CHAPTER 15
IMPLEMENTATION

Lecture — Software Engineering

Topics
- Introduction
- Choice of Programming Language
- Fourth-Generation Languages
- Good Programming Practice
- Coding Standards
- Code Reuse
- Integration
- The Implementation Workflow
- The Implementation Workflow: The Art Dealer
- The Test Workflow: Implementation
- Test Case Selection

Lecture — Software Engineering

Topics
- Black-Box Unit-Testing Techniques
- Black-Box Test Cases: The Art Dealer
- Glass-Box Unit-Testing Techniques
- Code Walkthroughs and Inspections
- Comparison of Unit-Testing Techniques
- Cleanroom
- Potential Problems when Testing Objects
- Management Aspects of Unit Testing
- When to Reimplement Rather than Debug a Code Artifact

Lecture — Software Engineering

Topics
- Integration Testing
- Product Testing
- Acceptance Testing
- CASE Tools for Implementation
- CASE Tools for the Test Workflow
- Metrics for the Implementation Workflow
- Challenges of the Implementation Workflow
Introduction

- Implementation — Process of translating detailed design into code
- Programming-in-the-many — Product implemented by a team
- Team working at the same time on different components of product

Choice of Programming Language

- Deciding on a programming language
- In most cases issue does not arise
  - Client wants product written in a specific language
  - Limitations due to the choice of computer
- Choice of most suitable programming language
  - Staff experience versus
  - New technologies
- Decision using cost-benefit analysis
  - Computation of implementation cost
  - Repeated for every language
- Decision using risk analysis
  - Potential risks for every language
- Pressure on organizations to use object-oriented languages
  - Smalltalk, C++, Java

Fourth-Generation Languages

- Language generations
- First: Computers programmed in binary
  - Hard-wired or setting switches
- Second: Assemblers
  - Developed in 1940s and 1950s
  - Symbolic notations translated into machine code
- Third: High-level languages
  - E.g., C, C++, Pascal, Java
  - One statement compiled into 5 to 10 machine code instructions
- Fourth: Fourth-generation languages (4GL)
  - Introduced in the late 1970s
  - E.g., Focus, Natural
  - One statement equivalent to 30 to 50 machine code instructions

Reference

Fourth-Generation Languages

- Example
  for every surveyor
  if rating is excellent add 6500 to salary
- Fourth-generation languages (4GL) are non-procedural
  - Instructions translated into a sequence of machine code instructions
  - Compiler or interpreter generally large and slow
- Results
  - Some organizations reported increase in productivity
  - Some organizations reported disappointment with results
- Reasons for inconsistent results
  - Unlikely one 4GL appropriate for all products
  - Many 4GLs supported by powerful CASE workbenches and environments: Inadvisable to introduce large scale CASE into organization not ready

No one 4GL dominates the software market
- There are hundreds of 4GLs
- DB2, Oracle, and PowerBuilder have sizeable user groups
- Concept of end-user-programming
  - Design objective of 4GL
  - Programming by person who will use the product
  - Dangers of inexperienced programmers
    - Bad code
    - Corrupted databases
- Ultimate choice of a 4GL is made by management: Careful analysis of successes and failures

Good Programming Practice

- Many recommendations on good coding style are language specific
- Recommendations regarding language-independent good programming practice
  - Use of Consistent and Meaningful Variable Names
    - Meaningful variable names
      - From viewpoint of future maintenance programmers
    - Consistent variable names
      - Consistency in naming and ordering of components of name
      - e.g., frequencyMaximum and minimumFrequency
    - Use of naming conventions
    - Type information incorporated in name
      - E.g., ptrChTmp is a temporary variable of type pointer to a character

Hungarian Naming Conventions
- By Charles Simonyi (born in Hungary)
- Programs conforming to the convention as easy to read as Hungarian

Effectiveness of code inspection can be reduced
- Difficulty in pronouncing variable names

The Issues of Self-Documenting Code

- Self-documenting code does exist, but it is exceedingly rare
- Prologue comments at the beginning of code artifact
  - Mandatory minimum information
- In-line comments
  - To assist maintenance programmers
  - Not used to promote or excuse bad programming
**Good Programming Practice**

- **Figure 15.1**

  | The name of the code artifact  |
  | A brief description of what the code artifact does  |
  | The programmer’s name  |
  | The date the code artifact was coded  |
  | The date the code artifact was approved  |
  | The name of the person who approved the code artifact  |
  | The arguments of the code artifact  |
  | A list of the name of each variable of the code artifact, preferably in alphabetical order, and a brief description of its use  |
  | The names of any files accessed by this code artifact  |
  | The names of any files changed by this code artifact  |
  | Input-output, if any  |
  | Error-handling capabilities  |
  | The name of the file containing test data (to be used later for regression testing)  |
  | A list of each modification made to the code artifact, the date the modification was made, and who approved the modification  |
  | Any known faults  |

**Use of Parameters**

- There are very few genuine constants
  - E.g., sale tax: Can change
- Apparent constants should be treated as parameters
  - Should be read from a parameter file
  - At the beginning of the run

**Code Layout for Increased Readability**

- No more than one statement on a line
- Use of indentation for readability
- Use of blank lines to break up large blocks of code

**Nested if Statements**

- Complex if conditions changed to equivalent
- if statements not nested to a depth greater than three

**Coding Standards**

- Aim of coding standards is to make maintenance easier
- Effect is making the life of software developers difficult
- Overly restrictive coding standards are counterproductive
- Issues
  - Coding standards imposed from above tend to be ignored
  - Unless checked by machine, takes a lot of SQA group’s time
- Some rules can be checked by machine and are helpful if followed
  - Nesting of if statements should not exceed a depth of three
  - Modules should consist of between 35 to 50 executable statements
  - Use of goto statements should be avoided

**Code Reuse**

- Artifacts form all workflows of software process are reused
  - Specifications
  - Contracts
  - Plans
  - Design
  - Code artifacts
- Reuse of code is most common
**Integration**

- Possible approach to integration: Code and test each code artifact separately, then link them together and test product as a whole.
- Difficulty 1: Artifacts calling other artifacts cannot be tested separately.
  - Use of stubs — Empty artifact to call for testing or generating or printing messages.
  - Use of drivers — A code artifact that calls an artifact one or more times.
  - Efforts into constructing stubs and drivers thrown away after testing.
- Difficulty 2: Lack of fault isolation.
  - If product as a whole fails: Fault is difficult to isolate.
- Solution to both these difficulties is to combine unit and integration testing.

**Top-Down Integration**

- If code artifact above sends a message to artifact below: Artifact above is implemented and integrated before below.
- Example interconnection diagram:
  - Artifact a coded and tested with b, c, and d as stubs.
  - Next, stub b expanded into artifact b.
- Possible ordering: a, b, c, d, e, f, g, h, i, j, k, l, and m.
- Portions of the integration can proceed in parallel:
  - After a is coded and tested.
  - One programmer uses a to implement and integrate b, e, and h.
  - Another programmer uses a to work in parallel on c, d, f, and i.

**Figure 15.6**

- Artifacts of product can be divided into two groups:
  - Logic artifacts — Decision making flow: Generally situated close to root.
  - Operational artifacts — Performing actual operations: Generally found in lower levels.
- Strength of top-down integration: Major design flaws show up early.
- Coding and testing of logic artifacts before operational.
- If whole product completed before a major fault detected:
  - Large parts will have to be rewritten.
  - Many operational modules may be reusable.
  - Their interconnections will have to be changed.
- Earlier detection quicker and less costly.
Integration

- Weakness of top-down integration: Potentially reusable code artifacts may not be adequately tested
- Operational artifacts are probably reusable: If they have cohesion
- Operational artifacts are at lower levels
- Inadequate testing of operational artifacts
- Better the design: Less thoroughly artifacts are likely to be tested

Alternatives
- Defensive programming — Calling artifact include safety checks
  Operational artifacts less likely to be fully tested
- Responsibility-driven design — Necessary safety checks are built into invoked artifact

Integration

Bottom-Up Integration
- If code artifact above sends a message to artifact below:
  Artifact below is implemented and integrated before above
- Example possible ordering: l, m, h, i, j, k, e, f, g, b, c, d, and a
- Operational artifacts are tested thoroughly
- Testing is done with aid of drivers
- Advantage: Fault isolation
- Difficulty: Major design faults are detected late in implementation workflow
  - Logic artifacts are integrated last

Sandwich Integration
- Combines two strategies of top-down and bottom-up
  - Maximizing their strengths
  - Minimizing their weaknesses

Integration

- Partition modules into two groups
- Implement and integrate logic artifacts top down
- Implement and integrate operational artifacts bottom up
- Test interfaces between logic artifacts and operational artifacts
- Advantages
  - Major design faults caught early
  - Through testing of operational artifacts
  - Fault isolation at all times

Integration

- Figure 15.7
Integration

- Strengths and weaknesses of integration approaches
- Figure 15.8

<table>
<thead>
<tr>
<th>Approach</th>
<th>Strengths</th>
<th>Weaknesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implementation then integration (Section 15.6)</td>
<td>—</td>
<td>No fault isolation</td>
</tr>
<tr>
<td>Top-down integration (Section 15.6.1)</td>
<td>Fault isolation</td>
<td>Major design faults show up late</td>
</tr>
<tr>
<td>Bottom-up integration (Section 15.6.2)</td>
<td>Fault isolation</td>
<td>Potentially reusable code artifacts are not adequately tested</td>
</tr>
<tr>
<td>Sandwich integration (Section 15.6.3)</td>
<td>Fault isolation</td>
<td>Major design faults show up early</td>
</tr>
<tr>
<td></td>
<td>Potentially reusable code artifacts are adequately tested</td>
<td></td>
</tr>
</tbody>
</table>

Integration of Object-Oriented Products

- Objects can be integrated either bottom up or top down.
- Top-down
  - Stubs are used
- Bottom-up
  - First, objects that do not send messages are implemented and integrated
  - Next, objects that send messages are implemented and integrated
- In hybrid object-oriented languages (e.g., C++)
  - Classes are operational artifacts: Integrated bottom up
  - Artifacts that are not classes are logic artifacts: Integrated top down
  - Finally, all non-object artifacts are integrated with objects

The Implementation Workflow

- Aim of implementation workflow: Implement the target software product
- In the selected implementation language
- A large software product is partitioned into smaller subsystems (components or code artifacts)
- Implemented in parallel by coding teams
- After a code artifact is coded, the programmer tests it (unit testing)
- Once programmer decides artifact is correct, it is passed to the SQA group for further testing

Management of Integration

- In pure object-oriented languages (e.g., Java)
  - Class methods (static methods such as main and utility methods) are implemented top down: Similar to logic modules
  - Integrated with other objects

- Problem for management: Discovering at integration time that the code artifacts do not fit together
- Solution: Entire integration process should be run by SQA group
- Manager of SQA group should have responsibility for all aspects of integration testing
- At the end of integration process, all code artifacts will have been tested and combined into a single product
The Test Workflow: Implementation

- Different types of testing have to be performed during implementation workflow
  - Unit testing
  - Integration testing
  - Product testing
  - Acceptance testing
- Code artifacts (modules, classes) undergo two types of testing
  - Informal unit testing (by the programmer)
  - Methodical unit testing (by the SQA group)
- There are two types of methodical testing
  - Non-execution-based testing: Artifact reviewed by a team
  - Execution-based testing: Artifact is run against test cases

Test Case Selection

- Test case selection must be performed systematically
- Inefficient to use haphazard test data
- Only time for a fraction of all possible test cases

Testing to Specifications versus Testing to Code

- Test data for unit testing can be constructed systematically in two basic ways
- Test to specifications
  - Called Black-box, behavioral, data-driven, functional, and input-output driven testing
  - Code is ignored and specification document is used
- Test to code
  - Called glass-box, white-box, structural, logic-driven, and path-oriented testing
  - Ignore specification document when selecting test cases

Feasibility of Testing to Specification

- Exhaustive testing to specification is impossible in practice
- Combinatorial explosion

Feasibility of Testing to Code

- Most common form requires that each path through code artifact be tested at least once
- Huge number of possible paths impractical to test
- E.g., flowchart with over $10^{12}$ possible paths

Test Case Selection

- Five possible paths through central group
- Loop up to 18 times
- Total number of possible paths
  \[ 5^1 + 5^2 + 5^3 + \ldots + 5^{18} = \frac{5 \times (5^{18} - 1)}{(5 - 1)} = 4.77 \times 10^{12} \]
- Figure 15.10
Test Case Selection

- Possible to exercise every path without detecting every fault
- A path can be tested only if it is present
  - E.g., not testing for division by zero
- Neither exhaustive testing to specifications nor exhaustive testing to code is feasible
- A compromise is needed
- Using techniques that highlight as many faults as possible
- Accepting that there is no way to guarantee that all faults have been detected
- A reasonable way to proceed
  - Use black-box test cases first (test to specifications)
  - Then develop additional test cases using glass-box (test to code)

Black-Box Unit-Testing Techniques

- Since exhaustive black-box testing requires a very large number of test cases: A manageable set of test cases must be devised
- To maximize chances of detecting faults
- While minimizing chances of wasting a test case: Having same fault detected by more than one test case

Equivalence Testing and Boundary Value Analysis

- Equivalence class — Set of test cases such that any one member of class is as good a test case as any other
- Testing requires one test case from each equivalence class
- Process of equivalence testing
  - Five test cases for a range (A, B): <A, A, >A and <B, B, >B
  - Two cases for set membership: A member, and a non-member
  - Two cases for precise value: Specified value and anything else

Black-Box Unit-Testing Techniques

- Boundary value analysis — Based on concept that selecting a test case on or just to one side of the boundary of an equivalence class increases probability of detecting a fault
- High-payoff technique: Use of equivalence classes, together with boundary value analysis
- To test both input specifications and output specifications
- Generating a relatively small set of test data
- Potential of uncovering a number of faults

Functional Testing

- An alternative from of black-box testing is to base the test on functionality of a code artifact
- Each item of functionality or function implemented in code artifact is identified
- Test data are devised to test each function separately

Black-Box Unit-Testing Techniques

- If code artifact consists of a hierarchy of lower-level functions: Functional testing proceeds recursively
- In practice, lower-level functions can be intertwined in some way
- Functional analysis is required: A complex procedure
- Further complicating factor: Functionality frequently does not coincide with code artifact boundaries
- Distinction between unit testing and integration testing blurred
- One code artifact cannot be tested without testing other code artifacts
Black-Box Test Cases: The Art Dealer

- Black-box test cases
- Derived from equivalence classes and boundary value analysis
- Figure

**Painting data**

Equivalence classes for first name and last name
1. First character not alphabetic
2. First character alphabetic
3. <1 character
4. 1 character
5. Between 1 and 21 characters
6. 21 characters
7. >21 characters

Equivalence classes for title
1. <1 character
2. 1 character
3. Between 1 and 41 characters
4. 41 characters
5. >41 characters

**Equivalence classes for medium**
1. "oil"
2. "watercolor"
3. "other"
4. Any other string

(Additional tests could be made to check that the number of days is valid for the corresponding month; for example, the month of February should not have 31 days).

**Gallery data**

(Attributes classification, purchase date, sale date, algorithm price, and target price are determined by the system, and are not entered by the user.)

- First character not alphabetic
- <1 character
- 1 character
- Between 1 and 21 characters
- 21 characters
- >21 characters

- Characters instead of integers
- Error (invalid value)

- Error (not a number)
Black-Box Test Cases: The Art Dealer

- Black-box test cases
- Derived from equivalence classes and boundary value analysis
- Figure

<table>
<thead>
<tr>
<th>Equivalence classes for purchase price and selling price</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. &lt;=0.00</td>
</tr>
<tr>
<td>2. 0.00</td>
</tr>
<tr>
<td>3. 0.01</td>
</tr>
<tr>
<td>4. Between 0.01 and 999.99</td>
</tr>
<tr>
<td>5. 1000.00</td>
</tr>
<tr>
<td>6. &gt;1000.00</td>
</tr>
<tr>
<td>7. Characters instead of integers</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acceptable</td>
</tr>
<tr>
<td>Error (not a number)</td>
</tr>
</tbody>
</table>

Black-Box Test Cases: The Art Dealer

- Functional testing test cases
- Figure

The functions outlined in the specification document are used to create test cases.

1. Buy a masterpiece where the artist cannot be found in the auction records.
2. Buy a masterpiece where the artist can be found in the auction records.
3. Buy a masterwork where the artist cannot be found in the auction records.
4. Buy a masterwork where the artist can be found in the auction records.
5. Update the fashionability coefficients of several artists.
6. Buy an "ether" type of painting where the artist cannot be found in the fashion records.
7. Buy an "ether" type of painting where the artist can be found in the fashion records.
8. Buy a painting where the purchase price is less than the price suggested by the algorithm.
9. Buy a painting where the purchase price is equal to the price suggested by the algorithm.
10. Buy a painting where the purchase price is greater than the price suggested by the algorithm.
11. Sell a painting where the selling price is less than the target price.
12. Sell a painting where the selling price is equal to the target price.
13. Sell a painting where the selling price is greater than the target price.
14. For one or more artists, sell at least two paintings where every painting is sold over the target price (to be used in test case 17).
15. Display report of bought paintings.
17. Display report of fashionability trends.

In addition to these direct tests, it is necessary to perform the following further tests:

18. Attempt to buy a painting that is already in the gallery.
19. Attempt to sell a painting that does not exist in the gallery.
20. Attempt to sell a painting that has already been sold.

- Functional testing test cases
- Figure

<table>
<thead>
<tr>
<th>Fashionability data</th>
</tr>
</thead>
<tbody>
<tr>
<td>First name and last name</td>
</tr>
<tr>
<td>1. First character not alphanumeric Error</td>
</tr>
<tr>
<td>2. &lt;=1 character Error</td>
</tr>
<tr>
<td>3. 1 character Acceptable</td>
</tr>
<tr>
<td>4. Between 1 and 21 characters Acceptable</td>
</tr>
<tr>
<td>5. 21 characters Acceptable</td>
</tr>
<tr>
<td>6. &gt;21 characters Acceptable (truncated to 21 characters)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Equivalence classes for coefficient</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. &lt;=0.00</td>
</tr>
<tr>
<td>2. 0.00</td>
</tr>
<tr>
<td>3. 0.01</td>
</tr>
<tr>
<td>4. Between 0.01 and 999.99</td>
</tr>
<tr>
<td>5. 10000.00</td>
</tr>
<tr>
<td>6. &gt;10000.00</td>
</tr>
<tr>
<td>7. Characters instead of integers</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acceptable</td>
</tr>
<tr>
<td>Error (not a number)</td>
</tr>
</tbody>
</table>
Glass-Box Unit-Testing Techniques

- **Glass-box techniques**: Test cases are selected on basis of examination of the code, rather than the specifications.

**Structural Testing: Statement, Branch, and Path Coverage**

- **Statement coverage** — Running a series of test cases during which every statement is executed at least once
  - CASE tool for keeping a record of how many times each statement has been executed
  - E.g., PureCoverage
  - No guarantee that all outcomes of branches are properly tested

- **Branch coverage** — running a series of tests to ensure that all branches are tested at least once
  - Improvement over statement coverage
  - E.g., Generic Coverage Tool (gct) for C programs

- **Path coverage** is testing all paths
  - Most powerful form of structural testing

Complexity Metrics

- Computer scientists have developed a number of metrics of software complexity
- Used in determining which code artifacts are most likely to have faults
- Very complex code artifacts may be redesigned and re-implemented
- A simple metric is lines of code
- Assumes a constant probability a line of code contains a fault

Glass-Box Unit-Testing Techniques

- Number of paths can be very large: In a product with loops
- Ways for reducing number of paths to be examined
  - Restrict test cases to linear code sequences: Paths that begin and end with control flow jumps (branches, etc.)
  - All-definition-use-path coverage: Paths between definition of a variable and its use

Studies analyzing data available on fault densities

- Most complexity metrics show a high correlation with number of lines of code (deliverable, executable source instructions)
- Number of lines of code correlates strongly with the number of faults
- Complexity metrics provide little improvement over lines of code for predicting fault rates
**Code Walkthroughs and Inspections**

- Code walkthroughs and inspections lead to rapid, thorough, and early fault detection
- Time required more than repaid by increased productivity
- Presence of fewer faults during integration
- Reduction of up to 95% in corrective maintenance costs
- The alternative, execution-based testing, can be expensive
  - Time consuming
  - Inspections lead to detection and correction of faults earlier in life cycle
  - E.g., 80% of software budget of NASA Apollo Program was consumed by testing

**Comparison of Unit-Testing Techniques**

- Number of studies compared strategies for unit testing
- Experiment
  - 59 highly experienced programmers testing same product
  - Comparing: (1) Black-box testing (2) Combination of black-box testing and glass-box testing (3) Three-person code walkthroughs
  - Three techniques equally effective in finding faults
  - Code walkthroughs less cost-effective than other two techniques
- Experiment
  - Comparing: (1) Black-box testing (2) Glass-box testing (3) One-person code reading
  - All three techniques equally effective
  - Each technique having its own strengths and weaknesses

---

**Comparison of Unit-Testing Techniques**

- Experiment
  - 32 professional programmers and 42 advanced students
  - Comparing: (1) Black-box testing (2) Glass-box testing (3) One-person code reading
  - Each tested three products, using each testing technique once
  - Professional programmers
    - Detected more faults with code reading
    - Fault detection rate faster with code reading
  - One student group
    - No significant differences
  - The other student group
    - Code reading and black-box outperform glass-box
    - Same detection rate
  - Overall, code reading led to detection of more interface faults than other two techniques
  - Black-box testing was most successful at finding control faults

---

**Comparison of Unit-Testing Techniques**

- Subsequent experiment
  - Black-box testing and glass-box testing more efficient or more effective than inspections
- Subsequent experiment
  - Test cases and inspections tend to find different kinds of faults
  - Two techniques are complementary
    - Both need to be utilized
    - Conclusion used in Cleanroom software development technique
Cleanroom

- Cleanroom technique — Combination of different software development techniques
- An incremental life-cycle model
- Formal techniques for analysis and design
- Non-execution-based module-testing techniques
  - Code reading, code walkthroughs, and inspections
- Critical aspect: A code artifact is not compiled until it has passed an inspection
- Compiled only after non-execution-based testing
- Technique has had a number of great successes
  - Cases with no compilation error
  - Applied to small-scale software products
  - Not necessarily scale up to large-scale software
  - Results for larger products also impressive

Relevant metric is testing fault rate — Total number of faults detected per KLOC
- Relatively common metric in software industry
- Faults counted
  - Traditional development: From the time SQA begins testing
  - Cleanroom development: From the time of compiling
- Studies on use of Cleanroom
  - 17 Cleanroom products
  - Total of 1 million lines of code
  - Average testing fault rate of 2.3 per KLOC
  - Considered remarkable quality achievement

Potential Problems when Testing Objects

- Certain problems are specific to testing of objects in object-oriented paradigm
- Class — Abstract data type that supports inheritance
  - A class has no concrete realization
  - Impossible to perform execution-based testing of a class
  - Only non-execution-based testing
- Object — Instance of a class
  - An object is a physical piece of code
- Information hiding and methods with relatively few lines of code
  - Some methods do not return values to be tested
  - Some methods change the state of object: Modify attributes (state variables) of object
  - Additional messages must be sent to objects to test

Objects may not include methods that can be invoked: Additional methods needed for testing
- Inherited methods may still have to be tested
- These complications are no reason to abandon the object-oriented paradigm
  - They arise only through interaction of methods
  - It is possible to detect when retesting is needed
Management Aspects of Unit Testing

● An important decision to be made during development of every code artifact: How much time (and therefore money) to spend on testing that artifact

● Cost-benefit analysis can be used
  ▪ Comparing cost of running additional test cases
  ▪ Against cost of failure of delivered product

● Techniques of reliability analysis
  ▪ Provide statistical estimates of remaining faults
  ▪ E.g., fault detection rate decreasing steadily

● Zero-failure technique — Used for determining how long to continue testing a product
  ▪ Associated with statistical-based testing
  ▪ Underlying idea is that longer a product is tested without a single failure being observed: Greater likelihood that product is free of faults

Management Aspects of Unit Testing

● Reasonable assumption: Chances of a failure occurring decreases exponentially as testing proceeds

● Number of test hours required without a single failure occurring

\[ \frac{\ln\left(\frac{f_{\text{target}}}{0.5 + f_{\text{target}}}\right)}{\ln\left(\frac{f_{\text{total}}}{0.5 + f_{\text{target}}}\right)} \times t_h \]

where

- \( f_{\text{target}} \) — Target projected number of failures
- \( f_{\text{total}} \) — Total number of failures detected so far
- \( t_h \) — Total number of test hours up to the last failure

Management Aspects of Unit Testing

● Example: How many more hours of testing for a product with 50,000 lines of code, contract specifies not more than 0.02 failures per thousand lines of code, product has been tested for 400 hours, a total of 20 failures are detected, and product has run for 50 hours since last failure

\[ f_{\text{total}} = 20 \quad f_{\text{target}} = \frac{50000}{1000} \times 0.02 = 1 \quad t_h = 400 - 50 = 350 \]

\[ \frac{\ln\left(\frac{1}{0.5 + 1}\right)}{\ln\left(\frac{350}{20 + 1}\right)} \times 350 = 54 \text{ hours} \]

54 - 50 = 4 more hours

● If a failure is detected during last 4 hours: Additional testing is required

Management Aspects of Unit Testing

● On some occasions, it is preferable for code to be thrown away and redesigned and recoded from scratch
  ▪ Either by the original programmer
  ▪ Or more senior member of development team

● Probability of existence of more faults in a code artifact is proportional to number of faults already found in that code artifact

● Distribution of faults in modules not uniform

● Figure 15.18

When to Reimplement Rather than Debug a Code Artifact
When to Reimplement Rather than Debug a Code Artifact

- E.g., faults found by users in OS/370
  - 47% associated with only 4% of modules
- Management can predetermine maximum number of faults permitted during development
- Code artifact thrown away, redesigned, and recoded if maximum is reached
- Maximum varies depending on domains and code artifacts
- Maximum determined using fault data on similar code artifacts

Integration Testing

- In integration testing, each new code artifact must be tested when it is added to what has already been integrated
  - First, test the new code artifact
  - Then test that the rest of partial product continues to behave as before
- Special issues with regard to integration testing of a product with a graphical user interface (GUI)
  - General approach
    - Storing input data for a test in a file
    - CASE tools to automate testing
    - Run each test case
    - Compare actual results with expected results
    - Reports are generated for each case
    - Test case are stored for regression testing
    - E.g., SilkTest toolkit

Integration Testing

- When a product incorporates a GUI
  - General approach does not work
  - Test data for menus and using mouse cannot be stored in a file as conventional test data
  - Time consuming and boring to test a GUI manually
- Solution
  - CASE tool keeps a record of mouse clicks, key presses, etc.
  - GUI is tested once manually to set up CASE tool test file
  - File is used subsequently
  - E.g., QAPartner and XRUnner
- When integration process is complete: Product as a whole is tested (product testing)
- When developers are confident about correctness of every aspect of product: Product is handed over to client for testing (acceptance testing)

Product Testing

- SQA group must perform a number of testing tasks to ascertain that product will be successful
- COTS software
  - Ensuring product as a whole is free of faults
  - Afterwards, product undergoes alpha and beta testing by selected perspective buyers
- Custom software
  - Different product testing
  - Failure during acceptance testing poor reflection on management
- SQA group must test product using tests that SQA group believes closely approximate forthcoming acceptance tests
- Black-box testing of product as a whole
- Testing robustness of product as a whole
  - Stress testing: Operating under a peak load
  - Volume testing: Handling large files
Product Testing

- Checking that product satisfies all its constraints
  - Speed
  - Storage
  - Security
- Reviewing documentation to be handed over to client
  - Conforming to the standards set in SPMP
  - Verifying against product

Acceptance Testing

- Purpose of acceptance testing: Client to determine whether product satisfies its specifications as claimed by developers
- Acceptance tests are performed by
  - Client organization
  - SQA group in presence of client representatives
  - Independent SQA group hired by client
- Four major components of acceptance testing
  - Correctness
  - Robustness
  - Performance
  - Documentation
- Acceptance testing must be performed on actual data
  - Rather than test data

CASE Tools for Implementation

- CASE tools for implementation of code artifacts have been described
- CASE tools for integration
  - Version control tools
  - Build tools
  - Configuration management tools

CASE Tools for the Complete Software Process

- Ideally, every software development organization should utilize a CASE environment
  - Cost can be large
  - For smaller organizations: Workbench or a set of tools may suffice
  - If possible, an integrated environment should be utilized to support development and maintenance
**CASE Tools for Implementation**

**Integrated Development Environments**
- Most common integration is in terms of user interface integration
- All tools have same visual appearance
- Ease in learning and using another tool in the environment
- E.g., Macintosh applications with similar "look and feel"
- Tool integration — All tools communicate via same data format
  - E.g., UNIX Programmer’s Workbench (pipe assumes all data of ASCII stream format)
  - Easy to combine two tools: Directing output stream from one to input of the other
- Process integration — An environment that supports one specific software process
  - A subset of this class of environment is technique-based environment (supports a specific technique for developing software)

**CASE Tools for Implementation**

- Strength: Users are forced to use a specific technique correctly
- Weakness: Counterproductive unless organization incorporates the specific technique

**Environments for Business Applications**
- An important class of environments is used for building business-oriented products
- Emphasis is on ease of use
- A number of standard screens
  - Modified easily via a user-friendly GUI generator
- Code generator for languages
  - Input: Detailed design
  - Output: C, C++, Java
- Automatically generated code is compiled
  - No programming of any kind

**CASE Tools for Implementation**

- E.g., structure systems analysis, Jackson system development and Petri nets
- Provide graphical support for analysis and design
- Incorporate a data dictionary
- Some consistency checking is provided
- Support for managing development process is incorporated
- Commercially available environments
  - E.g., Analyst/Designer, Rhapsody
- Object-oriented methodologies
  - IBM Rational Rose supports Unified Process
  - Older environments extended to support object-oriented paradigm
    - E.g., Software through Pictures
  - Almost all object-oriented environments now support UML

**CASE Tools for Implementation**

- Languages for specifying detailed design
  - Could be programming languages of future
- A number of environments are currently available
  - E.g., Oracle Developer Suite
- Likely many more environments will be developed in future
  - Size of market for business-oriented CASE environments

**Public Tool Infrastructures**
- An infrastructure developed for supporting CASE tools
- Called Portable Common Tool Environment (PCTE)
- Developed by European Strategic Programme for Research in Information Technology (ESPRIT)
- It is not an environment
- It is an infrastructure that provides services needed by CASE tools
CASE Tools for Implementation

- PCTE has gained widespread acceptance
- PCTE and C and Ada interfaces to PCTE adopted as ISO/IEC Standard 13719
- It is hoped that
  - More CASE tools will conform to PCTE standard
  - PCTE will be implemented on a wider variety of computers
- A tool that conforms to PCTE would run on any computer that supports PCTE
- Should result in widespread availability of a broad range of CASE tools
- Should lead to better software processes and better quality software

Potential Problems with Environments

- No one environment is ideal for all products and all organizations
- Every environment has its
  - Strengths
  - Weaknesses
- Use of a CASE environment should be avoided until organization has attained CMM level 3
  - Automation of a nonexistence process can lead to chaos
- Every organization should use CASE tools
- Little harm in using CASE workbenches

CASE Tools for the Test Workflow

- Numerous CASE tools to support different types of testing
- Unit testing
  - XUnit testing frameworks: Open-source automated tools
  - E.g., JUnit for Java and CppUnit for C++
  - Test each class using a set of test cases
  - Checks that each of messages sent results in expected returns
  - Commercial tools produced by vendors, including Parasoft
- Integration testing
  - Commercial tools
  - E.g., SilkTest, IBM Rational Functional Tester
  - Pool unit testing test cases and utilize resulting set of test cases
- Defect-tracking
  - Essential for management to know status of all defects
  - Open-source product: Bugzilla
- Hyades project
  - Eclipse test and performance tools project
  - Open-source integrated test, trace, and monitoring environment
  - Currently used with Java and C++
  - Facilities for different testing tools
  - As tool vendors adapt their tools to work under Eclipse: Users can select from a wider choice of testing tools
**Metrics for the Implementation Workflow**

- A number of different complexity metrics for implementation workflow
  - Lines of code
  - Cyclomatic complexity
- Relevant metrics from a testing viewpoint
  - Total number of test cases
  - Number of test cases that resulted in failure
  - Total number of faults
  - Statistics regarding the types of faults detected
    - Misunderstanding design
    - Lack of initialization
    - Inconsistent use of variables
- Fault data
  - Can be incorporated into checklist
  - To be used during code inspection of future products

**Challenges of the Implementation Workflow**

- A major challenge of implementation workflow has to be met in workflows that precede it
  - Code reuse is an effective way of reducing software development cost and delivery time
  - Difficult to achieve code reuse if it is attempted as late as implementation workflow
- Code reuse has to be built into a software product from very beginning
  - Reuse has to be a user requirement and a constraint of specification document
  - Software project management plan must incorporate reuse
  - Design document must state which code artifacts are to be implemented and which are to be reused
  - Code reuse has to be incorporated into requirement, analysis, and the design workflows

**Metrics for the Implementation Workflow**

- Metrics specific to object-oriented paradigm
  - E.g., the height of the inheritance tree
  - Metrics questioned on both theoretical and experimental grounds
  - Remains to be shown that there is a need for specifically object-oriented metrics

**Challenges of the Implementation Workflow**

- Implementation workflow is relatively straightforward
  - If requirements, analysis, and design workflows carried out properly
- Management of integration is of critical importance
  - Challenges of implementation workflow are to be found in this area
- Typical make-or-break issues
  - Use of appropriate CASE tools
  - Test planning once specifications have been signed off by client
  - Ensuring that changes to design are communicated to all relevant personnel
  - Deciding when to stop testing and deliver product to client