UNIKASSEL VERSITÄT

Research Group Software Engineering

Programmiermethodik

Übung 2

Sommersemester 2011

Fachgebiet Software Engineering

Andreas Scharf
andreas.scharf@cs.uni-kassel.de

Andreas Koch
andreas.koch@cs.uni-kassel.de

Agenda

- **Organisatorisches**
- **Besprechung HA 1**
- **JUnit4**
- **Implementierung Klassendiagramm**
  - Klassen
  - Attribute
  - Methoden
  - Assoziationen
- **Vorstellung HA 2**

Organisatorisches

- **Abgabefrist beachten**
- **Abgaben nur als *.jar oder *.zip möglich**
- **Größenbeschränkung wurde erhöht**

Im Notfall an
pm@cs.uni-kassel.de
schreiben

Besprechung HA 1 – Aufgabe 1 I

- **Aufgabe 1 – Abstrakt vs. Konkret**

Abstrakt	Konkret
Baum	Birke
Fahrzeug	Silberner Audi R8
Tier	Schweizer Bergziege
…	….

…aber auch

Backwaren aus der Cafeteria der Universität Kassel Standort WA	Das Mohnbrötchen belegt mit einer Gurke, zwei saftig grünen Blättern Salat und zwei Scheiben schmackhaftem Schinken auf einem Bett von cremiger Butter zum Preis von 1,10 Euro in der Auslage

Besprechung HA 1 – Aufgabe 1 II

- **Aufgabe 1 - Abstrakt vs. Konkret: Was gibt‘s zu verbessern?**

Abstrakt	Konkret
Suchmaschine	Beim Eingeben der Wörter „Uni Kassel“ erscheint als 1. die Homepage der Uni-Kassel.
Viele Politiker, die viele Gesetze aufstellen und sehr viel reden	Angela Merkel – Deutsche Bundeskanzlerin

z.B.:

Suchmaschine	Google
Berühmte Person	Angela Merkel – Deutsche Bundeskanzlerin

Besprechung HA 1 – Aufgabe 1 III

- **Aufgabe 1 - Abstrakt vs. Konkret**
  - Es gibt verschiedene Abstraktionslevel, z.B.:

Abstrakt	Konkret
Pflanze	Baum
Baum	Birke

  - **Abstrakt**: Von (lat. *abstractus* – „abgezogen“). Bezeichnet meist das Weglassen von Einzelheiten und das Überführen auf etwas Allgemeineres oder Einfacheres.
  - **Konkret:** Von (lat. *concretus* „dicht, fest“). Bezeichnet etwas, das *wirklich, greifbar, bestimmt, gegenständlich* ist.

Besprechung HA 1 – Aufgabe 2 I

- **Aufgabe 2 – Textuelle Szenarien**
- **Bestehen aus:**
  - Titel
  - Startsituation
  - Ablauf
  - Endsituation
- **Szenarien sollten voneinander verschiedene Situationen beschreiben**
- **Sollten nicht voneinander abhängig sein**

Besprechung HA 1 – Aufgabe 2 II

- **Beispiel:**

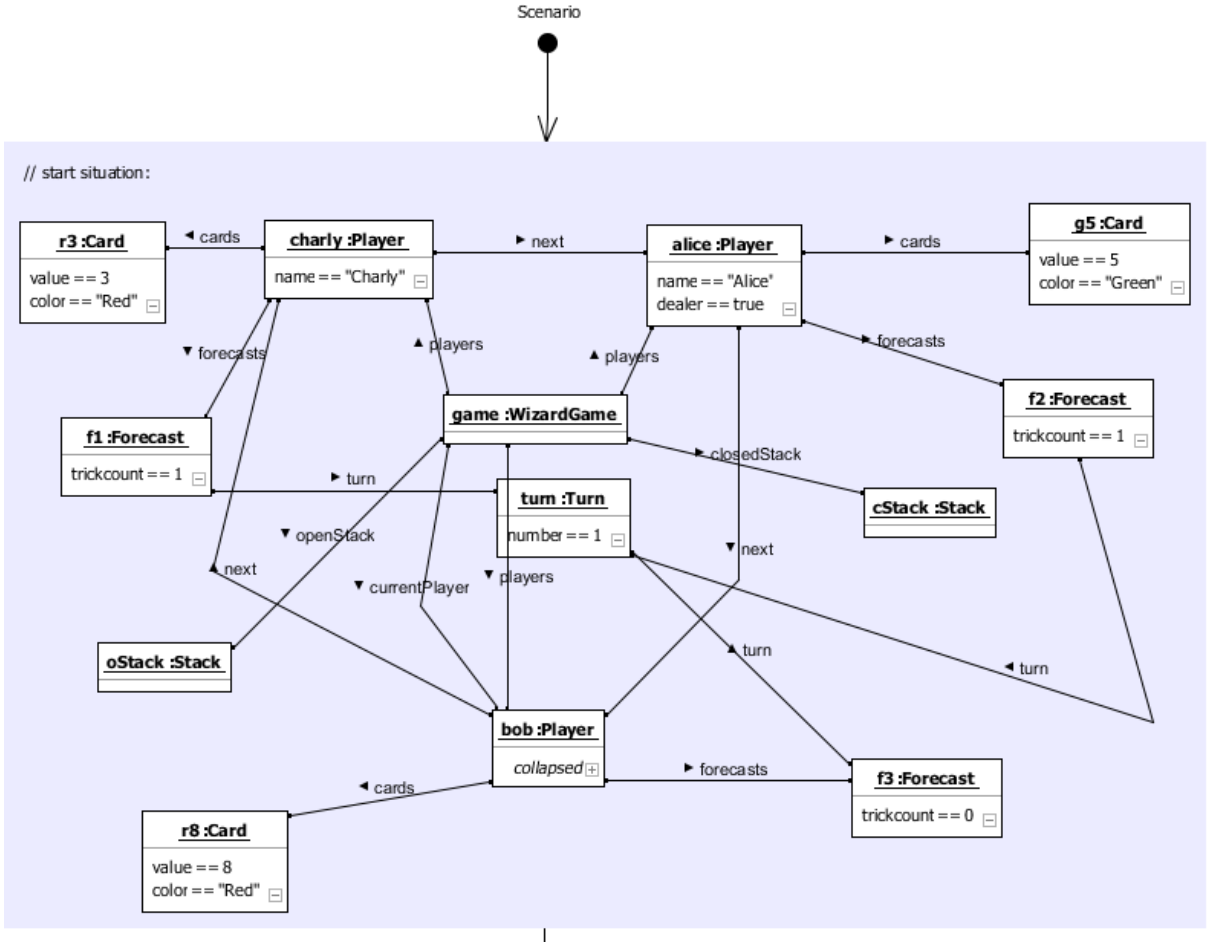
Szenario: Playing a card

Alice, Bob and Charlie are playing Wizard. Alice is the dealer. Bob sits next to Alice on her left side. Charlie sits next to Bob on his left side. All three players hold one card and have stated their biddings. Alice holds the green five, Bob the red eight and Charlie the red three. It's Bobs turn. He plays the red eight as lead card. The lead card is now the red eight. Bobs turn is finished and it's Charlies turn.

  - Titel
  - Startsituation
  - Ablauf
  - Endsituation

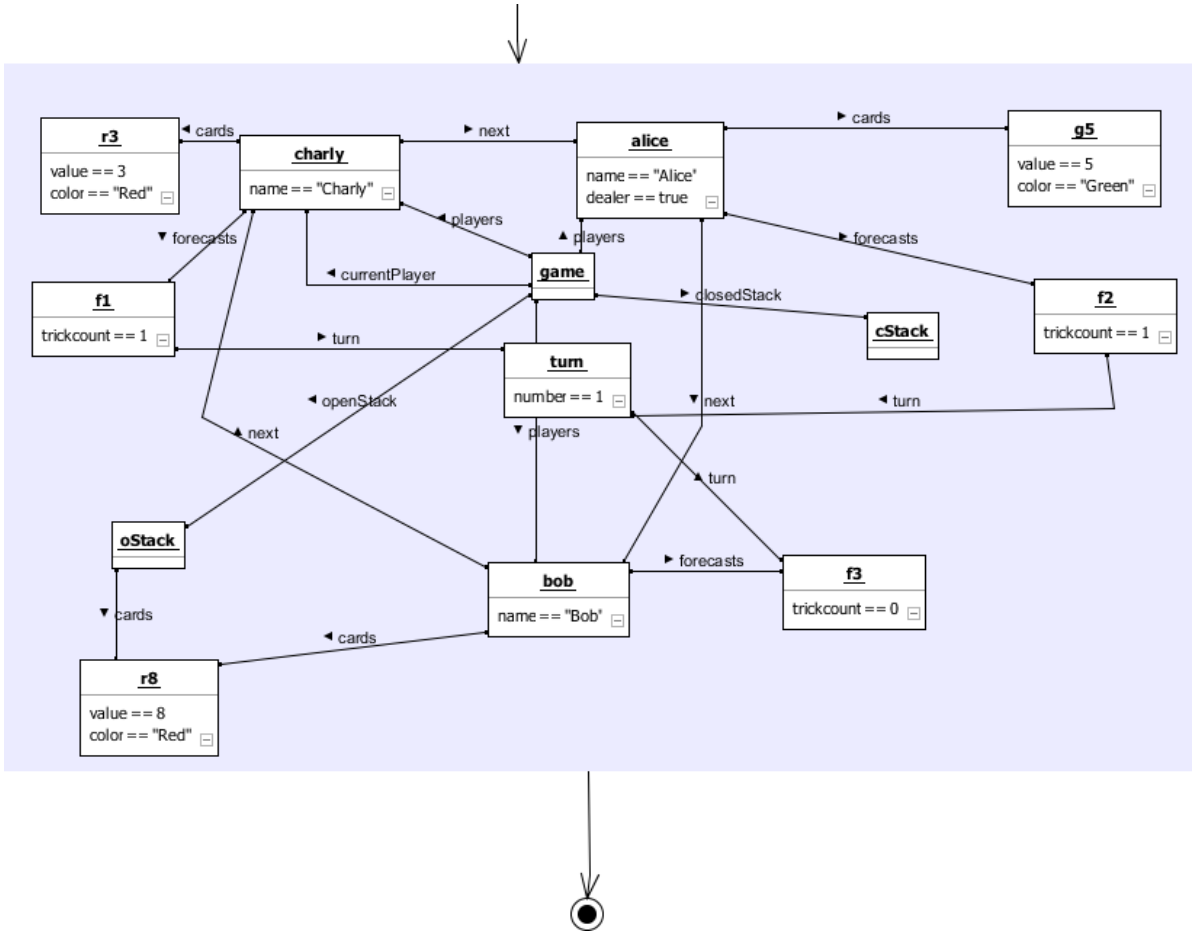
Besprechung HA 1 – Aufgabe 3 I

- **Aufgabe 3: Mögliches Wizard Objektdiagramm – Startszenario**



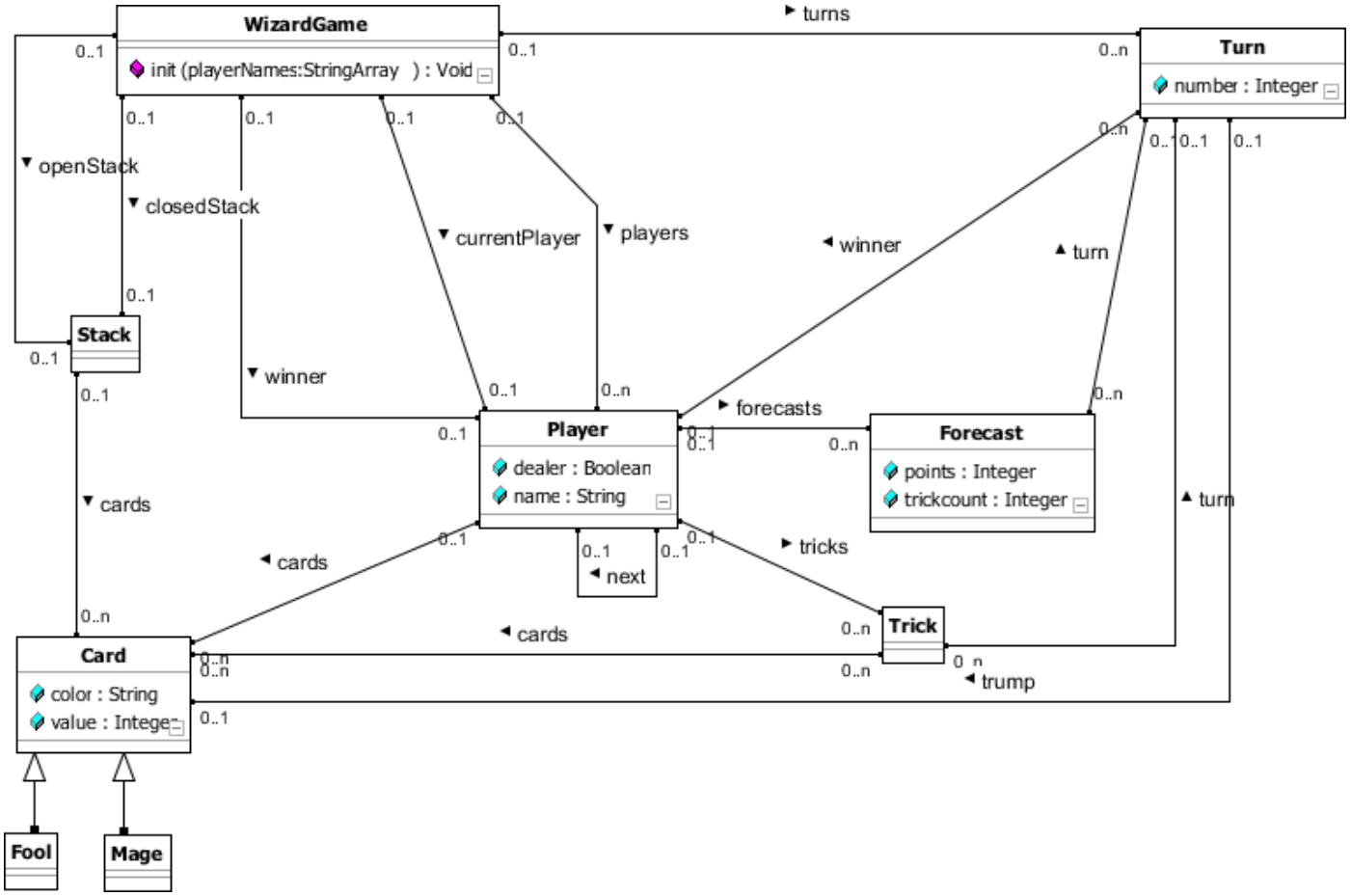
Besprechung HA 1 – Aufgabe 3 II

- **Aufgabe 3: Mögliches Wizard Objektdiagramm – Endszenario**



Besprechung HA 1 – Aufgabe 4

- **Aufgabe 3: Dazugehöriges Klassendiagramm**



JUnit – Motivation

- **Testen ist Ausprobieren?**
- **Unit Tests prüfen systematisch, ob das Programm das tut was es soll**
- **Vorteile:**
  - Reproduzierbar
  - Automatisierbar und selbst auswertend
  - Dokumentierend und Anwendungsbeispiel

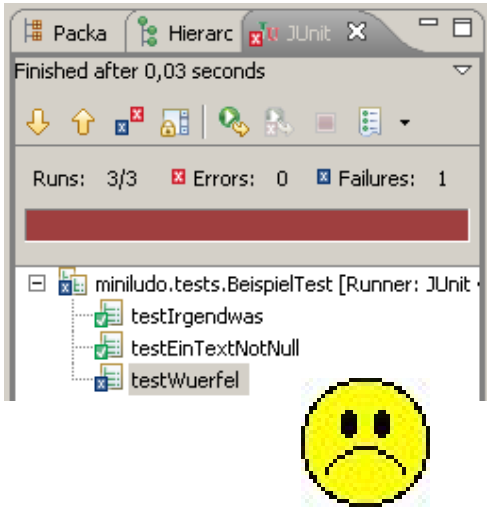
- **=> Weniger Fehler, Debugging und Fehlersuche**

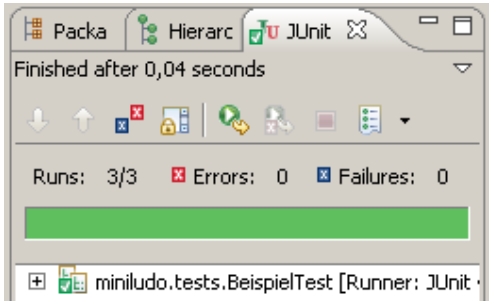
JUnit - Vorgehen beim Testen

- **Test vor der Implementierung:**
  - Funktionalität abprüfen
  - Alle möglichen Fehlerfälle prüfen
  - aber keine Trivialitäten
- **So wenig wie möglich implementieren**
  - Genau soviel, dass der Test erfolgreich ist
- **Umfang vom Testcode:**
  - 15-50% Anteil am Gesamtcode

JUnit - Framework (1)

- **Freies Open Source Framework**
  - http://www.junit.org/
  - aktuell: 4.8.2
- **Autoren: Kent Beck & Erich Gamma**
- **Grundeinstellung:**
  - Das Feature funktioniert erst, wenn ein Test geschrieben wurde!





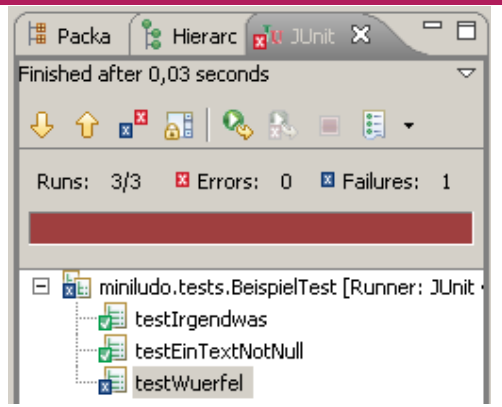
JUnit - Framework (2)

- **Tests in separaten Test-Klassen**
- **Funktionsweise der Test-Methoden**
  - Methode benennen
  - Beispiel-Situation aufbauen
  - Eine Aktion durchführen
    - Ein oder mehrere Methodenaufrufe
  - Rückgabewert oder Veränderungen prüfen
    - Mit Hilfe von Assertion-Methoden
    - Prüfung von Fehlerbehandlung (Exceptions)

JUnit - Assertion-Methoden

- **Ausdrücke, die wahr sein müssen, sonst Abbruch der aktuellen Test-Methode**
- ***assertEquals(soll, ist)***
  - Object, primitive Typen wie boolean, int, long, ...
- ***assertTrue(boolean)***
- ***assert[Not]Null(Object), assert[Not]Same(obj1, obj2)***
- ***fail()* (in Zusammenhang mit Kontrollfluss)**
- **Bei Fehlschlag oder *fail()*:**
  - Wirft *junit.framework.AssertionFailedError*
- **Alle Methoden auch mit ‚*String message*' Parameter**
  - Beispiel: `assertEquals(„Würfel zeigt 6", 6, wuerfel.getWert())`

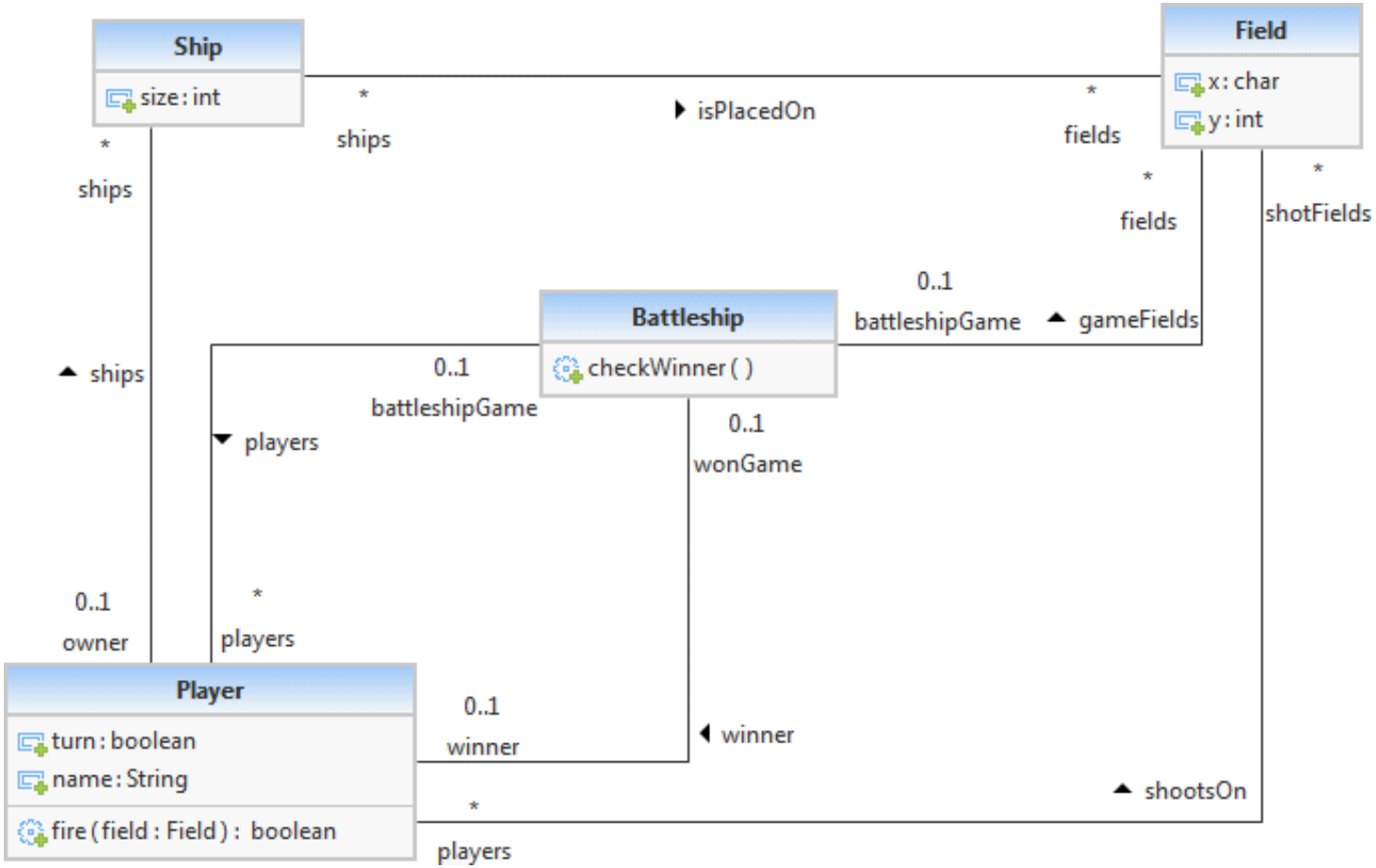
Testen - Vorgehen bei Rot



- **Funktionalität fertig implementiert?**
- **Ansonsten:**
  - Fehlermeldung!
  - Stacktrace
- **Debugging - Einkreisen eines Fehlers**
  - Breakpoints an „spannenden“ Stellen setzen
  - Variablenbelegung / Objektstruktur überprüfen
  - Methode schrittweise ausführen
  - in interessante Methoden reinsteppen
- **Tipp:**
  - Fehler gefunden => reproduzierenden Test schreiben

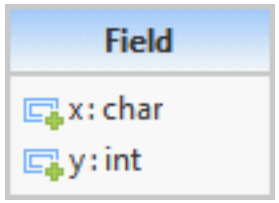
Implementierung Klassendiagramm I

- **Vorgehen bei Implementierung eines Klassendiagramms**



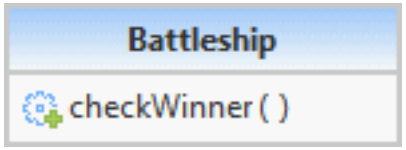
Implementierung Klassendiagramm II

- **Klassen erstellen**



Field.java

```
public class Field
{

}
```

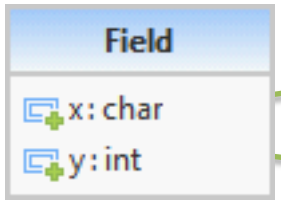


Battleship.java

```
public class Battleship
{

}
```

Implementierung Klassendiagramm III

- **Attribute implementieren**

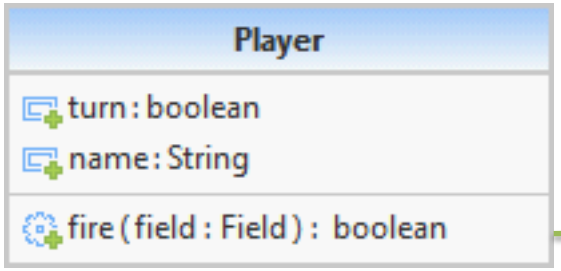



```
public class Field
{
    private char x;

    public char getX()
    {
        return this.x;
    }

    public void setX(char x)
    {
        this.x = x;
    }

    private int y;

    public int getY()
    {
        return this.y;
    }

    public void setY(int y)
    {
        this.y = y;
    }
}
```

Implementierung Klassendiagramm IV

- **Methoden implementieren**

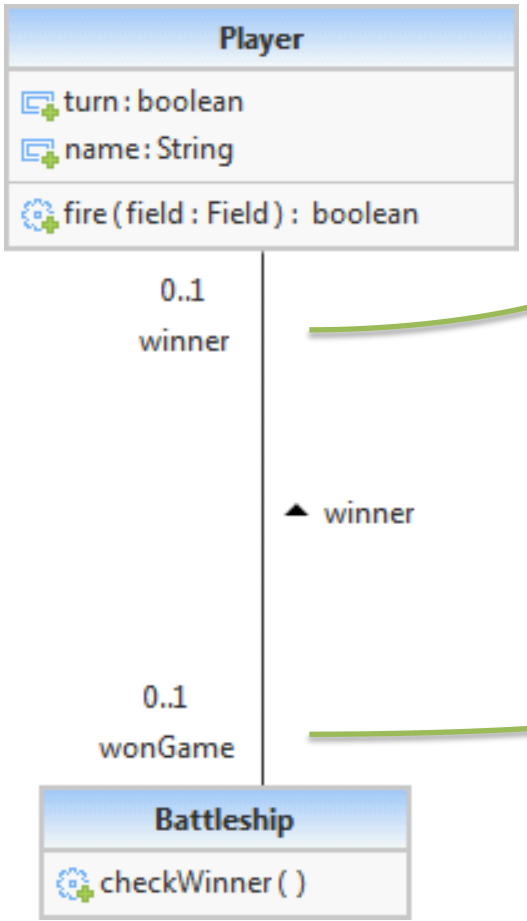

```
public class Battleship
{
    public void checkWinner()
    {
        …
    }
}
```




```
public class Player
{
    public boolean fire(Field field)
    {
        …
    }
}
```

Implementierung Klassendiagramm VII

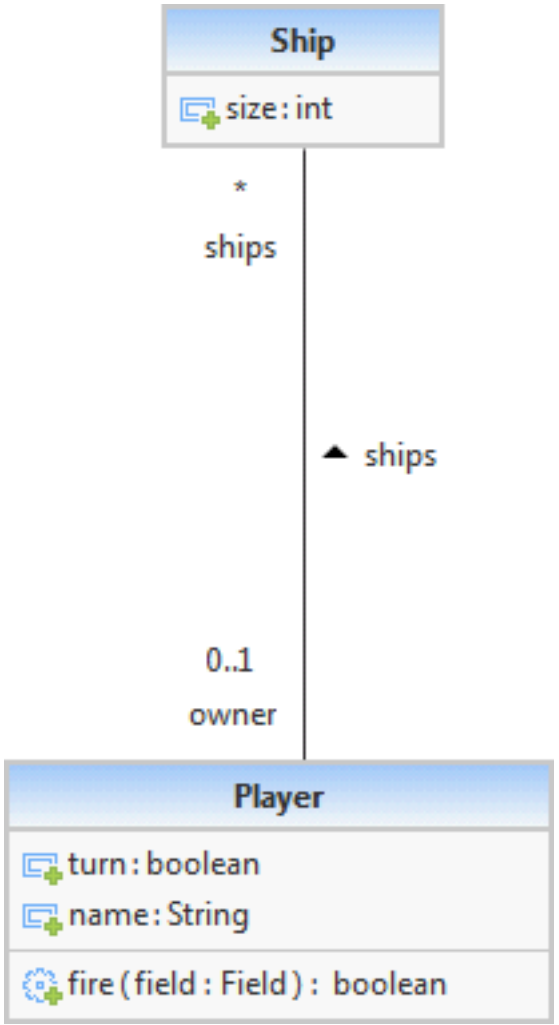
- **Assoziationen – Einfacher Fall: 1-zu-1**




```
public class Battleship
{
    private Player winner;

    public Player getWinner()
    {
        return this.winner;
    }

    public void setWinner(Player winner)
    {
        this.winner= winner;
    }
}
public class Player
{
    private Battleship wonGame;

    public Battleship getWonGame()
    {
        return this.wonGame;
    }

    public void setWonGame(Battleship wonGame)
    {
        this. wonGame = wonGame;
    }
}
```

Implementierung Klassendiagramm V

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



Implementierung VI: Collections

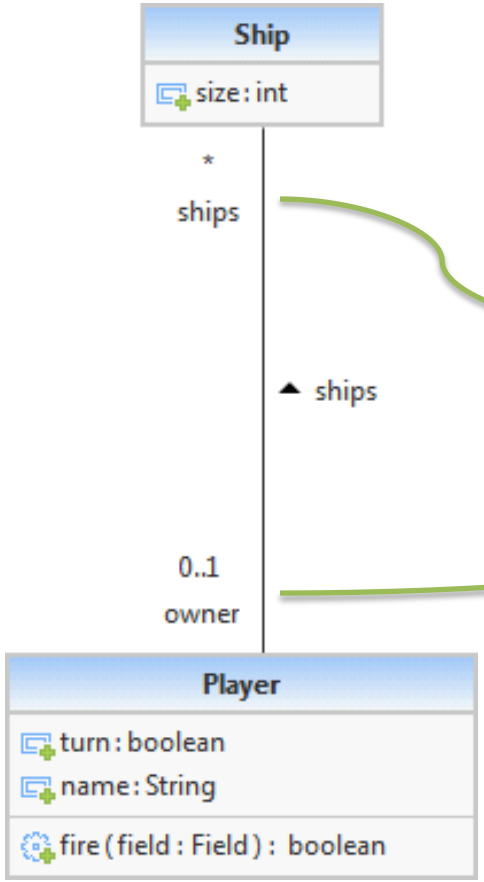
- **import java.util.***
- **ArrayList** (unsynchronized) **oder Vector** (synchronized)
  - Implementiert java.util.List (und somit java.util.Collection)
  - Basiert auf Arrays, aber vergrößert/verkleinert sich automatisch

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

  → …**Keine Duplikate**

Implementierung Klassendiagramm VIII

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




```
public class Player
{
   …
   private Set<Ship> ships =
                        new HashSet<Ship>();

   public void setShips(Set<Ship> ships)
   {
      this.ships = ships;
   }

   public Set<Ship> getShips()
   {
      return this.ships;
    }
}
```

```
public class Ship
{
   …
   private Player player;

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

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

Implementierung Klassendiagramm IX

```
public class Player
{
   …
 private Set<Ship> ships =
                    new HashSet<Ship>();

   public void setShips(Set<Ship> ships)
   {
      this.ships = ships;
   }

   public Set<Ship> getShips()
   {
      return this.ships;
    }
}
```

```
public class Ship
{
   …
 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();
    Ship s1 = new Ship();
    p1.getShips().add(s1);

     Assert.assertNotNull(s1.getOwner());
}
```

Implementierung Klassendiagramm X

- **Mögliche Lösung:**

```
public static void main(String[] args)
{
    Player p1 = new Player();
    Ship s1 = new Ship();
    p1.getShips().add(s1);

    s1.setOwner(p1);

    Assert.assertNotNull(s1.getPlayer());
}
```

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

Implementierung Klassendiagramm XI

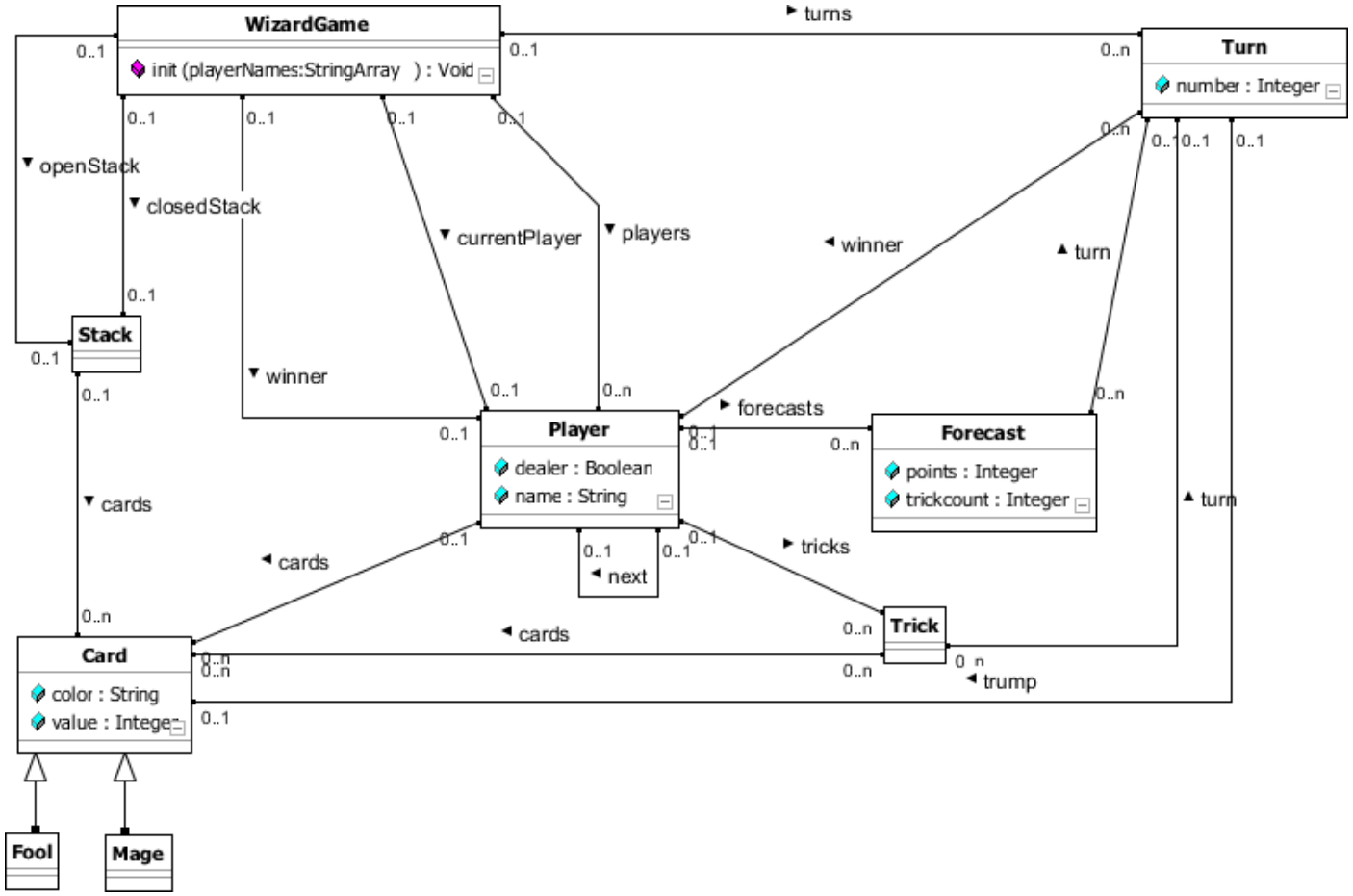
- **Besser: Methoden zum Hinzufügen/Setzen und Entfernen anbieten und Rückrichtung automatisch setzen.**

```
public class Player
{
    private Set<Ship> ships = new HashSet<Ship>();

    public void setShips(Set<Ship> ships)
    {
        this.ships = ships;
    }

    public Set<Ship> getShips()
    {
        return this.ships;
    }

    public void addShip(Ship ship)
    {
        if(getShips().add(ship))
        {
            ship.setPlayer(this);
        }
    }

    public void removeShip(Ship ship)
    {
        if(getShips().remove(ship))
        {
            ship.setPlayer(null);
        }
    }
    …
}
```

```
public class Ship
{
    private Player player;

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

            this.player = player;

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

    …
}
```

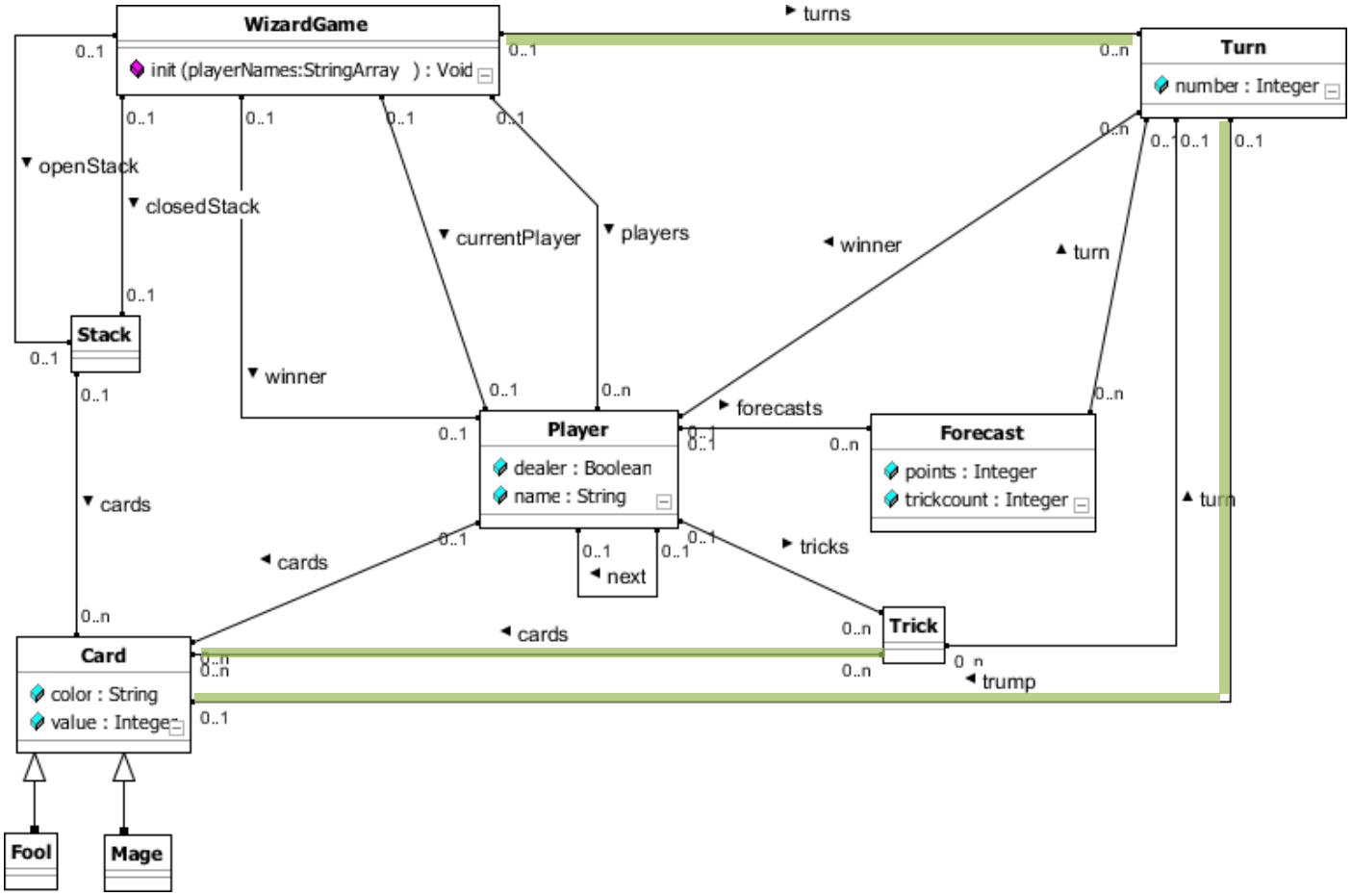
Vorstellung HA 2 I

- **Aufgabe 1: Implementierung des Klassendiagramms**



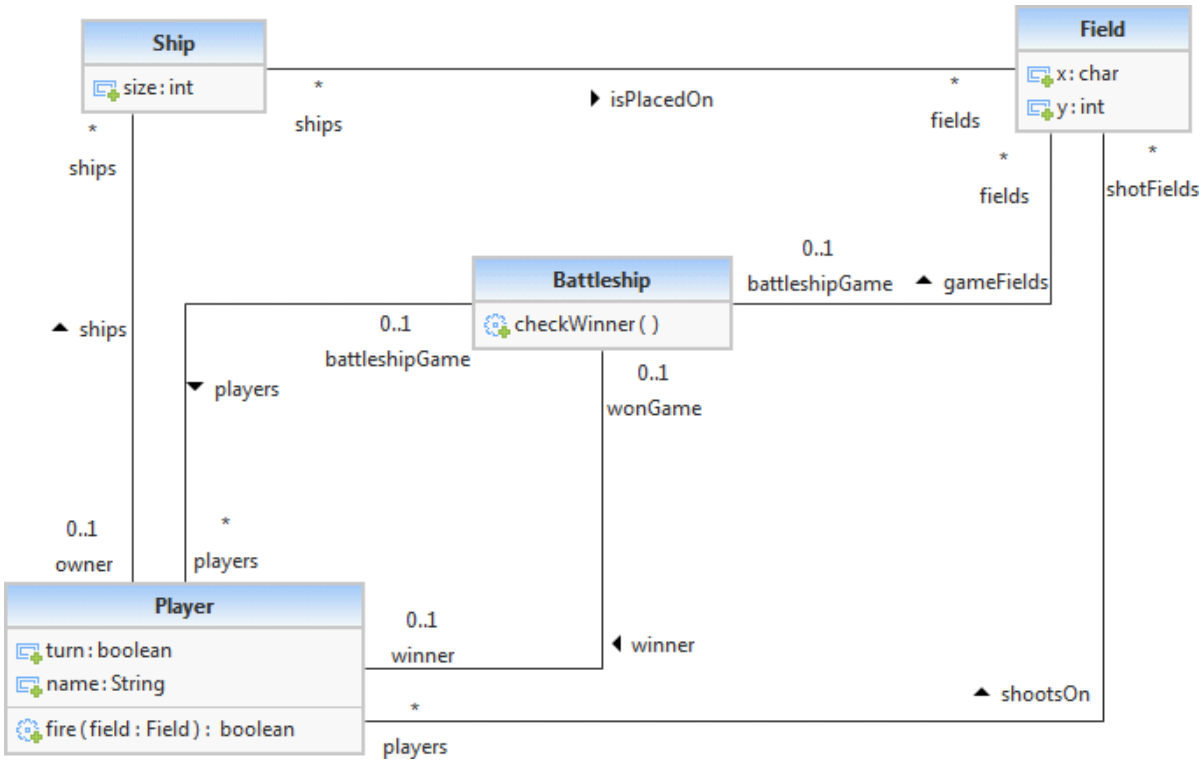
Vorstellung HA 2 II

- **Zusatzaufgabe: JUnit Tests zu referentieller Integrität**



Übung: Schiffe versenken Klassendiagramm implementieren

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



Ende

Schönes WE!