

From Magic to Comet: A Case Study of Business Software Development

From December 2002 to June 2003 I participated in the design, development, and deployment of software for a mobile data card service provider in Tokyo, Japan (henceforth referred to as Company J). Mobile data cards can be plugged into laptop computers or PDAs to enable Internet access via cellular dial-up or, more recently, WiFi. A customer purchases both a card and a service plan from Company J. The type of card and the type of plan purchased determine which services the customer may access and how the customer is billed.

Company J started business in December 2001 and grew rapidly, so that by December of 2002 it had over 100,000 customers. Due in part to this rapid growth, the management of the company decided that the computer system for managing its services was inadequate and hired the company for which I worked (Company V) to develop a new system.

The legacy system, known as Magic, consisted of a single large database, which contained both historical data on cards and customers (used by staff to manage inventory, billing, and customer support) and real-time data (needed to support the systems which authenticated customers and granted access to services). There were several small pieces of software that had been written in-house to provide access to the database, but most access to the data was done manually by Company J's engineering staff.

Magic made the engineering staff a bottleneck when other departments wanted reports. Even worse, engineers making manual changes to the database could make mistakes that necessitated taking the database off-line. Since both historical and real-time data were combined in a single database, this meant that customers could not access services while problems were being fixed, resulting in lots of angry calls to customer support. Furthermore, any engineer could access sensitive data like credit card numbers—a significant security risk.

Company V proposed and built a new system, known as Comet, in which historical and real-time data were stored in separate databases to ensure maximum uptime for customers. Direct manual access to the databases was eliminated in favor of a browser-based user interface that combined the functionality of the several existing systems and also allowed the kinds of queries and changes that previously had been done by hand. It was decided that Comet would replace Magic in parts rather than all at once. The project began a couple of months before I joined the company, and two of these partial deployments were completed during my tenure there.

Although this project is still ongoing, the general consensus seems to be that it has been successful. Having been involved in a number of software development projects with varying degrees of success, I agree with this assessment. I will argue, however, that this success was achieved in spite of a failure to critically examine our design and development practices, and that such examination might have alleviated many of the problems we encountered during the project.

These problems, as reported by the participants in the project, fell into three categories: company politics, communication breakdown, and user acceptance. These are problems common to many software projects, and many developers accept them as inevitable. But as I will demonstrate, these issues are really just symptoms of deeper problems in the software

development process.

Engineers often complain about “politics” derailing otherwise smoothly-proceeding projects. Politics are assumed to lie outside the proper scope of software development, an irrational and unnecessary distraction from “real work.” Yet in truth politics are inherent to the process of building software. Bowker and Star refer to the “practical politics of classifying and standardizing.”¹ In software development, classification and standardization take the form of defining interfaces (both for users and other systems), developing metaphors for business entities and processes, deciding what data will be stored and how it will be stored, and so on. Classifying and standardizing, rather than writing and testing code, are the primary activities of software development.

These activities always require coordination among several groups of people. Generally, the backgrounds, goals, and power of these groups differ significantly. Negotiations among these groups exemplify Winner’s definition of politics: “arrangements of power and authority in human associations as well as the activities that take place within these arrangements.”² To understand the political nature of software development one must try to identify and describe these groups.

The relevant groups in this project were as follows: at Company J, the Operators, the Engineers, the IT Staff, the Marketers, and the Bosses. The Bosses were the decision-makers of the company, and the people responsible for hiring us. The Marketers were the people responsible for developing products and plans and selling them. The IT Staff were responsible for keeping the company’s computer systems in working order so that operations could proceed smoothly. The Engineers were responsible for developing the company’s internal computer systems (a role being partially relinquished to us). The Operators were the employees involved in procurement, customer support, and other departments who had to use the computer systems on a daily basis. The final two groups were the Managers and the Developers from Company V. The former communicated with the Bosses and were responsible for keeping the project on track and on time, while the latter were responsible for designing and implementing Comet. I was a member of this final group.

Although I have loosely based the relevant social groups on the various organizational units at each company, they were not isomorphic with these organizational units. For example, not every developer saw Comet in “Developer” terms. But it was the case that most employees in an organizational unit shared a certain set of meanings for Comet. Thus I have used the name of an organizational unit as shorthand for the group of people who shared these meanings, whether or not each person in that group actually belonged to that organizational unit.

Together, these groups had to achieve consensus on what Comet would do. This process of consensus-building is known as the “specification phase” in software development parlance. Due to their different backgrounds and goals, each group had a different idea about what constituted a working system, and the system presented each group with different problems. The Bosses wanted Comet to save the company time and money, to make the company more competitive, to provide better visibility of business data, and to allow greater flexibility in business planning. The Marketers wanted Comet to allow new kinds of products to be offered

¹ Geoffrey C. Bowker and Susan Leigh Star, *Sorting Things Out: Classification & Its Consequences* (Cambridge: MIT Press, 1999), 44.

² Langdon Winner, “Do Artifacts Have Politics?” in *The Whale and the Reactor: A Search for Limits in an Age of High Technology* (Chicago: The University of Chicago Press, 1986), 22.

and to enable these products to be rolled out more quickly. The IT Staff wanted Comet to be easy to manage and to not increase their workload. The Engineers wanted Comet to integrate easily with other in-house systems and to cut down on bugs and mistakes, but not to make their jobs unnecessary. The Operators wanted Comet to be easy to use and to make their jobs easier. The Managers wanted Comet to please the client, resulting in new business. The Developers wanted Comet to be well-written, well-tested, and easy to change.

Juggling these different interpretations to produce a satisfactory design involved a long series of meetings. The iterative process of negotiation and design during the specification phase is an attempt to achieve stabilization—but only certain groups have a voice in this process. As Winner points out, "In the processes by which structuring decisions are made, different people are situated differently and possess unequal degrees of power as well as unequal levels of awareness."³ This was certainly the case in this project. The Bosses, Managers, and Developers held a great deal of power, the Engineers and Marketers held less power, and the IT Staff and Operators held very little power at all.

Moreover, while the decisions made about Comet were reversible in theory, they were rarely reversed in practice. Although our development processes and tools allowed us to make quite radical changes to the system even at the later stages of development, the fact that we would charge handsomely for such changes meant that unless the primary decision-makers were willing to pay for it, "the original flexibility vanishe[d] for all practical purposes once the initial commitments [were] made."⁴

One commitment made early on that proved politically (if not technically) hard to reverse was the decision to identify customers by the phone numbers of their mobile data cards. While this is standard practice in the mobile phone industry (out of which the mobile data card business grew), it has certain disadvantages. Since cards are tightly linked to specific customers, those customers cannot share or re-sell their cards. Despite the fact that the Bosses and Marketers actually wanted to encourage the development of a resale market for their mobile data cards, they were unable to move beyond the "standard way of doing things" and choose a different way of identifying customers.

This was an example of "preexisting bias" rooted in "social institutions, practices, and attitudes."⁵ In this case there was an implicit industry bias toward associating customers with particular pieces of hardware, which precluded the development of a system in which cards were fungible and a customer was associated with some other kind of identifier, such as a username. As a result, Comet discriminated against, among others, corporate customers who may have wanted to have their employees share a pool of cards to cut down on the administrative costs involved in transferring cards.

The processes of "arriving at categories and standards, and, along the way, deciding what will be visible or invisible within the system"⁶ took a long time. Even once these decisions had been made, the developers still had the task of deciding how to express them in formalized languages. As Bowker and Star observe, "Even where everyone agrees on how classifications or

³ Winner, "Do Artifacts Have Politics?" 28-29.

⁴ Winner, "Do Artifacts Have Politics?" 29.

⁵ Batya Friedman and Helen Nissenbaum, "Bias in Computer Systems," in *Human Values and the Design of Computer Technology*, ed. Batya Friedman (Cambridge: Cambridge University Press, 1997), 26.

⁶ Bowker and Star, *Sorting Things Out*, 44.

standards should be established, there are often practical difficulties about how to craft them.”⁷ In this project these practical difficulties resulted in communication breakdown well after we thought the important design decisions had been settled.

This communication breakdown followed a pattern familiar to software developers everywhere: after a long series of meetings with the Managers and the Engineers in which all the specifications of Comet were hashed out, the Developers retreated to their cubicles and spent several weeks turning those specifications into a working prototype. But when the prototype was shown to the Engineers, it was discovered that something had gone wrong: the prototype was not what the Engineers wanted after all. Something had been lost during the process of formalization.

Agre defines formalization as “the recasting of human affairs in mathematical terms for purposes of implementation on computers.”⁸ Formalization eliminates “fuzzy” concepts and informal processes in favor of clear distinctions and algorithms. As Developers, we enjoyed this process of “making clear something that was formerly obscure.”⁹ Unfortunately, we failed to notice the gap we were creating between the reality of our customers' actual work practice and our formalized system.

This gap manifested itself most clearly in subsequent meetings to discuss problems with the prototype. We spent a lot of time arguing over terminology like “customer,” “individual,” “product,” and “plan.” When our clients would use these terms “loosely” or use alternate terms, we would insist that they use the “correct” terms and pay attention to their “proper” meanings. It seemed to us that our clients were being “sloppy” and that we had to constantly monitor their use of language. Our clients, of course, thought we were just being pedantic. Agre cuts to the heart of the problem: “Programmers and users will have great difficulty communicating about the problem because one is thinking and speaking in formalized terms and the other is not.”¹⁰

Software developers think and speak in formalized terms because the complexities of programming languages and interfaces demand it. As Bowker and Star point out, “When computer programmers write some lines of Java code, they move within conventional constraints.”¹¹ The formalizations we created had to mesh with many other formalizations created by other organizations, such as the various Java APIs created by Sun Microsystems¹², the Structured Query Language¹³ for querying and manipulating relational databases, and the RADIUS protocol for authenticating and authorizing dial-up users¹⁴. “Although it is possible to pull out a single classification scheme or standard for reference purposes, in reality none of them stand alone.”¹⁵

⁷ Bowker and Star, *Sorting Things Out*, 46.

⁸ Philip E. Agre, “Conceptions of the User in Computer Systems Design,” in *Social and Interactional Dimensions of Human-Computer Interfaces*, ed. Peter J. Thomas (Cambridge: Cambridge University Press, 1995), 75.

⁹ Agre, “Conceptions of the User,” 76.

¹⁰ Agre, “Conceptions of the User,” 79.

¹¹ Bowker and Star, *Sorting Things Out*, 39.

¹² For an overview of the various interlinked APIs, see “Java API Map,” *ONJava.com*, http://www.onjava.com/pub/a/onjava/api_map/.

¹³ International Organization for Standardization, “ISO/IEC 9075,” 1999.

¹⁴ RADIUS is actually a series of related protocols. See the list of RADIUS RFCs at <http://www.freeradius.org/rfc/> for more information.

¹⁵ Bowker and Star, *Sorting Things Out*, 38.

By focusing intensely on the formalization process and thinking solely in terms of Java classes and database tables, we could not see that we were making implicit decisions about things like what kinds of plans could be sold. In Magic, salesmen could customize the way important corporate customers were billed by talking to the engineers and improvising a system to make the numbers “work out.” Marketers could come up with an idea for a short-term pricing campaign and rely on the engineers to hack together a way to support it.

Our attempts to “formalize the non-formal”¹⁶ and the limitations of our programming languages, databases, and other tools constrained the types of products we could offer. This resulted in a technical bias against sales and marketing schemes that may have been viable had humans, rather than algorithms, been in a position to make the necessary judgments. In fact Magic had allowed some more complex sorts of plans to be sold, although the maintenance of these plans over time involved a lot of manual labor to update the database by hand. By replacing this manual intervention with software, Comet discriminated against these more complex products, to the detriment of the sales and marketing staff.

In retrospect, a weaker formalization may have alleviated some of the issues related to this divide. Agre suggests that the use of a “double-level language” that clearly distinguishes between the words that users use when talking about their work and the formalizations created by developers might be appropriate. This is an intriguing idea, yet one that is in direct conflict with prevailing approaches to software development, which stress unification of the domain model created by developers and the language used to discuss the problem domain with users.¹⁷ The result is the classic battle between developers and users in which “divergences between formalism and reality... [are] blamed on reality.”¹⁸

Just how far we had diverged from reality was not apparent until days before Comet was to be deployed for actual use. It was at this point that the actual users of the system encountered it. This encounter occurred much later than we had intended. We had been under the impression that we were engaged in participatory design of Comet with its eventual users. It was not until it was ready to be deployed that we realized that we had not yet even met the actual users. The team of engineers with whom we had been meeting had assumed the authority to speak for the actual users, who were regarded as mere operators with nothing to contribute to the design process. Thus Comet ended up being “used by a population with a different knowledge base from that assumed in the design,” a form of what Friedman and Nissenbaum call “emergent bias.”¹⁹ It is no surprise that the gap between the users for which the system was designed and the real users created difficulties for the latter.

Agre argues that participatory design practices in which the user is directly involved in the development process are needed to bridge the gap between engineers' formalizations and actual work practices. But although prevailing software development practices stress the importance of involving the customer in development,²⁰ the concept of a single generalized “customer” does not adequately reflect the reality of the multiple different groups involved in

¹⁶ Friedman and Nissenbaum, “Bias in Computer Systems,” 25.

¹⁷ Eric Evans, *Domain-Driven Design* (Boston: Addison-Wesley, 2003).

¹⁸ Agre, “Conceptions of the User,” 76.

¹⁹ Friedman and Nissenbaum, “Bias in Computer Systems,” 27.

²⁰ Don Wells, “The Customer is Always Available,” *Extreme Programming: A Gentle Introduction*, <http://extremeprogramming.org/rules/customer.html> (accessed December 7, 2003).

any real computer system. In this case, we generalized the “user” to include anyone at the client company, which turned out to be a mistake.

When the operators of Comet had their first training session, they angrily rejected the software, claiming that it was incomprehensible and too hard to use. We were shocked, given the time we had spent on specification. As Agre might have predicted, we dismissed the users' resistance as “irrational.”²¹ Had we been more familiar with patterns of user resistance we might have recognized that this rejection stemmed from the same divergence between formalization and actual work practice that caused the “endless series of complaints” we witnessed during the prototyping phase.²²

Despite the fact that we misidentified the actual users of Comet, we still embedded in the software a certain idea who the user was and what his duties were (and were not). By replacing direct intervention in the database with a software interface, we tightly constrained what users could do. This control was necessary for creating a secure and stable system with a minimum of mistakes, but it also created a “machine boundary” where none had existed before and prescribed certain ways of interacting with the system. In the terminology of Grint and Woolgar, we “configured the user,” specifying constraints on how the system could be used in order to control the relationship between the user and the system.²³

By standardizing work practices, we also defined which cases were “normal” and which were problematic. In Magic everything had been done with direct ad-hoc changes to the database, so in effect all cases (or no cases) were “normal.” Comet allowed “normal” customer accounts, products, and plans to be administered with ease, but cases that fell outside these parameters became “problems” that had to be worked around. There was enormous pressure to make these problem cases normal, since to not do so would involve custom work on our part and thus more expenses for Company J.

Much of this user configuration was the result of specifications given to us by Company J. We were often explicitly told not to expose certain functions to users so that the company could maintain centralized control of customer data. As Agre points out, “The technical conception is not exclusively the property of computer programmers but resides in a larger bureaucratic tradition.”²⁴ The bureaucratic strain of management held sway over the corporatist tradition at Company J: there was never any rhetoric about empowering users or “selling” Comet to its users: it was expected that employees would accept without complaint whatever system they were told to use.

I have attempted here to analyze the problems encountered in an average business software development project using a variety of sociological tools. First I looked at the issue of “company politics” and tried to use the concepts of relevant social groups and interpretive flexibility to explain why these politics are not simply an irrational epiphenomenon but are an intrinsic part of software development. Seen in this light, the specification phase becomes an attempt by the groups that hold power to achieve closure and stabilization.

²¹ Agre, “Conceptions of the User,” 67.

²² Agre, “Conceptions of the User,” 79.

²³ Keith Grint and Steve Woolgar, “Configuring the User: Inventing New Technologies,” in *The Machine at Work: Technology, Work, and Organization* (Cambridge, UK: Polity Press, 1997), 83.

²⁴ Agre, “Conceptions of the User,” 99.

Next I examined the problem of communication breakdown during the prototyping phase, and argued that the real issue was the process of formalization and its tendency to “delete the social”²⁵ aspects of work. Despite the efforts made to converge on common meanings during the specification phase, formalization results in a divergence between the meanings users find in their work and the meanings encoded in the developers' models of that work. This divergence is the root cause of many common complaints heard by developers during the prototyping phase.

To explore the problem of user rejection at the deployment phase, I used the idea of “configuring the user.” I tried to show that this rejection was due not to irrationality or recalcitrance on the part of the users but resulted from a “misconfiguration” of Comet. In this case, besides being configured for the wrong users, the system defined a role for the operators that was far more constrained and standardized than what they expected or wanted.

Finally, at each stage I tried to show the subtle forms of bias that can creep into computer systems. During the specification phase the existing practices and attitudes of the decision-making parties can be embedded into the system as a form of pre-existing bias. While the system is being prototyped, technological constraints can cause technical bias. And during the deployment phase, bias can emerge from the context in which the system is actually used if this context differs from the one for which the system was designed.

Some of these ideas proved to be more useful than others. Looking at the relevant social groups and the problems they wanted solved proved to be quite powerful. Software developers have a tendency to see things in binary terms of “us and them,” so that even developers who are sympathetic to “the user” wonder why “they” are so inconsistent. SCOT gives a far more nuanced view that makes sense of this inconsistency by showing that “they” are actually comprised of many different groups. It suggests that for the development process to proceed more smoothly developers should make an effort to identify these groups and their views and needs as soon as possible, and approach the specification process as an attempt to find a solution that balances these needs.

Looking at the process of formalization and its tendency to throw out the social aspects of work also helped to explain some mysteries. Specifically, it showed why users complain about systems that are seemingly built to their specifications. But while having a description of the problem is useful to some extent, unlike SCOT this diagnosis does not immediately suggest practical steps developers might take to minimize or avoid the problem. While it certainly couldn't hurt to make developers more aware of the importance of the non-formal, it is unclear how to translate this into changes in software development practice. The notion of a “double-level language” may prove useful, but I would need to look at the idea more closely to make any judgments about its feasibility.

“Configuring the user” is an interesting way of looking at the interaction between computer systems and users, but I find it too vague to be of much real-world use. More traditional concepts of usability and user satisfaction are easier to understand and provide better practical guidance when building systems. I don't expect that software developers will be thinking of their creations as “texts” anytime soon.

²⁵ Diana Forsythe, “Blaming the User in Medical Informatics: The Cultural Nature of Scientific Practice,” in *Studying Those Who Study Us: An Anthropologist in the World of Artificial Intelligence* (Stanford, CA: Stanford University Press, 2001), 15.

Finally, Friedman and Nissenbaum's concepts of bias highlight some real problems that developers ought to be aware of. However, for a system like this one I feel that the term "bias" is too strong. The cases of systematic unfairness discussed here are the results of inevitable trade-offs made during the development process. Certainly these trade-off ought to be made consciously rather than unconsciously, but labeling them with the term "bias" dilutes the meaning of that term and diminishes the actual cases of bias that Friedman and Nissenbaum discuss in their article.

Sociological analysis is mostly alien to the practicing software developer. While participatory design practices are gaining in popularity, the technical conception of the user still reigns supreme. This narrow view of the world results in frustration for both developers and their customers and may have something to do with the incredibly high failure rate of software projects. One might hope that a wider understanding of the social aspects of software would help developers understand and perhaps even solve some very common problems encountered during development.

Unfortunately, I do not expect this to happen anytime soon. Charles Simonyi, a former Chief Architect at Microsoft who has founded a new company "dedicated to assisting software developers in capturing the tremendous latent value that is usually lost in the design and development process,"²⁶ shows just how far developers have to go before they stop "deleting the social" when he suggests that what is needed is "a special editor—a sort of super Power Point—that assists [users] to record and maintain their intentions and also the meta-data... about their notations and terms."²⁷ Simonyi's solution to recovering the value lost to formalization is... more formalization. If only it were that simple.

²⁶ Intentional Software Corporation, "Corporate Info," <http://www.intentionalsoftware.com/> (accessed December 9, 2003).

²⁷ Charles Simonyi, response to Jaron Lanier, "Why Gordian Software Has Convinced Me to Believe in the Reality of Cats and Apples," *Edge*, no. 129 (2003), http://www.edge.org/3rd_culture/lanier03/lanier_index.html#simonyi (accessed December 9, 2003).