C++: Eine kompakte Einführung

von
André Willms

1. Auflage

dpunkt.verlag 2015

Verlag C.H. Beck im Internet:
www.beck.de
ISBN 978 3 86490 229 1

Zu Inhaltsverzeichnis

21 Das Spiel

In diesem Kapitel wollen wir abschließend noch ein paar Elemente des Spielprojekts ins Auge fassen.

21.1 Aktionen

Für die Spielregeln fehlen uns nur noch die Aktionen. Diese folgen einem ähnlichen Schema wie die Bedingungen in Abschnitt 18.8. Es gibt eine gemeinsame abstrakte Basisklasse `Aktion`, von der die konkreten Aktionen ableiten, wie Abbildung 21–1 zeigt.

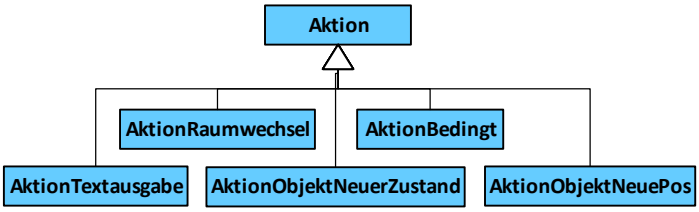

Abb. 21–1 *Die Hierarchie der Aktionen*

Die Basisklasse `Aktion` sieht so aus:

```
class Aktion : public Spielobjekt {
public:
  Aktion(Spiel* spiel) : Spielobjekt{spiel} {}
  virtual ~Aktion() = default;

  virtual bool ausfuehren()  =0;

};
```

Die Methode `ausfuehren` liefert einen booleschen Wert zurück, ob die Methode ausgeführt wurde. Betrachten wir exemplarisch einige Aktionen, zuerst `AktionRaumwechsel`:

```
class AktionRaumwechsel : public Aktion {
  id_type raum;
public:
  AktionRaumwechsel(Spiel* spiel, id_type r)
  : Aktion{spiel}, raum{r} {}
  virtual bool ausfuehren();
};
```

Die Aktion speichert den Raum, zu dem gewechselt werden soll. Über `ausfuehren` wird dann die entsprechende Spielmethode aufgerufen:

```
bool AktionRaumwechsel::ausfuehren() {
  get_spiel()->wechsel_raum(raum);
  return true;
}
```

Die Methode ist an keinerlei Bedingungen geknüpft, deshalb gibt sie immer `true` zurück. Anders sieht es bei `AktionBedingt` aus. Objekte dieser Klasse bestehen aus einer oder mehreren Aktionen, die nur dann ausgeführt werden, wenn vorher definierte Bedingungen erfüllt sind:

```
class AktionBedingt : public Aktion {
  public:
  using bedingung_ptr = std::unique_ptr<Bedingung>;
  using aktion_ptr = std::unique_ptr<Aktion>;

private:
  std::vector<bedingung_ptr> bedingungen;
  std::vector<aktion_ptr> aktionen;
  bool single_action;

  bool pruefe_bedingungen() const;
  bool fuehre_aktionen_aus();

public:
  AktionBedingt(Spiel* spiel, bool single=false);

  virtual ~AktionBedingt() = default;

  virtual bool ausfuehren();

  void add_bedingung(Bedingung* b) {
    bedingungen.push_back(bedingung_ptr{b});
  }

  void add_aktion(Aktion* b) {
    aktionen.push_back(aktion_ptr{b});
  }
};
```

Wie auch die Listen aus Abschnitt 19.6 übernimmt die bedingte Aktion den Besitz der in ihr gespeicherten Bedingungen und Aktionen und verwaltet diese mit `unique_ptr`-Objekten.

Interessant ist die boolesche Variable single_action. Diese Variable entscheidet, ob immer alle Aktionen ausgeführt werden (false) oder die Abarbeitung der Aktionen abgebrochen wird, sobald eine erfolgreich ausgeführt wurde (true).

Die Methode ausfuehren greift hauptsächlich auf die Methoden pruefe_bedingungen und fuehre_aktionen_aus zurück:

```
bool AktionBedingt::ausfuehren() {
  if(!pruefe_bedingungen())
    return false;

  return fuehre_aktionen_aus();
}
```

Die Methoden pruefe_bedingungen und fuehre_aktionen_aus sehen so aus:

```
bool AktionBedingt::pruefe_bedingungen() const {
  for(auto iter=bedingungen.begin();
      iter!=bedingungen.end();
      ++iter) {
    if(!(*iter)->pruefen())
      return false;
  }
  return true;
}

bool AktionBedingt::fuehre_aktionen_aus() {
  bool erfolg=false;
  for(auto iter=aktionen.begin(); iter!=aktionen.end(); ++iter) {
    if((*iter)->ausfuehren()) {
      if(single_action)
        return true;
      erfolg=true;
    }
  }
  return erfolg;
}
```

Die Methode pruefe_bedingungen liefert nur dann true zurück, wenn alle Bedingungen true ergeben. Damit handelt es sich um eine UND-Verknüpfung. Für ODER-Verknüpfungen von Bedingungen haben wir in Abschnitt 18.8 die Klasse BedingungOder programmiert.

Für das Ausführen der Aktionen ist es wichtig, ob alle Aktionen ausgeführt werden sollen oder die Ausführung nach der ersten erfolgreichen Aktion abgebrochen werden soll. Darüber entscheidet die Variable single_action, die dafür in fuehre_aktionen_aus abgefragt wird.

21.2 Spielregeln

Eine sehr wichtige Klasse im Spiel ist `Spielregel`. Mit ihren Objekten werden faktisch alle möglichen Aktivitäten und Beschreibungen im Spiel definiert.

Eine Spielregel besteht aus folgenden Teilen:

- Einem oder mehreren Triggern
- Einer oder mehreren Bedingungen
- Einer oder mehreren Aktionen

Die Klasse `Spielregel` benötigt folgende Methoden:

- `ist_trigger`, liefert `true`, falls einer der Trigger zum übergebenen Befehl passt.
- `pruefe_bedingungen`, liefert `true`, wenn alle Bedingungen erfüllt sind.
- `fuehre_aktionen_aus`, führt entweder alle Aktionen aus oder stoppt nach der ersten erfolgreichen Ausführung einer Aktion (darüber entscheidet das Attribut `single_action`).
- `ausfuehren`, führt die Aktionen aus, wenn die Spielregel einen passenden Trigger besitzt und alle Bedingungen erfüllt sind.

Bedenken Sie, dass manche Befehle symmetrisch sein können. Diesen Sachverhalt haben wir beispielsweise bei den Triggern in Abschnitt 7.11 umgesetzt.

Übung

Programmieren Sie die Klasse `Spielregel`. Viele der notwendigen Funktionalitäten haben wir in ähnlicher Form bereits in anderen Spielklassen programmiert. Versuchen Sie bei Unsicherheiten, sich an diesen zu orientieren.

Überlegen Sie sich, ob und wie eine Spielregel Ownership für andere Objekte übernimmt.

Lösung

```
class Spielregel : public Spielobjekt {
public:
  using bedingung_ptr = typedef std::unique_ptr<Bedingung>;
  using aktion_ptr = std::unique_ptr<Aktion>;

private:
  std::vector<Trigger> trigger;
  std::vector<bedingung_ptr> bedingungen;
  std::vector<aktion_ptr> aktionen;
  bool symmetrie;  // true, wenn Objektreihenfolge egal
                   // (Verwende X mit Y oder Verwende Y mit X)

  bool single_action; // Bricht die Aktionsverarbeitung ab,
                      // sobald eine Aktion erfolgreich
                      // durchgeführt wurde
```

```
  bool ist_trigger(id_type c, id_type o1, id_type o2) const;
  bool pruefe_bedingungen() const;
  bool fuehre_aktionen_aus();

public:
  Spielregel(Spiel* spiel,
             id_type b, id_type o1=-1, id_type o2=-1,
             bool sym = false, bool single=false);
  virtual ~Spielregel() = default;
  virtual bool ausfuehren(id_type b, id_type o1, id_type o2);

  void add_bedingung(Bedingung* b) {
    bedingungen.push_back(bedingung_ptr(b));
  }
  void add_aktion(Aktion* b) {
    aktionen.push_back(aktion_ptr(b));
  }
  void add_trigger(id_type b, id_type o1, id_type o2) {
    trigger.push_back(Trigger(b,o1,o2));
  }
};
```

Weil bei Bedingungen und Aktionen Polymorphie eingesetzt wird, werden Objekte dieser Klassen dynamisch erzeugt und deren Ownership an `unique_ptr`-Objekte übergeben. Bei Triggern ist das unnötig, weil es sich nicht um einen polymorphen Typ handelt. Wir speichern `Trigger`-Objekte daher direkt in einem Vektor.

Der Konstruktor benötigt mindestens die Informationen für einen Trigger. Später können dann immer noch Bedingungen und Aktionen über `add_bedingung` und `add_aktion` hinzugefügt werden:

```
Spielregel::Spielregel(Spiel* spiel,
                       id_type b, id_type o1, id_type o2,
                       bool sym, bool single)
  : Spielobjekt{spiel}, symmetrie{sym}, single_action{single} {

  add_trigger(b, o1, o2);
}
```

Die Methode `ausfuehren` prüft zunächst, ob die Spielregel einen passenden Trigger besitzt. Wenn ja, werden die Bedingungen geprüft und für den Fall, dass alle erfüllt sind, die Aktionen ausgeführt:

```
bool Spielregel::ausfuehren(id_type b, id_type o1, id_type o2) {
  if(!ist_trigger(b, o1, o2))
    return false;

  if(!pruefe_bedingungen())
    return false;

  return fuehre_aktionen_aus();
}
```

Das Prüfen der Trigger ist einfach, weil die Funktionalität bereits in die Klasse `Trigger` eingebaut wurde. Sobald ein passender Trigger gefunden wurde, wird `true` zurückgegeben:

```
bool Spielregel::ist_trigger(id_type c,
                             id_type o1, id_type o2) const {
  for(auto iter=trigger.begin(); iter!=trigger.end(); ++iter) {
    if(iter->ist_trigger(c, o1, o2, symmetrie))
      return true;
  }
  return false;
}
```

Da zur Ausführung der Aktionen alle Bedingungen gelten müssen, kann `pruefe_bedingungen` den Wert `false` zurückgeben, sobald eine Bedingung nicht erfüllt ist:

```
bool Spielregel::pruefe_bedingungen() const {
  for(auto iter=bedingungen.begin(); iter!=bedingungen.end(); ++iter) {
    if(!(*iter)->pruefen())
      return false;
  }
  return true;
}
```

Die Methode `fuehre_aktionen_aus` ist identisch mit der aus `AktionBedingt`:

```
bool Spielregel::fuehre_aktionen_aus() {
  bool erfolg=false;
  for(auto iter=aktionen.begin(); iter!=aktionen.end(); ++iter) {
    if((*iter)->ausfuehren()) {
      if(single_action)
        return true;
      erfolg=true;
    }
  }
  return erfolg;
}
```

21.3 Räume

Als letzte Klasse aus dem Spiel wollen wir uns `Raum` ansehen:

```
class Raum : public BenanntesObjekt {
private:
  std::string beschreibung;
  std::vector<id_type> ausgaenge;

public:
  Raum(Spiel* spiel, id_type id,
       const std::string& n, const std::string& beschr);
```

```
  virtual ~Raum() = default;

  void schreibe_verfuegbare_richtungen() const;

  int get_ausgangid_by_partial_name_match(const std::string& s)
                                                          const;

  std::string get_beschreibung() const {
    return beschreibung;
  }
  void add_ausgang(id_type id) {
    ausgaenge.push_back(id);
  }

};
```

Die Ausgänge werden nicht als Verweise auf Objekte, sondern über deren IDs gespeichert. Dieses Prinzip zieht sich durch das gesamte Spiel. Der große Vorteil davon ist die Unabhängigkeit von der konkreten Programmstruktur. Ein Raum hat immer dieselbe ID, aber bei jedem Programmstart eine andere Adresse im Speicher. Deshalb lassen sich die Beziehungen über IDs später viel einfacher speichern und laden.

Mit all den Klassen, die wir bisher schon programmiert haben, fällt der Konstruktor nicht mehr schwer.

```
Raum::Raum(Spiel* spiel, id_type id,
           const std::string& n, const std::string& beschr)
    : BenanntesObjekt{spiel, id, n}, beschreibung{beschr} {

}
```

Die Methode `schreibe_verfuegbare_richtungen` ist wieder eine vereinfachte Version der Methode aus dem Spiel. Hier betrachten wir die Grundidee der Ausgabe der Ausgänge und vernachlässigen Zeichencodierung und Textformatierung. Alle sichtbaren Ausgänge werden aufgelistet. Hier wird zum ersten Mal bewusst eingesetzt, dass Ausgänge auch nur Gegenstände sind, allerdings besitzen die Ausgänge keine für den Rest des Spiels sichtbare Entsprechung, was mit der Position -1 umgesetzt wird (Abschnitt 11.8):

```
void Raum::schreibe_verfuegbare_richtungen() const {
  for(auto iter=ausgaenge.begin();
      iter!=ausgaenge.end();
      ++iter) {
    Gegenstand* ptr = get_spiel()->get_gegenstaende()->
                      get_gegenstand_by_id(*iter);
    if(ptr->ist_sichtbar()) {
      cout << ptr->get_name() << endl;
    }
  }
}
```

Die Methode get_ausgangid_by_partial_name_match liefert die ID des ersten Ausgangs, der sichtbar ist und dessen Name mit dem übergebenen String beginnt. Sollte es einen solchen Ausgang nicht geben, wird -1 zurückgegeben:

```
int Raum::get_ausgangid_by_partial_name_match(const string& s) const {
  for(auto iter=ausgaenge.begin();
      iter!=ausgaenge.end();
      ++iter) {
    id_type id=*iter;
    Gegenstand* ptr=get_spiel()->get_gegenstaende()->
                    get_gegenstand_by_id(id);
    if(ptr->ist_sichtbar() && ptr->partial_name_match(s))
      return id;
  }
  return -1;
}
```

21.4 Die Erzeugung der Spielwelt

Bisher diente die gesamte Programmierung der technischen Spielstruktur. Wir haben gewissermaßen die Infrastruktur geschaffen, mit der wir eine Spielwelt erzeugen und mit ihr interagieren können. Wie das funktioniert, schauen wir uns in den folgenden Abschnitten an.

Die Klasse Spiel beinhaltet alle zum Spielen notwendigen Methoden, und nur diese. Um bei der Erstellung der Spielstruktur komfortablere Möglichkeiten zu besitzen, habe ich von der Klasse Spiel die Klasse SpielBauer abgeleitet, die nur Hilfsmethoden für den Bau der Spielwelt besitzt.

21.4.1 Erstellen der Räume

Zum Erstellen der Räume habe ich in SpielBauer die Hilfsmethode erzeuge_raum programmiert:

```
void SpielBauer::erzeuge_raum(id_type id,
      const std::string& name, const std::string& beschr) {
  Raum* tmpraum;
  tmpraum=new Raum{this, id, name, beschr};
  get_raeume()->add_objekt(tmpraum);
}
```

Die Methode erzeugt einen neuen Raum und fügt ihn der Raumliste des Spiels hinzu. Als Beispiel sehen Sie die Erzeugung der ersten Räume, die Ihnen im Spiel begegnen. Aus Platzgründen habe ich hier zu lange Beschreibungen abgekürzt:

```
erzeuge_raum(100, "Flur, Erdgeschoss",
  "Du befindest dich auf einem Flur im Erdgeschoss.");
erzeuge_raum(200, "Flur, Erdgeschoss, hinterer Teil",
  "Du befindest dich im hinteren Teil des Flurs. Hier … ");
```

```
erzeuge_raum(600, "Toilette",
  "Das ist das Gästeklo des Hauses. Klein, aber fein.");
```

21.4.2 Erzeugen der Gegenstände

Für das Erzeugen von Gegenständen habe ich ebenfalls eine Hilfsmethode programmiert, die folgendermaßen aussieht:

```
void SpielBauer::erzeuge_gegenstand(id_type id, id_type raum,
          bool fest, bool ablegbar, bool sichtbar,
          const std::string& name, const std::string& bez,
          const std::string& beschr, int z) {

  get_gegenstaende()->add_objekt(
    new Gegenstand{this, id, name, bez, fest, ablegbar,
                   sichtbar, raum, z});
  if(beschr!="") {
    Spielregel* tmpregel;
    get_regeln()->add_regel(
      tmpregel=new Spielregel{this, CMD_BESCHREIBEN, -1, -1});
    tmpregel->add_bedingung(
                      new BedingungObjektInAktRaum{this, id});
    tmpregel->add_bedingung(
                     new BedingungObjektIstSichtbar{this, id});
    if(!fest)
      tmpregel->add_bedingung(
                       new BedingungObjektAnOrgPos{this, id});
    tmpregel->add_aktion(new AktionTextausgabe{this, beschr});
  }
}
```

Die Methode bekommt die ID des neuen Gegenstands, den Raum/Ort, wo der Gegenstand sein soll, sowie alle für den Gegenstand wichtigen Attribute und eine Beschreibung übergeben.

Die Beschreibung ist der Text, der erscheint, wenn der Gegenstand an seinem ursprünglichen Platz ist und der Spieler im entsprechenden Raum steht. Zu Beginn des Spiels im Flur steht bei der Beschreibung des Raums unter anderem der Text »Direkt neben der Treppe in einer kleinen Ecke haben sich Haare gesammelt. Nicht gerade sauber hier.« Das ist der Beschreibungstext des Haarknäuels.

Schauen wir uns einmal an, wie die Methode einen Gegenstand erzeugt. Als Erstes wird der neue Gegenstand erzeugt und zur Gegenstandsliste hinzugefügt. Sollte eine Beschreibung angegeben worden sein, dann wird eine Spielregel erzeugt, die dafür sorgt, dass die Beschreibung bei der Raumbeschreibung angezeigt wird.

Als Trigger wird der Befehl `CMD_BESCHREIBEN` verwendet. Ich habe für alle Befehle Konstanten erstellt, damit die Regeln leichter lesbar sind. Der Befehl `CMD_BESCHREIBEN` ist ein virtueller Befehl. Er wird nicht direkt vom Spieler angege-

ben, sondern die Spiellogik ruft bei jeder Raumbeschreibung diesen Befehl auf, damit die Gegenstände die Gelegenheit haben, ihre Beschreibung auszugeben.

Die Bedingungen zur Ausgabe des Beschreibungstextes sind:

- Der Gegenstand muss sich in dem Raum befinden, in dem der Spieler gerade ist.
- Der Gegenstand muss sichtbar sein.
- Und für den Fall, dass der Gegenstand nicht fest an seiner Position ist, muss er sich noch an seiner Originalposition befinden.

Wenn alle Bedingungen erfüllt sind, wird die `AktionTextausgabe` ausgeführt.

Hier als Beispiel die Erzeugung des Haarknäuels:

```
erzeuge_gegenstand(102,100,false,false,true,"Haarknäuel",
"ein Haarknäuel", "Direkt neben der Treppe in einer…");
```

21.4.3 Erstellen der Ausgänge

Wie der Implementierung der `Raum`-Klasse zu entnehmen ist, kennt ein Raum nur die Wege, die von ihm weg führen. Die Wege, die zu ihm hin führen sind in den Räumen untergebracht, von denen sie weg führen.

Das hat den Vorteil, dass auch Einbahnwege problemlos umgesetzt werden können. Springt der Spieler zum Beispiel von einer hohen Mauer, dann führt kein Weg zurück, weil es zu hoch ist.

In den meisten Fällen führen Wege aber in beide Richtungen, wie zu Beginn des Spiels vom vorderen Flur zum hinteren Flur und zurück. Der Einfachheit halber habe ich dafür wieder eine Hilfsmethode programmiert.

```
void SpielBauer::erzeuge_zweiweg_verbindung(
 id_type raum1, id_type raum2,
 id_type hinid, const std::string& hinname, bool hinsichtbar,
 id_type zurueckid, const std::string& zurueckname,
 bool zuruecksichtbar
                              ) {
  Raum* r1 = get_raeume()->get_raum_by_id(raum1);
  Raum* r2 = get_raeume()->get_raum_by_id(raum2);
  Spielregel* tmpregel;

  get_gegenstaende()->add_objekt(
     new Gegenstand{this, hinid, hinname, hinname,
                    true, false, hinsichtbar, -1});
  get_gegenstaende()->add_objekt(
     new Gegenstand{this, zurueckid, zurueckname, zurueckname,
                    true, false, zuruecksichtbar, -1});
  r1->add_ausgang(hinid);
  r2->add_ausgang(zurueckid);
  tmpregel=new Spielregel{this, CMD_GEHE, hinid, -1};
  tmpregel->add_aktion(new AktionRaumwechsel{this, raum2});
  get_regeln()->add_regel(tmpregel);
```

```
    tmpregel=new Spielregel{this, CMD_GEHE, zurueckid, -1};
    tmpregel->add_aktion(new AktionRaumwechsel{this, raum1});
    get_regeln()->add_regel(tmpregel);
  }
```

Die Methode bekommt die beiden zu verbindenden Räume und die Daten für die beiden Ausgänge übergeben.

Die Ausgänge sind auch Gegenstände, deshalb müssen diese neu erzeugt und der Gegenstandsliste hinzugefügt werden. Als ID werden die übergebenen IDs verwendet und die Position wird auf -1 gesetzt (kein greifbares Objekt).

Anschließend wird für jeden Ausgang eine Spielregel erzeugt, die als Trigger `CMD_GEHE` mit dem jeweiligen Ausgang hat und als Aktion einen Raumwechsel zum Zielraum.

Die folgenden Anweisungen erzeugen die Verbindung vom vorderen Flur zum hinteren Flur und vom vorderen Flur zur Toilette:

```
erzeuge_zweiweg_verbindung(100,200,
                           100000,"vor", true,
                           100010,"zurück", true);
erzeuge_zweiweg_verbindung(100,600,
                           100100,"toilette", true,
                           100110,"flur", true);
```

21.4.4 Erzeugen von Türen

Die Räume sind über Ausgänge verbunden. Im Beispiel der Verbindung zwischen dem vorderen und dem hinteren Flur sind diese Ausgänge immer verwendbar.

Es gibt aber auch Türen, wie beispielsweise zwischen dem Flur und der Toilette. Die Ausgänge dürfen nur dann sichtbar (und damit verwendbar) sein, wenn die Türen offen sind.

Prinzipiell ist das Spiel flexibel genug, um die verschiedensten Türen zu erzeugen. Um die Erstellung der Türen zu vereinheitlichen, habe ich meine Türen auf die folgenden Zustände beschränkt:

- `ST_TUER_VERSCHLOSSEN`, die Tür ist verschlossen und kann erst geöffnet werden, wenn sie aufgeschlossen wurde.
- `ST_TUER_GESCHLOSSEN`, die Tür ist geschlossen und kann geöffnet werden.
- `ST_TUER_OFFEN`, die Tür ist offen und kann geschlossen werden.
- `ST_TUER_OFFENBLOCKIERT`, die Tür ist offen, aber blockiert. Sie lässt sich nicht mehr schließen.

Um es nochmal hervorzuheben, ich habe mich auf diese vier Zustände beschränkt, um das Spiel nicht unnötig kompliziert zu machen. Wenn Sie eine eigene Spielwelt erstellen möchten, können Sie jede Tür umsetzen.

Das die Ausgänge bei geschlossener Tür nicht sichtbar sind, aber sichtbar werden, wenn die Tür geöffnet wird, und dann nach dem Schließen der Tür wie-

der verschwinden, muss wie alles in der Spielwelt über Spielregeln definiert werden. Zur Vereinheitlichung habe ich dazu die etwas aufwendigere Hilfsmethode `erzeuge_tuer` programmiert:

```
void SpielBauer::erzeuge_tuer(id_type raum1id, id_type raum2id,
    id_type exit1id, id_type exit2id,
    id_type tuer1id, id_type tuer2id, int z,
    const std::string& name1, const std::string& bez1,
    const std::string& otxt1,
    const std::string& name2, const std::string& bez2,
    const std::string& otxt2,
    const std::string& stxt, const std::string& blocktxt) {

  erzeuge_gegenstand(tuer1id,raum1id,true,false,true,
                     name1, bez1, "", z);
  erzeuge_gegenstand(tuer2id,raum2id,true,false,true,
                     name2, bez2, "", z);
  if(z==ST_TUER_VERSCHLOSSEN || z==ST_TUER_GESCHLOSSEN) {
    get_gegenstaende()->get_gegenstand_by_id(exit1id)->
                        set_sichtbar(false);
    get_gegenstaende()->get_gegenstand_by_id(exit2id)->
                        set_sichtbar(false);
  }
  else {
    get_gegenstaende()->get_gegenstand_by_id(exit1id)->
                        set_sichtbar(true);
    get_gegenstaende()->get_gegenstand_by_id(exit2id)->
                        set_sichtbar(true);
  }

  Spielregel* tmpregel;
  get_regeln()->add_regel(
    tmpregel=new Spielregel{this, CMD_OEFFNE, tuer1id, -1});
  tmpregel->add_bedingung(new BedingungObjektHatZustand{this,
                          tuer1id, ST_TUER_GESCHLOSSEN});
  tmpregel->add_aktion(new AktionTextausgabe{this, otxt1});
  tmpregel->add_aktion(
           new AktionObjektSetSichtbar{this, exit1id, true});
  tmpregel->add_aktion(
           new AktionObjektSetSichtbar{this, exit2id, true});
  if(blocktxt!="") {
    tmpregel->add_aktion(new AktionObjektNeuerZustand{this,
                         tuer1id, ST_TUER_OFFENBLOCKIERT});
    tmpregel->add_aktion(new AktionObjektNeuerZustand{this,
                         tuer2id, ST_TUER_OFFENBLOCKIERT});
  }
  else {
    tmpregel->add_aktion(
  new AktionObjektNeuerZustand{this, tuer1id, ST_TUER_OFFEN});
    tmpregel->add_aktion(
  new AktionObjektNeuerZustand{this, tuer2id, ST_TUER_OFFEN});
  }
```

```
    get_regeln()->add_regel(
      tmpregel=new Spielregel{this, CMD_OEFFNE, tuer2id, -1});
    tmpregel->add_bedingung(new BedingungObjektHatZustand{this,
                              tuer2id, ST_TUER_GESCHLOSSEN});
    tmpregel->add_aktion(new AktionTextausgabe{this, otxt2});
    tmpregel->add_aktion(new AktionObjektSetSichtbar{this,
                                            exit1id, true});
    tmpregel->add_aktion(new AktionObjektSetSichtbar{this,
                                            exit2id, true});
    if(blocktxt!="") {
      tmpregel->add_aktion(new AktionObjektNeuerZustand{this,
                           tuer1id, ST_TUER_OFFENBLOCKIERT});
      tmpregel->add_aktion(new AktionObjektNeuerZustand{this,
                           tuer2id, ST_TUER_OFFENBLOCKIERT});
    }
    else {
      tmpregel->add_aktion(
    new AktionObjektNeuerZustand{this, tuer1id, ST_TUER_OFFEN});
      tmpregel->add_aktion(new AktionObjektNeuerZustand{this,
                                     tuer2id, ST_TUER_OFFEN});
    }

    if(blocktxt!="") {
      get_regeln()->add_regel(tmpregel=new Spielregel{this,
                                  CMD_SCHLIESSE, tuer1id, -1});
      tmpregel->add_trigger(CMD_SCHLIESSE, tuer2id, -1);
      tmpregel->add_bedingung(new BedingungObjektHatZustand{this,
                             tuer1id, ST_TUER_OFFENBLOCKIERT});
      tmpregel->add_aktion(new AktionTextausgabe{this, blocktxt});
    }
    else {
      get_regeln()->add_regel(tmpregel=new Spielregel{this,
                                    CMD_SCHLIESSE, tuer1id, -1});
      tmpregel->add_trigger(CMD_SCHLIESSE, tuer2id, -1);
      tmpregel->add_bedingung(new BedingungObjektHatZustand{this,
                                       tuer1id, ST_TUER_OFFEN});
      tmpregel->add_aktion(new AktionObjektNeuerZustand{this,
                                  tuer1id, ST_TUER_GESCHLOSSEN});
      tmpregel->add_aktion(new AktionObjektNeuerZustand{this,
                                  tuer2id, ST_TUER_GESCHLOSSEN});
      tmpregel->add_aktion(
               new AktionObjektSetSichtbar{this, exit1id, false});
      tmpregel->add_aktion(
               new AktionObjektSetSichtbar{this, exit2id, false});
      tmpregel->add_aktion(new AktionTextausgabe{this, stxt});
    }
  }
```

Definitiv ist das die längste Methode im Buch. Zuerst werden die beiden Türen erstellt.

Dann wird geprüft, ob die Türen offen oder geschlossen sind, dementsprechend werden die Ausgänge auf sichtbar oder unsichtbar gesetzt.

Im nächsten Schritt wird für die erste Tür das Öffnen als Spielregel formuliert. Geöffnet werden kann eine Tür nur dann, wenn sie geschlossen ist. Beim Öffnen werden die Ausgänge sichtbar gemacht und je nachdem, ob die Tür danach blockiert sein soll, der Zustand der Türen auf `ST_TUER_OFFEN` oder `ST_TUER_OFFENBLOCKIERT` gesetzt.

Danach folgt die Spielregel zum Öffnen der zweiten Tür nach demselben Schema.

Im Anschluss wird das Schließen der beiden Türen umgesetzt. Sollten die Türen blockiert sein, wird nur Text ausgegeben. Andernfalls werden die Türen geschlossen und die Ausgänge auf unsichtbar gesetzt.

Die Tür vom Flur zur Toilette wird so erzeugt:

```
erzeuge_tuer(100,600,100100,100110, 200000, 200010,
  ST_TUER_GESCHLOSSEN,
  "klotür", "eine Klotür", "Du öffnest die Tür und siehst…",
  "tür","eine Tür", "Der Weg zum Flur ist wieder frei.",
  "Die Klotür ist nun wieder geschlossen.","");
```

21.4.5 Erstellen der Interaktionen

Das Erstellen der Interaktionen läuft über die Erzeugung von Spielregeln. Für die meisten Befehle habe ich Hilfsmethoden programmiert. Im Folgenden sind einige der Interaktionen mit dem Haarknäuel aufgeführt:

```
r=erzeuge_nehmen(102, -1, true, "Das fängt ja gut an…"); {
    r->add_bedingung(
            new BedingungObjektNichtInInventar{this, 102});
  }
erzeuge_untersuchung(102, -1, POS_INVENTAR, "Die Haare seh… ");
erzeuge_untersuchung(102, -1, POS_RAUM, "Aus der Entfernun… ");
erzeuge_benutzung(102, -1, POS_RAUM, "Zu weit weg.");
erzeuge_benutzung(102, -1, POS_INVENTAR, "Falls du lichtes…");
```

Exemplarisch zeige ich hier noch die Methode `erzeuge_nehmen`:

```
Spielregel* SpielBauer::erzeuge_nehmen(id_type id, int z,
                  bool ins_inv, const std::string& text) {
  Spielregel* tmpregel;
  get_regeln()->add_regel(
            tmpregel=new Spielregel{this, CMD_NIMM, id, -1});
  if(z!=-1)
    tmpregel->add_bedingung(
                new BedingungObjektHatZustand{this, id, z});
  if(ins_inv)
    tmpregel->add_aktion(
                      new AktionObjektNeuePos{this, id, 0});
```

```
    if(text!="")
      tmpregel->add_aktion(new AktionTextausgabe{this, text});

    return tmpregel;
  }
```

Zuerst wird eine neue Spielregel erstellt und der Trigger hinzugefügt. Sollte der Befehl nur möglich sein, wenn der Gegenstand in einem bestimmten Zustand ist, dann wird dies als zusätzliche Bedingung hinzugefügt.

Wenn der Gegenstand in das Inventar übertragen werden soll, dann wird die notwendige Aktion zur Spielregel hinzugefügt. Soll das Nehmen des Gegenstands mit einem Text quittiert werden, dann wird eine Textausgabe als Aktion hinzugefügt.

Die Methode liefert die Adresse der Spielregel zurück, falls noch weitere Bedingungen oder Aktionen hinzugefügt werden müssen (wie im oberen Beispiel).