http://flickr.com/photos/nicholaslaughlin/1571649629/

- The Regulators
- The Spark Plugs
- A
- B
- E
- F
- G
- H

Go teams!

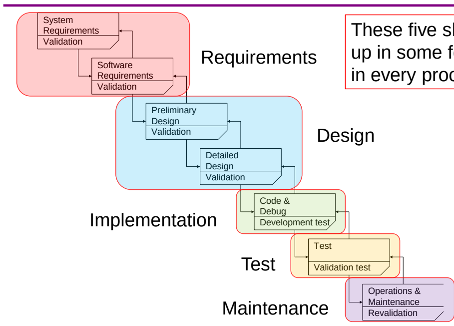
CSE403: Software Engineering

David Notkin
Winter 2009

Today

- Stages: requirements, design, etc.
- Milestones: deadlines and artifacts representing progress through particular phases
- The Budget Project: open discussion (~10 minutes)
  - Comments, concerns, complaints, ideas, etc.

Classic waterfall



These five show up in some form in every process

Requirements

- Functional requirements are intended to describe the functions that the system is to execute – more broadly, the goals the system is intended to achieve
- Non-functional requirements are intended to constrain the solution – these might include constraints on performance, maintainability, reliability, etc.

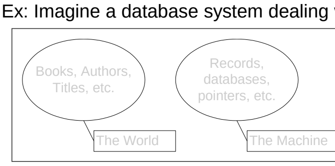
- The classic and overly simplistic distinction is that the requirements represent “what” the system should do and the design/implementation represent “how” it should do it

What vs. how

What	**How**
• Requirements	• Design
• Specification	• Implementation
• Declarative	• Operational
• Higher-level	• Lower-level
• Interface	• Implementation

The machine and the world

- Michael Jackson suggests a more fundamental distinction between requirements and program
  - The requirements are in the application domain
  - The program defines the machine that has an effect in the application domain
  - Ex: Imagine a database system dealing with books



Success

- A system is judged not by properties of the program, but by the effects of the machine in the world
- You don't care how Caller ID works, just that it works
- TCAS is a collision-avoidance system for commercial aircraft
  - Pilots love it (on the whole) because it helps them fly more safely and easily — not because it has great data structures

Failures: havoc in the world

- The Therac-25 killed real people
- The Word 3.0 failures caused real people to lose real information
- Security holes in Internet browsers allow confidential information to be stolen

Two requirements challenges

- To figure out the desired effects (requirements) of the machine in the world
- To figure out how to write this down in an effective way

Determining the "right" requirements

- Requirements analysis, requirements discovery, requirements elicitation, requirements engineering, etc.
- This is extremely hard: largely, it's ill-defined and the customers are usually (legitimately) unsure about what they really want
- I won't present a high-level discussion today, but will cover a specific (but general) technique on Monday

Writing it down

- It will help clarify what you think
- It is necessary to communicate with your users
- It is necessary to communicate with your team members
- It could form the basis for a contractual relationship

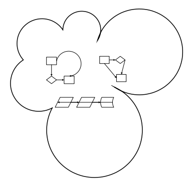
- Approaches include
  - Natural language
  - Structured natural language
  - Formal language

Use cases: a very quick preview

- A use case is a description of an example behavior of the system as situated in the world
  - Jane has a meeting at 10AM; when Jim tries to schedule another meeting for her at 10AM, he is notified about the conflict
- Similar to CRC (class responsibility collaborator) and eXtreme programming "stories"

Design [any noun can be verbed]

- There are many designs that satisfy a given set of requirements (functional and non-functional)
- There are also many designs that may at first appear to satisfy the requirements, but don't on further study
- Collectively, these form a design space
- A designer walks this space evaluating designs



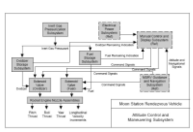
Software design

- Largely a process of finding decompositions that help people manage the complexity
  - Understand that the design satisfies the requirements
  - Allow relatively independent progress of team members
  - Support later changes effectively
- Not all decompositions are equally good
- A decomposition specifies a set of components (modules) and the interactions among those modules
  - At various levels
- Different methods for finding decompositions
  - Structured analysis and design
  - Object-oriented design
  - Aspect-oriented design
  - …
- Different criteria for assessing designs
  - Coupling and cohesion, complexity, correspondence, correctness, …

Design representations

- As many (at least) as there are design methods
- UML, dataflow, JSD, ERD, …

http://www.ukoln.ac.uk/qa-focus/documents/case-studies/case-study-03/qa-uml



http://upload.wikimedia.org/wikipedia/commons/thumb/3/31/IDEF_Diagram_Example.jpg/120px-IDEF_Diagram_Example.jpg

Implementation

- There is an "I" in implementation

Testing, verification, validation

- Validation: "Did we build the right system?"
  - Primarily a requirements-level (upper lifecycle) issue
- Verification: "Did we build the system right?"
  - Primarily a lower lifecycle issue (design, implementation, testing)

Approaches to verifying software

- Testing
  - A dynamic approach
- Program verification
  - Use math to show an equivalence between a specification and a program
- Process
  - Improving the likelihood that code is correct
  - Software inspections, walkthroughs, reviews; CMMI, ISO 9000, …

A false dichotomy

Testing **Proofs**

Why would we use one approach?

Terminology

- A *failure* occurs when a program doesn't satisfy its specification
- A *fault* occurs when a program's internal state is inconsistent with what is expected (usually an informal notion)
- A *defect* is the code that leads to a fault (and perhaps to a failure)
- An *error* is the mistake the programmer made in creating the defect

Kinds of testing

- Unit
- White-box
- Black-box
- Gray-box
- Bottom-up
- Top-down
- Boundary condition
- Syntax-driven
- Big bang
- Integration
- Acceptance
- Stress
- Regression
- Alpha
- Beta
- Fuzz
- …

Maintenance

- Use an existing code base as an asset
  - Cheaper and better to get there from here, rather than starting from scratch
  - Anyway, where would you aim for with a new system?
- The usual joke is that in anything but software, you'd love to receive a legacy

Why does software change?

- Software changes does not change primarily because it doesn't work right
  - Maintenance in software is different than maintenance for automobiles
- But it changes instead because the technological, economic, and societal environment in which it is embedded changes
- This provides a feedback loop to the software
  - The software is usually the most malleable link in the chain, hence it tends to change
  - [Counterexample: Space shuttle astronauts have thousands of extra responsibilities because it's safer than changing code]

Kinds of change

- Corrective maintenance
  - Fixing bugs in released code
- Adaptive maintenance
  - Porting to new hardware or software platform
- Perfective maintenance
  - Providing new functions

- Oft-cited data from Lientz and Swanson (1980) focused on IT systems – about 17%, 18%, 65%, respectively
- Modern data? There is some … not too different

Total life cycle cost

- Lientz and Swanson determined that at least 50% of the total life cycle cost is in maintenance
- There are several other studies that are reasonably consistent
- General belief is that maintenance accounts for somewhere between 50-75% of total life cycle costs

Open question

- How much maintenance cost is "reasonable?"
  - Corrective maintenance costs are ostensibly not "reasonable" (OK, this is easy)
  - How much adaptive maintenance cost is "reasonable"?
  - How much perfective maintenance cost is "reasonable"?
- Measuring "reasonable" costs in terms of percentage of life cycle costs doesn't make sense

High-level answer

- For perfective maintenance, the objective should be for the cost of the change in the implementation to be proportional to the cost of the change in the specification (design)
  - Ex: Allowing dates for the year 2000 is (at most) a small specification change
  - Ex: Adding call forwarding is a more complicated specification change
  - Ex: Converting a compiler into an ATM machine is …

(Common) Observations

- Maintainers often get less respect than developers
- Maintenance is generally assigned to the least experienced programmers
- Software structure degrades over time
- Documentation is often poor and is often inconsistent with the code

- Is there any relationship between these?

Laws of Program Evolution
Lehman & Belady

- Law of continuing change
  - "A large program that is used undergoes continuing change or becomes progressively less useful."
- Law of increasing complexity
  - "As a large program is continuously changed, its complexity, which reflects deteriorating structure, increases unless work is done to maintain or reduce it."
  - Cleaning up structure is done relatively infrequently: even with the recent interest in refactoring, this seems true. Why?

Milestones

- Artifacts that are intended to explicitly represent information about a particular stage at specific points in time in a software lifecycle
- A zillion variants

403: we'll use two

- SRS: requirements
- SDS: ~design
- Templates on project page
- Examples of both on
  http://www.cs.washington.edu/education/courses/403/08sp/projects403.html

Questions?

Budget game: open for discussion