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.

“... 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. Terlouw, Ph.D.
UCLA Brain Research Institute

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

*For non-commercial use

MANNING $49.95 US/$74.95 Canada
In this chapter you'll...

- Create a window
- Fill it with graphical components
- Write event handlers
- … all from the Jess language
These days, virtually all computer applications come with a graphical user interface (GUI). Graphical interfaces make many users feel more comfortable than command lines. Furthermore, by providing a bounded set of commands (explicit in menus or implicit in radio buttons, checkboxes, and other controls), GUIs provide guidance that helps users use the application efficiently and correctly. One advantage of a GUI is perhaps less obvious: Because it limits user actions, it can make invalid user inputs impossible, thereby simplifying the application’s internal error-checking code.

Your first real application, the Tax Forms Advisor, had only a command-line interface. The PC Repair Assistant you’re developing now should have a friendlier GUI. You’ll develop such an interface in this chapter using only Jess’s Java reflection capabilities—that is, you’ll write only Jess scripts, not Java code. In the process, we’ll look at how Jess lets you develop software using the Java libraries in an iterative and interactive way.

The GUI you’ll develop in this chapter will use Java’s Swing library. I’ll assume some basic knowledge of Swing (and Java GUIs in general) on your part. You might want to keep some documentation for the `javax.swing` package handy for reference.

Note that you’ll be using an iterative approach to develop this GUI; you’ll see the same Jess functions presented several times with various refinements applied each time. The final code for the completed application is available from this book’s web site.

### 13.1 Getting started

You’ll be using many of the classes in Java’s `javax.swing`, `java.awt`, and `java.awt.event` packages. Although you could spell out these package names when you need them, it will be easier if you use Jess’s `import` function to make this unnecessary. Jess’s `import` function, like the Java keyword, makes it possible for you to refer to all the classes in a given Java package by using their class names, without spelling out the package name each time. Note how after importing `javax.swing.*`, you can create a `javax.swing.JButton` using just its class name:

```jess
Jess> (import javax.swing.*)
TRUE
Jess> (import java.awt.*)
TRUE
Jess> (import java.awt.event.*)
TRUE
Jess> (new JButton "Example Button")
<External-address:javax.swing.JButton>
```
The graphical components of the application will be accessed from different sections of Jess code. In particular, several functions need access to the application’s top-level window component (a JFrame). You’ll therefore store a reference to it in a defglobal. Unfortunately, the reset function by default resets the values of defglobals, and it would complicate the application if you had to worry about the possibility of your only reference to the JFrame disappearing. To deal with this situation, you use the set-reset-globals function to tell Jess to leave defglobals alone when reset is called:

```
Jess> (set-reset-globals FALSE)
FALSE
```

With these details taken care of, you can start coding the GUI.

### 13.2 Displaying a window

The first and most fundamental graphical element you need to create is a javax.swing.JFrame—a top-level window. Let’s create one with an appropriate title, size it, make it visible, and keep a reference to it in a defglobal named ?*frame*:

```
Jess> (defglobal ?*frame* = (new JFrame "Diagnostic Assistant"))
TRUE
Jess> (?*frame* setSize 520 140)
Jess> (?*frame* setVisible TRUE)
```

I’m cheating a little here: I determined the appropriate size after the whole GUI was designed, and then came back here and typed it in. In your own programs, you will learn the dimensions necessary through experience or experimentation. When you enter the last line of code, an empty window appears on the screen, as shown in figure 13.1.

![Figure 13.1 An empty JFrame created via a short Jess script as shown in the text](image)
13.3 Displaying questions

Right now, the application you’re developing displays questions and accepts answers from the command line. As a first step toward producing a real GUI, let’s change the application to display the text of the questions in the window you’ve created (for the moment, you’ll still accept input from the command line). You can use a JTextArea inside a JScrollPane to accomplish this. You need to access the JTextArea from the ask module, so you’ll store a reference to it in a defglobal named qfield. Here’s the code to create a JTextArea and add it to your JFrame:

```
Jess> (defglobal ?*qfield* = (new JTextArea 5 40))
TRUE
Jess> (bind ?scroll (new JScrollPane ?*qfield*))
<External-Address:javax.swing.JScrollPane>
Jess> ((?*frame* getContentPane) add ?scroll)
Jess> (?*qfield* setText "Please wait...")
Jess> (?*frame* repaint)
```

You set the text of the new text area to the string "Please wait...". Figure 13.2 shows what the window looks like now.

As you’ll recall from chapter 12, you used the ask module (as originally defined in chapter 10) to provide the temporary command-line interface to your application. In this chapter, you’ll modify the ask module yet again to connect it to your GUI. In particular, you need to modify the function ask-user. Listing 13.1 is a new version of ask-user that displays the questions in the JTextArea instead of at the command line. If you run the whole application with this modified ask-user, you’ll need to look at the JFrame to see the questions and at the command line to type your answers. You’ll return to this function soon—the final version will no longer read from the command line.

![Figure 13.2 Application window with an added JTextArea for displaying questions](image)
Getting answers

13.4 Getting answers

Now that your application displays questions to the user via its new GUI, it ought to collect answers the same way. Currently, the input routine ask-user reads answers as strings the user types at the command line. As a first step, you’ll modify the ask module to use a simple text field to collect its answers for all question types. This will give you a framework to build on (later you’ll replace this input field with a combo box and other more user-friendly components). First you’ll build a panel containing a JTextField and a JButton labeled OK; then you’ll add it to the bottom of the application window. Listing 13.2 shows the code to do this; the result looks like figure 13.3. I used get-member to obtain the value of the constant SOUTH in the BorderLayout class. The calls to validate and repaint are necessary whenever you modify the contents of a window that’s already visible on the screen.

Listing 13.1  ask-user function to display questions (changed lines in bold)

(deffunction ask-user (?question ?type ?valid)
  "Ask a question, and return the answer"
  (bind ?answer "")
  (while (not (is-of-type ?answer ?type ?valid)) do
    (bind ?s ?question "")
    (if (eq ?type multi) then
      (bind ?s (str-cat ?s ?*crlf*
                    "Valid answers are "))
      (foreach ?item ?valid
          (bind ?s (str-cat ?s ?item " ")))
    (bind ?s (str-cat ?s ":"))
    (?*qfield* setText ?s)
    (bind ?answer (read))
  )
  (return ?answer))

Listing 13.2  Adding an input area to the application window

;; Add a JTextField and an OK button to our application window
(defglobal ?*apanel*  = (new JPanel))
(defglobal ?*afield*  = (new JTextField 40))
(defglobal ?*afield-ok* = (new JButton OK))
(?*apanel* add ?*afield*)
(?*apanel* add ?*afield-ok*)
((?*frame* getContentPane) add ?*apanel* (get-member BorderLayout SOUTH))
(?*frame* validate)
(?*frame* repaint)

;; Now attach an event handler
(deffunction read-input (?EVENT)
In Java programming, you add behaviors to components like buttons and text fields by writing event handlers. An event handler is an implementation of a special interface, with a member function that performs the desired behavior. The interface used for button clicks is `java.awt.event.ActionListener`. You register the event handlers you write by calling an appropriate method on the component; for `ActionListener` the method is called `addActionListener`. When some user input occurs, the component creates an event object and passes it as an argument to the appropriate method of all the event handlers that are registered with it. In the case of button clicks, the event object is an instance of the class `java.awt.event.ActionEvent`, and the method in the `ActionListener` interface is called `actionPerformed`.

The function `read-input` in listing 13.2 is an event handler written in Jess. The class `jess.awt.ActionListener` is an event adapter that lets you specify that a deffunction should be invoked when a Java `ActionEvent` occurs. Jess supplies a whole family of these event adapters, one for each event type defined by the Java APIs, and they all are used the same way. You can create one using the name of a deffunction and a `jess.Rete` object (see the next section) as constructor arguments. Then you use the matching Java method on an AWT or Swing component to register the listener, just as you would in Java. When the component sends an
event, the event adapter invokes the deffunction, passing it the event object as the lone argument (the parameter ?EVENT in the listing). This arrangement is shown in figure 13.4.

Now you have an input field that asserts a Jess fact in response to a button click or the user pressing Enter. Before you can modify the program to use this asserted fact, however, we need to look in a bit more depth at the architecture of a GUI-based application.

13.4.1 The main thread vs. the event thread

A thread is a single flow of control in a Java program. Individual streams of code can be, and often are, running in multiple separate threads simultaneously. You can create threads explicitly using the java.lang.Thread class, and Java creates some threads on its own—for example, the main thread from which the main() function is called.

Every Java program is a multithreaded program. Even in a “Hello, World” program, where the user’s code clearly runs only in the main thread, the JVM creates a number of other threads to perform background duties like garbage collection.

In graphical programs, user code typically runs in several threads. Some of the code you write runs on the main thread (like the setup code you’ve written so far), while other code runs on the event thread. When a menu item is chosen or a button is clicked, the event handlers that are invoked run on the event thread. Sometimes this is unimportant, but when code on the event thread and code on the main thread need to coordinate their activities, you have to think clearly about what code is running where.

In the program you’re developing, the (run) function will be called on the main thread, and so the code for all the rules that fire will execute on the main thread. On the other hand, you want the user to click the OK button on the GUI. This action will trigger some code to run on the event thread, and you want to have that affect the behavior of the rules. The situation is depicted in figure 13.5.
You'll adopt the following strategy to get the main thread and the event thread to cooperate:

1. The main thread sets up the GUI to display the question.
2. The main thread sleeps until a reply is available.
3. When the user presses Enter or clicks the OK button, the event thread asserts a fact containing a reply.
4. The main thread wakes up and processes the reply. If the reply is invalid, go back to step 1. If it is valid, assert the `answer` fact and return from the module.

Jess contains a mechanism that makes this process simple to implement. There is a function you can call to make the current thread sleep when no activated rules are on the agenda of the current module, and then wake up and continue once a new activation arrives. No Jess language function lets you access this mechanism directly, however—you have to call it as a Java method.

### The class `jess.Rete`

Jess’s rule engine is embodied in a class named `jess.Rete`. Many of the most important functions you’ve seen so far in the Jess language—`run`, `reset`, `clear`,

<table>
<thead>
<tr>
<th>Main thread</th>
<th>Event thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>new JTextField(40);</td>
<td>(new JTextField 40)</td>
</tr>
<tr>
<td>(run-until-halt)</td>
<td>(run-until-halt)</td>
</tr>
<tr>
<td>Rete.runUntilHalt();</td>
<td>Rete.runUntilHalt();</td>
</tr>
<tr>
<td><code>rule fires</code></td>
<td><code>rule fires</code></td>
</tr>
<tr>
<td><code>ask-user</code></td>
<td><code>user-clicks-OK</code></td>
</tr>
<tr>
<td><code>action-performed();</code></td>
<td><code>read-input();</code></td>
</tr>
<tr>
<td></td>
<td><code>assert (user-input &quot;Text&quot;);</code></td>
</tr>
<tr>
<td></td>
<td><code>Rete.assert();</code></td>
</tr>
<tr>
<td></td>
<td><code>pattern-match user-input fact</code></td>
</tr>
</tbody>
</table>

Figure 13.5 Some of the code in a GUI application runs on the main thread, and other code runs on the event thread. The separate threads can communicate by calling Jess functions. In the diagram, time flows down. Note how calling a Jess function like `run` results in a Java function like `Rete.run()` being called a short time later.
assert—correspond to single Java method calls on instances of this class (run(), reset(), clear(), assertFact()). When you start jess.Main from the command line, a single instance of jess.Rete is created, and all of your program’s rules and facts belong to it. You can get a reference to this jess.Rete object using the engine function in Jess. The code

```jess
((engine) run)
```

is therefore more or less equivalent to

```jess
(run)
```

Now let’s return to our discussion of coordinating multiple threads in your application. The function you’re interested in is called waitForActivations. If this method is called when the current module’s agenda is empty, it doesn’t return right away. Rather, it uses the wait() method from the java.lang.Object class to wait for new activations to arrive. Note that the only way waitForActivations can return is for code on some other thread to call a function that modifies working memory. You can call waitForActivations from Jess like this:

```jess
((engine) waitForActivations)
```

Listing 13.3 shows the changed parts of a new version of the ask module that implements this idea. Whereas the old version of ask contained just one rule that asked the question and returned the answer, this new version contains two rules: one that sets up the question (ask-question-by-id) and one that validates the answer and either returns the answer or asks the question again (collect-user-input).

### Listing 13.3 ask module that expects GUI input as user-input facts

```jess
(deffunction ask-user (?question ?type ?valid)
  "Set up the GUI to ask a question"
  (bind ?s ?question "")
  (if (eq ?type multi) then
    (bind ?s (str-cat ?s ?*crlf* "Valid answers are "))
    (foreach ?item ?valid
      (bind ?s (str-cat ?s ?item "")))))
  (bind ?s (str-cat ?s ":"))
  (?*qfield* setText ?s)
  (?*afield* setText "")

(defrule ask::ask-question-by-id
  "Given the identifier of a question, ask it"
  (declare (auto-focus TRUE))
  (MAIN::question (ident ?id) (text ?text) (type ?type) (valid $?valid))
  (bind ?id (str-cat ?id "")))
```

```jess
((engine) waitForActivations)
```
Here’s what happens when the new ask module is used:

1. You assert a MAIN::ask fact giving the identifier for a question.
2. On the main thread, the rule ask-question-by-id is activated by the MAIN::ask fact and the MAIN::question fact with the given identifier. This rule has the auto-focus property, so the ask module immediately gets the focus.
3. ask-question-by-id calls the function ask-user and then uses waitForActivations to pause until another rule in this module is activated.
4. ask-user sets up the GUI to display the proper question, and clears the answer area.
5. Nothing happens until the user enters some text and presses Enter or clicks the OK button.
6. The event handler read-input, running on the event thread, asserts an ask::user-input fact containing the text entered by the user, as a symbol.
7. The ask::user-input fact, together with the MAIN::question fact and the MAIN::ask fact, activate the rule collect-user-input. The method waitForActivations finally returns due to this activation, and the right-hand side of ask-question-by-id completes.
8 Back on the main thread, collect-user-input asserts a MAIN::answer fact, retracts the MAIN::ask and ask::user-input facts, and returns from the ask module.

13.5 Better input components

As it stands, the user of your program has to answer questions by typing a reply into a JTextField. Of course, most GUIs don’t work that way. For example, users aren’t accustomed to typing x86 or Macintosh into a GUI; they’re used to selecting choices from a combo box. This application should work the same way.

Recall that the question fact template has a slot named type, indicating the kind of answer expected for each question. Your interface should display different user-input panels depending on the value of this type field. What panels will you need? Looking back at the questions defined in the previous chapter, you’ll need only two:

- Multiple choice—A panel with a pop-up menu (a combo box) containing a list of choices.
- Numeric—A text field that accepts numeric input. You’ve essentially already done this one.

Once again, the ask module can be modified to accommodate this latest requirement. You can change the ask-user function so that it sets up the appropriate input component for each question based on the type of question being asked. For questions with numeric answers, you use the existing JTextField, and for multiple-choice questions, you use a new JComboBox.

Setting up the JComboBox and its associated OK button is similar to setting up the JTextField. After the components are created, the JButton needs an event handler to assert the selected item as an answer fact:

```lisp
(deffunction combo-input (?EVENT)
"An event handler for the combo box"
(assert
 (ask::user-input (sym-cat (?*acombo* getSelectedItem))))
/bind ?handler (new jess.awt.ActionListener combo-input (engine))
(?*acombo-ok* addActionListener ?handler)
```
One interesting bit about this code snippet is the call to the JComboBox constructor. The constructor expects an array of Object as an argument. In this code, you pass a Jess list made by create$; Jess automatically converts it to the needed array.

With the JComboBox defined, you can modify ask-user one final time:

```
(deffunction ask-user (?question ?type ?valid)
  "Set up the GUI to ask a question"
  (?*qfield* setText ?question)
  (?*apanel* removeAll)
  (if (eq ?type multi) then
    (?*apanel* add ?*acombo*)
    (?*apanel* add ?*acombo-ok*)
    (?*acombo* removeAllItems)
    (foreach ?item ?valid
      (?*acombo* addItem ?item))
  else
    (?*apanel* add ?*afield*)
    (?*apanel* add ?*afield-ok*)
    (?*afield* setText "")
  )
  (?*frame* validate)
  (?*frame* repaint))
```

This version is somewhat simpler because it doesn’t have to compose a complex prompt string. Instead, based on the question type, it installs one of two possible sets of components in the JPanel at the bottom of the window. Figure 13.6 shows what the application looks like with the combo box panel installed.

### 13.6 Finishing touches

The current version of the recommend-action function still prints its recommendations to the command line. Java offers the convenient JOptionPane class as a quick way to post a modal dialog box. You can easily modify recommend-action to use it:
(deffunction recommend-action (?action)
  "Give final instructions to the user"
  (call JOptionPane showMessageDialog ?*frame*
    (str-cat "I recommend that you " ?action)
    "Recommendation"
    (get-member JOptionPane INFORMATION_MESSAGE)))

The arguments to the static showMessageDialog method are, in order, the parent window, the message, the window title, and a constant that indicates what kind of icon to display.

Another limitation of the application is that there’s no way to exit. You can tell the JFrame to call System.exit method when it is closed using JFrame’s setDefaultCloseOperation method:

(?(?*frame* setDefaultCloseOperation (get-member JFrame EXIT_ON_CLOSE))

Finally, during testing I noticed that it’s a bit disorienting to have question text remain in the window after the application has finished. It would be nice to have the old question removed when the program is through. You can’t put code to do this into recommend-action, because sometimes this function is called to make an intermediate recommendation, and more questions will follow. You certainly don’t want to go back and change all the rules to add GUI-specific code. What can you do?

If you look at your rules, you’ll recall that they call halt as appropriate to terminate execution of the system. You can therefore use the defadvice function to modify the behavior of halt (see section 4.5 to learn more about defadvice):

(defadvice before halt
  {?(?"qfield* setText "Close window to exit")})

Now, when a rule calls halt to halt execution, the contents of the question field will automatically change.

13.7 Testing the interface

Testing graphical applications automatically is difficult. There are commercial tools to help you do it, but most of them are fragile, hard to use, and expensive. The classic problem with automated GUI-testing tools is that they make your life harder rather than easier when you need to make changes to your interface, because changes to the GUI may force changes to the test. It’s easy to write tests for a GUI that are so fragile that they must be rewritten any time the GUI layout changes even a little. Tests that expect particular GUI components to lie at specific
screen coordinates are the most delicate in this respect. If you do use a GUI-testing tool, try not to write tests based on absolute screen coordinates. Tests that refer to components by name are much more robust.

You can use the Java class `java.awt.Robot` to write your own automated tests. This class lets you move the mouse pointer, press mouse buttons, and type text as if from the keyboard, all under programmatic control. I’ve personally found the combination of `java.awt.Robot` and Jess’s scripting capabilities to be particularly powerful.

Besides automated testing, you can of course do manual testing. If (as was the case here) you wrote and tested the underlying logic of the application first, testing the GUI by hand isn’t quite so bad. Rather than trying to test every path through the knowledge base, your GUI testing effort should be concentrated on trying every situation the GUI is meant to handle. For this application, you’d want to try both multiple-choice and numeric input, and some recommendations that call `(halt)` and some that don’t. You can also run through some of the same test scenarios you used with the command-line version of the application, entering the data into the GUI instead.

### 13.8 Summary

In part 4 of this book, you’ve built a fairly sophisticated application in about 230 lines of Jess code. You collected knowledge in the form of flowcharts and then translated the flowcharts into rules. The flowcharts are a compact and useful form of documentation for the resulting knowledge base.

The PC Repair Assistant uses backward chaining to automatically form hypotheses about possible problems and ask the appropriate questions, based just on the antecedents of the rules in the knowledge base. This approach scales well to large systems, unlike the more unwieldy approach used by the Tax Forms Advisor in part 3. Because you used backward chaining to let Jess choose the questions automatically, you needed only half as many rules compared to the tax advisor.

Once the application logic was written, tested, and debugged, you wrapped it in a graphical user interface using only Jess code. Adding a GUI required you to make some changes to the `ask` module used for asking questions, but you didn’t need to touch any of the rules. The final incarnation of the `ask` module uses Jess’s `waitForActivations` Java API function to communicate between Java’s event thread and the main thread where rules fire.

In this and the previous part, you developed complete, isolated applications from the ground up using Jess. In the real world, however, Jess usually needs to
cooperate with Java code outside of itself. In the next part of this book, you will learn how to put Java objects into Jess’s working memory so that Jess can reason about the outside world. You’ll also learn how to extend the Jess language by writing functions in Java.
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.

“... 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. Terlese, Ph.D. UCLA Brain Research Institute

---

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

*For non-commercial use*

---

Ernest Friedman-Hill

$49.95 US/$74.95 Canada

*ERNEST FRIEDMAN-HILL*

MANNING