Software-Praktikum

SS 2021, Blatt 4: Design Patterns (Teil 2)

Ausgabe: 02.09.2021

Prof. Dr. Sven Apel

Aufgabe 1 Factory Pattern

Nachdem es für Johannes mit der Informatik nicht geklappt hat, hat er sich entschlossen nach Amerika auszuwandern und eine Pizzeria zu eröffnen. Die erste öffnet er in New York City, kurz darauf kommt auch schon eine in Chicago dazu. Da Johannes ja mal Informatik studiert hat, hat er sich seine eigene Software zum online bestellen gebaut. Jede Pizzeria hat die gleiche Karte, allerdings unterscheiden sich die Zutaten der Pizzen regional.

```
abstract class Pizza {

    Dough dough
    Sauce sauce
    Cheese cheese
    List<Topping> toppings

    void bake() {
        print("Bake for 5 minutes")
    }
}
```

```
class CheesePizzaNY extends Pizza {

    CheesePizzaNY() {
        this.sauce = FancyTomatoSauce()
        this.dough = ThinPizzaDough()
        this.cheese = MozzarellaCheese()
        this.toppings = List<Topping>()
    }

}
```

```
class CheesePizzaChicago extends Pizza {

    CheesePizzaChicago() {
        this.sauce = SimpleTomatoSauce()
        this.dough = ThickPizzaDough()
        this.cheese = CheddarCheese()
        this.toppings = List<Topping>()
    }

}
```

```
class PeperoniPizzaNY extends Pizza {

    PeperoniPizzaNY() {
        this.sauce = FancyTomatoSauce()
        this.dough = ThinPizzaDough()
        this.cheese = MozzarellaCheese()
        this.toppings = List<Topping>()
        this.toppings.add(PeperoniTopping())
    }

}
```

```
class PeperoniPizzaChicago extends Pizza {

    PeperoniPizzaChicago() {
        this.sauce = SimpleTomatoSauce()
        this.dough = ThickPizzaDough()
        this.cheese = CheddarCheese()
        this.toppings = List<Topping>()
        this.toppings.add(PeperoniTopping())
    }

}
```

Da seine beiden Pizzerien so gut laufen, entscheidet sich Johannes einen weiteren Store in Miami zu eröffnen. Da er durch die drei Shops jedoch so eingespannt ist, bittet er Sie um Hilfe.

a) Fügen Sie neue Klassen für die beiden Pizzen, welche in Miami angeboten werden hinzu. Beide Pizzen haben als Tomatensauce eine `SpicyTomatoSauce()`, den gleichen Teig wie in New York und Käse wie in Chicago, die Salami ist `HotPeperoniTopping()`.

b) Natürlich ist Ihnen schon beim ersten Blick auf dieses Design aufgefallen, warum Johannes nicht weiter studiert hat. Nennen Sie Probleme mit seinem Design.

c) Nutzen Sie das *Abstract Factory Pattern* um diese Probleme auszumerzen.

Lösung

a)
```
class CheesePizzaMiami extends Pizza {

    CheesePizzaMiami() {
        this.sauce = SpicyTomatoSauce()
        this.dough = ThinPizzaDough()
        this.cheese = CheddarCheese()
        this.toppings = List<Topping>()
    }

}
```

```
class PeperoniPizzaMiami extends Pizza {

    PeperoniPizzaMiami() {
        this.sauce = SpicyTomatoSauce()
        this.dough = ThinPizzaDough()
        this.cheese = CheddarCheese()
        this.toppings = List<Topping>()
        this.toppings.add(HotPeperoniTopping())
    }

}
```

b) Ein Problem ist, dass man wenn man einen neuen Store aufmachen will, man jede Pizza für diesen Store neu schreiben muss. Außerdem muss man, wenn sich beispielsweise die Sauce in New York ändert jede Pizza im New York Store anpassen.

c) Wir nutzen Factories um die Pizzabeläge zu instanziieren. Diese unterscheiden sich pro Pizzeria. Die Nutzung der Factory hilft uns dabei alle Pizzen unverändert in beliebig vielen Shops zu nutzen, man muss lediglich eine neue Factory anlegen.

```
interface PizzaFactory {

    Sauce getSauce()
    Dough getDough()
    Cheese getCheese()
    PeperoniTopping getPerperoni()

}
```

```
class PizzaFactoryNY implements PizzaFactory {

```

```
    Sauce getSauce() {
        return FancyTomatoSauce()
    }

    Dough getDough() {
        return ThinPizzaDough()
    }

    Cheese getCheese() {
        return MozzarellaCheese()
    }

    PeperoniTopping getPerperoni() {
        return PeperoniTopping()
    }

}
```

```
class PizzaFactoryChicago implements PizzaFactory {

    Sauce getSauce() {
        return SimpleTomatoSauce()
    }

    Dough getDough() {
        return ThickPizzaDough()
    }

    Cheese getCheese() {
        return CheddarCheese()
    }

    PeperoniTopping getPerperoni() {
        return PeperoniTopping()
    }

}
```

```
class PizzaFactoryMiami implements PizzaFactory {

    Sauce getSauce() {
        return SpicyTomatoSauce()
    }

    Dough getDough() {
        return ThinPizzaDough()
    }

    Cheese getCheese() {
        return CheddarCheese()
    }

    PeperoniTopping getPerperoni() {
        return HotPeperoniTopping()
    }

}
```

```
class CheesePizza extends Pizza {

    CheesePizza(PizzaFactory factory) {
        this.sauce = factory.getSauce()
        this.dough = factory.getDough()
        this.cheese = factory.getCheese()
        this.toppings = List<Topping>()
```

```
        this.toppings.add(factory.getPeperoni())
    }

}
```

```
class PerperoniPizza extends Pizza {

    PeperoniPizza(PizzaFactory factory) {
        this.sauce = factory.getSauce()
        this.dough = factory.getDough()
        this.cheese = factory.getCheese()
        this.toppings = List<Topping>()
        this.toppings.add(factory.getPeperoni())
    }

}
```

Aufgabe 2 Command Pattern

Durch seine drei erfolgreichen Pizzerien, braucht Johannes immer mehr Mitarbeiter. Da er ja außerdem mal Informatik studiert hat, entscheidet er sich dazu einen Roboter für sein New Yorker Restaurant anzuschaffen, der künftig die Pizzen zubereiten soll. Dazu hat er auch schon etwas programmiert, was den Roboter steuern und die Pizzen nach Rezept zubereiten soll. Der Roboter kann dabei über ein Interface gesteuert werden, indem man seine Arme neu positioniert. Alles Andere muss von Johannes Programm gesteuert werden.

```
interface Pizza {

    void preparePizza(Robot robot)

}
```

```
class PeperoniPizza implements Pizza {

    void preparePizza(Robot robot) {
        // Move robot to prepare dough
        robot...
        // Move robot to add sauce
        robot...
        // Move robot to add cheese
        robot...
        // Move robot to add peperoni
        robot...
        // Move robot to put pizza in oven
        robot...
        // Let robot wait for pizza
        robot.wait()
        // Move robot to take the pizza out of the oven
        robot...
    }

}
```

```
class CheesePizza implements Pizza {

    void preparePizza(Robot robot) {
        // Move robot to prepare dough
        robot...
        // Move robot to add sauce
        robot...
        // Move robot to add cheese
        robot...
        // Move robot to put pizza in oven
        robot...
        // Let robot wait for pizza
```

```
13          robot.wait()
14          // Move robot to take the pizza out of the oven
15          robot...
16      }
17
18  }
```

a) Welches Problem tritt bei dem aktuellen Design auf, wenn man neue Pizzen hinzufügt?

b) Wie kann das Command Pattern helfen dieses Problem zu beseitigen?

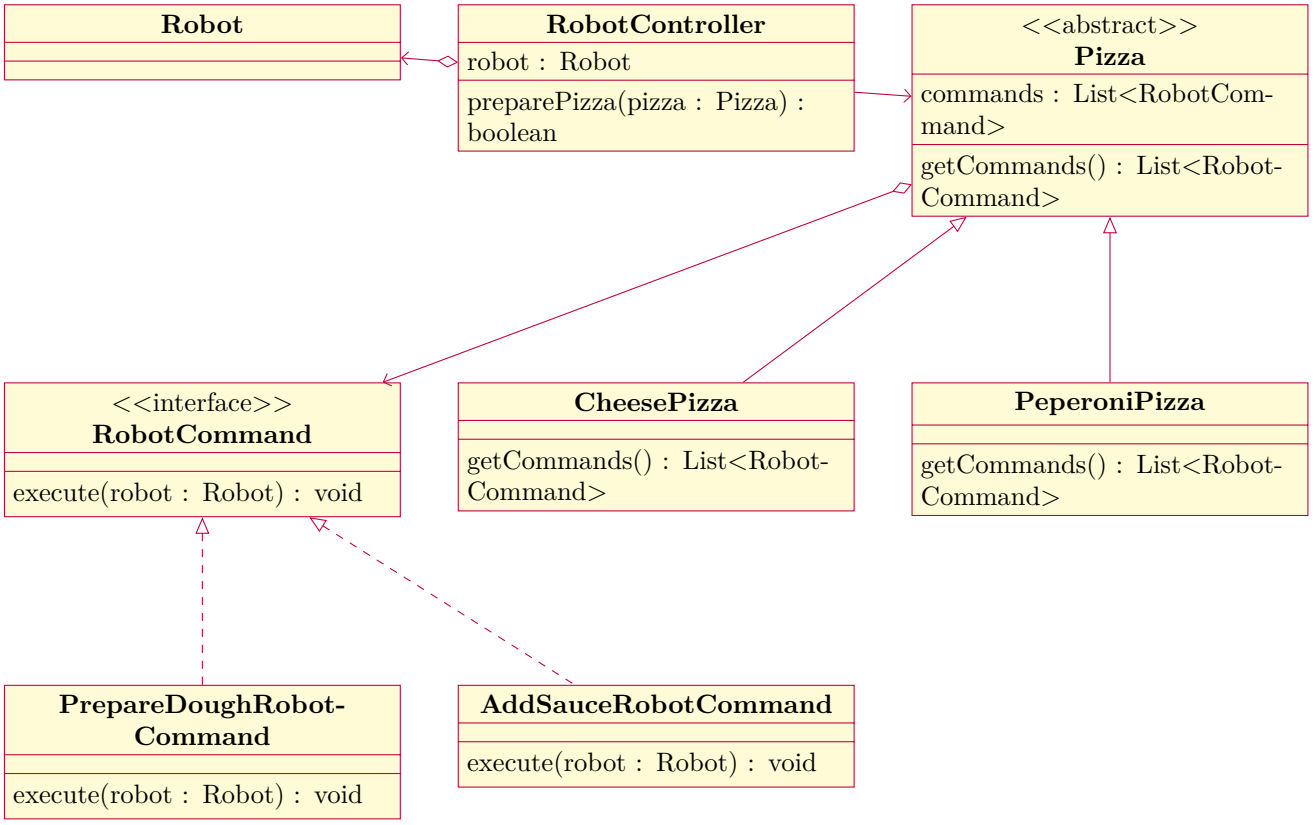
c) Redesignen Sie das Programm mit Hilfe des Command Patterns.

Lösung

a) Das größte Problem ist, dass man im aktuellen Design **alle** Bewegungen des Roboters für die neue Pizza erneut implementieren muss.

b) Das Command Pattern kann dafür genutzt werden bestimmte Aktionen des Roboters zu implementieren und so für alle zugänglich zu machen. Anstelle davon dass jede Pizza selbst weiß, wie Sie den Roboter bewegen muss, weiß sie nurnoch, welche Aktionen nötig sind um die Pizza zuzubereiten.

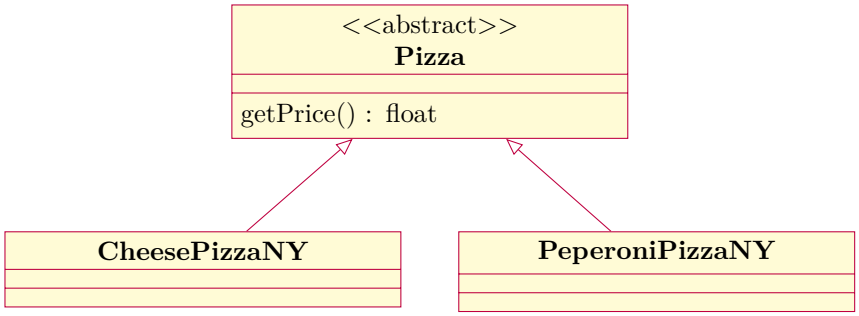
c) *Hinweis: Die Lösung ist nicht vollständig, wir verzichten der wegen der Übersichtlichkeit auf ein paar Commands.*



Aufgabe 3 Decorator Pattern

Nach anfänglichem Erfolg, hat Johannes sich dazu entschieden, in jedem Shop alle Pizzen anzubieten und hat seinen Store entsprechend angepasst. Außerdem will er noch andere Pizzen, als nur Cheese und Peperoni anbieten. Dazu hat er sich eine ganze Palette an Zutaten besorgt: Salami, Cheddar, Mozzarella, Schinken, Ananas, Champignons und Mais.

Das aktuelle Design:



a) Fügen Sie 4 weitere Pizza-Kombinationen Ihrer Wahl zum aktuellen Angebot hinzu.

b) Johannes ist zufrieden mit Ihrer Arbeit, will jetzt allerdings beliebige Kombinationen der aktuellen Zutaten anbieten. Sie machen ihn darauf aufmerksam, dass es keinen Sinn macht, das aktuelle Design beizubehalten. Warum?

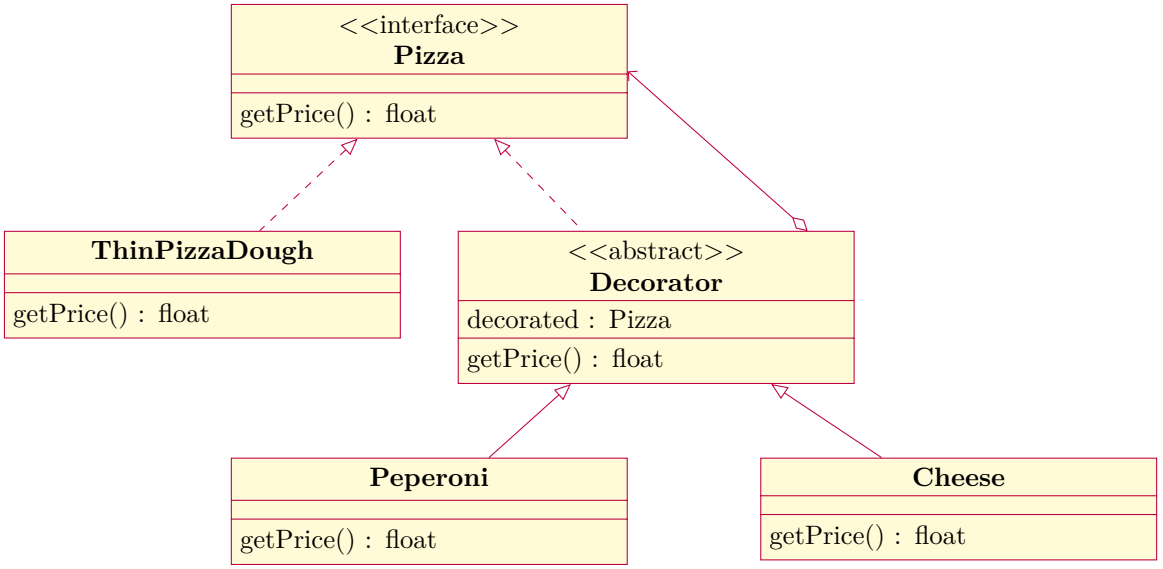
c) Nutzen Sie das Decorator Pattern, um die Flexibilität zu erhöhen und ähnliche Probleme in Zukunft zu vermeiden.

Lösung

a) *trivial*

b) Beim aktuellen Design müsste man um beliebig viele Kombinationen von den n Zutaten zu ermöglichen, 2^n viele Klassen anlegen. Außerdem kann man dann auch noch keine Zutaten doppelt wählen.

c) *Hinweis: Die Lösung ist nicht vollständig, wir verzichten der wegen der Übersichtlichkeit auf ein paar Zutaten.*



Aufgabe 4 State Machine Pattern

Johannes hat sich überlegt einen Online-Store zum Bestellen in seiner Pizzeria anzubieten. Er will Kunden dabei die Möglichkeit bieten mehrere Pizzen gleichzeitig zu bestellen, die Bestellung zu bezahlen, die Bestellung abzubrechen und den Status der Bestellung zu erfragen. Abhängig vom Fortschritt des Bestellvorgangs sind allerdings nur bestimmte Aktionen möglich:

- Zu Beginn gibt man seine Adresse an, um den Bestellvorgang zu starten.
- Danach kann man eine Pizza adden oder den Bestellvorgang abbrechen.
- Anschließend kann man beliebig viele weitere Pizzen hinzufügen, den Bestellvorgang abbrechen oder bezahlen.
- Nach dem Zahlen kann man die Bestellung abbrechen oder den Status ansehen. Gleichzeitig ist es ab jetzt für den Store möglich, die Bestellung zu stornieren oder mit der Zubereitung zu beginnen.
- Nachdem mit der Zubereitung begonnen wurde, kann der Kunde nur noch den Status der Bestellung checken, der Store nur noch den Status zu in Lieferung ändern.
- Während der Auslieferung kann man weiterhin den Status checken. Nachdem die Pizzen geliefert wurden, kann man dann einen neuen Bestellvorgang beginnen.

a) Modellieren Sie das oben angegebene Verhalten als State-Machine.

b) Johannes hat schon einen Prototypen gebaut:

```
class Order {

    Address address
    List<Pizza> pizzas
    boolean payed = false
    boolean inPreparation = false
    boolean shipped = false

    boolean addAddress(Address address) {
        if (this.address == null) {
            this.address = address
            return true
        }
        return false
    }

    boolean addPizza(Pizza pizza) {
        if (this.address == null) {
            return false
        }
        if (this.pizzas == null) {
            this.pizzas = List<Pizza>()
        }
        this.pizzas.add(pizza)
        return true
    }

    boolean pay() {
        if (this.address != null && this.pizzas != null) {
            // Pay
            payed = true
            return true
        }
        return false
    }

    boolean cancel() {
        if (this.inPreparation || this.shipped) {
            return false
        }
        this.pizzas = null
        this.address = null
    }

    boolean startPreparation(int uid) {
        // uid 1 means shop
        if (uid == 1 && this.pizzas != null && this.address != null && !this.inPreparation
        ↪  && !this.shipped) {
            this.inPreparation = true
            return true
        } else {
            return false
        }
    }

    boolean startShipping(int uid) {
        // uid 1 means shop
        if (uid == 1 && this.inPreparation) {
            this.inPreparation = false
            this.shipped = true
            return true
        } else {
            return false
        }
```

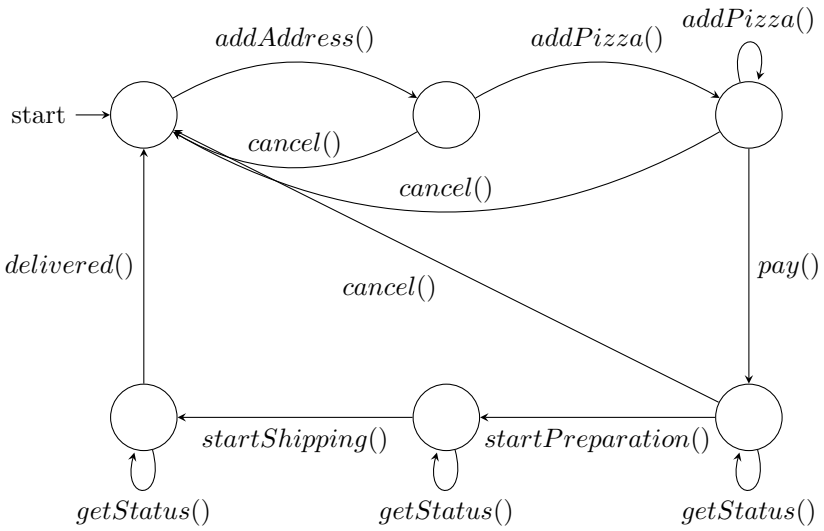
```
64      }
65
66      boolean delivered(int uid) {
67          // uid 1 means shop
68          if (uid == 1 && this.shipped) {
69              this.inPreparation = false
70              this.shipped = false
71              this.address = null
72              this.pizzas = null
73              return true
74          } else {
75              return false
76          }
77      }
78
79      String getStatus() {
80          if (this.address == null && this.pizzas == null) {
81              throw IllegalStateException()
82          }
83          if (payed && !inPreparation && !shipped) {
84              return "Bezahlt"
85          }
86          if (inPreparation) {
87              return "Wird zubereitet"
88          }
89          if  (shipped) {
90              return "Wird geliefert"
91          }
92      }
93  }
```

Wie zu erwarten, ist auch dieses Design nicht sonderlich gut. Was sind Probleme?

c) Nutzen Sie das State Machine Pattern, um die Probleme zu beheben.

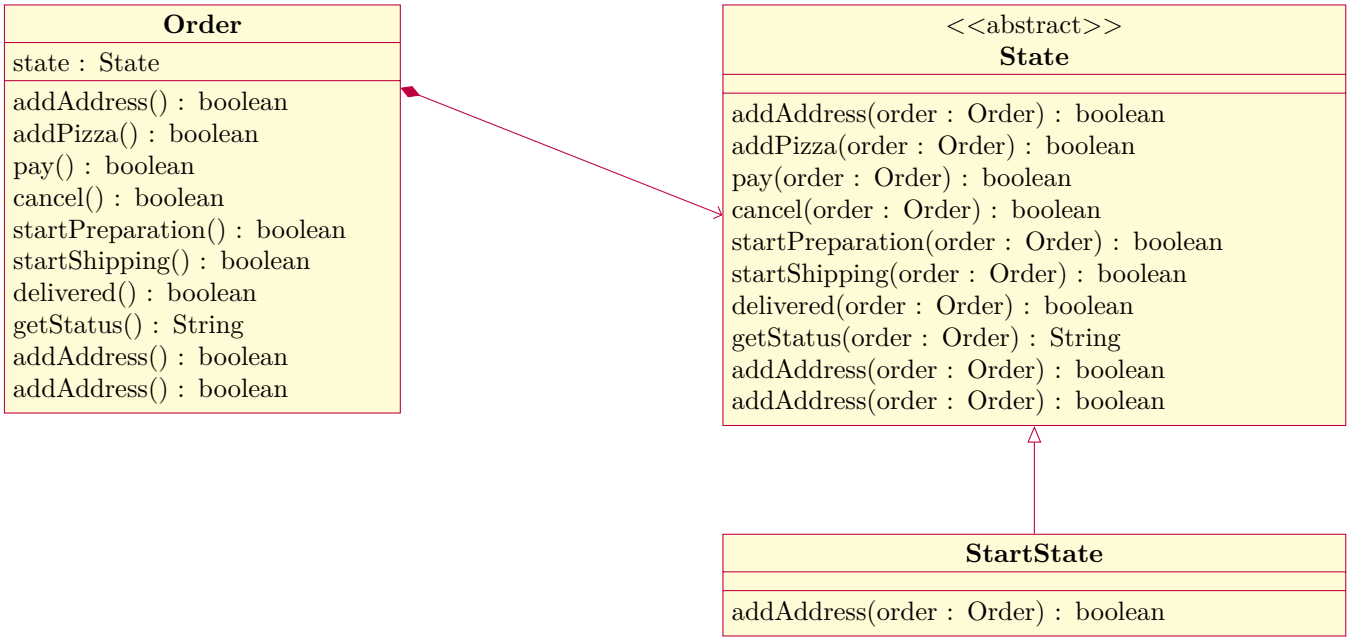
Lösung

a) State Machine:



b) Das größte Problem hier ist, dass wir nicht wissen in welchem Zustand wir uns aktuell befinden. Das bedeutet, dass wir bei jeder Funktion anhand von Konditionalen erstmal herausfinden müssen, welches der aktuelle Zusatand ist. Wenn wir jetzt einen neuen Zustand einführen, müssen wir die Konditionale aller anderen Zustände ebenfalls anpassen. Weiterhin ist das Verhalten der Zustände nicht abgetrennt voneinander, sondern alles zusammen in einer Klasse. Das erschwert die Lokalisierung von Fehlern und somit das Debuggen.

c) *Hinweis: Die Lösung ist nicht vollständig, wegen der Übersichtlichkeit geben wir nur einen State an.*



Aufgabe 5 Fun with Flags

Karibik 1713, das goldene Zeitalter der Piraterie. Die See ist voll mit Piraten. Ein Problem für Händler, welche ihre Schiffe mit immer mehr Kanonen und Schießpulver ausstatten müssen, um nicht überfallen zu werden. Gleichzeitig sind die Meere voll von Schätzen und anderem Treibgut, welches man einsammeln kann, um den Profit zu vergrößern. Es ist allerdings immer ein Risiko.

Ihre Aufgabe ist es, dieses Szenario mit Hilfe der gelernten Patterns und Prinzipien als Spielserver zu designen.

- Der Spieler steuert ein Schiff durch die Weiten der Karibik. Das Schiff hat 5 große Speicher und 5 Leben. Ziel ist es, Havanna zu erreichen.
- Wie schon beschrieben befinden sich auch einige Piraten in den Gewässern. Diese sind eine Gefahr für den Spieler und greifen ihn, wenn sie max. 3 Felder entfernt sind, an. Sie haben 1 Leben.
- Eine Attacke verringert das Leben des Getroffenen um 1.
- Man kann nicht mehr Speicher mit Ladung gefüllt haben, als man Leben hat. Sollte man zu viel Ladung an Bord, haben verliert man automatisch so viel Ladung, wie zu viel ist.
- Man kann sich in Häfen gegen Geld heilen und Kanonenkugeln kaufen. Man kann maximal 5 Kanonenkugeln (unabhängig vom Leben) geladen haben.
- Auf dem Meer schwimmen Schätze, Ladung und Kanonenkugeln. Schätze können immer aufgenommen werden und sie erhöhen den Kontostand des Spielers. Kanonenkugeln und Ladung kann nur dann aufgenommen werden, wenn man noch genug Platz an Bord hat.
- Das Spiel ist rundenbasiert. Ein Spieler hat so viele Aktionen, wie er -zu Beginn der Runde- Leben hat. Piraten haben 2 Aktionen.
- Die Karte besteht aus Hexagonen. Auf einem Hexagon kann immer nur ein Schiff und ein Treibgut sein.
- Es gibt Felsen, die nie passierbar sind, Meer, Häfen, und Havanna. Ein Hexagon hat genau einen dieser Typen.
- Ein Schiff kann verschiedene Aktionen durchführen, jede davon dauert eine Aktion. Die verschiedenen Die Aktionen sind:
  - `move(Direction)`, bewegt das Schiff in die angegebene Richtung.
  - `repair()`, repariert das Schiff.
  - `buy()`, kauft so viele Kanonenkugeln, wie der Spieler Platz/Geld hat.
  - `fire(Direction)`, feuert die Kanone in die angegebene Richtung ab.
  - `pickup()`, hebt das Treibgut der aktuellen Kachel auf.
- Der Server empfängt die Befehle in seiner Connection.

```
1 interface Connector<T> {
```

```
2
3       // Returned den nächsten Befehl des Spielers. Wie dieser aussieht, können Sie selbst
        ↪  entscheiden.
4       public T getOrder()
5   }
```

Hinweis: Der Connector gibt Ihnen ein Objekt mit gewünschtem Typ zurück. Weiterhin verzichten wir -der Einfachheit wegen- auf die Kommunikation von Server zu Client.

Lösung

Zu dieser Aufgabe gibt es keinen Lösungsvorschlag. Mögliche Lösungen können allerdings im Forum diskutiert werden. Die Minitests werden keine Aufgabe mit einem solchen Umfang beinhalten.