Comp 410
Participation Portfolio
Jason Bell 3078931
May 26, 2015

Discussion Forum 1

In Unit 1, you learned about the scope of software engineering. What are the main differences between software engineering and traditional engineering? What are the similarities?

The following excellent comparison is found in Software Engineering, by K.K.Aggarwal and Yogesh Singh, citation found at the end. I've enjoyed this book quite a bit.

The following is a comparison between building a bridge (traditional engineering) and writing a program (software engineering).

Constructing a Bridge	Writing a Program
The problem is well understood	Some parts are understood, others are not.
There many existing bridges.	Each program is different with different applications.
Requirements usually do not change during construction.	Requirements usually do change at all phases of development.
The strength and stability of a bridge can be calculated with reasonable precision.	Its not possible to calculate correctness of a program with existing methods.
When a bridge collapses there is a detailed investigation and report.	When a program fails, the reasons are often unavailable or even deliberately concealed.
Engineers have been constructing a bridges for thousands of years.	Deverlopers have been writing programs for 50 years or so.
Materials and techniques change slowly.	Hardware and software changes rapidly.

The text also makes the following points that I think are worth mention:

1) Software does not wear out. The lifetime of hardware reaches a point where it loses usefulness due to physical wear.
2) Software is not manufactured (in reference to the process of creating copies of the product). Creating new copies of a software product does not incur a cost.
3) Software components can be re-used. Most would deem it unacceptable for a manufactured product to have older, re-used parts.
4) Software is flexible and can do almost anything. However, this has the price of making it harder to plan, monitor, and control.

In summary, the differences between software engineering and traditional engineering largely come down to two things: how long the methodologies have been used and the nature of what is used to "build" with. Traditional engineering has an ancient legacy of accumulated tradition and experience, while software engineering does not. Traditional engineering uses materials with well known,

measurable properties, while software engineering must constantly adapt to new technologies.

K.K. Aggarwal, Yogesh Singh, *Software Engineering*, Revised 2nd ed. New Age International, 2005. [Online] Available at: https://books.google.ca/books?id=ytdKQGJ8f_AC&dq=differences+between+software+engineering+and+traditional+engineering&source=gbs_navlinks_s

Discussion Forum 2

In Unit 2, you learned about different life-cycle models. Select one of these models, and provide an example where the life-cycle model you have chosen is the best one to use in a specific situation for a specific product. Be accurate in describing the characteristics of the situation and product so that the selected model is clearly the best option. You cannot select the Chocoholics Anonymous product as an example.

For this question, I will be discussing my Comp 495 project, which I will be starting in a week.

The project is to create a compiler for a custom language. This is easier than it sounds, thanks to the .NET framework. The .NET standard library has excellent tools to generate intermediate language code for .NET applications: you can translate your language to C# and compile that, or directly produce the intermediate language code yourself.

However, this has still been a serious learning experience. It has highlighted one scenario where rigorous processes as described in this course are less than ideal. Simply, it is hard to create detailed plans and tests when you don't understand how to reach the solution.

There are a number of steps when compiling source code to machine language. Follows is a simplified version of descriptions found online [1]:

1) Tokenize the source code into discrete tokens or words.
2) Parse the sequence of tokens into a tree of expressions: expressions being namespaces, classes, methods, variables, etc.
3) Walk through the parse tree and validate the expressions: do expressions have appropriate child expressions? Are there duplicate names?
4) Walk through the tree again and generate .NET intermediate language for each expression.

The challenges were demonstrated when learning how to parse a programming language. I went through several prototypes, each of improved quality, while learning how to create grammar rules and match them to patterns in source code.

I'd like to call this rapid prototyping, but I'm not sure that prototypes of slightly decreasing levels of spaghetti code are worthy of that name. Rather, the learning process has been code-and-fix. Create a prototype, and experiment with different techniques until something works.

This has been messy, but extremely useful. It has provided me with the knowledge required to create detailed plans and tests when the course starts.

So, I would argue that while code-and-fix is a poor choice when creating the final product, it can be perfectly fine for gaining knowledge needed to do more sophisticated planning. In fact, the process made me realise that the parser alone was worthy of a COMP 495 project. Further, it's also a solved problem, and has been for decades. There are many high quality parsers freely available for many languages, all derived from Flex [2].

[1] Phases of Compiler. [Online]. Available at: http://www.personal.kent.edu/~rmuhamma/Compilers/MyCompiler/phase.htm
[2] Flex and Bison. [Online]. Available at: http://aquamentus.com/flex_bison.html

Discussion Forum 3

In Unit 3, you learned about the workflows and phases of the Unified Process. What are the main advantages and disadvantages of a two-dimensional life-cycle model such as the Unified Process compared to a one-dimensional life-cycle model?

The two dimensional model of the unified process has two disadvantages that I can see compared to a one dimensional process:

1) It requires that the team be experienced with the methodologies of the unified process. I have experience managing people, and I can see how convincing your staff to enthusiasticly adopt such a rigorous process would be very challenging. Adopting processes that must be tracked in two dimensions and strictly followed likely requires a major cultural shift.

2) The process is very complex [1], requiring rigorous controls and making it overkill for simple projects. I suspect that there is a threshold below which the unified process becomes more work than is worth the benefits gained from it.

There are two advantages for large, complex projects. There are many other advantages to be found for the Unified Process, but these apply particularly to the differences between single and two dimensional processes.

1) The time boxed phases of each workflow are inherently designed to support iterative and incremental development. [2]

2) Two dimensional processes support splitting a problem into simpler processes [2]. They are then easier to develop in parallel, with different time frames, and prioritized by risk.

In summary, two-dimensional systems facilitate development of large projects by divideing and conquering the problem, and allowing subproblems to be solved independently. For small projects, it is likely more work than it is worth.

[1] The Advantages and Disadvantages / Best Practices of RUP Software Development. [Online] Available at:http://www.my-project-management-expert.com/the-advantages-and-disadvantages-of-rup-software-development.html
[2] The Software Process.
[Online] http://people.eecs.ku.edu/~mcalnon/eecs448/lectures/448Lecture03.pdf

Discussion Forum 4

In Unit 4, you learned about different team approaches. What do you think is the most appropriate team approach for todays' software development projects? Describe the situations in which the development team works as well as the types of projects they work on, and argue, in the light of these situations and projects, why your approach is the most suitable one.

The choice of team doesn't just depend upon the type of project: it also depends on the nature of the developers and management and the capability maturity model of the organization. For example, for a democratic team to work, all the players must be willing. If experienced developers or controlling managers resist a democratic team, a classical team will likely have to be used. Secondly, a democratic team where all players have significant ownership over their tasks likely requires an organization with the culture and training to make it possible: one that doesn't have a low CMM. Everyone doing their own thing when there's no way to ensure proper coordination, measurements, and accountability sounds like it would turn into a mess!

As an article in MSDN magazine discusses, it also depends upon whether everyone is local or remote [1]. For example, a project that is developed in a distributed environment requires much more careful balancing of workloads: the reduction in contact between team members may result in unequal distribution going unnoticed. In fact, each team member must be given special instructions regarding their responsibilities to maintain strong communications when working remotely. The author describes this as "de-agiling": you can't use all agile methods when working remotely, but can still pick and choose. So, here we have a case where what is generally considered an "ideal" team structure must be adapted.

In summary:

Classical teams are best when:

There are strong egos, and a low CMM (people aren't trained or used to more modern teams). A classical team may be preferred for distributed development, where extra controls are required.

Democratic teams are best when:

Management and developers are comfortable working together, and the CMM of the organization indicates that the culture and training is there. If this is the case, then it may work for distributed development. I suspect that more centralized control would be needed than is typical, but can still be agile (agile-ish?).

[1] Sandeep Joshi. *Working With Agile in a Distributed Team Environment*. [Online] Available at: https://msdn.microsoft.com/en-us/magazine/hh771057.aspx.

Discussion Forum 5

In Unit 5, you learned about different tools and techniques used in software engineering. In which situations (or projects) might theoretical tools, CASE tools, or version/configuration control not be needed? Give one example for each of the types of tools (theoretical tools, CASE tools, version control and configuration control), and provide arguments why they are not needed.

The answer to most of these are the same: when cost, the size of the project, and the size of the team don't warrant their use. I'll do my best to flesh this out.

I will assume that "theoretical tools" refers to methods such as cost/benefit analysis, divide and conquer, and metrics.

Cost/benefit analysis: This is not needed if the project is an open source project with developers investing time and resources for free. It also wouldn't matter for something that is absolutely needed regardless of cost: suppose a new missile guidance system is needed to target an asteroid plummetting towards the earth. I don't think cost would be of much concern.

Divide and conquer: This would not be needed for anything simple. For example, even if a project is large, if it's processes are simple, divide and conquer may not be needed. It also would not make sense to divide the project into subprojects if it's being developed by one person.

Metrics: It would not make sense to use metrics for open source projects where it's impossible to push participants to work more efficiently and there isn't a deadline. It would also be unnecessary for simple projects that will be completed quickly (the extra work from recording metrics may even slow down the process). Finally, metrics will be meaningless in an organization with a low CMM: the processes and cultures just aren't in place for it and metrics will be inaccurate, if recorded at all.

CASE tools is a broad term. Clearly, most would consider an IDE to be necessary for anything but simple scripts. Modeling tools are useful for creating the artifacts of the requirements and analysis workflows, and even the implementation workflow. For example, I just discovered that visual studio ultimate 2013 is available on dreamspark, and am very impressed by it's UML modelling tools, and ability to generate c# code from them. I'm kicking myself for not using it earlier in the course. I won't bother mentioning e-mail, spreadsheets, etc, as those tools should be ubiquitous in any development environment. The fact that the text gives them special meaning makes me wonder how dated it's material is.

Version control isn't necessary for solo projects. I would still recommend using them as a safe repository for your work, and as a way of reverting to previous versions. So, I don't see a situation where it shouldn't be used.

Configuration control isn't necessary for solo projects. It's purpose is to control who is working on what, so it only makes sense to use in larger teams.

No citations: everything here is my own work experience.

Discussion Forum 6

In Unit 6, you learned about testing and different ways in which testing is performed. You also learned that testing is a difficult task and, for example, statements such as “Even though some organizations spend up to 50 percent of their software budget on testing, delivered 'tested' software is notoriously unreliable” [1]. Why is testing so difficult? What are, in your opinion, the three most critical challenges when it comes to testing a product?

In answering this question, I found an interesting article [1], and consulted with a friend who is a QA engineer in the infosec industry. The english in the article is poor, but the content is excellent. Both the article and my friend's comments match what the text has to say as well. Their remarks about the challenges of testing can be summarized as:

1) Finding all bugs is impossible. The best that you can do is find as many as you can. You will never find them all: the number of tests that would be required would result in a combinatorial explosion.

2) Developers and managers viewing discovered bugs as being a negative thing, rather than an opportunity for improvement. Clearly this is a cultural issue in an organization: if discovering and reporting bugs is seen in a negative light, who would want to? Thankfully, my infosec friend doesn't generally experience this issue: in his workplace the developers are generally happy to fix defects in their work... although sometimes they require convincing of the severity of the bug. He does, however, have experience with the next point, which is also described by the article that I cite.

3) High expectations are placed on testers and their task is just as difficult as any other... if not harder. Yet, in the culture of many organizations, their importance is often overlooked. Products being released without QA's stamp of approval, new features being added without proper tests being created: all too often testing gets sacrificed on the altar of pushing a marketable product out the door. In that case the tester is forced to be the negative voice that prevents the release... potentially upsetting stakeholders who don't fully understand the technical details.

Bhumika M. Why Software Testing is a Tough Job. [Online] Available at: http://www.softwaretestinghelp.com/why-software-testing-is-tough-job/

Discussion Forum 7

In Unit 7, you learned about the object-oriented paradigm. Assume a software company is going to be established and the CEO asks you for advice on whether to use the object-oriented paradigm or classical paradigm for future software development projects of this company. The CEO has heard a lot of positive arguments for the classical paradigm, including the long list of successes in the past. What are your three main arguments to convince the CEO that the object-oriented paradigm is superior to all other present-day techniques and the best way to go for today's software development projects?

I'd present a number of advantages of object orientated programming, but I would not try to convince him that it's superior to all other techniques, because, in my experience, it isn't. There are too many factors to consider for someone to say that one technique is always perfect. For small projects, object orientated techniques may be more effort than it's worth. I say this as someone who prefers object orientated programming, but who sometimes just wants to write some functions. To back up my assertion, I cite a brief paper by the Saylor Foundation, a nonprofit organization aiming to make education free for all [1], that agrees with what I'd say.

The advantages described by the article are:

1) Improved productivity: the division of the project into discrete project makes distribution of work easier, and hence development can proceed in parallel.

2) Maintenance is simpler due to the encapsulation into objects. In theory, changes made to one object should not affect other objects. So, changes to one aspect of the product don't have to mean changes to the entire project.

3) Faster development due to high quality libraries available for object orientated languages. I don't necessarily agree with this one: functional languages can have good libraries too.

4) Object orientated development requires more planning, which lowers cost as planning is cheaper than implementation.

5) Higher quality software as the faster development and lower cost allows more time to be spent on testing. While pitching this point, I'd be sure to point out that this is only the case if the free time and money is actually used for testing, and not used as an opportunity to save money!

The article then lists some negatives, not all of which I agree with.

1) Steep learning curve: I'm not sure that this is relevant today: object orientated methods are all that is taught in university, at least at Athabasca University. I suspect many university graduates have the opposite problem: finding it hard to wrapping their minds around functional programming.

2) Larger programs due to more code. In my experience this is true: I've worked in assembly and have experience how much machine code can result from one high level construct. However, I think that the increase in size is exaggerated, and a non-issue in this age of limitless storage space.

3) Slower programs. Yes, but they aren't that much slower. In my experience, on the order of single

digit percentages.

4) But, here's the most important point. Directly quoting the article:

"Not suitable for all types of problems: There are problems that lend themselves well to functional-programming style, logic-programming style, or procedure-based programming style, and applying object-oriented programming in those situations will not result in efficient programs." [1]

In other words, it's situational. For large projects of some complexity, object orientated methods are ideal to ensure easier division of labor and future maintenance. With many of these applications, the small loss in performance that this costs is worth it. But, suppose I'm creating a compact high performance math library: functional programming may be the better idea.

I'd pitch the object orientated paradigm as ideal for most projects, and make sure to describe situations where it is not.

[1] The Saylor Foundation. Advantages and Disadvantages of Object Orientated Programming. [Online] Availabe at: http://www.saylor.org/site/wp-content/uploads/2013/02/CS101-2.1.2-AdvantagesDisadvantagesOfOOP-FINAL.pdf

Unit 14 Discussion Forum

In Unit 14 you learned about good programming practices. Think about other good programming practices that have not been mentioned in the textbook, and list *three* more good programming practices together with arguments explaining why they are important. Provide both positive and negative examples. In your answer you can address independent-language programming practices, object-oriented language programming practices, and programming practices for a specific language.

In this discussion I cite a slide presentation from University College London [1]. Some of the practices recommended include:

1) Consistent use of spaces and tabs. Mixing spaces and tabs, and using varying number of spaces, reduces readability. My own input having worked with Python, is that it also really confuses languages that use white spaced scoping!

2) The article suggests enforcing a cap on the length of a single line of code. There is a middle ground between having one really long line of code, and having it broken up into too many.

3) In addition to requiring indentation inside control blocks, also require indentation when code continues on the next line: this makes it obvious that the indented line is a continuation. For example:

```
int i = 1 + someNumber
+ someOtherNumber;
```

As opposed to:

```
int i = 1 + someNumber
```

+ someOtherNumber;

Another interesting code practice that I see frequently is the question of whether the open brace should be on the same line or the next line after the start of the code block.

```
void foo() {
```

vs

```
void  foo()
{
```

While I prefer the latter, as it helps separate the method declaration from it's definition, people who argue about this one are pedantic. I'd say use whichever you prefer, unless your organization's code standards require otherwise.

[1] Patrick Guio. Good programming practices. [Online] Available at: http://www.ucl.ac.uk/~ucappgu/seminars/good-practice.pdf

Comment #1

In response to danielbo22's post in discussion forum 1:

1.This is the annotation margin. See the annotation drop down at the top of the page for help. Click x to remove this message.
The life-cycle model that I have chosen to discuss is that of the Open-Source life-cycle model. The Open-Source model is quite different compared to many of the other popular life-cycle models, as it is often characterized by a team consisting of developers who are volunteering their time to the project and who accomplish their work in their spare time.1 This isolated fact alone makes this model limited in its applicability, as anyone working on the project must have a keen interest in order to be willing to contribute. This nature means that for a project to have success using this model, it must be of a subject matter that is of great interest to a wide variety of people, in order to attract enough human effort to see the project to completion. On the other side of things, because any of those contributing to the development of the project are freely giving their time, the individuals tend to be very committed to the cause and often are very skilled software developers – a recipe for high quality products.
The type of project that is best suited to the Open-Source model is that of infrastructure projects, which are not just isolated to a commercial organization. This makes sense, as the alternative would not be able to attract the widespread attention needed for the model to be successful. Popular projects that have been completed using this model in the past are the Linux and OpenBSD operating systems, the Firefox and Netscape web browsers, and the gcc compiler. Aside from operating systems, web browsers, and compilers, other potential projects that would fit the model well are web servers and database management systems. Usually the development begins with rapid-prototyping producing a very early initial working model, completed by a small number of developers who do not bother with specifications or design. This initial model is then released to the community of those interested and each of the users become co-developers, submitting defect reports for the core group to install fixes on.

Reference Notes

Retrieved April 19th, 2015.

From http://digital.cs.usu.edu/~xqi/Teaching/CS2450F08/Notes/Ch2.LifeCycle.pdf

My response was:

I'm not sure what you mean by "infrastructure projects": you then proceed to list a diverse range of projects that contradicts your assertion that open source is limited in it's applicability.

I use open source products extensively, for everything. I use LibreOffice for work processing, spread sheets, presentations, etc, and while it may be not quite as good as Office, the price agrees with me more . I've used open sourced IDEs, and I used open source libraries all the time when programming.
I think the assertion that open source projects require a project that people are interested in to draw skilled and enthusiastic contributors is a very valid one. The idea that it's in any way limited in what type of products it can be used to develop is not.

Comment #2:

In response to denisbl1's post in discussion forum 6:

Generally, the tester has to deal with limited input data to test a product [1]. Unless it is a highly automated process, it is not possible to test all possible inputs and validate that all outputs are correct. The tester has to pick up the input data that is most likely to be used by the client and maybe a subset of possible invalid input data. The tester then has to confirm that valid input data produces valid output and also that invalid data generates some type of error message, reports the error and certainly does not crash the product.

Another critical challenge is the hardware and the software environment that the product is expected to run on [1]. Most shrink-wrap product will include a minimum set of requirements that the computer hardware must meet (e.g. RAM size, hard drive space needed, CPU speed) for the product to run correctly. It is also difficult to ensure that a product will continue to run if a new Operating System is installed on the a computer, therefore a specific OS may be specified in the requirements.
A third critical challenge has to do with selected inputs [1]. Even if the tester wants to, she may not be able to generate inputs that the product needs to handle. Let’s say a that a plane’s black box has to transmit technical data to a satellite if a plane crashes, how is it possible to test the product without actually crashing a plane? It may possible to do such testing with the help of a simulator but this may be quite an expensive proposition. But even with a simulator, it is difficult to ensure that it is a 100% reproduction of the “real thing.” Still, using a simulator may be the best and only viable alternative.

References
[1] S.R. Schach, Object-Oriented and Classical Software Engineering, 8th ed. New York: McGraw-Hill, 2011, p. 163.

My response was:

I have to disagree with two of your points:

First, you imply that it is possible to test all possible inputs and outputs through automation. This isn't true. Automated tests make life easier, but they aren't a magic fix that will be able to test an infinite number of possible inputs and outputs. It still requires prioritizing the most likely or riskiest inputs and outputs.

Regarding testing of an airplane black box: it's amazing what kind of testing those things get put through. They're shot out of cannons, set on fire, immersed in acid, and so on. According to wikipedia (and other articles I saw with some googling), a black box must be able to survive 3400 Gs, penetrations, high and low temperatures, pressures, and immersion [1]. If it's still able to record data under these conditions, I'd say it's safe to assume it would survive a plane crash.

I have to wonder though, has anyone deliberately crashed a plane to test one? I don't see anything online about that, but I don't see why it isn't feasible.

[1] Wikipedia. Flight Recorder. [Online]. Available at : http://en.wikipedia.org/wiki/Flight_recorder