Java Finite State Machine Framework

1. Requirements
   • JDK 1.4.x (http://java.sun.com/j2se/1.4.2/download.html)

2. Overview

Java Finite State Machine Framework contains classes that define FSM meta-model, allows to manipulate with model, compile model, validate model and execute model.

Framework separates model of concrete FSM – model static structure - and rules of interpreting of this structure (operational semantics). Typically process of framework usage consists of the following steps:
1. Somehow create FSM model – **Design Phase**;
2. Create implementation classes for EventProviders and ControlledObjects – **Development Phase**;
3. Create new (or configure existing) FSM model interpreter - **Development** or **Deployment Phase**;
4. Execute FSM model using created interpreter – **Application Life Phase**.

There three different views of FSM model:
   • in-memory FSM model – supported by Java Finite State Machine Framework;
   • FSM model *XML* description – supported by Java Finite State Machine Framework;
   • FSM model graphically represented with a help of Class and Statechart diagrams – supported by **Eclipse Plug-in**;

There are several ways to create in-memory FSM model:
   • programmatically on Java;
   • create FSM model *XML* decryption and then transform it into in-memory FSM model using transformer;
   • using Eclipse plug-in create graphical FSM model, export it into *XML* description, transform *XML* description into in-memory model.

See [Core Classes JavaDocs](#) for more details on framework classes and transformers utilities.

3. Finite State Machine Meta-Model

Fig. 1 below shows FSM meta-model. White classes represent design-time elements that are...
not intended to be subclassed. Blue classes represent runtime elements that must be subclassed. Complete reference on runtime classes will be given in next section that describes Runtime Framework. In this section they are mentioned just to show relationships between design time classes and runtime classes.

FSM Metamodel

Figure 1. FSM Metamodel

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
<th>Relationships</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>FSM Model entry point. Acts as container for model components and as factory for Model, StateMachine, EventProviderHandlers, ControlledObjectHandlers.</td>
<td>Aggregates StateMachines, EventProviderHandlers and ControlledObjectHandlers. Knows about root StateMachine.</td>
</tr>
<tr>
<td>ClassElement</td>
<td>Base abstract implementation for model elements that may have associations. Also implements delegate to ImplementationHandler.</td>
<td>Aggregates outgoing and incoming Associations.</td>
</tr>
<tr>
<td>ImplementationHandler</td>
<td>Utility class. Handles implementation class name during design time, lazily instantiate implementation class and store it in transient field during runtime.</td>
<td></td>
</tr>
<tr>
<td>StateMachine</td>
<td>Acts as container for States and Transitions and as factory for Association, State, Transition, Event, Action, Guard, Parameter.</td>
<td>Aggregates top State. Implements delegate to ConfigStore – runtime class that will be discussed later.</td>
</tr>
<tr>
<td>EventProviderHandler</td>
<td>Acts as handler for runtime implementation of EventProvider. EventProvider role in model is to provide root StateMachine with Events.</td>
<td></td>
</tr>
<tr>
<td>ControlledObjectHandler</td>
<td>Acts as handler to runtime implementation of ControllableObject. ControllableObject role is to store Actions implementations.</td>
<td></td>
</tr>
</tbody>
</table>
### States

FSM State.  
Has type. Aggregates included States, included StateMachines, inout Actions that must be executed on-enter, incoming and outgoing Transitions.

### Transitions

Transition between States.  
Has trigger Event, Guard condition and list of Actions that must be executed if this transition trigs.

### Actions

FSM input and output action.  
Stores action identifier. Has similar to UML Common Behaviour.Procedure meaning – represents reference to ControlledObject method.

Directly knows about ControlledObjectHandler that in runtime will know about ControllableObject that must contains method associated with Action. For example, consider Action that stores identifier o1.z1 and is associated with ControlledObject that stores handle to class test.TestCO1. In runtime, if such Action will be executed, method z1() of class test.TestCO1 will be called.

### Events

Event that trigs Transition.  
In runtime may has associated Parameters.

### Guards

Transition guard condition.  
Initialized with Boolean expression. Syntax and semantics of Boolean expression is described below.  
After compilation, Guard has reference to Abstract Syntax Tree (AST) of Boolean expression.

---

**Table 1: FSM Metamodel Classes Description**

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>State</td>
<td>FSM State. Has type. Aggregates included States, included StateMachines, inout Actions that must be executed on-enter, incoming and outgoing Transitions.</td>
</tr>
<tr>
<td>Transition</td>
<td>Transition between States. Has trigger Event, Guard condition and list of Actions that must be executed if this transition trigs.</td>
</tr>
<tr>
<td>Action</td>
<td>FSM input and output action. Stores action identifier. Has similar to UML Common Behaviour.Procedure meaning – represents reference to ControlledObject method. Directly knows about ControlledObjectHandler that in runtime will know about ControllableObject that must contains method associated with Action. For example, consider Action that stores identifier o1.z1 and is associated with ControlledObject that stores handle to class test.TestCO1. In runtime, if such Action will be executed, method z1() of class test.TestCO1 will be called.</td>
</tr>
<tr>
<td>Event</td>
<td>Event that trigs Transition. In runtime may has associated Parameters.</td>
</tr>
<tr>
<td>Guard</td>
<td>Transition guard condition. Initialized with Boolean expression. Syntax and semantics of Boolean expression is described below. After compilation, Guard has reference to Abstract Syntax Tree (AST) of Boolean expression.</td>
</tr>
</tbody>
</table>

---

**3.1. Guard Condition Boolean Expression Syntax and Semantics**

Well-formed Boolean expression may contain:

- constants `true` and `false`;
- input actions identifiers;
- logical operators `!` (not), `&&` (and), `||` (or);
- relation operators `>`, `<`, `!=`, `==`, `>=`, `<=`.

---

*Copyright © 2003 eVelopers Corporation All rights reserved.*
Formal BNF grammar definition for well-formed Boolean expression is:

```
S -> S || I1 | I1
I1 -> I1 && I2 | I2
I2 -> !I3 | I3
I3 -> (S) | I4
I4 -> I5 rel I5 | IB | CB
I5 -> IN | CN
```

Where

- **IB** – Action identifier that corresponds to method with boolean return type,
- **CB** – Boolean constant true or false,
- **IN** – Action identifier that corresponds to method with int return type,
- **CN** – natural number constant.

Correct Boolean expressions examples:

- !o1.x1 && o2.x2 > 10
- (o1.x1 || o1.x2 || o2.x2 != 10) && o2.x10
- true && o1.x2

**Semantics** of Boolean expression or **interpretation rules** are the following: after compilation, Boolean expression is parsed into AST, leaves of AST are action identifiers and constants; AST value is calculated using depth first search; to calculate action identifier value, corresponding method of corresponding ControlledObject is being called.

For example, to calculate value of Boolean expression !o1.x1 && o2.x2 > 10 the following steps us needed to be done:

1. calculate o2.x2: call method x2() of ControlledObject with identifier o2;
2. calculate o2.x2 > 10 using already calculated value of o2.x2;
3. calculate o1.x2: call method x2() of ControlledObject with identifier o2;
4. calculate !o1.x1 using already calculated value of o1.x1;
5. calculate !o1.x1 && o2.x2 > 10 using already calculated value for expression parts !o1.x1 and o2.x2 > 10.

### 3.2. How to Create In-Memory FSM Model Programmatically

Consider you have to programmatically create model that is shown on fig. 2.

**Figure 2. Sample FSM Graphical Model**
Resulting Java code will look like:

```java
// create model
Model m = Model.createModel("test");
// create state machine A1
StateMachine A1 = m.createStateMachine("A1");
// create event provider and associate it with state machine A1
A1.createAssociation(m.createEventProviderHandler("p1", "test.Provider1"), "p1");
// create controlled object and associate it with state machine A1
// using identifier "o1"
A1.createAssociation(m.createControlledObjectHandler("o1", "test.ControllableObject1"), "o1");
// create top state and set it as top for state machine A1
State top = A1.createTopState("TOP");
A1.setTop(top);
// create other states
State initial1 = A1.createState("initial1", StateType.INITIAL);
State s2 = A1.createState("s2", StateType.NORMAL);
State initial2 = A1.createState("initial1", StateType.INITIAL);
State s4 = A1.createState("s4", StateType.NORMAL);
// add on-enter actions for state s4
s4.addOnEnterAction(A1.createAction("o1.z2"));
State final1 = A1.createAction("final1", StateType.FINAL);
// setup child-parent relations for states
top.addSubstate(initial1);
top.addSubstate(s2);
top.addSubstate(final1);
s2.addSubstate(initial2);
s2.addSubstate(s4);

// create transitions between states
Transition t1 = A1.createTransition(initial1, s2, null, null);
t1.addOutputAction(A1.createAction("o1.z1"));
Transition t2 = A1.createTransition(initial2, s4, null, null);
Transition t3 = A1.createTransition(s4, s4, A1.createGuard("o1.x1"), A1.createEvent("e1"));
Transition t4 = A1.createTransition(s4, final1, A1.createGuard("!o1.x1"), A1.createEvent("e1"));
```

If you want to pass this model to interpreter, you have to compile model with a help of the following code:

```java
StateMachineCompiler smc = new StateMachineCompiler(null);
DefaultCompilationListener dcl = new DefaultCompilationListener();
smc.addCompilationListener(dcl);
smc.compileWithIncluded(A1);
```

The only initialization parameter of StateMachineCompiler is OperationResolver.
interface should be implemented in case when actions associated with methods of ControlledObject could be resolved. There is no any default implementation of this interface in framework, but there is one in Eclipse plug-in. If you pass implementation of this interface to StateMachineCompiler, it will make additional semantic checks.

To see compilation errors use code:

```java
for (int i = 0; i < dcl.getErrors().length; i++) {
    System.out.println(dcl.getErrors()[i]);
}
```

Actually compilation process does the following:

- checks that every model element has name;
- checks that ControlledObjectHandlers and EventProviderHandlers have not empty implementation class;
- checks that every State has StateType;
- parses and translates to AST all Guards Boolean expressions;
- parses and translates Actions identifiers to pair (ControlledObject reference, method name).

During compilation the following errors may be found:

- model element has empty name;
- EventProviderHandler or ControlledObjectHandler has empty implementation class;
- empty state type;
- incorrect Action identifier;
- incorrect Guard Boolean expression syntax;
- unresolved reference to ControlledObject in Action identifier;
- unresolved reference to method of ControllableObject in Action identifier.

After compilation model is ready for interpreting, but also it may be checked for semantics errors using model validation facilities.

### 3.3. FSM Model Validation

FSM model validation facilities allow finding errors in model during **Design Phase**. Some validation methods may be called “model static verification”.

You may validate in-memory FSM model programmatically using code:

```java
// init validator
StateMachineValidator validator = StateMachineValidator.createValidator(A1);
// create default implementation of validation errors listener
ValidationMessenger validationMessenger = new ValidationMessenger();
```
// setup default validation error listener
validator.addStructureListener(validationMessenger);
validator.addAttainabilityListener(validationMessenger);
validator.addConsistencyListener(validationMessenger);
validator.addCompletenessListener(validationMessenger);

// validate structure
validator.validateStructure();
// validate states attainability
validator.validateAttainability();
// validate transitions consistency
validator.validateConsistency();
// validate transitions completeness
validator.validateCompleteness();

FSM XML-description may be checked with command line validation tool:
java -jar UniMod-Core-XX.XX.XXX.jar A1.xml

During model validation the following errors may be found:
• composite state has no initial state;
• composite state has more then one initial state;
• state is unattainable;
• transition set for given event is incomplete;
• two transitions are inconsistent;
• state has more then one outgoing “else” transition;
• transition from initial state has event on it;
• not root state machine has associated EventProviderHandlers.

3.4. Transformers
In-memory FSM model may be serialized into XML file, XML file may be de-serialized into in-memory model. For storing model in XML file statemachine.dtd is used.

To support model serialization to XML and de-serialization from XML special XML Transformers classes exist. The following code demonstrates how store model in XML:

```java
// create file
FileOutputStream f = new FileOutputStream("model.xml");
// write model to file
ModelToXML.write(model, f);
// close file
f.close();
```

Next code shows how to load model from XML:

```java
// open file
FileInputStream f = new FileInputStream("model.xml");
```
Also, there is Transformer for generating C++ code for Symbian platform (coming soon).

4. Runtime Framework

After creating and compiling in-memory FSM model, this model may be executed using Runtime Framework. Fig. 3. shows classes of runtime framework. Blue classes are intended to be subclassed by client. Red classes are entry point classes.

Runtime Framework

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
<th>Relationships</th>
</tr>
</thead>
<tbody>
<tr>
<td>StateMachineEngine</td>
<td>Entry point for runtime engine. Implements factory for itself. Must be initialized with FSM model, Units, type of execution thread model. ConfigStore implementation class name also may be passed as initialization parameter. Details on ConfigStore and StateMachineEngine lifecycle see below.</td>
<td>Has EventProcessor that responsible for Event execution.</td>
</tr>
<tr>
<td>EventHandler</td>
<td>This interface may be gotten from StateMachineEngine instance. It's used to post Events to engine.</td>
<td></td>
</tr>
<tr>
<td>EventProcessor</td>
<td>Subclasses of this abstract class responsible for performing Event processing inside one StateMachine. For processing given Event it is passed with active State and StateMachineContext.</td>
<td>Has reference to StateMachine</td>
</tr>
<tr>
<td>EventProcessorFactory</td>
<td>Responsible for creating itself and EventProcessor implementation.</td>
<td></td>
</tr>
<tr>
<td>Unit</td>
<td>Implementation of Unit</td>
<td></td>
</tr>
</tbody>
</table>
interface will be notified about steps during Event processing. It may be used for logging, for example. SimpleLogger implementation exists in package com.evelopers.unimod.runtime.eventProcessorListenertProcessorListenertProcessorListener. Find Unit contract below.

<table>
<thead>
<tr>
<th>ExceptionHandler</th>
<th>Implementation of this interface must be notified about Exceptions that occurs during Event processing. Add implementation to StateMachineEngine using method addExceptionHandler()</th>
</tr>
</thead>
<tbody>
<tr>
<td>EventProvider</td>
<td>Implementations of this interface is lazily created by associated EventProviderHandler. StateMachineEngine inits EventProviders on startup. Implementation class responsible for providing Events to StateMachineEngine throught EventHandler interface. Details about EventProvider contract see below</td>
</tr>
<tr>
<td>ControlledObject</td>
<td>Implementations of this interface is lazily created by associated ControlledObjectHandler. Implementation responsible for providing methods associated with input and output Actions. Details about ControlledObject implementation see below</td>
</tr>
<tr>
<td>ConfigStore</td>
<td>StateMachine may has associated ConfigStore implementation. ConfigStore responsible for storing and loading StateMachine Config. Details about ConfigStore</td>
</tr>
</tbody>
</table>

Copyright © 2003 eVelopers Corporation All rights reserved.
<table>
<thead>
<tr>
<th><strong>Table 1: Runtime Framework Classes Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Config</strong></td>
</tr>
<tr>
<td><strong>StateMachineContext</strong></td>
</tr>
</tbody>
</table>

**4.1. StateMachineEngine Lifecycle**

1. On startup engine calls `EventProvider.init()` for all event providers associated with root StateMachine, setup EventHandler implementation depending of chosen thread model, set given ConfigStore to root StateMachine if root StateMachine has empty ConfigStore;
2. On Event post engine calls EventProcessor implementation. EventProcessor do the following:
   1. Loads current StateMachine Config using associated ConfigStore. If loaded Config is null, then Initial State of Top State is treated as active State;
   2. Gets outgoing Transitions from active State that has Events equals to posted Event; looks through all selected Transitions and calculates Guard Boolean expressions; stops when calculated Boolean expression is true; Transition that owns such Boolean expression is triggered;
   3. Executes output Actions on Transition that is being trigged, executes on-enter Actions in target State;
   4. If target State is composite, continues processing from step 2.2;
   5. If target State is not composite, executes included StateMachines starting from step 2.1. Note, that if included StateMachine has empty ConfigStore, parent ConfigStore will be used: engine will ask Config of parent StateMachine to create sub-Config for
included StateMachine and the whole parent Config with sub-Configs will be stored when all included StateMachines finish event processing;
6. Saves new StateMachine Config using associated ConfigStore;
3. If root StateMachine active State has type `StateType.FINAL`, calls `EventProvider.dispose()` for all event providers associated with root StateMachine and stops accepting Events.

<table>
<thead>
<tr>
<th>Note:</th>
</tr>
</thead>
<tbody>
<tr>
<td>StateMachine in-memory model passed to engine defines “class” (do not confuse with Java class) of StateMachine, but not instance. Instance of StateMachine is defined by pair (StateMachine, Config), because Config defines active State. Such approach allows using the same in-memory model for different clients, which may be very useful in multi-user applications.</td>
</tr>
</tbody>
</table>

4.2. Execution Thread Model

Two execution thread models are supported by runtime engine:

- **Strict.** Thread that comes with Event is used for event processing. Caller of method `EventHandler.handle()` is blocked until the end of Event processing. If more then one Event will be posted concurrently, their thread will not be joined, so there will be more then one thread processing Event against the same engine. It’s not bad and rather good for developing Web application that works inside Servlet container. See Messenger Web for more details;
- **Queued.** Event queue exists. Event dispatcher thread checks for new Event in queue and if there is one – starts its processing. It’s guaranteed that Events are dispatched from queue one-by-one, so in any time only one Event is being processed against same engine. Caller thread is not blocked. It’s not allowed to change in-memory model after StateMachineEngine created.

<table>
<thead>
<tr>
<th>Note:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event dispatcher thread in Queued thread model is daemon thread, so if you do not start any non daemon threads in your EventProviders, application will exit (see <a href="http://java.sun.com/j2se/1.4.2/docs/api/java/lang/Thread.html">http://java.sun.com/j2se/1.4.2/docs/api/java/lang/Thread.html</a> for more details on daemon and non daemon threads).</td>
</tr>
</tbody>
</table>

4.3. EventProcessorFactory and EventProcessor Contracts

Runtime engine is delivered with default implementations of EventProcessorFactory and EventProcessor classes. If you want to create custom implementation, follow next rules.

EventProcessorFactory subclass must has public no args constructor.

EventProcessorFactory has public static method `create()` that is used by StateMachineEngine to obtain EventProcessorFactory instance. Method `create()` tries to find name of class that subclasses EventProcessorFactory using algorithm:
1. get system property (System.getProperty()) with name
   com.evelopers.unimod.runtime.EventProcessorImplFactory;
2. if not found, tries to load properties from file unimod.properties using method
   EventProcessorFactory.class.getClassLoader().getResourceAsStream()
   and get property with name
   com.evelopers.unimod.runtime.EventProcessorFactory from loaded
   bundle;
3. if not found, creates default implementation
   com.evelopers.unimod.runtime.impl1.EventProcessorImplFactory1;

Subclass of EventProcessorFactory must implement the only method that must return
instance of EventProcessor:

```
public EventProcessor newExecutor(StateMachine sm) {...}
```

EventProcessor subclass must implement the only method:

```
protected abstract State process(
    Config config,
    State activeState,
    Event event,
    StateMachineContext context)
    throws EventProcessorException, InterpreterException;
```

### 4.4. ConfigStore Contract

ConfigStore subclass must have public no args constructor.

ConfigStore subclass must implement methods for storing and loading Config. StateMachineContext is passed to ConfigStore methods, so StateMachineContext may be
used as underlining Config storage. For example, interpreter for Web Applications associates
User Context with HttpSession, it allows to store Config user HttpSession.

Config may be stored in database. It’s useful when you model behavior of some business
entity such as Order, for example. Consider that Order has some behavior that is described
using UniMod methodology, so you can implement ConfigStore that will store Order state in
database, but all business logic will be implemented on Java. Such approach suits for
applications with thin database – database that has no stored procedures that controls entities
behavior.

### 4.5. ControlledObject Contract

ControlledObject subclass must have public no args constructor. Methods that correspond to
input Actions must have signatures like this:

```java
/**@unimod.action.descr is initialization successful */
public boolean xM(StateMachineContext ctx)
    throws MyException1, MeException2 {...}

/**@unimod.action.descr number of logged-in users */
public int xM(StateMachineContext ctx)
    throws MyException1, MeException2 {...}
```

Methods that correspond to output Actions must have signature:

```java
/**@unimod.action.descr makes some action */
public void zK(StateMachineContext ctx)
    throws MyException1, MyException2 {...}
```

JavaDoc tag `@unimod.action.descr` allows to define action description, that will be shown on Connectivity Diagram. All exceptions thrown by methods of ControlledObject will be passed to registered ExceptionHandlers method `handleException()`.

### 4.6. Unit Contract

Unit subclass must implement the only method:

```java
public void handle(
    final Position p,
    final Config config,
    final Event event,
    final State processedState,
    final Transition processedTransition,
    final Action processedAction);
```

This method will be called by EventProcessor during Event processing in places that are described by class `Position`.

### 4.7. EventProvider Contract

EventProvider subclass must have public no args constructor.

EventProvider subclass must start **none daemon thread** in `EventProvider.init()` method to notify EventHandler about new Events. It's not permitted to hang in `EventProvider.init()`. In `EventProvider.dispose()` methods thread must be stopped. Such technique allows `JVM` to exit when root StateMachine comes to Final State, because on-enter to Final State StateMachineEngine calls `EventProvider.dispose()` method.
for all event providers.

To post Event EventProvider must pass implementation of StateMachineContext interface to EventHandler, this implementation will be delegated to all methods of ControllableObjects that will be executed during Event processing, it means that StateMachineContext acts as **shared data bus**.

EventProvider subclass may define Events that it throws with a help of code snippet:

```java
/**@unimod.event.descr timer tick */
public static final String EN = "eN";
```

Special JavaDoc tag `@unimod.event.descr` allows to define Event description, that will be shown on *Connectivity Diagram*.

### 4.8. StateMachineContext Contract

StateMachineEngine guarantees that there are no concurrent calls to StateMachineContext during Event processing inside this particular engine, but if you create more then one instance of StateMachineEngine that will operate concurrently and will pass them the same StateMachineContext for Event processing, there is no guarantee that there will be no concurrent calls to StateMachineContext.

There is another example of satiation, when concurrent calls are possible. Consider you’ve created engine with **Strict** thread model, but your EventProvider post Events in different threads. Strict thread model doesn’t join Event threads, so concurrent threads against the same engine will take place and, consequently, there will be concurrent calls to StateMachineContext.

StateMachineContext consists of three contexts:

- **Event context** – shared data bus that must “live” only during event processing;
- **User context** – shared data bus that must be associated with application user session (for Web applications it is HttpSession);
- **Application context** – shared data bus that must “live” while application alive.

Developers are free to implement StateMachineContext as they want, but we strongly recommend to follow the rules described above.

**Note:**

As you can see, Runtime Framework is rather complicated thing. To help you using it we’ve developed two ready to use interpreters for Standalone and Web applications. Next sections describe them.
5. Interpreter for Standalone Applications

Interpreter for standalone application is implemented using Runtime Framework and packaged as `/lib/UniMod-Adapter-Standalone-XX.XX.XXX.jar`. To start interpreter, create FSM model XML-description and pass it as parameter for interpreter:

```
java –jar UniMod-Adapter-Standalone-XX.XX.XXX.jar A1.xml
```

**Note:**
Do not forget to setup classpath, so interpreter found your ControlledObjects and EventProviders.

Standalone interpreter implements Runtime Framework in the following way:

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread model</td>
<td>Queued</td>
</tr>
<tr>
<td>Startup</td>
<td>On startup interpreter converts FSM model XML-description into in-memory model and creates StateMachineEngine. Also on startup interpreter throws Event with name “e0” and with parameters arg0..argN – command line arguments, so in methods of your ControlledObjects you may get command line parameters using call <code>StateMachineContext.getEventContext().getParameter(&quot;arg0&quot;)</code></td>
</tr>
<tr>
<td>ConfigStore for root StateMachine if it has no one</td>
<td>In-memory</td>
</tr>
<tr>
<td>StateMachineContext</td>
<td>There is StateMachineContextImpl, that can be used in EventProviders</td>
</tr>
<tr>
<td>Units</td>
<td>SimpleLogger</td>
</tr>
<tr>
<td>ExceptionHandler</td>
<td>Simply logs Exception stack trace.</td>
</tr>
</tbody>
</table>

**Table 1: Standalone Interpreter Features**

6. Interpreter for Web Applications

Interpreter for web application is implemented using Runtime Framework and Java Servlet 2.3 specification. It’s packaged as `/lib/UniMod-Adapter-Servlet-XX.XX.XXX.jar`.

Interpreter is implemented as Servlet, which must be deployed into some Servlet 2.3 container with FSM model XML-description and classes for EventProviders,
ControlledObjects and ConfigStores.

Interpreter HttpSession do the following:

1. In HttpSession.init() method it parses ServletConfig and creates StateMachineEngine with Strict thread model. See table below for possible Servlet init parameters;
2. In HttpSession.service():
   1. creates StateMachineContext implementation associated with HttpServletRequest, HttpSession and ServletContext;
   2. creates Event with name from HttpServletRequest parameter evt;
   3. calls EventHandler.handle() passing created StateMachineContext and Event to it.

As you can see, Interpreter Servlet acts as EventProvider, because it provides runtime engine with Events from HttpServletRequest, i.e. from web application client.

There are two predefined implementations of ConfigStore that may be used with interpreter for Web applications:

- com.evelopers.unimod.adapter.servlet.AppContextConfigStore stores Config in AppContext (i.e. in associated ServletContext);
- com.evelopers.unimod.adapter.servlet.UserContextConfigStore stores Config in UserContext (i.e. in associated HttpSession).

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Mandatory</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATE_MACHINE_NAME</td>
<td>Name of FSM model XML-description file without &quot;.xml&quot; extension. Example: A1</td>
<td>Yes</td>
</tr>
<tr>
<td>UNIT_CLASSx (where x is number)</td>
<td>Unit class name that will be passed to runtime engine. Unit class must has no args constructor.</td>
<td>No</td>
</tr>
<tr>
<td>LOGGER_NAME</td>
<td>Logger name. If set to &quot;CONSOLE&quot;, System.out will be used for logging, otherwise Log4j logger with given name will be used.</td>
<td>No</td>
</tr>
<tr>
<td>CALLBACK_CLASS</td>
<td>Name of class that implements ExceptionHandler. If not defined – default implementation will be used.</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 1: Supported Servlet init parameters
Example of web.xml descriptor:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app
PUBLIC '-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN'
'http://java.sun.com/j2ee/dtds/web-app_2_2.dtd'>

<web-app>
  <display-name>
    messenger
  </display-name>

  <servlet>
    <servlet-name>
      HttpServletAdapter
    </servlet-name>
    <servlet-class>
      com.evelopers.unimod.adapter.servlet.HttpServletAdapter
    </servlet-class>
    <init-param>
      <param-name>STATE_MACHINE_NAME</param-name>
      <param-value>A1</param-value>
    </init-param>
    <init-param>
      <param-name>LOGGER_NAME</param-name>
      <param-value>CONSOLE</param-value>
    </init-param>
  </servlet>

  <servlet-mapping>
    <servlet-name>
      HttpServletAdapter
    </servlet-name>
    <url-pattern>
      /controller
    </url-pattern>
  </servlet-mapping>

</web-app>
```

Table below summarizes interpreter features:

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread model</td>
<td>Strict</td>
</tr>
<tr>
<td>Startup</td>
<td>On startup interpreter converts FSM model XML-description into in-memory model and creates StateMachineEngine.</td>
</tr>
<tr>
<td>ConfigStore for root StateMachine if it has no</td>
<td>No default ConfigStore</td>
</tr>
</tbody>
</table>
one

StateMachineContext | Interpreter acts as EventProvider and provides its own StateMachineContext implementation that associates EventContext with HttpServletRequest parameters, UserContext with HttpSession parameters and AppContext with ServletContext parameters

Units | No default Units

ExceptionHandler | Default implementation throws Exceptions back as event e1000 with parameter EXCEPTION_DATA that stores occurred Exception.

<table>
<thead>
<tr>
<th>Table 2: Web Interpreters Features</th>
</tr>
</thead>
</table>

Web Interpreter doesn’t define default ConfigStore, it means that it is obligatory to define ConfigStore implementation class in FSM model XML-description. See Messenger Web sample for more details on using interpreter for Web applications.