Object-Oriented and Classical Software Engineering

KEY MATERIAL FROM PART A

Overview

- Software development: theory versus practice
- Iteration and incrementation
- The Unified Process
- Workflow overview
- Teams
- Cost–benefit analysis
- Metrics
- CASE
- Versions and configurations

Overview (contd)

- Testing terminology
- Execution-based and non-execution-based testing
- Modularity
- Reuse
- Software project management plan

10.1 Software Development: Theory vs. Practice

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

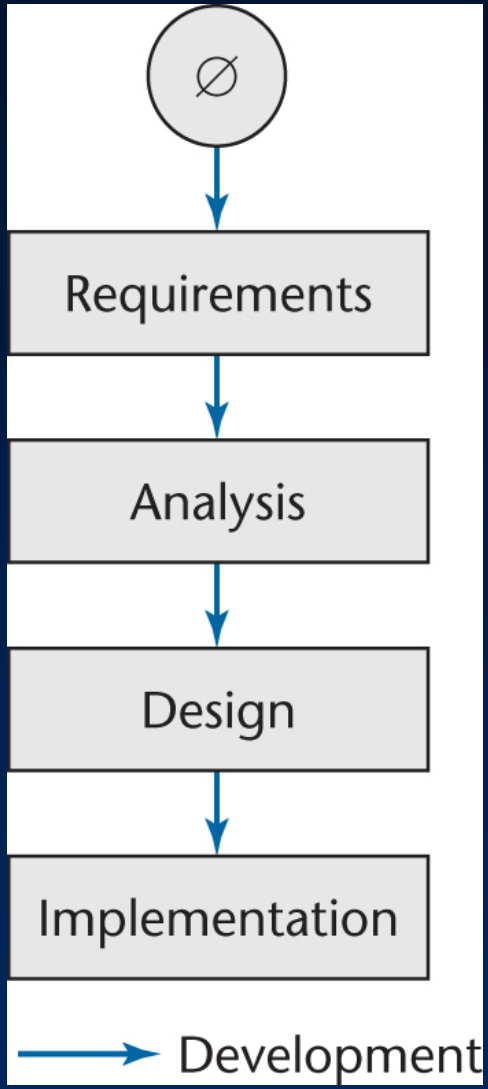

Figure 10.1

Software Development: Theory vs. Practice (contd)

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

10.2 Iteration and Incrementation

- In real life, we cannot speak about “the design phase”
  - Instead, the operations of the design phase are spread out over the life cycle
  - We keep returning to earlier workflows
- The basic software development process is *iterative*
  - Each successive version is 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 seven 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

Incrementation

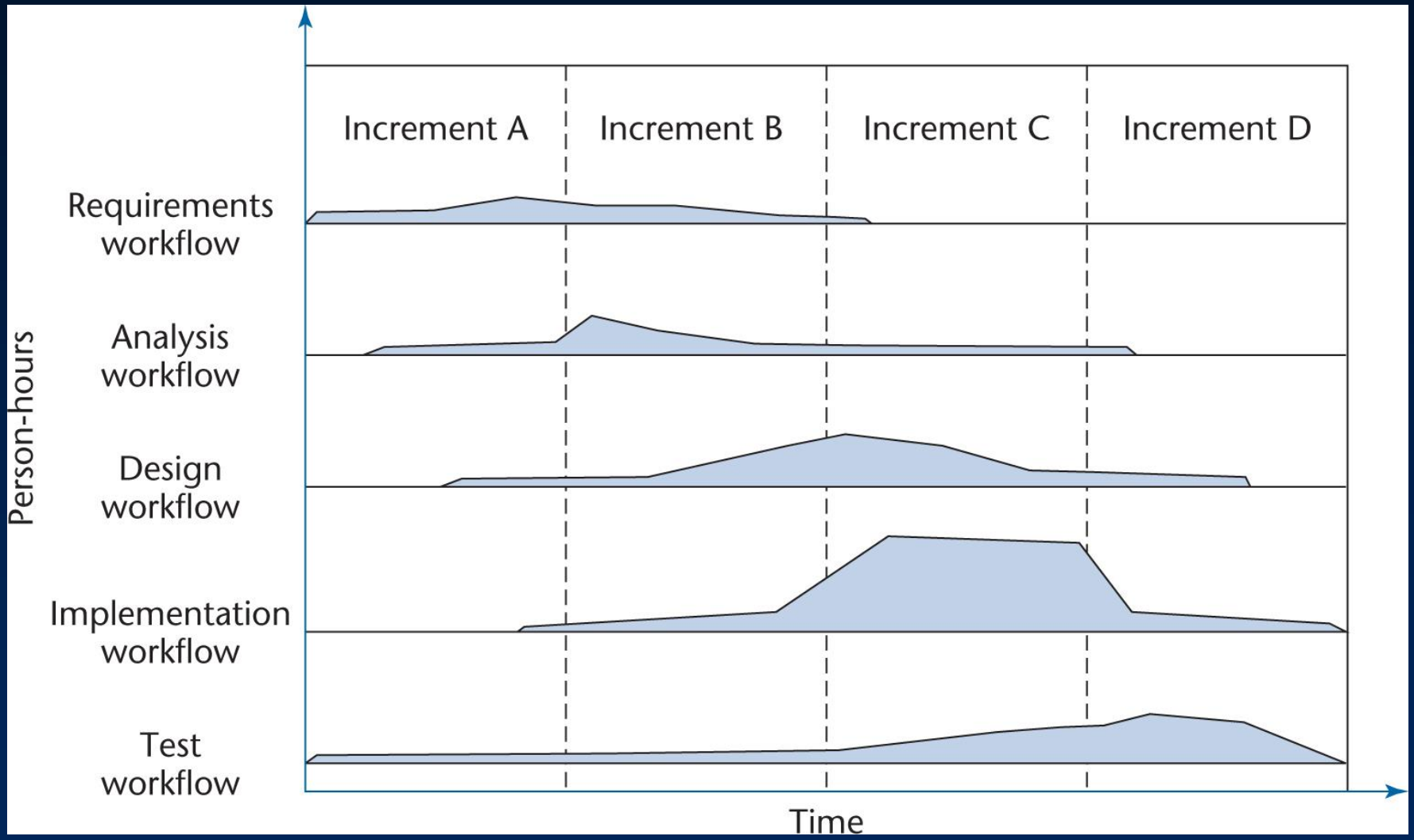

Figure 10.2

Incrementation (contd)

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

- Iteration is performed during each incrementation

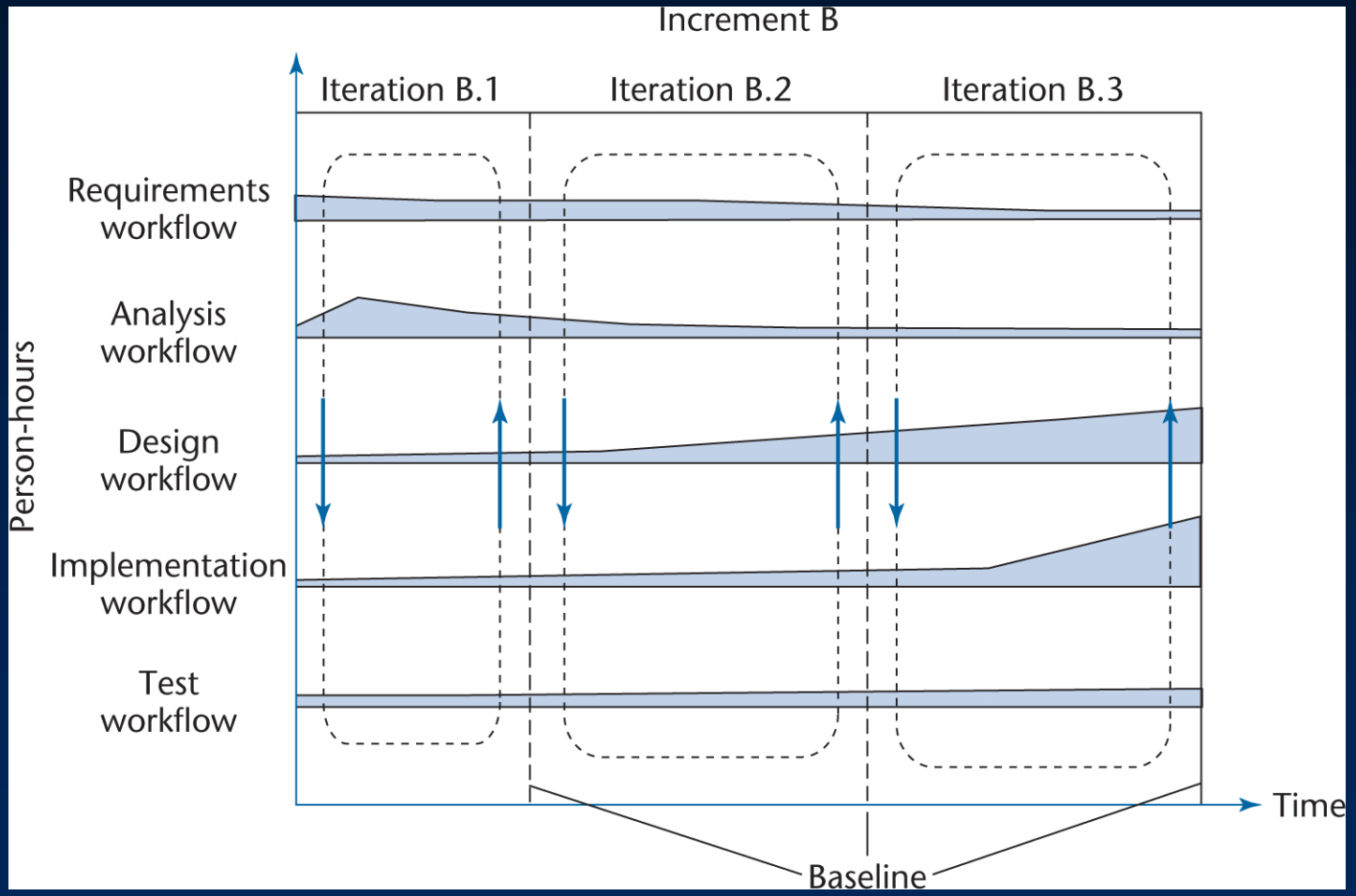

Figure 10.3

Iteration (contd)

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

10.3 The Unified Process

- The *software process* is the way we produce software

- It incorporates
  - The methodology
    - » With its underlying software life-cycle model and techniques
  - The tools we use
  - The individuals building the software

The Unified Process (contd)

- Despite its name, the *Unified Process* is not a process!
  - It's a *methodology*
- The Unified Process is an adaptable methodology
  - It has to be modified for the specific software product to be developed

The Unified Process (contd)

- The Unified Process uses a graphical language, the *Unified Modeling Language* (UML) to represent the software being developed

- A *model* is a set of UML diagrams that represent various aspects of the software product we want to develop

The Unified Process (contd)

- The object-oriented paradigm is iterative and incremental in nature
  - There is no alternative to repeated iteration and incrementation until the UML diagrams are satisfactory

10.4 Workflow Overview

- *Requirements workflow*
  - Determine exactly what the client needs
- *Analysis workflow*
  - Analyze and refine the requirements
    - » To achieve the detailed understanding of the requirements essential for developing a software product correctly and maintaining it easily

Workflow Overview (contd)

- *Design workflow*
  - Refine the artifacts of the analysis workflow until the material is in a form that can be implemented by the programmers
- *Implementation workflow*
  - Implement the target software product in the chosen implementation language(s)

Workflow Overview (contd)

- *Test workflow*
  - Testing is carried out in parallel with the other workflows, from the beginning
  - Every developer and maintainer is personally responsible for ensuring that his or her work is correct
  - Once the software professional is convinced that an artifact is correct, it is handed over to the software quality assurance group for independent testing

10.5 Teams

- Software products are usually too large (or too complex) to be built by one software engineering professional within the given time constraints
- The work has to be shared among a group of professionals organized as a *team*

Teams (contd)

- The team approach is used for each of the workflows
- In larger organizations there are specialized teams for each workflow

10.6 Cost–Benefit Analysis

- *Cost–benefit analysis* is a way of determining whether a possible course of action would be profitable
  - Compare estimated future benefits against projected future costs
- Cost–benefit analysis is a fundamental technique in deciding whether a client should computerize his or her business
  - And if so, in what way

10.7 Metrics

- We need measurements (or *metrics*) to detect problems early in the software process
  - Before they get out of hand

Metrics (contd)

- There are five fundamental metrics
  - Each must be measured and monitored for each workflow:

- 1. Size (in lines of code or, better, in a more meaningful metric)
- 2. Cost (in dollars)
- 3. Duration (in months)
- 4. Effort (in person-months)
- 5. Quality (number of faults detected)

Metrics (contd)

- Metrics serve as an early warning system for potential problems
- Management uses the fundamental metrics to identify problems
- More specialized metrics are then utilized to analyze these problems in greater depth

10.8 CASE

- *CASE* stands for *Computer-Aided Software Engineering*
  - Software that assists with software development and maintenance

CASE Taxonomy

- A CASE *tool* assists in just one aspect of the production of software
  - Examples:
    - A tool that draws UML diagrams
    - A report generator, which generates the code needed for producing a report

CASE Taxonomy (contd)

- A CASE *workbench* is a collection of tools that together support one or two activities
  - Examples:
    - » A requirements, analysis, and design workbench that incorporates a UML diagram tool and a consistency checker
    - » A project management workbench that is used in every workflow
- A CASE *environment* supports the complete software process

10.9 Versions and Configurations

- During development and maintenance, there are at least two versions of the product
  - The old version, and
  - The new version
- There will also be two or more versions of each of the component artifacts that have been changed

Versions and Configurations (contd)

- The new version of an artifact may be less correct than the previous version
- It is therefore essential to keep all versions of all artifacts
  - A CASE tool that does this is called a *version control tool*

Versions and Configurations (contd)

- A *configuration* is
  - A set of specific versions of each artifact from which a given version of the complete product is built
  - A *configuration-control tool* can handle problems caused by development and maintenance by teams
    - » In particular, when more than one person attempts to change the same artifact
- A *baseline* is a configuration of all the artifacts in the product
  - After each group of changes has been made to the artifacts, a new baseline is attained

Versions and Configurations (contd)

- If a software organization does not wish to purchase a complete configuration-control tool, then, at the very least,
  - A version-control tool must be used in conjunction with
  - A *build tool*, that is, a tool that assists in selecting the correct version of each compiled-code artifact to be linked to form a specific version of the product
    - » Build tools, such as *make*, have been incorporated into a wide variety of programming environments

10.10 Testing Terminology

- A *fault* is injected into a software product when a human makes a mistake
- A *failure* is the observed incorrect behavior of the software product as a consequence of a fault
- The *error* is the amount by which a result is incorrect
- The word *defect* is a generic term for a fault, failure, or error

Testing Terminology (contd)

- The *quality* of software is the extent to which the product satisfies its specifications
- Within a software organization, the primary task of the *software quality assurance* (SQA) group is to test that the developers' product is correct

10.11 Execution-Based and Non-Execution-Based Testing

- There are two basic forms of testing:
  - *Execution-based testing* (running test cases), and
  - *Non-execution-based testing* (carefully reading through an artifact)

Non-Execution-Based Testing

- In a *review*, a team of software professionals carefully checks through a document
  - Examples:
    - » specification document
    - » design document
    - » code artifact
- There are two types of review
  - *Walkthrough* − less formal
  - *Inspection* − more formal

Non-Execution-Based Testing (contd)

- Non-execution-based testing has to be used when testing artifacts of the requirements, analysis, and design workflows
- Execution-based testing can be applied to only the code of the implementation workflow
- Non-execution-based testing of code (code review) has been shown to be as effective as execution-based testing (running test cases)

10.12 Modularity

- A *module* is
  - A lexically contiguous sequence of program statements,
  - Bounded by boundary elements (that is, `{…}` pairs),
  - Having an aggregate identifier

- Examples:
  - Procedures and functions of the classical paradigm
  - Objects
  - Methods within an object

Three Design Objectives

- Ensure that the *coupling* (degree of interaction between two modules) is as low as possible
  - An ideal product exhibits only *data coupling*
    - » Every argument is either a simple argument or a data structure for which all elements are used by the called module
- Ensure that the *cohesion* (degree of interaction within a module) is as high as possible

Modularity (contd)

- Maximize *information hiding*
  - Ensure that implementation details are not visible outside the module in which they are declared
    - » In the object-oriented paradigm, this can be achieved by careful use of the **`private`** and **`protected`** visibility modifiers

10.13 Reuse

- *Reuse* refers to using components of one product to facilitate the development of a different product with a different functionality
  - Examples of a reusable component:
    - » Module
    - » Class
    - » Code fragment
    - » Design
    - » Part of a manual
    - » Set of test data, a contract
    - » Duration and cost estimate

Reuse (contd)

- Reuse is most important because
  - It takes time (= money) to
    - » specify,
    - » design,
    - » implement,
    - » test, and
    - » document

    a software component

- If we reuse a component, we must retest the component in its new context

10.14 Software Project Management Plan

- The components of a *software project management plan*:
  - The work to be done
  - The resources with which to do it
  - The money to pay for it

Resources

- Resources needed for software development:
  - People
  - Hardware
  - Support software
- Use of resources varies with time
  - The entire software development plan must be a function of time

Three Work Categories

- Project function
  - Work carried on throughout the project
  - Examples:
    - » Project management
    - » Quality control

Three Work Categories (contd)

- Activity
  - Work that relates to a specific phase
  - A major unit of work
    - » With precise beginning and ending dates,
    - » That consumes *resources*, and
    - » Results in *work products* like the budget, design, schedules, source code, or users' manual
- Task
  - An activity comprises a set of *tasks* (the smallest unit of work subject to management accountability)

Completion of Work Products

- A *milestone* is the date on which the work product is to be completed
- It must first pass *reviews* performed by
  - Fellow team members
  - Management
  - The client
- Once the work product has been reviewed and agreed upon, it becomes a *baseline*

Software Project Management Plan (contd)

- A *work package* is
  - A work product, *plus*
    - » Staffing requirements
    - » Duration
    - » Resources
    - » The name of the responsible individual
    - » Acceptance criteria for the work product
    - » The detailed budget as a function of time, allocated to
      - Project functions
      - Activities

Software Project Management Plan (contd)

- *Money* is a vital component of the plan
  - A detailed budget must be worked out
  - The money must be allocated, as a function of time, to the project functions and activities
- Key components of the plan include
  - The c*ost estimate,* and
  - The *duration estimate*