Casting Software Design in the Function-Behavior-Structure Framework

Philippe Kruchten, *University of British Columbia*

Design is far more widespread and encompassing than what we usually call *software design*. The Function-Behavior-Structure framework can help you and your teammates speak the same language.

What is software design? Most of you probably take it for granted, as I did until I was challenged by colleagues in other professions and started to wonder.

The Australian design scientist John Gero and his colleagues have developed a framework—the Function-Behavior-Structure framework—applicable to any engineering discipline, for reasoning about and explaining the nature and process of design.[1–3] In this article, I cast the software engineering process into the FBS framework and thus into the broader framework of engineering design. By doing so, we can draw some lessons about the state of our favorite engineering discipline. The most important lesson might be that many of the analogies we've drawn from other engineering disciplines, especially civil engineering, are somewhat flawed or biased.

What is design?

Derived from the ancient Greek worldview and culminating with the theories of philosophers such as Descartes, science—particularly analysis—regards the world as a sea of facts and ideas to be explained and understood. In contrast, design and engineering disciplines consider the world inadequate and needing repair. They seek to change and improve it by creating new artifacts:

> *The engineer, and more generally the designer, is concerned with how things ought to be—how they ought to be in order to attain goals, and to function … With goals and "oughts" we also introduce into the picture the dichotomy between normative and descriptive. Natural science has found a way to exclude the normative and to concern itself solely with how things are.*[4]

The act of design is a "goal-oriented, constrained, decision-making, exploration, and learning activity which operates within a context which depends on the designer's perception of the context."[1] The designer explores and plays with numerous variables, and the context shifts as the designer's perceptions change and the designer becomes more aware of the design process.

This article is not about design models, but about models of design. From Leonardo da

0740-7459/05/$20.00 © 2005 IEEE

Vinci ("disegno") and Giambattista Vico ("ingenio") to Nobel Prize laureate Herbert Simon, designers and philosophers have examined the act of design in various domains. I found the FBS framework particularly useful for reflecting on how we design software products. My first discovery was that software developers' use of the term *design* differs from that of engineers in other disciplines, making some of our comparisons and analogies somewhat skewed or simply invalid. The limits of software science hurt us in that we lack the proper tools to reason about our designs; however, we can exploit software's "soft" nature to our advantage.

A general framework for engineering

In the FBS framework, eight processes link a set of five elements (see Figure 1). The framework aims to analyze how engineers design—that is, the processes and intermediate artifacts we produce.

FBS elements

Engineering design aims to transform a set F of *functions* into a *design description* D such that an artifact conforming to D (manufactured according to D, for example) will fulfill the functions in F. Say, for example, we want a window that gives us daylight, ventilation control, access to a view, and so on. Naively, we could think of designing as the transformation F → D. Then, D is all the information a manufacturer needs to build a given window.

But we haven't found such a straightforward route from the set of required functions to the design description. The design description consists of elementary artifacts and their relationships, described in a *structure* S. S is the structured set of existing design decisions for our target object. We can derive some of D from S, expressed in terms of the elements of S—that is, S → D.

Thus, in the window example, the structure includes glass panes, frames, hinges, and handles. D specifies which part to pick, what material to use, the window's dimensions, and so on.

F → S expresses another model of designing. This is nothing more than a sort of catalog lookup, where we find a structure associated to a certain function. It's not really much of a creation, and unless we're lucky enough to have a complete catalog of all structures for all functions, we aren't there yet. For windows, we might be lucky: We could find one ready-made that matches our needs.

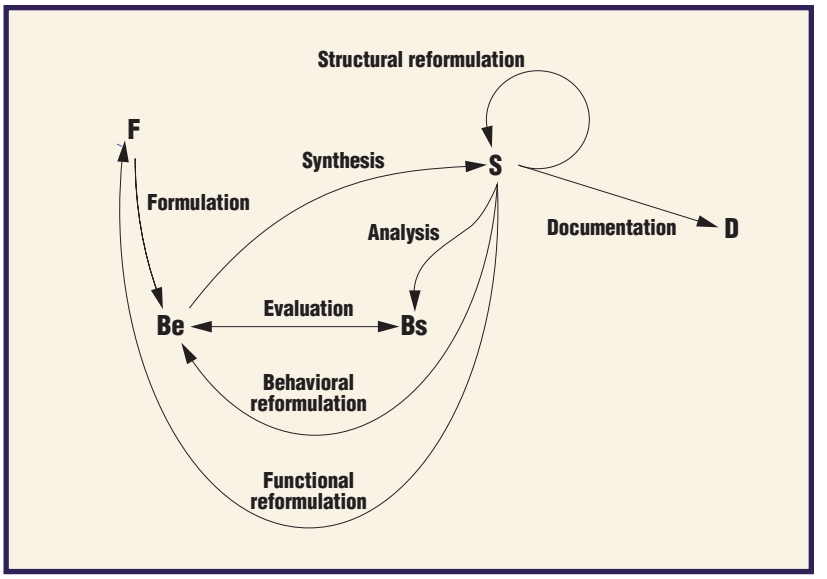

Figure 1. John Gero's Function-Behavior-Structure framework, in which eight processes link a set of five elements.[1] (figure courtesy of John Gero)

The structure S exhibits a *set of behaviors* Bs, which we can derive from S through analysis or experimentation: S → Bs. For example, a certain type of glass pane has certain properties in light transmission, heat transmission, weight, impact resistance, and cost. These behaviors (Bs) might not be quite what we expect; rather, we expect a set Be of *expected behaviors* to fulfill F. In fact, we tend to reformulate the functions in F in terms of expected behavior: F → Be. In some sense, Be will be a precise formulation or specification of the window we want: this amount of light, and this level of insulation, for this maximal price.

This leaves us with a tension between Be and Bs. We'll need to evaluate this tension, Be ↔ Bs, and revisit our assumptions about F, Be, S, and D. So, a more complex model of design is F → Be, Be → S → Bs, concluded by the obvious S → D.

This whole process is one of synthesis. But Bs might differ from Be too much to be acceptable, leading to various reformulations of S, Be, F, and ultimately a very different D.

FBS processes

In the FBS framework, eight subprocesses connect the various elements:

- *Formulation* (F → Be) transforms the design problem, expressed in function (F), into behavior (Be), which is expected to enable F.
- *Synthesis* (Be → S) transforms the expected

Table 1

Artifact mapping from FBS to RUP

FBS element	Definition	RUP artifacts
F	Set of functions	Vision document Stakeholders' needs Use case survey Business case
S	Synthesized structure	Design models Code
Be	Set of expected behaviors	Requirements Use case model Supplementary specification Use case storyboard, UI prototypes Test cases
Bs	Set of actual behaviors	Test results Inspection and review reports Measurements
D	Description	Executable code (installers, etc.) Other deployment artifacts (data, user guide, training material, etc.)

behavior (Be) into a solution structure (S) that's intended to exhibit the desired behavior.

- *Analysis* (S → Bs) derives the actual behavior (Bs) from the synthesized structure (S).
- *Evaluation* (Be ↔ Bs) compares the behavior derived from structure (Bs) with the expected behavior (Be) to prepare the decision if the design solution is accepted.
- *Documentation* (S → D) produces the design description (D) for constructing or manufacturing the product. The manufacturer has no further design choices to make.
- *Structural reformulation* (S → S) addresses changes in the design state space in terms of structure variables or the variables' value ranges. Structure evolves and is refined over time.
- *Behavioral reformulation* (S → Be) addresses changes in the design state space in terms of behavior variables or the variables' value ranges. Expectations evolve, change, or are refined.
- *Functional reformulation* (S → F) addresses changes in the design state space in terms of function variables or the variables' value ranges. The needs are evolving.

Applying FBS to software engineering

Although Gero and his colleagues didn't have software in mind when developing their framework, we can map software engineering to it. To represent software engineering practice, I use the Rational Unified Process.

Mapping artifacts

I first map FBS elements to RUP artifacts (see Table 1).

- F corresponds to the stakeholders' needs, the vision, the services the system will provide (such as the use case survey), and the constraints, including financial ones, expressed in the business case.
- Be is the system's documented requirements: use case model and supplementary requirements, user interface prototypes, use case storyboards, and so on. (Yes, the detailed requirements are part of the design.)
- S is what we in software call the software system's design, including its architecture. It identifies the solution's elementary artifacts and their relationships. Therefore, in addition to our design models, S includes the programs, which are the continuation of the design at a finer granularity level. (Yes, the source code is part of the design.)
- Bs is the system's actual behaviors as derived from S—the design and code—and witnessed in terms of test results and other evaluations. It includes anything we can derive from S that would show us how close we are from meeting Be.
- D is the detailed blueprint that customers will use to instantiate, install, and create their software system. It's more than the software design and code—it's the gold master CD-ROM that engineering sends to production.

You might have noticed that to map software engineering (or design) to the FBS framework, I had to extend the boundary of "software design" to include much more than software practitioners' traditional activities, according to the Software Engineering Body of Knowledge (www.swebok.org). In SWEBOK, software design covers only a narrow set of processes and artifacts.[5] But if we accept that design is making choices that will shape the final product, we must include some requirements activities and all coding and testing activities. Formulating the problem and analyzing the requirements or code is "doing design"—making

choices that impact the ultimate system. However, software engineers traditionally define "design" more narrowly, as building a model of the system to be constructed up to the point at which coding can begin.[6,7]

Mapping processes

Similarly, we can map the RUP processes to the FBS framework's processes (see Table 2).

Formulation maps to RUP's business modeling and requirements disciplines. We formulate the functions into expected system behaviors. Some processes call this *requirements analysis*.

Synthesis is the RUP *analysis and design* discipline, and to some degree, the *implementation* discipline. The meaning of "analysis" in software engineering differs from Gero's, who uses it to look at the result of synthesis, not as a prelude. This process is where most software design occurs. Using our bag of tools and techniques, and with a good grasp of the requirements (Be), we invent a design (S).

Analysis includes all the processes we use to derive the actual system behavior implied by S. In software, analysis involves inspection, reviews, and mostly experimentation—that is, testing a running application, even a partial one (a prototype). Software engineers perform only a small fraction of analysis through reasoning based on S. This is the area of formal methods, schedulability analysis, queuing theory, and so on. Hopefully we'll get there some day, but other engineering disciplines are far more advanced than software engineering in this area because they have the laws of physics at their disposal and eons of experience behind them.[7]

Again, a significant gap exists between *analysis* in software, which maps to *synthesis* in FBS, and *analysis* in other engineering disciplines, which we could call *static software analysis*: the techniques and tools we can apply to our design models and code to derive some of their properties, correctness included, without having to finish the code and run it (for example, security analysis, dead code analysis, or conformance to coding standards).

Evaluation is the assessment of the gap between expected behaviors (the "specs") and observed behaviors (what we see running in the lab). Evaluation consists mostly of analyzing test results and static software analysis results, and partly of reviews and inspections. It corresponds to RUP's test discipline and some project management activities.

Table 2

Process map, from the FBS framework to RUP

FBS process	From → To	RUP process chunks
Formulation	F → Be	Business modeling Requirements definition
Synthesis	Be → S	Analysis and design Implementation
Analysis	S → Bs	Testing Review activities
Evaluation	Be ↔ Bs	Assessment activities
Documentation	S → D	Implementation Deployment
Structural reformulation	S → S	Refinement of design, code, refactoring Fixing defects in design and code
Behavioral reformulation	S → Be	Scope management Requirements change
Functional reformulation	S → F	Change in needs

Documentation is covered by the RUP deployment discipline: producing all the bits and pieces required for manufacturing and installing the system. Here also a major shift in the use of "documentation" occurs; the documentation process produces the detailed blueprints that let a manufacturer produce the software system. So we aren't speaking about what software developers casually call "software documentation," nor are we speaking about user documentation. The software documentation is in S in the form of design documents, models, and comments in the code. And if the manufacturer must produce user documentation, D must contain, for example, the PDF files to send to the print shop. Documentation activities are completing the bill of materials and preparing a master CD-ROM with the installers, binaries, data, help files, and other elements to be reproduced and delivered.

Structural reformulation is where development spends most of its time: evolving the design, expanding the code, refactoring, fixing the design, and fixing the code. Coding is simply a reformulation, at a more precise level, of the initial rough or high-level structure S (the software design). In Gero's terms, however, coding is part of design. Coding isn't a mere manufacturing process, as discussions about software construction might lead us to think (in the SWEBOK, for example). Software engi-

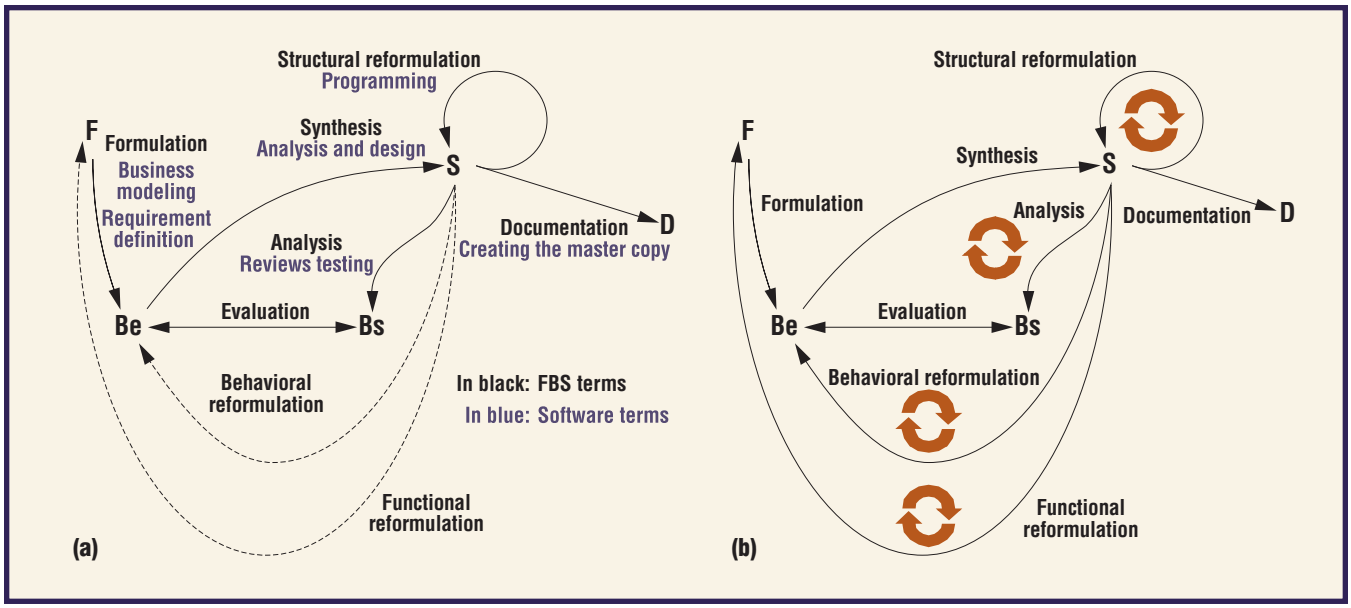

Figure 2. Software life cycles. (a) The waterfall lifecycle model (solid arrows) tries to go directly from F to D, with almost no loops or reformulations. Behavioral and functional reformations are strongly discouraged (dotted arrows). (b) Iterative development encourages many loops. Refactoring as the design emerges and encouraging customer involvement lead to constant reformulations (thick orange arrows).

neers make many design decisions while coding and refactoring.

As we know, other forms of reformulation occur, especially in iterative development. *Behavioral reformulation* entails adjusting the specification based on a design—for example, dropping or postponing a feature, changing a constraint, or introducing an unexpected "free" feature—and often takes the form of scope management.

And finally, possibilities found in the design influence users' expectations and desires for services, leading to *functional reformulation*. Often users want higher performance, and sometimes new functions arise. For example, because cell phones have screens and keyboards, we can play games with them, although this wasn't the original function. The agile practice of having customers on site during software development certainly encourages this process.

Software life cycles

The classic *waterfall lifecycle* process is a limited walk through the FBS framework graph, as Figure 2a shows. We expect synthesis to happen in one attempt, crossing our fingers for a short and successful analysis and evaluation in one shot. Structural reformulation is limited (just going from design to code), and behavioral and functional reformulations are strongly discouraged: "Sorry, this wasn't in the original requirement specification."

Iterative life cycles, illustrated in Figure 2b, exploit the loop more fully (synthesis, analysis, and evaluation). They break formulation and synthesis processes into small increments, gradually building the set of expected behaviors and leading to an evolutionary synthesis, supported by incremental evaluation. Iterative development lets us work on both sides of the evaluation process (Be to Bs), and not simply fix S to achieve the right Bs. We call this process *design emergence* and *refactoring*.

Iterative development and agile methods exploit the reformulation processes, in particular behavioral and structural reformulation. Involving customers throughout the development process by showing them intermediate versions of the structure results in changes in the set of functions or expected behaviors.

The evolution of software development life cycles, techniques, and tools for supporting the various processes led to this shift of emphasis from straight synthesis to loops and reformulations. Traditionally, software designers performed the synthesis process on paper, while coders and testers did the coding, analysis, and evaluation. Today, RUP and agile processes encourage a more seamless approach, with modeling closely tied to programming (*round-trip engineering*, as Rational Software used to call it), and exploit the very fast turnaround on the edit-compile-debug loop. Supported by tool automation, analysis and evaluation can continue repetitively, daily, hourly, relentlessly rerunning suites of regression tests.

Lessons from the FBS framework

Having mapped the FBS framework to software, we can exploit it further to reason about other aspects of software engineering.

Lack of fundamental theory

As noted in earlier work,[7] our lack of a fundamental software theory makes the evaluation process experimental. In fact, we deal with a very concrete S and almost nothing added to D, and can't rely much on static analysis. Hence, the RUP focuses on executable releases at each iteration to objectively assess the design.

But to our advantage, software is "soft," letting us perform rapid and low-cost loops on structural reformulation. Civil engineers can't do this when building a bridge—once they start using the description D to pour the concrete, they're committed—so they must validate absolutely everything in the analysis process.

Legacy systems

We don't always start at F to go to D. To evolve a legacy system, the process can follow the path D → S → Be → F. Or, we can start at D (the system), reverse-engineer it to a simple form of S (the design) from which we reverse-engineer Be (the specification), and from there move forward, with or without adding new functions to F. Other engineering disciplines sometimes do this as well, in particular in forensic activities following a major catastrophe such as a collapsed bridge or when upgrading very old structures.

Reuse

Reusing software components (or development based on COTS products or packages) shortens the cycle to F → Be → S, allowing the design to be more of the kind Gero called *catalog look-up*[8]: "for these enterprise functions, our ERP (Enterprise Resource Planning) system offers this and that design, just tick the boxes on your order form." But in software we need to look beyond the "Lego block" aspect and try to reuse the patterns and processes, not just the chunks of software.

Modeling

Software modeling can range from sketching design elements that support developer communication, to developing complete blueprints for implementers, to a form of programming where the code is almost entirely generated from a complete and detailed model. During specification or programming, modeling considerably enhances the analysis and evaluation processes of projects that use, for example, UML (Unified Modeling Language) and proper tool support. Modeling should also considerably simplify the description process by automatically producing the various deployment artifacts. Finally, we can use modeling to partially describe a system's expected behavior, and thereby facilitate the synthesis process by modeling the software requirements.

Patterns and design by analogy

As Gero and Lena Qian said, "Two designs are analogous if they have a similar function (F) or similar behavior (Bs). They may or may not have similar structures (S)."[2] However, the whole-pattern movement has shown how identifying classes of analogous designs to share (if possible) identical structures can benefit us: We can name and communicate design fragments and share and reuse practical solutions for recurrent problems. The key concept is "similar, but not identical."

What practical lessons can the FBS framework offer software engineers? First, we're doing more design than it might seem at first, and software design isn't limited to object-oriented design, UML models, or the structured analysis and design technique. How we elicit, capture, organize, and describe software requirements includes a good deal of design: we're making choices that will be reflected in the final product. Programming is primarily a design activity, and even testing is to some extent. Neither are purely administrative, clerical, or construction activities. Our design decisions shape the software product, and we have responsibility as designers, even though our titles and job descriptions don't make it clear that we're doing design.

Next, all engineering endeavors require some compromise to balance expected and actual behavior; we achieve these compromises by working on either end of the equation.

We can also see that because we've all but eliminated software "manufacturing," software production is almost solely a design activity.

Because we've all but eliminated software "manufacturing," software production is almost solely a design activity.

About the Author

Philippe Kruchten is a professor of software engineering at the University of British Columbia. His research interests include software architecture, software development processes, and software project management. He received his doctorate degree from l'École Nationale Supérieure des Télécommunications. He is a professional engineer in British Columbia, a senior member of the IEEE Computer Society, and author of *The Rational Unified Process—An Introduction* (3rd ed., Addison-Wesley, 2003). Contact him at Univ. of British Columbia, 2356 Main Mall, Vancouver, BC Canada V6T1Z4; kruchten@ieee.org.

In addition, we must be careful with our engineering analogies. When we speak to other engineers about "software construction," "software manufacturing," and "software factory," or when we say that "design is complete and implementation starts," we might not be well understood. Words like "design," "analysis," and "documentation" are overloaded, and their casual, unqualified use can lead to misunderstandings.

With a better grasp of the general engineering design pattern, software folks will better understand how their concerns and activities fit into larger systems-engineering projects that might involve computer hardware, machines, and even buildings—that is, it gives them ammunition to stand up to the big guys.

The key lessons, for me, are that the waterfall model isn't absolutely fundamental to engineering design, that iteration loops are inherent to all engineering design processes, and that in software we're more likely to exploit these loops because our object—software—can be more easily and continuously modified than, say, civil engineering objects.

There are lessons for researchers, too. Casting software design in the FBS framework exposes some of software engineering's strengths and weaknesses. Because software engineering relies mostly on experimentation rather than static analysis, our discipline needs to progress, particularly in finding approaches to describing software that easily translate to static analysis and code generation. This is actually the objective of model-driven development. Hopefully, MDD will close the gaps both upstream (using static analysis to match our design with the requirements without having to test) and downstream (producing code automatically—that is, correct by construction and hardly needing testing).

We can refine the model to dig further into the nature of software design. In particular, we can consider the structure as a structured set of design decisions, not just the structure of the system to build. We can also split the formulation, synthesis, and evaluation processes into two levels to isolate an architectural level consisting of architectural formulation, architectural synthesis, and architectural analysis.

Gero and his colleagues have since gone further with their framework and have added a reflective component to it, involving three nested worlds: the external world, the interpreted world, and the expected world. Food for further thought.

Acknowledgments

Many thanks to Mike Perrow, Joe Marasco, Walker Royce, Martin Fowler, and Catherine Southwood for their feedback on an early draft, as well as to Warren Harrison, Paul Stooper, and the *IEEE Software* anonymous reviewers.

References

1. J.S. Gero, "Design Prototypes: A Knowledge Representation Scheme for Design," *AI Magazine*, vol. 11, no. 4, 1990, pp. 26–36.
2. L. Qian and J.S. Gero, "Function-Behavior-Structure Paths and Their Role in Analogy-Based Design," *Artificial Intelligence for Eng., Design, Analysis and Manufacturing*, vol. 10, no. 4, Sept. 1996, pp. 289–312.
3. J.S. Gero and N. Kannengiesser, "The Situated Function-Behavior-Structure Framework," *Design Studies*, vol. 25, no. 4, 2004, pp. 373–391.
4. H. Simon, *The Sciences of the Artificial*, 3rd ed., MIT Press, 1996.
5. A. Abran et al., eds., *Guide to the Software Engineering Body of Knowledge*, IEEE CS Press, 2004.
6. J.W. Reeves, "What Is Software Design?" *C++ J.*, vol. 2, no. 2, 1992; available at www.bleading-edge.com/Publications/C++Journal/Cpjour2.htm.
7. P. Kruchten, "The Nature of Software—What's So Special about Software Engineering?" *Proc. Int'l Conf. Sciences of Design*, INSA, 2002.
8. J.S. Gero, "Conceptual Designing as a Sequence of Situated Acts," *Artificial Intelligence in Structural Engineering*, I. Smith, ed., Springer-Verlag, 1998, pp. 165–177.