IEEE Software
www.computer.org/software

Characterizing Classes

Rebecca J. Wirfs-Brock

Vol. 23, No. 2
March/April 2006

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

Characterizing Classes

Rebecca J. Wirfs-Brock

Quiddity n. 1. The real nature of a thing; the essence. 2. A hairsplitting distinction; a quibble.
—The American Heritage Dictionary

There are many ways to understand the nature of a class, but I start by looking at its name. *Aptronyms* are names that match a person's occupation—Joe Strong is a weight lifter; Suzie Snow is a ski instructor. Because I look at a class's name to suggest its role in a design, I expect class names to be aptronyms. For example, in Java, a StringTokenizer picks apart segments of a string, and the ClassLoader loads classes. But names aren't always illuminating. So I also scan a class for intention-revealing method names that suggest the class's behavior. Of course, the definitive source is always the code, but I shouldn't have to pore over details just to get the gist of a class.

In this column, I introduce several characteristics I ascribe to classes when trying to understand their nature and purpose. I hope you find these useful quiddities and not mere quibbles. (I'm both delighted by and distrustful of a word that has definitions with opposite meanings. The technical term for such a beast is autoantonym.)

Role stereotypes

Purposeful oversimplifications, or *role stereotypes* from Responsibility-Driven Design (see R.J. Wirfs-Brock and A. McKean, *Object Design: Roles, Responsibilities and Collaborations*, Addison-Wesley, 2003), help me identify and understand the work that objects do. Following are the stereotypes I find most useful:

- *Information holder*: an object designed to know certain information and provide that information to other objects.
- *Structurer*: an object that maintains relationships between objects and information about those relationships. Complex structurers might pool, collect, and maintain groups of many objects; simpler structurers maintain relationships between a few objects. An example of a generic structurer is a Java HashMap, which relates keys to values.
- *Service provider*: an object that performs specific work and offers services to others on demand.
- *Controller*: an object designed to make decisions and control a complex task.
- *Coordinator*: an object that doesn't make many decisions but, in a rote or mechanical way, delegates work to other objects. The Mediator pattern is one example.
- *Interfacer*: an object that transforms information or requests between distinct parts of a system. The edges of an application contain user-interfacer objects that interact with the user and external interfacer objects, which communicate with external systems. Interfacers also exist between subsystems. The Facade pattern is an example of a class designed to simplify interactions and limit clients' visibility of objects within a subsystem.

A well-defined object ideally supports a clearly defined purpose. Early in the design process, role stereotypes help me characterize initial candidate objects and communicate my

0740-7459/06/$20.00 © 2006 IEEE

ideas. Pushing on an object's role stereotype leads me to assign it some initial responsibilities. I can ask, "What requests should a service provider handle?" and then restate these requests as "doing" or "performing" responsibilities on a CRC (class, responsibilities, and collaborators) card.

Similarly, knowing what events a controller is handling, I can further describe how it responds to requests. I can also ask if it's doing too much and should delegate work to others, possibly reconsidering whether it should even be a controller in the first place. Reworking a design, I can turn an overbearing controller into a coordinator by redistributing its work to others. I can shape an object's character by refining its responsibilities, so periodically revisiting its role stereotype helps me better understand an object as it evolves throughout the design process.

Role stereotypes are also useful when taking a broad look at an existing design. What role stereotypes predominate? How do they interact? Are there recurring patterns and themes?

Following the design heuristic "make objects do something with what they know," experienced designers often blend role stereotypes to make certain objects more intelligent. Their designs often have

- information holders that compute;
- service providers that cache information, using it to improve performance or give clients more control over their operations; and
- carefully crafted structurers that answer intelligent questions about the objects they organize.

But role stereotypes aren't limited to designing new objects. I also use them to dissect design patterns. I gain deeper insight by recognizing the role stereotypes that pattern elements play. Aha! The Strategy pattern is just a private service provider; state objects in the State pattern are one way to delegate behavior out of a complex controller into a set of private service providers. Pattern authors might have given participants intention-revealing names, but the mental exercise of assigning role stereotypes to them adds zest to the sauce.

Uniqueness

In *Domain-Driven Design* (Addison-Wesley, 2004), Eric Evans distinguishes between *entity* and *value* objects. I often use these characterizations when modeling classes that represent domain concepts, but they are useful elsewhere as well. According to Evans, an entity object is distinguished by a thread of continuity and identity. An entity sometimes has an interesting life cycle, even changing in form and content. Consequently, you might need to add mechanisms to keep track of an entity's uniqueness and maintain continuity as it undergoes these changes.

One simple way to keep track of an entity is to assign it a unique key. A simplistic view of what that key should be might cause problems. Consider my recently stolen credit card. When the fraud detection group noticed my card being misused, they gave me a new credit card with a new ID number. My account with the old credit card didn't go away when they gave me the new card with a new ID. My legitimate purchases and credit history had to be associated with that new card and ID. This interesting wrinkle in the life cycle of a credit card account illustrates why a credit card number isn't a good way to uniquely identify my account, although it's associated with my account.

I gain deeper insight by recognizing the role stereotypes that pattern elements play. Aha! The Strategy Pattern is just a private service provider.

On the other hand, value objects typically represent another object's characteristics—for example, account status, date opened, and current balance. They need not be unique, and references to immutable value objects can be freely shared within an application. Some designers dismiss value classes as unimportant and not worthy of much design effort. Many times I hear impatient designers exclaim, "But it's just an attribute!" Sure, but whenever I can assign a couple of interesting behaviors or have an inkling that I might want some wiggle room to change the data structures that represent the values, I create distinct value classes. Using programming primitives—such as an integer or float data types, or pre-existing generic classes—eliminates the possibility of ascribing any application-specific behavior to these value objects.

Quantities

Thinking about value objects leads me to another handy characterization: *whole values*. Ward Cunningham first described whole-value objects in his Checks Pattern Language, described in *Pattern Languages of Programming Design* (Addison-Wesley, 1995). Whole values are a variant of value classes that represent meaningful quantities in a specific domain. Examples are classes that model color, speed, temperature, currency, time intervals, or trading dates. Whole-value objects encapsulate a value and unit of measure and often define behaviors for comparison, conversion, or calculation services on the basis of their values (for example, what's the next trading date?). Typically, whole-value classes are designed so that the values aren't changeable after creation. Designed carefully, whole-value classes can be small, useful design additions.

Use and impact

Classes aren't designed in isolation. Whenever I look at a class for "goodness" or design fit, I need to understand its larger design context. I like to characterize classes according to their relative importance and impact on a design. It's one thing to judge the quality of a class that extends an existing framework class.

It's quite another thing to assess a critical concept, key abstraction, or suite of classes that are designed to work together and provide a framework for extension.

UML stereotyping mechanisms can be used to annotate classes (as well as other model elements) with distinguishing characteristics. Scanning Tom Pender's *UML Bible* (John Wiley & Sons, 2004), I was intrigued to find *focus*, *auxiliary*, and *framework* on the list of standard UML 2.0 stereotypes. (UML stereotypes are bracketed by guillemets or French quotation marks, so technically I should have written «focus», «auxiliary», and «framework».) Each standard UML stereotype comes with a brief description: A focus class defines core logic or control behavior; an auxiliary class supports a more central one; and a framework stereotype tags a package that contains reusable classes or templates.

However, the UML 2.0 standard doesn't enumerate the definitive list of stereotypes. In fact, only a handful of UML class or classifier stereotypes exist in addition to the ones I've mentioned—and most focus on the level of abstraction that a class represents. By intent, UML stereotypes are extensible. Although predefined standard stereotypes hold sway among UML-informed designers, nothing prevents us from defining new stereotypes to suit our current design situation.

Notably missing from the official list were «active» and «abstract» stereotypes, which I often use when I draw classes. Technically, other visual cues can represent these stereotypes, but italic class names and bold (UML 1.4) or double-hatched classes (UML 2.0) that represent threads of control are too subtle for my poor eyesight and for many printers to accurately render. It helps to document what you mean by a stereotype—a sentence or two is a good start. Once I've done that, I can judiciously sprinkle these stereotypes into my design drawings and discussions. But I don't try to overwhelm my classes (or fellow designers) by tagging each class with a bevy of characterizations. If I can squeeze a characteristic into a class name, I don't repeat that characterization with a UML stereotype on a class diagram. Communicating the essence means saying or showing enough and stopping there.

Class characterization is a key step of object design. It serves two purposes—to clarify at a glance some important aspects of a class's expected behavior and to communicate its design intent to others. Depending on the type of systems you design and build, you'll naturally find certain characterizations to be more compelling than others. Recognizing and choosing effective characterizations is an essential skill that all class designers should master.

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.