ECE 458
Engineering Software for Maintainability

Introduction and Course Overview
Tyler Bletsch

(Adapted from work by Drew Hilton)
Welcome to ECE 458: Engineering Software for Maintainability

Your Senior Design Course!
Evolution!

• You four years ago:

Fig. 1: Freshman year

• You now:

Fig. 2: You now have freaky robot hands I guess??
What this class is about

• Real software has a long lifespan
  • In industry, you might work the same code base for years or decades
• Contrast with code you write in school:
  • Turn it in, forget about it.
• Real world software’s requirements evolve
  • New features
  • Changing requirements
  • ...

Fig. 3: Software is like pokeymans

• How do we design software to ease later changes?
  • Goal of this class: learn this by doing and reflection
What this class is not

- This class is not about learning to program, you know that (well, you better know that...)

- This is not a lecture class
  - These are the first, last, and only slides I’ve prepared
  - You’ve been taught some software engineering skills, but...

- You learn by doing!
**Reflection**: To take time to think on what you’ve done, critically evaluate how it went, and extract lessons (both positive and negative) from it.

**Other courses**: I vomit green swirlies at you.

**This course**: You produce your *own* green swirlies.
What are we doing?

• One semester long project:
  • Requirements staged into **4 evolutions**.
    • Changes will usually be substantive restructuring of core ideas:
      Not “add this form”, but “change what this concept means”
  • After each evolution, submit a report. Major parts:
    • Retrospective (analysis of past design choices):
      • How did your past designs set you up to win or struggle?
      • How did these outcomes align with your prior analyses?
    • Forward looking (analysis of current design):
      • What are its key features? Why did you design it this way?
      • What do you see as its strengths?
      • How about its weaknesses?
Where’s all the stuff?????

It’s here:

http://people.duke.edu/~tkb13/courses/ece458/

Fig. 4: internet.
Project groups

• You will do your project in groups of 4
  • Pick carefully: fixed for the semester

• Considerations:
  • Language/framework choices
    • Note: subject of next discussion
  • Other tool choices
    • Revision control, ...
  • Skills and expertise
    • Ideal: strong skills, complimentary expertise

• End of class: find groups, start planning ev1

Fig. 5: All the best groups have four members
Project reports

- No specific page limit/requirement
  - Say what you need to say. Don’t say more, don’t say less.

- Expect document to be
  - Well-written:
    - Organized, clear, precise.
    - Include figures if they help
  - Analytical:
    - Delve into **why** your design is/was good/bad
    - Tell me what was bad, and how it could have been better
      - Hindsight is 20/20
    - Include discussion of **testing plan** (part of design)

Fig. 6a: WikiHow illustrations will never stop being hilarious.
Oral Presentations

- Day that evolutions are due: oral presentations
  - Each group member presents once
- 10 minutes per group
  - The *seemingly least prepared* group goes first!
  - Have your AV and laptop crap sorted out!
- Rough outline (2–3 min each)
  - Quick demo of working project
  - Retrospective from previous evolution
  - Overview of current design
  - Analysis of current design (include: why, strengths, weaknesses)

Fig. 6b: Please don’t look like this when presenting.
1. Class discussions:
   - Topics posted on class webpage (all posted now)
   - Some topics have readings – you need to read it before you come
   - Prepare ~1-2 pages of outline/notes on discussion, turn in at end
     - The notes are NOT a summary of the reading, but your thinking on the whole of the topic (reactions, opinions, questions, etc.).

2. Workdays
   - Work with your group on your project
   - I’ll circulate around, answer questions, offer advice, etc.

3. Presentation day
   - If presenting: present. Else: support your group!

4. Reflection day
   - Session after a due date
   - Reflect on work so far, discuss newly released requirements
Ask questions, please!

- Discussions are a great place to ask questions!
- In the past, students reported that they felt intimidated to ask about concepts that were introduced in discussion
  - But it turns out it was MOST students who felt this way!

- **Imposter Syndrome**: The phenomenon wherein “high-achieving individuals are marked by an inability to internalize their accomplishments and a persistent fear of being exposed as a ‘fraud’.”

- Affects 90% of Duke I think... (including myself)
Stand-up meetings

- A brief meeting where teams go over status, blockers, and open questions.
- Every Wednesday except for presentation days
- Noted on the class schedule

Fig. 8: Hahaha look at these blob people
Grading (1)

• 45% software deliverables:
  • How well did your system meet requirements?
  • Based on a **demo session** and **instructor functional testing**
  • We want to be as objective as possible, but in assessing “quality” without the benefit of a giant spec doc, there will be some subjectivity.
    • The system must actually be **good** from a customer perspective, not merely tick all the boxes.
    • Especially true for problems reported to you that you do not fix!
    • In other words, don’t try to “Air Bud” us.
      • [https://www.youtube.com/watch?v=Jvf0WWxrYRM](https://www.youtube.com/watch?v=Jvf0WWxrYRM)

• 25% written deliverables:
  • Technical/analytical content: how well did you describe/analyze?
  • Writing: how well written are your documents?
Grading (2)

- **10% oral presentation:**
  - Each group member does one evolution’s presentation
  - Rubric will be posted

- **20% class attendance/participation:**
  - Come to class regularly (2 free absences).
  - For discussion:
    - Have your discussion notes prepared (grading: 0, 70, or 100)
    - Actively participate in the discussions
  - For workdays/presentations/reflections:
    - Attend and participate as appropriate

- **No exams!**
NOW IS THE PART OF THE PRESENTATION WHERE I GIVE YOU ALL THE ANSWERS ABOUT HOW TO DO WELL
Advice from past students: 2017 (1)

From a survey given at the end of the semester.

- 3 useful lessons:
  1. Use a web framework that helps you hit the ground running, lets you learn quickly (good docs really help here), and is flexible enough to tweak as needed (look for 3rd party packages / plugins).
  2. Good organization of tasks, who is working on them, and when they need to be completed is essential (full-stack is your friend).
  3. TESTING IS EVERYTHING. IT MAKES ALL THE DIFFERENCE. DO IT.

- Take the time to actually design your UI with the same care that you do your backend, otherwise you will end up with a messy, unintuitive interface.

Set milestones, both individually and as a team, and hold each other accountable to them. Procrastinating work in this class will screw you, arguably even more so than in CS 308.

Keep it simple, especially early in the project. The teams that are best at adapting for requirements late in the semester are the ones who didn't try anything too crazy early on.

Take a look at when the first evolution is due, subtract at least three days from it, and go ahead and set in stone that you will have your code "done" by that time. You will need the extra days to actually test your software and fix bugs.
Advice from past students: 2017 (2)

- Stick to technologies that are either (a) familiar to you or (b) really well documented and supported. **Don't choose tech just because it sounds cool** -- the class is too short to learn how to do things the right way so use something that is easy to pick up.

Set aside a good amount of time for **testing**. This really really really really really really really really really really really really really really really really really really really really really really really really really really really really really really helped our team a lot. I **literally cannot stress how important it is to do this**. There was a huge(, visible) difference between teams that took the time test their software and those that didn't. Set aside a weekend just for testing and then fixing the bugs that you found during that period. Do this even if you haven't completely finished all the dev yet -- unless you are super far behind, it's definitely better to test most of your software than to wait to test all of your software only to realize that you don't have time to do any testing.

**Don't wait until the last day to deploy to prod**. Prod is messy. Shit breaks. You don't want late nights in dev ops hell.

**Pick a good team**. You are going to be spending a lot of time talking to/with each other.

- If learning a development framework feels overwhelming, **create a specific plan for learning** it. Ex: rather than "spend 2 hours today, 2 hours tomorrow" use "watch this tutorial series today, read through this documentation tomorrow, implement xyz the next day".
Advice from past students: 2018 (1)

- The four most important things are to
  1) PICK A GOOD GROUP and decide how decisions will be made and how work will be allocated within the group before starting evolution 1.
  2) LISTEN TO DR. BLETSCH, and make a fine-grained schedule with dates right when a new evolution is released. Make sure this schedule has time for a code freeze, testing (both locally and on test/prod), and ALLOCATIONS of jobs to people within your group (beware the bystander effect)
  3) START EVERYTHING EARLY I DON'T CARE HOW GOOD YOU THINK YOU ARE AT CODING AND DEBUGGING.
  4) COMMUNICATE, COMMUNICATE, COMMUNICATE. If you need help, ask for it (again, I DON'T CARE HOW GOOD YOU THINK YOU ARE AT CODING AND DEBUGGING).

- Actually do research about different frameworks/platforms for the project. Don't put this off! This one decision determines your fate for the rest of the semester.

- One piece of advice: learn the language before you start writing production code. For example, if you are using javascript, but haven't coded in javascript before, take about 5 hours to fully understand the nuances of the language. Understand array operations. Understand promises, and how they are almost always better than callbacks. You will ultimately regret "diving in" and writing code without any knowledge of the language.

- Make sure that you meet with your team as soon as possible after evolution requirements are posted so that you can make a fine grain schedule with dates to avoid unnecessary all nighters.

- Make a schedule and stick to it! A deadline needs a schedule.
Advice from past students: 2018 (2)

- In your first evolution: get automated testing done for one thing on your backend. Then in your second evolution: Extend that automated testing to everything else. "I don't automate because I don't have time. I don't have time because I don't automate!" My group fell into this loop, and it would have been really nice if we hadn't.

- Don't procrastinate, communicate with your team effectively, don't be afraid to ask for help from your teammates if you don't know what's happening. Meet your code freezes and spend a lot of time testing, test extensively. Start the evolutions early, just because it seems like you have a lot of time, doesn't mean you actually have a lot of time. Don't be afraid to re-write or re-factor if you think it'll be worth it; saying we already started that way so let's keep going is a bad excuse. Meet with your team frequently, even if it is a quick meeting. Break down the evolutions and assign tasks to specific member instead of letting members pick up tasks as they go, and have a schedule. Finish the blockers first, or else it's a waste of time.

- Logistics is a really big part of this course, so spend at least a day each evolution planning. Good planning will lead to success in the evolution.

- For each evolution, set a code freeze date. But make sure this code freeze date does not include testing and then allow a few days for testing because you will find unexpected bugs right before the deadline. Setting a code freeze date too close to the deadline does not allow enough debugging time.

- Plan out your design well. Although it may feel like you're not making a lot of progress by planning it will pay back over the course of the evolution.

Read all 2018 advice here
so just do all that stuff and you'll be good
Academic Integrity

• Expect academic integrity from all of you
  • Duke community standard
    • I will not lie, cheat, or steal in my academic endeavors, nor will I accept the actions of those who do
    • I will conduct myself responsibly and honorably in all my activities as a Duke student.

• Concrete rules:
  • Discuss anything you want
    • Give credit where it's due if you use other groups' ideas
  • All code should be produced within your group
    • Don't share code outside your group
    • Can use libraries for graphics, sound, etc (e.g., SDL) as needed

• Not sure? **ASK**
Specifics of the Project

Food Manufacturing Production Lifecycle and Business Decision Support System

• Many specifics are left up to you
  • Web based? Desktop application? Mobile app? Your choice

• All 4 evolutions are (almost) already written
  • I will not change them in response to your status
    (Some students worried that I’d put in their worst fears)
Requirements

• Requirements will be distributed as PDFs
  • New requirements in blue
  • Changed requirements have old requirements in grey
    • (replacements in blue)

• Unclear on requirements? Ask
  • Happy to clarify anything

• Unspecified requirements/behavior?
  • Do anything reasonable

• Contradiction? Is a requirement stupid?
  • Discuss it in class or on the forum; changes may be possible

• Don’t need to be artistic
  • But it does need to be efficient and usable!
Variance request: A formal process for requesting a change in the requirements for your group.

To submit a variance request, post a Piazza thread with the "variance" tag as well as the appropriate evolution tag. In the body, fill out the following template.

This way we can track variances on a per-group basis easily and get your precise change down in writing. (CAREER TIP: Always get stuff in writing!)

Group number: ___
Group name: ___
Requirement number(s) affected: ___
Requested change: ___
Rationale: ___
Submission

- Submission of projects by repository pull
  - Your choice of revision control system
  - Coordinate with TA on submission

- Server:
  - Have \textit{at least} a test server and a production server
  - \textbf{Production server} \textit{should only be touched in the week before the evolution is due} – \textit{otherwise it’s frozen!}
    - We test on this deployment when you’re not around!
  - Recommend VM from OIT: \url{https://vcm.duke.edu}
  - NOT recommended: the various fly-by-night “free” hosting providers
    - Students have been screwed by this before...

- A note on platform:
  - You must document ALL environmental pre-requisites and instructions for setup of your product
  - If you do anything mobile, please include instructions for emulator
QUESTIONS?
Minor logistics

• For next time, need to select:
  • Four to present a programming language + framework
    • See course site for details
A realistic beginning

- The formal requirements have been published, but they may not be clear yet...

- To get started, I recommend you interview the customer:
  (as impersonated by me)
Meet your customer

- Large, industrial-scale food manufacturer
- Production and business decisions currently run by spreadsheets and manual effort
- They want a system to organize everything
- Each team is a contractor vying for their business
A brief primer on industrial food manufacturing

- The company **produces food SKUs** (Stock Keeping Unit) on **manufacturing lines** from **ingredients**.
- SKUs are packed in **cases** which are sold to **customers** (e.g., Walmart, Food Lion, 7-11, etc. – not you the consumer)
- The **production scheduling manager** decides when to produce which SKUs and in what quantity based on anticipated demand.
  - Anticipating demand is a hugely difficult problem.
- The **product manager** evaluates the **profitability** of SKUs to propose new SKUs and to recommend discontinuing underperforming ones.
  - Accounting for costs and revenue is a complex problem.
Understanding food manufacturing

- Factory produces **units** packed into **cases**.
  - A **unit** is an individual consumable, e.g. a can of soup.
  - A **case** is a carton or other container of many such units packed for sale to a customer’s store (e.g. a box of 128 cans of soup).
  - **Customers** (e.g. Walmart) buy **cases**, and cases are how sales of company products are tracked.
  - **Consumers** (e.g. you and me) buy **units**, and units are how consumption of products is tracked.
  - Sales from a customer to a consumer are called **turns**.
Additional info

• Have to respect **shelf life:**
  • **Total shelf life:** Maximum time from production to consumer use
  • **Internal shelf life:** Portion of total shelf life before product goes to customer
  • **External shelf life:** Portion of total shelf life from being held by a customer to being bought+used by a consumer

• Need to **exchange data** with other people/systems
  • The company is a worldwide multinational organization
  • Sales information is a complex and ugly patchwork of human effort, spreadsheets, and variously ancient databases both internal and from the customers.

• Need to produce **reports** to compute relevant information for various kinds of users
  • **Report:** a set of computed data derived from stored data intended for human consumption. Usually table-formatted.
Evolution 1: Go!

- Find your groups

- Start trying to get the requirements out of the customer (me)

- Maybe even talk about your design?
  - What are the key objects to model?
  - Decide how to split up the work?
  - What do you think the main challenges will be?
  - How should you design to accommodate whatever changes I throw at you?
  - What programming language do you want to use?
    - Detailed discussion on Monday.
  - What procedures and tools to use?