Journal of Software Engineering and Applications, 2011, 1, 48-58
doi:10.4236/jsea.2011.41006 Published Online January 2010 (http://www.scirp.org/journal/jsea)

Software Reuse: Developers' Experiences and Perceptions

William W. Agresti

Carey Business School, Johns Hopkins University, Baltimore, USA,
Email: agresti@jhu.edu

Received November 27th, 2010; revised December 14th, 2010; accepted December 27th, 2010.

ABSTRACT

Reusing programs and other artifacts has been shown to be an effective strategy for significant reduction of development costs. This article reports on a survey of 128 *developers to explore their experiences and perceptions about using other people's code: to what extent does the "not invented here" attitude exist? The survey was structured around a novel and simple "*4*A" model, which is introduced in this article: for an organization to obtain any benefits from reusing code, four conditions must obtain: availability, awareness, accessibility, and acceptability. The greatest impediments to reuse were shown to be awareness of reusable code and developers' perceptions of its acceptability for use on their new projects. For* 72% *of developers, the complexity of the old code was cited as a reason that the code was not reused. The survey also included developers' suggestions for ways to take greater advantage of existing code and related artifacts.*

Keywords: *Software Reuse, Survey, Software Development*

1. Introduction

This article reports on a survey of software developers to gain an understanding of their experiences and perceptions about incorporating already existing code into their new software systems. The motivation for the survey is that software reuse can be a source of cost savings in software development. The survey may help to understand more about the role of a "not invented here" attitude toward using someone else's code. Developers report on their experiences when they attempted to reuse code, and they offer recommendations on ways to achieve higher levels of reuse. Conducting such a survey is consistent with observations in [1] that more investigation is needed on the non-technical factors (such as prevailing attitudes and perceptions) that are barriers to reuse. Developers' viewpoints on reusing code may also provide insight concerning what they look for when considering incorporating code from multiple sources, such as public software libraries, open source software, or off-the-shelf components.

2. Background and Motivation

Reusing programming code is a widely practiced technique to take advantage of existing resources in development projects. Research studies and experience reports over the years have explored many aspects of software reuse, such as the following:

- What are the mechanisms for reusing code and other artifacts; that is, how does it happen, what are the steps involved?
- What artifacts are reused, and what are the effects when different artifacts (e.g., reusing design versus code) can be reused for the same project?
- How much does reuse (of various artifacts) reduce development costs?
- What is the effect of reuse (of various artifacts) on software quality?
- How can development tools, environments, and languages support reuse?
- What are the differences between reusing knowledge (e.g., by consulting with colleagues) and reusing artifacts (e.g., referring to documents) as alternative ways to obtain the same information?
- How does software reuse relate to design patterns, templates, software product lines, versioning, maintenance, and knowledge sharing?
- To what extent can developers be trained to maximize reuse?
- What are the differences (e.g., in additional development time required) when software is intention-

ally designed to be reused?

- Don't developers really have a not-invented-here viewpoint toward other people's code and want to write new code all the time?

Very useful overview studies over the years (e.g., [2-5]) have helped to organize the diverse technical, organizational, behavioral, and economic issues suggested by these questions.

There is an abundant literature, summarized here, showing that reusing software components can lower the cost of software development projects. The motivation for this article is that if we improve our understanding of developers' experiences and perceptions regarding reuse, we may identify ways to increase reuse, and, by doing so, further reduce the cost of software development. Again, there is support for this motivation in [1], "It is imperative to understand the behavior of software developers. The key to successful reuse programs obviously depends on its acceptance by the users."

One of the earliest empirical results on the effects of reuse on cost was reported in [6], analyzing data from a development environment that classified modules by the reuse categories of:

- Verbatim (complete reuse of a module from a previous project without changing any of the code)
- Adapted (there was reuse of a module, and there was slight modification of its code, that is, less than 25% of the lines of code were changed)
- Rebuilt (there was reuse, but the module was extensively modified, that is, 25% or more of the code was changed)
- Newly developed (that is, no reuse of a previous module)

The resulting analysis found that reused components cost less, as measured in median hours per 100 executable statements to develop the modules:

- Verbatim – 11 hours
- Adapted – 18 hours
- Rebuilt – 19 hours
- New – 24 hours

More recently, the effect of the extent of reuse on development cost was explored in [7], showing similar results that verbatim reuse has a significantly stronger effect than partial reuse (such as the categories of adapted and rebuilt above). With partial reuse, developers have to spend time to understand the code sufficiently to change the lines of code correctly and not introduce new defects.

In some investigations, the cost savings by reusing code are analyzed at the project level rather than at the level of individual modules. In two industrial projects in [8], the first project had 38% reused code and showed a 57% increase in productivity over development from scratch. The second project, with 31% reuse, resulted in a 40% productivity improvement.

A comprehensive cost model and framework for reuse operations was developed in [9]. In an industrial case study, various reuse scenarios showed a range of cost savings of 42-81% from reusing software compared to the cost of new development. Another study found project cost savings of approximately 50% [10]. So, to the extent that projects reused code, those projects reported lower total project development costs.

3. Conditions for Reuse: The 4A Model

The literature overview provided support that reusing more code can reduce project costs. The focus of the current research was to analyze what developers do when they consider using other people's code. The approach here is to propose a simple model, called the "4A" model, based on a sequence of conditions that must be met for reuse of artifacts (such as code or documents) to occur:

- Availability: the reusable artifact must be available; that is, it must not have been destroyed, erased, or lost
- Awareness: the developer must know of the existence of the reusable artifact
- Accessibility: a transfer mechanism must exist to get the reusable artifact to where it is needed
- Acceptability: the reused artifact must be acceptable to the developer for use in the new project and its environment

The 4A model may be seen in the context of the reuse life cycle in [11] as an expansion of their "reuse utilization phase" into four distinct conditions for utilization. If any of these conditions is not met, reuse cannot occur. In this sense, the 4A's can also be viewed as successive obstacles or impediments to reuse. Taking the 4A's in order, the path to successful reuse can break down at any stage. For example, associated with an organization is the amount of code that is made available for later reuse based on organizational policies and practices. When there is a potentially reusable artifact that the organization has archived and made available, a developer may not be aware of it. Even if the developer is aware of it, she may not have a way to access the artifact. Finally, even if there is a way to access the artifact, the developer may examine it and either determine that it is not suitable for reuse after all or simply exercise a preference to develop the code from scratch.

A survey was designed and administered around this 4A model to probe these four points of potential derailing of the reuse process and understand more about which ones pose the greatest obstacles to reuse. Among the reused artifacts of interest, the survey focused on the reuse of code, but specific questions, discussed later, did explore the reuse of other artifacts such as documents and test plans.

The survey was administered to 128 personnel who participate in teams that develop technical applications in an environment that is described in [7]. Because the focus in the survey and in this article is on developer experiences and perceptions, it is suggested that the survey results will be of interest to other development environments. The development teams in this environment are composed of both U.S. Government employees and contractors. Of the 128 personnel surveyed, 30 were Government employees and 98 were contractors. The survey participants had an average of five years of development experience in this environment and seven years of experience in total. There were several candidate artifacts that could be reused in this environment, including code, designs, specifications, and other documents produced in support of a development project.

Figure 1 shows the results of a survey question that directly addressed the 4A's and reused code. Developers were asked, "How often has each of the following major factors prevented you from reusing code?" As the Figure indicates, the greatest perceived impediments to reuse were awareness and acceptability from the 4A's. More respondents answered that these two factors were obstacles either a few times, several times, or often. In this environment, the other two of the 4A's, availability and accessibility, were not perceived as major impediments. Clearly, this information can be valuable when decisions are made to try to increase the level of reuse: specifically, more attention and resources can be dedicated to enhancing the awareness of reused code and addressing reasons that developers found code was not acceptable.

4. Awareness

Two survey questions probed the awareness factor, which was shown in **Figure 1** to be one of the major obstacles to code reuse. One question asked, "How did you become aware of the existence of reusable code?" The other question was similarly phrased but addressed "… items other than code (e.g., subsystem designs, sections of documents)". The results are shown in **Figure**s **2** and **3**, respectively. The possible responses to each question are shown below, along with the ways they were labeled in **Figures 2** and **3**. The respondent was aware because he or she:

- "Knew from my own experience" (denoted "Self-Experience" in **Figures 2-3**)
- "Looked at documents from previous projects" (Self-Docs)
- "Asked a colleague" (Peer-Ask)
- "Told/suggested by a colleague" (Peer-Told)
- "Asked a task leader/manager" (Super-Ask)
- "Told by a task leader/manager" (Super-Told)

The results were instructive in several dimensions. The developers indicated the extent to which they rely on themselves or others. If other personnel are involved, are they superiors (task leaders/managers) or colleagues? When there are interactions with others, is the information sharing more a "push" ("told by a task leader/manager" or "told/suggested by a colleague") or a "pull" ("asked task leader/manager" or "asked a colleague"). Finally, from a knowledge management viewpoint, the response, "looked at documents of previous projects" refers to explicit knowledge, that is relying on tangible artifacts, while the other responses referred to tacit knowledge.

As shown in **Figures 2** and **3**, developers often rely on their own experiences. A contributing factor to this result is that there is a conscious effort by management in this environment [7] to develop the programming staff by bringing them along into positions of successively greater responsibility on projects over time. For example, a junior programmer will typically be introduced into a team in the implementation phase to code some modules, each of which has a narrow and well-defined scope. Also used as a way to bring on junior staff members is to have them start with the testing phase, under the guidance of a seasoned developer, so they can start seeing the modules and how they fit together. These early experiences typically bring the developers into contact with single-purpose modules that are prime candidates for later reuse. As the developers mature, they are brought into new teams in more significant roles, such as programmer analysts from the start of the project, so they work on the specifications and design. Still later on, they may assume roles as task leaders for portions of the design, implementation, and testing phases. So, it is not surprising that developers report that they often rely on their own experiences to reuse both code and non-code items.

A general observation from **Figures 2** and **3** is that the developers make use of all the means to become aware of reusable items: they ask others; they are told by others; and they look at documents. The most notable result, although not surprising, appears to be in **Figure 2**, that developers are much more likely to ask peers about reusable objects and are much more likely to be told by task leaders/ managers. There is much more rigorous exploration of the social processes at work in reuse in [12].

5. Accessibility and Acceptability

Three questions probed the extent to which developers found that available reuse artifacts were accessible and acceptable. As discussed earlier and shown in **Figure 1**, accessibility turned out not to be as significant an obstacle to reuse as did acceptability. Three questions shed more light on both factors. The first question asked, "If

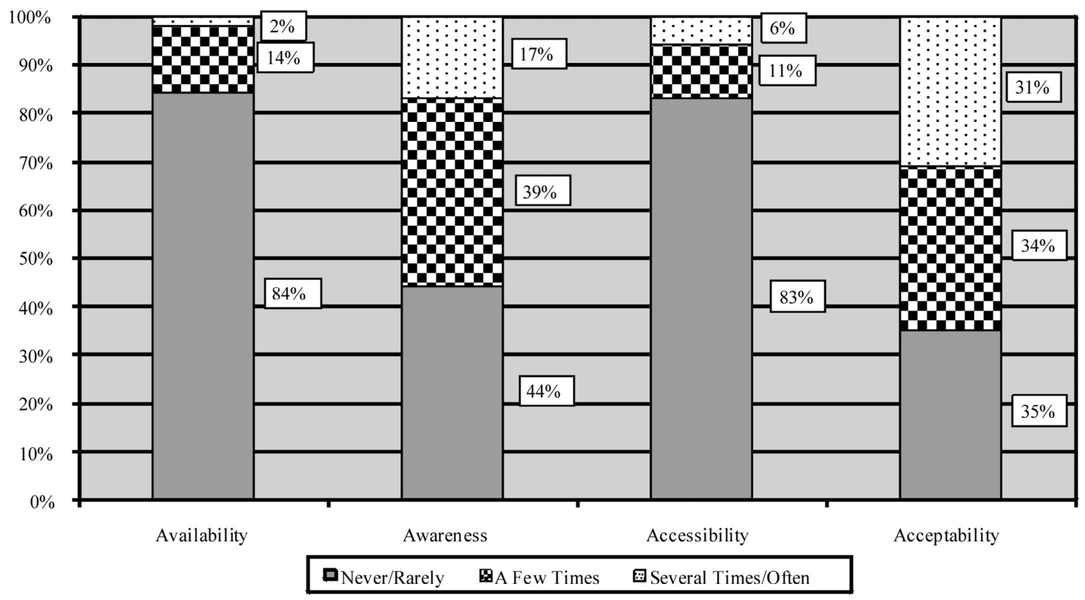

Figure 1. How often has each of the following major factors prevented you from reusing code?

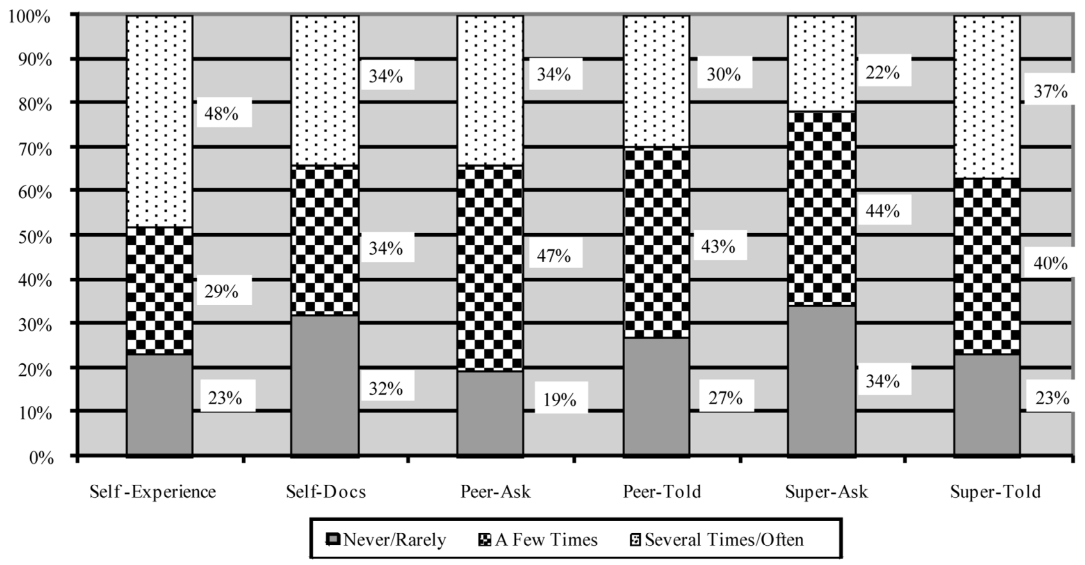

Figure 2. How did you become aware of the existence of reusable code?

there were times when you did not reuse code because the old code was not accessible or acceptable, what were the reasons?" **Figure 4** shows the responses of the developers. The responses substantiate the earlier result that accessibility is not a major factor for these developers. The Figure indicates that, while code was nominally available "somewhere," the fact that it was not actually available on the computer system needed or in the language needed was not a major obstacle (that is, relatively few responded that it happened several times or often).

When 67% of the respondents said that, at least a few times, the old code "did not meet the new functional requirements," it raises concerns when considered along with the responses just examined in **Figures 2** and **3**. If developers relied on their own knowledge, then it may simply be that, upon further inspection, their recollection about what the old code did was not accurate. If they received information from superiors or colleagues, then those personnel thought the old code was a match for the new requirements, but it was not the case. There may have been some miscommunication or misunderstanding in the exchange of information.

If the developers relied on documentation that was intended to support reusability (e.g., a catalog of modules

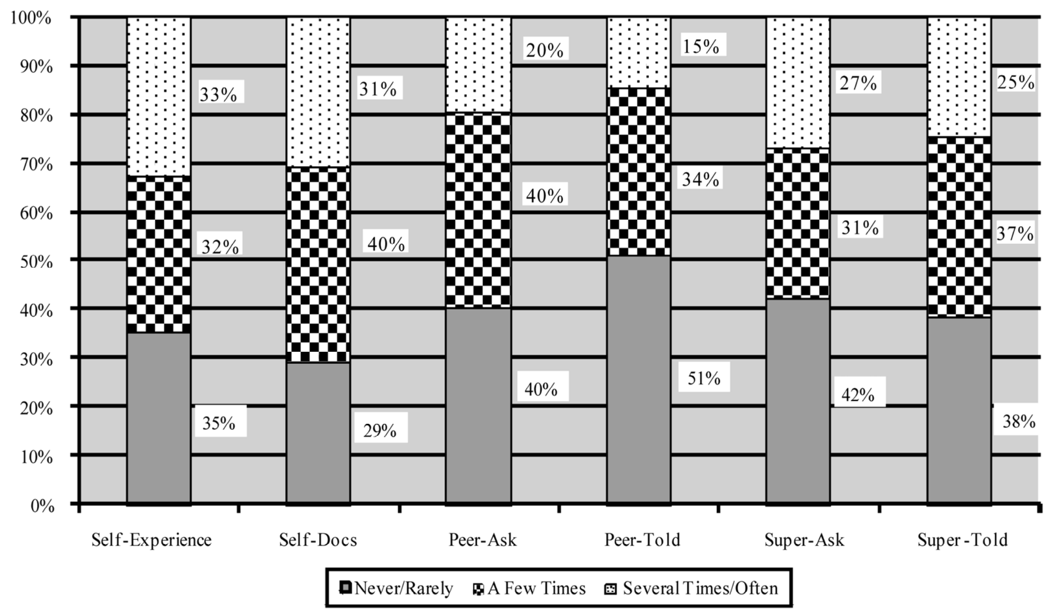

Figure 3. How did you become aware of the existence of items other than code (e.g., subsystem designs, sections of documents)?

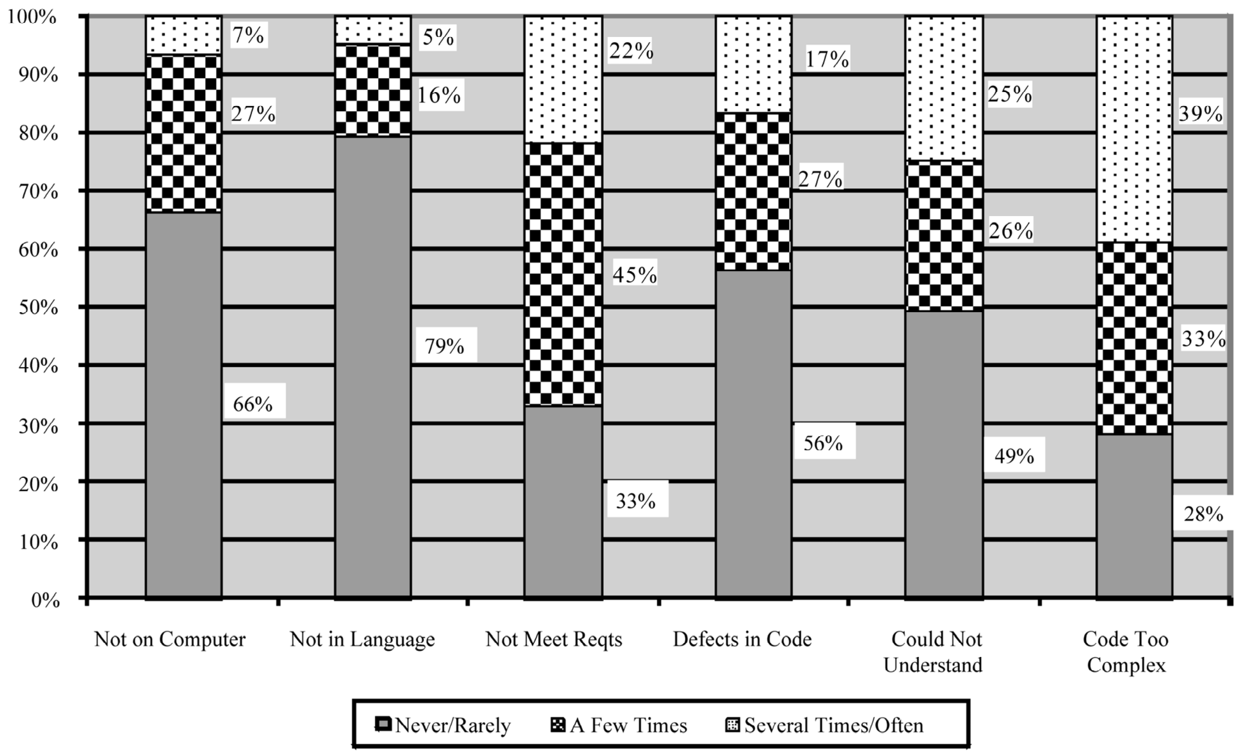

Figure 4. If there were times when you did not reuse code because the old code was not accessible or acceptable, what were the reasons?

thought to be good candidates for reuse), then it suggests that the documentation was incomplete in providing the information needed by developers to establish the acceptability of the old code or the documentation was an inaccurate representation of the old code in terms of specifying the functional and operational requirements being met. Perhaps these responses can provide data useful for making a business case that creating an online catalog of modules is a worthwhile investment.

According to **Figure 4**, 17% of respondents reported that defects in the old code were the reasons (several times or often) for not reusing the code. It should be mentioned that no attempt was made to follow up in these cases, such as to identify the respondents who answered this way, determine which old code they examined, and whether, in fact, that code really did have defects. It may be that the developers perceived that the code was defective when it was not. If the code was indeed defective, that fact should be troubling for the organization. Defective code is the surest way to discourage reuse!

Figure 4 also reveals that 51% of developers experienced difficulty understanding old code. Such a result motivates follow-up to determine if there should be efforts to improve the code style or self-documentation (e.g., via comments) to make it easier for others to understand the code. Improvements to coding style and documentation can be enforced with development tools and with practices such as code inspections, code reading, and code walkthroughs.

By far the greatest impediment to acceptability of old code was that it was perceived as too complex, with 39% reporting that "several times or often," this was their experience, and an additional 33% reporting that it occurred "a few times". This result is more evidence (e.g., shown also in [12]) for complexity being a deterrent to reuse. There are many aspects to this survey outcome. On some occasions, the old code is complex because it has been modified from its original creation. As with many organizations, this one trains its developers to keep track of the modification history of the module in the prolog (front matter comments) of each module. In this way, a developer can readily tell if a module has been modified, and how extensively. Extensively modified code may have rendered what was once a clearly coded module into a much less clear amalgam with patches and added-on functionality. Another indication of what can result from extensive modification is that these rebuilt modules (as defined earlier) have been shown to have a higher defect density than newly developed ones [13].

It takes effort to understand old code. The developer is trying to establish whether the old code will meet some or all of the new requirements so it can be part of a new system. When that old code is written such that it makes it especially difficult to understand, a developer can reasonably conclude that her effort is better spent developing new code from scratch. At least by taking that action, the newly written code will most certainly address the new requirements. So, the need to expend effort to understand old code provides encouragement and further rationale for a not-invented-here attitude by the developer.

When there are many occasions of this phenomenon (*i.e.*, expecting to reuse a module, then rewriting it), it can adversely affect the cost and schedule of a project if the effort required by the developer is more than expected. The survey had an "expectation of reuse" question and also a question that addressed the opposite situation, expecting to write a new module and then finding code to reuse. The survey questions were as follows:

- In your experience in the coding phase of projects, how often did you end up extensively modifying or completely rewriting a module when the design called for a module to be reused?
- How often did you end up reusing a module (in whole or in part) when the design called for a new module to be written?

Figure 5 shows the survey results, that it is much more likely that an "expected-to-be-reused" module was either extensively modified or completely rewritten rather than an "expected-to-be-developed-from-scratch" module was, instead, reused. Among the respondents, 40% of them reported that in 25% or more of the cases, they ended up extensively modifying or completely rewriting a module that had planned to be reused. It is acknowledged that the inclusion of "extensively modified" in the responses makes this result less clear in its message.

However, the basic result—having to extensively modify or rewrite a module when the plan was to reuse the module—has strong implications for projects to exceed their budgets in cases where the initial cost estimate (and budget) was dependent on a significant amount of reuse. The cost of module development shown earlier makes it clear that the costs will be higher on average for extensively modified and newly coded modules. So, when it comes time that developers actually reach the point of deciding on reusing modules, this survey is saying that some of the planned reuse may not occur. When more modules are developed from scratch than was planned, the costs increase.

Another question probed further into the developers' attitudes about reusing code, asking them to respond on the extent to which they agree with statements about the effect of time pressure, the extent of a not-invented-here orientation of wanting to develop from scratch, and the relationships to reliability and design integrity. **Figure 6** shows the responses. Time pressure on developers makes them more likely to try to reuse code, with 55% saying that they agree or strongly agree with that statement. This result is not obvious. Attempting to reuse code typically requires an investment in time to locate the code, examine it, compare it to what is needed, and test it. With time pressure, it may have been expected that developers would go into a "bunker" mentality, and prefer to dedicate themselves to coding the required modules from scratch rather than searching for possible reusable code and

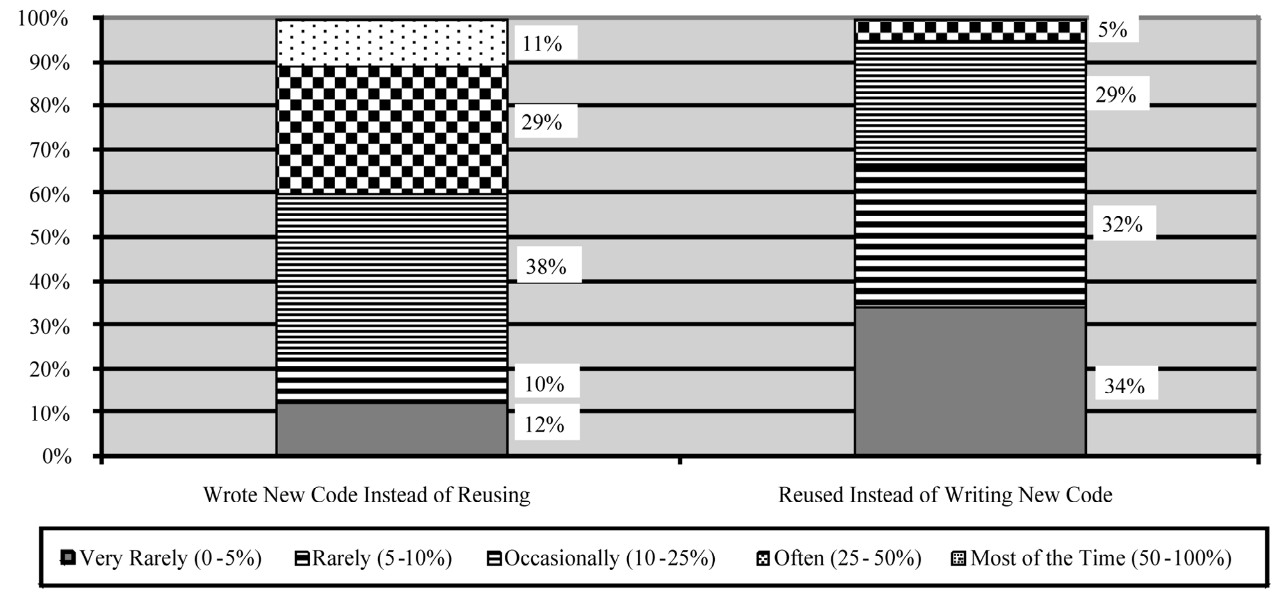

Figure 5. In your experience in the coding phase of projects, how often did you end up extensively modifying or completely rewriting a module when the design called for a module to be reused or reusing a module (in whole or in part) when the design called for a new module to be written?

examining it for suitability. The survey result suggests that developers do see reusable code as a way to save time. A concern for the organization might be that, with the time pressure, there will be less attention paid to 1) ensuring that the reused module is, in fact, a correct match for the requirements and 2) testing the reused module, and, instead, relying more on documentation and comments that the module will operate as described.

Only 21% of developers agreed that it is better to thoroughly rewrite old code than to try to reuse it. This result is consistent with the previous one, in that developers are giving evidence of a commitment to seek to reuse code when possible and not immediately decide to rewrite it. At least in this environment, developers do not report an especially strong not-invented-here attitude. This question has been one of those closely related to the observation that programmers, when given the chance, will want to "go it alone" and demonstrate that they are skilled at generating superior code. In the context of reuse, this may translate into a belief that reusing code will limit creativity [11] and more pointedly, "Only wimps use someone else's software." [14]. There are additional dimensions to this issue however. For example, new developers are more likely than their experienced colleagues to reuse existing artifacts, believing that designing and coding from scratch increases the risk of being criticized [15].

While the first two questions in **Figure 6** show an orientation to try to reuse code, the final question suggests one limitation to unbridled reuse. When reusing old code may "slightly perturb the system design," then 48% of developers say that the old code should not be reused. Developers may be expressing their understanding of the key role of an overall design structure and not wanting reused code to cause design changes that would likely threaten the integrity of the design and have ripple effects for other developers on a project.

6. Developers' Perceptions

The survey also probed the perceptions of developers about the benefits of reuse. The motivation for this line of questioning was to enable researchers to compare perceived effort saved to actual effort saved. If there are gaps between perception and reality, it may lead to follow-on activities to try to understand why perceptions may be higher than reality, whether the perceptions offer opportunities for improvement to actually deliver on the perceived benefit. For example, perhaps developers perceive very high value from reusing documents, but the actual effort data shows that much less benefit was achieved from reusing documents. Such an observation may lead to providing document templates that are more reusable for developers.

One question directly addressed the perceptions of developers: "On projects you have worked on, how much work have you saved through reuse of the following project elements?" The responses were chosen to probe the differences in perceived reuse among documents, delivered code, and non-delivered code (e.g., test drivers, debugging code). The results are shown below with per-

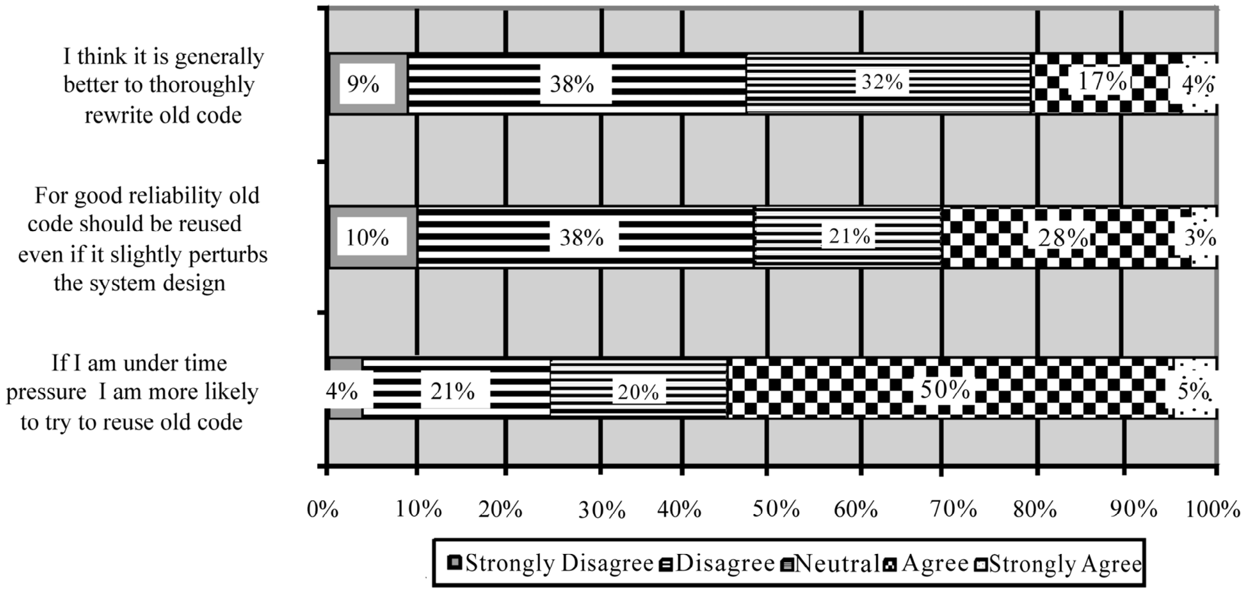

Figure 6. To what extent to you agree or disagree with the following statements?

ceived percentage of work saved by reuse of the item:

- Requirements documents, 9%
- Specifications documents, 10%
- Design documents, 18%
- Planning and management documents, 11%
- User's guides and systems descriptions, 22%
- Actual design of the system, 21%
- Delivered source code, 26%
- Test drivers and other non-delivered software, 12%

The results showed modest perceived benefits from the reuse of planning, requirements, and specification documents, from 9-11% expected savings of effort. This makes sense because these documents refer to activities that are early in the development process and very specific to the particular application being developed. There was much higher perceived reuse of design documents. This result is consistent with actual data on staff hours of effort saved by reuse of design, as found in [7]. The perception of high reuse of user's guides and system description documents (22%) can be readily understood because of the existence of "boilerplate" (*i.e.*, relatively consistent) content that needs to present in such documents, such as to describe the operational environment and modes of user interaction.

There was a developer perception that the reuse of previously delivered source code saved 26% of effort. This perceived value is very close to actual data on staff hours of effort saved by reuse of code, on average, in this environment for adapted or rebuilt modules presented earlier (18 or 19 median hours per 100 executable statements versus 24 hours for new code). Using 18.5 as an average of the adapted and rebuilt means that the actual effort savings is 31.25%. Perceived reuse of nondelivered code was much lower at 12%, which is to be expected because such code is often very specific to the tests being run or the nature of the debugging required.

One additional question about perceptions of developers explored a very different aspect of reuse and reusability: "If you were assigned to implement a module that is to be reused in several projects (as opposed to a module that no one else will likely every examine), what percentage additional effort would you put into the coding of that module?" The average response of the developers was that they would put in 29% additional effort. Data shows that the actual additional effort is much more. In [10] the average effort was 160-250% of the cost of developing a non-reusable asset, while the additional effort was 111% in [8]. The underestimation by developers is understandable. There is no widely accepted meaning of what is necessary for a module to be considered reusable. Also, by any definition, making a module reusable is influenced by the programming language used and by the rigor of the acceptance criteria for reusability. In particular, most developers underestimate the percentage of code that is needed for exception handling and recovery to make a module robust in the presence of off-nominal input and behavior.

7. Opportunities to Increase Reuse

While the objective of the survey was to obtain a deeper understanding of attitudes and perceptions about reuse, the ultimate aim was to start identifying opportunities for improvement so that development costs can be reduced by more effective reuse and higher levels of it. Three questions in the survey were focused on ideas for improvement:

- Which artifacts offer the most potential for additional reuse?
- Which of several proposed improvements would be most helpful?
- What suggestions do you have to improve reuse?

On the question to indicate which artifacts offered the most potential for additional reuse, the respondents were asked to consider the same set of artifacts listed above (requirements documents, et al.) and rate the potential for additional reuse on a five-point scale from low (1 point) to high (5 points). Because the research interests were in obtaining a relative indication of which artifacts were the best targets for reuse initiatives, the responses were combined for each artifact, such that, if all the respondents rated an item low (1 point), that would correspond to 20%; if all respondents rated an item high (5 points), that would correspond to 100%:

- Requirements documents, 45%
- Specifications documents, 49%
- Design documents, 51%
- Planning and management documents, 59%
- User's guides and systems descriptions, 51%
- Actual design of the system, 57%
- Delivered source code, 61%
- Test drivers and other non-delivered software, 48%

While the previous result showed that developers perceive they get the most savings from reusing code, they also believe that reusing code offers the most potential for even higher levels of reuse. From the expressed viewpoints of developers in this survey, there are potential improvements still to be obtained by focusing on code reuse. The survey results show that developers believe that, after source code, the next best targets for reuse initiatives would be to focus on the reuse of planning/management documents and the reuse of system designs.

Developers were asked to indicate their perceptions of how much various changes would help to increase the amount of reuse. Developers indicated their perceptions using a five-point scale ranging from "not helpful" (1 point) to "extremely helpful" (5 points). The results are expressed again in percentage terms, so that 100% means that all respondents indicated that a change would be extremely helpful. The results are as follows:

- Better comments in the code, 84.1%
- Better prologs, 81.4%
- Better offline tools, such as catalogs, listings, and documentation, 80.0%
- Better online tools for software search, description, and retrieval, 78.9%
- Better structured software modules (highly cohesive, etc.), 76.0%
- Better access to personnel who are knowledgeable about the code, 74.8%
- Use of different programming languages, 52.6%
- Use of different operating systems, 46.6%

The responses suggest that most of the items were perceived as helpful: better comments in the code, better prologs, and better offline and online tools, and access to knowledgeable personnel. These results can be a guide to where best to make changes to improve reuse. However, they offer only one half of any cost-benefit analysis. These results would need to be matched with estimates for the costs to make each change. There was a gap between these five changes and the last two. Moving to other operating systems or languages was not perceived as offering much improvement for reuse. Of course, this result must be tempered by the realization that there is no knowledge of the extent to which the respondents are familiar or not with the reuse-supportive constructs of other languages or operating systems.

The survey also included an open-ended question, asking for suggestions on how to improve reuse. Of the 128 developers surveyed, 58 provided suggestions. The responses clustered around suggestions related to personnel, tool support, and cost/budget:

Personnel Related:

- Assign staff members to projects specifically to advise the project team on what resources may be available to reuse
- Provide training on how to reuse and incorporate reused code into systems.
- Formalize a list of people who know about project X or topic Y.
- Provide training on where to look for help.
- Provide training on how to build reusable objects and how to build systems from them.
- Educate managers too !

Tool Support Related:

- Provide an online keyword-search capability that cuts across different sources of potentially reusable content.
- Write a reuse guidebook and update other handbooks to describe how to reuse; update the manager's handbook to show how reuse fits into the process model being used.
- Clean up, unify, and publicize existing code libraries; put them under control so that code and documentation are in agreement.
- Put the library under an administrator who can add to it.
- Establish criteria (enforced by tool?) to qualify code for library.
- Develop reusable code outside of the constraints of a project.

Cost and Budget Related:

- Identify high-payoff areas and invest in them.
- Specify a separate budgetary charge number that programmers can use when they are contributing to a code library and when they are searching for code to reuse (so they will be more willing to look for reusable code without penalizing their project by charging their time to it while looking).
- Run a contest: have programmers submit code; programmers who use reused code get 1 point for reusing it; the programmer who submitted the code gets one point every time that code is used.

The recommendations on cost and budget are particularly interesting because they raise the practical issue of providing incentives for developers to increase levels of reuse. The first suggestion, to invest in high-payoff areas, is a proactive approach. There could be an assessment of the application domains of the developing organization to identify knowledge areas or types of artifacts that would be the best candidates for strategic investments to increase reuse. This approach is consistent with strategies in knowledge management in which a large consultancy may have created dozens of strategic plans. A consultant wanting to develop a new plan has a difficult task to choose which ones to use as guides. The consultancy can reduce overall costs by investing in a proactive exercise for staff members to review all the strategic plans and develop a few archetype plans or templates that are the best models for consultants to use in the future [16]. In software development, an analogy may be the need to increase quality by being more consistent with non-functional code that is broadly applicable, such as the introduction of effective exception handling and security-related code into a wide range of modules. Instead of each developer going it alone, the company can dedicate some experienced software engineers to develop some code patterns that can be the starting point for developers on future projects.

8. Summary

Obtaining cost reductions from reusing code depends on developers actually accomplishing the reuse during a project. This article used a survey of developers to gain an understanding of some of their experiences and perceptions about software reuse. The greatest impediments to reuse were awareness of reusable code and its acceptability for use on a new project.

Some key survey results were that:

- Developers relied mostly on their own experiences for awareness of reusable artifacts
- 72% of developers found the complexity of the examined old code to be an obstacle to establishing its acceptability for reuse
- 55% of developers responded that time pressure makes them more likely to try to reuse code
- Developers' perceptions were much more accurate concerning the effort saved by reuse than they were about the effort required to create reusable modules

One survey result had implications for a role that reuse may play in large software projects exceeding their budgets: 40% of developers reported that in 25% or more of the cases, they ended up extensively modifying or completely rewriting a module that had planned to be reused. Given the additional effort required by this change, this result can affect the ability of projects to stay within their budgets, when those budgets had been planned on more reuse than was actually achieved. The developers also made recommendations for increasing reuse. The ideas tended to cluster into those that were related to personnel, tool support, and cost/budget.

There is an obvious threat to the validity of these results. Given that this is a survey, the results depend on the particular respondents and their personal experiences with reuse. The study was conducted in a particular development environment, so the results cannot be expected to hold in different environments (e.g., development platforms, tools, languages, application domains).

However, it is suggested that the 4A framework may be a useful model generally as a way to consider the sequence of steps in reuse and the possible impediments to its realization. Furthermore, because the survey revealed information about experiences and perceptions of developers, the results may provide insights for the organizations wanting to enhance their understanding of the climate for reuse. Lastly, researchers and managers may find that the developers' suggestions for improving reuse may resonate as being good ideas for consideration in their own organizations.

9. Acknowledgements

The author acknowledges the support of the U.S. National Aeronautics and Space Administration and thanks F. McGarry and D. Solomon for their contributions to this research.

REFERENCES

[1] V. Mellarkod, R. Appan, D. Jones and K. Sherif, "A Multi-level Analysis of Factors Affecting Software Developers' Intention to Reuse Software Assets: An Investigation," *Information & Management*, Vol. 44, No. 7, 2007, pp. 613-625. doi:10.1016/j.im.2007.03.006

[2] W. Agresti and F. McGarry, "Minnowbrook Workshop on Software Reuse: A Summary Report," W. Tracz, Ed., *Software Reuse: Emerging Technology*, Computer Society Press, Washington DC, 1988, pp. 33-40.

[3] Y. Kim and E. Stohr, "Software Reuse: Survey and Re-

search Directions," *Journal of Management Information System*, Vol. 14, No. 4, 1998, pp. 113-147.

[4] W. Frakes and K. Kang, "Software Reuse Research: Status and Future," *IEEE Transactions on Software Engineering*, Vol. 31, No. 7, 2005, pp. 529-536. doi:10.1109/TSE.2005.85

[5] P. Mohagheghi and R. Conradi, "Quality, Productivity and Economic Benefits of Software Reuse: A Review of Industrial Studies," *Empirical Software Engineering*, Vol. 12, No. 5, 2007, pp. 471-516.doi:10.1007/s10664-007-9040-x

[6] D. Card, V. Church and W. Agresti, "An Empirical Study of Software Design Practices," *IEEE Transactions on Software Engineering*, Vol. 12, No. 2, 1986, pp. 264-271.

[7] R. Selby, "Enabling Reuse-based Software Development of Large-scale Systems," *IEEE Transactions on Software Engineering*, Vol. 31, No. 6, 2005, pp. 495-510. doi:10.1109/TSE.2005.69

[8] W. Lim, "Effect of Reuse on Quality, Productivity and Economics," *IEEE Software*, Vol. 11, No. 5, 1994, pp. 23-30. doi:10.1109/52.311048

[9] A. Tomer, L. Goldin, T. Kuflik, E. Kimchi and S. Schach, "Evaluating Software Reuse Alternatives: A Model and its Application to an Industrial Case Study," *IEEE Transactions on Software Engineering*, Vol. 30, No. 9, 2004, pp. 601-612. doi:10.1109/TSE.2004.50

[10] S. Morad and T. Kuflik, "Conventional and Open Source Software Reuse at Orbotech—An Industrial Experience," *Proceedings of the IEEE International Conference on Software, Science, Technology and Engineering*, Herzlia, February 2005, pp. 110-117. doi:10.1109/SWSTE.2005.11

[11] K. Sherif and A. Vinze, "Barriers to Adoption of Software Reuse," *Information & Management*, Vol. 41, No. 2, 2003, pp. 159-175. doi:10.1016/S0378-7206(03)00045-4

[12] W. Boh, "Reuse of Knowledge Assets from Repositories: A Mixed Methods Study," *Information & Management*, Vol. 45, No. 6, 2008, pp. 363-375. doi:10.1016/j.im.2008.06.001

[13] W. Agresti and F. McGarry, "Defining Leverage Points for Increasing Reuse," Paper Presented at the Minnowbrook Workshop on Software Reuse, Blue Mountain Lake, New York, 1987.

[14] W. Tracz, "Confessions of a Used-Program Salesman," W. Tracz, Ed., *Software Reuse: Emerging Technology*, Computer Society Press, Washington DC, 1988, pp. 92-95.

[15] K. Desouza, Y. Awazu and A. Tiwana, "Four Dynamics for Bringing Use Back into Software Reuse," *Communications of the ACM*, Vol. 49, No. 1, 2006, pp. 97-100. doi:10.1145/1107458.1107461

[16] W. Agresti, "Knowledge Management," *Advances in Computers*, Vol. 53, No. 1, 2000, pp. 171-283. doi:10.1016/S0065-2458(00)80006-6