Übung 04

JULIA ZAMAITAT [redacted] LOTTE UNCKELL [redacted]

REPORT

1. Implementierung Erbauer

Unser Code: https://github.com/lotteunckell/Informatik03

Da und das Klassendiagramm bereits vorgegeben war, fiel der Einstieg in die Implementierung des Erbauers nicht schwer. Wir fingen mit den Interfaces an und definierten drei davon: *IFrame*, *IGearShift* und *ITire*. Die einzige Methode, die diese Interfaces von uns bekamen, war eine toString()-Methode, welche später von den einzelnen konkreten Klassen implementiert wird. Danach schrieben wir die konkreten Klassen der drei Interfaces, und zwar *MountainbikeFrame, MountainbikeGearShift, MountainbikeTire, RacerFrame, FacerGearShift* und *RacerTire*. Wir bemerkten an dieser Stelle, dass wir, um weitere Felder wie Größe anzugeben, abstrakte Klassen zwischen den Interfaces und den Implementierungen derer zwischenschalten wollen, also ergänzten wir den *AbstractFrame, AbstractGearShift* und *AbstractTire*. Diese abstrakten Klassen bekamen jeweils Felder mit Angaben wie Farbe, welche wir auf private setzten und getter und setter definierten. Wir überschrieben die default-Konstruktoren, um Parameter mit genau diesen Angaben übergeben zu können, um sie dann in diesen Feldern zu speichern.

Zurück in unseren konkreten Klassen mussten wir jeweils im Konstruktor als ersten Aufruf den super()-Konstruktor aufrufen, da diese Konstruktoren eben Parameter verlangen. Zudem implementieren wir hier auch jeweils die toString()-Methode und geben dort Details zu den Feldern aus den abstrakten Klassen aus. Diese Methode wird später implizit benutzt durch unsere konkreten Bikes.

Als nächstes schrieben wir unsere *AbstraktBike* Klasse. Sie bekommt drei protected fields, die die Interface Typen *ITire, IFrame* und *IGearShift* enthalten. Auch hier definieren wir eine abstrakte Methode toString(), welche später von den konkreten Bikes implementiert wird. Der Konstruktor bekommt von außen, in unserem Fall dem konkreten Builder, Angaben zu diesen Feldern, womit diese initialisiert werden.

Nach dem *AbstraktBike* folgen unsere konkreten Bikes *Mountainbike* und *Racer*. Die Konstruktoren dieser Klassen bekommen die Interface Typen *ITire, IFrame* und *IGearShift* als Parameter und rufen mit diesen den super()-Konstruktor auf. Sie implementieren die toString()-Methode und geben somit Details über ihre zuvor bekommenen Parameter aus, die implizit ihre überschriebene toString()-Methode aufrufen.

Somit waren wir mit der rechten Seite des Klassendiagramms fertig und machten mit der linken weiter: Wir implementierten den abstrakten *BikeBuilder*, der von uns fünf abstrakte Methoden bekam. Vier davon dienen der Erstellung der Komponenten und des Bikes und bekomme alle ihre dazugehörigen Parameter wie size und color. Eine weitere Methode ist die getResult()-Methode, die als Rückgabetypen das AbstractBike hat und

später von den konkreten Buildern genutzt wird, um das erstellte Bike zurückzugeben. Damit das funktioniert, mussten wir ein protected field vom Typ AbstractBike in den *BikeBuilder* schreiben, das wir *bike* genannt haben. Dieses Feld machen wir uns in den konkreten Buildern zu Nutze: Im *RacerBuilder* und im *MountainbikeBuilder* implementieren wir alle abstrakten Methoden des *BikeBuilders*. In der buildBike()-Methode speichern wir den konkret erstellten Typ von Bike (also Racer, Mountainbike) in dem bike-field des abstrakten *BikeBuilders* und returnen dieses als Ergebnis des buildBike()-Aufrufes. Auch in der überschriebenen getResult()-Methode returnen wir dieses Objekt.

Wir haben jetzt unsere Komponenten und die Builder, aber wir brauchen noch einen Direktor, der nun die Anweisung gibt, diese Komponenten zu konstruieren und zusammenzufügen. Das ist unser *BikeDirector*. Dieser hat ein privates Feld vom Typ BikeBuilder, in welches wir durch den Konstruktor, der einen solchen BikeBuilder als Parameter bekommt, diesen konkreten Typen dann speichern. Ansonsten hat unser *BikeDirector* nur noch eine weitere Methode namens construct(), die void als Rückgabetypen hat. Diese Methode bekommt alle Informationen, die unser Bike später braucht, also alle Parameter, die von den einzelnen Komponenten benötigt werden. Wir speichern uns diese Angaben in Variablen vom Typ *IFrame, IGearShift* und *ITire* und rufen in construct() am Ende buildBike() mit diesen Variablen als Parameter vom konkreten Builder auf.

Ein letzter Schritt fehlt nun noch: unsere Main-Klasse. Diese besteht aus dem Erstellen unserer konkreten Builder und Direktoren, denen wir diese konkreten Builder geben. Alles, was noch zu tun war, ist director.construct() aufzurufen, gewünschte Parameter zu übergeben und uns das Ergebnis des konkreten Builders durch getResult() zurückzugeben. Das Verhalten zeigte sich wie gewünscht.

2. Singleton

Das Singleton-Pattern wird verwendet, wenn man unter jeden Umständen nur eine Instanz von einer Klasse erzeugen lassen will. In der Vorlesung haben wir hierzu mehrere Male die beiden Beispiele der Runtime und des File Systems besprochen. Beide möchte man nur ein mal haben, da mehrere Instanzen zu Chaos führen würden.

Es gibt zwei Varianten, wie das Singletion-Pattern umgesetzt werden kann:

In beiden wird der Konstruktor auf private gesetzt, damit keine andere Klasse nach Belieben Objekte erzeugen kann. In der ersten Variante wird ein Feld eingeführt, welches über einen static-Block direkt getInstance() aufruft, sodass beim Laden der Klasse direkt eine Instanz erzeugt wird. Das belegt natürlich gleich Speicher, verbessert aber die Performance.

Bei der zweiten Variante wird auf den getInstance()-Aufruf gewartet, um das Objekt zu erzeugen. Sollte zum Zeitpunkt, wenn getInstance() aufgerufen wird, schon eine Instanz existieren, wird diese zurückgegeben. Diese Lösung ist nicht Thread-Safe, so lange man nicht mit synchronized arbeitet, dafür belegt sie nicht sofort Speicher.

2.1 Vor- / Nachteile des Singleton

Der Vorteil des Singleton ist ganz klar, dass der Konstruktor NUR von der Klasse selbst aufgerufen werden kann. So kann sichergestellt werden, dass nur unter bestimmten Umständen überhaupt eine Instanz erzeugt wird. Ansonsten wird eine bestehende Instanz zurückgegeben. So hat man strenge Kontrolle über den Zugriff auf diese Klasse.
In der Vorlesung hatten wir besprochen, dass das Singleton-Pattern es einem allerdings schwierig macht sicherzustellen, dass in verschiedenen JVMs oder Threads auch wirklich nur eine Instanz dieser Klassen existiert. Laut Wikipedia (https://de.wikipedia.org/wiki/Singleton_(Entwurfsmuster)#Nachteile) erhöht sich unter Verwendung dieses Entwurfsmusters wegen des exzessiven Benutzen von globalen Variablen auch die Gefahr, statt objektorientiert prozedural zu programmieren. Außerdem wurde aufgeführt, dass das Testen einer Implementierung mit Singleton-Muster sehr aufwendig ist, da man nicht so einfach Mock-Objekte erzeugen kann.
Da es außerdem schwierig ist, das Singleton-Muster korrekt und sinnvoll zu planen, wird dieses Muster teilweise als Anti-Pattern bezeichnet und es wird davon abgeraten, es allzu häufig zu verwenden. Wenn möglich sollen andere Muster benutzt werden. Wenn es allerdings unbedingt nötig ist, kann das Singleton-Muster trotzdem verwendet werden.

3. Reflection

Diese Woche haben wir ausgewürfelt, mit wem wir zusammenarbeiten. Wir hatten großes Glück, dass wir beide unsere Wunschpartnerin bekamen :)
Wir beide arbeiten schon seit über einem Semester zusammen an der IMImap. Daher haben wir unseren Rhythmus beim Pair Programming schon gefunden und mussten nicht mehr viel Zeit darauf verwenden, uns erst noch “einzugrooven”. Auch, dass wir in dieser Woche nicht mehr auf jeden Fall nach 20 Minuten tauschen mussten, hat die Arbeit viel angenehmer gemacht. So konnten wir die Arbeit in für uns angebracht erscheinende Teile aufsplitten. Nachdem so ein Teil beendet war, wurde das Ergebnis auf Github gepusht und die Andere war dran.