UNIKASSEL VERSITÄT

Research Group Software Engineering

Programmiermethodik

Übung 4

Sommersemester 2010

Fachgebiet Software Engineering

Andreas Scharf

andreas.scharf@cs.uni-kassel.de

Agenda

- **Organisatorisches**
- **Besprechung HA 3**
- **Implementierung Klassendiagramm**
  - Klassen
  - Attribute
  - Methoden
  - Assoziationen
- **Nochmal: JUnit4**
- **Besprechung HA 4**
- **Praktische Übung: Mau-Mau Klassendiagramm implementieren**

Organisatorisches

- **Abgaben ohne Anhang werden nicht mehr berücksichtigt**
- **Im PDF Namen ALLER Gruppenmitglieder**
- **Möglichst alle Teile des Dokuments (ausser Code) in ein PDF**
- **In die Mail Betreffzeile alle Matrikelnummern aufnehmen. Bsp.:**

PMSS2010 HA4 1233456,19838439,3828920

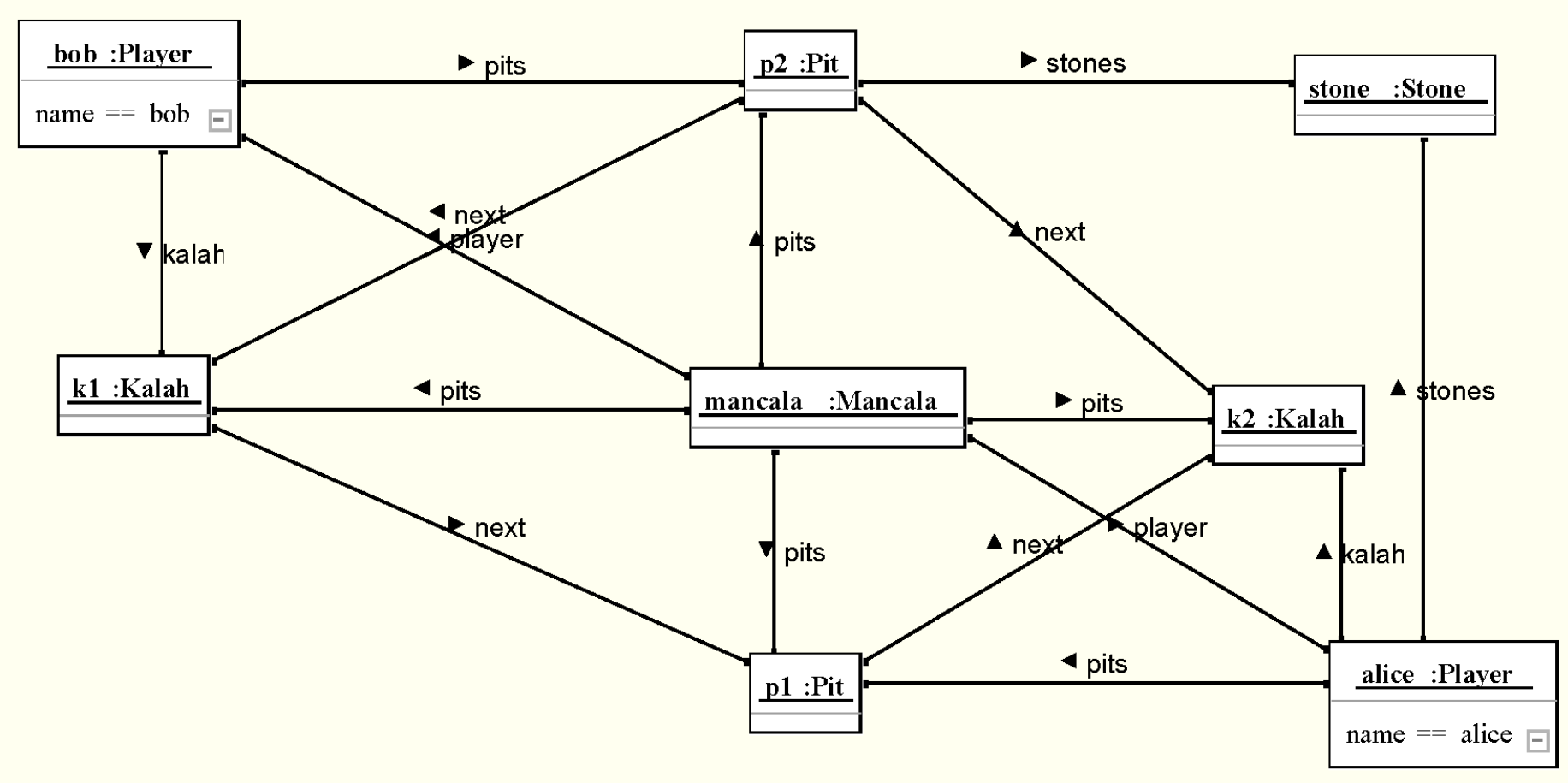
- **Eclipse Projekte genauso benennen. Bei mehreren Projekten:**

PMSS2010 HA4 1233456,19838439,3828920 **1.1**

PMSS2010 HA4 1233456,19838439,3828920 **1.2**

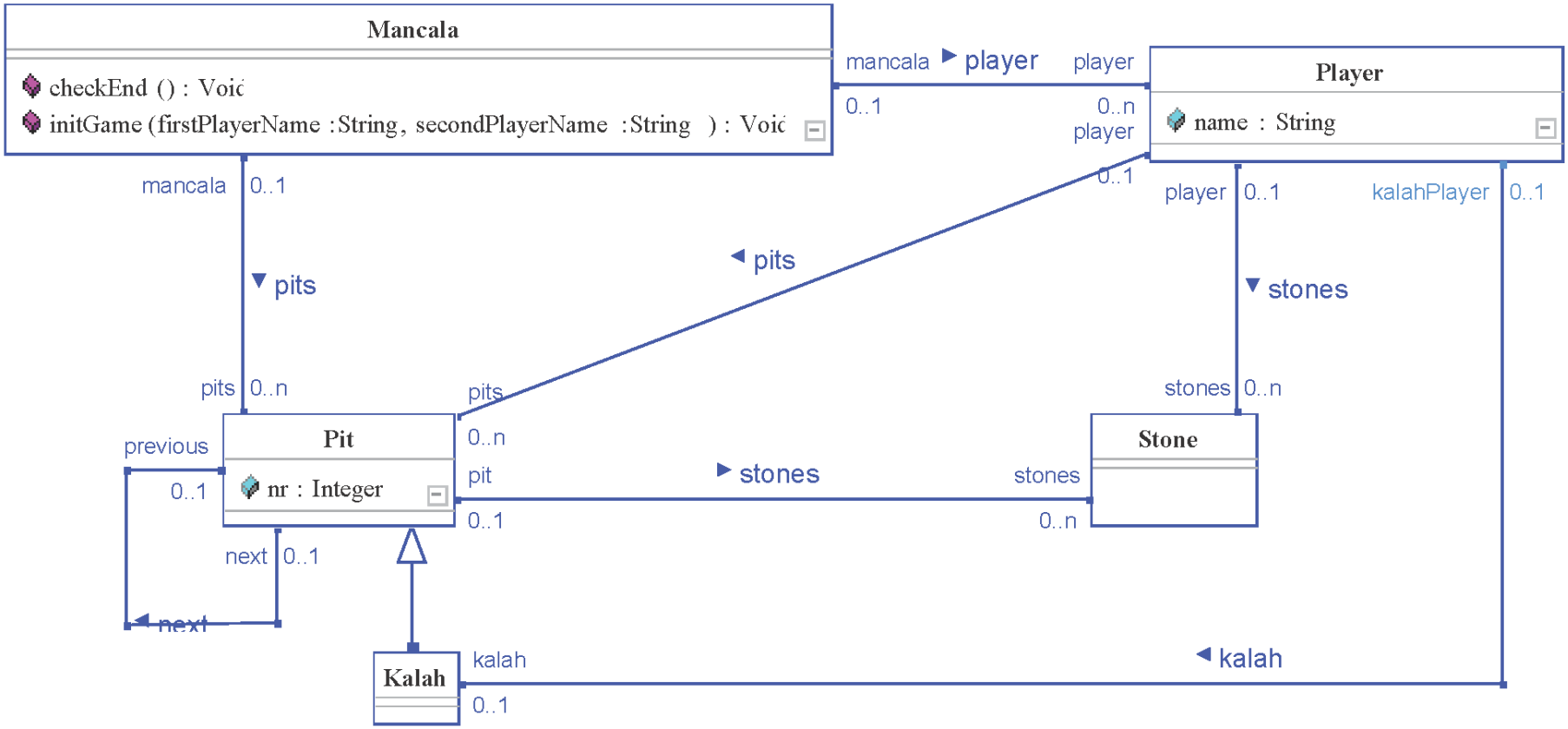
Besprechung HA 3 I

- **Aufgabe 2: Mögliches Mancala Objektdiagramm (stark vereinfacht!)**



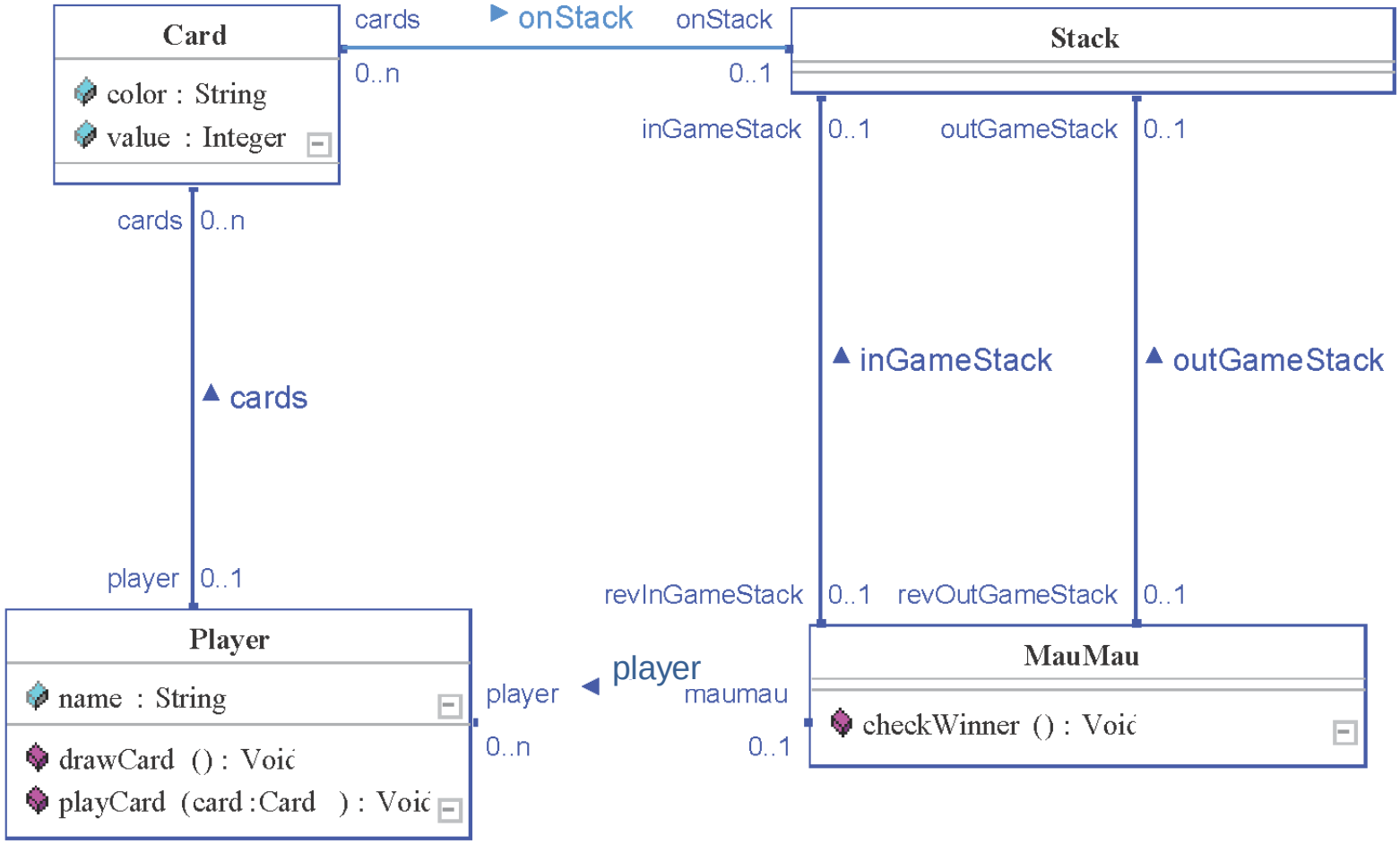
Besprechung HA 3 II

- **Aufgabe 3: Dazugehöriges Klassendiagramm**



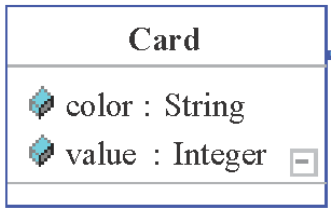
Implementierung Klassendiagramm I

- **Vorgehen bei Implementierung eines Klassendiagramms**



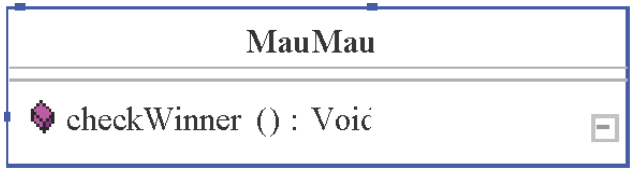
Implementierung Klassendiagramm II

- **Klassen erstellen**



Card.java

```
public class Card
{

}
```

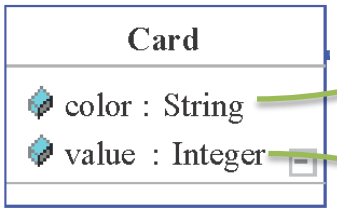


MauMau.java

```
public class MauMau
{

}
```

Implementierung Klassendiagramm III

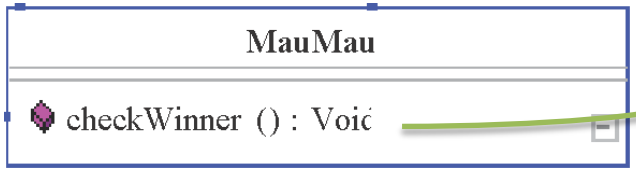
- **Attribute implementieren**




```
public class Card
{
    private String color;

    public String getColor()
    {
        return this.color;
    }

    public void setColor(String color)
    {
        this.color = color;
    }

    private int value;

    public String getValue()
    {
        return this.value;
    }

    public void setValue(int value)
    {
        this.value = value;
    }
}
```

Implementierung Klassendiagramm IV

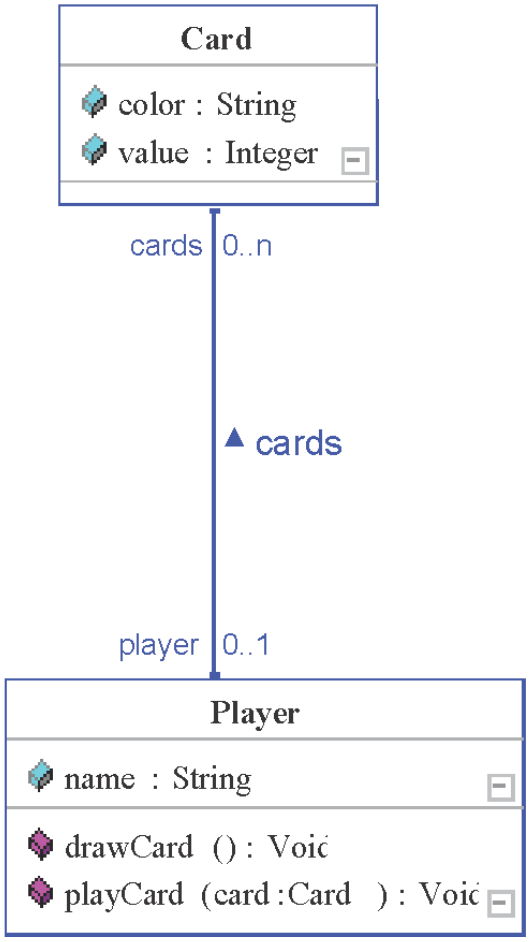
- **Methoden implementieren**




```
public class MauMau
{
    public void checkWinner()
    {
        ...
    }

}
```

Implementierung Klassendiagramm V

- **Assoziationen implementieren**
  - Schwieriger. Mehrere Fälle: zu-1 und zu-n
  - Bei zu-n:
    - Entscheidung welche Containerklasse



Implementierung VI: Collections

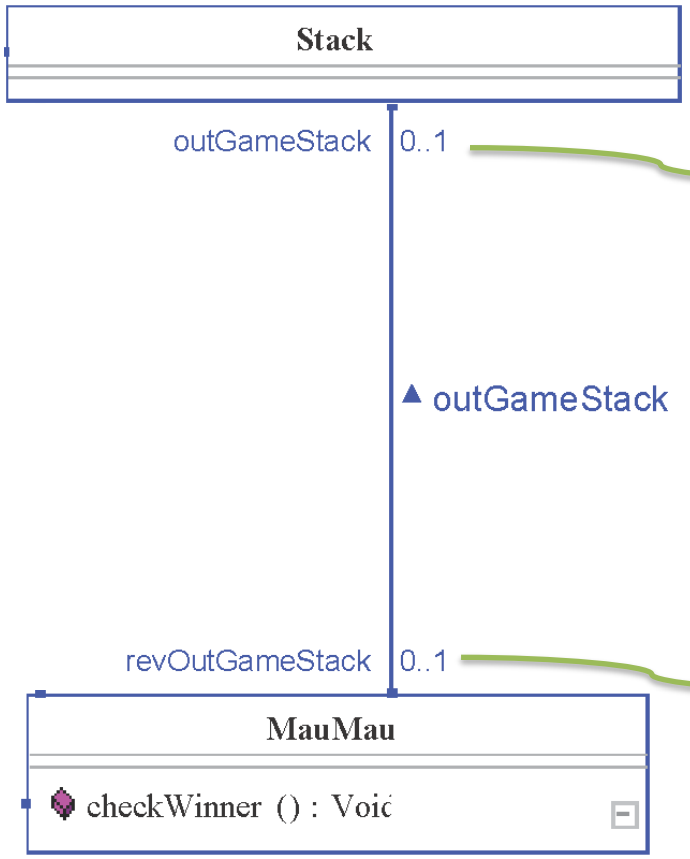
- **import java.util.***
- **ArrayList** (unsynchronized) **oder Vector** (synchronized)
  - Implementiert java.util.List (und somit java.util.Collection)
  - Basiert auf Arrays, aber vergrößert/verkleinert sich automatisch

  → **Erlaubt Duplikate**
- **HashSet**
  - Implementiert java.util.Set (und somit java.util.Collection)
  - Menge mit hash-Zugriff

  → …**Keine Duplikate**

Implementierung Klassendiagramm VII

- **Assoziationen – Einfacher Fall: 1-zu-1**

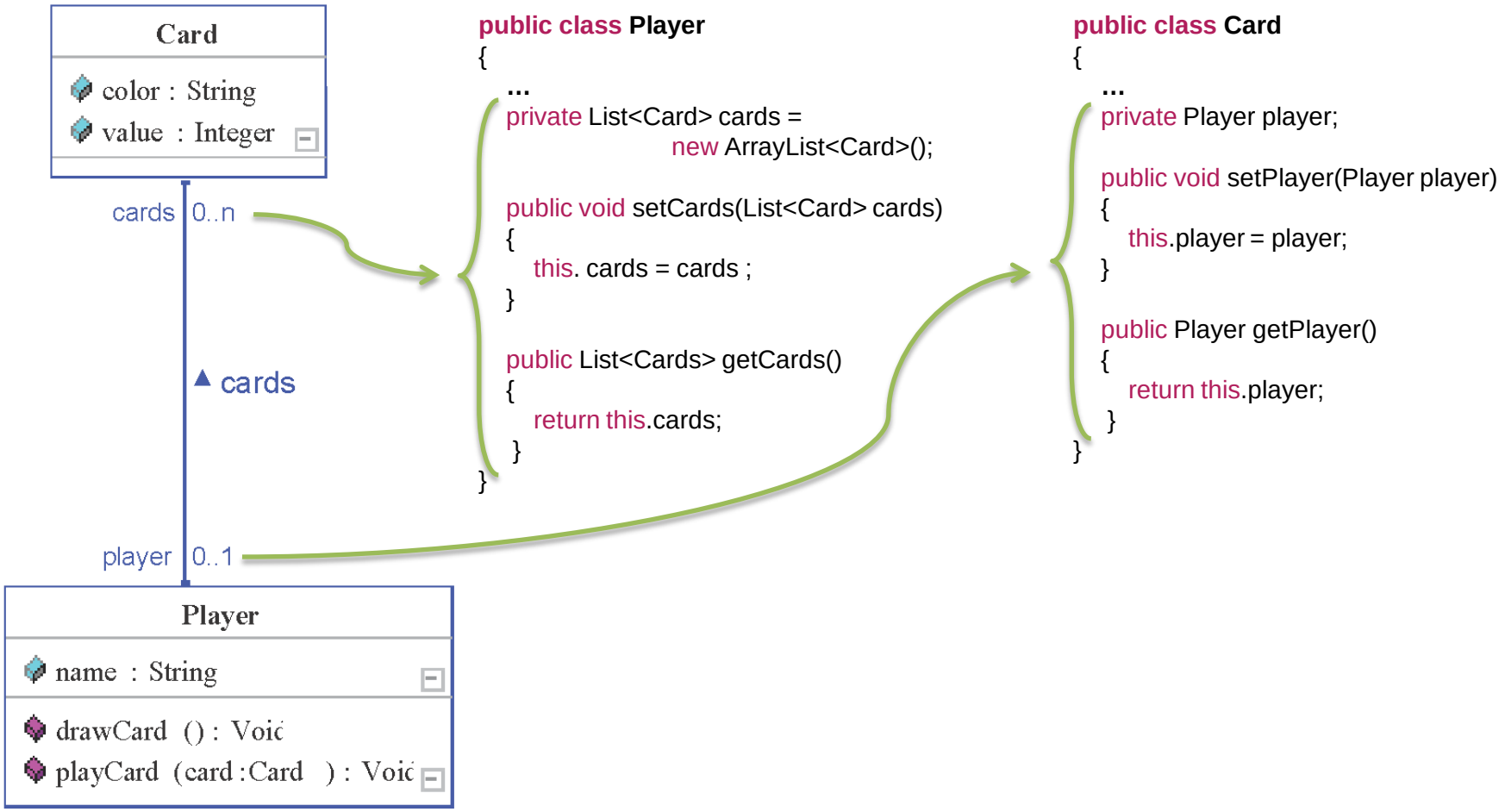



```
public class MauMau
{
    private Stack outGameStack;

    public Stack getOutGameStack()
    {
        return this.outGameStack;
    }

    public void setOutGameStack (Stack outGameStack)
    {
        this. outGameStack = outGameStack;
    }
}
```

```
public class Stack
{
    private MauMau revOutGameStack;

    public MauMau getRevOutGameStack()
    {
        return this.revOutGameStack;
    }

    public void setRevOutGameStack (MauMau revOutGameStack)
    {
        this.revOutGameStack = revOutGameStack;
    }
}
```

Implementierung Klassendiagramm VIII

- **Schwieriger: zu-n. Einfache Implementierung**




```
public class Player
{
   ...
   private List<Card> cards =
                        new ArrayList<Card>();

   public void setCards(List<Card> cards)
   {
      this. cards = cards ;
   }

   public List<Cards> getCards()
   {
      return this.cards;
    }
}
```

```
public class Card
{
   ...
   private Player player;

   public void setPlayer(Player player)
   {
      this.player = player;
   }

   public Player getPlayer()
   {
      return this.player;
    }
}
```

Implementierung Klassendiagramm IX

```
public class Player
{
   ...
   private List<Card> cards =
                        new ArrayList<Card>();

   public void setCards(List<Card> cards)
   {
      this. cards = cards ;
   }

   public List<Card> getCards()
   {
      return this.cards;
   }
}
```

```
public class Card
{
   ...
   private Player player;

   public void setPlayer(Player player)
   {
      this.player = player;
   }

   public Player getPlayer()
   {
      return this.player;
    }
}
```

- **Erwartung nach Ausführung von**

```
public static void main(String[] args)
{
   Player p1 = new Player();
   Card c1 = new Card();
   p1.getCards().add(c1);

   Player playerOfCard = c1.getPlayer();
}
```

Implementierung Klassendiagramm X

- **Mögliche Lösung:**

```
public static void main(String[] args)
{
    Player p1 = new Player();
    Card c1 = new Card();
    p1.getCards().add(c1);

    c1.setPlayer(p1);

    Player playerOfCard = c1.getPlayer();
}
```

- **Probleme?**
  - Fehleranfällig (vergisst man oft)

Implementierung Klassendiagramm XI

- **Besser: Methoden zum Hinzufügen/Setzen und Entfernen anbieten und Rückrichtung automatisch setzen.**

```
public class Player
{
   private List<Card> cards = new ArrayList<Card>();

   public void setCards(List<Card> cards)
   {
      this. cards = cards ;
   }

   public List<Card> getCards()
   {
      return this.cards;
   }

   public void addCard (Card card)
   {
      if(getCards().add(card))
      {
          card.setPlayer(this);
      }
   }

   public void removeCard(Card card)
   {
      if(getCards().remove(card))
      {
          card.setPlayer(null);
      }
   }
   …
}
```

```
public class Card
{
   private Player player;

   public void setPlayer(Player player)
   {
     if(this.player != player)
     {
        if(getPlayer() != null)
        {
           getPlayer().removeCard(this);
        }

        this.player = player;

        if(getPlayer() != null)
        {
          getPlayer().addCard(this);
        }
     }
   }

   …
}
```

Nochmal: JUnit4 I

- **Annotationen**
  - @Test (Markiert Methode als JUnit4 Test)
  - @Before (Methode wird vor jedem Test ausgeführt)
  - @After (Methode wird nach jedem Test ausgeführt)
  - @BeforeClass (Methode muss statisch sein. Wird einmalig am Anfang ausgeführt)
  - @AfterClass (Methode muss statisch sein. Wir einmalig am Ende ausgeführt)
- **Assertions**
  - assertTrue(), assertFalse()
  - assertEquals(), assertSame()
  - assertNull(), assertNotNull()
  - …

Nochmal: JUnit4 II

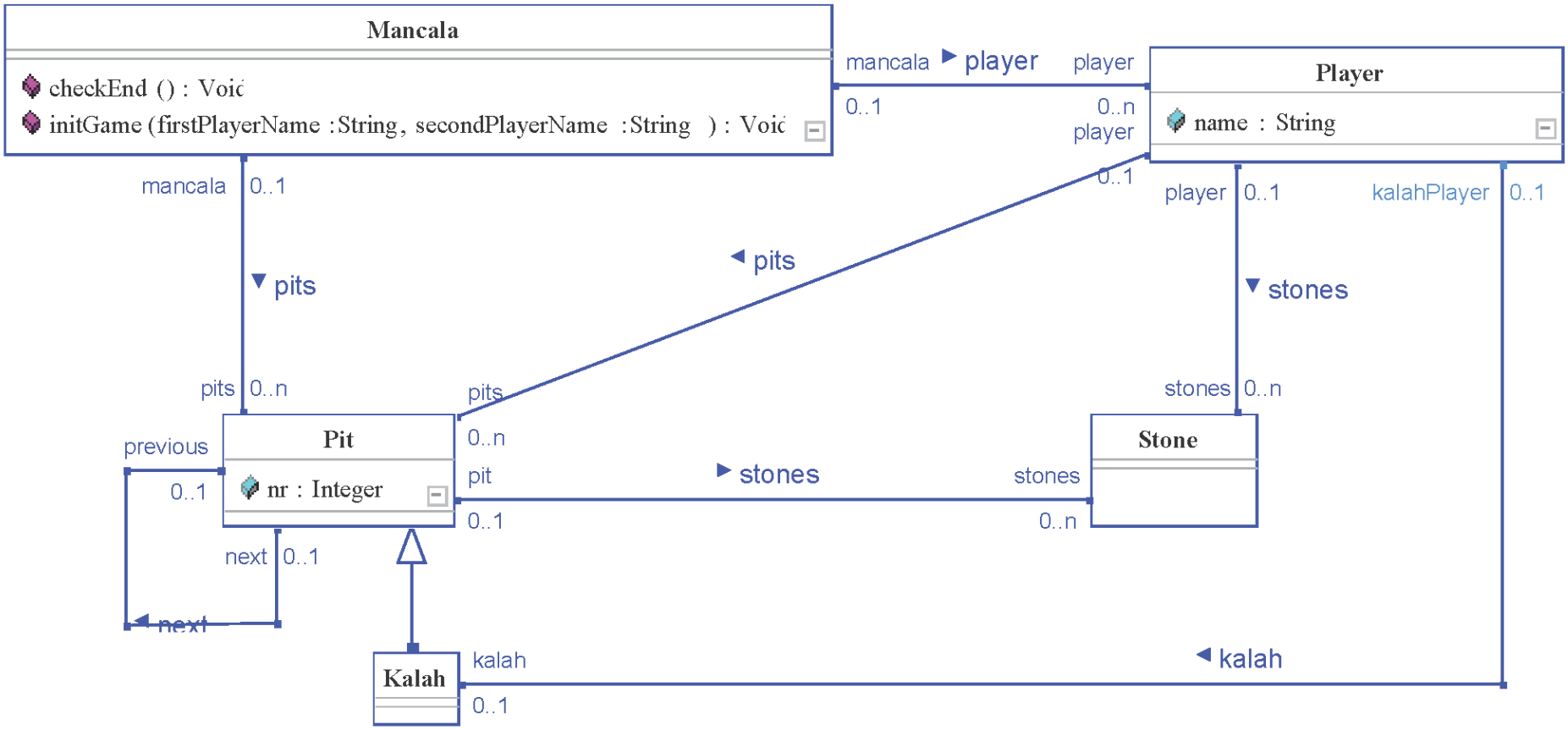
- **Testen von Fehlerfällen, z.B.**

```
public class JUnitTests
{

   @Test
   public void testExpectedFailure()
   {
      try {
         failureSituation();
         Assert.fail();
      } catch(ExcpectedException e){
         // Erwarteter fehler
      }


   }
   …
}
```

```
public class JUnitTests
{

   @Test
   public void testExpectedFailure2()
   {
      try {
         failureSituation();
      } catch(ExcpectedException e){
         // Erwarteter fehler
         return;
      }

       fail();
   }
   …
}
```

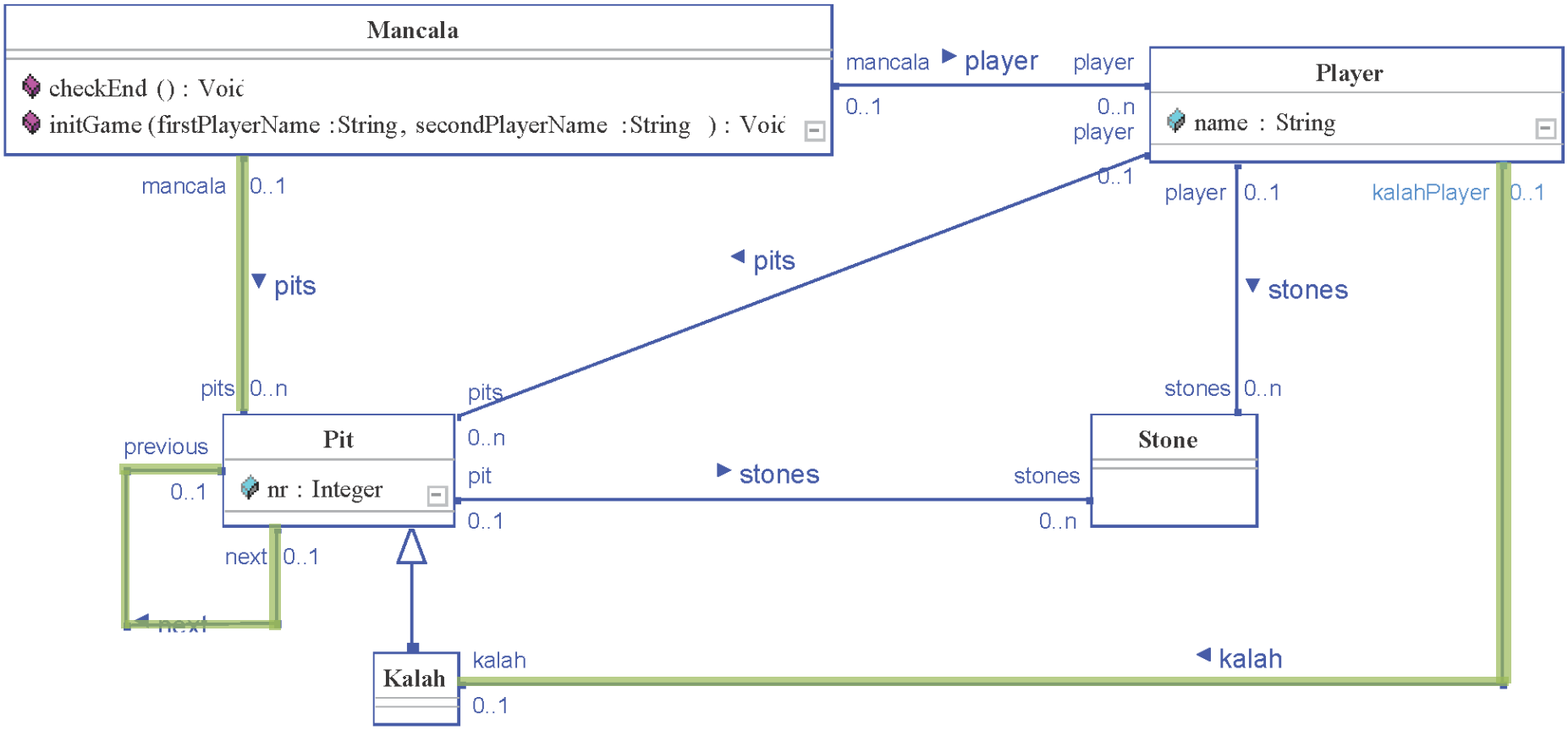
Besprechung HA 4 I

- **Aufgabe 1: Implementierung des Klassendiagramms**



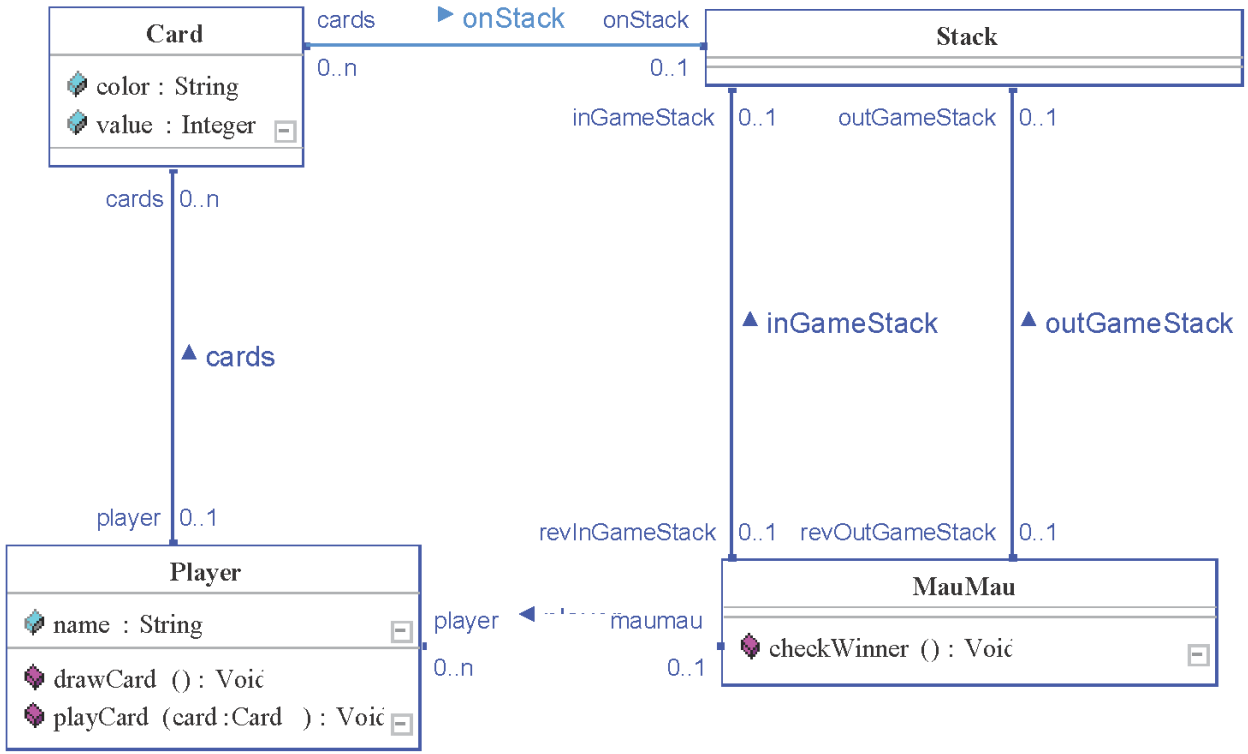
Besprechung HA 4 II

- **Aufgabe 2: JUnit Tests zu referentieller Integrität**



Übung: Mau-Mau Klassendiagramm implementieren

- Implementiert das Klassendiagramm zu Mau-Mau
- Logging (Stichpunktartig Protokoll führen)
- Pair Programming
- Abwechseln!



Ende

Schönes WE!