CS1530, LECTURE 4

PRINCIPLES OF SOFTWARE ENGINEERING

Building The System

Building a small system, much of what we covered may seem unnecessary or even a waste of time.

As systems get larger and more complex, the benefits of good design, test plans, predetermined system configuration, etc. are manifest.

“

Simplify, simplify, simplify!
I say, let your affairs be as two or three, and not a hundred or a thousand…

- Henry David Thoreau

”

PROBLEM AND DESIGN DECOMPOSITION

- Need to move from complex problems to simple problems
- Necessary for both design and implementation
- Modern software is generally too complex to keep in one person’s head
- There are various techniques for simplifying design and code

PROBLEM AND DESIGN DECOMPOSITION

- Decomposition - Turning large problems into lots of small problems
- Modularization - Grouping the system into different subsystems
- Separation - Making sure that parts of the program work as independently as possible, reducing the number of dependencies
- Incremental iterations - Shipping/releasing/versioning multiple iterations, instead of trying to get everything done in one release

“CONVENTION OVER CONFIGURATION”

- Not mentioned in the book, but an idea which is becoming more popular is assuming convention over configuration
- Example: Ruby on Rails
  - Basic applications always uses Model-View-Controller architecture - models in /models subdirectory, controllers in /controllers subdirectory, views in /views subdirectory, linked by names
  - Configuration using YAML files
  - Can generate a skeleton app usings “rails new”, modify as needed

BENEFITS OF CONVENTION OVER CONFIGURATION

- Avoids “bike-shedding” - arguing over unimportant details (analogy is debating color of bike shed
- Easy to switch from one project to another
- Can start with a “working” project and make changes
    - Parallels how humans think

DRAWBACKS OF CONVENTION OVER CONFIGURATION

- Can be stifling, hard to do something in a different manner
- Leads to engineers perhaps not understanding the system nearly as well as one they built from scratch
- Who makes the conventions? Can lead to centralization (e.g. DHH in Rails world)
    - This could be a benefit or a drawback!
    - See "Worse is Better" article, especially in regards to splintering: https://www.dreamsongs.com/WIB.html

TECHNICAL CONSIDERATIONS

- Which programming language to use?
- What database or kind of database?
- Which operating systems should it run on?
- Which software process methodology to use?
- How will we support/maintain the software?
- How will the software be tested?
- What will count as sufficient test coverage?

Note: some of these may be covered as design constraints, others will be technical decisions during implementation.

NON-TECHNICAL CONSIDERATIONS

- How we will communicate status to, and receive feedback from, stakeholders?
- How do we determine deadlines and estimate completion?
- How do we get the “correct” people on our team and maintain morale?
- What do we do if we cannot complete the software or otherwise meet the requirements?
- Are there any legal issues which we should be aware of?

WE KNOW PROJECTS FAIL

- Book references 1995 CHAOS Report, in which researchers found that only 16.1% of software projects are considered “successful”, where success means..
    - On-time
    - On-budget
    - Meets the needs of customers
- Other classifications: “challenged” (completed, but does not meet one of the criteria) - 52.7% and “failed” (cancelled or not completed) - 31.1%

1994! THAT'S ANCIENT HISTORY!

- 2015 CHAOS report from Standish Group
- 29% of all software projects are successful.
- At first blush, this is good - almost twice as many percentage points
- But this still means that over two-thirds of all successful projects are not successful!

WHY DO THEY FAIL - FUNDAMENTAL CAUSES

Challenged projects:

1. Lack of user input
2. Incomplete requirements / specification
3. Changing requirements / specifications

Failed projects:

1. Incomplete requirements / specifications
2. Lack of user involvement
3. Lack of resource

WHY DO THEY FAIL - PHASES

- Requirements errors -12.5%
- Design errors - 24.17%
- Code errors - 28.33%
- Documentation errors - 13.33%
- Bad-fix errors - 11.67%

BUT…

- Errors earlier on propagate forward, e.g.
  - Requirements error -> Design error -> Implementation error
  - Design error -> Implementation error
- Earlier problem is introduced - bigger cost to fixing
- Later problem is caught - bigger cost to fixing
- Fixing problems in earlier phases (problem-preventative) may be most efficient way of reducing problems

WHY DO THEY FAIL - SIZE

- Grand - 2% successful
- Large - 6% successful
- Medium - 9% successful
- Moderate - 21% successful
- Small - 62% successful

WHY DO THEY SUCCEED?

- User involvement
- Executive management support
- Clear requirements statements
- Proper planning

NOTE WHAT IS MISSING FROM THE PREVIOUS SLIDES!

- Programming issues
- Programming language choice
- Framework choice
- Poor algorithms
- Text editor choice
- Using tabs instead of spaces

SOFTWARE ENGINEERING

"…an engineering discipline whose focus is the cost-effective development of high-quality software."

-David Parnas

WHO GETS TO CALL THEMSELVES AN ENGINEER?

- In some fields, you can become a Professional Engineer by getting a license - only PEs can call themselves an engineer.
- This usually involves work experience, training, testing, etc.
- In Software Engineering, this is still in its infancy (Texas Board of Professional Engineers offers a Software Engineering license, Australia, several Canadian provinces as well)
- But mostly, anyone can call themselves a software engineer
- Benefits and drawbacks to this!

PRINCIPLES OF SOFTWARE ENGINEERING

- Unlike physics or math, there are no real underlying physical laws related to software engineering. We have heuristics.
- Alan Davis's Principles of Software Engineering was an early attempt at this
- I am going to cover the most important of Royce's Principles , but first I want to cover Davis's last principle:

> ***Take responsibility****: If you developed the system, then you should take responsibility to do it right. Blaming the failure on others, on the schedule, or the process is irresponsibe.*

PROFESSIONALISM AND MANAGING UP

- As a software engineer, even if you are not a licensed Professional Engineer, you should be a *professional*
- A professional is someone whose greatest allegiance is to the *profession,* not their current job
- You are hired for your knowledge, not just your labor. This means that you are not just expected, but obligated, to:
    - "Manage up" - Inform your manager if you think they are following an incorrect course, piling up too much work on you, etc.
    - Act ethically (what this means is complex!)
    - Taking responsibility for your actions

ROYCE’S PRINCIPLES OF SOFTWARE ENGINEERING

1. Base the process on an architecture-first approach.

2. Establish an iterative process that addresses risks early in the process.

3. Emphasize component-based development to reduce the coding effort.

4. Change management should be established to handle an iterative process.

5. Enhance the iterative development process environment, called round-trip engineering, where multiple changes may occur constantly across multiple artifacts with automated tools.

6. Use model-based and machine-processable notation to capture design.

7. Establish the process for objective quality control and project progress assessment of all the intermediate artifacts.

8. User a demonstration-based approach where intermediate artifacts are transitioned to executable demonstration of the user scenario so that these artifacts can be assessed earlier.

9. Plan to have incremental releases, each composed of a group of usage scenarios, with evolving levels of detail.

10. Establish a configurable process, because no one process is suitable for all software development.

ARCHITECTURE-FIRST APPROACH

“Base the process on an architecture-first approach.”

- Think about how the subsystems should interact and what is general data flow of the application before writing code
- Example: file line sorter vs video game
- File line sorter: straightforward data flow (input -> output), no significant internal state, single output
- Video game: complex event-driven data flow, very significant and modifiable internal state, constant output

ITERATIVE PROCESSES

" Establish an iterative process that addresses risks early in the process."

- Risk is unavoidable - we should mitigate it, not try to remove it entirely
- Part of mitigation is figuring out and addressing risks early on
- Have an "eat that frog" mentality:
    - *"Eat a live frog first thing in the morning and nothing worse will happen to you the rest of the day."* -often erroneously attributed to Mark Twain, probably a paraphrase of Nicholas Chamfort
- Risk-based decision-making

COMPONENT-BASED DEVELOPMENT

"Emphasize component-based development to reduce the coding effort."

- Think of the system as a collection of components - relentlessly componentize!
- Opposite of this would be a *monolith,* where every part of the system can influence any other part
- Work as much as possible on getting individual components to work, and then work at a higher level getting them to work together

OBJECTIVE QUALITY CONTROL

"Establish the process for objective quality control and project progress assessment of all the intermediate artifacts."

- There should be an independent auditor/assessor of quality and progress
- This should be as objective as possible (percentage of features completed, for example)
- Software developers are always about 90% done and their software is always great (if you ask them)

DEMONSTRATION-BASED APPROACH

" User a demonstration-based approach where intermediate artifacts are transitioned to executable demonstration of the user scenario so that these artifacts can be assessed earlier."

- It's harder to lie (to yourself or the customer) when showing a demo than showing a whitepaper
- A demonstration is worth a thousand words - lets users see problems/issues/give feedback much more easily
- Gives an objective goal to developers for intermediate artifacts

INCREMENTAL RELEASES

"Plan to have incremental releases, each composed of a group of usage scenarios, with evolving levels of detail."

- Walker Royce (despite sometimes being credited with coming up with the Waterfall model of software development, aka, the SDLC we saw earlier) knew the importance of incremental releases
- Find issues early, calibrate, adjust
- Always have working software, even during intermediary stages!

A CONFIGURABLE PROCESS

"Establish a configurable process, because no one process is suitable for all software development."

- We have already mentioned this - what is appropriate for developing control for a web start-up is not appropriate for nuclear power plant safety controls, and vice-versa
- In the next lecture, we will cover a group of these methodologies, Agile, in which process re-configuration is a key aspect