Object-Oriented and Classical Software Engineering

THE TOOLS OF THE TRADE

Overview

- Stepwise refinement
- Cost–benefit analysis
- Divide-and-conquer
- Separation of concerns
- Software metrics
- CASE
- Taxonomy of CASE
- Scope of CASE

Overview (contd)

- Software versions
- Configuration control
- Build tools
- Productivity gains with CASE technology

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

5.1.1 Stepwise Refinement Mini Case Study

- Design a product to update a sequential master file containing name and address data for the monthly magazine *True Life Software Disasters*

- Three types of transactions
  - Type 1: INSERT (a new subscriber into the master file)
  - Type 2: MODIFY (an existing subscriber record)
  - Type 3: DELETE (an existing subscriber record)

- Transactions are sorted into alphabetical order, and by transaction code within alphabetical order

Typical File of Input Transactions

Transaction Type	Name	Address
3	Brown	
1	Harris	2 Oak Lane, Townsville
2	Jones	Box 345, Tarrytown
3	Jones	
1	Smith	1304 Elm Avenue, Oak City

Figure 5.1

Decompose Process

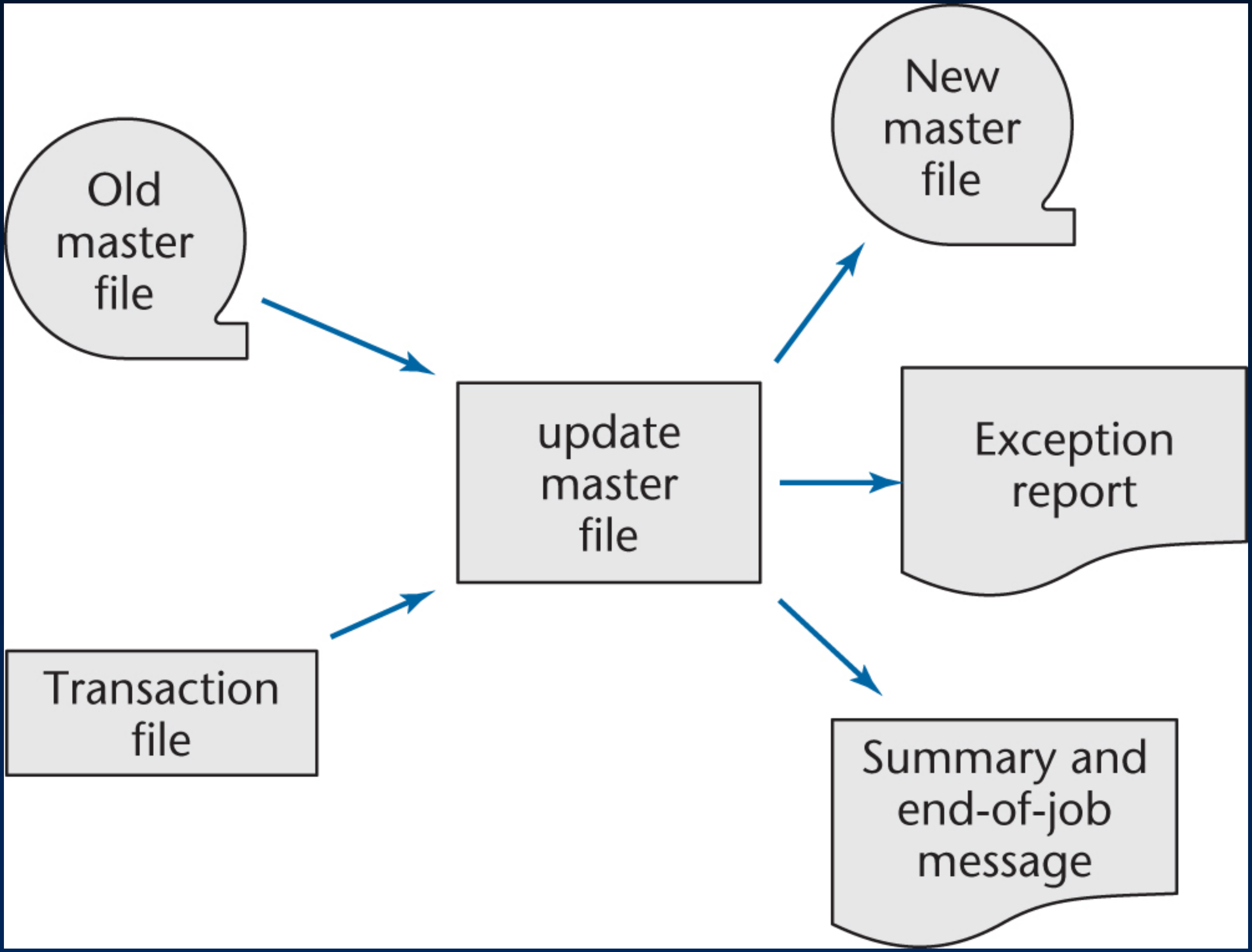

Figure 5.2

- No further refinement is possible

First Refinement

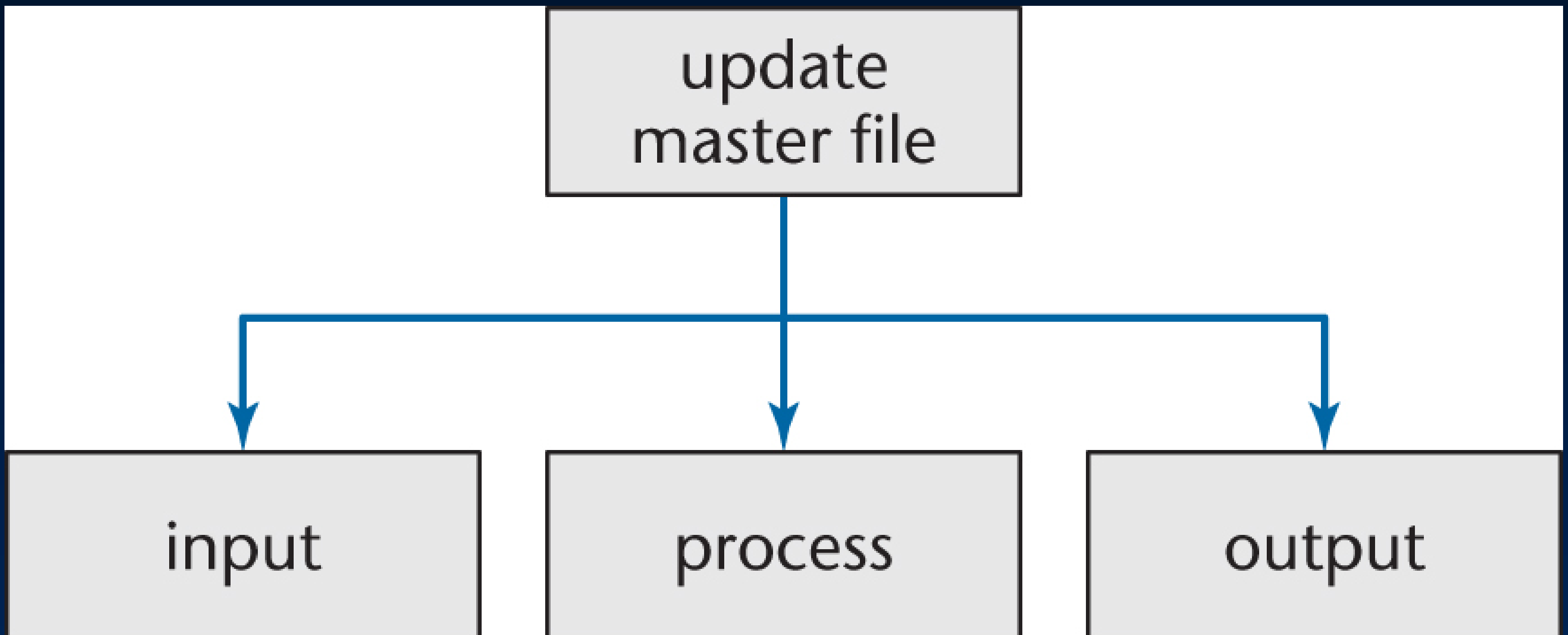

Figure 5.3

Stepwise Refinement Case Study (contd)

- Assumption
  - We can produce a record when PROCESS requires it
- Separate INPUT and OUTPUT, concentrate on PROCESS

Stepwise Refinement Case Study (contd)

- What is this PROCESS?
- Example:

Transaction file

3	Brown
1	Harris
2	Jones
3	Jones
1	Smith

Old master file

Abel
Brown
James
Jones
Smith
Townsend

New master file

Abel
Harris
James
Smith
Townsend

Exception report

Smith

Figure 5.4

Stepwise Refinement Case Study (contd)

- More formally:

Transaction record key = old master file record key	1. INSERT: Print error message 2. MODIFY: Change master file record 3. DELETE: *Delete master file record
Transaction record key > old master file record key	Copy old master file record to new master file
Transaction record key < old master file record key	1. INSERT: Write transaction record to new master file 2. MODIFY: Print error message 3. DELETE: Print error message

*Deletion of a master file record is implemented by not copying the record onto the new master file.

Figure 5.5

Second Refinement

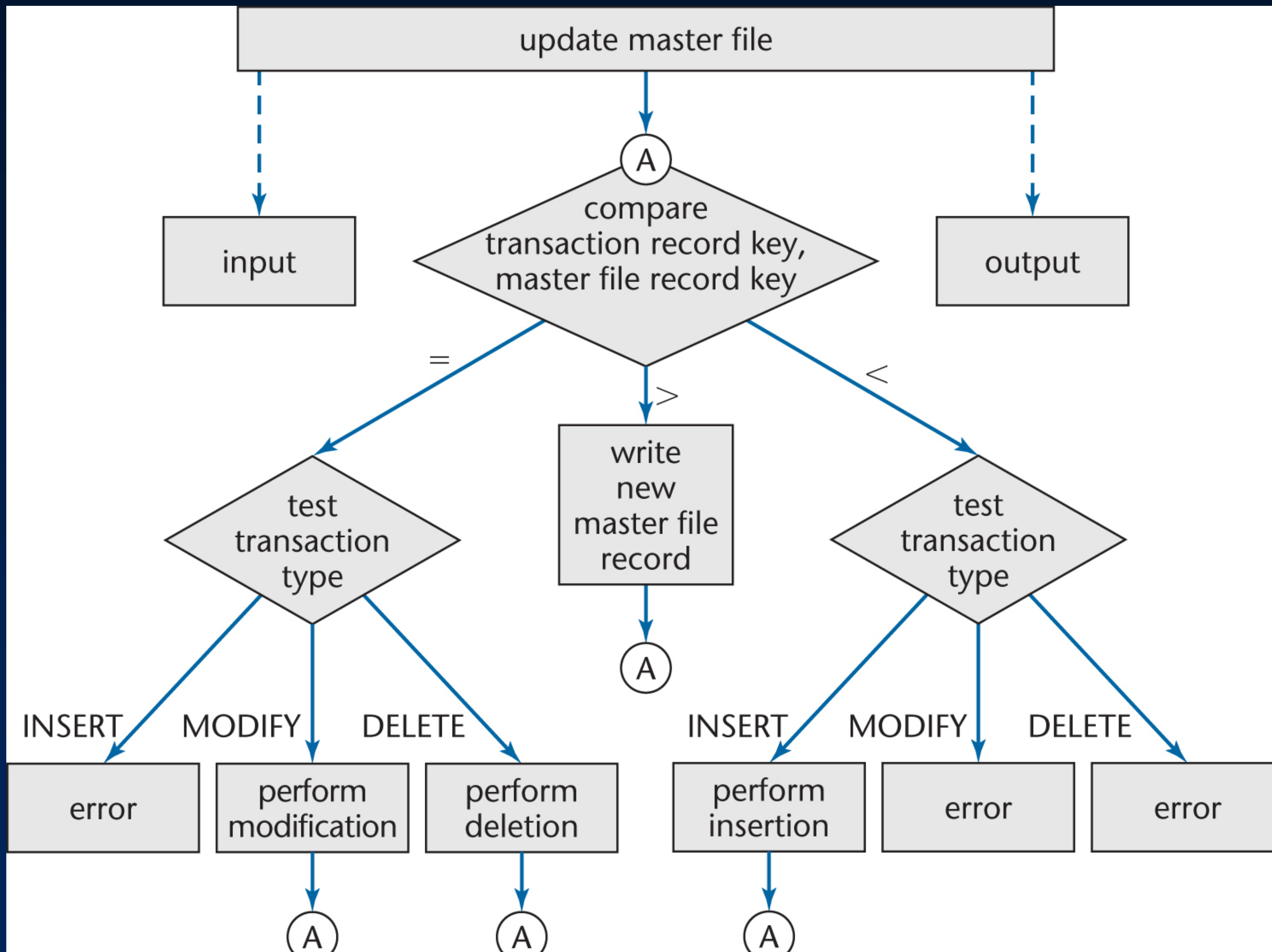

Figure 5.6

Third Refinement

- This design has a major fault

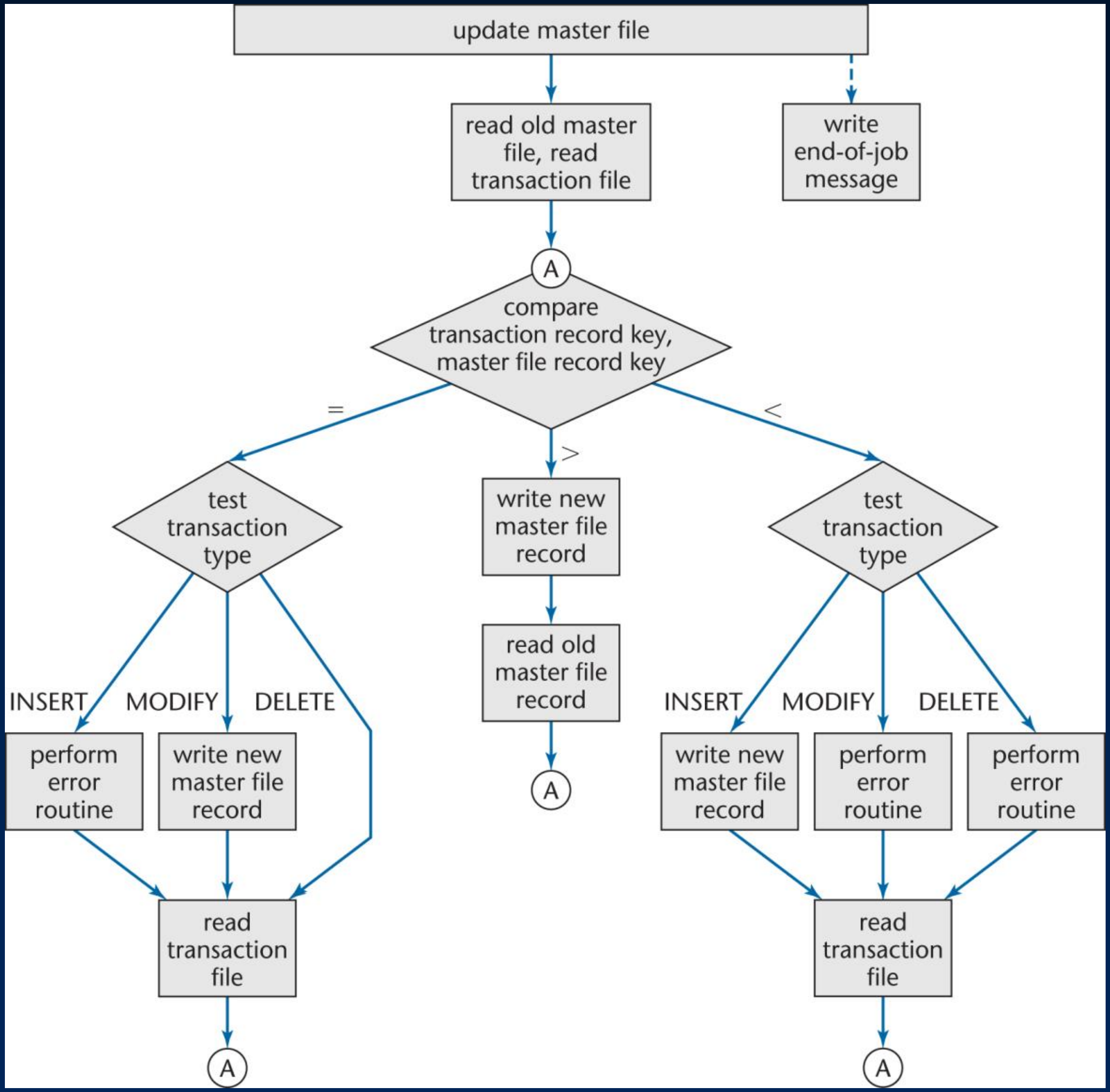

Figure 5.7

Stepwise Refinement Case Study (contd)

- The third refinement is WRONG
  - “Modify JONES” followed by “Delete JONES” is incorrectly handled

Stepwise Refinement Case Study (contd)

- After the third refinement has been corrected
  - Details like opening and closing files have been ignored up to now
  - Fix these after the logic of the design is complete
  - The stage at which an item is handled is vital

- Opening and closing files is
  - Ignored in early steps, but
  - Essential later

Appraisal of Stepwise Refinement

- A basic principle used in
  - Every workflow
  - Every representation
- The power of stepwise refinement
  - The software engineer can concentrate on the relevant aspects
- Warning
  - Miller's Law is a fundamental restriction on the mental powers of human beings

5.2 Cost–Benefit Analysis

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

Cost–Benefit Analysis (contd)

- Example: Computerizing KCEC

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 (contd)

- Tangible costs/benefits are easy to measure
- Make assumptions to estimate intangible costs/benefits
  - Improving the assumptions will improve the estimates

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*

Divide-and-Conquer (contd)

- A problem with divide-and-conquer
  - The approach does not tell us *how* to break up a software product into appropriate smaller components

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

Separation of Concerns (contd)

- Instances include:
  - Modularization with maximum interaction within each module (“high cohesion”) (Chapter 7)
  - Modularization with minimum interaction between modules (“low coupling”) (Chapter 7)
  - Information hiding (or physical independence)
  - Encapsulation (or conceptual independence)
  - Three-tier architecture (Section 8.5.4)
  - Model-view-controller (MVC) architecture pattern, (Section 8.5.4)

5.5 Software Metrics

- To detect problems early, it is essential to measure

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

Different Types of Metrics

- Product metrics
  - Examples:
    - » Size of product
    - » Reliability of product
- Process metrics
  - Example:
    - » Efficiency of fault detection during development
- Metrics specific to a given workflow
  - Example:
    - » Number of defects detected per hour in specification reviews

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

5.7 Taxonomy of CASE

- UpperCASE (front-end tool)

    versus

- LowerCASE (back-end tool)

Some Useful Tools

- Data dictionary
  - Computerized list of all data defined within the product

- Consistency checker

- Report generator, screen generator

Taxonomy of CASE (contd)

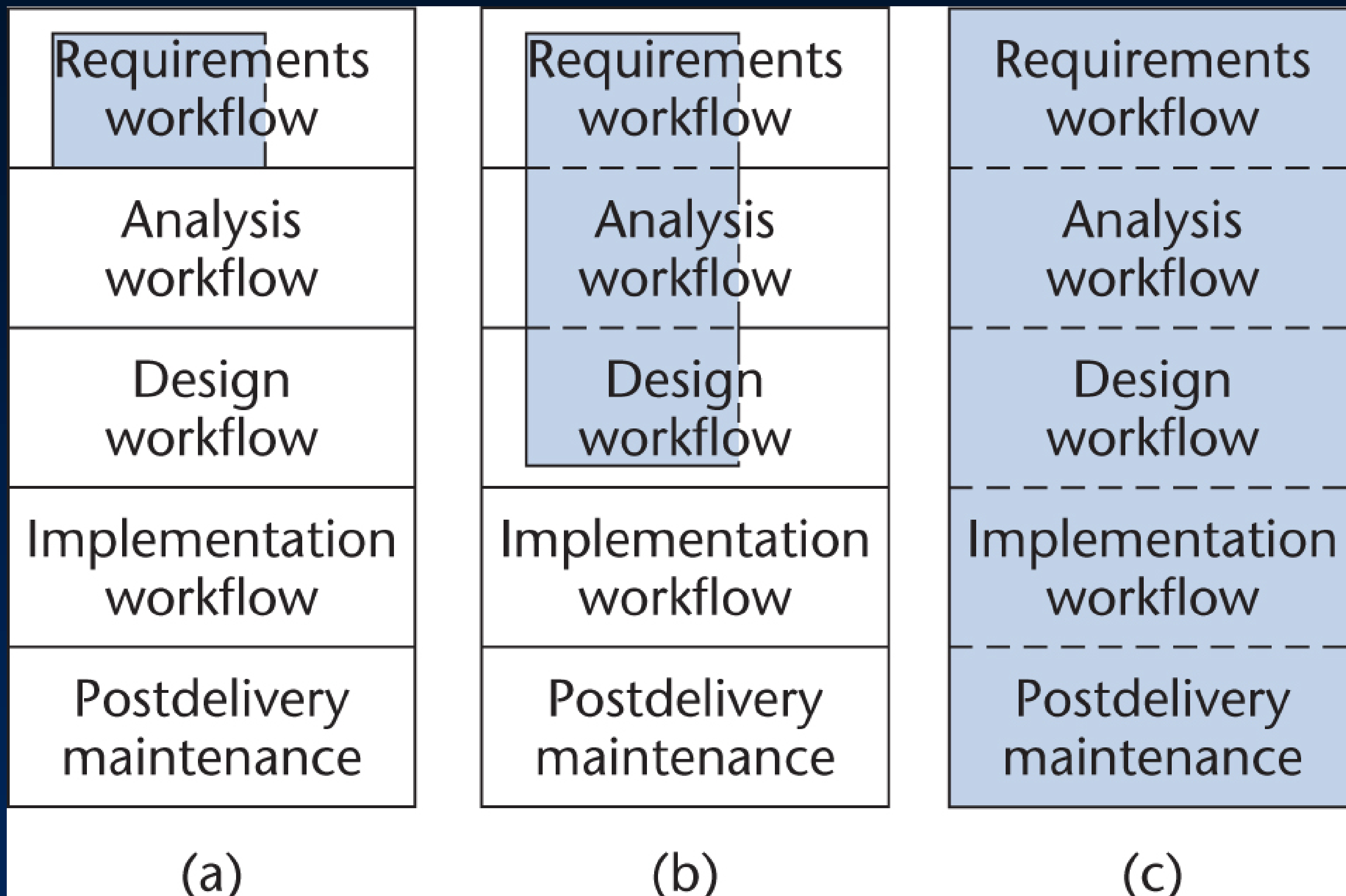

Figure 5.9

- (a) Tool versus (b) workbench versus (c) environment

5.8 Scope of CASE

- Programmers need to have:
  - Accurate, up-to-date versions of all project documents
  - Online help information regarding the
    - Operating system
    - Editor
    - Programming language
  - Online programming standards
  - Online manuals
    - Editor manuals
    - Programming manuals

Scope of CASE (contd)

- Programmers need to have:
  - E-mail systems
  - Spreadsheets
  - Word processors
  - Structure editors
  - Pretty printers
  - Online interface checkers

Online Interface Checker

- A structure editor must support online interface checking
  - The editor must know the name of every code artifact
- Interface checking is an important part of programming-in-the-large

Online Interface Checker (contd)

- Example
  - The user enters the call

```
average = dataArray.computeAverage (numberOfValues);
```

  - The editor immediately responds

```
Method computeAverage not known
```

- The programmer is given two choices
  - Correct the name of the method to `computeMean`
  - Declare new procedure `computeAverage` and specify its parameters
- This enables full interface checking

Online Interface Checker (contd)

- Example
  - Declaration of `q` is

```
void q (float floatVar, int intVar, String s1, String s2);
```

  - Call (invocation) is

```
q (intVar, floatVar, s1, s2);
```

  - The online interface checker detects the fault

- Help facility
  - Online information for the parameters of method `q`
  - Better: Editor generates a template for the call
    - » The template shows type of each parameter
    - » The programmer replaces formal by actual parameters

Online Interface Checker (contd)

- Advantages
  - There is no need for different tools with different interfaces
  - Hard-to-detect faults are immediately flagged for correction
    - » Wrong number of parameters
    - » Parameters of the wrong type
- Essential when software is produced by a team
  - If one programmer changes an interface specification, all components calling that changed artifact must be disabled

Online Interface Checker (contd)

- Even when a structure editor incorporates an online interface checker, a problem remains
  - The programmer still has to exit from the editor to invoke the compiler (to generate code)
  - Then, the linker must be called to link the product
  - The programmer must adjust to the JCL, compiler, and linker output
- Solution: Incorporate an operating system front-end into the structure editor

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
```

Source Level Debugger (contd)

- The programmer works in a high-level language, but must examine
  - Machine-code core dumps
  - Assembler listings
  - Linker listings
  - Similar low-level documentation
- This destroys the advantage of programming in a high-level language
- We need
  - An interactive source level debugger (like *dbx*)

Source Level Debugger (contd)

- Output from a typical source-level debugger

```
OVERFLOW ERROR

   Class:    cyclotronEnergy

 Method:     performComputation

 Line 6:     newValue = (oldValue + tempValue) / tempValue;
                 oldValue = 3.9583          tempValue = 0.0000
```

Figure 5.10

Programming Workbench

- Structure editor with
  - Online interface checking capabilities
  - Operating system front-end
  - Online documentation
  - Source level debugger

- This constitutes a simple programming environment

Programming Workbench (contd)

- This is by no means new
  - All the above features are supported by FLOW (1980)
  - The technology has been in place for years
- Surprisingly, some programmers still implement code the old-fashioned way

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*

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

5.9.2 Variations

- A variation is a version for a different operating system–hardware
- Variations are designed to coexist in parallel

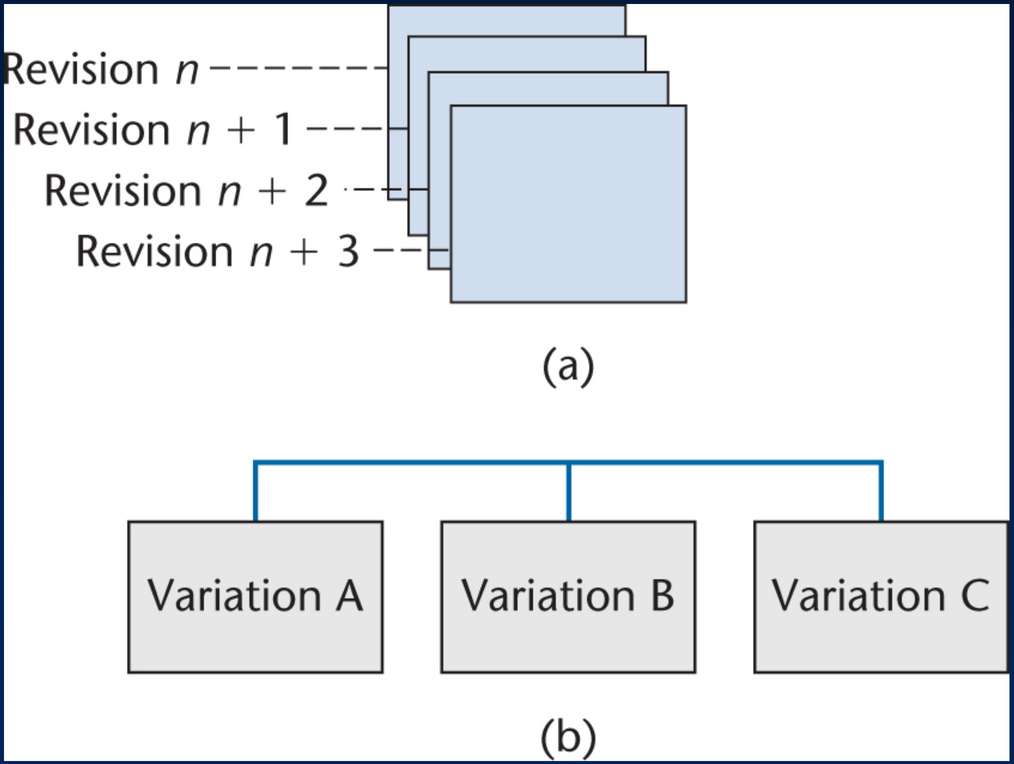

Figure 5.11

5.10 Configuration Control

- Every code artifact exists in three forms
  - Source code
  - Compiled code
  - Executable load image

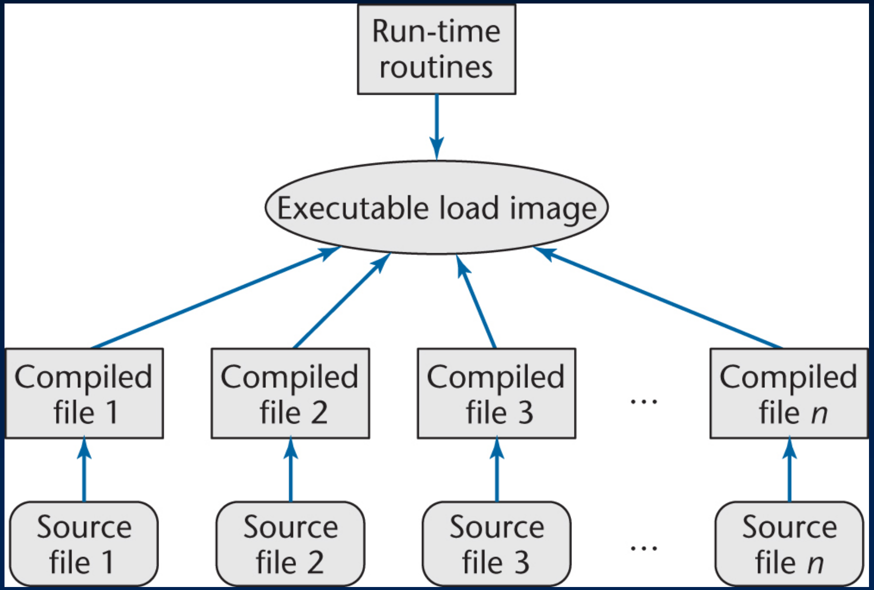

Figure 5.12

- 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

Version-Control Tool (contd)

- Notation for file name, variation, and version

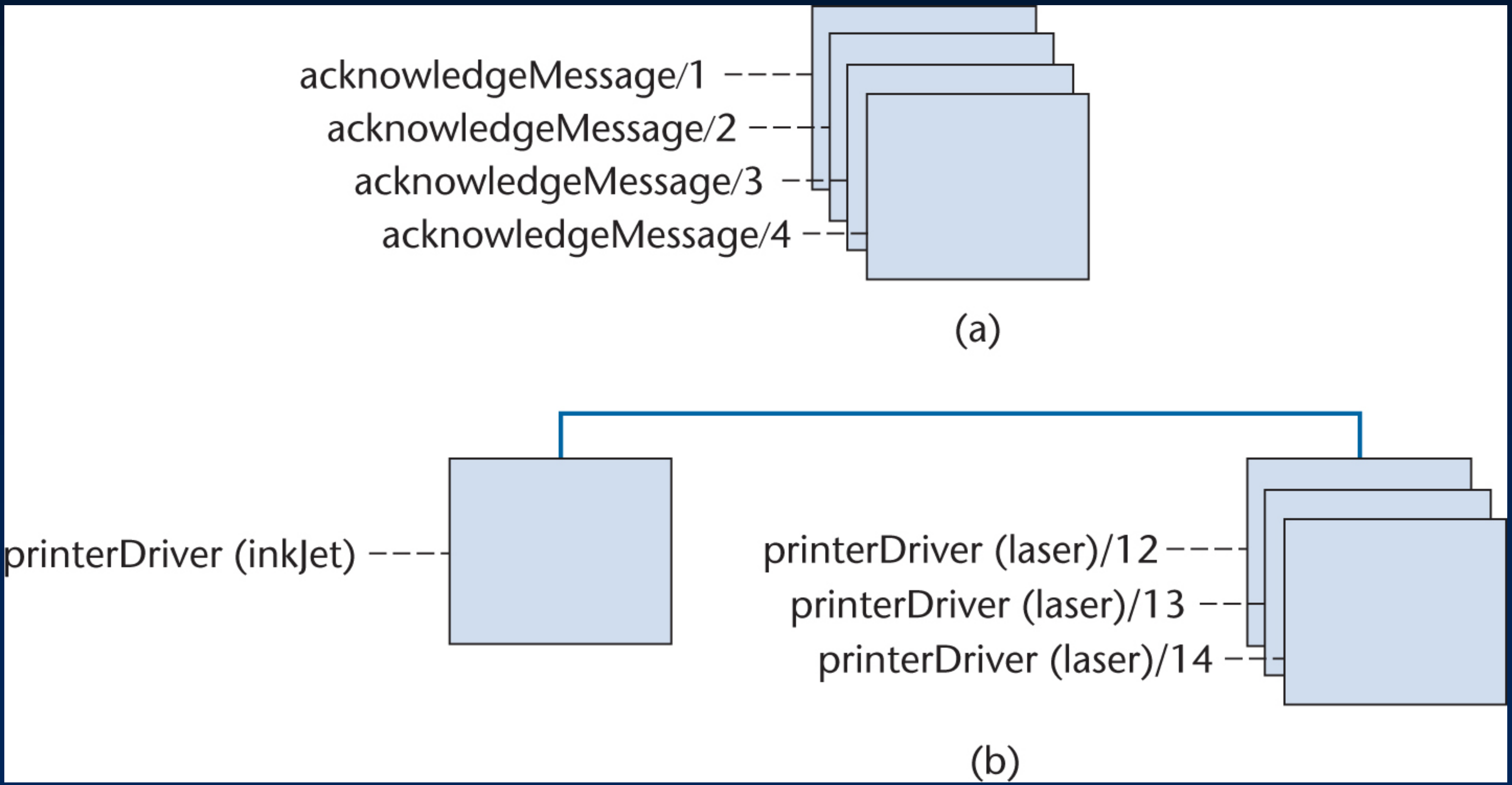

Figure 5.13

Version-Control Tool (contd)

- Problem of multiple variations
  - Deltas

- Version control is not enough — maintenance issues

5.10.1 Configuration Control during Postdelivery Maintenance

- Two programmers are working on the same artifact `mDual/16`

- The changes of the first programmer are contained in `mDual/17`

- The changes of the second programmer are contained in `mDual/18`
  - The changes of the first programmer are lost

5.10.2 Baselines

- The maintenance manager must set up
  - Baselines
  - Private workspaces

- When an artifact is to be changed, the current version is *frozen*
  - Thereafter, it can never be changed

Baselines (contd)

- Both programmers make their changes to `mDual/16`

- The first programmer
  - Freezes `mDual/16` and makes changes to it
  - The resulting revision is `mDual/17`
  - After testing, `mDual/17` becomes the new baseline

- The second programmer
  - Freezes `mDual/17` and makes changes to it
  - The resulting revision is `mDual/18`
  - After testing, `mDual/18` becomes the new baseline

5.10.3 Configuration Control during Development

- While an artifact is being coded
  - The programmer performs informal testing
- Then the artifact is given to the SQA group for methodical testing
  - Changes from now on can impact the product
- An artifact must be subject to configuration control from the time it is passed by SQA

Configuration-Control Tools

- UNIX version-control tools
  - *sccs*
  - *rcs*
  - *cvs*

- Popular commercial configuration-control tools
  - PVCS
  - SourceSafe

- Open-source configuration-control tools
  - *cvs*
  - Subversion

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

5.12 Productivity Gains with CASE Tools

- Survey of 45 companies in 10 industries (1992)
  - Half information systems
  - Quarter scientific software
  - Quarter real-time aerospace software

- Results
  - About 10% annual productivity gains
  - Cost: $125,000 per seat

Productivity Gains with CASE Tools (contd)

- Justifications for CASE
  - Faster development
  - Fewer faults
  - Easier maintenance
  - Improved morale

Productivity Gains with CASE Tools (contd)

- Newer results on fifteen Fortune 500 companies (1997)

- It is vital to have
  - Training, and
  - A software process

- Results confirm that CASE environments should be used at CMM level 3 or higher

- “A fool with a tool is still a fool”

Summary of Tools in Chapter 5

Analytical Tools

Cost–benefit analysis (Section 5.2)
Divide-and-conquer (Section 5.3)
Metrics (Section 5.5)
Separation of concerns (Section 5.4)
Stepwise refinement (Section 5.1)

CASE Taxonomy

Environment (Section 5.7)
LowerCASE tool (Section 5.7)
UpperCASE tool (Section 5.7)
Workbench (Section 5.7)

CASE Tools

Build tool (Section 5.11)
Coding tool (Section 5.8)
Configuration-control tool (Section 5.10)
Consistency checker (Section 5.7)
Data dictionary (Section 5.7)
E-mail (Section 5.8)
Interface checker (Section 5.8)
Online documentation (Section 5.8)
Operating system front end (Section 5.8)
Pretty printer (Section 5.8)
Report generator (Section 5.7)
Screen generator (Section 5.7)
Source-level debugger (Section 5.8)
Spreadsheet (Section 5.8)
Structure editor (Section 5.8)
Version-control tool (Section 5.9)
Word processor (Section 5.8)
World Wide Web browser (Section 5.8)

Figure 5.14