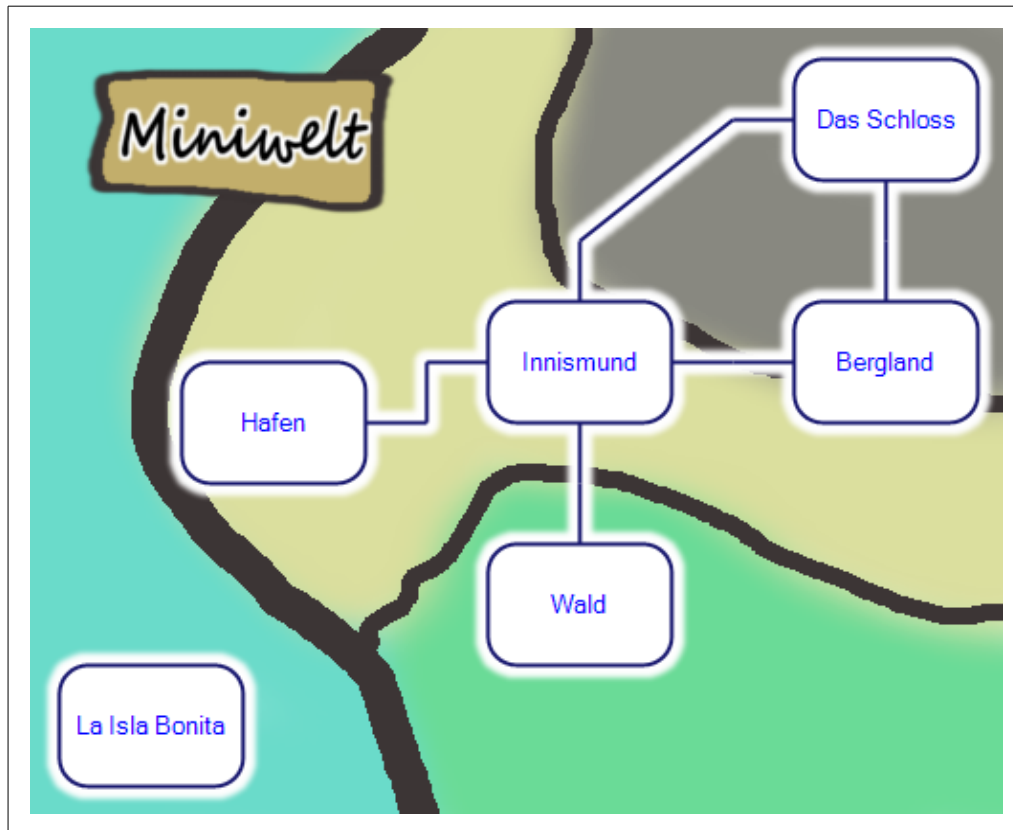


Modulares Geschichtenerzählen mit endlichen Automaten



ILTB 2017 Passau

Thomas Rau

LMU München

Graf-Rasso-Gymnasium

Fürstenfeldbruck

thomas.rau@ifi.lmu.de

Inhaltsverzeichnis

Arbeitsblatt 1a: Die kleinste mögliche Geschichte.....	3
Arbeitsblatt 1b: Bilder und Landkarte.....	4
Arbeitsblatt 2a: Eine Geschichte mit Optionen.....	5
Arbeitsblatt 2b: Optionen, bei denen etwas geschieht.....	6
Arbeitsblatt 3: Umsetzen eines Zustandsautomaten.....	7
Arbeitsblatt 4: Geschichten, die an mehreren Orten spielen.....	9
Zustandsdiagramm Froschkönig, linear.....	11
Zustandsdiagramm Froschkönig, nicht linear.....	11
Anhang 1a: Zentrale Methoden der Geschichten.....	12
Anhang 1b: Mögliche ausgelöste Aktionen.....	13
Anhang 1c: Mögliche Bedingungen.....	14
Anhang 2: Wie man eine Welt erzeugt.....	15
Anhang 3: Wie man eine Location anlegt.....	15
Anhang 4: Was in GameBuilder (oder Unterklassen von TestBuilder) eingestellt werden kann.....	16
Anhang 5: Das automatische Starten von Geschichten.....	17
Anhang 6: Beispiele für ganz einfache Geschichten.....	18
Anhang 7: Das Zusammenspiel von createReport() und receiveMessage(String) im Diagramm....	21
Anhang 8: Erweiterungen.....	22
Anhang 9: Zur Programmiertechnik.....	23
Anhang 10: Gute Geschichten.....	24

Arbeitsblatt 1a: Die kleinste mögliche Geschichte

AUFGABE 1: Ergänzen Sie im BlueJ-Projekt auf der obersten Ebene die folgende Klasse und testen Sie sie durch Anlegen eines Objekts dieser Klasse. Nennen Sie die Klasse aber anders!

```
1  import storyworld.*;
2  public class Arbeitsblatt1 extends SimpleStory {
3      public Arbeitsblatt1() { test(); }
4      public String getName() { return "Ein Minibeispiel."; }
5      public String startStoryAt() { return "Hafen"; }
6      public Report createReport() {
7          Report r = new Report("Hier gibt es nichts zu sehen.");
8          return r;
9      }
10 }
```

Zeile 1: Importiert nötiges package.

Zeile 2: Für den Anfang erbt jede Geschichte von SimpleStory.

Zeile 3: Am Ende des Konstruktors wird die Methode `test()` aufgerufen, die für das Spiel nicht nötig ist, für das automatische Starten zum Testen allerdings schon.

Zeile 4: Jede Geschichte braucht einen Namen. Diese (notwendige) Methode gibt den Namen der Geschichte zurück.

Zeile 5: Jede SimpleStory braucht einen Ort, an dem sie spielt. Diese (notwendige) Methode gibt den Namen des Orts zurück. Erlaubt sind in der Miniwelt nur: "Hafen", "Innismund", "Wald", "Bergland", "Das Schloss" und – erst einmal vermeiden – "La Isla Bonita".

Zeile 6: Immer wenn der Spieler sich am Schauplatz der Geschichte befindet, wird diese (notwendige) Methode aufgerufen und ein Bericht der Geschichte eingefordert. Der kann auch null sein, aber dann sieht der Spieler nichts. Besser ist es, ein Objekt der Klasse Report zurückzugeben. Der einfachste Konstruktor dieser Klasse verlangt lediglich einen String als Argument.

AUFGABE 2: Ändern Sie auch die Rückgabewerte der Methoden aus Zeile 4, 5 und 6. Machen Sie aus der Geschichte einen alten Mann, eine Ruine, eine Haus, eine Frau, ein Tier, ein Schiff, einen Turm.

Optional: Ergänzen Sie die folgende Methode, damit es ein Bild zu Ihrer Geschichte gibt (eine Liste der bereits im Projekt vorhandenen Bilder ist auf der Rückseite, da können Sie wählen):

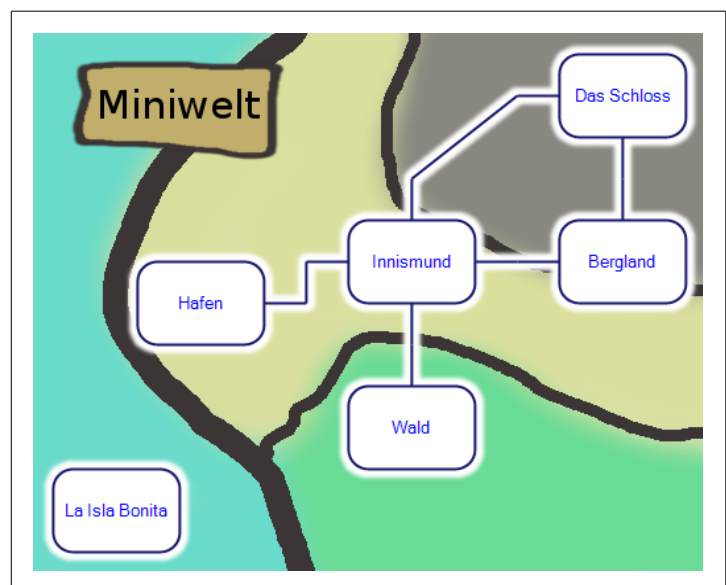
```
public String getImageName() { return "resources/zirkus.png"; }
```

Arbeitsblatt 1b: Bilder und Landkarte

Im Hauptverzeichnis StoryWorld-BlueJ befinden sich diese Bilddateien, alle im Ordner “resources”, der jeweils mit angegeben werden muss. Die meisten Dateien sind im .png-Format, es sind aber auch einige wenige .jpg von Schülern dabei.

resources/angler.png
resources/berg.png
resources/berge.png
resources/berge2.png
resources/bergwald.png
resources/bergwerk.png
resources/bibliothek.jpg
resources/boot.jpg
resources/bote.png
resources/bote2.png
resources/dame.png
resources/dame2.png
resources/einkaufskorb.png
resources/einkaufskorb2.png
resources/festung.png
resources/festung2.png
resources/festung3.png
resources/fliegenpilze.png
resources/friedhof.png
resources/gasthaus.png
resources/grenze.jpg
resources/hafen.png
resources/hafen2.png
resources/hafen3.png
resources/herrenhaus.jpg
resources/haus.jpg
resources/haus2.jpg
resources/haus3.jpg
resources/haus4.jpg
resources/haus5.jpg
resources/hoehle.jpg
resources/insel.png
resources/insel2.png
resources/junge.png
resources/kamel.png
resources/kaufhaus.png
resources/kirche.png
resources/kirche2.png
resources/kiste.png
resources/laden.png
resources/maedchen.png
resources/mann_kutte.png
resources/mann.png
resources/oase.jpg
resources/rathaus.jpg
resources/ruinen.png
resources/ruinen2.png
resources/scheune.png
resources/schiff.jpg
resources/schiff.png
resources/schloss.png
resources/sonnenuntergang

resources/stadt.png
resources/stadt2.jpg
resources/turm.png
resources/wald.png
resources/waldweg.png
resources/wanderer.png
resources/wegweiser.png
resources/zauberer.png
resources/zelt.png
resources/zirkus.png



Arbeitsblatt 2a: Eine Geschichte mit Optionen

AUFGABE 3: Ergänzen Sie in Ihrer Klasse (die jetzt nicht mehr “Arbeitsblatt” heißen sollte) den zurückgegebenen Report um die fett gedruckten Zeilen mit Optionen und testen Sie Ihre Klasse danach.

```
1  import storyworld.*;
2  public class Arbeitsblatt2a extends SimpleStory {
3      public Arbeitsblatt2a() { test(); }
4      public String getName() { return "Ein Angler."; }
5      public String startStoryAt() { return "Hafen"; }
6      public Report createReport() {
7          Report r = new Report("Ein Fischer sitzt am Kai und angelt.");
8          r.addOption("Du fragst ihn, wie es so läuft", "fragen");
9          r.addOption("Du schaust aufs Meer.", "schauen");
10         return r;
11     }
12 }
```

Zeile 8: Dem Report wird eine Option hinzugefügt. Eine Option wird dem Spieler als anklickbarer Knopf angezeigt. Die einfachste Methode, eine Option hinzuzufügen, geschieht mit der Report-Methode `addOption(String, String)` – dabei ist das erste String-Argument das, was der Spieler zu lesen bekommt, und das zweite String-Argument die Nachricht, die übergeben wird, wenn der Spieler genau diese Option wählt.

Zeile 9: Es können beliebig viele Optionen hinzugefügt werden. Der erste String sollte, der zweite muss sich unterscheiden von den anderen.

AUFGABE 4: Ändern Sie den Text oder die Anzahl der Optionen und schauen Sie sich das Ergebnis an.

Sie haben sicher schon gemerkt, dass jetzt zwar Knöpfe vorhanden sind, dass aber noch nichts geschieht, wenn man auf sie drückt. Wie Sie das ändern, steht auf der Rückseite dieses Blattes.

Arbeitsblatt 2b: Optionen, bei denen etwas geschieht

AUFGABE 5: Ergänzen Sie in Ihrer Klasse die neue, fett gedruckten Methode.

```
1  import storyworld.*;
2  public class Arbeitsblatt2b extends SimpleStory {
3      public Arbeitsblatt2b() { test(); }
4      public String getName() { return "Ein besserer Angler."; }
5      public String startStoryAt() { return "Hafen"; }
6      public Report createReport() {
7          Report r = new Report("Ein Fischer sitzt am Kai und angelt.");
8          r.addOption("Du fragst ihn, wie es so läuft", "fragen");
9          r.addOption("Du schaust aufs Meer.", "schauen");
10         return r;
11     }
12     public void receiveMessage(String nachricht) {
13         if (nachricht.equals("fragen")) {
14             increase("Neugier", 1);
15         }
16         if (nachricht.equals("schauen")) {
17             set("Innerer Frieden", 20);
18         }
19     }
20 }
```

Zeile 12: Die Methode `receiveMessage` wird aufgerufen, wenn der Spieler eine Option wählt, und übergeben wird dabei als Parameter eben dieser zweite String, der der Option im Konstruktor mitgegeben wurde.

Zeile 14: Wenn der Spieler die `fragen`-Option gewählt hat, wird eine Aktion ausgelöst: Mit `increase(String, int)` wird ein globaler Marker der Welt, der von allen Geschichten aus erreichbar ist, verändert. Hier wird die Neugier der Spielfigur jedesmal um 1 erhöht. Wenn es den Marker "Neugier" noch nicht gibt, wird er erzeugt; die aktuellen Marker kann sich der Spieler anzeigen lassen, wenn er auf die Spielfigur klickt.

Welche Aktionen noch von einer Geschichte ausgelöst werden können, steht in einem Anhang.

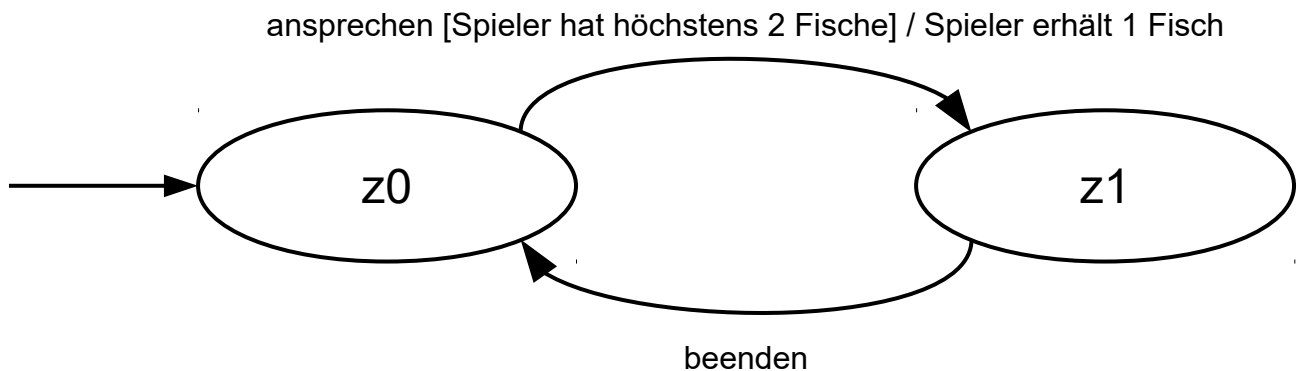
Natürlich kann man vor die ausgelöste Aktionen noch eine Bedingung setzen, die erfüllt sein muss, um die Aktion auszulösen, zum Beispiel: `if (get("Neugier")<10)`. Welche Methoden dabei hilfreich sind, steht ebenfalls in einem Anhang.

Zeile 17: Hier wird der Marker "Innerer Frieden" auf 10 gesetzt. Wenn es ihn noch nicht gibt, wird er erzeugt.

AUFGABE 6: Ändern Sie die ausgelösten Aktionen, so dass Sie mehr zu Ihrer Geschichte passen. Bauen Sie eine Bedingung ein, wenn Sie wollen.

Arbeitsblatt 3: Umsetzen eines Zustandsautomaten

Geschichten – also Gebäude, Personen, oder abstraktere Konstruktionen – haben häufig einen Zustand. Ein Angler mit zwei Zuständen könnte so modelliert werden. In eckigen Klammern (optional) eine Bedingung für den Zustandsübergang, nach dem Schrägstrich (optional) eine ausgelöste Aktion



Im Zustand z0 kann man den Angler ansprechen, sofern man höchstens 2 Fische hat (und erhält danach einen weiteren Fisch). Im Zustand z1 kann man das Gespräch beenden. In jedem Zustand wird unterschiedlicher Text gezeigt und werden unterschiedliche Optionen angeboten.

Implementiert wird diese rudimentäre Anglergeschichte so (Fortsetzung Rückseite):

```
1  import storyworld.*;
2  public class Arbeitsblatt3 extends SimpleStory {
3      int state = 0;
4      public Arbeitsblatt3 () { test(); }
5      public String startStoryAt() { return "Hafen"; }
6      public String getName() { return "Der beste Angler"; }
7      public String getImageName() { return "resources/angler.png"; }
8      public Report createReport() {
9          if (state == 0) {
10             Report r = new Report("Ein Fischer sitzt am Kai und angelt.");
11             r.addOption("Du fragst ihn, wie es so läuft.", "ansprechen");
12             return r;
13         } else {
14             Report r = new Report("Er erzählt dir eine lange Geschichte.");
15             r.addOption("Du dankst für Gespräch und Fisch.", "beenden");
16             return r;
17         }
18     }
19 }
```

```

19     public void receiveMessage(String s) {
20         if (state==0) {
21             if (s.equals("ansprechen") ) // ausloesende Aktion
22             {
23                 if (get("Fische")<3) { // Bedingung des Zustandsuebergangs
24                     state = 1;
25                     increase("Fische", 1); // ausgeloeeste Aktionen
26                 } else {
27                     sendMessage("Du hast schon genug Fische.");
28                 }
29             }
30         }
31         else if (state==1) {
32             if (s.equals("beenden")) {
33                 state = 0;
34             }
35         }
36     }
37 }

```

(Natürlich kann man die Zustandsübergangsmethode auch anders aufbauen.)

Zeile 3: Deklaration eines Zustands-Attributs.

Zeile 9/13: Abhängig vom Zustand wird entweder der eine Report zurückgegeben, oder der andere.

Zeile 20/31: Abhängig vom aktuellen Zustand...

Zeile 21/32: ...und der auslösenden Aktion...

Zeile 24/33: ...wird ein Zustandsübergang durchgeführt.

Zeile 23: Eine Bedingung für den Zustandsübergang. Spieltechnisch sollte man den Spieler vorher oder nachher über diese informieren, etwa durch sendMessage("Hat nicht geklappt, weil...")

Zeile 25: Eine ausgelöste Aktion: Erhöhen eines globalen Markers.

AUFGABE 7: Kopieren Sie den Code aus der Klasse Arbeitsblatt3 in eine neue Klasse und ändern Sie den Klassenbezeichner und den Konstruktor. Ändern Sie die Texte und die ausgelösten Aktionen, so dass z.B. ein Kind daraus wird, das einem immer wieder den gleichen Witz erzählt. (Danach, wer Lust hat: Per Zufallsgenerator immer wieder einen anderen Witz.)

Vorschlag: Machen Sie aus dem Fischer eine Fähre, die im "Hafen" beginnt und nach dem Zustandsübergang in "La Isla Bonita" ist – benutzen Sie dazu die ausgelösten Aktionen movePlayerTo(String) und moveStoryTo(String). Und dann auch wiederzurückfahren können?

Arbeitsblatt 4: Geschichten, die an mehreren Orten spielen

Eine Geschichte der Klasse `SimpleStory` spielt immer nur an einem Ort gleichzeitig. Mit der Definition der Methode `String startStoryAt()` wird dieser festgelegt, mit dem Aufruf der Methode `void moveStoryTo(String)` kann er geändert werden. Nur wenn der Spieler an diesem Ort ist, wird die Geschichte aufgefordert, einen Report abzugeben. Das erleichtert das Gestalten der Methode `Report createReport()`.

Wenn aber eine Geschichte von der Klasse `MultiplaceStory` erbt, wird die Methode `createReport()` jedesmal aufgerufen, wenn der Spieler an irgendeinen neuen Ort kommt. Es ist so, als spielte die Geschichte gleichzeitig überall.

Diese Methode muss dann also sinnvollerweise so angepasst werden, dass sie den zurückgegebenen Report vom aktuellen Schauplatz abhängig macht:

```
1  import storyworld.*;
2  public class Arbeitsblatt4 extends MultiplaceStory {
3      String imageName;
4      public Arbeitsblatt4() { test(); }
5      public String startStoryAt() { return "Das Schloss"; }
6      public String getName() { return "Romeo & Julia"; }
7      public Report createReport() {
8          if (playerIsIn("Das Schloss")) {
9              imageName = "resources/maedchen.png";
10             Report r = new Report("Hier sitzt Julia und wartet auf Romeo.");
11             return r;
12         }
13         else if (playerIsIn("Bergland")) {
14             imageName = "resources/junge.png";
15             Report r = new Report("Hier sitzt Romeo und wartet auf Julia.");
16             return r;
17         }
18         else return null;
19     }
20     public void receiveMessage(String s) { }
21     public String getImageName() { return imageName; }
22 }
```

Zeile 3: Reiner Schmuck: Das Name des aktuellen Bildes wird jetzt in einem Attribut gespeichert.

Zeile 8: Der folgende Report wird nur dann abgegeben, wenn der Spieler im Schloss ist.

Zeile 9: Schmuck: Wenn ein Report für den Ort “Das Schloss” angefragt wird, wird der Bild-Attributwert angepasst.

Zeile 13: Der folgende Report wird nur dann abgegeben, wenn der Spieler im Bergwald ist.

Zeile 14: Schmuck: Wenn ein Report für den Ort “Bergwald” angefragt wird, wird der Bild-Attributwert angepasst.

Zeile 18: In allen anderen Fällen wird kein Report zurückgegeben und der Spieler sieht nichts.

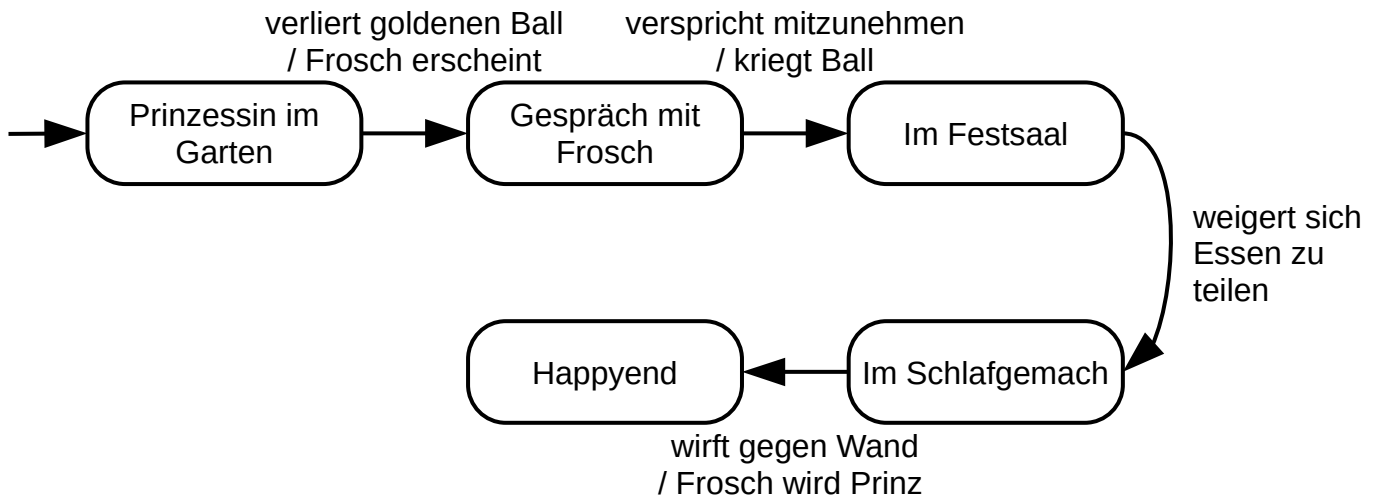
Zeile 21: Statt eines hardcodierten Bildnamens wird jetzt ein Attributwert zurückgegeben.

AUFGABE 8: Legen Sie eine Unterklasse von MultispaceStory an. Kopieren Sie dazu den Code aus der Klasse Arbeitsblatt4 hinein, damit sie nicht alles abschreiben müssen. Ändern Sie den Code – gerne erst einmal ohne Options – dann so, dass

- in Innismund Rotkäppchens Großmutter wartet
- im Wald der Wolf ist
- im Bergland Rotkäppchen
- und an allen anderen Ort ein null-Report zurückgegeben wird.

Wenn Sie wollen, können Sie Optionen hinzufügen, und ein Zustandsattribut, das den Wolf verschwinden lässt, sobald der Spieler einmal bei der Großmutter war. Natürlich können Sie sich auch eine bessere Geschichte ausdenken.

Zustandsdiagramm Froschkönig, linear

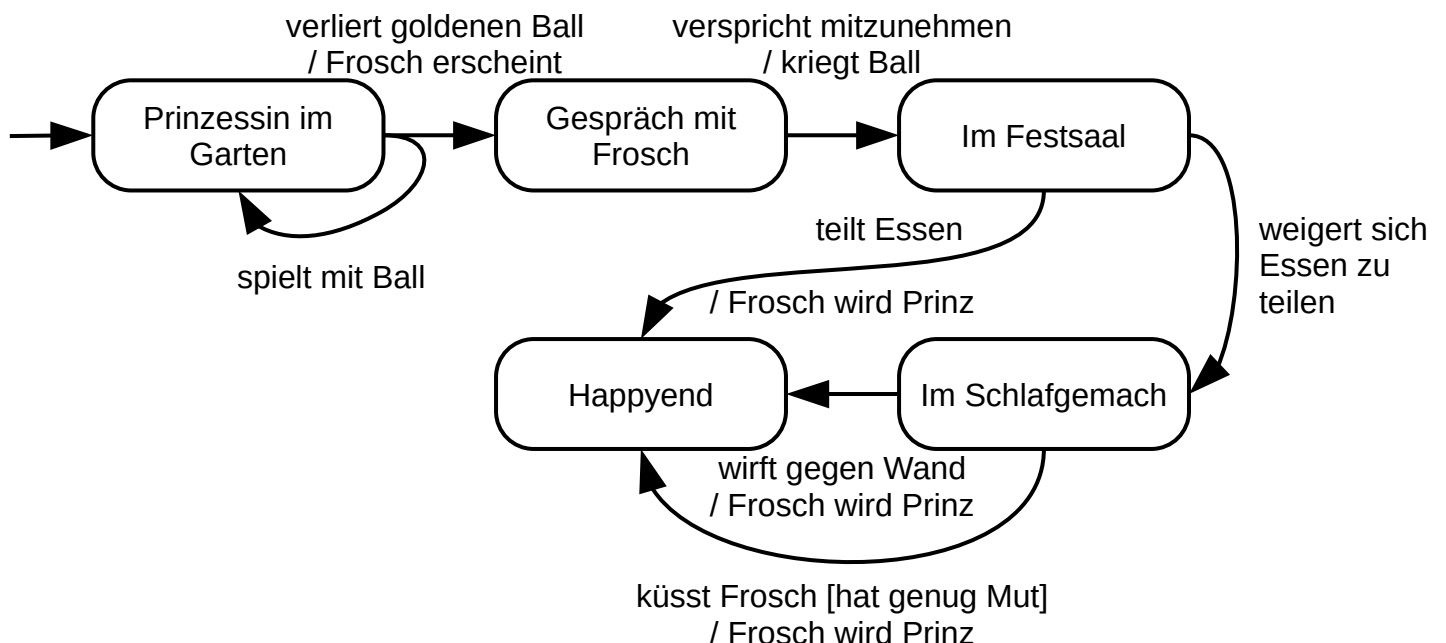


```

public Report createReport() {
    if (state==0) {
        Report r = new Report("Du bist im Garten eines Schlosses mit einem Ball.");
        r.addOption("Spiele mit dem Ball", "verlieren");
        return r;
    }
    ...
}

public void receiveMessage(String s) {
    if (state==0 && s.equals("verlieren")) {
        state=1;
    }
    ...
}
  
```

Zustandsdiagramm Froschkönig, nicht linear



Anhang 1a: Zentrale Methoden der Geschichten

Diese Methoden der Klasse **AbstractStory**, der obersten Geschichten-Klasse, müssen grundsätzlich von jeder Unterklasse implementiert werden. Ausnahme: `createReport(String)`.

Zentrale Methoden, zum Teil abstrakt, so dass sie implementiert werden müssen.	
<code>String getName()</code>	Wichtig: Legt den Namen der Geschichte fest, der zur Identifizierung und im GUI verwendet wird.
<code>void setImageName(String)</code>	Optional: Legt fest, ob es zur Geschichte ein Bild geben soll.
<i><code>Report createReport(String location)</code></i>	<i>Sehr wichtige Methode, die in einfachen Geschichten aber durch eine andere ersetzt wird und in diesen nicht implementiert wird.</i>
<code>void receiveMessage(String)</code>	Wichtig. Legt fest, wie sich die Geschichte verhält, wenn sie eine Entscheidung des Spielers erhält – siehe dazu auch einen separaten Abschnitt.

Methoden der Unterklasse SimpleStory, die für die einfachen Geschichten verwendet wird, mit denen wir uns am Anfang beschäftigen.

Abstrakte, wichtige Methoden, die implementiert werden müssen:	
<code>String startStoryAt()</code>	Diese Methode dient vor allem zum Testen, sie gibt den Namen des Ortes zurück, an dem sich die Geschichte am Anfang befindet. (Dieser Ort kann jederzeit durch <code>moveStoryTo</code> geändert werden, siehe oben.)
<code>Report createReport()</code>	Befindet sich der Spieler an dem Ort, der dem aktuellen Aufenthaltsort der einfachen Geschichte entspricht, dann wird die Geschichte gebeten, einen Bericht abzuliefern. Zu Berichten siehe separaten Abschnitt.

Anhang 1b: Mögliche ausgelöste Aktionen

Werden eigentlich nur innerhalb der Methode `receiveMessage(String)` aufgerufen.

Eine Auswahl von Methoden:

Auf verschiedene Weise den Zustand der Welt verändern:	
<code>void set(String, int)</code>	Einen Marker auf einen bestimmten Wert setzen, z.B. <code>set("Geld", 10)</code> . Wenn der Marker noch nicht existiert, wird er angelegt.
<code>void increase(String, int)</code>	Den Wert eines Markers um einen Wert verändern, kann auch negativ sein.
<code>void movePlayerTo(String)</code>	Den Spieler an einen anderen Ort bringen.
<code>void moveStoryTo(String locationName)</code>	Jede einfache Geschichte spielt immer nur an einem Ort gleichzeitig. Diesen Ort kann man aber ändern: eine Geschichte, die eine Person oder ein Schiff darstellt, kann mit <code>moveStoryTo("Amsterdam")</code> an diesen Ort versetzt werden. Diese Methode gibt es nur bei Objekten der Klasse <code>SimpleStory</code> .
<code>void introduceStory(AbstractStory)</code>	Eine neue Geschichte in die Welt einführen.
<code>void advanceTime(int)</code>	Erhöht die seit Spielbeginn vergangene Zeit.
Nachrichten senden, die in einem Textfeld der GUI dargestellt werden:	
<code>void sendMessage(String)</code>	Zum Beispiel <code>sendMessage("Du hast nicht genug Geld.");</code>
Spezielle Methoden, um Geschichten zu betreten oder zu verlassen, erst mal nicht wichtig:	
<code>void enter()</code>	Ab diesem Zeitpunkt kann der Spieler nicht mehr zu einer anderen Geschichte oder einem anderen Ort wechseln, sondern ist in der Geschichte gefangen...
<code>void exit()</code>	...bis die Geschichte ihm erlaubt, sie wieder zu verlassen.

Anhang 1c: Mögliche Bedingungen

Eine Auswahl von Methoden von Objekten der Klasse AbstractStory und allen Unterklassen:

Informationen über die Welt erfragen, zu verwenden zum Beispiel in bedingten Anweisungen:	
<code>int get(String)</code>	Gibt den aktuellen Wert eines Markers zurück, z.B. <code>get("Geld")</code> → 10
<code>boolean playerIsIn(String)</code>	Erfragt den Namen des Ortes, an dem sich der Spieler gerade befindet. Ist nur für eine bestimmte Unterart von Geschichten sinnvoll.
<code>int getTime()</code>	Gibt die aktuelle Zeit zurück. Standardmäßig wird Zeit nur beim Bewegen zwischen Orten erhöht.

Sonstige Methoden:

Weniger wichtige Methoden:	
<code>String getRandomNeighbourName(String)</code>	Gibt den Namen eines zufälligen Nachbarorts eines Ortes an.

Anhang 2: Wie man eine Welt erzeugt

Eine Welt besteht unter anderem aus Objekten der Klasse Location und Connections. Mit einer Klassenmethode erzeugt man ein World-Objekt, und mit folgenden Methoden kann man die Welt ändern. (Ein Beispiel für den Einsatz ist in der Klasse Quickstart.)

<code>World w = World.createEmptyWorld()</code>	Einzigste statische Klassenmethode als Ersatz für einen Konstruktor
<code>w.setWorldName(String)</code>	Der Name der Spielwelt, erscheint z.B. als Titel des Fensters
<code>w.setIntroduction(String)</code>	Kurzer Text zur Einführung, erscheint zum Spielstart...
<code>w.setStartingLocationName(String)</code>	Der Name des Orts, an dem der Spieler beginnt. Den muss es natürlich geben.
<code>w.createMarker(String,int)</code>	Legt einen beliebigen globalen Zähler an, der als quasi-globales Attribut den Zustand der gesamten Welt beschreibt und von allen Geschichten gelesen und verändert werden kann.
<code>w.addLocation(Location)</code>	Dieser Ort wird hinzugefügt. Wie man Orte anlegt: Siehe weiter unten.
<code>w.addConnection(Location,Location,int)</code>	Erzeugt eine bidirektionale Verbindung zwischen den zwei Orten von einer bestimmten Länge. (Die bleibt vorerst unberücksichtigt.)
<code>w.addStory(AbstractStory);</code>	Eine Geschichte wird hinzugefügt; AbstractStory ist die Oberklasse für jegliche Art von Geschichten.
<code>GameBuilder.start()</code>	Mit dieser Klassenmethode wird die angelegte Welt gestartet.

Anhang 3: Wie man eine Location anlegt

Es ist vor allem wichtig, der Welt Objekte der Klasse Location hinzuzufügen.

<code>Location(String)</code>	Konstruktor, der den Namen des Ortes als dessen Schlüssel festlegt.
<code>setDescription(String)</code>	Eine kurze Beschreibung des Ortes.
<code>setDescription(Long)</code>	Eine ausführlichere Beschreibung, ist optional.
<code>setImageName(String)</code>	Optional: Ein Bild zum Ort festlegen.
<code>setArea(String)</code>	Optional: Eine Gegen (Kontinent, Landschaftsart usw. zuweisen.)

(Alternativ kann man mit einer Unterklasse von TestBuilder beginnen, im Beispielprojekt: WorldSetup. Siehe dazu weiter unten.)

Anhang 4: Was in GameBuilder (oder Unterklassen von TestBuilder) eingestellt werden kann

Der Aufruf dieser statischen Klassenmethoden ist optional. Wichtig ist die erste Methode, die den Pfad *innerhalb des BlueJ-Projekts* zu dem Verzeichnis angibt, in dem sich das Material, also vor allem die Bilddateien, befinden. Das ist standardmäßig das oberste Verzeichnis innerhalb des Projekts. Alle Pfadangaben, um Bilder für Geschichten oder das GUI zu setzen, sind relativ zu diesem Oberverzeichnis. Das ermöglicht es, innerhalb eines BlueJ-Projekts verschiedene Spielwelt-Ordner zu haben.

Vielleicht wäre es vernünftiger, als Pfad anzugeben `setMaterialsFolder("resources")`, so dass man sich das später dann sparen und nur `setBackgroundImageName("game_background.png")` schreiben kann.

<code>setMaterialsFolder("");</code>	<p>Wichtig: Alle Pfadangaben zu Bildern und anderen Dateien gelten a) innerhalb des BlueJ-Projekts auf der obersten Ebene und b) relativ zu diesem Unterverzeichnis.</p> <p>Ist dieser String leer (""), wird also gesucht in "StoryWorld-Blue/", ist dieser String "project10b", wird gesucht in "StoryWorld-Blue/project10b/".</p>
<code>setBackgroundImageName("resources/game_background.png");</code>	Legt Bild fest für den Hintergrund.
<code>setHeaderImageName("resources/game_header.png");</code>	Legt Bild fest für den Titel.
<code>setCharacterImageName("resources/game_character_image.png");</code>	Legt Bild fest für die Figur rechts oben.
<code>setMapImageName("resources/miniwelt_karte.png");</code>	Wichtiger, aber es geht auch ohne: legt Karte fest für die ganze Welt.
<code>setDevelopmentMode(true);</code>	Spielt im Moment keine große Rolle.

Anhang 5: Das automatische Starten von Geschichten

Normalerweise legt man einfach eine Welt an und fügt ihr Geschichten hinzu und startet danach mit `GameBuilder.start()` das Spiel. Das ist eine statische Klassenmethode, der die Welt nicht als Argument übergeben werden muss, da die Welt als Singleton implementiert ist und über statische Methoden der Klasse World erreichbar ist.

- **Wenn** aber eine Main-Klasse existiert wie die im Beispielprojekt,
- und wenn die Geschichte eine Unterklasse von TestableStory ist,
- dann kann am Ende des Konstruktors dieser Geschichte die von TestableStory ererbte Methode `test()` aufgerufen werden.

Diese Methode legt a) die in WorldSetup angelegt Welt an, sofern Sie noch nicht existiert, und fügt b) die Geschichte in die Welt an. Wenn die Geschichte die erste Geschichte in dieser Welt ist, wird c) der Spieler an den Schauplatz der Geschichte versetzt.

(Weitere auf diese Art gestartete Geschichten werden ebenfalls in diese Welt gesetzt, aber der Aufenthaltsort des Spielers wird nicht geändert.)

Das ermöglicht schnelles Testen der Geschichten-Klasse.

Anhang 6: Beispiele für ganz einfache Geschichten

Jede *einfache* Geschichte spielt immer nur an *einem* Ort gleichzeitig. Befindet sich der Spieler an diesem Ort, wird die Geschichte gefragt, ob sie einen Report abgeben will. Die Geschichte kann entweder null zurückgeben, oder einen mehr oder weniger elaborierten Report. Dazu implementiert die einfache Geschichte die Methode `Report createReport()` – hier einige Beispiele im Zusammenhang einer ganzen Klasse.

Ein leeres Haus 1

Ein Haus, das nichts tut außer für den Spieler sichtbar zu sein.

```
import storyworld.*;

public class LeeresHaus1 extends SimpleStory {

    public LeeresHaus1() { test(); }

    @Override public String startStoryAt() { return "Hafen"; }

    @Override public String getName() { return "Das öde Haus"; }

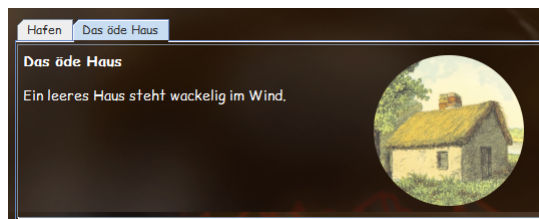
    @Override public Report createReport() {
        Report r = new Report("Ein leeres Haus steht wackelig im Wind.");
        return r;
    }

    @Override public String getImageName() {return "resources/haus.png";}

}
```

Das Aufrufen der `test()`-Methode am Ende des Konstruktors ist für das Spiel nicht nötig, ermöglicht aber ein einfaches Testen der Klasse. Die Methode `getImageName()` ist optional, aber hübsch.

Wenn der Spieler den Ort mit dem Namen “Hafen” betritt, wird die Methode `createReport()` des Geschichten-Objekts aufgerufen und gibt einen einfachen Text zurück, den der Spieler sieht:



Ein leeres Haus 2

Ein Haus, das dem Spieler eine Option anbietet, die aber zu nichts führt.

```
@Override public Report createReport() {
    Report r = new Report("Ein leeres Haus steht wackelig im Wind.");
    r.addOption("Du schaust es dir näher an", "anschauen");
    return r;
}
```

Hier wird jeweils nur die `createReport()`-Methode aufgeführt, der Rest bleibt gleich.

Wenn dem Report eine Option hinzugefügt wird (das können beliebig viele sein!), ändert sich die Darstellung für den Spieler. Pro Option erscheint ein Knopf mit dem ersten String-Argument der addOption-Methode (“Du schaust es dir näher an.”) als Beschreibung:



Jetzt hat der Spieler die Wahl, zurück zum Hafen oder zu einem anderen Ort zu gehen, oder er kann den Knopf für das Anschauen drücken. Wenn das geschieht, wird das zweite String-Argument als Nachricht an die Geschichte zurückgegeben, die daraufhin entscheidet, wie es weitergeht. Da unser LeeresHaus noch keine Methode dafür hat, geschieht beim Drücken des Knopfes erst einmal gar nichts.

Ein leeres Haus 3

Ein Haus, das dem Spieler eine Option anbietet, und darauf reagiert, wenn der Spieler sie wählt. Dazu muss lediglich die Methode receiveMessage(String) implementiert werden:

```
import storyworld.*;

public class LeeresHaus3 extends SimpleStory {

    public LeeresHaus3() { test(); }

    @Override public String startStoryAt() { return "Hafen"; }

    @Override public String getName() { return "Das öde Haus"; }

    @Override public Report createReport() {
        Report r = new Report("Ein leeres Haus steht wackelig im Wind.");
        r.addOption("Du schaust es dir näher an", "anschauen");
        return r;
    }

    @Override public String getImageName() {return "resources/haus.png";}

    @Override public void receiveMessage(String s) {
        if (s.equals("anschauen")) {
            sendMessage("Niemand zu Hause");
            if (get("Neugier")<10) increase("Neugier", 1);
        }
    }
}
```

Wenn die Geschichte die Nachricht “anschauen” erhält, werden zwei Aktionen ausgelöst: Eine Textnachricht wird gesendet (eine Notlösung, schöner wäre es, einen ganz neuen Report anzubieten, wobei dann der Report vom Zustand der Geschichte abhinge – siehe separaten Abschnitt). Außerdem wird der globale Marker “Neugier” um 1 erhöht, sofern er aktuell einen Wert geringer als 10 hat.

Ergänzungen

Natürlich könnte man den abgegebenen Report vom Zustand der Welt abhängig machen:

```
@Override public Report createReport() {
    Report r = null;
    if (get("Steuern") > 10) r = new Report("Niemand rührt sich.");
    else r = new Report("Ein Mann winkt fröhlich aus dem Haus");
    return r;
}
```

Oder, vielleicht häufiger, vom Zustand der Geschichte:

```
@Override public Report createReport() {
    Report r = null;
    if (this.anzahlBesuche>3) {
        r = new Report("Das öde Haus ist völlig verlassen.");
    }
    else {
        r = new Report("Du hörst ein leises Kratzen von innen.");
        r.addOption("Du durchsuchst das Haus.", "durchsuchen");
    }
    return r;
}
```

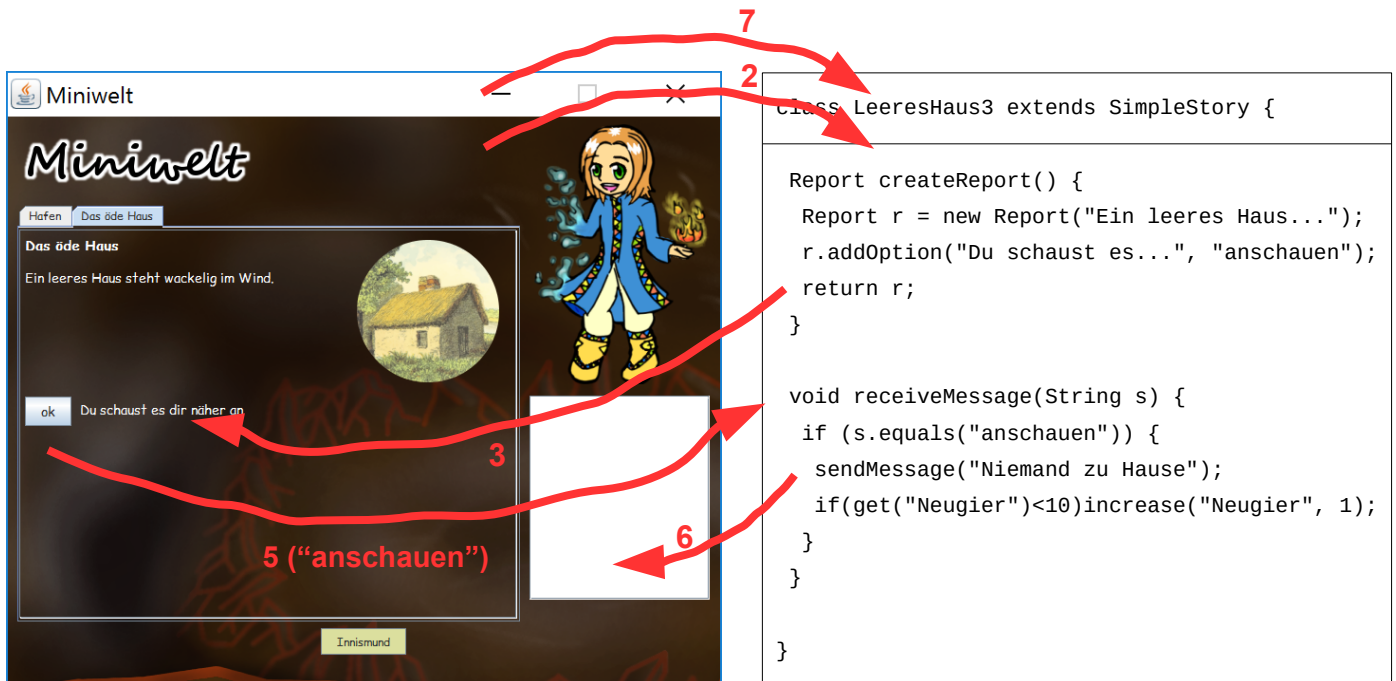
Immer wenn der Spieler die Option “durchsuchen” auswählt, wird `anzahlBesuche` um 1 erhöht, bis nach dem dritten Besuch die Option nicht mehr zur Verfügung gestellt wird. Dazu braucht es natürlich eine Methode wie diese:

```
@Override public void receiveMessage(String s) {
    if (s.equals("durchsuchen")) {
        anzahlBesuche++;
    }
}
```

Anhang 7: Das Zusammenspiel von createReport() und receiveMessage(String) im Diagramm

(Siehe auch die Präsentation.)

1. Spieler betritt Ort „Hafen“.
2. Geschichte an diesem Ort: Aufruf von createReport()
3. Liefert Report ab, der im entsprechenden Reiter dargestellt wird, mit allen Optionen.
4. Spieler klickt Knopf neben einer Option.
5. Methode receiveMessage wird mit der Nachricht der Option als Parameter aufgerufen.
6. Geschichte reagiert...
7. ...und wird sofort danach um erneuten Report gebeten. (Der hier allerdings gleich ausfällt.)



Anhang 8: Erweiterungen

Geschichten beenden

Wenn eine Geschichte fertig ist, kann sie entweder einen abschließenden Report präsentieren, der jedesmal zu sehen ist, oder einen null-Report abliefern. Sie kann auch mit `setActive(false)` deaktiviert oder mit `deleteStory(this)` ganz aus dem Repertoire verfügbarer Geschichten gelöscht werden.

`enter()` und `exit()`

Standardmäßig kann man jederzeit eine Geschichte verlassen und zu einem anderen Ort und einer anderen Geschichte gehen, auch wenn der Text der Geschichte sagt, dass man gerade in einem tiefen Keller gefangen ist. Das ist für den Schuleinsatz auch sinnvoll. Nach `receiveMessage(String)` kann man aber auch diese beiden Aktionen auslösen: Mit `enter()` betritt man die Geschichte so, dass es keinen Rückweg zur Karte mehr gibt, man also in der Geschichte gefangen ist – bis irgendwann die Aktion `exit()` ausgelöst wird, so dass man wieder zur Navigation auf der Karte kommt.

ActionStory

Wenn eine Geschichte das Interface `ActionStory` implementiert, muss sie eine Methode `public void act()` besitzen, und diese Methode wird automatisch bei jeder Bewegung des Spielers zu einem anderen Ort aufgerufen. Beispiele zur Verwendung in der Klasse `Wanderer`, der sich bei jedem Zug in einen zufälligen Nachbarort begibt, und in der Klasse `Tribble`, wo bei jedem Zug die Chance besteht, dass ein neues `Tribble`-Objekt an einen zufälligen Ort angelegt wird, sofern dieser Ort als `area`-Attributwert "land" hat. Das gibt exponentielles Wachstum, deshalb gibt es eine Obergrenze und die Möglichkeit, Tribbles einzusammeln. Sind alle Tribbles kassiert, kommen auch keine mehr nach.

ShopStory

Eine `ShopStory` ist ein Laden, der sich als Unterklasse zu `SingleLocationStory` immer an einem Ort befindet. Es gibt Methoden, diesem Laden Waren hinzuzufügen – zu Kauf und Verkauf, oder nur zu einem davon, mit einer Startzahl, wieviel davon am Lager sind. Waren werden über Marker realisiert. Man zahlt für Waren mit anderen Markern, etwa mit diesem ausführlichen Methodenaufruf:

```
addItem("Kaffee", 20, "Geld", 10, "Geld", 5)
```

=> der Laden hat 20 Kaffee vorrätig und verkauft jeweils 1 Kaffee für 10 Geld und kauft neuen Kaffee für 5 Geld. Verschiedene andere Methoden sind ebenfalls möglich.

Weiteres

`WorldEvents`, Reisen zwischen Orten, und das Vergehen der Zeit: Soweit kommen wir eh nicht. Gibt es alles, mehr oder weniger rudimentär.

Anhang 9: Zur Programmierertechnik

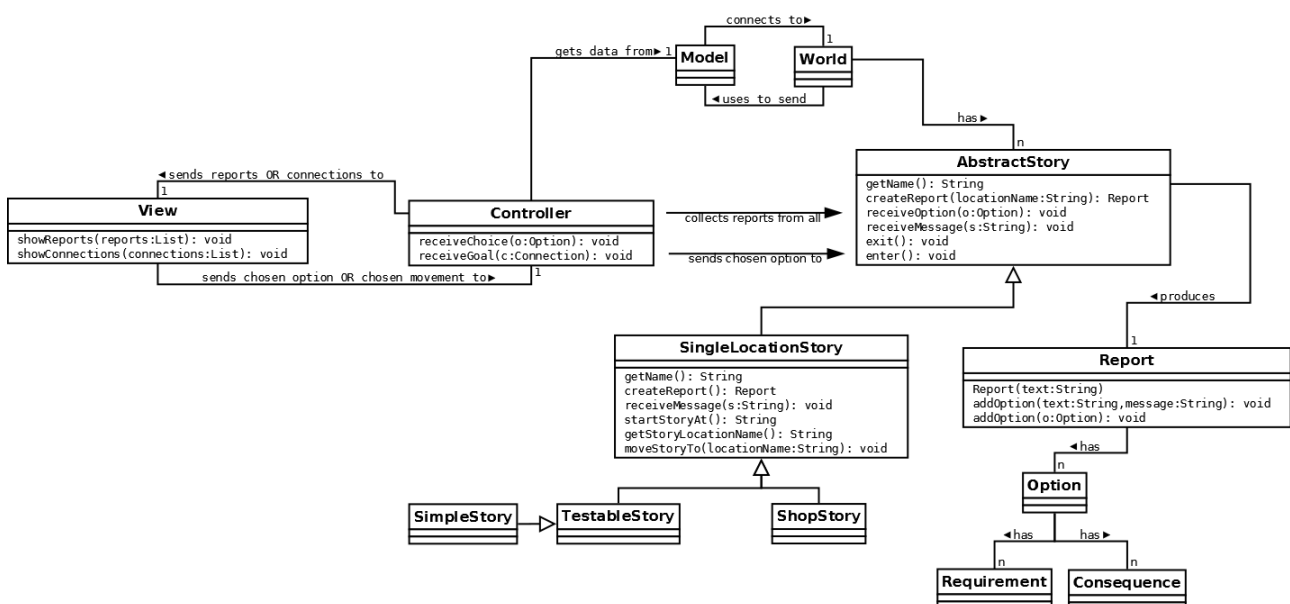
Vermutlich würde man für ein vergleichbares professionelles Projekt eine Makrosprache definieren, in der sich die Geschichten ohne Java-Syntax knapper erzeugen lassen. Aber schließlich sollen die Schülerinnen und Schüler ja Java lernen. Auch die Ausrichtung auf endliche Zustandsautomaten ist vielleicht gar nicht so sinnvoll, aber die gehören zum Stoff der 10. Jahrgangsstufe.

Unbefriedigend ist auch das Wechselspiel von createReport und receiveMessage, tatsächlich könnte man dazu auch das Command-Entwurfsmuster benutzen. Das ist auch bereits implementiert: Man kann beim Erstellen jedes Reports jeder Option ein Consequence-Objekt (oder mehrere) übergeben, dessen execute-Methode ausgeführt wird, wenn die Option gewählt wird. Dann kann man auf die receiveMessage-Methode ganz verzichten. Außerdem kann man jeder Option ein Requirement-Objekt (oder mehrere) übergeben, so dass diese Option zwar sichtbar, aber deaktiviert ist, wenn die Bedingungen nicht erfüllt sind. Fertige Consequence- und Requirement-Unterklassen erleichtern das Arbeiten damit. Trotzdem dürfte das für die 10. Klasse zu abstrakt sein.

Auf Effizienz wurde nicht viel Wert gelegt.

Am unsaubersten ist alles, was mit GUI zu tun hat.

Kleines, nicht vollständiges Klassendiagramm

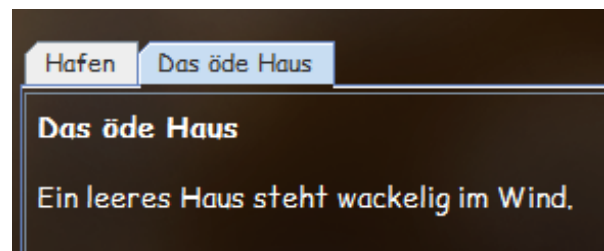
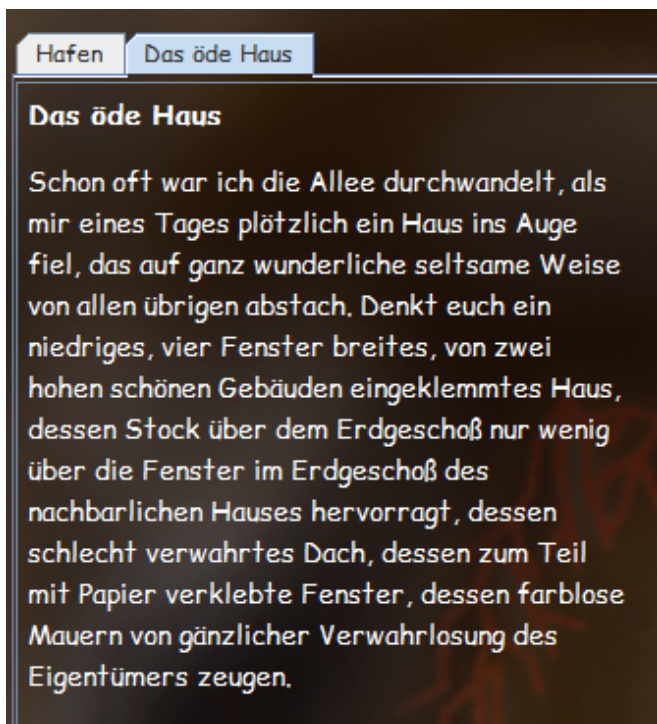


(Die Klasse `abstractStory` verfügt noch über sehr viel mehr Methoden.)

Anhang 10: Gute Geschichten

Ob das Projekt ganze als Spiel befriedigend ist, hängt in überraschendem Maß nicht von der Programmiertechnik oder technischen Finessen ab, sondern – bei Filmen und Computerspielen eben genauso wie bei erzählender Literatur – vom Text. Dazu gehören knappe, aber dennoch anschauliche und lebendige Schilderungen, dramatische Situationen. Die Ortsbeschreibungen und die Reporte sollten nicht zu kurz und nicht zu lang sein. Lieber weniger Zustände bei einer Geschichte, die liebevoll erzählt sind, als viele, vielleicht auch noch nur knapp beschriebene, die das ganze dann nur zu einem Labyrinth machen.

Hier eine Beschreibung in einem Satz verglichen mit einem Auszug aus der Novelle “Das öde Haus” von E.T.A. Hoffmann:



Vom Zusammenspiel

Jede Geschichte kann für sich alleine stehen, aber auch mit anderen Geschichten zusammenhängen. Ein böser Schurke wird erst dann aktiv und sichtbar, wenn der Moralwert des Spielers auf einen negativen Wert sinkt? Wenn der Unheil-Marker der Welt über einen bestimmten Wert gerät, reagieren Personen anders als bisher.

Über die Welt

Meiner Erfahrung nach geht es schneller, wenn man der Klasse eine nicht zu große, relativ generische Spielwelt vorgibt.

Denkbar ist auch, dass nicht alle Geschichten von Anfang an aktiv sind, sondern erst nach und nach freigeschaltet oder hinzugefügt werden, wenn bestimmte Bedingungen erfüllt sind.

Ideen für Geschichten

Gebäude: Turm, Kirche, Gruft. Rathaus (ein bürokratisches Labyrinth). Ein Pokestop oder Spielcasino, bei dem man etwas gewinnen kann. Eine Wasserquelle oder ein Gasthaus, um sich zu erfrischen. (Gedacht für mehrere Instanzen an beliebigen Orten.)

Pflanzen und Tiere: Giftige Pilze. Ein Reh.

Menschen: Ein Witzeerzähler, der bei jedem Kontakt einen zufällig ausgewählten Witz erzählt.

Fahrzeuge: Eine Fähre, die zwischen einer Insel und dem Hafen verkehrt.

Tribbles: Diese Geschichte erzeugt mit einer gewissen Wahrscheinlichkeit (und einer Obergrenze, da exponentielles Wachstum) neue Instanzen seiner Klasse, die eingefangen werden können, und fügt sie der Welt hinzu.

Ein Wanderer: Zieht zufällig durch die Welt, jeweils an einen Nachbarort des aktuellen Aufenthalts, so dass der Spieler ihm Nachlaufen kann.

Zwei Wanderer (aber *eine* Geschichte), die einander suchen; wenn sie sich finden, verschwinden sie – außer der Spieler ist gerade am gleichen Ort, dann trifft er sie als gemeinsame Geschichte an.

Ein Wanderer, der sich auf einer bestimmten Route bewegt (ein String-Array mit Ortsnamen).

Leicht: Ringförmige Route, etwas schwerer: hin und her pendelnd. Dann: Die Route auslagern und eigene Klasse dafür anlegen, so dass man Wanderern verschiedene Routen zuweisen kann.

Aufgabe: Eine Geschichte, die selber einmalig Locations anlegt und den Spieler dorthin setzt – quasi mit eigenem Gewölbe, das man untersuchen kann. (Ein Problem ist bisher nur, dass das Namen-Attribut von Locations als Schlüssel verwendet wird und deshalb eindeutig sein muss.)