Object-Oriented and Classical Software Engineering

Eighth Edition, WCB/McGraw-Hill, 2011

Stephen R. Schach

CHAPTER 14

DESIGN

Overview

Slide 14.3

- Design and abstraction
- Operation-oriented design
- Data flow analysis
- Transaction analysis
- Data-oriented design
- Object-oriented design
- Object-oriented design: The elevator problem case study
- Object-oriented design: The MSG Foundation case study

Overview (contd)

Slide 14.4

- The design workflow
- The test workflow: Design
- Formal techniques for detailed design
- Real-time design techniques
- CASE tools for design
- Metrics for design
- Challenges of the design workflow

Data and Actions

Slide 14.5

- Two aspects of a product
  - Actions that operate on data
  - Data on which actions operate
- The two basic ways of designing a product
  - Operation-oriented design
  - Data-oriented design
- Third way
  - Hybrid methods
  - For example, object-oriented design

14.1 Design and Abstraction

Slide 14.6

- Classical design activities
  - Architectural design
  - Detailed design
  - Design testing
- Architectural design
  - Input: Specifications
  - Output: Modular decomposition
- Detailed design
  - Each module is designed
    - » Specific algorithms, data structures

14.2 Operation-Oriented Design

Slide 14.7

- Data flow analysis
  - Use it with most specification methods (Structured Systems Analysis here)
- Key point: We have detailed action information from the DFD

Figure 14.1

Data Flow Analysis

Slide 14.8

- Every product transforms input into output
- Determine
  - "Point of highest abstraction of input"
  - "Point of highest abstract of output"

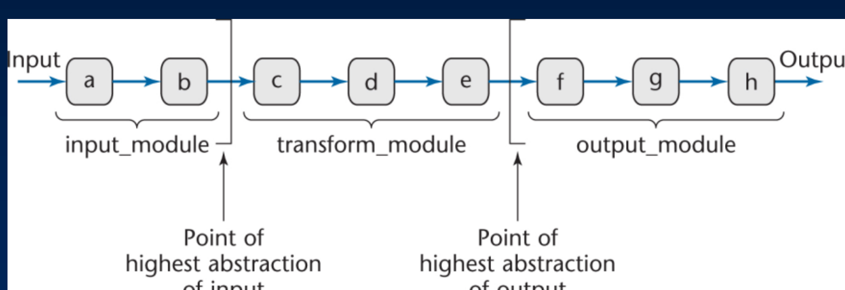

Figure 14.2

Data Flow Analysis (contd)

Slide 14.9

- Decompose the product into three modules
- Repeat stepwise until each module has high cohesion
  – Minor modifications may be needed to lower the coupling

14.3.1 Mini Case Study: Word Counting

Slide 14.10

- Example:

  Design a product which takes as input a file name, and returns the number of words in that file (like UNIX *wc*)

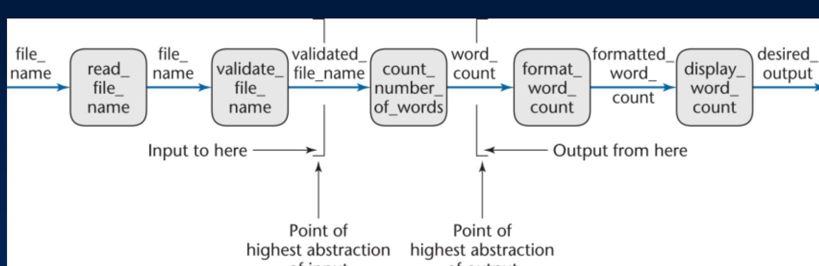

Figure 14.3

Mini Case Study: Word Counting (contd)

Slide 14.11

- First refinement

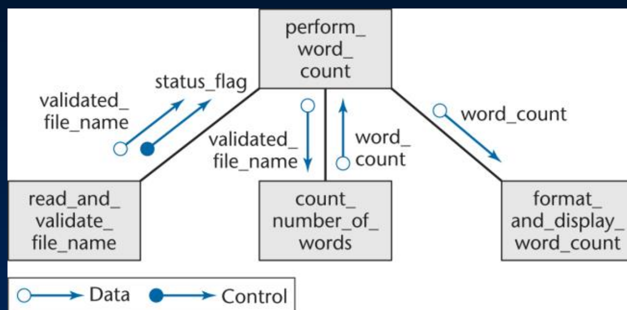

Figure 14.4

- Now refine the two modules of communicational cohesion

Mini Case Study: Word Counting (contd)

Slide 14.12

- Second refinement

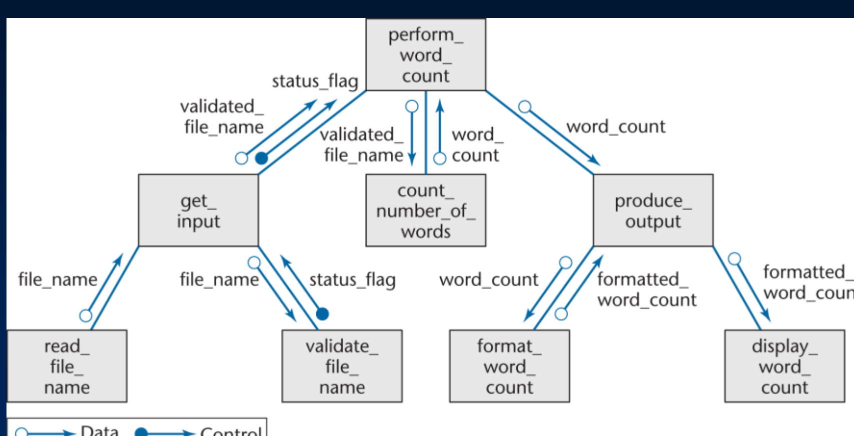

Figure 14.5

- All eight modules now have functional cohesion

Word Counting: Detailed Design

Slide 14.13

- The architectural design is complete
  - So proceed to the detailed design
- Two formats for representing the detailed design:
  - Tabular
  - Pseudocode (PDL — program design language)

Detailed Design: Tabular Format

Slide 14.14

Module name	**read_file_name**
Module type	Function
Return type	**string**
Input arguments	None
Output arguments	None
Error messages	None
Files accessed	None
Files changed	None
Modules called	None
Narrative	The product is invoked by the user by means of the command string **word_count <file_name>** Using an operating system call, this module accesses the contents of the command string input by the user, extracts <**file_name**>, and returns it as the value of the module.

Figure 14.6(a)

Detailed Design: Tabular Format (contd)

Slide 14.15

Module name	**validate_file_name**
Module type	Function
Return type	**Boolean**
Input arguments	**file_name : string**
Output arguments	None
Error messages	None
Files accessed	None
Files changed	None
Modules called	None
Narrative	This module makes an operating system call to determine whether file **file_name** exists. The module returns **true** if the file exists and **false** otherwise.

Figure 14.6(b)

Detailed Design: Tabular Format (contd)

Slide 14.16

Module name	**count_number_of_words**
Module type	Function
Return type	**integer**
Input arguments	**validated_file_name : string**
Output arguments	None
Error messages	None
Files accessed	None
Files changed	None
Modules called	None
Narrative	This module determines whether **validated_file_name** is a text file, that is, divided into lines of characters. If so, the module returns the number of words in the text file; otherwise, the module returns **−1**.

Figure 14.6(c)

Detailed Design: Tabular Format (contd)

Slide 14.17

Module name	**produce_output**
Module type	Function
Return type	**void**
Input arguments	**word_count : integer**
Output arguments	None
Error messages	None
Files accessed	None
Files changed	None
Modules called	**format_word_count** arguments: **word_count : integer** **formatted_word_count : string** **display_word_count** arguments: **formatted_word_count : string**
Narrative	This module takes the integer **word_count** passed to it by the calling module and calls **format_word_count** to have that integer formatted according to the specifications. Then it calls **display_word_count** to have the line printed.

Figure 14.6(d)

Detailed Design: PDL Format

Slide 14.18

```
void perform_word_count ( )
{
  String          validated_file_name;
  Int             word_count;

  if (get_input (validated_file_name) is null)
     print "error 1: file does not exist";
  else
  {
     set word_count equal to count_number_of_words (validated_file_name);
     if (word_count is equal to –1)
        print "error 2: file is not a text file";
     else
        produce_output (word_count);
  }
}

String get_input ( )
{
     String                 file_name;

     file_name = read_file_name ( );
     if (validate_file_name (file_name) is true)
     {
        return file_name;
     }
     else
        return null;
}

void display_word_count (String formatted_word_count)
{
     print formatted_word_count, left justified;
}

String format_word_count (int word_count);
{
     return "File contains" word_count "words";
}
```

Figure 14.7

14.3.2 Data Flow Analysis Extensions

Slide 14.19

- In real-world products, there is
  – More than one input stream, and
  – More than one output stream

Data Flow Analysis Extensions (contd)

Slide 14.20

- Find the point of highest abstraction for each stream

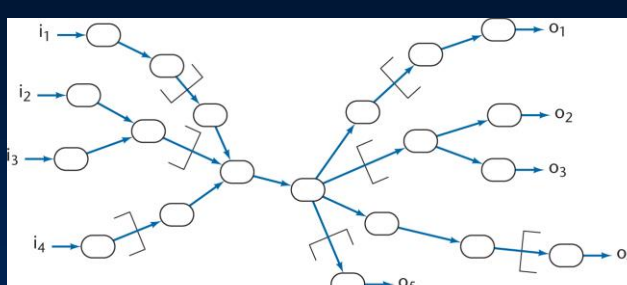

Figure 14.8

- Continue until each module has high cohesion
  – Adjust the coupling if needed

14.4 Transaction Analysis

Slide 14.21

- DFA is poor for transaction processing products
  - Example: ATM (automated teller machine)

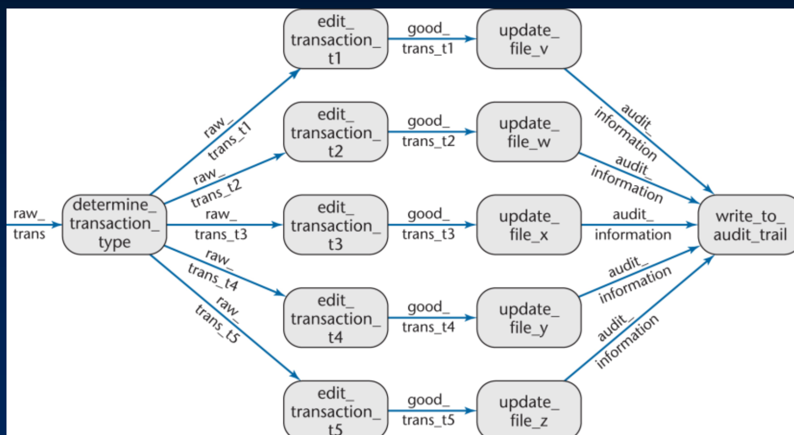

Figure 14.9

Transaction Analysis (contd)

Slide 14.22

- This is a poor design
  - There is logical cohesion and control coupling

Corrected Design Using Transaction Analysis

Slide 14.23

- Software reuse
- Have one generic `edit` module, one generic `update` module
- Instantiate them 5 times

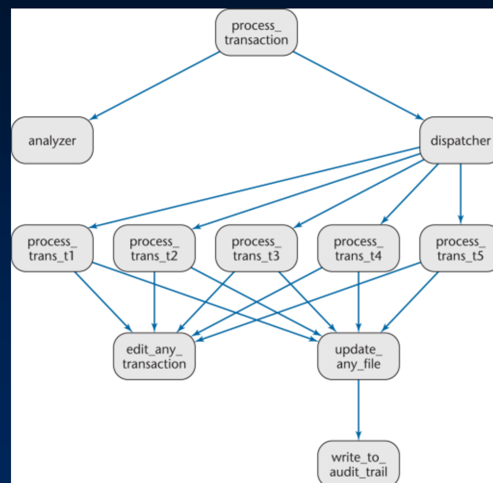

Figure 14.10

14.5 Data-Oriented Design

Slide 14.24

- Basic principle
  – The structure of a product must conform to the structure of its data
- Three very similar methods
  – Michael Jackson [1975], Warnier [1976], Orr [1981]
- Data-oriented design
  – Has never been as popular as action-oriented design
  – With the rise of OOD, data-oriented design has largely fallen out of fashion

14.6 Object-Oriented Design (OOD)

Slide 14.25

- Aim
  - Design the product in terms of the classes extracted during OOA
- If we are using a language without inheritance (e.g., C, Ada 83)
  - Use abstract data type design
- If we are using a language without a type statement (e.g., FORTRAN, COBOL)
  - Use data encapsulation

Object-Oriented Design Steps

Slide 14.26

- OOD consists of two steps:
- Step 1. Complete the class diagram
  - Determine the formats of the attributes
  - Assign each method, either to a class or to a client that sends a message to an object of that class
- Step 2. Perform the detailed design

Object-Oriented Design Steps (contd)

Slide 14.27

- Step 1. Complete the class diagram
  - The formats of the attributes can be directly deduced from the analysis artifacts
- Example: Dates
  - U.S. format (mm/mm/yyyy)
  - European format (dd/mm/yyyy)
  - In both instances, 10 characters are needed
- The formats could be added during analysis
  - To minimize rework, *never* add an item to a UML diagram until strictly necessary

Object-Oriented Design Steps (contd)

Slide 14.28

- Step 1. Complete the class diagram
  - Assign each method, either to a class or to a client that sends a message to an object of that class
- Principle A: Information hiding
- Principle B: If an operation is invoked by many clients of an object, assign the method to the object, not the clients
- Principle C: Responsibility-driven design

14.7 Object-Oriented Design: The Elevator Problem Case Study

Slide 14.29

- Step 1. Complete the class diagram
- Consider the second iteration of the CRC card for the elevator controller

OOD: Elevator Problem Case Study (contd)

Slide 14.30

- CRC card

CLASS
Elevator Controller Class
RESPONSIBILITY
1. Send message to **Elevator Button Class** to turn on button 2. Send message to **Elevator Button Class** to turn off button 3. Send message to **Floor Button Class** to turn on button 4. Send message to **Floor Button Class** to turn off button 5. Send message to **Elevator Class** to move up one floor 6. Send message to **Elevator Class** to move down one floor 7. Send message to **Elevator Doors Class** to open 8. Start timer 9. Send message to **Elevator Doors Class** to close after timeout 10. Check requests 11. Update requests
COLLABORATION
1. **Elevator Button Class** (subclass) 2. **Floor Button Class** (subclass) 3. **Elevator Doors Class** 4. **Elevator Class**

Figure 13.9 (again)

OOD: Elevator Problem Case Study (contd)

Slide 14.31

- Responsibilities
  - 8. Start timer
  - 10. Check requests, and
  - 11. Update requests

  are assigned to the elevator controller

- Because they are carried out by the elevator controller

OOD: Elevator Problem Case Study (contd)

Slide 14.32

- The remaining eight responsibilities have the form
  - "Send a message to another class to tell it do something"
- These should be assigned to that other class
  - Responsibility-driven design
  - Safety considerations
- Methods `open doors`, `close doors` are assigned to class **Elevator Doors Class**
- Methods `turn off button`, `turn on button` are assigned to classes **Floor Button Class** and **Elevator Problem Class**

Detailed Class Diagram: Elevator Problem

Slide 14.33

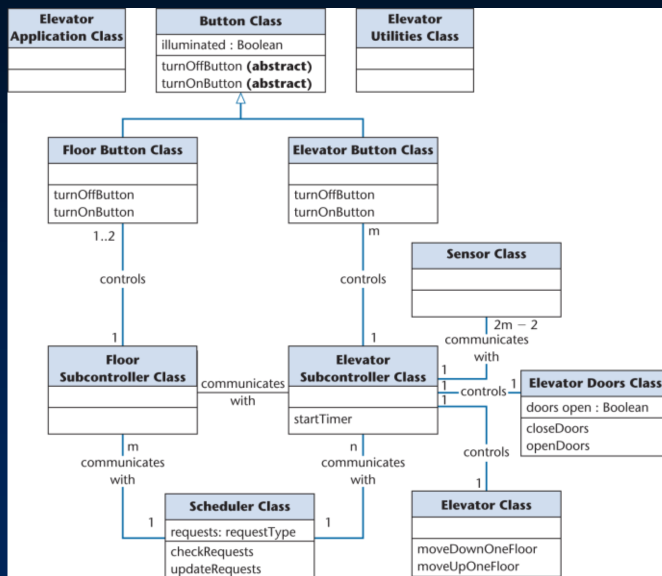

Figure 14.11

Detailed Design: Elevator Problem

Slide 14.34

- Detailed design of `elevatorEventLoop` is constructed from the statechart

```
void elevatorSubcontrollerEventLoop (void)
{
  while (TRUE)
  {
    if (an elevatorButton has been pressed)
       if (elevatorButton is off)
       {
             elevatorButton::turnOnButton;
             scheduler::newRequestMade;
       }
    else if (elevator is moving up)
    {
       wait for sensor message that elevator is arriving at floor;
       scheduler::checkRequests;
       if (there is no request to stop at floor f)
          elevator::moveUpOneFloor;
       else
       {
          stop elevator by not sending a message to move;
          if (elevatorButton is on)
             elevatorButton::turnOffButton;
          elevatorDoors::openDoors;
          startTimer;
    }
  }
  else if (elevator is moving down)
       [similar to up case]
  else if (elevator is stopped and request is pending)
  {
       wait for timeout;
       elevatorDoors::closeDoors;
       determine direction of next request;
       elevator::moveUp/DownOneFloor;
       wait for sensor message that elevator has left floor;
       floorSubcontroller::elevatorHasLeftFloor;
    }
    else if (elevator is at rest and not (request is pending))
    {
       wait for timeout;
       elevatorDoors::closeDoors;
    }
    else
       there are no requests, elevator is stopped with elevatorDoors closed, so do nothing;
  }
}
```

 Figure 14.12

14.8 Object-Oriented Design: The MSG Foundation Case Study

- Step 1. Complete the class diagram

- The final class diagram is shown in the next slide
  - **Date Class** is needed for C++
  - Java has built-it functions for handling dates

Final Class Diagram: MSG Foundation

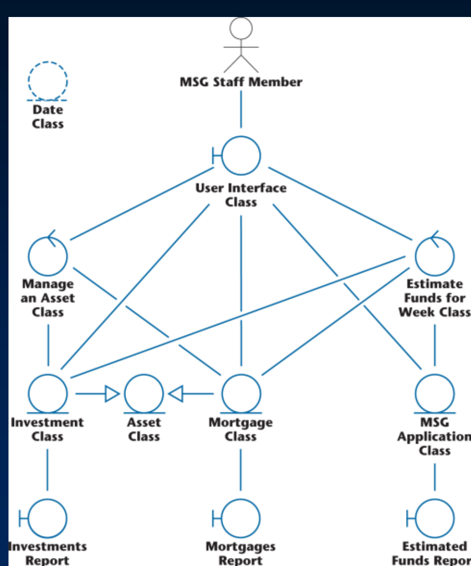

Figure 14.13

Class Diagram with Attributes: MSG Foundation

Slide 14.37

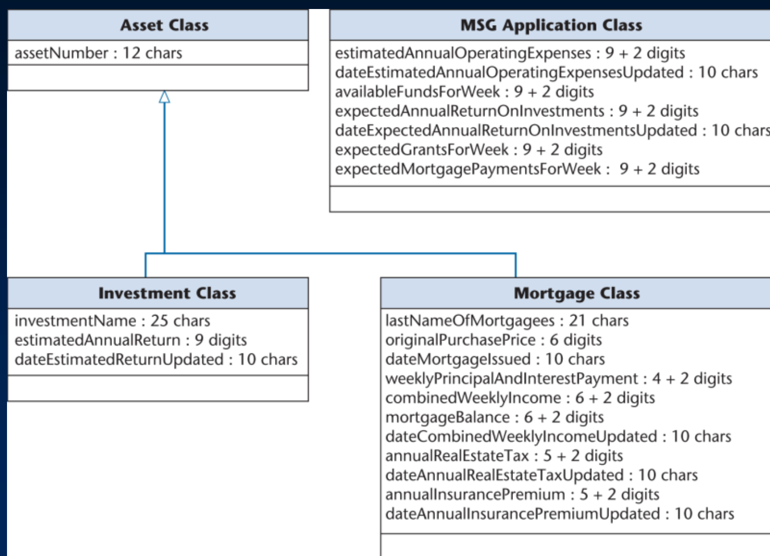

Figure 14.14

Assigning Methods to Classes: MSG Foundation

Slide 14.38

- Example: `setAssetNumber, getAssetNumber`
  – From the inheritance tree, these accessor/mutator methods should be assigned to **Asset Class**
  – So that they can be inherited by both subclasses of **Asset Class (Investment Class** and **Mortgage Class)**

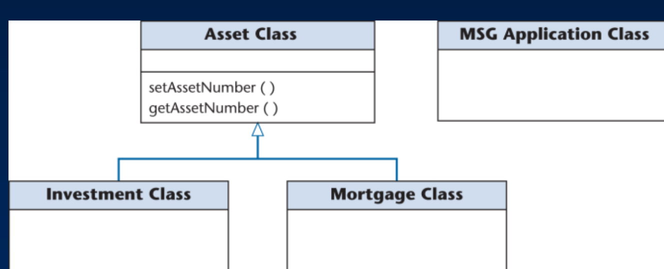

Figure 14.15

Assigning Methods to Classes: MSG Foundation (contd)

Slide 14.39

- Assigning the other methods is equally straightforward
  - See Appendix G

Detailed Design: MSG Foundation

Slide 14.40

- Determine what each method does
- Represent the detailed design in an appropriate format
  - PDL (pseudocode) here

Method `EstimateFundsForWeek::computeEstimatedFunds`

Slide 14.41

```
public static void computeEstimatedFunds( )
This method computes the estimated funds available for the week.
{
  float expectedWeeklyInvestmentReturn;                 (expected weekly investment return)
  float expectedTotalWeeklyNetPayments = (float) 0.0;
                                                        (expected total mortgage payments
                                                        less total weekly grants)
  float estimatedFunds = (float) 0.0;                   (total estimated funds for week)
Create an instance of an investment record.
  Investment inv = new Investment ( );
Create an instance of a mortgage record.
  Mortgage mort = new Mortgage ( );
Invoke method totalWeeklyReturnOnInvestment.
  expectedWeeklyInvestmentReturn = inv.totalWeeklyReturnOnInvestment ( );
Invoke method expectedTotalWeeklyNetPayments            (see Figure 14.17)
  expectedTotalWeeklyNetPayments = mort.totalWeeklyNetPayments ( );
Now compute the estimated funds for the week.
  estimatedFunds = (expectedWeeklyInvestmentReturn
       – (MSGApplication.getAnnualOperatingExpenses ( ) / (float) 52.0)
       + expectedTotalWeeklyNetPayments);

Store this value in the appropriate location.
  MSGApplication.setEstimatedFundsForWeek (estimatedFunds);
} // computeEstimatedFunds
```

Figure 14.16

Method `Mortgage::totalWeeklyNetPayments`

Slide 14.42

```
public float totalWeeklyNetPayments ( )
This method computes the net total weekly payments made by the mortgagees, that is, the expected total weekly
mortgage amount less the expected total weekly grants.
{
  File mortgageFile = new File ("mortgage.dat");                 (file of mortgage records)
  float expectedTotalWeeklyMortgages = (float) 0.0;              (expected total weekly mortgage payments)
  float expectedTotalWeeklyGrants = (float) 0.0;                 (expected total weekly grants)
  float interestPayment;                                         (interest payment)
  float escrowPayment;                                           (escrow payment)
  float capitalRepayment;                                        (capital repayment)
  float weeklyPayment;                                           (mortgage payment for week)
  float maximumPermittedMortgagePayment;                         (maximum amount the couple may pay)
Open the file of mortgages, name it inFile, and read each element in turn.
{
  read (inFile);
Compute the interest payment, escrow payment, and capital repayment for this mortgage.
  interestPayment = mortgageBalance * INTEREST_RATE / WEEKS_IN_YEAR ;
  escrowPayment = (annualPropertyTax + annualInsurancePremium) / WEEKS_IN_YEAR;
  capitalRepayment = weeklyPrincipalAndInterestPayment – interestPayment;
  mortgageBalance –= capitalRepayment;
First assume that the couple can pay the mortgage in full, without a grant.
  weeklyPayment = weeklyPrincipalAndInterestPayment + escrowPayment;
Add the weekly Principal and Interest payment to the running total of mortgage payments
  expectedTotalWeeklyMortgages += weeklyPrincipalAndInterestPayment;
Now determine how much the couple can actually pay.
  maximumPermittedMortgagePayment = currentWeeklyIncome *
      MAXIMUM_PERC_OF_INCOME;
If a grant is needed, add the grant amount to the running total of grants
  if (weeklyPayment > maximumPermittedMortgagePayment)
  expectedTotalWeeklyGrants += weeklyPayment – maximumPermittedMortgagePayment;
  }
Close the file of mortgages. Return the total expected net payments for the week.
  return (expectedTotalWeeklyMortgages – expectedTotalWeeklyGrants);
} // totalWeeklyNetPayments
```

Figure 14.17

14.9 The Design Workflow

Slide 14.43

- Summary of the design workflow:
  – The analysis workflow artifacts are iterated and integrated until the programmers can utilize them

- Decisions to be made include:
  – Implementation language
  – Reuse
  – Portability

The Design Workflow (contd)

Slide 14.44

- The idea of decomposing a large workflow into independent smaller workflows (*packages*) is carried forward to the design workflow

- The objective is to break up the upcoming implementation workflow into manageable pieces
  – *Subsystems*

- It does not make sense to break up the MSG Foundation case study into subsystems — it is too small

The Design Workflow (contd)

Slide 14.45

- Why the product is broken into subsystems:
  - It is easier to implement a number of smaller subsystems than one large system
  - If the subsystems are independent, they can be implemented by programming teams working in parallel
    - The software product as a whole can then be delivered sooner

The Design Workflow (contd)

Slide 14.46

- The *architecture* of a software product includes
  - The various components
  - How they fit together
  - The allocation of components to subsystems
- The task of designing the architecture is specialized
  - It is performed by a software *architect*

The Design Workflow (contd)

Slide 14.47

- The architect needs to make *trade-offs*
  - Every software product must satisfy its functional requirements (the use cases)
  - It also must satisfy its nonfunctional requirements, including
    - Portability, reliability, robustness, maintainability, and security
  - It must do all these things within budget and time constraints
- The architect must assist the client by laying out the trade-offs

The Design Workflow (contd)

Slide 14.48

- It is usually impossible to satisfy all the requirements, functional and nonfunctional, within the cost and time constraints
  - Some sort of compromises have to be made
- The client has to
  - Relax some of the requirements;
  - Increase the budget; and/or
  - Move the delivery deadline

The Design Workflow (contd)

Slide 14.49

- The architecture of a software product is critical
  - The requirements workflow can be fixed during the analysis workflow
  - The analysis workflow can be fixed during the design workflow
  - The design workflow can be fixed during the implementation workflow
- But there is no way to recover from a suboptimal architecture
  - The architecture must immediately be redesigned

14.10 The Test Workflow: Design

Slide 14.50

- Design reviews must be performed
  - The design must correctly reflect the specifications
  - The design itself must be correct
- Transaction-driven inspections
  - Essential for transaction-oriented products
  - However, they are insufficient — specification-driven inspections are also needed

14.11 The Test Workflow: The MSG Foundation Case Study

- A design inspection must be performed
  - All aspects of the design must be checked
- Even if no faults are found, the design may be changed during the implementation workflow

14.12 Formal Techniques for Detailed Design

- Implementing a complete product and then proving it correct is hard
- However, use of formal techniques during detailed design can help
  - Correctness proving can be applied to module-sized pieces
  - The design should have fewer faults if it is developed in parallel with a correctness proof
  - If the same programmer does the detailed design and implementation
    - The programmer will have a positive attitude to the detailed design
    - This should lead to fewer faults

14.13 Real-Time Design Techniques

Slide 14.53

- Difficulties associated with real-time systems
  - Inputs come from the real world
    - Software has no control over the timing of the inputs
  - Frequently implemented on distributed software
    - Communications implications
    - Timing issues
  - Problems of synchronization
    - Race conditions
    - Deadlock (deadly embrace)

Real-Time Design Techniques (contd)

Slide 14.54

- The major difficulty in the design of real-time systems
  - Determining whether the timing constraints are met by the design

Real-Time Design Techniques (contd)

Slide 14.55

- Most real-time design methods are extensions of non-real-time methods to real-time
- We have limited experience in the use of any real-time methods
- The state-of-the-art is not where we would like it to be

14.14 CASE Tools for Design

Slide 14.56

- It is critical to check that the design artifacts incorporate all aspects of the analysis
  - To handle analysis and design artifacts we therefore need upperCASE tools
- UpperCASE tools
  - Are built around a data dictionary
  - They incorporate a consistency checker, and
  - Screen and report generators
  - Management tools are sometimes included, for
    - » Estimating
    - » Planning

CASE Tools for Design (contd)

Slide 14.57

- Examples of tools for object-oriented design
  - Commercial tools
    - Software through Pictures
    - IBM Rational Rose
    - Together
  - Open-source tool
    - ArgoUML

14.15 Metrics for Design

Slide 14.58

- Measures of design quality
  - Cohesion
  - Coupling
  - Fault statistics
- Cyclomatic complexity is problematic
  - Data complexity is ignored
  - It is not used much with the object-oriented paradigm

Metrics for Design (contd)

Slide 14.59

- Metrics have been put forward for the object-oriented paradigm
  - They have been challenged on both theoretical and experimental grounds

14.16 Challenges of the Design Phase

Slide 14.60

- The design team should not do too much
  - The detailed design should not become code
- The design team should not do too little
  - It is essential for the design team to produce a complete detailed design

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Challenges of the Design Phase (contd)

Slide 14.61

- We need to “grow” great designers
- Potential great designers must be
  - Identified,
  - Provided with a formal education,
  - Apprenticed to great designers, and
  - Allowed to interact with other designers
- There must be a specific career path for these designers, with appropriate rewards

Overview of the MSG Foundation Case Study

Slide 14.62

Object-oriented design	Section 14.8
Overall class diagram	Figure 14.13
Part of overall class diagram with attribute formats added	Figure 14.14
Detailed design	Appendix G

Figure 14.18

Overview of the Elevator Problem Case Study

Object-oriented design	Section 14.7
Detailed class diagram	Figure 14.11

Figure 14.19