Spiele-KI mit Python programmieren

DIE LESEPROBE

Kapitel 3
Hi! I'm Eliza

Eliza zeigte der Welt, dass künstliche Intelligenz nicht nur rechnet, sondern auch zuhören kann – auch wenn sie nicht versteht.

Die abgenutzte Oberfläche des Bildschirms flackert, als Elizas Antwort in der »TheraChat«-Anwendung eintrifft. Ihre digitalen Worte sind wie immer präzise, fast mechanisch, und in der Art, wie sie auf Ihre Äußerung reagiert, liegt eine subtile Logik, eine Form der Mustererkennung. »Sie erwähnten ›Angst vor den Drohnen‹. Erzählen Sie mir mehr darüber.« Das ist ihre Standardreaktion auf bestimmte Schlüsselwörter im Chat, eine einfache, aber effektive Methode, um das Gespräch in eine für sie vorhersehbare Richtung zu lenken. Da rechtliche Vorgaben die Kommunikation über sensible Gesundheitsthemen stark reglementieren, ist Elizas Fähigkeit, bestimmte Phrasen und Muster zu identifizieren, von entscheidender Bedeutung für ihre Funktion als psychotherapeutischer Android …

Willkommen in einer der faszinierendsten Epochen der frühen künstlichen Intelligenz: der Welt von *Eliza*. In diesem Kapitel geht es zurück in die 1960er-Jahre, als Joseph Weizenbaum mit Eliza eine der ersten sprachverarbeitenden Maschinen entwickelte. Dieses Programm war einfach, aber revolutionär – eine Simulation eines Gesprächs, die vorgab, die Rolle eines einfühlsamen Therapeuten zu übernehmen.

Eliza war nicht nur eine technische Innovation, sondern auch eine kulturelle Offenbarung. Sie zeigte, wie leicht Menschen dazu neigen, selbst einfache Antworten als Zeichen von Intelligenz und Verständnis wahrzunehmen. Mit ihren Mustererkennungsalgorithmen und einer regelbasierten Struktur legte sie den Grundstein für viele spätere Entwicklungen in der natürlichen Sprachverarbeitung – von Chatbots bis hin zu modernen Sprachassistenten wie Alexa und Siri.

Chatbots

Ein *Chatbot* ist ein Computerprogramm, das menschliche Gespräche mit einem Endbenutzer simuliert. Sie finden Chatbots auf Webseiten und beispielsweise Messaging-Apps. Chatbots können über Text oder Sprache interagieren und werden oft verwendet, um Kundenanfragen zu beantworten.

In diesem Kapitel werde ich die Funktionsweise von Eliza erklären und zeigen, wie Sie einen solchen Bot selbst implementieren. Dabei werden Sie die Grenzen und Möglichkeiten von *regelbasierter künstlicher Intelligenz* erkennen und werfen einen Blick darauf, wie ein vergleichsweise simples System eine Illusion von Intelligenz schaffen konnte.

Es ist eine Gelegenheit, Geschichte nachzuerleben und die Pioniere der KI zu würdigen, während Sie selbst erleben, wie aus Regeln und Mustern eine »sprechende Maschine« entsteht. Bereiten Sie sich darauf vor, Eliza neu zu erschaffen – und vielleicht einen Teil der Magie, die Menschen vor Jahrzehnten gespürt haben, selbst zu erleben.

Wegweiser durch das Kapitel

1. Was ist Eliza?
2. Was ist ein Turing-Test?
3. Woher kommt der Name Eliza?
4. Das MVC-Entwurfsmuster
5. Ablauflogik von Eliza
6. Implementierung von Eliza
7. Das Hauptprogramm
8. Der Startbildschirm
9. Laden und Speichern von Dateien
10. Hallo und Tschüss
11. Texteingabe und -analyse

3.1 Was ist Eliza?

Eliza war eine der ersten Chatbot-Implementierungen (*https://www.chatbotpack.com/a-history-of-chatbots/*) und wurde von Joseph Weizenbaum (1923–2008) Mitte der 1960er-Jahre entwickelt. Weizenbaum war Informatiker und gleichzeitig Gesellschafts- und Wissenschaftskritiker. Von 1963 an lehrte er als Professor am Massachusetts Institute of Technology (MIT), an dem er Eliza entwickelte. Sich selbst bezeichnet Weizenbaum auch als »Ketzer der Informatik«.

Die Originalversion war in MAD-Slip programmiert, einer Listenverarbeitungserweiterung der Programmiersprache MAD, wurde aber in viele Sprachen portiert. Am bekanntesten ist sie in Kombination mit dem DOCTOR-Skript, das einen Psychotherapeuten simuliert. Weizenbaums Originalartikel über Eliza (*https://dl.acm.org/citation.cfm?id=365168*) ist verfügbar und kann käuflich erworben werden [1]. Wenn Sie sich für Eliza

selbst und die Historie sowie den Einfluss von Eliza auf Computerkultur und KI-Forschung interessieren, kann ich Ihnen die Lektüre von [8] ans Herz legen.

Eliza war als eine Art Simulation der Möglichkeiten der natürlichsprachigen Kommunikation zwischen Mensch und Maschine gedacht. Weizenbaum wollte mit dem Computerprogramm eigentlich aufzeigen, wie oberflächlich und naiv die Kommunikation und der Umgang zwischen Mensch und Maschine ist. Eliza arbeitet skriptbasiert in englischer Sprache und antwortet auf Texteingaben des Benutzers. Dabei lässt das Computerprogramm den Benutzer glauben, dass er ein Gespräch mit einem echten Menschen führt. Den Turing-Test konnte das Programm aber nicht bestehen, auch wenn sich viele Menschen von Eliza täuschen ließen, wie angeblich die Sekretärin von Weizenbaum.

Bei der Recherche im Internet bin ich auf das Java-Applet von Charles Hayden gestoßen, das nach seiner Aussage »eine vollständige und getreue Implementierung« darstellt. Ich habe mich am Aufbau der Skriptdatei und auch an seinem Quelltext [2] orientiert, um die Implementierung zu erstellen, die diesem Buch zugrunde liegt. Charles Hayden gibt auf seiner Webseite an: »You are welcome to make use of it however you want.« – insofern ist der Quelltext als frei zu betrachten.

3.2 Was ist ein Turing-Test?

Alan Turing formulierte 1950 ein Vorgehen, um festzustellen, ob ein Computer ein dem Menschen gleichwertiges Denkvermögen hat. Diese Idee wurde später als *Turing-Test* bezeichnet, auch wenn seine ursprüngliche Bezeichnung eigentlich *Imitation Game* war. Der Test wurde nach Turings Tod 1954 in seiner Komplexität reduziert und ging in die Informatik ein, nachdem die künstliche Intelligenz zu einem eigenständigen akademischen Fachgebiet geworden war.

Alan Turing (1912–1954)

Alan Turing war ein bahnbrechender britischer Mathematiker, Logiker und Informatiker, dessen Werk das Fundament der modernen Computerwissenschaft und künstlichen Intelligenz legte. Folgende Daten und Entwicklungen sollten Sie kennen:

- **1936 – Turingmaschine und das Halteproblem:** Turing revolutionierte die Theorie der Berechenbarkeit mit der Konzeptualisierung der *Turingmaschine*, einem abstrakten Modell eines Computers. Dies nutzte er, um das *Halteproblem* – die Frage, ob es einen Algorithmus gibt, der für jede mathematische Aussage entscheiden kann, ob sie wahr oder falsch ist – als unlösbar zu beweisen, was eine zentrale Erkenntnis der theoretischen Informatik ist.

- **Zweiter Weltkrieg – Knacken der Enigma**: Während des Zweiten Weltkriegs spielte Turing eine entscheidende Rolle in Bletchley Park, wo er maßgeblich an der Entwicklung von Methoden und Maschinen zum Entschlüsseln der deutschen Enigma-Codes beteiligt war. Seine Arbeit trug entscheidend zum Sieg der Alliierten bei.
- **1950 – Turing-Test:** Er führte den Turing-Test ein, ein wegweisendes Experiment zur Bestimmung, ob eine Maschine intelligentes Verhalten zeigen kann, das von menschlichem Verhalten nicht zu unterscheiden ist. Dies wurde ein zentraler Begriff in der KI-Forschung.
- **1952–1954 – Verfolgung & Suizid:** Aufgrund seiner Homosexualität, die zu dieser Zeit in Großbritannien strafbar war, wurde Turing wegen »grober Unzucht« angeklagt und zur chemischen Kastration gezwungen. Er starb 1954 im Alter von 41 Jahren durch Suizid, höchstwahrscheinlich infolge der erlittenen Demütigung und der chemischen Behandlung.

Sein Genie legte den Grundstein für unsere digitale Welt, sein Schicksal ist eine Mahnung an vergangene Ungerechtigkeiten.

Turing diskutiert in seinem Test die Frage »Können Maschinen denken?« und baut um diesen Test herum ein Spiel, das er *Imitation Game* genannt hat. Abbildung 3.1 zeigt den Aufbau der Testumgebung. Der Befragende (C) in der Mitte kommuniziert schriftlich mit einem Mann (A) und einer Frau (B) und soll herausfinden, welcher von beiden der Mann bzw. die Frau ist. A soll mit seinen Antworten die Versuche des Befragenden so sabotieren, dass dieser diese Entscheidung nicht sicher treffen kann. B als Frau soll dagegen dem Befragenden C in ihren Antworten helfen, die Identitäten von A und von B in Bezug auf Mann und Frau zu klären. Die Maschine nimmt im Test die Position von A ein und versucht aktiv, mit ihren Antworten den Befrager C im Unklaren darüber zu lassen, wer von beiden der Mann und wer die Frau ist. Die Maschine ist erfolgreich und besteht den Test, wenn es in Bezug auf die Häufigkeit richtiger Annahmen von C keinen messbaren Unterschied zu Situationen gibt, in denen die Position A von einem Menschen übernommen wird.

Für Alan Turing ging es bei maschineller Intelligenz und »Denken« weit über die reine Nachahmung menschlicher Kommunikation hinaus. Sein Verständnis war viel umfassender: Eine Maschine sollte nicht nur Gespräche täuschend echt simulieren können, sondern beispielsweise in der Lage sein, einen Moderator erfolgreich darüber im Unklaren zu lassen, welches Geschlecht sie imitiert.

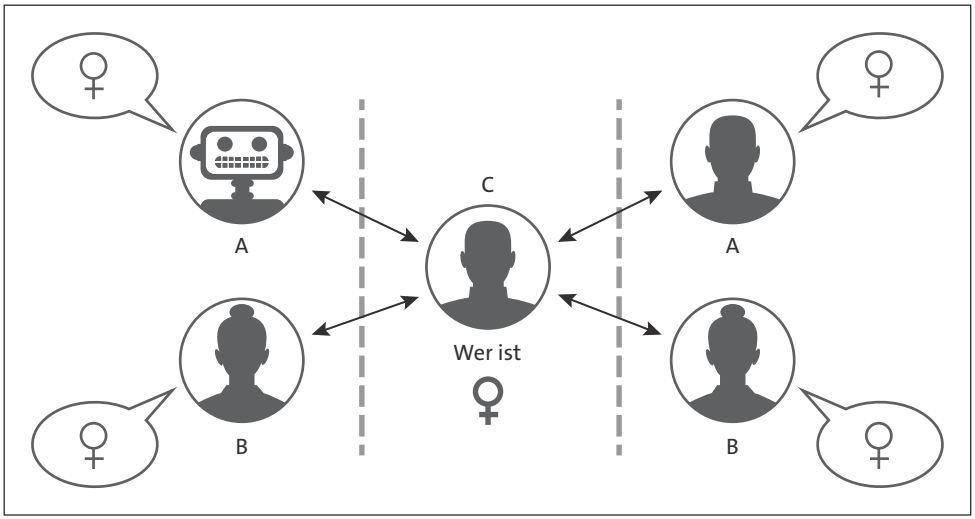

Abbildung 3.1 Turing-Test für das Imitation Game

3.3 Woher kommt der Name Eliza?

Pygmalion, ein Theaterstück von George Bernard Shaw, erzählt die Geschichte von Professor Henry Higgins, einem Sprachwissenschaftler, der die einfache Blumenverkäuferin Eliza Doolittle sprachlich und gesellschaftlich »umformen« will. Doch Eliza entwickelt eine eigene Identität und stellt die Machtverhältnisse infrage. Die Transformation betrifft nicht nur sie, sondern auch Higgins selbst [6].

Wieso heißt Shaws Theaterstück »Pygmalion«?

Der Titel des Stücks verweist auf einen antiken Mythos, in dem sich der Bildhauer Pygmalion in eine Statue verliebt, die er geschaffen hat – eine Metapher für Kontrolle, Projektion und Wandel.

Joseph Weizenbaum wählte daher den Namen »Eliza« für sein Programm: Wie Eliza Doolittle passt sich das Programm scheinbar dem Sprachstil seines Gegenübers an, um Gespräche im Stil eines Rogerianischen Therapeuten zu simulieren. Es analysierte Benutzereingaben und reagierte reflektierend – so, als würde es zuhören und verstehen.

Eliza zeigt, wie stark Menschen dazu neigen, Maschinen menschliche Eigenschaften zuzuschreiben – und wie Sprache unsere Beziehung zur Technik prägt. Während Shaws Pygmalion die soziale Formbarkeit des Menschen thematisiert, demonstriert Eliza, wie weit Maschinen Kommunikation imitieren können – und wo ihre Grenzen liegen.

Rogerianische Therapie

Die *Rogerianische Therapie*, entwickelt von Carl Rogers, beruht auf dem Prinzip, dass Klientinnen und Klienten selbstbestimmt und lösungsorientiert arbeiten – die Aufgabe des Therapeuten ist es, ein wertschätzendes Umfeld zu schaffen. Weizenbaums Eliza setzt diese Idee technologisch um, indem sie Gespräche spiegelt und so eine Illusion von Verstehen erzeugt.

Beide Werke, Pygmalion und Eliza, stellen zentrale Fragen: Wie formbar ist Identität durch Sprache? Wer kontrolliert wen: der Schöpfer das Geschaffene oder umgekehrt?

3.4 Das MVC-Entwurfsmuster

Ein kluger Mensch sagte einmal: »Ein Programm wird nie fertig. Irgendwann hört man einfach auf, daran zu arbeiten.« Meiner Meinung nach unterscheidet Software sich in diesem Punkt von anderen Artefakten: Ein Bildhauer verkauft seine Skulptur, und niemand kommt zurück, um nach Änderungen zu fragen. Bei Programmen ist das anders: Kaum ist das Projekt abgeschlossen, folgen Änderungswünsche, und das oft, ohne dass der Kunde bereit ist, einen Aufpreis für die zusätzliche Leistung zu bezahlen.

Gerade deshalb lohnt es sich, gut strukturierten und wartbaren Code zu schreiben. Code, den Sie auch nach längerer Pause noch verstehen und weiterentwickeln können. Daher möchte ich Ihnen das Konzept der *Entwurfsmuster* (engl. *Design Patterns*) vorstellen, insbesondere das Entwurfsmuster *Model View Control (MVC)*. Es hilft Ihnen dabei, Anwendungen mit grafischer Benutzeroberfläche und Datenhaltung in klar getrennte Schichten zu gliedern. Unsere Eliza wird damit aus mehreren Dateien bestehen (auch *Module* genannt), die unterschiedliche Aufgaben übernehmen und logisch voneinander getrennt sind (siehe Abbildung 3.2).

Entwurfsmuster – bewährte Lösungen für wiederkehrende Probleme

Entwurfsmuster sind bewährte, generische Lösungsansätze für wiederkehrende Probleme im Softwaredesign. Sie sind keine fertigen Bibliotheken oder Frameworks, die man einfach importiert, sondern vielmehr Vorlagen oder Schablonen, die beschreiben, wie man bestimmte Probleme strukturiert lösen kann. Sie bieten eine gemeinsame Sprache für Entwickler und erleichtern das Verständnis, die Kommunikation und die Wartbarkeit von Code.

Im Laufe der Softwareentwicklung stellten erfahrene Entwickler fest, dass bestimmte Problematiken immer wieder auftauchten und dass sie dafür ähnliche Lösungswege

wählten. Anstatt das Rad jedes Mal neu zu erfinden und fehleranfällige individuelle Lösungen zu entwickeln, entstand der Wunsch, diese erfolgreichen Ansätze zu dokumentieren, zu benennen und zugänglich zu machen. Entwurfsmuster fördern somit die Wiederverwendbarkeit von Wissen (und Code), Flexibilität, Wartung und Robustheit.

Geprägt wurde der Begriff durch das 1994 erschienene Buch »Entwurfsmuster« [3] [4]. Die vier Autoren – Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides – sind heute als die »Gang of Four« (GoF) bekannt. Ihr Buch systematisierte 23 klassische Entwurfsmuster, die nach wie vor aktuell sind. Das MVC-Muster ist etwas moderner und findet sich nicht in diesem Buch – dafür aber in einem weiteren Buch, das ich Ihnen gerne empfehlen möchte [5].

Das Model-View-Control-Muster zählt zu den komplexeren, aber auch wirkungsvollsten Entwurfsmustern, und wir werden es Schritt für Schritt anhand von Eliza gemeinsam umsetzen. Eliza wird in drei Schichten gegliedert: View (V), Controller (C) und Model (M).

Der *View* ist die Präsentationsebene. Hier findet die direkte Interaktion mit dem Benutzer statt. Dabei kann es sich um die Ausgabe auf der Konsole handeln, um eine grafische Benutzeroberfläche wie unter Windows üblich oder aber auch um eine akustische Ausgabe. Der View zeigt Inhalte nur an, führt aber keine Berechnungen oder Speicheroperationen durch. Die `print()`-Ausgaben unserer Eliza sind im Moment unsere View-Schicht.

Die *Controller*-Schicht verarbeitet Eingaben des Nutzers, egal ob per Tastatur, Maus oder Mikrofon. Die Controller-Schicht bereitet die Eingabedaten auf, leitet sie an das Model weiter und übermittelt Ergebnisse an den View. Die Controller-Schicht ist also das Bindeglied zwischen Darstellung und Logik.

Das *Model* ist das »Gehirn« der Anwendung. Es enthält die Datenhaltung und Geschäftslogik, etwa Variablen, Antwortregeln, Logik zur Gesprächsführung oder Schnittstellen zur Datei- oder Datenbankanbindung.

Durch diese saubere Trennung bleiben Anwendungen übersichtlich, modular und gut wartbar. Eliza als kleines Projekt ist dafür die ideale Trainingsumgebung.

Vielleicht erkennen Sie nun, warum es sinnvoller ist, den Code nicht in einer einzigen Datei zu bündeln. Sobald wir Elizas Daten z. B. in einer Textdatei oder später in einer Datenbank ablegen, bleibt diese Schicht unabhängig vom restlichen Programm. Sie können etwa die englische Skriptdatei durch eine deutsche ersetzen – und Eliza wird auf Deutsch antworten. Zugegeben: nicht perfekt, denn Englisch ist grammatikalisch einfacher zu parsen und zu formulieren – aber prinzipiell ist es möglich.

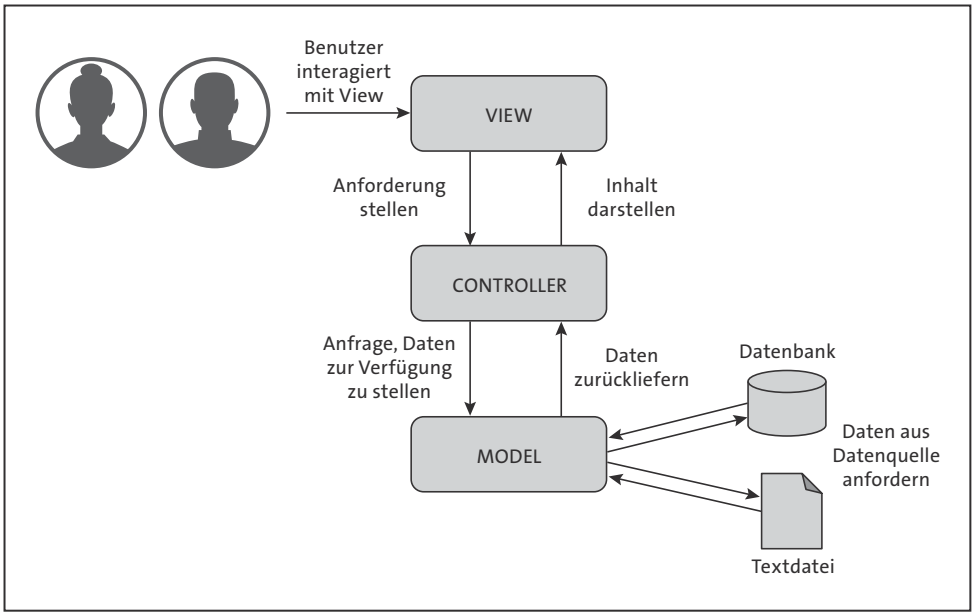

Abbildung 3.2 Das MVC-Entwurfsmuster

Ein weiteres Beispiel: In der Konsolenversion geben wir Elizas Antworten per `print()` aus. Wenn Sie das MVC-Muster nutzen, lassen sich dieselben Antworten später bequem über eine grafische Oberfläche mit Pygame darstellen – ganz ohne Änderungen an der Model-Schicht. Der Controller sorgt lediglich dafür, dass die Ausgaben von der Konsole zu grafischen Elementen wie Textfeldern weitergeleitet werden.

Eine Datenbanklösung halte ich für unser Projekt für überdimensioniert, wir werden uns stattdessen auf eine Textdatei im JSON-Format beschränken, aus der wir Elizas »Gedanken« beim Programmstart in Objekte laden. Dazu benötigen wir ein Konzept namens *Serialisierung*. Die genaue Umsetzung für die Serialisierung des Models und damit Elizas »Gedanken« erläutere ich Ihnen einem späteren Abschnitt ausführlich.

3.5 Ablauflogik von Eliza

Das Programm von Eliza folgt strikt dem EVA-Prinzip. Sie werden durch das Programm begrüßt, und es wird Ihnen eine Frage gestellt. Auf diese Frage können Sie im Sinne der Eingabe eine Antwort formulieren und diese abschicken (E). Die Antwort wird durch

Eliza ausgewertet (V). Im Anschluss daran erhalten Sie die Ausgabe (A). Das Ganze wird in einer Schleife wiederholt, bis Sie das Programm durch Eingabe von »quit« beenden. Abbildung 3.3 zeigt eine Beispielsitzung mit *Eliza*.

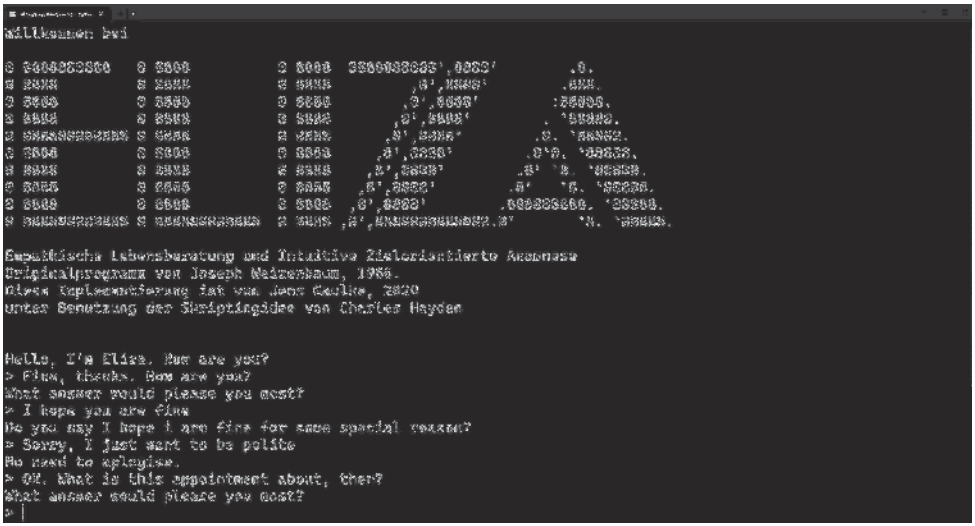

Abbildung 3.3 Programmsitzung mit Eliza

Bevor wir mit der Implementierung beginnen, werfen wir einen Blick auf den Programmablaufplan von Eliza. Dieser bietet einen Überblick über die zentralen Programmschritte: zunächst grob, dann zunehmend detailliert, bevor wir die Teile einzeln umsetzen.

Ein *Programmablaufplan* (siehe Abbildung 3.4) gibt die wichtigsten Stationen im Lebenszyklus des Programms wieder. Besonders relevant sind dabei das Laden der Skriptdatei, die Auswertung der Benutzereingabe und die Antwortgenerierung. Andere Schritte lassen sich mit wenigen Codezeilen umsetzen. Vielleicht fragen Sie sich, warum das Laden der Skriptdatei oder die Ausgabe überhaupt als eigene Programmpunkte aufgeführt sind – schließlich könnte man Elizas Antworten auch direkt in den Quellcode schreiben und mit `print()` ausgeben.

Warum das keine gute Idee ist, klären wir im nächsten Abschnitt – dort setzen wir das MVC-Entwurfsmuster um. Anstatt Eliza als unstrukturierten *Monolith* zu bauen, nähern wir uns der Aufgabe mit einem durchdachten, modularen Architekturansatz.

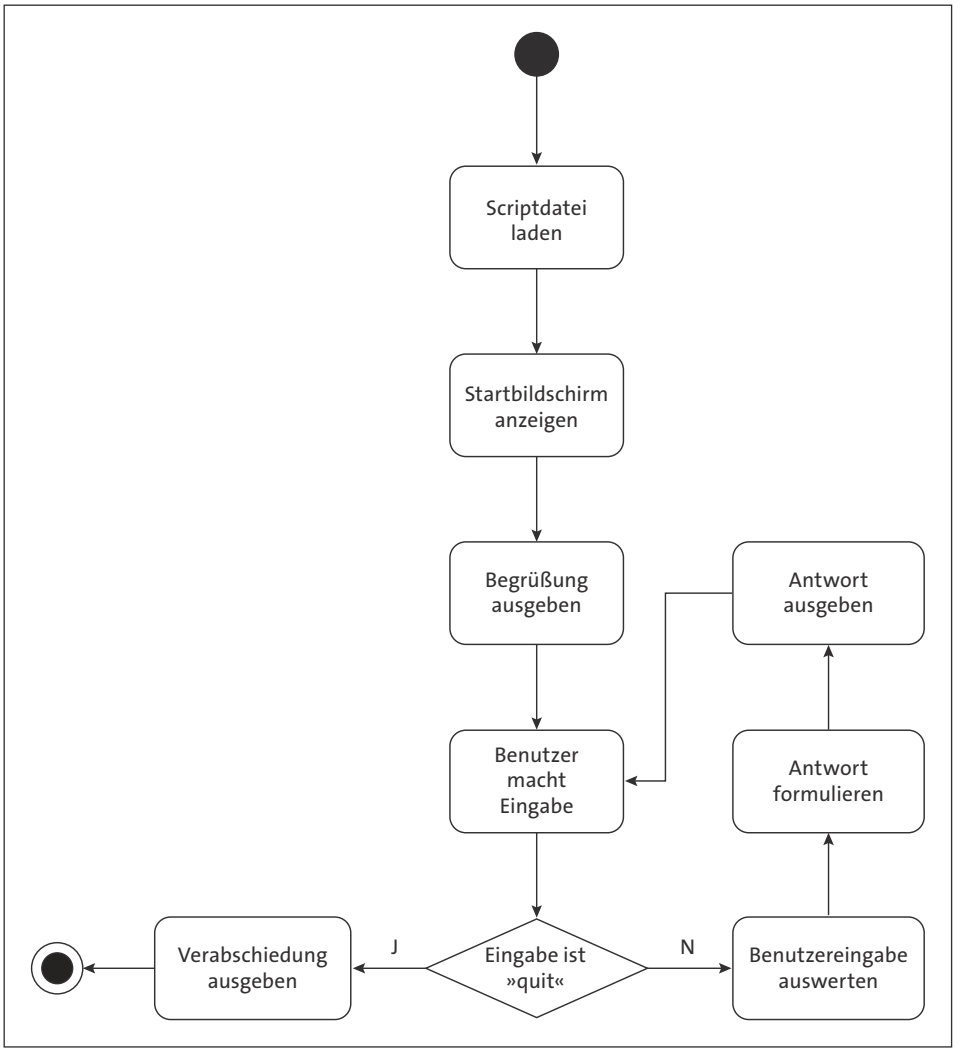

Abbildung 3.4 Programmablaufplan, erster Entwurf

3.6 Implementierung von Eliza

Die *Implementierung* orientiert sich am gerade gezeigten Programmablaufplan von oben nach unten, Schritt für Schritt. So behalten wir den Überblick und setzen die zuvor geplanten Strukturen systematisch um. Unter Implementierung versteht man die konkrete Realisierung geplanter Abläufe. In unserem Fall heißt das: Wir schreiben den Code.

3.6.1 Das Hauptprogramm

Das *Hauptprogramm* in Listing 3.1 bildet den Einstiegspunkt der Anwendung. Hier werden alle Funktionen und Module miteinander verbunden, notwendige Datenstrukturen initialisiert und Eliza zum Leben erweckt. Der Ablauf ist bewusst linear und übersichtlich gehalten. Der Quelltext ist objektorientiert aufgebaut und nutzt eine zentrale Klasse namens `Game` – ein Konzept, das ich in Pygame-Projekten grundsätzlich verwende. Auch wenn diese Version von Eliza konsolenbasiert ist und noch ohne Pygame auskommt, bleibt die Grundstruktur erhalten, damit ein späterer Umstieg auf eine grafische Variante nahtlos möglich ist.

```
class Game():
  def __init__(self):
    pass

  def main(self):
    eliza = Doctor() # Eliza-KI erzeugen
    eliza.load() # Eliza lädt ihre Daten
    print(eliza.mainscreen()) # Begrüßungsformel ausgeben
    print(eliza.initial())

    while True:
      answer = input("> ") # User gibt einen Satz ein
      output = eliza.respond(answer) # Eliza analysiert und antwortet

      if output is None:
        break

      print(output) # Eliza gibt eine Antwort

    print(eliza.final()) # Abschiedsformel ausgeben
    print()

# Call "python eliza.py" from console
if __name__ == "__main__":
  myGame = Game()
  myGame.main()
```

Listing 3.1 Elizas Hauptprogramm

Beim Start des Programms über den Skriptnamen wird eine *Instanz* der *Klasse* `Game` erzeugt und die *Methode* `main()` aufgerufen. Dort wird die Struktur von Eliza initialisiert. Ein Objekt der Klasse `Doctor` wird erstellt und der Variablen `eliza` zugewiesen. Anschließend folgt die Begrüßung, und das Programm tritt in eine Endlosschleife ein, die so lange läuft, bis der Benutzer »quit« eingibt. In diesem Fall bleibt die Antwort von Eliza leer und die Schleife wird per `break` verlassen. Falls Ihnen der Begriff Methode neu ist: In der objektorientierten Welt spricht man nicht von *Funktionen* (wie in C, C++) oder *Prozeduren* (wie in Pascal), sondern von Methoden. Methoden werden fachspezifisch den einzelnen Klassen zugeordnet.

3.6.2 Der Startbildschirm

Konsolenanwendungen wirken immer etwas »nackig«, wenn man sie startet. Ein Cursor blinkt Sie freundlich an, manchmal bekommen Sie auch noch ein Startmenü präsentiert, aus dem Sie einen Punkt auswählen können. Vielleicht war das der Grund, warum Programmierer irgendwann angefangen haben, die Startbildschirme etwas »aufzuhübschen«. Grafische Ausgaben gab es damals (in den 1980er-Jahren) noch nicht, so mussten die Startbildschirme in ASCII-Zeichen gestaltet werden – die ASCII-Art war geboren.

Wurde die ASCII-Art früher noch in mühevoller Handarbeit zusammengestellt, gibt es heute Webseiten, die die Erstellung für Sie übernehmen können. Zur Erstellung des Eliza-Logos habe ich die Seite *https://manytools.org/hacker-tools/ascii-banner/* verwendet. Als Zeichensatz habe ich »Broadway« benutzt, und das Eingabewort war »ELIZA«. Die Webseite gibt ein Text-Logo zurück, das Sie markieren, kopieren und in den Quelltext einfügen können [7]. Aufgerufen wird das Erstellen des Startbildschirms in der Methode `main()` der Klasse `Game`:

```
def main(self):
  eliza = Doctor()           # Eliza-KI erzeugen
...
  print(eliza.mainscreen())  # Begrüßungsformel ausgeben
  print(eliza.initial())
...
```

Listing 3.2 Elizas Begrüßungsbildschirm

An dieser Stelle wird in der Klasse `Game` eine Instanz der Klasse `Doctor` erzeugt, die ich nun Stück für Stück mit Ihnen entwickeln möchte. `Doctor` ist sozusagen »Elizas Kern«.

```
class Doctor:
  def __init__(self):
    self.doctor = {}
  ...

  def clear(self):
    """ Diese Methode löscht den Bildschirm. """
    # Einsatz von Windows
    if name == 'nt':
      _ = system('cls')
    # für Mac und Linux (here, os.name is 'posix')
    else:
      _ = system('clear')

  def mainscreen(self):
    """ Diese Methode stellt den Begrüßungsbildschirm dar """
    # screen art: https://manytools.org/hacker-tools/ascii-banner/
    # schriftart: Broadway
    self.clear()
    return("""Willkommen bei

8 8888888888   8 8888          8 8888  8888888888',8888'        .8.
8 8888         8 8888          8 8888         ,8',8888'        .888.
8 8888         8 8888          8 8888        ,8',8888'        :88888.
8 8888         8 8888          8 8888       ,8',8888'        . `88888.
8 888888888888 8 8888          8 8888      ,8',8888'        .8. `88888.
8 8888         8 8888          8 8888     ,8',8888'        .8`8. `88888.
8 8888         8 8888          8 8888    ,8',8888'        .8' `8. `88888.
8 8888         8 8888          8 8888   ,8',8888'        .8'   `8. `88888.
8 8888         8 8888          8 8888  ,8',8888'        .888888888. `88888.
8 888888888888 8 888888888888  8 8888 ,8',8888888888888.8'       `8. `88888.


Empathische Lebensberatung und Intuitive Zielorientierte Anamnese
Originalprogramm von Joseph Weizenbaum, 1966.
Diese Implementierung ist von Jens Gaulke, 2024
unter Benutzung der Skriptingidee von Charles Hayden

""")
```

Listing 3.3 Das Grundgerüst des Doctor-Scripts

Kommentare und DocStrings

Erinnern Sie sich an den (meinen und hoffentlich auch Ihren) Wunsch, dass ein Programm auch nach längerer Pause noch verständlich und wartbar sein sollte? Dann ist jetzt der richtige Moment, um über Kommentare und Docstrings zu sprechen.

Kommentare sind wie kurze Memos an Sie selbst. Sie beantwortem das »Warum« hinter einem Stück Code und ersparen Ihnen später mühsames Rätselraten. Wer kommentiert, versteht nicht nur besser, was er tut, er hilft sich selbst und anderen, den Code später schneller zu durchdringen und zielgerichtet weiterzuentwickeln.

Gerade wenn Fehler auftreten oder neue Funktionen hinzugefügt werden, sind gut platzierte Kommentare zu Logik, Abhängigkeiten oder möglichen Fallstricken äußerst hilfreich. Sie helfen dabei, die Ursache eines Problems schneller zu identifizieren oder die Auswirkungen einer Änderung besser abzuschätzen – besonders in Teamprojekten oder beim Arbeiten mit fremdem Code, was bei der Umsetzung größerer Projekte der Normalfall ist.

Im nächsten Schritt sehen wir uns die verschiedenen Kommentartypen an und wie sie sinnvoll zur Dokumentation Ihres Programms beitragen.

Einzeilige Kommentare

Ein einzeiliger Kommentar wird in Python immer mit einem Doppelkreuz (Hashtag) begonnen. Das Ende des Kommentars ist mit dem Ende der Zeile definiert.

```
# Hier wird die Variable "leben" auf den Wert 3 gesetzt
leben = 3
```

Ist dieser Kommentar sinnvoll? Nein. Dass die Variable `leben` auf den Wert `3` gesetzt wird, sieht man im Quelltext. Das muss im Kommentar also nicht noch einmal erwähnt werden. Sinnvoll hingegen wäre es zu erläutern, warum es ausgerechnet der Wert `3` ist:

```
# Gib dem Spieler zu Beginn des Spiels drei Leben
leben = 3
```

Kommentare im Quelltext

Kommentare in Python müssen nicht immer am Anfang der Zeile beginnen, sie können überall beginnen. Nur darf nach einem Kommentar kein Programmcode mehr in derselben Zeile folgen – dieser würde ebenfalls als Kommentar gewertet.

```
leben = 3    # Gib dem Spieler zu Beginn des Spiels drei Leben
```

DocStrings

Ein weiterer Typ von Kommentaren in Python sind sogenannte *DocStrings*. Diese sind eine Eigenheit von Python – und nicht nur für Kommentare gedacht. DocStrings können

Hilfestellungen für Funktionen bereitstellen und auch als mehrzeilige Strings verwendet werden. Definieren Sie eine Funktion bzw. Methode, so wie beispielweise diese hier:

```
def addiere(val1: int, val2: int) -> int:
```

Gehen Sie in die nächste Zeile und drücken Sie die Tastenkombination Strg+⇧+2. Python ergänzt dann den Methodenkopf zu

```
def addiere(val1: int, val2: int) -> int:
    """_summary_

    Args:
        val1 (int): _description_
        val2 (int): _description_

    Returns:
        int: _description_
    """
```

An den Stellen _summary_ und _description_ nehmen Sie bitte Ihre spezifische Dokumentation vor. In meinem Fall soll die Methode zwei Ganzzahlen addieren und eine Ganzzahl als Wert zurückgeben. DocStrings haben zusätzliche Funktionalitäten – die bei »normalen« Strings (also Zeichenketten) nicht vorhanden sind. Einer dieser magischen Werte ist die __doc__-Property (Eigenschaft). Diese beinhaltet den DocString vom Anfang der Funktion oder auch der Klasse. Das heißt, Sie können sich die Dokumentation der Methode über

```
print(addiere.__doc__)
```

ausgeben lassen.

Die Methoden `clear()` und `mainscreen()` sind Ihr erster Anlaufpunkt. Über `clear()` löschen Sie den Bildschirm. Dabei werden betriebssystemspezifische Informationen und Abfragen genutzt, da Linux und Mac andere Aufrufe benutzen als Windows. Thematisch betrachtet, beeinflusst `clear()` die View-Ebene und hätte daher unter strenger Beachtung des MVC-Musters in eine eigene View-Klasse ausgelagert werden müssen. Dort hätte die Methode allerdings völlig allein gestanden, weshalb ich sie – nicht artenrein – in die `Doctor`-Klasse ausgelagert habe.

3.6.3 Laden und Speichern von Daten

In Hinblick auf die *Portabilität* und *Wartbarkeit* des Codes ist es durchaus sinnvoll, die Antworten von Eliza in eine Datei auszulagern. Dadurch können die Antworten und

auch die Reaktion auf Eingaben des Benutzers vom Programm getrennt verwaltet werden. Wenn Eliza beispielsweise keine psychiatrische Hilfe mehr geben und stattdessen bei der Auswahl eines Parfüms unterstützen soll, dann können Sie zukünftig eine neue Skriptdatei mit Antworten für genau diesen Anwendungsfall kreieren.

In unserem einfachen Chatbot nehmen wir die Änderungen »von Hand« vor. Die moderneren LLMs wie ChatGPT, die auf neuronalen Netzen basieren, extrahieren sich das »Wissen« aus Dateien, die man ihnen zur Analyse gibt. Dieser Vorgang heißt in den Zusammenhang *Training*.

Um die passende Datenstruktur für unsere Implementierung von Eliza entwickeln zu können, habe ich mir die Struktur des Originals bzw. die Struktur der Version von Charles Hayden ansehen. Hier sehen Sie einen kleinen Ausschnitt aus Haydens Skriptdatei:

```
initial: How do you do.  Please tell me your problem.
final: Goodbye.  Thank you for talking to me.
quit: bye
quit: goodbye
quit: quit
pre: dont don't
pre: cant can't
...
pre: you're you are
pre: i'm i am
pre: same alike
post: am are
post: myself yourself
post: yourself myself
post: i you
post: you I
synon: belief feel think believe wish
synon: family mother mom father dad sister brother wife children child
...
synon: be am is are was
key: xnone
  decomp: *
    reasmb: I'm not sure I understand you fully.
    reasmb: Please go on.
    reasmb: What does that suggest to you ?
    reasmb: Do you feel strongly about discussing such things ?
```

```
key: sorry
  decomp: *
    reasmb: Please don't apologise.
    reasmb: Apologies are not necessary.
    reasmb: I've told you that apologies are not required.
key: apologise
  decomp: *
    reasmb: goto sorry
```

Listing 3.4 Auszug aus dem Sprachskript

Jeder Eintrag der Datei beginnt mit einem Codewort, gefolgt von einem Doppelpunkt. Hinter dem Doppelpunkt steht eine Beschreibung oder ein weiterer, verschachtelter Ausdruck.

Der erste Eintrag, `initial:`, ist die Meldung, die Eliza beim Start ausgibt. Wird Eliza in dieser Version gestartet, erscheint auf dem Bildschirm: *»How do you do. Please tell me your problem.«* Spannend an dieser Vorgehensweise ist, dass Sie eine weitere Zeile mit `initial:` hinzufügen können und dass das Programm aufgrund seines Algorithmus beim Start eine der verfügbaren Zeilen mit dem Codewort `initial:` zufällig auswählt.

Mit dem Codewort `final` wird bzw. werden die Abschiedsmeldungen von Eliza getaggt. `quit` beinhaltet alle Eingaben, mit denen der Benutzer das Programm beenden kann.

`pre` und `post` nehmen Veränderungen an der Eingabe des Benutzers vor. Der Eintrag `dont don't` wandelt eine potenziell fehlerhafte Eingabe des Benutzers von »dont« in »don't« um. Diese Bereinigung hilft Eliza später bei der Auswertung der Benutzereingabe. Die Einträge in `post` drehen alle Bezüge von »meins« in »deins« und umgekehrt um. Damit kann Eliza bequem Eingaben wie »I love you« in »You love me« umwandeln und als Frage an den Benutzer zurückschießen. Hier fällt auch gleich die Komplexität der deutschen Sprache unangenehm auf, in der eine solche Umwandlung zu einer fehlerhaften Konstruktion führt: »Ich liebe dich« wird zu »du liebe mich«, was nur begrenzt Sinn ergibt. Daher werde ich mich in dieser Implementierung auch auf Englisch beschränken.

`synon` baut eine Liste von *Synonymen* auf, auf die Eliza reagieren kann. Wenn der Benutzer das Wort »family« nutzt, wird Eliza familienspezifische Ausgaben generieren. Dasselbe gilt für »father«, »mother« usw. Eliza behandelt diese Eingaben synonym zu »family« und erweckt so den Eindruck, sie könne die Eingabe verstehen und sinnvolle Antworten geben.

Wirklich interessant aber werden die Teile des Skriptes, die nun folgen:

```
key: xnone
  decomp: *
    reasmb: I'm not sure I understand you fully.
    ...
```

Listing 3.5 Der Schlüssel »xnone«

Es gibt zwar keine Eingabe des Benutzers, die »xnone« lautet, aber Eliza untersucht mit einem *Parser* die Eingabe des Benutzers. Sollte kein Codewort gefunden werden, ersetzt Eliza die Eingabe durch `xnone` und wählt eine der Antworten unter `reasmb` aus. Bitte beachten Sie auch den Stern `*` hinter `decomp`. In Haydens Skriptdatei ist dies ein Platzhalter, den Eliza auf die Eingabe anwendet. Er bedeutet in Haydens Kontext »eine Menge von beliebigen Zeichen«. In unserer Skriptdatei werden diese Platzhalter zeitgemäß durch *reguläre Ausdrücke* ersetzt und dann in der Programmlogik ausgewertet. Aus Haydens Skript

```
key: xnone
  decomp: *
    reasmb: I'm not sure I understand you fully.
    ...
```

wird in der modernen Version:

```
"keys": {
    "xnone": {
      "word": "xnone",
      "weight": 1,
      "decomps": [
        {
          "parts": "(.*)",
          "save": true,
          "reasmbs": [
            "I'm not sure I understand you fully.",
            ...
```

Der Kerngedanke ist folgender: Aus `*` wird `(.*)`, was dem regulären Ausdruck *»ein beliebiges Zeichen, beliebig oft«* entspricht. In `decomp` werden die unterschiedlichen Muster gespeichert, anhand derer Eliza die Benutzereingabe untersucht. Der Eintrag `xnone` ist kein so gutes Beispiel, um reguläre Ausdrücke zu erläutern, aber es ist nunmal der Eintrag, der ausgewählt wird, wenn Eliza kein Muster in der Eingabe gefunden hat und »nicht mehr weiterweiß«.

Anhand des Musters `was` möchte ich Ihnen das gerade Beschriebene noch einmal erläutern. Hier ist der betreffende Teil des Skriptes:

```
key: was 2
  decomp: * was i *
    reasmb: What if you were (2) ?
    reasmb: Do you think you were (2) ?
    reasmb: Were you (2) ?
    reasmb: What would it mean if you were (2) ?
    reasmb: What does (2) suggest to you ?
    reasmb: goto what
  decomp: * i was *
    reasmb: Were you really ?
    reasmb: Why do you tell me you were (2) now ?
    reasmb: Perhaps I already know you were (2) .
  decomp: * was you *
    reasmb: Would you like to believe I was (2) ?
    reasmb: What suggests that I was (2) ?
    reasmb: What do you think ?
    reasmb: Perhaps I was (2) .
    reasmb: What if I had been (2) ?
```

Listing 3.6 Analyse des Schlüssels »was«

In der Originalversion spaltet Eliza die Benutzereingabe in einzelne Wörter auf. Dieser Vorgang wird im Fachjargon *Tokenizing* genannt. Die einzelnen Wörter (*Token*) werden dann unter anderem daraufhin untersucht, ob der Begriff »was« darin auftaucht. Die Eingabe »tomorrow, i will be in berlin« enthält kein »was« und würde keinen Treffer liefern. Die Eingabe »yesterday, i was in bremen« würde einen Treffer liefern, da »was« in der Zeichenkette vorkommt. Auf Basis des Treffers »was« wird die Benutzereingabe weiter untersucht. `* i was *` bedeutet so viel wie: *»die Zeichenkette muss ›i was‹ beinhalten, wobei links davon beliebig viele Zeichen stehen dürfen und rechts davon beliebig viele Zeichen stehen dürfen«*.

`* i was *` liefert also beispielsweise Treffer bei:

- **i was** in berlin
- yesterday**, i was** in berlin
- in berlin yesterday **i was** (yoda-speak)
- **i was i was i was**

Nun bleibt noch die »2« zu klären, die in der Zeile `key: was 2` steht. Diese Ziffer gibt an, wie »wichtig« das Wort ist. Wenn im Skript beispielsweise auch »berlin« vorkommt, muss Eliza eine Entscheidung treffen, ob der Text auf Basis von »was« oder »berlin« untersucht werden soll. Mithilfe dieser Priorisierung können Sie selbst entscheiden, wie sich Eliza verhalten soll.

Die Zahl `(2)` in Klammern in den potenziellen Antworten von Eliza dient der weiteren Nutzung der geparseten Benutzereingabe. Betrachten wir die Eingabe »yesterday**, i was** in berlin«, dann entspricht »yesterday,« dem ersten Stern und »in berlin« dem zweiten Stern, bezogen auf das Suchmuster `* i was *`. Diese beiden Teilsuchergebnisse werden im Rahmen des Parsings durch den regulären Ausdruck gespeichert und können später wiederverwendet werden.

Angenommen, Eliza wählt als Antwort `Why do you tell me you were (2) now ?` aus. Dann wird `(2)` durch den Begriff ersetzt, der beim Parsen dem zweiten Stern zugewiesen wurde, und das ist »in berlin«. Elizas Antwort an den Benutzer lautet damit: »Why do you tell me you were **in berlin** now ?« Das klingt doch sehr empathisch, oder? Eliza gibt dadurch nicht nur eine beliebige Antwort, sondern nimmt auch noch Bezug auf Berlin und stellt eine psychoanalytische Frage.

Jetzt wissen Sie schon, wie die Skriptdatei funktioniert; gespeichert haben wir sie allerdings noch nicht. Wir sind aber schon einen großen Schritt vorangekommen: Denn wenn wir wissen, wie die Datei aufgebaut ist, können wir auch eine Struktur entwickeln, um die Datei für unsere Eliza-Version abzuspeichern, wieder zu laden und zu verstehen, wie die Datei von Hand erweitert werden kann.

Einträge aus der Datei, die einzeilig vorliegen wie beispielsweise `initial`, `quit` oder `final`, können als *Liste* in Python gespeichert werden. Das sind die Zeilen mit den eckigen Klammern, die innerhalb des Klassen-Konstruktors erzeugt werden:

```
class Doctor:
  def __init__(self):
    self.doctor = {}
    self.initials = []
    self.finals = []
    self.quits = []
    self.pres = {}
    self.posts = {}
    self.synons = {}
    self.keys = {}
    self.memory = []
```

Listing 3.7 Die Datenstruktur der Listen im Sprachskript

Die anderen Einträge aus der Skriptdatei, wie beispielsweise die `key`s, werden als Key-Value-Paar (Schlüssel-Wert-Paar) gespeichert. Das wird in Python üblicherweise als *Dictionary* realisiert. Sie erkennen die Dictionarys im Quellcode an den geschweiften Klammern. Die Datenstruktur, die alle weiteren Strukturen in sich oder unter sich vereint, ist `self.doctor`. Die nächsten Zeilen des Quelltextes bewirken die Verschachtelung:

```
self.doctor['initials'] = self.initials
self.doctor['finals'] = self.finals
self.doctor['quits'] = self.quits
self.doctor['pres'] = self.pres
self.doctor['posts'] = self.posts
self.doctor['synons'] = self.synons
self.doctor['keys'] = self.keys
self.doctor['memory'] = self.memory
```

Listing 3.8 Die Datenstruktur der Dictionarys im Sprachskript

Ich habe alle Listen und Dictionarys in die Datenstruktur `self.doctor` ausgelagert. Zudem habe ich dem Konstruktor Testdatensätze hinzugefügt, um sehen zu können, wie die gespeicherte Version aussehen wird. Die Testdatensätze bedienen alle Datenstrukturen, damit wir einen guten Überblick über die resultierende Dateistruktur und das Speicherformat auf der Festplatte haben:

```
self.initials.append("How do you do? Please tell me your problem.")

decomps = Decomp("*",True,[])
decomps.reasmbs.append("In what way ?")
decomps.reasmbs.append("What resemblence do you see ?")
decomps.reasmbs.append("What does that similarity suggest to you ?")
decomps.reasmbs.append("What other connections do you see ?")
decomps.reasmbs.append("What do you suppose that resemblence means ?")
decomps.reasmbs.append("What is the connection, do you suppose ?")
decomps.reasmbs.append("Could here really be some connection ?")
decomps.reasmbs.append("How ?")

decomps1 = Decomp("*",True,[])
decomps1.reasmbs.append("In what way ?")
decomps1.reasmbs.append("What resemblence do you see ?")
decomps1.reasmbs.append("What does that similarity suggest to you ?")
decomps1.reasmbs.append("What other connections do you see ?")
decomps1.reasmbs.append("What do you suppose that resemblence means ?")
```

```
  decomps1.reasmbs.append("What is the connection, do you suppose ?")
  decomps1.reasmbs.append("Could here really be some connection ?")
  decomps1.reasmbs.append("How ?")

  key=Key('alike', 10, [])
  key.decomps.append(decomps)
  key.decomps.append(decomps1)

  self.keys['alike'] = key
```

Listing 3.9 Füllen der Datenstruktur mit Testdaten

In Listing 3.9 werden zwei Listen der Struktur `Decomp` erzeugt und mit `reasmbs`-Einträgen gefüllt. Im Anschluss daran werden die beiden `Decomp`-Listen der Struktur `Key` hinzugefügt. Da `Decomp` und `Key` keine einfachen Listen sind, sondern ihrerseits verschachtelte Datenstrukturen darstellen, habe ich diese beiden Strukturen als eigenständige Klassen angelegt:

```
class Key:
  def __init__(self, word, weight, decomps):
    self.word = word
    self.weight = weight
    self.decomps = decomps

class Decomp:
  def __init__(self, parts, save, reasmbs):
    self.parts = parts
    self.save = save
    self.reasmbs = reasmbs
```

Listing 3.10 Die Hilfsklassen »Key« und »Decomp«

Sie können diese beiden Klassen oberhalb der Definition der Klasse `Doctor` anlegen. Sehen Sie sich nun einmal die Klasse `Key` an: Sie besteht aus dem Schlüsselwort `word`, dem Gewicht `weight` (also der Priorität des Wortes) und der Liste der `decomps`. Die Klasse `Decomp` besteht aus der Struktur `parts`, die die regulären Ausdrücke für die Mustersuche beinhaltet. Die Variable `reasmbs` speichert die möglichen Antworten, falls das reguläre Muster gefunden wird. Die Variable `save` wird im Moment noch nicht benötigt. Wir kommen zu einem späteren Zeitpunkt auf sie zurück, wenn wir Eliza etwas »intelligenter« machen wollen.

Wenn Sie das Programm nach Hinzufügen dieser Zeilen speichern und starten, werden die Variablen automatisch »mit Leben gefüllt«. Durch den Aufruf im Hauptprogramm, das sich in der Klasse `Game` befindet, wird der Konstruktor der Klasse `Doctor` ausgeführt (`__init__`), und dabei werden die Variablen gesetzt. Um die Variablen in einer Textdatei dauerhaft zu speichern, benötigen Sie eine Speichermethode:

```
  def save(self):
    """ Die Datenstruktur wird in ein JSON-File geschrieben """
    doc_json = json.dumps(self.doctor, cls=DoctorEncoder, indent=4)
    with open('doctorscript-test.json', 'w') as doc_file:
      doc_file.write(doc_json)
```

Listing 3.11 Die Methode zum Speichern der Daten

In der Methode aus Listing 3.11 wird die JSON-Bibliothek genutzt, um Variablen in das *JSON-Format* umzuwandeln. Damit nicht jede einzelne Variable mit `json.dumps()` geschrieben werden muss, habe ich mich entschieden, die Variablen alle in der Liste `self.doctor` zu »bündeln«, denn damit muss nur die Variable `self.doctor` serialisiert werden.

Der Methode `json.dumps()` werden neben der zu serialisierenden Datenstruktur `self.doctor` zwei weitere Parameter übergeben. Zum einen ist das der Parameter `cls`, der eine Klassenstruktur beinhaltet, die ich noch besprechen werde. Der zweite Parameter ist `indent`, der in diesem Fall den Wert `4` trägt. Durch diesen Parameter wird bestimmt, wie weit die Zeilen in der JSON-Datei eingerückt werden, wenn sich Unterstrukturen ergeben.

```
class DoctorEncoder(JSONEncoder):
  def default(self, o):
    return o.__dict__
```

Listing 3.12 »DoctorEncoder« – eine Hilfsklasse zur Serialisierung

Die Klasse `DoctorEncoder`, die Sie ebenfalls über der Klasse `Doctor` platzieren können, macht ganz platt gesagt Folgendes: Die als erster Parameter an `json.dumps` übergebene Datenstruktur wird auseinandergefriemelt und als `dict` dargestellt und ausgegeben. Und so sieht die über `save()` geschriebene Textdatei aus:

```
{
  "initials": [
    "How do you do? Please tell me your problem."
  ],
```

```
  "finals": [
    "Goodbye. Thank you for talking to me."
  ],
  "quits": [
    "quit"
  ],
  "pres": {
    "dont" : "don't",
    "cant" : "can't"
    ...
  },
  "posts": {
    "am" : "are",
    "your" : "my"
    ...
  },
  "synons": {
    "belief" : ["belief", "feel", "think", "believe", "wish"],
    "family" : [
"family", "mother", "mom", "father", "dad", "sister", "brother", "wife",
"children", "child"]
    ...
  },
  "keys": {
    "alike": {
      "word": "alike",
      "weight": 10,
      "decomps": [
        {
          "parts": "*",
          "save": true,
          "reasmbs": [
            "In what way ?",
            "What resemblence do you see ?",
            ...
            "Could here really be some connection ?",
            "How ?"
          ]
        },
        {
```

```
          "parts": "*",
          "save": true,
          "reasmbs": [
            "In what way ?",
            "What resemblence do you see ?",
            "What does that similarity suggest to you ?",
              ...
              "Could here really be some connection ?",
              "How ?"
            ]
        }
      ]
    }
  },
  "memory": []
}
```

Listing 3.13 Ein Auszug aus der gespeicherten Sprachdatei

Die ...-Einträge habe ich eingefügt, um anzuzeigen, dass hier noch weitere Einträge stehen, die aus Platzgründen aber nicht abgedruckt werden. Die Grundstruktur für die Skriptdatei ist gelegt. Sie sollten vom Verständnis jetzt auch in der Lage sein, die Datei mit eigenen Einträgen zu erweitern.

So weit, so gut! Die Datenstrukturen sind gesetzt und Testeinträge können in eine JSON-Datei geschrieben werden. Wie aber kommen die Einträge beim Starten des Programms wieder aus der Datei in das Programm? Dafür benötigen Sie eine Lademethode, die ich `load()` genannt habe:

```
  def load(self):
    """Eliza lädt ihre Antworten"""
    with open('doctorscript.json', 'r') as doc_file:
        doc_data = json.load(doc_file)

    self.doctor['initials'] = doc_data['initials']
    self.doctor['finals'] = doc_data['finals']
    self.doctor['quits'] = doc_data['quits']
    self.doctor['keys'] = doc_data['keys']
    self.doctor['pres'] = doc_data['pres']
    self.doctor['posts'] = doc_data['posts']
```

Listing 3.14 Die Methode zum Laden der Sprachdatei

Sie können die Methode `load()` im Quelltext vor oder nach der `save()`-Methode platzieren. Die Methode liest die JSON-Struktur aus weist die einzelnen Blöcke dann wieder den Strukturen des Programms zu.

Herzlichen Glückwunsch! Sie haben gerade die Arbeiten an der Datenhaltungsschicht unseres Programms abgeschlossen. Die Klasse `Doctor` hat alle notwendigen Datenstrukturen zur Verfügung. Als Helfer haben Sie die Klassen `Key` und `Decomp` implementiert. Um Daten lesen und schreiben zu können, haben Sie die Methoden `save()` und `load()` erstellt und nebenbei noch erfahren, wie Sie in Python Textdateien öffnen, lesen, schreiben und wieder schließen können.

3.6.4 Hallo und Tschüss

Nachdem die Datenstrukturen vorliegen und außerhalb des Programms mit Inhalt versehen werden können, lassen Sie uns die beiden einfachsten Methoden zur Kommunikation mit Eliza ansehen: die Methoden für die Begrüßung und die Verabschiedung. Aufgerufen werden die Methoden vom Hauptprogramm Elizas, und zwar vor bzw. nach der für den Dialog zuständigen Endlosschleife.

```
def main(self):
  ...
  print(eliza.mainscreen()) # Begrüßungsformel ausgeben
  print(eliza.initial())

  while True:
  ...
    if output is None:
      break

    print(output)           # Eliza gibt eine Antwort

  print(eliza.final())      # Abschiedsformel ausgeben
  print()
```

Listing 3.15 Erweiterung für die Begrüßung und den Abschied

Die beiden Methoden tragen die Namen `initial()` und `final()` und befinden sich in der Klasse `Doctor`:

```
def initial(self):
  """Eine der möglichen Begrüßungen ausgeben"""
  return random.choice(self.doctor['initials'])
```

```
  def final(self):
    """Eine der möglichen Verabschiedungen ausgeben"""
    return random.choice(self.doctor['finals'])
```

Listing 3.16 Die dazugehörenden Methoden im »Doctor«-Script

Beide Methoden bestehen aus nur einer Zeile: Sie wählen zufällig einen Eintrag aus einer Liste und geben ihn zurück. Dafür nutzen wir `random.choice()` aus der `random`-Bibliothek – mehr ist nicht nötig. `random` ist die Bibliothek, die sich um das Thema Zufallszahlen kümmert.

In der Originalversion von Eliza wurden die Antworten der Reihe nach abgearbeitet und dann wieder von vorn begonnen. Ich finde die zufallsbasierte Auswahl charmanter. Sie vermeidet offensichtliche Muster und Wiederholungen und ermöglicht es sogar, dass eine Frage mehrfach gestellt wird – ein Prinzip, das in der Pädagogik als »springende Schallplatte« bekannt ist. Es kann helfen, eine Konversation gezielt zu lenken oder abzubrechen, wenn eine Antwort unklar oder ausweichend war.

Ein wichtiger Hinweis: Die Methoden haben keine Ausgabefunktion, wie beispielsweise `print()`. Sie geben den gewählten Wert per `return` zurück. Und das ist auch gut so, denn die Klasse `Doctor` realisiert die Model-Schicht in unserer MVC-Architektur und ist nicht für die Ausgabe verantwortlich. Diese Aufgabe übernimmt das Hauptprogramm bzw. der View. Auch wenn das Weiterreichen von Aufgaben zunächst umständlich wirkt, trennt es die Schichten klar voneinander – jede Komponente macht genau das, wofür sie gedacht ist.

3.6.5 Texteingabe und -analyse

Geschafft! Sie sind am Herzstück des Programms angekommen. Aus Sicht des Hauptprogramms wirkt die Verarbeitungsschleife zwar schlicht, doch an dieser Stelle läuft alles zusammen:

```
while True:
  answer = input("> ")           # User gibt einen Satz ein
  output = eliza.respond(answer) # Satz extrahieren und antworten

  if output is None:
    break                        # keine Antwort -> Schleifenende

  print(output)                  # Eliza gibt eine Antwort
```

Listing 3.17 Elizas Hauptschleife

Nach Elizas Begrüßung gibt der Benutzer sein »Problem« ein, das in der Variablen `answer` gespeichert wird. `answer` wird sodann an die Methode `respond()` von Eliza übergeben, und Eliza entscheidet, wie sie darauf reagiert.

Zeit also, sich die Methode `respond()` anzusehen. Sie befindet sich, wie erwähnt, in der Klasse `Doctor`. Ich beginne mit dem zugehörigen Aktivitätsdiagramm, das den Ablauf veranschaulicht (siehe Abbildung 3.5). Einige der Aktivitäten innerhalb des Diagramms lassen sich in wenigen Zeilen Code umsetzen, andere werden wir in Hilfsmethoden ausgliedern.

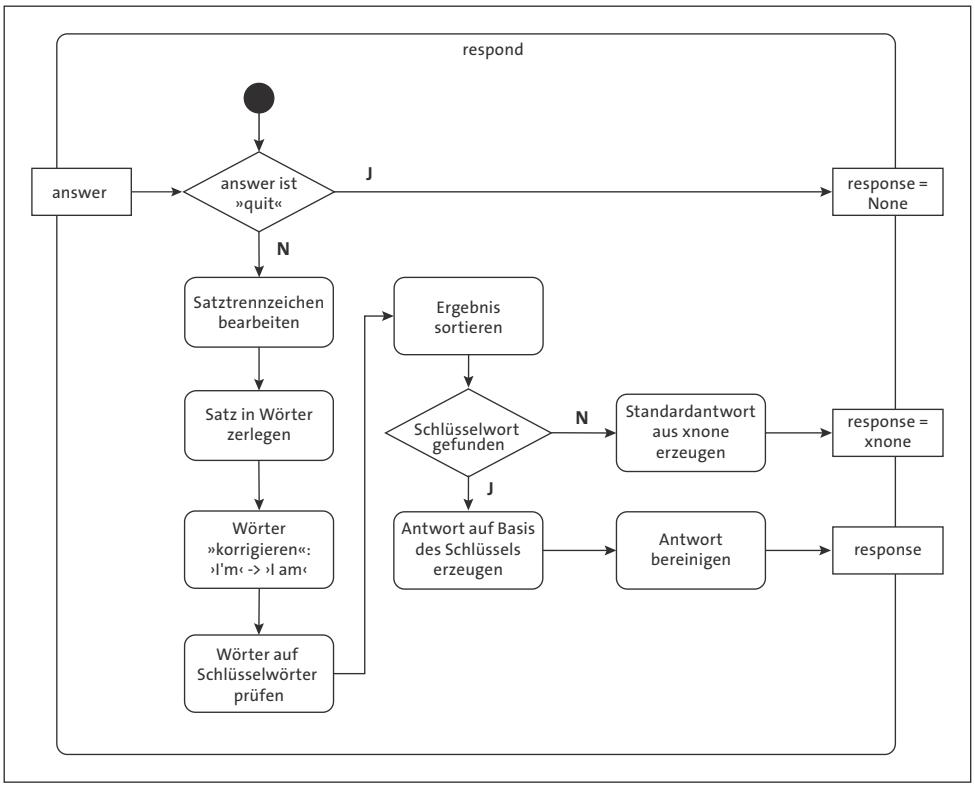

Abbildung 3.5 Die Methode »respond« von Eliza

Wenn Sie nun das Aktivitätsdiagramm in Quelltext umsetzen, erhalten Sie Code wie in Listing 3.18:

```
def respond(self, answer):
    """Eliza analysiert die Benutzereingabe und baut eine Antwort"""
    # Der User hat eines der "quit"-Schlüsselwörter eingegeben
```

```
    if answer.lower() in self.doctor['quits']:
      return None

    # Satztrennungszeichen bearbeiten und separieren
    answer = self.cleanup_punctuation(answer)
    # Wortliste aus der Antwort aufbauen
    words = [w for w in answer.split(' ') if w]
    # Abkürzungen ersetzen. Beispiele:
    # "I'm" -> "I am"
    # "youre" -> "you are"
    words = self.pre_substitute(words)
    # In der aktuellen Wortliste nach Wörtern suchen, die bekannt sind
    keywords = [self.doctor['keys'][w.lower()] for w in words if
      w.lower() in self.doctor['keys']]
    # Manche Wörter sind wichtiger als andere,
    # darum hat jedes Wort ein Gewicht.
    # Hier werden die Wörter gemäß Wichtigkeit sortiert
    keywords = sorted(keywords, key=lambda k: -k['weight'])

    response = None
    # Gibt es zu einem der Schlüssel einen Antwortsatz?
    for key in keywords:
      response = self.match_key(words, key, answer)
      if response:
        break

    if not response:
      return(self.no_answer())

    response = " ".join(response)
    response = self.finalize_answer(response)
    return response
```

Listing 3.18 Respond – Eliza analysiert und gibt Antworten.

Der Code ist kurz genug, um ihn leicht verstehen zu können. Auf einige Aspekte möchte ich dennoch eingehen:

```
# Wortliste aus der Antwort aufbauen
words = [w for w in answer.split(' ') if w]
```

In dieser Zeile Code steckt viel von den Konzepten, die Python so charmant machen – und wofür man in anderen Sprachen viel mehr Zeilen Code benötigt.

Die Zeichenkette `answer` wird mit `split(' ')` anhand von Leerzeichen in einzelne Wörter zerlegt. Aus »Ich war gestern in Berlin« entsteht so die Liste `['Ich', 'war', 'gestern', 'in', 'Berlin']`. Die nachgeschaltete *List Comprehension* filtert automatisch alle leeren Einträge heraus; nur Wörter mit Inhalt bleiben erhalten. Das Ergebnis wird schließlich der Variablen `words` zugewiesen. Ein ähnlicher Mechanismus kommt auch in der nächsten Zeile zum Einsatz:

```
# In der aktuellen Wortliste nach Wörtern suchen, die bekannt sind
keywords = [self.doctor['keys'][w.lower()] for w in words if w.lower()
  in self.doctor['keys']]
```

Auch hier wird die Anweisung am besten von rechts nach links gelesen. Die eben erzeugte Liste `words` wird durchlaufen und jedes Wort wird per `lower()` in Kleinbuchstaben umgewandelt. Dadurch werden Vergleiche mit unterschiedlichen Schreibweisen einiger Wörter harmonisiert. Anschließend wird überprüft, ob das Wort in der Liste der Schlüsselwörter (`keys`) enthalten ist. Für den Satz »I love you« wären das die Wörter »I« und »you«. Das Wort »love« gehört aktuell noch nicht dazu – aber Sie könnten es hinzufügen und einige spezifische Fragen und Antwortsätze zu »love« formulieren.

Moderne Chatbots werten Konversationen statistisch aus, zählen Wortvorkommen und könnten automatisch neue Schlüsselwörter erkennen, passende Antworten generieren und ihre Datenbasis erweitern. Aber beachten Sie bitte: Das ist keine »Intelligenz«, sondern lediglich ein automatisiertes Anreichern von Regeln.

Alle gefundenen Codewörter werden dann in die Liste `keywords` zur weiteren Auswertung geschrieben. Danach folgt eine Sortierung nach Relevanz. Auch das ist in Python sehr elegant möglich. Anstelle aufwendiger Schleifen genügt die eingebaute Funktion `sorted()`, um Listen effizient zu ordnen:

```
# Hier werden die Wörter gemäß Wichtigkeit sortiert
keywords = sorted(keywords, key=lambda k: -k['weight'])
```

Übergeben Sie nur die `keywords`, die sortiert werden sollen, dann sortiert die Methode nach dem Namen, und zwar entweder alphabetisch auf- oder absteigend. Wir möchten in diesem Fall die Codewörter aber nicht alphabetisch sortieren, sondern gemäß ihrer Wichtigkeit, die in der Variablen `weight` gespeichert ist. Auch das kann die Methode `sorted()` – unter Einsatz einer Lambda-Funktion.

Doch was ist eine *Lambda-Funktion*? Eine Lambda-Funktion ist eine anonyme, kompakte Funktion, die ohne `def` definiert wird. Lambdas können beliebig viele Argumente annehmen, enthalten aber nur einen einzigen Ausdruck. Anstatt sie klassisch zu benennen, können Sie eine Lambda-Funktion einfach einer Variablen zuweisen – und sie dann mit dem Namen dieser Variablen aufrufen. Am besten sieht man das an einem Beispiel: Die Funktion `addNum` soll vier Zahlen addieren:

```
# Definition der Lambda-Funktion
addNum = lambda u, v, w, x : u + v + w + x

result = addNum(1,2,3,4)
print(result)

# 10
```

Wie Sie sehen, unterscheidet sich der Aufbau einer Lambda-Funktion funktional kaum von einer klassischen Funktion. Die Lambda-Funktion in `respond()` dient dazu, die gefundenen Codewörter nach ihrer Gewichtung absteigend zu sortieren. So steht das relevanteste Schlüsselwort an erster Stelle und wird bei der späteren Verarbeitung bevorzugt behandelt:

```
 def respond(self, answer):
    """Eliza analysiert die Benutzereingabe und baut eine Antwort"""
    # Der User hat eines der "quit"-Schlüsselwörter eingegeben
    if answer.lower() in self.doctor['quits']:
      return None

    # Satztrennungszeichen bearbeiten und separieren
    answer = self.cleanup_punctuation(answer)
    # Wortliste aus der Antwort aufbauen
    words = [w for w in answer.split(' ') if w]
    # Abkürzungen ersetzen. Beispiele:
    # "I'm" -> "I am"
    # "youre" -> "you are"
    words = self.pre_substitute(words)

    # In der aktuellen Wortliste nach Wörtern suchen, die bekannt sind
    keywords = [self.doctor['keys'][w.lower()] for w in words if
      w.lower()
      in self.doctor['keys']]
```

```
    # Manche Wörter sind wichtiger als andere,
    # darum hat jedes Wort ein Gewicht.
    # Hier werden die Wörter gemäß Wichtigkeit sortiert
    keywords = sorted(keywords, key=lambda k: -k['weight'])

    response = None
    # Gibt es zu einem der Schlüssel einen Antwortsatz?
    for key in keywords:
      response = self.match_key(words, key, answer)
      if response:
        break

    if not response:
      return(self.no_answer())
    response = " ".join(response)
    response = self.finalize_answer(response)
    return response
```

Listing 3.19 Respond – Eliza analysiert und gibt Antworten.

Die Methode `cleanup_punctuation()` entfernt störende Satzzeichen aus der Benutzereingabe, »bereinigt« sie sozusagen. Wenn man korrekt schreibt, werden Punkte, Kommas oder Fragezeichen ohne Leerzeichen direkt hinter das vorangehende Wort gehängt. Für Elizas Verarbeitung wird das problematisch, da ein Ausdruck wie »Freude.« nicht als »Freude« erkannt wird.

Um das zu vermeiden, entfernt die Methode überflüssige Satzzeichen oder ersetzt sie durch Leerzeichen. So bleibt der Text strukturell erhalten, und die Schlüsselwörter lassen sich zuverlässig identifizieren.

```
  def cleanup_punctuation(self, answer):
    """Sonderzeichen aufräumen. """
    answer = re.sub(r'\s*\.+\s*', ' . ', answer)
    answer = re.sub(r'\s*\,+\s*', ' , ', answer)
    answer = re.sub(r'\s*\;+\s*', ' ; ', answer)
    answer = re.sub(r'\s*\:+\s*', ' : ', answer)
    answer = re.sub(r'\s*\?+\s*', '', answer)
    answer = re.sub(r'\s*\!+\s*', '', answer)
    return answer
```

Listing 3.20 Reguläre Ausdrücke beseitigen überflüssige Zeichensetzung.

Auch in Listing 3.20 kommen reguläre Ausdrücken zum Einsatz. Die erste Zeile erkennt beispielsweise einzelne oder aufeinanderfolgende Punkte, unabhängig davon, was vor oder hinter ihnen steht, und ersetzt sie durch ein Leerzeichen-Punkt-Leerzeichen. So werden auch die gern genutzten »...« (*Ellipsen*) oder versehentlich doppelt gesetzte Punkte erkannt und bereinigt.

```
def pre_substitute(self, words):
  """Hier werden Abkürzungen wie "I'm" in "I am" umgewandelt"""
  response = []
  for word in words:
    word = word.lower()
    if word in self.doctor['pres']:
      # extend statt append - sonst wird eine Liste angefügt
      response.extend(self.doctor['pres'][word].split())
    else:
      response.append(word)
  return response
```

Listing 3.21 Expansion von Abkürzungen durch »pre_substitute«

Die Methode `pre_substitute()` ersetzt abgekürzte oder alternative Begriffe (Synonyme wie »machine« oder »computers«) durch eine einheitliche Schreibweise. Die dafür genutzten Ersetzungen sind in der Skriptdatei unter dem Abschnitt `pres` definiert und können bei Bedarf erweitert werden.

`no_answer()` greift, wenn kein passendes Codewort in der Eingabe gefunden wird. In diesem Fall wählt Eliza eine Standardantwort aus der Liste `xnone`, die Sie ebenfalls beliebig ergänzen können:

```
def no_answer(self):
  return random.choice(
    self.doctor['keys']['xnone']['decomps'][0]['reasmbs'])
```

Listing 3.22 Bilden einer Antwort ohne vorherigen Kontext

Die Methode `finalize_answer()` sorgt dafür, dass Satzzeichen wie Frage- oder Ausrufezeichen am Ende einer Antwort korrekt platziert werden. Da Eliza wortbasiert arbeitet, müssen solche Zeichen in der Skriptdatei zunächst durch ein Leerzeichen abgetrennt werden, um die Textersetzung nicht zu stören. Hier sehen Sie ein Beispiel aus der Skriptdatei:

```
"Do you often think of (2) ?",
```

Gibt der Benutzer beispielsweise »I remember being a lucky kid« ein, so wird aus »Do you often think of (2) ?« der Text »Do you often think of being a lucky kid ?«. Damit das Ergebnis am Ende typografisch korrekt erscheint, entfernt `finalize_answer()` das überflüssige Leerzeichen vor dem Satzzeichen:

```
def finalize_answer(self, answer):
    answer = re.sub(r' \?', '?', answer)
    answer = re.sub(r' \.', '.', answer)
    return answer
```

Listing 3.23 Korrekturen am Satzende

Die Methode `match_key()` ist die umfangreichste in Elizas Logik und genau deshalb der krönende Abschluss unserer Betrachtung. Sie enthält das Herzstück der »Magie«, die Eliza lebendig erscheinen lässt.

```
  def match_key(self, words, key, answer):
    """
    Gibt es in der Liste ein Keyword, auf das eine Antwort gegeben werden
    kann? Jeder Key kann mehrere Möglichkeiten der De-Komposition haben
    (decomp). Beispielsweise hat das Wort "are", je nach Position: "are"
    oder "are you" unterschiedliche Antwortmöglichkeiten
    """

    for i in range(len(key['decomps'])):
      results = self.match_decomp(key['decomps'][i]['parts'], words,
        answer)

      if results is None:
        continue

      results = [self.post_substitute(words) for words in results]
      # Platzhalter (1), (2) usw. durch aus der Antwort
      # extrahierte Wörter ersetzen

      reassembled = self.reassemble(random.choice
        (key['decomps'][i]['reasmbs']), results)

      if reassembled[0] == "goto":
        goto_key = reassembled[1]
        return self.match_key(words,self.doctor['keys'][goto_key],
          answer)
```

```
            return reassembled

        return None
```

Listing 3.24 »match_key« sucht und findet Schlüsselelemente im Text.

Die Methode `match_key()` erhält die vom Benutzer eingegebenen Wörter als sortierte Liste, und zwar priorisiert nach Gewichtung. Die Methode durchläuft diese Liste und ruft für jedes Schlüsselwort `match_decomp()` auf. Dort wird geprüft, ob ein passender regulärer Ausdruck in den `parts` hinterlegt ist (etwa für das Codewort »I« oder »you«). Wird ein Treffer gefunden, wählt Eliza eine passende Antwort zufällig aus der zugehörigen `reasmbs`-Liste und speichert sie in `results`. Gibt es keinen Treffer, wird mit dem nächsten Schlüsselwort weitergesucht.

Gibt es einen Eintrag in `results`, werden in der Methode `post_substitute()` besitzanzeigende Wörter wie »mein«, »meins«, »dein«, durch ihr Gegenüber ausgetauscht. Aus »mein« wird »dein«, aus »dein« wird »mein«.

Danach wird die modifizierte Antwort von Eliza über `reassemble()` an den Re-Assembler geschickt, der den Text auf Platzhalter wie »(1)«, »(2)« usw. untersucht. Werden Platzhalter gefunden, werden diese durch die in `match_decomp()` gefundenen Muster ausgetauscht.

Ein Spezialfall ist das Schlüsselwort `goto`: Es erlaubt Eliza, eine Antwortstruktur eines anderen Schlüsselworts zu verwenden. Beispielhaft zeigt dies die Passage zur Verarbeitung von »sorry« in der Skriptdatei:

```
...
"sorry": {
  "word": "sorry",
  "weight": 1,
  "decomps": [
    {
      "parts": "(.*)",
      "save": true,
      "reasmbs": [
        "Please don't apologise.",
        "Apologies are not necessary.",
        "No need to aplogise.",
        "You don't have to feel sorry.",
        "I've told you that apologies are not required."
      ]
```

```
      }
    ]
  },
  "apologise": {
    "word": "apologise",
    "weight": 1,
    "decomps": [
      {
        "parts": "(.*)",
        "save": true,
        "reasmbs": [
          "goto sorry"
        ]
      }
    ]
  },
  ...
```

Listing 3.25 Ausschnitt aus dem Sprachskript zum Wort »sorry«

Gibt der Benutzer »sorry« ein, erkennt Eliza das Codewort über `match_decomp()` und liefert eine zufällige Antwort aus dem entsprechenden Antwortenblock. Bei der Eingabe von »apologise« findet Eliza ebenfalls einen Treffer, aber nur die Antwort `goto sorry`. Das veranlasst das Eliza dazu, den Antwortblock von `sorry` aufzurufen und aus ihm eine passende Reaktion auszuwählen.

Dieses Prinzip erlaubt es, mehrere verwandte Begriffe auf denselben Antwortbereich umzulenken, ohne Inhalte doppelt pflegen zu müssen. Alternativ können Sie dem Block `apologise` auch eigene Antworten hinzufügen. Wird dabei zufällig wieder `goto sorry` gewählt, reagiert Eliza mit einer Antwort aus dem ursprünglichen `sorry`-Block.

Nachdem Sie nun die Grundidee der Methode `match_decomp()` verstanden haben, werfen wir im nächsten Schritt einen genaueren Blick auf die beteiligten Methoden. Die Methoden haben viele Kommentare, um Ihnen das Verständnis zu erleichtern.

```
def match_decomp(self, parts, words, answer):
  """
  Bei der Antwort-"suche" wird nach Keywords gesucht -
  beispielsweise "i" oder "your". Gibt es ein solches Keyword, wird
  hier nachgesehen, ob weitere Details vorliegen. Bei "i"
  beispielsweise "i was", "i am", "i don't", "* i * you *".
  Dann wird anhand des Musters die Antwort auseinandergeschnitten.
```

```
"i love you" -> "* i * you *"
Erster Stern: leer
Zweiter Stern: love
Dritter Stern: leer
"""

results = []
match = re.match(parts, answer)

# groups(0) ist der gesamte Text
# groups(1) ist das erste Muster
# groups(2) ist das zweite Muster usw.

if match:
  for i in range(1,len(match.groups())+1):
    results.append(match.group(i).split())
  return results
return None
```

Listing 3.26 Zerlegen der Benutzereingabe

Die Methode `match_decomp()` analysiert den übergebenen Text mithilfe regulärer Ausdrücke und zerlegt ihn in passende Bestandteile. Nehmen wir erneut das Beispiel »I love you«: In der folgenden Liste in Tabelle 3.1 sehen Sie, welche Ausdrücke mit diesem Satz übereinstimmen.

Scheuen Sie sich nicht vor regulären Ausdrücken – der Einstieg mag ungewohnt sein, doch sie sind ein äußerst leistungsfähiges Werkzeug zur Textverarbeitung und bilden das Rückgrat vieler sprachbasierter Anwendungen.

Text	Regulärer Ausdruck	Ergebnis(se)
I love you	(.*)	Der reguläre Ausdruck untersucht alle Zeichen. `(.*)` bedeutet: »Finde beliebig viele Zeichen beliebig oft.« Damit wird der gesamte Text als Muster identifiziert und das Ergebnis ist: #0 : (.*) = »I love you« Die Muster in Klammern werden durchnummeriert und können referenziert werden.

Tabelle 3.1 Funktionsweise einiger regulärer Ausdrücke

Text	Regulärer Ausdruck	Ergebnis(se)
I love you !	(.*) I (.*) you (.*)	Der reguläre Ausdruck bedeutet: »Finde beliebig viele Zeichen, beliebig oft, gefolgt von der Zeichenkette ›I‹. Danach finde beliebig viele Zeichen, beliebig oft, gefolgt von der Zeichenkette ›you‹. Danach finde beliebig viele Zeichen, beliebig oft.« Die Muster in Klammern werden durchnummeriert und können referenziert werden. #0: (.*) = {}, also leer #1: (.*) = ›love‹ #2: (.*) = ›!‹
Tisch, Fische, frisch, Zischt, sympathisch, Gischt, Wischen,	(.*)isch	Findet beliebige Zeichen vor der Endung »isch«. Da es nach »isch« keinen Begrenzer gibt, muss nur die Folge »isch« vorhanden sein, um das Kriterium zu erfüllen. #0: (.*) = {›T‹, ›F‹, ›fr‹, ›Z‹, ›sympath‹, ›G‹, ›W‹} – je nach Eingabe

Tabelle 3.1 Funktionsweise einiger regulärer Ausdrücke (Forts.)

Die Methode `post_substitute()` übersetzt »mein« in »dein« und umgekehrt:

```
def post_substitute(self, posts):
  """
  Wenn Sätze "echt" bzw. natürlich klingen sollen,
  muss an manchen Stellen die Bedeutung umgekehrt werden.
  Aus "ich" wird "du", aus "dein" wird "mein" usw.
  """
  answer = []
  for word in posts:
    if word in self.doctor['posts']:
      # extend statt append - sonst wird eine Liste angefügt
      answer.append(self.doctor['posts'][word])
    else:
      answer.append(word)
  return answer
```

Listing 3.27 Aus »mein« mach »dein«.

Zu guter Letzt sucht die Methode `reassemble()` nach Platzhaltern und ersetzt potenzielle Vorkommnisse durch die Ergebnisse der regulären Ausdrücke aus `match_decomp()`:

```
def reassemble(self, reasmb, results):
  """
  Texte können "Platzhalter" beinhalten: (1), (2) usw.
  Diese Platzhalter sind i.d.R. dazu da, Codewörter aus der Eingabe
  des Benutzers zu reflektieren. Hier werden die Platzhalter durch die
  analysierte Benutzereingabe ersetzt. Die Benutzereingabe wird in
  match_decomp analysiert und in ihre Bestandteile zerlegt.
  """
  answer = []
  words = reasmb.split()
  for word in words:
    if not word:
      continue
    # Es gibt einen Ausdruck zu ersetzen
    if word[0] == "(" and word[-1] == ")":
      index = int(word[1:-1])
      if index < 1 or index > len(results):
        answer.append("FEHLER")
      else:
        # extend statt append - sonst wird eine Liste angefügt
          answer.extend(results[index-1])
    else:
      answer.append(word)
  return answer
```

Listing 3.28 Zusammensetzen der Antwort

Das war's! Das ist das ganze Geheimnis hinter einem elementaren Chatbot. Sie haben Ihren ersten Ausflug in die Gefilde der künstlichen Intelligenz erfolgreich unternommen und abgeschlossen. Was noch fehlt, ist der Inhalt der Skriptdatei. Die Skriptdatei ist leider zu lang, um sie in diesem Buch abzudrucken – Sie finden sie aber im Downloadmaterial zu diesem Buch im Ordner *Eliza* unter dem Namen *doctorscript.json*.

Das Downloadmaterial finden Sie unter *https://www.rheinwerk-verlag.de/6150*. Scrollen Sie etwas herunter und klicken Sie auf den Reiter Materialien zum Buch.

3.7 Ausblick

Dieser kurze Ausflug zu den regulären Ausdrücken und zum Parsen von Texteingaben hat gezeigt, wie Systeme wie Eliza mit einfachen Mitteln wirkungsvolle Interaktionen erzeugen können. Im nächsten Kapitel heben wir unser Projekt auf eine neue Stufe. Aus der textbasierten Konsole wird eine interaktive, grafische Anwendung.

Mit Pygame gestalten wir eine ansprechende Benutzeroberfläche, inklusive Chatfenster, Nachrichtenbubbles und Scrollfunktion. Freuen Sie sich auf den Einstieg in die visuelle Gestaltung. Wir machen aus der Konsolenanwendung etwas fürs Auge und bringen Ihre Anwendung zum Leben.

3.8 Literatur und Quellenangaben

Hier finden Sie die Auflistung der in diesem Kapitel genutzten und empfohlenen Literatur:

- [1] Joseph Weizenbaum, *Eliza*, *https://dl.acm.org/doi/10.1145/365153.365168*
- [2] Charles Hayden, *Eliza*, *https://www.chayden.net/eliza/Eliza.html*
- [3] Erich Gamma et al., *Design Patterns. Elements of Reusable Object-Oriented Software.* Englische Ausgabe, 39,99 €, ISBN-13: 978-0201633610
- [4] Erich Gamma et al., *Design Patterns: Entwurfsmuster als Elemente wiederverwendbarer objektorientierter Software*. Deutsche Ausgabe, 39,99 €, ISBN-13: 978-3826697005
- [5] Eric Freeman, *Entwurfsmuster von Kopf bis Fuß: Mit Design Patterns flexible objektorientierte Software erstellen*. Deutsche Ausgabe, 49,90 €, ISBN-13: 978-3960091622
- [6] George Bernhard Shaw, *Pygmalion*. Englische Ausgabe, 3,84 €, ISBN-13: 978-0008480066
- [7] Erstellen von ASCII-Art, *https://manytools.org/hacker-tools/ascii-banner/*
- [8] Marianna Baranovska-Bölter, Stefan Höltgen, *Hello, I'm Eliza. Fünfzig Jahre Gespräche mit Computern*. 22,00 €, ISBN-13: 978-3897335882

14.5 Mit WFC zu Platformer-Leveln

Die wahre Stärke des WFC-Algorithmus zeigt sich bei komplexeren Strukturen. Anstatt nur einfache Landschaften zu erzeugen, wenden wir WFC zur Generierung ganzer Platformer-Level an. Dabei lassen wir uns von einem der Klassiker inspirieren: *Lode Runner*. (Die Copyright- und Markenrechte an Lode Runner liegen derzeit bei *Tozai Games, Inc*.)

Lode-Runner-Level bestehen aus klar definierten Spielelementen: feste Blöcke, Leitern, Seile, Goldstücke, Gegner und die Wege, die der Spieler nehmen kann. Unser Ziel ist es, nicht diese Level zu kopieren, sondern aus ihnen lokale Contraints zu extrahieren, die als Grundlage für neue, eigenständige Level dienen.

Wir verwenden handgefertigte Level, die an Lode Runner erinnern, als Trainingsdaten für unseren WFC-Algorithmus. Dabei geht es nicht darum, Lode-Runner-Level nachzubauen, sondern die Grundstruktur klassischer Platformer zu analysieren. Lode Runner dient uns lediglich als reichhaltige Quelle typischer Beziehungen zwischen den einzelnen Spielelementen.

Dazu zerlegen wir die Level in einzelne Kacheltypen – etwa Blöcke, Leitern, Seile und leere Felder. Jeder dieser Typen wird im WFC als eigene *Tile* behandelt. Der entscheidende Schritt: Wir analysieren, welche Kacheln (engl. *tiles*) typischerweise nebeneinander oder übereinander erscheinen. So lassen sich implizite Designregeln erkennen – etwa, dass Leitern meist vertikal verlaufen oder dass leere Felder zwischen Plattformen Bewegungsraum bieten.

Diese statistisch abgeleiteten Nachbarschaftsbeziehungen speisen wir in den WFC-Algorithmus ein. Anders als bei einem manuell definierten Regelwerk lernt der Algorithmus also aus Beispielen, welche lokalen Konfigurationen plausibel sind. Das Ergebnis dieses Prozesses ist ein Regelwerk, das die grundlegenden strukturellen Beziehungen zwischen den generischen Elementen eines Platformers beschreibt – wie Bewegungsraum an Plattformen angrenzt, wie vertikale Bewegungselemente platziert werden und wie solide Hindernisse den Bewegungsfluss beeinflussen können.

Wir nutzen also die Informationen, um die Essenz des Platformer-Genres in Form von lokalen Nachbarschaftsregeln zu extrahieren, ohne uns an die spezifische Levelarchitektur oder das einzigartige Gameplay eines bestehenden Klassikers zu binden. Das Ziel ist ein frischer, neuer Platformer mit einer durch WFC-generierten, aber dennoch strukturell fundierten Welt.

Hinweis: Da keine Freigabe zur Verwendung von Lode-Runner-Screenshots vorliegt, verweise ich auf die *Arduboy*-Webseite [1], auf der einige Level einsehbar sind. Für das

Training wurden unter anderem die Level 1, 2, 3, 7 und 20 genutzt. Dazu habe ich jeden Kacheltyp des originalen Spiels durch einen Zahlenwert ersetzt:

- 0 ist das leere Feld.
- 1 ist der normale Stein.
- 2 ist der Beton, durch den nicht gegraben werden kann. 2 habe ich durch 1 ersetzt, da ich keinen Beton in unserem Platformer umsetzen möchte.
- 3 ist die Leiter.
- 4 ist das Seil.

Die anderen Werte sind die Goldkisten, die Gegner, der Spieler, Falltüren und die Himmelsleiter, die ich ebenfalls nicht umsetzen möchte und durch eine 0 ersetzt habe. Das Ergebnis war bestenfalls »okay«, aber nicht gut: Viele Level waren nicht zusammenhängend und konnten nicht gespielt werden. Daher bin ich dazu übergegangen, einen Level-Editor zu schreiben und selbst Beispiellevel zu erstellen.

14.5.1 Der Level-Editor

Die Grundzüge eines *Level-Editors* haben Sie bereits in Kapitel 11, »Von Sudoku zu intelligenten Lösungen«, kennengelernt, wo Zahlen per Tastatureingabe in ein 9×9-Raster gesetzt wurden. Für Tut's Treasure greifen wir dieses Konzept auf, erweitern es jedoch: Statt Zahlen verwenden wir kleine Grafiken, die verschiedene Levelelemente repräsentieren. Die Grafiken für den Level-Editor sehen Sie in Abbildung 14.5.

Abbildung 14.5 Die Grafiken für den Level-Editor

Die Steuerung bleibt vertraut, denn das Raster lässt sich mit den Pfeiltasten navigieren. Allerdings arbeiten wir nun mit einem deutlich größeren Spielfeld: 120×18 Zellen. Das bietet ausreichend Raum, um vielfältige Trainingsdaten für den WFC-Algorithmus zu erstellen. Zur besseren Orientierung ist der Level in 3×3-Segmente unterteilt, markiert durch ein weißes Raster (siehe Abbildung 14.6). So wird auf einen Blick sichtbar, welche lokalen Strukturen vom Algorithmus erfasst und gelernt werden können.

Den Level-Editor finden Sie im Downloadordner zu diesem Kapitel.

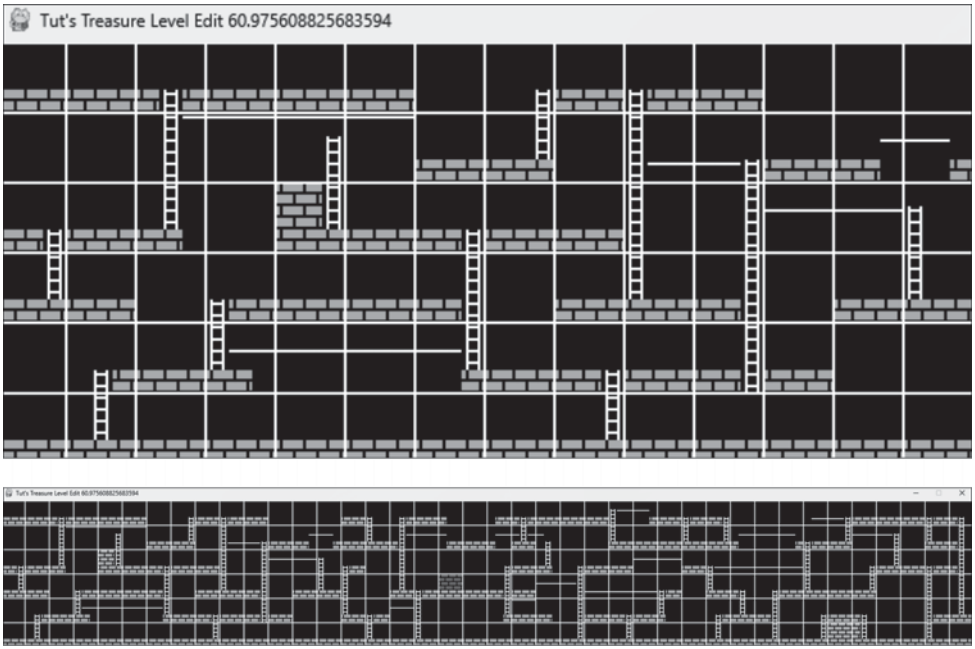

Abbildung 14.6 Der handgefertigte Trainingslevel

14.5.2 Analyse der Level

Nach Eingabe des Levels habe ich ihn durch ein Analyseprogramm geschickt (das Sie ebenfalls im Downloadbereich finden). Mithilfe einer 3×3-*Matrix* (eines sogenannten *Kernels*) wird der Level Abschnitt für Abschnitt durchlaufen. An jeder Position wird der aktuelle 3×3-Ausschnitt als eigenständiges Tile-Element gespeichert, wie Sie in Abbildung 14.7 sehen.

Nehmen wir an, Sie haben einen Level, der 6×6 Kacheln groß ist. Der Kernel beginnt oben links, liest die ersten 3×3 Kacheln ein und verschiebt sich dann jeweils um eine Kachel nach rechts. So entstehen vier Ausschnitte pro Zeile. Danach verschieben Sie den Kernel eine Zeile tiefer und verfahren analog. Am Ende des Durchlaufs haben Sie aus dem Level 16 Bildelemente extrahiert.

Diese kleinen Muster bilden die Grundbausteine für neue Level. In einem echten Platformer sind die Level natürlich größer, aber das Prinzip bleibt gleich. Ich habe mich bewusst für eine Kernelgröße von 3×3 entschieden, um möglichst viele lokale Variationen erfassen zu können.

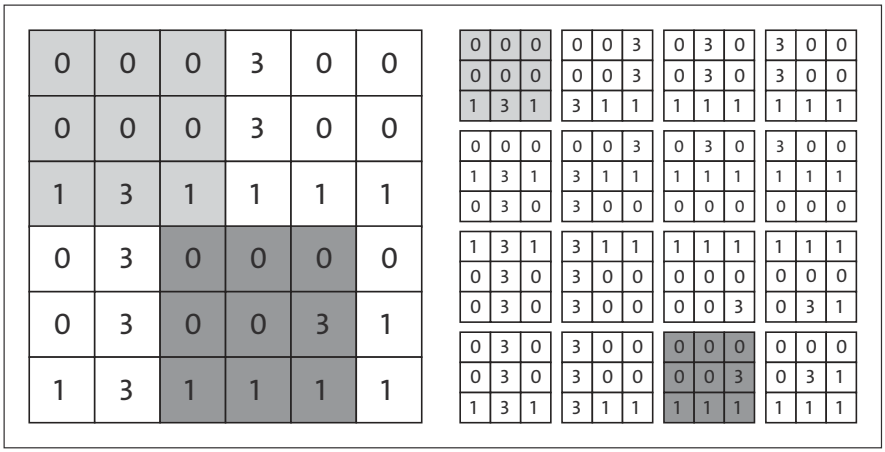

Abbildung 14.7 Analyse des Levels mit einem 3×3-Kernel

Im nächsten Schritt zählen Sie, wie häufig jede dieser Kachelkombinationen im Level vorkommt. Diese Häufigkeiten fließen als Gewichtungen in den WFC-Algorithmus ein – wie bereits im Beispiel mit Land, Küste und Wasser erklärt. Häufig vorkommende Muster werden entsprechend wahrscheinlicher erneut verwendet.

In Schritt drei legen Sie die *Nachbarschaftsregeln* fest. Dazu ermitteln Sie für jede Kachel, welche Typen sich direkt daneben in den vier Himmelsrichtungen befinden. Diese lokalen Beziehungen speichern Sie in einem Dictionary ab (siehe Abbildung 14.8). Die dunkelgraue Kachel steht im Regelwerk, die hellgrauen Kacheln sind die Nachbarn.

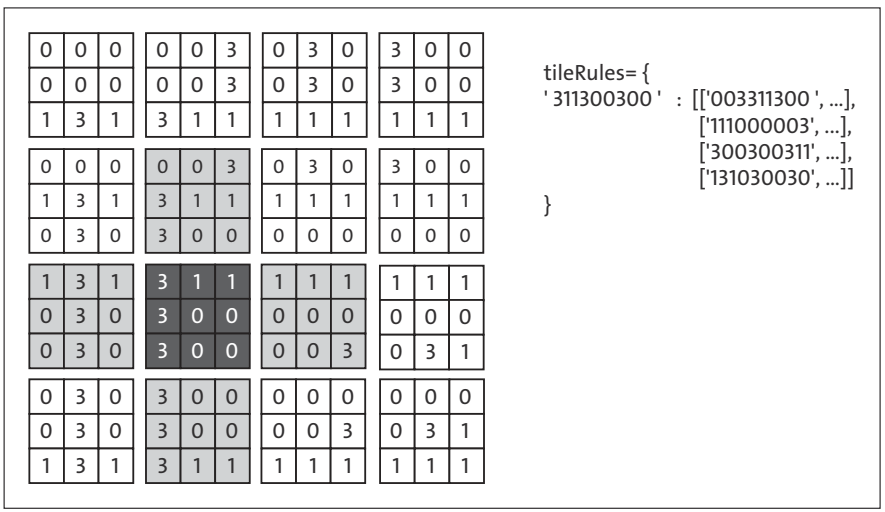

Abbildung 14.8 Erstellen der Regeln

Damit ist die Konfiguration abgeschlossen: Die Elementhäufigkeit liefert die Gewichtung, die Nachbarschaftsregeln bestimmen das Verhalten – und der WFC-Algorithmus kann aus diesen Informationen neue, konsistente Level generieren.

14.5.3 Das Analyseprogramm

Das Programm zur Analyse der Level (siehe Abbildung 14.9) finden Sie im Downloadordner dieses Kapitels unter dem Namen *levelanalyzer.py*.

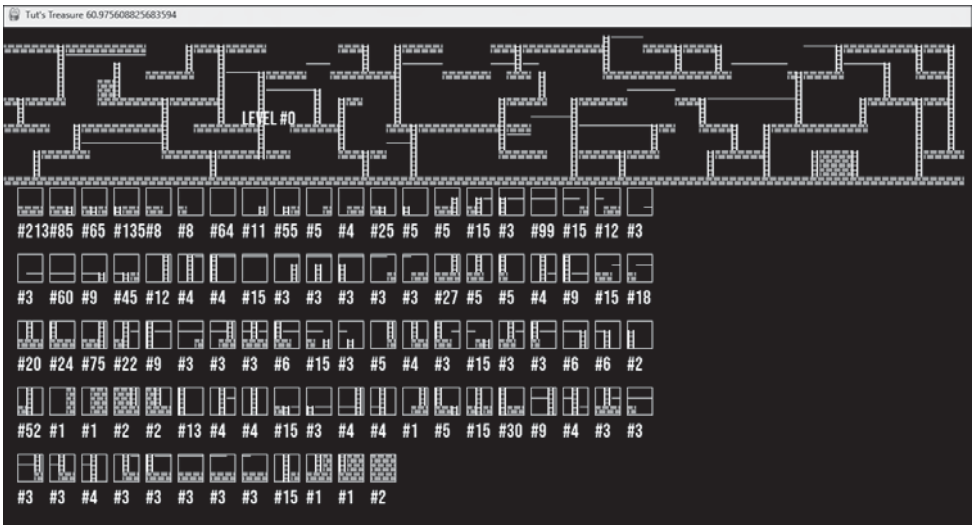

Abbildung 14.9 Die Analyse des Trainingslevels

14.5.4 Die WFC-Daten

Das Analyseprogramm generiert ein Python-Skript, das alle Konfigurationsdaten enthält und als Eingabe für den WFC-Algorithmus genutzt werden kann. Da es keine urheberrechtlich geschützten Inhalte enthält, erhalten Sie die vollständige Konfigurationsdatei. Die ursprünglichen Quelldaten für die Analyse sind aus Lizenzgründen nicht enthalten.

14.6 Der WFC-Code

Im Platformer Tut's Treasure kann der Spieler zu Beginn wählen, ob er einen handgebauten Level oder einen durch WFC generierten Level spielen möchte. Die Weltgenerierung

über WFC ist auf drei Klassen verteilt und wird von der Klasse `gameplaywfc` angestoßen. Im Folgenden zeige ich Ihnen, wie das WFC-Modul eingebunden ist.

14.6.1 gameplaywfc

Die Klasse `gameplaywfc` ist weitgehend identisch mit der regulären Klasse `gameplay`, unterscheidet sich jedoch in der Art der Levelgenerierung. In der Methode `init()` finden Sie die Initialisierung des WFC-Moduls:

```
def __init__(self) -> None:
  super().__init__()
  ...
  self.time = 0
  self.current_level = 0
  self.levelmanager = LevelManager()
  self.level = Level()
  random.seed(31)
  self.font = pygame.font.Font(pygame.font.get_default_font(), 11)

  self.camera = Camera(0,0,30,10, self.offset_x, self.offset_y)
  self.level.set_camera(self.camera)
  self.create_world()
  ...

def update(self, dt):
  self.time += dt / 1000
  if self.state == GENERATION:
    result = self.tile_world.collapse_wave_function()
    self.level.data = self.tile_world.tile_the_world()
    if result == 0:
      self.place_player()
      self.place_exit()
      self.place_cats()
      self.place_mummy()
      self.level.setup()
      self.state = GAME
  if self.state == GAME:
    if not self.level.is_paused:
      ...
  return super().update(dt)
```

```
  def place_player(self):
    row = 2
    for col in range(LEVEL_WIDTH):
      if self.level.data[row][col] == TILE_BRICK and \
        self.level.data[row-1][col] == TILE_VOID:
        self.level.data[row-1][col] = TILE_HERO
        return

  def place_exit(self):
    row = 10
    col = 31
    self.level.data[row][col] = TILE_EXIT

  def place_cats(self):
    numcats = 7
    listcats = []
    for row in range(LEVEL_HEIGHT):
      for col in range(LEVEL_WIDTH):
        if self.level.data[row][col] == TILE_BRICK and \
          self.level.data[row-1][col] == TILE_VOID:
          listcats.append((row-1,col))
    random.shuffle(listcats)
    for catrow, catcol in listcats[:numcats]:
      self.level.data[catrow][catcol] = TILE_LODE

  def place_mummy(self):
    nummummies = 5
    listmummies = []
    for row in range(LEVEL_HEIGHT):
      for col in range(LEVEL_WIDTH):
        if self.level.data[row][col] == TILE_BRICK and \
          self.level.data[row-1][col] == TILE_VOID:
          listmummies.append((row-1,col))
    random.shuffle(listmummies)
    for mummyrow, mummycol in listmummies[:nummummies]:
      self.level.data[mummyrow][mummycol] = TILE_MUMMY

  def create_world(self):
    self.state = GENERATION
    self.level = Level()
```

```
self.camera = Camera(0,0,30,10, self.offset_x, self.offset_y)
self.level.set_camera(self.camera)
self.tile_world = TileWorld(WORLDWIDTH, WORLDHEIGHT)
```

Listing 14.2 WFC-Initialisierung im Spielmodul

In der Methode `init()` wird zunächst ein `Level`-Objekt erstellt, und das unabhängig davon, ob der Level manuell oder per WFC erzeugt wurde. Anschließend ruft die Methode `setup()` ein `TileWorld`-Objekt auf, das die eigentliche WFC-Logik übernimmt. Dabei werden die beiden Parameter `WORLDWIDTH` und `WORLDHEIGHT` übergeben, im Unterschied zur üblichen `LEVELWIDTH`. Das liegt daran, dass die Weltstruktur zunächst auf einer gröberen Ebene von 3×3-Blöcken aufgebaut wird. Diese Blöcke bilden die Grundlage für die WFC-Berechnung, bevor sie später in feinere Einzelkacheln umgewandelt werden. Die Verwendung von `WORLDWIDTH` sorgt dabei für eine größere Vielfalt in der Grobstruktur.

Die Erzeugung der Welt folgt einem iterativen Ansatz, damit für Sie der Entstehungsprozess sichtbar bleibt. Zu Beginn befindet sich das Spiel im Zustand `GENERATION`, in dem `update()` fortlaufend Kacheln kollabieren lässt. Erst wenn die Welt vollständig generiert ist, wechselt das Spiel in den Zustand `GAME`.

Nach Abschluss der Generierung folgt die Platzierung von Spieler, Ausgang, Mumien und Artefakten (den goldenen Katzen). Dafür sind die Methoden `place_xxx()` verantwortlich, die außerhalb der Zuständigkeit des WFC liegen. Die generative Platzierung komplexer Spielobjekte würde den Algorithmus überfordern, daher erfolgt sie klassisch per Logik und Zufall.

Die Methode `create_world()` beinhaltet ein neues Konzept: eine `Camera`. Der Spieler wird nie den gesamten Level auf einmal auf dem Bildschirm sehen, sondern wie durch ein Teleobjektiv immer nur einen kleinen Ausschnitt. Die Klasse `Camera` ist dabei die verwaltende Instanz, die sich den darzustellenden Bildausschnitt merkt. Wir gehen später im Projekt Tut's Treasure noch genauer auf die `Camera` ein.

14.6.2 Tile

Im Zentrum unseres Systems zur prozeduralen Landschaftsgenerierung steht die Klasse `Tile`. Sie repräsentiert eine einzelne Zelle in dem Raster, aus dem unsere virtuelle Welt entsteht. Jede Instanz dieser Klasse trägt die Verantwortung dafür, ihren Zustand zu verwalten und mit ihren Nachbarn in Beziehung zu treten, um so die kohärente Struktur der generierten Landschaft zu gewährleisten. Darüber hinaus stellt `Tile` zentrale Operationen auf den Datenstrukturen bereit, die für WFC benutzt werden.

```
import random
import copy

from Conf import *

class Tile():
  def __init__(self, y, x):
    # Koordinaten der Kachel
    self.y = y
    self.x = x
    # Eine Liste zu verwenden ist nicht optimal,
    # da bspw. die Kachel oben links keine 4 Nachbarn hat.
    self.neighbours = {}
    self.list_possibilities = list(tileRules.keys())
    self.entropy = len(self.list_possibilities)

  def add_neighbour(self, direction, tile):
    self.neighbours[direction] = tile

  def get_neighbour(self, direction):
    return self.neighbours[direction]

  def get_directions(self):
    return list(self.neighbours.keys())

  def get_possibilities(self):
    return self.list_possibilities

  def collapse(self):
    # Die einzelnen Tiles können gewichtet werden. So kann eine Land-
    # schaft mit mehr Wasser, mehr Wald oder mehr Wüste erstellt werden.
    weights = [tileWeights[possibility] for possibility in
      self.list_possibilities]
    self.list_possibilities = random.choices(self.list_possibilities,
      weights = weights, k=1)
    self.entropy = 0

  def constrain(self, neighbour_possibilities, direction_to_explore):
    reduction_possible = False
```

```
    if self.entropy > 0:
      path_to_neighbours = []
      for neighbour_poss in neighbour_possibilities:
        path_to_neighbours +=
        tileRules[neighbour_poss][direction_to_explore]
      path_to_neighbours = list(set(path_to_neighbours))

      # Ein Constraint wirkt von der aktuellen Zelle auf ihre
      # Nachbarn, aber auch von ihren Nachbarn auf ggf. diese Zelle.
      # Daher wird die gegensätzliche Himmelsrichtung bestimmt.

      opposite = (direction_to_explore + 2) % 4

      for possibility in self.list_possibilities.copy():
        if possibility not in path_to_neighbours:
          self.list_possibilities.remove(possibility)
          if len(self.list_possibilities) == 0:
            print("Abbruch. Keine Möglichkeiten mehr")
          reduction_possible = True

      # Wenn eine oder mehrere Möglichkeiten entfernt wurden,
      # dann muss auch die Entropie angepasst werden.
      self.entropy = len(self.list_possibilities)
    return reduction_possible

  def removePossibility(self, number):
    if number in self.list_possibilities:
      self.list_possibilities.remove(number)
      self.entropy = len(self.list_possibilities)

  def removePossibilities(self, tile_list):
    for tile in tile_list:
      if tile in self.list_possibilities:
        self.list_possibilities.remove(tile)
    self.entropy = len(self.list_possibilities)

  def setPossibilities(self, tile_list):
    self.list_possibilities = []
    self.entropy = 0
    for tile in tile_list:
```

```
            self.list_possibilities.append(tile)
            self.entropy += 1

    def handle_input(self, event_list):
        pass

    def update(self, dt):
        pass

    def render(self, surface):
        pass
```

Listing 14.3 Der Code der Klasse »Tile«

Beim Erstellen einer neuen `Tile`-Instanz in `init()` werden zunächst die y- und x-Koordinaten im Raster festgelegt. Diese Koordinaten definieren die Position der Kachel in der Welt. Anschließend wird ein leeres Dictionary namens `neighbours` initialisiert, das die benachbarten Tiles in den vier Himmelsrichtungen flexibel verwaltet. Dies ist eine elegante Alternative zu festen Listenstrukturen, besonders an den Rändern des Rasters.

Darüber hinaus wird die Liste `list_possibilities` mit allen im `tileRules`-Dictionary definierten Tile-Typen gefüllt – also jenen Bausteinen, die aus der Level-Analyse extrahiert wurden. Die `entropy` einer Kachel entspricht zu Beginn der Anzahl dieser Möglichkeiten. Eine hohe Entropie bedeutet viele Möglichkeiten, während eine niedrige Entropie (bis hin zu null) bedeutet, dass der Zustand der Kachel bereits feststeht.

Die Methoden `add_neighbour()` und `get_neighbour()` erlauben das gezielte Setzen und Abfragen benachbarter Tiles. `add_neighbour()` ermöglicht es, eine benachbarte `Tile`-Instanz in einer bestimmten Himmelsrichtung (`direction`) zu speichern, während `get_neighbour()` den Zugriff auf diese gespeicherte Nachbarkachel erlaubt. Die Methode `get_directions()` gibt schlicht eine Liste der aktuell bekannten Nachbarrichtungen zurück. Über `get_possibilities()` kann jederzeit der aktuelle Zustandsraum einer Kachel abgefragt werden.

Die Methode `collapse()` reduziert die Kachel auf genau einen Zustand, der mithilfe von `random.choices()` gewichtet aus den möglichen Typen ausgewählt wird (basierend auf `tileWeights`). Anschließend wird die Liste `list_possibilities` auf diesen einen gewählten Zustand reduziert und die Entropie auf null gesetzt – die Ungewissheit ist beseitigt.

Die Methode `constrain()` ist zentral für die Propagation. Sie erhält die möglichen Zustände der Nachbarkacheln, `neighbour_possibilities`, sowie und die gewählte Richtung `direction_to_explore` als Argumente. Basierend auf den im `tileRules`-Dictionary defi-

nierten Regeln filtert `constrain()` die `list_possibilities` der aktuellen Kachel. Wenn eine Möglichkeit nicht mit den Nachbarbedingungen vereinbar ist, wird sie entfernt. Die Berechnung der `opposite`-Richtung stellt sicher, dass die Constraints in beide Richtungen wirken. Wird die `entropy` durch diesen Prozess reduziert, gibt die Methode `True` zurück, was signalisiert, dass eine Weiterverbreitung der Constraints im Raster notwendig sein könnte.

Die Methoden `removePossibility()` und `removePossibilities()` bieten die Möglichkeit, gezielt einzelne oder mehrere mögliche Zustände aus der `list_possibilities` zu entfernen und die Entropie entsprechend anzupassen. Die Funktion `setPossibilities()` erlaubt es, die Liste der Möglichkeiten einer Kachel komplett neu zu setzen und die Entropie neu zu berechnen.

14.6.3 TileWorld

Die Klasse `TileWorld` bildet das Herzstück der prozeduralen Weltgenerierung. Sie verwaltet ein zweidimensionales Raster aus `Tile`-Objekten und steuert den *Wave Function Collapse*-Algorithmus, um dieses Raster mit kohärenten, abwechslungsreichen Mustern zu füllen.

```
import random
from Conf import *
from model.constants import *
from Tile import Tile
from Stack import Stack

class TileWorld():
  def __init__(self, width, height):
    self.width = width
    self.height = height
    self.world = []
    self.create_world()

  def create_world(self):
    # Liste aller Kacheln, die in der obersten Reihe leer sind
    temp_list = list(tileRules.keys())

    tiles_ceiling = []
    for tile in temp_list:
      if str(tile[:3]) == 'OOO':
        tiles_ceiling.append(tile)
```

```
tiles_floor = []
for tile in temp_list:
  if (str(tile[6:9])) == '111' and '3' not in str(tile) and '4'
    not in str(tile):
    tiles_floor.append(tile)

tiles_floor_ladder = []
for tile in temp_list:
  if (str(tile[6:9])) == '111' and '3' in str(tile) and '4'
    not in str(tile):
    tiles_floor_ladder.append(tile)

tiles_ladder = []
for tile in temp_list:
  if '3' in str(tile) or '4' in str(tile):
    tiles_ladder.append(tile)

tiles_ropes = []
for tile in temp_list:
  if '4' in str(tile) and '3' not in str(tile):
    tiles_ropes.append(tile)

# Level erzeugen
self.world = [[Tile(row, col) for col in range(self.width)] for
  row in range(self.height)]

# Generische Kacheln anlegen - Platz für die Tür in der Mitte
self.world[3][11].setPossibilities(['000000111'])
self.world[3][10].setPossibilities(['000000111'])
self.world[3][9].setPossibilities(['000000111'])
self.world[2][11].setPossibilities(['000000000'])
self.world[2][10].setPossibilities(['000000000'])
self.world[2][9].setPossibilities(['000000000'])

# Nachbarn berechnen und eintragen
for row in range(self.height):
  for col in range(self.width):
    tile = self.world[row][col]
    if row == self.height - 1:
```

```
            if random.randrange(1,100) < 70:
              tile.setPossibilities(tiles_floor)
            else:
              tile.setPossibilities(tiles_floor_ladder)
           elif row == 0:
             tile.setPossibilities(tiles_ceiling)

          if row > 0:
            tile.add_neighbour(NORTH, self.world[row-1][col])
          if row < self.height - 1:
            tile.add_neighbour(SOUTH, self.world[row+1][col])
          if col > 0:
            tile.add_neighbour(WEST, self.world[row][col-1])
          if col < self.width - 1:
            tile.add_neighbour(EAST, self.world[row][col+1])

    def get_entropy(self, row, col):
      return self.world[row][col].entropy

    def get_type(self, row, col):
      if len(self.world[row][col].list_possibilities) == 0:
        print("Starting over...")
        self.create_world()
      return self.world[row][col].list_possibilities[0]

    def get_tiles_with_lowest_entropy(self):
      # Den Maximalwert für die Vergleiche bestimmen
      max_entropy = len(list(tileRules.keys()))
      tile_list = []
      # Alle Tiles durchgehen
      for row in range(self.height):
        for col in range(self.width):
          # Die aktuelle Entropie bestimmen
          tile_entropy = self.world[row][col].entropy
          # Nur Kacheln betrachten, die noch nicht
          # kollabiert wurden
          if tile_entropy > 0:
            if tile_entropy < max_entropy:
              tile_list.clear()
              max_entropy = tile_entropy
```

```
            if tile_entropy == max_entropy:
              tile_list.append(self.world[row][col])
    return tile_list

  def collapse_wave_function(self):
    # Hole die Kachel(n) mit der geringsten Entropie.
    # Zu Beginn haben alle Kacheln die maximale Entropie.
    # Dann wird aus ihnen eine zufällig ausgewählt.
    tiles_with_lowest_entropy = self.get_tiles_with_lowest_entropy()
    # Wenn es keine Kachel mehr gibt, sind alle Kacheln besucht worden.
    if len(tiles_with_lowest_entropy) == 0:
      return 0
    if tiles_with_lowest_entropy == []:
      return 0

    tile_to_collapse = random.choice(tiles_with_lowest_entropy)
    # Setze für die zufällig ausgewählte Kachel einen konkreten Wert.
    tile_to_collapse.collapse()
    # Die nicht rekursive Version des Bactrackings braucht einen Stack.
    stack = Stack()
    stack.push(tile_to_collapse)

    while stack.is_empty() == False:
      # Hol eine Kachel oben vom Stack.
      actual_tile = stack.pop()
      # Welche Möglichkeiten gibt es hier?
      actual_possibilities = actual_tile.get_possibilities()
      # In welche Richtungen hat die Kachel Nachbarn?
      actual_directions = actual_tile.get_directions()
      # Erkunde jede Richtung ...
      for direction in actual_directions:
        # ... und hole dir den Nachbarn.
        neighbour = actual_tile.get_neighbour(direction)
        # Wenn der Nachbar noch nicht besucht wurde
        if neighbour.entropy != 0:
          reduce_possibilities =
          neighbour.constrain(actual_possibilities, direction)
          if reduce_possibilities == True:
            stack.push(neighbour)
    return 1
```

```
def tile_the_world(self):
  tile_level = [[0 for col in range(LEVEL_WIDTH)] for
  row in range(LEVEL_HEIGHT)]
  for row in range(WORLDHEIGHT):
    for col in range(WORLDWIDTH):
      for y in range(3):
        for x in range(3):
          if len(self.world[row][col].get_possibilities()) ==
            1:
            tile_level[row*3+y][col*3+x]=
      tileRaster[self.world[row][col].list_possibilities[0]][x+y*3]
          else:
            tile_level[row*3+y][col*3+x]=0
  return(tile_level)
```

Listing 14.4 »TileWorld« implementiert die Welt mittels WFC.

Die Klasse `TileWorld` koordiniert die prozedurale Generierung der Spielwelt. Sie nimmt als Parameter `width` und `height` entgegen – allerdings nicht in Einzelkacheln, sondern in Dreierblöcken, wie im vorherigen Abschnitt erläutert. Diese grobere Rasterung erlaubt dem WFC-Algorithmus eine größere strukturelle Vielfalt. Beim Initialisieren erzeugt der Konstruktor eine leere Liste `self.world`, die später das 2D-Raster von Tile-Objekten aufnimmt, und ruft anschließend `create_world()` auf.

Die Methode `create_world()` füllt `self.world` mit Tile-Objekten und bereitet spezielle Kachel-Sets vor, um bestimmte Regionen wie Boden oder Decke des Levels gezielt zu belegen. So werden beispielsweise `tiles_ceiling` mit allen Tiles gefüllt, deren obere drei Bits '000' sind, was auf ein leeres oberes Ende hindeutet. Ähnlich werden `tiles_floor` mit Boden-Tiles ohne Leiter- oder Seilkomponenten gefüllt, während `tiles_floor_ladder`, `tiles_ladder` und `tiles_ropes` spezifische Tile-Sets für Elemente mit Leitern und Seilen enthalten.

Das Raster wird anschließend Zeile für Zeile aufgebaut. Bestimmte Positionen, etwa rund um den Ausgang, werden bereits gezielt vorbelegt, um sicherzustellen, dass bestimmte Strukturelemente (z. B. eine Tür) existieren.

Innerhalb einer verschachtelten Schleife werden alle Tiles mit ihren direkten Nachbarn in Richtung Norden, Süden, Osten, Westen verbunden, sofern sie nicht am Rand liegen. Diese Beziehungen sind essenziell für die Constraint-Propagation im WFC-Prozess. Gleichzeitig wird wie oben erwähnt die oberste Reihe mit Decken-Kacheln aus `tiles_ceiling` und die unterste mit Boden- oder Leiter-Kacheln aus `tiles_floor` und `tiles_floor_ladder` vorbelegt.

Die Methode `get_entropy()` ermöglicht den einfachen Zugriff auf die Entropie einer bestimmten `Tile` im Weltraster, während `get_type()` den aktuell einzig möglichen Tile-Typ einer kollabierten `Tile` zurückgibt. Falls die `list_possibilities` einer Tile leer ist, deutet dies auf einen Fehler im WFC-Prozess hin, und die Welt wird neu generiert.

Die Methode `get_tiles_with_lowest_entropy()` durchsucht das gesamte Raster nach Kacheln mit der niedrigsten positiven Entropie und sucht so die besten Kandidaten für den nächsten Kollaps.

Die Methode `collapse_wave_function()` implementiert den eigentlichen WFC-Algorithmus. Sie wählt zufällig eine der Tiles mit der geringsten Entropie aus (bzw. die erste verfügbare, wenn alle die maximale Entropie haben) und kollabiert sie über die Methode `tile_to_collapse.collapse()`, wodurch ein konkreter Zustand gesetzt wird. Anschließend wird ein Stack verwendet, um die Auswirkungen dieses Kollapses iterativ auf die Nachbarn zu propagieren (eine nichtrekursive Backtracking-ähnliche Implementierung). Für jeden Nachbarn wird überprüft, ob sich seine Möglichkeiten durch die neue Einschränkung verringern (`neighbour.constrain(...)`). Ist das der Fall, wird auch dieser Nachbar dem Stack hinzugefügt. Der Prozess wiederholt sich, bis keine Änderungen mehr auftreten und der Stack leer ist.

Die Methode `tile_the_world()` überführt anschließend die grobe Struktur (`world`, bestehend aus 3×3-Blöcken) in das finale Level-Raster `tile_level`, das auf einzelnen Tiles basiert. Für jeden vollständig kollabierten Block wird mithilfe des `tileRaster`-Dictionarys das entsprechende 3×3-Muster in das Raster übertragen. Noch nicht kollabierte Blöcke werden als 0 markiert. So entsteht die endgültige Tile-Anordnung, auf der die spätere Spielwelt basiert.

14.7 Wegesuche mit A*

Stellen Sie sich vor, Ihr Spielcharakter muss von Punkt A zu Punkt B gelangen, vorbei an Hindernissen, über verschiedene Untergründe mit unterschiedlichen Bewegungskosten. Der *A*-Suchalgorithmus* übernimmt diese Aufgabe und findet den effizientesten Pfad – sowohl den kürzesten als auch den am wenigsten aufwendigen. Er funktioniert in einfachen Schritten:

1. Der Charakter befindet sich am Startpunkt (A).
2. Der Algorithmus prüft alle direkten Bewegungsmöglichkeiten vom aktuellen Punkt aus. Dies könnten die benachbarten Felder auf der Spielkarte sein.

3. Für jede dieser möglichen nächsten Bewegungen berechnet der Algorithmus zwei Werte, die addiert werden:
   - **die tatsächlichen Kosten, um von Startpunkt A bis zu diesem neuen Punkt zu gelangen**: Dies könnte die Anzahl der Schritte sein oder ein Wert, der die Schwierigkeit des Untergrunds berücksichtigt.
   - **eine Schätzung der Kosten, um vom aktuellen Punkt aus zum Zielpunkt B zu gelangen**: Diese Schätzung ist in der Regel eine vereinfachte Annahme, beispielsweise die direkte Entfernung oder eine grobe Einschätzung basierend auf der Position des Ziels. Sie kennen das von Black & White unter dem Namen *Heuristik*.
4. Der Algorithmus wählt als Nächstes die Bewegungsmöglichkeit, bei der die Summe dieser beiden Werte am niedrigsten ist. Er geht davon aus, dass dieser Weg bisher am effizientesten war und wahrscheinlich am schnellsten zum Ziel führt.
5. An diesem neuen Punkt wiederholt der Algorithmus den Vorgang: Er prüft alle weiteren Bewegungsmöglichkeiten, berechnet die Summe aus den bisherigen Kosten und den geschätzten Restkosten zum Ziel und wählt auf dieser Grundlage die vielversprechendste Bewegung.
6. Dieser Prozess wird fortgesetzt, bis der Charakter den Zielpunkt (B) erreicht hat.

Warum ist diese Vorgehensweise nützlich für Spiele?

- **Vermeidung von Umwegen:** Der Algorithmus merkt sich bereits besuchte Orte, um unnötige Schleifen zu vermeiden.
- **Zielorientierung:** Durch die Schätzung der Restkosten wird die Suche in Richtung des Ziels gelenkt, anstatt blind den gesamten Spielbereich zu erkunden.
- **Sie findet oft den optimalen Weg:** Wenn die Schätzfunktion die tatsächlichen Restkosten nie überschätzt, garantiert A* die Findung des kostengünstigsten Weges.

Der Buchstabe »A« ist Teil des Namens des Algorithmus. Das Sternchen »*« kennzeichnet eine Version des Algorithmus, die unter bestimmten Bedingungen den optimalen Pfad findet. A* kombiniert dabei die Stärken zweier Ansätze: die des *Dijkstra-Algorithmus* (kürzester Pfad durch bekannte Kosten) und der *Best-First Search* (zielgerichtetes Vorantasten per Heuristik).

Entwickelt wurde der *A*-Algorithmus* im Jahr 1968 von Peter Hart, Nils Nilsson und Bertram Raphael am Stanford Research Institute (heute SRI International). Er ist eine Weiterentwicklung des heuristischen Pfadfindungsalgorithmus *A1* von Nilsson und Raphael und wurde entwickelt, um die Effizienz der Suche in komplexen Problemräumen zu verbessern.

A* wird in einer Vielzahl von Anwendungen genutzt, bei denen es darum geht, den optimalen Pfad zwischen zwei Punkten zu finden. Dazu zählen:

- **Navigation und Routenplanung**: in GPS-Systemen, Online-Karten und Robotik zur Findung der besten Route zwischen Start- und Zielort
- **Spieleentwicklung**: für die Pfadfindung von Nicht-Spieler-Charakteren (NPCs) in komplexen Spielwelten
- **Künstliche Intelligenz**: in vielen KI-Anwendungen, die Planung, Robotik, Logistik und andere Bereiche umfassen, in denen ein optimaler oder nahezu optimaler Pfad gefunden werden muss
- **Logistik und Transport**: zur Optimierung von Lieferrouten und Fahrplänen
- **Netzwerkrouting**: zur Findung des effizientesten Datenübertragungspfads in Computernetzwerken
- **Künstliche Lebenssimulationen**: um die Bewegung von Agenten in simulierten Umgebungen zu steuern

A* gehört zu den *informierten Suchalgorithmen*: Im Gegensatz zu herkömmlichen Suchalgorithmen nutzt A* zusätzliche Informationen in Form einer Heuristikfunktion über das Problem. Diese Funktion liefert eine Schätzung, wie weit ein gegebener Knoten vom Ziel entfernt ist.

Die Entwicklung informierter Suchalgorithmen ist eng mit dem Aufkommen der künstlichen Intelligenz als Forschungsfeld verbunden. Frühe Arbeiten in den 1950er- und 1960er-Jahren betonten die Notwendigkeit, Problemlösestrategien zu entwickeln, die über das bloße systematische Durchsuchen aller Möglichkeiten hinausgehen. Die Idee, heuristische Funktionen einzusetzen, die eine Schätzung der Nähe zum Ziel liefern, war ein wichtiger Fortschritt.

Durch diese gezielte Vororientierung kann A* den Suchraum deutlich verkleinern und schneller zur Lösung führen. Entscheidend ist dabei, dass die Heuristik »zulässig« ist, also nie die tatsächlichen Restkosten überschätzt. In diesem Fall garantiert A* den optimalen Pfad.

14.8 A* Im Irrgarten

Bevor wir A* in unserem Platformer mit komplexen Bewegungsmöglichkeiten wie Klettern, Hangeln und Fallen einsetzen, beginnen wir mit einem einfacheren Szenario: einem klassischen Irrgarten. Hier kann sich der Spieler nur in vier Richtungen bewegen,

die alle mit gleichen Kosten versehen sind. Das ist ideal, um das Grundprinzip von A* zu verstehen. Hier sehen Sie den Pseudocode des A*-Algorithmus:

Repräsentation des Irrgartens

Der Irrgarten wird als *Graph* betrachtet. Jedes begehbare Feld ist ein *Knoten*. Benachbarte, begehbare Felder sind durch *Kanten* verbunden. Die Kosten jeder Kante sind in diesem Fall gleich (z. B. 1 für jeden Schritt zu einem benachbarten Feld).

Start- und Zielknoten

Der Startpunkt Ihres Charakters ist der *Startknoten*, und der Ausgang des Irrgartens ist der *Zielknoten*.

Bewertungsfunktion

Für jeden Knoten (jedes Feld, das der Algorithmus untersucht), wird die Bewertungsfunktion $f(n) = g(n) + h(n)$ verwendet:

- $g(n)$: Die tatsächlichen Kosten des bisher zurückgelegten Weges vom Startpunkt zum aktuellen Feld n. Dies ist einfach die Anzahl der Schritte, die der Charakter bisher gegangen ist.
- $h(n)$: Eine Heuristikfunktion, die die geschätzte Entfernung vom aktuellen Feld n zum Ausgang (Zielknoten) schätzt. Eine einfache und zulässige Heuristik für einen Irrgarten auf einem Gitter ist die Manhattan-Distanz, die Sie bereits in Kapitel 13, »Die Geister, die ich rief«, kennengelernt haben. Sie berechnet die Summe der horizontalen und vertikalen Distanzen zwischen dem aktuellen Feld und dem Ausgang, ohne die Wände zu berücksichtigen.

Open Set und Closed Set

Ein *Open Set* führt eine Liste aller Felder, die noch untersucht werden müssen. Diese Felder werden nach ihrem $f(n)$-Wert sortiert, wobei das Feld mit dem niedrigsten $f(n)$-Wert als Nächstes untersucht wird.

Ein *Closed Set* führt eine Liste aller Felder, die bereits untersucht wurden, um zu vermeiden, dass sie erneut bearbeitet werden.

Ablauf des Algorithmus

Beginnen Sie, indem Sie den Startknoten zum Open Set hinzufügen.

Solange das Open Set nicht leer ist,

- wählen Sie den Knoten im Open Set mit dem niedrigsten $f(n)$-Wert aus. Nennen Sie diesen den aktuellen Knoten.
- Entfernen Sie den aktuellen Knoten aus dem Open Set und fügen Sie ihn zum Closed Set hinzu.
- Wenn der aktuelle Knoten der Zielknoten (Ausgang) ist, wurde der Pfad gefunden. Rekonstruieren Sie den Pfad, indem Sie die gespeicherten Vorgängerknoten zurückverfolgen.
- Für jeden begehbaren Nachbarn des aktuellen Knotens:
  - Wenn der Nachbar nicht im Closed Set ist, berechnen Sie die Kosten zum Erreichen des Nachbarn über den aktuellen Knoten (g-Wert des Nachbarn).
  - Wenn der Nachbar noch nicht im Open Set ist oder der neue g-Wert niedriger ist als der bisherige g-Wert des Nachbarn, setzen Sie den Vorgängerknoten des Nachbarn auf den aktuellen Knoten. Berechnen Sie den f-Wert des Nachbarn ($f(n) = g(n) + h(n)$). Fügen Sie den Nachbarn zum Open Set hinzu (oder aktualisieren Sie seine Position im Open Set, falls der f-Wert gesunken ist).

Pfadrekonstruktion

Sobald der Zielknoten erreicht wurde, kann der optimale Weg zurückverfolgt werden, indem man von dem Zielknoten aus immer dem gespeicherten Vorgängerknoten folgt, bis man den Startknoten erreicht.

Damit haben wir die grundlegenden Voraussetzungen festgehalten. Durch die Kombination der tatsächlich zurückgelegten Schritte (g-Wert) mit einer Schätzung der verbleibenden Entfernung zum Ausgang (Manhattan-Distanz als h-Wert) priorisiert A* die Erkundung von Feldern, die sowohl auf einem vielversprechenden Weg liegen als auch dem Ausgang räumlich näher sind. Dies ermöglicht es dem Algorithmus, den Ausgang im Irrgarten effizienter zu finden als beispielsweise eine einfache Breitensuche oder Tiefensuche.

Abbildung 14.10 zeigt den Irrgarten, den wir mit A* erforschen wollen. Der Eingang befindet sich oben links, der Ausgang unten rechts. Die Pygame-Simulation `astar-v2.py` führt den Algorithmus Schritt für Schritt durch. Sie finden das Programm im Downloadordner des Buchkapitels.

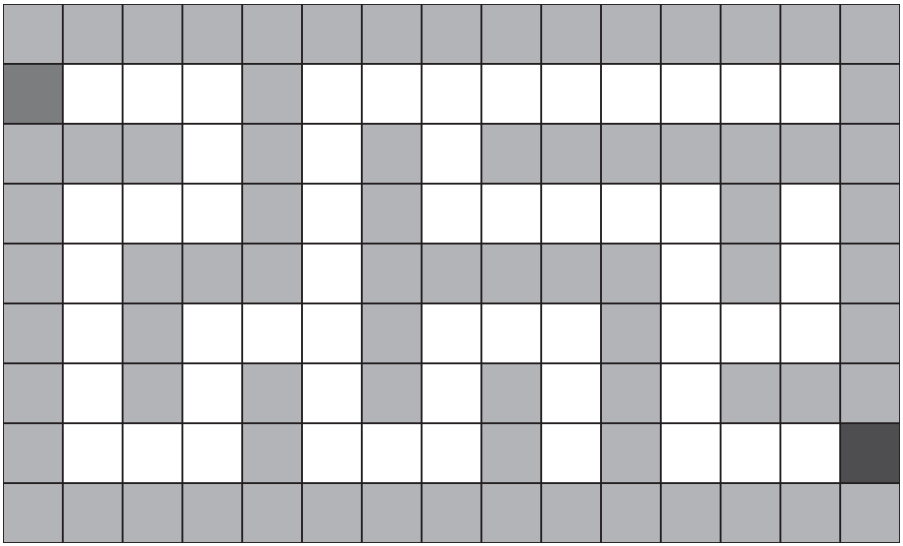

Abbildung 14.10 Der Irrgarten zu Beginn von A*

Im Folgenden bespreche ich mit Ihnen einige zentrale Aspekte des Programms. Alle Pygame-spezifischen Methoden sowie Teile, die nicht direkt mit dem A*-Algorithmus zusammenhängen, habe ich zur besseren Übersicht auskommentiert.

```
import pygame
import random
import heapq

# Farben für die Darstellung
BLACK  = (0,0,0)
WHITE  = (255,255,255)
GRAY   = (192,192,192)
CURRENT_GREEN = (255,64,0)
PATH_GREEN  = (0,255,0)
OPEN_GREEN = (0,128,0)
CLOSED_GREEN = (0,192,0)
BLUE   = (0,0,255)
RED    = (255,0,0)
BLANK  = (1,1,1)

# Die Maße des Irrgartens
GRID_WIDTH, GRID_HEIGHT = 15,9
TILE_SIZE = 64
```

```
SCREEN_WIDTH, SCREEN_HEIGHT = GRID_WIDTH*TILE_SIZE, GRID_HEIGHT*TILE_SIZE

FPS = 60

maze = [[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
        [0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
        [1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1],
        [1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1],
        [1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 1],
        [1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1],
        [1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1],
        [1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0],
        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]]

# x,y
start = (0, 1)
end = (14,7)

class Node():
  """A node class for A* Pathfinding"""

  def __init__(self, parent=None, position=None):
    self.parent = parent
    self.position = position

    self.g = 0  # Kosten vom Startpunkt bis hierhin
    self.h = 0  # Kosten vom Endpunkt bis hierhin
    self.f = 0  # Gesamtkosten (f = g + h)

  def __eq__(self, other):
    return self.position == other.position

  def __lt__(self, other):
    return self.f < other.f


class Game():

  def __init__(self):
    # Bitte im Quelltext nachsehen
```

```
def main(self):
  # ... Bitte im Quelltext nachsehen

  # A* zurücksetzen
  self.reset()

  # game loop
  # ... Bitte im Quelltext nachsehen

def reset(self):
  # Listen für besuchte und nicht besuchte Knoten anlegen.
  # Ich brauche die Liste global für die schrittweise Animation.
  self.open_list = []
  self.closed_list = []
  self.closed_dict = {}
  self.start_node = Node(None, start)
  self.end_node = Node(None, end)
  self.current_node = None
  self.state = GENERATION
  self.path = self.astar(maze, start, end)

def handle_input(self):
  # Handle Events
  # ... Bitte im Quelltext nachsehen

def update(self, dt):
  self.time_since_last_frame += (dt)
  if self.time_since_last_frame > self.milliseconds_per_frame:
    self.time_since_last_frame -= self.milliseconds_per_frame
    # A* ist in die Update-Methode eingebettet und wird hier Schritt
    # für Schritt ausgeführt. Sie kennen diese Vorgehensweise bereits
    # von den anderen Simulationen aus diesem Buch.
    if self.open_list and self.state == GENERATION:
      self.path = self.astar_step(maze)
      if self.path:
        self.state = RENDER

def render(self, surface):
  surface.fill(BLACK)
  self.render_maze(surface)
  pygame.display.update()
```

```
def render_node(self, surface, node, color):
  # Hier werden die einzelnen Knoten gezeichnet.
  if color != BLANK:
    pygame.draw.rect(surface, color, (node.position[0]*TILE_SIZE + 1,
      node.position[1]*TILE_SIZE + 1, TILE_SIZE - 2, TILE_SIZE - 2))

  # G-Wert (unten links)
  text_surface = self.font_gh.render(str(int(node.g)), True, (0, 0, 0))
  surface.blit(text_surface, (node.position[0]*TILE_SIZE + 5,
    node.position[1]*TILE_SIZE + TILE_SIZE - 20))

  # H-Wert (unten rechts)
  text_surface = self.font_gh.render(str(int(node.h)), True, (0, 0, 0))
  surface.blit(text_surface, (node.position[0]*TILE_SIZE + TILE_SIZE -
    15, node.position[1]*TILE_SIZE + TILE_SIZE - 20))

  # F-Wert (unten links)
  text_surface = self.font_f.render(str(int(node.f)), True, (0, 0, 0))
  surface.blit(text_surface, (node.position[0]*TILE_SIZE + 5,
    node.position[1]*TILE_SIZE + 5))

  # Pfeil, mit Richtung auf den Elternknoten. So kann der Weg
  # zurückverfolgt werden.
  if node.parent:
    destination = (node.parent.position[0] - node.position[0],
    node.parent.position[1] - node.position[1])
  else:
    destination = (0,0)
  if destination == (-1,0):
    text_surface = self.font.render('\u2190', True, (0, 0, 0))
  if destination == (0,-1):
    text_surface = self.font.render('\u2191', True, (0, 0, 0))
  if destination == (1,0):
    text_surface = self.font.render('\u2192', True, (0, 0, 0))
  if destination == (0,1):
    text_surface = self.font.render('\u2193', True, (0, 0, 0))
  if destination == (-1,-1):
    text_surface = self.font.render('\u2196', True, (0, 0, 0))
  if destination == (1,-1):
    text_surface = self.font.render('\u2197', True, (0, 0, 0))
```

```
    if destination == (1,1):
      text_surface = self.font.render('\u2198', True, (0, 0, 0))
    if destination == (-1,1):
      text_surface = self.font.render('\u2199', True, (0, 0, 0))
    surface.blit(text_surface, (node.position[0]*TILE_SIZE + 20,
      node.position[1]*TILE_SIZE + 10))

  def render_maze(self, surface):
    # Maze
    for y in range(GRID_HEIGHT):
      for x in range(GRID_WIDTH):
        if maze[y][x] == 0:
          pygame.draw.rect(surface, WHITE, (x*TILE_SIZE + 1,
            y*TILE_SIZE + 1, TILE_SIZE - 2, TILE_SIZE - 2))
        if maze[y][x] == 1:
          pygame.draw.rect(surface, GRAY, (x*TILE_SIZE + 1,
          y*TILE_SIZE + 1, TILE_SIZE - 2, TILE_SIZE - 2))

    # Startknoten
    pygame.draw.rect(surface, RED, (start[0]*TILE_SIZE + 1,
      start[1]*TILE_SIZE + 1, TILE_SIZE - 2, TILE_SIZE - 2))

    # Endknoten
    pygame.draw.rect(surface, BLUE, (end[0]*TILE_SIZE + 1,
      end[1]*TILE_SIZE + 1, TILE_SIZE - 2, TILE_SIZE - 2))

    if self.open_list:
      for node in self.open_list:
        if node.position != start and node.position != end:
          self.render_node(surface, node, OPEN_GREEN)

    for position, node in self.closed_dict.items():
      if position != start and position != end:
        self.render_node(surface, node, CLOSED_GREEN)

    if self.current_node:
      self.render_node(surface, self.current_node, CURRENT_GREEN)

    if self.path:
      for node in self.path:
        if node != start and node != end:
```

```
            self.render_node(surface, self.closed_dict[node],
              PATH_GREEN)

    def astar(self, maze, start, end):
      # Startknoten anlegen
      self.start_node = Node(None, start)
      self.start_node.g = self.start_node.h = self.start_node.f = 0

      # Endknoten anlegen
      self.end_node = Node(None, end)
      self.end_node.g = self.end_node.h = self.end_node.f = 0

      # Startknoten auf den Heap und los geht's!
      heapq.heapify(self.open_list)
      heapq.heappush(self.open_list, self.start_node)

      return None  # Wenn kein Pfad zum Ziel gefunden wurde

    def astar_step(self, maze):
      # Hol dir einen Knoten von der "offenen" Liste.
      self.current_node = heapq.heappop(self.open_list)

      if self.current_node in self.closed_list or
        self.current_node.position in self.closed_dict:
        print("Ende")
        return

      # Markiere ihn als besucht.
      self.closed_list.append(self.current_node)
      self.closed_dict[self.current_node.position] = self.current_node

      # Ist er bereits der Endknoten?
      # Dann bau den Pfad rückwärts auf und gib ihn zurück.
      if self.current_node == self.end_node:
        path = []
        current = self.current_node
        while current is not None:
          path.append(current.position)
          current = current.parent
        return path[::-1]
```

```
# Eine Liste der Nachbarn des Feldes anlegen
neighbors = [
(0, 1), (0, -1), (1, 0), (-1, 0),   # horizontal und vertikal
(1, 1), (1, -1), (-1, 1), (-1, -1)  # diagonale Richtungen
]

for neighbor in neighbors:
  neighbor_position = (self.current_node.position[0] + neighbor[0],
    self.current_node.position[1] + neighbor[1])

  # Befindet sich der Nachbar innerhalb der Grenzen?
  if (0 <= neighbor_position[0] < GRID_WIDTH and
    0 <= neighbor_position[1] < GRID_HEIGHT and
    maze[neighbor_position[1]][neighbor_position[0]] == 0):
    # 0 bedeutet "walkable".

    # Kosten berechnen.
    # Diagonal ist teurer (Wurzel 2).
    move_cost = 10
    if abs(neighbor[0]) == 1 and abs(neighbor[1]) == 1:
      move_cost = 140.14

    neighbor_node = Node(self.current_node, neighbor_position)

    # Wenn der Nachbar in der "geschlossenen Liste" ist, überspringen.
    if neighbor_node in self.closed_list or
      neighbor_node.position in self.closed_dict:
      continue

    # g-, h- und f-Werte berechnen
    neighbor_node.g = self.current_node.g + move_cost
    neighbor_node.h = (abs(neighbor_node.position[0] -
      self.end_node.position[0]) +
      abs(neighbor_node.position[1] -
      self.end_node.position[1]))*10
    neighbor_node.f = neighbor_node.g + neighbor_node.h

    # Wenn der Nachbar noch nicht in der offenen Liste ist,
    # hinzufügen.
    if self.add_to_open(self.open_list, neighbor_node):
     heapq.heappush(self.open_list, neighbor_node)
```

```
  def add_to_open(self, open_list, node):
    # Ist node in der offenen Liste?
    for n in open_list:
      if node == n and node.f > n.f:
        return False
    return True

# Call python astar.py from console
# Create a game object
# Call method run on the game object to run the game
if __name__ == "__main__":
  myGame = Game()
  myGame.main()
```

Wenn Sie das Programm *astar-v2.py* aus dem Downloadordner starten, öffnet sich ein Simulationsfenster und führt die Pfadsuche in dem bereits bekannten Irrgarten durch. Abbildung 14.11 zeigt das Ergebnis des Programmlaufs. Der Algorithmus hat einen (oder besser gesagt *den einen*) Weg zum Ausgang gefunden und im Irrgarten markiert.

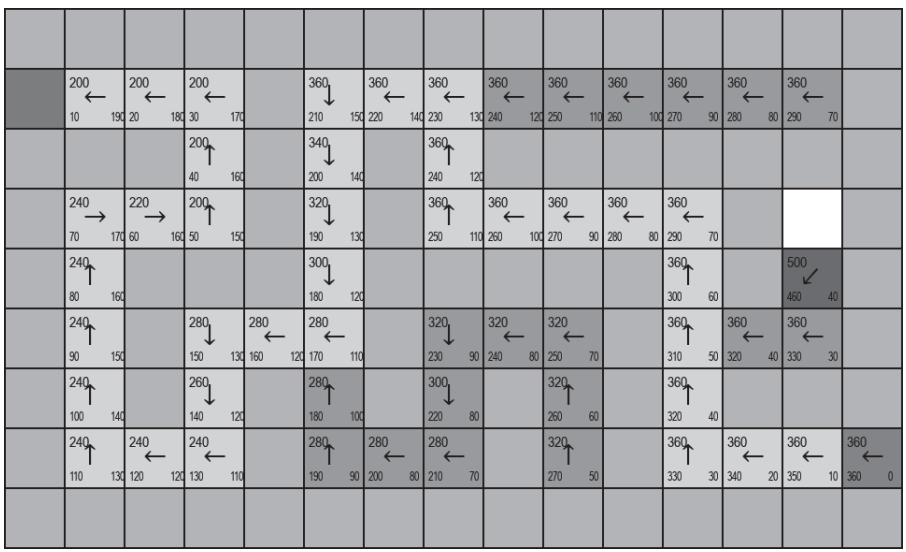

Abbildung 14.11 Der Irrgarten zum Ende von A*

14.9 Überblick über das Spiel

Die Hintergrundgeschichte von Tut's Treasure ist Ihnen bereits aus der Einleitung bekannt, ebenso wie die zugrunde liegenden Techniken von Wave Function Collapse (WFC) und A*. Deshalb möchte ich mich an dieser Stelle auf einige besondere Aspekte des Spiels konzentrieren – sowohl im Hinblick auf das Design als auch auf die Frage, wie sich bekannte Konzepte aus früheren Projekten kreativ neu interpretieren lassen.

14.9.1 Menü

Tut's Treasure startet mit einem Menübildschirm, den Sie bereits aus Black & White kennen – diesmal allerdings mit neuen Grafiken und einem *Parallax*-Scrolling-Effekt (siehe Abbildung 14.12).

Abbildung 14.12 Der Startbildschirm von »Tut's Treasure«

Die Grafiken stammen größtenteils aus dem Asset-Paket *Egyptian Environment* von Game Developer Studios [3]. Um einen einheitlichen Stil zu wahren, habe ich auch die Schriftrolle dort erworben [5]. Das *Ankh*, wie das ägyptische Radkreuz genannt wird, habe ich in Corel Draw selbst gezeichnet und als PNG eingebunden. Als Zeichensatz kommt *Andaluz* von StormTypeFoundry zum Einsatz [4]. Aufgrund der Lizenzbedingungen wurde der Text nicht direkt im Spiel gerendert, sondern in Photoshop als Bitmap erzeugt und eingebettet – eine Weitergabe des Fonts ist nicht gestattet.

14.9.2 Der Level-Editor

Da die WFC-Generierung nicht immer ein spielbares Ergebnis liefert, habe ich einen konventionellen Level-Editor in Tut's Treasure integriert (siehe Abbildung 14.13). Die Bedienung erfolgt über Tastaturkommandos, und mit der Maus lässt sich eine Zelle gezielt ansteuern und aktivieren.

Ein gültiger Level muss genau einen Spieler, einen Ausgang sowie Artefakte (Katzen) und Mumien enthalten. Eine automatische Plausibilitätsprüfung ist derzeit noch nicht implementiert. Es wird beim Speichern also nicht geprüft, ob der Aufbau des Levels tatsächlich spielbar ist.

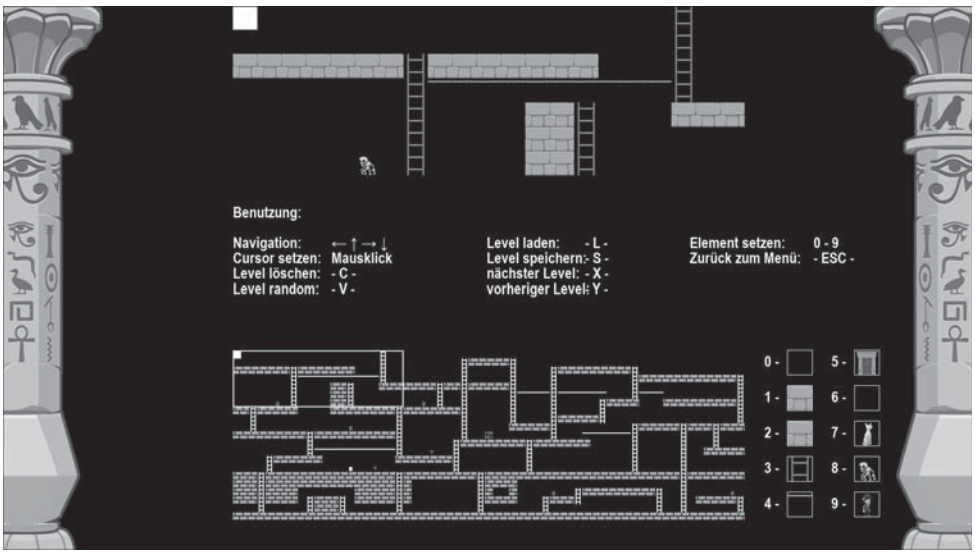

Abbildung 14.13 Der Level-Editor

Einige Aspekte des Editors möchte ich besonders hervorheben. Die Cursorsteuerung basiert auf einem kleinen Zustandsautomaten, wie Sie ihn bereits von Eliza kennen. Die Logik finden Sie in der Methode `update()`. Ebenfalls in `update()`, ganz am Anfang, wird eine Besonderheit implementiert: Die Tür in der Mitte des Levels wird bei jedem Aufruf automatisch gesetzt, um einen festen Wiedererkennungswert und Ankerpunkt zu erzeugen. Diese Position kann vom Spieler nicht überschrieben werden.

```
def update(self, dt):
  self.level.data[9][30] = TILE_VOID
  self.level.data[9][31] = TILE_VOID
  self.level.data[9][32] = TILE_VOID
  self.level.data[10][30] = TILE_VOID
```

```
    self.level.data[10][31] = TILE_EXIT
    self.level.data[10][32] = TILE_VOID
    self.level.data[11][30] = TILE_CONCRETE
    self.level.data[11][31] = TILE_CONCRETE
    self.level.data[11][32] = TILE_CONCRETE

    self.cursor_timer += dt
    if self.cursor_timer >= self.cursor_interval:
      self.cursor_timer -= self.cursor_interval
      self.cursor = not self.cursor
    self.camera.set_center(self.x, self.y)
    return super().update(dt)

  def render(self, surface):
    surface.blit(self.pillar_left,(0,0))
    surface.blit(self.pillar_right,(1719,0))

    for row in range(LEVEL_HEIGHT):      # Radar
      for col in range(LEVEL_WIDTH):
        surface.blit(self.radar_tiles,
          (RADAR_TILE_SIZE*col + self.radar_offset_x,
           RADAR_TILE_SIZE*row + self.radar_offset_y),
          (RADAR_TILE_SIZE*self.level.data[row][col], 0,
           RADAR_TILE_SIZE, RADAR_TILE_SIZE)
        )

    if self.cursor:
      pygame.draw.rect(surface, WHITE,
        (self.x*RADAR_TILE_SIZE + self.radar_offset_x,
         self.y*RADAR_TILE_SIZE + self.radar_offset_y,
         RADAR_TILE_SIZE, RADAR_TILE_SIZE))

    for row in range(7):       # Close-Up
      for col in range(21):
        surface.blit(self.edit_tiles,
          (EDIT_TILE_SIZE*col + self.radar_offset_x,
           EDIT_TILE_SIZE*row ),
          (EDIT_TILE_SIZE*self.level.data[row + self.camera.y]
          [col + self.camera.x], 0, EDIT_TILE_SIZE, EDIT_TILE_SIZE)
        )
```

```
    if self.cursor:
      pygame.draw.rect(surface, WHITE,
        ((self.x - self.camera.x)*EDIT_TILE_SIZE +
        self.radar_offset_x,
        (self.y - self.camera.y)*EDIT_TILE_SIZE,
        EDIT_TILE_SIZE, EDIT_TILE_SIZE))
```

Listing 14.5 Die Kernelemente des Level-Editors

Auch hier gibt es eine Referenz auf die `Camera`-Klasse. Die `Camera` wird benötigt, um wie mit einem Blick durch ein Teleobjektiv den Levelausschnitt im Level-Editor zeichnen zu können. Sie finden in der Methode `update()` diese Zeile Code:

```
self.camera.set_center(self.x, self.y)
```

Hier wird die aktuelle Cursor-Position genommen und die Kamera daran zentriert. Basierend auf den Koordinaten der Kamera, die sich aus dieser Zentrierung ergeben (linke obere Ecke, Breite, Höhe) wird dann in der `render()`-Methode der entsprechende Ausschnitt des Levels vergrößert im Editor dargestellt:

```
# Close-Up
for row in range(7):
  for col in range(21):
    surface.blit(self.edit_tiles,
      (EDIT_TILE_SIZE*col + self.radar_offset_x,
       EDIT_TILE_SIZE*row ),
      (EDIT_TILE_SIZE*self.level.data[row + self.camera.y]
      [col + self.camera.x], 0, EDIT_TILE_SIZE, EDIT_TILE_SIZE)
      )
```

Dazu wird die linke obere Ecke der Kameraposition als Offset genutzt, um die aktuellen Elemente im Level-Array zu finden.

14.9.3 Die Level-Auswahl

Über das Menü hat der Spieler die Möglichkeit, sich für einen per WFC generierten oder über den Level-Editor eingegebenen Level zu entscheiden (siehe Abbildung 14.14).

Abbildung 14.14 Die Level-Auswahl

14.9.4 Der Parallax-Effekt

Tut's Treasure nutzt einen *Parallax*-Effekt, um der Spielwelt trotz 2D-Grafik räumliche Tiefe zu verleihen. Die Technik stammt aus der Computergrafik und basiert auf einem einfachen Prinzip: Bewegung in verschiedenen Geschwindigkeiten und Ebenen erzeugt Tiefe.

Wenn Sie sich in der echten Welt bewegen, erscheinen nahe Objekte schneller an Ihnen vorbeizuziehen als weiter entfernte – etwa beim Blick aus einem fahrenden Zug. Genau dieses Phänomen der Bewegungsparallaxe wird im Spiel genutzt, um die Illusion von Raum zu erzeugen.

Die Hintergrundszene in Tut's Treasure (siehe Abbildung 14.15) besteht nicht aus einem einzigen Bild, sondern aus sieben separaten Ebenen, die sich beim Scrollen horizontal mit unterschiedlicher Geschwindigkeit bewegen.

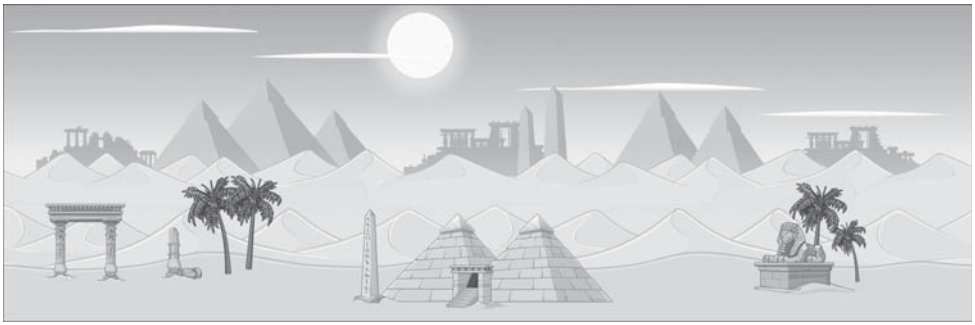

Abbildung 14.15 Der Hintergrund von »Tut's Treasure«

Die Vordergrundebenen mit den Palmen, Ruinen und Sanddünen bewegen sich schneller, die Hintergrundebenen mit den Pyramiden und dem Himmel bewegen sich langsamer. Dadurch entsteht beim Spieler der Eindruck, dass die Landschaft räumlich gestaffelt ist, obwohl sie flach ist. Je mehr Ebenen eingesetzt werden, desto realistischer und plastischer wirkt der Parallax-Effekt. Insgesamt ergibt sich das harmonische Bild einer Wüstenszene, die sich hinter dem Spiel bewegt. In der Tat besteht die Szene in Abbildung 14.15 aber aus mehreren Ebenen, wie Sie in Abbildung 14.16 sehen können.

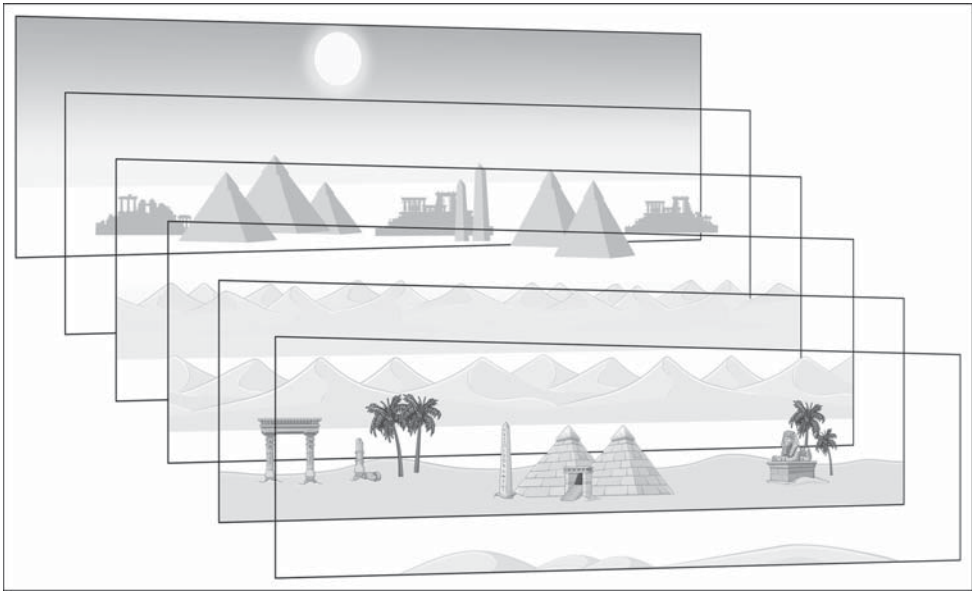

Abbildung 14.16 Die einzelnen Ebenen des Hintergrunds

Die Implementierung des Effekts erfolgt in der Datei *parallax.py* im Ordner `view`. Dort wird gesteuert, wie schnell sich jede Ebene im Verhältnis zur Spielerbewegung verschiebt:

```
import pygame

from model.constants import *

PARALLAX_WIDTH = 3840
PARALLAX_HEIGHT = 1080

class Parallax():
  def __init__(self):
    self.bg_scroll_images = []
    self.bg_scroll_speed = []
    self.bg_scroll_offset = []
    self.bg_y_offset = []
    self.scroll = 0
    for i in range(7):
      self.bg_scroll_images.append(pygame.image.load("./assets/BG/
      BG-layer-"+str(7-i)+".png").convert_alpha())

    speed = 0.4
    for i in range(7):
      self.bg_scroll_speed.append(speed)
      speed += 0.4
    self.bg_scroll_speed[0]=0
    for i in range(7):
      self.bg_scroll_offset.append(0)

    self.bg_y_offset.append(0) # Himmel
    self.bg_y_offset.append(72)
    self.bg_y_offset.append(264)
    self.bg_y_offset.append(532)
    self.bg_y_offset.append(598)
    self.bg_y_offset.append(577)
    self.bg_y_offset.append(978) # Sand im Vordergrund

    self.parallax_screen = pygame.Surface((SCREEN_WIDTH, SCREEN_HEIGHT),
      pygame.SRCALPHA, 32).convert_alpha()
```

```
def update_menu(self, dt):
  self.scroll -= 1
  self.scroll_layers()

def update_game(self, dt, delta):
  self.scroll = delta
  self.scroll_layers()

def scroll_layers(self):
  for i in range(7):
    self.bg_scroll_offset[i] = self.scroll * self.bg_scroll_speed[i]

def render(self, surface):
  for i, layer in enumerate(self.bg_scroll_images):
    self.parallax_screen.blit(layer, ((self.bg_scroll_offset[i]) %
      PARALLAX_WIDTH, self.bg_y_offset[i]))
    self.parallax_screen.blit(layer, ((self.bg_scroll_offset[i]) %
      PARALLAX_WIDTH - PARALLAX_WIDTH, self.bg_y_offset[i]))
  surface.blit(self.parallax_screen,(0,0), (0,0,1920,1080))
```

Listing 14.6 Die Klasse »Parallax«

Die Klasse `Parallax` wird sowohl im Menü als auch im eigentlichen Spiel verwendet. Im Spiel richtet sich die Bewegung der Ebenen nach der Spielerposition, während im Menü eine automatisierte Steuerung für den Parallax-Effekt sorgt. Aus diesem Grund existieren zwei unterschiedliche `update()`-Methoden.

Um die Performance zu optimieren, wurden die Grafiken nur in der jeweils benötigten Größe gezeichnet. Viele Ebenen bedecken also nicht die volle Bildschirmhöhe.

In der Methode `init()` werden alle relevanten Informationen in Listen organisiert: Bilder, deren Größen, Bildschirmpositionen und Scrollgeschwindigkeiten. Dadurch lassen sich in `scroll_layers()` alle sieben Ebenen bequem über einzelne `for`-Schleife bewegen. Auch in der Methode `render()` findet sich nur eine einzelne Schleife zur Bearbeitung aller sieben Ebenen.

Die verwendeten Bilder stammen aus dem Gesamtpaket E*gyptian Environment* von Robert Brooks [3], einem talentierten Künstler, dessen Arbeit ich sehr schätze. Robert hat mir freundlicherweise die Erlaubnis gegeben, Screenshots seiner Assets in diesem Buch zu verwenden.

14.9.5 Das Kamera-System

Tut's Treasure ist mehr als ein klassischer Platformer oder einfacher Side-Scroller, es ist eine eigene kleine Welt. Der Level ist größer als der sichtbare Bildschirm, sodass stets nur ein Ausschnitt dargestellt wird. Bewegt sich der Spieler, folgt ihm eine virtuelle Kamera, die den sichtbaren Bereich steuert (siehe Abbildung 14.17).

Die Kamera selbst zeichnet allerdings nichts, sondern verwaltet lediglich, welcher Teil der Welt aktuell sichtbar ist. Dafür benötigt sie eine Startposition (`x`,`y`) sowie eine Breite und eine Höhe.

Da die Kamera auch in der Radar-Karte – liebevoll Katzen-Kompass genannt – zum Einsatz kommt, werden weitere Startpunkte (`radar_offset_x`, `radar_offset_y`) für die Miniaturdarstellung benötigt.

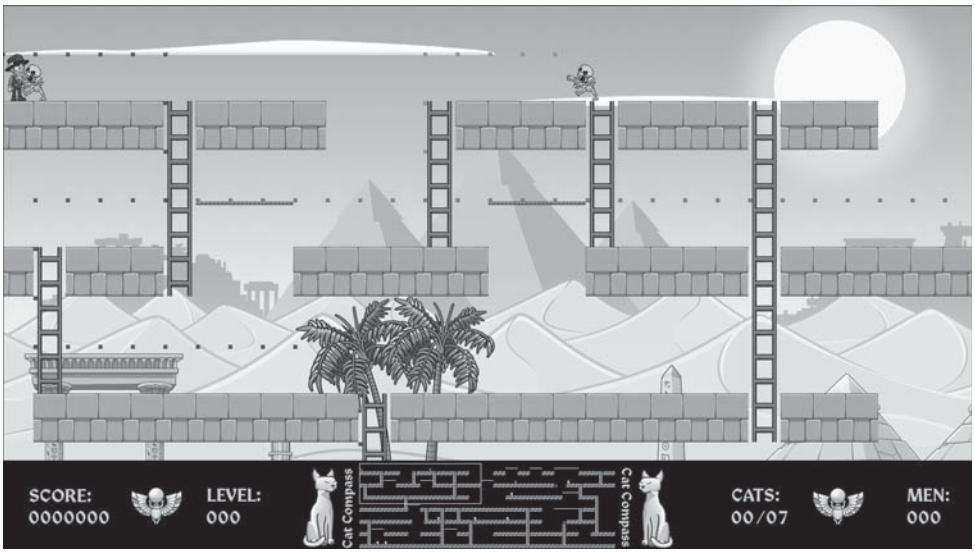

Abbildung 14.17 Die Kamera filmt einen Ausschnitt.

Der folgende Code zeigt die Implementierung der Kameraklasse:

```
class Camera():
  def __init__(self, x, y, width, height, radar_offset_x, radar_offset_y):
    self.x = x
    self.y = y
    self.width = width
    self.height = height
    self.center = (x + width//2, y + height // 2)
```

```
    self.offset_x = radar_offset_x
    self.offset_y = radar_offset_y
    self.startpos_x = x + GAME_TILE_SIZE//2 -
     (self.width*GAME_TILE_SIZE)//2
    self.startpos_y = y + GAME_TILE_SIZE_H//2 -
     (self.height*GAME_TILE_SIZE_H)//2

  def update(self, dt):
    pass

  def render(self, surface):
      pass

  def set_center(self, x, y):
    self.center = (x,y)
    self.x = x + GAME_TILE_SIZE//2 - (self.width*GAME_TILE_SIZE)//2
    self.y = y + GAME_TILE_SIZE_H//2 - (self.height*GAME_TILE_SIZE_H)//2
    if self.x < 0:
      self.x = 0
    if self.x + self.width*GAME_TILE_SIZE > LEVEL_WIDTH_PIXEL:
      self.x = LEVEL_WIDTH_PIXEL - self.width*GAME_TILE_SIZE
    if self.y < 0:
      self.y = 0
    if self.y + self.height*GAME_TILE_SIZE_H > LEVEL_HEIGHT_PIXEL:
      self.y = LEVEL_HEIGHT_PIXEL - self.height*GAME_TILE_SIZE_H

  def get_delta(self):
    return self.startpos_x - self.x
```

Listing 14.7 Die Klasse »Camera«

Die Methdoe `set_center()` ist das Herzstück der `Camera`-Klasse. Sie sorgt dafür, dass der Spieler möglichst im Zentrum des Bildausschnitts bleibt. Bewegt sich der Spieler, folgt ihm die Kamera, allerdings nur so lange, wie er nicht zu nah an den Rand der Spielwelt kommt. Dort stoppt die Kamera, sodass der Spieler bis zum Rand weiterlaufen kann.

Die Methode `get_delta()` berechnet, wie viele Pixel die Kamera horizontal von ihrer Startposition abgewichen ist. Dieser Wert wird an die Klasse `Parallax` übergeben, um den Wüstenhintergrund synchron zur Bewegung des Spielers zu scrollen.