Software Engineering in a Capstone Team Environment: An Experience Report

Ian R. Hawkins

An advanced project report submitted in partial fulfillment of the requirements for graduation with Honors in Computer Science.

Whitman College
2020

Certificate of Approval

This is to certify that the accompanying advanced project report by Ian R. Hawkins has been accepted in partial fulfillment of the requirements for graduation with Honors in Computer Science.

John A. Stratton

Whitman College
May 21, 2020

Abstract

The aim of this paper is to share software engineering lessons gained from the 2019-2020 capstone project for the organizers of Walla Walla's Great Explorations STEM program. Reflections on the strengths and weaknesses of our team's approach are generalized to provide suggestions on a team development process. I discuss maintaining low technical debt, focusing the client's needs, and getting the most out of the code review process as well as managing responsibilities within a team and working in intermittent sessions. These suggestions are geared towards student projects, but often apply to professional and individual projects as well.

1 Introduction

Whitman College's Computer Science senior capstone program enables teams of students to offer their skills to the community. Throughout most of an academic year, teams of senior Computer Science majors each take on a project that tests and expands their software engineering skills. CS professors offer a number of projects from students to choose from, some of which address problems in other departments or elsewhere in the community.

My team's capstone project serves Walla Walla's Great Explorations program. Great Explorations is a STEM event for fifth through eighth grade girls, hosted every other year at Whitman College. 450 students each attend three of the 30 workshops available. When a student signs up, they get to list their top six preferred workshops in order. Our goal for this project was to make a tool to place students into workshops, filling workshops evenly and respecting student preferences [2].

It's possible that problem of finding an optimal matching for all students has no polynomial time solution; in other words, an algorithm that makes the students as satisfied as possible might take an impractically long time to run. Instead, our goal was to produce an excellent matching between students and workshops, where two of a student's three workshops are in their list of preferences.

Software engineering is a systematic and scientific approach to software development, attending not just to the technical elements but the methods and theories that support them. This paper studies the software engineering practices of our capstone team, examining our methods and considering alternatives.

2 Product Quality

Throughout our project we focused on high code quality and low technical debt. Technical debt arises when developers implement quick and cheap solutions to problems that need attention. "Interest" is accrued as developers are forced to work around the poor-quality code, and the debt can be "paid off" by reworking the code. It's important to note that code with heavy technical debt can still meet requirements, but code with debt is harder to improve and maintain.

By avoiding technical debt, we hoped to maintain our rate of progress throughout the project and avoid introducing new problems. Since we won't be the ones to maintain or extend our project in the future, it was important to us that we maintain code quality in order to keep our project accessible to any future maintainers. Our team agreed that our code quality efforts were very successful. We came across very few points in our project where our progress was obstructed by convoluted code, allowing us to focus instead on the challenges of the project itself.

One contributing factor was the use of conceptual objects provided by the client when building our model. Our client used nouns like workshop, student, and session; we took the obvious step and implemented those ideas as classes in our model. By designing intuitive classes, we were able to approach implementation with a clear idea of what each class should be responsible for.

3 Project Management

Apart from development itself, a handful of project management practices helped keep our project moving forward. While these practices have a lot in common with professional development practices, they had to be adjusted to suit our development environment.

3.1 Best Practices via Code Reviews

Code reviews are a widespread software engineering practice for improving code quality. A code author puts their work under review for one or more associates can check the quality of the changes to the code base. The reviewers may make changes or, more often, ask the original author to fix or improve the code under review. A team often requires changes to be reviewed before those changes can apply to the shared code base.

Code reviews were a great opportunity for us to share and normalize certain best practices. When reviewing teammates' code, reviewers got to learn from the code author. Thorough code reviews helped us propagate important skills; for example, when we implemented our model, reviewers became familiar with our solution for using classes in Google Apps Script. Code reviews also gave reviewers the opportunity to suggest improvements to the author. Giving suggestions was more useful than taking it into one's own hands; by each modifying our own code to the suggestions of others, we got to immediately put good practices into action, while still leaving things in the hands of the one who knows the code best.

These experiences are in line with the findings of Bacchelli and Bird, who say that code reviews are not as focused on code defects as one might expect [1]. Code reviews "instead provide additional benefits such as knowledge transfer, increased team awareness, and creation of alternative solutions to problems". Nearly all of their interviewees brought up the topic of "knowledge transfer", unprompted, as an important part of code reviews. This knowledge transfer is

bi-directional, for the benefit of both the author and the reviewers: reviewers get to learn from the author of the code under review, while their suggestions and corrections inform the author of useful tricks and practices. Code reviews allow for the transfer of both general and project-specific knowledge: if the code under review is accepted, the reviewers are likely to work with that code at some point in the future. A code review is an opportunity for reviewers to familiarize themselves with the latest changes to the project.

3.2 Resource Limitations

Students have many responsibilities outside of the project, and each team member is usually able to contribute a modest amount of work each week to their capstone project. This is significantly different from full-time software development; a student's time is more limited and fragmented.

We found that keeping our code clean and readable helped us pick up where we left off, streamlining what would otherwise be a stop-and-go development process. A code author could more quickly orient themselves in their previous work, reducing the risk of getting lost in their own code.

Faced with certain large tasks, we had a choice to make: either leave large tasks in one piece and let an individual take it on, or split up large tasks to distribute them among multiple team members. Leaving tasks in one piece allows a team member to gain expertise on their own code, but adds the risk that other progress could be held up until it finishes. Breaking tasks down and distributing smaller tasks can avoid that risk and get the work done in a smaller time frame, but at the cost of some productivity.

Since each team member only has so much time to work on the project, we chose to split up our big tasks. As one might expect, we found that these small units of work were tied closely with one another. Clean and readable code helped us piece the smaller solutions back together to complete a larger task. This also helped us make changes to code authored by others without depending directly on the author's expertise on their own code.

3.3 Timely Optimization

Although some projects may be concerned with runtime, we set that concern aside for much of our project. Our program is meant to be run infrequently by the client, so we focused broadly on algorithmic runtime when implementing our early versions. If the algorithm takes exponential or even high-polynomial time, it could run for long enough to be impractical or even time out on the server. Since many problems are easier to solve in earlier phases of development, we addressed algorithm runtime in very general terms in an early phase. This planning helped guide us away from broken versions of the algorithm without optimizing prematurely. We also documented one bottleneck and later returned to it once we were sure it would improve the runtime. Without that simple improvement, the client would not see the results for around 20 seconds; after the change, runtime is now below five seconds.

3.4 Focus on Due Dates

Throughout the project, we found that we were best able to maintain our rate of progress by setting short-term goals for ourselves. Our use of deadlines varied dramatically over the course of the project, but we were most successful when either setting goals to reach before the next meeting or setting aside time to work together in person. At each meeting, we would ask one another "What will you have to show on Sunday?" Our Trello board helped us document these expectations. Given that a week's work can take much more or less time than predicted, we used these dates as guidelines for our progress rather than hard deadlines.

When it comes to goal-setting, a client meeting can be a particularly useful deadline. We met every other week with our team and up to three people from Great Explorations, and we took the opportunity to check that our targets were in line with the client's needs. We also got to demonstrate and describe our ongoing progress to the client. These meetings offered us a hard deadline by which to wrap up our latest work, and the client's presence at the meeting helped hold us all accountable. The meetings also helped us remember the end user and ask how the product can be improved to better meet their needs.

4 Challenges and Alternatives

4.1 Automated Testing

A 2011 case study by Taipale et al. examined the benefits and challenges of automated testing. The main advantages are functional quality improvement, saving time, and testware reuse, while disadvantages include greater implementation time, maintenance costs, training costs, and unreliability. They observed that test automation does not remove "the need for human involvement in supervising, maintaining, and developing test automation". Automated tests can reduce the time spent on manual testing at the cost of certain drawbacks. Resistance by developers against adopting new testing practices can also be an obstacle; testers have limited time and resources for learning new skills [3].

Early in our process, we discussed the option to use automated testing. Our project operates on a large body of form data to produce a large output, and automated testing would help us verify the correctness of the output without trawling through the data manually. It's also very likely that our project will be either maintained by the client or perhaps extended by another capstone group; automated tests would help future developers maintain the health of the project and avoid regression.

Many capstone teams choose not to implement automated testing, and despite the potential advantages, we decided against it as well. We expected that the costs of introducing automated testing were greater than the advantages gained throughout the lifetime of our project. In particular, all of us were new to Google Apps Script and needed to take time to learn it; under strict time limitations, a mis-estimation of training and test development time could have

thrown our project off course. To help mitigate the risk of introducing flaws, we compensated with thorough manual testing and in-depth code reviews. We took manual tests seriously and performed them each time code in review was changed. On top of that, the quality of the existing code made it less likely that a new addition would introduce faults.

4.2 Management Distribution

One principle of Whitman's CS capstone program is that responsibility for task management should be divided equally among team members. For example, team members might take turns setting the agenda for a meeting and keeping the discussion on task.

Our team struggled with this principle, settling into particular habits of responsibility without explicitly setting expectations. Working outside of a professional hierarchy, we found it very important to explicitly assign each of our group responsibilities to an individual person. Without an individual holding responsibility for a task, some jobs left as a "group responsibility" can fall by the wayside.

5 Conclusion

Based on our experiences with this project, these are my suggestions for small team projects. Keep technical debt low. We maintained code quality in the short run through our code reviews and maintained it in the long run by establishing best practices and conventions within the team.

Focus on short-term milestones to maintain progress at a steady pace. Maintaining regular contact with a client can help keep provide accountability and direction, as clients can provide feedback on progress as well as the opportunity to re-align the project with their needs. In a team with distributed responsibility, decide who will bear which responsibilities throughout the course of the project.

Standard meta-work development practices almost always pay off in a professional context, but consider closely how your development environment may impact the value of some meta-work. Automated testing usually makes sense, but its place in your project deserves thought and attention. Consider the costs and long-term benefits of your practices.

Overall, attention to meta-level project concerns is just as important as the development process itself. While software engineering practices rarely cause dramatic increases in productivity, many are worthwhile and have well established net-positive results.

6 Acknowledgements

I thank my teammates Trung Vu, Isaiah Standard, and Christopher Pyles for their excellent work on this project. I also thank our clients, Carol Morgan and

Ruth Ladderud from Great Explorations, as well as our advisor John Stratton for his guidance on our project and on this paper.

References

[1] Alberto Bacchelli and Christian Bird. 2013. Expectations, outcomes, and challenges of modern code review. In *2013 35th International Conference on Software Engineering (ICSE)*. ISSN: 1558-1225. (May 2013), 712–721. DOI: 10.1109/ICSE.2013.6606617.

[2] Ian Hawkins, Isaiah Standard, Christopher Pyles, and Trung Vu. [n. d.] Great Explorations Scheduling. (). https://github.com/WhitmanCSCapstone/ge-scheduling/.

[3] Ossi Taipale, Jussi Kasurinen, Katja Karhu, and Kari Smolander. 2011. Trade-off between automated and manual software testing. en. *International Journal of System Assurance Engineering and Management*, 2, 2, (June 2011), 114–125. ISSN: 0976-4348. DOI: 10.1007/s13198-011-0065-6. Retrieved 05/15/2020 from https://doi.org/10.1007/s13198-011-0065-6.