Software Engineering
CS5704: Class 14 - 4/27/01

Instructor: Shawn A. Bohner
Voice: (703) 538-8374
Email: bohner@nvc.cs.vt.edu
Teaching Assistant: Sapna Georges
Voice: (703) 538-8381
Email: sgeorge@vt.edu

Agenda

▲ Hand in Project! Yes!!! It’s done…
▲ Class 13 Review
  ● Software Maintenance/Evolution and Reengineering
  ● Chapter 26 – Cleanroom Software Engineering

▲ Chapter 27 – Component-Based Software Engineering
▲ Break
▲ Final-Exam Review
Next week is the final class and final examination... we can see the finish line... It is time!

## Maintenance versus Development

- **Must work within the constraints of an ???**
  - Large part of effort is in understanding the change in the context of existing system artifacts
- **??? needed for estimates**
- **Requires an ??? rather than construction mindset**
- **Typically changes are smaller scale than original development**
- **Manage change with multiple ??? rather than functional builds**
- **The closer development gets to delivery the more it looks like maintenance**
The Software Maintenance Challenge

▲ Seemingly minor changes often turn out to be more extensive than expected
▲ Must know the consequences of software changes...risks include:
  • ??? changes (maybe discovered by user...)
  • ??? implemented changes (patches and spaghetti)
  • Effort, resource, and estimate errors (due to low visibility)
  • Difficulty augmenting software design
  • Reduced ??? and useful life of the software

Software Reengineering Techniques

▲ Redocumentation
▲ Restructuring
▲ Reverse Engineering
▲ Conversion
▲ Software Salvaging
▲ Migration

Why is it so hard to capture information/artifacts above the detailed design level?
Cleanroom Software Engineering

▲ Definition: Development and evolution of software products that:
- Meet ??? needs
- Are correct with respect to ???
- And have scientific certification of quality

▲ Benefit: Creates near zero defect software

▲ Cost’s are high
- Specially ??? staff and team configurations
- Longer requirements and design specification

When is Cleanroom SE most effective?
The Cleanroom Process Model

What are the key components of Cleanroom Process? What impact does this have on the development staff?

Refinement and Verification

Procedural constructs in hierarchy must be ???
Advantages of Design Verification

- Reduces verification to a ... process
- Cleanroom teams can verify every line of ... and code
- Results in a near zero defect level
- Design verification scales up to ... situations
- Produces better code than unit testing

Cleanroom Testing

- Statistical “???” Testing
  - Tests the actual usage of the program
- Determine a “Usage Probability Distribution”
  - Analyze the specification to identify a set of stimuli
  - ??? cause software to change behavior
  - Create usage scenarios
  - Assign probability of ??? to each stimuli
  - Test cases are generated for each stimuli according to the ??? probability distribution
Chapter 27: Component-Based Software Engineering

The purpose of this section is to describe component-based software engineering (CBSE) as a process emphasizing design and construction of systems out of highly reusable software components.

- CBSE has two parallel engineering activities – domain engineering and component-based software development.

Objectives

- Explore CBSE principles and requirements
- Examine CBSE process

The Key Questions

- When faced with the possibility of reuse, the software team asks:
  - Are commercial off-the-shelf (COTS) components available to implement the requirement?
  - Are internally-developed reusable components available to implement the requirement?
  - Are the interfaces for available components compatible within the architecture of the system to be built?

- At the same time, they are faced with the following impediments to reuse ...
Impediments to Reuse

- Few companies and organizations have anything that even slightly resembles a comprehensive software reusability plan
- Although an increasing number of software vendors currently sell tools or components that provide direct assistance for software reuse, the majority of software developers do not use them
- Relatively little training is available to help software staff understand and apply reuse
- Many software practitioners continue to believe that reuse is “more trouble than it’s worth”
- Many companies continue to encourage software methodologies which do not facilitate reuse
- Few companies provide an incentives to produce reusable program components

The CBSE Process
Domain Engineering

1. Define the domain to be investigated
2. Categorize the items extracted from the domain
3. Collect a representative sample of applications in the domain
4. Analyze each application in the sample
5. Develop an analysis model for the objects

Identifying Reusable Components

▲ Is component functionality required on future implementations?
▲ How common is the component's function within the domain?
▲ Is there duplication of the component's function within the domain?
▲ Is the component hardware-dependent?
▲ Does the hardware remain unchanged between implementations?
▲ Can the hardware specifics be removed to another component?
▲ Is the design optimized enough for the next implementation?
▲ Can we parameterize a non-reusable component so that it becomes reusable?
▲ Is the component reusable in many implementations with only minor changes?
▲ Is reuse through modification feasible?
▲ Can a non-reusable component be decomposed to yield reusable components?
▲ How valid is component decomposition for reuse?
Structural Modeling

▲ Every application has structural patterns that have the potential for reuse
▲ Structure point (SP) is a construct of a structure
  • An abstraction with a limited number of instances (in OO context, the size of the class hierarchy should be small)
  • Rules that govern the use of the SP should be easily understood
  • Interface to the SP should be relatively simple
  • Should implement information hiding by hiding all complexity contained within the structure point itself -- reduces the perceived complexity of the overall system

Component-Based Development

▲ A library of components must be available
▲ Components should have a consistent structure
  • Example standards (e.g., OMG/CORBA, COM, EJB)
▲ Key CBSE Activities
  • Component qualification
  • Component adaptation
  • Component composition
  • Component update
Qualification

Before a component can be used, consider:
- Application Programming Interface (API)
- Development and integration tools required by the component
- Run-time requirements including resource usage (e.g., memory or storage), timing or speed, and network protocol
- Service requirements including operating system interfaces and support from other components
- Security features including access controls and authentication protocol
- Embedded design assumptions including the use of specific numerical or non-numerical algorithms
- Exception handling

Adaptation

Implication of “easy integration” is:

1. That consistent methods of resource management have been implemented for all components in the library

2. That common activities such as data management exist for all components

3. That interfaces within the architecture and with the external environment have been implemented in a consistent manner
Composition

▲ An infrastructure must be established to bind components together

▲ Architectural ingredients for composition include:
  • Data exchange model
  • Automation
  • Structured storage
  • Underlying object model

Classification

▲ Enumerated classification—components are described by defining a hierarchical structure in which classes and varying levels of subclasses of software components are defined

▲ Faceted classification—a domain area is analyzed and a set of basic descriptive features are identified

▲ Attribute-value classification—a set of attributes are defined for all components in a domain area
Promise of “Architected” Systems

Some capabilities are beyond those offered on COTS. Postponing them could result in a much lower cost than custom code.

Development resources are expended on small amount of software built to exploit capabilities of commercial products.

Substantial portion of system requirements are served by Commercial products.

Acquiring Cots-Based Systems: A Generic Model

Requirements
- Market Survey
- Ref Model
- Standards Profiles

Constraints
- System Specification
- Design
- Implement
- Integrate & Test

Deliver

Prototype
Integrate COTS Package Measures

COCOTS: Development Model

1. COTS Assessment
2. COTS Tailoring
3. Glue Code Development
4. System Effort due to COTS Volatility

- LCO (requirements review)
- LCA (preliminary design review)
- IOC (system delivery)

New System Development Not Involving COTS Components

LCO – Lifecycle Objectives
LCA – Lifecycle Architecture
IOC – Initial Operational Capability

COCOMO II Effort Estimate

Source: Barry Boehm

COTS Package Measurement Framework

Key COTS Characteristics
- Functionality
- Performance
- Size
- Scalability
- Product Maturity
- Vendor Characteristics
- # of Products Used
- Standards Adherence
- Security

Adaptability
Application Development
Customization
Integration
Configuration
Personalization
Coexistent
Cooperating
Interlocked

Management Tools
Development Tools
Applications
Common Services
Low Level Utilities
Operating System
Hardware

Source: Barry Boehm
Lightweight Development Methodologies

Lean Development by Robert Charette

Extreme Programming (or XP)
by Kent Beck, Ward Cunningham, and Ron Jeffries

Crystal by Alistair Cockburn

SCRUM by a Rutgers University Team

Rapid Development by Steve McConnell

Common Factors
- Risk/Value Focused
- Schedule Driven
- Team Leveraged

Extreme Programming
- Frequent Releases
  - Daily builds and two-week releases to control scope and complexity
- Merciless Refactoring
  - Iterative discovery process incorporating learning principles
- Continuous Integration / Relentless Testing
  - Unit Test and Feature Driven
- Planning Game and Story Cards
- System Metaphor
- Success is Spelled “T E A M”
Powerful XP Team Principles

▲ Right Size Team
- Size teams for efficiency, stability, and effective communications

▲ Self-Selecting Team
- Developers "apply" for positions
- Self-motivation results in higher performance

▲ Pair Programming
- Enhances collaboration, knowledge transfer, and creativity. Two developers at the same machine.
- Most effective during design

▲ Internal Control for Knowledge Retention
- Ensures internal developers are being used
- Maintains business control of projects

The Planning Game

- **Business Creates Stories** — Describe desired function
- **Development Assigns Costs** — Time estimates should be between 1 and 3 weeks
- **Business Creates Commitment Set** — Can be story- or data-driven
- All processes should be collaborative and use iteration
The Story Card

- **Story Title** — names for each story (a "story" is a basic feature or scenario)
- **Estimate** — ideal number of weeks to produce this story (if greater than 3, must factor; if less than 1, should combine)
- **Story Description** — text description of the story
- **Business Value** — rating of importance of this story (allows sorting to rank)
- **Back of Card** — used for notes and dependencies

A Simple but Effective System Enables Collaboration, Communications, and Traceability

Continuous Integration/Relentless Testing

Each phase utilizes iteration and communication — speeding creation and delivery of correct business function
Feature Driven

▲ Creating an Overall Model: Initial requirements are captured and documented in UML (use case, class, sequence) and as stories. Best carried out in a storyboard fashion.

▲ Creating a Features List: A feature is a client-valued function that can be implemented in two weeks. Process is iterative -- carefully triaging requests and coordinating efforts of infrastructure/application groups.

▲ Planning by Feature: Milestones now sequenced and placed into a development plan. Feature sets have chief programmers assigned to coordinate the design and development of the set.

▲ Designing by Feature: Feature teams create the models to describe each feature using the "write unit test first" pattern. Interaction and design review ensure uniformity of design and use of common design patterns.

▲ Building by Feature: Often leads to discovery of design issues. When feature is completed, the code is reviewed and on acceptance by the chief programmer it is promoted into the daily build.

Homework Assignment for 5/4/01

▲ Study hard for a challenging Final Exam
  • Review Chapters 1-27 (only assigned reading) of SEPA text and SEPA website
  • Class slides and notes with homework answers
  • Assigned papers
  • Closed book, multiple choice, True/False, essay,

▲ Have a great week!