Rule-Based Systems in Java

JESS IN ACTION

Ernest Friedman-Hill

MANNING

14 The reality connection

In this chapter you'll...

- Write a simulator in Java
- Interface it to Jess using JavaBeans
- Begin to write rules about JavaBeans

> *No man is an island, entire of itself; every man is a piece of the continent, a part of the main.*
>
> —John Donne

These days, very little software is written to stand alone. Component architectures, application servers, and networking have made it easier than ever to assemble software out of disparate parts. It's becoming increasingly rare to code a completely self-contained application. After all, with so many high-quality libraries, components, tools, services, and containers available, why reinvent the wheel?

So far in this book, you've written two applications using Jess. In each case, you used Jess as a standalone system. You didn't write or use a single line of Java code except Jess itself. You'll occasionally write software this way, but for the remainder of the book, we'll look at more realistic systems that use Jess as a part of a larger whole. Jess was designed to be embedded in other Java software. You've already seen how Jess can call Java functions—now you'll see that outside Java code can just as easily call on Jess. Each of the remaining applications you build will include a component written in Java.

In the next few chapters, you'll develop the control software for the HVAC (heating, ventilation, and air conditioning) systems in a large building. Jess will receive real-time temperature data and, based on these readings, will send commands to the multiple heating and cooling units in the building. Although this is a specialized example, the same principles would apply to controlling chemical plants, intelligent traffic-light sequencing, monitoring automated assembly lines, and even implementing manufacturing resource planning (MRP) systems. In every case, Jess has to receive data from the outside world and try to take action to affect the readings.

The Tax Forms Advisor from part 3 of this book and the PC Repair Assistant from part 4 only reasoned about facts in working memory. Facts are `jess.Fact` objects, and they live entirely inside Jess. When Jess is being used to directly react to events in the real world, it only makes sense that Jess needs to be able to reason about objects outside of itself. Section 6.5 talked about using the `definstance` function to add JavaBeans to working memory. The application you're about to build uses this technique to interact with a collection of thermostats, geothermal heat pumps, and automated air vents.

Your first task, and the topic of this chapter, is to learn about the system you're going to control. You will build a set of JavaBeans to represent and interact with the hardware, and then begin to use them from Jess. You'll also build a software simulator so that you can test your application.

To complete the control rules cleanly, you need to extend the Jess language by adding some custom functions written in Java; this is the subject of chapter 15. We'll look at Jess's external function interface and learn something about how Jess's function call mechanism works under the hood. Finally, in chapter 16, you'll write the rules, and the application will come together.

14.1 The system

Imagine a tall building (figure 14.1). The building is heated and cooled by a number of *heat pumps*. A heat pump is both a heating and cooling device. At any time, it can be in one of three modes: heating, cooling, or off. In this building, each heat pump services three adjacent floors

Warm or cold air from the heat pumps enters the building through adjustable vents. Each vent can be either open or closed, and although there may be several vents on each floor, they are wired in such a way that they must open and close in concert—that is, they act like a single vent.

You probably know that warm air rises and cold air sinks—it's basic physics. This adds a bit of nonlinearity to the system, because each heat pump can affect not only the floors to which it directly sends heated or cooled air, but the floors above and below it as well..

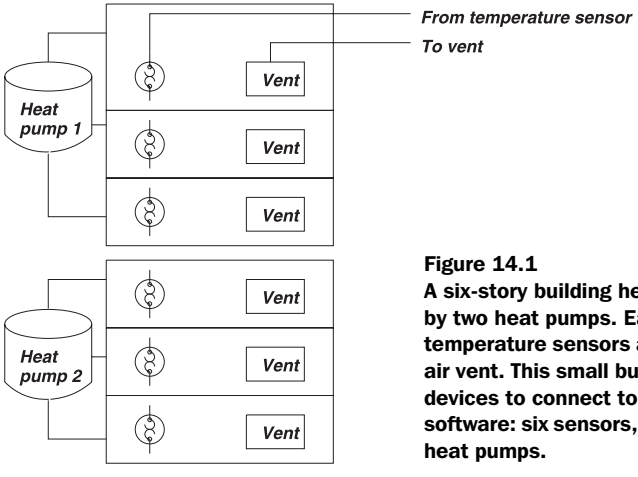

Figure 14.1
A six-story building heated and cooled by two heat pumps. Each floor has temperature sensors and an automated air vent. This small building contains 14 devices to connect to the control software: six sensors, six vents, and two heat pumps.

14.2 Defining the hardware interface

This system includes hardware and software components. This is a book about software, so perhaps we should get the hardware considerations out of the way first. Assume that the building's control systems, purchased from Acme HVAC Systems, Inc., include all the sensors and actuators you need. In fact, they're already connected to one master computer on which the control software will run. Acme HVAC Systems has provided a library of functions you can call from C language code to turn the heat pumps on and off, read the temperatures, and operate the vents. To make this concrete, imagine that this is a complete list of the available library functions:

```
int getHeatPumpState(int heatPump);
void setHeatPumpState(int heatPump, int state);

double getTemperature(int sensor);

int getVentState(int vent);
void setVentState(int vent, int state);
```

Note that these are C functions, so on and off values like the state of each vent must be represented as type `int`, because classic C doesn't have a `boolean` type like Java. The first argument to every function is an index; for example, you would pass the value 3 to `getTemperature` to get the temperature reading from sensor number 3. By convention, all these index numbers start at 1, not 0.

14.2.1 Native methods

Of course, you're writing the software in Java, so you need to make the connection between your Java code and the C library. This is easy enough to do using the *Java Native Interface* (JNI). A *native method* is a method of a Java class that happens to be written in some system-level language like C. Native methods use a special JNI library to bridge the gap between the Java virtual machine and the real machine outside. The knowledge that your program will be dealing with external hardware can be encapsulated inside a single Java class; that class will have one native method for each of the C library functions.

Writing native methods is outside the scope of this book. The best reference for writing them is probably Sun's web-based Java tutorial (http://java.sun.com/docs/books/tutorial/native1.1/index.html). For our purposes, all you need to know is that from the Java side, you declare a native method like this:

```
public native void myMethod(String myArgument);
```

A native method has no body (just like an abstract method) and includes the keyword `native` in its declaration; otherwise, there's nothing special about it. Java expects to find the definition of this method in an external library you must provide.

The important thing to realize here is that any Java code can call `myMethod` without knowing that it's a native method. From the caller's perspective, a native method looks like any other Java method. A subclass can override a normal parent method using a native method, and an implementation of an interface can provide a native method to satisfy the interface.

You can use this last fact to your advantage. You'll define a Java interface to represent the functions in the C library. Then you can write multiple implementations of the interface: one that uses native methods to access the real hardware, and one that implements a simulation of the hardware. You can then use the simulator to develop and test the software. When the software is ready, you could swap in the hardware implementation and try it out on the real system. For the purposes of this book, you'll only write the simulator.

The interface has one method for each of the C library functions, so it looks like this:

```
package control;
public interface Hardware {
    public final int OFF=0, HEATING=1, COOLING=2;

    int getHeatPumpState(int heatPump);
    void setHeatPumpState(int heatPump, int state);
    int getNumberOfHeatPumps();

    double getTemperature(int sensor);

    boolean getVentState(int vent);
    void setVentState(int vent, boolean state);
}
```

Note that you'll put all the Java code you write for this application into a package named `control`.

The methods in this interface look a lot like the C functions, with a few exceptions. Some of the `int` types are now `boolean`, because in Java, `boolean` is the best type to represent an on or off value. The heat pump state is still represented as an integer, though, and three constants represent the legal values (which presumably came from the C library manual).

14.2.2 Writing a simulator

The hardware simulator is fairly simple. The implementations of methods like `getVentState` and `setVentState` are trivial. The simulator object contains an array of `boolean` to record the state of each vent, and these two methods simply set and get

the appropriate elements of that array. The only complicated method is `getTemperature`, because it must return a realistic temperature based on the complete state of the system. The simple part of the simulator is shown in listing 14.1; you'll work on `getTemperature` next. Note that the implementation shown here, like most published code, skimps on error handling. All the getter and setter methods should check that their first arguments are greater than zero and less than or equal to the number of devices, but we've omitted this code here to keep things short. This version of `Simulator` is an abstract class because we haven't implemented `getTemperature` yet.

Listing 14.1 The basic parts of the hardware simulator

```
package control;
public abstract class Simulator implements Hardware {
    private int[] m_heatpumps;
    private boolean[] m_vents, m_sensors;

    public Simulator(int numberOfFloors) {
        if (numberOfFloors % 3 != 0)
          throw new RuntimeException("Illegal value");

        m_heatpumps = new int[numberOfFloors/3];
        m_vents = new boolean[numberOfFloors];
        m_sensors = new boolean[numberOfFloors];
    }

    public int getHeatPumpState(int heatPump) {
        return m_heatpumps[heatPump-1];
    }

    public void setHeatPumpState(int heatPump, int state) {
        switch (state) {
        case OFF: case HEATING: case COOLING:
            m_heatpumps[heatPump-1] = state; break;
        default:
            throw new RuntimeException("Illegal value");
        }
    }

    public int getNumberOfHeatPumps() {
        return m_heatpumps.length;
    }

    public boolean getVentState(int vent) {
        return m_vents[vent-1];
    }

    public void setVentState(int vent, boolean state) {
        m_vents[vent-1] = state;
    }

}
```

14.2.3 Simulating getTemperature

The meat of the HVAC simulator is the `getTemperature` method, which returns a value based on the complete state of the system. Each time you call `getTemperature` with a given argument, you can get a different answer back, because the temperatures will constantly change based on many factors. At least four things go into the calculation of the temperature on each floor of the building:

- The current temperature on that floor
- Whether the floor is actively being heated or cooled
- Heat leakage from outside, through the walls
- Heat leakage from other floors

The simulator should take each of these factors into account. The current temperature on each floor can be held in an array of `double`, so that's easy. Whether a given floor is being heated or cooled depends on what the corresponding heat pump is doing and whether that floor's vent is open or closed. To account for heat leakage from outside, you need a variable to hold the outside temperature. Finally, the same array of current temperatures is all you need to compute the heat leakage from other floors.

The simplest way to write the simulator is so that it works in real time—it includes a loop that continuously recomputes the temperature. This loop runs in a separate thread, so the temperatures continue to update even if no calls to `getTemperature` are made. Therefore, all you need to do is to figure out an equation to calculate the current temperature from the temperature at the last time step and the changes due to the factors just listed.

A law of physics called *Newton's Law of Cooling*, simply put, states that the larger the temperature difference between two bodies, the faster heat flows between them. Therefore, a reasonable way to calculate the temperature change per second on a given floor is to calculate the difference between that floor's current temperature and some other body (such as the outside air or the hot air coming from the heat pump) and multiply this difference by some constant value giving the actual heat transfer rate. The constant varies depending on the materials involved; we'll arbitrarily choose the value 0.01. If you do this for each source of heat, for each floor, for each time step, you'll have a reasonable simulation of the temperatures in an office building. The code to do this is shown in listing 14.2. `HOT` and `COLD` represent the temperatures of the hot and cold air coming from the heat pump, and `m_outdoor` is the (variable) temperature of the outside air. The

big `while` loop looks at each floor in turn, calculating the contribution of each of the factors listed earlier to the new temperature for that floor. `Simulator`'s constructor starts a background thread that periodically updates the temperature for each floor.

Listing 14.2 An implementation of *`getTemperature`*

```
package control;
import java.util.Arrays;

public class Simulator implements Hardware {
    private final double RATE = 0.01;
    private final double HOT = 100, COLD=50;
    private double[] m_temperature;
    double m_outdoor = 90;

    public Simulator(int numberOfFloors) {
        //...
        m_temperature = new double[numberOfFloors];
        Arrays.fill(m_temperature, 70);
        new Thread(this).start();
    }
    //...
    public double getTemperature(int sensor) {
        return m_temperature[sensor-1];
    }

    public void run() {
        while (true) {
            for (int i=0; i<m_temperature.length; ++i) {
                double temp = m_temperature[i];

                // Heating and cooling, and heat rising
                switch (state(i)) {
                case HEATING:
                    temp += (HOT-temp)*RATE; break;
                case COOLING:
                    temp += (COLD-temp)*RATE; break;
                case OFF:
                    temp += (i+1)*0.005; break;
                }

                // Outdoor leakage
                temp += (m_outdoor-temp)*RATE/2;

                m_temperature[i] = temp;
            }
            try {Thread.sleep(1000);}
            catch (InterruptedException ie) { return;}
        }
    }
```

```
    private int state(int floor) {
        if (getVentState(floor + 1))
            return getHeatPumpState(floor/3 + 1);
        else
            return OFF;
    }
}
```

14.2.4 Adding a graphical interface

Now the simulator runs, but currently there's no way to see what it's doing. A graphical display that shows the full state of the building at a glance would be very helpful. A simple GUI could display a row of labels for each floor: one label each for heat pump state, vent state, and temperature. If you add a text field that lets you set the outdoor temperature, the simulator will be ready to test the HVAC Controller system. Figure 14.2 is a screen shot of this simple interface. I won't show the code here, but it's included in the sample code for this chapter. The `SimulatorGUI` class has a constructor that accepts a `Simulator` as a constructor argument and then uses a thread to poll the `Simulator` to determine its state over time.

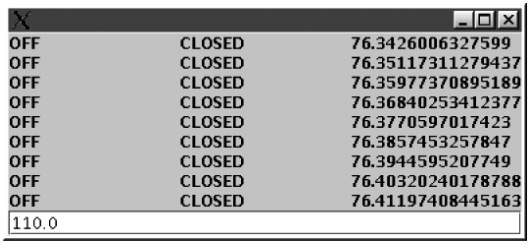

Figure 14.2
A GUI for the HVAC simulator. The three columns of labels represent the heat pump state, vent state, and temperature for each floor. The text field on the bottom shows the current outdoor temperature.

The simulator is now a reasonable stand-in for a real HVAC system. Next you need a way to connect the simulator—or, in its place, the HVAC system itself—to Jess. You'll do this by wrapping the simulator's interface in a set of JavaBeans.

14.3 Writing the JavaBeans

Jess's working memory can hold not only plain facts, but also *shadow facts*, which are placeholders for Java objects outside of Jess (see section 6.5 for the details). More specifically, they are placeholders for JavaBeans. From Jess's perspective, a JavaBean is just an instance of a Java class with one or more *properties*—specially named methods that let you read and/or write to a named characteristic of the object. A very simple JavaBean to represent a temperature sensor for the HVAC system could look like this:

```
package control;

public class Thermometer {
    private Hardware m_hardware;
    private int m_floor;
    public Thermometer(Hardware hw, int floor) {
        m_hardware = hw;
        m_floor = floor;
    }

    public double getReading() {
        return m_hardware.getTemperature(m_floor);
    }
}
```

This Bean has one read-only property named `reading`. If you used this as is, Jess could create shadow facts for `Thermometer` instances, and each `Thermometer` fact would have a `reading` slot—but if the value in that slot changed, Jess wouldn't know it. The shadow fact's `reading` slot would never change.

A more useful Bean includes a mechanism for notifying interested parties that the value of a property has changed. The standard JavaBeans mechanism for doing this is via `PropertyChangeEvents`. Jess works best with Beans that send one of these events whenever the value of any property changes. It's fairly easy to implement this behavior in your own Beans using the helper class `PropertyChangeSupport` in the `java.beans` package. Most of the code you need to write is boilerplate. A class that supports `PropertyChangeListeners` must include the methods `addPropertyChangeListener` and `removePropertyChangeListener`; they always look the same, but they still have to be included in every JavaBean you write. Many people create a base class for their Beans that contains a protected `PropertyChangeSupport` member and implementations of these two methods; you'll do this here. The helper class looks like this:

```
package control;

import java.beans.*;

public abstract class BeanSupport {
    protected PropertyChangeSupport m_pcs =
        new PropertyChangeSupport(this);

    public void
    addPropertyChangeListener(PropertyChangeListener p) {
        m_pcs.addPropertyChangeListener(p);
    }
    public void
    removePropertyChangeListener(PropertyChangeListener p) {
        m_pcs.removePropertyChangeListener(p);
    }
}
```

Note that `PropertyChangeSupport` itself includes the `addPropertyChangeListener` and `removePropertyChangeListener` methods. It would be nice if you could use `PropertyChangeSupport` as a base class for your JavaBean—but you can't: `PropertyChangeSupport's` only constructor accepts the source object (the JavaBean) as an argument, and `super(this)` is invalid Java. A default constructor that assumed the current object was the source would have been useful!

You can now add a `run` method to `Thermometer` that polls the temperature on a given floor, sending out `PropertyChangeEvents` to notify Jess when the temperature changes. You'll also add a read-only `floor` property to identify the individual `Thermometer`. The result is shown in listing 14.3.

Listing 14.3 The `Thermometer` class, including automated notification

```
package control;

public class Thermometer extends BeanSupport
    implements Runnable {
    private Hardware m_hardware;
    private int m_floor;
    private double m_oldReading;

    public Thermometer(Hardware hw, int floor) {
        m_hardware = hw;
        m_floor = floor;
        new Thread(this).start();
    }

    public int getFloor() {
        return m_floor;
    }

    public double getReading() {
        return m_hardware.getTemperature(m_floor);
    }

    public void run() {
        while (true) {
            double reading = getReading();
            m_pcs.firePropertyChange("reading",
                new Double(m_oldReading),
                new Double(reading));
            m_oldReading = reading;
            try { Thread.sleep(1000); }
            catch (InterruptedException ie) { return; }
        }
    }
}
```

The `PropertyChangeSupport` class needs the name of the property along with the old and new values to create the appropriate event. Note that `PropertyChangeSupport` is smart enough to send an event only if the old and new values are different, so you don't need to bother with that test yourself. There are a few overloaded versions of the `firePropertyChange` method to handle different properties of different types. For some types, though, like the `double` value here, you need to use wrapper objects to pass the old and new values.

14.3.1 Rules about Thermometers

Given this improved version of `Thermometer`, it's possible to write rules that react to temperatures. If this class and the `Simulator` class have been compiled and are available on the `CLASSPATH`, the following code will print a warning message after a short interval. Remember that all the code for the HVAC Controller is available from this book's web site:

```
;; Create a simulator
(bind ?hardware (new control.Simulator 3))

;; Monitor the temperature on the first floor
(defclass Thermometer control.Thermometer)
(definstance Thermometer
    (new control.Thermometer ?hardware 1))

;; Report when the temperature gets to be over 72
(defrule report-high-temperature
    (Thermometer (reading ?r&:(> ?r 72)))
    =>
    (printout t "It's getting warm on the first floor" crlf)
    (halt))

(run-until-halt)
(exit)
```

The `defclass` function (first discussed in section 6.5) tells Jess that you're going to define shadow facts for objects of a certain class (here `control.Thermometer`). The `definstance` function installs an object of that class into Jess. Finally, the rule here matches only `Thermometer`s that are reading over 72 degrees. The temperatures in the simulator start at 70 and drift up quickly, so this rule fires after a short interval. Notice how once a JavaBean is installed into Jess, it looks just like an ordinary fact, and the `Thermometer` pattern that matches the Bean here is just an ordinary pattern. You use `(run-until-halt)` instead of `(run)` because you want Jess to wait for the rule to be activated—the agenda will be empty when you call `(run-until-halt)`, but when the temperature becomes high enough, the rule will activate and fire.

14.3.2 Writing the other Beans

The `Hardware` interface, like the C library it wraps, represents the state of a vent as a Boolean value and the state of a heat pump as an integer. Because symbols like `open` and `closed` would be easier to work with than the corresponding Boolean values `true` or `false`, it would be a good idea to write the `Vent` and `HeatPump` JavaBeans to use meaningful strings as property values. You need to convert both ways between these symbols and the underlying integer and Boolean values, and you'll need to do so in the code you write in the next chapter, too, so let's isolate the code to do these conversions in a single class named `State`. Part of the straightforward `State` class is shown in listing 14.4.

Listing 14.4 Converting between the `Hardware` states and convenient symbolic names

```
package control;

public class State {
    public static final String
        OPEN="open",
        CLOSED="closed",
        OFF="off",
        HEATING="heating",
        COOLING="cooling";

    public static String vent(boolean val) {
        return val ? OPEN : CLOSED;
    }

    public static boolean vent(String val) {
        if (val.equals(OPEN))
            return true;
        else if (val.equals(CLOSED))
            return false;
        else
            throw new IllegalArgumentException(val);
    }

    // Analagous heatpump() methods not shown
}
```

By collecting these conversions together in a single class and defining the symbols as constants in one place, you may avoid a lot of debugging.

The Vent bean

The other JavaBeans you need to write are a little different from the `Thermometer` class. Whereas the `Hardware` interface only has a method for reading the temper-

ature, there are methods for both getting and setting the heat-pump and vent states. The corresponding `HeatPump` and `Vent` Beans therefore need both setting and getting methods. Calling `Vent.setState` (for example) should fire a `PropertyChangeEvent`. Of course, there should also be a background thread watching for changes from the outside. You can reuse the `BeanSupport` base class to help implement the event support. The `Vent` class is shown in listing 14.5. Note how it uses `State` to convert between Boolean values and the symbolic constants `open` and `closed`. The `HeatPump` class (not shown) is very similar.

Listing 14.5 A JavaBean to represent an automated vent

```
package control;

public class Vent extends BeanSupport
    implements Runnable {
    private Hardware m_hardware;
    private int m_floor;
    private boolean m_oldState;

    public Vent(Hardware hw, int floor) {
        m_hardware = hw;
        m_floor = floor;
        new Thread(this).start();
    }

    public int getFloor() {
        return m_floor;
    }

    public String getState() {
        return State.vent(m_hardware.getVentState(m_floor));
    }

    public void setState(String szState) {
        boolean state = State.vent(szState);
        m_hardware.setVentState(m_floor, state);
        m_pcs.firePropertyChange("state",
            new Boolean(m_oldState),
            new Boolean(state));
        m_oldState = state;
    }

    public void run() {
        while (true) {
            boolean state = m_hardware.getVentState(m_floor);
            m_pcs.firePropertyChange("state",
                new Boolean(m_oldState),
                new Boolean(state));
            m_oldState = state;
```

```
                try { Thread.sleep(1000); }
                catch (InterruptedException ie) { return; }
            }
        }
    }
```

14.4 JavaBeans and serialization

The notion of *pickling* is part of the JavaBeans concept. The state of an application made out of JavaBeans is fully specified by the values of all the properties of those Beans together with information about the connections between the Beans. A JavaBeans framework can create an instant application by storing the property values and connection information in a file and then later reconstituting the original Beans. Generally, Java's serialization API is used to do the pickling, so most JavaBeans implement the `java.io.Serializable` interface. It's easy to add this capability to your own JavaBeans: just declare that the class `implements Serializable`. There are no methods in the `Serializable` interface; it is a *tagging interface* used to signify to the Java virtual machine that it is OK to serialize the data from instances of the class.

Jess includes two built-in functions, `bload` and `bsave`, that can save and restore Jess's state on any Java input and output streams using Java's serialization API. If you intend to use these functions, be sure any Java objects you add to working memory implement `java.io.Serializable`, or these functions will fail. You won't be using `bload` or `bsave` in the HVAC Controller application, so it won't be necessary to make the Beans from this chapter implement this interface.

14.5 Summary

In this chapter, you began the work of using Jess to control the HVAC systems of an office building. You defined a Java interface to represent Acme HVAC Systems' C language library. You wrote a simulator in Java to test your control algorithms against that interface. You then wrote some JavaBeans, which you can inject into Jess's working memory so that Jess can monitor the state of the HVAC system in real time. The sample rule in section 14.3.1 suggests how these JavaBeans will be used.

In the next chapter, you will write additional Java functions to help control the HVAC systems. You will then use these Java functions to extend the Jess language by adding new commands. While we're at it, we'll look at Jess's extension facilities in general, and the `jess.Userfunction` interface in particular, and you'll learn how to write a range of different Jess extensions.

Armed with all the Java code developed in this and the next chapter, writing the rules to control the HVAC system will be straightforward. That will be the task of chapter 16.

EXPERT SYSTEMS/JAVA

JESS IN ACTION Rule-Based Systems in Java

Ernest Friedman-Hill

Imagine a different way to program in which you specify *rules* and *facts* instead of the usual linear set of instructions. That's the idea behind rule-based programming. A *rule engine* automatically decides how to apply the rules to your facts and hands you the result. This approach is ideal for expressing business rules and is increasingly used in enterprise computing.

Jess is a popular rule engine written in Java. It's supported by Sandia Labs and has an active online community. If you have a problem that can be solved with rules, *Jess in Action* will show you how. (If you are not sure, read chapter 2.) Written by the creator of Jess, this book is an accessible and practical guide to rule-based system development in Java.

Jess in Action first introduces rule programming concepts and teaches you the Jess language. Armed with this knowledge, you then progress through a series of fully-developed applications chosen to expose you to practical rule-based development. The book shows you how you can add power and intelligence to your Java software.

What's Inside

- Introduction to rule-based thinking
- Jess language tutorial
- Complete examples of ...
  - Tax forms advisor
  - Diagnostic assistant
  - Fuzzy logic controller
  - Web agent
  - J2EE apps

Dr. Friedman-Hill is the developer of Jess. A Principal Member of the Technical Staff at Sandia National Laboratories, he lives in Gaithersburg, MD.

FREEBIES

- Binary version of Jess*
- Complete examples on the web

*For non-commercial use

"... clear, crisp, well-focused ... the organization is smooth, well-thought-out, ... this book rocks."

—Ted Neward, Author
Server-Based Java Programming

"... *the* Jess book. A nice balance between an introduction and a reference ...."

—John D. Mitchell, Coauthor
Making Sense of Java

"Friedman-Hill writes clearly. The topic is complicated, and he does an excellent job explaining it ... I recommend this book."

—Roedy Green, Author
The Java Glossary

"... intuitive and clever examples that show the reader how to build intelligent Java applications with Jess."

—Robert B. Trelease, Ph.D.
UCLA Brain Research Institute

www.manning.com/friedman-hill

AUTHOR ONLINE Author responds to reader questions

Ebook edition available

9 781930 110892 54995
ISBN 1-930110-89-8

MANNING $49.95 US/$74.95 Canada