Other Key Software Processes:

Maintenance/Migration/Evolution

Reuse, Reverse Engineering, Reengineering

Computer Science 520/620
Spring 2014
Prof. Leon Osterweil

End of the Semester Schedule

• April 22: Assignment #5 Distributed
• April 24: 620 Term Paper abstracts due
• April 29: Take-Home Final Exam Distributed
• May 2: Assignment #5 is due
• May 8: Take-Home Final is due
• May 8: 620 Term Papers due

Student Panel Discussion with UMass CS Alumni and Advocates

Friday, May 2, 2014
3:00 - 4:00 p.m.
Computer Science Building, Rm. 150/151

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
• 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
What is Involved in Doing This?

- Recall: > 2/3 of total lifetime cost of a software system is attributable to "maintenance"
- Maintaining interartifact consistency is what makes this so hard
- Altering separate artifacts is straightforward: Keeping them consistent is not

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
  -- New requirements are basis for new testcases/test results
- 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

Software Configuration

- Intuitive notion: a configuration is an assembly of parts into an integrated whole
- Corresponds nicely to our notions of a software product as an interconnected set of software artifacts
Software objects also have internal structure, they are either
- atomic
- composite

Atomic objects are indivisible

Composite objects consist of
- recordlike structures
- arraylike structures
...of "records" "arrays" atomic objects

A configuration can well be thought of as a composite of
software objects having properties as just described

There are many different (sub?) types of configurations

Relations comprising configurations

- Source objects are immutable:
  - y is-a-revision-of x (ordered binary relation)
    -- Arises, for example, when an editor is applied to x
    -- This relation is: anti-symmetric, irreflexive, transitive
  - y is-a-variant-of x (ordered binary relation)
    -- When x and y are identical under some abstraction (ie. differ only in some details that are invisible).
    -- Actually a ternary relation where abstraction is object too
    -- This relation is: symmetric, transitive, reflexive (ie. an equivalence relation)
  - y is-a-derivation-of x (ordered binary relation)
    -- When y is the output of a tool, applied to x
    -- This relation is transitive

CM Processes

- CM has different requirements in different organizations
  and under different circumstances
  -- Need to track modification requests (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 Need Maintenance Too

- 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
  -- Parallelizability (eg. changes in resources, like people)
  -- Execution time changes (ie. deadline slippage/acceleration)
  -- Product quality requirements changes => 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
- Often only code is available to a would-be maintainer
- The rest is in heads of people (some of them are dead)

Called “legacy” systems

- Reverse Engineering recovers sufficient non-code artifacts (and interconnections) to enable at least some migration

Motivations for Reengineering

- Estimated Trillions of lines of code are in use worldwide
  – a formidable asset in need of protection/insurance
- Much of it is written in obsolete languages (e.g. 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

What Can Be Reverse Engineered?

- Requirements:
  --Inferred from known/observed system characteristics
  --Inferred from manuals
  --May require experimentation
  --System often used in only limited contexts, behavior beyond this is unknown
- Testcases/test results:
  --Closely related to the above
  --Generally overlooked
  --Important to reverse engineer testcases: basis for knowing whether or not reengineered system has execution characteristics that are acceptably close to original system
- Design information:
  --Most usual type of non-code artifact sought
  --Most clearly useful basis for reengineering
  --Basis for rehosting on new platform
  --Basis for streamlining involved code
  --Basis for sealing up implementation decisions that need revitualization

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 than
  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 imbue them in reusing project’s product?

Good example: Japanese Software Factories
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?

Domain Engineering

- Discipline of developing explicit structure of an area/domain
  --Usually called a "Domain Model"
- Akin to knowledge representation ??
- Express the key concepts in an area and their relations to each other
- Identify the key concepts: requires expertise in the domain
  --Maybe capture this from outside experts
  --Maybe use expertise within a development organization
- Decomposition of key concepts into contributory notions
  --Not likely to be possible with a simple tree
  --Not unlike an object hierarchy in important ways
- Capture relations among the key concepts
  --Uses, includes, is-produced-from, ...

Uses of a Domain Model

- Use domain model to infer "useful" abstractions
- Use these abstractions as specifications for coding reusable modules
- Domain model functions as conceptual design/requirements for these modules—in some ways
- For abstractions to be useful more is needed
- Need sense of direction(s) in which an area is moving too
  --New, emerging concepts need to be related, tracked
  --The impact of new implementation technologies (eg. superior algorithms, hardware) must be considered
  --New implementations have to be developed, evaluated
- Goal: Coherent, coordinated set of reusable components
  --Not just code, but design, evaluation, artifacts too
  --Also a sense of when to use which components
  --Quantifiable costs and benefits too

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

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
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
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

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
- Coding languages come and go: don’t get too attached to any of them

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
- Coding languages come and go: don’t get too attached to any of them

The software systems you build will change the world

Do it responsibly

The software systems you build will change the world

Do it responsibly