Evolving Concepts
or
Why Users Often Don't Recognize the Software They Asked For

Gary Mrenak

Top-Down Software, Inc.
1335 Carlsbad Drive
Gaithersburg, Maryland 20879

ABSTRACT

When a development project delivers a software product that is not usable without modifications, it is because the product does not serve the current needs of the product users. Since only trivial problems are fully grasped at first consideration, users will always find it necessary to modify their early concepts of what they need. In the traditional linear development model, the early needs of users are captured during the system and software requirements definition phase, but the evolving needs of users are effectively ignored through the development phases. An improved development model will be user driven and responsive to changing requirements whenever they occur.

INTRODUCTION

There are two difficult problems in software development: figuring out what to do, and figuring out how to do it. Up to now, the software industry has spent nearly all of its time on the second problem: figuring out how to translate requirements into code. Pick up any popular text on software engineering, say [1], and compare the depth of material devoted to requirement capturing vs the extensive discussions of post-requirements development: planning, design, coding and test. It's not surprising that the industry has chosen to approach the problems in this order, since there would be little reason to know precisely what has to be done if it was not possible to do it. But to accomplish the task of learning how to do it, the industry has taken a stationary view of the first difficult problem - what is to be done. It has regarded the concepts of system and software requirements as static and determinable at the beginning of the process, and it has used these formally determined requirements as the foundation of the software development process (See Figure 1). It formalized this view by embracing the linear software development model with its phases, milestones, reviews and baselines. The linear model resists any modifications to assumptions or conclusions that have been "approved" in an earlier phase of the process. This is a development model with limited user interactions. It is a model that's responsive to a snapshot of perceived requirements taken early in the process. The result has been gradually increased productivity in the software development process, but, predictably, little increase in the usability of the products.

Software products are not usable because they do not serve the current needs of the users. When the user's needs were first examined in the requirements analysis phase of the traditional process, they may have been captured in careful and explicit detail. But, as Frederick Brooks observed in [2], "Much of present -day software-acquisition procedures rests upon the assumption that one can specify a satisfactory system in advance, get bids for its construction, have it built, and install it. I think this assumption is fundamentally wrong, and that many software-acquisition problems spring from that fallacy ... it is really impossible for a client ... to specify completely, precisely, and correctly the exact requirements of a modern software product before trying some versions of the product." So users rarely have well defined needs, least of all in the early stages of the product's development.

Software engineering will continue to improve our understanding of the translation process, improving productivity by continuing to

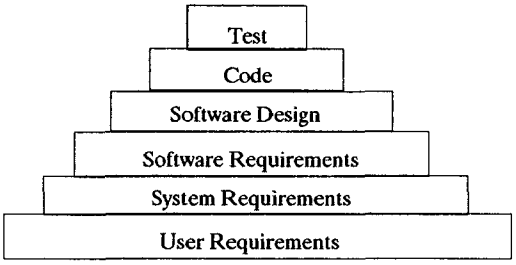

Figure 1. User requirements underly the traditional linear software development process model. They must remain substantially static to support the remaining process phases.

automate more and more of the process. But improvements to the efficiency of the process will have to be accompanied by improvements in the usefulness of the software that is produced, which implies an improved method of capturing what is to be done. In the terms suggested in this paper, this means improved methods of capturing the changing perceptions, preferences, and conceptions of users. It means replacing the stationary model of software requirements with a more effective model of dynamic requirements and a development process that is responsive to these evolving requirements.

There have been, of course, several recent development models that attempt to improve the quality of the requirements that support a software product, such as rapid prototyping [3] and incremental development. These models generally implement a specify-build-try approach that encourages users to improve their understanding of what they want by interacting with "working" software. Although these and other improved models are an important step forward, there is more to be understood about how users can interact more effectively with the software development process, and how to make a process respond more effectively to changing requirements.

USERS: The Critical Element

Users are the seed for most software development projects. They are the first to perceive the need for a computer-based solution and the first to conceive of the possible forms and functions of these solutions. Even when a new software product solves problems that a whole community of users may not have even perceived - for example, spreadsheets for accountants, word processors for secretaries, DBMSs for managers - someone intimately familiar with the problem, a user, first recognized it and conceived the solution.

We have paid very little attention to the user constituency and the nature of their contributions to the software development process. We know relatively little about:

> how users interact with their problem environment to form perceptions and to conceive solutions,
>
> how concepts mature and evolve through both internal processes of thought and external influences,
>
> how perceptions are influenced by internal biases,
>
> how to access the user's changing concepts effectively, so that the final software product reflects current needs.

We have paid little attention to users in the process because, in part, we have regarded the solutions to computer system problems as largely determinate and stationary. We have assumed that the correct solution to any problem exists, is independent of time, and only remains to be discovered by the application of a scheduled, semi-rigorous regimen known as "requirements analysis."

Following the analysis phases, where narrative requirements and specifications are typically generated, traditional processes concentrate on the management of the project - documentation, schedules, measurements, and budgets. Once requirements have been "approved" by the user, projects focus on their implementation, and contractually discourage all user involvement that may result in changes to "baseline" agreements. Users wait months or even years to exercise a non-narrative model produced by the software process, which, unfortunately, is likely to heavily represent only his early concepts and solutions.

In the traditional process, the user's internal concepts - complex, unstable, evolving, directly relevant to the perceived problem, usually supplemented by no more than occasional paper representations - are the only developmental mechanisms available until the software development process can provide external realizations. Meanwhile, using their internal concept models, spreadsheets, and doing some off-hours hacking with their favorite PC language, users continue to interact with their real-world problem. They continue to explore and improve their understanding and continue to refine their concept of a useful solution. Their ideas of a useful software product are evolving.

CONCEPT HANDOFFS

By "concept" we mean the model of a problem that exists in the minds of individual people. According to Carroll and Olson in [4], a **mental model** "is ... a rich and elaborate structure, reflecting the user's understanding of what the system contains, how it works, and why it works that way." Concepts reflect personal perceptions, perspectives and experience, and they evolve as the individual explores and thinks about the real-world problem. They are influenced by the related concepts of other individuals when they are encountered and when those concepts alter and improve the understanding of the problem. Concepts are elusive and difficult to access:

> it is difficult for one person to communicate a complex or poorly understood concept to someone else. Even if there were a suitable external mechanism to concisely express the concept - say, a universal symbol calculus - the dynamic vagueness of the concept would make that task impossible;
>
> an individual's concept tends to evolve over time, so that Thursday's communication of the concept - even if it was a coherent communication - is likely to describe a substantially different idea from Tuesday's.

Users generate early concepts of problems and solutions. For small problems, where time permits, users personally develop and translate these early concepts into software solutions themselves. In such circumstances, the "software developer" is continuously aware of the "user's" evolving concepts as a realization of these concepts is produced. On the other hand, for large or complex problems in traditional circumstances, particularly in the DoD, users do not directly develop these solutions, and instead hand off these perceptions and concepts to other individuals in the process - acquisition agents and software developers - who establish and conduct the software development project on the user's behalf. This hand off creates multiple concepts of the solution that are individually evolving, and raises the question of which concepts are reflected in the software realization that is produced by the process.

There are three groups of individuals that are usually involved with the software development process:

> the users group: those individuals with a direct or indirect interest in the software product and the real-world problem solved by the software product. User concepts reflect the most relevant perceptions, experience and preferences because they are most strongly influenced by the real-world problem and because it is the user who will

eventually apply the software product.

the acquisition agents, common in military software developments, who manage the acquisition of the software product on behalf of the users group. These agents can sometimes contribute their experiences from acquisitions with similar requirements, but they are most interested in establishing a manageable software development contract and conducting the process according to the contract.

the software developer, usually not associated with the users group, who translates the requirements of the problem into a computer and software solution. Software developers are interested in fulfilling the requirements of the software development contract.

These groups cooperate during a standard series of program steps:

the problem is recognized and the need for solution is established. In the early period of this step, users informally hand off their concepts to other users, and, less informally, to acquisition agents near the end of the step.

the boundaries of the problem and the requirements of the solution are defined and documented. Users and agents formally hand off concepts to developers in the form of written requirements.

the solution is developed and implemented.

Each concept handoff between individuals or groups of individuals is, of course, subject to misunderstanding and misinterpretation. But, in addition, the concepts of the users are changing throughout these steps, adding a "half-life" parameter to every concept handoff that continuously degrades its relevance to the process.

CONCEPT DISCONNECTS

When establishing the requirements of a software product for a development process where users and developers are separate, concept disconnects are fundamental problems. These disconnects are failures to anticipate, discover and communicate all system and software requirements, and can be categorized as **incomplete concept recognition**, **inadequate concept communication**, and **evolving concepts that diverge**. They are the principle reasons that software products fail to meet user expectations.

Incomplete concept recognition

The problem of incomplete concept recognition has two components:

1) the identified user constituency may be incomplete, and,

2) the identified user constituency may be under-represented.

A user constituency that is **incomplete** is a failure to recognize all direct and indirect users of the computer system solution. Where the scope of the real-world problem is not well understood the early identification of all users may be difficult or impossible. The problem may be new, or the problem may be only a component of a larger undetected problem with an expanded set of interested users. An incomplete user constituency will lead to a concept of a software product that does not address the perceptions, views and preferences of all users.

An **under-represented** user constituency is a failure of all identified users to participate in the development process. Under-represented users have been identified but they do not participate effectively. They are either unprepared to participate because of other demands on their time, they may be indirect users who do not fully grasp the implications of the proposed concepts, they may be biased against the software development and resist constructive participation, or, more commonly, they are simply never asked to participate. Users may be under-represented when they communicate with their acquisition agents to produce a tangible problem model that reflects their conceptual models, and in communications with the developer when they evaluate the interim software products during periodic reviews.

In a recent software development project, the developer was asked by the customer, the manager of a financial analysis group, to develop a computer-based system that centralized and enforced data integrity for data that was shared by a number of separate analysts in the group. Each analyst used a separate spreadsheet on separate PCs, and each spreadsheet contained information that was common to the tasks of several other analysts, but was separately and manually entered by each analyst as the information was needed. The developer carefully noted and included the requirements of the department manager and the liaison analyst that was assigned to help the developer, but failed to solicit the thoughts and preferences of the other analysts in the group. The developer chose to implement a relational database management system with a customized shell of menu selections. When the new system was delivered, it fulfilled the data management requirement beautifully and it handled some of the requirements of the liaison analyst reasonably well, but it was extremely slow and failed to match the spreadsheet flexibilities that some analysts relied on. The system, along with six months development time and over $200K development costs, was unusable and promptly shelved.

Inadequate concept communication

Even when all users have been identified, it is difficult to communicate with other individuals because of the nature of mental concepts and because they do not exist in a convenient form for narrative expression. The following characteristics of conceptual models are described in [4]:

- initial models may be crude, erroneous and filled with superstitious beliefs
- the rules of formal logic are not necessarily followed
- models change to account for new experiences
- aspects of an early model may be highly resistant to change because of deep-seated expectations or stereotypes
- models may be incomplete

We would add the following characteristics:

- when exercised mentally, they work, even though the problem may be poorly understood at first. Inconsisten-

cies are overlooked and missing functionality is magically created and inserted where needed.

- they are graphical. Exercising the model proceeds as a motion picture of actions and results, not as a series of paragraphs composed of sentences.

- they are simple at first, hiding complexities behind a vague mental gauze to be developed later in the evolution of the concept

- they are dynamic, changing as the conceptualizer considers the problem and reorganizes the model

- they can only be validated through instantiation.

To communicate a conceptual model, the model must be instantiated in some external form. Usually it is translated to narrative explanations, block diagrams, flow charts or other physical representations that can be observed and considered by an audience. But there are further complications. The quality of these communications depends on several factors, some of which are unrelated to the concepts themselves, such as:

- the narrative skills of the communicator
- the ability of the audience to comprehend the concept
- the conceptual divergence that has occurred since the last communication
- the complexity of the concept
- the clarity of the concept in the communicator's mind
- the communication tools available
- the degree of user representation
- the degree of user participation
- the communicative qualities of the review material
- the amount of new material to be communicated
- the consistency of the review material

So effective communication between individuals is a very complex matter that software engineers must address. Typical individuals in the software development process are no more likely to be skilled in communicating effectively than anyone else. Yet our traditional development models rely on early conceptual communications between many individuals to establish the bedrock requirements foundation for subsequent development.

Evolving concepts that diverge

Compounding the issues of incomplete concept recognition and inadequate concept communication is the reality of evolving perceptions and concepts. Since only trivial problems are fully grasped at first consideration, individuals will always find it necessary to modify or discard early concepts and understandings. When users hand off their concepts during system definition, the developer's concepts are formed and carried forward. In the linear development model, user and developer concepts evolve in separate contexts between well defined reviews: the user's concepts in the context of the real-world problem; the developer's concepts in the context of the development contract. Because the contexts of each constituent are significantly different, concepts evolves differently and tend to diverge.

We can think of concepts evolving "horizontally" or "vertically," where the characteristics of horizontal concept evolution are:

- concept changes result from problem-level perceptions and considerations, instead of implementation difficulties or constraints

- concept changes tend to be broad, frequently extending beyond the scope of the original problem. For example, if the original problem was the development of a word processor, a broad extension of the concept would incorporate some of mathematical features of a spreadsheet

And, by contrast, the characteristics of vertical concept evolution are:

- concept changes result from instantiation discoveries and implications. Word processor software, for example, may have to include virtual storage management because the operating system specified does not provide the capability.

- concept changes tend to be narrow, constrained by requirements specifications and the scope of the contract.

User conceptual models evolve horizontally through continued interactions with real world problems and through continued conceptual explorations. The developer's conceptual model evolves vertically in response to details of the development process, such as discoveries from the design of the software solution, and the constraints discovered from implementation and testing. With the infrequent communication milestones of the traditional process, users tend to approach each scheduled review with preferences, perceptions and concepts that are horizontally different from those from the last communication milestone and the requirements specification. Yet developers have been instantiating vertically from the requirements specification, unless contractually amended by interim communication milestones.

Each of these components contributes to concept disconnect and, as far as each is present, offers significant risk to the usability of the software product. For software products to reflect the current needs of users, those needs must be known to the developer. User concepts are the most **relevant** concepts since they represent a direct appreciation of the problem and the user's criteria for the eventual acceptance of the product. But the concepts of the developer are the most **important**, since the system and software products developed will directly reflect the developer's concepts.

IMPROVING THE PROCESS

The traditional linear development process does not recognize the possibility of concept disconnect. In fact, the probability of concept disconnect is increased by the linear process in all but the most well understood and stable linear development projects. Consider the following observations of the process:

- selected users are intimately involved only in the definition of system requirements. After the requirements definition phase, users are regarded as formal approval signatures on contractually obligated documents.

- after "baseline" system requirements are established, the

participation of users is formally limited to reviews, and to commenting on distributed review materials;

- formal reviews tend to be uni-directional, tutorial communications from the developer to the users and their agents. Developers make "presentations" while agent/ users follow along in written "review packages;"

- conceptual models developed during systems and software requirements analysis are "baselined" and changes to these baselines are discouraged by formal change procedures and contract scope constraints;

- communication of developer concepts is poor. Review materials tend to be paper models consisting mainly of narrative prose.

Today there is nearly a consensus in software engineering that the linear model of software development is generally inadequate for all but a few narrow circumstances, and there has been a growing quest for substitute models. The most widely publicized has been the spiral model of software development first published in [5]. In the spiral model, Barry Boehm describes a risk-driven approach to the software process that applies repetitive problem solving steps to the long-poles in program risk, starting with the longest and spiraling through the others until the product is delivered. The approach is flexible and provides for adopting more conventional software development models depending on the type of risk involved. The spiral format does not restrict devoting as much attention as necessary to reducing the risks of incomplete user concept recognition and inadequate concept recognition. But the application of the spiral to the time-dependent risks of concept evolution and divergence provides no guidance for handling the complications of restarting the spiral when requirements change midway or late in the process, or predicting when requirements are likely to have changed and should be re-established.

There has also been increased attention to the user constituency. In [6] Zahniser describes a system development technique invented by IBM called "Joint Application Design" or JAD. The idea of JAD is to get ".. the right people in a room together with a skilled neutral facilitator and, in a week or less, find out exactly what the user wants." The existence of techniques such as JAD show the growing awareness that the difficulty of determining the user's "wants" needs to be given more attention, although it still surrenders to the notion that the user's needs are static and determinable in some defined period of time.

The problems of concept handoffs and concept disconnects imply a shift from a static model of project requirements to a new model that anticipates changing user requirements. The new model would rule out a formally defined period of requirements analysis that was separate from design, implementation and testing, and in its place substitute a user interactive process that encouraged the discovery and expression of changed requirements whenever they occurred in development. Instead of a "requirements phase," the new model would have an "installation phase" where the network of users is identified and the mechanisms for their interactive participation in the process are developed and put in place.

In an improved process, a portion of the developer resources might be distributed according to the network of users as part of his "on call" staff. User interactions with working models of the software and suggestions for adjustments to those models could be leveraged by the direct assistance of developer personnel.

Communication in the improved process would minimize reliance on narrative expressions. Several guidelines have been suggested [1]:

- Consistency. Concept representations should reflect the context of the user, incorporating what the user already knows as the framework for communication.

- Simplicity. Concept representations should provide clear boundaries with a limited amount of functionality within those boundaries.

- Completeness. Concept representations should cover all aspects of the concept, possibly using a layered representation approach, proceeding from general to specific.

- Anticipatory. Concept representations should anticipate logical but erroneous user interpretations or perceptions to help improve concept reconstruction.

An improved process could be broadly summarized as follows:

- the software development process should surround the user. Evolving user ideas should drive the evolution of requirements and the user's ability to shake out requirements should be leveraged by the process. The process will need to become user driven; the software development staff amplifying the user's ability to exercise changes in concepts.

- the process should include a mechanism for identifying the user constituency. It should never be assumed that the limited group of individuals requesting the software development represents the whole of the user constituency.

- the process should include mechanisms to facilitate the participation of as many user constituents as possible. It is not enough to simply identify the user constituency, their ability to effectively participate in the process should be established before beginning the process.

- the process should provide frequent interactive communication events between the user constituency and the developers. Interactive communication events are distinguished from the "reviews" of traditional processes insofar as their purpose is to provide users with some form of active model of the software solution that can be exercised and evaluated.

- communication events within the process should provide mechanisms for conceptual (graphical, visual) communication methods in the user's environment.

- communication events within the process should provide mechanisms for bi-directional communications;

- the process should include a mechanism for monitoring, measuring and improving communication effectiveness.

SUMMARY

We have discussed some of the reasons for the unpredictable quality of software as it has been produced by the software engineering state of the art, which include:

- users rarely have well defined needs in the early stages of software development, yet widely used development models ignore user needs after the requirements analysis phase is completed,
- the linear development model increases the probability of an unusable software product by discouraging changes to requirements baselines

When a development project delivers a software product that is not usable, it is because the product does not serve the current needs of the product users. Since only trivial problems are fully grasped at first consideration, users will always find it necessary to modify their early concepts of what they need. In the traditional linear development model, the early needs of users are captured during the system and software requirements definition phase, but the evolving needs of users are effectively ignored through the development phases. Having dutifully conveyed their concepts for the software product during analysis, users develop improvements to their concepts independently from the development process and are given no effective means of updating the process. Consequently, the delivered software product may be seriously out of step with current user needs.

REFERENCES

1. RS Pressman, Software Engineering - A Practitioners Approach, McGraw-Hill, Inc., 1987
2. FP Brooks, Jr., "No Silver Bullet - Essence and Accidents of Software Engineering," IEEE Computer, April 1987.
3. JL Connell, LB Shafer, Structured Rapid Prototyping - An Evolutionary Approach to Software Development, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1989.
4. J Carroll, J Reitman, "Mental Models in Human-Computer Interaction," Handbook of Human Computer Interaction by M Helander, North-Holland, New York, 1988.
5. BW Boehm, "A Spiral Model of Software Development and Enhancement," 1985, TRW Technical report 21-371-85, TRW, Inc., 1 Space Park, Redondo Beach, CA
6. RA Zahniser, "How to speed development with group sessions," IEEE Software, May 1990.

ABOUT THE AUTHOR

Gary Mrenak is president of **Top-Down Software, Inc.** an Ada and software engineering consulting firm in Gaithersburg, MD. Mr. Mrenak received a BS in electrical engineering from Carnegie Mellon University in Pittsburgh, PA, and an MS in Computer Science from Johns Hopkins University in Baltimore, MD. Mr. Mrenak is a member of the IEEE Computer Society and the Association of Computing Machinery, and can be reached at 301.948.1645.