Tom Lancaster
CS 510 – Agile
Final
Summer 2019

## Short Answer (5 – 10 sentences)

### 1. Define refactoring.

Refactoring is “making your code better.” (Agile Session_7, slide #14) It is “the process of changing the design of your code without changing its behavior.” (The Art of Agile Development, pg. 303) It is taking the originally written code, keeping *what* it does the same, and changing *how* it does it. For example, in math, you can express ( $x^2 - 1$ ) to be (x+1)(x-1). They are equivalent to each other, but sometimes it is more advantageous to express it one form than the other. Sometimes, it is better to express the code in an easier to understand way than the most precise way. “Lone wolf” coders are a dying breed and therefore, your code has to be readable for others and efficient so the process of refactoring helps with these goals.

### 2. Discuss the point of “done done”.

“Done done” is the principle of having the code production ready for each iteration. In Agile, each project progresses by the number of stories accomplished. With the “done done” principle, once a story is completed that means it has been designed, coded, tested (all tests pass), integrated (with the code of past iterations), built (in the automated build), installed (included in the automated install package), migrated (connects to the database schema and data), fixed (all known defects are corrected or put into backlog), and accepted (by the product owner). Nobody should have to go back to a “done done” story (and therefore hold up the completion of other stories.) The cost of partially done stories could destabilize the whole program and negatively affect a company. This allows the project and the team to continue with the program confidently and allows the customer to be able to use any “done done” story as intended with no problems.

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

The 6th ingredient I would add to the five ingredients for code with no bugs is to hold your team to a high standard when it comes to bugs. The Art of Agile Development book calls this “inverting your expectations.” It means if you have successfully implemented the five other ingredients, then bugs should be a rarity. Whenever a bug does occur, everyone should be frustrated and feel accountable for it. These first five ingredients are processes that include each team member and therefore everyone is responsible for any bug that falls through the cracks. If the whole team has a high accountability attitude, then bugs will become even more uncommon since its importance is stressed by the team culture and more diligence is paid to the first five ingredients.

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

Automated builds are *what* is being built and continuous integration is *when* they are built. Automated builds "convert files and other assets under the developers' responsibility into a software product in its final or consumable form." This process automates the software that is being built instead of manually invoking the compiler. It is also a prerequisite for continuous integration. The continuous integration is an automatic process of building your software as developers check-in code, and run unit tests to ensure the code still works.

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

Unit tests "focus just on the class or method at hand." (The Art of Agile Development, pg. 297) You should be able to run at least 100 per second and shouldn't make any calls to a database or anything outside of the process. If your method does talk to a database, then use mock objects. Integration tests ensure that your code can communicate to a database, network, file system or anything that leaves the bounds of its process. These tests should also run fast, but not as fast unit tests. System tests are end-to-end tests which ensure that unit and integration tests mesh perfectly. These tests are very fragile and typically help test the UI code. They are normally performed after the test-driven development process.

**6. Describe a scenario where source control would not be needed in the software development lifecycle. If you cannot do so, explain why**.

It is my amateur opinion (which I am regurgitating from my more experienced mentors) that you are just asking for trouble if you do not use source control. Source control allows you to add features independently of each other, check for differences in code that isn't working, and keep track of your progress as you continuously add features. Most importantly, it backs up any recently working version of your program and everything related to the program including documentation, tools and the story backlog. You cannot foresee every problem and therefore must have a safety net that is a repository of previous working versions. Every software company benefits from version control and would be foolish not to use it in the software development cycle.

**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.

## Essay - Answer the three 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?**

According to *The Art of Agile Development*, code smells are “condensed nuggets of wisdom that help you identify common problems in design.” They are used to improve the design of existing code when refactoring. Sensing a “code smell” doesn’t necessarily indicate a problem. It just means to take a deeper look at the code. I will be discussing the most common ones listed in the book and comparing and contrasting one not found in the book.

Our textbook describes the eight most common “code smells.” They are divergent change, shotgun surgery, primitive obsession, data clumps, data class, wannabee static class, coddling nulls, and time dependencies. Divergent change and shotgun surgery “help you identify cohesion problems in your code.” (303) Divergent change happens when unrelated changes affect the same class. This means your class is too big and needs to be split into more focused classes. Shotgun surgery is the opposite. This happens when “you have to modify multiple classes to support changes to a single idea.” (303) To fix the problem, create or refocus a single home for all data. Primitive obsession and data clumps have similar issues in that they incorporate “high-level design concepts with primitive types.” (303) Primitive obsession is when you use a more primitive type instead of incorporating the data into a class. For example, you use a decimal type that represents dollars instead of incorporating it into a bigger “customer” class. Data clumps are when you have several primitive types that represent a concept. For example, a series of strings combined make an address. In this case, you would connect all the strings of the address into one “address” class. Data class and wannabee static class is when data and the methods that manipulate the data are in separate classes. Data class has the data without the methods, and the wannabee static class has the methods but no object state. Coddling nulls occur when you don’t handle a *null* result in some way like throwing an exception. Unhandled *null* values can lead to unpredictable application failures or cause other problems in your program. Time dependencies are “when a class’ methods must be called in a specific order.” (304) This means there is an encapsulation problem and therefore, you need to reorganize your class or classes. All of these “code smells” cause problems and bugs as the program becomes more complicated. Being aware and correcting these issues ahead of time makes for smoother programs and better products.

One code smell not mentioned in the book is “duh” or ambiguous comments. Every good programmer should be commenting their code to some extent. The problem is when they comment on obvious code or provide comments that don’t help the person reading the code much. In these scenarios, the comments are useless and can add hundreds of lines of code to already large code set. Comments could compound any of the code smells above. Is telling someone what a basic “for” loop does necessary? Does explaining your confusing class setup make it any less confusing? Do comments make your code perform any better? Of course not! Make sure your comments explain “why” you chose to perform this action or refactor the code so it is clearer to the reader.

Does this mean all “code smells” are bad? Not necessarily. You use “code smells” to improve and refine your code. It is a useful tool to improve the way code is written. It is no different from reading over any essay and rewriting sentences with grammatical mistakes or sentences that would be unclear to the intended audience. They are useful tools to either reaffirm the quality of your code or to modify and enhance it. At the end of the day, all code is read by people and for the use of people which is why “code smells” are a necessary, good tool for improving code.

**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) is a “software development process that relies on the repetition of a very short development cycle” that requires the passing of very specific tests. (Wikipedia) In non-TDD processes, code is written first and then unit tests are created to see if the code is performing the way it should. The opposite occurs in TDD and represents a fundamental shift in the coding process. Unit tests are written first and then you make code to pass those tests. This leads to a rapid cycle of testing, coding, and refactoring. It also creates more concise code and produces fewer bugs in the long run. I have been tasked with implementing the TDD process for my team. In the following paragraphs, I will explain my process of implementation and steps to ensure success and avoid fatigue.

My steps to implementing TDD are twofold - build buy-in and invest in training. I would first make a presentation on the benefits of TDD to all involved parties. I would explain how this process creates smaller, more efficient code, places planning front and center, maximizes test coverage, and results in lower development costs over time since you are building the minimum viable product. I would emphasize this is a long-term strategy and it won’t be an overnight change. I would then address any concerns or questions from my team. Once why of the change is explained and we have begun to persuade the team to TDD, I would invest in a two-to-three day training and hire an outside consultant to teach the course. During the course, they will learn about the five-step process of TDD and do example exercises.

To continue the success of the initial training and avoid fatigue of the process, I would make practical changes to the team and emphasize patience to all those involved. First, I would require everyone to do pair programming as they begin to implement TDD. This way you can hold each partner accountable to the TDD process and develop the discipline required to do TDD. I would also have brief weekly reminders of the five steps of TDD and explain why each step is important. Finally, from top management down, I would preach patience throughout the process. The results short-term are most likely not going to be there. This means we have to be committed to the process and the benefits will slowly come. TDD has a steep learning curve and it requires a certain amount of time to master for most people. As the saying goes, Rome wasn’t built in a day and an optimal work process in TDD won’t be either.

To effectively implement TDD and ensure its continued success, I would first build buy-in and invest in training. No process or methodology is going to work unless you have buy-in from all involved parties. Training is important in exposing the team to the methodology and understanding how it works. After training, I would require paired programming while in the first stages of TDD use. This holds each partner accountable in the process. I would also remind all involved parties the end goal of using TDD - better programs for our customers - as we slowly make progress. Patience will be needed to overcome TDD’s steep learning curve. TDD leads to fewer bugs, cleaner code, and faster results through the rapid life cycle of testing, coding, and refactoring. Discipline and perseverance go a long way for any successful project, but with the steps outlined here, my team and I should successfully implement TDD.

**3. You are tasked with defending a modified waterfall development process against Agile. Discuss why the modified waterfall method is better suited for software development. If you are unable to present an argument, explain why?**

The modified waterfall development process follows the same five phases involved in a regular waterfall and in agile. These phases are requirements, design, implementation, verification, and maintenance. In the modified version of waterfall, phases overlap each other where needed. Agile follows these same five phases but does them through smaller iterations or "sprints" of about 1-3 weeks. This allows greater flexibility in requirements and customer preferences and slowly increments the program over time. If someone asked me (like a professor in a CS 410/510 Agile/XP course) to explain why one software development process is better than the other, I would answer with the ubiquitous response of the software development industry - "it depends." Each process has its pros and cons and can be better suited depending on the project and industry. I am going to explain when to use the modified waterfall development process and agile and provide examples when one is better to use than the other.

The modified waterfall development process is ideal when there are clearly defined requirements, no changes are expected, and are limited to one or a few tests. Typically, smaller projects with obvious requirements fit this bill. For example, a program that simulates dice rolling better produce a number between one and six each time you call it to "roll" the dice. The modified waterfall is also used for larger projects that can't be tested frequently such as launching a rocket ship into space or implanting a pacemaker into a patient. These projects require success on the first test because if the rocket ship blows up or the pacemaker goes over the energy jolt needed, then lives and jobs are lost. In this type of industry, you have to go through each phase of the modified waterfall process and build your way up to the end project. Agile would not work because agile requires testing at the end of each iteration. Launching a rocket ship at the end of each iteration wouldn't be cost-effective which renders agile ill-suited for the project. Therefore, in instances of small, clearly defined projects and projects with limited testing, the modified waterfall is the better-suited software development cycle.

Agile is ideal when you have larger, undefined projects that involve a client and just a general direction. Since agile is performed in sprints, this allows for changes to be implemented at the start of each new sprint. Also, if you can test software without serious real-life consequences, agile would work the best because you can see if each feature is doing what it is supposed to do during testing. Examples of this would be pure digital projects like websites or learning software. As long as the minimum viable product provides some value to the customer, you can continue to add more features after each sprint and constantly improve upon the product. You can also hear the feedback of current users and make improvements accordingly. This profile fits many companies nowadays hence why agile is used in some shape or form in 71% of all organizations.

Whenever someone asks you if they should use modified waterfall or agile for their project, you should respond with "it depends." If you have a small, clearly defined project with no expected changes, use the modified waterfall. Also if each test has to be successful the first

time such as in the case of rocket ships or pacemakers, then, again, use the modified waterfall method. If your company is building a large program with only a general sense of direction, then recommend agile. Agile builds features in short sprints and is more flexible than the modified waterfall process. This allows for feedback from the client and/or customers and permits changes to the requirements as the client gets a better idea of what they want. Each methodology has its pros and cons so saying one is better than the other is foolish and should be avoided. It all depends on the project.