21. Bundeswettbewerb Informatik 2002/2003, 2. Runde

Lösungs- und Bewertungshinweise

Allgemeines

Bewertungsbögen Kein Kreuz in einer Zeile bedeutet, dass der entsprechende Aufgabenteil korrekt bearbeitet wurde. Häufig wurde nur das vermerkt, was von der korrekten Bearbeitung abweicht. Ein Kreuz in der Spalte „+“ bedeutet in der Regel Zusatzpunkte, ein Kreuz unter „ok“ bedeutet eine gute Lösung im Rahmen der korrekten Bearbeitung (also meist ohne Zusatzpunkte; für manche Dinge gab es bestenfalls ein „ok“) und ein Kreuz unter „-“ bedeutet Minuspunkte für Fehlendes oder Unzulängliches.

Termin der 2. Runde Es vermerken immer wieder Teilnehmer, dass die 2. Runde parallel zum Abitur liegt. Das ist uns bekannt und sicher nicht ideal, lässt sich aber leider nicht ändern. In der 2. Jahreshälfte läuft die 2. Runde des Mathewettbewerbs, dem wir keine Konkurrenz machen wollen. Also bleibt uns nur die erste Jahreshälfte. Und damit liegt der Abgabetermin der 2. Runde immer in der Zeit der Abiturtermine. Aber: Sie haben etwa vier Monate Bearbeitungszeit für die 2. Runde. Rechzeitig mit der Bearbeitung der Aufgaben zu beginnen ist der beste Weg, Konflikte mit dem Abitur zu vermeiden.

Dokumentation Eine Dokumentation beginnt nicht mit: „Die Prozedur abc übergibt einen Zeiger p auf ein Feld xy, worauf die Funktion f ...“. Wie in den allgemeinen Hinweisen zu den Aufgaben gesagt, sollte die Dokumentation mit der Idee beginnen, die Sie zur Lösung der jeweiligen Aufgabe entwickelt haben. Schildern Sie diese Lösungsidee erst grob und gehen dann darauf ein, wie Sie sie in ein abstraktes, computertaugliches Modell (in Form von Algorithmen und Datenstrukturen) umgesetzt haben. Ihre Implementierung dieses Modells als Programm beschreiben Sie anschließend in der Programmdokumentation, die die wichtigsten Funktionen, Variablen, Klassen, Objekte etc. in Bezug auf die Lösungsidee dokumentiert und idealerweise mitteilt, wo diese Komponenten im Quellcode zu finden sind. Beachten Sie: Wer nicht in der Lage ist, Idee und Modell präzise zu formulieren, bekommt auch keine saubere Umsetzung in welche Programmiersprache auch immer hin.

Lösungshinweise Bei den folgenden Erläuterungen handelt es sich um Vorschläge, nicht um die einzigen Lösungswege, die wir gelten ließen. Wir akzeptieren in der Regel alle Ansätze, die die gestellte Aufgabe vernünftig lösen und entsprechend dokumentiert sind. Einige Dinge gibt es allerdings, die – unabhängig vom gewählten Lösungsweg – auf jeden Fall diskutiert werden müssen.

Aufgabe 1: Restgeld

Allgemeine Hinweise

Diese Aufgabe enthält wenige grundsätzliche Schwierigkeiten, weshalb vor allem die Qualität der Überlegungen, des Designs und der Algorithmen von Bedeutung ist. Laut Aufgabenstellung ist nur eine Beschreibung der Automaten-Klasse, ihrer Methoden und der benötigten Algorithmen verlangt, aber keine Implementierung. Ein lauffähiges Programm ist daher ausnahmsweise eine Erweiterung – aber auch nicht mehr; für die Bewertung war (mehr noch als sonst) der schriftliche Teil der Einsendung entscheidend. Aber, ob mit oder ohne Umsetzung von Modell und Algorithmen in ein Programm: Das Funktionieren der beschriebenen Ideen sollte durch Beispiele demonstriert werden, auch wenn sie nur „trocken“ und ansatzweise durchexerziert werden.

Neben der Qualität des Beschriebenen musste auch die Qualität der Beschreibung in die Bewertung einer solchen Aufgabe besonders einfließen. Es gibt viele Möglichkeiten, Objektmodelle und Algorithmen verständlich zu beschreiben, seien es grafische Notationen oder deklarative Teile der Klassenbeschreibung in einer objektorientierten Programmiersprache für objektorientierte Strukturen, seien es Diagrammnotationen oder Pseudo-Code für Algorithmen. Sinnvoll ist die Verwendung von Möglichkeiten, die zwischen natürlicher Sprache und (detailliertem) Quellcode angesiedelt sind.

Teil 1: Modellierung der Automaten

In dieser Teilaufgabe ist eine Modellierung als *Klasse* gefordert, was insbesondere bedeutet, dass es auf die Identifikation von Attributen und Methoden, Datenkapselung (also Unterscheidung von öffentlichen und nicht-öffentlichen Bestandteilen) und Trennung von Interface und Implementierung ankommt. Methoden wie `BerechneRueckgeld` sind beispielsweise im öffentlichen Interface fehl am Platze, und auch der Vorrat an Wechselgeld sollte (außer für Wartungspersonal) nicht zugänglich sein.

Außerdem sollen tatsächlich die Automaten und nicht etwa nur die Steuerungslogik modelliert werden. Als *grobes Kriterium* kann man ansetzen, dass es möglich sein sollte, eine passende „Fahrgast“-Klasse zu definieren, deren Objekte an einem Automaten Fahrscheine kaufen können. Mit ein paar Variablen und einer Methode zum Berechnen des Rückgeldes ist es also im Normalfall nicht getan. Hier ist ein Beispiel für eine mögliche Modellierung:

Interface

1. *Abfrage von Informationen:* Kann der Automat wechseln (leuchtet die Warnanzeige)? Welcher Fahrschein wurde zuletzt gewünscht? Wie viel Geld muss noch eingeworfen werden? Liegt ein Fahrschein in der Ausgabe? Liegt Wechselgeld in der Ausgabe?

2. *Interaktion mit dem Automaten:* Fordere einen Fahrschein an; werfe Geld ein; breche den Kauf ab.

Implementierung

1. *Interner Zustand des Automaten:* Hierzu gehören das vorhandene Wechselgeld, der momentane Betriebsmodus des Automaten, das bisher vom Kunden eingeworfene Geld und im Ausgabefach liegende Fahrscheine, Münzen und Geldscheine. Häufig wurde von den Teilnehmern gefragt, ob Münzen und Scheine unterschieden werden müssen. Hierzu gab es keine Vorgabe, die Teilnehmer sollten selbst beurteilen, ob die Unterscheidung relevant ist. Sie ist es zunächst nicht und wird es nur dann, wenn eine der für Teil 4 entwickelten Strategien auf den Unterschied zwischen Münzen und Scheinen zurückgreift. Eine generelle Unterscheidung zwischen Münzen und Scheinen ist nicht sinnvoll.

2. *Methoden für die Implementierung benötigt (optional):* Berechnung des Wechselgeldes; Steuerung der Warnanzeige.

Ein solches Modell kann u.a. mit Hilfe von (Pseudo-)Quellcode (sinnvoll sind aber rein deklarative Code-Elemente) beschrieben werden. Hier eine Beschreibung der oben genannten öffentlichen Methoden:

```
public interface Automat {
    // Statusabfrage
    public boolean kannWechseln();
    public int gewuenschterFahrschein();
    public int nochFehlendesGeld();

    // Interaktion mit dem Automaten
    public int nimmFahrschein();
    public int nimmWechselgeld();
    public void wirfMuenzeEin (int betrag);
    public void wirfScheinEin (int betrag);

    // ... usw. ...
}
```

Wie auch immer die Modellierung im einzelnen ausgefallen ist: Wichtig ist (wie immer beim BWINF), den gewählten Entwurf und die beim Entwurf getroffenen Entscheidungen ausreichend zu begründen.

Teil 2: Berechnung des Wechselgeldes

Das Problem, ein Rückgeld zwischen 0 und 99 Fanten mit den im Automaten befindlichen Münzen und Scheinen herauszugeben, ist eine Variante des bekannten Rucksack-Problems. Ganz analog dazu kann man mit dynamischer Programmierung einen Algorithmus konstruieren, der entscheidet, ob und wie das Rückgeld herausgegeben werden kann. Für eine Beschreibung des Verfahrens siehe etwa R. Sedgewick, *Algorithmen*, oder U. Manber, *Introduction to Algorithms*.

Münzen und Scheine brauchen nicht unterschieden werden. Zu beachten ist bei der Umsetzung lediglich, dass Münzen und Scheine eines bestimmten Wertes mehrfach auftreten können.

Ein Lösungsalgorithmus, der auf naivem Backtracking beruht, ist auf modernen Rechnern sicher schnell genug, aber wenn keine Gedanken darauf verwandt wurden, den Suchraum zu begrenzen, wurden hier Punkte abgezogen.

Die Ein- und Ausgabevereinbarungen und der Algorithmus selbst sollten insbesondere Randfälle abdecken: Was passiert, wenn kein passendes Rückgeld herausgegeben werden kann? Funktioniert der Algorithmus auch, wenn er Null Fanten herausgeben soll?

Bei dieser wie auch bei der nächsten Teilaufgabe ist die Vollständigkeit und die Verständlichkeit der Beschreibung (von Eingabe- und Ausgabevereinbarung und Algorithmus) entscheidend. Auch hier sollten Entwurfsentscheidungen begründet sein. Schließlich stellt sich noch die Frage nach dem „Programmablaufprotokoll“. Da es kein Programm gibt, entfällt dies im engeren Sinne. Aber: Ein Programmablaufprotokoll soll die Funktionsfähigkeit eines Programms demonstrieren, und hier ist die Funktionsfähigkeit eines nur auf Papier vorhandenen Modells zu demonstrieren. Es ist deshalb auch bei einer solchen Aufgabe nötig, wie einleitend schon gesagt, Algorithmen nicht nur zu beschreiben, sondern sie auch anhand von Beispielen zumindest „trocken“ vorzuführen. Genau genommen handelt es sich dabei um den Entwurf von Testfällen: Eingabedaten und eine Beschreibung dessen, wie der Algorithmus mit diesen Daten arbeitet.

Teil 3: Warnanzeige

Zur Lösung dieser Teilaufgabe kann prinzipiell derselbe Algorithmus verwendet werden wie in Teil 2, der dann alle möglichen Rückgeldwerte (also zwischen 0 und 99) prüfen muss. Eine umfangreiche Diskussion der Ein- und Ausgabevereinbarungen ist hier nicht nötig. Aber auch hier wurde mindestens ein Beispiel erwartet.

Teil 4: Strategien für sinnvolle Geldrückgabe

Die vierte Teilaufgabe fragt nach zwei verschiedenen Dingen: Zum einen sollen *Kriterien* gefunden werden, nach denen die verschiedenen Möglichkeiten der Restgeldrückgabe bewertet werden können. Da es in der Praxis aber zu aufwändig sein kann, unter allen Möglichkeiten die beste herauszusuchen, sollen zusätzlich *Strategien* (etwa in Form von Heuristiken oder Änderungen des bisher verwendeten Algorithmus) beschrieben werden, mit denen sich die Kriterien möglichst gut erfüllen lassen.

Bei der Auswahl der Kriterien kommt es vor allem auf eine sinnvolle Begründung an: je nach Betrachtungsweise kann es beispielsweise sowohl besser als auch schlechter sein, Scheine statt Münzen oder viele kleine statt wenige große Scheine/Münzen herauszugeben. Zwei Beispiele für einfache Kriterien sind:

- Vermeide Situationen, in denen der Automat nicht mehr jeden Betrag herausgeben kann (Strategie: bevorzuge mehrfach vorhandene Münzen/Scheine).

- Versuche möglichst wenige Münzen/Scheine herauszugeben, um den Fahrgästen Taschen voller Kleingeld zu ersparen (Strategie: beginne Suche bei großen Münzen/Scheinen).

Dieser letzte Aufgabenteil lässt viel Raum für interessante Abwandlungen des Standard-Algorithmus. Besonders kreative oder anspruchsvolle Verfahren konnten mit Zusatzpunkten honoriert werden. Erwartet wird hingegen wiederum die Angabe von Beispielen und die Beschreibung der (Soll-)Funktionalität der Strategien.

Erweiterungen

Wie zu Beginn erwähnt, ist schon eine lauffähige Implementierung bereits als kleine Erweiterung zu rechnen. Die naheliegendsten „richtigen“ Erweiterungen sind in diesem Zuge eine vollständige Simulation mit Fahrscheinkäufern und dokumentierte Programmläufe. Interessant wäre es auch, verschiedene Wechselgeld-Strategien über längere Zeiträume zu untersuchen und miteinander zu vergleichen.

Aufgabe 2: Brandubh

Datenstruktur

Als Datenstruktur zur Repräsentation des Spielbretts drängt sich geradezu ein 2-dimensionales Array der Größe 7x7 auf. Für jedes einzelne Spielfeld gibt es die vier Zustände leer, weißer Getreuer, König und Belagerer. Eine Unterscheidung der „verschiedenen“ weißen bzw. schwarzen Figuren ist nicht sinnvoll, weil nicht benötigt; insbesondere gilt laut BWINF-FAQ beim Vergleich zweier Stellungen Folgendes: „Zwei Stellungen sind gleich, wenn Figuren gleichen Typs dieselben Felder besetzen ... .“ Um lästige if-Abfragen insbesondere beim Schlagen und bei der Prüfung der Gefangennahme des Königs zu vermeiden, bietet es sich an, intern ein 11x11 Array zu speichern, d.h. das Spielbrett wird auf jeder Seite jeweils um zwei leere Pseudospielfelder ergänzt.

Grundsätzlicher Ablauf eines Spielzuges

1. Start- und Zielfeld entgegennehmen
2. Regeln prüfen
3. Zug ausführen
4. ggf. schlagen
5. Spielende prüfen
6. Zug protokollieren

1. Start- und Zielfeld entgegennehmen

Für die Eingabe des Start- und des Zielfeldes sind verschiedene Möglichkeiten denkbar:

- erst das Start- und dann das Zielfeld mit der Maus anklicken
- Figur mit Drag and Drop vom Start- zum Zielfeld ziehen
- Koordinaten des Start- und des Zielfeldes eintippen
- Cursortasten zur Auswahl des Start- und des Zielfeldes nutzen

Am komfortabelsten ist sicherlich eine Kombination mehrerer dieser Möglichkeiten.

Auf jeden Fall sollte sichergestellt werden, dass nur Spielfelder gewählt werden können, die auch existieren. Ein Klicken neben das Spielbrett oder die Eingabe einer ungültigen Koordinate darf nicht zu einem Absturz führen.

2. Regeln prüfen

Die Einhaltung der verschiedenen Regeln muss nacheinander geprüft werden. Wenn eine Regel verletzt wurde, wird der Zug nicht ausgeführt, und die Eingabe des Start- und des Zielfeldes muss wiederholt werden. Wenn alle Regeln beachtet wurden, geht es weiter bei Schritt 3.

Man kann zwischen impliziten und expliziten Regeln unterscheiden. Zu den impliziten Regeln gehört Folgendes:

- eigene Figur: Auf dem Startfeld muss sich eine Figur befinden, die dem Spieler, der gerade am Zug ist, gehört.
- Zugzwang: Das Start- und das Zielfeld müssen sich unterscheiden, es muss also wirklich ein Zug ausgeführt werden.

Die expliziten Regeln kann man der Aufgabenstellung entnehmen:

- Zielfeld frei (Regel 1): Da sich auf einem Feld nur eine Figur befinden kann, muss geprüft werden, ob das Zielfeld frei ist.
- senkrecht/waagrecht (Regel 2): Da nur in senkrechter oder in waagrechter Richtung gezogen werden darf, müssen Start- und Zielfeld entweder dieselbe x- oder dieselbe y-Koordinate haben.
- Weg frei (Regel 2): Da beim Ziehen keine Figur übersprungen werden darf, muss (bspw. mit Hilfe einer einfachen for-Schleife) überprüft werden, ob der Weg vom Start- zum Zielfeld frei ist.
- Thron (Regel 3): Da der Thron nur vom König besetzt werden darf, muss sich auf dem Startfeld der König befinden oder das Zielfeld darf nicht der Thron sein.

3. Zug ausführen

Der Zug wird ausgeführt, indem die Figur vom Startfeld entfernt und auf das Zielfeld gesetzt wird.

4. ggf. schlagen

Nachdem der Zug ausgeführt wurde, muss gemäß Regel 4 geprüft werden, ob durch den Zug eine gegnerische Figur geschlagen wurde. Diese Überprüfung und ggf. die Entfernung der gegnerischen Figur(en) muss das Programm ohne Interaktion mit dem Benutzer übernehmen. (Der Benutzer soll z.B. nicht selbst die zu schlagenden Figuren anklicken müssen.)

Es werden nacheinander alle vier Richtungen betrachtet und überprüft, ob eine gegnerische Figur (aber nicht der König !) sich zwischen der gerade gezogenen und einer befreundeten Figur befindet. Falls ja, wird die gegnerische Figur entfernt. Zu beachten ist, dass es durchaus vorkommen kann, dass mehrere Figuren durch einen Zug geschlagen werden.

5. Spielende prüfen

Nach jedem Zug muss geprüft werden, ob das Spiel durch diesen Zug gemäß der Regeln 5 (Schwarz gewinnt), 6 (Weiß gewinnt) oder 7 (unentschieden) beendet wurde.

Schwarz gewinnt Wenn der weiße König gefangen genommen wurde, hat Schwarz gewonnen. Wenn Schwarz gerade gezogen hat, muss geprüft werden, ob der König neben der gerade gezogenen Figur steht und – falls ja – ob auch die drei anderen Nachbarfelder des Königs durch schwarze Figuren besetzt sind bzw. ob zwei der anderen Nachbarfelder durch Belagerer besetzt sind und das vierte Nachbarfeld das Thronfeld ist. Wenn Weiß gerade gezogen hat, darf man nicht vergessen, dass der König sich selbst in Gefangenschaft begeben kann, indem er von seinem Thron zu einem Nachbarfeld geht, das von drei schwarzen Figuren umzingelt ist. Dieser Fall sollte – auch wenn er vermutlich in der Praxis fast nie vorkommt – idealerweise auch überprüft werden; da die Aufgabenstellung in diesem Punkt aber undeutlich ist, ist es auch akzeptabel, wenn eine solche passive Gefangennahme nicht geprüft wird.

Weiß gewinnt Durch einen Zug von Schwarz kann Weiß nicht gewinnen. Wenn Weiß den König gezogen hat, muss überprüft werden, ob das Zielfeld ein Randfeld ist, denn dann hat Weiß gewonnen.

Unentschieden Wenn kein Zug mehr möglich ist, endet das Spiel unentschieden. Um zu überprüfen, ob nach dem aktuellen Zug noch ein weiterer Zug möglich ist, kann man einfach die Funktion des Hilfesystems „missbrauchen“, die auf Anfrage die regelgemäßen Möglichkeiten für den nächsten Zug aufzeigt. Wenn diese Funktion – die weiter unten beschrieben wird – keine Möglichkeit findet, ist das Spiel zu Ende. Selbst wenn alle schwarzen Figuren geschlagen wurden und Schwarz deshalb keinen Zug mehr machen kann, hat nicht Weiß gewonnen, sondern das Spiel endet nach Regel 7 unentschieden.

Das zweite Kriterium für ein Unentschieden ist die dreimalige *Wiederholung einer Stellung*. Hierzu noch einmal die Erklärung aus den BWINF-FAQ: „Wenn die gleiche Stellung zum dritten Mal im Spielverlauf vorkommt, endet die Partie unentschieden. Zwei Stellungen sind gleich, wenn Figuren gleichen Typs dieselben Felder besetzen und dieselbe Partei am Zug ist.“

Um dieses Kriterium überprüfen zu können, wird natürlich in irgendeiner Form eine Protokollierung der bisherigen Stellungen benötigt. Am wenigsten Speicherplatz wird verbraucht, wenn man nur die Züge speichert und daraus die erreichten Stellungen rekonstruiert oder durch direkte Analyse der Züge irgendwie feststellt, welche Züge sich gegenseitig aufheben und wann dadurch eine Stellungswiederholung erreicht wird. Diese Vorgehensweise scheint zu aufwendig zu sein, insbesondere da man in der Praxis davon ausgehen kann, dass ein Spiel – obwohl theoretisch möglich – nicht so lange dauert, dass das Speichern aller erreichten Stellungen den Speicherplatz sprengt.

Durch das direkte Speichern der erreichten Stellungen ist der Vergleich zweier Stellungen hingegen recht einfach. Natürlich ist es hierbei nicht sinnvoll, ggf. verwendete Pseudospielfelder (s.o.)

mitzuspeichern.

Alternativ kann man auch nur die Positionen der einzelnen Figuren speichern. Dabei muss man aber darauf achten, dass sich beim Vergleich zweier Stellungen Figuren der gleichen Farbe nicht unterscheiden. (Ausnahme: Der König unterscheidet sich natürlich schon von seinen Getreuen.) Um eine ungewollte Unterscheidung zweier Figuren zu vermeiden, ist es sinnvoll, die Positionen der gleichen Figuren sortiert zu speichern. Man gibt den Feldern Nummern, bspw. von links oben nach rechts unten zeilenweise von 0 bis 48. Die 1. weiße Figur, deren Position man speichert, ist dann der Getreue mit der kleinsten Position usw.. Ein Vergleich zweier Stellungen ist dann einfach möglich, indem man die Positionen aller Figuren vergleicht.

In jedem Fall sollte man noch folgende Punkte beachten: Beim Speichern der erreichten Stellungen (und natürlich erst recht beim Vergleich) muss man zwischen „Weiß am Zug“ und „Schwarz am Zug“ unterscheiden. Zu jeder gespeicherten Stellung muss ein Zähler geführt werden, der angibt, wie oft die Stellung schon erreicht wurde. Sobald eine beliebige Figur geschlagen wurde, können alle gespeicherten Stellungen verworfen werden, da sie niemals mehr erreicht werden können, da sich ja nun eine Figur weniger auf dem Spielbrett befindet. Dies spart Speicherplatz und beschleunigt den Vergleich.

Theoretisch ist es sinnvoll, die Stellungen „sortiert“ zu speichern, da dadurch die Suche nach der aktuellen Stellung (um festzustellen, ob sie schon einmal erreicht wurde) beschleunigt werden kann. Bspw. kann man jeder Stellung eine Zahl zuordnen, die man zum Sortieren verwenden kann: indem man ein leeres Feld durch 0, einen weißen Getreuen durch 1, den König durch 2 und einen schwarzen Belagerer durch 3 repräsentiert, die jeweilige Zahl mit 4^i multipliziert, wobei i die Nummer des jeweiligen Feldes ist, und alle Produkte aufaddiert. Eine ähnliche Ordnung kann man auch dann verwenden, wenn man nicht das ganze Brett, sondern nur die Positionen der Figuren speichert. Ein Geschwindigkeitsvorteil wird allerdings in der Praxis bei einem normalen Spiel (d.h. wenn die Spieler nicht absichtlich versuchen, möglichst viele Züge zu machen) vermutlich nicht auffallen.

6. Zug protokollieren

Zum Schluss müssen die Informationen gespeichert werden, die bei der oben beschriebenen Überprüfung einer Stellungswiederholung benötigt werden.

Hilfesystem

Das Hilfesystem soll zumindest die in der Aufgabenstellung genannten Punkte abdecken:

einführende Informationen zum Spiel Im Wesentlichen eine kurze Beschreibung der Spielidee, des Spielziels und der Spielregeln (vgl. einführende Informationen in der Aufgabenstellung). Darüber hinaus sollte die Bedienung des Programms kurz erläutert werden, insbesondere wenn sie nicht intuitiv ist.

Meldungen bei unerlaubten Zügen Wenn ein Spieler einen Zug ausführen will, der gegen eine Regel verstößt, muss eine entsprechende Meldung angezeigt werden. Wichtig ist hierbei, dass nicht nur die Meldung „Ungültiger Zug!“ erscheint, denn dies wäre nicht besonders hilfreich. Eine vollständige Aufzählung aller Regeln ist zwar schon etwas besser, aber der Spieler weiß dann evtl. immer noch nicht, was er genau falsch gemacht hat. Deshalb sollte wirklich genau die Regel angezeigt werden, die verletzt wurde. (Wenn mehrere Regeln verletzt wurden, könnte man entweder alle verletzten Regeln oder nur eine anzeigen.) Schön wäre, wenn ein Fehler auf dem Spielbrett entsprechend markiert wird, während die passende Meldung angezeigt wird. Bspw. könnte eine Figur, die ein Spieler unerlaubterweise überspringen wollte, rot hervorgehoben werden. Zusammenfassend kann man also festhalten, dass durch die Meldung wirklich erreicht werden soll, dass dem Spieler klar wird, warum der Zug nicht ausgeführt werden konnte.

regelgemäße Möglichkeiten für den nächsten Zug Auf Anfrage sollen dem Spieler alle legalen Züge gezeigt werden. Hierbei kann man sich überlegen, ob man alle möglichen Züge gleichzeitig zeigt (was bei ungeschickter Darstellung unübersichtlich werden kann) oder ob man die möglichen Züge nach Figuren ordnet; bspw. könnte der Spieler eine eigene Figur auswählen und sich die möglichen Züge dieser Figur anzeigen lassen, oder es werden nacheinander (z. B. im 5-Sekunden-Takt) die Möglichkeiten der einzelnen Figuren gezeigt. Die einfachste und allgemeinste Lösung für das Finden aller möglichen Züge einer Figur ist das Aufrufen der Funktion, die die Einhaltung der Regeln überprüft, nacheinander für alle $49 - 1$ Felder als Zielfeld. Eine auf die vorgegebenen Regeln angepasste Vorgehensweise ist die Untersuchung der Felder in alle vier Richtungen, bis eine andere Figur getroffen wird (die ja nicht „vertrieben“ oder übersprungen werden darf) oder der Rand des Brettes erreicht wird. Natürlich darf dabei auch die Thronregel nicht vergessen werden.

Schnittstelle zwischen Spielumgebung und Hilfesystem

Die Aufgabenstellung spricht von zwei Software-Modulen, die zu realisieren sind, nämlich Spielumgebung und Hilfesystem (als Teil der Spielumgebung). Hierbei ist eine softwaretechnisch saubere Lösung für die Schnittstelle zum Hilfesystem zu realisieren, die eine flexible Nutzung der Hilfeinformationen aus der Spielumgebung heraus erlauben muss. Der in Abbildung 1 vorgeschlagene objektorientierte Entwurf ist zwar unvollständig, zeigt aber eine Möglichkeit, wie die Regeln und die zugehörigen Informationen des Hilfesystems sinnvoll strukturiert und flexibel verwaltet werden können.

Die zentrale Klasse ist das Spiel – sie steuert den Spielablauf. Sie hat eine Referenz auf das Spielbrett, das Methoden zum Ziehen und Entfernen von Figuren (Letzteres wird beim Schlagen benötigt) zur Verfügung stellt. Außerdem kann die Belegung jedes einzelnen Spielfeldes abgefragt werden. Um die Regeln vom Spielablauf loszulösen, werden sie in einem RegelContainer verwaltet, auf den das Spiel zugreifen kann.

Obwohl die einzelnen Regeln sehr unterschiedlich sind, kann man eine gemeinsame Schnittstelle definieren, das Interface Regel: Wesentlich ist hierbei eine Methode regelPrüfen, die überprüft, ob der beabsichtigte Zug die Regel verletzt. Darüber hinaus gehören zu einer Regel verschiedene

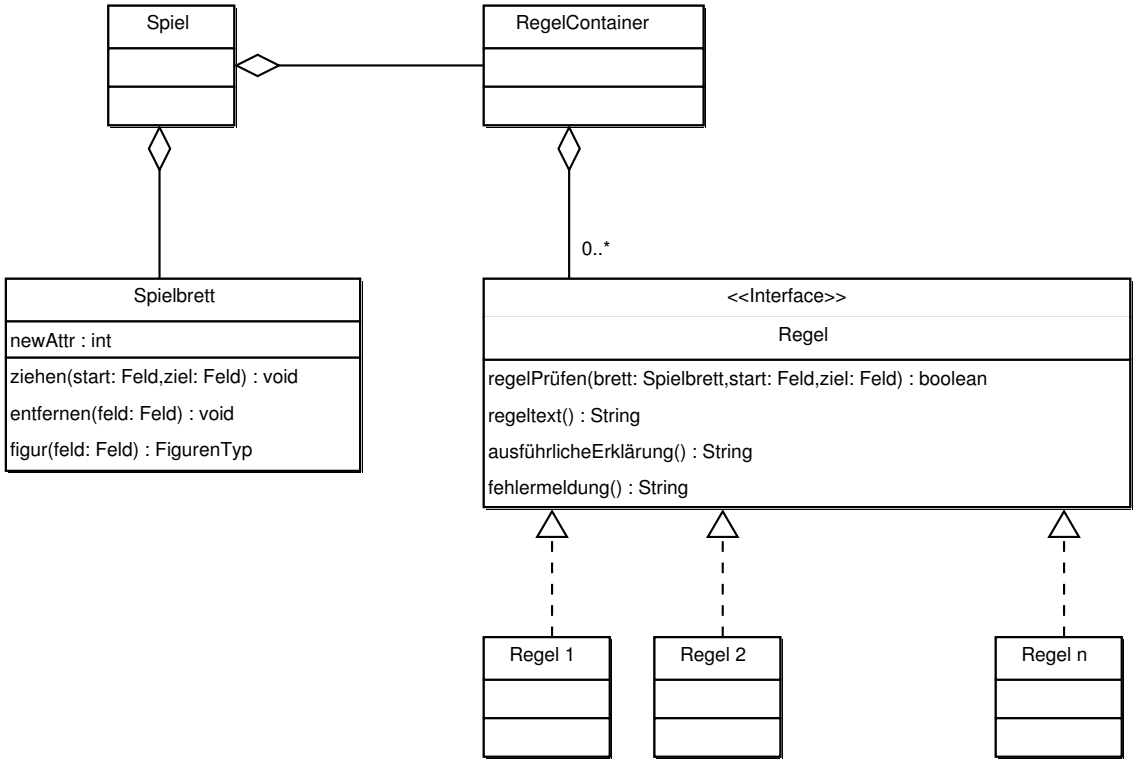


Abbildung 1: Modell für Schnittstelle zwischen Spielumgebung und Hilfeinformationen

Informationen des Hilfesystems. Zum Abrufen dieser Informationen kann man bspw. folgende Methoden verlangen:

- regeltext (die Regel selbst, bspw. die jeweilige Erklärung aus der Aufgabenstellung)
- ausführlicheErklärung (eine ausführliche Erklärung der Regel, evtl. mit Beispiel)
- fehlermeldung (eine aussagekräftige Fehlermeldung, bezogen auf den letzten Aufruf von regelPrüfen, der ein negatives Ergebnis lieferte)

Für jede Regel wird dann eine eigene Klasse erstellt, die das Regel-Interface implementiert.

Vor der Ausführung eines Zuges wird einfach die Methode regelPrüfen aller im RegelContainer enthaltenen Regeln aufgerufen. Ggf. kann danach eine entsprechende Fehlermeldung abgerufen werden. Bei den einführenden Informationen kann der Regeltext aller Regeln ausgegeben werden; bei Bedarf kann auf die ausführlichen Erklärungen der einzelnen Regeln zugegriffen werden.

Dieser Ansatz hat den Vorteil, dass es sehr einfach ist, die Spielregeln zu ändern (Regeln entfernen, neue Regeln hinzufügen, Regeln inhaltlich ändern). Auch die Informationen des Hilfe-

systems können leicht angepasst werden, insbesondere, wenn die Texte (oder auch Bilder) unabhängig vom Quellcode der Methoden organisiert sind, z.B. in einer separaten Datenstruktur oder mit Hilfe von Dateien.

Erweiterungsideen

- Auch wenn ausdrücklich nicht verlangt wurde, dass ein Mensch gegen den Computer spielen kann, spricht natürlich nichts dagegen, dies als Erweiterung zu implementieren. Dabei ist zum einen die Verwendung von Spielbäumen mit α/β-Suche und zum anderen die Implementierung einer festen Strategie möglich. Das Aufstellen einer geeigneten Bewertungsfunktion für die α/β-Suche bzw. die Realisierung einer konkreten Strategie ist allerdings ein komplexes Problem, das an dieser Stelle nicht näher behandelt werden kann.

- Das Protokollieren der Spielzüge, die Anzeige des Protokolls und die Möglichkeit, Züge zurückzunehmen, unterstützten eine Analyse des Spiels. Unter Verwendung solcher Features kann man sich, nachdem man bspw. verloren hat, die Partie noch einmal anschauen und nach den entscheidenden Fehlern suchen.

- Wenn der Spieler einen illegalen Zug durchführen will, kann es hilfreich sein, wenn das Programm nicht nur mitteilt, warum der Zug nicht ausgeführt werden kann, sondern wenn es einen legalen Gegenvorschlag macht, der „fast“ dem Zug entspricht, den der Spieler machen wollte. Bsp.: Der Spieler versucht, eine Figur zu überspringen oder auf ein Feld zu ziehen, wo bereits eine andere Figur sitzt. Das Programm schlägt vor, die eigene Figur nur bis zu dem Feld direkt vor der störenden Figur zu ziehen (natürlich nur, wenn dies erlaubt ist).

- Eine Vorschau der Auswirkungen eines Zuges, bevor dieser endgültig ausgeführt wird, kann sinnvoll sein. Wenn man bspw. eine Figur mit Drag and Drop zu einem Zielfeld zieht, könnte das Programm, noch bevor man loslässt, anzeigen, welche Figuren geschlagen werden, wenn man sich tatsächlich für das jeweilige Zielfeld entscheidet.

- Insbesondere wenn man einen Computer-Spieler realisiert hat, könnte folgende Unterstützung für den menschlichen Spieler hilfreich sein: Auf Anfrage werden die eigenen Figuren markiert, die bedroht sind, die also im nächsten Zug vom Gegner geschlagen werden könnten. Auf Anfrage werden Schlagmöglichkeiten gekennzeichnet, also legale Züge, die der Spieler sofort ausführen kann, um eine gegnerische Figur zu schlagen.

Aufgabe 3: Ampelsteuerung

Vorab: Die beiden Interpreter unterscheiden sich in ihrer Funktion teilweise. Der wichtigste Unterschied: Der Python-Interpreters erwartet für jede Zustand/Ereignis-Kombination eine (möglicherweise leere) Übergangsdeklaration, sonst meldet er einen Fehler. Bei Verwendung dieses Interpreters können die Quelltexte ziemlich umfangreich werden.

Lösungsidee

In dieser Aufgabe wird zur Realisierung unterschiedlich komplexer Betriebsmodi einer einfachen Ampelanlage eine formale Sprache S vorgegeben. Diese Sprache hat zwei wichtige Eigenschaften: Zum einen handelt es sich um eine im wesentlichen deklarative Sprache; die Abarbeitung von Sprachkonstrukten durch einen S-Interpreter führt häufig nur zu Änderungen der „inneren Welt“ des Interpreters (zu neuen Zuständen, Ereignissen oder Zustandsübergängen), aber nicht zu Veränderungen der Außenwelt, also der Ampelanlage. Insbesondere kann der Effekt der Abarbeitung des Ereignisteils einer Übergangsdeklaration gründlich missverstanden werden. Zum zweiten ist das in S beschriebene Konstrukt ein Automat, der eben aus den in S beschriebenen Zuständen und Übergängen besteht und Wörter des durch die Ereignisse definierten Alphabets abarbeitet. Besonderheiten gibt es ebenfalls zwei: Zum einen gibt es keine besonderen Endzustände, so dass die Ampelanlage prinzipiell unendlich lange arbeiten kann, ist sie erst einmal in Gang gesetzt worden. Zum anderen arbeitet der Automat in einem Zeittakt, der durch eine im Interpreter eingebaute und wie ein Kurzzeitwecker arbeitende Uhr realisiert wird; das Ablaufen der Uhr ist als besonderes Ereignis vordefiniert.

Die entscheidende Hürde bei der Bearbeitung dieser Aufgabe war, die genannten Eigenschaften zu verstehen. Wer hieran gescheitert ist, konnte für die Bearbeitung dieser Aufgabe letztlich nur noch Fleißpunkte erhalten.

Wenn ein S-Programm einen Automaten spezifiziert, ist ein enstprechendes Zustandsdiagramm eine gute Möglichkeit, in S realisierte Lösungen zu beschreiben. Da die Aufgabenstellung beinahe ein Zustandsdiagramm für die Lösung von Teilaufgabe 1 enthält, kann man sich überlegen, welche Schritte von einem Zustandsdiagramm zu einem Programm in S führen:

1. **Übernahme der Zustände aus dem Zustandsdiagramm**
   Für jeden Zustand wird mittels
   `(#ZUSTANDSDEKLARATION <name>)`
   ein Zustand festgelegt.

2. **Ereignisse**
   Für jedes Ereignis, das im Zustandsdiagramm erscheint, wird über
   `(#EREIGNISDEKLARATION <name>)`
   das entsprechende Ereignis in S deklariert.

3. **Übergänge**
   Die Übergänge des Zustandsdiagramms werden mit Hilfe der Funktion

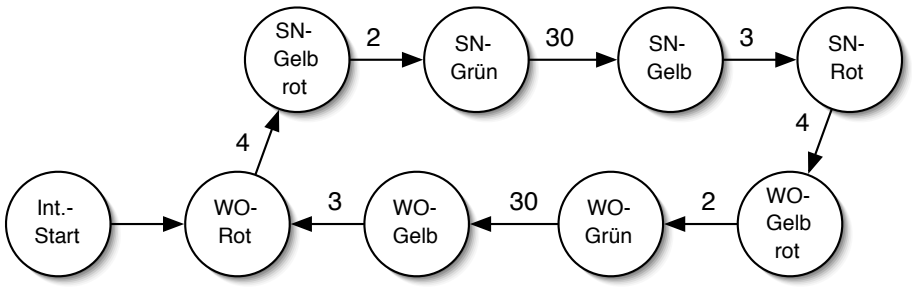
```
(#ÜBERGANGSDEKLARATION <...>)
```
in S umgesetzt. Im einfachen Fall, ohne die Sensoren und die besondere Ereignisse aus den Teilaufgaben 2 und 3, sieht eine typische Übergangsdeklaration dabei wie folgt aus:

```
(#ÜBERGANSDEKLARATION WO-ROT COUNTDOWN-IST-NULL
  (#EIN #SN #A #GELB)
  (#COUNTDOWN 2)
  SN-GELBROT)
```

Allgemeiner: Die Ampelsteuerung reagiert (in jedem Zustand) lediglich immer auf das Timer-Ereignis COUNTDOWN-IST-NULL, schaltet dann zunächst die Ampellichter, setzt den Timer auf den neuen Wert (siehe Übergangspfeile im vorgegebenen Diagramm) und wechselt anschließend in den neuen Zustand.

Teilaufgabe 1

Das angegebene Zustandsdiagramm ist nahezu vollständig. Lediglich der Übergang vom Zustand INTERPRETER-START in den „Normalzyklus“ der Ampelanlage muss noch hinzugefügt werden. Im Zustand INTERPRETER-START befindet sich die Ampel in einem nicht näher definierten Zustand. Insbesondere ist auch der Schaltzustand der einzelnen Lichtzeichen nicht bekannt. Je nach Konstruktion der Ampel ist es zum Beispiel denkbar, dass die Lampen noch so leuchten wie vor einem Stromausfall. Daher muss die Ampel zunächst in den sicheren Zustand gebracht werden: Alle Ampeln zeigen rot. Dies kann nur über zehn Steuerbefehle erfolgen. (4 x Grün aus, 2 x Gelb aus, 4 x Rot an). Vernachlässigung einiger dieser Schaltvorgänge, wie beispielsweise nur das Setzen der roten Lichter ist nicht zulässig. Den sicheren Zustand gibt es übrigens schon, und zwar gleich zweimal: WO-Rot und SN-Rot. Im folgenden Beispiel ist WO-Rot gewählt:



Bei allen weiteren Zustandsübergängen kann von der Lichtsituation des Vorgängerzustandes ausgegangen werden, so dass die Umschaltlisten entsprechend kürzer ausfallen können. Gezieltes Ausschalten der Ampel war in dieser Aufgabe nicht gefordert.

Ein letzter Knackpunkt: Ein S-Programm, das eine Umsetzung des obigen Diagramms darstellt, bewirkt noch nichts. Um die Ampelsteuerung zu starten, bedarf es eines Ereignisses. Am saubersten ist wohl die Verwendung eines speziellen Ereignisses sowohl in der Deklaration des Übergangs von INTERPRETER-START in den gewählten sicheren Zustand als auch in einem expli-

ziten EREIGNIS-Befehl z.B. am Ende des Programms. Alternativ kann (auch am Ende des Programms) die Uhr gesetzt werden, z.B. durch das Kommando `(#COUNTDOWN 1)`. Dann kann in der Deklaration des Übergangs von INTERPRETER-START auf das Ereignis COUNTDOWN-IST-NULL zurückgegriffen werden. *Achtung:* Der Python-Interpreter setzt ein solches Ereignis beim Start automatisch ab, so dass bei dessen Verwendung kein abschließendes `#COUNTDOWN`-Kommando benötigt wird.

Teilaufgabe 2

Nun sollen zusätzliche Ereignisse benutzt werden, um eine verkehrsabhängige Ampelsteuerung zu realisieren. Kommt man auf die Idee, dies nur mit Hilfe von Ereignissen zu tun, die den Sensorsignalen von Fußgängerknöpfen und Induktionsschleifen entsprechen (im weiteren „Sensorereignisse“ genannt), stößt man auf folgende Sachverhalte:

a) Ereignisse werden nicht taktübergreifend gespeichert, wenn sie in einem Takt keine Reaktion ausgelöst haben. Dies geschieht weder im Zustandsautomat, noch in den Sensoren. Ein Auto auf einem Induktionssensor löst also nur einmal ein Ereignis aus! (FAQ und Beschreibung der Interpreter)

b) Gibt es mehrere mögliche Übergänge, so wird einer davon zufällig ausgewählt. (Beschreibung der Interpreter) Damit und mit Voraussetzung (a) kann es nur einen Zustandsübergang pro Takt geben.

c) Es gibt keine aussagenlogische Verknüpfungen wie sensor1$\wedge\neg$sensor2 . (Sprachdefinition)

Allein durch (a) und (b) wäre Teilaufgabe 2 nur mit Hilfe von Sensorereignissen nicht lösbar. Ein einfaches Beispiel: Die Ampel ist gerade Rot und die Steuerung wartet darauf, dass entweder die Induktionsschleife in SN-Richtung oder die Schleife in WO-Richtung ausgelöst wird. Werden nun beide gleichzeitig bzw. innerhalb eines Taktes ausgelöst, so springt die Steuerung zufällig (b) in einen Zustand, um WO auf Gelb und anschließend auf Grün zu schalten, oder um SN entsprechend zu schalten. Da der Zustandsübergang aber schon einen Takt bedeutet, ist die Information über das zweite wartende Auto nach dem Zustandsübergang verloren (a). Dieses Fahrzeug wird nun bis in alle Ewigkeit warten.

Deshalb ist wichtig, durch Erweiterung des Normalzyklus (in der Aufgabenstellung gefordert) Sensor- und Zeitsteuerung miteinander zu kombinieren. Hierbei können zwei Dinge berücksichtigt werden: (1) Ereignisse, die durch Induktionsschleife und Fußgängerknopf aus derselben Richtung verursacht werden, müssen nicht unterschieden werden, da beide dazu führen sollen, dass die Wartezeit dieser Richtung reduziert wird. (2) Der wichtigste Effekt eines solchen Signals ist die Verkürzung der Countdown-Zeit nach dem entsprechenden „Grün“-Zustand. Hinzu kommt die Anforderung, dass der Ampelzyklus unverändert weiterläuft, wenn kein Sensorereignis eintritt. Eine entsprechende Lösung zeigt Abbildung 2.

Hierbei wird für jeden Zustand des Normalzyklus ein paralleler Zustand erzeugt. Für jede Richtung wird ein Ereignis eingeführt (KSN und KWO), das der Betätigung eines Sensors dieser

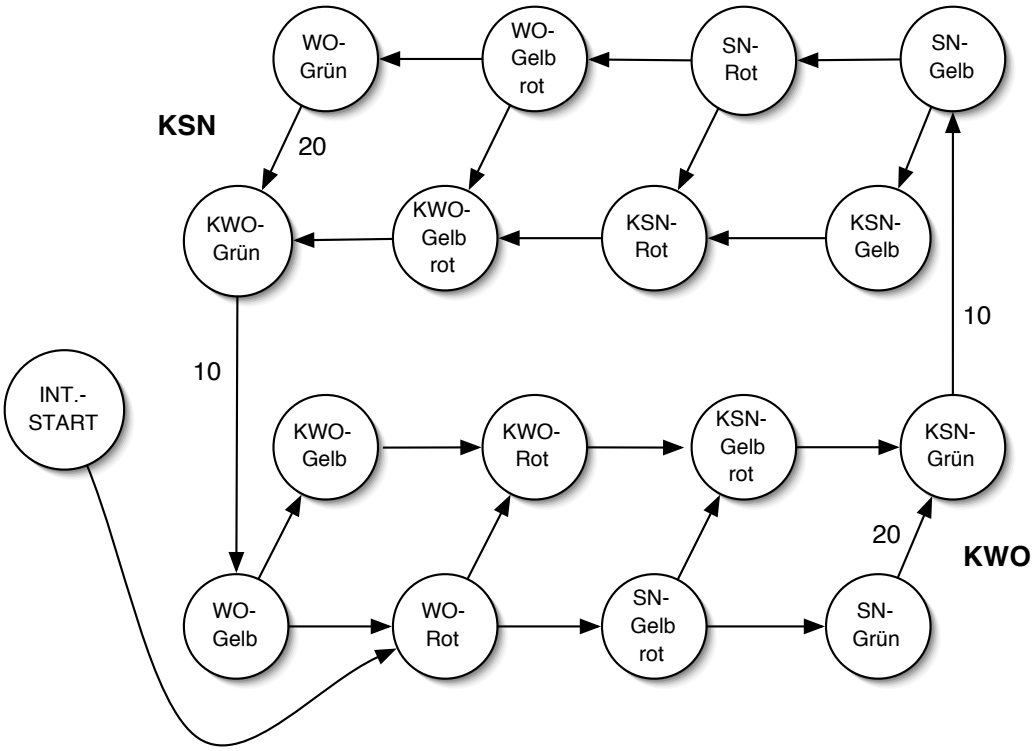


Abbildung 2: Mögliche Lösung zu Teilaufgabe 2

Richtung entspricht. Ein solches Ereignis leitet vom Normalzustand in den Parallelzustand über, und zwar für die „Wartezustände“ der entsprechenden Richtung. Beispiel: Autos und Fußgänger in Richtung SN müssen in den Zuständen SN-Gelb, SN-Rot, WO-Gelbrot und WO-Grün warten; das Ereignis KSN führt folglich zu Übergängen in die Zustände KSN-Gelb, KSN-Rot, KWO-Gelbrot und KWO-Grün. Eine Besonderheit gibt es bei den Grün-Zuständen: Die Übergangszeit der Normalzustände wird auf 20 reduziert, bei den K-Zuständen beträgt sie 10. Von den normalen Grün-Zuständen wird auf jeden Fall in die entsprechenden K-Zustände übergegangen, so dass ohne Sensorereignis die Grünphase weiterhin 30 lang ist, während sie nach einem Sensorereignis der kreuzenden Richtung auf 10 verkürzt wird. Nach einem K-Grünzustand wird zunächst wieder in einen normalen Gelb-Zustand gewechselt, ab dem dann wieder auf Sensorereignisse reagiert werden kann. Zu beobachten ist, dass Sensorereignisse der nicht wartenden Richtung verfallen, was aber sachlich richtig ist.

Es gibt sicher eine ganze Reihe von Möglichkeiten, diese Aufgabe zu lösen. Wichtig ist, dass die Anzahl der Zustände nicht zu groß und damit das Programm und auch eine grafische Darstellung des realisierten Automaten nicht zu unübersichtlich wird. Auch die Anzahl der Ereignisse sollte nicht unnötig groß sein, denn speziell bei Verwendung des Python-Interpreters wird der Quelltext dann schnell sehr umfangreich.

Teilaufgabe 3

Die Realisierung der dritten Aufgabe kann analog zu der unter Teilaufgabe 2 beschriebenen Idee erfolgen. Für jeden der weiteren Betriebsmodi wie Marathon, Unfall etc. wird eine Kopie des oben aufgeführten Zustandsautomaten erstellt – ähnlich wie die Parallelzustände oben für das Sensor-Ereignis. Per von außen ausgelöstem Ereignis kann nun zwischen den Kopien des Zustandsautomaten gewechselt werden. Im Gegensatz zu Aufgabe 2 ist hier der Wechsel hin und her möglich. Jeder Zustand ist durch eine Übergangsdeklaration mit den Zuständen in den kopierten Automaten verbunden.

In diesem Aufgabenteil kann davon ausgegangen werden, dass die Ereignisse sequentiell aufgerufen werden und nicht unnötig miteinander kombiniert werden. Ein Verkehrsunfall bei einer Sperrung wegen einer Marathonveranstaltung ist wohl recht selten. In diesem Fall ist es dem bedienenden Beamten wohl durchaus zumutbar, zu entscheiden, welchen der Betriebsmodi er haben möchte.

Unter dieser Voraussetzung ist auch eine vergleichsweise einfache Lösung dieses Aufgabenteils möglich: Es wird ein einziger weiterer Zustand eingeführt, der etwa AUSNAHMESITUATION heißen kann. Für die Betriebsmodi gibt es je zwei Ereignisse, und zwar zum An- und Abschalten. Ein An-Ereignis löst die gewünschten Ampelschaltungen aus und führt in die Ausnahmesituation (bei Verwendung des Python-Interpreters: dies gilt auch, wenn der Automat sich schon in der Ausnahmesituation befindet). In der Ausnahmesituation führt ein Aus-Ereignis dann in den Zustand INTERPRETER-START und setzt das nötige Startereignis bzw. einen kurzen Countdown. Durch den schon für Teil 1 realisierten Übergang von INTERPRETER-START in einen gesicherten Zustand wird damit auch die Ausnahmesituation gesichert verlassen. Bei Verwendung des Python-Interpreters muss in der Ausnahmesituation auch das Ereignis COUNTDOWN-IST-

NULL behandelt werden, am einfachsten, indem der Timer wieder auf einen relativ geringen Wert gesetzt wird.

Wie bei Teilaufgabe 2 gibt es auch hier verschiedene Möglichkeiten; je eleganter, übersichtlicher und nachvollziehbarer, desto besser.

Programmablaufprotokolle und Programmdokumentation

Programmablaufprotokolle im engeren Sinne sind hier nicht möglich. Die Ausführung von S-Programmen kann durch einige Screenshots der Interpreter illustriert werden. Wichtig ist, dass die Auswirkungen der entscheidenden Ideen hinter den Programmen zumindest an theoretischen Beispielen dargestellt und erläutert werden.

Leider wurde versäumt, in der Sprache S Kommentare zu erlauben. Und das, wo beim BWINF doch immer so viel Wert auf ausreichendes Kommentieren des Quellcodes gelegt wird! Kluge Teilnehmer haben eigene Vorverarbeitungen entwickelt, die Kommentare aus S-Quellen entfernen, bevor sie vom Interpreter verarbeitet werden. Ohne die Verwendung solcher Mechanismen sind ausführliche Erläuterungen der S-Quellen im Textteil besonders wichtig.

Programmtext

Hier soll als Beispiel nur der Programmtext für die Steuerung aus Teilaufgabe 1 angeführt werden.

```
(#ZUSTANDSDEKLARATION SN-Gelbrot)
(#ZUSTANDSDEKLARATION SN-Grün)
(#ZUSTANDSDEKLARATION SN-Gelb)
(#ZUSTANDSDEKLARATION SN-Rot)
(#ZUSTANDSDEKLARATION WO-Gelbrot)
(#ZUSTANDSDEKLARATION WO-Grün)
(#ZUSTANDSDEKLARATION WO-Gelb)
(#ZUSTANDSDEKLARATION WO-Rot)

(#EREIGNISDEKLARATION AMPEL-STARTEN)

(#ÜBERGANGSDEKLARATION INTERPRETER-START AMPEL-STARTEN
  (#AUS #SN #F #GRÜN) (#AUS #SN #A #GRÜN) (#AUS #SN #A #GELB)
  (#EIN #SN #F #ROT) (#EIN #SN #A #ROT)
  (#AUS #WO #F #GRÜN) (#AUS #WO #A #GRÜN) (#AUS #WO #A #GELB)
  (#EIN #WO #F #ROT) (#EIN #WO #A #ROT)
  (#COUNTDOWN 1)
  WO-Rot)

(#ÜBERGANGSDEKLARATION WO-Rot COUNTDOWN-IST-NULL
  (#EIN #SN #A #GELB)
```

```
  (#COUNTDOWN 2)
  SN-Gelbrot)

(#ÜBERGANGSDEKLARATION SN-Gelbrot COUNTDOWN-IST-NULL
  (#AUS #SN #A #ROT) (#AUS #SN #F #ROT) (#AUS #SN #A #GELB)
  (#EIN #SN #A #GRÜN) (#EIN #SN #F #GRÜN)
  (#COUNTDOWN 30)
  SN-Grün)

(#ÜBERGANGSDEKLARATION SN-Grün COUNTDOWN-IST-NULL
  (#AUS #SN #A #GRÜN) (#AUS #SN #F #GRÜN)
  (#EIN #SN #A #GELB) (#EIN #SN #F #ROT)
  (#COUNTDOWN 3)
  SN-Gelb)

(#ÜBERGANGSDEKLARATION SN-Gelb COUNTDOWN-IST-NULL
  (#AUS #SN #A #GELB)
  (#EIN #SN #A #ROT)
  (#COUNTDOWN 4)
  SN-Rot)

(#ÜBERGANGSDEKLARATION SN-Rot COUNTDOWN-IST-NULL
  (#EIN #WO #A #GELB)
  (#COUNTDOWN 2)
  WO-Gelbrot)

(#ÜBERGANGSDEKLARATION WO-Gelbrot COUNTDOWN-IST-NULL
  (#AUS #WO #A #ROT) (#AUS #WO #F #ROT) (#AUS #WO #A #GELB)
  (#EIN #WO #A #GRÜN) (#EIN #WO #F #GRÜN)
  (#COUNTDOWN 30)
  WO-Grün)

(#ÜBERGANGSDEKLARATION WO-Grün COUNTDOWN-IST-NULL
  (#AUS #WO #A #GRÜN) (#AUS #WO #F #GRÜN)
  (#EIN #WO #A #GELB) (#EIN #WO #F #ROT)
  (#COUNTDOWN 3)
  WO-Gelb)

(#ÜBERGANGSDEKLARATION WO-Gelb COUNTDOWN-IST-NULL
  (#AUS #WO #A #GELB)
  (#EIN #WO #A #ROT)
  (#COUNTDOWN 4)
  WO-Rot)

(#EREIGNIS AMPEL-STARTEN)
```

Perlen der Informatik – aus den Einsendungen

Allgemeines

Dieses Programm ist eine große Summe von verschiedenen Algorithmen.

Auf Fehler reagieren können Programme ... ganz gut. Zum Beispiel mit Festfahren, Unsinn auf den Bildschirm schreiben, Geräusche verursachen, den Computer abschalten und vieles mehr.

Hiermit wird ... dem berüchtigtsten aller informatischen Ansätze, dem Brute Force gehuldigt.

Hierfür kann ein recht simpler Logarithmus angewandt werden.

Aufgabe 1

Dem echten Automaten wird es sicherlich nicht ausreichen, nur per Tastatur eingegeben zu bekommen, was für Geld ihm angeblich eingeworfen wurde. Ihm ist echtes Geld viel lieber.

Grundsätzlich ist natürlich das Drucken des benötigten Geldscheines immer die Ideallösung, und da wir in Fantasien leben, ist dies natürlich auch sehr simpel zu lösen.

Auch hier könnte man als Eingabevereinbarung sagen, dass nichts die Berechnung stören darf, z.B. ein Kunde.

... die Warnanzeige (vielleicht ein brüllender Ork) ...

Andererseits kann ich mich an meine kleine Schwester erinnern, die Kleingeld früher liebte, denn sie hatte das subjektive Gefühl, dann „mehr“ zu besitzen.

Aber der Kunde hat bei Primzahlen immer noch Pech gehabt und auch in vielen anderen Fällen viel Kleingeld.

```
if (sprache == "Deutsch") print("Willkommen in Fantasien!");
if (sprache == "Fantasisch") print("fzju vgz uhgijk!");
```

Aufgabe 2

... um eine Verwirrung und eine dadurch möglicherweise hervorgerufene geistige Umnachtung des Benutzers zu verhindern.

Im Programm darf sich lediglich die x-Koordinate oder die y-Koordinate ändern; wenn beide sich ändern, handelt es sich um eine Diagonale.

Da ich vermute, dass andere dieses Spiel bis zur Erschöpfung mit Features aufgebläht haben, welche die Welt nicht braucht, ...

Aufgabe 3

Marathonverlaufsherkunfts-Windrichtung

... überall liegen Auto- und/oder Leichenteile herum, Explosionsgefahr.

So soll verhindert werden, dass unschuldige Fußgänger bei Grün über die Straße gehen, aber dann von fiesen, gemeinen Gelbfahrern überrollt werden.

Der Herr der Simulationen. Teil 2: Die zwei Straßen.

Da auf der Love Parade die Ampeln sowieso nicht gebraucht werden, könnten sie auch durcheinander blinken und zur Belustigung rot werden, wenn man drückt.

Ein lose im Quelltext herumgammelndes `(#EREIGNIS COUNTDOWN-IST-NULL)` hat also bei meinem Interpreter keinen Einfluss auf den Programmlauf.