Einführung in ObjectTeams/Java

Das Beispiel in dieser Übung findet im Kontext der Universität statt. Personen können durch Immatrikulation die Rolle Student annehmen und diese Rolle durch Exmatrikulation wieder ablegen.

ObjectTeams/Java (OT/J) ist eine Spracherweiterung der Programmiersprache Java um Konzepte der Rollen-basierten Programmierung. Klassen können daher weiterhin wie gewohnt spezifiziert werden.

Teams

Kollaborationen (Rollenmodelle) werden in OT/J als **Teams** bezeichnet. Diese können, wie Klassen, Methoden und Attribute haben. Ein Team wird mittels „`team class` *Name* { ... }“ deklariert. Die Methoden und Attribute eines Teams werden wie bei „normalen“ Klassen angegeben.

Am Beispiel:

```
team class Universität {
      String name;

      void immatrikulatieren(...) { ... }
      void exmatrikulatieren(...) { ... }
}
```

Rollen

Rollen werden als *innere Klassen* eines Teams deklariert. Jede Klasse die innerhalb eines Teams deklariert wird, wird als Rolle angesehen – daher wird kein Schlüsselwort „`role`“ o.Ä. benötigt.

```
team class Universität {
      ...
      class Student ... { ... }
      ...
}
```

Rollen können ebenso Attribute und Methoden haben, die wie bei Klassen deklariert werden können.

```
team class Universität {
      ...
      class Student ... {
            ...
            int matrikel;
            ...
            void studentIdentifizieren() {...}
            ...
      }
      ...
}
```


[1]siehe Literaturempfehlung

Binden von Rollen

Rollen benötigen einen Spieler. Dies wird in OT/J als **Binden** einer Rolle an ihre **Basis (Kernobjekt)** bezeichnet. Die Basis, d.h. der Spieler, ist entweder eine (normale) Klasse oder eine Rolle. OT/J verwendet das Schlüsselwort „`playedBy`“ um eine Rolle an eine Basis zu binden:

```
team class Universität {
      ...
      class Student playedBy Person { ... }
      ...
}

class Person {
      String name;
      ...
      void identifizieren();
}
```

Die Basis im obigen Beispiel ist eine normale Klasse, die entsprechend außerhalb des Teams `Universtät` deklariert wird.

Verhaltensintegration

Rollen verändern sowohl die Struktur als auch das Verhalten ihrer Spieler. OT/J bietet zwei Mechanismen zur Integration des Verhaltens: **callin** und **callout**.

Callin

Ein **callin** dient dem Unterbrechen von Methodenaufrufen der Basis und wird innerhalb einer Rolle deklariert. Im Beispiel hat die Klasse Person eine Methode „`identifizieren()`“ die den Namen der Person auf der Konsole ausgibt. Ein Student wird innerhalb der Universität jedoch nicht über seinen Namen, sondern über seine Matrikelnummer identifiziert. Um dies in OT/J auszudrücken, fügt man der Rolle Student zunächst eine Methode `studentIdentifizieren()` hinzu, die die Matrikelnummer eines Studenten auf die Konsole ausgibt. Dann deklariert man ein callin in der Rolle Student, welches jeden Aufruf der Methode „identifizieren“ unterbricht und vor deren Ausführung die Methode `studentIdentifizieren()` ausführt. Die Syntax hierfür ist:

```
team class Universität {
      ...
      class Student playedBy Person {
            studentIdentifizieren <- before identifizieren;

            int matrikel;
            ...

            void studentIdentifizieren() {
                  System.out.println(“Matrikelnummer: “+matrikel);
            ...
            }
      }
      ...
}
```


[1]siehe Literaturempfehlung

Ein callin leitet den Kontrollfluss also von der Basis zur Rolle um. Im obigen Beispiel wurde das Schlüsselwort `before` verwendet, welches besagt, dass die Methode auf der linken Seite **vor** der Methode auf der rechten Seite ausgeführt werden soll. Neben `before` bietet OT/J auch `replace` und `after`. Mit diesen Schlüsselwörtern kann man angeben, dass die Methode auf der linken Seite **anstatt** oder **nach** der Methode auf der rechten Seite ausgeführt werden soll.

Die Form „`Rollenmethodenname <- [replace|before|after] Basismethodenname`“ ist ein Spezialfall, der nur zulässig ist, wenn sich Rollen-Methode und Basismethode in ihrer Signatur nur im Namen unterscheiden. Die allgemeine Form ist:

```
returntype name(paramtype1 paramname1, ...) <- [replace|before|after]
returntype name(paramtype1 paramname1, ...)
```

Unterscheiden sich die Typen der Parameter oder der Rückgabe-Typ, muss zusätzlich angegeben werden, wie diese Typen miteinander in Beziehung stehen, was mit dem Schlüsselwort `with` gefolgt von einem Block getan werden kann. Für unser Beispiel reicht jedoch die einfache Form aus. Für Interessierte: siehe OTJLD[1] §4.4.

Callout

Ein **callout** dient dem Weitergeben des Kontrollflusses *von der Rolle zu ihrer Basis*, stellt in dieser Hinsicht also das Gegenteil eines callin dar. Die Syntax ist der von einem callin sehr ähnlich. Die Schlüsselwörter `replace`, `before` und `after` stehen für callouts jedoch nicht zur Verfügung. Stattdessen wir immer implizit `replace` verwendet. Der Grund hierfür ist, dass callouts nur dazu dienen von einer Rolle auf deren Basis zugreifen zu können. Eine Rollenmethode die durch ein callout an eine Basismethode gebunden ist, enthält daher i.d.R. keinen Code. Zur Verdeutlichung der Richtung des Kontrollflusses wird in der Deklaration eines callout statt einem Pfeil nach links, ein Pfeil nach rechts verwendet.

Die linke Seite einer callin- und callout-Spezifikation bezieht sich immer auf eine Rollenmethode. Die rechte Seite bezieht sich immer auf die Basis (neben Methoden können hier bei callouts auch Zugriffe auf Attribute angegeben werden). Die Richtung des Pfeils symbolisiert den Kontrollfluss.

Der lesende Zugriff auf Attribute der Basisklasse wird mit dem Schlüsselwort `get` angegeben. Die Rolle Student aus unserem Beispiel kann wie folgt erweitert werden:

```
team class Universität {
      ...
      class Student playedBy Person {
            studentIdentifizieren <- before identifizieren;
            String getName -> get name;

            void studentIdentifizieren() {...}
            abstract String getName();
      }
      ...
}
```

Neben dem callout muss eine abstrakte Methode in der Rolle deklariert werden. Diese Methode wird verwendet um auf das Attribut, welches durch das callout adressiert wird,

[1] siehe Literaturempfehlung

zuzugreifen. Analog zum lesenden Zugriff mittels `get`, wird das Schlüsselwort `set` verwendet um schreibend auf ein Attribut der Basisklasse zuzugreifen.

Verwendet man callouts auf Attributen der Basisklasse, kann man auch auf Attribute zugreifen, die als `private` markiert wurden! Da dies einem Bruch des Kapselungsprinzips entspricht, wird diese Fähigkeit von OT/J auch als „*decapsulation*“ bezeichnet. Weiterhin ist das Binden einer Rolle an eine, als privat deklarierte Basisklasse möglich, was als „*base class decapsulation*“ bezeichnet wird.

Aktivierung

Das Aufnehmen bzw. Ablegen *von Rollen* wird in OT/J als Aktivierung bezeichnet.

Die einfachste Form der Aktivierung ist die Team-Aktivierung. Jede Team-Klasse erbt die Methoden `activate()` und `deactive()`, die entsprechend für die Aktivierung bzw. Deaktivierung eines Teams genutzt werden können. Das Aktivieren eines Teams hat zur Folge, dass alle Rollen die innerhalb dieses Teams definiert sind, aktiviert werden. Das Aktivieren einer Rolle wiederum bedeutet, dass alle callins dieser Rolle aktiviert werden. D.h. das nach der Aktivierung eines Teams Methodenaufrufe ***aller*** Basisobjekte durch die definierten callins unterbrochen werden. Am Beispiel:

```
class Testprogramm {
      public static void main(String[] args) {
            Person hans = new Person(„Hans“);
            Person anja = new Person(„Anja“);

            System.out.println("--Kein Kontext--");
            hans.identifizieren(); //gibt den Namen von Hans aus
            anja.identifizieren(); //gibt den Namen von Anja aus

            Universität uni = new Universität();
            uni.activate(); //Aktivierung des Teams

            System.out.println("--Uni aktiv-----");
            hans.identifizieren(); //gibt Matrikel und Name von Hans aus
            anja.identifizieren(); //gibt Matrikel und Name von Anja aus

            uni.deactivate(); //Deaktivierung des Teams

            System.out.println("--Uni inaktiv---");
            hans.identifizieren(); //gibt den Namen von Hans aus
            anja.identifizieren(); //gibt den Namen von Anja aus
      }
}
```

Führt man dieses Programm aus, so erhält man folgende Ausgabe auf der Konsole:

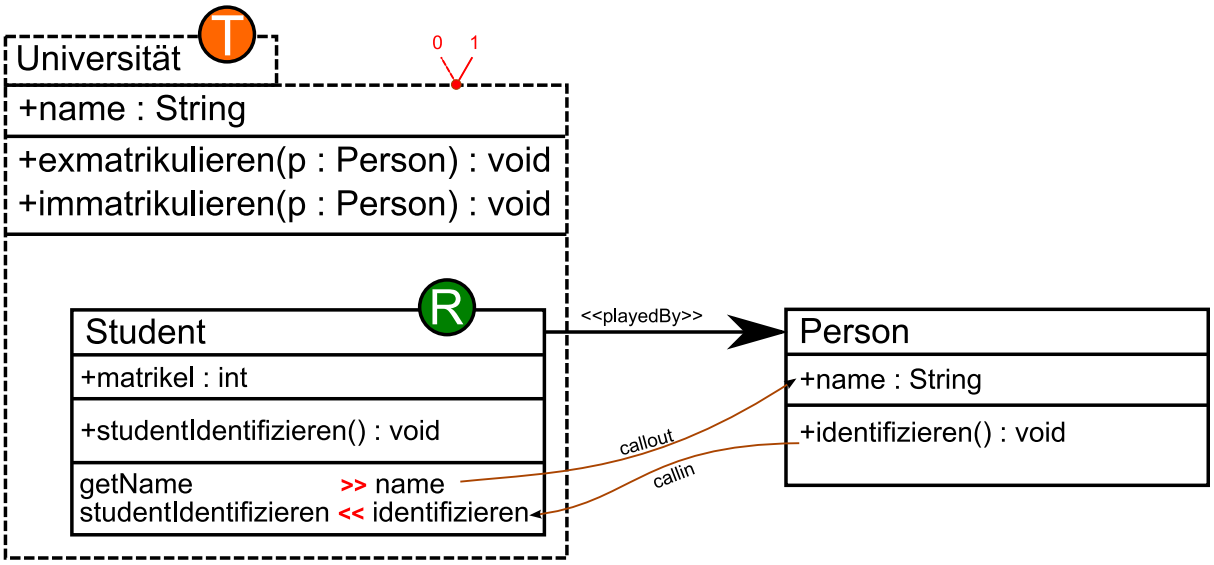
```
--Kein Kontext--
Hans
Anja
--Uni aktiv-----
Matrikelnummer: 0
Hans
Matrikelnummer: 0
Anja
--Uni inaktiv---
Hans
```


[1]siehe Literaturempfehlung

```
Anja
```

Die Aufrufe von `identifizieren()` werden, wenn der Kontext `Universität` aktiviert wurde, unterbrochen. Erst wird die Methode `studentIdentifizieren()` ausgeführt, dann die Methode `identifizieren()`.

Beide Aufruf von `identifizieren()` gaben „Matrikelnummer: 0“ gefolgt vom Namen der jeweiligen Person zurück. Der Grund für die 0 ist, dass die Instanzvariable `matrikel` nicht initialisiert wurde. Um die Matrikelnummer korrekt zu initialisieren muss zunächst das Erzeugen von Rollen-Instanzen und deren Relation zu Basisobjekten behandelt werden. Abbildung 1 im Anhang zeigt ein modifiziertes Sequenzdiagramm, welches das dynamische Verhalten des Beispiels darstellt.



Das obige UML-ähnliche Diagramm stellt das laufende Beispiel dar. Teams werden wie UML-Pakete mit einem dunkelgelben Kreis, in dem ein T steht, dargestellt. Rollen werden durch einen grünen Kreis mit einem R hervorgehoben. Der rote Schalter am Team (rechts oben) symbolisiert die Möglichkeit der De-/Aktivierung des Teams.

Rolleninstanzen

Die „Rollen“, die innerhalb eines Teams definiert werden, sind genau genommen Rollen-Typen/-Klassen. Eine Rolleninstanz kann entweder *implizit* oder *explizit* erzeugt werden. Der Normalfall ist die implizite Instanziierung, welche auch als **Lifting** bezeichnet wird.

Zur besseren Lesbarkeit wird im Folgenden „Rolle“ synonym mit „Rolleninstanz“ verwendet.

Rollentypen werden an Klassen gebunden, welche dann als Basisklassen dieses Rollentyps bezeichnet werden. Da Rollen einen Spieler benötigen, benötigt man für die Instanziierung einer Rolle immer auch ein **Basisobjekt**. Dieses Basisobjekt muss eine Instanz der Basisklasse des Rollentyps sein. Durch Polymorphie ist es möglich, dass das Basisobjekt auch von jeder Unterklasse der Basisklasse sein darf.

Merke: Ein Basisobjekt und seine Rollen bilden (nach Außen) immer eine Einheit.

[1]siehe Literaturempfehlung

Implizite Instanziierung von Rollen

Die Verhaltensmodifikation einer Basisklasse durch Rollentypen lässt sich, wie bereits gesagt, mit Hilfe der Mechanismen callin und callout spezifizieren. In unserem Beispiel haben wir folgendes callin definiert:

```
class Student playedBy Person {
      ...
      studentIdentifizieren <- before identifizieren;
      ...
}
```

Damit wird ausgedrückt, dass jedesmal, wenn die Methode `identifizieren()` auf einem ***Objekt*** der Klasse Person aufgerufen wird, die Methode `studentIdentifizieren()` einer ***zu diesem Basisobjekt gehörenden Rolleninstanz*** des Rollentyps Student aufgerufen wird. Die nötige Rolleninstanz wird dabei *implizit* erzeugt.

Merke: Durch ein callin werden implizit Rollen erzeugt.

Randbemerkung: Ein callout kann ebenfalls zur Erzeugung einer Rolleninstanz führen. Dies ist dann der Fall, wenn der Rückgabetyp der Rollenmethode ein Rollentyp ist.

Genau diese Art der Instanziierung fand bei unserem Testprogramm statt. Der Aufruf von `identifizieren()` auf dem Objekt `hans` der Klasse `Person` bei aktiviertem Team bewirkt eine Evaluation aller callins des Teams. Die Evaluation des von uns spezifizierten callins bewirkt wiederum, dass eine Rolleninstanz des Rollentyps Student für das Basisobjekt `hans` angelegt wird.

Declared Lifting

Eine weitere Form der impliziten Instanziierung von Rollen wird als „Declared Lifting“ bezeichnet. Wie im vorigen Unterabschnitt beschrieben, kennzeichnet ein callin eine Stelle an der Rollen implizit erzeugt werden.

Declared Lifting ermöglicht es, Stellen bei Parametern von Team-Methoden, an denen Rollen erzeugt werden, explizit anzugeben. Die Parameter werden mit zwei statt einem Typ spezifiziert. Der erste Typ entspricht der Basisklasse, der zweite Typ dem Rollentyp. Eine Parameterdefinition hat dann die Form „`Basistyp as Rollentyp parametername`“ Am Beispiel der Methode `immatrikulieren(..)` des Teams Universität:

```
team class Universität {
      ...
      void immatrikulatieren(Person as Student erstsemestler, int matrikel)
      {
            erstsemestler.matrikel = matrikel;
      }
      ...
}
```

Ruft man eine so definierte Methode auf und übergibt ein Basisobjekt als Parameter für das noch keine Rolleninstanz existiert, wird diese Rolleninstanz implizit erzeugt. Wenn bereits

[1]siehe Literaturempfehlung

eine Rolleninstanz für das übergebene Basisobjekt existiert, so wird keine neue Rolleninstanz erzeugt, sondern die bereits vorhandene Rolleninstanz verwendet. Weiterhin kann auch eine Rolleninstanz übergeben werden. Wenn also die Methode `immatrikulieren(..)` aufgerufen wird und eine Person als Parameter übergeben wird, die noch keine Instanz des Rollentyps Student hat, dann wird eine Instanz von Student erzeugt, welche mit dem Basisobjekt eine Einheit bildet – die übergebene Person nimmt also die Rolle Student auf.

Testprogramm

Mit Hilfe der Methode `immatrikulieren(..)` können wir nun den jeweiligen Studenten eine Matrikelnummer zuordnen. Schauen wir uns hierfür das letzte Testprogramm nochmal an und erweitern es um eine dritte Person (Peter) und die Aufrufe der Methode `immatrikulieren(..)`.

```
class Testprogramm {
      public static void main(String[] args) {
            Person hans = new Person(„Hans“);
            Person anja = new Person(„Anja“);
            Person peter = new Person(„Peter“);

            System.out.println("--Kein Kontext--");
            hans.identifizieren(); //gibt den Namen von Hans aus
            anja.identifizieren(); //gibt den Namen von Anja aus
            peter.identifizieren(); //gibt den Namen von Peter aus

            Universität uni = new Universität();
            uni.activate(); //Aktivierung des Teams
            uni.immatrikulieren(hans, 123); //Hans wird immatrikuliert
            uni.immatrikulieren(anja, 345); //Anja wird immatrikuliert
            //Peter wird nicht immatrikuliert!

            System.out.println("--Uni aktiv-----");
            hans.identifizieren(); //gibt Matrikel und Name von Hans aus
            anja.identifizieren(); //gibt Matrikel und Name von Anja aus
            peter.identifizieren(); //soll nur den Namen von Peter ausgeben
                                    //gibt jedoch Matrikel (0) und Name aus
            uni.deactivate(); //Deaktivierung des Teams

            System.out.println("--Uni inaktiv---");
            hans.identifizieren(); //gibt den Namen von Hans aus
            anja.identifizieren(); //gibt den Namen von Anja aus
            peter.identifizieren(); //gibt den Namen von Peter aus

      }
}
```

Die Ausgabe des Programms auf der Konsole ist:

```
--Kein Kontext--
Hans
Anja
Peter
--Uni aktiv-----
Matrikelnummer: 123
Hans
Matrikelnummer: 345
Anja
Matrikelnummer: 0
Peter
```


[1]siehe Literaturempfehlung

```
--Uni inaktiv---
Hans
Anja
Peter
```

Wie wir sehen werden bei aktiviertem Team die Aufrufe der Methode `identifizieren()` an die richtigen Rolleninstanzen geleitet. Abbildung 2 im Anhang zeigt ein modifiziertes Sequenzdiagramm, welches dieses Verhalten darstellt.

Der dunkelrote Kasten hebt die Instanziierung der Student-Rollen durch Declared Lifting dar. Im grünen Kasten sieht man, dass die bereits bestehenden Rollen verwendet werden.

Im roten Kasten beobachten wir jedoch auch ein unerwünschtes Verhalten – Peter, der nicht immatrikuliert wurde, wird bei aktiviertem Team auch zum Student, was sich in der Ausgabe der nicht initialisierten Matrikelnummer (0) und des Namens von Peter äußert. Der Grund hierfür ist, dass durch das callin im Rollentyp Student implizit für **alle** Basisobjekte, hier also Objekte der Klasse Person, zur Evaluation des callins, die zugehörige Rolle erzeugt wird.

Um dieses Fehlverhalten zu korrigieren betrachten wir den Aktivierungsmechanismus nochmals genauer.

Aktivierungsmechanismen

Um die Aktivierung kontrolliert zu beschränken bietet OT/J so genannte „guard predicates“.

Diese können auf verschiedenen Ebenen definiert werden: für Teams, Rollen, Rollenmethoden und callins. Der jeweiligen Definition wird ein Prädikat mit Hilfe des Schlüsselwortes „`when`“ angehangen.

Team-Level Guard Predicates

Um dies zu veranschaulichen erweitern wir unser Beispiel um den Fall, dass eine Universität geschlossen oder nicht geschlossen sein kann. Dies soll durch ein Team-Attribut „`boolean geschlossen`“ ausgedrückt werden. Die Rolle Student ergibt nur dann für Personen an dieser Universität Sinn, wenn die Universität nicht geschlossen ist. Dies lässt sich in OT/J wie folgt ausdrücken:

```
team class Universität when(!geschlossen) {
      ...
      boolean geschlossen;
      ...
      class Student playedBy Person { ... }
      ...
}
```

Ist die Universität geschlossen, findet bei einem Aufruf der Methode `activate()` auf einer Instanz des Teams Universität keine Aktivierung statt. Die Aktivierung wird somit eingeschränkt. Auf Ebene der Teams lässt sich dieses Verhalten noch recht leicht ohne *guard predicates* ausdrücken. Wenn man den Zustand der Universität im Testprogramm in Erfahrung bringen kann, so kann man an dieser Stelle die Entscheidung treffen, ob `activate()` aufgerufen werden soll oder nicht. Hinsichtlich Wartbarkeit und Verständlichkeit ist es jedoch sinnvoller diese Bedingungen beim jeweiligen Team anzugeben.

[1]siehe Literaturempfehlung

Role-Level Guard Predicates

Erweitern wir unser Beispiel um den Fall, dass Studenten Urlaubssemester einlegen können. Der Rollentyp Student wird dazu um ein Attribut „`boolean imUrlaub`“ erweitert. Wenn ein Student im Urlaub ist, verhält er sich anders, als wenn er in die Uni geht. Dies kann wie folgt ausgedrückt werden:

```
team class Universität ... {
      ...
      class Student playedBy Person when(!imUrlaub) {
            boolean imUrlaub;

            studentIdentifizieren <- before identifizieren;

            void studentIdentifizieren() {...}
      }
}
```

Wird das Team aktiviert, so werden auch sämtliche Rollen aktiviert. Für Studenten die nicht im Urlaub sind greift das callin weiterhin, wodurch die Rollenmethode weiterhin ausgeführt wird. Ist das Attribut `imUrlaub` einer Rolleninstanz `true`, greifen sämtliche callins der jeweiligen Rolleninstanz nicht mehr.

Achtung: An dieser Stelle ist es besonders wichtig den genauen Ablauf der Evaluation von *guard predicates* zu verstehen. Ein callin definiert, wie bereits beschrieben, einen Punkt an dem Rolleninstanzen implizit erzeugt werden. Wird die Methode `identifizieren()` auf einem Basisobjekt ausgeführt, greift das callin des Rollentyps Student vorerst. Für das Basisobjekt wird nun nach der entsprechenden Rolleninstanz gesucht. *Wenn noch keine Rolleninstanz vorhanden ist, wird diese erzeugt! Erst jetzt wird das guard predicate ausgewertet.* Liefert die Auswertung `false`, so wird der callin Mechanismus abgebrochen. Trotzdem wurde eine Rolleninstanz erzeugt.

Der Grund für diesen Ablauf ist, dass sich *guard predicates* auf Instanzvariablen der Rolle beziehen dürfen. Um das Prädikat auswerten zu können muss also eine Instanz vorhanden sein bzw. erzeugt werden.

Base Guard Predicates

Das Ziel in unserem Beispiel ist, dass wir selbst bestimmen wollen, wann eine Rolleninstanz von Student erzeugt wird. Dazu haben wir bereits die Methode `immatrikulieren(..)` geschrieben. Die implizite Erzeugung durch das callin wollen wir unterdrücken. OT/J bietet hierfür *base guard predicates*.

Diese werden wie normale *guard predicates* angegeben, beginnen jedoch mit dem zusätzlichen Schlüsselwort „`base`“. Für die Evaluation eines *base guard predicates*, wird keine Rolleninstanz erzeugt!

Innerhalb von *base guard predicates* kann man das Schlüsselwort „`base`“ verwenden, um auf das Basisobjekt zugreifen zu können. Da jedoch keine Rolleninstanz für die Evaluation erzeugt wird, kann entsprechend nicht auf Instanzvariablen der Rolle zugegriffen werden.

[1]siehe Literaturempfehlung

Erweitern wir unser Beispiel dahingehend, dass eine Person ein Alter hat. Dies soll durch ein Attribut „`int alter`“ ausgedrückt werden. Wenn wir voraussetzen wollen, dass ein Student mindestens 18 Jahre alt sein muss, kann dies wie folgt spezifiziert werden:

```
class Person {
      int alter;
      ...
}

team class Universität ... {
      ...
      class Student playedBy Person base when(base.alter >= 18) {
            ...
      }
}
```

Reflektive Methoden des Teams

Die Klasse Team bietet eine Reihe reflektiver Methoden. In dieser Übung interessiert uns nur eine dieser Methoden – die Methode `boolean hasRole(Basisobjekt, Rollentyp)`. Diese Methode kann auf einer Teaminstanz aufgerufen werden und gibt Auskunft darüber, ob für das übergebene Basisobjekt eine Rolleninstanz vom übergebenen Rollentyp existiert.

Genau das benötigen wir um auszudrücken, dass unser callin nur dann aktiv sein soll, wenn die Person bereits Student ist. In unserem Beispiel sieht die Definition des Rollentyps Student daher wie folgt aus:

```
team class Universität ... {
      ...
      class Student playedBy Person
            base when(Universität.this.hasRole(base, Student.class)) {
            ...
      }
}
```

Das Finale Testprogramm

Betrachten wir nun das letzte Testprogramm mit den drei Personen Hans, Anja und Peter. Hans und Anja werden an der Universität immatrikuliert - Peter nicht. Führen wir genau dasselbe Testprogramm mit dem neu hinzugefügten *base guard predicate* aus, erhalten wir folgende Ausgabe:

```
--Kein Kontext--
Hans
Anja
Peter

--Uni aktiv-----
Matrikelnummer: 123
Hans
Matrikelnummer: 345
Anja
Peter
--Uni inaktiv---
Hans
Anja
Peter
```


[1]siehe Literaturempfehlung

Peter wird also auch bei aktiviertem Team nicht mehr als Student behandelt, sondern als Person. Abbildung 3 im Anhang zeigt ein modifiziertes Sequenzdiagramm, welches diesen Ablauf darstellt.

Zusammenfassung

Es wurden zunächst die grundlegenden Konzepte **Team**, **Rolle**, **callin** und **callout** eingeführt. Anschließend wurde die allgemeine Team-**Aktivierung** eingeführt und aufgezeigt, dass diese auf Klassen-Ebene stattfindet. Auf dem Weg zur Instanz-basierten Aktivierung wurde zunächst die Erzeugung von Rolleninstanzen näher betrachtet, wobei die Konzepte **Lifting** und **Declared Lifting** eingeführt wurden. Abschließend wurden das Konzept der **Guard Predicates** und **Base Guard Predicates** zusammen mit einer Hilfsmethode der OT/J-API eingeführt, wodurch eine Instanz-basierte Aktivierung möglich wurde.

Diese Einleitung in ObjectTeams/Java umfasst nur einen Teil des gesamten Sprachumfangs. Eine vollständige Beschreibung der Sprache liegt in Form der ObjectTeams/Java Language Definition [OTJLD] vor.

Literaturempfehlung:

[OTJLD] Stephan Herrmann, Christine Hundt, Marco Mosconi: *ObjectTeams/Java Language Definition Version 1.2*. TU Berlin. 2008 (http://objectteams.org/def/1.2/OTJLDv1.2-current.pdf)

[SH07] Stephan Herrmann: *A Precise Model for Contextual Roles: The Programming Language ObjectTeams/Java.* Applied Ontology, Volume 2, Number 2 / 2007, pp. 181-207, IOS Press. 2007 (http://objectteams.org/publications/JAO07.pdf)

[1]siehe Literaturempfehlung

Anhang

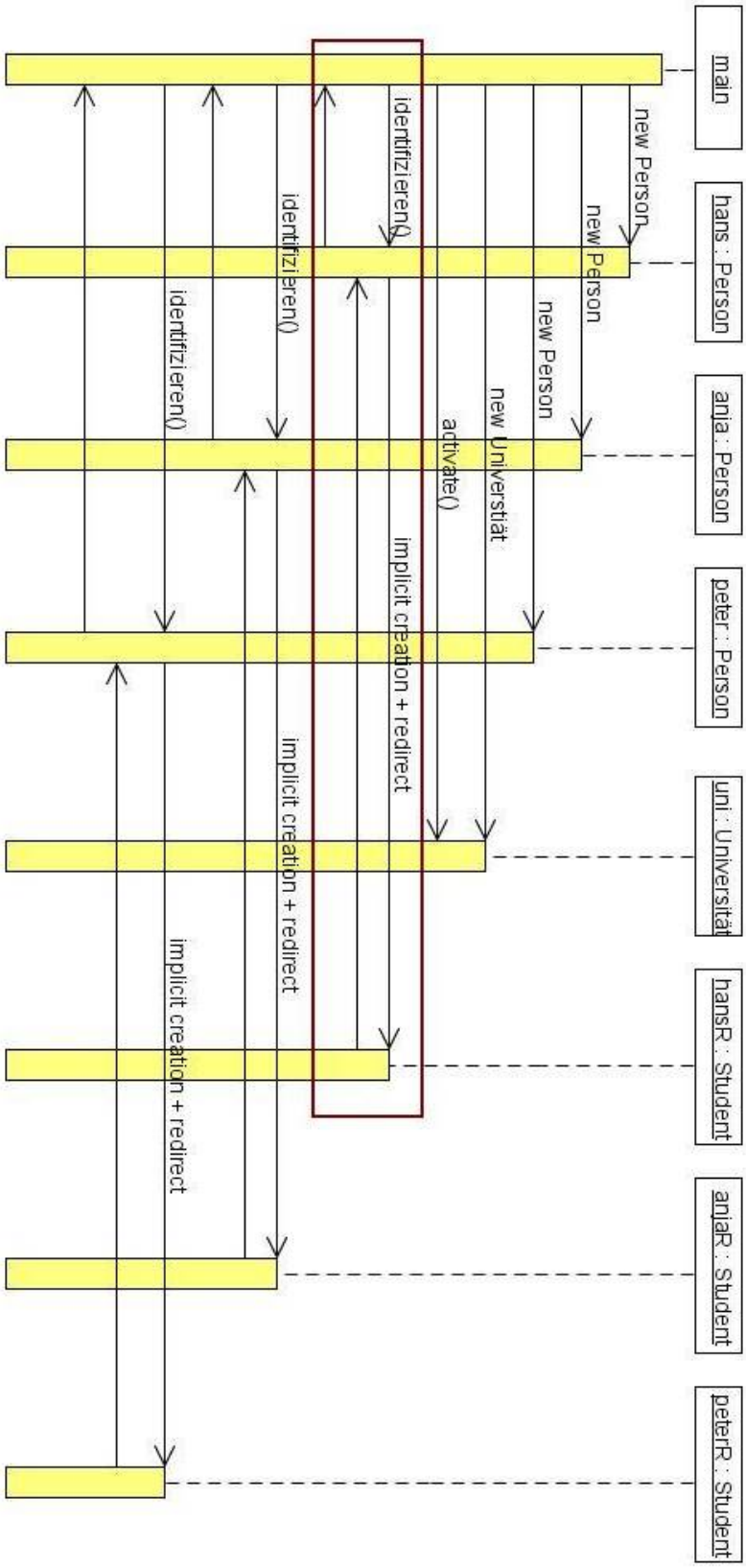


Abbildung 1: Ablauf der ersten Version des Beispiels. Die Rollen werden implizit, durch Evaluation der callins, erzeugt.

[1]siehe Literaturempfehlung

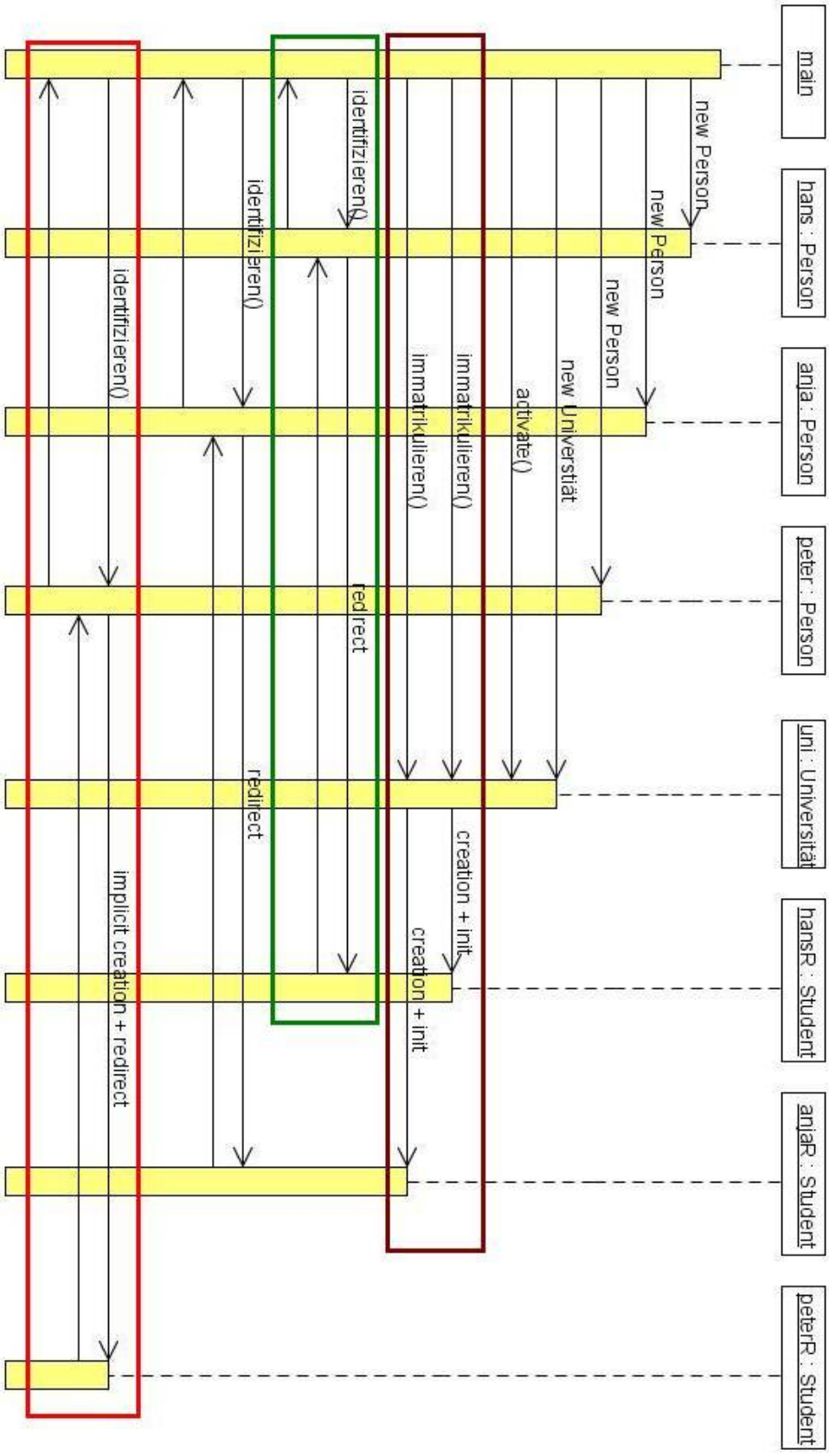


Abbildung 2: Ablauf der zweiten Version des Beispiels. Hans und Anja werden immatrikuliert. Per Declared Lifting werden die Student-Rollen für Hans und Anja erzeugt. Peters Student-Rolle wird jedoch weiterhin implizit, durch Evaluation des callins, erzeugt.

[1]siehe Literaturempfehlung

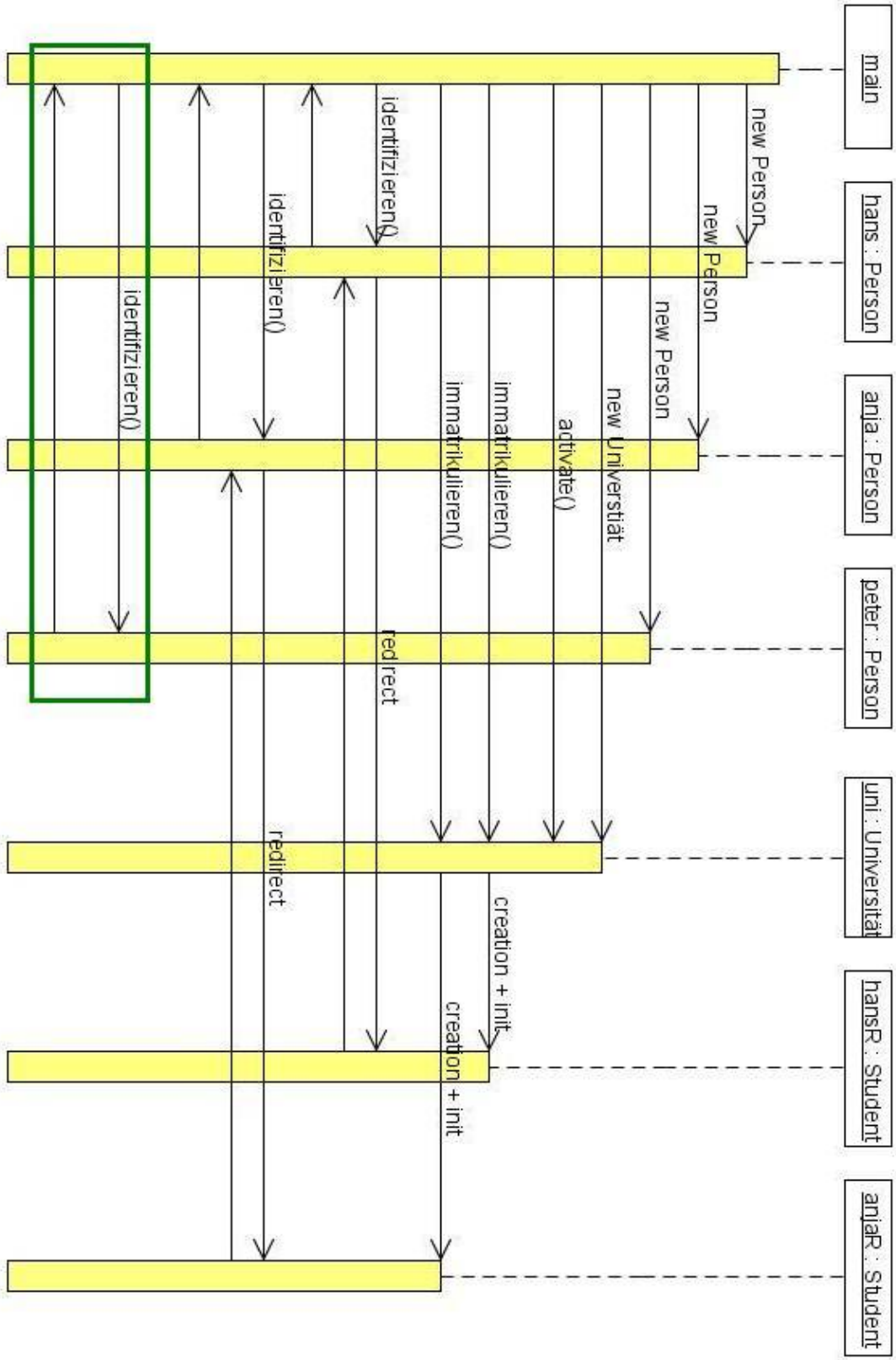


Abbildung 3: Ablauf der dritten Version des Beispiels. Hans und Anja werden immatrikuliert. Per Declared Lifting werden die Student-Rollen für Hans und Anja erzeugt. Peters Student-Rolle wird nun nicht mehr implizit erzeugt, da die Evaluation des callins durch ein „base guard predicate“ verhindert wird.

[1]siehe Literaturempfehlung