IEEE Software
www.computer.org/software

Explaining Your Design

Rebecca J. Wirfs-Brock

Vol. 23, No. 6
November/December 2006

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

Explaining Your Design

Rebecca J. Wirfs-Brock

If a listener nods his head when you're explaining your program, wake him up. —Alan Perlis

Have you ever tried to explain some aspect of your design and not known where to start? Perhaps you had to present how you solved a problem or justify your chosen design among several alternatives, and you weren't sure how to highlight key design aspects critical in achieving a certain requirement. Or maybe you weren't sure how to explain parts of your design that didn't seem to make sense yet were required.

Design decisions with widespread impact or design nuances that might confuse new team members can benefit from good definitions and narrative explanations. When fellow designers repeatedly ask, "Why did you do it that way?" it's good to have an effective presentation that explains the tricky parts of your design without losing people in the details.

So where should you start?

Understand the fundamentals

If people can't understand the fundamentals influencing your design, they can easily get lost in your explanation. Benjamin Kovitz, in *Practical Software Requirements* (Manning, 1999), discusses an ideal sequence for presenting requirements: from most to least fundamental. Kovitz inspired me to create a similar list of the design-related items you should consider explaining when presenting a design (ordered from most to least fundamental).

Things you can't change

Laws of physics, regulations, or unmovable design parameters are the most fundamental items to explain. If your audience doesn't know these constraints, they likely won't appreciate how these constraints have shaped your design.

Problem descriptions and requirements

Stating the problem is more fundamental than stating your solution. Nonnegotiable behaviors such as recurring business cycles or time-based events can profoundly impact your design—even if they aren't physical laws. These, in addition to use cases or functional descriptions, constrain and prescribe behaviors that your design must support. If people don't have a deep understanding of how your system is supposed to behave, they certainly won't appreciate your design's subtleties.

Things

Things are more fundamental than the relations between them. People won't understand why two objects or components communicate until they understand each's responsibilities. They won't understand the details of a subclass's implementation until they understand how a superclass's structure and inheritable behavior is laid out. Nor will they understand why classes are linked until they understand something about each class's role, and they certainly won't understand an associative entity's purpose and content until they understand how the associated classes relate.

Quite simply, objects, components, or services are more fundamental than how they act in a particular situation or how they're used. However, although I've found that a design element's role and purpose are more fundamental than its particular behavior in specific situations, you can't completely understand how something works in isolation—it's best to ex-

 0740-7459/06/$20.00 © 2006 IEEE

plain the design in the context of its use. This has far-reaching implications for how you explain your design (and why people can't comprehend design nuances until you've explained both things and how they relate).

Organizing structures

A system's overall shape and structure is more fundamental than any details that knit components, services, and subsystems together.

The typical case

The parts of your design that support "happy day" use cases or "normal operations" are more fundamental than the parts that handle myriad exception conditions.

Concrete examples

Although designers have been trained to create abstract concepts, explaining such concepts can be difficult. Illustrating abstractions with concrete examples lets you effectively communicate complex concepts.

I often sketch object diagrams that illustrate the subtleties of a fairly complex (and more abstract) class diagram, and I place notes on concretely drawn sequence diagrams to mark where design variations are supported. I'm also keen on using role names (proceeded by "/" in UML) to indicate abstract concepts, and I often include both an object's class and role name (designated by "/rolename:classname") on diagrams, especially when trying to tie a concrete example to a more general abstraction.

However, even if an audience understands a concrete example, they might not understand the abstraction it illustrates. So, although it's important to present a concrete pattern example to explain a pattern, it's equally important to separate that example's potentially distracting implementation details and limitations from the more general potential of that pattern abstraction.

Design principles

Your final design is more fundamental than the principles you applied to find an acceptable solution. The reasoning behind how you divided responsibilities among objects, chose to represent domain concepts, favored the use of inheritance or composition in certain situations, configured collaborators to know each other, or defined interfaces is generally based on a few design principles. If you articulate the criteria you applied when you made a particular design choice, don't expect everyone to immediately follow or appreciate your reasoning.

Plot your storyline

Kovitz admits that achieving an ideal sequence—in which every explanation precedes its use in the following descriptions—is extremely difficult. Even if you could organize a design story that way, it might make for a dull, pedantic presentation. The key to solving this dilemma is to introduce new ideas or aspects of your design in a way that logically builds on the previous elements you've presented. You should minimize confusion due to one bit of information logically depending on something presented later, but you can't always avoid forward references.

When plotting how to explain a design, consider how best to convey key ideas and decisions and what interactions or feedback you'd like. During early design stages, explanations tend to be sketchier and less precise. Often your goal is to communicate initial ideas in enough detail to receive constructive feedback. So a presentation that demonstrates how a core set of objects interact and what roles these objects play seems reasonable. In this case, I'd be satisfied with a roughly drawn sequence diagram (perhaps with less than fully formed message names and arguments) and a stack of CRC (class, responsibilities, and collaborators) cards listing object responsibilities. Such a presentation's tone would be casual but informative.

If your goal is to educate newcomers about your design, you should explain important fundamentals first. However, it's often inappropriate to present your design in this way to a group of seasoned veterans. If you're aware of your audience's background, you can tailor your explanations accordingly. Even if your audience understands fundamentals, you might still elect to reemphasize certain fundamentals at key points to justify your design decisions.

Deciding how much to tell (and at what level of detail) should be based on the state of your design, what your audience needs to know, what you think they already understand, and what you want to convey. Don't explain everything. How much territory your explanation covers and how comprehensively you cover it depend on your goals.

For example, you could give at least four different explanations for the same collaboration:

- To give an overview of participants while omitting interaction details, you could use a UML communication diagram.
- To explain the basic sequence of interactions, you could use a sequence diagram and do either a high-level presentation or go into more depth, showing the details of the specific actions you want to emphasize.
- To explain how objects react under exceptional conditions, although you could draw one or more exception-handling diagrams, you should probably just compactly summarize how your design handles each condition.
- To explain how to adapt a design by "plugging in" objects or components that support predefined roles, you could use one or more class and sequence diagrams, sample code, and a "how-to" discussion.

Choose your presentation format

Diagrams as well as word and presentation choices help set your presentation's tone. It can be formal, casual, educational, inspirational, or persuasive. Although not every situation warrants a polished exposition, most explanations benefit from clear, compelling storytelling. And a standard diagram isn't always the best way to explain things. Using words, pseudocode, code, Backus-Naur form grammar, decision tables, state tables, or pictures can be equally, if not more, effective.

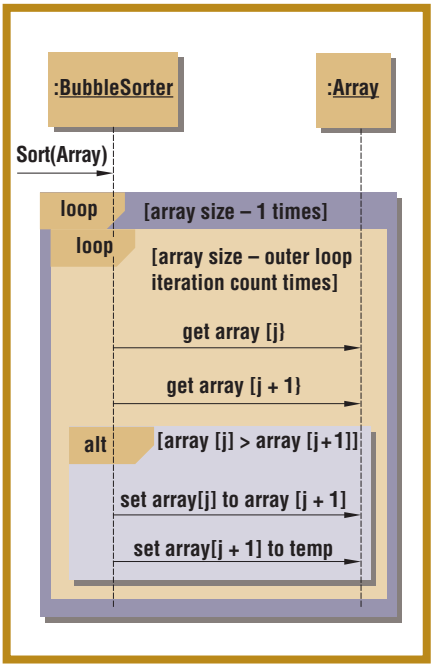

Figure 1. A sequence diagram illustrating a bubble sort.

Consider the array 42, 56, 13, 23
Let's start sorting…
42, 56, 13, 23 **no swap**
42, **56, 13**, 23 **swap**
42, 13, **56, 23** **swap**—end of first pass outer loop
42, 13, 23, 56 **swap**
13, **42, 23**, 56 **swap**—end of second pass outer loop
13, 23, 42, 56 **no swap**—end of third pass

Figure 2. An example of a bubble sort. (Blue indicates the elements being considered.)

On the first day of an object design class I teach to working professionals, I assign a pattern to each team to read, discuss, and present. Before presenting their pattern, the students have little exposure to key object and design concepts and UML. I don't prescribe how they should present their patterns, because I'm curious about whether they've picked up on the terms, vocabulary, and techniques I've briefly introduced.

Most students choose to show off newfound UML skills and illustrate their pattern with both a class and sequence diagram (their lack of UML drawing skills often makes for imprecise illustrations). Occasionally, a team explains a pattern by putting on a skit.

My favorite skit was a group presenting the Memento pattern—each student played an object role. The biggest guy on the team, playing the memento, demonstrated saving the originator's state by sizing up the student playing that role, then pretending to record pertinent measurements on a piece of paper. The memento then walked outside the classroom, refusing to divulge his contents to any who asked. When the caretaker wanted to restore the originator's state, he opened the door, called for the memento, and then passed it along to the originator (who huddled to confer with the memento to restore his state). Although they also presented a class and a sequence diagram, those seemed incidental to their story. Their skit was effective and led to a thoughtful discussion of which object should initiate a save-and-restore sequence.

Yet I keenly remember another situation where a skit fell flat. No one understood the pattern, and I had to step in, hastily drawing some diagrams as I talked through the pattern. At the next break, the students' manager, who was also in the class, asked me whether he should crack down and force his employees to use UML to explain their designs. I acknowledged his concern and pointed out that we'd have plenty of opportunities to hone UML skills over the next few days. I also pointed out that the exercise's main goal was for teams to effectively communicate a pattern's intent and how it worked—in whatever form they thought would be effective. The manager expected a more formal, detailed, and thoughtful presentation. To him, UML embodied good, clear design explanations. However, although UML diagrams can certainly help illustrate a pattern, the key to understanding a pattern is the accompanying verbal explanation.

Therein lays the heart of storytelling: explaining enough so that your audience understands your design and its nuances. I don't expect diagrams to convey every bit of information I need to impart. In fact, diagrams are rarely good at conveying technical minutiae, algorithmic details, or complex behaviors.

For example, consider how to explain a bubble sort. A bubble-sort algorithm consists of two nested loops. The inner loop traverses the array, comparing adjacent entries and swapping them if appropriate, while the outer loop causes the inner loop to make repeated passes. Figure 1 shows a bubble-sort sequence diagram, and figure 2 illustrates the bubble sort with a concrete animation.

When pressed to state a preference for how to understand a bubble sort, most people prefer a verbal explanation accompanied by an animation. Surprisingly, many find a textual description confusing and consider the code too detailed. Most also find a sequence diagram incomprehensible. Once, while presenting the bubble-sort algorithm at a conference, a designer asked me why I didn't use a UML activity diagram. So, he drew one, and he found it illuminating. I had to carefully study it to determine what decisions caused branching and what actions were being performed. The control flow didn't jump out at me until we laid out the actions and decisions using indentation that mimicked the nested code loops.

There's an art to communicating as much information as possible given a particular medium. No one picture or diagram can explain everything. As you tell stories about your design, expect to explain the same concepts at differing levels of detail and to present multiple perspectives. Even if you logically order your presentation, you'll still have to add color commentary to explain complex concepts. No one ever said that communicating design concepts was easy! However, if you plan each design presentation to meet your audience's needs, you should be able to keep them awake.

Rebecca J. Wirfs-Brock is president of Wirfs-Brock Associates and an adjunct professor at Oregon Health & Science University. Contact her at rebecca@wirfs-brock.com.