sven efftinge's blog

Monday, May 07, 2012

Martin Fowler's State Machine DSL with Xtext 2.3

In his book on domain-specific languages, Martin Fowler introduces a small example along which he shows different techniques to implement a domain-specific language. The example goes like this:

Imagine you work for a company specialized on developing and installing systems for secret compartments and you have many customers with very different mechanisms. Mrs. H for instance wants to have a secret panel in her bedroom, which can only be opened after the door has been closed, the second drawer in her chest has been opened and the bedside light was turned on. Also the panel should be closed and locked immediately after someone opens the door - no matter in what state the system is in that case.

Martin proposes the following script as an appropriate definition of Mrs. H's secret compartment system:

```
events
  doorClosed
  drawOpened
  lightOn
  reset doorOpened
  panelClosed
end

commands
  unlockPanel
  lockPanel
  lockDoor
  unlockDoor
end

state idle
  actions {unlockDoor lockPanel}
  doorClosed => active
end

state active
  drawOpened => waitingForLight
  lightOn    => waitingForDraw
end

state waitingForLight
  lightOn => unlockedPanel
end

state waitingForDraw
  drawOpened => unlockedPanel
end

state unlockedPanel
  actions {unlockPanel lockDoor}
  panelClosed => idle
end
```

The script starts with two sections which declare the events and the commands. After that the different states are declared. Some of them execute declared commands as a side effect (the actions block). Also they contain declarations of transitions, i.e. to what state the system switches when a certain event is fired.

As I have already implemented this language with previous versions of Xtext, I'd like to make it a bit more interesting this time. Let's replace the more or less useless declaration of commands with the possibility to **write and call real code**!

```
events
  doorClosed
  drawOpened
  lightOn
  reset doorOpened
  panelClosed
end

state idle
  do {
```

```
    println("opened the door.")
    println("locked the panel.")
  }
  doorClosed => active
end

state active
  drawOpened => waitingForLight
  lightOn    => waitingForDraw
end

state waitingForLight
  lightOn => unlockedPanel
end

state waitingForDraw
  drawOpened => unlockedPanel
end

state unlockedPanel
  do {
    println("opened the panel.")
    println("locked the door.")
  }
  panelClosed => idle
end
```

As you can see the main change is that you are now able to call Java libraries right from within your state machine. I want these expressions to be statically typed (incl. full support for Java generics) and to keep the code as readable and dense as the rest of the DSL we need to have type inference like in Scala. Feature-wise everything should be supported, from for-loops to conditional logic, from the typical literals to more advanced concepts like lambda expressions (it's 2012 after all).

But how would we talk to a door resp. panel service in order to open and close them? We could use static methods, but that's bad design since then the application is hardly testable and you cannot easily switch the concrete implementations behind these services. So let's use dependency injection.
To support that I added a *services* block to the language where you list all required services. Now we can use these services within the action code :

```
events
  doorClosed
  drawOpened
  lightOn
  reset doorOpened
  panelClosed
end

services
  DoorService door
  PanelService panel
end

state idle
  do {
    door.open
    panel.close
  }
  doorClosed => active
end

state active
  drawOpened => waitingForLight
  lightOn    => waitingForDraw
end

state waitingForLight
  lightOn => unlockedPanel
end

state waitingForDraw
  drawOpened => unlockedPanel
end

state unlockedPanel
  do {
    door.close
    panel.open
  }
  panelClosed => idle
end
```

Our language would now be very testable and should integrate nicely with any Java project.

How Do I Implement Such A Language In Xtext?

To get a full implementation of this DSL, including not only a parser, linker, unparser, etc. but also a compiler generating readable and executable Java code as well as having nice integration in Eclipse, you need to create a fresh Xtext project, using the wizard and define the following grammar:

```
grammar org.xtext.example.mydsl.MyDsl with org.eclipse.xtext.xbase.Xbase
```

```
generate myDsl "http://www.xtext.org/example/mydsl/MyDsl"

Statemachine :
  {Statemachine}
 ('events'
    events+=Event+
 'end')?
 ('services'
    services+=Service*
 'end')?
 states+=State*;

Service :
  type=JvmTypeReference name=ID;

Event:
  resetEvent?='reset'? name=ID;

State:
 'state' name=ID
    ('do' action=XBlockExpression)?
    transitions+=Transition*
 'end';

Transition:
 event=[Event] '=>' state=[State];
```

I don't want to go into a detailed explanation of the grammar language, since that is explained in the documentation. But note, that in order to be able to write full Java types in the service declaration we refer to a library grammar rule (*JvmTypeReference*). The same applies for the expressions: The library grammar *'org.eclipse.xtext.xbase.Xbase'* predefines the full expressions and all we have to do here is to import the grammar in the first line and call the rule *XBlockExpression* within the rule *State*.

Now that we have defined the syntax of our language, we need to tell how it is translated to Java. For that matter we write some code, which creates a Java DOM out of the state machine DOM produced by the parser. To make this very readable and concise Xtext comes with an internal DSL implemented in the programming language Xtend. The actual code looks like this:

```
def dispatch void infer(Statemachine stm,
                        IJvmDeclaredTypeAcceptor acceptor,
                        boolean isPreIndexingPhase) {

  // create exactly one Java class per state machine
  acceptor.accept(stm.toClass(stm.className)).initializeLater [

    // add a field for each service annotated with @Inject
    members += stm.services.map[service|
      service.toField(service.name, service.type) [
        annotations += service.toAnnotation(typeof(Inject))
      ]
    ]

    // generate a method for each state having an action block
    members += stm.states.filter[action!=null].map[state|
      state.toMethod('do'+state.name.toFirstUpper, state.newTypeRef(Void::TYPE)) [
        visibility = PROTECTED

        // Associate the expression with the body of this method.
        body = state.action
      ]
    ]

    // generate a method containing the actual state machine code
    members += stm.toMethod("run", newTypeRef(Void::TYPE)) [

      // the run method has one parameter : an event source of type Provider
      val eventProvider = stm.newTypeRef(typeof(Provider), stm.newTypeRef(typeof(String)))
      parameters += stm.toParameter("eventSource", eventProvider)

      // generate the body
      body = [append('''
        boolean executeActions = true;
        String currentState = "«stm.states.head.name»";
        String lastEvent = null;
        while (true) {
          «FOR state : stm.states»
            if (currentState.equals("«state.name»")) {
              «IF state.action != null»
                if (executeActions) {
                  do«state.name.toFirstUpper»();
                  executeActions = false;
                }
              «ENDIF»
              System.out.println("Your are now in state '«state.name»'. Waiting for [«
                state.transitions.map[event.name].join(', ')»].");
              lastEvent = eventSource.get();
              «FOR t : state.transitions»
                if ("«t.event.name»".equals(lastEvent)) {
                  currentState = "«t.state.name»";
                  executeActions = true;
                }
              «ENDFOR»
```

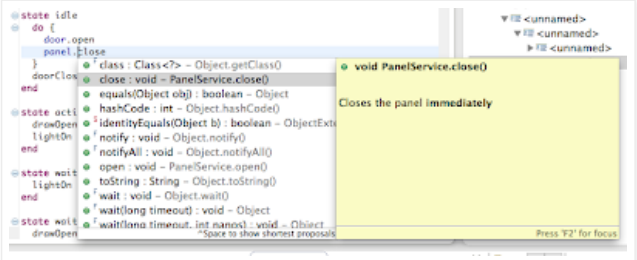
```
      }
    «ENDFOR»
    «FOR resetEvent : stm.events.filter[resetEvent]»
      if ("«resetEvent.name»".equals(lastEvent)) {
        System.out.println("Resetting state machine.");
        currentState = "«stm.states.head.name»";
        executeActions = true;
      }
    «ENDFOR»

  }
  ''')]
 ]
]
}
```

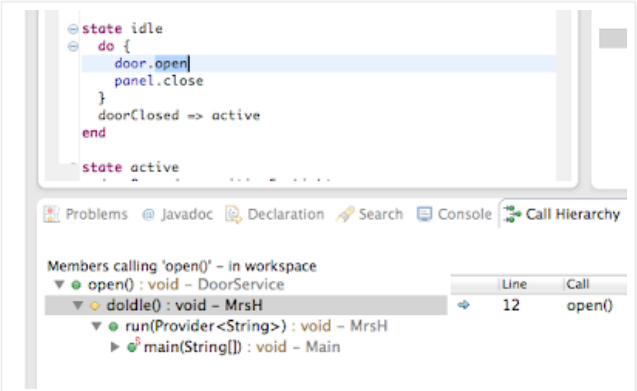
That's all you need. With just the grammar and the Xtend code above you have implemented the full language! I've pushed the full project to github.
This is what the Java code generated for Mrs. H's controller looks like.

What Do You Get?

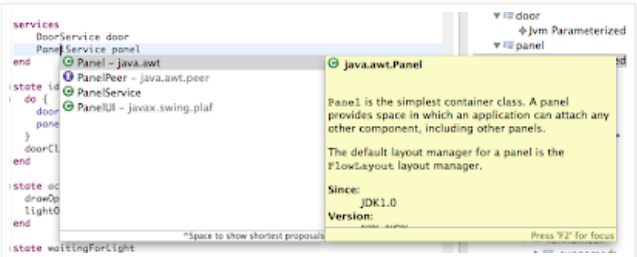
As already mentioned the expressions are feature complete and statically typed. The compiler can be run from any Java process (e.g. ant, maven, gradle, command line). Also the tooling is just like you would expect: content assist, coloring, hovers, refactorings, dead code analysis and even debugging work out of the box:



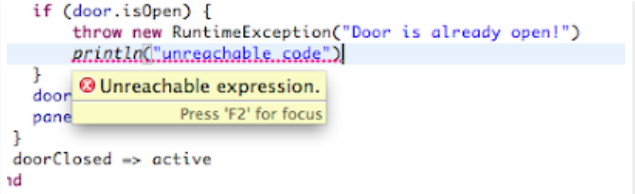
Content Assist For Expressions



Call-Hierarchy Integrated in JDT



Content Assist For Types



Dead Code Analysis

Eingestellt von Unknown um 8:59 AM

Labels: domain-specific languages, DSL, eclipse, java, xtext

Newer Post Home Older Post

About Me

Unknown

View my complete profile

Blog Archive

May (2)

Simple theme. Theme images by Storman. Powered by Blogger.