From Zulu To Hero

Anas Arodake, Jonas Becker, Julian Exner, Oliver Gorczyca, Bohdan Shevchenko *

March 2021

Inhaltsverzeichnis

*Keine finale Teilnahme am Projekt.

1 Analyse

— Phasenverantwortlicher: *Jonas Becker*
— Stichtag Expose: *23.03.2021*
— Stichtag Lastenheft: *06.04.2021*

1.1 Expose

1.1.1 Lösungsidee zur Umsetzung der geforderten Eigenschaften

Zur Umsetzung der geforderten Eigenschaften wird ein Discord-Chatbot konzipiert und implementiert, welcher über definierte Textbefehle steuerbar ist. Der Chatbot handelt automatisch und mit festgelegten Parametern die Ware (Buchstaben), er kann sie also kaufen, halten und verkaufen. Der Bot verwaltet außerdem ein Konto zur Erfassung der Währung (Punkte). Sowohl die SEG, als auch andere Bots sowie Menschen können als Handelspartner dienen.

Der Bot erfasst die Transaktionen in einem Log erfasst und gibt wichtige Ereignisse in einem Chat-Channel aus. Aus dem Log generiert der Bot zudem eine Statistik (Umsatz, Gewinn, Verlust,...). Der Bot gibt auf Anfrage alle laufenden Kaufs- und Verkaufsvorgönge sowie seinen Kontostand und Warenbestand als Direktnachricht aus.

1.1.2 Identifikation von Teilproblemen

Buchstaben kaufen, halten, verkaufen Der Bot wertet die Auktionen im Chat aus und bietet nach bestimmten Kriterien mittels eines Befehls mit. Im Besitz befindlice Buchstaben werden für die Dauer einer Session in einer geeigneten Speicherstruktur gehalten. Auf Anfrage greift ab und verkauft diese möglicherweise.

Ware zusammenstellen und liefern Der Bot gleicht Verkaufsanfragen mit seinem Inventar ab und entscheidet nach bestimmten Kriterien, ob er verkauft, erst die fehlenden Buchstaben ersteigern muss und dann verkauft, oder die Anfrage ablehnt. Inventar und Wallet werden anschließend entsprechend aktualisiert.

Punkte halten und überweisen Der Bot besitzt ein Wallet zur Erfassung seiner Kaufkraft in der Währung "Punkte". Er kann Umsätze erfassen und für die Dauer einer Session speichern. Zudem

kann er Punkte an andere Teilnehmer überweisen oder von ihnen erhalten und seinen Kontostand entsprechend anpassen.

Käufer- und Verkäuferdaten erfassen Der Bot speichert Informationen über Verkäufe und Einkäufe. Mit einem Befehl kann mindestens Datum und Uhrzeit sowie IDs der Teilnehmer einer Transaktion aller Transaktionen in dieser Session ausgegeben werden. Die Daten werden in einem standartisierten Datenformat festgehalten.

Profit, Absatz und Umsatz verwalten Der Bot loggt getätigte Interaktionen. Aus diesem Log wird ein Objekt generiert, in dem Umsätze, Absätze und Profite filterbar gespeichert werden.

Handel und Interaktionen ausgeben Der Bot kann wichtige Ereignisse aus dem Log auch in einem Discord-Channel ausgeben. Die Ausgabe wird durch das textuelle Interface getriggert.

User Interface Der Bot kann Nachrichten in Discord-Channels versenden und den dortigen Nachrichtenverlauf lesen. Der Bot reagiert auf bestimmte gelesene Nachrichten. Der Bot kann zur Lesebestätigung per Emoji auf Nachrichten reagieren.

Interaktion durch Reaktionsemojis Der Bot kann auf eine Kaufanfrage mit einem Emoji reagieren, wenn er den Kauf ablehnt, weil er nicht alle angefragten Buchstaben besitzt oder ein Fehler aufgetreten ist. Ebenso kann der Bot von Käufern als Bestätigung oder Ablehnung gesendete Emojis interpretieren.

Help Funktion Der Bot gibt auf an, wie mit ihm zu interagieren ist.

1.1.3 Aufwandsschätzung

Expose erstellen 1 Personentage

Lastenheft inkl. UML 3 Personentage

Präsentation 1/2 Personentage

1.2 Lastenheft

LF10 **Prozess:** Ware kaufen
Akteur: Der SEG Bot
Beschreibung: Der Bot kann auf Ware(Buchstaben) bieten. Das soll über den Chat automatisch geschehen. Wenn die Auktion gewonnen wird, soll Inventar und Kontostand entsprechend angepasst werden.

LF11 **Prozess:** Ware verkaufen
Akteur: Der SEG Bot, Handelspartner (User oder Bot)
Beschreibung: Der Bot kann Ware gebündelt oder einzeln in einem Warenkorb verkaufen. Die Ware wird im Chat von anderen Bots angefragt und über diesen verkauft/ausgegeben. (siehe auch LF20f.)

LF12 **Prozess:** Ware halten
Akteur: —
Beschreibung: Der Bot speichert die erhaltene Ware in einem Inventar, kann dieses ausgeben und und bei Ein- oder Verkauf verändern.

LF20 **Prozess:** Warenkorb erstellen
Akteur: Ein Handelspartner
Beschreibung: Der Bot kann eine Buchstabenanfrage als String annehmen und sie mit dem Inventar vergleichen. Wenn die Zutaten für den String im Inventar existieren wird ein Warenkorb erstellt und der Warenkorbwert errechnet. Falls wenige Buchstaben fehlen, wird versucht diese mit erhöhter Priorität zu kaufen und der letzte Schritt wird durchgeführt. Falls mehrere Buchstaben fehlen wird die Anfrage abgelehnt.

LF21 **Prozess:** Ware liefern
Akteur: Ein Handelspartner
Beschreibung: Ist ein Warenkorb erstellt, wird dessen Wert mit dem Gebot des Kunden verglichen. Ist das Gebot höher, wird ein Verkauf zu diesem Preis anfgeboten. Ist der Warenkorbwert geringfügig höher, wird ein Gegenangebot gestellt. Ist die Diskrepanz zu hoch, wird das Verkaufsangebot verworfen. Bestätigt der Kunde ein Verkaufsangebot oder ein Gegenangebot, reagiert der Bot darauf erneut und führt die Transaktion durch, er passt den Inhalt des Inventars sowie des Wallet entsprechend an.

LF30 **Prozess:** Punkte halten
Akteur: —
Beschreibung: Der Bot besitzt eine Wallet zur Erfassung seiner Kaufkraft in der Währung "Punkte". Er kann Einnahmen und Ausgaben erfassen und für die Dauer einer Session speichern.

LF31 **Prozess:** Punkte überweisen
Akteur: Ein Handelspartner
Beschreibung: Die Überweisung von Punkten funktioniert als bidirektionale Transaktion. Die Punkte werden der Richtung entsprechend dem Konto hinzugefügt oder abgezogen.

LF40 **Prozess:** Käufer- und Verkäuferdaten verwalten
Akteur: Ein Admin
Beschreibung: Ein Admin gibt einen entsprechenden Befehl ein und kann auf die gesamte Einkaufs- und Verkaufshistorie zugreifen.

Lastenheft (Fortsetzung)

LF50 **Prozess:** Daten für Umsatzanalyse lesen und formatieren
Akteur: —
Beschreibung: Der Bot speichert alle von ihm getätigten und von Handelspartner bestätigten Interaktionen (Einkäufe von der SEG, Handel mit Konkurrenzbots, Verkauf an Kunden) für die Dauer einer Session in einem maschinenlesbaren Format.

LF51 **Prozess:** Umsatzanalyse durchführen
Akteur: Ein Admin
Beschreibung: Der Bot wertet die vorliegenden Daten regelmäßig statistisch aus und gibt diese Ergebnisse in menschenlesbarer Form an einem geeigneten Ort bekannt. Hieraus kann ein Vorgehen für zukünftige Auktionen abgeleitet werden.

LF60 **Prozess:** Handel ausgeben
Akteur: Diverse Handelspartner
Beschreibung: Sämtliche Handelsoperationen werden durch den Bot per Textausgabe bekanntgegeben.

LF61 **Prozess:** Interaktion ausgeben
Akteur: Diverse Handelspartner
Beschreibung: Erfolgreiche Interaktionen zwischen dem Bot und einem User werden über eine Textausgabe im Channel bestätigt.

LF70 **Prozess:** Reaktion per Emoji
Akteur: (ggf. nur menschliche) Handelspartner
Beschreibung: Der Bot reagiert auf eine Kaufanfrage mit einem :thumbsdown: Emoji, wenn er den Kauf ablehnt. Wenn der Bot einen Kaufpreis liefert, kann der Kunde mit einem :thumbsup: Emoji den Preis akzeptieren oder mit einem :thumbsdown: Emoji den Preis ablehnen.

LF80 **Prozess:** Ausgabe möglicher Botinteraktionen
Akteur: (ggf. nur menschliche) Handelspartner
Beschreibung: Der Bot ist in der Lage einem Nutzer auf Anfrage mit !help seine Interaktionsmöglichkeiten auszugeben. Diese werden sowohl beispielhaft, als auch schematisch dargestellt.

1.3 UML-Diagramm

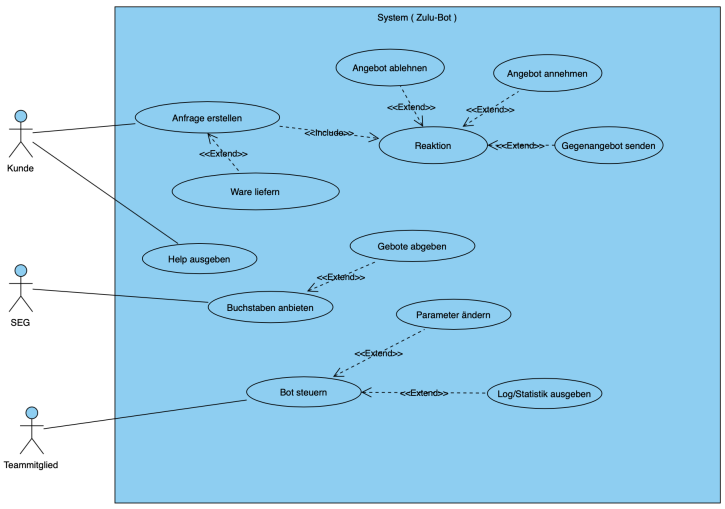

Abbildung 1: Visualisierung der Anwendungsfälle

2 Systementwurf

— Phasenverantwortlicher: *Julian Exner*
— Stichtag Grobentwurf: *20.04.2021*
— Stichtag Feinentwurf: *11.05.2021*

Übersicht In diesem Kapitel wird das Vorgehen in der Erarbeitung eines gemeinsamen Entwurfes beschrieben. Dies hat sich als sehr zentraler Bestandteil eines Softwareprojektes mit einem Team dieser Größe herausgestellt, da es sehr wichtig ist, eine gemeinsame Vision des finalen Produktes und des Weges dorthin zu haben. Divergierende Visionen resultieren im späteren Projektverlauf in Konflikte, Redundanzen und ineffizienten Implementierungen.

2.1 Grobentwurf

OOA-Diagramm

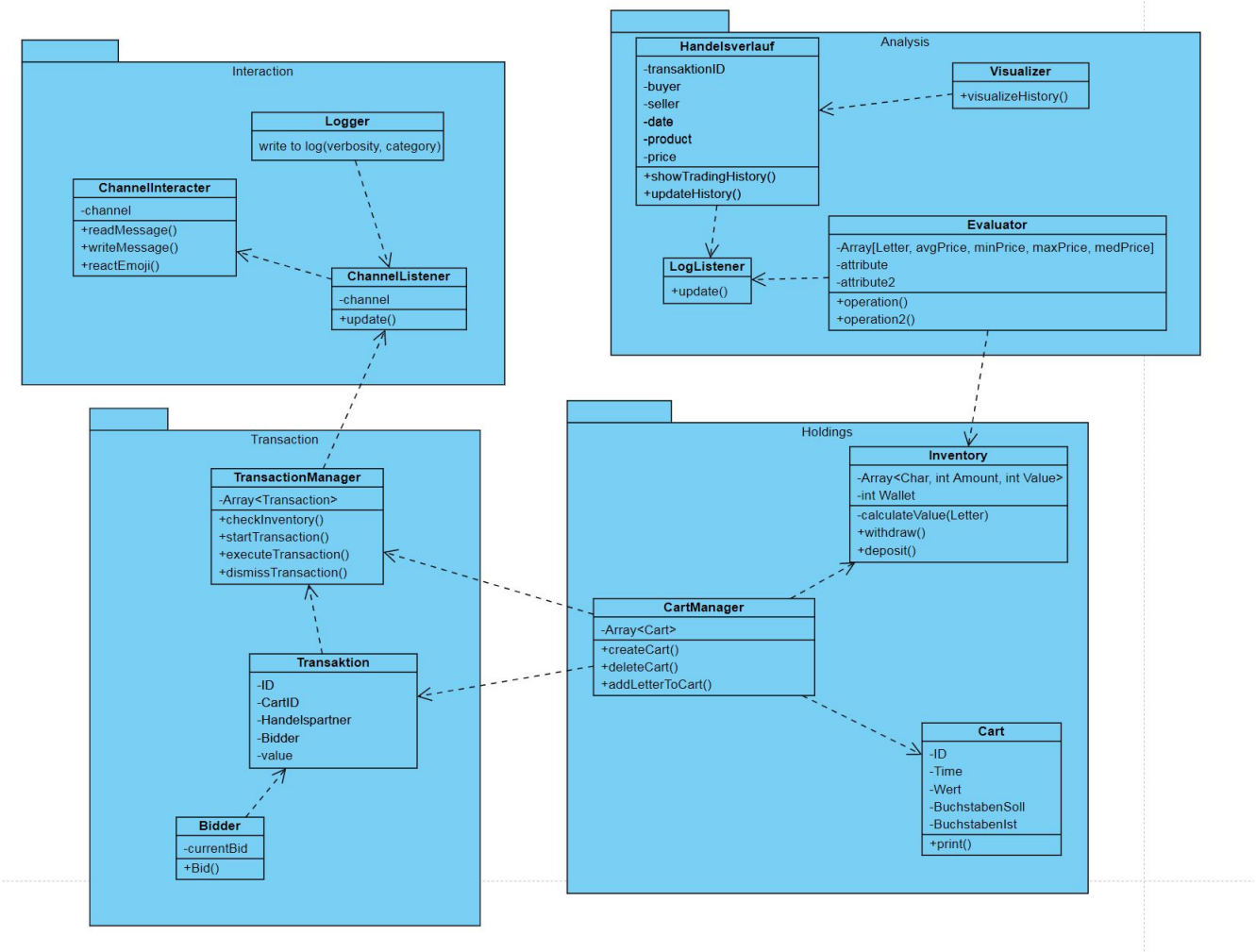

Abbildung 2: OOA-Diagramm

2.1.1 Ansatz

Zunächst haben wir uns in einem gemeinsamen Brainstorm auf eine grobe Paketstruktur verständigt. Es wurden diverse Entwürfe diskutiert, abschließend wurde sich mit dem Besusstsein, dass es sich um einen sehr frühen, dynamischen Entwurf handelt auf eine Struktur, die aus einem Interaktionspaket, Analysepaket, Warenverwaltungspaket, Handelspaket und Warenkorbpaket besteht, geeinigt. Daraufhin haben wir die Pakete auf die Gruppenmitglieder verteilt und in Einzelarbeit nach OOA erste Klassenstrukturen entwickelt.

2.1.2 Vorgehensweise OOA

Wir haben uns für die OMT-Methode (Datenorientierte Vorgehensweise) entschieden. Sie erschien für unser Projekt am sinnvollsten, da wir die Daten die wir verarbeiten werden bereits kennen. Intuitiv wurden von uns Klassen in den uns zugewiesenen Paketen erstellt und zentrale Memberfunktionen beschrieben, zunächst ohne Argumente und Membervariablen genauer zu betrachten. Anschließend haben wir uns die so zustandegekommenen Grobentwurfspakete gegenseitig vorgestellt und im Zuge dessen bereits viel verändert und präzisiert. Zuletzt haben wir gemeinsam Schnittstellen zwischen den Paketen bestimmt, mit dem Ziel zwischen zwei gegebenen Paketen eine einzelne, klare Schnittstelle zu definieren.

2.1.3 Package-Übersicht

Interaction Package Das Interaction Package umfasst drei Klassen. Der *ChannelListener* erfasst über den *ChannelInteracter* alle Events, die in den jeweiligen Channels publiziert werden. Die *Logger-Klasse* zeichnet spezifische Events auf, fügt sie in eine Log-Datei ein und weist ihnen eine Kategorie für eine spätere Filtermöglichkeit zu. Der *ChannelInteracter* stellt die dritte Klasse dar. Er dient als Schnittstelle für andere Klassen und ermöglicht den lesenden sowie den schreibenden Zugriff auf die jeweiligen Channel.

Transaction Package Das Transaktion Package umfasst drei Klassen. Der *TransactionManager* wird aktiv, sobald er durch den *ChannelListener* getriggert wird. Er erstellt bei Kaufs- und Verkaufsprozessen je eine neue Transaktion und beendet diese nach Abschluss wieder. Die *Bidder-Klasse* führt die einzelnen Bietvorgänge bis zu einem spezifischen Bieterlimit aus.

Analysis Package Das Analyse Package umfasst vier Klassen. Der *LogListener* wird durch den *ChannelListener* / oder *Logger* getriggert. Mithilfe dieses *LogListeners* erfasst der *Handelsverlauf* alle getätigten Transaktionen und persistiert relevante Daten pro Transaktion in einer passenden Speicherstruktur. Diese Daten können mit dem *Visualizer* graphisch/textuell dargestellt werden. Der *Evaluator* weist den einzelnen Buchstaben einen Wert zu. Hierbei wird die Häufigkeit der Buchstaben in der deutschen Sprache, sowie die im *Handelsverlauf* gespeicherten Daten berücksichtigt.

Cart Package Eine der zwei zugehörigen Klassen ist der *CartManager*. Dieser erstellt einzelne *Carts* für angefragte Waren, verwaltet diesen und löscht/archiviert ihn, sobald die Transaktion beendet wurde. Der Einzelne *Cart* enthält die geforderten Buchstaben und besitzt einen aggregierten Wert als initialen Verkaufspreis.

Holdings Package Das Holdings Package. Dieses Package dient der Datenpersistenz pro Session. Sowohl die einzelen Buchstaben im Besitz des Zulu Bots, als auch die Finanzkraft werden hier verwaltet.

2.1.4 Aufwandsschätzung

Paketdefinition 2 Personentage

Paketausarbeitung 4 Personentage

2.2 Feinentwurf

überarbeitetes OOA-Diagramm

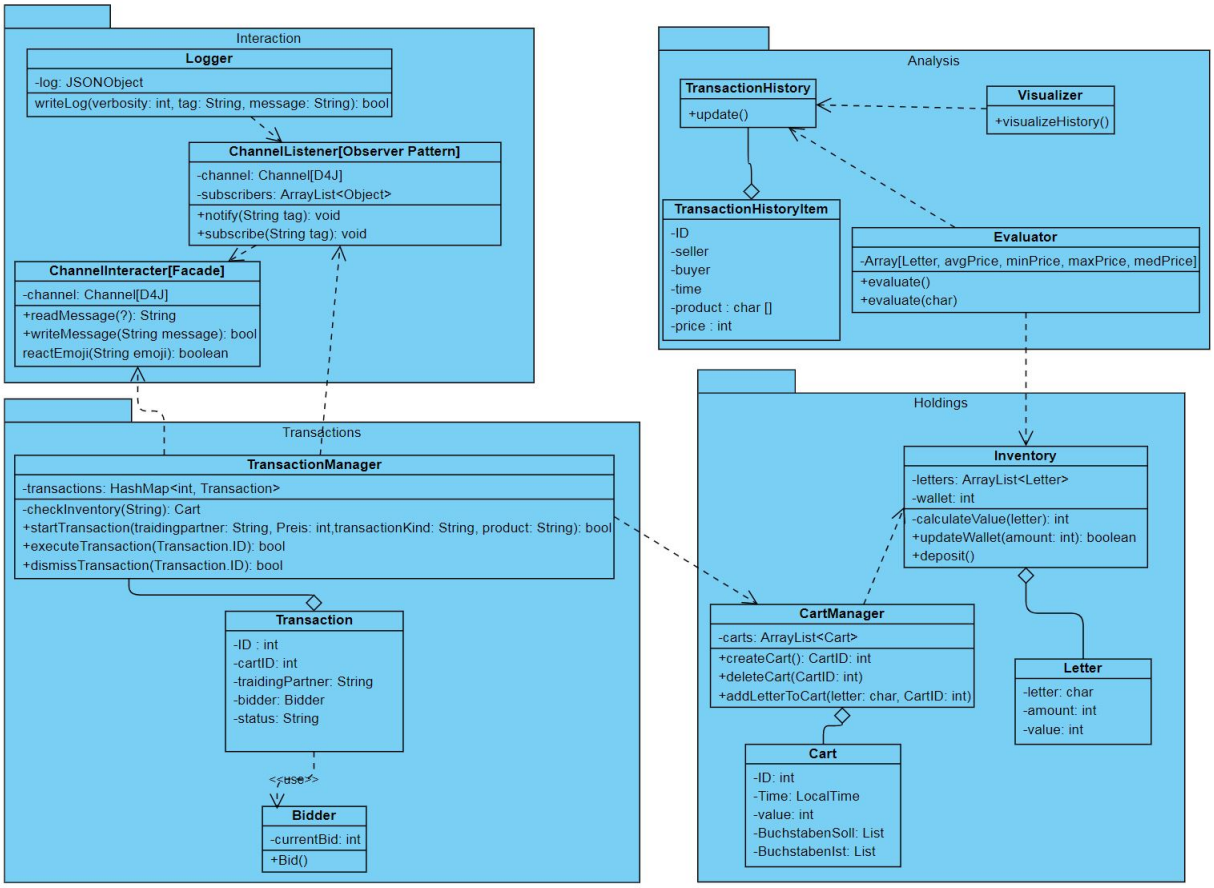

Abbildung 3: überarbeitetes OOA-Diagramm

2.2.1 Arbeitsablauf

Zunächst wurden in Gruppen von zwei bis drei Teilnehmern die einzelnen Pakete besprochen. Ziel war es, Abhängigkeiten sowie zentrale Funktionen und Variablen der einzelnen Klassen zu erarbeiten. Anschließend wurde über die mögliche Implementation von Designpattern sowie striktere Kapselung der Pakete im Plenum diskutiert. Dies geschah rekursiv, was sich als nicht besonders zeiteffizient herausstellte, allerdings für ein einheitliches Verständnis des Arbeitsplans sorgte.

2.2.2 Themenübersicht

Designpattern Früh im Entwurfsprozess wurde klar, dass die reaktive Natur eines Chat-Bot eine hohe Synergie mit dem Observer-Designpattern bietet. Der Entschluss, dieses im *ChannelListener* anwenden zu wollen, war daher naheliegend. Dank der Implementation des Entwurfsmusters kann die Klasse sehr effizient ihre diversen Subscriber benachrichtigen, die auf diesem Wege bereits verarbeitete Daten basierend auf den Commands im Discord-Chat erhalten und darauf reagieren. Auch die *TransactionHistory* sollte dieses Pattern einsetzen, um den *Evaluator*, der aus ihr, sowie statischen Daten, den Wert, von dem sich die maximalen Einkaufsgebote und minimalen Verkaufspreise des ZuluBots ableiteten, sowie den *Visualizer* über eine aktualisierte Datenlage zu benachrichtigen. Weiterhin war der *ChannelInteractor* ohne aktive Entscheidung eine Implementation des Entwurfsmusters *Facade* geworden. Es erschien uns sehr

effizient, die Implementation von Discord4J auf diese eine Klasse zu limitieren und eine sehr schlanke IO-Funktionen im verbleibenden Code zu ermöglichen.

Kapselung In großer Überschneidung zum Themenkomplex Designpattern wurde hier der Einsatz des *Singleton*-Pattern kontrovers diskutiert. Der einfachere Zugriff und Schutz vor Instanzdoppelungen in den Manager-Klassen, aber auch beim *Logger* und *Evaluator* stand die Verletzung der OOP-Prinzipien und der Verlust der Kapselung gegenüber. Es wurde zunächst kein finaler Beschluss gefasst, eine nachträgliche Implementation des Singleton-Entwurfsmusters wurde jedoch als unproblematisch empfunden. Es wurde außerdem überlegt, ob der Logger in einen Stream schreiben kann, aus dem die TransactionHistory liest, sodass keine direkte Abhängigkeit der Klassen voneinander entsteht. Dies wurde aus Zeitgründen sowie fehlender Kenntnis derartiger Implementierungen nicht weiter verfolgt.

2.2.3 Finaler Entwurf

Am Ende der Entwurfsphase wurde der Entschluss gefasst, das Klassendiagramm (Siehe Abb. 3) nicht weiterzuentwickeln. Abschließende Versuche einer detaillierteren Ausarbeitung der weniger präzise beschriebenen Pakete führten mehrmals zu einem hin und her zwischen mehreren konkurrierenden und wahrscheinlich qualitativ fast ebenbürtigen Implementationsansätzen. Es wurde sichergestellt, dass alle verbliebenen Gruppenmitglieder ein klares und übereinstimmendes mentales Bild von der Projektstruktur haben und damit begonnen, das Kanban-Board mit Aufträgen zur Implementation der Klassen zu füllen.

2.2.4 Aufwandsschätzung

Klassenausarbeitung 2 Personentage

Überarbeitung 4 Personentage

2.3 Reflektion

Während der Entwurfsphase wirkte unser eigener Ansatz auf uns selbst ineffizient. Das rekursive Besprechen der Pakete in verschiedenen Besetzungen führte oftmals zu kaum wahrnehmbaren Fortschritt. Zusammen mit den technischen Problemen, die größtenteils durch Schwächen in dem Tool Visual Paradigm verursacht wurden (Klassen wurden nach gleichzeitiger Bearbeitung, aber gelegentlich auch ohne offensichtlichen Grund unbearbeitbar und teilweise unlöschbar, ohne kostenpflichtige Lizenz konnten nur zwei Teilnehmer gleichzeitig bearbeiten) sorgte dies auch für einen merklichen Abfall der Motivation. Retrospektiv betrachtet wirkt unser diskussionsorientierter Ansatz durchaus wertvoll, da er dafür sorgte, dass eine einheitliche Zielvorstellung entstand und viel Einigkeit über die Vorgehensweise in der Entwicklungsphase herrschte. Daher behält auch in diesem Kontext die Weißheit von Dwight D. Eisenhower ihre Validität - Ï have always found that plans are useless, but planning is indispensable. denn die finale Implementation hat nahezu jegliche Reminiszenz zum überarbeiteten OOA-Diagramm verloren.

3 Entwickung

— Phasenverantwortlicher: *Anas Arodake*
— Stichtag: *01.06.2021*

Übersicht Das Kapitel Entwicklung beschäftigt sich mit der tatsächlichen Umsetzung des zuvor konzipierten Bots. Hier werden die verwendeten Werkzeuge und Frameworks vorgestellt und das Vorgehen des Teams beschrieben und mit dem angestrebten »Test Driven Development« in Bezug gesetzt. Abschließend

beschäftigt sich dieses Kapitel mit einer Reflektion über die Entwicklungsarbeit und leitet Erkenntnisse und Wünsche für mögliche Folgeprojekte ab.

3.1 Werkzeuge und Frameworks

3.1.1 Discord4J

Für die Entwicklung des Bots wird das Discord4J Framework verwendet. Hierbei handelt es sich um ein in Java verfasstes reaktives Interface, welches eine Abstraktionsschicht der offiziellen Discord API darstellt. Durch die Nutzung dieses Frameworks wurde sich zudem auf die Nutzung der Programmiersprache Java als technische Basis verständigt.

3.1.2 Visual Paradigm

Wie durch die Auftraggeber vorgeschlagen, verwendeten wir das Modellierungs-Tool Visual Paradigm, um unsere Klassendiagramme im Grobentwurf 2.1 und Feinentwurf 2.2 zu erstellen. Der Export der Klassendiagramme in Java-Klassen-Rümpfe soll die Grundlage der Entwicklung darstellen.

3.1.3 Git

Um eine ortsunabhängige, asynchrone, konfliktarme und ausfallsichere Arbeitsweise zu ermöglichen wird Git als Versionskontrollsystem eingesetzt. Durch die Einhaltung des »Feature Branch Workflows« ist es möglich parallel an verschiedenen Funktionalitäten des Bots zu arbeiten, ohne die Arbeit eines anderen Teammitgliedes unkontrolliert zu beeinflussen. Hierbei wird die in Jira gegebene Möglichkeit genutzt, Branches direkt auf Basis eines Tickets erstellen zu können. Nach dem Auschecken und bearbeiten des Tickets wird der fertige Code auf das remote repository gepushed und anschließend ein Pull-Request erstellt, in dem ein anderes Teammitglied den Code überprüfen und abschließend mergen kann (Siehe 4.3.2).

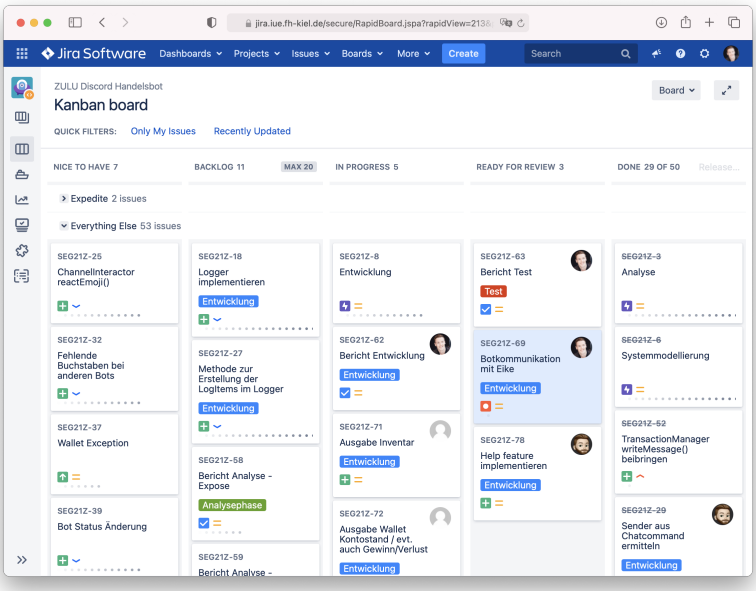

Abbildung 4: Screenshot des verwendeten Kanban-Boards

3.1.4 Jira

Um diese verteilte Arbeitsweise zu ermöglichen, die den verschiedenen zeitlichen Kapazitäten der Teammitglieder gerecht wird, nutzt das Team Jira als Projektmanagement-Tool. Jira bietet die Möglichkeit Projekte nach dem Vorbild des agilen Projektmanagements zu organisieren. Zentrales Verwaltungswerkzeug ist hierbei das Kanban-Board (siehe Abb. 4).

Hier sind alle Aufgaben einsehbar, die erledigt werden müssen, damit die einzelnen Milestones erreicht und das Projekt als Ganzes erfolgreich abeschlossen werden kann. Diese »Single Source of Truth« ermöglicht ein hohes Maß an Transparenz. Die einzelnen Aufgaben werden in Form von Tickets gesammelt und zu Beginn im gemeinsamen Product Backlog – einer initialen Sammelstelle aller Arbeitspakete - abgelegt. Beispielsweise werden die im Lastenheft festgelegten Lastenfunktionen in kleine, möglichst atomare Arbeitspakete aufgeteilt und in der Form von Tickets dem Board hinzugefügt. Diese Tickets bieten eine kurze Beschreibung der Funktionalität und sind somit von allen Mitgliedern des Teams einsehbar und jederzeit umsetzbar.

Als Beispiel dient die zu implementierende Funktion *LF80 Ausgabe möglicher Botinteraktionen* des Lastenheftes. Diese Lastenfunktion hat bereits eine atomare Größe erreicht, so dass die Erstellung eines einzelnen Tickets ausreicht. Die Erstellung erfolgt unter Angabe eines beschreibenden Textes, der es jedem Teammitglied ermöglicht, die Arbeit am Ticket aufzunehmen (siehe Abb. 5).

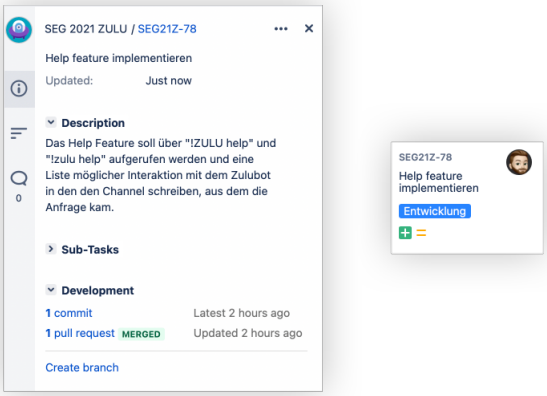

Abbildung 5: Erstellung eines Tickets auf Basis der *Lastenheftfunktion LF80*

3.2 Test Driven Development als Ansatz und kritische Bewertung

»Test Driven Development« bezeichnet ein Software-Entwicklungs-Vorgehen, bei dem zuerst die Testklassen auf Basis der funktionalen Anforderungen geschrieben werden und anschließend die jeweiligen Klassen implementiert und schrittweise erweitert werden, so dass sie durch die zuvor geschriebenen Tests validiert werden. Das Vorgehen resultiert in präzisen Klassen, da diese lediglich soweit programmiert werden, bis sie eine weitere funktionale Anforderung erfüllen. Das Verfahren erhöht jedoch den Arbeitsaufwand für alle Klassen, weshalb die Anwendung sehr ressourcen- bzw. kostenintensiv ist. Test Driven Development befähigt den Programmierer die funktionalen Anforderungen an ein Produkt präzise zu erfüllen. Hierfür ist jedoch die Voraussetzung, dass alle Anforderungen und Ihre Implementierung von Beginn an bekannt sind und präzise formuliert vorliegen. So stellt es sich beispielsweise die Realisierung eines Taschenrechner und seiner Rechenfunktionen durch Test Driven Development als relativ einfach dar, die Überprüfung der Funktionsweise eines Observer-Patterns als weitaus schwieriger.

Dem Umstand geschuldet, dass die Entwicklung des Zulu Bots sowohl unbekannte Frameworks, als auch fortgeschrittene OOP Funktionalitäten und Design-Patterns nutzt, war es dem Team nicht möglich

dieses Vorgehendmodell durchgängig anzuwenden. Test Driven Development schränkt die Kreativität des Programmierers stark ein und ist somit bei der Umsetzung von Software-Projekten, welche nicht zu Beginn präzise ausformuliert sind, schwierig anzuwenden. Da die Art und Weise, wie gewisse Funktionalitäten implementiert werden könnten, zu Beginn unbekannt ist, stellt es sich als besonder schwierig dar, Tests zu formulieren.

Das Vorgehen wurde für die Klassen CartManager und Inventory des ZuluBots den in ersten Entwürfen angewendet aber später verworfen, nachdem sich herausstellte, dass zu viel Zeit in das Schreiben der Tests investiert werden musste, ohne das produktiver Code entstanden war.

3.3 Reflektion

3.3.1 Visual Paradigm und Code Generierung

Bei der Generierung der Java-Klassen durch Visual Paradigm stellte sich schnell Ernüchterung ein. Durch teilweise unpräzise formulierte Felder und Methodennamen schlug der Export zunächst fehl. Nachdem die Defintionsfehler behoben waren, gelang der Export technisch. Die Nutzbarkeit des generierten Codes war trotzdem eingeschränkt, da das Klassendiagramme nicht einwandfrei geplant und formuliert waren, sodass am Ende diverse Felder und Methoden umgeschrieben, ergänzt oder entfernt werden mussten. Dies brachte uns zu der Erkenntnis, dass die Klassenmodellierung eine sehr präzise ausgearbeitet sein muss, damit eine Code-Generierung die Entwicklungsphase erleichtert. Hierfür muss das Projekt in einer sehr frühen Phase in seiner Ganzehit ausformuliert und modelliert werden. Dazu waren wir zu dem Zeitpunkt nicht in der Lage.

3.3.2 Jira als Projektmanagementtool

Die Nutzung von Jira als Projektmanagementtool für die Softwareentwicklung wurde von allen Teammitgliedern als sehr positiv wahrgenommen. Die Bedienung ist hervorragend und die Integration mit Bitbucket erleichtert organisatorische Abläufe, wie das korrekte Erstellen und Benennen von Feature Branches. Hierfür würden wir uns eine stärkere Nutzung in hochschulischen Projekten wünschen, sofern der organisatorische Overhead gerechtfertigt ist.

3.3.3 Arbeit mit Ticketingsystemen

Obwohl allen die Mechanik der Ticketverwaltung gefallen hat, entstand die Erkenntnis, dass die Festlegung des Umfanges einzelner Tickets viel Übung benötigt. Da wir aufgrund mangelnder Erfahrung Aufwände mehrmals falsch abgeschätzt haben, sind teilweise zu große Tickets entstanden. Diese haben anschließend in der Bearbeitung zu zeitlichen Engpässen geführt.

3.3.4 Test Driven Development

Wie bereits im Unterkapitel 3.2 ausgeführt, scheiterten wir an der Anwendung des Modells. Hier wäre eine weitere kritische Auseinandersetzung im Rahmen einer Vorlesung wünschenswert, um vor Projektbeginn die Voraussetzungen für eine Entwicklung gemäß »TDD« abschätzen zu können. Zudem wären Demonstrationen dieser Herangehensweise mit Bezug zu kommplexen Funktioen und Klassen wünschenswert.

3.3.5 Personelle Lage

Durch geringes Engagement und den letzendlichen Wegfall eines Teammitgliedes fehlte gegenüber anfänglichen Aufwandsschätzungen ein erhebliches Maß an Personentagen. Darüber hinaus wurde ein Teammitglied zu Beginn des Projektes Vater. Dies schränkte die personellen Ressourcen in der Analyse- und Entwurfsphase weiter ein.

3.3.6 Eigendynamik der Entwicklung durch mangelhafte Planung

Durch den Umstand, dass einige Funktionalitäten nicht so zu implementieren waren, wie es im OOA Diagramm geplant war, ergaben sich während der Entwicklungsphase einige Änderungen am Konzept. So führte beispielsweise der Wegfall des CartManagers zum Anwachsen des TransactionManagers und die damit einhergehende Abhängigkeit vom selbigen. Dies beeinträchtigte die Parallelisierung der Arbeit und führte zu Engpässen. Ausserdem hatten wir in der Entwurfsphase keine hinreichende Kenntnis über die Implementierung der Design-Patterns und so entstanden während der Entwicklung weitere Abweichungen vom ursprünglich angedachten Konzept. Tiefere Vorkenntnisse über Design-Patterns und ihre Anwendung hätte uns hier sicherlich geholfen, präziser planen zu können.

3.3.7 Aufwandsschätzung

Entwicklung 40 Personentage

4 Testen

— Phasenverantwortlicher: *Jonas Becker*
— Stichtag: *01.06.2021*

4.1 Inhalt

Das folgende Kapitel beschäftigt sich mit der Implementation des Testvorgehens während der gemeinschaftlichen Entwicklung des ZuluBots. Das Testvorgehen dient einerseits der allgemeinen Qualitätssicherung des Sourcecodes, bspw. durch Garantie der Einhaltung genereller Code Conventions, zum anderen der Stärkung des Produktverständnis des Bots als Summe seiner Funktionen. Die Zusammenarbeit verschiedener Teammitglieder mit individuellen Arbeitsweisen und Ideen birgt die Gefahr, dass sich eine heterogene Code-Qualität entwickelt. Zudem ensteht während der Entwicklung einzelner Module spezifisches Wissen, welches für das Nachvollziehen der Funktionsweisen essentiell ist. Um sicherzustellen, dass alle Teammitglieder das benötigte Wissen besitzen und wenig individuelles Domänenwissen existiert, erfolgen Review-Sessions im Rahmen der Testphase. Nachfolgend wird zuerst die vorgeschlagene Herangehensweise des Auftraggebers und sich ergebende praktische Schwierigkeiten dargestellt. Anschließend wird die gewählte Vorgehensweise dargelegt und anhand von Beispiel illustriert.

4.2 Vorgeschlagenes Vorgehen

Das angedachte Vorgehen sieht das Testen einzelner Module durch ein Peer-Review vor. Hierbei findet ein gemeinsamer Termin der beiden Reviewer mit dem Programmierer des entsprechenden Moduls statt. Beide Reviewer überprüfen nacheinander den Sourcecode des entsprechenden Moduls und haben durch die Anwesenheit des verantwortlichen Programmierers die Möglichkeit offene Fragen zu kären und eventuelle Verbesserungen anzumerken. Um einen geregelten Arbeitsablauf gewährleisten zu können, wird ein Plan erstellt, der die Termine des gemeinsamen Code-Reviews festsetzt. Während des Termins werden Anmerkungen und Änderungsvorschläge in einem Dokument festgehalten und anschließend in neuen Arbeitspaketen zusammmengefasst. Gegebenenfalls wird ein Folgetermin für ein Review der Änderungen festgesetzt.

4.3 Bewertung der Vorgehensweise und alternative Implementation

4.3.1 Bewertung

Grundsätzlich sind die Ziele der Phase wichtig für die erfolgreiche und harmonische Zusammenarbeit der Teammitglieder sowie ein funktional hochwertiges und qualitativ homogenes Produkt. Durch das

beschriebene Vorgehen der Terminfindung, der verpflichtenden Teilnehmerzusammensetzung von drei Personen sowie der Verwendung von Checklisten und Ergebnisprotokollen entsteht eine rigide Routine. Diese erzeugt einen der Projektgröße unangemessenen Verwaltungsaufwand und beeinflusst die Entwicklungsgeschwindigkeit negativ. In der Konsequenz dieser Erkenntnis wurde, basierend auf den selben Projektmanagement-Zielen und unter Nutzung der zur Verfügung stehenden technischen Mittel, ein abweichendes Vorgehen definiert und angewendet.

4.3.2 Zulu Testvorgehen

Wie bereits beschrieben nutzt nutzt das Team Jira von Atlassian, um den Entwicklungsprozess für alle Teammitglieder sichtbar abbilden und ein hohes Maß an Transparenz gewährleisten zu können. In Kombination mit dem git-basierten Feature-Branch-Workflow wird ein konsistener Review-Prozess ermöglicht. Folgendes Vorgehen hat sich im Team etabliert:

— Fertigstellung des Features und Push auf den Feature Branch.
— Erstellen eines Pull Requests, um den Feature Branch in den Develop Branch zu mergen.
— Digitales Review-Meeting durch den Programmierer und mindestens einen Reviewer zur Validierung der Codequalität
— Sofern die Freigabe des Features erteilt wird, erfolgt der Merge.
— Bei nicht erteilter Freigabe erfolgt die gemeinsame Erstellung eines Bug-Tickets in Jira durch die am Review beteiligten Team-Mitglieder.
— Nach Bearbeitung des Bug-Tickets wird der Ablauf wiederholt, bis das Feature die Freigabe erhält.

4.4 Reflektion

Durch das angewendete Vorgehen waren wir in der Lage, für ein gemeinsames Verständnis des Produktes zu sorgen und folglich auch zusammenhängende Arbeitspakete aufzuteilen. Die Flexibilität der Review-Meetings bezüglich der Teilnehmer wurde dem Umstand gerecht, dass alle Teammitglieder individuelle außerhochschulische Verpflichtungen haben. So konnte ein Termin auch stattfinden, wenn nur ein Reviewer teilnehmen konnte und der Entwicklungsprozess wurde nicht aufgehalten. Durch Review-unabhängige Zusatztermine konnte der gemeinsame Wissensstand zur Laufzeit der Entwicklung wieder harmonisiert werden und die Mitglieder konnten produktiv arbeiten.

4.4.1 Aufwandsschätzung

Review 10 Personentage

5 Zusammenfassung

— Phasenverantwortlicher: —
— Stichtag: *08.06.2021*

5.1 Aufwandsanalyse

Die Aufwandseinschätzungen waren zum Teil unrealistisch bzw. stellten sich als nicht präzise dar. Dies ist auf die Tatsache der fehlenden Entwicklungserfahrung bei komplexeren Softwareprojekten seitens des Teams zurückzuführen.

5.2 Selbsteinschätzung

Es ist ein funktionierender Bot entstanden, der die Zielsetzung im Testumfeld erfüllt und viele architektonisch, objektorientiert und inhaltlich gute Ansätze beinhaltet.

5.3 Reflektion

Das Projekt hat den Teammitgliedern viel Spaß gemacht und es fand ein kontinuierlicher Entwicklungs- und Lernprozess statt. Auch das strukturierte Arbeiten mit Git und Jira stellte sich als bereichernde Erfahrung heraus. Teilweise gute Softwansätze hätten durch geeignetere Designpatterns und eine effektivere Nutzung der verschiedenen Java-Bibliotheken noch weiter optimiert werden können. Hier fand bereits während des Projektes ein allgemeiner Wissensgewinn durch Selbststudium und Austausch statt. Mit einer breiteren Wissensbasis hätte das Team bereits in der Konzeptionsphase die Module und Packages präziser formulieren können. Dies hätte auch zur einer besseren Code-Generation in Visual Paradigm geführt. Während der Entwicklung kam es vor, dass Bug-Fixes oder auch kleinere Funktionalitäten in verschiedenen Feature-Branches implementiert wurden. Durch eine strengere Einhaltung des Ticket-Workflow und transparentere Kommunikation hätte man dies verhindern können.

5.4 Ausblick

Bei einer Fortführung des Projektes oder einer größeren Kapazität an Personentagen gäbe es verschiedene Funktionalitäten, die das Team implementieren und Überarbeiten würde. Hierzu zählt das Refactoring des ChannelInteractors. In der aktuellen Fassung übernimmt dieser sowohl die Schnittstelle zur Discord API, als auch die Zusammenstellung und Validierung der einzelnen Messages zur internen Kommunikation. Diese Funktionalitäten können in mehrere Klassen aufgeteilt werden. Die dadurch entstehende Modulariät mit getrennten Verantwortlichkeiten würde auch eine bessere Parallelbearbeitung durch das Team ermöglichen. So kommt der Validierung eintreffender Nachrichten aus dem Chat eine wichtige Rolle zu, da auf dieser Grundlage die EventItems entstehen, welche intern allen Subjekten des Observer-Patterns die nötigen Daten bereitstellen. Sobald EventItems unvollständig erstellt werden, da beispielsweise ein unbekanntes String-Pattern im Chat die korrekte Befüllung nicht zulässt, können Laufzeitfehler entstehen. Die Implementation der Validierung findet gegenwärtig im ChannelInteracter statt, somit hat diese Klasse mehrere Verantwortungsbereiche. Dies widerspricht den Prinzipien der OOP.

In der aktuellen Version des Bots müssen Anfragen, auf deren Basis der Bot ein Angebot erstellt wird, durch den kaufenden aktiv bestätigt oder abgelehnt werden. Sofern keine Bestätigung erfolgt, sondern beispielsweise eine erneute Anfrage gesendet wird, bleibt die Transaktion im ZuluBot bestehen. Hier würden wir ein Feature implementieren, welches die Teilnehmer- und Zeitbasierte Löschung offener Transaktionen vornehmen kann.

Eine weitere spannende Funktionalität wäre die Erstellung eines Gegenangebotes auf Grundlage der zur Verfügung stehenden Ressourcen. So könnteder Bot auf die Anfrage des Wortes »SUPER« bei fehlenden Buchstaben 'R' und zwei vorhandener Buchstaben 'P' ein Gegenanbot über das Wort »SUPPE« erstellen.

Darüber hinaus wäre die im OOA Diagramm angedachte Implementierung eines sinnvollen Loggings eine weitere Verbesserung des Produktes. Aus kapazitären Gründen und dem Wegfall einer Arbeitskraft, konnte der Logger nicht mehr implementiert wurden.