03*

Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim Geisenheim

Hochschule **RheinMain**
Medieninformatik

24. Mai 2012

Fachbereich Design Informatik Medien

Designspezifikation Campusadventure

im Fach Softwaretechnik im SS2012

Vorgelegt von

Dominik Schuhmann

Tino Landmann

Simon Hardt

David Gens

bei Prof. Dr. Wolfgang Weitz

am 24. Mai 2012

in der Version 182.

- Neues Spec + Version

B:04.06.12

Rev	Autor	Datum	Beschreibung	Dateien
182	dgens001	24-05-2012	design final	/design/meta/svnlog.tex /design/design.pdf /design/kapitel/laufzeitsicht.tex
181	tland001	24-05-2012	lautfzeitdiagram bla2	/design/kapitel/laufzeitsicht.tex
180	dgens001	24-05-2012	svn tag mal gesetzt.. wäre clever gewesen	/design/kapitel/bausteinsicht.tex /design/meta/svnlog.tex /design/kapitel/musterundkonzepte.tex /design/meta/glossar.tex /design/design.pdf /design/meta/titel.tex /design/kapitel/laufzeitsicht.tex /design/kapitel/einfuehrung.tex /design/design.tex /design/kapitel/entwurfsentscheidungen.tex

Auflagenbörde
– 1x Designtapete (LED)

Inhaltsverzeichnis

1 Einführung

In folgendem Dokument werden die grundlegenden Designentscheidungen des Softwaretechnikprojektes definiert und spezifiziert. Die Anwendung wird in Java entwickelt.

Im Abschnitt 2 werden die verschiedenen Bausteine der Anwendung erklärt und näher beschrieben. Abschnitt 3 befasst sich dann mit deren Zusammenspiel - dargestellt anhand verschiedener Use-Cases. Die angewendeten Muster und Konzepte werden in Abschnitt 3 behandelt.

Bezug auf beigefügte Spec (Lu. Oberon) wäre gut

2 Bausteinsicht

In diesem Abschnitt soll die Struktur der Spiele-Software dargestellt werden. Im Vordergrund stehen die Bausteine, deren Beziehungen und eine Übersicht darüber mit welchen Klassen und Paketen die zuvor erarbeiteten Anforderungen umgesetzt werden. Die Übersicht wird dabei schrittweise verfeinert.

2.1 Übersicht

Das *Spiel* ist grundlegend in drei Schichten eingeteilt, Darstellungs-, Logik- und Datenschicht. Die genauen Zusammenhänge werden im Folgenden erläutert und durch das unten stehende Schichten-Diagramm sowie die angehängte UML-"Tapete"verdeutlicht.

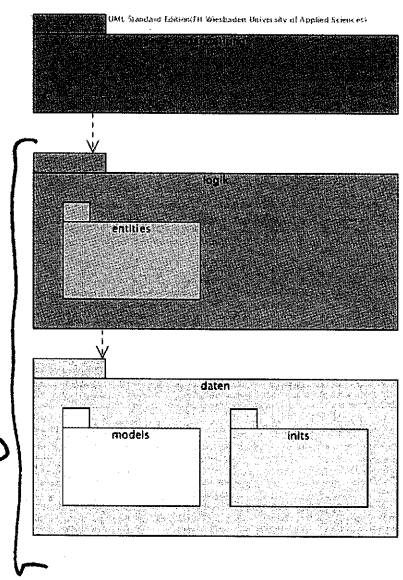


Abbildung 2.1: Architekturdiagramm

2.2 Darstellung

Die Darstellungsschicht beinhaltet verschiedene Klassen, die im Zusammenspiel letztlich die *Anzeige* füllen. Die benötigte Kommunikation erfolgt dabei jeweils über *PropertyChange-Listener*, sodass Klassen der Logikschicht deren Pendants der Darstellungsschicht benachrichtigen können.

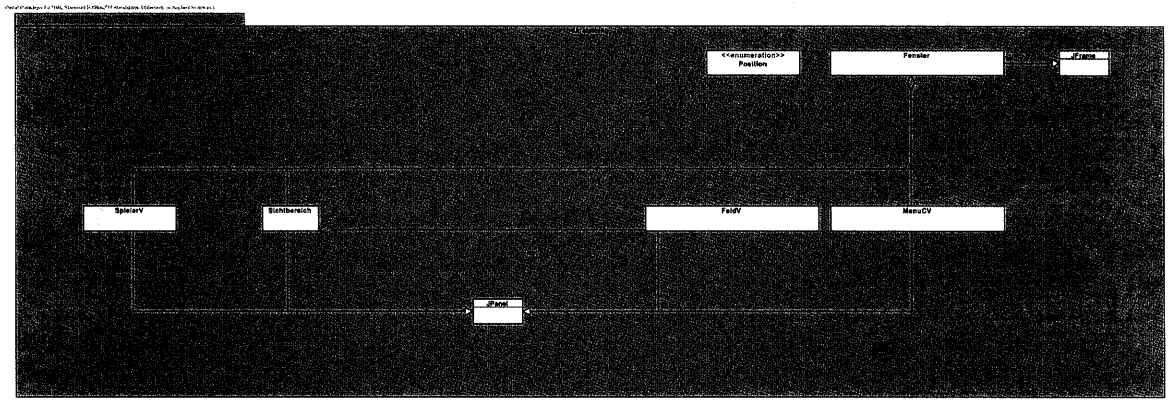
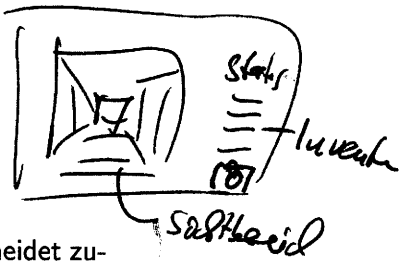


Abbildung 2.2: Darstellungsschicht

2.2.1 Fenster

Das Fenster erweitert das Java *JFrame* und zeigt die beiden Bereiche *Statusbereich* (SpielerV) und *Sichtbereich* (Szene) an. Falls sich der *Spieler* in einem Menu befindet, zeigt das Fenster statt der beiden o.g. Bereiche nur das passende Menu (MenuCV) an.



2.2.2 MenuCV

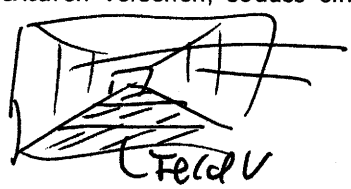
Der *Menümodus* bietet dem Spieler verschiedene Optionen an. Das Menu unterscheidet zudem selbst, welches Untermenu gezeigt wird. Dazu stehen z.B. die Funktionen *drawPause()* oder *drawSave()* bereit.

2.2.3 Sichtbereich

Der Sichtbereich zeigt die *Spielwelt* in einer Pseudo-3D-Perspektive an. Es werden dabei immer die nächsten drei *Felder* in *Blickrichtung* des Spielers mit ihren *GameObjects* dargestellt. Die anderen Felder eines Raumes werden ohne evtl. GameObjects gezeigt.

Die Darstellung der Felder mit GameObjects erfolgt über drei *FeldV*, welche die drei vorderen Felder in Blickrichtung des Spielers darstellen. Diese werden von *Sichtbereich* mit dem darzustellenden Feld und der benötigten Position des Feldes (vorne, mitte, hinten).

Alle anderen *JPanels* des Sichtbereichs werden je nach Position des Spielers und dessen Blickrichtung mit verschiedenen Texturen versehen, sodass ein räumlicher Eindruck entsteht.



2.2.4 FeldV

Die eigentliche Darstellung von GameObjects auf den drei vorderen Felder wird von *FeldV* übernommen, welches ein Raster anzeigt.

Dieses Raster zeigt die einzelnen *Item-*, *Non-Player-Character (NPC)-* und *Eingang*sanordnungen des Feldes. Dazu wird die *getObjects*-Methode der Feld-Klasse vom View aufgerufen. Rückgabe ist hierbei eine Liste von GameObjects, von welchen mit *getImg()* deren Bild und evtl. Name angezeigt werden kann.

2.2.5 SpielerV

Der *SpielerV* zeigt die Statusleiste an. Diese wird je nach Bedarf mit den passenden Inhalten gefüllt. Es wird z.B. das *Inventar* und der Avatar des Spielers gezeigt. Tritt der Spieler in Interaktion mit einem GameObject, wird auch dazu der passende View gewählt. Die Auswahl erfolgt mit Hilfe der Funktionen *drawInventar()*, *drawInteraktion()*, *drawHand()* und *drawAvatar()*. Was zu tun ist, wird von *Spieler* durch ein *PropertyChangeEvent* angestoßen.

Infobereich Im *Infobereich* werden je nach Situation das Inventar des Spielers mit dessen Slots und den darin befindlichen *Gegenständen* gezeigt oder textuelle Information bei Interaktionen dargestellt.

Handbereich Wählt der Spieler die Interaktion mit einem Gegenstand wird im *Handbereich* das Bild des jeweiligen Items angezeigt. Im Normalfall wird eine leere Hand gezeigt.

Statusbereich Im Statusbereich kann der Spieler verschiedene Informationen zu sich selbst oder einem NPC erhalten. Meist wird ein Portät des jeweiligen Charakters dargestellt.

Interaktion

Die Methode *drawInteraktion()* unterscheidet zudem den Typ des gewählten GameObjects und füllt Infobereich, Statusbereich und Handbereich mit den nötigen Inhalten.

Interaktion mit Gegenständen Bei der Interaktion mit einem Gegenstand wird dessen Beschreibung und die möglichen Interaktionen, die der Spieler damit tätigen kann, angezeigt. Er kann diese beispielsweise in sein Inventar aufnehmen. Nimmt der Spieler einen Gegenstand auf, wird er zuerst in der Hand „zwischengespeichert". Das Bild des Gegenstands wird daher im Handbereich angezeigt.

Interaktion mit NPCs Spricht der Spieler hingegen einen NPC an, werden im Statusbereich dessen Bild und Name gezeigt. Anhand des *DialogModel*, stellt *SpielerV* den Dialog mit Text, Antworten und evtl. einem Item dar, welches der Spieler durch den letzten Dialog erhalten hat.

Inventar

Der Spieler besitzt ein Inventar mit 10 Slots und eine *Hand*. Diese kann lediglich einen einzigen Gegenstand aufnehmen und dient als „Zwischenspeicher" bei einer Interaktion. Dargestellt werden Inventar und Hand jeweils von *drawInventar()* und *drawHand()* der Klasse *SpielerV*. Über den Gegenständen im Inventar werden Nummern angezeigt, die auf den passenden Index verweisen. Drückt der Spieler eine dieser Nummern, wird der Gegenstand aus dem Inventar in die Hand genommen und beide Bereiche neu gezeichnet.

2.3 Logik

In der Logik-Schicht befinden sich das Paket *entities* und die Controller *TastaturC* und *GameC*.

Die hierin befindlichen Klassen sind jeweils möglichst autark - jegliche Funktionen, Interaktionen oder mögliches Zusammenspiel zwischen den Klassen/Objekten geschieht ohne Zutun einer dritten Instanz.

Ein Beispiel: Möchte der Spieler einen Gegenstand aufnehmen, wird die gesamte Logik vom Spieler selbst übernommen - es gibt keine extra Klasse, die sich um den Ablauf kümmert. Zur Kommunikation zwischen den Klassen wird das Interface *interagiere* verwendet.

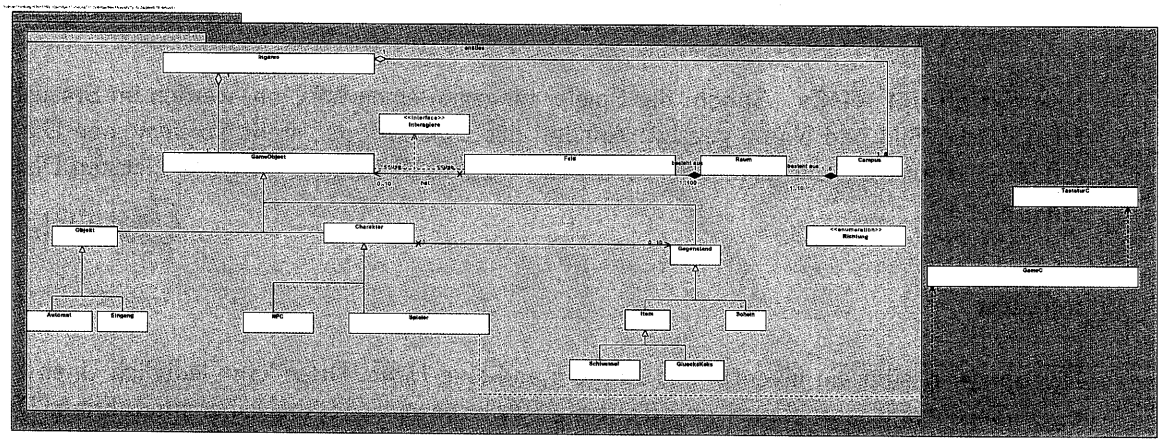


Abbildung 2.3: Logikschicht

2.3.1 Entities

Das Paket *entities* umfasst alle Klassen, die autark Funktionen erfüllen. Die evtl. benötigten Models sind im Paket *models* der Datenschicht ausgelagert. Es wird hierbei das MVC-Pattern verwendet, um eine saubere Trennung zu gewährleisten.

Ingame Das *Ingame*-Objekt beinhaltet Referenzen zu allen anderen Objekten des *entities* Pakets. Das hat den Vorteil, dass beim Speichern nur dieses Objekt serialisiert werden muss und alle anderen Objekte automatisch mit gespeichert werden.

Feld Das *Feld* bildet die zentrale Anlaufstelle für *Spieler* und *FeldV*, sodass es möglich ist, Gegenstände aufzunehmen, abzulegen und anzuzeigen. Die Kommunikation läuft über das Interface *interagiere*.

Raum Der *Raum* grenzt eine Menge von Feldern ein und macht sie zugänglich oder nicht. Hierdurch ist es beispielsweise möglich, dass der Spieler nur Räume betritt, für die er einen Schlüssel bei sich trägt. ↳ wie soll/will der/die das? "nimmt"? Wo steht?

Campus Der *Campus* umfasst alle Räume und grenzt die Spielwelt ein. Der Spieler kann sich nur innerhalb eines Campus bewegen.

GameObject Die Klasse GameObject stellt die grundlegende Klasse für alle in der Spielwelt lebenden *Objekte* dar. GameObjects haben einen Namen und eine textuelle Beschreibung sowie ein Bild (an Stelle eines eigenen Views).

Charakter *Charakter* sind menschliche Spielfiguren, die ein eigenes Inventar haben (also Gegenstände aufnehmen können) und mit mit anderen Charakteren in Dialog treten können.

NPC *Non-Player-Characters (NPCs)* sind vom Computer gesteuerte Charaktere, die zusätzlich einen *Dialog* haben.

Spieler Die *Spieler*-Klasse ist zentrales Objekt im Spiel, denn sie stellt die virtuelle Repräsentation des menschlichen Spielers dar. Jegliche Art von Interaktion wird von Spieler (über das *interagiere*-Interface) angestoßen. Zusätzlich kann der Spieler sich bewegen.
↳ das unfreds implementiert wird (S. Tapete)

Gegenstand Gegenstände sind Objekte die der Spieler aufnehmen kann. Gegenstände liegen z.B. auf Feldern. Aber auch Charaktere können dem Spieler Gegenstände im Verlauf eines Dialog überreichen. Gegenstände können beliebig spezialisiert sein und so zum Spielinhalt beitragen.

Objekt Objekte sind im Gegensatz zu Gegenständen nicht aufnehmbar. Sie belegen aber einen Slot auf dem jeweiligen Feld. Ein Beispiel für ein spezialisiertes Objekt ist der Automat, welcher bei Interaktion Gegenstand liefert.
↳ bei Feld noch erläutern

Eingang *Eingang* ist ein *Objekt* und kann offen oder verschlossen sein und ist mit einem passenden *Schlüssel* aufschließbar. WIE fest das?

Beschreibungen sind zusammenfassend, aber teils wenig detailliert – man muss noch der Lektüre aber klar wissen, wie's umgesetzt werden soll.

weg Satz

Richtung Ein Enum um die Blickrichtung des Spielers zu definieren. Wird z.B. von Klassen der Darstellungsschicht verwendet, um die korrekten Felder im Sichtbereich des Spielers zu finden.

2.3.2 Controller

(≠MVC)

Die Controller liegen in der Logikschicht, verwenden aber kein eigenes Package. Sie kümmern sich um den Wechsel der Ansichten (Menu, Ingame) und um Tastatureingaben des Spielers.

In der Logikschicht???

TastaturC und GameC Zur Eingabe über die Tastatur hört *TastaturC* und leitet die Eingaben an *GameC* weiter. *GameC* benachrichtigt dann, abhängig davon ob gerade ein Menu angezeigt wird oder nicht, die passende Klasse (*Spieler* oder *MenuCV*) via *PropertyChangeEvent*.

2.4 Daten

Die Datenschicht beinhaltet die Pakete *models* und *init*. Hier befinden sich alle Klassen, die selbst keine direkten Spiel-Funktionen anbieten. Meist werden die Models durch entsprechende Factories aus Konfigurationsdateien erstellt und stehen dann Klassen aus der Logikschicht zur Verfügung.

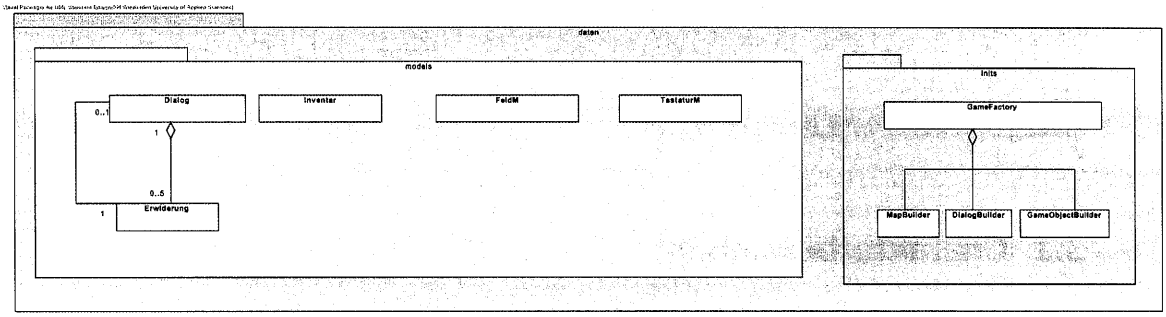


Abbildung 2.4: Datenschicht

Dialog und Erwiderung Die Dialoge bilden das Rückgrat der Interaktion mit NPCs. Der Gesprächsablauf wird dabei allerdings von Klassen in der Logikschicht übernommen. Dialoge beinhalten daher nur Referenzen zu Erwiderungen und evtl. Geschenken, die der Spieler erhält. Stößt der Spieler auf einen Dialog, werden ihm mögliche Erwiderungen angeboten, die er wählen kann. Jede Erwiderung leitet dabei auf einen neuen Dialog oder auf das Ende der Unterhaltung.

Inventar Das *Inventar* bietet nur rudimentäre Funktionen zum Füllen und Entnehmen von Gegenstände und wird einem Charakter als Model mitgegeben.

FeldModel Jedes Feld aus der Logikschicht erhält ein *FeldM*, welches alle nötigen Daten aufnimmt und rudimentäre Funktionen (z.B. ob das Feld begehbar ist) anbietet. Möchte der Spieler einen Gegenstand aufnehmen oder ablegen, werden diese ähnlich zu einem Inventar im *FeldModel* verwaltet.

Factories und Builder Die Klassen im Paket *init* sind hauptsächlich Factories, die die benötigten Models aus Konfigurationsdateien erstellen und die passenden Models bereitstellen.

3 Laufzeitsicht

Dieser Abschnitt stellt das Zusammenwirken der Bausteine zur Laufzeit dar. Es wird dargelegt, wie die zuvor erarbeiteten Anforderungen erfüllt werden, wie die Klassen und Bausteine erzeugt, benutzt und beendet werden.

3.1 Charakterbewegung

Wenn der Spieler eine der Steuerungstasten 'W', 'S', 'A' und 'D' für vor- oder rückwärtige bzw. links- oder rechtsseitige Bewegung (in dieser Reihenfolge) betätigt, wird diese Aktion von *TastaturC* per Keyboard-Listener registriert. Dieser feuert ein Event in Richtung des Spielers, der zuvor beim *TastaturC* als Listener angemeldet wurde.

Die *Spieler*-Klasse wird aufgefordert ihre *gehe()*-Methode aufzurufen mit der entsprechenden *Richtung* in Form eines enum-Wertes - den Himmelsrichtungen - als Funktionsparameter. Bevor die Aktion durchgeführt werden kann, bedarf es einer Überprüfung der Zugänglichkeit des betroffenen Nachbarfeldes. Bevor die Aktion durchgeführt werden kann, bedarf es einer Überprüfung der Zugänglichkeit des betroffenen Nachbarfeldes. Felder implementieren eine *begehbar()*-Methode, die Zugänglichkeitsinformationen für jede Richtung in einem Array verwaltet und für die angefragte Bewegung einen Wahrheitswert zurückliefert.

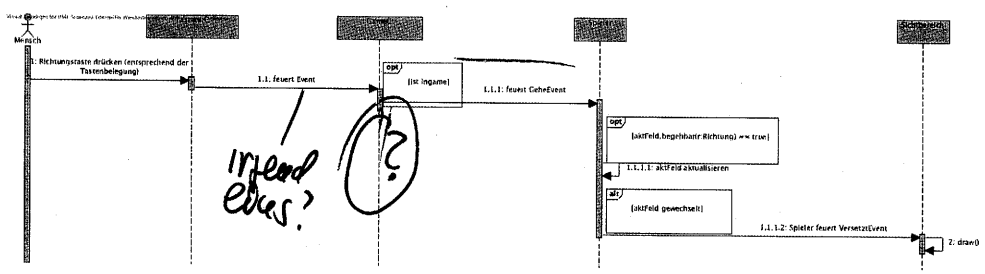


Abbildung 3.1: Charakterbewegung

Ist das Feld zugänglich, wird der Spieler versetzt und der Sichtbereich wird unter Verwendung der Darstellungsinformationen des neuen Feldes aktualisiert. Ist das Feld nicht begehbar, wird eine Exception geworfen und der Spieler wird per Textanzeige darüber informiert. Auch

die Tasten für das Drehen nach links und rechts - 'Q' bzw. 'E' - feuern Events, die *Spieler*-Klasse veranlassen, ihre *drehe()*-Methode aufzurufen. Auch sie erhält die gewünschte *Richtung* als Parameter. Das Drehen verändert die Darstellung der Felder, die vor dem Spieler liegen, nicht aber die des Feldes auf dem er sich aktuell befindet.

3.2 Interaktion

Betritt der Spieler ein Feld auf dem sich ein NPC, ein Eingang zu einem benachbarten Raum oder Gegenstands befinden, hat er die Möglichkeit, in letzterem Fall diese aufzunehmen bzw., in den beiden ersteren, mit ihnen zu interagieren.

Der Spieler verwendet das *interagiere*-Interface um mit *Feld* und *NPC* zu interagieren. Dies geschieht sobald der *TastaturC* die Betätigung der Interaktionstaste registriert. Der Spieler bekommt zunächst, in textueller Form, eine Auflistung der möglichen Interaktionspartner angezeigt. Aus diesen kann nun per Zifferntaste einer ausgewählt werden.

3.2.1 mit einem NPC

Jeder von diesen besitzt selbst eine eigene *interagiere()*, welche festlegt, was passiert, wenn er als Interaktionspartner gewählt wird. Grundsätzlich hat jeder *NPC* einen *Dialog*. Der Dialog liegt in Form einer Baumstruktur vor, aufgebaut aus Dialoge und *Erwiderungen*.

Analog zur Auswahl des Interaktionspartners erfolgt die Wahl der Erwiderung über die Zifferntasten. Manche NPCs erwarten Gegenstände vom Spieler. Im Laufe des Dialoges würde an einem festgelegten Knoten des Baumes überprüft, ob der entsprechende Gegenstand im Handslot des Spielers vorhanden ist. Ausgehend vom Ergebnis dieser Überprüfung, wird entlang des entsprechenden Astes weiter durch die Baumstruktur navigiert.

In der Basisversion der Anwendung ist vorgesehen, dass der Spieler die *interagiere()*-Methode des NPCs mit sich selbst als Funktionsparameter aufruft. Hiermit wird ihm signalisiert, dass ein Dialog gestartet werden soll. Für Erweiterungen sind alternative Interaktionsmöglichkeiten denkbar, ausgelöst durch Übergabe weiterer Parameter, wie beispielsweise einer Waffe im Handslot des Spielers.

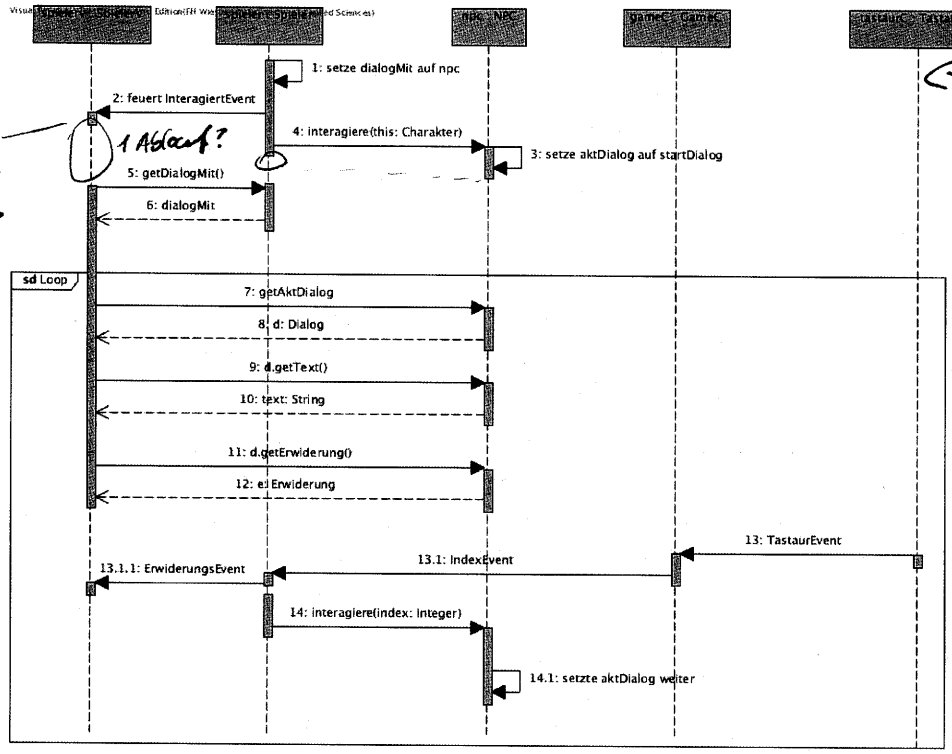
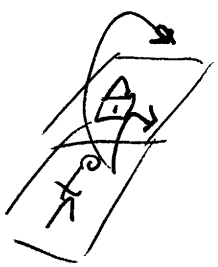


Abbildung 3.2: Dialog mit NPC

3.2.2 mit einem Objekt



Ein unverschlossener Eingang, dessen *interagiere()*-Methode aufgerufen wird, öffnet sich und gibt den dahinterliegenden Weg frei. Ist er aber verschlossen, würde der Eingang, ähnlich wie der NPC, einen Gegenstand erwarten. Sinnvollerweise eine Art von *Schluessel*, je nach Öffnungsmechanismus. Das matchen von *Schluessel* und *Eingang* geschieht anhand ihrer Nummer.

3.2.3 mit einem Gegenstand

In diesem Fall findet keine Interaktion wie in den ersten beiden Fällen statt. Will der Spieler einen Gegenstand vom *Feld* aufnehmen, auf dem er sich aktuell befindet, geschieht das durch das *interagieren*-Interface.

Die Referenz im *FeldM* auf das entsprechenden Gegenstand wird gelöscht und eine neue auf den Handslot des Spielers hergestellt. Dieser muss zum Aufnehmen eines neuen Gegenstands frei sein. Ist er es nicht, kommt es zur Exception. Für spätere Versionen der Anwendung ist

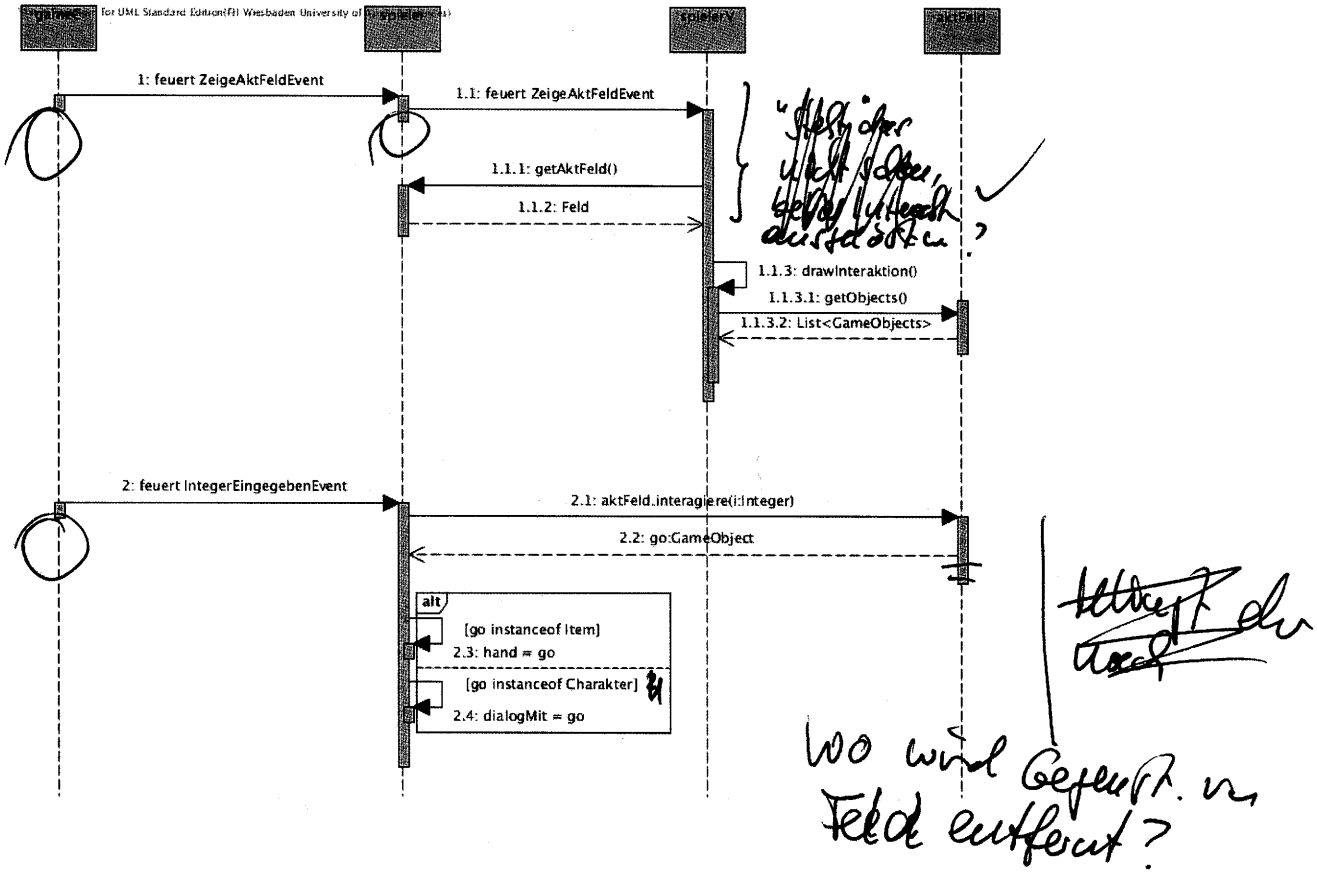


Abbildung 3.3: Gegenstand aufnehmen

es denkbar einen zweiten Handslot einzuführen, der es dann erlaubt, zwei in den Händen befindliche Gegenstände über ihre *interagiere()*-Methode zu kombinieren.

3.2.4 mit dem Inventar

Möchte der Spieler ein Item aus seinem Handslot in sein Inventar ablegen, geschieht dies ähnlich wie bei der Übergabe eines Items vom Feld an den Handslot. Die Referenz zwischen Handslot und Item wird entfernt, während zwischen Inventar und Item eine neue geschaffen wird. Soll ein Item aus dem Inventar in den Handslot genommen werden, erhält der Spieler beim Betätigen der *inDieHand*-Taste, wieder in textueller Form, eine Auswahl der Gegenstände, die sich in seinem Inventar befinden. Per Zifferntaste kann ausgewählt werden. Die Referenzen werden entsprechend aufgelöst. Analog zum Feld stellt das Inventar sowohl eine nehmen-, als auch eine *geben*-Methode zur Verfügung.

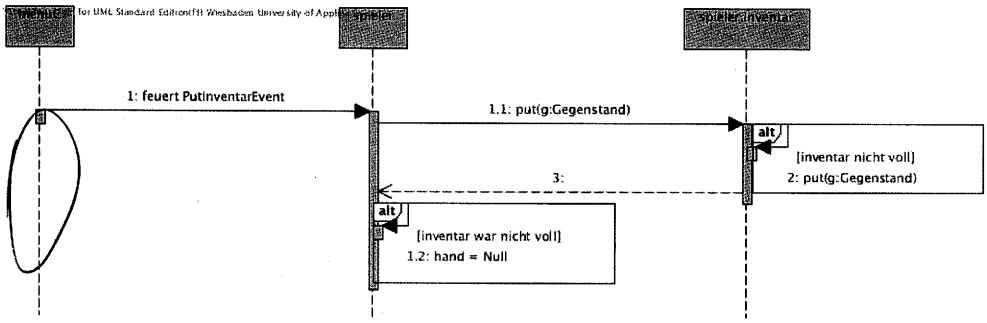


Abbildung 3.4: Item ins Inventar

3.2.5 Neues Spiel

Der Spieler befindet sich nach dem Spielstart im Hauptmenü. Hier steht ihm die Option Neues Spiel zur Verfügung. Startet der Spieler ein neues Spiel ruft dies die *neuesSpiel()*-Methode in *GameC* auf. Diese startet die *getGameInstance()*-Methode in der *GameFactory*. Dadurch werden der *MapBuilder*, *DialogBuilder* und der *GameObjectBuilder* ausgeführt, welche ein spielbares *Ingame* (Spiel) initialisieren. Der *GameC* prüft darauf hin ob ein *Ingame*-Objekt vorhanden ist und setzt das Attribut *ingame*(Boolean) entsprechend auf "true". Ist der Boolean *ingame* auf true, wird ein Event gefeuert das die *drawIngame()*-Methode der *Fenster*-Klasse innerhalb der Darstellungsebene aufruft. Durch *drawIngame()* wird dann die Ansicht des Spiels gezeichnet. Zudem startet der *GameC* das Spiel(Gameloop).

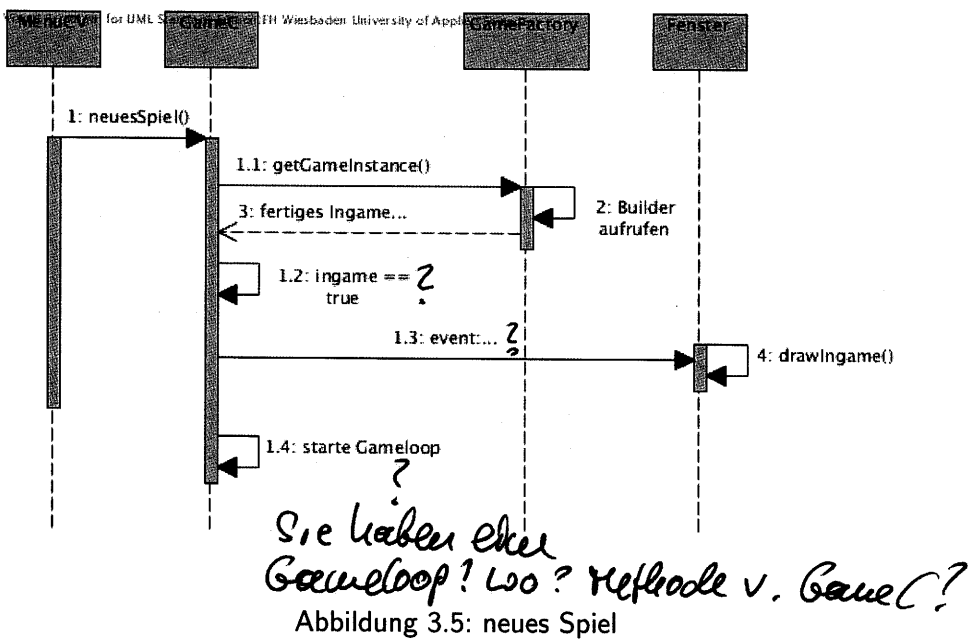
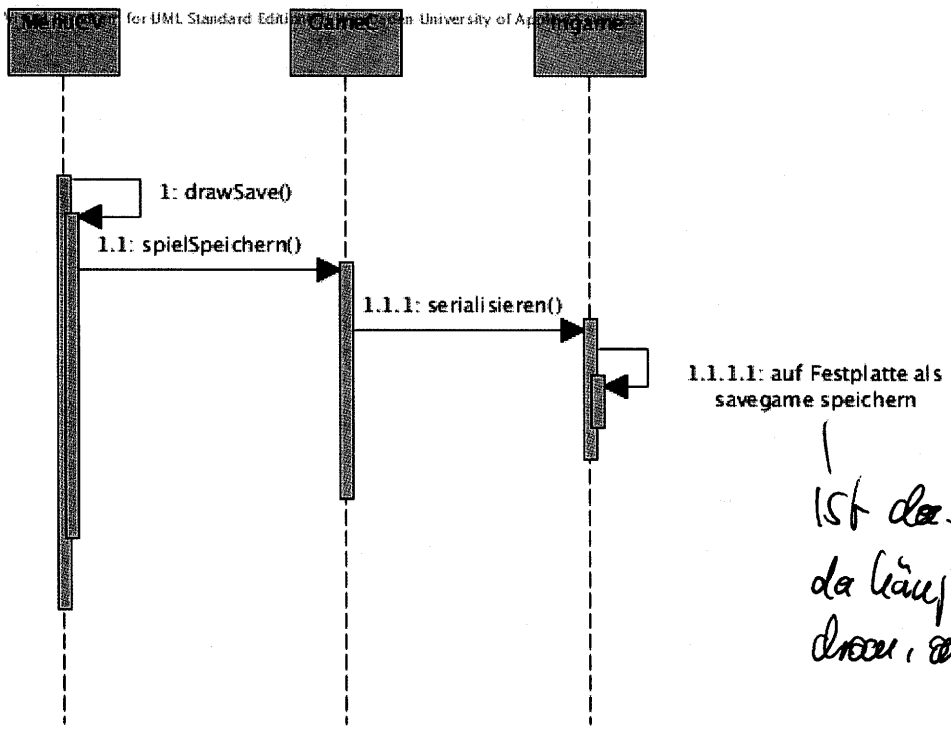


Abbildung 3.5: neues Spiel

3.2.6 Spiel speichern

Möchte der Spieler seinen aktuellen Spielstand speichern, gelangt er über die entsprechende Taste in das Pausemenü. Hier findet er den Spiel speichern-Button. Der Spiel speichern-Button löst nach dem betätigen die *spielSpeichern()*-Methode des *GameC* aus. Da dieser das Objekt *Ingame* besitzt, kann er die *serialisieren()*-Methode der *Ingame*-Klasse aufrufen. Diese speichert alle aktuellen Spieldaten als Savegame auf der Festplatte.



Ist das so okay? Da kommt vielleicht noch etwas, als nur Will...

Abbildung 3.6: Spiel speichern

Wie kommt der GameC denn an das "Ingame" (Code im KD nachvollziehen)?

3.2.7 Spiel laden

Über das Pausemenü, während einer Spielunterbrechung, oder das Hauptmenü nach dem Spielstart kann der Spieler einen Spielstand laden. Wenn ein Spiel geladen werden soll, wird zunächst über die *drawLoad()*-methode des *MenuCV* das entsprechende Fenster zur Auswahl eines Savegames gezeichnet. Wurde ein Spielstand gewählt, wird die *spielLaden()*-Methode zusammen mit dem Savegame als Parameter im *GameC* aufgerufen. Der *GameC* startet die *getGameInstanceFromSavegame()*-methode der *GameFactory*, welche das übergebene Savegame deserialisiert und ein fertiges *Ingame*-Objekt erstellt. Der *GameC* prüft daraufhin ob ein solches spielbares *Ingame*-Objekt vorhanden ist und gibt dann den Boolean *ingame*(true) zurück. Ist *ingame* auf true gesetzt feuert der *PropertyChangeListener* ein Event an die *Fenster*-Klasse in der Darstellungsebene. Diese zeichnet über die *drawIngame*-Methode das sichtbare Spiel anhand der des *Ingame*-Objekts, dass aus den Savegamedaten entstanden ist. Zudem startet der *GameC* das Spiel.

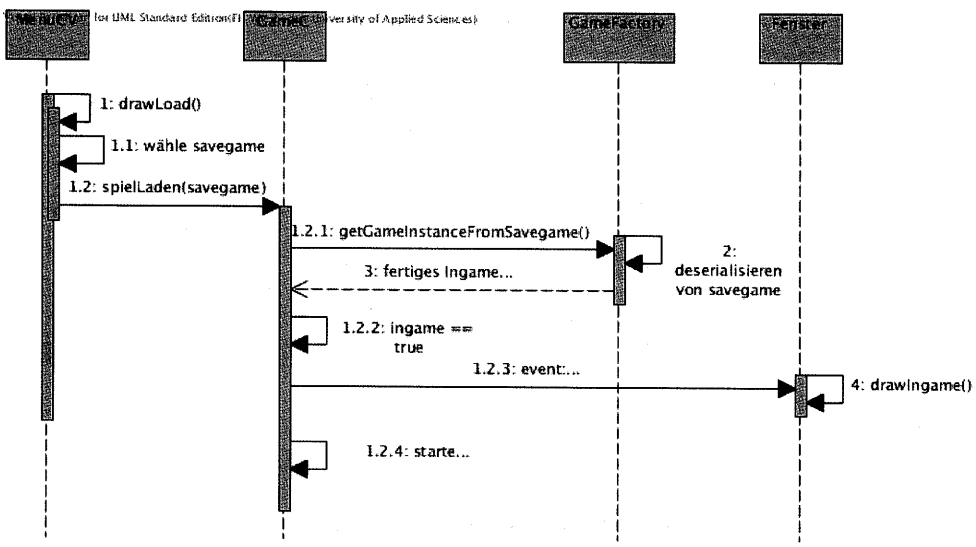


Abbildung 3.7: Spiel laden

4 Muster und Konzepte

4.0.8 Listener-Konzept

Die Anwendung verwendet *PropertyChangeListener* und *PropertyChangeSupport*. Erstere werden von solchen Klassen verwendet, deren Objekte auf Ereignisse aus anderen Bereichen des Spiels warten. Sie melden sich als Listener für bestimmte Attribute und Funktionen an. Ändern sich diese, erfahren die Listener das und können im eigenen Objekt für Methodenaufrufe sorgen.

Für das Erfahren dieser Änderungen ist der *PropertyChangeSupport* zuständig. Aus der Klasse heraus, in der er sich befindet, *feuert* er *PropertyChangeEvents*, die an die zugehörigen *PropertyChangeListener* adressiert sind.

4.0.9 MVC

Für die Darstellung der Anwendung ist die Klasse *Fenster* verantwortlich, unabhängig davon in welchem Modus - Menü oder Ingame - sich das Spiel im Momemt befindet. Sie setzt sich wiederum aus verschiedenen Klassen, den Views, zusammen. Diese vereinen in sich jeweils die Informationen, die für die Darstellung der entsprechenden View von Nöten sind.

Die *SpielerV* ist für die Anzeige des Statusbereichs verantwortlich. Ihre Informationen kommen zum einen vom *Spieler* selbst und andererseits von *NPC*s, *Objekt*en oder *Feld*ern mit denen aktuell interagiert wird. *Spieler* feuert Events, die den View anweist, die benötigten Informationen aus den entsprechenden Models zu lesen.

Beispiele für die Darstellung durch die *SpielerV* sind Dialoge und das Aufnehmen von Items von einem *Feld* in die *Hand* von *Spieler*.

Bei der Klasse *Sichtbereich* handelt es sich um die Spielwelt aus der Ego-Perspektive des Spielers. Der gesamte *Sichtbereich* setzt sich aus 32 *JPanel*s zusammen, durch die sichergestellt wird, dass jede in der Anwendung vorgesehene Kombination von Texturen darstellbar ist.

Es lassen sich zwei Unterbereiche abgrenzen. Das *Feld* auf dem sich der Spieler aktuell befindet und die beiden in Blickrichtung vor ihm liegenden Felder werden samt darauf befindlicher Gegenstände und NPCs dargestellt. Alle anderen Felder und zugehörige Wände werden ausschließlich als Texturen angezeigt.

Unabhängig davon werden alle Felder einzeln durch eine separate *FeldV* erstellt. Die Informationen hierzu kommen wiederum von den Models der einzelnen Felder(*FeldM*).

Die letzte verbleibende View ist die *MenuCV*, die gleichzeitig als Controller fungiert. Da das Menü nur eine grafische Oberfläche ohne abgrenzbare Logik ist, ist diese Zusammenführung sinnvoll. Das Menü wird vor dem Start eines neuen oder Laden eines bereits begonennen Spiels als Hauptmenü angezeigt. Es dient gleichzeitig als Pausemenü, sollte das Spiel zwischendurch unterbrochen werden. In diesem Fall behalten die oben genannten Views ihre aktuellen Informationen, werden aber nicht angezeigt während das Menü aktiv ist.

Die einzelnen Views erben jeweils von der Klasse *JPanel*. Die Klasse *Fenster*, die, wie oben erklärt die Views zur gesamten Anzeige zusammensetzt, erbt von der Klasse *JFrame*. Die Views und das *Fenster* implementieren Funktionen des o.g. Listener-Konzeptes, um ihre Darstellungs-Informationen von den entsprechenden Models zu erhalten.

Glossar

Anzeige Der gesamte, darzustellende Bereich der Anwendung. Jeweils entweder durch den Menümodus oder den *Aktivitätsbereich* ausgefüllt.. 9, 24

Blickrichtung Bezeichnet die Richtung des Spielers, in die er gerade schaut. Der Spieler kann sich drehen und die Blickrichtung dadurch ändern. 10, 14, 23

Campus Campus im Bezug auf das CampusAdventure. Synonym und Welt. 13

Charakter Das Alter Ego (seltener: Avatar) des Spielers *Ingame*. 13, 15

Dialog Meist eine Frage oder ein einfacher Satz mit (mehreren) möglichen Antworten, welche auf einen weiteren Dialog verweisen oder das Ende der Unterhaltung markieren. 11, 13, 15, 17, 23

Eingang Ein möglicher Zugang zu einem *Raum*. Eingänge können verschlossen sein. 10, 17, 18

Erwiderung Eine einfache Aussage oder Frage als Reaktion auf die eine NPC. 17

Feld Felder sind räumliche (ebene) Einheiten quadratischer Größe. Felder können sowohl Items als auch NPCs oder it Quests enthalten. Der Spieler bewegt sich von Feld zu Feld. 10, 13–17, 23, 24

GameObject Elemente im Spiel, z.B. Charaktere, Gegenstände oder Scheine . 10, 11, 13

Gegenstand Ein virtueller Gegenstand, der wenigstens eine Form der Interaktion mit dem Spieler zulässt. 10–13, 15, 17–19, 23

Handbereich Einer der vier Aktivitätsbereiche des Spiels *Ingame*. Hier wird entweder eine leere Hand oder das momentan in der Hand gehaltene Objekt dargestellt. 10, 11

Infobereich Einer der vier Aktivitätsbereiche des Spiels Ingame. Hier wird das Inventar mit seinen Slots oder textuelle Information bei Interaktionen dargestellt. 10, 11

Inventar Eine Art virtueller Rucksack mit beschränkter Kapazität, in dem z.B. *Items* aufgenommen werden können. 10, 11, 13, 15

Item Ein Gegenstand, der vom Spieler in sein Inventar aufgenommen werden kann. 10, 11, 23

Menümodus Der Zustand der Applikation in dem nicht gespielt werden kann. Man kann aber in den Ingamemodus wechseln. 9

NPC Non-Player-Character. 10, 11, 17, 18

NPCs Non-Player-Characters. 13, 15, 17, 23

Objekt Elemente im Spiel, mit denen der Spieler interagieren kann, die er jedoch nicht in sein Inventar aufnehmen kann. 13

Raum Ein abgeschlossenes Areal innerhalb der Map. Räume müssen dem Spieler nicht zugänglich sein. Räume bestehen aus *Feldern*. 13, 17

Sichtbereich Einer der vier Aktivitätsbereiche des Spiels Ingame. Hier wird die virtuelle Szene in einer Pseudo-3D-Perspektive dargestellt. 9, 10, 14, 16

Spiel Die Interaktion mit der Anwendung nach dem Start eines neuen Spiels und dem Erreichen des *Spielziel*s. 8, 13, 23, 24

Spieler Der Spieler in Form der Spielfigur, welche vom Anwender kontrolliert wird. 9–18, 23

Spielwelt Die Gesamtheit der virtuellen Umgebung bestehend aus dem Campus, Räumen, Items, NPCs, Quests und Spieler. 10, 13, 23

Statusbereich Einer der vier Aktivitätsbereiche des Spiels Ingame. Hier wird ein Portät des Charakters dargestellt. 9, 11, 23