Other Key Software Processes:
Maintenance, Migration, Evolution, PDSS
Reuse, Reverse Engineering, Reengineering

Computer Science 520/620
Spring 2012
Prof. Leon Osterweil

The Processes of Software Engineering

• Artifact synthesis processes:
  --Requirements, design, coding, test planning
• Other topics are not as centrally focused on artifact synthesis. Examples include
  --Maintenance/Evolution
  --Reengineering
  --Reuse
  --Configuration Management

Maintenance/Migration/Evolution

• Meeting the need for software products to change after they have been delivered
• Maintenance: older term, now falling out of favor and use
  --Connotes “fixing” things--eg. errors
  --Also used to cover adding features, responding to changes in requirements, usage contexts
• Most “maintenance” effort is not devoted to fixing errors:
  --More usual motivations are: changes in requirements, altered usage contexts, addition of new features
  --New software systems change the world around them: change the requirements that spawned them
• Migration and Evolution are more descriptive--becoming more popular designations of this activity
  --Post Deployment Software Support (PDSS) is military term

Need to Change is Constant

• Software is part of larger systems that are part of real world
• The world changes ==> systems, and software must too
• Systems and their software change the way the world works
  --Changes requirements for the systems and software
• Inescapable conclusion: All software that gets used will require change
  --Change is the rule, not the exception
  --Changing software used to be considered an indication of trouble, weakness, problems
  --Changes were done hastily, stealthily, sloppily
  --Change must be planned for/designed in (recall: likely directions of change must be part of requirements spec)
  --Software migration/evolution should be orderly, visible process

What is Involved in Doing This?

• Recall: > 2/3 of total lifetime cost of a software system is attributable to “maintenance”
• Partly because software is continually evolving--so this process continues indefinitely
• Mostly because this process is very hard
• Maintaining interartifact consistency is what makes this so hard
• Altering separate artifacts is straightforward: Keeping them consistent is not
• Changing one artifact (eg. code or requirements) compels making consistent changes to all related artifacts
  --And transitively
• Helps to know the relations tying the product together

The Processes of Software Engineering

• Artifact synthesis processes:
  --Requirements, design, coding, test planning
• Other topics are not as centrally focused on artifact synthesis. Examples include
  --Maintenance/Evolution
  --Reengineering
  --Reuse
  --Configuration Management

Maintenance/Migration/Evolution

• Meeting the need for software products to change after they have been delivered
• Maintenance: older term, now falling out of favor and use
  --Connotes “fixing” things--eg. errors
  --Also used to cover adding features, responding to changes in requirements, usage contexts
• Most “maintenance” effort is not devoted to fixing errors:
  --More usual motivations are: changes in requirements, altered usage contexts, addition of new features
  --New software systems change the world around them: change the requirements that spawned them
• Migration and Evolution are more descriptive--becoming more popular designations of this activity
  --Post Deployment Software Support (PDSS) is military term

Need to Change is Constant

• Software is part of larger systems that are part of real world
• The world changes ==> systems, and software must too
• Systems and their software change the way the world works
  --Changes requirements for the systems and software
• Inescapable conclusion: All software that gets used will require change
  --Change is the rule, not the exception
  --Changing software used to be considered an indication of trouble, weakness, problems
  --Changes were done hastily, stealthily, sloppily
  --Change must be planned for/designed in (recall: likely directions of change must be part of requirements spec)
  --Software migration/evolution should be orderly, visible process

What is Involved in Doing This?

• Recall: > 2/3 of total lifetime cost of a software system is attributable to “maintenance”
• Partly because software is continually evolving--so this process continues indefinitely
• Mostly because this process is very hard
• Maintaining interartifact consistency is what makes this so hard
• Altering separate artifacts is straightforward: Keeping them consistent is not
• Changing one artifact (eg. code or requirements) compels making consistent changes to all related artifacts
  --And transitively
• Helps to know the relations tying the product together

The Processes of Software Engineering

• Artifact synthesis processes:
  --Requirements, design, coding, test planning
• Other topics are not as centrally focused on artifact synthesis. Examples include
  --Maintenance/Evolution
  --Reengineering
  --Reuse
  --Configuration Management

Maintenance/Migration/Evolution

• Meeting the need for software products to change after they have been delivered
• Maintenance: older term, now falling out of favor and use
  --Connotes “fixing” things--eg. errors
  --Also used to cover adding features, responding to changes in requirements, usage contexts
• Most “maintenance” effort is not devoted to fixing errors:
  --More usual motivations are: changes in requirements, altered usage contexts, addition of new features
  --New software systems change the world around them: change the requirements that spawned them
• Migration and Evolution are more descriptive--becoming more popular designations of this activity
  --Post Deployment Software Support (PDSS) is military term

Need to Change is Constant

• Software is part of larger systems that are part of real world
• The world changes ==> systems, and software must too
• Systems and their software change the way the world works
  --Changes requirements for the systems and software
• Inescapable conclusion: All software that gets used will require change
  --Change is the rule, not the exception
  --Changing software used to be considered an indication of trouble, weakness, problems
  --Changes were done hastily, stealthily, sloppily
  --Change must be planned for/designed in (recall: likely directions of change must be part of requirements spec)
  --Software migration/evolution should be orderly, visible process

What is Involved in Doing This?

• Recall: > 2/3 of total lifetime cost of a software system is attributable to “maintenance”
• Partly because software is continually evolving--so this process continues indefinitely
• Mostly because this process is very hard
• Maintaining interartifact consistency is what makes this so hard
• Altering separate artifacts is straightforward: Keeping them consistent is not
• Changing one artifact (eg. code or requirements) compels making consistent changes to all related artifacts
  --And transitively
• Helps to know the relations tying the product together
A Simple Example

Add a display to aircraft cockpit instrumentation

- Changing code is clearly required:
  - Add new code to drive new display
- But testplan must be changed too:
  - New testcases to test new display
  - New set of required test results to match
  - Rerun (some?) old testcases too; to assure that new code has not damaged old functionality
- Changes to requirements too:
  - Requirements specify what system is supposed to do
  - System must not do more than what requirements specify
- Changes to designs/models, too:
  - Capture/express structure of new code
  - Identify new modules
  - Provide basis for deciding future code changes
- There are more changes, and more types of changes

More Complex Situations

- Multiple code changes:
  -- Which to do first?
  -- Do they conflict/interfere? (at the code, design, requirements level)
- Multiple requirements changes, design changes
- Changes of different sorts: some code, some design, ...
- Most software products are being changed continuously, in various ways.
- How to keep the product operating during change?
- How to phase and stage the work?
- Bug tracking systems, configuration management systems --to be addressed later in the course

Configuration Management (CM)
Configuration Control (CC)
Version/Revision Control

How to control change/evolution of a software product? --and especially change in a software product line

CM Processes

- CM has different requirements in different organizations and under different circumstances:
  -- Need to track MR's accurately
  -- Need to dispose of MR's within a given time
  -- Need to identify conflicting revisions
  -- Need to restrict numbers of different types of versions
- Leads to different architectures & designs of CM processes
- Some common architectural features:
  -- Configuration manager: real person; makes final decision about which versions, variants, revisions, become permanent; about what "consistency" means in a baseline, etc.
  -- Configuration control board: considers/evaluates conflicting revisions. Recommends ways to assure continuing integrity of configurations
### Process Maintenance (ie. “Process Improvement”)
- Just as context of the product changes during its lifetime, so does the context of the process
- Development process may run for years.
- Development context changes during that time
  - Product template
  - Parallelizability (eg. changes in resources, like people)
  - Execution time changes (ie. deadline slippage/acceleration)
- Product quality requirements changes \(\Rightarrow\) more testing
- Evolving product may experience difficulties requiring process change
- Process problems may be recognized during execution
- Suggests need to improve them

### The Capability Maturity Model (CMM)
- Measures the ability of an organization to control software development with processes
- Measured in levels (1-5)
- The higher levels certify ability to change processes
  - ISO 9000, TickIT, etc. are other approaches

### Reverse Engineering/Reengineering
- Reverse Engineering: The process of recovering other types of artifacts of a software product from (only) the code
- Reengineering: Process of creating an improved version of a system: usually by starting with the products of reverse engineering
- Maintenance/evolution/migration/reengineering is not possible without these non-code artifacts
- Usually all or most of them are not explicitly available
- Often some (eg. requirements, some models, some testcases) are in heads of project personnel
- Sometimes most or all are not available at all
  - “Legacy” systems: Where code—and essentially nothing else is “inherited” by a would-be maintainer
- Reverse Engineering recovers sufficient non-code artifacts (and interconnections) to enable at least some migration

### Motivations for Reengineering
- Trillions of lines of code are in use in the world
  - estimated replacement value: \(>\$10\) TRILLION
    - a formidable asset (worth more than all the oil?)
- Most of it is written in Cobol
  - better to rewrite at least some in a newer language
- Much of it runs on antiquated “mainframes”
  - would be cheaper to run it on newer, faster hardware/software platforms
- Most makes little or no use of modern user-interface idioms
- Most of it is not accompanied by design, requirements or testcases artifacts, suitably related to the code
  - Hard to tell what it can really do, what it should do
  - Such systems “are what they are”
  - Changing them is not even contemplated
  - They become increasingly large obstacles
- Most new software being built today is “legacy” software

### Desirable Reengineered System Design Artifacts
- Set of abstractions/modules
- Sets of methods/accessing primitives for modules
- Object hierarchies
- Booch-style “uses” relation among modules
- Rationales for selections of design decisions (that are presumably hidden)
- Explanations of roles and functions of program variables
- Explanations of purposes of all loops (eg. loop invariants)
- Indications of which design (and code) elements are there to address which requirements (and vice versa)

### What Can Be Expected from Automated Reverse Engineering
- Control structural representations:
  - Flow graphs of code
  - Call graphs among code procedures
  - Inter-task communication graphs
  - Data flow diagrams
- Data representations:
  - Symbol Tables
  - Definition/reference cross reference tables
  - Variable dependency relations
  - Variable visibility and access relations
- A far cry from what is needed
- Some of this can be effective basis for inferring more important design information, eg. module specifications
Reuse/Software Composition

- Developing software with substantial use of large component parts that were previously developed (usually by others)
- Intuitive motivation: Why not manufacture software the way other large, important, expensive products are made?
- Compose large products out of large subassemblies, which are composed out of smaller subassemblies, etc...
- Sometimes called software composition
- Sometimes called megaprogramming—programming using a language whose primitives are themselves significant items of software

Advantages of Reuse

- Reusing project saves (considerable) cost of redeveloping sizeable software components
- Reused component is probably of higher quality because of testing, analysis, evolution in prior context(s)
- Product software has internal structure that has important similarities to other software—should facilitate maintenance
- Reusable components are a relatively more tangible organizational asset
- Reusable components may themselves suggest new products
- Reusable components may encourage organization to specialize in areas addressed by these components

Obstacles to Reuse

- Components must be built for reusability
  -- Not every hunk of code should be expected to be reusable
- Software products are intricately interconnected webs of artifacts of diverse types
  -- Reuse only one single artifact entails connecting it up to all objects to which it must be related
  -- Suggests that what must be reused is a structure of related artifacts (eg. code, design, testcases, etc.)
- Effectively reusable components cost more (often much) more to build
  -- What incentives do people have to incur this extra cost?
  -- "Why should I pay extra so someone else can save $$$"?
- How to make potentially reusable components accessible?
  -- What kind of database?
  -- How to store them in the database?
  -- How to catalog them so they can be found easily?
  -- How to help imbed them in reusing project’s product?

More Obstacles to Reuse

- Reusable components invariably require at least some modification/adaptation
  -- How to make them adaptable?
  -- How to decide when cost of adaptation exceeds cost of building from scratch?
- Software people prefer to create, not reuse
  -- How to incentivize people?
- How to decide what reusable components to build?
- How to keep an organization working mainly on projects that are likely to be substantial reusers?

A Reuse Success Story

-- In a Japanese Software Factory

- Large mainframe manufacturers built in Japan in 70’s, 80’s
- Actual building designed to support software development
  -- Ergonomic offices
  -- Private cubicles
- Suitable computing support
  -- Adequate computing power
  -- Networking
- Clearly understood development processes
  -- Based on functional decomposition of design/coding
  -- Clear, specified development process artifacts comprising an unmistakable audit trail
- Heavy use of productivity metrics and emphasis on demonstrable improvement
- Emphasis on REUSE

Reuse in the Software Factory

- Developers are assigned tasks (generally coding) by their supervisors
- Task specification includes suggestions about which components in reuse library are expected to be reusable
  -- In some cases reuse probabilities are assigned
  -- Include related artifacts (eg. design, testcases)
- Developer is measured in part by how closely the reuse target has been met
- Adapted versions of reused components are contributed to the reuse library
  -- Accompanied by related artifacts
  -- With explanations of why adaptation was necessary
- Explanations of why components that were expected to be reused were not reused

Some software factory projects have achieved factor-of-ten improvements in “quality” and “productivity”
Why Does This Work?

- Strong incentives to reuse (based on social pressure and productivity measurement)
- Investment in reuse
  --Costs extra to build components for reuse
  --Cost to maintain reuse database
  --Cost to resubmit adaptations to reuse database
- A software reuse process that works
- Focus on narrow software product line
  --One software factory builds only device drivers
  --Another may build operating system components
  --Another may build compilers for similar languages

**Much less successful when focus is not so narrow**

---

**Various Approaches to Doing This**

- Build systems out of "components"
  --Parts that have already been built
- Components are large bodies of code
- But also accompanying
  --Test results
    --Explicit or Implicit
  --Design elements
  --Requirements
  --Etc.

---

**Software Product Line Development**

- A Software Product Line is a family of related software products
- Aim is to achieve advantages that come from
  --Developing
  --Testing
  --Maintaining/evolving
  --Several products having significant similarities
- Advantages to thinking of a product line, instead of a product
  --Possible to reuse components during implementation
  --Possible to benefit from experiences with earlier products
  --Possible to amortize cost of a component across multiple uses
  --Possible to train workers deeply in a (narrow) area
- Reuse has a chance to work
- Products outside the product line will not be built
- Specialization in software manufacturing follows precedents in other manufacturing domains

---

**Different Approaches (Or Just Different Names?)**

- Component-Based Development
- Service Oriented Architectures
- Cloud Computing
- Systems of Systems
- Product-Line Development
  --??
  --There are differences, but there are more similarities

---

**The Double Life Cycle**

- Synergistic development of domain model and product line
- Products in the product line are intricately interconnected webs of software artifacts (as described earlier in course)
- Products in the product line share and reuse significant numbers of significant components
- The domain model is also an intricate web of related software artifacts
- Domain model should include components for key concepts
- Components include code plus related designs, evaluations, etc.
- Some artifacts and interconnections generally absent
  (e.g., executable code, some invocation and uses relations)
- Should be comparable in size, complexity to product webs
- Domain model must be developed and evolved
  --Considerations very much similar to those for products
- Domain, product line development/evolution support each other
The Double Lifecycle

Some Final Thoughts

• Expect that your software will live forever

• Accept that your software will need to change: plan for it

• Know your software’s stakeholders

• Don’t save “testing” until the end: think of it always

• Test what you run and run (only) what you test
Some Final Thoughts

- Expect that your software will live forever
- Accept that your software will need to change: plan for it
- Know your software’s stakeholders
- Don’t save “testing” until the end: think of it always
- Test what you run and run (only) what you test
- Don’t ever write a loop unless you know its invariant
- Modularize and hide key secrets

The software systems you build will change the world
Do it responsibly