The State Machine Compiler

by
Eitan Suez

UptoData, Inc.
http://u2d.com/
About the Speaker

- Eitan Suez is a Java programmer living and working in Austin, Texas

- Eitan is primarily known as the author of *ashkelon*, an open source tool for Java API documentation

- Eitan is an active member of the *Austin Java Users Group*

- Eitan maintains a weblog on [http://java.net/](http://java.net/)
Primary Purpose

1. In General:

To learn how Finite State Machines (FSMs) are useful for modeling stateful objects in a software system

2. Specifically:

To learn to implement such objects in a clean and agile way using the State Machine Compiler
A. Uncle Bob’s Turnstile

B. Finite State Machines Basics

C. What is SMC?

D. Solving Turnstile using SMC

E. SMC In Detail

F. Examples
A simple turnstile

Description of the normal operation of a turnstile:

1. Turnstile initially locked
2. A coin or ticket is inserted (event)
3. Triggers action: turnstile unlocks
4. A person passes through turnstile (event)
5. Triggers action: turnstile locks again
State Diagram

Locked

Unlocked

pass/lock()

coin/unlock()
Coding the turnstile

(First pass)

break into hands-on session
Discussion

Most implementations don’t think to separate actions into neat little packages:

alarm / thankyou / unlock / lock

So, the first improvement is to go ahead and perform this packaging:

1. put the code for each case in a separate method (the action)
2. gather the collection of actions into an interface

This helps with action code getting in the way of seeing the transition logic
More Issues

- Difficult to clearly interpret conditional logic. It’s difficult to read; you can get lost in it. It’s error-prone.

- Violates the Open-Closed Principle (OCP). Extending the system requires modifying existing code.
Let’s try the State Pattern

(second pass)

break into hands-on session
The State Pattern

- Replace conditional logic by using polymorphism
- The context class becomes much simpler; it delegates transitions handling to its current state.
- The code is cleaner.
If you need to extend the system, say you need to introduce a new state:

- you add a new class
- you don’t have to touch existing code

The state pattern respects the OCP:

*software entities should be open for extension but closed for modification*
1. The state logic is distributed across a number of classes, and so there’s no single place to see it all

2. Tedious to write
The state of affairs

- Employing the state pattern is usually as far as most people go

- State diagrams are typically used only passively, in our designs, and to help us understand the state logic

Let’s go back to our diagram and discuss some Finite State Machine (FSM) basics.
Agenda

A. Uncle Bob’s Turnstile

B. Finite State Machines Basics

C. What is SMC?

D. Solving Turnstile using SMC

E. SMC In Detail

F. Examples
Basic Terminology

States, Transitions, and Actions

Locked

Unlocked

States

Locked

Unlocked

Transitions

Actions

pass/lock()

coin/unlock()

coin/thankyou()

pass/alarm()
Other Concepts

- **Entry and Exit Actions**
  
  *Actions performed every time we enter and exit a state, respectively*

- **Transition Guards**
  
  *Conditions that must be met in order for transitions to proceed*
Transition Guard Example

Form Entry:

- Fill out a form (in "Edit" state)

- The "Submit" event (or transition) essentially contains a guard condition.

- If the form was not completed correctly (invalid), then we will remain in edit mode and have to make corrections.

- Conversely, if the guard condition is true (the form is valid), then we will proceed with transition to "Read" state/mode.
Transition Tables

- A common mechanism for describing transition diagrams clearly in text form
- For each state, we write out:
  - the transition
  - what the next state will be
  - what action to perform (if any)
### Transition table for turnstile

<table>
<thead>
<tr>
<th>State</th>
<th>Transition</th>
<th>Next State</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Locked</td>
<td>coin</td>
<td>Unlocked</td>
<td>unlock</td>
</tr>
<tr>
<td></td>
<td>pass</td>
<td>Locked</td>
<td>alarm</td>
</tr>
<tr>
<td>Unlocked</td>
<td>coin</td>
<td>Unlocked</td>
<td>thankyou</td>
</tr>
<tr>
<td></td>
<td>pass</td>
<td>Locked</td>
<td>lock</td>
</tr>
</tbody>
</table>
A. *Uncle Bob’s Turnstile*

B. Finite State Machines Basics

C. *What is SMC?*

D. Solving Turnstile using SMC

E. SMC In Detail

F. Examples
SMC is a tool that mates FSMs with Objects
SMC

(continued)

- An open-source project hosted on sourceforge.net


- Written by and maintained Charles Rapp
Essentially:

“you put your state diagram in one file using an easy-to-understand language. SMC generates the State Pattern classes for you.”
SMC History

- SMC is Robert Martin’s invention (it is discussed in Robert’s book *Agile Software Development* (Ch 29))

- Charles Rapp happened to have succeeded Robert at Clear Communications Corporation. He added many features, made design revisions, and open-sourced the project (more information in the preface of the SMC manual on sourceforge).
Agenda

A. *Uncle Bob’s Turnstile*

B. Finite State Machines Basics

C. What is SMC?

D. *Solving Turnstile using SMC*

E. SMC In Detail

F. Examples
Where were we?

Ah yes.. Issues with the State Pattern

1. The state logic is distributed across a number of classes, and so there’s no single place to see it all

2. Tedious to write

(Notice: there’s nothing wrong with the design from the point of view of the runtime implementation)
The .sm file

One place to see it all

%class Turnstile
%package turnstile

%start MainMap::Locked

%map MainMap
%%
Locked
{
  coin Unlocked { unlock(); }
  pass nil { alarm(); }
}
Unlocked
{
  pass Locked { lock(); }
  coin nil { thankyou(); }
}
%%
The Compiler

Generates the State Pattern code

```
java -jar Smc.jar -java -d turnstile Turnstile.sm
```

The SMC Compiler

Specify java language output

Optionally specify target directory

Specify the .sm file to process
A single file is produced, but it contains many classes:

- `TurnstileContext$MainMap.class`
- `TurnstileContext$MainMap_Default$MainMap_Locked.class`
- `TurnstileContext$MainMap_Default$MainMap_Unlocked.class`
- `TurnstileContext$MainMap_Default.class`
- `TurnstileContext$TurnstileState.class`
- `TurnstileContext.class`
Write the *AppClass*

```java
public class Turnstile implements TurnstileActions {
    TurnstileContext _fsm;
    TurnstileActions _actions;

    public Turnstile(TurnstileActions actions) {
        _fsm = new TurnstileContext(this);
        _actions = actions;
    }

    public void coin() { _fsm.coin(); }
    public void pass() { _fsm.pass(); }
    public void alarm() { _actions.alarm(); }
    public void lock() { _actions.lock(); }
    public void thankyou() { _actions.thankyou(); }
    public void unlock() { _actions.unlock(); }
}
```

- Define and instantiate “fsm” context class (generated)
- Expose transition calls to other parts of application, if necessary
- Implement (or delegate) actions
The term “AppClass” in SMC refers to a class you write that is designated to interact with the Context class that SMC generates.

The actions you specify are assumed to be methods defined in the AppClass.
Pictorially...

Turnstile.sm → SMC Compiler → TurnstileContext.java
<uses>  <uses>

Turnstile.java (the AppClass)
The context class invokes actions on the AppClass. Conversely, the AppClass invokes transitions on the Context Class.
Steps Review

1. Write the state diagram (.sm file)

2. Run the SMC tool (generates state pattern code)

3. Implement the actions (write the AppClass)

4. Interact with FSM by invoking transition methods
Agenda

A. *Uncle Bob’s Turnstile*
B. Finite State Machines Basics
C. What is SMC?
D. Solving Turnstile using SMC
E. *SMC In Detail*
F. Examples
SMC In Detail

1. Tool mechanics and project setup
2. SMC features in detail
3. SMC's design
1. Tool mechanics and project setup
Project Setup

- Separate directory **gen** for generated source code
- **gen** directory structure mirrors src structure
- **bin** directory contains the smc tool (Smc.jar)
- Ant target automatically reruns SMC when .sm file changes
- Include statemap.jar in runtime classpath
The Smc.jar command

```
java -jar Smc.jar -{targetlanguage}
  {options} {smfilename}.sm
```

Target languages:
- c++, java, tcl, vb, csharp
- table (generates HTML table representation of the .sm file)
- graph (generates a GraphViz .dot file diagram of the state machine logic)
Smc.jar command options

```
java -jar Smc.jar -{targetlanguage}
  {options} {smfilename}.sm
```

Options:

- `-suffix` (override output file name suffix)
- `-sync` (e.g. for Java: add synchronized keyword to transition method declarations)
- `-d` (specify output directory)
- `-serial` (generate serial IDs for states to allow persisting of states)
- `-g` (add debug output messages to generated source code)
Smc.jar command options

(continued)

```
java -jar Smc.jar -{targetlanguage}
   {options} {smfilename}.sm
```

Options (continued):

- `glevel` (applies only to -graphviz output: specify level of detail in the generation of .dot file diagram)
- `version` (prints smc version)
- `help` (prints help)
- `verbose` (generate verbose output during compilation phase)
Example

```java
java -jar Smc.jar -graph
   -glevel 1 Turnstile.sm
```

Produces `Turnstile_sm.dot`, which in Graphviz, looks like this:
<taskdef name="smc" classname="net.sf.smc.ant.SmcJarWrapper"
    classpath="lib/smc-ant.jar" />

<target name="gen" depends="init">
    <smc target="java" smfile="${smfile}"
        destdir="${gen.pkg.dir}"
        smcjar="${smc.jar}" />
</target>

<target name="compile" depends="gen">
    <javac debug="on" deprecation="on"
        classpathref="class.path"
        destdir="${build.classes.dir}">
        <src path="${src.dir}" />
        <src path="${gen.dir}" />
    </javac>
</target>
Important note about naming

In Java, it is possible to create a file for a class whose name is different from the declared class name (in the file)

In SMC:

- the name of the file is derived from the name of the .sm file
- the class name (in the file) is derived from the AppClass name

Therefore:

*make sure to always use the same name for both .sm file and AppClass*
2. SMC features in detail
Basic .sm file syntax

%class Turnstile
%package turnstile

%start MainMap::Locked
%map MainMap
%
Locked
{
  coin Unlocked { unlock(); }
  pass nil      { alarm(); }
}
Unlocked
{
  pass Locked { lock(); }
  coin nil    { thankyou(); }
}
%

The AppClass
Package name
The start state
States are grouped into Maps
Demarcates start of map
Essentially a transition table
Demarcates end of map
Simple Transition

<table>
<thead>
<tr>
<th>Transition</th>
<th>Next State</th>
<th>Actions</th>
</tr>
</thead>
</table>
| /* smc recognizes c-style comments */  
  // ..as well as c++ style comments |
| Idle       | Running    | {}      |
|            | Run        |         |

```c
Idle
{
   Run          Running                {}
}
```
Loopback Transition

Transition Table:

<table>
<thead>
<tr>
<th>Transition</th>
<th>Next State</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Idle</td>
<td>nil</td>
<td>{}</td>
</tr>
<tr>
<td>or</td>
<td>Idle</td>
<td>{}</td>
</tr>
</tbody>
</table>

Diagram:

```
Idle
Timeout
Idle
```

50
Transition with Actions

<table>
<thead>
<tr>
<th>Transition</th>
<th>Next State</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Idle</td>
<td>Running</td>
<td>{ StopTimer(&quot;Idle&quot;); DoWork(); }</td>
</tr>
<tr>
<td>{ Run</td>
<td>Running</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>
Transition Guards

Transition Guards

<table>
<thead>
<tr>
<th>Transition</th>
<th>Next State</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Idle</td>
<td>Running</td>
<td></td>
</tr>
<tr>
<td>{}</td>
<td>Running</td>
<td></td>
</tr>
<tr>
<td>Run [ ctxt.isValid() ]</td>
<td>Running</td>
<td></td>
</tr>
<tr>
<td>Run</td>
<td>nil</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Running</td>
<td></td>
</tr>
<tr>
<td>Idle</td>
<td>Running</td>
<td></td>
</tr>
<tr>
<td>{}</td>
<td>Running</td>
<td></td>
</tr>
<tr>
<td>Run</td>
<td>nil</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Running</td>
<td></td>
</tr>
</tbody>
</table>

Transition Guards

- Transition Guards
- Transition
- Next State
- Actions

Idle

{ Run [ ctxt.isValid() ]

Running

{ StopTimer("Idle"); DoWork();

}
Notes on transition guards

- Guard condition must evaluate to a boolean. Guard may contain ||, &&, comparison operators (>, ==, etc..) or it can be a method invocation.

- If guard condition is a method invocation on the AppClass, then prefix invocation with "ctxt."
  
  e.g.: `ctxt.isValid()`

- Transitions with guards have higher precedence than unguarded transitions.

- There's also a default / fallback transition mechanism that we'll discuss shortly.
LoggedOut
{
  onLogin(username: String, password: String)
    [ctxt.isLocked(username)]
    nil
    { displayLockedDlg(); }

  onLogin(username: String, password: String)
    [ctxt.authenticate(username, password)]
    LoggedIn
    { setupUser(username); }
  ..
}
### Entry & Exit Actions

**Logged In**

- **Entry**: `dismissLogin()`
- **Exit**: `clearUser()`

#### Transition Table

<table>
<thead>
<tr>
<th>Transition</th>
<th>Next State</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>LoggedIn</td>
<td>LoggedOut</td>
<td><code>{ dismissLoginDialog(); }</code> <code>{ clearUser(); }</code></td>
</tr>
<tr>
<td>Entry</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>{ onLogout</td>
<td>LoggedOut</td>
<td><code>{ }</code></td>
</tr>
</tbody>
</table>

---

55

---
SMC allows the definition of multiple maps. Each map should contain related states.

The idea is that you can use push and pop transitions to move across maps.
Push Transition Example

```
Running
{
  Blocked   push(WaitMap::Blocked)   {GetResource();}
}

"GetResource()" is invoked and the state changes to "Blocked," defined in WaitMap. SMC pushes (remembers) the existing state on the state stack (internal).
```
The "Granted" transition causes the state stack to pop and thus to revert to the state that the FSM was in prior to the push. The "OK" transition then takes place in that state.
The Default State

- You may define a state named "Default" and specify transitions for it.

- These transitions serve as "fallback" transitions for all the other states.

- That is, if you omit defining a transition in a certain state and that transition takes place, then SMC will fall back to calling the Default state's transition (assuming it's defined there)
Can define default behavior of turnstile in "Default" state and override default behavior (exceptions) in Locked and Unlocked states, like so:

```ruby
Locked
{
  pass nil { alarm(); }
}
Unlocked
{
  coin nil { thankyou(); }
}
Default
{
  coin Unlocked { unlock(); }
  pass Locked { lock(); }
}
```
Default Transitions

Yet another level of fallback can be defined as the "Default" transition within a state.

If a state S does not define a transition T and the "Default" state does not define one either, then T will be handled by S's "Default" transition.

Default transitions can be defined on the Default state
3. SMC's design
The SMC Pattern

TurnstileContext

AppClassContext

(state code, in statemap.jar)

MainMap

MainMap_Default

MainMap_Unlocked

MainMap_Locked

ConcreteStateA

ConcreteStateB

State

AppClassState

MapDefaultState

The SMC Pattern
Agenda

A. Uncle Bob’s Turnstile
B. Finite State Machines Basics
C. What is SMC?
D. Solving Turnstile using SMC
E. SMC In Detail

F. Examples
Where do FSMs apply in software development?
Generic GUI Uses / Examples

Generally, anything modal:

- Application Login Protocol
- Wizards (follows a series of steps from start to finish) and Paging
- Associating by picking from a list
  - list can be in "pickable" state, where it exposes a "pick" transition, that triggers an action that performs the association of the selected item with some other object
- GUI manifestation of the object life cycle: Create / Read / Update / Delete
  - e.g.: in Edit state, the object exposes "Save" and "Cancel" transitions. In Read state, it might expose "Edit" and "Delete" transitions.
LoggedInState
Entry { dismissLoginDialog(); }
Exit { clearUser(); }

onLogout LoggedOutState {}

LoggedOutState
Entry { showLoginDialog(); }

onLogin(username: String, password: String)
    [ctxt.isLocked(username)]
        nil { displayLockedDialog(); }
    [ctxt.authenticate(username, password)]
        LoggedInState { clearBadAttempts(username); setupUser(username); }
    [ctxt.tooManyBadAttempts(username)]
        nil { lock(username); displayLockedDialog(); }
    onLogin(username: String, password: String) nil { loginInvalid(); }
Another login implementation

```xml
<?xml version="1.0"?>
<fsm actions-interface="login.LoginActions">
  <state name="loggedout" startstate="true">
    <transition name="Log In"
               next-state="authenticatingUser" action="authenticate" />
  </state>
  <state name="authenticatingUser" transitory="true">
    <transition name="passedAuthentication"
               next-state="loggedin" action="login" />
    <transition name="failedAuthentication"
               next-state="loggedout" action="nil" />
  </state>
  <state name="loggedin">
    <transition name="Log Out"
               next-state="loggedout" action="showLoginDlg" />
  </state>
</fsm>
```
Think about GUI apps such as Photoshop or the GIMP, where each tool you pick puts you into a different mode (brush, eraser, selection tool, etc..)

Further, within each mode, you can have sub-modes. For example, the "selection" tool in Photoshop is modal. The first click which starts a selection has a different behavior compared to the second, which completes it. The UI reflects this: after the first click (transition), we're enter selection mode (state). In this mode onmousemove (transition) triggers the action to redraw a dashed rectangle, but stay in that mode. Another transition, "onclick" will take it back out of selection mode. The exit action is to record the selection.
Enabling and disabling GUI features in an application based upon whether those features are applicable (valid transitions) in the existing application state.

Example: in Keynote, unless I pick some text, the text inspector remains disabled
Network code

- General: Disconnected, Connecting, Connected, Disconnecting states
- File transfer: connect, send data, disconnect
SMC is great for finding patterns in character streams

Detecting tokens such as words, numbers, whether we're inside comments, etc..

As the tokenizer reads the text stream, it enters and leaves different states (inside comment, outside comment) and fires transitions (detected token)
A Comment Stripper

NormalState
{
    slash PerhapsEntering {  }
    asterisk nil { outputit(); }  
    other nil { outputit(); }  
}

PerhapsEntering
{
    slash nil { outputit(); }  
    asterisk InComment { }  
    other NormalState { outputwithslash(); }  
}

%%

PerhapsLeaving
{
    slash NormalState { }  
    asterisk nil {}  
    other InComment { }  
}

InComment
{
    slash nil {}  
    asterisk PerhapsLeaving {}  
    other nil {}  
}
Reader reader = new FileReader(filename);
char ch;
int read = reader.read();
while (read != -1) {
    ch = (char) read;
    _char = ch;
    switch (ch) {
        case '*': {
            _fsm.asterisk();
            break;
        }
        case '/': {
            _fsm.slash();
            break;
        }
        default: {
            _fsm.other();
        }
    }
    read = reader.read();
}
Anything with a user interface

- Turnstiles, Coke machines, Gumball machines, ATMs
Business Objects

- A Visit can be scheduled, confirmed, in progress, or complete
- A Bill can be outstanding, partially paid, or paid
- An Order can be placed, partially filled, filled, shipped
Wrapping up..
Current Status of Project

- SMC project is mature / stable
- Current version is v3.2
  (latest addition is the production of Graphviz diagrams from a .sm file)
- Expect ant task to be bundled with the next release
Conclusions

- SMC is a simple tool that can be used repeatedly to solve a specific category of problems that occur frequently in various domains in software development.

- SMC offers a "best of both worlds" solution to Finite State Machine problems:
  1. The design advantages of the State Pattern without the tedium of writing the state classes
  2. View and edit the entire finite state machine logic in a single file
References

1. Design Patterns
   by Gamma, Helm, Johnson, & Vlissides

2. Agile Software Development
   by Robert Martin

3. SMC Project
   http://smc.sourceforge.net/
   (maintained by Charles Rapp)
Contact

Eitan Suez
eitan@u2d.com
UptoData, Inc.
http://u2d.com/

Please remember to complete evaluation forms.