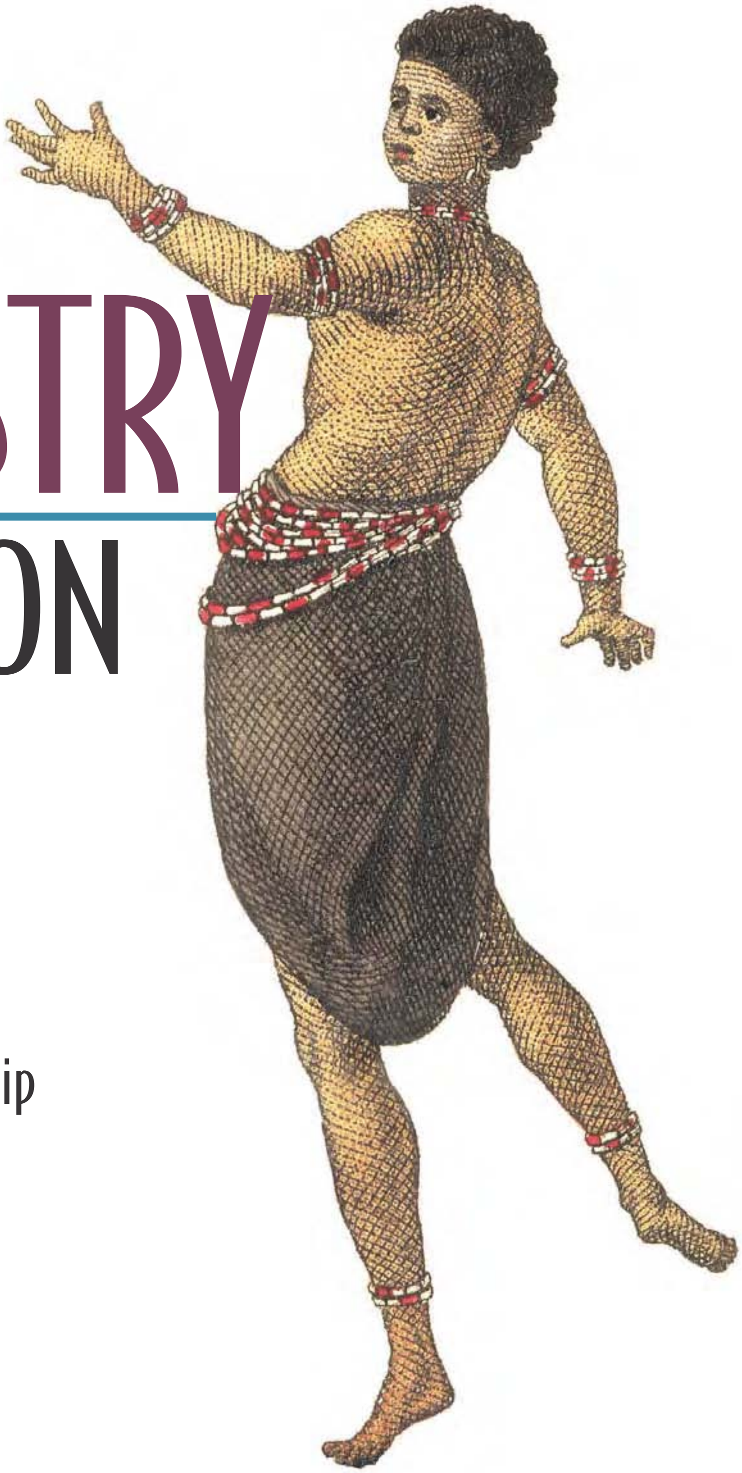
TAPESTRY IN ACTION

Howard M. Lewis Ship



MANNING

For online information and ordering of this and other Manning books, go to www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact:

Special Sales Department
Manning Publications Co.
209 Bruce Park Avenue
Greenwich, CT 06830

Fax: (203) 661-9018
email: orders@manning.com

Manning Publications Co.
209 Bruce Park Avenue
Greenwich, CT 06830

Copyeditor: Liz Welch
Typesetter: Denis Dalinnik
Cover designer: Leslie Haimes

ISBN 1-932394-11-7

Printed in the United States of America
1 2 3 4 5 6 7 8 9 10 – VHG – 08 07 06 05 04

brief contents

APPENDIXES

2 Getting started with Tapestry

This chapter covers

- Creating HTML templates, page specifications, and page classes
- Using Tapestry components inside an HTML template
- Creating clickable links
- Encoding extra information into link URLs
- Configuring Tapestry applications for deployment

In the first chapter, we made a number of claims about what Tapestry is capable of; now it is time to start backing up those claims with hard code. Launching into a complete Java 2 Enterprise Edition (J2EE) application right here would be a bit premature; instead, we'll start with more of a toy, an application that plays the simple word game Hangman.[†] In effect, Hangman is a "scale model" of a real Tapestry application; it demonstrates the basic capabilities of the framework and will give you an initial sense for what developing Tapestry applications is all about. Along the way, you'll see how to:

- Separate business logic from presentation logic, within the Model-View-Controller (MVC) pattern (described in chapter 1)
- Combine HTML templates, page specifications (in XML), and Java classes to form pages within the application
- Create HTML hyperlinks that activate application logic when clicked
- Encode custom application data into HTML hyperlinks
- Manage server-side state information
- Configure a Tapestry application for deployment inside a servlet container

More importantly, you'll see quite a bit about the work you *don't* have to do, because the framework takes care of it for you.

Appendix B covers how to obtain the source code for all the examples in the book, as well as how to build the examples on your own computer and deploy them into the Tomcat servlet container (Tomcat is an open source servlet container available from http://jakarta.apache.org/tomcat). Once Tomcat is running and you have downloaded the source code, you can launch the Hangman application by opening a web browser to http://localhost:8080/hangman1/app.

2.1 Introducing the Hangman application

Hangman is a simple word game for two players, played on a piece of paper or on a chalkboard. One player selects a secret target word; the other player attempts to guess the word. To start, you draw an empty gallows. The guessing player selects a letter from the alphabet; if the letter appears in the target word, the other player writes the letter in each position of the target word that the letter appears in. Each unsuccessful guess is marked by adding a line to a stick figure

[†] An even simpler example of a "Hello World" Tapestry application is available at http://www.manning.com/lewisship/helloworld. The helloworld.war file is pre-compiled and pre-built, containing all the necessary Tapestry libraries and deployment descriptors. It may be downloaded directly into your servlet container, Tomcat or otherwise, and accessed as http://localhost:8080/helloworld/app.

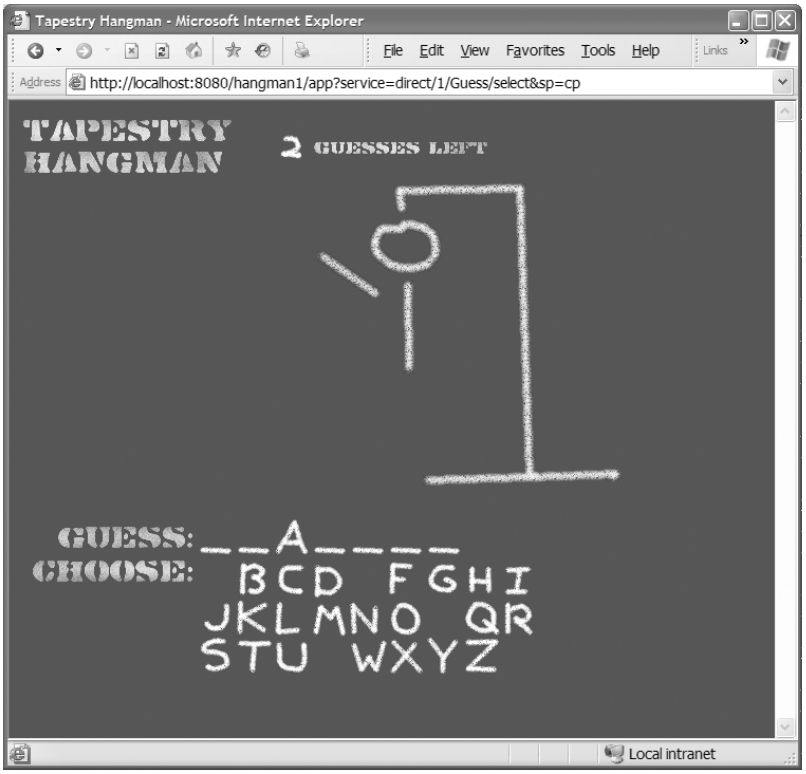

Tapestry Hangman - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Links
Address http://localhost:8080/hangman1/app?service=direct/1/Guess/select&sp=cp
TAPESTRY HANGMAN
2 GUESSES LEFT
GUESS: _ _ A _ _ _ _ _
CHOOSE: BCD FGHI JKLMNO QR STU WXYZ
Local intranet

Figure 2.1 A Tapestry Hangman game in progress. The player has successfully guessed the letter A, but has also guessed E, P, and V, which are not in the target word. An important aspect of this application is the look and feel, which should resemble a game played by hand on a chalkboard.

on the gallows: the head, torso, and then the limbs. The game is over when the word is guessed or the stick figure is completed.

The Tapestry Hangman application captures all of this functionality and, at the same time, attempts to capture the classic look of playing the game by hand on a chalkboard. The user interface makes use of images to represent the letters and other artifacts of the game, to provide a "hand–scrawled" look and feel. Figure 2.1 shows the middle of a game of Hangman; the player has made several wrong guesses, so parts of the stick figure are filled in, and one letter (A) has been guessed correctly so far.

At this point, all we have is a general idea for the application; before we can get to the coding stage, we must formalize this general idea into something a bit more concrete—and that begins with identifying the application flow.

2.1.1 Determining the application flow

The *application flow* is a model of how the end user will navigate through the application. Determining the flow occurs very early in the development cycle; it is driven by the specific requirements and use cases of the application. Application flow is the most abstract model of the application; it identifies the different pages in the application and how they are connected, but rarely has to precisely identify what is on any particular page. Key aspects of the application user interface are discernable from the flow diagrams, such as the need for common navigation menus or specific links between individual pages.

The flow of the Hangman application is quite simple: From the Guess page, the user makes guesses at the target word, eventually winning or losing the game. Figure 2.2 is a state diagram for this simple application; when the application is launched, the user is presented with a Start page (figure 2.4); from there, he or she can start a new game, making guesses that eventually reach either a win or a loss; from there, the player can restart the game with a new target word.

From this simple description, you can see that we'll have four distinct pages in the application:

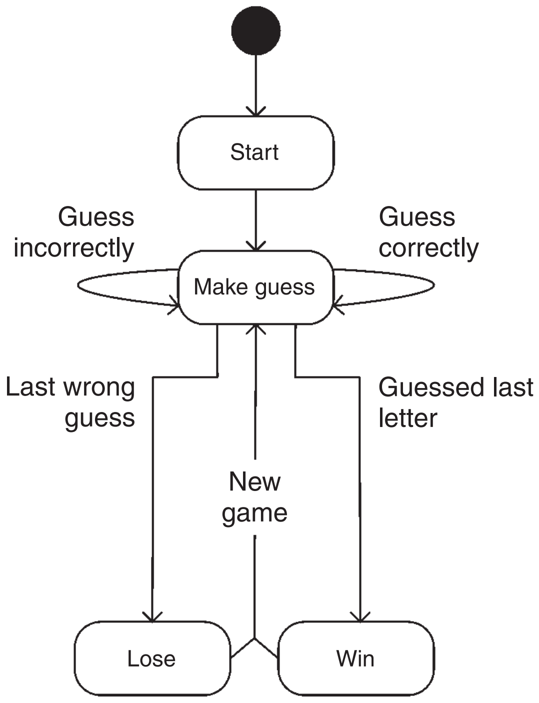


Figure 2.2
The player starts the game and makes guesses, eventually reaching the win or lose page, from which the player can start a new game (with a new word).

- **Start**—A welcome page to greet players before starting a new game
- **Make Guess**—The main page, from which players may guess letters of the target word
- **Win**—The page reached after the target word is successfully guessed
- **Lose**—The page reached after players have exhausted their guesses

Once the application flow has been determined, the next step is to prototype what the individual HTML pages will look like.

2.1.2 Creating page mockups

Page mockups are static HTML pages that represent what the active pages from the running application will look like. These are ordinary HTML pages with placeholder values representing the content that will eventually be generated dynamically by the application. The point of creating the mockups is to give the HTML developers a chance to work out the look and feel of the application, right down to fonts, colors, and graphics, without concern for how the application will be implemented.

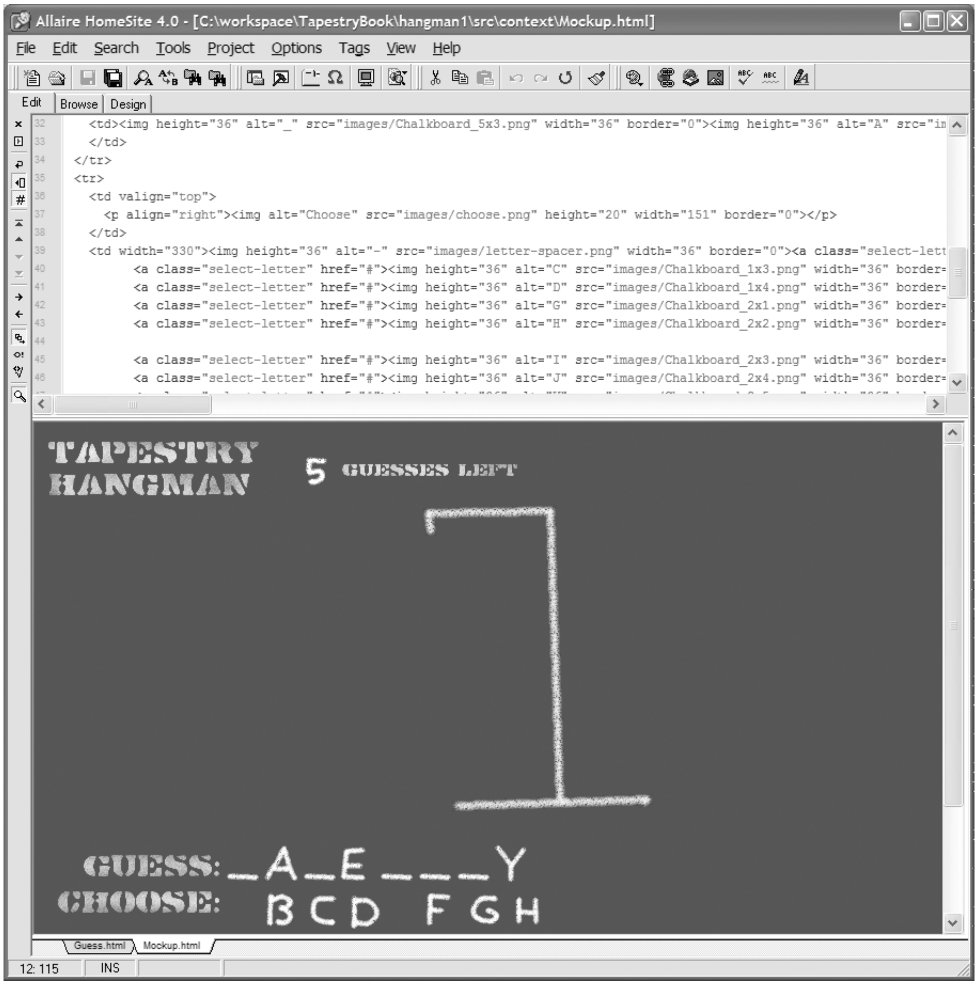
Figure 2.3 shows the mockup for the Guess page in an HTML editor. The HTML source is shown in the upper pane, and the WYSIWYG preview appears in the lower pane. This mockup will eventually be converted for use as the Guess page's HTML template.

Page mockups should display *all* the features of the running application, especially such features as error messages that are included only conditionally. For example, a mockup may include a snippet for an error message:

```
<img src="images/error.png" width="32" height="32" align="top"/>
<span class="error-message">
  Placeholder for error message.
</span>
```

This snippet is important for two reasons: It clearly identifies how a real error message should be displayed, and it identifies exactly *where* within the page the error message should be displayed. In the Guess page mockup, the Guess and Choose sections demonstrate what the page looks like in the middle of a game, with some letters of the target word filled in and several letters from the alphabet already guessed. Having clear examples of these dynamic aspects of the page will be invaluable to the Java developer when he or she is converting the mockup into a usable HTML template.

It is not an absolute requirement that you create a mockup for every page in the application; often, mockups for only a handful of key pages will suffice, and


Allaire HomeSite 4.0 - [C:\workspace\TapestryBook\hangman1\src\context\Mockup.html]
File Edit Search Tools Project Options Tags View Help
Edit Browse Design

```
    <td><img height="36" alt="_" src="images/Chalkboard_5x3.png" width="36" border="0"><img height="36" alt="A" src="in
    </td>
  </tr>
  <tr>
    <td valign="top">
      <p align="right"><img alt="Choose" src="images/choose.png" height="20" width="151" border="0"></p>
    </td>
    <td width="330"><img height="36" alt="-" src="images/letter-spacer.png" width="36" border="0"><a class="select-lett
          <a class="select-letter" href="#"><img height="36" alt="C" src="images/Chalkboard_1x3.png" width="36" border=
          <a class="select-letter" href="#"><img height="36" alt="D" src="images/Chalkboard_1x4.png" width="36" border=
          <a class="select-letter" href="#"><img height="36" alt="G" src="images/Chalkboard_2x1.png" width="36" border=
          <a class="select-letter" href="#"><img height="36" alt="H" src="images/Chalkboard_2x2.png" width="36" border=

          <a class="select-letter" href="#"><img height="36" alt="I" src="images/Chalkboard_2x3.png" width="36" border=
          <a class="select-letter" href="#"><img height="36" alt="J" src="images/Chalkboard_2x4.png" width="36" border=
```

Figure 2.3 The page mockup for the Guess page in an HTML editor.

developers can use these core mockups as templates for the remaining application pages.

As you'll see shortly, converting these HTML mockup pages into usable Tapestry page templates requires a minimal number of unobtrusive changes. A mockup is converted into a page template by adding *instrumentation*: Additional tags and tag attributes are used to identify and configure Tapestry components within the template. This instrumentation is designed to be nearly invisible. Tapestry's approach stands in stark contrast to the use of JSPs, where the conversion from HTML mockup to JavaServer Page (JSP) is a one-way process. Once the HTML mockup page has been converted to a JSP, it will not preview correctly in a standard HTML editor, making all subsequent changes to the JSP that much more difficult. Within Tapestry, a page template can still be edited by an HTML developer using standard HTML editing tools; in effect, the mockups evolve into the HTML page templates yet can still be treated as mockups.[1]

This is an important aspect of Tapestry because late changes to application flow and look and feel are simply a reality when creating web applications—there's always a last-minute change: a new page to add, a background color to change, or a column width to tweak. Even in an impossibly idealized project, one where no late changes ever occurred, a subsequent release of the application would inevitably update the application flow and at least some aspect of the look and feel. Tapestry accommodates these kind of late cycle changes quite well because of how unobtrusive the instrumentation (the additional tags and tag attributes used to identify components within a template) is. Much more work can be done by an HTML developer using standard HTML editing tools, without the involvement of Java developers.

Meanwhile, even as the HTML developers are working on the mockups, the Java developers should be getting a head start on the design of the actual application, and that begins with identifying the domain objects.

2.1.3 Defining the domain objects

The architects and developers on the Java side of the team are ultimately responsible for the running application; in most applications, this becomes a question of linking a user interface to your domain objects. *Domain objects* are the objects of the middle tier, the application tier, in the overall application—they are the entity objects for data stored in a database, or objects that implement your business's specific processes. Common problems to solve involve what information is stored by these objects, how the different objects are related, and how they are read from, or stored into, a database.

Even in a simple application such as Hangman, which does not make use of a database, there are still domain objects, and still advantages (in accordance with the MVC design pattern) to keeping these objects well separated from any code directly related to the user interface.

[1] Tapestry isn't magic, and there are some limitations on maintaining full WYSIWYG previews of HTML templates once more sophisticated custom components are created and used within a page template; this subject is covered in chapter 6.

The two domain objects used in the Hangman application are `WordSource` and `Game`. The first, `WordSource`, is simply a wrapper around a list of words read from a text file and is used to dole out random words for the player to guess. `Game` is a bit more interesting; it encompasses all the logic about the game. Specifically, the `Game` object knows:

- The target word the player is attempting to guess
- Which of the 26 letters of the alphabet the player has already guessed
- Which letters of the target word have been filled in by successful guesses
- How many incorrect guesses remain
- If the player has won the game (by guessing all the letters in the target word)

As promised, the implementation of the `Game` class (in listing 2.1) knows nothing about Tapestry or any other user interface.

Listing 2.1 Game.java: domain object for the Hangman application

```
package hangman1;

public class Game
{
  private String _targetWord;
  private int _incorrectGuessesLeft;
  private char[] _letters;
  private boolean[] _guessed = new boolean[26];
  private boolean _win;

  public boolean isWin()                  Returns true
  {                                       once word has
    return _win;                          been guessed
  }

  public char[] getLetters()                Returns array
  {                                         of letters in
    return _letters;                        the word
  }

  public int getIncorrectGuessesLeft()
  {
    return _incorrectGuessesLeft;
  }

  public boolean[] getGuessedLetters()           Returns 26 flags:
  {                                              letters guessed
    return _guessed;                             by player
  }
```

```
public void start(String word)
{
  _targetWord = word;
  _incorrectGuessesLeft = 5;
  _win = false;

  int count = word.length();

  _letters = new char[count];

  for (int i = 0; i < count; i++)
    _letters[i] = '_';

  for (int i = 0; i < 26; i++)
    _guessed[i] = false;
}
```

Starts a new game

```
public boolean makeGuess(char letter)
{
  char ch = Character.toLowerCase(letter);

  if (ch < 'a' || ch > 'z')
    throw new IllegalArgumentException(
      "Must provide an alphabetic character.");

  int index = ch - 'a';

  if (_guessed[index])
    return true;

  _guessed[index] = true;

  boolean good = false;
  boolean complete = true;

  for (int i = 0; i < _letters.length; i++)
  {
    if (_letters[i] != '_')
      continue;

    if (_targetWord.charAt(i) == ch)
    {
      good = true;
      _letters[i] = ch;
      continue;
    }

    complete = false;
  }

  if (good)
```

Processes a player's guess

```
      {
        _win = complete;

        return !complete;
      }

      if (_incorrectGuessesLeft == 0)
      {
        _letters = _targetWord.toCharArray();

        return false;
      }

      _incorrectGuessesLeft--;

      return true;
    }
  }
```

Processes a player's guess

The `makeGuess()` method is invoked to process a player's guess. It updates the target word and other properties and returns true if more guesses are allowed. It returns false if the player has either won or lost the game.

The `Game` class must provide some support for the user interface, but it does so in a generic fashion without being tied to the interface; it's the Model in the MVC pattern described in chapter 1. This support takes the form of JavaBeans properties that are exposed to the user interface, such as the number of incorrect guesses remaining or the list of letters already guessed. These properties are bound to Tapestry component parameters, allowing those components to display the number of guesses remaining, the partially guessed word, or the list of remaining unguessed letters. In addition, `Game` provides methods that can be invoked by the user interface code to start a new game or to process a guess made by the player.

A second class, `WordSource`, is also used. `WordSource` is responsible for providing a random word for the player to guess. The source of the words is a small file, WordList.txt, packaged with the `WordSource` class. The `WordSource` class is provided in listing 2.2.

Listing 2.2 WordSource.java: domain object for the Hangman application

```
package hangman1;

import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.LineNumberReader;
import java.io.Reader;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class WordSource
{
  private int _nextWord;
  private List _words = new ArrayList();

  public WordSource()
  {
    readWords();
  }

  private void readWords()
  {

    try
    {
      InputStream in =
        getClass().getResourceAsStream("WordList.txt");
      Reader r = new InputStreamReader(in);
      LineNumberReader lineReader = new LineNumberReader(r);

      while (true)
      {
        String line = lineReader.readLine();

        if (line == null)
          break;

        if (line.startsWith("#"))
          continue;

        String word = line.trim().toLowerCase();

        if (word.length() == 0)
          continue;

        _words.add(word);
      }

      lineReader.close();
    }
    catch (IOException ex)
    {
      throw new RuntimeException(
        "Unable to read list of words from file WordList.txt.",
        ex);
    }

    // Randomize the word order

    Collections.shuffle(_words);

  }

  public String nextWord()
  {
    if (_nextWord >= _words.size())
    {
      _nextWord = 0;
      Collections.shuffle(_words);
    }

    return (String) _words.get(_nextWord++);
  }
}
```

When `WordSource` is instantiated, it reads the list of words. Later, the `nextWord()` method is invoked to get a new word for the player to guess. The method is designed to not repeat a target word until every word in the list has been guessed.

As with the `Game` class, this class has no direct connection to Tapestry—these objects fit firmly into the Model category within the MVC pattern. This kind of decoupling from the user interface is very important, because it means the `Game` and `WordSource` classes can be tested without having to run the Tapestry application, which in turn means the code can be fully tested inside an automated test suite. Making code testable is always a worthy goal, because no matter how simple the code is, *when you write tests, you find bugs*.

Once all the details of the domain objects are worked out, the next step is to begin work on the pages that will interact with those domain objects.

2.1.4 Defining the pages

Like any other Tapestry application, the Hangman game consists of a set number of pages, which are themselves composed of components. In a Tapestry application, each page is constructed by combining three related artifacts: an HTML template, a page specification, and a Java class.[2]

[2] Refer back to section 1.5.1 to see how to properly package these artifacts for use within a servlet container. Appendix B provides examples of how to set up your development workspace and how to use Ant to build and deploy the WAR.

Each Tapestry page has a specific, unique name. The page name is used to locate the page specification and HTML template. Part of the page specification is the name of the Java class to instantiate; this is called the *page class*, and it will include properties and methods specific to your application.

The Hangman application contains only four pages: Home, Guess, Lose, and Win, corresponding to the four pages identified in the application flow state diagram (figure 2.2). The Home page here is the same as the Start page in figure 2.2. By default, when a Tapestry application is first launched, the framework renders the page named Home. Although there are several options for changing this behavior, the simplest approach is to follow Tapestry's naming convention—by naming the first page a user will see Home.

Creating a functioning Tapestry page starts with the HTML mockup for the page. This mockup must be *instrumented* to act as an HTML template instead of a mockup. Instrumenting a mockup inserts additional attributes and tags in the mockup that tell Tapestry which parts of the template are dynamic components. Most of a template, however, is exactly the same as the mockup—simple, static HTML.

NOTE In real projects, the mockups are not always available when needed by the Java developers creating the pages. In this situation, the Java developers will create simple, minimal HTML templates—just enough to wire up the functionality of the application. When the mockup is ready, some careful cut and paste from the mockup into the minimal HTML template will convert it to use the desired application look and feel.

Once the HTML template is instrumented, a *page specification* (a short XML document) can be created. The page specification has a number of responsibilities (many of which will be discussed in later chapters). Its most basic responsibility is to identify which Java class is to be instantiated as the page. In chapter 1, we described Tapestry as being a component object framework; this means that each component fits into an object hierarchy, either as a container of other components or as a containee of a specific component—or, in many cases, as both container *and* containee. Pages are still components, sitting at the root of the component object hierarchy.

As you'll see, the page class is specific to the application and contains a mixture of properties and methods that support both the rendering of the page and any user interaction in the page. Ultimately, the behavior of the page is defined by the page's properties and methods, combined with the components contained within the page—including the templates, properties, and methods of those components. This may seem a bit dizzying in theory, but in practice it all comes together simply and seamlessly. For our first example, let's start with the Home page—the simplest page in the Hangman application.

2.2 Developing the Home page

The Hangman application's Home page has only one small bit of user interaction: a link that starts a new game. This interaction is triggered by clicking the Start image, shown in figure 2.4. Like any page, the Home page is a combination of an XML page specification, an HTML template, and a Java class. Our first steps into Tapestry will be to examine how these three artifacts are combined to form a simple page.

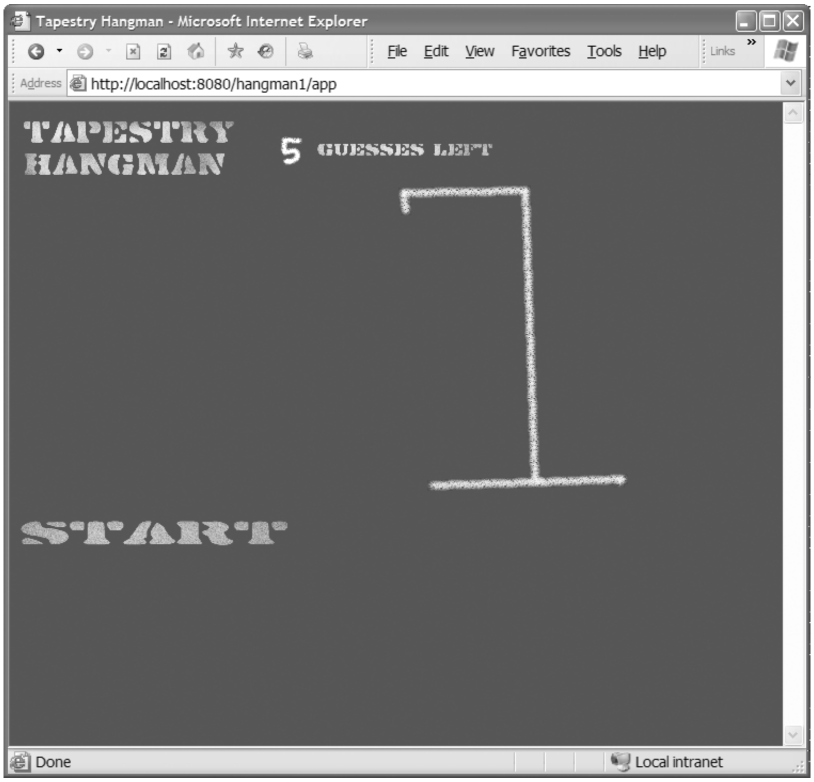

Tapestry Hangman - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Links
Address http://localhost:8080/hangman1/app
TAPESTRY HANGMAN
5 GUESSES LEFT
START
Done
Local intranet

Figure 2.4 The Home page of the Tapestry Hangman application. The player may click the word *Start* to begin a game.

The Home page is displayed when the application is first launched. The Web archive (WAR) for the application must be deployed into the servlet container, and the servlet container must itself be running. This WAR will contain the Tapestry framework JARs, the page templates and specifications, the static image files (and other assets), and the compiled Java classes (this is discussed in chapter 1, section 1.5.1). When the user launches the application (by opening a web browser to http://localhost:8080/hangman1/app), the framework responds by rendering the Home page.

The first step in rendering a page is to create an instance of the page. The framework reads the Home page's specification and HTML template and uses this information to create the page instance. A Tapestry page is not a single object; the page object is the root of a tree of objects, including Tapestry components from the page's template, the contents of the HTML template, and a number of objects used to connect the individual pieces together. There's no special assembly stage for Tapestry applications, nor are there any special build steps or compilation—all that is necessary is to package the specifications, templates, and Java classes inside the WAR.

NOTE You might be concerned about performance, given all this talk of parsing specifications and templates and instantiating trees of objects—but don't be. This parsing occurs very quickly, and, unlike with JSPs, there's no time spent compiling generated Java source code (JSP compilation causes a noticeable delay the first time a JSP is used within a traditional servlet application). In line with Tapestry's efficiency goal, all the specifications and templates are read and parsed just once, and then cached for fast access when needed again in future requests. Page instances are also stored and reused in later requests.

Let's dive a little deeper and see exactly how the Home page's specification is used by the framework.

2.2.1 Understanding the Home page specification

The framework's first step toward instantiating the Home page is to locate and read the page's specification. Page specifications are validated XML files (with a .page extension) that are stored in the WEB-INF folder of the web application. The page specification's first responsibility is to identify the page class it needs to instantiate—it has other many other, optional responsibilities that we'll cover later in this chapter and in subsequent chapters. Listing 2.3 contains the complete specification for the Home page of the Hangman application.

Listing 2.3 Home.page: specification for the Home page

```
<?xml version="1.0"?>
<!DOCTYPE page-specification PUBLIC
  "-//Apache Software Foundation//Tapestry Specification 3.0//EN"
  "http://jakarta.apache.org/tapestry/dtd/Tapestry_3_0.dtd">

<page-specification class="hangman1.Home"/>
```

This is about as simple as a page specification can get; its only purpose is to identify the page class, `hangman1.Home`. This is a Java class written for the Hangman application, which will be the runtime representation of the page (see section 2.2.3 for more details). By convention, the class name for a page is the same as the page's name (though often stored inside a Java package), but of course, you are free to ignore this convention and name pages and classes differently. It's important, however, that the `<!DOCTYPE>` declaration be exactly as shown in listing 2.3.

WARNING Use the correct `<!DOCTYPE>`. Tapestry uses a validating XML parser to read specifications. Tapestry is purposely finicky about the public ID (the first string after PUBLIC), since it uses the known public ID to access a copy of the document type definition (DTD) inside the framework's JAR rather than access it over the Internet using the system ID. The public ID must exactly match the value in listing 2.3, or an `ApplicationRuntimeException` is thrown. For example, changing *Foundation* to *Floundation* will result in an exception report with this error message: *Document context:/WEB-INF/Home.page has an unexpected public id of '-//Apache Software Floundation//Tapestry Specification 3.0//EN'.* Watch out for typos; this is one area where a little cut and paste will save you some grief.

In addition, there is nothing that keeps a single page class from being used for multiple pages. Each page will have a distinct instance of the page class, just as each component in a page is a distinct instance of a component class.

2.2.2 *Rendering the Home page*

After parsing the page specification, Tapestry locates the HTML template for the Home page. The HTML template, which is named Home.html, is located in the root of the web application archive. This template is shown in listing 2.4.

Listing 2.4 Home.html: HTML template for the Home page

```
<html>
<head>
<title>Tapestry Hangman</title>
<link rel="stylesheet" type="text/css" href="css/hangman.css"/>
</head>
<body>
<table>
<tr>
  <td><img alt="[Tapestry Hangman]"
    src="images/tapestry-hangman.png" width="197" height="50"
    border="0"/>
  </td>
  <td width="70" align="right"><img height="36" alt="5"
    src="images/Chalkboard_3x8.png" width="36" border="0"/>
  </td>
  <td><img alt="Guesses Left" src="images/guesses-left.png"
    width="164" height="11" border="0"/>
  </td>
</tr>
<tr>
  <td>
  </td>
  <td>
  </td>
  <td><img alt="" src="images/scaffold.png" border="0"/>
  </td>
</tr>
</table>
<br/>
<a href="#" jwcid="@DirectLink"
  listener="ognl:listeners.start">
    <img src="images/start.png" width="250" height="23"
      border="0" alt="Start"/></a>
</body>
</html>
```

Dynamic portion of template

The majority of the HTML template is standard, static HTML; only a single Tapestry extension beyond ordinary HTML is used, showing up in the portion of the template that provides the link to start the game.

The `<a>` tag declares a Tapestry component within the template, giving us our first whiff of a dynamic web application rather than a static web page. The attribute `jwcid` is the indicator that Tapestry uses to identify components within the template. The name `jwcid` is simply *Java Web Component ID*. The component is type `DirectLink`, one of over 40 components provided with the Tapestry framework.

The example here is an *implicit* component, where the type of component and its configuration are declared directly in the HTML template. The `@` symbol indicates to Tapestry that the component is implicitly declared. Later in this chapter, we'll show examples of *declared* components, which have their type and configuration stored inside the page specification.

The DirectLink component is used to create a particular type of callback into the application. This component is one of the two primary ways that interaction occurs in Tapestry; the other is user-submitted forms (which are covered starting in chapter 3). The DirectLink component renders an HTML `<a>` element, supplying a URL that, when clicked by the end user, causes a specific listener method of the page to be executed (we'll discuss what a listener method is shortly, in section 2.2.3).

The position of the DirectLink component within the template is delineated by the `<a>` and `</a>` tags. Everything else in this HTML template is static HTML—text that is sent through to the client web browser unchanged. Just the portion rendered by the DirectLink component is dynamic. Figure 2.5 shows how the dynamic and static portions of the template are integrated together to form the complete response.

The Home page's HTML template is divided into five individual "chunks." Each chunk is either a block of static HTML, the start tag for a component (recognized by Tapestry because of the presence of a `jwcid` attribute), or the matching end tag for a component. Chunk ❶ is the portion of the HTML template that precedes the DirectLink component. Chunk ❷ is the component itself. Chunk ❸ is the portion of the page enclosed by the DirectLink. Chunk ❹ is the close tag for the DirectLink component. Chunk ❺ is the remainder of the template after the DirectLink.

Chunks that are enclosed directly within a component's start and end tags are part of that component's body. This is a very important part of Tapestry: Components control if and when their bodies are rendered. We'll frequently refer to the body of the component: This is the static HTML and other components that are enclosed between a component's start and end tags.

In this example, chunk ❸, containing the `<img>` tag, is the entire body of the DirectLink component. The page itself has a body, the top-level static chunks (chunks ❶ and ❺) and the components that aren't enclosed by other components

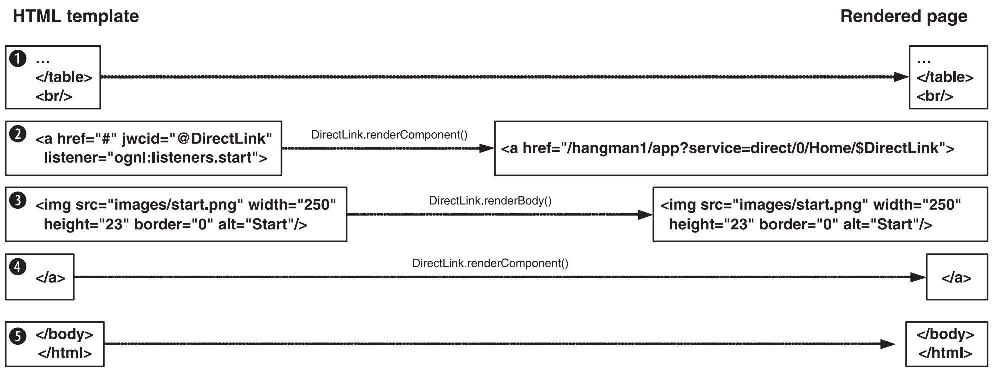

HTML template
Rendered page
❶ ... </table> <br/>
... </table> <br/>
❷ <a href="#" jwcid="@DirectLink" listener="ognl:listeners.start">
DirectLink.renderComponent()
<a href="/hangman1/app?service=direct/0/Home/$DirectLink">
❸ <img src="images/start.png" width="250" height="23" border="0" alt="Start"/>
DirectLink.renderBody()
<img src="images/start.png" width="250" height="23" border="0" alt="Start"/>
❹ </a>
DirectLink.renderComponent()
</a>
❺ </body> </html>
</body> </html>

Figure 2.5 The Home page template is broken into chunks of static HTML and component tags. Static HTML chunks render as themselves; the DirectLink renders in code, in its `renderComponent()` method, and causes its body (the `<img>` tag) to render by invoking its `renderBody()` method.

(chunk ❷). When the page renders, it renders just the chunks in its body. Static HTML chunks render as themselves; they are passed on through to the client web browser unchanged. Components are responsible for rendering themselves and their body.

Figure 2.5 references two methods related to the DirectLink component: `renderComponent()` and `renderBody()`. The `renderComponent()` method is implemented by components that render in Java code (rather than using their own template). The method is invoked by the component's container, in this case the Home page itself, as part of the Home page's render.

The second method, `renderBody()`, is inherited by the DirectLink component from the `AbstractComponent` base class. The component invokes this method from its own `renderComponent()` method to render the text and components in its own body—the static `<img>` tag enclosed by the DirectLink's `<a>` and `</a>` tags.

In this case, the body of the DirectLink is simple, static HTML. That's often not the case; a component may contain a mix of static HTML and other components. Tapestry figures it all out, properly slotting each chunk of the page's template into the body of the correct component. Rendering a page is a recursive process, since components may themselves have their own templates, containing other components. Chapters 6 and 8 go into great detail about creating new components, including components that have their own template.

Tapestry's HTML template parser is very forgiving; although the examples in this book all follow Extensible HTML (XHTML) conventions, the template parser can handle the kind of HTML you'll find in the wild: unquoted attribute values, mixed uppercase and lowercase, single or double quotes, unquoted attribute values, and lots of additional whitespace. As elsewhere in Tapestry, if the parser is unable to parse a template it will throw an exception providing line-precise reporting of the problem.

The last piece of the Home page puzzle is the page class; this is where we put our application-specific logic—the code that will actually start a new game.

2.2.3 Defining the Home page class

So, what happens when the user clicks the link that was created when the page rendered? In Tapestry, that's the million-dollar question,[3] the point where all this talk of simplicity, consistency, and components starts to make a difference. Here's the short answer: You tell the component about a method in your page class to execute, and it executes the method when the link is clicked. Now, let's see what this looks like in practice. We'll start with listing 2.5, the source code for the Home page class.

Listing 2.5 Home.java: Java class for the Home page

```
package hangman1;

import org.apache.tapestry.IRequestCycle;
import org.apache.tapestry.html.BasePage;

public class Home extends BasePage
{
  public void start(IRequestCycle cycle)
  {
    Visit visit = (Visit)getVisit();

    visit.startGame(cycle);
  }
}
```

A page class has many responsibilities defined by the framework, including the ability to act as a container of other components. Fortunately, the `BasePage` class, from which the `Home` class extends, contains the code needed to fulfill all these responsibilities; for the Home page, all we need to add is the little bit of application-specific logic to be executed when the Start link is clicked. That logic shows up as a method, `start()`, implemented by the Home page class.

[3] Since Tapestry is open source, money is not the best way to gauge status. Perhaps this should be the "million download question" instead!

The `start()` method is a *listener method*, a method that will be invoked in response to a user clicking a particular link. Its implementation is to defer to the `Visit` object to actually start a new game—we'll discuss what the `Visit` object is shortly; for the moment, we'll concentrate on how it is that the `start()` method is invoked when a user clicks the link.

Listener methods are ordinary instance methods, implemented by the page's class, that have a specific method signature:

```
public void method(IRequestCycle cycle)
```

The method must always be public, return void, and take a single parameter of type `IRequestCycle`.

Tapestry components may have any number of parameters, both optional and required. The DirectLink component has several optional parameters and one that's required (`listener`). The binding for the `listener` parameter was provided in the Home page's HTML template:

```
<a href="#" jwcid="@DirectLink"
  listener="ognl:listeners.start">
  . . .
</a>
```

TIP Tapestry checks that there's a binding for each required parameter. If you remove the `listener` attribute from the HTML template for the Home page, the page will not display. Instead, you'll get an exception report with this message: *Required parameter listener of component Home/$DirectLink is not bound.* Home/$DirectLink is the name of the page and the ID of the component.

The DirectLink component's `listener` parameter is used to find the listener method it should execute when the end user clicks the link visible in his or her web browser. The `ognl:` prefix on the attribute value informs Tapestry that the value is an Object Graph Navigation Language (OGNL) expression to be evaluated, rather than a literal string constant. In Tapestry terminology, the expression `listeners.start` is *bound* to the DirectLink's `listener` parameter.

WARNING Don't forget the `ognl:` prefix. If you omit the prefix, Tapestry treats the value as a string literal. Removing the prefix from the DirectLink's `listener` parameter will result in an error like this when you click the link: *Parameter listener (listeners.start) is an instance of java.lang.String, which does not implement interface org.apache.tapestry.IActionListener.* When you see exceptions such as this, or perhaps `ClassCastExceptions` within your own code, the likely cause is a missing `ognl:` prefix.

How does the OGNL expression `listeners.start` end up executing this method? All pages and components inherit a property, `listeners`, from the `AbstractComponent` base class. The `listeners` property contains a nested property for each listener method implemented by the class. Underneath the covers, there's an interface, `IActionListener`, and a little bit of Java reflection used to connect the DirectLink component with the page's listener method; this is shown in figure 2.6.

A class may have any number of listener methods, each with a unique and individual name. Listener methods inherited from superclasses are also available through the `listeners` property.

WARNING If your OGNL expression references a listener method that doesn't exist, you'll get an exception when you click the link. For example, changing the expression to `ognl:listeners.star` results in an exception with this message: *Unable to resolve expression 'listeners.star' for hangman1.Home@19b808a[Home].* You'll also see an `ognl.NoSuchPropertyException` for the property `star`.

An invalid listener method will result in the same exception: This will occur if the method is not public or has the wrong method signature.

Sit back and think about this for a moment: We've just extended the behavior of this page within the application by writing a very short method, the `start()` listener method. The provisions we've made in the HTML template to get this

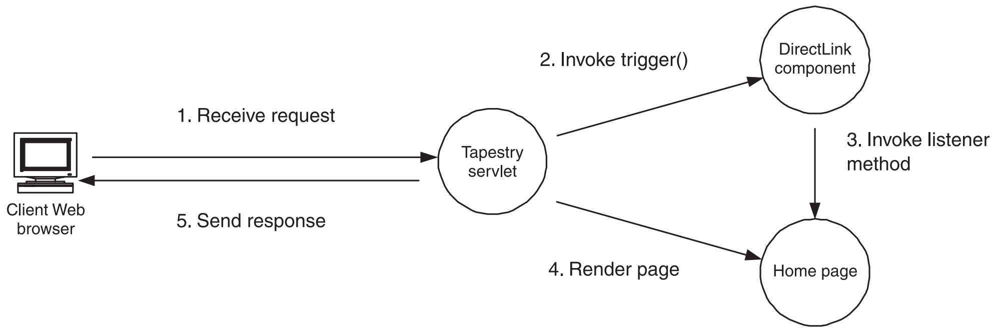


Figure 2.6 The Tapestry servlet receives and interprets the incoming request and invokes `trigger()` on the DirectLink component. The DirectLink invokes the listener method provided by the page. After the method is invoked, a page is rendered, forming the HTML response sent back to the client web browser.

listener method to execute on cue are so minor that they're barely worth considering. The Hangman application's Home page is unusual in that it has just the single bit of behavior—but you can imagine a more complicated page with many links (and, as you'll see in chapter 3, forms); adding each new bit of behavior is still just…adding another listener method.

This gets to the heart of the Tapestry goals described in chapter 1:

- **Simplicity**—Adding new operations takes minimal code and minimal changes to the HTML template.
- **Consistency**—Add as few or as many operations as you like, and the process stays the same. Look at any page in the application, and it *still* looks the same.
- **Feedback**—By working with the framework, errors in Java code, in the template, or in the specification are detected and verbosely reported by the framework.

A good practice is to keep listener methods short and focused on simply interfacing Tapestry components with business logic stored in domain objects. That's demonstrated here by having the `start()` listener method simply find the `Visit` object and let *it* do the work of actually starting a new game.

2.2.4 Examining the Visit object

The `Visit` object is an application-wide space for storing application logic and data. This object is accessible from all pages and components within the application and contains information specific to a single client of the web application. A single `Visit` object instance is shared by all pages within the application. The object fulfills much the same role as the `HttpSession` does in a typical servlet application, and in fact, the `Visit` object is ultimately stored as an `HttpSession` attribute.

All web applications eventually store some form of client-specific server-side state. The `HttpSession` acts like a map, storing named attributes. Simple as this seems, in real applications, a considerable amount of code must be written to retrieve attribute values from the `HttpSession`, cast them to the right type, create them on the fly as needed, and delete them when they are no longer needed.

Here again, Tapestry steps in to rethink this approach in terms of objects, methods, and properties. In chapter 7, we'll cover how Tapestry allows page properties to be stored persistently between requests, which is appropriate for values that are used only within a single page.

For more general data, used throughout an application, Tapestry allows for a single Visit object. Tapestry doesn't know or care about the type of the Visit object. There is no specific Visit class defined by the framework; each application defines its own Visit class. The accessor method for the Visit object provided by the page (defined by the interface IPage and implemented by the class BasePage) doesn't specify the type of the object:

```
public Object getVisit();
```

It then becomes a matter of casting to the application-specific type:

```
Visit visit = (Visit)getVisit();
```

The Visit object is automatically created by the framework the first time it is referenced; you must configure Tapestry, providing the name of the class to instantiate (this may be configured inside the web deployment descriptor; see section 2.6). Once the Visit object is created, it is stored in the HttpSession for persistent access in later requests.

Developer code never has to worry about the HttpSession. The HttpSession itself is created only as needed. A stateless application is more efficient than a stateful one, and a Tapestry application will operate in a stateless mode until there is actual server-side state to store. The framework takes care of this transition automatically, which would be very cumbersome to accomplish in ordinary servlet code because each and every servlet would need custom logic to check for the existence of the session and create it only as needed.

For our Hangman application, the Visit object is responsible for controlling page flow. It acts as a façade around the WordSource and Game objects, handles the process of starting a new game, and processes guesses made by the player. The Visit class for the Hangman application is provided in listing 2.6.

Listing 2.6 Visit.java: controller object for the Hangman application

```
package hangman1;

import org.apache.tapestry.IRequestCycle;

public class Visit
{
  private WordSource _wordSource = new WordSource();
  private Game _game = new Game();

  public void startGame(IRequestCycle cycle)
  {
    _game.start(_wordSource.nextWord());
```

1 Invoked by Home page listener method

```
    cycle.activate("Guess");            ❶
  }

  public void makeGuess(IRequestCycle cycle, char ch)
  {
    if (_game.makeGuess(ch))
      return;

    cycle.activate(_game.isWin() ? "Win" : "Lose");
  }

  public Game getGame()
  {
    return _game;
  }
}
```

❷ **Invoked by Guess page listener method**

❸ **Provides game property**

❶ The `startGame()` method is invoked by a listener method on the Home page to start a new game. It is also invoked by listener methods on the Win and Lose pages.

❷ The `makeGuess()` method is invoked by a listener method on the Guess page; the listener method passes in the character to be guessed and the request cycle (so that the `Visit` object can activate the Win or Lose page, if necessary).

❸ The `Game` object is exposed as a read-only property of the `Visit` object. You'll see references to the properties of the `Game` object in the template as `ognl:visit.game.`*`property`*.

When the Home page invokes the `startGame()` method on `Visit`, `Visit` gets a random word and sets up the Game instance with it by invoking `Game`'s `start()` method. The call to `activate()` is used to change the active application page; the active page is responsible for rendering the response. Initially, the Home page is the active page, because it contains the `DirectLink` component that was triggered. Invoking the `activate()` method allows the correct page, the Guess page, to render the response.

2.3 Implementing the Home page using standard servlets

Despite the fact that the previous discussion about the DirectLink component, listener methods, and the `Visit` object was unavoidably long-winded, in the end we've shown that creating a link and getting an application-specific method to execute when the link is clicked is extremely simple.

Let's see what would be involved in accomplishing the same thing using standard servlets and JSPs. In this simple example, the JSP is very straightforward—so much so that it could as easily be an entirely static HTML page. The DirectLink component is replaced by a standard HTML link to a servlet we'll provide:

```
<a href="startGame"> <img . . . /> </a>
```

Of course, this example is not representative. Most application operations will involve quite a bit more: more servlets to implement the operation, more query parameters to fill in the details, and more code to build and interpret the URLs—all things that the Tapestry framework provides you for free.

Regardless, this example uses a very simple operation with no parameters. We still need to add a few lines to the application's web deployment descriptor, web.xml:

```
<servlet>
  <servlet-name>startGame</servlet-name>
  <servlet-class>StartGameServlet</servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>startGame</servlet-name>
  <url-pattern>/startGame</url-pattern>
</servlet-mapping>
```

Finally, we need the actual servlet, shown in listing 2.7.

Listing 2.7 StartGameServlet.java: hypothetical servlet for starting a game

```
import java.io.IOException;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

public class StartGameServlet extends HttpServlet
{
  protected void doGet(HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, IOException
  {
    HttpSession session = request.getSession(true);    <-1 Gets or creates the session

    Visit visit = new Visit();
    session.setAttribute("visit", visit);    <-2 Stores Visit for later
```

```
      visit.startGame();          <-3 Chooses random
                                       word to guess
      RequestDispatcher d =
        request.getRequestDispatcher("/Guess.jsp");    4 Forwards to
      d.forward(request, response);                       the Guess.jsp
    }                                                     page
  }
```

❶ This accesses an existing `HttpSession` for the client or creates a new one if necessary.

❷ We store the `Visit` object as a session attribute so that it can be accessed in later requests or by a JSP.

❸ As in the Tapestry Hangman application, the hypothetical servlet `Visit` object is responsible for selecting a random word to guess.

❹ The `RequestDispatcher` object is used to bridge from the servlet to a JSP that can render the response.

This servlet creates a `Visit` instance, similar in scope and implementation to the `Visit` object used in the Tapestry application. Once created, the `Visit` object is stored in the `HttpSession`, where it will be available in subsequent requests. The implementation of this `Visit` class may use the same `Game` and `WordSource` domain objects used by the real Tapestry application.

Extending this comparison from one single interaction to the innumerable interactions in a typical web application underscores the amount of developer effort needlessly wasted on most web applications. You are forced to drop out of the world of objects and methods and deal directly with aspects of the HTTP protocol and the Servlet API. You must define a URL to trigger your operation, create a new servlet class to perform that operation, and record the mapping from the URL to the servlet in the web.xml deployment descriptor. In a team environment, you will be competing with your fellow developers to update the deployment descriptor and to lay claim to the possible URLs for the application.

Certainly, as you become more experienced writing servlet-based applications, you will find shortcuts to help you streamline this effort. Unfortunately, different developers are quite likely to create their own suite of shortcuts. In a large team effort, getting the bits and pieces of the application written by different developers interoperating properly can become quite a challenge because of the impedance caused by all of the developers' individual schemes. When using Tapestry, this is rarely an issue because Tapestry defines a standard way for different parts of the application to interoperate—using objects, methods, and properties.

Now that we've seen how the Home page and the Hangman application's `Visit` object work together to start a new game, we can continue to the Guess page, the primary page in the Hangman application.

2.4 Developing the Guess page

The Guess page is the central page for the Hangman application; it allows the player to guess at letters of the target word. Figure 2.1 shows an example of the Guess page in action.

The page has a number of responsibilities:

- It displays the number of guesses remaining (as a number) as well as the number of incorrect guesses so far (as the growing stick figure).
- It displays the partially guessed target word, with lines replacing the as-yet unguessed letters.
- It displays a grid of remaining letters to guess; each letter is a clickable link.
- It supports the "hand-scrawled" look and feel, using custom images to display numbers and letters.

To accomplish all these tasks, we'll be introducing several new concepts for Tapestry specifications, HTML templates, and Java classes, as well as new Tapestry components. We'll start with the full listings for the HTML template, the page specification, and the page class, and then show how the different responsibilities we've listed are implemented—as Tapestry markup in the HTML template combined with entries in the page specification and code in the Java class. We'll begin with listing 2.8, the HTML template for the Guess page.

Listing 2.8 Guess.html: HTML template for the Guess page

```
<html>
<head>
<title>Tapestry Hangman</title>
<link rel="stylesheet" type="text/css" href="css/hangman.css"/>
</head>
<body>
<table>
<tr>
  <td><img alt="Tapestry Hangman" src="images/tapestry-hangman.png"
    width="197" height="50" border="0"/>
  </td>
  <td width="70" align="right"><img jwcid="@Image"
    alt="ognl:visit.game.incorrectGuessesLeft"
    image='ognl:getAsset("digit" +                                ❶
      visit.game.incorrectGuessesLeft)'
    height="36" src="images/Chalkboard_3x8.png" width="36"
    border="0"/>
  </td>
  <td><img alt="Guesses Left" src="images/guesses-left.png"
    width="164" height="11" border="0"/>
  </td>
</tr>
<tr>
  <td>
  </td>
  <td>
  </td>
  <td><img jwcid="@Image"
    image='ognl:getAsset("scaffold" +
      visit.game.incorrectGuessesLeft)'                            ❷
    alt="[Scaffold]" src="images/scaffold.png" border="0"/>
  </td>
</tr>
</table>
<br>
<table>
<tr valign="center">
  <td width="160">
    <p align="right"><img alt="Current Guess"
    src="images/guess.png" align="middle" width="127" height="20"
    border="0"/></p>
  </td>
  <td><span jwcid="@Foreach" source="ognl:visit.game.letters"
    value="ognl:letter"><img jwcid="@Image"
    image="ognl:letterImage" alt="ognl:letterLabel" height="36"   ❸
    src="images/Chalkboard_5x3.png" width="36"
    border="0"/></span>

    <span jwcid="$remove$">
    <!--- Additional letters from the mockup --->
    <img height="36" alt="A" src="images/Chalkboard_1x1.png"
      width="36" border="0"/><img height="36" alt="_"
      src="images/Chalkboard_5x3.png" width="36"
      border="0"/><img height="36" alt="_"
      src="images/Chalkboard_1x5.png" width="36"                  ❹
      border="0"/><img height="36" alt="_"
      src="images/Chalkboard_5x3.png" width="36"
      border="0"/><img height="36" alt="_"
      src="images/Chalkboard_5x3.png" width="36"
      border="0"/><img height="36" alt="_"
      src="images/Chalkboard_5x3.png" width="36"
      border="0"/><img height="36" alt="_"
```

```
   src="images/Chalkboard_5x1.png" width="36"
   border="0"/>                                       4
  </span>
 </td>
</tr>
<tr>
 <td valign="top">
  <p align="right"><img alt="Choose" src="images/choose.png"
   height="20" width="151" border="0"/></p>
 </td>
 <td width="330"><span jwcid="selectLoop"><a href="#"
   jwcid="select" class="select-letter"><img jwcid="@Image"
   image="ognl:guessImage" alt="ognl:guessLabel" height="36"   5
   src="images/Chalkboard_5x3.png" width="36"
   border="0"/></a></span>

  <span jwcid="$remove$">
  <!-- Additional selectable letters from the mockup. --->

   <a class="select-letter" href="#"><img height="36" alt="B"
    src="images/Chalkboard_1x2.png" width="36" border="0"/></a>
   <a class="select-letter" href="#"><img height="36" alt="C"
    src="images/Chalkboard_1x3.png" width="36" border="0"/></a>
   <a class="select-letter" href="#"><img height="36" alt="D"
    src="images/Chalkboard_1x4.png" width="36" border="0"/></a>
   <img height="36" alt="-" src="images/letter-spacer.png"
    width="36" border="0"/>
   <a class="select-letter" href="#"><img height="36" alt="F"
    src="images/Chalkboard_1x6.png" width="36" border="0"/></a>
   <a class="select-letter" href="#"><img height="36" alt="G"
    src="images/Chalkboard_2x1.png" width="36" border="0"/></a>
   <a class="select-letter" href="#"><img height="36" alt="H"
    src="images/Chalkboard_2x2.png" width="36" border="0"/></a>
   <a class="select-letter" href="#"><img height="36" alt="I"      6
    src="images/Chalkboard_2x3.png" width="36" border="0"/></a>
   <a class="select-letter" href="#"><img height="36" alt="J"
    src="images/Chalkboard_2x4.png" width="36" border="0"/></a>
   <a class="select-letter" href="#"><img height="36" alt="K"
    src="images/Chalkboard_2x5.png" width="36" border="0"/></a>
   <a class="select-letter" href="#"><img height="36" alt="L"
    src="images/Chalkboard_2x6.png" width="36" border="0"/></a>
   <a class="select-letter" href="#"><img height="36" alt="M"
    src="images/Chalkboard_3x1.png" width="36" border="0"/></a>
   <a class="select-letter" href="#"><img height="36" alt="N"
    src="images/Chalkboard_3x2.png" width="36" border="0"/></a>
   <a class="select-letter" href="#"><img height="36" alt="O"
    src="images/Chalkboard_3x3.png" width="36" border="0"/></a>
   <a class="select-letter" href="#"><img height="36" alt="P"
    src="images/Chalkboard_3x4.png" width="36" border="0"/></a>
   <a class="select-letter" href="#"><img height="36" alt="Q"
    src="images/Chalkboard_3x5.png" width="36" border="0"/></a>
```

```
        <a class="select-letter" href="#"><img height="36" alt="R"
          src="images/Chalkboard_3x6.png" width="36" border="0"/></a>
        <a class="select-letter" href="#"><img height="36" alt="S"
          src="images/Chalkboard_4x1.png" width="36" border="0"/></a>
        <a class="select-letter" href="#"><img height="36" alt="T"
          src="images/Chalkboard_4x2.png" width="36" border="0"/></a>
        <a class="select-letter" href="#"><img height="36" alt="U"
          src="images/Chalkboard_4x3.png" width="36" border="0"/></a>
        <a class="select-letter" href="#"><img height="36" alt="V"
          src="images/Chalkboard_4x4.png" width="36" border="0"/></a>     ❻
        <a class="select-letter" href="#"><img height="36" alt="W"
          src="images/Chalkboard_4x5.png" width="36" border="0"/></a>
        <a class="select-letter" href="#"><img height="36" alt="X"
          src="images/Chalkboard_4x6.png" width="36" border="0"/></a>
        <img height="36" alt="-" src="images/letter-spacer.png"
          width="36" border="0"/>
        <a class="select-letter" href="#"><img height="36" alt="Z"
          src="images/Chalkboard_5x2.png" width="36" border="0"/></a>
      </span>
    </td>
  </tr>
  </table>
  </body>
  </html>
```

❶ This Image component selects and displays the correct image identifying the number of incorrect guesses remaining to the player.

❷ The second Image component selects and displays an image for the man on the scaffold, showing how many incorrect guesses the player has made so far.

❸ These components display the target word, with underscores marking unguessed letters within the word.

❹ This portion of the template is marked for removal (using the special `$remove$` value for the `jwcid` attribute). The `<img>` tags within the `<span>` exist for WYSIWYG preview but must be removed because they conflict with the dynamic content provided by ❸.

❺ These components provide an array of clickable letters, allowing the player to guess the next letter in the target word.

❻ This portion of the template is also marked for removal.

This page was converted directly from the HTML mockup; the bulk of the template consists of placeholder values (for the number of guesses, for the stick figure, for the partially guessed word, and for the grid of guessable letters) that will actually be discarded in favor of dynamically generated HTML. We'll go into more detail on each portion of the HTML template shortly.

Listing 2.9 is the page specification for the Guess page.

Listing 2.9 Guess.page: specification for the Guess page

```
<?xml version="1.0"?>
<!DOCTYPE page-specification PUBLIC
  "-//Apache Software Foundation//Tapestry Specification 3.0//EN"
  "http://jakarta.apache.org/tapestry/dtd/Tapestry_3_0.dtd">

<page-specification class="hangman1.Guess">

  <component id="selectLoop" type="Foreach">
    <binding name="source" expression="visit.game.guessedLetters"/>
    <binding name="value" expression="letterGuessed"/>
    <binding name="index" expression="guessIndex"/>
  </component>
                                                                       ❶
  <component id="select" type="DirectLink">
    <binding name="listener" expression="listeners.makeGuess"/>
    <binding name="parameters" expression="letterForGuessIndex"/>
    <binding name="disabled" expression="letterGuessed"/>
  </component>

  <context-asset name="digit0" path="images/Chalkboard_1x7.png"/>
  <context-asset name="digit1" path="images/Chalkboard_1x8.png"/>
  <context-asset name="digit2" path="images/Chalkboard_2x7.png"/>    ❷
  <context-asset name="digit3" path="images/Chalkboard_2x8.png"/>
  <context-asset name="digit4" path="images/Chalkboard_3x7.png"/>
  <context-asset name="digit5" path="images/Chalkboard_3x8.png"/>

  <context-asset name="scaffold5" path="images/scaffold.png"/>
  <context-asset name="scaffold4" path="images/scaffold-1.png"/>
  <context-asset name="scaffold3" path="images/scaffold-2.png"/>    ❸
  <context-asset name="scaffold2" path="images/scaffold-3.png"/>
  <context-asset name="scaffold1" path="images/scaffold-4.png"/>
  <context-asset name="scaffold0" path="images/scaffold-5.png"/>

  <context-asset name="space" path="images/letter-spacer.png"/>
  <context-asset name="dash" path="images/Chalkboard_5x3.png"/>

  <context-asset name="a" path="images/Chalkboard_1x1.png"/>
  <context-asset name="b" path="images/Chalkboard_1x2.png"/>
  <context-asset name="c" path="images/Chalkboard_1x3.png"/>
  <context-asset name="d" path="images/Chalkboard_1x4.png"/>        ❹
  <context-asset name="e" path="images/Chalkboard_1x5.png"/>
  <context-asset name="f" path="images/Chalkboard_1x6.png"/>
  <context-asset name="g" path="images/Chalkboard_2x1.png"/>
  <context-asset name="h" path="images/Chalkboard_2x2.png"/>
  <context-asset name="i" path="images/Chalkboard_2x3.png"/>
  <context-asset name="j" path="images/Chalkboard_2x4.png"/>
  <context-asset name="k" path="images/Chalkboard_2x5.png"/>
```

```
    <context-asset name="l" path="images/Chalkboard_2x6.png"/>
    <context-asset name="m" path="images/Chalkboard_3x1.png"/>
    <context-asset name="n" path="images/Chalkboard_3x2.png"/>
    <context-asset name="o" path="images/Chalkboard_3x3.png"/>
    <context-asset name="p" path="images/Chalkboard_3x4.png"/>
    <context-asset name="q" path="images/Chalkboard_3x5.png"/>
    <context-asset name="r" path="images/Chalkboard_3x6.png"/>
    <context-asset name="s" path="images/Chalkboard_4x1.png"/>     4
    <context-asset name="t" path="images/Chalkboard_4x2.png"/>
    <context-asset name="u" path="images/Chalkboard_4x3.png"/>
    <context-asset name="v" path="images/Chalkboard_4x4.png"/>
    <context-asset name="w" path="images/Chalkboard_4x5.png"/>
    <context-asset name="x" path="images/Chalkboard_4x6.png"/>
    <context-asset name="y" path="images/Chalkboard_5x1.png"/>
    <context-asset name="z" path="images/Chalkboard_5x2.png"/>

  </page-specification>
```

1. The `<component>` element is used to declare components. The type of component and the configuration of the component's parameters go here, in the page specification.
2. The `<context-asset>` element defines an asset file that is stored within the web application context. This first set of assets includes the digits used to display the number of remaining incorrect guesses. These assets are given logical names that are referenced in the Java page class.
3. The second group of `<context-asset>` elements defines the images used for the stick figure.
4. The remaining `<context-asset>` elements define the images used for the letters of the alphabet, as well as a blank space image and the underscore image (as `dash`).

This is a much longer specification than for the Home page, and it demonstrates a couple of new features: the ability to define the type and configuration of components in the specification rather than in the HTML template, and the ability to define *assets*, which are named references to static files such as images or stylesheets. Again, we'll revisit the relevant portions of this specification shortly.

Finally, listing 2.10 is the source for the `Guess` class: the Java class for the Guess page.

Listing 2.10 Guess.java: Java class for the Guess page

```
  package hangman1;

  import org.apache.tapestry.IAsset;
  import org.apache.tapestry.IRequestCycle;
```

```
import org.apache.tapestry.html.BasePage;

public class Guess extends BasePage
{
  private char _letter;
  private boolean _letterGuessed;
  private int _guessIndex;

  public void initialize()
  {
    _letter = 0;
    _letterGuessed = false;                 ❶ Resets page
    _guessIndex = 0;                           properties
  }

  public char getLetter()
  {
    return _letter;
  }

  public void setLetter(char letter)
  {
    _letter = letter;
  }

  public String getLetterLabel()
  {
    return ("" + _letter).toUpperCase();
  }

  public IAsset getLetterImage()
  {
    if (_letter == '_')                             ❷ Converts letter
      return getAsset("dash");                         property to an
                                                       image
    return getAsset("" + _letter);
  }

  public boolean isLetterGuessed()
  {
    return _letterGuessed;
  }

  public int getGuessIndex()
  {
    return _guessIndex;
  }

  public void setLetterGuessed(boolean letterGuessed)
  {
    _letterGuessed = letterGuessed;
  }
```

```
    public void setGuessIndex(int guessIndex)
    {
      _guessIndex = guessIndex;
    }

    public IAsset getGuessImage()
    {
      if (_letterGuessed)
        return getAsset("space");

      String name = "" + getLetterForGuessIndex();

      return getAsset(name);
    }

    public char getLetterForGuessIndex()
    {
      return (char) ('a' + _guessIndex);
    }

    public String getGuessLabel()
    {
      if (_letterGuessed)
        return " ";

      char ch = Character.toUpperCase(getLetterForGuessIndex());

      return new Character(ch).toString();
    }

    public void makeGuess(IRequestCycle cycle)
    {
      Object[] parameters = cycle.getServiceParameters();
      Character guess = (Character) parameters[0];

      char ch = guess.charValue();

      Visit visit = (Visit) getVisit();

      visit.makeGuess(cycle, ch);
    }
  }
```

3 Converts guessIndex property to an image

4 Listener method invoked when a letter is clicked

1 This method is invoked when the page is created and at the end of each request, to reset any properties back to pristine values, ready for the next request.

2 This method creates a read-only, synthetic property, `letterImage`, that provides the correct image for whatever the `letter` property currently is.

3 Likewise, this `guessImage` property returns the correct image based on the `guessIndex` and `letterGuessed` properties.

❹ This listener method is invoked when a letter image is clicked; it exists to determine the correct parameters to pass to the `Visit` object's `makeGuess()` method.

`Guess` is a typical Tapestry page class; it contains properties and methods that support the rendering of the page as well as listener methods activated by links on the page.

2.4.1 Displaying the remaining guesses

The first dynamic bit is the part of the HTML template that displays the number of incorrect guesses remaining to the player:

```
<img jwcid="@Image"
   alt="ognl:visit.game.incorrectGuessesLeft"
   image='ognl:getAsset("digit" +
     visit.game.incorrectGuessesLeft)'
   height="36"
   src="images/Chalkboard_3x8.png"
   width="36" border="0"/>
```

This snippet has an array of responsibilities:

- It must render an HTML `<img>` tag and fill in a number of attributes dynamically.
- It must convert the `incorrectGuessesLeft` property of the `Game` object into a string, as the `alt` attribute.
- It must select the correct image file to display the number of guesses left and build a URL to that file (as the `src` attribute).

Earlier we saw how the DirectLink component on the Home page inserted an `<a>` tag into the response sent to the client web browser. The `Image` component, another standard Tapestry component, is actually much simpler; it inserts an `<img>` tag, generating the tag's `src` attribute from its `image` parameter. Here we want it to provide the correct image (one of the hand-drawn digits) and the corresponding `alt` value.

NOTE To support WYSIWYG editing, the HTML template uses an `<img>` tag, knowing that the component will, at runtime, render an `<img>` tag. The `Image` component will override the `src` attribute in the template, which is also here just to help with the WYSIWYG preview of the template.

In a Tapestry template, each component must have properly balanced start and end tags. An alternative, used here, is to include an XML-style empty tag, one that ends with `/>`. Tapestry is flexible about attribute quoting; because the `image` parameter's expression uses double quotes, the entire expression is enclosed in single quotes.

WARNING Match your open and close tags. You must supply a matching close tag for each component's start tag. Tapestry even checks that all the start tags and end tags on a page properly nest (it is forgiving for all tags that aren't components). Changing the end of the `<img>` tag from `/>` to just `>` will result in the following exception: *Closing tag </td> on line 13 is improperly nested with tag <img> on line 12.* Tapestry matched the `</td>` on line 13 with the `<td>` on line 12 (before the `<img>` tag) and realized that the `<img>` tag hadn't yet been closed, even though it's a component.

The first OGNL expression, `visit.game.incorrectGuessesLeft`, is very straightforward; it retrieves the `incorrectGuessesLeft` property from the `Game` object (via the `Visit` object). The `incorrectGuessesLeft` property (a number) is converted to a string and becomes the value for the `<img>` tag's `alt` attribute. In the client web browser, this value becomes the tooltip for the image and is also used for accessibility (visually impaired users may have the value read to them by their computer).

The other expression, for selecting the image is more complicated. It also obtains the `incorrectGuessesLeft` property, but then it uses that value as a parameter when invoking the `getAsset()` method on the page. This underscores why OGNL is so useful and powerful; without OGNL, this access and manipulation would have to occur in Java code. Using OGNL, we are able to assemble the complete string and invoke a Java method, `getAsset()`, on our page, all in one step. The invoked method returns the asset object representing the image to use, which is ultimately converted into a URL by the Image component and inserted in the HTML response as the `src` attribute of the `<img>` tag.

NOTE Using OGNL expressions where possible allows you to assume a rapid application development cycle, free from the normal edit/compile/deploy cycle that occurs with Java code. You can simply edit your templates and specifications in place to see changes.[4] Later, you can recode

[4] It is possible to disable the normal caching that occurs inside Tapestry so that templates and specifications are reread for each new request. This allows changes to templates and specifications to take effect immediately. Consult the Tapestry reference documentation, distributed with the framework, for the details.

OGNL expressions as Java methods for greater application efficiency. Another good option, when not prototyping, is to move nontrivial OGNL expressions into the page specification (an example of this is shown in section 2.4.3); this results in a much improved separation of the View from the Model and application logic, which ultimately yields a more maintainable application.

Tapestry allows you, as the developer, to decide how pure a separation between the View and the Model you will maintain. At one extreme, the pragmatic view, you may put as much logic (in the form of OGNL expressions) as you want directly into the HTML template. This pushes together purely presentation-oriented aspects of the application (such as layout and fonts) with the behavioral aspects of the application (shown in this example as references to page properties, including `visit` and `visit.game`). Such an approach is perfectly acceptable for prototypes, or for small projects where a strong separation between developers isn't realistic. Most of the examples in this book use this pragmatic approach simply because it puts related information side by side, making it easier to comprehend.

At the other extreme, the purist view, your HTML template contains only placeholders for components; all details about the component configuration are stored outside the template, in the page specification. This is critical on larger projects, where a division can be expected between the HTML developers responsible for page mockups and the Java developers responsible for converting the mockups into a working application. Minimizing how much of the application's implementation is exposed to the HTML developers reduces the potential for conflicts between the Java developers and the HTML developers.

Assets are any kind of file that may be distributed as part of the WAR; the most common types of assets are images and stylesheets. The Image component's `image` parameter expects an asset object (an object that implements the `IAsset` interface), not a string, and this pairs up with the `getAsset()` method, which returns just such an object. The `getAsset()` method is inherited from the `AbstractComponent` base class; it allows access to the named assets defined in the page specification.

The names of the assets come from the `<context-asset>` elements in the page specification (in listing 2.9). What's happening is a mapping from a logical name (such as x or dash) to a particular file (such as images/Chalkboard_4x6.png or images/Chalkboard_5x3.png). The assets abstraction has some other important uses related to localization and to packaging components into reusable libraries. Those uses are covered in more detail in chapters 6 and 7.

Defining assets in the page specification

The page specification for the Guess page declares assets for the letters, digits, and underscore as well as all the images of the stick figure on the gallows. The Guess page specification includes the following lines to declare the six digits used in the user interface:

```
<context-asset name="digit0" path="images/Chalkboard_1x7.png"/>
<context-asset name="digit1" path="images/Chalkboard_1x8.png"/>
<context-asset name="digit2" path="images/Chalkboard_2x7.png"/>
<context-asset name="digit3" path="images/Chalkboard_2x8.png"/>
<context-asset name="digit4" path="images/Chalkboard_3x7.png"/>
<context-asset name="digit5" path="images/Chalkboard_3x8.png"/>
```

WARNING Tapestry checks that a file matching the provided asset path exists.[5] This check occurs when the page specification is first read and takes place regardless of whether anything ever *uses* the asset. Putting a typo into one of the names in the previous snippet results in the following exception: *Unable to locate asset 'digit0' of component Guess as context:/images/Challkboard_1x7.png.*

Here, we can see how the aliasing is useful. The letters and numbers were initially drawn onto a grid, and a slicing tool was used to generate a set of individual files from the cells of the grid. The filenames provided by the slicing tool are not intuitive (they are based on the position in the grid, rather the value of the image, and so are somewhat arbitrary), but the use of assets allows the code to reference them using more friendly names. Of course, we could have simply renamed the files output by the slicing tool, but by leaving the names as is, we can change the original letter grid image and then use the same slicing tool to regenerate all the images without having to go through the painful renaming process a second time. Tapestry has provided a little bit of abstraction and flexibility that ultimately makes the build process for this application more agile, because an annoying manual step (renaming the files) is not necessary.

Assets also provide a separation of concerns, dividing the HTML developers from the Java developers. For example, an HTML developer may decide to redo the graphics for the page and use a new tool to generate the images of the digits—which would result in new filenames, possibly even new types (perhaps GIF or JPEG), but no change to the logical names of the assets. Either the HTML

[5] This applies to the context assets defined here and the private assets we'll discuss in chapter 6. A third asset type, the external asset, is not checked.

developer or the Java developer would need to update the page specification to change the filenames, but there would be no change to the HTML template or even to the Java class (if the Java class ever accessed any assets by name).

Now that we have a way of mapping from logical names to actual asset files, we still need a way to figure out which logical name, and thus, which asset, should be used when displaying the remaining guesses.

Calculating the right asset

Displaying the digit image is a matter of selecting the correct asset as the `image` parameter to the Image component. This occurs in the HTML template using an OGNL expression:

```
image='ognl:getAsset("digit" + visit.game.incorrectGuessesLeft)'
```

Here, OGNL has done something fairly complex: building up the name of the asset and invoking the page's `getAsset()` method. There are penalties, however: This chunk of text is somewhat unwieldy and forces us to use single quotes, since the expression itself contains double quotes. Putting OGNL expressions into your template, especially expressions of this complexity, is not much better than putting Java scriptlets into a JSP: Such OGNL expressions strongly tie together the presentation of the page with the implementation.

One option would be to move more of this logic into equivalent Java code. This can be easily accomplished by referencing a new, read-only property in the HTML template:

```
<IMG jwcid="@Image"
   alt="ognl:visit.game.incorrectGuessesLeft"
   image="ognl:digitImage"          <-- Reference to the page's
   height="36"                          digitImage property
   src="images/Chalkboard_3x8.png"
   width="36" border="0"/>
```

We would then implement an accessor method for this new `digitImage` property in the `Guess` class:[6]

```
public IAsset getDigitImage()
{
  Visit visit = (Visit)getVisit();
  int guessesLeft = visit.getGame().getIncorrectGuessesLeft();

  return getAsset("digit" + guessesLeft);
}
```

[6] Because this approach is only hypothetical, you won't see this method in the `Guess` class in listing 2.10.

Another option, which we'll explore shortly, is to move the OGNL expression into the page specification. The decision to use OGNL expressions, Java code, or some mix of the two is left to you, according to your personal taste and the particular situation. The modest runtime performance penalty for using OGNL is easily offset by increased developer productivity.

Using informal component parameters

If you check the description for the Image component in appendix C, you'll see that it defines two possible parameters: a required `image` parameter and an optional `border` parameter. However, if you run the application and view the source of the page, you'll see that the other attributes included in the `<img>` tag in the template (`alt`, `width`, and `height`) are still present in the `<img>` tag rendered by the Image component. How can this be?

The majority of Tapestry components, including Image and DirectLink, allow *informal parameters*. Informal parameters are additional parameters for the component beyond those that are formally declared by the component. These additional parameters are simply added to the rendered tag as additional attributes. Informal parameters can be unevaluated static values, such as for `width`, or expressions, such as for `alt`. Some informal parameters are discarded so that they don't conflict with attributes rendered directly by the component. For example, it doesn't matter that the template provides a value for the `src` attribute (in the `<img>` tag for the Image component); the value in the template is discarded because the Image component will itself generate an `src` attribute from the asset provided in the `image` parameter. The `src` value in the template exists to support WYSIWYG previewing of the template; its value is discarded in favor of the real, dynamic URL computed on the fly in the live application. Only components that map directly to an HTML tag will accept informal parameters; each component indicates within its own component specification whether it accepts or discards informal parameters.

So, when the Image component renders, it will mix and match the informal parameters with the HTML attributes it generates from formal parameters. This is a capability missing from JSP tags, where specifying an undeclared JSP tag attribute is simply an error. With JSP tags, you are limited to just the attributes explicitly declared for the tag, no more.

Displaying the right stick figure image

Continuing with the rest of the Guess page, the next dynamic section of the HTML template is also related to the `incorrectGuessesLeft` property; it is used to display one of several images for the gallows, showing increasing amounts of the stick figure as the `incorrectGuessesLeft` property drops toward zero.

```
<img jwcid="@Image"
  image='ognl:getImage("scaffold" +
      visit.game.incorrectGuessesLeft)'
    alt="[Scaffold]"
    src="images/scaffold.png"
      border="0"/>
```

Again, we use the same trick; we come up with a logical name for the image asset and map that logical name to an actual file by way of the `<context-asset>` elements in the page specification. This is a good, simple example of the MVC pattern in action; the Model in this case is the `Game` object and its `incorrect-GuessesLeft` property, but there are two Views of the data: the first as a digit, the second as the stick figure on the gallows.

The remaining dynamic portions of the page are more complex and require using multiple components in concert to produce the desired output.

2.4.2 Generating the guessed word display

The next section of the Guess page displays the target word the player is attempting to guess, or at least as much of the target word as the player has guessed so far. Generating this portion of the page starts with the `Game` object, which has a property, `letters`, for just this purpose. The `letters` property is an array of each letter of the target word as an individual character. Each unguessed letter in the target word is replaced with an underscore character.

As with the previous examples, we can't simply output the individual letters as characters. To keep the hand-scrawled look and feel, each letter must be translated to the correct image. The template uses two different components to generate the display: a Foreach component (which performs a kind of loop) enclosing another Image component. The two components work together to display one letter after another.

```
<span jwcid="@Foreach"
   source="ognl:visit.game.letters"
   value="ognl:letter">
<img jwcid="@Image"
   image="ognl:letterImage"
   alt="ognl:letterLabel"
   height="36"
   src="images/Chalkboard_5x3.png"
   width="36"
   border="0"/>
</span>
```

Looping with the Foreach component

Foreach is a looping component; it iterates over the list of values provided by its `source` parameter[7] and *updates* its `value` parameter for each value from the source before rendering its body. This is a crucial feature of Tapestry component parameters; by binding a property to a component parameter, the component is free not only to read the value of the bound property, but also to update the property as well.

The Foreach component is represented in the template using a `<span>` tag, which is very natural: The HTML `<span>` tag is simply a container of other text and elements in a page. It doesn't normally display anything itself, but is commonly used in conjunction with a stylesheet to control how a portion of a page is rendered.

Although the Foreach's location in the template is specified using a `<span>` tag, when it renders, it does not produce any HTML directly; it simply renders the text and components in its body repeatedly. The sequence is shown in figure 2.7.

So, the Foreach component will render its body many times, but that doesn't help the Image component display the correct letter image. Just before the Foreach renders its body (on each pass through the loop), it sets a property of the page to the next letter in the word (from the array of characters provided by the `Game` object). The trick is to convert this letter into the correct image. The Guess page class includes a property, `letter`, which is bound to the Foreach component's `value` parameter so that it can be updated by the Foreach:

```
private char _letter;

public char getLetter()
{
  return _letter;
}

public void setLetter(char letter)
{
  _letter = letter;
}
```

[7] The Foreach component is flexible about how it defines "a list of values." It may be an array of objects, or a `java.util.List`, or even a single object (which is treated like an array of one object).

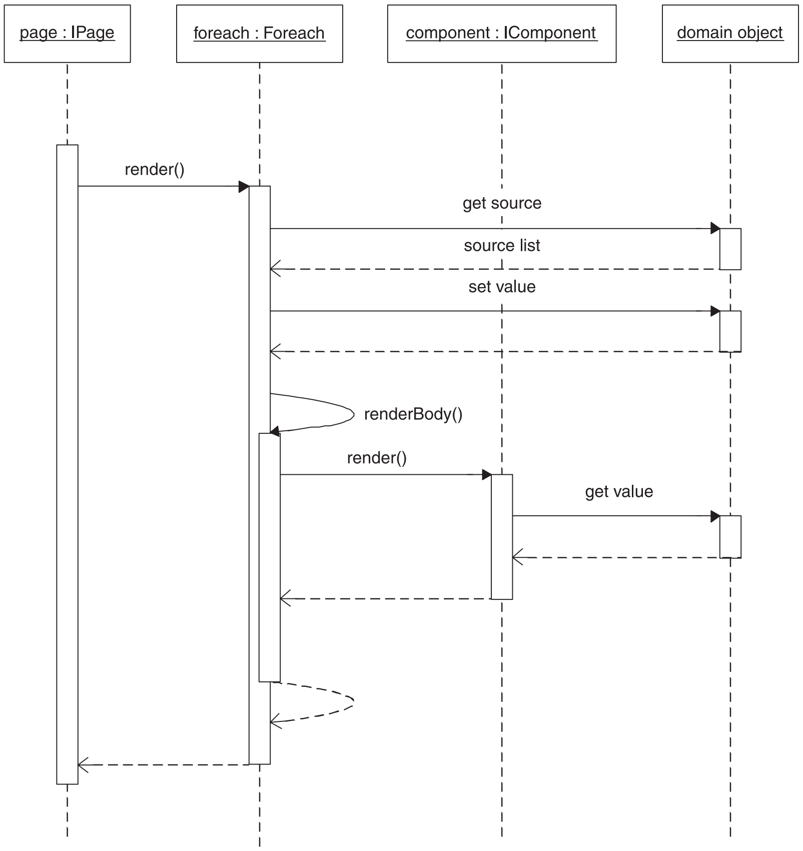


Figure 2.7 The Foreach component reads a list of values bound to its source parameter from a domain object (which is often the page that contains the component). For each item in the list, it updates a domain object property bound to its `value` parameter, and then renders its body. Components within its body can get the value from the domain object property.

NOTE In chapter 3, we'll see how Tapestry can automatically create properties at runtime (and the benefits of doing so beyond less typing). For now, we'll mechanically code these properties ourselves by supplying the instance variable and pair of accessor methods.

Translating letters to images

Once again, we are using assets to obtain the correct image to display within the page. The assets for the letters a through z are named, simply, a through z. However, there's a gotcha for the underscore character; its asset name is dash.

The Guess page class implements another method to provide the asset to display:

```
public IAsset getLetterImage()
{
  if (_letter == '_')
    return getAsset("dash");

  return getAsset("" + _letter);
}
```

This simple method captures the special rule about replacing the underscore character with the asset named dash. The Foreach component is responsible for invoking `setLetter()` with the correct letter well before `getLetterImage()` is invoked by the Image component.

NOTE Because this method is public and follows the naming convention for a JavaBeans property, it can be referenced in the HTML template as `ognl:letterImage`. This is a common approach in Tapestry—creating *synthetic properties*, properties that are computed on the fly, rather than just exposing a value in an instance variable.

The letters in the list (provided by the `Game` object) are all lowercase, but the tooltip (generated from the `<img>` tag's `alt` attribute) looks better if the letter is uppercase. This is another, minor example of the Controller (the page) mediating between the Model (the `Game` object) and the View (the Image component within the HTML template). This case conversion is accomplished by binding the value for the `alt` parameter to the `letterLabel` property of the page. The `getLetterLabel()` accessor method simply converts the letter to uppercase and returns it as a string:

```
public String getLetterLabel()
{
  char upper = Character.toUpperCase(_letter);

  return new Character(upper).toString();
}
```

Removing unwanted portions of the template

If you examine the complete HTML template in listing 2.4, you'll see that just after the `</span>` tag for the Foreach component is a long chunk of additional images—images for additional letters from the target word, as dashes. These images were copied over from the original HTML mockup and are left in place so that the HTML template will still preview properly. Without these additional images, the target word will appear as a single underscore, which may not be enough to validate the layout of the page. At the same time, these extra images must not be included in a live, rendered page or the target word will appear to be six letters longer than it actually is.

Earlier, you saw that Tapestry will drop unwanted HTML attributes that are provided in HTML tags to support WYSIWYG preview. This is a larger case, where an entire section of HTML is dropped. The block to be removed is surrounded by a `<span>` tag:

```
<span jwcid="$remove$"> . . . </span>
```

The special component ID, `"$remove$"`, is the trigger for Tapestry's template parser that this portion of the HTML template should be discarded. This is a second aspect of instrumenting an HTML mockup into an HTML template: marking portions of the mockup for removal, yet leaving them in for previewing purposes.

So far on this page, we've covered just output-only behaviors: displaying the right digit image, or the right letter from the target word. The most involved part of the page comes next—the part that allows players to select letters to guess.

2.4.3 Selecting guesses

This portion of the page is a grid of letters that the player may click on to make guesses. As usual, the letters are represented as images, to keep with the hand-scrawled look and feel. As the player makes guesses, the guessed letter is erased, and one or more positions in the target word are filled in or another segment is added to the stick figure.

To accomplish this, we'll use a combination of components: another Foreach to iterate over the different letters of the alphabet, a DirectLink to create a link, and an Image to display either the image for the letter or the image for a blank space for an already guessed letter. The three components appear in the HTML template:

```
<span jwcid="selectLoop">
<a href="#"
   jwcid="select"
   class="select-letter">
   <img jwcid="@Image"
```

```
        image="ognl:guessImage"
        alt="ognl:guessLabel"
        height="36"
        src="images/Chalkboard_5x3.png"
        width="36"
        border="0"/>
</a>
</span>
```

Two of these components look a little sparse compared to previous examples; that's because we've chosen to use the declared component option for them rather than configure them in-place as implicit components. For a declared component, we just put the component ID in the HTML template. Tapestry recognizes that the value for the first `jwcid` attribute is just an ID and not a component type, because it does not contain the `@` character (as the previous usages of components have done). For a declared component, the element in the HTML template is simply a placeholder; the `jwcid` attribute provides a component ID that is used to link to a `<component>` element in the page specification. The type and configuration of the component is provided in the page specification itself:[8]

```
<component id="selectLoop" type="Foreach">
  <binding name="source" expression="visit.game.guessedLetters"/>
  <binding name="value" expression="letterGuessed"/>
  <binding name="index" expression="guessIndex"/>
</component>

<component id="select" type="DirectLink">
  <binding name="listener" expression="listeners.makeGuess"/>
  <binding name="parameters" expression="letterForGuessIndex"/>
  <binding name="disabled" expression="letterGuessed"/>
</component>
```

The information that goes into the specification is the same as what would be put directly into the HTML template, but the format is slightly different. In the HTML template, we must mark OGNL expressions with the `ognl:` prefix; but in the XML we have a specific element, `<binding>`, that is always an OGNL expression (other elements are used for literal strings and other variations). There is no difference to Tapestry whether a component is declared in the specification or in

[8] The template may still specify additional formal and informal parameters. In keeping with the goal to provide the clearest separation of presentation and logic, the informal parameters, which are most often related purely to presentation, should go in the template, and the formal parameters, which are most often related to the behavior of the component, should go in the page specification.

the HTML template; here, the sheer number of parameters for the two components indicated that specification was a better home for the component configuration than the HTML template.

WARNING Mistakenly using the `ognl:` prefix inside a page or component specification will create an OGNL expression that is invalid. You'll see an exception, such as *Unable to parse expression 'ognl:visit.game.guessedLetters'.* The fact that the `ognl:` prefix shows up in the exception message as part of the expression is the indicator that you included the prefix where it is not allowed.

Once again we are combining the behaviors of different components and using the page to mediate between them. We are also making use of new features of the Foreach and DirectLink components by binding additional parameters of the components.

Getting the images for the letters

The source of all this data is the `guessedLetters` property of the `Game` object; this is an array of 26 flags, one for each letter in the alphabet. Initially, all the flags are false, but as the player makes guesses, the corresponding flags are set to true.

The Foreach component will loop through the 26 flags and set the `letterGuessed` property of the page to true or false on each pass through the loop. In addition, binding the `index` parameter of the Foreach component directs it to set the `guessIndex` property of the page. This value starts at zero and increments with each pass through the loop. The other components simply translate from this ordinal value to a letter in the range of a to z. This functionality is implemented by additional properties and methods in the `Game` class, as shown in the following snippet:

```
private boolean _letterGuessed;
private int _guessIndex;

public boolean isLetterGuessed()
{
  return _letterGuessed;
}

public void setLetterGuessed(boolean letterGuessed)
{
  _letterGuessed = letterGuessed;
}

public int getGuessIndex()
```

```
{
  return _guessIndex;
}

public void setGuessIndex(int guessIndex)
{
  _guessIndex = guessIndex;
}

public char getLetterForGuessIndex()
{
  return (char) ('a' + _guessIndex);
}
```

Getting the right letter image for the current letter within the loop is very similar to the previous examples. Although the dash will never occur, we do have to substitute a blank image for any letter that has already been guessed:

```
public IAsset getGuessImage()
{
  if (_letterGuessed)
    return getAsset("space");

  String name = "" + getLetterForGuessIndex()

  return getAsset(name);
}
```

That covers how we get the image for each letter display, but what about the link that the player uses to make a guess?

Handling the links for guesses

Were we to display the link for guesses using ordinary servlets, we'd define a query parameter whose value is the letter selected; that is, we would encode the letter into the URL. Since we're using Tapestry, we don't want to think in terms of query parameters, but instead, we want to think of objects and properties—but we still want the URL to carry this piece of information. When we render the link, we know which letter the link is for, and when the link is clicked, we need that information back. In Tapestry terms, we need to invoke a specific listener method (as before on the Home page) but also propagate along some additional data: the letter selected by the player.

We'll use a DirectLink component, as we did with the link on the Home page, but with two differences. First, we only want to display the link itself (the `<a>` and `</a>` tags) some of the time; we want to omit the link for letters that have already been guessed (the positions that show up as blank space), because letters may only be guessed a single time. Second, we need a way to know which letter has been selected. The DirectLink component includes formal parameters to satisfy both of these needs.

The `disabled` parameter is used to control whether the link renders the `<a>` and `</a>` tags. The `disabled` parameter is optional, and by default, the link is enabled. A DirectLink component will always render its body, regardless of the setting of the `disabled` parameter. The Guess page binds the `disabled` parameter to the `letterGuessed` property of the page—the same property that is set by the Foreach component and used in the `getGuessImage()` method:

```
<binding name="disabled" expression="letterGuessed"/>
```

This ensures that once a letter has been guessed, there will not be another link for that letter. Shortly, we'll see how we also ensure that the guessed letter is replaced by a blank space. Once again, we are working at the level of objects and properties, and not treating all of this HTML rendering as just a text processing problem. A common, ugly "JSP-ism" is to use embedded scriptlets to avoid writing the open and close tags, wrapping the `<a>` and `</a>` tags inside conditional blocks, which can be a messy affair.

The DirectLink embodies the Tapestry philosophy, solving a similar problem using JavaBeans properties and Tapestry component parameters. Every component decides, in its own code, whether to render; the DirectLink has a small conditional statement to control whether an `<a>` element is rendered—but that's Java code in a Java file, not cluttering up a JSP. The end result is a cleaner, simpler, easier-to-use solution.

To identify which letter is actually clicked by the player, we will use yet another component parameter, named `parameters`. We can bind a single value, or an array, or a `java.util.List` to the `parameters` parameter, which, like the `disabled` parameter, is optional (we didn't use it before with the Start link on the Home page). The collection of values provided by the `parameters` parameter is recorded into the URL constructed when the DirectLink component renders. When the link is submitted, the array of parameters is reconstructed and is available to the listener method.

For this case, we use a single value, provided by the property `letterForGuessIndex`:

```
<binding name="parameters" expression="letterForGuessIndex"/>
```

Each time the DirectLink component renders, within the Foreach component loop, the value for the `letterForGuessIndex` property will reflect the current letter in the loop and the URL written into the HTML response will be different, as a portion of the URL will be an encoding of the `letterForGuessIndex` property.

When the link is clicked, the listener method can get the parameters back:

```
public void makeGuess(IRequestCycle cycle)
{
  Object[] parameters = cycle.getServiceParmeters();
  Character guess = (Character) parameters [0];

  char ch = guess.charValue();
  Visit visit = (Visit) getVisit();

  visit.makeGuess(cycle, ch);
}
```

The parameters encoded into the URL by the DirectLink are available in the listener method as an array of object instances, which can be obtained from the `getServiceParameters()` method of the `IRequestCycle` object. Even when, as in this case, there's only a single parameter value, an array is returned. The lone character value is the first and only element in the array.

In addition, the value has been converted from a scalar type, `char`, to a wrapper object type, `Character`, but it is a simple chore to convert it back. The parameter value is not simply converted to a string; it retains its original type (which is encoded into the URL along with the value). You can see a bit of this in the web browser's Address field in figure 2.1; the URL shown contains much information used by Tapestry, but at the end is *cp*, an encoding of *character p* (the player had just clicked the letter P). Chapter 7 discusses how Tapestry encodes information into URLs.

From here, it's simply a matter of obtaining the `Visit` object and letting it do the rest of the processing of the player's guess, which may result in a win or a loss or more guessing. Because we pass the request cycle to the `Visit`, this object is fully capable of selecting which page will render the response by invoking the `activate()` method on the request cycle.

Adding this new interaction, the handling of guesses by the player, involved little more than creating the new listener method and pointing the DirectLink component at the method. Without Tapestry, this same functionality would entail not only writing a servlet and registering it into the web deployment descriptor, but also creating code to generate the hyperlink in the first place. This latter code could take the form of Java scriptlets in the JSP, or a new JSP tag in a JSP tag library. In either case, the HTML in the JSP file would deviate further from ordinary HTML, and the ability to preview the web page would be diminished. With Tapestry, the HTML template will continue to look and act like standard HTML.

Instead, we are making use of existing components, the DirectLink, and a consistent approach to encoding data into the URL. Once again, we're seeing the consistency goal: Anywhere in the application where we have a link that needs to pass along some data in the URL, we can use and reuse the same tool, the DirectLink component and its `parameters` parameter. In addition, because Tapestry properly encodes the data type with the data (rather than just converting all the parameters to strings), we can consistently pass any type of data in the URL: strings, characters, numbers, or even custom objects.

That wraps up the Guess page; we've discussed how to extract information from the `Game` domain object and present it in various ways and also figured out how to react to user input. We've kept the domain logic (in the `Game` and `WordSource` objects) separate from the presentation logic (the Guess page class, HTML template, and page specification), using listener methods and the `Visit` object as the bridge between the two aspects.

2.5 Developing the Win and Lose pages

The other two pages in the application, Win and Lose, are displayed when the player successfully guesses the word, or when the player exhausts all his or her incorrect guesses. There is nothing new on these pages; they duplicate bits and pieces of the Home and Guess pages. In fact, there's a bit of unwanted duplication in the HTML templates: the Java code and the page specifications. In chapter 6 we'll see how easy it is to create new components that encapsulate this functionality and remove this duplication. Remember: More code means more bugs!

Our Hangman application is nearly complete; all that's left is to fulfill our contract with the servlet container and create a deployment descriptor for the Hangman application WAR.

2.6 Configuring the web.xml deployment descriptor

All of these HTML templates and page specifications do not automatically become a web application. We still need a servlet to act as the bridge between the Servlet API and the Tapestry framework. Fortunately, this does not require any coding, since the framework includes the necessary servlet class. All that's necessary is to configure the web deployment descriptor, which is the file WEB-INF/web.xml. The deployment descriptor is provided in listing 2.11.

Listing 2.11 web.xml: web deployment descriptor for the Hangman application

```
<?xml version="1.0"?>
<!DOCTYPE web-app
  PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
  "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">

<web-app>
  <servlet>
    <servlet-name>hangman</servlet-name>
    <servlet-class>org.apache.tapestry.ApplicationServlet
    </servlet-class>
    <init-param>
      <param-name>org.apache.tapestry.visit-class</param-name>
      <param-value>hangman1.Visit</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>hangman</servlet-name>
    <url-pattern>/app</url-pattern>
  </servlet-mapping>
</web-app>
```

The deployment descriptor maps an instance of the Tapestry `ApplicationServlet` to the path /app within the servlet context. Using /app as the servlet path is a common convention for Tapestry applications but not a requirement; Tapestry will adapt to whatever servlet mapping is actually used.

The servlet context's name is based on the name of the web application archive (the WAR file), which is hangman1.war; therefore, the complete URL of the servlet is http://localhost:8080/hangman1/app. This means that the base URL for the application is http://localhost:8080/hangman1/, which is why relative URLs (in static HTML) to assets such as images/guess.png work, both when previewing the HTML template and at runtime.

The `org.apache.tapestry.visit-class` initial parameter is used to tell Tapestry what class to instantiate as the `Visit` object.

Using the `<load-on-startup>` element in the deployment descriptor is recommended, especially during development. Loading on startup causes the application servlet to be instantiated and initialized. Often, problems in the application or deployment descriptor will be detected immediately at startup; this is an even better idea for more advanced applications that use an application specification.[9]

2.7 Summary

In this chapter, we've seen the basics of creating a web application using Tapestry. A Tapestry application is divided into individual pages; those pages are constructed by combining components, an overall HTML template, and a small amount of Java code. Tapestry leverages the Model-View-Controller pattern to isolate domain logic from the user interface. We've also begun to see the "light touch" of Tapestry, where simple properties and short Java methods are woven together to create very complex, dynamic, interactive user interfaces.

This simple application demonstrates some of the key patterns that occur when developing in Tapestry. It shows how components interact with each other by reading and setting properties. It shows how the page can act as a Controller, coordinating the domain logic and mediating between its embedded components. We've also demonstrated how easy it is to add new interactions to a page, in the form of listener methods.

We've begun to demonstrate how Tapestry, by excusing developers from mundane "plumbing" tasks, really frees up developer energies. It enables you to implement more complicated behaviors in much less time and be more confident that your code is bug free. Tapestry can give projects the one thing money truly can't buy: time—time to test and debug back-end code, time to locate and fix performance problems, even time to add new features.

[9] Application specifications are an optional file described in chapter 6. They are needed only to access some advanced feature of Tapestry, such as referencing a component library.

JAVA

TAPESTRY IN ACTION

Howard M. Lewis Ship

TAPESTRY 3.0

Tapestry is an open source Java web framework with a unique approach: it represents all behavior and all state as standard Java objects, methods, and properties. In the stateless world that is the Web, the Tapestry developer is relieved of the onerous burden of managing state.

This book is an introduction to Tapestry and a guide to the world of Tapestry development. It shows you how you can create complex applications by combining HTML with the framework's components, and connecting them to small amounts of application-specific logic. It illustrates the practical benefits of Tapestry's inbuilt management of state and its clean separation of presentation logic from business logic.

The book is written to be accessible to new Tapestry users and even to developers new to Java web application development in general. Later chapters discuss more advanced topics including integration with J2EE and team development.

What's Inside

- Tapestry's Component Object Model
- Write new components
- Configure third party components
- Dynamic JavaScript integration
- Form validation
- Tapestry/JSP integration
- Localization/internationalization
- J2EE integration

A professional developer for fifteen years, **Howard Lewis Ship** has worked with Java web applications since 1997. He is the creator and the principal architect of the open source Tapestry project.

"*Tapestry in Action* is masterfully written, making this elegant framework accessible to all Java web developers."

—Erik Hatcher, co-author of *Java Development with Ant*

"There is a better, more elegant way to build web apps—*Tapestry In Action* absolutely rocks!"

—Bill Lear
Wayport Inc./DejaNews

"Tapestry is *the way* ... and this book amply demonstrates that there is no better authority on the subject than Howard Lewis Ship."

—Geoff Longman
Intelligent Works,
developer of Spindle for Eclipse

"I found this book just right—for newcomers and experienced Tapestry developers alike."

—Richard Lewis-Shell, Techcon

"Keep your html code-free—write OO web pages the Tapestry way!"

—Joel Trunick, SmartPrice.com

54495

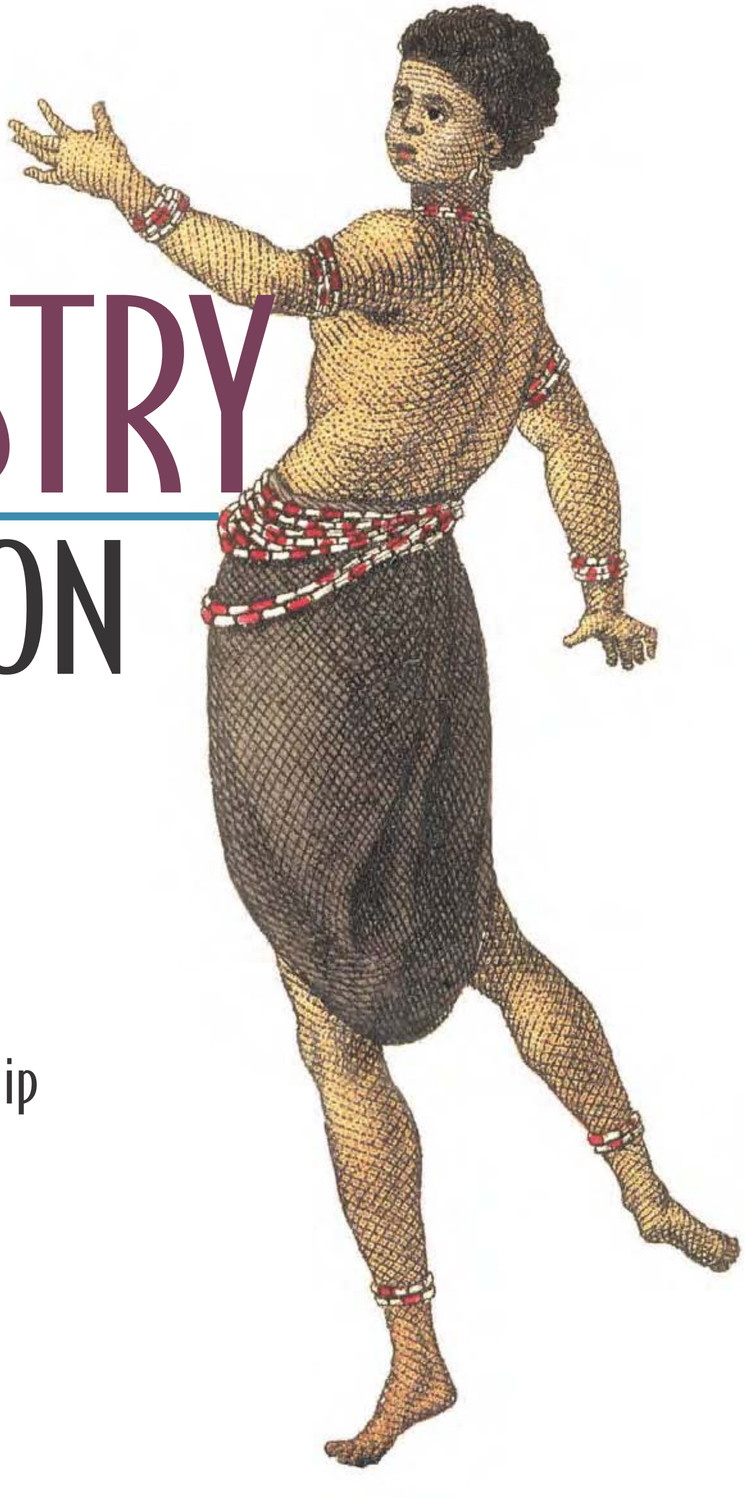
9 781932 394115

ISBN 1-932394-11-7

MANNING $44.95 US/$67.95 Canada

TAPESTRY IN ACTION

Howard M. Lewis Ship



MANNING

Manning Publications Co.
209 Bruce Park Avenue
Greenwich, CT 06830

Copyeditor: Liz Welch
Typesetter: Denis Dalinnik
Cover designer: Leslie Haimes

ISBN 1-932394-11-7

Printed in the United States of America
1 2 3 4 5 6 7 8 9 10 – VHG – 08 07 06 05 04

brief contents

APPENDIXES

5 Form input validation

This chapter covers

- Knowing the requirements of a usable, validating interface
- Using validators and validation delegates
- Using FieldLabel and ValidField components
- Adapting output for your application's look and feel
- Performing form-level validations

Processing forms is more than creating drop-down lists and text fields; the forms are just part of an overall cycle of user input and server validation. Users will make mistakes when entering data into forms. They'll leave a required field blank, enter letters into fields clearly marked for numeric input, ignore any kind of range requirement, and type whatever they want. It isn't enough just to reject invalid input from the user as an exception case; users can be expected to find creative ways to enter invalid input on even the simplest forms. Your responsibility is to handle invalid input consistently and gracefully—that is, to create a *usable* application. Fail to take your users' needs and expectations into account by creating a slipshod, unfriendly application and you run the risk of frustrating them—and driving them away from your site. Users have some basic expectations when it comes to input validation:

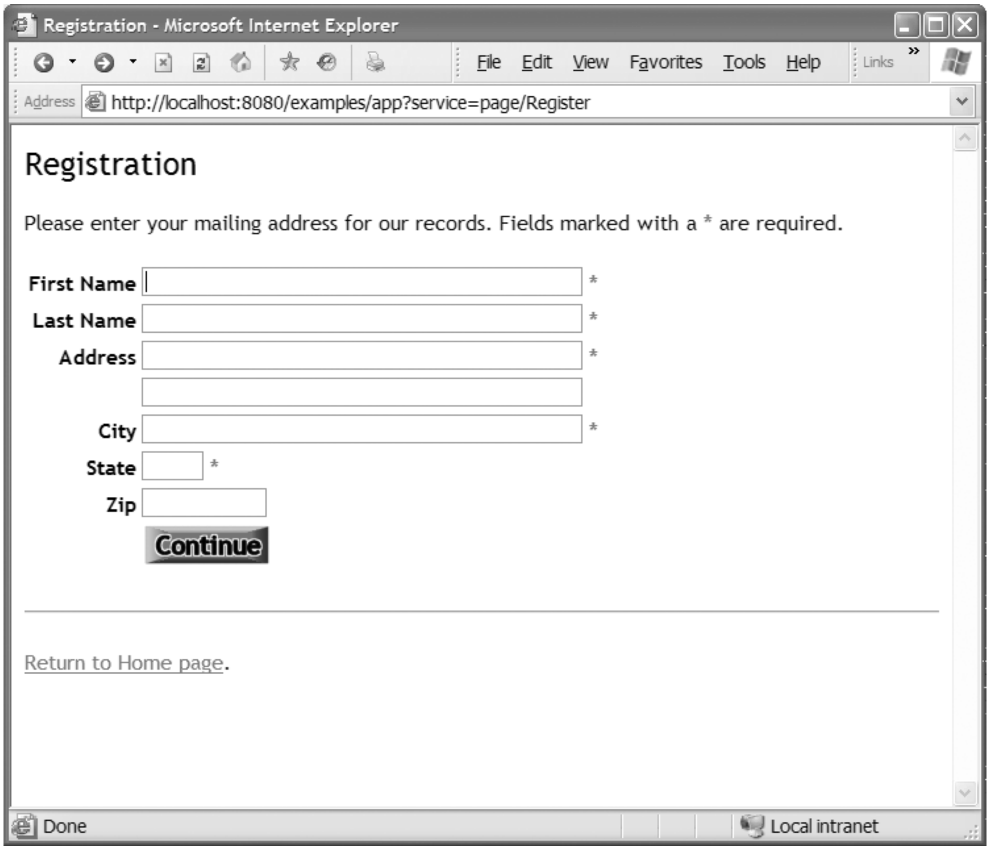
- Fields with invalid data will be marked as such; whether this is with color, icons, or some other approach, it should be possible to identify fields with problems at a glance. An unfriendly application will just display an error message and expect the user to deduce the invalid field.
- Invalid input should be maintained so that it can be corrected. For example, if a user types only 15 digits of a credit card number, you should give the user a chance to enter just the 16th digit. An unfriendly application forces the user to retype the entire number from scratch.
- All errors should be visible at once, to allow the user to correct all the errors without additional server requests. An unfriendly application can recognize only one error at a time, forcing the user to submit the form multiple times.
- Client-side validations are preferable to server-side validations.

Tapestry includes an entire subsystem for validating user input, centered on its ValidField component. The subsystem allows you to easily build highly usable forms that provide useful feedback and client-side validation. We'll begin by demonstrating how to use validation for a simple user registration form, and later, we'll show you how to mix and match the validation subsystem with the DatePicker component from chapter 4.

5.1 Validating user input

The ValidField component is a variation of the TextField component described in chapter 3. For the most part, a ValidField component is used in exactly the same way as a TextField component—with some additional parameters. ValidField components are capable of editing not just string properties, but dates and numbers as well, and can be adapted to any data type with a reasonable text representation. The validations that can be used with ValidField components can apply both client-side and server-side checks on input. These checks ensure that the user provides values for required fields, or force user input to be within a specified range. In many cases, the validations are tied in with conversions, such as converting input from a string to a date or to some type of number.

An example of ValidField components in use appears in figure 5.1, which shows a page that accepts a user's name and address—the kind of page you'll see in many online applications. Initially, all the fields are blank, and the cursor is


Registration - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Links
Address

http://localhost:8080/examples/app?service=page/Register

Registration

Please enter your mailing address for our records. Fields marked with a * are required.

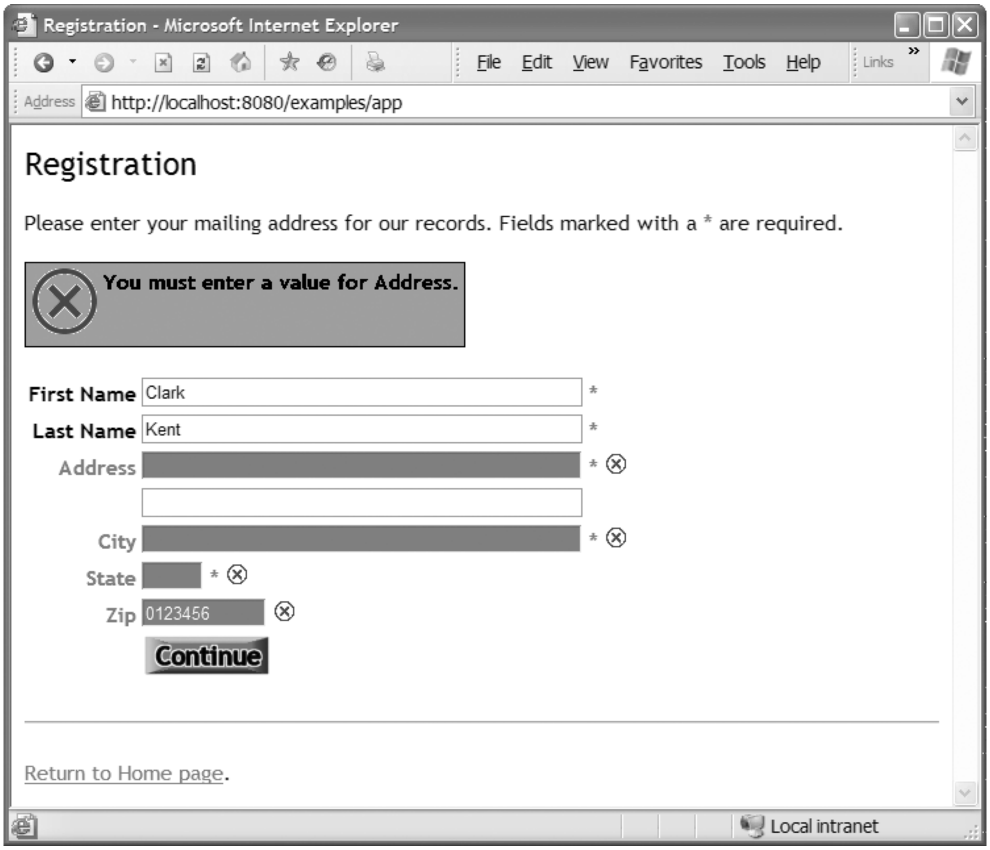
First Name		*
Last Name		*
Address		*
City		*
State		*
Zip		

Return to Home page.

Figure 5.1 The Register page in its initial state. Each required field is marked with an asterisk, and the cursor is placed into the first required field.

automatically placed into the first required field (using a snippet of client-side JavaScript, automatically generated by the ValidField component).

If you enter values into some fields but not others and submit the form (this form does not have client-side validation enabled), you'll see a different screen (figure 5.2). The page is redisplayed so that you may make corrections. Details about the first field error appear at the top; each field that is in error is so marked in several ways: the label for the field, and the field itself, both change color, and an error marker is added to each field.[1]


Registration - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Links
Address
Local intranet

http://localhost:8080/examples/app

Registration

Please enter your mailing address for our records. Fields marked with a * are required.

You must enter a value for Address.

First Name: Clark *
Last Name: Kent *
Address: *
City: *
State: *
Zip: 0123456

Continue

Return to Home page.

Figure 5.2 The same form partially filled out and submitted. The first error on the page is displayed prominently. All fields with errors are highlighted in three ways: the field label text is red, the field itself gets a red background, and an icon is displayed to the right of the field.

[1] As you'll see, you can easily customize the look and feel for validation to fit seamlessly into your application.

So, how does this all work? It's more than what a single component in isolation can accomplish; a component can normally affect only a small portion of the overall page, but figure 5.2 reflects changes to the output HTML scattered throughout the entire rendered page. Making this work requires one additional component (FieldLabel) and two additional objects (a validator that parses and validates the user input, and the validation delegate, which tracks fields and errors), with some subtle interactions between all four.

The next few sections give a quick overview of the various parts of the validation subsystem.

5.1.1 Using FieldLabels in conjunction with ValidFields

A FieldLabel is a companion component to a ValidField. Each FieldLabel is connected to ValidField via the FieldLabel's `field` parameter. The FieldLabel can adjust its visual look to reflect the field. If the field is in error (because of invalid user input), the FieldLabel can display itself differently. Figure 5.2 shows this; the labels on several fields have been marked red to indicate that the fields are in error.

In addition, the FieldLabel obtains the user-presentable name of the field to display from the field itself (ValidField has a `displayName` parameter used for this purpose). This ensures that the field label matches the name of the field used in any error messages, even when the name of the field is localized into the user's language, or when the field name is determined dynamically.

FieldLabels and ValidFields are largely responsible for presenting the interface to the user; the heart of input validation is provided in specialized validator objects.

5.1.2 Using validators

Validators are objects that are responsible for translating user input from strings into object types (such as numbers or dates) and performing other validation checks. Validators are not components themselves; they are objects that implement the `IValidator` interface. Validators are *used* by ValidField components, which delegate all the conversions and checks to the validator. A validator object has four responsibilities:

- Converting an object value (from a domain object property) to a string that can be used when rendering the page. The converted value is used as the `value` attribute of the `<input>` element rendered by the ValidField component.
- Converting a string value submitted with the form back into an object value so that it can be used to update a domain object property.

- Performing additional validations on the input. These validations are controlled by properties of the validator instance, which you must configure. For example, you may set the `maximum` property of a `NumberValidator` to enforce a maximum allowable value.
- Writing any client-side JavaScript needed to perform client-side validations.

Each ValidField component has a `validator` parameter, which is bound to the validator object for that field. Validator objects are meant to be shared; they don't store any information about a particular ValidField. Many ValidFields can share the same validator instance.

All validators include a `required` property of type boolean. When the `required` property is set to true, a validator will not allow an input field to be blank or to consist only of whitespace.

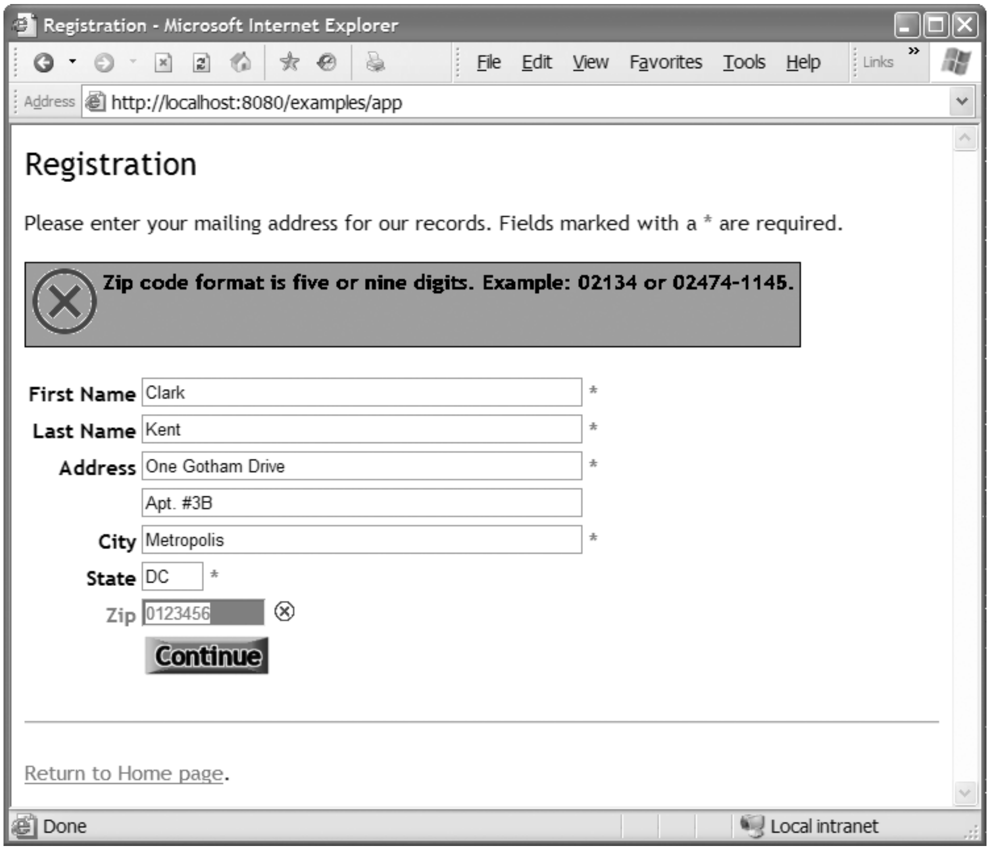
Tapestry provides a number of implementations of the `IValidator` interface that can be used and configured off the shelf. `StringValidator` is used for editing string properties of pages or domain objects and can apply an additional validation: a minimum number of characters that you want the field to accept.

`NumberValidator` is used for editing numeric properties of all types (`int`, `long`, `BigDecimal`, and so on). `NumberValidator` can enforce a minimum or a maximum value, or both. `DateValidator` is used for editing `Date` properties. As with `NumberValidator`, you can set a minimum or a maximum value. Later in this section, you'll see how to create a custom validator, one that enforces the format of a postal zip code (which can be seen in action in figure 5.3).

The final piece of the validation puzzle is the *validation delegate:* an object that is used to track which fields are in error within a form, and what error(s) are associated with each field.

5.1.3 Using validation delegates

The validation delegate has two distinct functions. First, it tracks the error state of each ValidField enclosed by a form. When the form is submitted, each ValidField passes the string provided by the user to the validator object for that field. The validator object may convert the string to an object type, such as `Long` or `Double`, or leave it as a string. It will also apply any validations, such as checking that the converted value fits into a specified range. If the converted value fails a validation, the validator reports the error back to the ValidField. The conversions and validations a validator can perform are flexible, and, as you'll see, it is easy to create new validators to handle new types of conversions and validations.


Registration - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Links
Address http://localhost:8080/examples/app

Registration

Please enter your mailing address for our records. Fields marked with a * are required.

Zip code format is five or nine digits. Example: 02134 or 02474-1145.

First Name	Clark	*
Last Name	Kent	*
Address	One Gotham Drive	*
	Apt. #3B	
City	Metropolis	*
State	DC	*
Zip	0123456	⊗

Continue
Return to Home page.
Done
Local intranet

Figure 5.3 A custom validator attached to the Zip field knows the correct format for zip codes and can generate a custom error message.

Validators report these errors by throwing a `ValidationException`. The ValidField catches the exception and uses the validation delegate to record the ValidField's element ID, the exception message, and the invalid input provided by the user. The recorded exception message is the indicator that the field is in error, and will be used when the page containing the form is rendered.

Validation delegates have a second, discrete function. The delegate is responsible for *decorating* the fields and labels that are in error. The validation delegate has opportunities to render additional HTML before and after the FieldLabel renders, and before and after the ValidField renders, and can even write additional attributes into the `<input>` element rendered by the ValidField itself. The exact sequence for the FieldLabel component is shown in figure 5.4.

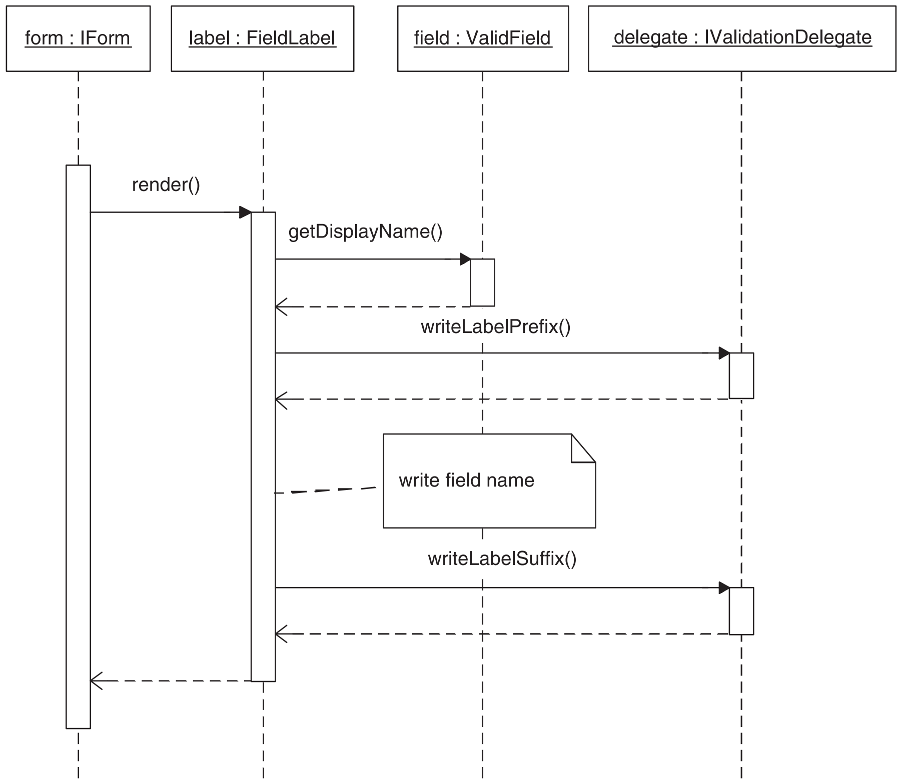


Figure 5.4 The field label gets the user-presentable name for the field from the ValidField component. It allows the delegate to render before and after it renders the field name, so that the delegate can decorate the label if the corresponding field is in error.

This additional rendering is how the labels for invalid fields, shown in figure 5.2, manage to be displayed in red. The validation delegate has a chance to wrap the FieldLabel's output in a `<span>` element, and the span references a Cascading Style Sheet (CSS) class that results in the label being displayed in red. The methods `writeLabelPrefix()`and `writeLabelSuffix()` allow the validation delegate to decorate the label by writing extra HTML around it. Likewise, both the validator and the validation delegate are integrated into the render of the ValidField, as shown in figure 5.5.

The validator's `renderValidatorContribution()` method is primarily used by the validator to write client-side JavaScript (which will perform client-side validations). The validation delegate methods (primarily, the `writeAttributes()`

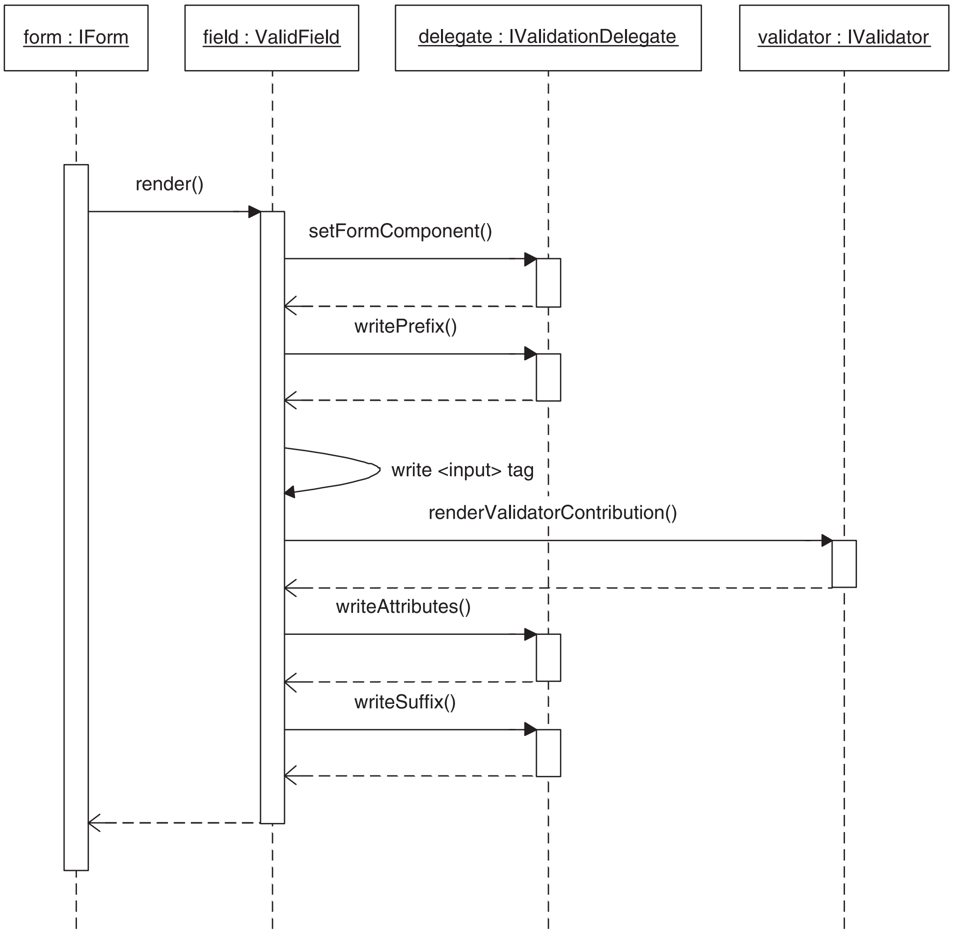


Figure 5.5 The delegate and the validator are both hooked into the rendering of the ValidField. The validator may use its hook to create client-side validation JavaScript. The delegate uses its hooks to decorate fields that are in error.

method) are used to decorate the `<input>` element (rendered by the ValidField) if the field is in error.

The primary purpose of creating a custom subclass of `ValidationDelegate` is to override these rendering methods so that an application-specific look and feel for decorating fields and labels can be created. In the examples in this chapter, we've done just that to mark the fields and labels in red, and to add an error icon after invalid fields.

The validator objects and the validation delegate object are needed by the Form and ValidField components. They are connected to the components using the components' parameters (the `delegate` parameter of the Form component, and the `validator` parameter of the ValidField component). These objects don't appear out of thin air; they must be instantiated and configured. This can be done in Java code (within your page class) or by using another feature of the page specification: helper beans.

5.1.4 Using helper beans

As we've been discussing, handling validation is about more than pages and components. It involves the participation of at least two other objects: the validator attached to each ValidField, and the validation delegate that tracks field input errors throughout a form.

Both of these objects are JavaBeans—but where and when do they get instantiated?

One possibility is to make instantiation the responsibility of the page. The page class could include fields and accessor methods, such as

```
private IValidator _required;

public IValidator getRequired()
{
  if (_required == null)
  {
    _required = new StringValidator();
    _required.setRequired(true);
  }

  return _required;
}
```

However, that kind of mechanical coding goes against the grain of Tapestry. Getting these *helper beans* instantiated and configured falls into the general "plumbing" category—the category of things that should be shifted into the framework. As you've seen before, when Tapestry provides an alternative to coding, it involves the page (or component) specification. In this case, we want to be able to control which Java class is instantiated and how it is configured. This is accomplished in the page specification using the `<bean>` element:

```
<bean name="required"
  class="org.apache.tapestry.valid.StringValidator"
  lifecycle="page">
  <set-property name="required" expression="true"/>
</bean>
```

The `<bean>` element shown here is equivalent to the Java code snippet. The `<bean>` element specifies three things:

- The Java class you want to instantiate
- The configuration of any properties of the helper bean
- The lifecycle of the bean

A helper bean may be accessed through the `beans` property of the page or component, using its name. For example, the OGNL expression `beans.required` would access the bean specified here. Helper beans are created only as needed, the first time the bean is referenced.

By default, the lifecycle of a bean is `request`, meaning that the bean will be released to the garbage collector at the end of the current request cycle. This is often appropriate, especially if the bean contains any state particular to the current user—such information should last only as long as the current request so that it won't be visible in a subsequent request from a different user. By using the `page` lifecycle, the bean is kept for as long as the page instance exists; this is appropriate for beans such as a validator that have no internal state. As you'll see in the following examples, a common use for helper beans is to define validators and validation delegates.

So far, you've seen an overview of how the various aspects of the validation subsystem (the components, the validators, and the validation delegate) operate. Next let's return to the Register page (shown in figure 5.1) and start seeing how these bits and pieces fit together to form a working, validated form.

5.2 Building the Register page

The ultimate aim of the Register page is to collect a user's name and address and store it in an `Address` (shown in listing 5.1).

Listing 5.1 Address.java: data object used with the Register page

```
package examples.register;

import java.io.Serializable;

public class Address implements Serializable
{
  private String _firstName;
  private String _lastName;
  private String _address1;
  private String _address2;
```

```
    private String _city;
    private String _state;
    private String _zip;

    public String getAddress1()
    {
      return _address1;
    }

    public String getAddress2()
    {
      return _address2;
    }

    public String getCity()
    {
      return _city;
    }

    public String getFirstName()
    {
      return _firstName;
    }

    public String getLastName()
    {
      return _lastName;
    }

    public String getState()
    {
      return _state;
    }

    public String getZip()
    {
      return _zip;
    }

    public void setAddress1(String address1)
    {
      _address1 = address1;
    }

    public void setAddress2(String address2)
    {
      _address2 = address2;
    }

    public void setCity(String city)
    {
```

```
    _city = city;
  }

  public void setFirstName(String firstName)
  {
    _firstName = firstName;
  }

  public void setLastName(String lastName)
  {
    _lastName = lastName;
  }

  public void setState(String state)
  {
    _state = state;
  }

  public void setZip(String zip)
  {
    _zip = zip;
  }

}
```

Armed with this definition of what goes into an address, we can create a user interface to let users enter an address.

5.2.1 Creating the Register HTML template

The HTML template for the Register page is shown in listing 5.2. This template introduces a number of new concepts, so we'll take it apart one small piece at a time:

- Using the `delegate` parameter of the Form component
- Using the page's `components` property to reference other components within the page's template
- Using the FieldLabel component in conjunction with the ValidField component

Listing 5.2 Register.html: HTML template for the Register page

```
<html jwcid="@Shell" title="Registration"
  stylesheet="ognl:assets.stylesheet">
<head jwcid="$remove$">                        ❶ Elements
<link rel="stylesheet" type="text/css"            marked for
  href="css/style.css"/>                          removal
</head>
<body jwcid="@Body">
```

```
<span class="title">Registration</span>

<p>Please enter your mailing address for our records.
Fields marked with a
<span class="required-marker">*</span>
are required.
</p>

<span jwcid="@Conditional"
  condition="ognl:beans.delegate.hasErrors">

<table class="error">
<tr valign="top">
<td>
<img height="52" alt="[Error]" src="images/form-error.png"
  width="52">
</td>
<td>
<span jwcid="@Delegator"
  delegate="ognl:beans.delegate.firstError">
  Error Message
</span>
</td>
</tr>
</table>

</span>

<form jwcid="@Form"
  listener="ognl:listeners.formSubmit"
  delegate="ognl:beans.delegate">        ❷ Validation delegate
                                            for the Form
<table class="form">

<tr>                                        Labels connected
<th><span jwcid="@FieldLabel"                  to ValidFields ❸
          field="ognl:components.inputFirstName">First Name
    </span></th>
<td><input type="text" jwcid="inputFirstName" size="50"/></td>
</tr>                                           ValidField as a ❹
                                             declared component
<tr>
<th><span jwcid="@FieldLabel"
          field="ognl:components.inputLastName">Last Name
    </span></th>
<td><input type="text" jwcid="inputLastName" size="50"/></td>
</tr>

<tr>
<th><span jwcid="@FieldLabel"
     field="ognl:components.inputAddress1">Address
    </span></th>
<td><input type="text" jwcid="inputAddress1" size="50"/></td>
</tr>

<tr>
<td></td>
<td><input type="text" jwcid="@TextField"
             value="ognl:address.address2" size="50"/></td>
</tr>

<tr>
<th><span jwcid="@FieldLabel"
          field="ognl:components.inputCity">City
    </span></th>
<td><input type="text" jwcid="inputCity" size="50"/></td>
</tr>
<tr>
<th><span jwcid="@FieldLabel"
          field="ognl:components.inputState">State
    </span></th>
<td><input type="text" jwcid="inputState" size="2"/></td>
</tr>

<tr>
<th><span jwcid="@FieldLabel"
          field="ognl:components.inputZip">Zip
    </span></th>
<td><input type="text" jwcid="inputZip" size="10"/>
</tr>

<tr>
<td></td>
<td><input type="image" src="images/continue.png"
           width="100" height="32"/>
</tr>

</table>

</form>

<hr/>

<p><a href="#" jwcid="@PageLink" page="Home">
  Return to Home page</a>.</p>

</body>
</html>
```

❶ This special ID means that the element will be removed from the template. The Shell component will provide all of this, and more, in the running application. This reference to the stylesheet is useful just for WYSIWYG preview.

❷ When using validation, the Form component's `delegate` parameter is used to identify the validation delegate shared by all ValidField components enclosed by the Form.

❸ Each FieldLabel component is connected to the corresponding ValidField.

❹ Because ValidField components have even more parameters than a TextField component, it is usually best to put all that information in the page specification instead.

As is often the case with Tapestry, the HTML template is just the starting point for understanding how the page will behave when the application is running. The following sections fill in the missing details.

Defining a stylesheet for the page

The Register page makes heavy use of CSS; that's how fields and labels are highlighted in red when they are in error. To support this, the rendered page must include that stylesheet. You include stylesheets by using a `<link>` element within the `<head>` element (within the `<html>` element). Because the Shell component is writing the `<html>` and `<head>` elements, that component must write the link to the stylesheet. This is accomplished by binding its `stylesheet` parameter to an asset. A declaration for that asset will appear in the Register page's specification:

```
<html jwcid="@Shell" title="Registration"
  stylesheet="ognl:assets.stylesheet">
```

Removing portions of the template

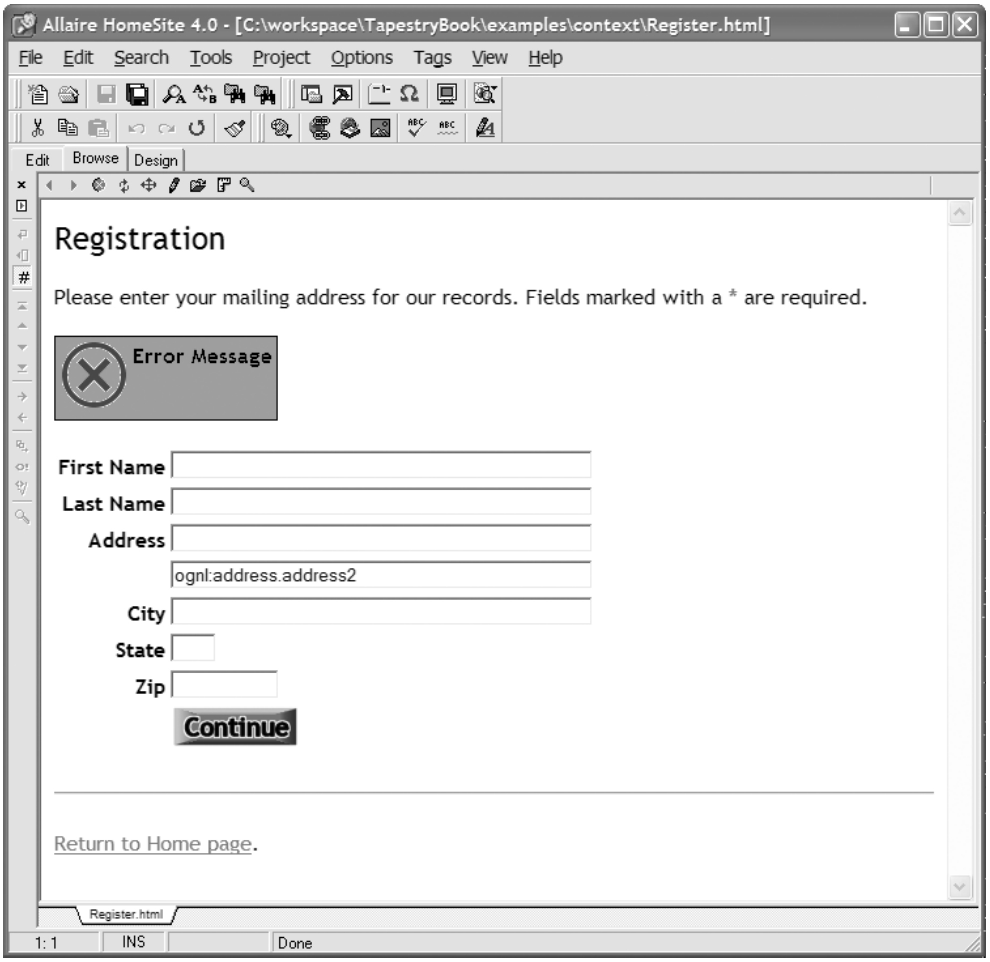
Now comes a conundrum. We still would like the Register page to preview properly while editing, but if the template includes the `<head>` and `<link>` elements needed to include the stylesheet, then the final rendered page will include two sets of `<head>` and `<link>` elements: one from the template, and one dynamically rendered by the Shell component.

Tapestry includes a little trick to sidestep this issue:

```
<head jwcid="$remove$">
<link rel="stylesheet" type="text/css" href="css/style.css"/>
</head>
```

That special component ID, `"$remove$"`, is the key. It isn't normally a valid ID (because it contains a dollar sign). However, Tapestry allows it as a special case but doesn't define a new component. Instead, this special ID cues Tapestry to remove the element and everything enclosed by the element, as if it were never in the template in the first place. With this in place, the page previews correctly when editing, as demonstrated in figure 5.6.

The Shell component will produce the `<html>` and `<head>` element of the rendered page. To support JavaScript within the page, we must use a Body component to render the `<body>` element.


Allaire HomeSite 4.0 - [C:\workspace\TapestryBook\examples\context\Register.html]
File Edit Search Tools Project Options Tags View Help
Edit Browse Design
Error Message
Continue
Register.html
1: 1 INS Done

Registration

Please enter your mailing address for our records. Fields marked with a * are required.

First Name
Last Name
Address
ognl:address.address2
City
State
Zip

Return to Home page.

Figure 5.6 The Register page still previews properly in a WYSIWYG HTML editor.

Using a Body component

The ValidField component will almost always produce some client-side JavaScript. At the very least, initialization JavaScript is included that moves the cursor to the first field that is either required (but empty) or in error. As we discussed in chapter 4, a Body component is necessary to organize the production of client-side JavaScript. The templates for all examples in this chapter make use of a Body component:

```
<body jwcid="@Body">
. . .
</body>
```

With the basic infrastructure of the page in place, we can move on to the first dynamic section of the page: the area where any validation error messages are displayed.

Displaying validation errors

The next section of the template is concerned with displaying validation errors. The validation delegate for the page is obtained and checked to see if it contains any errors. The delegate's `hasErrors` property will always be false when a page is initially rendered. The `hasErrors` property may be set to true during the Form's rewind, when a form submission takes place. As you'll see, the Form's listener method will also query the delegate's `hasErrors` property. If the property is false, it is safe to take the validated input and move forward to the next step in the process. If `hasErrors` is true, the Register page is redisplayed to reveal the errors and decorate any fields that are in error.

Although it is possible to display all error messages for all fields that are in error, such output will be unwieldy on large forms and not very useful. Instead, the error message for the first field that is in error is displayed, but all fields throughout the form that are in error are marked. You've seen examples of this in figure 5.2, where the message refers to only the first field in error, even though several fields are marked. The HTML template makes use of Conditional and Delegator components to produce the formatted error message:

```
<span jwcid="@Conditional"
  condition="ognl:beans.delegate.hasErrors">

<table class="error">
<tr valign="top">
<td>
<img height="52" alt="[Error]" src="images/form-error.png"
  width="52"/>
```

```
</td>
<td>
<span jwcid="@Delegator"
  delegate="ognl:beans.delegate.firstError">
  Error Message
</span>
</td>
</tr>
</table>

</span>
```

In this example, the validation delegate is obtained using the OGNL expression `beans.delegate`, a reference to the delegate helper bean (which itself is declared in the page specification). The Conditional component queries the delegate's `hasErrors` property and displays the error message (formatted inside a `<table>`) only if the property is true.

The error objects that the validation delegate returns (such as the `firstError` property) are not simply strings but objects that implement the `IRender` interface. Because these are objects and not strings, we can't use an Insert component to display them. Instead, we use a Delegator component, which invokes the `render()` method on the renderable object provided to it.

The fact that these are renderable objects, rather than simple strings, opens up a whole realm of possibilities. The error objects can render all sorts of HTML, not just text: images, JavaScript pop-up windows, links, formatting—anything HTML. The intent is that customized validators for the application can provide customized error objects that are more than just a wrapper around a string. Implementations that make use of these possibilities are very application specific, which is why the validator classes provided with the framework don't use this feature.

The validation delegate, as a convenience, includes a `firstError` property that is the renderable object for the first field error. The `firstError` property is passed to the Delegator, which results in the error message being displayed.

Following the error message output is the form containing the FieldLabel and ValidField components.

Starting the Form component

The Form component is used as in examples from the previous two chapters, with one difference. An additional parameter, `delegate`, is specified to link the Form to the validation delegate. Every FieldLabel and ValidField component enclosed by the Form must use the same validation delegate—and they all will, since they all will retrieve the validation delegate through the Form:

```
<form jwcid="@Form"
  listener="ognl:listeners.formSubmit"
  delegate="ognl:beans.delegate">
```

Again, the OGNL expression `beans.delegate` resolves to a validation delegate. All the components enclosed by the Form will share this one validation delegate instance.

Using the FieldLabel component

Each ValidField will be preceded in the HTML template by a FieldLabel. The FieldLabel is connected to its partner ValidField by the FieldLabel's `field` parameter:

```
<span jwcid="@FieldLabel"
      field="ognl:components.inputFirstName">First Name
</span>
```

The OGNL expression `components.inputFirstName` is a reference to the inputFirstName component of the page. Every page provides a read-only `Map` of all the components it contains. The keys of this `Map` are the component IDs. Using OGNL, you can access the values in a `Map` just as easily as the properties of an ordinary JavaBean. Of course, to build a reference, you must know the ID of the component, which is one reason the ValidField components are given explicit IDs.

During the render, the FieldLabel discards its body (the text *First Name*) and gets, from the ValidField, the correct field name to display as a label. This may seem cumbersome, but it is useful for two reasons. First, it ensures that the label in the output HTML matches the name for the field used in any error messages generated by the ValidField's validator. Second, if the ValidField localizes the name, the FieldLabel will still match, using the locale-specific value.[2] In addition, the validation delegate will have a chance to render before and after the FieldLabel; this allows the delegate to decorate the label when the field is in error.

Using the ValidField component

In this example, each ValidField is constructed as a declared component, not an implicit component. Each ValidField appears in the HTML template, but its type and most of its parameters are declared in the page's specification. The ValidField components may not be anonymous—they must have real IDs. This is necessary so that the placeholder in the template can be linked to the entry in the page specification, but also so that the FieldLabel can be connected to the ValidField.

[2] Localization of Tapestry applications is discussed in chapter 7.

Each ValidField appears in the template minimally, because the bulk of the component's configuration is in the page specification:

```
<input type="text" jwcid="inputLastName" size="50"/>
```

The FieldLabel and ValidField components blend into the HTML template like any other type of Tapestry component. They are just a bit more sophisticated in terms of how they render their HTML (to allow the validators and validation delegate to decorate them) and in how they are configured. When you use Tapestry form validation, the interesting part takes place inside the page specification.

5.2.2 Creating the Register page specification

The page specification for the Register page (listing 5.3) has two main sections (beyond the elements you've seen before, such as defining the page class and declaring assets). The first section defines additional helper beans used with this page, including the validation delegate and the validators used by the ValidField components. The second section defines each of the ValidField components.

Listing 5.3 Register.page: specification for the Register page

```
<?xml version="1.0"?>
<!DOCTYPE page-specification PUBLIC
  "-//Apache Software Foundation//Tapestry Specification 3.0//EN"
  "http://jakarta.apache.org/tapestry/dtd/Tapestry_3_0.dtd">

<page-specification class="examples.register.Register">

  <bean name="delegate"
     class="examples.register.RegisterDelegate"/>          1 Declares validation delegate

  <bean name="required"
    class="org.apache.tapestry.valid.StringValidator"      2 Declares shared validator
    lifecycle="page">
    <set-property name="required" expression="true"/>
  </bean>

  <bean name="stateValidator"
    class="org.apache.tapestry.valid.StringValidator"
    lifecycle="page">
    <set-property name="required" expression="true"/>
    <set-property name="minimumLength" expression="2"/>
  </bean>

  <bean name="zipValidator"
    class="examples.register.PatternValidator"
    lifecycle="page">
    <set-property name="pattern">
      "\\d{5}(-\\d{4})?"
    </set-property>
    <set-property name="errorMessage">
    "Zip code format is five or nine digits.
     Example: 02134 or 02474-1145."
    </set-property>
  </bean>

  <property-specification name="address"
    type="examples.register.Address"/>

  <component id="inputFirstName" type="ValidField">
    <static-binding name="displayName"
      value="First Name"/>
    <binding name="validator"
      expression="beans.required"/>
    <binding name="value"
      expression="address.firstName"/>
  </component>

  <component id="inputLastName" type="ValidField">
    <static-binding name="displayName" value="Last Name"/>
    <binding name="validator" expression="beans.required"/>
    <binding name="value" expression="address.lastName"/>
  </component>

  <component id="inputAddress1" type="ValidField">
    <static-binding name="displayName" value="Address"/>
    <binding name="validator" expression="beans.required"/>
    <binding name="value" expression="address.address1"/>
  </component>

  <component id="inputCity" type="ValidField">
    <static-binding name="displayName" value="City"/>
    <binding name="validator" expression="beans.required"/>
    <binding name="value" expression="address.city"/>
  </component>

  <component id="inputState" type="ValidField">
    <static-binding name="displayName" value="State"/>
    <binding name="validator" expression="beans.stateValidator"/>
    <binding name="value" expression="address.state"/>
  </component>

  <component id="inputZip" type="ValidField">
    <static-binding name="displayName" value="Zip"/>
    <binding name="validator" expression="beans.zipValidator"/>
    <binding name="value" expression="address.zip"/>
  </component>

  <context-asset name="stylesheet" path="css/style.css"/>

</page-specification>
```

3 Declares ValidField component

❶ The validation delegate is defined here, using a custom subclass, `examples.register.RegisterDelegate`. It can be referenced using the OGNL expression `beans.delegate`. The default lifecycle, “request”, means that the bean will be discarded at the end of the current request.

❷ Most of the fields share a single instance of `StringValidator` as the `required` bean. Since validators don’t store any internal state, the “page” lifecycle means that the bean, once created, will be kept for as long as the page instance itself.[3]

❸ Each ValidField follows this same pattern, providing a `displayName` (used by the FieldLabel and in any error messages), a `validator` (which may be shared between different fields), and a `value` to edit.

This specification includes yet another new element, `<static-binding>`. The `<static-binding>` element is used to set a component property to a fixed, literal string. Such a string is static (unchanging), in contrast to the dynamically evaluated OGNL expression used in a `<binding>` element. The `<static-binding>` element is used throughout the specification to provide the `displayName` parameter for each ValidField.

As an alternative, the value could appear in the HTML template:

```
<input type="text" jwcid="inputFirstName" size="50"
  displayName="First Name"/>
```

Either way, the `displayName` parameter will be set to a static string value. Tapestry is completely flexible as to where parameters are bound (in the specification or in the HTML template).

TIP A good standard is to define formal parameters only in the page specification and informal parameters only in the HTML template. This scheme reflects that formal parameters are most often behavioral and informal parameters are more often related to presentation. Using this division keeps you from wasting time tracking down the right file when a change is needed.

This specification declares several helper beans and a number of ValidField components. The first bean defined is the validation delegate.

Defining the validation delegate and validators

The `RegisterDelegate` class is an implementation of the `IValidationDelegate` interface that customizes the rendering hooks of the delegate to reference CSS

[3] Remember that page instances, once constructed, are cached for later reuse in subsequent requests.

styles and images specific to the Register application. We'll examine its code in detail shortly.

The majority of the fields in the form are used to edit simple string properties, and the only validation constraint applied to them is that a non-null value be supplied by the user. All such ValidField components will share a bean named `required` as their validator.

The `required` bean instantiates an instance of `StringValidator` and sets the `required` property of the `StringValidator` to true. This will force users to provide a non-empty value for the field. Like most validators, the `StringValidator` has no internal state and can therefore be retained indefinitely once created (which is what the `page` lifecycle does).

The delegate bean should not be given a lifecycle of `page` because it has a considerable amount of internal state (the error messages for fields that are in error) that is relevant only to a single request from a single user.

The ValidField for the `state` property uses a different validator, `stateValidator`. This validator is also declared as a helper bean in the page specification:

```
<bean name="stateValidator"
  class="org.apache.tapestry.valid.StringValidator"
  lifecycle="page">
  <set-property name="required" expression="true"/>
  <set-property name="minimumLength" expression="2"/>
</bean>
```

Like the `required` bean, the `stateValidator` bean sets the `required` property to true, forcing the user to provide a value. It adds a second constraint, configured through a second property, requiring the input to be at least two characters in length. The normal `size` attribute (specified as an informal parameter in the HTML template) ensures that no more than two characters are provided.

The final validator is for the zip code field. Zip codes have a pattern that is best described using a regular expression. Tapestry doesn't provide a validator along these lines, but it is easy enough to create one. Although we could create a validator that supports only the zip code pattern, it is virtually no extra work to create a flexible validator where we can configure both the regular expression pattern we want to validate against as well as the error message we want to display if the input fails to match the pattern.

In terms of maximizing usability, it is important that error messages be as helpful to the user as possible. Simply telling users that their input didn't match a regular expression would leave them frustrated and at a loss as to how to correct their input to satisfy the application. Instead, you can display a custom error message that tells users exactly what they need to do. For the zip code field, we can customize the error message to give users examples of the two zip code formats accepted. The regular expression pattern and the error message are both provided in the page specification:

```
<bean name="zipValidator"
  class="examples.register.PatternValidator"
  lifecycle="page">
  <set-property name="pattern">
    "\\d{5}(-\\d{4})?"
  </set-property>
  <set-property name="errorMessage">
    "Zip code format is five or nine digits.
     Example: 02134 or 02474-1145."
  </set-property>
</bean>
```

Here we are employing an alternate usage of the `<set-property>` element. Instead of specifying an expression attribute, as in the previous examples, we are putting the OGNL expression in the body of the `<set-property>` element. This approach is useful here, because we must enclose the literal string values (the pattern and the error message[4]) in double quotes. Putting the OGNL expression in the body is much easier when the expression is long or contains complex punctuation, such as a mix of single and double quotes.

Now that we have declared the validation delegate and the three validators, we can continue on to the ValidFields themselves.

Declaring the ValidField components

All the ValidField component declarations follow the same general template. A `displayName` for the field is provided; this is used by the FieldLabel and in any error messages created by the field's validator. Like an ordinary TextField component, the `value` parameter is bound to the property the ValidField will edit. Unlike a TextField component, however, this property can be bound to any type of property, not just string properties. Finally, a validator is specified. The validator defines the type of input acceptable in the field; similar fields can share validators. The first ValidField is used for entering the user's first name:

```
<component id="inputFirstName" type="ValidField">
  <static-binding name="displayName" value="First Name"/>
```

[4] Alternately, we could put the message into a string properties file, and use the `<set-message-property>` element to retrieve the localized message and set the bean property from it. Appendix D has a complete description of the specification DTD.

```
    <binding name="validator" expression="beans.required"/>
    <binding name="value" expression="address.firstName"/>
  </component>
```

The inputFirstName component uses the `required` bean as its validator and edits the `firstName` property of the address.

All the remaining ValidField components follow the same pattern, providing different display names, different properties, and different page properties for editing.

Implementing the Register page

The Java code for the Register page is concerned with providing the `address` property referenced by the many ValidField components as well as handling the form submission. Listing 5.4 shows the `Register` class. We use the same lifecycle technique as before to initialize the `address` property to a non-null value before the page initially renders (and when the form is submitted). We must check for null because if the page renders again (following a form submission with an input validation error), the `pageBeginRender()` method will be invoked again (first because of the form rewind, and then again for the page render).

Listing 5.4 Register.java: Java class for the Register page

```
package examples.register;

import org.apache.tapestry.IRequestCycle;
import org.apache.tapestry.event.PageEvent;
import org.apache.tapestry.event.PageRenderListener;
import org.apache.tapestry.html.BasePage;
import org.apache.tapestry.valid.IValidationDelegate;

public abstract class Register extends BasePage
  implements PageRenderListener
{
  public abstract Address getAddress();
  public abstract void setAddress(Address address);

  public void pageBeginRender(PageEvent event)        <-❶ Initializes the
  {                                                       address property
    if (getAddress() == null)
      setAddress(new Address());
  }

  public void formSubmit(IRequestCycle cycle)
  {
    IValidationDelegate delegate =
      (IValidationDelegate) getBeans().            ❷ Accesses the
        getBean("delegate");                          delegate bean
    if (delegate.getHasErrors())
      return;

    RegisterConfirm next =
      (RegisterConfirm) cycle.getPage("RegisterConfirm");

    next.setAddress(getAddress());    <-❸ Passes the Address
    cycle.activate(next);                 to the next page
  }
}
```

❶ This method will be invoked when the page initially renders, when the form within the page is submitted, and again if the page is rendered after the form is submitted (because of validation errors). The ValidField components require that a non-null `Address` be available in the `address` property.

❷ In Java code, accessing helper beans involves the `getBeans()` method, from which individual beans can be retrieved by name.

❸ Passing along information collected in a form to another page, shown here, is a standard technique.

The listener method, `formSubmit()`, is invoked when the Form is submitted. In order to determine if there were any errors, the code accesses the validation delegate. If there are errors, the method returns, causing the Register page to redisplay with the error message shown and invalid fields highlighted.

If there are no errors, then the listener method activates the next page, which shows a confirmation. A real application would save this address information to a database before continuing, but that is beyond the scope of this example.

Most of this example is implemented using standard components and objects, but to properly validate the input provided in the zip code field, we need a custom validator class.

5.3 Validating input based on regular expressions

Sometimes, you will need to perform validations for which there is no out-of-the-box validator. In the Register example, validating that a user's zip code is properly formatted falls into that category. There's a specific regular expression that can be used, which checks if the input is a traditional five-digit zip code or an extended nine-digit zip code.

The framework doesn't include a validator that can employ regular expressions; but creating validators in Tapestry involves just two methods, so it's simple to create one of our own. Listing 5.5 is the implementation of this validator.

Listing 5.5 PatternValidator.java: validator based on regular expressions

```
package examples.register;

import org.apache.tapestry.ApplicationRuntimeException;
import org.apache.tapestry.form.IFormComponent;
import org.apache.tapestry.valid.BaseValidator;
import org.apache.tapestry.valid.ValidatorException;
import org.apache.oro.text.regex.MalformedPatternException;
import org.apache.oro.text.regex.Pattern;
import org.apache.oro.text.regex.PatternCompiler;
import org.apache.oro.text.regex.Perl5Compiler;
import org.apache.oro.text.regex.Perl5Matcher;

public class PatternValidator extends BaseValidator
{
  private String _pattern;
  private Pattern _compiledPattern;
  private String _errorMessage;
  private Perl5Matcher _matcher;

  public String toString(IFormComponent field,
    Object value)
  {
    if (value == null)
      return null;

    return value.toString();
  }

  public Object toObject(IFormComponent field,
    String input)
    throws ValidatorException
  {
    if (checkRequired(field, input))
      return null;

    if (!match(input))
      throw new ValidatorException(errorMessage, null);

    return input;
  }

  protected boolean match(String input)
  {
    if (_compiledPattern == null)
    {
      PatternCompiler compiler = new Perl5Compiler();

      try
      {
```

1 Converts object to string

2 Converts submitted value

3 Returns if input blank

4 Matches input against regular expression

5 Compiles and caches regular expression

```
        _compiledPattern = compiler.compile(_pattern);
      }
      catch (MalformedPatternException ex)
      {
        throw new ApplicationRuntimeException(ex);
      }
    }

    if (_matcher == null)
      _matcher = new Perl5Matcher();

    return _matcher.matches(input, _compiledPattern);
  }

  public String getErrorMessage()
  {
    return _errorMessage;
  }

  public String getPattern()
  {
    return _pattern;
  }

  public void setErrorMessage(String errorMessage)
  {
    _errorMessage = errorMessage;
  }

  public void setPattern(String pattern)
  {
    _pattern = pattern;
    _compiledPattern = null;      ❻ Updates pattern
  }                                  and clears cache
}
```

❺

❶ The `toString()` method converts the value (obtained from the ValidField's `value` parameter) into a string, which becomes the `value` attribute of the `<input>` element.

❷ The `toObject()` method converts a string entered by the user back into an object value, throwing a `ValidatorException` if the input is improperly formed or otherwise invalid.

❸ The `checkRequired()` method checks to see if the input is blank. It throws a `ValidatorException` if the input is blank and the field is required. Otherwise, it returns true if the input is blank and false otherwise.

❹ The `match()` method does the regular expression pattern matching.

❺ Converting a string into a regular expression is somewhat expensive, so it's done only the first time `match()` is invoked.

❻ If the pattern changes, the cached compiled pattern should be discarded.

`PatternValidator` extends the framework class `BaseValidator`. `BaseValidator` is abstract and implements the `IValidator` interface. The `BaseValidator` class provides the boolean `required` property, plus a bit of support for client-side scripting (which allows the validator to generate client-side JavaScript to perform validations entirely within the client web browser).

The two key methods in a validator are `toString()` and `toObject()`. The `toString()` method is used to convert an object (read from the ValidField's `value` parameter) into a string. This method is used when the ValidField renders; the converted string is used as the `value` attribute of the HTML `<input>` element.

A validator should always be able to translate a null value to a string. It is acceptable to return null from `toString()` if the value passed in is null; this is what the validators provided with the framework do.

The meat of the validator is in the complementary `toObject()` method. This method is invoked when the form is submitted. The purpose of `toObject()` is to convert a string, supplied by the end user, into an object, such as an `Integer` or `Date`—whatever is appropriate for the specific type of validator. This is where all conversions and validations take place. If the string can't be converted, or the value is invalid for other reasons, the method throws a `ValidatorException` that is caught by the ValidField and used to record an error for the field.

The first step in the `PatternValidator`'s implementation of `toObject()` is to invoke the method `checkRequired()`, which is supplied by the `BaseValidator` class. This method performs two functions: It returns true if the input is null or empty (an empty string is length zero, or contains only whitespace), and it also throws a `ValidatorException` if the validator is required (and the input is null or empty). As configured in the Register page, the zip code field is optional (not required), so if the user decides not to enter a value, the field will not be in error.

Assuming a value was supplied by the user, the `toObject()` method continues by invoking the `match()` method. If the input from the user does not match the regular expression pattern configured for the validator, then a `ValidatorException` is thrown. The exception is built around the supplied error message (another configurable property of the `PatternValidator` class). This exception is caught by the validation delegate, which records the error message for later use when the page is rendered again (to display the errors to the user).

If `match()` returns true, then the input value becomes the return value for the `toObject()` method. Ultimately, this value will be assigned to the property bound to the ValidField's `value` parameter.

The `match()` method uses the Jakarta ORO framework to compile the pattern and match the compiled pattern against user input. The `PatternValidator` does a little caching, since compiling a string to a `Pattern` object is somewhat expensive; it shouldn't be done every time.

5.4 Customizing label and field decorations

In figure 5.1, all the required fields are marked with a red asterisk (to the right of the field). A glance at the HTML template for the page, in listing 5.2, shows that these markers are not in the template itself. This is an example of field decoration, one of the functions of the validation delegate. Additionally, you've seen that FieldLabels and ValidFields are also decorated when they are in error.

The base implementation of the `IValidationDelegate` interface, `ValidationDelegate`, provides all the support for tracking fields and errors, as well as simple support for decorating fields and labels. To customize the look and feel, as shown in figure 5.2, you create your own subclass, overriding several methods related to field and label decoration.

For the Register page, just such a subclass of `ValidationDelegate` is shown in listing 5.6. It overrides several methods supplied in the `ValidationDelegate` base class, supplying application-specific look and feel.

Listing 5.6 RegisterDelegate.java: validation delegate subclass for the Register page

```
package examples.register;

import org.apache.tapestry.IMarkupWriter;
import org.apache.tapestry.IRequestCycle;
import org.apache.tapestry.form.IFormComponent;
import org.apache.tapestry.valid.IValidator;
import org.apache.tapestry.valid.ValidationDelegate;

public class RegisterDelegate                    ❶ Subclasses from
  extends ValidationDelegate                       ValidationDelegate
{
  public void writeLabelPrefix(
    IFormComponent component,                   Is called before
    IMarkupWriter writer,                       FieldLabel
    IRequestCycle cycle)                        renders
  {
    if (isInError(component))
    {
      writer.begin("span");
      writer.attribute("class", "label-error");
    }
  }
```

❷ **Encloses label with <span>**

```
  public void writeLabelSuffix(
    IFormComponent component,
    IMarkupWriter writer,
    IRequestCycle cycle)
  {
    if (isInError(component))
    {
      writer.end(); // span
    }
  }
```

Is called after FieldLabel renders

```
  public void writeAttributes(
    IMarkupWriter writer,
    IRequestCycle cycle,
    IFormComponent component,
    IValidator validator)
  {
    if (isInError())
      writer.attribute("class", "field-error");
  }
```

❸ **Is called as ValidField renders**

```
  public void writeSuffix(
    IMarkupWriter writer,
    IRequestCycle cycle,
    IFormComponent component,
    IValidator validator)
  {
```

❹ **Is called after ValidField renders**

```
    if (validator != null &&
        validator.isRequired())
    {
      writer.printRaw("&nbsp;");
      writer.begin("span");
      writer.attribute("class",
        "required-marker");
      writer.print("*");
      writer.end();
    }
```

Marks required fields

```
    if (isInError())
    {
      writer.printRaw("&nbsp;");
      writer.beginEmpty("img");
      writer.attribute("src",
        "images/field-error.png");
```

Marks fields that are in error

```
        writer.attribute("width", 16);
        writer.attribute("height", 16);
      }
    }
  }
```

Marks fields that are in error

❶ The framework class `ValidationDelegate` provides the majority of the behavior for a validation delegate. Generally, a subclass is needed only as in this example: to override the methods used for decorating fields and labels.

❷ If the field for the label is in error, then an HTML `<span>` is wrapped around it, with a class of `label-error`. The stylesheet for the page modifies the enclosed text to make it red.

❸ `writeAttributes()` is invoked as the ValidField writes the `<input>` tag and allows the validation delegate to write additional attributes if the field is in error. Here, the CSS class `field-error` is written into the `<input>` tag if the field is in error, resulting in the white-on-red display.

❹ `writeSuffix()` is invoked after the ValidField renders the `<input>` tag. The `IValidationDelegate` interface defines an additional method, `writePrefix()`, which is invoked before the ValidField renders; and the `ValidationDelegate` base class provides a do-nothing implementation of that method.

The first two methods in the class, `writeLabelPrefix()` and `writeLabelSuffix()`, are related to decoration of field labels. The FieldLabel component invokes these methods, passing the ValidField it is connected to as the `component` parameter. These two methods rely on the `isInError()` method, which returns true if a validation error has been recorded for a component. When `isInError()` returns true, the delegate renders a `<span>` tag and writes the `class` attribute for it (as `label-error`). The `IMarkupWriter` interface is provided by Tapestry; it is much like a `java.io.PrintWriter`, but with additional methods for streamlining the output of elements and attributes, as shown in listing 5.6. Chapter 7 includes more information about `IMarkupWriter`.

The `<span>` tag started in `writeLabelPrefix()` is ended inside the `writeLabelSuffix()` method when it invokes `end()` on the writer. In between the two methods, the FieldLabel gets the `displayName` from the ValidField and writes that to the writer. For example, if the First Name field is in error (because the user submitted the form without filling in a value), then the FieldLabel, combined with the validation delegate, will render the following:

```
<span class="label-error">First Name</span>
```

Just as the FieldLabel delegates part of its rendering to the validation delegate, so does the ValidField. Three methods of the validation delegate are used to decorate fields: `writePrefix()`, `writeSuffix()`, and `writeAttributes()`. `RegisterDelegate` implements the latter two methods; the default `writePrefix()` method, inherited from the `ValidationDelegate` base class, does nothing. The `writePrefix()` method is invoked by the ValidField before it renders the `<input>` tag.

The `writeAttributes()` method is called by the ValidField after it has written the `<input>` tag and all of its attributes, but before it closes the tag. This gives the validation delegate a chance to add attributes to the tag. In this case, the delegate checks if the current component is in error; if so, a `class` attribute is added to the `<input>` tag. This CSS class is combined with the page's stylesheet to display offending fields as white text against a red background.

The `writeSuffix()` method is invoked by the ValidField after it closes the `<input>` tag. `RegisterDelegate` queries the ValidField's validator (which is passed in to the method as a parameter) to see if the validator requires a value. If so, the delegate writes HTML to display the required marker. Likewise, if the field is in error, HTML is written that marks the field using a specific image.

NOTE Normally, it is not necessary for your implementation of the `writeSuffix()` method to check that the validator parameter is not null. A ValidField component will always have a validator and will pass it into this method. This check is necessary because, in chapter 8, we create a new type of component that uses a validation delegate but does not have a validator.

As you can see, the methods defined by the `IValidationDelegate` interface are designed to be completely open-ended. With a minimal amount of effort, you can customize the look and feel of your application's labels and fields by creating and using your own subclass of the `ValidationDelegate` base class.

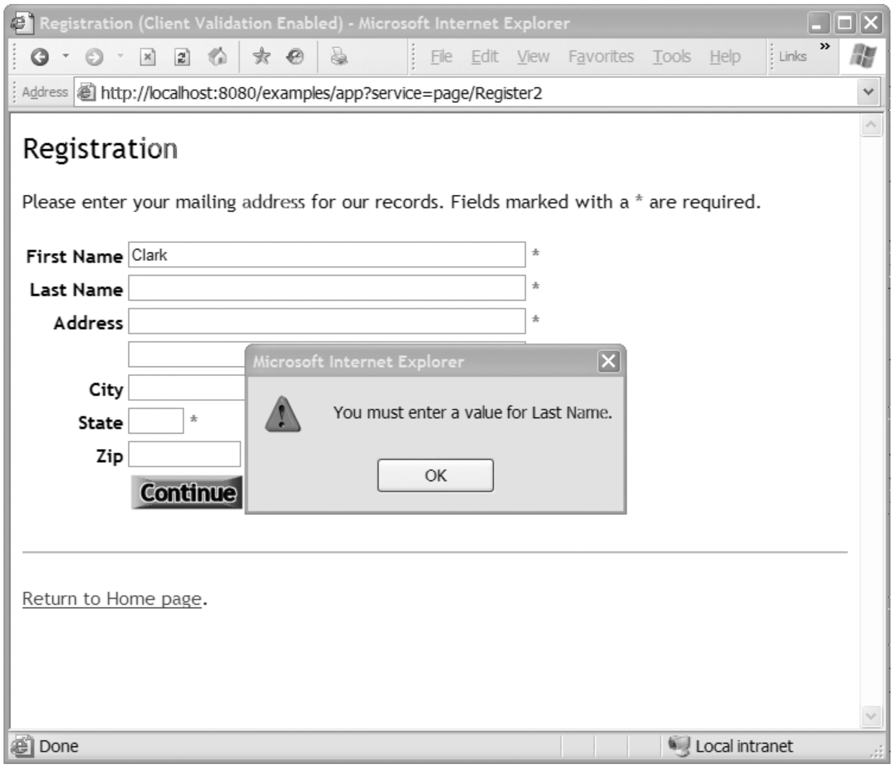
So far, we've covered how the validation subsystem extends normal form processing (both when rendering a page and when the form is submitted). That's only part of the story, however, since the validation framework can also contribute to the client side.

5.5 Enabling client-side validation

Performing validations when the form is submitted is a powerful approach, but an even better solution is to not submit the form until the fields are valid within the client web browser. Using a client-side solution gives users more immediate feedback (as soon as they click the submit button) because no additional round-trip to the server is required. Server-side validation will still occur when the form is submitted, since the client can't be counted on to have JavaScript enabled. In addition, for basic security and data integrity reasons, you must never rely on the client web browser to enforce any rules (in fact, you can't even count on the software at the other end of the HTTP connection to be a web browser; it could be a web spider, a screen scraper, or some kind of script).

Using Tapestry and the ValidField component, you can integrate this kind of client-side validation seamlessly into your applications. The ValidField's validator is responsible for generating this client-side JavaScript. When enabled, JavaScript event handlers are created and hooked into the form submission to perform validations within the client web browser similar to those that occur in the server.

The examples include a second version of the Register page, Register2. It is virtually identical to the first page, except that in the page specification, client scripting is enabled. If users attempt to submit the form with missing or invalid data, a client-side pop-up window advises them of the error. Figure 5.7 demon-


Registration (Client Validation Enabled) - Microsoft Internet Explorer

http://localhost:8080/examples/app?service=page/Register2

Registration

Please enter your mailing address for our records. Fields marked with a * are required.

First Name	Clark	*
Last Name		*
Address		*
City		
State		*
Zip		

Microsoft Internet Explorer

You must enter a value for Last Name.

Figure 5.7 With client validation enabled, submitting the form causes validations to occur. After the user clicks the OK button, the cursor will be moved to the Last Name field.

strates what happens if you submit the form without providing a value for the Last Name field, which is required.

Enabling this support requires a small change to the page specification. The bean property `clientScriptingEnabled` must be set to true for each of the validators:

```
<bean name="required"
  class="org.apache.tapestry.valid.StringValidator"
  lifecycle="page">
  <set-property name="required" expression="true"/>
  <set-property name="clientScriptingEnabled" expression="true"/>
</bean>

<bean name="stateValidator"
  class="org.apache.tapestry.valid.StringValidator"
  lifecycle="page">
  <set-property name="required" expression="true"/>
  <set-property name="minimumLength" expression="2"/>
  <set-property name="clientScriptingEnabled" expression="true"/>
</bean>
```

To support browsers where JavaScript does not exist or is disabled, all validations still occur on the server when the form is submitted. Some validations may not be possible on the client side; they may be too complex to express in JavaScript or require access to data that isn't available in the client, such as information from a database.

You've already seen how to create new validators that perform server-side validations. Adding client-side validations for a new validator is more involved; Tapestry includes the necessary tools for dynamically generating the JavaScript (this is covered in chapter 8), and the `IValidator` interface includes a method for this purpose, `renderValidatorContribution()`. Nevertheless, getting validations to work still requires a fairly deep understanding of both Tapestry and JavaScript.

To accomplish client-side validation, each validator must

- Create a JavaScript function that accesses the field value and performs the validations; if invalid, the function must display an error window and return false
- Adapt the function to Tapestry-generated IDs for the form and the text field
- Register the script function with the Form component as a client-side submit event handler

The `BaseValidator` class, a base class implementing `IValidator` from which most validator classes extend, includes several methods for supporting client-side scripting. The key challenge is the correct generation of the JavaScript needed in the client, which uses techniques discussed in chapter 8.

In addition to adding client-side scripting support to new validators, you can change the client-side scripting support for the existing validators. The existing validators provide reasonable client-side scripting support. As shown in figure 5.7, invalid input will cause a pop-up window to appear that indicates the problem and names the field; clicking the OK button will set input focus to the field in error and select all text in the field. A demanding application may want to do more—for example, change the CSS class for fields that are in error to visually highlight such fields for the user, or display the error messages for fields on the page itself, rather than in a pop-up window. Such application-specific functionality will require application-specific client-side JavaScript, which can be supplied by configuring the validators' instances with the custom scripts (again, using techniques described in chapter 8).

So far, all discussion of validation has concerned individual fields in isolation. In many cases, there are dependencies between fields that must also be validated—form-level validations that involve two or more fields.

5.6 Handling form-level validations

Validations that involve more than one field occur inside the Form's listener method. For example, a form-level validation may check that two input dates are in ascending order, and display an error message if they are not. This is demonstrated in figure 5.8.

There's no need to throw away the rest of the validation subsystem to handle these kinds of cases; it's possible for a Form's listener method to mark fields in error, just as easily as a validator can. This is accomplished by putting additional validation logic in the Form's listener method.

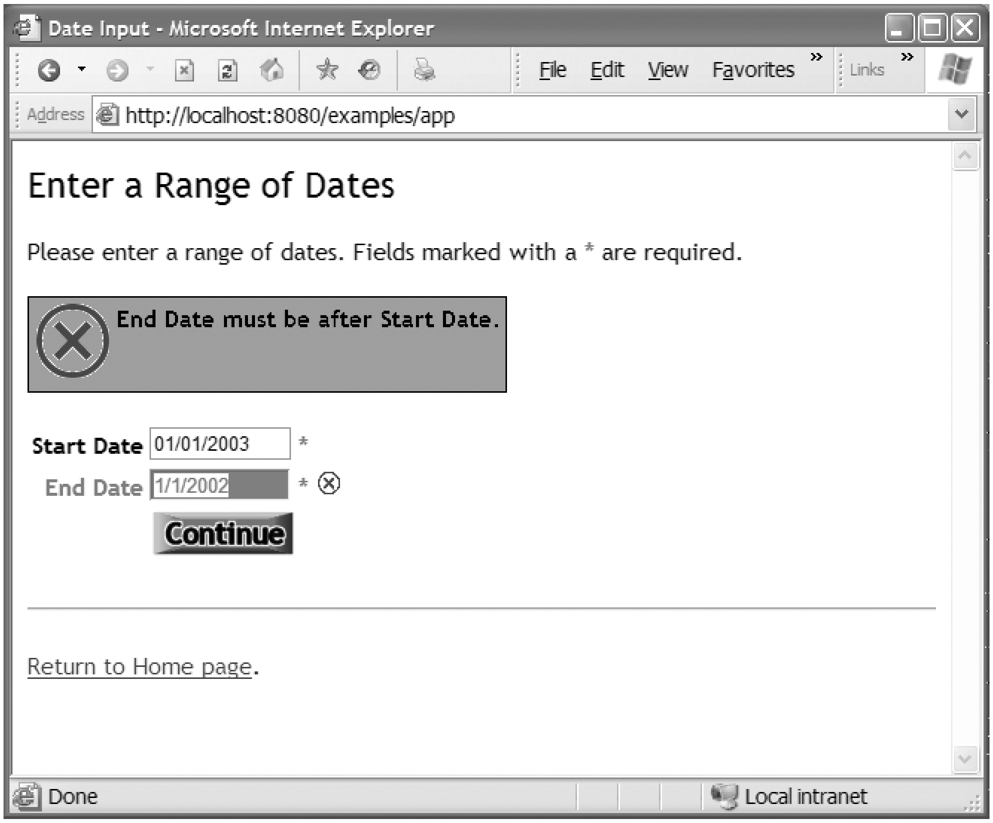
The example shown in figure 5.8 is for the Dates page. This page uses two ValidField components, inputStart and inputEnd, to edit two page properties (`startDate` and `endDate`, respectively). Listing 5.7 shows the page class for the Dates page.

Listing 5.7 Dates.java: Java class for the Dates page

```
package examples.dates;

import java.util.Date;

import org.apache.tapestry.IRequestCycle;
import org.apache.tapestry.form.IFormComponent;
import org.apache.tapestry.html.BasePage;
import org.apache.tapestry.valid.IValidationDelegate;
```


Date Input - Microsoft Internet Explorer
File Edit View Favorites Links
Address http://localhost:8080/examples/app

Enter a Range of Dates

Please enter a range of dates. Fields marked with a * are required.

End Date must be after Start Date.

Start Date 01/01/2003 *
End Date 1/1/2002 *

Continue
Return to Home page.
Done
Local intranet

Figure 5.8 Although the input values are valid dates, the end date precedes the start date and is rejected with an error message.

```
import org.apache.tapestry.valid.ValidationConstraint;

public abstract class Dates extends BasePage
{
  public abstract Date getStartDate();
  public abstract Date getEndDate();

  public void formSubmit(IRequestCycle cycle)
  {
    IValidationDelegate delegate =
     (IValidationDelegate) getBeans().
       getBean("delegate");

    if (delegate.getHasErrors())
      return;
```

Obtains validation delegate helper bean

Returns if ordinary validation errors occurred

```
        Date startDate = getStartDate();
        Date endDate = getEndDate();

        if (startDate.after(endDate))
        {
          IFormComponent inputEnd =
            (IFormComponent) getComponent("inputEnd");
          delegate.setFormComponent(inputEnd);
          delegate.record(
            "End Date must be after Start Date.",
            ValidationConstraint.CONSISTENCY);
          return;
        }

        DatesConfirm next =
          (DatesConfirm)cycle.getPage("DatesConfirm");
        next.setStartDate(startDate);
        next.setEndDate(endDate);
        cycle.activate(next);
      }
    }
```

❶ Records a form-level validation error

❶ The error is attributed to the second ValidField, the one that gets the end date. The call to `setFormComponent()` identifies the field to be assigned the error. The call to `record()` provides the error message.

This listener method starts in much the same way as in the previous example; it gets the validation delegate and simply returns if the delegate already has errors. Validation errors at this point are formatting errors in the user input, or are the result of the user omitting one of the fields (both of which are required). The remaining code in the method is executed only if there are no fundamental input errors, in which case both the `startDate` and `endDate` properties will have been supplied by the respective ValidField components (inputStart and inputEnd).

If the start date occurs after the end date, then a form-level validation error occurs. The first step is to identify the field to be associated with the error. This isn't strictly necessary; it is acceptable to not invoke the `setFormComponent()` method, in which case the error is recorded but will not be associated with any specific field within the form.

In this example, we identify the end date as the error field (we could just as easily mark the start date). The validation delegate's `record()` method lets us assign an error to the field. When the page is re-rendered, the inputEnd field will be marked in error. There are several implementations of the `record()`

method, each taking different parameter types for the message. The final parameter is of type `ValidationConstraint`, an `Enum` of different possible reasons the field is in error.[5] If none of the provided constraint values are appropriate, then null is an acceptable value. `CONSISTENCY` is used to indicate that a cross-field consistency error occurred.

Once the field is recorded as containing an error, the listener method returns. This action will cause a redisplay of the active page, complete with error messages and field decorations.

So far, you've seen a number of ways to mix and match FieldLabel and ValidField components with validators and validation logic inside listener methods. With some extra effort, it is possible to use much of the validation system without using the ValidField component, as you'll see in the next section.

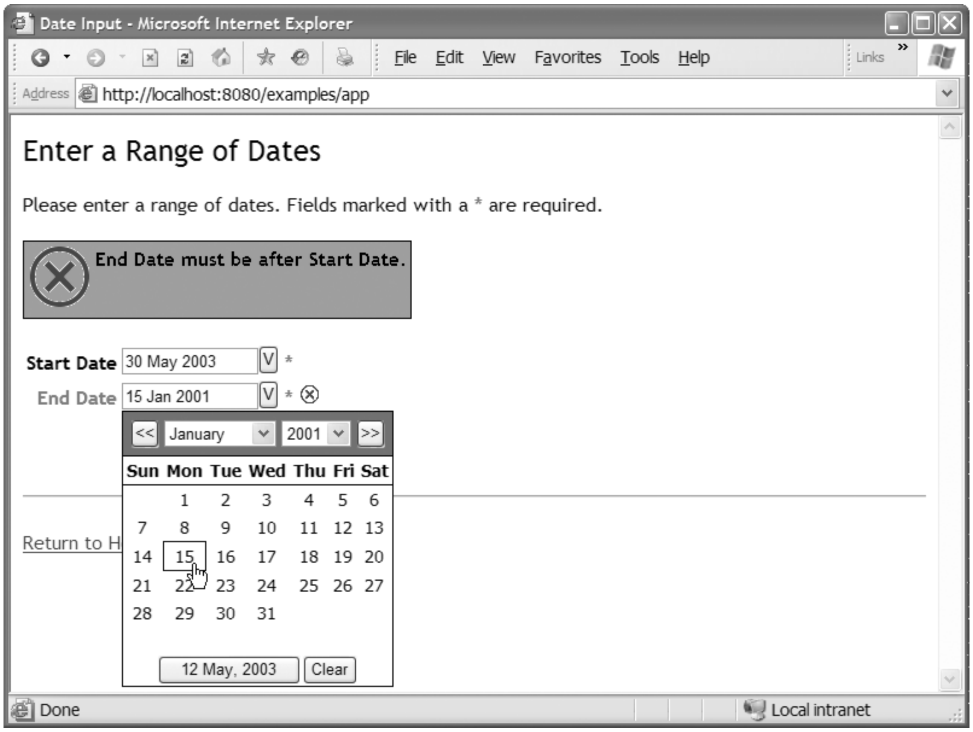
5.7 Using validation without ValidField

There are times when you will want to mix and match, leveraging portions of the validation subsystem without necessarily using the ValidField component. For example, figure 5.9 shows a different version of the page from the previous example. This page uses DatePicker components, instead of ValidField components, to collect the start and end date from the user, yet it still uses the rest of the validation framework (FieldLabel components error display, and a validation delegate).

For the most part, the HTML template for the revised Dates page is the same as in the prior example (using ValidField), with some minor differences around the labels and fields. It is possible to use the FieldLabel component, although the HTML template is somewhat more involved:

```
<tr>
<th><span jwcid="@FieldLabel"
      displayName="Start Date"
      field="ognl:components.inputStart">       ❶ Labels the field
      Start Date</span></th>                       as before
<td><input type="text" jwcid="inputStart"       ❷ Contains the DatePicker
      size="10"/>                                  component
```

[5] The validation constraint is not used by the default implementation of `IValidationDelegate` or by our custom subclass. It is provided in speculation that a more clever validation delegate implementation may have a need for it—such as customizing the decoration for an invalid field based on the type of constraint violated.


Date Input - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Links
Address http://localhost:8080/examples/app
Return to H
Done
Local intranet

Enter a Range of Dates

Please enter a range of dates. Fields marked with a * are required.

End Date must be after Start Date.

Start Date 30 May 2003 *

End Date 15 Jan 2001 *

<< January 2001 >>

Sun	Mon	Tue	Wed	Thu	Fri	Sat
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31			

12 May, 2003 Clear

Figure 5.9 The DatePicker component can be used with the form validation framework.

```
        <span class="required-marker">*</span>
  <span jwcid="@Conditional"
      condition="ognl:beans.delegate.inError">
        <img src="images/field-error.png"/>
  </span>
  </td>
  </tr>
```

Decorates the field

❶ The FieldLabel component can still be used, but the display name must be specified as a parameter.

❷ Without a ValidField, the validation delegate is not integrated into the render. The template must include the decorations normally provided by the validation delegate: the marker for required fields and, optionally, the image used to mark error fields. The validation delegate's `inError` property is true when the current field is marked as in error.

When using a ValidField, the `displayName` (used by the FieldLabel component) is provided as a parameter of the ValidField. Without a ValidField, you must instead set the `displayName` parameter directly on the FieldLabel component.

All form-related components within a Form are tracked by the validation delegate even if they are not ValidField components. This allows errors to be attached to any kind of form control component: TextField, TextArea, PropertySelection, even DatePicker (as in this example). However, the validation delegate is *not* integrated into the rendering of these other types of components; only ValidField has all the necessary hooks. The work the validation delegate normally does, decorating invalid fields, is instead done explicitly in the template, using a Conditional component.

The validation delegate's `inError` property is true if the most recently rendered component has an error. Because the Conditional in the template follows the DatePicker component, the `inError` property will be true if there was a validation error for the DatePicker.

Because there is no ValidField, and therefore no validator, involved in form processing, even simple checks for required fields must now be done inside the Form's listener method:

```
public void formSubmit(IRequestCycle cycle)
{
  IValidationDelegate delegate =
    (IValidationDelegate) getBeans().getBean("delegate");

  Date startDate = getStartDate();          Gets values set by
  Date endDate = getEndDate();              DatePicker components

  if (startDate == null)
    error(delegate, "inputStart",
      "Start Date is required.",
      ValidationConstraint.REQUIRED);          ❶ Checks for
                                                 missing
  if (endDate == null)                           required fields
    error(delegate, "inputEnd",
      "End Date is required.",
      ValidationConstraint.REQUIRED);

  if (delegate.getHasErrors())
    return;

  if (startDate.after(endDate))
  {
    error(
      delegate,                                    ❷ Performs
      "inputEnd",                                    form-level
      "End Date must be after Start Date.",          check
      ValidationConstraint.CONSISTENCY);
    return;
  }
```

```
    DatesConfirm next =
      (DatesConfirm) cycle.getPage("DatesConfirm");
    next.setStartDate(startDate);
    next.setEndDate(endDate);
    cycle.activate(next);
  }

  private void error(
    IValidationDelegate delegate,
    String componentId,
    String message,
    ValidationConstraint constraint)
  {
    IFormComponent component =
      (IFormComponent) getComponent(componentId);

    delegate.setFormComponent(component);
    delegate.record(message, constraint);
  }
```

❶ These checks are normally done by the validator for the ValidField. Since the form uses DatePicker components instead of ValidField components, there is no validator to perform even this basic check.

❷ This form-level check is the same as in the previous example (which used ValidField components).

Despite the larger amount of code (some of which could be factored out to base classes or helper beans), the process is still straightforward: We check the properties edited by the DatePicker components, compare the values, and inform the validation delegate if there is an error.

5.8 Summary

Tapestry's validation subsystem, centered around the ValidField component, is a tremendous boon to web application usability. The validation subsystem achieves the framework's goals for simplicity and consistency, both from the end user's perspective and from your perspective as the application developer. Tapestry comes with a number of predefined validations but is completely open-ended in terms of adding new ones. With only a small amount of coding, it is possible to precisely control the look and feel of label and field decorations.

Once again, large amounts of coding disappear into the framework, and what little coding remains (in terms of new validators and validation delegates) can be easily reused within a single application, or even across multiple applications. Tapestry makes it simple to create a polished, usable user interface.

Along the way, you've also seen another technique for extending the functionality of the application with little or no coding: helper beans. Helper beans fit into the overall Tapestry framework of objects, methods, and properties, providing another dynamic way to glue custom behavior into prepackaged components such as the ValidField.

In addition, you've seen that the validation subsystem can play well with ordinary form control components, such as the DatePicker component. You are free to mix and match the pieces that best solve your particular application's issues. The ordinary form control components, the advanced components such as PropertySelection and DatePicker, and the validation subsystem can all be used together to provide a comprehensive, yet still light and agile, solution to handling all variations of form input in a Tapestry application.

JAVA

TAPESTRY IN ACTION

Howard M. Lewis Ship

TAPESTRY 3.0

Tapestry is an open source Java web framework with a unique approach: it represents all behavior and all state as standard Java objects, methods, and properties. In the stateless world that is the Web, the Tapestry developer is relieved of the onerous burden of managing state.

This book is an introduction to Tapestry and a guide to the world of Tapestry development. It shows you how you can create complex applications by combining HTML with the framework's components, and connecting them to small amounts of application-specific logic. It illustrates the practical benefits of Tapestry's inbuilt management of state and its clean separation of presentation logic from business logic.

The book is written to be accessible to new Tapestry users and even to developers new to Java web application development in general. Later chapters discuss more advanced topics including integration with J2EE and team development.

What's Inside

- Tapestry's Component Object Model
- Write new components
- Configure third party components
- Dynamic JavaScript integration
- Form validation
- Tapestry/JSP integration
- Localization/internationalization
- J2EE integration

A professional developer for fifteen years, **Howard Lewis Ship** has worked with Java web applications since 1997. He is the creator and the principal architect of the open source Tapestry project.

"*Tapestry in Action* is masterfully written, making this elegant framework accessible to all Java web developers."

—Erik Hatcher, co-author of *Java Development with Ant*

"There is a better, more elegant way to build web apps—*Tapestry In Action* absolutely rocks!"

—Bill Lear
Wayport Inc./DejaNews

"Tapestry is *the way* ... and this book amply demonstrates that there is no better authority on the subject than Howard Lewis Ship."

—Geoff Longman
Intelligent Works,
developer of Spindle for Eclipse

"I found this book just right—for newcomers and experienced Tapestry developers alike."

—Richard Lewis-Shell, Techcon

"Keep your html code-free—write OO web pages the Tapestry way!"

—Joel Trunick, SmartPrice.com

www.manning.com/lewisship

AUTHOR ONLINE Author responds to reader questions

Ebook edition available

54495
9 781932 394115
ISBN 1-932394-11-7

MANNING $44.95 US/$67.95 Canada