Object-Oriented and Classical Software Engineering

SOFTWARE LIFE-CYCLE MODELS

Overview

- Software development in theory
- Winburg mini case study
- Lessons of the Winburg mini case study
- Teal tractors mini case study
- Iteration and incrementation
- Winburg mini case study revisited
- Risks and other aspects of iteration and incrementation
- Managing iteration and incrementation
- Other life-cycle models
- Comparison of life-cycle models

2.1 Software Development in Theory

- Ideally, software is developed as described in Chapter 1
  - Linear
  - Starting from scratch

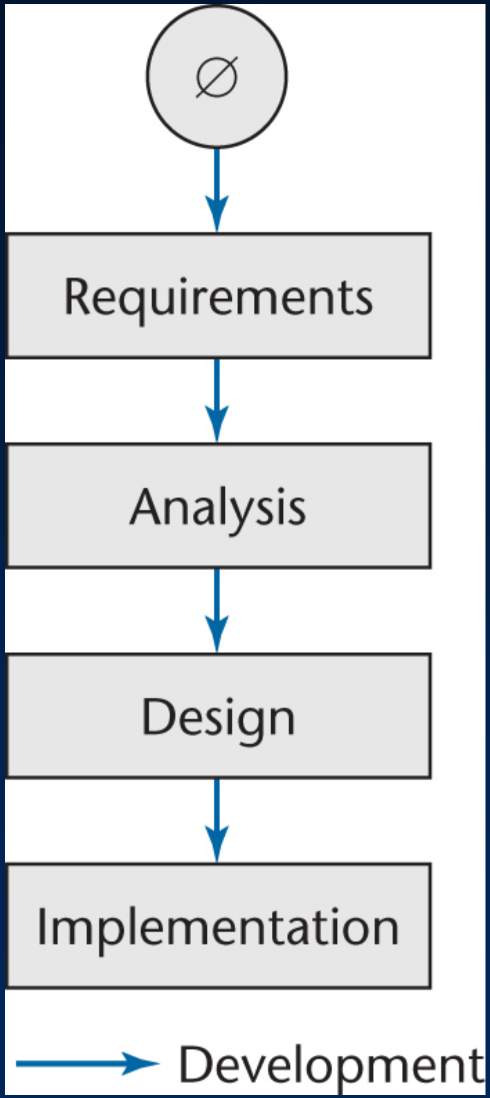

Figure 2.1

Software Development in Practice

- 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

- **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

Evolution-Tree Model

- Winburg Mini Case Study

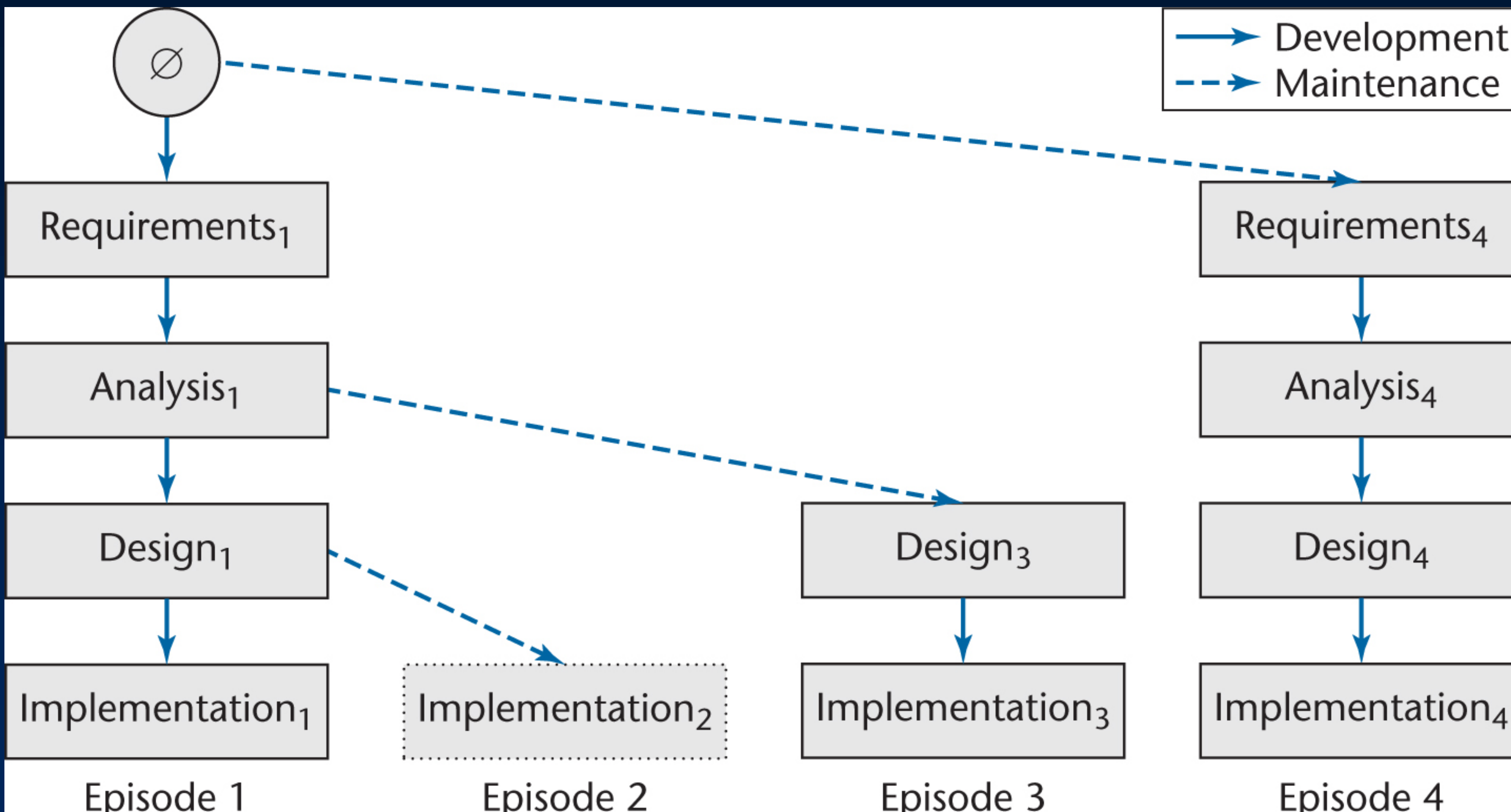

Figure 2.2

Waterfall Model

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

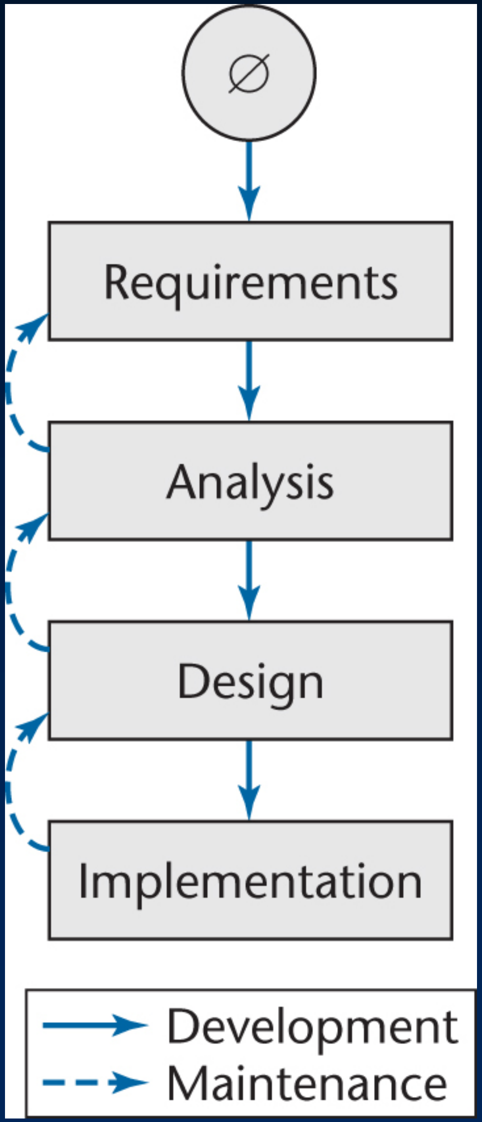

Figure 2.3

Return to the Evolution-Tree Model

- The explicit order of events is shown
- At the end of each episode
  - We have a *baseline*, a complete set of *artifacts* (constituent components)
- Example:
  - Baseline at the end of Episode 3:
    - » Requirements$_1$, Analysis$_1$, Design$_3$, Implementation$_3$

2.3 Lessons of the Winburg Mini Case Study

- 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

- 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)

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

Moving Target Problem

- A change in the requirements while the software product is being developed

- Even if the reasons for the change are good, the software product can be adversely impacted
  - Dependencies will be induced

Moving Target Problem (contd)

- Any change made to a software product can potentially cause a *regression fault*
  - A fault in an apparently unrelated part of the software

- If there are too many changes
  - The entire product may have to be redesigned and reimplemented

Moving Target Problem (contd)

- Change is inevitable
  - Growing companies are always going to change
  - If the individual calling for changes has sufficient clout, nothing can be done about it

- There is no solution to the moving target problem

2.5 Iteration and Incrementation

- In real life, we cannot speak about "the analysis phase"
  - Instead, the operations of the analysis phase are spread out over the life cycle

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

Miller's Law

- 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)

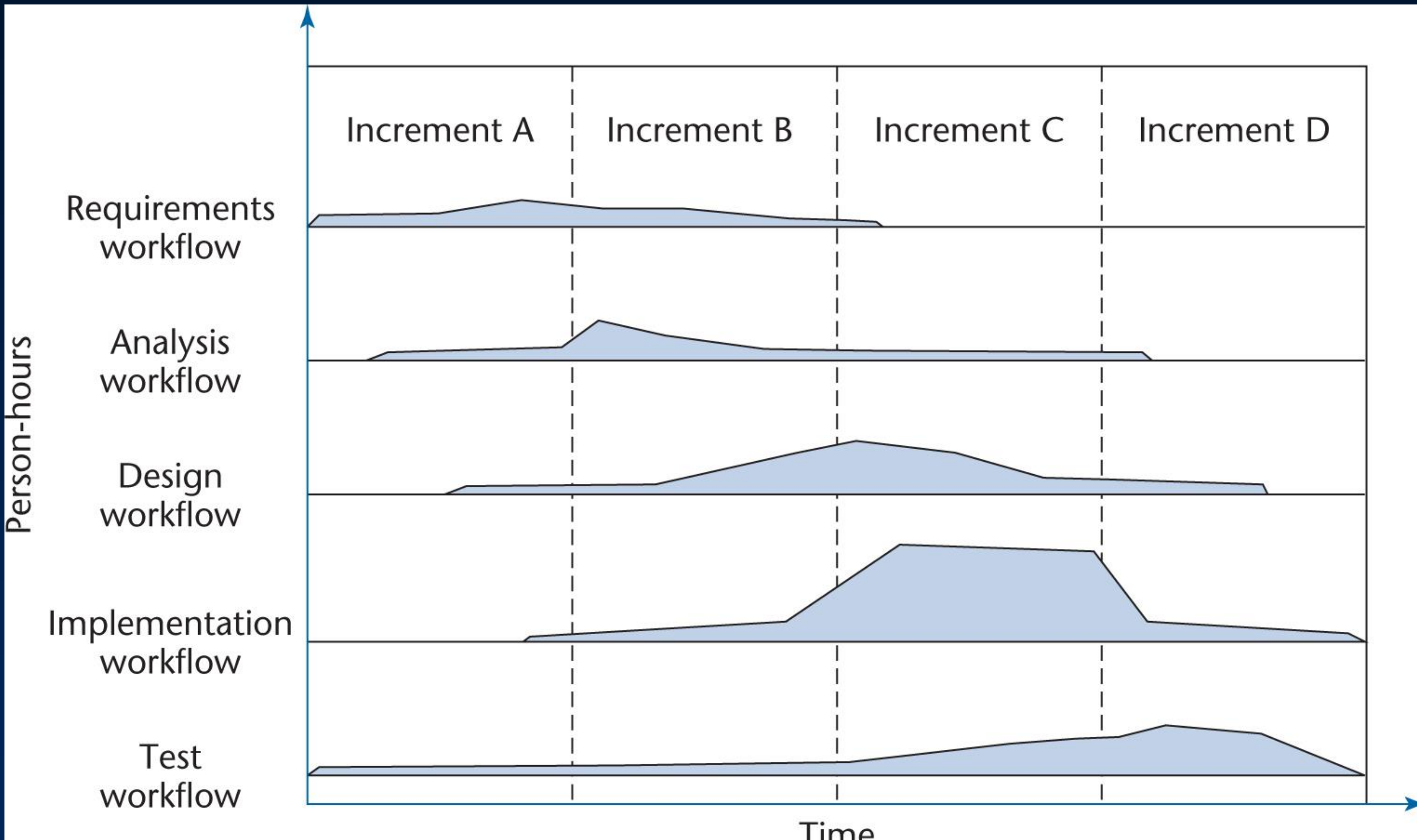

Figure 2.4

Iteration and Incrementation (contd)

- Iteration and incrementation are used in conjunction with one another
  - There is no single "requirements phase" or "design phase"
  - Instead, there are multiple instances of each phase

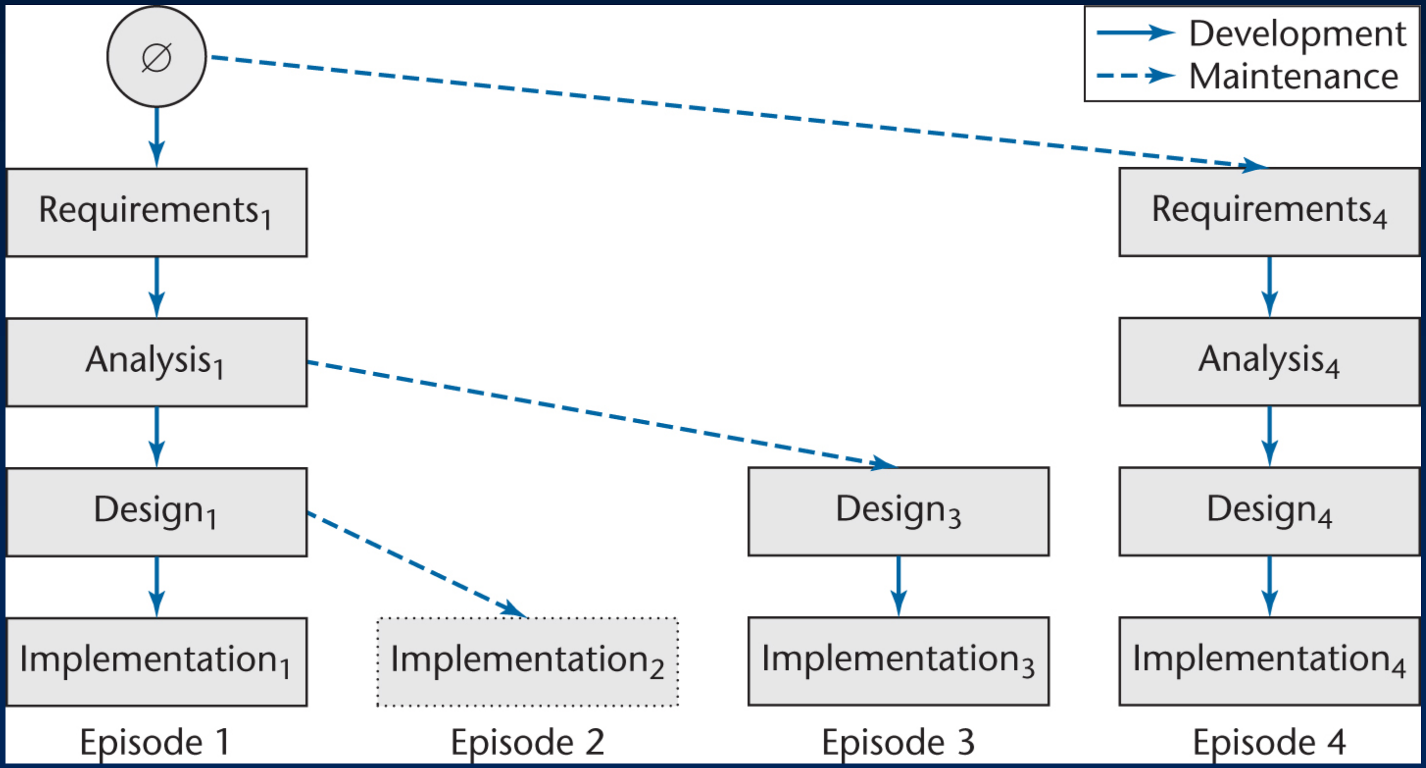

Figure 2.2 (again)

Iteration and Incrementation (contd)

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

Classical Phases versus Workflows

- 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

- 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

Iteration and Incrementation (contd)

- Iteration is performed during each incrementation

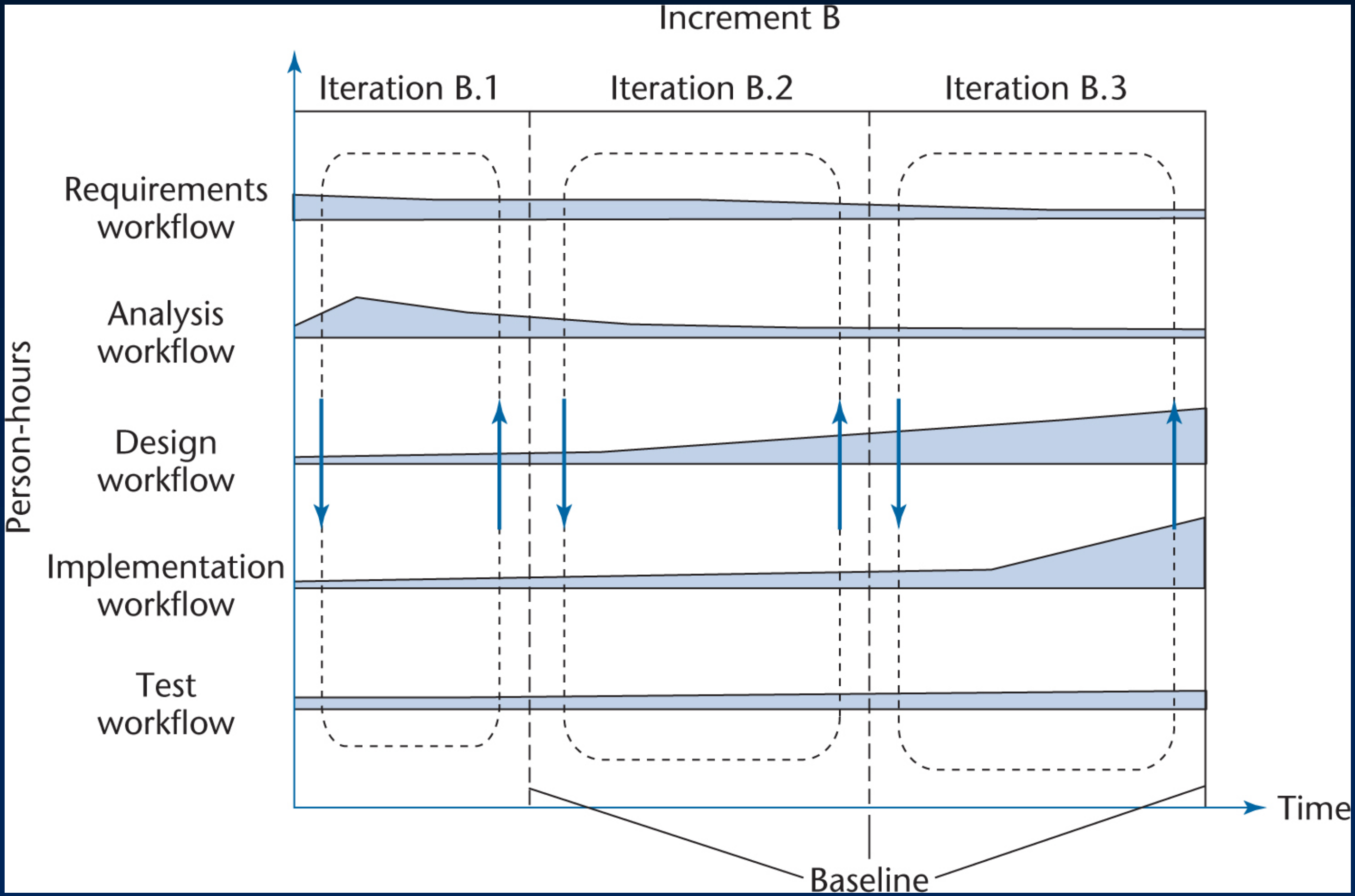

Figure 2.5

Iteration and Incrementation (contd)

- Again, the number of iterations will vary—it is not always three

2.6 The Winburg Mini Case Study Revisited

- Consider the next slide
- The evolution-tree model has been superimposed on the iterative-and-incremental life-cycle model
- The test workflow has been omitted — the evolution-tree model assumes continuous testing

The Winburg Mini Case Study Revisited

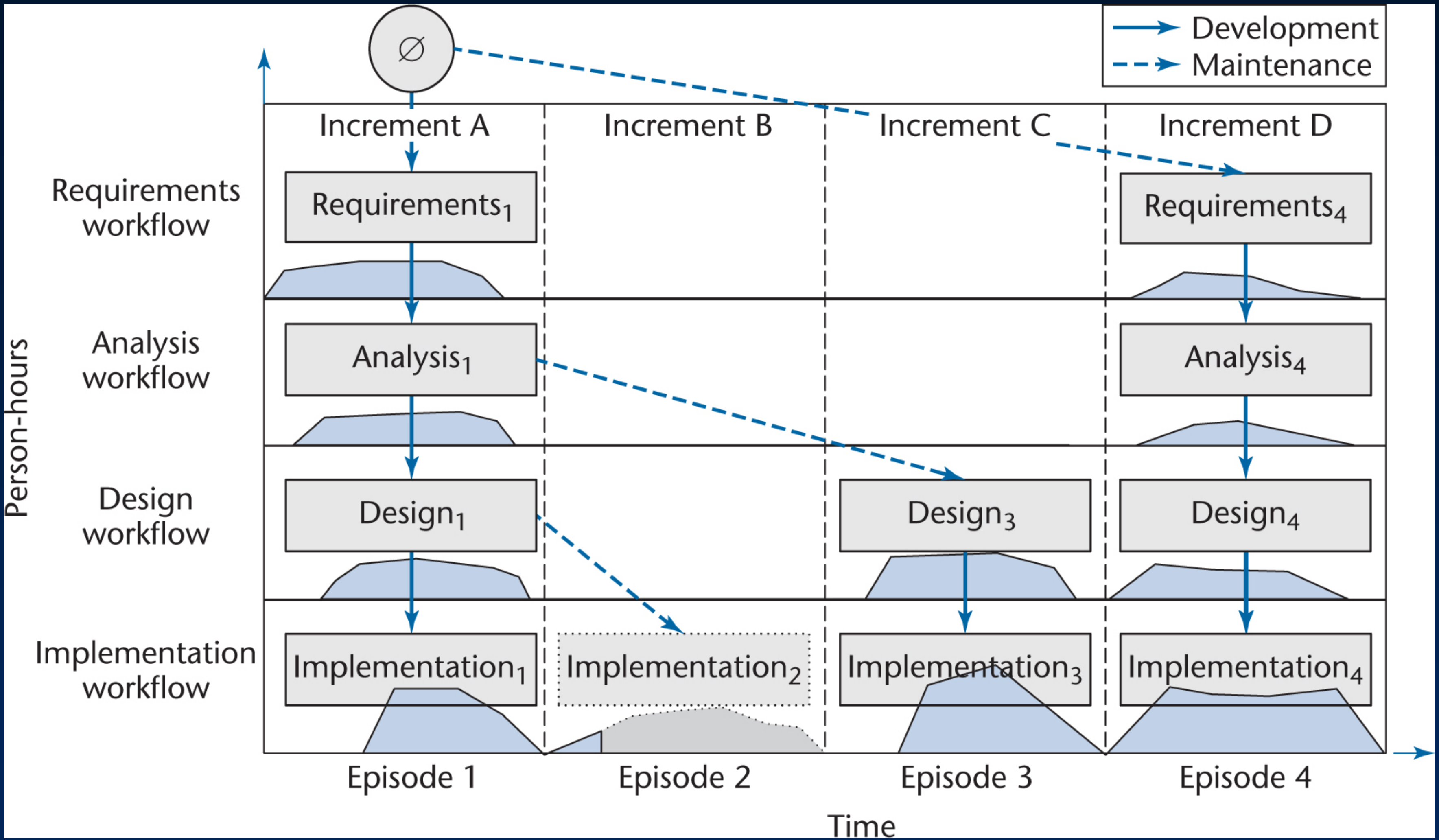

Figure 2.6

More on Incrementation (contd)

- Each episode corresponds to an increment
- Not every increment includes every workflow
- Increment B was not completed
- Dashed lines denote maintenance
  - Episodes 2, 3: Corrective maintenance
  - Episode 4: Perfective maintenance

2.7 Risks and Other Aspects of Iter. and Increm.

- 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)

- 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)

- 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
  – Classical requirements phase
  – Classical analysis phase
  – Classical design phase
  – Classical implementation phase

Strengths of the Iterative-and-Incremental Model

- 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 2006

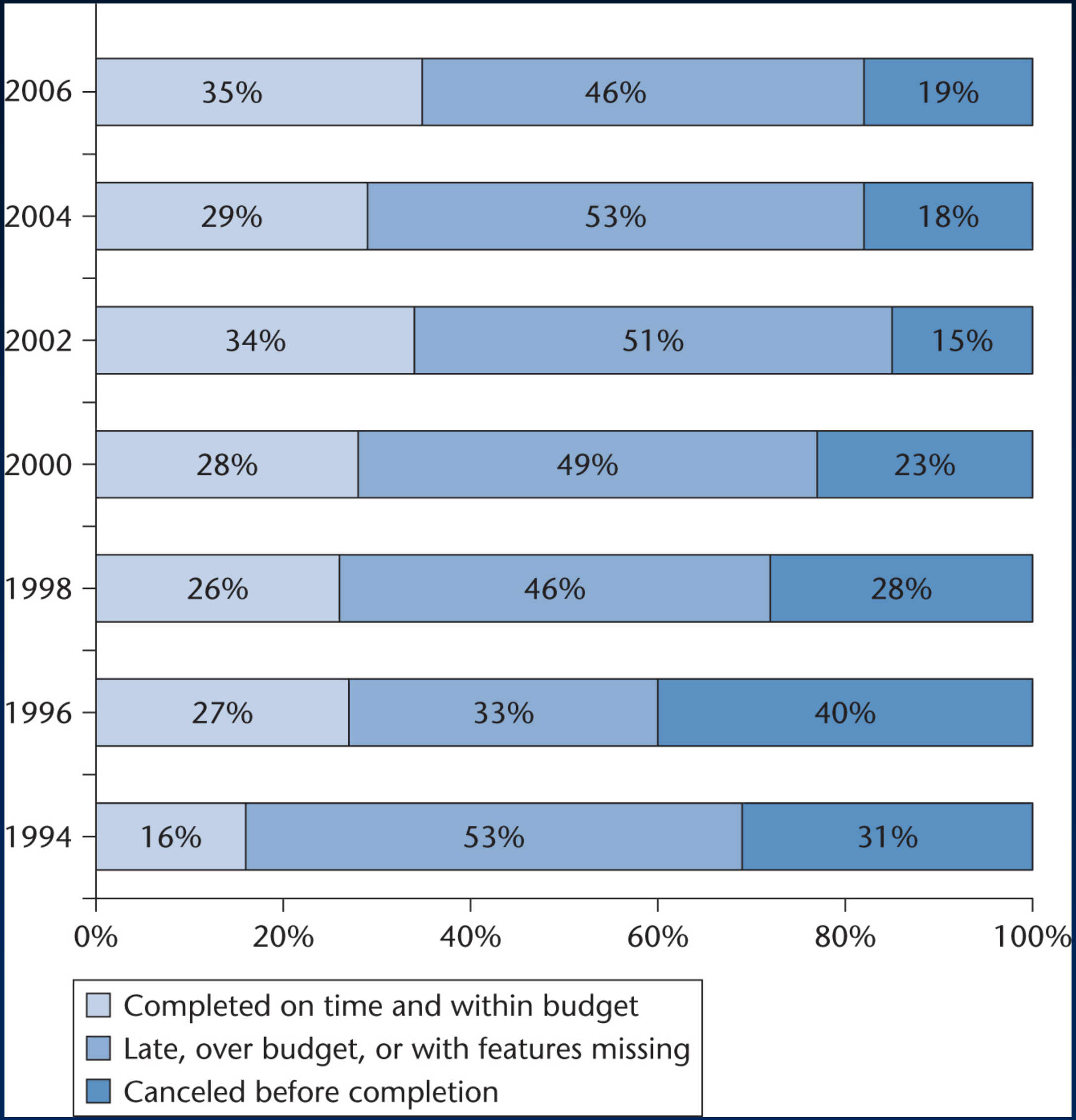

Figure 2.7

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

2.8 Managing Iteration and Incrementation

- The iterative-and-incremental life-cycle model is as regimented as the waterfall model …
- … because the iterative-and-incremental life-cycle model *is* the waterfall model, applied successively
- Each increment is a waterfall mini project

2.9 Other Life-Cycle Models

- 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

- No design
- No specifications
  - Maintenance nightmare

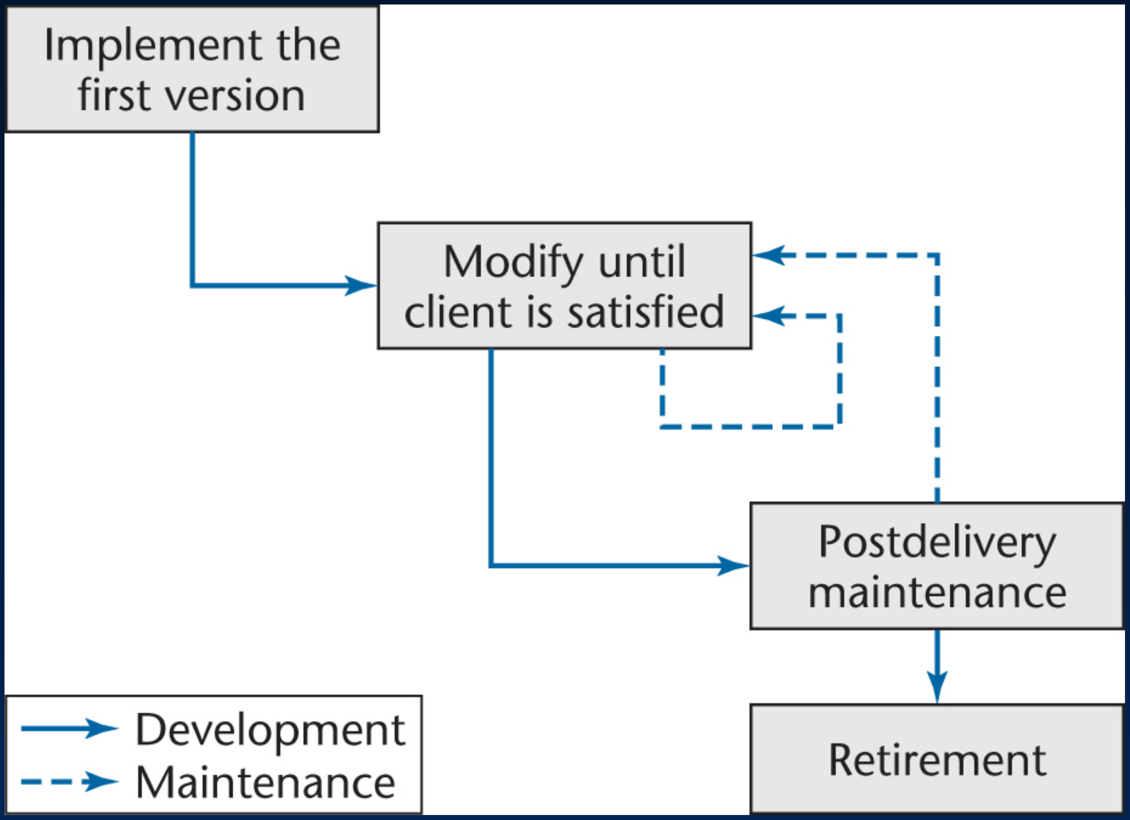

Figure 2.8

Code-and-Fix Model (contd)

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

2.9.2 Waterfall Model

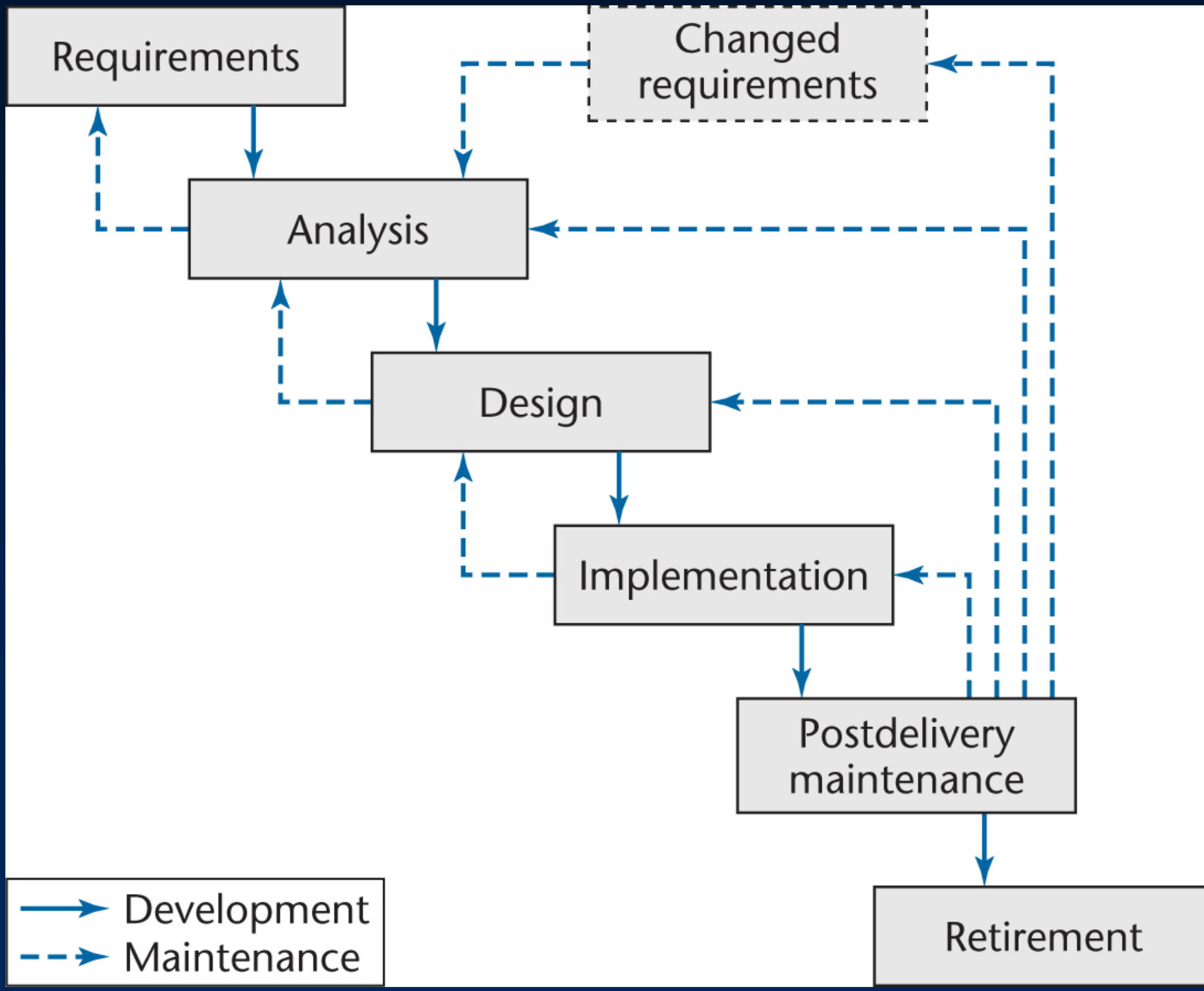

Figure 2.9

2.9.2 Waterfall Model (contd)

- Characterized by
  - Feedback loops
  - Documentation-driven

- Advantages
  - Documentation
  - Maintenance is easier

- Disadvantages
  - Specification document
    - » Joe and Jane Johnson
    - » Mark Marberry

2.9.3 Rapid Prototyping Model

- Linear model
- "Rapid"

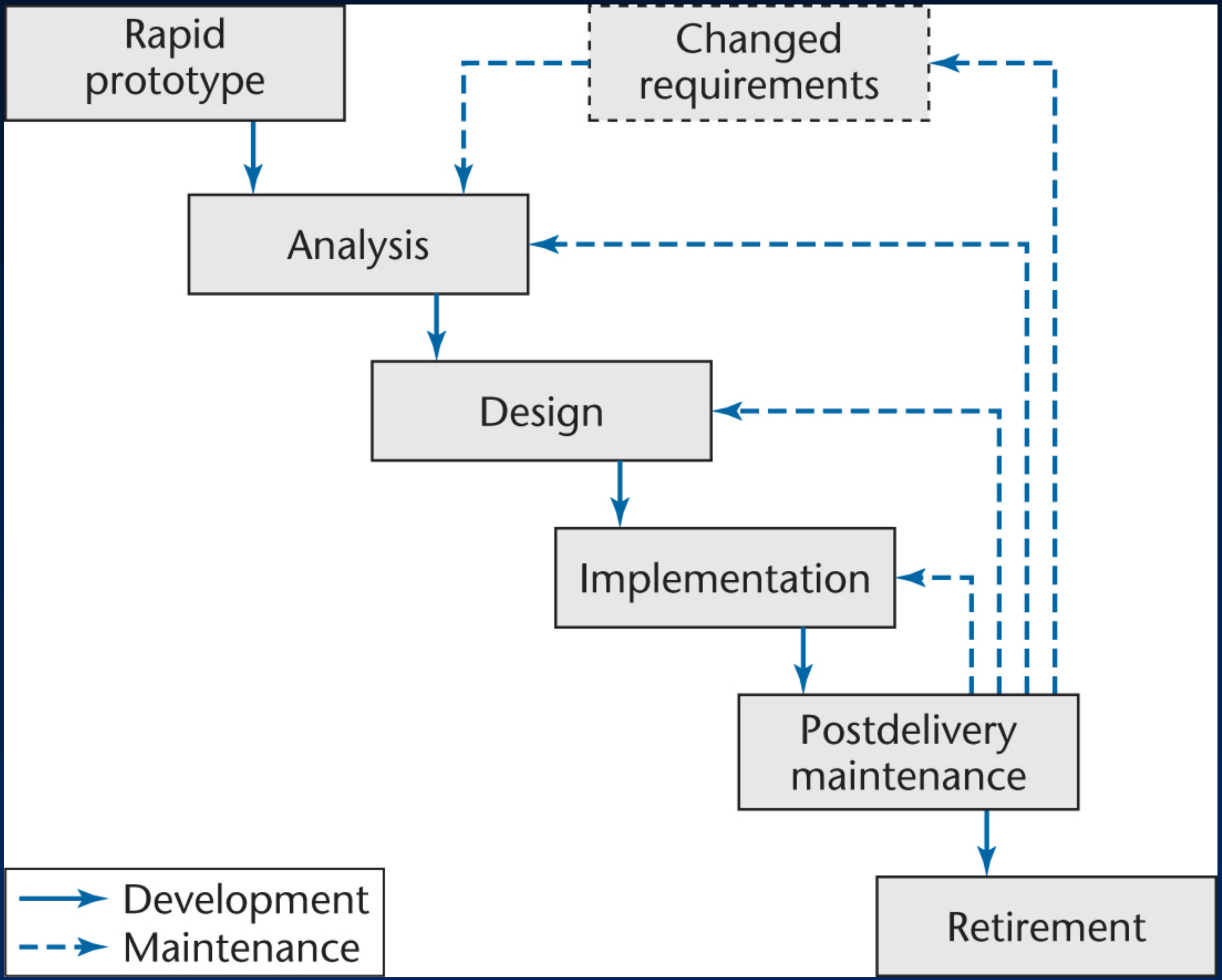

Figure 2.10

2.9.4 Open-Source Life-Cycle Model

- 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

- 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)

- Postdelivery maintenance life-cycle model

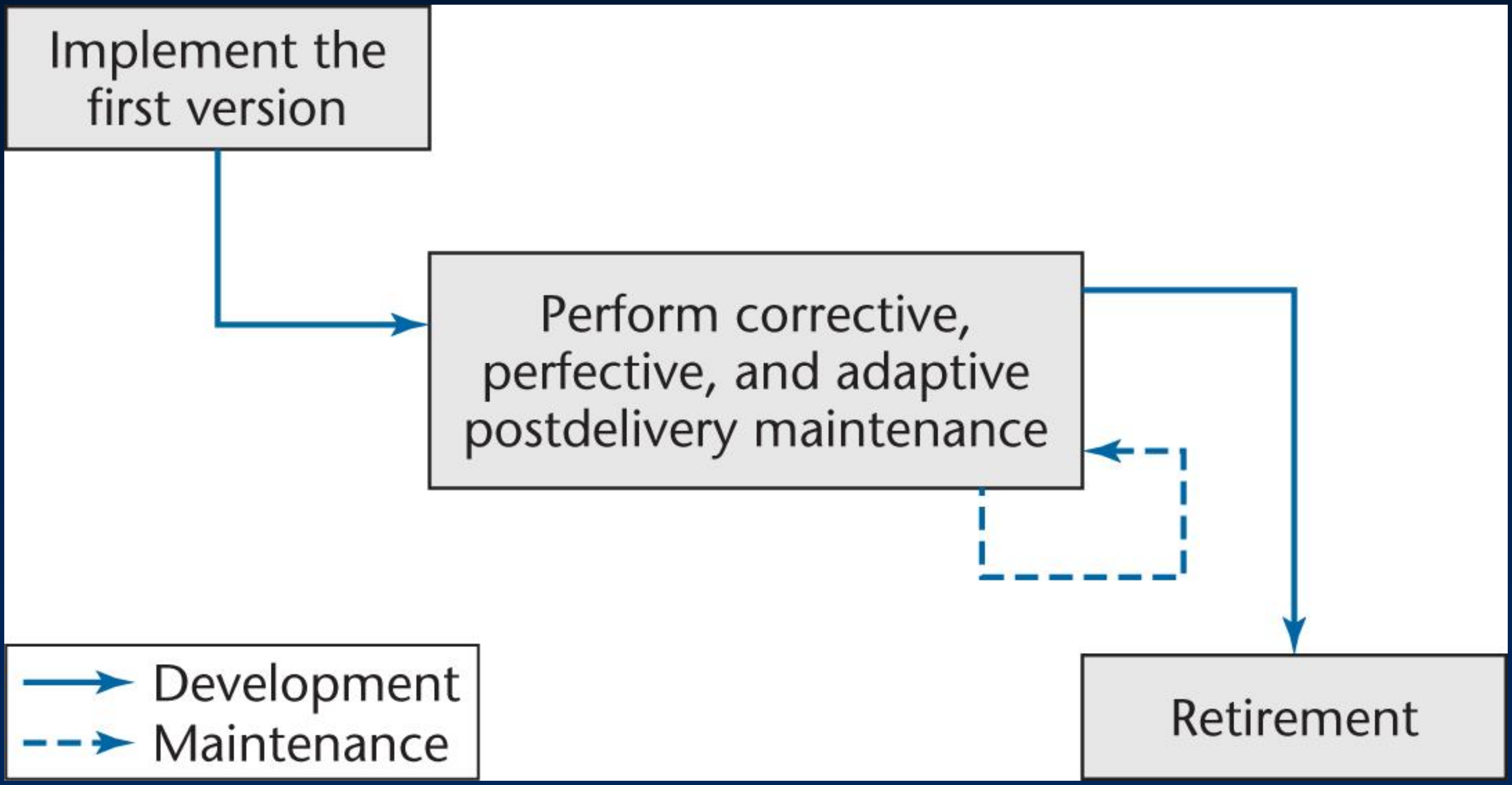

Figure 2.11

Open-Source Life-Cycle Model (contd)

- 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)

- 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)

- 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)

- 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

Open-Source Life-Cycle Model (contd)

- Consequently, in an open-source project, there are generally no specifications and no design

- How have some open-source projects been so successful without specifications or designs?

Open-Source Life-Cycle Model (contd)

- Open-source software production has attracted some of the world’s finest software experts
  - They can function effectively without specifications or designs

- However, eventually a point will be reached when the open-source product is no longer maintainable

Open-Source Life-Cycle Model (contd)

- The open-source life-cycle model is restricted in its applicability

- It can be extremely successful for infrastructure projects, such as
  - Operating systems (Linux, OpenBSD, Mach, Darwin)
  - Web browsers (Firefox, Netscape)
  - Compilers (gcc)
  - Web servers (Apache)
  - Database management systems (MySQL)

Open-Source Life-Cycle Model (contd)

- There cannot be open-source development of a software product to be used in just one commercial organization
  - Members of both the core group and the periphery are invariably users of the software being developed

- The open-source life-cycle model is inapplicable unless the target product is viewed by a wide range of users as useful to them

Open-Source Life-Cycle Model (contd)

- About half of the open-source projects on the Web have not attracted a team to work on the project
- Even where work has started, the overwhelming preponderance will never be completed
- But when the open-source model has worked, it has sometimes been incredibly successful
  - The open-source products previously listed have been utilized on a regular basis by millions of users

2.9.5 Agile Processes

- Somewhat controversial new approach

- *Stories* (features client wants)
  - Estimate duration and cost of each story
  - Select stories for next build
  - Each build is divided into tasks
  - Test cases for a task are drawn up first

- Pair programming

- Continuous integration of tasks

Unusual Features of XP

- The computers are put in the center of a large room lined with cubicles
- A client representative is always present
- Software professionals cannot work overtime for 2 successive weeks
- No specialization
- *Refactoring* (design modification)

Acronyms of Extreme Programming

- YAGNI (you aren’t gonna need it)

- DTSTTCPW (do the simplest thing that could possibly work)

- A principle of XP is to minimize the number of features
  - There is no need to build a product that does any more than what the client actually needs

Agile Processes

- XP is one of a number of new paradigms collectively referred to as *agile processes*
- Seventeen software developers (later dubbed the “Agile Alliance”) met at a Utah ski resort for two days in February 2001 and produced the *Manifesto for Agile Software Development*
- The Agile Alliance did not prescribe a specific life-cycle model
  - Instead, they laid out a group of underlying principles

Agile Processes

- Agile processes are a collection of new paradigms characterized by
  - Less emphasis on analysis and design
  - Earlier implementation (working software is considered more important than documentation)
  - Responsiveness to change
  - Close collaboration with the client

Agile Processes (contd)

- A principle in the *Manifesto* is
  - Deliver working software frequently
  - Ideally every 2 or 3 weeks

- One way of achieving this is to use *timeboxing*
  - Used for many years as a time-management technique

- A specific amount of time is set aside for a task
  - Typically 3 weeks for each iteration
  - The team members then do the best job they can during that time

Agile Processes (contd)

- It gives the client confidence to know that a new version with additional functionality will arrive every 3 weeks

- The developers know that they will have 3 weeks (but no more) to deliver a new iteration
  - Without client interference of any kind

- If it is impossible to complete the entire task in the timebox, the work may be reduced (“descoped”)
  - Agile processes demand fixed time, not fixed features

Agile Processes (contd)

- Another common feature of agile processes is *stand-up meetings*
  – Short meetings held at a regular time each day
  – Attendance is required

- Participants stand in a circle
  – They do not sit around a table
  – To ensure the meeting lasts no more than 15 minutes

Agile Processes (contd)

- At a stand-up meeting, each team member in turn answers five questions:
  - What have I done since yesterday's meeting?
  - What am I working on today?
  - What problems are preventing me from achieving this?
  - What have we forgotten?
  - What did I learn that I would like to share with the team?

Agile Processes (contd)

- The aim of a stand-up meeting is
  - To raise problems
  - Not solve them

- Solutions are found at follow-up meetings, preferably held directly after the stand-up meeting

Agile Processes (contd)

- Stand-up meetings and timeboxing are both
  - Successful management techniques
  - Now utilized within the context of agile processes

- Both techniques are instances of two basic principles that underlie all agile methods:
  - Communication; and
  - Satisfying the client's needs as quickly as possible

Evaluating Agile Processes

- Agile processes have had some successes with small-scale software development
  - However, medium- and large-scale software development are completely different
- The key decider: the impact of agile processes on postdelivery maintenance
  - Refactoring is an essential component of agile processes
  - Refactoring continues during maintenance
  - Will refactoring increase the cost of post-delivery maintenance, as indicated by preliminary research?

Evaluating Agile Processes (contd)

- Agile processes are good when requirements are vague or changing

- In 2000, Williams, Kessler, Cunningham, and Jeffries showed that pair programming leads to
  - The development of higher-quality code,
  - In a shorter time,
  - With greater job satisfaction

Evaluating Agile Processes (contd)

- In 2007, Arisholm, Gallis, Dybå, and Sjøberg performed an extensive experiment
  - To evaluate pair programming within the context of software maintenance
- In 2007, Dybå et al. analyzed 15 published studies
  - Comparing the effectiveness of individual and pair programming
- Both groups came to the same conclusion
  - It depends on both the programmer's expertise and the complexity of the software product and the tasks to be solved

Evaluating Agile Processes (contd)

- The *Manifesto for Agile Software Development* claims that agile processes are superior to more disciplined processes like the Unified Process

- Skeptics respond that proponents of agile processes are little more than hackers

- However, there is a middle ground
  - It is possible to incorporate proven features of agile processes within the framework of disciplined processes

Evaluating Agile Processes (contd)

- In conclusion
  - Agile processes appear to be a useful approach to building small-scale software products when the client's requirements are vague
  - Also, some of the proven features of agile processes can be effectively utilized within the context of other life-cycle models

2.9.6 Synchronize-and Stabilize Model

- 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)

- At the end of the day — *synchronize* (test and debug)
- At the end of the build — *stabilize* (freeze the build)
- Components always work together
  - Get early insights into the operation of the product

2.9.7 Spiral Model

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

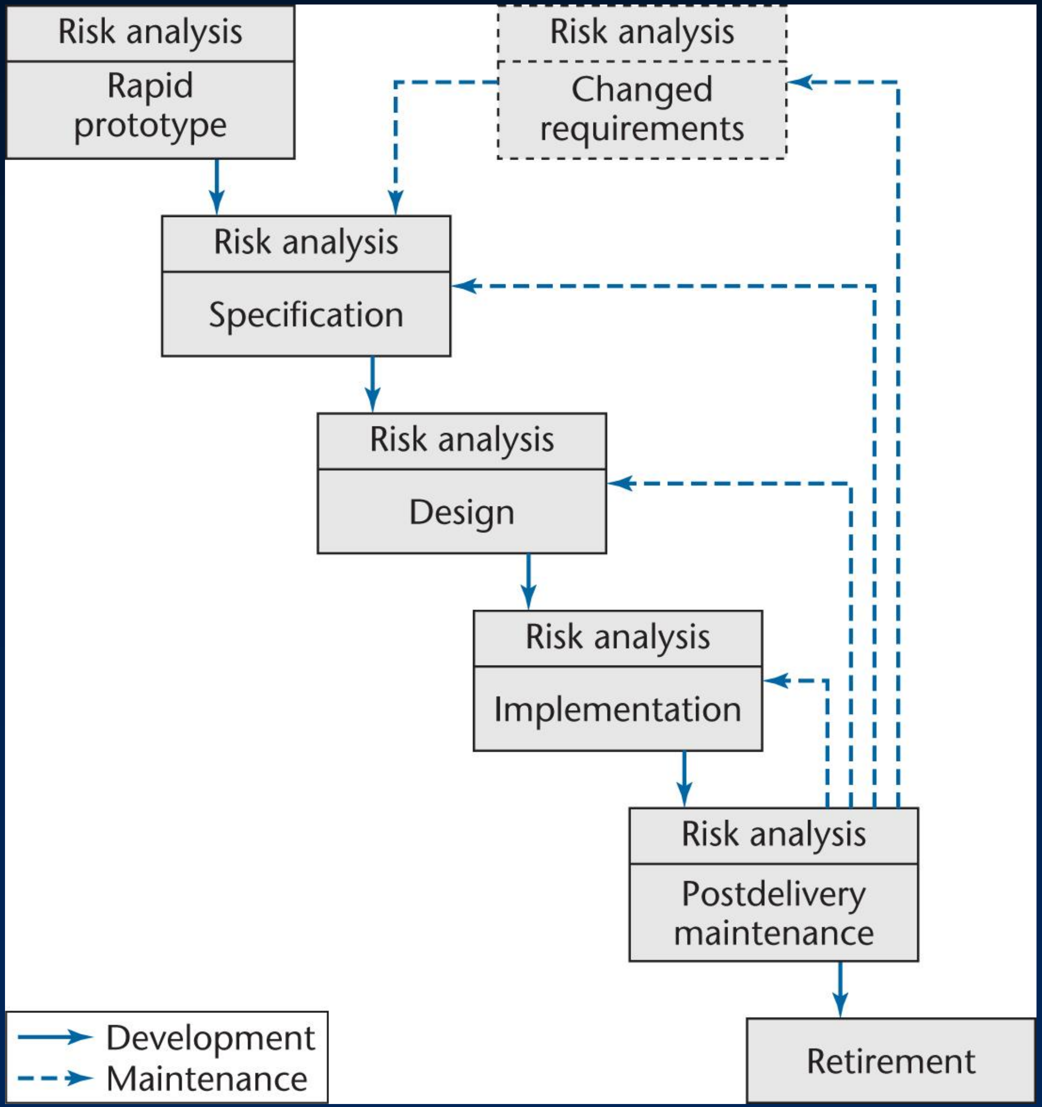

Figure 2.12

A Key Point of the Spiral Model

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

Full Spiral Model

- 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)

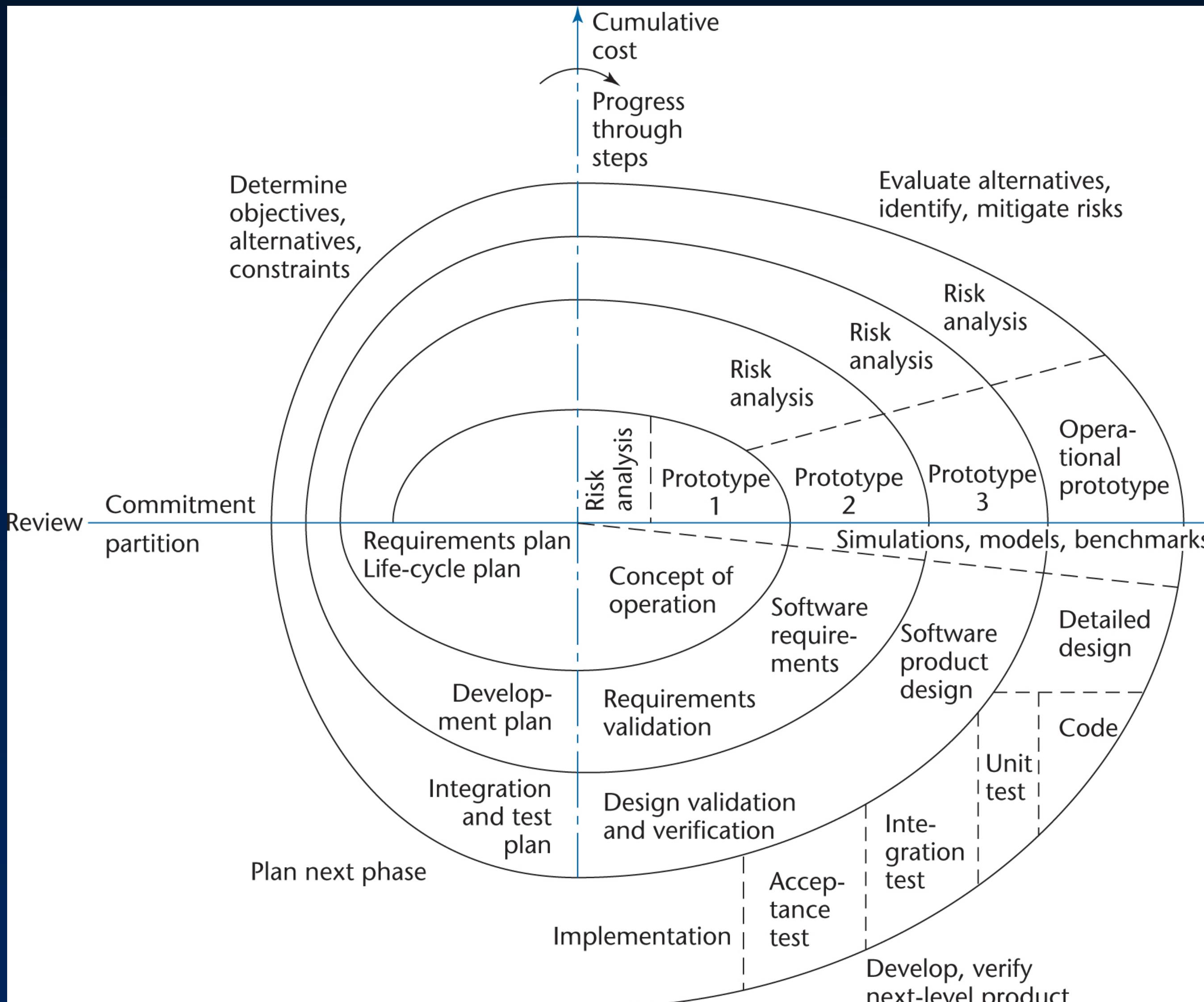

Figure 2.13

Analysis of the Spiral Model

- Strengths
  - It is easy to judge how much to test
  - No distinction is made between development and maintenance
- Weaknesses
  - For large-scale software only
  - For internal (in-house) software only

2.10 Comparison of Life-Cycle Models

- Different life-cycle models have been presented
  - Each with its own strengths and weaknesses

- Criteria for deciding on a model include:
  - The organization
  - Its management
  - The skills of the employees
  - The nature of the product

- Best suggestion
  - "Mix-and-match" life-cycle model

Comparison of Life-Cycle Models (contd)

Life-Cycle Model	Strengths	Weaknesses
Evolution-tree model (Section 2.2)	Closely models real-world software production Equivalent to the iterative-and-incremental model	
Iterative-and-incremental life-cycle model (Section 2.5)	Closely models real-world software production Underlies the Unified Process	
Code-and-fix life-cycle model (Section 2.9.1)	Fine for short programs that require no maintenance	Totally unsatisfactory for nontrivial programs
Waterfall life-cycle model (Section 2.9.2)	Disciplined approach Document driven	Delivered product may not meet client's needs
Rapid-prototyping life-cycle model (Section 2.9.3)	Ensures that the delivered product meets the client's needs	Not yet proven beyond all doubt
Open-source life-cycle model (Section 2.9.4)	Has worked extremely well in a small number of instances	Limited applicability Usually does not work
Agile processes (Section 2.9.5)	Work well when the client's requirements are vague	Appear to work on only small-scale projects
Synchronize-and-stabilize life-cycle model (Section 2.9.6)	Future users' needs are met Ensures that components can be successfully integrated	Has not been widely used other than at Microsoft
Spiral life-cycle model (Section 2.9.7)	Risk driven	Can be used for only large-scale, in-house products Developers have to be competent in risk analysis and risk resolution

Figure 2.14