Principles of Software Engineering

... and their embodiment in the JAVA2 Platform

What is [Software] Engineering?

- Engineering is about technology
  - Ultimately, engineering is concerned with the design of things involving technology
  - Engineers are the bridge between scientists (theory) and technicians (implementation)
- Software engineering is the process/study of creating technology using software
Isn’t Software Technology an Oxymoron?

- New technology usually means building new physical devices that solve new problems…
- Software is what makes a GPC into an application specific problem solver
- We can build new problem solvers in software the same way we can build new problem solvers in hardware

Hardware Simulations

- Anything that can be done in hardware can be done in software…with some limitations
  - If physical interaction with the real world is needed, the software must be able work in concert with real hardware that will do the dirty work … think of a car assembly line …
  - You have to have a fast enough processor to run your hardware simulation
- Bottom line – new technology CAN be built in software
So I’m a SE already. Right?

- Since I can write programs, that means that I can create new pieces of technology
- Therefore I must be a software engineer
- Well, maybe, but probably not

The Big Question

What’s the difference between a computer program and a piece of software?
## Popular Answers

- Nothing, two words, same exact thing
- Size does matter
  - Software is composed of many programs
  - Software are a really big programs
- Software is backed by a company…
  - Software comes in a shrink wrapped box
  - You have to pay for software
  - Software is a “product”

## A Comparison

<table>
<thead>
<tr>
<th>Software</th>
<th>Programs</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Long design phase before being built</td>
<td>- Usually designed while being built (F9 F9 F9 style)</td>
</tr>
<tr>
<td>- Documentation available (comments &amp; user guides)</td>
<td>- May not even have a single comment inline</td>
</tr>
<tr>
<td>- Tested at the module and integrated product level</td>
<td>- May not ever have been tested at all</td>
</tr>
<tr>
<td>- Backed by a support contract or SLA</td>
<td>- Support might be available depending on the author</td>
</tr>
<tr>
<td>- Revision control, end-of-life planning</td>
<td>- Author is just happy “it works”</td>
</tr>
</tbody>
</table>
In one word…

LIFECYCLE

Software has a lifecycle…

- Conception occurs during a design phase
- Birth occurs during a building phase
- Growing up with integration & testing phase
- Adulthood with a support phase
- And it even has a retirement plan

- As for programs… they tend to be haphazardly thrown together, aka “hack”
The Waterfall Model

- Design
- Build Modules
- Unit Test
- Integration
- Verify/Validate
- Production
- Support
- Retirement

Waterfall Model

- Design
- Build Modules
- Unit Testing
- System Integration
- Verification & Validation
- Production
- Support
- Retirement

- Good in theory & for teaching but not very realistic
- Never revisits the drawing board…
- What about future versions of the same software?
Cyclical Model

- Design
- Build
- Test
- Production/Support (get customer feedback)
- Design next version
- Build next version

- Popular in the mainframe days
- Good for projects where you can spend a long time between releases
- Once again, need to revisit design more

Spiral (RAD) Model

- Phases are designed to be very short and quickly recurring
- Permits redesign when errors are encountered
- Results in faster software revision cycle times
- Cost may be higher
The Microsoft Model

- Design
- Build
- Production/Ship It
  - Build Service Pack
  - Ship Service Pack
  - Repeat for next SP
- Build new version
- Production/Ship It
  - Build Service Pack
  - ... ...
- Written up in a book by a Microsoft Executive
- Claimed that fast cycle times is what allowed Microsoft to capture the market
- Where is/are the testing phases?

The Design Phase

- A single bug caught in support phase can easily cost millions of dollars
- The same bug caught at design costs only a few dollars
- When creating software, most of the time should be spent thinking about the design
Building Components

- Similar to typing an essay from an outline
- The key to good quality components is proper documentation
  - API (application programming interface) guide
  - High level users guide with notes on restrictions
  - Comments inline in the code

A study at CMU showed that the longer the code is, the more bugs there are

Modularized building is critical…
- Many people can work on the same project
- Bypasses problem of long code being buggy (system is composed of lots of small modules)
- OOP/OOD is based on this principle
Hierarchies

- You can try to super design everything then build it all in one shot…
  - … but only for relatively simple projects

- A better idea is to build and iterate
  - Start with simple general ideas, build and test objects that encapsulate those ideas
  - Extend those simple objects and you retain all of that work done in designing and testing

Documentation in JAVA

- Inline comments
  - Designated by //, continues to end of line
  - Should be as many lines of inline comments as there are executable comments

- User’s guide, API, etc., via javadoc
  - /**, *, */ before each field, method and class
  - Run javadoc myfile.java to generate HTML
/**
 * A class to represent some <B>abstract</B> object in space. This is
 * where I would put some <I>other</I> info about this class.
 */
public class SpaceObject {
  /**
   * A field to hold the goods. Should be < 10 at all times.
   */
  int theGoods = 0;
  /**
   * Run this method to get some stuff back. It's all the go
   */
  public int getStuff() {
    // save the output
    int output = theGoods;
    // clear out the holder for the goods (we're giving them away..)
    theGoods = 0;
    // send back what the user wants
    return output;
  }
}
Exceptions

- An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions
- Remember that events are user generated “things that happen” with a particular technical meaning in GUIs

Let’s say we’ve got a method…

- … that performs some sequence of events
- `readSomeDevice` {
  - open the device
  - figure out how much to read
  - create our destination object
  - actually do the reading
  - close the device
}
The traditional approach…

```
errno readFile() {
    int errorCode = 0;
    open the file;
    if (theFileIsOpen) { // determine file length
        if (gotTheFileLength) { // allocate memory;
            if (gotEnoughMemory) { // read the file
                if (readFailed) { errorCode = -1; }
            } else { errorCode = -2; }
        } else { errorCode = -3; }
        close the file;
        if (fileNotClosed && errorCode == 0) { errorCode = -4; }
        else { errorCode = errorCode and -4; }
    } else { errorCode = -5; }
    return errorCode;
}
```

The exception approach…

```
readFile() {
    try {
        open the file;
        determine its size;
        allocate that much memory;
        read the file into memory;
        close the file;
    } catch (fileOpenFailed) {
        doSomething;
    } catch (sizeDeterminationFailed) {
        doSomething;
        . . . . . . . . . . . . . . . . . . .
    } catch (fileCloseFailed) {
        doSomething;
    }
}
```
Exception Advantages

- Well organized (non-spaghetti) code
- Propagation of exceptions up the stack
  - If an exception occurs in myMethod which is called by otherMethod called in main, the exception can be handled (caught) in main
- Exceptions can be categorized hierarchically
  - Catching IOException handles all exceptions that are in the IO class, catching Exception gets them all

Creating an Exception

- Declare a class that extends Exception (or some other class that extends Exception)
- It is good practice to put the suffix “Exception” onto the name of the class
- Naming the exception properly is extremely important because the name needs to describes what is going wrong
Syntax details

- Declare your method with "throws myException" (or any predefined exception)
- In your method, put the code:
  ```java
  throw new myException("special messages")
  ```
- In your calling code, put a try/catch block:
  ```java
  try {
    myMethod();
  } catch(myException m) {
    // some error handling code
  } catch(Exception e) {
    // generic handling code
  } finally { // do some cleanup }
  ```

Unit Testing

- Make sure that your individual components function correctly before plugging them into the full system
- In JAVA, put a main method into every class
  - Run every method, check to make sure it does what it is supposed to do
  - These test harnesses will never be called by the final, fully integrated system
Unit Testing Example

```java
public class myStuff {
    private int theGoods = 0;
    public void pickItUp() { theGoods = 1;
    public void dropItOff() { theGoods = 0;

    /**
     * This is the test harness for our class. It fully
     * Exercises every method in the class.
     */
    public static void main(String [] args) {
        myStuff blah = new myStuff();
        blah.pickItUp();
        if (blah.theGoods != 1) { System.err.println("ERROR!"); } }
        blah.dropItOff();
        if (blah.theGoods != 0) { System.err.println("ERROR!"); } }
    }
}
```

Integration and V&V

- Large pieces software of software as usually integration incrementally
  - Additional test harnesses are then needed to make sure the increments work properly
- Sometimes, unit stubs need to be created
  - These simulate the functionality of a unit that is not yet ready or requires user interaction
Production, Support, EOL

- Usually, these are glossed over, since they are about business practice more than anything else.
- However, proper design and module building makes production, support and retirement / next revision production much easier.