Chapter 1: Principles of Programming and Software Engineering

Data Abstraction & Problem Solving with C++

Fifth Edition

by Frank M. Carrano

Software Engineering and Object-Oriented Design

- Coding without a solution design increases debugging time
- A team of programmers for a large software development project requires
  - An overall plan
  - Organization
  - Communication
- Software engineering
  - Provides techniques to facilitate the development of computer programs

An Examination of Problem Solving

- Problem solving
  - The process of taking the statement of a problem and developing a computer program that solves that problem
- Object-oriented analysis and design (OOA / D)
  - A process for problem solving
  - A problem solution is a program consisting of a system of interacting classes of objects
    - Each object has characteristics and behaviors related to the solution
    - A class is a set of objects having the same type

Aspects of an Object-Oriented Solution

- A solution is a C++ program consisting of:
  - Modules
    - A single, stand-alone function
    - A method of a class
    - A class
    - Several functions or classes working closely together
    - Other blocks of code

Aspects of an Object-Oriented Solution

- Functions and methods implement algorithms
  - Algorithm: a step-by-step recipe for performing a task within a finite period of time
  - Algorithms often operate on a collection of data

Aspects of an Object-Oriented Solution

- Create a good set of modules
  - Modules must store, move, and alter data
  - Modules use algorithms to communicate with one another
- Organize your data collection to facilitate operations on the data

Abstraction and Information Hiding

- Abstraction
  - Separates the purpose of a module from its implementation
  - Specifications for each module are written before implementation
  - Functional abstraction
    - Separates the purpose of a module from its implementation

Abstraction and Information Hiding

- Data abstraction
  - Focuses on the operations of data, not on the implementation of the operations
- Abstract data type (ADT)
  - A collection of data and a set of operations on the data
  - You can use an ADT's operations without knowing their implementations or how data is stored, if you know the operations' specifications

Abstraction and Information Hiding

- Data structure
  - A construct that you can define within a programming language to store a collection of data
- Develop algorithms and ADTs in tandem

Abstraction and Information Hiding

- Information hiding
  - Hide details within a module
  - Ensure that no other module can tamper with these hidden details
  - Public view of a module
    - Described by its specifications
  - Private view of a module
    - Implementation details that the specifications should not describe

Principles of Object-Oriented Programming (OOP)

- Object-oriented languages enable us to build classes of objects
- A class combines
  - Attributes (characteristics) of objects of a single type
    - Typically data
    - Called data members
  - Behaviors (operations)
    - Typically operate on the data
    - Called methods or member functions

Principles of Object-Oriented Programming

- Three principles of object-oriented programming
  - Encapsulation
    - Objects combine data and operations
    - Hides inner details
  - Inheritance
    - Classes can inherit properties from other classes
    - Existing classes can be reused
  - Polymorphism
    - Objects can determine appropriate operations at execution time

Object-Oriented Analysis and Design

- Analysis
  - Process to develop
    - An understanding of the problem
    - The requirements of a solution
      - *What* a solution must be and do
      - Not *how* to design or implement it
  - Generates an accurate understanding of what end users will expect the solution to be and do
  - Think about the problem, not how to solve it

Object-Oriented Analysis and Design

- Object-oriented analysis (OOA)
  – Expresses an understanding of the problem and the requirements of a solution in terms of objects within the problem domain
  – Objects can represent
    - Real-world objects
    - Software systems
    - Ideas
  – OOA describes objects and their interactions among one another

Object-Oriented Analysis and Design

- Object-oriented design (OOD)
  - Expresses an understanding of a solution that fulfills the requirements discovered during OOA
  - Describes a solution in terms of
    - Software objects
    - The collaborations of these objects with one another
      - Objects collaborate when they send messages (call each other’s operations)
      - Collaborations should be meaningful and minimal
  - Creates one or more models of a solution
    - Some emphasize interactions among objects
    - Others emphasize relationships among objects

Applying the UML to OOA/D

- Unified Modeling Language (UML)
  - A tool for exploration and communication during the design of a solution
  - Models a problem domain in terms of objects independently of a programming language
  - Visually represents object-oriented solutions as diagrams
  - Its visual nature is an advantage, since we are visual creatures
  - Enables members of a programming team to communicate visually with one another and gain a common understanding of the system being built

Applying the UML to OOA/D

- UML use case for OOA
  - A set of textual scenarios (stories) of the solution
    - Each scenario describes the system's behavior under certain circumstances from the perspective of the user
      - Focus on the responsibilities of the system to meeting a user's goals
    - Main success scenario (happy path): interaction between user and system when all goes well
    - Alternate scenarios: interaction between user and system under exceptional circumstances
  - Find noteworthy objects, attributes, and associations within the scenarios

Applying the UML to OOA/D

- An example of a main success scenario
  - Customer asks to withdraw money from a bank account
  - Bank identifies and authenticates customer
  - Bank gets account type, account number, and withdrawal amount from customer
  - Bank verifies that account balance is greater than withdrawal amount
  - Bank generates receipt for the transaction
  - Bank counts out the correct amount of money for customer
  - Customer leaves bank

Applying the UML to OOA/D

- An example of an alternate scenario
  - Customer asks to withdraw money from a bank account
  - Bank identifies, but fails to authenticate customer
  - Bank refuses to process the customer’s request
  - Customer leaves bank

Applying the UML to OOA/D

- UML sequence (interaction) diagram for OOD
  - Models the scenarios in a use case
  - Shows the interactions among objects over time
  - Lets you visualize the messages sent among objects in a scenario and their order of occurrence
  - Helps to define the responsibilities of the objects
    - What must an object remember?
    - What must an object do for other objects?

Applying the UML to OOA/D

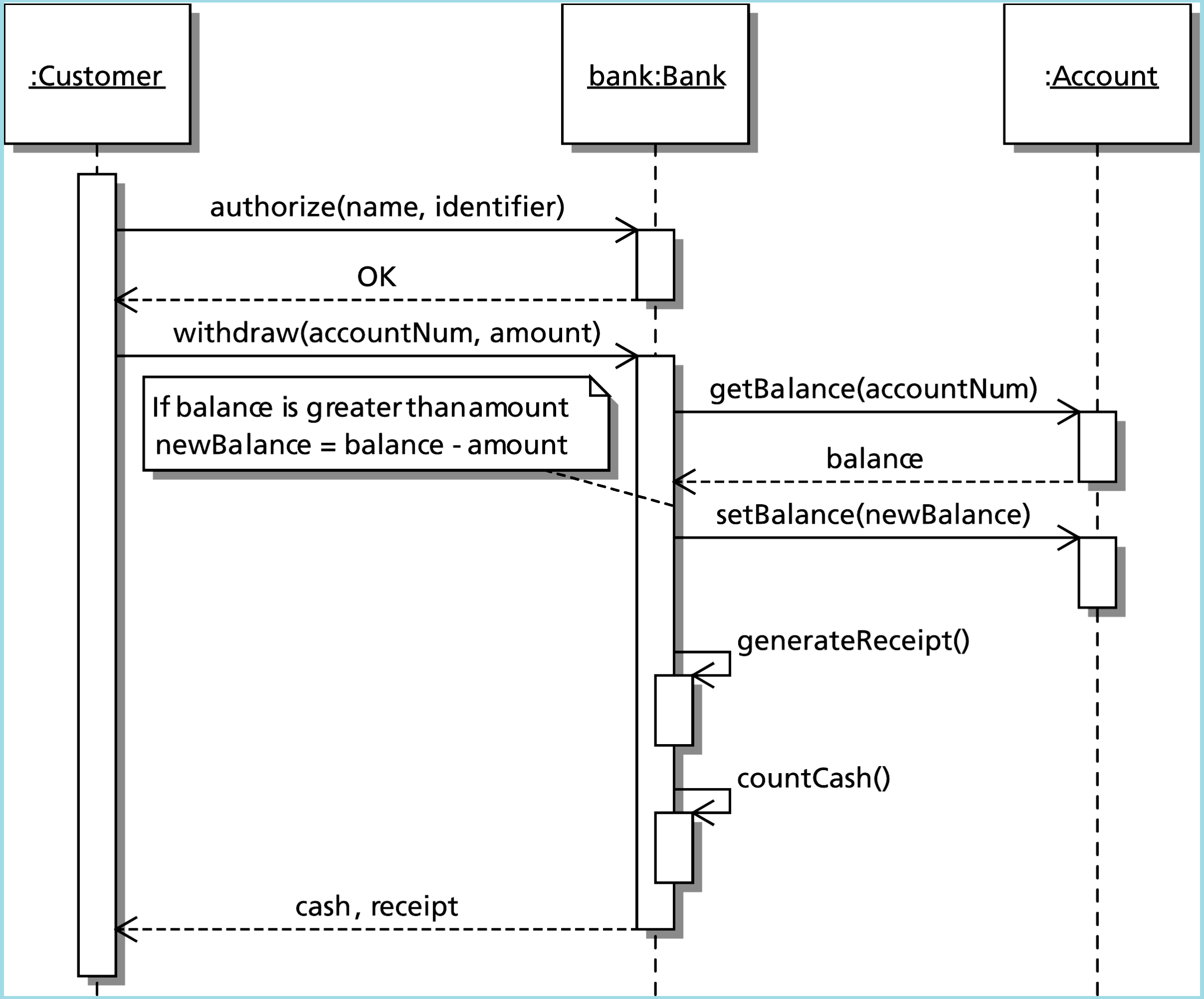

Figure 1-2 Sequence diagram for the main success scenario

Applying the UML to OOA/D

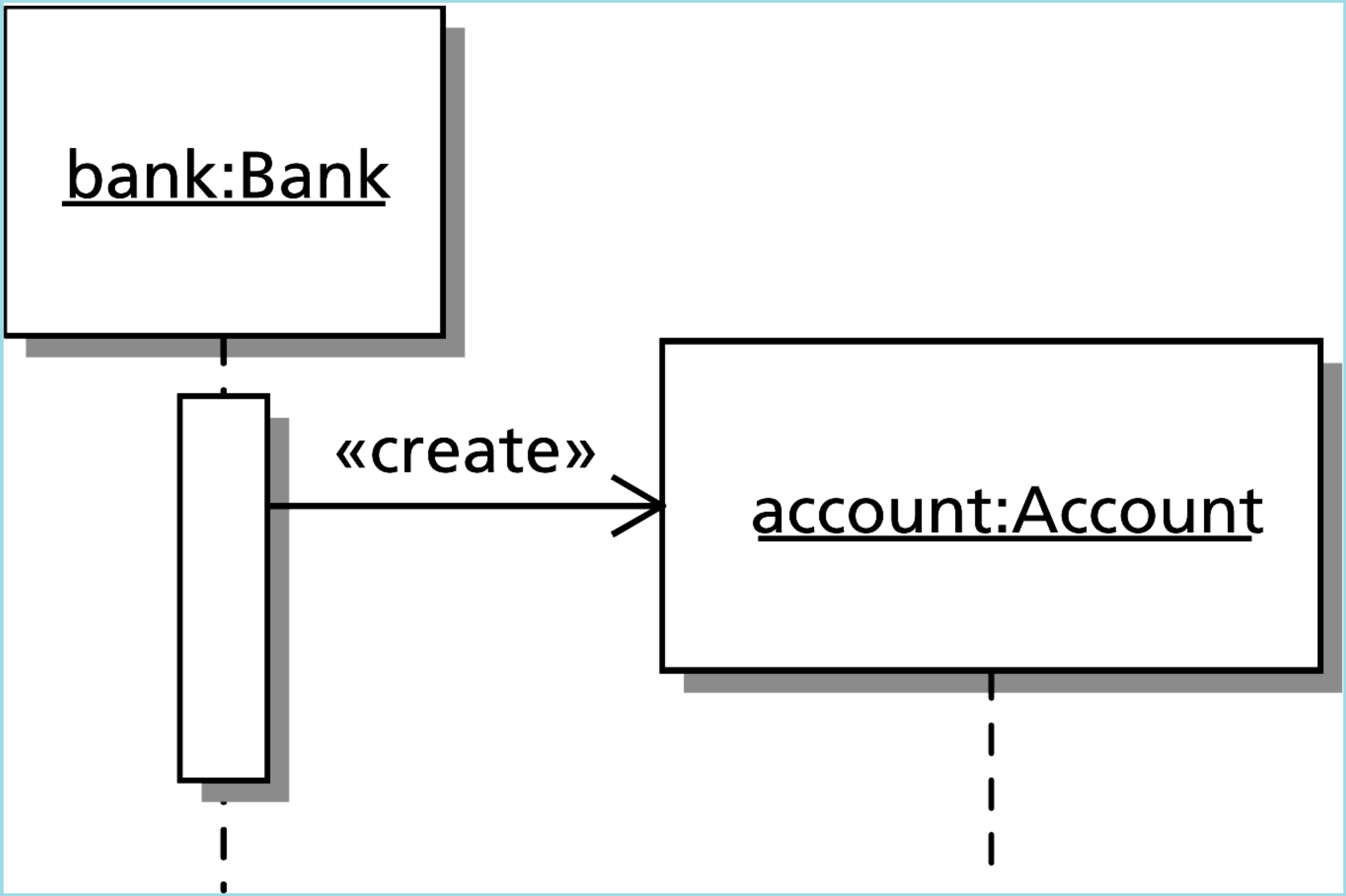

Figure 1-3 Sequence diagram showing the creation of a new object

Applying the UML to OOA/D

- UML class (static) diagram
  - Represents a conceptual model of a class of objects in a language-independent way
  - Shows the name, attributes, and operations of a class
  - Shows how multiple classes are related to one another

Applying the UML to OOA/D

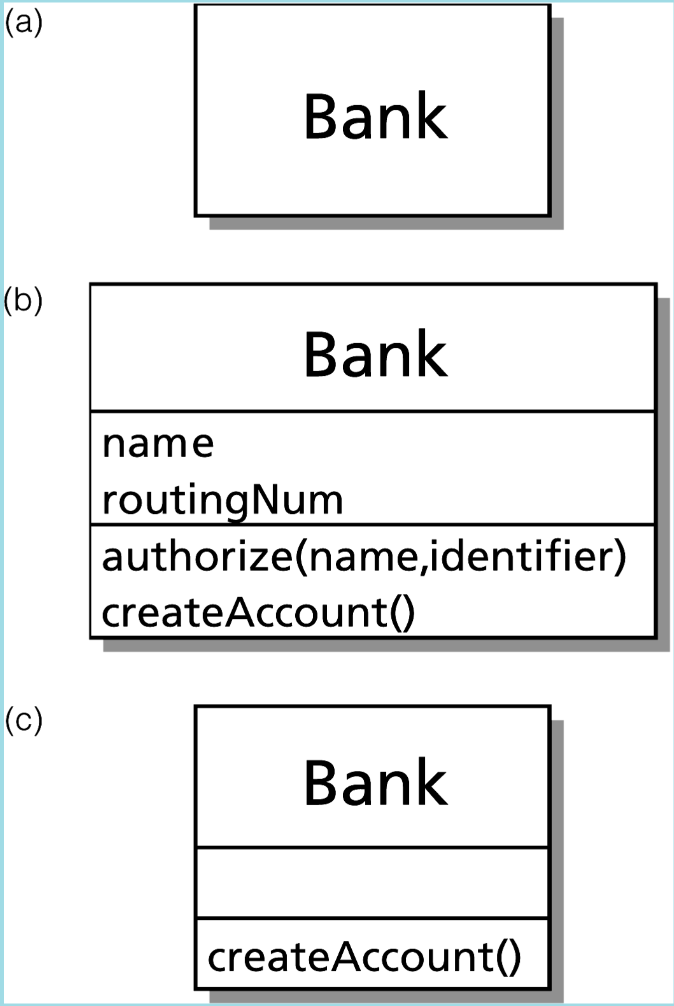

Figure 1-4 Three possible class diagrams for a class of banks

Applying the UML to OOA/D

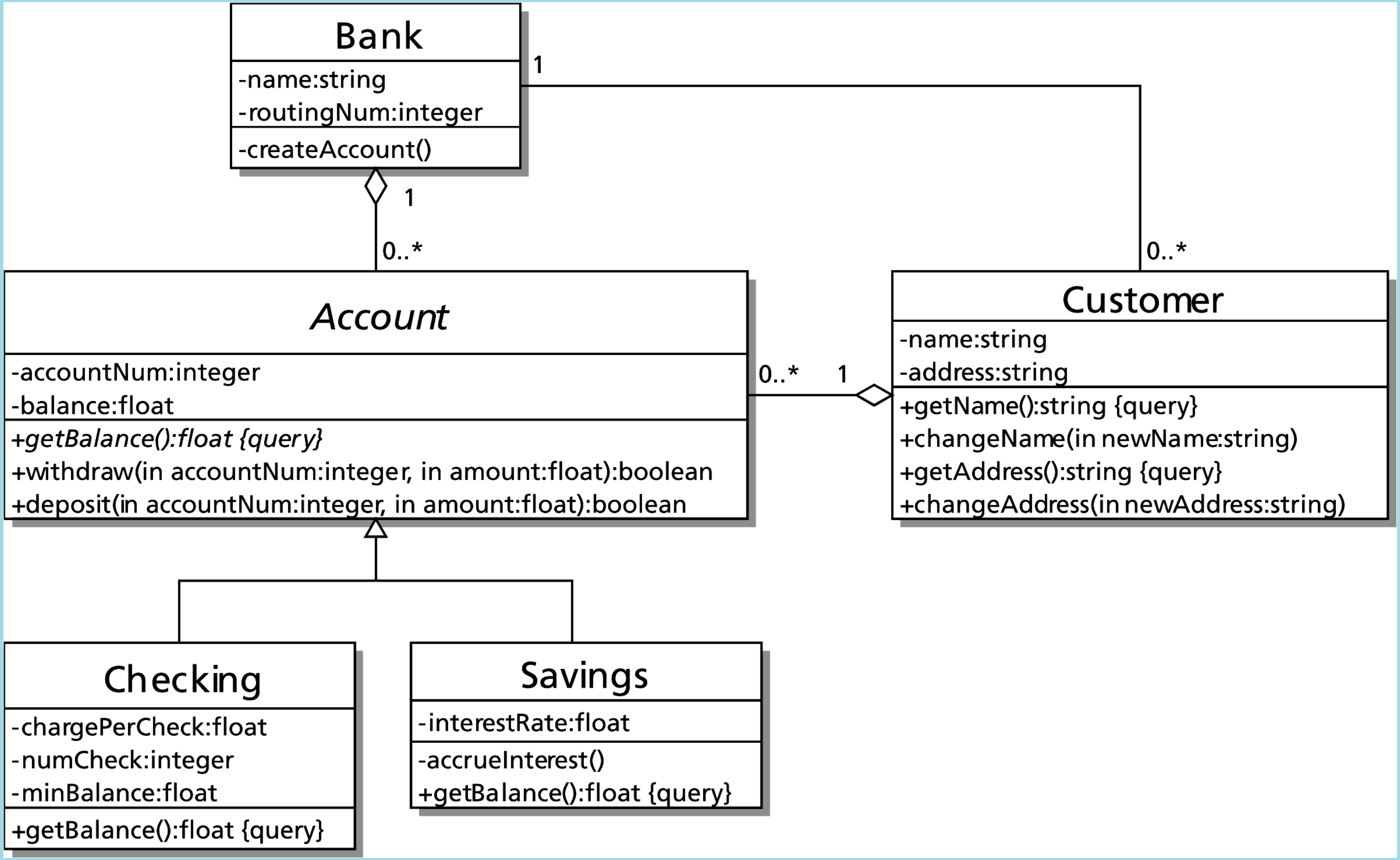

Figure 1-5 A UML class diagram of a banking system

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

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

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

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

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

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

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

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

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

Rational Unified Process (RUP) Development Phases

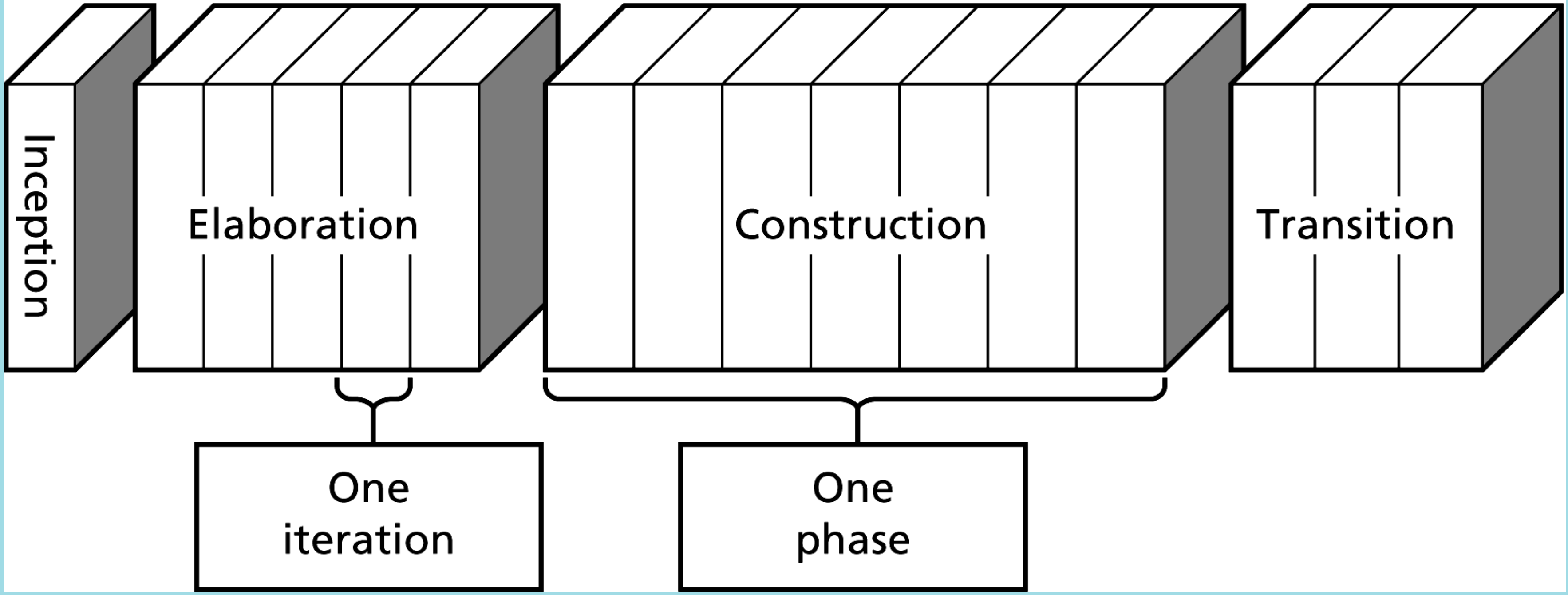

Figure 1-7 RUP development phases

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

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

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

Rational Unified Process (RUP) Development Phases

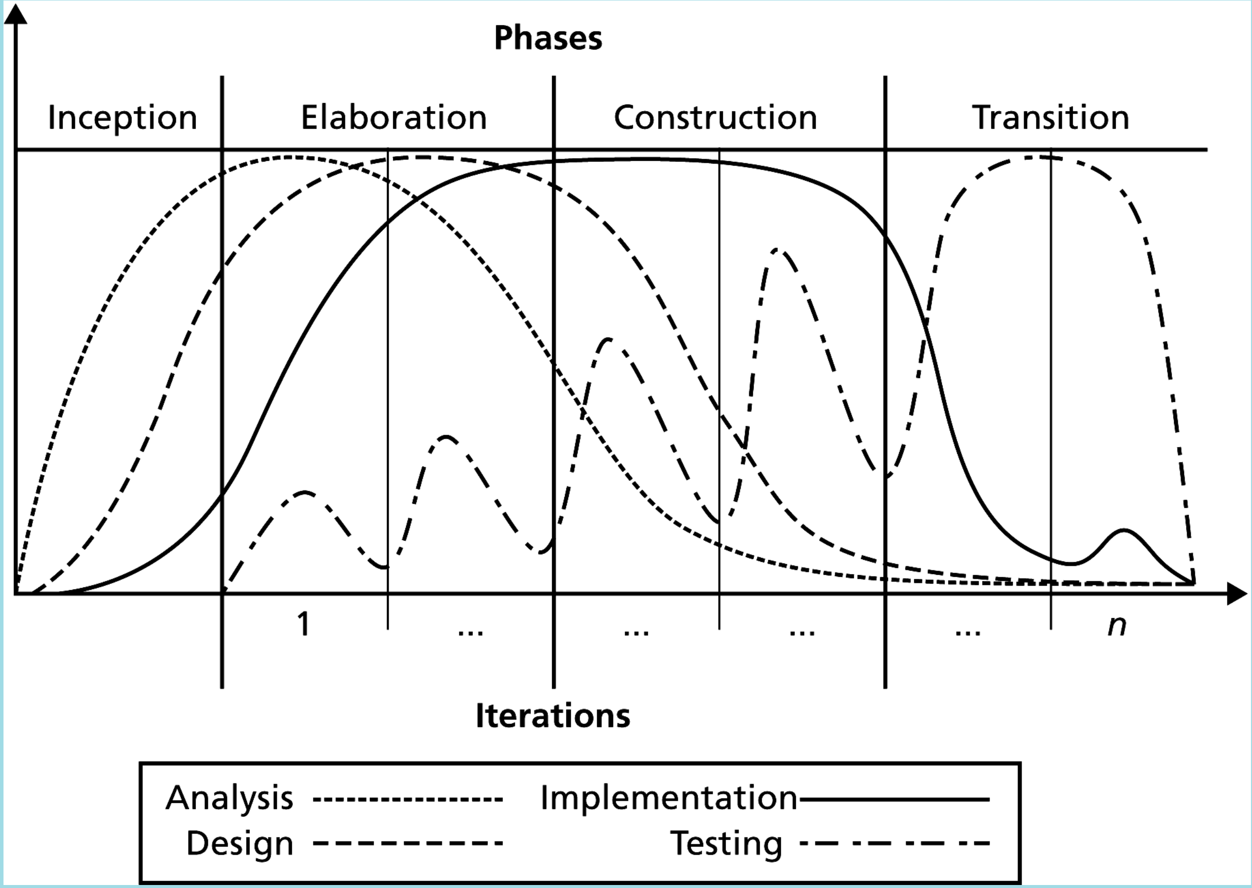

Figure 1-8 Relative amounts of work done in each development phase

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

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?

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

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

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

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`

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

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?

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

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.
```

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.
```

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

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

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

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

Key Issues in Programming

1. Modularity
2. Style
3. Modifiability
4. Ease of Use
5. Fail-safe programming
6. Debugging
7. Testing

Key Issues in Programming: Modularity

- Modularity has a favorable impact on
  - Constructing programs
  - Debugging programs
  - Reading programs
  - Modifying programs
  - Eliminating redundant code

Key Issues in Programming: Style

1. Use of private data members
2. Proper use of reference arguments
3. Proper use of methods
4. Avoidance of global variables in modules
5. Error handling
6. Readability
7. Documentation

Key Issues in Programming: Modifiability

- Modifiability is easier through the use of
  - Named constants
  - The `typedef` statement

Key Issues in Programming: Ease of Use

- In an interactive environment, the program should prompt the user for input in a clear manner
- A program should always echo its input
- The output should be well labeled and easy to read

Key Issues in Programming: Fail-Safe Programming

- Fail-safe programs will perform reasonably no matter how anyone uses it
- Test for invalid input data and program logic errors
- Check invariants
- Enforce preconditions
- Check argument values

Key Issues in Programming: Debugging

- Programmer must systematically check a program's logic to find where an error occurs
- Tools to use while debugging:
  - Single-stepping
  - Watches
  - Breakpoints
  - `cout` statements
  - Dump functions

Key Issues in Programming: Testing

- Levels
  - Unit testing: Test methods, then classes
  - Integration testing: Test interactions among modules
  - System testing: Test entire program
  - Acceptance testing: Show that system complies with requirements

Key Issues in Programming: Testing

- Types
  - Open-box (white-box or glass-box) testing
    - Test knowing the implementation
    - Test all lines of code (decision branches, etc.)
  - Closed-box (black-box or functional) testing
    - Test knowing only the specifications

Key Issues in Programming: Testing

- Developing test data
  - Include boundary values
  - Need to know expected results

Key Issues in Programming: Testing

- Techniques
  - `assert` statements to check invariants
  - Disable, but do not remove, code used for testing
    - /* and */
    - Booleans
    - Macros

Key Issues in Programming: Testing

- Stubs
  - An incompletely implemented method that simply acknowledges that it was called
- Drivers
  - A module that tests another module
  - For example, a `main` function

Summary

- Software engineering
  - Techniques to facilitate development of programs
- Software life cycle
  - Phases through which software progresses, from conception to deployment to replacement to deletion
- Loop invariant
  - Property that is true before and after each iteration of a loop
- Evaluating the quality of a solution
  - Correctness, efficiency, development time, ease of use, cost of modification

Summary

- Practice abstraction: Focus on what a module does, not how
  - For data-management problems
    - Encapsulate data with operations by forming classes
  - For algorithmic tasks
    - Break into subtasks
- UML is a modeling language used to express OO designs visually

Summary

- A solution should be easy to modify
  - Modular
  - Independent of implementation of it modules
- Each function/method should be as independent as possible and perform one well-defined task
- Each function/method should include a comment: purpose, precondition, and postcondition
- A program should be as fail-safe as possible
- Effective use of available diagnostic aids is one of the keys to debugging
- Use “dump functions” to help to examine and debug the contents of data structures