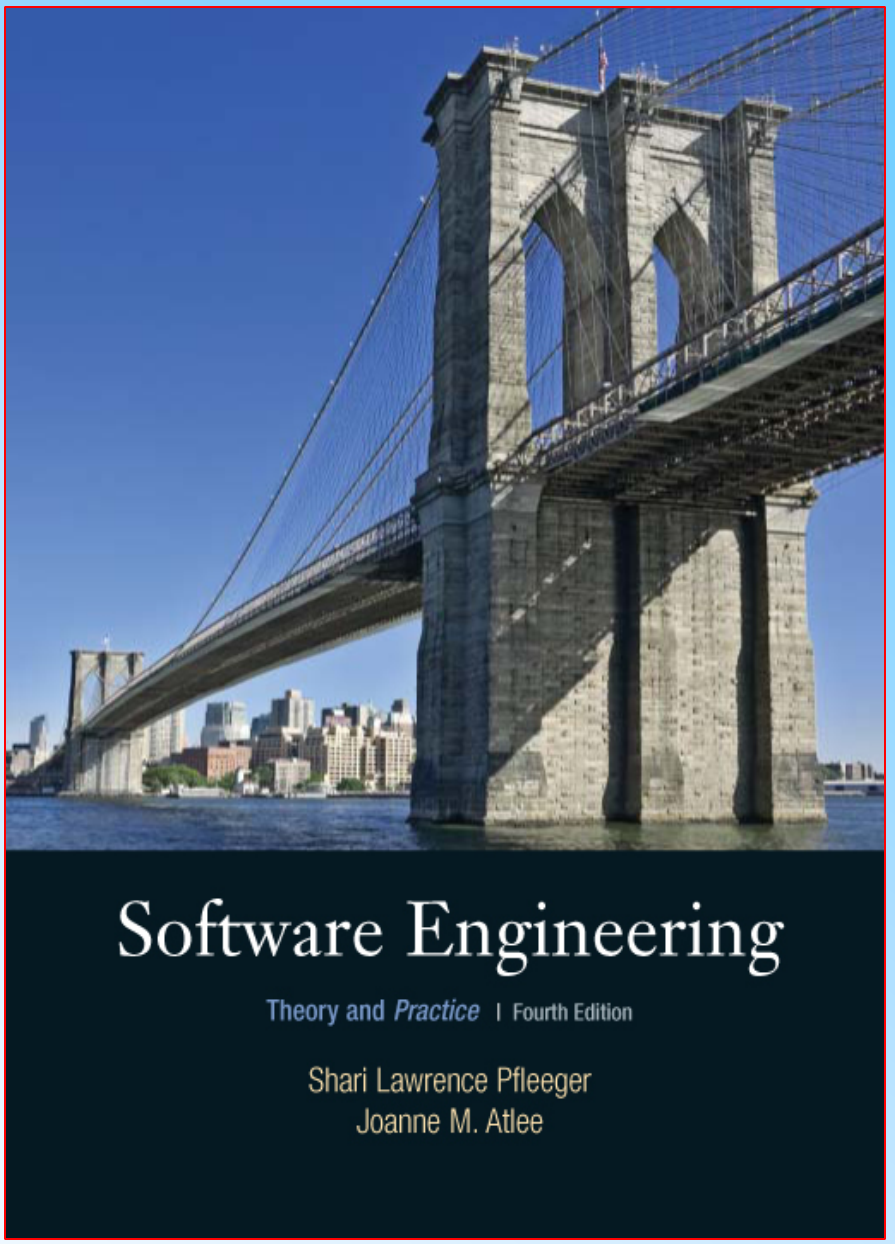
Chapter 6

Designing the Modules

Shari L. Pfleeger
Joanne M. Atlee

4th Edition



Contents

Chapter 6 Objectives

- Design principles
- Object-oriented design heuristics
- Design patterns
- Exceptions and exception handling
- Documenting designs

6.1 Design Methodology

- We have an abstract description of a solution to our customer's problem, a software architectural design, a plan for decomposing the design into software units and allocating the system's functional requirements to them
- No distinct boundary between the end of the architecture-design phase and the start of the module-design phase
- No comparable design recipes for progressing from a software unit's specification to its modular design
- The process taken towards a final solution is not as important as the documentation produced

6.1 Design Methodology
Refactoring

- Design decisions are periodically revisited and revised
- Refactoring
- Objective: to simplify complicated solutions or to optimize the design

6.2 Design Principles

- **Design principles** are guidelines for decomposing a system's required functionality and behavior into modules
- The principles identify the criteria
  - for decomposing a system
  - deciding what information to provide (and what to conceal) in the resulting modules
- Six dominant principles:
  - Modularity
  - Interfaces
  - Information hiding
  - Incremental development
  - Abstraction
  - Generality

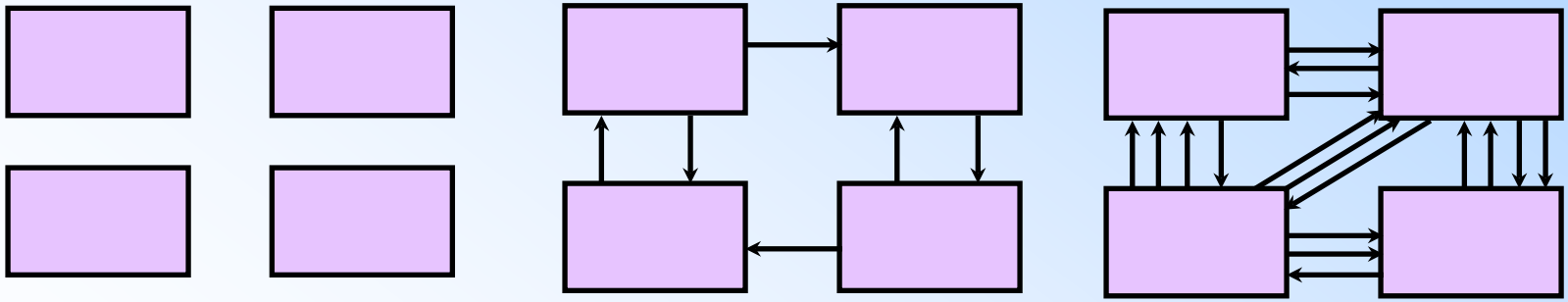
6.2 Design Principles
Modularity

- **Modularity** is the principle of keeping separate the various unrelated aspects of a system, so that each aspect can be studied in isolation (also called separation of concerns)
- If the principle is applied well, each resulting module will have a single purpose and will be relatively independent of the others
  - each module will be easy to understand and develop
  - easier to locate faults (because there are fewer suspect modules per fault)
  - Easier to change the system (because a change to one module affects relatively few other modules
- To determine how well a design separates concerns, we use two concepts that measure module independence: coupling and cohesion

6.2 Design Principles

Coupling

- Two modules are **tightly coupled** when they depend a great deal on each other
- **Loosely coupled** modules have some dependence, but their interconnections are weak
- **Uncoupled** modules have no interconnections at all; they are completely unrelated



-Uncoupled -
-no dependencies

-Loosely coupled -
-some dependencies

-Tightly coupled -
-many dependencies

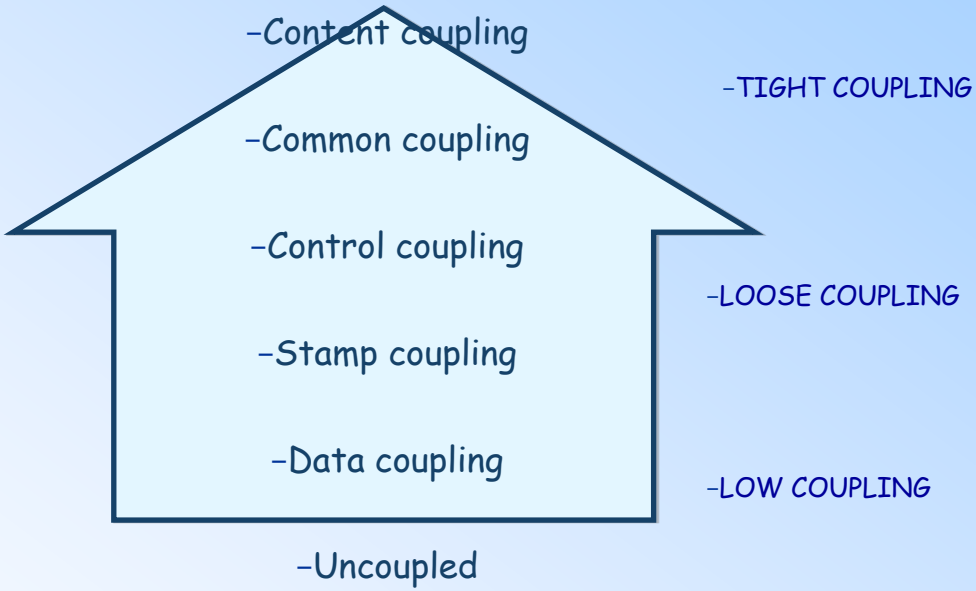
6.2 Design Principles
Coupling (continued)

- There are many ways that modules can be dependent on each other:
  - The references made from one module to another
  - The amount of data passed from one module to another
  - The amount of control that one module has over the other
- Coupling can be measured along a spectrum of dependence

6.2 Design Principles

Coupling: Types of Coupling

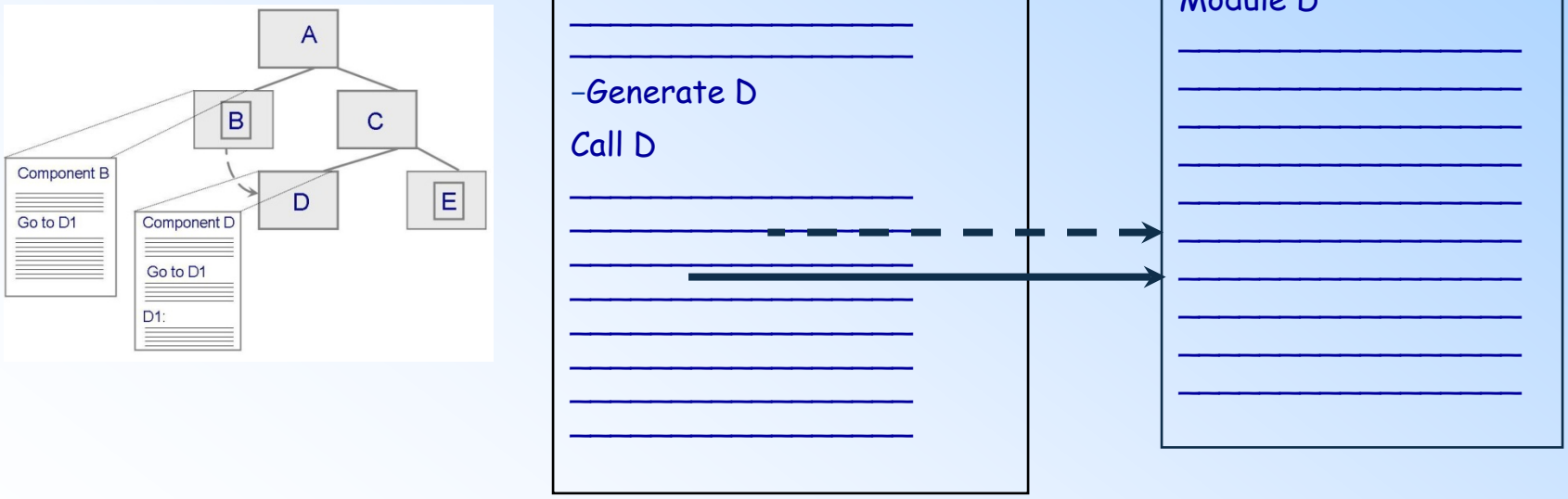
- Content coupling
- Common coupling
- Control coupling
- Stamp coupling
- Data coupling



6.2 Design Principles

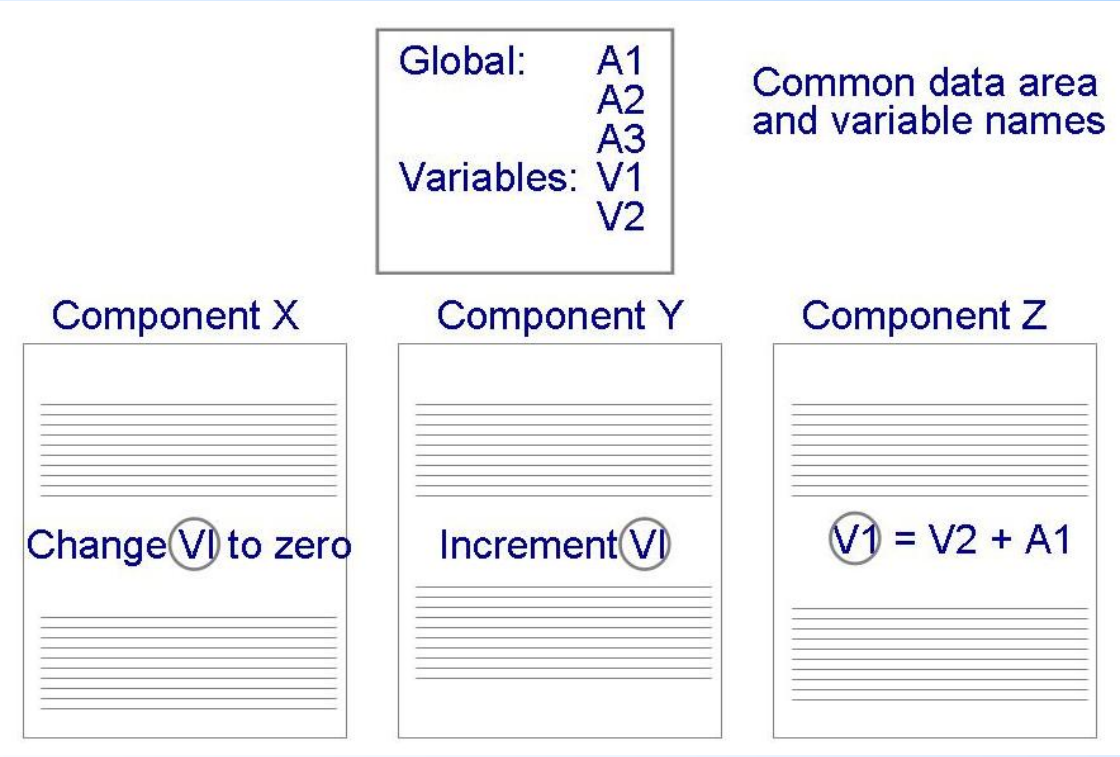
Content Coupling

- Occurs when one component modifies an internal data item in another component, or when one component branches into the middle of another component



6.2 Design Principles
Common Coupling

- Making a change to the common data means tracing back to all components that access those data to evaluate the effect of the change



6.2 Design Principles
Control Coupling

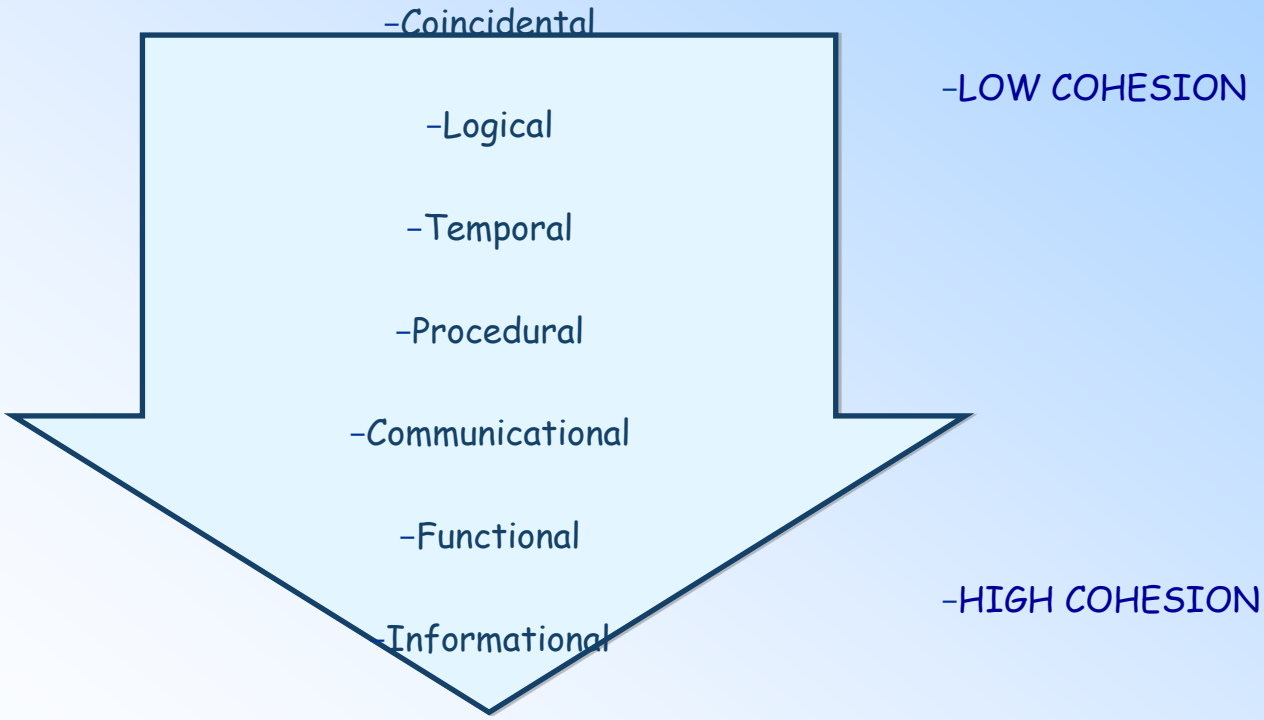
- When one module passes parameters or a return code to control the behavior of another module
- It is impossible for the controlled module to function without some direction from the controlling module

6.2 Design Principles
Stamp and Data Coupling

- Stamp coupling occurs when complex data structures are passed between modules
  - Stamp coupling represents a more complex interface between modules, because the modules have to agree on the data's format and organization
- If only data values, and not structured data, are passed, then the modules are connected by **data coupling**
  - Data coupling is simpler and less likely to be affected by changes in data representation

6.2 Design Principles
Cohesion

- **Cohesion** refers to the dependence within and among a module's internal elements (e.g., data, functions, internal modules)



6.2 Design Principles
Cohesion (continued)

- Coincidental (worst degree)
  - Parts are unrelated to one another
- Logical
  - Parts are related only by the logic structure of code
- Temporal
  - Module's data and functions related because they are used at the same time in an execution (avoid it through object constructors and destructors)
- Procedural
  - Similar to temporal, and functions pertain to some related action or purpose (module appears cohesive only in the context of its use)

6.2 Design Principles
Cohesion (continued)

- Communication
  - Operates on the same data set (the cure: place each data element in its own module)
- **Functional (ideal degree)**
  - All elements essential to a single function are contained in one module, and all of the elements are essential to the performance of the function
- Informational
  - Adaption of functional cohesion to data abstraction and object-based design

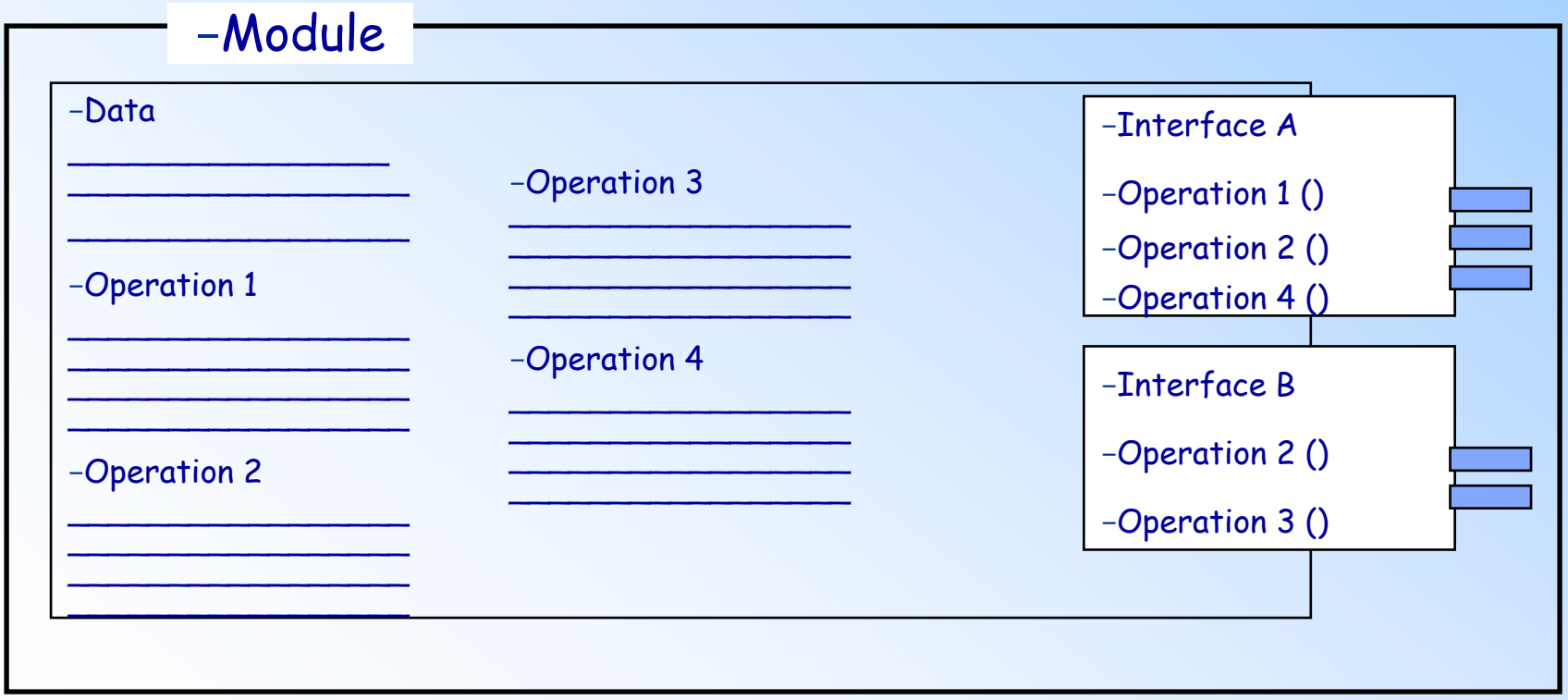
The design goal: put data, actions, or objects together only when they have one common purpose (e.g., OO design component is cohesive if all of the attributes, methods and action are strongly interdependent.)

6.2 Design Principles
Interfaces

- An **interface** defines what services the software unit provides to the rest of the system, and how other units can access those services
  - For example, the interface to an object is the collection of the object's public operations and the operations' **signatures**, which specify each operation's name, parameters, and possible return values
- An interface must also define what the unit requires, in terms of services or assumptions, for it to work correctly
- A software unit's interface describes what the unit requires of its environment, as well as what it provides to its environment

6.2 Design Principles
Interfaces (continued)

- A software unit may have several interfaces that make different demands on its environment or that offer different levels of service



6.2 Design Principles
Interfaces (continued)

- The **specification** of a software unit's interface describes the externally visible properties of the software unit
- An interface specification should communicate to other system developers everything that they need to know to use our software unit correctly
  - Purpose (functionality of each access function)
  - Preconditions (assumptions such as values of input parameters, program libraries)
  - Protocols (the order in which access function should be invoked)
  - Postconditions (visible effects such as return values, raised exceptions, changes to shared variables)
  - Quality attributes

6.2 Design Principles
Information Hiding

- **Information hiding** is distinguished by its guidance for decomposing a system:
  - Each software unit encapsulates a separate design decision (e.g., data format, operations on data, choice of algorithms) that could be changed in the future
  - Then the interfaces and interface specifications are used to describe each software unit in terms of its externally visible properties
- Using this principle, modules may show different cohesion levels
  - A module that hides a data representation may be informationally cohesive
  - A module that hides an algorithm may be functionally cohesive
  - A module that hides the sequence in which tasks are performed may be procedurally cohesive
- *A big advantage of information hiding is that the resulting software units are loosely coupled*

6.2 Design Principles

Sidebar 6.2 Information Hiding in OO Designs

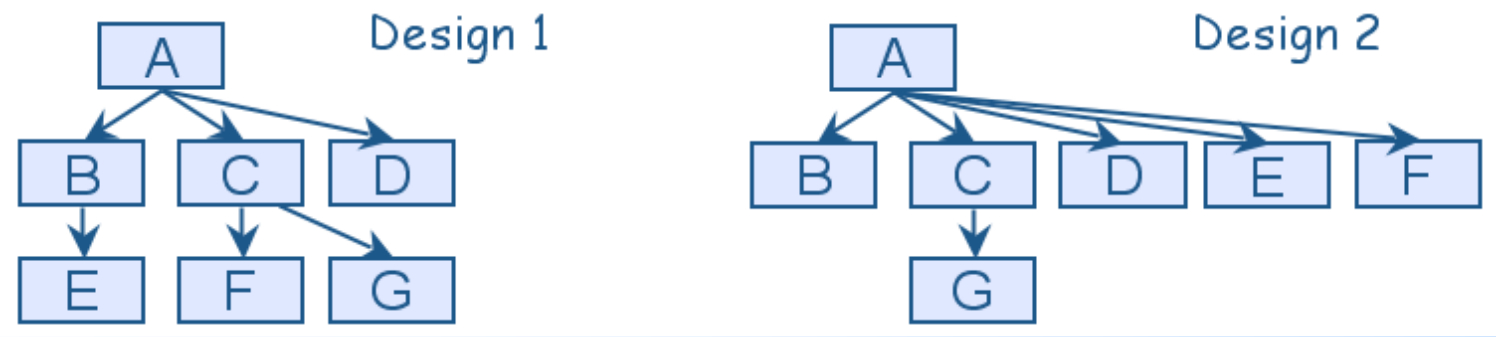
- In OO design, we decompose a system into objects and their abstract types
  - In this sense, each object hides its data representation from other objects
  - The only access that other objects have to a given object's data is via a set of access functions that the object advertises in its interface
  - This information hiding makes it easy to change an object's data representation without perturbing the rest of the system
- However, data representation is not the only type of design decision we may want to hide
  - May need to expand our notion of what an object is, to include types of information besides data types
- Objects cannot be completely uncoupled from one another, because an object needs to know the identity of the other objects so that they can interact.
  - Might mean that changing the name of an object, or the number of object instances, forces us also to change all units that invoke the object
  - Such dependence cannot be helped when accessing an object that has a distinct identity but it may be avoided when accessing an arbitrary object

6.2 Design Principles
Incremental Development

- Given a design consisting of software units and their interfaces, we can use the information about the units' dependencies to devise an incremental schedule of development
- Start by mapping out the units' **uses relation**
  - relates each software unit to the other software units on which it depends
- **Uses graphs** can help to identify progressively larger subsets of our system that we can implement and test incrementally

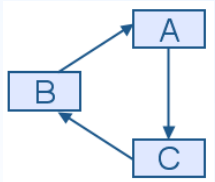
6.2 Design Principles
Incremental Development (continued)

- Uses graphs for two designs
  - **Fan-in** refers to the number of units that use a particular software unit (high fan-in)
  - **Fan-out** refers to the number of units used by particular software unit (low fan-out)

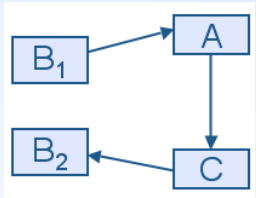


6.2 Design Principles
Incremental Development (continued)

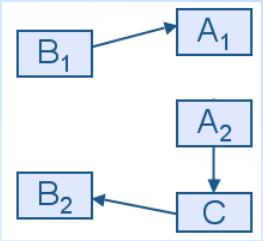
- We can try to break a cycle in the uses graph using a technique called **sandwiching**
  - One of the cycle's units is decomposed into two units, such that one of the new units has no dependencies
  - Sandwiching can be applied more than once, to break either mutual dependencies in tightly coupled units or long dependency chains



–(α)



–(β)



–(χ)

6.2 Design Principles
Abstraction

- An **abstraction** is a model or representation that omits some details so that it can focus on other details
- The definition is vague about which details are left out of a model, because different abstractions, built for different purposes, omit different kinds of details

6.2 Design Principles
Sidebar 6.3 Using Abstraction

- Suppose that one of the system functions is to sort the elements of a list L. The initial description of the design is:

  Sort L in nondecreasing order

  The next level of abstraction may be a particular algorithm:

```
DO WHILE I is between 1 and (length of L)-1:
 Set LOW to index of smallest value in L(I),...,
   L(length of L)
 Interchange L(I) and L(LOW)
ENDDO
```

- The algorithm provides a great deal of additional information, however, it can be made even more detailed

6.2 Design Principles
Sidebar 6.3 Using Abstraction (continued)

- The third and final algorithm describes exactly how the sorting operation will work:

```
DO WHILE I is between 1 and (length of L)-1
 Set LOW to current value of I
 DO WHILE J is between I+1 and (length of L)
    IF L(LOW) is greater than L(J)
       THEN set LOW to current value of J
    ENDIF
 ENDDO
 Set TEMP to L(LOW)
 Set L(LOW) to L(I)
 Set L(I) to TEMP
ENDDO
```

6.2 Design Principles
Generality

- **Generality** is the design principle that makes a software unit as universally applicable as possible, to increase the chance that it will be useful in some future system
- We make a unit more general by increasing the number of contexts in which can it be used. There are several ways of doing this:
  - Parameterizing context-specific information
  - Removing preconditions
  - Simplifying postconditions

6.2 Design Principles
Generality (continued)

- The following four procedure interfaces are listed in order of increasing generality:

```
PROCEDURE SUM: INTEGER;
POSTCONDITION: returns sum of 3 global variables

PROCEDURE SUM (a, b, c: INTEGER): INTEGER;
POSTCONDITION: returns sum of parameters

PROCEDURE SUM (a[]: INTEGER; len: INTEGER): INTEGER
PRECONDITION: 0 <= len <= size of array a
POSTCONDITION: returns sum of elements 1..len in array a

PROCEDURE SUM (a[]: INTEGER): INTEGER
POSTCONDITION: returns sum of elements in array a
```

6.3 OO Design

- Object oriented methodologies are the most popular and sophisticated design methodologies
- A design is **object oriented** if it decomposes a system into a collection of runtime components called objects that encapsulate data and functionality
  - Objects are uniquely **identifiable** runtime entities that can be designated as the target of a message or request
  - Objects can be **composed**, in that an object's data variables may themselves be objects, thereby encapsulating the implementations of the object's internal variables
  - The implementation of an object can be reused and extended via **inheritance**, to define the implementation of other objects
  - OO code can be **polymorphic:** written in generic code that works with objects of different but related types

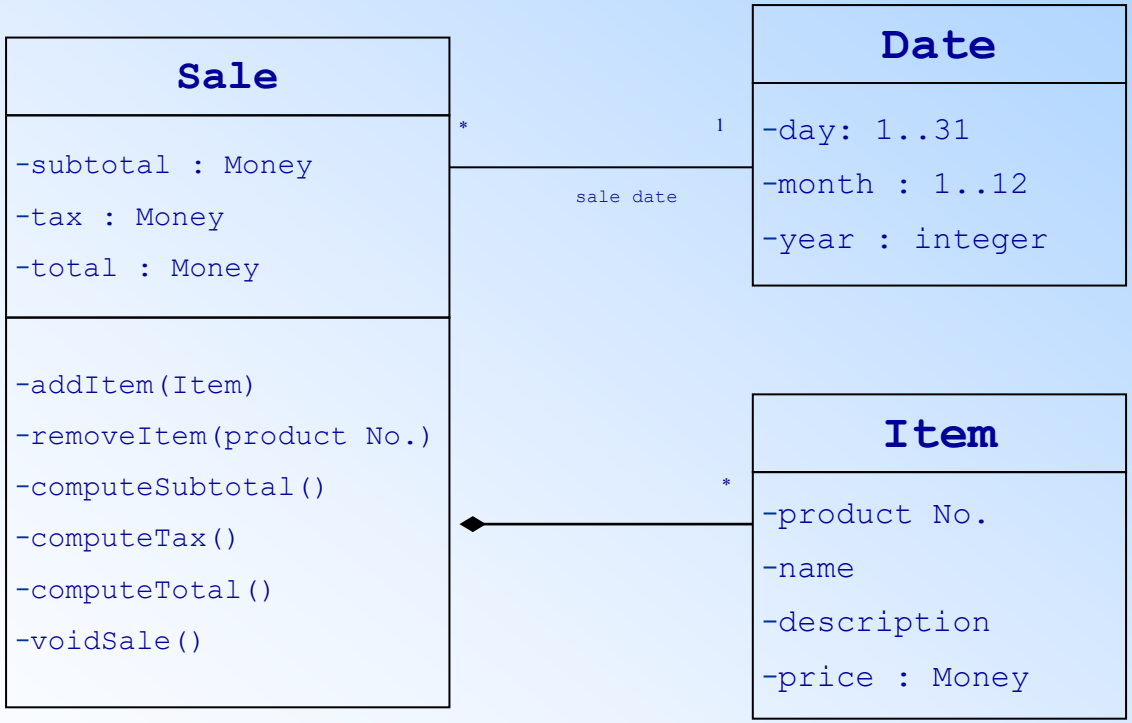
6.3 OO Design
Terminology

- A **class** is a software module that partially or totally implements an abstract data type
- If a class is missing implementations for some of its methods, we say that it is an **abstract class**
- The class definition includes **constructor** methods that spawn new object instances
- **Instance variables** are program variables whose values are references to objects

6.3 OO Design
Terminology (continued)

- The runtime structure of an OO system is a set of **objects**, each of which is a cohesive collection of data plus all operations for creating, reading, altering, and destroying those data
- An object's data are called **attributes**, and its operations are called **methods**
- An object may have multiple interfaces, each offering a different level of access to the object's data and methods
  - Such interfaces are hierarchically related by type: if one interface offers a strict subset of the services that another interface offers, we say that the first interface is a **subtype** of the second interface (the **supertype**)

6.3 OO Design
Terminology (continued)

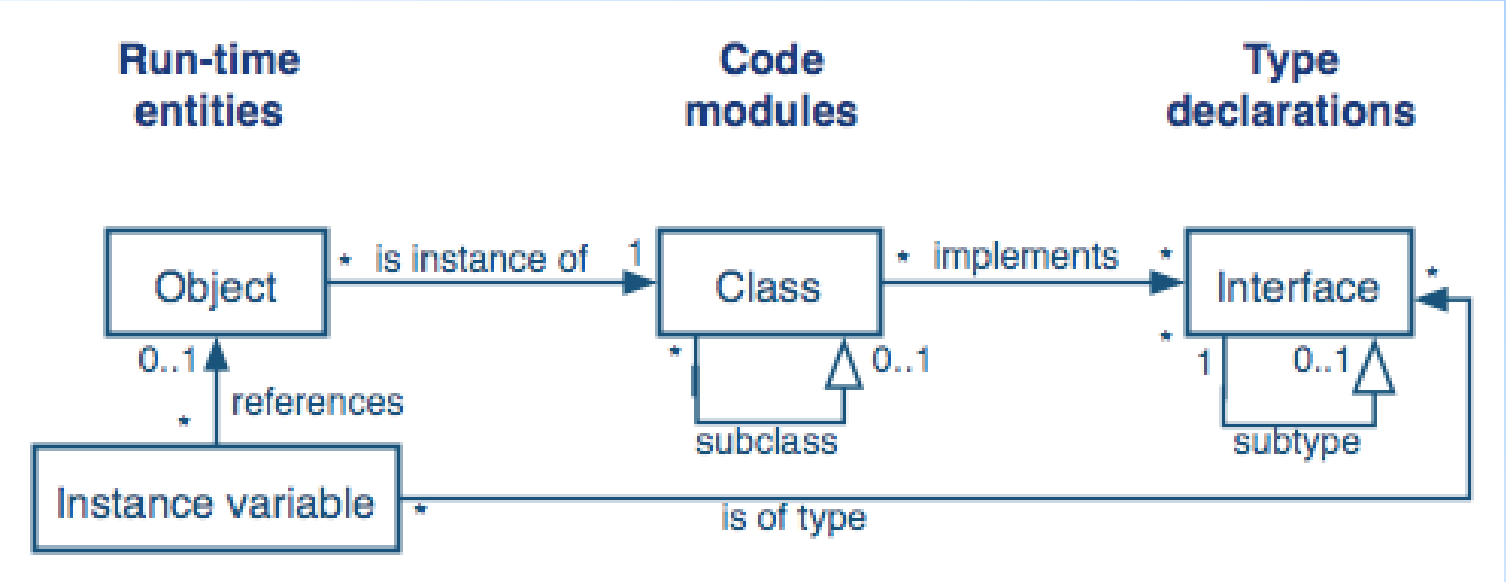


6.3 OO Design
Terminology (continued)

- Variables can refer to objects of different classes over the course of a program's execution, known as **dynamic binding**
- The directed arrows in the figure below depict the relationships between constructs, and the adornments at the ends of each arrow indicate the **multiplicity** (how many of an item may exist)

6.3 OO Design
Terminology (continued)

- Four OO constructs: *classes, objects, interfaces*, and *instance variables*

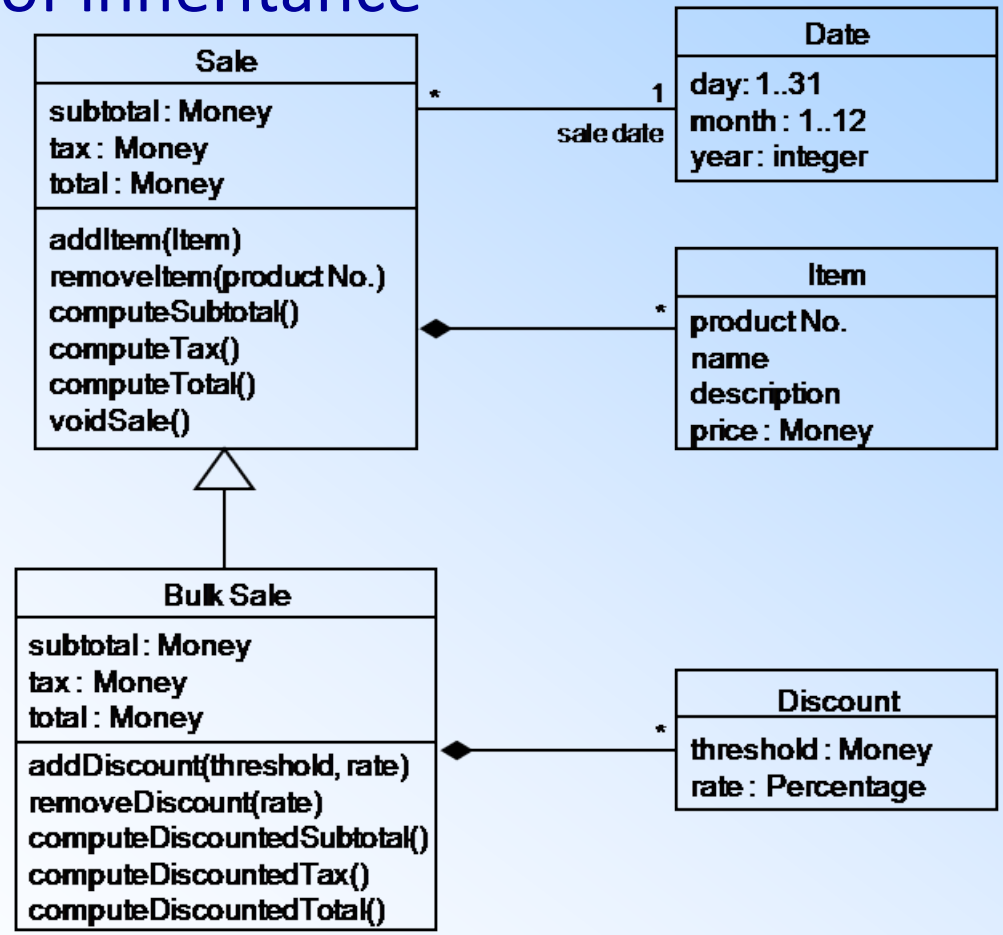


6.3 OO Design
Terminology (continued)

- Building new classes by combining component classes, much as children build structures from building blocks is done by **object composition**
- Alternatively, we can build new classes by extending or modifying definitions of existing classes
  - This kind of construction, called **inheritance**, defines a new class by directly reusing (and adding to) the definitions of an existing class

6.3 OO Design
Terminology (continued)

- Example of inheritance



6.3 OO Design
Terminology (continued)

- **Polymorphism** occurs when code is written in terms of interactions with an interface, but code behavior depends on the object associated with the interface at runtime and on the implementations of that object's method
- Inheritance, object composition, and polymorphism are important features of an OO design that make the resulting system more useful in many ways

6.3 OO Design

Inheritance vs. Object Composition

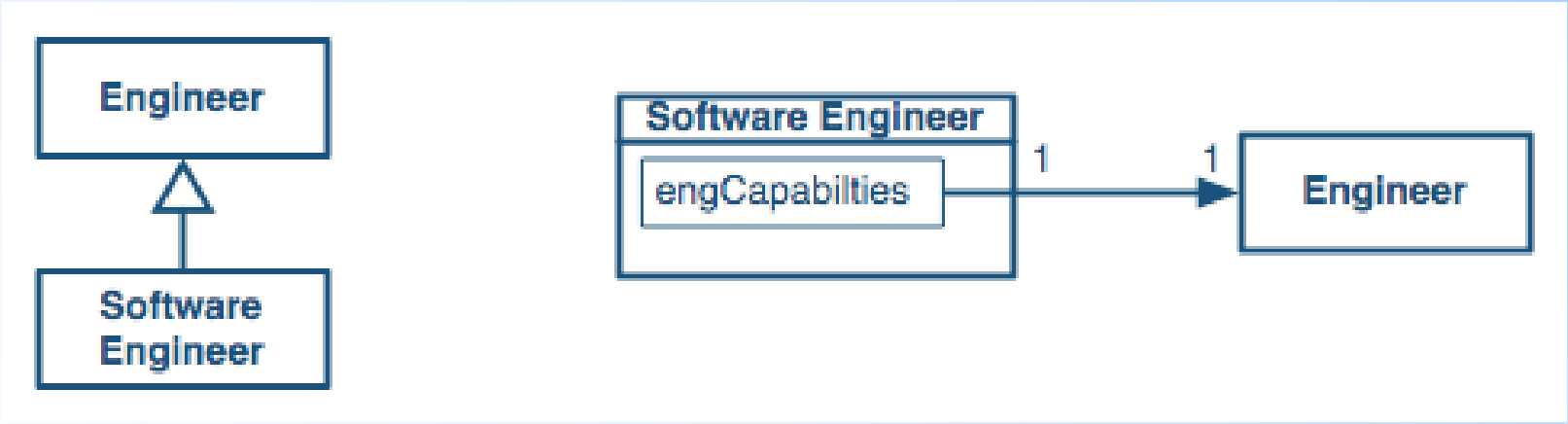
- A key design decision is determining how best to structure and relate complex objects
- In an OO system, there are two main techniques for constructing large objects
  - Inheritance
  - composition
- A new class can be created by extending and overriding the behavior of an existing class, or it can be created by combining simpler classes to form a composite class.

6.3 OO Design
Inheritance vs. Object Composition (continued)

- Each construction paradigm has advantages and disadvantages
- Composition is better than inheritance at preserving the encapsulation of the reused code, because a composite object accesses the component only through its advertised interface
- By contrast, using the inheritance approach, the subclass's implementation is determined at design time and is static
- The resulting objects are less flexible than objects instantiated from composite classes because the methods they inherit from their parent class cannot be changed at runtime
- The greatest advantage of inheritance is the ability to change and specialize the behaviors of inherited methods, by selectively overriding inherited definitions

6.3 OO Design

Inheritance vs. Object Composition (continued)



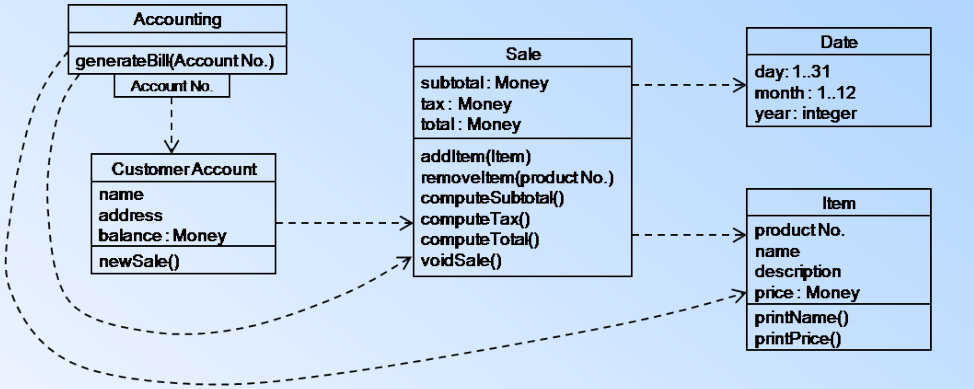
6.3 OO Design
Law of Demeter

- **Law of Demeter:** Allows reducing dependencies by including in each composite class methods for operating on the class's components
- Benefit: client code that uses a composite class needs to know only about the composite itself and not about the composites' components
- Designs that obey the Law of Demeter have fewer class dependencies, and classes with fewer dependencies tend to have fewer software faults

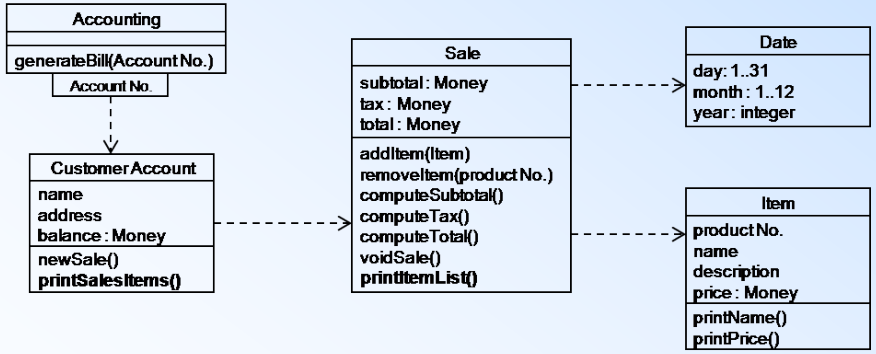
6.3 OO Design

Law of Demeter (continued)

Design 1

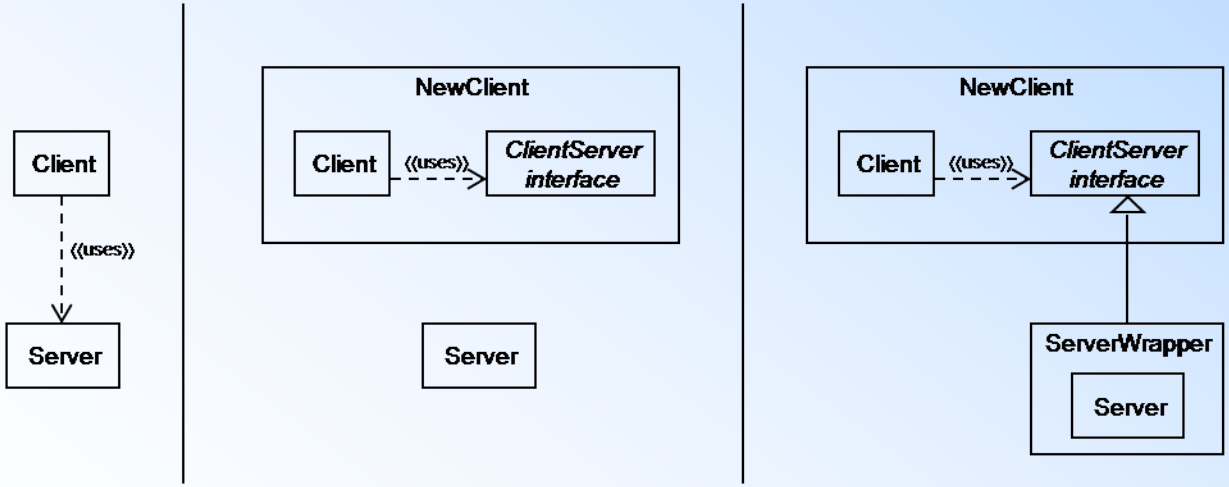


Design 2



6.3 OO Design
Dependency Inversion

- Used to reverse the direction of a dependency link between two classes
- Dependency inversion works by introducing interfaces
  - Create an interface that the client can depend (include specifications of all the methods the client expects from the server class).
  - Package together the original client class and the new interface into a new client module
  - Create a wrapper class for the server
  - Advantage: client and server code depend only on the new ClientServerInterface (increases maintainability)



6.4 Representing OO Designs in the UML

- The UML is a suite of design notations that is popular for describing OO solutions
- The UML can be used to visualize, specify, or document a software design
- UML especially useful for describing different design alternatives, and eventually for documenting design artifacts

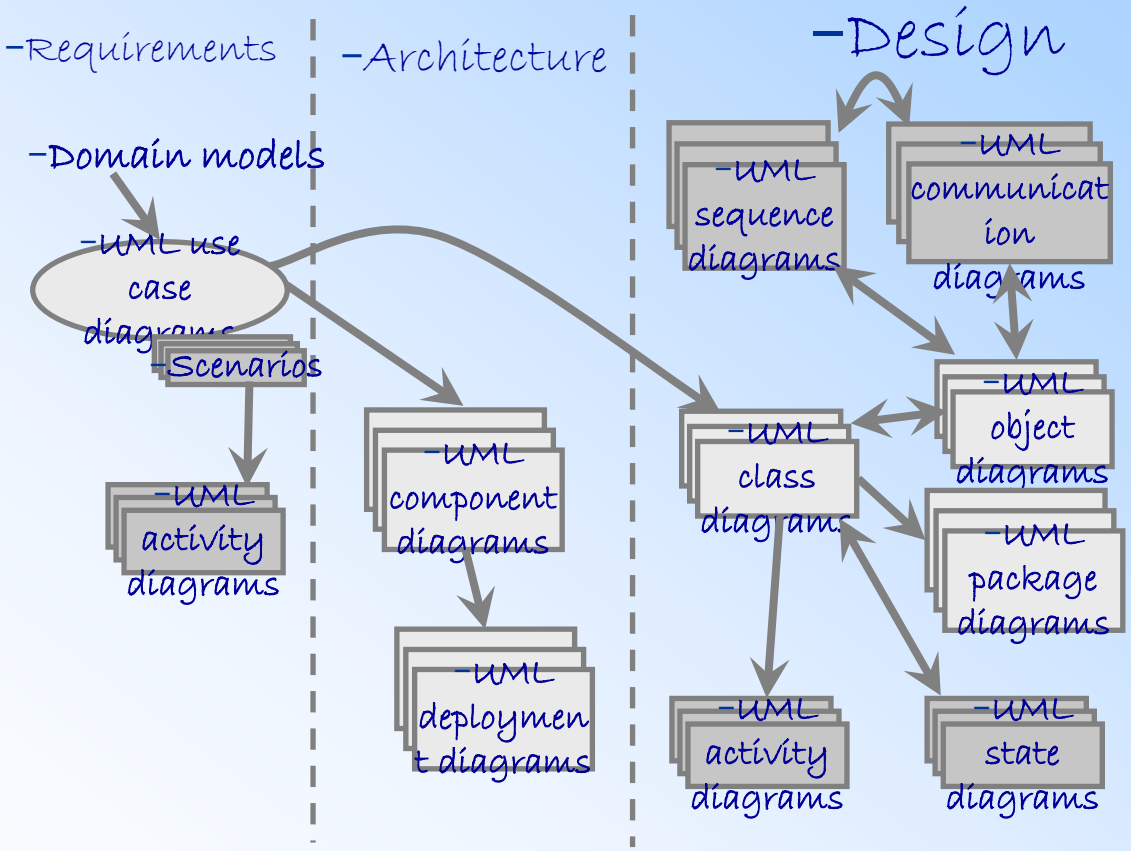
6.4 Representing OO Designs in the UML

UML in the Process

- Use case diagrams
- UML activity diagrams
- Domain model
- Component diagrams
- Deployment diagrams
- Class diagrams
- Interaction diagrams
- Sequence diagrams
- Communication diagrams
- Activity diagrams
- State diagrams
- Package diagrams

6.4 Representing OO Designs in the UML
UML in the Process (continued)

- How UML is used in the development process



6.4 Representing OO Designs in the UML

Sidebar 6.4 Royal Service Station Requirements

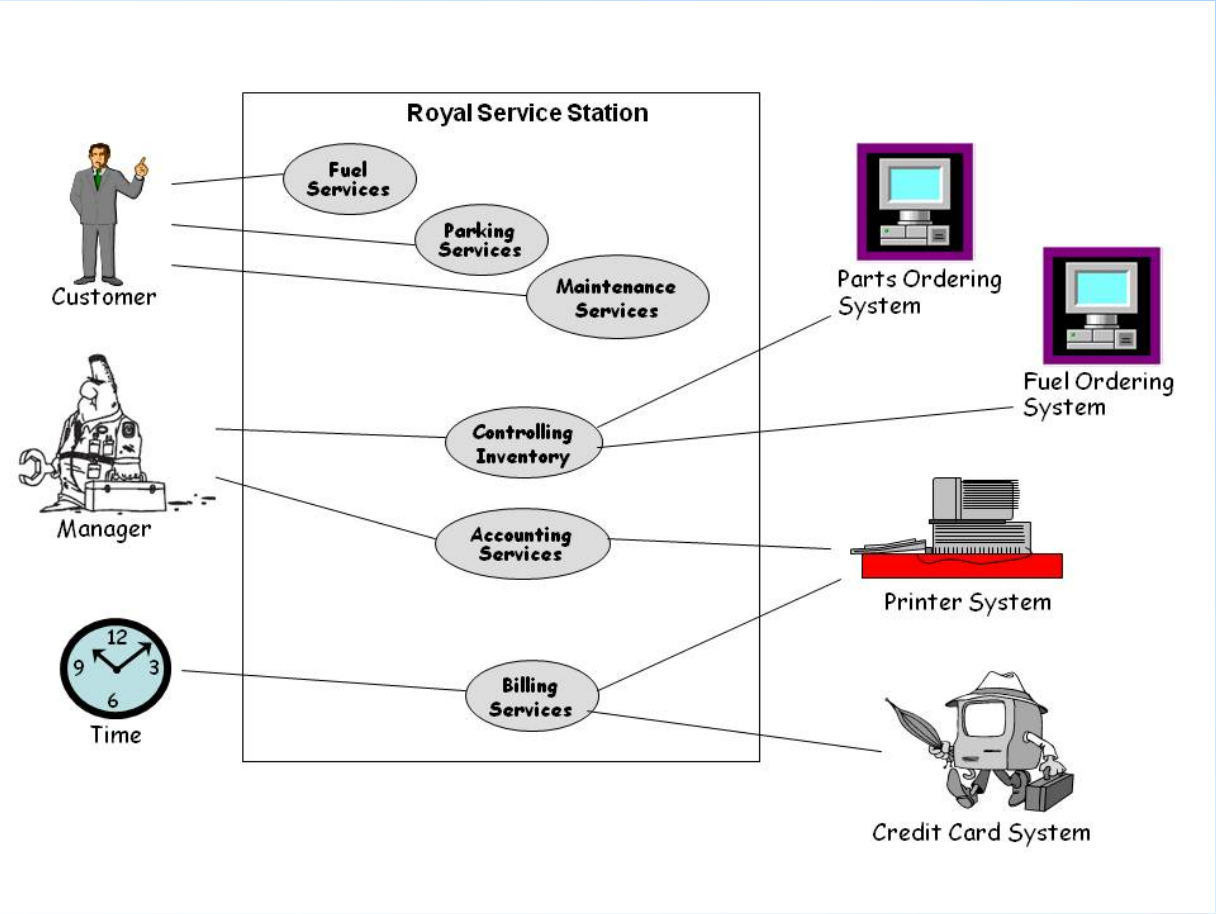
- Royal Service station provides three types of services
- The system must track bills, the product and services
- System to control inventory
- The system to track credit history, and payments overdue
- The system applies only to regular repeat customer
- The system must handle the data requirements for interfacing with other system
- The system must record tax and related information
- The station must be able to review tax record upon demand
- The system will send periodic message to customers
- Customer can rent parking space in the station parking lot
- The system maintain a repository of account information
- The station manager must be able to review accounting information upon demand
- The system can report an analysis of prices and discounts
- The system will automatically notify the owners of dormant accounts
- The system can not be unavailable for more than 24 hours
- The system must protect customer information from unauthorized access

6.4 Representing OO Designs in the UML
UML Class Diagram

- UML class diagrams describe the object types and their static relationships
  - Depict associations among objects and relationships between types and subtypes
  - Diagrams should illustrate the attributes of each object, their individual behaviors, and the restrictions on each class or object
- Look for and seek
  - Actors, physical objects, places, organizations, records, transactions, collections of things, operations procedures, things manipulated by the system to be built

6.4 Representing OO Designs in the UML
UML Class Diagram (continued)

- Royal Service Station use case diagram



6.4 Representing OO Designs in the UML
UML Class Diagram (continued)

- What needs to be "processed" in some way?
- What items have multiple attributes?
- When do you have more than one object in a class?
- What is based on the requirements themselves, not derived from your understanding of the requirements?
- What attributes and operations are always applicable to a class or object?

6.4 Representing OO Designs in the UML

Initial Grouping of Attributes and Classes: Step 1

Attributes	Classes
Personal check	Customer
Tax	Maintenance
Price	Services
Cash	Fuel
Credit card	Bill
Discounts	Purchase
	Station manager

6.4 Representing OO Designs in the UML

Initial Grouping of Attributes and Classes: Step 2

Attributes	Classes
Personal check	Customer
Tax	Maintenance
Price	Services
Cash	Parking
Credit card	Fuel
Discounts	Bill
Name	Purchase
Address	Maintenance reminder
Birthdate	Station manager

6.4 Representing OO Designs in the UML

Guidelines for Identifying Behaviors

- Imperative verbs
- Passive verbs
- Actions
- Membership in
- Management or ownership
- Responsible for
- Services provided by an organization

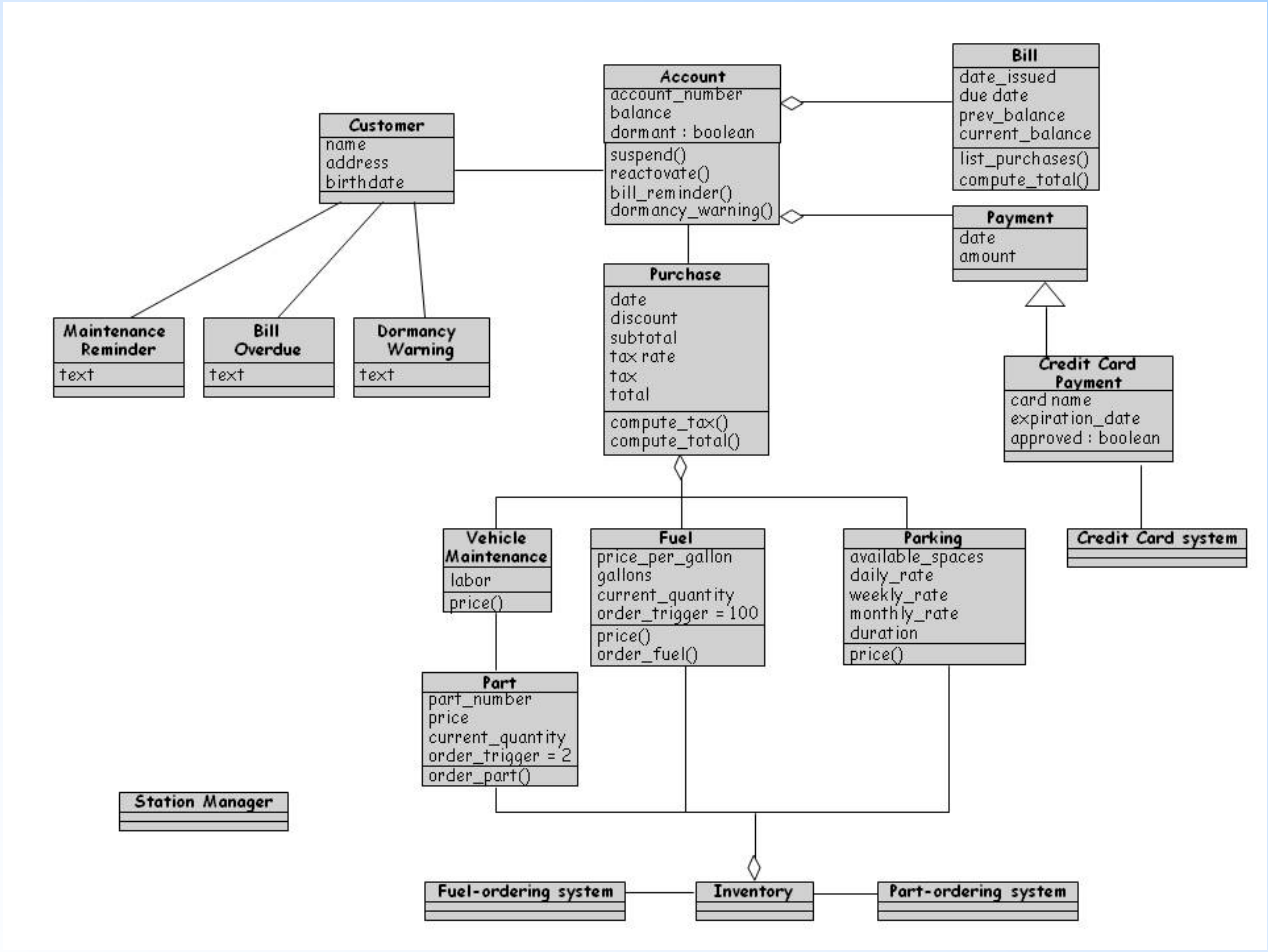
6.4 Representing OO Designs in the UML

Initial Grouping of Attributes and Classes: Step 3

Attributes	Classes
Personal check	Customer
Tax	Maintenance
Price	Services
Cash	Parking
Credit card	Fuel
Discounts	Bill
Name	Purchase
Address	Maintenance reminder
Birthdate	Station manager
	Overdue bill letter
	Dormant account warning
	Parts
	Accounts
	Inventory
	Credit card system
	Part ordering system
	Fuel ordering system

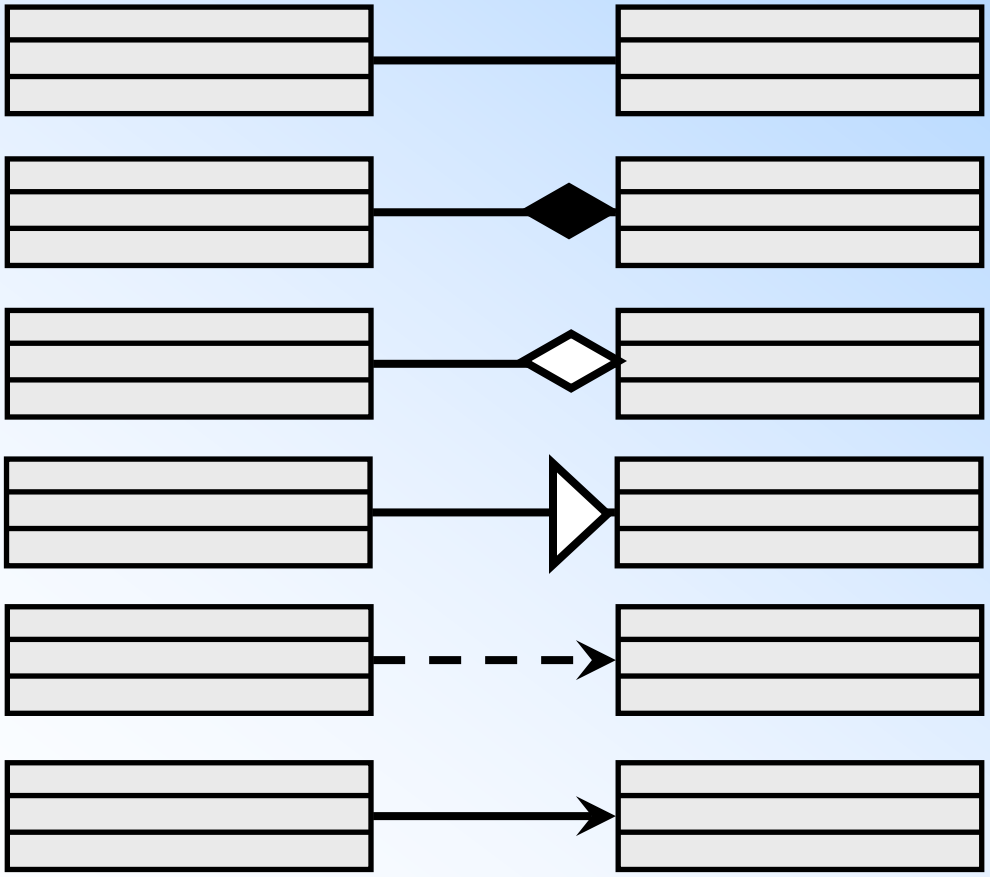
6.4 Representing OO Designs in the UML

First Cut at Royal Service Station Design



6.4 Representing OO Designs in the UML

Types of Class Relationships



- association
- composition
- aggregation
- generalization
- dependency
- navigation

6.4 Representing OO Designs in the UML
Other UML Diagrams

- Class description template
- Package diagrams
- Interaction diagrams
- Sequence diagrams
- Communication diagrams
- State diagrams
- Activity diagrams

6.4 Representing OO Designs in the UML

Other UML Diagrams – Class Description Template

```
Class name: Refuel
        Category: service
        External documents:
        Export control: Public
        Cardinality: n
        Hierarchy:
                Superclasses: Service
        Associations:
                <no rolename>: fuel in association updates
Operation name: price
        Public member of: Refuel
        Documentation:
                // Calculates fuel final price
        Preconditions:
                gallons > 0
                Object diagram: (unspecified)
```

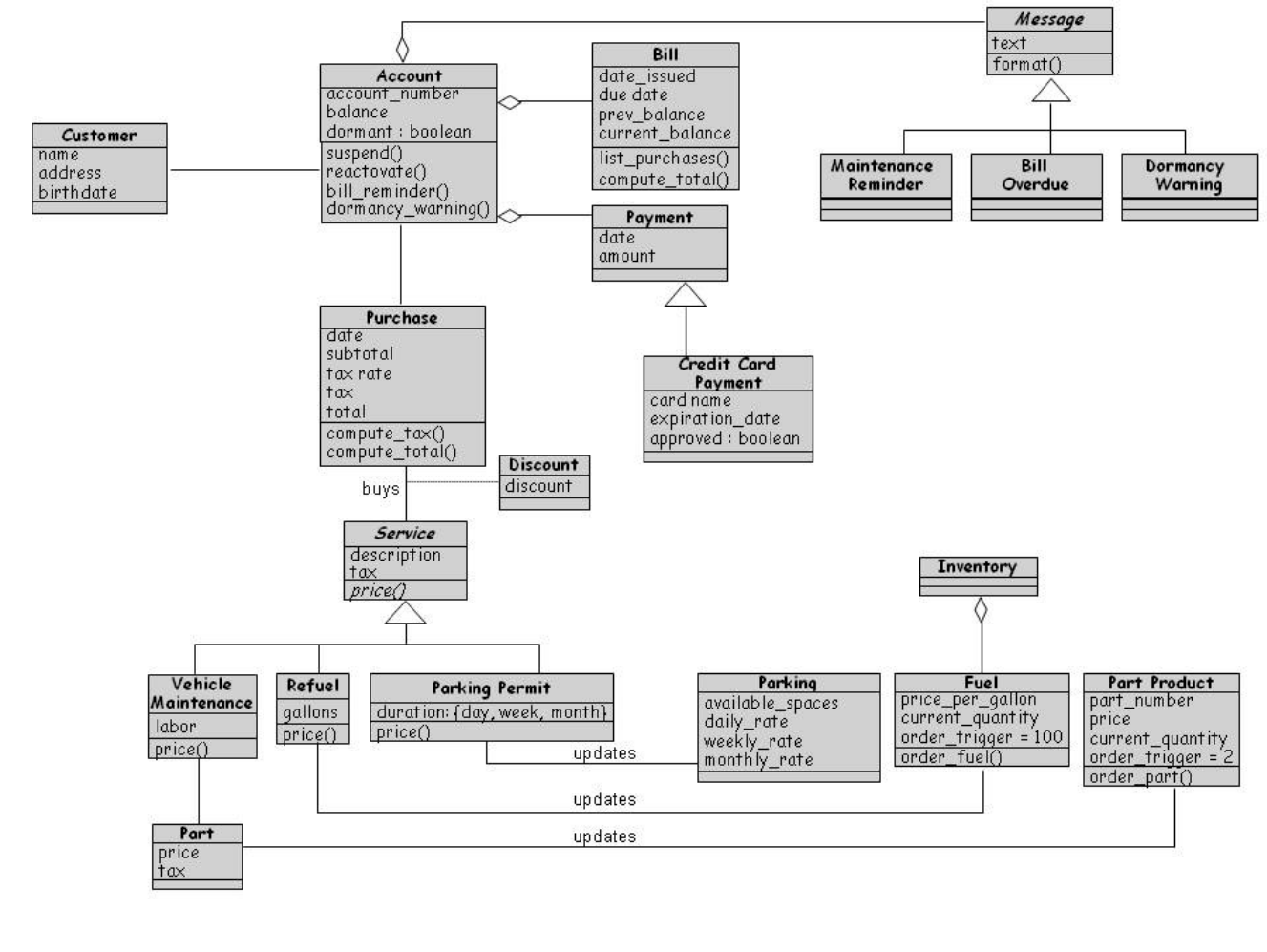
6.4 Representing OO Designs in the UML

Other UML Diagrams – Class Description Template (cont)

```
        Semantics:
                price = gallons * fuel.price_per_gallon
                tax = price * purchase.tax_rate
                Object diagram: (unspecified)
        Concurrency: sequential
Public interface:
        Operations:
                price
Private interface:
        Attributes:
                gallons
Implementation:
        Attributes:
                gallons
State machine: no
Concurrency: sequential
Persistence: transient
```

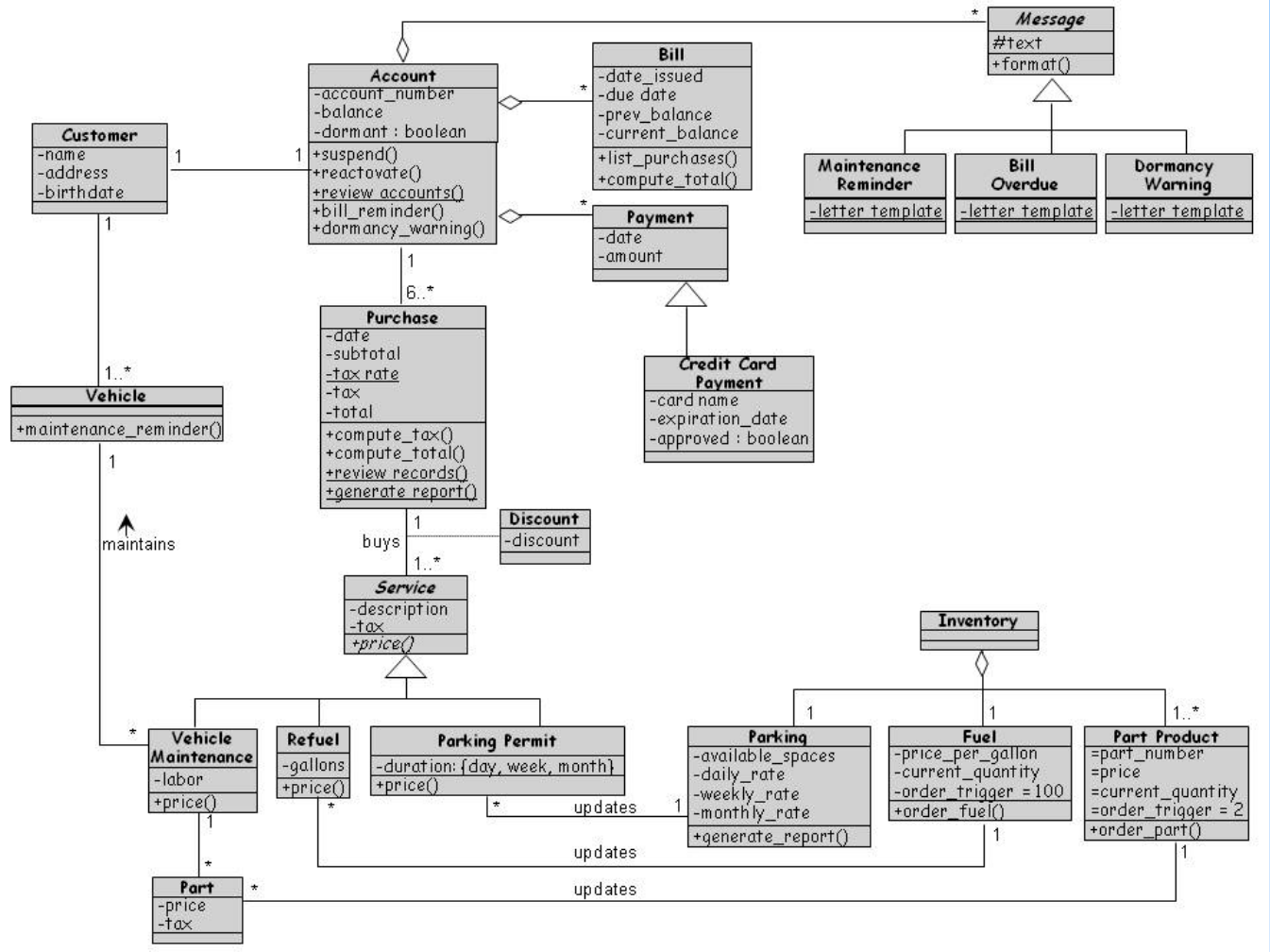
6.4 Representing OO Designs in the UML

Second Cut at Royal Service Station Design



6.4 Representing OO Designs in the UML

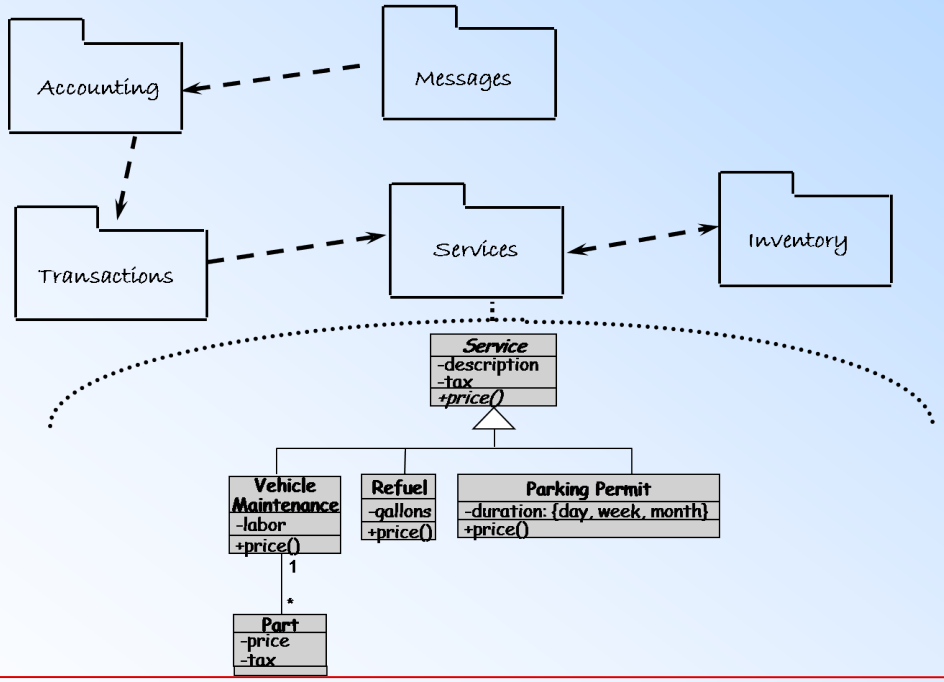
Final Cut at Royal Service Station Design



6.4 Representing OO Designs in the UML

Other UML Diagrams – Package Diagram

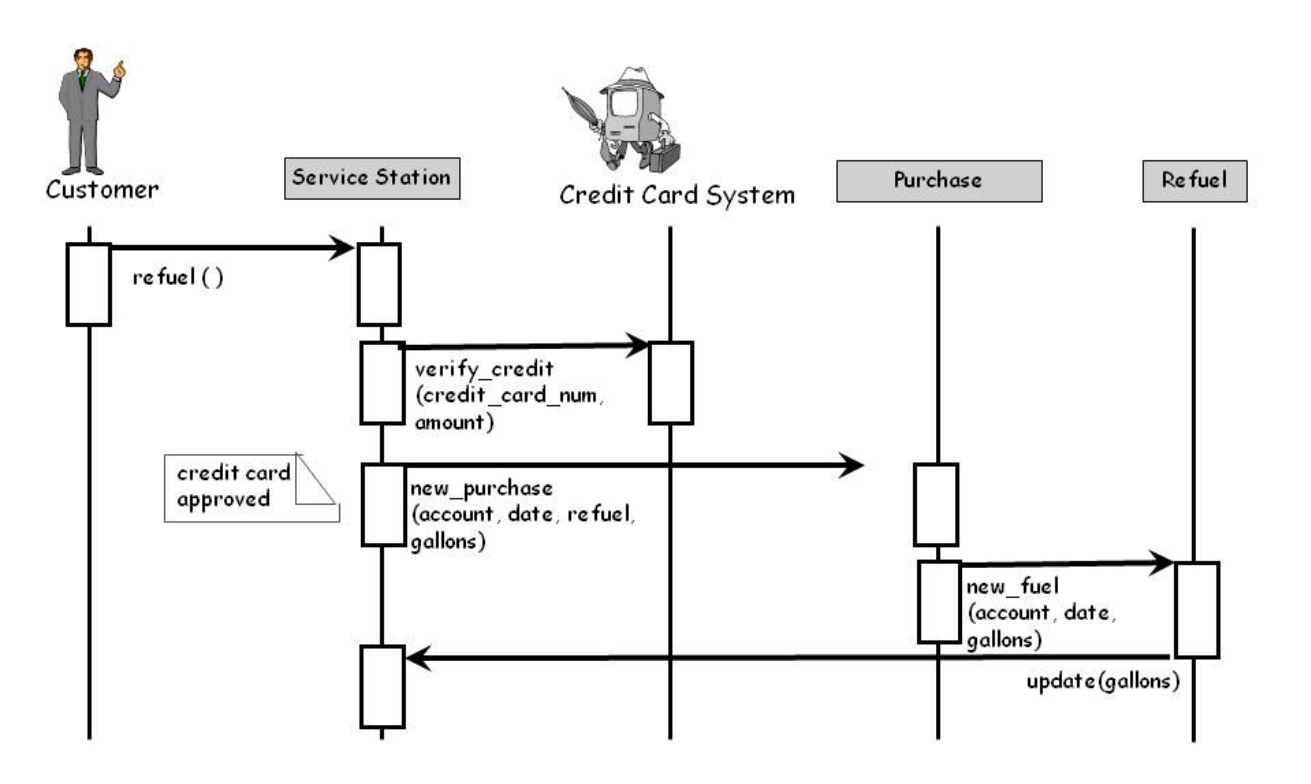
- UML package diagrams allow viewing a system as a small collection of packages each of which may be expanded to a larger set of classes
- Shows the **dependencies** among classes that belong to different packages – important during testing



6.4 Representing OO Designs in the UML

Other UML Diagrams – Sequence Diagram

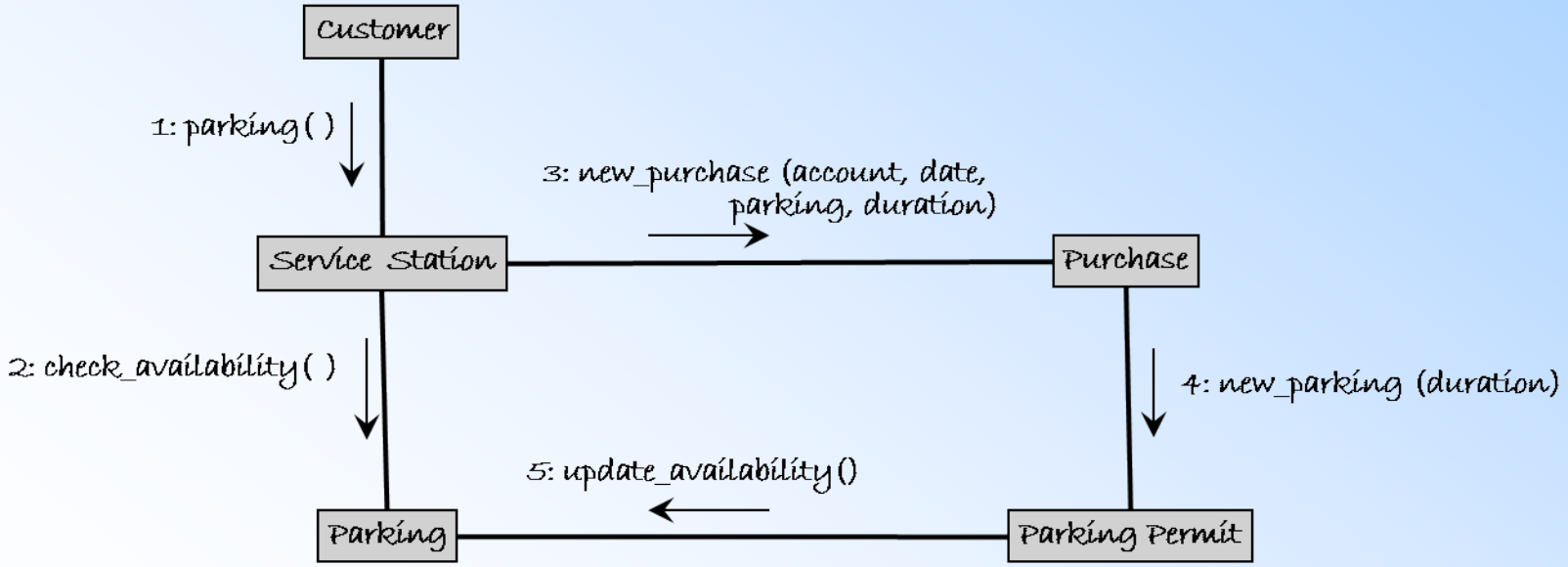
- Shows the sequence of activities or behavior occur (e.g., example of Refuel class of Royal Service Station)



6.4 Representing OO Designs in the UML

Other UML Diagrams – Communication Diagram

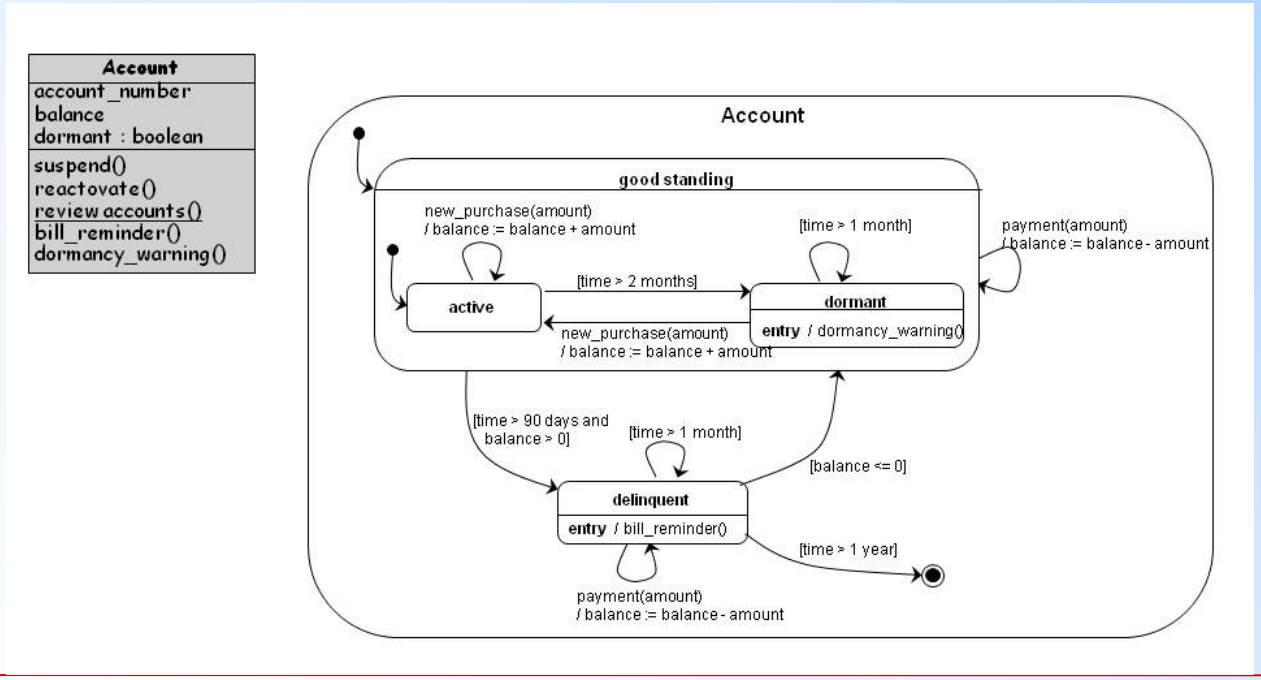
- A communication diagram depicts a sequence of messages between objects, but it is superimposed on an object and uses the links between object as implicit communication channels



6.4 Representing OO Designs in the UML

Other UML Diagrams – StateDiagram

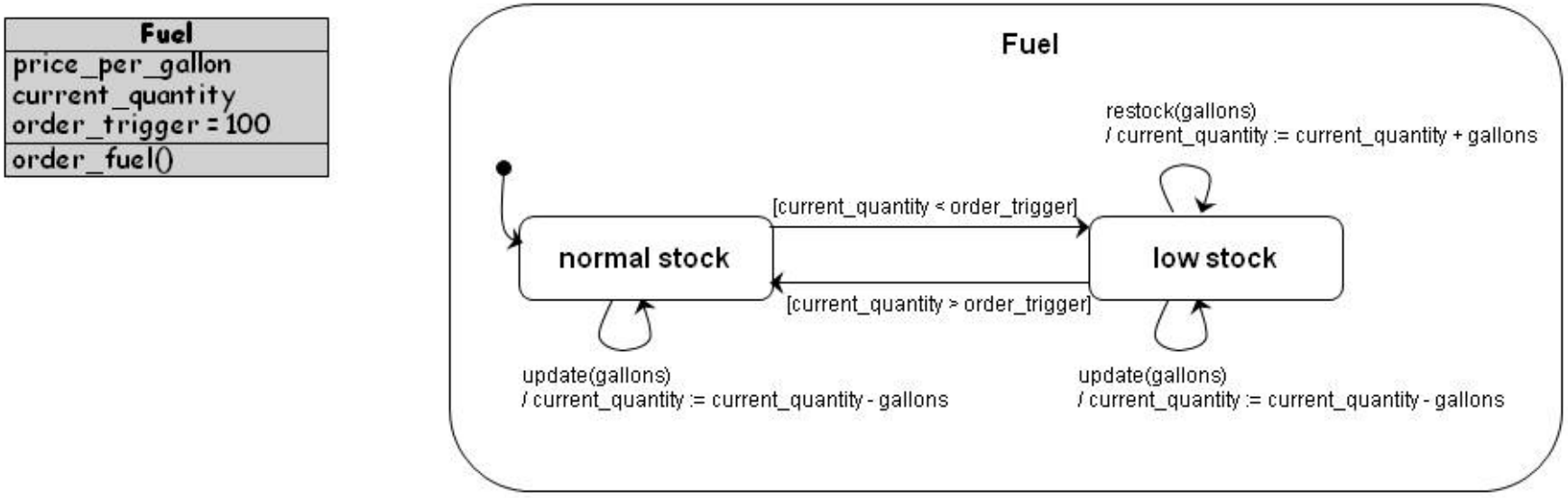
- A state diagram shows the possible states an object can take, the events that trigger the transition between one state to the next, and the actions that result from each state change



6.4 Representing OO Designs in the UML

Other UML Diagrams – State Diagram (continued)

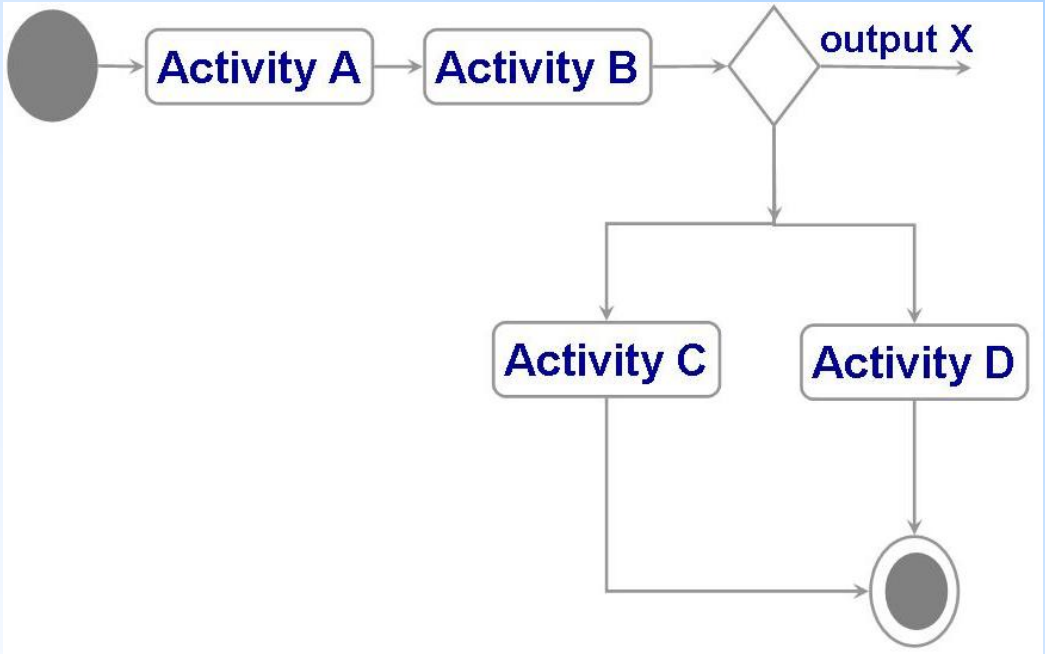
State diagram for Fuel class in the Royal Service Station



6.4 Representing OO Designs in the UML

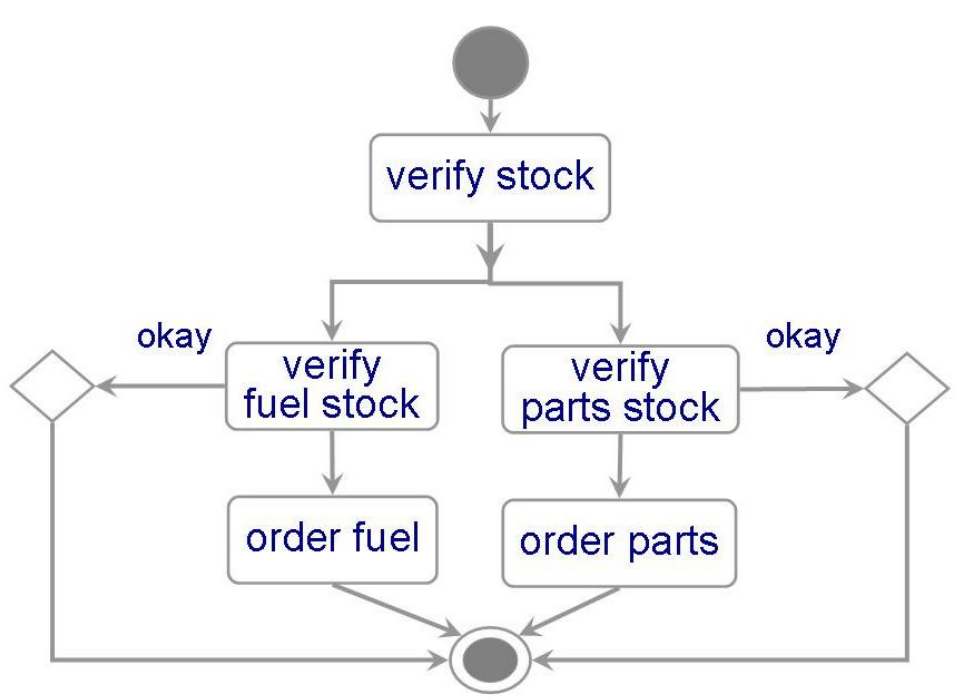
Other UML Diagrams – Activity Diagram

- Activity diagrams are used to model the flow of procedures or activities in a class
- A decision node is used to represent a choice of which activity to invoke



6.4 Representing OO Designs in the UML
Other UML Diagrams – Activity Diagram (continued)

- Activity diagrams are used to model the flow of procedures or activities in a class
- An activity diagram for the *inventory* class
- It may have two decisions
  - to verify that there are enough fuel
  - to verify that a part is in stock



6.5 OO Design Patterns

- A design pattern codifies design decisions and best practices for solving a particular design problem according to design principles
- Design patterns are not the same as software libraries; they are not packaged solutions that can be used as is. Rather, they are templates for a solution that must be modified and adapted for each particular use
- Design patterns provide more specific guidance than design principles do, but they are less detailed than software libraries
- *The main goal is to improve a design's modularity*
- Added complexity (e.g., extra classes, associations) improves modularity at the expense of quality parameters (e.g., performance, ease of development)

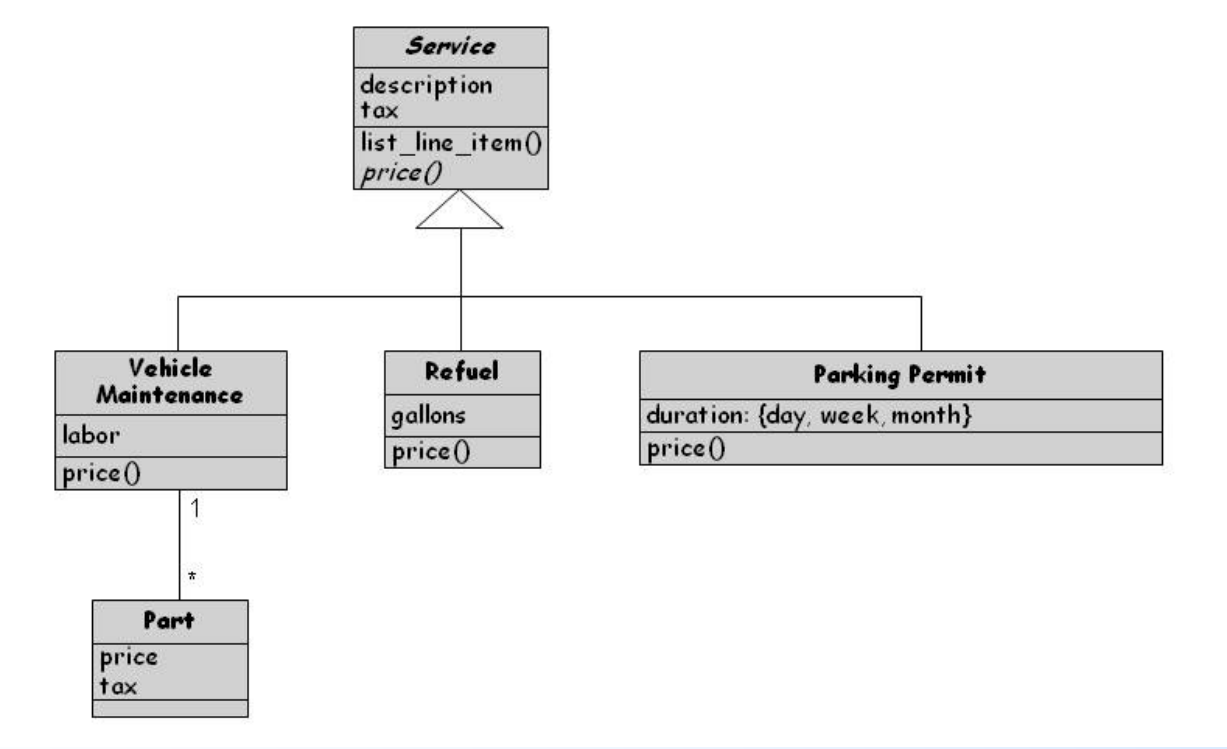
6.5 OO Design Patterns
Template Method Pattern

- The Template Method pattern aims to reduce the amount of duplicate code among subclasses of the same parent class
  - It is particularly useful when multiple subclasses have similar but not identical implementations of the same method
  - This pattern addresses this problem by localizing the duplicate code structure in an abstract class from which the subclasses inherit
- The abstract class defines a template method that implements the common steps of an operation, and declares abstract primitive operations that represent the variation points
- The subclasses override the primitive operations to realize the subclass-specific variations of the template method

6.5 OO Design Patterns

Template Method Pattern (continued)

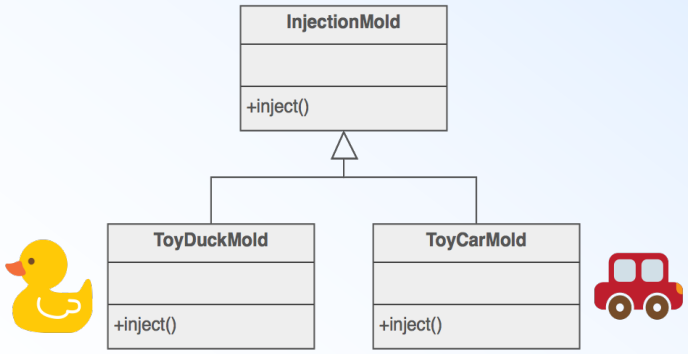
- Create “list_line_item()” method in Services class that prints the line item fields. This method calls a local abstract method class “price()” to print the item’s price. Each of the service subclasses overrides the “price()” method to reflect how the price for that service is computed



6.5 OO Design Patterns

Factory Method Pattern

- The Factory Method pattern is used to encapsulate the code that creates objects
- The similar but not identical methods are the constructor methods that instantiate objects. Create an abstract class that defines an abstract constructor method (Factory Method) and subclasses override the Factory Method to construct specific objects
- Example: Injection molding presses demonstrate this pattern. Manufacturers of plastic toys process plastic molding powder, and inject the plastic into molds of the desired shapes. The class of toy (car, action figure, etc.) is determined by the mold.

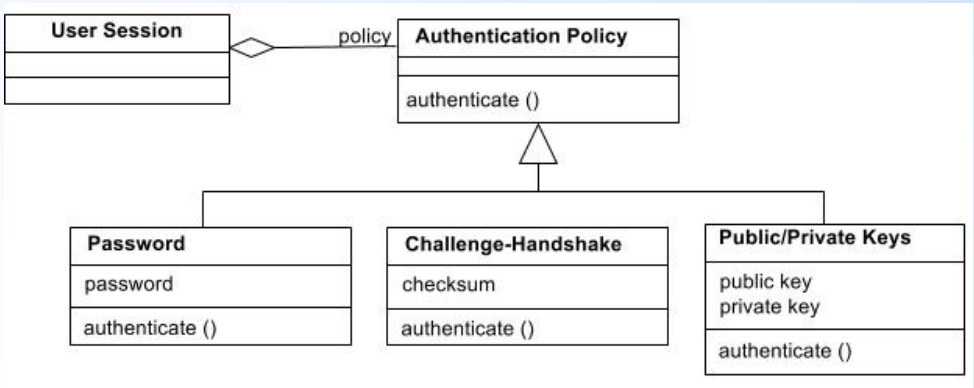


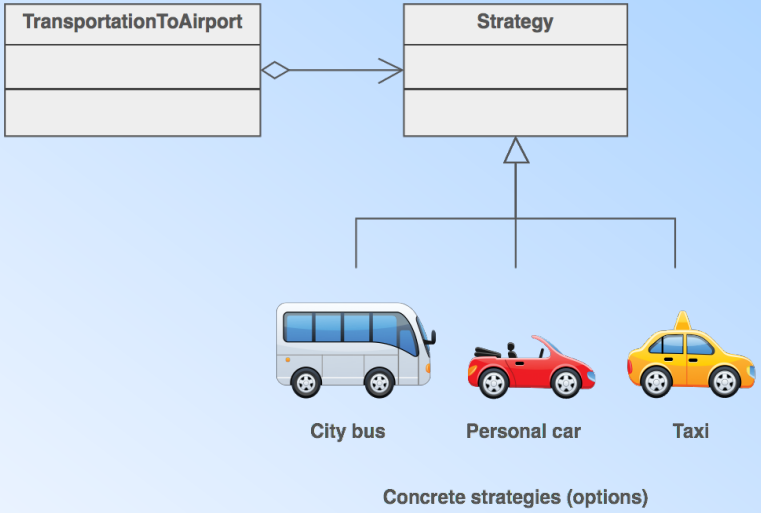
Reference: https://sourcemaking.com/design_patterns/factory_method

6.5 OO Design Patterns

Strategy Pattern

- The Strategy pattern allows algorithms to be selected at runtime since the choice of best algorithm may not be known till application is running
- Examples: user-authentication algorithm to use depends on the login request (left) and modes of transportation to an airport (right)



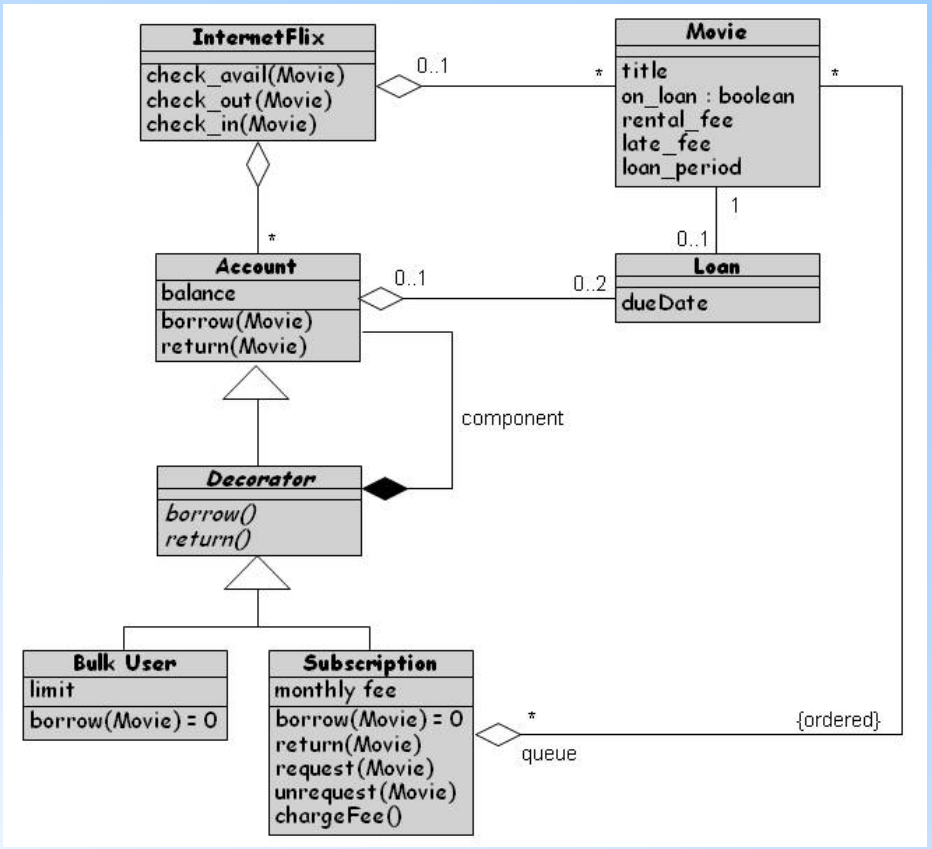


Reference: https://sourcemaking.com/design_patterns/strategy

6.5 OO Design Patterns

Decorator Pattern

- Extend an object's functionality at runtime
  - Decorator is a subclass of the object it decorates
  - Decorator contains a reference to the object it decorates
- Flexible alternative to using inheritance at design time to create subclasses that support new features
- Example: System for movie rental - base object (Account), each possible feature of an account is a subclass of Decorator class.



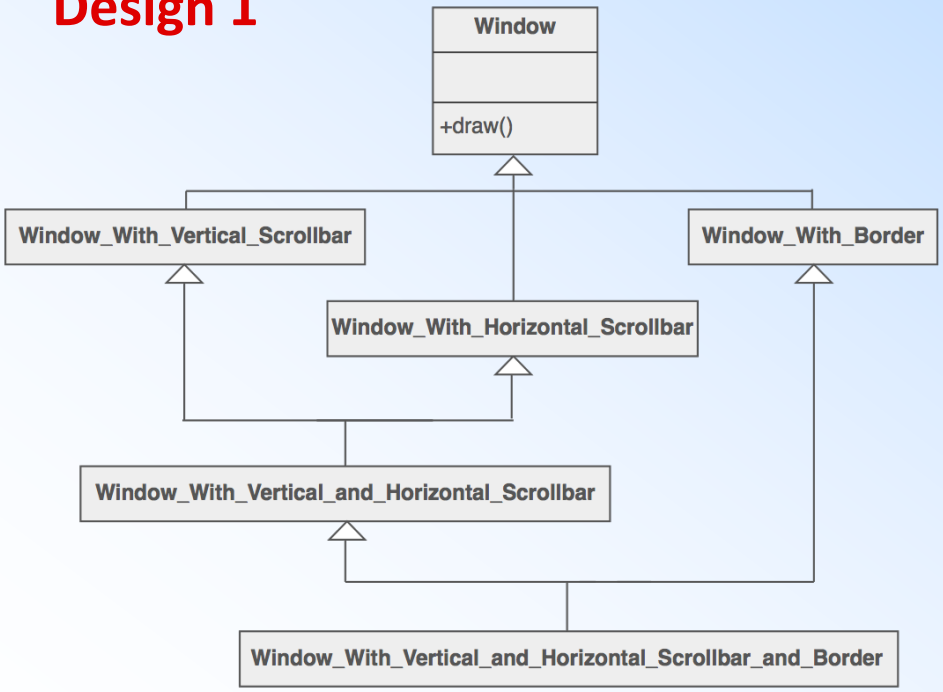
6.5 OO Design Patterns

Decorator Pattern – More examples

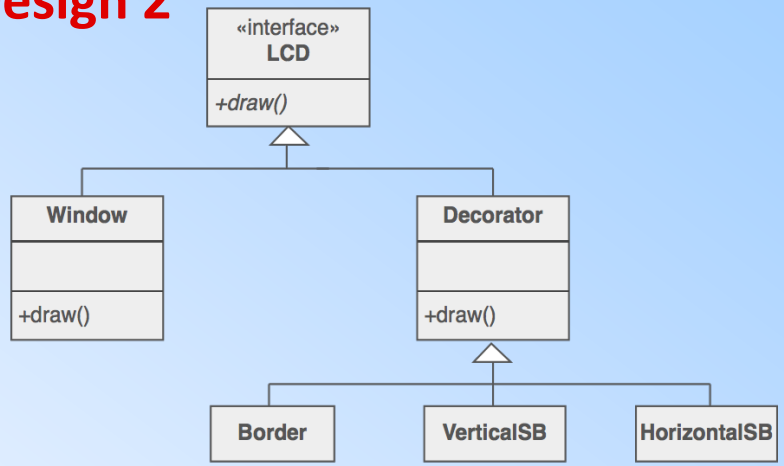
Analogy: wrapping a gift, putting in a box, and wrapping the box

Example: Interested in adding borders and scrollbars to the windows

Design 1



Design 2

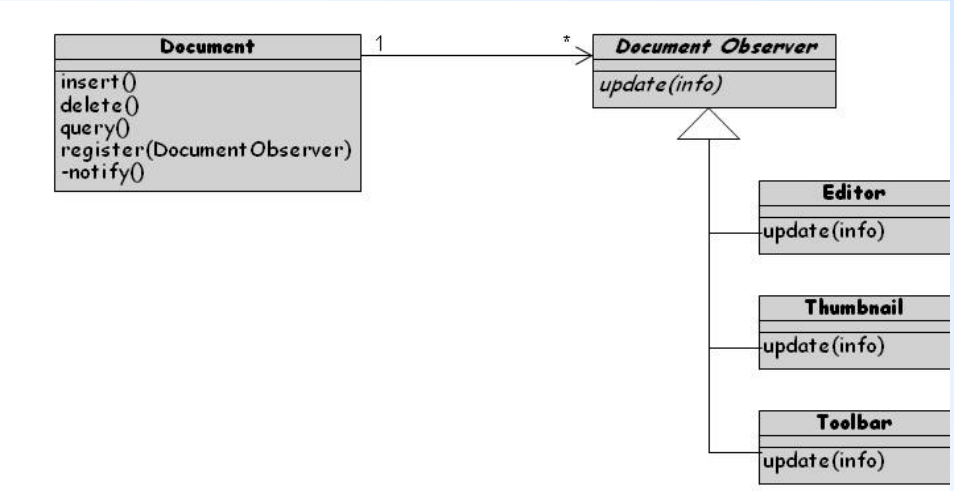


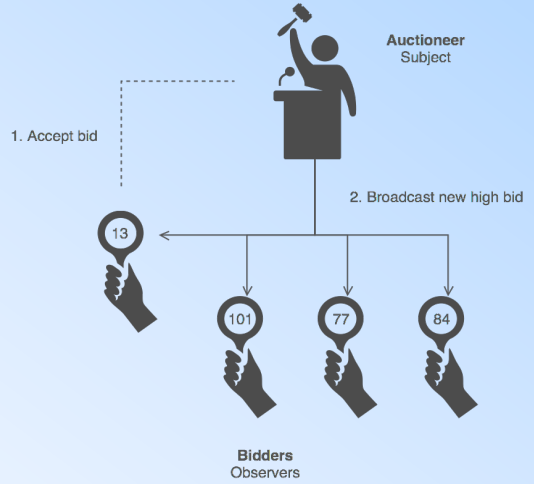
Reference: https://sourcemaking.com/design_patterns/decorator

6.5 OO Design Patterns

Observer Pattern

- An application of the publish–subscribe architecture style
- Useful when software needs to notify multiple objects of key events. Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically
- Examples: text editor offering multiple views of a document being edited (left) and auctions (right). Each bidder has a paddle used to indicate a bid. The auctioneer starts the bidding, and "observes" when a paddle is raised to accept the bid. The acceptance of the bid changes the bid price which is broadcast to all the bidders in the form of a new bid.



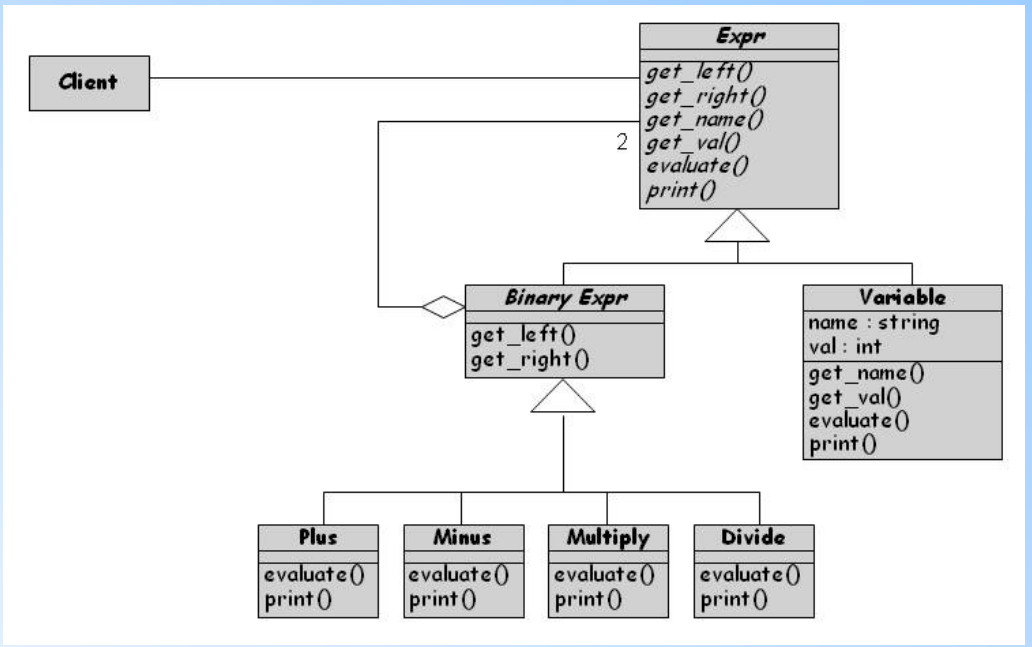


Reference: https://sourcemaking.com/design_patterns/observer

6.5 OO Design Patterns

Composite Pattern

- A composite is an object designed as a composition of one or multiple objects with similar functionality
- Composes objects in terms of a tree structure to represent part (or whole) of the hierarchy
- The composite pattern promotes the uses of a **single** uniform interface (e.g., Expr class)
- Useful whenever we have "composites that contain components, each of which could be a composite“
- For instance, menus that contain menu items, each of which could be a menu or directories that contain files, each of which could be a directory

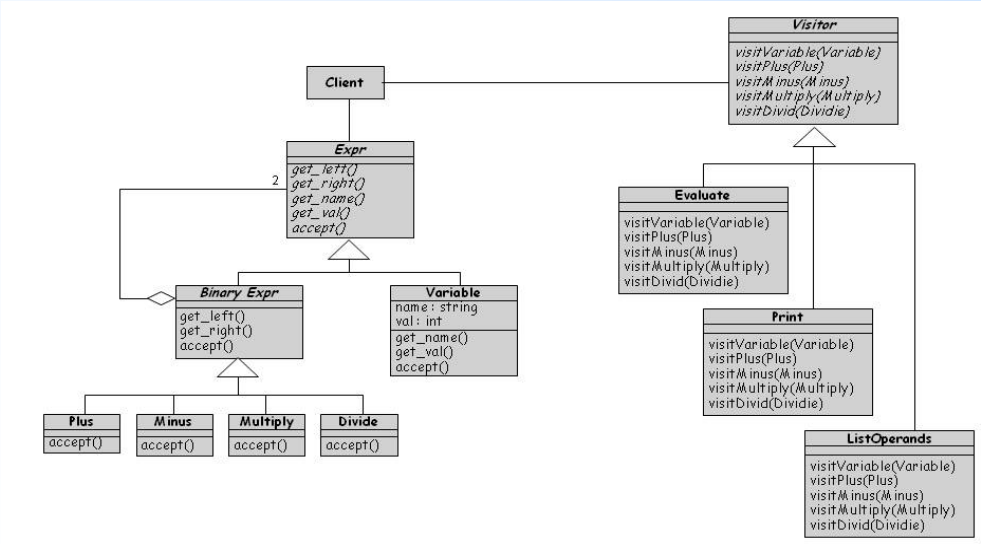


- Example: mathematical expressions modeled as tree structure in which nodes represent various operators and variable operands

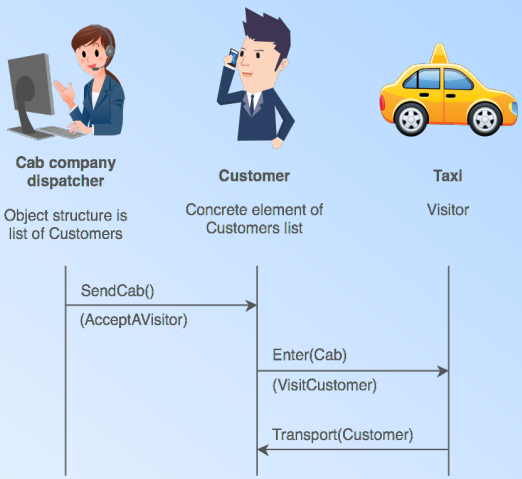
6.5 OO Design Patterns

Visitor Pattern

- Collects and encapsulates operation fragments into their own classes
- Each operation is implemented as a separate subclass of abstract `Visitor` class
- New operations can be added without changing the composite object's code
- Operations can be performed on elements of an object structure without changing the classes on which it operates
- Example: accept(Visitor) – single class for performing operations. For instance, accept() method in Divide always calls VisitorDivide().



- Example: When a person calls a taxi company (accepting a visitor), the company dispatches a cab to the customer. Upon entering the taxi, the customer, or Visitor, is no longer in control of his or her own transportation, the taxi (driver) is.



Reference: https://sourcemaking.com/design_patterns/visitor

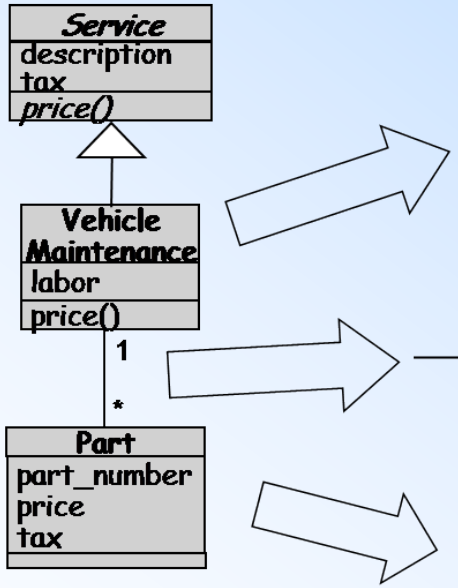
6.6 Other Design Considerations

Data Management

- Data management takes into account the system requirements concerning performance and space
- Four steps:
  - Identify the data, data structures, and relationships among them
  - Design services to manage the data structures and relationships
  - Find tools, such as database management systems, to implement some of the data management tasks
  - Design classes and class hierarchies to oversee the data management functions

6.6 Other Design Considerations

Data Management for the Royal Service Station



vehicle maintenance	ID	description	tax	labor
	1	oil change	0.25	5
	2	rotate tires	0.25	5
	3	replace timing belt	5	100

Maintenance X part	ID	part_number
	1	1X782
	2	-
	3	F89SERT

part	part_number	price	tax
	1X782	8	.40
	P3291	200	10
	E89WWY	34	1.70

6.6 Other Design Considerations

Exception Handling

- Allows making programs become more robust
- Helps separate error checking and recover from a program's main functionality

6.6 Other Design Considerations
Exception Handling (continued)

```
attempt_transmission (message: STRING) raises TRANSMISSIONEXCEPTION
        // Attempt to transmit message over a communication line
using   // the low-level procedure unsafe_transmit, which may fail,
        //triggering an exception.
        // After 100 unsuccessful attempts, give up and raise an
        exception
local
        failures: INTEGER
try
        unsafe_transmit (message)
rescue
        failures := failures + 1;
        if failures < 100 then
                retry
        else
                raise TRANSMISSIONEXCEPTION
        end
end
```

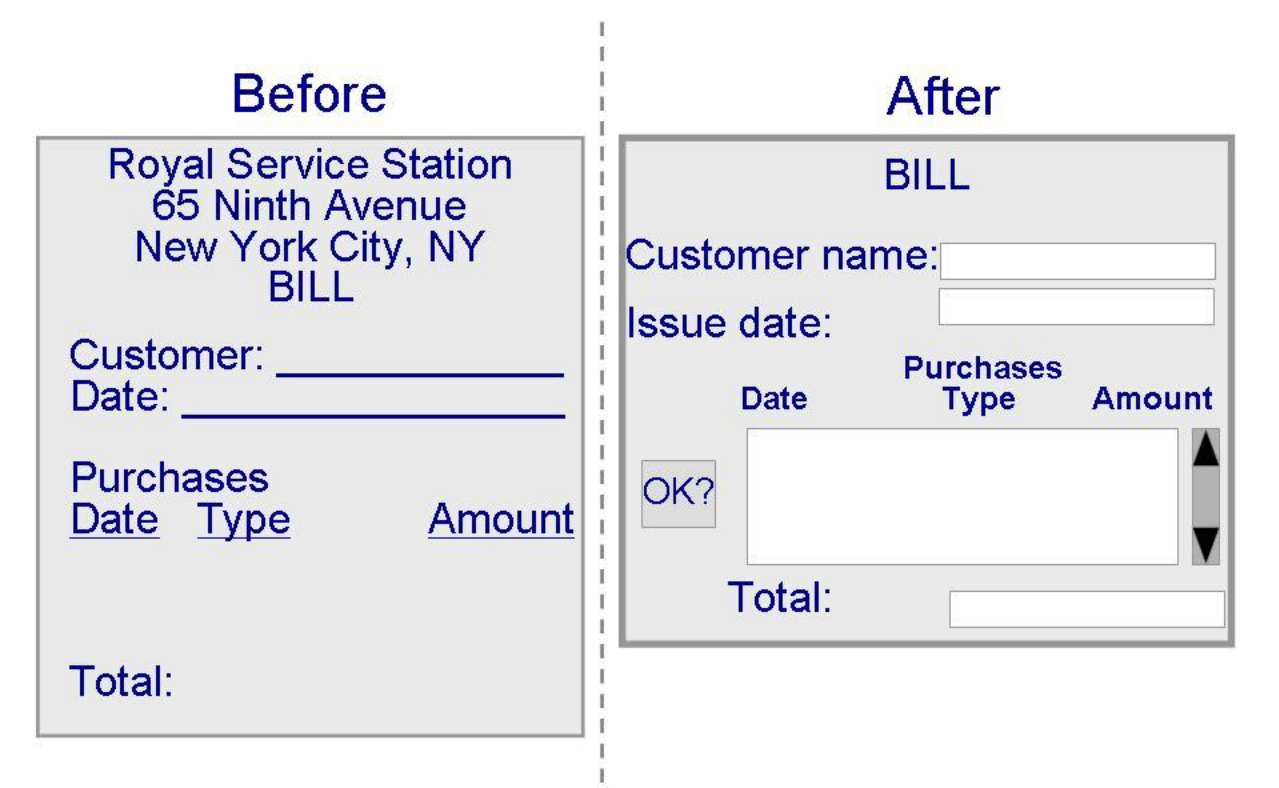
6.6 Other Design Considerations

Designing User Interfaces

- Must consider several issues:
  - identifying the humans who will interact with the system
  - defining scenarios for each way that the system can perform a task
  - designing a hierarchy of user commands
  - refining the sequence of user interactions with the system
  - designing relevant classes in the hierarchy to implement the user-interface design
  - decisions
  - integrating the user-interface classes into the overall system class hierarchy

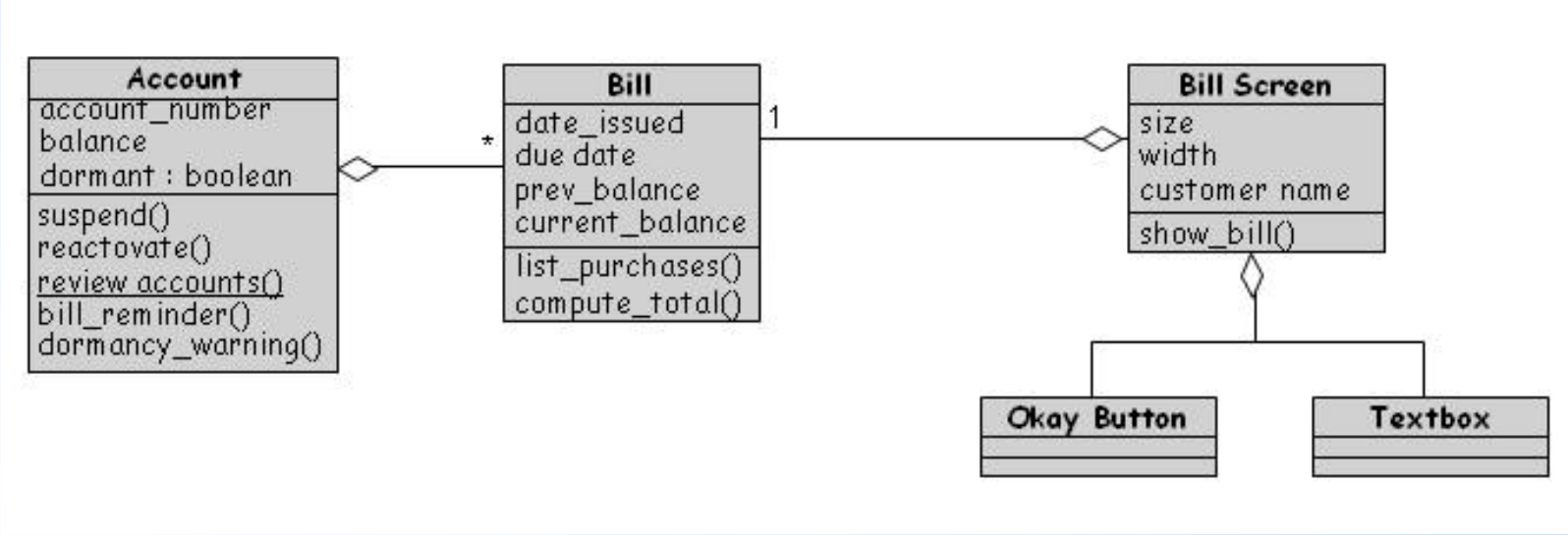
6.6 Other Design Considerations

Designing User Interfaces (continued)



6.6 Other Design Considerations

Designing User Interfaces (continued)



6.6 Other Design Considerations

Frameworks

- A framework is a large reusable design for a specific application domain
- GUI editors, web applications, accounting systems
- Different from software product lines
  - Product lines are developed by a company for its own use
  - Frameworks tend to be publically available resources like toolkits
    - High-level architectures whose low-level details need to be filled-in

6.7 OO Measurement

OO Size Measures

- Objects and methods as a basic size measure
- Lorenz and Kidd's nine aspects of size
  - Number of scenario script (NSS)
  - Number of key classes
  - Number of support classes
  - The average number of support classes per key classes
  - Number of subsystems
  - Class size
  - Number of operations overridden by a subclass (NOO)
  - Number of operation added by a subclass
  - Specialization index = (NOO X depth) / (total class methods)

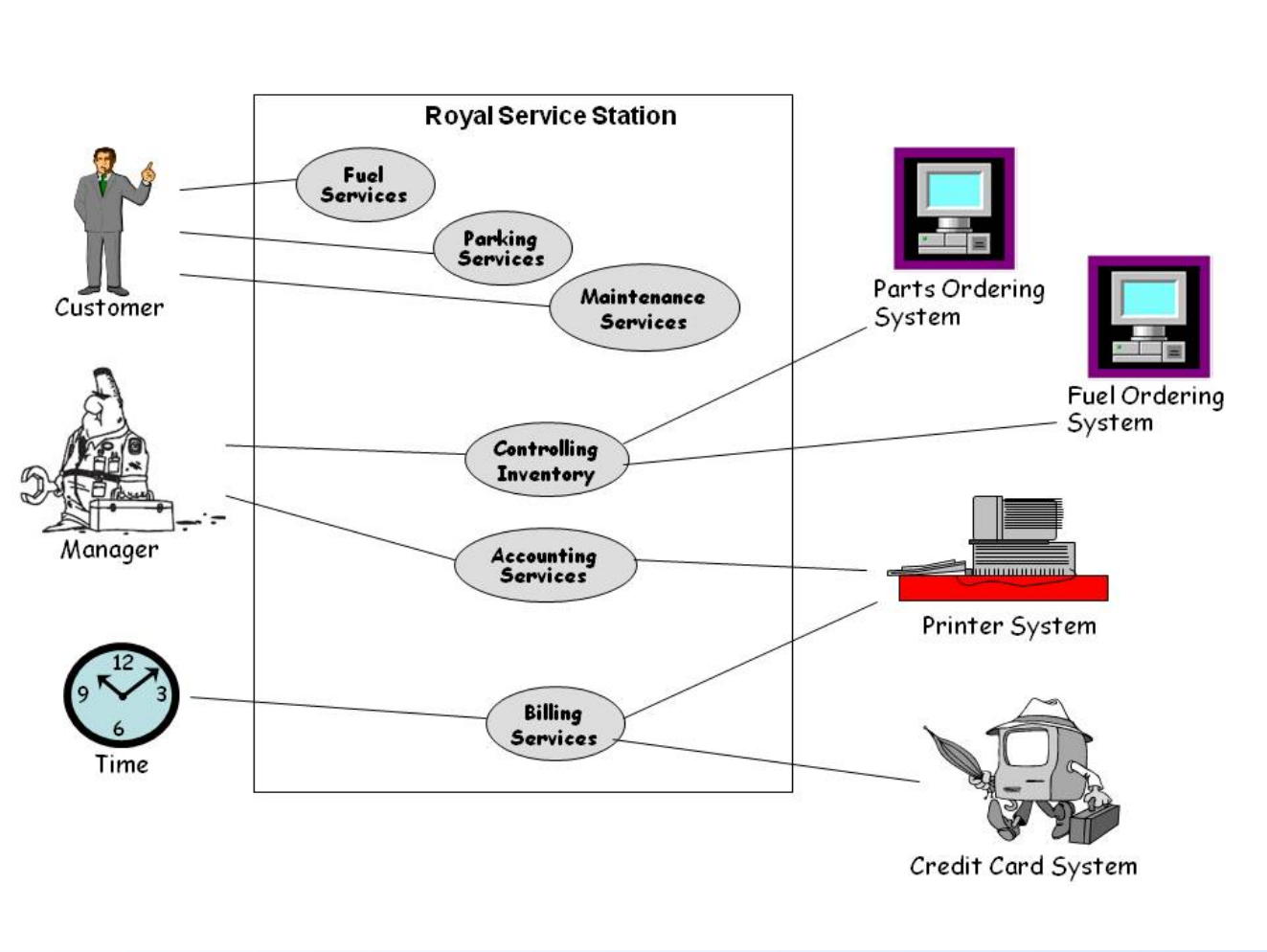
6.7 OO Measurement

Lorenz and Kidd Metrics Collection in Different Phases of Development

Metric	Requirements Description	System Design	Program Design	Coding	Testing
Number of scenario script	X				
Number of key classes	X	X			
Number of support classes			X		
Average number of support classes per key class			X		
Number of subsystem			X	X	
Class size		X	X	X	
Number of operations overridden by a subclass		X	X	X	X
Number of operations added by a subclass		X	X	X	
Specialization index		X	X	X	X

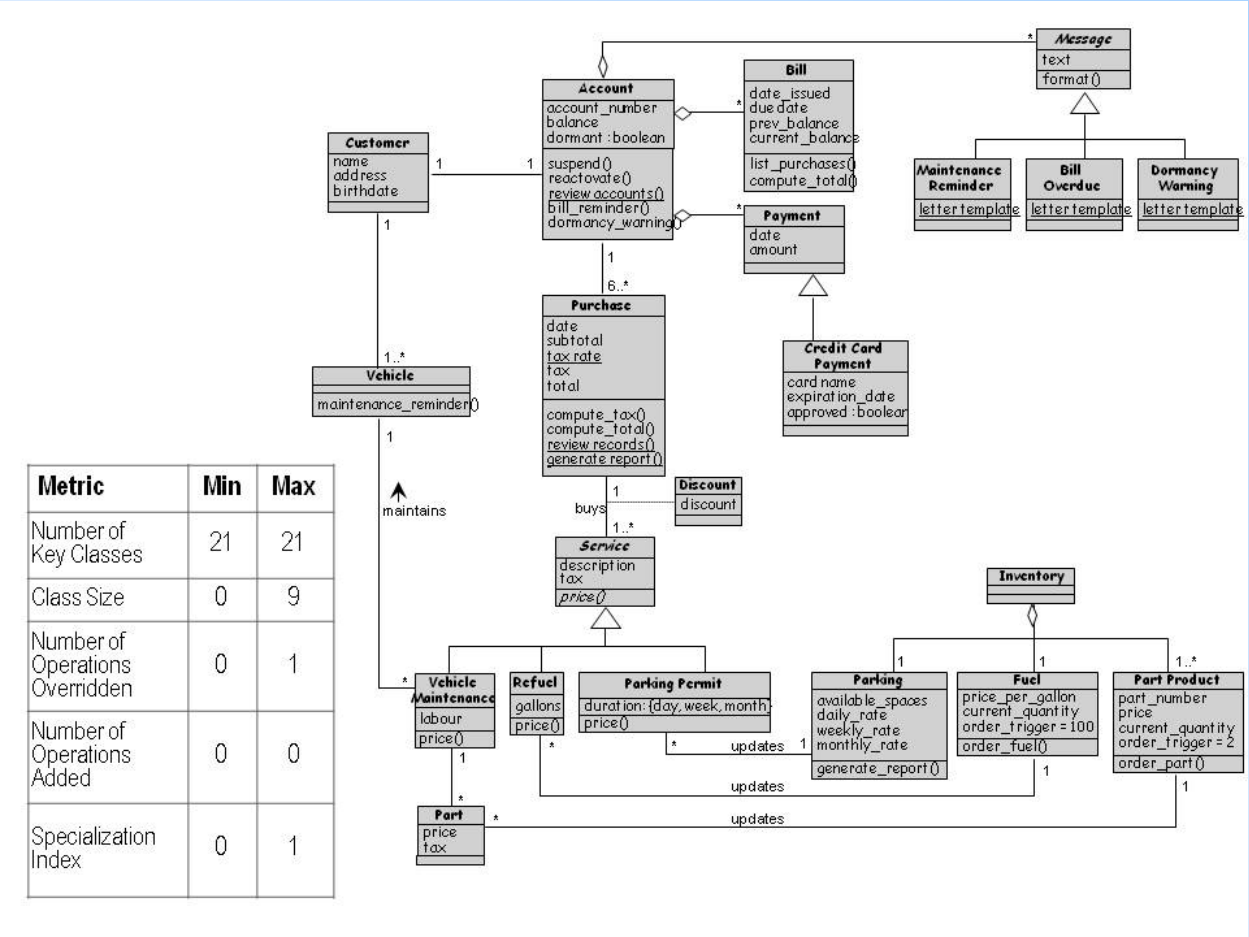
6.7 OO Measurement

Use Case Diagram of the Royal Service Station



6.7 OO Measurement

Class Hierarchy for the Royal Service Station



Metric	Min	Max
Number of Key Classes	21	21
Class Size	0	9
Number of Operations Overridden	0	1
Number of Operations Added	0	0
Specialization Index	0	1

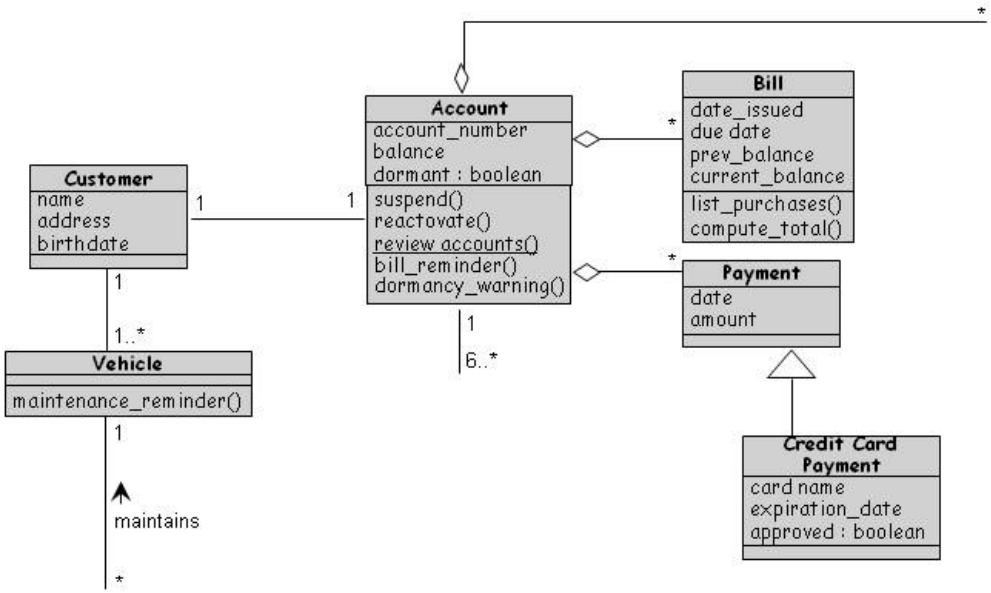
6.7 OO Measurement

OO Design Quality Measures

- Chidamber and Kemerer have also devised a suite of metrics for object-oriented development
- Focused on design quality (not size)
  - Weighted methods per class = $\sum^{n}_{i=1} c_i$
    - n: number of methods and c: complexity of each method
  - Depth of inheritance
  - Number of children
  - Coupling between objects
  - Response for a class
  - Lack of cohesion of methods

6.7 OO Measurement

Chidamber-Kemerer Metrics applied to the Royal Service Station's System Design



Metric	Bill	Payment	Credit Card Payment	Account	Customer	Vehicle
Weighted Methods / Class	2	0	0	5	0	1
Number of Children	0	1	0	0	0	0
Depth of Inheritance Tree	0	0	1	0	0	0
Coupling Between Objects	1	2	1	5	2	2

6.7 OO Measurement

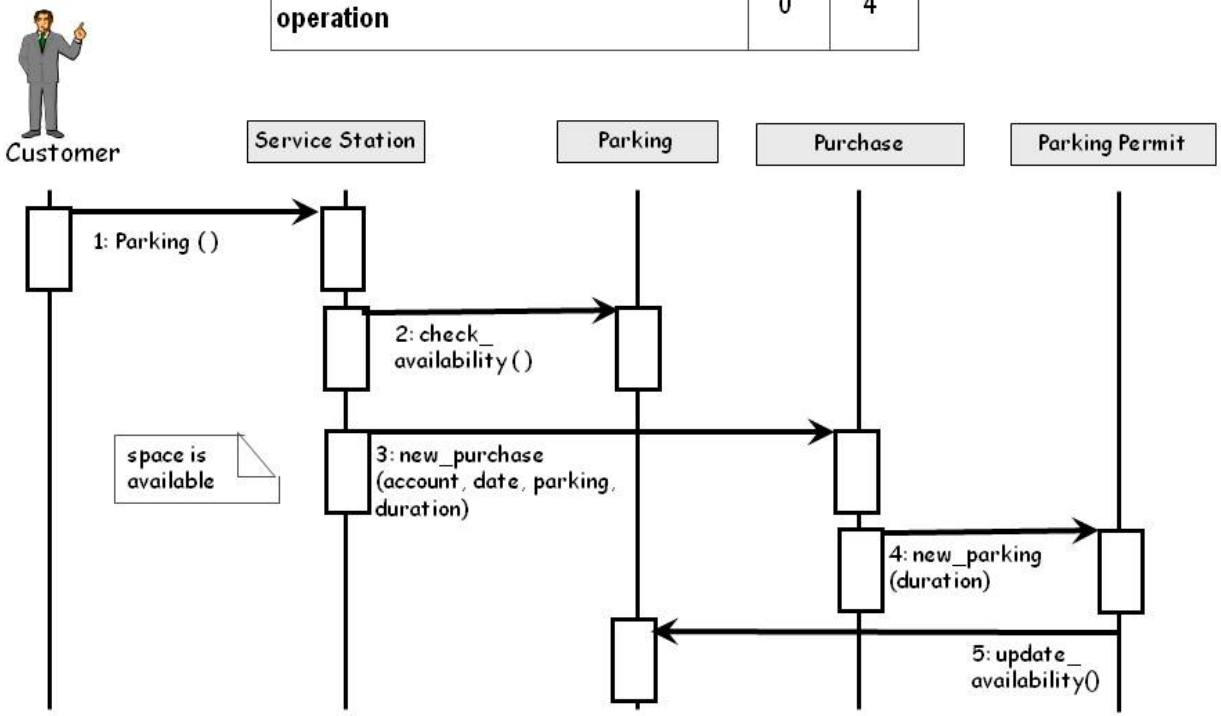
Calculating the Degree of Cohesion

- Given class C with n methods, M_1 through M_n
- Suppose $\{I_j\}$ is the set of instance variables used by the method M
- We can define P to be collection of pairs (I_r, I_s) where I_r and I_s, share no common members
  - $P = \{(I_r, I_s) \mid I_r \cap I_s = \emptyset\}$
- Q is the collection of pairs (I_r, I_s) where I_r and I_s, share at least one common member
  - $Q = \{(I_r, I_s) \mid I_r \cap I_s \neq \emptyset\}$
- Lack of cohesion in methods for C to be
  - $|P|-|Q|$ if $|P| > |Q|$
  - Zero if otherwise

6.7 OO Measurement

Measuring From a Sequence Diagram

Metric	Min	Max
Average operation size	5	5
Average number of parameters per operation	0	4



6.7 OO Measurement

Where to Do OO Measurement

- Measurement is only valuable when it increases our understanding, prediction, or control
- Metrics are available for many types of documents including
  - Use cases
  - Class diagrams
  - Interaction diagrams
  - Class descriptions
  - State diagrams
  - Package diagrams

6.7 OO Measurement

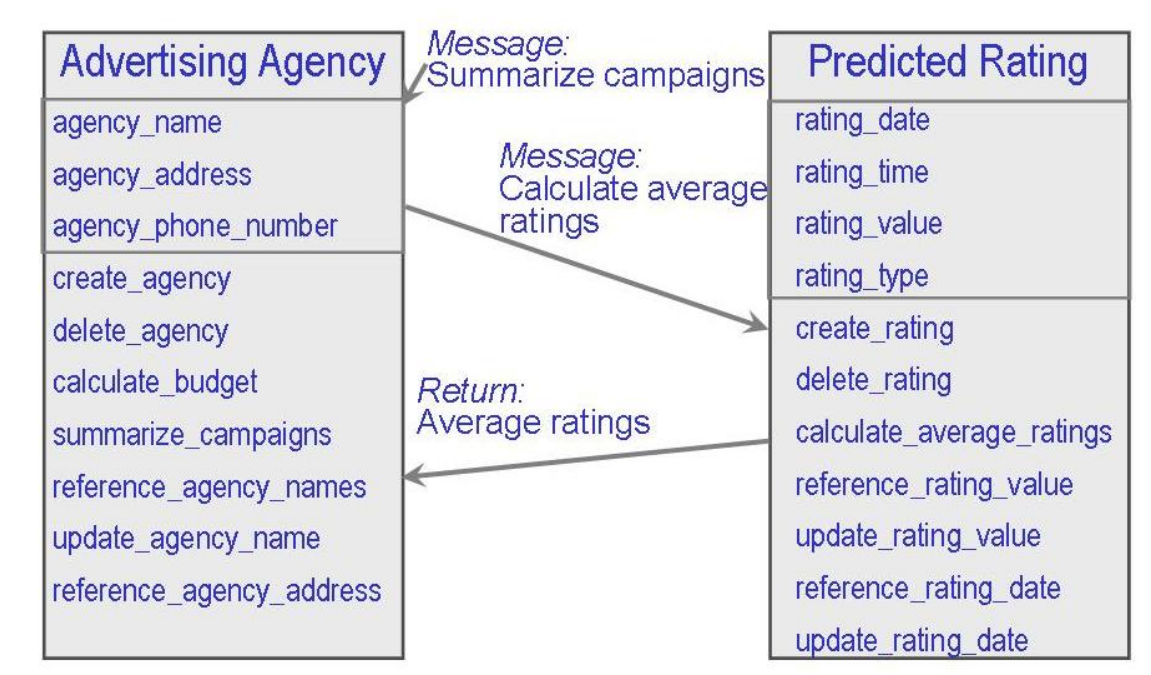
Where to Do OO Measurement (continued)

Metric	Use cases	Class diagram	Interaction diagram	Class description	State diagram	Package diagram
Number of scenario scripts	X					
Number of key classes		X				
Number of support classes		X				
Average number of support classes per key class		X				
Number of subsystems						X
Class size		X		X		
Number of operation overridden by a subclass		X				
Number of operation added by a subclass		X				
Specialization index		X				
Weighted methods in class		X				
Depth of inheritance		X				
Number of children		X				
Coupling between objects		X				
Response for a class				X		
Lack of cohesion in methods				X		
Average operation size			X			
Average number of parameters per operation			X			
Operation complexity				X		
Percent public and protected				X		
Public access to data members				X		
Number of root classes		X				
Fan-in/fan-out		X				

6.9 Information System Example

Data Model of `Opposition Programs` Broadcast by Piccadilly's Competition

- Domain elements and relationships that the Piccadilly database will maintain
- A closer examination will reveal that there are considerable commonality



6.10 Real-Time Example

Design by Contract

- Had Ariane-5 been implemented using an object-oriented approach, the reuse would have been either in terms of composition or inheritance
- In composition approach: the SRI (inertial reference software) is viewed as a black box and called from the main system
- In inheritance approach: the SRI structure and behavior are open to view, inheriting as much structure and behavior from parent classes as possible

6.11 What This Chapter Means For You

- The design process describes the system components using a common language with which to communicate
- Object orientation is a particularly appealing basis for such design, because it allows us to describe and reason about the system from its birth to its delivery in the same terms: classes, objects, and methods
- Consistent notation makes it easier for your team to understand the implications of using a particular object or class
  - Consistency assists the maintainers and testers, enabling them to build test cases and monitor changes more easily
  - Because the requirements, design, and code are expressed in the same way, it is easier for you to evaluate the effects of proposed changes to the requirements or designs