

Object-Oriented and Classical Software Engineering

Eighth Edition, WCB/McGraw-Hill, 2011

Stephen R. Schach

CHAPTER 1

THE SCOPE OF SOFTWARE ENGINEERING

Outline

- Historical aspects
- Economic aspects
- Maintenance aspects
- Requirements, analysis, and design aspects
- Team development aspects
- Why there is no planning phase

Outline (contd)

- Why there is no testing phase
- Why there is no documentation phase
- The object-oriented paradigm
- The object-oriented paradigm in perspective
- Terminology
- Ethical issues

1.1 Historical Aspects

- 1968 NATO Conference, Garmisch, Germany
- Aim: To solve the *software crisis*
- Software is delivered
 - Late
 - Over budget
 - With residual faults

Standish Group Data

- Data on projects completed in 2006

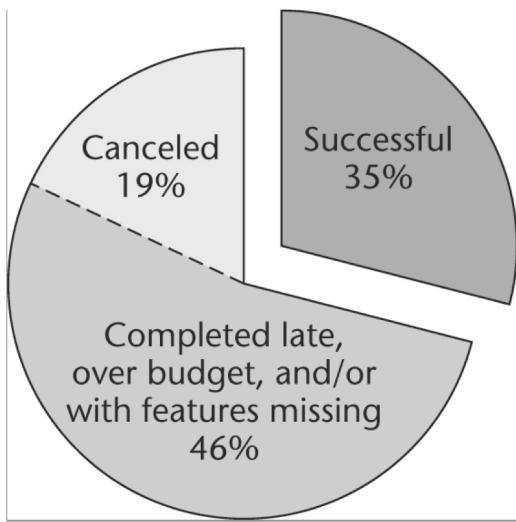


Figure 1.1

- Just over one in three projects was successful

Cutter Consortium Data

- 2002 survey of information technology organizations
 - 78% have been involved in disputes ending in litigation
- For the organizations that entered into litigation:
 - In 67% of the disputes, the functionality of the information system as delivered did not meet up to the claims of the developers
 - In 56% of the disputes, the promised delivery date slipped several times
 - In 45% of the disputes, the defects were so severe that the information system was unusable

Conclusion

- The software crisis has not been solved
- Perhaps it should be called the *software depression*
 - Long duration
 - Poor prognosis

1.2 Economic Aspects

- Coding method CM_{new} is 10% faster than currently used method CM_{old} . Should it be used?
- Common sense answer
 - Of course!
- Software Engineering answer
 - Consider the cost of training
 - Consider the impact of introducing a new technology
 - Consider the effect of CM_{new} on maintenance

1.3 Maintenance Aspects

- Life-cycle model
 - The steps (*phases*) to follow when building software
 - A theoretical description of what should be done
- Life cycle
 - The actual steps performed on a specific product

1.3 Maintenance Aspects

- Life-cycle model
 - The steps (phases) to follow when building software
 - A theoretical description of what should be done
- Life cycle
 - The actual steps performed on a specific product

1.3 Maintenance Aspects

- Life-cycle model
 - The steps (*phases*) to follow when building software
 - A theoretical description of what should be done
- Life cycle
 - The actual steps performed on a specific product

Waterfall Life-Cycle Model

- 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

- 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)

- 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)

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

1.3.1 Classical and Modern Views of Maintenance

- Classical maintenance
 - Development-then-maintenance model
- This is a temporal definition
 - Classification as development or maintenance depends on when an activity is performed

Classical Maintenance Defn—Consequence 1

- A fault is detected and corrected one day after the software product was installed
 - Classical maintenance
- The identical fault is detected and corrected one day before installation
 - Classical development

Classical Maintenance Defn —Consequence 2

- A software product has been installed
- The client wants its functionality to be increased
 - Classical (perfective) maintenance
- The client wants the identical change to be made just before installation (“moving target problem”)
 - Classical development

Classical Maintenance Definition

- The reason for these and similar unexpected consequences
 - Classically, maintenance is defined in terms of the time at which the activity is performed
- Another problem:
 - Development (building software from scratch) is rare today
 - Reuse is widespread

Modern Maintenance Definition

- In 1995, the International Standards Organization and International Electrotechnical Commission defined maintenance *operationally*
- Maintenance is nowadays defined as
 - The process that occurs when a software artifact is modified because of a problem or because of a need for improvement or adaptation

Modern Maintenance Definition (contd)

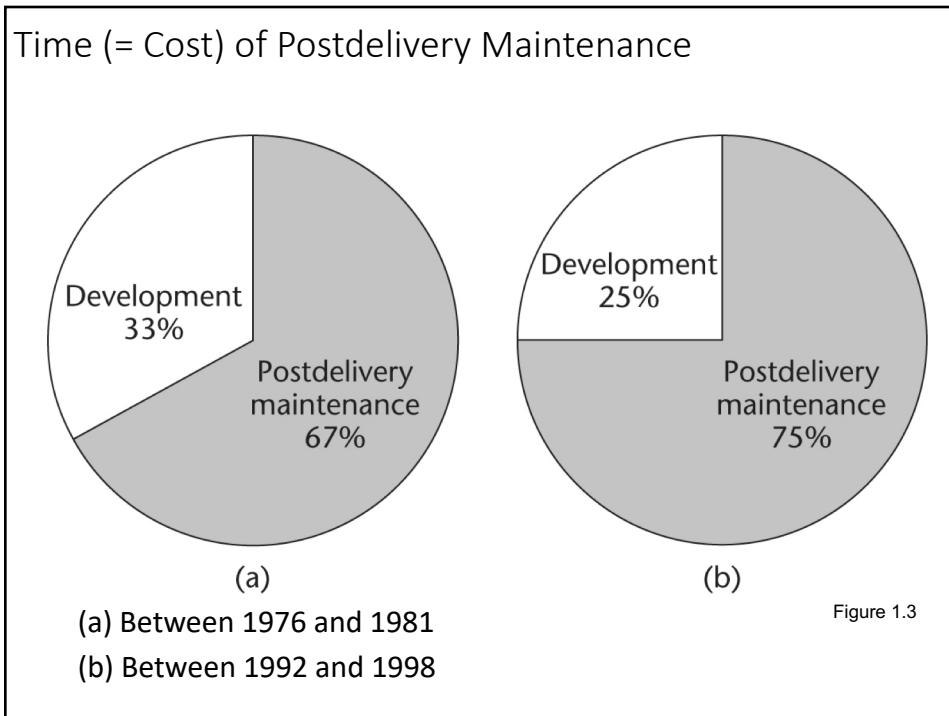
- In terms of the ISO/IEC definition
 - Maintenance occurs whenever software is modified
 - Regardless of whether this takes place before or after installation of the software product
- The ISO/IEC definition has also been adopted by IEEE and EIA

Maintenance Terminology in This Book

- *Postdelivery maintenance*
 - Changes after delivery and installation [IEEE 1990]
- *Modern maintenance* (or just *maintenance*)
 - Corrective, perfective, or adaptive maintenance performed at any time [ISO/IEC 1995, IEEE/EIA 1998]

1.3.2 The Importance of Postdelivery Maintenance

- Bad software is discarded
- Good software is maintained, for 10, 20 years, or more
- Software is a model of reality, which is constantly changing



The Costs of the Classical Phases

- Surprisingly, the costs of the classical phases have hardly changed

	Various Projects between 1976 and 1981	132 More Recent Hewlett-Packard Projects
Requirements and analysis (specification) phases	21%	18%
Design phase	18	19
Implementation phase		
Coding (including unit testing)	36	34
Integration	24	29

Figure 1.4

Consequence of Relative Costs of Phases

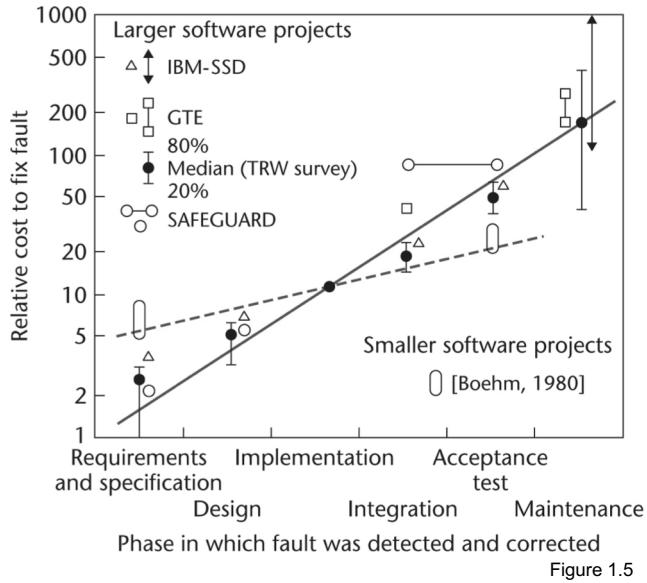
- Return to CM_{old} and CM_{new}
- Reducing the coding cost by 10% yields at most a 0.85% reduction in total costs
 - Consider the expenses and disruption incurred
- Reducing postdelivery maintenance cost by 10% yields a 7.5% reduction in overall costs

1.4 Requirements, Analysis, and Design Aspects

- The earlier we detect and correct a fault, the less it costs us

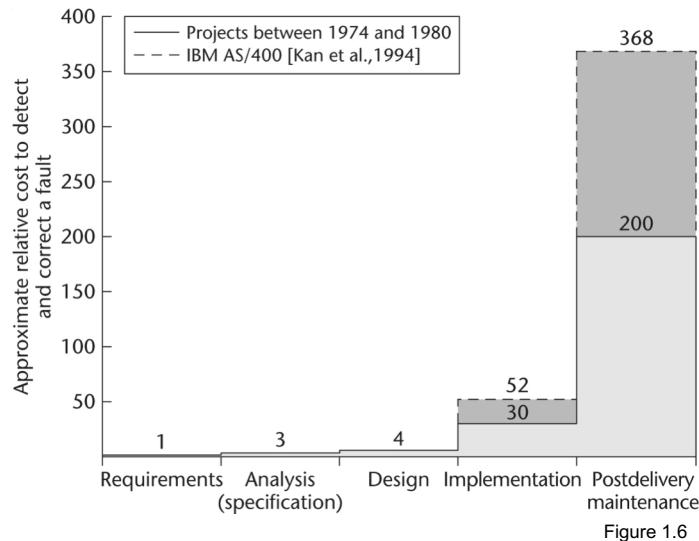
Requirements, Analysis, and Design Aspects (contd)

- The cost of detecting and correcting a fault at each phase



Requirements, Analysis, and Design Aspects (contd)

- The previous figure redrawn on a linear scale



Requirements, Analysis, and Design Aspects (contd)

- To correct a fault early in the life cycle
 - Usually just a document needs to be changed
- To correct a fault late in the life cycle
 - Change the code and the documentation
 - Test the change itself
 - Perform regression testing
 - Reinstall the product on the client's computer(s)

Requirements, Analysis, and Design Aspects (contd)

- Between 60 and 70% of all faults in large-scale products are requirements, analysis, and design faults
- Example: Jet Propulsion Laboratory inspections
 - 1.9 faults per page of specifications
 - 0.9 per page of design
 - 0.3 per page of code

Conclusion

- It is vital to improve our requirements, analysis, and design techniques
 - To find faults as early as possible
 - To reduce the overall number of faults (and, hence, the overall cost)

1.5 Team Programming Aspects

- Hardware is cheap
 - We can build products that are too large to be written by one person in the available time
- Software is built by teams
 - Interfacing problems between modules
 - Communication problems among team members

1.6 Why There Is No Planning Phase

- We cannot plan at the beginning of the project —we do not yet know exactly what is to be built

Planning Activities of the Classical Paradigm

- Preliminary planning of the requirements and analysis phases at the start of the project
- The software project management plan is drawn up when the specifications have been signed off by the client
- Management needs to monitor the SPMP throughout the rest of the project (Software Project Management Plan SPMP)

Conclusion

- Planning activities are carried out throughout the life cycle
- There is no separate planning phase

1.7 Why There Is No Testing Phase

- It is far too late to test after development and before delivery

Testing Activities of the Classical Paradigm

- Verification
 - Testing at the end of each phase (too late)
- Validation
 - Testing at the end of the project (far too late)

Conclusion

- Continual testing activities must be carried out throughout the life cycle
- This testing is the responsibility of
 - Every software professional, and
 - The software quality assurance group
- There is no separate testing phase

1.8 Why There Is No Documentation Phase

- It is far too late to document after development and before delivery

Documentation Must Always be Current

- Key individuals may leave before the documentation is complete
- We cannot perform a phase without having the documentation of the previous phase
- We cannot test without documentation
- We cannot maintain without documentation

Conclusion

- Documentation activities must be performed in parallel with all other development and maintenance activities
- There is no separate documentation phase

1.9 The Object-Oriented Paradigm

- The structured paradigm was successful initially
 - It started to fail with larger products (> 50,000 LOC)
- Postdelivery maintenance problems (today, 70 to 80% of total effort)
- Reason: Structured methods are
 - Action oriented (e.g., finite state machines, data flow diagrams); or
 - Data oriented (e.g., entity-relationship diagrams, Jackson's method);
 - But not both

The Object-Oriented Paradigm (contd)

- Both data and actions are of equal importance
 - Object:
 - A software component that incorporates both data and the actions that are performed on that data
 - Example:
 - Bank account
 - Data: account balance
 - Actions: deposit, withdraw, determine balance

Structured versus Object-Oriented Paradigm

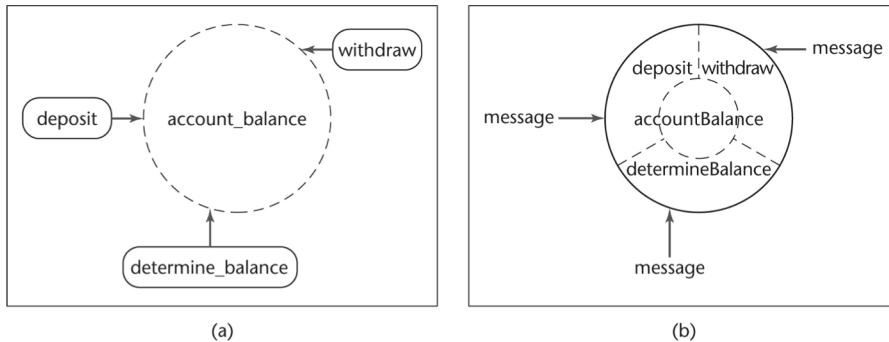


Figure 1.7

- Information hiding
 - Responsibility-driven design
 - Impact on maintenance, development

Information Hiding

- In the object-oriented version
 - The solid line around `accountBalance` denotes that outside the object there is no knowledge of how `accountBalance` is implemented
- In the classical version
 - All the modules have details of the implementation of `account_balance`

Strengths of the Object-Oriented Paradigm

- With information hiding, postdelivery maintenance is safer
 - The chances of a regression fault are reduced
- Development is easier
 - Objects generally have physical counterparts
 - This simplifies modeling (a key aspect of the object-oriented paradigm)

Strengths of the Object-Oriented Paradigm (contd)

- Well-designed objects are independent units
 - Everything that relates to the real-world item being modeled is in the corresponding object — *encapsulation*
 - Communication is by sending *messages*
 - This independence is enhanced by *responsibility-driven design* (see later)

Strengths of the Object-Oriented Paradigm (contd)

- A classical product conceptually consists of a single unit (although it is implemented as a set of modules)
 - The object-oriented paradigm reduces complexity because the product generally consists of independent units
- The object-oriented paradigm promotes reuse
 - Objects are independent entities

Responsibility-Driven Design

- Also called *design by contract*
- Send flowers to your mother in Chicago
 - Call 1-800-flowers
 - Where is 1-800-flowers?
 - Which Chicago florist does the delivery?
 - Information hiding
 - Send a message to a method [action] of an object without knowing the internal structure of the object

Classical Phases vs Object-Oriented Workflows

Classical Paradigm	Object-Oriented Paradigm
1. Requirements phase	1. Requirements workflow
2. Analysis (specification) phase	2'. Object-oriented analysis workflow
3. Design phase	3'. Object-oriented design workflow
4. Implementation phase	4'. Object-oriented implementation workflow
5. Postdelivery maintenance	5. Postdelivery maintenance
6. Retirement	6. Retirement

Figure 1.8

- There is no correspondence between phases and workflows

Analysis/Design “Hump”

- Structured paradigm:
 - There is a jolt between analysis (what) and design (how)
- Object-oriented paradigm:
 - Objects enter from the very beginning

Analysis/Design “Hump” (contd)

- In the classical paradigm
 - Classical analysis
 - Determine what has to be done
 - Design
 - Determine how to do it
 - Architectural design — determine the modules
 - Detailed design — design each module

Removing the “Hump”

- In the object-oriented paradigm
 - Object-oriented analysis
 - Determine what has to be done
 - Determine the objects
 - Object-oriented design
 - Determine how to do it
 - Design the objects
- The difference between the two paradigms is shown on the next slide

In More Detail

Classical Paradigm	Object-Oriented Paradigm
2. Analysis (specification) phase <ul style="list-style-type: none"> • Determine what the product is to do 	2'. Object-oriented analysis workflow <ul style="list-style-type: none"> • Determine what the product is to do • Extract the classes
3. Design phase <ul style="list-style-type: none"> • Architectural design (extract the modules) • Detailed design 	3'. Object-oriented design workflow <ul style="list-style-type: none"> • Detailed design
4. Implementation phase <ul style="list-style-type: none"> • Code the modules in an appropriate programming language • Integrate 	4'. Object-oriented implementation workflow <ul style="list-style-type: none"> • Code the classes in an appropriate object-oriented programming language • Integrate
• Objects enter here	

Figure 1.9

Object-Oriented Paradigm

- Modules (objects) are introduced as early as the object-oriented analysis workflow
 - This ensures a smooth transition from the analysis workflow to the design workflow
- The objects are then coded during the implementation workflow
 - Again, the transition is smooth

1.10 The Object-Oriented Paradigm in Perspective

- The object-oriented paradigm has to be used correctly
 - All paradigms are easy to misuse
- When used correctly, the object-oriented paradigm can solve some (but not all) of the problems of the classical paradigm

The Object-Oriented Paradigm in Perspective (contd)

- The object-oriented paradigm has problems of its own
- The object-oriented paradigm is the best alternative available today
 - However, it is certain to be superseded by something better in the future

1.11 Terminology

- Client, developer, user
- Internal software
- Contract software
- Commercial off-the-shelf (COTS) software
- Open-source software
 - Linus's Law

Terminology (contd)

- Software
- Program, system, product
- Methodology, paradigm
 - Object-oriented paradigm
 - Classical (traditional) paradigm
- Technique

Terminology (contd)

- Mistake, fault, failure, error
- Defect
- Bug 
 - “A bug  crept into the code”
instead of
 - “I made a mistake”

Object-Oriented Terminology

- Data component of an object
 - State variable
 - Instance variable (Java)
 - Field (C++)
 - Attribute (generic)
- Action component of an object
 - Member function (C++)
 - Method (generic)

Object-Oriented Terminology (contd)

- C++: A member is either an
 - Attribute (“field”), or a
 - Method (“member function”)
- Java: A field is either an
 - Attribute (“instance variable”), or a
 - Method

1.12 Ethical Issues

- Developers and maintainers need to be
 - Hard working
 - Intelligent
 - Sensible
 - Up to date and, above all,
 - Ethical
- IEEE-CS ACM Software Engineering Code of Ethics and Professional Practice www.acm.org/serving/se/code.htm

*Object-Oriented and
Classical Software
Engineering*

Eighth Edition, WCB/McGraw-Hill, 2011

Stephen R. Schach

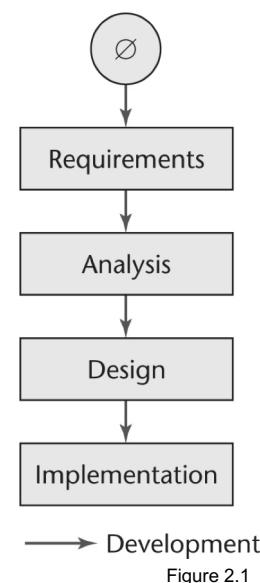
CHAPTER 2
SOFTWARE
LIFE-CYCLE
MODELS

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



Software Development in Practice

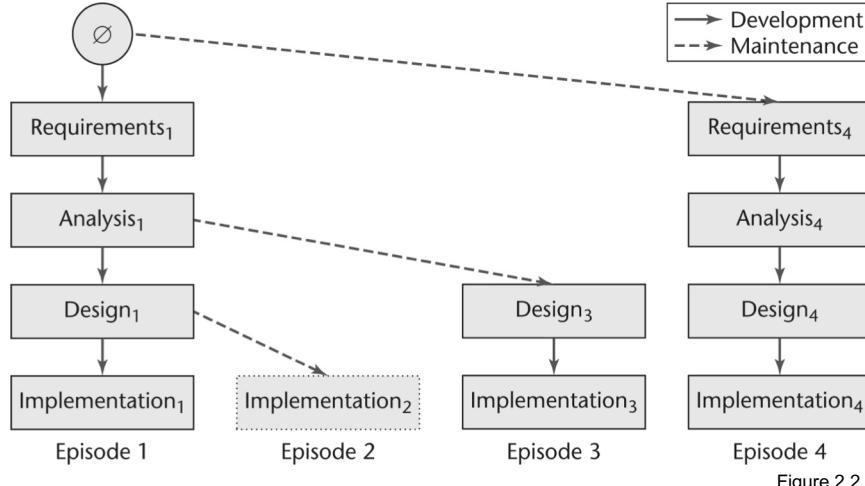
- In the real world, software development is totally different
 - We make mistakes
 - The client's requirements change while the software product is being developed

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

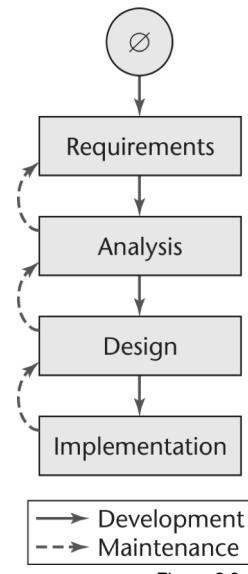
Evolution-Tree Model

- Winburg Mini Case Study



Waterfall Model

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



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:
 - Requirements₁, Analysis₁, Design₃, Implementation₃

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
 - Or a brand new fault in the same area
- If there are too many changes
 - The entire product may have to be redesigned and reimplemented
 - It won't be

Moving Target Problem

- 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)

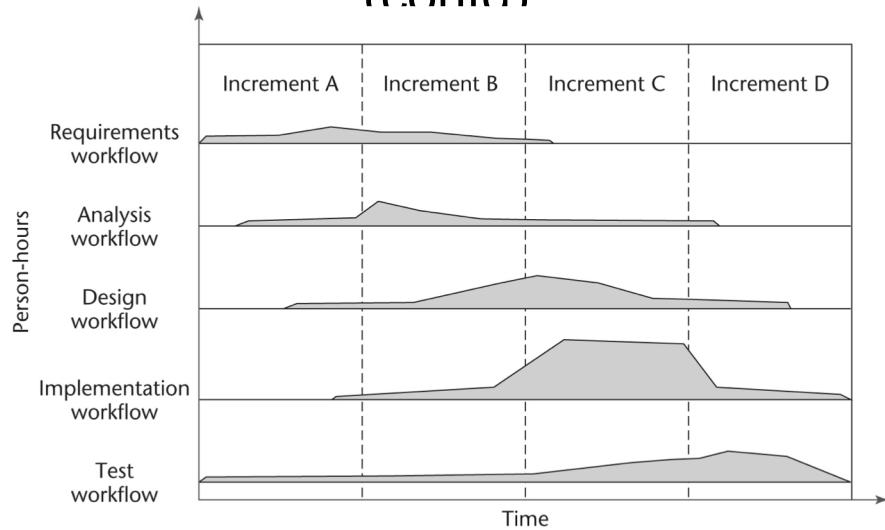


Figure 2.4

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

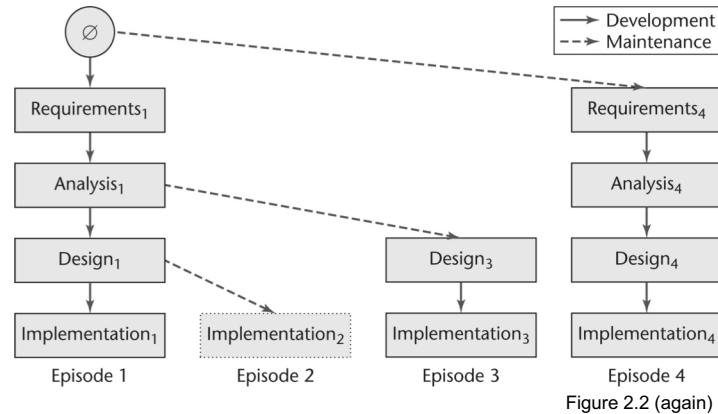


Figure 2.2 (again)

Iteration and Incrementation

- 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

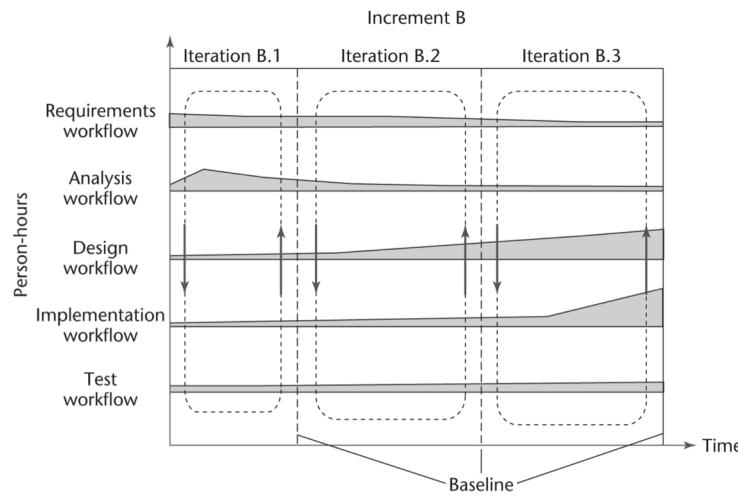


Figure 2.5

Iteration and Incrementation

- 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

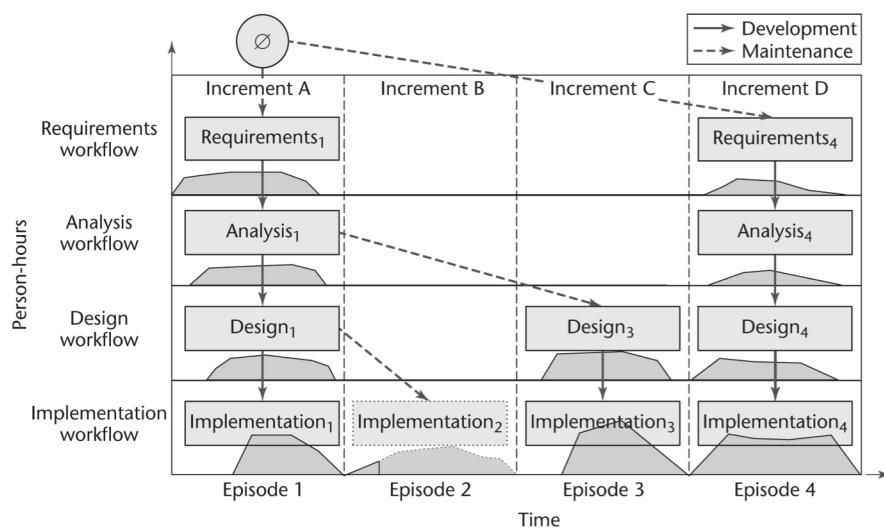


Figure 2.6

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.

- 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

- 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
- Page 51 in your book

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

- CHAOS reports from 1994 to 2006

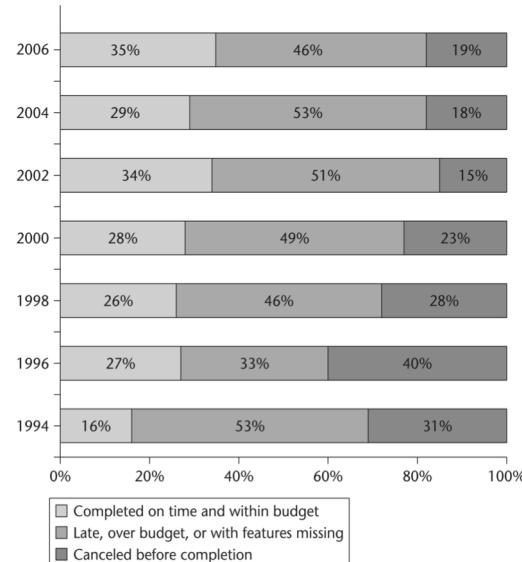


Figure 2.7

Strengths of the Iterative-and-Incremental Model

- 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

Thoughts

- Don't assign good vs. evil traits to anything in software.
- Don't assign good vs. evil to Life Cycle Models.

2.9.1 Code-and-Fix Model

- No design
- No specifications
 - Maintenance nightmare

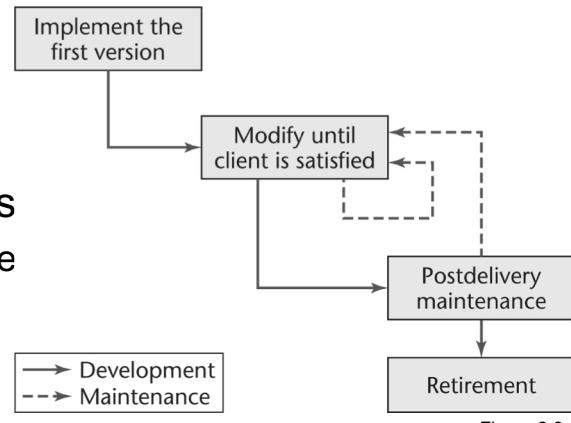
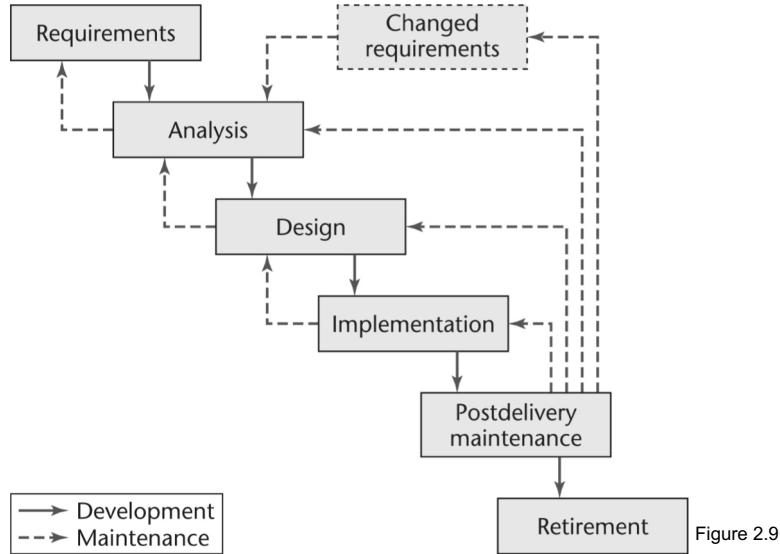


Figure 2.8

Code-and-Fix Model (contd)

- The easiest way to develop software
- Extremely, Extremely popular
- The most expensive way (most times)
 - But nobody believes that

2.9.2 Waterfall Model



2.9.2 Waterfall Model

- 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”

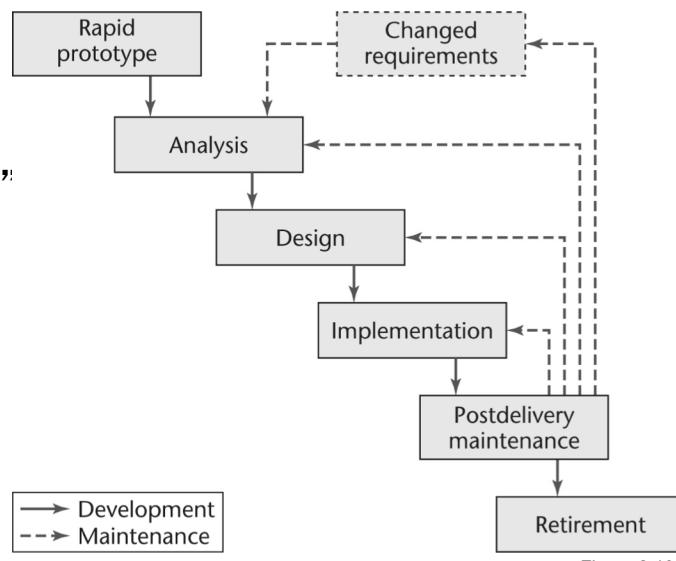


Figure 2.10

Strengths and Pitfalls of Rapid Proto

- Strengths
 - Early functionality
 - Mitigate risk.
 - You can identify your technical debt early (and often)
 - Darn fun!
- Pitfalls
 - Demo's of prototypes often imply completeness!
 - Prototypes sometimes never go away or impose an architecture.
 - Human emotions

2.9.4 Open-Source Life-Cycle Model

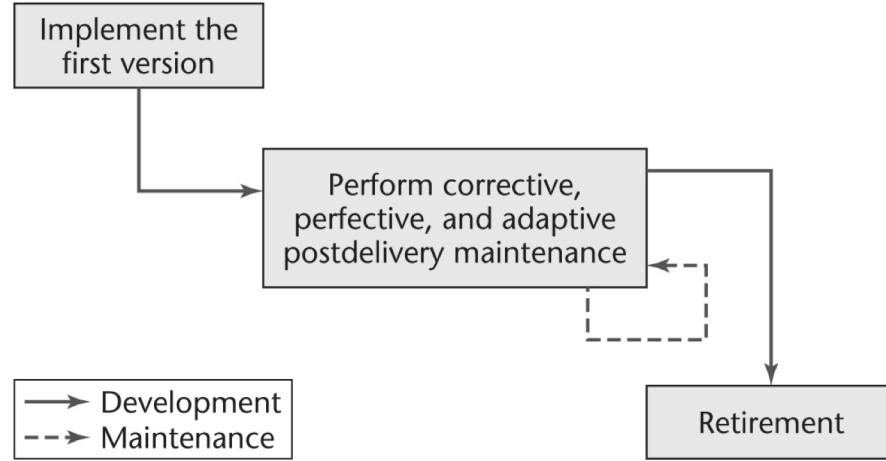
- Two informal phases
- First, one individual builds an initial version
 - Made available via the Internet (e.g., sourceForge.net)
- Then, if there is sufficient interest in the project
 - The initial version is widely downloaded
 - Users become co-developers
 - The product is extended
- Key point: Individuals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



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

- 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

- 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

- 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

- 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

- 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

- 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

- 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

- 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

- 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

- 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

- 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

- 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

- 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 post-delivery 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

- 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

- 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

- 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

- 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

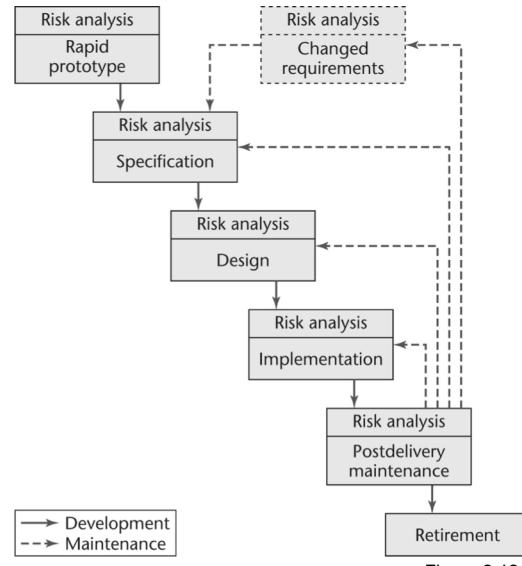
- Microsoft's life-cycle model
- Requirements analysis — interview potential customers
- Draw up specifications
- Divide project into 3 or 4 builds
- Each build is carried out by small teams working in parallel

Synchronize-and Stabilize Model

- 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



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)

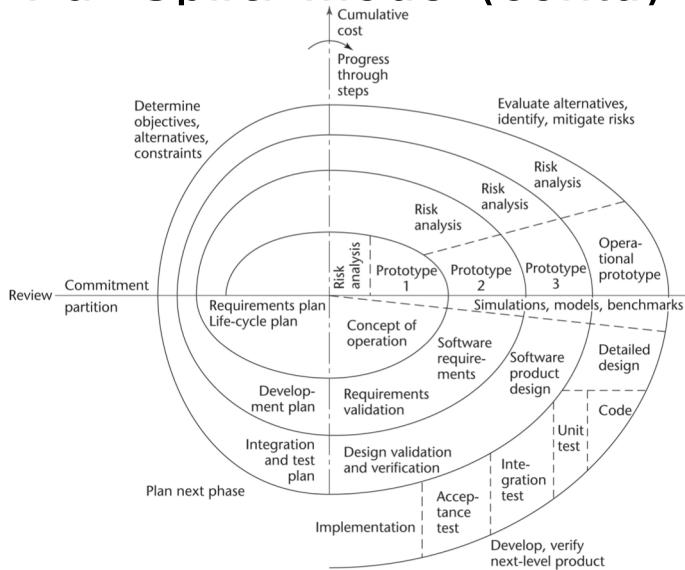


Figure 2.13

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

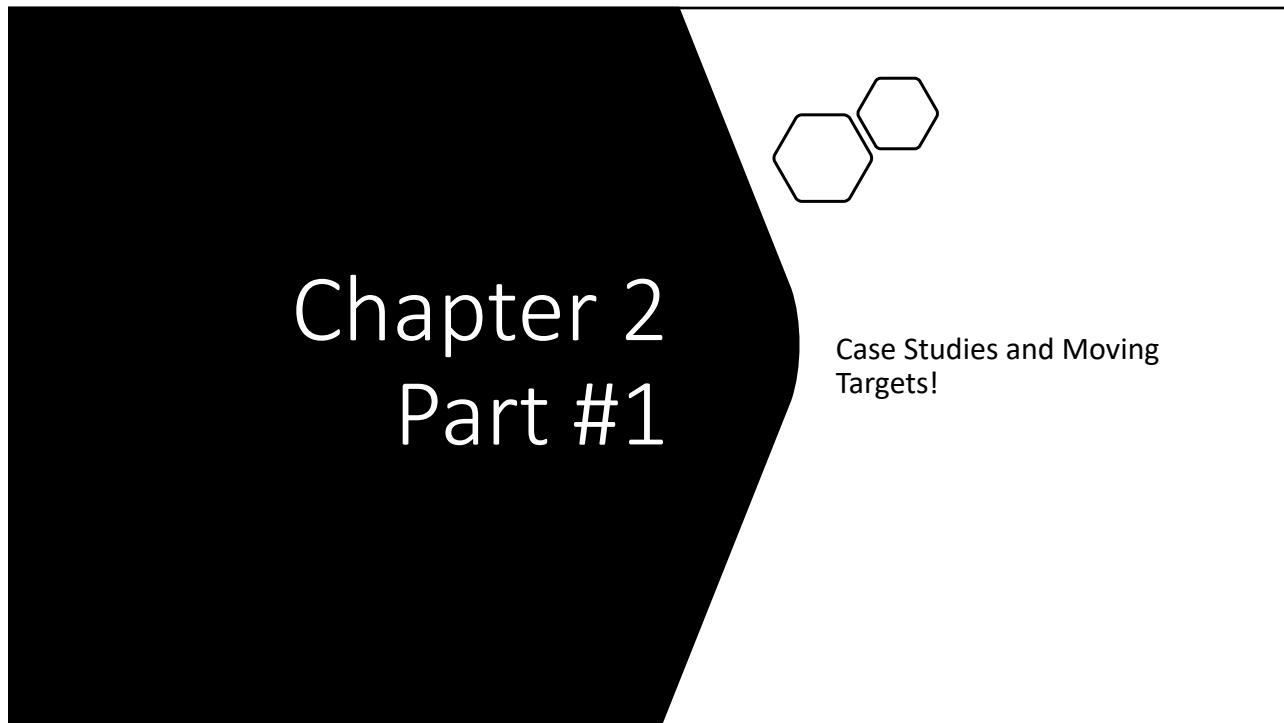
2.10 Comparison of Life-Cycle Models

- 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

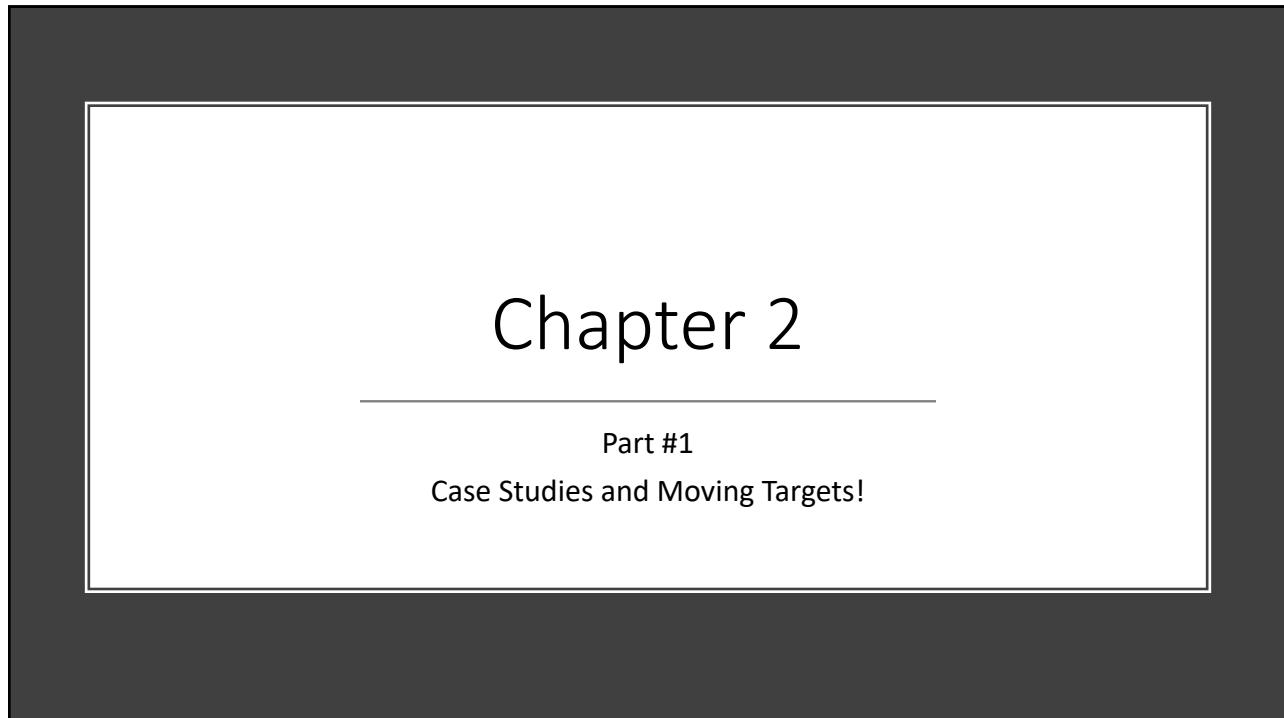
Comparison of Life-Cycle Models (contd)

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

Figure 2.14



1



2

2.1 Software Development in Theory

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

```
graph TD; Start((∅)) --> Requirements[Requirements]; Requirements --> Analysis[Analysis]; Analysis --> Design[Design]; Design --> Implementation[Implementation]; Implementation --> Development[Development]
```

Figure 2.1

3

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

4



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

5

Evolution- Tree Model

Winburg Mini Case Study

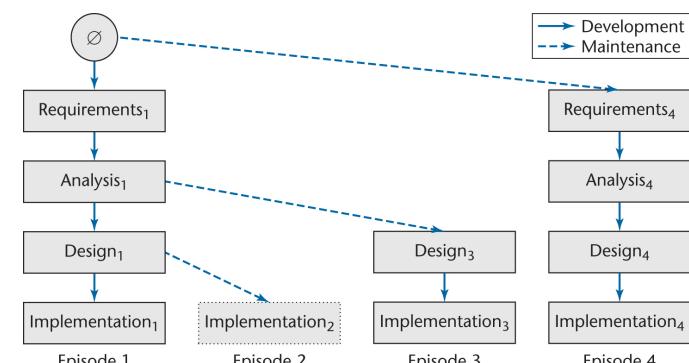
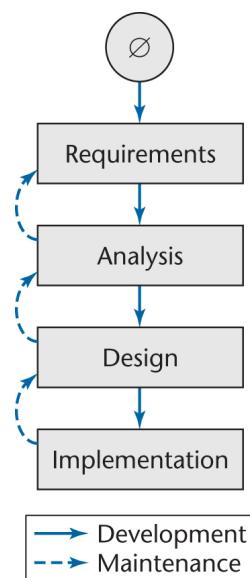


Figure 2.2

6

Waterfall Model

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



7

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:
 - Requirements₁, Analysis₁, Design₃, Implementation₃

8

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

9



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

10

Teal Tractors Mini Case Study (contd)

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

11

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

12

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
 - Or a brand new fault in the same area
- If there are too many changes
 - The entire product may have to be redesigned and reimplemented
 - It won't be

13

Moving Target Problem

- 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

14

Software Engineering

Iteration and
Incrementation

Art and Engineering
Discipline

1

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

2

Incremental vs. Iterative

The diagram illustrates two approaches to problem-solving or design, using the iconic Mona Lisa painting as an example.

Incremental: On the left, a person is shown thinking about the painting, with a thought bubble containing the numbers "1-3". An arrow points from this thought bubble to three separate images of the Mona Lisa, labeled 1, 2, and 3. Each image shows a different aspect of the painting being refined or added incrementally.

Iterative: On the right, a person is shown thinking about the painting, with a thought bubble containing the text "Woman in a personal Setting". An arrow points from this thought bubble to three versions of the Mona Lisa, labeled 1, 2, and 3. In this iterative process, the entire painting is refined in discrete steps, with each version building upon the previous one.

[http://www.infoq.com/resource/news/2008/01/iterating-and-incrementing/en/resources/](http://www.infoq.com/resource/news/2008/01/iterating-and-incrementing/en/resources/Patton_Incremental_Iterative_MonaLisa.jpg)
Patton_Incremental_Iterative_MonaLisa.jpg

3

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

4

Iteration and Incrementation (contd)

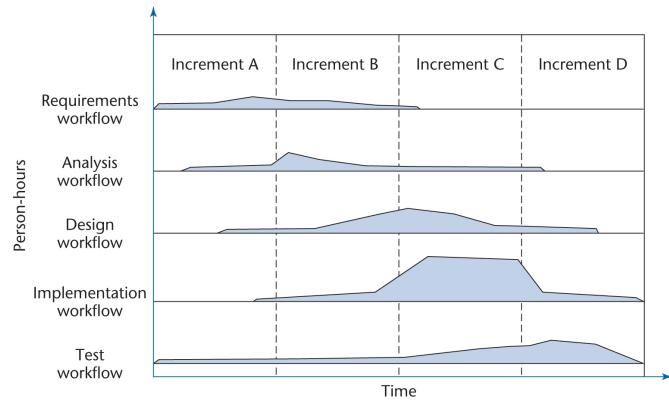


Figure 2.4

5

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

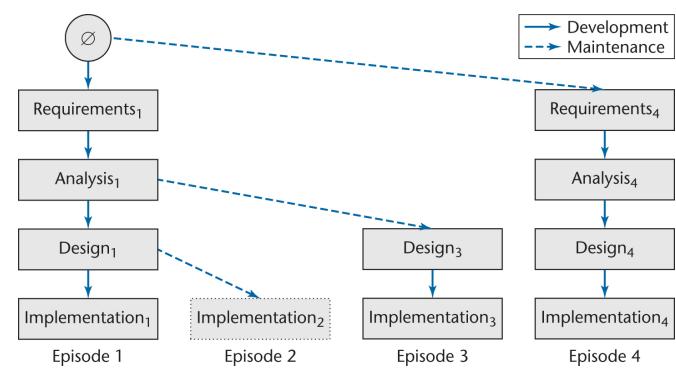


Figure 2.2 (again)

6

Iteration and Incrementation

- The number of increments will vary — ~~it does not have to be~~ It won't be four!

7

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

8

Iteration and Incrementation (contd)

Iteration is performed during each incrementation

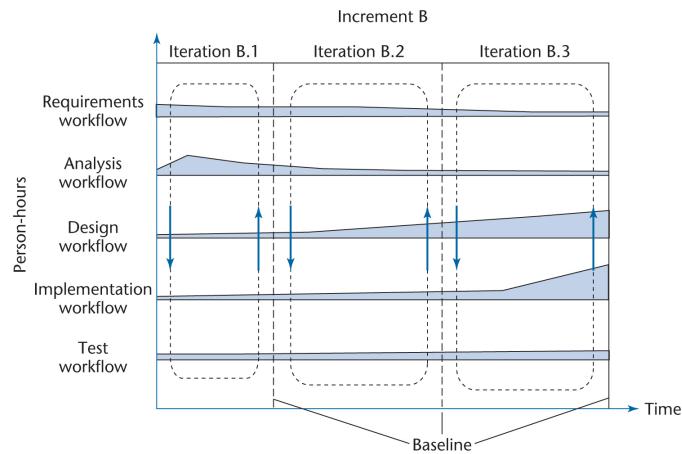


Figure 2.5

10

Iteration and Incrementation

- Again, the number of ***iterations*** will vary—it is not always three.
- **It won't be three.**

11

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

15

Risks and Other Aspects of Iter. and Increm.

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

16

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

17

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

18

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

19

Strengths of the Iterative-and-Incremental Model

- 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
- Page 51 in your book

20

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

23



Conclusion

Discipline is always required



24

Life Cycle Models

Part #3

1

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

Thoughts



Don't assign good vs. evil traits to anything in software.

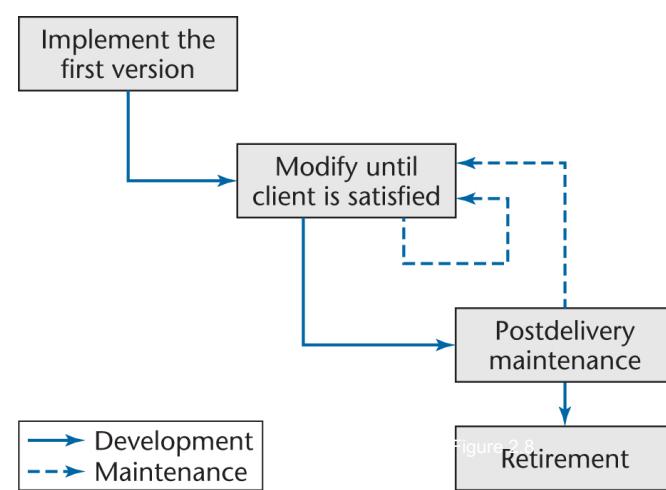


Don't assign good vs. evil to Life Cycle Models.

3

2.9.1 Code-and-Fix Model

- No design
- No specifications
 - Maintenance nightmare



4

Code-and-Fix Model (contd)



The easiest way to develop software



Extremely, Extremely popular



The most expensive way (most times)

But nobody believes that

5

2.9.2 Waterfall Model

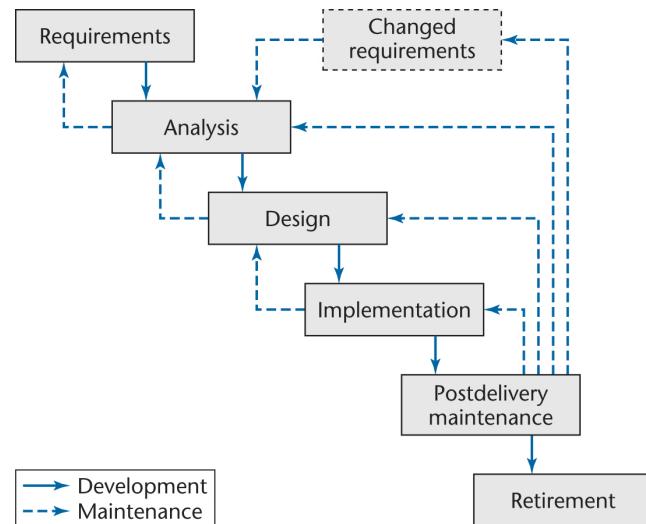


Figure 2.9

6

2.9.2 Waterfall Model

Characterized by

- Feedback loops
- Documentation-driven

Advantages

- Documentation
- Maintenance is easier

Disadvantages

- Specification document
- Joe and Jane Johnson
- Mark Marberry

7

2.9.3 Rapid Prototyping Model

- Linear model
- “Rapid”

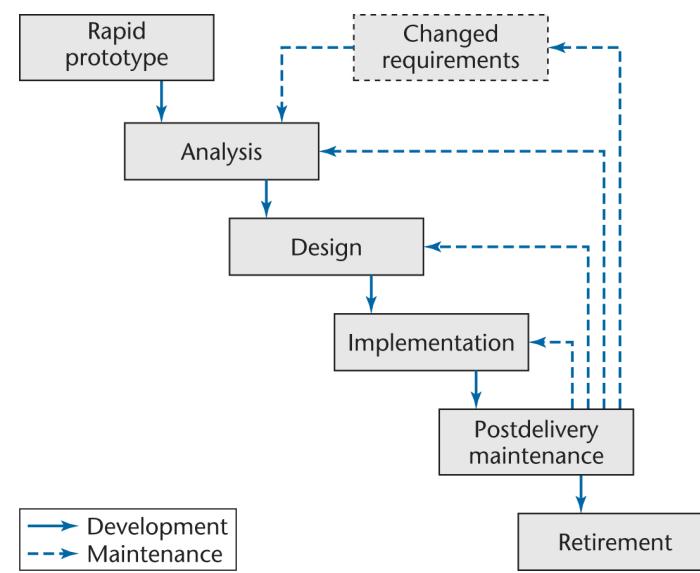


Figure 2.10

8

Strengths and Pitfalls of Rapid Proto

Strengths

- Early functionality
- Mitigate risk.
- You can identify your technical debt early (and often)
- Darn fun!

Pitfalls

- Demo's of prototypes often imply completeness!
- Prototypes sometimes never go away or impose an architecture.
- Human emotions

9

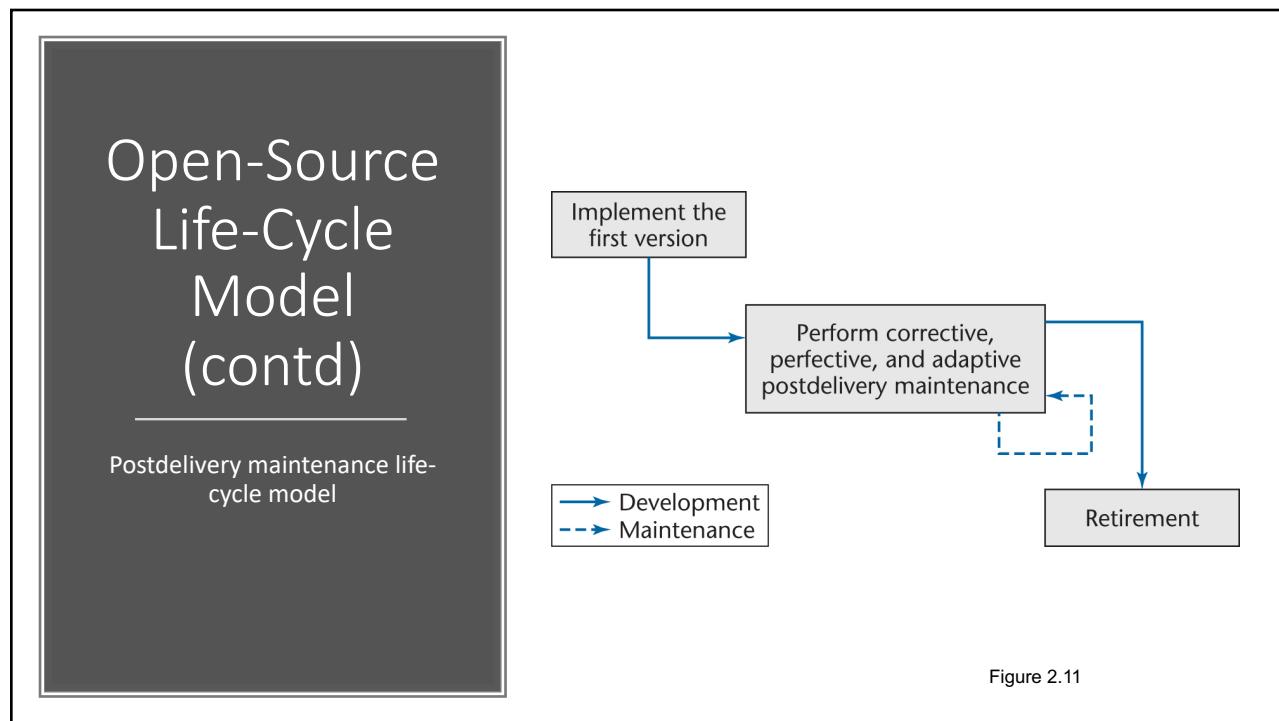
2.9.4 Open-Source Life-Cycle Model

Two informal phases

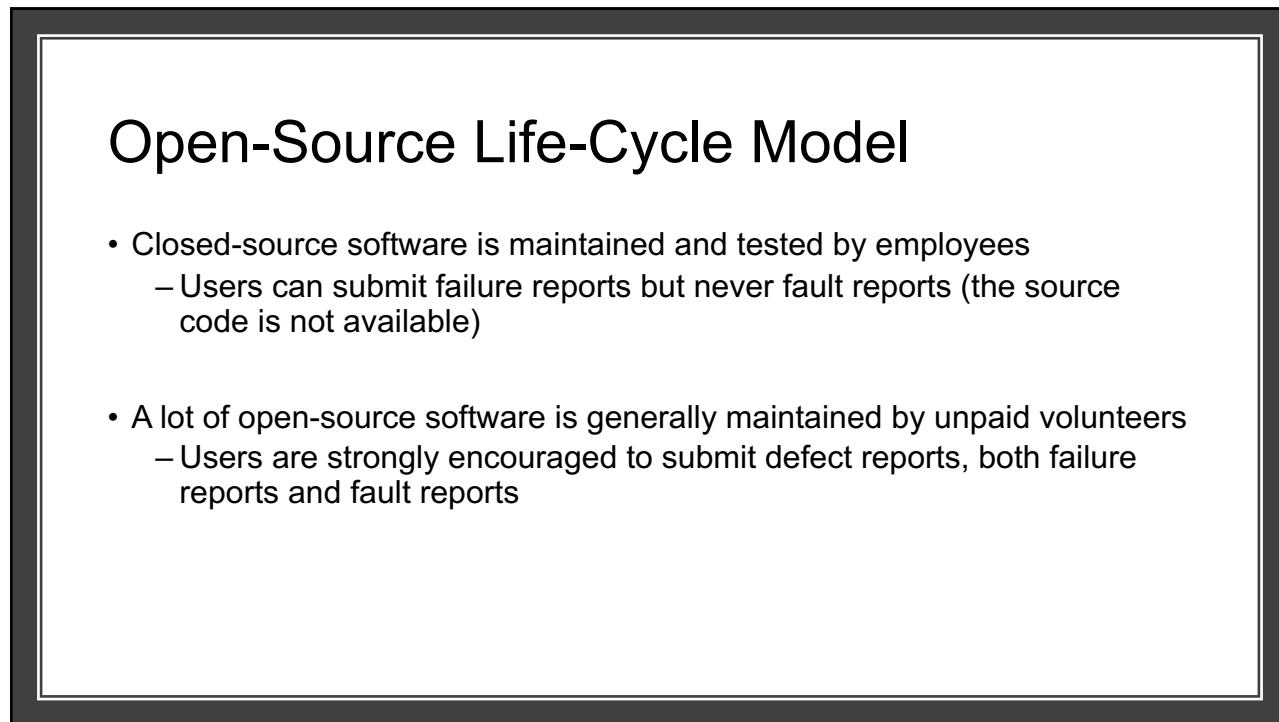
- First, one individual builds an initial version
 - Made available via the Internet (e.g., SourceForge.net)
- Then, if there is sufficient interest in the project
 - The initial version is widely downloaded
 - Users become co-developers
 - The product is extended

Key point: IN many projects individuals generally work voluntarily on an open-source project in their spare time

10



11



12

Open-Source Life-Cycle Model

- 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

13

Open-Source Life-Cycle Model

- Consequently, in an open-source project, in many open source projects, there are no specifications and no design
- How have some open-source projects been so successful without specifications or designs?

14

Open-Source Life-Cycle Model

- 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

15

2.9.5 Agile Processes

Began as controversial 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

16

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)

17

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

18

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

19

Agile Processes

- 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

20

Agile Processes

- 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

21

Agile Processes

- 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

22

Agile Processes

- 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?

23

Agile Processes

- 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

24

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 post-delivery 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?

25

Evaluating Agile Processes

- 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

26

Evaluating Agile Processes

- 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

27

Evaluating Agile Processes

- In conclusion
 - Agile processes appear to be a useful approach to building software products when the client's requirements are evolving and not mission critical.
 - User Interface Projects?
 - Also, some of the proven features of agile processes can be effectively utilized within the context of other life-cycle models

28

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

29

Synchronize-and Stabilize Model

- 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

30

2.9.7 Spiral Model

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

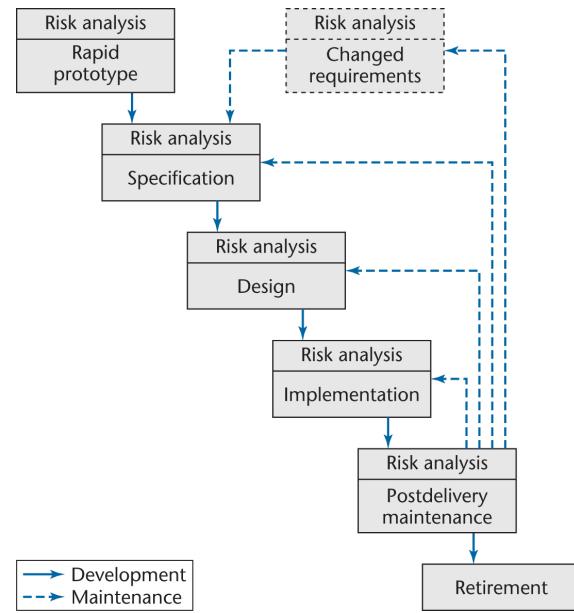


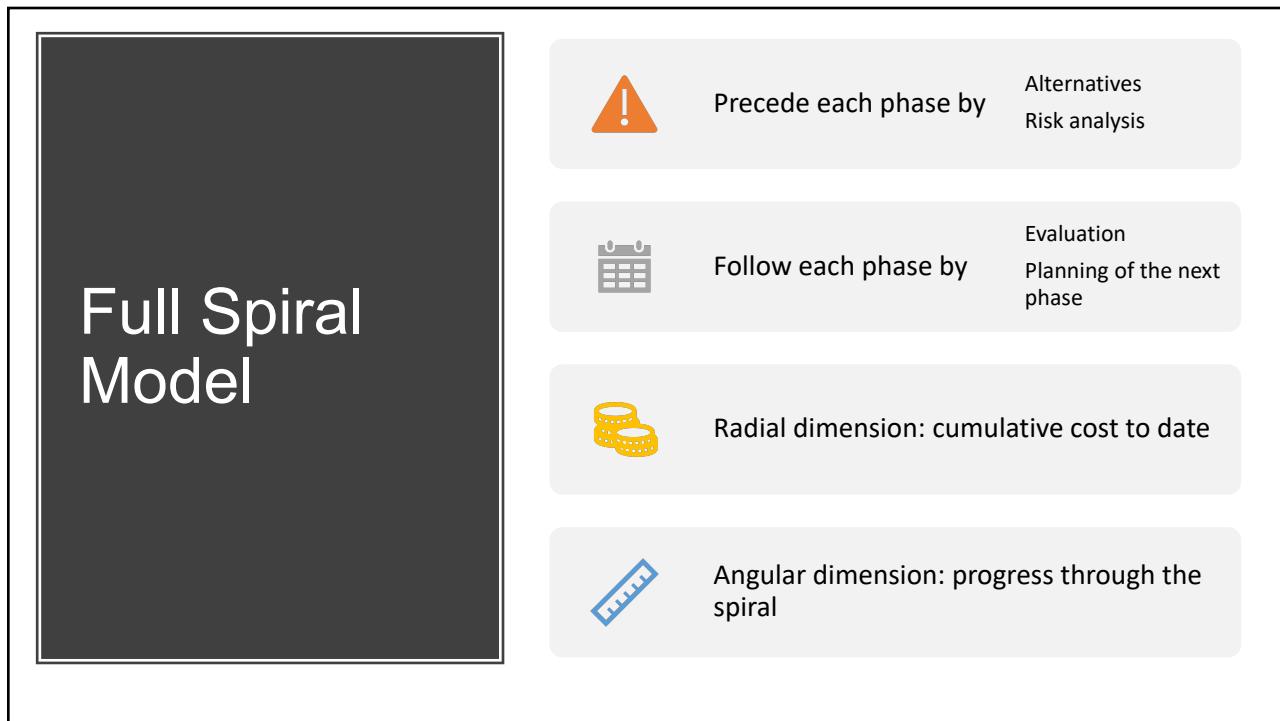
Figure 2.12

31

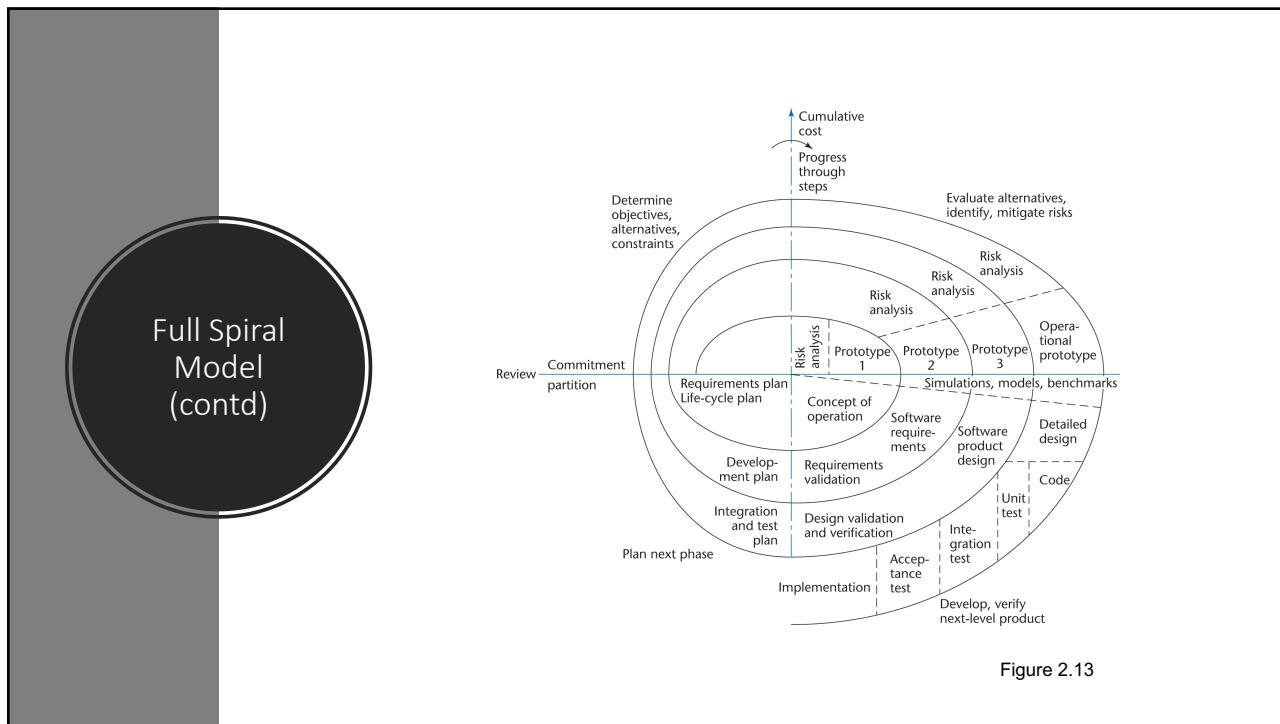
A Key Point of the Spiral Model

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

32



33



34

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

35

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

36

Conclusion

Best life cycle model depends on your particular project.

THE ROAD TO VOLKSWAGON

Software's impact



WHAT
do I
mean?





1960's

- Fake employees
- Robbing fractional cents
- Etc.



1970's John 'Captain Crunch' Draper





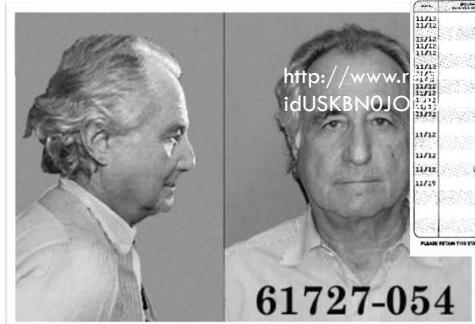
The rec-25

1985 – 1987

Radiation Therapy



Madoff



<http://www.ridjISKRNOJO>

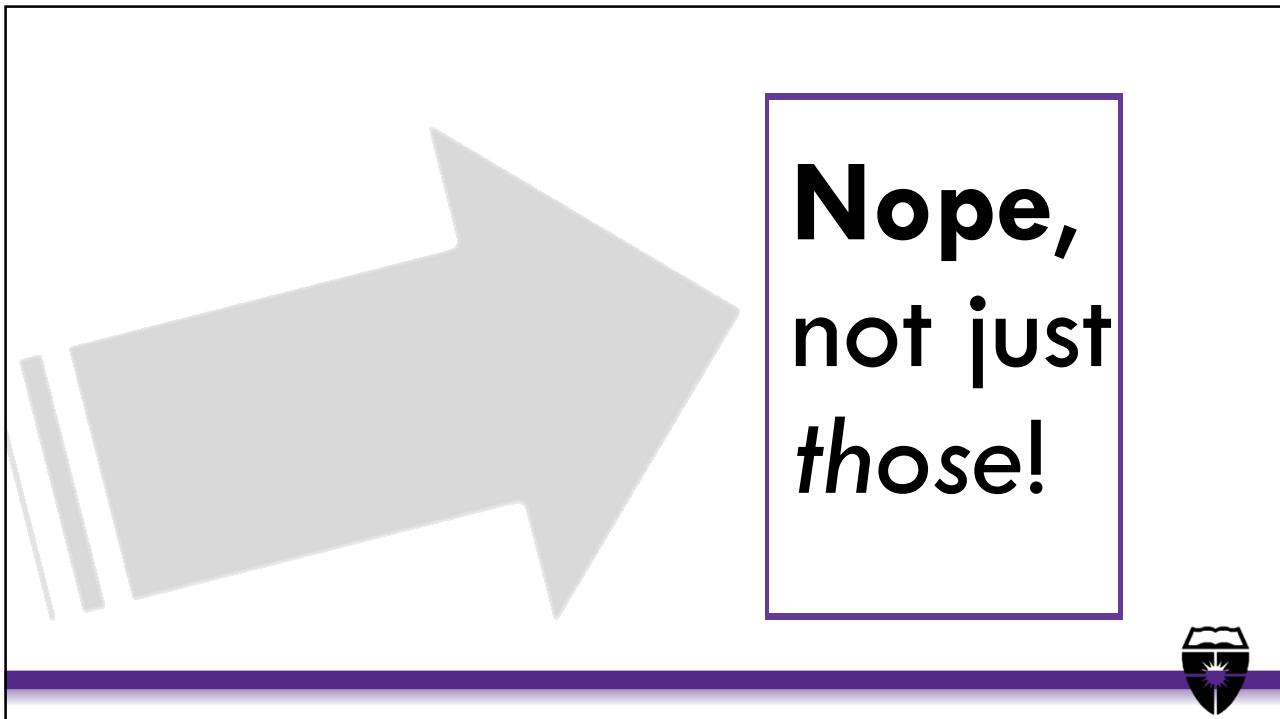
61727-054

		BERNARD H. MADOFF INVESTMENT GROUP LLC New York • London		805 Third Avenue New York, NY 10022 (212) 355-1341 (212) 355-8061		1000 Peachtree Street Atlanta, GA 30309 (404) 522-1022	
						RECEIVED LAW OFFICES OF ***** 2	
DATE	AMOUNT	DEBTOR	DEBTOR ADDRESS	AMOUNT	AMOUNT	AMOUNT	AMOUNT
11/1/21	\$70	727237	BLAUBERGER LTD	\$4,918.00	\$26,235.60	\$1,000.00	\$1,000.00
11/1/21	\$4,918	727237	BLAUBERGER LTD	\$4,918.00	\$26,235.60	\$1,000.00	\$1,000.00
11/1/21	\$2,690	709642	BL A & C INC	\$77	\$72,951.00		
11/1/21	604	800030	CUNKY CHILLIPS	\$51,510.00	\$35,943.00		
11/1/21	455	800030	CUNKY CHILLIPS	\$51,510.00	\$35,943.00		
11/1/21	\$2,794	804931	CLASS 5 SYSTEMS INC	\$16,730.00	\$44,519.62		
11/1/21	73	804931	CLASS 5 SYSTEMS INC	\$16,730.00	\$44,519.62		
11/1/21	73	737	V. S. L. CO., INC.	\$10,730.00	\$10,730.00		
11/1/21	450	826121	UNIFRO TECHNOLOGIES CORP	\$35,943.00			
11/1/21	450	826121	UNIFRO TECHNOLOGIES CORP	\$35,943.00			
11/1/21	\$1,102	866619	WEILSON COMMUNICATIONS	\$33,774.42			
11/1/21	1	23754000	WILLIAMS BROS. INC.	\$9,950.00			
			NET: 24,745,269.00				
11/1/21			FIDELITY SPARTAN		DEV		
11/1/21			U.S. TRUSTORY MONEY MARKET				
11/1/21		754440	DEV 11/12/200				
11/1/21			U.S. TRUSTORY MONEY MARKET		1		
11/1/21			U.S. TRUSTORY MONEY MARKET		3		
11/1/21			U.S. TRUSTORY MONEY MARKET		3		
			U.S. TRUSTORY MONEY MARKET				
			EX-12/1/2000				
			CONTINUED ON PAGE 3				



<http://www.reuters.com/article/us-madoff-workers-sentencing-idUSKBN0JO2BM20141211>





QUICK SURVEY

What did Matthias Müller (from VW) know and when did he know it?
(Informal Survey)

[] Knew exactly what was going on and what was required to make the ship date. (Approved or ordered)

[] Discovered the “trick” after shipment

Fun Fact!

What did Matthias Müller (from VW) know and when did he know it?
(Informally asked)

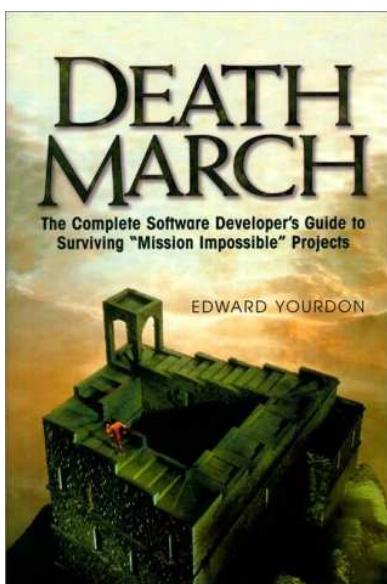
- Most Faculty responded:
 - Knew everything, very early.
 - High involvement.
- Most Engineers when asked
 - Knew nothing until later.





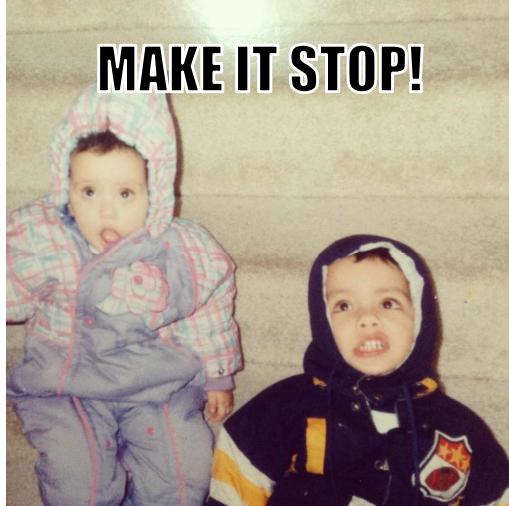
Situational Judgment

The action are you going to take based on a given situation



In project management, a **DEATH MARCH** is a project that the participants feel is destined to fail, or that requires a stretch of unsustainable overwork





PSYCHOLOGY OF CROWD

- Bystander Effect: "someone else will take care of it"
 - Groupthink
 - Deindividuation



PSYCHOLOGY OF INDIVIDUAL

- Positive Reinforcement
- Foot-In-The-Door



It's not like we are building a bridge you know!



Situational Judgment

What would you do?



THE PRACTICE OF ETHICS: DEVELOPING VIRTUE THROUGH SMALL CHOICES



- **40% admit to plagiarism** in undergraduate or graduate school
 - According to plagerism.org
- A Stanford study showed **computer science students** were especially likely to plagiarize



Engineering Ethics



- The first codes of engineering ethics were formally adopted by American engineering societies in 1912-1914.
- Directed engineers to concern themselves with the health, safety and welfare of those who are affected by their work
 - The so-called 'paramountcy clause'



SMALL CHOICES



- Practice doing the right thing everyday
- Do not represent work of others as your own
- Do not shirk responsibilities of design and code reviews
- Make it easy for employees to do the right thing
- Practice “Just Culture”
- Give opportunities to do the right thing



(ACM) Software Engineering Code of Ethics and Professional Practice

1. PUBLIC - Software engineers shall act consistently with the public interest.
2. CLIENT AND EMPLOYER - Software engineers shall act in a manner that is in the best interests of their client and employer consistent with the public interest.
3. PRODUCT - Software engineers shall ensure that their products and related modifications meet the highest professional standards possible.
4. JUDGMENT - Software engineers shall maintain integrity and independence in their professional judgment.
5. MANAGEMENT - Software engineering managers and leaders shall subscribe to and promote an ethical approach to the management of software development and maintenance.
6. PROFESSION - Software engineers shall advance the integrity and reputation of the profession consistent with the public interest.
7. COLLEAGUES - Software engineers shall be fair to and supportive of their colleagues.
8. SELF - Software engineers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession.

Thank you!



Bibliography

- "An Introduction to Software Engineering Ethics." Shannon Vallor, Ph.D. Associate Professor of Philosophy, Santa Clara University . <https://www.scu.edu/media/ethics-center/technology-ethics/Students.pdf>
- "Just Culture: Balancing Safety and Accountability." Choice Reviews Online (2013) . http://flightsafety.org/files/just_culture.pdf
- "Bystander Effect." Psychology Today. <https://www.psychologytoday.com/basics/bystander-effect>
- "Stanford finds cheating — especially among computer science students — on the rise." The Mercury News (2014). <http://www.mercurynews.com/2010/02/06/stanford-finds-cheating-especially-among-computer-science-students-on-the-rise>

Bibliography

- "The Psychology of Crowd Dynamics." Stephen Reicher School of Psychology University of St. Andrews. http://www.uni-kiel.de/psychologie/ispp/doc_upload/Reicher_crowd%20dynamics.pdf
- "Fatal Dose." (1994) Barbara Wade Rose. http://www.ccnr.org/fatal_dose.html
- The Hacker Who Inspired Apple: John 'Captain Crunch' Draper. (2015) Williams, H., Allan, P., & Cashrewards, T. F. <http://www.lifehacker.com.au/2015/11/the-hacker-who-inspired-apple-john-captain-cr>
- Plagiarism Facts and Stats. (2014) <http://www.plagiarism.org/resources/facts-and-stats/>

Chapter 3

Part 1

The Workflows and Activities
Specification Document
Artifacts

1

3.3 The Requirements Workflow

- The aim of the requirements workflow
 - To determine the client's needs

2

Overview of the Requirements Workflow

- First, gain an understanding of the *application domain* (or *domain*, for short)
 - That is, the specific business environment in which the software product is to operate
- Second, build a business model
 - Use UML to describe the client's business processes
 - If at any time the client does not feel that the cost is justified, development terminates immediately

3

Overview of the Requirements Workflow

- It is vital to determine the client's constraints
 - Deadline
 - Nowadays, software products are often mission critical
 - Parallel running
 - Portability
 - Reliability
 - Rapid response time
 - Cost
 - The client will rarely inform the developer how much money is available
 - A bidding procedure is used instead

4

Overview of the Requirements Workflow

- The aim of this *concept exploration* is to determine
 - What the client needs
 - *Not* what the client wants

5

3.4 The Analysis Workflow

- The aim of the analysis workflow
 - To analyze and refine the requirements
- Why not do this during the requirements workflow?
 - The requirements artifacts must be totally comprehensible by the client
- The artifacts of the requirements workflow must therefore be expressed in a natural (human) language
 - All natural languages are **imprecise**

6

The Analysis Workflow

- Example from a manufacturing information system:
 - “A part record and a plant record are read from the database. If it contains the letter A directly followed by the letter Q, then calculate the cost of transporting that part to that plant”
- To what does it refer?
 - The part record?
 - The plant record?
 - Both??
 - Could it mean the database??

7

The Analysis Workflow

- Two separate workflows are needed
 - The requirements artifacts must be expressed in the language of the client
 - The analysis artifacts must be precise, and complete enough for the designers

8

The Specification Document

- Specification document (“specifications”)
 - It constitutes a contract
 - It must not have imprecise phrases like “optimal,” or “98% complete”
- Having complete and correct specifications is essential for
 - Testing and
 - Maintenance

9

What we have seen so far

- Project Name
- Vision
- Boundaries
- Business Case
- Risks
- Supplemental Requirements (Non-Functional)
- Specific Requirements
 - –Supporting use-case
 - –Supporting user-story
- Glossary

10

The Specification Document

- The specification document must not have
 - Contradictions
 - Omissions
 - Incompleteness
- But it will

11

Software Project Management Plan

- Once the client has signed off the specifications, detailed planning and estimating begins
- We draw up the software project management plan, including
 - Cost estimate
 - Duration estimate
 - Deliverables
 - Milestones
 - Budget
- This is the earliest possible time for the SPMP

12

3.5 The Design Workflow

- The aim of the design workflow is to refine the analysis workflow until the material is in a form that can be implemented by the programmers
 - Many nonfunctional requirements need to be finalized at this time, including
 - Choice of programming language
 - Reuse issues
 - Portability issues

13

Classical Design

- Architectural design
 - Decompose the product into modules
- Detailed design
 - Design each module:
 - Data structures
 - Algorithms

14

Object-Oriented Design

- Classes are extracted during the object-oriented analysis workflow and
 - Designed during the design workflow
- Accordingly
 - Classical architectural design corresponds to part of the object-oriented analysis workflow
 - Classical detailed design corresponds to part of the object-oriented design workflow

15

The Design Workflow

- Retain design decisions
 - For when a dead-end is reached
 - To prevent the maintenance team reinventing the wheel

16

3.6 The Implementation Workflow

- The aim of the implementation workflow is to implement the target software product in the selected implementation language
 - A large software product is partitioned into subsystems
 - The subsystems consist of *components* or *code artifacts*

17

3.7 The Test Workflow

- The test workflow is the responsibility of
 - Every developer and maintainer, and
 - The quality assurance group
- Traceability of artifacts is an important requirement for successful testing

18

3.7.1 Requirements Artifacts

- Every item in the analysis artifacts must be traceable to an item in the requirements artifacts
 - Similarly for the design and implementation artifacts

19

3.7.2 Analysis Artifacts

- The analysis artifacts should be checked by means of a review
 - Representatives of the client and analysis team must be present
- The SPMP must be similarly checked
 - Pay special attention to the cost and duration estimates

20

3.7.3 Design Artifacts

- Design reviews are essential
 - A client representative is not usually present

21

3.7.4 Implementation Artifacts

- Each component is tested as soon as it has been implemented
 - *Unit testing*
- At the end of each iteration, the completed components are combined and tested
 - *Integration testing*
- When the product appears to be complete, it is tested as a whole
 - *Product testing*
- Once the completed product has been installed on the client's computer, the client tests it
 - *Acceptance testing*

22

Implementation Artifacts

- COTS software is released for testing by prospective clients
 - Alpha release
 - Beta release
- There are advantages and disadvantages to being an alpha or beta release site

23

3.8 Postdelivery Maintenance

- Postdelivery maintenance is an essential component of software development
 - More money is spent on postdelivery maintenance than on all other activities combined
- Problems can be caused by
 - Lack of documentation of all kinds

24

Postdelivery Maintenance

- Two types of testing are needed
 - Testing the changes made during postdelivery maintenance
 - Regression testing
- All previous test cases (and their expected outcomes) need to be retained

25

3.9 Retirement

- Software can be unmaintainable because
 - A drastic change in design has occurred
 - The product must be implemented on a totally new hardware/operating system
 - Documentation is missing or inaccurate
 - Hardware is to be changed — it may be cheaper to rewrite the software from scratch than to modify it
- These are instances of maintenance (rewriting of existing software)

26

Retirement

- True retirement is a rare event
- It occurs when the client organization no longer needs the functionality provided by the product

27

Technical Context Vs. Business Context

- Workflow
 - Technical context of a step
- Phase Increments and Episodes
 - Business context of a step

28

The End
of Part 1

Rational Unified Process Review

Chapter 3

Part #2

1

The Phases of the Unified Process

- The four increments are labeled
 - Inception phase
 - Elaboration phase
 - Construction phase
 - Transition phase
- The phases of the Unified Process are the increments

2

The Phases of the Unified Process

- In theory, there could be any number of increments
 - In practice, development seems to consist of four increments
- Every step performed in the Unified Process falls into
 - One of the five core workflows and *also*
 - One of the four phases
- Why does each step have to be considered twice?

3

The Phases of the Unified Process

- Workflow
 - Technical context of a step
- Phase
 - Business context of a step

4

3.10.1 The Inception Phase

- The aim of the inception phase is to determine whether the proposed software product is economically viable

5

The Inception Phase

1. Gain an understanding of the domain
2. Build the business model
3. Delimit the scope of the proposed project
Focus on the subset of the business model that is covered by the proposed software product
4. Begin to make the initial business case

6

The Inception Phase : The Initial Business Case

- Questions that need to be answered include:
 - Is the proposed software product cost effective?
 - How long will it take to obtain a return on investment?
 - Alternatively, what will be the cost if the company decides not to develop the proposed software product?
 - If the software product is to be sold in the marketplace, have the necessary marketing studies been performed?
 - Can the proposed software product be delivered in time?
 - If the software product is to be developed to support the client organization's own activities, what will be the impact if the proposed software product is delivered late?

7

The Inception Phase: The Initial Business Case

- What are the risks involved in developing the software product
- How can these risks be mitigated?
 - Does the team who will develop the proposed software product have the necessary experience?
 - Is new hardware needed for this software product?
 - If so, is there a risk that it will not be delivered in time?
 - If so, is there a way to mitigate that risk, perhaps by ordering back-up hardware from another supplier?
 - Are software tools (Chapter 5) needed?
 - Are they currently available?
 - Do they have all the necessary functionality?

8

The Inception Phase: The Initial Business Case

- Answers are needed by the end of the inception phase so that the initial business case can be made

9

The Inception Phase: Risks

- There are three major risk categories:
 - Technical risks
 - See earlier slide
 - The risk of not getting the requirements right
 - Mitigated by performing the requirements workflow correctly
 - The risk of not getting the architecture right
 - The architecture may not be sufficiently robust

10

The Inception Phase: Risks

- To mitigate all three classes of risks
 - The risks need to be ranked so that the critical risks are mitigated first
- This concludes the steps of the inception phase that fall under the requirements workflow

11

The Inception Phase: Analysis, Design Workflows

- A small amount of the analysis workflow may be performed during the inception phase
 - Information needed for the design of the architecture is extracted
- Accordingly, a small amount of the design workflow may be performed, too

12

The Inception Phase: Implementation Workflow

- Coding is generally not performed during the inception phase
- However, a *proof-of-concept prototype* is sometimes build to test the feasibility of constructing part of the software product

13

The Inception Phase: Test Workflow

- The test workflow commences almost at the start of the inception phase
 - The aim is to ensure that the requirements have been accurately determined

14

The Inception Phase: Planning

- There is insufficient information at the beginning of the inception phase to plan the entire development
 - The only planning that is done at the start of the project is the planning for the inception phase itself
- For the same reason, the only planning that can be done at the end of the inception phase is the plan for just the next phase, the elaboration phase

15

The Inception Phase: Documentation

- The deliverables of the inception phase include:
 - The initial version of the domain model
 - The initial version of the business model
 - The initial version of the requirements artifacts
 - A preliminary version of the analysis artifacts
 - A preliminary version of the architecture
 - The initial list of risks
 - The initial ordering of the use cases (Chapter 10)
 - The plan for the elaboration phase
 - The initial version of the business case

16

The Inception Phase: The Initial Business Case

- Obtaining the initial version of the business case is the overall aim of the inception phase
- This initial version incorporates
 - A description of the scope of the software product
 - Financial details
 - If the proposed software product is to be marketed, the business case will also include
 - Revenue projections, market estimates, initial cost estimates
 - If the software product is to be used in-house, the business case will include
 - The initial cost–benefit analysis

17

3.10.2 Elaboration Phase

- The aim of the elaboration phase is to refine the initial requirements
 - Refine the architecture
 - Monitor the risks and refine their priorities
 - Refine the business case
 - Produce the project management plan
- The major activities of the elaboration phase are refinements or elaborations of the previous phase

18

The Tasks of the Elaboration Phase

- The tasks of the elaboration phase correspond to:
 - All but completing the requirements workflow
 - Performing virtually the entire analysis workflow
 - Starting the design of the architecture

19

The Elaboration Phase: Documentation

- The deliverables of the elaboration phase include:
 - The completed domain model
 - The completed business model
 - The completed requirements artifacts
 - The completed analysis artifacts
 - An updated version of the architecture
 - An updated list of risks
 - The project management plan (for the rest of the project)
 - The completed business case

20

3.10.3 Construction Phase

- The aim of the construction phase is to produce the first operational-quality version of the software product
 - This is sometimes called the beta release

21

The Tasks of the Construction Phase

- The emphasis in this phase is on
 - Implementation and
 - Testing
 - Unit testing of modules
 - Integration testing of subsystems
 - Product testing of the overall system

22

The Construction Phase: Documentation

- The deliverables of the construction phase include:
 - The initial user manual and other manuals, as appropriate
 - All the artifacts (beta release versions)
 - The completed architecture
 - The updated risk list
 - The project management plan (for the remainder of the project)
 - If necessary, the updated business case

23

3.10.4 The Transition Phase

- The aim of the transition phase is to ensure that the client's requirements have indeed been met
 - Faults in the software product are corrected
 - All the manuals are completed
 - Attempts are made to discover any previously unidentified risks
- This phase is driven by feedback from the site(s) at which the beta release has been installed

24

The Transition Phase: Documentation

- The deliverables of the transition phase include:
 - All the artifacts (final versions)
 - The completed manuals

Chapter 3

Part 3
2-Dimensional Life Cycle Models Review
Improving the Process

1

3.11 One- and Two- Dimensional Life-Cycle Models

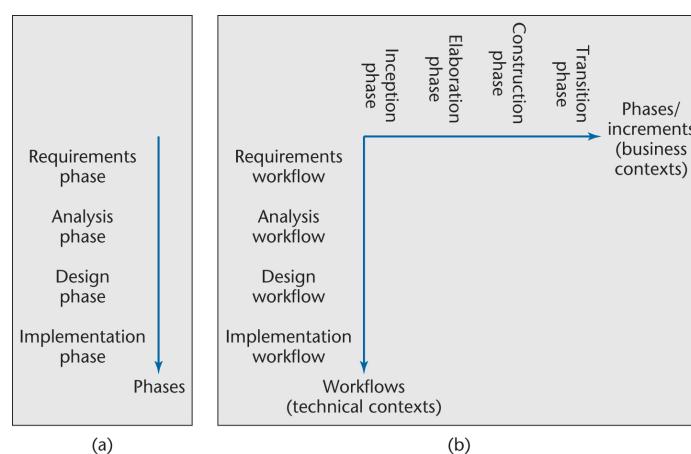


Figure 3.2

2

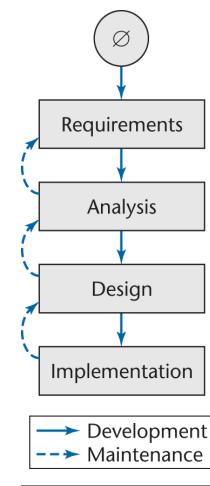
Why a Two-Dimensional Model?

- A traditional life cycle is a one-dimensional model
 - Represented by the single axis on the previous slide
 - Example: Waterfall model
- The Unified Process is a two-dimensional model
 - Represented by the two axes on the previous slide
- The two-dimensional figure shows
 - The workflows (technical contexts) and
 - The phases (business contexts)

3

Why a Two-Dimensional Model?

- The waterfall model
- One-dimensional



4

Why a Two-Dimensional Model?

- Evolution tree model
- Two-dimensional

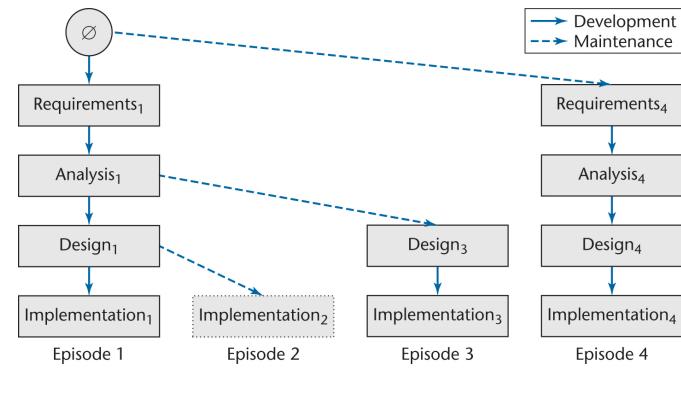


Figure 2.2 (again)

5

Why a Two-Dimensional Model?

- Are all the additional complications of the two-dimensional model necessary?
- In an ideal world, each workflow would be completed before the next workflow is started

6

3.12 Improving the Software Process

- Example:
- U.S. Department of Defense initiative
- Software Engineering Institute (SEI)
- The fundamental problem with software
 - The software process is badly managed

10

Improving the Software Process

- Software process improvement initiatives
 - Capability maturity model (CMM)
 - ISO 9000-series
 - ISO/IEC 15504

11

3.13 Capability Maturity Models

- Not life-cycle models
- Rather, a set of strategies for improving the software process
 - SW-CMM for software
 - P-CMM for human resources (“people”)
 - SE-CMM for systems engineering
 - IPD-CMM for integrated product development
 - SA-CMM for software acquisition
- These strategies are unified into CMMI (capability maturity model integration)

12

SW-CMM

- A strategy for improving the software process
- Put forward in 1986 by the SEI
- Fundamental ideas:
 - Improving the software process leads to
 - Improved software quality
 - Delivery on time, within budget
 - Improved management leads to
 - Improved techniques

13

SW-CMM

- Five levels of *maturity* are defined
 - Maturity is a measure of the goodness of the process itself
- An organization advances stepwise from level to level

14

Level 1. Initial Level

- Ad hoc approach
 - The entire process is unpredictable
 - Management consists of responses to crises
- Most organizations world-wide are at level 1

15

Level 2. Repeatable Level

- Basic software management
 - Management decisions should be made on the basis of previous experience with similar products
 - Measurements (“metrics”) are made
 - These can be used for making cost and duration predictions in the next project
 - Problems are identified, immediate corrective action is taken

16

Level 3. Defined Level

- The software process is fully documented
 - Managerial and technical aspects are clearly defined
 - Continual efforts are made to improve quality and productivity
 - Reviews are performed to improve software quality
 - CASE environments are applicable *now* (and not at levels 1 or 2)

17

Level 4. Managed Level

- Quality and productivity goals are set for each project
 - Quality and productivity are continually monitored
 - Statistical quality controls are in place

18

Level 5. Optimizing Level

- Continuous process improvement
 - Statistical quality and process controls
 - Feedback of knowledge from each project to the next

19

Summary

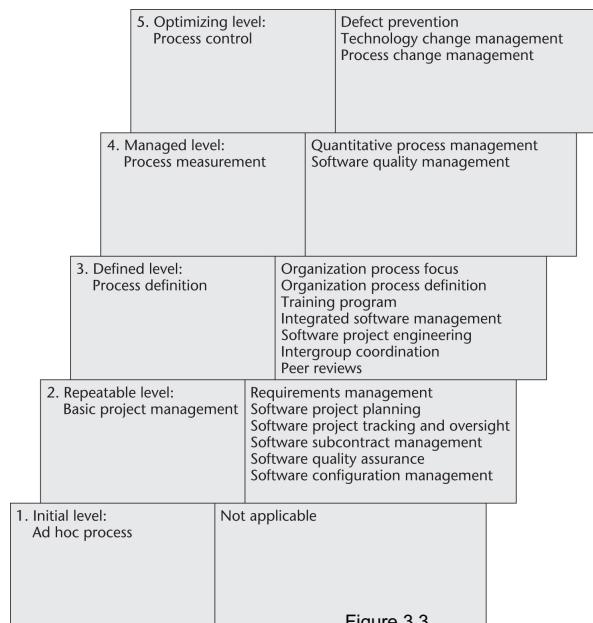


Figure 3.3

20

Experiences with SW-CMM

- It takes:
 - 3 to 5 years to get from level 1 to level 2
 - 1.5 to 3 years from level 2 to level 3
 - SEI questionnaires highlight shortcomings, suggest ways to improve the process

21

Key Process Areas

- There are key process areas (KPAs) for each level

22

Key Process Areas

- Level-2 KPAs include:
 - Requirements management
 - Project planning
 - Project tracking
 - Configuration management
 - Quality assurance
- Compare
 - Level 2: Detection and correction of faults
 - Level 5: Prevention of faults

23

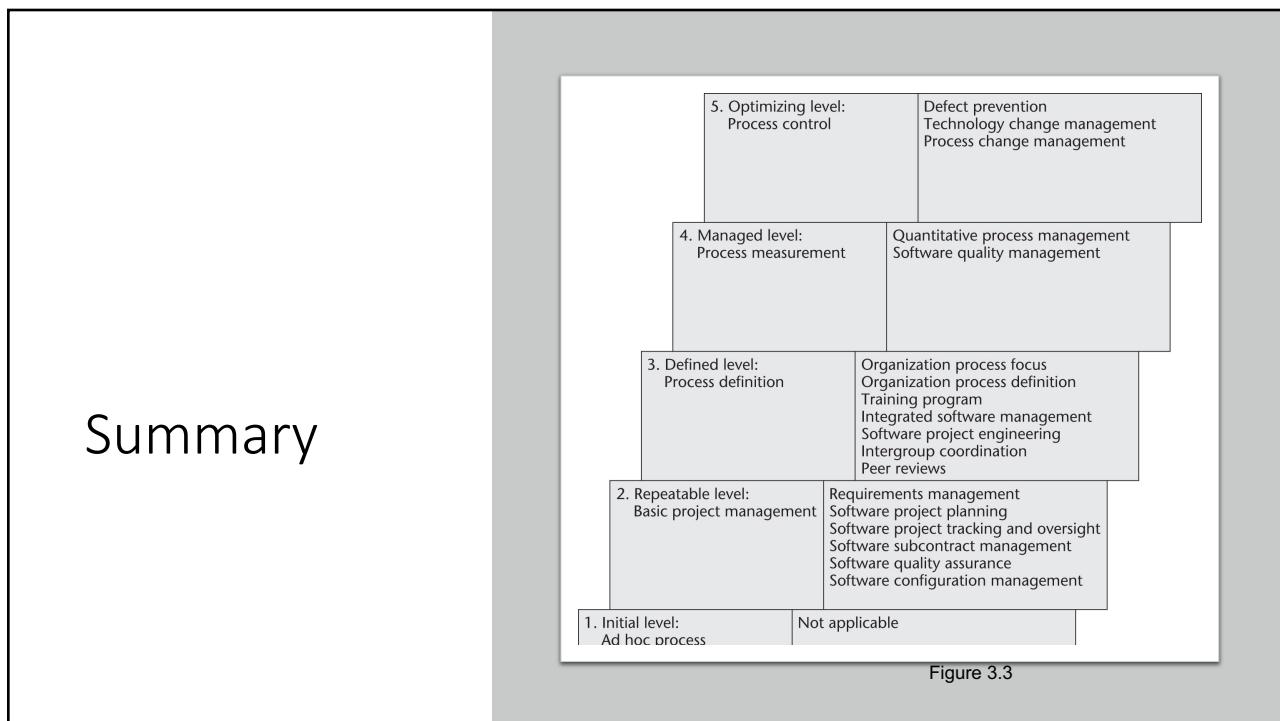


Figure 3.3

24

Goals

- Original goal:
 - Defense contracts would be awarded only to capable firms
- The U.S. Air Force stipulated that every Air Force contractor had to attain SW-CMM level 3 by 1998
 - The DoD subsequently issued a similar directive
- The CMM has now gone far beyond the limited goal of improving DoD software

25

3.14 Other Software Process Improvement Initiatives

- Other software process improvement (SPI) initiatives include:
 - ISO 9000-series
 - ISO/IEC 15504

26

ISO 9000

- A set of five standards for industrial activities
 - ISO 9001 for quality systems
 - ISO 9000-3, guidelines to apply ISO 9001 to software
 - There is an overlap with CMM, but they are not identical
 - *Not* process improvement
 - There is a stress on documenting the process
 - There is an emphasis on measurement and metrics
 - ISO 9000 is required to do business with the EU
 - Also required by many U.S. businesses, including GE
 - More and more U.S. businesses are ISO 9000 certified

27

The End

Chapter 3

CHAPTER 4

TEAMS

1

Copyright Info

- Most of these slides are from Author:
 - Stephen R Schach
 - These Slides are
 - Copyright © 2011 by The McGraw-Hill Companies
 - I have made a few modifications here and there but take no credit for the author or McGraw-Hill's work.
- Some slides are from:
 - John Barkai, Professor of Law, University of Hawaii School of Law

2

4.1 Team Organization

- A product must be completed within 3 months, but 1 person-year of programming is still needed
- Solution:
 - If one programmer can code the product in 1 year, four programmers can do it in 3 months
- Nonsense!
 - Four programmers will probably take nearly a year
 - The quality of the product is usually lower

3

Task Sharing

- If one farm hand can pick a strawberry field in 10 days, ten farm hands can pick the same strawberry field in 1 day

5

Programming Team Organization

- Example:
 - Sheila and Harry code two modules, `m1` and `m2`, say

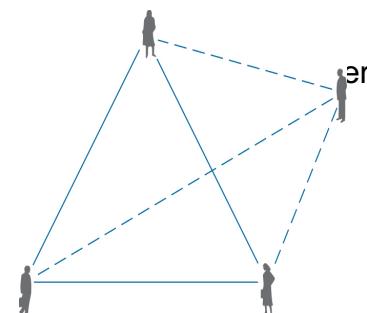
- What can go wrong
 - Both Sheila and Harry may code `m1`, and ignore `m2`
 - Sheila may code `m1`, Harry may code `m2`. When `m1` calls `m2` it passes 4 parameters; but `m2` requires 5 parameters
 - Or, the order of parameters in `m1` and `m2` may be different
 - Or, the order may be same, but the data types may be slightly different

6

Communications Problems

- Example
 - There are three channels of communication between the three programmers working on a project. The deadline is rapidly approaching but the code is not nearly complete

- “Obvious” solution:
 - Add a fourth to the team



7

Communications Problems

- But other three have to explain in detail
 - What has been accomplished
 - What is still incomplete
- Brooks' Law
 - Adding additional programming personnel to a team when a product is late has the effect of making the product even later

8

Team Organization

- Teams are used throughout the software production process
 - But especially during implementation
 - Here, the discussion is presented within the context of programming teams
- Two extreme approaches to team organization
 - Democratic teams (Weinberg, 1971)
 - Chief programmer teams (Brooks, 1971; Baker, 1972)

9

4.2 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

10

Democratic Team Approach

- Proposed solution
- Egoless programming
 - Restructure the social environment
 - Restructure programmers' values
 - Encourage team members to find faults in code
 - A fault must be considered a normal and accepted event
 - The team as whole will develop an ethos, a group identity
 - Modules will “belong” to the team as whole
 - A group of up to 10 egoless programmers constitutes a *democratic team*

11

Difficulties with Democratic Team Approach

- Management may have difficulties
 - Democratic teams are hard to introduce into an undemocratic environment
 - May not fit personality types

12

Strengths of Democratic Team Approach

- Democratic teams can be enormously productive
- They work best when the problem is difficult
- They function well in a research environment
- Problem:
 - Democratic teams have to spring up spontaneously

13

4.3 Classical Chief Programmer Team Approach

- Consider a 6-person team
 - Fifteen 2-person communication channels
 - The total number of 2-, 3-, 4-, 5-, and 6-person groups is 57
 - This team cannot do 6 person-months of work in 1 month

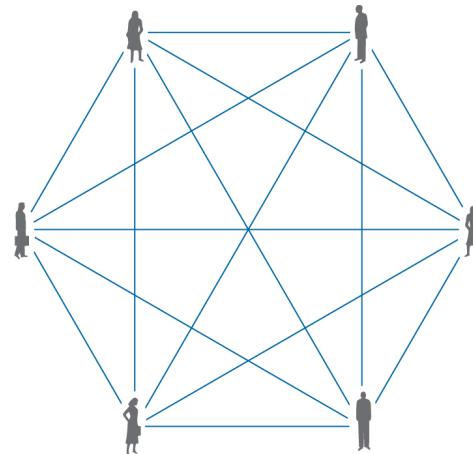


Figure 4.2

14

Classical Chief Programmer Team

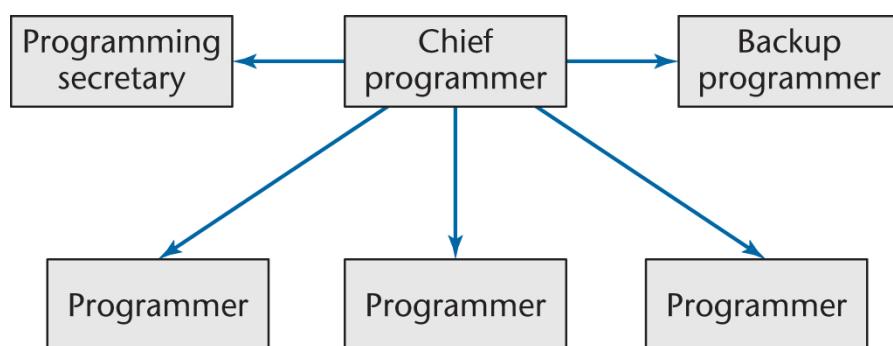


Figure 4.3

- Six programmers, but now only 5 lines of communication

15

Classical Chief Programmer Team

- Chief programmer
 - Successful manager *and* highly skilled programmer
 - Does the architectural design
 - Allocates coding among the team members
 - Writes the critical (or complex) sections of the code
 - Handles all the interfacing issues
 - Reviews the work of the other team members
 - Is personally responsible for every line of code

16

Classical Chief Programmer Team

- Programming secretary
 - A highly skilled, well paid, central member of the chief programmer team
 - Responsible for maintaining the program production library (documentation of the project), including:
 - Source code listings
 - JCL
 - Test data
 - Programmers 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!)

17

Classical Chief Programmer Team

- Programmers
 - Do nothing but program
 - All other aspects are handled by the programming secretary

18

The *New York Times* Project

- Chief programmer team concept
 - First used in 1971
 - By IBM
 - To automate the clippings data bank (“morgue”) of the *New York Times*
- Chief programmer — F. Terry Baker

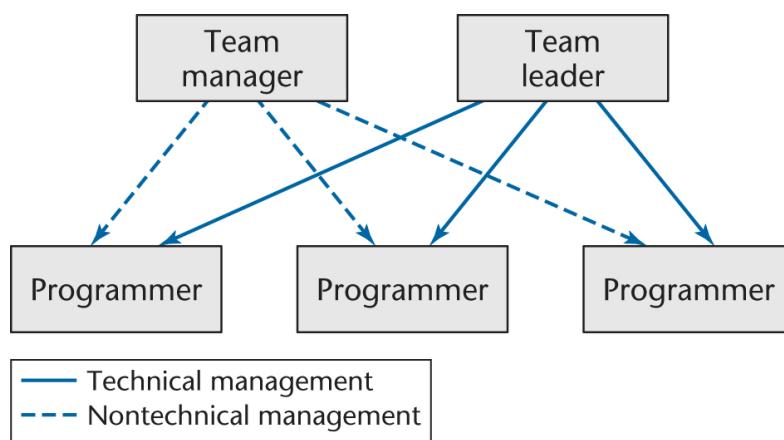
19

4.4 Beyond CP and Democratic Teams

- We need ways to organize teams that
 - Make use of the strengths of democratic teams and chief programmer teams, and
 - Can handle teams of 20 (or 120) programmers
- A strength of democratic teams
 - A positive attitude to finding faults

20

Beyond CP and Democratic Teams



- Solution
 - Reduce the managerial role of the chief programmer

Figure 4.4

21

Beyond CP and Democratic Teams

- It is easier to find a team leader than a chief programmer
- Each employee is responsible to exactly one manager — lines of responsibility are clearly delineated
- The team leader is responsible for only technical management

22

Beyond CP and Democratic Teams

- Budgetary and legal issues, and performance appraisal are not handled by the team leader
- The team leader participates in reviews — the team manager is not permitted to do so
- The team manager participates in regular team meetings to appraise the technical skills of the team members

23

Larger Projects

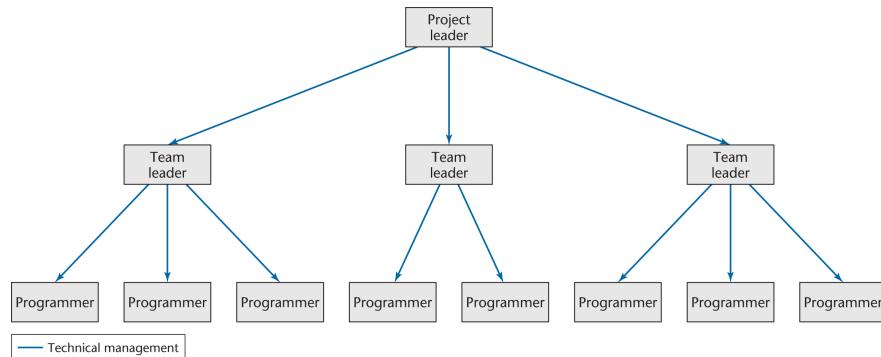


Figure 4.5

- The nontechnical side is similar
 - For even larger products, add additional layers

24

Beyond CP and Democratic Teams

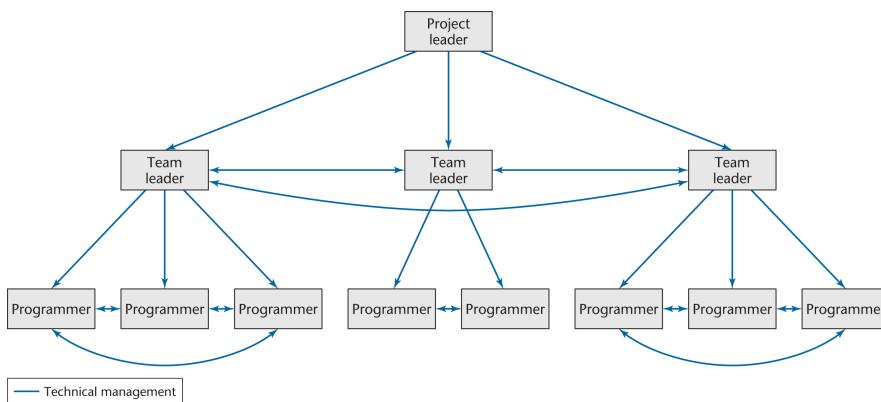


Figure 4.6

- Decentralize the decision-making process, where appropriate
 - Useful where the democratic team is good

25

4.5 Synchronize-and-Stabilize Teams

- Used by Microsoft
- Products consist of 3 or 4 sequential builds
- Small parallel teams
 - 3 to 8 developers
 - 3 to 8 testers (work one-to-one with developers)
 - The team is given the overall task specification
 - They may design the task as they wish

26

4.6 Teams For Agile Processes

- Feature of agile processes
 - All code is written by two programmers sharing a computer
 - “Pair programming”
- (This is no longer true)

27

Open-Source Programming Teams

- Individuals volunteer to take part in an open-source project for two main reasons
- Reason 1: For the sheer enjoyment of accomplishing a worthwhile task
 - In order to attract and keep volunteers, they have to view the project as “worthwhile” at all times
- Reason 2: For the learning experience

28

People Capability Maturity Model

- P-CMM is a framework for improving an organization's processes for managing and developing its workforce
- No one specific approach to team organization is put forward

29

Personality Types

- MBTI
 - Slides from Prof. John Barkai
 - William S. Richardson School of Law
 - University of Hawaii
- 4 Basic Types of Personalities

30

4.9 Choosing an Appropriate Team Organization

- There is no one solution to the problem of team organization
- The “correct” way depends on
 - The product
 - The outlook of the leaders of the organization
 - Previous experience with various team structures

44

Choosing an Appropriate Team Organization

- Exceedingly little research has been done on software team organization
 - Instead, team organization has been based on research on group dynamics in general
 - **Random??**
- Without *relevant* experimental results, it is hard to determine optimal team organization for a specific product

45

- The End Chapter 4

46

Teams

Chapter 4
Part #1

1

4.1 Team Organization

A product must be completed within 3 months, but 1 person-year of programming is still needed

- Solution:
 - If one programmer can code the product in 1 year, four programmers can do it in 3 months
- Nonsense!
 - Four programmers will probably take nearly a year
 - The quality of the product is usually lower

2

Task Sharing

- If one farm hand can pick a strawberry field in 10 days, ten farm hands can pick the same strawberry field in 1 day

3

Programming Team Organization

Example:

- Sheila and Harry code two modules, m1 and m2, say

What can go wrong

- Both Sheila and Harry may code m1, and ignore m2
- Sheila may code m1, Harry may code m2. When m1 calls m2 it passes 4 parameters; but m2 requires 5 parameters
- Or, the order of parameters in m1 and m2 may be different
- Or, the order may be same, but the data types may be slightly different

4

Communications Problems

- Example
 - There are three channels of communication between the three programmers working on a project. The deadline is rapidly approaching but the code is not nearly complete
- “Obvious” solution:
 - Add a fourth programmer to the team

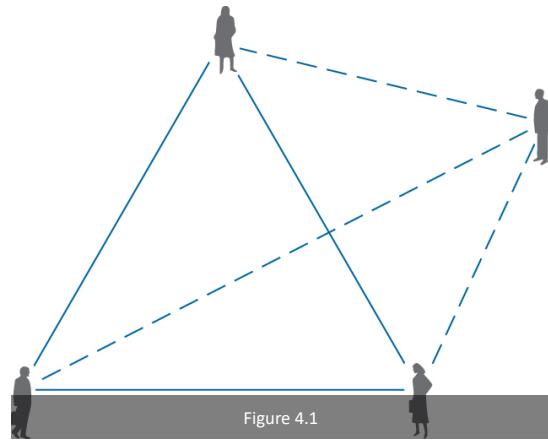


Figure 4.1

5

Communications Problems

- But other three have to explain in detail
 - What has been accomplished
 - What is still incomplete
- Brooks’ Law
 - Adding additional programming personnel to a team when a product is late has the effect of making the product even later

6

Team Organization

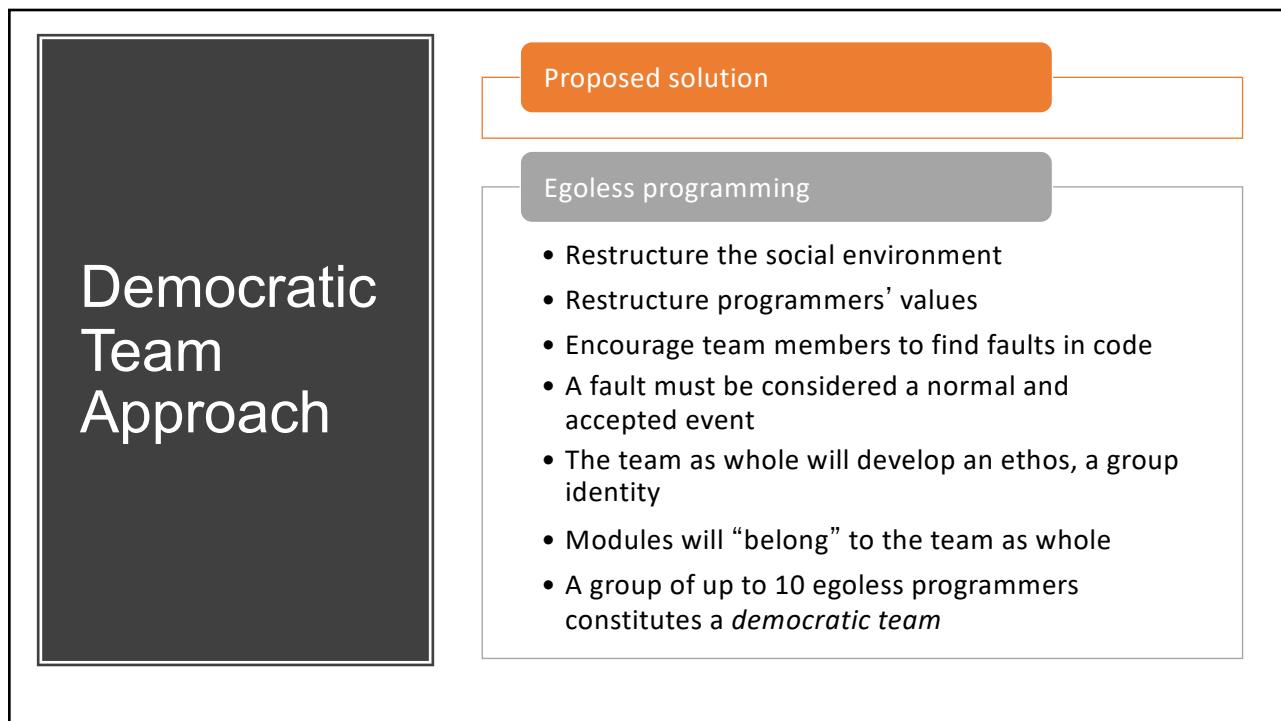
- Teams are used throughout the software production process
 - But especially during implementation
 - Here, the discussion is presented within the context of programming teams
- Two extreme approaches to team organization
 - Democratic teams (Weinberg, 1971)
 - Chief programmer teams (Brooks, 1971; Baker, 1972)

7

4.2 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

8



9

The diagram illustrates the difficulties with the Democratic Team Approach. It features a dark grey vertical bar on the left containing the title 'Difficulties with Democratic Team Approach'. To the right, a large white rectangular area contains a bulleted list of three items detailing the challenges.

Difficulties with Democratic Team Approach

- Management may have difficulties
 - Democratic teams are hard to introduce into an undemocratic environment
- May not fit personality types

10

Strengths of Democratic Team Approach

Democratic teams can be enormously productive

They work best when the problem is difficult

They function well in a research environment

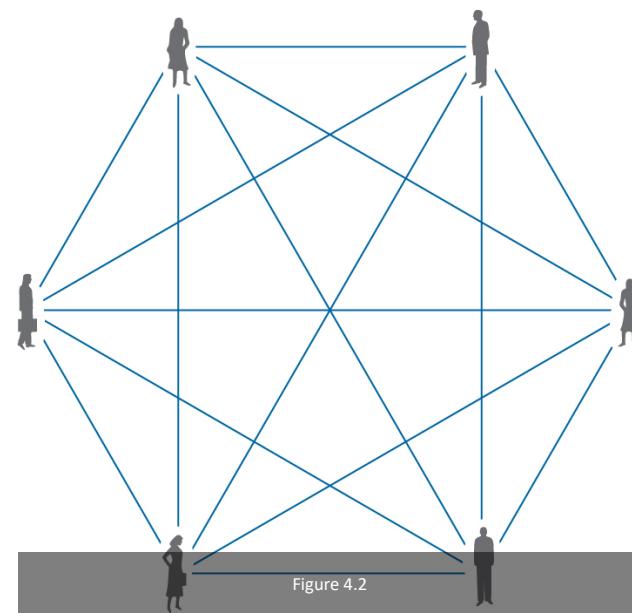
Problem:

- Democratic teams have to spring up spontaneously

11

4.3 Classical Chief Programmer Team Approach

- Consider a 6-person team
 - Fifteen 2-person communication channels
 - The total number of 2-, 3-, 4-, 5-, and 6-person groups is 57
 - This team cannot do 6 person-months of work in 1 month



12

Classical Chief Programmer Team

Six programmers, but now only 5 lines of communication

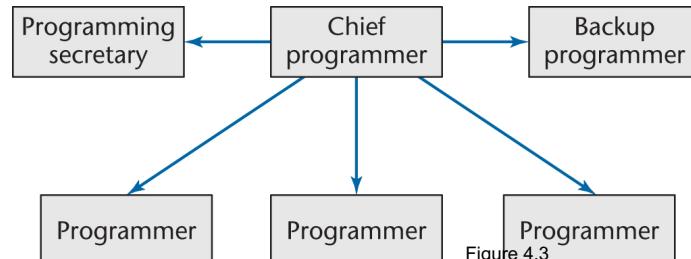


Figure 4.3

13

Classical Chief Programmer Team

- Chief programmer
 - Successful manager *and* highly skilled programmer
 - Does the architectural design
 - Allocates coding among the team members
 - Writes the critical (or complex) sections of the code
 - Handles all the interfacing issues
 - Reviews the work of the other team members
 - Is personally responsible for every line of code

14

Classical Chief Programmer Team

- Programming secretary
 - A highly skilled, well paid, central member of the chief programmer team
 - Responsible for maintaining the program production library (documentation of the project), including:
 - Source code listings
 - JCL
 - Test data
 - Programmers 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!)

15

Classical Chief Programmer Team

- Programmers
 - Do nothing but program
 - All other aspects are handled by the programming secretary

16

The New York Times Project

- Chief programmer team concept
 - First used in 1971
 - By IBM
 - To automate the clippings data bank (“morgue”) of the *New York Times*
- Chief programmer — F. Terry Baker

17

4.4 Beyond CP and Democratic Teams

- We need ways to organize teams that
 - Make use of the strengths of democratic teams and chief programmer teams, and
 - Can handle teams of 20 (or 120) programmers
- A strength of democratic teams
 - A positive attitude to finding faults

18

Beyond CP and Democratic Teams

- Solution
 - Reduce the managerial role of the chief programmer

The diagram illustrates a team structure with two managers at the top: 'Team manager' and 'Team leader'. Below them are three 'Programmer' boxes. Dashed blue arrows connect the managers to all three programmers, representing nontechnical management. Solid blue arrows connect each manager directly to their respective programmers, representing technical management.

Legend:

- Solid blue line: Technical management
- Dashed blue line: Nontechnical management

19

Beyond CP and Democratic Teams

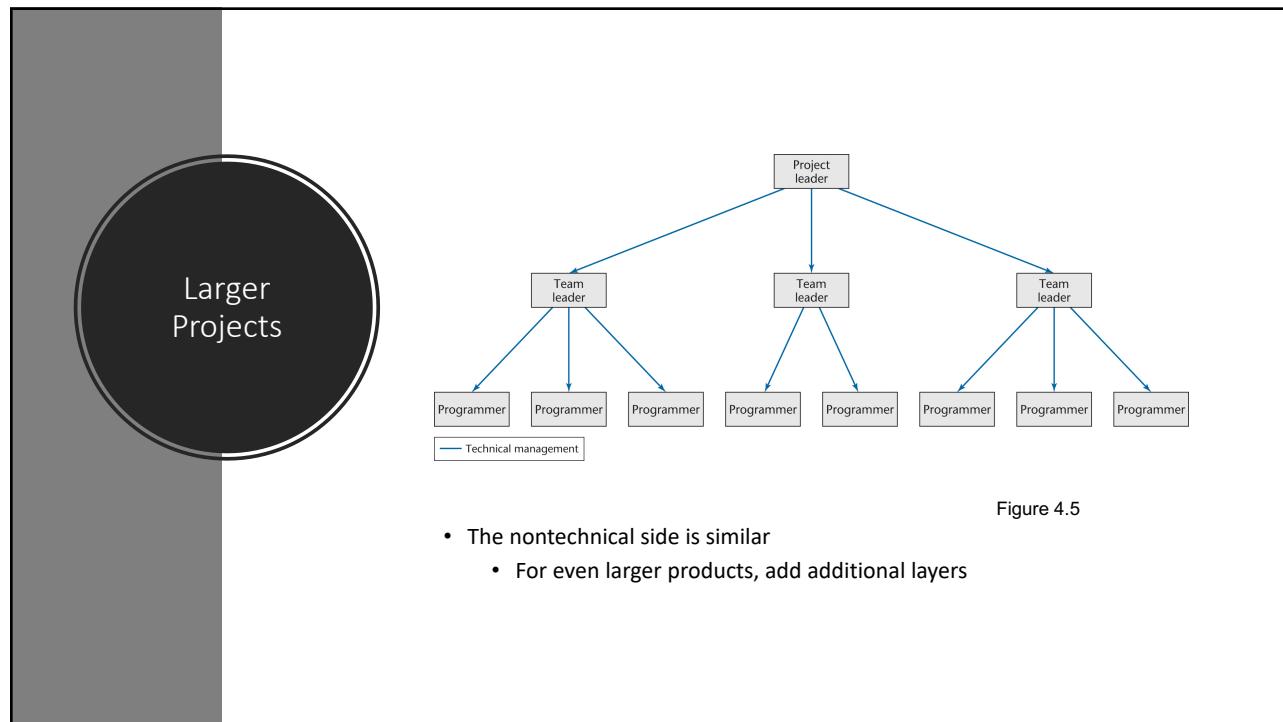
- It is easier to find a team leader than a chief programmer
- Each employee is responsible to exactly one manager — lines of responsibility are clearly delineated
- The team leader is responsible for only technical management

20

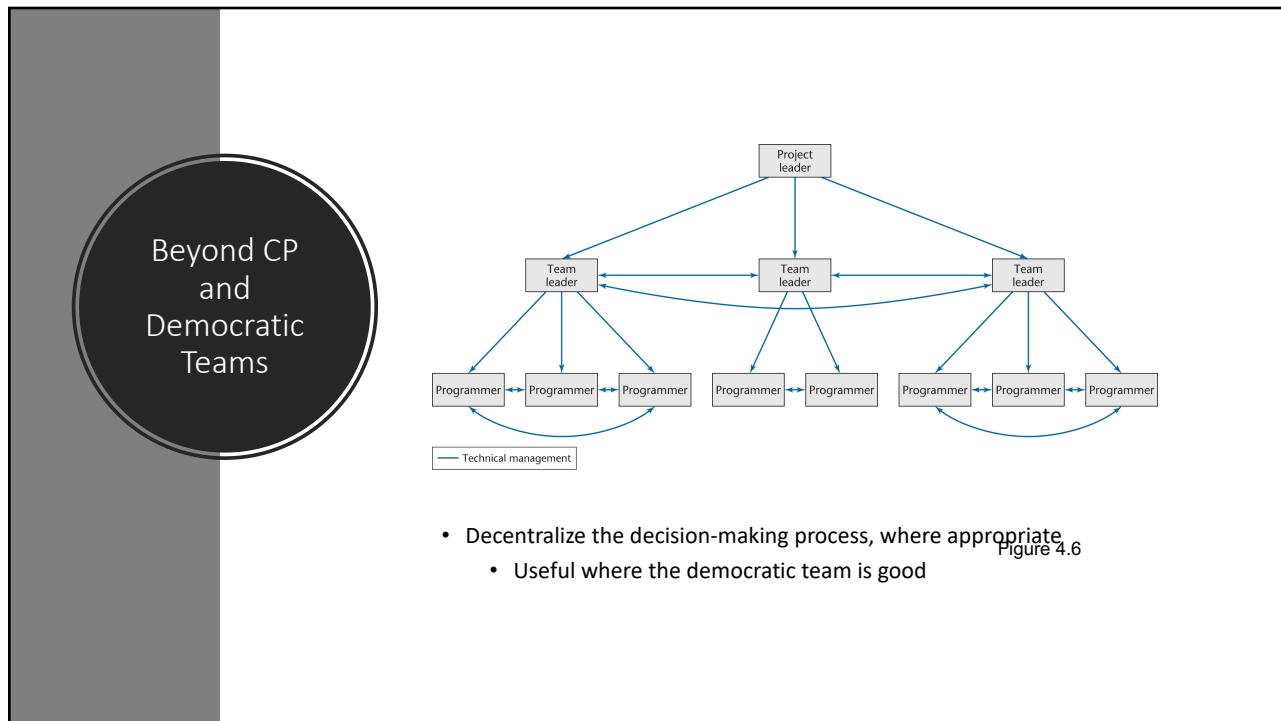
Beyond CP and Democratic Teams

- Budgetary and legal issues, and performance appraisal are not handled by the team leader
- The team leader participates in reviews — the team manager is not permitted to do so
- The team manager participates in regular team meetings to appraise the technical skills of the team members

21



22



23

4.5 Synchronize-and-Stabilize Teams

- Used by Microsoft
- Products consist of 3 or 4 sequential builds
- Small parallel teams
 - 3 to 8 developers
 - 3 to 8 testers (work one-to-one with developers)
 - The team is given the overall task specification
 - They may design the task as they wish

24

4.6 Teams For Agile Processes

- Feature of agile processes
 - All code is written by two programmers sharing a computer
 - “Pair programming”
- (This is no longer true)

25

Open-Source Programming Teams

- Individuals volunteer to take part in an open-source project for two main reasons
- Reason 1: For the sheer enjoyment of accomplishing a worthwhile task
 - In order to attract and keep volunteers, they have to view the project as “worthwhile” at all times
- Reason 2: For the learning experience

26

People Capability Maturity Model

- P-CMM is a framework for improving an organization's processes for managing and developing its workforce
- No one specific approach to team organization is put forward

27

The End!

28

MYERS BRIGGS

Chapter 4 – Teams Part 2
Working as team in Professional Environments

Some slides are from:
John Barkai, Professor of Law, University of Hawaii School of
Law

1



PART 1:
WHAT IS
MYERS
BRIGGS?

2

A Short History

- “The purpose of the Myers-Briggs Type Indicator® (MBTI®) personality inventory is to make the theory of psychological types described by C. G. Jung understandable and useful in people's lives. The essence of the theory is that much seemingly random variation in the behavior is actually quite orderly and consistent, being due to basic differences in the ways individuals prefer to use their perception and judgment.” ~The Myers & Briggs Foundation <http://www.myersbriggs.org/my-mbti-personality-type/mbti-basics/home.htm?bhcp=1>

3

Myers-Briggs Preferences

Extraversion – Introversion
Sensing – Intuitive
Thinking – Feeling
Judging - Perceiving

E – I
S – N
T – F
J - P

4



5

Description

Extraverts

Outer directed
Energy & excitement
Love "people" action

Introverts

Inner directed
Quiet
People drain them

6

Communication Style

Extraverts

- Fast & talkative
- Think out loud
- Ready, fire, aim

Introverts

- Slow & quiet
- Internally thoughtful
- Ready, ready, ready

7

Working with them

Extraverts

- Small talk is ok
- Ask open-ended questions
- Impasse?

Change energy

Introverts

- Draw them out
- Give them time to think
- Send it in writing

8

Tips for you (if happen to be..)

Extraverts

- Slow down & listen
- Warn them about you
- Get them to brainstorm

Introverts

- Be clear & forceful
- Once is not enough
- Smile

9

How do we do it?

- A short personality test that helps you identify your tendencies about working with others, and evaluating thoughts/concepts/situations.

10



PART 2: TIME TO TAKE THE EXAM!

We will be using a variation of the Myers Briggs, called 16 personalities.

Focus on the 1st four letters. Please visit this website to complete:

<https://www.16personalities.com/free-personality-test>

11



PART 3: UNDERSTANDING MYERS & BRIGGS

12

ISTJ “Take Your Time and Do It Right”	ISFJ “On My Honor, to Do My Duty...”	INFJ “Catalyst for Positive Change”	INTJ “Competence + Independence = Perfection”
ISTP “Doing the Best I Can With What I’ ve Got”	ISFP “It’ s the Thought That Counts”	INFP “Still Waters Run Deep”	INTP “Ingenious Problem Solvers”
ESTP “Let’ s Get Busy!”	ESFP “Don’ t Worry, Be Happy”	ENFP “Anything’ s Possible”	ENTP “Life’ s Entrepreneurs”
ESTJ “Taking Care of Business”	ESFJ “What Can I Do For You?”	ENFJ “The Public Relations Specialist”	ENTJ “Everything’ s Fine – I’ m in Charge”

13

Distribution of MBTI Types -Total

ISTJ 11-14%	ISFJ 9-14%	INFJ 1-3%	INTJ 2-4%
ISTP 4-6%	ISFP 5-9%	INFP 4-5%	INTP 3-5%
ESTP 4-5%	ESFP 4-9%	ENFP 6-8%	ENTP 2-5%
ESTJ 8-12%	ESFJ 9-13%	ENFJ 2-5%	ENTJ 2-5%

Source: Center of Applications of Psychological Type - 2006

14

What do these letters mean?

1. Are you outwardly or inwardly focused? If you:

- Could be described as talkative, outgoing
- Like to be in a fast-paced environment
- Tend to work out ideas with others, think out loud
- Enjoy being the center of attention

then you prefer

E
Extraversion

- Could be described as reserved, private
- Prefer a slower pace with time for contemplation
- Tend to think things through inside your head
- Would rather observe than be the center of attention

then you prefer

I
Introversion

2. How do you prefer to take in information? If you:

- Focus on the reality of how things are
- Pay attention to concrete facts and details
- Prefer ideas that have practical applications
- Like to describe things in a specific, literal way

then you prefer

S
Sensing

- Imagine the possibilities of how things could be
- Notice the big picture, see how everything connects
- Enjoy ideas and concepts for their own sake
- Like to describe things in a figurative, poetic way

then you prefer

N
Intuition

3. How do you prefer to make decisions? If you:

- Make decisions in an impersonal way, using logical reasoning
- Value justice, fairness
- Enjoy finding the flaws in an argument
- Could be described as reasonable, level-headed

then you prefer

T
Thinking

- Base your decisions on personal values and how your actions affect others
- Value harmony, forgiveness
- Like to please others and point out the best in people
- Could be described as warm, empathetic

then you prefer

F
Feeling

4. How do you prefer to live your outer life? If you:

- Prefer to have matters settled
- Think rules and deadlines should be respected
- Prefer to have detailed, step-by-step instructions
- Make plans, want to know what you're getting into

then you prefer

J
Judging

- Prefer to leave your options open
- See rules and deadlines as flexible
- Like to improvise and make things up as you go
- Are spontaneous, enjoy surprises and new situations

then you prefer

P
Perceiving

https://en.wikipedia.org/wiki/Myers%20Briggs_Type_Indicator#/media/File:MyersBriggsTypes.png

15

Google your Personality Type, and complete your “my Meyers & Briggs type”

16



PART 4: DISCUSSION

17

On Canvas Discussions

- Discuss your defined personality type or a random type:
 - #1) *What is the definition of this personality type?*
 - Strengths?
 - Weaknesses?
 - #2) *What resonates with you?*
 - #3) *What does not resonate with you?*
 - #4) *Did you find any really interesting information along the way?*
 - #5) *Can you use these findings to share your strengths in a job interview?*

18

Find your “differents”

Pick a personality that is vastly different from you.

- #1) Share a brief description of this personality
- #2) How can you use information in on different personalities in the workplace for form a better team?
- #3) When making teams, should everyone have the same/similar classifications, or should groups be a mix of personalities?

19



PART 5: “MORE OF THE REAL WORLD”

20

What's Your Personality Type?

Use the questions on the outside of the chart to determine the four letters of your Myers-Briggs type.
For each pair of letters, choose the side that seems most natural to you, even if you don't agree with every description.

<p>1. Are you outwardly or inwardly focused? If you:</p> <ul style="list-style-type: none"> • Could be described as talkative, outgoing • Like to be in a fast-paced environment • Tend to work out ideas with others, think out loud • Enjoy being the center of attention <p>then you prefer E Extraversion</p>	<p>ISTJ Responsible, sincere, analytical, reserved, realistic, systematic. Hardworking and trustworthy with sound practical judgment.</p> <p>ISFP Gentle, sensitive, nurturing, helpful, flexible, spontaneous. Enjoy adventure, skilled at understanding how mechanical things work.</p>	<p>INFJ Warm, considerate, gentle, compassionate, gentle. Devoted caretakers who enjoy being helpful to others.</p> <p>INFP Idealistic, perceptive, caring, loving. Value inner harmony and personal growth, focus on dreams and possibilities.</p>	<p>INTJ Innovative, independent, strategic, logical, reserved, insightful. Driven by their own original ideas to achieve improvements.</p> <p>INTP Intellectual, logical, precise, reserved, flexible, imaginative. Original thinkers who enjoy speculation and creative problem solving.</p>	<p>3. How do you prefer to make decisions? If you:</p> <ul style="list-style-type: none"> • Could be described as reasonable, level-headed • Make decisions in an impersonal way, using logical reasoning • Value justice, fairness • Enjoy finding the flaws in an argument <p>then you prefer T Thinking</p>
<p>2. How do you prefer to take in information? If you:</p> <ul style="list-style-type: none"> • Focus on the reality of how things are • Pay attention to concrete facts and details • Prefer ideas that have practical applications • Like to describe things in a specific, literal way <p>then you prefer S Sensing</p>	<p>ESTP Outgoing, realistic, action-oriented, curious, versatile, spontaneous. Pragmatic problem solvers and skillful negotiators.</p> <p>ESTJ Efficient, outgoing, analytical, systematic, dependable, realistic. Like to run the show and get things done in an orderly fashion.</p>	<p>ESFP Playful, enthusiastic, friendly, spontaneous, tactful, flexible. Have strong common sense, enjoy helping people in tangible ways.</p> <p>ESFJ Friendly, outgoing, reliable, conscientious, organized, practical. Seek to be helpful and please others, enjoy being active and productive.</p>	<p>ENFP Enthusiastic, creative, spontaneous, optimistic, supportive, playful. Value inspiration, enjoy starting new projects, see potential in others.</p> <p>ENFJ Caring, enthusiastic, idealistic, organized, diplomatic, responsible. Skilled communicators who value connection with people.</p>	<p>4. How do you prefer to live your outer life? If you:</p> <ul style="list-style-type: none"> • Make plans, want to know what you're getting into • Prefer to leave your options open • Think rules and deadlines should be respected • Prefer to have detailed, step-by-step instructions • Are spontaneous, enjoy surprises and new situations <p>then you prefer J Judging</p>
				<p>then you prefer F Feeling</p>

https://en.wikipedia.org/wiki/Myers%20-%20Briggs_Type_Indicator#/media/File:MyersBriggsTypes.png

21

Your Ideal Manager

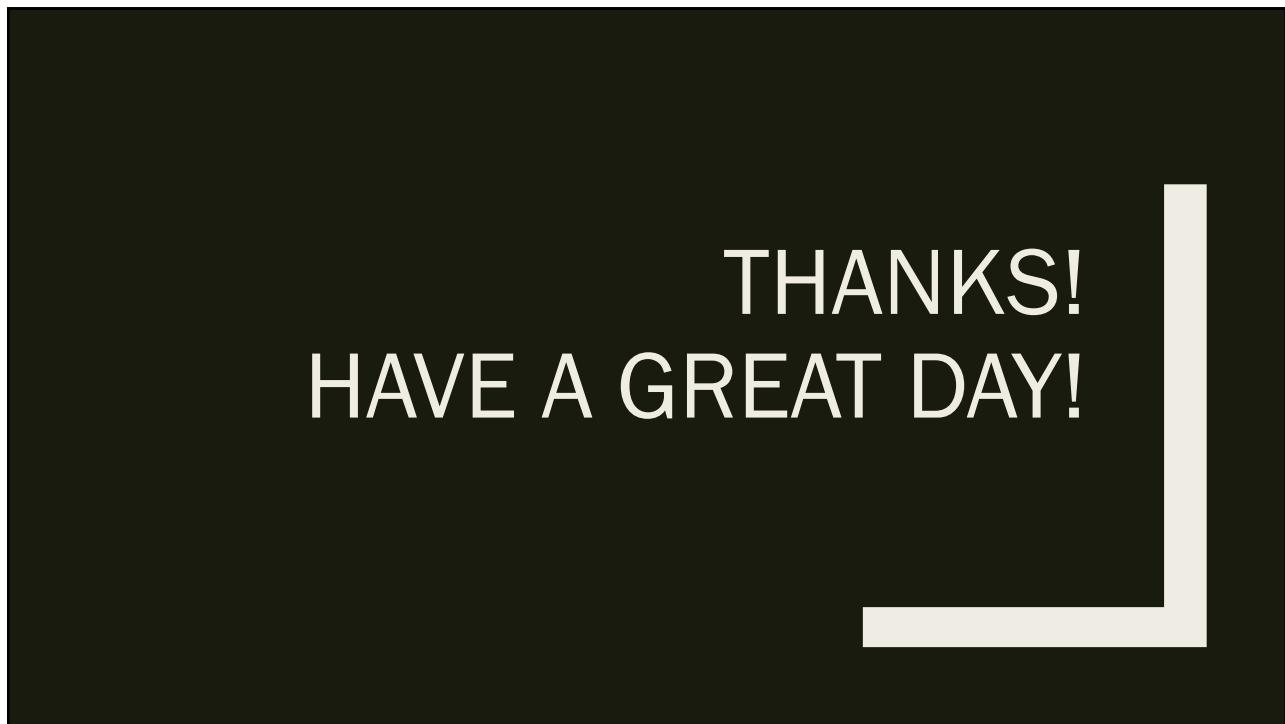
■ What personality will your ideal manager have??

22



23

PART 6: WRAP UP



24

Chapter 5

Tools of the Trade!

5.1 Stepwise Refinement

- A basic principle underlying many software engineering techniques
 - “Postpone decisions as to details as late as possible to be able to concentrate on the important issues”
- Miller’s law (1956)
 - A human being can concentrate on 7 ± 2 items at a time

5.2 Cost–Benefit Analysis

- Compare costs and future benefits
 - Estimate costs
 - Estimate benefits
 - State all assumptions explicitly

Benefits		Costs	
Salary savings (7 years)	1,575,000	Hardware and software (7 years)	1,250,000
Improved cash flow (7 years)	875,000	Conversion cost (first year only)	350,000
		Explanations to customers (first year only)	125,000
Total benefits	\$2,450,000	Total costs	\$1,725,000

Figure 5.8

Cost–Benefit Analysis

Example: Computerizing KCEC

5.3 Divide-and-Conquer

- Solve a large, hard problem by breaking up into smaller subproblems that hopefully will be easier to solve
- Divide-and-conquer is used in the Unified Process to handle a large, complex system
 - Analysis workflow
 - Partition the software product into analysis *packages*
 - Design workflow
 - Break up the upcoming implementation workflow into manageable pieces, termed *subsystems*

5.4

Separation of Concerns

- The process of breaking a software product into components with minimal overlap of functionality
 - Minimizes regression faults
 - Promotes reuse
- Separation of concerns underlies much of software engineering

5.5 Software Metrics

- To detect problems early, it is essential to measure
- Examples:
 - LOC per month
 - Defects per 1000 lines of code

The Five Basic Metrics



Size

In lines of code, or better



Cost

In dollars



Duration

In months



Effort

In person months



Quality

Number of faults detected

5.6 CASE (Computer- Aided Software Engineering)

- Scope of CASE
 - CASE can support the entire life-cycle
 - The computer assists with drudge work
 - It manages all the details

Operating System Front-End in Editor

- Single command
 - go or run
 - Use of the mouse to choose
 - An icon, or
 - A menu selection
- This one command causes the editor to invoke the compiler, linker, loader, and execute the product

Source Level Debugger

- Example:
 - Product executes terminates abruptly and prints
Overflow at 4B06
 - or
 - Core dumped
 - or
 - Segmentation fault

Programming Workbench

- Structure editor with
 - Online interface checking capabilities
 - Operating system front-end
 - Online documentation
 - Source level debugger
- This constitutes a simple programming environment

5.9 Software Versions

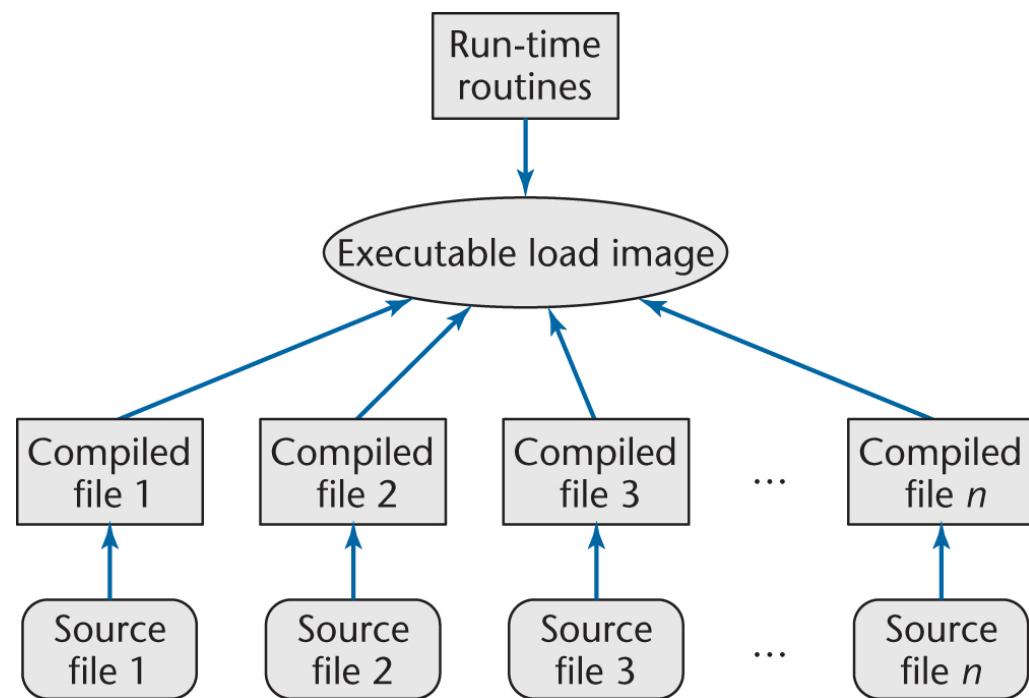
- During maintenance, at all times there are at least two versions of the product:
 - The old version, and
 - The new version
- There are two types of versions: *revisions* and *variations*
- *Revisions are a change to the existing product*
- *Variations are changes made to have in run on new platforms*

5.9.1 Revisions

- Revision
 - A version to fix a fault in the artifact
 - We cannot throw away an incorrect version
 - The new version may be no better
 - Some sites may not install the new version
- Perfective and adaptive maintenance also result in revisions

5.10 Configuration Control

- Every code artifact exists in three forms
 - Source code
 - Compiled code
 - Executable load image
- Configuration
 - A version of each artifact from which a given version of a product is built



Version- Control Tool

- Essential for programming-in-the-many
 - **A first step toward configuration management**
- A version-control tool must handle
 - Updates
 - Parallel versions
 - branches

Version- Control Tools

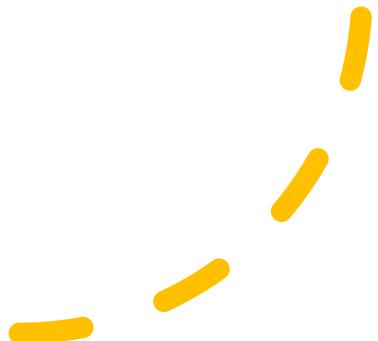
- UNIX version-control tools
 - *sccs*
 - *rcs*
 - *cvs*
- Popular commercial configuration-control tools
 - PVCS
 - SourceSafe
- Open-source configuration-control tools
 - *cvs*
 - Subversion
 - git

5.11 Build Tools

- Example
 - UNIX *make*
- A build tool compares the date and time stamp on
 - Source code, compiled code
 - It calls the appropriate compiler only if necessary
- The tool then compares the date and time stamp on
 - Compiled code, executable load image
 - It calls the linker only if necessary



- The end!



Chapter 5

Tools of the Trade!

5.1 Stepwise Refinement

- A basic principle underlying many software engineering techniques
 - “Postpone decisions as to details as late as possible to be able to concentrate on the important issues”
- Miller’s law (1956)
 - A human being can concentrate on 7 ± 2 items at a time

5.2 Cost–Benefit Analysis

- Compare costs and future benefits
 - Estimate costs
 - Estimate benefits
 - State all assumptions explicitly

Benefits		Costs	
Salary savings (7 years)	1,575,000	Hardware and software (7 years)	1,250,000
Improved cash flow (7 years)	875,000	Conversion cost (first year only)	350,000
		Explanations to customers (first year only)	125,000
Total benefits	\$2,450,000	Total costs	\$1,725,000

Figure 5.8

Cost–Benefit Analysis

Example: Computerizing KCEC

5.3 Divide-and-Conquer

- Solve a large, hard problem by breaking up into smaller subproblems that hopefully will be easier to solve
- Divide-and-conquer is used in the Unified Process to handle a large, complex system
 - Analysis workflow
 - Partition the software product into analysis *packages*
 - Design workflow
 - Break up the upcoming implementation workflow into manageable pieces, termed *subsystems*

5.4

Separation of Concerns

- The process of breaking a software product into components with minimal overlap of functionality
 - Minimizes regression faults
 - Promotes reuse
- Separation of concerns underlies much of software engineering

5.5 Software Metrics

- To detect problems early, it is essential to measure
- Examples:
 - LOC per month
 - Defects per 1000 lines of code

The Five Basic Metrics



Size

In lines of code, or better



Cost

In dollars



Duration

In months



Effort

In person months



Quality

Number of faults detected

5.6 CASE (Computer- Aided Software Engineering)

- Scope of CASE
 - CASE can support the entire life-cycle
 - The computer assists with drudge work
 - It manages all the details

Operating System Front-End in Editor

- Single command
 - go or run
 - Use of the mouse to choose
 - An icon, or
 - A menu selection
- This one command causes the editor to invoke the compiler, linker, loader, and execute the product

Source Level Debugger

- Example:
 - Product executes terminates abruptly and prints
Overflow at 4B06
 - or
 - Core dumped
 - or
 - Segmentation fault

Programming Workbench

- Structure editor with
 - Online interface checking capabilities
 - Operating system front-end
 - Online documentation
 - Source level debugger
- This constitutes a simple programming environment

5.9 Software Versions

- During maintenance, at all times there are at least two versions of the product:
 - The old version, and
 - The new version
- There are two types of versions: *revisions* and *variations*
 - *Revisions are a change to the existing product*
 - *Variations are changes made to have in run on new platforms*

5.9.1 Revisions

- Revision
 - A version to fix a fault in the artifact
 - We cannot throw away an incorrect version
 - The new version may be no better
 - Some sites may not install the new version
- Perfective and adaptive maintenance also result in revisions

5.10 Configuration Control

- Every code artifact exists in three forms
 - Source code
 - Compiled code
 - Executable load image
- Configuration
 - A version of each artifact from which a given version of a product is built

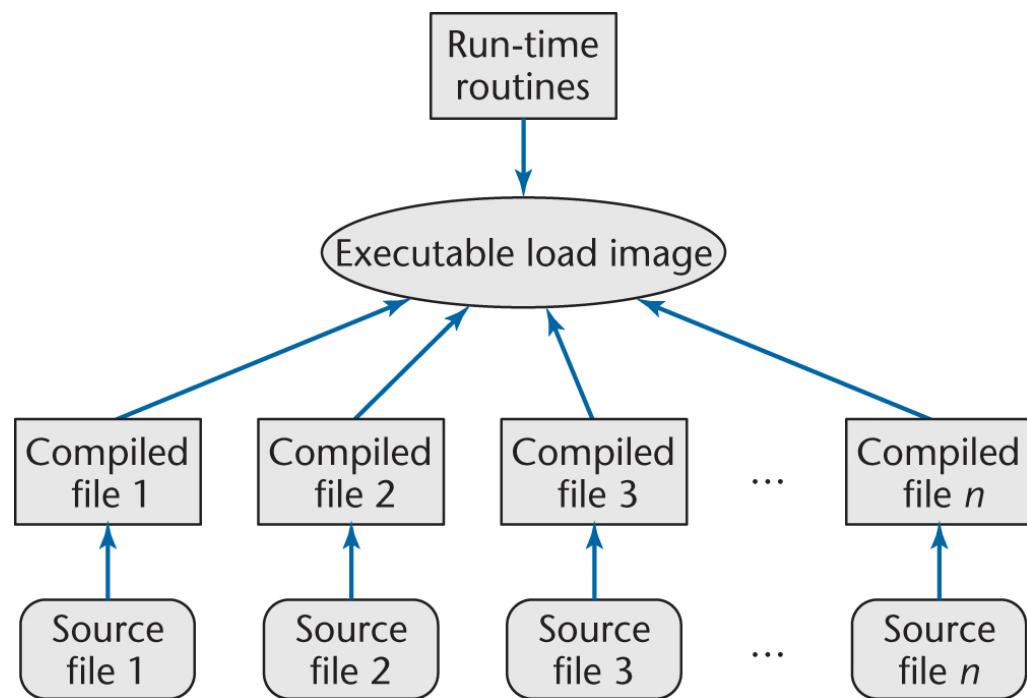


Figure 5.12

Version- Control Tool

- Essential for programming-in-the-many
 - **A first step toward configuration management**
- A version-control tool must handle
 - Updates
 - Parallel versions
 - branches

Version- Control Tools

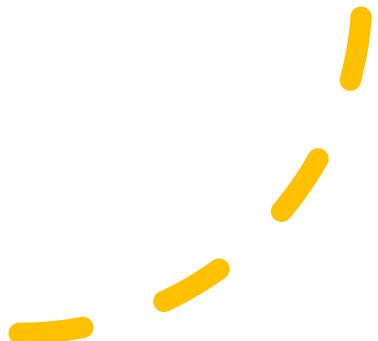
- UNIX version-control tools
 - *sccs*
 - *rcs*
 - *cvs*
- Popular commercial configuration-control tools
 - PVCS
 - SourceSafe
- Open-source configuration-control tools
 - *cvs*
 - Subversion
 - git

5.11 Build Tools

- Example
 - UNIX *make*
- A build tool compares the date and time stamp on
 - Source code, compiled code
 - It calls the appropriate compiler only if necessary
- The tool then compares the date and time stamp on
 - Compiled code, executable load image
 - It calls the linker only if necessary



- The end!



Chapter 5

Tools of the Trade!

5.1 Stepwise Refinement

- A basic principle underlying many software engineering techniques
 - “Postpone decisions as to details as late as possible to be able to concentrate on the important issues”
- Miller’s law (1956)
 - A human being can concentrate on 7 ± 2 items at a time

5.2 Cost–Benefit Analysis

- Compare costs and future benefits
 - Estimate costs
 - Estimate benefits
 - State all assumptions explicitly

Benefits		Costs	
Salary savings (7 years)	1,575,000	Hardware and software (7 years)	1,250,000
Improved cash flow (7 years)	875,000	Conversion cost (first year only)	350,000
		Explanations to customers (first year only)	125,000
Total benefits	\$2,450,000	Total costs	\$1,725,000

Figure 5.8

Cost–Benefit Analysis

Example: Computerizing KCEC

5.3 Divide-and-Conquer

- Solve a large, hard problem by breaking up into smaller subproblems that hopefully will be easier to solve
- Divide-and-conquer is used in the Unified Process to handle a large, complex system
 - Analysis workflow
 - Partition the software product into analysis *packages*
 - Design workflow
 - Break up the upcoming implementation workflow into manageable pieces, termed *subsystems*

5.4

Separation of Concerns

- The process of breaking a software product into components with minimal overlap of functionality
 - Minimizes regression faults
 - Promotes reuse
- Separation of concerns underlies much of software engineering

5.5 Software Metrics

- To detect problems early, it is essential to measure
- Examples:
 - LOC per month
 - Defects per 1000 lines of code

The Five Basic Metrics



Size

In lines of code, or better



Cost

In dollars



Duration

In months



Effort

In person months



Quality

Number of faults detected

5.6 CASE (Computer- Aided Software Engineering)

- Scope of CASE
 - CASE can support the entire life-cycle
 - The computer assists with drudge work
 - It manages all the details

Operating System Front-End in Editor

- Single command
 - go or run
 - Use of the mouse to choose
 - An icon, or
 - A menu selection
- This one command causes the editor to invoke the compiler, linker, loader, and execute the product

Source Level Debugger

- Example:
 - Product executes terminates abruptly and prints
Overflow at 4B06
 - or
 - Core dumped
 - or
 - Segmentation fault

Programming Workbench

- Structure editor with
 - Online interface checking capabilities
 - Operating system front-end
 - Online documentation
 - Source level debugger
- This constitutes a simple programming environment

5.9 Software Versions

- During maintenance, at all times there are at least two versions of the product:
 - The old version, and
 - The new version
- There are two types of versions: *revisions* and *variations*
 - *Revisions are a change to the existing product*
 - *Variations are changes made to have in run on new platforms*

5.9.1 Revisions

- Revision
 - A version to fix a fault in the artifact
 - We cannot throw away an incorrect version
 - The new version may be no better
 - Some sites may not install the new version
- Perfective and adaptive maintenance also result in revisions

5.10 Configuration Control

- Every code artifact exists in three forms
 - Source code
 - Compiled code
 - Executable load image
- Configuration
 - A version of each artifact from which a given version of a product is built

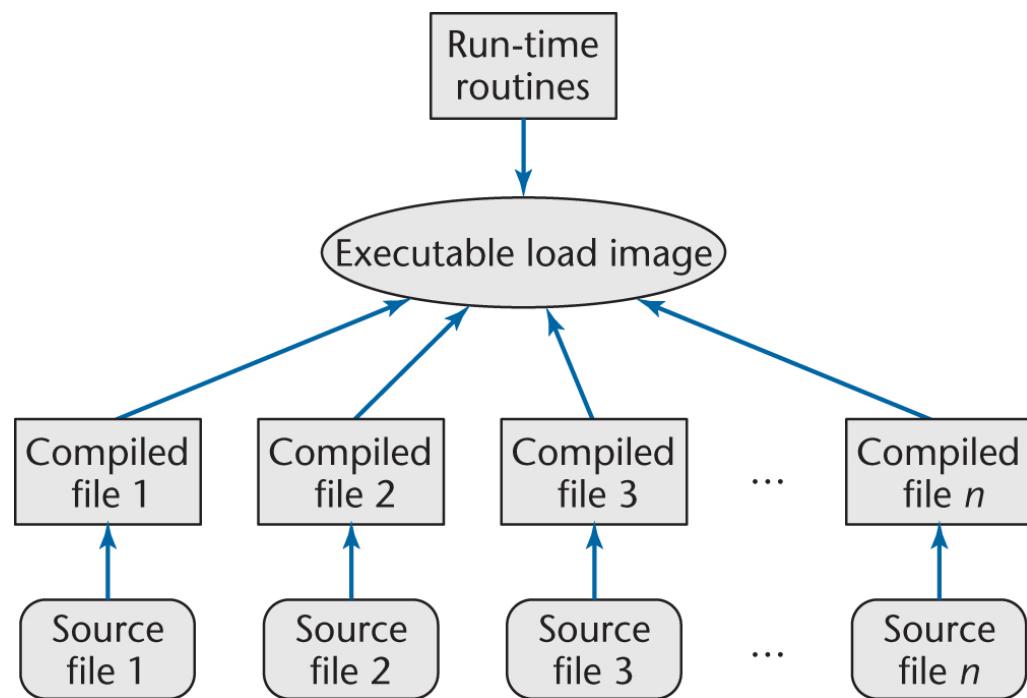


Figure 5.12

Version- Control Tool

- Essential for programming-in-the-many
 - **A first step toward configuration management**
- A version-control tool must handle
 - Updates
 - Parallel versions
 - branches

Version- Control Tools

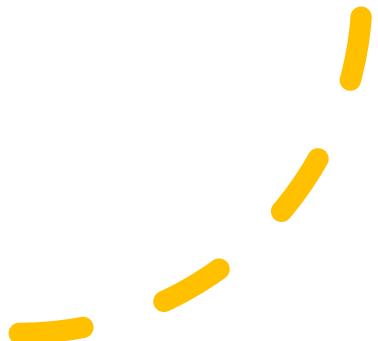
- UNIX version-control tools
 - *sccs*
 - *rcs*
 - *cvs*
- Popular commercial configuration-control tools
 - PVCS
 - SourceSafe
- Open-source configuration-control tools
 - *cvs*
 - Subversion
 - git

5.11 Build Tools

- Example
 - UNIX *make*
- A build tool compares the date and time stamp on
 - Source code, compiled code
 - It calls the appropriate compiler only if necessary
- The tool then compares the date and time stamp on
 - Compiled code, executable load image
 - It calls the linker only if necessary



- The end!



Chapter 5

Tools of the Trade!

5.1 Stepwise Refinement

- A basic principle underlying many software engineering techniques
 - “Postpone decisions as to details as late as possible to be able to concentrate on the important issues”
- Miller’s law (1956)
 - A human being can concentrate on 7 ± 2 items at a time

5.2 Cost–Benefit Analysis

- Compare costs and future benefits
 - Estimate costs
 - Estimate benefits
 - State all assumptions explicitly

Benefits		Costs	
Salary savings (7 years)	1,575,000	Hardware and software (7 years)	1,250,000
Improved cash flow (7 years)	875,000	Conversion cost (first year only)	350,000
		Explanations to customers (first year only)	125,000
Total benefits	\$2,450,000	Total costs	\$1,725,000

Figure 5.8

Cost–Benefit Analysis

Example: Computerizing KCEC

5.3 Divide-and-Conquer

- Solve a large, hard problem by breaking up into smaller subproblems that hopefully will be easier to solve
- Divide-and-conquer is used in the Unified Process to handle a large, complex system
 - Analysis workflow
 - Partition the software product into analysis *packages*
 - Design workflow
 - Break up the upcoming implementation workflow into manageable pieces, termed *subsystems*

5.4

Separation of Concerns

- The process of breaking a software product into components with minimal overlap of functionality
 - Minimizes regression faults
 - Promotes reuse
- Separation of concerns underlies much of software engineering

5.5 Software Metrics

- To detect problems early, it is essential to measure
- Examples:
 - LOC per month
 - Defects per 1000 lines of code

The Five Basic Metrics



Size

In lines of code, or better



Cost

In dollars



Duration

In months



Effort

In person months



Quality

Number of faults detected

5.6 CASE (Computer- Aided Software Engineering)

- Scope of CASE
 - CASE can support the entire life-cycle
 - The computer assists with drudge work
 - It manages all the details

Operating System Front-End in Editor

- Single command
 - go or run
 - Use of the mouse to choose
 - An icon, or
 - A menu selection
- This one command causes the editor to invoke the compiler, linker, loader, and execute the product

Source Level Debugger

- Example:
 - Product executes terminates abruptly and prints
Overflow at 4B06
 - or
 - Core dumped
 - or
 - Segmentation fault

Programming Workbench

- Structure editor with
 - Online interface checking capabilities
 - Operating system front-end
 - Online documentation
 - Source level debugger
- This constitutes a simple programming environment

5.9 Software Versions

- During maintenance, at all times there are at least two versions of the product:
 - The old version, and
 - The new version
- There are two types of versions: *revisions* and *variations*
 - *Revisions are a change to the existing product*
 - *Variations are changes made to have in run on new platforms*

5.9.1 Revisions

- Revision
 - A version to fix a fault in the artifact
 - We cannot throw away an incorrect version
 - The new version may be no better
 - Some sites may not install the new version
- Perfective and adaptive maintenance also result in revisions

5.10 Configuration Control

- Every code artifact exists in three forms
 - Source code
 - Compiled code
 - Executable load image
- Configuration
 - A version of each artifact from which a given version of a product is built

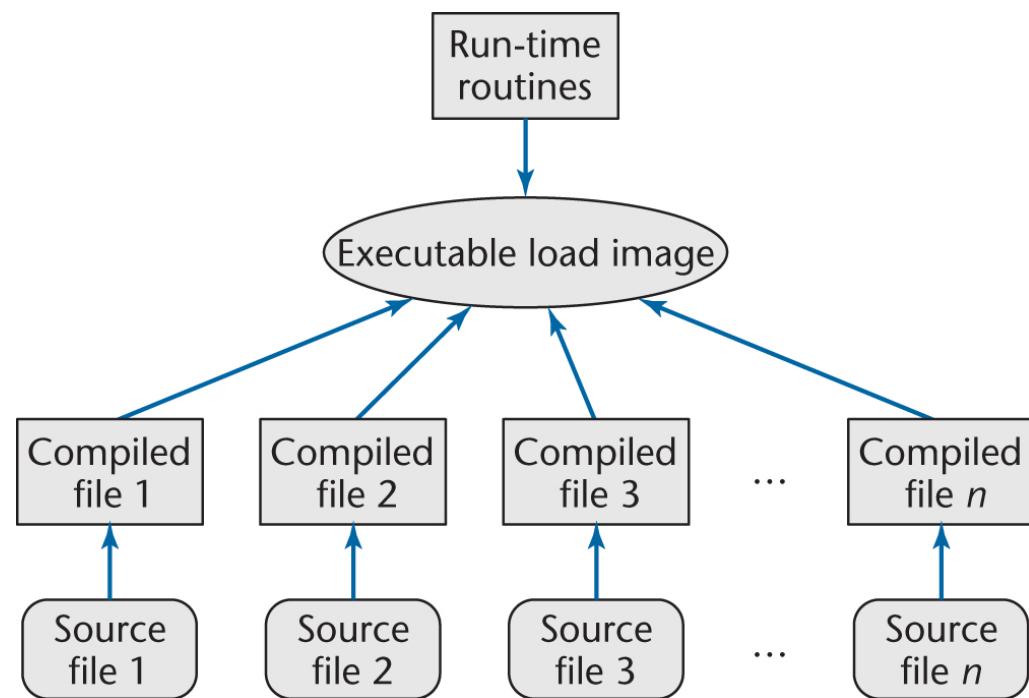


Figure 5.12

Version- Control Tool

- Essential for programming-in-the-many
 - **A first step toward configuration management**
- A version-control tool must handle
 - Updates
 - Parallel versions
 - branches

Version- Control Tools

- UNIX version-control tools
 - *sccs*
 - *rcs*
 - *cvs*
- Popular commercial configuration-control tools
 - PVCS
 - SourceSafe
- Open-source configuration-control tools
 - *cvs*
 - Subversion
 - git

5.11 Build Tools

- Example
 - UNIX *make*
- A build tool compares the date and time stamp on
 - Source code, compiled code
 - It calls the appropriate compiler only if necessary
- The tool then compares the date and time stamp on
 - Compiled code, executable load image
 - It calls the linker only if necessary



- The end!



Chapter 6

Testing

(slides from Stephen R. Schach, McGraw Hil)

1

Testing

- “V & V”
 - *Verification*
 - Determine if the workflow was completed correctly
 - *Validation*
 - Determine if the product as a whole satisfies its requirements
 - i.e. is it right
- There are two basic types of testing
 - Execution-based testing
 - Non-execution-based testing

2

6.1 Quality

- Extent to which product satisfies requirements/specifications.
- Software professionals built this in
 - not “added” after by sqa.
- Fault is based on human error.
- Failure is the observed incorrect behavior
- Error is the amount which the result is incorrect

3

6.1.1 Software Quality Assurance

- The members of the SQA group must ensure that the developers are doing high-quality work
 - At the end of each workflow
 - When the product is complete
- In addition, quality assurance must be applied to
 - The process itself
 - Example: Standards

4

6.1.2 Managerial Independence

- There must be managerial independence between
 - The development group
 - The SQA group
- Neither group should have power over the other

5

Managerial Independence (contd)

- More senior management must decide whether to
 - **Deliver the product on time but with faults**, or
 - Test further and deliver the product late
- The decision must take into account the interests of the client and the development organization

6

6.2 Non-Execution Based Testing

- Review review review!
- Underlying principles
 - We should **not** review our own work
 - **We should not be the only one**
 - Group synergy

7

6.2.1 Walkthroughs ("informal")

- A walkthrough team consists of from four to six members
- It includes representatives of
 - The team responsible for the current workflow
 - The team responsible for the next workflow
 - The SQA group
- The walkthrough is preceded by preparation
 - Lists of items
 - Items not understood
 - Items that appear to be incorrect

8

6.2.2 Managing Walkthroughs

- The walkthrough team is chaired by the SQA representative
- In a walkthrough we detect faults, not correct them
 - A correction produced by a committee is likely to be of low quality
 - The cost of a committee correction is too high
 - Not all items flagged are actually incorrect
 - A walkthrough should not last longer than 2 hours
 - There is no time to correct faults as well

9

Managing Walkthroughs

- A walkthrough must be document-driven, rather than participant-driven
- Verbalization leads to fault finding
- A walkthrough should never be used for performance appraisal

10

6.2.3 Inspections (“more formal”)

- An inspection has five formal steps
 - Overview
 - Preparation, aided by statistics of fault types
 - Inspection
 - Rework
 - Follow-up

11

Inspections

- An inspection team has four members
 - Moderator
 - A member of the team performing the current workflow
 - A member of the team performing the next workflow
 - A member of the SQA group
- Special roles are played by the
 - Moderator
 - Reader
 - Recorder

12

Fault Statistics (page 160-161)

- Faults are recorded by severity
 - Example:
 - Major or minor
- Faults are recorded by fault type
 - Examples of design faults:
 - Not all specification items have been addressed
 - Actual and formal arguments do not correspond

13

Fault Statistics (contd)

- For a given workflow, we compare current fault rates with those of previous products
- We take action if there are a disproportionate number of faults in an artifact
 - Redesigning from scratch is a good alternative
- We carry forward fault statistics to the next workflow
 - We may not detect all faults of a particular type in the current inspection

14

Statistics on Inspections

- IBM inspections showed up
 - 82% of all detected faults (1976)
 - 70% of all detected faults (1978)
 - 93% of all detected faults (1986)
- Switching system
 - 90% decrease in the cost of detecting faults (1986)
- JPL
 - Four major faults, 14 minor faults per 2 hours (1990)
 - Savings of \$25,000 *per inspection*
 - The number of faults decreased exponentially by phase (1992)

15

Statistics on Inspections

- Warning
- Fault statistics should never be used for performance appraisal
 - “Killing the goose that lays the golden eggs”

16

6.2.4 Comparison of Inspections and Walkthroughs (KNOW THE DIFFERENCE)

- Walkthrough
 - Two-step, **informal** process
 - Preparation
 - Analysis
- Inspection
 - Five-step, **formal** process
 - Overview
 - Preparation
 - Inspection
 - Rework
 - Follow-up

17

6.2.5 Strengths and Weaknesses of Reviews

- Reviews can be effective
 - Faults are detected early in the process
- Reviews are less effective if the process is inadequate
 - Large-scale software should consist of smaller, largely independent pieces
 - **The documentation of the previous workflows has to be complete and available online**

18

6.2.6 Metrics for Inspections

- Inspection rate (e.g., design pages inspected per hour)
- Fault density (e.g., faults per KLOC inspected)
- Fault detection rate (e.g., faults detected per hour)
- Fault detection efficiency (e.g., number of major, minor faults detected per hour)
- **Why is this important??**

19

Metrics for Inspections

- So what do you do if you see a 50% increase in the fault detection rate?

20

6.3 Execution-Based Testing

- Organizations spend up to 50% of their software budget on testing
 - But delivered software is frequently unreliable
- Dijkstra (1972)
 - “Program testing can be a very effective way to show the presence of bugs, **but it is hopelessly inadequate for showing their absence**”

21

6.4 What Should Be Tested?

- Definition of *execution-based testing*
 - “The process of inferring certain behavioral properties of the product based, in part, on the results of executing the product in a known environment with selected inputs”
- This definition has troubling implications

22

6.4 What Should Be Tested?

- “Inference”
 - We have a fault report, the source code, and — often — nothing else
- “Known environment”
 - We never can really know our environment
- “Selected inputs”
 - Sometimes we cannot provide the inputs we want
 - Simulation is needed

23

6.4 What Should Be Tested?

- “Inference”
 - We have a fault report, the source code, and — often — nothing else
- “Known environment”
 - We never can really know our environment
- “Selected inputs”
 - Sometimes we cannot provide the inputs we want
 - Simulation is needed **[AND DON'T FORGET BOUNDARY VALUES]**

24

6.4 What Should Be Tested?

- We need to test correctness (of course), and also
 - Utility
 - Reliability
 - Robustness, and
 - Performance

25

6.4.1 Utility

- The extent to which the product meets the user's needs
 - Examples:
 - Ease of use
 - Useful functions
 - Cost effectiveness

26

6.4.2 Reliability

- A measure of the frequency and criticality of failure
 - Mean time between failures
 - Mean time to repair
 - Time (and cost) to repair the *results* of a failure

27

6.4.3 Robustness

- A function of
 - The range of operating conditions
 - The possibility of unacceptable results with valid input
 - The effect of invalid input

28

6.4.4 Performance

- The extent to which space and time constraints are met
- Real-time software is characterized by *hard* real-time constraints
- If data are lost because the system is too slow
 - There is no way to recover those data

29

6.4.5 Correctness

- A product is correct if it satisfies its specifications

30

Correctness of specifications

- Incorrect specification for a sort:

<i>Input specification:</i>	$p : \text{array of } n \text{ integers, } n > 0.$
<i>Output specification:</i>	$q : \text{array of } n \text{ integers such that}$ $q[0] \leq q[1] \leq \dots \leq q[n - 1]$

Figure 6.1

- Function `trickSort` which satisfies this specification:

```
void trickSort (int p[ ], int q[ ])
{
    int i;
    for (i = 0; i < n; i++)
        q[i] = 0;
}
```

Figure 6.2

31

Correctness of specifications

- Incorrect specification for a sort:

<i>Input specification:</i>	$p : \text{array of } n \text{ integers, } n > 0.$
<i>Output specification:</i>	$q : \text{array of } n \text{ integers such that}$ $q[0] \leq q[1] \leq \dots \leq q[n - 1]$

Figure 6.1 (again)

- Corrected specification for the sort:

<i>Input specification:</i>	$p : \text{array of } n \text{ integers, } n > 0.$
<i>Output specification:</i>	$q : \text{array of } n \text{ integers such that}$ $q[0] \leq q[1] \leq \dots \leq q[n - 1]$

The elements of array q are a permutation of the elements of array p , which are unchanged.

32

Correctness

- Technically, correctness is
- A ~~Product~~ coding artifact is correct if it satisfies its output specifications, independent of its use of computing resources, when operating under permitted conditions. (page 166)
- *Not necessary*
 - Example: C++ compiler (bad example, don't use that compiler)
- *Not sufficient*
 - Example: `trickSort`

33

6.5 Testing versus Correctness Proofs

- A correctness proof is an alternative to execution-based testing

34

6.5.3 Correctness Proofs and Software Engineering

- Three myths of correctness proving (see over)

35

Three Myths of Correctness Proving

- Software engineers do not have enough mathematics for proofs
 - Most computer science majors either know or can learn the mathematics needed for proofs
- Proving is too expensive to be practical
 - Economic viability is determined from cost–benefit analysis
- Proving is too hard
 - Many nontrivial products have been successfully proven
 - Tools like theorem provers can assist us

36

Difficulties with Correctness Proving

- How do we find input–output specifications, loop invariants?
- What if the specifications are wrong?
- We can never be sure that specifications or a verification system are correct

37

Correctness Proofs and Software Engineering (contd)

- Correctness proofs are a vital software engineering tool, *where appropriate*:
 - **When human lives are at stake**
 - When indicated by cost–benefit analysis
 - When the risk of not proving is too great
- Also, informal proofs can improve software quality
 - Use the `assert` statement
- Model checking is a new technology that may eventually take the place of correctness proving (Section 18.11)

38

6.6 Who Should Perform Execution-Based Testing?

- Programming is *constructive*
- Testing is *destructive*
 - A successful test finds a fault
- So, programmers should not test their own code artifacts (should not be the only ones)

39

Who Should Perform Execution-Based Testing? (contd)

- Solution:
 - The programmer does informal (execution-based)testing
 - The SQA group then does systematic testing
 - The programmer debugs the module
- All test cases must be
 - Planned beforehand, **including planned input** and the expected output, and
 - Retained afterwards
 - **Expanded as bugs found**

40

6.7 When Testing Stops

- Only when the product has been irrevocably discarded

41

Extra Slide – Reminder (IMPORTANT)

- Black box testing is?
- Can we use "black box" testing techniques for unit tests?
- Can we use "black box" testing techniques for functional tests?
- Can we use "black box" testing techniques for regression tests?
- Can we use "black box" testing techniques for any kind of testing ever ever ever ever ever ever invented??

42

Extra Slide – Reminder (IMPORTANT)

- Black box testing is? **Testing to design (or specification)**
- Can we use "black box" testing techniques for unit tests? YES
- Can we use "black box" testing techniques for functional tests? YES
- Can we use "black box" testing techniques for regression tests? YES
- Can we use "black box" testing techniques for any kind of testing ever ever ever ever ever ever invented?? **YES**

43

Extra Slide (IMPORTANT)

- White (glass) box testing is?
- Can we use "white box" testing techniques for unit tests?
- Can we use "white box" testing techniques for functional tests?
- Can we use "white box" testing techniques for regression tests?
- Can we use "white box" testing techniques for any kind of testing ever ever ever ever ever ever invented??

44

Extra Slide (IMPORTANT)

- White (glass) box testing is? **Testing to code (seeing the code)**
- Can we use "white box" testing techniques for unit tests? **Yes**
- Can we use "white box" testing techniques for functional tests? **Yes**
- Can we use "white box" testing techniques for regression tests? **Yes**
- Can we use "white box" testing techniques for any kind of testing ever ever ever ever ever ever ever ever invented?? **Yes**

45

- End of chapter 6!

46

Chapter 7

Slides from Stephen Schach and McGraw Hill

1

7.1 What Is a Module?

- A lexically contiguous sequence of program statements, bounded by boundary elements, with an aggregate identifier
 - “Lexically contiguous”
 - Adjoining in the code
 - “Boundary elements”
 - { ... }
 - begin ... end
 - “Aggregate identifier”
 - A name for the entire module

2

Design of Computer

- A highly incompetent computer architect decides to build an ALU, shifter, and 16 registers with AND, OR, and NOT gates, rather than NAND or NOR gates

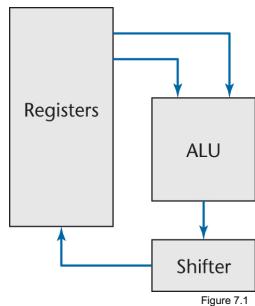


Figure 7.1

Design of Computer (contd)

- The architect designs Chip 1 three silicon chips

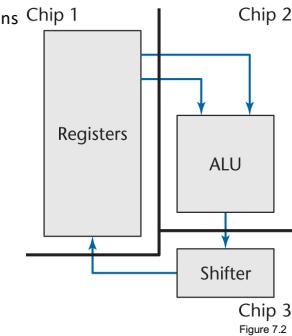


Figure 7.2

3

4

Design of Computer

- Redesign with one gate type per chip
- Resulting “masterpiece”

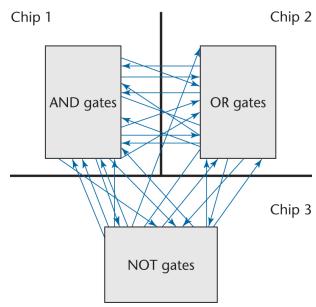


Figure 7.3

5

Computer Design

- The two designs are functionally equivalent
 - The second design is
 - Hard to understand
 - Hard to locate faults
 - Difficult to extend or enhance
 - Cannot be reused in another product
- Modules must be like the first design
 - Maximal relationships within modules, and
 - Minimal relationships between modules

6

Composite/Structured Design (C/SD p186)

- A method for breaking up a product into modules to achieve
 - Maximal interaction within a module, and
 - Minimal interaction between modules
- Module cohesion
 - Degree of interaction within a module
- Module coupling
 - Degree of interaction between modules

7

Function, Logic, and Context of a Module

- In C/SD, the name of a module is its function
- Example:
 - A module computes the square root of double precision integers using Newton's algorithm. The module is named `compute_square_root`
- The underscores denote that the classical paradigm is used here

8

7.2 Cohesion

- The degree of interaction within a module
- Seven categories or levels of cohesion (non-linear scale)

7. Informational cohesion	(Good)
6. Functional cohesion	
5. Communicational cohesion	
4. Procedural cohesion	
3. Temporal cohesion	
2. Logical cohesion	
1. Coincidental cohesion	(Bad)

Figure 7.4

9

7.2.1 Coincidental Cohesion

- A module has coincidental cohesion if it performs multiple, completely unrelated actions
- Example:
 - `print next_line,
reverse_string_of_characters_comprising_second_parameter,
add_7_to_fifth_parameter,
convert_fourth_parameter_to_floating_point`
- Such modules arise from rules like
 - “Every module will consist of between 35 and 50 statements”

10

Why Is Coincidental Cohesion So Bad?

- It degrades maintainability
- A module with coincidental cohesion is not reusable
- The problem is easy to fix
 - Break the module into separate modules, each performing one task

11

7.2.2 Logical Cohesion

- A module has logical cohesion when it performs a series of related actions, one of which is selected by the calling module

12

Logical Cohesion (contd)

- Example 1:

```
function_code = 7;  
new_operation (op code, dummy_1, dummy_2, dummy_3);  
// dummy_1, dummy_2, and dummy_3 are dummy variables,  
// not used if function code is equal to 7
```

- Example 2:

- An object performing all input and output

- Example 3:

- One version of OS/VS2 contained a module with logical cohesion performing 13 different actions. The interface contains 21 pieces of data

13

Why Is Logical Cohesion So Bad?

- The interface is difficult to understand
- Code for more than one action may be intertwined
- Difficult to reuse

14

Why Is Logical Cohesion So Bad?

- A new tape unit is installed
 - What is the effect on the laser printer?

1. Code for all input and output
2. Code for input only
3. Code for output only
4. Code for disk and tape I/O
5. Code for disk I/O
6. Code for tape I/O
7. Code for disk input
8. Code for disk output
9. Code for tape input
10. Code for tape output
:
37. Code for keyboard input

Figure 7.5

15

7.2.3 Temporal Cohesion

- A module has temporal cohesion when it performs a series of actions related in time
- Example:
 - open_old_master_file, new_master_file, transaction_file, and print_file; initialize_sales_district_table, read_first_transaction_record, read_first_old_master_record (a.k.a. perform_initialization)

16

Why Is Temporal Cohesion So Bad?

- The actions of this module are weakly related to one another, but strongly related to actions in other modules
 - Consider `sales_district_table`
- Not reusable
- Page 189

17

7.2.4 Procedural Cohesion

- A module has procedural cohesion if it performs a series of actions related by the procedure to be followed by the product
- Example:
 - `read_part_number_and_update_repair_record_on_master_file`

18

Why Is Procedural Cohesion So Bad?

- The actions are still weakly connected, so the module is not reusable

19

7.2.5 Communicational Cohesion

- A module has communicational cohesion if it performs a series of actions related by the procedure to be followed by the product, but in addition all the actions operate on the same data
- Example 1:
`update_record_in_database_and_write_it_to_audit_trail`
- Example 2:
`calculate_new_coordinates_and_send_them_to_terminal`

20

Why Is Communicational Cohesion So Bad?

- Still lack of reusability

21

7.2.6 Functional Cohesion

- A module with functional cohesion performs exactly one action

22

7.2.6 Functional Cohesion

- Example 1:
 - get_temperature_of_furnace
- Example 2:
 - compute_orbital_of_electron
- Example 3:
 - write_to_diskette
- Example 4:
 - calculate_sales_commission

23

Why Is Functional Cohesion So Good?

- More reusable
- Corrective maintenance is easier
 - Fault isolation
 - Fewer regression faults
- Easier to extend a product

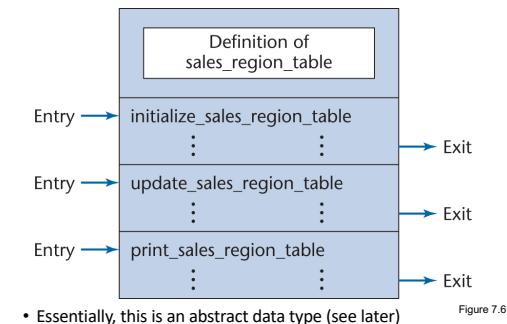
24

7.2.7 Informational Cohesion

- A module has informational cohesion if it performs a number of actions, each with its own entry point, with independent code for each action, all performed on the same data structure

25

Why Is Informational Cohesion So Good?



26

Figure 7.6

7.3 Coupling

- The degree of interaction between two modules
 - Five categories or levels of coupling (non-linear scale)

5. Data coupling	(Good)
4. Stamp coupling	
3. Control coupling	
2. Common coupling	
1. Content coupling	(Bad)

Figure 7.8

28

7.3.1 Content Coupling

- Two modules are content coupled if one directly **references contents of the other**
- Example 1:
 - Module p modifies a statement of module q
- Example 2:
 - Module p refers to local data of module q in terms of some numerical displacement within q
- Example 3:
 - Module p branches into a local label of module q

29

Why Is Content Coupling So Bad?

- Almost any change to module q , even recompiling q with a new compiler or assembler, requires a change to module p

30

7.3.2 Common Coupling

- Two modules are common coupled if they have write access to global data

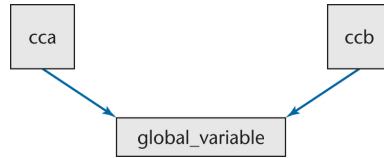


Figure 7.9

- Example 1

- Modules `cca` and `ccb` can access *and change* the value of `global_variable`

31

7.3.2 Common Coupling

- Example 2:
 - Modules `cca` and `ccb` both have access to the same database, and can both read *and* write the same record
- Example 3:
 - FORTRAN `common`
 - COBOL `common` (nonstandard)
 - COBOL-80 `global`

32

Why Is Common Coupling So Bad?

- It contradicts the spirit of structured programming
 - The resulting code is virtually unreadable
 - **What causes this loop to terminate?**

```
while (global_variable == 0)
{
    if (argument_xyz > 25)
        module_3 ();
    else
        module_4 ();
}
```

Figure 7.10

33

Why Is Common Coupling So Bad?

- Modules can have side-effects
 - This affects their readability
 - Example: `edit_this_transaction(record_7)`
 - The entire module must be read to find out what it does
- A change during maintenance to the declaration of a global variable in one module necessitates corresponding changes in other modules
- Common-coupled modules are difficult to reuse

34

Why Is Common Coupling So Bad?

- Common coupling between a module p and the rest of the product can change without changing p in any way
 - *Clandestine common coupling*
 - Example: The Linux kernel (p194)
- A module is exposed to more data than necessary
 - This can lead to computer crime

35

7.3.3 Control Coupling

- Two modules are control coupled if one passes an element of control to the other
- Example 1:
 - An operation code is passed to a module with logical cohesion
- Example 2:
 - A control switch passed as an argument

36

Control Coupling (contd)

- Module p calls module q
- Message:
 - I have failed — data
- Message:
 - I have failed, so write error message ABC123 — control

37

Why Is Control Coupling So Bad?

- The modules are not independent
 - Module `q` (the called module) must know the internal structure and logic of module `p`
 - This affects reusability
- Associated with modules of logical cohesion

38

7.3.4 Stamp Coupling

- Some languages allow only simple variables as parameters
 - `part_number`
 - `satellite_altitude`
 - `degree_of_multiprogramming`
- Many languages also support the passing of data structures
 - `part_record`
 - `satellite_coordinates`
 - `segment_table`

39

Stamp Coupling

- Two modules are stamp coupled if a data structure is passed as a parameter, but the called module operates on some but not all of the individual components of the data structure

40

Why Is Stamp Coupling So Bad?

- It is not clear, without reading the entire module, which fields of a record are accessed or changed
 - Example
calculate_withholding (employee_record)
- Difficult to understand
- Unlikely to be reusable
- More data than necessary is passed
 - Uncontrolled data access can lead to computer crime

41

Why Is Stamp Coupling So Bad? (contd)

- However, there is nothing wrong with passing a data structure as a parameter, provided that *all* the components of the data structure are **(required)** accessed and/or changed
- Examples:

```
invert_matrix (original_matrix, inverted_matrix);
print_inventory_record (warehouse_record);
```

42

7.3.5 Data Coupling

- Two modules are data coupled if all parameters are homogeneous data items (simple parameters, or data structures all of whose elements are used by called module)
- Examples:
 - `display_time_of_arrival (flight_number);`
 - `compute_product (first_number, second_number);`
 - `get_job_with_highest_priority (job_queue);`

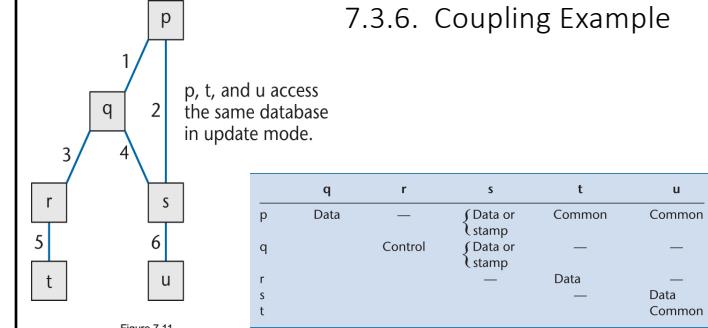
43

Why Is Data Coupling So Good?

- The difficulties of content, common, control, and stamp coupling are not present
- Maintenance is much much easier**

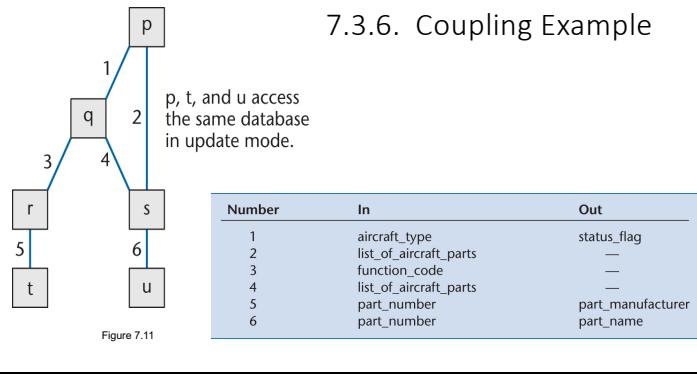
44

7.3.6. Coupling Example



45

7.3.6. Coupling Example



46

Coupling Example (contd)

Number	In	Out
1	aircraft_type	status_flag
2	list_of_aircraft_parts	—
3	function_code	—
4	list_of_aircraft_parts	—
5	part_number	part_manufacturer
6	part_number	part_name

Figure 7.12

- Interface description

47

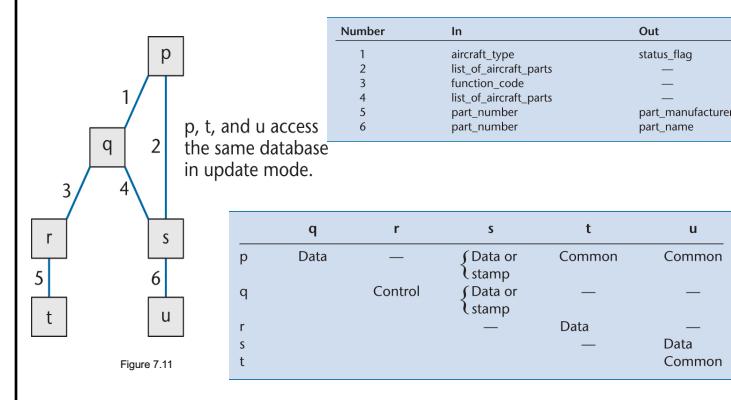
Coupling Example (contd)

	q	r	s	t	u
p	Data	—	{ Data or \ stamp	Common	Common
q	Control	{ Data or \ stamp	—	—	—
r	—	—	Data	—	—
s	—	—	—	Data	—
t	—	—	—	Common	Data

Figure 7.13

- Coupling between all pairs of modules

48



49

	q	r	s	t	u
p	Data	—	{ Data or \ stamp	Common	Common
q	Control	{ Data or \ stamp	—	—	—
r	—	—	Data	—	—
s	—	—	—	Data	—
t	—	—	—	—	Data
u	—	—	—	Common	Common

Figure 7.13

7.3.7 The Importance of Coupling

- As a result of tight coupling
 - A change to module p can require a corresponding change to module q
 - If the corresponding change is not made, this leads to faults
- Good design has high cohesion and low coupling
 - What else characterizes good design? (see over)

50

Key Definitions

- Abstract data type:** a data type together with the operations performed on instantiations of that data type (Section 7.5)
- Abstraction:** a means of achieving stepwise refinement by suppressing unnecessary details and accentuating relevant details (Section 7.4.1)
- Class:** an abstract data type that supports inheritance (Section 7.7)
- Cohesion:** the degree of interaction within a module (Section 7.1)
- Coupling:** the degree of interaction between two modules (Section 7.1)
- Data encapsulation:** a data structure together with the operations performed on that data structure (Section 7.4)
- Encapsulation:** the gathering together into one unit of all aspects of the real-world entity modeled by that unit (Section 7.4.1)
- Information hiding:** structuring the design so that the resulting implementation details are hidden from other modules (Section 7.6)
- Object:** an instantiation of a class (Section 7.7)

re 7.14

51

7.4 Data Encapsulation

- Example

- Design an operating system for a large mainframe computer. Batch jobs submitted to the computer will be classified as high priority, medium priority, or low priority. There must be three queues for incoming batch jobs, one for each job type. When a job is submitted by a user, the job is added to the appropriate queue, and when the operating system decides that a job is ready to be run, it is removed from its queue and memory is allocated to it

- Design 1 (Next slide)

- Low cohesion — operations on job queues are spread all over the product

52

Data Encapsulation — Design 1

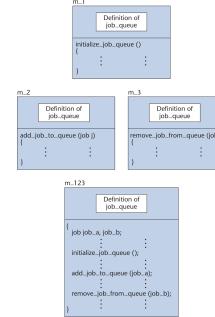
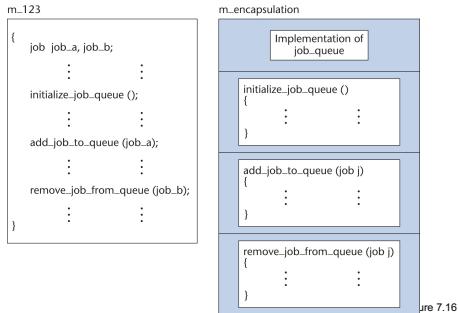


Figure 7.15

53

Data Encapsulation — Design 2



54

Data Encapsulation (contd)

- `m_encapsulation` has informational cohesion
- `m_encapsulation` is an implementation of data encapsulation
 - A data structure (`job_queue`) together with operations performed on that data structure
- Advantages
 - Development
 - Maintenance

55

Data Encapsulation and Development

- Data encapsulation is an example of *abstraction*
- Job queue example:
 - Data structure
 - job_queue
 - Three new functions
 - initialize_job_queue
 - add_job_to_queue
 - delete_job_from_queue

56

7.4.1 Data Encapsulation and Development

- Abstraction
 - Conceptualize problem at a higher level
 - Job queues and operations on job queues
 - Not at lower level
 - Records or arrays

57

Stepwise Refinement

1. Design the product in terms of higher level concepts
 - It is irrelevant how job queues are implemented
2. Then design the lower level components
 - Totally ignore what use will be made of them

58

Stepwise Refinement

- In the 1st step, **assume the existence of the lower level**
 - Our concern is the behavior of the data structure
 - job_queue
- In the 2nd step, ignore the existence of the higher level
 - Our concern is the implementation of that behavior
- In a larger product, there will be many levels of abstraction

59

7.4.2 Data Encapsulation and Maintenance

- Identify the aspects of the product that are likely to change
- Design the product so as to minimize the effects of change
 - Data structures are unlikely to change
 - Implementation details may change
- Data encapsulation provides a way to cope with change

60

7.5 Abstract Data Types

- The problem with both implementations (shown in book page 204)
 - There is only one queue, **not three**
- We need:
 - Data type + operations performed on instantiations of that data type
- Abstract data type

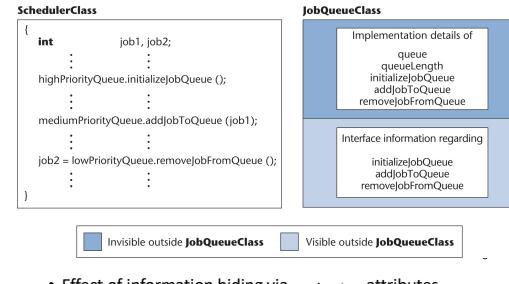
65

7.6 Information Hiding

- Data abstraction
 - The designer thinks at the level of an ADT
- Procedural abstraction
 - Define a procedure — extend the language
- Both are instances of a more general design concept, *information hiding*
 - Design the modules in a way that items likely to change are hidden
 - Future change is localized
 - Changes cannot affect other modules

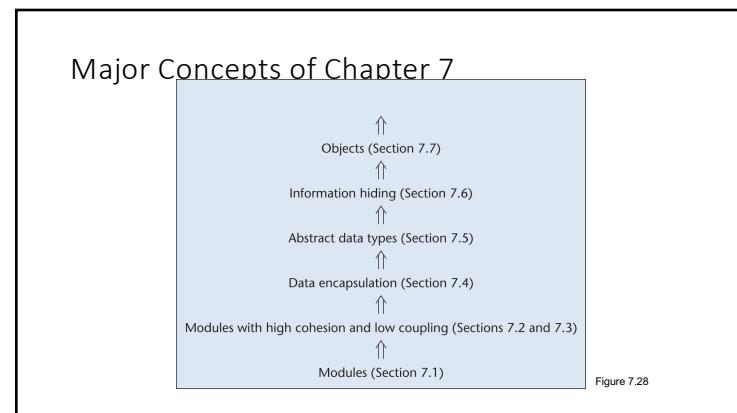
68

Information Hiding (contd)



- Effect of information hiding via `private` attributes

70



Chapter 7

Slides from Stephen Schach and McGraw Hill

7.1 What Is a Module?

- A lexically contiguous sequence of program statements, bounded by boundary elements, with an aggregate identifier
 - “Lexically contiguous”
 - Adjoining in the code
 - “Boundary elements”
 - { ... }
 - `begin ... end`
 - “Aggregate identifier”
 - A name for the entire module

Design of Computer

- A highly incompetent computer architect decides to build an ALU, shifter, and 16 registers with AND, OR, and NOT gates, rather than NAND or NOR gates

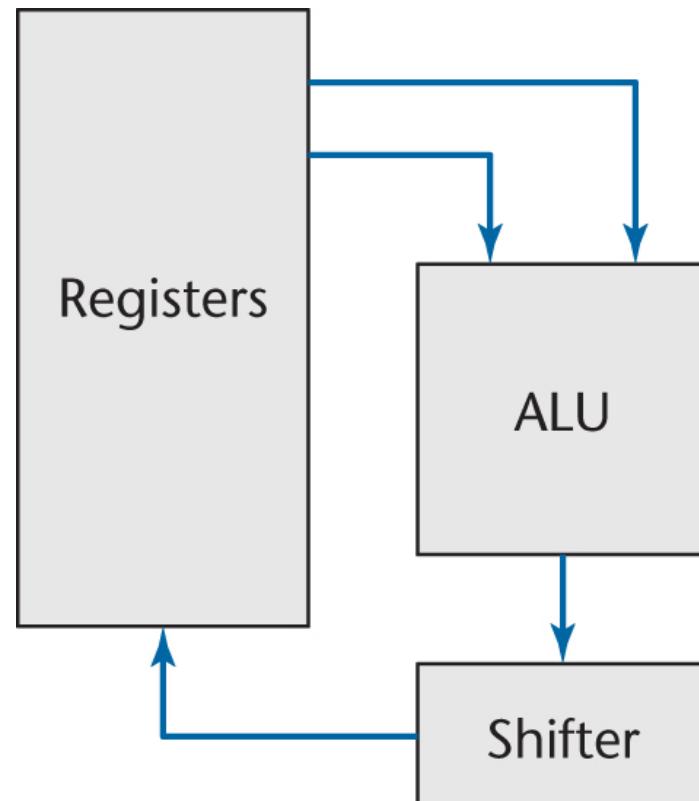


Figure 7.1

Design of Computer (contd)

- The architect designs three silicon chips

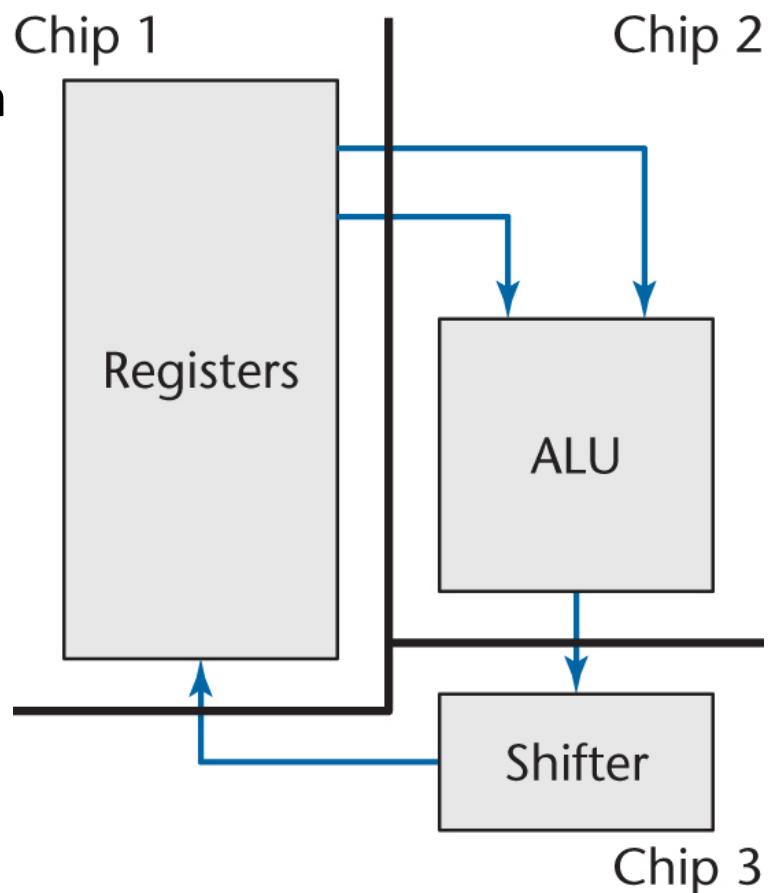


Figure 7.2

Design of Computer

- Redesign with one gate type per chip
- Resulting “masterpiece”

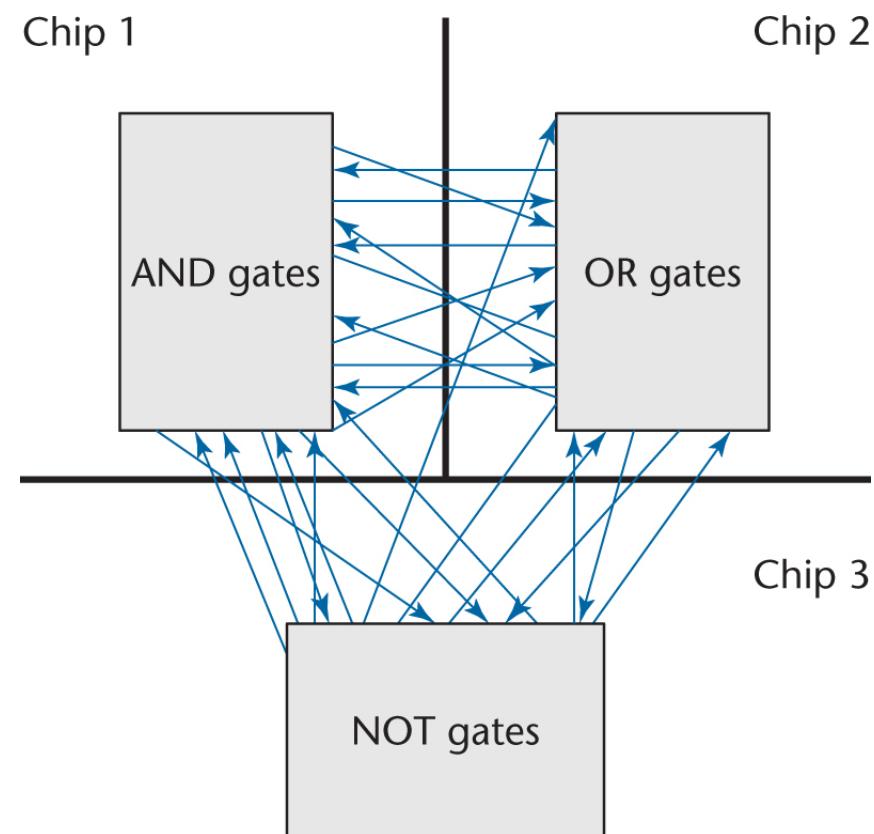


Figure 7.3

Computer Design

- The two designs are functionally equivalent
 - The second design is
 - Hard to understand
 - Hard to locate faults
 - Difficult to extend or enhance
 - Cannot be reused in another product
- Modules must be like the first design
 - Maximal relationships within modules, and
 - Minimal relationships between modules

Composite/Structured Design (C/SD p186)

- A method for breaking up a product into modules to achieve
 - Maximal interaction within a module, and
 - Minimal interaction between modules
- Module cohesion
 - Degree of interaction within a module
- Module coupling
 - Degree of interaction between modules

Function, Logic, and Context of a Module

- In C/SD, the name of a module is its function
- Example:
 - A module computes the square root of double precision integers using Newton's algorithm. The module is named `compute_square_root`
- The underscores denote that the classical paradigm is used here

7.2 Cohesion

- The degree of interaction within a module
- Seven categories or levels of cohesion (non-linear scale)

7.	Informational cohesion	(Good)
6.	Functional cohesion	
5.	Communicational cohesion	
4.	Procedural cohesion	
3.	Temporal cohesion	
2.	Logical cohesion	
1.	Coincidental cohesion	(Bad)

Figure 7.4

7.2.1 Coincidental Cohesion

- A module has coincidental cohesion if it performs multiple, completely unrelated actions
- Example:
 - `print_next_line,`
`reverse_string_of_characters_comprising_second_`
`parameter, add_7_to_fifth_parameter,`
`convert_fourth_parameter_to_floating_point`
- Such modules arise from rules like
 - “Every module will consist of between 35 and 50 statements”

Why Is Coincidental Cohesion So Bad?

- It degrades maintainability
- A module with coincidental cohesion is not reusable
- The problem is easy to fix
 - Break the module into separate modules, each performing one task

7.2.2 Logical Cohesion

- A module has logical cohesion when it performs a series of related actions, one of which is selected by the calling module

Logical Cohesion (contd)

- Example 1:

```
function_code = 7;  
  
new_operation (op code, dummy_1, dummy_2, dummy_3);  
// dummy_1, dummy_2, and dummy_3 are dummy variables,  
// not used if function code is equal to 7
```

- Example 2:

- An object performing all input and output

- Example 3:

- One version of OS/VS2 contained a module with logical cohesion performing 13 different actions. The interface contains 21 pieces of data

Why Is Logical Cohesion So Bad?

- The interface is difficult to understand
- Code for more than one action may be intertwined
- Difficult to reuse

Why Is Logical Cohesion So Bad?

- A new tape unit is installed
 - What is the effect on the laser printer?

1. Code for all input and output		
2. Code for input only		
3. Code for output only		
4. Code for disk and tape I/O		
5. Code for disk I/O		
6. Code for tape I/O		
7. Code for disk input		
8. Code for disk output		
9. Code for tape input		
10. Code for tape output		
:	:	:
37. Code for keyboard input		

Figure 7.5

7.2.3 Temporal Cohesion

- A module has temporal cohesion when it performs a series of actions related in time
- Example:
 - `open_old_master_file`, `new_master_file`, `transaction_file`, and `print_file`; `initialize_sales_district_table`, `read_first_transaction_record`, `read_first_old_master_record` (a.k.a. `perform_initialization`)

Why Is Temporal Cohesion So Bad?

- The actions of this module are weakly related to one another, but strongly related to actions in other modules
 - Consider `sales_district_table`
- Not reusable
- Page 189

7.2.4 Procedural Cohesion

- A module has procedural cohesion if it performs a series of actions related by the procedure to be followed by the product
- Example:
 - `read_part_number_and_update_repair_record_on_master_file`

Why Is Procedural Cohesion So Bad?

- The actions are still weakly connected, so the module is not reusable

7.2.5 Communicational Cohesion

- A module has communicational cohesion if it performs a series of actions related by the procedure to be followed by the product, but in addition all the actions operate on the same data

- Example 1:

```
update_record_in_database_and_write_it_to_audit_trail
```

- Example 2:

```
calculate_new_coordinates_and_send_them_to_terminal
```

Why Is Communicational Cohesion So Bad?

- Still lack of reusability

7.2.6 Functional Cohesion

- A module with functional cohesion performs exactly one action

7.2.6 Functional Cohesion

- Example 1:
 - `get_temperature_of_furnace`
- Example 2:
 - `compute_orbital_of_electron`
- Example 3:
 - `write_to_diskette`
- Example 4:
 - `calculate_sales_commission`

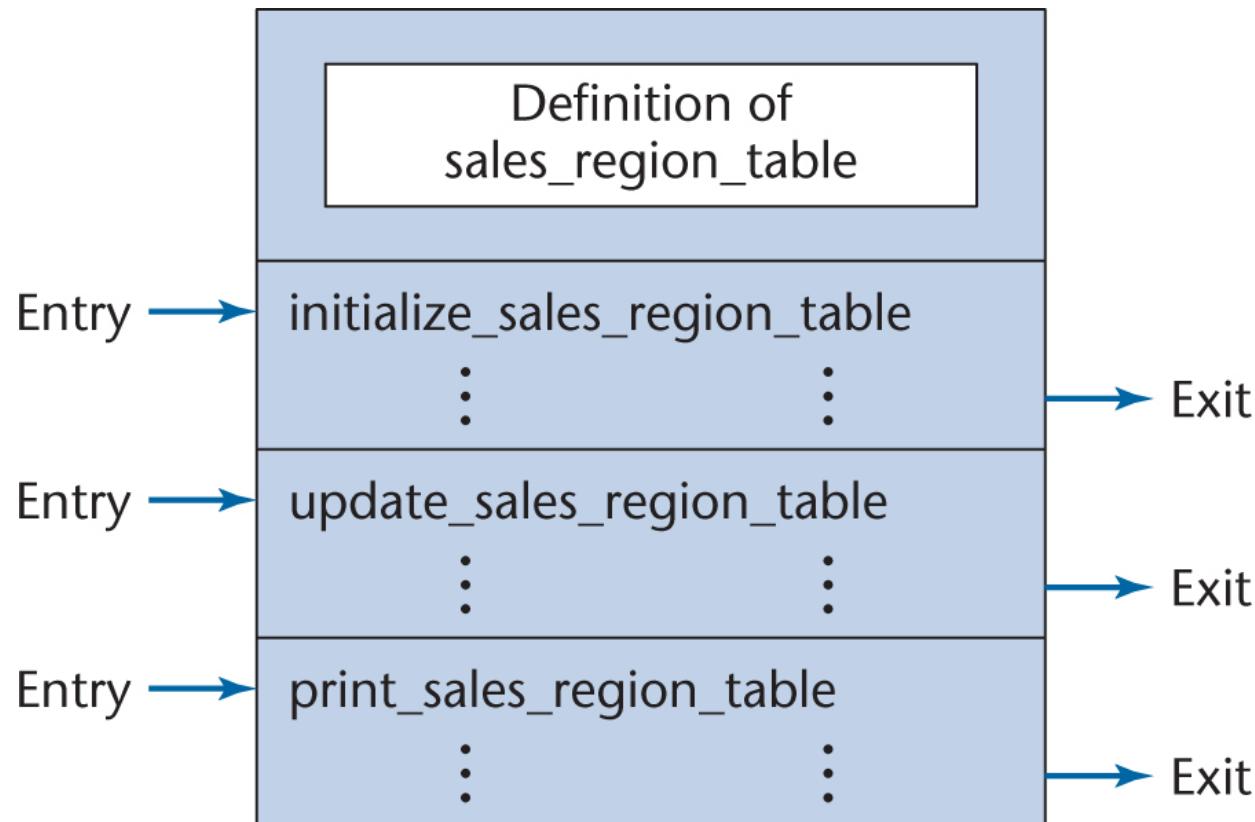
Why Is Functional Cohesion So Good?

- More reusable
- Corrective maintenance is easier
 - Fault isolation
 - Fewer regression faults
- Easier to extend a product

7.2.7 Informational Cohesion

- A module has informational cohesion if it performs a number of actions, each with its own entry point, with independent code for each action, all performed on the same data structure

Why Is Informational Cohesion So Good?



- Essentially, this is an abstract data type (see later)

Figure 7.6

7.3 Coupling

- The degree of interaction between two modules
 - Five categories or levels of coupling (non-linear scale)

5.	Data coupling	(Good)
4.	Stamp coupling	
3.	Control coupling	
2.	Common coupling	
1.	Content coupling	(Bad)

Figure 7.8

7.3.1 Content Coupling

- Two modules are content coupled if one directly **references contents of the other**
- Example 1:
 - Module p modifies a statement of module q
- Example 2:
 - Module p refers to local data of module q in terms of some numerical displacement within q
- Example 3:
 - Module p branches into a local label of module q

Why Is Content Coupling So Bad?

- Almost any change to module q , even recompiling q with a new compiler or assembler, requires a change to module p

7.3.2 Common Coupling

- Two modules are common coupled if they have write access to global data



Figure 7.9

- Example 1
 - Modules `cca` and `ccb` can access *and change* the value of `global_variable`

7.3.2 Common Coupling

- Example 2:
 - Modules `cca` and `ccb` both have access to the same database, and can both read *and write* the same record
- Example 3:
 - FORTRAN `common`
 - COBOL `common` (nonstandard)
 - COBOL-80 `global`

Why Is Common Coupling So Bad?

- It contradicts the spirit of structured programming
 - The resulting code is virtually unreadable
 - **What causes this loop to terminate?**

```
while (global_variable == 0)
{
    if (argument_xyz > 25)
        module_3 ();
    else
        module_4 ();
}
```

Figure 7.10

Why Is Common Coupling So Bad?

- Modules can have side-effects
 - This affects their readability
 - Example: `edit_this_transaction (record_7)`
 - The entire module must be read to find out what it does
- A change during maintenance to the declaration of a global variable in one module necessitates corresponding changes in other modules
- Common-coupled modules are difficult to reuse

Why Is Common Coupling So Bad?

- Common coupling between a module p and the rest of the product can change without changing p in any way
 - *Clandestine common coupling*
 - Example: The Linux kernel (p194)
- A module is exposed to more data than necessary
 - This can lead to computer crime

7.3.3 Control Coupling

- Two modules are control coupled if one passes an element of control to the other
- Example 1:
 - An operation code is passed to a module with logical cohesion
- Example 2:
 - A control switch passed as an argument

Control Coupling (contd)

- Module p calls module q
- Message:
 - I have failed — **data**
- Message:
 - I have failed, so write error message ABC123 — **control**

Why Is Control Coupling So Bad?

- The modules are not independent
 - Module q (the called module) must know the internal structure and logic of module p
 - This affects reusability
- Associated with modules of logical cohesion

7.3.4 Stamp Coupling

- Some languages allow only simple variables as parameters
 - part_number
 - satellite_altitude
 - degree_of_multiprogramming
- Many languages also support the passing of data structures
 - part_record
 - satellite_coordinates
 - segment_table

Stamp Coupling

- Two modules are stamp coupled if a data structure is passed as a parameter, but the called module operates on some but not all of the individual components of the data structure

Why Is Stamp Coupling So Bad?

- It is not clear, without reading the entire module, which fields of a record are accessed or changed
 - Example

```
calculate_withholding (employee_record)
```
- Difficult to understand
- Unlikely to be reusable
- More data than necessary is passed
 - Uncontrolled data access can lead to computer crime

Why Is Stamp Coupling So Bad? (contd)

- However, there is nothing wrong with passing a data structure as a parameter, provided that *all* the components of the data structure are **(required)** accessed and/or changed
- Examples:

```
invert_matrix (original_matrix, inverted_matrix);  
print_inventory_record (warehouse_record);
```

7.3.5 Data Coupling

- Two modules are data coupled if all parameters are homogeneous data items (simple parameters, or data structures all of whose elements are used by called module)
- Examples:
 - `display_time_of_arrival (flight_number);`
 - `compute_product (first_number, second_number);`
 - `get_job_with_highest_priority (job_queue);`

Why Is Data Coupling So Good?

- The difficulties of content, common, control, and stamp coupling are not present
- **Maintenance is much much easier**

7.3.6. Coupling Example

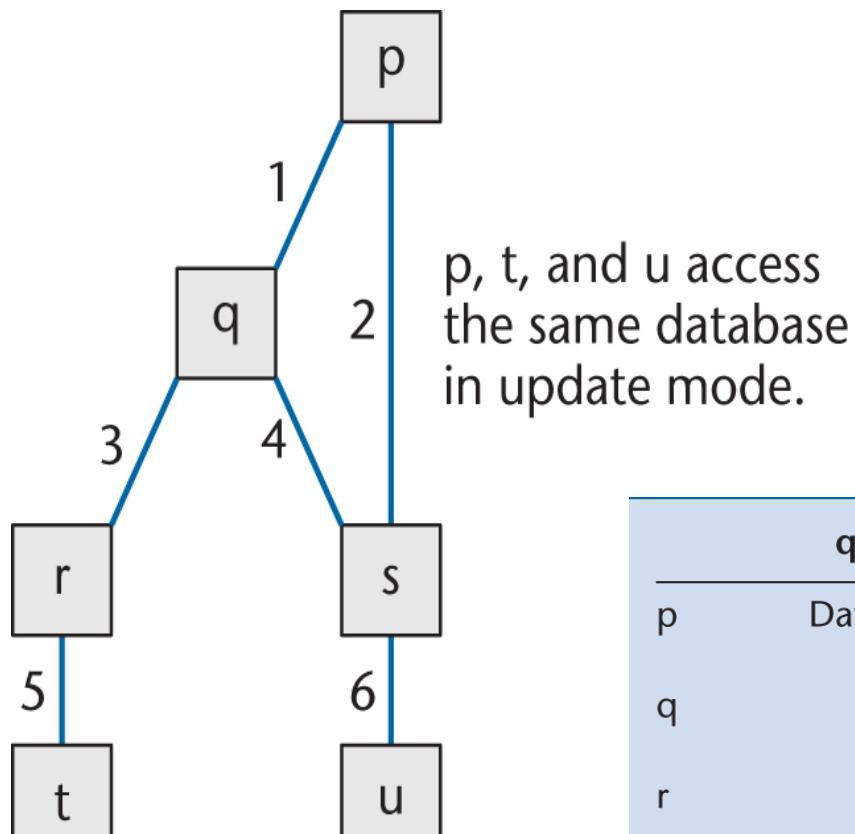


Figure 7.11

	q	r	s	t	u
p	Data	—	{Data or stamp}	Common	Common
q		Control	{Data or stamp}	—	—
r			—	Data	—
s				—	Data
t					Common

7.3.6. Coupling Example

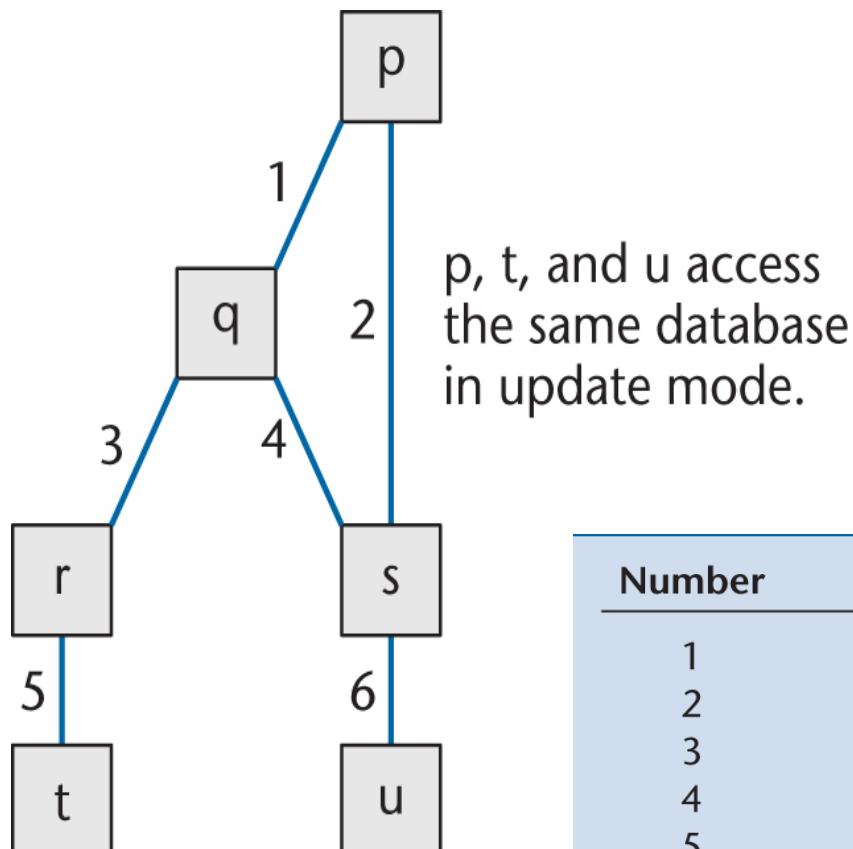


Figure 7.11

Number	In	Out
1	aircraft_type	status_flag
2	list_of_aircraft_parts	—
3	function_code	—
4	list_of_aircraft_parts	—
5	part_number	part_manufacturer
6	part_number	part_name

Coupling Example (contd)

Number	In	Out
1	aircraft_type	status_flag
2	list_of_aircraft_parts	—
3	function_code	—
4	list_of_aircraft_parts	—
5	part_number	part_manufacturer
6	part_number	part_name

Figure 7.12

- Interface description

Coupling Example (contd)

q	r	s	t	u
p	Data	—	{ Data or { stamp	Common
q	Control	{ Data or { stamp	—	—
r		—	Data	—
s			—	Data
t				Common

Figure 7.13

- Coupling between all pairs of modules

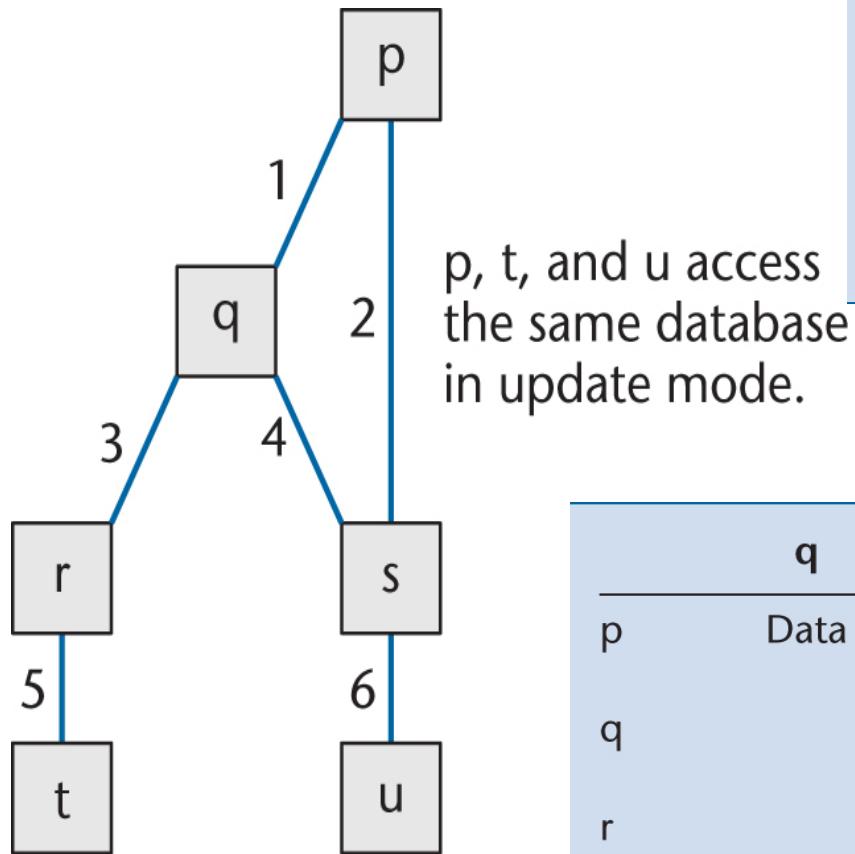


Figure 7.11

Number	In	Out
1	aircraft_type	status_flag
2	list_of_aircraft_parts	—
3	function_code	—
4	list_of_aircraft_parts	—
5	part_number	part_manufacturer
6	part_number	part_name

	q	r	s	t	u
p	Data	—	{ Data or stamp }	Common	Common
q		Control	{ Data or stamp }	—	—
r			—	Data	—
s				—	Data
t					Common

7.3.7 The Importance of Coupling

- As a result of tight coupling
 - A change to module p can require a corresponding change to module q
 - If the corresponding change is not made, this leads to faults
- Good design has high cohesion and low coupling
 - What else characterizes good design? (see over)

Key Definitions

Abstract data type: a data type together with the operations performed on instantiations of that data type (Section 7.5)

Abstraction: a means of achieving stepwise refinement by suppressing unnecessary details and accentuating relevant details (Section 7.4.1)

Class: an abstract data type that supports inheritance (Section 7.7)

Cohesion: the degree of interaction within a module (Section 7.1)

Coupling: the degree of interaction between two modules (Section 7.1)

Data encapsulation: a data structure together with the operations performed on that data structure (Section 7.4)

Encapsulation: the gathering together into one unit of all aspects of the real-world entity modeled by that unit (Section 7.4.1)

Information hiding: structuring the design so that the resulting implementation details are hidden from other modules (Section 7.6)

Object: an instantiation of a class (Section 7.7)

7.4 Data Encapsulation

- Example
 - Design an operating system for a large mainframe computer. Batch jobs submitted to the computer will be classified as high priority, medium priority, or low priority. There must be three queues for incoming batch jobs, one for each job type. When a job is submitted by a user, the job is added to the appropriate queue, and when the operating system decides that a job is ready to be run, it is removed from its queue and memory is allocated to it
- Design 1 (Next slide)
 - Low cohesion — operations on job queues are spread all over the product

Data Encapsulation – Design 1

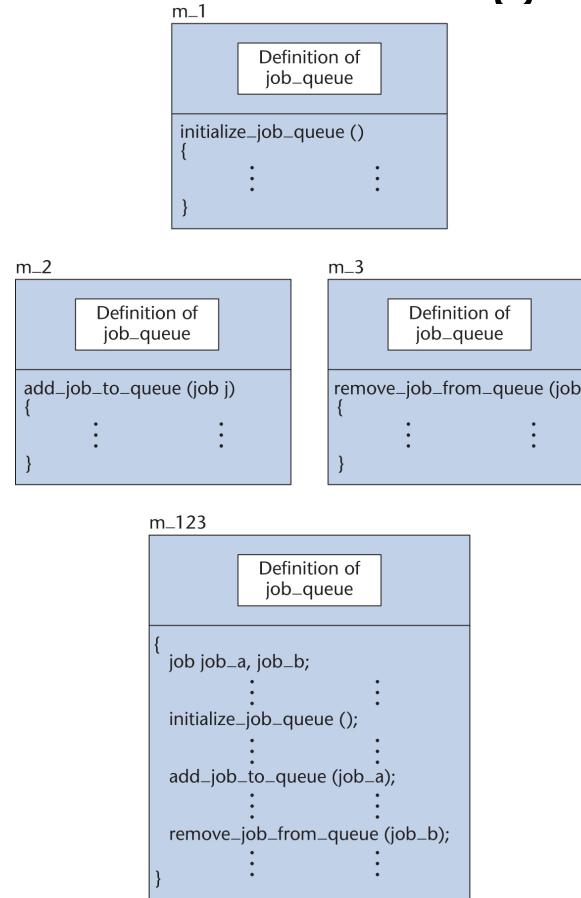


Figure 7.15

Data Encapsulation – Design 2

m_123

```
{  
    job job_a, job_b;  
    : :  
    initialize_job_queue ();  
    : :  
    add_job_to_queue (job_a);  
    : :  
    remove_job_from_queue (job_b);  
    : :  
}
```

m_encapsulation

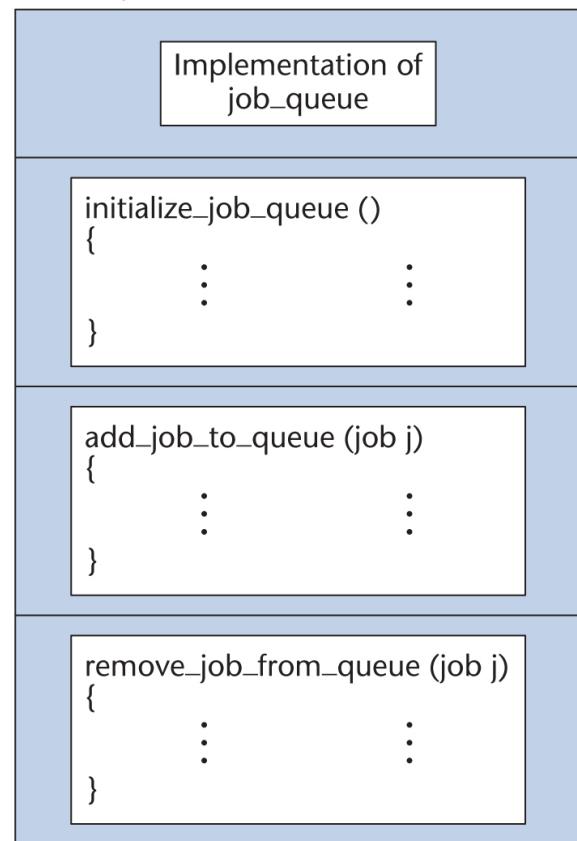


Figure 7.16

Data Encapsulation (contd)

- `m_encapsulation` has informational cohesion
- `m_encapsulation` is an implementation of data encapsulation
 - A data structure (`job_queue`) together with operations performed on that data structure
- Advantages
 - Development
 - Maintenance

Data Encapsulation and Development

- Data encapsulation is an example of *abstraction*
- Job queue example:
 - Data structure
 - `job_queue`
 - Three new functions
 - `initialize_job_queue`
 - `add_job_to_queue`
 - `delete_job_from_queue`

7.4.1 Data Encapsulation and Development

- Abstraction
 - Conceptualize problem at a higher level
 - Job queues and operations on job queues
 - Not at a lower level
 - Records or arrays

Stepwise Refinement

1. Design the product in terms of higher level concepts
 - It is irrelevant how job queues are implemented
2. Then design the lower level components
 - Totally ignore what use will be made of them

Stepwise Refinement

- In the 1st step, **assume the existence of the lower level**
 - Our concern is the behavior of the data structure
 - job_queue
- In the 2nd step, ignore the existence of the higher level
 - Our concern is the implementation of that behavior
- In a larger product, there will be many levels of abstraction

7.4.2 Data Encapsulation and Maintenance

- Identify the aspects of the product that are likely to change
- Design the product so as to minimize the effects of change
 - Data structures are unlikely to change
 - Implementation details may change
- Data encapsulation provides a way to cope with change

7.5 Abstract Data Types

- The problem with both implementations (shown in book page 204)
 - There is only one queue, **not three**
- We need:
 - Data type + operations performed on instantiations of that data type
- Abstract data type

7.6 Information Hiding

- Data abstraction
 - The designer thinks at the level of an ADT
- Procedural abstraction
 - Define a procedure — extend the language
- Both are instances of a more general design concept,
information hiding
 - Design the modules in a way that items likely to change are hidden
 - Future change is localized
 - Changes cannot affect other modules

Information Hiding (contd)

SchedulerClass

```
{  
    int          job1, job2;  
    :           :  
    highPriorityQueue.initializeJobQueue ();  
    :           :  
    mediumPriorityQueue.addJobToQueue (job1);  
    :           :  
    job2 = lowPriorityQueue.removeJobFromQueue ();  
    :           :  
}
```

JobQueueClass

Implementation details of
queue
queueLength
initializeJobQueue
addJobToQueue
removeJobFromQueue

Interface information regarding
initializeJobQueue
addJobToQueue
removeJobFromQueue



Invisible outside **JobQueueClass**



Visible outside **JobQueueClass**

7

- Effect of information hiding via `private` attributes

Major Concepts of Chapter 7

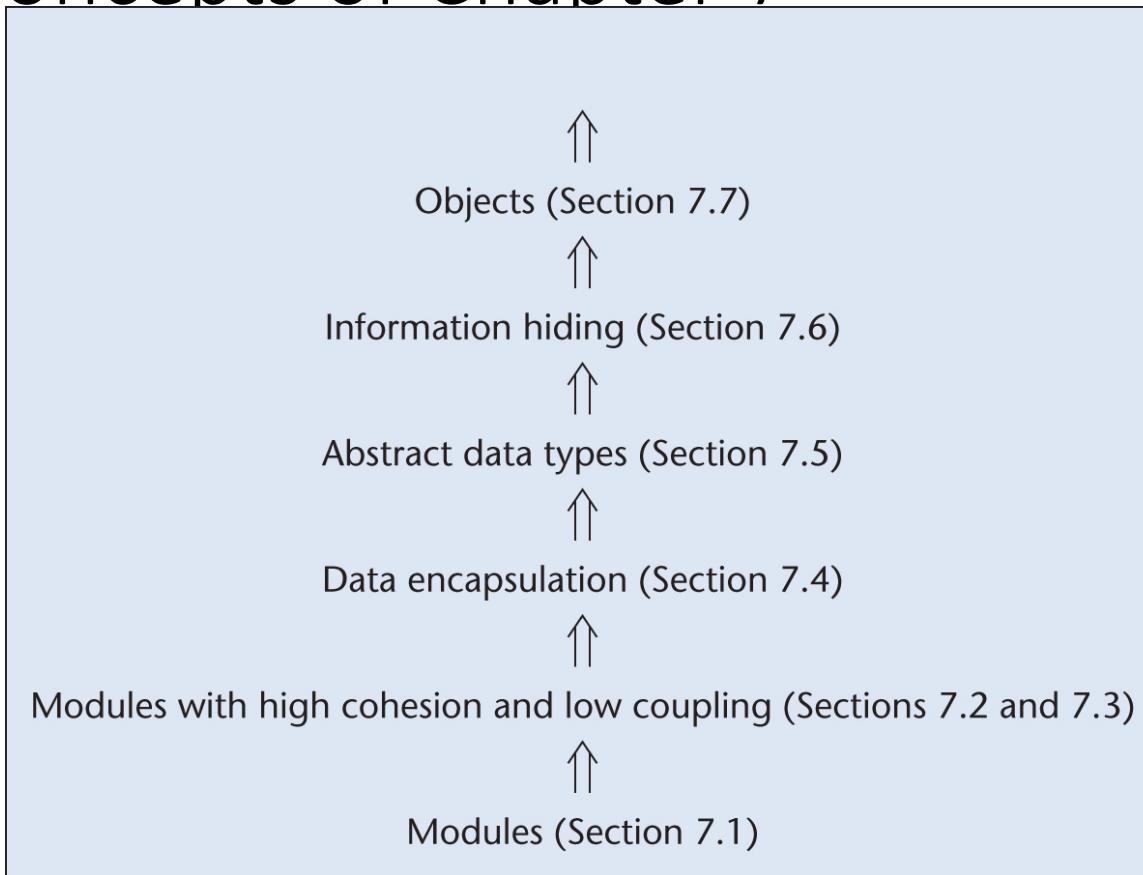


Figure 7.28

CHAPTER 8

REUSABILITY AND PORTABILITY

1

8.1 Reuse Concepts

- Reuse is the use of components of one product to facilitate the development of a different product with different functionality

2

The Two Types of Reuse

- Opportunistic (accidental) reuse
 - First, the product is built
 - Then, parts are put into the part database for reuse
- Systematic (deliberate) reuse
 - First, reusable parts are constructed
 - Then, products are built using these parts

3

Why Reuse?

- To get products to the market faster
 - There is no need to design, implement, test, and document a reused component
- On average, only 15% of new code serves an original purpose
 - In principle, 85% could be standardized and reused
 - In practice, reuse rates of no more than 40% are achieved
- Why do so few organizations employ reuse?

4

8.2 Impediments to Reuse

- Not invented here (NIH) syndrome
- Concerns about faults in potentially reusable routines
- Storage–retrieval issues

5

Impediments to Reuse

- Cost of reuse
 - The cost of making an item reusable
 - The cost of reusing the item
 - The cost of defining and implementing a reuse process
- Legal issues (contract software only)
- Lack of source code for COTS components
- The first four impediments can be overcome

6

8.3 Reuse Case Studies

- The first case study took place between 1976 and 1982
- Reuse mechanism used for COBOL design
 - Identical to what we use today for object-oriented application frameworks

7

8.3.1 Raytheon Missile Systems Division

- Data-processing software
- Systematic reuse of
 - Designs
 - 6 code templates
 - COBOL code
 - 3200 reusable modules

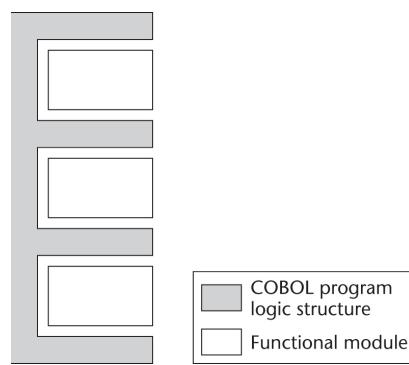


Figure 8.1

8

Raytheon Missile Systems Division (contd)

- Reuse rate of 60% was obtained
- Frameworks (“COBOL program logic structures”) were reused
- Paragraphs were filled in by functional modules
- Design and coding were quicker

9

Raytheon Missile Systems Division (contd)

- By 1983, there was a 50% increase in productivity
 - Logic structures had been reused over 5500 times
 - About 60% of code consisted of functional modules
- Raytheon hoped that maintenance costs would be reduced 60 to 80%
- Unfortunately, the division was closed before the data could be obtained

10

8.3.2 European Space Agency

- Ariane 5 rocket blew up 37 seconds after lift-off
 - Cost: \$500 million
- Reason: An attempt was made to convert a 64-bit integer into a 16-bit unsigned integer
 - The Ada `exception` handler was omitted
- The on-board computers crashed, and so did the rocket

11

The Conversion was Unnecessary

- Computations on the inertial reference system can stop 9 seconds before lift-off
- But if there is a subsequent hold in the countdown, it takes several hours to reset the inertial reference system
- Computations therefore continue 50 seconds into the flight

12

The Cause of the Problem

- Ten years before, it was mathematically proven that overflow was impossible — on the Ariane 4
- Because of performance constraints, conversions that could not lead to overflow were left unprotected
- The software was used, unchanged and untested, on the Ariane 5
 - However, the assumptions for the Ariane 4 did not hold for the Ariane 5

13

European Space Agency (contd)

- Lesson:
 - Software developed in one context needs to be retested when integrated into another context

14

8.4 Objects and Reuse

- Claim of CS/D
 - An ideal module has functional cohesion
- Problem
 - The data on which the module operates
- We cannot reuse a module unless the data are identical

15

Objects and Reuse (contd)

- Claim of CS/D:
 - The next best type of module has informational cohesion
 - This is an object (an instance of a class)
- An object comprises both data and action
- This promotes reuse

16

8.5 Reuse During Design and Implementation

- Various types of design reuse can be achieved
 - Some can be carried forward into implementation

17

8.5.1 Design Reuse

- Opportunistic reuse of designs is common when an organization develops software in only one application domain

18

Library or Toolkit

- A set of reusable routines
- Examples:
 - Scientific software
 - GUI class library or toolkit
- The user is responsible for the control logic (white in figure)

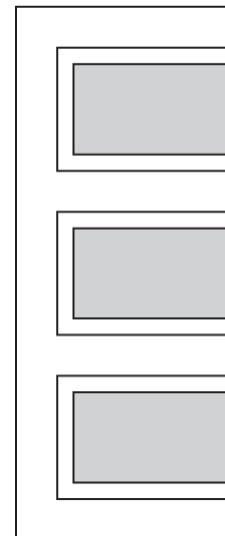


Figure 8.2(a)

19

8.5.2 Application Frameworks

- A framework incorporates the control logic of the design
- The user inserts application-specific routines in the “hot spots” (white in figure)
- Remark: Figure 8.2(b) is identical to Figure 8.1

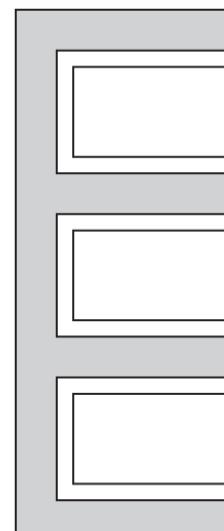


Figure 8.2(b)

20

Application Frameworks (contd)

- Faster than reusing a toolkit
 - More of the design is reused
 - The logic is usually harder to design than the operations

- Example:
 - IBM's Websphere
 - Formerly: e-Components, San Francisco
 - Utilizes Enterprise JavaBeans (classes that provide services for clients distributed throughout a network)

21

8.5.3 Design Patterns

- A pattern is a solution to a general design problem
 - In the form of a set of interacting classes

- The classes need to be customized (white in figure)

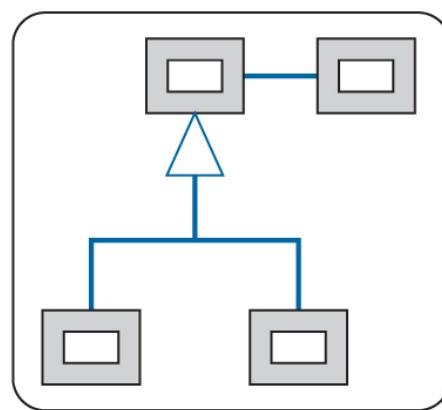


Figure 8.2(c)

22

Wrapper

- Suppose that when class P sends a message to class Q , it passes four parameters
- But Q expects only three parameters from P
- Modifying P or Q will cause widespread incompatibility problems elsewhere
- Instead, construct class A that accepts 4 parameters from P and passes three on to Q
 - Wrapper

23

Wrapper (contd)

- A wrapper is a special case of the *Adapter* design pattern
- *Adapter* solves the more general incompatibility problem
 - The pattern has to be tailored to the specific classes involved (see later)

24

Design Patterns (contd)

- If a design pattern is reused, then its implementation can also probably be reused
- Patterns can interact with other patterns

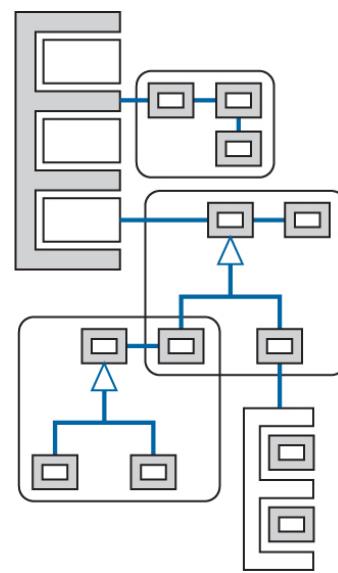


Figure 8.2(d)

25

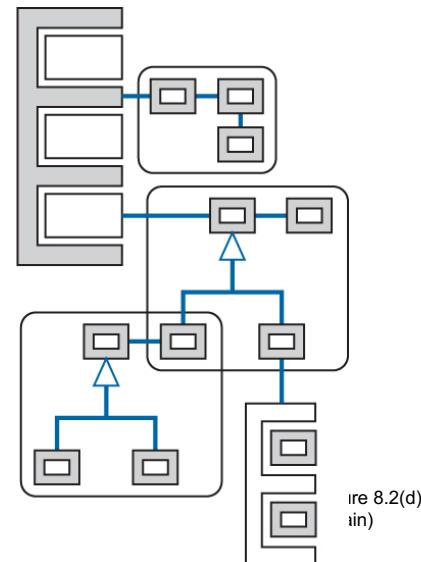
8.5.4 Software Architecture

- Encompasses a wide variety of design issues, including:
 - Organization in terms of components
 - How those components interact

26

Software Architecture

- An architecture consisting of
 - A toolkit
 - A framework, and
 - Three design patterns



27

Reuse of Software Architecture

- Architecture reuse can lead to large-scale reuse
- One mechanism:
 - Software product lines
- Case study:
 - Firmware for Hewlett-Packard printers (1995-98)
 - Person-hours to develop firmware decreased by a factor of 4
 - Time to develop firmware decreased by a factor of 3
 - Reuse increased to over 70% of components

28

Architecture Patterns

- Another way of achieving architectural reuse
- Example: The model-view-controller (MVC) architecture pattern
 - Can be viewed as an extension to GUIs of the input–processing–output architecture

MVC component	Description	Corresponds to
Model	Core functionality, data	Processing
View	Displays information	Output
Controller	Handles user input	Input

Figure 8.3

29

8.5.5 Component-Based Software Engineering

- Goal: To construct all software out of a standard collection of reusable components
- This emerging technology is outlined in Section 18.3

30

8.6 More on Design Patterns

- Case study that illustrates the *Adapter* design pattern

31

Adapter Design Pattern (contd)

- The *Adapter* design pattern
 - Solves the implementation incompatibilities; but it also
 - Provides a general solution to the problem of permitting communication between two objects with incompatible interfaces; and it also
 - Provides a way for an object to permit access to its internal implementation without coupling clients to the structure of that internal implementation
- That is, *Adapter* provides all the advantages of information hiding without having to actually hide the implementation details

32

8.6.3 *Bridge* Design Pattern

- Aim of the *Bridge* design pattern
 - To decouple an abstraction from its implementation so that the two can be changed independently of one another
- Often used in *drivers*
 - Example: a printer driver or a video driver

33

8.6.3 *Bridge* Design Pattern

- Suppose that part of a design is hardware-dependent, but the rest is not
- The design then consists of two pieces
 - The hardware-dependent parts are put on one side of the bridge
 - The hardware-independent parts are put on the other side

34

Bridge Design Pattern (contd)

- The abstract operations are uncoupled from the hardware-dependent parts
 - There is a “bridge” between the two parts
- If the hardware changes
 - The modifications to the design and the code are localized to only one side of the bridge
- The *Bridge* design pattern is a way of achieving information hiding via encapsulation

35

8.6.4 *Iterator Design Pattern*

- An aggregate object (or container or collection) is an object that contains other objects grouped together as a unit
 - Examples: linked list, hash table
- An iterator (or cursor) is a programming construct that allows a programmer to traverse the elements of an aggregate object without exposing the implementation of that aggregate

36

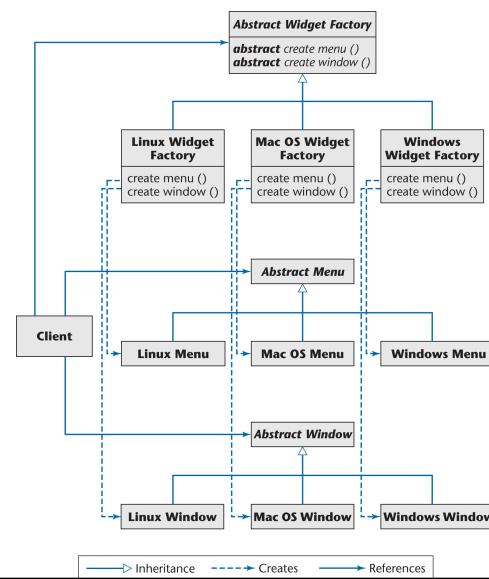
Iterator Design Pattern (contd)

- An iterator may be viewed as a pointer with two main operations:
 - Element access, or referencing a specific element in the collection; and
 - Element traversal, or modifying itself so it points to the next element in the collection

37

8.6.5 Abstract Factory Design Pattern

- We want a widget generator
 - A program that will generate widgets that can run under different operating systems



38

8.8 Strengths and Weaknesses of Design Patterns

- **Strengths**

- Design patterns promote reuse by solving a general design problem
- Design patterns provide high-level design documentation, because patterns specify design abstractions

39

Strengths and Weaknesses of Design Patterns (contd)

- **Implementations of many design patterns exist**
 - There is no need to code or document those parts of a program
 - They still need to be tested, however
- A maintenance programmer who is familiar with design patterns can easily comprehend a program that incorporates design patterns
 - Even if he or she has never seen that specific program before

40

Strengths and Weaknesses of Design Patterns (contd)

- Weaknesses

- The use of the 23 standard design patterns may be an indication that the language we are using is not powerful enough
- There is as yet no systematic way to determine when and how to apply design patterns

41

Strengths and Weaknesses of Design Patterns (contd)

- Multiple interacting patterns are employed to obtain maximal benefit from design patterns
 - But we do not yet have a systematic way of knowing when and how to use one pattern, let alone multiple interacting patterns
- It is all but impossible to retrofit patterns to an existing software product

42

Strengths and Weaknesses of Design Patterns (contd)

- The weaknesses of design patterns are outweighed by their strengths
- Research issue: How do we formalize and hence automate design patterns?
 - This would make patterns much easier to use than at present

43

8.9 Reuse and the World Wide Web

- A vast variety of code of all kinds is available on the Web for reuse
 - Also, smaller quantities of
 - Designs and
 - Patterns
- The Web supports code reuse on a previously unimagined scale
- All this material is available free of charge

44

Problems with Reusing Code from the Web

- The quality of the code varies widely
 - Code posted on the Web may or not be correct
 - Reuse of incorrect code is clearly unproductive

45

Problems with Reusing Code from the Web (contd)

- Records are kept of reuse within an organization
 - If a fault is later found in the original code, the reused code can also be fixed
- If a fault is found in a code segment that has been posted on the Web and downloaded many times
 - We cannot determine who downloaded the code, and
 - Whether or not it was actually reused after downloading

46

Problems with Reusing Code from the Web (contd)

- The World Wide Web promotes widespread reuse
- However
 - The quality of the downloaded material may be abysmal, and
 - The consequences of reuse may be severe

47

8.10 Reuse and Postdelivery Maintenance

- Reuse impacts maintenance more than development
- Assumptions
 - 30% of entire product reused unchanged
 - 10% reused changed

48

Results

Activity	Percentage of Total Cost over Product Lifetime	Percentage Savings over Product Lifetime due to Reuse	.13
Development	33%	9.3%	
Postdelivery maintenance	67	17.9	

- Savings during maintenance are nearly 18%
- Savings during development are about 9.3%

49

8.11 Portability

- Product P
 - Compiled by compiler C₁, then runs on machine M₁ under operating system O₁
- Need product P', functionally equivalent to P
 - Compiled by compiler C₂, then runs on machine M₂ under operating system O₂
- P is *portable* if it is cheaper to convert P into P' than to write P' from scratch

50

8.11.1 Hardware Incompatibilities

- Storage media incompatibilities
 - Example: Zip vs. DAT
- Character code incompatibilities
 - Example: EBCDIC vs. ASCII
- Word size

51

Hardware Incompatibilities (contd)

- IBM System/360-370 series
 - [One of] The most successful line of computers ever
 - Full upward compatibility

52

8.11.2 Operating System Incompatibilities

- Job control languages (JCL) can be vastly different
 - Syntactic differences
- Virtual memory vs. overlays

53

8.11.3 Numerical Software Incompatibilities

- Differences in word size can affect accuracy
- No problems with
 - Java
 - Ada

54

8.11.4 Compiler Incompatibilities

- FORTRAN standard is not enforced
- COBOL standard permits subsets, supersets
- ANSI C standard (1989)
 - Most compilers use the *pcc* front end
 - The *lint* processor aids portability
- ANSI C++ standard (1998)

55

Language Incompatibilities (contd)

- Ada standard — the only successful language standard
 - First enforced legally (via trademarking)
 - Then by economic forces
- Java is still evolving
 - Sun copyrighted the name to ensure standardization

56

8.12 Why Portability?

- Is there any point in porting software?
 - Incompatibilities
 - One-off software
 - Selling company-specific software may give a competitor a huge advantage

57

Why Portability? (contd)

- On the contrary, portability is **essential**
 - Good software lasts 15 years or more
 - Hardware is changed every 4 years
- Upwardly compatible hardware works
 - But it may not be cost effective
- Portability can lead to increased profits
 - Multiple copy software
 - Documentation (especially manuals) must also be portable

58

8.13 Techniques for Achieving Portability

- Obvious technique
 - Use standard constructs of a popular high-level language
- But how is a portable operating system to be written?

59

8.13.1 Portable System Software

- Isolate implementation-dependent pieces
 - Example: UNIX kernel, device-drivers
- Utilize levels of abstraction
 - Example: Graphical display routines

60

8.13.2 Portable Application Software

- Use a popular programming language
- Use a popular operating system
- Adhere strictly to language standards
- Avoid numerical incompatibilities
- **Document meticulously**

61

8.13.3 Portable Data

- File formats are often operating system-dependent
- Porting structured data
 - Construct a sequential (unstructured) file and port it
 - Reconstruct the structured file on the target machine
 - This may be nontrivial for complex database models

62

Strengths of and Impediments to Reuse and Portability

Strengths	Impediments
<p>Reuse</p> <p>Shorter development time (Section 8.1) Lower development cost (Section 8.1) Higher-quality software (Section 8.1) Shorter maintenance time (Section 8.10) Lower maintenance cost (Section 8.10)</p>	<p>NIH syndrome (Section 8.2) Potential quality issues (Section 8.2) Retrieval issues (Section 8.2) Cost of making a component reusable (opportunistic reuse) (Section 8.2) Cost of making a component for future reuse (systematic reuse) (Section 8.2) Legal issues (contract software only) (Section 8.2) Lack of source code for COTS components (Section 8.2)</p>
<p>Portability</p> <p>Software has to be ported to new hardware every 4 years or so (Section 8.12) More copies of COTS software can be sold (Section 8.12)</p>	<p>Potential incompatibilities: Hardware (Section 8.11.1) Operating systems (Section 8.11.2) Numerical software (Section 8.11.3) Compilers (Section 8.11.4) Data formats (Section 8.13.3)</p>

Figure 8.14

CHAPTER 9

PLANNING AND ESTIMATING

1

Planning and Estimating

- Before starting to build software, it is essential to plan the *entire development effort in detail*
- Planning continues during development and then postdelivery maintenance
 - Initial planning is not enough
 - Planning must proceed throughout the project
 - The earliest possible time that detailed planning can take place is after the specifications are complete

2

9.1 Planning and the Software Process

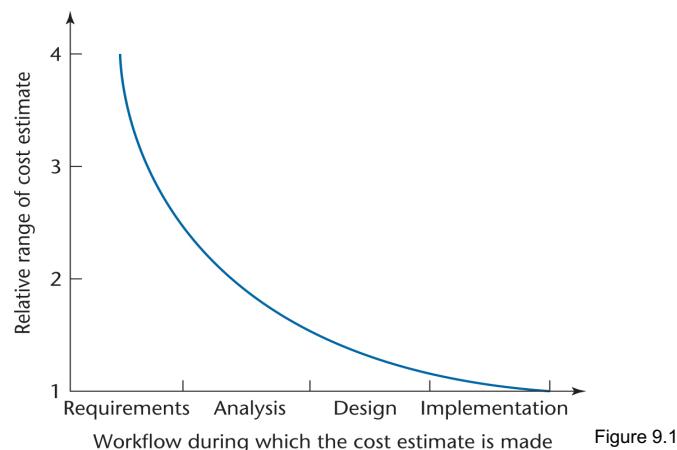


Figure 9.1

- The accuracy of estimation increases as the process proceeds

3

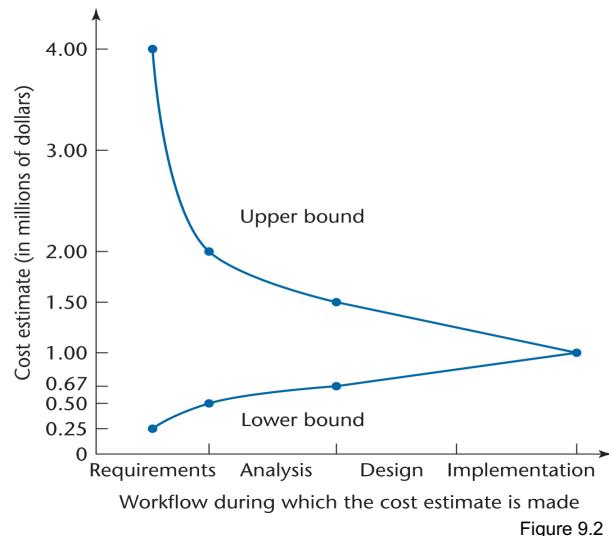
Planning and the Software Process

- Example
 - Cost estimate of \$1 million during the requirements workflow
 - Likely actual cost is in the range (\$0.25M, \$4M)
 - Cost estimate of \$1 million at the end of the requirements workflow
 - Likely actual cost is in the range (\$0.5M, \$2M)
 - Cost estimate of \$1 million at the end of the analysis workflow (earliest appropriate time)
 - Likely actual cost is in the range (\$0.67M, \$1.5M)

4

Planning and the Software Process

- These four points are shown in the *cone of uncertainty*



5

Planning and the Software Process

- This model is old (1976)
 - Estimating techniques have improved
 - But the shape of the curve is likely to be similar

6

9.2 Estimating Duration and Cost

- Accurate duration estimation is critical
- Accurate cost estimation is critical
 - Internal, external costs
- There are too many variables for accurate estimate of cost or duration

7

Human Factors

- Sackman (1968) measured differences of up to 28 to 1 between pairs of programmers
 - He compared matched pairs of programmers with respect to
 - Product size
 - Product execution time
 - Development time
 - Coding time
 - Debugging time
- Critical staff members may resign during the project

8

9.2.1 Metrics for the Size of a Product

- Lines of code (LOC, KDSI, KLOC)
- FFP (see page 273 & 274)
- Function Points
- COCOMO

9

Lines of Code (LOC)

- Alternate metric
 - Thousand delivered source instructions (KDSI)
- Source code is only a small part of the total software effort
- Different languages lead to different lengths of code
- LOC is not defined for nonprocedural languages (like LISP)

10

Lines of Code

- It is not clear how to count lines of code
 - Executable lines of code?
 - Data definitions?
 - Comments?
 - JCL statements?
 - Changed/deleted lines?
- Not everything written is delivered to the client
- A report, screen, or GUI generator can generate thousands of lines of code in minutes

11

Lines of Code

- LOC is accurately known only when the product finished
- Estimation based on LOC is therefore doubly dangerous
 - To start the estimation process, LOC in the finished product must be estimated
 - The LOC estimate is then used to estimate the cost of the product — an uncertain input to an uncertain cost estimator

12

Metrics for the Size of a Product

- Metrics based on measurable quantities that can be determined early in the software life cycle
 - FFP
 - Function points

13

Metrics for the Size of a Product

- Metrics based on measurable quantities that can be determined early in the software life cycle
 - FFP
 - Function points



14

FFP Metric

- For cost estimation of medium-scale data processing products
- The three basic structural elements of data processing products
 - Files
 - Flows
 - Processes

15

FFP Metric (page 273)

- Given the number of files (F_i), flows (F_l), and processes (P_r)
 - The size (s), cost (c) are given by

$$S = F_i + F_l + P_r$$

$$C = b \times S$$

- The constant b (efficiency or productivity) varies from organization to organization

16

FFP Metric

- The validity and reliability of the FFP metric were demonstrated using a purposive sample
 - However, the metric was never extended to include databases

17

Function Points (273 & 274)

- Based on the number of inputs (Inp), outputs (Out), inquiries (Inq), master files (Maf), interfaces (Inf)
- For any product, the size in “function points” is given by

$$UFP = 4 \times Inp + 5 \times Out + 4 \times Inq + 10 \times Maf + 7 \times Inf$$

- This is an oversimplification of a 3-step process
- [UFP>unadjusted function points]

18

Function Points

- Step 1. Classify each component of the product (Inp , Out , Inq , Maf , Inf) as simple, average, or complex
 - Assign the appropriate number of function points
 - The sum gives UFP (unadjusted function points)

Component	Level of Complexity		
	Simple	Average	Complex
Input item	3	4	6
Output item	4	5	7
Inquiry	3	4	6
Master file	7	10	15
Interface	5	7	10

Figure 9.3

19

Function Points

- Step 2. Compute the technical complexity factor (TCF)
 - Assign a value from 0 (“not present”) to 5 (“strong influence throughout”) to each of 14 factors such as transaction rates, portability

1. Data communication
2. Distributed data processing
3. Performance criteria
4. Heavily utilized hardware
5. High transaction rates
6. Online data entry
7. End-user efficiency
8. Online updating
9. Complex computations
10. Reusability
11. Ease of installation
12. Ease of operation
13. Portability
14. Maintainability

Figure 9.4

20

Function Points

- Add the 14 numbers
 - This gives the total degree of influence (DI)

$$TCF = 0.65 + 0.01 \times DI$$

- The technical complexity factor (TCF) lies between 0.65 and 1.35

21

Function Points

- Step 3. The number of function points (FP) is then given by

$$FP = UFP \times TCF$$

The real function points!

22

Analysis of Function Points

- Function points are usually better than KDSI — but there are some problems
- “Errors in excess of 800% counting KDSI, but *only* 200% in counting function points” [Jones, 1987]

23

Analysis of Function Points-KDSI

- We obtain nonsensical results from metrics
 - KDSI per person month and
 - Cost per source statement

	Assembler Version	Ada Version
Source code size	70 KDSI	25 KDSI
Development costs	\$1,043,000	\$590,000
KDSI per person-month	0.335	0.211
Cost per source statement	\$14.90	\$23.60
Function points per person-month	1.65	2.92
Cost per function point	\$3,023	\$1,170

Figure 9.5

- Cost per function point is meaningful

24

Analysis of Function Points

- Like FFP, maintenance can be inaccurately measured
- It is possible to make major changes without changing
 - The number of files, flows, and processes; or
 - The number of inputs, outputs, inquiries, master files, and interfaces
- In theory, it is possible to change every line of code without changing the number of lines of code

25

Mk II Function Points

- This metric was put forward to compute *UFP* more accurately
- We decompose software into component transactions, each consisting of input, process, and output
- Mark II function points are widely used all over the world

26

9.2.2 Techniques of Cost Estimation

- Expert judgment by analogy
- Bottom-up approach
- Algorithmic cost estimation models



27

Expert Judgment by Analogy

- Experts compare the target product to completed products
 - Guesses can lead to hopelessly incorrect cost estimates
 - Experts may recollect completed products inaccurately
 - Human experts have biases
 - However, the results of estimation by a broad group of experts may be accurate
- The Delphi technique is sometimes needed to achieve consensus

28

Bottom-up Approach

- Break the product into smaller components
 - The smaller components may be no easier to estimate
 - However, there are process-level costs

- When using the object-oriented paradigm
 - The independence of the classes assists here
 - However, the interactions among the classes complicate the estimation process

29

Algorithmic Cost Estimation Models

- A metric is used as an input to a model to compute cost and duration
 - An algorithmic model is unbiased, and therefore superior to expert opinion
 - However, estimates are only as good as the underlying assumptions

- Examples
 - SLIM Model
 - Price S Model
 - COnstructive COst MOdel (COCOMO)



30

9.2.3 Intermediate COCOMO

- COCOMO consists of three models
 - A macro-estimation model for the product as a whole
 - Intermediate COCOMO
 - A micro-estimation model that treats the product in detail
- We examine intermediate COCOMO



31

Intermediate COCOMO

- Step 1. Estimate the length of the product in KDSI
- Figure out the function points
- Pick your language
- <http://www.cs.bsu.edu/homepages/dmz/cs697/langtbl.htm>
- Multiply by **AVERAGE SOURCE STATEMENTS PER FUNCTION POINT** _____
- Convert to KDSI (Thousands of instructions)



32

Intermediate COCOMO

- Step 2. Estimate the product development mode (organic, semidetached, embedded)
- Easy, medium, and really hard

33

Intermediate COCOMO

- Step 3. Compute the nominal effort
- Nominal Effort = $a_i * (KDSI)^{b_i}$
- Example:
 - Organic product
 - 12,000 delivered source statements (12 KDSI) (estimated)

$$\text{Nominal effort} = 3.2 \times (12)^{1.05} = 43 \text{ person-months}$$



Software project	a_i	b_i
Organic	3.2	1.05
Semi-detached	3.0	1.12
Embedded	2.8	1.20

34

Intermediate COCOMO

- Step 4. Multiply the nominal value by 15 software development cost multipliers

Cost Drivers	Rating					
	Very Low	Low	Nominal	High	Very High	Extra High
Product Attributes						
Required software reliability	0.75	0.88	1.00	1.15	1.40	
Database size		0.94	1.00	1.08	1.16	
Product complexity	0.70	0.85	1.00	1.15	1.30	1.65
Computer Attributes						
Execution time constraint			1.00	1.11	1.30	1.66
Main storage constraint			1.00	1.06	1.21	1.56
Virtual machine volatility*		0.87	1.00	1.15	1.30	
Computer turnaround time		0.87	1.00	1.07	1.15	
Personnel Attributes						
Analyst capabilities	1.46	1.19	1.00	0.86	0.71	
Applications experience	1.29	1.13	1.00	0.91	0.82	
Programmer capability	1.42	1.17	1.00	0.86	0.70	
Virtual machine experience*	1.21	1.10	1.00	0.90		
Programming language experience	1.14	1.07	1.00	0.95		
Project Attributes						
Use of modern programming practices	1.24	1.10	1.00	0.91	0.82	
Use of software tools	1.24	1.10	1.00	0.91	0.83	
Required development schedule	1.23	1.08	1.00	1.04	1.10	

*For a given software product, the underlying virtual machine is the complex of hardware and software (operating system, database management system) it calls on to accomplish its task.

Figure 9.6

35

Intermediate COCOMO

- Example:
 - Microprocessor-based communications processing software for electronic funds transfer network with high reliability, performance, development schedule, and interface requirements
- Step 1. Estimate the length of the product
 - 10,000 delivered source instructions (10 KDSI)
- Step 2. Estimate the product development mode
 - Complex (“embedded”) mode

36

Intermediate COCOMO (contd)

- Step 3. Compute the nominal effort
 - Nominal effort = $2.8 \times (10)^{1.20} = 44$ person-months

- Step 4. Multiply the nominal value by 15 software development cost multipliers
 - Product of effort multipliers = 1.35 (page)
 - Estimated effort for project is therefore $1.35 \times 44 = 59$ person-months

http://sunset.usc.edu/research/COCOMOII/cocomo81_pgm/cocomo81.html

37

Intermediate COCOMO

- Software development effort multipliers

Cost Drivers	Situation	Rating	Effort Multiplier
Required software reliability	Serious financial consequences of software fault	High	1.15
Database size	20,000 bytes	Low	0.94
Product complexity	Communications processing	Very high	1.30
Execution time constraint	Will use 70% of available time	High	1.11
Main storage constraint	45K of 64K store (70%)	High	1.06
Virtual machine volatility	Based on commercial microprocessor hardware	Nominal	1.00
Computer turnaround time	2 hour average turnaround time	Nominal	1.00
Analyst capabilities	Good senior analysts	High	0.86
Applications experience	3 years	Nominal	1.00
Programmer capability	Good senior programmers	High	0.86
Virtual machine experience	6 months	Low	1.10
Programming language experience	12 months	Nominal	1.00
Use of modern programming practices	Most techniques in use over 1 year	High	0.91
Use of software tools	At basic minicomputer tool level	Low	1.10
Required development schedule	9 months	Nominal	1.00

Figure 9.7

38

Intermediate COCOMO

- Estimated effort for project (59 person-months) is used as input for additional formulas for
 - Dollar costs
 - Development schedules
 - Phase and activity distributions
 - Computer costs
 - Annual maintenance costs
 - Related items

39

Intermediate COCOMO

- Intermediate COCOMO has been validated with respect to a broad sample
- Actual values are within 20% of predicted values about 68% of the time
 - Intermediate COCOMO was the most accurate estimation method of its time
- Major problem
 - If the estimate of the number of lines of codes of the target product is incorrect, then everything is incorrect

40

9.2.4 COCOMO II

- 1995 extension to 1981 COCOMO that incorporates
 - Object orientation
 - Modern life-cycle models
 - Rapid prototyping
 - Fourth-generation languages
 - COTS software
- COCOMO II is far more complex than the first version

41

9.2.5 Tracking Duration and Cost Estimates

- Whatever estimation method used, careful tracking is vital

45

9.3 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

46

Resources

- Resources needed for software development:
 - People
 - Hardware
 - Support software

47

Use of Resources Varies with Time

- Rayleigh curves accurately depict resource consumption
- The entire software development plan must be a function of time

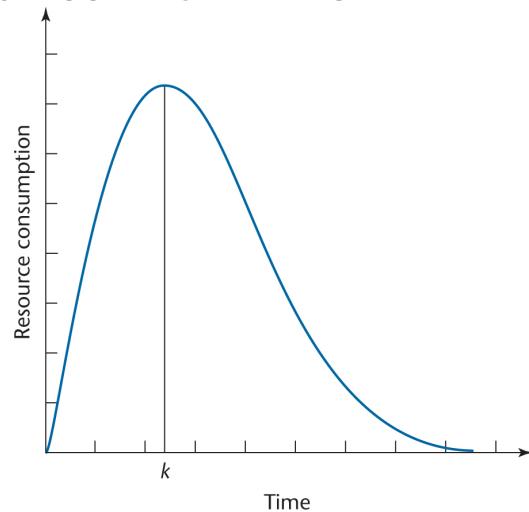


Figure 9.8

48

The End!

49