

Programmiermethodik

Übung 3

Wintersemester 13 / 14
Fachgebiet Software Engineering

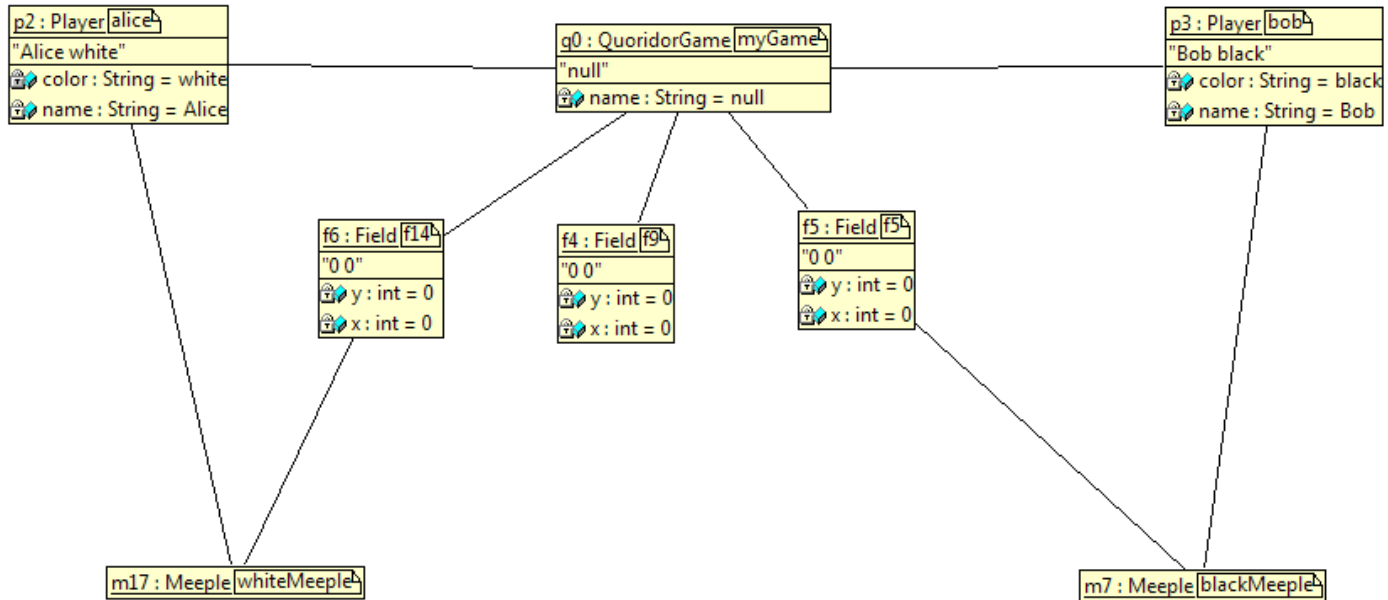
Stefan Lindel – Lennert Raesch
pm@cs.uni-kassel.de

Agenda

- **Besprechung HA2**
- **JUnit4**
- **Implementierung Klassendiagramm**
 - Klassen
 - Attribute
 - Methoden
 - Assoziationen
 - Vererbung
- **Vorschau HA3**
- **Praktische Übung**

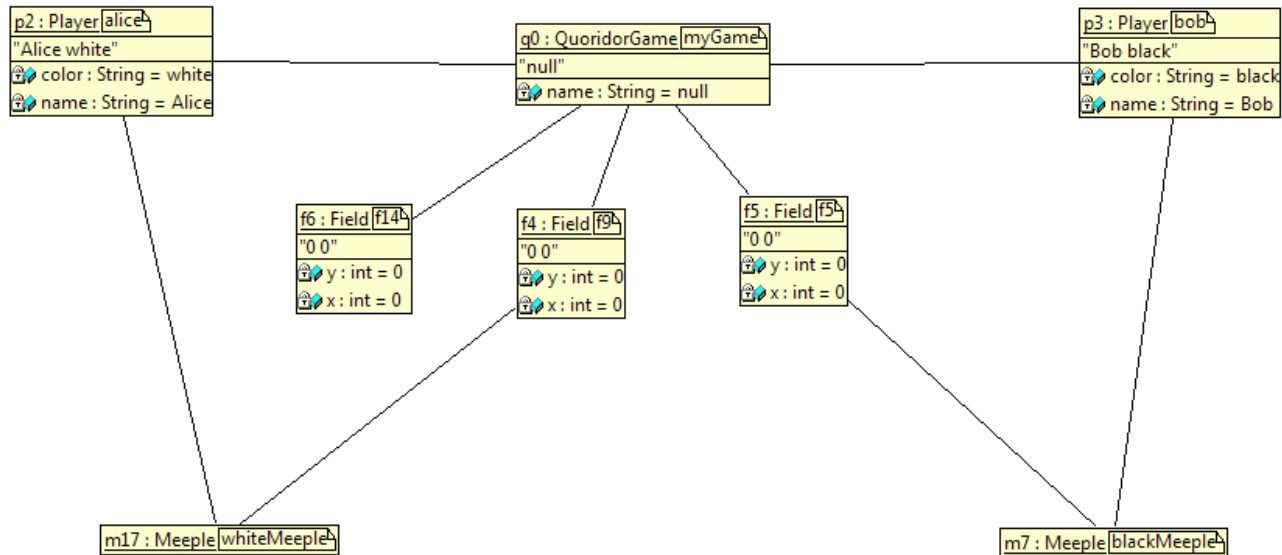
Besprechung HA 2 – Aufgabe 1

- Aufgabe 1: Mögliches Quoridor Objektdiagramm – Startsituation



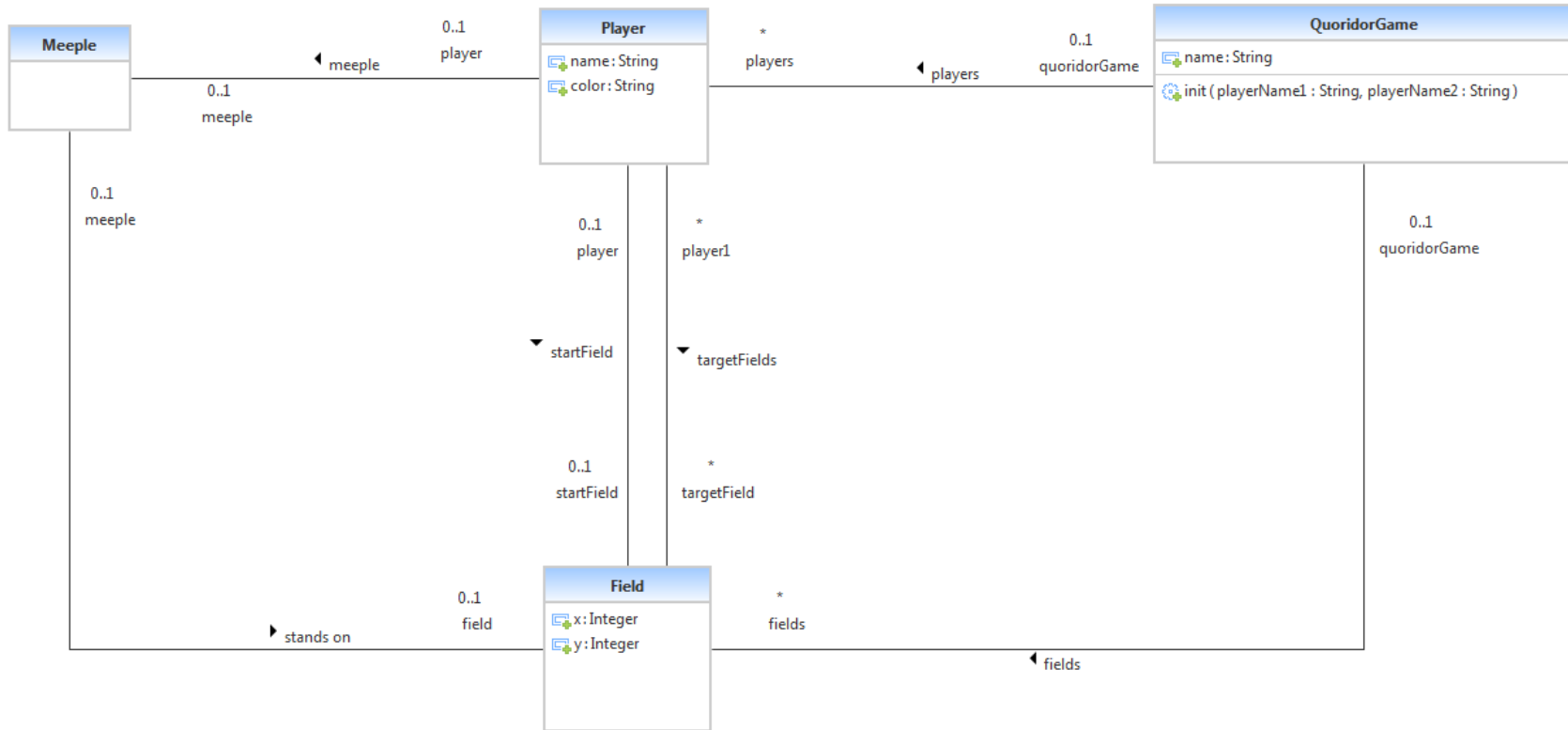
Besprechung HA 2 – Aufgabe 1

- Aufgabe 1: Mögliches Quoridor Objektdiagramm – Resultsituation



Besprechung HA 2 – Aufgabe 2

- Aufgabe 2: 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)

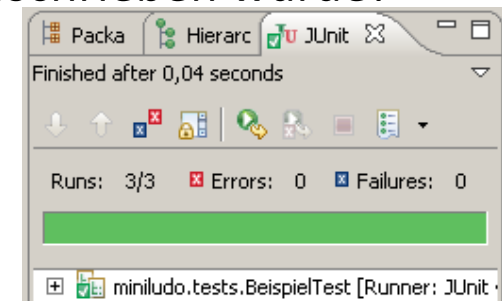
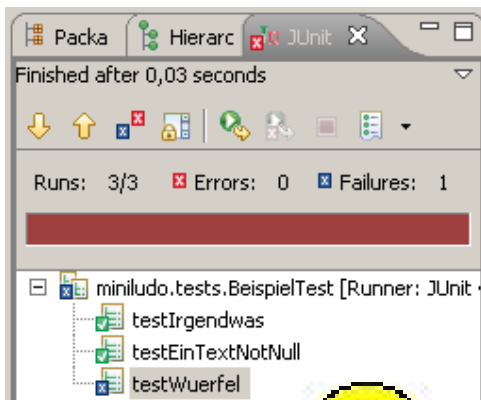
- **Freies Open Source Framework**

- <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)

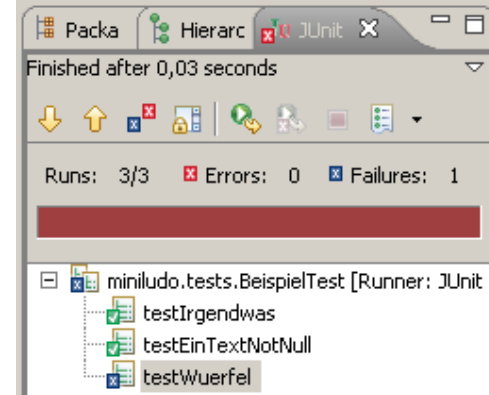
- **Tests in separaten Test-Klassen**
- **Funktionsweise der Test-Methoden**
 - Methode benennen
 - Beispiel-Situation aufbauen
 - Eventuell Ausgangssituation merken
 - Eine Aktion durchführen
 - Ein oder mehrere Methodenaufrufe
 - Rückgabewert oder Veränderungen prüfen
 - Mit Hilfe von Assertion-Methoden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

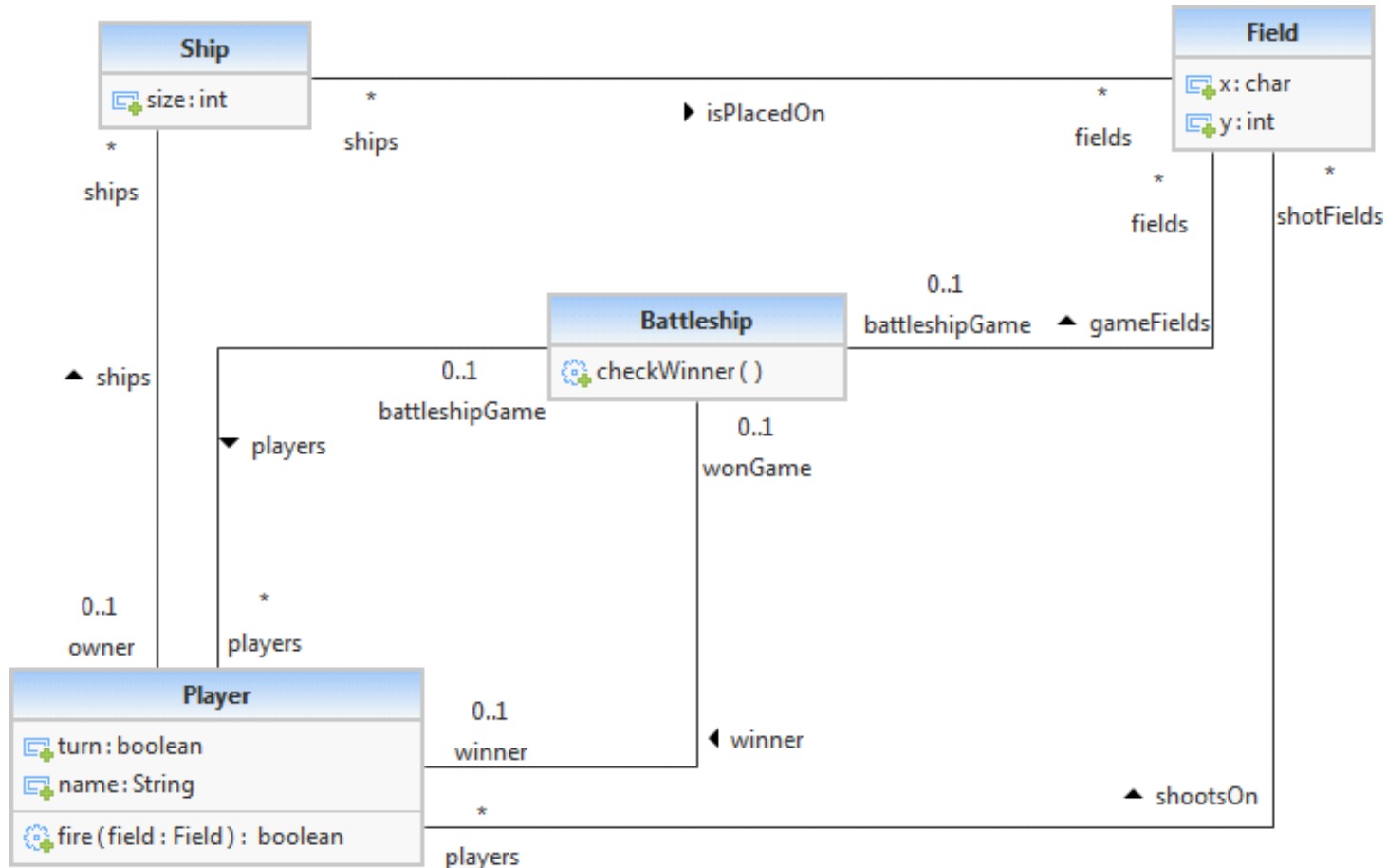
Testen - Vorgehen bei Rot

- **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



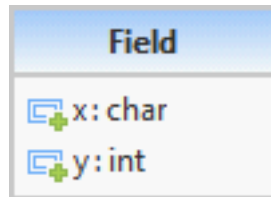
Implementierung Klassendiagramm I

- Vorgehen bei Implementierung eines Klassendiagramms



Implementierung Klassendiagramm II

- Klassen erstellen



Field.java



```

public class Field
{
}
    
```



Battleship.java

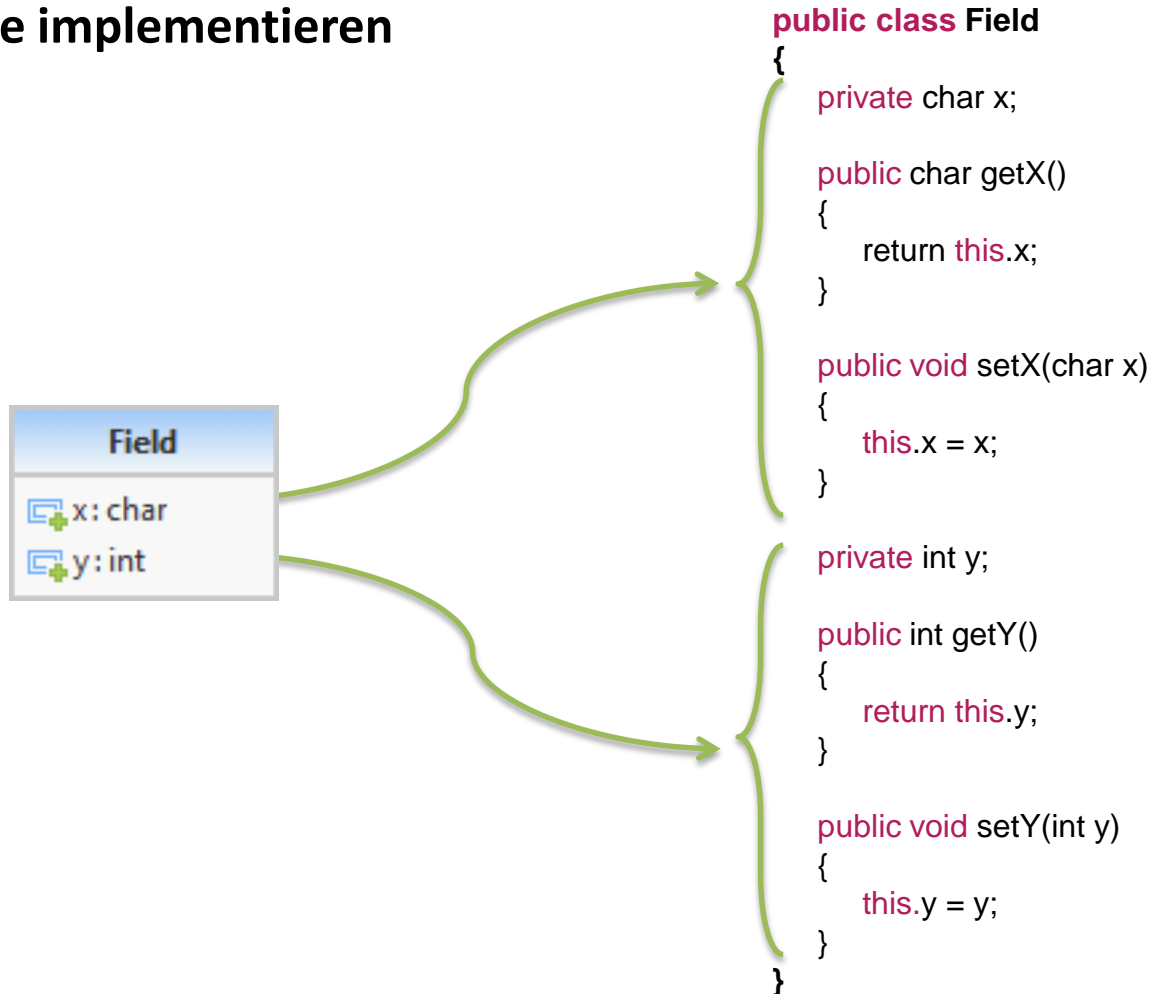


```

public class Battleship
{
}
    
```

Implementierung Klassendiagramm III

- Attribute implementieren

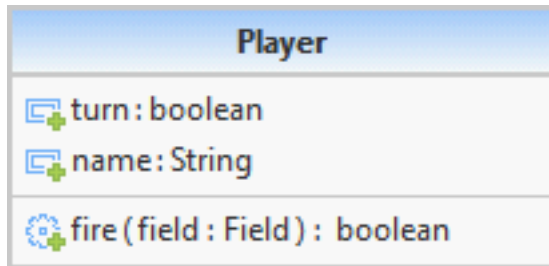


Implementierung Klassendiagramm IV

- Methoden implementieren



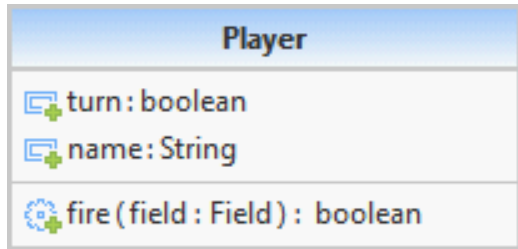
```
public class Battleship
{
    public void checkWinner()
    {
        ...
    }
}
```



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

Implementierung Klassendiagramm V

- Assoziationen – Einfacher Fall: 1-zu-1



0..1
winner

▲ winner

0..1
wonGame



```
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)
```

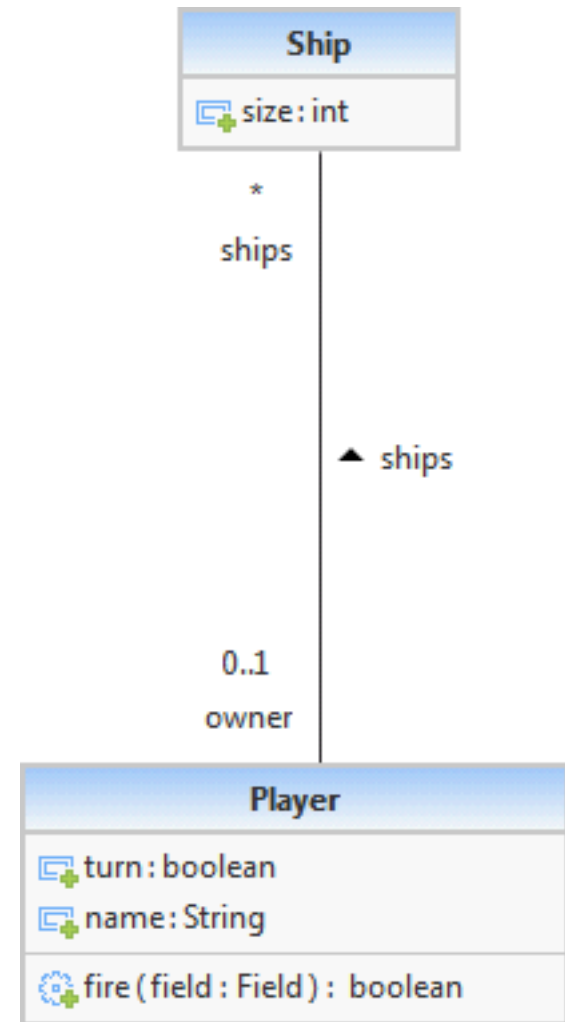
```
{
```

```
this.wonGame = wonGame;
```

```
}
```


Implementierung Klassendiagramm VI

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

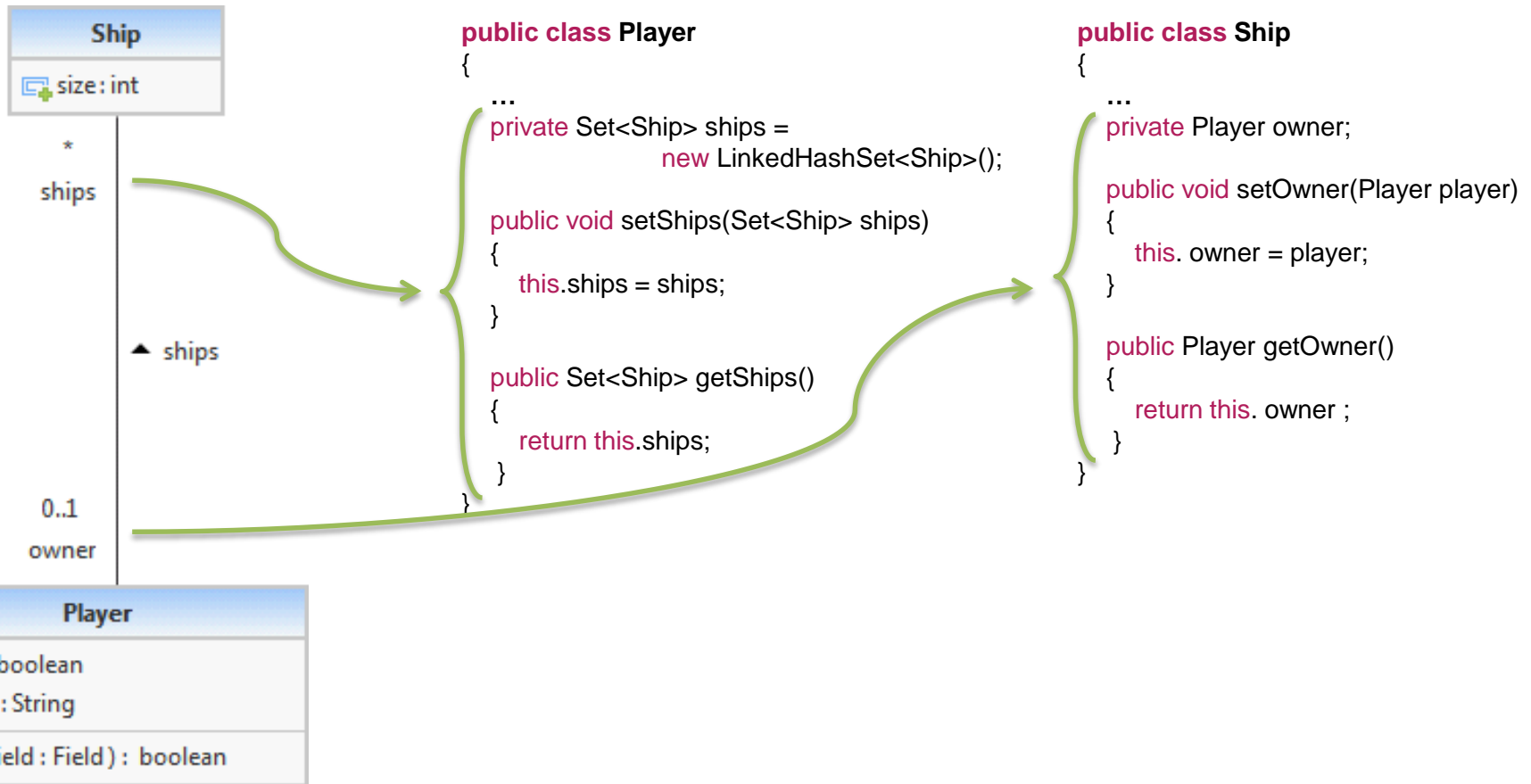


Implementierung VII: Collections

- **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 bzw. LinkedHashSet**
 - Implementiert java.util.Set (und somit java.util.Collection)
 - Menge mit hash-Zugriff
 - **...Keine Duplikate**

Implementierung Klassendiagramm VIII

- Schwieriger: zu-n. Einfache Implementierung



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 owner;

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

    public Player getOwner ()
    {
        return this.owner;
    }
}
```

- 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.getOwner());
}
```

- **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 LinkedHashSet<Ship>();

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

    public Set<Ship> getShips(){...}

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

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

```
public class Ship
{
    private Player owner;

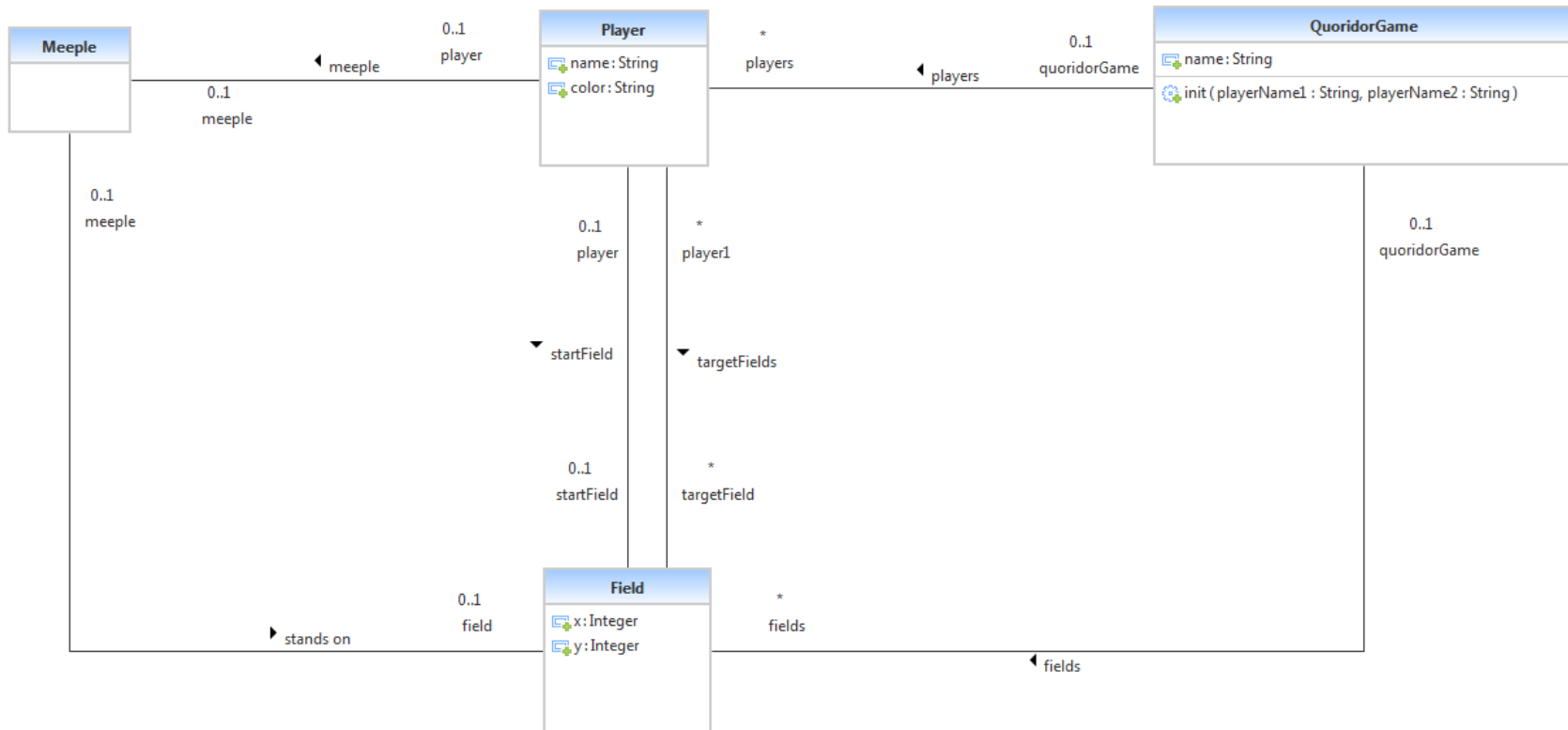
    public void setPlayer(Player player)
    {
        if(this.owner != player)
        {
            if(getOwner() != null)
            {
                getOwner().removeShip(this);
            }

            this.owner = player;

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

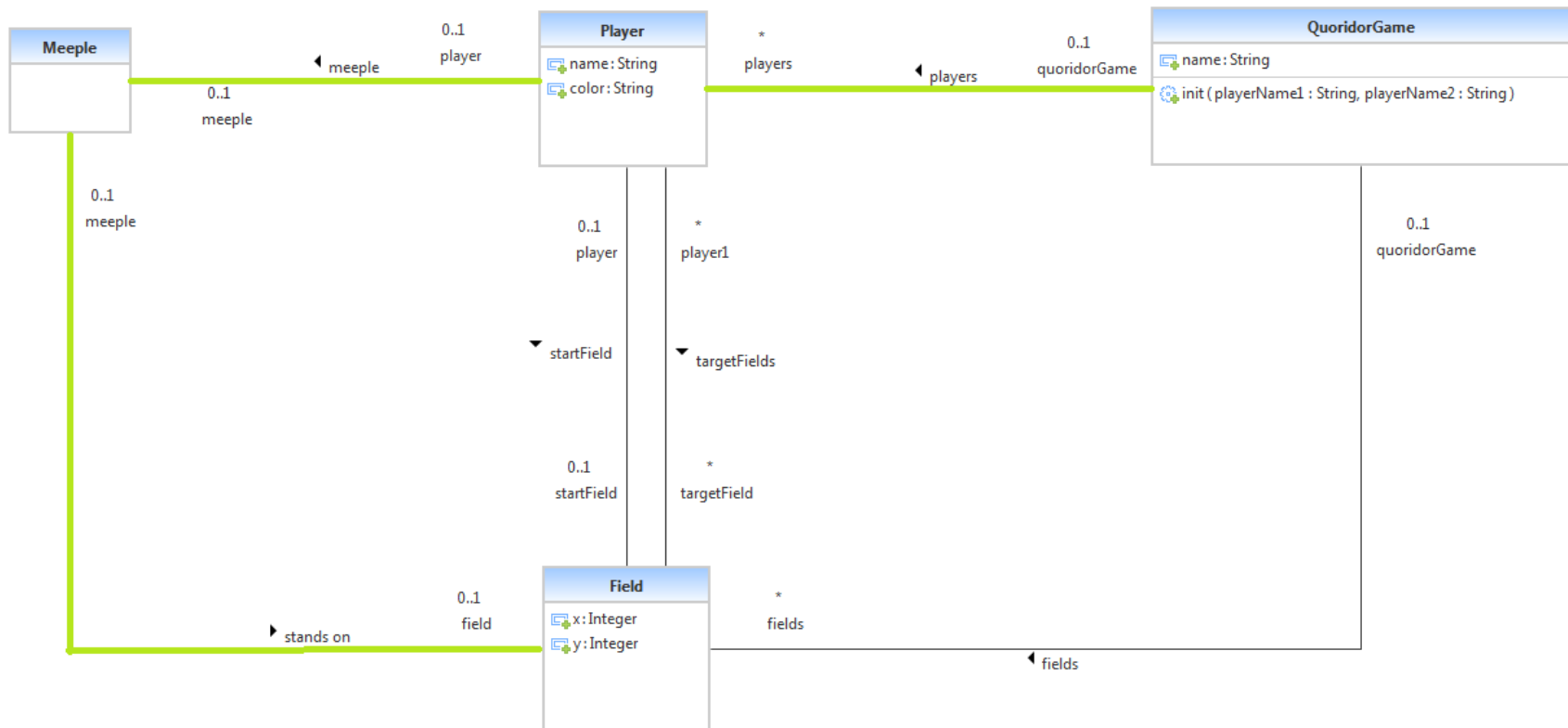
Vorschau HA 3 I

- **Deadline: 22.11.2011, 23:59 Uhr**
- **Aufgabe 1: Implementierung des Klassendiagramms (17P)**



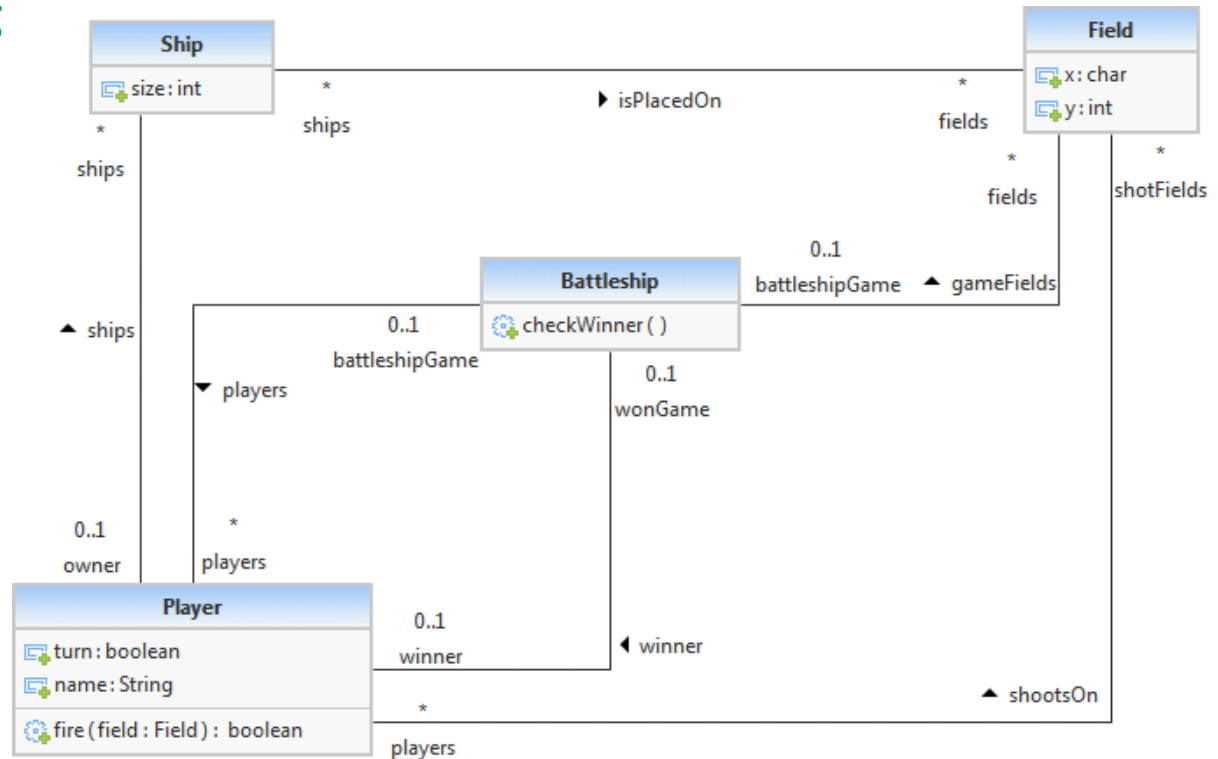
Vorschau HA 3 II

- Zusatzaufgabe: JUnit Tests zu referentieller Integrität (6 Zusatzpunkte)



Praktische Übung: Klassendiagramm implementieren

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



Ende

Jetzt: Betreutes Arbeiten

Ansonsten: Schönes WE!