Software Engineering
The University of Auckland

Quality Assurance
Java Emitter Templates (JET)

Part II - Lecture 11

Today's Outline

- Java Emitter Templates (JET)
- JET Examples
  - Optimization: Partial Evaluation
  - Design Patterns: The Factory Pattern
  - Algorithms: Finite State Machines

Software Engineering
The University of Auckland

Java Emitter Templates (JET)

"Hello world" Generator

helloworld.txtjet:

```
<%@ jet package="hello" class="HelloWorldGenerator" %>
Hello, world!
```

Generates **HelloWorldGenerator**:

```
protected final String TEXT_1 = NL + " Hello, world!";

public String generate(Object argument)
{
  final StringBuffer stringBuffer = new StringBuffer();
  stringBuffer.append(TEXT_1);
  return stringBuffer.toString();
}
```

Use it:

```
HelloWorldGenerator g = new HelloWorldGenerator();
String result = g.generate(null);
```

Parameterized Generators


```
<%@ jet package="hello" class="HelloGenerator" %>
Hello, <%=argument%>!
```

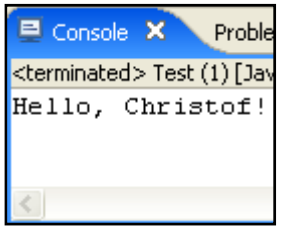
Generates `HelloGenerator`:

```
protected final String TEXT_1 = "Hello, ";
protected final String TEXT_2 = "!";

public String generate(Object argument) {
    final StringBuffer stringBuffer = new StringBuffer();
    stringBuffer.append(TEXT_1);
    stringBuffer.append(argument);
    stringBuffer.append(TEXT_2);
    return stringBuffer.toString();
}
```

Use it:

```
HelloGenerator g = new HelloGenerator();
String result = g.generate("Christof");
```



JET Syntax

- JET directive
  ```
  <%@ jet package="hello" class="HelloGenerator" %>
  ```
  More attributes for…
  - Importing packages `imports="java.io.* java.util.*"`
  - Changing JET tags `startTag="<@" endTag="@>"`
- Expressions `<%= argument+"xyz" %>`
- Scriptlets
  ```
  <% int x=0;  x++; %>
  <% if (dayTime.isMorning()) {%> Good Morning
  <% } else { %> Good Afternoon <% } %>
  ```
- Predefined variables:
  - **Object** `argument`: parameter of **generate** method
  - **StringBuffer** `stringBuffer`: generator output
    ```
    <% stringBuffer.append("Hello again!"); %>
    ```

JET Examples

Partial Evaluation (PE)

PE is an optimization technique

1. Evaluate constant parts of a program before runtime
2. Replace them by their result

Example of unoptimized code:

```
public class Computation {
    double x = Math.sqrt(21);
    int[] a = sortedRandomArray(10000);

    int[] sortedRandomArray(int n) {
        int [] a = new int[n];
        java.util.Random rnd = new java.util.Random();
        for(int i=0; i<n; i++) a[i] = rnd.nextInt();
        java.util.Arrays.sort(a);
        return a;
    }
}
```

Partial Evaluation (PE)

Using a template for partial evaluation:

```
<%@ jet class="ComputationGenerator" %>
public class Computation {
    double x = <%=Math.sqrt(21)%>;
    <% int[] a = sortedRandomArray(10000); %>
    int[] a = { <%=a[0]%>
    <% for(int i=1; i<a.length; i++) { %>
        , <%=a[i]%>
    <%}%> };
}
```

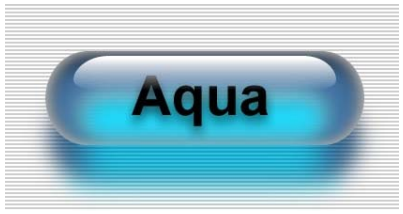
Generator output:

```
public class Computation {
    double x = 4.58257569495584;
    int[] a = { -7, -3, 1, 4, ... };
}
```

Factory Design Pattern

- Use a factory object with factory methods **to create other objects**
- **Abstraction of a constructor** to implement flexible allocation schemes
- A **factory method** for every kind of object that can be created, returning the created object
  - might choose the created object's class dynamically
  - might return it from an object pool
- Examples:
  - Configurable UI themes / look & feel
  - Exchangeable back-ends (e.g. compiler: for Java, .net, x86, …)
  - Configurable data structures (e.g. trees with different nodes)
- Generation useful for other design patterns as well (e.g. proxies)

Each button is created by a different class:



Factory Generator

We want to generate factory classes like this:

```
class CoolTheme implements Theme {
    public JButton newJButton() {
        return new CoolButton();
    }
    public JTextField newJTextField() {
        return new CoolTextField();
    }
    ...
}
```

The green parts vary from factory to factory.

They are our parameters.

We use the following argument type for our generator:

```
class Factory {
    String className;
    Class interf;
    Hashtable<String, String> classes;
}
```

Metatype **Class** can represent a Java interface.

Factory Generator


```
<%@ jet class="FactoryGenerator"
    imports="java.lang.reflect.*" %>
<% Factory f = (Factory) argument; %>
class <%=f.className%>
      implements <%=f.interf.getSimpleName()%> {
<% for(Method m : f.interf.getMethods()) { %>
   <% String tname = m.getReturnType().getSimpleName();%>
   public <%=tname%> new<%=tname%>() {
      return new <%=f.classes.get(tname)%>();
   }
<%}%>
}
```

```
public String generate(Object argument) { …
   Factory f = (Factory) argument; …
   for(Method m : f.interf.getMethods()) {
      String tname = m.getReturnType().getSimpleName();
      … stringBuffer.append(tname); …
}… }
```

Factory Generator Usage

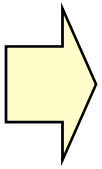

```
import java.util.*;
import java.io.*;
...
try {
   Factory f = new Factory();
   f.className = "CoolTheme";
   f.interf = Class.forName("Theme");
   f.classes = new Hashtable<String, String>();
   f.classes.put("JButton", "CoolButton");
   f.classes.put("JTextField", "CoolTextField");

   FactoryGenerator g = new FactoryGenerator();
   String result = g.generate(f);
   Writer output = new BufferedWriter(
      new FileWriter(f.className+".java"));
   output.write(result);
   output.close();
} catch(Exception ex) { ex.printStackTrace(); }
```

Getter/Setter Generator

- Common pattern: generating **extensions** for existing classes (class extensions)
- **Generate subclass** of the class we want to extended, i.e. generator gets superclass as argument
- Generated subclass can be used in place of its superclass, but has **additional functionality**
- Our example:
  - Generate class that defines getters/setters for some of the fields of a superclass
  - If field is meant to be read-only generate only getter
  - Getters & setters are one of the main characteristics of Java Beans

```
class X {
   String a;
   final int b = 0;
}
```



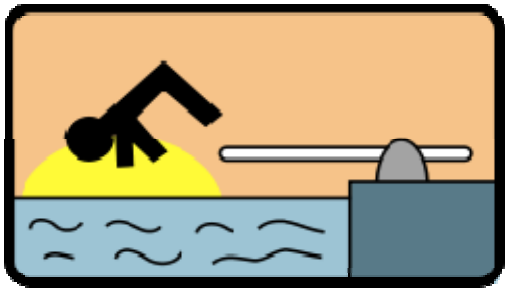
```
class XBean extends X {
   String getA() { return a; }
   void setA(String value)
   { a = value; }
   int getB() { return b; }
}
```

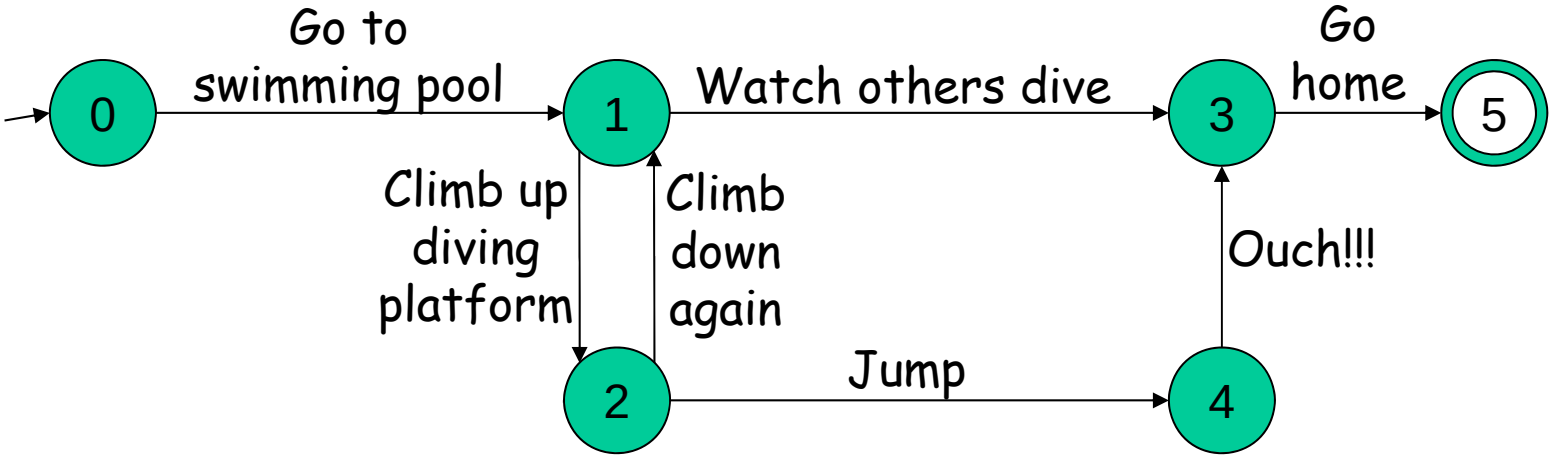
Getter/Setter Generator


```
<%@ jet class="BeanGenerator"
    imports="java.lang.reflect.*" %>
<% Class c = (Class) argument; %>
class <%=c.getSimpleName()%>Bean extends <%=c.getName()%>{
<% for(Field f : c.getFields()) { %>
    public <%=f.getType().getName()%>
    get<%=f.getName().substring(0,1).toUpperCase()
          + f.getName().substring(1)%>() {
        return <%=f.getName()%>;
    }
    <% if(!Modifier.isFinal(f.getModifiers())) { %>
    public void
    set<%=f.getName().substring(0,1).toUpperCase()
          + f.getName().substring(1)%>
    (<%=f.getType().getName()%> value) {
        <%=f.getName()%> = value;
    }
    <%}%>
<%}%> }
```

Finite State Machine (FSM) Generator

- Generation of algorithm code tailored to specific data
- Examples: scanners, parsers, numerical computation
- Our example: generate code for a FSM from a description of states and transitions
- Finite state machines:
  - One **start state**
  - Outgoing arrows represent **choices**
  - Choose a **transition** and go to next state
  - Execution stops in **end states**





Finite State Machine Model

- **FSM**: list of all states (for convenience) and reference to start state
- **State**: list of all possible transitions
- **Transition**: label and target state
- No designated end states: our end states simply have no outgoing transitions

```
State s0 = new State("0");
State s1 = new State("1");
…
FSM fsm = new FSM("SwimmingPool",
  new State[]{s0,s1,s2,s3,s4,s5}, s0);
s0.transitions.add(new Transition(
  "Go to swimming pool", s1));
s1.transitions.add(new Transition(
  "Watch others dive", s3));
…
```

```
class FSM {
  String name;
  State[] states;
  State start;
}
```

```
class State {
  String label;
  List<Transition>
      transitions;
}
```

```
class Transition {
  String label;
  State target;
}
```

Finite State Machine Generator


```
<%@ jet class="FSMGenerator" imports="java.util.*" %>
<% FSM fsm = (FSM) argument;
   int stateNum = 0;
   Hashtable<State, Integer> numForState
      = new Hashtable<State, Integer>();
%>
import java.io.*;
class <%=fsm.name%> {
<% for(State s : fsm.states) { %>
   public void
   <% if(!numForState.containsKey(s)) { %>
      <%="state"+stateNum%>
      <% numForState.put(s, stateNum); stateNum++; %>
   <% } else { %>
      <%="state"+numForState.get(s)%>
   <%}%> () { /* …method body on next slide… */ }
<%}%>
}
```

Finite State Machine Generator Cont'd


```
System.out.println("<%=s.label%>");
<% for(int i=0; i<s.transitions.size(); i++) { %>
   System.out.println(
      "<%=i%>...<%=s.transitions.get(i).label%>");
<%}%>
try {
   String input = new BufferedReader(
      new InputStreamReader(System.in)).readLine();
   <% for(int i=0; i<s.transitions.size(); i++) {
      State target = s.transitions.get(i).target; %>
      if(input.equals("<%=i%>"))
      <% if(!numForState.containsKey(target)) { %>
      <%="state"+stateNum %>
      <% numForState.put(target,stateNum); stateNum++; %>
      <% } else { %>
      <%="state"+numForState.get(target)%>
      <%}%>();
   <%}%>
} catch(IOException ex) { ex.printStackTrace(); }
```

FSM Generator Output Excerpt


```
public void state0() {
   System.out.println("0");
   System.out.println("0...Go to swimming pool");
   try {
      String input = new BufferedReader(…).readLine();
      if(input.equals("0")) state1();
   } catch(IOException ex) { ex.printStackTrace(); }
}
public void state1() {
   System.out.println("1");
   System.out.println("0...Watch others dive");
   System.out.println("1...Climb up diving platform");
   try {
      String input = new BufferedReader(…).readLine();
      if(input.equals("0")) state2();
      if(input.equals("1")) state3();
   } catch(IOException ex) { ex.printStackTrace(); }
} …
```

FSM Usage

Software Engineering
The University of Auckland

Start generated FSM:

```
SwimmingPool sp = new SwimmingPool();
sp.state0();
```

FSM execution (abbreviated output):

```
0...Go to swimming pool
   0
0...Watch others dive
1...Climb up diving platform
   1
0...Climb down again
1...Jump
   1
0...Ouch!!!
```

Today's Summary

- **JET syntax:** Expressions, Scriptlets, Predefined variables (**`argument`**, **`stringBuffer`**)
- Generators can solve different problems
  - **Partial evaluation** of program code
  - Support for **design patterns**
  - Generation of **extensions**
  - Generation of tailored **algorithm code**
  - Others (e.g. generation of **interfaces**)

Reference:

Eclipse JET Tutorial.
http://www.eclipse.org/articles/Article-JET/jet_tutorial1.html

Quiz

1. What is the JET directive? What are JET expressions and scriptlets?

2. Name four applications of JET templates. Describe how JET is applied in each of them.

3. How did the FSM generator manage state method names?