Wherefore* Software Engineering?

“SOMETHING MYSTERIOUS IS FORMED, BORN IN THE SILENT VOID. WAITING ALONE AND UNMOVING, IT IS AT ONCE STILL AND YET IN CONSTANT MOTION. IT IS THE SOURCE OF ALL PROGRAMS. I DO NOT KNOW ITS NAME, SO I WILL CALL IT THE TAO OF PROGRAMMING.”

-- GEOFFREY JAMES

*Actually means “Why” or “For what reason”
Why, Mr. Anderson?

First, a philosophical question:

*Why do we program?*
The Joys of the Craft [2]

- Brooks: “Why is programming fun?”
  - The sheer joy of making things
  - The pleasure of making things that are useful to other people
  - The fascination of fashioning complex puzzle-like objects
  - The joy of always learning → the task is always new in some way or other
  - The delight of working in such a tractable medium
    - Almost pure thought-stuff
    - Real in the sense that it produces output separate from itself
The Woes of the Craft [2]

- However, the very things we enjoy about programming can also bring us grief:
  - The complex objects must be constructed **perfectly** in order to work as intended
  - To make something useful, we often must meet the **requirements of others**
  - Some of the pieces we must use are **previously built code and tools that may be poorly designed and constructed** (with terrible documentation)
  - Designing grand concepts is fun, but **debugging** can be **painful** and often **takes longer than expected**
  - The flexibility of programming often means we are given **plenty of room to mess up**
  - The product you make is often **obsolete** (at least in design) the day you release it
Why Do We Need Software Engineering?

So, why do we need Software Engineering?
- Problem with Scale
  - The larger software gets, the more problems arise
- Problem with Novelty
  - The software you develop is often a new idea/product that has no exact predecessor/pattern to follow

Need a defined way to do the following with multiple people on board:
- Design and plan what you’re going to make
- Code and build the product
- Manage and schedule work
- Test the final product to make sure 1) it’s what you want and 2) it works
What is Software?

The “software” in “software engineering” can be:

- **Program**
  - Single executable
  - *Used by very few people* (maybe just the developer)

- **Product**
  - *Used by many people*
  - Used in environment(s) outside of development environment → *generalization*
  - Needs *testing, documentation, and maintenance*
    - I.e., a polished product

- **System**
  - *Group of programs that work together*
  - Interfaces and integration between parts can be complex

- **System Product**
  - Polish of a *product*
  - Multiple parts of a *system*
The Problem with Scale: Complexity

- The more complex the software becomes, the more time, energy, people, debugging, etc. the software will need.
- In general [2]:

<table>
<thead>
<tr>
<th>Program</th>
<th>Programming System</th>
</tr>
</thead>
<tbody>
<tr>
<td>3x cost</td>
<td>3x cost</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Programming Product</th>
<th>Programming Systems Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Generalization, Testing, Documentation, Maintenance)</td>
<td>9x cost!</td>
</tr>
</tbody>
</table>
The Problem with Scale: Teamwork

- Software projects are often large
  - Assign too few people → takes too long
    - Customer/client unhappy
    - Software will be obsolete on release
  - Assign more people:
    - Must ensure proper division of labor
      - Either stuff doesn’t get done and/or work overlaps
    - Increased communication lines
    - Issues with ensuring everyone on the same page

\[ C(n,2) = \frac{n!}{2!(n-2)!} \]
The Problem with Novelty

- Software projects are often new
  - Usually making a new design or addressing a new problem
    - Unlike other kinds of craftsmanship, once you’ve built a piece of software, you can copy it infinitely
  - Often a “wicked problem”
    - I.e., don’t know what issues you’ll have until you start building it
What is Software Engineering?

- **Software Engineering**
  - “…may be defined as the systematic design and development of software products and the management of the software process” [1]
  - Consists of three components [1]:
    - Design
    - Development
    - Management
Software Engineering Defined

- **Design**
  - System/module/class/program/data design
  - Requirements
  - Architecture
- **Development**
  - Coding
  - Testing
  - Debugging
- **Management**
  - Work breakdown / team organization
  - Cost and resource estimation
  - Scheduling and workflow
  - Quality control
“ONCE UPON A TIME, IN A LAND STEEPED IN METAPHOR, THERE LIVED AN ELEPHANT.”

-- “BALANCING AGILITY AND DISCIPLINE”
Software Development Metaphors

- If you understand something well, you can make a good metaphor to explain it
- Conversely, a good metaphor helps explain a concept or process

- Let’s look at some of the metaphors used to describe software development (and the pros and cons of each)...
Software Penmanship: Writing Code

- **Software as “writing code”**
  - No formal planning
  - Like writing a casual letter
  - Figure out what you need as you go
- **Doesn’t fully describe software development:**

<table>
<thead>
<tr>
<th>Writing</th>
<th>Software Development</th>
</tr>
</thead>
<tbody>
<tr>
<td>Usually one-person job</td>
<td>Multiple people</td>
</tr>
<tr>
<td>Not changed as easily or often</td>
<td>Usually modified heavily (typically 2/3 after initial release)</td>
</tr>
<tr>
<td>Originality important</td>
<td>Reuse of previous working designs/code/tests important</td>
</tr>
</tbody>
</table>

- **Rigid metaphor**
- **Implies “plan to throw one away” strategy**
  - Expensive trial and error rather than careful planning
Software Farming: Growing a System

- Creating software like planting seeds and growing crops
  - Design a piece
  - Code a piece
  - Test a piece
  - Add to system
  - Repeat
- Good → encourages incremental approach
- Bad → suggests you don’t have direct control over development
  - Also, most of the implications don’t make sense (rotating in a crop of C++?)

Accretion = any growth or increase in size by a gradual external addition or inclusion

Software Development should be built through accretion (like the way oyster makes pearl)
- Make simplest version of system that will run
  - Skeleton with dummy classes and functions
  - Doesn’t take realistic input or output
- Add to system a little at a time
  - E.g., add code to accept real input
- Related terms: incremental, adaptive, evolutionary

Good metaphor, but it leaves a lot of details unspecified
Building Software

- **Building software**
  - Implies planning, preparation, and execution
  - Compatible with software accretion, but provides more detailed guidance
  - Addresses issues of scale
    - Small scale (e.g., dog house)
      - Loose design
      - Mistakes not a big deal
      - Can build it yourself fairly quickly
    - Large scale (e.g., mansion/skyscraper)
      - More involved design and planning
      - Mistakes (whether in design or construction) more costly, in particular with labor
      - Need more people
      - Usually need quality control (inspections)

Building Software

- Software development like building a house:
  - Decide what kind of house → **problem definition**
  - Make general design → **software architectural design**
  - Draw detailed blueprints and hire contractor → **detailed software design**
  - Prepare site, lay foundation, frame house, put on siding/roof, plumb/wire it → **software construction**
  - Painting/decorating/landscaping → **software optimization**
  - Inspections of foundation, wiring, etc. → **software reviews and inspections**
Building Software

- **Buy vs. Build**
  - Probably going to **buy** prefabricated doors, dishwashers, windows, etc.
  - May decide to **build** custom cabinets, furniture, etc.

- **Planning**
  - *Underplanning* → fixing mistakes later will be more expensive
  - *Overplanning* → don’t need to know color of wallpaper when laying foundation

- **Variable quality control**
  - Dog house → low quality requirements
  - Nuclear reactor → high quality requirements
“THUS SPAKE THE MASTER PROGRAMMER:

‘AFTER THREE DAYS WITHOUT PROGRAMMING, LIFE BECOMES MEANINGLESS.’”

-- GEOFFREY JAMES
Software Construction

- **Software Construction** focuses on the following:
  - Detailed design
  - Coding
  - Debugging
  - Integration
  - Developer testing (unit testing and integration testing)
More specifically, Software Construction activities include:

- Verifying all groundwork (requirements, architecture designs, etc.) ready so construction can proceed successfully
- Determining how your code will be tested
- Designing/writing classes and routines
- Creating/naming variables and constants
- Selecting control structures and organizing blocks of statements
- Unit testing, integration testing, and debugging your own code
- Reviewing other team members’ low-level designs and code and having them review yours
- Polishing code by carefully formatting and commenting it
- Integrating software components that were created separately
- Tuning code to make it faster and use fewer resources
Why Software Construction?

- Large part of software development
  - About 30% to 80% of total project time
- Central activity in software development
  - After requirements/architecture, before testing
- Can improve individual programmer’s productivity
- Source code is often the only accurate description of the software
  - Code always up to date
  - Code therefore must be of the highest quality
- Construction is the only activity guaranteed to be done
Software Toolbox

- Remember, as you go through this course, different software development methods/approaches are like tools in your intellectual toolbox
  - No single tool is right for EVERY job
  - Choose the right tool(s) for the right job!
- Early document discussing what software and software engineering is, as well as the state of the art in software engineering pedagogy and techniques...in 1980.

- Easily one of the most important software engineering texts, and much of it still resonates today.