Software Engineering

CSC440/640
Prof. Schweitzer
Week 8

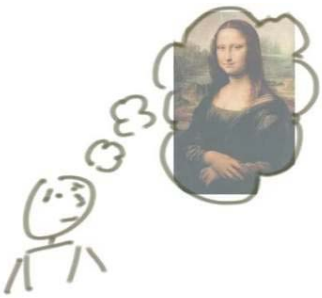
Classroom Discussion

- Off-topic Discussions
  - Talk to me during office hours
  - Email me
  - Would people participate in a message board for the class if it was setup?
- GitHub
  - Follow Me If You'd Like
  - I don't post there as often as you'd probably hope
  - Though I'm planning to put more up there soon-ish

Stepwise Refinement

- A basic principle underlying many software engineering techniques
  - "Postpone decisions as to details as late as possible to be able to concentrate on the important issues"
- Miller's law (1956)
  - A human being can concentrate on 7 ± 2 items at a time

Incremental Development



- Builds a bit at a time
- Incrementing often calls for a fully formed idea
- Changing while incrementing slows us down

1 

2 

3 

Iterative Development

- Builds a rough version, then slowly adds detail and additional quality
- Iterating allows you to move from a vague idea to realization
- While iterating we expect change

1 

2 

3 

Appraisal of Stepwise Refinement

- A basic principle used in
  - Every workflow
  - Every representation
- The power of stepwise refinement
  - The software engineer can concentrate on the relevant aspects
- Warning
  - Miller's Law is a fundamental restriction on the mental powers of human beings

Cost–Benefit Analysis

- Compare costs and future benefits
  - Estimate costs
  - Estimate benefits
  - State all assumptions explicitly
- Tangible costs/benefits are easy to measure
- Make assumptions to estimate intangible costs/benefits
  - Improving the assumptions will improve the estimates

Divide-and-Conquer

- Solve a large, hard problem by breaking up into smaller subproblems that hopefully will be easier to solve
- Divide-and-conquer is used in the Unified Process to handle a large, complex system
  - Analysis workflow
    - Partition the software product into analysis packages
  - Design workflow
    - Break up the upcoming implementation workflow into manageable pieces, termed subsystems
- A problem with divide-and-conquer
  - The approach does not tell us *how* to break up a software product into appropriate smaller components

Separation of Concerns

- The process of breaking a software product into components with minimal overlap of functionality
  - Minimizes regression faults
  - Promotes reuse
- Separation of concerns underlies much of software engineering

Separation of Concerns

- Instances include:
  - Modularization with maximum interaction within each module ("high cohesion") (Chapter 7)
  - Modularization with minimum interaction between modules ("low coupling") (Chapter 7)
  - Information hiding (or physical independence)
  - Encapsulation (or conceptual independence)
  - Three-tier architecture (Section 8.5.4)
  - Model-view-controller (MVC) architecture pattern, (Section 8.5.4)

Software Metrics

- To detect problems early, it is essential to measure
- Examples:
  - LOC per month
  - Defects per 1000 lines of code
  - Defects per Story Point

Different Types of Metrics

- Product metrics
  - Examples:
    - Size of product
    - Reliability of product
- Process metrics
  - Example:
    - Efficiency of fault detection during development
- Metrics specific to a given workflow
  - Example:
    - Number of defects detected per hour in specification reviews

The Five Basic Metrics

- Size
  - In lines of code, or better
- Cost
  - In dollars
- Duration
  - In months
- Effort
  - In person months
- Quality
  - Number of faults detected

CASE (Computer-Aided Software Engineering)

- Scope of CASE
  - CASE can support the entire life-cycle
- The computer assists with drudge work
  - It manages all the details

Some Useful Tools

- Data dictionary
  - Computerized list of all data defined within the product
- Consistency checker
- Report generator, screen generator

Scope of CASE

- Programmers need to have:
  - Accurate, up-to-date versions of all project documents
  - Online help information regarding the
    - Operating system
    - Editor
    - Programming language
  - Online programming standards
  - Online manuals
    - Editor manuals
    - Programming manuals

Scope of CASE

- Programmers need to have:
  - E-mail systems
  - Spreadsheets
  - Word processors
  - Structure editors
  - Pretty printers
  - Online interface checkers

Online Interface Checker

- A structure editor must support online interface checking
  - The editor must know the name of every code artifact
- Interface checking is an important part of programming-in-the-large

Online Interface Checker

- Example
  - The user enters the call
- average = dataArray.computeAverage (numberOfValues);
  - The editor immediately responds
- Method computeAverage not known

- The programmer is given two choices
  - Correct the name of the method to computeMean
  - Declare new procedure computeAverage and specify its parameters

- This enables full interface checking

Online Interface Checker

- Example
  - Declaration of `q` is

```
void q (float floatVar, int intVar, String s1,
String s2);
```

  - Call (invocation) is

```
q (intVar, floatVar, s1, s2);
```

  - The online interface checker detects the fault

- Help facility
  - Online information for the parameters of method `q`
  - Better: Editor generates a template for the call
    - The template shows type of each parameter
    - The programmer replaces formal by actual parameters

Online Interface Checker

- Advantages
  - There is no need for different tools with different interfaces
  - Hard-to-detect faults are immediately flagged for correction
    - Wrong number of parameters
    - Parameters of the wrong type
- Essential when software is produced by a team
  - If one programmer changes an interface specification, all components calling that changed artifact must be disabled

Online Interface Checker

- Even when a structure editor incorporates an online interface checker, a problem remains
  - The programmer still has to exit from the editor to invoke the compiler (to generate code)
  - Then, the linker must be called to link the product
  - The programmer must adjust to the JCL, compiler, and linker output
- Solution: Incorporate an operating system front-end into the structure editor

Source Level Debugger

- The programmer works in a high-level language, but must examine
  - Machine-code core dumps
  - Assembler listings
  - Linker listings
  - Similar low-level documentation
- This destroys the advantage of programming in a high-level language
- We need
  - An interactive source level debugger (like dbx)

Programming Workbench

- Structure editor with
  - Online interface checking capabilities
  - Operating system front-end
  - Online documentation
  - Source level debugger
- This constitutes a simple programming environment
- This is by no means new
  - All the above features are supported by FLOW (1980)
  - The technology has been in place for years
- Surprisingly, some programmers still implement code the old-fashioned way

Software Versions

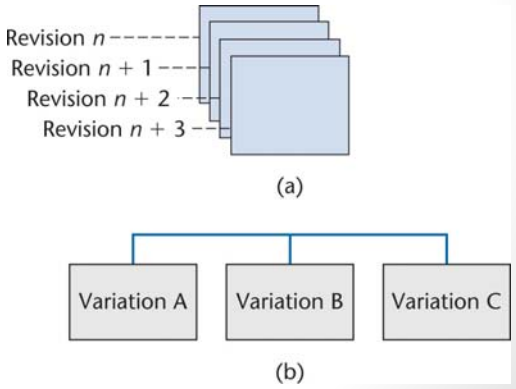
- During maintenance, at all times there are at least two versions of the product:
  - The old version, and
  - The new version
- There are two types of versions: revisions and variations

Revisions

- Revision
  - A version to fix a fault in the artifact
  - We cannot throw away an incorrect version
    - The new version may be no better
    - Some sites may not install the new version
- Perfective and adaptive maintenance also result in revisions

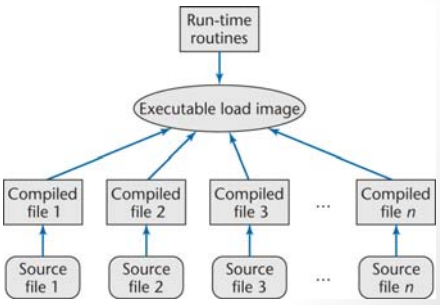
Variations

- A variation is a version for a different operating system–hardware
- Variations are designed to coexist in parallel



Configuration Control

- Every code artifact exists in three forms
  - Source code
  - Compiled code
  - Executable load image
- Configuration
  - A version of each artifact from which a given version of a product is built



Version-Control Tool

- Essential for programming-in-the-many
  - A first step toward configuration management
- A version-control tool must handle
  - Updates
  - Parallel versions

Defect Management

- Allows SQA Group to report bugs and support a fix lifecycle for them
  - Reported
  - Fixed
  - Verified
  - Closed
- Allows structured data analysis on where bugs are found at different levels
  - Module
  - Variation
  - Version
- Track defects back to source code changes

Break

Postdelivery Maintenance

- Postdelivery maintenance
  - Any change to any component of the product (including documentation) after it has passed the acceptance test
- This is a short chapter
  - But the whole book is essentially on postdelivery maintenance
- In this chapter we explain how to ensure that maintainability is not compromised during postdelivery maintenance

Why Postdelivery Maint. Is Necessary

- Corrective maintenance
  - To correct residual faults
    - Analysis, design, implementation, documentation, or any other type of faults
- Perfective maintenance
  - Client requests changes to improve product effectiveness
    - Add additional functionality
    - Make product run faster
    - Improve maintainability

Why Postdelivery Maint. Is Necessary

- Adaptive maintenance
  - Responses to changes in the environment in which the product operates
    - The product is ported to a new compiler, operating system, and/or hardware
    - A change to the tax code
    - 9-digit ZIP codes

What Is Required of Maintenance Programmers?

- At least 67% of the total cost of a product accrues during postdelivery maintenance
- Maintenance is a major income source
- Nevertheless, even today many organizations assign maintenance to
  - Unsupervised beginners, and
  - Less competent programmers
- Postdelivery maintenance is one of the most difficult aspects of software production because
  - Postdelivery maintenance incorporates aspects of all other workflows

What is Required of Maintenance Programmers?

- Suppose a defect report is handed to a maintenance programmer
  - Recall that a “defect” is a generic term for a fault, failure, or error
- What is the cause?
  - Nothing may be wrong
  - The user manual may be wrong, not the code
  - Usually, however, there is a fault in the code

Corrective Maintenance

- What tools does the maintenance programmer have to find the fault?
  - The defect report filed by user
  - The source code
  - And often nothing else
- A maintenance programmer must therefore have superb debugging skills
  - The fault could lie anywhere within the product
  - The original cause of the fault might lie in the by now non-existent specifications or design documents

Corrective Maintenance

- Suppose that the maintenance programmer has located the fault
- Problem:
  - How to fix it without introducing a regression fault
- How to minimize regression faults
  - Consult the detailed documentation for the product as a whole
  - Consult the detailed documentation for each individual module
- What usually happens
  - There is no documentation at all, or
  - The documentation is incomplete, or
  - The documentation is faulty

Corrective Maintenance

- The programmer must deduce from the source code itself all the information needed to avoid introducing a regression fault
- The programmer now changes the source code

The Programmer Now Must

- Test that the modification works correctly
  - Using specially constructed test cases
- Check for regression faults
  - Using stored test data
- Add the specially constructed test cases to the stored test data for future regression testing
- Document all changes

Corrective Maintenance

- Major skills are required for corrective maintenance
  - Superb diagnostic skills
  - Superb testing skills
  - Superb documentation skills

Adaptive and Perfective Maintenance

- The maintenance programmer must go through the workflows, using the existing product as a starting point
  - Requirements
  - Specifications
  - Design
  - Implementation and integration

Adaptive and Perfective Maintenance

- When programs are developed
  - Specifications are produced by analysis experts
  - Designs are produced by design experts
  - Code is produced by programming experts
- But a maintenance programmer must be expert in all three areas, and also in
  - Testing, and
  - Documentation

The Rewards of Maintenance

- Maintenance is a thankless task in every way
  - Maintainers deal with dissatisfied users
  - If the user were happy, the product would not need maintenance
  - The user's problems are often caused by the individuals who developed the product, not the maintainer
  - The code itself may be badly written
  - Postdelivery maintenance is despised by many software developers
  - Unless good maintenance service is provided, the client will take future development business elsewhere
  - Postdelivery maintenance is the most challenging aspect of software production — and the most thankless

The Rewards of Maintenance (contd)

- How can this situation be changed?
- Managers must assign maintenance to their best programmers, and
- Pay them accordingly

Defect Reports

- We need a mechanism for changing a product
- If the product appears to function incorrectly, the user files a defect report
  - It must include enough information to enable the maintenance programmer to recreate the problem
- Ideally, every defect should be fixed immediately
  - In practice, an immediate preliminary investigation is the best we can do

Handling New Defects

- The maintenance programmer should try to find
  - The cause,
  - A way to fix it, and
  - A way to work around the problem
- The new defect is now filed in the defect report file, together with supporting documentation
  - Listings
  - Designs
  - Manuals

Handling New Defects

- The file should also contain the client's requests for perfective and adaptive maintenance
  - The contents of the file must be prioritized by the client
  - The next modification is the one with the highest priority
- Copies of defect reports must be circulated to all
  - Including: An estimate of when the defect can be fixed
- If the same failure occurs at another site, the user can determine
  - If it is possible to work around the defect, and
  - How long until it can be fixed

Management of Postdelivery Maintenance

- In an ideal world
  - We fix every defect immediately
  - Then we distribute the new version of the product to all the sites
- In the real world
  - We distribute defect reports to all sites
  - We do not have the staff for instant maintenance
  - It is cheaper to make a number of changes at the same time, particularly if there are multiple sites

Authorizing Changes to the Product

- Corrective maintenance
  - Assign a maintenance programmer to determine the fault and its cause, then repair it
  - Test the fix, test the product as a whole (regression testing)
  - Update the documentation to reflect the changes made
  - Update the prologue comments to reflect
    - What was changed,
    - Why it was changed,
    - By whom, and
    - When

Authorizing Changes to the Product

- What if the programmer has not tested the fix adequately?
  - Before the product is distributed, it must be tested by the SQA group
- Postdelivery maintenance is extremely hard
- Testing is difficult and time consuming
  - Performed by the SQA group

Authorizing Changes to the Product

- The technique of baselines and private copies must be followed
- The programmer makes changes to private copies of code artifacts, tests them
- The programmer freezes the previous version, and gives the modified version to SQA to test
- SQA performs tests on the current baseline version of all code artifacts

Ensuring Maintainability

- Maintenance is not a one-time effort
- We must plan for maintenance over the entire life cycle
  - Design workflow — use information-hiding techniques
  - Implementation workflow — select variable names meaningful to future maintenance programmers
  - Documentation must be complete and correct, and reflect the current version of every artifact
- During postdelivery maintenance, maintainability must not be compromised
  - Always be conscious of the inevitable further maintenance
- Principles leading to maintainability are equally applicable to postdelivery maintenance itself

The Problem of Repeated Maintenance

- The moving target problem is frustrating to the development team
- Frequent changes have an adverse effect on the maintainability of the product

The Moving Target Problem

- The problem is exacerbated during postdelivery maintenance
- The more changes there are
  - The more the product deviates from its original design
  - The more difficult further changes become
  - Documentation becomes even less reliable than usual
  - Regression testing files are not up to date
  - A total rewrite may be needed for further maintenance

The Moving Target Problem

- Apparent solution
  - Freeze the specifications once they have been signed off until delivery of the product
  - After each request for perfective maintenance, freeze the specifications for (say) 3 months or 1 year
- In practice
  - The client can order changes the next day
  - If willing to pay the price, the client can order changes on a daily basis
- "He who pays the piper calls the tune"

Maintenance of Object-Oriented Software

- The object-oriented paradigm apparently promotes maintenance in four ways
  - The product consists of independent units
  - Encapsulation (conceptual independence)
  - Information hiding (physical independence)
  - Message-passing is the sole communication
- The reality is somewhat different
- Three obstacles
  - The complete inheritance hierarchy can be large
  - The consequences of polymorphism and dynamic binding
  - The consequences of inheritance

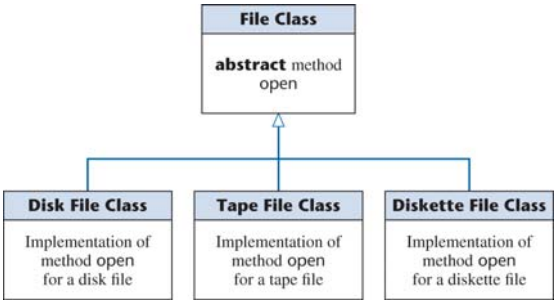
Size of the Inheritance Hierarchy

```
class UndirectedTreeClass
{
  ...
  void displayNode (Node a);
  ...
}// class UndirectedTreeClass

class DirectedTreeClass : public UndirectedTreeClass
{
  ...
}// class DirectedTreeClass

class RootedTreeClass : public DirectedTreeClass
{
  ...
  void displayNode (Node a);
  ...
}// class RootedTreeClass

class BinaryTreeClass : public RootedTreeClass
{
  ...
}// class BinaryTreeClass

class BalancedBinaryTreeClass : public BinaryTreeClass
{
  Node          hhh;
  displayNode (hhh);
}// class BalancedBinaryTreeClass
```

Size of Inheritance Hierarchy

- To find out what `displayNode` does in **`BalancedBinaryTreeClass`**, we must scan the complete tree
  - The inheritance tree may be spread over the entire product
  - A far cry from "independent units"
- Solution
  - A CASE tool can flatten the inheritance tree

Polymorphism and Dynamic Binding



- The product fails on the invocation `myFile.open ()`
- Which version of `open` contains the fault?
  - A CASE tool cannot help (static tool)
  - We must trace

Polymorphism and Dynamic Binding (contd)

- Polymorphism and dynamic binding can have
  - A positive effect on development, but
  - A negative effect on maintenance

Consequences of Inheritance

- Create a new subclass via inheritance
- The new subclass
  - Does not affect any superclass, and
  - Does not affect any other subclass
- Modify this new subclass
  - Again, no affect
- Modify a superclass
  - All descendent subclasses are affected
  - "Fragile base class problem"
- Inheritance can have
  - A positive effect on development, but
  - A negative effect on maintenance

Postdelivery Maintenance versus Development Skills

- The skills needed for maintenance include
  - The ability to determine the cause of failure of a large product
    - Also needed during integration and product testing
  - The ability to function effectively without adequate documentation
    - Documentation is rarely complete until delivery
  - Skills in analysis, design, implementation, and testing
    - All four activities are carried out during development

Postdelivery Maintenance vs. Development Skills

- The skills needed for postdelivery maintenance are the same as those for the other workflows
- Key Point
  - Maintenance programmers must not merely be skilled in a broad variety of areas, they must be highly skilled in all those areas
  - Specialization is impossible for the maintenance programmer
- Postdelivery maintenance is the same as development, only more so

Reverse Engineering

- When the only documentation for postdelivery maintenance is the code itself
  - Start with the code
  - Recreate the design
  - Recreate the specifications (extremely hard)
  - CASE tools can help (flowcharters, other visual aids)

Reverse Engineering

- Reengineering
  - Reverse engineering, followed by forward engineering
  - Lower to higher to lower levels of abstraction
- Restructuring
  - Improving the product without changing its functionality
  - Examples:
    - Prettyprinting
    - Structuring code
    - Improving maintainability
    - Restructuring (XP, agile processes)

Testing during Postdelivery Maintenance

- Maintainers tend to view a product as a set of loosely related components
  - They were not involved in the development of the product
- Regression testing is essential
  - Store test cases and their outcomes, modify as needed

CASE Tools for Postdelivery Maintenance

- Reengineering tools
  - Commercial tools
    - IBM Rational Rose, Together
  - Open-source tool
    - Doxygen
- Defect-tracking tools
  - Commercial tool
    - IBM Rational ClearQuest
    - Atlassian Jira
  - Open-source tool
    - Bugzilla

Metrics for Postdelivery Maintenance

- The activities of postdelivery maintenance are essentially those of development
  - Metrics for development workflows
- Defect report metrics
  - Defect classifications
  - Defect status

Challenges of Postdelivery Maintenance

- The chapter describes numerous challenges
- The hardest challenge to solve
  - Maintenance is harder than development, but
  - Developers tend to look down maintainers, and
  - Are frequently paid more

Next Week

- Reading
  - Chapter 17
- Project Goals
  - Work Breakdown
  - Start Coding
  - Lets Talk Technology