MUSTERGÜLTIGE ZUSAMMENARBEIT

SEW S06

17. DEZEMBER 2014

STEFAN ERCEG & MARTIN KRITZL
4AHITT

Inhalt

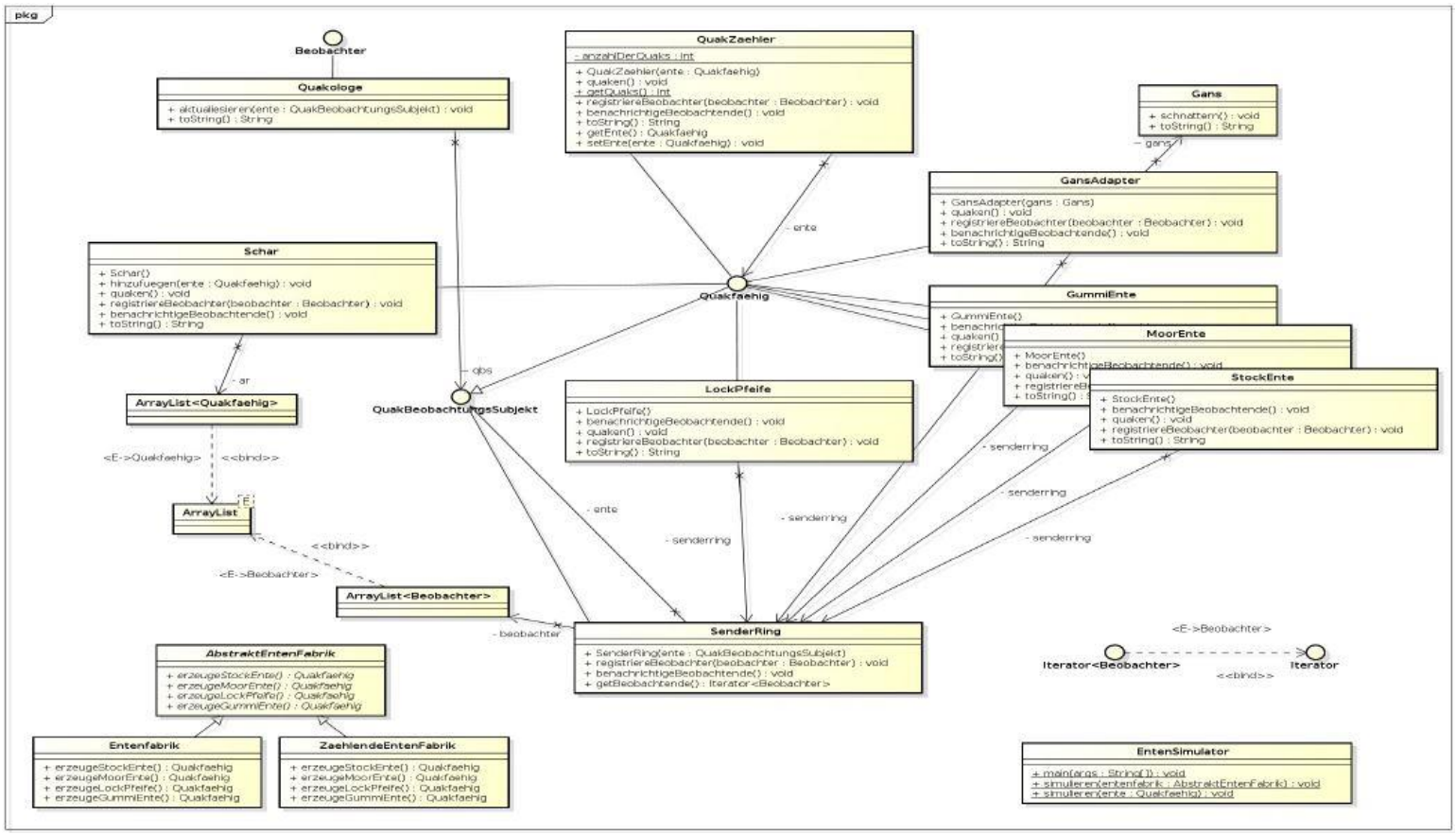
1. Github

Link: https://github.com/serceg-tgm/S06-Musterguelтige_Zusammenarbeit

Tag: erceg_kritzl_s06

2. Aufgabenstellung

Das UML-Diagramm für die Quakologie schaut folgendermaßen aus:



Folgendes ist zu erledigen:

- Implementieren Sie die Quakologie!
- Erkennen Sie die verwendeten Muster!
- Wann und wo wurden die Muster eingesetzt?
- Erkläre die verwendeten Muster (kleiner Tipp, es sind deren sechs!)

Geben Sie den entsprechenden Beispiel-Code (Java -> jar) und die gewünschte Dokumentation der Patterns (PDF) ab.

Es sind keine Test-Cases verlangt.

2. detaillierte Arbeitsaufteilung mit Aufwandsabschätzung

Teilaufgabe	benötigte Gesamtzeit
UML-Diagramm erstellen	60 Minuten
Implementierung des Programms inkl. JavaDoc	100 Minuten
Testung des Programms (JUnit)	100 Minuten
Protokoll schreiben	100 Minuten
Gesamt	**360 Minuten (6 h)**

3. anschließende Endzeitaufteilung

3.1 Erceg

Arbeit	Datum	Zeit in Minuten
UML, Implementierung und Protokoll	11.12.2014	100 Minuten
Protokoll	12.12.2014	40 Minuten
JavaDoc	13.12.2014	35 Minuten
Testen	17.12.2014	40 Minuten
Gesamt	*17.12.2014*	**215 Minuten (3 h 35 min)**

3.2 Kritzl

Arbeit	Datum	Zeit in Minuten
UML, Implementierung und Protokoll	11.12.2014	100 Minuten
JavaDoc und Protokoll	12.12.2014	40 Minuten
Testen	17.12.2014	40 Minuten
Gesamt	*12.12.2014*	**180 Minuten (3 h)**

3.3 Gesamtsumme

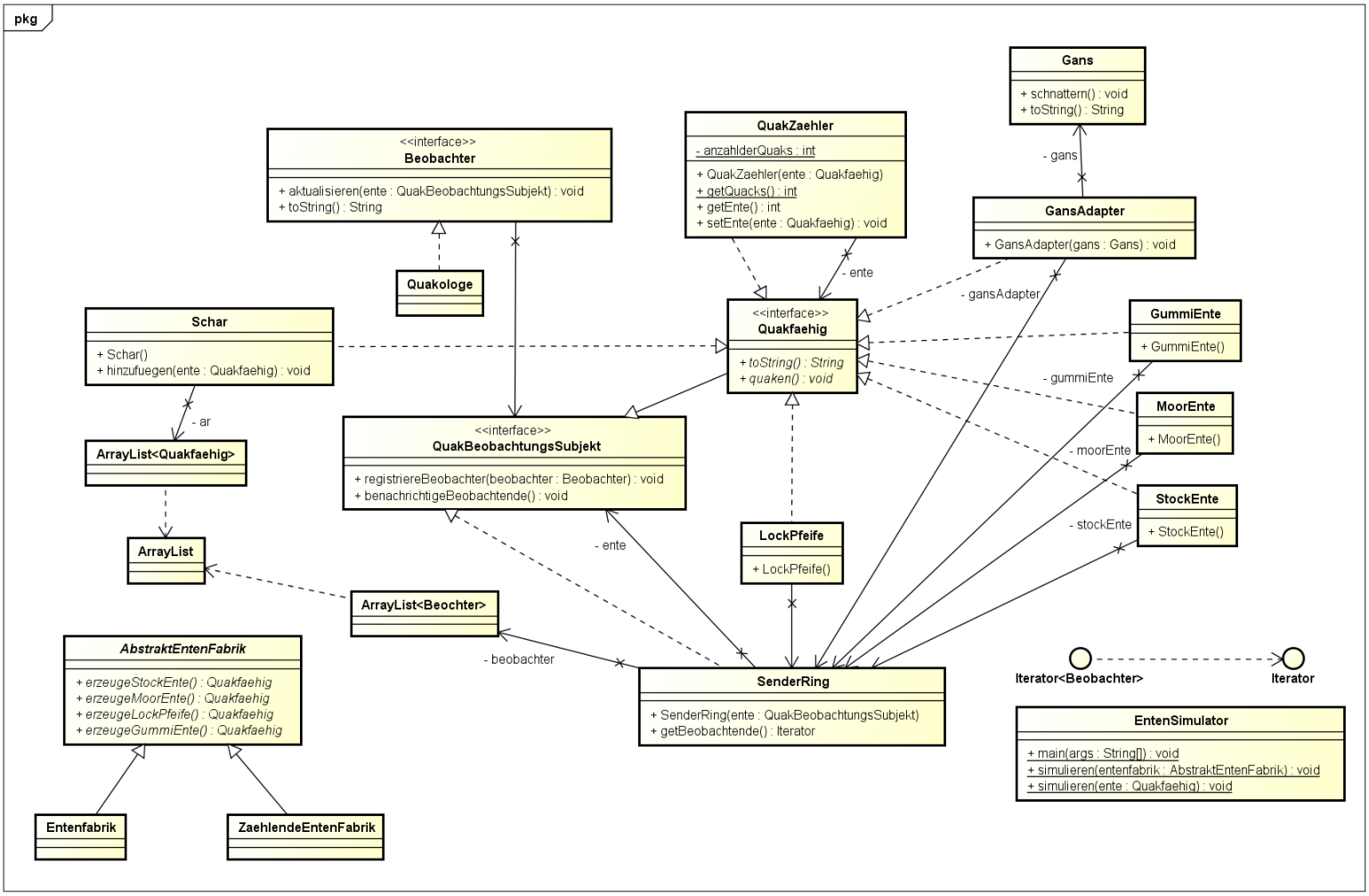
Insgesamt haben wir für diese Übung **6 Stunden und 35 Minuten** benötigt. Geschätzt wurden 6 Stunden, daher war unsere Einschätzung gar nicht so daneben.

Da wir den Code für dieses Beispiel von einer Internetseite [1] entnommen haben, benötigten wir keine Zeit für die Implementierung des Programms. Wir schrieben jedoch noch JavaDoc-Kommentare zu dem Code hinzu.

4. Designüberlegung

4.1 Abbildung und Beschreibung des UML-Diagramms

Das UML-Diagramm wurde mit dem Programm „Astah“ erstellt.



Die Struktur des UML-Diagramms wurde im Wesentlichen von der Vorlage übernommen. Jedoch wurden bei uns die Interfaces zu Stereotypes umgeändert. Ebenfalls müssen Methoden in Klassen, die von einem bestimmten Interface oder einer bestimmten abstrakten Klasse stammen, nicht explizit in dieser Klasse angegeben werden.

4.2 verwendete Muster

4.2.1 AbstractFactory Pattern

Die AbstractFactory kam für die AbstraktEntenFabrik zur Anwendung. In der Klasse sind Methoden enthalten, mittels der entschieden werden kann, um welchen Typ von Ente es sich handelt und welche erzeugt werden soll. Die Klassen Entenfabrik und ZaehlendeEntenFabrik erben von der AbstraktEntenFabrik. Dabei stellen die implementierten Klassen vom Interface Quakfaehig die Produktseite der AbstractFactory dar.

4.2.2 Observer Pattern

Die Klasse SenderRing, welche das Interface QuakBeobachtungsSubjekt implementiert, stellt das Subjekt des Observer Patterns dar. Durch diese Klasse werden alle Beobachter darüber informiert, dass eine Ente gesichtet wurde. Ebenfalls gibt es die Möglichkeit, weitere Beobachter hinzuzufügen, damit diese ebenfalls bei neuen Sichtungen von Enten benachrichtigt werden.

Der Quakologe, welcher das Interface Beobachter implementiert, stellt den Observer dar. Dieser besitzt eine Methode „aktualisieren“, welche bei einer Änderung beim Subjekt aufgerufen wird.

4.2.3 Iterator Pattern

Das Iterator Pattern wird bei dieser Übung dreimal angewendet, davon war es einmal in dem vorgegebenen UML-Diagramm vorhanden und zweimal wurde sie von uns noch zusätzlich hinzugefügt.

In der Klasse SenderRing ist es durch diese Anwendung möglich, alle eingetragenen Beobachter über eine Neuigkeit zu informieren.

Das Pattern kommt ebenfalls in der Klasse Schar bei den Methoden zur Registrierung und Benachrichtigung des Beobachters zum Einsatz.

4.2.4 Composite Pattern

Dieses Pattern kommt sowohl bei der Klasse SenderRing, als auch bei der Klasse Schar zur Anwendung.

Bei der Klasse SenderRing werden Objekte gleicher Typen (= Beobachter) durch die Methode „registriereBeobachter“ gesammelt und können mit Hilfe des Iterator Patterns (siehe Punkt 4.2.3) weiter verarbeitet werden.

Dasselbe Pattern wird auch bei der Klasse Schar verwendet, da dort Objekte gleicher Typen durch die Methode „add“ gesammelt werden. Danach erfolgt ebenfalls die Weiterverarbeitung mit Hilfe des Iterator Patterns (siehe Punkt 4.2.3).

4.2.5 Decorator Pattern

Sowohl bei der Klasse SenderRing, als auch beim Quakzaehler wird über den Parameter des Konstruktors eine Klasse, welche dasselbe Interface implementiert, übergeben. Dadurch kann dieser Vorgang ständig wiederholt werden. Damit könnte „QuakBeobachtungsSubjekt“ beim SenderRing bzw. „Quakfaehig“ beim Quakzahler ewig umhüllt werden.

4.2.6 Adapter Pattern

Das Pattern wird als Schnittstelle für die Gans verwendet, da diese keine „quaken“-Methode besitzt, sondern über die Methode „schnattern“ kommuniziert. Um diese Verbindung zu schaffen, wird der GansAdapter, welcher das Interface Quakfaehig implementiert, eingesetzt, bei welchem die Gans über den Parameter des Konstruktors übergeben wird. In der „quaken“-Methode wird die Methode „schnattern“ der Gans ausgeführt.

5. Arbeitsdurchführung

- Herunterladen des Beispiels von O'Reilly [1]
- Erstellen eines UMLs in deutscher Sprache, welches die Darstellung der Interfaces korrekt anwendet
- Implementieren der Methode registriereBeobachter und benachrichtigeBeobachter in der Klasse Schar
    - Sonst würde die gegebene Simulation keine „quak" ausgeben
- Dokumentieren der verwendeten Patterns und Verstehen des gewählten Designs
- Generieren von Java Doc für die Implementierungsklassen aufgrund des erlangten Wissens
- Erstellen der Testklassen
- Erstellen einer „AllTests"-Klasse, die alle Testklassen durchführt, um den Code-Coverage des gesamten Programms zu überprüfen

6. Testbericht

Es wurde für jede Klasse jeweils eine Testklasse erstellt. Dabei wurden alle Methoden der jeweiligen Klasse auf eine korrekte Funktionalität geprüft.

Die Testung der Klasse EntenSimulator konnte nicht sinnvoll getestet werden, da diese Klasse die Main-Klasse des Programms darstellt. Es wären nicht genügend Methoden vorhanden, um die Einzelteile der Klasse zu testen, dadurch wurde nur der gesamte Output der Klasse geprüft.

Das Code-Coverage der implementierten Klassen liegt dabei bei 97,3 %.
Die 100 % wurden nicht erreicht, da die Main-Methode der Klasse EntenSimulator nicht getestet werden kann.

7. Quellenangaben

[1] O'Reilly (2006). Head First Design Patterns – German Example [Online]. Available at: http://examples.oreilly.de/german_examples/hfdesignpatger/ [zuletzt abgerufen am 13.12.2014]