Software Development as Knowledge Creation

International Journal of Applied Software Technology, Vol. 3, No. 1, March, 1997.

Sidney C. Bailin
Knowledge Evolution, Inc.
1748 Seaton Street, NW
Washington, DC 20009, USA
sbailin@kevol.com

Abstract

This paper proposes a fundamental change in the way we view software development and the role of software in our society. We argue that the conventional understanding of software as a vehicle for automation is responsible for some of the most serious problems in the industry. As an alternative, we present a view of software as knowledge, and software development as knowledge creation. This shift in understanding can help to unsettle many deeply ingrained beliefs that have contributed to the software crisis; it points the way to alternative processes based on the goals of inquiry, discovery, and knowledge sharing.

1 Introduction

This paper proposes a fundamental change in the way we view software: not just the development of software, but its very nature and role in our society. We argue that the prevailing view of software as a vehicle for automation is a half-truth, one with far-reaching negative consequences. As an alternative, we offer a view of software as an encoding of knowledge. This view, when pursued rigorously, can point towards solutions of many problems that have dogged the software industry for decades.

The view of software as a vehicle for automation is, in essence, a functional paradigm. Its implications, however, go well beyond the technical disciplines of software development, such as analysis, design, implementation, and testing. In the second section of this paper, we describe how this viewpoint affects the major non-technical aspects of the software industry: how it leads to the institution of structures and practices that have caused, and that now prolong, the software crisis (see, for example, U.S. House of Representatives, 1989). In this section we also offer a glimpse of how the knowledge creation paradigm can lead to alternative structures and practices.

In the third section we discuss an intermediate view, which regards software systems as models of an enterprise. This view is closely related to the adoption of object-oriented

technology. We describe both its benefits, as a shift away from the prevailing functional view of automation, and its limits in addressing some basic, non-technical problems.

The fourth section presents our view of software development as knowledge creation. Our argument is that software development is, by its very nature, a process of discovery and invention. Development methodologies and tools should foster this process rather than reining it in.

Such a shift may seem ill-advised, likely to increase risk in what is already a risk-plagued field. Discovery and invention are usually considered more appropriate to research than to the engineering of large, complex systems. We are not, however, arguing for undisciplined approach; we argue, rather, for a different discipline. Software development should be based on scientific procedures of hypothesis, experimentation, and test, systematically building on and evolving what is already known.

Underlying this argument is our observation that software is not, as is often maintained, just like any other engineering discipline. It is true that many of the most important advances in software engineering (such as component-based development) have been motivated by the goal of emulating more established fields of engineering (such as circuit design). Our position recognizes the usefulness of this goal, but also its limitation, which is that it ignores the distinctive aspects of software development.

This is the hard reality: software is where we are most likely to put unprecedented functions; it is the medium of choice to hold the ever-increasing complexity of the things we want to build. The role of software in our society, for better or worse, has become that of a complexity sink.

1.1 Why is Software a Complexity Sink?

The root cause of this distinctive role is an unspoken belief that, in some sense, software is easy to change. Today we know better, as the complexity of software has made it, in fact, extremely difficult to change. But the unspoken belief is more basic, perhaps having something to do with the weightlessness of the medium. At some deep level we still understand software in terms of the early programming model shown in Figure 1-1.

It is easy to argue that this understanding is fallacious. Those who would make software just like any other engineering discipline have an implicit (or sometimes explicit) goal of replacing the model shown in Figure 1-1. In this battle they face quite a challenge, however. They must convince developers, who are after all human, that an extraordinarily empowering experience was really an illusion. Some reflection on this implicit goal may help us understand why progress has not come faster.

The powerful effect of the early programming experience is more relevant today than it has been for several decades. A new and vast generation of non-professional programmers has access to tools — scripting languages, world-wide web-based development kits, visual programming environments — that reinforce the experience summarized in Figure 1-1. These new programmers are not just writing private programs for their own use at home. They include doctors developing their own office management software, and entrepreneurs developing web sites. They represent a new segment of the software industry, and with it

the industry is undergoing a kind of second childhood that undermines much of what the field of software engineering has tried to accomplish.

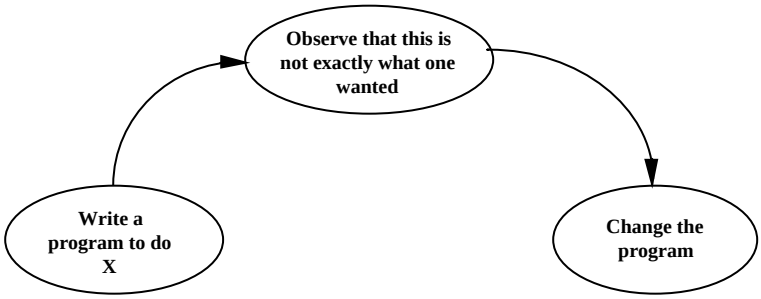

Figure 1-1: The early programming model has influenced our view of software as being fundamentally easy to change.

If we appreciate the power of the early programming experience, we can view the maturation process not as replacing beliefs but as placing them in context. The idea that software is easy to change is not entirely incorrect, but rather myopic: in the private relationship between a sole developer and a software system, change is relatively easy. It is only when the developer and the system operate in a context of external constraints and effects that change becomes difficult.

Despite these lessons, systems continue to evolve rapidly, and software persists as a complexity sink. Apparently, then, the empowering vision summarized in Figure 1-1 regularly overrides the dampening effect of engineering discipline. This is not hard to understand if we take the psychological implications of the maturity metaphor seriously. It suggests that the empowerment of commanding, of automation, can only be replaced by some other form of empowerment. The knowledge creation paradigm suggests an alternative: the empowerment of discovery.

2 Software as Automation

The conventional view of software is that it is a vehicle for sophisticated automation. In other words, software embodies our intentions for how machines should operate. Software development is a process of expressing those intentions in the language of machines. This view has its origins in the field of command and control, one of the earliest applications of software. It is equally prevalent in the field of information processing, and by now it seems basic and hardly questionable.

The depth of this mental model is suggested by the term *programming*, which used to mean what we would now call *software development*. In some circles, such as the new generation of non-professional developers, it still does mean this. In the industry, we have since become smart enough to know that software development involves more than just writing code. But, even as an industry, we have not outgrown the fundamental perception captured in the term *programming*, namely, that software is a tool for commanding machines.

Unfortunately this view, which we have come to regard as obvious, lies at the root of phenomena collectively known as the software crisis, such as the explosion of cost, unpredictability of schedules, and incorrect behavior of the end product. The automation paradigm, summarized in Figure 2-1, causes developers to continually underestimate the complexity and amorphousness of the design process and the consequent risks.

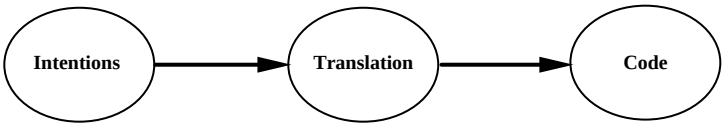

Figure 2-1: The automation paradigm's emphasis on end product accounts for many of the industry's most persistent problems.

The methodological and procedural advances in software development over the last fifty years have been a response to the recurrence of these problems. As important as they are, however, they have failed to fundamentally change the situation. They have not attacked the underlying beliefs that lead us continually to underestimate complexity and change.

As the industry has grown we have become more subtle in our understanding of the paradigm. We know, for example, that intentions start out as fuzzy and ill-defined, and must be clarified through analysis, and validated, and continually revisited. The disciplines of requirements analysis and rapid prototyping, and various alternatives to the waterfall lifecycle model, have grown up in response to this recognition (Patterson 1997). But these disciplines notwithstanding, we still think in terms of clarifying the requirements, and verifying and validating our decisions against the requirements. Although backtracking and iteration are expected, the emphasis is on reaching the end product (Figure 2-2). Development is viewed as a series of translations, from fuzzy to precise, from abstract to machine processable.

2.1 How the Automation Paradigm Governs the Industry

The view of software as automation permeates every aspect of the industry today: from the management structures we expect, to the processes for acquiring turnkey systems, to the legalities of ownership and data rights, and the technical procedures themselves. We describe here some of the ways this influence can be observed.

Management structures. When software is viewed as automation, software development is viewed as a service supporting corporate goals. The nature of a service is to accept orders (or requests) and respond with a product. Software requests take the form, "We need a system that does X," or "We need to change System X to do Y." The goal is the end function.

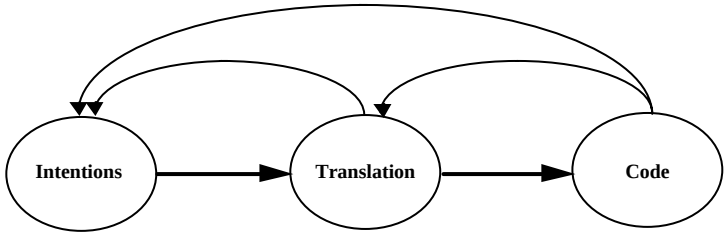

Figure 2-2: Iterative lifecycle models have not altered the emphasis on end product.

The service model reveals itself in top-down management structures. Needs are articulated at the top, then translated into tasks that are passed down as assignments to subordinates. There is often a flow of requests in the opposite direction. Infrastructure needs, for example, are solicited from those in the trenches. Typically, however, these needs are added to a list spanning wider segments of the organization, and are then subjected to the exigencies of allocated budgets. The flow of decisions remains, in essence, top-down.

The ineffectiveness of the top-down process becomes clear when infrastructure, training, or other needs are discovered at inopportune times — for example, midway into a project. In keeping with the service model, such events are attributed to inadequate planning and are viewed negatively.

Preview of a knowledge-creating alternative. What such events are, in fact, is evidence of discovery: new knowledge has been acquired or developed. While the service-oriented organization will see the event as an error or setback, the knowledge creating organization will try to convert it into leverage. In a discovery oriented process, failures and errors are expected and are treated as fuel to drive the expansion of knowledge.

Acquisition processes. The view of software as automation reveals itself in the hand-off approach to system acquisition. Needs and tasks flow between organizations in a top-down manner, similar to the management structure just discussed. This, too, is mitigated by enlightened participants: the wise system development company will keep its customer involved at every stage of decision making. But, in the end, it is the software system that the customer wants; the builders have an interest in maintaining a proprietary culture; so it is the rare project indeed in which the customer is a full participant in the design decisions.

The requirements specification is viewed as the customer's safeguard against a builder's potential abuse of this privilege. Frequently, however, specifications do not communicate the essence of a customer's intentions, or if they do, it is necessarily in language that cannot be considered contractually binding. When a developer then builds to the contractually binding words, the customer may be understandably dissatisfied.

Another intended safeguard is the customer's insistence on receiving design and code documentation along with the software as part of the delivered product. Unfortunately, most documentation (at best) tells *what* but not *why*. It does not explicitly call out the decisions that went into a system design. The decision process, as rich, complex, and fractal as it is — full of dead-ends, miscues, and the influence of multiple stakeholders with

different priorities and different views of the goals — is not easily captured in a product that can be handed off to a customer.

Preview of a knowledge-creating alternative. Imagine if the design record were treated as a first-class product, as important as the delivered system itself. The entire system lifecycle, from planning to operation and maintenance, could be viewed as a sustained dialog about the capabilities of the system. A "living" design record would be consulted and modified as a matter of course as the system evolves. Supporting such a process will not be technologically trivial. As important as the technology, however, is the change in mental models from an emphasis on the end product to an emphasis on the continuing dialog.

Ownership. The knowledge generated as a system is developed has competitive value. It may be in the interest of the developing organization to treat this knowledge as proprietary. When a customer pays for the development of a turnkey system, they may claim rights to any intermediate products created in the process. But they have no effective way to ensure the transfer of knowledge. If the transaction is formulated as buying a system, and the system is viewed as a vehicle of automation rather than as an encoding of knowledge, the developing organization's commitments may be satisfied by a mutually agreed upon set of operational tests.

There may, in addition, be documentation and training requirements. But these typically contain only a very thin slice of the knowledge generated during development. Because the boundaries are not clear, what ensues is a confused dependency relationship between the customer and the developer (or another support contractor who is necessarily less well endowed with knowledge of the decisions made during development). The customer both wants and does not want to be free of the dependency and assume the burden of system evolution. The underlying problem is that the customer never really knows what he is getting.

Preview of a knowledge-creating alternative. A "living" design record, treated as part of the system being bought, might help to alleviate this problem. It is not, however, a panacea. The recent emphasis on product-line architectures for large complex systems could actually be construed as movement in the opposite direction, towards a black-box approach in which less information is provided to the customer.

This movement is, however, quite consistent with the knowledge creating paradigm. On the one hand, product line architectures are an explicit embodiment of domain knowledge. On the other hand, the conformance of individual systems to a product line architecture considerably reduces the customer's risk in buying a turnkey system. The relationship between customer and developer becomes closer to what we see in the world of commercial off-the-shelf products.

Technical processes. The most obvious consequences of viewing software as automation appear in the practices of developers themselves. Software engineering has fostered a design discipline of sorts, but its emphasis has been on the representation of design decisions, not on the reasoning process through which the decisions are reached. The widely known methodologies certainly recommend identifying and evaluating design alternatives, but they do not offer procedures to do so systematically; they provide no

guidance for representing this information in a form that can be monitored, reassessed, and changed over time.

An even greater problem is the delegation of low-level design decisions to the coding process. This is a phenomenon that can hardly be appreciated, or even recognized, by anyone who has not coded a complex system. The entire notion of "coding to specification" is based on the fallacy that critical decisions stop when coding begins. Numerous choices are faced in the process of coding, many of which have significant impact on the performance, maintainability, and even the proper functioning of a system. The variations of alternatives can be numerous, and the tradeoffs between them subtle.

These choices do not surface during the recognized design process because they occur at too low a level of abstraction: they may not even be expressible in the adopted design notation. Attempts to drive the design process down to a level comparable to code have typically resulted in program design languages, or pseudocode. These have proven unsatisfactory for two reasons. On the one hand, "design" in such languages consists of no more than an initial coding effort, rather than a thorough exploration of design alternatives. On the other hand, because this "design" process is correctly viewed as a form of coding, it is seen by many as superfluous.

The result is that coders become, by default, critical designers. Yet they work in the belief that the critical design decisions have already been made. This may be true even of conscientious programmers who take pains to create maintainable code and document its logic. They do not see themselves as decision makers in a continuum that started with the initial planning of the system. At best, their documentation will support intellectual commerce with future maintainers or modifiers of the code, but not with the community of stakeholders as a whole. The value they produce is understood to lie in properly operating code, not in the trail of choices made along the way.

Preview of a knowledge-creating alternative. When programming is viewed as a process of raising issues, identifying and evaluating alternatives, and deriving decisions, then coding will really be just coding, the relatively routine process of writing in the syntax of a particular programming language. This is the goal of Literate Programming, a discipline that beautifully exemplifies the idea of software development as knowledge creation (Knuth, 1992).

3 Software as Models of an Enterprise

The growing use of object-oriented methods has fostered a change in beliefs about the role and nature of software. Software is coming to be viewed as more than a way of commanding machines; it is understood as a way of modeling an organization's structure and function. Trends such as enterprise modeling and business process reengineering have helped spread this understanding to the non-technical participants in development organizations.

There is no need, in this paper, to rehearse in detail the motivation and rationale for object orientation. It is a way of making systems more adaptable to change, and it achieves this goal by explicitly representing knowledge that is left implicit in functional approaches. The object-oriented approach requires developers (at all stages of the system lifecycle) to ask such questions such "What are the objects, what services do they perform, and how do

they interact?" In addressing such questions, developers must think beyond the immediate operational goals of the system. They must consider the kind of context that we discussed in Section 1.1.

There is a conundrum, though, in the modeling process, concerning the distinction between models as a representation of something that already exists, and models as a way of articulating thoughts. The first meaning is the usual interpretation. Modeling is understood as something similar to photography (or perhaps representational sculpture or painting): one sees what is out there, and one represents it in the chosen medium which in this case is software.

It is the second meaning, however, that more accurately describes object-oriented software development. As soon as one starts designing a software system, one begins to propose and experiment with objects that do not represent any previously recognized external entity. The definition of such objects cannot be verified as accurate representations; they must be assessed, instead, in terms of the goals of the system, such as performance, maintainability, reusability, and other quality factors. The question for designers, then, is not whether the model is correct, but whether it is useful. And that is a far less cut-and-dry issue (see, for example, the treatment of *purpose* as a dimension distinct from structure and function, in Sage, 1995).

The matter is further complicated by the fuzzy boundary between "things that already exist" and "proposed abstractions." The context modeled by a new system usually includes already existing systems that support the organization's practices. Perhaps the new system will interoperate with these systems, or perhaps it will replace one or more. The existing systems may have introduced their own artifacts, as well as human procedures, in their model of the enterprise. When the earlier systems were built, these artifacts may have had the status of abstractions because they did not correspond to any previously recognized entity in the organization. But as a system becomes established within an organization, the artifacts that it manages become recognized as part of the organization. Reengineering an organization involves asking whether these artifacts and procedures are (or remain) essential. This too is not a cut-and-dry issue.

Our conclusion is that modeling, whether object-oriented or of some other form (such as rule-based), is not photography. It is not simply a process of representing observations. It is, rather, an experimental process in which models are proposed, evaluated as to their usefulness, refined, rejected, or replaced. The resulting model, represented in a set of software classes and objects, is (like the documentation discussed in Section 2.1) only a thin slice of the knowledge generated along the way — knowledge that is required in maintaining the model as an organization evolves.

4 Software as Knowledge

The idea of a knowledge creating organization was introduced by Nonaka (1991, 1995). It is closely related to concept of the Experience Factory (Basili *et al,* 1994). In this section, we describe how software development may be viewed as a knowledge creating activity.

In certain key respects, software development resembles science and mathematics more than engineering. The proposal and evaluation of alternative models is a scientific process.

The definition of layer upon layer of abstractions, each of which becomes assimilated over time into the common vocabulary, is similar to the progress of mathematics. These two features, experimentation and continual abstraction, distinguish software development from other engineering disciplines. They are a direct result of the role of software as a complexity sink, as the medium of choice for implementing unprecedented functions.

Complexity and novelty introduce unknowns; as the software developer grapples with them, knowledge is created. The unknowns, which are the source of most risk in building a system, fall into three broad categories, shown in the first column of Table 4-1. A software development project of any significant complexity will face all three types of uncertainty. As shown in the second column of Table 4-1, software engineering offers methods for managing each type of uncertainty.

Not knowing exactly what we want	• Requirements analysis • Rapid prototyping
Not knowing all relevant legacy	• Software reuse
Not knowing the solution to a new problem	• Technology innovation and transfer

Table 4-1: Software engineering attempts to address each major type of uncertainty.

The problem is that we view these methods as countermeasures. Their value, in the conventional view, lies solely in how well they guide us to the end product. The "heat" generated by the activities — the knowledge produced along the way — is treated as a by-product and is rarely harnessed.

This orientation has an unfortunate consequence. In a world of cost and schedule constraints, the emphasis on the end product gets translated into a self-defeating proposition. Knowledge creation activities are minimized so that a project can reach the end product on time and within budget. The simplistic formula of translating intentions into code (Figure 2-1) wins out.

This process is self-defeating because, as we have tried to show, the knowledge creation activities cannot be avoided. By not harnessing the "heat" that they generate, we sentence ourselves to rediscovering the same mistakes and the same solutions over and over. We increase the likelihood that suboptimal decisions will be made because knowledge was incomplete. We cripple the sustaining engineering process, because the software to be maintained and evolved is documented as a set of unattributable facts, rather than as the result of careful reasoning.

The knowledge creation paradigm starts from the premise that the "heat" generated during software development has value. The product is not just the software system itself but also all of the knowledge generated in building, operating, and maintaining the system. This includes a trail of the key decisions made, their rationales, and the reasoning and criteria that went into the decisions. It includes lessons learned in the development process, experience gained over time through use of the system, and links to the broader base of knowledge acquired over time by the development and customer organizations.

The challenge of software development technology, in this paradigm, is to enable developers and other stakeholders in the process to acquire the knowledge they need, when they need it.

4.1 The Role of Discovery

It is the nature of software development to produce surprises: almost anyone who has had practical experience developing a complex system will agree. Nevertheless, this persistent characteristic is most frequently seen as something to be overcome. Surprises are considered to be evidence of inadequate planning, and their avoidance essential to a well-managed project. Recognizing that surprises will inevitably occur, planners may budget resources for unanticipated events, or they may factor a margin of error into their estimates. In this way, they try to estimate the unknown. If they are wrong, the project suffers.

In an industry that is trying to become more mature as an engineering discipline, it seems anathema to suggest that all software development is, at root, experimental.[1] One sign of maturity, however, is a respect for facts, and the persistence of surprises should indicate that something other than bad planning is going on.

In the knowledge creating approach, we see the development of a software system as a process of answering a long series of questions — a gradual reduction of uncertainty. We can think of this as an image (our image of the desired system) gradually coming into focus. A GIF image loaded over the internet provides a good example: the image is displayed as a set of tiles that is at first course-grained and offers only a crude resemblance to the desired picture; over time, the tiles becomes finer, more details are filled in, and the image becomes clearer and more accurate.

The questions begin at the planning stage (What do we need? Can we afford it?) and continue through implementation, test, operations, and maintenance. Uncertainty rules at each stage. Even seasoned programmers, for example, will find themselves painted into a corner as a result of a coding decision, and will have to backtrack to a different implementation approach. This happens because the network of design interdependencies is too complex to foresee all of the implications in advance. Sometimes the only way to understand the consequences of an approach is to try it out.

This is not poor engineering; it is disciplined experimentation. The successful developer is one who optimizes the process by quickly recognizing the need to backtrack, and by learning from each such event so that it need not be repeated.

A positive form of opportunistic discovery occurs when the developer recognizes a new abstraction: a technique developed as an *ad hoc* response to a task is recognized to be more broadly applicable. The unexpected discovery of abstractions lies at the heart of software evolution. Reuse and domain engineering methods must accommodate this phenomenon if they are to be taken seriously by working software developers.

---

[1] The author owes the observation that software development is inherently experimental to Professor Vic Basili of the University of Maryland.

4.2 Relationship with Software Reuse

The relationship between "software as knowledge" and "software reuse" is captured by a concept called Difference-Based Engineering (DBE). DBE emphasizes the building of *deltas* rather than entirely new software. The essential idea in DBE is to match the current task as closely as possible to something (or several things) that the development organization has done in the past. The current task is then performed by modifying, as necessary, the products of the previous work. The modification is a *delta* over the original products.

From one perspective, DBE may be considered as systematic software reuse. The emphasis, however, is different from the usual connotations of *reuse* as that term is often applied. DBE involves the reuse not just of software components or other lifecycle products (such as requirements models) but of any form of packaged knowledge that has been acquired through the processes of discovery and learning. These knowledge assets are represented in the organization's Experience Base. Unlike the widely held view of reuse, DBE emphasizes the differences between past efforts and the current task. By assuming that the Experience Base may not contain an exact match to the current task — but advocating building on the closest matches anyway — DBE provides a greater scope of reuse, and a closer integration of reuse with conventional engineering activities. The conventional disciplines can be focused on understanding the differences between the current task and previous efforts, and on evaluating alternative approaches to bridging those gaps.

DBE can be described as a generic problem solving paradigm. A task order arrives with a set of requirements, for a new software system. The software engineers interpret these requirements as a set of required features; in addition to requirements for the system itself, there may be other constraints that the engineers have to meet, and these too are interpreted as features. With the features as indices, the engineers search the Experience Base for the organization's most relevant knowledge, including past systems that share many (if not all) of the current requirements, or systems that share some especially important requirements with the current task.

The software engineers then assess the relevance of experience records they have found. In the best case, the organization has performed an essentially equivalent task. The previous product can then be reused, perhaps with minor modifications. If the organization has performed similar but not quite equivalent tasks, the engineers try to articulate the differences between the current task and the past accomplishments.

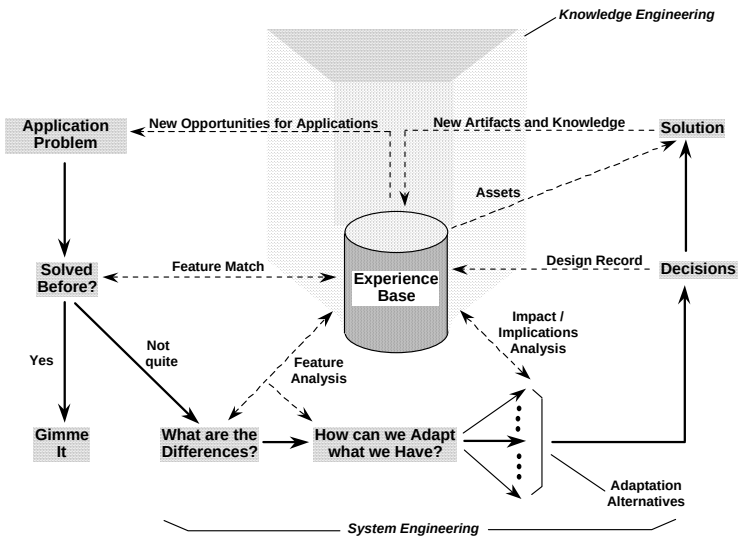

Figure 4-1: The Difference-Based Engineering model places software reuse in a broader context of knowledge creation.

They then investigate how the organization's past accomplishments can be modified to meet the current challenge. Typically. they will develop a number of alternative approaches. As they define the alternatives, they study the implications of each approach and the tradeoffs between them. On the basis of the tradeoffs, they select an approach. This is the classical engineering process, which lies at the center of DBE. The key observations of the analysis process become part of the new system's design record and are fed back into the Experience Base (see Section 4.3).
As the new system is developed, more decisions are made. These too become part of the design record, and are incorporated in the Experience Base to guide future efforts.

As this summary shows, the development organization must actively promote the incorporation of design records in the Experience Base. It must also continually filter, structure, and package the information in the Experience Base so that it is accessible and usable by engineers. In Figure 4-1, these responsibilities fall under Knowledge Engineering, which plays a role similar to Domain Engineering in more familiar scenarios of software reuse.

4.3 The Role of Design Rationale Capture

Design records play an essential role in Difference-Based Engineering. As we observed in Section 2, a "living" design record is one sign of the knowledge creating approach. What distinguishes a living from a moribund design record? Evolution is one criterion: the record must evolve with the software system through all phases of the lifecycle.

There are, however, other criteria. If the design record is viewed as an administrative burden, it may be maintained as a required form of documentation, but no one will ever use it The distinguishing feature of a living design record is that participants have a stake in it.

All stakeholders in the system being built (or maintained) should feel engaged with the content of the design record.

As a way of fostering such engagement, we offer the notion of a *design narrative*, which is an extension of the ideas of Literate Programming (Knuth 1992). Knuth's contention was that program documentation should read like literature, if the author expects other developers to pay attention to it. The notion of design narratives takes this a step farther by suggesting a story-telling approach:

> The design record should tell the story of the decision process, complete with characters (stakeholders and systems), objectives (system requirements and stakeholder goals), plot drivers (risks, issues, and obstacles), conflicts (alternatives and tradeoffs) and resolutions (decisions). The software design will take shape as a compelling and continually embellished story that has purpose and meaning to those who make decisions about the system.

As a paradigm for structuring the design record, narratives offer several benefits. They integrate a stakeholder model into the documentation of design rationales. They encourage discussion of rationales in the context of stakeholder beliefs, goals, motives, concerns, frustrations, and priorities (see, for example, the use of dramatic scenarios in Bailin *et al*, 1996). They encourage creativity in imagining design alternatives (see, for example, the discussion of micro-worlds in Senge, 1990).

The narrative structure raises the stakes of the design process in the eyes of the developers by engaging them in a dialog, inviting them to become actors in the ongoing story, highlighting and giving substance to the implications of decisions they have to make. (Similar themes, in the context of computer-user interfaces in general, are discussed in Laurel, 1993.)

Construction of a design narrative takes considerable effort, and it will only occur if the narrative, rather than the software itself, is viewed as the primary product. Such a change of mindset would constitute a major shift in beliefs on the part of developers, project managers, and customers. Belief shifts, however, always encounter resistance. The most common form of resistance to this particular change is the claim that it is not practical.

A change of this sort was, nevertheless, the original goal of Literate Programming. As a way of fostering the change in mindset, Literate Programming advocates the use of tools to reduce the burden of additional documentation. Tools are used to convert literate programs into either compilable code or readable documentation. The developer is expected to produce only one type of artifact, the literate program itself.

In the more ambitious realm of design narratives, multi-media tools can encourage the capture of design rationales, their integration into a narrative, and their playback to support system evolution. Voice annotations can capture the reasoning of a programmer, who is perhaps weighing the merits of a set of design alternatives, without requiring him to take time out from the evaluation process. Video clips of meetings can capture inflections, such as non-verbal cues or body language, undiluted by translation into meeting notes. Multi-media presentations can provide a sense of immersion in the design story to someone tasked with modifying a system.

These techniques are experimental, and work is underway (by the author as well as others) to test their feasibility. None of them will succeed, however, if we lose sight of the primary goal: to treat software development as a process of cultivating knowledge, as opposed to a process for encoding commands.

5 Conclusion

We have argued that software development is essentially a knowledge creating activity; our failure to treat it as such has resulted in the symptoms that we refer to, collectively, as the software crisis. The view of software as a vehicle for automation permeates every aspect of the industry. It leads to deeply ingrained practices that software engineering methodology has tried unsuccessfully to change. The emphasis on engineering has, ironically, become part of the problem because it tends to occlude the more scientific and exploratory aspects of software development. The shift towards object orientation has highlighted the importance of modeling in software development; unfortunately, though, modeling is too often understood as simple representation, rather than as a scientific process of hypothesis and evaluation.

We have described the key features of a knowledge creating software development organization. They include a respect for discovery and learning as core design processes, a discipline of building upon previously acquired knowledge, and an emphasis on capturing the reasoning behind design decisions.

References

Bailin, S., Simos, M., Levine, L., and Creps, R. (1997). *Learning and Inquiry-Based Reuse Adoption (LIBRA): A Field Guide to Reuse Adoption through Organizational Learning*. STARS Informal Technical Report No. STARS-PA33-AG01/001/02.

Basili, V. R., Caldiera, G. and Rombach, H. D. (1994). "The Experience Factory." *Encyclopedia of Software Engineering*, Wiley, New York.

Knuth, D. E. (1992). *Literate Programming*. CSLI Lecture Notes, no. 27, Center for the Study of Language and Information, Stanford, California.

Laurel, B. (1993). *Computers as Theatre*. Addison-Wesley, Reading, MA.

Nonaka, I. and Takeuchi, H. (1995). *The Knowledge-Creating Company: How Japanese Companies Create the Dynamics of Innovation*. Oxford University Press, Oxford.

Patterson, F. G., Jr. (1997). "Systems Engineering Life Cycles: Life Cycles for Research, Development, Test, and Evaluation (RDT&E); Acquisition; and Planning and Marketing." In A. P. Sage and W B. Rouse (Eds.), *Handbook of Systems Engineering and Management*. Wiley, New York (forthcoming).

Sage, A. P. (1995*). Systems Management for Information Technology and Software Engineering*. Wiley, New York.

Senge, P. (1990). *The Fifth Discipline: The Art and Practice of the Learning Organization*. Currency Doubleday, New York.

U. S. House of Representatives (1989). *Bugs in the Program - Problems in Federal Government Computer Software Development and Regulation.* Staff study by Subcommittee on Investigations and Oversight, Committee on Science, Space, and Technology, August 3, 1989.

Acknowledgements

The author has learned much about these issues from discussions with Mark Simos and Larry Levine of Organon Motives, Inc., and with Richard Evans of Engineering Research International.