

SEG Abgabebericht

Modul SEG 2021 - Software Engineering
Fachhochschule Kiel

Team bestehend aus:

Joris Willrodt, Rene Voß, Ninnias Bieler, Micha Wewers, Mira Pautz, Marvin Ottersberg



INHALTSVERZEICHNIS

INHALTSVERZEICHNIS	1
1. Einleitung	2
2. Lastenheft	3
2.1. Zielbestimmung	3
2.2. Produkteinsatz	3
2.3. Produktübersicht	4
2.4. Produktfunktionen	4
2.5. Produktdaten	5
2.6. Produktleistungen	6
2.7. Qualitätsanforderungen	6
2.8. Ergänzungen	6
3. Entwurf	7
3.1. Grobentwurf	7
3.2. Feinentwurf	8
4. Entwicklung	9
4.1 Vorgehen in der Entwicklungsphase	9
4.2 Implementierung	9
4.3 Eigene Features	10
4.4 Herausforderungen	11
4.5 Gantt-Chart	13
5. Testen	14
5.1 Test-Driven-Development	14
5.2 Codereview mit Checkliste	15
6. Fazit	16
7. Anhang	18
7.1 Lastenheft	18
7.2 Entwurf	19
7.3 Testen	23

1. Einleitung

Joris Willrodt ist der erste Phasenverantwortliche.

Als Semesterprojekt wurde ein eigener Discord-Bot, namentlich SCPF, entwickelt. Dieser kann mit Kunden und anderen Bots handeln. Für den Kauf von Buchstaben werden verschiedene Strategien integriert, die ein bestimmtes Kaufverhalten auslösen. Es soll Prioritätsstufen bei Buchstaben geben. Aufgrund dieser Einteilung kauft unser Bot automatisch ein und hält ein Repertoire an Buchstaben vor. Zusätzlich hat SCPF eine "eigene" Persönlichkeit, die durch Reaktionen auf Käufe anderer Bots widerspiegelt wird. Zudem bekommt der Käufer eines Wortes nach einer erfolgreichen Transaktion ein Kauf-Zertifikat.

Bei der Umsetzung dieser Projektidee gibt es verschiedene Teilprobleme, die gelöst werden müssen. Die Grundanforderungen werden wie folgt bearbeitet:

a. Speichern

Der Bot soll unterschiedliche Arten von Daten in einem Inventar speichern. Abhängig von der Dauer der Speicherung müssen passende Möglichkeiten (z.B. in einer CSV-Datei) gewählt werden. Es ergibt Sinn alle Informationen zu den einzelnen Käufen und Verkäufen zu speichern. Für bestimmte Daten (z.B. aktuelle Waren, Handelsdaten) müssen evtl. schnellere Speichermöglichkeiten (z.B. JSON) gewählt werden.

b. Interaktionen

Der Discord-Bot interagiert über den Chat mit drei verschiedenen Akteursgruppen. Diese werden gebildet aus den Kunden, dem SEG-Bot und den Bots anderer Studierender. Hierbei sollen nicht nur einzelne Buchstaben, sondern sogar ganze Wörter so wirtschaftlich effizient wie möglich untereinander gehandelt werden. Es wird eine standardisierte Kommunikation zwischen den Akteuren benötigt, zu der ggf. eigene Chat-Kommandos definiert werden müssen.

c. Ausgabe

Die Ausgabe von aktuellen Statusinformationen des Bots soll mittels Textnachrichten im Chat geschehen. Eine Reaktion des Kunden wäre z.B. mit Emotes/ GUI denkbar.

Bei der Realisierung der Projektidee gibt es folgende Teilprobleme (Tabelle 1):

TEILPROBLEM	LÖSUNG
Bot überschreitet das Punktelimit des Warenkorbs.	Durch geschickte Strategie verhindern, geforderte Punkte für Warenkorb entsprechend hoch setzen.
unterschiedliche Kaufstrategien	Beobachtung des Marktes und Reaktion auf Angebot und Nachfrage
eigene Persönlichkeit des Bots	authentische Sprüche bei der Interaktion mit dem Kunden
Priorisierung der Buchstaben	Aufgrund der Buchstabenhäufigkeit in der deutschen Sprache
grafisches Kauf-Zertifikat ausgeben	dynamische Erstellung einer png-Datei mit den Kaufinformationen (Wort, Name,...)

Tabelle 1: Teilprobleme

Es wird von gesamt 8 Mann-Tagen für die erste Phase ausgegangen. Bei der Festlegung wurde sich am Arbeitsaufwand des Exposé orientiert.

2. Lastenheft

2.1. Zielbestimmung

Die Software agiert eigenständig als aktiver Teilnehmer auf einem simulierten Handelsplatz und interagiert sowohl mit automatischen Geboten als auch allen anderen Teilnehmern um ein bestimmtes Ziel zu erreichen.

2.2. Produkteinsatz

Die Software wird als Teil eines virtuellen Handelsplatzes verwendet. Sie soll auf automatisch generierte Angebote einer Auktion bieten und mit einzelnen Teilnehmern individuelle Transaktionen durchführen. All dies soll auf einer definierten Strategie basieren. Während der Benutzung greifen Studierende, Dozenten und andere Bots auf die Software zu und interagieren mit ihr. Im späteren Verlauf soll sie so eigenständig wie möglich sein.

2.3. Produktübersicht

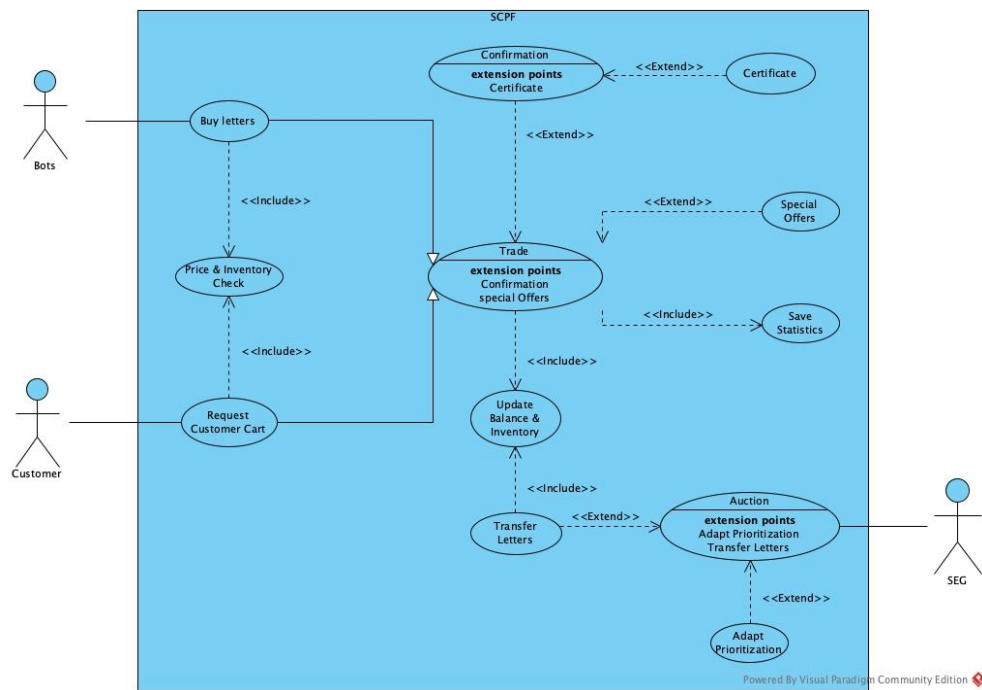


Abbildung 1: Use-Case-Diagramm

2.4. Produktfunktionen

Die Produktfunktionen nach einem ausführlichen Schema befinden sich im Anhang 7.1 auf Seite 18. In Tabelle 2 sind diese in Kurzform formuliert:

/LF10 /	Geschäftsprozess: Request Customer Cart Akteur: Customer Beschreibung: Ein Kunde wünscht ein Wort zu kaufen. Wenn SCPF das Wort vorrätig hat, werden Worte und Punkte ausgetauscht.
/LF20 /	Geschäftsprozess: Buy Letters Akteure: Bots. Kurzbeschreibung: Die Bots können sich jeweils Buchstaben gegen Punkte im Tausch anbieten
/LF30 /	Geschäftsprozess: Auction Akteure: SEG. Beschreibung: SCPF kauft Buchstaben für Punkte beim SEG-Auktionshaus nach eigener Strategie. Der Höchstbietende gewinnt.

/LF40 /	Geschäftsprozess: Adapt Prioritization Akteure: SEG. Kurzbeschreibung: Anpassen der Kaufstrategie, wenn SCPF immer überboten wird.
/LF50 /	Geschäftsprozess: Price & Inventory Check Akteure: Bots. Customer. Kurzbeschreibung: Abgleich, ob genug Ware und Punkte vorhanden sind.
/LF60 /	Geschäftsprozess: Confirmation & Certificate Akteure: Bots. Customer. Kurzbeschreibung: Käufer erhält eine Bestätigung in Form eines Zertifikats.
/LF70 /	Geschäftsprozess: Update Balance & Inventory Akteure: SEG. Bots. Customer. Kurzbeschreibung: Inventar und Punktestand updaten.
/LF80 /	Geschäftsprozess: Save Statistics Akteure: SEG. Bots. Customer. Kurzbeschreibung: Transaktionsdaten und Umsatz werden gespeichert.
/LF90 /	Geschäftsprozess: Special Offers Akteure: Bots. Customer. Kurzbeschreibung: Rabattaktionen für vordefinierte Fälle, z.B. Kauf mehrerer Buchstaben.

Tabelle 2: Produktfunktionen

2.5. Produktdaten

Wichtige Produktdaten (Tabelle 3):

/LD10 /	Botdaten: ID, ... (max. Daten von 3 weiteren Bots)
/LD20 /	Strategie: Handel aufzeichnen: SCPF Handel: letzten 300 Handelsvorgänge weitere Bots: letzten 300 Handelsvorgänge Daten: -Angebote -Gebote -erfolgreiche Kaufvorgänge

/LD30 /	Inventar (Punkte, Buchstaben)
------------	--------------------------------------

Tabelle 3: Produktdaten

2.6. Produktleistungen

Was gibt es für Bedingungen, die auf jeden Fall gegeben sein müssen (Tabelle 4)?

/LL10/	Der Verkauf von Wörtern an einen Kunden soll innerhalb von 60s abgeschlossen sein.
/LL20/	Bot-Angebote müssen innerhalb von 15s angenommen/abgelehnt werden.
/LL30/	Angenommene Verkäufe/Ankäufe sollten innerhalb von 10s vom Anbieter bestätigt werden.
/LL40/	Bei der Kommunikation zwischen Bot-Bot und Bot-SEG müssen die gemeinsamen Chatbefehle beachtet werden.
/LL50/	Das Kaufzertifikat sollte innerhalb von 5s nach dem Kauf per Nachricht an den Käufer übermittelt werden.

Tabelle 4: Produktleistungen

2.7. Qualitätsanforderungen

Folgende Anforderungen an die Qualität muss der Bot erfüllen (Tabelle 5):

<i>Produktqualität</i>	<i>sehr gut</i>	<i>gut</i>	<i>normal</i>	<i>nicht relevant</i>
<i>Benutzerfreundlichkeit</i>	x			
<i>Zuverlässigkeit</i>		x		
<i>Funktionalität</i>		x		
<i>Codequalität</i>		x		
<i>Übertragbarkeit</i>			x	

Tabelle 5: Qualitätsanforderungen

2.8. Ergänzungen

Der SCPF-Bot kauft eigenständig Buchstaben von der SEG/ anderen Bots, auch ohne direkten Kundenauftrag.

3. Entwurf

3.1. Grobentwurf

Unsere Systemmodellierung geschah mit Hilfe der Klassen-Ableitung nach OMT. Zunächst wurden Klassen, danach zugehörige Attribute und Assoziationen gefunden (siehe Anhang 7.2, Seite 19).

Aus den Ergebnissen wurde ein UML-Diagramm entwickelt (Abb. 2).

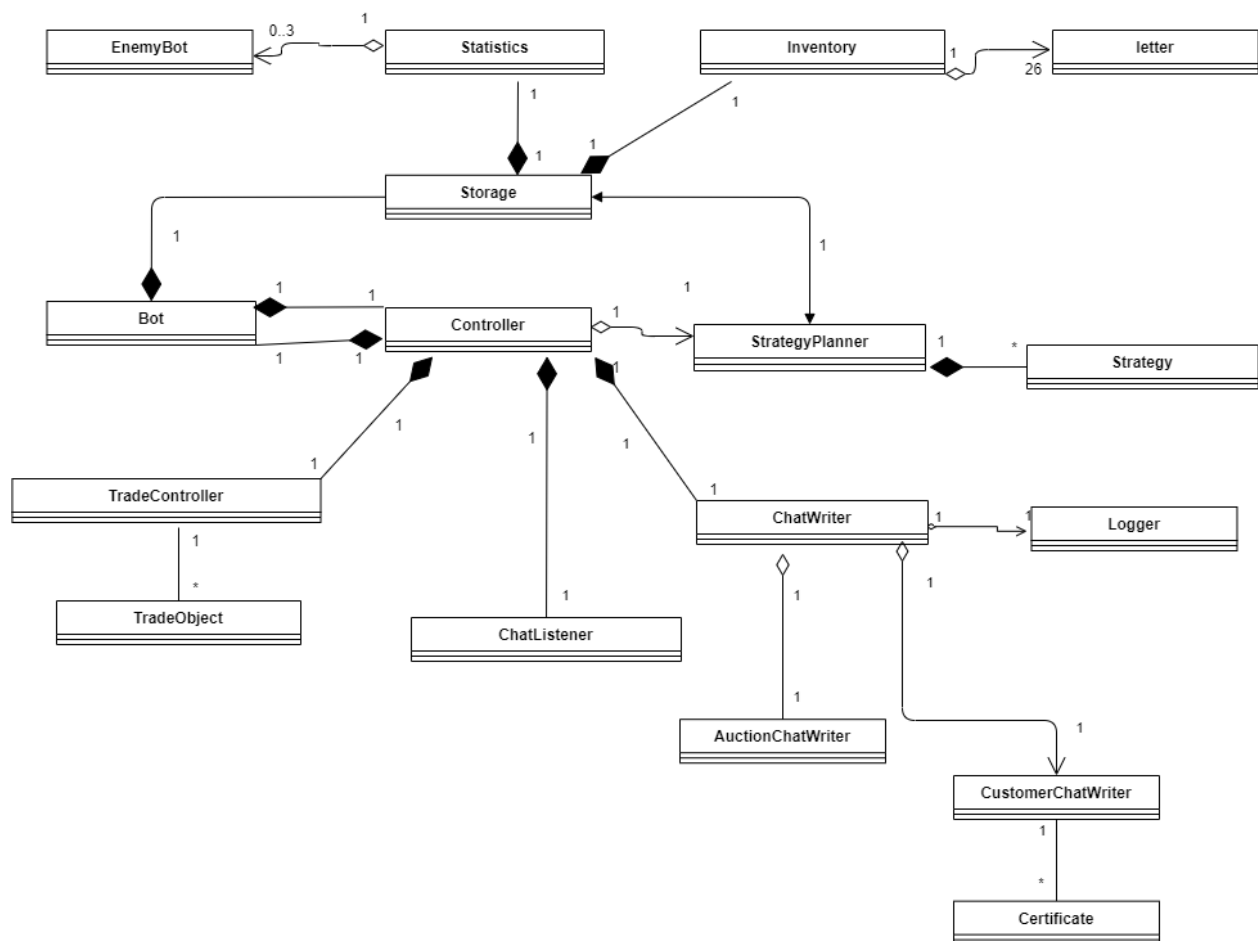


Abbildung 2: Grobentwurf mit Klassen

3.2. Feinentwurf

Im Feinentwurf (Abb. 3) wurden die Klassen um noch fehlende Attribute und Methoden ergänzt und zu sinnvollen Paketen zusammengefasst. Unser Bot wurde in die Pakete Trade, Storage und Chat aufgeteilt, die alle im Main-Package zusammen laufen. Im Feinentwurf wurden ebenfalls sinnvolle Design-Patterns, wie MVC, Singleton, Observer und Factory für die Entwicklungsphase vorgeschrieben. Der finale Feinentwurf wurde mit IntelliJ erstellt und befindet sich im Anhang 7.2 auf Seite 22.

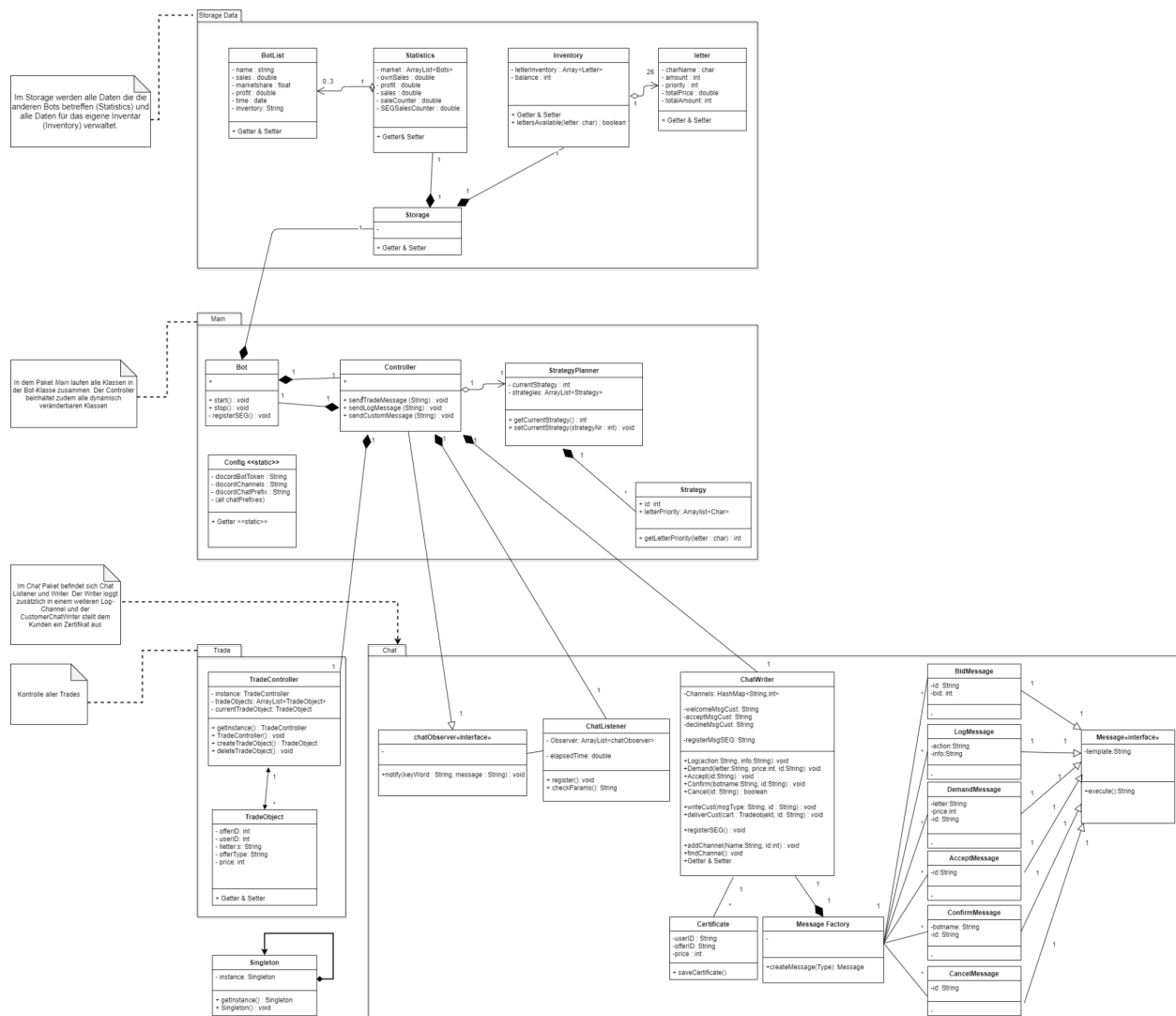


Abbildung 3: Feinentwurf

4. Entwicklung

4.1 Vorgehen in der Entwicklungsphase

Bevor mit der Implementierung gestartet wurde, ist ein Plan für die zu erledigenden Aufgaben aufgestellt worden. Alle Packages (bis auf das main) wurden jeweils zwei Personen zugeordnet, sodass von Beginn an nach dem Konzept des Test-Driven-Development (TDD) vorgegangen werden konnte (siehe Kapitel Testen XX). Ebenfalls wurden die Design-Patterns aus dem Feinentwurf implementiert.

Es wurde ein Gantt-Chart (Abbildung 5) zur Planung der Aufgaben und zur Festlegung der Deadlines erstellt. Zusätzlich musste von jedem Teammitglied eine Zeitdokumentation zu jeder Aufgabe nach dem Schema <Datum, geplante/tatsächliche Zeit, Beschreibung> erfolgen, die ebenfalls im Gantt-Chart festgehalten wurde.

Während der Implementierung aller Klassen und Teste wurde durchweg auf die Einhaltung der Code-Style-Conventions geachtet und eine JavaDoc-HTML-Dokumentation nach den Anforderungen der SEG-Vorlesung geschrieben.

Nachdem die Grundstruktur der Pakete stand, wurde die Arbeitsteilung effizienter gestaltet. Micha und Joris waren verantwortlich für das Mergen der Packages und die restliche Programmierung, Marvin hat das Testen und die Organisation übernommen, Ninnias, Mira und René den ganzen Rest mit Dokumentation, Präsentation und einzelne, weitere Aufgaben. Allgemein konnten notwendige Arbeiten gut aufgeteilt werden und die Deadlines wurden eingehalten.

4.2 Implementierung

Bei der Implementierung wurde streng nach den Grundanforderungen an den Bot und unseres eigenen Feinentwurfs vorgegangen. Die einzige Erweiterung im Vergleich zum Feinentwurf war die CustomerString-Verarbeitung im TradeController.

Zuallererst wurden die einzelnen Packages, namentlich Trade-, Chat-, Storage-Package in das Main-Package gemerged und der Controller wurde erstellt, der die Chatbefehle und die einzelnen Handels-Interaktionen (Trades) kontrolliert. Dann wurden alle Funktionen für die Verwaltung der Buchstaben und der Punkte, sowie von Profit, Absatz und Umsatz ergänzt, sodass erfolgte Trades Echtzeit-Änderungen hervor riefen. Es reicht, unserem Bot irgendeine Nachricht, bswp. "Hallo", in einem privaten Channel zu schreiben, um den Kaufprozess zu starten. Nachdem der komplette Kaufprozess bei

unserem Bot funktionierte, wurde das Bieten beim SEG-Bot mit einem weiteren, eigenen Bot simuliert und um eine eigene Strategie erweitert. Jeder Kauf und Verkauf eines Buchstabens wird gespeichert und der Durchschnittspreis berechnet. Auf diesen Durchschnitt wird ein gewisser Prozentsatz draufgerechnet, in dem der Bot sich bewegen kann. Bevor es in die Interaktion mit den anderen Bots ging, wurden Belastungstests durchgeführt, um etwaige Fehler zu finden und gleichzeitig die Klassen zu formatieren und aufzuräumen. Durch einen regelmäßig, qualitativ hochwertigen Austausch mit der Gruppe Oscar wurde der Test der Bot-zu-Bot-Interaktion erleichtert. Kleinere Probleme wie eine falsche Eingabe oder Erwartung konnten schnell behoben werden, sodass zum Schluss beide Bots miteinander handeln konnten. Ganz zum Schluss wurden alle Log-Nachrichten des Bots in der Konsole, die bisher mit System.out erfolgten durch eine Logging-Klasse ersetzt.

4.3 Eigene Features

Ein eigenes, nützliches Feature wurde bei der Anfragenbeantwortung eingebaut.

Eine Anfrage wird in einer Hashmap gespeichert, mit dessen Länge der Status der Anfrage überprüft wird. Bei erstem Kontakt wird ein Element hinzugefügt. Danach wird ein Wort erwartet, welches als zweites hinzugefügt wird. Als drittes Element wird der Preis ergänzt und zum Schluss eine finale Bestätigung mit Ja oder Abbruch mit Nein erwartet. Danach wird das Objekt aus der Hashmap entfernt. Der Vorteil dieser Herangehensweise ist eindeutig:

1. Anfragen werden erst weiter an den Controller geschickt, wenn diese bestätigt worden sind.
2. Mehrere Anfragen, auch von verschiedenen Nutzern, können gleichzeitig bearbeitet werden.
3. Ein Nutzer muss nicht warten, bis eine Anfrage fertig gestellt wurde, sondern kann direkt mehr einkaufen, wenn dies gewünscht ist.

Ein weiteres eigenes Feature ist die Strategie, die der SCPF-Bot ausführt, wenn mehr als 3000 Punkte zum Kauf zur Verfügung stehen. Ab dann wird ein Buchstabe vom SEG-Bot zur Hälfte des bisher gezahlten Durchschnittspreises gekauft und danach direkt wieder zum Durchschnittspreis angeboten.

Es ist maximal eine Offer Auction für die anderen Bots offen. Wird den anderen Bots ein Buchstabe angeboten und es kommt ein neuer Buchstabe hinzu, wird der zuvor Angebotene dem eigenen Inventar hinzugefügt. Somit geht kein Buchstabe verloren oder wird ewig angeboten.

Schon im Feinentwurf war angedacht, ein Kaufzertifikat für den/die Kunden/Kundin auszustellen, welches in der Entwicklungsphase in Form einer Bilddatei im Chat erfolgreich umgesetzt wurde (Abb. 4).

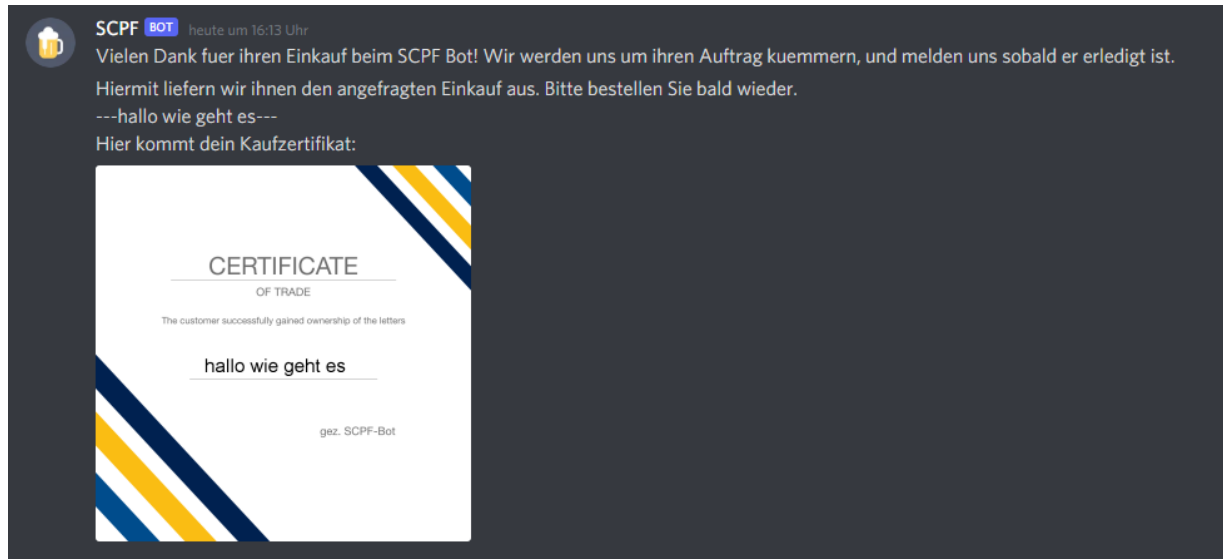


Abbildung 4: Optionales Feature eines Kaufzertifikats nach erfolgreichem Kauf

4.4 Herausforderungen

Während der Implementierung wurden verschiedenste Herausforderungen gemeistert, von denen beispielhaft zwei vorgestellt werden sollen.

Ein Problem, welches gelöst werden musste, war die Formatierung einer ID von einem String zu einem Snowflake-Objekt (Bsp. 1). In der ChatController-Klasse kann mit der `newMessageToWrite`-Methode ein neues Wort ausgeliefert werden. Das auszuliefernde Objekt ist vom Typ `Snowflake` und benötigt eine Channel-ID. Diese ID ist vom Typ `String` und beinhaltet allerdings durch vorherige Formatierung ein `Snowflake` sowie geschweifte Klammern drumherum. Dieser String muss nun so verarbeitet werden, dass nur noch die ID übrig bleibt, die im nächsten Schritt als `Snowflake`-Objekt ausgeliefert werden kann.

```
String idBefore = "Snowflake{3434}";           (1)
String id = values.get("id").replaceAll("[^0-9]+", ""); (2)
String idAfter = "3434";                        (3)
chatWriter.deliver(Snowflake.of(id), values.get("requestedString")); (4)
```

Beispiel 1: Formatierungsproblem durch Snowflake

Ein weiteres Problem, war eine Nullpointer-Exception (Bsp. 2), welche direkt beim Starten des Bots auftrat.

```
Caused by: java.lang.NullPointerException:
Cannot invoke "main.Controller.newMessageToProcess(String, String)"
because "this.controller" is null.
```

Beispiel 2: Nullpointer-Exception beim Start des Programms

Der SEG Bot läuft bereits von Anfang an und sendet dauerhaft Nachrichten in die Channels.

Beim Starten unseres Bots über die Main.main() Funktion wird eine Instanz der Klasse Bot erstellt. In dem Konstruktor dieser Klasse wird zum einen ein client-Objekt des Frameworks Discord4J und zum anderen unsere eigenen Klassen Storage und Controller instanziiert.

Das client-Objekt muss zuerst erstellt werden, da der Controller dieses als Übergabeparameter benötigt.

Durch die ankommenden Nachrichten wird direkt bei der Instanziierung des Bots die Funktion main.Controller.newMessageToProcess() aufgerufen, obwohl noch nicht alle benötigten Klasseninstanzen vollkommen erstellt sind.

Dieses Problem bestand sehr lange, und konnte zunächst ignoriert werden, da es nur sporadisch auftrat und ansonsten keine weiteren Auswirkungen hatte. Die Lösung erscheint dennoch sehr simpel. Es wurde eine globale "botsRunning" Variable eingeführt, die erst nach der kompletten Instanziierung der Klassen auf true gesetzt wird. Dieser Wert wird ebenfalls im ChatController abgefragt, und verarbeitet nun nur noch neue Nachrichten, wenn der Bot vollständig läuft.

4.5 Gantt-Chart

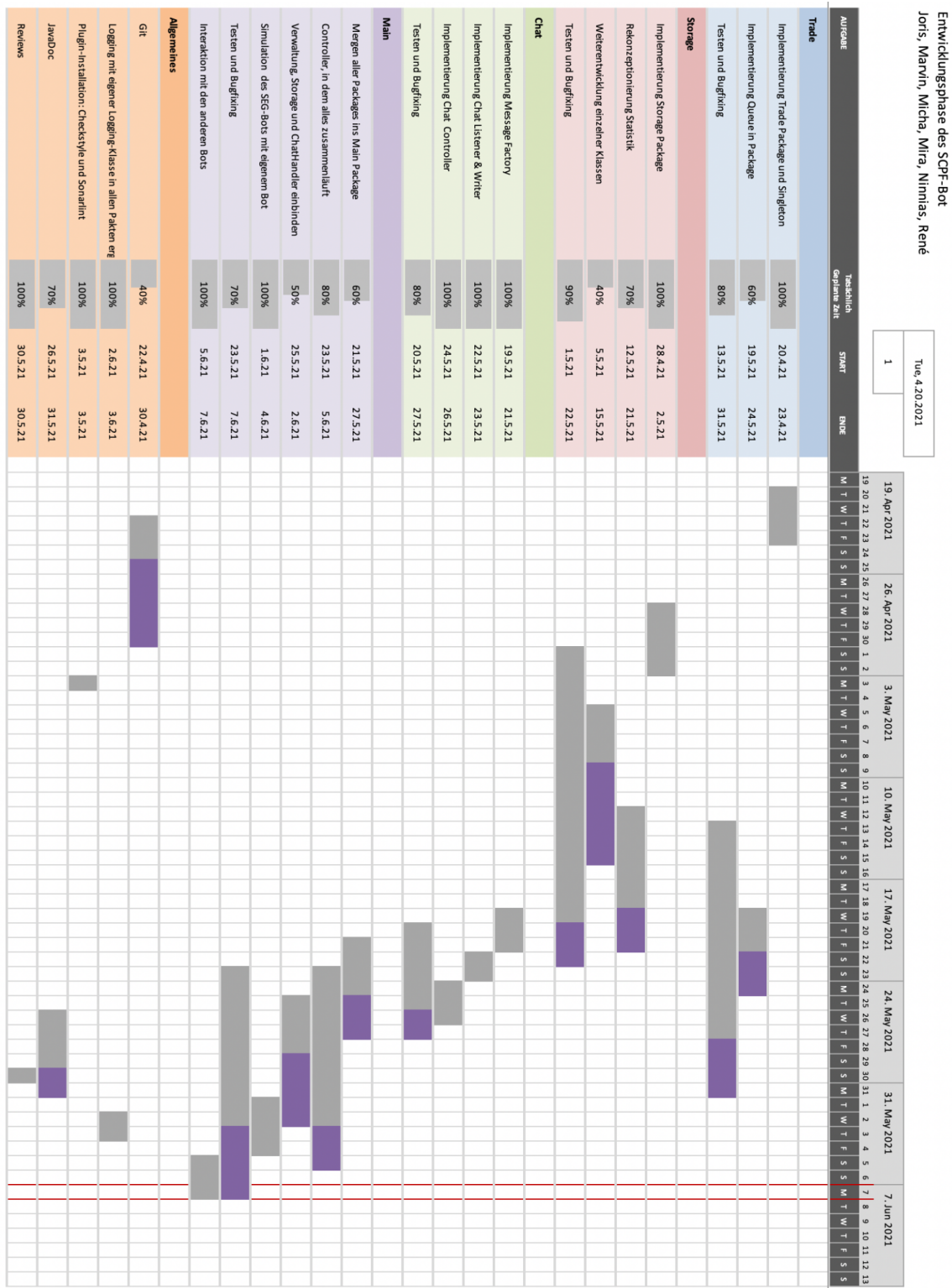
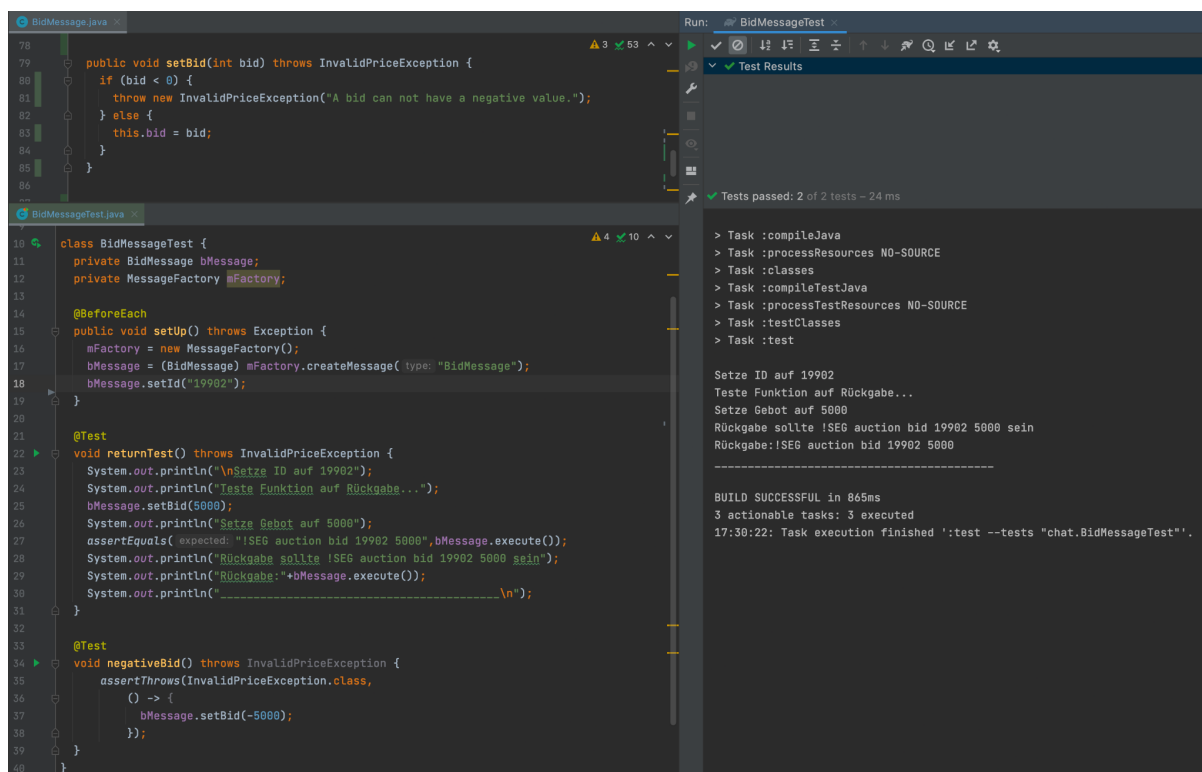


Abbildung 5: Gantt-Chart zur Aufgabendokumentation der Entwicklungsphase

5. Testen

5.1 Test-Driven-Development

Für die Testphase war Marvin Ottersberg der neue Phasenverantwortliche. Der Fokus war von Beginn an, Test-Driven-Development (TDD) durchzuführen. Dazu wurden die Methoden und Parameter zwischen Developer und Test-Schreiber ausgetauscht und zeitgleich zu den geschriebenen Klassen die Junit-Tests verfasst. So konnten schnell essentielle Fehler durch korrektes Exceptionhandling (Abb. 6) behoben werden.



```
78
79 public void setBid(int bid) throws InvalidPriceException {
80     if (bid < 0) {
81         throw new InvalidPriceException("A bid can not have a negative value.");
82     } else {
83         this.bid = bid;
84     }
85 }
86

class BidMessageTest {
    private BidMessage bMessage;
    private MessageFactory mFactory;

    @BeforeEach
    public void setUp() throws Exception {
        mFactory = new MessageFactory();
        bMessage = (BidMessage) mFactory.createMessage("BidMessage");
        bMessage.setBid("19902");
    }

    @Test
    void returnTest() throws InvalidPriceException {
        System.out.println("\nSetze ID auf 19902");
        System.out.println("Teste Funktion auf Rückgabe...");
        bMessage.setBid(5000);
        System.out.println("Setze Gebot auf 5000");
        assertEquals("expected: 'ISEG auction bid 19902 5000', bMessage.execute()");
        System.out.println("Rückgabe sollte ISEG auction bid 19902 5000 sein");
        System.out.println("Rückgabe:" + bMessage.execute());
        System.out.println("-----\n");
    }

    @Test
    void negativeBid() throws InvalidPriceException {
        assertEquals(InvalidPriceException.class,
            () -> {
                bMessage.setBid(-5000);
            });
    }
}
```

Run: BidMessageTest

Test Results

Tests passed: 2 of 2 tests - 24 ms

> Task :compileJava
> Task :processResources NO-SOURCE
> Task :classes
> Task :compileTestJava
> Task :processTestResources NO-SOURCE
> Task :testClasses
> Task :test

Setze ID auf 19902
Teste Funktion auf Rückgabe...
Setze Gebot auf 5000
Rückgabe sollte ISEG auction bid 19902 5000 sein
Rückgabe: ISEG auction bid 19902 5000

BUILD SUCCESSFUL in 865ms
3 actionable tasks: 3 executed
17:30:22: Task execution finished 'test --tests "chat.BidMessageTest"'

Abbildung 6: Exception Handlung und ungültige Parameter durch Tests abgefangen

5.2 Codereview mit Checkliste

Um die Qualität des Codes zu sichern wurde eine Checkliste verfasst, nach welcher jede Klasse von zwei Teammitgliedern gemeinsam mit einem Developer reviewed wurde. Die Liste umfasst die drei Oberpunkte "Dokumentation", "Code" und "Tests".

Die nachfolgende Tabelle zeigt eine Zusammenfassung der Bewertung dieser einzelnen Rubriken nach dem Codereview der jeweiligen Packages, detaillierte Ausarbeitungen aus der Review befinden sich im Anhang 7.3 ab Seite 23.

Ergebnisse der Checkliste für alle Packages:

Reviewed	Main (06.06.21)	Chat (31.05.21)	Trade(05.06.21)	Storage (31.05.21)
Dokumentation				
Codestyle				
JavaDoc				
Codequalität				
Fehler				
Anforderungen				
Exceptions				
Vollständigkeit				
Teste				
Coverage				
Prioritäten				

 = mangelhaft  = weitestgehend erfüllt  = erfüllt

In der "Dokumentation" wurde vor allem die Vollständigkeit und die Übereinstimmung mit den vorgegebenen Code-Style-Conventions mit Hilfe von "CheckStyle" und "SonarLint" überprüft. In einigen Fällen musste die JavaDoc sprachlich angeglichen werden.

Im Abschnitt "Code" wurde neben der Vollständigkeit geprüft, ob der Code kompiliert, ob die richtigen Exceptions bei ungültigen Parametern geworfen werden und ob Fehler, die durch die Tests aufgezeigt wurden, korrigiert wurden. Ebenso wurde reviewed, ob die eigenen Anforderungen an das Package und die Projektanforderung erfüllt wurden.

In der Rubrik “Test” wurde neben ihrer Funktion die Coverage der Tests kontrolliert. Hierbei wurde darauf geachtet, dass vor allem die Funktionen und Methoden mit einer hohen Relevanz abgedeckt wurden. Neben zulässigen Parametern wurden ausserdem Nullpointer-Exceptions und Arrays getestet.

Insgesamt war es möglich eine **Code-Coverage von 28 %** umzusetzen.

Aufgrund der hohen Komplexität der Klassen ChatWriter, ChatListener und ChatController im Chat-Package, hätte für die Tests ein zu hoher Aufwand betrieben werden müssen, den Discord-Bot zu mocken, als dass es die zeitlichen Kapazitäten zuließen. Für die Reviews wurde notiert (Anhang 7.2, Seite 24), dass im Chat-Package mehr auf diese Klassen und deren Methoden geachtet werden muss.

6. Fazit

Zum Abschluss unserer Projektphase wurde eine Retrospektive mit der “Keep-Drop-Try”-Methode á la Scrum durchgeführt, die verschiedene Aspekte der Zusammenarbeit offen gelegt hat.

Zusammenfassend kann man sagen, dass sich dieses Projektes in vielerlei Hinsicht als sehr lehrreich erwiesen hat. Die Struktur der Aufgaben und konkrete Deadlines, sowie weitere wöchentliche Meetings zu einem festen Zeitpunkt erlaubten ein zielgerichtetes Arbeiten. Bis zur Entwicklungsphase konnten viele Aufgaben immer gerecht auf das Team aufgeteilt werden. Es war essentiell, mit der Planung anzufangen, dann die Software-Architektur erst grob und dann fein festzulegen und erst dann mit der Entwicklung zu starten. Die meisten Projekte scheitern höchstwahrscheinlich an genau dieser fehlender Planung mit wichtigem Festlegen der Anforderungen.

Die Einarbeitung in Git und Discord4J hat einige Zeit in Anspruch genommen und zu vielen Problemen geführt. Durch gute Kenntnisse einiger Teammitglieder konnten diese erfolgreich behoben werden. Fehlendes Wissen musste sich individuell angeeignet werden und dann in den wöchentlichen Meetings vorgestellt werden, sodass alle davon profitieren konnten und Zeit gespart wurde.

Durch Kompetenz- und Zeit-Mangel wurden die Aufgaben in der Entwicklungsphase

aufgeteilt, sodass sich die einen um die Implementierung und Tests und die anderen um die organisatorischen Dinge kümmern mussten. Dies erlaubte gegenseitige Arbeitserleichterung, nichtsdestotrotz hat es auch zu Problemen geführt. Die Menge an zu erledigenden Aufgaben war nur durch Leadership einiger Teammitglieder und durch strikte Zuweisung von Aufgaben möglich. Diese war aber auch erst notwendig, nachdem die selbstständige Beteiligung zu Wünschen übrig ließ und Verantwortlichkeiten fehlten. Es ist zu erwähnen, dass das persönliche Verhältnis und die Gruppenmoral absolut unbeeinträchtigt waren, die professionelle Arbeitsebene aber noch sehr viel Luft nach oben hatte.

Wichtige Lehren für zukünftige Projektarbeit sind auf jeden Fall, früh genug eine Zeitdokumentation und ein Gantt-Chart anzulegen und diese auch rigoros durchzuführen. Alles in allem war es für die zukünftige Beschäftigung als Software-Entwickler sehr wertvoll den kompletten Software-Entwicklungsprozess einmal von vorne bis hinten durchzugehen und Wissen aus anderen Modulen wie *Objektorientierte Programmierung* und *Testen von Software* sinnvoll einbringen zu können.

7. Anhang

7.1 Lastenheft

Produktfunktionen nach ausführlichem Schema:

1. **Titel:** Request Customer Cart
2. **Kurzbeschreibung:** Ein Kunde wünscht ein Wort zu kaufen. Wenn SCPF das Wort vorrätig hat, werden Worte und Punkte ausgetauscht.
3. **Akteure:** Customer
4. **Vorbedingungen:** SCPF Besitzt Buchstaben, oder kann diese zeitig käuflich erwerben, und Kunde benötigt Buchstaben
5. **Beschreibung des Ablaufs:**
Der Kunde stellt eine Anfrage an SCPF für ein Wort.
SCPF prüft Preis aufgrund von Erfahrungswerten / Einstellungen & kann ablehnen oder annehmen.
Nimmt der SCPF das Angebot an, überprüft er ob er die benötigten Buchstaben vorrätig hat, oder ob er sie zusammen kaufen kann.
Sobald er das benötigte Wort zusammen hat, liefert er es an den Kunden aus.
Den Gewinn behält der SCPF für sich.
6. **Auswirkungen:**
Nimmt der SCPF das Angebot nicht an, muss der Kunde ein Gegenangebot machen oder das Angebot verfällt.
SCPF verliert Buchstaben für Punktgewinn (wenn er Buchstaben schon besitzt) bzw. verliert vorübergehend Punkte für Punktgewinn (wenn Buchstaben noch erst eingekauft werden müssen).
Bei erfolgreichem Handel hat Nutzer das Wort, und SCPF Gewinn gemacht.
7. **Anmerkungen:** /

1. **Titel:** Adapt Prioritization
2. **Kurzbeschreibung:** Anpassen der Kaufstrategie, wenn SCPF immer überboten wird. SCPF hat nach mehreren Durchläufen des Alphabets einen bestimmten Buchstaben nicht ein weiteres mal erworben.
3. **Akteure:** SEG.
4. **Vorbedingungen:** Kaufversuch muss vorher x-mal fehlgeschlagen sein. SCPF muss Anfrage von Nutzer haben, oder auf Vorrat kaufen wollen.
5. **Beschreibung des Ablaufs:**
SCPF möchte Buchstaben erwerben.

SCPF bietet in mehreren Läufen für einen Buchstaben und wird immer überboten.

SCPF passt das Angebot an und bietet nächstes Mal für diesen Buchstaben mehr

6. Auswirkungen:

Priorität eines Buchstaben erhöht sich. SCPF versucht in Zukunft den Buchstaben öfter / auch für einen höheren Preis zu kaufen.

7. Anmerkungen: /

7.2 Entwurf

Tabelle für erste Klassenbeschreibung:

Klasse	Attribute	Methode	Funktion der Klasse (Funktionsbeschreibung)
Controller	+Chat Writer +Chat Listener +Customer Request Handler +Bot +Trade Controller		Klasse in der alles zusammen läuft, was sich dynamisch verändert, Berechnungen erfordert, oder auf Dinge reagieren muss. Ausgenommen ist das Storage.
Strategy Planner:	+actualStrategy: int +ArrayList<Strategy>	+getActualStrategy() +updateStrategy(ArrayList<Strategy>)	adaptiert die Kaufstrategie durch vorhandene Daten
Trade-Controller:	+Trade-Objects: ArrayList<Trade-Object>	+offer(tradeObject) +buy() +sell() +confirmOrder() +declineOrder()	Kontrolliert /verwaltet alle geplanten Tradeobjekte.
Trade-Object	+OFFER_ID: int +USER_ID: int +OFFER_TYPE: String +requestedLetters: String +price:int +isNegotiable:boolean	+getter-Funktionen	Ein Trade Object ist ein zum Handel angefragte Objekt, dass u.a. gebraucht wird wenn ein Dozent einen Warenkorb anfordert.
Bot:	+Controller(Controller) +Storage(Storage)	+start() +stop() +registerSEG()	In der Bot Klasse laufen alle anderen Klassen zusammen. Die Bot Klasse ist die übergeordnete Klasse von allen Prozessen

			(Controller) und allen Daten (Storage)
Storage:	+Inventory(Inventory)		Im Storage werden alle Daten die die anderen Bots betreffen (Statistics) und alle Daten für das eigene Inventar (Inventory) verwaltet.
Inventory:	+letters: array +balance: int	+updateBalance() +displayInventory(Int erval) +isLetterAvailable():boolean +hasEnoughMoney():boolean	Das Inventory enthält alle Daten zum aktuellen Kontostand und Buchstaben des eigenen Bots.
Letter:	+name: char +amount:int +priority:int +price: double	+getter & setter()	Die Klasse Letter gibt es 1 mal für jeden der möglichen 26 Buchstaben. In ihr wird eine Anzahl sowie ein durchschnittlicher Preis festgehalten.
Statistics:	+Bots:ArrayList	+getBot(int Bot_ID) +addTradeToStatistic(int Bot_ID)	Die Klasse Statistics speichert Informationen über alle Handelsvorgänge.
Enemy Bot	+sumBoughtFromSEG: double +lettersBoughtFromSEG: double +profit: double +sales: double +boughtLetters: int +moneyGivenToSEG: double	+findBiggestEnemyAndAttack() +getter-Funktionen	Bot Klasse von der eine Instanz für jeden Bot (gegnerisch) vorliegt.
Customer Request Handler	+TradeObjekt(Request)	+newRequest()	Die Klasse Customer Request Handler ist die Steuerungsklasse für alle Anfragen die von einem echten Kunden an unseren Bot gestellt werden.
Chat Writer:	+commands: ArrayList <Commands>	+writeCommandX()	Die Klasse Chat Writer fasst die verschiedenen Unterklassen um in den Auction Chat, dem Customer Chat und den anderen Chats zu

			kommunizieren in sich zusammen. Sie ist auch für die Log Nachrichten verantwortlich.
Auctionchat Writer:	+Command: String	+write() +createCertificate()	Der Auction Chat Writer schreibt Nachrichten in den Auktions Channel.
Logger:	+Command: String +USER_ID +OFFER_ID +soldItems +price	+write() +export()	Der Logger schreibt die Transaktionen unseres Bots in den Log Channel.
Customer Chat Writer	+Request +Certificate	+sendCertificate() +answer() +confirm()	Der Customer Chat Writer antwortet auf Anfragen eines echten Kunden.
Certificate	+USER_ID +OFFER_ID +soldItems: Array<String> +price +timestamp: String	+createCertificate()	Eine Instanz der Klasse Zertifikat kann erzeugt werden, um dem Kunden ein Kaufzertifikat auszuhändigen.
Chat Listener	+Observer: ArrayList<Bots> +Observer: ArrayList<SEG> +Observer: ArrayList<Chat> +elapsedTime: double	+register() +notify() +logger()	Der Chat Listener überwacht alle möglichen Channel auf neue Nachrichten und leitet diese dann weiter.

SEG SS 2021



7.3 Testen

 = mangelhaft  = weitestgehend erfüllt  = erfüllt

Storage Package, Reviewed am 31.05:

Reviewed	Fortschritt	Notwendige Änderungen
Dokumentation		
Codestyle <ul style="list-style-type: none"> - Lesbarkeit - Kommentare 	Google, SonarLint	Zeilen kürzen, Umlaute entfernen
JavaDoc <ul style="list-style-type: none"> - Vollständigkeit 		Sprache vereinheitlichen
Codequalität		
Fehler <ul style="list-style-type: none"> - Testresultate - Compiler - if ohne else 		Testergebnisse interpretieren
Anforderungen <ul style="list-style-type: none"> - Eigene - Grundanforderungen 		
Exceptions <ul style="list-style-type: none"> - ungültige Parameter 		
Vollständigkeit		ToDo's abarbeiten
Teste		
Coverage <ul style="list-style-type: none"> - Laufen die Tests durch? 	Methoden: 80%	Storage.java abdecken, vereinzelt Setter testen
Prioritäten <ul style="list-style-type: none"> - Ungültige Parameter - Null-Pointer 		Storage.java Hauptfunktionen testen

Chat Package, Reviewed am 31.05:

Reviewed	Fortschritt	Notwendige Änderungen
Dokumentation		
Codestyle <ul style="list-style-type: none"> - Lesbarkeit - Kommentare 	Google, SonarLint	
JavaDoc <ul style="list-style-type: none"> - Vollständigkeit 		Vervollständigung
Codequalität		
Fehler <ul style="list-style-type: none"> - Testresultate - Compiler - if ohne else 		
Anforderungen <ul style="list-style-type: none"> - Eigene - Grundanforderungen 		Logging und Certificate fehlen noch
Exceptions <ul style="list-style-type: none"> - ungültige Parameter 		
Vollständigkeit		Package unvollständig
Teste		
Coverage <ul style="list-style-type: none"> - Laufen die Tests durch? 	Methoden: 35%	Coverage erhöhen
Prioritäten <ul style="list-style-type: none"> - Ungültige Parameter - Null-Pointer 		Chat Controller/Writer/Listener wegen zu hoher Komplexität nicht berücksichtigt. Sollen im Review von den Entwicklern besonders beachtet werden!

Trade Package, reviewed am 05.06:

Reviewed	Fortschritt	Notwendige Änderungen
Dokumentation		
Codestyle <ul style="list-style-type: none"> - Lesbarkeit - Kommentare 	Google, SonarLint	
JavaDoc <ul style="list-style-type: none"> - Vollständigkeit 		Vervollständigung
Codequalität		
Fehler <ul style="list-style-type: none"> - Testresultate - Compiler - if ohne else 		
Anforderungen <ul style="list-style-type: none"> - Eigene - Grundanforderungen 		
Exceptions <ul style="list-style-type: none"> - ungültige Parameter 		
Vollständigkeit		Package unvollständig
Teste		
Coverage <ul style="list-style-type: none"> - Laufen die Tests durch? 	Methoden: 35%	Coverage erhöhen
Prioritäten <ul style="list-style-type: none"> - Ungültige Parameter - Null-Pointer 		Hauptfunktionen

Main Package, Reviewed am 06.06:

Reviewed	Fortschritt	Notwendige Änderungen
Dokumentation		
Codestyle <ul style="list-style-type: none"> - Lesbarkeit - Kommentare 	Google, SonarLint	
JavaDoc <ul style="list-style-type: none"> - Vollständigkeit 		Fehlt vollständig
Codequalität		
Fehler <ul style="list-style-type: none"> - Testresultate - Compiler - if ohne else 		Deprecated Funktionen evtl. entfernen
Anforderungen <ul style="list-style-type: none"> - Eigene - Grundanforderungen 		
Exceptions <ul style="list-style-type: none"> - ungültige Parameter 		
Vollständigkeit		
Teste		
Coverage <ul style="list-style-type: none"> - Laufen die Tests durch? 		Strategie testen
Prioritäten <ul style="list-style-type: none"> - Ungültige Parameter - Null-Pointer 		