Warum betreiben wir modularen Klassenentwurf?

An dieser Stelle kann man sich mit zwei Fragestellungen befassen:

1. **Warum macht man das überhaupt?** Könnte man nicht einfach alles Funktionalität in einer Klasse unterbringen, anstatt das Programm auf viele einzelne Klassen(dateien) zu verteilen?
2. Wenn die OO-Modellierung eine Problems nicht eindeutig ist - **woran erkennt man dann, ob man es „gut“ gemacht hat?**

Warum verteilt man die Funktionalität und den Code auf mehrere Klassen?

Wenn man ein Problem sinnvoll modularisiert und modelliert, hat das viele Vorteile:

- **Lesbarkeit des Quellcodes** → Etwas stimmt mit dem Tor nicht? Also muss man in der „tor“-Klasse schauen und nicht 5000 Zeilen Code durchscrollen, bis man zu dem Teil kommt, der das Tor erzeugt. Es **erleichtert Änderungen** an der Funktionalität, wenn stets klar ist, wo bestimmte Eigenschaften und Fähigkeiten festgelegt sind.
- Wenn man **Klassen** geschickt modelliert, kann man Sie in anderen Programmen **wiederverwenden** - nicht umsonst spricht man von „Klassenbibliotheken“.
- **Neue Objekte** können durch **neue Klassen** ein ein Modell eingefügt werden - du willst Hindernisse auf dem Spielfeld? Kein Problem mit der zusätzlichen „hindernis“-Klasse.

Wann ist ein Klassenentwurf "gut"?

Ein Klassenentwurf ist also „gut“, wenn er die oben genannten Vorteile maximal unterstützt - hierfür kann man zwei Eigenschaften des Entwurfs betrachten:

- **Kohäsion**: Die Kohäsion einer Klasse definiert ihren logischen Zusammenhalt als Einheit. Erkennen kann man das daran, wie deutlich Sie sich von anderen Klassen des Modells abgrenzt, aber auch daran, dass ihre Methoden für klar umrissene Aufgaben zuständig sind. Beispiel: Wenn man bei der Implementation einer Liste eine Methode`isEmpty(): boolean` definiert, die nur schaut, ob der Startknoten null ist, erscheint das im ersten Augenblick übertrieben, erhöht jedoch die Kohäsion der Klasse. Anstatt an vielen Stellen eine interpretationsbedürftige Abfrage `start==null` im Code zu haben, hat man eine semantisch klare Methode `isEmpty()`. **Man möchte also eine hohe Kohäsion der Klassen haben.**
- **Kopplung**: Die Eigenschaft „Kopplung“ beschreibt die Bindung zwischen den Klassen. **Die Kopplung soll möglichst gering sein**, das erreicht man dadurch, dass die Abhängigkeiten zu anderen Klassen möglichst klein gehalten werden sollten. Schnittstellen zwischen Klassen sollten klar definiert sein, mit bedeutungsvollen Parametern. Anstatt also eine Getter-Methode zu schreiben, die einen Parameter benötigt, der ihr mitteilt, welches Attribut zurückgegeben werden soll, bekommt jedes Attribut einen eigenen Getter mit sprechendem Namen - und ohne Parameter.

Grundregeln für gute Klassenentwürfe

Kapselung und Geheimnisprinzip

Klassenvariablen niemals öffentlich (public) deklarieren. Zugriff auf Attribute von anderen nur über **sondierende** und **verändernde** Methoden (get- und set-Methoden) möglich. Änderungen am internen Aufbau der Klasse haben keine Auswirkungen auf andere Klassen, welche mit dieser assoziiert sind.

Die Verwaltung der Position der Münzen in unserem Beispiel ist in Verantwortung der Muenzen-Klasse. Sollte die Verwaltung intern später auf ein Array mit zwei Feldern für x- und y-Koordinate umgestellt werden, so müssten bei direktem Zugriff von außen alle zugreifenden Klassen mitverändert werden - wenn der Zugriff über Getter- und Setter-Methoden gekapselt ist, müssen die assoziierten Klassen nichts über den internen Aufbau der Muenzen-Klasse wissen. Man spricht vom **Geheimnisprinzip**.

Anhand von Zuständigkeiten modellieren heißt, dass eine Klasse einen **logisch sinnvollen** und **klar abgegrenzten** Aufgabenbereich besitzt. In unserem Münzspiel überprüft das Tor zwar, ob eine Münze getroffen hat, verwaltet aber nicht die Koordinaten der Münzen.

Klar abgegrenzte Klassen-Zuständigkeiten

Entlang Zuständigkeiten zu modellieren bedeutet, dass eine Klasse einen logisch sinnvollen und klar abgegrenzten Aufgabenbereich besitzt. Im Münz-Schnipssspiel nimmt das Tor zwar Münzen auf, verwaltet aber nicht deren Koordinaten, ebensowenig wie das Spielfeld: Die Koordinaten gehören logisch zu den Münzen, darum werden sie auch von den Münz-Objekten selbst verwaltet.

Semantische, klare Aufgabenverteilung auf Methoden

Dasselbe, was für die Aufteilung des Problems in Klassen gilt, gilt innerhalb der Klassen für die Methoden: Jede Methode sollte eine klare Aufgabe haben - und einen Namen, der die Aufgabe auch verdeutlicht.

Die Klasse Spieler hat zwei Methoden - „muenzeEinwerfen“ und „muenzeSchnipsen“, die beiden Methoden sind funktional sehr ähnlich, es macht jedoch Sinn, die beiden zu trennen, da sie logisch zwei Abläufe des Modells umsetzen.

Keine Redundanz (DRY-Prinzip)

Man sollte es tunlichst vermeiden, identischen, also redundanten, Code an mehreren Stellen eines Programms zu verwenden. Dieses Prinzip wird oft auch das DRY-Prinzip (*Don't repeat youself*) genannt. So spart man Schreibaufwand, das Programm ist leichter zu verstehen und zu warten, denn man muss den Code nur einmal ändern und nicht noch alle Redundanzen.

In unserem Beispiel gibt es eine Methode `setPosition()` des Münz-Objekts, diese wird von allen assoziierenden Klassen aufgerufen, wenn die Position der Münze verändert werden soll. Wäre es an mehreren Stellen des Programmcodes möglich die entsprechenden Attribute des Münz-Objekts zu beeinflussen, müssten bei einer internen Änderung alle diese Stellen angepasst werden.