Ben Bruser

Final

CS 410– Agile

Summer 2018

Short Answer (5 – 10 sentences)

1. **Define refactoring.**

The idea behind refactoring is that there are multiple ways to express the same concept in your source code. Refactoring is a process of cleaning up or improving source code without changing the functionality at all. This is done to improve code quality and readability. Examples of refactoring could be separating a function into two different functions, creating a function out of code that is repeated in other functions, changing variable names, etc. Refactoring makes the source code more efficient and more readable.

2. **Discuss the point of "done done".**

"Done done" refers to production-ready software. When you complete code that is untested and not integrated, it is easy to get a sense of complacency that you have are done with a particular story. "Done done" means you have features that are ready to show clients. Everything is tested, it's integrated and ready to roll out. The real idea behind "done done" is that once you finish the story item, you do not want to return to it. The goal is to have every story at the end of your iteration "done done" so you have software that is ready to deploy.

3. **We talked about five ingredients for code with no bugs. Come up with a 6th ingredient and describe how it would help.**

A sixth "ingredient" would almost certainly be communication between the coders of a system. As an individual, being conscientious of the design patterns you are using, refactoring code, fixing bugs quickly, and rigorous testing is required to achieve code with no bugs. When you add the extra layer of having multiple individuals working on the single system sometimes things can get lost in translation. Proper technical communication between co-workers or in-between pair-programming teams is essential to keeping bugs to a minimum. Let's say there is a database backend system that a team is coding. Bob makes an interface that changes tables without regard to certain constraint because they don't affect the overall performance and stability of the system. Now Steve comes in and builds a different feature that adds tables

with regard to a constraint Bob neglected and a bug arises in table data. Perhaps both were working off the five ingredients for code with no bugs. The only issue is that they were not on the same page of what standard a very specific database constraint would have. Perhaps it wasn't brought up in release planning. Always have good technical communication and overall good team communication.

4. **What is the difference between automated builds and continuous integration?**

Automated builds refer to the process of automating the build process without manually doing so, like manually invoking the compiler for instance. Continuous integration is a process of the continual invoking of the automated build process. So when a developer checks in code to source control, if the code is tested properly and those tests pass, the project will be rebuilt with the automation tools through the use of continuous integration software. Usually a server that hosts the continuous integration build will "wake up" or scan the server to see if any new successful check-ins have been completed. If so, the automation build software is invoked.

5. **Define and contrast unit tests, integration tests and system tests.**

Unit testing is the process of testing small pieces of functionality. A perfect example of what to test with a unit test is a single function. These tests are designed to validate that the function returns valid data or performs its function successfully. This is done through both successful tests, and even tests that fail on purpose.

Integration testing is for situations where unit testing does not work. Like for instance, when you have two different systems interacting like a database and a website. Integration testing can also be done if a function is too complex for a simple unit test.

System testing is the process of testing a full integrated software system. This can involve verifying of desired outputs from given inputs, and also testing of the UX of the application.

Ben Bruser

Multiple Choice

1. **Which is not true about velocity?**

   a. It can be used to determine how many stories to bring into a new sprint.

   b. It can be used to estimate number of features completed by a given date.

   c. It can be used to compare the productivity of different Agile teams.

   d. It can be used to estimate the date when all features in the release will be done.

   e. It usually takes 3 to 6 sprints before velocity can be established.

Ben Bruser

Essay – Answer the two questions (1 or more pages each)

1. Define and discuss the code smells we talked about in class. Find another code smell not included in the book and compare it to the others. Are all code smells bad smells? And are code smells a useful tool? Why or why not?

Divergent change occurs when changes that are completely unrelated effect the same class. This could be a complexity problem, meaning that the class should be partitioned or separated more efficiently. **Shotgun surgery** refers to the opposite of divergent change. Instead, you change multiple different classes/functions that effect a single idea or concept. **Primitive Obsession** refers to the representation of high level concepts with the use of primitive types like ints, floats, chars, etc. This can be fixed by incorporating these primitive types into a meaningful object. **Data Clumps** occurs when several primitive types represent a logical group. This causes data clumping which can often become unsustainable. Like primitive obsession, considered encapsulating these logical groups into a class. **Data Class** and **Wannabee Static Classes** occur when data and code are put into separate classes. This can often lead to duplicates of functionality. And this divergence of functionality that should be combined can lead to imperfect representation and functionality of classes. **Coddling Nulls** is the act of improper usage of null references. They are often improperly handled by being covered up with null returns by methods that can cascade into other parts of the system creating erratic behavior. Never accept null as a parameter unless it has well documented and controlled semantics. **Time Dependencies** often indicate an encapsulation error as they indicate that a class's method(s) must be called in specific orders in order to maintain functionality. **Half-Baked Objects** are a subset of time dependencies. They must be constructed and initialized with a method call before they can be used. These smells indicate that some of the burden of state management is put on the caller instead of managing the state itself.

A code smell not described in the book is that of the large class problem. Similar to primitive obsession and data clumps, large classes are bloaters. Sections of code that become too large they become hard to work with. The large class smell accumulates overtime as developers find it easier to add to existing classes rather than create new classes. Splitting up these large classes removes potential duplicate code and makes the system more approachable.

Code smells are incredibly useful as they provide guidelines and chunks of wisdom to keep in mind as they create software. Sometimes a large class isn't necessarily bad, although it is a smell. Neither is using loose primitives to represent high-level concepts. It all boils down to the context of each concepts use. The book provides the most apt analogy to refer to code smells. If you smell something in the kitchen it can mean several things. It could be a fire, it could mean that the trash just needs to be taken out. Or perhaps it means someone has made a meal with an ingredient you are not familiar with. It's a guideline that helps you understand where problems are likely to occur so you are conscientious of them. That alone makes them incredibly useful as one of the prime ingredients for code without bugs is forward thinking.

Ben Bruser

2. We talked about how TDD is a fundamental shift to how code is written. If your team were required to begin using TDD, how would you go about this task? How would you ensure continued success (and avoid fatigue of the use of the process)?

Test-driven development (TDD) as mentioned in class and in the book is a software development process that structures the development process around writing tests before any actual functional code is created. TDD has enormous benefits to reducing the amount of bugs present in your code. When you write tests before the code it requires you to consider what you want from the code that you will eventually write. You're always in a "How can I break this" mentality which helps you identify edge cases. TDD reduces your time spent in a debugger, reduces time spent on reworking, and tells you if you're latest change broke previously working code in a fast feedback manner. It makes code more maintainable, flexible and makes extensibility more feasible.

When designing the tests for our FTP client we would need to think about it at the highest level. What is the input and what is the output? So for instance, the collection of credentials from the user at the command line. Writing a test for this feature would have to consider a couple of things. The user would have to enter a hostname, a username, a password and maybe a port. The output of the test would determine if the hostname exists and can connect, and if the username and password are connected to an account that has access to the server. Starting out at the highest level lets you break apart the testing into logical segments.

If our team used TDD instead of our straight forward iteration I could foresee how we would need to adapt our process. As most may predict, and was brought up in a comment in response to a question in class; testing was left off till the very end. In fact we had completed most if not all our FTP functionality before we even though about the tests that we would add. While our application was relatively simple and we did not experience many bugs, problems could have easily sprouted up. Luckily our Jenkins CI build was able to determine if a build was stable or not, but that only told us that the latest commit failed the test. It did not tell us what part of the commit failed the test.

With that in mind shifting to TDD would be straight forward. We could simply pose restrictions on commits. No commit would be merged without a set of unit tests for that functionality. We could have selected the owner of the repo to observe pull requests to determine whether there was adequate testing being performed. This restriction would eliminate the issue of waiting till the last two weeks of the class to develop tests for our features. How do we encourage the group mates to write tests before the code though? Obviously we would have just written the feature and then wrote tests that would simply pass (in the worst case). This is where the benefits outlined in the outset would come into play. If we understood the breadth of pros that TDD offers we could shift our mindset from a typical college student scrambling to complete a coding assignment. This mindset shift allows the design of the ftp client, and in fact any software project to evolve past shallow understanding. It allows you to full understand the problem being solved and why something may or may not go wrong. Reaping the benefits and reduced overtime is as apt of a burnout deterrent as one can possibly hope for.