SOFTWARE TRANSFER:
A LIFE OR DEATH ISSUE FOR PROJECTS

David Braasch
Computer Science Department
St. Olaf College
1500 St. Olaf Avenue
Northfield, MN 55057
braasch@stolaf.edu

Abstract

The successful transfer of software projects from one development team to another is extremely important in multi-team development environments. Without it, software projects die because it is too difficult for a new team to resume development. We have identified three conditions necessary for successful software transfer from one team to another: *configuration*, *documentation*, and *distribution*. These requirements evolved from our experience in the Senior Capstone Seminar within the Computer Science department of St. Olaf College. During the term, the Capstone team resumed development on the Escher Web-Portfolio Manager, a project started in a previous course, and found it difficult to resume development. The paper expands on project transfer strategies, relates them to the literature available and reports on experience applying them

Introduction

When the Senior Capstone Team resumed development on the Escher software development project in the St. Olaf Senior Capstone Seminar (CS390), we gained a unique perspective on the transfer of such projects from one development team to another. Escher is a client-server application for creating web portfolios, written primarily in Java using agile methodology and the Eclipse IDE.[1] It was developed as a large team project in a prior course, Client-Server Applications (CS284). We discovered that some of Escher's functionality did not perform as expected, and much of the code was undocumented. We spent much of our time simply getting a demonstration to work properly, even though most of us had participated in the original development team. We needed some support features in the code we received from the original development team in order for us to begin making productive contributions to the project. It would have been much easier and faster for the original Escher team to provide those support features than for us to research and retrofit them.

We have identified three features that support the effective transfer of a software project from one development team to another: *configuration*, *documentation*, and *distribution*. We recognized the importance of these features the hard way: after roughly two months of work we only managed to get the Escher software working again, leaving no time to make any major code contributions. This failure, despite the efforts of the original development team to include helpful features like in-line documentation, is the symptom of a greater problem in software development: lack of focus on project transfer. While reference materials like *Software Engineering* by Roger S. Pressman and *The Mythical Man-Month* by Frederick P. Brooks mention issues such as documentation and configuration, they place little emphasis on successful transfer. More importantly, they ignore the problem of transfer itself. While a software project might only have one development team in the corporate world, the same is not true for the academic and research worlds. This lack of focus in teaching successful transfer in software development was evident in our Escher project; we had no frame of reference to set up the project for a successful transfer to a new team.

Configuration

The configuration problem we experienced during Escher development stemmed from our need to easily change variable values and settings. During testing, for example, the Escher portfolio manager required several independent servers. However, there was no easy way to make changes to server names and port numbers; we were forced to wade through the code, changing variable values until separate testing environments were created. We did not foresee this problem, even though our professor was well versed in the process of software engineering.

While Pressman mentions the problem of configuration in his well known-text, he does

[1] Escher Team.

so by focusing on the management and configuration of the development process, rather than the configuration of the software itself. He describes a Software Configuration Management System, or SCMS, his title for the organizational structure of a software development project.[2] As it turns out, the first development team followed Pressman's ideal of developing software in a SCMS. Pressman's SCMS is divided into four subcategories, three of which are pertinent to the discussion at hand: component elements, process elements, and construction elements.[3]

He defines *component elements* as "a set of tools coupled within a file management system (e.g. database) that enable access to and management …" of the software project. In the Escher project, both development teams used Concurrent Versions System, or CVS, which provides a system to control revisions of the code, with the ability to roll back to any version and track changes. By Pressman's classification, CVS is a component element.

His second category consists of *process elements*, "a collection of procedures and tasks that define an effective approach to change management (and related activities) for all constituencies involved in the management, engineering, and use of computer software. In the case of Escher development, the team adopted several policies regarding code change, reflecting the spirit of Pressman's subcategory. It was decided that to commit code changes, a developer must be wearing "the integration hat," so that conflicting changes would not be written to the CVS repository at the same moment. This procedure, coupled with proper use of the CVS repository (e.g. documenting code changes in the CVS file headers), comprise the process elements of the project.

The third subcategory identified by Pressman consists of *construction elements*, "a set of tools that automate the construction of software by ensuring that the proper set of validated components (i.e., the correct version) has been assembled." In the Escher project, the team employed the Eclipse Integrated Development Environment in conjunction with the CVS repository to achieve this goal. These combined tools allowed the team always to test the most current version of the software committed to the repository.

Despite the configuration steps taken by the first development team, following Pressman's principles, the second team still experienced difficulty. The reason for this difficulty was not for lack of Pressman's Software Configuration Management System (which was used), but rather the absence of flexible configuration options in the software being developed. It became apparent that Eclipse, the CVS repository, and development policies were not enough; we simply needed some way to quickly change settings in the code to allow easier testing. Our solution was to create a configuration class, or module, aptly named EscherConfig. This new class allows for centralized control of important variables, such as server addresses and port settings. As a result, developers can easily change a variable in EscherConfig rather than searching through multiple classes to find and change variables, and making sure consistency is maintained globally.

[2] Pressman, p. 740.
[3] Pressman, p. 742.

Additionally, the ease of configuration is important for deployment to the public, because without it the software is inflexible and not of use to anyone. Easy configuration allows an application to adapt to different milieus, e.g., changes in the network or database do not prevent the software from functioning correctly. For example, if the structure of the St. Olaf network changes or Escher is someday deployed to an off campus setting, configuration ought to allow easy setup without any changes to the code.

We feel such a configuration module should be included in most software development projects from the very beginning, so that the lack of flexible configuration is avoided. While configuration is taught in the world of software development as Pressman understands it, it is also important to remember the configuration of the software being developed, as well as the configuration of the software development environment.

While it is conceivable in some cases that a graphical user interface, or GUI, might take the place of a configuration class, such a feature might not be implemented until the end of the development cycle approached. In such a case, the developers are disadvantaged while testing the software. Furthermore, software designed for back end use, such as the Escher Save and Publish servers, will never get a GUI configuration menu. For such software, a configuration class is a must. Moreover, a GUI configuration menu could interact with a configuration class to complete its work.

Documentation

We also had difficulty transferring the Escher software development project due to deficiencies in the documentation provided by the first team. In many cases, when code was broken, the developers on the new team had difficulty understanding how the code was supposed to work, let alone how to fix it. This problem was the result of lacking or poor quality in-line documentation. Often, the code was incorrectly commented or difficult to understand. Roger Pressman proposes that good coding principles should be followed during software development: programmers should select meaningful variable names, use self documenting code, and make a visual layout that aids understanding, including indentation and blank lines.[4] These suggestions are all useful, if a bit general. The Escher team provided some Javadoc documentation, and the visual layout was maintained by Eclipse. However, self-documentation principles were not followed; meaningful variable names were not always chosen or descriptive enough. Additionally, the same variable name was often used across multiple classes, even if it represented a different variable entirely. This oversight made it very difficult to discern whether a variable in question was actually shared across classes, or was just named the same.

Brooks also discusses documentation in his 1975 book *The Mythical Man-Month*. While aging, this classic still provides valuable insight into successful software development. He describes documentation necessary to modify a program:[5]

1. "A flow chart or subprogram structure graph."

[4] Pressman, p. 113.
[5] Brooks, p. 166.

2. "Complete descriptions of the algorithms used, or else references to such descriptions in the literature."
3. "An explanation of the layout of all the files used."
4. "An overview of the pass structure – sequence in which data or programs are brought from tape or disk – and what is accomplished on each pass."
5. "A discussion of modifications contemplated in the original design, the nature and location of hooks and exits, and discursive discussion of the ideas of the original author about what modifications might be desirable and how one might proceed. His observations on hidden pitfalls are also useful."

While number four is dated and not particularly useful for modern software, the other points are very relevant. If the original team had provided us with UML diagrams, an update of Brooks' first point, we would have had a much easier time understanding how Escher functioned. Similarly, an explanation of the algorithms used would have gone a long way in helping us understanding some of the more complex operations, such as publishing a web site. Additionally, we would have benefited from a record of the original developers' intentions in writing the code, as suggested by Brooks' fifth point. If we had known the original concept and strategy for creating portions of the code, we could have approached our modifications with the same mindset. Furthermore, suggestions by the original architects for future modifications of the software could have directed our efforts.

The most important suggestions by Brooks deal with external documentation. Since most software developers know in-line code documentation is important for a small scale understanding of the code, it is usually completed. However, it can still be difficult to perceive the larger picture without additional tools. Javadoc documentation helps with the large scale picture when programming in the Java programming language. It is easy to include Javadoc during programming, like in-line documentation. However, it can be generated into a hierarchical html site, with all the classes linked together, which illustrates the overarching dependencies and inner workings of the code on a large scale. This has the advantage of providing the big picture, but does not require the developer to go back and figure it out, because the documentation is added during development. Unfortunately, Javadoc was done so sporadically by the first Escher development team on the Escher project that it wasn't much help to us.

The first Escher team left us with unfinished Javadoc and no other external documentation. Complete in-line documentation, Javadoc, UML diagrams, and external documentation of the algorithms used in the code would have been very valuable for us, and would have allowed us to contribute much more to the project rather then wasting time figuring out what had already been done. While these forms of documentation are not new in the software development world, our experience with the Escher project reaffirms the need for good documentation for successful project transfer.

Distribution

Good documentation is also important for the next important area of focus: distribution.

Pressman has two useful suggestions about deploying software. The first is "A complete delivery package should be assembled and tested," while the second is "Appropriate instructional materials must be provided to end users."[6] These suggestions are helpful, but once again his suggestions could be more specific. Nevertheless, his distinction between packaging of the software and documentation for the end user is a valuable one.

When the second development team resumed work on the Escher project, there was no user documentation, the code was not packaged for distribution, and the software lacked accompanying features necessary for release to the public. For example, Escher's two servers, required to save and publish portfolios were not configured to run on their own, outside of the Eclipse IDE. This problem was just one indication of a larger problem: the original team didn't take any significant steps towards creating a deployable package for distribution. It would have been wise to have consulted the reference materials of Pressman and Brooks, to better equip subsequent development teams for success.

The first development team followed neither of Pressman's suggestions. In that team's defense, the project was not considered mature enough to worry about releasing it to the public. However, it would have been easier for the original team to prepare a viable delivery package for the software than a later team.

Frederick Brooks also comments on software distribution. Brooks suggests the following user documentation:[7]

1. *Purpose.* What is the main function, the reason for the program?
2. *Environment.* What machines, hardware configurations, operating systems will it run on?
3. *Domain and range.* What domain of input is valid? What range of output can legitimately appear?
4. *Functions realized and algorithms used.* Precisely what does it do?
5. *Input-output formats.* Precise and complete.
6. *Operating instructions,* including normal and abnormal ending behavior, as seen at the console and on the outputs.
7. *Options.* What choices does the user have about functions? Exactly how are those choices specified?
8. *Running time.* How long does it take to do a problem of specified size on a specified configuration?
9. *Accuracy and checking.* How precise are the answers expected to be? What means of checking accuracy are incorporated?

Some of his suggestions reveal their age by being concerned with discrete output, whereas graphical user interfaces drive most applications today. *Running time* is not a particularly helpful documentation topic for modern software projects like Escher either, since Escher provides an interactive front end for content creation rather than batch number crunching. In spite of these changes, the principles behind his suggestions still have value. For example, his *domain and range* documentation suggestion may not help in a literal sense, but it would be appropriate to tell users what file types can and cannot be added to a website using Escher. In general, Brooks' documentation guidelines are still

[6] Pressman, p. 116.
[7] Brooks, p. 165.

useful: they simply need to be updated to modern software projects while preserving their principles. Neither development team had really considered this type of documentation important for distribution, but it will go a long way to making the user's learning experience a manageable one.

Brooks also suggests including test cases with the distribution package of software, to "reassure the user has a faithful copy, accurately loaded."[8] His rationale is that test cases can perform three functions: mainline cases that test the program's principal features, barely valid test cases that examine the boundaries of the input domain, and test cases that are scarcely invalid, to explore the domain boundaries from the other side.[9] While many modern applications don't have a strict input domains like programs from the 1970s when Brooks authored his book, it would possibly be helpful to include examples along with Escher when it is distributed. Several sample websites might be packaged with the software, along with walkthroughs on how they were created. The user could then follow the instructions, and verify his or her results against the examples. Furthermore, modern unit tests can perform the kinds of test that Brooks describe directly. Escher was developed using unit tests, but they fell into disrepair during the first development team's final push, and were only partially revived by our team.

In summary, we needed two aspects of distribution in Escher. First, we needed software to be packaged in a compact format that was easy to install and use. Second, we needed user documentation.

Conclusions

When we resumed software development on the Escher Web Portfolio Manager, we discovered the transferring a software project to a new team is a difficult task. While we followed standard development practices, we still found ourselves ill-equipped for the transition. The first team had no idea of the challenges in transferring a project, so the necessary preparations for transfer were not taken. Pressman's software engineering book suggested an entire infrastructure for constructing software, which the team effectively followed (without knowing it), and yet the transfer went poorly. The reason for this problem is that Pressman focuses on complete development the first time around, as opposed to resumed development or redevelopment of software. This phenomenon does not seem isolated to Pressman's book. In general, very little emphasis is placed on adding support features for future developers during software development, in spite of the fact that most developers work on previously developed code in industry.

In light of this deficiency, developers should be made aware of the difficulties of software development transfer to a new team. The reality is that software cannot remain a static enterprise: useful software must grow together with the environment of needs around it. To achieve this goal, software must be modified, and picked up by new developers, in

[8] Brooks, p. 165.
[9] Brooks, p. 165.

many cases when the original developers of the code are no longer present. For this reason, adding features to support project transfer helps everyone.

In the Escher project, our problems led us to a threefold solution. In the interest of flexibility and easier testing, we decided the best way to simplify configuration of Escher was to add an independent configuration class. This way, our original code could stay largely unmodified, while the configuration class allowed the flexibility we needed. In the case of documentation, we discovered that documentation truly is important. Additionally, the importance of external documentation was evident, since in-line documentation was not enough to fully understand the project. In several cases, e.g. Escher's code for publishing, we were forced to reverse engineer a flowchart of the code in order to gain enough understanding required to make useful code contributions of our own. Finally, we found that the project was not set up for successful distribution. Even as developers who had worked on the first team, we had problems getting the servers to run at first. This did not bode well for future users of Escher. Thus, we concluded that the software should be packaged for convenient distribution and execution. Additionally, it is advisable to include sufficient documentation targeted at the users, following Brooks' suggestion.

Future software development projects that incorporate *configuration*, *documentation*, and *distribution* will make it possible for future development teams to make progress on the software without the frustrating wasted time that our second Escher team experienced. Without such project transfer practice a large software project risks being terminated as not feasible for continued work.

References

Brooks, F (1995). *The Mythical Man-Month* (Anniversary ed.). New York: Addison-Wesley.

Escher Team. (2006). *Escher*. Retrieved January 10, 2006, from www.cs.stolaf.ed/projects/escher

Pressman, R., & Ince, D. (2000). *Software Engineering: A Practitioner's Approach* (5th ed.). New York: McGraw Hill.