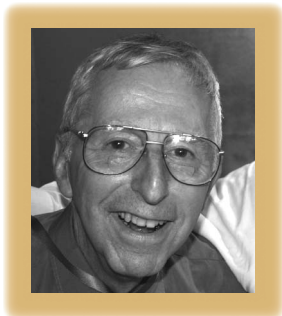


Why Context Matters— And What Can We Do about It?

Donald C. Gause

Every design problem begins with an effort to achieve fitness between two entities: the form in question and its context. The form is the solution to the problem; the context defines the problem.¹

In my software engineering interpretation of Christopher Alexander's statement, form is the final result of a software design and implementation effort—the delivered software product. Context, then, is everything else that's relevant to the given design problem, including the obvious, the subtle, the invisible, and the unknowable—the design's environment.



Because the software design environment contains limitless numbers of ill-defined factors, the key to our success is determining and incorporating “just the right number” of design-relevant factors.

The design environment

Each context-relevant class is a source of critical design ambiguity. We requirements analysts, designers, clients, and end users can avoid serious design errors by improving our visibility in each case.

The obvious

Every project includes environmental relevancies that are obvious to all interested parties, from end users to designers. The difficulty here is selecting the context issues for the design to address. The client and other users will likely expect the design to address many, if not all, of these issues, unless noted otherwise. The designers must meet cost, time, technology,

and other constraints and therefore selectively ignore all but the most important environmental factors. A brokerage firm that excluded each investment instrument's annual yield and yearly income estimate from its newly formatted monthly account statements serves as an actual case. The firm corrected this, but only after several months of complaints and a few lost customers.

A system scope statement—shared with and signed by all interested parties or their surrogates—addresses this issue. It's also important to thoroughly describe out-of-scope system limitations and the usual in-scope system features in the system requirements.²

One way to bring user viewpoints into the design process is to delegate user roles to the design team members. In this case, these members would sign important design documents for their surrogates as well as for themselves as designers.

The subtle

Other relevancies are obvious to some critical user constituencies but not to the designers. So, the subtle isn't likely to show up until the delivered system lacks functions that some constituencies are expecting.

This occurred recently in a financial system developed for giving strategic financial information to top management in real time. The system provided a strategic financial document to be presented to the monthly Board of Directors meeting. A remote data-entry mistake

Definitions

Functional requirements are the basic, essential functional transformations for the design team to implement. *Nonfunctional requirements* describe either modifications of these basic, essential functions or critical systems context factors. Functional requirements address more detailed issues just a step away from functional specifications, while nonfunctional requirements address higher-level issues (such as problem statements, users, attributes, limitations, or features).

Gee-whiz features are those features that are not just optional but outside the realm of the developers' imaginations. When developers discover and implement them, they can add substantial value to a product and differentiate it in competitive situations.

Context-free questions can apply to any design problem, independent of design discipline or context. Examples include asking the client or other interested parties, "What is the design solution really worth to you?" or "What have I forgotten to ask you?"¹

Reference

1. D.C. Gause and G.M. Weinberg, "Chapter 6: Context-Free Questions," *Exploring Requirements: Quality Before Design*, Dorset House, 1989, pp. 59–67.

running routine. The developers were embarrassed when the executives had to wait 20 minutes for the information. However, they were saved, this time, by the reminder that the presystem time for acquiring this same information was six weeks after each quarter closed.

We can enhance the odds of catching oversights by encouraging software engineers to introduce brainstorming and other ideation processes into requirements elicitation.³ Encouraging software engineers to incorporate practices, such as metaphorical thinking and idea sketching, into their normal work environment can also help.⁴ These mental tools are commonplace in other (hard) product design disciplines. In the Board of Directors example, brainstorming sessions created many features that weren't passed on to the developing group, once again illustrating the need to maintain level-to-level consistency of the full system requirements.

The unknowable

The unknowable relevancies consider those deep context factors that become visible primarily after product release. This includes the system's unanticipated impact on its environment as well as the environment's impact on the system. A multinational financial firm developed a system to improve customer loan services to global corporate clients. It designed the system to give loan officers the client loan status information they would need to answer queries and, if appropriate, offer tentative terms and advice. A post-release product review revealed only minor, fixable functional and usability issues. The review deemed the product a success with one nonfunctional (or extra-functional) exception. At marketing's request, the system tracked the phone-contact time for each loan officer and client conversation and issued monthly statistics to marketing, the loan officers, and their managers. Marketing had requested this information for demographic purposes only. These statistics, along with a recent organizational downsizing, had produced an unintended competitive situation that, in turn, made the

caused a substantial error in a widely used financial indicator. Fortunately, a staff member caught the error and corrected it before the meeting. The business requirements had specified out-of-range data checking, which would have flagged this error shortly after data entry. However, the system requirements didn't convey this requirement to the design and implementation organization, and therefore, it wasn't implemented. The design team, because of an aggressive development schedule, focused its attention on implementing the functional transformations required to perform the data-element to data-element calculations, or the *functional requirements*. Not enough effort went to the nonfunctional aspects, or the *nonfunctional requirements* (see the Definitions sidebar).

We can improve our chances of avoiding this kind of error by assigning explicit responsibility for monitoring conformance between business and system requirements. We as designers need to be more aware of the consequences of focusing too heavily on the functional specifications at the nonfunctional requirements' expense. Developing use scenarios and test cases early in the process and making them

an essential part of the system requirements also addresses this issue. Formal reviews, including system requirements' conformance to business requirements, are indispensable.

The invisible

Some parts of the environment are hard to see. We can ultimately recognize them, but they take effort to tease out. Imaginative, gee-whiz features are likely candidates for the invisible relevancies class, along with unincorporated and partially complete functions—functions that do *almost* everything required.

In the earlier example, the financial system's management requirements had numerous gee-whiz features. One feature applied to all queries requiring more than a 20-second response time. It required the long-running functions to provide well-specified "rationalization and pacification" information. Unfortunately, for reasons stated earlier, the functional requirements failed to include this gee-whiz feature, so the system didn't include it. Senior management viewed a demonstration a few weeks before the scheduled system turnover, and luckily, the managers asked for information requiring a long-


loan officers more abrupt with their clients. The system, designed to improve customer service, was actually degrading it. Once recognized, this design problem was easy to fix by aggregating the loan officer phone time while maintaining individual corporate client statistics.

Polling all parties early on unintended consequences is a useful way to raise consciousness and potentially discover these deep-context regions. Another tool that I've found especially useful is applying context-free questions. By their nature, such questions focus our attention on nonfunctional design aspects. We must continually remind ourselves that some relevant regions of context might reveal themselves only after product release. First-use monitoring of help functions, observing help-service activity, tracking user satisfaction before, during, and after delivery,⁵ and periodic post-release

reviews can provide useful avenues for discovering these hidden context issues and information for proactively managing design change activity.

Why does context matter? It matters because context, as I have used the term, is the fully relevant environment to which we're designing. Our understanding of this context defines our view of the design problem. Business, systems, and functional requirements—and, ultimately, functional specifications—are this contextual understanding's instantiation, which is our definition of the design problem at hand.

I've explored using several heuristics for getting a better look at relevant context issues and communicating these issues to all interested parties. The common factor to these heuristics is visibility. Anything that we can do to make our design thinking and processes

more visible will give us an improved look at relevant context issues, resulting in more complete design and better fit of form to context. 

References

1. C. Alexander, *Notes on the Synthesis of Form*, Harvard Univ. Press, 1979, pp. 15–16.
2. A. LaPlante, "Managing Project Scope," *Computerworld*, 20 Mar. 1995, p. 81.
3. D.C. Gause and B. Lawrence, "User Driven Design," *Software Testing & Quality Eng.*, vol. 1, no. 1, 1999, pp. 22–28.
4. D.C. Gause and G.M. Weinberg, "Part III: Exploring the Possibilities," *Exploring Requirements: Quality Before Design*, Dorset House, 1989, pp. 105–145.
5. D.C. Gause and G.M. Weinberg, "Chapter 21: Measuring Satisfaction," *Exploring Requirements: Quality Before Design*, Dorset House, 1989, pp. 238–248.

Donald C. Gause is a research professor of bioengineering in the Thomas J. Watson School of Engineering, Binghamton University, and principal of Savile Row, LLC. He teaches and consults in systems engineering, design processes, requirements engineering, and organizational innovation. Contact him at dgause@stny.rr.com.



RUG

Working at the frontiers of knowledge

The closing date for applications is **september 30, 2005**. Please send your written application, including a cv and a list of publications to: The University of Groningen Personnel & Organisation Department P.O. Box 72, 9700 AB Groningen The Netherlands Please state the vacancy number on the envelope and at the top of your letter.

Additional information about vacancies at the RUG is available on the university web site:

(www.rug.nl)

Faculty of Mathematics and Natural Sciences
Institute of Mathematics and Computing Science

• **Full Professor in Software Engineering**

VACANCY NUMBER 205152

The qualifications expected from candidates are described on the web site (www.math.rug.nl and www.cs.rug.nl) of the institute, under vacancies.

The University of Groningen can offer you a salary up to a maximum of € 6462 gross

per month for a full-time job (40 hours per week), depending on qualifications and work experience. Employment basis: permanent (tenure).

Further information can be obtained by prof. dr. D.K. Hammer, phone +31 50 3633941, e-mail: d.k.hammer@cs.rug.nl Websites: <http://www.rug.nl/informatica/onderzoek/programmas/SoftwareEngineering> or <http://www.rug.nl/informatica>



Rijksuniversiteit Groningen