Slide 2.1

Object-Oriented Software Engineering

WCB/McGraw-Hill, 2008

Stephen R. Schach
srs@vuse.vanderbilt.edu

CHAPTER 1, 2, and 3

Slide 2.2

SOFTWARE LIFE-CYCLE MODELS AND PROCESS

Overview

Slide 2.3

- Software life-cycle models
- Unified process
- Capability Maturity Models (CMM)

Software Development in Theory

Slide 2.4

- Classical model (1970)

1. Requirements phase
2. Analysis (specification) phase
3. Design phase
4. Implementation phase
5. Postdelivery maintenance
6. Retirement

Figure 1.2

Typical Classical Phases

Slide 2.5

- Requirements phase
  - Explore the concept
  - Elicit the client's requirements
- Analysis (specification) phase
  - Analyze the client's requirements
  - Draw up the specification document
  - Draw up the software project management plan
  - "What the product is supposed to do"

Typical Classical Phases (contd)

Slide 2.6

- Design phase
  - Architectural design, followed by
  - Detailed design
  - "How the product does it"
- Implementation phase
  - Coding
  - Unit testing
  - Integration
  - Acceptance testing

Typical Classical Phases (contd)

Slide 2.7

- Postdelivery maintenance
  - Corrective maintenance
  - Perfective maintenance
  - Adaptive maintenance
- Retirement

Software Development in Practice

Slide 2.8

- In the real world, software development is totally different
  - We make mistakes
  - The client's requirements change while the software product is being developed

2.2 Winburg Mini Case Study

Slide 2.9

- **Episode 1**: The first version is implemented
- **Episode 2:** A fault is found
  - The product is too slow because of an implementation fault
  - Changes to the implementation are begun
- **Episode 3:** A new design is adopted
  - A faster algorithm is used
- **Episode 4:** The requirements change
  - Accuracy has to be increased
- **Epilogue:** A few years later, these problems recur

Waterfall Model

Slide 2.10

- The linear life cycle model with feedback loops
  - The waterfall model cannot show the order of events

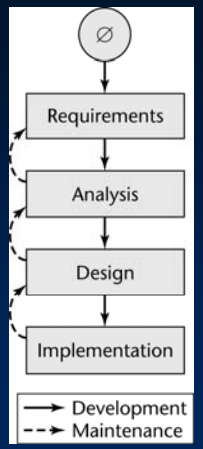

Figure 2.3

2.3 Lessons of the Winburg Mini Case Study

Slide 2.11

- In the real world, software development is more chaotic than the Winburg mini case study
- Changes are always needed
  - A software product is a model of the real world, which is continually changing
  - Software professionals are human, and therefore make mistakes

2.4 Teal Tractors Mini Case Study

Slide 2.12

- While the Teal Tractors software product is being constructed, the requirements change
- The company is expanding into Canada
- Changes needed include:
  - Additional sales regions must be added
  - The product must be able to handle Canadian taxes and other business aspects that are handled differently
  - Third, the product must be extended to handle two different currencies, USD and CAD

Teal Tractors Mini Case Study (contd)

Slide 2.13

- These changes may be
  - Great for the company; but
  - Disastrous for the software product

Moving Target Problem

Slide 2.14

- A change in the requirements while the software product is being developed
  - Regression fault
- No solution

2.5 Iteration and Incrementation

Slide 2.15

- The basic software development process is iterative
  - Each successive version is intended to be closer to its target than its predecessor

Miller's Law (1956)

Slide 2.16

- At any one time, we can concentrate on only approximately seven *chunks* (units of information)
- To handle larger amounts of information, use *stepwise refinement*
  - Concentrate on the aspects that are currently the most important
  - Postpone aspects that are currently less critical
  - Every aspect is eventually handled, but in order of current importance
- This is an *incremental* process

Iteration and Incrementation (contd)

Slide 2.17

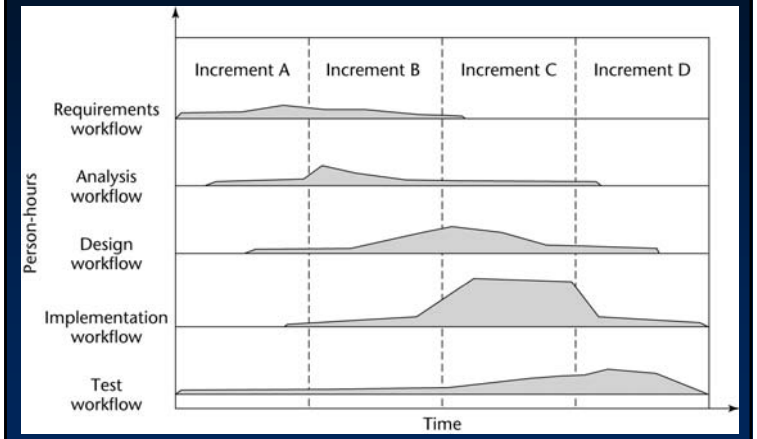

Figure 2.4

Iteration and Incrementation (contd)

Slide 2.18

- The number of increments will vary — it does not have to be four

Sequential Phases versus Workflows

Slide 2.19

- Sequential phases do not exist in the real world
- Instead, the five core workflows (activities) are performed over the entire life cycle
  - Requirements workflow
  - Analysis workflow
  - Design workflow
  - Implementation workflow
  - Test workflow

Workflows

Slide 2.20

- All five core workflows are performed over the entire life cycle
- However, at most times one workflow predominates
- Examples:
  - At the beginning of the life cycle
    - » The requirements workflow predominates
  - At the end of the life cycle
    - » The implementation and test workflows predominate
- Planning and documentation activities are performed throughout the life cycle

2.7 Risks and Other Aspects of Iter. and Increm.

Slide 2.21

- We can consider the project as a whole as a set of mini projects (increments)
- Each mini project extends the
  - Requirements artifacts
  - Analysis artifacts
  - Design artifacts
  - Implementation artifacts
  - Testing artifacts
- The final set of artifacts is the complete product

Risks and Other Aspects of Iter. and Increm. (contd)

Slide 2.22

- During each mini project we
  - Extend the artifacts (incrementation);
  - Check the artifacts (test workflow); and
  - If necessary, change the relevant artifacts (iteration)

Risks and Other Aspects of Iter. and Increm. (contd)

Slide 2.23

- Each iteration can be viewed as a small but complete waterfall life-cycle model
- During each iteration we select a portion of the software product
- On that portion we perform the
  - Requirements phase
  - Analysis phase
  - Design phase
  - Implementation phase

Strengths of the Iterative-and-Incremental Model

Slide 2.24

- There are multiple opportunities for checking that the software product is correct
  - Every iteration incorporates the test workflow
  - Faults can be detected and corrected early
- The robustness of the architecture can be determined early in the life cycle
  - *Architecture* — the various component modules and how they fit together
  - *Robustness* — the property of being able to handle extensions and changes without falling apart

Strengths of the Iterative-and-Incremental Model (contd)

- We can *mitigate* (resolve) risks early
  - Risks are invariably involved in software development and maintenance
- We have a working version of the software product from the start
  - The client and users can experiment with this version to determine what changes are needed
- Variation: Deliver partial versions to smooth the introduction of the new product in the client organization

Strengths of the Iterative-and-Incremental Model (contd)

- There is empirical evidence that the life-cycle model works
- The CHAOS reports of the Standish Group (see overleaf) show that the percentage of successful products increases

Strengths of the Iterative-and-Incremental Model (contd)

- CHAOS reports from 1994 to 2004

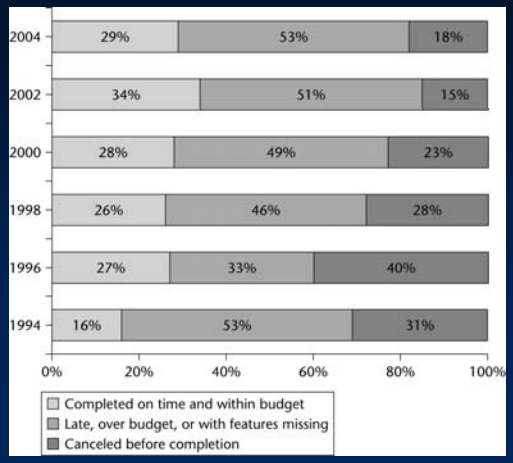

 Figure 2.7

Strengths of the Iterative-and-Incremental Model (contd)

- Reasons given for the decrease in successful projects in 2004 include:
  - More large projects in 2004 than in 2002
  - Use of the waterfall model
  - Lack of user involvement
  - Lack of support from senior executives

2.9 Other Life-Cycle Models

Slide 2.29

- The following life-cycle models are presented and compared:
  - Code-and-fix life-cycle model
  - Waterfall life-cycle model
  - Rapid prototyping life-cycle model
  - Open-source life-cycle model
  - Agile processes
  - Synchronize-and-stabilize life-cycle model
  - Spiral life-cycle model

2.9.1 Code-and-Fix Model

Slide 2.30

- No design
- No specifications
  - Maintenance nightmare

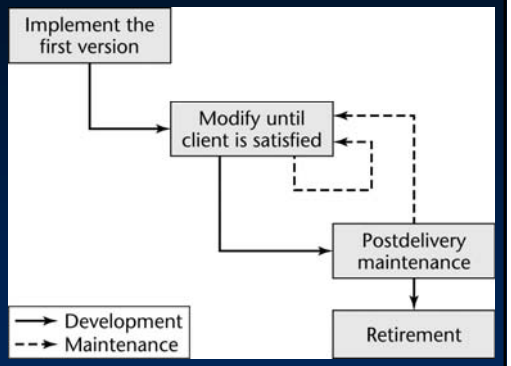

Figure 2.8

Code-and-Fix Model (contd)

Slide 2.31

- The easiest way to develop software
- The most expensive way

2.9.2 Waterfall Model (Royce 1970)

Slide 2.32

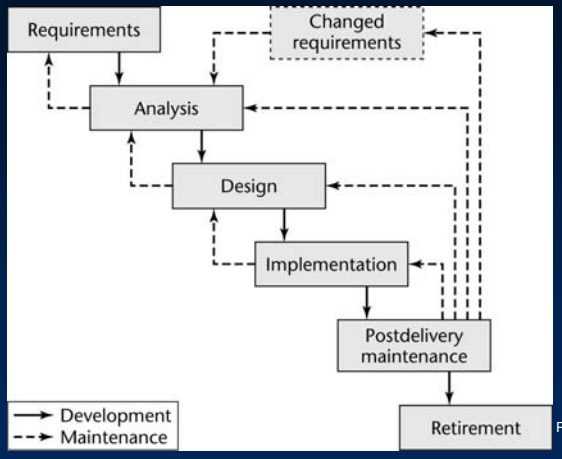

Figure 2.9

2.9.2 Waterfall Model (contd)

Slide 2.33

- Characterized by
  - Feedback loops
  - Documentation-driven
- Advantages
  - Documentation
  - Maintenance is easier
- Disadvantages
  - Specification document
    - » Hard to understand for clients (e.g., Z language)
  - Don't get to see the product till the end
    - » "I know this is what I asked for, but it isn't what I meant

2.9.3 Rapid Prototyping Model

Slide 2.34

- Linear model
- "Rapid"

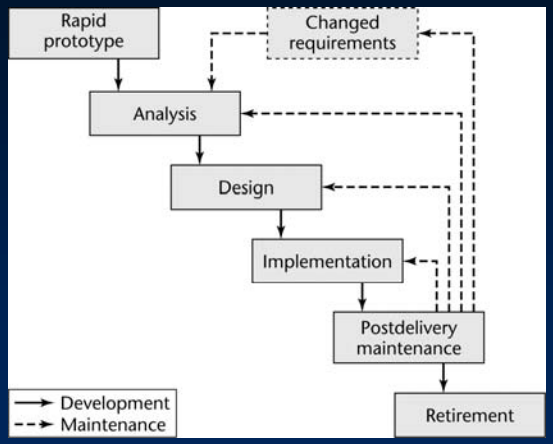

Figure 2.10

2.9.4 Open-Source Life-Cycle Model

Slide 2.35

- Two informal phases
- First, one individual builds an initial version
  - Made available via the Internet (e.g., SourceForge.net)
- Then, if there is sufficient interest in the project
  - The initial version is widely downloaded
  - Users become co-developers
  - The product is extended
- Key point: Individuals generally work voluntarily on an open-source project in their spare time

The Activities of the Second Informal Phase

Slide 2.36

- Reporting and correcting defects
  - Corrective maintenance
- Adding additional functionality
  - Perfective maintenance
- Porting the program to a new environment
  - Adaptive maintenance
- The second informal phase consists *solely* of postdelivery maintenance
  - The word "co-developers" on the previous slide should rather be "co-maintainers"

Open-Source Life-Cycle Model (contd)

Slide 2.37

- Postdelivery maintenance life-cycle model

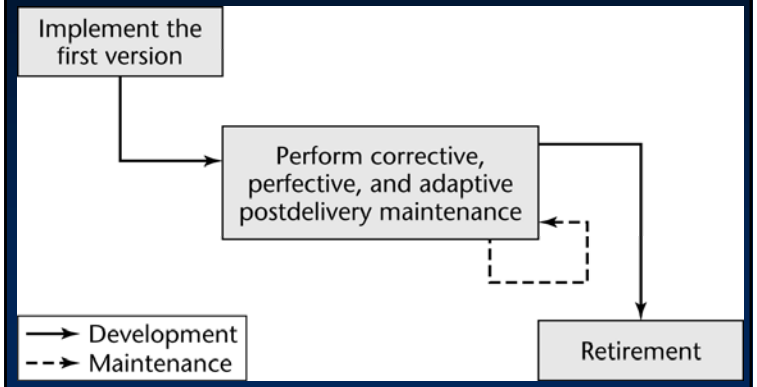

Figure 2.11

Open-Source Life-Cycle Model (contd)

Slide 2.38

- Closed-source software is maintained and tested by employees
  - Users can submit failure reports but never fault reports (the source code is not available)
- Open-source software is generally maintained by unpaid volunteers
  - Users are strongly encouraged to submit defect reports, both failure reports and fault reports

Open-Source Life-Cycle Model (contd)

Slide 2.39

- Core group
  - Small number of dedicated maintainers with the inclination, the time, and the necessary skills to submit fault reports (“fixes”)
  - They take responsibility for managing the project
  - They have the authority to install fixes
- Peripheral group
  - Users who choose to submit defect reports from time to time

Open-Source Life-Cycle Model (contd)

Slide 2.40

- New versions of closed-source software are typically released roughly once a year
  - After careful testing by the SQA group
- The core group releases a new version of an open-source product as soon as it is ready
  - Perhaps a month or even a day after the previous version was released
  - The core group performs minimal testing
  - Extensive testing is performed by the members of the peripheral group in the course of utilizing the software
  - “Release early and often”

Open-Source Life-Cycle Model (contd)

Slide 2.41

- An initial working version is produced when using
  - The rapid-prototyping model;
  - The code-and-fix model; and
  - The open-source life-cycle model
- Then:
  - Rapid-prototyping model
    - » The initial version is discarded
  - Code-and-fix model and open-source life-cycle model
    - » The initial version becomes the target product
    - » Successful projects get all the attention, but half of the 175,000 projects on SourceForge.org have 0 download.
    - » Open source software is not customized.

2.9.5 Agile Processes

Slide 2.42

- Extreme Programming (later lecture)

2.9.6 Synchronize-and Stabilize Model

Slide 2.43

- Microsoft's life-cycle model
- Requirements analysis — interview potential customers
- Draw up specifications
- Divide project into 3 or 4 builds
- Each build is carried out by small teams working in parallel

Synchronize-and Stabilize Model (contd)

Slide 2.44

- At the end of the day — *synchronize* (test and debug)
- At the end of the build — *stabilize* (freeze the build)
- Components always work together
  - Faults can be fixed in a early stage
  - Get early insights into the operation of the product

2.9.7 Spiral Model

Slide 2.45

- Simplified form
  - Rapid prototyping model plus risk analysis preceding each phase

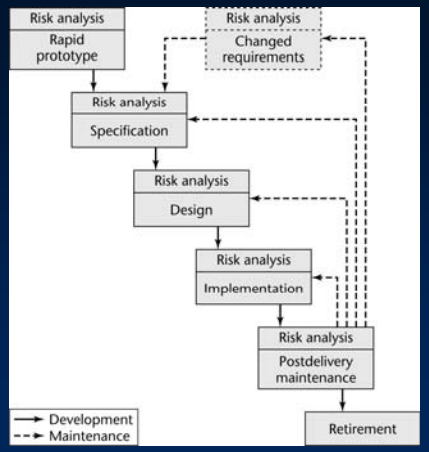

Figure 2.12

A Key Point of the Spiral Model

Slide 2.46

- If all risks cannot be mitigated, the project is immediately terminated

Full Spiral Model

Slide 2.47

- Precede each phase by
  - Alternatives
  - Risk analysis
- Follow each phase by
  - Evaluation
  - Planning of the next phase
- Radial dimension: cumulative cost to date
- Angular dimension: progress through the spiral

Full Spiral Model (contd)

Slide 2.48

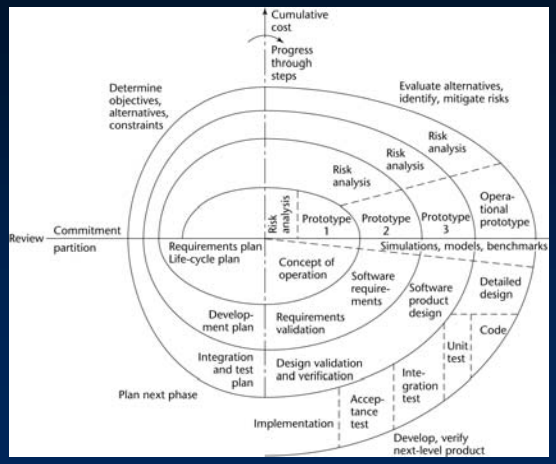

Figure 2.13

Analysis of the Spiral Model

Slide 2.49

- Strengths
  - Alternative analysis allows the reuse of existing software
  - It is easy to judge how much to test
    - » Terms it as risks and perform risks analysis
- Weaknesses
  - For large-scale software only
    - » Intensive risk analysis
  - For internal (in-house) software only
    - » Easy termination
  - Discrete phases