Universität Hamburg
Department Informatik
Students, SSE

Ein KUULer Praktikumsbericht

Praktikumsbericht

Sofrwareentwicklungspraktikum
Agile Softwareentwicklung

Louis Kobras, Utz Pöhlmann
Matr.Nr. 6658699, 6663579
4kobras@informatik.uni-hamburg.de, 4poehlma@informatik.uni-hamburg.de

24. September 2015

Inhaltsverzeichnis

1 Einleitung

Dieser Bericht befasst sich mit dem Praktikum Agile Softwareentwicklung[1] (im Folgenden "ASE") als Teil des Moduls Softwareentwicklungspraktikum[2] (im Folgenden "SEP"), absolviert im zweiten Fachsemester (im Folgenden "FS2") des Studiengangs Software-System-Entwicklung (im Folgenden "SSE").
Dieser Bericht ist geschrieben aus Sicht der Teilnehmer des Teilkurses 3 (ASE-SEP3), in welchem das Zuul-Ausgangssystem, welches von den Praktikumsbetreuern bereit gestellt wurde, in ein *Point& Click*-ähnliches *Adventure-Game* weiterentwickelt wurde, welches nun den Titel *KUUL* trägt.
In diesem Bericht wird tiefergehend auf die Implementation von interaktiven Objekten in das *KUUL*-System eingegangen. Dies umfasst die Erstellung der Gegenstände, deren Speicherung und die Implementation der Interaktion.

2 Hauptteil

Wie bereits in der Einleitung angesprochen, wird sich dieser Bericht primär mit dem Umgang mit den Gegenständen in *KUUL* befassen. Interaktiv sind diese Gegenstände insofern, als dass sie aufgesammelt, benutzt und kombiniert werden können.
Zunächst wird jedoch festgestellt werden, worum es sich bei *KUUL* handelt.

2.1 KUUL

KUUL ist ein Ausbau des von den Praktikumsbetreuern bereitgestellten Zuul-Ausgangssystems. Während Zuul ein konsolenbasiertes Text-Adventure ist, wird *KUUL* mit Maus und Tastatur gesteuert. Dies geschieht insofern, als dass Gegenstände, *NPCs*[3] und Türen angeklickt werden können, um mit ihnen zu interagieren. Mithilfe der Tastatur kann der *PC*[4] durch die Räume in der Welt von *KUUL* bewegt werden, ebenso lassen sich alle Funktionen des Spiels über bestimmte Tasten direkt ansteuern.
Zuul hatte bereits einige wenige Räume vorzuweisen, die durch die Namensgebung erkennen ließen, dass es sich um eine an ein Universitätsgelände angelehnte Welt handelte. Dies wurde vom Entwicklerteam aufgegriffen, sodass, während die Anzahl der Räume mehr als verdoppelt wurde, die Umgebung *Universität* beibehalten wurde. Auf Wunsch des Kunden hin wurde im Lauf des Praktikums eine weitere Welt hinzugefügt, die sich grob am Hamburger Kiez (im Folgenden "Kiez") orientiert.

[1] STiNE-Veranstaltungsnummer 64-142
[2] STiNE-Modulnummer SSE_PR
[3] *Non-Player-Character*, eine Figur im Spiel, die nicht vom Spieler, sondern vom Spiel kontrolliert wird.
[4] *Player-Character*, vgl. *NPC*

2.2 Entwicklungspozess

Da zunächst nur die Textwelt von Zuul zur Verfügung stand, wurden die ersten Gegenstände inklusive des provisorischen Gewinngegenstandes[1] *Handy* in den Räumen platziert, die nach der Erweiterung vorhanden waren. Auf Kundenwunsch wurde das Handy als Sieggegenstand recht schnell von der *Goldananas* abgelöst.

```
1 if (gegenstandElement.getElementsByTagName("beenden").
    getLength() != 0)
2 {
3     Node beendenNode = gegenstandElement.
          getElementsByTagName(
4               "beenden").item(0);
5     String nachricht = beendenNode.getTextContent();
6     _beendenCommandListe.put(id, nachricht);
7 }
```

Code 1: Auszug aus dem XML-Parser für Gegenstände

Der hier betrachtete Quellcode ist aus dem in *KUUL* verwendeten XML-Parser entnommen.
Die XML[2]-Sprache arbeitet insofern, als dass jedes Dokument eine Reihe von Elementen hat, und diese Elemente haben Attribute (im Folgenden "Tags"), welche wiederum Element-Reihen seien können. Im Gegensatz zu HTML[3] ist XML erweiterbar, das heißt es können benutzerdefinierte Tags hinzugefügt werden; die Syntax der beiden Sprachen ist nahezu gleich.
An dieser Stelle wird der Tag des Gegenstands mit der Bezeichnung "beenden" abgefragt: Ist sie vorhanden (Länge ungleich 0), so handelt es sich um einen Sieggegenstand. Anzumerken ist hierbei, dass der verwendete Parser abgefragte Tags immer als eine Form von Liste zurückgibt, weswegen der Tag mit `.item(0);` abgefragt werden muss.
Die Siegnachricht, welche ebenfalls extern in der XML-Datei gelagert wird, wird ausgelesen und als String gespeichert. Der Gegenstand wird als Tupel aus ID und der Siegnachricht in eine Liste gelegt, die an anderen Stellen im Programm aufgerufen werden kann, um festzustellen, ob der gerade verwendete Gegenstand ein Sieggegenstand war.
Wird der Sieggegenstand nun innerhalb des Spiels gefunden, aufgehoben und benutzt, so wird das Spiel beendet und es erscheint ein Popup, welches einen darüber informiert, dass man gewonnen hat, und den String `nachricht` (vgl. Code 1 l.5) anzeigt. Die beiden Welten haben unterschiedliche Sieggegenstände, sodass es auf dem Kiez erforderlich ist, das eigene Portmonnaie wiederzufinden.

[1] In *KUUL*, ist es auf zwei Arten möglich, zu gewinnen: Entweder findet der Spieler einen Schlüsselgegenstand, mit dem er einen Raum betreten kann, der als *Siegraum* betitelt wurde, oder er findet einen *Sieggegenstand*

[2] *eXtensible Markup Language*, findet Verwendung in der Strukturierung von Daten

[3] *HyperText Markup Language*, findet Gebrauch bei der Gestaltung von Web-Oberflächen

2.3 Funktionsumfang

Wie bereits mehrfach angesprochen, können Gegenstände aufgehoben und genutzt werden.
In *Zuul* funktionierte dies über die Konsolenbefehle. Um *KUUL* komfortabler zu machen, sind Gegenstände im Raum anklickbar. Ebenso können sie aufgehoben werden, wenn sich der Spieler direkt davor stellt und die Enter-Taste drückt.
Aufgenommene Gegenstände kommen in den *Rucksack*, welcher allerdings nur eine beschränkte Kapazität von vier beliebigen Gegenständen hat. Gegenstände, die man im Rucksack hat, können kombiniert werden. Allerdings müssen sie dazu in der XML-Datei den Tag "kombinierbar", welcher im Parser als `boolean` ausgelesen wird, gesetzt bekommen. Ist dieser Tag auf den Wert `"false"` gesetzt, so können die Gegenstände nicht kombiniert werden.
Eine weitere Besonderheit des Rucksack ist es, dass die meisten Gegenstände in der ersten Welt, die einem normalerweise Energie zurückgeben[1], mit der Zeit verfallen. Dies führt dazu, dass sie irgendwann nicht nur keine Energie zurückgeben, sondern Energie abziehen.
Illustriert wird dies erstens durch eine Veränderung des Aussehens des Gegenstandes und zweitens durch eine Änderung des Textes, der bei Nutzung des Gegenstandes angezeigt wird[2]. Graphisch hat jeder Gegenstand höchstens drei Phasen[3], der Abzug der Energie-Einheiten wächst allerdings stetig linear. Alle zwei Schritte zieht der Gebrauch eines solchen Gegenstandes eine Einheit mehr ab.

```
1  if (_verfallszeitpunkt > 1) {
2      _verfallszeitpunkt = 0;
3      lebensCommand.verringereLebenspunkte();
4      if (lebensCommand.getLebensEffekt() == 0) {
5          setIcon("Graphics/Items/Phase2" + "/" + gibName() +
               "2.png");
6          _verfallStatus = 1;
7          return true;
8      } else if (lebensCommand.getLebensEffekt() < 0) {
9          setIcon("Graphics/Items/Phase3" + "/" + gibName() +
               "3.png");
10         _verfallStatus = 2;
11         return true;
12     }
13 } else {
14     _verfallszeitpunkt++;
15 }
```

Code 2: Teilimplementation der Verringerung der Energieregeneration

[1]Energie wird benötigt, um sich zu bewegen. Der Spieler startet mit 15 Energie-Einheiten. Jeder Raumwechsel verbraucht eine Einheit.

[2]So wird z.B. *"Die Ananas stärkt Sie und gibt Ihnen Kraft. Sie erhalten 2 Lebenspunkte. Toll!"* zu *"Die Ananas ist bis auf den Kern vergammelt. Sie verlieren Lebenspunkte!"*, wenn man die Ananas lange genug im Rucksack behält

[3]Viele Gegenstände, wie zum Beispiel Schlüssel, haben nur eine Phase

Bei dem Feld `_verfallszeitpunkt`[1] handelt es sich um eine Exemplarvariable vom Typ Integer. Immer dann, wenn der hier gezeigte Block aufgerufen wird, wird diese Variable geprüft. Ist sie auf 0, wird sie inkrementiert; ist sie auf 1, so wird der Effekt des Lebenskommandos[2] dekrementiert und der `_verfallszeitpunkt` wird wieder auf 0 gesetzt. Was ebenfalls zu sehen ist, ist, dass ein neues Icon für den Gegenstand gesetzt wird, wenn er 0 Energie-Einheiten erreicht beziehungsweise sobald er anfängt, Energie abzuziehen (siehe den inneren if-Block).

Zur bereits angesprochenen Kombinationsfunktion sei gesagt, dass nicht alle Gegenstände kombinierbar sind. Es wurde bereits gesagt, dass ein Tag gesetzt werden muss, damit ein Gegenstand vom *Crafting Tool* (vgl. Abb. 1) akzeptiert wird. Desweiteren muss es ein Rezept geben, welches den fraglichen Gegenstand beinhält. Befindet sich dann der zweite Gegenstand ebenfalls im Rucksack des Spielers, so kann der Spieler die Kombination ausführen.
Da sich die zweite Welt an den Kiez anlehnt, befinden sich dort primär alkoholische Getränke. Alle alkoholischen Getränke können miteinander kombiniert werden; allerdings ist dort die Regelung ein wenig anders: Jedes Rezept hat als Ergebnis eine *Gute Mische*, welche Energie dazu gibt, jedes ungültige Rezept hat als Ergebnis eine *Schlechte Mische*, welche Energie abzieht. Die hier betrachtete Konfigurati-

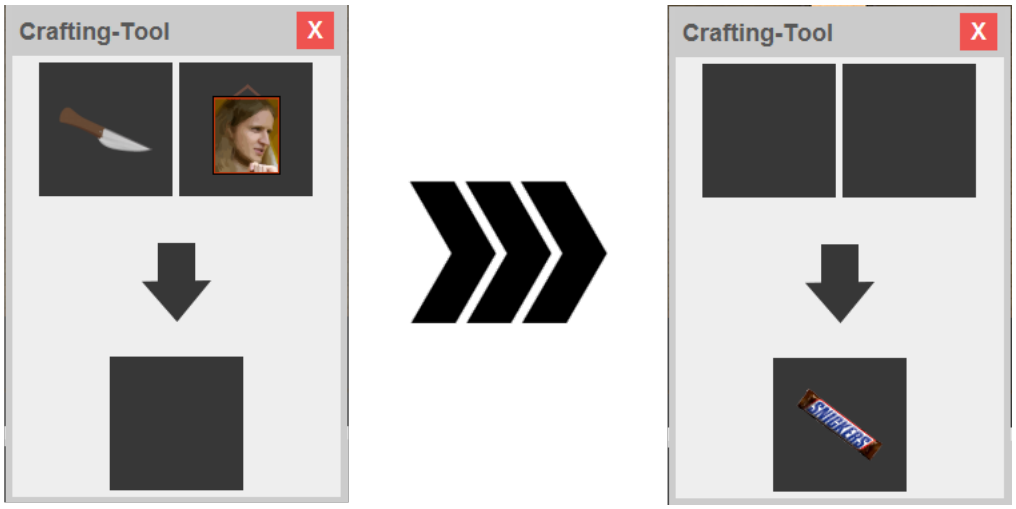

Abbildung 1: Crafting Tool

on des *Crafting Tools* entstammt der ersten Welt. Das Messer und das Gemälde werden kombiniert, um ein Snickers zu erhalten, welches die Energie vollständig wieder auffüllt. Messer und Gemälde werden dabei verbraucht.

[1] Nomenklaturregeln nach SE2-Format

[2] interne Umsetzung der Energieverwaltung, für die Verwendung der Gegenstände nicht weiter von Bedeutung

2.4 Commands

Allgemein bekommt ein jeder Gegenstand mindestens ein X-Command[1] zugeordnet. Diese Commands werden alle beim Einlesen aus den XML-Dateien in Listen gespeichert und dann den Gegenständen hinzugefügt. So hält jeder Gegenstand seine Liste an Commands, die abgefragt werden können, wenn mit dem Gegenstand interagiert wird[2].
Die Gegenstände sollten im Laufe des Praktikums immer mehr verschiedene Funktionen erhalten. Dies war insofern schwierig, als dass die zu implementierenden Funktionen immer komplexer wurden[3] und häufig auch ein Gegenstand eine ganz andere Sache können sollte, als bisher, ohne jedoch die alte(n) Funktion(en) einzubüßen.
Dafür wurde sich des Command-Befehlsmusters bedient und eine neue Command-Methode erstellt:

```
1 public interface Command {
2     public void apply();
3 }
```

Code 3: Unser Command-Interface

Diese tut nichts anderes, als als Interface zu dienen, auf welches andere Methoden zugreifen dürfen, ohne zu wissen, was nun wirklich genau im Code passiert. Dieses kann nun nämlich in den speziellen Commands festgelegt werden.
So sieht z.B. das Lebenscommand, welches natärlich das Interface Command implementiert, so aus:

```
1  public LebenCommand(Spieler spieler, int hp)
2      {
3          _hp = hp;
4          _spieler = spieler;
5      }
6
7      @Override
8      public void apply()
9      {
10         _spieler.getLebenspunkte()
11             .veraendereLebenspunkte(_hp);
12     }
```

Code 4: LebenCommand - eine implementation des Command-Interfaces

In diesem wird die `apply`-Methode überschrieben und tut jetzt etwas sinnvolles, wie in diesem Fall, das Leben des Spielers verändern. So kann jedes Mal, wenn

[1] z.B. Lebenscommand zur Erhöhung der Energie, oder Beendencommand zum Beenden des Spiels

[2] Damit nichts passiert, wenn der Gegenstand sich im selben Raum befindet, aber nicht benutzt wird.

[3] So gibt es z.B. eine Zeitmaschine, mit der man exakt 5 Räume zurückreisen kann.

mit einem Gegenstand interagiert wird, die Methode `benutzeGegenstand` aus der Klasse `Gegenstand` aufgerufen werden. Und dieser ist es egal, um welche Art von Command es sich letztendlich handelt.

```
1     public void benutzeGegenstand() {
2         if (!_commandList.isEmpty()) {
3             ArrayList<Command> iterierListe = _commandList;
4             for (Command command : iterierListe) {
5                 command.apply();
6             }
7         }
8     }
```

Code 5: die benutzeGegenstand-Methode aus der Klasse Gegenstand

Sie geht alle Commands durch, die in dem Gegenstand gespeichert sind, und führt sie nacheinander aus[1], ohne dass ihre konkrete Implementation, also die Funktion, die der Gegenstand hat, von Bedeutung für die Funktionalität dieser Methode ist. Somit lässt sich ganz einfach eine neue Funktion zu einem Gegenstand hinzufügen, indem einfach bei der Erstellung ein weiteres Command der Liste hinzugefügt wird, welches nun beim Benutzen des Gegenstandes abgearbeitet wird.

3 Schluss

[1]Die `iterierListe` ist nur dazu da, um Fehlern vorzubeugen, die entstehen würden, wenn ein `Command` in seiner `apply`-Methode die Liste verändert. So z.B. die vergammelbaren Gegenstände.