Chapter 1: Principles of Programming and Software Engineering

Data Abstraction & Problem Solving with C++

Fifth Edition

by Frank M. Carrano

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley. Ver. 5.0.

Applying the UML to OOA/D

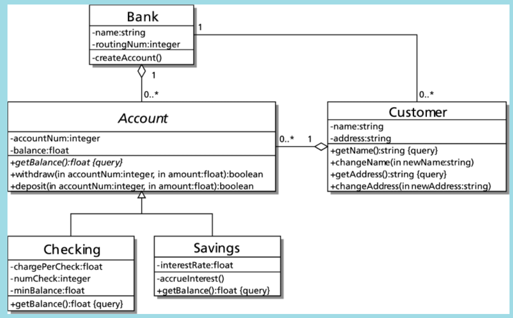

Figure 1-5 A UML class diagram of a banking system

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley. Ver. 5.0.

Applying the UML to OOA/D

- Class relationships
  - Association
    - The classes know about each other
    - Example: The Bank and Customer classes
  - Aggregation (Containment)
    - One class contains an instance of another class
    - Example: The Bank and Account classes
    - The lifetime of the containing object and the object contained are not necessarily the same
      - Banks "live" longer than the accounts they contain

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley. Ver. 5.0.

Applying the UML to OOA/D

- Class relationships (Continued)
  - Composition
    - A stronger form of aggregation
    - The lifetime of the containing object and the object contained are the same
    - Example: A ballpoint pen
      - When the pen "dies," so does the ball

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley. Ver. 5.0.

Applying the UML to OOA/D

- Class relationships (Continued)
  - Generalization
    - Indicates a family of classes related by inheritance
    - Example: Account is an ancestor class; the attributes and operations of Account are inherited by the descendant classes, Checking and Savings

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley. Ver. 5.0.

Applying the UML to OOA/D

- Notation
  - Association
    - A relationship between two classes is shown by a connecting solid line
    - Relationships more specific than association are indicated with arrowheads, as you will see
    - Multiplicities: Optional numbers at the end(s) of an association or other relationship
      - Each bank object is associated with zero or more customers (denoted 0..*), but each customer is associated with one bank
      - Each customer can have multiple accounts of any type, but an account can belong to only one customer

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley. Ver. 5.0.

Applying the UML to OOA/D

- Notation (Continued)
  - Aggregation (Containment)
    - Denoted by an open diamond arrowhead pointing to the containing class
  - Composition
    - Denoted by a filled-in diamond arrowhead pointing to the containing class
  - Generalization (Inheritance)
    - Denoted by an open triangular arrowhead pointing to the ancestor (general or parent) class
  - UML also provides notation to specify visibility, type, parameter, and default value information

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley. Ver. 5.0.

The Software Life Cycle

- Describes the phases of software development from conception to deployment to replacement to deletion
  - We will examine the phases from project conception to deployment to end users
  - Beyond this development process, software needs maintenance to correct errors and add features
  - Eventually software is retired

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley. Ver. 5.0.

Iterative and Evolutionary Development

- Iterative development of a solution to a problem
  - Many short, fixed-length iterations
  - Each iteration builds on the previous iteration until a complete solution is achieved
  - Each iteration cycles through analysis, design, implementation, testing, and integration of a small portion of the problem domain
  - Early iterations create the core of the system; further iterations build on that core

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley. Ver. 5.0.

Iterative and Evolutionary Development

- Each iteration has a duration called the timebox
  - Chosen at beginning of project
  - Typically 2 to 4 weeks
- The partial system at the end of each iteration should be functional and completely tested
- Each iteration makes relatively few changes to the previous iteration
- End users can provide feedback at the end of each iteration

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley. Ver. 5.0.

Rational Unified Process (RUP) Development Phases

- RUP gives structure to the software development process
- RUP uses the OOA/D tools we introduced
- Four development phases:
  - Inception: feasibility study, project vision, time/cost estimates
  - Elaboration: refinement of project vision, time/cost estimates, and system requirements; development of core system
  - Construction: iterative development of remaining system
  - Transition: testing and deployment of the system

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley. Ver. 5.0.

Rational Unified Process (RUP) Development Phases

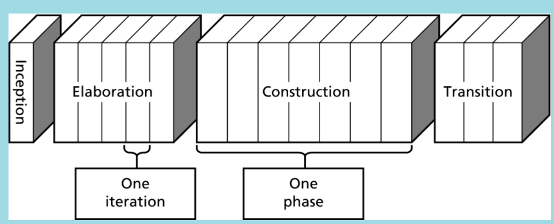

Figure 1-7 RUP development phases

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley. Ver. 5.0.

Rational Unified Process (RUP) Development Phases

- Inception phase
  - Define initial set of system requirements
  - Generate a core set of use case scenarios (about 10% of total number)
  - Identify highest-risk aspects of solution
  - Choose iteration timebox length

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley. Ver. 5.0.

Rational Unified Process (RUP) Development Phases

- Elaboration phase
  - Iteratively develop core architecture of system
  - Address highest-risk aspects of system
    - Most potential for system failure, so deal with them first
  - Define most of the system requirements
  - Extends over at least 2 iterations to allow for feedback
  - Each iteration progresses through OO analysis and design (use case scenarios, sequence diagrams, class diagrams), coding, testing, integration, and feedback

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley. Ver. 5.0.

Rational Unified Process (RUP) Development Phases

- Construction phase
  - Begins once most of the system requirements are formalized
  - Develops the remaining system
  - Each iteration requires less analysis and design
  - Focus is on implementation and testing
- Transition phase
  - Beta testing with advanced end users
  - System moves into a production environment

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley. Ver. 5.0.

Rational Unified Process (RUP) Development Phases

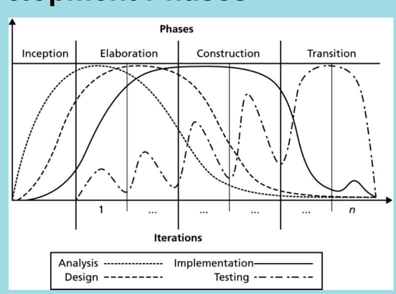

Figure 1-8 Relative amounts of work done in each development phase

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley. Ver. 5.0.

What About the Waterfall Method of Development?

- Develops a solution sequentially by moving through phases: requirements analysis, design, implementation, testing, deployment
- Hard to correctly specify a system without early feedback
- Wrong analysis leads to wrong solution
- Outdated and should not be used
- Do not impose this method on RUP development

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley. Ver. 5.0.

Achieving a Better Solution

- Analysis and design improve solutions
- What aspects of one solution make it better than another?
- What aspects lead to better solutions?

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley. Ver. 5.0.

Evaluation of Designs and Solutions

- Cohesion
  - A highly cohesive module performs one well-defined task
    - A person with low cohesion has "too many irons in the fire"
  - Promotes self-documenting, easy-to-understand code
  - Easy to reuse in other software projects
  - Easy to revise or correct
  - Robust: less likely to be affected by change; performs well under unusual conditions
  - Promotes low coupling

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley. Ver. 5.0.

Evaluation of Designs and Solutions

- Coupling
  - Modules with low coupling are independent of one another
  - System of modules with low coupling is
    - Easier to change: A change to one module won't affect another
    - Easier to understand
  - Module with low coupling is
    - Easier to reuse
    - Has increased cohesion
  - Coupling cannot be and should not be eliminated entirely
    - Objects must collaborate
  - Class diagrams show dependencies among classes, and hence coupling

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley. Ver. 5.0.

Evaluation of Designs and Solutions

- Minimal and complete interfaces
  - A class interface declares publicly accessible methods (and data)
    - Describes only way for programmers to interact with the class
  - Classes should be easy to understand, and so have few methods
    - Desire to provide power is at odds with this goal
  - Complete interface
    - Provides methods for any reasonable task consistent with the responsibilities of the class
    - Important that an interface is complete
  - Minimal interface
    - Provides only essential methods
    - Classes with minimal interfaces are easier to understand, use, and maintain
    - Less important than completeness

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley. Ver. 5.0.

Evaluation of Designs and Solutions

- Signature: the interface for a method or function
  - Name of method/function
  - Arguments (number, order, type)
  - Qualifiers such as `const`

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley. Ver. 5.0.

Operation Contracts

- A module's operation contract specifies its
  - Purpose
  - Assumptions
  - Input
  - Output
- Begin the contract during analysis, finish during design
- Use to document code, particularly in header files

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley. Ver. 5.0.

Operation Contracts

- Specify data flow among modules
  - What data is available to a module?
  - What does the module assume?
  - What actions take place?
  - What effect does the module have on the data?

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley. Ver. 5.0.

Operation Contracts

- Contract shows the responsibilities of one module to another
- Does *not* describe how the module will perform its task
- Precondition: Statement of conditions that must exist before a module executes
- Postcondition: Statement of conditions that exist after a module executes

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley. Ver. 5.0.

Operation Contracts

First draft specifications

```
sort(anArray, num)
// Sorts an array.
// Precondition: anArray is an array of num
   integers; num > 0.
// Postcondition: The integers in anArray are
   sorted.
```

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley. Ver. 5.0.

Operation Contracts

Revised specifications

```
sort(anArray, num)
// Sorts an array into ascending order.
// Precondition: anArray is an array of num
// integers; 1 <= num <= MAX_ARRAY, where
// MAX_ARRAY is a global constant that specifies
// the maximum size of anArray.
// Postcondition: anArray[0] <= anArray[1] <= ...
// <= anArray[num-1], num is unchanged.
```

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley. Ver. 5.0.

Verification

- Assertion: A statement about a particular condition at a certain point in an algorithm
  - Preconditions and postconditions are examples of assertions
- Invariant: A condition that is always true at a certain point in an algorithm
- Loop invariant: A condition that is true before and after each execution of an algorithm’s loop
  - Can be used to detect errors before coding is started

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley. Ver. 5.0.

Verification

- Loop invariant (continued)
  - The invariant for a correct loop is true:
    - Initially, after any initialization steps, but before the loop begins execution
    - Before every iteration of the loop
    - After every iteration of the loop
    - After the loop terminates

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley. Ver. 5.0.

Verification

- It is possible to prove the correctness of some algorithms
  - Like proving a theorem in geometry
  - Starting with a precondition, you prove that each assertion before a step in an algorithm leads to the assertion after the step until you reach the postcondition

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley. Ver. 5.0.

What is a Good Solution?

- A solution is good if:
  - The total cost it incurs over all phases of its life cycle is minimal
- The cost of a solution includes:
  - Computer resources that the program consumes
  - Difficulties encountered by users
  - Consequences of a program that does not behave correctly
- Programs must be well structured and documented
- Efficiency is one aspect of a solution's cost

Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley. Ver. 5.0.