SoPra der Toten

Gruppenphase

Inhaltsverzeichnis

1 Einleitung **2**

2 Organisatorisches **3**
2.1 Entwurf . . . 4
2.2 Implementierung . . . 5
2.3 Benötigte Tools . . . 6
2.4 Abgaben . . . 6

3 Das Spiel **8**
3.1 Spielaufbau . . . 8
3.2 Spieldetails . . . 9
3.2.1 Charaktere . . . 9
3.2.2 Kolonie . . . 10
3.2.3 Moral . . . 10
3.2.4 Standorte . . . 11
3.2.5 Infektionen . . . 11
3.2.6 Krisen . . . 13
3.2.7 Fähigkeiten . . . 13
3.2.8 Karten . . . 14
3.2.9 Aktionen . . . 14
3.3 Spielablauf . . . 18
3.3.1 Spielerphase . . . 18
3.3.2 Koloniephase . . . 19
3.4 Das Originalspiel . . . 20
3.5 FAQ . . . 21

4 Technische Details **24**
4.1 Spielablauf . . . 24
4.1.1 Ziel . . . 24
4.1.2 Registrierungsphase . . . 24
4.1.3 Vorbereitungsphase . . . 24
4.2 Zufallsberechnungen . . . 25
4.2.1 Karten . . . 25
4.2.2 Krisen . . . 25
4.2.3 Überlebende . . . 26

4.2.4 Aktionswürfel . . . 27
4.2.5 Infektionswürfel . . . 27
4.2.6 Zufallsbegegnungen . . . 27
4.3 Konfigurationsdatei . . . 29
4.4 Client-Server Kommunikation . . . 31
4.4.1 Commands & Events . . . 31
4.5 Command-Line-Interface . . . 32
4.6 Build-Skript . . . 32
4.7 Codequalität . . . 34

5 Tests **35**
5.1 Unit-Tests . . . 35
5.2 Integrationstests . . . 35
5.3 Systemtests . . . 36

A Anhang **37**
A.1 Fähigkeiten . . . 37
A.2 Karten . . . 39
A.3 Commands und Events . . . 41

Änderungsverzeichnis

23.09.2021: Uhrzeit Codeabnahme angepasst 7
08.09.2021: Sequenzdiagramme zu Interaktion bzgl. Commands und Events ergänzt . . 31
21.09.2021: Behanglung von Leave vereinfacht 41

1. Einleitung

Wir schreiben den Winter des Jahres 2021. Dank der zahlreichen Impfungen scheint COVID-19 endlich besiegt zu sein und ein normaler Alltag wird langsam wieder denkbar. Die Universitäten des Landes wechseln alle nacheinander wieder in den Präsenzbetrieb. Alle wiegen sich in Sicherheit, die Studierenden in den Hörsäalen wie die Professoren an ihren Lehrstühlen, als es plötzlich noch viel schlimmer kommt als erwartet...

Ein neuer, viel gefährlicherer Virus hatte sich still und leise eingeschlichen und große Teile der Gesellschaft in Zombies verwandelt.

2 Wochen später...

Überlebende der Apokalyse haben sich gemeinsam an der Universität verbarrikadiert und eine Gesellschaft gegründet. Das Hauptquartier der Überlebenden, genannt Kolonie, ist das Mensagebäude. Dort ist genug Platz für alle, auch für die Kindergartenkinder, die sich nicht selbst versorgen oder verteidigen können. Auch sie müssen geschützt, medizinisch versorgt und ernährt werden. Jeder und jede Überlebende muss sich mit seinen oder ihren jeweiligen Fähigkeiten einbringen, um das Überleben aller zu sichern. Sie müssen gegen die Zombies kämpfen, für Essen sorgen und andere Materialien in der Umgebung finden.

Und als ob das nicht schon genug wäre, muss man auch immer wieder gegen Krisen kämpfen, die die Kolonie heimsuchen...

2. Organisatorisches

Nach der einwöchigen Übungsphase beginnt nun der Praxisteil des Software-Praktikums. Dieser ist eine Gruppenphase (∼4 Wochen) und eine darauf folgende Einzelphase (∼2 Wochen) aufgeteilt.

In der Gruppenphase werden Sie gemeinsam mit Ihren Kommilitonen in einem Team von 5–7 Personen ein Spiel entwerfen, implementieren und testen. Die Gruppenzuteilung erfolgt durch den Lehrstuhl am ersten Tag der Gruppenphase. Sie werden für die Gruppenphase in MS-Teams einem Team hinzugefügt, welches Ihnen als virtueller Arbeitsraum dient. Ein Tutor wird Ihnen zur Seite stehen und regelmäßig mit Ihrem Team in Kontakt stehen.

freiwilliges Präsenzangebot

Wenn Sie zuvor angegeben haben, dass Sie am freiwilligen Präsenzangebot interessiert sind und einer Präsenzgruppe zugeteilt werden, so steht Ihrer Gruppe ab Mittwoch, 08.09.2021, für die Dauer der Gruppenphase ein Arbeitsraum an der Universität zur Verfügung. Ihr Tutor wird Ihnen mitteilen, welcher Raum für Sie zur Verfügung steht, und mit Ihnen vereinbaren, wann Sie Ihren Arbeitsraum erstmals aufsuchen können. Beachten Sie, dass dafür ein Nachweis über eine vollständige Impfung, Genesung, oder einen negativen PoC-Antigen-Schnelltest nicht älter als 24 Stunden erforderlich ist. Dieser Nachweis muss täglich erbracht werden.

Die ersten ca. eineinhalb Wochen der Gruppenphase werden Sie für den Entwurf verwenden, die darauffolgenden ca. zweieinhalb Wochen für die Implementierung (inkl. Testen). Zur Planung des Spiels gehört die Ausarbeitung von Grob- und Feinentwürfen. Sie werden Ihre Entwürfe einem Mitarbeiter des Lehrstuhls vorstellen und Feedback erhalten. Die Implementierung umfasst den Spielserver mit der Spiel-Logik. Zudem müssen Sie eine umfangreiche Testsuite entwickeln, welche möglichst alle Sonderfälle, die auftreten können, abtestet.

Genauso wie Kunden im realen Leben ist auch der Lehrstuhl sehr wechselhaft in seinen Anforderungen an das zu implementierende Spiel. Daher wird es nach dem Entwurfsreview eine Änderung an den Spielregeln geben. Ihr Entwurf muss flexibel genug gehalten sein, um die Änderungen in der Aufgabenstellung ohne größeren Aufwand berücksichtigen zu können. Die konkreten Änderungen werden dann nach den Entwurfsreviews in einem gesonderten Dokument bekannt gegeben.

Nach der Gruppenphase bekommen Sie in der Einzelphase eine neue Aufgabe, die von allen Teilnehmern alleine bearbeitet werden muss. Nähere Informationen zum Ablauf der Einzelphase erhalten Sie nach Ende der Gruppenphase. Voraussetzung für die Einzelphase ist die erfolgreiche Absolvierung der Gruppenphase.

Hinweis

Bitte beachten Sie für den Ablauf der Gruppenphase zusätzlich auch die Informationen aus dem Dokument „Organisatorisches“ , das Ihnen bereits zu Beginn des Software-Praktikums im CMS zur Verfügung gestellt worden ist. Dort ist auch der vollständige Zeitplan sowie eine genauere Beschreibung des Hybridbetriebs aus Online- und Präsenzsitzungen enthalten. Während der Gruppenphase besteht eine Anwesenheitspflicht, die Details dazu entnehmen Sie bitte ebenfalls dem Dokument „Organisatorisches“.

2.1. Entwurf

Im Entwurf muss Ihre Gruppe folgende Aufgaben erfüllen:

1. Sie müssen ein UML-Klassendiagramm Ihres Systems erstellen. Das Klassendiagramm muss sämtliche relevanten Klassen Ihrer Implementierung, sowie deren jeweils wichtigste Felder und Methoden enthalten. Hierdurch soll erkennbar werden, welche Daten und welche Funktionalität Sie in der jeweiligen Klasse kapseln. Die Relationen der Klassen untereinander müssen korrekt modelliert sein. Im Klassendiagramm müssen alle Klassen, sowie alle nicht-trivialen Methoden samt ihrer Parameter aufgeführt sein. Zu den trivialen Methoden zählen insbesondere Standard-Getter- und Setter-Methoden, sowie `hashCode`, `equals` und `toString`.
2. Sie müssen das Zusammenspiel Ihrer Klassen anhand von UML-Sequenzdiagrammen für die folgenden 2 Szenarien demonstrieren:
   - Initialisierung des Spiels
   - Durchsuchen eines Standortes mit Zufallsbegegnung bei der auch 1 Kind mit ins Spiel kommt

   In der Anforderungsänderung nach dem Entwurfsreview werden wir noch ein drittes Szenario bekannt geben, für das Sie das Zusammenspiel Ihrer Klassen anhand eines weiteren UML-Sequenzdiagramms demonstrieren müssen.
3. Sie müssen einen detaillierten Zeit- und Arbeitsplan für die Implementierungsphase inkl. Testen aufstellen. In diesem Plan wird festgehalten, wer für welche Teile der Implementierung zuständig sein wird und wer für das Testen welcher Komponenten verantwortlich ist. Beachten Sie in diesem Plan insbesondere die Abhängigkeiten zwischen verschiedenen Komponenten und die Reihenfolge der nächsten Meilensteine. Vergessen Sie nicht, auch das Schreiben der Integrations- und Systemtests im Plan zu berücksichtigen.

Entwurfsreview & Entwurfsabnahme

Im Entwurfsreview erhalten Sie von einem Mitarbeiter des Lehrstuhls Feedback zu Ihrem Entwurf. Bei der Entwurfsabnahme wird Ihr Entwurf von einem Mitarbeiter des Lehrstuhls abgenommen. Beide Termine finden verpflichtend in Präsenz statt – beachten Sie daher, dass dafür ein Nachweis über eine vollständige Impfung, Genesung, oder einen negativen PoC-Antigen-Schnelltest nicht älter als 24 Stunden erforderlich ist.

Nach den Entwurfsreviews werden wir Änderungen an der Aufgabenstellung vornehmen. Die Änderungen werden sich am Originalspiel orientieren. Versuchen Sie Ihren Entwurf so modular zu halten, dass etwaige Änderungen leicht in Ihren Entwurf zu integrieren sind. Sie sollten allerdings nicht alle möglichen Änderungen in den Entwurf vorab einbauen.

Beachten Sie in beiden Fällen, dass der aktuelle Stand Ihres Entwurfs, den Sie dem Lehrstuhlmitarbeiter vorstellen werden, bereits 1 Stunde vor dem Ihnen zugewiesenen Review- bzw. Abnahme-Termin in unserem *GitLab*-Repository sein muss (siehe dazu auch Abgaben).

2.2. Implementierung

In dieser Phase fertigen Sie in Ihrer Gruppe die eigentliche Implementierung des Spielsimulators an. Außerdem erstellen Sie Unit-Tests, Integrationstests und Systemtests. Ihre Spiel-Implementierung muss den folgenden Anforderungen genügen:

1. Ihre Implementierung muss die von uns erstellten Systemtests bestehen. Unsere Systemtests lassen wir während der Implementierungsphase regelmäßig auf Ihrer Implementierung laufen, die Testergebnisse werden Ihnen zu gegebener Zeit in regelmäßigen Abständen zur Verfügung gestellt.

2. Ihre Implementierung muss die von Ihnen erstellten Unit-Tests, Integrationstests und Systemtests bestehen. Die Unit-Tests dienen Ihnen als Hilfe bei der Fehlersuche und Korrektur. Wenn Sie Unit-Tests haben, die Ihre Implementierung nicht besteht, haben Sie entweder einen Fehler beim Erstellen der Tests gemacht oder Ihre Implementierung ist noch fehlerhaft. Mit Ihren Tests sollen Sie dabei eine möglichst hohe Code-Coverage erzielen und dabei möglichst sinnvolle Szenarien abtesten, was durch das Finden von Mutanten[1] sichergestellt wird.

3. Wir werden Ihre Implementierung mit den Codeanalysetools *PMD* und *SpotBugs* untersuchen. Wir erwarten, dass wir hierbei keine Probleme finden. Falls die Tools in Ihrem Code Probleme anzeigen, erwarten wir, dass Sie ausreichend begründen können, warum Sie diese nicht behoben haben. Mit dem Build-Skript, das wir für Sie erstellt haben, können Sie diese Tools ab dem Beginn der Implementierungsphase selbst ausführen und so prüfen, ob Ihr Code noch Probleme aufweist.

Das bloße Bestehen der Tests reicht nicht aus, Ihr Code muss auch gut strukturiert sein und den Konzepten der Vorlesung folgen. Insbesondere muss jedes einzelne Gruppenmitglied

[1] https://en.wikipedia.org/wiki/Mutation_testing

einen signifikanten Anteil zum Code Ihrer Gruppe beitragen. (Achten Sie hierbei darauf, auf Ihrem Computer *git* richtig konfiguriert zu haben, sodass Ihre Commits auch in unserem *GitLab* Ihnen zugeordnet werden.)

Codereview

Im Codereview wird ein Mitarbeiter des Lehrstuhls gemeinsam mit Ihnen Ihren Code ansehen, Schwachstellen aufzeigen und konkrete Verbesserungsvorschläge geben. Da es sich beim Codereview um einen Präsenztermin handelt, ist dafür ein Nachweis über eine vollständige Impfung, Genesung, oder einen negativen PoC-Antigen-Schnelltest nicht älter als 24 Stunden erforderlich.

2.3. Benötigte Tools

Für die Teilnahme am Software-Praktikum benötigen Sie einen eigenen Laptop, den Sie selbst mitbringen müssen. Sie müssen die folgenden Tools auf Ihrem Gerät installieren:

- *Java 16* (unsere Referenzplattform arbeitet mit *OpenJDK 16.0.2*)[2]
- Versionsverwaltungssystem *git*[3]
- IDE Ihrer Wahl (wir empfehlen *IntelliJ*[4])

Das Build-System *Gradle*, welches wir verwenden, braucht bei Ihnen nicht explizit installiert zu werden, da wir Ihnen ein Projekt zur Verfügung stellen, in welchem ein *Gradle-Wrapper* bereits mitgeliefert wird, der sich selbstständig um die Installation kümmert. Alle weiteren Tools, die Sie benötigen, brauchen nicht installiert zu werden, sondern werden von uns im Projekt bereits vorkonfiguriert zur Verfügung gestellt.

Es dürfen alle Bibliotheken verwendet werden, die in die *Gradle*-Konfigurationsdatei eingebunden sind. Weitere Bibliotheken dürfen nicht verwendet werden.

2.4. Abgaben

Wir stellen Ihnen für die Gruppenphase ein *git*-Repository im lehrstuhleigenen *GitLab*[5] zur Verfügung. Um das zur Verfügung gestellte Repository klonen zu können bzw. in das Repository pushen zu können, benötigen Sie einen *SSH*-Key. Wie Sie einen *SSH*-Key generieren können und welche sonstigen technischen Voraussetzungen für die Nutzung des Repositorys notwendig sind, haben Sie bereits in der praktischen Übung kennengelernt. In der Gruppenphase wird die gesamte Gruppe auf einem Repository arbeiten.

[2] https://jdk.java.net/16/
[3] https://git-scm.com/
[4] https://www.jetbrains.com/idea/
[5] https://sopra.se.cs.uni-saarland.de/

Für die Abgabe Ihres Entwurfs wird Ihnen ein separater Branch (nicht `master`) zur Verfügung gestellt. Alle Entwurfsdokumente und zugehörige Darstellungen müssen ins Repository gepusht werden. Sie können dazu u.a. auch Fotos von Ihren Diagrammen auf Papier etc. hochladen, sofern die Auflösung hoch genug ist und Details erkennbar sind. Die finale Version Ihres Entwurfs muss durch den vorgegebenen Tag[6] `entwurfsabnahme` gekennzeichnet sein, der an der abgegebenen Revision angebracht werden muss. Mit dem *git*-Kommando `git tag <tagname>` kann man einen Tag für den aktuellen Commit erstellen. Tags müssen explizit gepusht werden. Beachten Sie, dass die Abgabe Ihres Entwurfs spätestens eine Stunde vor dem Ihnen zugewiesenen Abnahmetermin ins Repository gepusht werden muss.

Für Ihre Implementierung müssen Sie dann nach der Entwurfsabnahme den `master`-Branch nutzen. Dort wird Ihnen bis zum Beginn der Implementierungsphase ein Projekt mit einem minimalen Code-Gerüst zur Verfügung gestellt werden. Andere Branches werden von uns in der Implementierungsphase nicht mehr berücksichtigt.

Für die finale Abgabe Ihrer Implementierung zählt der letzte Commit auf dem `master`-Branch, der bis spätestens ~~23:59 Uhr~~ 19:59 Uhr MESZ des Tages der Codeabnahme im Repository gemacht wurde. Es liegt in Ihrer Verantwortung zu überprüfen, dass der Push im Repository angekommen ist und alles rechtzeitig auf aktuellem Stand ist. Ihre Abgaben müssen auf unserer Referenzplattform (*Debian 10* mit *OpenJDK 16*) ausführbar sein.

[6] https://git-scm.com/book/en/v2/Git-Basics-Tagging

3. Das Spiel

3.1. Spielaufbau

SoPra der Toten ist ein kooperatives Brettspiel, bei dem die Spielerinnen und Spieler versuchen ein gemeinsam gesetztes Ziel zu erreichen bevor die Zeit abgelaufen ist oder die Nerven der Kolonie blank liegen.

Analog zu einem physischen Brettspiel gibt es eine Welt (sozusagen das Spielbrett), in der es die **Kolonie** und verschiedene **Standorte** gibt.

Die Spielerinnen und Spieler steuern verschiedene **Charaktere** (die *Überlebenden* der Krise), die jede Runde **Aktionen** ausführen können. Sie können z.B. nach Essen und Materialien in der Umgebung suchen, Zombies angreifen und Eingänge verbarrikadieren.

Wann immer ein Charakter auf einen Zombie treffen könnte, besteht eine gewissen Wahrscheinlichkeit sich eine **Infektion** bzw. Verwundung zuzuziehen.

Es gibt verschiedene **Karten** im Spiel, z.B. Essens-Karten, welche Nahrungsmarker dem Nahrungsvorrat hinzufügen, oder Krisen-Karten, die von dem Krisen-Kartenstapel gezogen werden.

Von Spielern und Spielerinnen ausgespielte Handkarten landen auf dem *Müllstapel*. Dieser Müll muss von der Kolonie entsorgt werden, damit die **Moral** nicht sinkt.

Welche Aktionen ein Charakter ausführen kann, ist zum einen abhängig von seinen **Fähigkeiten**, und zum anderen ggf. davon, wie viel ihn die Aktion kostet. In diesem Fall muss der Spieler bzw. die Spielerin mit einem sogenannten *Aktionswürfel* eine Augenzahl würfeln, die für den Charakter hoch genug ist, um diese Aktion ausführen zu können.

Jede Runde des Spiels besteht aus **zwei aufeinanderfolgenden Phasen:**

- Die *Spielerphase*, in der alle Spielerinnen und Spieler ihre Aktionen ausführen können und eine neue **Krise** aufgedeckt wird.
- Die *Koloniephase*, in der für die gesamte Kolonie bestimmte Punkte am Ende jeder Runde abgehandelt werden müssen.

Das Spiel endet, sobald das gemeinsame Ziel erfüllt wurde, die Runden abgelaufen sind oder die Moral der Kolonie auf 0 sinkt.

Hinweis

Statt alle Spielparameter bereits während der Implementierung festzulegen, werden wir stattdessen beim Start des Spielservers eine **Konfigurationsdatei** einlesen, die das Spiel zu großen Teilen modelliert. Mit dieser wird bespielsweise das Ziel des Spiels, die eigentliche Spielwelt, aber auch Charaktere und deren Fähigkeiten vorgegeben.

3.2. Spieldetails

3.2.1. Charaktere

Jeder Spieler erhält zu Beginn des Spiels vier Charaktere zur Auswahl, von denen er sich zwei Charaktere aussuchen muss. Alle Spielerinnen und Spieler starten also mit zwei sogenannten **Überlebenden** ins Spiel, allerdings können (und werden vermutlich) im Laufe des Spiels weitere Überlebende zur Gruppe des Spielers hinzugefügt werden. Die Anzahl von Charakteren pro Spieler ist nicht begrenzt.

Außerdem existieren hilflose **Kinder**. Diese sind keinem Spieler zugeordnet, sondern existieren einfach nur in der Kolonie und müssen versorgt werden.

Die Eigenschaften eines Charakters sind in der Konfigurationsdatei angegeben:

- Jeder Charakter hat eine ID (`identifier`) und einen Namen (`name`).
- Jeder Überlebende hat *genau eine* besondere **Fähigkeit** (`ability`). Manche besonderen Fähigkeiten von Überlebenden erfordern den Einsatz von Aktionswürfeln. Alle Fähigkeiten, die dies benötigen, geben es auch an. Eine Liste der besonderen Fähigkeiten finden Sie in Tabelle 1 und Tabelle 2.
- Der **Angriffswert** (`attack`) gibt an, wie hoch die Augenzahl eines Aktionswürfels sein muss, damit der Charakter ein Zombie angreifen kann.
- Der **Durchsuchewert** (`search`) gibt an, wie hoch die Augenzahl eines Aktionswürfels sein muss, damit der Charakter einen Standort durchsuchen kann.
- Jeder Charakter hat einen gewissen **Sozialstatus** (`status`) in der Kolonie, welcher in der Konfigurationsdatei angegeben ist. Startspieler ist derjenige, der den Überlebenden mit dem höchsten Sozialstatus hat. Der Charakter mit dem niedrigsten Sozialstatus wird bei Zombieangriffen immer zuerst angegriffen. Kinder haben an sich keinen Sozialstatus, und werden deshalb als die schwächsten Mitglieder der Kolonie behandelt.

  Sobald der letzte Charakter bzw. Überlebende eines Spielers getötet wurde, verliert dieser Spieler alle Handkarten (die Karten werden aus dem Spiel entfernt, sie kommen nicht auf den Müllstapel). Für den Fall, dass er gerade am Zug ist, wird dieser sofort

beendet. Danach wird ein neuer Überlebender ausgewählt, welcher dann direkt in der Kolonie spawnt (wie in Unterabschnitt 4.2.3 spezifiziert).

Hinweis

Spawnen (aus dem Englischen *to spawn: hervorbringen*) ist ein gängiger Begriff um den (Wieder-) Einstieg eines Charakters in Computerspielen zu beschreiben.

3.2.2. Kolonie

Alle Charaktere spawnen in der *Kolonie*, welche unbegrenzt viel Platz für die Überlebenden (inkl. Kinder) hat.

Von dort aus können sich die Überlebenden zu anderen Standorten bewegen (max. eine Bewegung pro Runde) um nach Essen und Materialien zu suchen. Die Kolonie hat insgesamt so viele Eingänge, wie in der Konfigurationsdatei spezifiziert ist. Vor jedem dieser Eingänge gibt es drei Plätze, auf denen jeweils ein Zombie platziert werden kann. Es können also bis zu **drei** Zombies vor jedem Eingang lauern.

Außerdem befindet sich Müll in der Kolonie. Dieser muss regelmäßig von einem Überlebenden in der Kolonie entleert werden, damit die Moral nicht sinkt. Kinder verlassen die Kolonie nie, da sie nichts tun können.

Die Eigenschaften der Kolonie, welche in der Konfigurationsdatei spezifiziert sind:

- die ID (`identifier`)
- die Anzahl der Eingänge (`entrances`)
- Die `startCards` repräsentieren den Startkartenstapel, von dem alle Spielerinnen und Spieler zu Beginn des Spiels ihre fünf Handkarten bekommen.

3.2.3. Moral

Die Kolonie verfügt über einen Moralwert. Dieser ist in der Konfigurationsdatei angegeben und verändert sich in folgenden Fällen:

- Sobald ein Überlebender oder Kind stirbt, sinkt die Moral um 1.
- Wird eine Krise nicht gemeistert, sinkt die Moral um den in der Krise angegebenen Wert.
- Werden zu einer Krise zwei Karten mehr als benötigt beigetragen, so **steigt** die Moral um 1.

- Pro Hungermaker im Nahrungsvorrat sinkt die Moral pro Runde um 1.
- Pro 10 nicht entsorgten Karten auf dem Müll verliert die Kolonie eine Moral.

Sinkt die Moral auf 0, so ist das Spiel verloren.

3.2.4. Standorte

Neben der Kolonie gibt es weitere Standorte, die in der Konfigurationsdatei spezifiziert werden. Ebenso gibt die Konfiguration an, *wie viele Überlebende* sich gleichzeitig an dem jeweiligen Standort aufhalten können, sowie die Anzahl der Eingänge des Standortes. Vor jedem dieser Eingänge gibt es auch hier drei Plätze, auf denen jeweils ein Zombie platziert werden kann. Es können also bis zu **drei** Zombies vor jedem Eingang lauern.

Die Eigenschaften eines Standortes sind in der Konfigurationsdatei angegeben:

- Jeder Standort hat eine ID (`identifier`) und einen Namen (`name`).
- die Anzahl der Eingänge (`entrances`)
- wie viele Überlebende sich gleichzeitig an dem jeweiligen Standort aufhalten können (`survivorSpaces`)
- Jeder Standort hat einen Kartenstapel (`cards`), der durchsucht werden kann.

Bei jedem Durchsuchen eines Standortes besteht eine gewisse Chance einem neuen Überlebenden zu begegnen. Dieser schließt sich dann der Gruppe des Spielers an kann noch in der gleichen Runde benutzt werden. Neue Charaktere spawnen allerdings immer in der Kolonie.

Es gibt folgende Fälle, die beim Durchsuchen eines Standortes eintreten können:

- Es kann nicht durchsucht werden. Das ist der Fall, wenn es keine Karten mehr an dem Standort gibt oder der Spieler keinen Aktionswürfel mehr übrig hat mit einer Augenzahl $\geq$ Durchsuchenwert des Charakters.
- Es findet eine Zufallsbegegnung statt (= ein neuer Charakter spawnt) anstelle des Durchsuchens.
- Es findet keine Zufallsbegegnung statt und es wird normal durchsucht (= eine Karte gezogen).

Solange Karten im Kartenstapel sind, wird also immer entweder durchsucht (= eine Karte gezogen), oder ein neuer Charakter (plus ggf. Kinder) kommt ins Spiel. Wie genau die Zufallsbegegnungen berechnet werden, ist in Unterabschnitt 4.2.6 beschrieben.

3.2.5. Infektionen

Durch das Würfeln eines *Infektionswürfels* werden Charakteren Verletzungen (Wunde, Erfrierung oder Biss) hinzugefügt. Verletzungen können immer dann passieren, wenn man mit

einem Zombie interagiert. Eine Interaktion mit einem Zombie passiert immer dann, wenn man sich normal bewegt (Ausnahme siehe Karte `FUEL`, Tabelle 3), einen Zombie angreift oder man von einem Zombie angegriffen wird.

Hinweis

Dieser Würfel hat insgesamt zwölf Seiten, wovon drei eine **Wunde** (`wound`), zwei eine **Erfrierung** (`frostBite`) und eine einen **Biss** (`bite`) bedeuten. Wie genau der Würfel implementiert wird, finden Sie in Abschnitt 4.2.

- Wird beim Werfen des Infektionswürfels **eine Wunde oder Erfrierung** gerollt, so erhält der Charakter die entsprechende Verwundung.
- Wird ein **Biss** gerollt, so stirbt der Charakter, die Moral der Kolonie sinkt deswegen um 1 und die Infektion breitet sich an dessen Standort aus. Dafür wird für den Charakter mit dem geringsten Sozialstatus ebenfalls der Infektionswürfel geworfen. Sollte er dabei eine Verletzung erleiden, so stirbt er, die Moral der Kolonie sinkt wieder deswegen um 1 und die Infektion breitet sich weiter aus. Ansonsten überlebt der Charakter und das Ausbreiten der Infektion endet. Sollte hierbei der letzte Charakter eines Spielers sterben, wird zuerst ein neuer Charakter nachgespawnt und danach die Infektionskette weiter abgehandelt.
- Für jede Erfrierung, erhält ein Überlebender zu Beginn des ersten Zuges der nächsten Runde seines Spielers eine weitere Wunde.
- Sobald ein Charakter die dritte Verletzung (Wunde oder Erfrierung!) erleidet, stirbt er.
- In jedem anderen Fall passiert nichts.

Falls während des Spiels alle Überlebenden eines Spielers bzw. einer Spielerin sterben, ist sein bzw. ihr Zug vorbei, er bzw. sie verliert alle Handkarten [1] und er bzw. sie bekommt zufällig einen neuen Überlebenden zugeteilt, welcher dann auch in der Kolonie spawnt. Mit diesem Überlebenden kann er bzw. sie dann in der nächsten Runde wieder normal weiterspielen.

Falls kein Überlebender nachspawnen kann, ist der Spieler bzw. die Spielerin raus.

[1] Diese werden aus dem Spiel entfernt, sie kommen nicht auf den Müllstapel.

***Beispiel:* Bewegung mit Infektion**
Die Spielerin Julia geht mit ihrem Charakter Bob von der Kolonie zum Standort *Tankstelle*, wo sich auch die Charaktere Curd (Sozialstatus 5), Henry (Sozialstatus 2) und Charlotte (Sozialstatus 3) befinden.

→ Julia wirft für Bob den Infektionswürfel und erhält eine 11. Bob wird also von einem Zombie gebissen und stirbt. Die Moral der Kolonie sinkt um 1. (Zum Glück ist die Moral dadurch noch nicht auf 0 gesunken, sonst wäre das Spiel an dieser Stelle verloren.) Danach wird für Henry der Infektionswürfel geworfen, da er den geringsten Sozialstatus der verbleibenden Charaktere an der Tankstelle hat.

Das Ergebnis ist 5, d.h. gerade noch Glück gehabt. Henry wird nicht verwundet, bleibt somit am Leben und die Infektion breitet sich nicht weiter aus.

3.2.6. Krisen

Die Überlebenden werden jede Runde von einer neuen Krise heimgesucht (d.h. es wird eine neue Krisen-Karte vom Krisen-Kartenstapel gezogen). Für jede Krise muss eine gewisse Anzahl an Karten eines bestimmten Kartentypes geopfert werden. Hierbei spielt der Wert der Karte keine Rolle.

In der Konfigurationsdatei ist spezifiziert:

- die ID der Krise (`identifier`).
- wie viel Moral die Kolonie verliert (`moralChange`), falls sie die Krise nicht meistern, d.h. nicht genug oder nicht die richtigen Karten beigetragen haben
- wie viele Karten beigetragen werden müssen, um die Krise zu meistern (`requiredCards`)
- welche Kartentypen beigetragen werden müssen (`food`, `stuff`, `medicine` oder `fuel`)

3.2.7. Fähigkeiten

Wie wir bereits wissen, besitzt jeder Überlebende genau eine besondere **Fähigkeit**, welche unterteilt sind in **vier aktive** und **vier passive** Fähigkeiten.

Aktive Fähigkeiten müssen aktiv bei Spielzügen eingesetzt werden, wohingegen passive Fähigkeiten automatisch die entsprechenden Aktionen beeinflussen.

Ein Charakter mit der AKTIVEN Fähigkeit

`BARRICADE` kann einfacher Barrikaden erstellen.
`KILL` kann einfacher Zombies töten.
`FEED` kann einfacher bzw. mehr Essen zum Nahrungsvorrat hinzufügen.
`HEAL` kann Verletzungen heilen.

Es gibt insgesamt vier PASSIVE Fähigkeiten:

`NO-INFECTION` hat Einfluss auf Verletzungen, die bei Angriffen passieren können.
`WOUND` hat Einfluss auf Verletzungen, die bei Bewegungen passieren können.
`SEARCH` beeinflusst, wie ein Charakter einen Standort durchsuchen darf.
`TRASH` beeinflusst, wie viel Müll ein Charakter entsorgen kann.

Sie finden alle (technischen) Details zu Fähigkeiten in Tabelle 1 und 2.

3.2.8. Karten

Alle Spielerinnen und Spieler starten mit fünf Handkarten ins Spiel. Man kann an neue Karten rankommen, indem man Standorte durchsuchen (= eine Karte vom Kartenstapel des Standortes zieht). Ausgespielte Handkarten landen sofort auf dem *Müllstapel*.

Es gibt insgesamt **sechs Kartentypen**:

`LOCK`: hat Einfluss darauf, wie einfach ein Eingang verbarrikadiert werden kann
`SCISSORS`: hat Einfluss darauf, wie viele Zombies getötet werden können
`STUFF`: man kann den Aktionswürfel mit der niedrigsten Zahl neu würfeln
`FUEL`: hat Einfluss Bewegungen bzw. auf das Töten von Zombies
`FOOD`: es kann Essen zum Nahrungsvorrat beigesteuert werden
`MEDICINE`: Verletzungen können geheilt werden

3.2.9. Aktionen

Die Spielerinnen und Spieler steuern verschiedene Charaktere (die *Überlebenden* der Krise), die jede Runde Aktionen ausführen können. Sie können z.B. nach Essen und Materialien in der Umgebung suchen, Zombies angreifen oder Standorte verbarrikadieren.

Für manche Aktionen müssen Aktionswürfel eingesetzt und somit verbraucht werden, für andere nicht. Jeder Spieler und jede Spielerin bekommt zu Beginn jeder Runde so viele Aktionswürfel, wie die Gruppe seiner bzw. ihrer Überlebenden groß ist (plus einen extra).

$$\#\text{Aktionswürfel}_{\text{Spieler}} = 1 + \#\text{Überlebende}_{\text{Spieler}}$$

Wie genau ein Spieler seine Aktionen mit den Aktionswürfeln planen kann, wird in Unterabschnitt 3.3.1 genauer erklärt.

In diesem Abschnitt wollen wir zunächst allgemein auf die möglichen Aktionen eingehen, sowie auf den Einfluss von Fähigkeiten auf diese. Die Details zur Konfigurierbarkeit entnehmen Sie bitte den entsprechenden Tabellen.

(1) Aktionen, die *keinen* Aktionswürfel benötigen:

- **Beitrag zur Krise leisten** (= Handkarten für die Krise opfern)
- **Charakter zwischen Standorten bewegen**

  Jeder Charakter kann sich 1x pro Runde bewegen. Da man außerhalb eines Standortes auf ein Zombie treffen könnte, besteht immer ein gewisses Risiko sich Verletzungen zuzufügen (s. Unterabschnitt 3.2.5).

  Einfluss von Fähigkeiten:
  Sollte der Charakter die passive Fähigkeit `WOUND` besitzen, erhält er bei einer Bewegung statt der gewürfelten Verletzung die Art der Verletzung, welche in der Konfigurationsdatei für dessen Fähigkeit spezifiziert ist.

  Hinweis: Mit Ausspielen der Karte `FUEL` kann sich der Charakter fortbewegen ohne sich eine Verletzung zuzuziehen.
- **Ausspielen von Karten und Einsatz von (fast allen) aktiven Fähigkeiten**

  Das Ausspielen von Karten und Nutzen von aktiven Fähigkeiten der Charaktere sind aktive Entscheidungen eines Spielers bzw. einer Spielerin.

  Viele Aktionen können entweder mit der entsprechenden Fähigkeit oder mit der entsprechenden Karte ausgeführt werden:

  - dem Nahrungsvorrat Essen hinzuzufügen
    - durch Ausspielen einer `FOOD`-Karte
    - durch Einsatz der `FEED`-Fähigkeit eines Charakters (1x pro Runde)
  - Verletzungen eines Überlebenden heilen
    - durch Ausspielen einer `MEDICINE`-Karte
    - durch Einsatz der `HEAL`-Fähigkeit (1x pro Runde)
  - Barrikade(n) erstellen
    - auf genau einem Platz eines Eingangs durch Ausspielen einer `LOCK`-Karte
    - durch Einsatz der Fähigkeit `BARRICADE` (stark konfigurierbar)

- Zombies töten
  - genau ein Zombie an einem Eingang durch Ausspielen einer `SCISSORS`-Karte
  - oder durch Einsatz der aktiven Fähigkeit `KILL` (stark konfigurierbar, braucht unter Umständen einen Aktionswürfel)

  Hierbei wollen wir vorab schon einmal erwähnen, dass sowohl das Ausspielen von Karten (`UseCard`-Command), als auch der Einsatz einer Fähigkeit (`UseAbility`-Command) vom Server nicht wie ein Angriff auf Zombies (`attack`-Command) abgehandelt wird. Technische Details dazu folgen in Abschnitt 4.4.
- Mit Ausspielen der Karte `FUEL` kann sich der Charakter fortbewegen ohne sich eine Verletzung zuzuziehen.
- Mit Ausspielen der Karte `STUFF` kann der Spieler bzw. die Spielerin ihren niedrigsten Aktionswürfel erneut würfeln.

(2) Aktionen, die Aktionswürfel benötigen:

- **Müll entsorgen**

  Pro Runde kann ein Spieler auf Kosten eines Aktionswürfels (Augenzahl egal) **drei** Karten vom Müllstapel entfernen.

  Einfluss von Fähigkeiten:
  Falls ein Charakter die passive Fähigkeit `TRASH` besitzt, kann er dabei so viele Karten entsorgen, wie in der Konfigurationsdatei für dessen Fähigkeit angegeben ist. Ein Spieler bzw. eine Spielerin kann mit *jedem* Charakter, der diese Fähigkeit hat, in einer Runde Müll entsorgen.

- **Verbarrikadieren**

  Pro Platz eines Eingangs, den ein Spieler in einer Runde verbarrikadieren will, muss er einen Aktionswürfel (Augenzahl egal) verbrauchen.

  Einfluss von Fähigkeiten:
  An dieser Stelle der Hinweis: Die Aktion *Verbarrikadieren* kann ein Spieler auf Kosten eines Aktionswürfels jederzeit durchführen (sofern es noch freie Plätze an einem Eingang des Standortes, an dem der Spieler steht, gibt).

  Diese Aktion *Verbarrikadieren* (`Barricade`-Command) ist technisch gesehen nicht das gleiche, wie der Einsatz der aktiven Fähigkeit `BARRICADE` eines Charakters (`UseAbility`-Command), um Barrikaden zu erstellen, bzw. die Karte `LOCK` auszuspielen (`UseCard`-Command), um eine Barrikade zu erstellen, auch wenn das Ergebnis natürlich in allen

drei Fällen das gleiche ist: Es wird ein (oder mehrere) Plätze eines Eingangs verbarrikadiert.

- **Zombies angreifen**

Damit ein Charakter an seinem Standort ein Zombie angreifen kann, muss der Spieler bzw. die Spielerin einen Aktionswürfel verbrauchen, mit

$$\text{Augenzahl} \geq \text{Angriffswert}_{\text{Charakter}}$$

Nachdem der Zombie getötet wurde, wird die Verletzung bestimmt, die sich der Charakter durch den Angriff zugezogen haben könnte (s. Tabelle 3.2.5). Im schlimmsten Fall wurde der Überlebende dabei gebissen und stirbt. Ein Zombie stirbt hingegen immer durch einen Angriff.

Einfluss von Fähigkeiten:
Sollte der Charakter die passive Fähigkeit `NO-INFECTION` besitzen, wird kein Infektionswürfel nach dem Angriff geworfen.

- **Standorte durchsuchen**

Durchsuchen bedeutet, dass man die oberste Karte vom Kartenstapel des Standortes zieht. Um einen Standort eines Charakters zu durchsuchen (s. Unterabschnitt 3.2.4), muss der Spieler bzw. die Spielerin einen Aktionswürfel verbrauchen, mit

$$\text{Augenzahl} \geq \text{Durchsuchewert}_{\text{Charakter}}$$

Einfluss von Fähigkeiten:
Sollte der Charakter für diesen Standort die passive Fähigkeit `SEARCH` besitzen, darf der Spieler mehr als eine Karte ziehen. Alle Details zu dieser Fähigkeit s. Tabelle 1.

***Beispiel:* Attackieren eines Zombies**
Ein Spieler steht mit seiner Überlebenden Tina am Standort *Schule* und möchte Zombies angreifen.

Wir nehmen an der Angriffs-Wert von Tina ist 5 $\leq$ der Augenzahl einer der übrigen Aktionswürfel des Spielers.

$\rightarrow$ Der Zombie wird getötet und aus dem Spiel entfernt. Als nächstes wird der Infektionswürfel für Tina gerollt, da sie mit einem Zombie interagiert hat. Das Ergebnis ist 7, d.h. Tina erleidet eine Wunde. Da er Tinas erste Wunde ist, passiert nichts weiter.

3.3. Spielablauf

3.3.1. Spielerphase

Zu Beginn des Spiels bekommen alle Spieler insgesamt fünf *Karten* vom Startkartenstapel. Diese Karten entsprechen verschiedenen Aktionen wie z.B. Essen generieren, einen Überlebenden heilen oder eine Barrikade erstellen. Um während des Spiels an weitere Karten dranzukommen, müssen Standorte durchsucht werden. Die Anzahl der Karten, die ein Spieler auf der Hand haben kann, ist nicht begrenzt.

Nachdem eine Karte ausgespielt wurde, kommt diese auf den Müllstapel.

Alle Kartentypen können in Tabelle 3 und Tabelle 4 nachgelesen werden.

Die Spielerphase läuft dann wie folgt pro Runde ab:

1. In jeder Runde gilt es eine **Krise** zu meistern: Es wird eine neue Krisen-Karte vom Krisen-Kartenstapel des Spiels aufgedeckt.
2. Danach werfen ALLE Spielerinnen und Spieler ihre **Aktionswürfel**. Die Aktionswürfel sind einfache sechsseitige Würfel.[2].

   Es werden insgesamt so viele Würfel geworfen, wie es Spieler und Überlebende im Spiel gibt. Pro Spieler und pro Überlebendem würfelt man ein Mal (Kinder zählen hier nicht dazu), d.h. wenn ein Spieler eine Gruppe von drei Überlebenden hat, so würfelt er insgesamt vier Mal.

   Dabei kann man seine Aktionswürfel beliebig auf Aktionen seiner Überlebenden aufteilen. Es gilt insbesondere, dass wenn ein Charakter eines Spielers bzw. einer Spielerin während der aktuellen Runde stirbt, geht dadurch kein Würfel verloren. Ebenso bekommt man auch keinen neuen hinzu, sollte man in der Runde einen zusätzlichen Charakter erhalten.

Hinweis

Die Anzahl der Würfel, welche man in einer Runde einsetzen kann, ändert sich während der Runde nicht. Sie wird immer nur zu Beginn der Runde an die Anzahl der verbleibenden Charaktere des Spieler bzw. der Spielere angepasst.

   Mit diesen gewürfelten Werten können die Spieler nun ihre Aktionen für diese Runde planen. Hierbei gilt es zu beachten, dass immer der kleinste Aktionswürfel (bzw. die kleinste gewürfelte Zahl) genutzt wird, mit der die Aktion durchgeführt werden kann.

   Man kann grundsätzlich beliebig viele Aktionen pro Runde ausführen. Die Anzahl der

[2]Wie die Würfel implementiert werden sollen, finden sie in Abschnitt 4.2

Aktionen, die ein Spieler mit seinen Überlebenden durchführen kann, ist im Prinzip nur begrenzt durch die Anzahl der Handkarten des Spielers, sowie davon, was einem Charakter in Abhängigkeit seiner Fähigkeiten etc. mit den gewürfelten Aktionswürfeln an Aktionen möglich ist.

3. Anschließend beginnen die Spielerinnen und Spieler in der gleichen Reihenfolge ihre Spielzüge durchzuführen. Sobald ein Spieler fertig mit all seinen Aktionen ist, ist der nächste Spieler dran. Die maximale Anzahl der Spielerinnen und Spieler wird in der Konfigurationsdatei angegeben.

3.3.2. Koloniephase

Sobald alle Spielerinnen und Spieler mit ihren Spielzügen fertig sind, beginnt die Koloniephase. Die Koloniephase handelt nach und nach verschiedene Punkte ab:

1. **Nahrung verteilen:** Für die Menschen in der Kolonie muss Nahrung aufgebracht werden. Die Anzahl der aufzubringenden Nahrung wird bestimmt wie folgt:

$$\#Food = \lceil (\#Survivors_{atColony} + \#Children)/2 \rceil$$

Das benötigte Essen wird dem **Nahrungsvorrat** entnommen. Der Nahrungsvorrat der Kolonie besteht aus den Nahrungsmarkern, welche durch ausgespielten `FOOD`-Karten ins Spiel gekommen sind(s. Tabelle 3), und ist dementsprechend zu Beginn des Spiels leer.

Sollte am Ende einer Runde nicht genug Essen im Nahrungsvorrat vorhanden sein, um ALLE entsprechend der obigen Formel zu versorgen, so wird kein Essen entnommen, sondern dem Nahrungsvorrat stattdessen ein **Hungermarker** hinzugefügt. Pro Hungermarker im Nahrungsvorrat verliert die Kolonie eine Moral pro Runde. Es gibt keine Möglichkeit die Hungermarker wieder zu entfernen, deswegen sollte die Kolonie im besten Fall immer auf ausreichend Nahrung achten.

2. **Abfall überprüfen:** Es wird die Anzahl an Karten auf dem Müllstapel überprüft. Pro 10 Karten auf dem Müllstapel verliert die Kolonie in diesem Schritt eine Moral. Jedes Mal, wenn eine Karte ausgespielt wird, landet sie auf dem Müllstapel. Davon ausgenommen sind Karten, welche zur Krise beigetragen werden. Ein Spieler ohne entsprechende besondere Fähigkeit kann drei Karten pro Runde an Müll entsorgen.

3. **Krise abhandeln:** In diesem Schritt wird überprüft, ob genügend Karten für die Krise beigetragen wurden. Ist dies der Fall, so wurde die Krise erfolgreich abgewandt. Sollten zwei Karten mehr als benötigt der Krise hinzugefügt worden sein, so steigt die Moral um eins. Schlägt die Krise fehl, so sinkt die Moral um den bei der Krise angegebenen Wert.

4. **Zombies ins Spiel bringen:** In diesem Schritt werden neue Zombies gespawnt:

(1) Die Zombies spawnen zuerst an der Kolonie. Die Anzahl der Zombies, die an der Kolonie spawnen, hängt von der Anzahl der Überlebenden in der Kolonie ab:

$$\#SpawningZombies_{Colony} = \lceil (\#Survivors + \#Children)/2 \rceil$$

(2) Danach spawnen Zombies an den anderen Standorten. Die Reihenfolge gibt dabei die `locationID` (aufsteigende Reihenfolge) an. Bei diesen Standorten spawnen so viele Zombies wie sich Überlebende am Standort aufhalten:

$$\#SpawningZombies_{Location} = \#Survivors$$

Beim Spawnen von Zombies werden erst alle freie Plätze eines Eingangs belegt. Hierbei verteilen sich die Zombies gleichmäßig über die Eingänge der Kolonie.

Dabei wird stets bei dem Eingang mit der niedrigsten `entrance` begonnen und dann in aufsteigender Reihenfolge fortgefahren. Sollten mehr Zombies als es Eingänge gibt spawnen müssen, so beginnt man wieder mit dem Eingang mit der kleinsten `entrance`, bis alle Zombies gespawnt sind.

Danach werden (falls vorhanden) Barrikaden entfernt anstatt dass ein Zombie spawnt.

Da nach dem Entfernen der ersten Barrikade wieder ein Platz frei ist, wird dieser Platz zuerst aufgefüllt, bevor eine weitere Barrikade zerstört wird.

Sollten alle Plätze mit Zombies belegt sein und ein weiterer Zombie spawnen, stirbt stattdessen ein Überlebender.

Welcher Überlebender stirbt hängt von dem Sozialstatus des Überlebenden ab. Der Überlebende mit dem geringsten Sozialstatus stirbt zuerst. Kinder haben immer einen geringeren Sozialstatus als andere Überlebende.

5. **Gemeinsames Ziel überprüfen:** Im letzten Schritt der Koloniephase wird das gemeinsame Ziel überprüft und es wird festgestellt, ob das Spiel gewonnen ist.

> ***Ein paar Beispiele für ein gemeinsames Ziel sind:***
>
> → überleben bis die Runden abgelaufen sind
> → eine gewisse Anzahl an Barrikaden erstellen
> → ...

Nach der Koloniephase beginnt eine neue Runde.

3.4. Das Originalspiel

Das diesjährige Spiel ist (sehr stark) inspiriert von "Winter der Toten"(*Dead of Winter*).

Dieses Tutorial Learn to Play Dead of Winter in 16 minutes (YouTube) kann Ihnen helfen einen Überblick über das Spiel zu bekommen, für welches Sie die nächsten Wochen zusammen einen Spielserver entwickeln.

Um die Implementierung des Spielservers einfacher zu gestalten, haben wir das Spiel etwas abgeändert.

- Der *Exposure Die* entspricht unserem Infektionswürfel.
- Die *Helpless Survivors* wurden in unserer Verson zu Kindern.
- Es gibt kein *First Player Token*, der Startspieler bleibt immer gleich.
- Es gibt kein *Noise* - im Originalspiel kann man Krach machen, um z.B. beim Durchsuchen eine weitere Karte zu ziehen. Dafür werden aber Zombies angelockt. Das gibt es in unserer Spielversion so nicht.

Ausdrücklich irrelevant:

- 02:37 bis 02:58 (Secret Objective)
  Es gibt weder das *Secret* noch *Personal Objective* in unserer Version. Es gibt lediglich das gemeinsame Ziel der Kolonie.
- 09:08 bis 10:42 (Nahrungsmarker zur Erhöhung von Aktionswürfel-Werten, Verräter, Regel für das Exil)
- 15:24 bis Ende (Wie Verrat funktioniert)

3.5. FAQ

- **Wieso bekommt man vier Charaktere zur Auswahl, wenn man dann nur zwei benutzt? Wieso kann man sich den Schritt nicht sparen?**

  Stellen Sie sich das so vor: Das Spiel beginnt für die Spieler damit, dass das gemeinsame Ziel vorgelesen bzw. angezeigt wird. D.h. jeder Spieler wird diejenigen Charaktere behalten wollen, deren Fähigkeiten wahrscheinlich zum Erreichen des gemeinsamen Ziels besonders hilfreich sein könnten. Den Spielserver, welchen Sie implementieren, interessiert diese Entscheidungsfreiheit allerdings herzlich wenig :-) Trotzdem muss er diese Funktionalität aber bereitstellen.

- **Wie war das nochmal mit den Würfeln?**
  Alle Aktionswürfel werden zu Beginn der Runde geworfen. Es wird ein Würfel pro Spieler und Überlebenden geworfen, d.h. wenn eine Spielerin fünf Charaktere steuert, hat sie $5 + 1 = 6$ Aktionswürfel für diese Runde.

  Der Infektionswürfel wird immer geworfen, wenn sich ein Charakter normal bewegt (ohne `FUEL`) oder einen Zombie angreift. Sprich: Jedes Mal wenn ein Überlebenden von einem Standort zum nächsten läuft, gibt es eine gewisse Wahrscheinlichkeit einem

Zombie zu begegnen und verletzt zu werden. Diese Wahrscheinlichkeit wird mittels Infektionswürfel modelliert.

- **Woher bekommt man jetzt welche Karten?**
  Zu Beginn des Spiels erhalten alle Spielerinnen und Spieler jeweils *fünf* Karten vom Startkartenstapel. Dieser Stapel ist in der Konfigurationsdatei spezifiziert. Während des Spiels, kann man nur über das Durchsuchen eines Standortes weitere Karten erhalten. Welche Karten an einem Standort sind, ist auch in der Spezifikationsdatei angegeben.
- **Wie viele Karten darf ein Spieler oder eine Spielerin auf der Hand haben?**
  Die Anzahl der Karten, die ein Spieler auf der Hand haben kann, ist nicht begrenzt.
- **Wie viele Aktionen darf denn ein Spieler oder eine Spielerin pro Runde durchführen?**
  Die Anzahl der Aktionen, die ein Spieler mit seinen Überlebenden durchführen kann, ist im Prinzip nur begrenzt durch die Handkarten des Spielers, sowie davon, was die Charaktere, die der Spieler steuert, in Abhängigkeit seiner Fähigkeiten etc. mit den gewürfelten Aktionswürfeln an Aktionen durchführen kann.
- **Welche Karten landen auf dem Müllstapel?**
  Alle von den Spielern und Spielerinnen ausgespielten Handkarten landen auf dem Müllstapel. Die einzige Ausnahme dazu sind Karten, die ein Spieler bzw. eine Spielerin verliert, wenn einer seiner oder ihrer Überlebenden stirbt. Krisen-Karten zählen nicht zu den möglichen Handkarten. Diese Karten dienen nur dazu die Krise pro Runde aufzudecken.
- **Muss jeder Spieler gleich viele Karten zur Krise beitragen?**
  Grundsätzlich nein. Es kann vorkommen, dass die Spieler unterschiedlich viele ihrer Handkarten opfern müssen. Das kommt drauf an, welche Karten benötigt werden und welche Karten die Spieler bzw. Spielerinnen auf der Hand haben. (Hier gilt es im Interesse der Gemeinschaft zu handeln ;-)
- **In welcher Reihenfolge wird gewürfelt?**
  Die Spielerphase beginnt damit, dass in der Reihenfolge ihrer Anmeldung beim Server die Aktionswürfel aller Spielerinnen und Spieler geworfen werden. Technische Details folgenden in Kapitel 4.
- **Was sind nochmal die Möglichkeiten, wie neue Charaktere ins Spiel kommen können?**
  Beim Durchsuchen von Standorten kann es zu Zufallsbegegnungen kommen, bei denen ein Charakter gespawnt wird. Ebenfalls kommen neue Charaktere ins Spiel, nachdem der letzte Charakter eines Spielers bzw. einer Spielerin gestorben ist. Neue Charaktere spawnen *immer* in der Kolonie. Technische Details siehe Kapitel 4.
- **Wie oft pro Runde kann sich ein Charakter bewegen?**
  Jeder Charakter darf sich pro Runde nur einmal bewegen. Insbesondere kann sich ein Charakter mit der `FUEL`-Karte nicht zusätzlich bewegen. Sie kann nur für eine Bewegung eingesetzt werden.

- **Kann es auch Kinder an Standorten geben?**
  Nein, Kinder existieren nur in der Kolonie. Falls man beim Standort durchsuchen auf Kinder trifft (s. Unterabschnitt 4.2.6), spawnen diese in der Kolonie und bleiben dort. Bei einer Zufallsbegegnung spawnen immer 0-3 Kinder.

- **Wenn am Ende der Runde überprüft wird, ob noch genug Nahrung da ist um alle zu versorgen, verliert dann die Kolonie eine Moral pro nicht-versorgtem Überlebendem?**
  Nein. Wenn am Ende der Runde nicht genug Essen für alle *in der Kolonie befindlichen Überlebenden* (inkl. Kinder) vorhanden ist, dann wird dem Nahrungsvorrat genau 1 Hungermarker hinzugefügt, egal wie viel Nahrung gefehlt hat. Gleichzeitig wird kein Nahrungsmarker aus dem Nahrungsvorrat entfernt. (*Sprich: Entweder alle werden versorgt, oder alle müssen hungern.*) Danach wird pro Hungermarker im Nahrungsvorrat eine Moral abgezogen. Die Hungermarker bleiben bis zum Ende des Spiels im Nahrungsvorrat und kosten jede Runde eine Moral.

- **Ist die Kolonie ein Standort?**
  Ja, allerdings gibt es keine Kartenstapel zum Durchsuchen und die Kolonie hat keine Platzbeschränkung für Überlebende.

- **Wenn mein Charakter die `KILL`-Fähigkeit hat...muss ich dann jedes Mal den Infektionswürfel werfen, wenn er ein Zombie tötet? Das wäre ja voll unpraktisch.**
  Das wäre sehr unpraktisch, genau. Und deswegen ist wichtig zu verstehen: `KILL` zählt NICHT als Angriff in dem Sinne, somit muss nur ein Infektionswürfel geworfen werden, wenn die Konfigurationsdatei das so spezifiziert.

4. Technische Details

In diesem Kapitel erklären wir Ihnen weitere technische Details – vom Build-Skript, über Codenanalysetools bis hin zu technischen Implementierungsdetails. Hier erklären wir Ihnen auch das Framework, das Ihnen von uns zur Verfügung gestellt wird und von Ihnen benutzt werden muss.

4.1. Spielablauf

Das Spiel ist in verschiedene Phasen aufgeteilt. Es beginnt mit der *Registrierungsphase*, auf welche die *Vorbereitungsphase* und schließlich das eigentliche Spiel folgt. Im Folgenden werden die einzelnen Phasen im Detail beschrieben.

4.1.1. Ziel

Für jedes Spiel wird in der Konfigurationsdatei (Abschnitt 4.3) ein **gemeinsames Ziel** definiert. Dieses Ziel gibt an, welche Bedingungen erfüllt sein müssen, damit das Spiel gewonnen ist. Außerdem gibt es an, mit wie vielen Zombies und mit wie vielen hilflosen Kindern ins Spiel gestartet wird.

4.1.2. Registrierungsphase

Während der Registrierungsphase wartet der Server darauf, dass sich alle Spielerinnen und Spieler registrieren, die am Spiel teilnehmen wollen. Dabei wird jedem Spieler bzw. Spielerin eine `playerID` zugeordnet, welche bei 0 startet und immer um 1 inkrementiert wird.

Während dieser Phase kann ein bereits registrierter Spieler das Spiel starten, auch wenn noch nicht die maximale Anzahl an Spielerinnen und Spielern registriert ist.

Während der Registrierungsphase führt ein Timeout zum Beenden des Servers. Sonst wird ein Timeout so behandelt, als hätte der Server einen `leave()`-Command empfangen.

4.1.3. Vorbereitungsphase

Die Vorbereitungsphase startet, nachdem entweder die maximale Anzahl an Spielerinnen und Spielern erreicht ist oder wenn ein registrierter Spieler das Spiel startet. Sobald die Phase startet, teilt der Server zuerst allen Clienten mit, welche Spieler mitspielen (`Player`-Events).

Anschließend werden für jeden Spieler (in aufsteigender Reihenfolge der `playerId`) vier Überlebende gezogen und mitgeteilt. Davon werden dann vom Spieler jeweils zwei ausgewählt. Danach werden für den gleichen Spieler noch die fünf Handkarten gezogen. Anschließend geht es mit der nächsten Spielerin weiter. Sobald alle Spielerinnen und Spieler abgehandelt wurden, beginnt das eigentliche Spiel.

4.2. Zufallsberechnungen

Um unser Spiel trotz der vielen Zufallsberechnungen deterministisch zu halten, nutzen wir für alle Aktionen, welche auf Zufall beruhen, den *java.util.Random*-Zufallszahlengenerator, welchen wir mit einem gegeben `seed` initialisieren.

Warum ist das wichtig?

"Wird der seed-Parameter übergeben, initialisiert der Zufallszahlengenerator seinen internen Zähler mit diesem Wert, und die anschließend erzeugte Folge von Zufallszahlen ist reproduzierbar. Wird dagegen der parameterlose Konstruktor aufgerufen, initialisiert er den Zufallszahlengenerator auf der Basis der aktuellen Systemzeit. Die Zufallszahlenfolge ist in diesem Fall nicht reproduzierbar." [1]

Damit alle Berechnungen mit der korrekten Zahl aus dieser initialisierten, zufälligen Zahlenfolge stattfinden, ist es unerlässlich, dass Sie sich genau an die spezifizierte Reihenfolge und Häufigkeit der Zufallsberechnungen halten.

4.2.1. Karten

Bei der Initialisierung des Spiels nutzen wir das `Random`-Objekt zunächst, um die Karten zu mischen.

Dabei mischen wir zuerst den Startkartenstapel und anschließend die Stapel der anderen Standorte in aufsteigender Reihenfolge ihrer ID's.

Wir mischen die Karten, indem wir *Collections.shuffle()* benutzen:

```
Collections.shuffle(cards, random);
```

Es wird immer die erste Karte vom gemischten Stapel während des Spiels gezogen.

4.2.2. Krisen

Nachdem die Kartenstapel gemischt wurden, müssen auch die Krisen gemischt werden. Diese werden ebenfalls mit *Collections.shuffle()* gemischt.

```
Collections.shuffle(cards, random);
```

[1] https://dbs.cs.uni-duesseldorf.de/lehre/docs/java/javabuch/html/k100105.html

Es wird auch hier während des Spiels immer die erste Karte vom Krisen-Kartenstapel gezogen.

4.2.3. Überlebende

Nachdem die Krisen-Karten gemischt wurden, werden zufällig Überlebende ausgewählt, die ins Spiel gebracht werden sollen. Dies realisieren wir wie folgt:

1. Bevor für einen Spieler die Charaktere gezogen werden, mischen wir alle Charaktere durch:

```
Collections.shuffle(survivors, random);
```

2. Wir wählen die ersten vier Charaktere aus.
3. Wir fügen die beiden vom Spieler nicht ausgewählten Charaktere wieder hinten an die Menge der ungenutzten Charaktere an.
4. Wir machen mit dem nächsten Spieler weiter.

Nachdem alle Spieler ihre Charaktere ausgewählt haben, mischen wir alle Charaktere noch ein letztes Mal durch (wie in Schritt 1.). **Ab jetzt wird immer der erste Charakter vom Beginn der Liste genommen, wenn ein neuer Charakter ins Spiel kommt.**

Es ist wichtig, *Collections.shuffle()* immer mit dem `Random`-Objekt aufzurufen!

Hinweis

Die Charaktere werden in der Reihenfolge des Einlesens aus dem JSON-Schema in einer Liste gespeichert. Der Zustand der geshuffelten Liste ist ja abhängig vom initialen Zustand der Liste, d.h. damit der Zustand des Spiels deterministisch in Abhängigkeit des Seeds, müssen logischerweise auch die Objekte aus den Schemata in der vorgegebenen Reihenfolge eingelesen werden.

4.2.4. Aktionswürfel

Aktionswürfel sind ganz normale sechsseitige Würfel. Um diese zu rollen, rufen wir die *nextInt()*-Methode des `Random`-Objektes wie folgt auf:

```
random.nextInt(6) + 1;
```

4.2.5. Infektionswürfel

Der Infektionswürfel ist ein zwölfseitiger Würfel, bei welchem allerdings keine Zahlen, sondern Verwundungen geworfen werden. Um dies zu simulieren, nutzen wir wieder die *random.nextInt()*-Methode:

```
random.nextInt(12);
```

Für das Ergebnis n gilt:

- $0 \leq n < 6$: **keine** Verwundung
- $6 \leq n < 9$: eine **Wunde**
- $9 \leq n < 11$: eine **Erfrierung**
- 11 : ein **Biss**

4.2.6. Zufallsbegegnungen

Charaktere können nicht nur zu Beginn des Spiels, sondern auch durch Zufallsbegegnungen ins Spiel kommen. Zufallsbegegnungen können immer beim Durchsuchen eines Standortes auftreten. Ob es zu einer Zufallsbegegnung kommt und ob dabei neben dem Charakter auch noch Kinder ins Spiel kommen, wird wie folgt berechnet:

1. Überprüfe, ob noch Karten auf dem Kartenstapel des Standortes vorhanden sind.
    - Falls nicht, schlägt das Durchsuchen fehl und die Zufallsbegegnung findet auch nicht statt.
2. Überprüfe, ob schon alle Charaktere im Spiel sind bzw. ob es noch ungenutzte Charaktere gibt.
    - Falls es keine ungenutzten Charaktere mehr gibt, findet keine Zufallsbegegnung statt und das Durchsuchen wird normal durchgeführt.
3. Überprüfe, ob es zu einer Zufallsbegegnung kommt:
    - Um dies zu tun, nehmen wir wieder eine zufällige Zahl mit *random.nextInt()*, wobei wir als obere Grenze die Anzahl der Karten, die initial auf dem Kartenstapel waren, nehmen.

      Ist die Zufallszahl eine 0, kommt es zu einer Zufallsbegegnung, d.h. der erste Charakter, der noch nicht im Spiel befindlichen Charaktere, wird ausgewählt

und spwant sofort in der Kolonie. Danach wird nun wieder *random.nextInt()* aufgerufen mit 4 als oberer Grenze. Das Ergebnis dieses Methodenaufrufes gibt an, wie viele Kinder ins Spiel kommen. Diese spawnen dann auch direkt in der Kolonie.

Damit ist das Durchsuchen beendet und es wird keine Karte mehr gezogen.

- Falls es zu keiner Zufallsbegegnung kommt, wird normal mit dem Durchsuchen (= Ziehen einer Karte) weitergemacht.

***Beispiel:* Durchsuchen ohne Zufallsbegegnung**
Eine Spielerin ist mit ihrer Überlebenden Alice am Standort *Kino*, deren Kartenstapel zwei Karten (von initial 7) enthält.

Wir nehmen an, die Augenzahl einer der übrigen Aktionswürfel der Spielerin ist $\geq$ Durchsuchewert von Alice. Es sind außerdem Charaktere übrig, die noch nicht im Spiel sind.

$\rightarrow$ Die Spielerin würfelt nun mit einem 7-seitigen Würfel. Das Ergebnis ist 3, es kommt also zu keiner Zufallsbegegnung und die Spielerin zieht eine Karte vom Kartenstapel des Kinos. Damit ist das Durchsuchen beendet.

***Beispiel:* Durchsuchen mit Zufallsbegegnung**
Ein Spieler ist mit seiner Überlebenden Bob am Standort *Bibliothek*, deren Kartenstapel noch eine Karten (von initial 5) enthält.

Wir nehmen an der Durchsuchewert von Bob ist $\leq$ der Augenzahl einer der übrigen Aktionswürfel des Spielers. Es sind außerdem Charaktere übrig, die noch nicht im Spiel sind.

$\rightarrow$ Der Spieler würfelt nun mit einem 5-seitigen Würfel. Das Ergebnis ist 0, es kommt also zu einer Zufallsbegegnung. Zuerst wird ein neuer Charakter gezogen, welcher in der Kolonie spawnt. Danach würfelt der Spieler mit einem 4-seitigen Würfel. Das Ergebnis ist 2, es spawnen jetzt also noch 2 Kinder in der Kolonie und das Durchsuchen ist beendet.

4.3. Konfigurationsdatei

Anstatt alle Spielparameter bereits während der Implementierung festzulegen, werden wir stattdessen beim Start des Spielservers eine Konfigurationsdatei einlesen, die das Spiel zu großen Teilen modelliert. Mit dieser wird bespielsweise das Ziel des Spiels, die eigentliche Spielwelt, aber auch Charaktere und deren Fähigkeiten vorgegeben.

Als Notation verwenden wir JSON[2]. Zusätzlich geben wir Schemata[3] vor, die den Aufbau und gegebenfalls Wertebereiche für Felder der Konfigurationsdatei bestimmen.

Es ist sinnvoll sich im Detail mit diesen Formaten auseinanderzusetzen, da hier bereits eine Vielzahl von Aspekten des Spiel definiert werden.

Vorraussetzungen für eine gültige Konfigurationsdatei:

- Es müssen ausreichend viele Überlebende vorhanden sein:

  $$\#Survivors \geq maxPlayers * 2 + 2$$

- Es müssen genügend Karten pro Standort vorhanden sein:

  $$\#Cards_{Location} \geq maxPlayers * 5$$

- Es muss mindestens so viele Krisen-Karten wie Runden geben.
- Es müssen genügend Startkarten vorhanden sein:

  $$\#StartCards \geq maxPlayers * 5$$

- Keine zwei Charaktere dürfen den gleichen Sozialstatus haben.
- Die `ID` einer Karte muss *einzigartig* sein, und eine Karte darf auch nur in genau einem Stapel max. 1 Mal vorkommen.

Hinweis: Anforderungen an die Konfigurationsdatei entsprechend nicht unbedingt den Anforderungen an den Spielzustand im Allgemeinen.

Beispiel einer Konfigurationsdatei:

```
{
  "cards": [
    {
      "food": {
        "identifier": 3,
        "amount": 3
      }
    },
```

[2] https://www.json.org
[3] https://json-schema.org

```
    {
      "hammer": {
        "identifier": 1
      }
    }
  ],
  "characters": [
    {
      "name": "Carla Thompson",
      "identifier": 1,
      "status": 22,
      "attack": 4,
      "search": 2,
      "ability": {
        "search": {
          "location": 1,
          "numCards": 1
        }
      }
    }
  ],
  "goal": {
    "rounds": 10,
    "moral": 7,
    "zombiesColony": 6,
    "zombiesLocations": 1,
    "survive": true,
    "childrenInColony": 4
  },
  "locations": [
    {
      "colony": {
        "identifier": 42,
        "entrances": 6,
        "startCards": [
          1,
          2,
          3,
          4,
          5
        ]
      }
    },
    {
```

```
        "name": "Police Station",
        "identifier": 1,
        "entrances": 1,
        "survivorSpaces": 3,
        "cards": [
          10105,
          10106,
          10107,
          10108,
          10010,
          10011
        ]
      }
    ]
}
```

4.4. Client-Server Kommunikation

4.4.1. Commands & Events

Das Spiel wird von einem Server gesteuert, welcher mit den Clienten (Spielern) mittels *Commands & Events* kommuniziert. *Commands* werden von den Clients an den Server geschickt um ihm mitzuteilen was der Spieler machen will. Der Server nutzt wiederum *Events* um den Clients mitzuteilen, was sich im Spiel geändert hat. Bei den Events unterscheiden wir zwischen individuellen Events und Broadcast Events. Individuelle Events werden nur an den Spieler geschickt den es betrifft, während Broadcast Events an alle Spieler geschickt werden müssen. Die CommLib sendet Events immer nur an genau einen Spieler, sprich der Server muss dafür sorge tragen, dass Broadcast Events auch tatsächlich an alle geschickt werden. Um dies zu tun, hat jeder Spieler (neben seiner ID) eine CommId. Diese wird automatisch von der CommLib vergeben und muss auf die SpielerId gemapped werden. Alle Commands bekommen hierfür die CommId des ausführenden Spielers als Parameter übergeben. Für das Versenden der Events wird ebenfalls die CommId als Parameter benötigt.

Alle Commands, welche von nicht-registrierten Clients geschickt werden, werden während dem Spiel ignoriert.

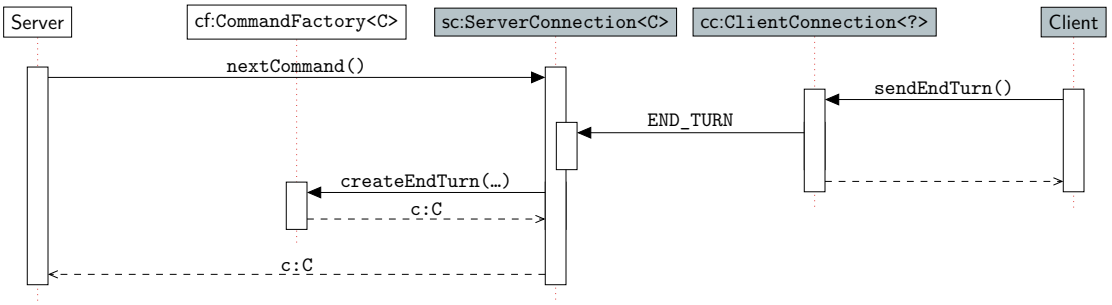

Abbildung 1: Serverseitige Interaktion der `CommandFactory` mit der `ServerConnection`.

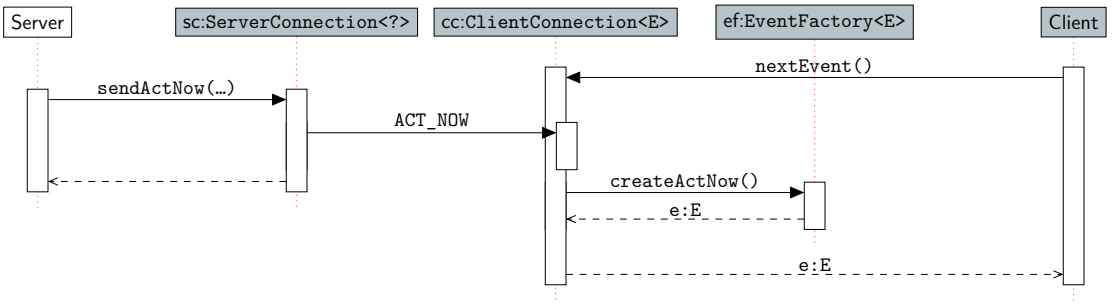

Abbildung 2: Clientseitige Interaktion der `EventFactory` mit der `ClientConnection`.

4.5. Command-Line-Interface

Der Server wird mit diesen Kommandozeilenparametern gestartet:

`--config <path>` Der Pfad zur Datei, aus welcher die Konfiguration im *JSON*-Format geladen werden soll.

`--port <int>` Der Port, auf welchem der Client mit dem Server kommunizieren kann.

`--seed <long>` Der Seed, mit dem der Random-Number-Generator des Spiels initialisiert wird (s. Zufallsberechnungen).

`--timeout <int>` Der Timeout des Servers in Sekunden (die maximale Zeit, die der Server auf Commands des Clients wartet).

4.6. Build-Skript

Wir stellen Ihnen ein Build-Skript zur Verfügung. Ein Build-Skript ist dafür zuständig, die Abhängigkeiten eines Softwareprojekts zu verwalten und aus den Projektdateien ein fertiges, ausführbares Programm zu erstellen. Als Build-Tool verwenden wir *Gradle*[4].

[4] https://gradle.org/

Das Build-Skript finden Sie in der Datei `build.gradle`. Änderungen an Ihrem Build-Skript werden vom Testserver für die Ausführung zurückgesetzt. Sie können (und sollten) das Projekt selbst bauen, indem Sie entweder den Befehl `./gradlew build` ausführen oder den *Gradle*-Task `build` direkt in einer IDE (z.B. *IntelliJ*) ausführen. Wir verwenden *Java 16*. Im Build-Skript sind Abhängigkeiten auf diverse Bibliotheken eingetragen. Wir empfehlen Ihnen, sich diese Bibliotheken anzusehen und zu überlegen, ob Sie diese verwenden können. Weitere Bibliotheken, die nicht im Build-Skript eingetragen sind, dürfen nicht verwendet werden. Aus Sicherheitsgründen erlauben wir keine Reflection, keine Netzwerkverbindungen oder sonstige Verbindungen zur Außenwelt mit Ausnahme der verwendeten *CommLib* (s. Client-Server Kommunikation). Außerdem dürfen Sie – mit Ausnahme des Ladens von Konfigurationsdateien – keine Dateien im Dateisystem lesen, anlegen oder verändern. Für das Laden von Konfigurationsdateien darf lediglich auf das bereits existierende `resources`-Verzeichnis im Projekt-Repository zugegriffen werden.

4.7. Codequalität

Gute Codequalität hilft dabei, dass Software lesbar bleibt und Fehler besser vermieden werden können. Im Software-Praktikum werden Sie automatisierte Werkzeuge kennenlernen, die Sie dabei unterstützen, guten und fehlerfreien Code zu schreiben. Sie können diese Tools ebenfalls mit *Gradle* nutzen.

Checkstyle Dieses Tool überprüft, ob Sie sich an den von ihnen spezifizierbaren Style-Guide halten. Wir liefern Ihnen eine Code-Style Config mit, die Sie anpassen oder unverändert nutzen können.

PMD ist ein statisches Analysewerkzeug, was bedeutet, dass es Ihren Code nicht ausführen muss, um Fehler zu finden. PMD untersucht den Quellcode nach bestimmten Mustern, von denen bekannt ist, dass sie Fehler verursachen. Beispiele für solche Muster sind unerreichbarer Code (wahrscheinlich haben Sie vergessen eine Methode aufzurufen) oder der Vergleich von zwei Strings mit == (in Java müssen Strings mit `equals()` auf Gleichheit geprüft werden). Der Bericht lässt sich mit einem Webbrowser öffnen und listet alle Problemstellen und Links zu den genauen Problembeschreibungen auf.

Sie werden feststellen, dass *PMD* es nicht zulässt `System.(out|err).println` zu verwenden. Solche Konsolenausgaben sollten nur während des Debuggens genutzt werden. Sie können stattdessen das Logging-Frameworks *SLF4J*[5] verwenden, das stärker konfigurierbar ist.

SpotBugs SpotBugs ist auch ein statisches Analysewerkzeug, aber es arbeitet nicht direkt auf dem Quellcode, sondern auf dem kompilierten Java-Bytecode. Dadurch kann es problematische Codestellen finden, die von *PMD* übersehen werden, wie z.B. wenn bestimmte Ausdrücke immer den Wert `null` haben. Auch hier gibt es einen Fehlerbericht, in welchem auf die Problematik hingewiesen wird.

[5] https://www.slf4j.org/manual.html

5. Tests

In der Gruppenphase erwarten wir System-, Unit- und Integrationstests von Ihnen. Die Anforderungen, welche an Ihre Anwendung gestellt werden, können Sie den Spezifikationen aus den vorherigen Kapiteln entnehmen.

Versuchen Sie in der Implementierungsphase eine möglichst hohe Statement- und Branch-Coverage zu erreichen. Das heißt, dass in Ihrem Code so viele Statements und Branches wie möglich von den Tests abgedeckt sein sollen. Die Coverage kann z.B. mit *JaCoCo*[1] errechnet werden, auch *IntelliJ* hat bereits integrierte Tools hierfür.

5.1. Unit-Tests

Verwenden Sie für die Unit-Tests *JUnit 5*[2] und *Mockito*[3]. Diverse Tutorials für die Verwendung dieser Frameworks finden Sie im Internet[4]. Wir werden Ihnen zu Anfang der Implementierungsphase einen Beispieltest zur Verfügung stellen.

Die Unit-Tests können mit `./gradlew test` (bzw. `gradlew.bat test`) ausgeführt werden.

Implementierungshinweis

Unit-Tests gehören in das vorgegebene Verzeichnis `src/test/java` im Projektordner.

5.2. Integrationstests

Es ist Ihre Aufgabe, auch Integrationstests zu schreiben, um die Zusammenarbeit der Module zu testen. In Integrationstests wird ähnlich zu den Unit-Tests auf einzelne Methoden zugegriffen. Grundsätzlich können Sie diese Tests wie Unit-Tests schreiben und ausführen.

[1] https://www.jacoco.org/
[2] https://junit.org
[3] https://site.mockito.org
[4] https://junit.org/junit5/docs/current/user-guide/

5.3. Systemtests

Es müssen auch Systemtests geschrieben werden, die den Spiel-Server ausreichend auf die gesamte Funktionalität testen. Das Framework bietet dieses Mal die Möglichkeit für unterschiedliche Systemtests.

Wir stellen Ihnen ein Framework zur Verfügung, das Sie zu Beginn der Implementierungsphase zusammen mit der *CommLib* erhalten. Das Framework übernimmt dabei die Kommunikation mit dem Server.

Ihre Systemtests werden außerdem auch auf der Referenzimplementierung ausgeführt, sodass Sie festellen können, ob Ihr Annahme über das Verhalten des Servers richtig ist.

Beachten Sie, dass Ihre Systemtests auch auf fehlerhaften Spiel-Servern (Mutanten) ausgeführt werden, also durch Ihre Tests fehlerhafte Implementierungen gefunden werden müssen.

Implementierungshinweis

Systemtests gehören in das vorgegebene Verzeichnis `src/systemtest/java` im Projektordner.

A. Anhang

A.1. Fähigkeiten

Tabelle 1: Passive Fähigkeiten

`WOUND`
: Ein Überlebender mit dieser Fähigkeit erhält ggf. eine andere Art der Verletzung bei einer Bewegung (falls der Infektionswürfel eine Verletzung ergibt). Welche Art der Verletzung (`wound, frostbite, bite`) in welche andere Verletzung umgewandelt wird, wird in der Konfigurationsdatei frei gewählt (`before/after`).

`NO-INFECTION`
: Ein Überlebender mit dieser Fähigkeit kann einen Angriff durchführen, ohne sich zu verletzen (= ohne, dass nach dem Angriff der Infektionswürfel geworfen wird).

`SEARCH`
: Eine Überlebende mit dieser Fähigkeit kann beim Durchsuchen eines Standortes mehr als eine Karten ziehen und behalten. Die Fähigkeit gilt für einen bestimmten Standort, welcher in der Konfigurationsdatei spezifiziert ist (`locationId`). Außerdem kann die Anzahl der Karten (`numCards`), welche gezogen werden können, variieren. Die Fähigkeit wird automatisch aktiviert, wenn die Überlebende den spezifizierten Standort durchsucht und kann so oft genutzt werden, wie in der Konfigurationsdatei spezifiziert ist (`maxActivations`). Nachdem die maximale Anzahl an Aktivierungen der Fähigkeit erfolgt ist, gelten beim weiteren Durchsuchen von Standorten die üblichen Regeln. Sollten an dem Standort weniger Karten übrig sein, als mit dieser Fähigkeit gezogen werden könnten, werden einfach alle verbleibenden Karten gezogen.

Tabelle 1: Passive Fähigkeiten

TRASH

Ein Überlebender mit dieser Fähigkeit kann beim Müll entsorgen (und somit nur 1x pro Runde) eine andere Anzahl an Karten entsorgen als üblich. Die genau Anzahl der Karten wird in der Konfigurationsdatei angegeben (`numCards`).

Tabelle 2: Aktive Fähigkeiten

HEAL

Mit dieser Fähigkeit kann ein Überlebender einer anderen Überlebenden am gleichen Standort eine Wunde oder Erfrierung heilen. Hierbei werden zuerst Erfrierungen und dann Verwundungen geheilt. Die zu heilende Überlebende wird hierbei durch die `characterId` als `target` angegeben. Diese Fähigkeit kann max. 1 Mal pro Runde eingesetzt werden.

FEED

Mit dieser Fähigkeit kann eine Überlebende dem Nahrungsvorrat eine gewisse Menge an Nahrungsmarkern hinzufügen. Die genaue Menge an Nahrungsmarkern, die hinzugefügt werden, ist frei konfigurierbar und muss in der Konfigurationsdatei angegeben werden (`numFood`). Diese Fähigkeit kann max. 1 Mal pro Runde eingesetzt werden.

BARRICADE

Eine Überlebende mit dieser Fähigkeit kann an einen Eingang des Standortes, an dem sie steht, eine Barrikade erstellen. Wie häufig diese Fähigkeit in einer Runde eingesetzt werden kann, wird in der Konfigurationsdatei angegeben (`maxActivations`). Ebenso wird dort angegeben, wie viele Plätze eines Eingangs durch den Einsatz der Fähigkeit verbarrikadiert werden (`numBarricades`). Das `target` ist in dem Fall die `entrance` des zu verbarrikadierenden Eingangs.

Tabelle 2: Aktive Fähigkeiten

`KILL`

Eine Überlebende mit dieser Fähigkeit kann Zombies einfacher töten als andere Überlebende. Die Konfigurationsdatei gibt an:

- an welchem Standort diese Fähigkeit aktiv ist (`locationId`),
- ob ein Aktionswürfel benötigt wird und wenn ja, wie hoch die Augenzahl sein muss (`dieValue`),
- wie viele Zombies durch das Aktivieren der Fähigkeit getötet werden (`numZombies`),
- ob der Infektionswürfel gewürfelt werden muss (`infectionDie`),
- ob Kinder in der Kolonie sein müssen, um die Fähigkeit zu aktivieren (`children`),
- und wie oft die Fähigkeit in einer Runde aktiviert werden kann (`maxActivations`).

A.2. Karten

Tabelle 3: Karten ohne Ziel

`FOOD`

Mit dieser Karte kann dem Nahrungsvorrat Essen hinzugefügt werden. Jede `FOOD`-Karte gibt eine Menge an Essen an, welches dem Nahrungsvorrat in Form von *Nahrungsmarkern* durch Ausspielen der Karte hinzugefügt werden. Ebenso kann diese Karte zu einer Krise, welche `FOOD` benötigt, beigetragen werden. Beim Beitragen zur Krise zählt jede Karte nur als eine Karte, unabhängig von der Menge an Essen, welche sie beim Ausspielen dem Nahrungsvorrat hinzugefügt hätte.

`FUEL`

Mit dieser Karte können Überlebende eine Bewegung durchführen, ohne den Infektionswürfel werfen zu müssen. Ebenso kann diese Karte zu einer Krise, welche `FUEL` benötigt, beigetragen werden. Falls eine Spielerin oder ein Spieler mehr als eine `FUEL`-Karte besitzt, wird die `FUEL`-Karte mit der geringsten `id` verwendet.

Tabelle 3: Karten ohne Ziel

`STUFF`
: Mit dieser Karte kann der Spieler seinen niedrigsten Aktionswürfel neu würfeln. Ebenso kann diese Karte zu einer Krise beigetragen werden, welche `STUFF` benötigt.

Tabelle 4: Karten mit Ziel (`target`)

`MEDICINE`
: Eine Karte heilt eine Wunde oder Erfrierung eines Überlebenden, welcher als `target` angegeben wird. Da Erfrierungen die gefährlicheren Wunden sind, werden bei der Anwendung dieser Karte zuerst Erfrierungen geheilt. Ebenso kann diese Karte zu einer Krise beigetragen werden, welche `MEDICINE` benötigt.

`LOCK`
: Mit dieser Karte kann ein Überlebender genau einen Platz eines Eingangs seines aktuellen Standortes verbarrikadieren. Dabei muss die `entrance`, an dem eine Barrikade aufgestellt werden soll, als `target` angegeben werden. Ebenso kann diese Karte zu einer Krise beigetragen werden, welche `STUFF` benötigt.

`SCISSORS`
: Mit dieser Karte kann ein Überlebender einen Zombie an einem Eingang seines Standortes töten. Dabei muss die `entrance`, an dem das Zombie getötet werden soll, als `target` angegeben werden. Sollte an diesem Eingang kein Zombie stehen, so schlägt das Ausspielen der Karte fehl. Ebenso kann diese Karte zu einer Krise beigetragen werden, welche `STUFF` benötigt. Ausspielen der Karte `SCISSORS` zählt NICHT als Angriff, somit wird hierbei auch kein Infektionswürfel geworfen.

A.3. Commands und Events

Tabelle 5: Administrative Commands

Administrative Commands
`Register(String playerName)` Das `Register`-Command wird von einem Client gesendet, um sich am Spiel anzumelden. Dabei gibt er auch seinen Namen an. Sollte ein registrierter Spieler versuchen sich zu registrieren, wird ein `CommandFailed` gesendet. Kann gesendet werden, ohne dass vorher ein `ActNow()` empfangen wird.
`StartGame()` Mit diesem Command teilt die Spielerin dem Server mit, dass das Spiel starten soll. Dieses Command kann genutzt werden um das Spiel zu starten, bevor die maximale Anzahl der Spielerinnen erreicht ist. Wird dieses Command gesendet, wenn das Spiel schon gestartet ist, wird ein `CommandFailed` gesendet. Kann gesendet werden, ohne dass vorher ein `ActNow` empfangen wird.
`SelectCharacters(int characterId0, int characterId1)` Mit diesem Command teilt der Spieler dem Server mit, mit welchen Charakteren er ins Spiel starten will. Dieses Command wird als Antwort zum `Characters`-Command erwartet.

Tabelle 6: In-Game Commands

In-Game Commands
`Leave()` Mit diesem Command teilt ein Spieler dem Server mit, dass er das Spiel verlassen möchte. Daraufhin wird der Spieler aus dem Spiel entfernt und alle seine Charaktere werden getötet (angefangen bei dem Spieler mit dem höchsten Sozialstatus und dann weiter in absteigender Reihenfolge). ~~Es muss nicht zwingend vorher ein `ActNow()` empfangen werden.~~ Ein `Leave`-Command darf nur gesendet werden, wenn der aktuelle Spieler an der Reihe ist. Ansonsten schlägt das Command fehl.

Tabelle 6: In-Game Commands

In-Game Commands

`UseAbility(int characterId)`

Mit diesem Command teilt die Spielerin dem Server mit, dass sie die Ability vom Charakter mit der angegebenen `characterId` nutzen möchte. Ist dies nicht möglich, schlägt das Command fehl und es wird ein `CommandFailed` gesendet.

`UseAbility(int characterId, int target)`

Mit diesem Command teilt die Spielerin dem Server mit, dass sie die Ability vom Charakter mit der angegebenen `characterId` nutzen möchte. Das Ziel der Ability wird mit der `target` angegeben. Ist dies nicht möglich, oder sollten sich der Charakter und das Ziel nicht am gleichen Standort befinden, so schlägt das Command fehl und es wird ein `CommandFailed` gesendet.

`Barricade(int characterId, int entrance)`

Mit diesem Command teilt der Spieler dem Server mit, dass er mit dem angegebenen Charakter eine Barrikade am angegebenen Eingang errichten will. Ist dies nicht möglich, schlägt das Command fehl und es wird ein `CommandFailed` gesendet.

`EndTurn()`

Mit diesem Command teilt die Spielerin dem Server mit, dass sie in dieser Runde keine weiteren Aktionen mehr durchführen möchte. Ihr Zug ist damit beendet. Sollte die Spielerin, welche das Kommand sendet nicht die aktive Spielerin sein, so wird ein `CommandFailed` gesendet und das Spiel normal fortgesetzt.

`Move(int characterId, int locationId, boolean fuel)`

Mit diesem Command teilt der Spieler dem Server mit, dass er den angegebenen Charakter an den angegebenen Standort bewegen möchte. Der Wert von `fuel` gibt an, ob der Spieler eine Benzin-Karte ausspielt, andernfalls wird für den Charakter, nachdem er zu seinem neuen Standort bewegt wurde, der Infektionswürfel geworfen. Pro Runde, kann jeder Charakter maximal einmal seinen Standort wechseln. Sollte das Command nicht ausgeführt werden können (z.B. weil kein Platz am Zielstandort ist, der Charakter sich schon bewegt hat, oder weil der Spieler nicht über ausreichend Benzin verfügt), wird ein `CommandFailed` gesendet.

Tabelle 6: In-Game Commands

In-Game Commands

`UseCard(int cardId)`

Mit diesem Command teilt die Spielerin dem Server mit, dass sie die angegebene Karte ausspielen möchte. Sollte sie die Karte nicht haben, oder nicht am Zug sein, wird ein `CommandFailed` gesendet.

`UseCard(int cardId, int characterId, int target)`

Mit diesem Command teilt die Spielerin dem Server mit, dass sie die angegebene Karte, mit dem angegebenen Charakter ausspielen möchte. Dabei wird auch ein Ziel spezifiziert, welches den Effekt der Karte abbekommen soll. Diese Version des Commands wird verwendet, um Charaktere zu heilen und sich mit einer entsprechenden Karte zu verbarrikadieren. Sollten sich der Charakter und das Ziel nicht am gleichen Standort befinden, so schlägt das Command fehl. Ebenso schlägt das Command fehl, sollte die Spielerin die angegebene Karte nicht haben, oder nicht am Zug sein. In beiden Fällen wird ein `CommandFailed` gesendet.

`ContributeCard(int cardId)`

Mit diesem Command teilt der Spieler dem Server mit, dass er die angegebene Karte der Kriese beisteuern möchte. Sollte er die Karte nicht haben, oder nicht am Zug sein, wird ein `CommandFailed` gesendet.

`CleanWaste(int characterId)`

Mit diesem Command teilt die Spielerin dem Server mit, dass sie mit dem angegebenen Charakter den Müll entsorgen möchte. Sollte der Charakter nicht in der Kolonie ein, oder die Spielerin keine Aktionswürfel mehr übrig haben, oder kein Müll vorhanden sein, so schlägt das Command fehl und es wird ein `CommandFailed` gesendet.

Tabelle 6: In-Game Commands

In-Game Commands
`Attack(int characterId, int entrance)` Mit diesem Command teilt der Spieler dem Server mit, dass er mit dem angegebenen Charakter einen Zombie am angegebenen Eingang angreifen möchte. Nachdem der Zombie getötet wurde, wird ein Infektionswürfel für den angreifenden Charakter geworfen. Sollte der Spieler mit dem angegebenen Charakter nicht angreifen können, weil er keinen Aktionswürfel mit einer Augenzahl $\geq$ dem Angriffswert des Chrakters übrig hat, kein Zombie am angegebenen Eingang steht, oder der Spieler nicht an der Reihe ist, so schlägt das Command fehl und es wird ein `CommandFailed` gesendet.
`Search(int characterId)` Mit diesem Command teilt die Spielerin dem Server mit, dass sie mit dem angegebenen Charakter den Standort des Charakters durchsuchen möchte. Sollte die Spielerin mit dem angegebenen Charakter den Standort nicht durchsuchen können, weil sie keinen Aktionswürfel mit einer Augenzahl $\geq$ dem Durchsuchenwert des Charakters übrig hat, keine Karten auf dem Kartenstapel des Standorts liegen, der Standort keinen Kartenstapel hat (außer die Kolonie) oder die Spielerin nicht an der Reihe sein, so schlägt das Command fehl und es wird ein `CommandFailed` gesendet.

Tabelle 7: Events

Individuelle Events (werden an genau einen Client geschickt)
`Config(String config)` Mit diesem Event sendet der Server dem Client die Konfigurationsdatei des Spiels als String. Es wird dem Client direkt nach einem erfolgreichen `Registration`-Command geschickt.

Tabelle 7: Events

Individuelle Events (werden an genau einen Client geschickt)

`ActNow()`
Mit diesem Event sendet der Server dem Client, dass er an der Reihe ist und der Server ein neues Command erwartet. Das Event wird jedes Mal geschickt, wenn ein Command erwartet wird. Eine Spielerin die keine Überlebenden mehr hat, kann keine Spielzüge durchführen und wird somit diese Event nichtmehr erhalten.

`Characters(int characterId0, int characterId1, int characterId2, int characterId3)`
Mit diesem Event sendet der Server dem Client die Auswahl an Charakteren aus denen sich die Spielerin zwei aussuchen muss, mit denen sie ins Spiel starten will.

`CommandFailed(String message)`
Mit diesem Event teilt der Server dem Client mit, dass das letzte gesendete Command fehlgeschlagen ist. Die `message` gibt hierbei den Grund für das Fehlschlagen an. *Hinweis: Wir testen nicht explizit, den Inhalt der Nachricht. Diese darf allerdings nicht leer sein.*

Tabelle 8: Events

Broadcast Events (werden an alle Clients geschickt.)

`RegistrationAborted()`
Dieses Event benachrichtigt den Client darüber, dass der Server einen ungültigen Command während der Anmeldephase bekommen hat und sich deshalb beendet. Ein ungültiges Command während der Anmeldephase ist jedes Command, welches nicht ein `Registration`-Command oder `StartGame`-Command ist.

`Player(int player, String playerName)`
Nach der Anmeldephase sendet der Server allen Clients dieses Event um sie über die Spielerinnen zu informieren, welche am Spiel teilnehmen. Dabei werden die Events in aufsteigender Reihenfolge der `player` versendet.

Tabelle 8: Events

Broadcast Events (werden an alle Clients geschickt.)

`CharacterSpawned(int player, int characterId)`
Mit diesem Event teilt der Server dem Client mit, dass ein neuer Charakter für den angegebenen Spieler gespawned ist. Spawnen mehrere Charaktere gleichzeitig, so werden die Events in aufsteigender Reihenfolge der `characterId` gesendet.

`ZombieSpawned(int locationId, int entrance)`
Mit diesem Event teilt der Server dem Client mit, dass ein Zombie am angegebenen Standort und dem angegebenen Eingang gespawned ist. Das Event wird direkt nach dem Spawnen des Zombies gesendet.

`ChildSpawned()`
Mit diesem Event teilt der Server dem Client mit, dass ein Kind gespawned ist.

`CardDrawn(int player, int cardId)`
Mit diesem Event teilt der Server dem Client mit, dass die angegebene Spielerin die Karte mit der angegebnen `cardId` gezogen hat. Dieses Event wird auch zu Beginn verschickt, wenn die Karten zum Spielstart verteilt werden.

`GameEnd(boolean win)`
Mit diesem Event teilt der Server dem Client mit, dass das Spiel geendet ist. Der Parameter `win` gibt dabei an, ob das Spiel gewonnen oder verloren wurde.

`GameStarted()`
Mit diesem Event teilt der Server dem Client mit, dass das Spiel gestartet ist.

`NextRound(int round)`
Mit diesem Event teilt der Server dem Client mit, dass eine neue Runde gestartet ist. `round` gibt dabei die Rundennummer an. Die erste Runde hat die Nummer 1.

Tabelle 8: Events

Broadcast Events (werden an alle Clients geschickt.)

`Crisis(int crisisId)`
Mit diesem Event teilt der Server dem Client mit, dass in dieser Runde die Krise mit der ID `crisisId` erfüllt werden muss. Das `Crisis`-Event ist das erste Event mach dem `NextRound`-Event.

`DieRolled(int player, int value)`
Mit diesem Event teilt der Server dem Client mit, dass ein Würfel für den angegebenen Spieler geworfen wurde. `value` steht hierbei für den Wert des geworfenen Würfels. Die `DieRolled`-Events werden nach dem `Crisis`-Event in aufsteigender Reihenfolge der SpielerId verschickt.

`AbilityUsed(int characterId)`
Mit diesem Event teilt der Server dem Client mit, dass der angegebene Charakter aktiv seine Fähigkeit benutzt hat. Dieses Event wird nur in dem Fall verschickt, indem die Fähigkeit kein Ziel hat.

`AbilityUsed(int characterId, int target)`
Mit diesem Event teilt der Server dem Client mit, dass der angegebene Charakter aktiv seine Fähigkeit benutzt hat. Dieses Event wird in dann verschickt, wenn die benutzte Fähigkeit einen ein Ziel benötigt, auf das sie angewandt wird.

`Barricaded(int characterId, int locationId, int entrance)`
Mit diesem Event teilt der Server dem Client mit, dass der angegebene Eingang am angegebenen Standort vom Charakter mit der angegebenen `characterId` verbarrikadiert wurde.

`Moved(int characterId, int locationId, boolean fuel)`
Mit diesem Event teilt der Server dem Client mit, dass sich der angegebne Charakter an den Standort mit der entsprechenden `entrance` bewegt hat. `fuel` gibt hierbei an, ob Benzin für die Bewegung genutzt wurde.

Tabelle 8: Events

Broadcast Events (werden an alle Clients geschickt.)

`Frostbitten(int characterId)`
Mit diesem Event teilt der Server dem Client mit, dass der angegebene Charakter eine Erfrierung erlitten hat. Dieses Event kann direkt im Anschluss an eine Bewegung ohne Benzin oder eine Attacke gesendet werden.

`Wounded(int characterId)`
Mit diesem Event teilt der Server dem Client mit, dass der angegebene Charakter eine Wunde erlitten hat. Dieses Event kann direkt im Anschluss an eine Bewegung ohne Benzin oder eine Attacke gesendet werden.

`Bitten(int characterId)`
Mit diesem Event teilt der Server dem Client mit, dass der angegebene Charakter gebissen wurde und somit stirbt. In diesem Fall kommt es zu einem Ausbruch, welcher abgehandelt werden muss, bevor das Spiel fortgesetzt werden kann. Dieses Event kann direkt im Anschluss an eine Bewegung ohne Benzin oder eine Attacke gesendet werden. Außerdem wird es gesendet, wenn beim Ausbreiten einer Infektion ein Überlebender stirbt.

`Bitten()`
Dieses Event wird gesendet, falls beim Ausbreiten einer Infektion ein Kind gebissen wird und stirbt. Es kommt zu einem Ausbruch, der abgehandelt werden muss, bevor das Spiel fortgesetzt werden kann.

`Contributed(int player, int cardId)`
Mit diesem Event teilt der Server dem Client mit, dass der Spieler die angegebene Karte zur Krise beigetragen hat. Diese Event wird nach dem `ContributeCard`-Command geschickt.

`CardUsed(int cardId)`
Mit diesem Event teilt der Server dem Client mit, dass die angegebene Karte benutzt wurde. Dieses Event wird nach dem `UseCard`-Command gesendet.

Tabelle 8: Events

Broadcast Events (werden an alle Clients geschickt.)

`CardUsed(int cardId, int characterId, int target)`
Mit diesem Event teilt der Server dem Client mit, dass die angegebene Karte vom Charakter mit der angegebenen `characterId` auf das angegebene Ziel angewandt wurde. Dieses Event wird nach dem `UseCard`-Command gesendet.

`FoodChanged(int amount, FoodChange reason)`
Mit diesem Event teilt der Server dem Client mit, dass sich die Anzahl an vorhandenem Essen in der Kolonie um `amount` geändert hat. Das Enum `reason` gibt hierbei an, warum sich die Anzahl an Essen geändert hat.

`ColonyPhaseStarted()`
Mit diesem Event teilt der Server dem Client mit, dass die Koloniephase begonnen hat. Dieses Event wird gesendet, nachdem die letzte Spielerin der Runde ihren Zug beendet hat.

`StarvationTokenAdded()`
Mit diesem Event teilt der Server dem Client mit, dass ein Hungermarker dem Nahrungsvorrat hinzugefügt wurde. Dieses Event wird anstelle dem `FoodChanged`-Event verschickt, falls nicht genug Nahrung im Nahrungsvorrat ist.

`MoralChanged(int amount, MoralChange reason)`
Mit diesem Event teilt der Server dem Client mit, dass sich die Moral um `amount` geändert hat. Das Enum `reason` gibt hierbei an, warum sich die Moral geändert hat. Falls die Moral dadurch auf 0 gesunken ist, wird das Spiel beendet.

`SurvivorKilled(int characterId)`
Mit diesem Event teilt der Server dem Client mit, dass der angegebene Charakter von einem Zombie angegriffen und getötet wurde. Dieses Event wird nur dann geschickt, wenn ein Zombie an einem Eingang spawnen sollte, dort aber kein Platz mehr war.

Tabelle 8: Events

Broadcast Events (werden an alle Clients geschickt.)

`ChildKilled()`
Mit diesem Event teilt der Server dem Client mit, dass ein Kind von einem Zombie angegriffen und getötet wurde. Dieses Event wird nur dann geschickt, wenn ein Zombie an einem Eingang spawnen sollte, dort aber kein Platz mehr war.

`BarricadeDestroyed(int locationId, int entrance)`
Mit diesem Event teilt der Server dem Client mit, dass eine Barrikade am angegebenen Standort und Eingang zerstört wurde.

`WasteChanged(int amount)`
Mit diesem Event teilt der Server dem Client mit, dass sich die Menge an Müll geändert hat. Die neue Menge an Müll beträgt `amount`. Diese Event wird immer dann gesendet, wenn eine Karte ausgespielt wurde (welche auf den Abfallstapel kommt) und wenn Müll entsorgt wurde.

`ZombieKilled(int characterId, int locationId, int entrance)`
Mit diesem Event teilt der Server dem Client mit, dass der angegebene Charakter einen Zombie am angegebenen Standort und Eingang angegriffen und getötet hat.

`Searched(int characterId, int locationId)`
Mit diesem Event teilt der Server dem Client mit, dass der angegebene Standort des angegebenen Charakters durchsucht wurde. Auf dieses Event folgen `CardDrawn`-Events.

`Left(int player)`
Mit diesem Event teilt der Server dem Client mit, dass der angegebene Spieler das Spiel verlassen hat.