

NO  
TIME  
TO  
SPY

---

# Standardisierungsdokument

---

Standardisierungsdokument für das Softwaregrundprojekt 2019/20

16. März 2020

**Bearbeitung:**

Florian Sihler

# NO TIME TO SPY

– 15. Dezember 2019 –

Dieses Layout wurde für das Softwaregrundprojekt im Wintersemester 2019/2020 von Florian Sihler im Rahmen des Informatikstudiums an der Universität Ulm vollständig in  $\text{\LaTeX}$  2<sub>ε</sub> erstellt.

# Inhaltsübersicht

*Hinweis: Diese Inhaltsübersicht liefert nur einen groben Überblick. Der Inhaltsreiter des PDF-Ansichtprogramms sollte eine feingranularere Gliederung enthalten.*

<b>1</b>	<b>Allgemeines</b>	<b>5</b>
	» Grundlegende Informationen, wie dieses Dokument zu lesen ist.	
1.1	Der Zweck dieses Dokuments . . . . .	5
1.2	Das Standardisierungskomitee . . . . .	5
1.3	Formales . . . . .	5
1.3.1	Anforderungen an eine Standarddefinition . . . . .	6
1.3.2	Begriffsdefinitionen . . . . .	6
<b>2</b>	<b>Dateien</b>	<b>11</b>
	» Definiert alle verpflichtenden und optionalen Dateien, samt Bezeichner und Format.	
2.1	Allgemeines . . . . .	11
2.1.1	Dateiformat und Zeichensatz . . . . .	11
2.2	Partiekonfiguration . . . . .	11
2.3	Szenariobeschreibung . . . . .	13
2.4	Charakterdefinition . . . . .	14
<b>3</b>	<b>Datentypen</b>	<b>16</b>
	» Hier sammeln sich alle geforderten Datentypen, die sich nicht einer spezifischen Kategorie zuordnen lassen.	
3.1	Eigenschaften und Charakteristika . . . . .	16
3.2	Die Gadgets . . . . .	17
3.3	Das Szenario . . . . .	18
3.4	Die Partiekonfiguration . . . . .	21
3.5	Der Spielablauf . . . . .	22
3.5.1	Ausführbare Operationen . . . . .	22
3.5.2	Fehlerbehandlung . . . . .	24
3.5.3	Spielende . . . . .	25
<b>4</b>	<b>Charaktere</b>	<b>26</b>
	» Die Ausprägungen und Eigenschaften eines Charakters. Hierunter fallen sowohl die Spieler als auch die NPCs.	
4.1	Eigenschaften eines Charakters . . . . .	26
4.1.1	Die PCs . . . . .	26
4.1.2	Die NPCs . . . . .	27
<b>5</b>	<b>Netzwerkstandard</b>	<b>28</b>
	» Definiert alle Nachrichten und die zugehörigen Aktionen die zwischen Client und Server auszutauschen sind. Hierzu zählt auch die optionale Replay-Funktion.	
5.1	Grundlegendes . . . . .	28
5.1.1	Die Abläufe . . . . .	28
5.1.2	Blaupause . . . . .	28
5.2	Die Nachrichten . . . . .	29
5.2.1	Kategorisierung . . . . .	29

5.2.2	Spielinitialisierung . . . . .	29
5.2.3	Spielstart . . . . .	31
5.2.4	Wahlphase . . . . .	31
5.2.5	Ausrüstungsphase . . . . .	32
5.2.6	Spielphase . . . . .	33
5.2.7	Spielende . . . . .	34
5.2.8	Kontrollnachrichten . . . . .	35
5.2.9	Strikes . . . . .	36
5.3	Optionale Komponenten . . . . .	37

## **6 Schnittstellen** **41**

» Setzt die Zugänglichkeiten für die Programme um eine einheitliche Variante des Startens von KI, Server, ... zu gewährleisten.

6.1	Kommandozeile . . . . .	41
6.1.1	Server . . . . .	41
6.1.2	KI-Client . . . . .	42

## **A Versionierung** **43**

A.1	Changelog . . . . .	43
A.1.1	Changelog für Version 1 (20. Februar 2020) . . . . .	43
A.1.2	Changelog für Version 2 (16. März 2020) . . . . .	45

## Allgemeines

### 1.1 Der Zweck dieses Dokuments

V. 1

Dieses Dokument bildet den für alle Teams des Softwaregrundprojekts 2019/2020 verpflichtenden Standard.

Ziel ist eine teamübergreifende Schnittstelle die das Kombinieren beliebiger Softwarekomponenten sowie das Zusammenspielen verschiedener Clients ermöglichen soll, ohne die Produktqualität jeweils einzuschränken. Modifikationen am Standard sind nur durch Bestimmung im Plenum des Standardisierungskomitees möglich, weitere Anmerkungen zur Dokumentgestaltung oder Verbesserungsvorschläge jeglicher Art an [florian.sihler@uni-ulm.de](mailto:florian.sihler@uni-ulm.de); Danke!

### 1.2 Das Standardisierungskomitee

Das Komitee bildet sich aus *Vertretern* aller 27 Teams, den Vorsitz hält *Jonas Mayer*, das Amt des stellvertretenden Vorsitzenden bekleidet *Eduard Seelig*.

Für weitere Informationen besteht die Mailing-Liste, die sich über das zugehörige GitLab-Repository beziehen lässt.

### 1.3 Formales

Das Dokument enthält eingebettete Dokumente, die sich leider nicht in allen PDF-Readern öffnen lassen (funktionieren sollte es auf jeden Fall mit dem Acrobat Reader, Evince und Okular). Für diesen Fall befinden sich alle eingebetteten, relevanten, Dokumente ebenfalls in den Quellen des Standardisierungskomitees und können, sofern gewünscht, von dort aus bezogen werden.

#### **Kommentar 1:** Versionierung Autor: Florian Sihler.

information  
(OK)

V. 1

So wie dieser Kommentar werden alle Änderungen an und in diesem Dokument versioniert und somit festgehalten. Im Anhang des Dokuments befindet sich ein zugehöriger Changelog über den alle Änderungen und Errata eingesehen werden können, die anderen Felder für beispielsweise Definitionen von Datentypen sollten sich logisch über die Benennung einordnen lassen.

#### **Kommentar 2:** JSON Schema Autor: Florian Sihler.

information  
(OK)

Da die von den Teams zur Verfügung gestellten Informationen und Ausführungen zu den Anforderungen an das zugrundeliegende JSON stark schwanken konstruiere ich alle JSON Schema Dokumente von Grund auf (*draft-07*). Fehler, Unstimmigkeiten, Verständnisprobleme oder andere Hinweise (die entstehen recht schnell, da ich initial auch nicht zu viel Zeit darauf verwenden möchte), jederzeit!

### 1.3.1 Anforderungen an eine Standarddefinition

Es gibt bereits eine Reihe an Namenskonventionen die beschlossen wurden:

- Enumerationen werden mit einem „Enum“-Suffix benannt (`FieldStateEnum`, ...)
- Der Bezeichner „Id“ wird immer mit einem kleinen „D“ geschrieben, es heißt also `playerId` und nicht `playerID`.
- Alle Felder die vom Typ `UUID` sind und somit ein Objekt referenzieren, tragen das Suffix „Id“, so wie `MessageContainer::playerId`. Gleiches gilt für Listen an derartigen Bezeichner, an sie wird das Suffix „Ids“ angefügt, so heißt es `RequestItemChoiceMessage::offeredCharacterIds`.
- Bei Netzwerknachrichten bezeichnet `REQUEST_` immer eine Anfrage, die vom gleichen Typ ohne das `REQUEST`-Präfix beantwortet wird. So zum Beispiel `MessageTypeEnum::REQUEST_ITEM_CHOICE` als Anfrage und `MessageTypeEnum::ITEM_CHOICE` als zugehörige Antwort. Die *einzige* Ausnahme ist beim verlassen eines Spiels, da es sich hierbei nicht um eine Anfrage handelt, wird hier ein `MessageTypeEnum::GAME_LEAVE`-Nachricht gesendet die mit einem `MessageTypeEnum::GAME_LEFT` beantwortet wird.
- Die Benennung aller hier genannter Datentypen dient nur zur Übersicht eine Implementation oder Adaption in andere Programmiersprachen ist jederzeit möglich und auch so gedacht. In diesem Kontext gilt es zu Verstehen, dass wenn von `null` gesprochen wird, die *Abwesenheit* eines Eintrags gemeint ist, dies sich in C-basierten Sprachen wie C++ beispielsweise durch einen `nullptr` behandeln lässt, auf dieser Konvention aber beliebige sprachinterne Regelungen möglich sind.

### 1.3.2 Begriffsdefinitionen

#### o Begriffsdefinition 1: Container

Autor: Marios Sirtmatsis.

verpflichtend  
(OK)

• v. 1

*Beschreibung:* Plattformunabhängiges Wrapping der jeweiligen Softwarekomponente. Dabei kann es sich sowohl um die KI, den Server, den Editor als auch den Client handeln. Ziel des „Wrappens“ ist die einfache Initialisierung der unterschiedlichen Komponenten.

*Ist ein:* Plattformunabhängiges Wrapping einer Softwarekomponente.

*Kann sein:*

*Aspekt:* Wird benötigt um bestimmte Softwarekomponenten zu initialisieren.

*Beispiele:* Starten der KI, des Servers, des Editors, des Clients.

#### o Begriffsdefinition 2: Docker

Autor: Marios Sirtmatsis.

verpflichtend  
(OK)

• v. 1

*Beschreibung:* Docker dient zur Erstellung von Containern, welche eine Anwendung aber auch alle ihre Ressourcen, welche zur Laufzeit benötigt werden, enthält.

*Ist ein:* Tool zum erstellen von Containern.

*Kann sein:*

*Aspekt:* Wird benötigt um bestimmte Softwarekomponenten in Containern zu verpacken.

*Beispiele:* Erstellen eines KI-Containers, Client-Containers, ...

#### o **Begriffsdefinition 3:** JSON

Autor: Marios Sirtmatsis.

verpflichtend  
(OK)

• V. 1

*Beschreibung:* Ein Datenformat in einfach lesbarer Form, welches für das Projekt zum Datenaustausch verwendet wird.

*Ist ein:* Datenformat.

*Kann sein:* Charakterbeschreibung, Szenariobeschreibung, ...

*Aspekt:* Notwendig für die einheitliche Übermittlung von Nachrichten.

*Beispiele:* Nachricht zum Betreten einer Lobby, zum Spielstart oder Verlassen eines Spiels, ...

#### o **Begriffsdefinition 4:** KI-Client

Autor: Marios Sirtmatsis.

verpflichtend  
(OK)

• V. 1

*Beschreibung:* Die KI kann über einen Container gestartet werden. Sie soll selbstständig mit der Software interagieren und nach vorgegebenen Kommandozeilenargumenten gestartet werden können.

*Ist ein:* Teil der zu erstellenden Software (Pflicht).

*Kann sein:*

*Aspekt:* Teil der zu erstellenden Software (Pflicht).

*Beispiele:*

#### o **Begriffsdefinition 5:** Strike

Autor: Marios Sirtmatsis.

verpflichtend  
(OK)

• V. 1

*Beschreibung:* Ein Strike ist eine Antwort des Servers, wenn der Benutzer zu lange keine Aktion tätigt.

*Ist ein:* Hinweismnachricht.

*Kann sein:*

*Aspekt:* Das Erkennen von Strikes ist essentiell um inaktive Nutzer zu erkennen.

*Beispiele:* Timeout.

#### o **Begriffsdefinition 6:** Phasen des Spiels

Autor: Marios Sirtmatsis.

verpflichtend  
(OK)

• V. 1

*Beschreibung:* Das Spiel unterteilt sich in drei Phasen: erstens: die Wahlphase, zweitens: die Ausrüstungsphase und zuletzt: die Spielphase.

*Ist ein:* Phase des Spiels.

*Kann sein:*

*Aspekt:* Notwendig um Spiel in sinnvolle Phasen zu unterteilen.

*Beispiele:* Phasen bereits oben genannt.

#### o **Begriffsdefinition 7:** Spielstart

Autor: Marios Sirtmatsis.

verpflichtend  
(OK)

• V. 1

*Beschreibung:* Durch einen Nachrichtenaustausch wird dem Client mitgeteilt, dass das Spiel beginnt, wodurch der Client die Partie-Konfiguration, die Charaktere und das Szenario.

*Ist ein:* Phase für den Beginn eines Spiels.

*Kann sein:*

*Aspekt:* Notwendig, um ein Spiel zu beginnen.

*Beispiele:*

○ **Begriffsdefinition 8:** Wahlphase

Autor: Marios Sirtmatsis.

verpflichtend  
(OK)

• V. 1

*Beschreibung:* In der Wahlphase bekommt der Client eine Reihe an Charakteren und Gadgets zur Auswahl, aus denen er wählen und seine Fraktion zusammenstellen kann.

*Ist ein:* Phase des Spiels.

*Kann sein:*

*Aspekt:* Verteilung der Charaktere und Gadgets auf die Spieler.

*Beispiele:* Der Spieler bekommt drei Charaktere und drei Gadgets zur Auswahl und wählt eine der Möglichkeiten solange, bis all seine acht Slots nach den Anforderungen belegt sind.

○ **Begriffsdefinition 9:** Ausrüstungsphase

Autor: Marios Sirtmatsis.

verpflichtend  
(OK)

• V. 1

*Beschreibung:* Die Ausrüstungsphase beginnt direkt nach der Wahlphase. Hierbei müssen in der Ausrüstungsphase die gewählten Gadgets auf die gewählten Charaktere verteilt werden.

*Ist ein:* Phase für die Ausrüstung der Charaktere mit initialen Gadgets.

*Kann sein:*

*Aspekt:* Notwendig, um initiale Gadgets für die Charaktere zu haben.

*Beispiele:* Dem Charakter James Bond wird ein `GadgetEnum` : JETPACK zugewiesen.

○ **Begriffsdefinition 10:** Spielphase

Autor: Marios Sirtmatsis.

verpflichtend  
(OK)

• V. 1

*Beschreibung:* Die Spielphase beginnt, sobald die Ausrüstungsphase abgeschlossen ist. Ein Spiel läuft in Runden ab, wobei in jeder Runde jeder Charakter einen Spielzug ausführen kann.

*Ist ein:* (Haupt-)Phase des Spiels.

*Kann sein:*

*Aspekt:* Teil der Spielablaufs.

*Beispiele:* Die beiden Spieler sind bereit und dürfen in zufälliger Reihenfolge für ihre Charaktere Züge ausführen.

○ **Begriffsdefinition 11:** Gadget

Autor: Marios Sirtmatsis.

verpflichtend  
(OK)

• V. 1

*Beschreibung:* Ein Gadget ist ein Gegenstand, welcher dazu dient während der Aktionsphase eine Gadget Aktion auszuführen.

*Ist ein:* Verwendbarer oder passives Gegenstand.

*Kann sein:* Ausprägungen aus `GadgetEnum`.

*Aspekt:* Teil der Spielregeln.

*Beispiele:* Der Spieler besitzt als Gadget einen Klingen-Hut (`GadgetEnum` : BOWLER\_BLADE), den er mit einer Gadget-Aktion werfen kann.

○ **Begriffsdefinition 12:** Playable Character

Autoren: Jonas Mayer, Florian Sihler.

verpflichtend  
(OK)

• V. 1



*Beschreibung:* Ein *playable character* (PC) ist eine Spielfigur die unmittelbar durch einen Mitspieler (die KI eingeschlossen) beeinflusst werden kann.

*Ist ein:* Kontrollierbarer Teil des Spiels.

*Kann sein:*

*Aspekt:* Teil der Spielregeln, anderweitig wäre der Spielgedanke hinsichtlich des Konzepts ziemlich nichtig.

*Beispiele:* Master Yoda, James Bond, Opossum Frieda, ...

#### o **Begriffsdefinition 13:** Non-Playable Character

Autoren: Marios Sirtmatsis, Jonas Mayer, Florian Sihler.

verpflichtend  
(OK)

• V. 1

*Beschreibung:* Ein *non-playable character* (NPC) ist eine Spielfigur die nicht durch einen Mitspieler (die KI eingeschlossen) beeinflusst werden kann und so direkt vom Server gesteuert wird. Da man sie durch das `GadgetEnum`: :NUGGET auch rekrutieren kann, drücken sie sich als ganz normale *Character* aus, für die lediglich *keine* `MessageTypeEnum`: :REQUEST\_GAME\_OPERATION vom Server eingeht. Hiervon weichen die im `NPCEnum` aufgeführten besonderen NPCs ab, welche nicht rekrutiert werden können und auch sonst *eine* besondere Rolle einnehmen.

*Ist ein:* Vom Server gesteuertes Spielelement.

*Kann sein:* Playable Character (unter Einschränkungen)

*Aspekt:* Teil der Spielregeln, im Fall der Katze sogar zielführend.

*Beispiele:* Diese weichen nicht von denen eines playable characters ab, hinzuzufügen sind die Katze und der Hausmeister.

#### o **Begriffsdefinition 14:** Szenario

Autor: Marios Sirtmatsis.

verpflichtend  
(OK)

• V. 1

*Beschreibung:* Ein Szenario beschreibt die Anordnung des Spielfeldes. Hierbei wird definiert, welche Feldtypen an welcher Stelle des Spielbretts sind. Die Feldtypen werden im `FieldStateEnum` definiert.

*Ist ein:*

*Kann sein:*

*Aspekt:* Darstellung des Spielfeldes.

*Beispiele:* Siehe Dateidefinition zur Szenariobeschreibung.

#### o **Begriffsdefinition 15:** Feld

Autor: Marios Sirtmatsis.

verpflichtend  
(OK)

• V. 1

*Beschreibung:* Ein Feld (`Field`) ist ein Teil des Szenarios (`Scenario`) und damit des Spielfelds.

*Ist ein:*

*Kann sein:* Ausprägungen von `FieldStateEnum`.

*Aspekt:* Notwendig, damit das Spielfeld eindeutig dargestellt werden kann.

*Beispiele:* Das Feld XY hat den Zustand `FieldStateEnum`: :ROULETTE\_TABLE.

#### o **Begriffsdefinition 16:** Charakter

Autor: Marios Sirtmatsis.

verpflichtend  
(OK)

• V. 1

*Beschreibung:* Ein Charakter ist symbolisch eine zu spielende Figur des Spiels, welche verschiedene Eigenschaften besitzen, Aktionen ausführen und sich auf dem jeweiligen Spielfeld bewegen kann.

*Ist ein:* Klasse, beziehungsweise Teil der Spielmechanik.

*Kann sein:*

*Aspekt:* Teil der Spielmechanik.

*Beispiele:* Der Charakter James Bond besitzt die `PropertyEnum::NIMBLENESS` Eigenschaft, hat einen `GadgetEnum::COCKTAIL` und die Koordinaten  $p$  (`Point`). Außerdem hat James Bond eine bestimmte Anzahl an Bewegungs-, Aktions-, Gesundheits-, Intelligence-Punkte und Spielchips.

o **Begriffsdefinition 17:** Operation

Autor: Marios Sirtmatsis.

verpflichtend  
(OK)

• v. 1

*Beschreibung:* Jeder Charakter auf dem Spielfeld kann eine Operation ausführen. Es gibt verschiedene Operationsarten, die im `OperationEnum` definiert werden.

*Ist ein:*

*Kann sein:*

*Aspekt:* Teil der Spielmechanik.

*Beispiele:* Der Charakter James Bond führt die Operation `OperationEnum::MOVEMENT` aus und bewegt sich um 1 Feld.

o **Begriffsdefinition 18:** Item

Autor: Marios Sirtmatsis.

verpflichtend  
(OK)

• v. 1

*Beschreibung:* In der Wahlphase werden Charaktere und Gadgets zur Vereinfachung als Items zusammengefasst.

*Ist ein:* Item.

*Kann sein:* Gadget oder Charakter in der Wahlphase des Spiels.

*Aspekt:* Vereinfachung für die Wahlphase.

*Beispiele:* Der User wählt aus Gadgets und Charakteren insgesamt acht Items für das Spiel aus.

## Dateien

### 2.1 Allgemeines

#### 2.1.1 Dateiformat und Zeichensatz

Alle folgenden Dateien sind im *UTF-8* Format unkomprimiert, verschlüsselt oder signiert zu halten. Es ist im Falle des JSON-Codes anzunehmen, dass der zugehörige JSON-Parser keinerlei Unterstützung für Kommentare bietet, da ein solches Verhalten im `MessageContainer` bisher auch noch nicht vorgesehen ist.

Da es keine offizielle Dateiendung für JSON-Schema formate gibt, enden die in diesem Dokument enthaltenen Schemas die Endung `.schema` eine Änderung in `.schema.json` oder ähnlichem ist jederzeit möglich und frei vom Rest des Dokuments.


### 2.2 Partiekonfiguration

#### Datei Definition 1: Partiekonfiguration

Autor: Florian Sihler.

verpflichtend  
(OK)

● v. 1

Es ist eine Konfigurationsdatei `<Konfigurationsname>.match` zu halten, deren Struktur ein *Objekt* von `Matchconfig` aller Einstellungen wie in der Beispieldatei ist. Alle Wahrscheinlichkeiten rangieren hierbei zwischen 0 und 1, die Beschreibungen lassen sich dem JSON Schema entnehmen. Eine Beispieldatei ist über  `matchconfig.match` gegeben.

```
1 {
2   "Moledie_Range": 1,
3
4   "BowlerBlade_Range": 1,
5   "BowlerBlade_HitChance": 0.25,
6   "BowlerBlade_Damage": 4,
7
8   "LaserCompact_HitChance": 0.125,
9
10  "RocketPen_Damage": 2,
11
12  "GasGloss_Damage": 6,
13
```

```
14     "MothballPouch_Range": 2,
15     "MothballPouch_Damage": 1,
16
17     "FogTin_Range": 2,
18
19     "Grapple_Range": 3,
20     "Grapple_HitChance": 0.35,
21
22     "WiretapWithEarplugs_FailChance": 0.64,
23
24     "Mirror_SwapChance": 0.35,
25
26     "Cocktail_DodgeChance": 0.25,
27     "Cocktail_Hp": 6,
28
29     "Spy_SuccessChance": 0.65,
30
31     "Babysitter_SuccessChance": 0.25,
32
33     "HoneyTrap_SuccessChance": 0.35,
34
35     "Observation_SuccessChance": 0.12,
36
37     "ChipsToIpFaktor": 12,
38     "RoundLimit": 15,
39     "TurnPhaseLimit": 6,
40     "CatIp": 8,
41     "StrikeMaximum": 4
42 }
```

Das JSON-Schema ist als [🔗 matchconfig.schema](#) ebenfalls angefügt und aufgrund seiner Länge hier nicht weiter angegeben.

## 2.3 Szenariobeschreibung

### 📄 Datei Definition 2: Szenariobeschreibung

Autoren: Benjamin Moosherr, Florian Sihler.

verpflichtend  
(OK)

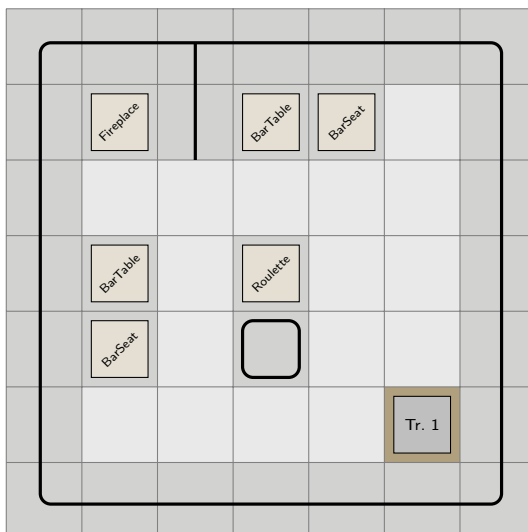
• v. 1

Es ist eine Konfigurationsdatei `<Szenarioname>.scenario` zu halten, deren Struktur ein Objekt der `Scenario`-Klasse ist. Eine Beispieldatei ist mit `example.scenario` gegeben.

```

1 {
2   "scenario": [
3     ["WALL", "WALL", "WALL", "WALL", "WALL", "WALL", "WALL"],
4     ["WALL", "FIREPLACE", "WALL", "BAR_TABLE", "BAR_SEAT", "FREE", "WALL"],
5     ["WALL", "FREE", "FREE", "FREE", "FREE", "FREE", "WALL"],
6     ["WALL", "BAR_TABLE", "FREE", "ROULETTE_TABLE", "FREE", "FREE", "WALL"],
7     ["WALL", "BAR_SEAT", "FREE", "WALL", "FREE", "FREE", "WALL"],
8     ["WALL", "FREE", "FREE", "FREE", "FREE", "SAFE", "WALL"],
9     ["WALL", "WALL", "WALL", "WALL", "WALL", "WALL", "WALL"]
10  ]
11 }
```

Wir erhalten das Feld:



Größe:  $7 \times 7$ , Valid: Ja.

Die vorläufige Standardisierung hat ergeben, dass die Nummerierung der Tresore vom Server zum Start zufällig (von 1 an) durchgeführt wird und sich so von Runde zu Runde unterscheiden kann (bei mehreren Tresoren versteht sich).

Weiter hat der Beschluss in Issue-Thread 72 ergeben, dass sich das `GadgetEnum: DIAMOND_COLLAR` im Tresor mit der höchsten Nummer befindet.

Ein Szenario muss übrigens *nicht* rechteckig sein, es kann durchaus andere Gestalten annehmen.

Weiter ist das JSON-Schema `scenario.schema` angefügt.

```

1 {
2   "$schema": "http://json-schema.org/draft-07/schema",
3   "type": "object",
4   "title": "Die_Szenariobeschreibung",
5   "description": "JSON-Schema für die Szenariobeschreibung. Basiert auf der von Benjamin",
6   "required": [
7     "scenario"
8   ],
9   "properties": {
10    "scenario": {
11      "$id": "#/properties/scenario",
12      "type": "array",
13      "title": "Das_Szenario-Schema",
14      "description": "Zweidimensionales Array aus Feldern.",
15      "items": {
16        "$id": "#/properties/scenario/items",
17        "type": "array",
18        "title": "Die_Zeilen_der_Karte",
19        "description": "Enthält für jede Zeile ein eindimensionales Array.",
```

```

20     "default": [],
21     "items": {
22       "$id": "#/properties/scenario/items/items",
23       "type": "string",
24       "title": "Die_jeweiligen_Spalteneinträge_der_Zeile",
25       "description": "Jeder_Spalteneintrag_in_dieser_Zeile_drückt_ein_Feld_aus.",
26       "enum": [
27         "BAR_TABLE",
28         "ROULETTE_TABLE",
29         "WALL",
30         "FREE",
31         "BAR_SEAT",
32         "SAFE",
33         "FIREPLACE"
34       ]
35     }
36   }
37 }
38 }
39 }

```



## 2.4 Charakterdefinition

### Datei Definition 3: Charakterdefinition

Autor: Team 14.

verpflichtend  
(OK)

• V. 1

Es ist eine Konfigurationsdatei `characters.json` zu halten, deren Struktur ein schlichtes `Array` aus `CharacterDescription`-Objekten ist. Eine Beispieldatei ist als  `characters.json`, ebenso wie das JSON-Schema  `characters.schema` angefügt.

```

1  {
2    "$schema": "http://json-schema.org/draft-07/schema",
3    "type": "array",
4    "uniqueItems": true,
5    "title": "CharacterDescription[]",
6    "description": "Das_Array_an_Character-Descriptions",
7    "items": {
8      "$id": "#/items",
9      "type": "object",
10     "title": "Eine_CharacterDescription",
11     "description": "Beschreibt_einen_Charakter",
12     "default": {},
13     "required": [
14       "name",
15       "features"
16     ],
17     "properties": {
18       "name": {
19         "$id": "#/items/properties/name",
20         "type": "string",
21         "title": "CharacterName",
22         "description": "Name_des_Charakters",
23         "minLength": 3
24       },
25       "description": {
26         "$id": "#/items/properties/description",
27         "type": "string",
28         "title": "CharacterDescription",

```

```
29     "description": "Die_Beschreibung_des_Charakters"
30   },
31   "gender": {
32     "$id": "#/items/properties/gender",
33     "type": "string",
34     "title": "CharacterGender",
35     "description": "Das_Geschlecht_eines_Charakters",
36     "enum": [
37       "MALE",
38       "FEMALE",
39       "DIVERSE"
40     ]
41   },
42   "features": {
43     "$id": "#/items/properties/features",
44     "type": "array",
45     "title": "CharacterFeatures",
46     "description": "Fähigkeiten/Eigenschaften_des_Charakters",
47     "items": {
48       "$id": "#/items/properties/features/items",
49       "type": "string",
50       "title": "PropertyEnum",
51       "description": "Die_verschiedenen_Eigenschaften",
52       "enum": [
53         "NIMBLENESS",
54         "SLUGGISHNESS",
55         "SPRYNESS",
56         "AGILITY",
57         "LUCKY_DEVIL",
58         "JINX",
59         "CLAMMY_CLOTHES",
60         "CONSTANT_CLAMMY_CLOTHES",
61         "ROBUST_STOMACH",
62         "TOUGHNESS",
63         "BABYSITTER",
64         "HONEY_TRAP",
65         "BANG_AND_BURN",
66         "FLAPS_AND_SEALS",
67         "TRADECRAFT",
68         "OBSERVATION"
69       ]
70     }
71   }
72 }
73 }
74 }
```

# 3

## Datentypen

### 3.1 Eigenschaften und Charakteristika

#### **{}** Datentyp Definition 1: RoleEnum

Autor: Jonas Mayer.

verpflichtend  
(OK)

• V. 1

Definiert die Rolle eines Spielers im Spiel.

```
1 enum RoleEnum {  
2     SPECTATOR,  
3     PLAYER,  
4     AI  
5 }
```

Hierbei bezeichnet SPECTATOR die Rolle des Zuschauers (Spectator), PLAYER die Rolle des Spielers (Player) und AI die Rolle als KI-Spieler. Diese wird beim Verbindungsaufbau bereits mit der `HelloMessage` übermittelt.

#### **{}** Datentyp Definition 2: PropertyEnum

Autoren: Team 14, Florian Sihler.

verpflichtend  
(OK)

• V. 1

In der Konfigurationsdatei `characters.json` werden Charakteren ihre Eigenschaften zugewiesen, welche folgende Werte annehmen können:

```
1 enum PropertyEnum {  
2     NIMBLENESS,  
3     SLUGGISHNESS,  
4     SPRYNESS,  
5     AGILITY,  
6     LUCKY_DEVIL,  
7     JINX,  
8     CLAMMY_CLOTHES,  
9     CONSTANT_CLAMMY_CLOTHES,  
10    ROBUST_STOMACH,  
11    TOUGHNESS,  
12    BABYSITTER,  
13    HONEY_TRAP,  
14    BANG_AND_BURN,  
15    FLAPS_AND_SEALS,  
16    TRADECRAFT,
```



```

17  OBSERVATION
18  }

```

Diese Eigenschaften können durch Operationen ge-/verändert werden.

**Datentyp Definition 3: GadgetEnum**  
Autoren: Jonas Mayer, Florian Sihler.

verpflichtend  
(OK)

• V. 1

Wir kennen folgende **Gadgets**:

```

1  enum GadgetEnum {
2      HAIRDRYER,
3      MOLEDIE,
4      TECHNICOLOUR_PRISM,
5      BOWLER_BLADE,
6      MAGNETIC_WATCH,
7      POISON_PILLS,
8      LASER_COMPACT,
9      ROCKET_PEN,
10     GAS_GLOSS,
11     MOTHBALL_POUCH,
12     FOG_TIN,
13     GRAPPLE,
14     WIRETAP_WITH_EARPLUGS,
15     DIAMOND_COLLAR,
16     JETPACK,
17     CHICKEN_FEED,
18     NUGGET,
19     MIRROR_OF_WILDERNESS,
20     POCKET_LITTER,
21     COCKTAIL
22 }

```

## 3.2 Die Gadgets

**Datentyp Definition 4: Gadget**  
Autoren: Jonas Mayer, Florian Sihler.

verpflichtend  
(OK)

• V. 1

Die grundlegende Ausprägung eines **Gadgets** über die **GadgetEnum**. Abseits des **Cocktail**-Gadget und des **WiretapWithEarplugs**-Gadget wurde kein Gadget explizit mit gesonderten Ausprägungen versehen und es obliegt daher den implementierenden Teams, ob es hierfür extra Klassen anlegt oder nicht. Die zu den Gadgets gebundenen Eigenschaften lassen sich über die **Matchconfig** beziehen.

```

1  class Gadget {
2      GadgetEnum gadget;
3      Integer usages;
4  }

```

**gadget:** Bezeichnet den Typ des Gadgets und damit auch die möglichen Aktionen.

**usages:** Gibt an, wie oft das Gadget noch benutzt werden kann. Gadgets die unendlich oft benutzt werden können oder bei denen ein solcher Wert keine Anwendung findet, ignorieren den Eintrag.

#### **{}** Datentyp Definition 5: Gadget - WiretapWithEarplugs

Autoren: Hannes Steck, Jonas Mayer, Florian Sihler.

verpflichtend  
(OK)

• v. 1

Explizite Ausprägung eines **Gadgets**, mit **GadgetEnum**: :WIRETAP\_WITH\_EARPLUGS fügt es die folgenden Attribute hinzu:

```
1 class WiretapWithEarplugs extends Gadget {
2     Boolean working;
3     UUID activeOn;
4 }
```

**working:** Gibt an, ob das Gadget in dieser Runde noch funktioniert oder ob es aufgrund der Ausfallwahrscheinlichkeit ausgefallen ist.

**activeOn:** Gibt an auf welchem Ziel das Gadget aktiv ist. Sofern es noch nicht aktiviert worden ist, setzt man dieses Feld explizit auf **null**.

*Hinweis:* nachdem eine Nachricht mit dem nun kaputten Gadget eingegangen ist (**working** = **false**), steht es dem Server frei dieses ab sofort zu ignorieren, also gänzlich verschwinden zu lassen und in keiner Nachricht mehr zu erwähnen.

#### **{}** Datentyp Definition 6: Gadget - Cocktail

Autor: Johannes Rauch.

verpflichtend  
(OK)

• v. 1

Explizite Ausprägung eines **Gadgets**s. Mit **GadgetEnum**: :COCKTAIL fügt es die folgenden Attribute hinzu:

```
1 class Cocktail extends Gadget {
2     Boolean isPoisoned;
3 }
```

**isPoisoned:** Dieser Wert gibt an, ob der Cocktail vergiftet ist (**true**) oder nicht (**false**).

## 3.3 Das Szenario

#### **{}** Datentyp Definition 7: FieldStateEnum

Autor: Unbekannt?.

verpflichtend  
(OK)

• v. 1

Ein Feld des Szenarios wird über das folgende Enum charakterisiert:

```
1 enum FieldStateEnum {
2     BAR_TABLE,
3     ROULETTE_TABLE,
4     WALL,
5     FREE,
6     BAR_SEAT,
7     SAFE,
8     FIREPLACE
9 }
```

**{}** Datentyp Definition 8: Field

Autor: Jonas Mayer.

verpflichtend  
(OK)

V. 1

Definiert ein Feld im Szenarioname.scenario vom Typ `FieldStateEnum`, wobei die Karte des Szenarios durch `FieldMap` gebildet wird!

```

1 class Field {
2     FieldStateEnum state;
3     Gadget gadget;
4
5     Boolean isDestroyed;
6     Boolean isInverted;
7     Integer chipAmount;
8
9     Integer safeIndex;
10
11    Boolean isFoggy;
12    Boolean isUpdated;
13 }
```

Die Attribute beschreiben sich wie folgt. Abseits des letzten sind alle verpflichtend (je Angabe auch explizit *nullable*):

**state:** Der Zustand/die Art des Feldes.

**gadget:** Dieses Attribut gibt an, um welches Gadget es sich handelt (sofern eines auf dem Feld platziert ist), ansonsten ist es *explizit null*!

**isDestroyed:** Nur relevant, wenn `state = FieldStateEnum::ROULETTE_TABLE`. Gibt dann an, ob der Tisch mit `PropertyEnum::PropertyEnum` zerstört (`true`) wurde oder nicht (`false`). Der Zustand für ein „Nicht-Roulette-Tisch-Feld“ ist nicht definiert und nicht von Relevanz.

**isInverted:** Gibt an, ob ein auf dem Feld platzierter Roulette-Tisch (hinsichtlich seiner Wahrscheinlichkeiten) invertiert wurde.

**chipAmount:** Nur relevant, wenn `state = FieldStateEnum::ROULETTE_TABLE`. Gibt dann an, wie viele Chips noch zum Spielen zur Verfügung stehen.

**safeIndex:** Nur relevant, wenn `state = FieldStateEnum::SAFE`. Gibt in diesem Fall den Index des Tresors an, wobei von 1 an nummeriert wird.

**isFoggy:** Gibt an, ob das Spielfeld vom Nebel einer `GadgetEnum::FOG_TIN` beeinflusst wird.

**isUpdated:** Optionale Hilfestellung des Servers, die nicht umgesetzt werden muss. Wenn sie vom Server geliefert wird, kann sich der Client dazu entscheiden die Information für eine effizientere Aktualisierung zu verwenden.

**{}** Datentyp Definition 9: FieldMap

Autor: Unbekannt?.

verpflichtend  
(OK)

V. 1

Definiert die Karte eines Szenarios:

```

1 class FieldMap {
2     Field[][] map;
3 }
```

**{}** Datentyp Definition 10: State

Autoren: Jonas Mayer, Florian Sihler.

verpflichtend  
(OK)

V. 1

Dieses Objekt vereinigt den *gesamten* aktuellen Spielzustand aus Sicht der Spielmechanik:

```
1 class State {
2     Integer currentRound;
3     FieldMap map;
4     Set<Integer> mySafeCombinations;
5     Set<Character> characters;
6     Point catCoordinates;
7     Point janitorCoordinates;
8 }
```

**currentRound:** Enthält die aktuelle Rundenzahl, die von 1 an gezählt wird. Das Spiel startet also mit der ersten Runde.

**map:** Aktueller Zustand der Karte (inklusive der platzierten **Characters**, samt der **Gadgets**)

**mySafeCombinations:** Enthält die dem Client bekannten Safe-Kombinationen, wobei der Zuschauer je nach Serverimplementation keine oder die beider Spieler erhalten kann – weitere Trennungen sind in dieser Iteration des Dokuments nicht vorgesehen, es wird der erstere Weg präferiert.

**characters:** Alle auf dem Feld aktiven Charaktere, samt ihres aktuellen Zustands.

**catCoordinates:** Die Koordinaten der Katze (**NPCEnum**: :CAT). Diese sind explizit *null* oder außerhalb der Spielfelddimensionen wenn sich die Katze nicht auf dem Spielfeld befindet, also nicht aktiv ist.

**janitorCoordinates:** Die Koordinaten des Hausmeisters (**NPCEnum**: :JANITOR). Diese sind explizit *null* oder außerhalb der Spielfelddimensionen wenn sich der Hausmeister nicht auf dem Spielfeld aufhält oder (noch) nicht aktiv ist.

**Datentyp Definition 11:** Szenario  
Autoren: Benjamin Moosherr, Simon Matt.

verpflichtend  
(OK)

v. 1

Repräsentiert das Objekt des in der Szenariobeschreibung definierten Szenarios:

```
1 class Scenario {
2     FieldStateEnum[][] scenario;
3 }
```

*Hinweis:* Ein Spielfeld muss hierbei nicht zwangsläufig rechteckig sein, so ist es durchaus gestattet die Zeilen *unterschiedlich Lang* zu gestalten und so Felder außerhalb von Mauern auch als frei zu markieren – die Darstellung „unerreichbarer Felder“ ist hierbei wieder dem jeweils implementierenden Team selbst überlassen und rein kosmetisch.

**scenario:** aktueller Zustand der Karte (inklusive der platzierten **Characters** samt der **Gadgets**)

## 3.4 Die Partiekonfiguration

### { Datentyp Definition 12: Matchconfig

Autor: Florian Sihler.

verpflichtend  
(OK)

v. 1

Repräsentiert das Objekt der in der Partiekonfiguration vermerkten Struktur:

```
1 class Matchconfig {
2     /* Gadget: Moledie */
3     Integer Moledie_Range;
4     /* Gadget: BowlerBlade */
5     Integer BowlerBlade_Range;
6     Double BowlerBlade_HitChance;
7     Integer BowlerBlade_Damage;
8     /* Gadget: LaserCompact */
9     Double LaserCompact_HitChance;
10    /* Gadget: RocketPen */
11    Integer RocketPen_Damage;
12    /* Gadget: GasGloss */
13    Integer GasGloss_Damage;
14    /* Gadget: MothballPouch */
15    Integer MothballPouch_Range;
16    Integer MothballPouch_Damage;
17    /* Gadget: FogTin */
18    Integer FogTin_Range;
19    /* Gadget: Grapple */
20    Integer Grapple_Range;
21    Double Grapple_HitChance;
22    /* Gadget: WiretapWithEarplugs */
23    Double WiretapWithEarplugs_FailChance;
24    /* Gadget: Mirror */
25    Double Mirror_SwapChance;
26    /* Gadget: Cocktail */
27    Double Cocktail_DodgeChance;
28    Integer Cocktail_Hp;
29    /* Aktionen */
30    Double Spy_SuccessChance;
31    Double Babysitter_SuccessChance;
32    Double HoneyTrap_SuccessChance;
33    Double Observation_SuccessChance;
34    /* Spielfaktoren */
35    Integer ChipsToIpFaktor;
36    Integer RoundLimit;
37    Integer TurnPhaseLimit;
38    Integer CatIp;
39    Integer StrikeMaximum;
40 }
```

Die Beschreibungen gehen aus den Anforderungen im Lastenheft (jeweils deutlich) hervor.

## 3.5 Der Spielablauf

### 3.5.1 Ausführbare Operationen

#### **Datentyp Definition 13:** OperationenEnum

Autor: Jonas Mayer.

verpflichtend  
(OK)

• v. 1

Stellt alle möglichen „Operationsarten“, also Aktionen die Charaktere ausführen können dar. Die Exfiltration (*Exfiltration*) ist hierbei eine besondere Aktion, da sie vom Server erzwungen und nicht vom Spieler ausgeführt wird.

```

1  enum OperationEnum {
2      GADGET_ACTION,
3      SPY_ACTION,
4      GAMBLE_ACTION,
5      PROPERTY_ACTION,
6      MOVEMENT,
7      CAT_ACTION,
8      JANITOR_ACTION,
9      EXFILTRATION,
10     RETIRE
11 }
```

#### **Kommentar 3:** Operation

Autoren: Jonas Mayer, Florian Sihler.

information  
(OK)

• v. 1

Eine *Operation* ist eine Handlung, die der Spieler dem Server schickt. Der Server fragt eine solche Operation für einen *Character* explizit mit einer *RequestGameOperationMessage* an, der Spieler-Client antwortet mit einer *GameOperationMessage*. Der Server spiegelt dann die Aktion und alle daraus resultierenden Aktionen im Statusupdate mittels einer *GameStatusMessage* auf beide Spieler (logischerweise auch die KIs) sowie alle potentiellen Zuschauer. Sollte sich durch die Aktionen eine Änderungen in den Fraktions-Konstellationen ergeben haben, so wird *in jedem Fall* (also auch wenn ein Spieler ausscheidet) eine *SpectatorRevealMessage* an die Zuschauer versendet (hihi, versendet natürlich, aber ich mag den Typo).

Aktionen die von der Katze (*NPCEnum::CAT*) oder vom Hausmeister (*NPCEnum::JANITOR*) ausgeführt werden, arbeiten ohne die *characterId* und greifen ihre Daten damit direkt von *BaseOperation* ab. Sie sind mit der *Exfiltration*, die nicht vom Spieler sondern nur vom Server angestoßen werden kann, die einzigen besonderen Operationen.

#### **Datentyp Definition 14:** BaseOperation

Autor: Florian Sihler.

verpflichtend  
(OK)

• v. 1

Definiert die Grundlage einer *Operation* und wird nicht direkt verwendet.

```

1  class BaseOperation {
2      OperationEnum type;
3      Boolean successful;
4      Point target;
5  }
```

Die Attribute haben jeweils die folgende Bedeutung:

**type:** Der Typ der Operation, ausgedrückt durch die Werte in der `OperationEnum`.

**successful:** Gibt an, ob die Operation gelingt.

**target:** Gibt das Zielfeld der Operation an.

Die `CatAction`, sowie `JanitorAction`-Operation werden beide lediglich über ihren Zustand im `OperationEnum` identifiziert, da sie keine weiteren Eigenschaften benötigen. Es obliegt den implementierenden Teams hierfür gesonderte Datenstrukturen anzuführen oder es dabei zu belassen.

### **{}** Datentyp Definition 15: Operation

Autor: Jonas Mayer.

verpflichtend  
(OK)

• V. 1

Stellt die Basisinformationen einer Operation und bildet damit die Grundlage für spezialisierte Klassen, die eine Operation ergeben, welche einen Spieler betrifft (`Character`).

```
1 class Operation extends BaseOperation {
2     UUID characterId;
3 }
```

Die Attribute haben (oder vielmehr das Attribut hat) jeweils die folgende Bedeutung:

**characterId:** Spezifiziert den `Character`, von dem die Aktion begangen wurde.

Die `SpyAction`, sowie `Retiring`-Operation werden beide lediglich über ihren Zustand im `OperationEnum` identifiziert, da sie keine weiteren Eigenschaften benötigen, es obliegt den implementierenden Teams hierfür gesonderte Datenstrukturen anzuführen oder es dabei zu belassen.

### **{}** Datentyp Definition 16: GadgetAction

Autor: Jonas Mayer.

verpflichtend  
(OK)

• V. 1

Eine Aktion bei der `Gadget` verwendet wird. Die Operation wird selbst über `OperationEnum::GADGET_ACTION` identifiziert:

```
1 class GadgetAction extends Operation {
2     GadgetEnum gadget;
3 }
```

**gadget:** Das in der Operation verwendete Gadget. Im Falle von `GadgetEnum::COCKTAIL` ergibt sich der „Vergiftet-Zustand“ über die `GameStatusMessage` und damit über den aktuellen Spielzustand.

### **{}** Datentyp Definition 17: GambleAction

Autor: Jonas Mayer.

verpflichtend  
(OK)

• V. 1

Diese Operation kann in Reichweite eines `FieldStateEnum::ROULETTE_TABLE` verwendet werden und wird über `OperationEnum::GAMBLE_ACTION` identifiziert.

```
1 class GambleAction extends Operation {
2     Integer stake;
3 }
```

**stake:** Die gesetzte Anzahl an Chips.

### **{}** Datentyp Definition 18: PropertyAction

Autor: Jonas Mayer.

verpflichtend  
(OK)

Diese Operation wird über `OperationEnum::PROPERTY_ACTION` identifiziert.

```
1 class PropertyAction extends Operation {
2     PropertyEnum usedProperty;
3 }
```

**usedProperty:** Identifiziert den Einsatz einer Eigenschaft. Nach bisherigem Stand ist dies ausschließlich für `PropertyEnum::BANG_AND_BURN` und `PropertyEnum::OBSERVATION` möglich/vonnöten.

### { Datentyp Definition 19: Movement

Autor: Jonas Mayer.

verpflichtend  
(OK)

• V. 1

Diese Operation wird über `OperationEnum::MOVEMENT` identifiziert:

```
1 class Movement extends Operation {
2     Point from;
3 }
```

Das Zielfeld ergibt sich über `Operation::target`.

**from:** Gibt das Startfeld aus Redundanzgründen an.

### { Datentyp Definition 20: Exfiltration

Autor: Johannes Rauch.

verpflichtend  
(OK)

• V. 1

Die Klasse stellt eine spezielle `Operation` dar, welche über `OperationEnum::EXFILTRATION` identifiziert wird. Wenn ein Charakter durch eine Aktion stirbt, fügt der Server eine Exfiltrationsoperation zu den ausgeführten Operationen hinzu. Die ermöglicht die Animation der Exfiltration. Das Zielfeld ergibt sich über `Operation::target`.

```
1 class Exfiltration extends Operation {
2     Point from;
3 }
```

**from:** Gibt das Startfeld (aus Redundanzgründen) an.

## 3.5.2 Fehlerbehandlung

### { Datentyp Definition 21: ErrorTypeEnum

Autor: Jonas Mayer.

verpflichtend  
(OK)

• V. 1

Alle möglichen Fehlerursachen für eine `ConnectionErrorMessage`, welche im Kontext eines Verbindungsaufbaus benötigt wird.

```
1 enum ErrorTypeEnum {
2     NAME_NOT_AVAILABLE,
3     ALREADY_SERVING,
4     GENERAL
5 }
```

Die näheren Bedeutungen sind wie folgt:

- **NAME\_NOT\_AVAILABLE:** Wird dann ausgelöst, wenn der gewünschte Name des Spielers bereits vergeben ist.



- *ALREADY\_SERVING*: Wird dann ausgelöst, wenn der Server bereits belegt ist.
- *GENERAL*: Diese Art von Fehler ist nicht näher bestimmt und dient damit als Platzhalter.

### 3.5.3 Spielende

#### **{}** Datentyp Definition 22: StatisticsEntry

Autor: Simon Matt.

verpflichtend  
(OK)

• V. 1

Eintrag einer einzelnen Statistik in *Statistics*:

```
1 enum StatisticsEntry {  
2     String title;  
3     String description;  
4     String valuePlayer1;  
5     String valuePlayer2;  
6 }
```

**title**: Titel der Statistik (wie „Roulette Siege“).

**description**: Beschreibung zur jeweiligen Statistik.

**valuePlayer1**: Wert des ersten Spielers.

**valuePlayer2**: Wert des zweiten Spielers.

#### **{}** Datentyp Definition 23: Statistics

Autor: Simon Matt.

verpflichtend  
(OK)

• V. 1

Die Sammlung der Statistiken, die in einer *StatisticsMessage* zum Spielende versendet wird.

```
1 enum Statistics {  
2     StatisticsEntry[] entries;  
3 }
```

**entries**: Alle Statistik-Einträge.

#### **{}** Datentyp Definition 24: VictoryEnum

Autor: Jonas Mayer.

verpflichtend  
(OK)

• V. 1

Der Siegesgrund, der in einer *StatisticsMessage* zum Spielende versendet wird.

```
1 enum VictoryEnum {  
2     VICTORY_BY_IP,  
3     VICTORY_BY_COLLAR,  
4     VICTORY_BY_DRINKING,  
5     VICTORY_BY_SPILLING,  
6     VICTORY_BY_HP,  
7     VICTORY_BY_RANDOMNESS  
8 }
```

Enthält alle im Lastenheft präsenten Sieggründe, die Erklärungen ergeben sich über die Definitionen im Lastenheft.

# 4

## Charaktere

### 4.1 Eigenschaften eines Charakters

#### 4.1.1 Die PCs

##### **{}** Datentyp Definition 25: CharacterDescription

Autoren: Team 14, Florian Sihler.

verpflichtend  
(OK)

● v. 1

In der Konfigurationsdatei `characters.json` werden die Charakterbeschreibungen festgelegt, die dann auch im Spiel verwendet werden. Dabei ist die Charakterbeschreibung ein Objekt folgender Klasse:

```
1 class CharacterDescription {
2     String name;
3     String description;
4     GenderEnum gender;
5     List<PropertyEnum> features;
6 }
```

Die ersten beiden Felder dürfen *nicht null* sein und sind somit verpflichtend:

**name:** Der Name des Charakters.

**features:** Liste an Eigenschaften des `PropertyEnums`, die den Charakter als solchen auszeichnen.

**description:** Eine (rein textuelle) Beschreibung des Charakters.

**gender:** Das Geschlecht des Charakters (kann für ein Default-Model verwendet werden, wenn für den Namen kein Sprite oder ähnliches vorliegt).

##### **{}** Datentyp Definition 26: CharacterInformation

Autoren: Florian Sihler, Jonas Mayer, Hannes Steck.

verpflichtend  
(OK)

● v. 2

Legt die Charakter-Informationen fest die der Server den Clients in der `HelloReplyMessage` beziehungsweise der `ConfigDeliveryMessage` übermittelt.

```
1 class CharacterInformation {
2     CharacterDescription character;
3     :UUID characterId;
4 }
```

**character:** Alle notwendigen Metadaten des Charakters.

**characterId:** Die vom Server zugewiesene `UUID` für den Charakter.

**{}** Datentyp Definition 27: GenderEnum

Autor: Team 14.

verpflichtend  
(OK)

V. 1

Das GenderEnum repräsentiert eine Konstante, die folgende Werte annehmen kann:

```
1 enum GenderEnum {  
2     MALE,  
3     FEMALE,  
4     DIVERSE  
5 }
```

**{}** Datentyp Definition 28: Character

Autor: Unbekannt?.

verpflichtend  
(OK)

V. 1

Die Klasse Character implementiert die Eigenschaften eines spezifischen Charakters. Diese Klasse modelliert den Charakter-*Zustand* während des Spiels und darf nicht mit der **CharacterDescription** verwechselt werden, die zu Beginn des Spiels verteilt wird.

```
1 class Character {  
2     UUID characterId;  
3     String name;  
4     Point coordinates;  
5     Integer mp, ap, hp, ip, chips;  
6     List<PropertyEnum> properties;  
7     List<Gadget> gadgets;  
8 }
```

Die Attribute haben jeweils die folgende Bedeutung:

**characterId:** Ist als **UUID** die eindeutige Identifikationsnummer für den Charakter.

**name:** Spiegel des Namens aus **CharacterDescription::name**.

**mp:** Bewegungspunkte des Charakters.

**ap:** Aktionspunkte des Charakters.

**hp:** Lebenspunkte des Charakters.

**ip:** Intelligencepunkte des Charakters.

**chips:** Anzahl der Chips des Charakters.

**properties:** Alle Eigenschaften aus **PropertyEnum** des Charakters, wird mit **CharacterDescription::features** initialisiert.

**gadgets:** Enthält alle **Gadgets**, die der Charakter aktuell besitzt.

### 4.1.2 Die NPCs

**{}** Datentyp Definition 29: NPCEnum

Autoren: Jonas Mayer, Florian Sihler.

verpflichtend  
(OK)

V. 1

```
1 enum NPCEnum {  
2     CAT,  
3     JANITOR  
4 }
```

Agenten werden hierbei als normaler **Character** repräsentiert, für den der Server aber keine Zuganforderung sendet, sondern die Züge direkt selbst übernimmt, ausführt und abhandelt.

## Netzwerkstandard

### 5.1 Grundlegendes

#### 5.1.1 Die Abläufe

Die Anforderung einer Spielpause läuft wie in Modell 5.1.1 (Kontrollnachrichten, Spielpause) vermerkt, analog findet sich der Vorgang beim Verlassen eines Spiels in Modell 5.1.2 (Verlassen des Spiels) und der Während der Ausrüstungsphase in Modell 5.1.3 (Die Ausrüstungsphase).

#### 5.1.2 Blaupause

**Datentyp Definition 30:** MessageContainer

Autor: Florian Sihler.

verpflichtend  
(OK)

● V. 1

Definiert das Containerformat für eine Nachricht, alle die folgenden Attribute und Felder lassen sich in jeder Netzwerknachricht wiederfinden, wobei eine *debugMessage* selbstredend optional ist.

```
1 class MessageContainer {  
2     UUID playerId;  
3     MessageTypeEnum type;  
4     Date creationDate;  
5     String debugMessage;  
6 }
```

**playerId:** Der von der Nachricht betroffene Spieler (wenn der (Client des) Spieler diese Nachricht sendet, trägt er sich selbst in das Feld ein).

**type:** Gibt den Typ der Nachricht an um sie entsprechend parsen zu können.

**creationDate:** Gibt den Zeitstempel der Nachricht an.

**debugMessage:** Kann während der Entwicklung dazu verwendet werden um zusätzliche Daten zwischen Server und Client zu transportieren. So kann im Falle einer nicht parse-baren Nachricht so eine Information an den Client getragen werden ohne direkt alle Logs durchsuchen zu müssen.

## 5.2 Die Nachrichten

### 5.2.1 Kategorisierung

#### { Datentyp Definition 31: MessageTypeEnum

Autor: Florian Sihler.

verpflichtend  
(OK)

v. 1

Listet alle Nachrichtentypen auf:

```
1 enum MessageTypeEnum {
2     // Spielinitialisierung
3     HELLO,
4     HELLO_REPLY,
5     RECONNECT,
6     CONNECTION_ERROR,
7     GAME_STARTED,
8     // Wahlphase
9     REQUEST_ITEM_CHOICE,
10    ITEM_CHOICE,
11    REQUEST_EQUIPMENT_CHOICE,
12    EQUIPMENT_CHOICE,
13    // Spielphase
14    GAME_STATUS,
15    REQUEST_GAME_OPERATION,
16    GAME_OPERATION,
17    SPECTATOR_REVEAL,
18    // Spielende
19    STATISTICS,
20    // Kontrollnachrichten
21    GAME_LEAVE,
22    GAME_LEFT,
23    REQUEST_GAME_PAUSE,
24    GAME_PAUSE,
25    REQUEST_CONFIG_DELIVERY,
26    CONFIG_DELIVERY,
27    STRIKE,
28    // Optionale Komponenten
29    REQUEST_REPLAY,
30    REPLAY
31 }
```

### 5.2.2 Spielinitialisierung

#### </> Nachricht Definition 1: HelloMessage

Autor: Jonas Mayer.

verpflichtend  
(OK)

v. 1

Spezifiziert den Body eines `MessageContainer` des `MessageTypeEnum::HELLO`-Typs. Diese Nachricht wird dann gesendet, wenn der Spieler die Verbindung zu einem Server aufbauen will. Ein Benutzer wählt dann eine der Rollen `RoleEnum::PLAYER` (Spieler) oder `RoleEnum::SPECTATOR` (Zuschauer). Die Rolle `RoleEnum::AI` ist der KI-Komponente vorbehalten, um sich beim Server als solche zu authentifizieren.

```
1 class HelloMessage extends MessageContainer {
2     String name;
3     RoleEnum role;
4 }
```

Der Server antwortet mit einer `HelloReplyMessage`.

#### </> Nachricht Definition 2: HelloReplyMessage

Autor: Jonas Mayer.

verpflichtend  
(OK)

● V. 1

Spezifiziert den Body eines `MessageContainer` des `MessageTypeEnum::HELLO_REPLY`-Typs. Diese Nachricht wird nach einem *hello* also einer `HelloMessage` vom Client mit Rolle `RoleEnum::PLAYER` geschickt, falls der Server verfügbar ist. Dies bedeutet, der Server ist „leer“ oder es kann unmittelbar ein Spiel gestartet werden.

```
1 class HelloReplyMessage extends MessageContainer {
2     UUID sessionId;
3     UUID playerId;
4     Scenario level;
5     Matchconfig settings;
6     CharacterInformation[] character_settings;
7 }
```

#### </> Nachricht Definition 3: ReconnectMessage

Autor: Jonas Mayer.

verpflichtend  
(OK)

● V. 1

Spezifiziert den Body eines `MessageContainer` des `MessageTypeEnum::RECONNECT`-Typs. Falls ein Spieler die Verbindung zum Server verloren hat, kann dieser das Spiel wieder aufnehmen. Dazu benötigt der Client die Session-ID und seine User-ID, um sich auch als ursprünglicher Spieler zu authentifizieren.

```
1 class ReconnectMessage extends MessageContainer {
2     UUID sessionId;
3     UUID playerId;
4 }
```

#### </> Nachricht Definition 4: ConnectionErrorMessage

Autor: Jonas Mayer.

verpflichtend  
(OK)

● V. 1

Spezifiziert den Body eines `MessageContainer` des `MessageTypeEnum::CONNECTION_ERROR`-Typs. Falls der Server nicht verfügbar sein sollte, wird stattdessen folgende Nachricht geschickt:

```
1 class ConnectionErrorMessage extends MessageContainer {
2     ErrorTypeEnum reason;
3 }
```

**reason:** Gibt den Grund für den gescheiterten Serverbeitritt an.

### 5.2.3 Spielstart

#### </> Nachricht Definition 5: GameStartedMessage

Autor: Jonas Mayer.

verpflichtend  
(OK)

• v. 1

Spezifiziert den Body eines `MessageContainer` des `MessageTypeEnum::GAME_STARTED`-Typs. Signaliert den Start des Spiels und überträgt alle dafür notwendigen Daten.

```
1 class GameStartedMessage extends MessageContainer {
2     UUID playerOneId;
3     UUID playerTwoId;
4     String playerOneName;
5     String playerTwoName;
6     UUID sessionId;
7 }
```

*Hinweis:* Diese Nachricht wird auch dann an einen Spectator gesendet, wenn dieser einem Spiel später beitrifft, sie ist also Teil seiner „Willkommensroutine“ (oder anders formuliert der „Zuschauer-Willkommensroutine“).

**playerOneId:** Die `UUID` des ersten Spielers.

**playerTwoId:** Die `UUID` des zweiten Spielers.

**playerOneName:** Der Name des ersten Spielers.

**playerTwoName:** Der Name des zweiten Spielers.

**sessionId:** Die Sitzungs-ID.

Die IDs werden auch an die KIs verliehen.

### 5.2.4 Wahlphase

*Die Spieler wählen ihre Charaktere und Gadgets in der Wahlphase. Insgesamt wählen sie acht Items, davon zwei bis vier Charaktere und vier bis sechs Gadgets. Wir bezeichnen Charaktere und Gadgets als Items, wie in den Begriffsdefinitionen vermerkt.*

#### </> Nachricht Definition 6: RequestItemChoiceMessage

Autoren: Unbekannt?, Florian sihler.

verpflichtend  
(OK)

• v. 1

Spezifiziert den Body eines `MessageContainer` des `MessageTypeEnum::REQUEST_ITEM_CHOICE`-Typs.

```
1 class RequestItemChoiceMessage extends MessageContainer {
2     Integer offerNr;
3     List<UUID> offeredCharacterIds;
4     List<GadgetEnum> offeredGadgets;
5 }
```

**offerNr:** Dient der Konsistenzerhaltung. Wenn eine Wahl nicht erfolgreich gewesen sein sollte wird die Verbindung abgebrochen.

**offeredCharacterIds:** Enthält die UUIDs der angebotenen `Characters`.

**offeredGadgets:** Enthält die angebotenen `Gadgets`.

#### </> Nachricht Definition 7: ItemChoiceMessage

Autoren: Unbekannt?, Florian sihler.

verpflichtend  
(OK)

• v. 1

Spezifiziert den Body eines `MessageContainer` des `MessageTypeEnum::ITEM_CHOICE`-Typs.

```

1 class ItemChoiceMessage extends MessageContainer {
2     Integer offerNr;
3     UUID chosenCharacterId;
4     GadgetEnum chosenGadget;
5 }

```

*Hinweis:* eines der beiden Felder `chosenCharacter` beziehungsweise `chosenGadget` *muss explizit null* sein!

**offerNr:** Dient dazu, dass der Server weiß, auf welches Angebot sich der Client bezieht. Der Server kann so schnell kontrollieren, ob der Client sinnvolle/erlaubte Dinge tut.

**chosenCharacterId:** Wenn nicht `null` wurde ein Charakter gewählt, dessen `UUID` dann hier enthalten ist.

**chosenGadget:** Wenn nicht `null` enthält das Feld das gewählte `Gadget`, welches über die `GadgetEnum` eindeutig identifiziert wird.

### 5.2.5 Ausrüstungsphase

#### </> Nachricht Definition 8: RequestEquipmentChoiceMessage

Autoren: Unbekannt?, Florian sihler.

verpflichtend  
(OK)

• v. 1

Spezifiziert den Body eines `MessageContainer` des `MessageTypeEnum::REQUEST_EQUIPMENT_CHOICE`-Typs. Diese Nachricht wird gesendet, um den Clients den Start der Ausrüstungsphase mitzuteilen.

```

1 class RequestEquipmentChoiceMessage extends MessageContainer {
2     List<UUID> chosenCharacterIds;
3     List<GadgetEnum> chosenGadgets;
4 }

```

**chosenCharacterIds:** Enthält die `UUIDs` aller `Characters`, die der jeweilige Spieler gewählt hat.

**chosenGadgets:** Enthält die `Gadgets`, die der jeweilige Spieler gewählt hat.

Diese Nachricht enthält die Informationen aus reinen Redundanzzwecken, sollten sich Unstimmigkeiten ergeben gewinnt der Server, in wie weit der Client dann hinsichtlich einer Fehleranalyse operiert, ist dem jeweils implementierenden Team selbst überlassen.

#### </> Nachricht Definition 9: EquipmentChoiceMessage

Autoren: Unbekannt?, Florian sihler.

verpflichtend  
(OK)

• v. 1

Spezifiziert den Body eines `MessageContainer` des `MessageTypeEnum::EQUIPMENT_CHOICE`-Typs. Diese Nachricht wird gesendet um dem Server die Wahl aller Ausrüstungsgegenstände mitzuteilen. Die Nachricht wird also einmalig vom Client an den Server übermittelt.

```

1 class EquipmentChoiceMessage extends MessageContainer {
2     Map<UUID, List<GadgetEnum>> equipment;
3 }

```

**equipment:** eine finale Zuordnung von `Character-UUIDs` auf `Gadgets`. Die `UUIDs`, sowie die `Gadgets` müssen aus dem in `RequestEquipmentChoiceMessage` enthaltenen Pool stammen, sonst wird die Verbindung zum Server abgebrochen und der Client fliegt raus.



## 5.2.6 Spielphase

Ein Spiel läuft in *Runden* ab. Eine *Runde* besteht aus:

1. Vorbereitungsphase, diese wird direkt mit dem Rundenstart initiiert. Sie enthält:
  - Alle *Cocktails* werden aufgefüllt.
  - Die *Ausfallwahrscheinlichkeitsproben* für die Gadgets (spezifischer: das eine Gadget, das es betrifft: *WiretapWithEarplugs*) werden durchgeführt.
  - Die *Zugreihenfolge* der Charaktere wird (zufällig!) ausgewürfelt.
2. Zug-/Operationsphase, die Charaktere sind gemäß der ausgewürfelten Reihenfolge am Zug.

Jeder *Zug* besteht aus mehreren *Zugphasen*, wobei der Server nach jeder Zugphase die Siegbedingungen überprüft und jede Zugphase validiert:

1. Die *Bewegungs- und Aktionspunkte* des aktiven Charakters werden für die Runde bestimmt.
2. Durch *Operationen* kann der Spieler die Bewegungs- und Aktionspunkte in einer beliebigen Reihenfolge ausgeben, jede Ausgabe stellt einen einzelnen Abschnitt dar!

### </> Nachricht Definition 10: GameStateMessage

Autoren: Unbekannt?, Florian sihler.

verpflichtend  
(OK)

• v. 1

Spezifiziert den Body eines *MessageContainer* des *MessageTypeEnum::GAME\_STATUS*-Typs. Diese Nachricht wird nach jeder *Operation* gesendet (und zwar an alle Clients)!

```

1 class GameStateMessage extends MessageContainer {
2     UUID activeCharacterId;
3     List<BaseOperation> operations;
4     State state;
5     Boolean isGameOver;
6 }
```

Wenn die Zugzeit überschritten wird, sendet der Server eine solche *GameStateMessage* ohne eine *Operation* und dem nächsten aktiven Charakter, das bedeutet, dass der Server den bisherigen Zug beendet und den Client mit einem *Strike* bestraft.

**activeCharacter:** Spezifiziert den Charakter der am Zug war. Ist das Feld *null*, ist der Status nicht an einen Charakterzug gebunden. Das Feld kann auch für die Anzeige des aktiven Spielers verwendet werden.

**operations:** Enthält die *BaseOperations* seit der letzten *GameStateMessage*, damit die Clients diese animieren können. So können durch einen Angriff die HP des jeweiligen Charakters auf 0 fallen, womit zusätzlich die *Exfiltration* für diesen ausgelöst wird. Die Liste an Operationen muss *geordnet* entsprechend der chronologischen Reihenfolge der Aktionen sein.

**state:** Enthält den Zustand des Spiels als *State*. Dies beinhaltet die Aktions- und Bewegungspunkte des aktiven Charakters.

**isGameOver:** Wenn *true* ist das Spiel vorbei und die nächste Nachricht des Servers wird eine *StatisticsMessage* sein, die alle weiteren Informationen enthält.

### </> Nachricht Definition 11: RequestGameOperationMessage

Autor: Johannes Rauch.

verpflichtend  
(OK)

• v. 1

Spezifiziert den Body eines *MessageContainer* des *MessageTypeEnum::REQUEST\_GAME\_OPERATION*-Typs. Der Server fragt den Client des Spielers, dessen *Character* aktuell am Zug ist, explizit nach einer Operation.

```

1 class RequestGameOperationMessage extends MessageContainer {
2     UUID characterId;
3 }

```

**characterId:** Der `Character`, von dem die Operation erwartet wird.

#### </> Nachricht Definition 12: GameOperationMessage

Autor: Johannes Rauch.

verpflichtend  
(OK)

• v. 1

Spezifiziert den Body eines `MessageContainer` des `MessageTypeEnum::GAME_OPERATION`-Typs. Der Client antwortet damit auf eine `RequestGameOperationMessage`-Nachricht des Servers.

```

1 class GameOperationMessage extends MessageContainer {
2     Operation operation;
3 }

```

**operation:** Spezifiziert die vom Spieler ausgeführte `Operation`. Es muss eine Operation des aktiven Charakters sein. Ist das nicht der Fall oder ist die Operation invalide, wird die Verbindung vom Sever aus gekappt und der Client fliegt raus.

#### </> Nachricht Definition 13: SpectatorRevealMessage

Autoren: Jonas Mayer, Florian Sihler.

verpflichtend  
(OK)

• v. 1

Spezifiziert den Body eines `MessageContainer` des `MessageTypeEnum::SPECTATOR_REVEAL`-Typs. Diese wird vom Server zu allen Zuschauern geschickt, wenn sie dem Spiel beitreten oder sich über ein `GadgetEnum::NUGGET` eine Veränderung in der Fraktionen-Konstellation ergibt.

```

1 class SpectatorRevealMessage extends MessageContainer {
2     Set<UUID> factionPlayerOneIds;
3     Set<UUID> factionPlayerTwoIds;
4     Set<UUID> factionNeutralIds;
5 }

```

**factionPlayerOneIds:** `UUIDs` aller Charaktere, die unter der Kontrolle von Spieler 1 stehen.

**factionPlayerTwoIds:** `UUIDs` aller Charaktere, die unter der Kontrolle von Spieler 2 stehen.

**factionNeutralIds:** `UUIDs` aller Charaktere, die unter der Kontrolle des Servers stehen (also *neutral* sind).

## 5.2.7 Spielende

#### </> Nachricht Definition 14: StatisticsMessage

Autor: Florian Sihler.

verpflichtend  
(OK)

• v. 1

Spezifiziert den Body eines `MessageContainer` des `MessageTypeEnum::STATISTICS`-Typs. Diese Nachricht wird vom Server nach Spielende an alle Spieler- und Zuschauer-Clients gesendet um diesen einen finalen Überblick über das Spielgeschehen zu präsentieren.

```

1 class StatisticsMessage extends MessageContainer {
2     Statistics statistics;
3     UUID winner;
4     VictoryEnum reason;
5     Boolean hasReplay;
6 }

```

Das erste Feld ist optional, da der Server ja keine Statistiken implementieren muss, in diesem Fall kann es leer sein, oder explizit auf `null` gesetzt (also nicht angegeben) werden.

**statistics:** Liefert über `Statistics` ein Array aller `StatisticsEntry`.

**winner:** Enthält den Sieger der Partie.

**reason:** Enthält die Begründung für den Sieger.

**hasReplay:** Gibt an, ob der Server das *Replay* Feature unterstützt und damit, ob er eine `RequestReplayMessage` zulässt/sinnvoll beantworten kann.

### 5.2.8 Kontrollnachrichten

#### </> Nachricht Definition 15: GameLeaveMessage

Autoren: Unbekannt?, Florian sihler.

verpflichtend  
(OK)

• V. 1

Spezifiziert den Body eines `MessageContainer` des `MessageTypeEnum::GAME_LEAVE`-Typs. Nachricht die gesendet wird, wenn ein Spieler das Spiel verlassen möchte, der Ablauf befindet sich in Modell 5.1.2 (Verlassen des Spiels).

```
1 class GameLeaveMessage extends MessageContainer {
2     // No further fields
3 }
```

#### </> Nachricht Definition 16: GameLeftMessage

Autoren: Unbekannt?, Florian sihler.

verpflichtend  
(OK)

• V. 1

Spezifiziert den Body eines `MessageContainer` des `MessageTypeEnum::GAME_LEFT`-Typs. Der Server antwortet damit auf eine `GameLeaveMessage` gewissermaßen an alle Spieler. Der Ablauf befindet sich in Modell 5.1.2 (Verlassen des Spiels).

```
1 class GameLeftMessage extends MessageContainer {
2     UUID leftUserId;
3 }
```

**leftUserId:** Gibt die `UUID` des verlassenden Spielers an.

#### </> Nachricht Definition 17: RequestGamePauseMessage

Autoren: Unbekannt?, Florian sihler.

verpflichtend  
(OK)

• V. 1

Spezifiziert den Body eines `MessageContainer` des `MessageTypeEnum::REQUEST_GAME_PAUSE`-Typs. Der Ablauf findet sich in Modell 5.1.1 (Kontrollnachrichten, Spielpause).

```
1 class RequestGamePauseMessage extends MessageContainer {
2     Boolean gamePause;
3 }
```

**gamePause:** Gibt an, ob das Spiel pausiert (`true`) oder fortgesetzt (`false`) werden soll.

#### </> Nachricht Definition 18: GamePauseMessage

Autoren: Unbekannt?, Florian sihler.

verpflichtend  
(OK)

• V. 2

Spezifiziert den Body eines `MessageContainer` des `MessageTypeEnum::GAME_PAUSE`-Typs. Der Ablauf findet sich in Modell 5.1.1 (Kontrollnachrichten, Spielpause).

```

1 class GamePauseMessage extends MessageContainer {
2     Boolean gamePaused;
3 }

```

**gamePaused:** Gibt an, ob das Spiel pausiert (`true`) oder fortgesetzt (`false`) wird.

#### </> **Nachricht Definition 19:** RequestConfigDeliveryMessage

Autor: Jonas Mayer.

verpflichtend  
(OK)

• V. 1

Spezifiziert den Body eines `MessageContainer` des `MessageTypeEnum::REQUEST_CONFIG_DELIVERY`-Typs. Fordert eine `ConfigDeliveryMessage` vom Server an.

```

1 class RequestConfigDeliveryMessage extends MessageContainer {
2     // Keine weiteren Attribute
3 }

```

#### </> **Nachricht Definition 20:** ConfigDeliveryMessage

Autor: Jonas Mayer.

verpflichtend  
(OK)

• V. 1

Spezifiziert den Body eines `MessageContainer` des `MessageTypeEnum::CONFIG_DELIVERY`-Typs. Antwort des Servers auf eine `RequestConfigDeliveryMessage` vom Client. Die Nachricht enthält alle Konfigurationen, die im aktuellen Spiel verwendet werden.

```

1 class ConfigDeliveryMessage extends MessageContainer {
2     Scenario level;
3     Matchconfig settings; // Partiekonfiguration
4     CharacterInformation[] character_settings;
5 }

```

**level:** Das (aktuell) gespielte Szenario.

**settings:** Die (aktuelle) Partiekonfiguration.

**character\_settings:** Die (gespielten) Charakter-Beschreibungen samt ihrer hauseigenen Eigenschaften.

### 5.2.9 Strikes

In Issue-Thread 64 wird eine Entscheidung zwischen disqualifizierenden und nicht-disqualifizierenden „Strikes“ vermerkt. In den folgenden Fällen wird die Verbindung des Servers *direkt getrennt*, weitere Informationen können dann über die `MessageContainer::debugMessage` erfolgen, allerdings beinhaltet dies Fälle, die bereits vom Client abgedeckt werden müssen (also eigentlich nur bei einer fehlerhaften Implementierung auftreten):

- Spieler, die in der KI-Rolle an einer Partie teilnehmen, werden disqualifiziert, sofern sie Nachrichten senden, welche nicht für sie vorgesehen sind (Spiel verlassen, Spiel pausieren, ...).
- Spieler, welche in der Wahlphase Items anfordern, welche ihnen nicht vorgeschlagen wurden, werden umgehend disqualifiziert.
- Spieler, welche eine ungültige Zusammenstellung von Charakteren und Items (Charaktere nicht vorhanden, Items nicht vorhanden) anfordern, werden disqualifiziert.
- Spieler, welche im Spiel semantisch invalide Nachrichten senden, werden disqualifiziert. Dies schließt Züge auf Felder, auf die nicht gezogen werden kann, ein Einsatz von Items, die ein Charakter nicht

besitzt den Einsatz von Items, der so dem Charakter gemäß den Regeln nicht möglich ist und das Schicken von Nachrichten mit fehlenden notwendigen Feldern mit ein.

In den anderen Fällen, die also zum Beispiel ein Spielerverschulden sein können, wird ein konventioneller **Strike** versendet, wobei dies aktuell nur das Ausbleiben eines Spielerzugs betrifft.

#### </> **Nachricht Definition 21: Strike**

Autor: Marco Deuscher.

verpflichtend  
(OK)

• V. 1

Spezifiziert den Body eines **MessageContainer** des **MessageTypeEnum**: :STRIKE-Typs. Der Server vergibt einen Strike, wenn der Spieler (also auch die KI), welcher *während des Spiels* an der Reihe ist, das Zeitlimit (definiert durch **Matchconfig**: :TurnPhaseLimit) überschritten hat. Bekommt ein Spieler in **strikeMax** aufeinander folgenden Zügen einen Strike, so trennt der Server die Verbindung zu diesem Spieler. Spielt der Spieler nach einem erhaltenen Strike wieder eine aktive Runde, wird der „Strikecounter“ zurückgesetzt.

```
1 class Strike extends MessageContainer {
2     Integer strikeNr;
3     Integer strikeMax;
4     String reason;
5 }
```

**strikeNr:** Gibt die Nummer des aktuell erhaltenen Strikes (Start bei 1) an.

**strikeMax:** Gibt an, nach wie vielen Strike der Server die Verbindung zum Spieler trennt. Ist **strikeNr** = **strikeMax** so wird die Verbindung getrennt.

**reason:** Eine Begründung für den Strike, die vom Client interpretiert oder dem Spieler direkt angezeigt werden kann. Das Format des Feldes ist bisher nicht weiter spezifiziert.

## 5.3 Optionale Komponenten

Dieser Abschnitt behandelt all die Komponenten der Netzwerkkommunikation, deren Umsetzung *nicht* verpflichtend sind – natürlich beinhaltet dies nicht (nur) die Nachricht sondern eine entsprechende Umsetzung im Client...

#### </> **Nachricht Definition 22: RequestReplayMessage**

Autor: Florian Sihler.

optional  
(OK)

• V. 1

Spezifiziert den Body eines **MessageContainer** des **MessageTypeEnum**: :REQUEST\_REPLAY-Typs, mit dem ein *Replay* vom Server angefordert werden kann, sofern dieser ein solches Feature unterstützt (dies wird über **StatisticsMessage**: :hasReplay mitgeteilt). Der Server antwortet mit einer **ReplayMessage** oder verweigert die Anfrage (nur dann, wenn kein Replay unterstützt wird).

```
1 class RequestReplayMessage extends MessageContainer {
2     // Keine weiteren Attribute
3 }
```

#### </> **Nachricht Definition 23: ReplayMessage**

Autor: Florian Sihler.

optional  
(OK)

• V. 1

Spezifiziert den Body eines **MessageContainer** des **MessageTypeEnum**: :REPLAY-Typs, die als Antwort auf eine **RequestReplayMessage** vom Server gesendet wird.

```
1 class ReplayMessage extends MessageContainer {
2     // Metadaten
3     UUID sessionId;
4     Date gameStart;
5     Date gameEnd;
6     UUID playerOneId;
7     UUID playerTwoId;
8     String playerOneName;
9     String playerTwoName;
10    Integer rounds;
11    // Konfigurationsdaten
12    Scenario level;
13    Matchconfig settings;
14    CharacterDescription[] character_settings;
15    // Alle Nachrichten
16    MessageContainer[] messages;
17 }
```

*Hinweis:* die Metadaten lassen sich in gewisser Weise alle aus den *messages* rekonstruieren, sie helfen dem Client allerdings, schnell die wichtigsten Informationen des Replays aufzugreifen und anzugeben:

**sessionId:** Die über die *HelloReplyMessage* erhaltene sessionId.

**gameStart:** Der Startzeitpunkt des Spiels.

**gameEnd:** Der Endzeitpunkt des Spiels.

**playerOneId:** Die *UUID* des ersten Spielers.

**playerTwoId:** Die *UUID* des zweiten Spielers.

**playerOneName:** Der Name des ersten Spielers.

**playerTwoName:** Der Name des zweiten Spielers.

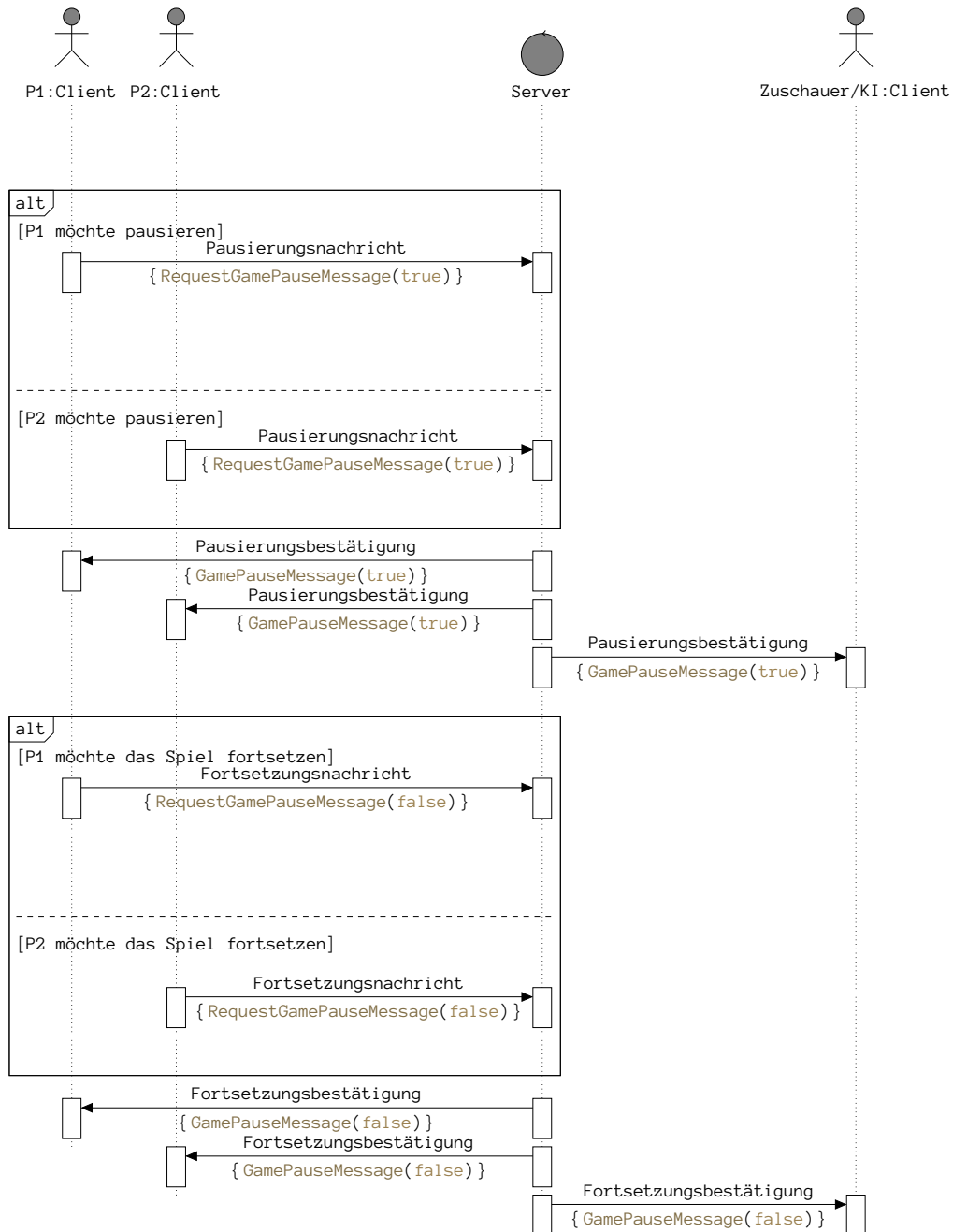
**rounds:** Die Anzahl der gespielten Runden (inklusive der Siegesrunde)

**level:** Das gespielte Szenario.

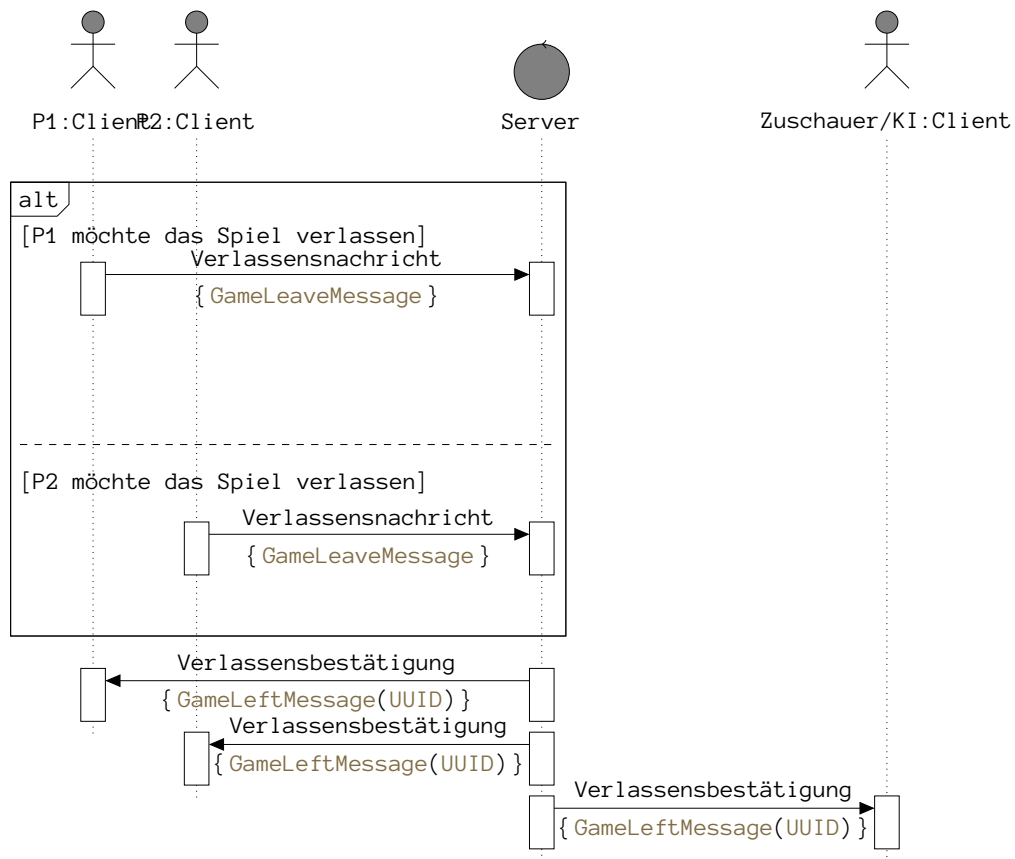
**settings:** Die Partiekonfiguration.

**character\_settings:** Die Charakter-Beschreibungen.

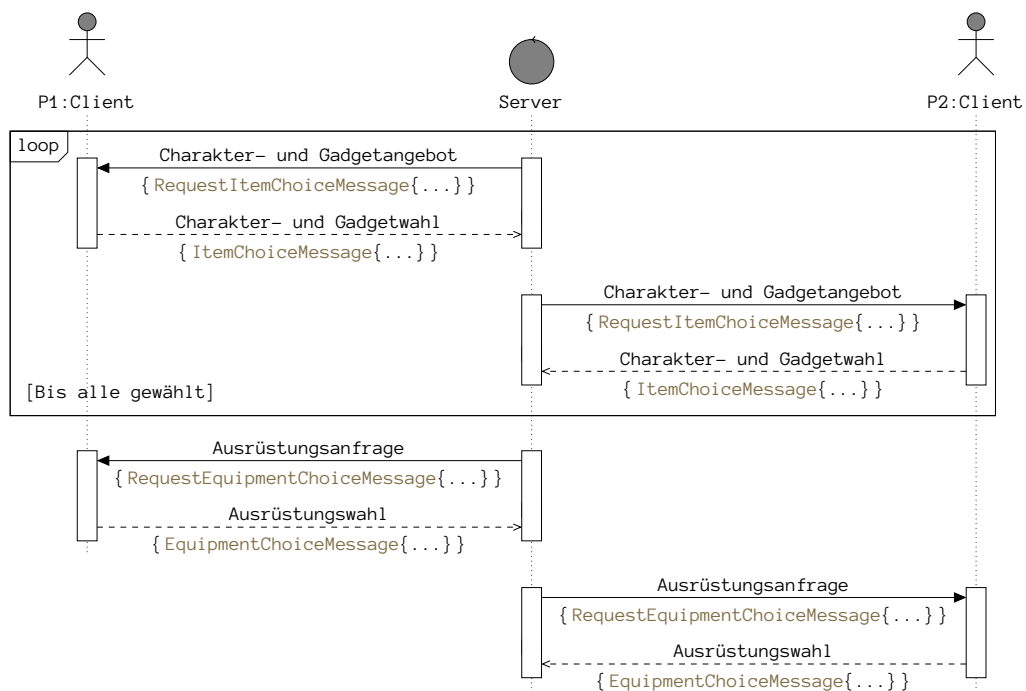
**message:** Eine Sammlung *aller* an den Client gesendeten Nachrichten (KISS und so...), dem Client bleibt es überlassen wie er mit den eventuell auftretenden Redundanzen umgeht.



**Modell 5.1.1:** Die Kontrollnachrichten für die Anforderung einer Spielpause.



**Modell 5.1.2:** Die Kontrollnachrichten zum Verlassen des Spiels



**Modell 5.1.3:** Die Kontrollnachrichten zur Steuerung der Ausrüstungsphase.



## Schnittstellen

### 6.1 Kommandozeile

*Hinweis: Die Reihenfolge der Parameter darf nicht gebunden sein, es darf also der Port sowohl vor als auch nach der Schwierigkeitsangabe definiert werden. Ob aber sinnfreie Parameterangaben, wie das Aufführen einer Schwierigkeitsangabe und das Aufrufen der Hilfe oder Mehrfachangaben von Parametern, ... mit einem Fehler behandelt werden oder nicht, bleibt der Implementation überlassen.*

Parameter	Alias
--config-charset	-c <path>
--config-match	-m <path>
--config-scenario	-s <path>
--x <key> <value>	
o --help	-h
o --port	-p <port>
o --verbosity	-v <int>

**(a)** Parameter für den Server.

Parameter	Alias
--address	-a <ip>
--x <key> <value>	
o --help	-h
o --port	-p <port>
o --verbosity	-v <int>
o --name	-n <string>
o --difficulty	-d <int>

**(b)** Parameter für den KI-Client.

**Tabelle 6.1:** Kurzübersicht über die geforderten Parameter, wobei die mit „o“ markierten optional sind.

#### 6.1.1 Server

Das Starten des Servers soll in der Kommandozeile mittels „server###“ möglich sein, wobei „###“ hierbei der auf drei Stellen aufgefüllten Teamnummer entspricht. Der Server von *Team 20* wird also mittels „server020“ angesprochen.

#### >\_ **Kommandozeilenparameter 1:** Server, Commandline-Interface

Autor: Julian Holl.

verpflichtend  
(OK)

• V. 1

Die in Tabelle 6.1a mit „o“ vermerkten Parameter sind jeweils *optional*. Im Folgenden werden diese Parameter nun etwas genauer beleuchtet:

**config-charset:** Fordert nachfolgenden einen absoluten oder relativen Pfad zur `characters.json`.

**config-match:** Fordert nachfolgenden einen absoluten oder relativen Pfad zur Partiekonfiguration.  
on.match.

**config-scenario:** Fordert nachfolgenden einen absoluten oder relativen Pfad zur Szenario-  
name.scenario.

**x:** Ermöglicht es optionale Schlüssel-Wert-Paare zu übermitteln, die verfügbaren Einstellungen sind proprietär.

**help:** Zeigt die Liste aller unterstützten Parameter an. Ist dann verpflichtend, wenn neben den in Tabelle 6.1a  
vermerkten Parametern noch weitere zur Verfügung stehen.

**port:** Fordert nachfolgend eine (gültige) Portnummer, wird keine angegeben oder sollte dieser Parameter  
nicht unterstützt werden, so wurde als Portnummer die 7007 vereinbart.

**verbosity:** Fordert nachfolgenden eine Ganzzahl, die die Ausführlichkeit in den Logs angibt. Diese soll von  
1 an zunehmen, das Maximum *m* kann selbst frei gewählt werden, soll aber auch mit 0 zur Verfügung  
stehen. Der Aufruf „-v 0“ soll also automatisch die höchste „Verbosität“ auswählen.

### 6.1.2 KI-Client

Das Starten des KI-Clients soll in der Kommandozeile mittels „ki####“ möglich sein, wobei „####“ hierbei der  
auf drei Stellen aufgefüllten Teamnummer entspricht. Der KI-Client von *Team 20* wird also über „ki020“  
angesprochen.

#### >\_ **Kommandozeilenparameter 2:** KI-Client, Commandline-Interface

Autor: Julian Holl.

verpflichtend  
(OK)

• v. 1

Die in Tabelle 6.1b mit „o“ vermerkten Parameter sind jeweils *optional*. Im Folgenden werden diese Parameter  
nun etwas genauer beleuchtet:

**address:** Gibt die IP-Adresse an, über die der Server zu erreichen ist, also des Servers, über den gespielt  
werden soll.

**x:** Ermöglicht es optionale Schlüssel-Wert-Paare zu übermitteln, die verfügbaren Einstellungen sind proprietär.

**help:** Zeigt die Liste aller unterstützten Parameter an. Ist dann verpflichtend, wenn neben den in Tabelle 6.1b  
vermerkten Parametern noch weitere zur Verfügung stehen.

**port:** Fordert nachfolgend eine (gültige) Portnummer, wird keine angegeben oder sollte dieser Parameter  
nicht unterstützt werden, so ist als Portnummer die 7007 vereinbart worden.

**verbosity:** Fordert nachfolgenden eine Ganzzahl, die die Ausführlichkeit in den Logs angibt. Diese soll von  
1 an zunehmen, das Maximum *m* kann selbst frei gewählt werden, soll aber auch mit 0 zur Verfügung  
stehen. Der Aufruf „-v 0“ soll also automatisch die höchste „Verbosität“ wählen.

**name:** Fordert nachfolgend den Name des Clients im Spiel, wobei dieser mindestens zwei Zeichen lang  
sein muss. Sollte dieser Parameter nicht unterstützt oder nicht angegeben werden, so ist „team####“  
zu wählen. Die KI für *Team 20* bekäme so den Namen „team020“. *Hinweis:* dies hat nichts mit  
dem Namen des KI-Programms selbst zu tun, sondern nur damit, wie sich der KI-Client im Spiel  
bezeichnet.

**difficulty:** Fordert nachfolgend eine Ganzzahl, die den Schweregrad der KI anpassbar macht, wobei wie bei  
der *verbosity* von 1 an mit der leichtesten Schwierigkeitsstufe bis zu einem beliebigen Maximum *m*  
gewählt werden kann und 0 wieder automatisch die höchste Schwierigkeitsstufe auswählt. Es kann  
damit also auch nur eine Schwierigkeitsstufe implementiert werden.

# A

## Versionierung

### A.1 Changelog

#### A.1.1 Changelog für Version 1 (20. Februar 2020)

- ❗ Erstes Plenum zum Standard, dritte Sitzung
- + *Dokumentrechtfertigung*: Initiale Rechtfertigung von Florian Sihler.
- + *Kommentar zur Versionierung*: Neues Element!
- + *Kommentar zu JSON Schema*: Neues Element!
- + *Begriff: Container*: Neues Element!
- + *Begriff: Docker*: Neues Element!
- + *Begriff: JSON*: Neues Element!
- + *Begriff: KI-Client*: Neues Element!
- + *Begriff: Strike*: Neues Element!
- + *Begriff: Phasen des Spiels*: Neues Element!
- + *Begriff: Spielstart*: Neues Element!
- + *Begriff: Wahlphase*: Neues Element!
- + *Begriff: Ausrüstungsphase*: Neues Element!
- + *Begriff: Spielphase*: Neues Element!
- + *Begriff: Gadget*: Neues Element!
- + *Begriff: Playable Character*: Neues Element!
- + *Begriff: Non-Playable Character*: Neues Element!
- + *Begriff: Szenario*: Neues Element!
- + *Begriff: Feld*: Neues Element!
- + *Begriff: Charakter*: Neues Element!
- + *Begriff: Operation*: Neues Element!
- + *Begriff: Item*: Neues Element!
- + *Partieconfiguration*: Existenz einer solchen Datei gefordert.
- + *Szenariobeschreibung*: Existenz einer solchen Datei gefordert.
- + *Charakterdefinition*: Existenz einer solchen Datei gefordert.
- + *RoleEnum*: Neues Element!
- + *PropertyEnum*: Die Eigenschaften, ins Englische übersetzt.
- + *GadgetEnum*: Die Eigenschaften, ins Englische übersetzt.
- + *Gadget*: Neues Element!
- + *Gadget: WiretapWithEarplugs*: Neues Element!
- + *Gadget: Cocktail*: Neues Element!
- + *FieldStateEnum*: Neues Element!

- ⊕ *Field*: Neues Element!
- ⊕ *FieldMap*: Neues Element!
- ⊕ *State*: Neues Element!
- ⊕ *Scenario*: Neues Element!
- ⊕ *Matchconfig*: Neues Element!
- ⊕ *OperationEnum*: Neues Element!
- ⊕ *Kommentar zu Operationen*: Klarstellung nach Issue-Thread 28
- ⊕ *BaseOperation*: Neues Element!
- ⊕ *Operation*: Neues Element!
- ⊕ *GadgetAction*: Neues Element!
- ⊕ *GambleAction*: Neues Element!
- ⊕ *PropertyAction*: Neues Element!
- ⊕ *Movement*: Neues Element!
- ⊕ *Exfiltration*: Neues Element!
- ⊕ *ErrorTypeEnum*: Neues Element!
- ⊕ *StatisticsEntry*: Neues Element!
- ⊕ *Statistics*: Neues Element!
- ⊕ *VictoryEnum*: Neues Element!
- ⊕ *CharacterDescription*: Ein Charakter mit Beschreibung, Geschlecht und Eigenschaften.
- ⊕ *GenderEnum*: Drei Geschlechter.
- ⊕ *Character*: Neues Element!
- ⊕ *NPCEnum*: Zwei NPCs.
- ⊕ *MessageContainer*: Neues Element!
- ⊕ *MessageTypeEnum*: Neues Element!
- ⊕ *Nachricht: HelloMessage*: Neues Element!
- ⊕ *Nachricht: HelloReplyMessage*: Neues Element!
- ⊕ *Nachricht: ReconnectMessage*: Neues Element!
- ⊕ *Nachricht: ConnectionErrorMessage*: Neues Element!
- ⊕ *Nachricht: GameStartedMessage*: Neues Element!
- ⊕ *Nachricht: Charakter- und Gadgetangebot*: Neues Element!
- ⊕ *Nachricht: Charakter- oder Gadgetwahl*: Neues Element!
- ⊕ *Nachricht: Start der Ausrüstungsphase*: Neues Element!
- ⊕ *Nachricht: Ausreistungswahl*: Neues Element!
- ⊕ *Nachricht: GameStatusMessage*: Neues Element!
- ⊕ *Nachricht: RequestGameOperationMessage*: Neues Element!
- ⊕ *Nachricht: GameOperationMessage*: Neues Element!
- ⊕ *Nachricht: SpectatorRevealMessage*: Neues Element!
- ⊕ *Nachricht: StatisticsMessage*: Neues Element!
- ⊕ *Nachricht: GameLeaveMessage*: Neues Element!
- ⊕ *Nachricht: GameLeftMessage*: Neues Element!
- ⊕ *Nachricht: RequestGamePauseMessage*: Neues Element!
- ⊕ *Nachricht: RequestConfigDeliveryMessage*: Neues Element!
- ⊕ *Nachricht: ConfigDeliveryMessage*: Neues Element!
- ⊕ *Nachricht: Strike*: Generelles Strike-Format
- ⊕ *Nachricht: RequestReplayMessage*: Neues Element!
- ⊕ *Nachricht: ReplayMessage*: Neues Element!
- ⊕ *Kommandozeile: Server*: Neues Element!
- ⊕ *Kommandozeile: KI-Client*: Neues Element!

### A.1.2 Changelog für Version 2 (16. März 2020)

- ❗ Zweite Iteration zur Konsistenzgewährleistung
- ⊕ *CharacterInformation*: Server muss `characterId` den Clients mitteilen
  - *Nachricht: HelloReplyMessage*: Verwendet nun *CharacterInformation* für UUIDs (Issue 79)
- ⊕ *Nachricht: GamePauseMessage*: Neues Element!
  - *Nachricht: ConfigDeliveryMessage*: Verwendet nun *CharacterInformation* für UUIDs (Issue 79)
  - *Nachricht: Strike*: Regeln gelten auch für die KI