IEEE Software
www.computer.org/software

Creating Sustainable Designs

Rebecca J. Wirfs-Brock

Vol. 26, No. 3
May/June 2009

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

Creating Sustainable Designs

Rebecca J. Wirfs-Brock

To live only for some future goal is shallow. It's the sides of the mountain that sustain life, not the top.
—Robert M. Pirsig

We value code that's comfortable—where everything fits neatly in place, contributing to its familiarity and ease of understanding. Richard Gabriel, in *Patterns of Software* (Oxford Univ. Press, 1996), describes software where "developers can feel at home, [and] place their hands on any item without having to think deeply about where it is." He calls such software *habitable*. So how should we go about creating habitable software? Should we just place our trust in really good software developers, or are there specific design qualities and practices that we should be paying more attention to?

Christopher Alexander, whose work inspired the software pattern movement, argues in *The Nature of Order: The Phenomenon of Life* (Center for Environmental Structure, 2004) that both natural and designed structures have a kind of life. Such living things are characterized by 15 properties:

- levels of scale,
- strong centers,
- boundaries,
- alternating repetition,
- positive space,
- good shape,
- local symmetries,
- deep interlock and ambiguity,
- contrast,
- gradients,
- roughness,
- echoes,
- the void,
- simplicity and inner calm, and
- not-separateness.

The more life a thing has, the more pleasant it is to live with, use, or dwell within. Perhaps habitable software should also exhibit some of these life-generating properties.

Centers, Scale, and Proportion

The notion of centers is fundamental to Alexander's ideas of well-designed things. In short, whatever draws your attention is a center. You can find centers in the pleasing geometry of interlocking tiles or the arrangement of rooms around an entranceway. Complex structures often consist of interlocking centers of differing sizes. For example, the south-facing windows in my sunroom constitute multiple centers, with the largest window surrounded by smaller windows above and to the side, forming a three-by-three grid.

Not every design element should be the same size or shape. Alexander claims that a good design has differing levels of scale, whereas ugly, lifeless designs don't take into account the interplay between design elements at different levels of scale. He cites a Josef Albers painting that contains three nested squares (see www.artquotes.net/masters/josef-albers/homage-to-the-square-63.jpg for a similar Albers work) as an example of poor levels of scale. Although the three squares in many of

0740-7459/09/$25.00 © 2009 IEEE

Albers' paintings are different sizes, they're too close in size. They're all roughly at the same level of scale. A better, more pleasing design would ensure that jumps between centers at different scales aren't too great or too small—say, approximately a 2- to 4-times jump in size between levels.

Once you're aware of physical centers, they're easy to spot. But what corresponds to centers in software? Jim Coplien, in "Space: The Final Frontier" (*C++ Report*, Mar. 1998, pp. 11–17), suggests we "look to the code for pleasing geometry and shape." Certainly, a properly nested function has a pleasing shape. I see centers in classes, components, software systems, and subassemblies as well. Different-sized centers are all around if you know how to look for them. Coplien even views design patterns and pattern languages as "stereotypical configurations of centers, centers that have specific relevance in a particular domain."

When I'm stymied in trying to comprehend a complex class diagram, I can't help but wonder whether it's because I can't find meaningful centers to lock onto. Every class is the same shape and relative size. If I could only locate the central classes, then I could explore their relationships to other meaningful centers. I'm sure strong classes exist, but a class diagram representation doesn't help them stand out.

Knowing about centers is one thing, but paying attention to their shape, relationship, and proportion to others is what improves a design. A good design strives to create harmony between elements at different levels of scale. So when designing a class, we should consider whether it's proportional to its role and whether it fits into the context of other classes it organizes, interacts with, or extends. At a lower level of scale, individual methods shouldn't be too big. At the next higher level, we don't want to pack too much behavior into any subsystem or component. Even being aware of these levels, I don't have a good feel for whether pleasing ratios for software centers at different levels have any correspondence to those ratios Alexander ascribes to physical centers.

Fortunately, to get a grasp of our software's overall shape, we can express our design of components, classes, and systems at different abstraction levels. Martin Fowler observes in *Analysis Patterns: Reusable Object Models* (Addison-Wesley, 1997) that we can talk about software objects at three levels:

- a conceptual level, where we speak of a class's responsibilities;
- a specification level of operations, attributes, and test specifications; and
- an implementation level of class, method, and variable definitions.

We don't always have to view our designs at the most detailed level. In fact, when we move between abstraction levels, we gain a better perspective.

Designing Strong Centers

At the class level, clearly defined roles make for strong centers. Domain objects also form centers. A domain concept's strength is based on its fit to the problem at hand. You know a domain concept is weak when too much work is shoved onto other objects that interact with it. In *Domain-Driven Design* (Addison-Wesley, 2003), Eric Evans gives guidance on how to shore up boundaries between different domains by defining translation layers, using "anti-corruption" mechanisms, and employing strategies for stylized domain-entity access. Using these patterns contributes to improved encapsulation and strengthened boundaries between domains.

Networks of collaborating objects form centers, too. Centers are strengthened by boundaries that surround, enclose, separate, and connect them to other centers. In software, we value encapsulation because it helps us manage complexity. Interfaces clarify the boundaries between classes or components and the services they offer. Specifying contractual agreements strengthens and formalizes the connections between collaborators. When we define contracts using mechanisms that pull out contract-enforcing logic into separate contract specifications (as in the Eiffel programming language or with aspects), the method's purpose becomes more evident and stronger.

Class hierarchies are strengthened by agreements between abstract classes and subclasses on which methods and variables are visible (and which are hidden), which behaviors can be extended and by what means, and which invariants must be preserved.

Programming languages don't always provide good support for clarifying these agreements, so we fall back on applying principles such as the Open-Closed principle (Bertrand Meyer, *Object Oriented Software Construction*, Prentice Hall, 1988) or reasoning about the intended agreements. One reason inheritance has fallen out of favor is the mental effort required to deduce these agreements. Defining contracts might strengthen class definitions, but formal, rigid contract definitions aren't always appropriate. When I'm doing exploratory design, I want to be able to freely alter a class's behavior. An overly rigid class definition will cause me to "bend" my design to use it.

Centers are strengthened by repetition. Repetition also helps us become familiar with a design and gain confidence in our abilities to extend it. Stylized, repeated collaboration patterns improve the design of a software control's center. Consistently repeating the ways we validate information and requests improves our software's reliability. Although frameworks impose necessary (and good) repetition, doggedly selecting and applying design patterns doesn't guarantee design goodness. Alexander cautions against banal repetition.

Alexander defines positive space as "when every space is substantial in itself, never the leftover from an adjacent shape." We appreciate classes and components that aren't tangled with unnecessary behaviors or interdependencies. I've been involved in many arguments about how best to refactor a design into smaller constituent parts. We spend time debating whether a

Although frameworks impose necessary (and good) repetition, doggedly selecting and applying design patterns doesn't guarantee design goodness.

proposed reassignment of responsibilities is pleasing or "balanced." Alexander asserts that "of all the properties … [positive space] is probably the most simple and the most essential, since it guarantees to every part of space the status of being a relatively strong center." Sometimes our disagreements are more about style than substance, but I vow not to so readily dismiss these discussions as irrelevant.

Symmetry also contributes to design familiarity, coherence, and understanding. In the name of symmetry, we refactor a lengthy method into roughly equivalent steps implemented by smaller helper methods. Yet we shouldn't apply symmetry-producing choices without thinking through their consequences. It's naïve to define getters and setters for all attributes. We must consider how an object should be used. We look to define symmetrical behavior where it fits—defining a do() operation should lead us to consider undo(). Our decision to add symmetrical operations is based on our notion of how a design element should work and fit into a larger scheme of things. That's why any thoughtful design has a degree of roughness or irregularity.

A center's strength can be increased when it's attached to a nearby strong center, through a design element that seemingly belongs to both. In the physical world, an obvious example is a tongue-and-groove joint. The interlock between centers can also form another center that can be a focus of design effort. That's one reason we define interface definitions and service contracts.

Process Matters

Alexander is an ardent advocate of emergent design. He claims that living structures, both biological and human designed, are brought about by a sequence of structure-preserving transformations. The design and construction processes are more important and larger in their design effect than any designer's ability or training.

Alexander asks us to think of the properties of living things not merely as static characteristics of a design but as names of particular kinds of structural transformations. For example, a levels-of-scale transformation might introduce intermediate-sized centers to fill out the hierarchy of scales in a design. A boundary transformation creates further distinctions between a center and its surrounding area. A simplicity transformation cleans up a design by removing unwanted centers, gratuitous differences, or other complexities.

These activities have obvious and direct connections with incremental, iterative software design. As our ideas evolve, designs are refactored and code is reshaped and transformed. So rather than looking for complex design tools with the hope of creating the ultimate design, we should continue to seek out practices, techniques, and tools that support a sustainable software design process and adaptable, habitable designs.

Rebecca J. Wirfs-Brock is president of Wirfs-Brock Associates. Contact her at rebecca@wirfs-brock.com; www.wirfs-brock.com.