Implementierungsvarianten von endlichen Zustandsautomaten mit Java

Matthias Haselmaier

Zusammenfassung
In dieser Arbeit werden Zustandsautomaten vorgestellt, welche in verschiedensten Anwendungsgebieten eingesetzt werden. Zuerst werden Automaten allgemein vorgestellt und besprochen. Anschließend wird auf drei verschiedene Implementierungsvarianten, prozedural, objektorientiert und mit einem Framework-Ansatz, eingegangen und die Vor- und Nachteile besprochen. Abschließend wird ein Anwendungsbeispiel gegeben, bei dem eine Digitaluhr mithilfe eines Zustandsautomaten gesteuert wird. Dieser wurde mit dem Framework-Ansatz realisiert und in einer XML-Struktur modelliert.

Keywords
Endlicher Zustandsautomat, Design, Java, Finite State Machines

Hochschule Kaiserslautern
Corresponding author: maha0039@stud.hs-kl.de

Inhaltsverzeichnis

Einleitung

Diese Arbeit befasst sich mit drei möglichen Implementierungsvarianten von endlichen Zustandsautomaten. Ein solcher Automat besteht im wesentlichen aus einer endlichen Zahl an Zuständen, Übergängen zwischen Zuständen und Aktionen. Durch den aktuellen Zustand in dem er sich befindet, können gewissen Information über die vorherigen Zuständen geschlossen werden. Ein Zustandsübergang verbindet immer einen vorherigen mit einem Folgezustand. Hierbei kann es sich auch um den selben Zustand handeln. Sie haben immer eine Bedingung die mit ihnen verknüpft ist, welche erfüllt sein muss, um diesen Übergang durchzuführen. Meist handelt es sich hierbei um die jeweilige Eingabe, welche einen bestimmten Übergang auslöst. Die Aktionen eines endlichen Zustandsautomaten sind seine Ausgaben. Sie werden zum Beispiel beim Ein- oder Austreten eines Zustandes ausgelöst oder als Übergangsaktion während dem Übergang von einem zum anderen Zustand. Die Automaten werden wie in Abbildung 1 dargestellt. Jeder Zustand wird mit einem Kreis

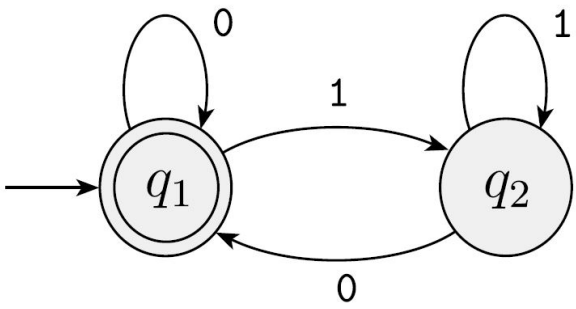

Abbildung 1. Ein Zustandsautomat mit zwei Zuständen.

repräsentiert. Von ihm gehen Pfeile zu den jeweiligen Folgezuständen, welche die Übergänge darstellen. Die Bedingungen der Übergänge werde direkt an die Pfeile geschrieben. Oft haben endlichen Zustandsautomaten auch definierte Endzustände. Werden diese nicht erreicht, lag eine invalide Eingabefolge vor. Diese Zustände werden mit einem zweiten Kreis umrahmt.

Der Automat in Abbildung1 zeigt also einen sehr einfachen, welcher aus zwei Zuständen besteht, `q1` und `q2`. Beide haben jeweils zwei Übergänge, einen auf sich und einen auf den anderen. Hierbei wird bei der Eingabe des Wertes `0` immer in den Zustand `q1` und bei Eingabe von `1` immer zu Zustand `q2` gewechselt. Nur wenn sich der Zustandsautomat am Ende der Eingabefolge im Zustand `q1` befindet, war die Folge valide.

Zustandsautomaten finden zum Beispiel Anwendung in

der Steuerung von Kartenautomaten an Bahnhöfen und steuern dort den Ablauf der einzelnen Schritte, oder auch in der Entwicklung von der KI von computergesteuerten Gegner in Videospielen. Neben den verschiedenen Anwendungszwecken gibt es auch unterschiedliche Implementierungsvarianten. Im nächsten Abschnitt wird auf drei verschiedene Arten der Umsetzung von Zustandsautomaten eingegangen, welche Anhand des Automaten in Abbildung 1 beispielhaft implementiert werden. Anschließend wird mit Hilfe des Framework-Ansatzes ein kleiner Automat erzeugt, welcher eine Digitaluhr steuert.

1. Implementierungsvarianten

Im folgenden wird auf drei verschiedene Implementierungsvarianten von Zustandsautomaten eingegangen. Zur Programmierung wurde Java verwendet. Hierbei werden die Unterschiede im Bezug auf Programmieraufwand, Bedienbarkeit und Anpassbarkeit sehr deutlich.

1.1 Prozedurale Implementierung

Bei der prozeduralen Implementierung wird der endliche Automat mithilfe von verschachtelten `Switch-Case` Anweisungen realisiert. Der aktuelle Zustand wird dabei in der `state` Variable gespeichert. Hierfür kann im einfachsten Fall ein `String` oder ein `Enum` Typen definieren werden.

```
switch(state) {
case A:
    switch(input) {
    case x:
        break;
    case y:
        break;
    }
    break;
case B:
    switch(input) {
    case x:
        break;
    case y:
        break;
    }
    break;
}
```

Listing 1. Modellierung des Automaten

Für jeden der möglichen Zustände besitzt der Automat auf der ersten Ebene einen `Case` Block. Im inneren dieser `Case` Blöcke behandelt die zweite `Switch-Case` Anweisung die Eingaben in den Automaten. Um den Zustand des Zustandsautomaten zu wechseln, wird der Wert der `state` Variable geändert. Die Eintrittsaktionen werden in dieser Implementierung direkt als erste in den `Case` Blöcken der Zustände ausgeführt. Am Ende der inneren `Case` Blöcke wird demnach die Austrittsaktion ausgeführt. Direkt danach würde die Übergangsaktionen formulieren werden.

Um den Automaten in Betrieb zu nehmen, iteriert man in einer Schleife über die `Switch-Case` Blöcke, solang weitere Eingabe vorhanden sind. Um zu prüfen, ob der Zustand nach dem Iterieren auch ein valider Endzustand ist, wird die `state` Variable betrachtet.

1.2 Objektorientierte Implementierung

In der objektorientierten Variante wird jeder Zustand durch ein Objekt des Typen `State`, wie in Listing 2 gezeigt, repräsentiert.

```
public class State {
    private String name;
    private Action entry;
    private Action exit;
    private Map<Object, Transition>
        transitions;

    public State(String aName) {
        this(aName, null, null);
    }

    public State(String aName,
         Action entry, Action exit) {
        this.name = aName;
        this.entry = entry;
        this.exit = exit;
        this.transitions = new HashMap<
            Object, Transition>()
    }

    public void addTransition(Event<?>
        anEvent, Transition aTransition) {
        this.transitions.put(anEvent.get(),
            aTransition);
    }

    public void entry() {
        if(null != this.entry) this.entry.
            action();
    }

    public void exit() {
        if(null != this.exit) this.exit.
            action();
    }

    public State process(Event<?> anEvent) {
        Transition transition = this.
            transitions.get(anEvent.get());
        if(null == transition) return null;
        transition.action();

        return transition.getTargetState();
    }

    // ...
}
```

Listing 2. Das State Objekt

Jeder Zustand besitzt einen Namen und eine `HashMap`, welche die Eingaben auf die Übergänge abbildet. Zusätzlich kann ein Zustand Eintritts- und Austrittsaktionen haben. Um jede Art von Eingaben zu ermöglichen, wurde ein generischer `Event<T>` Typ erstellt, welcher den jeweiligen Eingabetypen kapselt. Um flexible Aktionen zu ermöglichen, wurde hierfür ein funktionales `Action` Interface bereitgestellt. Für die Übergänge wurde ein `Transition` Typ implementiert, welcher einen Zielzustand besitzt. Er kann des Weiteren auch eine Übergangsaktion besitzen. Um den aktuellen Zustand des Automaten zu speichern, wurde ein `Context` Typ zur Verfügung gestellt. Er besitzt eine Referenz auf den aktiven Zustand und eine Liste mit allen validen Endzuständen. Über seine `process(Event<?>)` Methode wird die Eingabe am Automaten an den jeweiligen Zustand weitergeleitet.

Um den endlichen Zustandsautomaten zu modellieren, werden zuerst die benötigten Zustände und anschließend die Übergänge erstellt. Nach dem Instanziier der Eingabeevents, werden nun die jeweiligen Übergänge mit den auslösenden Events den Zuständen zu gewiesen. Abschließend wird der Kontext erstellt. Er bekommt den Startzustand und die möglichen Endzustände übergeben. Nun kann über die Eingabefolge iteriert und die Events der `process(Event<?>)` Methode des Kontextes übergeben werden. Wurde über sämtliche Eingaben iteriert, kann mit der `isAtFinalState()` Methode des Kontextes die Validität der Eingabefolge geprüft werden.

1.3 Framework-Implementierung

Die Variante der Framework-Implementierung bedient sich bei den Ergebnissen der objektorientierten Implementierung. Der endliche Zustandsautomat wird allerdings nicht mehr programmatisch aufgebaut, sondern mithilfe einer XML-Struktur modelliert. Listing 4 zeigt das gegeben XML-Schema, welches die Struktur zum Aufbau des Automaten vorgibt. Das Stammelement der XML-Datei ist hierbei das `fsm` Element, welches von Typ `FiniteStateMachine` ist. Es muss immer genau ein `startState` Element des Typen `State` besitzen und kann beliebig viele `endState` Elemente besitzen, welche ebenfalls vom Typ `State` sind. Des Weiteren besitzt das `fsm` Element mindestens ein `states` und ein `transitions` Element der Typen `States` und `Transitions`. Der `States` Typ besteht aus einer Sequenz von `State` Typen. Ebenso umschließt der `Transitions` Typ eine Sequenz aus mindestens einem `Transition` Typ. Der `State` Typ muss ein `name` Attribut besitzen und kann durch die Attribute `entryaction` und `exitaction` ergänzt werden. Um die Übergänge zu realisieren, muss der `Transition` Typ die Attribute `event`, `source` und `target` besitzen, welche die auslösende Eingabe, den Start- und den Zielzustand des jeweiligen Übergangs festlegen. Außerdem kann er ein Attribut `action` haben, welches die Übergangsaktion bestimmt. Wird der Automaten in Abbildung 1 mit dieser XML-Struktur modelliert, ergibt sich folgendes:

```
<fsm xmlns:xsi="http://www.w3.org/2001/
    XMLSchema-instance">
    <states>
        <state name="q1" />
        <state name="q2" />
    </states>

    <transitions>
        <transition source="q1" target="q1"
            event="0" />
        <transition source="q1" target="q2"
            event="1" />
        <transition source="q2" target="q1"
            event="0" />
        <transition source="q2" target="q2"
            event="1" />
    </transitions>

    <startState name="q1" />
    <endState name="q1" />
</fsm>
```

Listing 3. XML-Modell

Aufgrund der festgelegten Struktur ist es ein leichtes, einen XML-Parser zu schreiben, welcher den endlichen Automaten automatisch generiert und den fertigen Kontext zurück gibt. In Java kann man für die Aktionsattribute der XML-Elemente den vollständigen Klassennahmen angeben. Hiermit kann ein Objekt dieser Klassen dynamisch erzeugt und den Zuständen, bzw. den Übergängen, übergeben werden.

Wurde der Kontext generiert, wird wie bei der objektorientierten Variante über die Eingabefolge iteriert und die Validität der Folge überprüft.

1.4 Evaluierung der Implementierungsansätze

Zur Evaluierung der drei vorgestellten Implementierungsvarianten, wurde neben dem Automaten in Abbildung 1 auch folgender Automat realisiert.
Bei diesen beiden recht kleinen und simplen Automaten war die Umsetzung in der prozeduralen Variante sehr einfach und schnell erledigt. Allerdings hat sich der Code des Automaten in Abbildung 2 bereits über einige Zeile gezogen. Hier kommt der großer Nachteil dieser Variante zum Vorschein. Durch die NxM `Case` Blöcke, mit N gleich der Anzahl an Zuständen und M gleich der Anzahl an Eingabemöglichkeiten, bläht sich der Code sehr schnell auf und wird hierdurch unübersichtlich. Kommt ein neuer Eingabetyp hinzu, wird es auch recht aufwendig diesen zu integrieren, da man innerhalb jedes Zustandes an verschiedenen Stellen einen weiteren `Case` Block hinzufügen muss. Was diese Variante jedoch sehr interessant macht, ist die schnelle Laufzeit. Im Gegensatz zu der objektorientierten Implementierung gibt es hier keine Indirektionen durch Objektreferenzen. Die `Switch-Case` Anweisungen werden äußerst schnell durchlaufen, was diese Variante trotz der aufwendigen Wartungsarbeiten besonders für Performanz

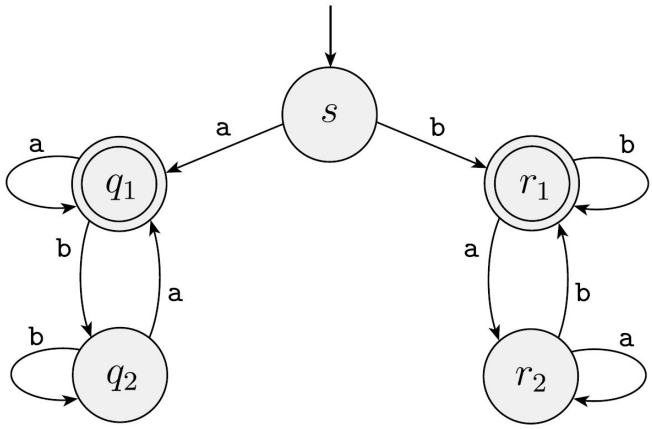

Abbildung 2. Ein weiterer Beispielautomat.

kritische Anwendungen interessant macht.

Bei größeren Automaten ist jedoch die objektorientierte Implementierung zu bevorzugen. Durch die Kapselung in Objekten, wird sehr viel Komplexität reduziert und Übersichtlich gestaltet. Das Anpassen des Automaten, oder Ergänzen von weiteren Zuständen, Übergängen oder Eingaben ist sehr einfach zu realisieren. Auch das Iterieren über die Eingabefolge kann auf kurze zwei bis drei Codezeilen reduziert werden. Kann die Anwendung die Einbusen in der Performanz verkraften, so wird diese Variante auch schon bei kleineren Automaten aufgrund der genannten Vorteile interessant.

Durch immer größer werdende Softwareprojekte wird immer stärker versucht die Codebasis übersichtlich und einfach zu halten. Daher ist es auch wichtig, Daten vom Programmcode zu trennen. Das Modell des endlichen Automaten muss nicht im Code fest codiert sein. besser wäre es, ihn davon zu separieren. Auf diese Weise können auch Nicht-Programmierer den Automaten modellieren und erstellen. Diese Möglichkeit bietet die Framework-Implementierung. Wurde bereits eine objektorientierte Variante implementiert ist es nicht mehr viel Aufwand, das Framework zu schreiben. Auf diese Weise könnte man den Automaten mit einem Tool erstellen und exportieren und sofort im Programm einbinden. Durch das Trennen von Daten und Code kann das selbe Automaten XML-Modell ohne Änderungen auch in Programmen mit anderen Programmiersprachen verwendet werden. Die Vorteile, welche die objektorientierte Variante mit sich bringt, gehen hier nicht verloren.

Zusammenfassend wird die prozedurale Variante demnach hauptsächlich für kleinere Automaten oder in Performanz kritischen Programmen verwendet. Wird der Automat etwas größer, so sollte auf die objektorientierte Variante zurückgegriffen werden und um zusätzlich zu der erleichterten Änderbarkeit und Übersichtlichkeit die Daten noch vom Programmcode zu trennen sollte die Framework-Implementierung verwendet werden.

2. Anwendungsbeispiele: Digitaluhr

Im folgenden wird die Thematik anhand eines praktischen Beispiels behandelt. Hierfür wird die Steuerung einer einfachen Digitaluhr mit integriertem Wecker verwendet. Eine Digitaluhr bietet ein sehr gutes Beispiel für einen endlichen Zustandsautomaten. Mit den begrenzten Eingabemöglichkeiten hat jeder Knopf in den verschiedenen Zuständen der Uhr eine andere Funktion.

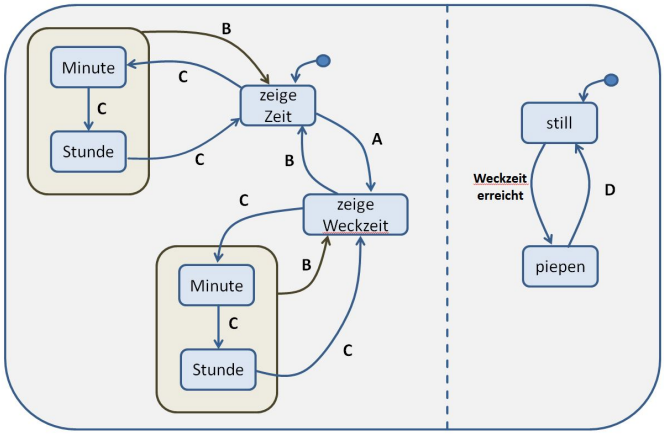

Abbildung 3. Zustandsdiagramm einer Digitaluhrsteuerung.

Abbildung 3 zeigt den zugrundeliegenden endlichen Automaten, anhand welchem die Uhr implementiert wird. Zur Umsetzung ist es allerdings notwendig, die Zustände in gewisse Hierarchien einzuteilen, sodass parallele und zusammengesetzte Zustände möglich sind.

2.1 Hierarchische Automaten

Für diese Uhr ist es wichtig, dass gewisse Zustände parallel aktiv sein können. Hierzu wird ein Überzustand herangezogen, dessen Unterzustände gleichzeitig aktiv sind, solang man sich im Überzustand befindet. Jeder Unterzustand besitzt einen Standardstartzustand, in welchem er sich anfänglich befindet, wenn kein anderer explizit angewählt wurde. Solange kein generalisierter Zustandsübergang des Überzustandes ausgelöst wird, oder einer der Unterzustände zu einem Zustand wechselt, welcher nicht Teil des Überzustandes ist, bleiben die parallelen Zustände aktiv. Bei dem Übergang eines inneren Zustands zu einem äußeren, können Bedingungen an die Zustände der parallelen Gruppen gesetzt werden. Diese werden in runden Klammern an die Pfeile geschrieben. Wie in Abbildung 3 zu sehen ist, befindet sich die Uhr jederzeit in zwei parallelen Zuständen. Die zwei Gruppen an Zuständen werden in der Darstellung durch die gestrichelte Linie separiert. Die Anzeige und das Einstellen der Zeiten verläuft parallel zum Weckvorgang des Weckers.

Neben den parallelen Zuständen sind auch zusammengesetzte Zustände für die Umsetzung der Uhr hilfreich. Hat man eine Menge an Zuständen, welche alle einen Übergang zum

selben Zustand haben und die selben Übergangsbedingung erfüllen müssen, so kann man diesen Übergang von der Menge der Zustände abstrahieren und zu einem Überzustand zusammenfassen. Befindet sich der Automat in diesem Überzustand, so befindet er sich immer in genau einem Unterzustand. Bei sehr großen endlichen Automaten sind diese zusammengesetzten Zustände essentiell, um die Komplexität gering zu halten und auch die Anzahl der Pfeile im Diagramm deutlich zu reduzieren. In Abbildung 3 sind zwei zusammengesetzte Zustände zu sehen. Sie umschließen jeweils die `Minute` und `Stunde` Zustände der aktuellen Zeit und der Weckzeit.

Die einzelnen Zustände der Uhr sind im Folgenden genauer beschrieben worden:

- `Still`
  Der Wecker der Uhr befindet sich anfänglich in diesem Zustand. Sobald die Weckzeit erreicht wurde, wird in den Zustand `Piepen` gewechselt.
- `Piepen`
  Solange sich der Wecker in diesem Zustand befindet, piept er. Wird der `D`-Knopf gedrückt, hört er auf und wechselt in den Zustand `Still`.
- `Zeige Zeit`
  Dieser Zustand ist der Startpunkt der Uhr. Solange sie sich in diesem Zustand befindet, wird die aktuelle Uhrzeit angezeigt. Mit dem `A`-Knopf wird in den Zustand `Zeige Weckzeit` gewechselt. Um die Zeit einzustellen, drückt man den `C`-Knopf.
- `Zeige Weckzeit`
  Wenn sich die Uhr in diesem Zustand befindet, wird die Zeit angezeigt, zu welcher der Wecker zu piepen beginnt. Über das Drücken des `B`-Knopfes gelang man in den Zustand `Zeige Zeit`. Um die Weckzeit einzustellen, drückt man den `C`-Knopf.
- `Minute`
  Dieser Zustand existiert zweimal, einmal für die Zeit und für Weckzeit. Über den <<, bzw. >>-Knopf können hier die Minuten eingestellt werden. Mit dem `C`-Knopf wird die Einstellung übernommen und es wird in den Zustand `Stunde` gewechselt. Mit dem `B`-Knopf geht es wieder in den vorherigen Anzeigezustand ohne die Änderungen zu übernehmen.
- `Stunde`
  Dieser Zustand stellt äquivalent zum Zustand `Minute` die Stunden der Zeit ein. Durch drücken des `C`-Knopfes werden die Änderungen übernommen und es wird in den Anzeigezustand gewechselt. Mit dem `B`-Knopf werden die Änderungen verworfen und ebenfalls in den Anzeigezustand gewechselt.

2.2 Implementierung

Zur Implementierung dieser Uhr wurde die Framework Variante gewählt. So fällt das modellieren des Automaten recht leicht, in der gewohnten XML Struktur. Für die parallelen und zusammengesetzten Zustände wurde die oben beschriebene Implementierung um zwei weitere Klassen, `ParallelState` und `ComposedState`, ergänzt. Hierbei leitet der parallele Zustand die Events die ihm übergeben werden an allen ihm zugewiesenen Unterzuständen weiter. Der zusammengesetzte Zustand leitet die Events immer nur an den aktuellen Unterzustand weiter. Sollten die Unterzustände das Event nicht bearbeiten, so macht es der Oberzustand. Auch das XML-Schema musste angepasst werden und wurde um die beiden Typen `Composed` und `Parallel` ergänzt. Diese erben die Eigenschaften von dem `State` Typen und ergänzen sie um die Möglichkeit ihnen Zustände zuzuweisen. Die Erweiterungen sind auf Listing 5 zu sehen. Modelliert man nun die Uhr ergibt sich für die Zustände der Aufbau in Listing 6. Durch diesen Aufbau kommen die Stärken des Frameworks sehr schön zur Geltung. Der eigentliche Code ist minimal und das Modell der Uhr ist auch für Menschen ohne Programmiererfahrung verständlich.

Die Logik der Uhr wird in den Aktionen festgelegt. Um hier eine Schnittstelle zwischen dem Code des eigentlichen Automaten und dem Teil der Anwendung, welche die grafische Benutzeroberfläche übernimmt, wurde eine Klasse `ClockAction` geschrieben, welche eine statische Methode hat, über die eine Referenz auf das Display gesetzt werden kann. Neben dem statischen Attribut dieser Referenz besitzt sie noch zwei Integer Attribute, welche dazu verwendet werden beim Eintritt in einen der beiden zusammengesetzten Zeiteinstellungszuständen die aktuelle Zeit zwischenzuspeichern, um sie erneut zu setzen, sollte dieser Vorgang abgebrochen werden. Zu Begründen ist dieses Vorgehen damit, dass das Framework nichts von der Benutzeroberfläche weiß und sie daher den einzelnen Aktionen nicht bei ihrer Erzeugung mitgeben kann. Die in dieser Implementierung verwendeten Aktionen sind alle Übergangsaktionen. Befindet sich die Uhr in den Zuständen `Zeige Zeit` oder `Zeige Weckzeit` und werden die Eingaben A bzw. B getätigt, so wird bei dem Übergang entsprechen die Aktion `ZeigeWeckzeit` oder `ZeigeZeit` ausgeführt. Sie rufen in diesem Fall nur die jeweilige Methode des Codes der Benutzeroberfläche auf. Wird in diesen Zuständen die Eingabe C getätigt, so gelangt die Uhr in den jeweiligen Einstellungszustand. Die Aktionen `EinstellenZeit` und `EinstellenWeckzeit` speichern dann die aktuelle Zeit in den oben genannten Attributen. Für beide Zeiten gibt es je eine Aktion um die Minuten und die Stunden zu inkrementieren oder dekrementieren. Hierfür werden die Methoden des Codes der Benutzeroberfläche genutzt. Werden bei dieser Einstellung mittels der Eingabe B die Änderungen verworfen, so setzen die Aktionen `EinstellenAbbrechenZeit` und `EinstellenAbbrechenWeckzeit` die Zeit auf die Werte, welche in den Attributen gespeichert waren. Wurde die Weckzeit erreicht, beginnt die Uhr zu piepen. Dies kann durch die Eingabe von D beendet werden, was die Aktion `StoppPiep` ausgeführt, welche die Methode zum Stoppen des Piepen aufruft.

3. Zusammenfassung

In dieser Arbeit wurde ein grobes Verständnis davon vermittelt, was Zustandsautomaten sind. Es wurden drei Implementierungsvarianten vorgestellt. Von der prozeduralen Implementierung, welche über `Switch-Case` Anweisungen realisiert wurde, über die objektorientierte Variante, welche die Zustände, Übergänge und Aktionen in Klassen abstrahierte, bis zum Framework-Ansatz, welcher mittels eines XML-Schemata eine Vorlage zur Modellierung eines Zustandsautomaten vorgab und so das Erstellen der Objektstruktur automatisieren konnte, wurden diese Implementierungen anhand eines Beispiels besprochen und einander gegenüber gestellt. Hierbei wurde festgestellt, dass sich die prozedurale Implementierung bei kleinen und sich nicht weiter verändernden Automaten anbietet. Bei mittleren Automaten, oder auch bei Automaten, welche in Zukunft noch weiter angepasst werden, eignet sich die objektorientierte Variante aufgrund der leichteren Übersichtlichkeit. Bei großen Automaten wurde geschlossen, dass die Framework-Implementierung am geeignetsten war, da so die Automaten leichter Modelliert werden konnten und den Code von der Struktur des jeweiligen Automaten trennt. Anschließend wurde gezeigt, wie mit Hilfe des Framework-Ansatzes ein Zustandsautomat für eine Digitaluhr realisiert werden kann.

Es wäre für weitere Arbeiten interessant zu Untersuchen, was für ein Gewinn durch den prozeduralen Ansatz gegenüber der objektorientierten Variante in Bezug auf die Laufzeit und den Speicherverbrauch zu erzielen ist. Zwischen dem objektorientierten und dem Framework-Ansatz würde sich in diesem Punkt lediglich die einmalige konstante Auslesung der XML-Datei und das Erstellen der Objektstruktur unterscheiden.

Literatur

[1] D. Harel, Y. Feldman, and M. Krieger. *Algorithmik*. Springer Berlin Heidelberg, 2006.

[2] Jilles Van Gurp and Jan Bosch. On the implementation of finite state machines. In *in Proceedings of the 3rd Annual IASTED International Conference Software Engineering and Applications, IASTED/Acta*, pages 172–178. Press, 1999.

[3] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, June 1987.

Erklärung zur Ausarbeitung

Hiermit erkläre ich, *Matthias Haselmaier (869995)*, dass ich die vorliegende Ausarbeitung selbstständig und ohne fremde Hilfe angefertigt habe und keine anderen als in der Abhandlung angegebenen Hilfen benutzt habe; dass ich die Übernahme wörtlicher Zitate aus der Literatur sowie die Verwendung der Gedanken anderer Autoren an den entsprechenden Stellen innerhalb der Arbeit gekennzeichnet habe. Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben kann.

_____

Unterschrift

1. Listings

```
<?xml version="1.0" encoding="UTF-8"?>

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <!-- Root-Element -->
    <xs:element name="fsm" type="FinitStateMachine" />

    <!-- FinitStateMachine-Type -->
    <xs:complexType name="FinitStateMachine">
        <xs:sequence>
            <xs:element name="states" type="States" minOccurs="1" maxOccurs="unbounded" />
            <xs:element name="transitions" type="Transitions" minOccurs="1" maxOccurs="
                unbounded" />
            <xs:element name="startState" type="State" minOccurs="1" maxOccurs="1" />
            <xs:element name="endState" type="State" minOccurs="0" maxOccurs="unbounded" />
        </xs:sequence>
    </xs:complexType>

    <!-- States-Type -->
    <xs:complexType name="States">
        <xs:sequence>
            <xs:element name="state" type="State" minOccurs="1" maxOccurs="unbounded" />
        </xs:sequence>
    </xs:complexType>

    <!-- Transitions-Type -->
    <xs:complexType name="Transitions">
        <xs:sequence>
            <xs:element name="transition" type="Transition" minOccurs="1" maxOccurs="
                unbounded" />
        </xs:sequence>
    </xs:complexType>

    <!-- State-Type -->
    <xs:complexType name="State">
        <xs:attribute name="name" type="xs:string" use="required" />
        <xs:attribute name="entryaction" type="xs:string" />
        <xs:attribute name="exitaction" type="xs:string" />
    </xs:complexType>

    <!-- Transition-Type -->
    <xs:complexType name="Transition">
        <xs:attribute name="event" type="xs:string" use="required" />
        <xs:attribute name="source" type="xs:string" use="required" />
        <xs:attribute name="target" type="xs:string" use="required" />
        <xs:attribute name="action" type="xs:string" />
    </xs:complexType>
</xs:schema>
```

Listing 4. Framework XML-Schema

```
<?xml version="1.0" encoding="UTF-8"?>

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    ...
    <!-- States-Type -->
    <xs:complexType name="States">
        <xs:sequence>
            <xs:element name="state" type="State" minOccurs="1" maxOccurs="unbounded" />
        </xs:sequence>
    </xs:complexType>
    ...

    <!-- Composed-Type -->
    <xs:complexType name="Composed">
        <xs:complexContent>
            <xs:extension base="State">
                <xs:sequence>
                    <xs:choice minOccurs="1" maxOccurs="unbounded">
                        <xs:element name="state" type="State" minOccurs="1" maxOccurs="
                            unbounded" />
                        <xs:element name="parallel" type="Parallel" minOccurs="1" maxOccurs=
                            "unbounded" />
                        <xs:element name="composed" type="Composed" minOccurs="1" maxOccurs=
                            "unbounded" />
                    </xs:choice>
                </xs:sequence>
                <xs:attribute name="first" type="xs:string" use="required" />
            </xs:extension>
        </xs:complexContent>
    </xs:complexType>

    <!-- Parallel-Type -->
    <xs:complexType name="Parallel">
        <xs:complexContent>
            <xs:extension base="State">
                <xs:sequence>
                    <xs:choice minOccurs="1" maxOccurs="unbounded">
                        <xs:element name="state" type="State" minOccurs="1" maxOccurs="
                            unbounded" />
                        <xs:element name="parallel" type="Parallel" minOccurs="1" maxOccurs=
                            "unbounded" />
                        <xs:element name="composed" type="Composed" minOccurs="1" maxOccurs=
                            "unbounded" />
                    </xs:choice>
                </xs:sequence>
            </xs:extension>
        </xs:complexContent>
    </xs:complexType>
</xs:schema>
```

Listing 5. Erweitertes XML-Schema

```
<states>
    <parallel name="Uhr">
        <composed name="Display" first="zeige Zeit">
            <state name="zeige Zeit" />
            <composed name="Zeiteinstellung" first="Minute Zeit">
                <state name="Minute Zeit" />
                <state name="Stunde Zeit" />
            </composed>
            <state name="zeige Weckzeit" />
            <composed name="Weckzeiteinstellung" first="Minute Weckzeit">
                <state name="Minute Weckzeit" />
                <state name="Stunde Weckzeit" />
            </composed>
        </composed>
        <composed name="Alarm" first="still">
            <state name="still" />
            <state name="piepen" />
        </composed>
    </parallel>
</states>
<transitions>
    <transition source="still" target="piepen" event="Weckzeit erreicht" />
    <transition source="piepen" target="still" event="D" action="clock.action.StoppPiep" />
    <transition source="zeige Zeit" target="Zeiteinstellung" event="C" action="clock.action.
        EinstellenZeit" />
    <transition source="zeige Zeit" target="zeige Weckzeit" event="A" action="clock.action.
        ZeigeWeckzeit" />
    <transition source="zeige Weckzeit" target="Weckzeiteinstellung" event="C" action="clock
        .action.EinstellenWeckzeit" />
    <transition source="zeige Weckzeit" target="zeige Zeit" event="B" action="clock.action.
        ZeigeZeit" />
    <transition source="Zeiteinstellung" target="zeige Zeit" event="B" action="clock.action.
        EinstellenAbbrechenZeit" />
    <transition source="Minute Zeit" target="Minute Zeit" event="hoch" action="clock.action.
        ZeitMinuteHoch" />
    <transition source="Minute Zeit" target="Minute Zeit" event="runter" action="clock.
        action.ZeitMinuteRunter" />
    <transition source="Minute Zeit" target="Stunde Zeit" event="C" />
    <transition source="Stunde Zeit" target="Stunde Zeit" event="hoch" action="clock.action.
        ZeitStundeHoch" />
    <transition source="Stunde Zeit" target="Stunde Zeit" event="runter" action="clock.
        action.ZeitStundeRunter" />
    <transition source="Stunde Zeit" target="zeige Zeit" event="C" action="clock.action.
        EinstellenSpeichernZeit"/>
    <transition source="Weckzeiteinstellung" target="zeige Weckzeit" event="B" action="clock
        .action.EinstellenAbbrechenWeckzeit" />
    <transition source="Minute Weckzeit" target="Minute Weckzeit" event="hoch" action="clock
        .action.WeckzeitMinuteHoch" />
    <transition source="Minute Weckzeit" target="Minute Weckzeit" event="runter" action="
        clock.action.WeckzeitMinuteRunter" />
    <transition source="Minute Weckzeit" target="Stunde Weckzeit" event="C" />
    <transition source="Stunde Weckzeit" target="Stunde Weckzeit" event="hoch" action="clock
        .action.WeckzeitStundeHoch" />
    <transition source="Stunde Weckzeit" target="Stunde Weckzeit" event="runter" action="
        clock.action.WeckzeitStundeRunter" />
    <transition source="Stunde Weckzeit" target="zeige Weckzeit" event="C" />
</transitions>
<startState name="Uhr" />
```

Listing 6. XML-Model der Digitaluhr