System building using Genesis’s box-and-wire mechanism

Patrick H. Winston

7 October 2014

Sources and deprecation of System.out.println

You can run the code examples given here by way of demonstration class in the tutorials package in the Genesis system.

Note that we do not use ordinary print statements in genesis. The reason is that our fancy print statements print on the Eclipse console with a clickable pointer to the place in the code where the fancy print statement lies. Hence:

// No good, cannot easily find the location of the printing statement.
System.out.println("Hello" + " " + "world");
// Great, one click takes you to the location of the printing statement.
Mark.say("Hello", "world");

In Genesis, modules are connected by wires that carry signals

In this document, I explain a Java mechanism that enables you to connect system modules to one another using a box-and-wire metaphor.

Suppose you want to connect distinct instances of a classes together. Further suppose the classes is named, unimaginatively, Test.

Your first step is to have the class implement the WiredBox class:

```java
public class Test extends AbstractWiredBox
...
```

Once created, you can wire the two class instances together using the wire static method of the Connections class. Suppose the instances are the values of source and destination:

```java
...
Connections.wire(source, destination);
...
```

Now you can transmit signals from source to destination as if there were a wire between them. Suppose, for example, that you want to send an object, the value of signal through the output port of instance source. All you need do is execute transmit as follows, in one of the Test class’s methods:

```java
...
Connections.getPorts(this).transmit(signal);
...
```

That object will be sent to the input port of instance destination. What happens when it gets there is determined by specifying which procedure is to be called when the object arrives. The specification is done, typically, in the Test class’s constructor, as in the following example:

```java
public class Test ()
...
Connections.getPorts(this).addSignalProcessor("processInput");
...
```
Thus, static methods in the `Connections` class perform a great deal of magic:

- One sets up connections between wired boxes.
- Another transmits a signal via an instance’s output port.
- Another specifies what is to happen when a signal is received via an instance’s input port.

**Note that for various reasons the procedure called when the signal is receive, `processInput` in the example, must be a void method with a single argument, which must be an instance of the `Object` class.**

Putting it all together, here is a complete micro demonstration:

```java
public class Test extends AbstractWiredBox {
    public Test() {
        Connections.getPorts(this).addSignalProcessor("processInput");
    }

    public void demonstrate() {
        String signal = "Hello world";
        Mark.say("Test instance named source transmits", signal);
        Connections.getPorts(this).transmit(signal);
    }

    public void processInput(Object input) {
        Mark.say("Test instance named destination receives", input);
    }

    public static void main(String[] args) {
        Test source = new Test();
        Test destination = new Test();
        Connections.wire(source, destination);
        source.demonstrate();
    }
}
```

Running the demonstration produces the following result:

```plaintext
Test instance named source transmits Hello world
Test instance named destination receives Hello world
```

**Wires can fan out from source ports and in to destination ports**

Wires can fan in and out:

```java
...
    Connections.wire(source, x);
    Connections.wire(source, y);
...
    Connections.wire(x, destination);
    Connections.wire(y, destination);
...
```

**You can name ports**

You can, if you wish, identify output and input ports other than the defaults, which are identified by the strings "output" and "input". For example, you might wish `source` to present a result on its "controlling" port, expecting it to be received on by the "controlled" port of `destination`. You do this by adding port names to the method that creates wires:

```java
    Connections.wire("controlling", source, "controlled", destination);
```
To put objects on this wire, you supply a port argument to the `transmit` method. Typically, you do this with public static variables.

```java
... public static final MY_OUTPUT = "controlling port";
... Connections.getPorts(this).transmit(MY_OUTPUT, signal);
... And, of course, you can specify which method is called by receiving a signal on a particular port:
... public static final MY_INPUT = "controlled port";
... Connections.getPorts(this).addSignalProcessor(MY_INPUT, "processControlledInput");
...
```

You can turn existing subclasses into boxes

Suppose you have a class that already extends some other class. How do you arrange for instances to be connected via wires? All you need do is implement the `WiredBox` interface, rather than extend the `AbstractWiredBox` class. Note that you will need to implement a `getName` method:

```java
public class AnotherTest implements WiredBox
    public String getName()
        return "Another test instance";
```

You can use Signal instances as the objects passed over wires

Because of characteristics of Java’s reflection mechanism, we required the signal processors to be void methods with single arguments. You can nevertheless send any number of objects at the same time by packaging them up in a `Signal` object, as in the following example.

```java
public class Test extends AbstractWiredBox
    public Test()
        Connections.getPorts(this).addSignalProcessor("processInput");

    public void demonstrate()
        String signalA = "Hello";
        String signalB = "World";
        Mark.say("Test instance named source transmits", signalA, signalB);
        Connections.getPorts(this).transmit(new Signal(signalA, signalB));

    public void processInput(Object input)
        if (input instanceof Signal)
            Signal signal = (Signal) input;
            String x = signal.get(0, String.class);
            String y = signal.get(1, String.class);
            Mark.say("Test instance named destination receives", x, y);

public static void main(String[] args)
    Test source = new Test();
    Test destination = new Test();
    Connections.wire(source, destination);
```
source.demonstrate();

Note that it is good programming hygiene to test the class of the object coming into a signal processing method to be sure it is what is expected. Note also that the idiom for disassembling a Signal instance is such that no casting is required.

Catching Errors

The transmitting done by wires is wrapped in try expressions. Hence, errors do not propagate when the receiving box blows out. If an error occurs, you will see something like the following:

Blew out while trying to apply method named ... to instance of class ...