ÜBUNGSZETTEL MIKROARCHITEKTUR

CODE IST ANBEI IN DER ZIP

BlueJ-Buch Klassenentwurf

Software wird oft geändert, deswegen soll sie gut wartbar sein. Um Programme wartbar zu machen und halten gibt es zwei Konzepte: Kopplung und Kohäsion.

Kopplung

Unter Kopplung versteht man die Verknüpfungen zwischen verschiedenen Einheiten in Programmen. Hängen Klassen stark voneinander ab, spricht man von einer engen Kopplung. Man versucht diese Abhängigkeiten weitgehend zu reduzieren. Geringe oder lose Kopplung ist bei Programmen wünschenswert.

Kohäsion

Kohäsion beschreibt die Anzahl und Vielfalt von Aufgaben, für die eine einzelne Einheit verantwortlich ist. Es wird versucht einen möglichst hohen Wert an Kohäsion zu erreichen. Eine hohe Kohäsion wird erzielt, wenn Klassen und Methoden nur wenige oder gar nur eine Aufgabe haben.

Refactoring

Programme die oft überarbeitet werden, werden mit der Zeit immer länger. Um die Wartbarkeit solch eines Codes zu erhöhen sollten Refactorings stattfinden. Bei einem Refactoring wird bestehender Code mit gleichbleibenden Funktionen neu geschrieben, um unnötigen Code zu entfernen und Lesbarkeit zu erhöhen.

Entwurfsmuster 1 & 2

Beziehungen zwischen Klassen sind sehr wichtig und können schnell komplex werden. Damit ein Programm mit vielen Beziehung übersichtlich aufgebaut werden kann, werden Entwurfsmuster eingesetzt. Solch ein Muster besteht aus:

- Einen Musternamen
- Einer Problembeschreibung
- Schritte zu einer Lösung (Strukturen, Teilnehmer und Partner)
- Seine Konsequenzen (Ergebnisse, Vor- und Nachteile)

Dekorierer / Decorator

Ein Dekorierer erweitert die Funktionen eines Objekts. Dieses Objekt wird vom Dekorierer umschlossen. Alle Anrufe auf das ursprüngliche Objekt werden an das Dekorierer-Objekt mit den gleichen Parametern weitergeleitet. Der Dekorierer kann so weitere Funktionen zum Objekt hinzufügen.

Singleton

Ein Singleton-Objekt stellt sicher, dass nur eine Instanz einer Klasse existiert. Diese Objekt wird von allen Klienten verwendet. Konstruktor ist privat, um externe Instanziierung zu verhindern. Zugriff erfolgt über eine statische Methode.

Fabrikmethode / Builder

Die Fabrikmethode hilft bei der Objekterzeugung. Klienten fordern über die Fabrikmethode ein Objekt eines bestimmten Interface- oder Superklassen-Typs an. Je nach Kontext trifft eine

Fabrikmethode die Wahl, ob eine Instanz einer implementieren Klasse oder eine Subklasse zurückgegeben wird.

Beobachter / Observer

Der Observer ermöglicht es, die Ansicht von einem internen Modul zu trennen. Observer definieren 1 zu n Beziehungen zwischen den Objekten. Durch den Observer informiert das beobachtete Objekt alle Beobachter über Zustandsänderungen.

Adapter

Durch einen Adapter können Interfaces, die gegenseitig inkompatibel sind gemeinsam verwendet werden. Adapter ändern ein Interface so um, wie es der Klient verwenden möchte.

SOLID UND SONSTIGE DESIGNPRINZIPIEN

S — Single responsibility principle

O — Open closed principle

L — Liskov substitution principle

I — Interface segregation principle

D — Dependency Inversion principle

Mit den vielen verschiedenen Solid-Prinzipien versucht man Code (meist OOP) zu vereinfachen und zu verbessern.

- “A class should have one and only one reason to change, meaning that a class should have only one job.”
- "Objects or entities should be open for extension, but closed for modification."
- "Let q(x) be a property provable about objects of x of type T. Then q(y) should be provable for objects y of type S where S is a subtype of T."
- "A client should never be forced to implement an interface that it doesn't use or clients shouldn't be forced to depend on methods they do not use."
- "Entities must depend on abstractions not on concretions. It states that the high level module must not depend on the low level module, but they should depend on abstractions."

KISS - Keep it simple, stupid

Code soll so einfach wie möglich gehalten werden. Je komplexer Code ist, desto anfälliger ist er für Fehler.

YAGNI - You aren't gonna need it

Code sollte nur Funktionalitäten beinhalten, die wirklich gebraucht werde. Unnötiger Code, oder Code der in Zukunft eventuell gebracht werden könnte, soll immer entfernt werden. Dies wird gemacht um die Übersichtlichkeit zu erhöhen und Fehler zu vermeiden.

DRY - Don’t repeat yourself

"Every piece of knowledge must have a single, unambiguous, authoritative representation within a system". Redundanzen sollen in allen Formen vermieden werden. Code-Duplikationen sind zu vermeiden. Bei Anpassungen des Codes müssen bei redundanten Funktionen mehrere Stellen ausgetauscht werden und dies kann schnell zu Fehlern und mehr Aufwand als nötig führen.