Luis Garcia, Tim Herrmann, Joe Maiocco

1. The two major concerns of any software project are monetary cost and projected length of time. Projects require funding in order to pay developers for their labors and to acquire any new technologies necessary for the development process, while projects also generally need to be completed within a specified frame of time to be of any use to customers. Generally, the concern about time seems to be more important, as a project that becomes overdue also has an increased likelihood of going over budget as new developers are (frantically and often to the detriment of progress, according to Brooks' Law) added to the team. Both money and time factor into the complete functionality of a project, that is, whether a delivered product has all the desired features. If a team is pressed for time or money and consults with the customer regarding these issues, then features may be dropped or pushed back to a later version release.
2. In the Agile software development process, each iteration demands that the requirements for the software be defined or refactored, that a design encapsulating all these requirements be crafted, that code implementing these designs be written, and that this code is adequately tested such that, at the end of each iteration, there is a working version of the given requirement. In theory, the requirements and design stages could be pulled out to only the beginning of a project. By removing these stages from each iterative phase, more time could be used in each iteration for building and testing working code. The downside of this approach would be that the requirements and design decisions would not adapt over the course of the project naturally. If an issue in either of these departments were to arise, the development team would have to return to either stage and work out the kinks, losing the time they should have been saving in the first place. Thus, the Agile development method ought to be effective without trying to factor out any of the iterative phases.
3. The Waterfall method for software development generally encompasses several phases: defining requirements, design (of the product and system), developing code, testing, and ensuring proper maintenance. Unlike in Agile development, where phases are repeated multiple times over the course of a project, Waterfall development typically handles each phase in sequence. The Waterfall requirements and design come together roughly during the first 20% of the time allotted for the project, and then the remaining time is spent building code and testing it rigorously. Thus, the documentation is generally complete before any heavy coding is done. In comparison, Agile development establishes the necessary documentation at the beginning of the project to a degree, but the requirements and design evolve alongside the code, which is helpful in avoiding any confusion regarding changes. Noticeably, Waterfall specifies an end phase for product maintenance whereas Agile does not. This makes sense, as a delivered product needs to be updated and patched adequately to survive in the market. A project developed in Agile would likely not need a designated maintenance phase if updates and patches were iteratively

added in iteration cycles following the project's initial release. However, a maintenance phase in Agile might be needed if the product's initial team was disbanded or let go of after the first product release, but the given product needed some upkeep despite these circumstances.

4.

- A user story is an imagined scenario detailing how a potential customer would use and interact with the envisioned product.
- Blueskying is the process of brainstorming with the customer and development team (as well as other relevant parties involved in the product's design) to capture any and all ideas about the crucial requirements that the given product needs to fulfill.
- A user story should be brief, designed by the customer, in a format and language that the customer is familiar with, and ultimately should only describe a single envisioned feature of the finished product.
- A user story should not be unintelligible to the customer, describe which technologies will be used to implement features, or be of excessive length.

5.

- All assumptions are bad, and no assumption is a "good" assumption.
    i. In terms of software development, this is a solid statement to make. A development team should never assume that they completely understand what a customer wants if there is some ambiguity. As the authors say, delivering a product that the customer is unhappy with is really just delivering the wrong product! Requirements should never be assumed when they can be clarified further.
- A "big" user story estimate is a "bad" user story estimate.
    i. While the wording of this phrase could be a bit more specific, the overall message makes sense from an Agile development perspective. User stories need to be split up into manageable chunks that fit (for the most part) neatly into iterations. If a particular user story takes up a lot of time for development, it could probably be broken down into multiple smaller user stories. However, there also exists the possibility that a user story estimate is "big" because nobody on the team is fully comfortable with the technologies or concepts necessary to implement the story in code. In this case, the estimate is not necessarily bad, but the team should also reevaluate the projects they are taking on if they do not quite have the collective skills or knowledge necessary to complete them.

6.

- You can dress me up as a use case for a formal occasion: User story
- The more of me there are, the clearer things become: User story
- I help you capture EVERYTHING: Blueskying, Observation

- I help you get more from the customer: Blueskying, Observation, Role playing
- In court, I'd be admissible as firsthand evidence: Observation
- Some people say I'm arrogant, but really I'm just about confidence: Estimate
- Everyone's involved when it comes to me: Blueskying

Most of these answers were consistent with the answers that the book provided. However, blueskying can arguably also be used to help get more from the customer alongside role playing and making observations. Some customers and the teams working with them might really enjoy the initial blueskying process and be able to really flesh out the requirements for the envisioned software without necessarily using the additional techniques of role playing or making observations.

7. A "better-than-best-case" estimate is an estimate that provides a projected timeframe or deadline for the completion of a specific feature by (wrongly) assuming that all of the circumstances surrounding development at that time will be perfect. As such, a "better-than-best-case" estimate fails to take into account the time spent on work not related to developing that specific feature, unexpected bugs, priority changes from the customer, maintaining the health of team members, etc. These kinds of estimates should be taken with a grain of salt and altered as needed.
8. A customer should be notified that the delivery schedule for a project will not be met *as soon as the situation becomes apparent to the team.* Giving the customer notice as early and as honestly as possible saves time and effort for both the customer and the development team. If the customer becomes frustrated by the circumstances, contracts can be effectively cancelled and parties on either side can go their separate ways. If the customer is irked but understands that sometimes projects do not pan out the way that they are anticipated, then some negotiations can occur to keep the project afloat but on a different delivery schedule. The worst possible choice to make might be to wait until far into development time to realize that a project will not be delivered on-time, and *then* to tell the customer. This last course of actions might bring about some legal troubles and be a blow to the team's reliability and credibility for future projects and customers.
9. Branching in a software configuration can be a necessary evil sometimes for productivity. Letting team members work on separate branches helps everyone work from a similar foundation of code. Developers do not have to wait for their teammates to checkout and check code back into a group repository before implementing the features that they are individually tasked with developing. Work can be handled in parallel. For example, Tim recently did some work on the beginnings of a battle system for *Cosmonarium* while Luis constructed the menu manager subsystem and Joe did some debugging on the scripts for the audio engine and actor engine. Each of us were able to work independently from one another by developing the code we needed on our respective branches. However, as our team has also discovered over the course of our project, merging and merge conflicts can be a massive pain to handle. Some merge conflicts can be easily dealt with in the chosen repository system, e.g. GitHub, but others require more digging to discover exactly where

the inconsistencies lie. Some merges can also add duplicate files and break dependencies if developers are not careful.

10. Godot, the game engine we are working with as our build tool, not only helps us package our software once development is finished (or a sample of it for now), but it also serves as a text editor and a design environment. Godot is very useful in allowing us to have more flexibility as we make our modules, and in fact we did choose it for its features as opposed to other alternatives such as Unity and Unreal Engine. Godot allows us to create, manage and visualize our project without having to stress about programming every single component. Asset management too is simplified because Godot takes care of importing as long as our paths remain consistent across versions. Nevertheless, since Godot does have a built-in file system, it forces us to pause development when we merge our components. If Godot is open during a merge, it will lock files that are essential to our merges, and this has been a rather annoying incompatibility we have had to debug with trial and error. In fact, Godot's file system may be the most poorly designed part of the environment, since it is also common for all our members to get lost in the folder hierarchy interface as we search for scenes or scripts.