Universität Bremen

Entwurf von Informationssystemen

Entwurf eines Videotheks-Informationssystems

Autor:
Pascal Knüppel

Sommersemester 2013

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

Abgabe:
15.08.2013

Inhaltsverzeichnis

1 Einleitung

Diese Arbeit ist im Rahmen der Veranstaltung **„Entwurf von Informationssystemen“** im Sommersemester 2013 entstanden, in welchem ein System modelliert werden soll, welches anhand von verschiedenen Invarianten, Vor- und Nachbedingungen, validiert und auf Korrektheit überprüft werden soll.
Als Hilfsmittel wurde hier in erster Linie das Tool **„USE“ (UML Based Specification Environment)** verwendet, welches die **„OCL“ (Object Constraint Language)** unterstützt und diese zur Validierung des Systems verwendet. OCL ist eine formale Sprache, die Ausdrücke beschreibt, welche auf **„UML“ (Unified Modeling Language)** Modelle angewendet werden können. Dabei sind diese Ausdrücke in der Regel Anfragen, oder Bedingungen, die für das System gelten sollen.
Es ist aber auch möglich in OCL Ausdrücke dazu zu verwenden, um Operationen zu deklarieren, die den Zustand des Systems verändern können.

Um das entwickelte System nun entsprechend zu erläutern, wird als nächstes eine Systembeschreibung folgen. Diese Beschreibung teilt sich in eine informelle und noch eine detailliertere Beschreibung. Weiter wird eine nähere Betrachtung der einzelnen Klassen durchgeführt und beschrieben, wofür welches Attribut steht und welche Operationen die Klassen verwenden können und wieso sie das können.
Danach wird eine Beschreibung der Assoziationen folgen und es wird erläutert warum wo welche Assoziation verwendet wurde.
Im Anschluss werden dann valide Zustände des Systems aufgezeigt und mittels diverser Tests gezeigt, dass das System korrekt funktioniert.

Im laufe dieses Dokuments wird ggf. des öfteren der Begriff „Datenbank“ verwendet. Dabei bezieht der Begriff nicht auf die Zustände des Tools „USE“ sondern vielmehr wird davon ausgegangen, dass das System lauffähig in der Praxis über ein Datenbanksystem funktioniert, weshalb der Begriff „Datenbank“ des öfteren seine Anwendung innerhalb dieses Dokuments findet.

2 Systembeschreibung

In der folgenden Abbildung 2.1 wird das entworfene System über UML beschrieben. Zu dieser Abbildung folgen nun zwei Erläuterungen. Eine informelle Beschreibung, welche die Beschreibung über das System in einfachen Worten wiedergeben soll und eine detaillierte Beschreibung, die den genauen Sachverhalt, des UML-Diagramms beschreibt.

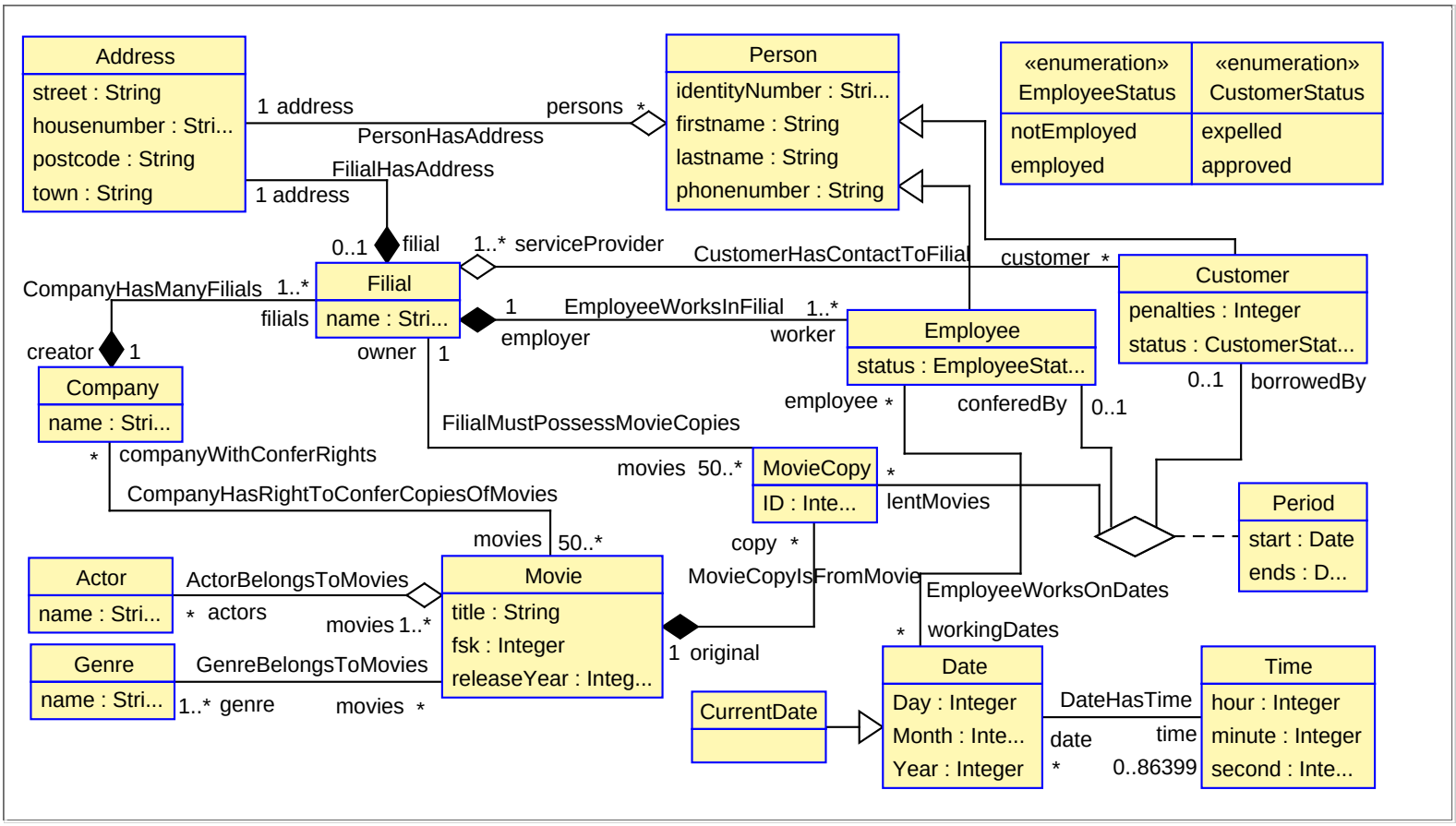

Abbildung 2.1: Modell einer Videothekenkette

2.1 Informelle Systembeschreibung

Entwickelt wurde ein System, welches als eine Gesamtübersicht für eine Videothekenkette dienen soll. Dabei ist hier bitte davon abzusehen, ob das System nun wirklich genau allen Regeln der wirklichen Welt folgt oder nicht (Die Begründung hierfür ist die, dass ich nicht alle exakten Regeln kenne und entsprechend nicht weiß, ob ich das System der realen Welt gegenüber korrekt modelliert habe).

Ausgehend vom entwickelten System gibt es eine Videothekenkette, welche zu einer übergeordneten Firma gehört. Dies soll bedeuten, dass eine Firma bspw. viele Filialen besitzt, welche dazu berechtigt sind Filmkopien zu verleihen.
Dabei soll jede Firma sich die Rechte zu einer Kollektion von Filmen angekauft haben. Sind diese Rechte erworben, ist es der Firma erlaubt Kopien dieser Filme in den einzelnen Filialen

unterzubringen, welche dann an Kunden verliehen werden dürfen. Hier ist darauf zu achten, dass die Kopien, die eine Filiale zum Verleih anbietet, immer eine Untermenge der Filme ist, zu denen die Firma die Rechte erworben hat.
Damit die Kunden außerdem von den Angestellten kompetent beraten werden können, ist es den Angestellten möglich, Anfragen an das System zu stellen, wie bspw. welche Filme mit bestimmten Schauspielern in der eigenen Filiale vorhanden sind, oder welche Filme mit bestimmten Genres, oder auch Filme aus bestimmten Jahren, oder einer bestimmten Altersfreigabe. Auch ist es möglich, den Kunden ggf. an eine andere Filiale zu verweisen, da die Angestellten auch deren Bestände einsehen können, wenn die eigene Filiale die entsprechende Kopie nicht mehr verfügbar oder überhaupt im Bestand hat.
Wenn nun ein Kunde einen oder mehrere Filme entleihen möchte, wird dies im System so vermerkt, dass man jederzeit einsehen kann, wann die Filme entliehen wurden, bis wann sie spätestens zurückgebracht werden müssen und welcher Angestellte dem Kunden die Filme bereitgestellt hat, um diesen Angestellten bei eventuellen Fehlern, auf die Fehler hinzuweisen.
Damit dies auch alles richtig funktionieren kann, muss jede Filiale ihre Kunden und ihre Angestellten im System vermerken. Dabei werden von jeder Person, egal ob Kunde oder Angestellter die Adressen, die Personalausweisnummer und die Telefonnummer, wie auch Vor- und Nachname gespeichert, damit man die Personen bei entsprechenden Anliegen erreichen kann.

2.2 Detaillierte Systembeschreibung

Es ist anzunehmen, dass die Attribute in den Klassen IMMER instanziiert sein müssen, es sei denn es wird etwas anderes in den folgenden Punkten gesagt.

- Jedes Filmobjekt muss einzigartig in der Datenbank vorhanden sein. Dabei wird die Einzigartigkeit über alle Attribute der Objekte bestimmt. Dazu muss jedes Filmobjekt mindestens ein Genre zugewiesen bekommen. Denn es gibt keine Filme, die nicht in ein Genre eingeordnet werden können.

- Jedes Schauspieler Objekt, muss mit mindestens einem Film verbunden sein.

- Jedes Genre Objekt soll aufgrund von Speicherplatzbedingungen einzigartig sein und darf zu jeder Zeit in die Datenbank eingetragen werden, egal ob es einem Filmobjekt zugewiesen wurde, oder nicht.

- Ein CompanyObjekt muss die Rechte für mindestens 50 Filme haben und muss mindestens eine Filiale eröffnet haben.

- Jede Filiale muss mindestens einen Angestellten haben, braucht genau eine Addresse und soll zumindest 50 Filmkopien zur Verfügung stellen, die verliehen werden können.

- Jede registrierte Filmkopie muss im Besitz einer Filiale sein und muss auch auf genau ein Filmobjekt abgebildet werden können.

- Jede Addresse muss entweder zu einer Person, oder zu einer Filiale abgebildet werden [xor]. (Da dies in der USE-GUI nicht darstellbar ist, ist dieser Sachverhalt über Invarianten beschrieben worden.)

- Jede Person muss genau eine Addresse haben.

- Angestellte dürfen in maximal einer Filiale arbeiten und können Filmkopien an Kunden verleihen. Jeder Angestellte kann zudem eine Reihe von Arbeitstagen zugewiesen bekommen. - Es gilt zu beachten, dass gekündigte Angestellte ggf. zeitweise in der Datenbank erhalten bleiben, wenn Filmkopien, die unter ihren Namen verliehen wurden, noch nicht zurückgebracht wurden, um eventuell noch einmal Rücksprache zu halten. -

- Kunden können sich in mehreren Filialen registrieren lassen und können sich beliebige Filmkopien aus verschiedenen Filialen ausleihen. Dabei muss jede Filmkopie, die entliehen wurde, spätestens binnen 7 Tage wieder in die entsprechende Filiale zurückgebracht werden. Sollte diese Regel verletzt werden, wird der Kunde ermahnt. Sollte der Kunde eine 3. Ermahnung erhalten, wird er gesperrt und darf zukünftig keine Filme mehr entleihen - in keiner der Filialen -.

- Die Periodklasse gibt immer den Tag der Entleihung einer Filmkopie an und setzt das Ende der Leihfrist auf 7 Tage danach.

- Die DateKlasse kann mit einem Zeitobjekt verknüpft werden, was damit zusammenhängt, dass die Filialen sagen könnten, dass der neue Tag bspw. ab 19 Uhr beginnt und der Film dann als einen Tag länger ausgeliehen gilt. Da ein solcher Sachverhalt in USE jedoch unmöglich darzustellen ist, steht die Klasse lediglich noch symbolisch für diesen Sachverhalt da.

- Das CurrentDate Objekt ist ein Singleton und soll von allen Filialen gleichermaßen verwendet werden können, damit die Zeitperiode, über die die Kunden die Filmkopien entleihen dürfen, vernünftig berechnet werden kann.

3 Klassenbeschreibung

Im folgenden Teil sollen nun die einzelnen Klassen im Detail zusammen mit ihren Attributen, Operationen und ihren Bedingungen erläutert werden. Damit stets eine Übersicht über die Klassen vorhanden ist, werden die entsprechenden Programmstücke zu diesem Zweck mit aufgelistet.
Als wichtigen Punkt gilt noch zu erwähnen, dass in den folgenden Programmstücken einige Invarianten direkt an den Klassen selbst deklariert werden und andere werden am Ende des Programms deklariert. In den Programmstücken, wird dies dann wie folgt dargestellt.

```
1  class {classname}
2     [attributes]
3        [...]
4     [operations]
5        [...]
6     [constraints]
7        [inv ...]
8  end
9  ...
10 constraints
11    ...
12    context {classname}
13       [inv ...]
```

Der Grund für diese Aufteilung liegt darin, dass die Invarianten die innerhalb des Constraint-Body's für jedes Objekt getestet einmal getestet werden. Bei einigen Invarianten ist es jedoch sinnvoll, sie nur einmal testen zu lassen, da sonst bei bspw. sehr großer Objekt-Anzahl die Laufzeit bei der Auswertung der Invarianten erheblich beeinträchtigt wird.

Um den Unterschied solcher eben genannten Invarianten untereinander noch einmal zu verdeutlichen, sollen folgende Beispiele dienen, die aus der Klasse Actor stammen, welche später noch genauer erläutert werden.

```
class Actor
   attributes
      ...
   operations
      ...
   constraints
      inv Must_Have_Name: self.name.size > 0
end
...
constraints
   ...
   context a1,a2:Actor
      inv Actor_Objects_Must_be_Unique: a1 <> a2 implies a1.name <> a2.name
```

Die Invariante die innerhalb der Klasse beschrieben ist, greift auf seine Objekt-internen Attribute zu und muss somit für jedes einzelne Objekt getestet werden. Die Invariante aus den Constraints am Ende des Files gilt statisch für die Klasse Actor und muss daher zu jedem Zeitpunkt immer nur einmal getestet werden.
Die Aufteilung um dieses Verhalten zu berücksichtigen, gehört jedoch nicht zum OCL-Standard, es ist stattdessen eine Eigenheit der USE-Umgebung.

3.1 Class Movie

Die Movieklasse ist sozusagen das Herzstück des Systems. Sie repräsentiert Filme und ihre Informationen, welche bspw. für Kundenberatungen oder der Einsortierung in die Regale benötigt werden.

```
class Movie
   attributes
      title: String
      fsk: Integer
      releaseYear: Integer
   operations
      -- F"ugt einen Schauspieler zu diesem Objekt hinzu
      -- Zur Vereinfachung wird jetzt einfach angenommen, dass
      -- keine zwei Schauspieler innerhalb eines Filmes den selben
      -- Namen haben.
      addActor(name: String)
         begin
            declare ac : Actor;
            ac := Actor.allInstances() → select(a | a.name = name).any(true);
            if ac.isUndefined() then
               ac := new Actor;
               ac.name := name;
            end;
            insert(self,ac) into ActorBelongsToMovies;
         end
         pre Actor_Is_Unknown_To_Movie: self.actors → excludes(Actor.allInstances()→
                                          select(a|a.name=name)→any(true))
         post Actor_is_Known_To_Movie: self.actors → includes(Actor.allInstances()→
                                          select(a|a.name=name)→any(true))

      addGenre(name:String)
         begin
            declare g:Genre;
            g := Genre.allInstances() → select(g | g.name = name).any(true);
            if g.isUndefined() then
               g := new Genre;
               g.name := name;
            end;
            insert (self,g) into GenreBelongsToMovies;
```

```
35          end
36          pre Genre_Is_Unknown_To_Movie: self.genre → excludes(Genre.allInstances()→
37                                                  select(a|a.name=name)→any(true))
38          post Genre_is_Known_To_Movie: self.genre → includes(Genre.allInstances()→
39                                                  select(a|a.name=name)→any(true))
40
41      constraints
42          inv Title_Needs_Description: self.title.size > 0
43          inv FSK_Is_Set: self.fsk = 0 or self.fsk = 6 or self.fsk = 12 or self.fsk = 16
44                      or self.fsk = 18
45          -- Ich geh einfach mal davon aus, dass niemand je ein Release von vor Jahr 1000
46          -- eintragen will^^
47          inv ReleaseYear_has_Exactly_4_Digits: self.releaseYear ≥ 1000 and
48                                          self.releaseYear ≤ 3000
49  end
```

3.1.1 Movie-Attribute

title
Der Titel speichert den Namen des Films.

fsk
Das FSK gibt die Altersfreigabe zurück, welche nur die geläufigen Werte 0, 6, 12, 16 und 18 akzeptiert.

releaseYear
Gibt das Jahr an, an dem der Film oder Aufführung etc. das erste mal erschienen ist.

3.1.2 Movie-Operationen

addActor(name:String)
Verlinkt einen Schauspieler der zum übergebenen Namen passt mit dem Filmobjekt. Sollte ein Schauspieler dieses Namens bereits in der Datenbank vorhanden sein, wird ein Link zwischen den beiden Objekten erstellt. Gibt es einen Schauspieler dieses Namens jedoch noch nicht, wird das Objekt erstellt und dann der Link gesetzt.

pre Actor_Is_Unknown_To_Movie
Soll sicherstellen, dass dass dieses Filmobjekt noch nicht mit dem genannten Schauspieler verlinkt ist, wenn die Operation aufgerufen wird.

post Actor_is_Known_To_Movie
Soll sicherstellen, dass der Film im Anschluss der Operation mit dem genannten Schauspieler verlinkt ist.

addGenre(name:String)
addGenre funktioniert analog zur Operation addActor, nur das hier halt mit Genreobjekten gearbeitet wird.

pre Genre_Is_Unknown_To_Movie
Soll sicherstellen, dass dass dieses Filmobjekt noch nicht mit dem genannten Genre verlinkt ist, wenn die Operation aufgerufen wird.

post Genre_is_Known_To_Movie
Soll sicherstellen, dass der Film im Anschluss der Operation mit dem genannten Genre verlinkt ist.

3.1.3 Movie-Bedingungen

inv Title_Needs_Description
Soll sicherstellen, dass ein Titel nicht leer gelassen wird. Als Definition dazu muss ein Titel mindestens ein Zeichen beinhalten.

inv FSK_Is_Set
Die Altersfreigabe muss einen von 5 vordefinierten Werten erhalten (0, 6, 12, 16, 18). Alle anderen Werte sind unzulässig.

inv ReleaseYear_has_Exactly_4_Digits
Dies soll lediglich aussagen, dass die Filme mit Sicherheit nicht vor dem Jahr 1000 und nach dem Jahr 3000 erschienen sind. Zusätzlich wird dadurch auch festgelegt, dass das Attribut nicht undefiniert sein darf.

3.2 Class Actor

Die Klasse Actor stellt die Instanzen der Schauspieler dar, die in einem Film mitspielen können. Hierbei wird darauf geachtet, dass jede Schauspieler Instanz nur ein einziges mal erzeugt wird. Dies ist selbstverständlich eine Vereinfachung, da der Fall, dass 2 Schauspieler mit demselben existieren, nicht berücksichtigt wird. Dies wäre aber hier auch zu vernachlässigen, da lediglich die Namen der Schauspieler interessant sind.

```
1  class Actor
2     attributes
3        name: String
4     operations
5        -- wird verwendet, um einen Schauspieler mit gleichen Namen nicht zweimal zu
             erzeugen.
6        -- Kann im Prinzip zwar falsch sein, aber da beim Schauspieler als Attribut nur der
7        -- Name betrachtet wird, kann es mir auch egal sein.^^
8        equals(a:Actor) : Boolean = if self.name = a.name then true else false endif
9     constraints
10       inv Must_Have_Name: self.name.size > 0
11 end
12 ...
13 constraints
14    context a1,a2:Actor
15       inv Actor_Objects_Must_be_Unique: a1 <> a2 implies a1.name <> a2.name
```

3.2.1 Actor-Attribute

name
Der Name des Schauspielers.

3.2.2 Actor-Operations

equals(a:Actor)
Diese Operation wird dazu verwendet um herauszufinden, ob es bereits ein Schauspieler-objekt mit demselben Namen gibt, wie jenes, das als Parameter übergeben wird.

3.2.3 Actor-Bedingungen

inv Must_Have_Name
Das Schauspielerobjekt muss in jedem Falle mit einem Namen ausgestattet werden.

inv Actor_Objects_Must_be_Unique
Jeder Schauspielername soll nur ein einziges mal in der Datenbank auftauchen.

3.3 Class Genre

Die Klasse Genre verhält sich ähnlich wie die Klasse Actor. Ein Unterschied ist, dass Genreobjekte auch dann noch valide in der Datenbank liegen können, wenn diese nicht mit Filmobjekten verbunden sind.
Außerdem ist einer der Gründe, wieso diese Klasse extra entworfen wurde, der um tausendfache Datenredundanz bei den Filmobjekten zu vermeiden, wenn jeder Film seine Genres selber in einer Liste speichern würde.

```
1  class Genre
2     attributes
3        name:String
4     operations
5        equals(g:Genre) : Boolean = self.name = g.name
6     constraints
7        inv Must_Have_Name: self.name.size > 0
8  end
9  ...
10 constraints
11    context g1,g2 : Genre
12       inv Genre_Objects_Must_be_Unique: g1 <> g2 implies g1.name <> g2.name
```

3.3.1 Genre-Attribute

name
Der Name des Genres.

3.3.2 Genre-Operationen

equals(g:Genre)
Diese Operation wird dazu verwendet um herauszufinden, ob es bereits ein Genreobjekt mit demselben Namen gibt, wie jenes, das als Parameter übergeben wird.

3.3.3 Genre-Bedingungen

inv Must_Have_Name
Das Genreobjekt muss in jedem Falle mit einem Namen ausgestattet werden.

inv Genre_Objects_Must_be_Unique
Jedes Gemre soll nur ein einziges mal in der Datenbank auftauchen.

3.4 Class Company

Die Firmenklasse ist quasi das Superobjekt für alle anderen. Eine Firma soll über den gesamten Betrieb einen überblick behalten können, weshalb sich jeder Faden zur Firma zurückführen lässt.

```
class Company
   attributes
      name: String
   operations
      -- Eine Filiale bekommt einen Namen, eine Liste von Filmkopien,
      -- die ein Subset der Filmliste der Firma ist und als ersten
      -- angestellten den Filialleiter.
      createFilial(name:String, address: Address, movies: Set(Movie), manager: Employee)
         begin
            declare f : Filial, mc: MovieCopy;
            f := new Filial;

            insert(self,f) into CompanyHasManyFilials;

            f.name := name;
            insert(f,manager) into EmployeeWorksInFilial;

            for m in movies do
               f.requestMovieCopy(m);
            end;

            insert(f,address) into FilialHasAddress;

            f.redistributeWorkingDaysOnEmployees();
         end
         pre Filial_Must_Get_A_Name: name.size > 0
         -- Selbst damit lohnt sich die Filiale kaum.^^
         pre Movies_Must_Have_At_Least_50_Entries: movies→size ≥ 50
         pre Movies_Is_Subset_Of_Companies_Movies: movies→includesAll(movies)
         post One_More_Filial_After_Add: (self.filials→size - self.filials@pre→size) =
             1

      -- Da der Parameter filial, wenn er gel"oscht wird, auf die pre-Condition nicht
          mehr
      -- abgefragt werden kann, gibt es in dieser Funktion einen weiteren
          Operationsaufruf,
      -- der als Workaround dienen soll.
      closeFilial(filial: Filial)
         begin
            self.closeFilialPreparation(filial);
            destroy filial;
         end
         pre Filial_Does_Exist_Before_Remove: self.filials→includes(filial)
         post One_Less_Filial_After_Remove: (self.filials@pre→size -
                                    self.filials→size) = 1

      -- Diese Operation soll als workaround dienen, um das Problem mit dem gel"oschten
      -- Parameter aus der Funktion "closeFilial" zu l"osen.
      closeFilialPreparation(filial:Filial)
         begin
            filial.deleteMePreparations();
            for employee in filial.worker do
               employee.deleteMePreparations();
               destroy employee;
            end;
            for copy in filial.movies do
               filial.sellMovieCopy(copy);
            end;
            for customer in filial.customer do
               delete (filial,customer) from CustomerHasContactToFilial;
               if (customer.serviceProvider → size = 0) then
                  destroy customer;
               end;
```

```
61              end;
62          end
63          post Employees_Got_Fired: Employee.allInstances()→ excludesAll(filial@pre.
                worker)
64          post Customer_With_No_Other_ServiceProviders_got_Deleted:
65              Customer.allInstances()→ forAll(c | c.serviceProvider → size > 0 and
66                                          c.serviceProvider→excludes(filial))
67          post Address_Of_Filial_Got_Deleted: filial.address.isUndefined()
68
69      addMovie(movie: Movie)
70          begin
71              insert (self,movie) into CompanyHasRightToConferCopiesOfMovies
72          end
73          pre Company_Has_No_Rights_Yet: self.movies → excludes(movie)
74          post Company_Now_Has_Rights: self.movies → includes(movie)
75
76    constraints
77       inv Must_Have_Name: self.name.size > 0
78 end
79 ...
80 constraints
81    context c1,c2 : Company
82       inv Name_Must_Be_Unique: c1 <> c2 implies c1.name <> c2.nam
```

3.4.1 Company-Attribute

name

Jede Firma braucht ihren eignenen einzigartigen Namen.

3.4.2 Company-Operationen

createFilial(name:String, address: Address, movies: Set(Movie), manager: Employee)

Eine Firma kann beliebig viele Filialen eröffnen, muss einer Filiale allerdings eine Starthilfe mit zur Verfügung stellen, damit diese beginnen kann, sich selbstständig zu verwalten. Dazu gehören ein Name den die Filiale erhalten soll, ein Standort um zu wissen, wo die Filiale eröffnet wird, eine Liste von Filmen, von denen entsprechende Kopien an die Filiale mitgegeben werden, welche diese dann verleihen darf. Und zu guter letzt braucht die Filiale mindestens einen Angestellten, der die Filiale leitet und andere Personen für diese Filiale anstellen darf.

pre Filial_Must_Get_A_Name

Soll sicherstellen, dass für den Namen der Filiale kein leerer String übergeben wird.

pre Movies_Must_Have_At_Least_50_Entries

Soll sicherstellen, dass die übergebene Liste von Filmen, die an die Filiale übergeben wird, mindestens 50 Filme beinhält.

pre Movies_Is_Subset_Of_Companies_Movies

Soll sicherstellen, dass die übergebene Liste von Filmen, im Rechtebereich der Firma liegt.

post One_More_Filial_After_Add

Soll sicherstellen, dass die Filiale nach Ausführung der Operation auch wirklich existiert.

closeFilial (filial : Filial)

Soll eine Filiale schließen, die irgendwann einmal eröffnet wurde.

pre Filial_Does_Exist_Before_Remove

Soll sicherstellen, dass die Filiale die diese Firma schließen möchte auch wirklich zur eigenen Firma gehört.

post One_Less_Filial_After_Remove

Nach der Ausführung der Operation soll es eine Filiale weniger geben, die zu dieser Firma gehört.

closeFilialPreparation (filial : Filial)

Diese Operation dient als ein Workaround und soll niemals vom Benutzer aufgerufen werden. Der einzige Aufruf dieser Operation soll einzig und allein innerhalb der Operation „closeFilial“ stattfinden dürfen.

Da bei der Schließung einer Filiale die Angestellten entlassen werden, ggf. Kunden aus der Datenbank entfernt und die vorhandenen Filmkopien innerhalb der Filiale verkauft werden müssen, soll dies in einigen Nachbedingungen festgelegt werden. Da der Parameter jedoch nach seiner Löschung nicht mehr erreichbar ist, dient diese Funktion als Workaround um diese Nachbedingungen zu überprüfen, bevor das Filialobjekt gelöscht wird.

post Employees_Got_Fired

Soll sicherstellen, dass die Angestellten der zu schließenden Filiale nicht mehr in der Datenbank gelistet sind.

post Customer_With_No_Other_ServiceProviders_got_Deleted

Soll sicherstellen, dass ein Kunde, der in keiner anderen Filiale gelistet ist, als in der zu Schließenden, aus der Datenbank entfernt wurde.

post Address_Of_Filial_Got_Deleted

Soll sicherstellen, dass die Adresse, an der die zu schließende Filiale zu finden war, wieder frei gegeben wird.

addMovie (movie : Movie)
Diese Operation wird aufgerufen, wenn die Firma die Rechte für einen neuen Film erworben hat.

pre Company_Has_No_Rights_Yet
Soll sicherstellen, dass die Firma zum Aufruf dieser Operation die Rechte für den entsprechenden Film noch nicht hat.

post Company_Now_Has_Rights
Soll sicherstellen, dass die Firma nach dem Aufruf dieser Operation die Rechte für den entsprechenden Film hat.

3.4.3 Company-Bedingungen

inv Must_Have_Name
Jede Firma muss einen Namen haben.

inv Name_Must_Be_Unique
Jede Firma trägt die Rechte an ihrem einzigartigen Namen.

3.5 Class Filial

Die Filialenklasse repräsentiert einen physischen Standort einer Firma. Dabei mag dies nur ein kleiner Teil der Firma sein, oder bei einer kleinen Firma auch gleich die ganze Firma. Eine Filiale ist dafür zuständig, dass Leute eingestellt werden, Kunden angeworben werden und das weitere Filmkopien in die Filiale kommen. Der erste Satz, den eine Filiale erhält muss nicht unbedingt der ganze Listensatz der Firma sein. Entsprechend muss eine Firma sich im Anschluss selber um weiteren Nachschub bemühen.

```
1  class Filial
2     attributes
3        name: String
4     operations
5        requestMovieCopy(movie:Movie)
6           begin
7              declare mc:MovieCopy;
8              mc := new MovieCopy;
9              -- Ich wei"s, das ist nicht ganz korrekt, aber f"ur USE hier ist das gerade
                   eine
10             -- Notl"osung, die ich leider gerade nur auf umst"andliche Art und Weise zu
11             -- korrigieren wei"s, was erstmal nicht sein muss, daher muss die Notl"osung
12             -- reichen.
13             mc.ID := MovieCopy.allInstances()→size+1;
14             insert(movie,mc) into MovieCopyIsFromMovie;
15             insert(self,mc) into FilialMustPossessMovieCopies
16          end
17          pre Company_Must_Have_Movie_Rights: self.creator.movies→includes(movie)
18
19       getWorkingEmployee() : Employee = self.worker→select(e:Employee | e.status = #
             employed
20                                                and e.worksOnDate(
21                                               CurrentDate.allInstances().any(true) ) )→
22                                            asSequence()→getRandom().oclAsType(Employee)
23
24       -- gibt undefined zur"uck, wenn es keine verf"ugbaren Kopien in der Filiale gibt.
25       getAvailableCopy(movie:Movie) : MovieCopy =
26                      -- schau nach, welche Kopien in dieser Filiale vorhanden sind
27                      (movie.copy→select(m:MovieCopy | m.owner→includes(self)))→
28                      -- nun w"ahle alle Filme aus, die noch nicht entliehen sind.
29                      select(m:MovieCopy | m.isAvailable()).any(true);
```

```
30
31      sellMovieCopy(movie:MovieCopy)
32         begin
33            destroy movie;
34         end
35         post There_Must_Be_One_Less_MovieCopy_In_Filial: self.movies@pre→size -
36                                                self.movies→size = 1
37
38      employ(identityNumber: String, fname: String, lname:String, phonenumber:String,
39            street: String, housenumber:String, postcode: String, town: String)
40         begin
41            declare employee: Employee, address: Address;
42            employee := new Employee;
43            employee.status := #employed;
44            employee.identityNumber := identityNumber;
45            employee.firstname := fname;
46            employee.lastname := lname;
47            employee.phonenumber := phonenumber;
48            insert (self,employee) into EmployeeWorksInFilial;
49
50            address := new Address;
51            address.street := street;
52            address.housenumber := housenumber;
53            address.postcode := postcode;
54            address.town := town;
55            insert (employee,address) into PersonHasAddress;
56            self.distributeWorkingDaysOnEmployees();
57         end
58         pre All_Parameters_Must_Be_Valid: identityNumber.size > 0 and
59                                   fname.size > 0 and lname.size > 0 and
60                                   phonenumber.size > 0 and
61                                   street.size > 0 and housenumber.size > 0 and
62                                   postcode.size > 0 and town.size > 0
63         post A_New_Employee_Has_Been_Added: self.worker→size - self.worker@pre→size =
               1
64
65      -- bitte nicht versuchen, diese Implementierung nachzuvollziehen. Hier ist aufgrund
             der
66      -- Restriktionen von soil und ocl ein sehr h"asslicher workaround entstanden. Es
67      -- funktioniert und das sollte reichen ^^
68      distributeWorkingDaysOnEmployees()
69         begin
70            -- richte den ersten Tag des neuen Monats, dessen Tage verteilt werden sollen
                  ,
71            -- ein.
72            declare d:Date, d2:Date, pDate: Date, helpSet:Sequence(Integer),
73                                          helpSet2: Sequence(Integer), e: Employee;
74            d := new Date;
75            d.copyDateIntoSelf(CurrentDate.allInstances().any(true));
76            d.Day := 1;
77            d.Month := d.Month+1;
78            if d.Month > 12 then
79               d.Month := 1;
80               d.Year := d.Year+1;
81            end;
82            d2 := new Date;
83            d2.copyDateIntoSelf(d);
84            d2.Month := d2.Month+1;
85            if d2.Month > 12 then
86               d2.Month := 1;
87               d2.Year := d2.Year+1;
88            end;
89
90            -- verteile die Arbeitstage auf die Angestellten.
91            -- 31 einsen
92            helpSet := Sequence{1,1,1,1,1,1,1,1,1,1,1,
93                           1,1,1,1,1,1,1,1,1,1,1,
94                           1,1,1,1,1,1,1,1,1,1,1};
95            -- 3 einsen
96            helpSet2 := Sequence{1,1,1};
97            -- laufe durch 31 Tage
```

```
98              for z in helpSet do
99                 if (d.before(d2)) then
100
101                    pDate := new Date;
102                    pDate.copyDateIntoSelf(d);
103                    for z2 in helpSet2 do
104                       e := self.worker→asSequence()→getRandom().oclAsType(Employee);
105                       if (not e.worksOnDate(pDate)) then
106                          insert (e,pDate) into EmployeeWorksOnDates;
107                       end
108                    end;
109                    d.addDay();
110                 end;
111              end;
112              destroy Date.allInstances()→select(d| d.equals(d2)
113                                 or d.before(CurrentDate.allInstances()→any(true)));
114              -- wird nicht anerkannt und ist ein Fehler, warum auch immer.
115              -- self.worker→iterate(e: Employee, es: Set(Employee) | es.including(e));
116           end
117           -- w"urde ich gerne machen, aber daf"ur m"usste ich zuverl"assig den n"achsten
                  Monat
118           -- bestimmern k"onnen in dieser condition, was ich in OCL leider nicht tun kann.
119           -- post Days_Of_Next_Month_Must_Be_Distributed: ...
120
121        -- verteilt die Tage des aktuellen Monats nochmal neu auf alle Angestellten
122        redistributeWorkingDaysOnEmployees()
123           begin
124              declare thisDate: CurrentDate;
125              thisDate := CurrentDate.allInstances()→any(true);
126              thisDate.Month := thisDate.Month-1;
127              if (thisDate.Month < 1) then
128                 thisDate.Month := 12;
129                 thisDate.Year  := thisDate.Year - 1;
130              end;
131              self.distributeWorkingDaysOnEmployees();
132              thisDate.Month := thisDate.Month + 1;
133              if (thisDate.Month > 12) then
134                 thisDate.Month := 1;
135                 thisDate.Year  := thisDate.Year + 1;
136              end;
137           end
138
139        fireEmployee(employee:Employee)
140           begin
141              if employee.lentMovies→size > 0 then
142                 employee.status := #notEmployed;
143              else
144                 employee.deleteMePreparations();
145                 destroy employee;
146              end
147           end
148           -- sichert den Zweig im else-Fall der postcondition ab, damit es nur
                  funktioniert,
149           -- wenn das Objekt bei der "ubergabe definiert war.
150           pre Employee_Parameter_Must_Be_Defined: employee.isDefined()
151           -- pre und postconditions schwierig hier, da diese ggf. auf ein undefined object
152           -- zugreifen w"urden. Daher bin ich ein wenig eingeschr"ankt.
153           post Employee_Is_No_Longer_Employed:    if employee.isDefined() then
154                                                      employee.status = #notEmployed
155                                                   else
156                                                      true
157                                                   endif
158
159        createCustomer(identityNumber: String, fname: String, lname:String, phonenumber:
               String,
160                       street: String, housenumber:String, postcode: String, town: String)
161           begin
162              declare customer:Customer, address: Address;
163
164              customer := Customer.allInstances()→ select(c | c.identityNumber =
165                                                      identityNumber)
```

```
                                                    →any(true);
            address := Address.allInstances()→select(a | a.street = street and
                                                a.housenumber = housenumber and
                                                a.postcode = postcode and
                                                a.town = town)→any(true);
            if customer.isUndefined() then
               customer := new Customer;
               customer.identityNumber := identityNumber;
               customer.firstname := fname;
               customer.lastname := lname;
               customer.phonenumber := phonenumber;
               customer.status := #approved;
               customer.penalties := 0;
               -- Wenn die eben erstellte Addresse bereits existiert, soll der neue Kunde
               -- mit eben dieser Addresse verlinkt werden.
               if address.isUndefined() then
                  address := new Address;
                  address.street := street;
                  address.housenumber := housenumber;
                  address.postcode := postcode;
                  address.town := town;
               end;

            else
               -- Egal ob der Kunde umgezogen ist, oder eine neue Addresse hat, die
               -- Addresse wir im n"achsten Schritt neu zugeordnet.
               address := customer.address;
               delete (customer,customer.address) from PersonHasAddress;
            end;

            insert (customer,address) into PersonHasAddress;
            insert (self,customer) into CustomerHasContactToFilial;
         end
         pre Identity_Number_Must_Not_Already_Exist_In_This_Filial: self.customer→forAll
             (
                                        c | c.identityNumber <> identityNumber)
         post A_New_Customer_Has_Been_Added: self.customer→size-self.customer@pre→size
             = 1
         post Do_Not_Create_A_Customer_Twice: Customer.allInstances()→
                                    forAll(c1,c2| c1<>c2 implies not c1.equals(c2))

      deleteMePreparations()
         begin
            destroy self.address;
         end
   constraints
      inv Must_Have_Name: self.name.size > 0
      inv Must_Have_At_Least_50_Different_MovieCopies:
                                       self.movies.original→asSet()→size ≥ 50

end
```

3.5.1 Filial-Attribute

name
: Der Name der Filiale. Dieser Name muss nicht zwangsläufig mit dem Namen der Firma übereinstimmen.

3.5.2 Filial-Operationen

requestMovieCopy (movie : Movie)

Mit dieser Operation kann eine neue Filmkopie für eine Filiale angefordert werden. Dabei ist es egal, ob es sich um eine Kopie hat, die bereits in der Filiale vorhanden ist, oder um eine Kopie, die biser noch gar nicht in der Filiale gelistet ist.

pre Company_Must_Have_Movie_Rights

Soll sicherstelen, dass die Firma, zu der diese Filiale gehört, auch die notwenidgen Rechte besitzt, um Kopien von diesem Film zu verleihen.

getWorkingEmployee ()

Sucht per Zufall einen Mitarbeiter aus, der am aktuellen Tag (nach der Klasse CurrentDate) arbeitet und gibt diesen Mitarbeiter zurück, wenn er denn Angestellt ist.

sellMovieCopy (movie : MovieCopy)

Verkauft die Kopie eines Filmes, falls eine Filiale sich davon trennen möchte und löscht diese anschließend aus der Datenbank. (Hier werden keine Gewinnspannen oder andere Geldoperationen mit einbezogen)

post There_Must_Be_One_Less_MovieCopy_In_Filial

Soll sicherstelen, dass nach dem Aufruf der Operation eine Filmkopie weniger als zuvor in dieser Filiale existiert.

employ(identityNumber: String, fname: String, lname:String, phonenumber:String, street: String, housenumber:String, postcode: String, town: String)

Mit dieser Operation soll eine neue Person in dieser Filiale angestellt werden.

post All_Parameters_Must_Be_Valid

Soll sicherstellen, dass alle Parameter definiert und beschrieben sind, damit keine ungültigen Zustände entstehen.

post A_New_Employee_Has_Been_Added

Soll sicherstellen, dass nach dem Aufruf dieser Operation ein neuer Angestellter in dieser Filiale arbeitet.

distributeWorkingDaysOnEmployees ()

Soll die Arbeitstage des aktuellen Monats auf die Angestellten verteilen.

redistributeWorkingDaysOnEmployees ()

verteilt die Tage des aktuellen Monats auf die Angestellten, wenn eine Filiale gerade eröffnet wird.

(Hier muss leider noch erwähnt werden, dass die Implementierung scheiterte und nicht so umzusetzen war, wie gewollt. Für jeden Angestellten werden jetzt extra Date-Objekte angelegt und diese dann verlinkt, anstatt die Bestehenden zu nehmen. Beim Versuch die andere Version zu implementieren, trat ein seltsames Verhalten auf, dass nicht zu erklären war, weshalb diese Version als Notlösung dienen soll.)

fireEmployee (employee : Employee)
Soll einen Kunden aus dem Dienst der Filiale entlassen. Sollte ein Angestellter jedoch noch in Verbindung mit einem Kunden stehen, dem er Filme entliehen hat, wird seine Instanz noch nicht gelöscht. Dies geschieht erst dann, wenn all diese Beziehungen aufgelöst sind.
pre Employee_Parameter_Must_Be_Defined
Soll sicherstellen, dass der entsprechende Angestellte der übergeben wird, nicht undefined ist.
post Employee_Is_No_Longer_Employed
Soll sicherstellen, dass der Angestellte keinen Angestellten Status mehr besitzt, oder aus der Datenbank entfernt wurde.

createCustomer (identityNumber : String , fname : String , lname :String , phonenumber:String, street : String ,housenumber :String , postcode : String , town : String)
Soll einen neuen Kunden in der Datenbank vermerken. Wenn dieser Kunde jedoch schon in einer anderen Filiale gelistet ist, wird lediglich ein neuer Link erstellt.
pre Identity_Number_Must_Not_Already_Exist_In_This_Filial
Soll sicherstellen, dass der selbe Kunde nicht zweimal in der gleichen Filiale gelistet wird.
post A_New_Customer_Has_Been_Added
Soll sicherstellen, dass nach der Ausführung ein neuer Kunde in der Filiale gelistet ist.
post Do_Not_Create_A_Customer_Twice
Soll sicherstellen, dass der selbe Kunde im Anschluss nicht zweimal in der Datenbank steht.

deleteMePreparations ()
Soll als Hilfsoperation dienen um das Filialobjekt zu löschen.

3.5.3 Filial-Bedingungen

inv Must_Have_Name
Soll sicherstellen, dass der Name der Filiale definiert ist und keine leere Zeichenkette ist.

inv Must_Have_At_Least_50_Different_MovieCopies
Soll sicherstellen, dass diese Filiale mindestens 50 Kopien von unterschiedlichen Filmen aufweist.

3.6 Class Address

Diese Klasse soll den Wohnort verschiedener Personen, oder den Aufenthaltsort einer Filiale beschreiben. Dabei kann eine Adresse zu mehreren Personen gehören, oder zu einer einzigen Filiale. Beides ist nicht möglich.

```
1  class Address
2    attributes
3       street: String
4       housenumber: String
5       postcode: String
6       town: String
7    operations
8       equals(a:Address) : Boolean =
9          self.street.toLowerCase() = a.street.toLowerCase() and
10         self.housenumber.toLowerCase() = a.housenumber.toLowerCase() and
```

```
11          self.postcode.toLowerCase() = a.postcode.toLowerCase() and
12          self.town.toLowerCase() = a.town.toLowerCase()
13    constraints
14       inv Sreet_Must_Have_Name: self.street.size > 0
15       inv Housenumber_Must_Have_Definition: self.housenumber.size > 0
16       inv Postcode_Must_Be_Defined: self.postcode.size > 0
17       inv Town_Must_Have_Name: self.town.size > 0
18       inv Belongs_To_Person_Or_Filial: self.filial.isDefined() xor self.persons→size > 0
19 end
20 ...
21 constraints
22    ...
23    context a1,a2 : Address
24       inv Address_Must_Be_Unique: a1 <> a2 implies not a1.equals(a2)
```

3.6.1 Address-Attribute

street
Der Name der Straße des entsprechenden Wohnortes.

housenumber
Die Hausnummer innerhalb der Straße des entsprechenden Wohnortes.

postcode
Die Postleitzahl des entsprechenden Wohnortes.

town
Der Ort bzw. die Stadt in der sich die entsprechende Addresse befindet.

3.6.2 Address-Operationen

equals(a:Address)
überprüft, ob diese Addresse mit einer anderen identisch ist.

3.6.3 Address-Bedingungen

inv Sreet_Must_Have_Name
Das Attribut „street“ muss eine Bezeichnung haben.

inv Housenumber_Must_Have_Definition
Das Attribut „housenumber“ muss gesetzt sein.

inv Postcode_Must_Be_Defined
Das Attribut „postcode“ muss gesetzt sein.

inv Town_Must_Have_Name
Das Attribut „town“ muss gesetzt sein.

inv Belongs_To_Person_Or_Filial
Stellt sicher, dass die Addresse nur zu einer Filiale gehört, oder zu einer bis mehreren Personen. Aber zu einem von beiden muss die Addresse in jedem Falle gehören.

inv Address_Must_Be_Unique
Kein Addressenobjekt darf ein Duplikat von sich in der Datenbank aufweisen.

3.7 Class Person

Die Klasse Person ist die Superklasse zu Employee und Customer und stellt damit sicher, dass diese Klassen eine Addresse, einen vollständigen Namen und eine eindeutige Identifikationsnummer erhalten, was in diesem Fall die Personalausweisnummer sein soll.

```
1  class Person
2     attributes
3        identityNumber: String
4        firstname: String
5        lastname: String
6        phonenumber: String
7     constraints
8        inv Must_Have_IdentityNumber: self.identityNumber.size > 0
9        inv Must_Have_First_Name: self.firstname.size > 0
10       inv Must_Have_Last_Name: self.lastname.size > 0
11       inv Must_Have_Address: self.address.isDefined()
12       -- ich geh mal davon aus, dass jede Telefonnummer mindestens 3 Zeichen enth"alt
13       inv Must_Have_Phnonenumber: self.phonenumber.size ≥ 3
14 end
15 ...
16 constraints
17    context p1,p2 : Person
18       inv IdentityNumber_Is_Unique: p1<>p2 implies p1.identityNumber <> p2.identityNumber
```

3.7.1 Person-Attribute

identityNumber
Die Personalausweisnummer einer Person, mit der sich die Person eindeutig identifizieren lässt.

firstname
Der Vorname einer Person.

lastname
Der Nachname einer Person.

phonenumber
Die Telefonnummer einer Person, über die diese Person zuhause zu erreichen ist, falls ein Anliegen eines Dienstanbieters auftritt.

3.7.2 Person-Operationen

Die Klasse Person benötigt keine Operationen.

3.7.3 Person-Bedingungen

inv Must_Have_IdentityNumber
Es ist absolut notwendig, dass die Identifikationsnummer definiert ist!

inv IdentityNumber_Is_Unique
Keine Identifikationsnummer kann zweimal auftreten. Andernfalls muss ein Fehler vorliegen, da jede Nummer eindeutig auf eine Person zurückzuführen ist.

inv Must_Have_First_Name
Der Vorname muss angegeben werden.

inv Must_Have_Last_Name
Der Nachname muss angegeben werden.

inv Must_Have_Address
Die Addresse muss angegeben werden.

inv Must_Have_Phnonenumber
Jede Telefonnummer muss mindestens 3 Zeichen enthalten.

3.8 Class MovieCopy

Die Filmkopie Klasse stellt die Objekte dar, die eine Filiale physisch an die Kunden weiterverleihen darf. Dabei darf eine Filiale von jedem Film über diverse Filmkopien verfügen.
Jede Filmkopie wird vermerkt, wenn sie an einen Kunden verliehen wird. Dabei werden Das Ausleihdatum das Enddatum - also wann der Kunde den Film spätestens zurückgebracht haben muss -, der Kunde selbst und der Angestellte der ihn bedient hat, zu dieser Filmkopie

gespeichert.

```
class MovieCopy
   attributes
      ID: Integer
   operations
      isAvailable() : Boolean = self.period→size = 0
   constraints
      inv If_Borrowed_Copy_Is_Linked_With_Exactly_One_Customer_And_Employee:
                  if self.period→size ≥ 1 then
                     self.borrowedBy→size = 1 and
                     self.conferedBy→size = 1
                  else
                     true
                  endif
end
...
constraints
   context mc1, mc2: MovieCopy
      inv ID_Must_Be_Unique: mc1 <> mc2 implies mc1.ID <> mc2.ID
```

3.8.1 MovieCopy-Attribute

ID
: Eine eindeutige Identifikationsnummer, über die diese Filmkopie ihrer Filiale und ggf. dem Kunden, der die Filmkopie entliehen hat, zugeordnet werden kann.

3.8.2 MovieCopy-Operationen

isAvailable()
: überprüft, ob diese Filmkopie aktuell verliehen ist, oder ob sie noch verliehen werden kann.

3.8.3 MovieCopy-Bedingungen

inv If_Borrowed_Copy_Is_Linked_With_Exactly_One_Customer_And_Employee
: Wenn diese Filmkopie verliehen wurde, muss es genau einen Kunden und genau einen Angestellten geben, die mit diesem Objekt verlinkt sind.

inv ID_Must_Be_Unique
: Jede Filmkopie muss eine einzigartige ID haben, die auch in den anderen Filialen nicht wiederzufinden ist.

3.9 Class Employee

Die Angestellten-Klasse stellt die Personen dar, die in einer Filiale arbeiten. Dabei ist es jeder Person lediglich erlaubt in einer Filiale zu arbeiten.

```
class Employee < Person
   attributes
      status: EmployeeStatus
   operations
      worksOnDate(date:Date) : Boolean = self.workingDates→collect(d:Date | d.equals(
          date))→
                                 excluding(false)→size = 1
      deleteMePreparations()
```

```
        begin
            if self.address.persons→size = 1 then
                destroy self.address;
            end;
        end
    constraints
        inv Status_Must_Be_Set: self.status.isDefined()
        inv Workplace_Should_Be_Defined: self.employer.isDefined()
end
```

3.9.1 Employee-Attribute

status
: Gibt den Status dieser Person an, ob sie angestellt ist, oder nicht. Diese Variable ist dazu notwendig, dass wenn Angestellte noch Links zwischen Filmkopien, die sie an Kunden verliehen haben, nicht gelöscht werden, solange diese Filmkopien nicht zurückgebracht wurden, damit weiter Rücksprache gehalten werden kann.

3.9.2 Employee-Operationen

worksOnDate(date:Date)
: Eine einfache Abfrage, die aussagt, ob der Angestellte am entsprechenden Tag arbeitet, oder nicht.

deleteMe()
: Soll einen Angestellten löschen und das dazugehörige Addressenobjekt, sofern dieses mit keiner weiteren Person verbunden ist.

3.9.3 Employee-Bedingungen

inv Status_Must_Be_Set
: Soll sicherstellen, dass der Status des Angestellten zu jederzeit definiert ist, womit auch zeitgleich sichergestellt ist, dass der Wert valide ist, da es nur zwei Werte gibt.

inv Workplace_Should_Be_Defined
: Soll sicherstellen, dass der Angestellte zu jeder Zeit über seinen Arbeitsplatz bzw. Ex-Arbeitsplatz abgerufen werden kann.

3.10 Class Customer

Der Kunde ist wichtig für die Filialen, denn jede Filiale braucht Kunden, an die sie ihre Filme verleihen kann. Somit ist ein Kunde dazu im Stande, in beliebig vielen Filialen als Kunde aufzutreten und in jeder dieser Filialen beliebig viele Filme zu entleihen.

```
class Customer < Person
    attributes
        -- Angabe, wie oft die maximale Leihfrist von 6 Tagen "uberschritten wurde.
        penalties: Integer
        /* Wenn die Maximumleihfrist 3 mal "uberschritten wurde, wird der Kunde gesperrt
           und darf nichts mehr entleihen */
        status: CustomerStatus
```

```
8   operations
9      equals(c:Customer) : Boolean = self.identityNumber = c.identityNumber
10
11     returnMovies(movies:Set(MovieCopy), filial:Filial)
12        begin
13           /* Wenn Filme zur"uckgebracht werden, m"ussen folgende Punkte beachtet werden
                :
14            * 1. Wurde der Film innerhalb der Befristung zur"uckgebracht?
15            * 2. Wurden "uberhaupt alle Filme zur"uckgebracht? Sind die, die noch nicht
16            *   zur"uckgebracht worden, noch innerhalb der Befristung?
17            * 3. Aufpassen, dass alle Links und "uberfl"ussigen Objekte hierzu gel"oscht
18            *      werden und dass die MovieCopy und der Employee wieder frei gegeben
19            *      wurden.
20            * 4. Wenn Ein Angestellter gefeuert wurde und er noch in der Datenbank
21            *      vorhanden ist und die Filme, mit denen er in Verbindung steht, nun
                  alle
22            *   wieder zur"uck sind, kann dieser Angestellte gel"oscht werden.
23            * 5. Wurden die Filme in die richtige Filiale zur"uckgebracht? (precondition
                  )
24            */
25           declare counter: Integer;
26           counter := 0;
27           for m in movies do
28              -- Wurde der Film innerhalb der Befristung zur"uckgebracht?
29              if CurrentDate.allInstances()→any(true).
30                                          after(m.period→any(true).ends) then
31                 -- Film wurde zu sp"at zur"uckgebracht.
32                 -- Kunde soll hier nur einmal ermahnt werden und nicht f"ur jeden
33                 -- geliehenen Film.
34                 if counter = 0 then
35                    self.penalties := self.penalties + 1;
36                    counter := counter + 1;
37                 end;
38                 if self.penalties = 3 then
39                    self.status := #expelled;
40                 end;
41              end;
42              -- zerstÃ¶re die start und ends Date-Objekte aus den Period Klassen.
43              destroy m.period→any(true).start;
44              destroy m.period→any(true).ends;
45              -- Die Links der ternaeren Beziehung werden aufgeloest
46              delete (m,m.conferedBy→any(true),self) from Period;
47           end;
48        end
49        pre Movies_Must_Belong_To_Visited_Filial: filial.movies→includesAll(movies)
50        post If_Movies_Came_Back_Too_Late_For_Third_Time_Cusomer_Got_Expelled:
51              if (self.penalties = 3) then
52                 self.status = #expelled
53              else
54                 self.status = #approved
55              endif
56
57     borrowMovies(movies:Set(Movie), filial:Filial)
58        begin
59           -- erst muss "uberpr"uft werden, ob die gew"unschten Filme auch verf"ugbar
                sind.
60           declare availables : Set(MovieCopy);
61           availables := movies→collect(m:Movie|filial.getAvailableCopy(m))→
62                                                   excluding(null)→asSet();
63           for mc in availables do
64              insert(mc,filial.getWorkingEmployee(),self) into Period;
65              -- setzt die TimePeriod fest, die der Film maximal im Besitz des Kunden
66              -- bleiben darf.
67              mc.period.any(true).initialize();
68           end
69        end
70        pre Customer_Must_Be_Approved: self.status = #approved
71  constraints
72     inv Status_Must_Be_Set: self.status.isDefined()
73     inv Customer_Has_Service_Provider_If_Approved:
74           if (self.status = #approved) then
```

```
75                  self.serviceProvider->size > 0
76              else
77                  self.serviceProvider->size = 0
78              endif
79  end
```

3.10.1 Customer-Attribute

penalties

Dieses Attribut soll die Information über den Kunden geben, wie oft er bereits ermahnt wurde, weil er die Leihfristen überschritten hat. Dies wird auch Filialenübergreifend bestimmt. Das soll bedeuten, wenn er in zwei verschiedenen Filialen eine Strafe erhält, werden im für beide Filialen trotzdem 2 Strafen angerechnet.

status

Der Status eines Kunden ist initial zulässig („approved“), kann aber ausgeschlossen („expelled”)’ werden. Wenn der Kunde 3 Strafen erhalten hat, egal durch welche Filialen, wird er aus allen Filialen ausgeschlossen („expelled“).

3.10.2 Customer-Operationen

equals(c:Customer)

überprüft ob dieser Kunde identisch zu einem anderen ist. Da die Identifikationsnummer dies eindeutig bestimmen soll, reicht die Abfrage über dieses Attribut.

returnMovies(movies:Set(MovieCopy), filial:Filial)

über diese Operation soll der Kunde die Filme wieder zurückbringen, die er sich entliehen hat. Dabei wird darauf geachtet, dass er die einzelnen Filme auch zur richtigen Filiale zurückbringt und zwar die, die gerade besucht wird. Filme aus anderen Filialen werden nicht akzeptiert.

pre Movies_Must_Belong_To_Visited_Filial

Stellt sicher, dass die Filme die zurückgegeben werden sollen, auch alle zur richtigen Filiale gehören.

post If_Movies_Came_Back_Too_Late_For_Third_Time_Cusomer_Got_Expelled

Stellt sicher, dass wenn die Filme zurückgebracht wurden und die Filme zu spät zurückgebracht wurden und die Anzahl der Strafen die 3 erreicht hat, dass der Kunde dann ausgeschlossen wurde, damit er sich in keiner der Filialen mehr Filme ausleihen darf.

borrowMovies (movies :Set(Movie), filial : Filial)

über diese Operation kann der Kunde sich Filme in einer Filiale die er besucht ausleihen.

pre Customer_Must_Be_Approved

Stellt sicher, dass der Kunde sich nur dann Filme ausleihen darf, wenn dieser nicht ausgeschlossen wurde.

3.10.3 Customer-Bedingungen

inv Status_Must_Be_Set
Stellt sicher, dass der Status immer definiert ist und damit einer der beiden entsprechenden enum-Werte angenommen wurde.

inv Customer_Has_Service_Provider_If_Approved
Wenn der Kunde zugelassen ist und in der Datenbank steht, muss er mit mindestens einer Filiale als Kunde verlinkt sein. Wenn er allerdings ausgeschlossen wurde, werden diese Links aufgehoben und er darf nie wieder einen Film ausleihen.

3.11 Associationclass Period

Die Assoziationsklasse Period stellt eine wichtige Information zu jeder Entleihung dar. Eine Entleihung findet zwischen 3 Objekten statt und Period speichert den Zeitraum, über den ein Kunde den entliehenen Film behalten darf - Dies sind 7 Tage -, womit Period zum 4. Objekt der Dreiecksbeziehung wird. Sollte das End-Datum überschritten werden, gibt es eine Strafe für den Kunden.

```
associationclass Period
   between
      MovieCopy[*] role lentMovies
      Employee[0..1] role conferedBy
      Customer[0..1] role borrowedBy
   attributes
      start : Date
      ends : Date
   operations
      initialize()
         begin
            self.start := new Date;
            self.start.copyDateIntoSelf(CurrentDate.allInstances().any(true));
            self.ends := new Date;
            self.ends.copyDateIntoSelf(self.start);
            self.ends.addDays(7);
         end
   constraints
      inv Start_Must_Be_Defined: self.start.isDefined()
      inv Ends_Must_Be_Defined: self.ends.isDefined()
      inv CurrentDate_Must_Be_Before_Or_Equals_Ends:
                           self.ends.after(CurrentDate.allInstances().any(true)) or
                           self.ends.equals(CurrentDate.allInstances().any(true))
      --inv
      -- H"atte gerne noch eine Invariante, die aussagt, dass Ends immer 7 Tage nach
          start ist.
      -- l"asst sich aber in OCL nicht richtig ausdr"ucken...
end
```

3.11.1 Period-Attribute

start
Das Datum, an dem der entsprechende Film entliehen wurde.

ends
Das Datum an dem der entsprechende Film spätestens zurückgebracht werden muss.

3.11.2 Period-Operationen

initialize()
Initialisiert die Klasse mit den richtigen Werten, welche von der Klasse CurrentDate abhängen.

3.11.3 Period-Bedingungen

inv Start_Must_Be_Defined
Stellt sicher, dass das Startdatum definiert ist.

inv Ends_Must_Be_Defined
Stellt sicher, dass das Enddatum definiert ist.

inv CurrentDate_Must_Be_Before_Or_Equals_Ends
Wenn ein Kunde dabei ist einen Film zu spät zurückzubringen, ist diese Invariante verletzt. Also wenn der aktuelle Tag das Enddatum von Period überschreitet.

3.12 Class Date

Die Klasse Date gibt lediglich ein beliebiges Datum an.

```
class Date
   attributes
         Day : Integer
         Month : Integer
         Year : Integer
   operations
      equals(d : Date) : Boolean =
         self.Year = d.Year and self.Month = d.Month and self.Day = d.Day
      after(d : Date) : Boolean =
         if self.Year > d.Year then true else
            if self.Year = d.Year then
            if self.Month > d.Month then true else
               if self.Month = d.Month then
               if self.Day > d.Day then true else false endif
               else false endif endif
            else false endif
         endif

      before(d : Date) : Boolean =
         if self.Year < d.Year then true else
            if self.Year = d.Year then
            if self.Month < d.Month then true else
               if self.Month = d.Month then
               if self.Day < d.Day then true else false endif
               else false endif endif
            else false endif
         endif

      addDay()
         begin
             self.Day := self.Day+1;
               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
               if self.Day > 31 then
                  self.Day := 1;
                  self.Month := self.Month+1;
```

```
37                   if self.Month > 12 then
38                   self.Year := self.Year+1;
39                   self.Month := 1;
40                   end
41                end
42                else
43                if self.Month = 2 then
44                   if self.Year.mod(4) = 0 and (self.Year.mod(100) <> 0 or
45                                                self.Year.mod(400) = 0) then
46                   if self.Day > 29 then
47                      self.Day := 1;
48                      self.Month := 3;
49                   end
50                   else
51                   if self.Day > 28 then
52                      self.Day := 1;
53                      self.Month := 3;
54                   end
55                   end
56                else
57                   if self.Day > 30 then
58                   self.Day := 1;
59                   self.Month := self.Month + 1;
60                   end
61                end
62                end
63          end
64       addDays(days : Integer)
65          begin
66             for v in Sequence{1..days} do self.addDay() end;
67          end
68       copyDateIntoSelf(date: Date)
69          begin
70             self.Day := date.Day;
71             self.Month := date.Month;
72             self.Year := date.Year;
73          end
74    constraints
75       inv Day_Must_Fit_Into_Gregorian_Numbers: self.Day ≥ 1 and self.Day ≤ 31
76       inv Month_Must_Fit_Into_Gregorian_Numbers: self.Month ≥ 1 and self.Month ≤ 12
77       inv Year_Must_Be_After_1000_And_Before_3000: self.Year ≥ 1000 and self.Year < 3000
78 end
```

3.12.1 Date-Attribute

Day
Stellt den Tag des Datums dar.

Month
Stellt den Monat des Datums dar.

Year
Stellt das Jahr des Datums dar.

3.12.2 Date-Operationen

equals(d : Date)
Sagt aus, ob dieses Datum mit einem anderen identisch ist.

after(d : Date)
Fragt, ob dieses Datum nach dem übergebenen Datum liegt.

before(d: Date)
Fragt, ob dieses Datum noch vor dem übergebenen Datum liegt.

addDay()
Fügt einen Tag zu diesem Datum hinzu.

addDays(days : Integer)
Fügt mehrere Tage zu diesem Datum hinzu.

copyDateIntoSelf(date: Date)
Macht dieses Datum zu einer Kopie eines anderen Datums.

3.12.3 Date-Bedingungen

inv Day_Must_Fit_Into_Gregorian_Numbers
Die Angabe des Tages muss sich zwischen 1 und 31 befinden.

inv Month_Must_Fit_Into_Gregorian_Numbers
Die Angabe des Monats muss sich zwischen 1 und 12 befinden.

inv Year_Must_Be_After_1000_And_Before_3000
Sinnvoller weise sollte die Angabe des Jahres zwischen 1000 und 3000 liegen.

3.13 Class CurrentDate

Die Klasse CurrentDate leitet sich von Date ab und repräsentiert ein Singleton von dem zu jeder Zeit ein Objekt existieren muss (Länderübergreifende Zeitverschiebung wird nicht berücksichtigt).

```
1 class CurrentDate < Date
2   constraints
3     existential inv Current_Date_Is_Existing_Singleton: CurrentDate.allInstances()→
          size = 1
4 end
```

3.13.1 CurrentDate-Attribute

Diese Klasse verfügt über keinerlei eigene Attribute, da keine benötigt werden.

3.13.2 CurrentDate-Operationen

Diese Klasse verfügt über keinerlei eigene Operationen, da keine benötigt werden.

3.13.3 CurrentDate-Bedingungen

existential inv Current_Date_Is_Singleton
Schlägt alarm, falls es mehr als nur eine CurrentDate Klasse geben sollte.
(Das existential inv ist kein OCL-Standard sondern nutzt nur wieder eine Eigenheit von USE. Würde die Anzahl der Objekte der Klasse CurrentDate = 0 betragen, so würde die Invariante bei einem einfachen „**inv**“ zu true auswerten, da der Kontext, für den die Invariante auswertet, leer ist. Für einen leeren Kontext ist ein forAll natürlich immer wahr. Daher wird hier das „**existential inv**“ verwendet, da dieses den Kontext über ein „**exists**“ auswertet, wodurch die Invariante dann schließlich so ausgewertet wird, wie sie es soll.)

3.14 Class Time

Die bereits zuvor erwähnte symbolische Klasse, die hier nicht effektiv mit eingebunden werden kann.
Die Time Klasse sollte, wenn sie es denn könnte, den Umstand regeln, dass die Filme an einem Tag bis 19 Uhr zurückgebracht werden müssen, da die Leihdauer sonst für den nächsten Werktag gilt. Sprich, wer heute einen Film leiht und ihn erst nach 19 Uhr zurückbringt, hat ihn 2 Tage gehabt.

```
1  class Time
2     attributes
3        hour: Integer
4        minute: Integer
5        second: Integer
6     constraints
7        inv Hour_Must_Be_Specified: self.hour ≥ 0 and self.hour ≤ 23
8        inv Minute_Must_Be_Specified: self.minute ≥ 0 and self.minute ≤ 59
9        inv Second_Must_Be_Specified: self.second ≥ 0 and self.second ≤ 59
10 end
```

3.14.1 Time-Attribute

hour
Stellt die Stunde der Zeit dar.

minute
Stellt die Minute der Zeit dar.

second
Stellt die Sekunde der Zeit dar.

3.14.2 Time-Operationen

Da die Klasse nicht effektiv eingesetzt werden kann, gibt es auch keine Operationen.

3.14.3 Time-Bedingungen

inv Hour_Must_Be_Specified
Die Angabe der Stunde muss zwischen 0 und 23 liegen.

inv Minute_Must_Be_Specified
Die Angabe der Minuten muss zwischen 0 und 59 liegen.

inv Second_Must_Be_Specified
Die Angabe der Sekunden muss zwischen 0 und 59 liegen.

4 Assoziationsbeschreibungen

Im folgenden Kapitel soll näher auf die Assoziationen und deren Bedeutungen im Zusammenhang des Systems eingegangen werden.

4.1 Associationsclass Period

```
1 associationclass Period
2    between
3       MovieCopy[*] role lentMovies
4       Employee[0..1] role conferedBy
5       Customer[0..1] role borrowedBy
6    ...
7 end
```

Die Assoziationsklasse Period stellt ein Zusatzattribut in der ternären Beziehung zwischen einer Filmkopie, eines Angestellten und eines Kunden dar. Der Gedanke dahinter ist, dass jede Filmkopie, zu jeder Zeit, zu einem Kunden zurückverfolgt werden können soll, damit der Kunde im Falle einer Verspätung benachrichtigt werden kann. Der Angestellte soll insofern in diese Beziehung hineinpassen, dass in einem solchen Falle ggf. Rücksprache mit dem Angestellten gehalten werden kann, damit dieser eventuell noch weitere Informationen über den entsprechenden Kunden geben kann.
Die Multiplizitäten zwischen dem Ganzen sollen wie folgt aufgelöst werden:
Zu jeder Filmkopie darf immer nur exakt ein Kunde und ein Angestellter in Beziehung stehen. Dabei muss es so sein, dass es entweder ein Angestellter und ein Kunde sind, oder gar kein Kunde und gar kein Angestellter. Allerdings dürfen zu jedem Kunden-Angestellten Paar beliebig viele Filmkopien existieren.

4.2 Composition CompanyHasManyFilials

```
1 composition CompanyHasManyFilials between
2    Company[1] role creator
3    Filial[1..*] role filials
4 end
```

Als Verbindungstyp wurde hier eine Komposition gewählt, da eine Filiale ohne eine Firma nicht existieren soll. Außerdem muss eine Firma mindestens eine Filiale haben, also einen Standort,

an dem man sie besuchen kann.

4.3 Composition FilialHasAddress

```
composition FilialHasAddress between
   Filial[0..1]
   Address[1]
end
```

Auch hier wurde als Verbindungstyp die Komposition gewählt, da eine Addresse zu einer Filiale gehört. Dabei ist die Multiplizität durch 0..1 statt 1 dadurch zu erklären, dass ein Addressenobjekt entweder zu einer Reihe von Personen oder, zu einer Filiale gehört. Zudem muss jeder Filiale jedoch genau ein valides Addressenobjekt zugeordnet sein.

4.4 Aggregation PersonHasAddress

```
aggregation PersonHasAddress between
   Person[*] role persons
   Address[1]
end
```

Als Verbindungstyp dient hier die Aggregation, eine Addresse zu diversen Personen zugeordnet werden kann, wenn sie bspw. in einem Hochhaus wohnen. Hingegen wird jede Person allerdings so betrachtet, dass sie nur einen einzigen privaten Wohnsitz haben kann, an dem sie zu erreichen ist.

4.5 Association CompanyHasRightToConferCopiesOfMovies

```
association CompanyHasRightToConferCopiesOfMovies between
   Company[0..*] role companyWithConferRights
   Movie[50..*] role movies
end
```

Die Assoziation wurde hier gewählt, da ein Film nicht als ein Teil einer Videothek gilt. Vielmehr gilt ein Film als eigenständiges Objekt, das sich von einer Videothek zunutze gemacht wird. Somit kann ein Filmobjekt auch ganz ohne Verbindungen zu einer Firma existieren, wohingegen eine Firma, um als Fimra zu gelten die Rechte an mindestens 50 Filmen haben sollte [1].

4.6 Composition EmployeeWorksInFilial

```
composition EmployeeWorksInFilial between
   Filial[1] role employer
   Employee[1..*] role worker
end
```

Hier gibt es wieder eine Komposition, da es einen Angestellten nicht ohne Ort geben kann, an dem er angestellt sein wird. Außerdem darf ein Angestellter in maximal einer Filiale arbeiten.

[1] persönliche Anmerkung: Ich weiß, das ist immernoch echt lächerlich, aber ich wollte es für diese Ausarbeit einfach nicht übertreiben :-)

eine Filiale hingegen muss mindestens einen Angestellten haben und darf beliebig viele einstellen.

4.7 Aggregation CustomerHasContactToFilial

```
aggregation CustomerHasContactToFilial between
   Filial[1..*] role serviceProvider
   Customer[0..*] role customer
end
```

Jeder Kunde kann sich in beliebig vielen Filialen der gleichen Firma anmelden kann, die ihre Daten über diesen Kunden entsprechend teilen. Dabei muss nicht jeder Kunde mit einer Filiale verlinkt sein. Die Verbindungen werden gelöscht, wenn ein Kunde ausgeschlossen wird, der Kunde bleibt jedoch als Fragment erhalten, damit im Nachhinein noch eingesehen werden kann, ob dieser Kunde, wenn er wiederkommt, früher schon einmal ausgeschlossen wurde.
Dabei wurde die Aggregation hier gewählt, da ein Kunde niemals ohne Filiale in der Datenbank sein kann. somit ist er gewissermaßen ein Teil der Filiale, womit auch die Multiplizitäten erklärt wären.

4.8 Association FilialMustPossessMovieCopies

```
association FilialMustPossessMovieCopies between
   Filial[1] role owner
   MovieCopy[50..*] role movies
end
```

Diese Assoziation wurde gewählt, da eine Filiale über diverse Filmkopien verfügen sollte, sie aber nicht direkt ein Teil der Filiale sind. Es ist einer Filiale gestattet die Filmkopien zu verkaufen, wenn dies gewünscht ist, daher ist die Assoziation hier die passenderere Wahl.
Abgesehen davon, wird sich nur für eine Filmkopie interessiert, solange sie im Besitz einer Filiale ist. Sprich wenn eine Kopie verkauft wird, wird der Eintrag zu dieser Kopie gelöscht, womit sich die 1-Multiplizität rechtfertigt. Zusätzlich sollte jede Filiale ein Minimum von 50 Filmen im Bestand haben.

4.9 Composition MovieCopyIsFromMovie

```
composition MovieCopyIsFromMovie between
   Movie[1] role original
   MovieCopy[0..*] role copy
end
```

Die Komposition dürfte sich hier eigentlich von allein erklären, denn jede Filmkopie braucht eine Film, von dem sie kopiert wurde und jede Kopie kann nur auf genau einen Film verweisen. Von einem Film jedoch kann es beliebig viele Filmkopien geben.

4.10 Association DateHasTime

```
association DateHasTime between
   Date[0..*]
   Time[0..86399]
end
```

Diese Assoziation beruht darauf, dass ein Time Objekt jederzeit auch für sich alleine stehen kann und verwendet werden kann, wie es benötigt wird. Das gleiche gilt für das Date-Objekt, wobei man in beideitiger Richtung diverse Zeitobjekte den Dateobjekten zuordnen kann und

diverse Dateobjekte den Zeitobjekten. Hier kann man zusätzlich noch davon ausgehen, dass die einzelnen Zeitobjekte jeweils einzigartig sein sollen, was die Anzahl der Objekte auf 86399 begrenzt [2].

4.11 Aggregation ActorBelongsToMovies

```
aggregation ActorBelongsToMovies between
   Movie[1..*] role movies
   Actor[0..*] role actors
end
```

Die Aggregation hier sollte sich ebenfalls von selbst verstehen. Ein Schauspieler muss in mindestens einem Film mitgespielt haben, um sich als Schauspieler bezeichnen zu dürfen, womit ein Schauspieler ein Teil eines Filmes ist. Dabei ist ein Film jedoch nicht dazu verpflichtet eine Beziehung zu einem Schauspieler zu haben, da ein Film auch gänzlich ohne menschliche Darsteller gedreht werden kann.

4.12 Association GenreBelongsToMovies

```
association GenreBelongsToMovies between
   Movie[0..*] role movies
   Genre[1..*] role genre
end
```

Die Assoziation hier begründet sich damit, dass Genres heutzutage bereits fest definiert sind und Filme und auch andere Dinge diesen Genres zugeordnet werden können. Es kann nicht behauptet werden, dass ein Genre ein Teil eines Films wäre, aber kann auch nicht behauptet werden, dass ein Film ein Teil eines Genres wäre. Vielmehr gehören beide eher gleichwertig zusammen als ein Ganzes.
Zu dieser Behauptung stehen die Multiplizitäten jedoch konträr, da jeder Film mindestens einem Genre zugeordnet werden muss, umgekehrt jedoch nicht.

4.13 Association EmployeeWorksOnDates

```
association EmployeeWorksOnDates between
   Employee[*]
   Date[*] role workingDates
end
```

Diese Assoziation begründet sich darauf, dass ein Angestellter regelmäßig Arbeitstage zugewiesen bekommt, an denen er arbeiten muss. Da dieser jedoch nicht immer welche zugewiesen bekommen muss, stehen die Multiplizitäten in beide Richtungen auf beliebig viele.

5 Systemzustände

Jetzt soll gezeigt werden, welche Systemzustände im Programm gültig sind und was beachtet werden muss, damit diese Zustände auch gültig bleiben. Zusätzlich wird es einige Beispiele geben,

[2] Die eigentliche symbolische Beziehung, die ich ursprünglich mit der Timeklasse ausdrücken wollte, lässt sich nur schwer darstellen über Assoziationen, weshalb hier lediglich eine Verbindung zwischen Date und Time angegeben wurde.

die Invarianten, oder auch einige spezielle pre- bzw- post-conditions verletzen.
Zu diesem Zweck sollen nun verschiedene Szenarios bearbeitet werden, welche zu dem auch die einzelnen gültigen Systemzusätnde demonstrieren. Das erste Szenario zeigt, wie Film-Objekte so zuammengestellt werden müssen, dass der System-Zustand valide bleibt. Im nächsten Szenario, wird dann eine Firma gegründet, die eine Filiale eröffnet. Im dritten Szenario werden dann Kunden für die Filialen registriert und neue Personen in der Filiale angestellt.
Weiter geht es dann mit dem 4. Szenario in dem ein Kunde einige Filme aus einer Videothek ausleihen wird und auf dem Nachhauseweg in einer zweiten Filiale der selben Videothek halten und weitere Filme ausleihen wird. Im 5. Szenario kündigt der Angestellte, der den Kunden in der ersten Filiale beraten hatte, da dieser einen neuen Job angenommen hat und die Filme werden im Anschluss zusätzlich mit Verspätung in beiden Filialen wieder zurückgebracht.
Das 6. Szenario demonstriert was passiert, wenn eine Filiale neue Filmkopien anfordert und erklärt welche Auswirkungen dies auf den Systemzustand hat.
Im 7. Szenario wird dann eine Filiale wieder geschlossen und gezeigt, zu welchen Ergebnissen dies führt. Im 8. und letzten Szenario dann wird lediglich noch eine weitere Besonderheit des Systems gezeigt, die in einem vorhergegangenen Szenario bereits angesprochen sein wird, aber nicht aufgezeigt wurde.

5.1 Szenario 1

Um überhaupt einen ersten gültigen Zustand zu erzeugen, muss als aller erstes ein spezielles Objekt erzeugt werden, welches im Systemzustand als Singleton fungieren soll. Dieses spezielle Objekt soll aus der „**CurrentDate**-Klasse“ stammen. Das aktuelle Datum soll für alle Filialen gleichermaßen zugänglich sein, wobei unterschiedliche Zeitzonen nicht berücksichtigt werden.
Es gilt außerdem zu beachten, dass so lange kein Objekt der „**CurrentDate**-Klasse“ vorliegt und die Werte des Objektes nicht innerhalb der zulässigen Werte eines gregorianischen Kalenders liegen [3], gilt der Zustand als invalide.
Dazu erzeugen wir dann ein Filmobjekt, welches mindestens einem Genre zugeordnet sein muss, so wie es die Multiplizitäten im UML-Diagramm verlangen. Abgesehen von einem Genre-Objekt darf das Film-Objekt für sich alleine stehen und bedarf, so lange seine Attribute alle definiert sind, keinerlei weiterer Beachtung von Abhängigkeiten. Dennoch werden dem Film-Objekt zusätzlich zwei weitere Genre-Objekte und 4 Schauspieler-Objekte hinzugefügt. Wichtig zu beachten ist hier, dass jedes Schauspieler-Objekt mindestens einem Film-Objekt zugeordnet werden muss. Andernfalls resultiert dies in einem ungültigen Systemzustand. Aus diesem Grund werden die Genre- und Schauspieler-Objekte, wie gleich in Abbildung 5.1.1 gezeigt wird, mittels einer Funktion, die von der „**Movie**-Klasse“ zur Verfügung gestellt wird, erzeugt. Diese Funktion stellt sicher, dass Schauspieler nicht doppelt in der Datenbank eingetragen werden und immer das gleiche Objekt mit verschiedenen Film-Objekten verlinkt wird.

[3] Grob gesprochen, da die überprüfung der Einfachheit halber für jeden Monat 31 Tage zuläst.

```
!create currentDate : CurrentDate
!currentDate.Day := 16
!currentDate.Month := 7
!currentDate.Year := 2013
!create Movie1 : Movie
!Movie1.title := '12 Monkeys'
!Movie1.fsk := 16
!Movie1.releaseYear := 2004
!Movie1.addGenre('Drama')
!Movie1.addGenre('Thriller')
!Movie1.addGenre('Science Fiction')
!Movie1.addActor('Bruce Willis')
!Movie1.addActor('Madeleine Stowe')
!Movie1.addActor('Brad Pitt')
!Movie1.addActor('Christopher Plumme')
```

Abbildung 5.1.1: szenario1.soil

Sollte jetzt noch versucht werden über ein Objekt, das gleiche Genre, oder den gleichen Schauspieler ein weiteres mal mittels der Operation hinzuzufügen, wird in beiden Fällen die precondition fehlschlagen, dass ein solcher Link noch nicht vorhanden sein darf, wenn die Operation aufgerufen wird.
Da dies hier nicht geschehen ist, befindet sich das System derzeit in einem vollständig validen Zustand. Um einen tieferen Einblick in einen solchen Zustand zu gewähren, soll Abbildung 5.1.2 dienen, welche den vollständigen Systemzustand zum jetzigen Zeitpunkt aufzeigt.

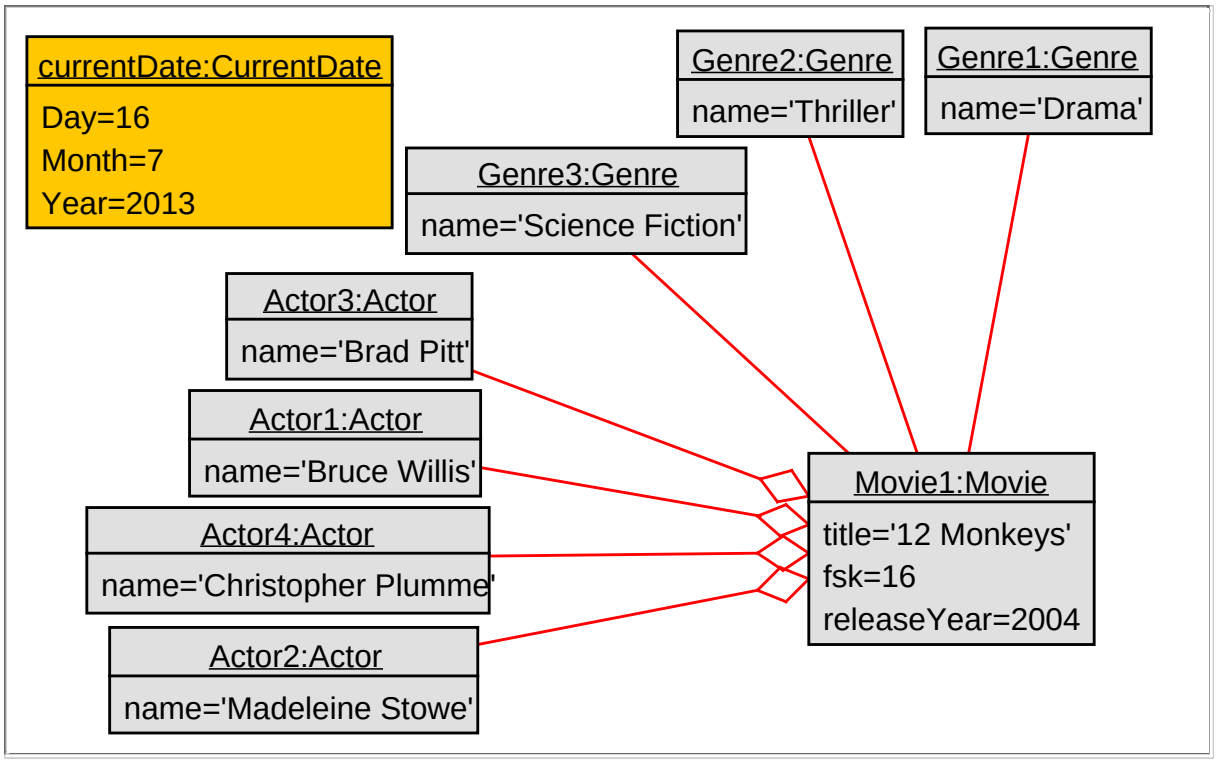

Abbildung 5.1.2: Szenario 1 - valides Objektdiagramm

5.2 Szenario 2

Jetzt soll eine Firma eröffnet werden, welche dazu 2 Mini-Filialen eröffnen soll. Eine Firma soll sich im Modell durch seine Filialen nach außen hin demonstrieren. Da eine Filiale zudem jederzeit für Kunden erreichbar sein muss, brauchen die Filialen eine Addresse. Aber auch dies reicht noch nicht aus. Jede Filiale braucht mindestens einen Angestellten, der die Filiale betreuen kann. Hierzu muss natürlich noch beachtet werden, dass die Personaldaten der Angestellten, wie auch deren Addresse aufgenommen werden und dass die Filiale einen Warenbetand benötigt, welchen sie den Kunden anbieten kann. Der Minimum des Warenbestandes ist hier auf 50 festgelegt worden, damit die Mini-Filialen ihren Kunden zumindest ein kleines Sortiment bieten können. Abbildung 5.2.2 zeigt die Befehlskette die den nächsten validen Zustand des Systems erstellt. [4] Zunächst ist es allerdings wichtig, dass die Firma sich erst einmal einen Bestand an Filmrechten sichert, von welchem die Firma dann berechtigt ist Kopien an ihre Filiale weiterzugeben. Dieser Bestand muss sich, wie zuvor schon einmal erwähnt, auf eine Menge von Mindestens 50 Film-Objekten beziehen, damit die Firma sich als eine „Videotheken-Firma“ bezeichnen kann. Auch muss darauf geachtet werden, dass jede Filiale Kopien von mindestens 50 unterschiedlichen Filmen hat, weshalb die Firma im aktuellen Zustand dazu gezwungen ist, ihren gesamten Bestand an beide Filialen weiter zu geben. Die Erzeugung dieser Film-Objekte würde sehr viel Platz einnehmen, weshalb Abbildung 5.2.1 nur einen kleinen Einblick geben soll.[5]

```
1  ...
2  !create Company1 : Company
3  !Company1.name := 'Goldfishs Golden Video Pool'
4  !create Movie1 : Movie
5  !Movie1.title := '12 Monkeys'
6  !Movie1.fsk := 16
7  !Movie1.releaseYear := 2004
8  !Movie1.addActor('Bruce Willis')
9  !Movie1.addActor('Madeleine Stowe')
10 !Movie1.addActor('Brad Pitt')
11 !Movie1.addActor('Christopher Plumme')
12 !Movie1.addGenre('Drama')
13 !Movie1.addGenre('Science Fiction')
14 !Movie1.addGenre('Thriller')
15 !Company1.addMovie(Movie1)
16 !create Movie65 : Movie
17 !Movie65.title := 'Der 13te Krieger'
18 !Movie65.fsk := 16
19 !Movie65.releaseYear := 2000
20 !Movie65.addActor('Antonio Banderas')
21 !Movie65.addActor('Vladimir Kulich')
22 !Movie65.addActor('Dennis StorhÃ¸i')
23 !Movie65.addActor('Daniel Southern')
24 !Movie65.addGenre('Abenteuer')
25 !Movie65.addGenre('Action')
26 !Movie65.addGenre('Fantasy')
27 !Company1.addMovie(Movie65)
28 ...
```

Abbildung 5.2.1: szenario2-movies.soil

[4] Dabei sei angenommen, dass der vorherige Systemzustand wieder auf den initial-Zustand zurück gesetzt wurde.

[5] Der vollständige Code befindet sich in der Datei, dessen Name in der Beschreibung unter der Abbildung 5.2.1 genannt ist.

Wie man in Abbildung 5.2.1 sehen kann, wird hier auch bereits das Firmen-Objekt erstellt, welche das erstellte Film-Objekt in seinen Bestand mittels seiner Operation „**addMovie**“ hinzufügt. Diese Operation sorgt dafür, dass kein Filmobjekt zweimal der gleichen Firma zugewiesen wird, indem die pre-condition fehlschlägt, sollte versucht werden, gegen diese Auflage zu verstoßen. Hat die Firma sich nun die Rechte an 50 Film-Objekten gesichert, ist es an der Zeit die beiden gewünschten Filialen zu eröffnen, wobei die zuvor genannten Bedingungen eingehalten werden müssen. Zu diesem Zweck dienen die Befehle in Abbildung 5.2.2

```
1  !MovieSubset1 := Company1.movies→select( m | m.fsk = 0 or m.fsk = 6 or m.fsk = 12 or m.
       fsk = 16 or m.fsk = 18)
2   !create Employee1 : Employee
3  !Employee1.status := #employed
4  !Employee1.phonenumber := '495148495655'
5  !Employee1.identityNumber := 'isrophbswj1'
6  !Employee1.firstname := 'Martin'
7  !Employee1.lastname := 'Gogolla'
8   !create Address1 : Address
9  !Address1.street := 'Angestellten1 Strasse'
10 !Address1.housenumber := '5152'
11 !Address1.postcode := '535649545448'
12 !Address1.town := 'Bremen'
13  !insert (Employee1, Address1) into PersonHasAddress
14 !create Address2 : Address
15 !Address2.street := 'Filial1 Strasse'
16 !Address2.housenumber := '5549'
17 !Address2.postcode := '545652535448'
18 !Address2.town := 'Bremen'
19  !Company1.createFilial('Golden Video Pool', Address2, MovieSubset1 , Employee1)
20 !MovieSubset2 := Company1.movies→select( m | m.fsk = 0 or m.fsk = 6 or m.fsk = 12 or m.
       fsk = 16 or m.fsk = 18)
21  !create Employee2 : Employee
22 !Employee2.status := #employed
23 !Employee2.phonenumber := '564955535056'
24 !Employee2.identityNumber := 'ftwzdchtsb2'
25 !Employee2.firstname := 'Lars'
26 !Employee2.lastname := 'Hamann'
27  !create Address3 : Address
28 !Address3.street := 'Angestellten2 Strasse'
29 !Address3.housenumber := '5455'
30 !Address3.postcode := '504956534851'
31 !Address3.town := 'Bremen'
32  !insert (Employee2, Address3) into PersonHasAddress
33  !create Address4 : Address
34 !Address4.street := 'Filial2 Strasse'
35 !Address4.housenumber := '5556'
36 !Address4.postcode := '515056545650'
37 !Address4.town := 'Bremen'
38  !Company1.createFilial('Golden Video Pool', Address4, MovieSubset2 , Employee2)
```

Abbildung 5.2.2: szenario2.soil

Die in Abbildung 5.2.2 genutzte Operation der „**Company**-Klasse“ „createFilial“ übernimmt die Aufgabe alle Objekte so miteinander zu verbinden, dass ein valider Zustand entsteht. Hierbei achtet die Operation „createFilial“ bspw. in ihrer pre-condition darauf, dass genügend Film-Objekte an die Filiale übergeben wurde. Wenn dies nicht der Fall ist, schlägt die pre-condition sofort fehl und bricht den Versuch eine Filiale zu eröffnen ab. Es wird dabei auch überprüft, ob die Film-Objekte, die übergeben werden im Rechtebereich der Firma liegen. Sollte dies auch nicht der Fall sein, schlägt eine pre-condition fehl. Die Datei „szenario2-fail-pre3.soil“ aus welcher ein kleiner ausschlaggebender Teil in Abbildung 5.2.3 gezeigt wird, provoziert genau diesen Sachverhalt.

```
!Movie448.title := 'Zombieland'
!Movie448.fsk := 16
!Movie448.releaseYear := 2010
!Movie448.addActor('Jesse Eisenberg')
!Movie448.addActor('Woody Harrelson')
!Movie448.addActor('Emma Stone')
!Movie448.addActor('Abi')
!Movie448.addGenre('Horror')
!Movie448.addGenre('Kom"odie')
!Company1.addMovie(Movie448)
!MovieSubset1 := Company1.movies→select( m | m.fsk = 0 or m.fsk = 6 or m.fsk = 12 or m.
    fsk = 16 or m.fsk = 18) → including(Movie448)
...
 !Company1.createFilial('Golden Video Pool', Address2, MovieSubset1 , Employee1)
...
```

Abbildung 5.2.3: szenario2-fail-pre3.soil

Die Vorraussetzung dafür, dass sich die Datei aus Abbildung 5.2.3 verwenden lässt, ist, dass der Zustand soweit zurückgesetzt wird, dass die zuvor erstellten Filialen mit all ihren Objekten nicht mehr existieren.

Der reultierende Systemzustand des ganzen ist sehr groß und recht komplex, weshalb in den folgenden beiden Abbildungen 5.2.4 und 5.2.5 Der Eindruck des gesamten Systemzustands anhand von kleinen Ausschnitten des Systemzustands verdeutlicht werden soll.
Dabei soll Abbildung 5.2.4 die Beziehung unter der Firma mit seinen Filialen, den Angestellten und Addressen vermitteln und Abbildung 5.2.5 soll zeigen, wie die Filialen im Zusammenhang mit den Filmen und die Filme im Zusammenhang mit der Firma stehen.

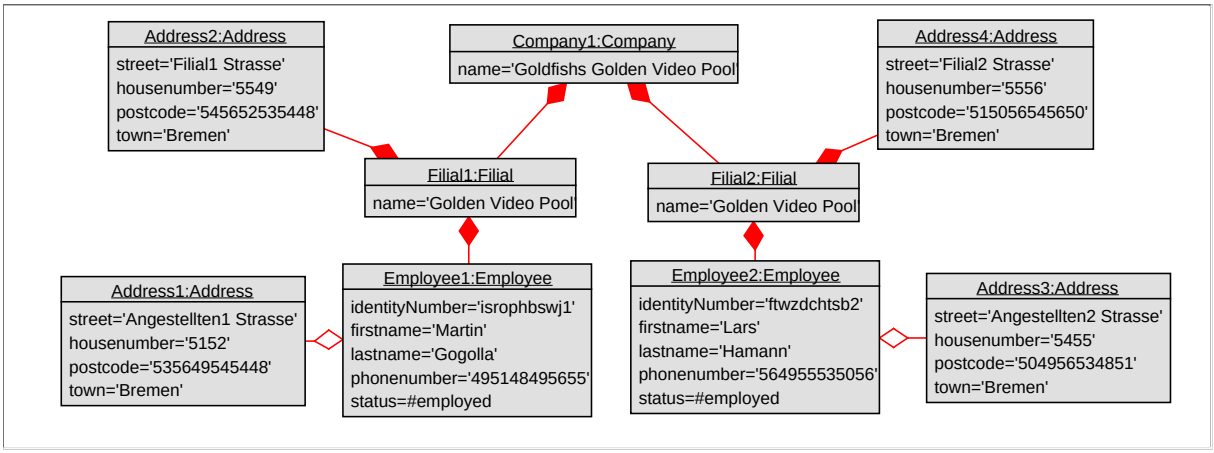

Abbildung 5.2.4: Szenario 2 - teilhaftes Objektdiagramm - Part 1

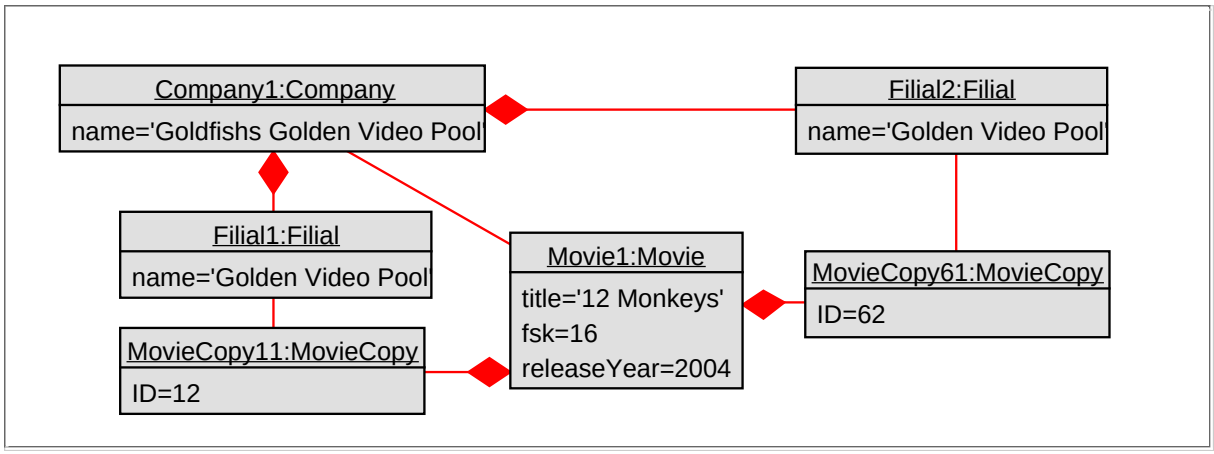

Abbildung 5.2.5: Szenario 2 - teilhaftes Objektdiagramm - Part 2

5.3 Szenario 3

In Szenario 3 sollen die Filialen einen Kunden in ihrem System aufnehmen. Dabei wird witzigerweise der gleiche Kunde in beiden Filialen Kunde Nummer 1 werden. Zusätzlich wird der Filialleiter der Filiale Nummer 1 merken, dass er bereits mit einem einzigen Kunden überfordert ist und stellt entsprechend jemanden ein, der diese Aufgabe für ihn übernimmt. Und damit dem Filialleiter die Arbeit nicht über den Kopf wächst, wird er auch noch die Arbeitstage für den nächsten Monat auf beide Angestellten verteilen, so dass beide Angestellten mal zusammen arbeiten und mal nicht.[6]
Abbildung 5.3.1 zeigt wie dieser Vorgang mit Kommandos in Soil bearbeitet werden kann.

```
1 !Filial1.createCustomer('asdf2134qwer', 'Subi', 'Aili', '0123456', 'Kundenstrasse',
      '3242', '464646464', 'Bremen' )
2 !Filial2.createCustomer('asdf2134qwer', 'Subi', 'Aili', '0123456', 'Kundenstrasse',
      '3242', '464646464', 'Bremen' )
3 !Filial1.employ( '123654789', 'Mirco', 'Kuhlmann', '0123555', 'Angestelltenstrasse',
      '666', '222666444', 'Bremen' )
```

Abbildung 5.3.1: szenario3.soil

Wenn die Befehle aus Abbildung 5.3.1 ausgeführt wurden, führt dies zu dem Zustand, der sich mit Abbildung 5.3.2 sehr gut nachvollziehen lässt. Dabei kann die Menge der Date-Objekte ein wenig verwirrend sein, soll aber lediglich zeigen, dass die Tage des aktuellen Monats, wie auch die Tage des darauffolgenden Monats auf die Angestellten verteilt wurden.

[6] Die Implementierung ist trotz größter Bemühungen nicht optimal verlaufen, weshalb sich hier mit einer suboptimalen Lösung zufrieden gegeben werden musste, die für die zwecke des Systems jedoch vollkommen ausreichend ist.

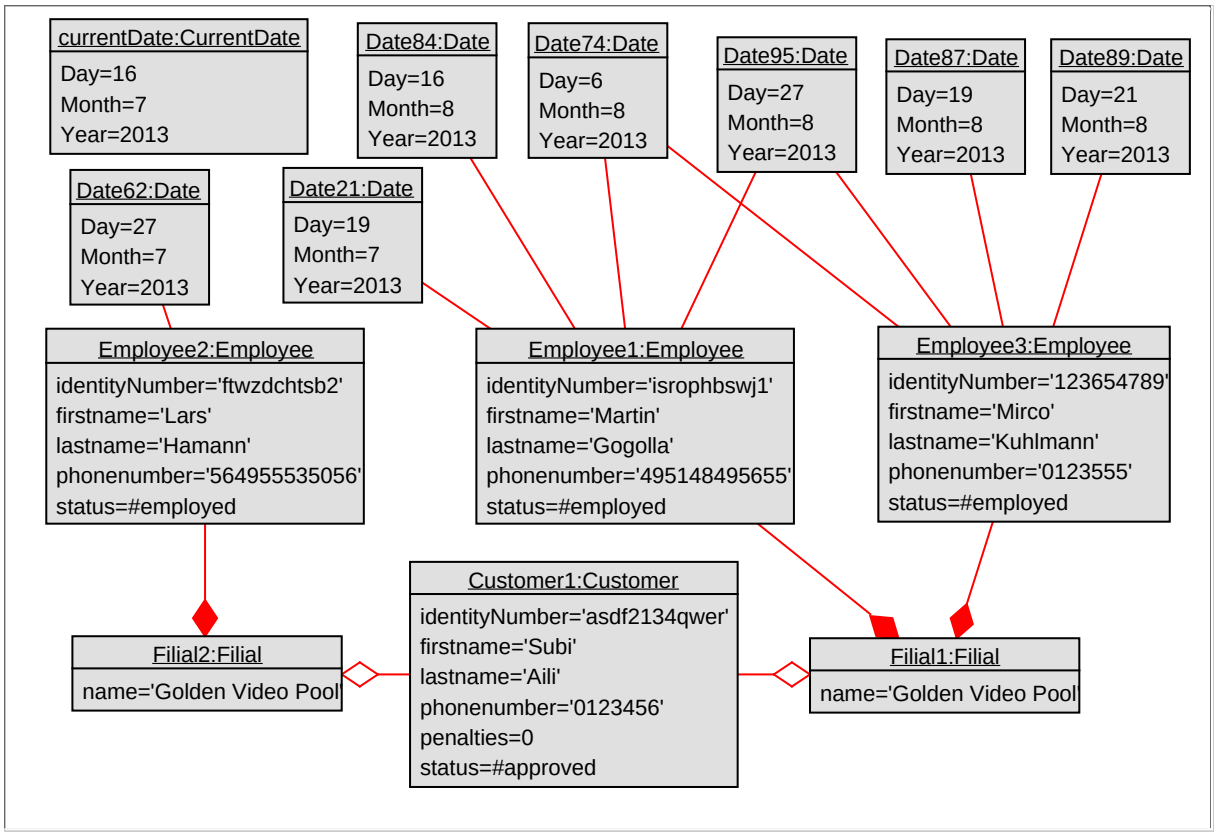

Abbildung 5.3.2: szenario3: Objektdiagramm

5.4 Szenario 4

Jetzt wird der zuvor in der Datenbank aufgenommene Kunde, einige Filme aus einer Filiale entleihen. Dies lässt sich mit den Kommandos aus Abiildung 5.4.1 erreichen.

```
1 !Customer1.borrowMovies(Movie.allInstances()→select(m | m.title = 'Asterix und die
      Wikinger' or m.title = 'Auf der Jagd'), Filial1)
2 !Customer1.borrowMovies(Movie.allInstances()→select(m | m.title = '300' or m.title = '
      Beverly Hills Cop'), Filial2)
```

Abbildung 5.4.1: szenario4.soil

Bei Ausführung dieser Befehle, werden diverse Schritte ausgeführt. Zuerst, der Kunde sucht sich in beiden Filialen jeweils 2 Filme aus. Dabei wird zufällig ein Angestellter ausgewählt, der am aktuellen Tag (**CurrentDate**) arbeitet und den Kunden bedient. Nun wird ein Eintrag gemacht, dass der Kunde sich in der jeweiligen Filiale 2 Filme ausgeliehen hat und welcher Angestellte den Kunden bedient hat. Die Filme können eindeutig über ihre ID identifiziert werden und gehören jeweils immer nur zu genau einer Filiale. Dazu wird für jeden entliehenen Film ein Ausleihdatum angelegt und ein Datum zu welchem Tag der Film spätestens wieder zurückgebracht werden muss.
Die Folgende Abbildung 5.4.2 zeigt den Teilzustand der aus aus den vorhergegangenen Befehlen resultiert. Wie man sieht steht der entsprechende Kunde im Mittelpunkt des Ganzen und hat auf die ein oder andere Weise eine **direkte** Verbindung zu jedem Objekt, das in diesem Diagramm zu finden ist.

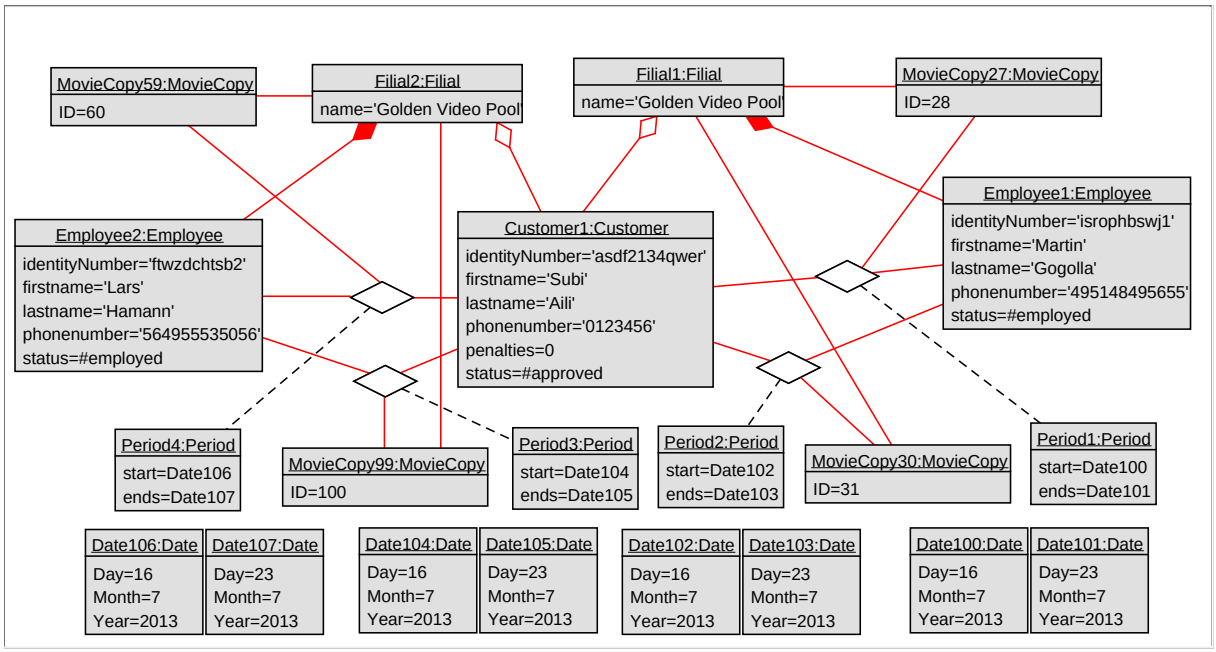

Abbildung 5.4.2: Objektdiagramm: Kunde entleiht Filme

5.5 Szenario 5

Wie man auf der letzten Abbildung 5.4.2 deutlich erkennen konnte, steht der Kunde mit den Angestellten, die ihm die Filme entliehen haben, in unmittelbarer Verbindung. Was passiert aber mit dem Systemzustand, wenn der Angestellte plötzlich kündigt, weil dieser einen neuen Job angenommen hat? Die Antwort auf diese Frage ist „fast gar nichts“. Wenn der Angestellte kündigt wird sein Status auf „unemployed“ gesetzt und gilt damit als nicht mehr in der Filiale angestellt. Als Folge daraus wird er nicht mehr zur Beratung von Kunden ausgewählt und wird in dem Moment gelöscht, wo alle Kunden - die in Verbindung mit diesem Angestellten stehen - ihre Filme zurückgebracht haben.
Zunächst also wird erst einmal der Angestellte 1 aus Filiale 1 kündigen. Das Ergebnis zeigt sich in Abbildung 5.5.2 welches durch das Kommando das in Abbildung 5.5.1 zu sehen ist erzeugt wird. Es sei dabei zu beachten, dass der Zustand noch immer derselbe ist wie zuvor, mit Ausnahme dass der status-Wert des Angestellten1 nun auf „notEmployed“ steht.

```
1 !Filial1.fireEmployee(Employee1);
```

Abbildung 5.5.1: szenario5-part1.soil

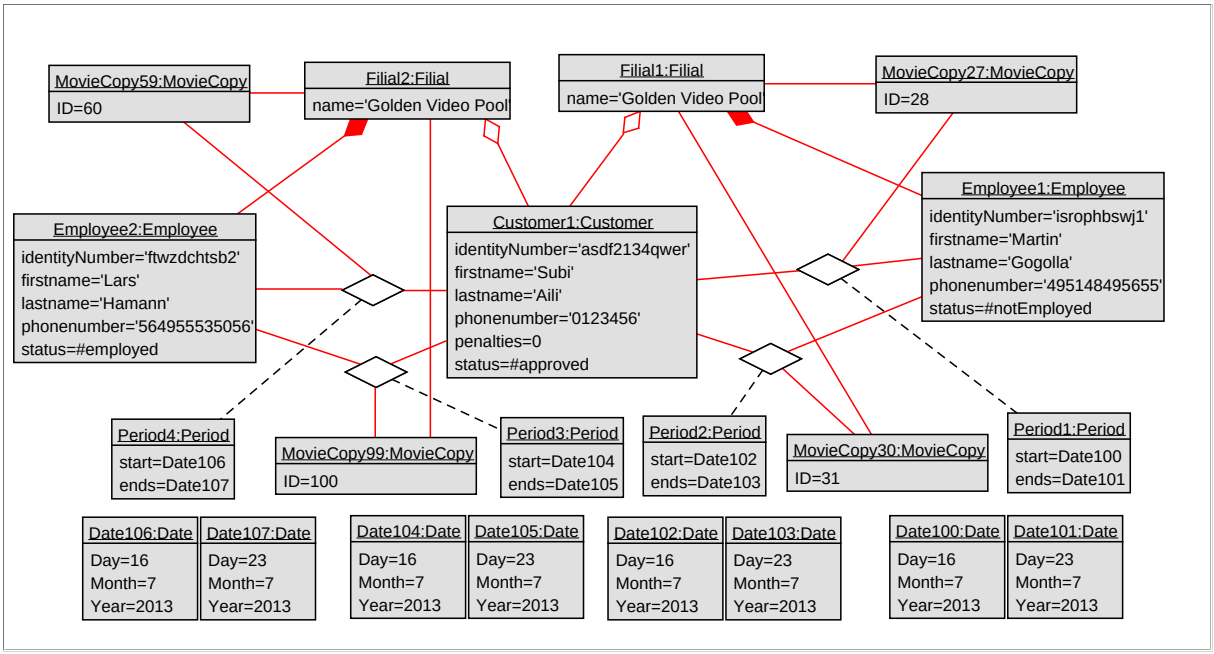

Abbildung 5.5.2: Objektdiagramm: Angestellter1 hat gekündigt

Im Vergleich zu Abbildung 5.4.2 mag der Unterschied enttäuschend sein, was sich allerdings im nächsten Schritt wieder ändern wird. Es werden 8 Tage vergehen und dann wird der Kunde die Filme aus Filiale1 zurückbringen und dabei zusätzlich versuchen, die Filme aus Filiale2 mit in Filiale1 zurück zu geben, was fehlschlagen wird, da Filiale1 das Eigentum von Filiale2 nicht akzeptieren wird. Um sich dies einmal anzusehen, wird der Befehl aus Abbildung 5.5.3 verwendet, welcher in der Fehlermeldung aus Abbildung 5.5.4 endet.

```
1 !Customer1.returnMovies(Customer1.period.lentMovies→asSet(),Filial1);
```

Abbildung 5.5.3: szenario5-fail.soil

```
[Error] 1 precondition in operation call 'Customer::returnMovies(self:Customer1, movies:
    Set{MovieCopy27,MovieC
opy30,MovieCopy59,MovieCopy99}, filial:Filial1)' does not hold:
  Movies_Must_Belong_To_Visited_Filial: filial.movies→includesAll(movies)
    filial : Filial = Filial1
    filial.movies : Set(MovieCopy) = Set{MovieCopy1,MovieCopy10,MovieCopy11,MovieCopy12,
        MovieCopy13,MovieCopy1
4,MovieCopy15,MovieCopy16,MovieCopy17,MovieCopy18,MovieCopy19,MovieCopy2,MovieCopy20,
    MovieCopy21,MovieCopy22,M
ovieCopy23,MovieCopy24,MovieCopy25,MovieCopy26,MovieCopy27,MovieCopy28,MovieCopy29,
    MovieCopy3,MovieCopy30,Movi
eCopy31,MovieCopy32,MovieCopy33,MovieCopy34,MovieCopy35,MovieCopy36,MovieCopy37,
    MovieCopy38,MovieCopy39,MovieC
opy4,MovieCopy40,MovieCopy41,MovieCopy42,MovieCopy43,MovieCopy44,MovieCopy45,MovieCopy46,
    MovieCopy47,MovieCopy
48,MovieCopy49,MovieCopy5,MovieCopy50,MovieCopy6,MovieCopy7,MovieCopy8,MovieCopy9}
    movies : Set(MovieCopy) = Set{MovieCopy27,MovieCopy30,MovieCopy59,MovieCopy99}
    filial.movies→includesAll(movies) : Boolean = false

  call stack at the time of evaluation:
    1. Customer::returnMovies(self:Customer1, movies:Set{MovieCopy27,MovieCopy30,
        MovieCopy59,MovieCopy99}, fil
ial:Filial1) [caller: Customer1.returnMovies(Customer1.period→collectNested($e : Period
    | $e.lentMovies)→asS
et, Filial1)@<input>:1:0]
```

Abbildung 5.5.4: pre-condition-fail

Dies beweist, dass die pre-condition ordnungsgemäß funktioniert und der Kunde seine Filme nur in der Filiale zurückgeben kann, in welcher er sie auch entliehen hat. Deshalb wird dies nun mit dem Befehl aus Abbildung 5.5.5 durchgeführt und der Kunde bringt seine Filme erstmals in die Filiale1 zurück.

```
!currentDate.addDays(8)
!Customer1.returnMovies(Customer1.period.lentMovies→select(mc | mc.owner = Filial1)→
    asSet(),Filial1);
```

Abbildung 5.5.5: szenario5-part2.soil

Das entsprechende Ergebnis dazu sieht man nun in Abbildung 5.5.6

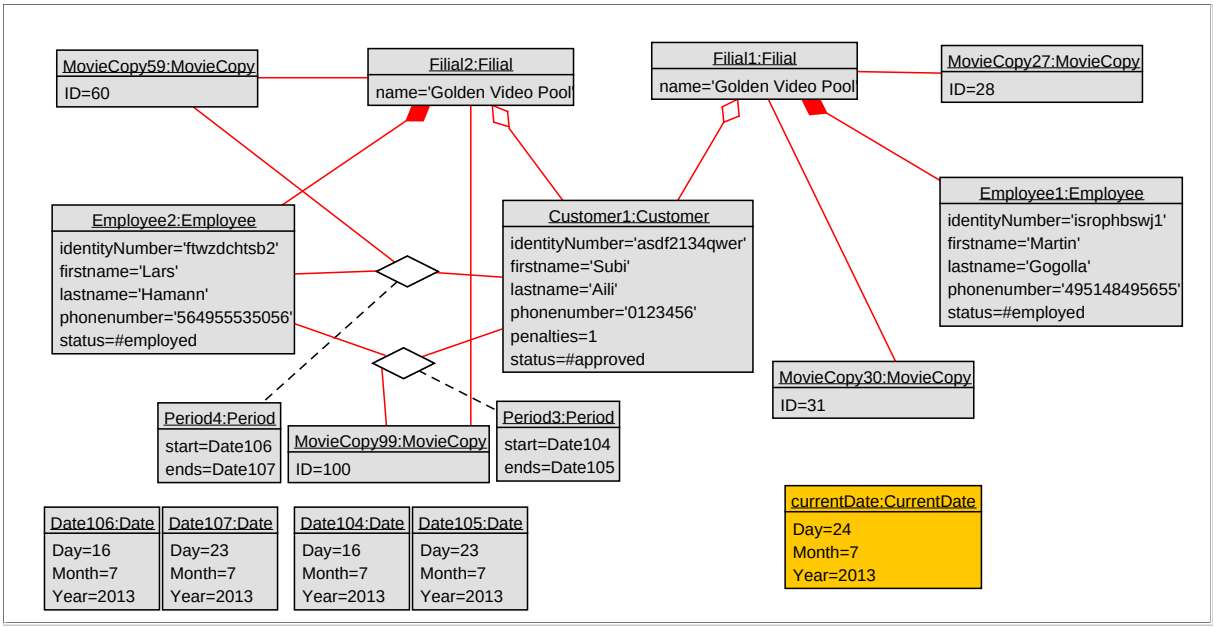

Abbildung 5.5.6: Rückgabe der Filme an Filiale1

Wie man sieht, hat sich der Zustand deutlich verkleinert. Als Beispiel wurden die Date-Objekte die zu den Period-Objekten gehörten gelöscht. Der Kunde ist entsprechend nicht mehr mit dem Angestellten verlinkt, da alle Filme zurückgegeben wurden. Der Kunde hat die Filme jedoch einen Tag zu spät zurückgegeben weshalb er einen Strafpunkt erhalten hat, was unter dem Attribut des Kunden „penalties“ zu sehen ist. Dieser änderte sich von 0 auf 1. Als nächstes wird der Kunde seine Filme auch an die Filiale2 zurückbringen, welches einen weiteren interessanten Effekt hat. Der Befehl aus Abbildung 5.5.7 erzeugt das Diagramm aus Abbildung 5.5.8

```
1 !currentDate.addDays(8)
2 !Customer1.returnMovies(Customer1.period.lentMovies→select(mc | mc.owner = Filial2)→
      asSet(),Filial2);
```

Abbildung 5.5.7: szenario5-part3.soil

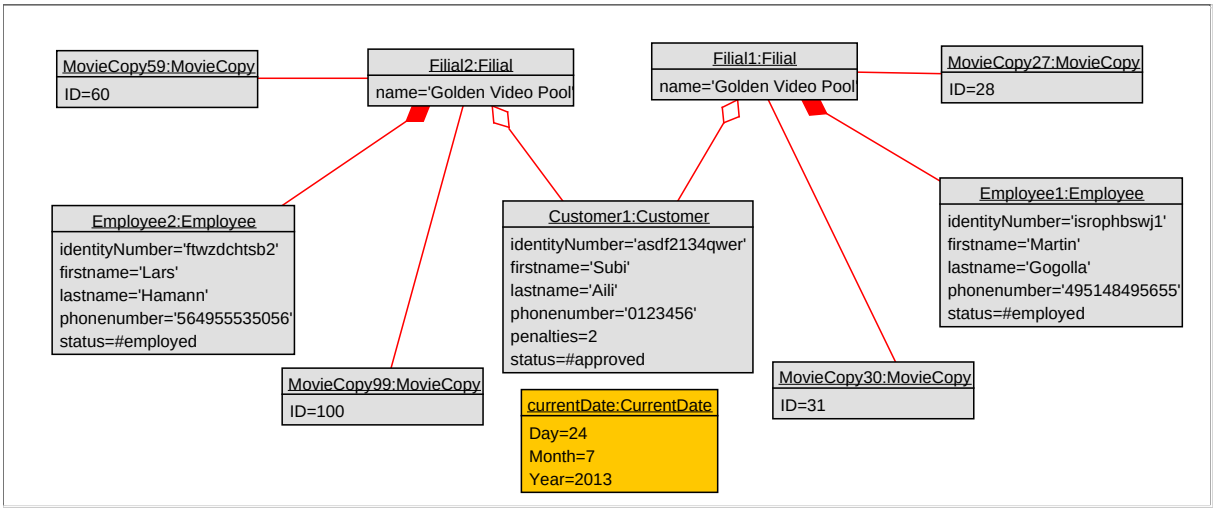

Abbildung 5.5.8: Kunde hat alle Filme zurückgebracht

In Abbildung 5.5.8 sehen wir nun, dass der Kunde einen zweiten Strafpunkt erhalten hat. Anstatt für jede Filiale einzeln abgebildet zu werden, wird der Kunde auf die Firma abgebildet, weshalb er egal in welcher Filiale er Filme zu spät zurückgibt, für jede Verspätung einen Strafpunkt erhalten. Dabei werden de Strafpunkte jedoch nicht auf die Menge der Filme gerechnet, sonder auf die Menge der Rückgaben. Daher erhielt der Kunde bei jeder Rückgabe, in denen er jeweils 2 Filme zurückbrachte genau einen Strafpunkt, weshalb er nun bei 2 Strafpunkten ist. Erhält er einen weiteren, wird als Kunde gesperrt und darf keine Filme mehr entleihen. Aber dazu später noch mehr.

5.6 Szenario 6

In diesem Szenario wird die Filiale 2 zusätzliche Filmkopien eines bestimmten Filmes anwerben. Die Auswirkungen sind im ersten Moment wenig spürbar, bringen aber doch ein paar interessante Aspekte mit sich. Zu diesem Zweck wird der Systemzustand, der zum Ende von Szenario 4 erreicht wurde wiederhergestellt. Im Anschluss werden 2 weitere Kunden registriert. Ein Kunde wird Miglied nur in Filiale 2 und der andere wird Mitglied in beiden Filialen. Der Grund dafür ist der, dass Kunde 3 in Filiale 1 gerne einen Film entleihen würde, welcher bereits vom ersten Kunden entliehen wurde. Da Filiale 1 keine weiteren Kopien dieses Filmes besitzt, kann der Angestellte ihn aufgrund des Einblicks in den Bestand der anderen Filialen an die 2. Filiale verweisen, in welcher noch eine von 2 Kopien vorhanden ist. Die erste Kopie wurde mittlerweile von Kunde 2 entliehen und die zusätzliche Kopie wird jene sein, die wie zu Anfang erwähnt von Filiale 2 entsprechend angefordert wurde. Um diesen Zustand zu erreichen dient der Befehlssatz aus Abbildung 5.6.1 welcher noch einen zusätzlichen Aspekt hervorhebt, der bisher nicht näher erläutert wurde.

```
1 !Filial1.createCustomer('asdf9874rewq', 'Pascal', 'Knueppel', '0123456', 'Kundenstrasse',
      '3242', '464646464', 'Bremen' )
2 !Filial2.createCustomer('asdf9874rewq', 'Pascal', 'Knueppel', '0123456', 'Kundenstrasse',
      '3242', '464646464', 'Bremen' )
3 !Filial1.createCustomer('aaaa0011bbbb', 'Daniel', 'Baden', '0128556', 'Kundenstrasse',
     '3242', '464646464', 'Bremen' )
```

Abbildung 5.6.1: szenario6-part1.soil

Wie man hier sehen kann, werden den beiden Kunden die gleichen Adressen zugewiesen. Diese Adresse stimmt außerdem mit der Addresse des ersten Kunden überein. In diesem System wurde davon ausgegangen, dass eine Filiale jeweils immer eine komplett eigene Addresse erhält, Kunden und Angestellte jedoch durchauch im selben Gebäude wohnen können, wie sich in Abbildung 5.6.2 zeigt. Es kann dank einer bestimmten Invariante auch nicht dazu kommen, dass ein Kunde oder ein Angestellter als Wohnsitz die gleiche Addresse zugewiesen bekommt wie eine Filiale, da jedes Addressen-Objekt als einzigartig gilt aufgrund dieser Invariante.

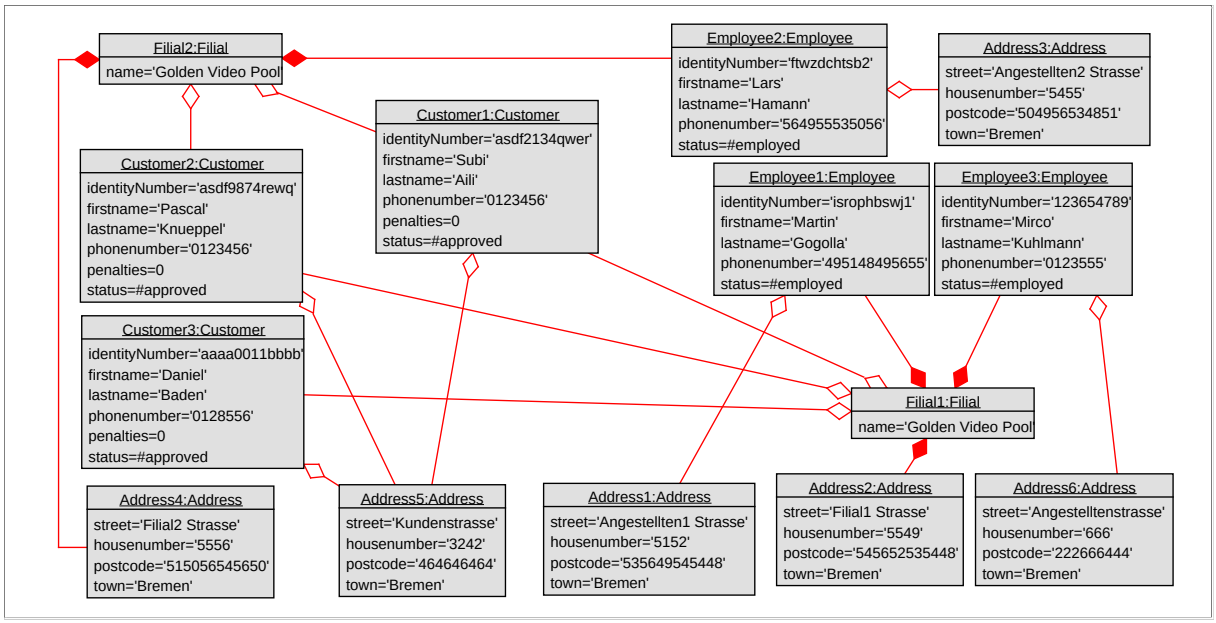

Abbildung 5.6.2: Neue Kunden mit identischen Adressen

Der Zustand der allerdings erreicht werden soll, wie im ersten Absatz dieses Szenarios beschrieben, wird jedoch erst mit den kommenden Kommandos erreicht. Dabei dient das alleinstehende Kommando aus Abbildung 5.6.3 dazu, zu zeigen, dass ein Kunde keinen Film aus einer Filiale entleihen kann, wenn alle Kopien dieser Filiale bereits verliehen worden sind. Dafür soll der Film „Auf der Jagd“ dienen. Dieser Film wurde in Filiale1 bereits vom ersten Kunden entliehen und nun möchte der 2. Kunde diesen Film ebenfalls in der gleichen Filiale entleihen, was nicht gehen wird, da jede Filiale zur Eröffnung jeweils nur eine Filmkopie zu genau jedem Filmexemplar erhalten hat. Entsprechend führt dies zur Fehlermeldung aus Abbildung 5.6.4

```
1 !Customer2.borrowMovies(Movie.allInstances()→select(m | m.title = 'Auf der Jagd'),
      Filial1)
```

Abbildung 5.6.3: szenario6-borrow-fail.soil

```
1 use> !Customer2.borrowMovies(Movie.allInstances()→select(m | m.title = 'Auf der Jagd'),
      Filial1)
2 [Error] 1 precondition in operation call 'Customer::borrowMovies(self:Customer2, movies:
      Set{Movie472}, filial:
3 Filial1)' does not hold:
4   Movies_Must_Be_Available: movies→forAll(m : Movie | filial.getAvailableCopy(m).
        isDefined)
```

Abbildung 5.6.4: Fehlermeldung wegen nicht vorhandener Filmkopie

Nun wird sich der Kunde 3 eben jenen Film in Filiale2 ausleihen, dann wird Filiale2 von diesem Film eine neue Kopie anfordern, woraufhin Kunde2 in der Filiale eintrifft und sich eben jene Kopie ebenfalls entleiht. Abbildung 5.6.5 zeigt die notwendigen Kommandos und Abbildung 5.6.6 zeigt den entsprechenden Systemzustand, der zwar etwas unübersichtlich ist, aber dennoch deutlich zeigt, dass es in Filiale 2 nun 2 Filmkopien zum Film „Auf der Jagd“ gibt, welche zudem auch beide entliehen wurden.

```
1 !Customer3.borrowMovies(Movie.allInstances()→select(m | m.title = 'Auf der Jagd'),
      Filial2)
2 !Filial2.requestMovieCopy(Movie.allInstances()→select(m | m.title = 'Auf der Jagd')→any
      (true))
3 !Customer2.borrowMovies(Movie.allInstances()→select(m | m.title = 'Auf der Jagd'),
      Filial2)
```

Abbildung 5.6.5: szenario6-part2.soil

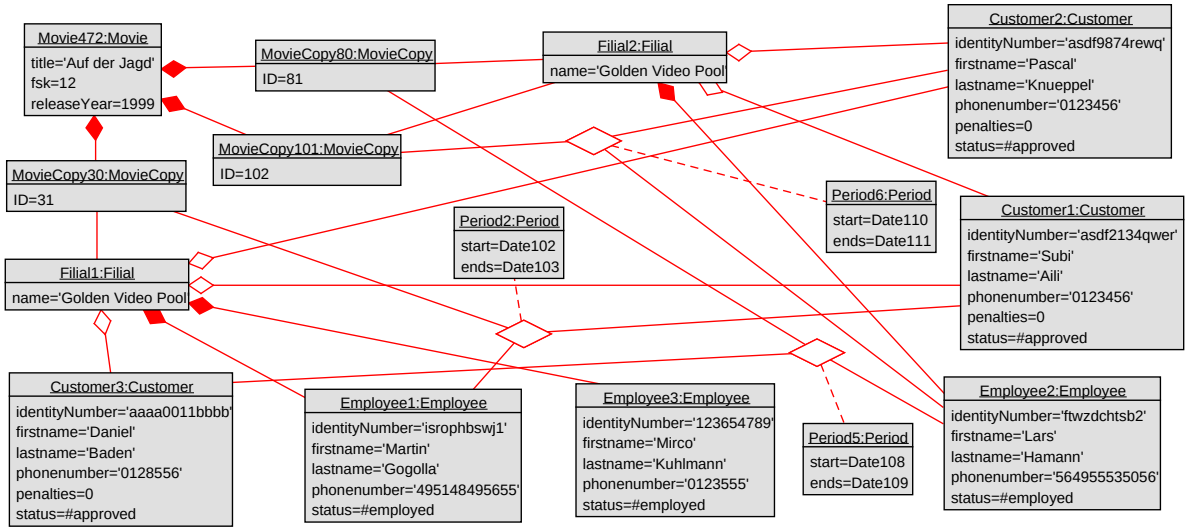

Abbildung 5.6.6: Objektdiagramm mit mehreren Filmkopien des gleichen Filmes

5.7 Szenario 7

Im 7. und vorletzten Szenario soll nun noch eine Filiale geschlossen werden. Entschieden wurde sich dabei für Filiale1 da die Schließung dieser Filiale die eine der letzten Besonderheiten des Systems aufdeckt, die bei der Schließung von Filiale2 nicht zur Geltung gekommen wären. Dabei handelt es sich jedoch nur um Kleinigkeiten, denen normalerweise kaum Beachtung geschenkt werden würde. Wenn man sich an das letzte Objektdiagramm aus Abbildung 5.6.6 erinnert, gibt es aktuell 3 Kunden. Von sind 2 Mitglied in beiden Filialen und Kunde3 ist nur in Filiale1 Mitglied. Wenn Filiale1 geschlossen wird, werden Kunde1 und Kunde2 in der Datenbank mit einer Verbindung zu Filiale2 erhalten bleiben. Der Kunde3 jedoch wird gelöscht, da dieser keinerlei Verbindung zu den anderen Filialen hat. Zusätzlich werden die Filmkopien die im Besitz von Filiale1 sind verkauft, dabei ist es hier gerade egal, ob die Filme noch entliehen sind oder nicht. Somit führt der Befehl aus Abbildung 5.7.1 zum Zustand aus Abbildung 5.7.2

```
1 !Filial1.closeFilial(Filial1);
```

Abbildung 5.7.1: szenario7.soil

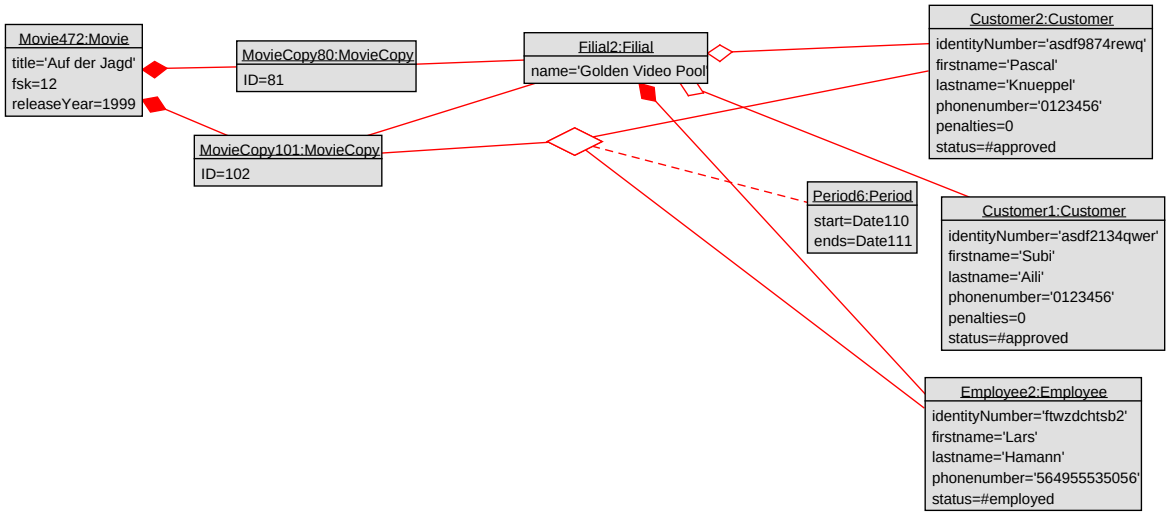

Abbildung 5.7.2: Objektdiagramm: Filiale1 wurde geschlossen

5.8 Szenario 8

Im letzten Szenario soll nun noch die letzte Besonderheit vorgeführt werden, welche zuvor in Szenario 5 bereits angesprochen, aber nicht näher erläutert wurde. Dabei handelt es sich um den speziellen Fall, wie das System reagiert, wenn ein Kunde einen Film entleihen möchte, nachdem er bereits 3 mal Filme zu spät zurückgebracht hat. Um diesen Zustand zu erreichen wird hier nun der Endzustand aus Szenario 5 wiederhergestellt. Im Anschluss entleiht der Kunde sich wieder einen neuen Film, wartet erneut 8 Tage, bringt den Film zurück und versucht sich entsprechend in der nächsten Filiale einen neuen Film zu entleihen. In Abbildung 5.8.1 lässt sich der Kunde betrachten, welcher wie auf dieser Abbildung zu sehen ist, 2 Strafen erhalten hat.

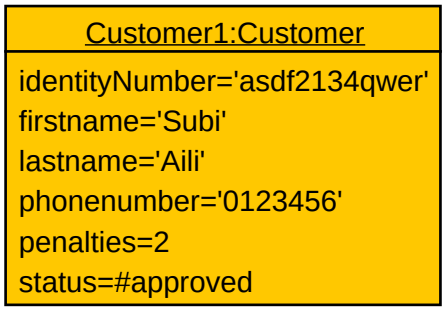

Abbildung 5.8.1: Objektdiagramm: Kunde mit 2 Strafen

Die Kommandos aus Abbildung 5.8.2 tun nun genau das, was eben beschrieben wurde und sorgen dafür, dass der Kunde seine 3. Strafe erhält, was dann den Zustand aus Abbildung 5.8.3 hervorbringt.

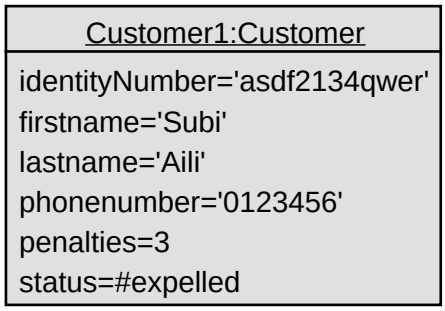

Abbildung 5.8.3: Objektdiagramm: Ausgeschlossener Kunde

```
!Customer1.borrowMovies(Movie.allInstances()→select(m | m.title = '21'), Filial2)
!currentDate.addDays(8)
!Customer1.returnMovies(Customer1.period.lentMovies→select(mc | mc.owner = Filial2)→
    asSet(),Filial2);
```

Abbildung 5.8.2: szenario8.soil

Wie man nun in Abbildung 5.8.3 sehen kann, wurde der Kunde ausgeschlossen (mit dem Status „#expelled“ markiert). Dies hat nun zur Folge, dass jeder weitere Versuch einen Film zu entleihen in einem Fehlschlag der entsprechenden pre-condition enden wird. Um zu zeigen, dass es auch egal ist, in welcher Filiale er versucht einen Film zu entleihen, wird das Ganze hier für Filiale1 gezeigt, da der Kunde den letzten Film in Filiale2 entliehen und dann wieder zurückgebracht hat.

Somit führt das Kommando aus Abbildung 5.8.4 zu der Fehlermeldung aus Abbildung 5.8.5.

```
!Customer1.borrowMovies(Set{Movie48}, Filial2)
```

Abbildung 5.8.4: szenario8-part2.soil

```
use> !Customer1.borrowMovies(Set{Movie48}, Filial2)
[Error] 1 precondition in operation call 'Customer::borrowMovies(self:Customer1, movies:
    Set{Movie48}, filial:F
ilial2)' does not hold:
  Customer_Must_Be_Approved: (self.status = CustomerStatus::approved)
```

Abbildung 5.8.5: szenario8-fail