Übungsblatt 5

Ausgabe: 17.01.2022 Besprechung: 24.-27.01.2022 30.5 Punkte

Bitte laden Sie die Übungsbeispiele rechtzeitig am Sonntag vor der Übungsstunde bis spätestens 23:55 als Zip Datei (`<Matrikelnummer>_<Name>_SE1-ÜB<Nummer>.zip`) in Moodle hoch und tragen Sie Ihre Kreuze in ZEUS ein. Da es in ZEUS keine halben Punkte gibt, wurde die Anzahl der Punkte für ZEUS verdoppelt → laut ZEUS können Sie für diese Übung 61 Punkte bekommen. Für die Übung selbst werden aber die am Übungsblatt angegebenen Punkte gezählt → es können maximal 30.5 Punkte erreicht werden (= Anzahl der Punkte in ZEUS / 2). Sollten Sie Fragen zur Übung haben oder Unklarheiten auftreten, nutzen Sie bitte das Moodle-Forum.

Generelle Anmerkung: Benutzen Sie für diese Übung das im Moodle bereitgestellte Projekt. Vermerken Sie bitte Ihren **Namen** und Ihre **Matrikelnummer** als Kommentar in all Ihren Klassen. Weiters werden in dieser Einheit die Tools JUnit[1] zum automatisierten Testen, Mockito[2] zum Mocken, JaCoCo[3] zur Messung der Code Coverage, und PIT[4] zur Durchführung der Mutations Analyse verwendet.

Abgabedateien

Die Zip Datei `<Matrikelnummer>_<Name>_SE1-ÜB<Nummer>` sollte folgende Dateien beinhalten:

- `5.1_Analyse.pdf`
- `5.1_Schritt_<N>_reports.*`
- `5.2_State_Chart.pdf`
- `5.3_Refactoring_Steps.pdf`
- `5.4_Refactoring_Steps.pdf`
- `5.5_Refactoring_Steps.pdf`
- `5.6_Chidamber_Kemerer.pdf`
- `project.zip`

Testing

1.1 Unit Testen/Integrations Testen/Mocking (8,5 Punkte)

Das Projekt ist in einer Schichtenarchitektur[5] implementiert. Führen Sie folgende Aufgaben durch:

[1] `http://junit.org/junit4/`, Tutorial, `http://www.vogella.com/tutorials/JUnit/article.html`
[2] `http://site.mockito.org`, Tutorial `http://www.vogella.com/tutorials/Mockito/article.html`
[3] `http://www.eclemma.org/jacoco/`
[4] `http://pitest.org`
[5] `https://de.wikipedia.org/wiki/Schichtenarchitektur`

1. Orientieren Sie sich im Projekt und machen Sie sich mit der Schichtarchitektur im Projekt vertraut. Sie können hierzu die bereits vorgestellten Techniken und Tools aus den vorangegangenen Übungen verwenden. Halten Sie Ihre Ergebnisse in einem Textdokument `4.1_Analyse.pdf` fest.

2. Analysieren Sie die Schichtenarchitektur des Projektes und identifizieren Sie die verschiedenen Schichten und wie diese miteinander kommunizieren. Halten Sie Ihre Ergebnisse in einem Textdokument `4.1_Analyse.pdf` fest.

3. Designen und schreiben Sie für die Klassen aus dem Paket `at.aau.ue5.bsp1.dao.impl` **Unit Tests**. Benutzen Sie dafür die Klassen im Test-Paket `at.aau.ue5.bsp1.dao`. **(2 Punkte, K1)**

4. Designen und schreiben Sie für die Klasse `at.aau.ue5.bsp1.service.InvoiceService Impl` **Integrations Tests**. Verwenden Sie hierzu die Klasse `at.aau.ue5.bsp1.service. InvoiceServiceImplIntegrationTest`. **(3 Punkte, K2)**

5. Designen und schreiben Sie für die Klasse `at.aau.ue5.bsp1.service.InvoiceService Impl` **Unit Tests**. Verwenden Sie hierzu die Klasse `at.aau.ue5.bsp1.service.Invoice ServiceImplUnitTest`. Verwenden Sie `Mockito`[6] um etwaige Abhängigkeiten zu mocken und Unit Tests der Klasse möglich zu machen. Verwenden Sie ebenfalls Mockito um erwartete Aufrufe etc. zu prüfen. **(3,5 Punkte, K3)**

Achten Sie dabei genau auf die Definition von Unit Tests und Integrations Tests und stellen Sie sicher, dass all Ihre Tests auch wirklich eindeutig zuzuordnen sind. Zeigen Sie mit den bereits bekannten Tools (JaCoCo und PIT) nach jedem Schritt, dass Sie die Klassen ausreichend und gut getestet haben (=erstellen Sie die Reports).

1.2 State basiertes Testen/Parameter Tests (5 Punkte)

Dieses Beispiel verwendet die Klassen des Pakets `at.aau.ue5.bsp2`. Die Klasse `at.aau.ue5. bsp2.CashMachine` bildet eine (vereinfachte) Software eines Bankomaten ab. Folgende Regeln gelten:

- Ein normaler Ablauf einer Bargeldbehebung sieht wie folgt aus: Zuerst muss der Kunde die Bankomartkarte einführen. Als zweiten Schritt muss der Kunde den PIN der Karte richtig eingeben um anschließend einen gültigen, gewünschten Betrag auszuwählen. Danach müssen das Geld und die Karte aus der Maschine genommen werden.
- Eine ungültige Karte endet mit einer 00, z.B. `12300`.
- Als PIN wird nur `0815` akzeptiert.
- Die PIN Eingabe darf maximal 5 Mal versucht werden (= 4 Wiederholungen).
- Da der Bankomat nur über Geldscheine im Wert von 100, 50 und 10 verfügt, können nur Beträge ausbezahlt werden, die ausschließlich mit diesen Geldscheinen ausgehändigt werden können. Andere Beträge sind ungültig.

[6] `http://site.mockito.org`

Führen Sie folgende Aufgaben durch:

1. Untersuchen Sie die Klasse `at.aau.ue5.bsp2.CashMachine`. Zeichnen Sie einen **State Chart** mit den Informationen die Sie in der Klasse finden. Kennzeichnen Sie jeden Status und jeden Übergang (`4.2_State_Chart.pdf`). **(1 Punkt, K4)**

2. Implementieren Sie Tests um den Standard Pfad (wie oben beschrieben) mit diversen Wertekombinationen zu testen. Implementieren Sie diese Tests in Form von **parametrisierten Tests**[7] von JUnit. Verwenden Sie dafür die Klasse `at.aau.ue5.bsp2.CashMachine ParameterTest`. **(2 Punkte, K5)**

3. Erstellen Sie mittels **Random Testing** Testfälle für die Klasse `at.aau.ue5.bsp2.Cash Machine`. Verwenden Sie dafür die Klasse `at.aau.ue5.bsp2.CashMachineRandomTest`. Sie können auch hier auf die Funktionalität der parametrisierten Tests von JUnit zurück greifen. **(1 Punkt, K6)**

4. Implementieren Sie Tests für **State-based Testing**. Versuchen Sie alle Pfade abzudecken und denken Sie hierbei auch an Fehlerfälle und Abweichungen vom Standardpfad. Verwenden Sie hierzu die Klasse `at.aau.ue5.bsp2.CashMachineStateTest`. **(1 Punkt, K7)**

Refactoring

1.3 Refactoring Smells 1 (5 Punkte)

Gegeben ist die Klasse `at.aau.ue5.bsp3.Person` aus dem Package `at.aau.ue5.bsp3`. Diese Klasse ist sehr schlecht implementiert und beinhaltet einige Code Smells. Ihre Aufgabe besteht darin, soviele **Refactoringschritte** wie notwendig **durchzuführen**, um die **Klasse zu verbessern** (Sie dürfen die Refactoring Tools der IDE verwenden, müssen jedoch jeden Schritt selbst durchführen und erklären können). Ein Schritt beinhaltet das Festhalten des Smells, die Durchführung des Refactorings, und ein abschließendes Commit in Git. Einige Anmerkungen/Hinweise zum Zweck der Klasse:

- Es gibt verschiedene Arten von Personen, die von dieser Klasse verwaltet werden.
- Jede Person hat einen Vornamen, einen Nachnamen, sowie ein Alter.
- Eine Person ist immer genau einem Beruf (`beruf`) zugeordnet. Es gibt keine Personen ohne Beruf. Die möglichen Berufe für dieses Beispiel sind: Entwickler, Architekt, und Tester.
- Ein Entwickler hat eine bevorzugte Sprache (`bevorzugteProgrammierSprache`), eine Entwicklungsumgebung (`ide`, kann nur *ECLIPSE*, *INTELLIJ* oder *NANO* sein) und eine Variable, die anzeigt, ob der Entwickler mit Datenbanken entwickeln kann (`datenbanken`).
- Ein Architekt hat ein Tätigkeitsfeld (`feld`, kann ENTERPRISE oder APPLIKATION sein).
- Ein Tester hat ein bevorzugtes Test Framework (`bevorzugtesTestFramework`)
- Jede Person hat eine Methode `getJobBeschreibung()`, die den Titel des Jobs retourniert.

Die Seiten `http://www.tutego.de/java/refactoring/catalog/index.html` und `https://sourcemaking.com/refactoring` bieten weiters Hinweise und Hilfen zum Thema Refactoring.

[7] `https://www.baeldung.com/parameterized-tests-junit-5`

1.4 Refactoring Smells 2 (5 Punkte)

Gegeben sind die Klassen aus dem Package `at.aau.ue5.bsp4`. Führen Sie die **notwendigen Refactoring-Schritte** (speziell in der Klasse SmellyClass) durch und verbessern Sie die Klassen (Sie dürfen die Refactoring Tools der IDE verwenden, müssen jedoch jeden Schritt selbst durchführen und erklären können). Ein Schritt beinhaltet das Festhalten des Smells, die Durchführung des Refactorings, und ein abschließendes Commit in Git. Achten Sie diesmal **besonders auf Smells** wie z.B. Duplicated Code, Long Method oder Feature Envy.

1.5 Refactoring von Tests (4 Punkte)

Gegeben sind die **Tests** aus dem Package `at.aau.ue5.bsp5`. Die Tests beinhalten einige **Test Smells**. Führen Sie **notwendigen Refactoring Schritte** durch und verbessern Sie die Tests. Ein Schritt beinhaltet das Festhalten des Smells, die Durchführung des Refactorings, und ein abschließendes Commit in Git. Als Lektüre kann das Buch „xUnit Test Patterns“ und die dazugehörige Webseite dienen.[8] Achten Sie besonders auf Testprinzipien wie in der Vorlesung vorgestellt.

Code Metriken

1.6 Metriken nach Chidamber und Kemerer (3 Punkte)

Gegeben ist das Klassendiagramm aus Abbildung 1.

Bestimmen Sie die folgenden Code Metriken und erklären Sie deren Bedeutung und Berechnung:

a) DIT(A), DIT(C), DIT(G)

b) NOC(B), NOC(C), NOC(D)

c) CBO(D), CBO(J)

d) RFC(D) - Gehen Sie davon aus, dass die Methoden sowohl `mJ1(m,n)`, als auch `mJ2(u,v)` durch eine der Methoden in `D` aufgerufen wird.

e) LCOM(D) - Gehen Sie davon aus, dass Parameter mit dem gleichen Namen in unterschiedlichen Methoden dieselben Instanzvariablen nutzen.

f) WMC(D) - Folgende Komplexitäten für die Methoden sind gegeben:
V(m1) = 2, V(m2) = 3, V(m3) = 4, V(m4) = 2,
V(mL1) = 3, V(mL2) = 2, V(mL3) = 1, V(mL4) = 5,
V(mJ1) = 2, V(mJ2) = 3, V(mH1) = 2, V(mH2) = 3,
V(mA) = 2, V(mB) = 3, V(mE) = 2, V(mG) = 3, V(mK) = 2, V(mI) = 3.

[8] `http://xunitpatterns.com/Test\%20Smells.html`

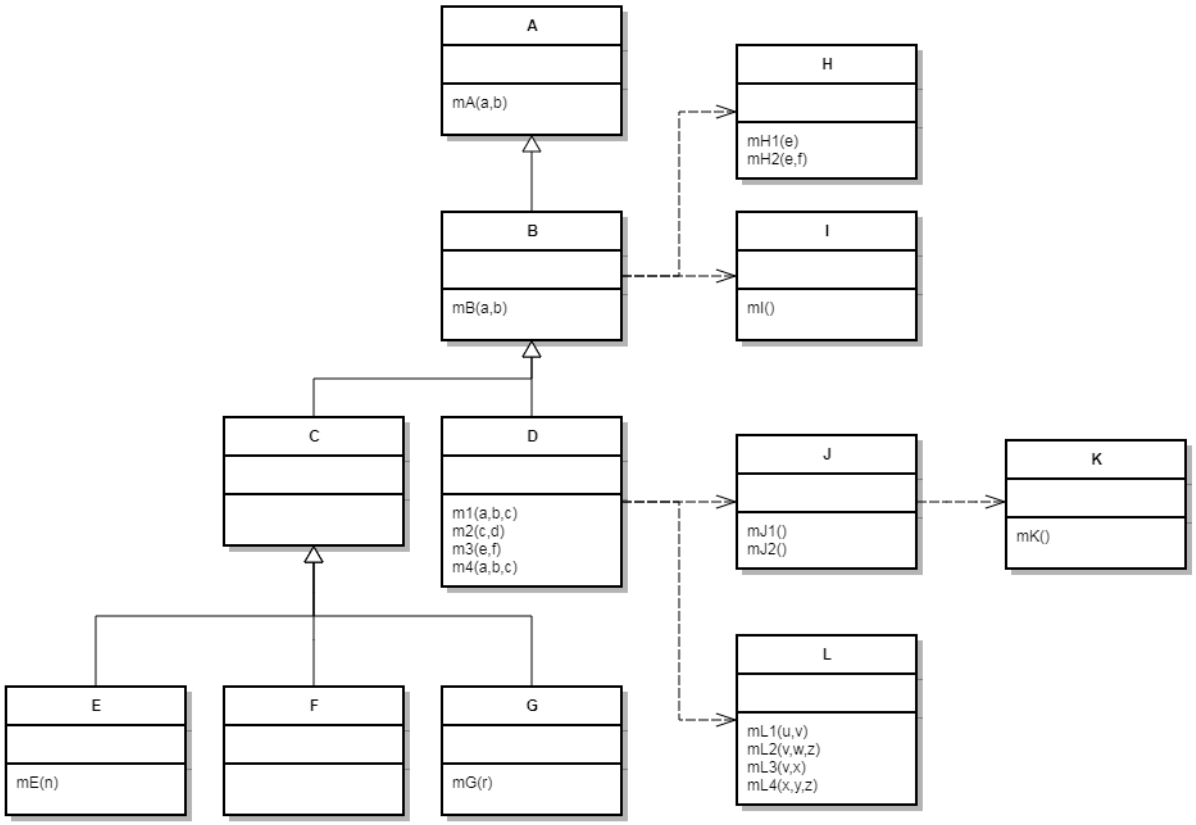

Abbildung 1: Klassendiagramm