Drew Salyer

Prof. Daugherty

Radford University

ITEC 370

2011-05-01

Cooking to Goal Postmortem

1 - Introduction

The Cooking to Goal project was a Software Engineering effort aimed at creating a program to help users easily manage their nutritional goals by automatically scaling portion sizes of selected recipes to meet those goals. The user could enter their own recipes, then create a meal plan with those recipes, and finally, with a click, scale those recipes and generate a weekly shopping list for the necessary ingredients.

2 - Things We Did Right

We had a great team. Not really a factor we had under our control, but I felt the quality of our people was worth mentioning. Not everyone on the team was at the same level of hard technical knowledge, but we all brought something unique to the table which helped us mesh rather well and allowed us to delegate and tackle most of the required tasks quickly and efficiently.

We worked well remotely. Despite being geographically distributed and unable to meet face-to-face, we managed this problem very well. We set up a Google Group, which effectively gave us a mailing list so we could easily communicate among everyone in the group at once. The real kicker, however, was the Google Docs suite of online MS Office clones. Docs allowed us to store all of our files online, manage their permissions, organize them, share them, and best of all, *modify them simultaneously in real time,* with a *chat* feature to boot. We would often have one person working on one section of a document while another worked in a different area, with someone else reviewing the work being done, and all the while, we could communicate about what we were doing in chat. It was really an indispensible tool for collaborating, and best of all, *it was free*.

Equally as wonderful was access to the Adobe Connect (formerly Breeze) group meeting software provided to us by the college. This gave us further access to one another with additional features such as audio and video communication, the ability to share our desktops (and thus, any software or document on our computers), and even allowed us to record our meeting sessions for later

review. Without Connect and Docs, we really would have struggled to work cohesively as a geographically distributed team.

We met all of the deadlines. We spent some hectic evenings cranking out design documents, but we made the deliverables, come hell or high water, every time. The nitty-gritty work ethic of this team was outstanding. Everyone really stepped up when we were in a pinch.

The GUI looked great. The final result of the Cooking to Goal application, in this respect, exceeded my expectations. The program ended up a much more modern-looking "web layout"-styled (or .Net-inspired) application, with which, I think today's users can easily relate and feel comfortable at first glance. My vision for the product was much simpler, and conformed to the basic Java GUI applications I was used to seeing in textbooks – rather boring and uninspired, though very utilitarian.

3 - Things We Did Wrong

The GUI looked great. While the interface turned out great, it went against everything we had designed and planned for. It looked nothing like the drawings we made, and even worse, its creation introduced complexity by bringing another IDE into the equation. We had been using Eclipse, but now Netbeans was being used to create the interface. With Netbeans came additional ".form" files for every GUI class and additional settings and project files for the IDE, which cluttered up the repository. And while Netbeans' WYSIWYG, drag-n-drop GUI editor was easy to use, the code it generated was hard to follow (and if you tweaked certain parts of it, Netbeans wouldn't let you visually edit it anymore). We should have standardized on an IDE and stuck to what we designed – even if it meant a less inspired GUI.

We should have implemented a formal code review and approval process. We used a revision control tool coupled with an online code repository service to help us manage our source code. This was both a blessing and a curse. I think we should have required group approval of code before it was allowed to be pushed to the shared repository, or restricted access to the repository in some way, rather than allowing any team member to update the code base at any time. Since no such safeguards were in place, it was *far too common* of an occurrence to sit down for a coding session, pull in the previous night's code changes, and *not recognize a single thing*. Then the day was spent, rather than productively continuing work on the code you pushed out yesterday, but instead combing over the changes and trying to re-learn everything that was going on in the source. There were also multiple occasions when the code in the repository did not compile, and that should never happen. This is perhaps our biggest mistake. We should have approved code before making it a part of the application.

We didn't stick to the plan. Spending 4/5ths of our time working on project description documents, requirements specifications, design documents, and diagrams galore felt almost like wasted time at the end because we barely stuck to the design we had so meticulously laid out. I realize if there's a *problem* in the design, the design should be revisited, but many times, we simply deviated from the design *for no good reason.* I don't think a single one of our class implementations carried the

same naming convention we designed for it, and we ended up with a staggeringly greater number of classes than we had anticipated. Because of all this, I believe the project became too complex and crept far out of our original scope. We should have iterated back on the design documents, or stuck to them more rigorously (though perhaps that might have been another sort of slippery slope).

We weren't always on the same page. I think throughout the project, we all thought we were on the same page. We put our heads together and fleshed out all of the necessary documents. After spending such a great deal of time on these design aspects, in the end, I realized we weren't really on the same page – we just thought we were. When it was time to start implementing the code, we weren't seeing eye-to-eye on certain things we thought we had nailed down in the requirements and design phases, but when we started coding, there was too much confusion and dissent about what actually needed to be done. I think a different development method, such as Agile or Extreme Programming may have served us better than the traditional Waterfall method. That way we could start iterating on the code *early* in the process, instead of hitting a brick wall in implementation at the end – at the point when things should have been crystal clear.

We failed at design because we were coding in our heads the whole time. While I believe our project was a success and the end result was surprisingly good, we failed at the entire design process on a certain meta-level. Sure, we put in many hard hours slapping things down on paper that *looked* like a design, but it wasn't really *an abstraction of the problem*. We were seeing the whole thing in classes, routines, libraries, and development tools the entire time. I don't think we ever really put all of that out of our minds and focused on the pure design of the program. All along the way, we were mapping program features to code in our minds (I know this because we were suggesting tools and services and libraries for implementation from *day one*). We should have forgotten about the code completely – obfuscated it by layer upon layer of abstraction – and focused solely on the user experience and functionality of the program, focused on the design, and then iterated on it and explored other options and iterated on those until we found the best solution. That's how the best software is built. It makes me wonder if programmers should even play a role in the design process if they can't *not* dream in code.

The software didn't deliver on everything we promised. A lot of time was spent during implementation in areas out of scope of what we promised to deliver. As a result, we didn't complete everything we agreed to, leaving some major functionality for "later iterations" of the software. I felt like this was unacceptable, and the delivered program was not a "minimum viable product", but did have *advanced* features that, sadly, would not be utilized in the real world since the program does not perform the entire scope of its *basic* functions. We should have focused on what was important in making the product viable first.

4 - How We Managed Risk

This project was a learning experience for all of us. I'm not sure anyone on the team had ever gone through a thorough, structured, team-based software engineering process prior to the class. The *whole thing* felt risky since we had never done it before. There wasn't an excess of guidance through

the process, so we had to manage ourselves, and often felt like we had our necks stuck out, not knowing whether an axe was about to fall on us or not. While it was a bit uncertain and scary at times, we always rallied together and confidently pushed out our deliverables to the client. Sometimes in business, you just *have* to jump, even if you can't see where you're going to land because standing still is *death for certain*.

We took a few risks in planning, picking tools and libraries we had never used before, knowing that attempting to utilize them would be a great learning experience for us, but not *quite* knowing if they would work out, whether in their intended usage or in their actual implementation. There was a very real risk that the libraries would not turn out to be what we had thought or be beyond our technical ability or time constraints to implement, but luckily, their usage was nonessential to the final product. We always had the option that if the library or tool didn't work out, there was a simpler (though perhaps less desirable) solution in place as a safety net. We didn't *need* any external libraries or anything more than a basic text editor to build such a basic program, but we chose to risk utilizing those tools to enhance our learning experience.

5 - How We Managed Change

The most unanticipated changes were those we brought upon ourselves in the form of feature creep. We were constantly back and forth about what was going into the program and what wasn't. The client obviously had a laundry list of things to include, but we agreed only to specific items and they were documented in our requirements and design papers. Still, the undocumented features, and even others we thought would be neat to implement, kept creeping back in over and over again. Several times, both in our design documents and in our code, external features popped up from one place or another, and someone else would have to chime in – *Wait a minute, we didn't agree to implement that* or *I thought we said that would go in the "next iteration".* So usually, we would have to backtrack a little bit and fix the problem, but in the case of implementation, we ended up further from shore than we realized, so instead of swimming back to England, we had to cross the Channel into France. It wasn't where we had planned to be, but at least we didn't drown. Sometimes you have a do-or-die situation, and we managed change by treading water and trying not to die – probably not the best solution.

6 - Meaningful Conclusions

This project has been a tremendous learning experience for me – perhaps the most I've learned of computer science thus far in my collegiate career. The formal process of planning a project, analyzing requirements, designing a solution and implementing it is something I had never fully considered before. Sure, when I was working on a project, I might sketch out some interface designs (typically for the web) on a napkin or store a few ideas for how to implement something in my head (rarely on paper), but never before had I considered (or even really known about) the software engineering process. I feel like a missing piece has been added to the puzzle – and a very large piece at that.

If I learned anything from going through this process, specifically the traditional Waterfall Method, it would be that said method is not the best, except on very small scale, perhaps even one-man projects. Even then, going linearly through the process, much as our team did, is not sufficient. A software developer *must* iterate. *Iterate, iterate, iterate*. And then iterate some more. The Modified Waterfall Method may be a better approach, where returning to previous phases and reviewing and reiterating on those is part of the process. I suspect, though I have not yet fully explored, that other development methods, such as Scrum, Agile, or Extreme Programming might prove to be higher quality approaches to software development.

Perhaps the most important thing I've learned during this process (I mentioned it briefly already): The idea that you should put the code *completely out of mind* until such a time as is appropriate – in the later stages of design and the actual implementation. I've read quite a bit outside of my college coursework in the topics of software startup companies (the sort of small companies who come up with great ideas and either dominate some niche with their product or get bought out by Google), software design, and the current trending of software in general (especially mobile and web applications), and it's become clear to me that programming is perhaps the *least important* part of developing software. In fact, it's a common occurrence for a team with *zero* programmers to win software design competitions, simply by coming up with a design, iterating on it, and, perhaps most importantly, *exploring other solutions and iterating on those*. That's the only way to find out what works best, and, in my mind, it is the only way to produce innovative software of the highest quality. How the software is coded – the language and tools used – is becoming less and less relevant, and hardcore, low-level programming skills are less and less necessary as hardware technology increases and software languages evolve to higher levels of abstraction. Perhaps this is why software engineers typically earn far more than programmers: the success of software lies in its design.

We probably made more mistakes than we got things right in this project, and I'm *more* than OK with that. This was our first turn at software engineering, and what better way to learn the hard lessons, than through our own mistakes. Instead of a flawless run through the engineering process and a perfect end result, we traveled a rocky road and came out with a product that didn't fully meet our goals, but in many ways surprised us and exceeded our expectations. And going forward, we all now carry with us a set of tools, while well worn through repeated use, are better suited to fulfill our software engineering craft, as well as our career goals and aspirations.