methode übereinstimmt, die als Parameter übergebene Periode jedoch vor der vorhandenen Instanz endet.

Parameter:

p Eine Instanz der Klasse `Period`, die genutzt werden soll, um sie mit der aktuellen Instanz zu vergleichen.

Rückgabewert:

Die Funktion liefert einen boolschen Wert zurück, also entweder wahr oder falsch.

Preconditions:

Diese Funktion besitzt keine Preconditions.

Postconditions:

Diese Funktion besitzt keine Postconditions.

3.4.50. Period::finishes()

Die Funktion `finishes` vergleicht eine Instanz der Klasse `Period` mit einer anderen Instanz dieser Klasse und bestimmt, ob die Periode, die von dieser Instanz repräsentiert wird, der Endpunkt der als Parameter übergebenen Periode ist. Dies würde bedeuten, dass das Ende der vorhandenen Instanz mit dem Ende der als Parameter übergebenen Methode übereinstimmt, die vorhandene Instanz jedoch vor der als Parameter übergebenen Instanz startet.

Parameter:

p Eine Instanz der Klasse `Period`, die genutzt werden soll, um sie mit der aktuellen Instanz zu vergleichen.

Rückgabewert:

Die Funktion liefert einen boolschen Wert zurück, also entweder wahr oder falsch.

Preconditions:

Diese Funktion besitzt keine Preconditions.

Postconditions:

Diese Funktion besitzt keine Postconditions.

3.4.51. Period::finished_by()

Die Funktion `finished_by` vergleicht eine Instanz der Klasse `Period` mit einer anderen Instanz dieser Klasse und bestimmt, ob die Periode, die als Parameter übergeben wurde, der Endpunkt der von dieser Instanz repräsentierten Periode ist. Dies würde bedeuten, dass das Ende der vorhandenen Instanz mit dem Ende der als Parameter übergebenen Methode übereinstimmt, die als Parameter übergebene Periode jedoch vor der vorhandenen Instanz startet.

Parameter:

p Eine Instanz der Klasse `Period`, die genutzt werden soll, um sie mit der aktuellen Instanz zu vergleichen.

Rückgabewert:

Die Funktion liefert einen boolschen Wert zurück, also entweder wahr oder falsch.

Preconditions:

Diese Funktion besitzt keine Preconditions.

Postconditions:

Diese Funktion besitzt keine Postconditions.

3.5. Invarianten

3.5.1. Person::valuesDefined

Die Invariante *valuesDefined* der Klasse `Person` stellt sicher, dass alle benötigten Klassenvariablen (`firstname`, `lastname`, `birth`, `gender`, `address` und `phone`) definiert wurden und sich keine dieser Variablen im Zustand *undefined* befindet.

Die Tests für diese Invarianten befinden sich in Abschnitt 5.23 auf Seite 86.

3.5.2. Instructor::noDoubleOccupancy

Die Invariante *noDoubleOccupancy* der Klasse `Instructor` soll zu jedem Zeitpunkt sicherstellen, dass ein Dozent nicht zur selben Zeit für mehrere `Appointment` eingetragen ist. Doppelbelegungen eines Dozenten sollen somit verhindert werden.

Die Tests für diese Invarianten befinden sich in Abschnitt 5.4 auf Seite 69.

3.5.3. Company::valuesDefined

Die Invariante *valuesDefined* der Klasse `Company` stellt sicher, dass alle benötigten Klassenvariablen (`name`, `address` und `phone`) definiert wurden und sich keine dieser Variablen im Zustand *undefined* befindet.

Die Tests für diese Invarianten befinden sich in Abschnitt 5.23 auf Seite 86.

3.5.4. Category::valuesDefined

Die Invariante *valuesDefined* der Klasse `Category` stellt sicher, dass alle benötigten Klassenvariablen (in diesem Fall gibt es nur die Klassenvariable `title`) definiert wurden und sich keine dieser Variablen im Zustand *undefined* befindet.

Die Tests für diese Invarianten befinden sich in Abschnitt 5.23 auf Seite 86.

3.5.5. Training::valuesDefined

Die Invariante *valuesDefined* der Klasse `Training` stellt sicher, dass alle benötigten Klassenvariablen (`id`, `title`, `description` und `cost`) definiert wurden und sich keine dieser Variablen im Zustand *undefined* befindet.

Die Tests für diese Invarianten befinden sich in Abschnitt 5.23 auf Seite 86.

3.5.6. Training::costGTEZero

Die Invariante *costGTEZero* der Klasse `Training` dient dazu, sicherzustellen, dass die angesetzten Kosten für eine Schulung immer größer oder gleich Null sind und somit niemals kosten im negativen Zahlenbereich definiert werden.

Die Tests für diese Invarianten befinden sich in Abschnitt 5.5 auf Seite 70

3.5.7. Training::idUnique

Da die Klassenvariable `id` eine im System eindeutige ID für ein Training darstellen soll, die niemals doppelt vergeben sein darf, wurde für die Klasse `Training` die Invariante *idUnique* definiert, die genau diese Anforderungen sicherstellen soll.

Die Tests für diese Invarianten befinden sich in Abschnitt 5.6 auf Seite 71

3.5.8. TrainingSession::valuesDefined

Die Invariante *valuesDefined* der Klasse `TrainingSession` stellt sicher, dass alle benötigten Klassenvariablen (in diesem Fall gibt es nur die Klassenvariable `state`)) definiert wurden und sich diese Variablen niemals im Zustand *undefined* befindet.

Die Tests für diese Invarianten befinden sich in Abschnitt 5.23 auf Seite 86.

3.5.9. TrainingSession::settledMustHaveRooms

Die Invariante *settledMustHaveRooms* der Klasse `TrainingSession` stellt sicher, dass für alle `Appointments`, die mit dieser `TrainingSession` verbunden sind, eine Raumbelegung zugewiesen wurde, im Fall dessen, dass sich die `TrainingSession` entweder im Zustand `settled` oder `held` befindet.

Die Tests für diese Invarianten befinden sich in Abschnitt 5.7 auf Seite 72.

3.5.10. TrainingSession::settledMustHaveInstructors

Die Invariante *settledMustHaveInstructors* der Klasse `TrainingSession` garantiert, dass für alle `Appointments`, die mit dieser `TrainingSession` verbunden sind, ein Dozent definiert ist, falls sich die `TrainingSession` entweder im Zustand `settled` oder `held` befindet.

Die Tests für diese Invarianten befinden sich in Abschnitt 5.8 auf Seite 73.

3.5.11. TrainingSession::canceledHasNoRoom

Die Invariante *canceledHasNoRoom* der Klasse `TrainingSession` stellt sicher, dass für alle `Appointments`, die mit dieser `TrainingSession` verbunden sind, keine Raumreservierung vorhanden ist, im Fall dessen, dass sich die `TrainingSession` im Zustand `canceled` befindet.

Die Tests für diese Invarianten befinden sich in Abschnitt 5.9 auf Seite 73.

3.5.12. TrainingSession::canceledHasNoInstructor

Die Invariante *canceledHasNoInstructor* der Klasse `TrainingSession` stellt sicher, dass für alle `Appointments`, die mit dieser `TrainingSession` verbunden sind, kein Dozent verbunden ist, falls sich die `TrainingSession` im Zustand `canceled` befindet.

Die Tests für diese Invarianten befinden sich in Abschnitt 5.10 auf Seite 74.

3.5.13. TrainingSession::instructorIsNoParticipant

Die Invariante *instructorIsNoParticipant* der Klasse `TrainingSession` soll sicherstellen, dass alle über `Appointments` mit dieser Instanz verbundenen Dozenten niemals auch zeitgleich Teilnehmer dieser Schulung sind.

Die Tests für diese Invarianten befinden sich in Abschnitt 5.11 auf Seite 75.

3.5.14. Appointment::valuesDefined

Die Invariante *valuesDefined* der Klasse `Appointment` stellt sicher, dass alle benötigten Klassenvariablen (`date` sowie `time`) definiert wurden und sich keine dieser Variablen im Zustand *undefined* befindet.

Die Tests für diese Invarianten befinden sich in Abschnitt 5.23 auf Seite 86.

3.5.15. Appointment::instructorHasKnowledge

Die Invariante *instructorHasKnowledge* der Klasse `Appointment` soll sicherstellen, dass der für dieses `Appointment` angegebenen Dozent die erforderliche Qualifikation für das zu schulende `Tool` besitzt und somit befähigt ist die Schulung zu geben.

Die Tests für diese Invarianten befinden sich in Abschnitt 5.12 auf Seite 76.

3.5.16. Appointment::enoughSeats

Die Invariante *enoughSeats* der Klasse `Appointment` soll sicherstellen, dass der für dieses `Appointment` angegebenen Raum über genügend Sitze für die Schulung verfügt, d.h. die Anzahl der Sitze im angegebenen Raum (Instanz der Klasse `Room`) muss größer oder gleich der Anzahl der registrierten Schulungsteilnehmer sein, die für diese Schulung akzeptiert worden sind.

Die Tests für diese Invarianten befinden sich in Abschnitt 5.13 auf Seite 77.

3.5.17. Room::valuesDefined

Die Invariante *valuesDefined* der Klasse `Room` stellt sicher, dass alle benötigten Klassenvariablen (`address`, `seats`, `internal` und `cost`) definiert wurden und sich keine dieser Variablen im Zustand *undefined* befindet.

Die Tests für diese Invarianten befinden sich in Abschnitt 5.23 auf Seite 86.

3.5.18. Room::costGTEZero

Die Invariante *costGTEZero* der Klasse `Room` soll dafür sorgen, dass die Kosten, die für diesen Raum angegeben wurden immer größer oder gleich Null sind und somit niemals negative Kosten für einen Raum angegeben werden können.

Die Tests für diese Invarianten befinden sich in Abschnitt 5.14 auf Seite 78.

3.5.19. Room::noDoubleOccupancy

Die Invariante *noDoubleOccupancy* der Klasse `Room` soll dafür sorgen, dass ein Raum niemals zeitgleich, dass heißt am selben Tag innerhalb desselben Zeitraums, von zwei oder mehr `Appointments` genutzt wird.

Die Tests für diese Invarianten befinden sich in Abschnitt 5.15 auf Seite 79.

3.5.20. Catering::valuesDefined

Die Invariante *valuesDefined* der Klasse `Catering` stellt sicher, dass alle benötigten Klassenvariablen (`description` und `cost`) definiert wurden und sich keine dieser Variablen im Zustand *undefined* befindet.

Die Tests für diese Invarianten befinden sich in Abschnitt 5.23 auf Seite 86.

3.5.21. Catering::costGTEZero

Die Invariante *costGTEZero* der Klasse `Catering` soll dafür sorgen, dass die Kosten, die für ein `Catering` angegeben wurden immer größer oder gleich Null sind und somit niemals negative Kosten für eine Bewirtung angegeben werden können.

Die Tests für diese Invarianten befinden sich in Abschnitt 5.16 auf Seite 80.

3.5.22. Tool::valuesDefined

Die Invariante *valuesDefined* der Klasse `Tool` stellt sicher, dass alle benötigten Klassenvariablen (in diesem Fall gibt es nur die Klassenvariable `name`)) definiert wurden und sich diese Variablen niemals im Zustand *undefined* befindet.

Die Tests für diese Invarianten befinden sich in Abschnitt 5.23 auf Seite 86.

3.5.23. Document::valuesDefined

Die Invariante *valuesDefined* der Klasse `Document` stellt sicher, dass alle benötigten Klassenvariablen (in diesem Fall gibt es nur die Klassenvariable `title`)) definiert wurden und sich diese Variablen niemals im Zustand *undefined* befindet.

Die Tests für diese Invarianten befinden sich in Abschnitt 5.23 auf Seite 86.

3.5.24. Registration::valuesDefined

Die Invariante *valuesDefined* der Klasse `Registration` stellt sicher, dass alle benötigten Klassenvariablen (in diesem Fall gibt es nur die Klassenvariable `accepted`)) definiert wurden und sich diese Variablen niemals im Zustand *undefined* befindet.

Die Tests für diese Invarianten befinden sich in Abschnitt 5.23 auf Seite 86.

3.5.25. Bill::valuesDefined

Die Invariante *valuesDefined* der Klasse `Bill` stellt sicher, dass alle benötigten Klassenvariablen (`price` und `paid`) definiert wurden und sich keine dieser Variablen im Zustand *undefined* befindet.

Die Tests für diese Invarianten befinden sich in Abschnitt 5.23 auf Seite 86.

3.5.26. Bill::priceGTEZero

Die Invariante *priceGTEZero* der Klasse `Bill` soll dafür sorgen, dass die Kosten, die für ein `Schulung` anfallen und dem Teilnehmer in Rechnung gestellt wurden, immer größer oder gleich Null sind und somit niemals negative Kosten für eine Schulung angegeben werden können.

Die Tests für diese Invarianten befinden sich in Abschnitt 5.17 auf Seite 81.

3.5.27. Bill::participantAccepted

Die Invariante *participantAccepted* der Klasse `Bill` soll dafür sorgen, dass eine Rechnung für einen Schulungsteilnehmer erst dann ausgestellt werden kann, wenn dieser für eine Schulung auch (verbindlich) akzeptiert wurde.

Die Tests für diese Invarianten befinden sich in Abschnitt 5.18 auf Seite 82.

3.5.28. Teaches::valuesDefined

Die Invariante *valuesDefined* der Klasse `Teaches` stellt sicher, dass alle benötigten Klassenvariablen (in diesem Fall gibt es nur eine Klassenvariable `cost`) definiert wurden und sich keine dieser Variablen im Zustand *undefined* befindet.

Die Tests für diese Invarianten befinden sich in Abschnitt 5.23 auf Seite 86.

3.5.29. Teaches::costGTEZero

Die Invariante *costGTEZero* der Klasse `Teaches` soll dafür sorgen, dass die Kosten, die durch das Gehalt eines Dozenten bestimmt wurden immer größer oder gleich Null sind und somit niemals negative Kosten angegeben werden können.

Die Tests für diese Invarianten befinden sich in Abschnitt 5.19 auf Seite 83.

3.5.30. Date::valuesDefined

Die Invariante *valuesDefined* der Klasse `Date` stellt sicher, dass alle benötigten Klassenvariablen (`day`, `month` und `year`) definiert wurden und sich keine dieser Variablen im Zustand *undefined* befindet.

Die Tests für diese Invarianten befinden sich in Abschnitt 5.23 auf Seite 86.

3.5.31. Date::correctDate

Die Invariante *correctDate* der Klasse `Date` soll sicherstellen, dass es sich bei dem von dieser Instanz repräsentierten Datum tatsächlich um ein gültiges Datum im Sinne des gregorianischen Kalenders handelt. Einen detaillierten Einblick in diese Berechnung bekommt man im Listing der Klasse, dass sich im Anhang dieses Dokuments befindet.

Die Tests für diese Invarianten befinden sich in Abschnitt 5.20 auf Seite 84.

3.5.32. Date::dateUnique

Die Invariante *dateUnique* der Klasse `Date` soll sicherstellen, dass es sich bei dem von dieser Instanz repräsentierten Datum um ein Datum handelt, dass von noch keiner anderen Instanz im System repräsentiert wird. Somit kann jedes Datum immer nur genau von einer Instanz der Klasse `Date` berücksicht werden und niemals zweimal definiert sein.

Die Tests für diese Invarianten befinden sich in Abschnitt 5.21 auf Seite 85.

3.5.33. Period::valuesDefined

Die Invariante *valuesDefined* der Klasse `Period` stellt sicher, dass alle benötigten Klassenvariablen (`start` und `ends`) definiert wurden und sich keine dieser Variablen im Zustand *undefined* befindet.

Die Tests für diese Invarianten befinden sich in Abschnitt 5.23 auf Seite 86.

3.5.34. Period::positiveTimeFrame

Die Invariante *positiveTimeFrame* der Klasse `Period` stellt sicher, dass es sich beim Zeitrahmen, der durch diese Instanz von `Period` repräsentiert wird, um einen positiven Zeitrahmen handelt, dies bedeutet, dass der Start- und Endpunkt gleich definiert sind, oder dass der Startpunkt zeitlich gesehen vor dem Endpunkt liegt und niemals umgekehrt.

Die Tests für diese Invarianten befinden sich in Abschnitt 5.22 auf Seite 85

4. Objektdiagramm

Als Vorbereitung für die Erstellung von Testfällen und Anfragen, welche in den nachfolgenden Kapiteln vorgestellt werden, haben wir ein umfangreiches Objektdiagramm erstellt, welches alle Invarianten erfüllt und mit 292 Objekten (Abbildung 4.1) und 289 Links und Linkobjekten (Abbildung 4.2) vielfältige Testdaten in allen Bereichen des Modells bereitstellt.

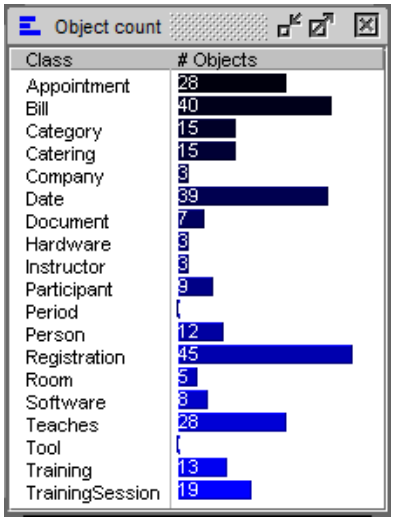

Abbildung 4.1.: Anzahl der Objekte

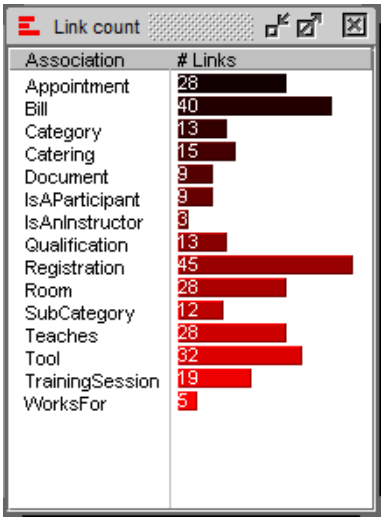

Abbildung 4.2.: Anzahl der Links

Der Aufbau des Objektdiagramms wurde sinngemäß nach den Klassen des Modells in Dateien unterteilt. Diese Dateien befinden sich im Unterordner `intern/` und deren Name beginnt mit `model-*`. Um das gesamte Objektdiagram zu laden, genügt es die Datei `intern/model-base.cmd` zu öffnen, welche alle anderen Dateien in der entsprechenden Reihenfolge lädt.

Auf diese Weise haben wir auch den folgenden kleinen Ausschnitt unseres Objektdiagramms erstellt, der in der Abbildung 4.3 auf der nächsten Seite zu sehen ist und den wir nun einmal vorstellen möchten. Zur besseren Überschaubarkeit kommt in diesem Ausschnitt von jedem Objekttyp meist nur eine Instanz vor.

Zentraler Ausgangspunkt ist das Training `miw`, welches zur Unterkategorie `word` in der Kategorie `office` gehört, wobei die letztgenannte noch drei weitere Unterkaegorien beinhaltet: `access`, `excel` und `visio`. Das Dokument `miwHandout` sowie die

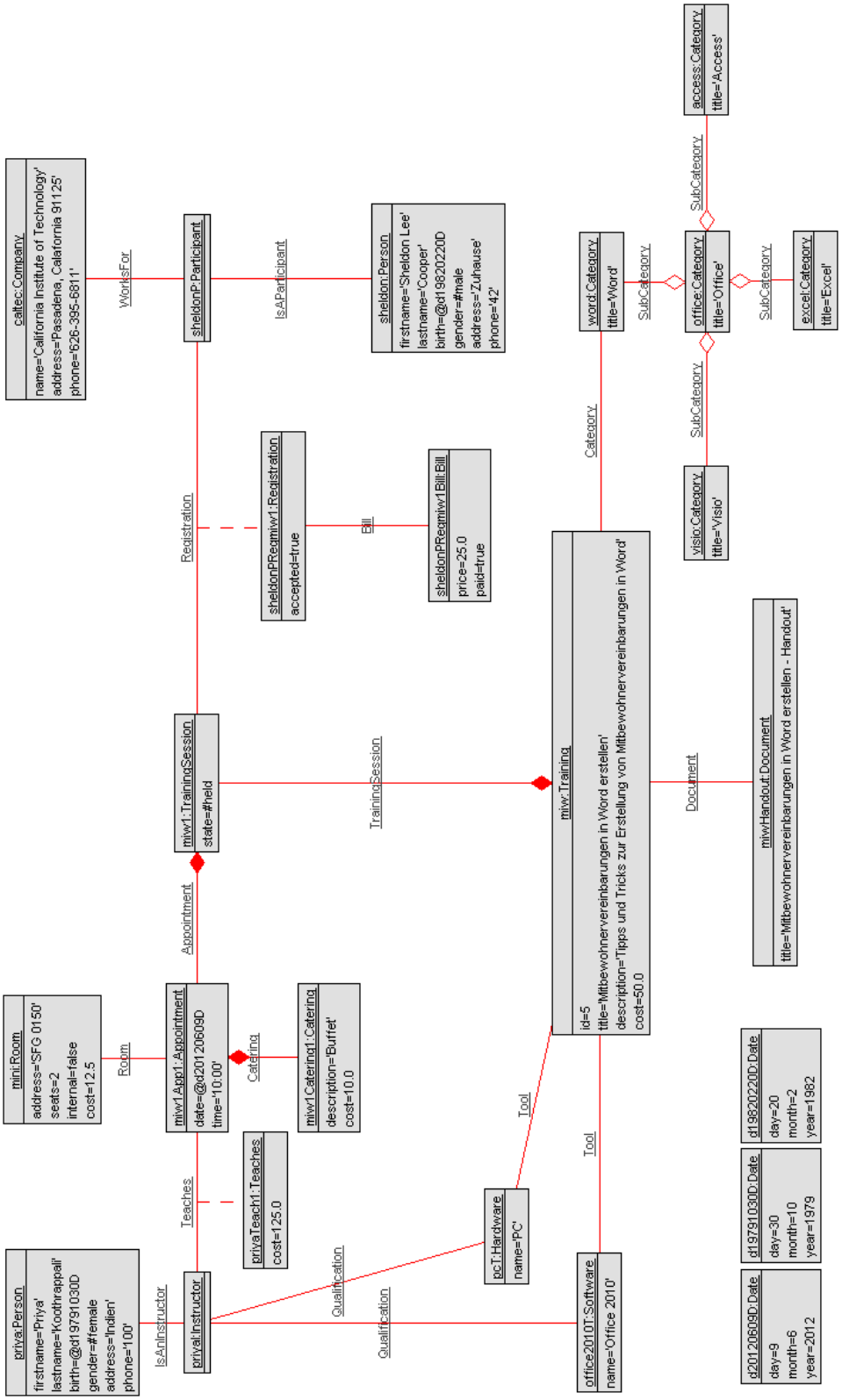

Abbildung 4.3.: Ausschnitt des Objektdiagramms

Trainingssession `miw1`, die breits stattgefunden hat, sind dem Training `miw` zugeordnet. Für `miw1` hat sich der Teilnehmer `sheldonP` angemeldet, dessen Registrierung akzeptiert wurde und der die Rechnung bereits bezahlt hat. Der Teilnehmer `sheldonP` arbeitet für die Firma `caltec`. Hinter `sheldonP` verbirgt sich die Person `sheldon`. Für die Trainingssession `miw1` ist der Termin `miw1App1` festgelegt. Für diesen Termin sind wiederum der Raum `mini` und das Catering `miw1Catering1` gebucht. Die Person `priya` ist Dozentin `priyaI`, die den Termin `miw1App1` unterrichtet hat und über ihre Qualifikationen `pcT` und `office2012T` wiederum dem Training `miw` als geeignet zugeordnet ist. Die Geburtstage der Personen und der Tag des Termins werden durch die Objekte der Klasse Date `d19820220`, `d19791030` und `d20120609D` repräsentiet.

Dieser Ausschnitt deckt alle Bereiche unseres Modells ab, schöpf jedoch noch nicht alle Möglichkeiten und Kombinationen aus, die unser System vorsieht. Hierfür haben wir in unserem Objektdiagramm weitere Beispiele, die von Dozenten ohne Qualifikation über unbezahlte Rechnung bis zu voll besetzten Trainingssessions reichen.

5. Testfälle

In diesem Kapitel geht es um Testfälle auf Invarianten. Eine Invariante ist ein boolscher Ausdruck der immer *true* sein muss. Um dies zu überprüfen wird zu jeder Invariante ein Testfall konstruiert. Um eine graphische Übersicht über den Ablauf des Testfalls zu bekommen wird neben dem Quellcode zu jedem Testfall ein Objektdiagramm erstellt.

5.1. Testfall 01 (TC_01_validSetup.cmd)

In diesem Testfall wird überprüft ob ein gültiges Model aufgebaut wird. Dazu laden wir das Skript *model-base.cmd*, welches wiederum alle anderen Skripte liest um das Model aufzubauen.

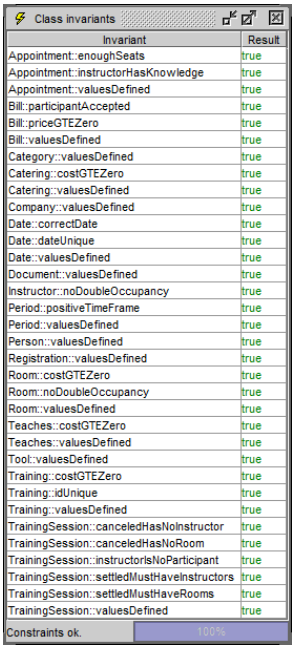

Abbildung 5.1.: vom Modell erfüllte Invarianten

```
1 open ../intern/model-base.cmd
```

Quellcode 5.1: TC_01_validSetup.cmd

5.2. Testfall 02 (TC_02_subcategoryOfSelf.cmd)

In diesem Testfall wird überprüft ob eine Kategorie selbst wieder eine Unterkategorie sein kann. Wir erstellen eine neue Kategorie und setzen für die eben erzeugte Kategorie den Titel auf *cat*. Nun wird die eben erzeugte Kategorie selbst wieder eine Unterkategorie sein. Dies führt zu einem Fehler, da eine Kategorie nicht selbst wieder eine Unterkategorie von sich sein kann.

```
1 !create cat : Category
2 !set cat.title := 'cat1'
3
4 !insert (cat, cat) into SubCategory
```

Quellcode 5.2: TC_02_subcategoryOfSelf.cmd

Abbildung 5.2.: TC_02_subcategoryOfSelf.cmd

5.3. Testfall 03 (TC_03_subcategoryOfSelfDeep.cmd)

In diesem Testfall werden zwei Kategorien erstellt wobei eine Kategorie hier jeweils die Unterkategorie der Anderen sein soll. Wir erzeugen zwei Kategorien mit dem Namen *cat* und *cat*. Dann Fügen wir die Kategorien so ein, dass die eine Kategorie jeweils die Unterkategorie der Anderen ist. Dies führt zu einem Fehler, da zwei verschiedene Kategorien nicht Unterkategorien der jeweils Anderen sein können.

```
1 !create cat : Category
2 !set cat.title := 'cat1'
3
4 !create cat2 : Category
5 !set cat2.title := 'cat2'
6
7 !insert (cat, cat2) into SubCategory
8 !insert (cat2, cat) into SubCategory
```

Quellcode 5.3: TC_03_subcategoryOfSelfDeep.cmd

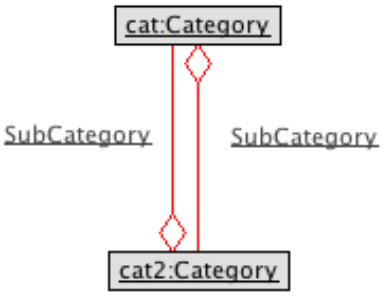

Abbildung 5.3.: TC_03_subcategoryOfSelfDeep.cmd

5.4. Testfall 04 (TC_04_instructorNoDoubleOccupancy.cmd)

In diesem Testfall wird getestet, ob ein Ausbilder zwei Termine zur selben Zeit haben kann. Dazu wird ein neuer Termin erstellt, der das gleiche Datum und die gleiche Uhrzeit besitzt wie ein schon vorhandener Termin. Schließlich bekommt ein Ausbilder eine neue Trainingseinheit mit dem eben erstellten Termin. Da der Ausbilder zu dem eben erstellten Termin bereits einen Termin hat führt dies zu einem Fehler und somit zu einer Verletzung der Invariante.

```
1  open ../intern/model-base.cmd
2
3  !create testApp : Appointment
4  !create testAppD : Date
5  !set testAppD.day := 7
6  !set testAppD.month := 4
7  !set testAppD.year := 2012
8  !set testApp.date := testAppD
9  !set testApp.time := '10:00'
10 !insert (bfu1, testApp) into Appointment
11
12 !create drGabelhauserTeach28 : Teaches between (drGablehauserI, testApp)
```

Quellcode 5.4: TC_04_instructorNoDoubleOccupancy.cmd

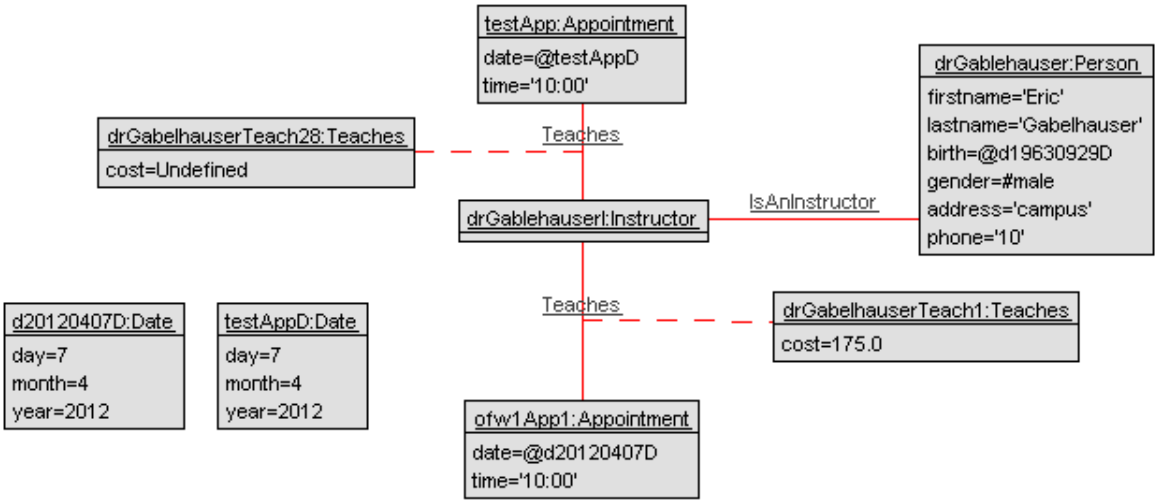

Abbildung 5.4.: TC_04_instructorNoDoubleOccupancy.cmd

5.5. Testfall 05 (TC_05_trainingsCostsGreaterEqZero.cmd)

In diesem Testfall werden die Kosten für einige Trainingseinheiten verändert. Wir setzen die Kosten für die Trainingseinheiten:

- Office für Wissenschaftler auf -5
- Access für Neurobiologen auf -12
- Excel für Biologen auf -9

Da die Kosten hierbei größer gleich 0 sein müssen, werden für die folgenden Manipulationen Fehler geworfen.

```
1  open ../intern/model-base.cmd
2
3  -- manipulating costs
4  -- Office fuer Wissenschaftler
5  !set ofw.cost := -5.0
6  -- Access fuer Neurobiologen
7  !set afn.cost := -12.0
8  -- Excel fuer Biologen
9  !set efb.cost := -9.0
```

Quellcode 5.5: TC_05_trainingsCostsGreaterEqZero.cmd

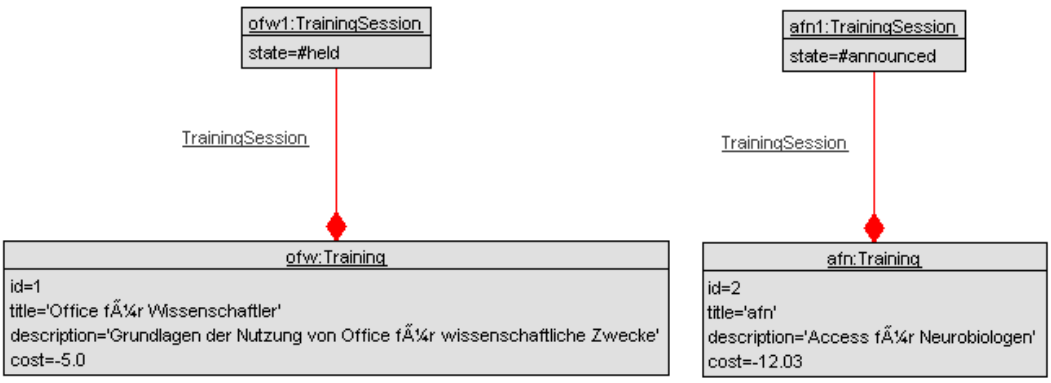

Abbildung 5.5.: TC_05_trainingsCostsGreaterEqZero.cmd

5.6. Testfall 06 (TC_06_trainingsIdIsUnique.cmd)

In diesem Testfall wird überprüft ob jede Trainingseinheit eine eindeutige zugewiesene ID besitzt. Um dies zu überprüfen wird die ID einer zuvor definierten Trainingseinheit manipuliert. Die ID der Trainingseinheit *miw* wird auf 1 gesetzt. Da nun die Trainingseinheiten *ofw* und *miw* die gleiche ID besitzen führt dies zu einer Verletzung der Invariante.

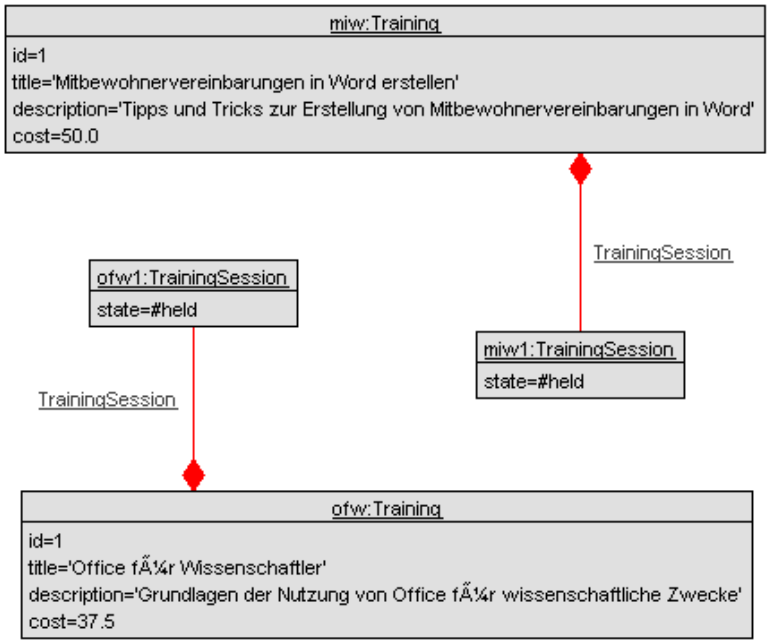

Abbildung 5.6.: TC_06_trainingsIdIsUnique.cmd

```
1  open ../intern/model-base.cmd
2
3  !set miw.id := 1
```

Quellcode 5.6: TC_06_trainingsIdIsUnique.cmd

5.7. Testfall 07 (TC_07_settledHaveRoom.cmd)

In diesem Testfall geht es um eine festgelegte Trainingssitzung. Eine festgelegte Sitzung muss auch einen Raum besitzen. Um dies zu testen erstellen wir einen neuen Termin und weisen diesen neuen Termin einem Ausbilder zu. Somit hat dieser Termin zwar einen Ausbilder aber keinen Raum und führt somit zu einem Fehler bzw. zu einer Verletzung der Invariante.

```
1  open ../intern/model-base.cmd
2
3  !create iDontKnowApp : Appointment
4  !create iDontKnowAppD : Date
5  !set iDontKnowAppD.day := 7
6  !set iDontKnowAppD.month := 4
7  !set iDontKnowAppD.year := 2012
8  !set iDontKnowApp.date := iDontKnowAppD
9  !set iDontKnowApp.time := '10:00'
10 !insert (ofw1, iDontKnowApp) into Appointment
11
12 !create pennyTeach1 : Teaches between (pennyI, iDontKnowApp)
```

Quellcode 5.7: TC_07_settledHaveRoom.cmd

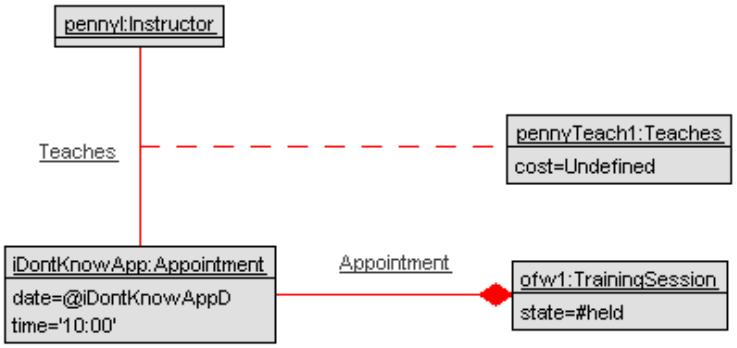

Abbildung 5.7.: TC_07_settledHaveRoom.cmd

5.8. Testfall 08 (TC_08_settledHaveInstructor.cmd)

In diesem Testfall geht es um eine festgelegte Trainingssitzung. Eine festgelegte Sitzung muss auch einen Ausbilder haben. Um dies zu testen erstellen wir einen neuen Termin und weisen diesen Termin keinen Ausbilder zu. Da der Termin keinem Ausbilder zugewiesen ist führt dies zu einer Fehlermeldung und somit zur Verletzung einer Invariante.

```
1  open ../intern/model-base.cmd
2
3  !create iDontKnowApp : Appointment
4  !create iDontKnowAppD : Date
5  !set iDontKnowAppD.day := 7
6  !set iDontKnowAppD.month := 4
7  !set iDontKnowAppD.year := 2012
8  !set iDontKnowApp.date := iDontKnowAppD
9  !set iDontKnowApp.time := '10:00'
10 !insert (ofw1, iDontKnowApp) into Appointment
```

Quellcode 5.8: TC_08_settledHaveInstructor.cmd

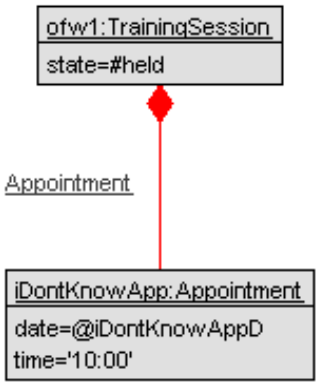

Abbildung 5.8.: TC_08_settledHaveInstructor.cmd

5.9. Testfall 09 (TC_09_canceledHasNoRoom.cmd)

In diesem Testfall geht es um eine abgesagte Trainingssitzung. Eine abgesagte Sitzung darf keinen Raum in Anspruch nehmen, da diese Sitzung nicht mehr stattfinden wird. Um dies zu testen setzen wir eine bereits anstehende Sitzung auf *#canceled* und

bekommen somit eine Verletzung der Invariante, da diese Sitzung immernoch einen Raum in Anspruch nimmt.

```
1 open ../intern/model-base.cmd
2
3 !set ofw1.state := #canceled
```

Quellcode 5.9: TC_09_canceledHasNoRoom.cmd

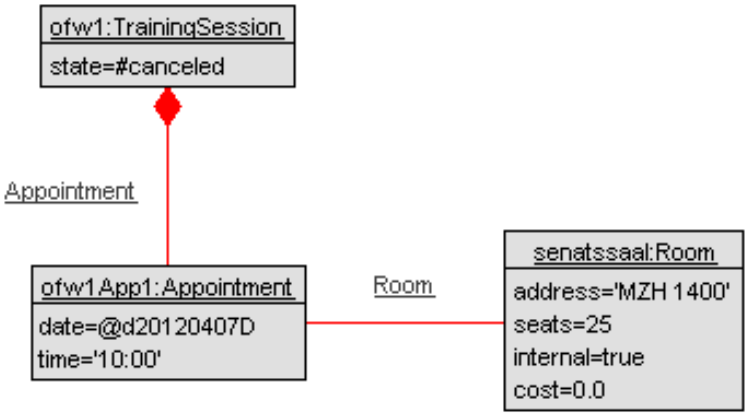

Abbildung 5.9.: TC_09_canceledHasNoRoom.cmd

5.10. Testfall 10 (TC_10_canceledHasNoInstructor.cmd)

In diesem Testfall geht es um eine abgesagt Trainingssitzung. Eine abgesagte Sitzung darf keinem Ausbilder zugeordnet sein, da diese Sitzung nicht mehr stattfinden wird. Um dies zu testen setzen wir eine bereits anstehende Sitzung auf *#canceled* und bekommen somit eine Verletzung der Invariante, da diese Sitzung immernoch einem Ausbilder zugewiesen ist.

```
1 open ../intern/model-base.cmd
2
3 !set ofw1.state := #canceled
```

Quellcode 5.10: TC_10_canceledHasNoInstructor.cmd

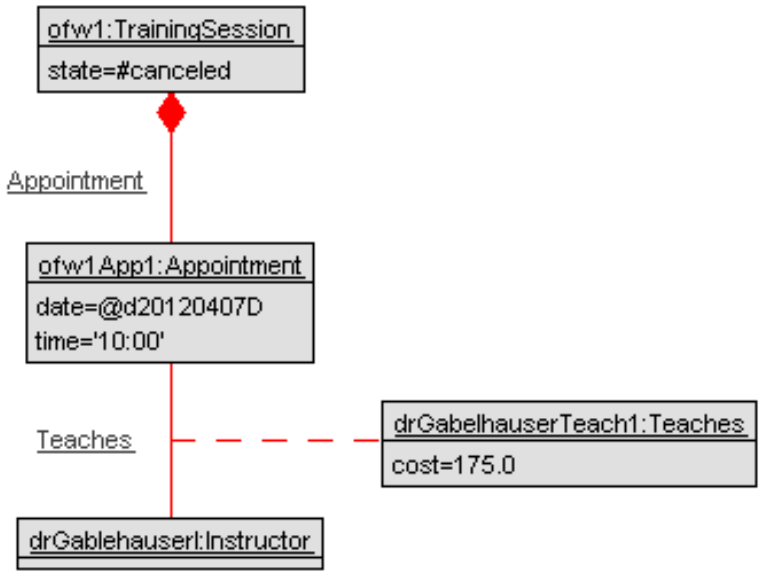

Abbildung 5.10.: TC_10_canceledHasNoInstructor.cmd

5.11. Testfall 11 (TC_11_instructorIsNoParticipant.cmd)

In diesem Testfall wird überprüft ob ein Ausbilder zugleich Teilnehmer einer Sitzung sein darf. Dazu erstellen wir für einen Ausbilder eine neue Teilnahme an einer Sitzung und akzeptieren diese. Somit ist der Ausbilder drGabelhauser auch Teilnehmer seiner Sitzung. Dies führt allerdings zu einer Verletzung der Invariante, da ein Ausbilder nicht Teilnehmer seiner eigenen Sitzung sein kann.

```
1 open ../intern/model-base.cmd
2
3 !create drGablehauserP : Participant
4 !insert (drGablehauser, drGablehauserP) into IsAParticipant
5 !create drGablehauserPRegofw1 : Registration between (ofw1,
    drGablehauserP)
6 !set drGablehauserPRegofw1.accepted := true
```

Quellcode 5.11: TC_11_instructorIsNoParticipant.cmd

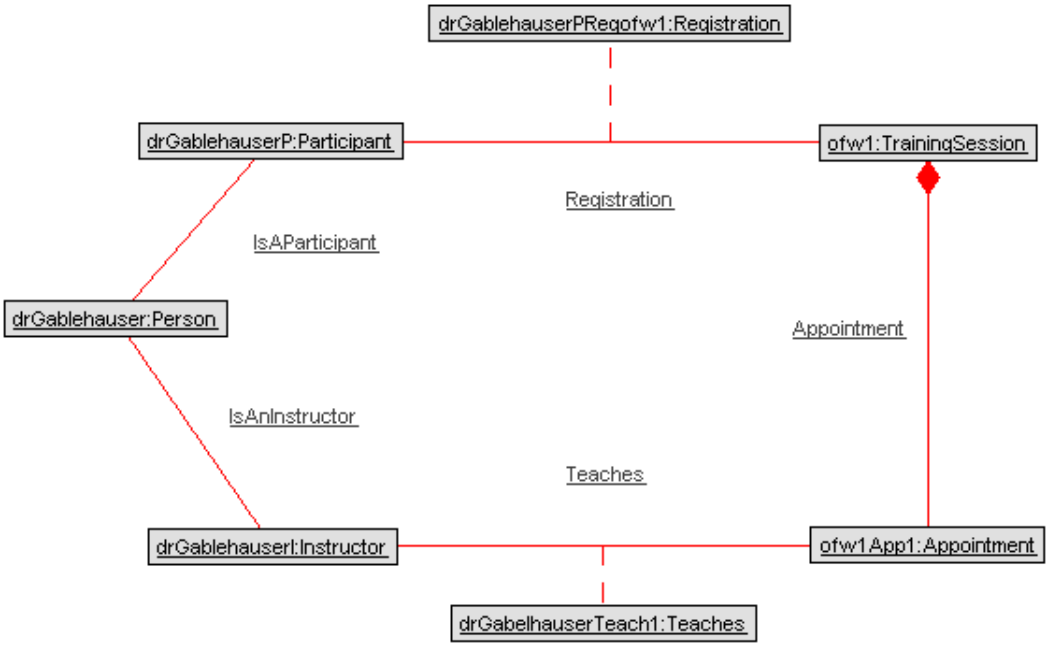

Abbildung 5.11.: TC_11_instructorIsNoParticipant.cmd

5.12. Testfall 12 (TC_12_instructorHasNoKnowledge.cmd)

In diesem Testfall geht es darum, dass ein Ausbilder eine gewisse Qualifikation zu einem bestimmten Tool haben muss, um eine Trainingssitzung halten zu können. Um ein wenig Kenntnis über diesen Ausbilder zu erlangen gibt es einen kleinen Ausschnitt aus der **model-instructors.cmd**, in der eine Person mit dem Namen *penny* zu einem Ausbilder wird:

```
1 !create pennyI : Instructor
2 !insert (penny, pennyI) into IsAnInstructor
```

Anhand dieses Ausschnittes können wir sehen, dass die Ausbilderin *penny* zwar Ausbilderin ist, aber über keine Qualifikation verfügt. Nun wird ein neuer Termin erstellt und an penny übergeben. Dies führt zu einer Verletzung der Invariante, da die Ausbilderin einen Termin übergeben bekommt ohne jegliche Kenntnisse oder Qualifikationen.

```
1  open ../intern/model-base.cmd
2
3  !create iDontKnowApp : Appointment
4  !create iDontKnowAppD : Date
5  !set iDontKnowAppD.day := 7
6  !set iDontKnowAppD.month := 4
7  !set iDontKnowAppD.year := 2012
8  !set iDontKnowApp.date := iDontKnowAppD
9  !set iDontKnowApp.time := '10:00'
10 !insert (ofw1, iDontKnowApp) into Appointment
11
12 !create pennyTeach1 : Teaches between (pennyI, iDontKnowApp)
```

Quellcode 5.12: TC_12_instructorHasNoKnowledge.cmd

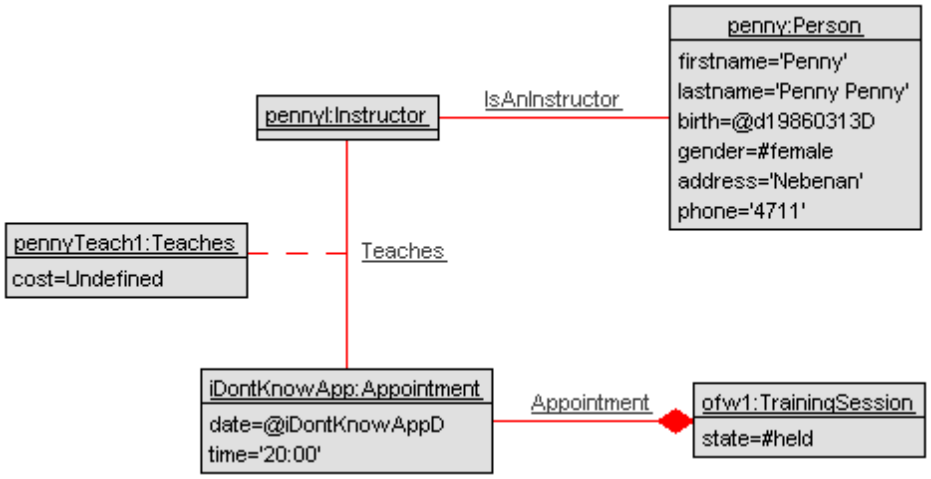

Abbildung 5.12.: TC_12_instructorHasNoKnowledge.cmd

5.13. Testfall 13 (TC_13_enouthSeats.cmd)

In diesem Testfall geht es um die verfügbaren Sitzplätze eines Raumes, in dem eine Sitzung stattfinden wird. Es muss immer sicher gestellt werden, dass in einer Trainings-Sitzung genug Plätze für alle teilnehmenden Personen verfügbar sind. Um dies zu testen, setzen wir die Anzahl der Sitzplätze des Senatssaals auf 2. Im Senatssaal findet die Trainings-Sitzung *ofw1* statt, die zu einem Termin *ofw1App1* zugeordnet ist. Laut Registrierung sind 8 Teilnehmer für diese Sitzung eingetragen. Da der Senatssaal mit 25 Sitzplätzen vordefiniert ist, verursacht die Minderung auf 2 Sitzplätze eine Verletzung der Invariante.

```
1 open ../intern/model-base.cmd
2
3 !set senatssaal.seats := 2
```

Quellcode 5.13: TC_13_enouthSeats.cmd

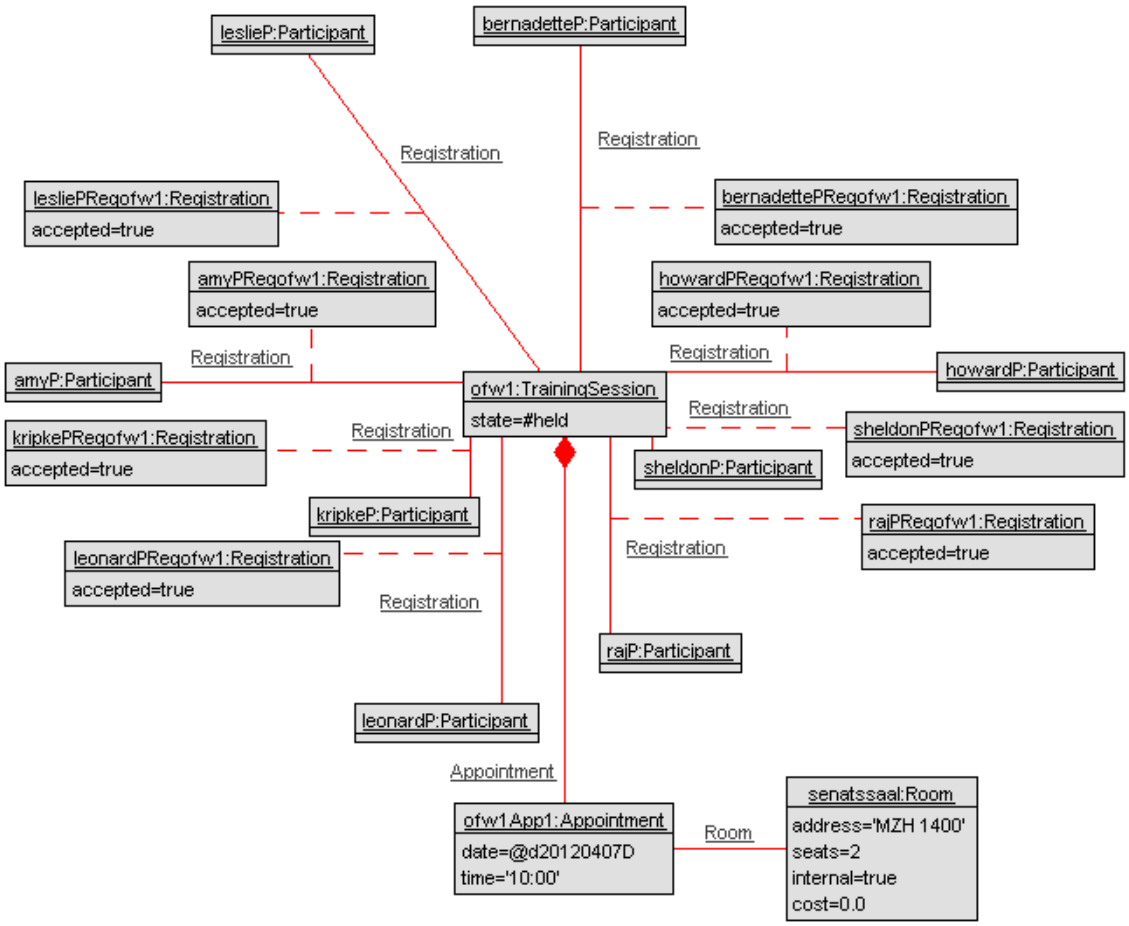

Abbildung 5.13.: TC_13_enouthSeats.cmd

5.14. Testfall 14 (TC_14_roomCostsGreaterEqZero.cmd)

In diesem Testfall werden die Kosten für einige Räume verändert. Wir setzen die Kosten für die Räume:

- Battlefield auf -50

- Playground auf -12
- Wiisportsresort auf -2.5

Da die Kosten hierbei größer gleich 0 sein müssen, werden für die folgenden Manipulationen Fehler geworfen.

```
1 open ../intern/model-base.cmd
2
3 -- manipulating costs
4 -- Battlefield room costs
5 !set battlefield.cost := -50.00
6 -- Playground room costs
7 !set playground.costs := -12.00
8 -- Wiisportsresort room costs
9 !set wiisportsresort.cost := -2.5
```

Quellcode 5.14: TC_14_roomCostsGreaterEqZero.cmd

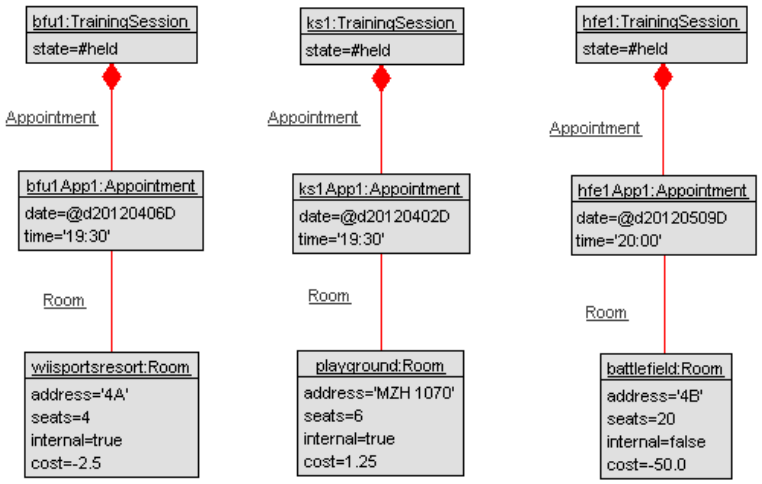

Abbildung 5.14.: TC_14_roomCostsGreaterEqZero.cmd

5.15. Testfall 15 (TC_15_roomNoDoubleOccupancy.cmd)

In diesem Testfall geht es um die Aufteilung der Räume. Es ist nicht möglich, dass zwei Termine, die zur selben Zeit stattfinden, an den selben Raum vermietet werden. Um dies zu testen wird die Zeit eines anderen Termins manipuliert. Das Datum des Termins *miw1App1* bekommt das selbe Datum zugewiesen an dem eigentlich der Termin *ofw1App1* stattfinden würde. Da der Termin *ofw1App1* um 10:00 stattfindet, wird

die Zeit des Termins m1w1App1 auf 10:00 gesetzt. Schließlich bekommt miw1App1 den Senatssaal zugewiesen. Beide Termine finden nun zur selben Zeit im Senaatssaal statt und führen somit zur Verletzung der Invariante.

```
1 open ../intern/model-base.cmd
2
3 !set miw1App1.date := d20120407D
4 !set miw1App1.time := '10:00'
5 !insert (senatssaal, miw1App1) into Room
```

Quellcode 5.15: TC_15_roomNoDoubleOccupancy.cmd

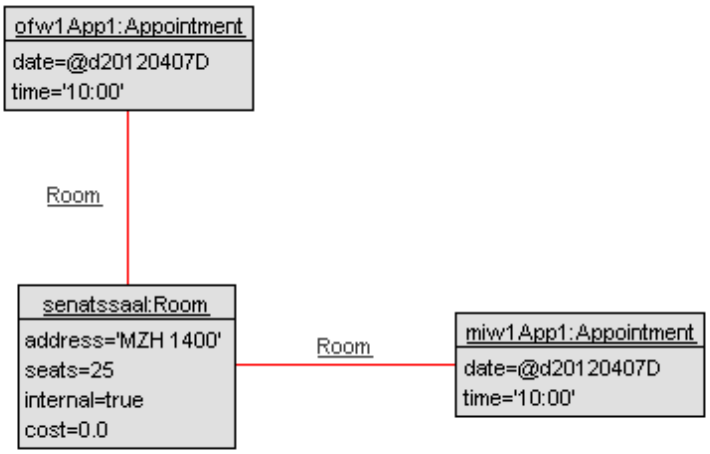

Abbildung 5.15.: TC_15_roomNoDoubleOccupancy.cmd

5.16. Testfall 16 (TC_16_cateringCostsGreaterEqZero.cmd)

In diesem Testfall werden die Kosten für Caterings verändert. Wir setzen die Kosten für die Caterings:

- hfn1Catering1 auf -8
- hfn2Catering4 auf -12
- hfe1Catering1 auf -9.5

Da die Kosten hierbei größer gleich 0 sein müssen, werden für die folgenden Manipulationen Fehler geworfen.

```
1  open ../intern/model-base.cmd
2
3  -- manipulating costs
4  -- Catering hfn1Catering1 costs
5  !set hfn1Catering1.cost := -8.0
6  -- Catering hfn2Catering4 costs
7  !set hfn2Catering4.cost := -12.0
8  -- Catering hfe1Catering1 costs
9  !set hfe1Catering1.cost := -9.5
```

Quellcode 5.16: TC_16_cateringCostsGreaterEqZero.cmd

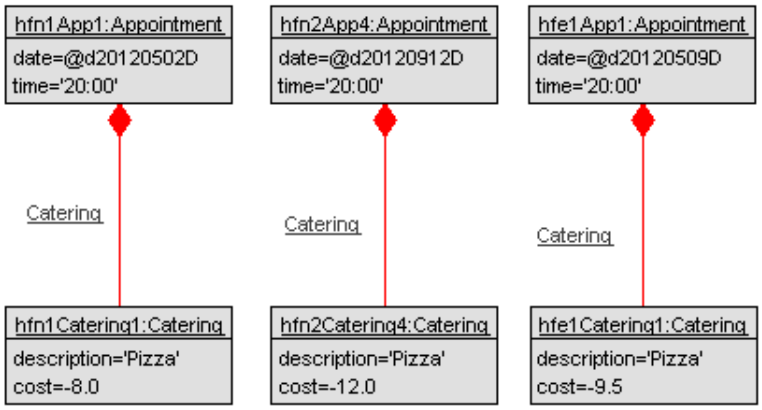

Abbildung 5.16.: TC_16_cateringCostsGreaterEqZero.cmd

5.17. Testfall 17 (TC_17_billPriceGreaterEqZero.cmd)

In diesem Testfall werden die Preise für Registrierung verändert. Wir setzen die Preise für die Registrierungen:

- howardPRegofw1Bill auf -12
- lesliePRegofw1Bill auf -18.5
- rajPReghfn1Bill auf -75

Da die Preise hierbei größer gleich 0 sein müssen, werden für die folgenden Manipulationen Fehler geworfen.

```
1 open ../intern/model-base.cmd
2
3 -- manipulating price
4 -- Bill howardPRegofw1Bill price
5 !set howardPRegofw1Bill.price := -12
6 -- Bill lesliePRegofw1Bill price
7 !set lesliePRegofw1Bill.price := -18.50
8 -- Bill rajPReghfn1Bill price
9 !set rajPReghfn1Bill.price := -75
```

Quellcode 5.17: TC_17_billPriceGreaterEqZero.cmd

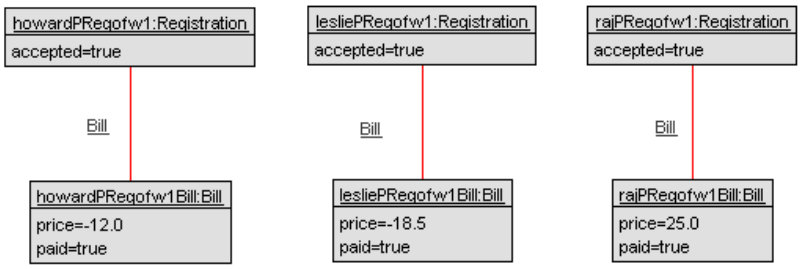

Abbildung 5.17.: TC_17_billPriceGreaterEqZero.cmd

5.18. Testfall 18 (TC_18_billParticipantAccepted.cmd)

In diesem Testfall wird überprüft, dass einem Schulungsteilnehmer erst dann eine Rechnung ausgestellt werden kann, wenn dieser für eine Schulung auch akzeptiert wurde. Um dies zu überprüfen wird die Registrierung der Person *raj* nicht akzeptiert und auf false gesetzt. Dies führt zu einer verletzung der Invariante.

```
1 open ../intern/model-base.cmd
2
3 !set rajPRegofw1.accepted := false
```

Quellcode 5.18: TC_18_billParticipantAccepted.cmd

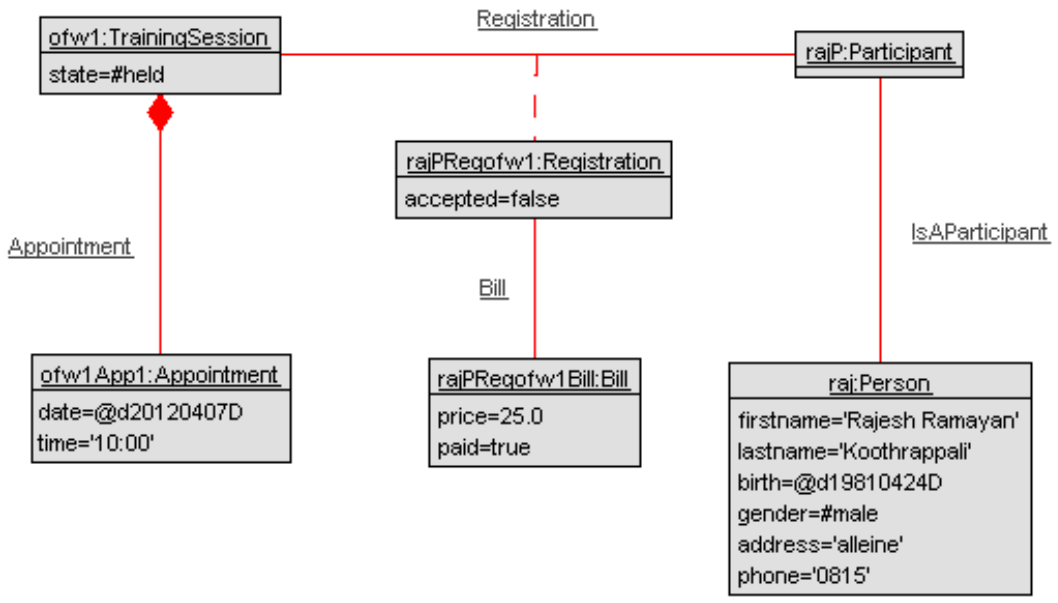

Abbildung 5.18.: TC_18_billParticipantAccepted.cmd

5.19. Testfall 19 (TC_19_teachesCostGreaterEqZero.cmd)

In diesem Testfall werden die Kosten für Trainings-Sitzungen verändert. Wir setzen die Kosten für die Trainingssitzungen:

- drGabelhauserTeach1l auf -200
- drGabelhauserTeach19 auf -28.5
- drGabelhauserTeach27l auf -600

Da die Preise hierbei größer gleich 0 sein müssen, werden für die folgenden Manipulationen Fehler geworfen.

```
1 open ../intern/model-base.cmd
2
3 -- manipulating costs
4 -- Instructor drGabelhauserTeach1 costs
5 !set drGabelhauserTeach1.cost := -200.00
6 -- Instructor drGabelhauserTeach19 costs
7 !set drGabelhauserTeach19.cost := -28.50
8 -- Instructor drGabelhauserTeach27 costs
9 !set drGabelhauserTeach27.cost := -600.00
```

Quellcode 5.19: TC_19_teachesCostGreaterEqZero.cmd

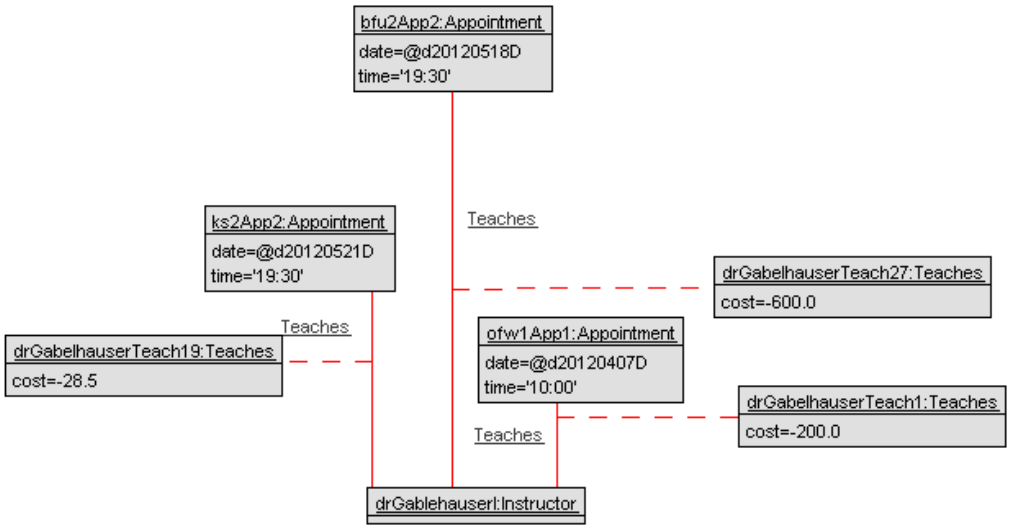

Abbildung 5.19.: TC_19_teachesCostGreaterEqZero.cmd

5.20. Testfall 20 (TC_20_correctDate.cmd)

In diesem Testfall wird überprüft ob es sich bei einem Datum auch wirklich um ein gültiges Datum handelt. Um dies zu überprüfen setzen wir den Tag an dem ein Termin stattfinden soll auf -2. Da ein Tag größer gleich null und kleiner gleich 31 sein muss, wird in diesem Testfall die Invariante verletzt.

```
1 open ../intern/model-base.cmd
2
3 !set d20120407D.day := -2
```

Quellcode 5.20: TC_20_correctDate.cmd

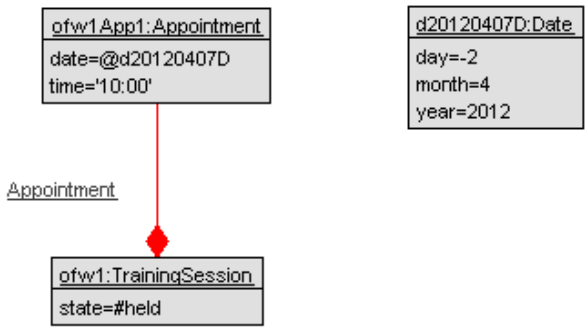

Abbildung 5.20.: TC_20_correctDate.cmd

5.21. Testfall 21 (TC_21_dateUnique.cmd)

In diesem Testfall wird überprüft ob das Datum eindeutig beziehungsweise einmalig vertreten ist. Um dies zu überprüfen setzen wir das Datum eines Termines auf einen anderen vorhanden Termin. Somit finden zwei Termine zur selben Zeit statt und führt somit zu einer Verletzung der Invariante.

```
1 open ../intern/model-base.cmd
2
3 !set miw1App1.date := d20120407D
4 !set miw1App1.time := '10:00'
```

Quellcode 5.21: TC_21_dateUnique.cmd

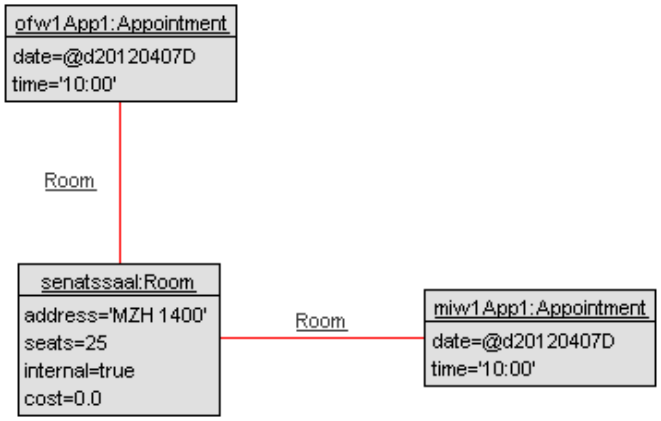

Abbildung 5.21.: TC_21_dateUnique.cmd

5.22. Testfall 22 (TC_22_positiveTimeFrame.cmd)

In diesem Testfall wird überprüft ob es sich hierbei um einen gültigen Zeitrahmen handelt. Bei einem Zeitrahmen müssen Startpunkt und Endpunkt gleich definiert sein und der Startpunkt des Zeitrahmens muss zetlich gesehen vor dem Endpunkt liegen. Da der Startpunkt in diesem testfall zetilich gesehen nach dem Endpunkt gesetzt wurde führt dieser Testfall zu einer Verletzung der Invariante.

```
1  !create startDate : Date
2  !set startDate.day := 12
3  !set startDate.month := 12
4  !set startDate.year := 1912
5
6  !create endDate : Date
7  !set endDate.day := 12
8  !set endDate.month := 12
9  !set endDate.year := 1910
10
11 !create testPeriod : Period
12 !set testPeriod.start := endDate
13 !set testPeriod.ends := startDate
```

Quellcode 5.22: TC_22_positiveTimeFrame.cmd

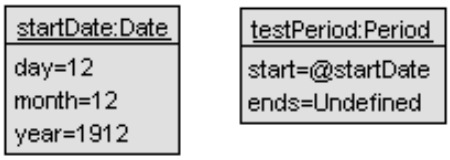

Abbildung 5.22.: TC_22_positiveTimeFrame.cmd

5.23. Testfall 23 (TC_23_valuesDefined.cmd)

In diesem Testfall geht es um die Überprüfung beziehungsweise Vollständigkeit der einzelnen Klassen. Für jede Klasse sind Werte vordefiniert, die diese Klasse enthalten muss. Besitzen Werte einen falschen Typ oder werden einige Werte erst garnicht gesetzt, so fürt dies zu einer Verletzung der Invariante(n). Folgende Klassen werden hier getestet:

- **Person:**
  In diesem Testfall wird eine neue Person mit dem Namen *testPerson* angelegt. Da eine Person sechs Parameter erwartet und hier das Geburtsdatum nicht mit eingetragen wird, führt dies zu einer Verletzung der Invariante.

- **Company:**
  In diesem Testfall wird eine neue Firma mit dem Namen *testCompany* angelegt. Da eine Firma drei Parameter erwartet und hier der Wert für die Telefonnummer nicht mit angegeben wurde, führt dies zu einer Verletzung der Invariante.

- **Category:**
  In diesem Testfall wird eine neue Kategorie mit dem Namen *testCategory* angelegt. Da eine Kategorie einen Parameter erwartet und hier kein Titel für die Kategorie angegeben wurde, führt dies zu einer Verletung der Invariante.

- **Training:**
  In diesem Testfall wird ein Training angelegt. Ein Training besitzt vier Parameter und hier wurde nur ein Titel und eine Beschreibung des Trainings angegeben. Dies führt nun zu einer Verletzung der Invariante.

- **TrainingSession:**
  In diesem Testfall wird eine Trainings-Sitzung angelegt. Eine Trainings-Sitzung benötigt einen Termin, eine akzeptierte Registrierung und einen Status. Da hier nichts weiter angegeben wurde, führt dies zu einer Verletzung der Invariante.

- **Appointment:**
  In diesem Testfall wird ein neuer Termin angelegt. Ein vollständiger Termin verfügt über einen Raum. einen Zeitpunkt wann der Termin stattfinden soll, einen Ausbilder und ein Catering. Hier wurde lediglich eine Zeit angegeben, um wie viel Uhr der Termin stattfinden soll. Dies führt letzendlich zu einer Verletung der Invariante.

- **Room:**
  In diesem Testfall wird ein neuer Raum angelegt. Ein neuer Raum benötigt vier Parameter. Da wieder weniger als vier Parameter übergeben wurden, führt dies zu einer Verletzung der Invariante.

- **Catering:**
  In diesem Testfall wird ein neues Catering angelegt. Ein Catering besteht aus einer Beschreibung und einem Kostenpunkt. Da hier nur eine Beschreibung angegeben wurde, führt dies zu einer Verletzung der Invariante.

- **Document:**
  In diesem Testfall wird ein neues Dokument angelegt. Ein Dokument besitzt lediglich eine Beschreibung des Dokumentes. Hier wurden allerdings keine Parameter übergeben und führt deshalb zu einer Verletzung der Invariante.

- **Bill:**
  In diesem Testfall geht es um den Betrag, den eine Person bezahlen muss, um sich registrieren zu können. Da hier ein Parameter übergeben wurde und Bill keine Parameter benötigt, führt dies zu einer Verletzung der Invariante.

- **Date:**
  In diesem Testfall geht es um ein Datum, welches mit einem Tag, einem Monat und einem Jahr definiert werden muss um als solches zu existieren. Da hier nur der Tag und das Jahr definiert wurde, führt dies zu einer Verletzung der Invariante.

- **Period:**
  In diesem Testfall geht es um einen Zeitrahmen. Bei einem Zeitrahmen müssen Startpunkt und Endpunkt gleich definiert sein und der Startpunkt des Zeitrahmens muss zetlich gesehen vor dem Endpunkt liegen. In diesem Testfall wurden allerdings nur Startpunkt und nicht Endpunkt definiert. Dies führt zu einer Verletzung der Invariante und ist somit nicht gültig.

```
1  -- Zum testen dieser Klasse einmal TC_01_validSetup.cmd laden und dann
2  -- die einzelnen Faelle testen.
3
4  -- Persons
5  !create testPerson : Person
6  !openter testPerson init('test', 'person', '#male', 'atHome', '1337')
7
8  -- Company
9  !create testCompany : Company
10 !openter testCompany init('test', 'atCompany')
11
12 -- Category
13 !create testCategory : Category
14 !openter testCategory init(5)
15
16 -- Training
17 !create testTraining : Training
18 !openter testTraining init('testTraining', 'One test training session')
19
20 -- TrainingSession
21 !create testTrainingSession : TrainingSession
22 !openter testTrainingSession init()
23
24 -- Appointment
25 !create testAppointment : Appointment
26 !openter testAppointment init('10:00')
27
28 -- Room
29 !create testRoom : Room
30 !openter testRoom init('theRoomplace', 12, true)
31
32 -- Catering
33 !create testCatering : Catering
34 !openter testCatering init('this is the catering description')
35
36 -- Document
37 !create testDocument : Document
38 !openter testDocument init()
39
40 -- Bill
41 !create testBill : Bill
42 !openter testBill init('10000')
43
44 -- Date
```

```
45 !create testDate : Date
46 !set startDate.day := 12
47 !set startDate.year := 1912
48
49 -- Period
50 !create startDate : Date
51 !set startDate.day := 12
52 !set startDate.month := 12
53 !set startDate.year := 1912
54
55 !create testPeriod : Period
56 !set testPeriod.start := startDate
```

Quellcode 5.23: TC_23_valuesDefined.cmd

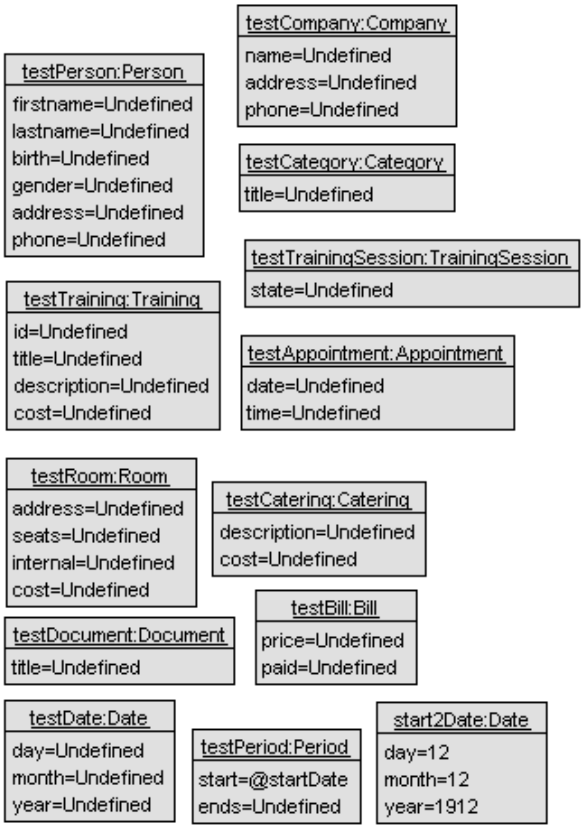

Abbildung 5.23.: TC_23_valuesDefined.cmd

6. Anfragen an das System

Die Anfragen geben Beispiele für Informationen, die man aus einem Systemzustand des Modells gewinnen kann. Die Daten werden z.B. gefiltert oder es werden verschiedene Daten miteinander verbunden, um daraus weitere Schlüsse zu ziehen.

Die Anfragen gehen dabei von einem korrekten Systemzustand aus, welcher alle Bedingungen, wie Invarianten und Multiplizitäten, erfüllt.

Wenn eine Anfrage auf Eingabewerte basiert, werden diese vorher mit `let`-Ausdrücken definiert.

6.1. Trainings an einem Datum

Die Anfrage aus Quellcode 6.1 gibt alle Trainings aus, welche einen Termin an dem angegebenen Datum haben. Dabei werden zuerst die Termine ausgewählt, die an dem ausgewählten Datum stattfinden und von dort aus die Trainings selektiert. Die Umwandlung in eine Menge am Ende entfernt doppelte Einträge.

Der Eingabeparameter ist ein `Date`-Objekt.

```
1  let
2    theDate:Date = @d20120407D
3  in
4  Appointment.allInstances()->select( a |
5      a.date = theDate
6  ).trainingSession.training->asSet()
```

Quellcode 6.1: Trainings an einem Datum

Ein mögliches Ergebnis der Anfrage sieht wie folgt aus:

```
1  Set{@ofw} : Set(Training)
```

6.2. Trainings an einem Datum mit freien Plätzen

Quellcode 6.2 stellt eine Erweiterung der Anfrage aus Abschnitt 6.1 dar. Hinzu kommt die Bedingung, dass das Training nicht nur eine TrainingSession mit einem Termin an dem bestimmten Datum hat, sondern auch mindestens ein freier Platz in allen Terminen der TrainingSession vorhanden ist.

Um dies zu erreichen werden die TrainingSessions dahingehend gefiltert, dass für jeden Termin gilt, wenn sie bereits einen Raum zugewiesen bekommen haben, muss dieser mindestens einen Platz frei haben. Plätze können dabei nur von bereits akzeptierten Teilnehmern belegt werden.

Der Eingabeparameter ist ein `Date`-Objekt.

```
1  let
2    theDate:Date = @d20120407D
3  in
4  Appointment.allInstances()->select( a |
5      a.date = theDate
6  ).trainingSession->select( ts |
7      ts.appointment->forAll( a | a.room.isDefined implies a.room.seats >
           ts.registration->select( r | r.accepted )->size() )
8  ).training->asSet()
```

Quellcode 6.2: Trainings an einem Datum mit freien Plätzen

Ein mögliches Ergebnis der Anfrage sieht wie folgt aus:

```
1  Set{@ofw} : Set(Training)
```

6.3. Anzahl an Teilnehmern von vergangenen Trainings

Die Anfrage aus Quellcode 6.3 gibt die Gesamtzahl der Teilnehmer aller Vergangenen TrainingSessions an. Dazu werden zuerst die TrainingSessions auf diejenigen gefiltert, welche bereits abgehalten sind und von diesen ausgehend die Anzahl der Teilnehmer summiert. Einzelne Teilnehmer können dabei öfter als ein Mal gezählt werden.

```
1  TrainingSession.allInstances()->select( ts |
2      ts.state = #held
3  )->collect( ts |
4      ts.registration->select( r | r.accepted )->size()
5  )->sum()
```

Quellcode 6.3: Anzahl an Teilnehmern von vergangenen Trainings

Ein mögliches Ergebnis der Anfrage sieht wie folgt aus:

```
1 32 : Integer
```

6.4. Einnahmen pro Training

Die Anfrage aus Quellcode 6.4 erzeugt eine Liste aller Trainings mit den dazugehörigen Einnahmen in absteigender Reihenfolge. Diese wird als `Sequence` von Tupeln mit dem eingenommenen Geld und dem Titel des Training ausgegeben. Die Einnahmen werden über die bezahlten Rechnungen der einzelnen `TrainingSession` pro `Training` aufsummiert. Die Sortierung geschieht am Ende über alle Einnahmen.

```
1 Training.allInstances()->collect( t |
2     Tuple{
3         title: t.title,
4         money: t.trainingSession.registration.bill->select( b | b.paid
              )->collect( b | b.price )->sum()
5     }
6 )->sortedBy( t | -t.money )
```

Quellcode 6.4: Einnahmen pro Training

Das Ergebnis für das Objektdiagram aus Abschnitt 4 sieht wie folgt aus:

```
1  Sequence{
2    Tuple{money=600.0, title='wii-bowling für Unsportliche'},
3    Tuple{money=600.0, title='Klingonen-Scrabble'},
4    Tuple{money=300.0, title='HALO Nerd-Kurs'},
5    Tuple{money=200.0, title='Office für Wissenschaftler'},
6    Tuple{money=100.0, title='HALO Einsteiger-Kurs'},
7    Tuple{money=25.0, title='Mitbewohnervereinbarungen in Word
       erstellen'},
8    Tuple{money=0.0, title='wii-sports für Unsportliche'},
9    Tuple{money=0.0, title='Visio für Astrophysiker'},
10   Tuple{money=0.0, title='Excel für Biologen'},
11   Tuple{money=0.0, title='3D-Schach'},
12   Tuple{money=0.0, title='wii-dance für Unsportliche'},
13   Tuple{money=0.0, title='3-Personen-Schach'},
14   Tuple{money=0.0, title='afn'}
15 } : Sequence(Tuple(money:Real, title:String))
```

6.5. Einnahmen pro Training pro Jahr

Quellcode 6.5 ist eine Erweiterung der Anfrage aus Abschnitt 6.4, insofern dass nur die Einnahmen eines Jahres berücksichtigt werden. Hierfür werden die `TrainingSessions`

dahingehend gefiltert, dass sie mindestens einen Termin in dem angegebenen Jahr haben müssen.

```
1  let
2      theYear : Integer = 2012
3  in
4  Training.allInstances()→collect( t |
5      Tuple{
6          title: t.title,
7          money: t.trainingSession→select( ts |
8                  ts.appointment→exists( a | a.date.year = theYear )
9              ).registration.bill→select( b | b.paid )→collect( b |
                   b.price )→sum()
10     }
11 )→sortedBy( t | -t.money )
```

Quellcode 6.5: Einnahmen pro Training pro Jahr

Ein mögliches Ergebnis der Anfrage für das Jahr 2012 sieht wie folgt aus:

```
1  Sequence{
2    Tuple{money=600.0, title='wii-bowling für Unsportliche'},
3    Tuple{money=600.0, title='Klingonen-Scrabble'},
4    Tuple{money=300.0, title='HALO Nerd-Kurs'},
5    Tuple{money=200.0, title='Office für Wissenschaftler'},
6    Tuple{money=100.0, title='HALO Einsteiger-Kurs'},
7    Tuple{money=25.0, title='Mitbewohnervereinbarungen in Word
       erstellen'},
8    Tuple{money=0.0, title='wii-sports für Unsportliche'},
9    Tuple{money=0.0, title='Visio für Astrophysiker'},
10   Tuple{money=0.0, title='Excel für Biologen'},
11   Tuple{money=0.0, title='3D-Schach'},
12   Tuple{money=0.0, title='wii-dance für Unsportliche'},
13   Tuple{money=0.0, title='3-Personen-Schach'},
14   Tuple{money=0.0, title='afn'}
15 } : Sequence(Tuple(money:Real, title:String))
```

6.6. Offene Rechnungen

Die Anfrage in Quellcode 6.6 gibt Informationen zu der Person, dem Kurs und dem ausstehendem Betrag für alle nicht bezahlten Rechnungen zurück. Dazu werden alle Rechnungen der vergangenen Veranstaltungen durchsucht und von den nicht bezahlten die Informationen als Tupel bereit gestellt.

```
TrainingSession.allInstances()→select( t | t.state = #held
    ).registration→select( r | r.accepted ).bill→select( b | not
    b.paid )→collect( b |
     Tuple{
          person: b.registration.participant.person,
          course: b.registration.trainingSession,
          amount: b.price
     }
)
```

Quellcode 6.6: Offene Rechnungen vergangener Veranstaltungen

Ein mögliches Ergebnis der Anfrage sieht wie folgt aus:

```
Bag{
     Tuple{amount=50.0, course=@hfe1, person=@penny}
} : Bag(Tuple(amount:Real, course:TrainingSession, person:Person))
```

6.7. Zertifikate eines Teilnehmers

Die Anfrage in Quellcode 6.7 gibt die vorhandenen Zertifikate einer Person zurück. Die Bedingung, um ein Zertifikat zu erhalten, sind dabei:

- Die `Person` muss für das `Training` registriert und akzeptiert werden.
- Die `TrainingSession` muss bereits abgehalten worden sein.
- Die `Person` muss die Rechnung für dieses `Training` bezahlt haben.

Die Registrierungen werden nach diesen Bedingungen gefiltert und die verbleibenden `Trainings` werden zurückgegeben. Dabei können mehrere Zertifikate für ein Training vergeben werden.

```
let
     thePerson : Person = @amy
in
thePerson.participant.registration→select( r |
     r.accepted and r.trainingSession.state = #held and r.bill.isDefined
          and r.bill.paid
).trainingSession.training
```

Quellcode 6.7: Zertifikate eines Teilnehmers

Ein mögliches Ergebnis der Anfrage sieht wie folgt aus:

```
Bag{@hfe,@ofw} : Bag(Training)
```

6.8. Qualifizierte Instruktoren für einen Trainingstermin

Die Anfrage in Quellcode 6.8 gibt zu einem Termin für eine Veranstaltung alle freien Dozenten an, welche die nötigen Qualifikationen erfüllen, diese Schulung durchzuführen. Hierzu wird für alle Instruktoren geprüft, ob sie an dem Tag bereits einen Termin haben, um die Invariante `Instructor::noDoubleOccupancy` aus Abschnitt 3.5.2 auf Seite 56 nicht zu verletzen. Außerdem muss der Dozent die nötigen Qualifikationen erfüllen, um den Kurs leiten zu können.

```
1  let
2      theAppointment : Appointment = @miw1App1
3  in
4  Instructor.allInstances()→select( i |
5      not i.appointment→exists( a |
6          a.date.equals( theAppointment.date )
7      )
8      and theAppointment.trainingSession.training.tool→forAll( t |
9          i.tool→includes( t )
10     )
11 ).person
```

Quellcode 6.8: Qualifizierte Instruktoren für einen Trainingstermin

Ein mögliches Ergebnis der Anfrage sieht wie folgt aus:

```
1  Bag{@drGablehauser} : Bag(Person)
```

6.9. Katalog

Der Katalog, definiert in Quellcode 6.9, ist eine Auflistung aller Trainings sortiert nach den Kategorien. Die Sortierung der Kategorien geschieht durch eine Tiefensuche, bei der auf jeder Gliederungsebene die Kategorien alphabetisch nach dem Titel sortiert werden. Die Operation `Category::mkFlatOrderedTree` aus Abschnitt 3.4.9 auf Seite 31 implementiert diese Eigenschaft. Mit dieser werden nun zuerst die Kategorien in die richtige Reihenfolge gebracht. Dafür werden der Operation alle Wurzelkategorien übergeben. Das Ergebnis ist eine sortierte Sequenz aller Kategorien. Diese werden im nächsten Schritt mit einem Tupel ersetzt, das die Kategorie und die zugehörigen Trainings enthält. Mit der Operation `Category::mkBreadcrumb` aus Abschnitt 3.4.10 auf Seite 32 werden zu den Kategorien jeweils die Pfade zur Wurzel hinzugefügt. Dies dient der besseren Möglichkeit der Einordnung.

```
Category.allInstances().any(true).mkFlatOrderedTree(
    Category.allInstances()→select( c | c.parent.isUndefined
        )→asSequence()
)→collect( c |
    Tuple{ category: c.mkBreadcrumb(), trainings: c.training }
)
```

Quellcode 6.9: Sortierter Katalog

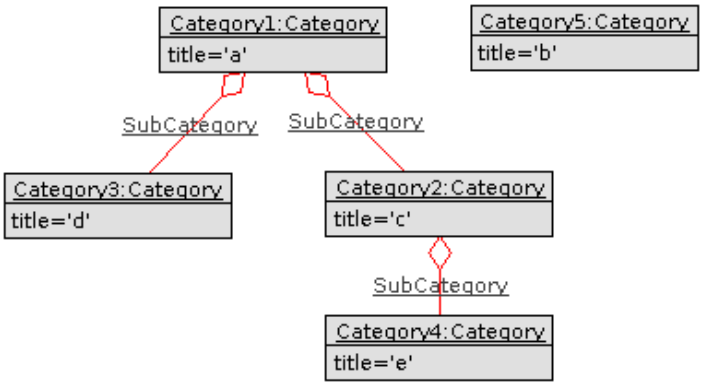

Abbildung 6.1.: Beispiel für einen Kategorienbaum

Das Ergebnis, mit dem in Abbildung 6.1 gezeigtem Objektdiagram, sieht wie folgt aus:

```
Sequence{
    Tuple{category=Sequence{@Category1}, trainings=Set{}},
    Tuple{category=Sequence{@Category1,@Category2}, trainings=Set{}},
    Tuple{category=Sequence{@Category1,@Category2,@Category4},
        trainings=Set{}},
    Tuple{category=Sequence{@Category1,@Category3}, trainings=Set{}},
    Tuple{category=Sequence{@Category5}, trainings=Set{}}
} : Sequence(Tuple(category:Sequence(Category),trainings:Set(Training)))
```

6.9.1. Erweitertes Nutzungsbeispiel

In diesem Beispiel soll anhand des Katalogs gezeigt werden, was man mit den Daten aus den Anfragen tun kann und wie man sie weiterverarbeiten kann.

Quellcode 6.10 definiert einen OCL-Ausdruck, welcher aus dem Ergebnis der Anfrage für den Katalog ein vollwertiges LaTeX-Dokument erstellt.

Zuerst wird der Katalog aus der Anfrage in der Variable `catalog` mit Hilfe eines `let`-Ausdrucks gespeichert. Zusätzlich wird eine Sequenz von Befehlen für die Gliederungsebene in der Variable `command` gespeichert. Somit ist es möglich mit dem Ausdruck `command->at( <Gliederungsebene> )` den passenden LaTeX-Ausdruck für die Überschrift auszuwählen.

Das Ergebnis des Ausdrucks ist ein `String`, welcher zuerst mit der Präambel von LaTeX beginnt. Hierzu wird eine Dokumentklasse ausgewählt, das Dokument begonnen und das Inhaltsverzeichnis ausgegeben. Nun wird über jeden Katalogeintrag – welche bereits von der Anfrage sortiert sind – iteriert und der Titel der Kategorie in der entsprechenden Gliederungseben an die Zeichenkette angefügt. Die Gliederungsebene entspricht der Anzahl an Kategorien in dem `category` Eintrag des Katalogs.

Unter jede dieser Kategorien wird über die einzelnen Trainings iteriert und Name sowie Beschreibung dessen ausgegeben. Um LaTeX-Fehler vorzubeugen muss hier geprüft werden, dass mindestens ein Training dieser Kategorie zugeordnet ist. Zuletzt wird das LaTeX-Dokument noch abgeschlossen.

Die resultierende Zeichenkette kann in einer Datei gespeichert und mit `pdflatex` in eine PDF-Datei übersetzt werden.

```
1  let
2    catalog:Sequence(Tuple(category:Sequence(Category),
         trainings:Set(Training)))=
3      Category.allInstances().any(true).mkFlatOrderedTree(
4          Category.allInstances()->select( c | c.parent.isUndefined
               )->asSequence()
5      )->collect( c |
6          Tuple{ category: c.mkBreadcrumb(), trainings: c.training }
7      )
8  in let
9    command:Sequence(String) = Sequence{'section', 'subsection',
         'subsubsection', 'paragraph'}
10 in
11 '\\documentclass{scrartcl}\n\\begin{document}\n\\tableofcontents\n'
12 .concat( catalog->iterate( entry ; res:String = '' |
13     res.concat('\\').concat( command->at( entry.category->size() )
           ).concat('{').concat( entry.category->last().title
           ).concat('}\n')
14       .concat( if entry.trainings->size() > 0
15         then
16           '\\begin{description}\n'.concat( entry.trainings->iterate(
                 tEntry ; ret:String = '' |
17         ret.concat('\\item[').concat( tEntry.title ).concat(']
               ').concat( tEntry.description ).concat('\n')
18       ) ).concat('\\end{description}\n')
19     else
20         ''
21     endif )
22 ) ).concat('\\end{document}')
```

Quellcode 6.10: Katalog zu LaTeX-Konverter

Die Ausgabe dieses Ausdrucks ist in Quellcode 6.11 sieht wie folgt aus. Um Platz zu sparen ist lediglich der Kategorienbaum von der Kategorie „Office“ gezeigt.

```
1  '\documentclass{scrartcl}
2  \begin{document}
3  \tableofcontents
4  \section{Office}
5  \begin{description}
6  \item[Office für Wissenschaftler] Grundlagen der Nutzung von Office für
       wissenschaftliche Zwecke
7  \end{description}
8  \subsection{Access}
9  \begin{description}
10 \item[afn] Access für Neurobiologen
11 \end{description}
12 \subsection{Excel}
13 \begin{description}
14 \item[Excel für Biologen] Möglichkeit für Biologen Excel effizient zu
       nutzen
15 \end{description}
16 \subsection{Visio}
17 \begin{description}
18 \item[Visio für Astrophysiker] Möglichkeit für Astrophysiker Visio
       effizient zu nutzen
19 \end{description}
20 \subsection{Word}
21 \begin{description}
22 \item[Mitbewohnervereinbarungen in Word erstellen] Tipps und Tricks zur
       Erstellung von Mitbewohnervereinbarungen in Word
23 \end{description}
24 \end{document}' : String
```

Quellcode 6.11: Ausgabe des Katalog zu LaTeX-Konverters

7. Sequenzdiagramme

Ein Sequenzdiagramm ist eine Diagrammart der UML (Unified Modeling Language) und beschreibt das Verhalten bzw. eine bestimmte Sicht auf die dynamischen Aspekte unseres Systems, meistens um Anwendungsfälle darzustellen. Durch den Einsatz von Sequenzdiagrammen werden in diesem Kapitel verschiedene kleinere Systemabläufe demonstriert.

Für die Erstellung der Sequenzdiagramme konnte das vorhandene Objektdiagramm leider nicht genutzt werden, da ein Fehler in USE zum Absturz führt, wenn Sequenzdiagramme basierend auf zu vielen Objekten dargestellt werden sollen.

7.1. Sequenzdiagramm 01 (SEQ_01_TrainingDocument.cmd)

In diesem Sequenzdiagramm wird eine neue Trainingseinheit mit dem Namen *t1* und ein neues Dokument mit dem Namen *d1* erstellt. Nachdem beide Klassen erstellt wurden, wird das Dokument dem Training zugeordnet, so dass zu der Trainingseinheit *t1* ein entsprechendes Dokument *d1* existiert.

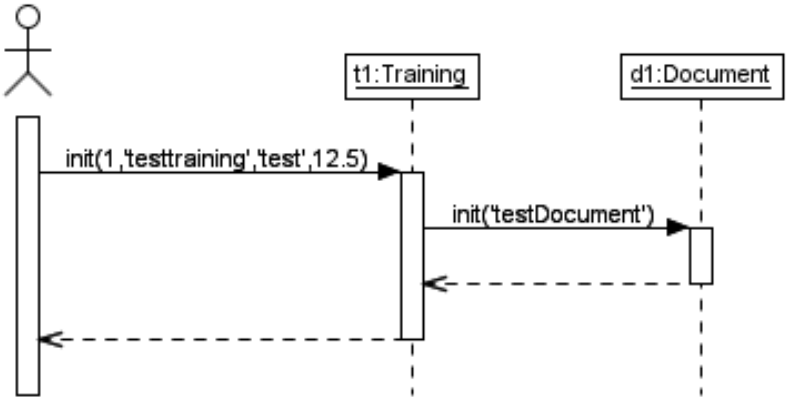

Abbildung 7.1.: SEQ_01_TrainingDocument.cmd

```
!create d160712 : Date
!set d160712.day := 16
!set d160712.month := 7
!set d160712.year := 2012

!create t1 : Training
!openter t1 init(1, 'testtraining', 'test', 12.5)
!set t1.id := 1
!set t1.title := 'testtraining'
!set t1.description := 'test'
!set t1.cost := 12.5

!create d1 : Document
!openter d1 init('testDocument')
!set self.title := 'testDocument'

!opexit

!insert (t1, d1) into Document

!opexit
```

Quellcode 7.1: SEQ_01_TrainingDocument.cmd

7.2. Sequenzdiagramm 02 (SEQ_02_PersonInstructor.cmd)

In diesem Sequenzdiagramm wird eine neue Person mit dem Namen *testPerson* erstellt. Diese Person wird durch die Funktion *becomeInstructor()* zu einem Ausbilder erweitert. Ein Ausbilder hat die Möglichkeit eine Qualifikation über bestimmte Tools zu erlangen. Somit wird eine neue Software mit dem Namen *testSoftware* erstellt und dem Ausbilder hinzugefügt.

```
!create testPersonD : Date
!set testPersonD.day := 2
!set testPersonD.month := 4
!set testPersonD.year := 1980

!create testPerson : Person
!openter testPerson init('Test', 'Person', testPersonD, #male,
    'atHome', '1337')
!set self.firstname := 'Test'
!set self.lastname := 'Person'
!set self.birth := testPersonD
!set self.gender := #male
!set self.address := 'atHome'
!set self.phone := '1337'
```

```
14
15 !openter testPerson becomeInstructor()
16 !create testPersonI : Instructor
17 !insert (testPerson, testPersonI) into IsAnInstructor
18
19 !create testSoftware : Software
20 !openter testSoftware init('testSoftware')
21 !set self.name := 'testSoftware'
22
23 !openter testPersonI addQualification(testSoftware)
24
25 !insert (testPersonI, testSoftware) into Qualification
26
27 !opexit
28 !opexit
29 !opexit testPersonI
30 !opexit
```

Quellcode 7.2: SEQ_02_PersonInstructor.cmd

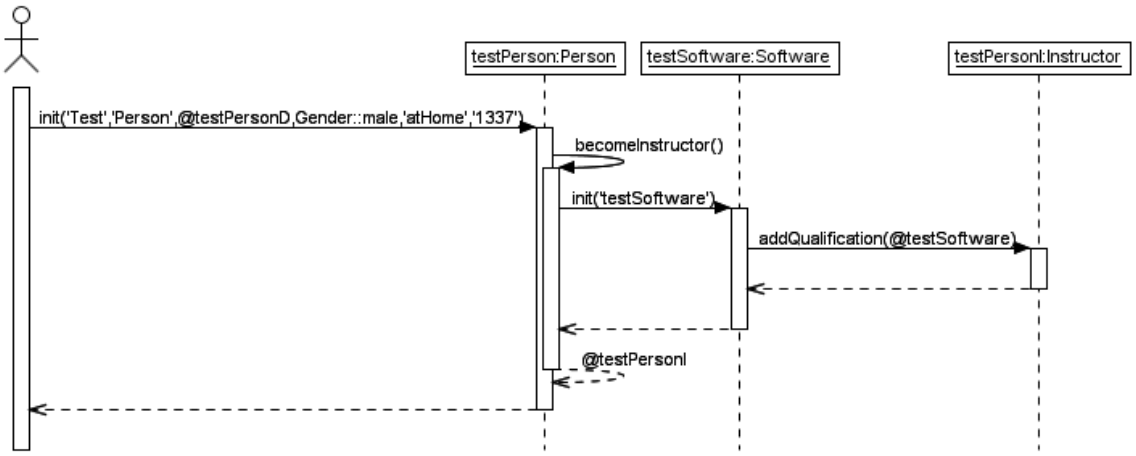

Abbildung 7.2.: SEQ_02_PersonInstructor.cmd

7.3. Sequenzdiagramm 03 (SEQ_03_AppointmentRoom.cmd)

In diesem Sequenzdiagramm wird ein neuer Termin mit dem Namen *testAppointment* und ein neuer Raum mit dem Namen *testRoom* erstellt. Zu jedem Termin, an dem eine Trainingssitzung stattfindet, kann auch ein entsprechender Raum zugewiesen werden. Nachdem also der Raum erstellt wurde, wird dieser dem Termin zugewiesen.

```
1  !create appDate : Date
2  !set appDate.day := 2
3  !set appDate.month := 4
4  !set appDate.year := 1980
5
6  !create testAppointment : Appointment
7  !openter testAppointment init(appDate, '10:00')
8  !set self.date := appDate
9  !set self.time := '10:00'
10
11 !create testRoom : Room
12 !openter testRoom init('roomAdress', 10, true, 20.5)
13 !set self.address:= 'roomAdress'
14 !set self.seats := 10
15 !set self.internal := true
16 !set self.cost := 20.5
17
18
19 !openter testAppointment assignRoom(testRoom)
20 !insert (testRoom, testAppointment) into Room
21 !opexit
22 !opexit
23 !opexit
```

Quellcode 7.3: SEQ_03_AppointmentRoom.cmd

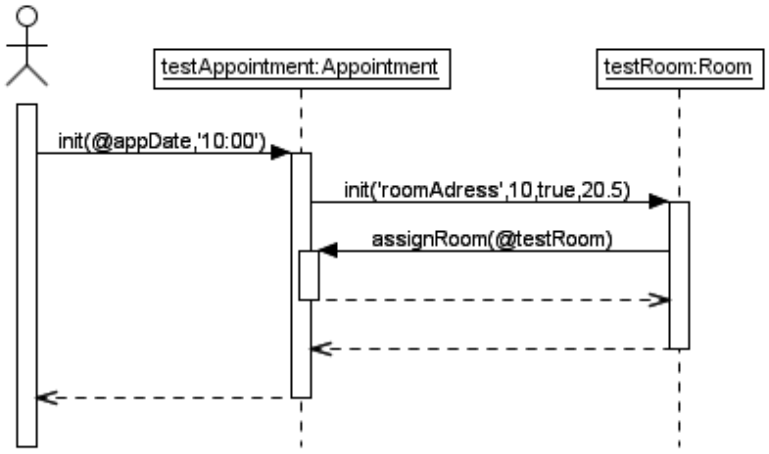

Abbildung 7.3.: SEQ_03_AppointmentRoom.cmd

7.4. Sequenzdiagramm 04 (SEQ_04_PersonParticipant.cmd)

In diesem Sequenzdiagramm wird eine Person mit dem Namen *testPerson* erstellt. Eine Person kann, indem die Funktion *becomeParticipant()* aufgerufen wird, um die Eigenschaft Teilnehmer erweitert werden. Es wird also eine Person erzeugt, die um die Eigenschaft Teilnehmer erweitert wird.

```
1  open ../intern/model-base.cmd
2
3  !create testPersonD : Date
4  !set testPersonD.day := 2
5  !set testPersonD.month := 4
6  !set testPersonD.year := 1980
7
8  !create testPerson : Person
9  !openter testPerson init('Test', 'Person', testPersonD, #male,
       'atHome', '1337')
10 !set self.firstname := 'Test'
11 !set self.lastname := 'Person'
12 !set self.birth := testPersonD
13 !set self.gender := #male
14 !set self.address := 'atHome'
15 !set self.phone := '1337'
16
17 !openter testPerson becomeParticipant()
18 !create testPersonP : Participant
19 !insert (testPerson, testPersonP) into IsAParticipant
20
21 !opexit testPersonP
22 !opexit
```

Quellcode 7.4: SEQ_04_PersonParticipant.cmd

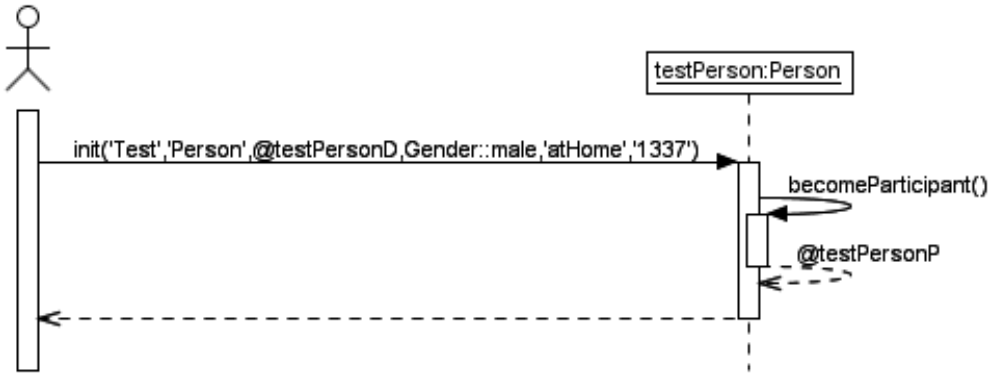

Abbildung 7.4.: SEQ_04_PersonParticipant.cmd

7.5. Sequenzdiagramm 05 (SEQ_05_AppointmentCatering.cmd)

In diesem Sequenzdiagramm wird ein neuer Termin mit dem Namen *app1* und ein neues Catering mit dem Namen *testCatering* erstellt. Durch die Funktion *addCatering()* wird dem neuen erstellten Termin das neu Catering hinzugefügt.

```
1  !create appD : Date
2  !set appD.day := 2
3  !set appD.month := 4
4  !set appD.year := 1980
5
6  !create app1 : Appointment
7  !openter app1 init(appD, '10:00')
8  !set self.date := appD
9  !set self.time := '10:00'
10
11 !create testCatering : Catering
12 !openter testCatering init('This is a test catering', 12.5)
13 !set self.description := 'This is a test catering'
14 !set self.cost := 12.5
15
16 !openter app1 addCatering()
17 !insert (app1, testCatering) into Catering
18
19 !opexit testCatering
20 !opexit
21 !opexit
```

Quellcode 7.5: SEQ_05_AppointmentCatering.cmd

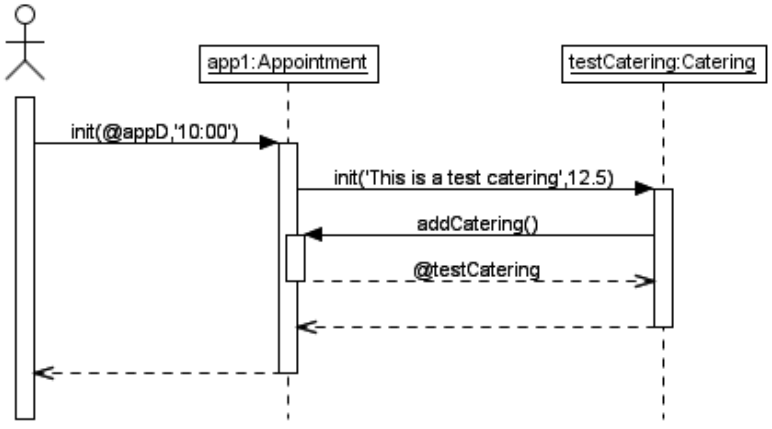

Abbildung 7.5.: SEQ_05_AppointmentCatering.cmd

7.6. Sequenzdiagramm 06 (SEQ_06_AppointmentTrainingSession.cmd)

In diesem Sequenzdiagramm wird eine neue Trainingssitzung mit dem Namen *ts1* und ein neuer Termin mit dem namen *app1* erstellt. Eine Trainingssitzung besitzt die Eigenschaft einen neuen Termin zugewiesen zu bekommen. Mit der Funktion *addAppointment()* wird der neuen Trainingssitzung der so eben erstellte Termin hinzugefügt.

```
1  !create appD : Date
2  !set appD.day := 16
3  !set appD.month := 7
4  !set appD.year := 2012
5
6  !create ts1 : TrainingSession
7  !openter ts1 init()
8
9  !set self.state := #announced
10
11 !create app1 : Appointment
12 !openter app1 init(appD, '10:00')
13 !set self.date := appD
14 !set self.time := '10:00'
15
16 !openter ts1 addAppointment(appD, '10:00')
17
18 !insert (self, app1) into Appointment
19
20 !opexit app1
21 !opexit
22 !opexit
```

Quellcode 7.6: SEQ_06_AppointmentTrainingSession.cmd

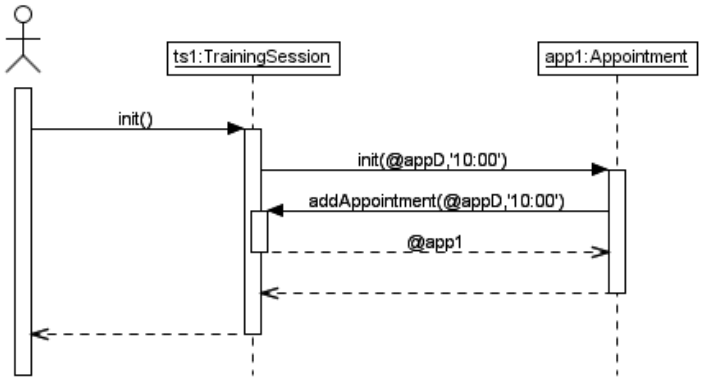

Abbildung 7.6.: SEQ_06_AppointmentTrainingSession.cmd

7.7. Sequenzdiagramm 07 (SEQ_07_TrainingCategory.cmd)

In diesem Sequenzdiagramm wird eine neue Trainingseinheit mit dem Namen *t1* und eine neue Kategorie mit dem Namen *cat1* erstellt. Trainingseinheiten haben die Eigenschaft in eine Kategorie eingeordnet zu werden. In diesem Fall wird die Trainingseinheit *i1* der Kategorie *cat1* hinzugefügt.

```
1  !create appD : Date
2  !set appD.day := 2
3  !set appD.month := 4
4  !set appD.year := 1980
5
6  !create t1 : Training
7  !openter t1 init(1, 'testtraining', 'test', 12.5)
8  !set self.id := 1
9  !set self.title := 'testtraining'
10 !set self.description := 'test'
11 !set self.cost := 12.5
12
13 !create cat1 : Category
14 !openter cat1 init('testCategory')
15 !set self.title := 'testCategory'
16
17 !opexit
18
19 !insert (cat1, t1) into Category
20
21 !opexit
```

Quellcode 7.7: SEQ_07_TrainingCategory.cmd

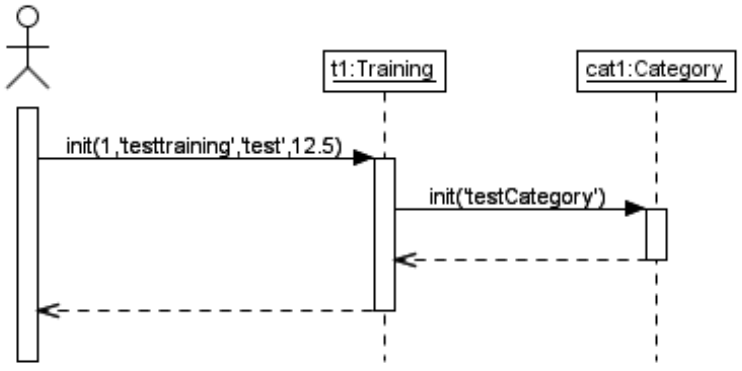

Abbildung 7.7.: SEQ_07_TrainingCategory.cmd

7.8. Sequenzdiagramm 08 (SEQ_08_PersonCompany.cmd)

In diesem Sequenzdiagramm wird eine neue Person mit dem Namen *testPerson* einer Firma mit dem Namen *nasa* hinzugefügt. Damit die jeweilige Person der Firma hinzugefügt werden kann, muss sie als Schulungsteilnehmer im System erfasst wurden sein. Durch die Funktion *becomeParticipant()*, wird die Person als Schulungsteilnehmer im System erfasst und kann der Firma *nasa* hinzugefügt werden.

```
1  open ../intern/model-base.cmd
2
3  !create testPersonD : Date
4  !set testPersonD.day := 2
5  !set testPersonD.month := 4
6  !set testPersonD.year := 1980
7
8  !create testPerson : Person
9  !openter testPerson init('Test', 'Person', testPersonD, #male,
       'atHome', '1337')
10 !set self.firstname := 'Test'
11 !set self.lastname := 'Person'
12 !set self.birth := testPersonD
13 !set self.gender := #male
14 !set self.address := 'atHome'
15 !set self.phone := '1337'
16
17 !openter testPerson becomeParticipant()
18 !create testPersonP : Participant
19 !insert (testPerson, testPersonP) into IsAParticipant
20
21 !openter nasa addEmployee(testPersonP)
22 !insert (testPersonP, nasa) into WorksFor
23
24 !opexit testPersonP
25 !opexit
```

Quellcode 7.8: SEQ_08_PersonCompany.cmd

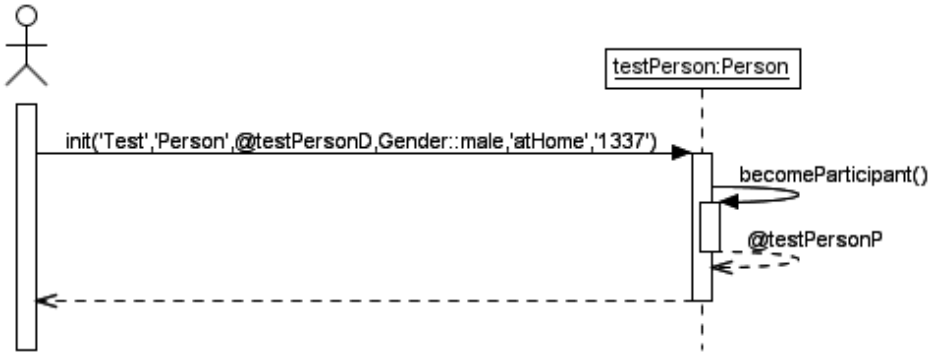

Abbildung 7.8.: SEQ_08_PersonCompany.cmd

7.9. Sequenzdiagramm 09 (SEQ_09_TrainingTool.cmd)

In diesem Sequenzdiagramm wird eine neue Trainingseinheit mit dem Namen *t1* erstellt. In einer Trainingsheit wird der Umgang mit bestimmten Software- oder Hardwaretools dem Teilnehmer nahe gelegt. Damit die Trainingseinheit um Tools erweitert werden können, benötigen wir vorher diese Tools, die mit Hilfe der Funktion *init* initialisiert werden. Es wird also eine neue Hardware mit dem Namen *hard1* und eine neue Software mit dem Namen *soft1* erstellt und der Trainingseinheit hunzugefügt.

```
1  !create t1 : Training
2  !openter t1 init(1, 'testtraining', 'test', 12.5)
3  !set t1.id := 1
4  !set t1.title := 'testtraining'
5  !set t1.description := 'test'
6  !set t1.cost := 12.5
7
8  !create soft1 : Software
9  !openter soft1 init('testSoftware')
10 !set self.name := 'testSoftware'
11
12 !opexit
13
14 !create hard1 : Hardware
15 !openter hard1 init('testHardware')
16 !set self.name := 'testHardware'
17
18 !opexit
19
20 !insert(t1, soft1) into Tool
21 !insert(t1, hard1) into Tool
22 !opexit
```

Quellcode 7.9: SEQ_09_TrainingTool.cmd

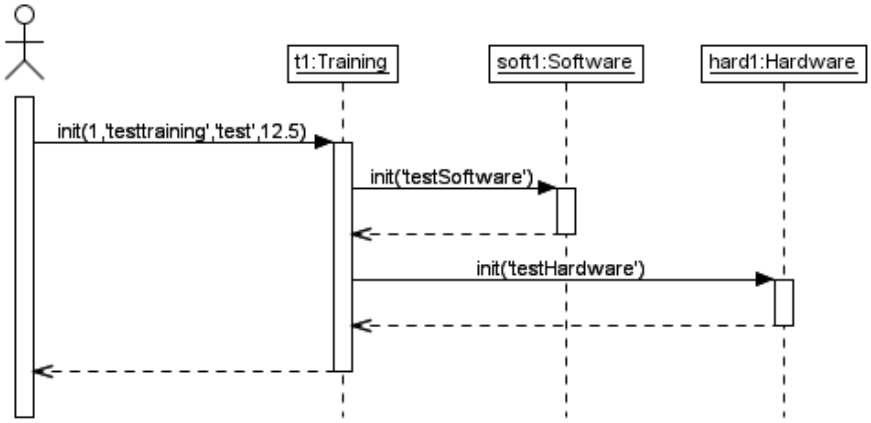

Abbildung 7.9.: SEQ_09_TrainingTool.cmd

7.10. Sequenzdiagramm 10 (SEQ_10_TrainingTrainingSession.cmd)

In diesem Sequenzdiagramm wird eine neue Trainingseinheit mit dem Namen *t1* und eine neue Trainingssitzung mit dem Namen *ts1* erstellt. Trainingseinheiten besitzen die Eigenschaft aus ihnen eine Trainingssitzung zu erstellen. Für diese Trainingssitzungen können sich dann Personen registrieren und daran teilnehmen. Nachdem die Trainingseinheit erstellt wurde, wird dieser eine Trainingssitzung hinzugefügt. Nachdem die Trainingssitzung erstellt wurde, kann die Trainingseinheit der Trainingssitzung hinzugefügt werden.

```
1  !create t1 : Training
2  !openter t1 init(1, 'testtraining', 'test', 12.5)
3  !set t1.id := 1
4  !set t1.title := 'testtraining'
5  !set t1.description := 'test'
6  !set t1.cost := 12.5
7
8  !openter t1 addTrainingSession()
9
10 !create ts1 : TrainingSession
11 !openter ts1 init()
12 !set self.state := #announced
13
14 !opexit
15
16 !insert (self, ts1) into TrainingSession
17
18 !opexit ts1
19 !opexit
```

Quellcode 7.10: SEQ_10_TrainingTrainingSession.cmd

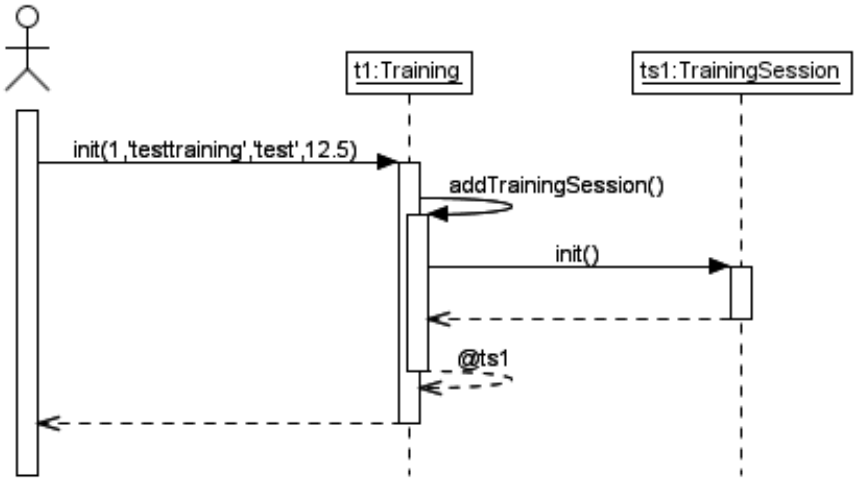

Abbildung 7.10.: SEQ_10_TrainingTrainingSession.cmd

A. Modell

```
1  model Schulungsverwaltung
2
3  enum Gender { male, female }
4
5  enum TrainingState { announced, settled, canceled, held }
6
7  class Person
8  attributes
9    firstname:String
10   lastname:String
11   birth:Date
12   gender:Gender
13   address:String
14   phone:String
15 operations
16   init(aFirstname:String, aLastname:String, aBirth:Date,
       aGender:Gender, anAddress:String, aPhone:String)
17     pre freshInstance: firstname.isUndefined and lastname.isUndefined
         and birth.isUndefined and gender.isUndefined and
         address.isUndefined and phone.isUndefined
18     post firstnameDefined: firstname = aFirstname
19     post lastnameDefined: lastname = aLastname
20     post birthDefined: birth = aBirth
21     post genderDefined: gender = aGender
22     post addressDefined: address = anAddress
23     post phoneDefined: phone = aPhone
24   becomeParticipant():Participant
25     pre isNoParticipant: participant.isUndefined
26 --  post participantIsNew: result.oclIsNew
27     post isParticipant: participant = result
28   becomeInstructor():Instructor
29     pre isNoInstructor: instructor.isUndefined
30 --  post instructorIsNew: result.oclIsNew
31     post isInstructor: instructor = result
32 constraints
33   inv valuesDefined: firstname.isDefined and lastname.isDefined and
       birth.isDefined and gender.isDefined and address.isDefined and
       phone.isDefined
34 end
35
36 class Participant
37 operations
```

```
38    registerForTrainingSession(aTrainingSession :
          TrainingSession):Registration
39      pre isNotRegistered: not trainingSession→includes(
            aTrainingSession )
40      post isRegistered: trainingSession→includes( aTrainingSession )
41      post numIncreased: trainingSession→size() =
            trainingSession@pre→size() + 1
42  --  post registrationIsNew: result.oclIsNew
43      post isNotAccepted: not result.accepted
44  end
45
46  class Instructor
47  operations
48    addQualification(t : Tool)
49      pre isNotQualified: not tool→includes( t )
50      post isQualified: tool→includes( t )
51      post numIncreased: tool→size() = tool@pre→size() + 1
52  constraints
53    inv noDoubleOccupancy: appointment→forAll( a1, a2 | a1 <> a2 implies
          not a1.date.equals(a2.date) )
54  end
55
56  class Company
57  attributes
58    name:String
59    address:String
60    phone:String
61  operations
62    init(aName:String, anAddress:String, aPhone:String)
63      pre freshInstance: name.isUndefined and address.isUndefined and
            phone.isUndefined
64      post nameDefined: name = aName
65      post addressDefined: address = anAddress
66      post phoneDefined: phone = aPhone
67    addEmployee(p : Participant)
68      pre isNoEmployee: not employee→includes( p )
69      post isEmployee: employee→includes( p )
70      post numIncreased: employee→size() = employee@pre→size() + 1
71  constraints
72    inv valuesDefined: name.isDefined and address.isDefined and
          phone.isDefined
73  end
74
75  class Category
76  attributes
77    title:String
78  operations
79    init(aTitle:String)
80      pre freshInstance: title.isUndefined
81      post titleDefined: title = aTitle
82    mkFlatOrderedTree( cats : Sequence(Category) ) : Sequence(Category)=
```

```
83      cats->sortedBy( c | c.title )->iterate( c ; ret :
            Sequence(Category) = Sequence{} |
84        if c.child->isEmpty() then
85          ret->append( c )
86        else
87          ret->append( c )->union( c.mkFlatOrderedTree(
                c.child->asSequence() ))
88        endif )
89    mkBreadcrumb() : Sequence(Category)=
90      if self.parent.isUndefined then
91        Sequence{ self }
92      else
93        self.parent.mkBreadcrumb()->append( self )
94      endif
95  constraints
96    inv valuesDefined: title.isDefined
97  end
98
99  class Training
100 attributes
101   id:Integer
102   title:String
103   description:String
104   cost:Real
105 operations
106   init(anId:Integer, aTitle:String, aDescription:String, aCost:Real)
107     pre freshInstance: id.isUndefined and title.isUndefined and
            description.isUndefined and cost.isUndefined
108     post idDefined: id = anId
109     post titleDefined: title = aTitle
110     post descriptionDefined: description = aDescription
111     post costDefined: cost = aCost
112   addTrainingSession():TrainingSession
113 --  post sessionIsNew: result.oclIsNew
114     post sessionIsLinked: result.training = self
115 constraints
116   inv valuesDefined: id.isDefined and title.isDefined and
        description.isDefined and cost.isDefined
117   inv costGTEZero: cost >= 0
118   inv idUnique: Training.allInstances()->forAll( t | t <> self implies
        t.id <> self.id )
119 end
120
121 class TrainingSession
122 attributes
123   state:TrainingState
124 operations
125   init()
126     pre freshInstance: state.isUndefined
127     post stateDefined: state = #announced
128   addAppointment(aDate:Date, aTime:String):Appointment
129 --  post appointmentIsNew: result.oclIsNew
```

```
130       post appointmentIsLinked: result.trainingSession = self
131       post valuesCorrect: result.date = aDate and result.time = aTime
132     confirmRegistration(aParticipant : Participant)
133       pre participantRegistered: participant->includes( aParticipant )
134       pre isNotAccepted: registration->exists( r | r.participant =
              aParticipant implies not r.accepted )
135       pre roomsHaveEnoughSeats: appointment->forAll( a | a.room.isDefined
              implies a.room.seats > registration->size() )
136       post isAccepted: registration->select( r | r.participant =
              aParticipant )->any(true).accepted
137     settleTrainingSession()
138       pre stateAnnounced: state = #announced
139       pre hasParticipants: registration->select( r | r.accepted
              )->notEmpty()
140       post stateSettled: state = #settled
141       post billsExist: registration->forAll( r | r.accepted implies
              r.bill.isDefined )
142     cancelTrainingSession()
143       pre isNotCanceled: state <> #canceled
144       pre isNotHeld: state <> #held
145       post stateCanceled: state = #canceled
146       post noAcceptedParticipants: registration->forAll( r | not
              r.accepted )
147       post roomsFree: appointment->forAll( a | a.room.isUndefined )
148       post instructorsFree: appointment->forAll( a |
              a.instructor.isUndefined )
149     finishTrainingSession()
150       pre isSettled: state = #settled
151       post isHeld: state = #held
152   constraints
153     inv valuesDefined: state.isDefined
154     inv settledMustHaveRooms: (state = #settled or state = #held) implies
          appointment->forAll( a | a.room.isDefined )
155     inv settledMustHaveInstructors: (state = #settled or state = #held)
          implies appointment->forAll( a | a.instructor.isDefined )
156     inv canceledHasNoRoom: state = #canceled implies appointment->forAll(
          a | a.room.isUndefined )
157     inv canceledHasNoInstructor: state = #canceled implies
          appointment->forAll( a | a.instructor.isUndefined )
158     inv instructorIsNoParticipant: participant.person->intersection(
          appointment.instructor.person )->isEmpty()
159   end
160
161   class Appointment
162   attributes
163     date:Date
164     time:String
165   operations
166     init(aDate:Date, aTime:String)
167       pre freshInstance: date.isUndefined and time.isUndefined
168       post dateDefined: date = aDate
169       post timeDefined: time = aTime
```

```
170     assignRoom(r : Room)
171       pre hasNoRoom: room.isUndefined
172       post roomAssigned: room = r
173     assignInstructer(i : Instructor)
174       pre hasNoInstructor: instructor.isUndefined
175       post instructorAssigned: instructor = i
176     addCatering():Catering
177       pre hasNoCatering: catering.isUndefined
178   --  post cateringIsNew: result.oclIsNew
179       post cateringIsLinked: result.appointment = self
180   constraints
181     inv valuesDefined: date.isDefined and time.isDefined
182     inv instructorHasKnowledge: instructor.isDefined implies
          trainingSession.training.tool->forAll( t | t.instructor->includes(
          instructor ) )
183     inv enoughSeats: room.isDefined implies room.seats >=
          trainingSession.registration->select( r | r.accepted )->size()
184   end
185
186   class Room
187   attributes
188     address:String
189     seats:Integer
190     internal:Boolean
191     cost:Real
192   operations
193     init(anAddress:String, numSeats:Integer, isInternal:Boolean,
          aCost:Real)
194       pre freshInstance: address.isUndefined and seats.isUndefined and
            internal.isUndefined and cost.isUndefined
195       post addressDefined: address = anAddress
196       post seatsDefined: seats = numSeats
197       post internalDefined: internal = isInternal
198       post costDefined: cost = aCost
199   constraints
200     inv valuesDefined: address.isDefined and seats.isDefined and
          internal.isDefined and cost.isDefined
201     inv costGTEZero: cost >= 0
202     inv noDoubleOccupancy: appointment->forAll( a1, a2 | a1 <> a2 implies
          not (a1.date = a2.date) )
203   end
204
205   class Catering
206   attributes
207     description:String
208     cost:Real
209   operations
210     init(aDescription:String, aCost:Real)
211       pre freshInstance: description.isUndefined and cost.isUndefined
212       post descriptionDefined: description = aDescription
213       post costDefined: cost = aCost
214   constraints
```

```
215   inv valuesDefined: description.isDefined and cost.isDefined
216   inv costGTEZero: cost >= 0
217 end
218
219 abstract class Tool
220 attributes
221   name:String
222 operations
223 constraints
224   inv valuesDefined: name.isDefined
225 end
226
227 class Software < Tool
228 operations
229   init(aName:String)
230     pre freshInstance: name.isUndefined
231     post nameDefined: name = aName
232 end
233
234 class Hardware < Tool
235 operations
236   init(aName:String)
237     pre freshInstance: name.isUndefined
238     post nameDefined: name = aName
239 end
240
241 class Document
242 attributes
243   title:String
244 operations
245   init(aTitle:String)
246     pre freshInstance: title.isUndefined
247     post titleDefined: title = aTitle
248 constraints
249   inv valuesDefined: title.isDefined
250 end
251
252 associationclass Registration
253 between
254   TrainingSession [*]
255   Participant [*]
256 attributes
257   accepted:Boolean
258 operations
259   init()
260     pre freshInstance: accepted.isUndefined
261     post acceptedDefined: not accepted
262 constraints
263   inv valuesDefined: accepted.isDefined
264 end
265
266 class Bill
```

```
267 attributes
268   price:Real
269   paid:Boolean
270 operations
271   init()
272     pre freshInstance: price.isUndefined and paid.isUndefined
273     post priceDefined: price =
          registration.trainingSession.training.cost +
          (registration.trainingSession.appointment→collect( a | if
          a.catering.isDefined then a.teaches.cost + a.room.cost +
          a.catering.cost else a.teaches.cost + a.room.cost endif )→sum())
274     post paidDefined: not paid
275   updatePrice()
276     pre isNotPaid: not paid
277     post priceCorrect: price =
          registration.trainingSession.training.cost +
          (registration.trainingSession.appointment→collect( a | if
          a.catering.isDefined then a.teaches.cost + a.room.cost +
          a.catering.cost else a.teaches.cost + a.room.cost endif )→sum())
278   pay()
279     pre isNotPaid: not paid
280     post isPaid: paid
281 constraints
282   inv valuesDefined: price.isDefined and paid.isDefined
283   inv priceGTEZero: price >= 0
284   inv participantAccepted: registration.accepted
285 end
286
287 associationclass Teaches
288 between
289   Instructor [0..1]
290   Appointment [*]
291 attributes
292   cost:Real
293 operations
294   init(aCost:Real)
295     pre freshInstance: cost.isUndefined
296     post costDefined: cost = aCost
297 constraints
298   inv valuesDefined: cost.isDefined
299   inv costGTEZero: cost >= 0
300 end
301
302 class Date
303 attributes
304   day : Integer
305   month : Integer
306   year : Integer
307 operations
308   equals(d : Date) : Boolean =
309     self.year = d.year and self.month = d.month and self.day = d.day
310   after(d : Date) : Boolean =
```

```
311       if self.year > d.year then true else
312         if self.year = d.year then
313           if self.month > d.month then true else
314             if self.month = d.month then
315               if self.day > d.day then true else false endif
316             else false endif endif
317         else false endif
318       endif
319     before(d : Date) : Boolean =
320       if self.year < d.year then true else
321         if self.year = d.year then
322           if self.month < d.month then true else
323             if self.month = d.month then
324               if self.day < d.day then true else false endif
325             else false endif endif
326         else false endif
327       endif
328     liesBetween(start:Date, ending:Date) : Boolean =
329       if start.before(ending) then
330         self.after(start) and self.before(ending)
331       else
332         self.after(ending) and self.before(start)
333       endif
334     toString() : String =
335       self.day.toString().concat('.').concat( self.month.toString()
            ).concat('.').concat( self.year.toString() )
336     liesIn(p : Period) : Boolean =
337       self.equals(p.start) or self.equals(p.ends) or (self.after(p.start)
            and self.before(p.ends))
338   constraints
339     inv valuesDefined: self.day.isDefined() and self.month.isDefined()
          and self.year.isDefined()
340     inv correctDate:
341       self.year > 1900 and
342       self.month >= 1 and self.month <= 12 and
343       if self.month = 1 or self.month = 3 or self.month = 5 or self.month
            = 7 or self.month = 8 or self.month = 10 or self.month = 12 then
            self.day >= 1 and self.day <= 31
344       else
345         if self.month = 2 then
346           if self.year.mod(4) = 0 and (self.year.mod(100) <> 0 or
                self.year.mod(400) = 0) then
347             self.day >= 1 and self.day <= 29
348           else
349             self.day >= 1 and self.day <= 28
350           endif
351         else
352           self.day >= 1 and self.day <= 30
353         endif
354       endif
355     inv dateUnique: Date.allInstances()->forAll( d | self <> d implies
          not self.equals(d) )
```

```
356 end
357
358 class Period
359 attributes
360   start : Date
361   ends : Date
362 operations
363   equals(p : Period) : Boolean = self.start.equals(p.start) and
        self.ends.equals(p.ends)
364   before(p : Period) : Boolean = self.ends.before(p.start)
365   after(p : Period) : Boolean = self.start.after(p.ends)
366   during(p : Period) : Boolean =
367     (self.start.after(p.start) and (self.ends.equals(p.ends) or
          self.ends.before(p.ends)))
368       or ((self.start.equals(p.start) or self.start.after(p.start)) and
            self.ends.before(p.ends))
369   contains(p : Period) : Boolean = p.during(self)
370   overlaps(p : Period) : Boolean = self.start.before(p.start) and
        self.ends.after(p.start) and self.ends.before(p.ends)
371   overlapped_by (p : Period) : Boolean = p.overlaps(self)
372   meets(p : Period) : Boolean = self.ends.equals(p.start)
373   met_by(p : Period) : Boolean = p.meets(self)
374   starts(p : Period) : Boolean = self.start.equals(p.start) and
        self.ends.before(p.ends)
375   started_by (p : Period) : Boolean = p.starts(self)
376   finishes(p : Period) : Boolean = self.ends.equals(p.ends) and
        self.start.after(p.start)
377   finished_by(p : Period) : Boolean = p.finishes(self)
378 constraints
379   inv valuesDefined: self.start.isDefined() and self.ends.isDefined()
380   inv positiveTimeFrame: self.start.equals(self.ends) or
        self.start.before(self.ends)
381 end
382
383 association IsAnInstructor
384 between
385   Person [1]
386   Instructor [0..1]
387 end
388
389 association IsAParticipant
390 between
391   Person [1]
392   Participant [0..1]
393 end
394
395 association WorksFor
396 between
397   Participant [*] role employee
398   Company [0..1] role employer
399 end
400
```

```
401 association Bill
402 between
403   Registration [1]
404   Bill [0..1]
405 end
406
407 association Category
408 between
409   Category [1]
410   Training [*]
411 end
412
413 aggregation SubCategory
414 between
415   Category [0..1] role parent
416   Category [*] role child
417 end
418
419 composition TrainingSession
420 between
421   Training [1]
422   TrainingSession [*]
423 end
424
425 composition Appointment
426 between
427   TrainingSession [1]
428   Appointment [1..*]
429 end
430
431 association Room
432 between
433   Room [0..1]
434   Appointment [*]
435 end
436
437 composition Catering
438 between
439   Appointment [1]
440   Catering [0..1]
441 end
442
443 association Document
444 between
445   Training [1]
446   Document [*]
447 end
448
449 association Qualification
450 between
451   Instructor [*]
452   Tool [*]
```

```
453 end
454
455 association Tool
456 between
457   Training [*]
458   Tool [*]
459 end
```

Quellcode A.1: Gesamtes USE-Modell

Abbildungen und Quellcode

Abbildungsverzeichnis

Quellcodeverzeichnis

Literaturverzeichnis

[All83] Allen, James F.: Maintaining knowledge about temporal intervals. In: *Commun. ACM* 26 (1983), November, Nr. 11, 832–843. http://dx.doi.org/10.1145/182.358434. – DOI 10.1145/182.358434. – ISSN 0001–0782

[Bre07] Bremen, Database Systems Group U.: USE – A UML based Specification Environment. 0.1 (2007)

[OMG11] OMG: Unified Modeling Language (UML). 2.4.1 (2011)

[OMG12] OMG: Object Constraint Language (OCL). 2.3.1 (2012)