Manning Writing Devices

1.1 Introduction

When thinking about writing technical books, it's useful to introduce an organizational artifice. Think of writing as occurring at two different levels. At the base level, you're informing: You're explaining the technical facts, relationships, and so on. At this base level, your mind is focused on getting the technical content right and making sure what you say is complete. At a second level, hovering above the first, you're focused on something entirely different: whether you're getting through to the reader. At the higher level, the content may still contain technical information, but its reason for being is to help readers understand, to pique their interest in the subject, to orient them in a multistep technical description, and so on. This second level

- Helps readers keep track of where they are
- Gives them an idea of what they're about to read
- Introduces the background for coming material
- Interests and motivates them to continue reading
- Helps them get started in a difficult topic
- Builds their trust that you understand their needs
- Helps their orientation: where they were, are, and are going within the content of your book

A collective term for text written at this level is *writing devices*. Having effective writing devices is a necessary prerequisite for a successful book. This kind of material is often difficult for new authors. The purpose of this document is to guide you in writing this kind of text. For each type of writing device, we'll give a brief description, some examples drawn from Manning books, a short description of the process for deciding when to use it, and some rules of thumb on its use.

The writing devices we'll describe have names such as *background*, *crutch*, *segue*, *color*, and *metatext*. We also discuss how code annotations are created in Manning books, and we define a broad term, *browsables*, that we use to describe a *way of presenting* standard components like figures and tables.

1.2 Segue

A *segue* is a sentence or a few sentences that make a transition from what came before to what is coming next. Segues are used when the topic is changing. They serve to reduce the "mental friction" as the reader moves through the text. A segue is a smaller version of metatext (described next) in that it only mentions what came before and what comes afterward. It can be used at any level of the manuscript, for transitions between paragraphs, subsections, sections, and chapters. A useful way to think of segues is as a connection between different lines of thought or argument. They can of course occur within sections when an argument gets long or difficult to follow. Other (possibly better) terms for *segue* are *bridge* and *transition*.

Segues should also be used for transitions to other devices, such as metatext, background, and crutches.

Usage rules of thumb

Segues should be used fairly often but not on every transition. They should be used at the end of every chapter. They should be used in every section where the content changes. They should also be used to announce most of the other, larger devices. That is, they're the supporting devices described in the following devices.

Process

Using segues requires two steps recognizing that a transition is taking place, and recognizing that it's a big enough transition to warrant a segue.

Examples

Example 1, from *Spring in Action* (Craig Walls and Ryan Breidenbach, 2005):

> [end of section 7.3.1] …Unfortunately, that's a limitation of using Java's Timer. You can specify how often a task runs, but you can't specify exactly when it will be run. In order to specify precisely when the e-mail is sent, you'll need to use the Quartz scheduler instead. ❶
>
> ***7.3.2 Using the Quartz scheduler***
>
> The Quartz scheduler provides richer support for scheduling jobs. Just as with Java's Timer, you can…

❶ This is a relatively large segue. It connects the previous text, a problem with Timer in Java, with the next subsection on the Quartz scheduler.

Example 2, from *JUnit In Action* (Vincent Massol with Ted Husted, 2003):

> Happily, the JUnit team has saved you the trouble. The JUnit framework already supports registering or introspecting methods. It also supports using a different classloader instance for each test, and reports all errors on a case-by-case basis. Now that you have a better idea of why you need unit testing frameworks, let's set up JUnit and see it in action. ❷
>
> **1.4 Setting up JUnit**
>
> JUnit comes in the form of a jar file (`junit.jar`). In order to use JUnit to write your application tests, …

❷ This is a more typical segue. It acknowledges what came before and announces what is coming next.

Example 3, from *JUnit In Action* (Massol):

> Unit tests can easily simulate error conditions, which is extremely difficult to do with functional tests (it's impossible in some instances). However, making a decision about your need for unit tests based solely on test coverage criteria is a mistake. Unit tests provide much more than just testing, as explained in the following sections. ❸

❸ Another simple segue

1.3 Metatext

Metatext helps the reader locate the present point in a long chain of an argument or a multistep explanation. It isn't a part of the argument per se but is instead a navigational aid. Whereas background (described next) prepares readers for what's to come by giving them prerequisites and the larger-scale knowledge they need, metatext guides them through the argument or explanation. Background comes ahead of the core text; metatext is inserted within the text. Background is generally longer, metatext shorter.

Metatext consists of words that step back from the flow of the book content and make observations about the argument: where it is, where it was, and where it's going. Metatext differs from a segue in that the point of view is from a place "above" the content. Metatext often includes acknowledgement of readers' reactions to the content. It recognizes and acknowledges readers' feelings at this particular point in the book, and assures them that they're in good hands. Metatext can recognize reactions to the material just completed, provide motivation for what comes next, and justify what came before.

Use metatext to reward your reader for getting through a particularly difficult section. Sometimes it's a good idea to say something encouraging to your reader just because they're still reading your book.

Rewards such as, "That's it, that's all you need to know about topic XYZ"" bring closure and can be encouraging to readers.

Although metatext is often used to acknowledge potential difficulty, it's also a good idea to insert metatext with some regularity in the text. Think of metatext as route signs on highways, or blazes on a trail in the woods. There are always signs before and after every branch or intersection; these signs give the traveler comfort and confidence that they're on the right road.

Supporting devices

Metatext can be introduced with something like, "Let's pause here and see where we are headed."

Usage rules of thumb

A mechanical approach to metatext is to say that any topic of more than two pages should have some metatext in the middle. In addition, every unit (section or chapter) that is of above-average complexity for your book should be introduced with metatext.

Rationale; process

The most effective metatext anticipates the reader's reaction to the material. So, the process for identifying places for metatext should be to explicitly ask yourself what readers will be feeling as they read this text. If answers to that question don't come easily, have some friends read the text, and ask them how they felt at each of the transitions.

Examples

Example 1, from *Windows Forms Programming with C#* (Erik Brown, 2002):

> If you have prior experience with Windows programming, you will see many similarities in the names of the .NET controls. This chapter will show some of these names, and introduce some new terms and features as well. If you're new to Windows programming, you'll find this chapter a good foundation for the remainder of the book. This chapter is a bit of a wild ride through .NET, so don't worry too much about the details here. The concepts and topics in this chapter should become clearer as we progress through the book. [4]
>
> This chapter assumes you have successfully installed the Microsoft .NET Framework SDK on your computer.
>
> **1.1 Programming in C#**
>
> Let's create a blank form in C# to see how a program compiles and runs in the .NET Framework. Such a form is shown in figure 1.1. …

[4] This metatext recognizes that readers may find the coming chapter somewhat overwhelming. It assures readers that the topic swill be covered again later in the book.

Example 2, from *Windows Forms Programming in C#* (Brown):

> Perhaps because of this history, many developers take the usefulness and popularity of menus for granted and don't spend sufficient time laying out a consistent, usable interface for their application. While graphical elements such as menus, toolbars, and other constructs make applications much more friendly, this isn't an excuse to ignore good user design and rely on customers to become "experienced" to make effective use of the interface.
>
> Well, if that little lecture doesn't get your creative juices flowing, then nothing will. Back in .NET-land, Visual Studio .NET provides a rather intuitive interface for the construction of menus that does away with some of the clunkiness found in earlier … 5

5 This metatext acknowledges that the author was lecturing rather than explaining.

Example 3, from *Object Oriented Perl* (Damian Conway, 1999):

> Those of us who hate having injections usually appreciate when our doctor says: "Okay, I'll count to 3: …1…2…<*jab!*>…3", and the nastiness is over before it begins.
>
> Guess what.
>
> If you've been apprehensive about this chapter—either because you've heard polymorphism is "difficult" or because you've had trouble with it in other languages—you can relax. The nastiness is over. You've already seen everything you need to know about polymorphism in the past four chapters. This chapter merely re-presents those ideas in the accepted jargon and extends them a little. 6

6 This text introduces the chapter on polymorphism. It acknowledges the fact that poyorphism is usually a difficult topic for people who are new to objects, and it tries to assure them it won't be too bad.

1.4 Background

Background material provides the technical information that readers need to understand before they can make sense of the material that is coming. Background text is placed at the beginning of the unit it serves. It can work in one of three ways:

- Provide information that is a prerequisite for understanding upcoming material. We speak of that as *looking back*. The information can be a summary of previously provided material, or it can be new.
- Provide context for the upcoming material. You can visualize this as placing the content in a wider picture or *looking outward*.
- Provide a summary of what is coming—*looking forward*.

Use background text often. Consider whether readers will benefit from a concise review of prerequisite knowledge or of the context, or from a peek at the entire picture before you ask them to sweat over all the details you have in store for them. Use it when readers are facing a big step in the upcoming material.

Background is usually bigger than the other devices we have looked at. At the beginning of a book, it's common for a whole chapter or more to be devoted to necessary background information.

Supporting devices

Background is often introduced and closed with segues of the type, "Before we look at the *main topic*, let's consider what we have already seen about ...." Closing segues announce the return to the main thread: "Now we can get back to the *main topic*."

Usage rules of thumb

Use backward-looking background (reviewing prerequisites) whenever the upcoming material may be hard to understand without a good understanding of the earlier material. It doesn't hurt to make seasoned readers go through a piece of background about prerequisite knowledge; we believe they get a kick out of noticing that they know everything the author is describing. But be wary of going on too long—that kick may quickly turn into annoyance. Use an outward-looking background (set the context) to show the connections and utility of the upcoming material. Use a forward-looking background (a heads-up as to where you're taking them) when the going will be long and difficult and you don't want readers to get lost along the way.

The length of background text depends on the unit being introduced. For a complete book, the background can easily take an entire chapter. At the smaller scale of a single chapter, the background can occupy a few paragraphs or as much as an entire section extending over a few pages. Subsections can only support short background text.

A preview-type background (forward looking) can be used in every chapter. For smaller units (section and subsection), you should be more selective.

To summarize, give readers the prerequisites if the issue is their ability to understand the unit; give them outward-looking context if the unit's relationship with other topics is complicated or unclear. Use a forward-looking preview to prevent losing readers as you plow through the upcoming content. In all cases, background text should help motivate both the content and the readers' perseverance in reading the unit.

Rationale; process

Background and the crutch (described next) are the most demanding of the author's understanding of readers' situation and needs. Your task is to step inside the reader's head to look at the material from their point of view and then decide where background text is needed.

You must take two steps before you can write an effective background:

1. Recognize the need.
2. Determine what type of background to use.

Background is always placed at the beginning of a unit containing new information. If the information is more complex or quite different from other information in the book, or if it's very important that the reader understand the information, then some background text is indicated. The act of putting off the main discussion until prerequisites, context, or preview are established serves to emphasize the coming content. Use background with that in mind.

If a chapter or section gets a lot of negative comments in the manuscript reviews, but it's written reasonably well, consider introducing some background text.

Examples

Example 1, from *Spring in Action* (Walls and Breidenbach):

> **4.1 Learning Spring's DAO philosophy**
>
> Before we jump into Spring's different DAO frameworks, let's talk about Spring's DAO support in general. ❻ From the first section, you know that one of Spring's goals is to allow you to develop applications following the sound object-oriented (OO) principle of coding to interfaces. ❼ Well, Spring's data access support is no exception.

❻ This segue announces the background text. The title of the section indicates that this material is background to the main thread.

❼ This sentence is a review type of background.

Example 2, from *JUnit in Action* (Massol):

> Earlier chapters in this book took a very pragmatic approach to designing and deploying unit tests. This chapter steps back and looks at the various types of software tests, the role they play in the application's life cycle, how to design for testability, and how to practice test-first development. Why would you need to know all this? Because performing unit testing isn't just something you do out of the blue. In order to be a good developer, you have to understand why you're doing it and why you're writing unit tests instead of (or to complement) functional, integration, or other kinds of tests. Once you understand why you're writing unit tests, then you need to know how far you should go and when you have enough tests. Testing isn't an end goal. ❽
>
> Finally, we'll show you how Test-Driven Development (TDD) can substantially improve the quality and design of your application by placing unit tests at the center of the development process.

Once you understand why you're writing unit tests, then you need to know how far you should go and when you have enough tests. Testing isn't an end goal. **#8**

Finally, we'll show you how Test-Driven Development (TDD) can substantially improve the quality and design of your application by placing unit tests at the center of the development process.

❽ This outward-looking background provides motivation for the material to come.

2.1 Crutch

Let's assume that your task is to introduce readers to a Perl feature called *reference*. The concept of reference is considered difficult for people to grasp. The leading book on Perl for beginners doesn't even mention it. As you grapple with how to present the concept to your readers, you realize that the difficulty is in the abstractness of the concept. So, if you present it in the context of an easy-to-understand example, the difficulty may go away. You have hit upon the idea of the *crutch.*

Whenever you discuss an important or difficult concept, start with an example that motivates the topic. The crutch is a special kind of introduction: It describes a scenario and illustrates how the situation can be difficult to resolve in the absence of the to-be-described concept. Another form of crutch is a scenario that demonstrates the upcoming material with a situation that is analogous to the situation being addressed. A crutch can involve readers in the text by using words like, "Imagine what you would do if…" or, "Suppose you were trying to…." After introducing the crutch, the scenario can serve as the running context in which readers can better understand the discussion of the concept.

A crutch uses an example as a story that entangles readers' thoughts. It provides a concrete environment for any abstractions you may have in store. The first paragraph in this section can itself be considered a crutch for the discussion of the writing device *crutch*. In this section, we show you an example of an implementation of a crutch for the case of Perl's concept of reference.

Supporting devices

Crutches aren't introduced with segues the way backgrounds are. They start with something like, "Assume you have to develop an *xyz* feature. You could try doing it by…." Closing the crutch with a segue, to return to the main thread, is usually OK. However, it's better to weave the example into your discussion of the main unit without a perceptible transition.

Usage rules of thumb

Crutches livens up your writing and should be used frequently, perhaps as often as at the beginning of every third subsection. Always look for opportunities to start with a crutch. You'll find that using one in some cases would be silly—akin to using a sledgehammer for a small nail. But when the nail is bigger (as in important or difficult units), a crutch should be your device of choice.

Rationale; process

The situations that call for a crutch are similar to those calling for background: Use a crutch when new, complex, or different material is being introduced. The motive for using a crutch isn't to inform readers but to give them a concrete example to help them understand a general concept, technique, or approach. (This is the reason we use the word *crutch*.)

Examples

Examples 1 and 2 are from two different Perl books.

Example 1, from *Object Oriented Perl* (Conway):

Suppose that you wanted to create a subroutine to return the items that two arrays have in common. This might come up in a program that had to compare a master list of files with a list of files that had changed, in preparation for saving the changes. How could you pass two lists to one subroutine? You couldn't just pass the lists one right after the other, like this:

```
@master_list = qw(a b c d e f g);_
@change_list = qw(b c f);_
$intersect = get_intersect @master_list @change_list;
```

Because of flattening, `get_intersect` wouldn't receive two arguments. It would get ten, one for each element of both arrays.

```
(a b c d e f g b c f) not (a b c d e f g) (b c f) ❾
```

The solution is to pass references. A *reference* is… ❿

❾ This crutch sets up a situation in the domain of the book, tries to solve it with previous knowledge, and shows that the solution doesn't work.

❿ This sentence leads to the new material that can successfully handle the problem.

Example 2, from *Need a Title Here* (Author Name, Year):

Suppose we wanted to implement a subroutine called listdir that provides the functionality of our operating system's directory listing command (i.e., dir or ls). Such a subroutine might take arguments specifying which files to list, what type of files to consider, whether to list hidden files, what details of each file should be reported, whether files and directories should be listed separately, how to sort the listing, whether directories should be listed recursively, how many columns to use, and whether the output should be pages or just dumped. But we certainly don't want to have to specify every one of those nine parameters every time we call `listdir`:

```
Listdir("*", "any", 1, 1, 0, 0, "alpha", 4, 1);
```

Even if we arranged things so that specifying an undefined value for an argument selects a default behavior for that argument, the call is no easier to code and no more readable:

```
Listdir(undef, undef, 1, 1, undef, undef, undef, 4, 1); ⓫
```

Some programming languages provide a mechanism for naming the arguments passed to a subroutine. This facility is especially useful when implementing a subroutine like listdir, where there are many potential parameters, but only a few of them may be needed for a particular call. Perl supports named arguments in a cunning way… ⓬

⓫ This crutch uses a problem from the domain of the book. It shows that the problem is dificult to address using information presented to this point.

⓬ This text provides a transition from the crutch situaion to the new material that follows.

Example 3, from *Spring in Action* (Walls and Briedenbach):

Consistent DAO support

You have probably traveled by plane before. If so, you'll surely agree that one of the most important parts of traveling is getting your luggage from point A to point B. There are lots of steps to this process. You have to drop it off at the counter. Then it has to go through security and then be placed on the plane. If you need to catch a connecting flight, your luggage needs to be moved as well. When you arrive at your final destination, the luggage has to be removed from the plane and placed on the carousel. Finally, you go down to the baggage claim area and pick it up.

As we said, there are many steps to this process. But you're only actively involved in a couple of those steps. The carrier itself is responsible for driving the process. You're only involved when you need to be; the rest is just "taken care of." [13] And believe or not, this mirrors a very powerful design pattern: the template method pattern.

A template method defines the skeleton of a process. In our example, the process is moving luggage from departure city to arrival city. The process itself is fixed; it never changes. The overall sequence of events for handling luggage occurs the same way every time: luggage is checked in, luggage is loaded on the plane, etc. Some steps of the process are fixed as well. That is, some steps happen the same way every time. When the plane arrives at its destination, every piece of luggage is unloaded one at a time and placed on a carousel to be taken to baggage claim.

But at certain points, the process delegates to other collaborators to fill in some implementation-specific details. This is the part of the process that is variable. For example, the handling of luggage starts with a passenger checking in the luggage at the counter. This part of the process always has to happen at the beginning, so its sequence in the process is fixed. But each passenger's luggage check-in is different. The implementation of this process is determined by the passenger. In software terms, a template method delegates the implementation specific portions of the process to an interface. Different implementations of this interface define specific implementations of this portion of the process. Spring applies this pattern to data access. [14]

[13] The baggage scenario is a crutch using an analogous situation that is familiar to readers.

[14] Here the scenario from the crutch is used to motivate several parts of the discussion.

2.2 Color

Color consists of words—usually adjectives and adverbs, phrases, and examples—that recognize, or appeal to, the reader's emotional reactions to the content. Examples of color adjectives include *interesting*, *unusual*, *strange*, *unexpected*, *weird*, and *silly*.

For examples of writing that contains no color, look at the user manuals that come with many software programs, or the help text for those programs. Instructions that come with your income tax forms are another place to look for colorless writing. Colorless writing is tolerable in short pieces, like a help topic, but it gets very tedious in longer pieces, like a book. That is why appropriate color is an important feature of successful books.

Color is closely connected to the "voice" of a book. Color words mark the difference between a dry user manual and an interesting book. We aren't going to discuss voice in this piece; but because color is important, authors should be aware of the personality you're projecting and choose color words and phrases so that they're consistent throughout the book and consistent with the personality. (It's easiest to project your own personality in your writing, but you still need to be aware of what that is.)

Usage rules of thumb

At Manning, we've found that books with a definite personality tend to be more popular with readers. Having said that, the amount and type of color text that is appropriate varies widely, depending on the voice and personality being used. Regardless of the style, color text is often concentrated in the writing devices described so far. A common purpose of those devices is to respond to the reader's feelings, and such a response necessarily involves the author's personality.

To be consistent, and to prevent discontinuities in the reader's experience (speed bumps), color text should probably appear every page or so. There is also an upper limit, which is when the use of color starts to intrude on, or become obvious in, the flow of the content.

Examples

The boxes indicate the color words in each of the following examples:

Visual Studio .NET provides a rather intuitive interface for the construction of menus that does away with some of the clunkiness found in earlier Windows development environments from Microsoft. Any class that inherits from _Initializable as its leftmost parent, inherits this generic constructor and, therefore, doesn't need to provide its own. The clever bit is that, having created that empty object, `_Initializable::new` then calls its `_init` method.

What Internet search sites do behind the scenes is amazing. They deal with so many information sources—millions or hundreds of millions of documents—and they usually give results within seconds.

This chapter is a bit of a wild ride through .NET, so don't worry too much about the details here. The concepts and topics in this chapter should become clearer as we progress through the book.

We could just be very careful here, and ensure that the two dialogs appear and behave in a similar manner. But who wants to be careful? Instead, this is a great chance to use a common Form for both windows to see how form inheritance works.

Example 2, from *Need a Title Here* (Author Name, Year):

Polymorphism

If you've ever gone up to someone in a bar or club and introduced yourself, you know that people can respond to the same message—*I'd like to get to know you better*—in very different ways. If we categorize those ways, we can create several classes of person: `ReceptivePerson`, `IndifferentPerson`, `ShyPerson`, `RejectingPerson`, `RejectingWithExtremePrejudicePerson`, `JustPlainWeirdPerson`.

Turning that around, we can observe that the way in which a particular person will respond to your message depends on the class of person they are. A `ReceptivePerson` will respond enthusiastically, a `ShyPerson` will respond tentatively, and a `JustPlainWeirdPerson`[5] will probably respond in iambic pentameter. The original message is always the same; the response depends on the kind of person who receives it.

Language theorists[5] call this type of behavior *polymorphism*. When a method is called on a particular object, the actual method that's involved may depend on the class to which the object belongs.

[5] …most of whom live at ground-zero in the JustPlainWeirdPerson category… ⓯

⓯ In this crutch, the color appears as the choice of the scenario to use. Many other situations could have been used. The choice of a bar situation is consistent with the light, humorous voice used in this book. The footnote is a relatively extreme use of color for a technical book.

2.3 Code examples

Having published programming books for some time now, we have learned some things about what readers want in their code examples. In addition to an abundance of code, they like and we recommend three other things:

- Show the results before you show the code.
- The discussion of the code is important—as important as the code itself.
- Readers like the Manning annotation style. Use it!

The Manning Author's Template document has much more detail about the mechanics of code annotations, but we'll also walk through their use here.

Show the results first

For every example that has an intelligible result, show the result before you show the code. Point out the relevant features of the output (that you'll be discussing in the code example), and then show the code listing. If the code produces output on the screen, show the screen shot first. If the code retrieves data, show a sample of the returned data set. This is a big help to users as they try to understand your code.

The value is in the discussion

Readers have consistently told us that the discussion of the code examples is the most valuable part of our books. Keep this in mind as you write. The relevant parts of all examples should be, at least, annotated, and often explicitly discussed in the text. This is where much of the teaching and mentoring is done.

Annotations improve the discussion

The term *code annotations* refers to comments placed in line with the code listings. For simple code, annotations can be the complete discussion of the example. For every significant listing, inline annotations serve as the connection between the code and the discussion of the code. Code annotations are also a major source of difficulty, and hence delay, during production. So, for the sake of your reader and your time to market, we recommend that you pay close attention to the annotations.

The decision about what to annotate depends on knowing what parts of the code contribute to the functionality you're describing. Another factor that influences the choice of what to annotate is the principles of *browsables*. Browsables are code examples, graphics, tables, and other nontext elements of a book. Readers often scan the browsables rather than read the code, particularly when they're looking at the book in a store and trying to decide whether to buy it. (We discuss browsables further in the next section.) Therefore, a useful test of the code annotations is to see whether the purpose and value of the code example can be obtained by reading just the inline annotations.

You can create code annotations using three styles:

- *Hedgehog diagram*—Used to describe a single line of code. Such annotations can also be used for other visuals, like screenshots. This device is most often used when you're discussing the grammar and syntax of a programming language. Two styles are shown in the examples. Hedgehog diagrams should be created in a drawing program and imported into your manuscript. Keep the individual annotation short.
- *Inline annotations*—Used for features of an example that only need to be identified; no further explanation is required. Inline annotations should be only four to six words long. The entire annotation appears inline with the listing.
- *Bulleted annotation*—Used when more explanation of the example's elements is required than can be conveyed in four to six words. Bulleted annotations can include the four- to six-word inline annotation, described earlier, and they also have a numbered bullet that is linked to a corresponding discussion below the listing.

Usage rules of thumb

Inline code annotations should be very brief in order to fit beside your line(s) of code on the typeset page. (Four to six words is ideal.) If a longer explanation is needed, link the annotation to its explanation in text that follows the code.

Annotations should have parallel construction (deletes…, defines…., creates…) and no period at the end, even if the annotation is a full sentence.

Only longer segments of code should have annotations. Ideally, listings or segments that are at least a page long should be annotated, but you should use only four or five annotations per page.

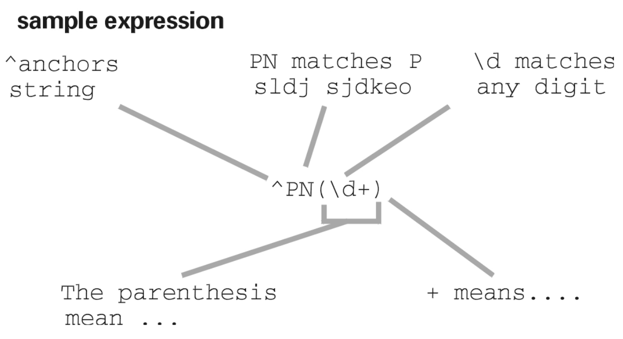
Don't crowd the annotations. It will be difficult for the typesetter to place annotations next to the code if there isn't enough space between annotations on the page.

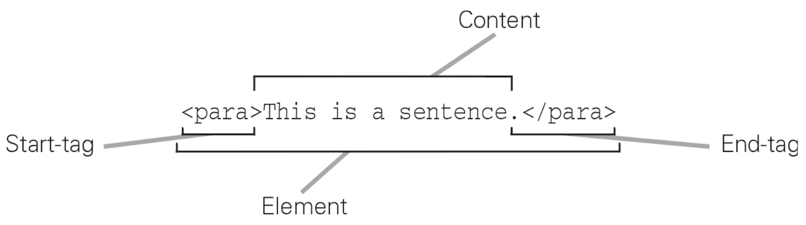
See the Author_template.doc file for more details.

Examples

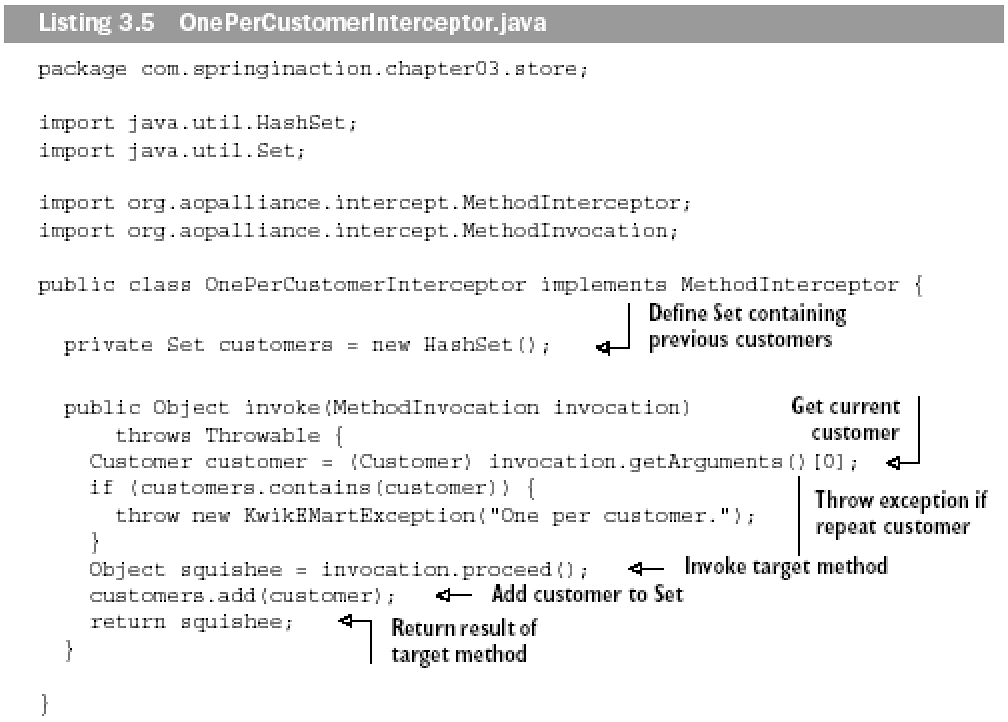
These examples show the final, typeset results. The actual entries in your manuscript use a set of conventions described in the Author_template.doc file.

Hedgehog diagrams:





Inline annotations:



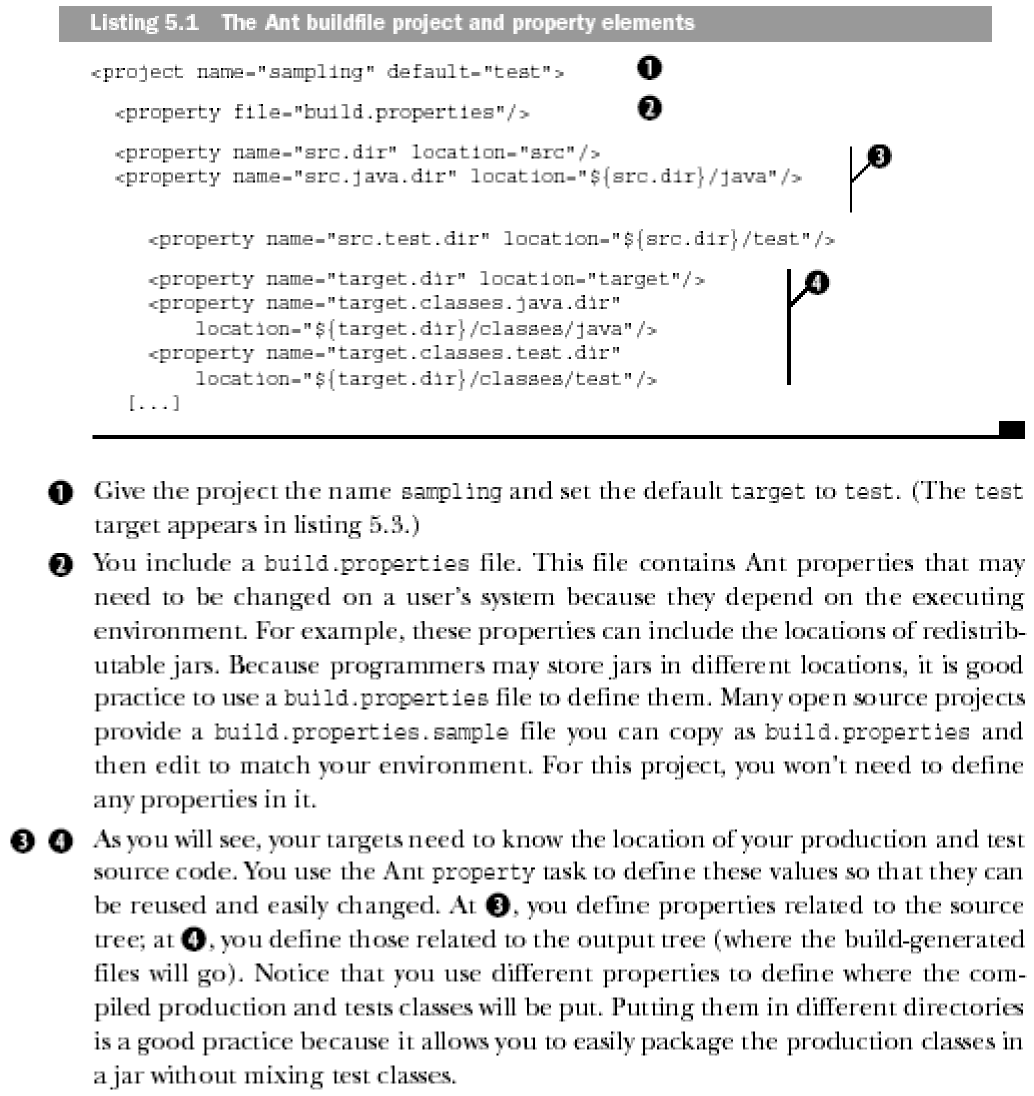
Listing 3.5 OnePerCustomerInterceptor.java

```
package com.springinaction.chapter03.store;

import java.util.HashSet;
import java.util.Set;

import org.aopalliance.intercept.MethodInterceptor;
import org.aopalliance.intercept.MethodInvocation;

public class OnePerCustomerInterceptor implements MethodInterceptor {

  private Set customers = new HashSet();

  public Object invoke(MethodInvocation invocation)
      throws Throwable {
    Customer customer = (Customer) invocation.getArguments()[0];
    if (customers.contains(customer)) {
      throw new KwikEMartException("One per customer.");
    }
    Object squishee = invocation.proceed();
    customers.add(customer);
    return squishee;
  }

}
```

Define Set containing previous customers
Get current customer
Throw exception if repeat customer
Invoke target method
Add customer to Set
Return result of target method

This example is at the upper limit of annotation density. A better approach would have been to use bulleted annotations to describe the details. For example, the inline annotation could say "Get current customer," and the discussion in the text coulde be, "Get the current customer and throw an exception if it's a repeat customer."

Bulleted annotations:



Listing 5.1 The Ant buildfile project and property elements

```
<project name="sampling" default="test">          ❶

  <property file="build.properties"/>             ❷

  <property name="src.dir" location="src"/>                          ❸
  <property name="src.java.dir" location="${src.dir}/java"/>

     <property name="src.test.dir" location="${src.dir}/test"/>

     <property name="target.dir" location="target"/>           ❹
     <property name="target.classes.java.dir"
         location="${target.dir}/classes/java"/>
     <property name="target.classes.test.dir"
         location="${target.dir}/classes/test"/>
   [...]
```

❶ Give the project the name `sampling` and set the default `target` to `test`. (The `test` target appears in listing 5.3.)

❷ You include a `build.properties` file. This file contains Ant properties that may need to be changed on a user's system because they depend on the executing environment. For example, these properties can include the locations of redistributable jars. Because programmers may store jars in different locations, it is good practice to use a `build.properties` file to define them. Many open source projects provide a `build.properties.sample` file you can copy as `build.properties` and then edit to match your environment. For this project, you won't need to define any properties in it.

❸ ❹ As you will see, your targets need to know the location of your production and test source code. You use the Ant `property` task to define these values so that they can be reused and easily changed. At ❸, you define properties related to the source tree; at ❹, you define those related to the output tree (where the build-generated files will go). Notice that you use different properties to define where the compiled production and tests classes will be put. Putting them in different directories is a good practice because it allows you to easily package the production classes in a jar without mixing test classes.

2.4 Browsables

Manning terminology recognizes two types of chapter content: the narrative and the browsables. We use *narrative* in a conventional way, but *browsable* is our invention.

Standalone pieces of the manuscript that can be understood on their own are *browsables*. They're visually identifiable as separate, and they draw the reader's eye. Each browsable contains a complete, self-standing piece of information. Browsables are common things such as tables, definition boxes, annotated code segments, and figures. These are the types of information a "quick study" (someone who knows the topic and just needs the specifics) will find useful—and, possibly, sufficient. This might be a confident reader; or a reader in a hurry; or someone who has read the chapter before and is now returning to it looking for information.

You might imagine the browsable pieces as the writing remaining on a whiteboard after a technical discussion. What would you see? Code segments with words and arrows pointing to pieces of the code; pictures with words and arrows; definitions written out in full; tables with comments attached to them, and so on. Of course, you're writing a book, not leading a whiteboard discussion, so the browsables are *complete pieces of information*, not sketches. They should be useful even if they're read in isolation from anything else in the chapter. The quick study may scan the browsable pieces in a chapter and never read the rest.

What kinds of content should you consider putting into the browsable form? Here are a few examples:

- Show the big picture in the form of a diagram that shows how everything fits together.
- Illustrate how the concepts work through several lines of code, annotated so the main points are immediately identified. (We discussed code annotations in the previous section.)
- Present rules of syntax that might require a lot of words through a hedgehog diagram (also described in the previous section).
- State definitions and principles tersely and abstractly, and present them in a specially designed text format (equivalent to a text box).
- Put any factual text with special cases and/or relationships (that often require lots of words) into something more economical and clearer: a table.

Rationale; process

Browsables present *the facts* and *only the facts*—the "what" but not the "why." Readers actively search through the book to find what they're interested in; unless they're beginners who know nothing about the topic, the browsables will tell them much of what they need.

When you're writing browsables, remember that they won't be read in order; so, try to make them independent of previous parts of the book.

Here are some suggestions of the most useful browsables. (You can find more information about code-related browsables in the previous section.)

Table

Things you want to present systematically, that include conditions, special cases, relationships, multiple examples, or values, are candidates for a table. A table must have a long, detailed caption. The caption should say enough to make the table stand alone—that is, it should be understandable (and interesting) without reading the narrative.

Definition

An important definition should be presented as a visually separate item—a text box. This definition must be written properly. Authors commonly "define" a concept by stating what it does. That's wrong. Instead, a definition must state what the concept *is*. A definition *must* identify the concept's location in the conceptual framework of which it's a part. To do that, you identify the broader concept (*genus*) of which the concept is a special case *and* identify one or more of its distinguishing features (*differentiae*).

For example, we define an insulator by identifying the genus in which it belongs—the genus of all materials—and then specifying what sets it apart from other materials: An insulator is a material that does not conduct electricity. A common mistake is skipping the genus, as in "A conductor conducts electricity." Yes, it does, but so does your hand. Is your hand a conductor? Well, not really, because when we say *conductor* we mean a type of material, which a hand isn't. This ambiguity doesn't arise when the genus is properly identified: *A conductor is a material that conducts electricity.* (For a complete discussion of this topic, see www.manning.com/Kovitz, chapter15, section "definitions.")

Figures

So many computer books are impoverished visually, we sometimes wonder whether the programming community has a cultural bias against pictures. The few illustrations you do find in such books are often overly complicated, requiring a large effort for the reader to decipher. We expect Manning authors to do better. Our rule is: Every important concept should be illustrated.

Don't try to make the picture complex. In fact, try *not* to make the picture complex. The simpler, the better. A picture showing two boxes connected with an arrow, plus a few words, is effective. Our typesetters will make the picture pretty and small and place it on the page so the text flows around it. That way, you gain the benefits without wasting space and without the reader's eye traveling too far from the related text. Letting a simple picture take up half a page insults the reader's intelligence and aesthetic sense. Making it small is aesthetically appealing.

Pictures do the following:

- Help explain the concept(s)
- Emphasize the concept(s)
- Help the reader navigate the book

The caption is an integral part of the figure, and it needs to be detailed enough that someone browsing the book can understand the point of the illustration without reading the surrounding narrative. People who look through a book first look at the illustrations and read their captions. Captions are a great opportunity to draw the reader—and the bookstore browser—into your book. The same guideline applies to tables.

Screen shots

Screen shots play the usual role of illustrating how something looks on the screen. We like to keep our screen shots small, so plan for the important parts to be visible. That means you must crop the image so that only the necessary parts are shown. Cut out the windows frames unless they're really needed—they're tiresome. Also, remember that the image will be printed in black ink, and the colors will all become shades of gray. This makes features of the image even harder to distinguish, so look for strong contrasts between light and dark. A detailed caption is required.

Information maps

An *information map* is a table-like representation of a list, a procedure, or a set of rules and requirements. It's not a conventional table, though. It transforms list-like information into a structured table in which the number of steps is reduced and related substeps are clustered into related steps. For example, the four steps outlined in the first table that follows can be grouped into two, as shown in the second table:

Step	Description
1	Grumpy opens the door and leaves
2	Bashful follows him
3	Sleepy is the last to leave
4	They walk and sing, "Hi-ho, Hi-ho …"

Step	Description
1. Dwarfs leave for work	Grumpy first He opens door, steps out and walks Bashful and Sleepy follow They fall in line behind Grumpy
2. They walk	Single-file and sing in unison: "Hi-ho, Hi-ho …"

Information maps effectively take typesetting to another level, but they aren't just a matter of visual presentation: They require writing that expressly organizes and groups flat lists into fewer items of more complex, related information. This idea shouldn't be strange to anyone who has heard of the advantages of object-oriented programming over procedural programming.

Look at the examples at www.infomap.com. In the left column of this page, find the heading The Method, and then link to Demos. The before and after demos will give you an idea of how to use information maps.

Minimal pairs

Now here's a beautiful technique. Suppose you're trying to explain the `step` attribute in the following Java code. Put the code and its output into two side-by-side columns as follows:

SOURCE: <jx:forEach var="c" status="s" items="$customers" step="**1**"> <jx:expr value="$s.index"/> <jx:expr value="$c"/> </jx:forEach> OUTPUT: 1. Andrew Aaronson 2. Brian Bournelli 3. Charles Colworth 4. Daniel Devoe 5. Erica Ellbridge	SOURCE: <jx:forEach var="c" status="s" items="$customers" step="**2**"> <jx:expr value="$s.index"/> <jx:expr value="$c"/> </jx:forEach> OUTPUT: 1. Andrew Aaronson 3. Charles Colworth 5. Erica Ellbridge

Appropriate differences are bolded: 1 vs. 2 in the source code. Such a minimal pair might be more valuable in cases where the differences are more subtle, but this example makes it clear how to use this type of browsable. You can easily set one up in a table with one row and two columns.