Leon Pan (lpan21) & Austin Ha (austinha)
CSE 403, Spring 2018 | Project 1: Pitch

MOMENTUM

Motivation

Programmers have a reputation of being terrible at estimating how long it will take to complete a project. It's not the programmer's fault. Developing software is inherently uncertain: there is no way to forecast every detail of a design from the outset, understanding the full requirements of the project as well as the hurdles that an individual will encounter along the way. This is only exacerbated by the fact that a huge number of programming projects are not completed by a single person, but by teams — if an individual can't predict how long something will take for themselves, what hope do they have doing this for every other person in the team? Ultimately, this is a problem of uncertainty that can have huge monetary, emotional, and interpersonal costs. The reality is that we are usually much too optimistic — we try to fit too many features into a release, and some have to be cut; we think that we can focus effort in one area of a project because it will be quick, but it ends up taking two, three, four times as long as we had thought initially. *Having better time estimates lets us choose how we focus our efforts*.

Currently, there are many hand-wavy approaches to solving this problem: various equations with constant- or variable-factor multipliers that attempt to correct for programmer optimism, scheduling tools that allow programmers to record their time estimates for certain tasks or subtasks, time-trackers that track on how you are focusing your time (e.g. programming, bug-fixing, meeting, goofing off, waiting for a build etc.), and frameworks meant to help make software development less dependent on scheduling (i.e. more flexible [e.g. SCRUM]). The most compelling option currently is much more empirical than these, though it does take cues from all of the aforementioned approaches. This attempt at an empirical approach toward project time estimation is called *Evidence Based Scheduling*[1] (EBS) The premise is simple: use historical data on programmers' predictions and performance to model many possible outcomes for the project. It is based heavily on comparing a programmer's estimated time for tasks with their actual time spent on tasks. This approach also seems to work well for projecting project completion dates after a given project has started, but doesn't focus as much on making accurate predictions from the outset, or even when a product is in maintenance. As of now, the only implementations of EBS that we were able to find were:

a. Hacky implementations using advanced functionality of spreadsheets, and
b. Manuscript[2] (formerly FogBugz), which originates from the same place where EBS was formulated and developed.

Obviously, the spreadsheet method is not ideal, so we will ignore it as a meaningful implementation for now.

Manuscript, while carefully crafted, also has its shortcomings. First, it seems to be (almost) completely manually logged. As a programmer, you shouldn't have to be burdened with *managing* tools in order to get the job done. Instead, the tools should take care of as much of the process as is reasonable and possible. Second, Manuscript doesn't provide initial estimates for projects, tasks, and subtasks. It is up to the developer to provide these, which, as discussed earlier, can be wildly inaccurate. Third, these capabilities are not Manuscript's primary functionality. The GUI provided does not seem to afford treating this as a primary task, making it seem like second-rate functionality in Manuscript. Finally, Manuscript is expensive. While this isn't a problem for everyone, smaller teams might want a much lighter-weight tool that is specifically geared towards time estimation and time management, rather than bug-fixes and other project management capabilities.

[1] Spolsky, Joel. "Evidence Based Scheduling." *Joel on Software*, 26 Oct. 2007 https://www.joelonsoftware.com/2007/10/26/evidence-based-scheduling/

[2] Fog Creek Software. *Manuscript - Project Management for Software Teams*. https://www.manuscript.com/. Accessed 29 Mar. 2018.

Approach
In short, we propose a standalone implementation of *extended* EBS. EBS seems to be the most consistently praised and proven as a framework for time projection, so using these core ideas seems like a nice basis to start with. In addition, we hope to extend EBS in several meaningful ways, including automation of time tracking and automation of initial time estimates (hopefully without need for historical data for a new user). In order to automate time-tracking, we propose an eclipse plugin that aids a developer in tracking their time spent on a given task accurately.

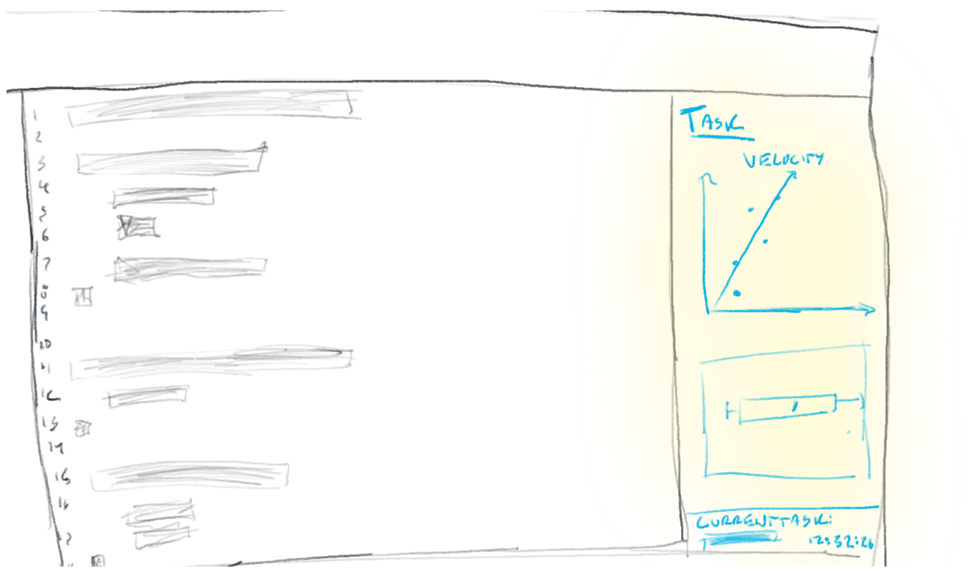

Figure 1. Eclipse Plugin Sketch

The user would first break down the project into many small tasks. For each of these tasks, the tool would provide an initial time estimate based on the average time needed to complete similar tasks and, if available, historical data of the user. When the user is ready to begin a task, they could use an Eclipse plugin to begin tracking the time spent on the task. When the task is completed the user could tell the plugin that he/she is finished, and the data would be recorded to future estimations. Instead of having to manually create scenarios for the Monte Carlo simulations[1], the user can give the tool a list of dependencies on which tasks must be completed before another task begins. The tool can then automatically generate scenarios for the simulations based on the given dependencies to calculate the probability that the project will be completed on any given date.

Challenges and Risks
One of the challenges to implementing this tool would be in designing the algorithm to produce good initial time estimates for the tasks to be used to calculate an accurate completion time for the project. We would have to design and test multiple algorithms in order to find the one that would produce the most accurate result.

Another challenge would be in determine what it means for two tasks to be “similar” to one another. One possible way to determine this is by using text recognition to check whether the two tasks share a relatively high percentage of their code. For example, a getter method for one variable is likely to share the some of the same code as a getter method for a different variable.