Andrew Thomas presents

Learn the Crap Out of EECS 448

Since Gibbons is always light on things like slides and readings and, y'know, things that are good for studying before a test, I'm putting this together for the test tomorrow, hand-forged from my excellent decent good enough notes. So, going down his topics list,

Git

Gibbs calls this "**version control**," which it is, but the de facto standard of version control is git, so I take it as a sort of Kleenex vs. tissues-class semantics argument. The basis for git is a collection of files known as a **repository**. It's essentially a folder, but with added functionality. The main added functionality is the fact that keeps track of the state of the repo (because all the cool kids abbrev it) and creates a sort of timeline of repo states. These discreet states are known as **commits**. Each commit, other than the initial commit, has one parent that it points back to. This is helpful for when someone, say, your new bit of code introduces a major bug into the program. You roll back a version and the bug should be gone.

As we are all nerds and I mentioned a timeline, yes, there can be alternate timelines in git. Those are called **branches**. If, after a few commits, I want to add new functionality to a stable build, I would create a new branch off the **master branch** (the first branch made) and add and tweak the code until it worked as intended. This other branch should have a name, in this case "Otherbranch," because a branch name acts like a pointer into the heap of commits (Remember pointers and heap allocated memory from 268?).

WHAT THE SOFTWARE ENGINEER SAYS

If the branch **checks out** a different commit that is behind the **head** of the current branch, you run the risk of having a **detached head** on this branch.

WHAT THE SOFTWARE ENGINEER MEANS

If the branch **pointer moves to** a different commit that is behind **the latest commit** of the current branch, you run the risk of having a **commit or commits lost in the heap of commits, causing a pseudo-memory leak** on this branch.

Now you've got all these branches, but you only want to put out one finished product. That's when you **merge** the branches together. Remember earlier when I said that a commit has one parent? Not with merging. Merging has two parents. Now the biggest potential problem with

merging is that there could be wild differences between the two branches. These are **merge conflicts**. An important thing to remember is that *git does not resolve merge conflicts, you must do that manually*. These usually result from two different programmers working on adding different functionality to the project, so get the programmers together to hash it out, actually talk to the filthy meatbag that ruined your masterpiece.

Github

You see how I just mentioned more than one programmer working on a project? Git is a start, but Github really makes that work. Github is a website that hosts repositories that the public can see, **fork**, and download. When you fork a repo, you are copying a repo from someone else's Github page to yours. Forking takes place entirely in Github. If you want to work on the code in the repo on your own machine, you would **clone** it. Cloning creates a copy of the repo at the time to your machine with the git architecture in place. You can make new commits to the repo, but *these commits exist only on your machine*. To put these new commits on the Github repo, they need to be **pushed**. You can give specific Github users permission to push to your repositories, in fact, you should have already in your projects. The inverse of pushing is, huge surprise, **pulling**. Pulling updates your local repo with any new commits from the Github repo.

Agile Software Development/Scrum

Here's all you really need: http://www.scrumreferencecard.com/ Gibbs taught off this, so it should be good enough for you. The highlights are as follows:

- The old way was known as the **Waterfall method**: you plan, then act, then test, then launch, no going back at any point.
- **Agile development**, specifically Scrum (which is only one type of agile development), allows for cycling of the steps for constant testing and improvement of code until full launch. Much better than waterfall.
- Scrum is based around short meetings between the team, usually daily at the beginning of the work day, to get status updates and potentially troubleshoot. These are referred to as **Scrum Meetings or Scrums**.
- Scrum is based around **sprints**, shorter segmentations of the timeframe for the project for the team to focus on minute parts that need to be done.
- Scrum uses **backlogs** to note what needs to be done in a sprint/project, what has been done, and what can't be done.
- There are three titles in a Scrum team:
    1. **Product Owner** – He/She speaks for and to the customer, and acts as the de facto leader. He/She shouldn't take the credit for success, but they should take the blame for failure. May actually know what the rest of the team does, and therefore potentially help, emphasis on "potentially."
    2. **Scrum Team** – Everyone else in the team. They do as they are told…hopefully.

3. **Scrum Master** – A member of the Scrum Team who has the extra responsibility of reminding everyone that we do need to have the meeting today, guys, and general inter-team relationship counselor. Seriously, that's pretty much it. Heavy is the head that wears the Scrum Master crown.

UML

Unlike myself and anyone reading this, most of humanity has no idea how to read programming code. Unfortunately, this majority usually includes the people that decide if our professional projects go forward and any monetary compensation for said projects. Thus, UML fulfills the need to communicate code to the plebeians. The three types of UML we were introduced to were

1. **State Diagrams** – Remember the idea of a finite state machine? I know Mahmood hit it in 140 and Professor Minden seems infatuated with it from my time in 388 with him, but the highlight is that a computer program has a finite number of states and there are set way to change between these states. I may be biased, my part of my team's project four does use an enumeration to explicitly name my states, but if you just go for straight finite state machine in your project, this isn't that big of a problem.
2. **Use Case Diagrams -** I didn't believe him until I checked it out on Gliffy, but Gibbs was right: you use stick figures to show how different types of users can use the program. You know, like standard users versus admins versus plumbers or something.
3. **Class Diagrams** – This is the complicated one. The classes in the project are made into blocks with the name in the top part, the member variable in the middle, and the member methods in the bottom. Then the classes are connected via arrows and lines based on the nature of the connection like inheritance, implementation, association, etc. I don't have the resources in Word to do all the drawings here, so I recommend looking that stuff up on the Google.

Gantt Charts

I don't know about you, but this really feels like more UML. The difference is that this bit probably shouldn't be seen any users or customers. A Gantt Chart lists the individual tasks that need to be done and the time it would take to complete this task. It also links tasks together, showing certain tasks that cannot be started until a previous task has been completed. It's essentially a better sort of calendar-timeline thing.

Testing

To test a program properly you must test it thoroughly. To test a program thoroughly you must test every possibility. The idea of "every possibility" is known in this case as **coverage**. There are four coverages that need to be fulfilled for complete coverage.

1. **Function coverage** – Every function in a project, in every class, must be completely tested.

2. **Statement Coverage** – Every statement is executed.
3. **Branch Coverage** – Every if/else or switch branch must be entered/attempted.
4. **Condition Coverage** – Every piece of a Boolean expression gets set to both true and false.

The good news is that a single function call can cover multiple types of coverage, so you've got that going for you.

Code Smell

Gibbons calls it that, but I call it "Code Stank." This is how any programmer worth their weight can tell if an implementation is shit or not just by looking at it. The list of bad things is best left in bullet point form.

- One big main function
- One big main class
- One big class
- One big method
- Repeated code
- Extremely similar subclasses
- Subclass implementation that violates the expected behavior of the base class
- Method or methods with crazy long parameter lists (five or more)
- Excessive use of literals (variables are better than hard values)
- Extremely long variable or type names
- Extremely short variable or type names
- Methods that return too much
- Methods that require too much
- Excessive nesting in loops/ifs
- Lazy classes (they don't do much of anything)

SOLID Principles

On the other end of programming are the good programming indicators. The version Gibbons taught us is the SOLID Principles.

1. **Single Responsibility Principle** – A class should only have one responsibility
2. **Open-Close Principle** – A class needs to be open to extension
3. **Liskov Substitution Principle** – Derived types need to do essentially the same thing as base types
4. **Interface Segregation Principle** – Many small interfaces are better than one big interface
5. **Dependency Inversion Principle** – Your program should not depend on specific implementations