

Extreme Translation

A Rapid Prototyping Methodology to Bridge the Gap
Between Programmers, Designers and Users

Scott Davidoff
Managing Software Development
F03-17653-A
01 December 2003

Executive Summary

The hardest single part of building a software system is deciding precisely what to build. No other part of the conceptual work is so difficult as establishing the detailed technical requirements, including all the interfaces to people, to machines, and to other software systems. No other part of the work so cripples the resulting system if done wrong. No other part is more difficult to rectify later. (Brooks 95)

Users possess domain expertise, and programmers possess technical implementation expertise. Knowledge must transfer between the two groups for innovative and appropriate software development to occur.

Both Extreme Programming and Contextual Design offer structured protocols to manage the communication process, and to guide the creation of requirements. But while each process proves highly capable, they each also fail to provide complete coverage across the large body of risks during that communication process.

By developing a theoretical understanding of the communication process, managers will be better able to identify the risks involved in the requirements elicitation process.

XP involves the customer in the decision-making process, but provides no real structure to ensure that the customer provides accurate information, nor does it create a structured process to manage that communication. XP fundamentally assumes that a client can determine both what is wrong with their business processes, and recognize a capable solution.

CD provides a highly specific structure to accomplish just those oversights. But it lacks a specific, code-based iterative prototyping method to guide the development process.

By merging these two processes, we will possibly see an increase in innovative products developed.

Managers reading this paper will

- Gain deeper understanding of the communication risks involved in the requirements-gathering process
- Understand when to apply different design strategies
- Pick and choose concepts from various approaches
- Integrate these various approaches
- Provide guidance to managers looking to follow in this experimental development process

Introduction

The field of Software Engineering has emerged in recent years to add the discipline of engineering to the previously ad-hoc practice of developing large-scale software systems (Basili & Musa 1991).

Evolving along with the field are distinct methodologies to help the manager plan and structure a project.

To apply engineering methods to the practice of software development, managers must understand all the choices that lay before them, understand what these methods will do well -- and what they will not. Managers will understand the development opportunity that lies before them, and ultimately make a good match between project needs and process capabilities.

The most challenging and important decision a manager must confront to develop a large scale software project is how to define the requirements, or what will be built. The baffling paradox remains that while in principle the requirements process sounds relatively simple -- determine what you are building -- it is in actuality extremely difficult to accomplish.

The hardest single part of building a software system is deciding precisely what to build. No other part of the conceptual work is so difficult as establishing the detailed technical requirements, including all the interfaces to people, to machines, and to other software systems. No other part of the work so cripples the resulting system if done wrong. No other part is more difficult to rectify later. (Brooks 95)

Simply put, decisions early on in the software development process affect the entire remaining pieces of the development process. So while managers must struggle with ways to deliver the large scale systems that will deliver what their clients need, determining what those clients need is the single most important decision managers must undertake. In fact, 3 of the top ten risks to a software development project Boehm identifies involve the requirements process (Boehm 1988).

Research into the requirements process has turned towards the more abstract concepts that surround the social interactions between the developer and user (Basili & Musa 1991). Researchers identify four subject areas where process breakdowns contribute most to the failure of the process: (1) Communication between developers and users, (2) the constrained perspectives of information system developers, (3) cultural differences between developers and users, and (4) mutual misunderstandings between designers and users (Bahn 95).

This paper will focus on the communication relationships between developers and users. Much research has been done to understand the dimensions of this relationship.

Clients have domain expertise. Programmers have technology expertise. Any successful system will depend on the successful transfer of domain knowledge between developer and user.

The information arising from the communication between the user and the analyst represents the foundation of information systems design, and therefore is a key factor in determining success or failure of a project (Velenti, Panti & Cucchiarelli 1998).

Salaway goes even further in his understanding of communication

Communication between the analyst and the user is the single most important part of Requirements Elicitation, forming the foundation for the success or failure of the project (Salaway 1987).

But precisely what information needs to be transmitted? And how can we increase our chances that information so vital to the success of a project be transmitted efficiently and perceived correctly? As managers, what structures and methods are appropriate to facilitate this process, and push it towards a successful completion?

Research into this area is noticeably absent. This paper will combine research from several fields to attempt to synthesize how, as managers, we can identify spots of process failure, and take steps to correct these flaws.

A theoretical framework for requirements process communication

Early models defined the communications of information according to "tube model," where a virtually concrete entity that could be passed from one person to another (Kensing & Munk-Madsen 1993). This theory put forward that a speaker broadcast information another person received it, much like an radio transmission, or the passing of a ball.

But recent work suggests that the communication process involves much more sophisticated transactions. Another model of communication suggests that

Successful communication depends on the ability to establish situations in which mutual perturbations trigger changes in the state of those involved, which in turn lead to structural congruence (Kensing & Munk-Madsen 93)

Here, communication does not directly transfer knowledge, but rather initiates a state change in the receiver, that depends on how they internalize and interpret what is communicated.

If communication is indeed a potentially imperfect transaction, each exchange has the potential to expend effort without receiving any reward. As managers if we examine the characteristics of each individual transaction, we might understand better what causes these communication breakdowns, and help understand how we might employ software development methodologies that can better support these communications.

The requirements specification process covers six different fundamental types of communication between user and programmer (see Figure 1)

	Users' present work	Technological options	New system
Abstract knowledge	Relevant structures on users' present work (2)	Overview of technological options (4)	Visions and design proposals (5)
Concrete experience	Concrete experience with users' present work (1)	Concrete experience with technological options (3)	Concrete experience with the new system (6)

Figure. 1

*Six areas of knowledge in user-developer communication
Adapted from Kensing & Munk-Madsen (1993)*

Users start out with knowledge of the domain, which they need to transfer to the developers (Communication 1). Programmers also possess an overview of the technological options, which they communicate to users (Communication 4). Both groups benefit from this communication, and the outcome of this exchange is the vision for the software product and preliminary design solutions to the problem situation (Kensing & Munk-Madsen 1993).

We can examine the flow of communication through the requirements specification process sequentially (Figure 2)

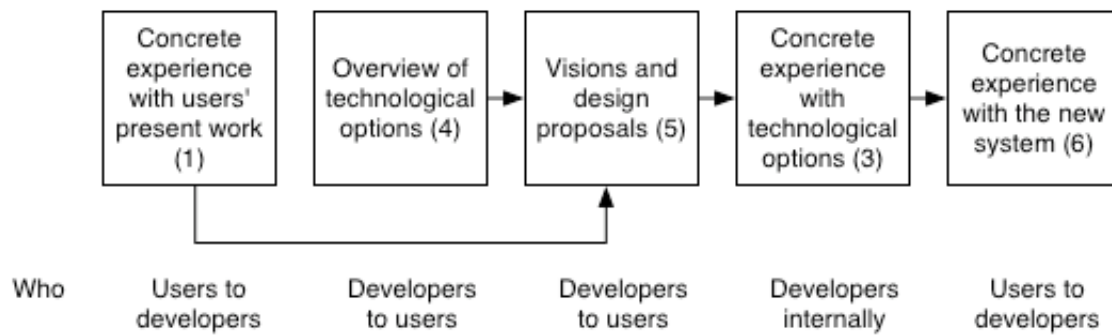


Figure 2
Communication flow in the requirements process

Risks in the current requirements model

Risks to communication 1 and 2

To apply relevant abstract structures on users' present work -- or Communication 2 -- developers mostly apply their knowledge of systems to the domain that they have learned about through Communication 1. Communication 2 is mostly internal amongst developers, and presented as conclusions to users. But often, users have little access to this communication, and so cannot correct errors, mis-assumptions, or flaws due to miscommunications of communication 1.

The principal work of the programmers is then to turn their understanding of the work domain into visions and design proposals. Often this process is more of a sales pitch to users than a collaboration.

To develop the software, then developers communicate amongst themselves about the concrete experience of technical options (3). If prototyping or refinement process are used, then developers present prototype systems to users, who then are given an opportunity to respond, and refine the prototype systems.

Overall communication risks

There are several primary challenges to the current method of requirements gathering

1. Users' inability to communicate their process
2. Users' inability to understand and communicate their needs
3. Inaccuracy of self-reporting
4. Lack of cultural context
5. Differences in vocabulary
6. Differences in goals and priorities

7. Relevant structures on users' present work

Users' inability to communicate their process

Users understand how to conduct their business. But often, it is more difficult for users to *explain how* they conduct their business.

Users' inability to understand and communicate their needs

Patients go to doctors not simply to provide they healing they know they need, but to get a check-up, understand alternative treatments and their consequences, and to make informed and collaborative decisions.

Clients are no different. Whether they come to developers because they lack the resources, skills, or experience to cure themselves, part of the services developers provide is helping to identify the problem, and proposing sufficient solutions.

Brooks is more adamant in his diagnosis of client capabilities.

I would go a step further and assert that it is really impossible for clients, even those working with software engineers, to specify completely, precisely, and correctly the exact requirements of a modern software product (Brooks 95)

Inaccuracy of self-reporting

Market research shows that 50% of the United States reads the New York Times. If that were that the case, the NYT circulation would be well over !25 million. But it is significantly less. The fact remains that people tend to report what they want you to know about them, not necessarily what is true. Whether intentional misrepresentation or self-deceptive peccadilloes, we would be wise to question the information clients tell us about their business processes.

This often means that users describe the official versions of processes that few people would follow. Basing designs upon these versions result in many design failures, as few companies really work "by the book."

Lack of cultural context

While users might actually describe their needs, developers cannot predict how changes to software will affect the cultural context in which the software will be forced to exist. But users often exclude this essential cultural context from their descriptions for reasons of privacy, discretion, lack of information or politics.

So it is a great risk to design a software without understanding the real context in which it will ultimately need to be adopted.

Differences in vocabulary

Different people use different words to describe the same things. Differences in education, training, work practices, goals and culture exacerbate these conditions.

This is especially true between clients and developers.

Users cannot be expected to describe their work and needs from the point of view of an engineer. Conversely, engineers seldom have an intuitive grasp of their users' working life, or of the environment in which a product will be used (Williams & Begg 102).

Engineers understand the world in highly logical terms, and are comfortable with abstract concepts and modeling. But users often think concretely. Both groups tend to speak in highly-specific industry jargon.

How XP manages these risks

Extreme Programming (XP) emerged as software development methodology to create products that deliver more value for business. XP aggregates many innovations from the software engineering heritage, and produces some interesting innovation of its own.

Among its many principles, XP directly involves the customer in the development process. This aids the requirements communication process fundamentally by providing more frequent and structured interaction client interaction.

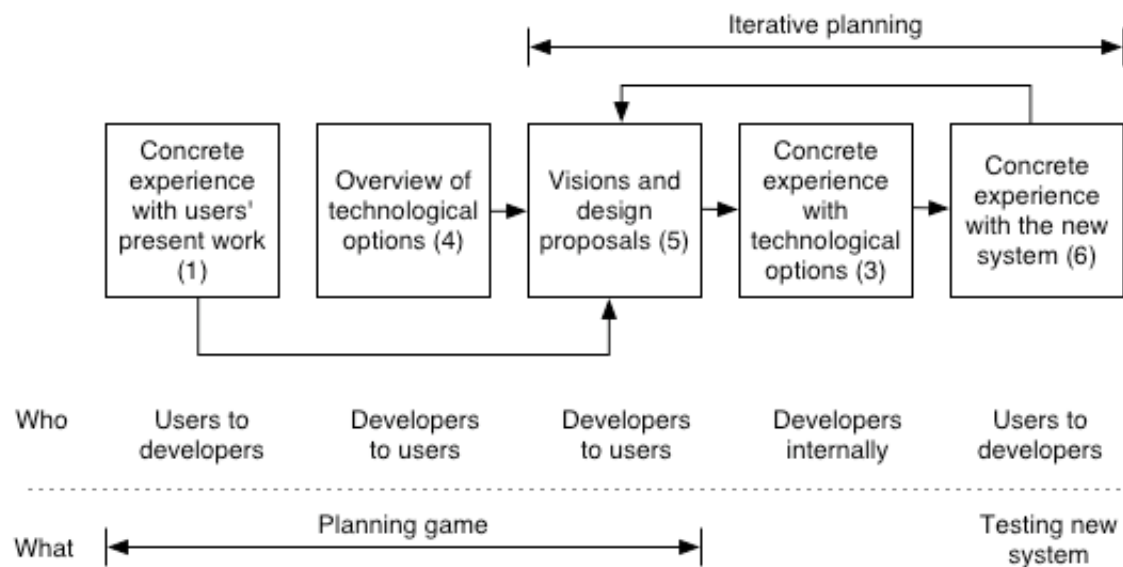


Figure 3

The XP communication flow

XP principally brings the client directly into the design process. The business representative, called "the customer," provides the information for the requirements process through a coordinated collaboration with the development team.

The Planning Game

The principal vehicle for this communication is a process called the Planning Game (Beck 99). In the Planning Game, a representative of the client sits with the XP team. The team works together to create user stories.

User Stories are written by the customers as things that the system needs to do for them. They are similar to usage scenarios, except that they are not limited to describing a user interface. They are in the format of about three sentences of text written by the customer in the customer's terminology without techno-syntax (Beck 99).

This protocol provides a structured process where users create a list of goals, and to prioritize them. It gives them a context through which to explain their concrete work experience (Communication 1). It also forces developers to speak less abstractly as they explain the technological options (Communication 4).

As the customer learns about and chooses from a menu of desired features, the Planning Game also helps users and developers to collaborate to create a shared vision of what is possible (Communication 5).

Pair programming

Extreme Programming teams work in groups of two, sharing ownership of the code they create together. Interestingly, this produces more reliable and understandable code in less time than would be possible individually (Beck 99).

This intense collaboration also increases the programming team's internal dialog (Communication 3), creating the solid conceptual integrity of a strong shared vision, and facilitating easier control of its implementation.

Conspicuous absences

XP lacks a defined process for facilitating abstract understanding of the user's work process (Communication 2). This process often relies on the developers to apply abstraction to the work processes of the users. It is the earliest and most raw form of design.

So while XP allows for the user to collaborate on these processes, it does not facilitate how they might make more meaning from these interactions. And as the communication model suggests, simple transmission and reception is insufficient to ensure the intense kind of information transfer that occurs during these phases.

How Contextual Design manages these risks

Contextual Design (CD), a method championed by Hugh Beyer and Karen Holtzblatt, combine many of the innovations of ethnography, cognitive psychology and design into a single method with a specific vocabulary for the software creation process.

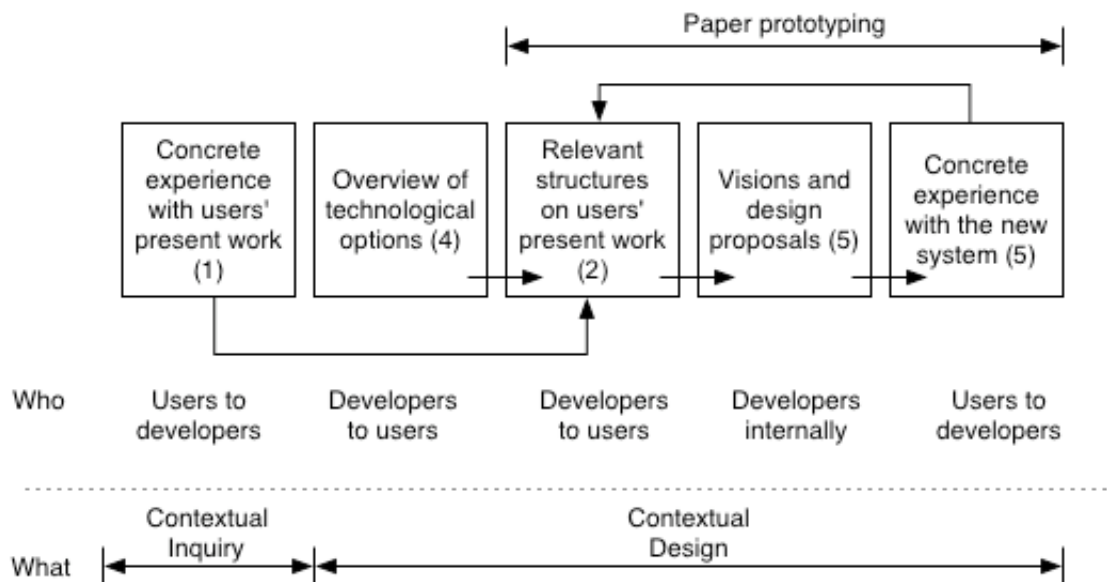


Figure 4
The CD communication workflow

CD is specifically engineered to enable teams with various backgrounds to efficiently collaborate

Contextual design is a method that helps a cross-functional team come to agreement on what their customers need and how to design a system for them (B&H 1999).

In-context observation

CD focuses most specifically on ways to get reliable data from customers (Communication 1). It does not trust the information that the customer provides

in isolation, but rather, suggests that reliable data can only be gathered in conjunction with in-context observation. This is the only way to

provide reliable knowledge about what customers do and what they care about (Beyer and Holtzblatt 1999)

CD analysts are trained in fieldwork methods to gather more reliable information, and takes an interesting approach to Communication 1, treating with skepticism the customer's abilities to express what they ultimately need.

Artifact walkthrough

Studies have shown that humans suffer from selective perception and make judgements easily (Valenti, Panti, Chccchiarelli 1998). CD analysts are also trained specifically to mine users for concrete data using a technique called Artifact walkthrough. Rather than asking "How much time do you spend on a particular activity," artifact walkthroughs ask specific questions about particular artifacts of the workplace, and allow the analyst to choose which details merit inclusion and generalization. This ties users to answering specific questions that they are much more likely to provide reliable answers to.

Modeling from data

CD then provides a concrete language for formal modeling of the in-context data. This modeling is to be a collaborative process, where analysts and field-workers transform their data into explicit diagrams representing the flow of communication and artifacts, the cultural environment, the artifacts themselves and the physical workspace.

Interpret the data in a cross-functional team, which creates a shared perspective on the data. Then we consolidate the data across multiple customers, creating a single statement of work practice for your customer population (Beyer & Holtzblatt 1999)

Prototyping

CD also suggests the use of prototyping to refine the results of the design process. But here it offers little guidance or protocols, other than suggesting that the teams use paper. There is no real formal structure for refining the specific designs.

Proposition: Combine XP and CD

Both XP and CD present interesting methods to manage the communication protocols during the requirements gathering process. But the stated

communication model reveals that they both overlook certain important aspects of the communication workflow.

By integrating the CD and XP processes, an interesting hybrid might successfully account for the oversights of each individual method. Figure 5 shows the combined workflow of this XP/CD process.

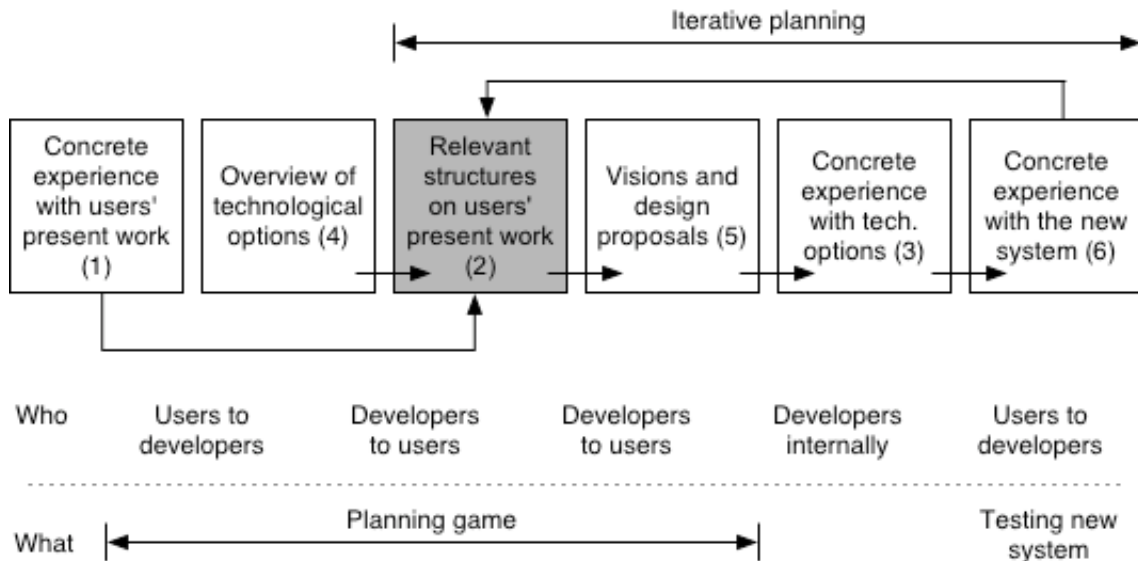


Figure 5
The combined XP/CD communication workflow

While XP takes great steps towards encouraging user-developer communication, it still fundamentally relies on the ability of the user to faithfully represent their workplaces. For the many aforementioned reasons, this proves a dubious hypothesis. XP faithfully recognizes its own shortcomings, and attempts to overcome them using a stringent prototyping regiment and effective method of producing code.

But it still does not provide a structured process for how to get reliable data from users. And this is its greatest shortcoming.

CD, on the other hand, excels at specific and structured ways to elicit data from the users. But CD lacks the methodical rigor of code-based prototyping. In this way, it lack much of the real iterative spirit of the sustained development and bi-weekly releases the XP suggests.

Hypothesis: CD Analyst as translator

In the hybrid XP/CD process, the CD analyst sits with the customer, and serves to facilitate both the extraction of useful data from the client representative. In this way, the analyst plays the role of translator, both in literal and figurative ways (Williams & Begg 1993)

The designer is the metaphoric bridge between the worlds of the user and the programmer, having experience in both relating to users and programmers effectively. The analyst also serves to ensure that there is little misunderstanding between the two worlds. And the formal system modeling of the analyst can help the user and the programmer reach a common understanding of the workplace by creating a protocol to structure the elicitation of information from the user, and repetition of that information back to the user.

Hopefully, this cross-functional team will produce more innovative solutions. If they are able to get more reliable data, and that data in turn will drive more effective design based on a shared communication process, then this hybrid might create a situation where innovation is more likely to result.

successful projects all had one or more project members [who] spanned organizational boundaries. One type of boundary spanner was the chief system engineer, who translated customer needs into terms understood by developers (Curtis, Krasner, & Iscoe 1988)

Risks in managing a XP/CD process

The combined XP/CD process will bring together two different teams, for the purpose of creating more innovative products. But any time two cultures, with different goals and world views come together, there are process-related risks.

Project size

XP is currently known to not scale well to large-scale development environments. So, until it is determined how to bring XP into the domain of large programming situations, it is doubtful this particular version will be appropriate either. How to measure the exact size that is "too large" is beyond the scope of this inquiry. But managers considering this approach as an alternative should consult the large body of literature surrounding XP, and the other agile processes, to apply the XP methodology to a software development project.

Resources

Hiring an HCI analyst requires more resources, and increases the overall cost of the project. In order to justify the increase in cost, the project will have to similarly increase some visible, testable and measurable, benefit. The hypothesis of this paper is that this kind of teamwork will produce better more innovation at a higher cost. So projects that require a requisite innovation to deliver on the needs of the business tend to be the best fit.

An idea experimental environment will be a project of sufficient size to bring in the requisite budget, but not big enough to push the effective ceiling size of XP projects in general. And among projects of that size, those with a particular requirement for innovation might consider this experimental mix.

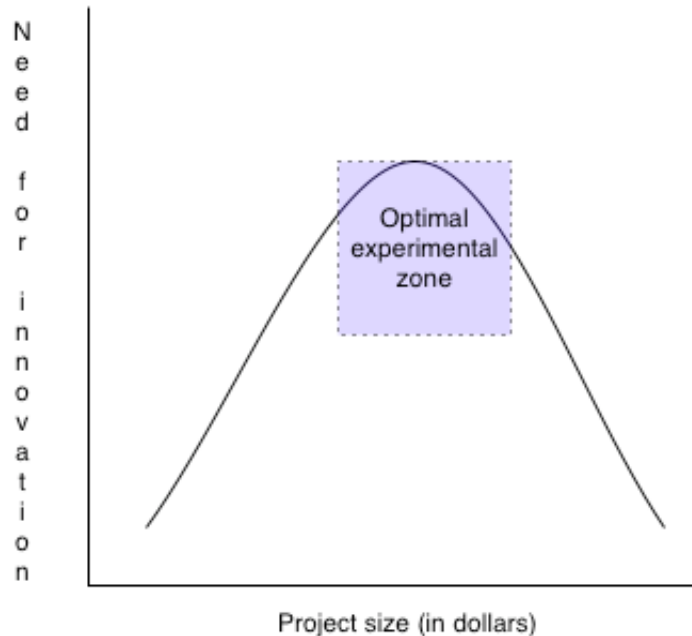


Figure 6

Need for innovation plotted against project size, measured in dollars.

Team Dynamics

In general, team diversity "brings more creativity to problem solving and product development" (Gladstein & Caldwell 1992). And this XP/CD hybrid certainly suggests adding to team diversity. But there will be a need to manage this change.

simply changing the structure of teams...will not improve performance. The team must find a way to garner the positive process effects of diversity and to reduce the negative direct effects. At the team level, greater negotiation and conflict resolution skills may be necessary. (Gladstein & Caldwell 1992)

Early stage experiments following this kind of software development process will want to allocate additional resources to manage the conflict-resolution process, perhaps by including a more experienced team coach, or by simply building time into the schedule for positive group affects such social time together.

Works Cited

Principal Sources

Beck, Ken (1999). Extreme Programming Explained. Embrace Change. Addison-Wesley Publishers.

Beyer , Hugh & Karen Holtzblatt (1997). Contextual Design : A Customer-Centered Approach to Systems Designs. Morgan Kaufman Publishers.

Kensing, Finn & Andreas Munk-Madsen (1993) "PD: Structure in the Toolbox," Communications of the ACM. Participatory Design. 36(4), 78 - 85.

Williams, Marian G. & Vivienne Begg (1993) "Translation Between Software Designers and Users," Communications of the ACM. Participatory Design. 36(4), 102 - 103.

Other Resources

Ancona, Deborah Gladstein & David F. Caldwell (1992) "Demography and Design: Predictors of New Product Team Performance," Organization Science, 3(3) 321 - 341.

Bahn, David L. (1995) "System Designer-User Interaction: An Occupational Subcultures Perspective," ACM SIGCPR, 4, 72 - 80.

Basili, Victor R. & John D. Musa (1991) "The Future Engineering of Software: A Management Perspective." IEEE Computer, 20(4), 90-06.

Beyer, Hugh & Karen Holtzblatt (1999) "Contextual Design." interactions, 6(1), 32 - 42.

Boehm, Barry W (1988) "A Spiral Model of Software Development and Enhancement," IEEE Computer, 5, 61-72.

Brooks, Frederick (1995). "No Silver Bullet -- Essence and Accident in Software Engineering." The Mythical Man-Month. Addison Wesley Longman.

Curtis B., H. Krasner, & N. Iscoe (1988) "A Field Study of the Software Design Process for Large Systems," Communications of the ACM, 31(11), 1268 - 1287.

Gladstein, Deborah Ancona and David F. Caldwell (1992) "Demography and Design: Predictors of New Product Team Performance." Organization Science. 3(3), 321 - 341.

Mills, H.D (1971) "Top-Down Programming in Large Systems," Debugging Techniques in Large Systems, R. Rustin, ed., Prentice-Hall.

Royce, W. W. (1970) "Managing the Development of Large Software Systems: Concepts and Techniques," Proceedings of Wescon, 8.

Salaway, G. (1987) "An Organizational Learning Approach to Information Systems Development," MIS Quarterly, 11(2), 245 - 264.

Valenti, S, M. Panti & A. Chcchiarelli (1998) "Overcoming Communication Obstacles in User-Analyst Interaction for Functional Requirements Elicitation." ACM SIGSOFT Software Engineering Notes, 23(1), 50 - 55.