SE 323 - Software Construction Testing and Maintenance

MODULE ONE
SOFTWARE CONSTRUCTION

CHAPTER FOUR
IMPLEMENTING SOFTWARE CONSTRUCTION
PART TWO

GUIDELINES AND CONVENTIONS FOR GOOD SOFTWARE CONSTURCTION PRACTICES

Module 1
Software Construction

OUTLINE

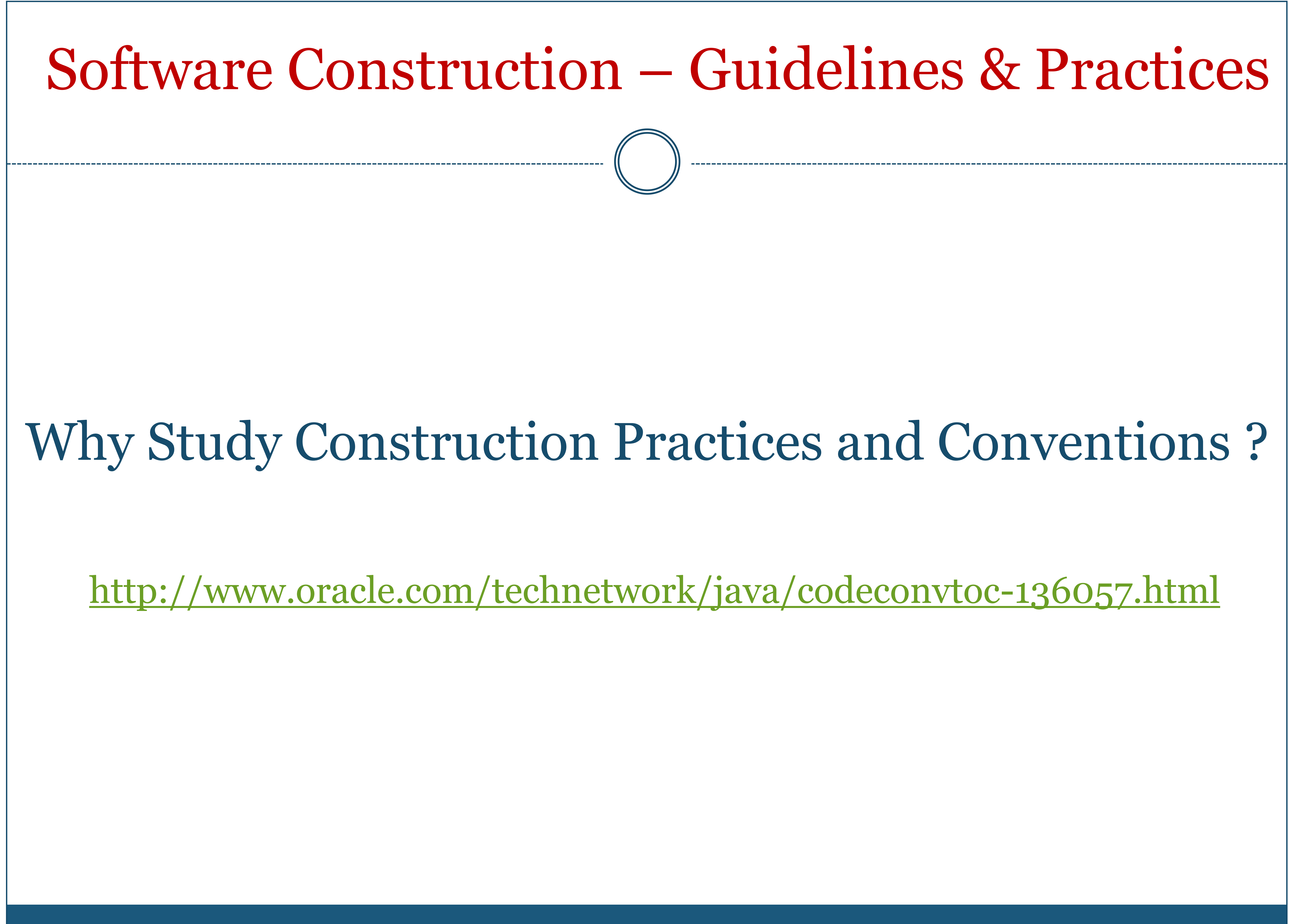
1. ~~Overview of Software Construction~~
2. ~~Planning/Preparing for Software Construction~~
3. *Doing Software Construction*
4. ~~Documenting Software Construction~~
5. ~~Measuring/Monitoring Software Construction~~

Construction Practices -Implementation

Module 1
Software Construction

OUTLINE

1. ~~Overview of Software Construction~~
2. ~~Planning/Preparing for Software Construction~~
3. *Doing Software Construction*
   - ~~3.1 Software Construction Process & Activities~~
   - 3.2 *Software Construction Guidelines & Practices*
4. ~~Documenting Software Construction~~
5. ~~Measuring/Monitoring Software Construction~~


Software Construction – Guidelines & Practices
Why Study Construction Practices and Conventions ?
http://www.oracle.com/technetwork/java/codeconvtoc-136057.html

3.2 Software Construction - Guidelines & Practices

SE Principles, guidelines and good practices for:

- 3.2.1 Class Design & Implementation
- 3.2.2 Routine Design & Implementation
- 3.2.3 Statement Design & Implementation
- 3.2.4 Variable Design & Implementation
- 3.2.5 Program Layout & Formatting style

3.2 Software Construction - Guidelines & Practices

SE Principles, guidelines and good practices for:

- 3.2.1 Class Design & Implementation
- ~~3.2.2 Routine Design & Implementation~~
- ~~3.2.3 Statement Design & Implementation~~
- ~~3.2.4 Variable Design & Implementation~~
- ~~3.2.5 Program Layout & Formatting style~~

Construction Practices – Classes

3.2 Software Construction -Guidelines & Practices

3.2.1 Class Design & Implementation

- Object Identification
- Object Modelling & Abstraction
- Encapsulation & Information Hiding
- Inheritance & Containment
- Naming Convention

Construction Practices - Classes

3.2.1 Construction Practices - Classes

- Object Identification
  - Domain Classes

    Corresponds to a requirements paragraph (SRS), e.g. *Student*, *School*
  - Design Classes

    Specified in the software design (SDD), e.g. *Registration*
  - Implementation Classes

    Too minor to specify in design, needed for implementation

Construction Practices - Classes

3.2.1 Construction Practices - Classes

Object Identification

Example:

Identifying objects (domain classes) from the requirements document of an ATM system

Nouns and noun phrases in the ATM requirements document			
bank	money / funds	account number	ATM
screen	PIN	user	keypad
bank database	customer	cash dispenser	balance inquiry
transaction	$20 bill / cash	withdrawal	account
deposit slot	deposit	balance	deposit envelope

Construction Practices - Classes

3.2.1 Construction Practices - Classes

- Object Modeling & Abstraction
  - Find Real-World Object
    - Identify the objects and their attributes
    - Determine what can be done to each object
    - Determine what each object is allowed to do to other objects
    - Determine the parts of each object that will be visible to other objects - public vs. private
    - Define each object's interface, based on the above

Process Iteration: Top level vs. Class Level

Construction Practices - Classes

3.2.1 Construction Practices - Classes

- Object Modeling & Abstraction - Example

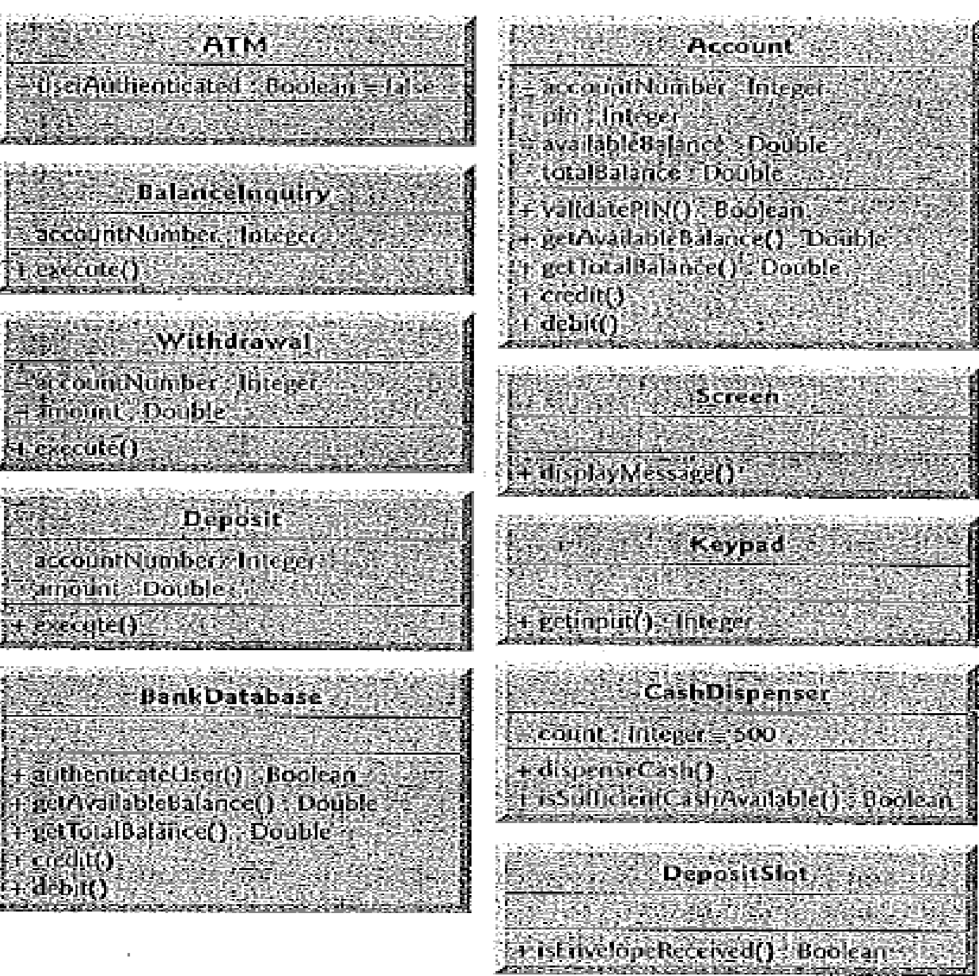


Fig. 13.1 | Class diagram with visibility markers.

Construction Practices - Classes

3.2.1 Construction Practices - Classes

- Object Modeling & Abstraction - Example

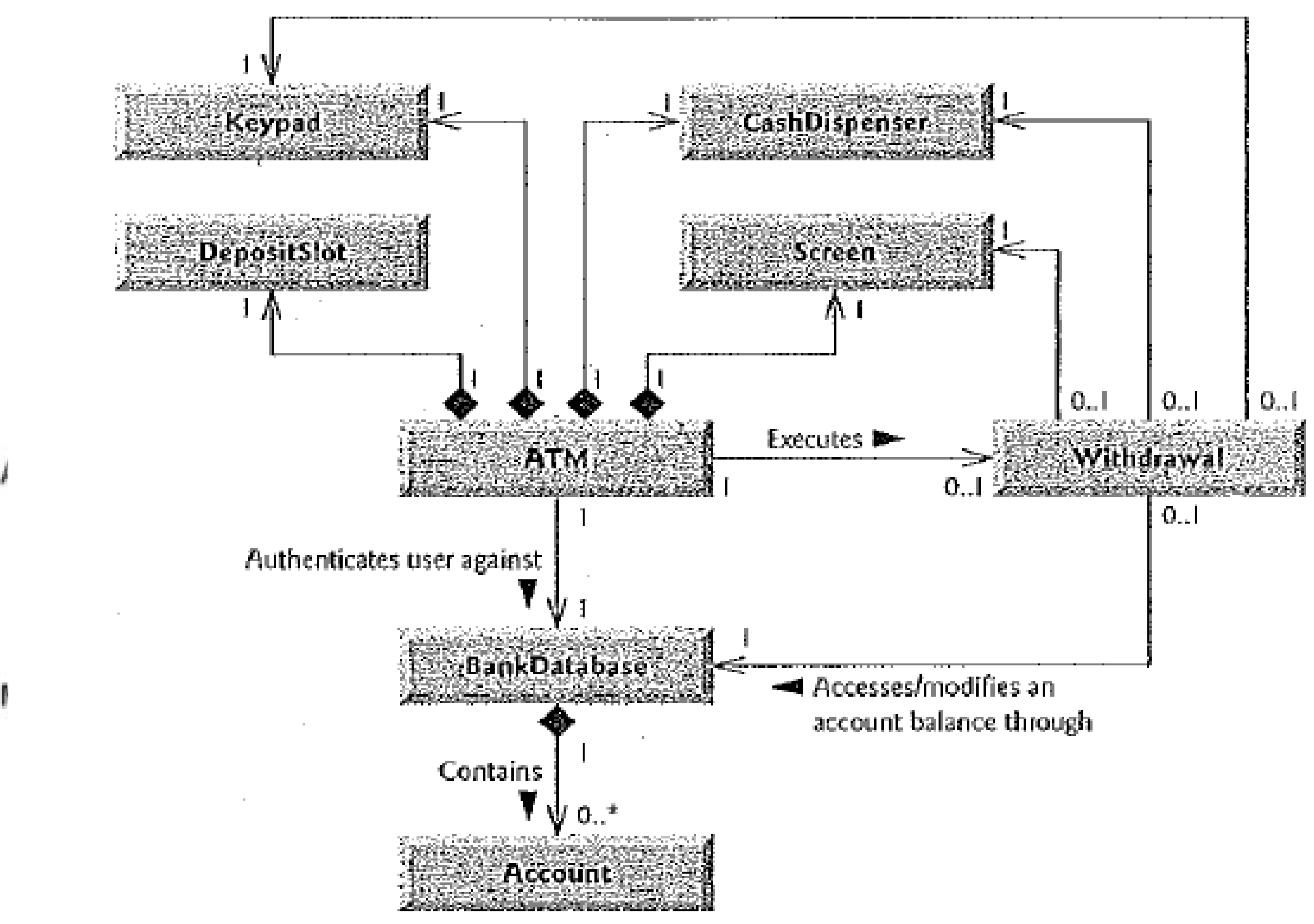
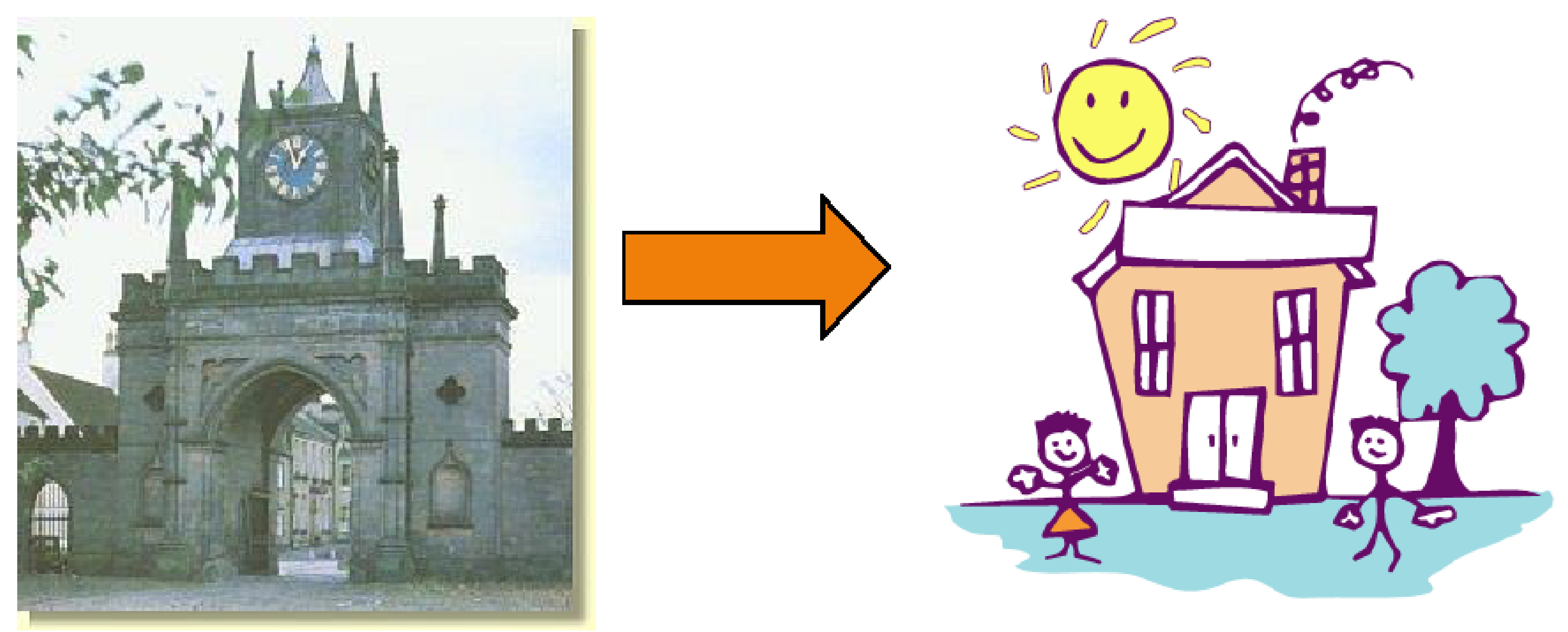


Fig. 13.2 | Class diagram with navigability arrows.

Construction Practices - Classes

- **Object Modeling & Abstraction**
  - Form Consistent Abstraction



Construction Practices - Classes

Object Modeling & Abstraction

Form Consistent Abstraction

- Engage with a concept/an aggregate
  - ignore some of its details
- Handle different details at different levels
  - e.g. routine-interface level, class-interface level, package-interface level
- Base class abstraction
  - focus on common attributes of a set of derived classes
  - ignore details of specific classes while working on base class
- Class interface abstraction
  - focus on the interface
  - ignore the internal workings of the class

Construction Practices - Classes

Object Modeling & Abstraction

Form Consistent Abstraction

Class Interface for Good Abstraction - Guidelines

- Provides an abstraction of the implementation that's hidden behind the interface.
- Offer a group of routines that clearly belong together.
- Understand what abstraction the class is implementing
- Present a consistent level of abstraction in the class interface

Construction Practices - Classes

Object Modeling & Abstraction

Class Interface for Good Abstraction - Example

```
class Program {
public:    …
    void InitializeCommandStack();
    void PushCommand(Command command)
    Command popCommand();
    void shutdownCommand Stack();
    void InitializeReportFormatting();
    void FormatReport(Report report);
    void Print Report(Report report);
    void InitializeGlobalData();
    void ShudownGlobalData();
    …
private:
    …
};
```

→

```
class Program {
public:    …
    void InitializeUserInterface();
    void ShutdownUserInterface();
    void InitializeReports();
    void shutdownReports();
…
private:
    …
};
```

→ *Real-world Entities NOT Low-level Implementation!*

Construction Practices - Classes

Object Modeling & Abstraction

Class Interface for Good Abstraction - Example

```
class EmployeeCencus: public ListContainer {
public:    …
    void AddEmployee (Employee employee);
    void RemoveEmployee (Employee employee);

    Employee NextItemInList();
    Employee FirstItem();
    Employee LastItem();
    …
private:

    …
}
```

"employee" level

"list" level

→ *Inconsistent level of abstraction in the class interface!*

Construction Practices - Classes

Object Modeling & Abstraction

Class Interface for Good Abstraction - Example

```
class EmployeeCencus {
public:    …
    void AddEmployee (Employee employee);
    void RemoveEmployee (Employee employee);
    Employee NextEmployee();
    Employee First Employee();
    Employee LastEmployee();
    …
private:
    ListContainer m_EmployeeList;
    …
}
```

Now abstraction of all routines is at "employee" level

"List" is hidden

→ *Now a consistent level of Abstraction in the class interface!*

SC Planning – SDLC - Design

Design Principle & Good Practice - Cohesion - Example

```
public class CashRegister
{
   public static final double NICKEL_VALUE = 0.05;
   public static final double DIME_VALUE = 0.1;
   public static final double QUARTER_VALUE = 0.25;
   . . .
   public void enterPayment(int dollars, int quarters,
         int dimes, int nickels, int pennies)
   . . .
}
```

Bad Example
Low cohesion class
&
Does not represent
a single abstract concept

```
public class Coin
{
   . . .
   public Coin(double aValue, String aName) { . . . }
   public double getValue() { . . . }
   . . .
}
public class CashRegister
{
   . . .
   public void enterPayment(int coinCount, Coin coinType) { . . . }
   . . .
}
```

Good Example
High cohesion class
&
Consistent level
of abstraction

Construction -Implementation

3.2 Software Construction -Guidelines & Practices

3.2.1 Class Design & Implementation

- ~~Object Identification~~
- ~~Object Modelling & Abstraction~~
- Encapsulation & Information Hiding
- ~~Inheritance & Containment~~
- ~~Naming Convention~~

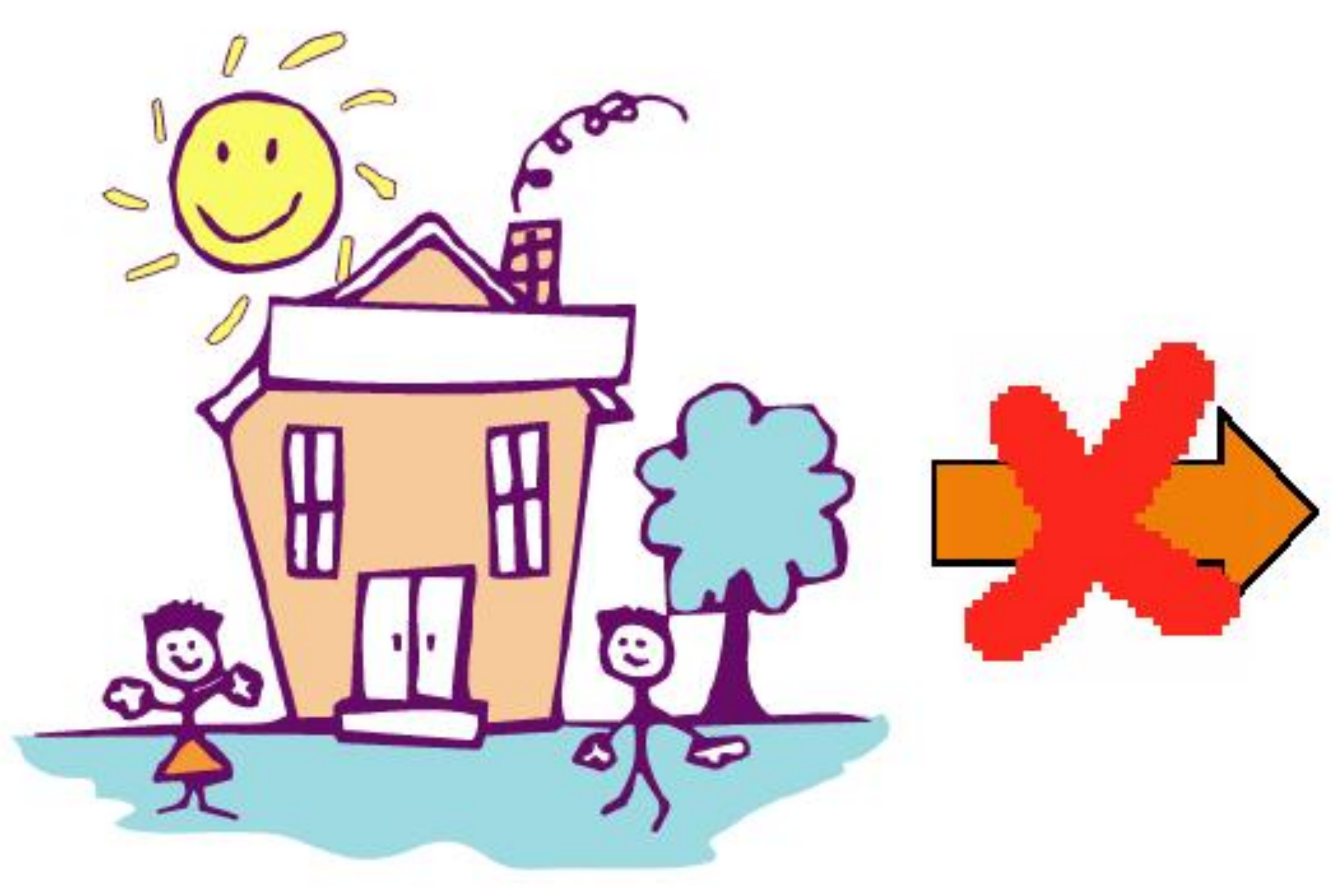
3.2.1 Construction Practices - Classes

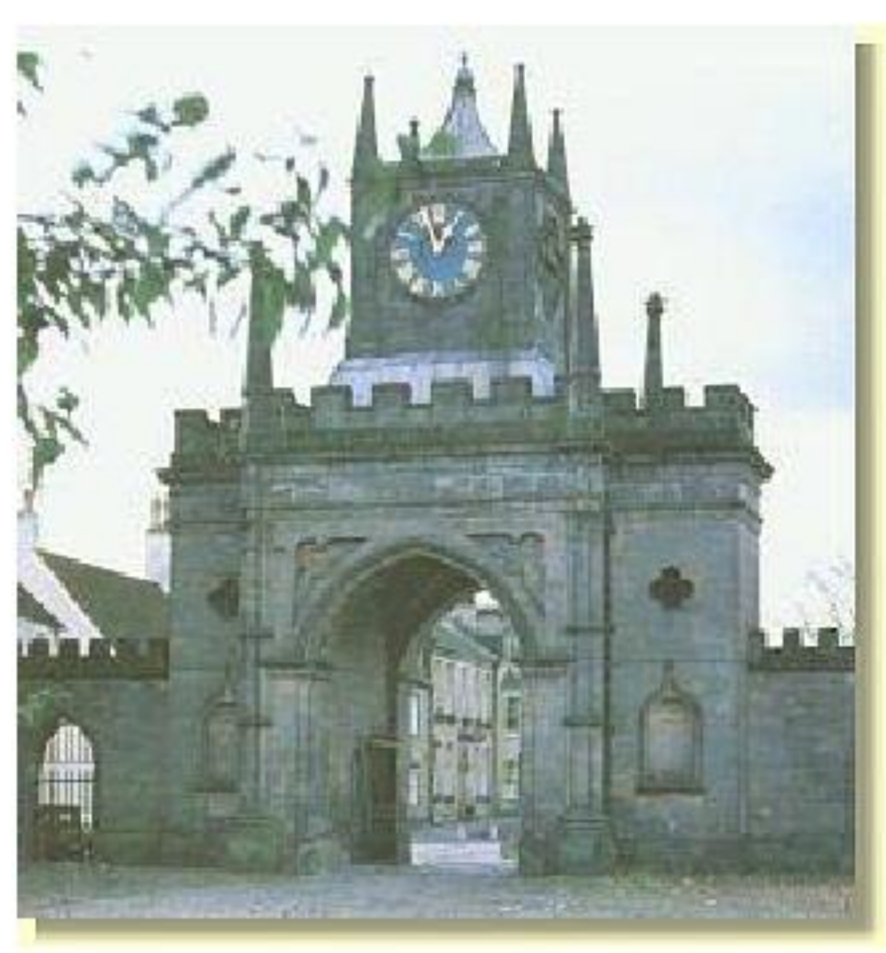
❑ Encapsulation & Information Hiding

- Next level of abstraction
- No visibility of an object at any other level of details
- Managing complexity by forbidding others to look at the complexity

Construction Practices - Classes

- **Encapsulation & Information Hiding**





Construction Practices - Classes

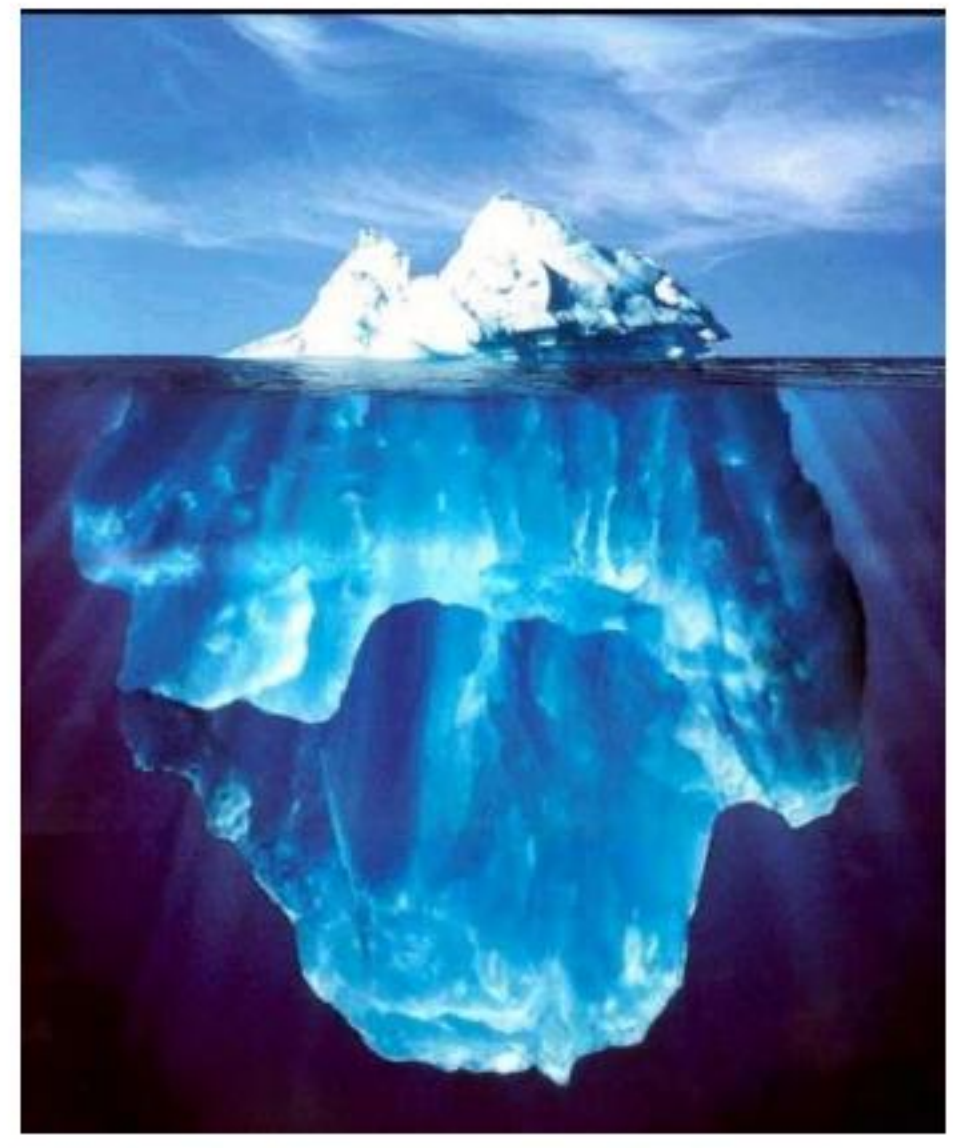
Encapsulation & Information Hiding

- Minimize accessibility of classes and members
- Don’t expose member data in public
- Avoid putting private implementation details into a class’s interface
- Don’t make assumptions about the class’ users
- Don’t put a routine into the public interface just because it uses only public routines
- Be wary of semantic violations of encapsulation
- Watch for coupling that’s too tight

Construction Practices - Classes

- **Encapsulation & Information Hiding**

  Hiding Secret



Construction Practices - Classes

Encapsulation & Information Hiding

- Modularity – Complexity hiding
  - Structural Design: Black boxes
  - Object Oriented Design: Abstraction & Encapsulation
- Class design for information hiding
  - Hiding secret attributes
  - Hiding internal routines
  - Interface should reveal as little as possible
  - Visibility: within vs. outside the class
- Always asking "What should I hide?," "What does this class need to hide?"

Construction Practices - Classes

Encapsulation & Information Hiding

- Information hiding at all levels
  - Data type level
    - Use ***named constants*** rather than ***literals***
    - User ***defined data type*** rather than ***generic data type***
    - Use ***enumerated type*** rather than ***boolean type***
    - ***Use access routines*** rather than ***global variables***
  - Object Level
    - Class design
    - Routine design
    - Subsystem design

Construction Practices – Classes - Example

❑ Encapsulation & Information Hiding

Use access routines, set() and get(), rather than global variables

ch03/account/BankAccount.java

```
 1  /**
 2     A bank account has a balance that can be changed by
 3     deposits and withdrawals.
 4  */
 5  public class BankAccount
 6  {
 7     private double balance;
 8
 9     /**
10        Constructs a bank account with a zero balance.
11     */
12     public BankAccount()
13     {
14        balance = 0;
15     }
16
17     /**
18        Constructs a bank account with a given balance.
19        @param initialBalance the initial balance
20     */
21     public BankAccount(double initialBalance)
22     {
23        balance = initialBalance;
24     }
25
26     /**
27        Deposits money into the bank account.
28        @param amount the amount to deposit
29     */
30     public void deposit(double amount)
31     {
32        balance = balance + amount;
33     }
34
35     /**
36        Withdraws money from the bank account.
37        @param amount the amount to withdraw
38     */
39     public void withdraw(double amount)
40     {
41        balance = balance - amount;
42     }
43
44     /**
45        Gets the current balance of the bank account.
46        @return the current balance
47     */
48     public double getBalance()
49     {
50        return balance;
51     }
52  }
```

Construction Practices - Classes

Encapsulation & Information Hiding

- Value of Information Hiding
  - Easier to modify*
  - Minimal impacts by changes & errors
  - Useful for designing class public interface
- Barriers to information hiding
  - Excessive distribution of information
  - Circular dependencies
  - Class data mistaken for global data
  - Perceived performance penalties

---

* According to statistics, large programs that use information hiding are 4 times easier to modify than programs that don't

Construction Practices - Classes

3.2 Software Construction -Guidelines & Practices

3.2.1 Class Design & Implementation

- ~~Object Identification~~
- ~~Object Modelling & Abstraction~~
- ~~Encapsulation & Information Hiding~~
- Inheritance & Containment
- ~~Coupling & Cohesion~~
- ~~Naming Convention~~

Construction Practices - Classes

Inheritance & Containment

Inheritance – Principles & good practices

- Organizing objects with common properties
  - General types vs. specific types
  - Similarities vs. differences
- Objects which are like other objects, except for a few differences
  - Full-time & part-time employee as specific types of an employee object
- Benefits
  - General routines to handle anything that depends on the objects' general properties
  - Specific routines to handle specific operations on specific types of objects
  - Polymorphism, e.g. *open(), close ()*

Construction Practices - Classes

Inheritance & Containment

<u>Inheritance – Principles & good practices</u>

- Implement "*is-a*" relationship through public inheritance
- Adhere to the Liskov Substitution Principle (LSP)
  - Shouldn't inherit from base class unless the derived class truly "*is a*" more specific version of the base class, e.g.
    - Employee → Person
    - Truck → Car
  - Subclasses must be usable through the base class interface without the user's knowing the difference

Construction Practices - Classes

❑ Inheritance & Containment

Inheritance – Principles & good practices

- Don't "override" a non-overridable member function (i.e. reuse names of base-class routines)
- Move common interfaces, data and behavior as high as possible in the inheritance
- Be suspicious of classes with
  - one instance
  - only one derived class
  - Overriding derived routines which do nothing inside

Construction Practices - Classes

Inheritance & Containment

Inheritance – Principles & good practices

- Avoid deep inheritance tree
  - Maximum of 6 (7± 2) levels
- Prefer polymorphism to extensive type checking
- Avoid protected members in the base classes

  "Inheritance breaks encapsulation!"
  - Make all data private, not protected
  - Provide protected accessor functions

Construction Practices - Classes

Inheritance & Containment

Inheritance – Principles & good practices - Example

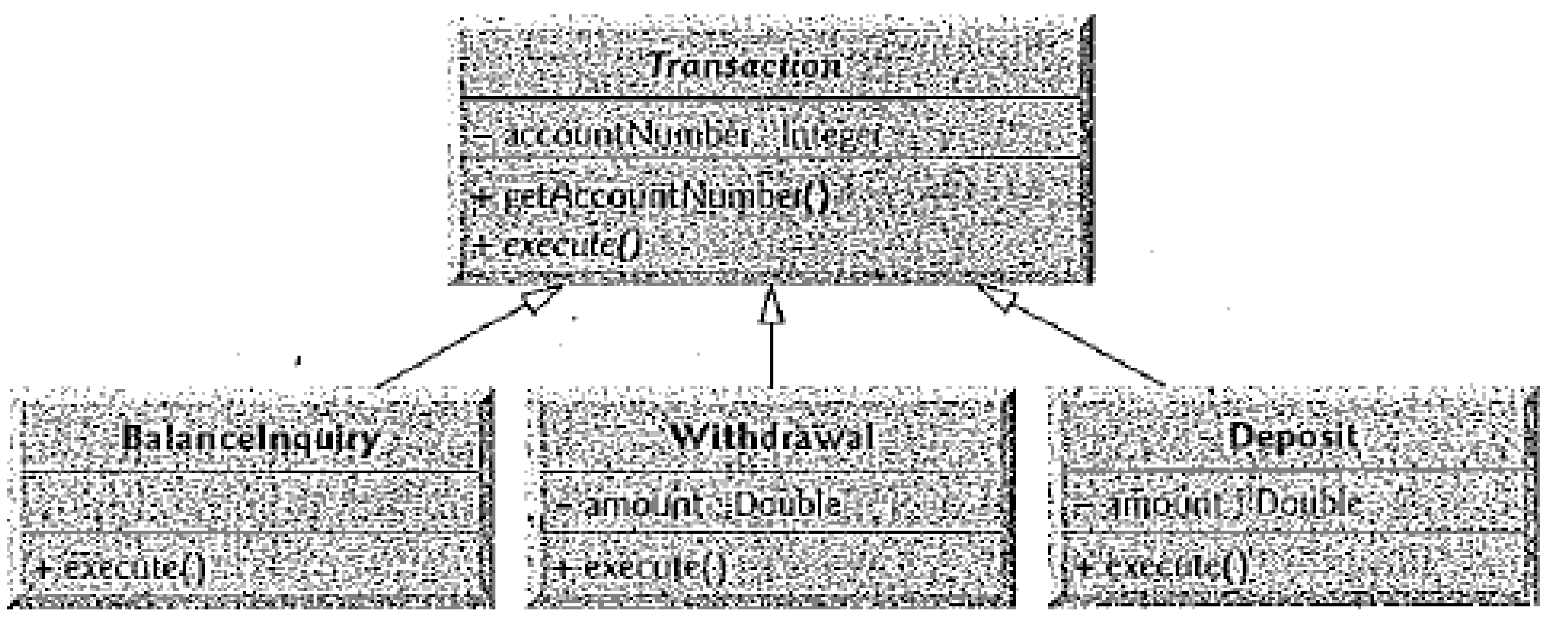


Fig. 13.8 | Class diagram modeling generalization of superclass `Transaction` and subclasses `BalanceInquiry`, `Withdrawal` and `Deposit`. Abstract class names (e.g., `Transaction`) and method names (e.g., `execute` in class `Transaction`) appear in italics.

Construction Practices - Classes

Inheritance & Containment

Inheritance – Principles & good practices - Example

```
// Withdrawal.java
// Represents a withdrawal ATM transaction

public class Withdrawal extends Transaction
{
   private int amount; // amount to withdraw
   private Keypad keypad; // reference to keypad
   private CashDispenser cashDispenser; // reference to cash dispenser

   // constant corresponding to menu option to cancel
   private final static int CANCELED = 6;

   // Withdrawal constructor
   public Withdrawal( int userAccountNumber, Screen atmScreen,
      BankDatabase atmBankDatabase, Keypad atmKeypad,
      CashDispenser atmCashDispenser )
   {
      // initialize superclass variables
      super( userAccountNumber, atmScreen, atmBankDatabase );

      // initialize references to keypad and cash dispenser
      keypad = atmKeypad;
      cashDispenser = atmCashDispenser;
   } // end Withdrawal constructor
```

Construction Practices - Classes

Inheritance & Containment

Containment – Principles & Good Practices

- Implement "*has-a*" relationship through containment
- Implement "*has-a*" relationship through private inheritance as a last resort
- Be critical of classes that contain more than about seven (7± 2) data members
- Also referred to as "*Aggregation*"

Construction Practices - Classes

Inheritance & Containment

Containment – Principles & Good Practices-Example

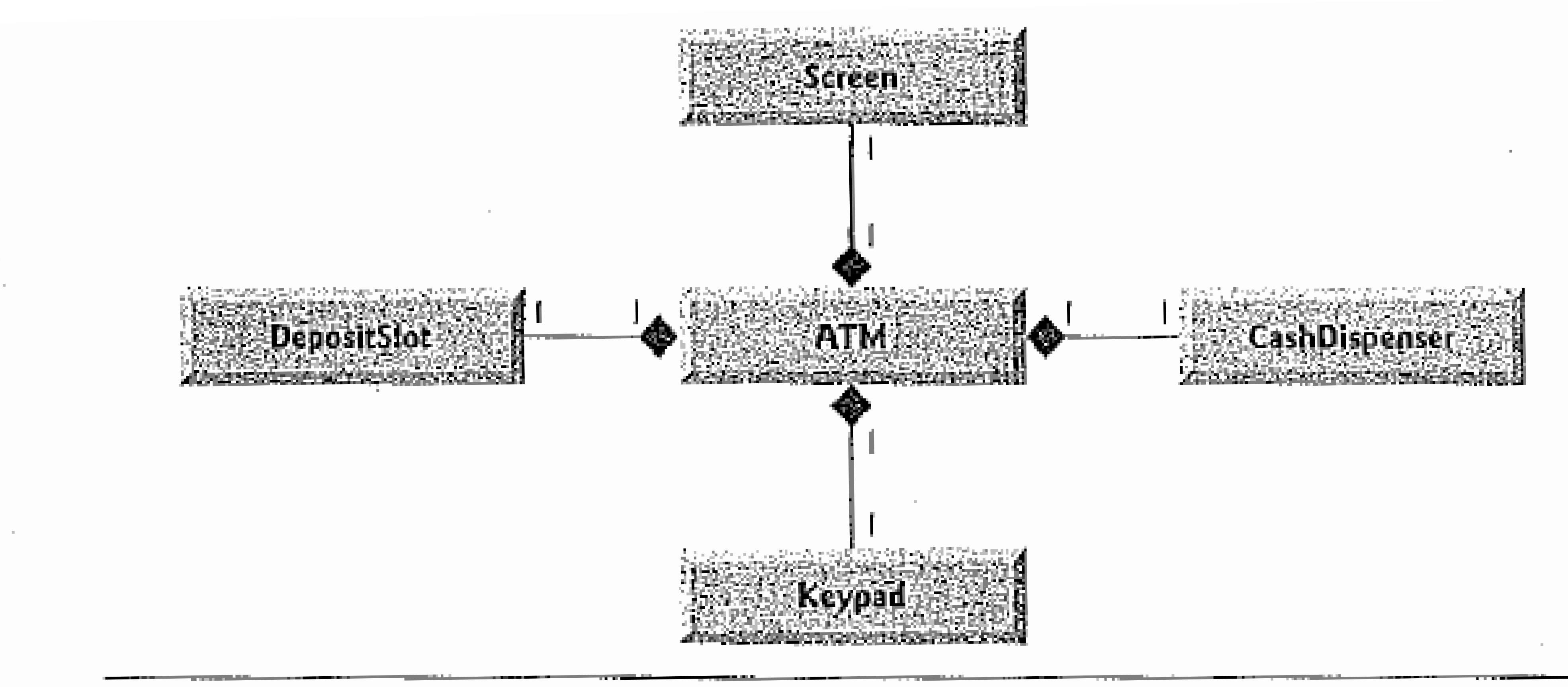


Fig. 12.9 | Class diagram showing composition relationships.

Construction Practices - Classes

Inheritance & Containment

Containment – Principles & Good Practices-Example

```
1   // ATM.java
2   // Represents an automated teller machine
3
4   public class ATM
5   {
6      private boolean userAuthenticated; // whether user is authenticated
7      private int currentAccountNumber; // current user's account number
8      private Screen screen; // ATM's screen
9      private Keypad keypad; // ATM's keypad
10     private CashDispenser cashDispenser; // ATM's cash dispenser
11     private DepositSlot depositSlot; // ATM's deposit slot
12     private BankDatabase bankDatabase; // account information database
13
14     // constants corresponding to main menu options
15     private static final int BALANCE_INQUIRY = 1;
16     private static final int WITHDRAWAL = 2;
17     private static final int DEPOSIT = 3;
18     private static final int EXIT = 4;
19
20     // no-argument ATM constructor initializes instance variables
21     public ATM()
22     {
23        userAuthenticated = false; // user is not authenticated to start
24        currentAccountNumber = 0; // no current account number to start
25        screen = new Screen(); // create screen
```

Fig. 13.13 | Class ATM represents the ATM. (Part 1 of 4.)

Inheritance & Containment

Inheritance VS. Containment

When to use containment ?

- “Has-a” Relationship

When to use inheritance ?

- “Is-a” Relationship

Construction Practices - Classes

❑ Inheritance & Containment

<u>Inheritance VS. Containment</u> - Example

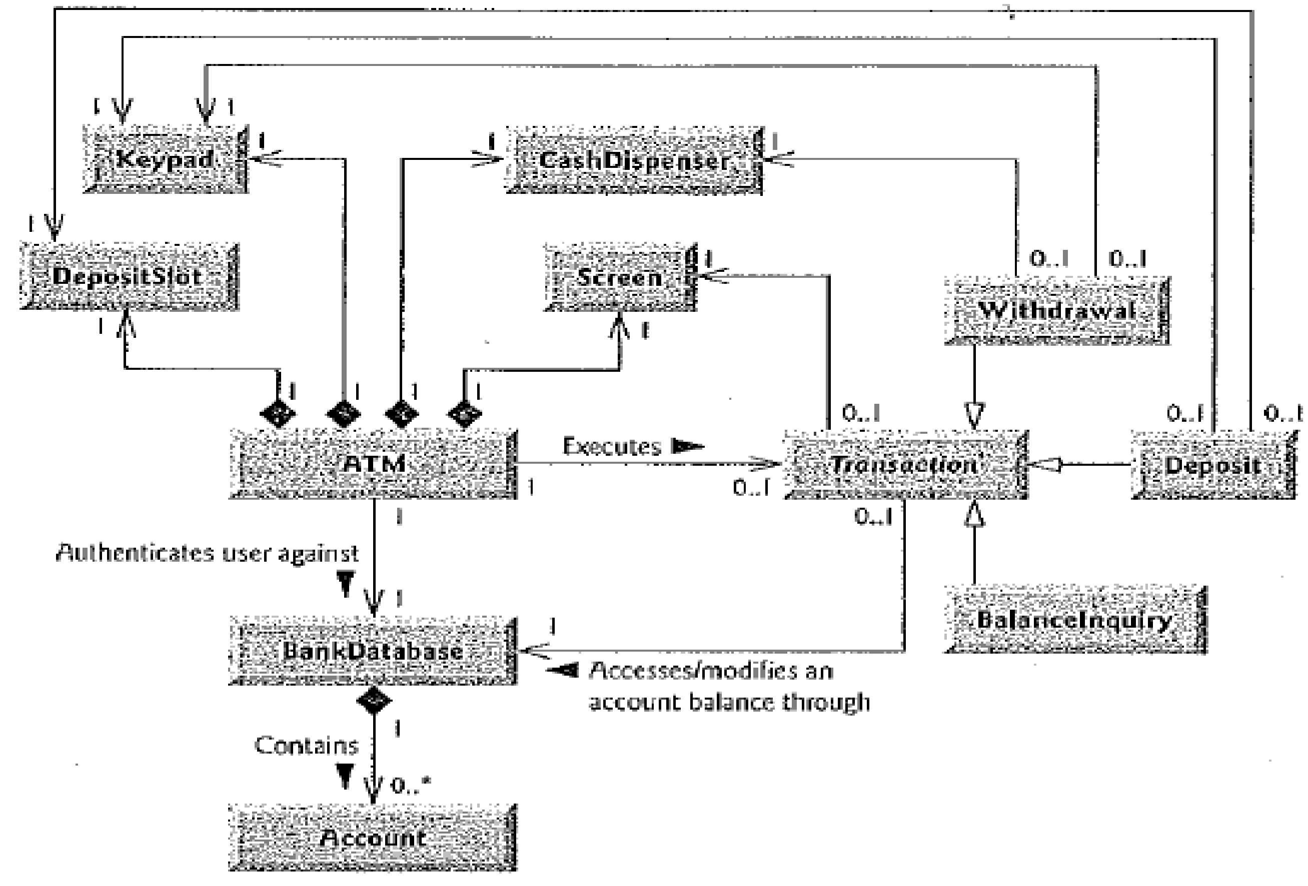


Fig. 13.9 | Class diagram of the ATM system (incorporating inheritance). The abstract class name Transaction appears in italics.

Construction Practices - Classes

Inheritance & Containment

Inheritance VS. Containment

- When to use inheritance ?
  - Share common behavior but not data
  - Share common data and behavior
  - Want base class to control interface
- When to use containment ?
  - Share common data but not behavior, create common object that those classes can contain

Construction Practices - Classes

3.2 Software Construction -Guidelines & Practices

3.2.1 Class Design & Implementation

- ~~Object Identification~~
- ~~Object Modelling & Abstraction~~
- ~~Encapsulation & Information Hiding~~
- ~~Inheritance & Containment~~
- Coupling & Cohesion
- ~~Naming Convention~~

3.2.1 Construction Practices - Classes

Coupling & Cohesion

<u>Coupling</u> the external relationship (dependency) among classes

- *Good practice → Loose Coupling*

<u>Cohesion</u> the internal relationship (relatedness) among all parts within a single class

- *Good practice → High Cohesion*

*** For more details, see Chapter 2, slides 60-66***

Construction Practices - Classes

3.2 Software Construction -Guidelines & Practices

3.2.1 Class Design & Implementation

- ~~Object Identification~~
- ~~Object Modelling & Abstraction~~
- ~~Encapsulation & Information Hiding~~
- ~~Inheritance & Containment~~
- ~~Class Dependency & Integrity~~
- Naming Convention

3.2.1 Construction Practices - Classes

- Naming Convention

The class name should:

- Be expressive and meaningful
- Not include vague, ambiguous terms
- Imply a coherent object modelling entity
- Convey the abstraction concept the class represents
- Normally made up of concatenated words in lowercase letters except for the first letter of each word which is capitalised. In some cases the name begins with some standard letters such as 'C' for classes, e.g. CCustomer

Construction Practices - Classes

REFERENCE

Textbook: *"Code Complete"*, second edition

Chapter 6: *"Working Classes"*

Software Construction – Guidelines & Practices

3.2 Software Construction - Guidelines & Practices

SE Principles, guidelines and good practices for:

- ~~3.2.1 Class Design & Implementation~~
- 3.2.2 Routine Design & Implementation
- ~~3.2.3 Statement Design & Implementation~~
- ~~3.2.4 Variable Design & Implementation~~
- ~~3.2.5 Program Layout & Formatting style~~

Construction Practices - Routines

3.2 Software Construction - Guidelines & Practices

3.2.2 Routine* Design & Implementation

- Low-Quality Routines
- Reasons to Create a Routine
- Routine Cohesion
- Good Routine Names
- Size of Routine
- Routine Parameters
- Use of Functions

Definition

*_Routine_: A smallest software module called for a single purpose, _aka method, procedure_

Construction Practices – Routines - Example

C++ Example Of a Low-Quality Routine

```
1  void HandleStuff( CORP_DATA & inputRec, int crntQtr, EMP_DATA empRec, double
2     & estimRevenue, double ytdRevenue, int screenX, int screenY, COLOR_TYPE &
3     newColor, COLOR_TYPE & prevColor, StatusType & status, int expenseType )
4  {
5  int i;
6  for ( i = 0; i < 100; i++ ) {
7     inputRec.revenue[i] = 0;
8     inputRec.expense[i] = corpExpense[ crntQtr ][ i ];
9     }
10 UpdateCorpDatabase( empRec );
11 estimRevenue = ytdRevenue * 4.0 / (double) crntQtr;
12 newColor = prevColor;
13 status = SUCCESS;
14 if ( expenseType == 1 ) {
15      for ( i = 0; i < 12; i++ )
16            profit[i] = revenue[i] - expense.type1[i];
17      }
18 else if ( expenseType == 2 )  {
19           profit[i] = revenue[i] - expense.type2[i];
20           }
21 else if ( expenseType == 3 )
22           profit[i] = revenue[i] - expense.type3[i];
23           }
```

Routine – Low-Quality Routine

- **Low-Quality Routines* - features:**
  - Has bad name, e.g. *HandleStuff()*
  - Is not documented
  - Has a bad layout
  - Change value of input variables
  - Read and write global variables
  - Does not have a single purpose

---

* *For coding example, see "Code Complete" page 162*

Routine – Low-Quality Routine

- **Low-Quality Routines – features (cont.)**
  - Does not defend itself against bad data
    - e.g. divide-by-zero
  - Use several magic numbers
    - e.g. 100, 4.0 etc.
  - Contains unused parameters
  - Incorrect type of parameters, e.g. pass-by-reference
  - Has too many parameters
    - upper limit is about 7
  - Parameters are poorly ordered and not documented

Routine – Reasons

Reasons to Create a Routine

- Reduce complexity
- Introduce an intermediate, understandable abstraction
- Avoid duplicate code
- Support subclassing
- Hide sequences
- Hide pointer operations
- Improve portability
- Simplify complicated boolean tests
- Improve performance

Routine - Cohesion

Routine Cohesion*

- Study on cohesion
  - 50 % of routines with high cohesion were fault free
  - 18 % of routines with low cohesion were fault free
- Study on coupling/cohesion ratio
  - Routines with highest coupling/cohesion ratios had 7 times as many errors as those with the lowest coupling/cohesion ratios
  - Also 20 times as costly to fix

---

*Definition: *"Cohesion"*

- *How closely the operations within a single routine are related*
- *For more detailed explanation, see Chapter 2, pages 82 - 89*

Routine - Cohesion

- **Routine Cohesion (cont.)**

  Types of acceptable cohesion within a routine:

  - Functional Cohesion
    - Routine with one and only one operation
    - Strongest and best kind of cohesion
  - Sequential Cohesion
    - Routine with operations performed in a specific order
  - Communicational Cohesion
    - Routine with operations using the same data
  - Temporal Cohesion
    - Routine with operations all done at the same time

Routine - Cohesion

- **Routine Cohesion (cont.)**

  Types of Unacceptable Cohesion* within a Routine:

  - Procedural Cohesion

    Operations are done in a specific order
  - Logical Cohesion

    One from several operations selected by a control flag
  - Coincident Cohesion

    No cohesion

---

* Routines with such cohesion result in poorly organized, hard to debug, hard to modify code

Routines –Routine Names

Good Routine Names - Exercise

Thinking of a routine name which ....

- Create a monthly report
- Handle a mouse-click event
- Indicate the number of students in a classroom
- Check whether a particular student has registered for the class

Routines –Routine Names

- **Good Routine Names – Guidelines**
  - Describe everything the routine does
    - *computeReportTotal()*
    - *computeReportTotalsAndOpenOutputFile()*
  - Avoid meaningless and vague verbs
    - *handleOutput(), performService()*
    - *formatAndPrintOutput()*
  - Don't differrentiate routine names solely by numbers
    - *outputUser, outputUser1, outputUser2*
  - Use a description of the return value
    - *cos(), CustomerId.next(), Printer.isReady()*

Routines –Routine Names

Good Routine Names – Guidelines (con.t)

- Make names of routines as long as necessary
  - Variable name 9-15 characters
  - Routine name is slightly longer
- Use a strong verb followed by an object
  - PrintDocment(), CheckOrderInfo()
  - For OO: Document.print(), OrderInfo.check()
- Use opposites precisely
  - add & remove, begin & end, first & last
- Naming Convention
  - Normally made up of concatenated words in lowercase letters with all but the first words begins with a capital letter.

Routines – Routine Size

Size of Routine

- Appropriate size of routine can ....
  - Reduce error
  - Reduce cost
  - Reduce change
  - Easy to understand
- From many researches:
  - Maximum routine size is ___200___ lines of code

Routines – Routine Parameters

Routine Parameters – Guidelines

- Put parameters in input-modify-output order
  - Maybe language specific
- If several routines use similar parameters, put the similar parameters in a consistence order
- Use all parameters passed to a routine
- Put status or error variables last

Routines – Routine Parameters

Routine Parameters – Guidelines (cont.)

- Don't use routine parameters as working variables
- Limit the number of a routine's parameter to about seven
- Document interface assumptions about parameters
- Make sure actual parameters match formal parameters

Routines – Functions

Use of Functions

Return Value of a Function - Guidelines

- Check all possible return paths
- Don't return references or pointers to local data

Definition:

- A function is a routine that returns some value
- A procedure is routine which does not return any value

Construction Practices - Routines

Reference

Textbook: *"Code Complete"*, 2nd edition

- Chapter 7: *High Quality Routines*

3.2 Software Construction - Guidelines & Practices

SE Principles, guidelines and good practices for:

- ~~3.2.1 Class Design & Implementation~~
- ~~3.2.2 Routine Design & Implementation~~
- 3.2.3 Statement Design & Implementation
- ~~3.2.4 Variable Design & Implementation~~
- ~~3.2.5 Program Layout & Formatting style~~

Construction Practices - Statements

3.2 Software Construction - Guidelines & Practices

3.2.3 Statement Design & Implementation

- Sequential Statements
- Conditional Statements
- Iterative Statements

Statements - Sequential

Sequential Statements

- Statements which are executed in sequential order, line-by-line from top to bottom, as they are laid out in the program
- Two types of sequential statements
  - Statements that must be in a specific order
  - Statements whose order doesn't matter

Statements - Sequential

- **Sequential Statements (cont.)**

  Statements that must be in a specific order – Guidelines:

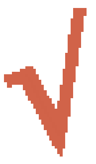
  - Organize code so that dependencies are obvious
  - Name routines so that dependencies are obvious
  - Use routine parameters to make dependencies obvious
  - Document unclear dependencies with comments
  - Check for dependencies with assertions or error-handling code

Statements - Sequential

Sequential Statements (cont.)

Statements that must be in a specific order – Examples:

```
Data = ReadData ();
results = CalculateResultsFromData (data);
PrintResult (results)
```



```
InitializeExpenseData();
ComputeMarketingExpense();
ComputeSalesExpense();
computeTravelExpense();
ComputePersonnelExpense();
DisplayExpenseSummary();
```

X →

```
InitializeExpenseData(expenseData);
ComputeMarketingExpense(expenseData);
ComputeSalesExpense(expenseData);
computeTravelExpense(expenseData);
ComputePersonnelExpense(expenseData);
DisplayExpenseSummary(expenseData);
```

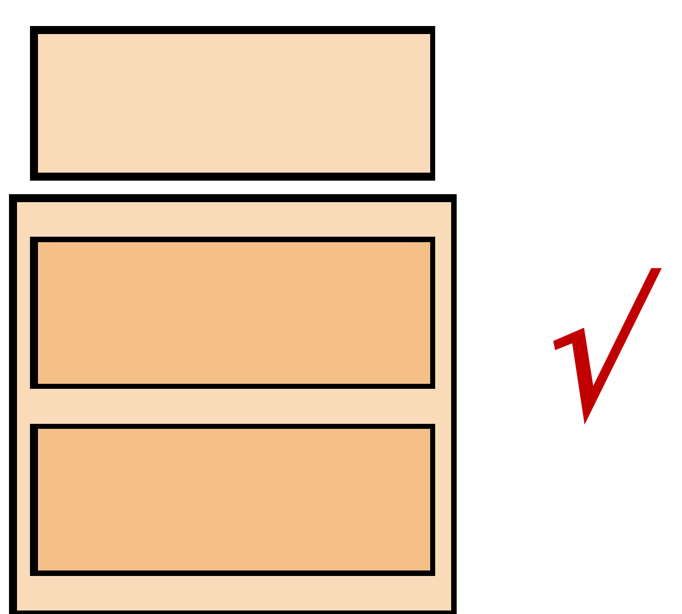
√

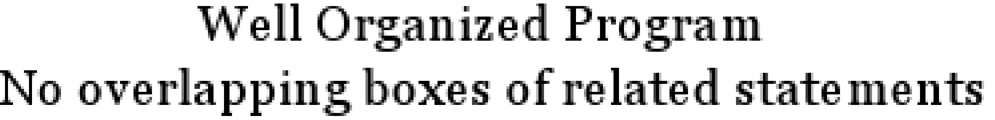
Statements - Sequential

Sequential Statements (cont.)

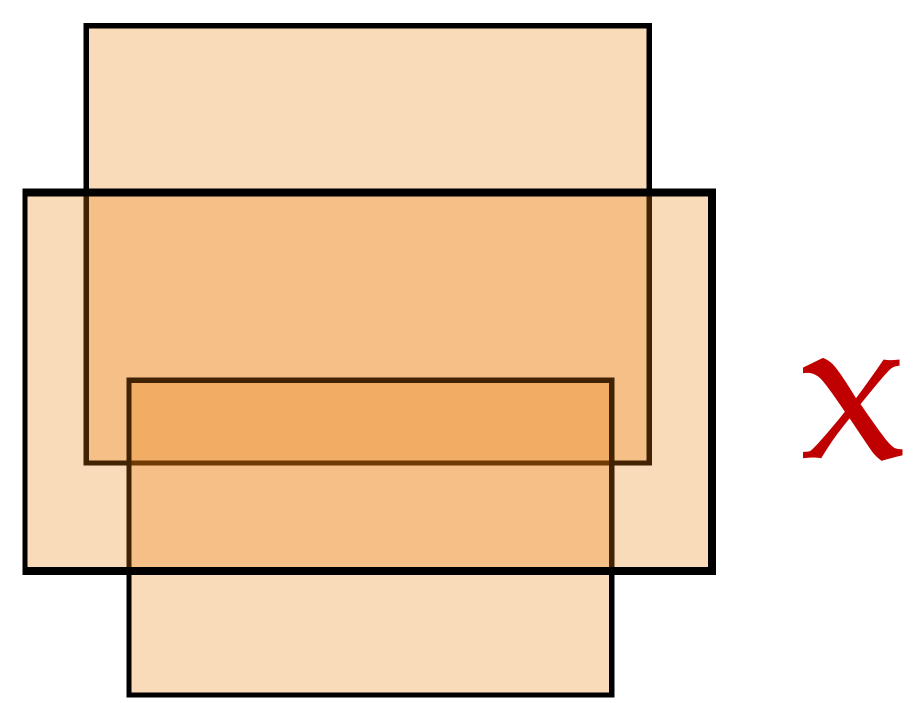
Statements whose order doesn't matter – Guidelines:

- Principle of Proximity: Keep related actions together
  - Make code read from top to bottom
  - Group related statements





Well Organized Program
No overlapping boxes of related statements



Poorly Organized Program
Overlapping boxes of related statements

Statements - Conditional

- **Conditional Statements**
  - IF Statements
  - CASE Statements

Statements - Conditional

IF Statements - *Guidelines for "if….then"*

- Write the normal path thru the code first, then write the unusual cases
- Check the endpoint conditions, i.e when the loop index equals to the min and max limits
  - Use '<=' and '>=' instead of '<' and '>' (more error prone)
- Put the normal case after the IF not ELSE
- Follow the IF with a meaningful statement
  - Use a negative predicate instead of a null statement
- Check for reversal of the IF and ELSE clauses

Statements - Conditional

- **IF Statements - *Guidelines for "if....then....else"***
  - Simplify complicated test with boolean function calls
  - Put the most common cases first
  - Make sure that all cases are covered
  - Avoid too many levels of nesting (good practice <=3)
  - Replace "*if-then-else*" chains with other constructs if supported by the language

    e.g. a "*case*" statement

Statements - Conditional

CASE Statements - Guidelines

- Use most effective ordering of cases
  - Alphabetical or numerical order
  - Normal case first
  - Most frequency first
- Keep the actions of each case short and simple
- Use the "*default*" clause only to detect legitimate defaults or errors

Statements - Conditional

- **CASE Statements – Guidelines (cont.)**
  - Avoid dropping through the end of a *case* statement
    - In C++ and Java, no automatic break out of each case
    - Must code the break out at the end of each case explicitly
  - If such dropping through is intentional…
    - Always comment at the spot and explain why

Statements - Iterative

Iterative Statements

Type of Loop:

- Counted loop

  Looping for a specific number of times

- Continuously evaluated loop

  Do not know how many time will loop execute

- Endless loop

- Iterative loop

  Perform its action once for each element in a container class

Statements - Iterative

- **Iterative Statements (cont.)**

  Guidelines for Entering the Loop

  - Enter the loop from one location only
  - Put initialization code directly before the loop
  - Use "*while(true)*" for infinite loops

Statements - Iterative

Iterative Statements (cont.)

Guidelines for Processing the Middle of the Loop

- Use { } to enclose the statements in a loop
- Avoid empty loops
- Keep loop-housekeeping chores* at the beginning or at the end of the loop
- Make each loop perform only one function

---

* *Expressions whose main purpose isn't to do the work of the loop but to control the loop*

Statements - Iterative

- **Iterative Statements (cont.)**

Guidelines for Exiting the Loop

- Assure that the loop ends
- Make loop-termination conditions obvious

  - *'for'* loop: don't temper with the loop index; don't use *goto* or *break* to get out the loop

  - *'while'* loop: put all the control in the while clause
- Avoid code that depends on the loop index's final value

Statements - Iterative

Iterative Statements (cont.)

Guidelines for Exiting the Loop

- Use *break* rather than a boolean *flag* in a '*while*' loop
- Avoid a loop with a lot of breaks scattered through it
- Use *continue* for tests at the top of a loop
- Use *break* and *continue* with caution

  - No possibility of treating the loop as a black box

Statements - Iterative

Iterative Statements (cont.)

Guidelines for Using Loop Variables

- Use ordinal or enumerated types for limits on both arrays and loops
- Use meaningful variable names to make nested loops readable

  - No *i*, *j*, *k* for nested loops
- Avoid reuseing the same loop index in a loop or outside the loop

Statements - Iterative

Iterative Statements (cont.)

Guidelines for Loop Length

- Make loops short enough to view all at once
  e.g. __15-20__ *lines*
- Limit nesting to three levels
- Move code inside of long loops into routines

Construction Practices - Statements

REFERENCE

Textbook: *"Code Complete"*, 2nd edition

- Chapter 14: Organizing Straight-Line Code
- Chapter 15: Using Conditionals
- Chapter 16: Controlling Loops

Routines – Routine Parameters

Routine Parameters – Guidelines

- Put parameters in input-modify-output order
  - Maybe language specific
- If several routines use similar parameters, put the similar parameters in a consistence order
- Use all parameters passed to a routine
- Put status or error variables last

3.2 Software Construction - Guidelines & Practices

SE Principles, guidelines and good practices for:

- ~~3.2.1 Class Design & Implementation~~
- ~~3.2.2 Routine Design & Implementation~~
- ~~3.2.3 Statement Design & Implementation~~
- 3.2.4 Variable Design & Implementation
- ~~3.2.5 Program Layout & Formatting Style~~

Construction Practices - Variables

3.2 Software Construction - Guidelines & Practices

3.2.4 Variable Design & Implementation

- Variable Initialization
- Data Types of Variables
- Variable Names
- Naming Conventions

Variables - Initialization

Variable Initialization - Guidelines

- Initialize each variable as it's declared
  - char SE323StudentGrade = 'A';
- Declare, and define/initialize each variable close to where it's first used
- Use *final* or *const* when possible
- Pay attention to counters and accumulators
  - *i, j, k, sum, count, total*
- Initialize a class's member data in its constructor
- Check input parameters for validity

Variables - Data Types

Numbers - Guidelines

- Avoid "*magic numbers*"*
  - Use named constants or otherwise global variables
- Exception of 0 and 1 as loop control variables
- Special attention to "divide-by-zero" possibilities
- Make type conversion obvious
  - y = x + (float) i
- Avoid mixed-type comparisons

---

* *Literal numbers appear inside a program without explanation, e.g. 10, 323*

Variables - Data Types

Integers - Guidelines

- Integer division
  - 7/10 vs. 0.7
- Integer overflows

Integer type	Signed Range	Unsigned Range
8 bit	-128 – 127	0 – 255
16 bit	-32,768 – 32,767	0 – 65,535
32 bit	-2,147,483,648 – 2,147,483,647	0 – 4,294,967,295
64 bit	-9,223,372,036,854,775,808 – 9,223,372,036,854,775,807	0 – 18,446,744,073,709,551,615

Variables - Data Types

Floating Point Numbers - Guidelines

- Avoid additions and subtractions on numbers with greatly different magnitudes
  - 1,000,000.00 + 0.1
  - 5,000,000.02 – 5,000,000.01
- Avoid equality comparison
- Anticipate rounding error
- Check language and library support for specific data types, e.g. Currency (VB)

```
double norminal = 1.0;
double sum = 0.0;
for (int i = 0; I < 10; i++)
      sum +=0.1
if (nominal == sum) ??
```

X

Variables - Data Types

Characters and Strings - Guidelines

- Avoid "*magic characters and strings*"*
- Use named constants or global variables
- Watch for off-by-one string indexing errors
- Decide on internationalization/localization strategy early in the lifetime of a program
- Check character sets, e.g. Unicode, ISO 8859

* *Literal characters/strings appear inside a program, e.g. 'A', "SE323"*

Variables - Data Types

Boolean - Guidelines

Use boolean variables to document program

```
If ((elementIndex < 0) || (MAX_ELEMENTS < elementIndex) ||
   (elementIndex == lastElementIndex)) {
…
}
```

X

```
finished = ( (elementIndex < 0) || (MAX_ELEMENTS < elementIndex));
repeatedEntry = (elementIndex = = lastElementIndex);
if (finished || repeatedEntry) {
…
}
```

√

Variables - Data Types

- **Boolean – Guidelines (cont.)**
  - Use boolean variables to simplify tests

```
If ((document.AtEndOfStream()) and (Not inputError)) And _
   ((MIN_LINES <= lineCount) And (lineCount <= MAXLINES)) And _
   ( Not ErrorProcessing()) then
   ' do something or other
  ...
End If
```

X

```
allDataRead = (document.AtEndOfStream() and (Not inputError)
legalLineCount = ((MIN_LINES <= lineCount) And (lineCount <= MAXLINES))
If (allDataRead) and (legalLineCount) And (Not Error Processing()) then
' do something or other
  ...
End If
```

√

Variables - Data Types

Enumerated Types - Guidelines

```
Public Enum Color
        Color_Red
        Color_Blue
        Color_Green
End Enum
```

- Use enumerated types for readablility, e.g.
  - *"if chosenColor == 1"* *// bad example*
  - *"if chosenColor == Color_Red"* *// good example*
- Use as an alternative to boolean variables
- Use to check for invalid values
- Special attention to the first and last elements of an enumeration

Variables - Data Types

Enumerated Types - Example

Const AREA_CODE_LENGTH = 3

```
For  i = 1 to 12
        profit(i) = revenue(i) – expense(i)
Next
```

X

```
For  i = 1 to Num_MONTHS_IN_YEAR
        profit(i) = revenue(i) – expense(i)
Next
```

√

```
For month = 1 to Num_MONTHS_IN_YEAR
        profit(month) = revenue(month) – expense(month)
Next
```

√ √

```
For month = Month_January to Month_December
        profit(month) = revenue(month) – expense(month)
Next
```

√ √ √

Variables – Variable Names

Variable Names – Guidelines

- Wording Selection

  The name fully and accurately describes the entity the variable represents, e.g. *current date*:

  - *x, x1, x2, c, cd, current, date* *// bad example*
  - *currentDate, todaysDate* *// good example*

- The name reflects the problem (what) not solution (how). Logical not implementation level, e.g. *a record of employee data*:

  - inputRec, bitFlag *// bad example*
  - employeeData, printerReady *// good example*

Variables – Variable Names

Variable Names – Guidelines (cont.)

Optimal Name Length

- ___8-20___ characters, e.g.
  - *numberOfPeopleOnTheUSOlympicTeam, numberOfSeatsInTheStadium* *// bad example*
  - *N, np, ntm, n, ns, nsisd* *// bad example*
  - *numTeamMembers, teamMemberCount, numSeatInStadium, seatCount* *// good example*

Variables – Variable Names

Variable Names – Guidelines (cont.)

- Computed-Value Qualifiers in Variable Names

  Put the modifier at the end of the name, e.g.

  - totalRevenue, averageExpense, numCustomers *// bad example*
  - revenueTotal, expenseAverage, customerCount *// good example*

- Common Opposites in Variable Names

  Use common-language opposites precisely, e.g.

  - begin & end
  - first & last
  - min & max

Variables – Variable Names

Variable Names – Guidelines (cont.)

Loop Index Variables

- i, j, k only for short simple loops

```
For ( i=firstItem;  i < lastItem;  i++ )
      data[ i ] = 0;
```

- Meaningful names if the loop is longer than a few lines or if an index variable to be used outside the loop

```
for ( teamIndex = 0;  teamIndex < teamCount;  teamIndex++)
      for ( eventIndex = 0;  eventIndex < eventCount[ teamIndex ];  eventIndex++)
            score [teamIndex ] [ eventIndex ] = 0;
```

Variables – Variable Names

Variable Names – Guidelines (cont.)

Status Variables

- Meaningful names instead of *"flag"*

```
if (flag) ......   →   if (dataReady) .......
```

// Meaningful flag name

```
if (printFlag == 16)   →   if (reportType == ReportType_Annual)
```

// With enumerated type

```
statusFlag = 0X80   →   characterType = CONTROL_CHARACTER
```

// With named constant

Variables – Variable Names

Variable Names – Guidelines (cont.)

Boolean Variables

- Try to use typical boolean names, e.g.
  - *Done, error, found, success, ok*
- Use the names that imply only true or false
  - *status, sourcefile   // Bad example*
  - *statusOK, sourceFileAvailable   // Good example*
- Use ‘positive’ not ‘negative’ names
  - *notFound, notDone, notSucessful*
  - *If not notFound ....   !?*

Variables – Variable Names

Variable Names – Guidelines (cont.)

Enumerated Type Variables

- Names of members such that all belong to the same group, e.g. using a group prefix: *Color_*, *Month_*

```
Public Enum Color
	Color_Red
	Color_Green
	Color_Blue
	Color_White
	Color_Black
End Enum
```

```
Public Enum Month
	Month_January
	Month_February
	.........
	.........
	Month_December
End Enum
```

Variables –Naming Conventions

Naming Conventions – What?

- Standards for how variables should be named in a program
- Enforced by coding practices (human) not compliers (languages & computers)
- Examples:
  - '*i*,' '*j*', '*k*' for integer variables
  - "*ALL_CAPITAL_LETTERS*" for named constants

Variables –Naming Conventions

Naming Conventions – Why?

- One global decision over many local ones
- Knowledge transfer across projects
- Other people code can be learnt more quickly
- Name proliferation can be reduced
  - *pointTotal, totalPoint*
- Compensate for language weaknesses
  - Named constants, enumerated types, local, global etc.
- Emphasize relationships among related items

Variables –Naming Conventions

- **Naming Conventions – When?**
  - Multiple programmers on a project
  - Turn a program over to another programmer for modifications and maintenance
  - Programs are reviewed by other programmers
  - Program is so large that must think in pieces
  - Programs will be long-lived
  - Have a lot of unusual terms in a project

Variables –Naming Conventions

Naming Conventions – How?

- Differentiate: Variable vs. Routine names
  - *variableName* vs. *RoutineName()*
- Differentiate: Types vs. variables of the types
  - *TypeName* vs. *variableName*
- Identify global variables
  - g_var
- Identify member variables
  - m_var

Variables –Naming Conventions

- **Naming Conventions – How? (cont.)**
  - Identify type definitions
    - COLOR, t_Color
  - Identify named constants
    - *LINES_PER_PAGE_MAX, c_LinesPerPageMax*
  - Identify elements of enumerated types
  - Identify input-only parameters
  - Format names to enhance readability
    - Capitalization or spacing characters to separate words

Variables –Naming Conventions

Naming Conventions – Examples

- c, ch are character variables
- i, j are integer indexes
- p is a pointer
- Constants, typedefs, and preprocessor are ALL_CAPS
- Class/type names are in MixedUpperAndLowerCase()
- Variable/method names are: variableOrRoutineName
- The *get* and *set* prefixes are used for accessor methods

Variables – Naming Conventions

Naming Conventions – Names to Avoid

- Misleading names or abbreviations
  - *"FALSE" ("Fig and Almond Season")*
- Names with similar meanings
  - *input* vs. *inputValue*, *recordNum* vs. *numRecord*
- Different meanings but similar names
  - clientRecords vs. clientReports
- Sound similar
  - wrap vs. rap

Variables – Naming Conventions

Naming Conventions – Names to Avoid (cont.)

- Numerals in names
  - *total1, total2*
- Misspelled and commonly misspelled words
  - *prefered, reciept, conceieve*
- Multiple natural languages
  - *color* vs. *colour*, *check* vs. *cheque*
- Containing hard-to-read characters
  - **1** - *l*, *l* - *I*, **O** - *O*, 2 - *Z*, 5 - *S*, 6- *G*

Construction Practices - Variables

REFERENCE

Textbook: *"Code Complete"*, 2nd edition

- *Chapter 10: General Issues in Using variables*
- *Chapter 11: The Power of Variable Names*
- *Chapter 12: Fundamental Data Types*

3.2 Software Construction - Guidelines & Practices

SE Principles, guidelines and good practices for:

- ~~3.2.1 Class Design & Implementation~~
- ~~3.2.2 Routine Design & Implementation~~
- ~~3.2.3 Statement Design & Implementation~~
- ~~3.2.4 Variable Design & Implementation~~
- 3.2.5 Program Layout & Formatting Style

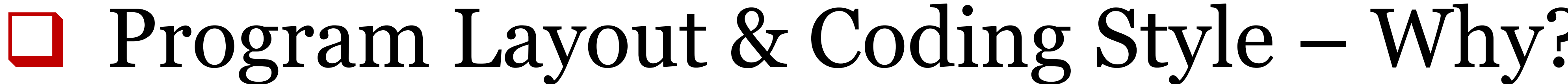
Construction Practices - Layout & Style

3.2 Software Construction - Guidelines & Practices

3.2.5 Program Layout & Formatting Style

- Layout Fundamentals
- Layout Techniques

Construction Practices – Layout & Style



- Program Layout & Coding Style – Why?

How Much Is Good Layout Worth?

> *Our studies support the claim that knowledge of programming plans and rules of programming discourse can have a significant impact on program comprehension. In their book called* [The] Elements of [Programming] Style, *Kernighan and Plauger also identify what we would call discourse rules. Our empirical results put teeth into these rules: It is not merely a matter of aesthetics that programs should be written in a particular style. Rather there is a psychological basis for writing programs in a conventional manner: programmers have strong expectations that other programmers will follow these discourse rules. If the rules are violated, then the utility afforded by the expectations that programmers have built up over time is effectively nullified. The results from the experiments with novice and advanced student programmers and with professional programmers described in this paper provide clear support for these claims.*
>
> Elliot Soloway and Kate Ehrlich

Construction Practices - Layout & Style

Layout Fundamentals – Program Layout Examples

```
/* Use the insertion sort technique to sort the "data" array in ascending order.
This routine assumes that data[ firstElement ] is not the first element in data and
that data[ firstElement-1 ] can be accessed. */ public void InsertionSort( int[]
data, int firstElement, int lastElement ) { /* Replace element at lower boundary
with an element guaranteed to be first in a sorted list. */ int lowerBoundary =
data[ firstElement-1 ]; data[ firstElement-1 ] = SORT_MIN; /* The elements in
positions firstElement through sortBoundary-1 are always sorted. In each pass
through the loop, sortBoundary is increased, and the element at the position of the
new sortBoundary probably isn't in its sorted place in the array, so it's inserted
into the proper place somewhere between firstElement and sortBoundary. */ for ( int
sortBoundary = firstElement+1; sortBoundary <= lastElement; sortBoundary++  ) { int
insertVal = data[ sortBoundary ]; int insertPos = sortBoundary; while ( insertVal <
data[ insertPos-1 ] ) { data[ insertPos ] = data[ insertPos-1 ]; insertPos =
insertPos-1; } data[ insertPos ] = insertVal; } /* Replace original lower-boundary
element */ data[ firstElement-1 ] = lowerBoundary; }
```

Example #1 *Bad Layout*

```java
/* Use the insertion sort technique to sort the "data" array in ascending
order. This routine assumes that data[ firstElement ] is not the
first element in data and that data[ firstElement-1 ] can be accessed. */
public void InsertionSort( int[] data, int firstElement, int lastElement ) {
/* Replace element at lower boundary with an element guaranteed to be first in a
sorted list. */
int lowerBoundary = data[ firstElement-1 ];
data[ firstElement-1 ] = SORT_MIN;
/* The elements in positions firstElement through sortBoundary-1 are
always sorted. In each pass through the loop, sortBoundary
is increased, and the element at the position of the
new sortBoundary probably isn't in its sorted place in the
array, so it's inserted into the proper place somewhere
between firstElement and sortBoundary. */
for (
int sortBoundary = firstElement+1;
sortBoundary <= lastElement;
sortBoundary++
) {
int insertVal = data[ sortBoundary ];
int insertPos = sortBoundary;
while ( insertVal < data[ insertPos-1 ] ) {
data[ insertPos ] = data[ insertPos-1 ];
insertPos = insertPos-1;
}
data[ insertPos ] = insertVal;
}
/* Replace original lower-boundary element */
data[ firstElement-1 ] = lowerBoundary;
}
```

***Example #2* Better Layout**

```java
/* Use the insertion sort technique to sort the "data" array in ascending
order. This routine assumes that data[ firstElement ] is not the
first element in data and that data[ firstElement-1 ] can be accessed.
*/

public void InsertionSort( int[] data, int firstElement, int lastElement ) {
   // Replace element at lower boundary with an element guaranteed to be
   // first in a sorted list.
   int lowerBoundary = data[ firstElement-1 ];
   data[ firstElement-1 ] = SORT_MIN;

   /* The elements in positions firstElement through sortBoundary-1 are
   always sorted. In each pass through the loop, sortBoundary
   is increased, and the element at the position of the
   new sortBoundary probably isn't in its sorted place in the
   array, so it's inserted into the proper place somewhere
   between firstElement and sortBoundary.
   */
   for ( int sortBoundary = firstElement + 1; sortBoundary <= lastElement;
      sortBoundary++ ) {
      int insertVal = data[ sortBoundary ];
      int insertPos = sortBoundary;
      while ( insertVal < data[ insertPos - 1 ] ) {
         data[ insertPos ] = data[ insertPos - 1 ];
         insertPos = insertPos - 1;
      }
      data[ insertPos ] = insertVal;
   }

   // Replace original lower-boundary element
   data[ firstElement - 1 ] = lowerBoundary;
}
```

Example #3 Best Layout

Construction Practices - Layout & Style

Layout Fundamentals – Program Layout Examples

Example #1

- Syntactically correct // will compile
- Semantically correct // will run with good results
- Good comments thoroughly
- Good variable names
- Clear logic
- ***The only problem is the bad program layout!***

Example #2

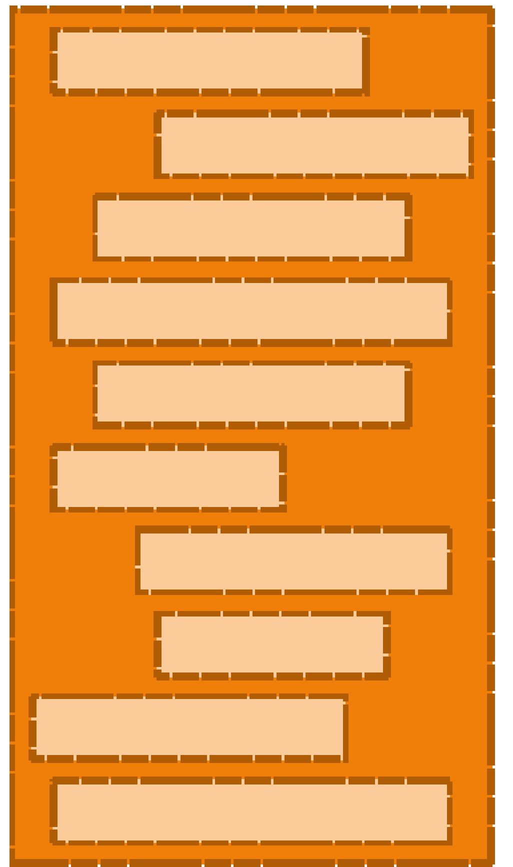
- An improvement of example #1 but still not readable by human

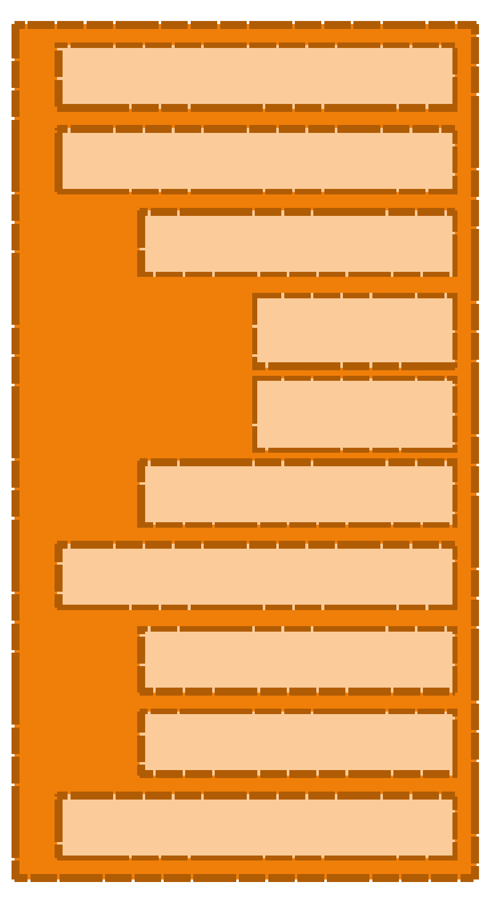
Example #3

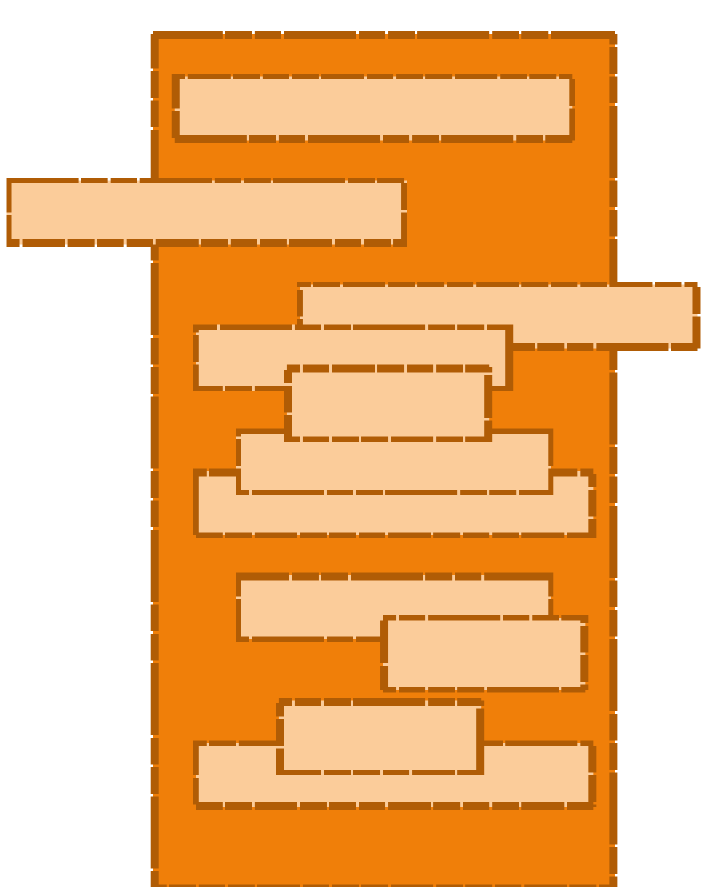
- Good practice // The only difference is the use of white spaces!

Construction Practices - Layout & Style

- Layout Fundamentals – Good Practice Principle
  - *"Good visual layout shows the logical **structure of a program.**"*
  - *"Write source code for **human** beings not computers to **see**."*







Construction Practices - Layout & Style

Layout Fundamentals – Objectives of Good Layout

- Accurately representing the logical structure of the code
- Consistently representing the logical structure of the code
- Improve readability
- Withstand modification

```
x = 3+4 * 2+7;
```

```
// swap left and right elements for whole array
for ( i = 0; i < MAX_ELEMENTS; i++ )
   leftElement = left[ i ];
   left[ i ]   = right[ i ];
   right[ i ]  = leftElement;
```

Examples *Different human VS computer interpretation of program code !*

3.2 Software Construction - Guidelines & Practices

3.2.5 Program Layout & Formatting Style

- ~~Layout Fundamentals~~
- Layout Techniques

Construction Practices - Layout & Style

Layout Techniques

- White Space
  - ✓ spaces
  - ✓ tabs
  - ✓ line breaks
  - ✓ blank lines
- Parentheses: ()
- Braces: { }
- Indentation

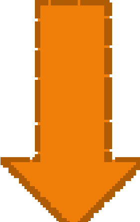
Construction Practices - Layout & Style

- Layout Techniques – Spaces

  Use spaces to make code more clear, distinct and standout

  - Use spaces to separate identifiers

```
while(pathName[startPath+position]<>';') and
   ((startPath+position)<length(pathName)) do
```



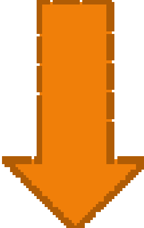
```
while ( pathName[ startPath+position ] <> ';' ) and
   (( startPath + position ) < length( pathName )) do
```

Construction Practices - Layout & Style

Layout Techniques – Spaces

- Use spaces to separate array references

```
grossRate[census[groupId].gender,census[groupId].ageGroup]
```



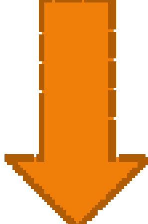
```
grossRate[ census[ groupId ].gender, census[ groupId ].ageGroup ]
```

Construction Practices - Layout & Style

Layout Techniques – Spaces

- Use spaces to separate routine arguments

```
ReadEmployeeData(maxEmps,empData,inputFile,empCount,inputError);
```



```
GetCensus( inputFile, empCount, empData, maxEmps, inputError );
```

Construction Practices - Layout & Style

- Layout Techniques – Spaces

Additional suggestion for using spaces:

- Put one space after each comma and semicolon
- Put one space on either side of a binary operator
- Do not put spaces immediately after a left parenthesis or before a right parenthesis
- Do not put spaces before a semicolon
- Put one space before a left parenthesis, except for an expression, routine calls, or before an empty parameter list
- When referring to the type of an array, do not put any spaces between the element type and the square brackets, e.g. *int[]*

Construction Practices - Layout & Style

Layout Techniques – Tabs

- Used for the purpose of indentation (see indentation)
- To vertically line up statements/comments (endline layout)
- Common practice: tab character is normally set to 4-8 spaces

```
If ( soldCount > 10 And prevMonthSales > 10 ) Then
   If ( soldCount > 100 And prevMonthSales > 10 ) Then
      If ( soldCount > 1000 ) Then
                         markdown = 0.1
                         profit = 0.05
                       Else
                         markdown = 0.05
                       End If
                                            Else
                                               markdown = 0.025
                                            End If
                                        Else
                                           markdown = 0.0
                                        End If
```

```
bool ReadEmployeeData( int                  maxEmployees,
                       EmployeeList         *employees,
                       EmployeeFile         *inputFile,
                       int                  *employeeCount,
                       bool                 *isInputError )
...
void InsertionSort( SortArray   data,
                    int         firstElement,
                    int         lastElement )
```

Construction Practices - Layout & Style

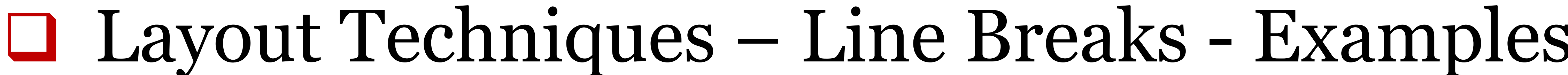
Layout Techniques – Line Breaks

- For easier comprehension of lengthy statements
- Good practice: keep related elements together while make the next line continuation obvious

```
if ((('0' <= inChar) && (inChar <= '9')) || (('a' <= inChar) &&
   (inChar <= 'z')) || (('A' <= inChar) && (inChar <= 'Z')))
   ...
```

```
if ( ( ( '0' <= inChar ) && ( inChar <= '9' ) ) ||
   ( ( 'a' <= inChar ) && ( inChar <= 'z' ) ) ||
   ( ( 'A' <= inChar ) && ( inChar <= 'Z' ) ) )
   ...
```

Construction Practices – Layout & Style



Layout Techniques – Line Breaks - Examples

```
customerBill = previousBalance( paymentHistory[ customerID ] ) + lateCharge(
  paymentHistory[ CustomerID ] );
```

```
customerBill = previousBalance( paymentHistory[ customerID ] )
  + lateCharge( paymentHistory[ CustomerID ] );
```

```
customerBill = previousBalance( paymentHistory[ customerID ] ) +
  lateCharge( paymentHistory[ CustomerID ] );
```

Construction Practices - Layout & Style

Layout Techniques – Blank Lines

- To divide groups of related statements into paragraphs (blocks)
- To separate modules from each other

```
cursor.start = startingScanLine;
cursor.end   = endingScanLine;
window.title = editWindow.title;
window.dimensions      = editWindow.dimensions;
window.foregroundColor = userPreferences.foregroundColor;
cursor.blinkRate       = editMode.blinkRate;
window.backgroundColor = userPreferences.backgroundColor;
SaveCursor( cursor );
SetCursor( cursor );
```

```
window.dimensions = editWindow.dimensions;
window.title = editWindow.title;
window.backgroundColor = userPreferences.backgroundColor;
window.foregroundColor = userPreferences.foregroundColor;

cursor.start = startingScanLine;
cursor.end = endingScanLine;
cursor.blinkRate = editMode.blinkRate;
SaveCursor( cursor );
SetCursor( cursor );
```

Construction Practices - Layout & Style

- Layout Techniques – Blank Lines
  - To separate parts of a module, e.g. class/routine header, variable declaration, class/routine body
  - To separate comments from actual code

```
// comment zero
CodeStatementZero;
CodeStatementOne;
//comment one
CodeStatementTwo;
CodeStatementThree;
```

```

// comment zero
CodeStatementZero;
CodeStatementOne;

//comment one
CodeStatementTwo;
CodeStatementThree;
```

```

// comment zero

CodeStatementZero;
CodeStatementOne;

//comment one

CodeStatementTwo;
CodeStatementThree;
```

Construction Practices - Layout & Style

Layout Techniques

Braces { }

- To form blocks of statements under control structure
- Opening and closing braces should line up

```
While (i < n)   {
     System.out.println (a[i]);
      i++;
}
```

?

```
While (i < n)
{
     System.out.println (a[i]);
      i++;
}
```

- To clarify the nested statements

```
If (……)
    if (……)   ……….;
else ……….;
```

```
If (……)
{
    if (……)   ……….;
}
else ……..;
```

```
If (……)
{
    if (……)   ……….;
    else ……..;
}
```

Construction Practices - Layout & Style

- Layout Techniques

<u>Parentheses</u> ( )

- To clarify expressions with more than 2 terms,
  e.g. 12 + 4 % 3 * 7 / 8 → ((12 + 4) % (3 * 7)) / 8

<u>Indentation</u>

- To show the logical structure of a program
- To designate subordinate statements
- To form a block of code
- To designate the next level of nesting

Construction Practices - Layout & Style

- Layout Techniques – Indentations & Comments
  - Indent a comment with its corresponding code

Listing 31-56. Visual Basic example of poorly indented comments.

```
For transactionId = 1 To totalTransactions
' get transaction data
   GetTransactionType( transactionType )
   GetTransactionAmount( transactionAmount )

' process transaction based on transaction type
   If transactionType = Transaction_Sale Then
      AcceptCustomerSale( transactionAmount )

   Else
      If transactionType = Transaction_CustomerReturn Then

' either process return automatically or get manager approval, if required
         If transactionAmount >= MANAGER_APPROVAL_LEVEL Then

' try to get manager approval and then accept or reject the return
' based on whether approval is granted
            GetMgrApproval( isTransactionApproved )
            If ( isTransactionApproved ) Then
               AcceptCustomerReturn( transactionAmount )
            Else
               RejectCustomerReturn( transactionAmount )
            End If
         Else

' manager approval not required, so accept return
            AcceptCustomerReturn( transactionAmount )
         End If
      End If
   End If
Next
```

Listing 31-57. Visual Basic example of nicely indented comments.

```
For transactionId = 1 To totalTransactions
   ' get transaction data
   GetTransactionType( transactionType )
   GetTransactionAmount( transactionAmount )

   ' process transaction based on transaction type
   If transactionType = Transaction_Sale Then
      AcceptCustomerSale( transactionAmount )

   Else
      If transactionType = Transaction_CustomerReturn Then

         ' either process return automatically or get manager approval, if required
         If transactionAmount >= MANAGER_APPROVAL_LEVEL Then

            ' try to get manager approval and then accept or reject the return
            ' based on whether approval is granted
            GetMgrApproval( isTransactionApproved )
            If ( isTransactionApproved ) Then
               AcceptCustomerReturn( transactionAmount )
            Else
               RejectCustomerReturn( transactionAmount )
            End If
         Else
            ' manager approval not required, so accept return
            AcceptCustomerReturn( transactionAmount )
         End If
      End If
   End If
Next
```

Construction Practices - Layout & Style

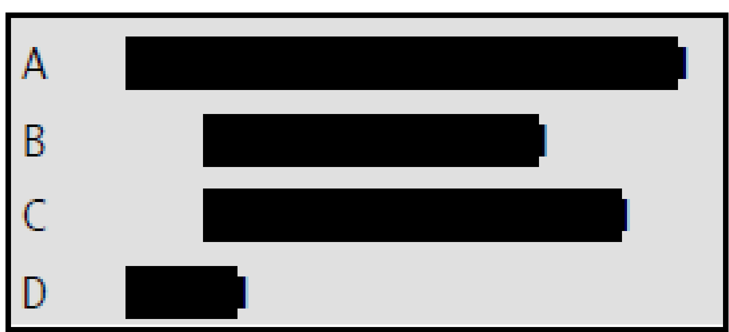
- Layout Techniques – Indentations & Parameters
  - Use standard indentation for routine parameters

```
bool ReadEmployeeData(int maxEmployees,EmployeeList *employees,
   EmployeeFile *inputFile,int *employeeCount,bool  *isInputError)
...

void InsertionSort(SortArray data,int firstElement,int lastElement)
```

```
public bool ReadEmployeeData(
   int maxEmployees,
   EmployeeList *employees,
   EmployeeFile *inputFile,
   int *employeeCount,
   bool *isInputError
)
...

public void InsertionSort(
   SortArray data,
   int firstElement,
   int lastElement
)
```

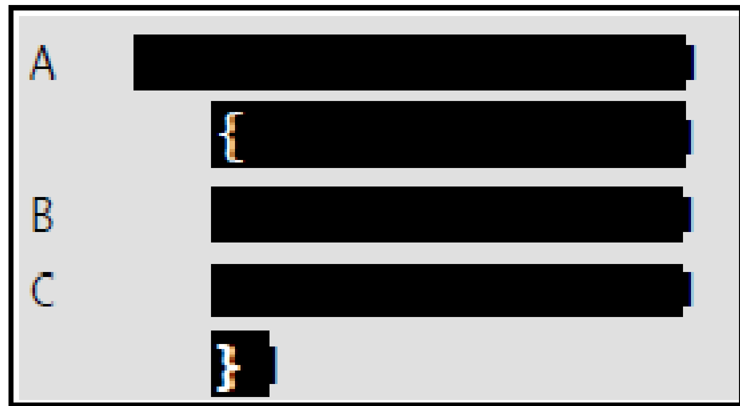
Construction Practices - Layout & Style

- Layout Techniques – Indentations & Braces = Blocks




```
While pixelColor = Color_Red
   statement1
   statement2
   ...
Wend
```

```
while ( pixelColor == Color_Red ) {
   statement1;
   statement2;
   ...
}
```




```
while ( pixelColor == Color_Red )
   {
   statement1;
   statement2;
   ...
   }
```

Construction Practices - Layout & Style

REFERENCE

Textbook: *"Code Complete"*, 2nd edition

- *Chapter 31: Layout and Style*
- *http://www.oracle.com/technetwork/java/codeconvtoc-136057.html*
- http://geosoft.no/development/javastyle.html