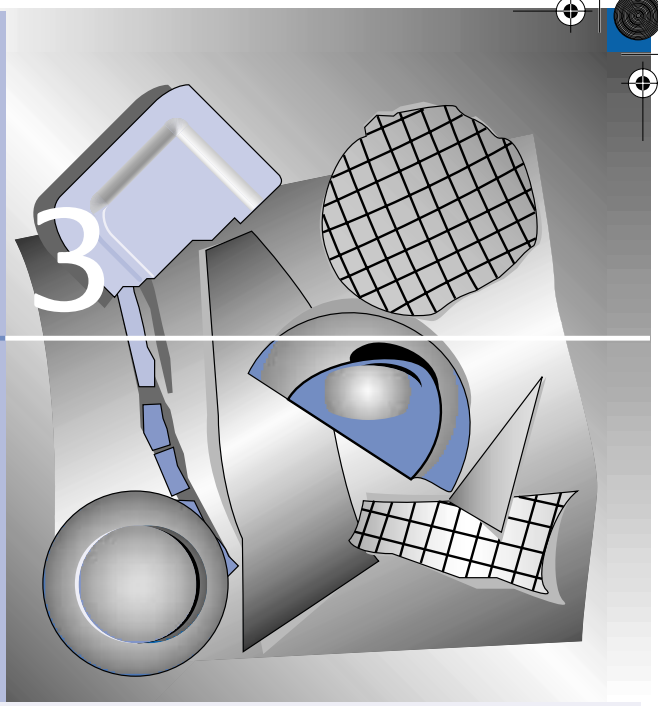
Chapter 3

Finding Objects



Joseph Albers could make colors dance or retreat: "I see color as motion . . . To put two colors together side by side really excites me. They breathe together. It's like a pulse beat . . . I like to take a very weak color and make it rich and beautiful by working on its neighbors. I can kill the most brilliant red by putting it with violet. I can make the dullest grey in the world dance by setting it against black." Albers, one of the great graphics artists of the twentieth century, was a master at making visual imagery emerge from form and color. By careful juxtaposition of colors, textures, and shapes, the artist can make images leap off the page. Albers calls this the "1 + 1 = 3" effect. A good design is more than the sum of its parts. A bad design muddles what should be emphasized. Chartjunk—misuse of bold lines and color or addition of pretty stuff that adds no value—shifts attention away from vital information. In graphic design, composition, form, and focus are everything! An object design poses similar challenges. It is strengthened by vivid abstractions and well-formed objects that fit into an overall structure. It can be weakened by glaring inconsistencies or muddled concepts.

A graphics designer enhances important information by layering and separating it, giving focus to the data rather than its container, and by using multiple signals to remove ambiguity.

The abstractions you choose greatly affect your overall design. At the beginning, you have more options. As you look for candidate objects, you create and invent. Each invention colors and constrains your following choices. Initially, it's good to seek important, vivid abstractions—those that represent domain concepts, algorithms, and software mechanisms. Highlight what's important. If you invent too many abstractions, your design can get overly complex. Not enough abstraction, and you'll end up with a sea of flat, lifeless objects.

Your goal is to invent and arrange objects in a pleasing fashion. Your application will be divided into neighborhoods where clusters of objects work toward a common goal. Your design will be shaped by the number and quality of abstractions and by how well they complement one another. Composition, form, and focus are everything.

A Discovery Strategy

Well-formed abstractions and careful attention to how they complement one another have a direct effect on the quality of an object design. This chapter discusses how to find and arrange software objects in an initial object design. The ultimate goal is to develop a practical solution that solves the problem. However, we find that such designs typically are also esthetically pleasing ones.

So let's get to it! Conceiving objects is a highly creative activity, but it isn't very mysterious. Finding good candidate objects isn't a topic that has received a lot of attention. Early object design books, including *Designing Object-Oriented Software*, speak of finding objects by identifying things (noun phrases) written about in a design specification. In hindsight, this approach seems naïve. Today, we don't advocate underlining nouns and simplistically modeling things in the real world. It's much more complicated than that. Finding good objects means identifying abstractions that are part of your application's domain and its execution machinery. Their correspondence to real-world things may be tenuous, at best. Even when modeling domain concepts, you need to look carefully at how those objects fit into your overall application design.

Although software objects aren't just waiting for you to find them, you can identify them somewhat systematically. Although many different factors may be driving your design, there are standard places to search for objects, and you'll find many sources of inspiration. You can use your knowledge of your application domain, your notions about needed application machinery, lessons learned from others, and your past design experience.

Our recipe for finding and assessing candidates has a number of steps:

- Write a brief design story. In it, describe what is important about your application.
- Using this story, identify several major themes that define some central concerns of your application.

- Search for candidate objects that surround and support each theme. Draw on existing resources for inspiration: descriptions of your system's behavior, architecture, performance, and structure.
- Check that these candidates represent key concepts or things that represent your software's view of the world outside its borders.
- Look for candidates that represent additional mechanisms and machinery.
- Name, describe, and characterize each candidate.
- Organize your candidates. Look for natural ways to divide your application into neighborhoods—clusters of objects that are working on a common problem.
- Check for their appropriateness. Test whether they represent reasonable abstractions.
- Defend each candidate's reasons for inclusion.
- When discovery slows, move on to modeling responsibilities and collaborations.

This chapter will cover each of these steps in greater detail. But be aware that you don't always complete each step before moving on to the next. The process of discovery and invention is more fluid than that. Sometimes you perform several steps at the same time. You may discard some candidates and start over if they don't seem to fit in to your emerging design. But if you start by characterizing what is vital to your application's success in a design story, you can then proceed with an organized search for objects that support this core.

At the end of your initial exploration, you will have several handfuls of carefully chosen, justified candidates. Many more will be invented as you proceed. These initial candidates are intentionally chosen to support some key aspect of your system. They will seed the rest of your design. Finding and inventing this first batch of candidates takes careful thought.

Looking for Objects and Roles, and Then Classes

The first candidates to look for should represent important things: concepts, machinery, and mechanisms. Typically these kinds of candidates are smart—they do things. They may know things, too, but they perform actions based on what they know. Initially, think very concretely. Abstraction will come later, after you see more concrete

Initially, we recommend you look for candidate roles and objects. Once you have an idea that they'll stick around, make decisions on how they are realized as interfaces and classes.

objects and understand their relationships to others. To start, identify distinct objects that have clear roles. Next, decide what candidates should know and do (their responsibilities) and whom they work with (their collaborators).

Then, thinking more abstractly, you can turn to identifying aspects that are common to a number of candidates. Shift your focus from thinking about objects and their individual roles to deciding what objects have in common. Only after you've made decisions about common responsibilities that are shared by different candidates can you define common roles. We deem our objects and roles candidates until their value has been proven. Only then do we decide how they will be realized as classes and interfaces.

Abstract and concrete classes are the building blocks we use to specify an implementation. Declaring interfaces is one means to make it more flexible and extensible. A reusable role is best specified as an interface that can be implemented by one or more classes.

When you transition from candidates to classes and interfaces, you have options. You can employ inheritance, abstraction, interfaces, and collaborations to construct a well-factored, flexible design. You will specify abstract and concrete classes as well as interfaces. An *abstract* class provides a partial implementation of responsibilities. It leaves subclasses with the obligation to implement specific responsibilities. A *concrete* class provides a complete implementation. An *interface* specifies responsibilities more precisely as method signatures but leaves their implementation open. Any class can implement an interface, regardless of its position in any class inheritance hierarchy.

Why Tell a Design Story?

We suggest you create a framework for searching for potential candidates by writing a story about your application. After you've done this, the candidates you identify should fall into place and support various aspects of your story. When you state things in your own words, you get to decide what's important. Everybody may have been talking about what your design should do and what will make it great, but you should make a few bold statements of your own. In this design story, identify the things about your application that you know with certainty, as well as things you don't yet know. Rather than being driven by one particular view of your software—whether it be use cases, requirements, architecture, users, or sponsors—pull together all these factors and craft your own description.

Write a rough story—two paragraphs or less is ideal. Don't take a lot of time revising and polishing it. Be quick and to the point. What is notable about your application? What is it supposed to do? How will it support its users? Is it connected to a real-world example that you can study? Have you done something similar? What will make your

design a success? What are the most challenging things to work out? What seems clear? What seems ill defined? You need not answer all these questions. Simply write about the critical aspects of your application. If it helps you make your point, draw a rough sketch or two. Focus on the main ideas.

Here are two design stories that were written quickly. The first one rambles. It tells of an online banking application:

> This application provides Internet access to banking services. It should be easily configured to work for different banks. It should support fast access to banking services for potentially thousands of users at a time. There is a limited number of software resources, such as database connections and connections to backend banking software, that are available. A critical element in the design is the declaration of a common way to call in to different backend banking systems and a reliable means of sharing scarce resources. We will define a common set of banking transactions and a framework that will call into banking-specific code that "plugs into" the standard layer implementing the details. The rest of our software will only interface with the bank-independent service layer.
>
> We've developed a prototype implementation of this layer and have configured it to work for two different banks. Although it is still a prototype, we understand how to write a common banking service layer. Lately, our bank has been busy acquiring other banks and integrating their software. We've been through three system conversions in the past year. We want to focus on making this service layer easy to implement and test. At the heart of our system is the ability to rapidly configure our application to work for different backends and to put a different pretty face on each. This includes customizing screen layouts, messages, and banner text. The online banking functions are fairly simple: Customers register to use the online banking services and then log in and access their accounts to make payments, view account balances and transaction histories, and transfer funds. This is straightforward, easy to implement. There is added complexity. Customers record information and notes about each online transaction. This extra information will be maintained by our application in its own database because preexisting bank software has no way to store it. We want a customer to view human-readable information, not ancient bank software detailed transaction records. When a customer asks to view an account's transaction history we'll have to merge this data with records supplied from the backend software. Multiple users can access a customer's accounts, each with potentially different access rights. Certain users might have no access to sensitive accounts. A company executive might view only account balances, whereas a clerk in the accounts payable department could make payments and a comptroller might be able to transfer funds between accounts.

If you are a member of a larger design team, write your own story first and then share it with your team. See how your concerns differ from others'. The team can draft a single, unified story, but this isn't necessary. More importantly, identify the important themes in these design stories. Then look for candidates that support these themes.

This next, more focused, story is about a Web-based game. It describes new design challenges as well as, to us, familiar territory:

> This game playing application supports an Internet variant of chess called Kriegspiel. Kriegspiel is a chess version of the popular game Battleship. The novelty is that players make moves not knowing where their opponent's pieces are located. Our immediate concern is how to distribute responsibilities among major software components. In this distributed application, we need to consider time lags and limited communication bandwidth between architectural components. We also need to consider the unpredictability of Internet communications. Each player interacts with our application via a Web browser. Hundreds of games can be played simultaneously. A user logs in and requests to play a game with another. If no one is available, the user can elect to play a game with the computer. We will need to design our software to play a credible game of Kriegspiel as well as referee games played by humans. A game can be suspended and resumed. From our computer gaming experience, we know that computerized games generally have player input directives, rules about legal actions, some representation of the current state of the game, and animations. In this application, our animations are simple and not a major concern. It is worth stating how Kriegspiel is played, although our application won't mimic the real-world game. We will draw design ideas from this description.
>
> In the game of Kriegspiel, three boards and sets of chessmen are used. There is a referee, whose chess set is in the center, with two players seated back-to-back, each at his own board. Each player moves his own chessmen, and the referee duplicates each move on his own board. The referee tells a player when his attempted move is impossible. Each player tries to guess what move his opponent is making. When a player completes a legal move, the referee announces, "Black (or White) has moved." When a player tries an illegal move, the referee waves his hand to prevent it but does not let the opponent know. When a move results in a capture, the referee announces, "Black (or White) captures on (the rank, file, long or short diagonal)" and removes the captured piece from the board of the player who lost it. A player may ask, "Any?" and be told by the referee if he has a possible capture with a pawn. That's the only question he is permitted. Having asked the question he must try at least one pawn capture before making a different move. To summarize, players make moves, ask "Any?," suspend or resume a game, claim a draw, or concede.

Let's contrast what we can glean from each story and then sketch out our candidate search strategies. The underlying requirement for the online banking system is flexibility. Functionality, implementation,

and information need to be configurable. The application will maintain additional user-supplied information and construct account history from online and other banking transactions.

Our strategy for identifying candidates for this application will be to focus initially on modeling concepts that represent online banking services, the common interface to backend banking systems, and accounts. We should have objects that are responsible for performing banking functions and storing application-specific information about online transactions. Because we are building a multiuser online system, we also need objects that are responsible for managing access to limited resources such as the database and backend banking system connections. The key themes in the banking story are

- Modeling online banking services
- Flexibly configuring behavior
- Sharing scarce software resources among thousands of users
- Supporting different views of accounts and access privileges

The Kriegspiel application, even though it too is an Internet application, has fundamentally different drivers. As with any gaming application, we need to take a step back from our vivid real-world reference of the physical board game and ponder what mechanisms and inventions are needed by a computerized game. This is always a major design challenge with gaming applications. It is one we are familiar with from past experience. Our goal in designing Internet Kriegspiel isn't to simulate the real world but instead to construct a model that represents what is needed to run a computerized game. Choosing the right abstractions to represent the game and moves will be critical. We also need to consider how running over the Internet will impact our design. This will affect how we divide the work between application components. Finally, we'll need to implement a semi-intelligent computerized game player—something that is smart enough to play a decent game against a human opponent. Our central concerns for Internet Kriegspiel:

- Game modeling
- Computer playing a game
- Partitioning responsibilities across distributed components

SEARCH STRATEGIES

Once you have identified major themes, you can use them as sources of inspiration. Make educated guesses about the kinds of inventions that you will need based on the nature of your application and the things that are critical to it. Candidates generally represent the following:

- The work your system performs
- Things directly affected by or connected to the application (other software, physical machinery, hardware devices)
- Information that flows through your software
- Decision making, control, and coordination activities
- Structures and groups of objects
- Representations of real-world things the application needs to know something about

We guide our search from these perspectives. The kinds of inventions we seek are closely related to the role stereotypes.

If an application's central mission boils down to computation, look to populate it with objects playing the role of service providers that calculate, compute, transform, and figure. You will likely invent objects that represent algorithms or operations along with objects that control work processes. If your application's major activity is to assemble and move information from one place to another, identify candidates that model this information as objects along with others to coordinate their movement. If your application connects with other systems, invent external interfacers that form these connections. Most designs need objects that control or coordinate the work of others. Depending on the complexity of the control, this design decision may or may not be a prominent one. If your application needs to sort through, organize, and make connections between related objects, structurers need to be identified. There are relatively direct links between the kinds of objects you look for and the nature of the work your software carries out.

The best way to evaluate potential candidates that represent external things is to shift perspective. Climb into your software and look out at the world. Take your application's viewpoint. Ask what you need to know about your users, the systems you connect to, and things out there that you affect.

As you look for candidates one question to ask is, "How much does our software need to know about things in the external and virtual worlds it is connected to?" At the borders, model connections to other systems as interfacer objects. You may include in your design objects that represent these other software systems. These service providers will be called upon by other parts of the application. But

when should you model things that are outside a computer, such as your software's users? If it is only their actions that matter and not whom they are, leave them out of the design. Users' actions can be conveyed via user interface objects (objects charged with translating user requests and information to other parts of the system). There is no need to know who is pushing your application's buttons! On the other hand, if whom users are makes your software behave differently, include some representation of them as a candidate. Some knowledge of its users (and objects to represent that knowledge) is needed if your software bases any decisions on whom it interacts with. For example, if different users have different access rights to accounts or if the ability to resume a game requires knowledge of whom the players are, then some representation of these users should be part of the design.

Tables 3-1 and 3-2 outline our search strategies for our two applications. Although we consider each perspective, typically only one or two are relevant to any particular theme. If we find that a particular perspective does not yield any insights, we move on. For each theme, we briefly summarize the perspectives that yielded insights and the kinds of candidates we are looking for.

Table 3-1 *The initial search for online banking application candidates is based on exploring four themes.*

Theme	Perspective	Candidates That Specifically Support. . .
Online banking functions	The work our system performs	Performing financial transactions, querying accounts
	Things our software affects	Accounts, backend banking system transactions
	Information that flows through our software	Information about transactions, account balances, transaction amounts, account history, payments
	Representations of real-world things	Customers, users, and the accounts they access

Continues

Table 3-1 *The initial search for online banking application candidates is based on exploring four themes. (Cont.)*

Theme	Perspective	Candidates That Specifically Support. . .
Flexibly configuring behavior	Things our software affects	A common interface to back-end systems
	Information that flows through our software	Configurable display of Web page banners, text, messages, and account formats
Sharing scarce resources	Structures and groups of objects	Managing limited connections to backend systems and our online banking application database
Different views of and access to accounts	The work our system performs	Restricting users' views of and ability to perform banking transactions that modify account balances
	Decision making, coordination, and control	Prohibiting access to accounts unless user has specific privileges

Table 3-2 *The initial search for Kriegspiel application candidates is based on the themes of game modeling, intelligent computerized game playing, and distributed games.*

Theme	Perspective	Candidates That Specifically Support. . .
Game modeling	The work our system performs	Assigning players to games, refereeing, storing and resuming suspended games, playing a game, determining the legality of a move, determining the outcome of a move, displaying the state of each player's board
	Information that flows through our software	Information about moves and player requests

Table 3-2 *The initial search for Kriegspiel application candidates is based on the themes of game modeling, intelligent computerized game playing, and distributed games. (Cont.)*

Theme	Perspective	Candidates That Specifically Support. . .
Game modeling *(Cont.)*	Representations of real-world things	Players and their actions
	Structures and groups of objects	Managing saved games, the various games, game pieces, and their locations on a game board
Computer playing a game	The work our system performs	Playing a game with a user
	Decision making, control, and coordination	Determining a reasonable move to make based on the current view of the game (which should be just as limited as any human player's view)
Partitioning responsibilities across distributed components	Decision making, control, and coordination	Communicating a player request to the referee and game state between players, detecting whether a player is still connected
	Information that flows through our software	Player moves, updated boards, and game state

We will identify candidates that support the relevant perspectives. Sometimes candidates leap right out of the page from our brief descriptions; are Player and PlayerAction good candidates based on the fact that we need to have candidates that support our game's real-world view of "players and their actions"? Highly likely. At other times, we must speculate about exactly how our software might work in order to come up with candidates; perhaps there should be a BankingServicesConnectionManager that manages BankingServicesConnections or a DatabaseConnectionManager to manage DatabaseConnections that are scarce resources? Often, different themes and perspectives reiterate and reinforce the need for certain kinds of candidates. This is good. It builds confidence in the relevance a

particular candidate has to our application. At other times, ideas do not come so quickly, and we must think more deeply to come up with potential candidates.

We won't find all the key candidates in this first pass; nor will our initial ideas about our candidates remain fixed. Our notions change as we give candidates further definition. The initial candidates that we come up with will seed our design. So it is particularly important to give each candidate a strong name that suggests its role and purpose. So before we continue searching for candidates, let's explore what it takes to find useful names.

What's in a Name?

"... the relation of thought to word is not a thing but a process ... Thought is not merely expressed in words; it comes into existence through them. Every thought tends to connect something with something else, to establish a relationship between things. Every thought moves, grows and develops, fulfills a function, solves a problem."

—Lev Vygotsky

Good names increase design energy and momentum. You can build on a good name. When the name of a software object is spoken, designers infer something about an object's role and responsibilities. That's why grizzled object designers say, "Choose names carefully." A well-formed name creates a link to past experience and common practice. Meaning comes along with any name, whether we like it or not. Our brains are wired to find connections to things we already know. So the key to giving an object a good name is to make its name fit with what you already know while giving a spin on what it should be doing. Most names fit into a system of names. Different naming schemes coexist, even within a single application. There isn't one universal naming system.

Qualify generic names. One scheme for naming things that are special cases of a more generic concept is to tack on to the generic name a description of that special case.

> A Calendar represents a system of dates and time at a particular location. GregorianCalendar extends the Calendar class. Following convention, we could invent JulianCalendar or ChineseCalendar classes. Others familiar with this scheme could make educated guesses about how their implementations would differ from GregorianCalendar.

Include only the most revealing and salient facts in a name. The downside of any descriptive scheme is that names can become lengthy. Don't name every distinguishing characteristic of

an object; hide details that might change or should not be known by other objects.

> Should people really have to care that they are using a MillisecondTimerAccurateWithinPlusOrMinusTwoMilliseconds, or will Timer suffice? Detailed design decisions should not be revealed unless they are unlikely to change and they have a known impact on the object's users. Exposing implementation details makes them hard to change.

Consider the Singleton pattern described in the *Design Patterns* book. This pattern ensures that a class has only one instance with a global point of access. We could name every concept that applies this pattern a MumbleMumbleSingleton. Following our guideline, we recommend against this. Singleton is a distinction that is more important to a class implementer than to a client who uses a singleton. Give names that will be meaningful to those who will be using the candidate, not those who will be implementing it. If someone using your candidate must know the details of its implementation, you have likely missed an opportunity to do a better job of abstraction. One possible exception to this rule is to append Singleton to a class name when it is crucial for its users to know this.

Give service providers "worker" names. Another English language naming convention is to end job titles with "er." Service provider objects are "workers," "doers," "movers," and "shakers." If you can find a "worker name," it can be a powerful clue to the object's role.

> Many Java service providers follow this "worker" naming scheme. Some examples are StringTokenizer, SystemClassLoader, and AppletViewer.

If a worker-type name doesn't sound right, another convention is to append Service to a name. In the CORBA framework, this is a common convention—for example, TransactionService, NamingService, and so on.

Look for additional objects to complement an object whose name implies broad responsibilities. Sometimes a candidate represents a broad concern; sometimes its focus is more

narrow. If you come across a name that implies a large set of responsibilities, check whether you've misnamed a candidate. It could be that your candidate should have a narrower focus. Or it might mean that you have uncovered a broad concept that needs to be expanded. Looking for objects that round out or complement a broad name can lead to a family of related concepts—and a family of related candidates. Many times we need both specific and general concepts in our design. The more generic named thing will define responsibilities that each specific candidate has in common.

> An object named AccountingService likely performs some accounting function. The name AccountingService isn't specific. We cannot infer information about the kinds of accounting services it performs by looking only at its name. Either AccountingService is responsible for performing every type of accounting function in our application, or it represents an abstraction that other concrete accounting service objects will expand upon. If this is so, we'd expect additional candidates, each with a more specific name such as BalanceInquiryService, PaymentService, or FundsTransferService. These more specifically named candidates would support specific accounting activities.

Forming an abstraction by looking at two specific cases might work, but comparing and contrasting three or four cases is even better. The more closely related concepts you can compare and contrast in order to identify what they have in common, the better.

Highlight a general concept with more specific candidates. If you can think of at least three different special cases, keep both the general concept and specific ones. If later on, you find that these more specific candidates don't share any responsibilities in common, the more abstract concept can always be discarded. However, if you have simply assigned a candidate a name that is too generic, by all means rename it.

> If your candidate could represent historical records of many other things, better to leave it with a more generic name, History, instead. If you intend to model transaction history, rename your candidate TransactionHistory. You decide how specific you want to be.

Therein lies the art of naming: choosing names that convey enough meaning while not being overly restrictive. Leave open possibilities for giving a candidate as much responsibility as it can handle, and for using it in different situations with minor tweaks. It certainly is a more powerful design when a candidate can fit into several different situations. The alternative—having a different kind of object for each different case—is workable, but not nearly so elegant.

Choose a name that does not limit behavior. Don't limit a candidate's potential by choosing a name that implies too narrow a range of actions. Given the choice, pick a name that lets an object take on more responsibility.

Our thoughts shape our words, and our words influence our thoughts. Names subtly shape our ideas about our candidate's expected behaviors.

Consider two alternatives for a candidate: Account or AccountRecord. Each could name an object that maintains customer information. From common knowledge we know one meaning of record is "information or facts set down in writing." An AccountRecord isn't likely to have more than information holding responsibilities if we fit its role to conventional usage of this name. The name Account, however, leaves open the possibility for more responsibilities. An Account object could make informed decisions on the information it represents. It sounds livelier and more active than AccountRecord.

Choose a name that lasts for a candidate's lifetime. Just as it seems funny to hear a 90-year old called "Junior," it's a mistake to name a candidate for its earliest responsibilities, ignoring what else it may do later on. And don't be content to stay with the first name you give a candidate if its work changes.

An object that defines responsibilities for initializing an application and then monitoring for external events signaling shutdown or re-initialization, is better named ApplicationCoordinator than ApplicationInitializer. ApplicationInitializer doesn't imply having ongoing responsibilities after the application is up and running. ApplicationCoordinator is a better name because its more general meaning encompasses more responsibilities.

Choose a name that fits your current design context. When you choose names, select ones that fit your current design surroundings. Otherwise, your candidates' names may sound strange. What sounds reasonable in an accounting application may seem jarring in an engineering application.

A seasoned Smalltalker tried hard to set aside his biases when he started working with Java. Although he expected Java classes to have totally different responsibilities, he was surprised to find the Java Dictionary class to be abstract. In Smalltalk, Dictionary objects are created and used frequently.

Shed your past biases when they don't fit your current situation.

A Java designer can define classes with the same name, each residing in a different package. You should do so only if one package is designed as a replacement for another.

Do not overload names. Unlike spoken language, where words often have multiple meanings, object names should have only one meaning. It isn't good form to have two different types of Account objects with radically different roles that coexist in the same application. Some object-oriented programming languages let you assign the same name to different classes but then force you to uniquely qualify a name when you reference a particular class in code. In Java, for example, classes from different packages can have the same name. In order to uniquely designate a specific one, its name must be qualified by the name of the package where it is defined.

Names of things that can simultaneously coexist within a single application should be given different names. Don't overload a name. Programmers have only one context—the running application—in which to interpret names. They already have enough to think about without adding yet another source of confusion. Compilers are good at automatically applying the correct qualification to a name. Humans aren't!

Eliminate name conflicts by adding an adjective. Sometimes the best names are already chosen. Still, you need to name your candidate. By adding a descriptive phrase to a name, you can come up with a unique name.

> The candidate TransactionProperties might be a reasonable name for a candidate whose preferred name conflicts with the preexisting Java class named Properties.

A word of caution: If your candidate has a radically different meaning, don't co-opt a familiar name. Follow convention. Designers familiar with existing names will expect your candidate to fit in and work similarly.

Eliminate conflicts by choosing a name with a similar meaning. Sometimes, your best bet is to look for a synonym. Each synonym has a slightly different shade of meaning, so finding a satisfactory name may be hard.

> The synonyms for Property, a class defined in the Java libraries, include these words: characteristic, attribute, quality, feature, and trait. Although "attribute" or "feature" might work, "characteristic" seems stuffy, and "quality" seems strained.

Choose names that are readily understood. A name shouldn't be too terse. Don't encode meaning or cut corners to save keystrokes. If you want others to get a sense of an object's role without having to dig into how it works, give it a descriptive name. A name can be descriptive without being overly long.

"Acct" is too cryptic. "Account" is better.

If your problem domain has well-known and understood abbreviations—such as USD in banking, or Mhz or Gbyte in technology—it is reasonable to include these in a candidate's name.

DESCRIBING CANDIDATES

We judge an object by how well its name suits its role and how well its role suits its situation. Stereotyping a candidate's role provides a handy means for quickly creating an image about an object's intended use. When you find a candidate, name it and then characterize it as fitting one or more stereotypes. Each candidate could be a service provider, controller, coordinator, structurer, information holder, or interfacer. To be even more specific, you may want to distinguish between three different types of interfacers: *user interfacers* (objects that interface with users), *external interfacers* (objects that interface between your application and others) or *intersystem interfacers* (objects that bridge different parts of an application).

To be more explicit with your intentions, you can distinguish whether an object is designed to be passive and just hold on to related information (an information holder), or whether you expect it take a more active role in managing and maintaining that information (an information provider). If these finer distinctions seem too subtle, don't fret about them. Don't worry about giving an object the "right" stereotype. If your application is populated with objects that don't seem to fit these predefined stereotypes, come up with your own stereotypes. Stereotyping is intended to help get you started thinking about your candidates, not to bog you down.

If you aren't sure about the role your candidate will play, make an educated guess. Use its stereotype as a guide to build a simple definition. In that definition, explain what your candidate might do and list any traits that distinguish it from others. Write this brief definition on the unlined side of a CRC card (see Figure 3-1).

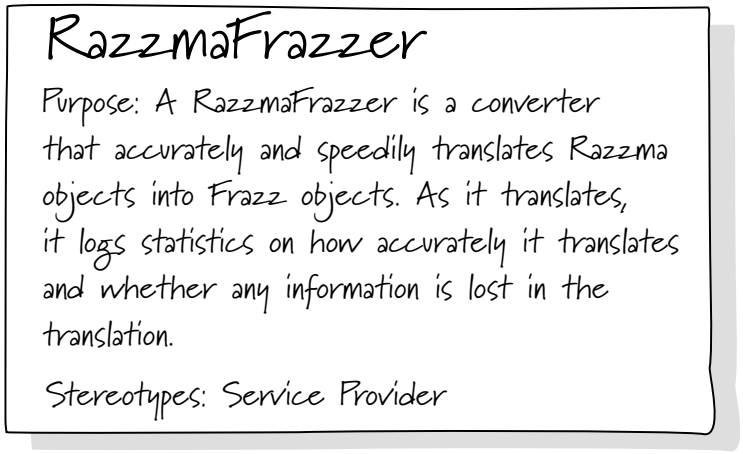


Figure 3-1
The unlined side of a CRC card is used to describe an object's purpose and stereotypes. In this case, a RazzmaFrazzer has only one stereotype.

More generally, a pattern to follow when describing an object is as follows:

> An object is a type of thing that does and knows certain things. Briefly, say what those things are. Then mention one or more interesting facts about the object, perhaps a detail about what it does or knows or who it works with, just to provide more context.

Service providers, controllers, and coordinators are distinguished by what they do. Here's a simple way to describe these stereotypes:

> A service provider (or controller or coordinator) is some kind of thing that does some kind of work. Briefly, describe this work. Then mention something about what is important or interesting about the work it performs or whom it interacts with.

If you are working on your own, you may feel less of an urge to write down these thoughts. After all, you know what you mean! Even so, it still can be helpful to jot down an abbreviated thought. You don't want to forget what was so important about that darned RazzmaFrazzer by next Friday. Similarly, if you are working in a team, others

likely won't know what's important about a candidate unless you tell them. Any description you can write about a candidate's purpose and what you expect it to do will help.

Consider this definition:

> A compiler is "a program that translates source code into machine language."

Contrast it with this slightly abbreviated definition:

> "A compiler translates source code into machine language."

The two definitions are nearly identical. The first adds that a compiler is a software program. This seems nit-picky—as software designers, we all know that compilers are programs. But the first definition provides just enough context so that someone not on our same wavelength can relate a compiler to other computer programs. Whenever you can relate something to a widely understood concept (such as a computer program), its meaning will be clearer to all.

If you and your fellow designers eat, sleep, and breathe design 24 hours a day, a lot may remain unspoken and unwritten. You understand one another because you think alike. However, if there's ever a question or disagreement about what a candidate is, it could be that you are making different assumptions. To make intentions clear, add enough detail to remove any doubt; then expect to have a discussion about whose ideas are better. Describe both what a candidate is and what it is not. Relate it to what's familiar.

We provide even more context by giving examples of how a candidate will be used and a general discussion of its duties. This is particularly important when you are describing a role that can be assumed and extended by several different objects.

> A FinancialTransaction represents a single accounting transaction performed by our online banking application. Successful transactions result in updates to a customer's accounts. Specific FinancialTransactions communicate with the banking systems to perform the actual work. Examples are FundsTransferTransaction and MakePaymentTransaction.

If a common meaning suits a candidate, use it to form a basic definition. Don't invent jargon for invention's sake. In the case of alternative definitions, choose one that most closely matches your application's themes. Start with a standard meaning, if it fits. Then describe what makes that object unique within your application.

The American Heritage Dictionary has six definitions for *account:*

1. A narrative or record of events
2. A reason given for a particular action
3. A formal banking, brokerage, or business relationship established to provide for regular services, dealings, and other financial transactions
4. A precise list or enumeration of financial transactions
5. Money deposited for checking, savings, or brokerage use
6. A customer having a business or credit relationship with a firm

It isn't too much of a stretch to conceive of different candidates that reflect each of these definitions. In our online banking application, accounts most likely represent money (definition 5). Rules that govern access to and use of funds are important. Different types of accounts have different rules. Although it is conceivable that an account could also be "a precise list of financial transactions" (definition 4), we reject that usage as being too far off the mark. People in the banking business think about accounts as money, assets, or liabilities and not as a list of transactions. In the same fashion, we reject definition 6. It doesn't specifically mention assets. We easily reject definitions 1 and 2 as describing something very different from our notion of accounts in banking. In banking, accounts represent money. We choose definition 5 because it is the most central concept to the world of banking:

> An account is a record of money deposited at the bank for checking, savings, or other purposes.

Add application-specific facts to generic definitions. The preceding definition is OK, but it is too general for online banking. In the online banking application, users can perform certain transactions and view their balances and transaction histories. We add these application specifics to our original description:

> An account is a record of money deposited at the bank for checking, savings, or other purposes. In the online banking system customers can access accounts to transfer funds, view account balances and transaction historical data, or make payments. A customer may have several bank accounts.

The more focused a candidate is, the better. Of course, a candidate may be suited to more than one use. Objects can be designed to fit into more than one application. A framework operates in many different contexts. A utilitarian object can be used in many cases. If you want your candidate to have a broader use, make this intent clear by writing the expected usage on the CRC card.

Distinguish candidates by how they behave in your application. If distinctions seem blurry in the world outside your software, it is especially important to clarify your software objects' roles. Even if you can distinguish between a customer and an account, you still need to decide whether it is worth having two candidates or to have one merged idea. (Don't expect the business experts to help make this decision. It is a purely "technical" modeling one.) A candidate that reflects something meaningful in the world outside your application's borders may not be valuable to your design.

Let's look at the sixth definition of account:

> "An account is a customer having a business or credit relationship with a firm."

What is the difference between a customer and an account? Are they the same? If we had chosen this definition, would we need both customer and account objects in our banking application?

When you discover overlapping candidates, refine their roles and make distinctions. Discard a candidate or merge it with another when its purpose seems too narrow (and could easily be subsumed by another candidate). When in doubt, keep both.

For both Customer and Account to survive candidacy and stick in a design, their roles must be distinct and add value to the application. We could conceive of a Customer as a structurer that manages one or more Account objects. And, in the online banking application, one or more users can be associated with a Customer. For example, the customer "Joe's Trucking" might have four authorized users, each with different privileges and access rights to different accounts. Another option would be to give an Account responsibility for knowing the customer and users. We could then eliminate Customer. We decide to include both Customer and Account in our design because giving those responsibilities to Account objects doesn't seem appropriate—we can envision customers and users sticking around even when their accounts are closed (and perhaps new accounts are opened). So customers are somewhat independent of accounts.

During exploratory design, expect a certain degree of ambiguity. You can always weed out undistinguished candidates when you find they don't add any value. Put question marks by candidates that need more definition. A candidate is just that—a *potential* contributor.

Characterizing Candidates

Before eliminating any possibility, consider how a candidate might work and how it relates to others. It is best to consider a candidate in a larger context. We can characterize candidates according to their

- Work habits
- Relationships with others
- Common obligations
- Location within an application architecture
- Abstraction level

To explore a candidate's work habits, ask, "What does it do, and how does it fit in?" Take one point of view—from the outside looking in. This is the same view a peer or client would take. Speculate about what services it might offer or how it might affect others. Think about these things, but don't assign responsibilities just yet. Ask whether the object is self-contained, working on its own initiative, or directed by others. Will it be constantly busy? Or will it need to be prodded into action? Is it an important, central character, or is it somewhere on the periphery? Ask what each candidate might do

and be. If you haven't any idea, dig in and look for its potential value. If you are undecided, spend a few minutes speculating how it might fit into its neighborhood and about the nature of its role:

> We think of an Account as an information holder. So we do not think of it adjusting its balance on its own—it is probably changed by outside requests (both online banking transactions and other account activity). An Account knows its balance and transaction history. An account doesn't manage its customer, so it doesn't have much of a structuring role, but it is associated with its customers (does it need to know its customer, or does its customer know about it?). It isn't obvious how backend banking transactions that affect an account's status will be controlled (will an Account be involved in delegating this work or not?) —so we are uncertain how much work it will actually do. We'll defer thinking through these issues until we develop a more detailed blueprint for our application's control architecture.

CONNECTING CANDIDATES

Given its limited space, what you can say on a CRC card will be brief. But CRC cards are much more than a compact space to record design ideas. They are real and tangible. You can pick up a card and talk about it as if it were the object itself, forgetting that the card "stands in" for a "real" object. You can use CRC cards to explore what candidates are and how they relate to others. You can move a card closer to any collaborators. You can poke at them, making as many connections and distinctions as you can. You can pick them up and lay out a new arrangement that amplifies a fresh insight, looking for patterns and similarities and differences. Which objects do similar things? Put them in a pile. Which objects are part of a neighborhood working on part of the problem? Move them closer. Get a sense of how your candidates fit and relate. Some useful ways to cluster candidates are as follows:

CRC cards, as invented by Ward Cunningham and Kent Beck, were originally used to teach object-oriented concepts. They have far broader applicability than as teaching aids. They can help you think about and link candidates.

- By application layer
- By use case
- By stereotype role
- By neighborhood
- By abstraction level
- By application theme

There is no standard way to fill out or use CRC cards. Several books have been written on the "art" and "practice" of CRC card modeling. David Bellin and Susan Suchman Simone's *The CRC Card Book* (Addison-Wesley, 1997) talks much about the process and people aspects of CRC cards. In Nancy Wilkinson's *Using CRC Cards: An Informal Approach to Object-Oriented Development* (SIGS, 1995), a CRC model for a library application is worked out and its translation to a C++ implementation is described.

Figure out what works best for you. Use CRC cards to express your ideas. Jot down initial ideas on the unlined side: At the very minimum, record a candidate's name, a brief description, and its role stereotypes (see Figure 3-2). That's mainly what you're initially looking for. Later you'll get more specific.

But you can also note things of interest: Does a candidate play a role in a well-known design pattern? Name that pattern and the candidate's role in it. Is it intended to fit into a narrow context, or, if carefully designed, might it be used in different applications? Note anything unusual and worth remembering. Is it an important abstraction? Put a big star by its name. As shown in Figure 3-3, use CRC cards to express what you think is important to know about a candidate.

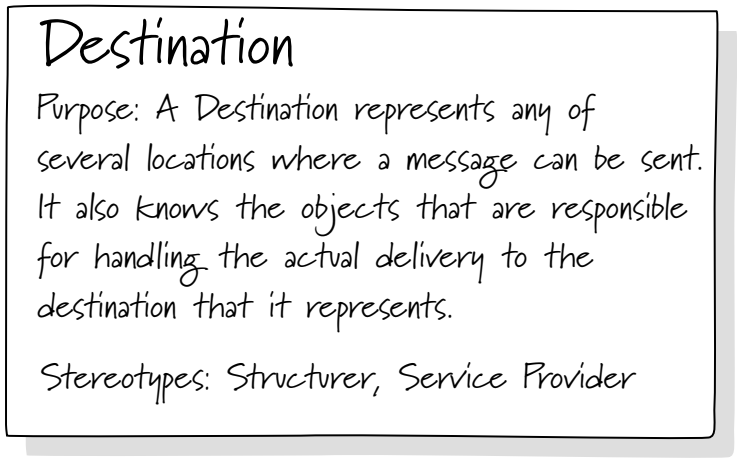


Figure 3-2
The purpose of a candidate is recorded on the unlined side of a CRC card.

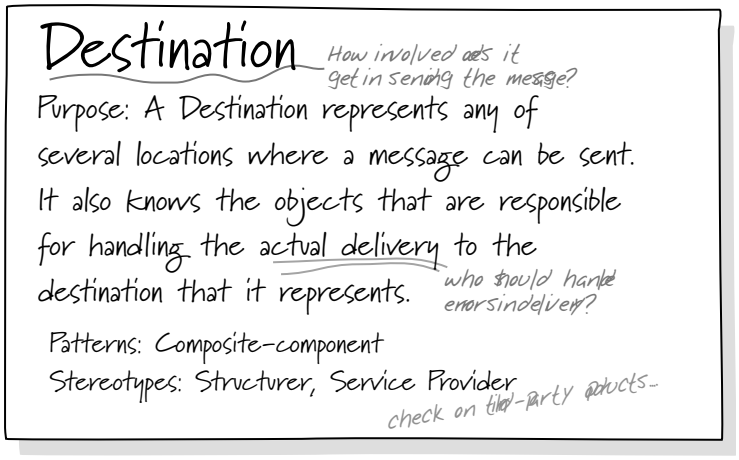


Figure 3-3
You can add scribbles, questions, and comments to a CRC card to help you remember key points.

LOOKING FOR COMMON GROUND

Earlier, we suggested that you make sharp distinctions between candidates. If you couldn't find enough differences, we recommended that you merge candidates that have overlapping roles. Now we suggest that you take another, closer look at your candidates. This time you want to see what your candidates have in common. You should always be on the lookout for common roles and responsibilities that candidates share. If you can identify what candidates have in common, you can consciously make your design more consistent by recognizing these common aspects and making them evident. You can identify a common category that objects fit into. You can define a common role that all objects in a category play. Shared responsibilities can be defined and unified in interfaces. Objects that collaborate with them can ignore any differences and treat them alike. Furthermore, a class can be defined to implement shared responsibilities that make up a shared role, guaranteeing that the implementation of classes that inherit these implemented responsibilities works consistently.

You are likely to find several ways to organize your candidates. Some will be more meaningful than others. Each one that seems useful will likely contribute to your design's clarity. The more you can identify what objects have in common, the more opportunities you have to make things consistent. Eventually you may define several new roles that describe commonly shared responsibilities. Your initial cut at this won't be your last. Keep looking for what objects have in common and for ways to exploit commonalities to simplify your design.

Common behavior could also imply the need for another candidate that is the supplier of that shared behavior.

Look for powerful abstractions and common roles. Things in the real world do not directly translate to good software objects! Form candidates with an eye toward gaining some economy of expression. Carefully consider which abstractions belong in your object design.

In our Kriegspiel game, there are various actions that a player can perform: "propose a move," "ask whether a pawn can capture in a move," "suspend a game," and so on. It's a pretty safe bet that we have a different candidate for each action: ProposeAMove, SuspendAGame, and so on. Proposing a move seems quite distinct from suspending a game. A harder question is whether we should define PlayerAction as a common role shared by each of these action-oriented candidates. If we can write a good definition for PlayerAction, we should do so and define a role that is shared by all player action candidates. There seem to be several things common to all actions (such as who is making the request and how long it is active). Eventually, if we find enough common behavior for PlayerAction, it will be realized in our detailed design as a common interface supported by different kinds of PlayerAction objects. We may define a superclass that defines responsibilities common to specific player action subclasses. Or common behavior might imply the need for another candidate that is the supplier of that shared behavior.

Look for the right level of abstraction to include in your design. Finding the right level of abstraction for candidates takes practice and experimentation. You may have made too many distinctions and created too many candidates—a dull design that works but is tedious. At the end of the day, discard candidates that add no value, whether they are too abstract or too concrete. Having too many candidates with only very minor variations doesn't make a good design. Identify candidates that potentially can be used in multiple scenarios.

Certain actions affect the position of pieces on a board. Should we have different candidates for each piece's potential types of moves? Not likely. This solution is tedious and offers no design economy. If you can cover more ground with a more abstract representation of something, do so. A single candidate can always be configured to behave differently under different situations. Objects encapsulate information that they can use to decide how to behave. The ProposeAMove candidate can easily represent all moves suggested by any chess piece. This single candidate will know what piece is being moved and its proposed position.

Discard candidates if they can be replaced by a shared role. To find common ground, you need to let go of the little details that make objects different in order to find more powerful concepts that can simplify your design.

What do books, CDs, and calendars have in common? If you are a business selling these items over the Internet, they have a lot in common. Sure, they are different, too. Books likely belong to their own category of items that can be searched and browsed. But all these kind of things share much in common. They all have a description (both visual and text), a set of classifications or search categories they belong to, an author, an availability, a price, and a discounted price. It sounds as if their common aspects are more important, from the Web application's perspective, than their differences. This suggests that all these different kinds of things could be represented by a single candidate, InventoryItem, that knows what kind of thing it is and the categories it belongs to.

Purely and simply, you gloss over minor differences. You don't need to include different candidates for each category of thing. In fact, those distinctions may not be as important to your software as they are to those who buy and use the items.

When you are shopping for items, you may be thinking of how they are used—books are read, calendars hung on a wall, and CDs played—but those distinctions are not important if you are designing software to sell them. Sure, you want to allow for your software to recognize what category something belongs to. You want to list all books together. But you probably want to categorize things in the same subcategory, whether or not they are the same kind of thing. Books about jazz and jazz CDs are in the "jazz items" category.

Only if objects in different categories behave differently in your software do you need to keep different categories as distinct candidates. The real test of whether a category adds value to a design is whether it can define common responsibilities for things that belong to it.

Blur distinctions. There are times when both concrete candidates and their shared role add value to a design. There are times when they do not. If you clearly see that candidates that share a common role have significantly different behavior, then keep them. Test whether the distinctions you have made are really necessary.

> What value is there in including different kinds of bank accounts, such as checking or savings accounts in our online banking application? Checking accounts, savings accounts, and money market accounts have different rates of interest, account numbering schemes, and daily account draw limits. But these distinctions aren't important to our online banking application. We pass transactions to the banking software to handle and let them adjust account balances. In fact, because our application is designed to support different banks, each with its own account numbering scheme, a distinction made on account type (checking or savings) isn't meaningful. Our application doesn't calculate interest. So we choose to include only BankAccount as a candidate. If we were designing backend banking software that calculated interest, our decision would be different.

Defend Candidates and Look for Others

For a candidate to stay in the running, you should be able to state why it is worth keeping, along with any ideas you want to explore:

> "A user accesses accounts to transfer funds, make payments, or view transaction history." In the next breath you can add, "Accounts contain information that enables a customer to perform financial transactions. Accounts know how to describe themselves; they know and adjust their balance; they are affected by different financial transactions; they know their transaction history. Are there any other candidates we should be identifying to support accounts in their role?"

Marvin Minsky theorizes about the many agents working at different levels during problem solving. Most people don't forget that they are packing a suitcase to go on a trip when they stop to fill a toiletry bag. Side excursions are a normal part of problem solving.

By taking short side excursions to look for more candidates, you will come back with a better sense of whether you are on target. You can find more candidates by looking for ways to support and complement the ones you've already found:

Potential candidates that complement and support Account:

AccountHistory—A record of transactions against an account

FinancialTransaction—An operation applied to one or more accounts. A service provider could represent each type of transaction that affects an account. There are multiple types of transactions that we support with our online banking application. What's the difference between a transaction that affects an account's balance, and an inquiry into some aspect of an account such as its balance, history, or activation status? How should we model each inquiry?

Searching can go on for quite a while if you are full of ideas. Stop when you feel you are looking too far afield. You need enough candidates so that you can compare and contrast them and to seed your further design work. There isn't any magic number. The more you know about a problem, the more candidates you are likely to invent in a first pass. Fifty candidates may seem like a lot, but it's not an unreasonable number. Twenty is OK, too. You find candidates in bursts as you consider your design's themes. It's pretty common for candidates to support more than one theme. All this means is that your objects fit into and support more than one central concern.

You are always free to decide all your candidates stink, toss them, and start over. At the beginning this is cheap and relatively painless. Defend candidates on their merits, and don't protect them from close scrutiny.

Stop brainstorming candidates when you run out of energy. Then review how these candidates might collectively support the responsibilities implied by a theme. When you think you have enough candidates, review them once more for their merit.

Keep any candidate and put it on the "A" list, for acceptable, when you can

- Give it a good name
- Define it
- Stereotype it
- See that it might be used in support of a particular use case
- See that it is an important architectural element
- Assign it one or two initial responsibilities
- Understand how other objects view it
- See that it is important
- Differentiate it from similar candidates

Discard a candidate when it

- Has responsibilities that overlap with those of other candidates that you like better
- Seems vague
- Appears to be outside your system's boundaries
- Doesn't add value
- Seems insignificant or too clever or too much for what you need to accomplish

You may still be uncertain about some candidates. Put these on the "D," or deferred, list to revisit later. For now, keep them in the running. The best way to make more progress is to design how these objects will work together. The very next step we'll take is to assign each candidate specific responsibilities. And during that activity, we will come up with more candidates and reshape those we've already found.

SUMMARY

You can approach the finding of objects somewhat systematically. Establish a framework for searching for candidates by writing a story about your application. In this story, write about the important aspects of your application. The candidates you identify should support various aspects of your story. You can use CRC cards to record your preliminary ideas about these candidates. CRC stands for candidates, responsibilities, collaborators.

Candidates generally represent work performed by your software, things your software affects, information, control and decision making, ways to structure and arrange groups of objects, and representations of things in the world that your software needs to know something about.

Good names for candidates are important. Choose them with care. Choose names that fit within a consistent naming scheme and aren't too limiting or overly specific. Once you've named and described each candidate's purpose, you can compare and contrast the candidates. For a candidate to stay in the running, you should be able to defend why it is worth keeping.

But your initial ideas are just educated guesses about the kinds of objects that you will need based on the nature of our application and the things that are critical to it. The real test of each candidate's

worth will be when you can assign it specific responsibilities and design it to collaborate with others.

FURTHER READING

Timothy Budd, in *An Introduction to Object-Oriented Programming* (Addison-Wesley, 2002), presents a thoughtful discussion of abstraction and object-oriented design. Another source of inspiration is Martin Fowler's *Analysis Patterns: Reusable Object Models* (Addison-Wesley, 1996). This book reveals how a good modeler and analyst thinks through issues and comes up with useful abstractions.