UiO : Department of Informatics
University of Oslo

Combined Modeling and Programming with State Machines

Kjetil Andresen

Master's Thesis Spring 2014

Combined Modeling and Programming with State Machines

Kjetil Andresen

1st May 2014

Abstract

As part of a more general effort on the design of a combined modeling and programming language, support for modeling and programming with state machines has been investigated. While earlier approaches have represented composite states/state hierarchies by means of class inheritance hierarchies between state classes, it is demonstrated that it is more powerful to support composite states by means of delegation. The inheritance mechanism may then be used to represent real specialization of state machines.

Contents

1 Introduction **1**
1.1 Background and motivation . . . 1
1.2 Contribution . . . 1
1.3 Research approach and method . . . 2
1.4 Structure of thesis . . . 3

2 State machines **5**
2.1 Introduction . . . 5
2.2 UML state machines . . . 6
2.2.1 States, events and transitions . . . 6
2.2.2 Composite states . . . 6
2.2.3 History . . . 7
2.2.4 Entry and exit points . . . 8
2.2.5 Entry and exit actions . . . 8
2.2.6 Orthogonality . . . 9
2.3 State machines in programming . . . 9
2.3.1 State design pattern . . . 10
2.3.2 ThingML . . . 12

3 State machine framework **15**
3.1 Introduction . . . 15
3.2 Media player example . . . 16
3.3 State hierarchy structure and algorithms . . . 17
3.3.1 State hierarchy analysis . . . 17
3.3.2 Entry and exit actions . . . 19
3.3.3 History . . . 21
3.3.4 Pseudo states . . . 22
3.3.5 State transitions . . . 23
3.4 Framework overview . . . 23
3.4.1 A basic framework . . . 25
3.4.2 Making the framework generic . . . 26
3.4.3 State references . . . 27
3.4.4 `StateMachine` . . . 27
3.4.5 Pseudo states in the framework . . . 28
3.4.6 `IVertex` . . . 29
3.4.7 Pseudo states . . . 30
3.4.8 `IState` . . . 31

3.4.9 `State` . . . 31
3.4.10 Events and transitions . . . 32
3.5 Summary . . . 34

4 State hierarchy by inheritance **35**
4.1 Introduction . . . 35
4.2 Overview . . . 35
4.2.1 Framework classes . . . 37
4.3 Framework implementation . . . 37
4.3.1 State hierarchy traversal . . . 37
4.3.2 `StateMachine` . . . 37
4.3.3 `State` . . . 39
4.4 Framework usage . . . 41
4.4.1 Design principles . . . 41
4.4.2 Implementation of a media player . . . 41
4.5 Summary . . . 45

5 State hierarchy by delegation **47**
5.1 Introduction . . . 47
5.2 Delegation . . . 47
5.3 Delegation in Java . . . 49
5.3.1 Delegation pattern . . . 49
5.4 Overview . . . 50
5.4.1 Framework classes . . . 51
5.5 Framework implementation . . . 51
5.5.1 State hierarchy traversal . . . 51
5.5.2 `StateMachine` . . . 51
5.5.3 `State` . . . 54
5.5.4 State transitions . . . 54
5.6 Framework usage . . . 55
5.6.1 Design principles . . . 55
5.6.2 Implementation of a media player . . . 56
5.6.3 Implicit delegation by inheritance . . . 58
5.7 Summary . . . 59

6 Specialization of state machines **61**
6.1 Introduction . . . 61
6.2 A specialized switch . . . 62
6.3 Using the extensible state machine pattern . . . 62
6.3.1 Implementing the on/off switch . . . 62
6.3.2 Implementing the media switch . . . 64
6.3.3 Summary of design principles . . . 69
6.4 Using inheritance and virtual classes . . . 71
6.4.1 Virtual classes . . . 71
6.4.2 Java with virtual classes . . . 72
6.4.3 Modified delegation framework . . . 72
6.4.4 Implementing the on/off switch . . . 73
6.4.5 Implementing the specialized media switch . . . 75

6.4.6 Summary of design principles 76
6.5 Summary 77

7 Conclusion and future work **79**

A Framework source code **81**
A.1 Directories and source files 81
A.2 Running the examples 81

List of Figures

2.1 On/off switch with states, events and transitions 6
2.2 Media player with composite state and compound transition . 7
2.3 Media player with shallow history 8
2.4 Media player with entry point 8
2.5 Media player with entry/exit actions 9
2.6 Media player with orthogonal state 9
2.7 The state design pattern 10
2.8 On/off switch class diagram 11

3.1 UML state machine representing a media player 16
3.2 State hierarchy represented as a tree structure 17
3.3 State hierarchy analysis 18
3.4 Entry/exit actions 19
3.5 Saving shallow history 21
3.6 Saving deep history 22
3.7 The process of changing state 24
3.8 State design pattern applied to the media player 25
3.9 Use of the state machine framework 25
3.10 Framework extended with `IState` 26
3.11 Framework with pseudo states 29
3.12 Event with corresponding transition represented as a method 33
3.13 Statechart vs. UML state machine transitions 33

4.1 Class hierarchy of the media player using inheritance 36

5.1 `Boss/Worker` sequence diagram 48
5.2 Combining inheritance and delegation 49
5.3 Delegation framework class hierarchy 50
5.4 State hierarchy represented by delegation links between objects 51
5.5 Automatic state instantiation 53
5.6 Implicit delegation by inheriting `MediaPlayerState` 58

6.1 Specialized media switch 62
6.2 Class hierarchy of the on/off switch 63
6.3 Specialization by inheritance 64
6.4 Specialization of states 64
6.5 Class hierarchy of the media switch 67

6.6 Specialized state machines using the extensible state machine pattern 71
6.7 Virtual classes with subclasses 72
6.8 Modified delegation framework with virtual classes 73
6.9 On/off switch with virtual classes 73
6.10 Media switch with virtual classes 75

List of Tables

3.1 Methods of StateMachine that are visible for the user 28
3.2 Methods of StateMachine that are visible for the framework . . 29
3.3 Methods of the interface IVertex 30
3.4 Methods of EntryPoint and ExitPoint 30
3.5 Methods of ShallowHistory and DeepHistory 31
3.6 Methods of the interface IState 31

4.1 Overview of the inheritance framework classes and constructors 38
4.2 User methods of State 40

5.1 Overview of the delegation framework classes and constructors 52
5.2 User methods of State 54

Acknowledgments

A special thanks to my supervisor, Birger Møller-Pedersen, for guiding me through the entire period, and giving me good advice and feedback. I want to thank my family and friends for support. Finally, a big thanks to my fellow students and friends at the 8th floor of Ole-Johan Dahls hus.

Oslo, Norway. May 1, 2014.

Kjetil Andresen

Chapter 1

Introduction

1.1 Background and motivation

In general purpose languages like *Java* [6] there is no direct support for state machines. For these there are design patterns like the state design pattern [4] which support simple state machines, i.e. state machines with the most primitive mechanisms and flat *state hierarchy*.

Harel statecharts [8] introduced the notion of state hierarchies with depth, where a state, referred to as a *composite state*, may contain substates. The main feature of composite states is that events with corresponding transitions defined for an enclosing state by default apply all contained substates, unless the substates specify otherwise. Statecharts later led to the introduction of UML state machines [16], which has many of the same type of mechanisms, and several new ones. Probably the most common mechanisms are composite states, *history*, *entry/exit actions*, *entry/exit points*, and recently also *specialization* of state machines has been paid attention to.

It is proposed in [14] that a definition of a combined modeling and programming language should be researched. In such a language, it could be argued that e.g. full UML state machines should be supported. In a perfect world, the language should be designed from scratch instead of just extending already existing languages with new mechanisms. E.g. extending Java with the full UML state machine specification will add over 20 new keywords to the language, which will clutter the language, see [7]. However, before attempting to design a combined modeling and programming language, it is important to learn from already existing language mechanisms.

1.2 Contribution

As a contribution to the research of designing a combined modeling and programming language, this thesis will investigate the support for modeling and programming with state machines by using already existing programming language mechanisms. While earlier approaches have represented state hierarchies by means of class inheritance hierarchies, see

e.g. [12], it is demonstrated that it is more powerful to support composite states by means of state objects linked by *delegation*. Inheritance may then be used to represent real specializations of state machines.

1.3 Research approach and method

Previous approaches have shown that with the state design pattern it is possible to support composite states in isolation by allowing contained states to inherit their enclosing state, see [12]. By representing events with corresponding transitions as *virtual methods*, the event methods of a composite state are inherited by contained states, unless the contained states override the event methods, and this is exactly how composite states are supposed to work. Even though the inheritance approach sounds promising, [18] recognize that the approach becomes difficult when introducing new mechanisms, e.g. entry/exit actions.

A starting point for this thesis was to look at exactly why inheritance did not work as state hierarchy representation. In order to investigate this, a framework intended to be used together with the state design pattern was implemented in Java. Java was chosen because it is a widely used object-oriented programming language. The purpose of the framework was to add support for history, entry/exit actions and entry/exit points in the state design pattern. The framework combined with representing composite states as subclasses was then investigated to pin point the advantages and disadvantages of using inheritance as a means to state hierarchy representation.

With the problems of the inheritance approach being uncovered, another approach was investigated where state objects are linked together by delegation. By having the delegation link of a contained state denote its enclosing state, methods not explicitly defined in the contained state will be delegated to its enclosing state. Transitions defined in event methods of enclosing states will then by default apply for contained states.

The added benefit of representing composite states by delegation is that inheritance may be used to define state machines as specializations of more general state machines. The *extensible state machine pattern* [1] gives a solution on how to specialize state machines with the state design pattern. By applying the extensible state machine pattern to the delegation approach, it is possible to create specialized state machines with the more advanced mechanisms supported by the framework. As the pattern showed to be fairly difficult to use, the combination of inheritance and *virtual classes* [13] was applied to the pattern, simplifying things dramatically.

A recurring example used in this thesis is an old fashioned media player state machine that plays CDs and cassettes. The media player is used as an example because it is using most of the mechanisms supported by the framework.

1.4 Structure of thesis

The thesis has the following structure.

Chapter 2: State machines Gives an introduction to state machines, showing how they are used in modeling and in programming.

Chapter 3: State machine framework Describes and implements a framework that is used together with the state design pattern in order to support advanced mechanisms found in UML state machines.

Chapter 4: State hierarchy by inheritance Investigates the possibility of using *inheritance* as a means of representing state hierarchy in the state machine framework described in chapter 3.

Chapter 5: State hierarchy by delegation Investigates the possibility of using *delegation* as a means of representing state hierarchy in the state machine framework described in chapter 3.

Chapter 6: Specialization of state machines Combines the delegation approach from chapter 5 with the notion of using inheritance as a means of representing specialized state machines. Two approaches will be discussed, one using the extensible state machine pattern, and one that combines the use of inheritance and virtual classes.

Chapter 7: Conclusion and future work Gives a summary of the results of previous chapters, and discuss possibilities for future work.

The *related work* of this thesis will be covered in the chapters where it naturally belongs.

Chapter 2

State machines

2.1 Introduction

State machines can be used to model reactive systems, which are systems that must react on external events, e.g. mobile phones, digital watches and media players.

In modeling, there are developed many different notations for state machines, like *Harel statecharts* [8] and the more modern extension of statecharts, namely *UML state machines* [16]. These notations has added some properties to state machines that makes it more useful in software engineering. While conventional state machines, e.g. *finite state machines* used in mathematics, are useful for modeling simple systems, statecharts and UML state machines support several abstraction mechanisms that makes it possible to manage more complex systems.

In programming, there are many ways to represent different types of state machines. Programmers may use general purpose languages to represent simple state machines by using e.g. *design patterns* [4]. To get support for more advanced mechanisms like those found in UML, there are *frameworks*[1]. There exist a few language extensions that adds state machine specific keywords to a language, see e.g. [7]. There are also *domain specific languages* (DSL) that are programming languages specialized for state machines.

The purpose of this chapter is to give an introduction on how to model with state machines, and a few alternative representations of state machines in programming. Section 2.2 will, from a modeling perspective, give a brief introduction of UML state machines. Section 2.3 will, from a programming perspective, introduce state machines by looking at the *state design pattern*, and the DSL named *ThingML*.

[1] E.g. http://northstatesoftware.com/uml-state-machine-code-generation-framework.html

2.2 UML state machines

In the UML specification [16] there are two types of state machines, namely *protocol state machines* and *behavioral state machines*. The latter, which will be the type of state machine used in this thesis, is based on Harel statecharts which is an extension of the conventional finite state machines, adding features like *state hierarchy* and *orthogonal regions* among others.

While the UML specification gives a technical overview of state machines, this section will give a more practical introduction, showing the graphical notation and give simple examples where one of the examples will be extended and used in later chapters. Going through all the mechanisms with their details will be out of the scope of this thesis as only a subset of them will be investigated. For more extensive information about UML state machines, see e.g. [16, 17, 3].

2.2.1 States, events and transitions

A state in UML is illustrated as a rounded rectangle with a label placed inside that represents the name of the state. States are connected by transitions, where a transition is a directed arrow from a source state to a target state. A transition consist of a *triggering event*, a *guard* and an *action*. The guard is optional, and is a condition that must be true in order for a transition to be executed. The action is also optional, and is action code that will be executed right before the state is changed. The syntax of a transition is `Event[Guard]/Action`.

In order to illustrate how UML state machines works, a on/off switch will be developed in stages, creating a old fashioned media player with the possibility to play CDs and cassettes. The on/off switch of figure 2.1 has two states named `Off` and `On`. The black circle connected to `Off` is called the *initial pseudo state*, and specifies the initial state of the state machine. The states are connected by two *transitions*. The transition pointing at `On` will be executed on the event `powerOn`, and the transition pointing at `Off` is executed on the event `powerOff`. The transitions have no guards or actions.

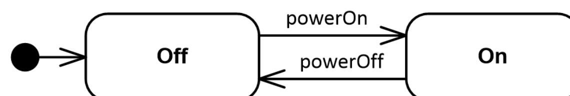


Figure 2.1: On/off switch with states, events and transitions

2.2.2 Composite states

A composite state is a state with contained states (also called substates). Composite states contain one or more *regions*, where a region is a part of either a composite state or a state machine. A composite state with two or more regions is called an orthogonal composite state, and will be the subject in section 2.2.6. Regions contains states and transitions, and zero or one initial pseudo state. E.g. the state machine in figure 2.1 has one region

with one initial state. The main benefit of composite states is abstraction, where it is possible to generalize a group of states into one composite state. Outgoing transitions from a composite state will by default apply to all contained states, unless substates specifies otherwise. Outgoing transitions from composite state will be referred to as *compound transitions*.

Figure 2.2 is an extension of the simple on/off switch, where `On` is transformed into a composite state with two modes, i.e. the states `CDMode` and `CassetteMode`. The state machine still start in the state `Off`. On the event `powerOn`, the media player will transition to `On`, and into the initial state of `On` which is `CDMode`. The media player can change mode by the event `changeMode`. At any time, the media player can turn off by the event `powerOff` which will execute the compound transition to `Off`.

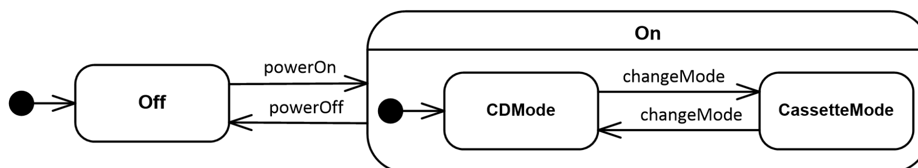


Figure 2.2: Media player with composite state and compound transition

2.2.3 History

History is a mechanism that remembers the last visited state of a composite state. There are two types of history pseudo states:

Shallow history A pseudo state that transition to the last visited state of the region it is part of. The shallow history pseudo state is illustrated as a circle with the label **H**.

Deep history A pseudo state that transition to the last visited substate of all the composite states that are in the same region as itself. Deep history does not include shallow history. The deep history pseudo state is illustrated as a circle with the label **H***.

History for a composite state will be saved when the composite state is exited. This means that the first time the state machine enters a composite state, history pseudo states have no history. Both history pseudo states therefore has an outgoing transition that represents the *default history*, i.e. the transition that will be executed if there is no history.

Figure 2.3 extends the media player even further by adding shallow history in the composite state `On`. The first time `On` is entered, the media player will transition to `CDMode`, because `CDMode` is the default history. If the current state is `CassetteMode` and the event `powerOff` occurs, followed by `powerOn`, the shallow history of `On` will be `CassetteMode`, which then will be entered.

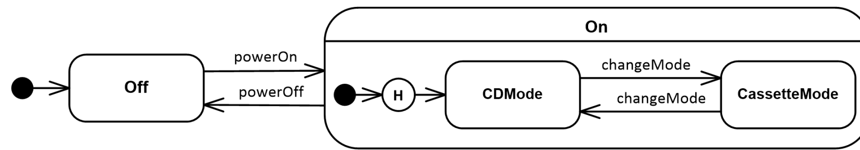


Figure 2.3: Media player with shallow history

2.2.4 Entry and exit points

In UML, a transitions from a state must point to a state in the same region, which means that it is not allowed for a transition to cross boundaries into composite states.[2] Entry points are pseudo states that are connected to composite states, and works as a link to a specific state inside the composite state it is connected to. Similarly, exit points are the opposite where it is possible to exit a composite state and transition to a target specified by the exit point.

For illustration purpose, figure 2.4 shows the use of entry points on the media player, even though it is not needed. Instead of `Off` transitioning to the boundary of the state `On` as in figure 2.3, the transition targets an entry point that is connected to the boundary of `On`. When entering the entry point, the state will automatically change to the shallow history pseudo state.

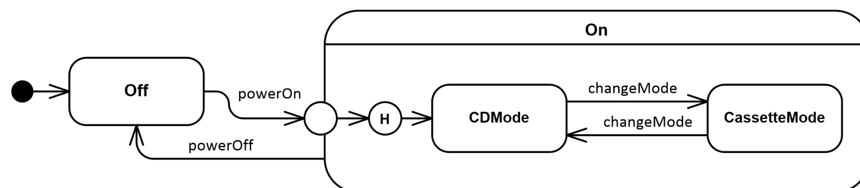


Figure 2.4: Media player with entry point

2.2.5 Entry and exit actions

Another useful mechanism is the ability to define *state actions*. Entry and exit actions are two types of state actions. Entry actions define actions that are executed when a state is entered, and exit actions are actions that are executed when a state is exited. The syntax of entry and exit actions are `Entry/Action` and `Exit/Action`, and are labeled inside the state rectangle.

Figure 2.5 makes an extension to the media player by adding entry and exit actions to the modes. When entering `CDMode` the method `cdMotorOn` will be executed automatically. When making a transition, either to `CassetteMode` or `Off`, the method `cdMotorOff` will be executed. `CassetteMode` has the same behaviour, but turns the cassette motor *on* and *off*.

[2]Transitions in Harel statecharts may target any state, even if contained in another composite state.

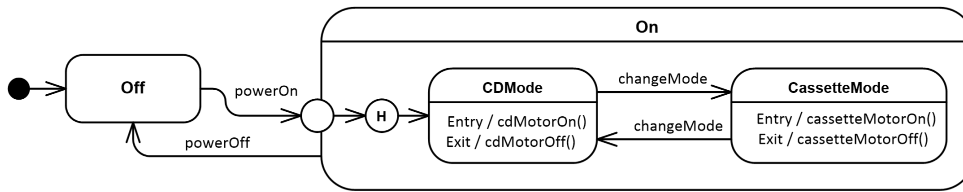


Figure 2.5: Media player with entry/exit actions

2.2.6 Orthogonality

A composite state may contain several regions. By having several regions, the state machine may be in several states at the same time. This mechanism is useful for concurrency, and to fragment the state machine into more manageable parts. There are mainly two pseudo states that are intended for orthogonal composite state, namely *fork* and *join*. A fork splits a single incoming transition into two or more outgoing transitions, where each transition points to different regions of a composite state. Join is the opposite of fork, where several transitions with origins from different regions are joined together to one single outgoing transition. A join pseudo state is like a *barrier*, it waits until all incoming transitions are made before changing state to the single outgoing transition.

Figure 2.6 illustrates a version of the media player where there are added volume control. The transition triggered by `powerOn` enters a fork which splits the transition into two regions contained in `On`. The media player will initially be in `CDMode` with the volume set to `VolumeLow`. The event `changeMode` will change the mode, but the volume remains the same. However, the event `adjustVolume` turns the volume up and down. At any time the media player can power off.

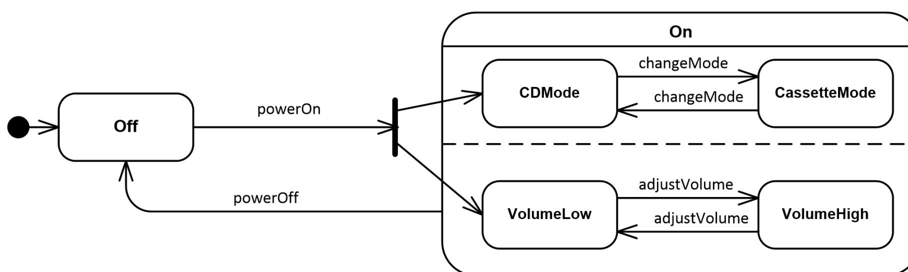


Figure 2.6: Media player with orthogonal state

2.3 State machines in programming

This section will look at how state machines can be represented in programming languages.

2.3.1 State design pattern

In object-oriented programming, the *state design pattern* [4] gives a general solution on how to represent simple state machines, i.e. state machines without advanced mechanisms found in statecharts and UML. [4] describes the intent of the design pattern as:

> *"Allow an object to alter its behavior when its internal state changes. The object will appear to change its class."*

The state design pattern is divided into three parts which makes up the solution.

`Context` The class that serves as the interface for the user, i.e. the state machine interface. Keeps track of the current state, and offer all events of the state machine as methods. The events are implemented such that they will be dispatched to the current state. When the current state changes, the behaviour of `Context` will therefore change.

`State` An abstract class that is the superclass of all states. Defines all events of the state machine as empty methods. States will therefore inherit all events with empty behaviour, and must override the event methods that concerns them.

Concrete states The actual state classes that are subclasses of `State`. Must override the event methods that is inherited from `State` to specify behaviour.

Figure 2.7 illustrates the structure of the general state design pattern solution as described above.

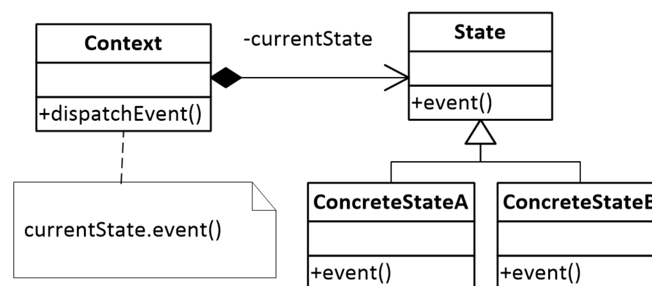


Figure 2.7: The state design pattern

The state design pattern does not specify where state transitions are defined, but typically each state[3] has a reference to the *context object* which offers a method for changing state. This makes it possible to define transitions in the states.

[3]A concrete state will be referred to as a *state*.

Implementation of a switch

For the purpose of illustrating how the state design pattern can be utilized in a practical situation, the on/off switch state machine from figure 2.1 on page 6 is implemented in Java.

Figure 2.8 illustrates the class diagram of the switch. In order for the states to have a reference to the context object, the states are implemented as *inner classes* of the context class. The outer class `SwitchStateMachine` serves as the context class by keeping track of the current state, and offer the method `changeStateTo` that changes the current state. In addition, `SwitchStateMachine` contains the dispatching event methods `powerOn` and `powerOff`. `SwitchState` is an abstract inner class that defines the event methods `powerOn` and `powerOff` with empty method bodies. The concrete states `On` and `Off` are subclasses of `SwitchState`.

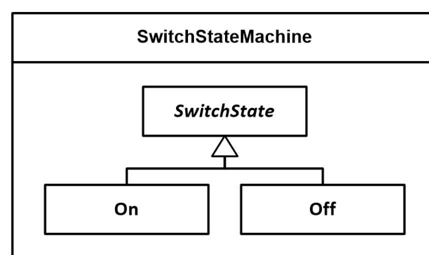


Figure 2.8: On/off switch class diagram

Listing 2.1 shows a possible Java implementation of the on/off switch with the class `SwitchStateMachine` representing the state machine. The events are defined with empty method bodies in `SwitchState`. The classes `On` and `Off` overrides their event method in order to define their transitions. Notice that for each transition a new state instance will be created. In a more advanced implementation, the states should only be instantiated once and reused, see e.g. the *singleton pattern* [4].

Listing 2.1: `SwitchStateMachine`

```
1  class SwitchStateMachine {
2    private SwitchState currentState = new Off();
3    protected SwitchState currentState() {
4      return currentState;
5    }
6    protected void changeStateTo(SwitchState target) {
7      currentState = target;
8    }
9
10   // Dispatching event methods
11   void powerOn() {
12     currentState().powerOn();
13   }
14   void powerOff() {
15     currentState().powerOff();
16   }
17
18   // State implementations
```

```
19    protected abstract class SwitchState {
20      void powerOn() { }
21      void powerOff() { }
22    }
23    protected class On extends SwitchState {
24      void powerOn() {
25        changeStateTo(new Off());
26      }
27    }
28    protected class Off extends SwitchState {
29      void powerOn() {
30        changeStateTo(new On());
31      }
32    }
33  }
```

To use the state machine, the user can create an instance of `SwitchStateMachine`, and use the dispatch event methods, as shown below.

```
SwitchStateMachine context = new SwitchStateMachine();
context.powerOn();
context.powerOff();
...
```

2.3.2 ThingML

The programming language named ThingML[4] is a DSL developed by SINTEF, with the purpose of modeling embedded and distributed systems. Their website describes the idea of ThingML as:

> The idea of ThingML is to develop a practical model-driven software engineering tool-chain which targets resource constrained embedded systems such as low-power sensor and microcontroller based devices.

Integrated into the language are state machines, containing mechanisms found in statecharts and UML state machines. For the purpose of illustrating how state machines may look like in a programming language, this section will explain some of the ThingML syntax and implement one of the examples of section 2.2.

The basics

A state machine is defined with a name and an initial state. The body of a state machine contains one or more states as shown below.

```
statechart MyStateMachine init S1 {
    state S1 {}
    state S2 {}
    ...
}
```

[4] http://thingml.org

Transitions between states are defined inside a state, where each transition may define a possible triggering event, guard and action. The events have a special syntax that looks like `port?eventName`, where `port` is the name of the port that the state is listening to, and `eventName` is the name of the event.

```
state S1 {
    transition t1 -> S2
    event port?eventName
    guard /* Condition */
    action /* Action code */
}
```

A state may also define entry and exit actions.

```
state S1 {
    on entry do /* Action code */ end
    on exit do /* Action code */ end
}
```

States defined as composite states must specify their initial state, and may define substates inside their bodies.

```
composite state S2 init S3 {
    state S3 {}
    state S4 {}
}
```

A composite state may track history. The first time the composite state is entered the initial state is set as current state. History will be saved when the composite state is exited, meaning that the next time the composite state is entered, the last visited state will be entered automatically.

```
composite state S2 init S3 keeps history { ... }
```

Media player example

With the basics of ThingML covered, the code below illustrates the implementation of the media player of figure 2.5 on page 9.

```
statechart MediaPlayer init Off {
    state Off {
        transition t1 -> On
        event port?powerOn
    }
    composite state On init CDMode keeps history {
        transition t2 -> Off
        event port?powerOff
```

```
        state CDMode {
            on entry do /* CD motor on */ end
            on exit do /* CD motor off */ end
            transition t3 -> CassetteMode
            event port?changeMode
        }
        state CDMode {
            on entry do /* Cassette motor on */ end
            on exit do /* Cassette motor off */ end
            transition t4 -> CDMode
            event port?changeMode
        }
    }
}
```

Chapter 3

State machine framework

3.1 Introduction

The state design pattern [4] gives a simple solution to how to make the behaviour of a class dependent on its state. The solution works for implementing simple state machines, but as claimed in [7], the pattern is primitive in its functionality, making it difficult to extend to support *composite states* and other advanced mechanisms found in statecharts [8] and UML state machines [16]. Extending the state design pattern with these mechanisms will be demanding for the user, and will introduce unnecessary complications in the task of making a state machine. The task would be best accomplished by using a framework.

Chapters 4 and 5 will implement such a framework in Java, with the purpose of investigating two different approaches to *state hierarchy* representations; one approach using *inheritance* and the other using *delegation*. The term "state hierarchy" is used to describe the hierarchical structure of composite and simple states of a state machine. For the purpose of the investigation, only a subset of the mechanisms found in UML state machines, which mainly concerns composite states, will be supported. Probably the most common mechanisms for composite states are *entry and exit points*, and *shallow and deep history*. Even though not directly related to composite states, *entry and exit actions* are interesting to investigate as they are slightly more challenging to support when working with composite states than for simple states. The mechanisms will be supported by the framework, along with the other mechanisms already supported by the state design pattern, which are *simple states*, *events* and *transitions*.

Despite the differences with respect to the state hierarchy representations in chapters 4 and 5, the frameworks that are implemented are similar in many ways:

- They have the same state hierarchy structure, but with different representations.
- They use the same algorithms, but with different implementations based on the representations of the state hierarchy structure.

- They have the same framework classes and interfaces, with more or less the same methods. However, there are a few differences that will be pointed out in chapters 4 and 5.

This chapter will explain all the things that applies to both framework implementations. Section 3.2 gives an example state machine that will be used for illustration purpose throughout this and the next chapters. Section 3.3 goes through the state hierarchy structure, *pseudo state* representations and algorithms needed in the framework, all explained from a principal point of view as the frameworks have different state hierarchy representations. Section 3.4 gives an overview of the framework, explaining the different classes and interfaces, and giving an overview of the methods that are common for both frameworks. Section 3.5 gives a brief summary.

3.2 Media player example

For the purpose of illustrating the framework, an example based on the media player from chapter 2 will be used, but with some modifications. Figure 3.1 defines the state machine behavior of a media player. It specifies that a media player will start in the state `Off`. When powered on the first time it will enter the `CDMode`. The mode may be changed by `changeMode`, and the player can be asked to `play` and `stop`. At any state the player may be powered off, entering the state `Off`.

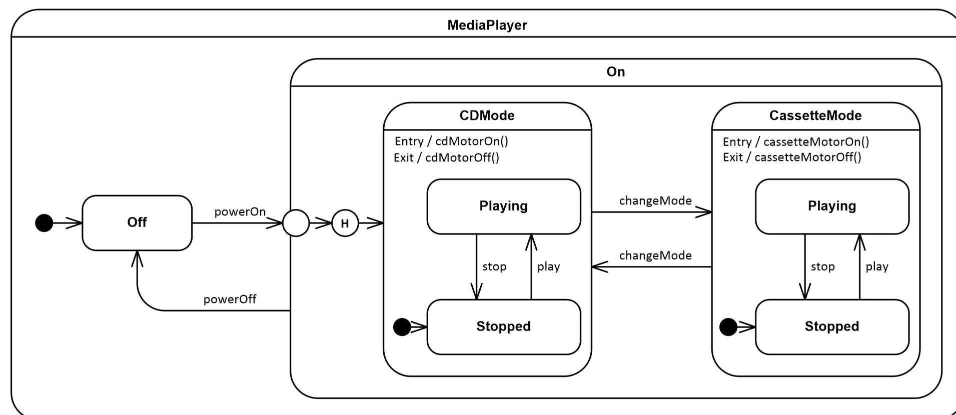


Figure 3.1: UML state machine representing a media player

The state `On` is a composite state, which means that the transition triggered by the event `powerOff` is *compound*, i.e. applies to all contained states at any depth, with the same transition (to `Off`). The events `play` and `stop` also apply to the state `On`, however, these are defined to be different for different contained states.

Whenever the state `CDMode` is entered, the action `cdMotorOn()` is executed. Whenever the state `CDMode` is exited, the action `cdMotorOff()` is executed. The state `CasetteMode` has similar entry/exit actions.

A transition to a shallow history node (state symbol with an **H**) implies a transition to the contained state that was exited last time the composite state was exited. For the media player this means that when powering on, the next state will be the state that was exited when the composite state was exited with a `powerOff`. First time powered on the state will be `CDMode` as specified by the default history transition from **H**.

3.3 State hierarchy structure and algorithms

Even though the state hierarchy representations of chapters 4 and 5 will be different, the state hierarchy structure are the same, and hence the algorithms for implementing the different mechanisms will be the same in principle.

3.3.1 State hierarchy analysis

State hierarchies can have different representations, but in general it will form a tree structure like the one in figure 3.2. The nodes with children represents composite states with any number of substates, and the leaf nodes are simple states. The root node will be referred to as the *root state* (e.g. the root state of the media player is `MediaPlayer`). The edges between the nodes represent the relationship between composite states and their contained states.

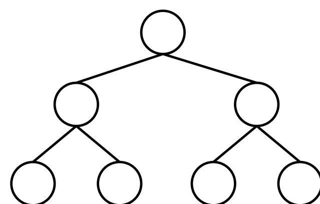


Figure 3.2: State hierarchy represented as a tree structure

When changing state in a state machine, it is always the case of changing state *from* a source state *to* a target state. In order to support mechanisms like history and entry/exit actions, it is necessary to know the location of states, and the relationship between their locations. This is called *state hierarchy analysis*.

Same enclosing state

To check whether states has the *same enclosing state*, the framework must compare the parent node of the states. If the states has the same parent they are said to have the same enclosing state. How to get and compare the parent nodes depends on how the state hierarchy is represented. E.g. if inheritance is the representation of state hierarchy (where states are represented by classes), the enclosing state is the immediate superclass of a state class.

Figure 3.3(a) shows an example where the state *S* has the same enclosing state as the black states (including itself).

Substates

A state is *substate* of another if the other state is an enclosing state.

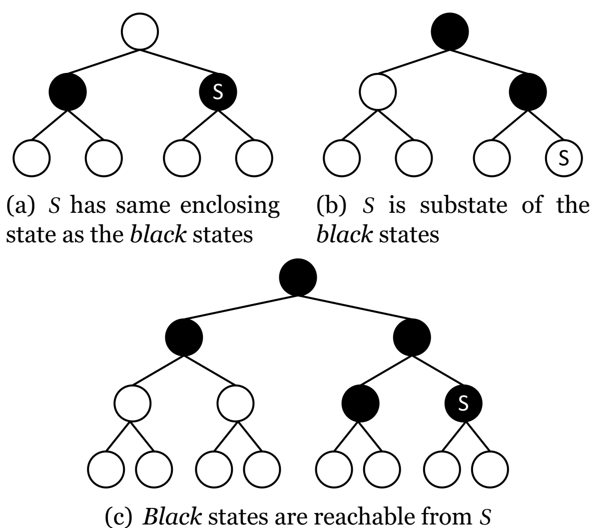
Figure 3.3(b) illustrates a state hierarchy where the state *S* is compared to all states. *S* is substate of the *black* states which are all enclosing states of *S*. Notice that *S* is *not* substate of itself.

Reachable states

A state is *reachable* from another if a transition to it is valid without the use of entry and exit points. A transition is valid if it points to a state that has the same enclosing state. Compound transitions are also legal and points to reachable states, which also means that the root state is reachable as it may define a transition to itself.

Example 3.1. *The event `powerOff` of the media player defines a compound transition for all substates of `On`. No matter the location inside `On`, the event `powerOff` will transition to `Off`. `Off` is therefore reachable from `On` and from any substates of `On`.*

Figure 3.3(c) illustrates a state hierarchy where the black states are reachable from *S*. Notice that a direct transition from *S* to any of the black states will result in *one* entry action. *S* cannot explicitly define a transition to all the black states without using exit points, but the black states may, however, be reached by potential compound transitions that are defined in an enclosing states of *S*.



(a) *S* has same enclosing state as the *black* states

(b) *S* is substate of the *black* states

(c) *Black* states are reachable from *S*

Figure 3.3: State hierarchy analysis

3.3.2 Entry and exit actions

Supporting entry/exit actions in a flat state hierarchy is not as difficult as with state hierarchy consisting of composite states. In a flat state machine a transition will always result in *one* exit action and *one* entry action per transition, while in a state machine with composite states and entry/exit points there may be several states that are exited and entered in one single transition.

Example 3.2. *Figure 3.4 illustrates the state hierarchy that represents the media player. Assume that the current state is `Stopped` contained in `CDMode`, and that the event `powerOff` occurs, resulting in the transition to `Off`. The transition involves executing the following sequence of exit actions (1) `Stopped`, (2) `CDMode` and (3) `On`, followed by the entry action of `Off`. The dashed arrows of the figure is the flow of the transition that shows where and in what order the different actions are executed.*

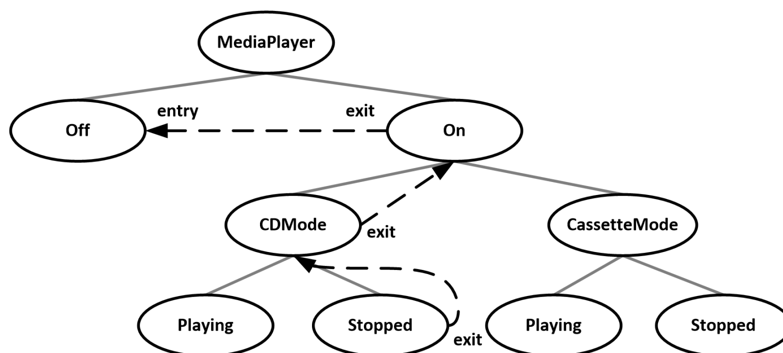


Figure 3.4: Entry/exit actions

Based on a transition from a *source state* to a *target state*, the framework must be able to execute the correct exit actions followed by the correct entry actions.

Algorithm 3.1. *A transition from source state `S` to target state `T` involves the following exit actions.*

1. *If `T` is not a substate of `S` then execute the exit action of `S`.*
2. *If `S` has an enclosing state `E`, and `T` is not a substate of `E`, then start at (1) with `S` set as `E`.*

Example 3.3. *This example will trace how algorithm 3.1 works by running example 3.2.*

`[S=Stopped, T=Off]` *According to (1) `Off` is not a substate of `Stopped` resulting in execution of the exit action of `Stopped`. `Stopped` has the enclosing state `CDMode`, and `Off` is not a substate of `CDMode` resulting in (2).*

[S=CDMode, T=Off] *According to (1) `Off` is not a substate of `CDMode` which executes the exit action of `CDMode`. `CDMode` has the enclosing state `On`, and `Off` is not a substate of `On` resulting in (2).*

[S=On, T=Off] *According to (1) `Off` is not a substate of `On` which results in executing the exit action of `On`. `On` has the enclosing state `MediaPlayer`, and `Off` is a substate of `MediaPlayer` which means that the algorithm is finished.*

Even though a transition to a reachable state will result in exactly one entry action, the introduction of entry/exit points will make it possible to enter any other state which will result in potentially several entry actions. The algorithm for executing entry actions must therefore support transitions between all states.

Algorithm 3.2. *A transition from source state `S` to a target state `T` involves the entry actions listed below. To ensure the correct ordered sequence of entry actions, a stack will be used to reverse the order of the algorithm.*

1. *If `T` is reachable from `S` then execute the entry action of `T`. Execute all entry actions of the stack if any. The algorithm is finished.*
2. *Else, if `T` is a substate of `S`, push the entry action of `T` on the stack. If the enclosing state of `T` is `S`, then execute all entry actions on the stack, else start at (1) with `T` set as the enclosing state of `T`.*
3. *Else, start at (1) with `S` set as the enclosing state of `S`.*

Example 3.4. *In this example the media player is about to be powered on for the first time. With the current state `Off` and the incoming event `powerOn` there will be a direct transition to `CDMode` via the entry point of `On` (for this example the initial state of `CDMode` is ignored). Algorithm 3.2 will produce the following trace.*

[S=Off, T=CDMode] *According to (3) start at (1) with `S` set as the enclosing state of S which is `MediaPlayer`.*

[S=MediaPlayer, T=CDMode] *According to (2) `CDMode` is a substate of `MediaPlayer`. Push the entry action of `CDMode` on the stack and start at (1) with `T` set to `On`.*

[S=MediaPlayer, T=On] *According to (2) push the entry action of `On` onto the stack. The enclosing state of `T` is `MediaPlayer` which means that the entry actions on the stack will be executed in the order (1) `On` (2) `CDMode`, and the algorithm is finished.*

3.3.3 History

In a state machine, each composite state has two pairs, one for shallow and one for deep history. A pair can be written as `(C,H)` where `C` represents the composite state and `H` is the shallow or deep history state of `C`. Every time the state machine is changing state, the state machine will update these pairs according to the transition.

Shallow history

Shallow history will be saved every time a composite state is exited. Based on the algorithms presented in section 3.3.2, this functionality is implemented by making sure that every time the framework executes the exit action of a composite state, the last visited state contained in the composite state will be saved as shallow history.

Example 3.5. *Figure 3.5 exemplifies the transition from `Stopped` contained in `CDMode`, to the state `Off`. The transition is possible if the current state is `Stopped` and the incoming event is `powerOff`. The dashed arrows is the flow of the transition. The pairs represents the shallow history that is saved, and the location of the pair represents at which time it is saved. The transition will cause the framework to execute the exit actions of the composite states (1) `CDMode` and (2) `On`, and hence also save the shallow history at the same locations.*

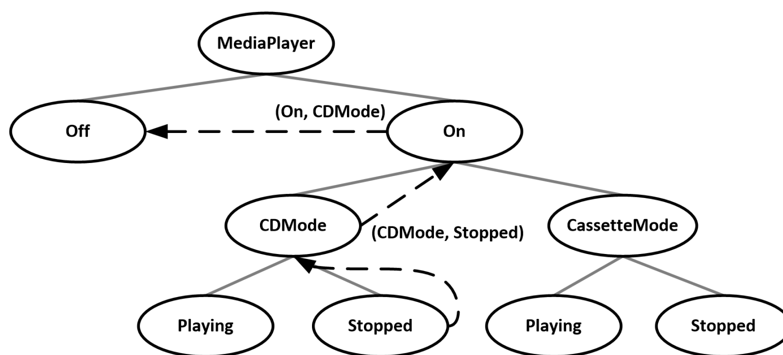


Figure 3.5: Saving shallow history

Deep history

Deep history is saved for a composite state when the composite state is exited. If a composite state is exited, the framework will check if the current state of the state machine is a substate, but not an immediate substate, of the composite state.

Example 3.6. *Figure 3.6 illustrates the same situation as in figure 3.5, where there is a transition from `Stopped` contained in `CDMode` to `Off`. The current state of the state machine will be `Stopped` until all entry and exit actions are finished. First the transition exit `Stopped`. The current state is*

not substate of `Stopped`, so no history is saved. Then `CDMode` is exited, and the current state is substate, but is an immediate substate, so no history is saved. Lastly the state `On` is exited, and the current state is a substate and not an immediate substate, which means that the current state will be saved as deep history for `On`.

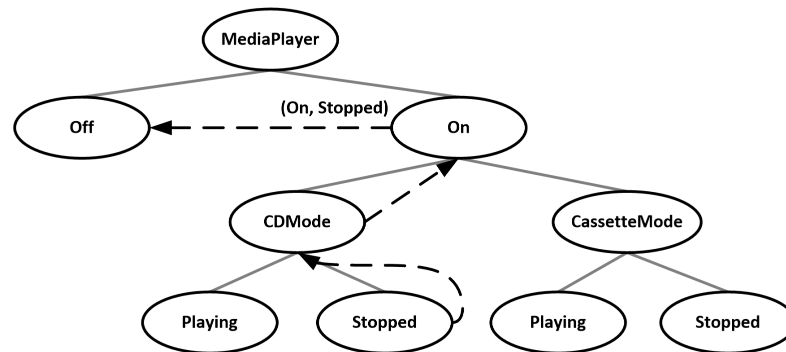


Figure 3.6: Saving deep history

3.3.4 Pseudo states

This section will introduce the pseudo state representations of the framework.

Entry and exit points

An entry/exit point is represented as a pair (C_{point}, T_{point}) where C_{point} is the composite state the entry/exit point is connected to, and T_{point} is the target state that the state machine will make a transition to.

Shallow and deep history

Shallow and deep history are represented as the pair $(C_{history}, T_{history})$ where $C_{history}$ is the enclosing composite state of the history state, and $T_{history}$ is the default history target state. The state machine will use $C_{history}$ to look up for shallow or deep history. If nothing is found, the state machine will automatically transition to $T_{history}$.

Validation criteria

To ensure valid UML pseudo states (i.e. the validity of their outgoing transitions), there are some validation criteria for each of the framework classes when it comes to valid input values. Breaking the validity will cause an error in the state machine.

Entry points – The enclosing state of T_{point} must be the same state as C_{point}.

Exit points – T_{point} and C_{point} must have the same enclosing state.

Shallow and deep history – The enclosing state of $T_{history}$ must be the same state as $C_{history}$.

3.3.5 State transitions

A state machine must be able to make transitions between states. With composite state, pseudo states and entry/exit actions, changing state is to some extent a complex process.

Figure 3.7 shows the flow chart of how the process of changing state is executed in a state machine. Without going to much into details, the process is divided into two main parts; the first part deals with transitions between states, and the second part deals with transitions to pseudo states.

Changing state to a target that is not a pseudo state involves executing the exit actions from S to T according to algorithm 3.1, followed by executing the entry actions according to algorithm 3.2. T will then be set as the current state of the state machine. If T has an initial state, then change state to the initial state of T.

If the target state is a pseudo state, all pseudo states that are chained together (if more than one pseudo state) will be iterated until a state is reached (e.g. entry points may be linked to other entry points or history states). Entry/exit points are iterated by updating T to T_{point}. History is iterated by updating T to the shallow/deep history of $C_{history}$, or if the history is empty the value of $T_{history}$ is used instead. Both iterations will check if the new updated T is a pseudo state, and if so loop back. When T becomes a state, the process of changing state between two states will be executed as described previously. Notice that a pseudo state is never set as the current state.

3.4 Framework overview

Having the theory around the framework covered, this section will define the framework classes, and integrate them with the state design pattern. The implementation details that apply to both frameworks will be covered, e.g. implementation of pseudo states and history. The other implementation details will be covered in chapters 4 and 5.

Employing the state design pattern to the media player is difficult as there is no support for composite states. Figure 3.8 illustrates how far one will get with the state design pattern trying to implement the media player, resulting in a simple on/off switch.

To add support for composite states, the user would have to implement it on top of the pattern resulting in a demanding and time consuming task as there are many details to consider. First of all the user must decide how to represent the state hierarchy, and based on this choice implement the rest of the mechanisms.

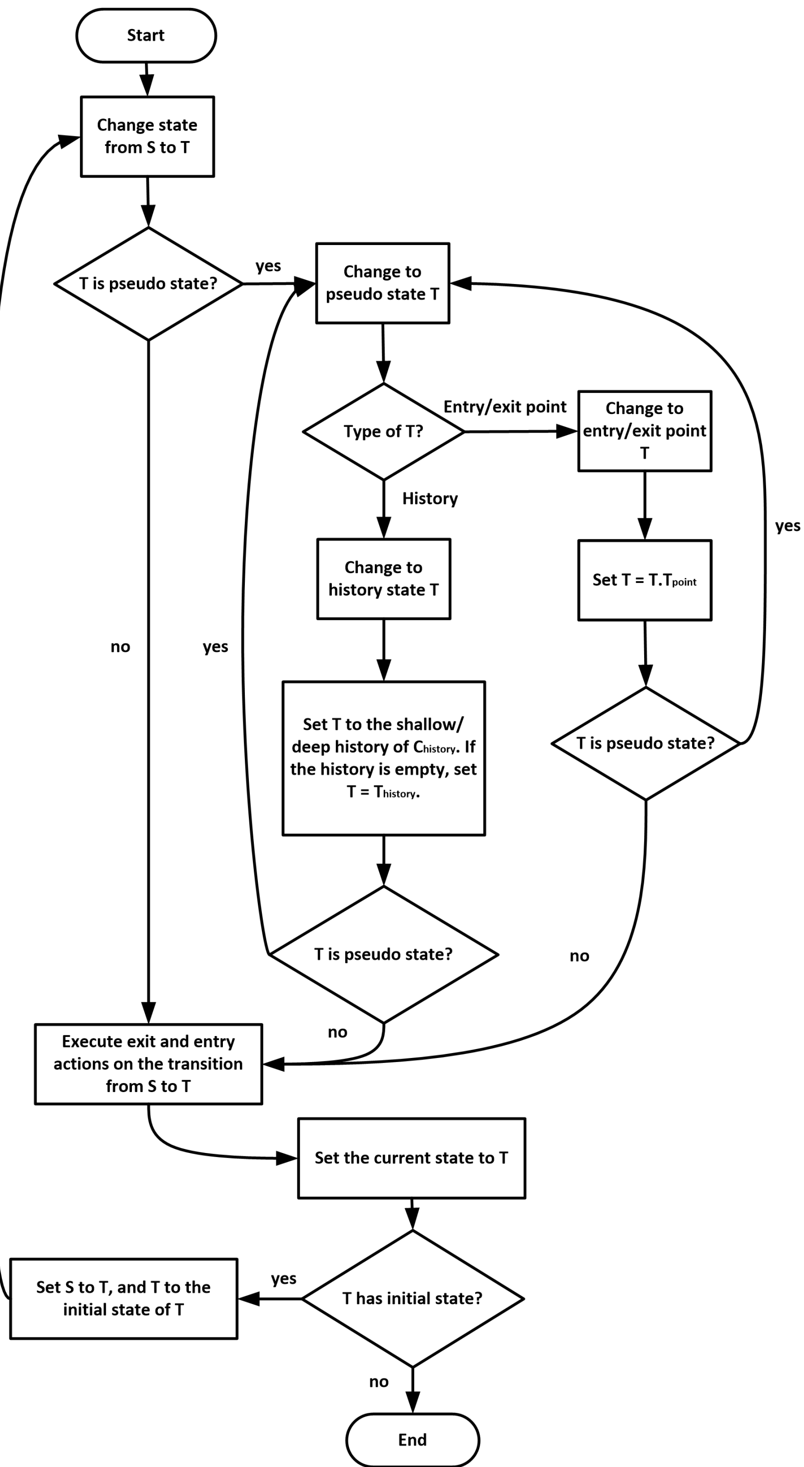


Figure 3.7: The process of changing state

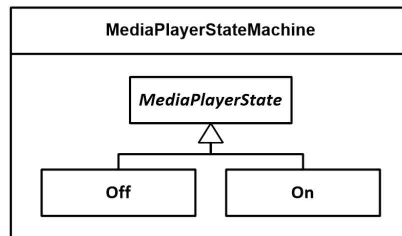


Figure 3.8: State design pattern applied to the media player

3.4.1 A basic framework

The basic idea of a framework is to move the functions which are valid for all state machines into the classes `StateMachine` and `State`, see figure 3.9. Users of the framework will then make a subclass of `StateMachine` for creating state machines, and subclasses of `State` to create user defined states with event methods. All state classes will be inner classes of the extended `StateMachine`, which is illustrated as `MediaPlayerStateMachine`. The *dashed* lines in the figure indicate classes that are inherited from a superclass, in this case from `StateMachine`.

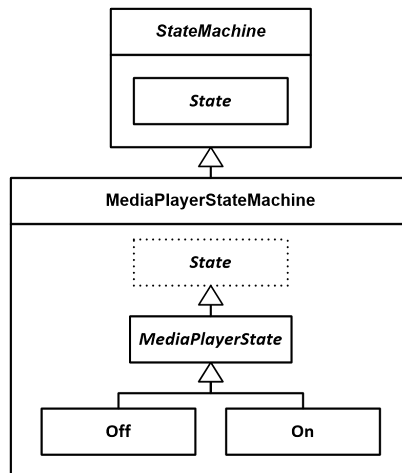


Figure 3.9: Use of the state machine framework

`StateMachine` will contain functionality for doing state hierarchy analysis, handling state transitions, keeping history, etc. `State` will contain functionality that is specific for a state, for instance methods that defines entry/exit actions and the initial state. `State` will also have functionality such that the user is able to define state transitions.

The classes `StateMachine` and `State` will be defined according to the state hierarchy representation. The state hierarchy can be represented by e.g. inheritance where contained states inherit their enclosing state, or with delegation where there is a object reference to the enclosing state, see chapters 4 and 5.

3.4.2 Making the framework generic

In order to create a state machine with the framework from figure 3.9, one would have to create a state machine class which is a subclass of `StateMachine`, and state classes which are subclasses of `State`.

```
class MediaPlayerStateMachine extends StateMachine {
    protected abstract class MediaPlayerState extends State { ... }
    ...
}
```

The class `State` does not include any event methods as they are defined in subclasses, like in `MediaPlayerState` above. This is a problem because the framework only knows about the type `State`, but not the user defined states, which means that states returned from framework methods will not give the user access to the event methods unless the user do unsafe *type casting*.

The solution is to make the class `StateMachine` *generic*[1] with *one* type parameter that represents the type of the states in the state machine. The type parameter must contain all methods of `State`, in addition to the event methods. The code below defines the type parameter `StateType` which is required to extend `State`, and may define event methods as well.

```
abstract class StateMachine<StateType extends State> { ... }
```

However, it is not possible to use `State` because it is an inner class of `StateMachine`, and will be *protected* such that no one from the outside can see it. All the methods of `State` will therefore be defined in the interface `IState` which is implemented by `State`, see figure 3.10. The generic type parameter `StateType` is therefore required to implement `IState` instead as shown below.

```
abstract class StateMachine<StateType extends IState> {
    protected class State implements IState { ... }
    ...
}
```

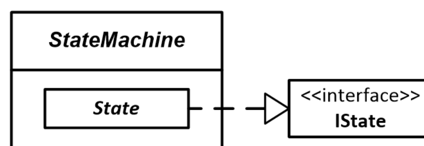


Figure 3.10: Framework extended with `IState`

The user can now create a state machine by creating an interface that extends `IState` with event methods, and create a state machine class that is a subclass of `StateMachine`. The extended interface will be put as argument to the generic type parameter of `StateMachine` such that the states of the state machine are required to implement the interface.

[1] http://docs.oracle.com/javase/tutorial/extra/generics/index.html

```
interface IMedia extends IState {
    void powerOn();
    void powerOff();
    ...
}
class MediaPlayerStateMachine extends StateMachine<IMedia> {
    protected abstract class MediaPlayerState extends State
      implements IMedia {
        public void powerOn() { ... }
        public void powerOff() { ... }
        ...
    }
    ...
}
```

3.4.3 State references

In the Java runtime environment there is a singleton object for each class in the program with the type `java.lang.Class`. A reference to such objects can be obtained at compile time by writing `ClassName.class`.

In order to have a consistent way of referencing states and avoid that the user instantiate states manually, the framework use *state references* that are `Class`-objects of the state classes. The user will reference a state by `StateName.class`, and the framework will use the state reference to reuse the state instance that is associated with the reference (if previously instantiated), or using *automatic state instantiation* in order to instantiate the state automatically, see section 3.4.4.

3.4.4 `StateMachine`

`StateMachine` is the class that defines the functionality that is common for all state machines. The class contains functionality for changing the current state, doing state hierarchy analysis and saving history. Most of the functionality is hidden from the user, such that the design process of a state machine is as simple as possible.

User methods

The methods that are available for the user, i.e. declared as `protected`, are called *user methods*. `StateMachine` defines a very simple interface to the user in table 3.1.

Framework methods

The methods that are only visible in the framework classes are called *framework methods*.[2] These methods are used only by the framework such

[2]The framework classes are inner classes of `StateMachine`, such that methods declared `private` in `StateMachine` are visible.

`protected final StateType currentState()` Returns the current state of the state machine. Notice that the current state can never be a pseudo state.
`protected Class<? extends StateType> initialState()` Defines the initial state of the state machine. Returns `null` by default, but may be overridden in order to specify otherwise.

Table 3.1: Methods of `StateMachine` that are visible for the user

that the user can focus on the state machine design without cluttering the design with implementation details of the different mechanisms. Table 3.2 gives an overview of the most important framework methods.

Notice that the methods `changeToState` and `changeToPseudoState` are only visible for framework classes located inside `StateMachine`, and not visible for the user in state machine classes that are subclasses of `StateMachine`. The user methods for defining trantransitions will be located in `State`.

Automatic state instantiation

States are referenced through objects of type `java.lang.Class`, referred to as state references. In order to reuse already instantiated states, `StateMachine` will instantiate states automatically and keep the instances in a hash map such that they can be reused. The implementations of automatic state instantiation are slightly different in the different frameworks implementations. More details on how state references are automatically instantiated will therefore be covered in chapters 4 and 5.

History

How and when history is saved is explained in section 3.3.3. In `StateMachine` there are two hash maps, one for shallow and one for deep history. The *key* for both hash maps is the state reference to the composite states with history, and the *value* associated with the key is a reference to the state that is saved as history. The framework saves history automatically in the method `changeToState`, and fetches history using the methods `getShallowHistory` and `getDeepHistory`, all methods defined in table 3.2.

3.4.5 Pseudo states in the framework

In similarity with states, pseudo states are user defined classes, but are subclasses of either `EntryPoint`, `ExitPoint` `ShallowHistory` or `DeepHistory`, see figure 3.11. Since pseudo states do not support entry/exit actions nor have an initial state, a new interface `IVertex` is created such that the functionality that is common for states and pseudo states is defined in a separate interface.

`private void changeToState(Class<? extends StateType> target)` Changes state from the current state to the state referenced by `target`. If `target` has no corresponding instance it will be instantiated automatically.
`private void changeToPseudoState(Class<? extends PseudoState> target)` Changes state from the current state to the pseudo state referenced by `target`. If `target` has no corresponding instance it will be instantiated automatically. The current state of a state machine is never a pseudo state, meaning that the method always results in setting the current state to a state. See more details in section 3.3.5.
`private StateType getShallowHistory(Class<? extends StateType> state)` Returns the shallow history of the state referenced by `state`. If no history exist, `null` is returned.
`private StateType getDeepHistory(Class<? extends StateType> state)` Returns the deep history of the state referenced by `state`. If no history exist, `null` is returned.
`private boolean isRoot(Class<? extends StateType> state)` Checks whether the state referenced by `state` is the root state. See section 3.3.1 for details.
`private boolean hasSameEnclosingState(Class<? extends IVertex> stateA, Class<? extends IVertex> stateB)` Checks whether `stateA` and `stateB` has the same enclosing state. See section 3.3.1 for details.
`private boolean isSubstateOf(Class<? extends StateType> fromState, Class<? extends StateType> toState)` Checks whether `fromState` is substate of `toState`. See section 3.3.1 for details.
`private boolean isReachable(Class<? extends IVertex> fromState, Class<? extends IVertex> toState)` Checks whether `toState` is reachable from `fromState`. See section 3.3.1 for details.

Table 3.2: Methods of `StateMachine` that are visible for the framework

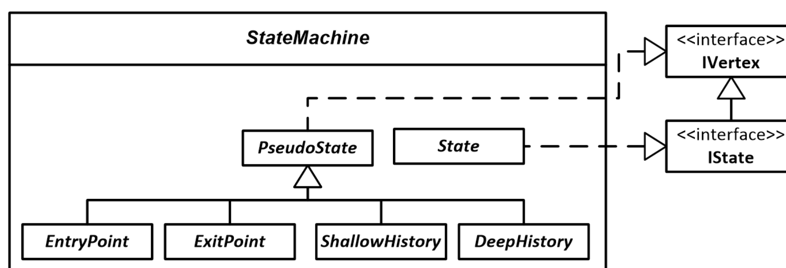


Figure 3.11: Framework with pseudo states

3.4.6 `IVertex`

`IVertex` is an interface that defines the methods that are common for states and pseudo states. Table 3.3 gives a method summary.

public IVertex enclosingState() Returns the enclosing state of this state/pseudo state.
public void validate() Validates this state/pseudo state.

Table 3.3: Methods of the interface IVertex

3.4.7 Pseudo states

There are four different pseudo state classes in the framework, i.e. EntryPoint, ExitPoint, ShallowHistory and DeepHistory. These classes are subclasses of PseudoState. Because all pseudo states, including those not supported by the framework, are different, the class PseudoState does not define any functionality that is common for all pseudo states. Even the methods defined by the interface IVertex is implemented differently in all pseudo states.

Entry and exit points

Table 3.4 lists the methods that is implemented by EntryPoint and ExitPoint, in addition to the methods of IVertex. The methods are by default defined as empty, and must be overridden in user defined subclasses to define specific behaviour. An entry/exit point has only one outgoing transition, either to a state or to a pseudo state. Therefore, only one of the methods targetState and targetPseudoState may return a state reference.

public Class<? extends IState> compositeState() Returns the state reference to the composite state that this entry/exit point is connected to.
public Class<? extends IState> targetState() Returns the state reference to the target state of this entry/exit point.
public Class<? extends IVertex> targetPseudoState() Returns the state reference to the target pseudo state of this entry/exit point.
public void action() Defines the actions performed by the outgoing transition of the entry/exit point.

Table 3.4: Methods of EntryPoint and ExitPoint

Shallow and deep history

Table 3.5 gives a method summary of ShallowHistory and DeepHistory. All the methods are by default empty, and must be overridden by user defined subclasses in order to define behaviour.

`public Class<? extends IState> compositeState()` Returns the state reference to the composite state that this history pseudo state is enclosed by.
`public Class<? extends IState> defaultHistoryState()` Returns the state reference to the default history state of this history pseudo state.
`public Class<? extends IVertex> defaultHistoryPseudoState()` Returns the state reference to the default history pseudo state of this history pseudo state.
`public void action()` Defines the actions performed by the outgoing transition (default history transition) of this history pseudo state.

Table 3.5: Methods of `ShallowHistory` and `DeepHistory`

3.4.8 `IState`

`IState` is a subinterface of `IVertex`, and defines all the methods that `State` must implement. Table 3.6 gives a method summary.

`public void entry()` Defines the entry action of this state.
`public void exit()` Defines the exit action of this state.
`public Class<? extends IState> initialState()` Defines the initial state of this composite state if the initial state is a *state*.
`public Class<? extends IVertex> initialPseudoState()` Defines the initial state of this composite state if the initial state is a *pseudo state*.

Table 3.6: Methods of the interface `IState`

3.4.9 `State`

`State` is an inner class of `StateMachine`, and implements the interface `IState`. The actual implementation of `State` will be different for both frameworks, however, entry/exit actions and the definition of the initial state will be the same.

Entry and exit actions

The methods `entry` and `exit` of table 3.6 defines the entry and exit actions of a state. These methods are by default defined with empty bodies in `State`, and must be overridden in user defined states to specify any behaviour.

`StateMachine` will execute the entry/exit actions automatically when making transitions.

Initial states

The initial state of a composite state is defined by overriding the method `initialState` defined in table 3.6. When changing the current state of the state machine, `StateMachine` will check if the new current state has an initial state, and if so, change to this automatically.

In UML, initial states are defined by initial pseudo states. To simplify the implementation process of state machines, the initial pseudo state is not part of the framework.

Making `State` generic by type overloading

A problem with `IVertex` and `IState`, which are implemented by `State`, is that both interfaces are defined outside of `StateMachine`, and will therefore not get access to the generic type parameter `StateType` which should be used by `State`. However, in Java it is possible to do *type overloading*. E.g. `State` overloads its return types as shown below.

```
abstract class StateMachine<StateType extends IState> {
    ...
    protected abstract class State implements IState {
        public StateType enclosingState() { ... }
        public Class<? extends StateType> initialState() { ... }
        public Class<? extends PseudoState> initialPseudoState() { ... }
        ...
    }
    protected abstract class PseudoState implements IVertex { ... }
}
```

The code above is valid because `StateType` is a subtype of `IState`, and `PseudoState` is a subtype of `IVertex` (compare return types of `State` with tables 3.3 and 3.6).

3.4.10 Events and transitions

Events and transitions are based on the notion of method signatures representing events, and method bodies containing actions with corresponding transitions. This approach is adopted from the state design pattern and is also proposed in other research papers [**?**, 12].

Figure 3.12 illustrates the approach where e is the event and a is the action that will be triggered by the event. The method body of `e()` executes `a` which is code that may change internal and external variables, followed by changing state to the target state `T`. It is important that the action code is defined to be executed before the transition such that the order of execution is correct.

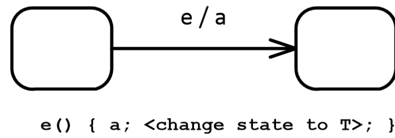


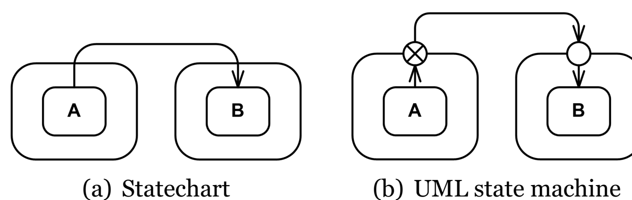
Figure 3.12: Event with corresponding transition represented as a method

Like with the state design pattern, the user must implement the event methods in the user state classes, and the state machine class must implement all event method signatures and will dispatch the event calls to its current state.

```
class MediaPlayerStateMachine extends StateMachine<IMedia> {
    void powerOn() { currentState().powerOn(); }
    void powerOff() { currentState().powerOff(); }
    ...
    protected class Off ... {
        void powerOn() { <Change to state On>; }
        ...
    }
}
```

Valid transitions

Transitions in Harel statecharts may point to any state in the state machine, even if contained in another composite state. UML is more restricted in a sense that a transition must point to a state that is contained by the same enclosing state.[3] To accomplish similar transitions as with statecharts, UML use entry and exit points, see figure 3.13.



(a) Statechart (b) UML state machine

Figure 3.13: Statechart vs. UML state machine transitions

The framework will restrict the user to only be able to make transitions as specified in UML. This is done by validating all transitions such that if there are any illegal ones they will result in an error. E.g. a direct transition from `A` to `B` like in figure 3.13(a) is illegal because the states have different enclosing states. Validation will be discussed in chapters 4 and 5.

[3] In reality, UML transitions must point to a state in the same *region*, but regions is not supported by the framework.

3.5 Summary

With just a subset of the functionality supported by UML state machines, there are already a lot of functionality to implement into the state design pattern that would ruin its simplicity. By moving the complexity into framework classes, the user can focus on the actual state machine design and utilize the more advanced functionality by extending these classes.

Chapter 4

State hierarchy by inheritance

4.1 Introduction

In order to extend the state design pattern with UML state machine functionality, a design choice must be taken concerning state hierarchy representation. A promising approach proposed by [12] is to represent state hierarchy by *inheritance*, where contained states are represented by subclasses of the state class that represents the enclosing state. Since the state design pattern already support events with corresponding transitions through virtual methods, it seems obvious that the approach will work, because contained states will inherit methods from their enclosing states. Virtual methods will then function as events with compound transitions because transitions defined by composite states are inherited by their substates. However, the combination with other advanced mechanisms like entry/exit actions will create problems when using inheritance; it is not only the events that will be inherited, but also everything else.

To further investigate inheritance and to pin point its advantages and disadvantages, the framework from chapter 3 will be implemented in Java. Section 4.2 gives an overview of the framework. Section 4.3 shows the most important implementation details. Section 4.4 gives a tutorial on how the framework is used, revealing the many shortcomings of inheritance. Section 4.5 gives a summary of the investigation.

4.2 Overview

While chapter 3 gives a structure of how to integrate a state machine framework on top of the state design pattern, there are just indications concerning how *state hierarchy* is represented. The basic idea of state hierarchies is that regardless of their representation, the structure must form a *tree*, where states (the nodes) may have zero or more substates, and where there is exactly *one* root state. The Java class hierarchy gives such a structure as a consequence of *single inheritance*. The idea is then to

extend the rules of the framework integration by allowing contained states to *extend* their enclosing state. State classes without subclasses will serve as simple states, while state classes with subclasses serve as composite states.

In order to present the inheritance framework, the media player state machine from section 3.2 on page 16 will be used. Figure 4.1 illustrates how the media player will look like from a class hierarchy point of view. The framework class `StateMachine` is extended by `MediaPlayerStateMachine` in order to integrate the user defined state machine with the framework. The *dashed* lines represents inner classes that are defined in a superclass, in this case from `StateMachine`.

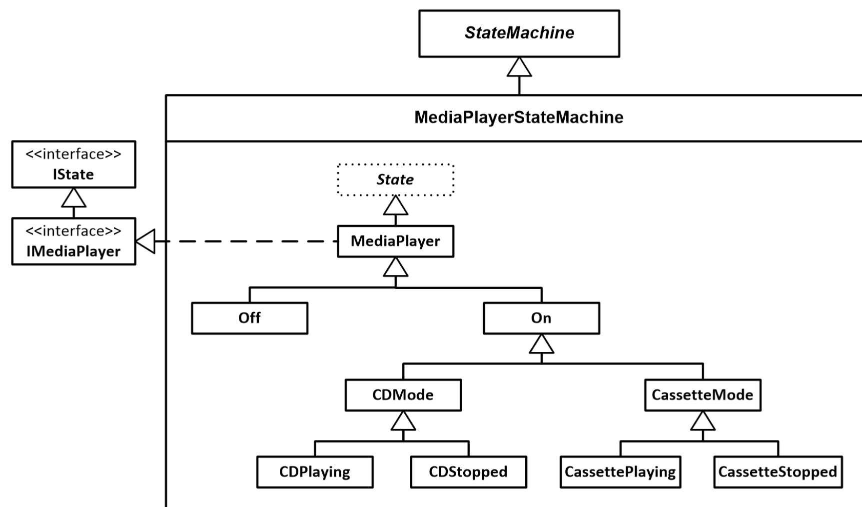


Figure 4.1: Class hierarchy of the media player using inheritance

The user will define all the states as inner classes of the state machine class named `MediaPlayerStateMachine`, where each state class for a contained state is a subclass of its enclosing state. The state machine class is responsible for implementing all events of the state machine such that incoming events gets dispatched to the current state, in addition to defining the initial state of the state machine which in this case should be `MediaPlayer`.

Recall that the class `StateMachine` is generic, and has a type parameter that is required to be a subtype of the interface `IState`, see section 3.4.8 on page 31. The generic type parameter of `StateMachine` represents the type of the user defined states in the state machine. The interface `IMediaPlayer` is therefore created in order to define this type. `IMediaPlayer` extends `IState` and defines all the event methods of the media player, i.e. `powerOn`, `powerOff`, etc.

`MediaPlayer` will serve as a root state because it has no user defined state class as superclass, and will implement `IMediaPlayer` such that the methods required by `IState` and the event methods of `IMediaPlayer` is implemented. To satisfy the `IState` interface, `MediaPlayer` extends the framework class `State`, which is a framework class that will be implemented in this chapter. In addition, all the event methods of `MediaPlayer` will be implemented as

empty because the root state has no outgoing transitions. This means that all subclasses of `MediaPlayer` will have empty behaviour in all events, unless specified otherwise.

By using inheritance as a means of state hierarchy representation, the contained states, i.e. `On`, `Off`, `CDMode`, etc., will inherit event methods from enclosing states. Overriding an event method in a given state will define behaviour that is specific for that state. However, substates of the given state will also have the same behaviour for the same event method ,unless they override the event method, which is very similar to how composite states works. E.g. the example below illustrates the situation, where `On` specifies a transition to `Off` in the event `powerOff`. `CDMode`, which is a substate and hence a subclass of `On`, will inherit the event methods from `On`. If the current state of the state machine is `CDMode` and the incoming event is `powerOff`, the state machine will transition to `Off` as defined in `On`, which is the wanted behaviour.

4.2.1 Framework classes

Table 4.1 lists the classes of the inheritance framework with corresponding constructors. All classes are declared `abstract` such that they cannot be instantiated, unless they are made *concrete* by subclasses.

4.3 Framework implementation

Chapter 3 has gone through most of the framework implementation, this section will fill in the “holes” by explaining how the framework can utilize inheritance as a means of representing state hierarchy, and by this implement the remaining functionality.

4.3.1 State hierarchy traversal

In order to implement state hierarchy analysis, see section 3.3.1 on page 17, entry/exit actions etc., it is important to know how to traverse the state hierarchy. The state hierarchy is represented by the class inheritance hierarchy of state classes. The class inheritance hierarchy in Java can be traversed by using the method `getSuperclass()` defined in `java.lang.Class`. Getting the class of an object can be done by using the method `getClass()` defined in `java.lang.Object`. In order to get the enclosing state of a state class, it is possible to write `getClass().getSuperclass()`.

4.3.2 `StateMachine`

Most of the details about `StateMachine` are explained in section 3.4.4. This section will cover the remaining details about automatic state instantiation, and implementation details of state hierarchy analysis.

Class name	Constructor(s)/description
StateMachine <StateType extends IState>	State() State(Class<? extends StateType> initial) A generic class that represents a general state machine with no events. The type parameter StateType represents the type of the user defined states. There are two constructors available: The parameterless constructor requires that the initial state is defined in the method initialState of StateMachine. As an alternative constructor, the user can set the initial state through the parameter initial.
State	State() An inner class of StateMachine, and superclass of all user defined states. Implements the interface IState, see section 3.4.8. Defines its enclosing state by extending it.
PseudoState	An inner class of StateMachine, and superclass of all pseudo states, i.e. EntryPoint, ExitPoint, ShallowHistory and DeepHistory. Implements the interface IVertex, and also defines functionality that is common for all pseudo state types the inheritance framework.
EntryPoint	EntryPoint() The superclass of psuedo states representing entry points.
ExitPoint	ExitPoint() The superclass of pseudo states representing exit points.
ShallowHistory	ShallowHistory() The superclass of pseudo states representing shallow history.
DeepHistory	DeepHistory() The superclass of pseudo state representing deep history.

Table 4.1: Overview of the inheritance framework classes and constructors

Automatic state instantiation

In order to avoid instantiating states more than once, the framework support automatic state instantiation. This basically means that the user is operating with *state references* which are represented by objects of type java.lang.Class, and StateMachine keeps two internal hash maps, one for state instances and one for pseudo state instances. An instance may then

be fetched by looking it up using the corresponding state reference. If there is no instance of the state reference, `StateMachine` makes sure to instantiate it automatically by using *Java reflection* and putting it into one of the hash maps.

A state reference can make an instance by using the method `newInstance()` defined in `java.lang.Class`. For this to work, the framework will assume that all states has a parameterless constructor. If this is not the case, the automatic state instantiation will fail, throwing the exception `java.lang.InstantiationException`.

Implementation of state hierarchy analysis

The implementation details of the state hierarchy analysis in `StateMachine` is dependent on the state hierarchy representation. Below is a short description of all the methods that do the analysis by using the class inheritance hierarchy of state classes.

`isRoot(state)`: If the immediate superclass of `state` is the class of `State`, the state is considered to be the root state.

`hasSameEnclosingState(stateA, stateB)`: If the immediate superclass of `stateA` and `stateB` are the same, the states has the same enclosing state.

`isSubstateOf(fromState, toState)`: If `toState` is a superclass of `fromState`, then `fromState` is a substate of `toState`. In order to find this out, the method will traverse all the superclasses of `fromState` until it reaches the root state of the class inheritance hierarchy. If one of the classes along the way is the class `toState` then the method will return `true`, else `false`.

`isReachable(fromState, toState)`: The method traverse all superclasses of `fromState` until the root state class is reached. If `toState` is an immediate subclass of any of the superclasses, or `toState` is the root state class, then the method will return `true`, else `false`.

4.3.3 `State`

In addition to the functionality presented in section 3.4.9, `State` is responsible of offering methods to the user for defining transitions. These methods are different compared to those in the delegation framework presented in chapter 5.

User methods

The purpose of the user methods is to define transitions in event methods, see table 4.2. The methods are declared `protected` such that it is only

Method	Description
`protected final void changeToState(Class<? extends StateType> source, Class<? extends StateType> target)`	Makes the state machine, that this state is instantiated by, to change from the current state to the state referenced by `target`. In order to validate the transition, the parameter `source` is needed and specifies the state in which the transition is defined. The states `source` and `target` must have the same enclosing state for the transition to be valid.
`protected final void changeToPseudoState( Class<? extends StateType> source, Class<? extends PseudoState> target)`	The same as `changeToState`, but `target` is a state reference to a pseudo state.

Table 4.2: User methods of `State`

possible to define transitions inside event methods that is defined in state classes.

The user methods first validates the transition, and then use the methods of `StateMachine` (see table 3.2 on page 29) to make the state machine change state.

State transitions

States may transition to other states by using the methods `changeToState` and `changeToPseudoState` defined in table 4.2. The methods takes two parameters, where `source` specifies which state that defines the transition, and `target` specifies the target state. The source state and the target state must have the same enclosing state for the transition to be valid.

An intuitive way of defining the source state is to use the method `getClass()` as illustrated below.

```
...
protected class On extends MediaPlayer {
    void powerOff() { changeToState(getClass(), Off.class); }
    ...
}
...
```

The wanted value of the source state above is `On`. However, the problem is that `getClass` returns different values for contained states. The consequence is that contained states will fail the validation of the transition defined for `powerOff`. To solve the problem, the state reference of the source state must be “hard coded” at the location where the transition is defined.

```
...
protected class On extends MediaPlayer {
    void powerOff() { changeToState(On.class, Off.class); }
    ...
```

```
}
...
```

4.4 Framework usage

As a means of illustrating the usage of the inheritance framework, the media player will be implemented.

4.4.1 Design principles

Implementing a state machine using the inheritance framework involves the following steps:

1. Define an interface that extends `IState` and adds the event method signatures of the state machine.
2. Define a state machine class that is a subclass of `StateMachine`. The generic parameter of `StateMachine` is set to the user defined interface, such that the states of the state machine will be required to implement all the events.
3. Define the event dispatch methods and the initial state in the state machine class. The initial state is defined by overriding the method `initialState` of `StateMachine`, or by using the constructor of `StateMachine` that takes the initial state as argument, see constructors in table 4.1.
4. Define the root state as an inner class of the state machine class, and as subclass of `State`. The root state must implement the user defined interface. All states of a state machine are subclasses of the root state.
5. Define each state as subclass of their enclosing state. Each state must override the events that are specific for them. Entry and exit actions can be defined by overriding the methods `entry` and `exit` inherited from `State`. If a state is a composite state, the initial state can be set by overriding the method `initialState`, also inherited from `State`.
6. Define the pseudo states as inner classes of the state machine class, and as subclasses of a either `EntryPoint`, `ExitPoint`, `ShallowHistory` or `DeepHistory`.

It is important to remember that all types of states *must* have a parameterless constructor in order for automatic state instantiation to work.

4.4.2 Implementation of a media player

Listing 4.1 implements the interface `IMediaPlayer`, the state machine class `MediaPlayerStateMachine` and the root state `MediaPlayer`.

Lines 1-5 `IMediaPlayer` is the user defined interface that defines the type of the states of the media player state machine. The interface extends the framework interface `IState`, and defines the event method signatures `powerOn`, `powerOff`, etc.

Line 6 Defines the state machine class `MediaPlayerStateMachine` as subclass of `StateMachine`. The type of the states in the state machine class is required to implement `IMediaPlayer` by setting `IMediaPlayer` as argument to the generic parameter of `StateMachine`.

Lines 8-10 Defines the initial state of the media player to be `MediaPlayer` by overriding the method `initialState` of `StateMachine`.

Lines 13-15 Defines the event dispatch methods such that incoming events gets dispatched to the current state of the state machine.

Lines 18-25 The class `MediaPlayer` will serve as the root state of the state machine by extending `State` and implementing the user defined interface `IMediaPlayer`. The initial state is set to `Off` by overriding the method `initialState`. All event methods are implemented with empty bodies such that contained states, i.e. subclasses, will by default have empty behaviour for all events unless specified otherwise.

Listing 4.1: `IMediaPlayer`, `MediaPlayerStateMachine` and `MediaPlayer`

```
1  interface IMediaPlayer extends IState {
2    void powerOn();
3    void powerOff();
4    ...
5  }
6  class MediaPlayerStateMachine extends StateMachine<IMediaPlayer>
       {
7    // The initial state of the state machine
8    Class<? extends IMediaPlayer> initialState() {
9      return MediaPlayer.class;
10   }
11
12   // Event dispatch methods
13   void powerOn() { currentState().powerOn(); }
14   void powerOff() { currentState().powerOff(); }
15   ...
16
17   // State classes
18   protected class MediaPlayer extends State implements
         IMediaPlayer {
19     Class<? extends IMediaPlayer> initialState() {
20       return Off.class;
21     }
22     void powerOn() { }
23     void powerOff() { }
24     ...
25   }
26   ...
27 }
```

Listing 4.2 implements the composite states `On` and `CDMode`, and the simple state `CDStopped`. For the first time there will be problems on the user side regarding inheritance.

Lines 3-8 Defines `On` as a substate of `MediaPlayer` by extending the class `MediaPlayer`. The event method `powerOff` is overridden in order to define the transition from `On` to `Off`. Notice that the state reference to `On` is included in the call to `changeToState` such that the transition is validated for this and all contained states of `On`. Since `On` has no initial state, the method `initialState` is overridden to return `null`. The reason this is necessary is because `MediaPlayer` from listing 4.1 defines its initial state to be `Off`, and `On` is a subclass of `MediaPlayer` and hence will inherit the method `initialState` as defined in `MediaPlayer`.

Lines 9-18 Defines the state `CDMode` as substate of `On` which means that `CDMode` inherits the compound transition to `Off` triggered by the event `powerOff`. The initial state is set to `CDStopped`, and the transition to `CassetteMode` is defined by overriding the event method of `changeMode`. The entry and exit actions of `CDMode` is defined by overriding the methods `entry` and `exit`. Assume that the methods `cdMotorOn` and `cdMotorOff` exists.

Lines 19-28 Defines the state `CDStopped` as substate of `CDMode`. The transition to `CDPlaying` is defined by overriding the event method `play`. `CDStopped` has no entry and exit actions, and not an initial state. The methods `entry`, `exit` and `initialState` are inherited from the superclass `CDMode`, and must therefore be overridden as empty.

Listing 4.2: `On`, `CDMode` and `CDStopped`

```
1  class MediaPlayerStateMachine extends StateMachine<IMediaPlayer>
     {
2    ...
3    protected class On extends MediaPlayer {
4      Class<? extends IMediaPlayer> initialState() {
5        return null;
6      }
7      void powerOff() { changeToState(On.class, Off.class); }
8    }
9    protected class CDMode extends On {
10     Class<? extends IMediaPlayer> initialState() {
11       return CDStopped.class;
12     }
13     void entry() { cdMotorOn(); }
14     void exit() { cdMotorOff(); }
15     void changeMode() {
16       changeToState(CDMode.class, CassetteMode.class);
17     }
18   }
19   protected class CDStopped extends CDMode {
20     Class<? extends IMediaPlayer> initialState() {
21       return null;
22     }
```

```
23      void entry() { }
24      void exit() { }
25      void play() {
26        changeToState(CDStopped.class, CDPlaying.class);
27      }
28    }
29  }
```

With the problems of listing 4.2, where entry/exit actions and initial states from enclosing states are inherited by contained states, one could think of what would happen if the state classes included other properties like constructors with behaviour, class attributes, methods that are not events etc. Obviously these will cause problems for contained states as the contained states will contain properties from enclosing states that are not intended for them. The intent of composite states is not to share any properties with substates (only event methods), which is a good indication that inheritance has a major weakness when it comes to representing state hierarchy.

Listing 4.3 implements entry point `OnEntryPoint` and the shallow history pseudo state `OnShallowHistory`.

Lines 3-10 The entry point connected to `On`. The methods `compositeState` and `targetPseudoState` defined in `EntryPoint` (see table 3.4 on page 30) are overridden in order to define specific behaviour. The method `compositeState` returns the state reference to `On`, because `On` is the composite state the entry point is connected to. The method `targetPseudoState` returns the state reference to the shallow history pseudo state of `On`, namely `OnShallowHistory`.

Lines 11-18 The shallow history pseudo state contained in `On`. Defines its enclosing state by overriding the method `compositeState` to return the state reference to `On`. The default history is defined by overriding the method `defaultHistoryState` to return the state reference to `CDMode`. The methods of `ShallowHistory` is described in table 3.5 on page 31.

Listing 4.3: `OnEntryPoint` and `OnShallowHistory`

```
1  class MediaPlayerStateMachine extends StateMachine<IMediaPlayer>
     {
2    ...
3    protected class OnEntryPoint extends EntryPoint {
4      public Class<? extends IMediaPlayer> compositeState() {
5        return On.class;
6      }
7      public Class<? extends PseudoState> targetPseudoState() {
8        return OnShallowHistory.class;
9      }
10   }
11   protected class OnShallowHistory extends ShallowHistory {
12     public Class<? extends IMediaPlayer> compositeState() {
13       return On.class;
14     }
```

```
15      public Class<? extends IMediaPlayer> defaultHistoryState() {
16        return CDMode.class;
17      }
18    }
19  }
```

The remaining states of the media player will not be implemented because they do not introduce any new concepts, nor do they introduce new type of problems.

4.5 Summary

Inheritance has in some areas proved to be a good candidate for representing state hierarchy in a combined modeling and programming language.

- Compound transitions are made possible through virtual methods.
- The tree structure of a state hierarchy is satisfied by the class inheritance hierarchy of state classes. This makes the implementation of entry/exit actions, entry/exit points and history possible, as described in chapter 3.

While the above results are positive, there are also several problems.

- Constructors, class attributes and methods that are not event methods are inherited in contained states. The consequence is that contained states will have properties that is not intended for them.
- Methods like `initialState`, `entry` and `exit` defined in `State` will be inherited from enclosing states. This will potentially create problems for contained states as their behaviour may become incorrect. As a solution to the problem, the user could override these methods either as empty or with some behaviour. However, this solution is not desirable as it introduce unnecessary coding and potential maintenance problems.
- In order to validate transitions, the user must explicitly define the source state and the target state of the transition. It would be better to only define the target state.

At first inheritance looks like a good solution to the state hierarchy problem. For composite states in isolation it seems to work, but when introducing new mechanisms the solution becomes unsatisfactory. It seems that inheritance will do as much good as bad to the problem, and is therefore not a satisfactory solution.

Chapter 5

State hierarchy by delegation

5.1 Introduction

Having excluded the possibility of using inheritance as a means of representing state hierarchy, a different approach is investigated using state objects that are linked by *delegation*. In short, delegation is a mechanism where objects automatically delegate method calls of methods they do not have to other objects through explicitly defined *delegation links*. The delegation link of a contained state will denote the enclosing state, which implies that events that are not explicitly defined in a contained state will automatically be delegated to its enclosing state. Delegation links are obviously similar to subclasses with virtual methods, and hence also a good candidate for supporting compound transitions. However, delegation has the benefit that contained states do not inherit from their enclosing states.

For the purpose of investigating the delegation approach, the framework from chapter 3 will be completed by adding the missing pieces regarding state hierarchy representation and analysis. Section 5.2 will briefly look at delegation as a language mechanism to get an overview of the basic principles. Since Java do not support delegation, section 5.3 will explain how delegation can be simulated using the *delegation pattern*. An overview of the framework will be presented in section 5.4. In section 5.5 the remaining parts of the state machine framework will be implemented. Section 5.6 gives a tutorial on the usage of the delegation framework, which will show the several advantages that delegation has compared to inheritance. Section 5.7 gives a summary.

5.2 Delegation

The language mechanism called *delegation* was originally presented by Lieberman [11] as an answer to the traditional philosophical controversy on how to reuse behaviour of objects and classes. The notion of

classes, object instances and inheritance is the most commonly used language mechanisms found in object-oriented languages like Simula 67 [2], Smalltalk [5] and Java [6]. The idea of delegation was to share behaviour specifications between objects in prototype-based languages, i.e. languages with only objects and not classes.

With delegation, an object, referred by Lieberman as the *delegator*, has one or more delegation links to *delegatees*. A delegator may have several delegation links to delegatees, meaning that if the delegator receives a message it cannot handle, it will automatically forward the message to all delegation links (one at the time), and hopefully one of the delegatees has the functionality to handle the message and respond back.

Example 5.1. *Figure 5.1 illustrates the sequence diagram of a company containing the objects `Boss` and `Worker`, in addition to an object `Company` that calls operations on the `Boss`. The `Boss` (delegator) are able to boss around, while the `Worker` (delegatee) do work. None of them can do both operations. If the `Boss` object gets the message from `Company` to boss around, it will do so and make a response back. If, on the other hand, the `Boss` gets the message that tells it to do work, the `Boss` will delegate the message to the `Worker` object and eventually get a response back. The response from the `Worker` is then forwarded to `Company` making it seem like the `Boss` has done the work. The delegation will take place because the operation for doing work is not implemented in the `Boss` object. If the `Boss` possess both operations, the message would not be delegated (the same effect as overriding a virtual method). A message to the `Worker` which tells it to boss around will be ignored because the object itself has no implementation for it, nor does it have a delegation link to delegate the message to.*

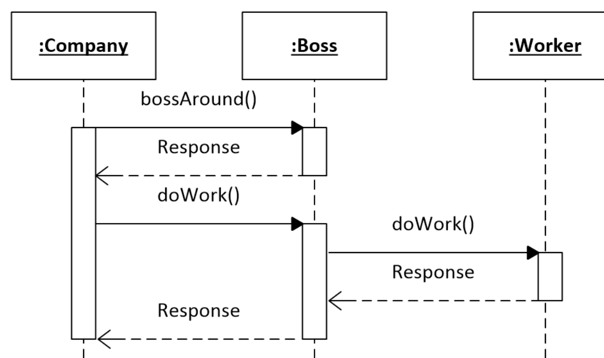


Figure 5.1: `Boss`/`Worker` sequence diagram

Delegation is a mechanism that is often considered an *alternative* to inheritance, but taken literally there is no reason that a language may not support both inheritance and delegation. Inheritance is a mechanism for specifying specialization and therefore a relationship between classes, while delegation involves relationships between objects. To illustrate this idea, figure 5.2 shows that the class `C` is a subclass of `A` and thereby inherit methods of `A`. Calls to methods of `C` inherited from `A` (and possibly redefined)

will be executed by a `C`-object, while calls of the method `call()` are delegated to the `B`-object denoted by the delegation link `b`.

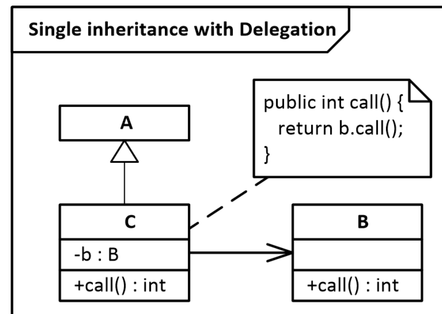


Figure 5.2: Combining inheritance and delegation

5.3 Delegation in Java

The Java language is not prototype-based and hence does not support delegation between objects implicitly. The mechanism can however be simulated by the programmer.

5.3.1 Delegation pattern

The delegation pattern[1] is a design pattern that simulates method delegation. The idea is simple: for each method not supported by the delegator, use the delegation link to delegate the method call.

Listing 5.1 shows the Boss-Worker example from example 5.1 using the principles of the delegation pattern. The class `Worker` has no delegation link and hence no implementation of the method `bossAround`. The method `doWork` contains code for doing work. The class `Boss` implements code for `bossAround()`, but have no implementation of `doWork` which means that the method delegates the call to the `Worker` delegatee.

Listing 5.1: `Boss` and `Worker`

```
1 class Worker {
2   void doWork() { <code> }
3 }
4 class Boss {
5   Worker delegatee;
6
7   void bossAround() { <code> }
8   void doWork() { delegatee.doWork(); }
9 }
```

[1] http://best-practice-software-engineering.ifs.tuwien.ac.at/patterns/delegation.html

5.4 Overview

With the inheritance approach, the class hierarchy of state classes formed a tree data structure that represented the state hierarchy structure. This is not the case with delegation. In order to avoid that contained states inherit properties from enclosing states, the state classes will all be immediate subclasses of `State`. Figure 5.3 illustrates the media player from chapter 3 using the delegation framework.

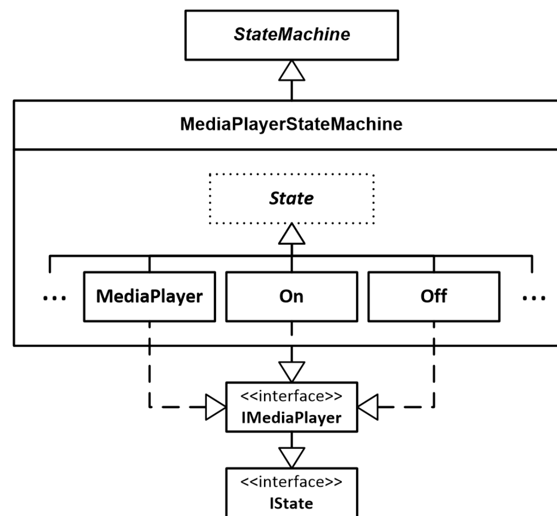


Figure 5.3: Delegation framework class hierarchy

The user defined state machine `MediaPlayerStateMachine` is a subclass of `StateMachine` and contains the user defined state classes. As a consequence of the framework being generic, see section 3.4.2 on page 26, the interface `IMediaPlayer` is defined and set as the generic type of the framework, such that all states of the state machine must implement the event methods defined in `IMediaPlayer`.

All user defined state classes are subclasses of `State` and implements the interface `IMediaPlayer`. This means that each state are responsible of implementing all the event methods, and must simulate delegation for the events that are not specific for them.

By looking at the class hierarchy of figure 5.3, it is impossible to know how the state hierarchy will look like. The state hierarchy first reveals itself when looking at the object hierarchy of a state machine. Figure 5.4 shows the object hierarchy of figure 5.3, where e.g. `CDMode` is the enclosing state of `CDStopped` and `CDPlaying`. `CDMode` will serve as an enclosing state, because any event method that are not specified by `CDStopped` and `CDPlaying` will be delegated to `CDMode`. This means that any transitions defined in event methods of `CDMode` will be compound for `CDStopped` and `CDPlaying`.

Example 5.2. *Assume that the current state of figure 5.4 is* `CassettePlaying`*, and that the event* `powerOff` *is occurring.* `CassettePlaying` *has no definition of* `powerOff`*, so the event is delegated to* `CassetteMode`*.* `CassetteMode` *has no*

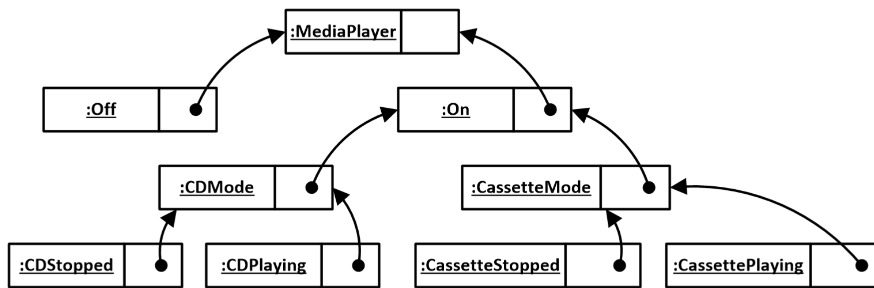


Figure 5.4: State hierarchy represented by delegation links between objects

definition of `powerOff`, so the event is delegated to `On`. Finally, `On` defines `powerOff` to cause a transition to `Off`.

5.4.1 Framework classes

Table 5.1 lists the most important classes of the framework with a short description of each with corresponding constructors. All classes of the framework is abstract which makes them impossible to instantiate without making them *concrete* by subclasses.

5.5 Framework implementation

Most of the framework are already implemented in chapter 3. This section will explain how delegation is implemented as part of the framework in order to make the framework complete.

5.5.1 State hierarchy traversal

An important detail in the delegation framework is how the state hierarchy is traversed. This will especially be important when implementing state hierarchy analysis.

The state hierarchy is represented by an object hierarchy of states linked together by delegation links. This means that the framework must use the delegation links defined in `State` to traverse the state hierarchy. To accomplish this the method `enclosingState()` defined in `State` gives the framework access to the delegation link of a state.

5.5.2 `StateMachine`

In order to complete the `StateMachine` class, this section will explain how automatic state instantiation and the state hierarchy analysis are implemented.

Class name	Constructor(s)/description
`StateMachine <StateType extends IState>`	`State()` `State(Class<? extends StateType> initial)` A generic class that represents a general state machine with no events. The type parameter `StateType` represents the type of the user defined states. There are two constructors available: The parameterless constructor requires that the initial state is defined in the method `initialState` of `StateMachine`. As an alternative constructor, the user can set the initial state through the parameter `initial`.
`State`	`State(Class<? extends StateType> enclosing)` An inner class of `StateMachine`, and superclass of all user defined states. Implements the interface `IState`, see section 3.4.8. Defines its enclosing state by setting its delegation link through the parameter `enclosing`.
`PseudoState`	An inner class of `StateMachine`, and superclass of all pseudo states, i.e. `EntryPoint`, `ExitPoint`, `ShallowHistory` and `DeepHistory`. Implements the interface `IVertex`, and also defines functionality that is common for all pseudo state types the inheritance framework.
`EntryPoint`	`EntryPoint()` The superclass of psuedo states representing entry points.
`ExitPoint`	`ExitPoint()` The superclass of pseudo states representing exit points.
`ShallowHistory`	`ShallowHistory()` The superclass of pseudo states representing shallow history.
`DeepHistory`	`DeepHistory()` The superclass of pseudo state representing deep history.

Table 5.1: Overview of the delegation framework classes and constructors

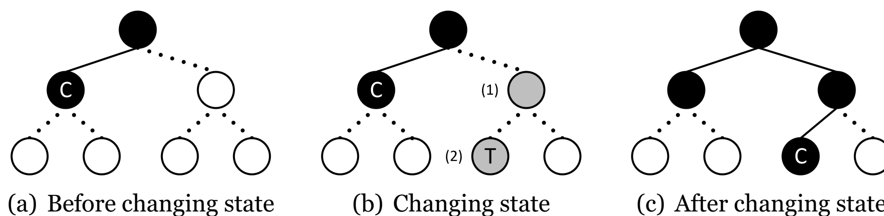
Automatic state instantiation

Java is based on the notion of classes, while delegation on the notion of objects. To enable delegation between states in Java, the user must manually instantiate each state, set the correct delegation links and so forth. In order to make it simpler for the user, the framework support automatic state instantiation by using Java reflection.

In order to reuse state instances, `StateMachine` will maintain two hash maps, one with state instances and one with pseudo state instances. Each instance is mapped by its state reference. If a state reference has no corresponding instance, `StateMachine` will instantiate it automatically and put it into one of the hash maps.

To enable automatic state instantiation, the user must specify delegation links in each state by setting a state reference to their enclosing state, see the constructor of `State` in table 5.1. In addition, it is required that each user state class must have a parameterless constructor such that reflection will work.[2]

Figure 5.5 illustrates how automatic state instantiation works, where C is the current state and T is the target state. The *black* states are already instantiated (referenced in the hash map of `StateMachine`), the *grey* states are about to be instantiated and the *white* states are not instantiated. The *solid* lines between the states are delegation links that are set, while the *dotted* lines are not set. In figure 5.5(a) only C and the enclosing state of C is in the hash map of `StateMachine` marked as black. Figure 5.5(b) shows the process of what is happening when changing state to T. Both T and its enclosing state are marked grey. They will be instantiated in the order marked by (1) followed by (2). The reason for this ordering is that T must set its delegation link to an actual object. Notice that the root state is not instantiated because it is already black. Figure 5.5(c) is the result of changing state from C to T.



(a) Before changing state (b) Changing state (c) After changing state

Figure 5.5: Automatic state instantiation

Implementation of state hierarchy analysis

The implementation of state hierarchy analysis is dependent on how the state hierarchy is represented. Below is the implementation details of the analysis methods of `StateMachine` when using the object hierarchy of states.

`isRoot(state)`: If the enclosing state of `state` is a reference to `null`, the state is the root state.

`hasSameEnclosingState(stateA, stateB)`: If the reference to the enclosing state of `stateA` and `stateB` are equal, the method will return `true`.

[2]The framework assumes that a state has no parameters in their constructor. If the framework attempts to instantiate a state that does not satisfy this requirement, the exception `java.lang.InstantiationException` will be thrown.

isSubstateOf(fromState, toState): If toState is a state object on the path from fromState to the root state, the state fromState is a substate of toState. The method will traverse all enclosing states of fromState until it finds toState and returns true. If the traversal reaches the root state, and toState is not found, the method will return false.

isReachable(fromState, toState): The method traverse all enclosing states of fromState in the object hierarchy. If one of the enclosing states of fromState are the immediate enclosing state of toState, or toState is the root state, the method will return true.

5.5.3 State

The delegation framework defines State differently than in the inheritance framework of chapter 4. First of all, the constructor is different as it requires a state reference to the enclosing state. Secondly, the user methods defining transitions are simpler. See section 3.4.9 for general information about State.

User methods

State has methods methods for defining transitions, see table 5.2. The methods are declared protected such that it is only possible to define transitions inside event methods that are defined in user state classes.

protected final void changeToState(Class<? extends StateType> target) Makes the state machine, that this state is instantiated by, to change from the current state to the state referenced by target. The method will validate the transition automatically by checking that the source state where the transition is defined has the same enclosing state as target.
protected final void changeToPseudoState(Class<? extends PseudoState> target) The same as changeToState, but target is a state reference to a pseudo state.

Table 5.2: User methods of State

5.5.4 State transitions

The methods changeToState and changeToPseudoState of table 5.2 are used for defining state transitions. Both methods takes one parameter that specifies the target state of the state machine. The framework will then validate the transition by dynamically getting the state reference to the state where the transition is defined, and checking that the *source state* has the same enclosing state as the *target state*.

With the inheritance framework, the source state had to be hard coded for each transition, see section 4.3.3 on page 40. With the delegation approach, the methods `changeToState` and `changeToPseudoState` takes one parameter that specifies the target state. The framework will dynamically get the source state by using the method `getClass` defined in `java.lang.Object`. This approach works because delegated events are executing in other objects than the current state, and compound transitions are therefore called from the objects where the transitions are defined.

Example 5.3. *Assume that the current state of the media player is `CDPlaying`, and that the event `powerOff` is occurring.*

With the inheritance approach, the state class `CDPlaying` inherits the method `powerOff` from enclosing states which means that the source state will be `CDPlaying`. The source state `CDPlaying` and the target state `Off` do not have the same enclosing state, and hence the transition is not valid.

With the delegation approach, the event is delegated to `On` which defines the method `powerOff`. The source state is therefore `On` because the program is executing on the object instance of `On`. `On` and `Off` has the same enclosing state, and hence the transition is valid.

5.6 Framework usage

The main advantages of delegation first reveals itself from an users point of view. To illustrate these advantages, the media player will be implemented using the delegation framework.

5.6.1 Design principles

The design process of a state machine when using the delegation framework is much similar to inheritance approach, but with a few distinct differences. Implementing a state machine with the delegation framework involves the following steps:

1. Define an interface that extends `IState` and adds the event method signatures of the state machine.
2. Define a state machine class that is a subclass of `StateMachine`. The generic parameter of `StateMachine` is set to the user defined interface, such that the states of the state machine will be required to implement all the events.
3. Define the event dispatch methods and the initial state of the state machine. The initial state is defined by overriding the method `initialState` of `StateMachine`, or by using the constructor of `StateMachine` that takes the initial state as argument, see constructors in table 5.1.
4. Define all the state classes, including the root state, as inner classes of the state machine class, and as subclasses of `State`. Each state class

must implement the interface, i.e. the event methods, and simulate delegation for each event that is not specific for them.

5. Define the pseudo states as inner classes of the state machine class, and as subclasses of a either `EntryPoint`, `ExitPoint`, `ShallowHistory` or `DeepHistory`.

5.6.2 Implementation of a media player

Listing 5.2 implements the interface `IMediaPlayer`, the state machine class `MediaPlayerStateMachine` and the root state `MediaPlayer`.

Lines 1-5 Defines the interface `IMediaPlayer` as a subinterface of `IState`. The interface is extended to contain all the event methods of the media player state machine.

Line 6 The state machine class extends `StateMachine`, and makes sure that the states are required to implement `IMediaPlayer` by setting the generic type parameter of `StateMachine` to `IMediaPlayer`.

Lines 8-10 Defines the initial state of the media player by overriding the method `initialState` of `StateMachine` to return the state reference to `MediaPlayer`.

Lines 13-15 The event dispatch methods that makes sure that incoming events are dispatched to the current state. Notice that the user gets access to the methods `powerOn()`, `powerOff` etc., because the type returned from `currentState` is set to `IMediaPlayer` via the generic type parameter of `StateMachine`.

Lines 18-26 Defines the state `MediaPlayer` which will be the root state, because the enclosing state (delegation link) is set to `null` via the constructor of `State`. The initial state is set to `Off` by overriding the method `initialState` inherited from `State`. The event methods are implemented with empty bodies, first of all because there is no defined behaviour in the actual state machine, and secondly because there is no enclosing state that the events can be delegated to.

Listing 5.2: `IMediaPlayer`, `MediaPlayerStateMachine` and `MediaPlayer`

```
1  interface IMediaPlayer extends IState {
2    void powerOn();
3    void powerOff();
4    ...
5  }
6  class MediaPlayerStateMachine extends StateMachine<IMediaPlayer>
     {
7    // Initial state of the state machine
8    Class<? extends IMediaPlayer> initialState() {
9      return MediaPlayer.class;
10   }
11
```

```
12    // Event dispatch methods
13    void powerOn() { currentState().powerOn(); }
14    void powerOff() { currentState().powerOff(); }
15    ...
16
17    // State classes
18    protected class MediaPlayer extends State implements
          IMediaPlayer {
19      MediaPlayer() { super(null); } // Root state
20      Class<? extends IMediaPlayer> initialState() {
21        return Off.class;
22      }
23      void powerOn() { }
24      void powerOff() { }
25      ...
26    }
27    ...
28  }
```

Listing 5.3 implements the composite states `On` and `CDMode`, and the simple state `CDStopped`.

Lines 3-8 Definition of the composite state `On` where its enclosing state is defined to be `MediaPlayer`. The event method `powerOn` delegates the event to the enclosing state, while the event method `powerOff` defines a transition to the state `Off`.

Lines 9-18 Definition of the composite state `CDMode` where its enclosing state is defined to be `On`. The entry and exit action methods are overridden such that the methods `cdMotorOn` and `cdMotorOff` are executed.[3] The event method `changeMode` defines a transition to `CassetteMode`. The other event methods must delegate to the enclosing state of `CDMode`.

Lines 19-23 Definition of the simple state `CDStopped` where its enclosing state is defined to be `CDMode`. The event method `play` defines the transition to `CDPlaying`. The other event methods must delegate to the enclosing state of `CDStopped`.

Listing 5.3: `On`, `CDMode` and `CDStopped`

```
1  class MediaPlayerStateMachine extends StateMachine<IMediaPlayer>
       {
2    ...
3    protected class On extends State implements IMediaPlayer {
4      On() { super(MediaPlayer.class); }
5      void powerOn() { enclosingState().powerOn(); }
6      void powerOff() { changeToState(Off.class); }
7      ...
8    }
9    protected class CDMode extends State implements IMediaPlayer {
10     CDMode() { super(On.class); }
11     Class<? extends IMediaPlayer> initialState() {
```

[3]Assume that the methods `cdMotorOn` and `cdMotorOff` exists.

```
12        return CDStopped.class;
13      }
14      void entry() { cdMotorOn(); }
15      void exit() { cdMotorOff(); }
16      void changeMode() { changeToState(CassetteMode.class); }
17      ...
18    }
19    protected class CDStopped extends State implements IMediaPlayer
          {
20      CDStopped() { super(CDMode.class); }
21      void play() { changeToState(CDPlaying.class); }
22      ...
23    }
24  }
```

It is important to point out that even though the states of listing 5.3 are substates of a composite state, they are all immediate subclasses of `State`, and not their enclosing states as in the inheritance approach. The benefit is that the methods `initialState`, `entry` and `exit` are not inherited by contained states, and hence the result is a cleaner and less error prone code. In order to fully see the benefits of delegation, compare `CDStopped` in listing 5.3 with `CDStopped` in listing 4.2 on page 43.

The entry point and history pseudo states are implemented exactly the same way as with the inheritance approach. See listing 4.3 on page 44.

The remaining states of the media player will not be implemented as they do not introduce any new concepts.

5.6.3 Implicit delegation by inheritance

The task of simulating delegation for all events in all the states may be a tiresome task. In order to simplify the simulation, one could create a class `MediaPlayerState`, which is a subclass of `State` and superclass of all states, see figure 5.6. `MediaPlayerState` will implement the interface `IMediaPlayer`, and implement all events of the interface. The events will contain code that simulates delegation. The states, which are subclasses of `MediaPlayerState`, will then implicitly delegate for each event that is not explicitly overridden.

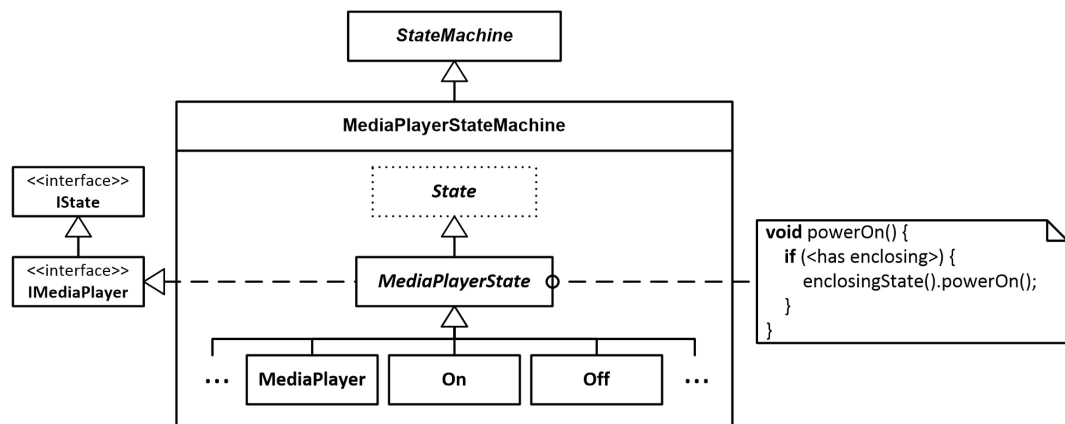


Figure 5.6: Implicit delegation by inheriting `MediaPlayerState`

Listing 5.4 illustrates how `MediaPlayerState` will look like. With respect to the root state, there must be a *general way* of delegating. The tests on line 5 and 10 are therefore needed to ensure that the enclosing state of the current state is not `null`. Without these checks, the state machine may crash by calling a reference to `null`, resulting in a Java `NullPointerException`.

Listing 5.4: `MediaPlayerState`

```
1  class MediaPlayerStateMachine extends StateMachine<IMediaPlayer>
       {
2    ...
3    protected class MediaPlayerState extends State implements
         IMediaPlayer {
4      void powerOn() {
5        if (enclosingState() != null) {
6          enclosingState().powerOn();
7        }
8      }
9      void powerOff() {
10       if (enclosingState() != null) {
11         enclosingState().powerOff();
12       }
13     }
14     ...
15   }
16 }
```

5.7 Summary

The goal of investigating inheritance and delegation was to find a good way to represent state hierarchy. The investigation is presented through two similar frameworks that implement the most basic functionality in UML state machines. The frameworks alone are not very interesting, but is used as tools to show what challenges that needs to be considered. Based on the investigation, delegation has shown to have several advantages compared to inheritance. The problems with inheritance are solved as well as keeping the positive sides.

- Method delegation is a better choice than virtual methods when it comes to compound transitions, because the framework is able to validate transitions automatically, without forcing the user to explicitly define the source state as with the inheritance approach.
- The object hierarchy has the same tree structure as the class hierarchy, and hence a good representation of state hierarchy.
- Implementation of entry/exit actions, entry/exit points and history is as simple as in the inheritance framework, only using the object hierarchy instead of the class hierarchy.
- Framework methods like `entry`, `exit` and `initialState` defined in `State` are *not* inherited by contained states, because contained states are no

longer represented by subclasses, which makes the code cleaner and easier to maintain.

- As a consequence of delegation representing state hierarchy rather than subclasses, constructors, class attributes and other methods are *not* inherited by contained states.

Even though not directly related to the state hierarchy problem, there has been introduced some new problems that are related to the Java language.

- Java does not support delegation as a language mechanism. The consequence is that the user must simulate delegation with the delegation pattern which means extra coding for the user.

- Java is based on the notion of classes. Since the abstraction of contained states are represented by object references, the code has to make sure to instantiate each state and set the correct delegation links at runtime.

As may be observed above, a problem with the investigation is Java. The language is strict in its type system, and the extra coding needed to simulate delegation is frustrating. However, delegation as a language mechanism appears to be a good abstraction for state hierarchy and should be investigated further.

Chapter 6

Specialization of state machines

6.1 Introduction

Object-oriented programming languages offer the ability to create *specializations* of existing classes through language mechanisms like e.g. inheritance. In modeling, the notion of specialized state machines has been discussed in [15], and has later become a part of UML2, see e.g. [9]. In such specialized state machines, the states and transitions are inherited from super state machines. New states and transitions may then be *added*. Inherited states and transitions in a super state machine may be overridden.

In an object-oriented language that supports advanced state machines with delegation representing state hierarchy, it is natural to think that inheritance can be used as a means of representing specialized state machines.

Even though only for simple state machines, [1] presents an *extensible state machine pattern* which is based on the state design pattern. The extensible state machine pattern supports specialization by using inheritance, interfaces, generics and *factory methods* [4]. Fortunately, it turns out that parts of the pattern also applies to the delegegation framework, but with some modifications in order to intergrate with the framework.

This chapter will pursue the idea of combining the delegation framework from chapter 5, and subclassing of state machines and states to create specialized state machines. The ideas from [1] will be applied, which will eventually lead to the notion of using inheritance and *virtual classes* [13] as a means of representing specialized state machines.

Section 6.2 presents a specialized switch that will be used as an example in the following sections. Section 6.3 will implement the switch by using the extensible state machine pattern applied to the framework. Section 6.4 drastically simplifies the extensible state machine pattern by replacing the use of interfaces, generics and factory methods with virtual classes. Section 6.5 gives a summary.

6.2 A specialized switch

For the purpose of illustrating specialization of state machines, a simple on/off switch will be extended to a media switch with modes. The media switch is a simplified version of the media player from chapter 3

Figure 6.1 shows how the on/off switch named `SwitchStateMachine` is extended to the specialized media switch named `MediaSwitchStateMachine`. The states `CDMode` and `CassetteMode`, and the event `changeMode` with corresponding transitions are added. The *dashed* lines are the states and transitions that are inherited from `SwitchStateMachine`.

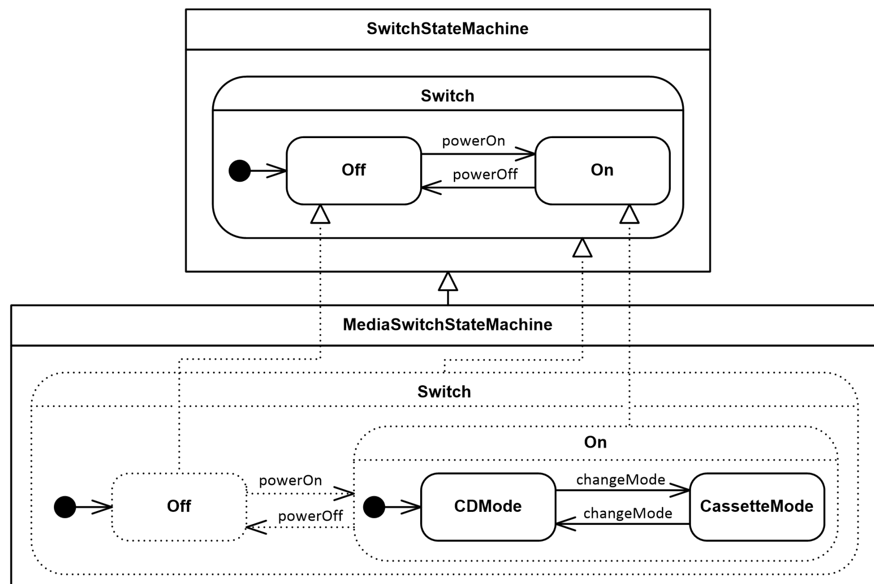


Figure 6.1: Specialized media switch

6.3 Using the extensible state machine pattern

This section will build the media switch in stages, starting with the simple on/off switch and extend it by overriding states, adding states and adding events. The principles of the extensible state machine pattern will be introduced when necessary.

6.3.1 Implementing the on/off switch

The starting point of implementing the media switch is to implement the on/off switch according to the delegation approach, see chapter 5. Figure 6.2 illustrates the class hierarchy of the on/off switch. All states are immediate subclasses of `State`, and implements the interface `ISwitch` which contains the events `powerOn` and `powerOff`. The state machine represented by the class `SwitchStateMachine` must support all events such that incoming events are dispatched to the current state.

Listing 6.1 implements the on/off switch.

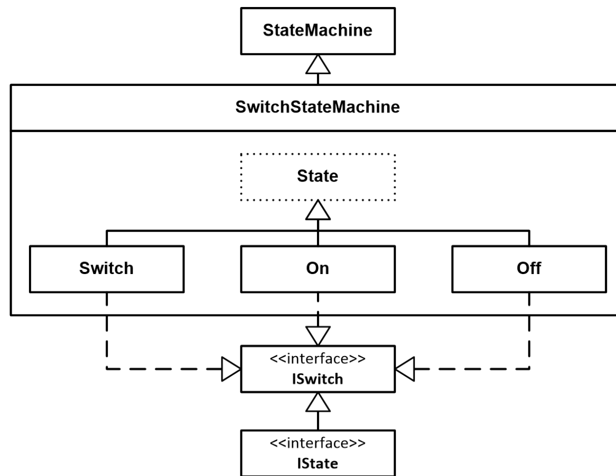


Figure 6.2: Class hierarchy of the on/off switch

Lines 1-4 The interface that defines the events `powerOn` and `powerOff`. Is a subinterface of `IState` as required by the delegation approach.

Lines 7-17 The state classes which are immediate subclasses of `State` and implements the interface `ISwitch`. `On` defines the enclosing state to be `Switch`, simulates delegation for the event `powerOn` and defines the transition to `Off` in the event `powerOff`. `Off` has the enclosing state `Switch` and defines the transition to `On` in the event `powerOn`.

Listing 6.1: `SwitchStateMachine`

```
1  interface ISwitch extends IState {
2    void powerOn();
3    void powerOff();
4  }
5  class SwitchStateMachine extends StateMachine<ISwitch> {
6    // State classes
7    protected class Switch extends State implements ISwitch { ... }
8    protected class On extends State implements ISwitch {
9      On() { super(Switch.class); }
10     void powerOn() { enclosingState().powerOn(); }
11     void powerOff() { changeToState(Off.class); }
12   }
13   protected class Off extends State implements ISwitch {
14     Off() { super(Switch.class); }
15     void powerOn() { changeToState(On.class); }
16     void powerOff() { enclosingState().powerOff(); }
17   }
18
19   // Event dispatch methods, initial state, etc...
20   ...
21 }
```

6.3.2 Implementing the media switch

The idea of the specialized media switch is to extend `SwitchStateMachine` to the class `MediaSwitchStateMachine` by using inheritance, see figure 6.3. `MediaSwitchStateMachine` will then inherit the dispatch event methods and the initial state definition from `SwitchStateMachine`, and get access to the state classes `Switch`, `On` and `Off`.

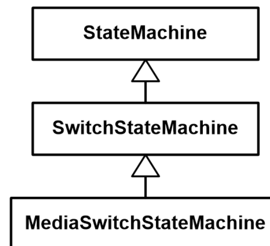


Figure 6.3: Specialization by inheritance

Creating the specialization involves adding the new event `changeMode`, the new states `CDMode` and `CassetteMode`, and setting `CDMode` as the initial state of `On` which means that `On` must be overridden.

The first principle that will be discussed is overriding of states *without* introducing new events: In order to override `On`, a subclass is created which overrides its `initialState`-method to return the state reference to `CDMode`, see figure 6.4.

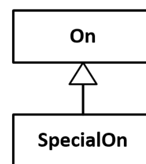


Figure 6.4: Specialization of states

Unfortunately, the transition from `Off` to `On` as defined on line 15 in listing 6.1, is not using the state reference to the specialized state. This leads to the first constraint which is adapted from the extensible state machine pattern:

> *Constraint: There must exist a consistent way of reference to states that will allow future specializations to override the implementation of a state class.*

As a consequence of this constraint, factory methods are introduced as a means of referencing to states. States must therefore be referenced through the factory methods, see listing 6.2.

Listing 6.2: `SwitchStateMachine` with factory methods

```
1 class SwitchStateMachine extends StateMachine<ISwitch> {
2   // Factory methods
3   protected Class<? extends ISwitch> stateSwitch() {
```

```
4      return Switch.class;
5    }
6    protected Class<? extends ISwitch> stateOn() {
7      return On.class;
8    }
9    protected Class<? extends ISwitch> stateOff() {
10     return Off.class;
11   }
12
13   // State classes
14   protected class Switch extends State implements ISwitch { ... }
15   protected class On extends State implements ISwitch {
16     On() { super(stateSwitch()); }
17     void powerOn() { enclosingState().powerOn(); }
18     void powerOff() { changeToState(stateOff()); }
19   }
20   protected class Off extends State implements ISwitch {
21     Off() { super(stateSwitch()); }
22     void powerOn() { changeToState(stateOn()); }
23     void powerOff() { enclosingState().powerOff(); }
24   }
25
26   // Event dispatch methods, initial state, etc...
27   ...
28 }
```

The factory methods can then be overridden to return specialized states in specialized state machines as shown below.

```
class MediaSwitchStateMachine extends SwitchStateMachine {
    @Override
    protected Class<? extends ISwitch> stateOn() {
        return SpecialOn.class;
    }
    ...
}
```

The second principle that will be discussed is the adding of new events. In order to add the event `changeMode`, the interface `ISwitch` must be extended in order for the framework to get the correct type. Secondly, all states must implement the new interface which means that `Switch`, `On` and `Off` must be specialized.

With these steps in mind, there are obviously problems with listing 6.2 concerning types. The main problem is that the framework is locked to the type `ISwitch`. The states returned from the framework will therefore be of type `ISwitch` instead of the type of the current interface that also contains the event method `changeMode`. Another problem is that if the user forgets to override the factory methods to return the specialized states, the compiler will not complain, which means that some states do not support all events. These problems leads to the second constraint, also adopted from the extensible state machine pattern:

> *Constraint: Each state must abstract over the events it responds to. While it may require that certain events exist, it may*

not limit what events can be added by future specializations.

In order to satisfy this constraint, generics are introduced to `SwitchStateMachine`, see listing 6.3. A consequence of using generics with the factory methods is that the factory methods must be declared `abstract`, making the state machine class `abstract` as well. The compiler has simply no way of telling what type `StateType` will be.

Listing 6.3: `SwitchStateMachine` with generics

```
1  abstract class SwitchStateMachine<StateType extends ISwitch>
     extends StateMachine<StateType> {
2    // Factory methods
3    protected abstract Class<? extends StateType> stateSwitch();
4    protected abstract Class<? extends StateType> stateOn();
5    protected abstract Class<? extends StateType> stateOff();
6
7    // State classes
8    ...
9    // Event dispatch methods, initial state, etc...
10   ...
11 }
```

The downside of using generics is that the state machine classes must be declared `abstract`, and therefore the state machine classes cannot be instantiated. If `SwitchStateMachine` is to be used, the class must be made *concrete* by creating a subclass that implements the factory methods, see listing 6.4. The framework and factory methods is now using the type `ISwitch`. The compiler can type check the return values of the factory methods such that the state references that is returned are guaranteed to implement the interface `ISwitch`.

Listing 6.4: `ConcreteSwitchSM`

```
1  class ConcreteSwitchSM extends SwitchStateMachine<ISwitch> {
2    protected Class<? extends ISwitch> stateSwitch() {
3      return Switch.class;
4    }
5    protected Class<? extends ISwitch> stateOn() {
6      return On.class;
7    }
8    protected Class<? extends ISwitch> stateOff() {
9      return Off.class;
10   }
11 }
```

With these principles the specialized media switch can be implemented. Figure 6.5 gives a graphical overview of how the on/off switch is specialized to the media switch. `IMedia` is the new interface that extends `ISwitch` and adds the event `changeMode`. The previously defined states from `SwitchStateMachine` are extended to the classes `SpecialSwitch`, `SpecialOn` and `SpecialOff`, which all implements the `IMedia` interface. The classes `CDMode` and `CassetteMode` are new states and are therefore immediate subclasses of `State`, and implements `IMedia`.

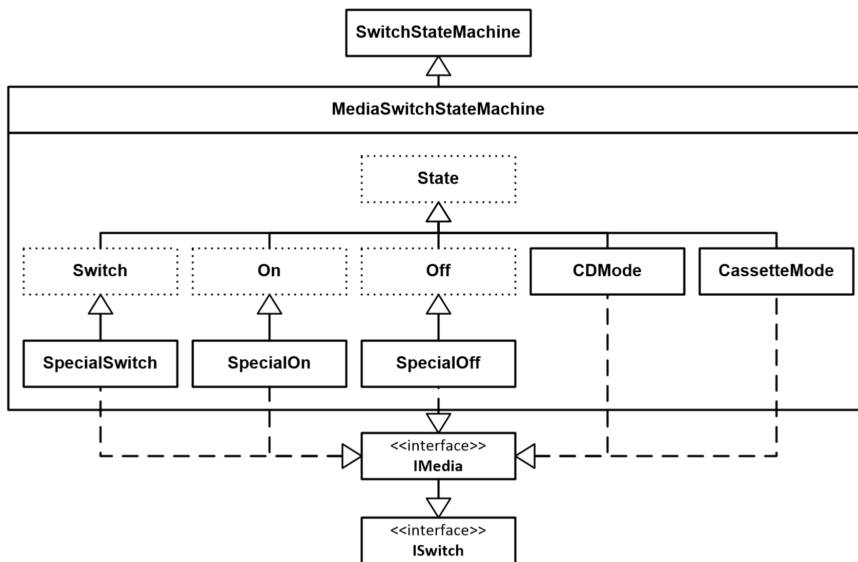


Figure 6.5: Class hierarchy of the media switch

Listing 6.5 implements the media switch.

Lines 1-3 The `IMedia` interface that defines the event `changeMode`.

Line 4 The generic type `StateType` is now required to implement the `IMedia` interface. `StateType` is used as argument to the generic type of `SwitchStateMachine` such that the inherited factory methods,i.e. `stateSwitch`, `stateOn` and `stateOff`, are expected to return class objects that implements all events of `IMedia`.

Line 6 Adds the event method `changeMode` that dispatches the event to the current state.

Lines 9-14 Adds the states `CDMode` and `CassetteMode`.

Lines 17-18 For each new state, an abstract factory method must be added. The factory methods for `CDMode` and `CassetteMode` is therefore added. The factory methods for the states `On`, `Off` and `Switch` are inherited from `SwitchStateMachine`.

Lines 21-34 Extends the inherited states with the `changeMode` event defined in `IMedia`. `SpecialOn` must override the method `initialState` to define the initial state to be `CDMode`.

Listing 6.5: `MediaSwitchStateMachine`

```
1 interface IMedia extends ISwitch {
2   void changeMode();
3 }
4 abstract class MediaSwitchStateMachine<StateType extends IMedia>
      extends SwitchStateMachine<StateType> {
5   // New state machine events
```

```
6    void changeMode() { currentState().changeMode(); }
7
8    // New states
9    protected class CDMode extends State implements IMedia {
10     ...
11   }
12   protected class CassetteMode extends State implements IMedia {
13     ...
14   }
15
16   // Factory methods for new states
17   protected abstract Class<? extends StateType> stateCDMode();
18   protected abstract Class<? extends StateType> stateCassetteMode
       ();
19
20   // Creating specialized states
21   protected class SpecialOn extends On implements IMedia {
22     public Class<? extends StateType> initialState() {
23       return stateCDMode();
24     }
25     public void changeMode() {
26       enclosingState().changeMode(); // Delegation
27     }
28   }
29   protected class SpecialOff extends Off implements IMedia {
30     ...
31   }
32   protected class SpecialSwitch extends Switch implements IMedia
       {
33     ...
34   }
35 }
```

In order to use `MediaSwitchStateMachine`, the class must be extended to make it concrete. Listing 6.6 implements the concrete media switch by implementing the factory methods to return the correct state references. Notice that the factory methods are required to return state references to states that have all the methods of `IMedia`. An attempt of returning the state references to the previously defined `Switch`, `On` and `Off` will give a compilation error because they do not implement `IMedia`.

Listing 6.6: `ConcreteMediaSwitchSM`

```
1  class ConcreteMediaSwitchSM extends MediaSwitchStateMachine<
       IMedia> {
2    protected Class<? extends IMedia> stateSwitch() {
3      return SpecialSwitch.class;
4    }
5    protected Class<? extends IMedia> stateOn() {
6      return SpecialOn.class;
7    }
8    protected Class<? extends IMedia> stateOff() {
9      return SpecialOff.class;
10   }
11   protected Class<? extends IMedia> stateCDMode() {
12     return CDMode.class;
13   }
```

```
14    protected Class<? extends IMedia> stateCassetteMode() {
15      return CassetteMode.class;
16    }
17 }
```

6.3.3 Summary of design principles

This section will summarize the design principles on how to create specialized state machines by using the delegation framework combined with the extensible state machine pattern.

Creating the base state machine

The *base state machine* is responsible for integrating with the framework, and offer the possibility of creating state machine specializations. E.g. `SwitchStateMachine` is the base state machine of the specialized media switch. The list of principles below describes how the base state machine is created.

- Define an interface that is a subinterface of `IState` and that contains all event method signatures of the state machine.

  ```
  interface IBase extends IState { ... }
  ```

- Create a state machine class declared `abstract` that extends `StateMachine`. Define an event dispatch method for each event, and define the initial state of the state machine. The state machine class must be generic with *one* type parameter that is required to extend the defined interface.

  ```
  abstract class BaseStateMachine<StateType extends IBase>
      extends StateMachine<StateType> { ... }
  ```

  The type parameter named `StateType` is used as argument to the generic type parameter of `StateMachine`, such that the framework will work with the same type of states as the user defined state machine.

- Define all the state classes as inner classes of `BaseStateMachine`. All states must implement `IBase` and extend `State`.

  ```
  ...
  protected class AState extends State implements IBase { ... }
  ...
  ```

- For each state, define a factory method declared `abstract` in `BaseStateMachine`, where the return type is a state reference to a state class that is required to extend the generic type parameter of the state machine.

```
...
protected abstract Class<? extends StateType> aStateReference();
...
```

Creating a specialized state machine

Having the base state machine created, the list of principles below is used to create specializations. The term *super state machine* will be used to refer to the state machine that will be specialized. E.g. `SwitchStateMachine` is the super state machine of `MediaSwitchStateMachine`.

- If new events are added, a new interface that extends the interface used in the super state machine must be created. The new interface adds the new event method signatures. Creating a specialization of the base state machine would imply that the new interface extends `IBase`, but in order define a general principle, `ISuper` will denote the interface of the super state machine.

  ```
  interface ISpecial extends ISuper { ... }
  ```

- Create a new state machine class declared `abstract` that is a subclass of the super state machine denoted as `SuperStateMachine`. Add new dispatching event methods if any. The new state machine class is generic with *one* type parameter that is required to extend the current interface as shown below (assuming that `ISpecial` is the current interface).

  ```
  abstract class SpecialStateMachine<StateType extends ISpecial>
      extends SuperStateMachine<StateType> { ... }
  ```

  The type parameter `StateType` must be used as argument to `SuperStateMachine`, such that the type parameter is the same for all super state machines of `SpecialStateMachine`, including the base state machine and the framework.

- If there are new events, all inherited states from `SuperStateMachine` must be extended such that they implement the new interface.
- Add any new states such that they extend `State` and implements the current interface.
- For each *new* state, add a factory method declared `abstract` in `SpecialStateMachine`.

Concrete state machines

In order to use a state machine, it has to be made concrete. A concrete state machine class denoted `ConcreteStateMachine`, is a subclass of an abstract state machine class denoted `AbstractStateMachine`. This is illustrated below, where the interface `I` denotes the interface that defines the state type of `ConcreteStateMachine`.

```
class ConcreteStateMachine extends AbstractStateMachine<I> { ... }
```

The only reason for creating concrete state machines is to implement the abstract factory methods of the abstract state machine to return the correct state references.

Figure 6.6 illustrates how to apply the extensible state machine pattern with the delegation framework. The figure assumes that a new interface is created for each state machine, but this is only necessary if new events are introduced.

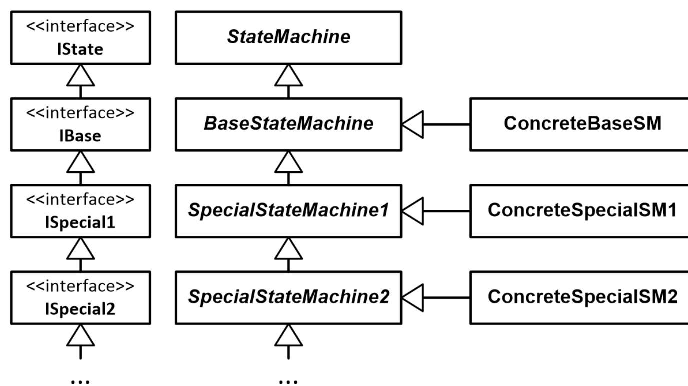


Figure 6.6: Specialized state machines using the extensible state machine pattern

6.4 Using inheritance and virtual classes

In this next approach, virtual classes are introduced to make the specialization of state machines very simple and elegant. Interestingly, the introduction of virtual classes removes the use of generics in the framework, and hence removes the need for interfaces. The use of factory methods will also be removed.

6.4.1 Virtual classes

A virtual class is an inner class that may be overridden in subclasses of the outer class. The type of a virtual class will, like with virtual methods, depend on the type of the outer class. However, unlike virtual methods, it is not possible to completely override a virtual class with a new type. In order to ensure that code inherited from superclasses of the outer class is valid, virtual classes is overridden by extending them such that previously defined functionality is kept, while having the ability to extend the classes with new functionality.

Subclasses of a virtual class will always inherit the class that is associated with the outer class. This means that overriding a virtual class in a given outer class will effect all the subclasses for that outer class. This feature is the reason why virtual classes is an interesting mechanism for specialization of state machines.

Figure 6.7 illustrates how virtual classes work by having a class `OuterA` and a subclass `OuterB`. `OuterA` defines the virtual class `Inner` with subclasses `InnerA` and `InnerB`. In `OuterB` the class `Inner` is overridden by being extended. In an instance of `OuterA`, the classes `InnerA` and `InnerB` will have the original superclass of `Inner`, while in an instance of `OuterB` the superclass will be the extended `Inner`.

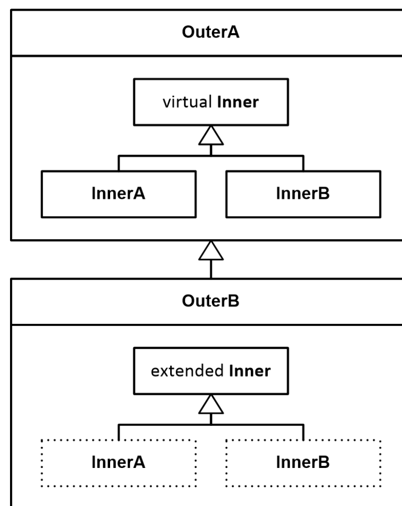


Figure 6.7: Virtual classes with subclasses

6.4.2 Java with virtual classes

Java has no support for virtual classes, nor is it possible to fully simulate them. For this section, the support for virtual classes is therefore assumed in Java. The definition of a virtual class must then be declared with the keyword `virtual` as shown below.

```
class OuterA {
    protected virtual class Inner { ... }
    ...
}
```

In order to override a virtual class, the new class must have the same name and extend the previous definition as shown below.

```
class OuterB extends OuterA {
    protected virtual class Inner extends OuterA.Inner { ... }
    ...
}
```

6.4.3 Modified delegation framework

Generics was implemented into the delegation framework such that states returned from certain methods (e.g. the method `currentState` in

StateMachine) gave the user access to the event methods of states without the need of unsafe type casting. Since all states are subclasses of State, the idea of the modified delegation framework is to make State virtual (see figure 6.8), and for each specialized state machine with new events, override State by extending it with new event methods. The states of the specialized state machines will then automatically inherit the extended State and hence inherit its events. Methods of the framework that was previously defined to return objects of the generic type parameter of StateMachine is now assumed to return objects of type State.

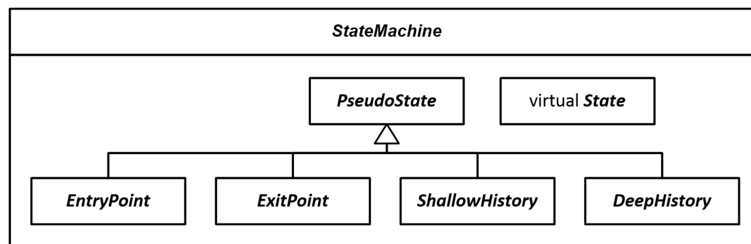


Figure 6.8: Modified delegation framework with virtual classes

6.4.4 Implementing the on/off switch

In order to implement the on/off switch, the class SwitchStateMachine extends StateMachine, see figure 6.9. State, which is inherited from StateMachine, is then overridden by extending it with the events powerOn and powerOff. State will now contain the functionality from the original State in addition to the events. Both event methods must contain code that simulates delegation such that subclasses of State only must override the events that is specific for them, and the other events will implicitly be delegated.[1] The states Switch, On and Off are defined as immediate subclasses of the extended State. The states are also declared virtual because it is then possible to override them in specialized state machines.

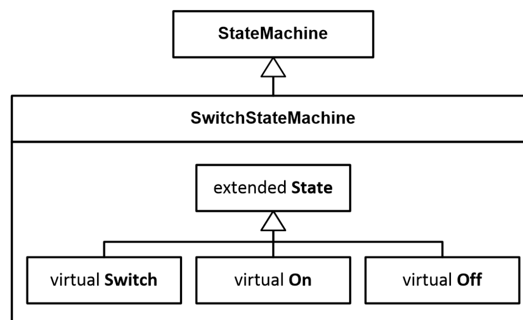


Figure 6.9: On/off switch with virtual classes

Listing 6.7 implements the on/off switch by using virtual classes.

Line 1 Defines the class SwitchStateMachine as a subclass of StateMachine.

[1] State will have the same role as MediaPlayerState in section 5.6.3 on page 58.

Lines 2-4 Defines the initial state of the state machine to be `Switch`.

Lines 5-16 The virtual class `State` is overridden to include the events `powerOn` and `powerOff`. The event methods contain code that simulates delegation. It is now necessary to have a general way of delegating since all states are subclasses of `State`, and therefore inherit the delegation code. The tests on line 7 and 12 is therefore needed in case the root state is trying to delegate to its enclosing state which is `null`. Notice that since the method `enclosingState` now returns a reference to an object of type `State`, the user gets access to the event methods `powerOn` and `powerOff` without unsafe type casting.

Lines 19-20 Dispatch methods for the events `powerOn` and `powerOff`.

Lines 23-36 The state classes which are subclasses of `State`. `Switch` defines its initial state to `Off`. `Off` overrides the event method `powerOn` with a transition to `On`. The event method `powerOff` is inherited from `State` and will therefore result in a delegation to the enclosing state `Switch`. `On` overrides the event method `powerOff` with a transition to `Off`. The event `powerOn` will be delegated to `Switch`.

Listing 6.7: `SwitchStateMachine`

```
1  class SwitchStateMachine extends StateMachine {
2    protected Class<? extends State> initialState() {
3      return Switch.class;
4    }
5    protected virtual class State extends StateMachine.State {
6      void powerOn() {
7        if (enclosingState() != null) {
8          enclosingState().powerOn();
9        }
10     }
11     void powerOff() {
12       if (enclosingState() != null) {
13         enclosingState().powerOff();
14       }
15     }
16   }
17
18   // State machine events
19   void powerOn() { currentState().powerOn(); }
20   void powerOff() { currentState().powerOff(); }
21
22   // States
23   protected virtual class Switch extends State {
24     Switch() { super(null); }
25     Class<? extends State> initialState() {
26       return Off.class;
27     }
28   }
29   protected virtual class Off extends State {
30     Off() { super(Switch.class); }
31     void powerOn() { changeToState(On.class); }
32   }
```

```
33   protected virtual class On extends State {
34     On() { super(Switch.class); }
35     void powerOff() { changeToState(Off.class); }
36   }
37 }
```

6.4.5 Implementing the specialized media switch

Inheritance is now used to define the class `MediaSwitchStateMachine` which is an extension of `SwitchStateMachine`, see figure 6.10. `State` is overridden by extending it with the new event `changeMode`. The states inherited from `SwitchStateMachine` will automatically inherit the extended `State`, and therefore inherit `changeMode` with code that simulates delegation. The states `CDMode` and `CassetteMode` is added as immediate subclasses of the extended `State`. `On` is overridden such that its initial state is set to `CDMode`. The state reference to `On` inherited from `SwitchStateMachine` will automatically be updated to reference to the overridden `On`.

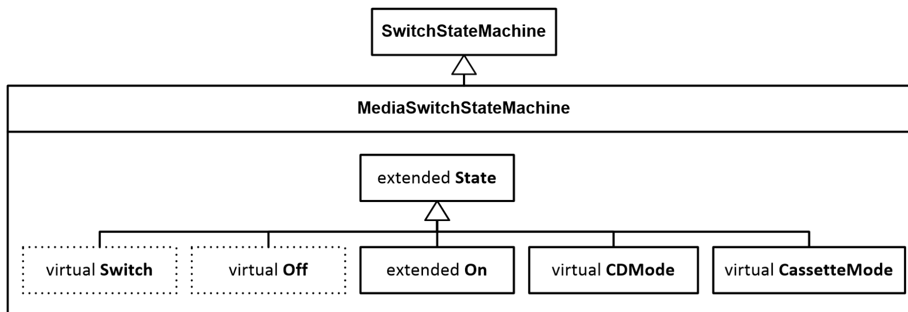


Figure 6.10: Media switch with virtual classes

Listing 6.8 implements the media switch with virtual classes.

Line 1 Using *inheritance* to extend `SwitchStateMachine`.

Lines 2-8 Overrides the virtual class `State` by extending it with the event method `changeMode`. The inherited classes `Switch`, `On` and `Off` will now support `changeMode` implicitly as they are subclasses of the new extended `State`.

Line 11 The dispatch method for the event `changeMode`. Since `State` is overridden and the method `currentState` returns an object of type `State`, it is possible to call `changeMode` on the current state without any type cast.

Lines 14-25 Adds the new virtual state classes `CDMode` and `CassetteMode` which are subclasses of `State`.

Lines 28-32 Since `On` becomes a composite state in the specialized media switch, the initial state must be defined. `On` is therefore overridden by extending it with the initial state definition. Recall that the

state reference to `On`, i.e. `On.class`, is referenced in the previously defined transition from `Off` to `On` (see lising 6.7). The transition is automatically updated because the class `On` is virtual and overridden.

Listing 6.8: `MediaSwitchStateMachine`

```
class MediaSwitchStateMachine extends SwitchStateMachine {
  protected virtual class State extends SwitchStateMachine.State
      {
    void changeMode() {
      if (enclosingState() != null) {
        enclosingState().changeMode();
      }
    }
  }

  // New state machine events
  void changeMode() { currentState().changeMode(); }

  // New states
  protected virtual class CDMode extends State {
    CDMode() { super(On.class); }
    void changeMode() {
      changeToState(CassetteMode.class);
    }
  }
  protected virtual class CassetteMode extends State {
    On() { super(On.class); }
    void changeMode() {
      changeToState(CDMode.class);
    }
  }

  // Override states
  protected virtual class On extends SwitchStateMachine.On {
    Class<? extends State> initialState() {
      return CDMode.class;
    }
  }
}
```

6.4.6 Summary of design principles

Below is a list of the general principles that is applied when implementing specialized state machines by using inheritance and virtual classes.

- Create a state machine class that extends another state machine. If the state machine class is the first specialization, i.e. the base state machine, then include the framework by extending `StateMachine`.
- If there are new events to the state machine, then override `State` by extending it with the event methods. Each event method must simulate delegation such that subclasses of `State` will delegate all events by default. Also implement the new event methods in the

state machine class such that the events gets dispatched to the current state. If desirable, the user may use interfaces that contain the event method signatures such that the state machine class and the states are guaranteed to implement all event methods.

- Implement new states as virtual classes that are subclasses of `State`. By defining states as virtual, the user is able to override states in specialized state machines such that previously defined state references will automatically be updated to the specialized state.

6.5 Summary

The extensible state machine pattern solved the problem of creating specialized state machines with the delegation framework. Even though the pattern solved the problem in a somewhat complex manner by using generics and factory methods, it has several positive sides to it.

- The language mechanisms used, like class inheritance and generics, are widely supported, e.g. in Java.
- The pattern is based on the state design pattern, which is a well-known design pattern.

However, even for the simplest specializations, like the media switch implemented in this chapter, the pattern require a fairly amount of work from the user. This leads to some negative sides.

- By using generics and factory methods as state references, state machine classes are forced to be declared as `abstract`. This means that there must be two state machine classes for each state machine, i.e. one abstract and one concrete.
- The *scalability* of the pattern is poor. When the number of states increases, the amount of work needed for creating specializations increases as well. The reason for this is that when new events are introduced in a specialized state machine, all state classes of the super state machine must be redefined in the specialized state machine in order to include the new events.
- There are a lot of constraints compared to the state design pattern. An important principle of a design pattern is *simplicity*, and the extensible state machine pattern breaks this principle to some degree, which decreases its usability.

The reason why the extensible state machine pattern is complicated to use is because of how inheritance and the type system works in Java. By introducing virtual classes, many of the weaknesses of the pattern are removed.

- The scalability is much better as the language makes sure that state classes are automatically updated with new events in specialized state machines.
- There are fewer design principles to follow, which improves the usability.

When using the delegation approach, there is no doubt that inheritance combined with virtual classes is the way to go when it comes to specialization of state machines. Again, Java is the only problem as it lacks support for virtual classes.

Chapter 7

Conclusion and future work

As there are no general purpose language or design pattern that directly support state machines with advanced mechanisms found in UML, it is recognized that combining the state design pattern with a state machine framework is a good solution, at least in order to support composite states, history, entry/exit actions and entry/exit points.

By using the principles of the state design pattern, where methods represent events with corresponding transitions, it is shown that representing state hierarchy by means of inheritance works for representing composite states in isolation, but when introducing advanced mechanisms like pseudo states, entry/exit actions etc., the solution becomes unsatisfactory. The inheritance approach is therefore countered by representing state hierarchy by delegation. At the same time as conserving the positive sides of the inheritance approach, e.g. supporting compound transitions, the problematic areas like inheritance of entry/exit actions are solved.

The extensible state machine pattern gives a solution on how to create specializations of simple state machines. It has been shown that the pattern can be applied to the delegation approach, giving the possibility of creating specialized state machines with the more advanced mechanisms supported by the state machine framework. By applying the idea of combining inheritance and virtual classes, the pattern goes from being quite demanding to use, to be a simple and maintainable pattern.

As the state machine framework only implements a subset of the functionality found in UML, there are a lot of subjects concerning state machines that are not investigated. Probably the most important mechanisms to investigate is regions. It would be interesting to see how regions, and especially orthogonal regions, can be represented in a programming language and supported by the framework. By supporting regions, it will also be interesting to see if the results of this thesis still apply.

This thesis shows that there are good *indications* that modeling and programming with state machines will be best supported in a language that supports delegation, inheritance and inner, virtual classes. Composite states are supported by delegation, while the combination of inheritance and virtual classes support the specialization of state machines. While there are languages that support delegation (see e.g. Self [19]), and languages

that support inheritance and virtual classes (see e.g. Beta [10]), none of the languages support all three language mechanisms. It is therefore natural to think of extending a popular language like Java with these mechanisms, or design a new language from scratch with support for these three mechanisms.

Appendix A

Framework source code

The source code of the framework implemented in this thesis is available for download: http://folk.uio.no/kjetand/framework.zip. In order to run the examples, the files and directories must be unzipped.

A.1 Directories and source files

inheritance/ This directory contains the Java framework implemented with *inheritance* representing state hierarchy. Also contains an example of the media player state machine.

delegation/ This directory contains the Java framework implemented with *delegation* representing state hierarchy. Also contains an example of the media player state machine, and an example of the specialized media switch.

A.2 Running the examples

Compile the source files with `javac`. E.g. in linux terminal:

1. `javac inheritance/*.java`
2. `javac delegation/*.java`.

The class `Main` in both directories runs the examples. Run with the program `java`. E.g. in linux terminal:

1. `cd inheritance/` and `java Main`
2. `cd delegation/` and `java Main`

Bibliography

[1] Brian Chin and Todd D. Millstein. An extensible state machine pattern for interactive applications. In *ECOOP*, pages 566–591, 2008.

[2] Ole-Johan Dahl. *SIMULA 67 Common Base Language, (Norwegian Computing Center. Publication).* 1968.

[3] Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3 edition, 2003.

[4] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*, pages 305–313. Addison-Wesley Professional, 1st edition, 1994.

[5] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley Longman Publishing Co., Inc., 1983.

[6] James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification.* Addison-Wesley Longman Publishing Co., Inc., 1st edition, 1996.

[7] Morten Olav Hansen. Exploration of uml state machine implementations in java. Master's thesis, University of Oslo, 2011.

[8] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[9] Øystein Haugen, Birger Møller-Pedersen, and Thomas Weigert. Uml for real. chapter Structural Modeling with UML 2.0: Classes, Interactions and State Machines, pages 53–76. Kluwer Academic Publishers, 2003.

[10] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-oriented programming in the BETA programming language.* Addison-Wesley, 1993.

[11] Henry Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. *ACM SIGPLAN Notices*, 21(11):214–223, 1986.

[12] Ole Lehrmann Madsen. Towards integration of state machines and object-oriented languages. *Proceedings of the Technology of Object-Oriented Languages and Systems*, pages 261–274, 1999.

[13] Ole Lehrmann Madsen and Birger Møller-Pedersen. Virtual classes: A powerful mechanism in object-oriented programming. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '89, pages 397–406, New York, NY, USA, 1989. ACM.

[14] Ole Lehrmann Madsen and Birger Møller-Pedersen. A unified approach to modeling and programming. In *Proceedings of the 13th international conference on Model driven engineering languages and systems: Part I*, MODELS'10, pages 1–15. Springer-Verlag, 2010.

[15] Birger Møller-Pedersen and Dagbjørn Nogva. Scalable and object oriented sdl state(chart)s. In Jianping Wu, Samuel T. Chanson, and Qiang Gao, editors, *FORTE*, volume 156 of *IFIP Conference Proceedings*, pages 59–73. Kluwer, 1999.

[16] OMG. *OMG Unified Modeling Language (tm) (OMG UML), Superstructure*. OMG, 2.2 edition, February 2, 2009.

[17] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The (2Nd Edition)*. Pearson Higher Education, 2004.

[18] Asher Sterkin. State-oriented programming. In *Multiparadigm Programming with Object-Oriented languages*, 2008.

[19] David Ungar and Randall B. Smith. Self: The power of simplicity. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '87, pages 227–242, New York, NY, USA, 1987. ACM.