Programming Principles

In diesem Dokument wird der Programmcode der geschriebenen Anwendung BlackJack auf die in der Vorlesung angesprochenen Programming Principles SOLID, GRASP und DRY analysiert.

SOLID

Single Responsibility Principle

Das Single Responsibility Principle besagt, dass seine Klasse nur eine Zuständigkeit und nur eine Aufgabe übernehmen soll. Eine Klasse soll nur eine Ursache oder einen Grund haben sich zu ändern. Durch das Refactoring, welches im Zuge dieses Projektes getan wurde, wurde das Erfüllen des Single Responsibility Principle verbessert. Die vorher bestehende Klasse BlackJackGame hatte sehr viele unterschiedliche Aufgaben. Sie hat das Spiel dirigiert, das Spiel ausgewertet, das Spielerguthaben verwaltet und sich auch um Passwort Änderungen gekümmert. Durch das Refactoring wurden diese Aufgaben in Klassen verschoben, welche einer Aufgabe dienen. Der EvaluationHandler ist nur zum Auswerten der Spielsituation da. Der MoneyHandler hat die einzige Aufgabe das Geld und Guthaben des Spielers zu verändern. Der CardHandler dient nur zum Austeilen von Karten an den Spieler und den Croupier.

Open Closed Principle

Durch das Open Closed Principle soll der Programmcode offen für Erweiterungen aber geschlossen für Änderungen sein. Dies kann durch Abstraktion und Vererbung von Klassen geleistet werden. Das Open Closed Principle ist in dem Programmcode von BlackJack nicht sehr stark eingebunden. An vielen Stellen gibt es Switch oder If/Else Anweisungen, die bei neuen Anforderungen erweitert oder abgeändert werden müssen. An manchen Stellen ist das Open Closed Principle jedoch auffindbar. Zum Beispiel muss bei der Einführung einer neuen Spielerklasse oder Person für die Anwendung nicht in die Klasse Person eingegriffen werden. Hier kann, genau wie in der Klasse Player, von Person vererbt werden und Person erweitert werden. Des Weiteren lässt sich das Open Closed Principle in der GameController Klasse und den dazugehörigen Listenern finden. Sollte ein Listener aus der GameController Klasse eine Erweiterung benötigen, so ist es einfach den Listener zu vererben und eigene erweiterte Logik einzubauen. Dieser erweiterte Listener könnte dann ohne Änderungen anstelle des normalen Listeners verwendet werden.

Liskov Substitution Principle

Das Liskov Substitution Principle kann in der Person und Player Klassen gesehen werden. Die Player Klasse erbt von Person und verhält sich wie eine Person, nur mit erweiterten Funktionen. Dies ist auch in abschnitten des Programmcodes sichtbar, wo in einer Methode als Parameter eine Person erwartet wird und die Methode bei einer Person oder bei einem Player dieselbe Auswirkung hat. Beispiele dafür wären die Methoden CardHandler.givePlayingCard() oder CardHandler.checkForAce().

Interface Segregation Principle

In der BlackJack Anwendung ist der Anwender nur von den Oberflächen Klassen abhängig, welche sehr leichte Klassen mit wenig Funktionen des Codes beinhaltet sind. Die Schweren und funktioslastigen Klassen sind in der Controller- und Modelschicht der Anwendung zu finden. Mit diesen kommt der Anwender jedoch nicht direkt in Kontakt. Der Controller leitet die benötigten Daten und Funktionen an die Oberfläche weiter ohne großes Wissen der Oberfläche über diese Funktionen und Daten.

Dependency Inversion Principle

Das Dependency Inversion Principle ist schwer im Programmcode der BlackJack Anwendung zu finden. Es gibt keine abstrakten Klassen und keine großen Hierarchien, die abgeleitet werden, wodurch das Problem, welches das Dependency Inversion Principle versucht zu lösen, nicht aufkommt.

GRASP

Low Coupling

Low Coupling wurde in BlackJack nicht sehr gut berücksichtig und umgesetzt. Es gibt viele Orte und Klassen die hohe Kopplung miteinander haben. Zum Beispiel sind die GameView, CardHandler, EvaluationHandler und MoneyHandler sehr stark mit dem GameController gekoppelt und verbunden. Objekte diese Klassen sind als Field im GameController vorhanden und werden oft über den GameController verwendet.

High Cohesion

Ein Ansatz von High Cohesion kann in der Player Klasse gesehen werden. Hier wurde die Klasse Money eingeführt, um Geld des Spielers darzustellen. Jedoch wäre hier auch mehr Potenzial für Kohäsion. Es wäre hier auch weiterhin möglich gewesen eine Klasse Password einzufügen, welche die Felder iteration, salt und password vereint und somit ein Passwort darstellt.

DRY

Don’t Repeat Yourself wurde durch das durchgeführte Refactoring verstärkt beachtet und Dubletten wurden aus dem Code entfernt, um dem Programming Principle stärker zu folgen. Zum Beispiel wurde in mehreren Klassen Code

geschrieben, um die vom Croupier gezogenen Karten mit einer Verzögerung auf der Oberfläche darzustellen. Um diesen doppelten Code zu entfernen, wurde die Methode GameController.drawCroupierCards() eingefügt, welche die Aufgabe zum Darstellen der Croupier Karten mit Verzögerung implementiert und nun anstelle von eigenem Code in den Klassen verwendet wird. Des Weiteren dienen Klassen wie der CardHandler dem DRY Principle. Die Funktionen aus CardHandler könnten auch jeweils an den benutzten Stellen selbst geschrieben werden. Um jedoch Wiederholung und Dopplung zu vermeiden können Funktionen wie CardHandler.givePlayingCard() ausgeführt werden.