

# Improving Adoptability by Preserving, Leveraging, and Adding Cognitive Support To Existing Tools and Environments

Andrew Walenstein  
Software Research Laboratory  
Center for Advanced Computer Science  
University of Louisiana at Lafayette  
walenste@ieee.org

## Abstract

*Being adoption-centric means focusing research on what technologies would be helpful to real users and trying to ensure that the results are more likely to be adopted. Too little is known about how to improve adoptability. This paper describes preliminary steps towards a framework for understanding methods for injecting innovations in a way that makes the results more likely to be adopted. The framework defines taxonomy of adaptations that tools and users undergo in the face of innovations. It then employs theories of distributed cognition to suggest which potential adaptations would be considered potentially desirable to users because they preserve, leverage, or add cognitive supports. An example is given illustrating how this framework is being used in exploratory design.*

## 1. Introduction

The rate at which practitioners adopt the products of software engineering (SE) tools research suggests that much improvement is possible in making research results available and adoptable. In some cases the lack of adoption may very well be due to the way that software research prototypes are developed. Frequently a simple, stand-alone “demonstration” implementation is developed. Often this is a bare-boned and impoverished environment or tool when compared to the robust, full-featured, and highly usable tools and environments that practitioners normally work with.

An alternative approach to tools research is to employ an “adoption-centric” approach of building innovations as adaptations of the rich tools and environments currently existing in practice. Here, “tools” and “environments” are considered broadly, and would include editors, shells, browsers, word processors, personal information managers,

and ordinary software development environments. An adoption-centric approach would perform the tools research with a concern for easy adoption of the tool by some real user community.

Several potential advantages can be offered for this approach. First, reusing an existing environment can make prototype development easier since the researchers do not need to spend time implementing and perfecting common but necessary infrastructure (undo, copy and paste, printing, help, etc.). Simple, bare-boned “toy” tools frequently miss these features or implement them awkwardly. This will normally seriously affect user performance and satisfaction, which will in turn make successful evaluation of the innovation exceedingly difficult. Second, by using an existing toolset one is more likely to find a user population to evaluate the tool on.

Being adoption-centric while adapting existing systems means that close attention will need to be paid to the ways software engineers currently work, and to how innovations can be fit into this work. This point is, in my opinion, the most critical aspect of the approach, and the place where the greatest research benefits can be expected. Being concerned for current work practices grounds the entire research effort in real user needs and situations. In addition, being concerned with how the innovations fit into real work helps assure that the research is in a position to make practical impacts sooner than the 17 or more years that some SE innovations have taken [12]. The adoption-centric approach is therefore not merely another attempt to build extensible development environments or to implement specific SE tools on top of other tools. There is a real concern for making innovations match realistic scenarios, and for introducing innovations to practical environments in ways that are likely to be more readily adopted. Attention to other factors such as marketing and organizational structure may also aid adoption (e.g., Fowler *et. al* [4]), but the prototype implementation is the main adoptability factor under the direct

control of most tools researchers.

Once the above general goals of adoption-centricity are stated, however, the question of how to actually go about achieving them looms large. How and why users adopt new technologies is not well known, and even less is known regarding how, exactly, one can build innovations that are more likely to be adopted.

This paper outlines a general framework for understanding adoption factors and recognizing opportunities for implementing innovations in ways that are more likely to be adopted. There are three main components to the framework. The first component is a taxonomy of types of adaptations, both for tools and users. This taxonomy is presented in Section 2. The second component is an analysis of how to interpret adaptations—and the resistance to such adaptations—in terms of adaptations to distributed cognitive systems. This analysis is presented in Section 3. The third component is a technique for analyzing existing toolsets for opportunities to inject new technologies in a ways that are likely to be adopted. The way this is currently being approached outlined in Section 4. Section 4 also outlines how the framework is being considered in a project relating to software clone detection and copyright violation litigation.

## **2. Types of adaptation in adoption**

Users *adapt* to new tasks and technology. Such user adaptations include learning new concepts, skills, and problem solving techniques or strategies. As Mackay [11] pointed out, users and their environment actually *co-adapt* (also see Fowler *et. al* [4]). Users adapt their tools and environments to better suit their tasks and individual characteristics. Tool adaptations along these lines include setting key bindings, scripting, and programming. Users also adapt their entire information space in order to help solve problems. For instance software engineers implement file naming conventions in part because this makes their browsing and searching tools effective [7].

When any new technology or innovation is adopted by users, it means they adapt again to the changes. It seems likely that these adaptations could be effected in fundamentally different ways. A vocabulary for describing the different forms of adaptation is desirable. This section extends Mackay's analogy by using concepts from biological evolution to understand tool and user adaptation types.

### **Biological evolution and adaptation**

Biological evolution can be seen as an extended process of adaptations to changing conditions. A naive conception of evolution is that it makes steady progress towards organisms of greater complexity and fitness. It is true that some

organism features are associated with a history of gradual and incremental refinements of similar-but-less-fit features. For example, Dawkins reconstructed a history of how complicated eyes were built out of a series of additions and refinements of previous structures [2]. Even so, the fossil record also suggests that evolution should not be exclusively characterized as a uniform process of gradual refinement. The late evolutionary theorist Steven Jay Gould conspicuously argued that the evolutionary history is “punctuated” with periods of alternating relative stability and astonishingly rapid and wholesale changes which include radical changes to basic organism design [5]. This sort of distinction in evolutionary progression resonates with certain theories of knowledge acquisition and learning [14] which posit differences between learning by “accretion” and by “restructuring”. Accretion occurs when the knowledge can be absorbed with only minor changes to the existing knowledge structures, whereas restructuring is made necessary by concepts and data that cannot be accommodated within the existing structures.

Why do adaptations even occur? Adaptations occur, at least in part, as responses to changes in living environment (e.g., climate). Adaptations in this sense improve the fitness of an organism to some ecosystem. Sometimes these adaptations are specific to particular ecosystems—the organisms become “specialists”. An example is the giant panda, is adapted to survive on bamboo shoots and nothing else. In contrast, some organisms are generalists and can survive well in many ecosystems. An example the brown bear, which is omnivorous and ranges very widely.

How do new adaptations arise? One part of the story is simply by design variation through mutations which result in improved fitness to the ecosystem. Another part of the story is that an organism's existing features might be used for additional or new purposes. Gould called this “exaptation” [6]. An example he uses is how feathers may have originally provided warmth, but were eventually a step towards achieving airflow.

### **Applying the evolution metaphor to technology adoption**

It is possible to see biological evolution as a metaphor for user and tool adaptation. Based on the above discussion, three basic contrasts might be helpfully applied to classify user and tool adaptations.

First, user and tool adaptation may be divided into gradual accumulation of design changes, and rapid, wholesale changes. The former is common in the slow evolution of product lines (e.g., creeping product features). Users also gradually adapt by learning different problem solving skills and tool features. Wholesale and rapid changes occur when users adopt radically different tools such as new operating systems, development environments, or office products.

Users are often painfully aware of how they need to adapt to such wholesale changes. Thus a first question for adoption-centric SE is “what type of adoption is being attempted: gradual accretion of localized design variations, or wholesale design changes?” The adoption-centric researcher must know which is being attempted—and which is needed.

Second, the issue of specialization versus generality needs frequently be considered for tool design. Specialized tools may be more fit for certain tasks but require specific learning (user adaptation) and may not “survive” changes in tasks. Task specificity is a common argument for or against various programming languages. An advantage of generalized tool capabilities is that once users learn them they can apply them in many situations. The drawback is that the general capabilities might work less well than the specialized versions, or users may need to do more work, or to customize them. In terms of adoption-centric design, the researcher should likely be encouraged to recognize specialized and generalized capabilities in both tools and users and take advantage of both when opportunities present themselves. For example, in certain work domains, programmers may be specialized to be highly familiar with spreadsheets. This specialized expertise might be exploited by implementing the innovation as an extension to a spreadsheet program. But also note that a spreadsheet itself is general in that it can be applied to many different tasks (compare, for example, a tool that performs fixed calculations).

Third, it may be helpful to distinguish two different classes of adaptations to existing tools, or aspects thereof. The first is by simple design mutation or accretion. For example, adding a call-graph visualization view to a toolset might be termed simple accretion, whereas changing the way error messages are displayed might be considered mutation. The second main class of adaptation type is by exaptation. Tool based exaptation might be said to occur if new uses are made for existing specialized functionality. For example, in Mackay’s study [11], mail filtering functionality was used to implement smart filing of messages. In the realm of software development, Bellamy [1] noted that Smalltalk developers would use a cross-referencer as a way of locating semantically-related code. The developers would “tag” class methods as belonging to an application by inserting references to a dummy class. The cross-referencing was therefore not being used to trace down real calls, but to approximate a mechanism for clustering conceptually-related methods from multiple classes.

The adoption-centric researcher will want to be aware of whether changes are being made by mutation, accretion, or exaptation. One reason for wanting to know this, clearly, is that the adaptations made to tools are likely to induce similar types of adaptations to the users’ knowledge of how to use the tools. Tool mutation implies that the user must adapt by modifying their skills and mental models for using

the affected features. Accretion, on the other hand, is likely to allow simple knowledge accretion by the user. Exaptation is relevant to tool researchers because users may already be skilled in using the tool feature that is being exapted for a different purpose, or they may be able to exapt an existing feature or technique for use in conjunction with the new innovation.

### **3. Distributed cognition & legacy user systems**

I am a *legacy user*. So, in all likelihood, are you and everyone else. My favorite editors are Emacs and vi. This fact might be viewed with considerable disdain by combatants on both sides of the long-running “vi versus Emacs Editor Wars”. I use both editors practically every day for writing papers, programs, email, and numerous other activities. Many other and newer editors exist—certainly many specifically tailed for programming. Some of these, perhaps, are even superior to both of Emacs or vi (at least for some tasks and in some ways), although I may never truly know it. Myself and my computing environment in combination form a *legacy user system*.

I use the term “legacy user” in the way a software maintainer would expect: a legacy user is analogous to a *legacy software system* in SE. This term is intentionally selected. Cognitive science regularly views human minds as computer systems. Learning (i.e., user adaptation) serves to program and maintain the cognitive system.

The term “legacy user system” is no accident either. The cognitive science field of distributed cognition (DC) treats cognition as a computational process distributed between humans and tools (see Hutchins [9], Zhang *et. al* [17]). Thus users in combinations with tools are seen to form DC systems. From this point of view, external artifacts are seen to represent knowledge or cognitive states (goals, intentions, etc.), and both users and computers are seen to process such external knowledge and cognitive states. For instance, Flor *et al.* [3] analyzed programmer pairs from the DC point of view. In their analysis, they likened code scavenging to case-based knowledge use: when code is scavenged, it is copied with appropriate modifications, which is an analogue of schematic abstraction and instantiation. Only instead of occurring “in the head”, it occurs in a text editor.

Legacy user systems are also computational systems: *legacy* computational systems. Legacy systems are not necessarily poorly maintained systems. Instead, they are identified by other characteristics: they are typically (1) considered “mission critical” for the organization using them, (2) not implemented using the most up-to-date technologies, although for various reasons it is desirable to bring them into compliance, and (3) poorly documented and understood. These are all characteristics of legacy users systems; the term is apt:

1. The DC system as a whole is critical for effective work. Clearly the user's own mind is mission critical, but just as clearly their normal environments are critical (or else adoption would not be a problem—they would be able to effectively use whatever environment is in front of them).
2. Updates to the legacy DC system is frequently desired and, in the case of adoption-centric research, assumed necessary.
3. The DC systems are almost entirely undocumented. As Hollan *et. al* [8] point out, the roles that artifacts play in cognition are often difficult to recognize. Careful field studies are therefore frequently needed in order to *reverse engineering* and *redocument* legacy DC systems in preparation for reengineering. It is well known that humans generally are unable to articulate how it is they think and act, and they certainly do not come with cognitive design documentation.

The main value in bringing DC theory into the present discussion is that it identifies aspects of legacy user systems which are important for effective distributed cognition.

More specifically, users rely on their external environment to provide *cognitive support*, i.e., to assist or help them in their cognition [16]. This support is partially attributable to the makeup of the tools. For instance most web browsers maintain link visitation history mechanisms, and will display the visitation status of links by rendering non-visited links in a different colour. Those features can support users by acting as external memory: users no longer need to remember where they have been. In other cases the support can be said to be “built up” in the environment through the various adaptations and customizations they make. For instance, the collection and organization of bookmarks is a lasting external memory that users often depend upon. Another example, already mentioned above, is the tagging of methods which was observed by Bellamy. These tags are needed if the programmer is to use the code location tools they are accustomed to. Over time, user and environmental adaptations generate a DC system in which the tools and their features support cognition, and in which the users have the skills, knowledge, and preferences for utilizing them effectively. Users are reluctant to adopt new technologies because doing requires new adaptation (learning), and may destroy the cognitive support built up in their environments, or make it less efficiently usable.

This analysis is helpful because it delves a little deeper into the barriers to adoption. Existing theories of adoption are compatible with this basic analysis (at least, the parts dealing with the adoption factors associated with individuals). For example the so-called “diffusion of innovation” theories [13] posit that individual evaluations of “usefulness”, “compatibility” and “ease of use” greatly influence

decisions to adopt. But what, precisely, is “usefulness” and “compatibility” and how can it be identified in tools? The answer I am working towards is a partial one, but it is a step in the right direction. Usefulness is a function of the cognitive support provided, and “compatibility” should mean, in part, the retention of built-up cognitive support. To make innovations more adoption-centric, therefore, one needs to reverse engineer and redocument existing legacy DC systems, and then build tools that can reengineer them in ways that retain and build cognitive support.

#### **4. Framework application**

The overall aim of the present framework is to help in reengineering existing legacy user systems. Below is a brief outline as to how this might happen in the future. The basic method is to first use the DC ideas to either guess or empirically determine the cognitive infrastructure (i.e., adaptations) users have built up in themselves and their environments. Currently we base this on an inventory of cognitive support possibilities derived from a feature analysis of the environments. Then opportunities are examined for adapting existing features in ways suitable for introducing the new technologies. This general idea is explored below.

#### **Eclipse and clone detection**

At the Software Research Laboratory one of our projects is to investigate techniques for detecting software clones, copyright violations, and plagiarism. Software clones are sections of code that are very similar. These commonly occur because of code “scavenging” in which code is copied and then modified to suit local needs. Copyright and plagiarism cases involve finding and verifying the fact that code or design aspects were copied from one code base to another. In each case, code similarities of various types need to be detected and investigated.

Our research is taking an adoption-centric approach to developing and inserting suitable technologies into practices of SE and copyright litigation. This will result in separate tools for software engineers and for legal analysts, but we are planning and developing both sets of innovations on top of three main existing technologies: Eclipse, Microsoft Office tools, and Microsoft Windows<sup>1</sup> These platforms are suitable starting points as our anticipated user base is expected to be familiar with them. In particular, in legal cases the users are expected to use Office tools such as Word, Excel, and PowerPoint in the generation of legal documents and presentations. We expect synergy in our work on both clone detection and copyright violation.

Our research strategy has identified three technological additions that are intended to implement three activities.

<sup>1</sup>Eclipse, Office, and Windows are registered trademarks.

FEATURE	DESCRIPTION	REUSE	MUTATE	ACCRETE	EXAPT
perspectives	customizable/sharable views and visible actions	new projects new tasks	new cmdnd	embed view auto generate new type new views	search metadata
wizards	steps users through common tasks				
OLE integ.	editors, views, and toolbar objects can be embedded				
Task View	users can edit and step through task (i.e., to do) list				
Search View	multiple search types, results filtering, sorting				
Compare View	two files can be compared side by side				

**Table 1. Inventory of some Eclipse features and some adaptation possibilities**

These additions address activities typical in reverse engineering [15]. First, data must be gathered on where code similarities occur. For this we wish to introduce various automated and semi-automated code comparison technologies. Second, similar items must be classified into clone or non-clone (or copy or non-copy), and then grouped or aggregated into function-relevant clusters. For example, when reengineering a software system the engineer may wish to cluster related clones together so that new modules can be generated by abstracting related and duplicated functionality into a set of related methods. In copyright litigation contexts, the clustering might pertain to collecting together different types of copyright violations (e.g., code copying versus design copying). Third, exploration, visualization, and evaluation must be performed. For instance, the overall distribution of clones across sub-project boundaries may need to be known by project managers. Likewise lawyers will want to see visualizations and analyses of the instances and extent of copying. We are working on novel visualizations for code comparison and system visualizations. We wish to add various automated measurements.

In order to continue forward we need to be able to analyze Eclipse, Office, and Windows so that our innovations can be well matched to these technologies, and can be inserted in a manner that eases barriers to adoption. The remainder of this section describes an approach we are considering for evaluating tools for adaptation possibilities and cognitive support roles. The work is ongoing and preliminary, but the overview below gives a flavour of the type of analyses we are considering. The overview may give others ideas as to how to improve the analysis, and to apply it to their own adoption-centric SE research.

A cursory inventory of Eclipse features yielded a list suitable for determining adaptation possibilities. A subset of this inventory appears in Table 1. This list of features were then examined to see what adaptation mechanisms were provided by Eclipse. These were categorized using the adaptation taxonomy; examples are listed in Table 1. The column “reuse” indicates an instance where a generic feature might be used for new purposes. The entries in these columns indicate ideas about how the adaptations might be made (e.g., adding a new command to the `Task View` mu-

tates it). The inventory and classification generation took about an hour to collect, although we refined this list and its categorizations after various discussions. It is unclear at this time how helpful this exercise has been, although obviously *some* similar type of analysis (perhaps a more informal and haphazard one) would need to be performed if one is to build tool extensions. It may be worth noting, however, that the columns of Table 1 may be helpful in identifying adaptations that are less disruptive to existing cognitive support.

The features in the list were then examined in light of various theories of cognitive support [16]. The aim was to help understand their potential roles in supporting developer cognition. Although this is hardly a substitute for studies of real users (users may not actually use the support, or may have many other types of built-up support not knowable through armchair analyses), it seems to be a prudent first analytic step. The next analytic step is to consider how we might best implement our planned innovations on top of this infrastructure. In this step we expect the theories and models of cognitive support to be helpful, although we have little to report at this point. Nonetheless a flavour of the analysis can be relayed.

We know that current technological limitations make it impossible to automatically and accurately detect all software clones within a system. Thus the user must cooperate with the tools in order to jointly determine which potential clones—i.e., “clone candidates”—should be considered true clones. The classic way of doing this is to have one or more clone detectors generate a list of clone candidates that the user steps through and classifies as clone or non-clone (or copy vs. non-copy in copyright tools). It seems clear that the generic `Task View` functionality can be adapted: the clone detector would generate a clone candidate list in the task list (an accretion of functionality), and the user would need to step through the task list and examine the candidates, deleting ones that are non-clones. This would, in fact, require a mutation (in particular, a specialization) of the task view functionality since the potential clones are clone *pairs* and the `Task View` behaviour would have to be modified so that it browsed to the two clone locations when the user double-clicks on the clone candidate.

The DC support point of view notes that this functionality is an example of distributed planning and plan following [16]. What the clone detector is doing is generating a *plan* for checking the results, which the user (more or less) follows to make those checks. The `Task View` functions as an external memory for the plan, and for where one is in following the plan. Reusing the `Task View` effectively *leverages* existing cognitive support. Once this is realized, opportunities for improving distributed plan following can be explored. For instance, the above envisioned extension forces the user to make a series of decisions about clones. It is likely that ordinary users will not be able to decisively classify clone candidates in the first pass. Said in other words, they will likely have uncertainty in their decisions. It makes sense to try to *add support* for uncertain knowledge management by allowing the uncertainty to be externalized [10]. Otherwise the engineer will need to remember the uncertain clone pairs in order to return to them, if necessary.

Various designs can be entertained for implementing the uncertain knowledge management support. Table 1 provides clues as to which ones might affect adoptability. For instance, the `Task View` might be mutated in order to encode the uncertainty in the classification. It might be preferable, however, to create a more specialized version of the `Task View`. Since there are already several specializations of the `Task View` (the `Compare` and `Search` views are both specialized task steppers), it may be preferable to add an entirely and obviously new view (that is, by accreting similar functionality) that allows clone candidates to be classified with varying degrees of uncertainty. Although we are nowhere near being able to decide what implementation is best, the vocabulary of cognitive support and adaptation taxonomy appears to be helpful in understanding design options and then reasoning about their potential adoptability implications.

## 5. Summary

In order to achieve the goals of adoption-centric SE research, one must have an idea of what factors affect adoptability and what changes can or should be made to existing environments in order to introduce innovations. The direction presented here is to examine the cognitive support present in target tool environments and look for appropriate support to preserve, leverage, or add. Although progress has at times seemed glacial, the cognitive aspects of adoption resistance appear critical, and my personal feeling is that there is no choice but to continue the difficult and long-term work necessary to understand and address the role of cognitive support in tools and how various adaptations to them affect adoption.

## References

- [1] R. K. E. Bellamy. Strategy analysis: An approach to psychological analysis of artifacts. In D. J. Gilmore, R. L. Winder, and F. Détienne, editors, *User-Centred Requirements for Software Engineering Environments*, pages 57–67. Springer-Verlag, 1994.
- [2] R. Dawkins. *River Out Of Eden: A Darwinian View Of Life*. HarperCollins, 1995.
- [3] N. V. Flor and E. L. Hutchins. Analyzing distributed cognition in software teams: A case study of team programming during perfective software maintenance. In J. Koenemann-Bellinveau, T. G. Mohen, and S. P. Robertson, editors, *Empirical Studies of Programmers: Fourth Workshop*, pages 36–64, Norwood, NJ, 1991. Ablex.
- [4] P. Fowler and L. Levine. A conceptual framework for software technology transition. Technical Report CMU/SEI-93-TR-31 and ESC-TR-93-317, Software Engineering Institute, Carnegie Mellon University, Dec. 1993.
- [5] S. J. Gould and N. Eldredge. Punctuated equilibria: the tempo and mode of evolution reconsidered. *Paleobiology*, pages 115–151, 1977.
- [6] S. J. Gould and E. Vrba. Exaptation – a missing term in the science of form. *Paleobiology*, 8:4–15, 1982.
- [7] W. G. Griswold. Coping with crosscutting software changes using information transparency. In A. Yonezawa and S. Matsuoka, editors, *Metalevel Architectures and Separation of Crosscutting Concerns*, volume 2192 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.
- [8] J. Hollan, E. Hutchins, and D. Kirsh. Distributed cognition: Toward a new foundation for human–computer interaction research. *ACM Transactions on Computer-Human Interaction*, 7(2):174–196, June 2000.
- [9] E. Hutchins. *Cognition in the Wild*. MIT Press, 1995.
- [10] J. H. Jahnke and A. Walenstein. Reverse engineering tools as media for imperfect knowledge. In *Proceedings of the 7th Working Conference on Reverse Engineering (WCRE'2000)*, pages 22–31. IEEE Computer Society Press, 2000.
- [11] W. E. Mackay. Responding to cognitive overload: Co-adaptation between users and technology. *Intellectica*, 30(1):177–193, 2000.
- [12] S. L. Pfleeger. Understanding and improving technology transfer in software engineering. *Journal of Systems and Software*, 47(2-3):111–124, July 1999.
- [13] E. M. Rogers. *Diffusion of Innovations*. The Free Press, New York, NY, 1995.
- [14] D. E. Rumelhart and D. A. Norman. Accretion, tuning, and restructuring: three modes of learning. In J. W. Cotton and R. Klatsky, editors, *Semantic Factors in Cognition*. Erlbaum, Hillsdale, NJ, 1978.
- [15] S. R. Tilley. The canonical activities of reverse engineering. *Annals of Software Engineering*, 9(1–4):249–271, 2000.
- [16] A. Walenstein. *Cognitive Support in Software Engineering Tools: A Distributed Cognition Framework*. PhD thesis, School of Computing Science, Simon Fraser University, May 2002.
- [17] J. Zhang and D. A. Norman. Representations in distributed cognitive tasks. *Cognitive Science*, 18:87–122, 1994.