Abgabe: Aufgabe 8 ST WS 14/15
HS Mannheim – Studiengang Informatik
Sebastian Schuler (1332256)
Thomas Kämmerling (1320565)

Aufgabe 8: Clean Code

Fragen

8.2.1)

Erklären Sie mit eigenen Worten, warum die Regel „Avoid Encodings“ sinnvoll ist.

Es ist sinnvoll weitere Codes zu vermeiden, da man es damit einfacher macht Code zu verstehen. Man vermeidet so, dass zum Verständnis des Code auch noch das Lernen einer Kodier-„Sprache“ nötig ist. Das verstehen von Code allein ist oft anspruchsvoll genug und kodierte Informationen sind eine weitere, gedänkliche Hürde, um ein Problem zu lösen. Außerdem sind Encodings oft nicht auszusprechen und man kann sich schnell vertippen.

8.2.2)

Erklären Sie mit eigenen Worten, warum die Regel „Use Unchecked Exceptions“ sinnvoll ist.

Zunächst einmal benutzen viele der modernen und weit verbreiteten Programmiersprachen gar keine checked Exceptions. Außerdem führt eine checked Exception dazu, dass man gegen das Open/Closed Principle verstößt. Durch die throw/catch-Befehle kann eine Exception über mehrere Ebenen gehen und so muss jede Ebene angefasst werden, wenn ich etwas ändert. (Änderungen auf niedrigen Leveln führen zu Änderungen auf hohen).

8.2.3)

Nennen Sie eine Regel, die Sie nicht als sinnvoll erachten und geben Sie an einem konkreten Beispiel an, warum nicht.

Teile der Regel „Avoid Encodings“ erscheinen uns nicht sinnvoll. Die Vermeidung von Hungarian Notation macht heutzutage auf jeden Fall Sinn. Jedoch finden wir es gut, Membervariablen und Interfaces zu markieren. So kann man zum Beispiel in einem Code gut erkennen, wann man Acht geben muss, dass man gerade eine Membervariable manipuliert. Beispielsweise kann dies bei Überladung von Methoden und Vererbung wichtig sein.

8.2.4)

Was fanden Sie schwierig, oder haben Sie nicht verstanden? Es mag sein, dass Sie hier nichts angeben können. Dann antworten Sie bitte mit „nichts“.

Den Code für die Aufgabe zu verstehen und zu Refaktorisieren war sehr zeitaufwendig, wenn man es gut machen wollte. Um Regeln wie „Eine Funktion, eine Aufgabe“ umzusetzen musste man sehr tief in den Code einsteigen und ihn versuchen zu verstehen. Was uns nicht ganz gelungen ist. Hier sollte dann ggf. eingeschränkt werden wie stark wir den Code verändern sollen.

8.2.5)

Beschreiben Sie, was Sie am interessantesten oder gewinnbringend fanden.

Trotz allem waren die Regeln für Clean Code und auch die Anwendung auf ein „Worst Practice“ sehr interessant

8.2.6)

Welche Anknüpfungspunkte sehen Sie zwischen diesem Stoff und dem, was Sie bereits wissen?

Es war spannend zu sehen, dass wir oft gepredigt bekommen, dass wir immer schön kommentieren sollen, es aber bei Regeln für Clean Code heißt, dass Kommentare eigentlich „Failure“, also Versagen, zeigen. Wir sollten von Anfang an also eher lernen, dass wir ausführlicheren und selbsterklärenderen Code schreiben. Aktuell lernen wir eher: Sei faul und kommentiere dann.

8.2.7)

Wie lange haben Sie gelesen, Fragen beantwortet, Aufgaben bearbeitet? Gefragt ist jeweils der Gesamtaufwand aller Gruppenmitglieder.

6 Stunden (beide zusammen).

Aufgaben

4.3.1)

Auf den folgenden Seiten finden Sie eine Anforderung, eine Testeingabe dafür (Klasse „JavaDemoFile“) und eine Implementierung der Anforderung (Klasse „LinesOfCode“).

Lesen Sie alles und überlegen Sie sich, wie Sie die Klasse „LinesOfCode“ refaktorisieren würden, um

das Programm im Sinne der gelesenen und anderer Clean Code-Regeln besser zu machen.

Erstellen Sie dafür eine Liste von Refaktorisierungsanweisungen, z.B.:

1. Variable „x“ sollte besser „y“ heißen und vom Typ „float“ sein.

2. Methode „m1“ sollte einen zweiten Parameter „p2“ vom Typ „int“ bekommen, um … .

3. Methode „m2“ liefert eigentlich … und sollte daher in „…“ umbenannt werden.

Wir haben den Code versucht in Eclipse zu refaktorisieren und sind gescheitert, da wir leider gewisse Details der Implementierung nicht verstanden haben. Hier ist jedoch unser erster Ansatz, wie wir die Methoden entsprechen aufteilen würden:

```
package working;

import java.io.BufferedReader;
import java.io.FileReader;

public class LinesOfCode {

	public int getCount(String filename){
		try{
			return countLines(filename);
		}
		catch (Exception e) {
			e.getMessage();
			return 0;
		}
	}
	private int countLines (String filename){
		int lineCounter = 0;
		if (!isComment(filename)){
			lineCouner++;
		}
		return lineCounter;
	}
	private boolean isComment (String original)
	{
		boolean isComment = false;
		return isComment;
	}
	private String deleteSpaces(String filename){
		String codelines = "";


		return codelines;
	}
}
```

Es folgt jetzt noch eine Liste an Refaktorisierungsanweisungen:

1. Es fehlt der Import des genutzten FileReaders.
2. Der erste Kommentar zur Erklärung der Methode ist überflüssigen, da Name und Signatur der Methode selbsterklärend sind.

3. Die drei if-Statements in der Methode counLines sollten ersatzlos wegfallen und durch ein unchecked Exceptionhandling ersetzt werden. (Wie im Codebeispiel oben umgesetzt)
4. Der Kommentar „Lokale Variablen“ kann entfallen.
5. Wie oben beschrieben sollten die Funktionalitäten aufgeteilt werden. Damit entfallen folgende Probleme von vorne herein:
    a. Die beiden schlecht benannten Strings cache und cacheRest. Bei Nutzung wären Namen wie „line“ und „lineWithoutSpaces“ selbsterklärender.
    b. Die drei ints zu den verschiedenen Arten von Kommentaren fallen weg. Diese sollten ohnehin wenn dann nur zwei sein, eine für „lineComment“ und eine für „blockComment“
    c. Damit verbunden ist in der vorliegenden Implementierung viel zu viel if in der Funktion countLines. Wenn man mit den Variablen comment 2 und 3 prüft ob ein Multilinecomment vorliegt muss man dies nicht danach erneut abfragen bevor man den lineCounter erhöht.
    d. Der Returnwert im catch-Block sollte nicht vorhanden sein, sondern nur die Massage zurückgeben und einen Wert 0. (Wenn man ein int als return fordert)
6. Die if-Abfrage in der Mehtode deleteSpaces ist überflüssig, da sich bei leerer Datei ohnehin ein leerer String ergibt.
7. Es reicht hier eine while-Schleife, die nur über die Länge zählt. Eine Vergrößerung des „rest“ kann man dann über eine if-Abfrage auf Absätze und Spaces ermöglichen.
8. Die Methode „strScan“ prüft, ob ein Kommentar im Code bzw. ein Indikator dafür (//, /*) vorliegt. Dies sollte man als eine boolsche Abfrage umgestalten. Diese kann dann wiederum in der neuen Methode countLines verwendet werden um Zeilen mit Kommentaren auszuschließen.
9. „needle“ ist hier auch kein besonders guter Name, da es hier um die Symbole für Kommentare geht. Jedoch fällt uns auch keine selbsterklärendere Benennung ein, da die Methode recht kompliziert umgesetzt scheint.

10. Die Benennung der Variablen in der Methode „strScan“ sollte einheitlich auf Englisch sein, also kein „längeOriginal“ sondern „lengthOriginal“.
11. Das return-Statement „return -1“ führt unserem Verständnis des Codes nach auch zu Fehlern.