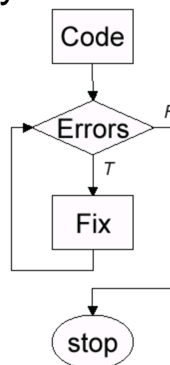
Introduction to Software Engineering and the Software Lifecycle

CSCE A401

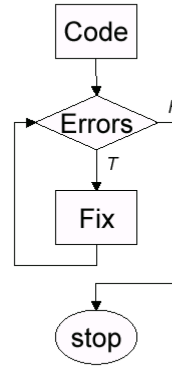
Software Engineering

- Theories and practices used to construct high-quality large-scale software
- How you may have created many programs:



Code and Test

- Works fine for small problems
- Undesirable for larger problems
  - No well defined phases
  - How do you know you're building the right thing?
  - Little/No time to test
  - Success achieved by hacking skills and luck
  - Difficult/Impossible to repeat successes
    - Future maintenance



Large Systems – Software Engineering

- Present day applications are big
  - Curiosity Rover: 2 MLOC
  - Ubuntu Linux Kernel: 14 MLOC
  - Windows 7: 50 MLOC
  - Are generally not developed by their users
- Relying on programming ability alone is not adequate
  - Scope too large, too many people, modules, processes, ill-defined requirements and perspectives
- This class is not about how to program
  - Software engineering is still considered an art rather than a craft

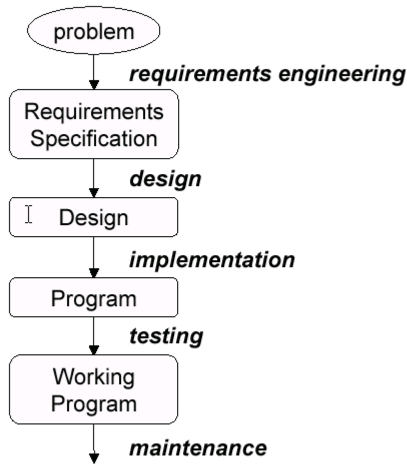
Software in the 60's

- Increasingly large software systems, increasing problems (and still today!)
  - Delivered late
  - Did not behave as expected
  - Not adaptable
  - Have maintenance problems
- This became known as the **Software Crisis**
- Solution
  - Develop software using a more theoretical, sound, and proven basis, like engineers
  - Hence **Software Engineering**
  - Building software should be done like building bridges or automobiles?

What is Software Engineering?

- First NATO Conference, 1968
  - Software engineering is the establishment and use of sound principles in order to obtain economically software that is reliable and works efficiently on real machines.
- IEEE Std Glossary of Software Eng. Terminology
  - Software engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software; that is the application of engineering to software.

Simple Lifecycle Model



Requirements Engineering

- Objective: a description of the problem to be solved, the requirements posed by the environment
- Requirements
  - Functional (What the system should do)
  - Non-functional (Hardware, users, etc)
- The description includes: functionalities, future extensions, amount/type of required documentation, performance and response time
- Part can be a Feasibility study
- The more careful the requirements engineering phase, the larger the chance that the ultimate system will meet expectations
  - All people must collaborate intensively
- Resulting document is the **Requirements Specification**

Design

- A model of the whole system is developed
  - Not programmed yet, but when programmed would solve the user's problem
  - Decomposed into modules, components, interfaces
- Global description of the system captured in the **architecture**
  - May be evaluated, serve as template for similar system, reusable components
- Separate the what from the how
  - Annoying preamble to the real work??
  - End of the design phase can include pseudocode

Implementation

- Concentrates on individual modules
  - Adheres to the software architecture and specifications from the design phase
  - First goal should be well-documented, reliable, easy to read, correct program – not one full of tricks!
- Result of the implementation phase is an executable program
- Often eased by use of pseudocode during the design phase

Testing

- Testing should actually be performed throughout all phases, not only after implementation is finished
  – Cheaper to correct errors the earlier they are detected; errors can occur in requirements, design, and implementation
- Testing at phase boundaries
  – Verification: transition between subsequent phases is correct
  – Validation: on track meeting system requirements

Maintenance

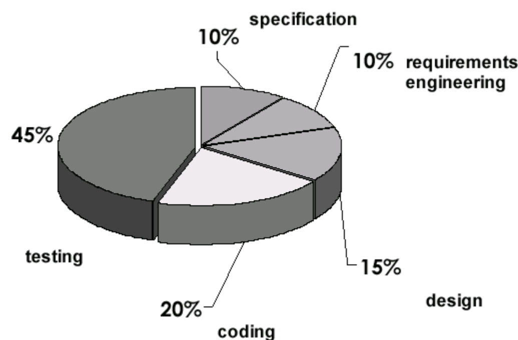
- Manage changes after delivery
  – Perfective (changes in user requirements)
  – Adaptive (changes in the environment)
  – Corrective (removal of faults)
  – Preventive (for future maintenance of the system)

Spanning All Phases

- Project Management
  – Planning, team organization, quality issues, cost, schedule estimation, etc. to ensure the project is delivered on time and on budget
- Documentation
  – Must start early
  – Often a balancing item; tends to be pushed back for other items
  – Software not well documented has higher costs later when changes occur

Typical Effort for Each Activity

- 40-20-40 rule: Only 20% of the effort is spent on actual coding



Maintenance Activities

- Not shown in previous chart
  - Over lifetime of systems, maintenance grows to 50%
- Typical percentages for maintenance
  - Perfective 50%
  - Adaptive 25 %
  - Corrective 21%
  - Preventive 4 %

Spectacular Failures – Need for SW Engineering

- Therac-25
  - Radiation treatment machine malfunction
  - Delivers small doses of radiation through filters to treat cancers, tumors
  - Six deaths due to lethal dose of radiation before fixed

Therac-25

- Updated version of Therac-20
  - Hardware interlocks stopped machine if errors occurred
  - Therac-25 designers thought the software was good since techs never reported any problems with Therac-20
  - Software errors resulted with no ill effect, so many errors on screen they were ignored
- Therac-25 : hardware interlocks replaced with software
  - Flag: when no errors in setup, flag set to zero
  - But only 1 byte for errors, if 256 errors there was overflow back to 0
  - Machine thought tests passed when they really failed

Therac-25

- "That means that on every 256th pass through Set-Up Test, the upper collimator will not be checked and an upper collimator fault will not be detected.

  The overexposure occurred when the operator hit the "set" button at the precise moment that Class3 rolled over to zero. Thus Chkcol was not executed, and F$mal was not set to indicate the upper collimator was still in field-light position. The software turned on the full 25 MeV without the target in place and without scanning.

  …

  AECL described the technical "fix" implemented for this software flaw as simple: The program is changed so that the Class3 variable is set to some fixed nonzero value each time through Set-Up Test instead of being incremented. "

An Investigation of the Therac-25 Accidents
Leveson & Turner

Therac-25

- Two errors here: human process and accuracy
  - Took two years to diagnose and fix
  - Lesson: Can't separate software process from hardware
  - Need for robust software testing

Mars Climate Observer

- Observer lost 9/99
- Lockheed Martin provided thrust data in pounds, JPL entered data in Newtons
- Ground control lost contact trying to settle observer into orbit

- Process/Communications/Human error, not really a software problem

Real-Time Anomaly

- Example: Mars Pathfinder
  - Lander/relay for Sojourner robot
  - Onboard computer would spontaneously reset itself
  - Reported by the media as a “software glitch”
  - Used embedded real-time operating system, vxWorks

Pathfinder Problem – Priority Inversion

- Pathfinder contained an information bus
  - Data from Pathfinder’s sensors, Sojourner went on bus toward earth
  - Commands from earth send along the bus to sensors
- Must schedule the bus to avoid conflicts
  - Used semaphores
  - If high-priority thread was about to block waiting for a low priority thread, there was a split-second where a medium-priority thread could jump in
  - Long-running medium priority thread kept low priority thread from running which kept the high-priority thread from running
- Good news: watchdog timer noticed thread did not finish on time, rebooted the whole system
- Noticed during testing, but assumed to be “hardware glitches”. The actual data rate from Mars made the “glitch rate” much higher than in testing

Pathfinder

- Fortunate that JPL engineers left debugging code that enabled the problem to be found and remotely invoke patch
- Patch: Priority Inheritance
  - Have the low priority thread inherit the priority of the high priority thread while holding the mutex, allowing it to execute over the medium priority thread
- Such race conditions hard to find, similar problem existed with the Therac-25
- Reeves, JPL s/w engineer: “Even when you think you’ve tested everything that you can possibly imagine, you’re wrong.”

Mars Rover : Spirit

- “Spirit began acting up last week, when it stopped sending data and began rebooting its computer, resetting it roughly 130 times. At one point, the rover thought it was 2053.”
- Bug Description
  - Engineers found that the rover's 256 megabyte flash memory had retained hundreds of files containing flight data and was still juggling them along with the daily flood of new data from its activities in Mars' Gusev Crater.

Spirit

- Workaround
  - By commanding Spirit each morning into a mode that avoids using the flash memory, engineers plan to begin deleting hundreds of unneeded files to make the memory more manageable for the rover's RAM.

- WHY WASN'T THIS CAUGHT IN TEST?
  - The bug had not been detected in operational tests of the rover on Earth because the longest tests lasted only eight or nine days.

Approximation/Accuracy

- Patriot Missile Example
  - More embedded software
  - Fault in the guidance software
  - Cumulative timing fault
- Radar detects missile, calculates where the Scud will be within its range gate
- Requires accurate determination of velocity

Patriot Missile

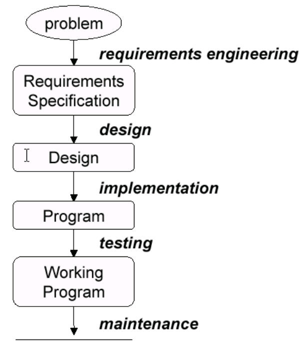
- Patriot's internal clock: 100 ms
- Time: 24 bit integer
- Velocity: 24 bit float
- Loss of precision converting from integer to float!
  - Precision loss proportional to target's velocity and the length of time that the system is running
- When running for over 100 hours, range gate shifted by a whopping 687 meters
- Perhaps just even worse: bug known beforehand, not fixed until after incident due to lack of procedures for wartime bug fixes

Failures

- These could have been caught by:
  - Software environments to detect errors
  - Better requirements and specifications
  - Better design
  - Better testing
  - Closer involvement between programmers and stakeholders

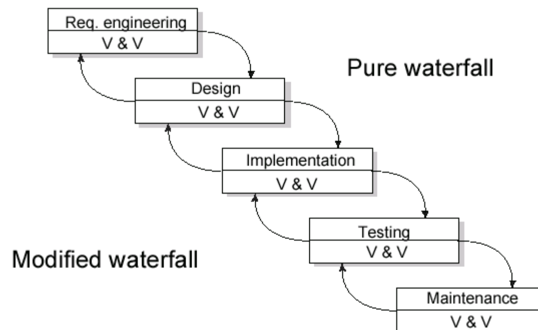
Lifecycle Models

- Simple Model
  - Document Driven
    - Next phase reached as documents are produced
  - Problems
    - Feedback lacking
    - Maintenance is often really evolution
- Other models are available
  - We will focus on a different model, Agile Programming, in this class



Waterfall Model

- Slight variation of simple model
- Verification (meets specs) and Validation (meets user requirements)
- Emphasis on a careful analysis before the system is actually built



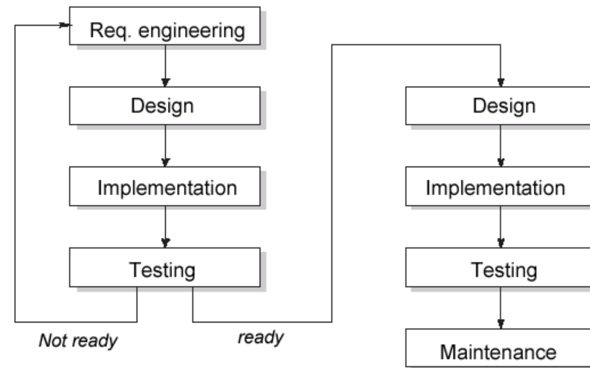
Waterfall Model

- Verification and Validation after each step
  - Attempts to find and fix errors early
- Like building a house
  - Ensure a solid foundation, frame, build your way up from there
- Problems
  - Too rigid, developers cannot move between phases
  - User might not be able to express what they want
  - Imagine putting in an order for a software system upon entering the store; no opportunity to look around, try things out, customize, etc.

Prototyping

- Motivation: Requirements elicitation is difficult
  - Software is developed because the present situation is unsatisfactory
  - However, the desirable new situation may be as yet unknown
- Aspects
  - Prototyping is used to obtain the requirements of some aspects of the system
  - Prototyping should be a relatively cheap process
  - Use rapid prototyping languages and tools
  - Not all functionality needs to be implemented
  - Production quality is not required

Prototyping as a Tool for Requirements Engineering



Types of Prototyping

- Throwaway prototyping
  - The nth prototype is followed by a waterfall-like process (as shown on the previous slide)
  - Recommended but rarely used; difficult to discard a (partly) working system
- Evolutionary prototyping
  - The nth prototype is delivered
  - More common

- Pro's and Con's of both approaches?

Prototyping Advantages

- The resulting system is easier to use
- User needs are better accommodated
- The resulting system has fewer features
- Problems are detected earlier
- The design is of higher quality
- The resulting system is easier to maintain
- The development incurs less effort

Prototyping Disadvantages

- The resulting system has more features
- The performance of the resulting system is worse
- The design is of less quality
- The resulting system is harder to maintain
- The prototyping approach requires more experienced team members

Prototyping Recommendations

- Use prototyping when the requirements are unclear or ambiguous.   Good way to clarify the requirements.
- Particularly useful for systems with emphasis on the user interface.
- The users and the designers must be well aware of the approach and its pitfalls. Users must realize the prototype is not production-quality.
- Prototyping needs to be planned and controlled to avoid limitless iterations.

Incremental Development

- A software system is delivered in small increments
  - E.g. a few features at a time
  - Avoids the “big bang” effect
- The waterfall model is employed in each phase
- The user is closely involved in directing the next steps
  - additional functionality is added if and when it is really needed; this prevents over-functionality