Karlsruher Institut für Technologie

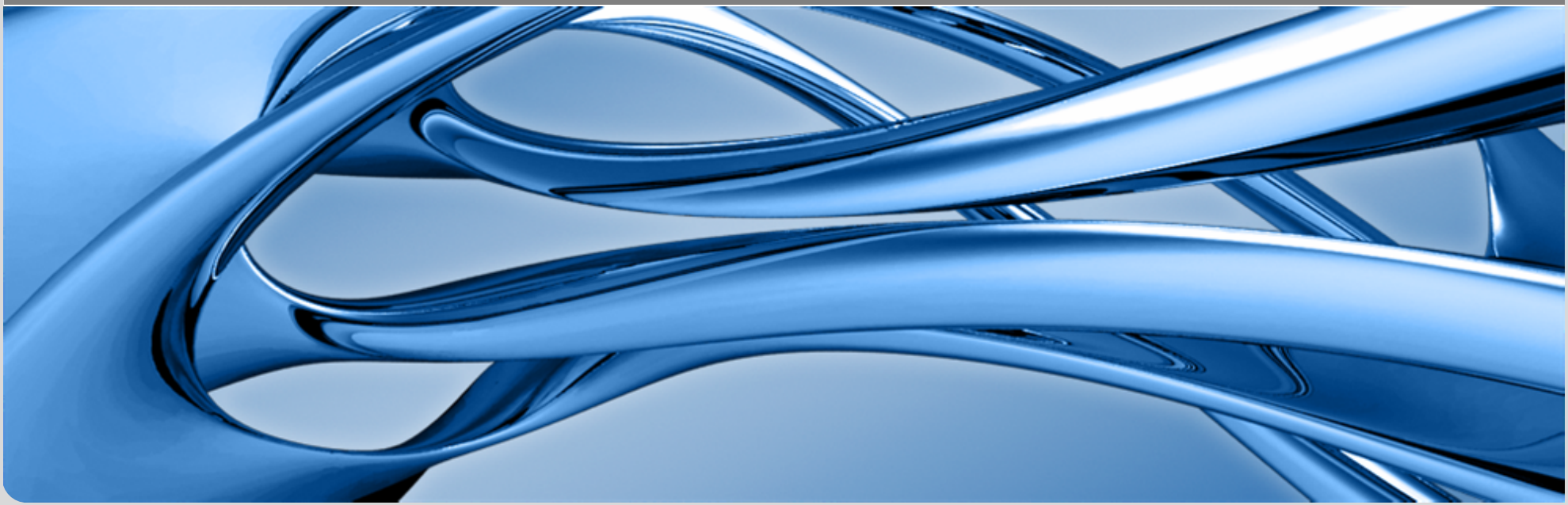
SDQ Software Design and Quality

Tutorium 12

Design-Prinzipien, Regex und Abschlussaufgaben

Christian Zielke | 5. Februar 2019

CHAIR FOR SOFTWARE DESIGN AND QUALITY



Gliederung

Abschlussaufgaben

- Anmeldezeitraum: 10.02.2019 - 20.02.2019 jeweils um 12:00 Uhr

Parsen

Frage

Wie kann eine Zeichenkette in seine semantisch bedeutsamen Teile zerteilt und seine Struktur festgestellt werden?

Lösung

- Formale Sprachen und Grammatiken verwenden.
- Top-Down Ansatz
- Bottom-Up Ansatz

Regex

- Reguläre Ausdrücke auf String anwenden: s.matches(regex)
- regex ist ein String
- Pattern sind hier nachzulesen `Java Dokumentation`
- + mindestens einmal, * beliebig oft, | oder
- Beispiel in Eclipse

Schnittstellenprogrammierung

- Wenn vorhanden (und möglich), dann sollte gegen die Schnittstelle programmiert werden
- Vorteil: Clients kennen genaue Klasse nicht
  - eigentliche Objekte und Interfaceimplementierungen austauschbar
  - losere Koppelung von Objektverbindungen
- Nachteil: Evtl. höhere Design-Komplexität

Komposition

Komposition

- Wiederverwendung durch Zusammensetzen bestehender Objekte zu einem neuem mit erweiterter Funktionalität
- Manchmal auch Aggregation
  - Teil-Ganzes-Beziehung im Gegensatz zur has-a-Beziehung

- neue Funktionalität durch Delegation an Komponenten
- Vorteil: Zugriff nur über deren Interface (Black-Box)
  - erhöht Wiederverwendbarkeit, Datenkapselung
  - weniger Abhängigkeiten, Details nicht sichtbar
  - dynamischer

Vererbung

Vererbung

- Wiederverwendung durch Erweiterung bestehender Objekte

- Vorteile: leichter; Basisklasse kann leicher erweitert oder modifiziert werden
- Nachteile
  - Schlechte Datenkapselung; White-Box-Wiederverwendung
  - Änderung in Basisklasse bedingt evtl. auch Änderungen in allen erbenden Klassen

Allgemein

- Vorausplanung ist in der Softwaretechnik wichtig
- SW wird sich ändern, daher muss sie änderbar sein
- Andere Personen müssen sich auf bestimmte Klassen und Methoden etc verlassen können
- Festlegung von Modulen, Klassen und Schnittstellen unabhängig von Implementierungsdetails
- SOLID
  - Single responsibility design
  - Open/closed principle
  - Liskov substitution principle
  - Interface segregation principle
  - Dependency inversion principle

Single Responsibility Design

- Jede Klasse sollte genau eine Verantwortung haben
- Wenn sich etwas an der Klasse ändert, müssten sonst mehrere Bereiche geändert werden
- Methoden sollten möglichst geringe Abhängigkeiten (voneinander) haben
- Klasse wird robuster
- Beispiel: Modul, dass einen Bericht kompiliert und dann ausgibt
  - Hier können sich zwei unterschiedliche Dinge ändern: Inhalt und Format
  - Daher zwei unterschiedliche Verantwortungen, die aufgeteilt werden sollten

Open/Disclosure Principle

- Module sollten sowohl offen (für Erweiterungen) als auch geschlossen (für Modifikationen) sein.
- Erweiterungen sollen somit möglich sein, ohne Schnittstelle oder Sourcecode ändern zu müssen
- Durch geeignete Vererbung realisierbar
- Gegen Schnittstellen implementieren!

Liskov Substitution Principle

- Ersetzbarkeitsprinzip
- Objekte sollen jederzeit mit Instanzen des Subtyps ersetzbar sein
- Korrektheit des Programms sollte dabei nicht gefährdet sein
- Schnittstellenmethoden müssen sich als ähnlich verhalten etc
- Alle Eigenschaften der Oberklasse müssen vorhanden sein
- zusätzliche Eigenschaften dürfen die Unterklasse spezieller machen (mit Vorsicht!)

Interface Segregation Principle

- Lose Kopplung durch Entkoppelung von Interfaces
- Interfaces in kleinere Interfaces aufteilen, um sich Anforderungen einzelner Clients anzupassen
- Sollen nur jeweils das Können, was Clients benötigen
- Clients sollen nicht dazu gezwungen werden, von Interfaces abzuhängen, die sie nicht verwenden.
- Somit müssen Änderungen auch nur in den jeweils betroffenen Clients etc abgeändert werden

Dependency Inversion Principle

- Module sollen weniger gekoppelt werden
  - Module höherer Ebenen sollen nicht von Modulen niedriger Ebenen abhängen. Beide sollten von Abstraktionen abhängen.
  - Abstraktionen sollen nicht von Details abhängen. Details sollten von Abstraktionen abhängen
- Höhere Ebene definiert Schnittstelle, niedrigere Ebene implementiert diese.
- Bsp: Im Konstruktor oder durch Setter eine Klasse übergeben, anstatt sie in der Klasse selbst zu initialisieren

Abschlussaufgabentipps

- Ausreichend Zeit einplanen
- Gewissenhaft die Aufgaben lesen (+markieren etc)
- Objektorientiert Programmieren
- Code ausreichend vor Abgabe testen (am besten eigene Testfälle schreiben!)

weitere Tipps und Ressourcen

- /r/java und andere subreddits
- Awesome Java (Liste aller möglichen Tools, Ressourcen etc)
- FindBugs Tool
- Being Productive with Eclipse
- Stackoverflow (Meistens auch unter den ersten Google Ergebnissen)

Codingtipps

- Checkstyle beachten
- Nicht zu lange Methoden, sondern Sachen auslagern
- Vor allem keine tiefen if-Verschachtelungen
- Sinnvolle Benennung von Variablen/Klassen/Methoden/Parametern etc.
- Sinnvolle OO Modellierung verwenden
  - Keine Gottklassen
  - Zuständigkeiten aufteilen
  - SOLID etc

Benutzerschnittstelle

- Schaut in die Musterlösungen der Übungsblätter.

Übungsaufgabe

Einfacher Parser

- Eingabe:
  - calculate Befehl
    zwei Integer und eine Operation aus {+, -, *, /}, z.B.: calculate 4+3
  - bracketCheck Befehl
    bracketCheck String, z.B.: bracketCheck $(te(st))$
- Ausgabe: Ergebnis der Rechnung oder true/false
- Aussagekräftige Fehlermeldungen bei falscher Eingabe
- Testklasse schreiben