Advanced Software Engineering

CS 780
School of Computing
WSU

DESIGN

Data and Actions

- 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

- 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

- Data flow analysis
  - A classical design technique for achieving modules with high cohesion.
  - Use it with most specification methods (Structured Systems Analysis here)

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

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

Data Flow Analysis

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

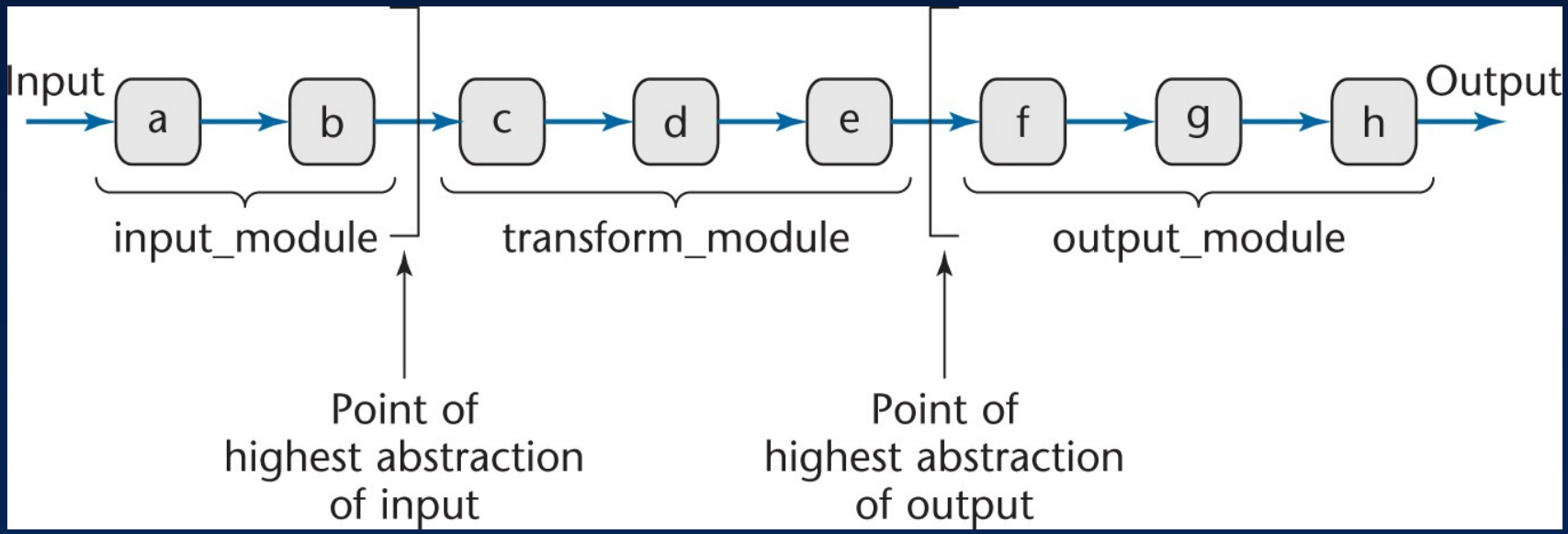


Figure 14.2

Data Flow Analysis (contd)

- Decompose the product into three modules
  - Input_module
  - Transform_module
  - Output_module
- 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

- Example:

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

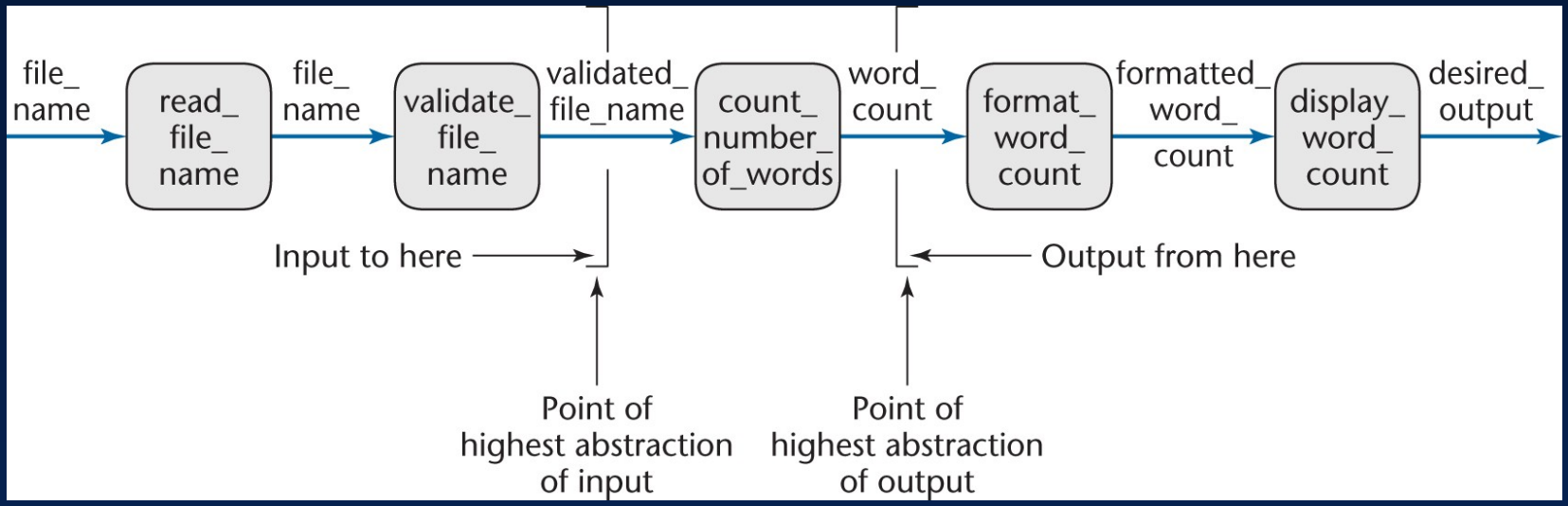


Figure 14.3

Mini Case Study: Word Counting (contd)

- First refinement

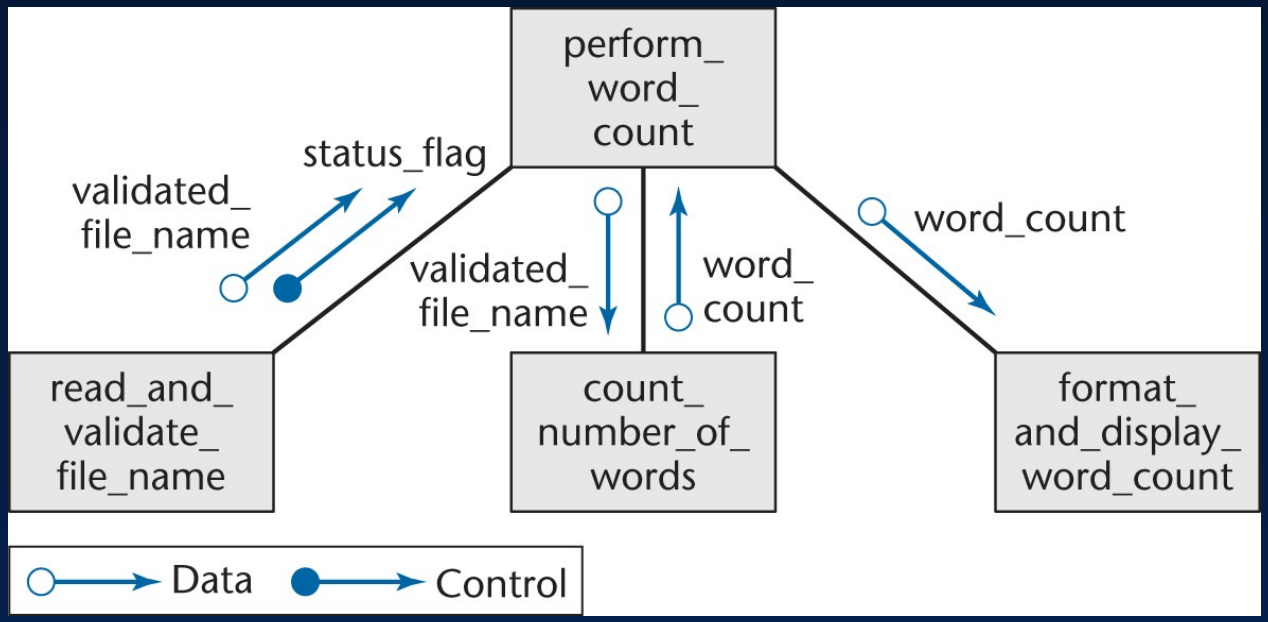


Figure 14.4

- Now refine the two modules of communicational cohesion

Mini Case Study: Word Counting (contd)

- Second refinement

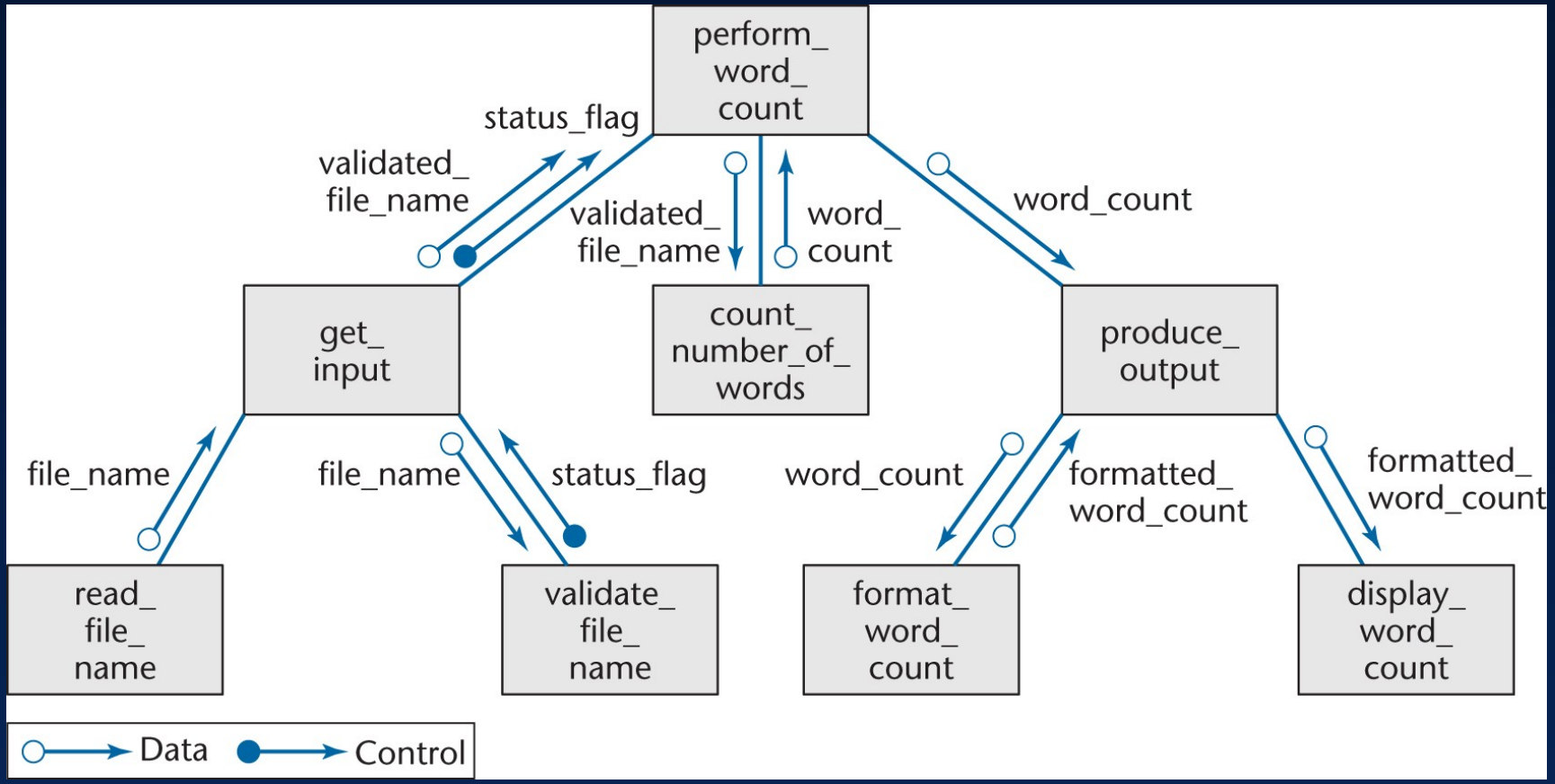


Figure 14.5

- All eight modules now have functional cohesion

Word Counting: Detailed Design

- 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

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)

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.

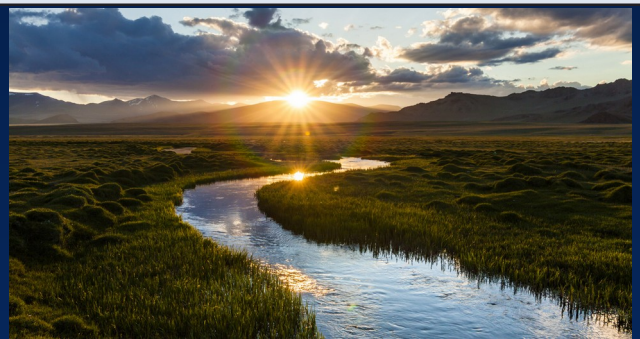


Figure 14.6(b)

Detailed Design: Tabular Format (contd)

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)

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


```
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

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

Data Flow Analysis Extensions (contd)

- Find the point of highest abstraction for each stream

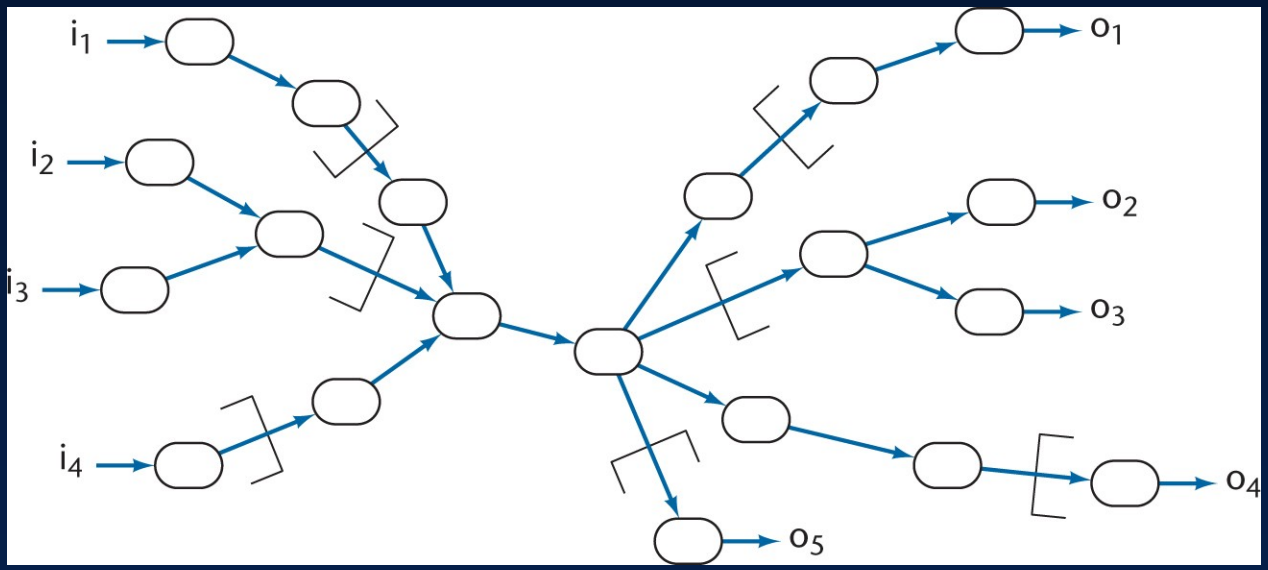


Figure 14.8

- Continue until each module has high cohesion
  - Adjust the coupling if needed (combine modules if they have high coupling, divide a module if it has low cohesion)

How to Perform Data Flow Analysis — Box 14.1

- **Iterate**

  Find the point of highest abstraction of input of each input stream.

  Find the point of highest abstraction of output of each output stream.

  Decompose the data flow diagram using these points of highest abstraction.
- **Until** the resulting modules have high cohesion.
- If a resulting coupling is too high, adjust the design.

Exercise

- Design the library circulation system using data flow analysis:

Consider an automated library circulation system. Every book has a bar code, and every borrower has a card bearing a bar code. When a borrower wishes to check out a book, the librarian scans the bar codes on the book and the borrower's card, and enters C at the computer terminal. Similarly, when a book is returned, it is again scanned and the librarian enters R. Librarians can add books (+) to the library collection or remove them (−). Borrowers can go to a terminal and determine all the books in the library by a particular author (the borrower enters A= followed by the author's name), all the books with a specific title (T= followed by the title), or all the books in a particular subject area (S= followed by the subject area). Finally, if a borrower wants a book currently checked out, the librarian can place a hold on the book so that, when it is returned, it will be held for the borrower who requested it (H= followed by the number of the book).

14.6 Object-Oriented Design (OOD)

- Aim
  - Design the product in terms of the classes extracted during OOA

Object-Oriented Design Steps

- 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
  - Determine what each method does

Object-Oriented Design Steps (contd)

- 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/dd/yyyy)
  - European format (dd/mm/yyyy)
  - In both instances, 10 characters are needed

Object-Oriented Design Steps (contd)

- 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
  - The state variables of a class should be declared private or protected. Accordingly, operations performed on the variables must be local to that class.
- Principle B: Shared operations
  - 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.
  - If a client sends a message to an object, then that object is responsible for carrying out the request of the client.

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

OOD: Elevator Problem Case Study (contd)

- 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)

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

  are assigned to the elevator controller and scheduler.

- Because they are carried out by the elevator controller and scheduler.

OOD: Elevator Problem Case Study (contd)

- 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 Button Class**

Detailed Class Diagram: Elevator Problem

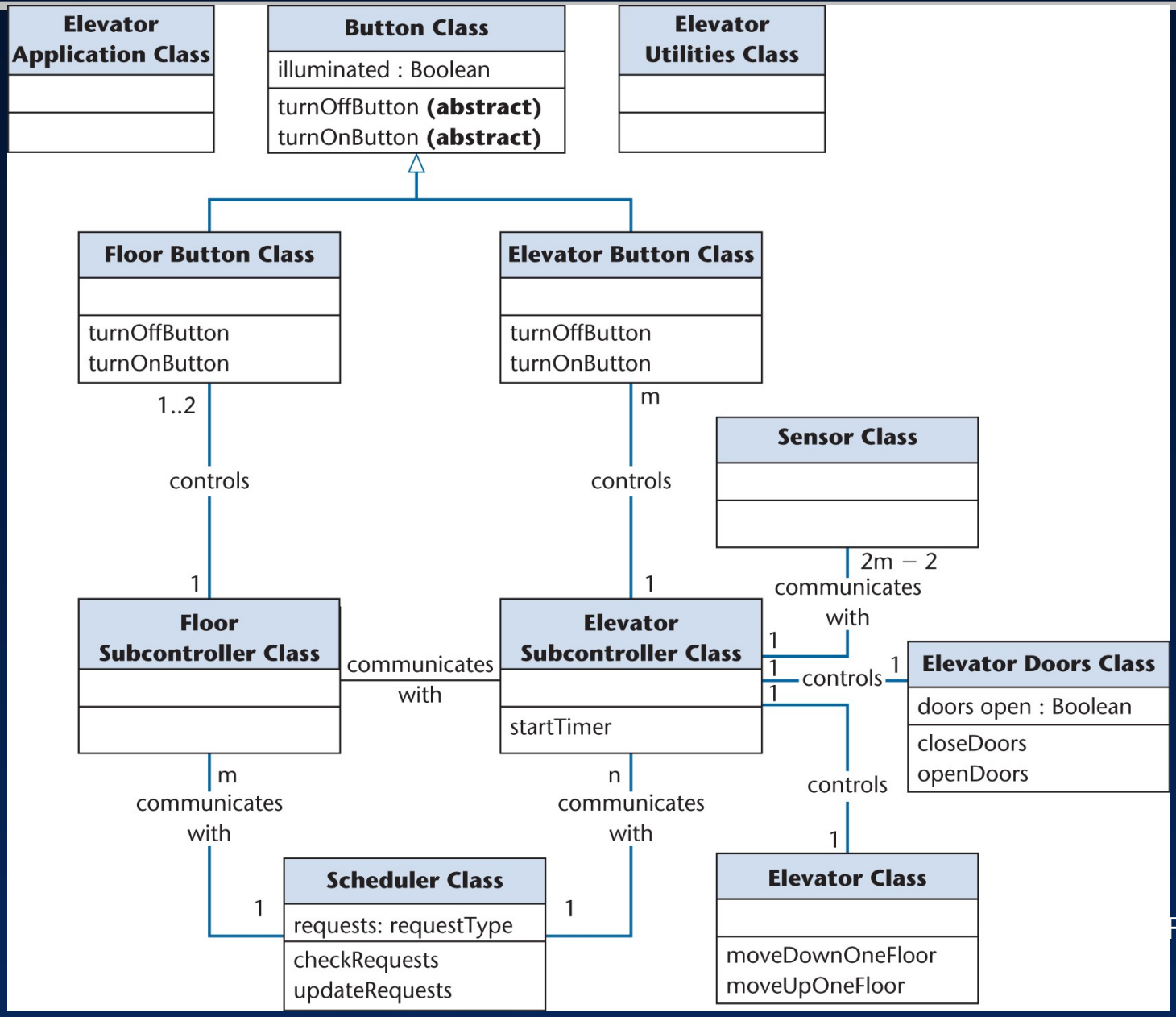


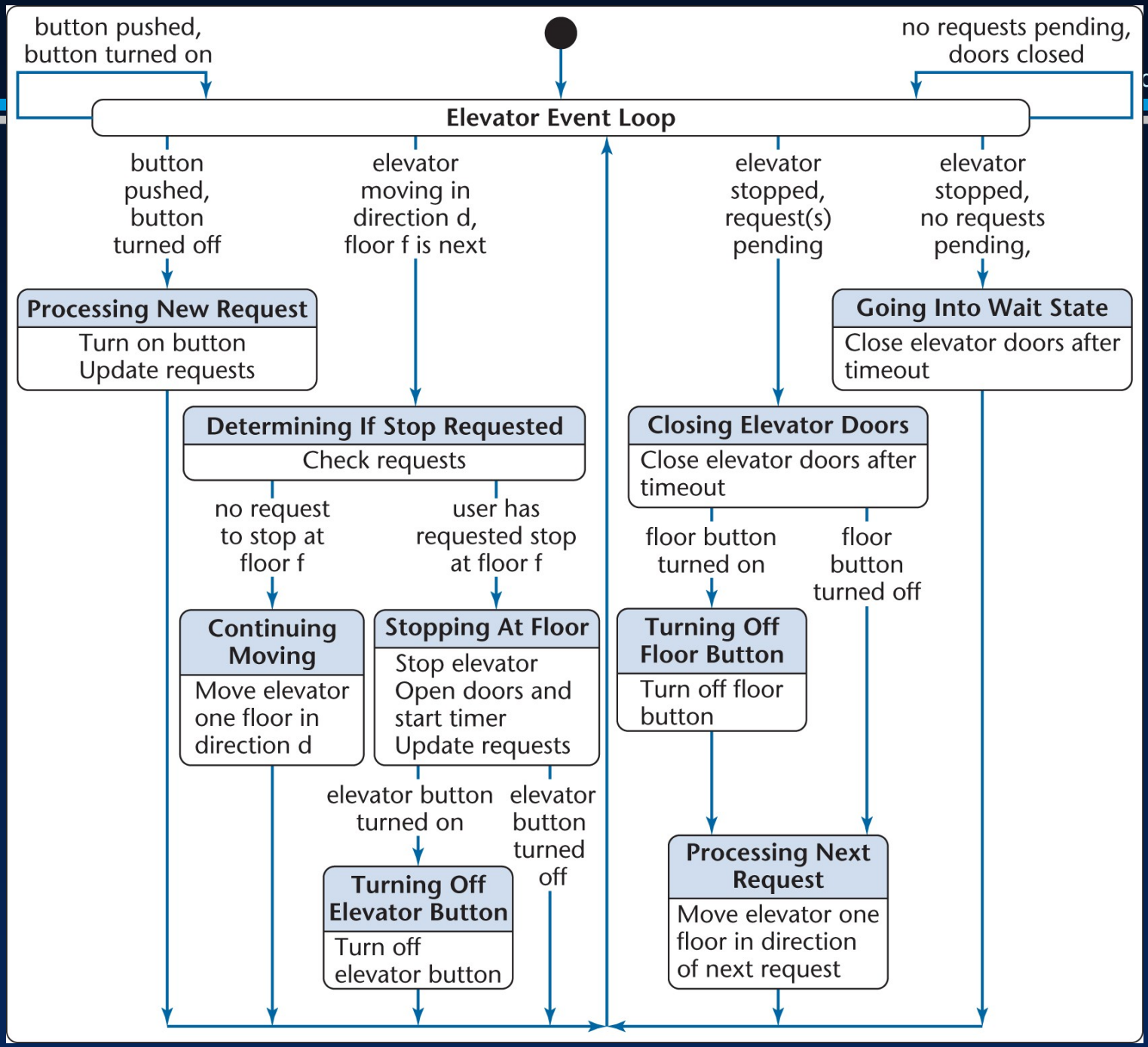
Figure 14.11

Step2: Perform Detailed Design

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

Figure 14.12

```
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
  }
}
```


button pushed, button turned on
no requests pending, doors closed
Elevator Event Loop
button pushed, button turned off
elevator moving in direction d, floor f is next
elevator stopped, request(s) pending
elevator stopped, no requests pending,
Processing New Request
Turn on button
Update requests
Going Into Wait State
Close elevator doors after timeout
Determining If Stop Requested
Check requests
Closing Elevator Doors
Close elevator doors after timeout
no request to stop at floor f
user has requested stop at floor f
floor button turned on
floor button turned off
Continuing Moving
Move elevator one floor in direction d
Stopping At Floor
Stop elevator
Open doors and start timer
Update requests
Turning Off Floor Button
Turn off floor button
elevator button turned on
elevator button turned off
Turning Off Elevator Button
Turn off elevator button
Processing Next Request
Move elevator one floor in direction of next request

- 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

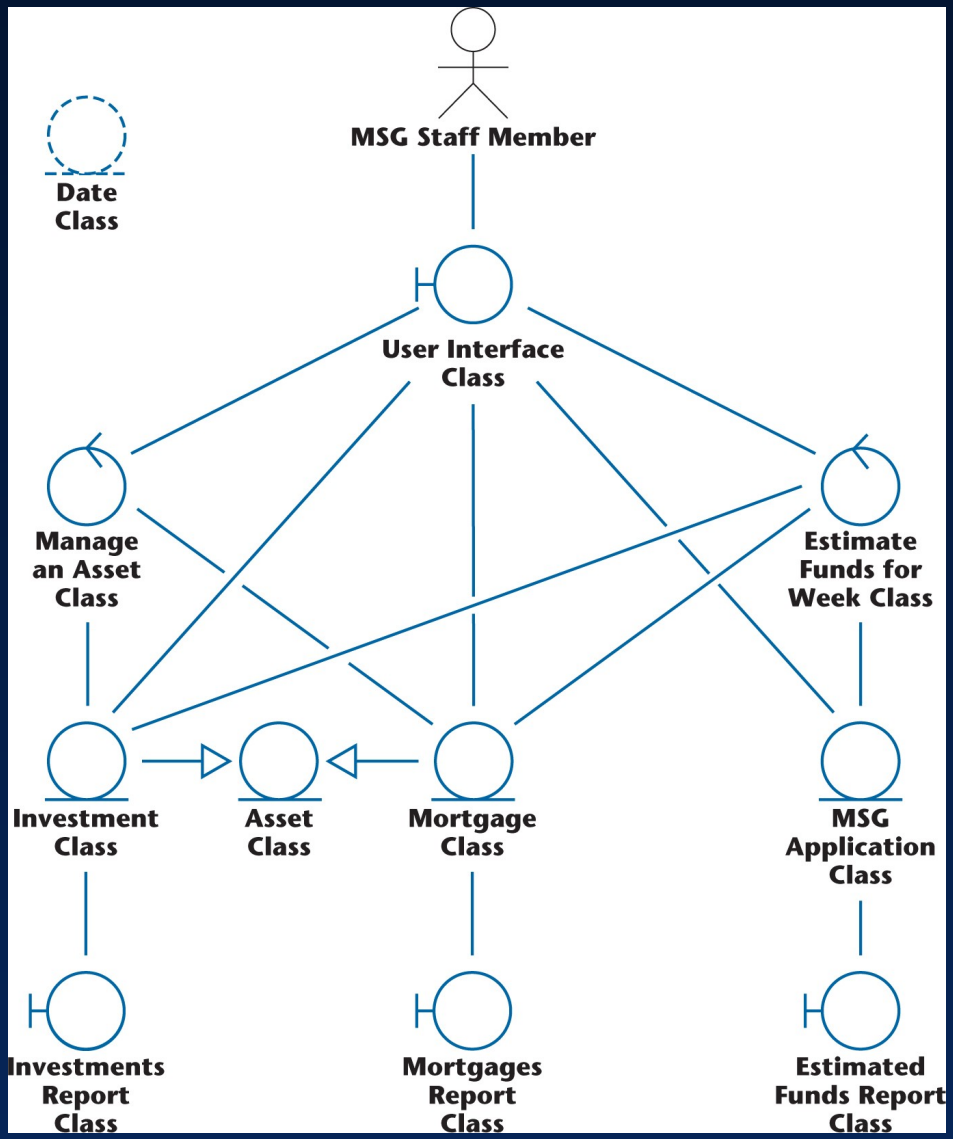


Figure 14.13

Class Diagram with Attributes: MSG Foundation

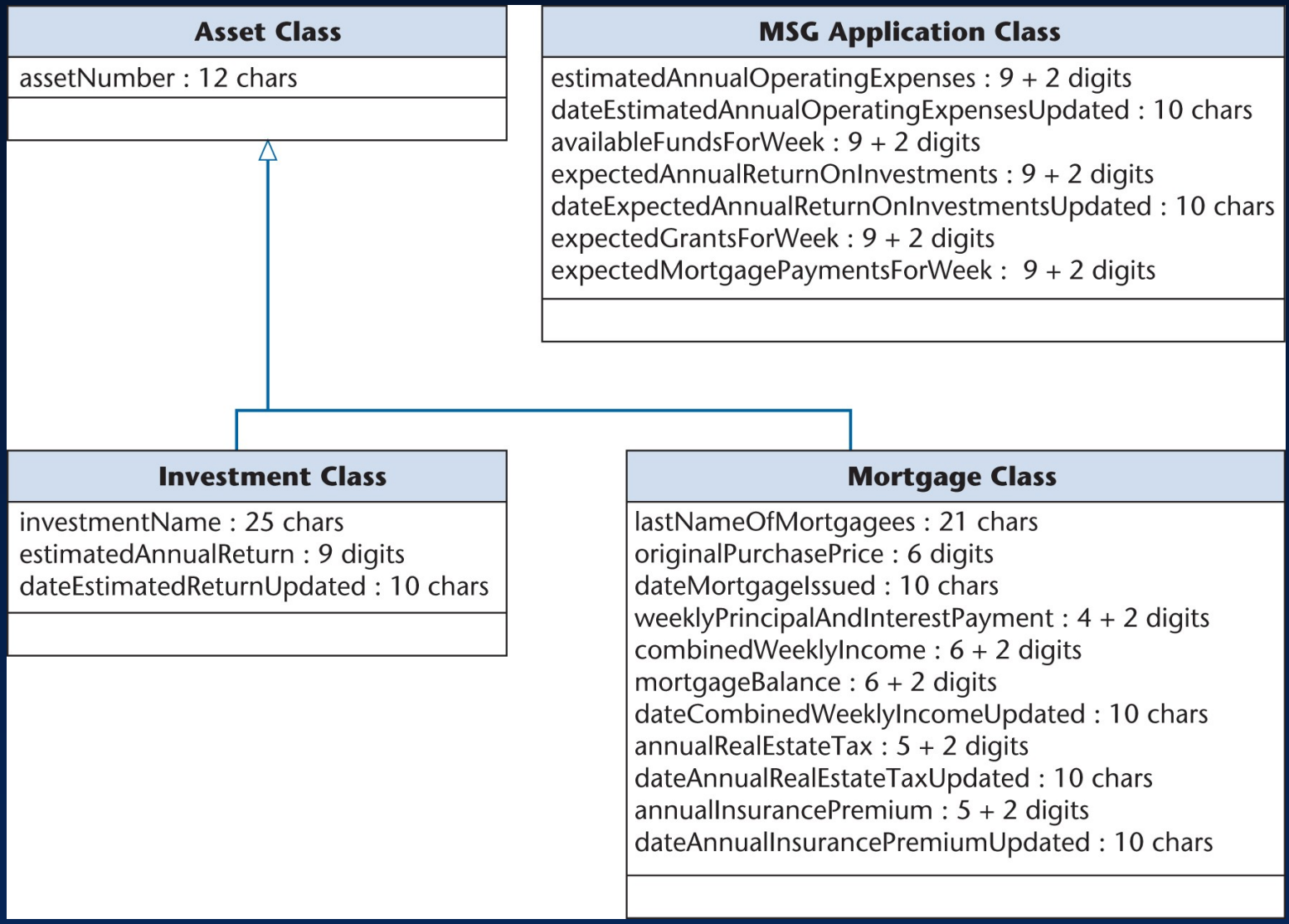


Figure 14.14

Assigning Methods to Classes: MSG Foundation

- 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)**

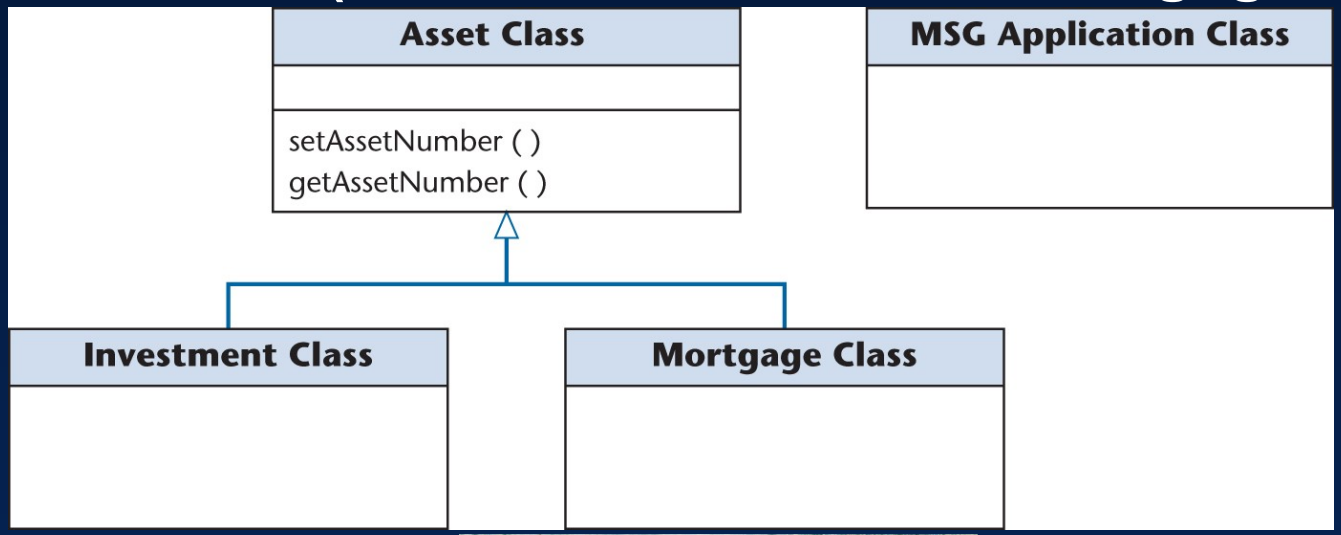


Figure 14.15

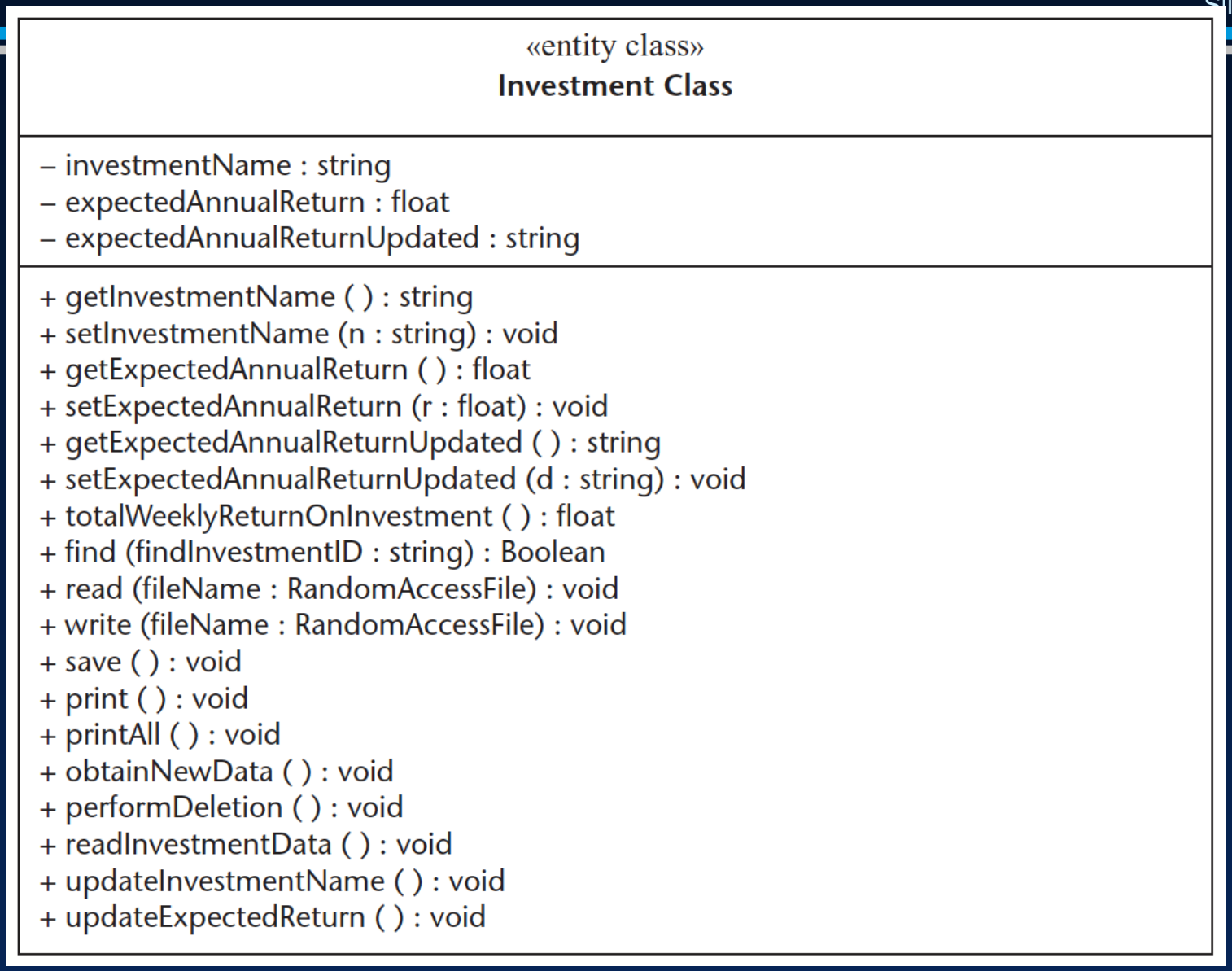
Assigning Methods to Classes: MSG Foundation (contd)

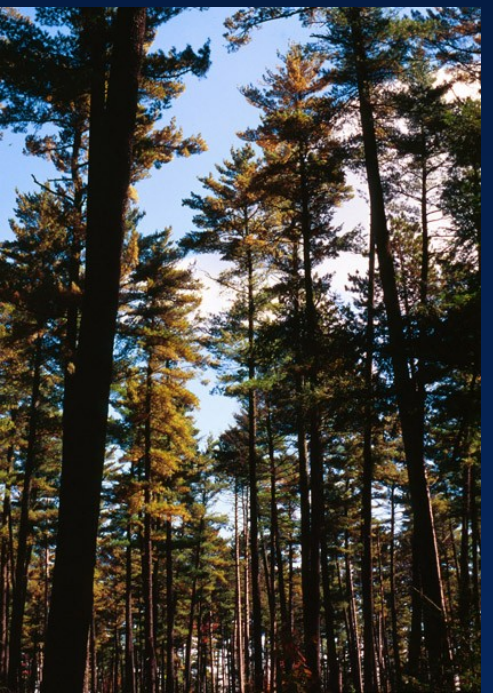
- Assigning the other methods is equally straightforward
  - See Appendix G

«entity class»
Asset Class

assetNumber : string

+ getAssetNumber ( ) : string
+ setAssetNumber (a : string) : void
+ **abstract** read (fileName : RandomAccessFile) : void
+ **abstract** obtainNewData ( ) : void
+ **abstract** performDeletion ( ) : void
+ **abstract** write (fileName : RandomAccessFile) : void
+ **abstract** save ( ) : void
+ **abstract** print ( ) : void
+ **abstract** find (s : string) : Boolean
+ delete ( ) : void
+ add ( ) : void


«entity class»
Investment Class
– investmentName : string
– expectedAnnualReturn : float
– expectedAnnualReturnUpdated : string
+ getInvestmentName ( ) : string
+ setInvestmentName (n : string) : void
+ getExpectedAnnualReturn ( ) : float
+ setExpectedAnnualReturn (r : float) : void
+ getExpectedAnnualReturnUpdated ( ) : string
+ setExpectedAnnualReturnUpdated (d : string) : void
+ totalWeeklyReturnOnInvestment ( ) : float
+ find (findInvestmentID : string) : Boolean
+ read (fileName : RandomAccessFile) : void
+ write (fileName : RandomAccessFile) : void
+ save ( ) : void
+ print ( ) : void
+ printAll ( ) : void
+ obtainNewData ( ) : void
+ performDeletion ( ) : void
+ readInvestmentData ( ) : void
+ updateInvestmentName ( ) : void
+ updateExpectedReturn ( ) : void



«entity class»

Mortgage Class

– mortgageeName : string
– price : float
– dateMortgageIssued : string
– currentWeeklyIncome : float
– weeklyIncomeUpdated : string
– annualPropertyTax : float
– annualInsurancePremium : float
– mortgageBalance : float
+ <<static final>> INTEREST_RATE : float
+ <<static final>> MAX_PER_OF_INCOME : float
+ <<static final>> NUMBER_OF_MORTGAGE_PAYMENTS : int
+ <<static final>> WEEKS_IN_YEAR : float
+ getMortgageeName ( ) : string
+ setMortgageeName (n : string) : void
+ getPrice ( ) : float
+ setPrice (p : float) : void
+ getDateMortgageIssued ( ) : string
+ setDateMortgageIssued (w : string) : void
+ getCurrentWeeklyIncome ( ) : float
+ setCurrentWeeklyIncome (i : float) : void
+ getWeeklyIncomeUpdated ( ) : string
+ setWeeklyIncomeUpdated (w : string) : void
+ getAnnualPropertyTax ( ) : float
+ setAnnualPropertyTax (t : float) : void
+ getAnnualInsurancePremium ( ) : float
+ setAnnualInsurancePremium (p : float) : void
+ getMortgageBalance ( ) : float
+ setMortgageBalance (m : float) : void
+ totalWeeklyNetPayments ( ) : float
+ find (findMortgageID : string) : Boolean
+ read (fileName : RandomAccessFile) : void
+ write (fileName : RandomAccessFile) : void
+ obtainNewData ( ) : void
+ performDeletion ( ) : void
+ print ( ) : void
+ <<static>> printAll ( ) : void

Detailed Design: MSG Foundation

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

Method `EstimateFundsForWeek::computeEstimatedFunds`


```
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

Figure 14.17

```
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
```

14.9 The Design Workflow

- Summary of the design workflow:
  - It is to refine the artifacts of the analysis workflow until the material is in a form that can be implemented by the programmers.

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

The Design Workflow (contd)

- 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)

- 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)

- 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)

- 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)

- 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

CASE Tools for Design (contd)

- 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

- Measures of design quality
  - Cohesion
  - Coupling
  - Fault statistics

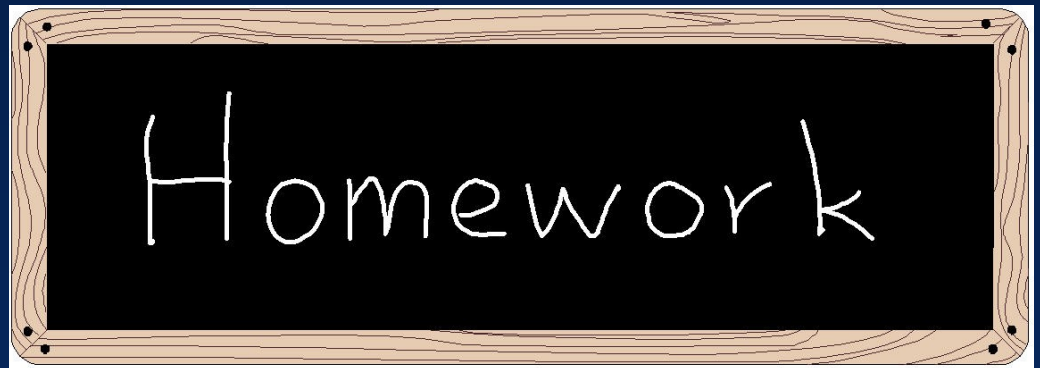
14.16 Challenges of the Design Phase

- 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

Exercise

- Design the library circulation system using object-oriented design:

Consider an automated library circulation system. Every book has a bar code, and every borrower has a card bearing a bar code. When a borrower wishes to check out a book, the librarian scans the bar codes on the book and the borrower's card, and enters C at the computer terminal. Similarly, when a book is returned, it is again scanned and the librarian enters R. Librarians can add books (+) to the library collection or remove them (−). Borrowers can go to a terminal and determine all the books in the library by a particular author (the borrower enters A= followed by the author's name), all the books with a specific title (T= followed by the title), or all the books in a particular subject area (S= followed by the subject area). Finally, if a borrower wants a book currently checked out, the librarian can place a hold on the book so that, when it is returned, it will be held for the borrower who requested it (H= followed by the number of the book).



Exercise

- Design the ATM system using object-oriented design:

Consider an automated teller machine (ATM). The user puts a card into a slot and enters a four-digit personal identification number (PIN). If the PIN is incorrect, the card is ejected. Otherwise, the user may perform the following operations on up to four different bank accounts:

(i) Deposit any amount. A receipt is printed showing the date, amount deposited, and account number.

(ii) Withdraw up to $200 in units of $20 (the account may not be overdrawn). In addition to the money, the user is given a receipt showing the date, amount withdrawn, account number, and account balance after the withdrawal.

(iii) Determine the account balance. This is displayed on the screen.

(iv) Transfer funds between two accounts. Again, the account from which the funds are transferred must not be overdrawn. The user is given a receipt showing the date, amount transferred, and the two account numbers.

(v) Quit. The card is ejected.