

Flash Application Design Solutions

The Flash Usability Handbook

Ka Wai Cheung
Craig Bryant



Flash Application Design Solutions: The Flash Usability Handbook

Copyright © 2006 by Ka Wai Cheung and Craig Bryant

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN-13 (pbk): 978-1-59059-594-7

ISBN-10 (pbk): 1-59059-594-7

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit www.apress.com.

The information in this book is distributed on an "as is" basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is freely available to readers at www.friendsofed.com in the Downloads section.

Credits

Lead Editor **Assistant Production Director**
Chris Mills Kari Brooks-Copony

Technical Reviewer **Production Editor**
Paul Spitzer Ellie Fountain

Editorial Board **Composer**
Steve Anglin, Dan Appleman, Ewan Buckingham,
Gary Cornell, Jason Gilmore, Jonathan Hassell,
James Huddleston, Chris Mills, Matthew Moodie,
Dominic Shakeshaft, Jim Sumser, Matt Wade

Proofreader
Dan Shaw

Associate Publisher **Indexer**
Grace Wong Lucie Haskins

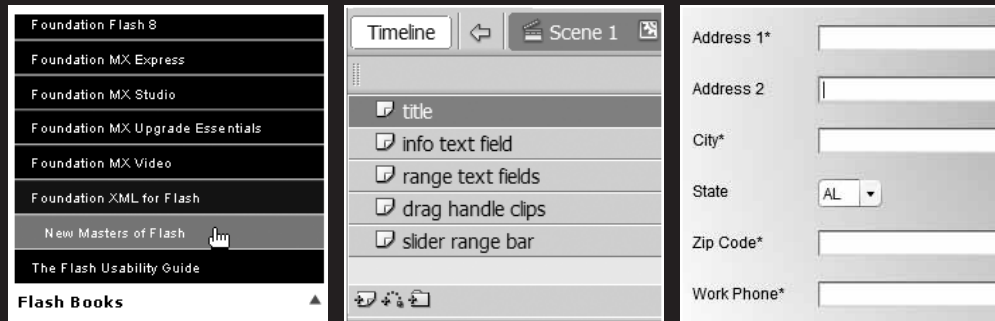
Project Manager **Artist**
Beth Christmas April Milne

Copy Edit Manager **Cover Designer**
Nicole LeClerc Kurt Krames

Copy Editors **Manufacturing Director**
Nicole LeClerc and Marilyn Smith Tom Debolski



6 INVENTORY VIEWS AND SELECTION DEVICES



In the previous chapter, you saw how to design and construct informative loaders to bridge the gap in time between a user's requests and the machine's downloading of information. Of course, once that information has loaded, the loader clips have faded away, and the dust has settled, then the application truly begins. So, what techniques can we use now to allow users to best view and interact with this newly downloaded data?

We could spend thousands of pages trying to discuss every possible kind of tool involving data displays you could feasibly build in Flash. With each one, we could have a slightly different answer to our posed question. So, instead of trying to cover a lot of different applications too abstractly, we've decided to focus on data display and interaction using one fairly common web application as an example: *the online store*. We'll do this by examining how the classic "inventory and shopping cart" scenario is implemented via HTML and find ways of improving this metaphor using Flash.

In this and the next chapter, we're going to replace the old-fashioned methods of how users interact with store inventories in HTML by employing some usability techniques that can be uniquely achieved in the Flash environment. In this chapter, we'll cover how to set up an inventory display grid for a set of store items (using the shopping cart metaphor as an example), as well as using drag-and-drop functionality to achieve multiple tasks with our items. In Chapter 7, we will expand on the inventory display grid from this chapter by covering innovative and far more seamless ways to allow users to filter and refine their choices before adding items to their cart.

As you read through these discussions, keep in mind that a vast number of different Flash applications could profit from employing inventory display and selection techniques. Although we are discussing an implementation of an online store, consider how these techniques (and even more generally, the thought process behind these techniques) may apply to your own Flash applications. Usability is as much art as it is science. Our intention is to let the discussion and examples here spark ideas for your own Flash-enhanced projects.

Let's roll up our sleeves and start thinking about usable inventory view and selection devices. Here's what we'll cover in this chapter:

- Metaphor-based design and the shopping cart metaphor
- Limitations of HTML-based inventory display and selection devices
- How to design drag-and-drop functionality in Flash to improve on the shopping cart metaphor
- How to build a reusable, elegant inventory display grid in Flash
- The architecture of drag-and-drop functionality using class interfaces

A brief interlude into metaphor-based design

Before we get into the guts of a usable store inventory and selection implementation, we would like to take a brief interlude and talk about *metaphor-based design*. This is one of

the more notable concepts in usability engineering, and since there's some good overlap between this concept and our chapter example, we think it's appropriate to give it some airtime.

We'll define metaphor-based design as nothing more than *creating an interface that mimics a physical-world mechanism*. The inventory view and shopping cart scenario is one of many real-world metaphors that engineers have assimilated to the Web. By looking at the usability benefits (and consequences) of metaphor-based design, we can better draw conclusions as to how to best implement the online store inventory view and selection process we'll tackle in the rest of this chapter.

Technology is laden with interactive metaphors adopted from the physical world. From the concept of desktops and files to e-mail, metaphors help us more easily learn how to interact with sometimes initially unintuitive concepts on the Web. They help us adapt to new environments by representing them in a more familiar physical form. The essence of a really good metaphor implementation is one that does the following:

- Helps users quickly grasp how to accomplish a task
- Hides the underlying functional implementations from the user
- Doesn't get too grounded in the physical-world example as to hinder the advantages inherent to the Web

6

It's easy to see why a metaphor can help users grasp new concepts on the Web faster. By and large, nearly every online store during the dot-com era incorporated the concept of a shopping cart because it immediately made sense to most people. They were comforted by understanding how to shop and knowing that they could click a link to add an item to a cart without having to purchase that item immediately.

This comfort blends into the second real benefit of metaphor-based design. The shopping cart scenario also means that potential customers don't have to concern themselves with how the actual functionality behind the interface works. A "shopping cart" is nothing more than a list of selected quantities of items that are stored either in a client-side cache or in some intermediary state on the server (that is, the use of application session variables or database tables). How this data is actually stored is both unhelpful and unnecessary information to be giving the user. By masking functional implementations through metaphorical interfaces, we keep an application's complexity away from the user, and instead, make it intuitive (one of our basic tenets of usability!).

However, it's also very easy to stick to a metaphor too closely. A clear example of this is brought up in *About Face 2.0: The Essentials of Interaction Design* by Alan Cooper and Robert Reimann (Wiley). We'll do our best to paraphrase their example here.

We're all familiar with the metaphorical "paper-based" calendar accessible from most operating system desk trays. Paper-based calendars are organized in pages by month. In the physical world, breaking down the calendar pages into months makes sense, as it makes pages a reasonable size and the calendar, as a whole, relatively portable to carry around in briefcases, pockets, and so on. Most computer-based calendars act the very same way. The benefit is that users instantly understand how to read and interact with

such an implementation. However, the traditional month-segmented calendar limits us from seamlessly tying a week together that is separated by two months; for example, using the calendar shown in Figure 6-1 to schedule something from August 28 to September 4. A better, more usable desktop calendar could simply implement a continuous scrollbar instead of adhering to the real-world concept of monthly pages.

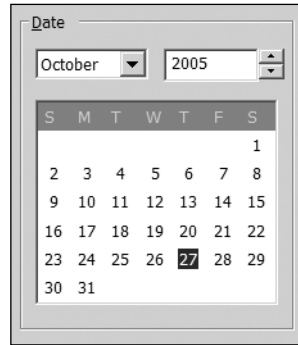


Figure 6-1. A desk tray calendar that follows the physical-world paper calendar too closely. You're still forced to change the settings to an entirely new month to see September 1, rather than being able to continuously scroll through each week.

In summary, a successful metaphor-based design helps the user more easily intuit an interface without hindering the ways in which technology can improve its physical-world counterpart. With these parameters in mind, let's now move to the meat of our chapter discussion and see how HTML-based shopping carts fare as a metaphor.

Understanding the HTML shopping cart metaphor

In the real world, items in a shopping cart can be checked out at the cashier or put back on the shelves. Similarly, items in an online shopping cart can be purchased or removed. Figure 6-2 shows a typical online shopping cart implementation.



Figure 6-2. Amazon.com's right-margin shopping cart view gives you a quick look at their cart without having to go to a separate page, but you still need to click through to another page for detail views or to remove an item.

INVENTORY VIEWS AND SELECTION DEVICES

However, one of the major disadvantages to the HTML shopping cart scenario is the stop-and-go nature of browsing through inventory items and adding them to a cart. Here is a typical workflow when adding an item to a cart:

1. Browse through a grid of items on a web page, as shown in Figure 6-3.



Figure 6-3. Red Envelope employs the classic inventory grid layout. You must click the link underneath each image to go to a detail view and to add the item to a shopping cart.

2. Select a particular item to view.
3. Be redirected to some item detail page.
4. To add an item to the cart, click another button, as shown in Figure 6-4, and wait for the server to process the transaction.



Figure 6-4. Most online stores require you to click a button to add an item to a shopping cart. This also requires a round-trip from the client to the server and a browser refresh.

5. Be redirected to a page displaying a full list of items that are in the metaphorical shopping cart.
6. Be presented with further options to search for more items, and if you're lucky, a link to redirect you to the original list of items you were browsing through.

The stop-and-go feel of adding items to a cart in HTML can be frustrating and incredibly inefficient. Shoppers are constantly bombarded by page redirects and server refreshes at every corner. The workflow and constant round-tripping from client to server makes the metaphor of placing and removing items from a shopping cart a weak one at best. It doesn't have the fluid feel of a real shopping scenario.

What you've just seen is how many metaphors tend to degrade on the Web. We get some hints of the metaphor, but only to the extent that they remain feasible to implement. One of the sore spots of HTML design is that it tends to be too rigid a model to really take advantage of all the positive characteristics of the physical-world metaphor. We rely on the user to conform to the inflexibility of HTML architecture, rather than find ways to better implement the metaphor.

Exacerbating the problem with the HTML shopping cart workflow is the fact that this constant round-tripping usually has little "bang for the buck." For instance, when a user adds an item to a shopping cart, we typically store the ID of the item along with the ID of the user in some database table. This involves a call to another page, a database connection, a database submission, and a re-rendering of the entire HTML page to display the update. Yet, all that has been passed down to the server is a few bytes' worth of data (the item ID and perhaps some other small bits of information, depending on the particular application structure).

A better-performing solution would be to store these cart items on the client's machine until users are ready to purchase their items. While you could store data on the client side using cookies in HTML, you don't always see this done because cookies are browser-dependent and could potentially be rejected. As HTML developers, we're more willing to risk tarnishing the user experience in exchange for the assuredness of storing cart items securely on the server side.

In Flash, we could easily store this light piece of data on the client side as an ActionScript array (or for the more ambitious, in some hand-built object), and not need to worry about browser dependencies of the client-side cookie. Only when the user is ready to purchase an item do we submit everything to the server, in one bulk load.

Note that we've omitted the actual storing and submission of items in the shopping cart from this chapter's solution to focus on the other aspects of usability in this chapter.

In Flash, we don't need to conform (at least not nearly as much) to a few development options. It allows us, instead, to flex our creative muscles and implement innovative interactions that better fit a real-world metaphor. The great thing about Flash is that we can better mold a real-world concept into the online world. We don't need to mask metaphors behind rigid HTML components and settle for just giving metaphorical terms to otherwise old-hat HTML design.

Now, let's try to improve the real-world shopping scenario in Flash and see how we can provide a much better-fitting design.

Devising a better shopping cart solution in Flash

To circumvent the problem points with the HTML design, our general usability goal should be to make the entire interface more seamless, like a real shopping experience would be. Instead of the stop-and-start nature caused by constant server round-trips we're accustomed to in HTML, let's find a way to build on the interactive metaphor of putting items into a cart that is fluid and continuous.

One easy way to achieve this goal is to implement *drag-and-drop functionality* with the inventory items. Going back to Figure 6-3, suppose that when a customer drags her mouse off an item, she were able to then "carry" the item around to different areas of the application interface. If she then dropped the item over some shopping cart panel, it would immediately be added to her cart. Drag-and-drop plays off the metaphor of picking up an item and placing it into a real shopping cart.

Remember that one key to good metaphor-based implementations is to not constrain yourself to the metaphor too closely if it means limiting the ease of use of your application. So, one way we could extend the metaphor is to allow the customer to, say, drag a copy of the item over a detail panel. When a customer releases the item, specific product information could be displayed. Although there isn't a strong parallel physical-world concept of placing an item over some apparatus that will then display detailed product information about the item (you might consider those price-checking bar code scanners as a rudimentary form of this), it is something we can implement in the online world to add to the usability of our application.

To be smart, however, we shouldn't just limit user interactions to solely drag-and-drop actions. For instance, what would happen if you just clicked an item instead? Traditional HTML web applications would probably send you to a new page showing a detailed description and perhaps an enlarged photo of the item. In a usability-enhanced Flash application, we can just as easily make the clicking of an item invoke the same response as dragging the item over the detail panel.

Let's take these general ideas and see how they could work in a real application. To demonstrate, we've built a book store inventory catalog with items that can be seamlessly dragged into a different portion of the application, depending on what the user wants to do.

Download the `Chapter6_Final.zip` file from this book's download page at www.friendsofed.com and export the files within the ZIP file into a directory on your machine. Then open the `Chapter6_Final.swf` file in the `/source/swf/` folder.

FLASH APPLICATION DESIGN SOLUTIONS: THE FLASH USABILITY HANDBOOK

You should see the shopping catalog application, as shown in Figure 6-5. Because the concept of drag-and-drop in a web application may not be obvious to new users, we've included a set of simple instructions at the top of the page to get users started on the right path.

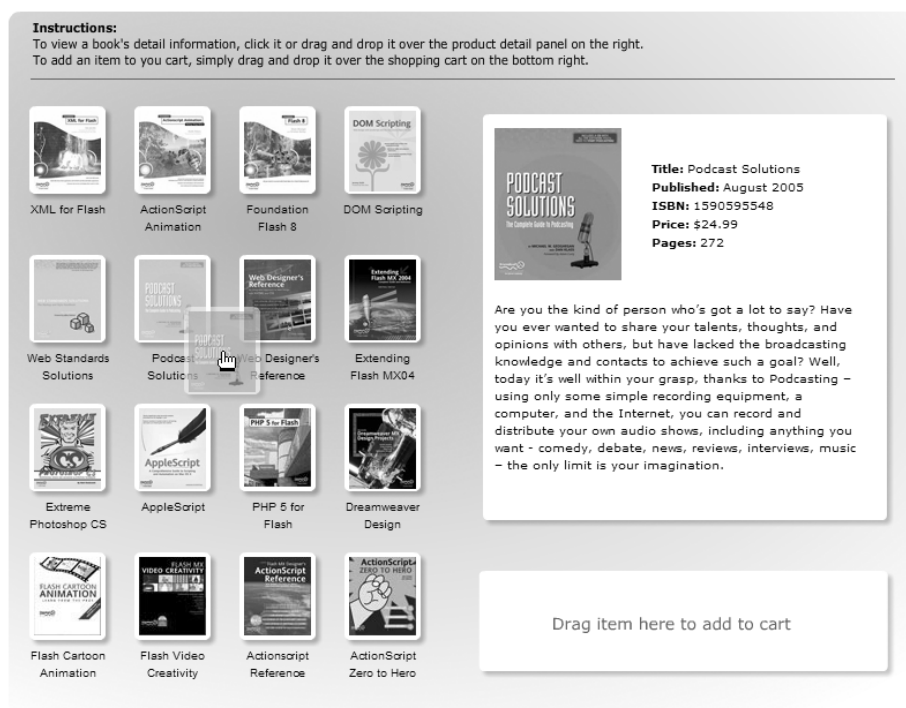


Figure 6-5. A usability-enhanced shopping catalog with draggable inventory items, including a detail panel and shopping cart

Use your mouse cursor to drag over a book thumbnail item to other parts of the application. Notice that when you drag the mouse off the item, a semitransparent copy of the item attaches to your mouse arrow. Now, by dragging it over the detail panel, you should see the panel highlight itself. This makes it clear that you can drop the item copy here to get more information. The same concept applies to the shopping cart clip in the lower right. Drag-and-drop a book thumbnail over the cart, and you should see the cart accept the item.

Also, try releasing your book thumbnail while it's not over either the detail or cart panel. You'll see that the item snaps back to its original position and fades out of view. This makes it clear to the user that no changes have taken place. You'll find that such attention to *little details* can make a *huge usability difference!*

You might be wondering whether or not drag-and-drop is actually a usability enhancement for viewing and selecting items in an inventory. After all, drag-and-drop interactions, while a staple on desktops (such as moving files from one folder to another), are not common on the Web. Some may argue that introducing this new kind of metaphor will confuse, and possibly deter, customers from using such an application.

While that's certainly a possibility, remember that our definition of usability also involves an ingredient of innovation. As long as the benefits of innovative design outweigh the drawbacks of ramping up a user's understanding of the application, we feel it is well worth it. Stepping outside the traditional bounds of web design is critical to enhancing usability.

Of course, another critical key is your user base. More advanced Web users will probably welcome and benefit from the change more readily than users who are comfortable with old staples. You may want to have a group of people representative of the kinds of users who would interact with the real application test a demo to determine whether it's truly the right design decision.

Notice the improvement on the usual HTML solution. The multistep, one-way process of clicking an item to view its detail, and then clicking a button to add it to a cart, and then finding your way back to the inventory view is replaced by a user-controlled workflow. If you want to learn more about an item, you just click it or drag it over the detail panel. If you know what you want, you can just drop it over the shopping cart icon. The user now has more leverage in deciding how to interact with the store inventory, leaving behind the rigid workflow we see in HTML-based inventory selections.

Now that you've seen the final solution, it's time to turn our attention to how to build it. As you probably have noticed in this solution, we haven't fully developed the shopping cart. Our intention in this chapter is to keep the focus on how to build drag-and-drop functionality so that any object within your application can respond to something being dropped over it. For now, the shopping cart responds with a simple "Item added to cart!" message. In a public-ready solution, you would obviously want to display the cart items in an elegant way.

In Part 3, we'll show you a fully functioning online book store solution that uses the same metaphor in this chapter, but with a fully developed shopping cart. It will also allow a user to add multiple copies of a book to the cart (or any other customization for that matter). So, if you wanted to purchase three *ActionScript: Zero to Hero* books, you could drag three copies into the cart, or you could drag a book item into the detail panel, and then key in a 3 in a Quantities field before clicking an Add to cart button.

Building the Flash solution

If you carefully examine the kinds of interactivity going on in the solution, you should notice a lot of subtle interactions occurring. Building the application involves the following:

- Extending the selection system classes we discussed in Chapter 3 to create an inventory grid (similar to Figure 6-3)
- Adding the drag-and-drop functionality with the book thumbnails
- Creating a methodical way to add in new “droppable” areas (such as the detail panel and shopping cart)

We’ll employ an object-oriented methodology by using a construct called an interface to build the droppable areas. But, before we get to the code, let’s examine the various Flash assets we’ll need to get this application off the ground.

Creating the Flash UI assets

Open `Chapter6_Final.fla` from the `/source/fla/` folder extracted from the ZIP file for this chapter’s example. You’ll notice that our root timeline is fairly simple. We have a background layer, which is a subtle gray-to-white gradient fill, intended to give the entire application a nice textured feel. In the layer above this, labeled instructions, we provide a quick set of instructions regarding the drag-and-drop functionality. On the top layer, we include an instance of a movie clip whose symbol name is `MC Interface`. This houses pretty much all of the components of this application. Open this clip so you can see how the interface is composed.

Let’s see what elements reside in each layer of our store catalog application interface.

The cart layer

On the bottom layer, labeled `cart`, resides our shopping cart movie clip. This clip will have a symbol name of `MC Cart` and an instance name of `cart_mc`. It’s also linked to the AS2 class file `Cart.as`, which we’ll discuss later. If you double-click into this clip, you will notice three layers, as shown in Figure 6-6.

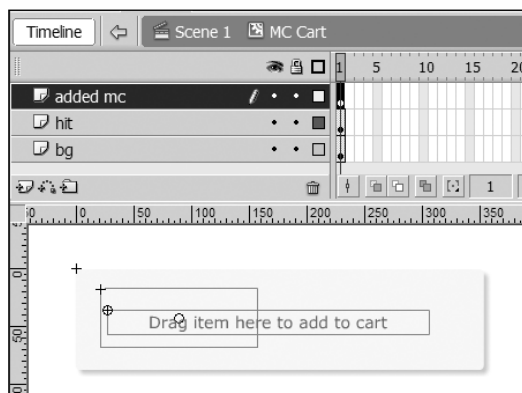


Figure 6-6.
The timeline for the basic cart movie clip

The bottom layer, `bg`, holds a background image for the cart with a white background to contrast with the darker main background image. We employ a drop shadow filter here (Chapter 4 discusses using filters in Flash 8) to make it stand out from the main background even more.

The next layer, `hit`, contains a rectangular movie clip with an instance name of `hit_mc`. The clip is centered on top of the background and is about 80% of the background's area. This clip will serve as the active hit area for the cart. In other words, only when an item has made contact with any portion of `hit_mc` will the cart "accept" the item if dropped. The `_alpha` property of the clip has been set to 0 to make it invisible to the user.

You might be wondering why we've decided to make this hit area somewhat smaller than the actual size of the background of the cart. Quite simply, a customer who really wants to add an item to his cart will undoubtedly drag the item toward the central portion of the cart. In order to have the action of adding an item actually activate, a significant portion of the item must be over the icon. This prevents accidental additions to the cart when, say, a user has decided to not select an item but doesn't realize that a corner of that item has hit the cart area. This forgiving nature is a standard quality of good usable interaction design.

Finally, on the top layer, we add an MC Added Item clip (instance name of `added_mc`) that, for the purposes of this solution, will display a status message to the user. When an item has been added to the cart, the "Item added to cart" message will display. Otherwise, "Drag item here to add to cart" will display. Again, while we haven't implemented it here, in our fully functioning book store application in Part 3, we'll make the cart actually store the item.

The product detail layer

Let's now move back to the MC Interface timeline. Above the shopping cart layer, we have the product detail layer. This layer holds an instance of the MC ProductDetail clip called `productDetail_mc`. MC ProductDetail provides the placeholders for the product information data we'll pass to it when an item is dropped over its surface. Its associated AS2 class, `ProductDetail.as`, defines the code that presents the selected book's information. Double-click the MC Product Detail clip to see what's inside, as shown in Figure 6-7.

First, you'll notice the bottom layer of our product detail clip, named `bg`, contains a background image that is light gray with a drop shadow filter underneath it. This helps to better distinguish the panel from the background of the application, even when no product information is being displayed.

On the layer above this is the hit layer, for our defined hit area. This is the actual area that will trigger the drag-and-drop sequence. As we did for the cart, the `hit_mc` movie clip is centered on top of the background and is slightly smaller than the background area. And, just as in the cart layer, we've set this clip to 0% `_alpha` so that it's invisible to the user.

Above this is a layer called `image`, which holds an empty movie clip called `imageHolder_mc`. This clip will hold the larger-sized image that corresponds to the selected thumbnail item. We've placed it toward the upper-left area of our product detail panel.

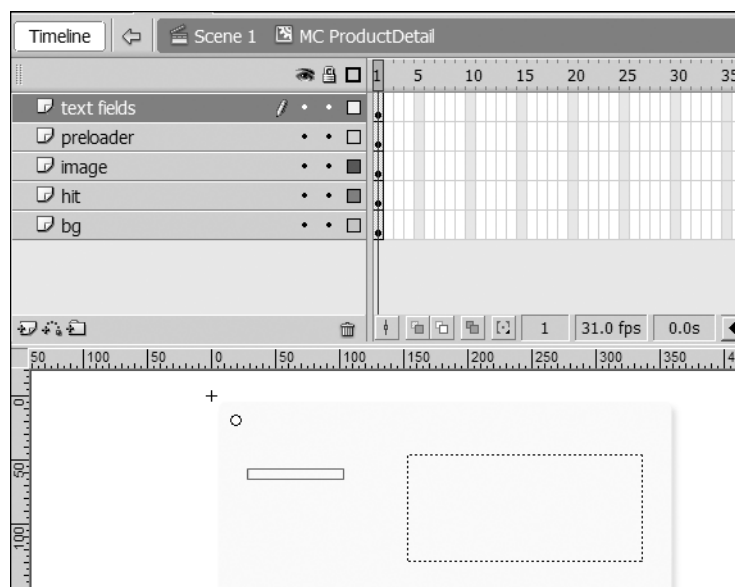


Figure 6-7. The timeline for the product detail clip

In the preloader layer above the image layer, we have an instance of a MC MovieClipLoaderUI clip called `preLoader_mc`. This loader will be in charge of displaying load-status information as the image of the product is loading. MC MovieClipLoaderUI is, in fact, the very same loader clip we built in Chapter 5!

Above the preloader layer, we have a text fields layer that holds text fields for the various pieces of textual information associated with each book. In this case, we'll leave it fairly simple. The `description_txt` field will post a brief paragraph about the book selected, and the `info_txt` field will display the title, number of pages, ISBN number, and price for each book.

The product grid layer

Back on the MC Interface timeline, in the product grid layer, is an empty movie clip called MC ProductGrid. This empty clip will be a container for all of the book item thumbnails. We've labeled the instance of this clip `grid_mc` in the Properties panel. Also, we've linked this movie clip to an AS2 class we'll construct called `ProductGrid`. `ProductGrid.as` will be in charge of structuring the book thumbnail items into the classic inventory grid layout.

Where else have you seen an empty movie clip used to hold a collection of items? That's right. Once again, we will be taking advantage of the work we did in Chapter 3, extending the `SelectionSystem` class to create `ProductGrid`!

You'll notice a script layer on top of the main MC Interface layer set. This layer contains a bit of ActionScript code that will initialize the product grid and the resulting drag-and-drop functionality. We'll get to this at the very end of our solution discussion after you've seen how to code these clips' underlying functionality.

The thumbnail item clip

Of course, without having an actual clip that represents an inventory item, this chapter wouldn't go very far! So, the last piece of the puzzle, as far as our UI goes, is to create a thumbnail item clip along with a clip that will appear as a copy when a user drags the item to another location. We've separated these two into different clips, rather than just reusing the same one for the copy because some elements of the base thumbnail aren't really needed. From a usability point of view, we should show just enough information about the original inventory item so users see what they are dragging. Let's explain.

As shown in Figure 6-8, the draggable item doesn't retain the title display below it. By *not* including the title underneath the draggable item, the bounding area (the area that must pass over the cart's hit_mc clip to activate it) is simply a clearly defined rectangle. Had we included the title, the invisible rectangular area surrounding the title text field would also become part of the bounding area, making it a bit more confusing for a user to handle.



Figure 6-8.

When dragging a book item to another destination, we include only the image of the book itself rather than the text. Including the text is just unnecessary overhead from a visual standpoint. Additionally, the draggable item has an intuitive rectangular bounding area for our hit tests when we drag over the shopping cart or product detail panel.

Double-click MC ThumbItemButton from the Library panel to see its composition. Starting at the very bottom layer, bg, we have a background graphic that has rounded corners and a drop shadow filter to give it a nice, raised effect when we place it against the background of MC Interface. Above this layer is the thumbholder layer, which contains an empty movie clip into which our book thumbnail image will load. We've given this an instance name of thumbHolder_mc.

Above the thumbholder layer, you'll see yet another instance of MC MovieClipLoaderUI called preloader_mc, which will be in charge of monitoring the loading of the book thumbnail image to this clip. While our book thumbnails will be quite small in size, don't forget that many items will be loading asynchronously. Even if it's just a split second, it's comforting to see a loader bar appear instead of just a blank item; it lets users know that the application is working.

Finally, the title layer contains an instance of the MC Title clip, labeled as title_mc. This clip simply contains a dynamic text field called title_txt.

Notice that we have linked this clip to a class named `ThumbItemButton`. In conjunction with the `ProductGrid`, it will form a selection system with the added drag-enabling functionality we'll work on in the code base for this chapter.

The draggable thumbnail item clip

Now, take a look at `MC ThumbDragItem` by double-clicking it from the Library panel. The guts of this clip are very similar to what you saw for the `MC ThumbItemButton`. The only visual difference here is that we've removed the title text layer, as we noted earlier. Also, note that this clip is linked to a class named `ThumbDragItem`.

Coding the solution

Now that we have taken a thorough look at our various assets, let's get started with the code development. We'll forewarn you that this is one of the more complex programmatic solutions you'll see in this book! It involves a bunch of separate pieces that work together to create the final solution. Rather than going through every line of code from every class we create (as we do for some of our simpler implementations), we'll elaborate on only the more integral pieces of code that drive this application.

We will go back and forth between classes a few times so you get a better understanding of where and how these classes integrate with one another. Of course, you have the complete source code available from the ZIP file for this chapter, so if you're one of those types that learns better through brute-force code surveying, feel free to simply scan through our solution and use our discussion here as an aid. To keep you steered on the right path in our code discussion, here's a road map of our basic approach:

- We'll set up the `ProductGrid` class (based on the `SelectionSystem` base class from Chapter 3) and show you how to implement a well-structured grid layout for our various book item thumbnails.
- We'll outline the `ThumbItemButton` class (based on the `UIButton` base class from Chapter 3) and explain how it defines where to position each instance of `MC ThumbItemButton`. You'll see how this class works with the `ProductGrid` class to create the book thumbnail dragging effect.
- We'll set up a `DragDropManager` class, whose job will be to monitor when a user drags or drops a book thumbnail item over an area of the interface. We'll also discuss the `ThumbDragItem` class, which works in tandem with `DragDropManager` to sense when the dragged item lands over another part of the application. `DragDropManager` will then be responsible for telling the product detail panel and the cart that a user has dragged or dropped something over their surface, so that they know to react in their own predefined manner.
- We'll create the classes for the product detail panel and shopping cart objects. For simplicity's sake, we'll refer to these objects as *droppable UI objects* in our code discussion. We'll show you how to use an object-oriented concept called *interfaces* to better structure your code for these classes.
- We'll put it all together by creating an instance of `DragDropManager`, registering our droppable UI objects with it, and then passing a book data array to the `ProductGrid` class to create the inventory view.

Creating the grid layout structure

If you recall from Chapter 3, we built two extendable classes (`SelectionSystem` and `UIButton`) that form a basic, reusable selection system. Whether it's a set of navigation links or a set of store catalog items, as in this case, we can use our selection system base classes to handle the initial load, attachment, and positioning of items into a container clip.

We will override two key methods here in the extension of these two classes. Recall that the `attachButtonItems()` method within the `SelectionSystem` is in charge of actually attaching the `UIButton` elements to a container clip and then passing whatever information the `UIButton` needs to function. We will modify this method to pass it an additional set of information that it needs to position itself in the grid layout. Secondly, we'll overwrite the `setPosition()` method within the `UIButton` class to take this new information and convert it to the exact (x,y) pixel location of where it should position itself on stage. After doing these two modifications, we'll have the code ready to structure our items in the inventory grid layout you see in Figure 6-5. So let's get started!

MC `ProductGrid` is the clip that acts as a container for all of our book thumbnail items. Since we've already built the methods to handle the load of individual items within the `SelectionSystem` class, we'll need to extend this class through MC `ProductGrid`'s linked class `ProductGrid`. Our focus in this discussion will be on how to extend and customize this base class to set up our product grid layout.

What's unique about the MC `ProductGrid` clip (compared to other ways we've extended `SelectionSystem`) is the way the items within the clip will be positioned. Because it will be a grid layout, rather than a straight left-to-right or top-to-bottom layout, we'll need to do a little more work to determine how the book thumbnail items will position themselves. In `ProductGrid.as`, inside the `/source/classes/` folder of the chapter download, we add a few private variables.

```
private var itemsPerRow:Number = 4;

private var currentRow:Number = 0;
private var currentColumn:Number = 0;

//space between rows and columns
private var horizSpace:Number = 16;
private var vertSpace:Number = 6;
```

We have an `itemsPerRow` variable to determine the number of rows in our book grid layout. As you can see in Figure 6-5, our grid is 4 × 4. So, we'll set this value to 4. Because the `SelectionSystem` class will load our book thumbnail items one at a time, we'll also need to keep track of the current row and column assigned to each book thumbnail item, namely `currentRow` and `currentColumn`. Think of these values together as the table cell within the 4 × 4 grid. Finally, we'll want to adjust how much horizontal and vertical space should fit between each column and row. The `horizSpace` and `vertSpace` variables will store this data. In this case, we'll set the horizontal padding between items to 16 pixels and the vertical padding to 6 pixels.

Now, recall from Chapter 3 that the `SelectionSystem` class uses an `attachButtonItems()` method (called through its `doInit()` method) to loop through a passed-in array of data to

populate its associated parent movie clip with items. The `attachButtonItems()` method not only attached each new item clip to the stage, but also called an `init()` method defined within the `UIButton` class, which provided a generic object with all the essentials it needed to function. In our specific case, we'll need to pass into this `init()` method some *additional* way of letting the eventual `ThumbItemButton` class know where to place itself within this MC `ProductGrid` container clip. To accomplish this, we'll modify the `init()` method in the `ThumbItemButton` class to include a fourth parameter. We'll override the `attachButtonItems()` method as follows:

```
public function attachButtonItems():Void
{
    for(var i:Number = 0; i < systemData.length; i++)
    {
        //get placement object according to counts
        updateGrid(i);

        //attach a clip
        var item:MovieClip = attachMovie("MC ThumbItem", "Item_" + i, i,
        {id:i});
        item.init(this, i, systemData[i], getItemPositionObj());
        listItems.push(item);
    }

    //by default, select the first UIButton item and display its detail
    //in the product detail panel using initDetail()
    setSelection(0);
    initDetail();
}
```

Notice a couple methods called in this class (emphasized in the preceding code) that we haven't yet defined. First, the `updateGrid()` method will be in charge of updating the `currentRow` and `currentColumn` variables as each new book thumbnail is attached to the grid. Notice that when we pass in the current index of the item to this method, it will use the mod function (%) to figure out if the currently loaded item needs to move to a new row. If `index % itemsPerRow` evaluates to 0 (that is, the index divides into the number of rows evenly), we know that the placement of that particular item must be in a new row. Here is the `updateGrid()` method:

```
private function updateGrid(_index:Number):Void
{
    if(!(_index % itemsPerRow))
    {
        currentColumn = 0;
        currentRow++;
    }
    else
    {
        currentColumn++;
    }
}
```

The other new method is `getItemPositionObj()`. This method returns the essential positioning information for each of the thumbnail items, which is assigned to the new fourth parameter of `ThumbItemButton`'s `init()` method. This information will come in the form of a generic `Object` instance populated with grid position information. Here is the `getItemPositionObj()` method:

```
private function getItemPositionObj():Object
{
    var obj:Object = new Object();

    obj.horizSpace = horizSpace;
    obj.vertSpace = vertSpace;

    obj.grid_x = currentColumn;
    obj.grid_y = currentRow;

    return obj;
}
```

Notice that the returned object in this method contains the horizontal and vertical padding between each item in our grid, along with the row and column number for the clip. This is enough information to calculate the exact `_x` and `_y` positions for the book thumbnail item. In our discussion of the `ThumbItemButton` class, we'll show you how a thumbnail item will use this information to position itself accordingly on stage.

These methods complete the `ProductGrid` class to create the easily recognizable grid interface for our book item inventory. We'll dig back into this class later on when we delve into our dragging functionality discussion, so you may want to leave this file open if you're following along using the chapter solution.

But, before we get to dragging, let's round out our implementation of the grid layout by implementing the necessary functionality in our `ThumbItemButton` class.

Setting the position of the thumbnails

Just as we've extended the selection system class to take advantage of its loading methods for `ProductGrid`, we'll also create the class linked to our `MC ThumbnailButton` clip to extend `UIButton` (our generic button item class from Chapter 3). Open `ThumbItemButton.as` inside the `/source/classes/` folder of the chapter download and let's go through the particular way we extend this class to fully implement our grid layout.

Remember that our `ProductGrid` class will call this class's `init()` method. Here, you see the inclusion of the fourth parameter `_gridObj`, our generic object passed from `ProductGrid`'s `getItemPositionObj()` method:

```
public function init(_selectionSystem:SelectionSystem, _id:Number,
    ➤ _itemData:Object, _gridObj:Object):Void
{
    gridObj = _gridObj;
    super.init(_selectionSystem, _id, _itemData);
    loadThumb();
}
```

We won't do the positioning work just yet. Instead, we'll simply set the passed-in `_gridObj` to a private property `gridObj` (while not shown, in our class code, we added `gridObj` as a private property we define in the class).

Recall from Chapter 3 that our clip's stage position is set by invoking the `setPosition()` method within the parent `UIButton` class. Once again, we'll override the method in the `ThumbItemButton` class to implement the positioning specific to this particular solution. Here's how we've coded the `setPosition()` method:

```
private function setPosition():Void
{
    _x = Math.round((gridObj.grid_x * _width) +
        ➤ (gridObj.horizSpace * gridObj.grid_x));
    _y = Math.round(((gridObj.grid_y-1) * _height) +
        ➤ (gridObj.vertSpace * (gridObj.grid_y-1)));
}
```

Notice that the `setPosition()` method takes the `grid_x` and `grid_y` values from `gridObj` and calculates the true `_x` and `_y` position for the book thumbnail. To calculate the appropriate `_x` position, we need to take the total width of one book thumbnail item (its `_width` + `gridObj.horizSpace` buffer width) and multiply it by its column position within the grid (`grid_x`). We can calculate the appropriate `_y` position in a similar fashion.

In a nutshell, that's the essential code needed within our grid and book thumbnail item clip classes to generate our simple but well-designed grid layout. This demonstrates where *reusability* of code helps us out. We've extended both the `SelectionSystem` and `UIButton` classes from Chapter 3 to handle the loading and selection states of our store catalog collection, and we just needed to override the `SelectionSystem.attachButtonItems()`, `UIButton.init()`, and `UIButton.setPosition()` methods to create this new grid layout format.

Now, let's move on to perhaps the more visually interesting portion of the code: our usable drag-and-drop functionality.

Creating the drag-and-drop functionality

You learned from our usability discussion that what's a little different about our drag-and-drop functionality is that you don't simply drag-and-drop a book thumbnail item; rather, you drag a *copy* of the item (the `ThumbDragItem` clip). What this implies is that we can't simply use the built-in `startDrag()` and `stopDrag()` functions on each book thumbnail to accomplish the task, because they apply to dragging the original item.

Take a look at `ThumbItemButton.as` once again. We've overwritten the `handlePress()` and `handleRelease()` methods from the extended base `UIButton` class. Recall that these methods are invoked when a user presses the item and releases the item, respectively. The `handlePress()` method will call a new method in our button class, `startDragTest()`. The `handleRelease()` method will call a new method called `stopDragTest()`. Let's take a look at these two new methods now.

```

private function startDragTest():Void
{
  this.onMouseMove = function()
  {
    if(!this.bg_mc.hitTest(_root._xmouse, _root._ymouse, false))
    {
      stopDragTest();
      SelectionSystem.cueDragDropManager(id);
    }
  }
}

private function stopDragTest():Void
{
  this.onMouseMove = undefined;
}

```

It may seem a bit confusing what we're doing here, but it's actually quite simple. When a user has pressed the mouse button on a particular book item, we set up an `onMouseMove()` event handler that waits to see if the mouse cursor rolls *off* the book thumbnail item. From a usability perspective, whenever this happens, it's a sure sign that the user wants to drag the item to some other destination. If so, we kill the event handler through the `stopDragTest()` method and tell the `ProductGrid` class to help create the dragging effect through a method called `cueDragDropManager()`.

Let's move back to `ProductGrid.as` and take a look at our rather thin `cueDragDropManager()` method:

```

public function cueDragDropManager(_id:Number):Void
{
  _parent.dragDropManager.attachDragItem(systemData[_id]);
}

```

Now, if you aren't confused, you should be! One thing we haven't discussed yet is how we set up not only the dragging effect, but how other objects in our application interact with an item that is dragged or dropped over their respective surfaces. In essence, we need some sort of class that observes the entire application, knowing when to start a drag of a thumbnail item and signaling to other objects in the application when a user drags or drops the item over their surfaces. In this solution, we've built a class called `DragDropManager` to do this work. The `_parent.dragDropManager` is a reference to an instance of this class that will be created on the timeline of the MC Interface clip (our master clip for the whole inventory application).

We use `_parent` here because we know beforehand that we're going to create this `DragDropManager` class instance on the same parent timeline as the `ProductGrid` instance. This isn't the best way to make our code reusable, as it now places a dependency on where we put the `DragDropManager` instance if we want to reuse this code. But we do this here to simplify the code for this solution.

So, now we know that when a user drags off a particular book item, we invoke some method called `attachDragItem()` from the instance of `DragDropManager`, passing with it the book thumbnail's item data (`systemData[id]`). This class's `attachDragItem()` method then creates that thumbnail item copy you saw in Figure 6-8.

To examine the `DragDropManager` class, open the `DragDropManager.as` file inside the `/source/classes/` folder of the chapter download. First, let's quickly go through a few properties this class will need in particular for the `attachDragItem()` method to work:

```
private var targetClip:MovieClip;
private var linkage:String;
private var dragEnabled:Boolean;
private var dropEnabled:Boolean;
private var dropAreas:Array;
private var hitClip:DropArea;
```

We'll need a target movie clip, `targetClip`, that tells us to which movie clip's timeline the draggable book item thumbnail will be attached. In this case, it will be the MC Interface clip instance itself. Also, we'll need the linkage name, `linkage`, of the movie clip we want to use as the draggable item. In this case, it will be the MC `ThumbDragItem` movie clip. Of course, we could easily just hard-code these names within the `DragDropManager` class but, always with an eye toward reuse, we'll abstract these as variables. The other variables will be discussed a little later in this section.

Later on, when we write the initialization code within the timeline of MC Interface, we'll pass this data into the `DragDropManager`'s constructor, which is shown here:

```
public function DragDropManager(_targetClip:MovieClip,_linkage:String)
{
    targetClip = _targetClip;
    linkage = _linkage;
    dropAreas = new Array();
}
```

Now, let's take a look at `DragDropManager`'s `attachDragItem()` method.

```
public function attachDragItem(_itemData:Object):Void
{
    targetClip.attachMovie(linkage, "dragItem_mc", 1000);
    targetClip.dragItem_mc.doInit(_itemData);

    Mouse.addListener(this);
    dragEnabled = true;
}
```

Here, we attach an instance of the linkage movie (in this case it will be the MC `ThumbDragItem` clip) onto the master interface clip for this application. Also, we call a `doInit()` function that's defined in `ThumbDragItem.as`, passing along the original book thumbnail item's data object which then loads the copy of the book thumbnail image to this MC `ThumbDragItem` instance.

The `DragDropManager` class also has the responsibility of telling our droppable UI clips (the product detail panel and cart panel) what to do when a user drags this newly attached draggable item. Going back to our set of private properties for this class, we have an array called `dropAreas`, which will store a list of these droppable UI clip paths.

Now, notice that in the `attachDragItem()` method, this class also registers itself as an event listener of the `Mouse` object. What this means is that any event handler actions that can be invoked through the `Mouse` object, such as `onMouseMove()` and `onMouseUp()`, will also call the methods of the same name in our `DragDropManager` class. In our code, the `onMouseMove()` method will invoke `testForHit()`, a method that cycles through each of the droppable UI clips within our `dropAreas` array and sees whether our book thumbnail item has made contact with any of them. Here's the implementation of the `onMouseMove()` and `testForHit()` methods:

```
public function onMouseMove():Void
{
    testForHit();
}

private function testForHit():Void
{
    for(var i=0; i<=dropAreas.length-1; i++)
    {
        var testClip:DropArea = dropAreas[i];

        if(testClip.getHitArea().hitTest(targetClip.dragItem_mc))
        {
            testClip.setEngageDrop(true);
            hitClip = testClip;
            dropEnabled = true;
            disableOtherDropAreas();
            break;
        }
        else
        {
            disableAllDropAreas();
        }
    }
}
```

The essential calls are highlighted in the preceding listing. While the mouse is dragging the `MC ThumbDragItem` clip around the application, this code will cycle through each object in the `dropAreas` array to see whether the dragged item has made contact with a droppable area (`if(testClip.getHitArea().hitTest(targetClip.dragItem_mc) { })`). We'll explain the `getHitArea()` method a bit later. For now, you should just know that it simply returns a movie clip that represents the active hit area for the droppable UI clip.

If the dragged item has made contact with a droppable area, with the call to `testClip.setEngageDrop(true)`, the droppable UI movie clip can display an effect to show the user that the thumb item can be dropped over its surface. In this application, the product detail panel slightly fades. We'll explain the `setEngageDrop()` method in more detail in our discussion of droppable UI areas in the next section. Also, we store the reference to the droppable UI movie clip in another private property called `hitClip`, so that when a user decides to drop the clip, our `DragDropManager` class will remember which clip should respond to the drop. The class's `dropEnabled` Boolean is set to `true`, indicating that the thumb item is ready to drop over one of the droppable UI areas. You'll see how we use this Boolean in just a moment.

Did you also notice that we've set the variable `testClip` to a `DropArea` type (not a `MovieClip` type as you might expect)? `DropArea` is an *interface* we'll implement next. If you're not yet familiar with interfaces, we'll explain them in detail in just a bit. For now, all you need to know is that `setEngageDrop()` (along with `getHitArea()`) will be defined in every movie clip within the `dropAreas` array, by virtue of the fact that it is a type of `DropArea`.

It's worth mentioning how `DropArea` objects are added to the `dropAreas` array. This is done with a simple public method within `DragDropManager` called `addDropArea()`, shown here. We'll add these drop areas to the array after creating an instance of `DragDropManager` on the MC Interface clip's root timeline, in the "Putting it all together" section later in this chapter.

```
public function addDropArea(_dropArea:DropArea):Void
{
    dropAreas.push(_dropArea);
}
```

Of course, now that we have a method that will sense when a user has *dragged* an item over a droppable UI area, we also will need one that will sense when a user has *dropped* an item over a droppable UI area. This is where `testForDrop()` comes in. Here's the `testForDrop()` method within the `DragDropManager` class:

```
private function testForDrop():Void
{
    if(dropEnabled)
    {
        hitClip.doDrop(this);
        targetClip.dragItem_mc.remove();
        dropEnabled = false;
        Mouse.removeListener(this);
    }
    else
    {
        removeDragItem();
    }
}
```

Again, the essential calls are highlighted in the preceded listing. First, with `hitClip.doDrop(this)`, the droppable UI movie clip can display an effect to show the user that the thumb item has been dropped over its surface (we'll discuss the `doDrop()` method later in the upcoming section about the droppable UI areas). Notice that this line is invoked only if `dropEnabled` is set to `true`. This was set in the `testForHit()` method. The `if/else` statement is necessary because if `dropEnabled` is not `true` (meaning that the current item being dragged is not actually over any droppable UI clip), we call the `removeDragItem()` method, which will then create the "snapback" effect of the draggable book item we discussed earlier in the chapter.

Although we won't go into detail about the implementation of the snapback effect, for posterity, here is the method that does the trick within the `ThumbDragItem` class. We first set the original `x`- and `y`-positions of the draggable book thumbnail item in the `doInit()` method that is loaded from when we attached the clip in the `DragDropManager` class. Here, we snap the clip back to this position using a few tween calls:

```
public function returnToOrigin():Void
{
    xTween = new Tween(this, "_x", Regular.easeOut, _x, origX,.2, true);
    yTween = new Tween(this, "_y", Regular.easeOut, _y, origY,.2, true);
    fadeTween = new Tween(this, "_alpha", Regular.easeOut, _alpha, 0, .3,
    ➤ true);
    fadeTween.addListener("onMotionFinished", this);
}
```

So, in a nutshell, we've now created the structure for how items are dragged and dropped. If you recall, we've saved two methods that are owned by our droppable UI clips for later: `setEngageDrop()` and `doDrop()`.

Building the droppable UI areas

Our next step in the process is to build the classes that are linked to our *droppable UI clips* (the clips that a book thumbnail item can drop into): the product detail and the cart. We've combined the discussions of these two classes together because, if you think about it, both need to define the same actions:

- They need to change their physical states when a draggable book item touches their respective hit areas, so the user knows that by releasing an item, it will interact with the respective object.
- They need to define what actually happens when a draggable book item is dropped over their hit area. For instance, the product detail clip would then display information pertaining to the item, whereas the cart would add the item to the user's checkout list.
- And finally, they actually need to define what part of the UI should be considered the active hit area. Recall from how we created the cart and product detail panels in the Flash environment that we added invisible `hit_mc` clips to define the active hit areas.

For the OOP whizzes out there, you might be thinking about defining an *interface* that contains these three methods, and then having the product detail and cart classes implement them. Well, that's what we're about to do!

Be careful not to confuse the term *interface* here with the design of graphical interfaces. An interface, in OOP terms, is a construct that can be implemented by a class that creates the shell of methods that are both required and further defined by the classes that implement it. Essentially, an interface is a contract that the implementing class agrees to conform to which aids in cross-class communication. If you're still a little confused, it will make a whole lot more sense if we actually look at our interface code.

Open the `DropArea.as` interface. You can see it's rather skinny, defining only three methods.

```
interface DropArea
{
    public function setEngageDrop(_engage:Boolean):Void;
    public function doDrop():Void;
    public function getHitArea():MovieClip;
}
```

The first method, `setEngageDrop()`, will define what happens to the clips when an item has been dragged over a droppable UI clip's hit area. It expects a Boolean value that tells the object whether the thumbnail item has come in contact with the object. The second method, `doDrop()`, will define what happens when an item is dropped over a droppable clip's hit area. The final method, `getHitArea()`, simply returns a reference to the movie clip within the droppable UI clip that represents its active hit area.

You can see why this construct is called an interface. When a class implements an interface, it defines the methods that are created from the interface. The interface itself has no knowledge of what the method definitions look like. It only knows that the methods are defined within each respective class. In essence, it is an "interface" to all the classes that implement it. We can safely assume any class that implements `DropArea` will have the `setEngageDrop()`, `doDrop()`, and `getHitArea()` methods ready to go!

The other great benefit is that we can now type an object as a `DropArea` object, as we did in the `DragDropManager` class. When doing so, we can reference the `setEngageDrop()`, `doDrop()`, and `getHitArea()` methods with the knowledge that they are implemented for any kind of `DropArea`.

In our specific case, we use an interface as a class contract. If we wanted to add more droppable UI clips in the future, they must define these three methods in their own class definitions. If they don't adhere to the contract, when we go to compile our Flash movie, we'll get compilation errors. Quite simply, it helps us ensure that each droppable UI clip will define its hit area as well as what happens when an item is dragged or dropped over one of its hit areas.

Note that in this particular chapter example, things would work perfectly fine if we didn't have a DropArea interface but just ensured that the setEngageDrop(), doDrop(), and getHitArea() methods existed in each of our droppable UI clips. However, adding in the interface helps us remember what methods we need to define. If we didn't have the DropArea interface, we would not get compilation errors if we had forgotten to define these methods in a droppable UI clip, which makes debugging the code a lot more difficult.

Now that we have the DropArea interface handy, let's see how easy it is to implement it for each of our three droppable UI clips.

The most basic implementation of either of the droppable UI clips is with the MC Cart clip. Open `Cart.as` in the `/source/classes/` folder, the class linked to the MC Cart clip. Notice that it not only extends the `MovieClip` object, but it also implements the `DropArea` interface. Here is the implementation of the three interface methods:

```

    public function setEngageDrop(_engage:Boolean):Void
    {
        if(_engage)
        {
            _alpha = 60;
        }
        else
        {
            _alpha = 100;
        }
    }

    public function doDrop():Void
    {
        setEngageDrop(false);
        added_mc.play();
    }

    public function getHitArea():MovieClip
    {
        return hit_mc;
    }

```

Notice that when the user drags a clip over the cart, `setEngageDrop()` simply readjusts its `_alpha` channel to 60% transparency. Otherwise, it will be set to the normal 100% transparency. When a user drops a clip into the cart, we play the `added_mc` clip, the simple text-based clip that just displays the message that the item was added to the user's cart.

FLASH APPLICATION DESIGN SOLUTIONS: THE FLASH USABILITY HANDBOOK

Next, let's look at our implementation of the interface for the MC ProductDetail clip. Open `ProductDetail.as` in the `/source/classes/` folder. Some of the properties shown here make sense only in the context of the entire class file.

```
public function setEngageDrop(_engage:Boolean):Void
{
    if(_engage)
    {
        _alpha = 60;
    }
    else
    {
        _alpha = 100;
    }
}

public function doDrop():Void
{
    setDetail(_parent.grid_mc.getSelectedData());
    setEngageDrop(false);
}

public function setDetail(_detailObj:Object):Void
{
    detailObj = _detailObj;

    info_txt.htmlText = "<b>Title: </b>" + detailObj.title + "<br>" +
    ➤ "<b>Published: </b>" + detailObj.published + "<br>" +
    ➤ "<b>ISBN: </b>" + detailObj.isbn + "<br>" +
    ➤ "<b>Price: </b> $" + detailObj.price + "<br>" +
    ➤ "<b>Pages: </b>" + detailObj.pages + "<br>";

    description_txt.htmlText = detailObj.description;

    imageHolder_mc._alpha = 0;
    preloader_mc._visible = true;
    preloader_mc.init(detailObj.fullsize, imageHolder_mc, this,
    ➤ imageLoaded);
}

public function getHitArea():MovieClip
{
    return hit_mc;
}
```

The `setEngageDrop()` method is exactly the same as it was in the `Cart.as` file. However, the `doDrop()` method will call a `setDetail()` method, which will then populate the `info_txt` and `description_txt` fields. Both are shown in the preceding code.

So, with that, we've shown you with fairly broad strokes how to implement the code behind our droppable UI clips. You should see that by using an interface, we ensure that the three critical methods—`setEngageDrop()`, `doDrop()`, and `getHitArea()`—are implemented properly.

As we mentioned at the onset, we've taken you through only the critical methods that drive the drag-and-drop interface. Other methods and properties sprinkled throughout the code files do bits of work that haven't been discussed in this chapter. We encourage you to comb through all of our source files to see how everything works!

Putting it all together

If you're still with us now, you deserve a pat on the back, or perhaps a hug. There's no doubt that this was one of the more intense discussions of code, but sometimes good usability requires some fairly intricate code development!

Now that we have defined all the critical parts that will implement our grid layout and drag-and-drop functionality, we need to add just a bit of initialization code to our FLA file to get the ball rolling.

If you head back up to the script layer of our MC Interface clip within `Chapter6_Final.fla` and view our frame actions, you'll see that this is where we create a new instance of the `DragDropManager`. We use the `addDropArea()` method to add in our droppable UI clips. When added, they are immediately included in the `dropAreas` array (discussed earlier) using the `addDropArea()` method.

```
var dragDropManager = new DragDropManager(this, "MC ThumbDragItem");
dragDropManager.addDropArea(productDetail_mc);
dragDropManager.addDropArea(cart_mc);
```

We then create an array of data for our book inventory items, called `productData`. This looks very similar to the `selectionData` array from Chapter 3, only here we've included the title, ISBN, publication date, description, and book thumbnail image path into each element of the array. The array is rather lengthy, so we've omitted it from this chapter, but it's available in the source code. Note that in a more robust implementation, you might consider an external XML file to pass in the data from a store database.

Finally, we write this one short, but powerful, line of code to begin the attachment of book inventory items to our MC `ProductGrid` instance:

```
grid_mc.doInit(productData);
```

Just like that, we've set the wheels in motion!

Summary

In this chapter, we've demonstrated a Flash solution for data display and interaction using the example of an online store. As always, we suggest that you think about new ways to extend the functionality we've shown you in this chapter. For instance, consider the following:

- Use sound to indicate successful thumbnail drops. While sounds can be annoying if they're too obnoxious, subtle ones can make it more obvious when a user successfully adds an item to a cart for instance.
- Make other objects draggable. Think about other ways you might use the drag-and-drop paradigm to enhance the user experience. How about if you could drag the product detail panel into the cart and have it create the same functionality as when you drag the corresponding book thumbnail item into the cart?
- Modify the grid layout structure. Our current grid layout allows for only 16 thumbnail items at a time. What methods could be employed to allow for more items? Perhaps you could use a mouse-position-based scroll, as we did for our menus in Chapter 4. Or, perhaps you could implement a numbered paging system that could load other pages asynchronously, and switch grid views through numbered page links.

Now that we've completed our look at setting up an inventory view and selection device, let's expand the store concept a bit further and talk about effective ways of filtering content for users. That's coming up in Chapter 7!

