1. Implementierungsvarianten

Es werden nun im folgenden die prozedurale, objektorientierte und Framework Implementierung vorgestellt.
Abbildung 2 zeigt einen weiteren Beispielautomat mit mehrere Zuständen. Dieser kann die Events a und b verarbeiten.

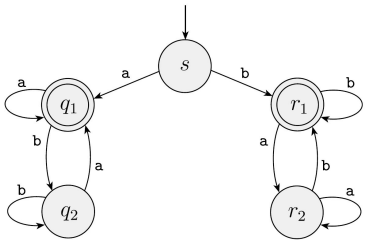

Abbildung 2. Ein weiterer Beispielautomat[4].

1.1 Prozedurale Implementierung

Bei dieser Art der Implementierung wird der aktuelle Zustand in einer Variable festgehalten. Sobald ein Ereignis ausgelöst wird, erfolgt über eine `switch-case`-Konstruktion die Abfrage nach dem aktuelle Zustand und resultiert ggf. mit einem Zustandsübergang in einen anderen Zustand.
Das nachfolgende Listing verdeutlicht diese Implementierungsvariante. Der Anfangszustand Q1 wird in einer Variable `state` gespeichert. Bei der Ereignisverarbeitung wird zunächst in der ersten switch-case-Konstruktion der aktuelle Zustand geprüft. Entspricht dieser Beispielsweise dem Zustand Q1 geht es in das nächste Konstrukt um das eingetretene Ereignis (Event) zu überprüfen und in Abhängigkeit davon den nächstmöglichen Zustand der Variable `state` zuzuordnen.
Die Klassifizierung ob es sich um eine gültige Ereignisfolge handelt wird anhand einer boolischen Variable zum Schluss festgestellt. Dabei erfolgt eine einfache Überprüfung der Variable `state`, ob sie sich in einem für den Automat gültigen Endzustand befindet.

```
String eventSequenz = "10";
String state = "Q1";
boolean istGueltigerEndzustand = false;
int index = 0;

while (index < eventSequenz.length()) {
        char event = eventSequenz.charAt(
            index);
        switch (state) {
        case "Q1": switch (event) {

                case '0': state = "Q1";
                break;
                case '1': state = "Q2";
                break;
                default:
                System.err.println("Falsches
                     Event");
                }
        break;

        case "Q2": switch (event) {
        ...
        }
        }


if (state.equals("Q1")){
        istGueltigerEndzustand = true;
} else {
System.out.println("Ungueltiger Endzustand!
    (Q2)");
}
return istGueltigerEndzustand;
}
```

Listing 1. Ein Auszug aus der Prozeduralen-Implementierung einer switch-case-Konstruktion für die Zustands- und Ereignisauswahl.

1.2 Objektorientierte Implementierung

Beim `State-Pattern` werden alle Zustände als eigene Klassen implementiert. Mit einem Interface lässt sich sicherstellen das die benötigten Ereignisse die bei einem Automaten auftreten können als Methoden vorhanden sind und Zustandsspezifisch implementiert werden müssen.

Listing 2 zeigt den Zustand Q1 als eigene Klasse und implementiert das besagte Interface. Bei einem möglichen Event 1, liefert der Automat einen Zustand als neues Objekt der Klasse Q2 zurück. Trifft hingegen das Event 0 ein, so findet keine Zustandsänderung statt.

```
public class Q1 implements Zustand{

  @Override
  public Zustand processEvent0(){
  return this;
  }

  @Override
  public Zustand processEvent1(){
  return new Q2();
  }
}
```

Listing 2. Ein Auszug aus der Objektorientierten-Implementierung mit dem Ansatz State-Pattern.

Eine weitere objektorientierte Möglichkeit ist eine Realisierung als `State-Graph`.

Dabei werden die Möglichen Events als eigener Datentyp (enum) implementiert (Listing 3).

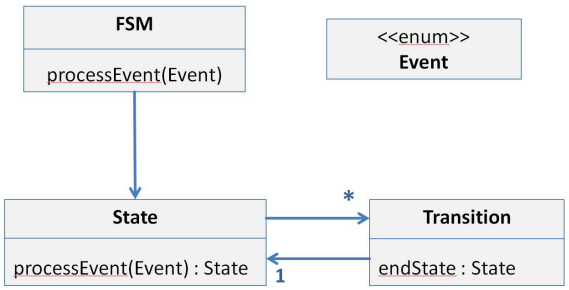

Abbildung 3. Entwurf eines Zustandsgraphen[4].

```
public enum Event {
 ZERO, ONE;
}
```

Listing 3. Mögliche Events.

Die State-Klasse stellt eine Methode `addTransition` bereit um den Events die Übergänge zuzuordnen.

Wird der Automat damit konfiguriert, ist es nun möglich über eine weitere Methode `processEvent`, das gewünschte Event anzufordern und bei Übereinstimmung den vorher bestimmten Zustandsübergang herbeizuführen.

```
public final class State {

 private final HashMap<Event, Transition>
     transitions = new HashMap<>();
 ...
 public void addTransition(Event event,
     Transition transition){
 this.transitions.put(event, transition);
 }

 public State processEvent(Event event){
 Transition transition = this.transitions.
     get(event);
 if (transition != null){
 return transition.getEndState();
 } else {
 return this;
 }
...
}
```

Listing 4. Ein Auszug aus der State-Klasse.

1.3 Framework Implementierung

Einen Systemübergreifenden Datenaustausch, der Hersteller- und Plattformunabhängig ist, erhält man durch den Einsatz von XML (Extensible Markup Language).

Zustandsautomaten können mit XML beschrieben werden, wie in Listing 5 zu sehen ist. Damit erhält man auf einfache Art und Weise eine vollständige Konfiguration eines Automaten. Unter `states` wird der jeweilige `state` mit seinem Namen eingetragen. Äquivalent dazu werden die möglichen Übergänge unter `transitions` mit ihrer `source`, dem zugehörigen `target` und dem gewünschten `event` vermerkt.

Der Startzustand des Automaten wird unter `startState` festgelegt, sowie die gültigen Endzustände unter `endState` eingetragen.

Mithilfe eines zugehörigen XML-Schema kann ein Parser das XML-Dokument auf Gültigkeit überprüfen und bei Vollständigkeit eines korrekt beschriebenen Automaten diesen daraus erzeugen.

```
<?xml version="1.0" encoding="UTF-8"?>
<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 5. XML-Konfiguration des Automaten aus Abbildung 1.

Die Konfiguration eines Automaten, der in einer XML-Datei beschrieben wird, kann durch eine Framework Implementierung eingelesen und erzeugt werden. Der nachfolgende Auszug aus einem Listing stellt eine Möglichkeit der Implementierung dar. Hierbei werden Beispielsweise alle `States` eingelesen und in einer Collection der FSM gehalten. Ebenso erfolgt eine ähnliche Prozedur mit den möglichen `Transitions` und `Events` dieses Automaten. Daraufhin werden mithilfe des Zustandsgraphen die `Transitions` und `Events` den zugehörigen `States` zugeordnet.

```
...
// Read States
NodeList stateList = document.
    getElementsByTagName("state");
for (int tmp = 0; tmp < stateList.getLength
    (); tmp ++)
{
Node nNode = stateList .item (tmp);
if ( nNode.getNodeType() == Node.
    ELEMENT_NODE)
```

```
{
Element eElement = (Element) nNode;
String name = eElement.getAttribute("name");
State state = new State(name);
states.add(state);
}
}
...
```

Listing 6. Auszug aus dem XMLParser.

Mit Hilfe dieser Framework Implementierung lässt sich somit aus jeder gültigen XML-Datei, die einen Automaten vollständig und korrekt beschreibt, ein Automat erstellen.

1.4 Evaluierung der Implementierungsansätze

Prozedural: Eine `Prozedurale Implementierung` eignet sich vor allem, wenn das Projekt klein und überschaubar ist. Man geht dabei intuitiv und verständlich vor und erhält eine kompakte einzige Klasse. Allerdings muss der bestehende Code bei einer geplanten Erweiterung geändert werden. Ebenso ist ein größeres Projekt damit kaum zu bewerkstelligen, da die Übersicht schnell verloren geht und das System dadurch nur noch schwer wartbar wird. Dies spiegelt sich vor allem auch in den `if-else`-Konstrukten wieder.
Objektorientiert: Bei der objektorientierten Implementierung `State-Pattern` erhält man hingegen eine gute Übersicht. Jeder Zustand wird explizit über eine eigene Klasse definiert. Der Code ist dadurch leicht verständlich und inkonsistente Zustände werden somit vermieden, denn der neue Zustand erfolgt über einen Rückgabewert. Neue Zustände können auch einfacher ins bestehende System integriert werden, ohne dabei den bestehenden Code aufwändig ändern zu müssen. Als Nachteil dieser Implementierung erweist sich die Anzahl der Klassen, die mit jedem neuen Zustand wächst.

Mit dem `Zustandsgraph` bzw. `State-Graph` der weiteren objektorientierten Implementierung erhält man ebenso eine gute Übersicht. Hierbei werden auch weniger Klassen benötigt als beim State-Pattern, da sich die Zustände nicht als eigene Klassen sondern in einem einzigen Datentyp (enum) festlegen lassen. Änderungen am zustandsabhängigen Verhalten betreffen lediglich eine Zustandsklasse und nicht den Context, was für die Änderung/ Erweiterung dieser Variante spricht.

2. Anwendungsbeispiele: Digitaluhr

Im folgenden Abschnitt soll nun die in Abbildung 4 angezeigte Steuerung einer Digitaluhr als konkrete Anwendung dienen und entsprechend implementiert werden. Diese lässt sich über die Buttons A bis D steuern. Dabei ist zu beachten das der Wecker als eigenständiger nebenläufiger Parallelzustand implementiert wird (als eigener Automat).

2.1 Hierarchische Automaten

Die Abbildung 4 zeigt die Statechart einer abgespeckten Steuerung einer Digitaluhr. Diese wird in zwei Automaten, einen für

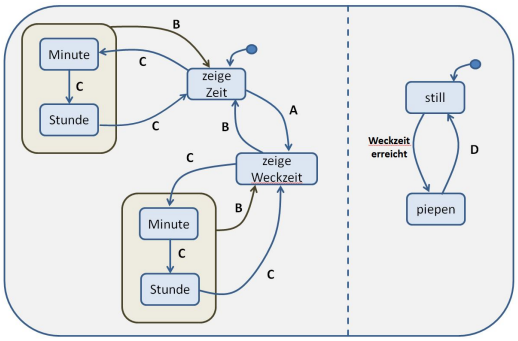

Abbildung 4. Statechart einer Digitaluhrsteuerung[4].

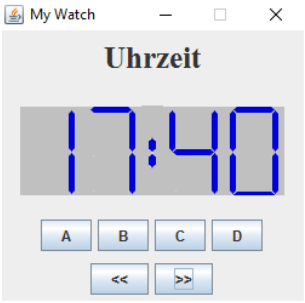

Abbildung 5. GUI der Digitaluhr.

den Wecker und einen für die eigentliche Uhranzeige, aufgeteilt. Dadurch erhält man zwei nebenläufige Parallelzustände der Automaten.

Der Automat der für die Uhrzeit zuständig ist kann mithilfe der Events A, B und C in die jeweiligen anderen Zustände übergehen, um so z.B. die Uhrzeit einzustellen. Ein Zustandsübergang kann dabei nur erfolgen, wenn der Automat sich vorher in dem dafür vorgesehenen Zustand befand und das entsprechende Event ausgelöst wurde. Somit wird gewährleistet das der Automat „Uhr“ in dem Zustand „zeigeZeit“ bei einem eingetretenen Event „C“ in den Unterzustand „UhrMinute“ übergeht und in keinen anderen Fehlzustand.

Der Wecker hingegen besitzt nur zwei Zustände und startet in dem Zustand „Still“. Dieser wird bei erreichen der eingestellten Weckzeit in den Zustand „Piepen“ überführt. Danach kann ein Zustandswechsel und die Abschaltung des Wecktons nur über den D-Button erfolgen.

2.2 Implementierung

Bei der Implementierung kann zur Strukturierung der Anwendung das MVC-Muster (Model View Controller) angewendet werden. Abbildung 6 illustriert wie so eine Event Verarbeitung ablaufen kann. Dabei enthält das Model die eigentlichen Daten und somit auch die Zustandsautomaten (Listing 7). Wird auf der Oberfläche nun ein Event z.B. „A“ ausgelöst, ändert der Controller den Zustand des einen Automaten über das Model (Listing 8). Anschließend teilt das Model der View mit, die Anzeige zu ändern.

In dem Listing 7 ist der Zustandsübergang des „Wecker“

Automaten abgebildet. Sollte die eingestellte Weckzeit erreicht sein, so muss neben dem einschalten des Pieptons auch der Zustand des Automaten von „Still“ zu „Piepen“ übergehen. Eine entsprechende Event-Verarbeitung wird dabei angestoßen.

```
public class Watch
{
...
private UhrFSM uhrzeitFSM = automat.uhr.
    UhrFSM.getUhrFSMInstance();
private WeckerFSM weckzeitFSM = automat.
    wecker.WeckerFSM.getWeckerFSMInstance();

...

 // Weckzeit erreicht
if (stunde == alarmStunde && minute ==
    alarmMinute)
{
Watch.this.startBeep();
//Sobald die Weckzeit erreicht ist soll der
    Zustand des Weckers geaendert werden.
weckzeitFSM.process(WeckerEvent.
    WECKZEIT_ERREICHT);
}
...
}
```

Listing 7. Ein Auszug aus der Watch-Klasse.

```
@Override
public void event(String event)
{

String currentStateUhrzeitFSM = display.
    getWatch().getUhrFSM().getState().getName
    ();

System.out.println( event );
switch( event )
{
case "A":
if (currentStateUhrzeitFSM.equals("
    showUhrzeit")){
display.showWeckzeit();
}

display.getWatch().getUhrFSM().process(
    UhrEvent.A);

break;
case "B":
...
```

Listing 8. Ein Auszug aus dem Controller für die Event-Verarbeitung.

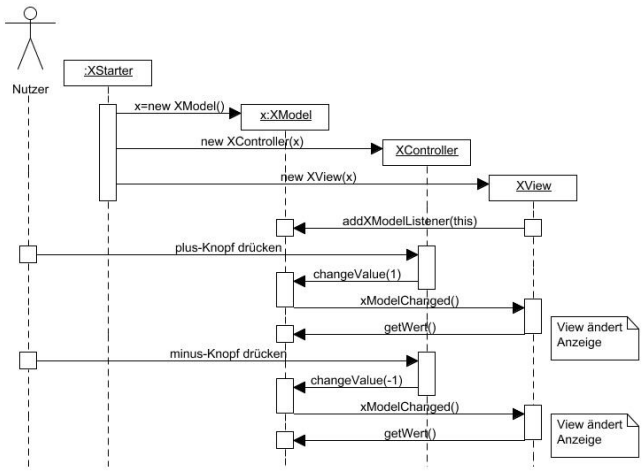

Abbildung 6. Model-View-Controller Muster[5].

3. Zusammenfassung

Bei kleineren Projekten ist es möglich, diese auch durch eine Prozedurale Implementierung zu realisieren. Allerdings überwiegen die Vorteile von den objektorientierten Varianten die sich auch im Laufe der Jahre durchgesetzt haben. Bei der Implementierung eines endlichen Zustandsautomaten trägt die ausgewählte Implementierungsvariante eine tragende Rolle, da die grundlegende Software Architektur bei einer Fehlentscheidung im Laufe der Zeit nur sehr schwierig zu ändern oder anzupassen ist. Trotzdem sollte man auch bei Feststellung eines Fehlers einen Umbau der Struktur bevorzugen. Ansonsten entsteht ein schlecht strukturierter, kaum wartbarer Automat, der unter Umständen nur sehr schwierig erweitert werden kann.
Anhand eines konkreten Anwendungsbeispiels der Steuerung einer Digitaluhr wurde der Einsatz von Automaten näher dargelegt und erörtert. Hierbei wurde das MVC-Muster zur besseren Strukturierung einer Anwendung umgesetzt.

Im weiteren lohnt es sich die sogenannten Entry- bzw. Exit-Aktionen näher anzuschauen. Dabei können im Zuge des Zustandsübergangs bestimmte Exit-Aktionen des Quellzustands, sowie Entry-Aktionen des Zielzustands ausgeführt werden[3]. Somit lässt sich z.B. ein Blinken der Anzeige als Aktion bei einem Eintritt in den Einstellungszustand der Uhr realisieren.

Literatur

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

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

[3] Jilles Van Gurp and Jan Bosch. On the implementation of finite state machines. Press, 1999.