Programmentwurf
Schiffe Versenken

Name: Falk, Peter
Martrikelnummer: 8014538

Abgabedatum: 28.05.23

Allgemeine Anmerkungen:

- *es darf nicht auf andere Kapitel als Leistungsnachweis verwiesen werden (z.B. in der Form "XY wurde schon in Kapitel 2 behandelt, daher hier keine Ausführung")*
- *alles muss in UTF-8 codiert sein (Text und Code)*
- *sollten mündliche Aussagen den schriftlichen Aufgaben widersprechen, gelten die schriftlichen Aufgaben (ggf. an Anpassung der schriftlichen Aufgaben erinnern!)*
- *alles muss ins Repository (Code, Ausarbeitung und alles was damit zusammenhängt)*
- *die Beispiele sollten wenn möglich vom aktuellen Stand genommen werden*
  - *finden sich dort keine entsprechenden Beispiele, dürfen auch ältere Commits unter Verweis auf den Commit verwendet werden*
  - *Ausnahme: beim Kapitel "Refactoring" darf von vorne herein aus allen Ständen frei gewählt werden (mit Verweis auf den entsprechenden Commit)*
- *falls verlangte Negativ-Beispiele nicht vorhanden sind, müssen entsprechend mehr Positiv-Beispiele gebracht werden*
  - *Achtung: werden im Code entsprechende Negativ-Beispiele gefunden, gibt es keine Punkte für die zusätzlichen Positiv-Beispiele*
  - *Beispiele*
    - *"Nennen Sie jeweils eine Klasse, die das SRP einhält bzw. verletzt."*
      - *Antwort: Es gibt keine Klasse, die SRP verletzt, daher hier 2 Klassen, die SRP einhalten: [Klasse 1], [Klasse 2]*
      - *Bewertung: falls im Code tatsächlich keine Klasse das SRP verletzt: volle Punktzahl ODER falls im Code mind. eine Klasse SRP verletzt: halbe Punktzahl*
- *verlangte Positiv-Beispiele müssen gebracht werden*
- *Code-Beispiel = Code in das Dokument kopieren*

Kapitel 1: Einführung

Übersicht über die Applikation

Das Projekt „Schiffe Versenken“ realisiert das beliebte Gelegenheitsspiel Schiffe Versenken, welches normalerweise mit Stift und Papier gespielt wird, als Java-Konsolen-Applikation. Für das Spiel sind gelten folgende Regeln:

- Schiffe Versenken ist ein Zwei-Spieler-Spiel.
- Jeder Spieler platziert seine Schiffe auf einem 10x10-Gitterfeld.
- Die Schiffe dürfen horizontal oder vertikal platziert werden und dürfen nicht überlappen.
- Die Größe und Anzahl der Schiffe sind standardisiert: 1x5er-Schiff, 1x4er-Schiff, 2x3er-Schiffe, 2x2er-Schiffe.
- Die Spieler wechseln sich ab, um einen einzelnen Schuss auf das gegnerische Gitterfeld abzugeben, indem sie Koordinaten nennen. In diesem Spiel werden Koordinaten standardhaft in (X/Y) – Form angegeben.
- Wenn ein Schuss auf ein Feld abgegeben wird, das ein Schiff enthält, gilt es als "Treffer".
- Ziel des Spiels ist es, alle Schiffe des Gegners zu versenken, bevor er oder sie alle eigenen Schiffe verliert.

Die Umsetzung per Java-Applikation ermöglicht es auf Blatt und Papier zu verzichten und zu zweit an einem Gerät zu spielen

Wie startet man die Applikation?

Voraussetzung: IDE
Vorgehensweise:

- Importieren des Projekts von Git
- Ausführen der Main-Methode in src/adapters/Main.java

Wie testet man die Applikation?

Zum Testen der Applikation, wird das Repository in einer IDE benötigt. Da das Entwickeln mit eclipse geschah, wird hier das Vorgehen beispielsweise mit eclipse erklärt.

Nachdem das Projekt fertig geladen ist, finden sich zwei Packages: src und test, Durch Rechtsklicken auf das Test-Package kann die Option Run As > JUnit ausgewählt werden. Mit diesem Vorgehen lassen sich alle Tests auf einmal durchführen für jede Klasse.

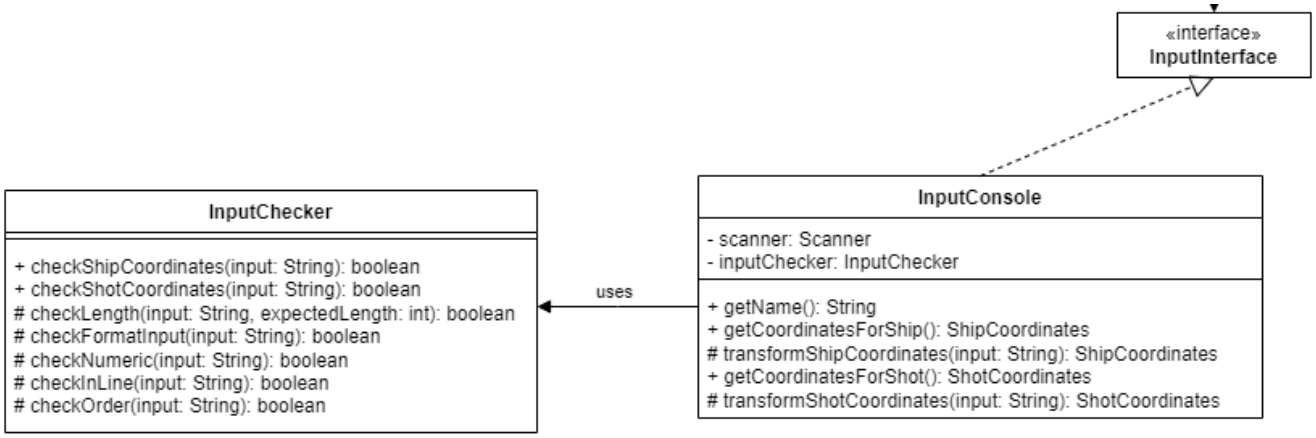
Falls einzelne Funktionen getestet werden sollen, finden sich zu den meisten Klassen und Methoden jeweilige Testklassen und -methoden in dem test-Package. Diese können wiederum über Rechtsklick > Run As > JUnit ausgeführt werden. Einzelne Methoden haben auch mehrere Testmethoden, die andere Fälle abdecken. Der genaue Fall wird jeweils am Ende der Testmethoden-Deklaration angefügt.

Kapitel 2: Clean Architecture

Was ist Clean Architecture?

Clean Architecture ist ein Software-Architekturmuster, das die Unabhängigkeit von externen Frameworks, Datenbanken und Benutzeroberflächen gewährleisten soll. Ziel ist es, die Geschäftslogik von technischen Details zu entkoppeln und die Wartbarkeit, Testbarkeit und Skalierbarkeit der Software zu verbessern. Das Kernprinzip einer Clean Architecture besteht darin, Code in mehreren Schichten zu organisieren, wobei jede Schicht eine klar definierte Verantwortung hat und von den anderen Schichten unabhängig ist. Die inneren Schichten enthalten die Kerndomänenlogik und sind am stabilsten und werden am wenigsten von externen Faktoren beeinflusst. Externe Schichten sind für die Kommunikation mit externen Systemen wie Datenbanken, Benutzeroberflächen oder Frameworks verantwortlich.

Analyse der Dependency Rule



Positiv-Beispiel: Dependency Rule

Als Positiv-Beispiel für eine korrekte Dependency Rule lässt sich die Klassen *InputConsole* und nennen. *InputConsole* ist in der Adapter-Schicht, denn es dient als Schnittstelle zum Benutzer und nimmt Benutzereingaben entgegen. Es hat 4 Abhängigkeiten: ein Interface zum Realisieren einer Dependency Inversion und drei Klassen in der Domain-Schicht.

Die *InputConsole*-Klasse hat einen *InputChecker* als Variable und ruft nach dem Erhalten der Benutzereingabe, die jeweilige check-Methode im *InputChecker* auf. *InputConsole* implementiert das Interface *InputInterface* und wird in der Game-Klasse indirekt verwendet. Erst in der Main-Klasse, das ebenfalls in der Adapter-Schicht ist, wird die konkrete Implementierung der InputConsole verwendet.

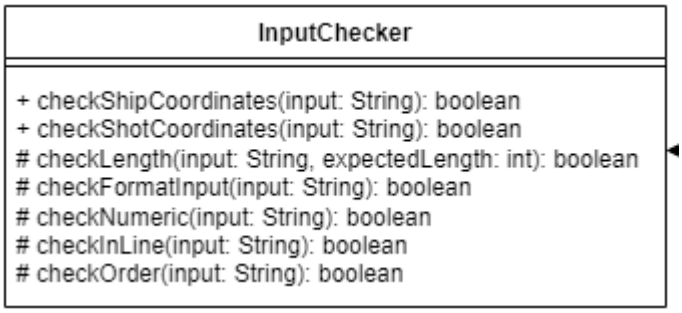
Negativ-Beispiel: Dependency Rule

Ein Negativ-Beispiel ist der *InputChecker*. Zwar befindet sich dieser in der Domain-Schicht und hat eigentlich keine Abhängigkeiten auf dem ersten Blick. Jedoch ruft diese Klasse *System.out.print* auf, welches eine Möglichkeit der Ausgabe darstellt. Somit wird eigentlich in der *InputChecker*-Klasse eine Abhängigkeit zur Ausgabe gemacht, die eigentlich in der Adapter-Schicht erst realisiert werden soll. Besser wäre eine Verwendung einer abstrakten Ausgabemöglichkeit. Da bereits die Game-Klasse eine abstrakte Ausgabelogik hat, kann die erhaltene Fehlermeldung als String mit dem boolschen-Wert, ob die Eingabe korrekt ist, zurückgegeben werden und dann in der Game-Methode über das Interface *OutputConsole* ausgeben werden.

Analyse der Schichten

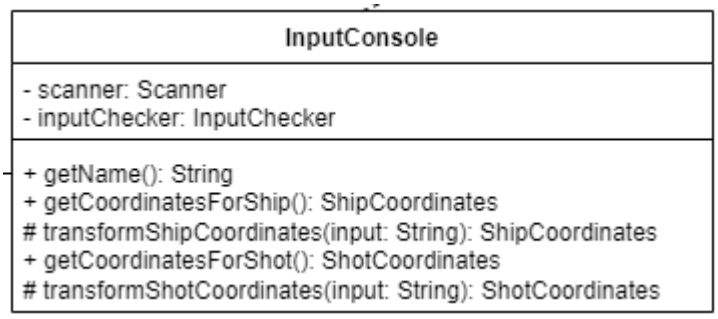
Schicht: Domain

- Klasse: *InputChecker*
- Aufgabe: Untersuchung der Eingabe auf Sinnhaftigkeit, d.h. Überprüfung des Formats der Koordinaten-Eingabe *ShipCoordinates* für das Platzieren eines Schiffs und der Koordinaten eines Schuss-Versuchs *ShotCoordinates*
- Einordnung: Domain, da eng an Geschäftslogik verbunden. Die Regeln des Spiels sind festgelegt und das Format der Eingabe ändert sich auch nicht. Die Klasse ist unabhängig von anderen Details und Benutzerschnittstellen.



Schicht: Adapters

- Klasse: *InputConsole*
- Aufgabe: Die Klasse nimmt die Benutzereingabe aus der Konsole entgegen und lässt diese vom *InputChecker* kontrollieren. Falls, die Werte stimmen, werden diese in ein jeweiliges *ShipCoordinates*-Objekt bzw. *ShotCoordinates*-Objekt umgewandelt, und danach an das Spiel weitergegeben.
- Einordnung: Adapter, da Klasse direkt mit Konsole verbunden und somit als Schnittstelle zum Benutzer fungiert. Durch die Idee des Implementierens eines Interfaces, kann die Schnittstelle zum Spiel einfach ausgetauscht werden. Das heißt, wenn man das Spiel jetzt in ein Spiel mit richtiger UI erweitern würde, muss lediglich die Benutzerschnittstelle neu implementiert werden, jedoch die Spiellogik selbst nicht verändert werden. Somit ist die Anwendungslogik entkoppelt von der Eingabelogik



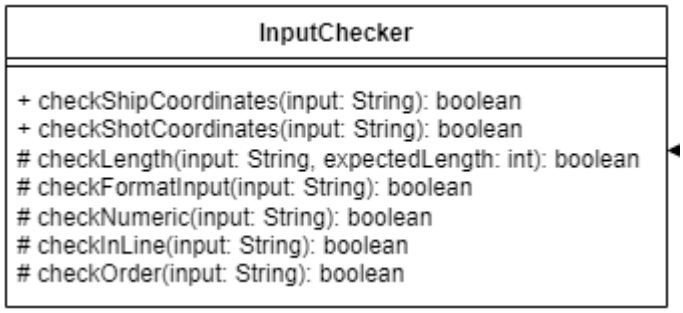
Kapitel 3: SOLID

Analyse Single-Responsibility-Principle (SRP)

Positiv-Beispiel

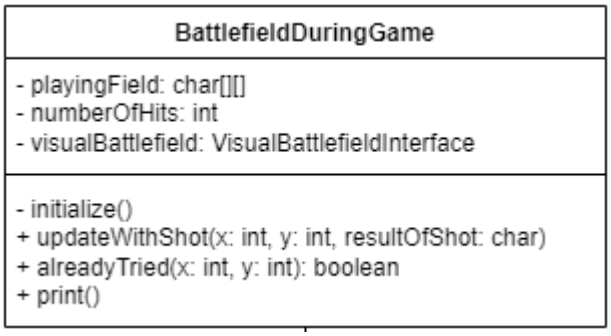
Das anschaulichste Beispiel ist wieder die *InputChecker*-Klasse. Ihr einziger Zweck ist die Überprüfung des Inputs. Dabei lassen sich zwei verschiedene Anwendungsfälle unterscheiden: Überprüfung der Schiffskoordinaten und der Schusskoordinaten. Man kann zwar argumentieren, dass die Klasse zwei Verantwortungen hat, jedoch überschneidet sich die Überprüfung der beiden etwa und somit müssen einzelne Methoden nicht öfter implementiert werden.

Die Überprüfung des Inputs erfolgt in mehreren Schritten. Die Schritte werden dann in einzelnen Methoden ausgeführt und von einer Hauptmethode *checkShipCoordinates* deligiert.



Negativ-Beispiel

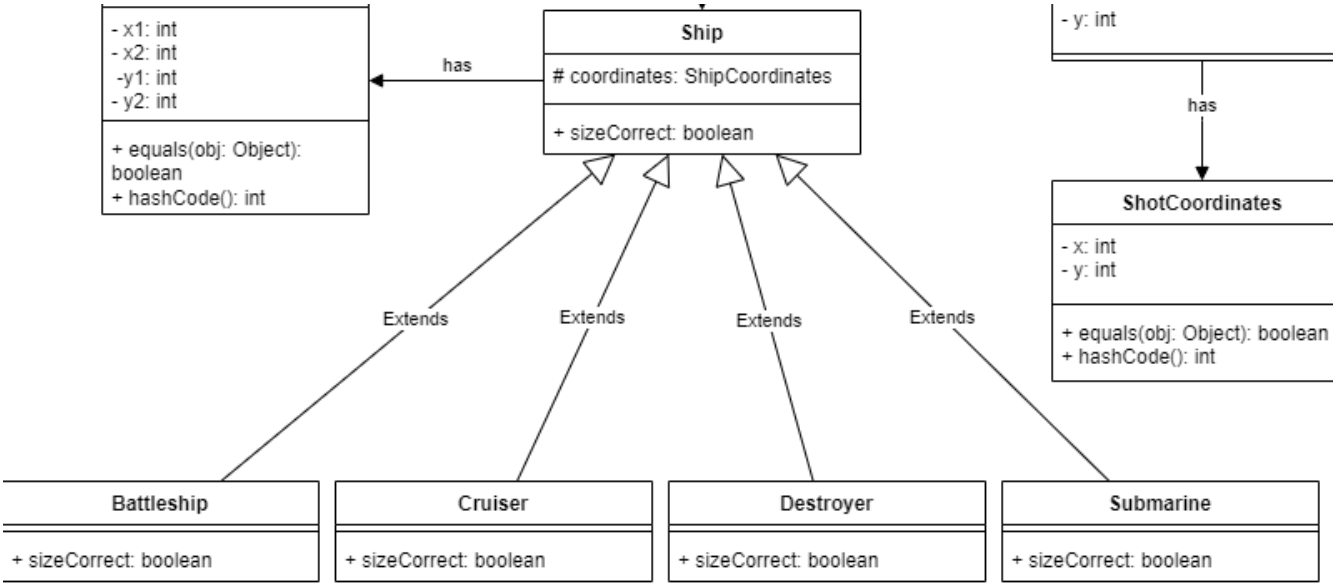
Als ein mögliches Negativ-Beispiel wäre die Klasse *BattlefieldDuringGame* zu nennen. Hier wird z.B. in der Methode updateWithShot() nicht nur die eigene Darstellung des Spielfelds, als einfaches 10x10 char-Array aktualisiert, sondern auch die visuelle Darstellung, welche als weitere Klasse realisiert wurde, upgedatet und zudem noch der aktuelle Stand ausgegeben. Besser wäre es nachdem die eigene Darstellung aktualisiert wurde, die Aktualisierung der visuellen Darstellung in eine eigene Methoden zu extrahieren und das Ausgeben des aktuellen Stands in die Game-Klasse zu verschieben, da dieses vollkommen unabhängig vom updaten sein soll und nur rein auf das Spielerlebnis einwirkt.



Analyse Open-Closed-Principle (OCP)

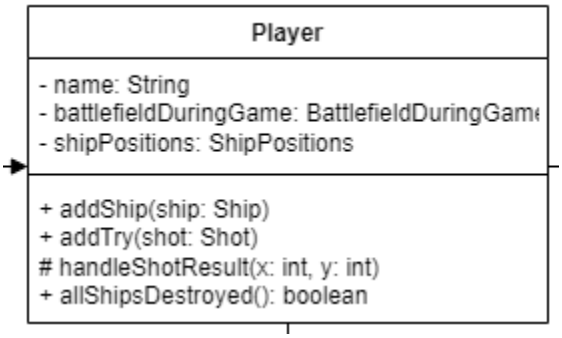
Positiv-Beispiel

Beim Spiel gibt es 4 Arten von Schiffen: Schlachtschiff, Kreuzer, Zerstörer und U-Boot, die sich jeweils in ihrer Größe unterscheiden. Da sie sonst alle gleich funktionieren sollen, bietet sich eine Implementierung einer abstrakten Klasse Ship an, die das Erstellen eines Schiffs standardisiert, jedoch die abstrakte Methode *checkSize()* hat, die in den einzelnen Schiffsklassen dann für die jeweilige korrekte Größe definiert wird. Somit muss nicht mithilfe mehrerer if-Anweisungen geprüft werden, ob es sich um das jeweilige Schiff handelt. Auch kann so eine neue Schiffsart hinzugefügt werden, ohne dass der Code verändert werden muss, sondern nur erweitert.



Negativ-Beispiel

Die *Player*-Klasse kann als Negativbeispiel angebracht werden. Es kann je nach Erweiterung unterschiedliche Spieler geben, z.B. natürlicher Spieler, Computerspieler. Mit der jetzigen Implementierung ist eine einfache Erweiterung verwehrt. Lösung: Implementierung einer abstrakten *Player*-Klasse.

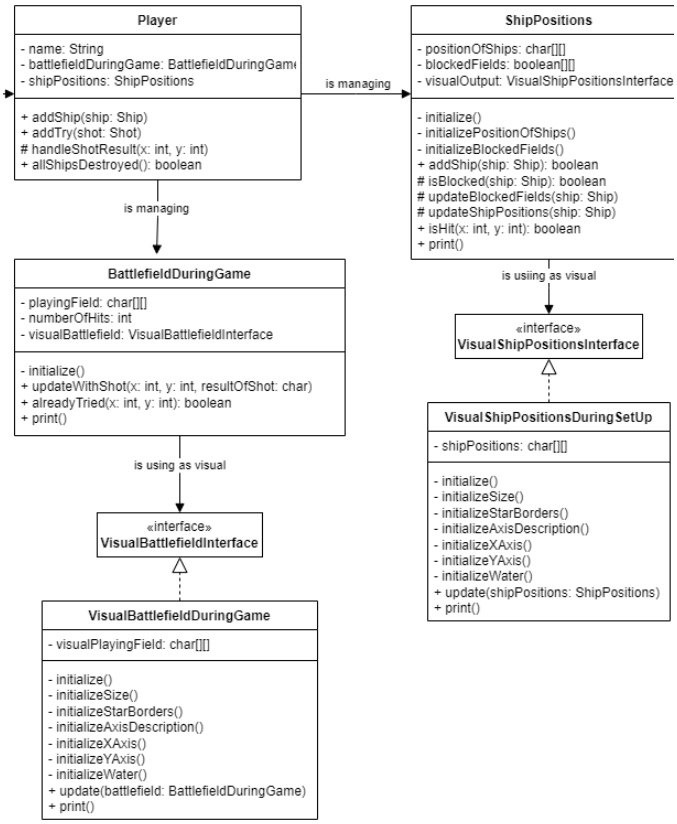


Analyse Liskov-Substitution- (LSP), Interface-Segreggation- (ISP), Dependency-Inversion-Principle (DIP)

Positiv-Beispiel

Beide visuellen Darstellungen VisualBattlefieldDuringGame und VisualShipPositionsDuringSetUp erfüllen das Dependency-Inversion-Principle. Die Idee hinter den beiden Klassen ist es, dass während des Spielsetups, d.h. die Platzierung der Schiffe, und dem Spielablauf selbst, eine visuelle Ausgabe durch die Konsole realisiert werden soll, die sich zwar an den dazugehörigen Klassen *BattlefieldDuringGame* und *ShipPositions* orientieren soll, aber für ein besseres Spielerlebnis besser gestaltet werden sollen. Da aber die Ausgabe je nach Geschmack veränderbar und erweiterbar sein soll, sollte diese aus der eigentlichen Logik des Speicherns des aktuellen Spielstands extrahiert werden.

Die dazugehörigen Klassen *BattlefieldDuringGame* und *ShipPositions* sind eng mit der Geschäftslogik verbunden und sind durch die Spielregeln festgesetzt. Somit befinden sich auch in der Domain-Schicht und sollten keine Abhängigkeit zu einer Implementierung einer visuellen Ausgabe sein. Deswegen wurde, um diese Abhängigkeit aufzulösen, jeweils ein *VisualInterface* eingeführt, welches einfach eine abstrakte Möglichkeit der Ausgabe ist. Somit hängen die beiden Klassen nicht von einer tatsächlichen Implementierung, sondern einer abstrakten Ausgabe, die einfach ausgetauscht werden kann.



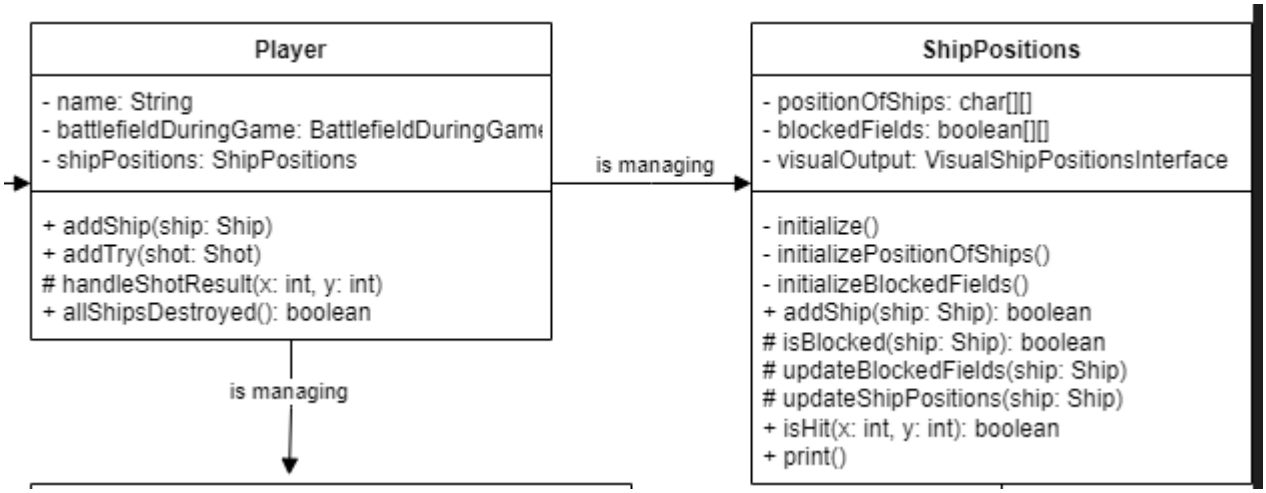
Kapitel 4: Weitere Prinzipien

Analyse GRASP: Geringe Kopplung

Positiv-Beispiel

Als Beispiel einer Klasse mit geringer Kopplung ist die Klasse *ShipPositions*, welche lediglich an ein Interface, also eine abstrakte Implementierung, welches für die visuelle Ausgabe des aktuellen Stands des Schiff-Setups ist. Änderungen in allen anderen Klassen hat keine Auswirkungen auf die Funktionsweise der Klasse. Zwar wird die Klasse *Ship* verwendet, jedoch ist diese auch in der Domain-Schicht und beinhaltet nur vom Spiel festgelegte Werte. Trotzdem könnte man hier noch weiter entkoppeln durch Ersetzen von *Ship* als Funktionswert in den Methoden *addShip* und *isBlocked*. Die Klasse lässt sich auch, bis auf die Ausnahme, dass ein Schiff für die beiden Methoden benötigt wird, vollständig unabhängig von anderen Klassen testen.

Die Hauptaufgabe der Klasse ist es die Schiffspositionen während des Setups zu verwalten und auch während des Spiels zu testen, ob auf dem Feld tatsächlich ein Schiff platziert worden ist.

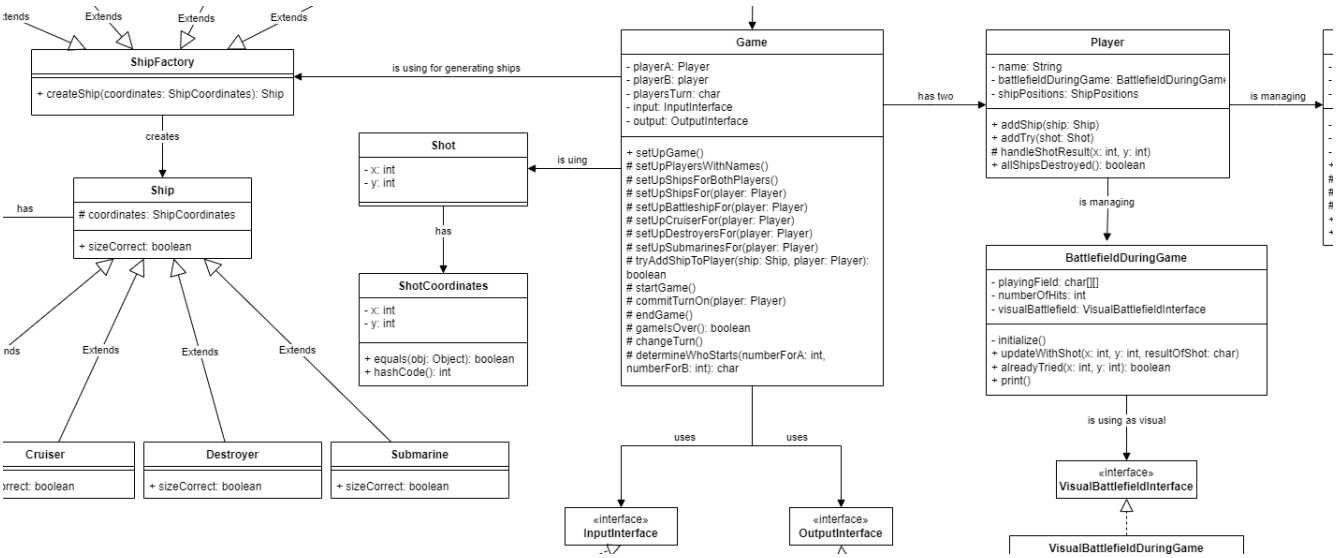


Negativ-Beispiel

Die Klasse *Game* hat im Gegensatz zur Klasse *ShipPositions* sehr viele Abhängigkeiten definiert. Es befindet sich in der Application-Schicht und ist zuständig für den generellen Ablauf des Spiels. Zwar liegen die meisten Abhängigkeiten in der Domain-Schicht, jedoch nicht alle. Eine wichtige Komponente des Spiels bilden natürlich die Spieler, welche direkt in der *Game* Klasse verwendet werden. Die Klassen innerhalb der Domain-Schicht werden sich wohl nicht großartig ändern, jedoch

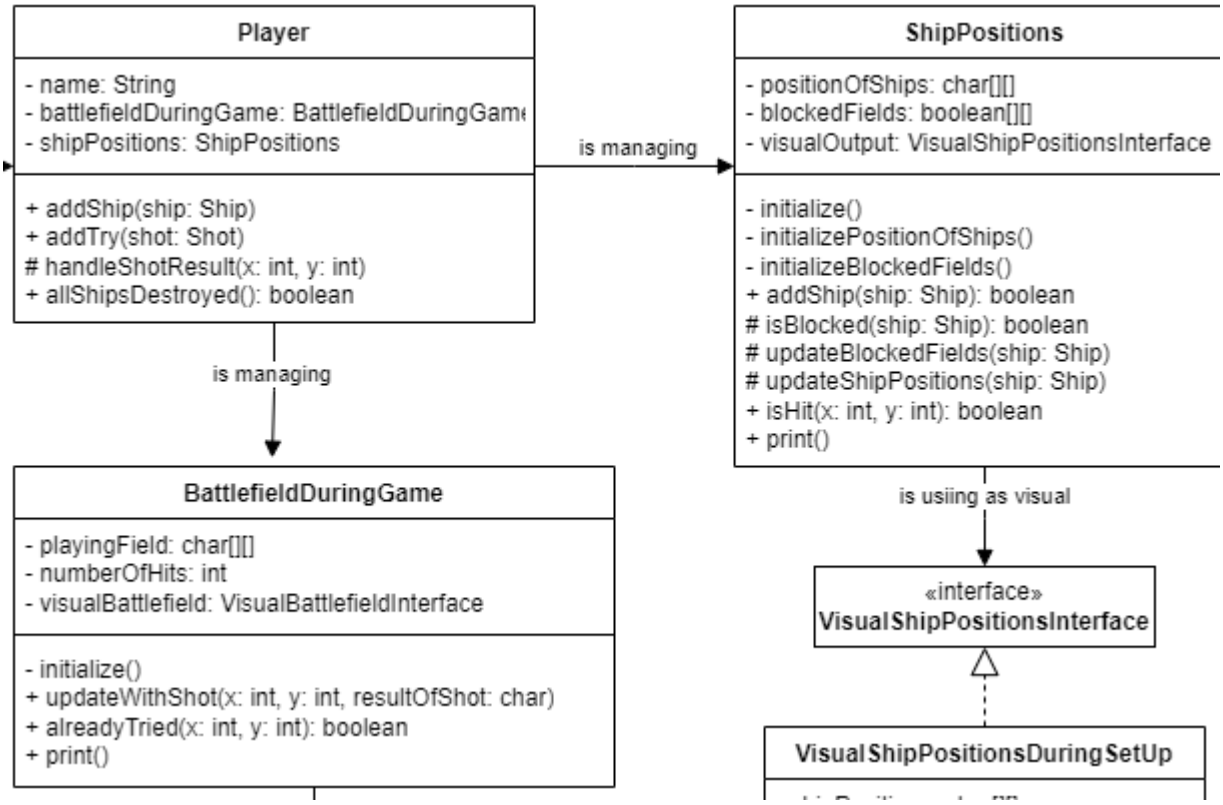
kann sich durch Refactoring o.ä. dort noch was verändern, jedoch noch umso mehr in der *Player*-Klasse.

Ein Lösungsansatz hier ist bereits wie schon öfter gesehen, das Einführen eines Interfaces für den Spieler. Das hätte auch den Vorteil, dass, falls das Spiel mit einem Spieler, der vom Computer gesteuert werden soll, erweitert werden soll, eine Umsetzung weniger Aufwand bedeuten würde.



Analyse GRASP: Hohe Kohäsion

Die Klasse *Player* ist ein Beispiel einer Klasse mit hoher Kohäsion. Es befindet sich in der Application Schicht und kennzeichnet den jeweiligen Spieler. Es verwaltet neben dem eigenen Namen, zwei weitere Klassen *BattlefieldDuringGame* und *ShipPositions*, die den Spielstand bzw. die Schiffspositionen des dazugehörigen Spielers speichern. Dadurch, dass diese Informationen fest im Spiel verankert sind und somit zur Domain gehören, kann *Player* davon abhängen, ohne dass man sich Sorgen machen muss, dass es durch Änderungen in den beiden Klassen zu Änderungen in *Player* kommt. Weitere Abhängigkeiten sind nur noch Interfaces, die an die beiden Klassen weitergegeben werden.



Don’t Repeat Yourself (DRY)

Commit: DRY v1 (886320a)

Ausgangslage: Für jede Art Schiff gibt es seine eigene Methode zum Weiterleiten des Schiffs an die Player-Klasse.

```
public boolean setUpCruiser(ShipCoordinates coordinatesOfCruiser, Player player) {
        Cruiser cruiser = new Cruiser(coordinatesOfCruiser);
        if (!cruiser.sizeCorrect()) {
                output.printLine("Error: Wrong size! Try again!");
                return false;
        }

        if (!player.getShipPositions().isBlocked(cruiser))
                player.addShip(cruiser);
        else
                return false;
        return true;
}
```

Umsetzung: Schiff ist bereits als abstrakte Klasse über allen Schiffstypen und somit wird das Einfügen des spezifischen Schifftyps durch eine Methode, die unabhängig von dem Schiffstyp funktionieren soll. Dadurch wird auch redundanter Code, der in den weiteren *setUp[Schiffstyp]* entsteht, gelöst.

```
public boolean setUpShip(Ship ship, Player player) {
                if (!ship.sizeCorrect()) {
                        output.printLine("Error: Wrong size! Try again!");
                        return false;
                }

                if (!player.getShipPositions().isBlocked(ship)) {
                        player.addShip(ship);
                        return true;
                } else
                        return false;
        }
```

Auswirkung: Eine einzige Methode zur Übermittlung der Schiffe an die Player-Klasse, Reduzierung des Codes und einheitliche Methode, die leichter erweiterbar und veränderbar ist.

Kapitel 5: Unit Tests

10 Unit Tests

Unit Test	Beschreibung
TestInputConsole#testTransformShipCoordinates	Transformation von Benutzereingabe zu *ShipCoordinates*
TestVisualBattlefieldDuringGame#testUpdate	Ausgabe für das Spielfeld während einer Runde soll geupdatet werden anhand von char-array

TestVisualShipPositionsDuringSetUp#testUpdate	Ausgabe für das Spielfeld während der Platzierung der Schiffe soll anhand von char-array aktualisiert werden.
TestGame#testSetUpShip	Erfolgreiches Aufstellen eines Schiffs auf dem Feld von Player A
TestGame#testSetUpShipFailIsBlocked	Aufstellen eines Schiffs schlägt fehl, aufgrund von einem blockierten Feld
TestPlayer#testAddShip	Schiff wird erfolgreich auf dem eigenen Spielfeld platziert
TestPlayer#testAddShipTouching	Zweites Schiff kann nicht platziert werden, da es sonst das andere Schiff berühren würde.
TestBattlefieldDuringGame#testAlreadyTried	Prüft, ob bereits auf das Feld geschossen wurde.
TestBattlefieldDuringGame#testUpdateWithShotSuccess	Fügt einen erfolgreichen Schuss auf der Karte ein.
TestInputChecker#testCheckShipCooridnates	Eingabe soll als korrekt identifiziert werden.

ATRIP: Automatic

Zur Realisierung der Automatic-Eigenschaft wurde zuerst eine Vielzahl an Tests geschrieben, die möglichst viele Funktionen des Spiels abdecken. Danach wurde mithilfe von Maven konfiguriert, dass bei jedem Build der Anwendung automatisch alle Unit-Tests im test-package ausgeführt werden und abschließend ausgwertet.

ATRIP: Thorough

Positiv-Beispiel

Die Klasse *TestPlayer* ist ein Beispiel für eine Testklasse, die Thorough erfüllt. Dafür spricht:

- Hohe Testabdeckung: bis auf einzelne setter- und getter-Methoden ist die komplette Klasse mit Tests abgedeckt
- Allen Methoden-Pfade wird nachgegangen. Somit sind alle mögliche Fälle abgedeckt.
- Die Methode testet nicht nur eine einzelne Komponente, sondern das Zusammenspiel mit anderen Klassen wie *ShipPositions* und *BattlefieldDuringGame*

```
public void testAddShip() ()
public void testAddShipTouching ()
public void testAddShipsNextToEachOther ()

public boolean addShip(Ship ship) {
    if (shipPositions.addShip(ship))
            return true;
    else
            return false;
}
```

Negativ-Beispiel

Die Testabdeckung der Game-Klasse kann als nicht Thorough angesehen werden. Dafür spricht vor allem die niedrige Testabdeckung der Klasse, obwohl sie zu den wichtigsten gehört. Grund dafür ist, dass die

Klasse ein Zusammenspeil von vielen unterschiedlichen Klassen ist. Dazu gehört die Input-, die Output- sowie die Player-Klasse mit ihren weiteren Unterklassen. Ein Merkmal von Thorough ist, dass das Zusammenspiel von Komponenten getestet wird.

Grund dafür ist, dass einige Methoden wie addSubmarines mehrere Inputs in einer Schleife entgegennehmen. Innerhalb eines JUnit Tests kann ich jedoch nur zu Beginn des Tests einen Input einsetzen.

```
public boolean setUpSubmarines(Player player) {
        int numberOfSubmarines = 0;

        while (numberOfSubmarines < 2) {
                output.printLine("Enter the desired coordinates in format: startX startY endX endY");
                Submarine submarine = new Submarine(input.getCoordinatesForShip());
                if (setUpShip(submarine, player))
                        numberOfSubmarines++;
        }
return true;
}
```

Weiteres Beispiel ist die Methode *startGame*(), in der mindestens ein Spieler 19 Mal schießen muss, um alle Schiffe zu versenken. Lösungsweg hier wäre, eine andere Implementierung von der Methode, um zu bestimmen, ob ein Spieler verloren hat.

Auch hängen Methoden zum späteren Zeitpunkt vom Spiel von Zuständen, die in anderen Methoden ausgeführt werden. Wie gerade beschrieben muss ein Spieler 19 Treffer laden, um zu gewinnen, was auch bedeutet, dass davor 19 Schiffsfelder platziert worden sein müssten.

ATRIP: Professional

Positiv-Beispiel

Eine gute Testklasse im Bezug auf Professional ist *TestShipPositions*. Hier wird zu Anfang in der @Before-Sektion bereits einmal ein *ShipPositions*-Objekt angelegt, welches im Anschluss von allen Methoden verwendet wird, was das ganze effizienter macht.

Die Leserlichkeit der Tests ist recht gut, vor allem die Namen der jeweiligen Testmethoden, sagen immer klar aus, welche genaue Methode bzw. welcher Pfad getestet wird.

testIsNotBlocked()

testUpdateShipPositionsForHorizontalShip()

testAddShipFailBecauseOfBlock()

Jeder der Tests kann unabhängig voneinander getestet werden, sodass jede einzelne Komponente isoliert getestet werden kann und ein Fehler genau lokalisiert und ausgebessert werden kann.

Negativ-Beispiel

Die Testklasse *TestInputConsole* ist ein Beispiel für fehlende Professionalität. Zum einen ist die Ersetzung der User-Eingabe in die Konsole durch die aktuelle Umgehung recht unleserlich. Sie erfüllt zwar ihren Zweck, jedoch auf Kosten der Verständlichkeit.

```
String input = "0 0 1 0";
InputStream in = new ByteArrayInputStream(input.getBytes());
Scanner scanner = new Scanner(in);
inputSource.setScanner(scanner);
```

Die Effizienz kann man hier auch verbessert werden. Zwar macht das in dem aktuellen Rahmen kein Grund zur Sorge, dass die Effizienz leidet, jedoch geht es um das Prinzip dahinter. Denn im jetzigen Stand wird für jede einzelne Methode ein neues *InputConsole*-Objekt erzeugt. Das könnte über eine @Before Lösung wie in Klasse *TestPlayer* umgestaltet werden, dass lediglich einmal das Objekt initialisiert werden muss.

```
public class TestPlayer {
	Player testPlayer;
	@Before
	public void setUp() {
		VisualShipPositionsDuringSetUp positions = new VisualShipPositionsDuringSetUp();
		VisualBattlefieldDuringGame battlefield = new VisualBattlefieldDuringGame();
		testPlayer = new Player("Test", positions, battlefield);
	}
```

Code Coverage

Klasse	**Coverage**	**Begründung**
BattlefieldDuringGame	~ 91%	Keine direkte Abdeckung der setter- und getter
Game	~ 11%	Viele Methode nicht sinnvoll testbar, da entweder zu einfach wie *changeTurn*(), oder Methoden mit mehreren Benutzereingaben. Komplexere nicht getestete Methoden wurden bereits in Einzelheit getestet
InputConsole	~ 88%	Nicht abgedeckte Methoden / Abläufe entweder zu einfach, s. getName() oder rekursiver Aufruf der Methode
InputChecker	~ 79%	Nicht abgedeckte Abzweigungen in den Methoden *checkShipCoordinates()* und *checkShotCoordinates()* über die Einzelmethoden abgetestet

OutputConsole	0 %	Ganz primitive Methoden, die keine Testabdeckung benötigen
Player	~ 75%	Nur setter- und getter-Methode nicht abgedeckt, sowie *allShipsDestroyed*(), da eigentliche Logik in *BattlefieldDuringGame*
ShipPositions	~ 86%	Nur print-Methoden zum Debugging nicht abgedeckt
VisualBattlefieldDuringGame	~ 88%	Lediglich einzelne setter- und getter nicht getestet, sowie Ausgabemethode für Konsole
VisualShipPositionsDuringSetUp	~ 91%	Nur print-Methode nicht getestet.
GESAMT	~ 70%	Application-Layer recht wenig durch UnitTest abdeckbar, sondern mehr durch manuelle Tests

Fakes und Mocks

Keine Mocks und Fakes verwendet.

Kapitel 6: Domain Driven Design

Ubiquitous Language

Alle Beispiele aus der *Player*-Klasse:

Bezeichung	**Bedeutung**	**Begründung**
Ship	Schiff	Stellt ein Schiff im Spiel dar
BattlefieldDuringGame	Schlachtfeld während des Spiels	Repräsentiert das Schlachtfeld während des Spiels
addShip	Schiff hinzufügen	Fügt ein Schiff zum Spielfeld hinzu
Player	Spieler	Repräsentiert eine Person, die am Spiel teilnimmt

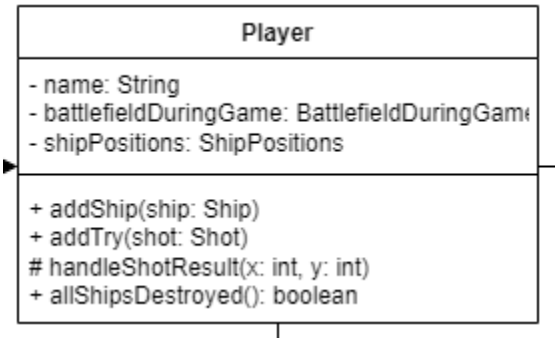
Entities

Der Einsatz einer Entity-Klasse für den *Player* hat mehrere Gründe:

- Identität: Ein Spieler hat eine eindeutige Identität, die durch seinen Namen repräsentiert wird. Die Player-Klasse erfasst diese Identität und ermöglicht die eindeutige Zuordnung von Spielerinformationen im Spiel. Klar, kann man sagen, dass ein Name nicht eindeutig ist, jedoch wird hier auf den Verstand der beiden Spielparteien gesetzt, die sich durch gleiche Namen, selbst das Spiel zerstören. Das Spiel selber würde trotzdem funktionieren, nur wissen die Spieler selbst nicht, wer wann dran ist.
- Zustand: Die Player-Klasse enthält den Zustand des Spielers, einschließlich des aktuellen Schlachtfelds während des Spiels und der Positionen der Schiffe. Der Zustand kann sich im Laufe

des Spiels ändern, z. B. durch das Hinzufügen von Schiffen oder das Aktualisieren des Schlachtfelds mit Schüssen.

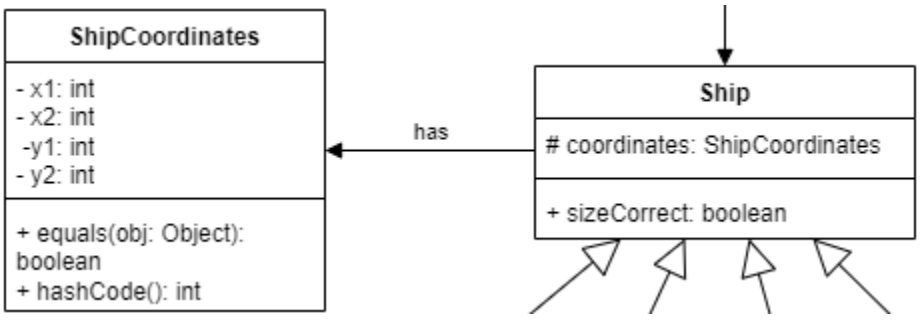
- Interaktionen: Andere Komponenten oder Klassen im Spiel können mit einem Player-Objekt interagieren, um Informationen über den Spieler abzurufen oder den Zustand des Spielers zu aktualisieren. Zum Beispiel können Schüsse auf das Schlachtfeld des Spielers hinzugefügt werden, um zu überprüfen, ob ein Schiff getroffen wurde.



Value Objects

Klasse *ShipCoordinates* bietet sich optimal als Value Object an. Folgende Eigenschaften erfüllt die Klasse:

- Unveränderlichkeit: Die Koordinaten sind unveränderlich, also müssen nach der erstmaligen Erstellung nicht mehr verändert werden.
- Semantische Klarheit: *ShipCoordinates* repräsentieren die bestimmten Werte der Koordinaten eines Schiffs und wird nicht durch eine Identität definiert, sondern rein über die Eigenschaften x1, x2, y1 und y2.
- Gleichheitsvergleich: Da *ShipCoordinates* rein auf die vier Koordinatenbestandteile aufbaut, ist ein Gleichheitsvergleich leicht zu implementieren. Siehe dafür die Methode *equals*
- Vereinfachte Modellierung: Ein Schiff kann noch weitere Eigenschaften haben, was es recht komplex und unübersichtlich macht, wenn jede Koordinate als einzelnes Attribut definiert wird. Indem man die Koordinaten in einem eigenen Objekt zusammenfasst, vereinfacht man die Schiffs-Domäne.



Repositories

Im vorliegenden Projekt wurde kein Repository verwendet. Die Hauptgründe für die Verwendung eines Repositories sind unter anderem:

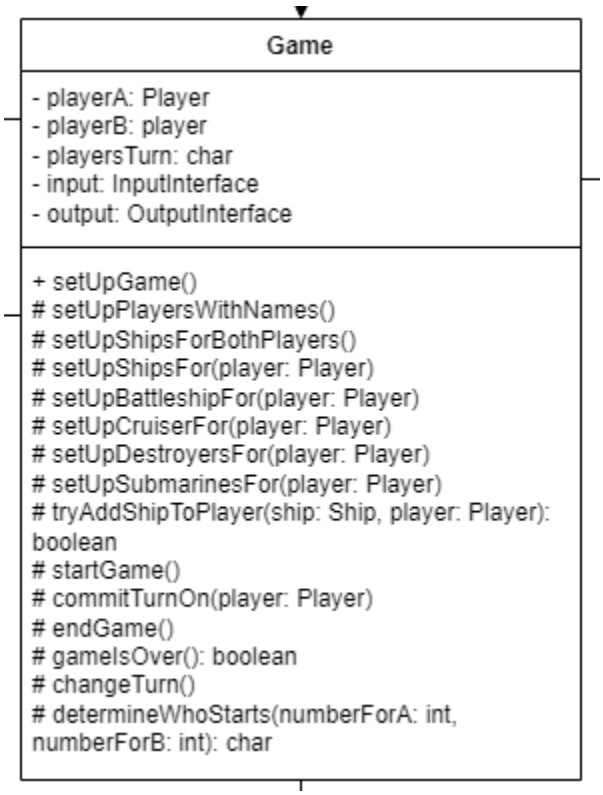
- Abstraktion des Datenzugriffs: Ein Repository abstrahiert den Zugriff auf die Datenquelle, egal ob Datenbank, Datei oder ein sonstiger externer Dienst. Es bietet eine einheitliche Schnittstelle, um Daten zu speichern, abzurufen, zu aktualisieren und zu löschen.
- Trennung von Domänenlogik und Datenzugriff: Ein Repository fördert die Trennung von Domänenlogik und Datenzugriff. Es ermöglicht, dass die Domänenlogik unabhängig von der Art der Datenquelle bleibt.

Im konkreten Fall dieses Projektes ist keine Datenpersistenz angedacht, denn die Datenstrukturen soll direkt im Speicher gehalten werden und nicht über das Spiel hinaus noch benötigt werden. Die Domänenlogik soll direkt auf die *Player* Datenstruktur zugreifen können, ohne eine weitere Abstraktionsebene einzufügen. Ein Repository würde an dieser Stelle keinen signifikanten Nutzen bringen und eher den Code komplexer machen.

Aggregates

Die Klasse "Game" in Ihrem Code könnte als Aggregat betrachtet werden. Es enthält die zentralen Operationen und Geschäftsregeln für das Schiffe-Versenken-Spiel und agiert als die Aggregatwurzel. Hier sind einige Gründe, warum die Klasse "Game" als Aggregat betrachtet werden könnte:

- Zusammengehörende Objekte: Die Klasse "Game" enthält Instanzen der Klasse "Player", die eng mit dem Spiel verbunden sind und die Spielerdaten und Spiellogik repräsentieren.
- Konsistenz und Integrität: Die Klasse "Game" enthält Methoden wie "setUpShipsFor", "setUpShip" und "commitTurnOn", die sicherstellen, dass die Schiffe korrekt platziert werden und die Spielregeln eingehalten werden.
- Transaktionale Einheit: Das "Game"-Objekt ermöglicht den Ablauf des Spiels, indem es die Spielzüge der Spieler verwaltet und den Spielverlauf kontrolliert.
- Aggregatwurzel: Die Klasse "Game" fungiert als Eingangspunkt für den Zugriff auf andere Objekte innerhalb des Aggregats und enthält die zentralen Operationen und Geschäftsregeln des Spiels.



Kapitel 7: Refactoring

Code Smells

Duplicated Code

Commit: Code Smells v1 (3f3f00b)

Vor Refactoring: In der *Game*-Klasse wird an mehreren Stellen der Spieler aufgefordert, Koordinaten einzugeben. Dabei wird über die *OutputConsole*-Klasse in der Konsole aufgefordert, diese Koordinaten in einem speziellen Format einzugeben. Je nach Art der Ausgabe, die ja unabhängig von der eigentlichen Spiellogik ablaufen soll, müsste bei einer Änderung der Ausgabe, an mehreren Stellen in *Game* auch mehrere Änderungen vorgenommen warden.

Lösung: Auslagerung in *OutputConsole* und Aufruf einer Methode des Interfaces zur Konsolenausgabe, also statt: *output.printLine("Enter the desired coordinates in format: startX startY endX endY");* folgenden Code:

```
@Override
public void askForShipCoordinates() {
        System.out.println("Enter the desired coordinates in format: startX startY endX endY");
}
```

Long Method

Commit: Code Smell: Long Method (446d2a0)

Die Methode addTry(Shot shot) ist relativ lang und enthält mehrere Verzweigungen. Über diese Verzweigungen wird die genaue Funktionweise nicht klar lesbar. Um diese Methode zu verbessern, wurde folgender Block aus der Methode in eine eigene Methode extrahiert und anschließend die if-Anweisung vereinfacht:
Ursprünglich:

```
if (shipPositions.isHit(shot.getX(), shot.getY())) {
        battlefieldDuringGame.updateWithShot(shot.getX(), shot.getY(), 'X');
} else {
        battlefieldDuringGame.updateWithShot(shot.getX(), shot.getY(), 'O');
}
```

Neu:

```
public void handleShotResult(int x, int y) {
        char result = shipPositions.isHit(x, y) ? 'X':'O';
        battlefieldDuringGame.updateWithShot(x, y, result);
}
```

2 Refactorings

Commit: Refactoring v1 (695d027)

Es erweist sich schwieriger, da von Anfang an schon versucht wurde, sprechende Namen und gekapselte Methodenumfänge zu verwenden. Verbesserungspotenzial wurde in Klase *Game* gefunden.

Extract Method

Methode *setUpShipsFor* war zuvor sehr unübersichtlich und hat unter anderem das Zusammensetzen von den beiden einzelnen Schiffen *Cruister* und *Battleship* übernommen. Die Erstellung der Schiffe und das Hinzufügen auf dem Feld wurde für jeden Schiffstyp nun in eine einzelne Methode gekapselt. Somit sind die unterschiedlichen Phasen einfacher zu identifizieren und der Code besser lesbar und wartbarer.

Rename Method

In der *Game*-Klasse finden sich viele Methoden, die *setUpShip* o.ä. heißen. Aufgrund dessen wurden einige Methoden umbenannt, um klarzumachen, welche Aufgabe jede einzelne Methode wirklich hat.

Kapitel 8: Entwurfsmuster

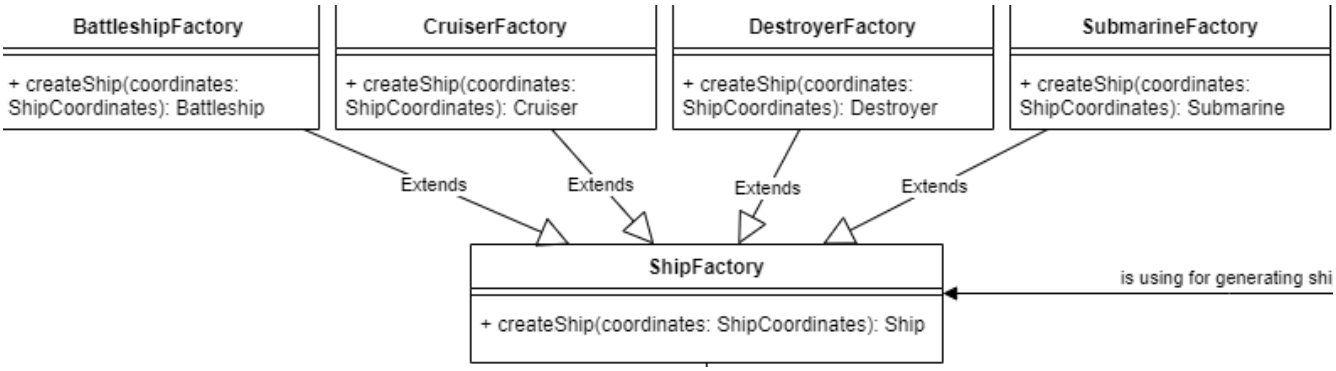
Entwurfsmuster: Erbauer

Für die Erzeugung der Schiffe wurde auf das Erbauer-Pattern gesetzt. Folgende Gründe sprechen dafür:

- Entkopplung von Objekterzeugung und Verwendung: Durch die Verwendung des Erzeugermusters wird die Erzeugung von Schiffen von ihrer Verwendung entkoppelt. Statt direkt Objekte zu instanziieren, wird die entsprechende Methode der Fabrikklassen aufgerufen, um ein Schiff zu erstellen. Dadurch muss man sich nicht mehr um die Details der Objekterzeugung kümmern und erhält eine saubere Trennung zwischen Erzeugung und Verwendung von Schiffen.
- Vereinfachte Erweiterbarkeit: Das Erzeugermuster ermöglicht es, neue Schiffstypen hinzuzufügen, indem du einfach eine neue Fabrikklasse erstellt wird und die Methode

createShip() entsprechend anpasst. Dadurch wird die Erweiterung des Codse erleichtert, ohne dass bestehender Code geändert werden musst.

- Zentrale Kontrolle der Objekterzeugung: Durch die Verwendung des Erzeugermusters hat man eine zentrale Stelle, an der die Objekterzeugung stattfindet. Dieses erleichtert die Wartung und Verwaltung des Codes, da man nur an einer Stelle auf die Erzeugung von Schiffen zugreifen kann.



Kein 2. Entwurfsmuster verwendet