What is Engineering? Software Engineering?

Motivation

Software Methodology

Modularity

Reading: Brookshear 7.1 -- 7.3, 7.8 -- 7.9

---

Software Engineering

**What is Engineering? Software Engineering?**

**Motivation**

**Software Methodology**

**Modularity**

Reading: Brookshear 7.1 -- 7.3, 7.8 -- 7.9

---

**What is Engineering?**

- As an introduction to software engineering, consider what engineers in other fields do.
- According to Wikipedia the main branches of engineering are: aerospace engineering, chemical engineering, civil engineering, electrical engineering, mechanical engineering. What do practitioners in these fields do?

---

**Motivation**

- The need for a discipline of software engineering is illustrated by examples of systems that failed.
- Quotes from *After the Gold Rush*, by Steve McConnell:
  - Problems with the baggage handling system caused a delay of more than a year in opening Denver International Airport. Estimates of the delay’s cost ranged as high as $1.1 million per day.
  - The FAA’s Advanced Automation System overran its planned budget by about $3 billion.
  - The IRS bumbled an $8 billion software modernization program that cost U.S. taxpayers $50 billion per year in lost revenue.
  - After topping $44 million, a long series of overruns forced California to cancel its motor vehicle registration system.
  - The B-2 bomber wouldn’t fly on its maiden flight because of a software problem.
Motivation

- Quotes (cont’d):
  - The Ariane 5 rocket blew up on its maiden launch because of a software error.
  - In Seattle, computer-controlled ferries caused more than a dozen dock crashes in the 1980s, resulting in damage worth more than $7 million. The State of Washington recommended spending more than $3 million to revert the ferries to manual controls.
- To summarize:
  - designing and implementing software is hard
  - potential for errors increases dramatically with larger projects
- Society depends more and more on computers and computer software
  - developing reliable “best practices” for software engineering is an important part of computer science

Software Engineering

- How does the design and implementation of software systems compare to engineering in other fields?
- Software engineers also take a problem-solving approach to their discipline
  - analyze requirements for a system
  - consider how existing software might be applied, or define requirements for new software
  - design new software systems
  - implement and test applications
- Software engineers also belong to professional societies
  - IEEE, ACM, others
  - this is still a new field: certification, standardization, professional ethics are still evolving

Software Engineering

- But there are several differences between software and technology used in other fields

**Materials**
  - chemical, mechanical, electrical engineers rely on science to describe materials they work with
  - attributes of materials help define what is possible (e.g. strength of steel determines length of span)
  - software applications are made out of .... ? ideas?

**Metrics**
  - engineers estimate amount of material required, which helps predict costs for projects (e.g. cost of steel is a major component in overall project cost)
  - years of experience also help estimate time involved for a project
  - how does one measure the complexity of a software project?
  - how does one predict how long it will take to implement a system?

Reliability

- bridges, building, electrical grids, plastics, and other artifacts are built to specified strength (e.g. adding more steel will increase the reliability of a bridge by a predictable amount)
- the tiniest defect in software can cause a catastrophic failure
- [see the Fortran example on the next slide]

- In spite of these differences computer scientists working in SE look to the other fields of engineering as a model
  - comparison of similarities, differences helps the field grow and mature
Famous Fortran Bug

- Fortran was one of the first programming languages
  - idea for a “formula translator” was conceived in 1953
  - first compiler was available in 1957
- Modern versions are radically different but the language is still widely used
  - popular for large scale “number crunching” (weather prediction, fluid dynamics, ...)
  - “we don’t know what the programming language of the future will be, but we do
    know it will be called ‘Fortran’”
- Early versions of the language used line numbers to label statements
  - a “do loop” in FORTRAN 77:

```fortran
DO 20 I = 1,100
... 20 CONTINUE
```

In Ruby:

```ruby
for i in 1..20
...
end
```

Famous Fortran Bug

- An odd detail about Fortran: the compiler didn’t care if you put spaces in
  variable names
  - these assignment statements are all equivalent:
    ```fortran
    my int = 1
    myint = 1
    my i n t = 1
    ```
- What is the difference between the following two programs?
  - what does the compiler think the programmer intended to do?

```fortran
DO 20 I = 1,100
...
20 CONTINUE
```

```fortran
DO 20 I = 1.100
...
20 CONTINUE
```

Famous Fortran Bug

- The program on the left has a DO loop that iterates 100 times
- The program on the right has an assignment that sets a variable named DO20I
  to 1.1
- There is literally just a one bit difference between these programs
  - ASCII comma = 2C (00101100), period = 2E (00101110)
- This famous bug caused a NASA program to compute the wrong trajectory for
  a rocket
  - folklore: this bug caused Mariner 1 to veer off course and need to be destroyed

```fortran
DO 20 I = 1,100
...
20 CONTINUE
```

An Example: wc

- To illustrate some of the pitfalls of software engineering, let’s develop an
  application to count words in a file
- Plan:
  - read each line
  - break a line into words
  - update counters for number of lines, words, characters
  - print the counters after the last line has been read
- Since Ruby already has methods to split lines and count lengths
  of strings and arrays this should be trivial

```ruby
nlines = 0
nwords = 0
nchars = 0
while line = gets
  nlines += 1
  nwords += line.split.length
  nchars += line.length
end
puts [nlines,nwords,nchars]
```

How does this compare to the Unix program named wc?
An Example: wc

- Let's try it out:
  ```bash
  % wc.rb < voters.txt
  7 28 187
  Success! We're ready to release wc.rb 1.0
  ```

- But 10 minutes after PayPal tells us someone downloaded the program and bought a license we get an angry e-mail from a blogger:
  ```shell
  Your program doesn't work! My post has 7 words, but wc.rb says it has 6! I want my $10 back!!
  ```

An Example: wc

- We go to his blog and find the post:
  ```plaintext
  The Republicans didn't steal the election!
  ```

- What's going on here? Let's use irb to see what happens with his string:
  ```ruby
  >>> s = "The Republicans didn't steal the election!"
  >>> s.split
  => ["The", "Republicans", "didn't", "steal", "the", "election!"]
  >>> s.split.length
  => 6
  ```

- Looks like six words -- but the blogger claims "didn't" is two words.

- After a lengthy e-mail debate we finally agree to add an option to expand contractions into two words and release wc 1.1

An Example: wc

- But soon we start getting more e-mail
  ```plaintext
  Why are URLs counted as one word? There should be four words in "http://www.blogosphere.com"
  ```

- Pretty soon we have to hire a programming staff, contract with tech writers to produce a manual to explain all the options, raise the price for "WC Pro" to $195, ...

An Example: wc

- The moral of this story?
  ```plaintext
  even a little bit of planning and analysis -- in this case a little bit of thought about the requirements for this program -- could have saved a lot of trouble
  ```

- Postscript: one simple definition of "word" is "consecutive string of letters and/or digits"
  ```bash
  line.split(\W+) note upper case W
  ```

- using this definition in wc.rb 1.0 could have avoided a lot of trouble:
  ```bash
  $ = "You can't tell me the Republicans didn't steal the election!!!
  $ = $ .split(\W+/
  $$ = ["You", "can", "t", "tell", "me", "the", "Republicans", "didn", "t", "steal", "the", "election"]
  ```
The Software Life Cycle

- The "life cycle" of a software application is not very different from other artifacts
  - houses, roads, airplanes, and other things are designed and built
  - as they are used they start to wear and need to be maintained
- Software maintenance has a different focus
  - the algorithms and code don’t wear out
  - "maintenance" usually means fixing bugs or adding to the functionality
  - changes to the environment (new hardware, upgraded operating systems, etc) may also require maintenance of applications

Development Process

- The development phase of the software life cycle is often broken into four parts
- Requirements Analysis
  - after the initial idea is conceived, developers need to be very specific about what the software will do
  - what are the inputs? the outputs? the environment the program will run in?
  - who are the stakeholders? who will use the software? who will be affected by it?

Development Process

Design

- in this phase the major components are identified
  - will the system need a graphical user interface? a database?
  - the main goal of this phase is to break the project into manageable pieces
  - different groups can work independently on the parts
  - one of the difficulties: make sure the parts are consistent

Development Process

Coding

- this is the phase that is often associated with "programming"
  - implementers translate the design specifications into working code

Testing

- the completed code needs to be thoroughly tested
  - each individual module should be checked ("unit testing")
  - interaction between modules ("integration tests") and other tests are also done

It was a really bad strategy to test wc . z?b on just one file...
Software Methodology

- There is widespread agreement on what the major phases of development are.
- How to move from one phase to another is not as clear.
- A software methodology is a set of guidelines and methods for carrying out each phase.

Waterfall

- In the waterfall methodology one phase is completed before moving on to the next.
- Often associated with “top-down design” or “stepwise refinement.”
- This process seems logical but can be very difficult in practice.
- Failures in any phase can trigger re-designs of each earlier phase.
- People who like to write and test code are frustrated by long, drawn-out design phases.

Incremental Design

- With an incremental design methodology one builds a very simple version of the initial system.
- Often just a prototype with little or no functionality.
- E.g. show the placement of windows and menus.
- Maybe show some fixed content so clients get early feedback on what the final product will look like.

Incremental Design

- Through a series of very small incremental changes new pieces of the software are implemented and tested.
- Each new version moves successively closer to the final system.
- Advantages:
  - Stakeholders can participate early in the testing phase.
  - Errors are often uncovered soon after they are made.
    - They don’t lie “dormant” until exposed by users.
Extreme Programming

- A radical form of incremental design is **extreme programming** (aka “agile programming”)
- Proponents advocate starting the coding and testing phases right away, even before requirements are completely known
- Requirements evolve along with the rest of the system
- Use teams of programmers
- Continual testing -- test after each small change

Modules

- No matter which methodology is used the final result will be a collection of **modules**
- The modules might be separate programs (e.g. clients and servers)
- They might be functions in a library (e.g. math routines, layers in a protocol, ...)
- They might be separate procedures or functions in the same program
- The correct **functional decomposition** of a complex operation into separate pieces is crucial
- Techniques for structuring programs in Ruby include
  - **Methods**
  - **Classes** (related collection of methods and data)
  - **Modules** (major pieces of code, also collections of methods that can be “mixed in” with classes)
    - Example: Comparable is a module that implements comparison operations

Example: wc.rb 1.1

- As a simple example of functional decomposition compare the original organization of wc.rb (left) and version 1.1 (right):

```ruby
while line = gets
  nlines += 1
  nchars += line.length
  nwords += line.split.length
end
puts [nlines,nwords,nchars]
```

- The new version has two parts:
  - A **main routine** that reads the input, and a subroutine that breaks lines into words

```ruby
def word_count(s)
  t = s.gsub("n't"," not")
  return t.split.length
end
while line = gets
  nlines += 1
  nchars += line.length
  nwords += word_count(line)
end
puts [nlines,nwords,nchars]
```

As the definition of "word" changes make updates to the subroutine without modifying the rest of the program

Refactoring

- The first step in updating wc.rb is to add the new method
- But note that this change should not alter the output
- The goal is to simply restructure the program so it has two modules instead of one
- Immediately test the new version to make sure the restructuring did not introduce a bug
- Making a change to a program's structure without modifying it's input/output behavior is known as **refactoring**
- An important part of iterative design
Reuse

- One of the benefits of refactoring and modular design is that pieces of code developed for one program can be reused in other programs
- Examples of software reuse:
  - subroutine libraries
  - class libraries in object-oriented languages
  - utility programs (ls, wc, sort, and others in Unix) that can be “pasted together” in shell scripts
  - client and server applications that can be called from Perl, Ruby, and other scripting languages

Summary

- Software engineering is an increasingly important field
  - some view it as a specialization or subarea in computer science
  - others argue it is a new field, requiring expertise in computer science, business, and other areas
- Software engineering is more than simply “programming”
  - methods used to implement simple programs do not work for large, complex systems designed by groups of people over several years
- The four main phases of a software methodology are
  - requirements analysis (with participation of stakeholders)
  - design (including functional decomposition)
  - coding
  - testing

Summary

- The traditional waterfall methodology moves sequentially from requirements to testing
- iterative design starts with construction and testing of a simple prototype
  - additional functionality is added in small steps
  - test after each modification
- To learn more:
  - read about testing in Brooksheer (“white box testing”, “black box testing”, quality assurance, and other issues)
  - read about Test::Unit in Ruby
- Later this week:
  - Wed: open source software development
  - Fri: guest lecture on human-computer interaction