Software Engineering
CS5704: Class 13 - 4/20/01

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

▲ Class 12 Review
  ● Chapter 19 – Technical Metrics for Software
  ● Chapter 24 – Technical Metrics for Object-Oriented Systems

▲ Software Maintenance/Evolution and Reengineering
▲ Break
▲ Chapter 26 – Cleanroom Software Engineering
▲ Homework Assignments
2 weeks, 2 sessions to go... we’re in the home stretch... Still, so much to do and so little time...

**Architectural Design Metrics**

▲ *Architectural Design Metrics* *(Card and Glass)*
- ?? complexity = \( S(\text{fan-out}) \)
- ?? complexity = \( D(\text{input & output variables, fan-out}) \)
- ?? complexity = \( C(\text{structural + data complexity}) \)

▲ *Information Flow Metric*: *(Henry and Kafura)*
- ?? complexity as a function of fan-in and fan-out

▲ *Morphology Metrics* *(Fenton)*:
  - a function of the number of modules and the number of interfaces between modules
Example SW Maintenance GQM

<table>
<thead>
<tr>
<th>Goal</th>
<th>Question</th>
<th>Metric(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximize customer</td>
<td>How many problems are affecting the customer?</td>
<td>Discrepancy Report (DR) &amp; Service Request (SR)</td>
</tr>
<tr>
<td>satisfaction</td>
<td>How long does it take to fix a problem?</td>
<td>Open Duration</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Software Reliability</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Break/Fix Ratio</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DR/SR Closure</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DR/SR Open Duration</td>
</tr>
</tbody>
</table>

What is Software Complexity?

▲ **Complexity** – degree to which a system or component is difficult to **???, ???, or ???.**

▲ **Why is it important?**
  - Provides a means of ***the complexity*** of modules produced by the same process, in the same language, etc.
  - It provides a systematic way of allocating ***resources*** to the modules which show the greatest need
  - Furnishes information on ***complexity*** in the **???** lies: data structures, flow control, interfaces...
Software Reliability

▲ Software reliability is the probability that a software system will perform its ??? without ??? for a specified ??? in a ???

Class-Oriented Metrics

▲ Weighted methods per class (WMC) -- Sum of complexities of all methods of a class
  • More methods means ???
▲ Depth of the inheritance tree (DIT) -- The maximum length from the node to the root of the tree
  • Deeper trees constitute ??? design complexity
▲ # of children (NOC) -- # of immediate subclasses
  • More children => the greater the ???, chance of improper ??? of parent class, & ??? testing of the methods
▲ Coupling between object classes (CBO) -- # of classes to which a class is coupled
  • Higher coupling increases the sensitivity to changes in other parts of the design, and reducing ???

Source: Chidamber and Kemerer

Source: George Stark (IBM)
Class-Oriented Metrics

▲ Class size (CS): overall size of class
  • # of operations
  • # of attributes
▲ NOO - # of operations ??? by a subclass
  • Large NOO shows abstraction violation of superclass
▲ NOA - # of operations ??? by a subclass
  • Increasing NOA may indicate drift in abstraction
▲ Specialization index (SI): degree of specialization
  • (NOO X DIT Level) / Total # of methods

Source: Lorenz and Kidd [LOR94]

The MOOD metrics set: ratios for
- inheritance (??? and ???)
- polymorphism (PF)
- message-passing (???)
- encapsulation (??? and ???)

Who Suggested these factors for OO metrics?
What does MOOSE stand for?

▲ Method Inheritance Factor (MIF)
▲ Polymorphism Factor (PF)
▲ Coupling Factor (CF)
▲ Attribute Inheritance Factor (AIF)
▲ Method Hiding Factor (MHF)
▲ Attribute Hiding Factor (AHF)

Operation-Oriented Metrics

▲ Average operation size ($O_{avg}$)
  - LOC is a weak indicator
  - # of messages sent is an alternative
▲ Operation complexity (OC)
  - Target ??? OC
▲ Average number of parameters per operation ($N_{avg}$)
  - Target ??? $N_{avg}$

Source: Lorenz and Kidd [LOR94]

OO Project Metrics

▲ # of scenario scripts (NSS) -- # of scenario scripts or use-cases
  - Indicator of ???
▲ # of key classes (NKC)
  - Key classes focus on ???
  - High NKC indicates lots of ???
▲ # of subsystems (NSUB)
  - Like NKC, high NSUB indicates more work
  - Good for effort, resource allocation, and schedule

Source: Lorenz and Kidd [LOR94]
Software Maintenance/Evolution and Reengineering

▲ The purpose of this section is to reexamine software maintenance as a means of evolving software and reengineering’s role
  • Combines Software Maintenance with Chapter 30 on Reengineering
  • Introduce Software Change Impact Analysis through the Year 2000 Problem

▲ Objectives
  • Explore software maintenance/evolution principles
  • Examine impact analysis in the software change process

What I Mean When I Say...

▲ Software – part of a computer system that is intended to change (programs)

▲ Software Maintenance/Evolution – means change to existing software systems
  • Software Change (corrective, adaptive, perfective)
  • Support for Change (process, organization, CM,...)

▲ Impact Analysis – assessment of potential consequences resulting from the planned changes to software systems
Maintenance versus Development

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

Software Maintenance Mindset Calibration

▲ Traditional Mindset: Maintenance is a problem!
▲ Software Maintenance activities consume 50-70% of many software budgets
  ● Reflects mature organization concerned with software assets
  ● ~ 20% of maintenance devoted to fixing errors (Corrective)
  ● Rest for making improvements (Adaptive & Perfective)
▲ Better Mindset: Maintenance Sustains and Evolves Software Assets
Once the Software is in Operation...

Software change is like rebuilding a boat on the open sea... 
...the work has to proceed one board at a time on existing boards.

If you change too much at once.. 
...the boat sinks!

If you change without visibility... 
...the boat increasingly leaks!

The “Hydra Effect” in Corrective Maintenance

The problem that got fixed . . .

. . . versus the REAL problem
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:
  - Incomplete changes (maybe discovered by user...)
  - Poorly implemented changes (patches and spaghetti)
  - Effort, resource, and estimate errors (due to low visibility)
  - Difficulty augmenting software design
  - Reduced maintainability and useful life of the software

Software Change Impact Analysis

*Impact analysis identifies potential effects of proposed software changes*

What is the impact of a software change?

- Data Flow
- Control Flow
- Program Slicing
- Dependency Analysis
- Code & Test
- Requirements & Design
- Software Documentation
- Requirements Traceability
Software Change Impact Analysis Process

1. Examine Software and Change Specification
2. Trace Potential Impacts
3. Perform Software Change

- Candidate Impact Set (CIS)
- False Positive Impact Set (FPIS)
- Discovered Impact Set (DIS)

AIS = CIS + DIS - FPIS

Software Change Process

▲ Top-Level Activities
- Manage
- Understand and Determine Impact
- Specify and Design
- Implement
- Test

▲ Ripple-Effects

▲ Impacts
- Requirements
- Design
- Program
- Testing
A1: Manage Software Change

Key Activities:
- Establish Quality Goals
- Determine Risks
- Produce Estimates
- Track Progress
- Assign Resources
- Schedule Releases

A2: Understand Software Change and Determine Impact

Key Activities:
- Review SW Documentation
- Identify SW Impacts
- Clarify SW Change Request
- Record Impacts
- Determine Stability
A22: Identify Software Change Impacts

Key Activities:
- Examine Traceability
- Classify and Explore Change
- Req’ts Impacts
- Design Impacts
- Program Impacts
- Test Impacts

- Determine patterns for date names -> SIS
- Evaluate with structure (dependencies) -> CIS

A3: Specify and Design Software Change

Key Activities:
- Analyze Change Requirements
- Examine SW Arch. Changes
- Derive Related Change Requirements
- Design Program Changes

- Iterative process
- Start making change and capturing AIS
A4: Implement Software Change

Key Activities:
- Determine Modules to be Changed
- Identify Statement level Changes
- Apply Change
- Unit test

Discovered ripple-effects -> AIS

For Y2K, some tools can be used here for date translation

A5: (Re)Test Affected Software

Key Activities:
- Generate test cases
- Update test suite
- Perform integration tests
- Conduct system testing
- Conduct acceptance testing

Multiple integrated systems
Regression test suite
Three Areas of Reengineering Support

- Business Needs
- Information Needs
- Information Processing Needs

Technology Changes
- Industry Changes

Data Reengineering

Software Reengineering

Reengineered Information Systems

Reengineering Staff, Methods, Tools

Business Process Reengineering

Business Process Engineering (BPE)

Engineering Objectives
- Understand Existing Processes
- Determine Process(es) to be (Re)Designed
- Design Process
- Pilot Newly Selected Process(es)
- Implement New Process(es)

Current Business Process

New Business Process

BPE Tools
(Re)Engineering Business and Its Processes

Develop Business Vision and Process Objectives

Determine Processes to be Redesigned

Understand Existing Processes

Determine Change Levers

Design Process Alternatives

Implement New Process

Pilot Selected Process

Improve New Process

Using Information Technology to support Business Processes

Order Management Process

Functions

- Sales
- Manufacturing
- Logistics
- Finance

Resources

(People and Information Technology)
Relating Business Reengineering and Information Technology (Software)

Using IT to support Business Processes

(Re)Designing Business Processes using IT

Using Information Technology to support Business Processes

Information Technology as an Business Process Enabler

**Automating**: eliminate human labor

**Organizing**: capture information for understanding

**Sequencing**: alter process sequence, eliminate intermediates, and enable parallelism

**Tracking**: monitor status and objects

**Analyzing**: improve information analysis and decision making

**Networking**: connect processes across distances

**Integrating**: coordinate between processes and tasks

**Intellectual**: capture and distribute intellectual assets
The Five “W” Sisters and “Howie”

▲ Who => Customer, Organization, Staff, ...stakeholders
  - Human resource strategy
  - Planner, modeler/designer, Domain
▲ What => Information and Products
  - Knowledge (application domain, system architecture, system engineering)
  - Data, Metadata, Information to support business
▲ Where => Location, Communications, Networks
▲ When => Schedules
▲ Why => Motivation, goals/objectives, plans...
▲ How => Process, Function, System

Software Reengineering Techniques

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

Redocumentation

Restructuring

Reverse Engineering

Conversion

Software Salvaging

Migration

Reverse Engineer  Forward Engineer
Software Reengineering Techniques

- Redocumentation
- Reverse Engineering
- Restructuring
- Conversion

Software Salvaging and Migration

- Software Salvaging (for reuse)
- Software Migration
Software Salvaging for Reuse

- Identify existing AMSS/MMSS function and data to support business processes
- Analyze AMSS/MMSS applications portfolio
- Reverse engineer high-leverage components
  - Requirements (high value, low potential)
  - Architecture (high value, low potential)
  - Design (medium value, medium potential)
  - Code (medium/low value, high potential)
- Identify and prepare reusable parts from captured artifacts
- Integrate Reusable parts with Newly developed parts to construct new system
  - Effort moved from development to integration
  - Savings from reuse rather than redevelopment
Chap. 26: Cleanroom Software Engineering

▲ The purpose of this chapter is to introduce a radically different paradigm for software work called “Cleanroom Software Engineering.”
  - Emphasizes a special specification approach, formal design, correctness verification, “statistical” testing, and certification as the set of salient activities for software engineering.

▲ Objectives
  - Examine Cleanroom principals and determine where applied
  - Outline key Cleanroom techniques for effective software engineering

Cleanroom Software Engineering

▲ Definition: Development and evolution of software products that:
  - Meet business needs
  - Are correct with respect to specifications
  - And have scientific certification of quality
▲ Pioneers: Harlan Mills, Richard Linger, M. Dyer, ...
▲ Benefit: Creates near zero defect software
▲ Cost’s are high
  - Specially trained staff and team configurations
  - Longer requirements and design specification
Consequences of Cleanroom Approach

▲ Development
- Rigorous, theory-based specification, design, and correctness verification is required to produce zero-defect software

▲ Testing
- Statistical testing is required for valid certification of product quality

▲ Management
- Incremental development is needed for continual integration and quality feedback from certification to development
- Independent development and certification is required for accountability and valid certification

The Cleanroom Process Model
The Cleanroom Strategy

Increment Planning—adopts the incremental strategy

Requirements Gathering—defines a description of customer level requirements (for each increment)

Box Structure Specification—describes the functional specification

Formal Design—specifications (called “black boxes”) are iteratively refined (with an increment) to become analogous to architectural and procedural designs (called “state boxes” and “clear boxes,” respectively).

Correctness Verification—verification begins with the highest level box structure (specification) and moves toward design detail and code using a set of “correctness questions.” If these do not demonstrate that the specification is correct, more formal (mathematical) methods for verification are used.

The Cleanroom Strategy (continued)

Code Generation, Inspection and Verification—the box structure specifications, represented in a specialized language, are transmitted into the appropriate programming language.

Statistical Test Planning—a suite of test cases that exercise of “probability distribution” of usage are planned and designed

Statistical Usage Testing—execute a series of tests derived from a statistical sample (the probability distribution noted above) of all possible program executions by all users from a targeted population

Certification—once verification, inspection and usage testing have been completed (and all errors are corrected) the increment is certified as ready for integration.
**Box Structure Specification**

Black Box—specifies behavior for system or part

State-Box—encapsulates state data and operations

Clear-Box—defines provable procedural specification

**Refinement and Verification**

Black Box

\[ f : S^* \rightarrow R \]

Refinement

Verification

State Box

Clear Box
Black-Box Structure

Customer view, no internals. Specifies behavior for system or part – an abstraction that maps stimuli to response

S)timuli — one or more inputs that transform (history)
R)esponse — an output as a function of the stimuli

Like OO systems, data and function encapsulated, and inheritance or usage hierarchy

State-Box Structure

Intermediate view of transition function. Encapsulates state data and services (operations) much like objects. State box represents stimulus history

sT)ate — one or more system states that change
g) — a subfunction of black-box that is tied to a state

State-Box introduces behavior modeling
Clear-Box Structure

Aligns closely with procedural design and structured programming – programming constructs that implement g

c)onstructs — one of constructs that transform S to R
g) — a subfunction of black-box that is tied to a state

Procedural constructs in hierarchy must be provable...

Design Refinement & Verification

If a function f is expanded into a sequence g and h, the correctness condition for all input to f is:

- Does g followed by h do f?

When a function f is refined into a conditional (if-then-else), the correctness condition for all input to f is:

- Whenever condition <c> is true does g do f and, whenever <c> is false, does h do f?

When function f is refined as a loop, the correctness conditions for all input to f is:

- Is termination guaranteed?

- Whenever <c> is true does g followed by f do f, and whenever <c> is false, does skipping the loop still do f?
Advantages of Design Verification

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

Cleanroom Testing

- Statistical Use Testing
  - Tests the actual usage of the program
- Determine a “Usage Probability Distribution”
  - Analyze the specification to identify a set of stimuli
  - Stimuli cause software to change behavior
  - Create usage scenarios
  - Assign probability of use to each stimuli
  - Test cases are generated for each stimuli according to the usage probability distribution
Certification

1. Usage scenarios must be created
2. A usage profile is specified
3. Test cases are generated from the profile
4. Tests are executed and failure data are recorded and analyzed
5. Reliability is computed and certified

Cleanroom SE Certification Models

Sampling Model: Software testing executes $m$ random test cases and is certified if less than a specified numbers of failures occur.

The value of $m$ is derived mathematically to ensure that required reliability is achieved.

Component Model: A system composed of $n$ components is to be certified.

The component model enables the analyst to determine the probability that component $i$ will fail prior to completion.

Certification Model: The overall reliability of the system is projected and certified.
Homework Assignment for 4/27/01

▲ Read Pressman Chapters
  • Chapter 27 – Component-Based Software Engineering

▲ Deadline Extended on “Dog E-Dating System” Project
  • Final Project Package Due April 27th, 2001 – Next week!

▲ Study hard for Final on May 4th.
▲ Have a great week!