COPYRIGHT © 2012 Stefan Trapp

Dieses Manuskript ist urheberrechtlich geschützt. Es darf ohne Genehmigung des Urhebers nicht verwertet werden. Insbesondere darf es nicht ganz oder teilweise oder in Auszügen abgeschrieben oder in sonstiger Weise vervielfältigt werden.

Ein modellgetriebener Ansatz zur Implementierung von Zustandsmaschinen in objektorientierten Programmiersprachen

Stefan Trapp

Palmerstraße 31
20535 Hamburg

stefan.trapp@stefan-trapp-consulting.de
www.stefan-trapp-consulting.de

1 Motivation

Objekte oder Systeme mit einem komplexen zustandsabhängigen Verhalten sind in den Ingenieurwissenschaften ein häufig auftretendes Phänomen. Die naive Implementierung in Software führt zu umfangreichen, unübersichtlichen und fehlerträchtigen `switch` oder `if-else` Blöcken. Das Zustandsmuster (*State Pattern*) der „*Gang of Four*“ ist ein häufig verwendeter Ansatz zur Vermeidung dieser Problematik, der allerdings auch seine Limitierungen hat.

Hier soll ein anderer Ansatz vorgestellt werden: Eine generische Zustandsmaschine, deren „Kernimplementierung“ zur Lösung konkreter Aufgabenstellungen (wieder-) verwendet wird. Zusammen mit einer in XML zu modellierenden Zustandsbeschreibung und der jeweils benötigten zustandsabhängigen Funktionalität (siehe Abschnitt 2) ergibt sich eine „ausführbare Zustandsmaschine“. Dabei kann das von der generischen Zustandsmaschine lesbare XML-Modell mit einem modellgetriebenen (*model driven development – mdd*) Ansatz aus einem *Unified Modeling Language* (UML) Modell generiert werden.

2 Zustandsmodellierung

Zustandsmaschinen (*Finite State Machines, FSM*) sind ein oft genutztes mathematisches Modell beim Design von Systemen oder Software. Das Modell der Zustandsmaschine abstrahiert das zu modellierende Objekt auf eine endliche Anzahl von Zuständen (*States*) und Transitionen (Zustandsübergänge, *Transitions*), die die Zustände miteinander verknüpfen. Im Lebenszyklus des Systems ist immer jeweils genau ein Zustand aktiv (*Current State*). Durch Ereignisse (*Triggering Events*) kann eine Transition, also der Übergang in einen anderen Zustand, ausgelöst werden. Weitere mögliche Elemente einer Transition sind

- ein „Wächterausdruck“ (*Guard*), der eine Bedingung definiert, die erfüllt sein muss, damit die Transition ausgeführt wird, sowie
- eine Aktion (oder Effekt, *Action/Effect*), die beim Durchlaufen der Transition ausgeführt wird.

Auf Grund der großen Bedeutung der Zustandsmodellierung ist das Zustandsdiagramm eine der vierzehn Diagrammarten in der Modellierungssprache UML.

3 Entwurfsziele

Dem hier vorgestellten Design liegen folgende Entwurfsziele zugrunde:

- Definition einer wiederverwendbaren (generischen) Zustandsmaschine, die mit gängigen Programmiersprachen (z. B. Java oder C#) implementiert werden kann.
- Mit der generischen FSM sollen anschließend vielerlei konkrete Problemstellungen, die für eine Zustandsmodellierung geeignet sind, schneller, einfacher und besser gelöst werden können.
- Die Definition einer spezifischen FSM zur Lösung eines konkreten Entwurfsproblems soll in Form einer XML Konfigurationsdatei (Zustandsmodell) erfolgen.
- Möglichkeit zur Modellierung der jeweils konkreten Zustandsmaschine durch UML Zustandsdiagramm(e) mit einem UML-Modellierungstool (z. B. SparxSystems Enterprise Architect, http://www.sparxsystems.com/).
- Automatische Generierung der spezifischen FSM-Definition, beispielsweise durch Konvertierung aus dem XMI-Format des UML-Modellierungstools in das durch die generische FSM lesbare XML-Format.

- Mögliche Validierung des XML-basierten Zustandsmodells durch ein XML-Schema (XSD).
- Möglichkeit zur (teilweise) automatischen Code-Generierung für die generische FSM aus dem XML-Schema.
- Bei Bedarf kann aus dem XML-Zustandsmodell einer etwaig bereits vorhandenen konkreten FSM das zugehörige XML-Schema unter Verwendung eines Schema Generators automatisch generiert werden.

4 Design der Zustandsmaschine

In diesem Kapitel wird das Design der FSM erläutert. In Sektion 4.1 wird gezeigt, welche Elemente und Schritte zur Erzeugung der generischen FSM sowie von darauf basierenden, ausführbaren (spezifischen) Zustandsmaschinen notwendig sind. In Abschnitt 4.2 wird anhand eines Beispiels die Transformation eines Zustandsmodells vom UML Zustandsdiagramm in die – durch die generische FSM ausführbare – XML-Beschreibung erläutert. In Abschnitt 4.3 wird schließlich ein Klassendiagramm der generischen FSM vorgestellt.

4.1 Elemente und Schritte zur Erzeugung von generischer und ausführbarer Zustandsmaschine

Abbildung 1 zeigt notwendige und optionale Schritte zu Erzeugung von generischer und ausführbarer FSM.

Das einfachste Szenario zur Erzeugung einer ausführbaren FSM ist folgendes:

1. Manuelle Implementierung der generischen FSM.

Anschließend sind zur Nutzung der generischen FSM für eine konkrete Problemstellung jeweils folgende Schritte durchzuführen:

2. Manuelle Erzeugung der spezifischen XML-Zustandsbeschreibung, die von der generischen FSM ausgeführt wird.
3. Manuelle Implementierung der in der XML-Zustandsbeschreibung definierten *Actions* und *Guards*.

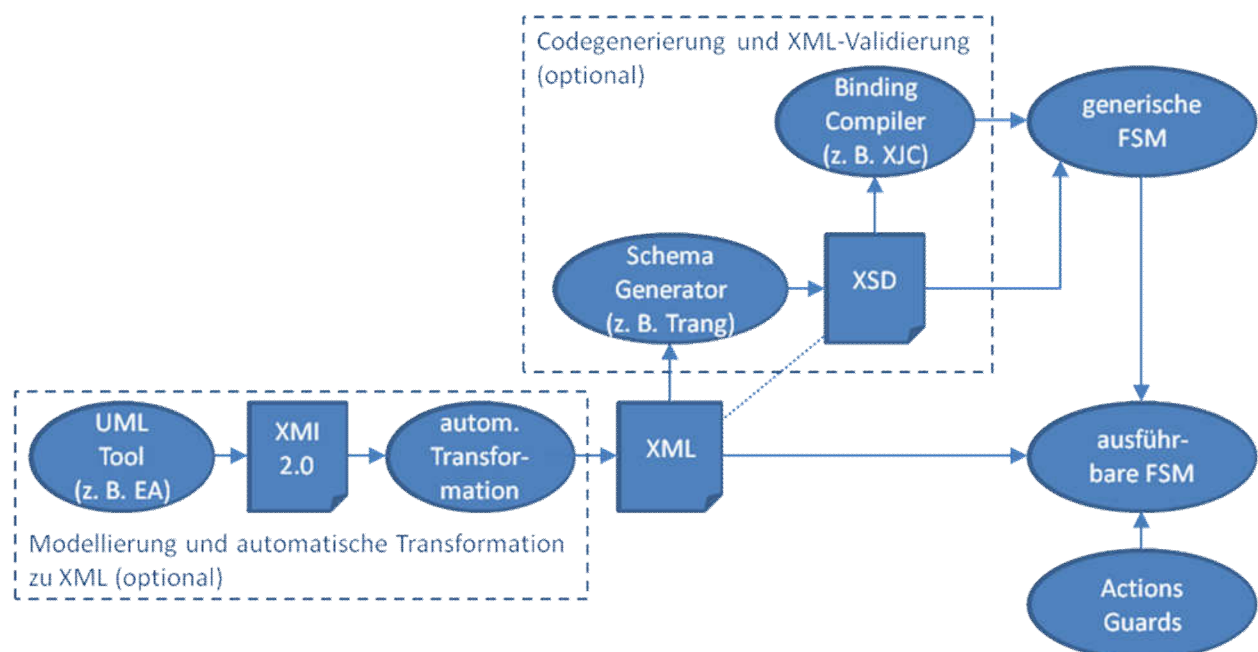

Abbildung 1: Erzeugung von generischer und ausführbarer FSM

Dieses Szenario kann durch folgende Optionen ergänzt werden:

1. Bei der Implementierung der generischen FSM kann teilweise automatische Codegenerierung benutzt werden.
    a. Aus einem manuell fertiggestellten XML-Schema (XSD) für die Zustandsmodelle können mit einem Binding Compiler (z. B. „XJC“ aus der JAXB – Java Architecture for XML Binding) einige Klassen der generischen FSM automatisiert erzeugt werden.
    b. Sollte zuerst eine XML-Zustandsbeschreibung manuell erzeugt worden sein, kann ein zugehöriges XML-Schema mit einem entsprechenden Generator (z. B. „Trang“, https://code.google.com/p/jing-trang/) kreiert werden.
2. Die generische FSM kann das XML-Schema zur Validation „konkreter“ XML-Zustandsmodelle benutzen.
3. Die Modellierung einer spezifischen Zustandsmaschine kann mit Hilfe eines Modellierungstools für UML erleichtert werden, z. B. SparxSystems Enterprise Architect (EA). Wird ein entsprechender Übersetzer implementiert (der Block „automatische Transformation“ in Abbildung 1), kann die XML-Erzeugung anschließend für sämtliche Modelle automatisch erfolgen. Dieser Übersetzer kann beispielsweise einen XMI-Export (*XML Metadata Interchange*) aus dem Modellierungstool als Eingangsformat für die Transformation verwenden.

4.2 Zustandsdiagramm und XML-Definition einer ausführbaren FSM

Dem UML Zustandsdiagramm in Abbildung 2 entspricht die nachfolgende XML-Definition, die durch die generische FSM eingelesen und ausgeführt werden kann.

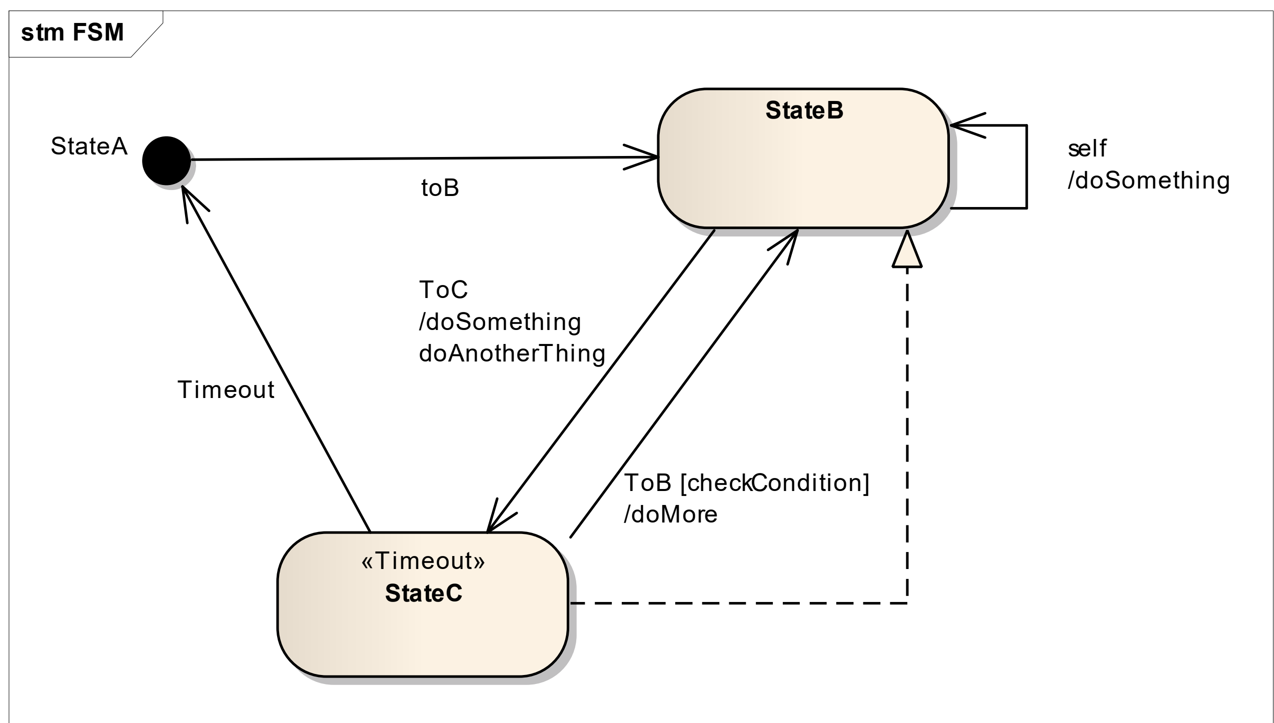

Abbildung 2: UML Zustandsdiagramm (Beispiel)

```xml
<?xml version="1.0" encoding="utf-8" ?>
<FSM xmlns:gxr="http://stc/FsmSchema.xsd">
  <State name="StateA" type="Initial">
    <Transition event="toB" toState="StateB" />
  </State>
  <State name="StateB">
    <Transition event="ToC" toState="StateC" action="doSomething
      doAnotherThing" />
    <Transition event="self" toState="StateB" action="doSomething" />
  </State>
  <State name="StateC" timeout="5000" parents="StateB">
    <Transition event="Timeout" toState="StateA" />
    <Transition event="ToB" toState="StateB" guard="checkCondition"
      action="doMore" />
  </State>
</FSM>
```

Auf einige Features der generischen FSM, die auch aus diesem Beispiel ersichtlich sind, soll besonders hingewiesen werden:

Vererbung zwischen Zuständen

Ein Zustand kann von einem anderen Zustand oder auch mehreren anderen Zuständen erben (im Beispiel erbt „StateC" von „StateB"). Dies ist im UML Zustandsdiagramm durch eine „*realize*" Beziehung der Zustände modellierbar. In diesem Fall übernimmt der *Child-State* sämtliche Transitionen der *Parent-States*. *Self-Transitions* der *Parent-States* werden dabei auch beim *Child-State* zu *Self-Transitions*, d. h. geerbte *Self-Transitions* zeigen nicht auf den jeweiligen *Parent-*, sondern auf den *Child-State*.

Zusammengesetzte Aktionen und Guards

Es können bei Bedarf mehrere Aktionen in einer Transition ausgeführt werden (z. B. `action="doSomething doAnotherThing"`). Dies führt zu granularen Teil-Aktionen, die besser wiederverwendet werden können. Dieses Vorgehen ist bei Bedarf analog auch für *Guards* möglich.

Timeout-States

Zustände können optional einen *Timeout* definieren, nach dessen Ablauf eine spezielle Transition mit dem Namen „Timeout" ausgeführt wird. Im Beispiel hat Zustand „StateC" einen Timeout von 5000 Millisekunden. (Dieser Wert ist auch im UML-Modell für den Zustand „StateC" hinterlegt, im Diagramm aber nicht sichtbar.) *Timeout-States* müssen die Transition „Timeout" besitzen.

4.3 Klassendiagramm der generischen Zustandsmaschine

Abbildung 3 zeigt ein vereinfachtes Klassendiagramm der generischen Zustandsmaschine. Die Klasse `StateMachine` referenziert – neben einer Warteschlange für *Events* und dem aus dem XML-Modell durch Datenbindung (*XML data binding*) erzeugten `(I)StateModel` – die Interfaces `IActions` und `IGuards`. Bei diesen letzteren beiden Interfaces handelt es sich lediglich um Markierungsschnittstellen (*tag interfaces*). Die im `StateModel` beschriebenen spezifischen *Actions* und *Guards* werden, wenn sie durch die `StateMachine` ausgeführt werden sollen, durch Reflexion/Introspektion (*reflection*) des Methodennamens ermittelt, d. h. jeder *Action* bzw. jedem *Guard* im `StateModel` entspricht eine Methode gleichen Namens in den implementierenden Klassen `SpecificActions` bzw. `SpecificGuards`.

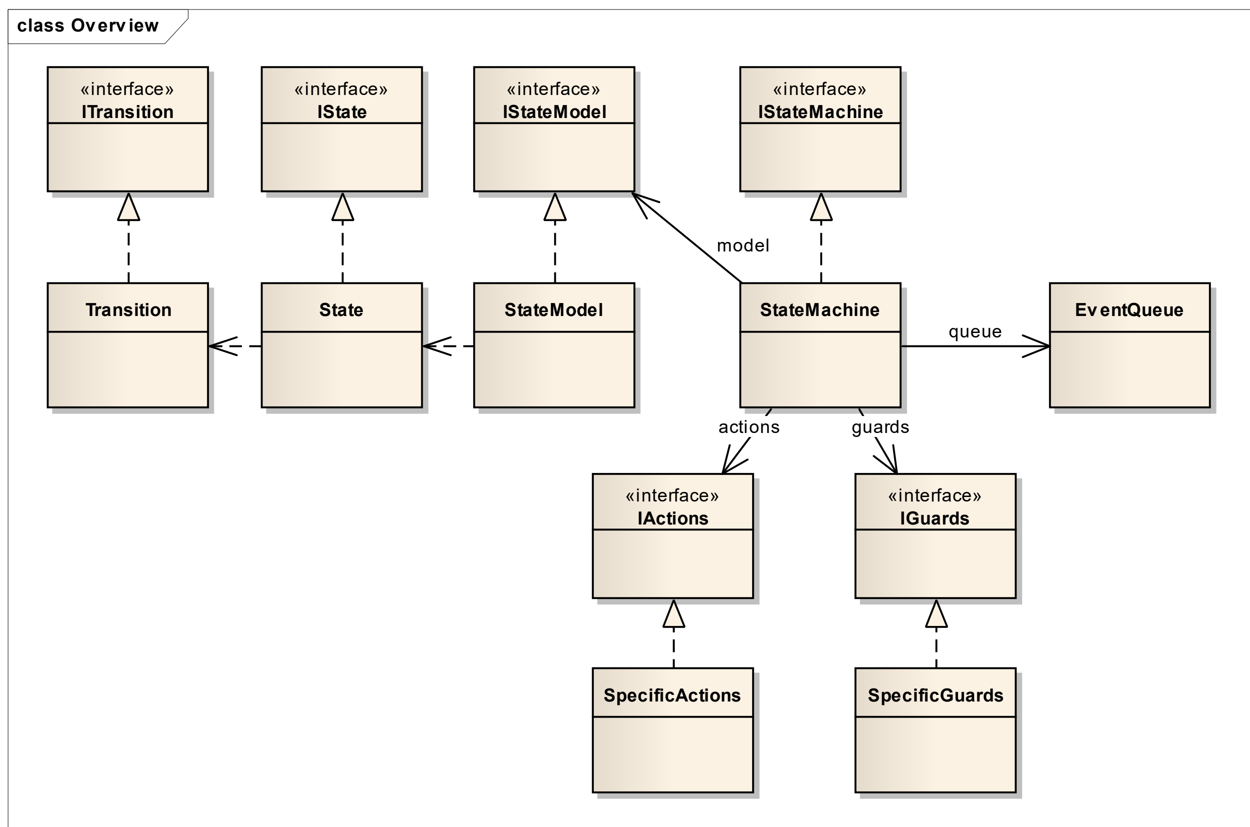

Abbildung 3: Vereinfachtes Klassendiagramm der generischen Zustandsmaschine

Nach der Instanziierung befindet sich die Zustandsmaschine in ihrem initialen Zustand und „wartet“ auf das Eintreffen von *Events* in der `EventQueue`. Sobald ein *Event* in die Warteschlange eingestellt wurde, beginnt die FSM mit dessen Auswertung, d. h. der Prüfung auf eine zugehörige Transition im aktiven Zustand. Wird eine entsprechende Transition gefunden, erfolgt die Auswertung von deren *Guard* (wenn vorhanden). Ist die durch den *Guard* definierte Bedingung nicht erfüllt, wird diese Transition nicht ausgeführt und stattdessen werden die Transitionen des aktiven Zustands weiter nach „Kandidaten“ für das *Event* durchsucht. Ist dagegen die Bedingung des *Guards* erfüllt (oder kein *Guard* vorhanden), erfolgt die Ausführung der *Action(s)* (wenn vorhanden) und schließlich der Wechsel in den neuen Zustand. Anschließend wartet die FSM entweder auf das Eintreffen weiterer *Events* oder fährt sequenziell mit der Auswertung bereits in der Warteschlange befindlicher *Events* fort.

Einige Details sind in dieser Übersicht nicht dargestellt, etwa

- die Instanziierung der Zustandsmaschine (z. B. durch eine *Factory*),
- die Wandlung des XML-Zustandsmodells in das `StateModel`, d. h. in Objekte der verwendeten Programmiersprache (XML-Datenbindung, z. B. mit JAXB),
- die Validierung des XML-Zustandsmodells, z. B. mit Hilfe eines XML-Schemas oder
- weitere Maßnahmen zur Prüfung der syntaktischen Korrektheit eines Zustandsmodells, z. B. ein „Abgleich“ des XML-Zustandsmodells mit den implementierten *Actions* und *Guards*.

5 Fazit

Eine Implementierung des hier gezeigten Designs lohnt sich insbesondere, wenn beim Entwurf eines Systems oder einer Software-Plattform komplizierte Zustandsmodelle zum Einsatz kommen sollen. In diesem Fall bietet die Lösung folgende Vorteile:

- Hoher Grad an Wiederverwendung
- Übersichtlichkeit auch bei komplizierten Zustandsmodellen
- Zustandsmodellierung in UML
- Durchgängigkeit vom Modell zur Implementierung durch automatisierte Generierung des XML-Zustandsmodells
- Möglichkeit zur Prüfung der syntaktischen Korrektheit von Zustandsmodellen

Andererseits erfordert die Implementierung der generischen Zustandsmaschine einen höheren Anfangsaufwand als alternativ denkbare Lösungen.

Das vorgestellte Design wurde – mit geringen plattformspezifischen Unterschieden – bereits in Java und .NET erfolgreich um- und eingesetzt.

6 Literatur

Balzert, Heide: Lehrbuch der Objektmodellierung – Analyse und Entwurf mit der UML 2, Spektrum Akademischer Verlag, 2. Auflage 2004, ISBN 978-3827429032

Gamma, Erich; Helm, Richard; Johnson, Ralph E.; Vlissides, John: Design Patterns. Elements of Reusable Object-Oriented Software, Addison-Wesley Longman, 1. Auflage 1994, ISBN 978-0201633610