IEEE Software
www.computer.org/software

# Looking for Powerful Abstractions

**Rebecca J. Wirfs-Brock**

Vol. 23, No. 1
January/February 2006

Rebecca J. Wirfs-Brock ■ Wirfs-Brock Associates ■ rebecca@wirfs-brock.com

# Looking for Powerful Abstractions

**Rebecca J. Wirfs-Brock**

Betty Edwards, in *Drawing on the Artist Within* (Fireside Press, 1987), poses this thought experiment: Imagine we believed that only those endowed with an innate gift could learn to read. Teachers would think the best way to instruct would be to expose a child to lots of reading materials and then wait to see what happens. Fear of stifling creativity would dampen attempts to guide new readers. If a child asked how to read something, the teacher would say, "Try whatever you think works. Enjoy it and explore. Reading is fun!" Perhaps one or two in any class would possess that rare reading talent and spontaneously learn to read.

This is absurd. But if everyone believed that the ability to read was a rare and innate talent, no one would teach reading fundamentals.

Edwards claims that artistic talent only seems rare and out of the ordinary because we expect it to be. She believes that learning to draw "is simply a matter of learning basic perceptual skills—the special ways of seeing required for drawing." She also claims that "anyone can learn enough seeing skills to draw a good likeness of something seen 'out there' in the real world." She has devoted a lifetime to discovering effective techniques for teaching students how to "see" so that they can draw reasonable representations of real-world scenes.

In software, I've encountered similar biases, especially when it comes to designing object-oriented applications. Some claim that only innately talented designers can form good abstractions, develop a well-factored object design, or construct a domain model. The implication seems to be that design talent is rare and that only gifted designers are up to the more challenging task of creating good abstractions. Balderdash! Although design is a highly creative activity, we can still learn fundamental design skills—and accomplish a lot with them.

In this and future columns, I'll explore ways of seeing that have helped me become a better designer. Object technology is just one tool in a very rich toolbox, but since it's my area of expertise, I start by discussing the fundamental design skill of finding objects.

## Finding objects means finding good abstractions

Designers must be able to see the problem out there in the real world and solve it by applying the right tools and technologies. Like artists, object designers need to see in special ways to create good solutions—call this modeling or designing, if you will. However, since no two designers ever come up with identical objects, how can they agree on what makes good objects?

Early object design books showed how to pick out objects (noun phrases) by examining a requirements specification. When my colleagues and I did this in our book, *Designing Object-Oriented Software* (Prentice Hall, 1990), many

0740-7459/06/$20.00 © 2006 IEEE

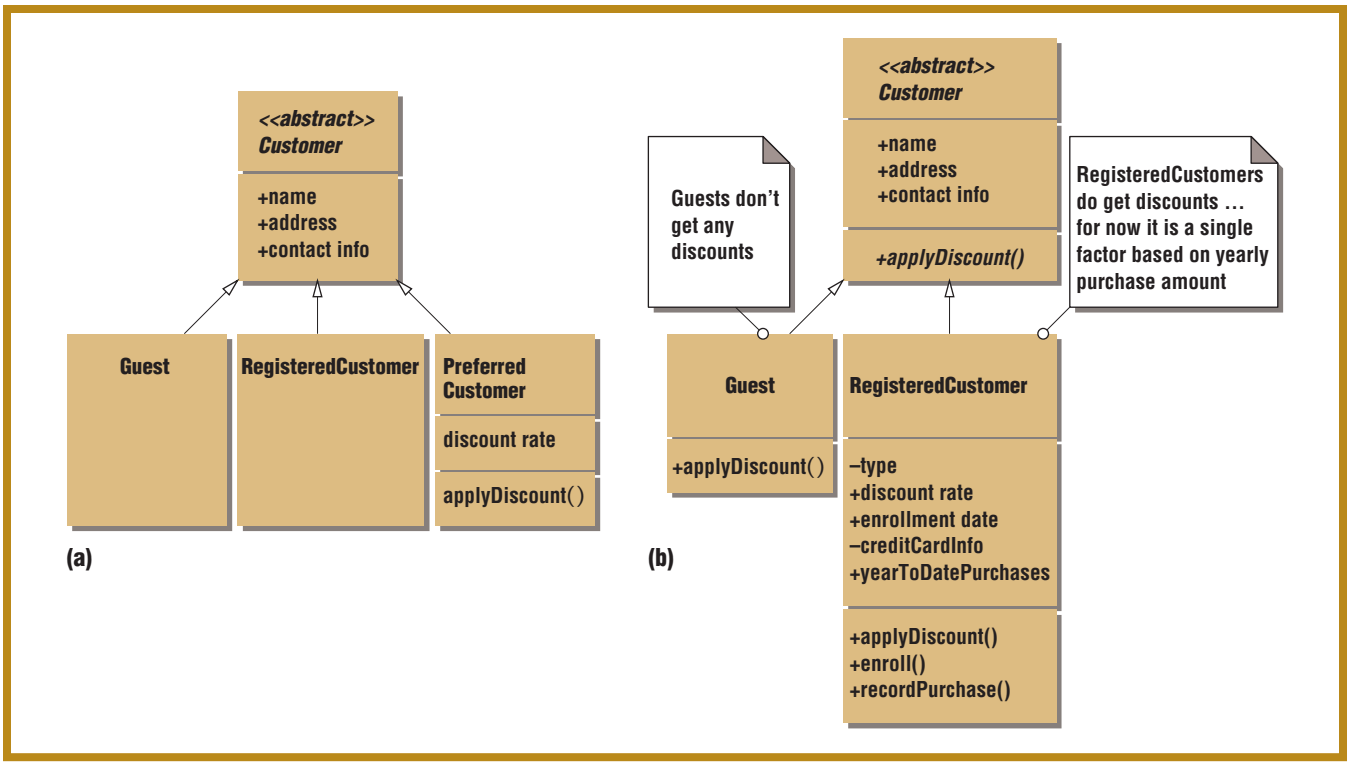


**Figure 1. (a) A simple design solution with one class per concept; (b) a preferred design, which separates RegisteredCustomer and Guest classes and defines a minimal Customer abstraction.**

mistook our example as suggesting the best way to find objects: write a problem description, scan it for nouns, sort through synonyms to come up with worthy class names, then apply a few heuristics to sort out duds. In fact, we only intended to demonstrate the typical thinking involved in coming up with a reasonable first cut at design objects.

In practice, I never underline nouns. More efficient ways exist to identify key object concepts. Some of these represent domain concepts—what Eric Evans calls in *Domain-Driven Design* (Addison-Wesley, 2004) the *ubiquitous language* that's structured around a software's domain. Written requirements, problem descriptions, and use cases are just a few sources for discovering this language. It's especially fruitful to talk with expert users about how they use your software. The idea is that some of the names these experts have for things and for some of the activities they perform might end up represented in your software as domain objects—with your design spin on them, of course.

In addition to domain objects, you'll need to find many other kinds of objects, and many other ways to identify potential design objects exist. My more recent design book, *Object Design: Roles, Responsibilities, and Collaborations* (Addison-Wesley, 2003), devotes a whole chapter to effective search strategies. Although some objects have a connection to your software's domain, many are pure inventions that represent abstractions of

- the work your system performs;
- things directly affected by or connected to your application (other software, physical machinery, hardware devices);
- information that your software uses or creates;
- decision-making, control, and coordination activities;
- structures and groups of objects; and
- representations of real-world things that your software needs to know about.

What's important when looking for these kinds of objects is to identify abstractions that form the basis of a well-factored design.

## Identifying good abstractions

Designing an object-oriented program is like organizing a community of individuals whose collective tasks achieve the community's larger goals. Finding good abstractions means identifying individual classes of objects whose behaviors mesh well with others. Each individual object should stand on its own and add value, but objects aren't designed in isolation. Designing software objects is more like scripting a movie than drawing a still-life portrait. Each character—er, object—should have a well-defined role to play.

### Start with candidates

It's best to start with candidates, not classes. In 1988, Ward Cunningham and Kent Beck invented CRC cards to help teach object-oriented concepts. The 4-by-6-inch index cards were a handy way to record initial design ideas about a *class* and its *responsibilities* and *collaborators*. In my current thinking, I still use CRC cards, but I look for *candidates* instead of classes because I want to give myself some wiggle room. When I first start thinking about a candidate, I'm not sure whether it will represent

- a common role that could be shared by many different classes of objects (which I'd eventually define an interface for),
- an abstract class that might form the base of an inheritance hierarchy, or
- a concrete class.

I want the flexibility to decide how to realize that candidate—once I think it will stick around.

### Consider the power of the abstractions

You need to justify candidates based on the power of the abstractions they represent. What makes a good abstraction? Economy of expression is important. Finding the right level of abstraction takes practice and experimentation. You might draw too many distinctions and create too many classes—a dull design that works but is tedious.

Consider this small design problem: a customer is registered on a system and has a unique name and password. A preferred customer is one who spent over $500 in the past year. Preferred customers receive special discounts on merchandise and special notices of sales. Guest customers can make purchases but aren't registered. How many different classes of objects should you define?

A simple design solution might model four classes—an abstract Customer class and three concrete classes: Guest, RegisteredCustomer, and PreferredCustomer (see figure 1a). A more compact design might represent these different customers in just one Customer class. We can get away with this if we encode what's common to all types of customers within that class. We also might want to define a customer-type attribute to constrain other customer attributes' values. However, it could get ugly if the Customer class defines many attributes that don't apply to Guests, so maybe we should distinguish between guests and other types of customers. Yet another design might model two different classes: RegisteredCustomer and Guest. Which design is better?

The first design encodes behavior differences within each class. If we decided to add a different customer type, we would likely create a new class. But should we? What if we offered a better discount rate to customers who spend over $2,000 each year? Is this worthy of another class—PremierCustomer? Or is this splitting hairs? All three designs work.

Given a choice, I prefer fewer, more powerful classes than many simpler ones. Proper encoding lets one class represent several different variations. But tricky encoding can lock you into premature abstractions. So, after thinking a bit, I'd probably settle on a design that models three different classes—one abstract class for Customer and two concrete classes for Guest and RegisteredCustomer (see figure 1b). Customer would define minimal common behavior shared by both RegisteredCustomer and Guest. I don't want to pack too much into that abstraction—only information and behavior that guests and registered customers have in common. Realistically, discounts could apply to RegisteredCustomers along many dimensions. Even starting very simply for now, I'd hide those details inside the implementation of the RegisteredCustomer class. I wouldn't fold an encoding of discount rate into the abstract class Customer unless I was certain that discounting rules wouldn't change over time.

### Model what's important to your software's behavior

Stop trying to model the real world. It's a myth that objects should accurately reflect real-world things. Even if we end up with domain concepts in our design, these objects are at best loosely connected to their real-world counterparts. Evans likens modeling a domain to making a movie. Even a documentary film doesn't show unedited real life; film-makers edit raw film to make a point. We software designers similarly shape objects to our purpose. Even if we're modeling things that have real-world counterparts—such as orders, customers, and shipments—we only model what's important to our software's behavior. And we invent other attributes and behaviors for these objects that aren't present in their real-world counterparts.

So, break away from thinking you need to capture objects from the real world. Instead, think about designing objects that work in your software's world.

### Consider differential behavior

Finally, look for the right level of abstraction based on differential behavior.

I remember a heated discussion at a "Finding Objects" birds-of-a-feather session at a recent OOPSLA (Object-Oriented Programming: Systems, Languages, and Applications) conference. One gentleman insisted that accurately modeling the real world was important and stated that if your application supports the sale of books, CDs, and electronics, it should represent these as distinct classes. Two developers who worked for one of the world's largest online retailers were also in the room. Fortunately, they emphatically stated that they designed a generic Resource class that defined characteristics and information for all items sold—whether they were books, calendars, CDs, or toasters. While the distinction between a book and CD is important to a purchaser, they didn't need to define different classes for these different types of things because they could treat them alike in their application. To identify a common shared abstraction—Resource—that represented all sold items, these designers let go of the little details that differentiated each item to focus on what they have in common.

Finding the right level of abstraction takes practice and experimentation. There are times when both concrete classes and their common abstraction add value to a design, and there are times when they don't. To find good classes, experienced designers make distinctions based on significant behavior differences.

**Rebecca J. Wirfs-Brock** is president of Wirfs-Brock Associates and an adjunct professor at Oregon Health & Science University. She is also a board member of the Agile Alliance. Contact her at rebecca@wirfs-brock.com; www.wirfs-brock.com.