OBJECT-ORIENTED AND CLASSICAL SOFTWARE ENGINEERING

EIGHTH EDITION, WCB/MCGRAW-HILL, 2011

STEPHEN R. SCHACH

Edited by Hyungbae Park **University of Central Missouri**

What is Software?

The product that software professionals build and then support over the long term.

Software encompasses: (1) instructions (computer programs) that when executed provide desired features, function, and performance; (2) data structures that enable the programs to adequately store and manipulate information and (3) documentation that describes the operation and use of the programs.

Software products

- Generic products
  - Stand-alone systems that are marketed and sold to **any customer** who wishes to buy them.
  - Examples – PC software such as editing, graphics programs, project management tools; CAD software; software for specific markets such as appointments systems for dentists.
- Customized products
  - Software that is commissioned by **a specific customer** to meet their own needs.
  - Examples – embedded control systems, air traffic control software, traffic monitoring systems.

Software Engineering

- 1968 NATO Conference, Garmisch, Germany
  - NATO study group coined the term ***software engineering***
  - Building software is similar to other engineering tasks

- Aim: To solve the ***software crisis***
  - Software is delivered
    - Late
    - Over budget
    - With residual faults

Standish Group Data

- Data on projects completed in 2006

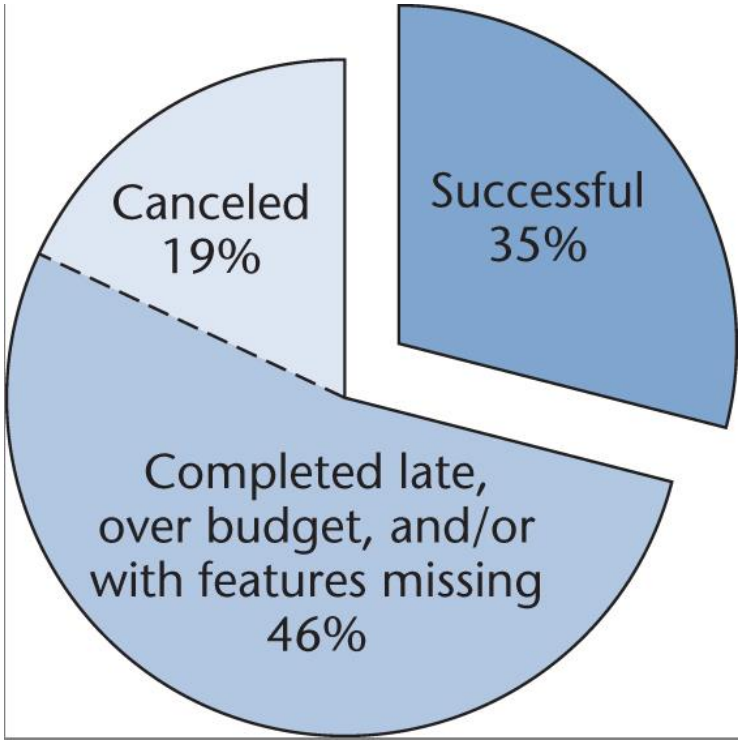


Figure 1.1

Just over one in three projects was successful

Software Engineering (SE)

- Aim is the production of
  - fault-free software
  - delivered on time
  - within budget
  - satisfies the client's needs
  - easy to modify when the user's needs changes

- SE can be categorized as
  - mathematics or computer science
  - economics, management or psychology

Software Engineering

"A discipline whose aim is the production of fault-free software, delivered on-time and within budget, that satisfies the user's needs. Furthermore, the software must be easy to modify when the user's needs change." [Schach]

(Note: This is intended as an explanation and not as a strict definition that should be memorised!)

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
  - **Waterfall life-cycle model** (fig. 10.1)
    - Ø denote empty set
    - Client's Requirements are determined
    - Analysis is performed
    - Design is produced
    - Followed by Implementation of the complete software produce
    - Installed on the client's computer

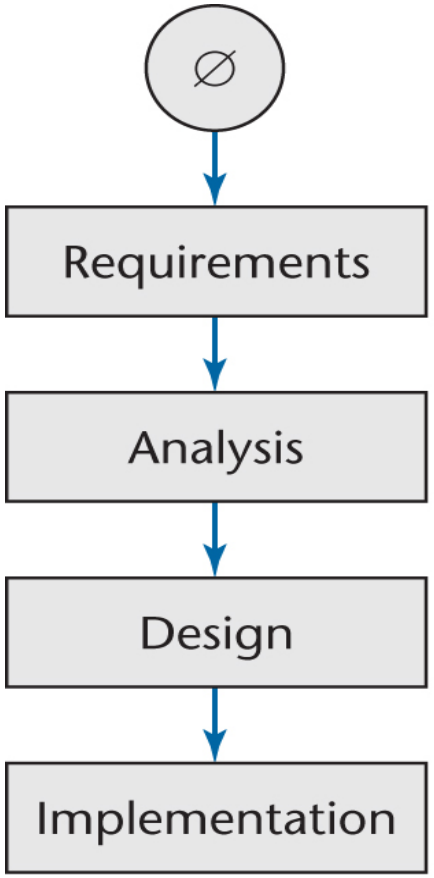


Figure 10.1

Software Development: Theory vs. Practice (contd)

- This is a **life-cycle model** rather than a **life cycle**
  - Life-cycle model: theoretical description of how to build software
  - Life cycle: the actual series of steps followed in the building of a specific product

Software Development: Theory vs. Practice (contd)

- In the real world, software development is totally different
  - We make mistakes
    - During design, if a major fault in requirements is discovered then it has to be fixed before development can proceed
    - During implementation, design flaws often come to light (omissions, ambiguities, or contradictions)

    → life-cycle model breaks down

Software Development: Theory vs. Practice (contd)

- In the real world, software development is totally different
  - The client's requirements change while the software product is being developed
    - ***Moving target problem***: *changes to the requirements before the product is complete*
      - *Client may be expanding into new markets and need additional functionality*
      - *Client company may be losing money and can now afford only a scaled-back version*

10.2 Iteration and Incrementation

- In real life, we cannot speak about "the design phase"
  - The life cycle of actual software cannot be linear
  - We keep returning to earlier phase
  - The operations of the design phase are spread out over the life cycle
- The basic software development process is *iterative*
  - Each successive version is closer to its target than its predecessor
  - Finally construct a version that is satisfactory

Miller's Law

- At any one time, we can concentrate on only approximately seven *chunks* (units of information)
  - Typical software artifact has far more than seven chunks
    - More than seven variables
    - More than seven requirements
- 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
    - Consider seven the most important requirements
    - Then, consider the seven next most important requirements
- This is an *incremental* process

Incrementation

- Iterative-and-incremental life-cycle model
  - One possible way a software product can be decomposed into increments
  - Others may need 2 increments or 13 increments

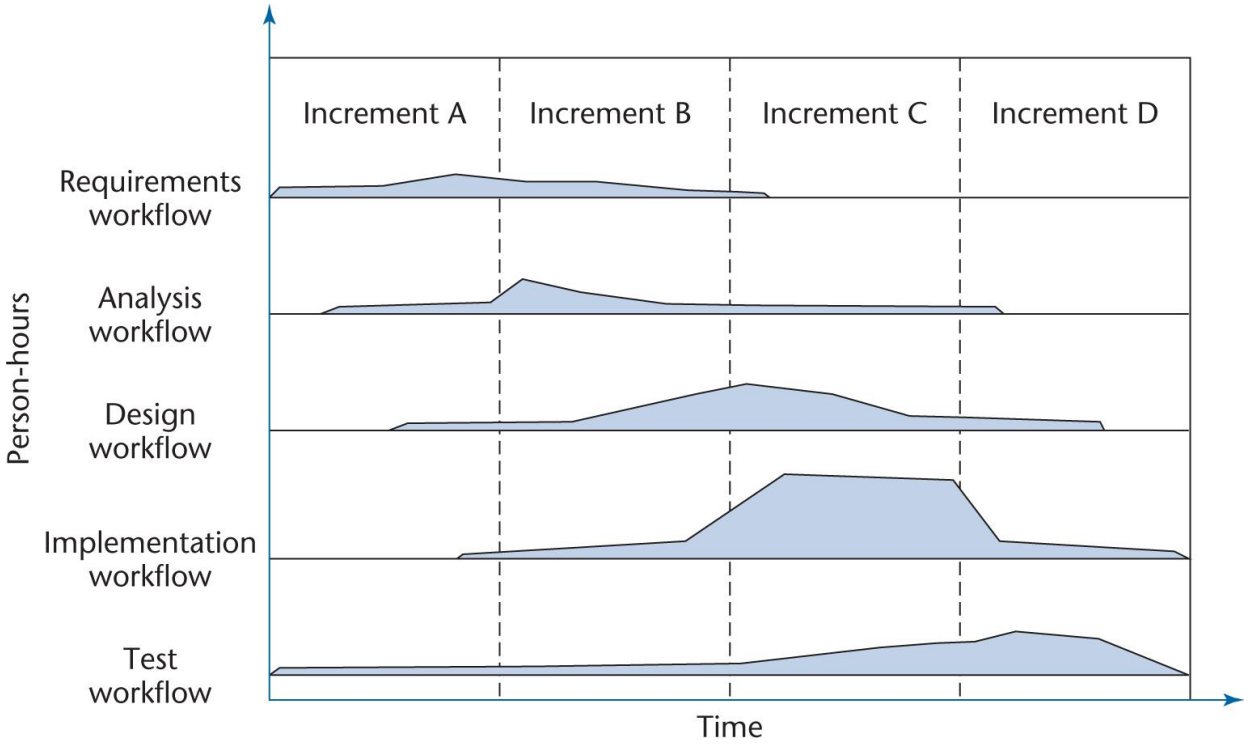


Figure 10.2

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

Requirements workflow

- For development organization to determine the client's needs
  - Acquire a basic understanding of the application domain
    - Banking, automobile manufacturing, or nuclear physics
  - The task of the developers – determine exactly what the client needs and find out from the client what constraints exists
    - Deadline
    - Reliability
    - Cost

Requirements workflow (contd)

- Up to now, everything seems to be straightforward

"I know that this is what I asked for, but it isn't really what I wanted." – Client

- Reasons
  - Client may not truly understand what is going on in his or her own organization
  - Software complexity – main reason
    - Difficult for a software professional to visualize a piece of software and its functionality
    - Unified Process can help - UML diagrams

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
  - The output of the requirements workflow must be totally comprehended by the client
    - Artifacts of the requirements workflow must be expressed in the language of the client
  - Problems can be arise
    - Ambiguity – intrinsic to natural languages
    - Incompleteness – some relevant fact or requirement may be omitted
    - Contradictions – specification document may contain contradictions

Analysis workflow (contd)

- Once the client has approved the specifications, detailed planning and estimating commences
  - Software project management plan
    - Deliverables – what the client is going to get
    - Milestones – when the client gets them
    - Budget – how much it is going to cost

Design workflow

- Refine the artifacts of the analysis workflow until the material is in a form that can be implemented by the programmers
  - Specification of a product spell out ***what*** the product is to do;
  - Design shows ***how*** the product is to do it
  - Designers decompose the product into **modules**
  - Next, the detailed design is performed – algorithms and data structures are selected for each module
  - Now, turn to the object-oriented paradigm **class**

Implementation workflow

- Implement the target software product in the chosen implementation languages(s)
  - Large software product is partitioned into smaller subsystems, then implemented in parallel by coding teams
  - Programmer is given the detailed design of the module he or she is to implement
    - Enough information for the programmer to implement the code artifact without too much difficulty

Test workflow

- Testing is carried out in parallel with the other workflows, starting 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

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

Iterative-and-incremental life-cycle model

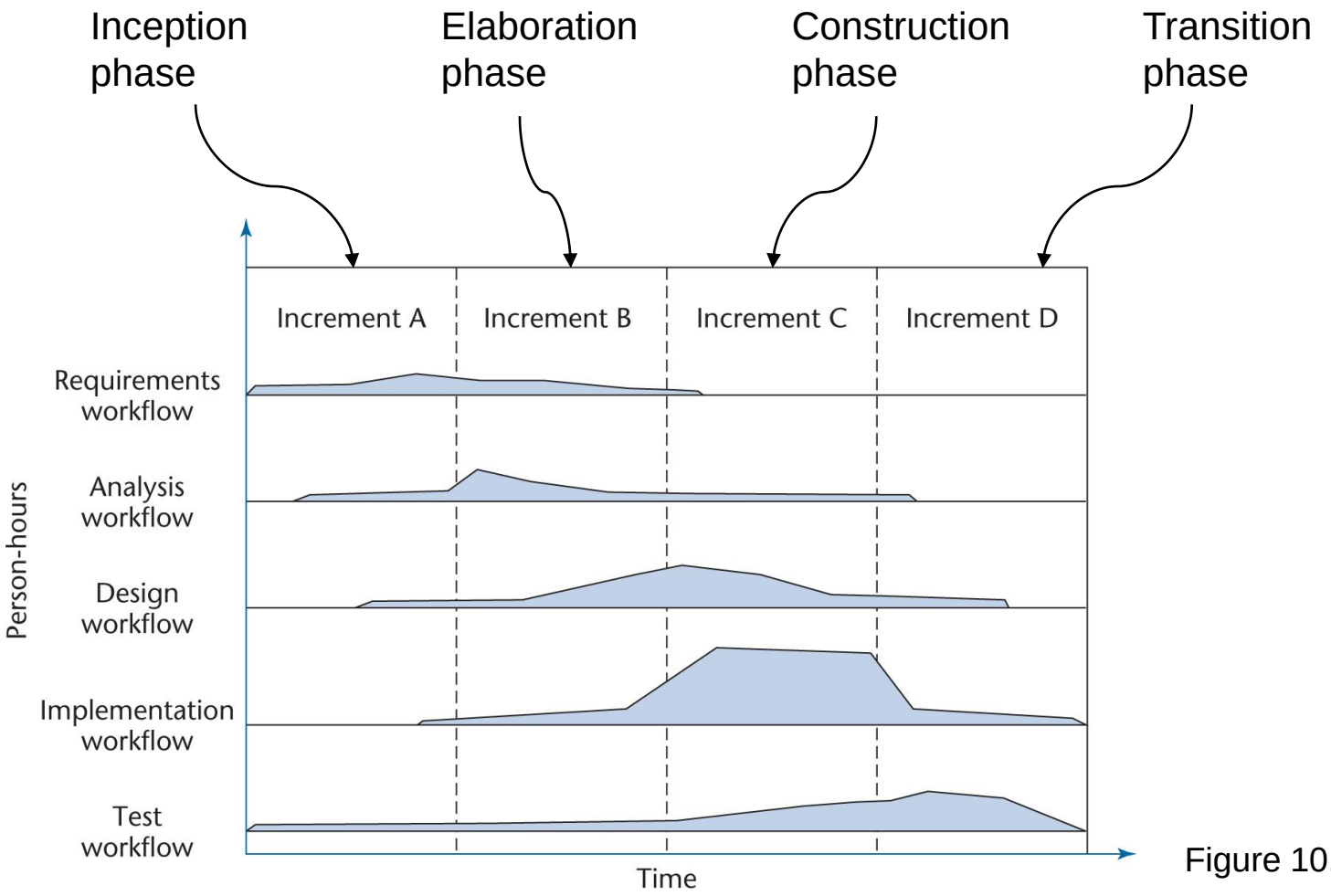


Figure 10.2

Iteration

- Iteration is performed during each incrementation

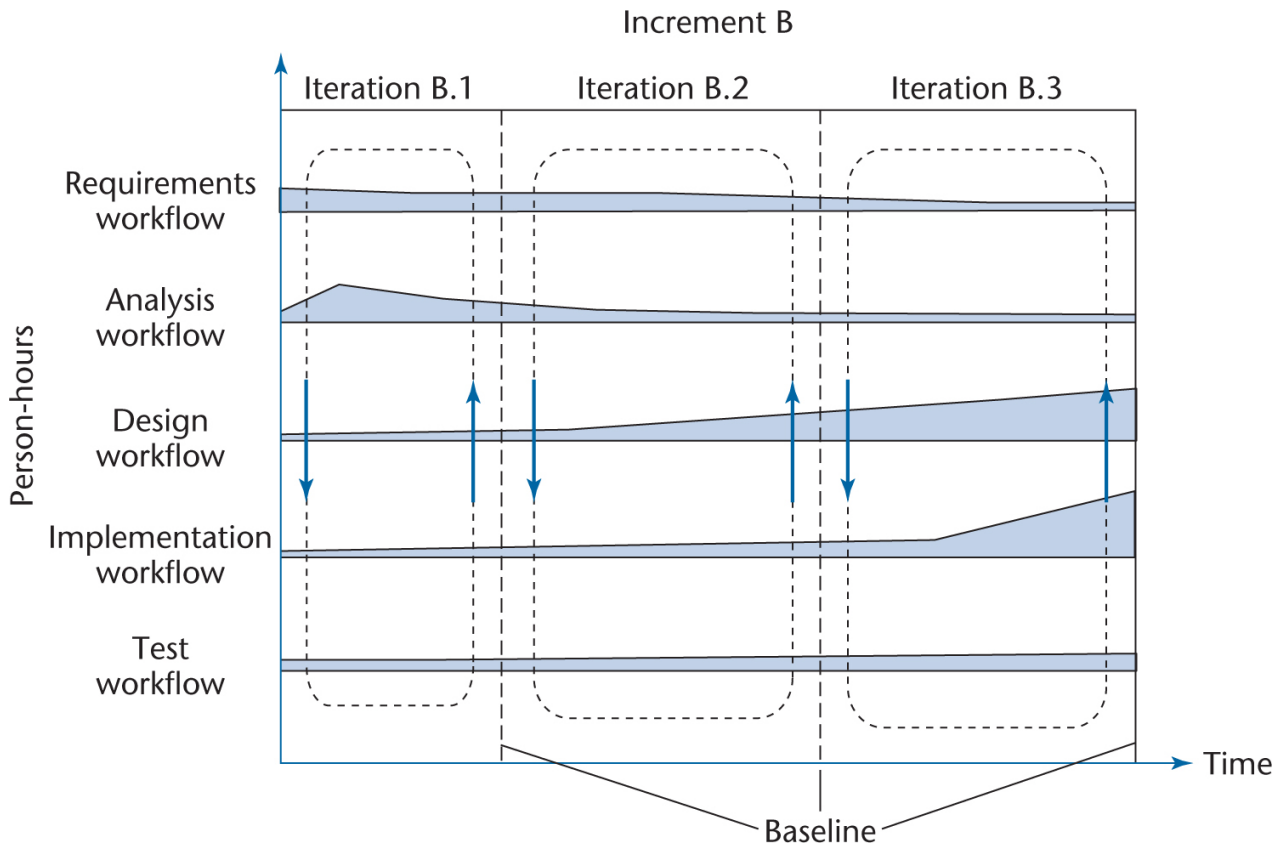


Figure 10.3

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*

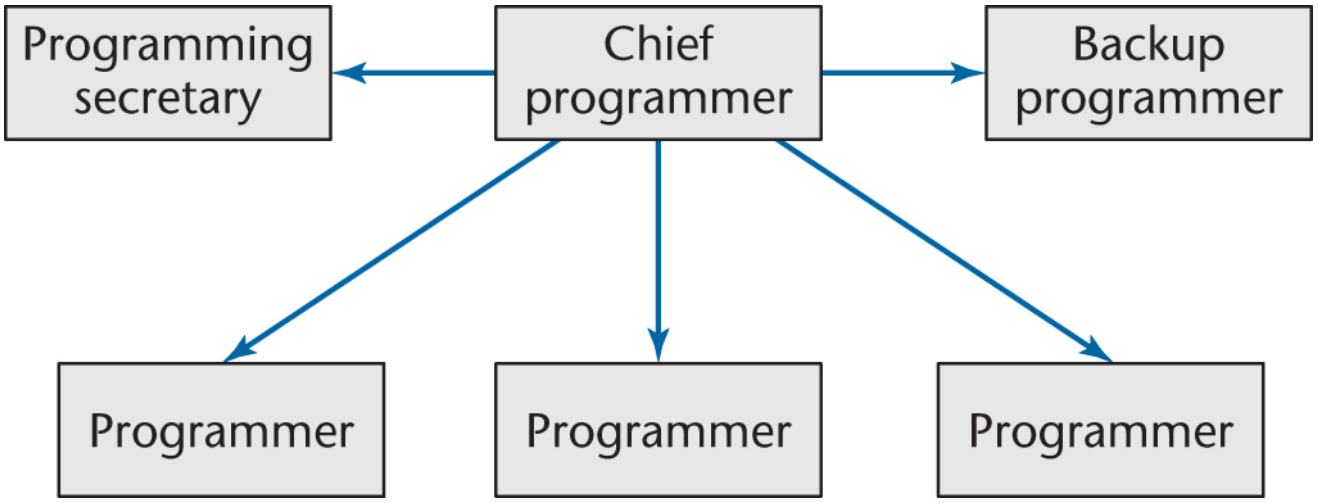
Teams (contd)

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

Democratic Team Approach

- Basic underlying concept — *egoless programming*
- Programmers can be highly attached to their code
  - They even name their modules after themselves
  - They see their modules as extension of themselves
- No leader
  - All decisions in it are making through discussion and with consensus
  - Every member of democratic team have communication interfaces with all other team members, i.e. every team member have **n-1** communication interface, where n is team size

Classical Chief Programmer Team



- Six programmers, but now only 5 lines of communication

Classical Chief Programmer Team (contd)

- Two key aspects
  - Specialization
    - Each member of the team carries out their own tasks
  - Hierarchy
    - The chief directs the actions of all other members of the team and is responsible for every aspect of the operation

Classical Chief Programmer Team (contd)

- Chief programmer
  - Successful manager *and* highly skilled programmer
  - Does the architectural design
  - Allocates coding among the team members
  - Is personally responsible for every line of code

- Back-up programmer
  - The back-up programmer must be in every way as competent as the chief programmer, and
  - Must know as much about the project as the chief programmer

Classical Chief Programmer Team (contd)

- Programming secretary
  - A highly skilled, well paid, central member of the chief programmer teamProgrammers hand their source code to the secretary who is responsible for
    - Conversion to machine-readable form
    - Compilation, linking, loading, execution, and running test cases (this was 1971, remember!)
- Programmers
  - Do nothing but program
  - All other aspects are handled by the programming secretary

Team Organization

Team Organization	Strengths	Weaknesses
Democratic teams (Section 4.2)	High-quality code as consequence of positive attitude to finding faults Particularly good with hard problems	Experienced staff resent their code being appraised by beginners Cannot be externally imposed
Classical chief programmer teams (Section 4.3)	Major success of *The New York Times* project	Impractical
Modified chief programmer teams (Section 4.3.1)	Many successes	No successes comparable to *The New York Times* project
Modern hierarchical programming teams (Section 4.4)	Team manager/team leader structure obviates need for chief programmer Scales up Supports decentralization when needed	Problems can arise unless areas of responsibility of the team manager and the team leader are clearly delineated
Synchronize-and-stabilize teams (Section 4.5)	Encourages creativity Ensures that a huge number of developers can work toward a common goal	No evidence so far that this method can be utilized outside Microsoft
Agile process teams (Section 4.6)	Programmers do not test their own code Knowledge is not lost if one programmer leaves Less-experienced programmers can learn from others Group ownership of code	Still too little evidence regarding efficacy
Open-source teams (Section 4.7)	A few projects are extremely successful	Narrowly applicable Must be led by a superb motivator Requires top-caliber participants

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

Cost–Benefit Analysis (contd)

- Example (yearly)

Cost-Benefit Analysis - Automated Customer Invoicing System							
Costs	**Year**						
	0	**1**	**2**	**3**	**4**	**5**	**6**
Development costs	-50,000						
Operating costs		-75,000	-82,500	-90,750	-99,825	-109,808	
Total Costs	**-50,000**	**-75,000**	**-82,500**	**-90,750**	**-99,825**	**-109,808**	
Discount Factor (Discount rate = 15% p.a.)	1.00	0.87	0.76	0.66	0.57	0.50	
Present Value of Costs	-50,000	-65,217	-62,382	-59,670	-57,075	-54,594	
Cumulative PV Costs	-50,000	-115,217	-177,599	-237,269	-294,344	-348,938	-348,938
Benefits							
Tangible Benefits from new System		110,000	121,000	133,100	146,410	161,051	
Intangible Benefits from new System		10,000	11,000	12,100	13,310	14,641	10,000
Total Benefits		**120,000**	**132,000**	**145,200**	**159,720**	**175,692**	**10,000**
Discount Factor (Discount rate = 15% p.a.)	1.00	0.87	0.76	0.66	0.57	0.50	0.43
Present Value of Benefits		104,348	99,811	95,471	91,320	87,350	4,323
Cumulative PV Benefits		104,348	204,159	299,630	390,951	478,301	482,624
Cumulative PV Benefits+Costs	-50,000	-10,870	26,560	62,361	96,606	129,363	133,686

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)

- Examples:
  - LOC per month
  - Defects per 1000 lines of code

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

- A CASE *tool* assists in just one aspect of the production of software
  - Examples:
    - A tool that draws UML diagrams
    - Data dictionary, a computerized list of all items defined within a product
    - A report generator, which generates the code needed for producing a report

CASE (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*

* *Team project*

Keep all the versions

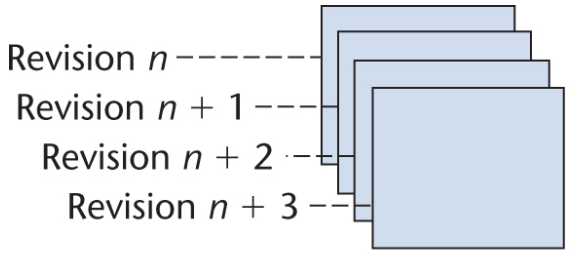
Revisions & Variations

- Revision
  - A version to fix a fault in the artifact
  - The presence of multiple version is easy to solve – any old versions should be thrown away, leaving just the correct one
  - → that would be most unwise

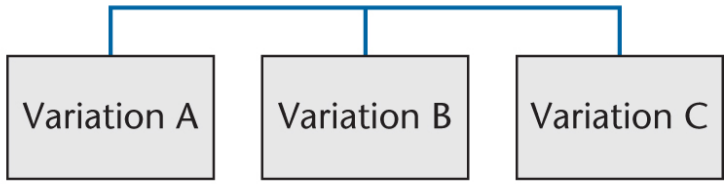
- Variations (example)
  - Most computers support more than one type of printer
  - Computer support and ink-jet printer and a laser printer
  - Operating system must contain two **variations** of the printer driver

Revisions & Variations (contd)

- A variation is a version for a different operating system–hardware
  - Revision: Each of which is implemented specifically to replace its predecessor
  - Variations: Designed to coexist in parallel



(a)



(b)

Figure 5.11

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*
  - Include reviewing software (carefully reading through it)
  - Analyzing software mathematically
    - Review task must be assigned to someone other than the original author of the document
    - Having only one reviewer may not be adequate
- Non-execution-based testing of code (code review) has been shown to be as effective as execution-based testing (running test cases)

Execution-Based Testing

- Execution-based testing can be applied to only the code of the implementation workflow

  "Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence". [Dijkstra, 1972]

  - If a product is executed with test data and the output is wrong, then the product definitely contains a fault
  - If the output is correct, then there still may be a fault in the product
  - The only information that can be deduced from that particular test is that the product runs correctly on that particular set of test data

10.12 Modularity

- Why modularity?
  - When a large product consists of a single monolithic block of code
    - Maintenance is a nightmare
    - Even for the author, attempting to debug the code is extremely difficult
    - For another programmer to understand it is virtually impossible
- Solution
  - Break the product into smaller pieces, called *modules*

10.12 Modularity (contd)

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

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

Modularity (contd)

- Design objectives
  1. Ensure that the *coupling* is as low as possible
     - Coupling: degree of interaction between two modules
     - 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
  2. Ensure that the *cohesion* is as high as possible
     - Cohesion: degree of interaction within a module

Modularity (contd)

3. 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
  - but the other tasks need not be repeated

Reuse (contd)

- Reuse during design and implementation
  - Design team may realize that a class from an earlier design can reused in the current project
    - With or without minor modifications
  - Advantages
    - Tested designs are incorporated into the product
      - The overall design therefore can be produced more quickly and is likely to have a higher quality than when the entire design is produced from scratch
    - If the design of a class can be reused, then the implementation of that class also can be reused
      - If not the actual code then at least conceptually

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
    - OSs, text editors, and version control software
- Use of resources varies with time
  - The entire software development plan must be a function of time

Three Work Categories

Work carried on throughout the project

1. *Project function*
   - Does not related to any specific workflow of software development
   - Examples:
     - Project management
     - Quality control

Three Work Categories (contd)

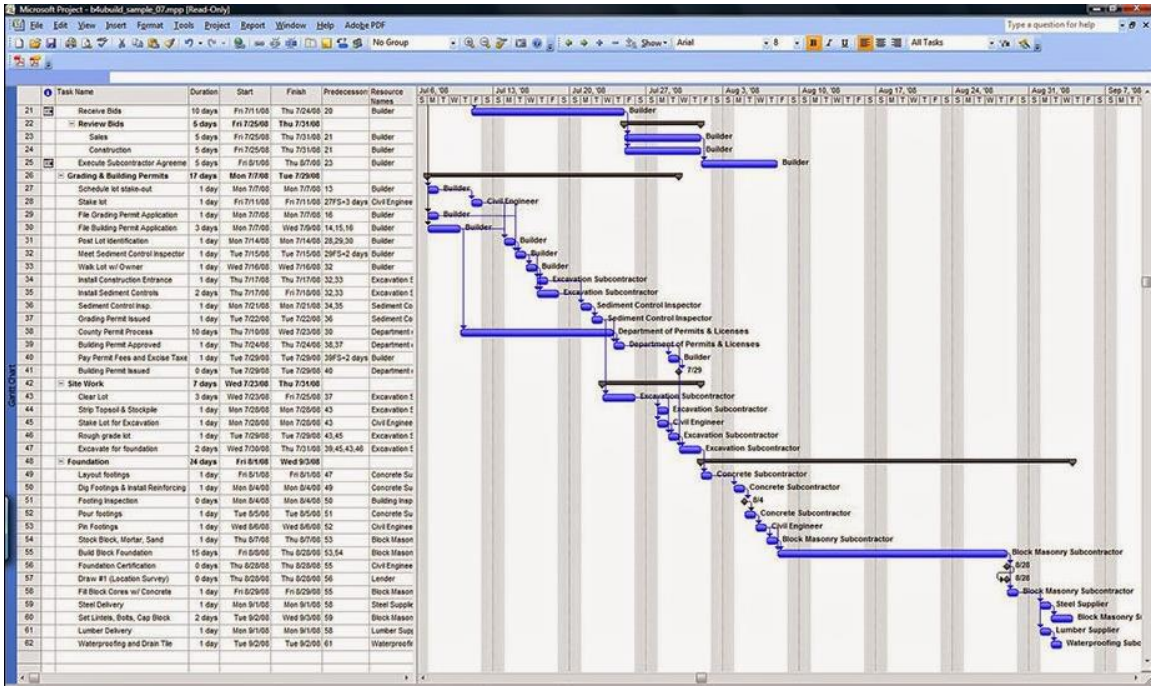
Works that relates to a specific workflow in the development of the product

2. *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

3. *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



Example of milestone