Object-Oriented and Classical Software Engineering

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

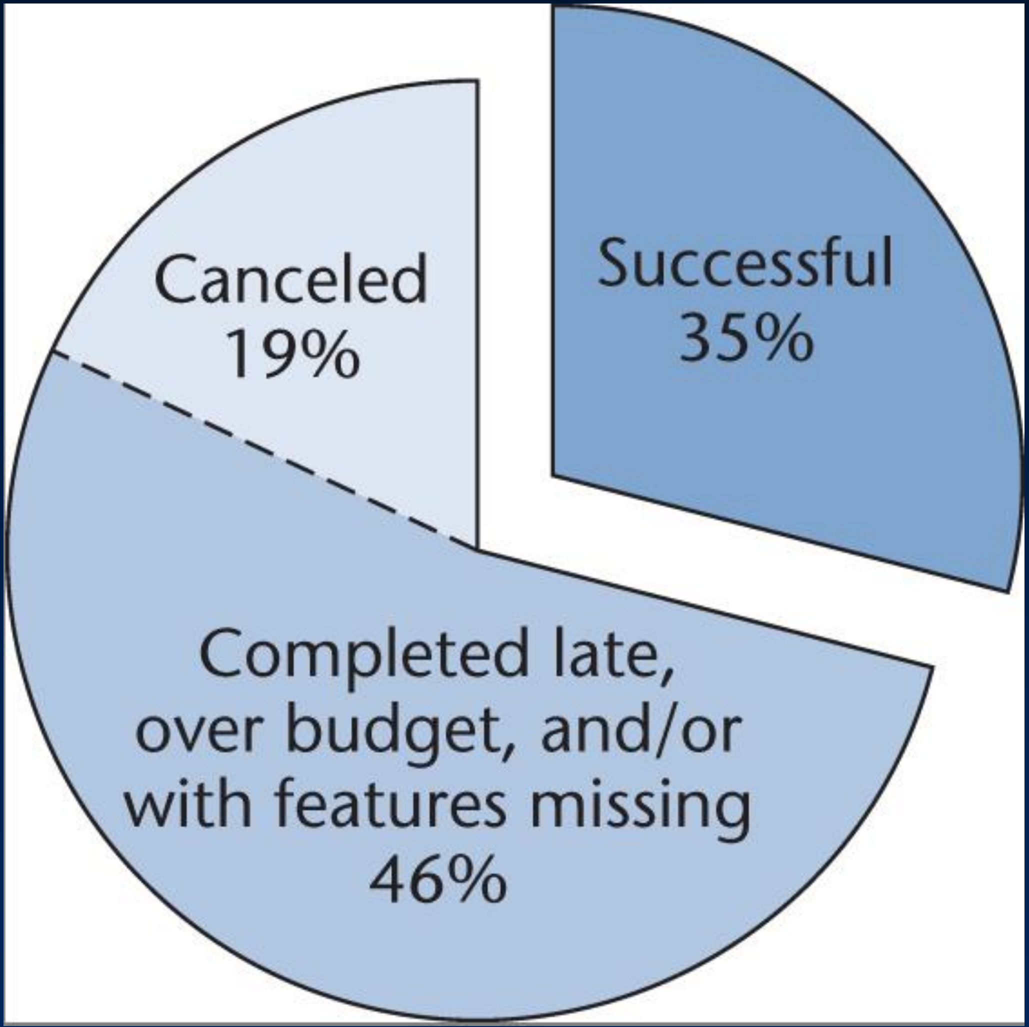

Figure 1.1

- 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

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

Time (= Cost) of Postdelivery Maintenance

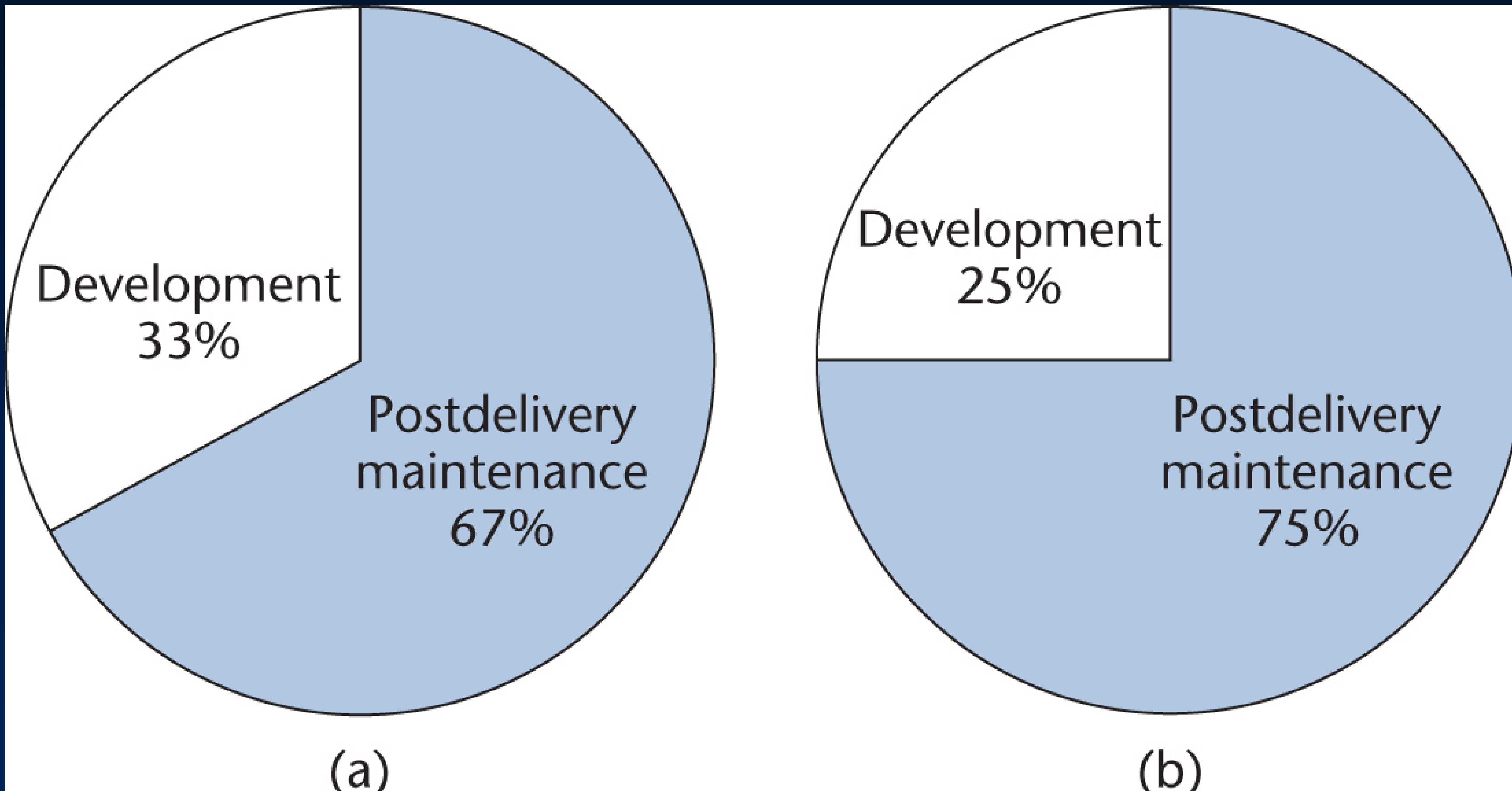

Figure 1.3

(a) Between 1976 and 1981

(b) Between 1992 and 1998

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

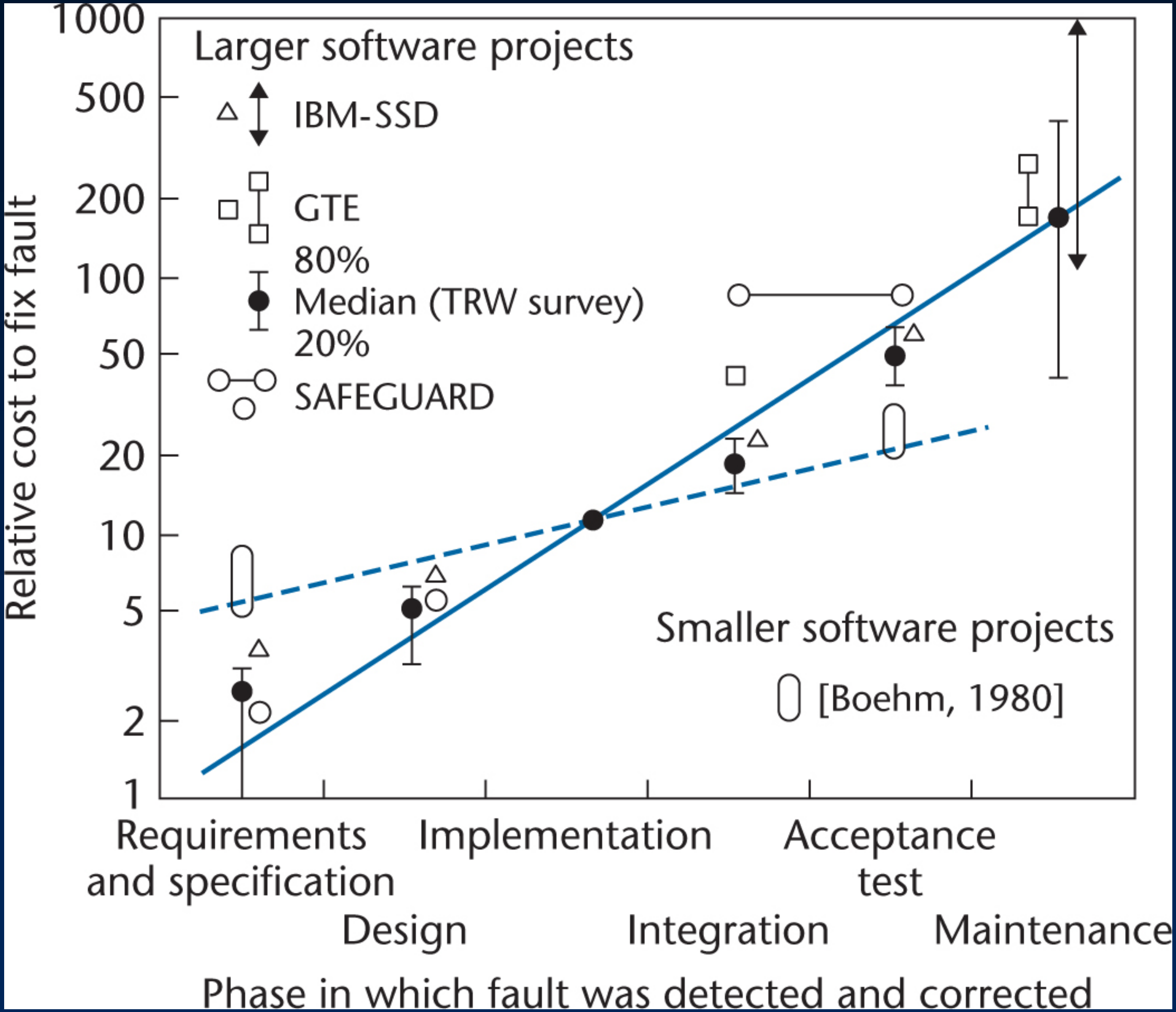

Figure 1.5

Requirements, Analysis, and Design Aspects (contd)

- The previous figure redrawn on a linear scale

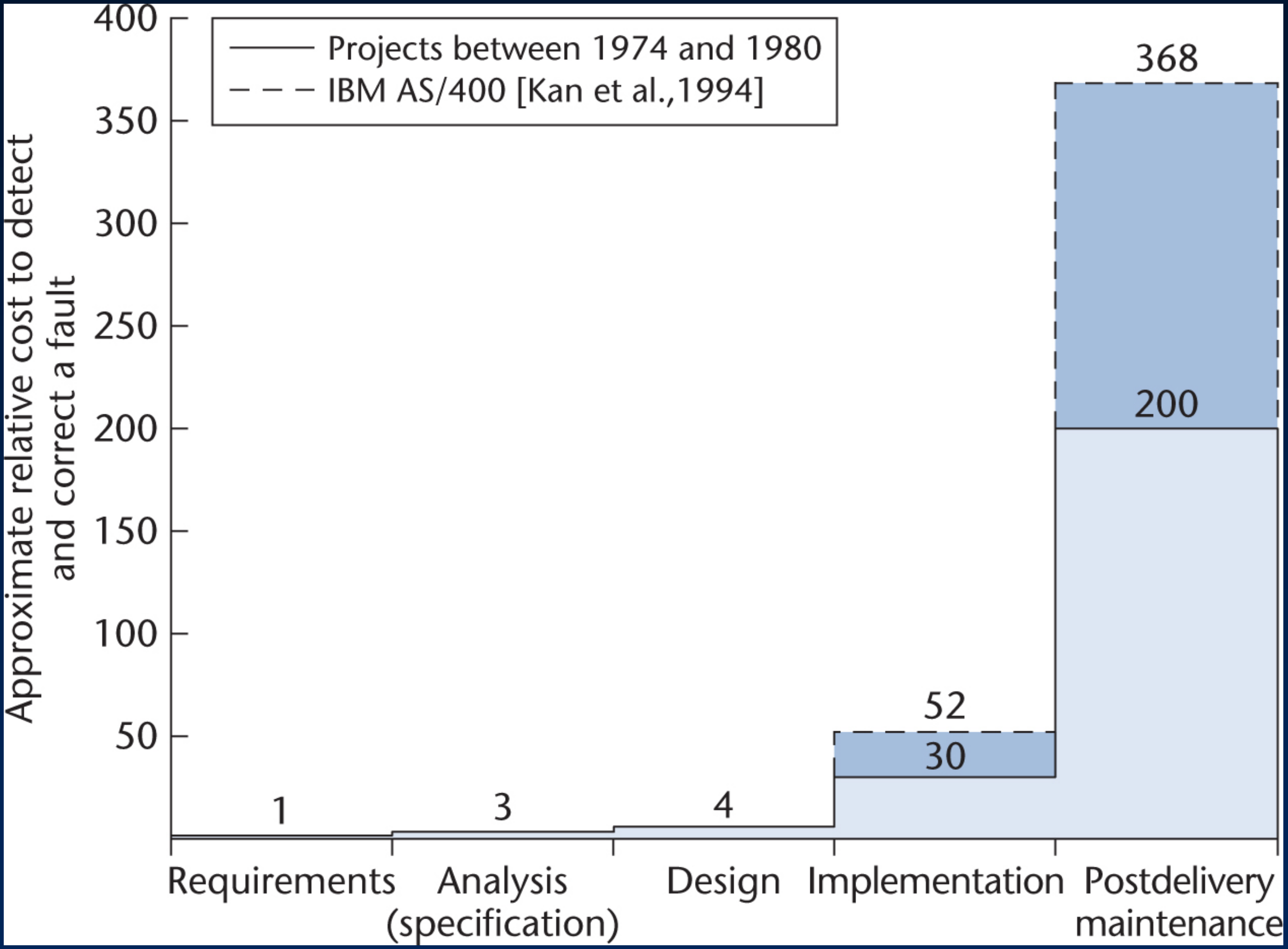

Figure 1.6

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

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

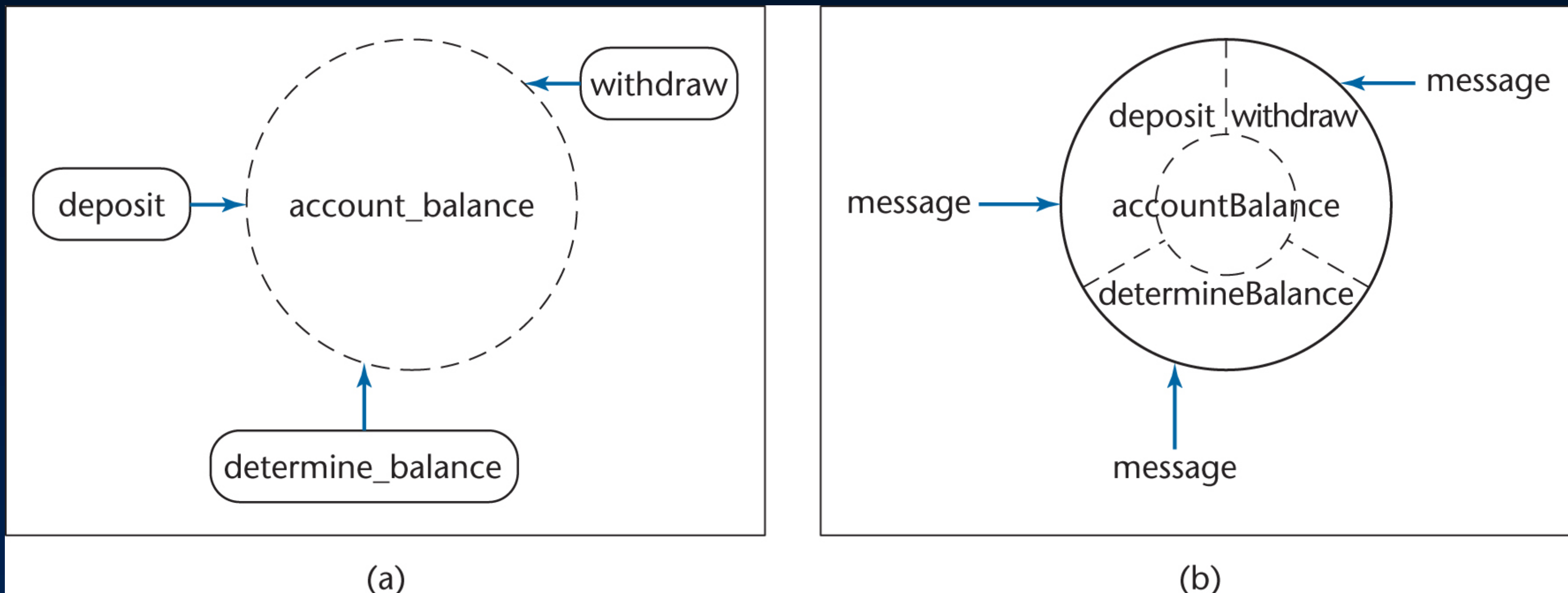

Figure 1.7

- 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 • Determine what the product is to do	2′. Object-oriented analysis workflow • Determine what the product is to do • Extract the classes
3. Design phase • Architectural design (extract the modules) • Detailed design	3′. Object-oriented design workflow • Detailed design
4. Implementation phase • Code the modules in an appropriate programming language • Integrate	4′. Object-oriented implementation workflow • Code the classes in an appropriate object-oriented programming language • Integrate

Figure 1.9

- Objects enter here

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