Chapter 8, Object Design: Reusing Pattern Solutions

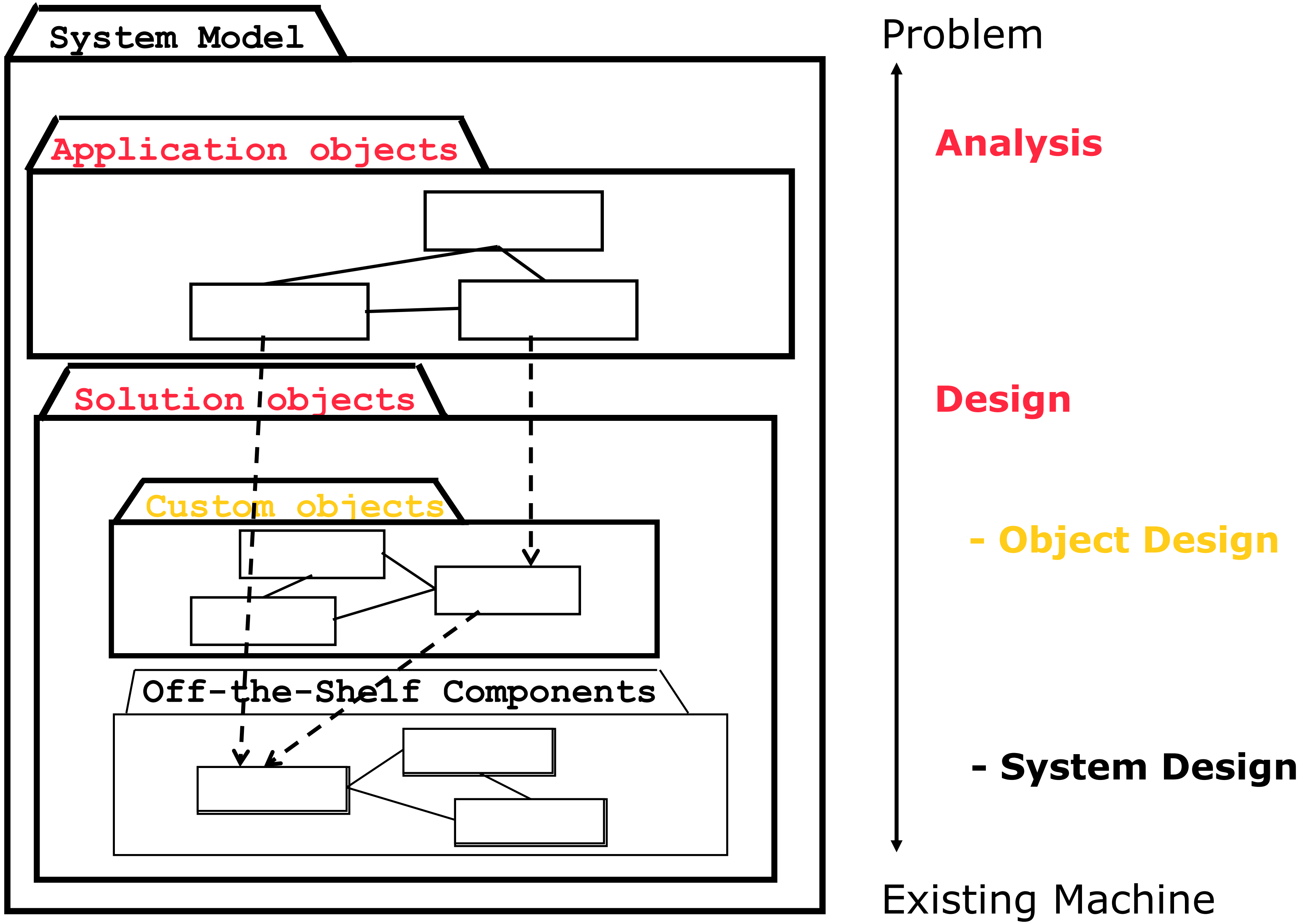
Application Objects and Solution Objects

- Application objects, also called “ domain objects,” represent concepts of the domain that are relevant to the system.
- Solution objects represent components that do not have a counterpart in the application domain, such as persistent data stores, user interface objects, or middleware.

When do we identify objects?

- During analysis, we identify entity objects. Most entity objects are application objects that are independent of any specific system.
- During analysis, we also identify solution objects that are visible to the user, such as boundary and control objects representing forms and transactions defined by the system.
- During system design, we identify more solution objects in terms of software and hardware platforms.
- During object design, we refine and detail both application and solution objects and identify additional solution objects (closing the gap).

System Development as a Set of Activities



Off-the-shelf components

- During system design, we describe the system in terms of its architecture and define the hardware/software platform on which we build the system.
- This allows the selection of off-the- shelf components that provide a higher level of abstraction than the hardware.
- During object design, we close the gap between the application objects and the off-the-shelf components by identifying additional solution objects and refining existing objects.

Object Design Activities

1. Reuse: Identification of existing solutions
   - Use of inheritance
   - Off-the-shelf components and additional solution objects
   - Design patterns
2. Interface specification
   - Describes precisely each class interface
3. Object model restructuring
   - Transforms the object design model to improve its understandability and extensibility
4. Object model optimization
   - Transforms the object design model to address performance criteria such as response time or memory utilization.

Object Design

Mapping Models to Code

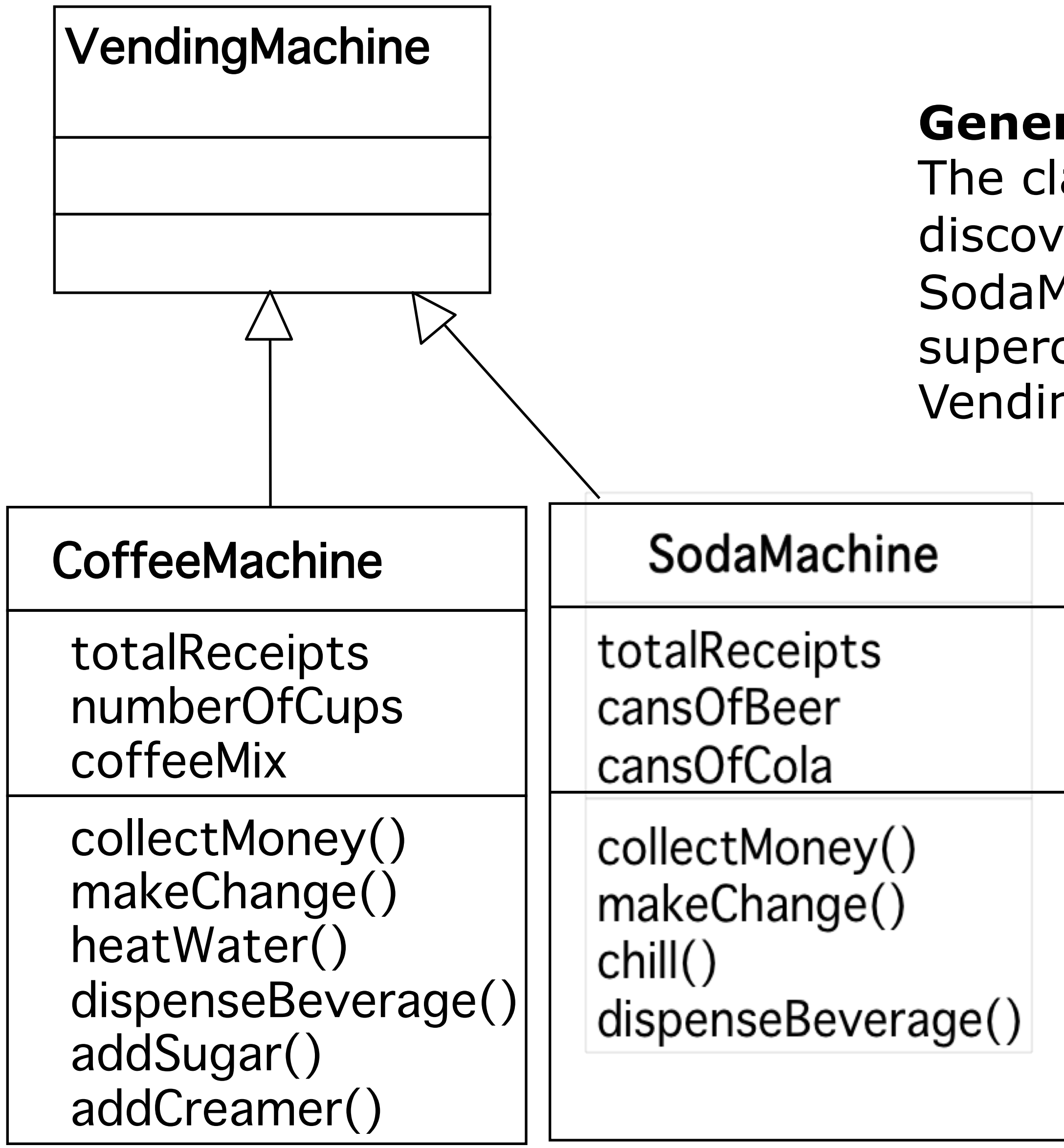
Reuse

- Main goal:
  - Reuse knowledge from previous experience to current problem
  - Reuse functionality already available
- Composition (also called Black Box Reuse)
  - New functionality is obtained by aggregation
  - The new object with more functionality is an aggregation of existing components
- Inheritance (also called White-box Reuse)
  - New functionality is obtained by inheritance.
  - Three ways to get new functionality:
    - Implementation inheritance
    - Interface inheritance
    - Delegation

Discovering Inheritance

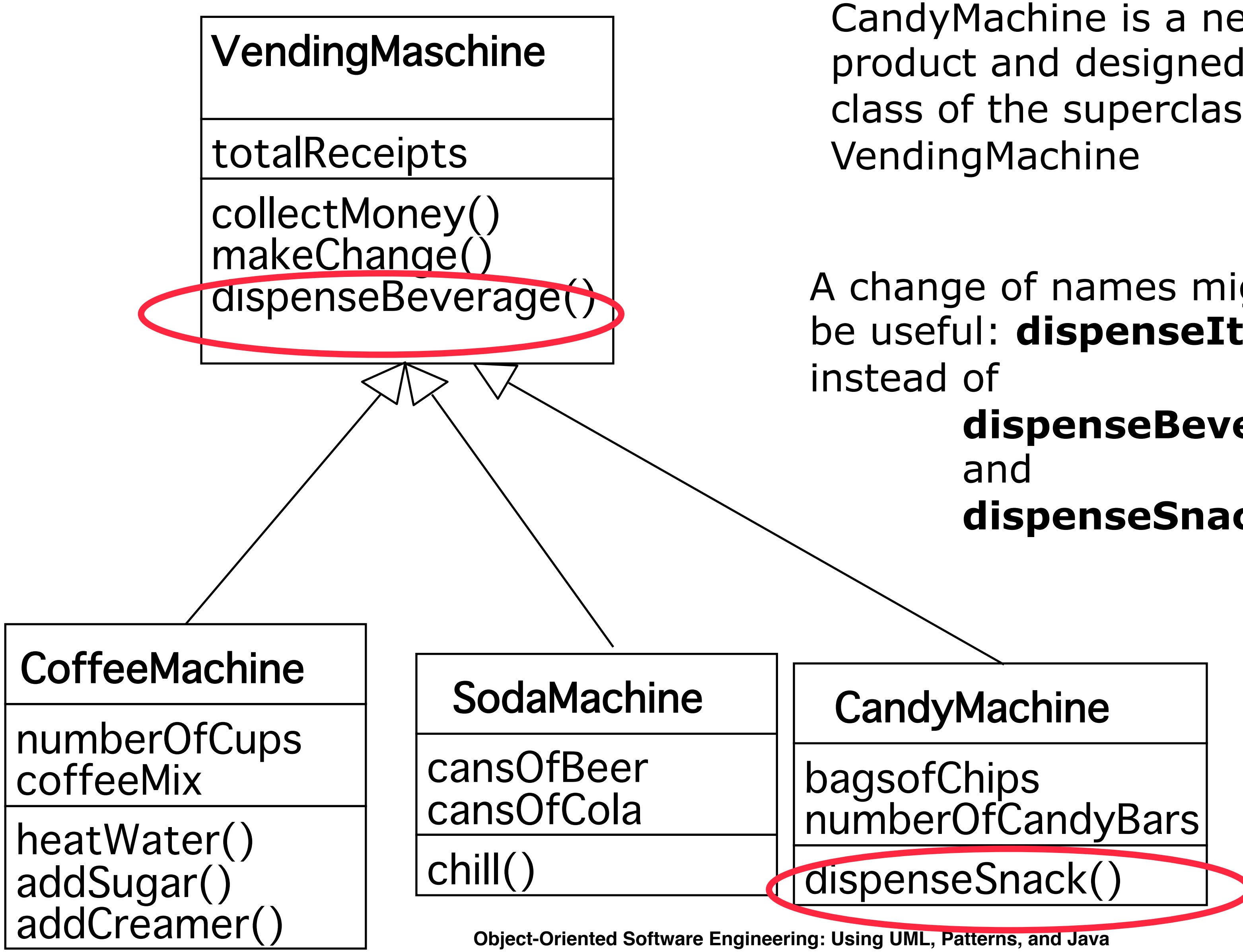
- To “discover“ inheritance associations, we can proceed in two ways, which we call specialization and generalization
- Generalization: the discovery of an inheritance relationship between two classes, where the sub class is discovered first.
- Specialization: the discovery of an inheritance relationship between two classes, where the super class is discovered first.

Generalization Example: Modeling a Coffee Machine



Generalization:
The class CoffeeMachine is discovered first, then the class SodaMachine, then the superclass VendingMachine

Example of a Specialization



CandyMachine is a new product and designed as a sub class of the superclass VendingMachine

A change of names might now be useful: **dispenseItem()** instead of

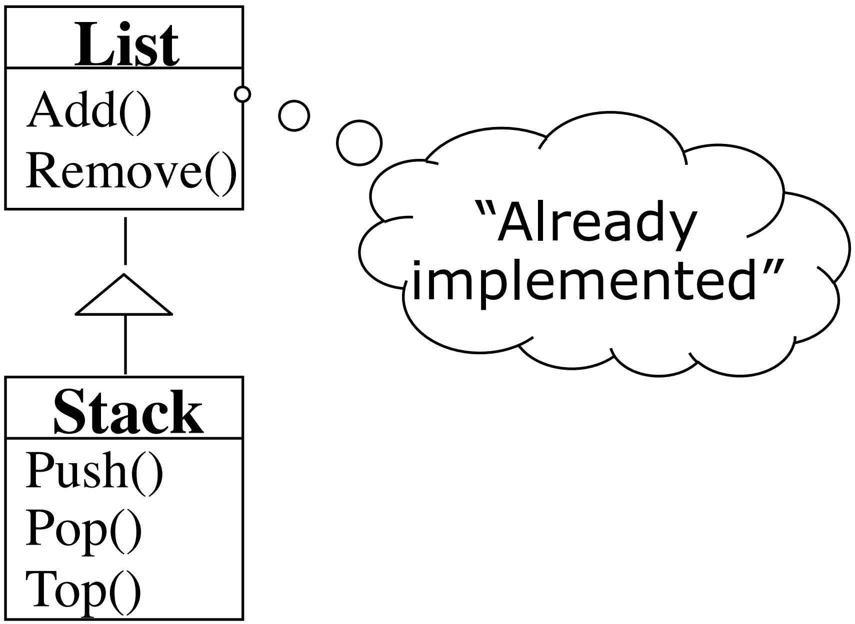
dispenseBeverage()
and
dispenseSnack()

Implementation Inheritance and Specification Inheritance

- Implementation inheritance
  - Also called class inheritance
  - Goal:
    - Extend an applications' functionality by reusing functionality from the super class
    - Inherit from an existing class with some or all operations already implemented
- Specification Inheritance
  - Also called subtyping
  - Goal:
    - Inherit from a specification
    - The specification is an abstract class with all operations specified, but not yet implemented.

Example for Implementation Inheritance

- A very similar class is already implemented that does almost the same as the desired class implementation



Example:

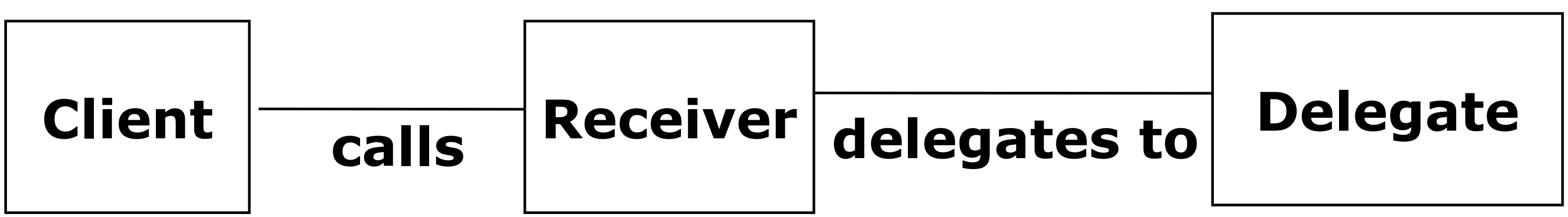
- I have a **List** class, I need a **Stack** class
  - How about subclassing the **Stack** class from the **List** class and implementing **Push()**, **Pop(), Top()** with **Add()** and **Remove()**?

❖ Problem with implementation inheritance:

- The inherited operations might exhibit unwanted behavior.
- Example: What happens if the Stack user calls **Remove()** instead of **Pop()**?

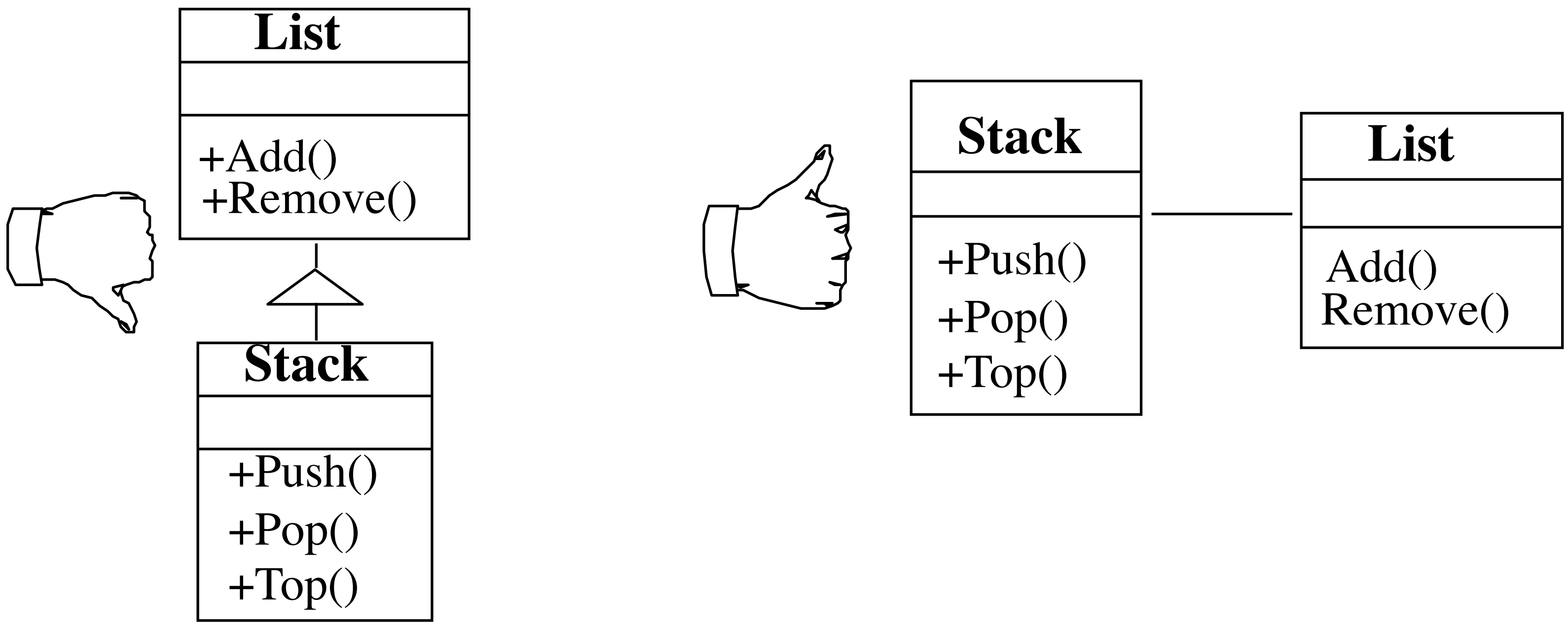
Delegation

- Delegation is a way of making composition as powerful for reuse as inheritance
- In delegation two objects are involved in handling a request from a Client
  - The Receiver object delegates operations to the Delegate object
  - The Receiver object makes sure, that the Client does not misuse the Delegate object.



Delegation instead of Implementation Inheritance

- Inheritance: Extending a Base class by a new operation or overwriting an operation.
- Delegation: Catching an operation and sending it to another object.
- Which of the following models is better?



Comparison: Delegation v. Inheritance

- Code-Reuse can be done by delegation as well as inheritance
- Delegation
  - Flexibility: Any object can be replaced at run time by another one
  - Inefficiency: Objects are encapsulated
- Inheritance
  - Straightforward to use
  - Supported by many programming languages
  - Easy to implement new functionality
  - Exposes a subclass to details of its super class
  - Change in the parent class requires recompilation of the subclass.

Frameworks

- A framework is a reusable partial application that can be specialized to produce custom applications.
- The key benefits of frameworks are reusability and extensibility:
  - Reusability leverages the application domain knowledge and prior effort of experienced developers
  - Extensibility is provided by hook methods, which are overwritten by the application to extend the framework.

Frameworks in the Development Process

- Infrastructure frameworks aim to simplify the software development process
  - Used internally, usually not delivered to a client.
- Middleware frameworks are used to integrate existing distributed applications
  - Examples: MFC, DCOM, Java RMI, WebObjects, WebSphere, WebLogic Enterprise Application [BEA].
- Enterprise application frameworks are application specific and focus on domains
  - Example of application domains: telecommunications, avionics, environmental modeling, manufacturing, financial engineering, enterprise business activities.

White-box and Black-box Frameworks

- White-box frameworks:
  - Extensibility achieved through *inheritance* and dynamic binding.
  - Existing functionality is extended by subclassing framework base classes and overriding specific methods (so-called hook methods)
- Black-box frameworks:
  - Extensibility achieved by defining interfaces for components that can be plugged into the framework.
  - Existing functionality is reused by defining components that conform to a particular interface
  - These components are integrated with the framework via *delegation*.

Another Source for Finding Objects : Design Patterns

- What are Design Patterns?
  - A design pattern describes a problem which occurs over and over again in our environment
  - Then it describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same twice

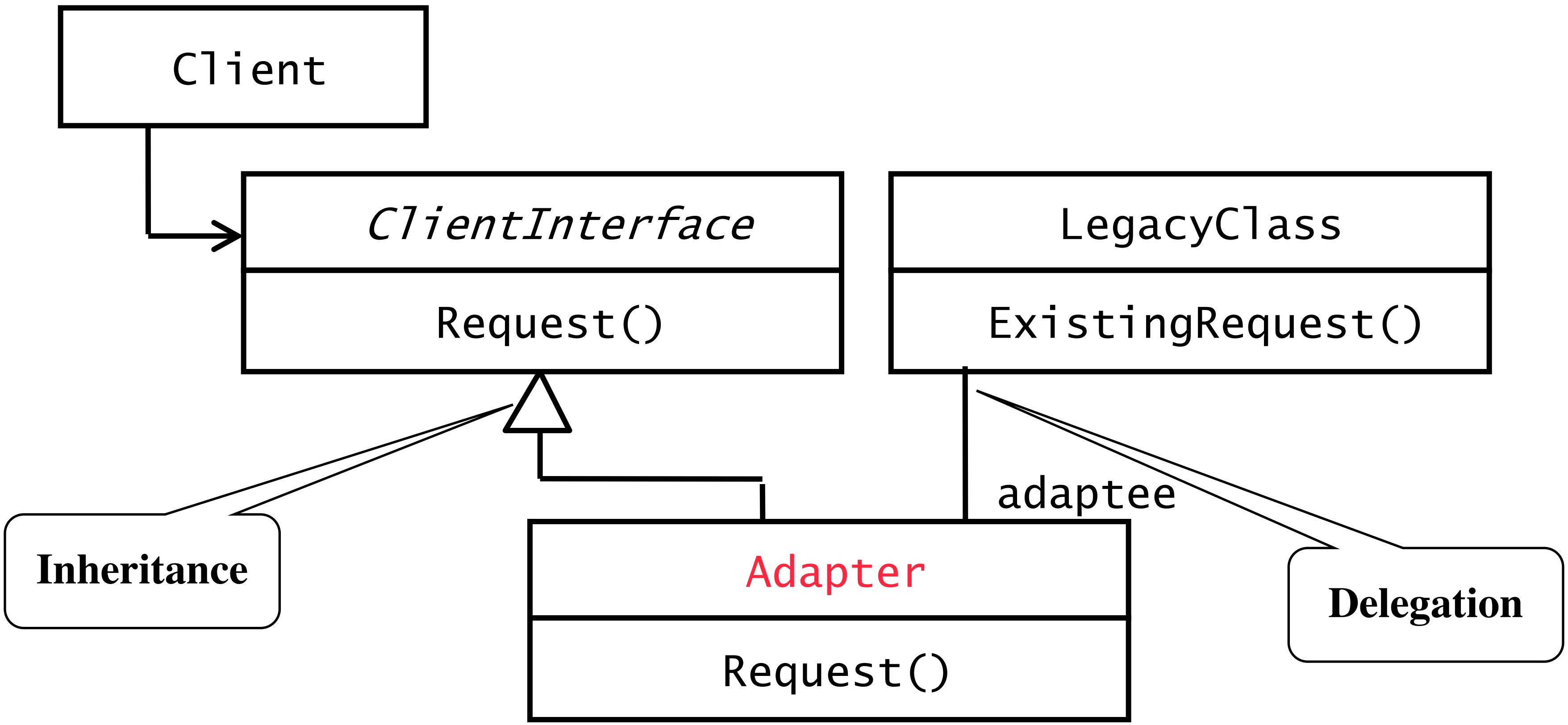
More on Design Patterns

- Design patterns are partial solutions to common problems such as
  - separating an interface from a number of alternate implementations
  - wrapping around a set of legacy classes
  - protecting a caller from changes associated with specific platforms
- A design pattern consists of a small number of classes
  - uses delegation and inheritance
  - these classes can be adapted and refined for the specific system under construction

Adapter Pattern

- Adapter Pattern: Connects incompatible components.
  - It converts the interface of one component into another interface expected by the other (calling) component
  - Used to provide a new interface to existing legacy components (Interface engineering, reengineering)

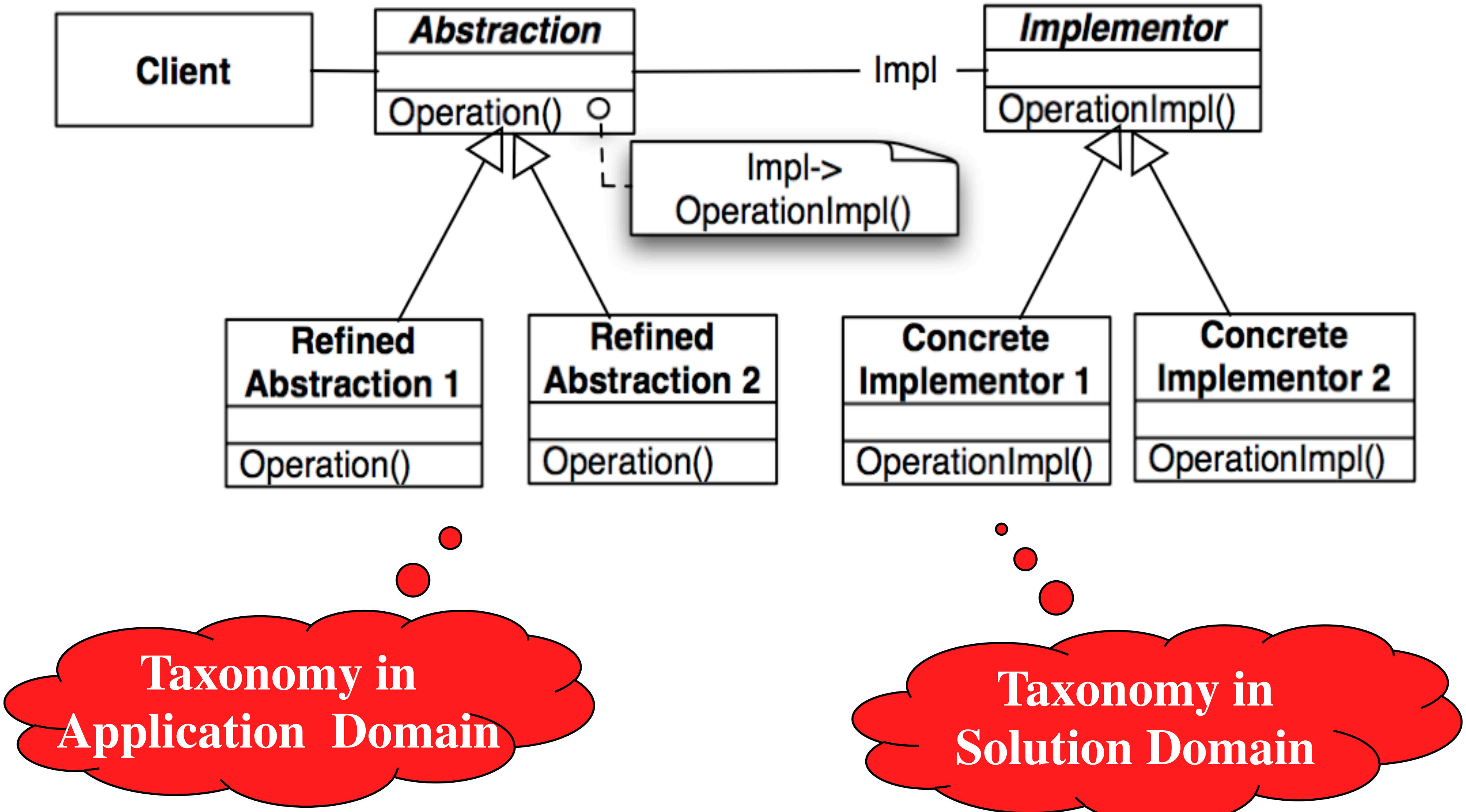
Adapter Pattern



Bridge Pattern

- Use a bridge to “decouple an abstraction from its implementation so that the two can vary independently” (From [Gamma et al 1995])
- Allows different implementations of an interface to be decided upon dynamically.

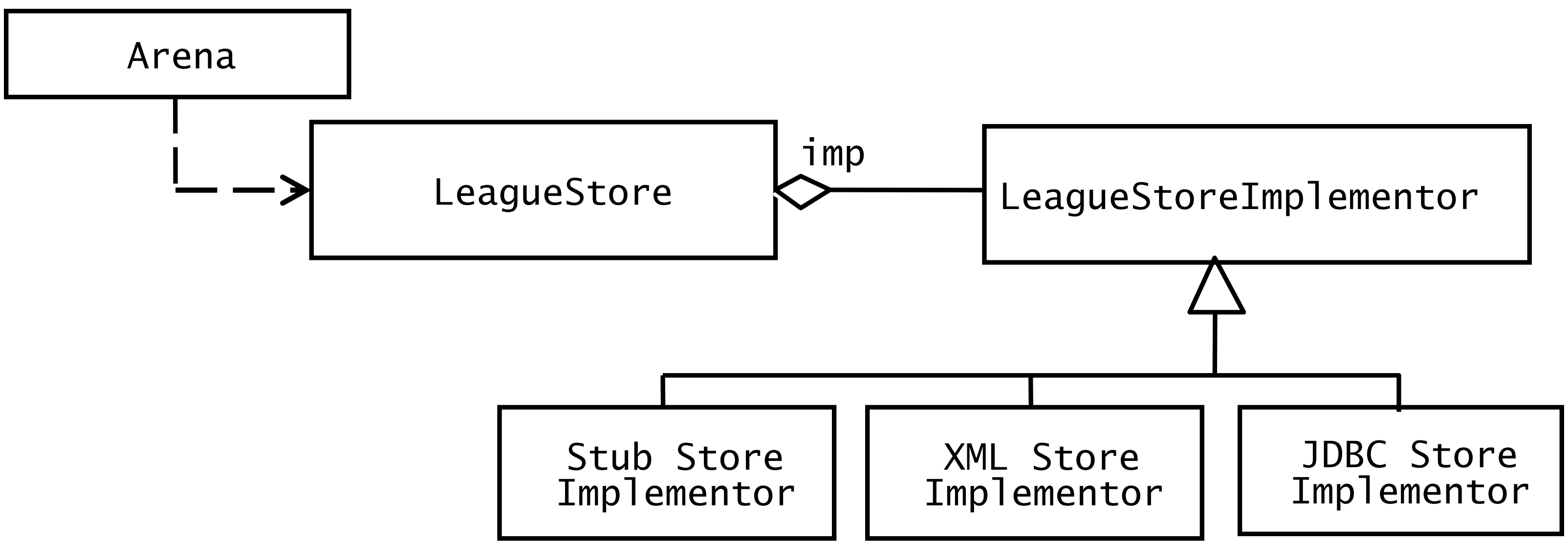
Bridge Pattern



Motivation for the Bridge Pattern

- Decouples an abstraction from its implementation so that the two can vary independently
- This allows to bind one from many different implementations of an interface to a client dynamically
- Design decision that can be realized any time during the runtime of the system
  - However, usually the binding occurs at start up time of the system (e.g. in the constructor of the interface class)

Use of the Bridge Pattern: Support multiple Database Vendors

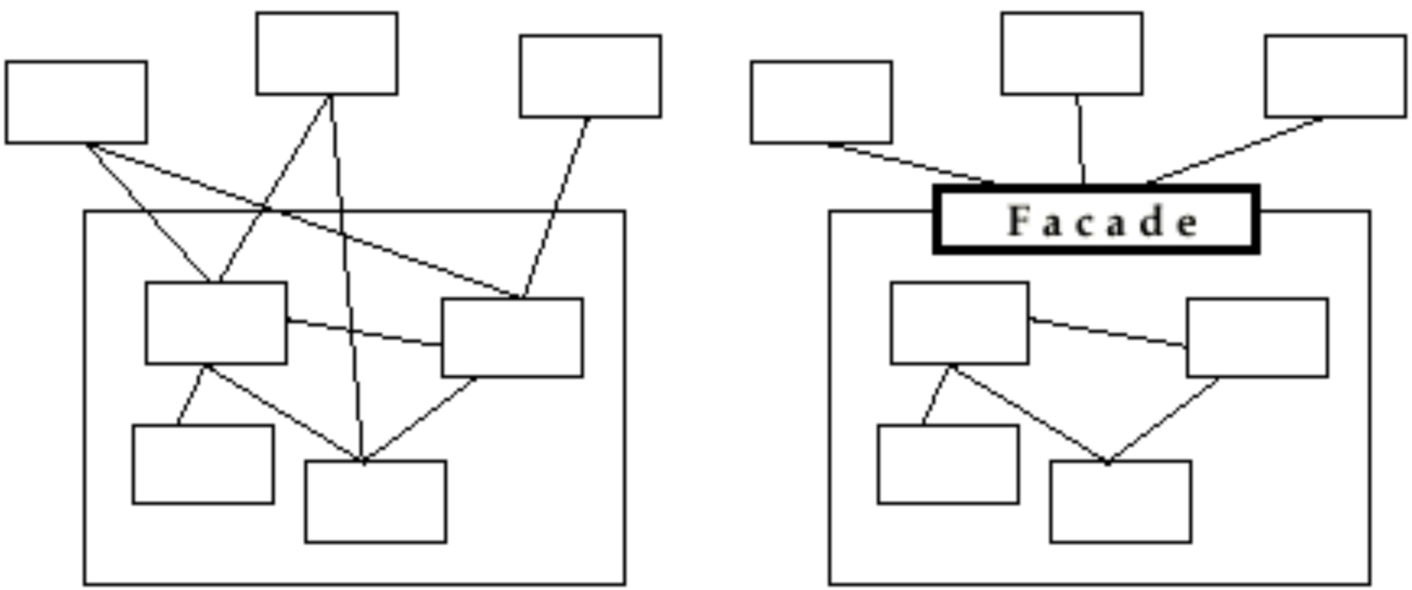


Adapter vs Bridge

- Similarities:
  - Both are used to hide the details of the underlying implementation.
- Difference:
  - The adapter pattern is geared towards making unrelated components work together
    - Applied to systems after they're designed (reengineering, interface engineering).
    - "Inheritance followed by delegation"
  - A bridge, on the other hand, is used up-front in a design to let abstractions and implementations vary independently.
    - Green field engineering of an "extensible system"
    - "Delegation followed by inheritance"

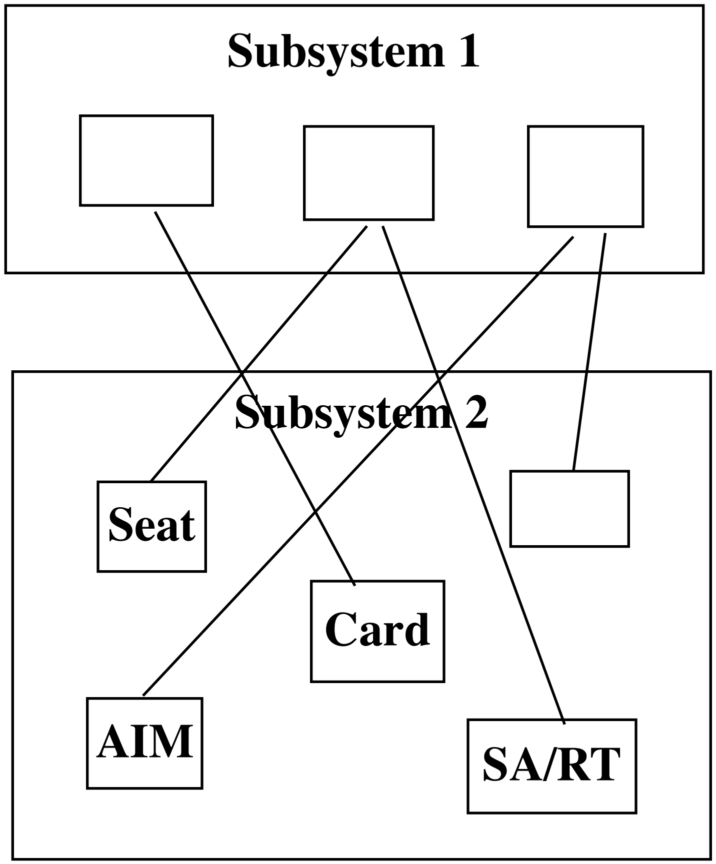
Facade Pattern

- Provides a unified interface to a set of objects in a subsystem.
- A facade defines a higher-level interface that makes the subsystem easier to use (i.e. it abstracts out the gory details)



Design Example

- Subsystem 1 can look into the Subsystem 2 (vehicle subsystem) and call on any component or class operation at will.
- This is “Ravioli Design”
- Why is this good?
  - Efficiency
- Why is this bad?
  - Can’t expect the caller to understand how the subsystem works or the complex relationships within the subsystem.
  - We can be assured that the subsystem will be misused, leading to non-portable code



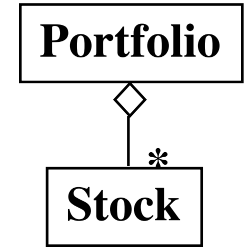
Subsystem Design with Façade, Adapter, Bridge

- The ideal structure of a subsystem consists of
  - an interface object
  - a set of application domain objects (entity objects) modeling real entities or existing systems
    - Some of the application domain objects are interfaces to existing systems
  - one or more control objects
- We can use design patterns to realize this subsystem structure
- Realization of the Interface Object: Facade
  - Provides the interface to the subsystem
- Interface to existing systems: Adapter or Bridge
  - Provides the interface to existing system (legacy system)
  - The existing system is not necessarily object-oriented!

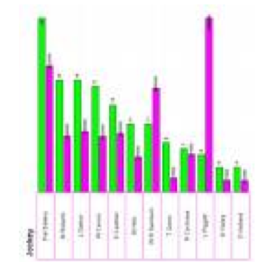
When should you use these Design Patterns?

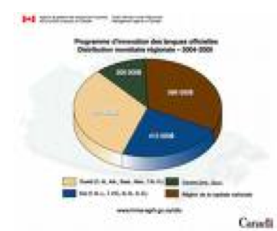
- A façade should be offered by all subsystems in a software system with services
- The adapter design pattern should be used to interface to existing components
- The bridge design pattern should be used to interface to a set of objects
  - where the full set of objects is not completely known at analysis or design time.
  - when a subsystem or component must be replaced later after the system has been deployed and client programs use it in the field.

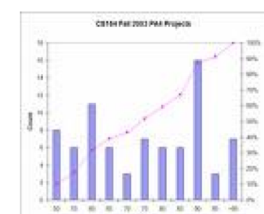
Observer Pattern Motivation



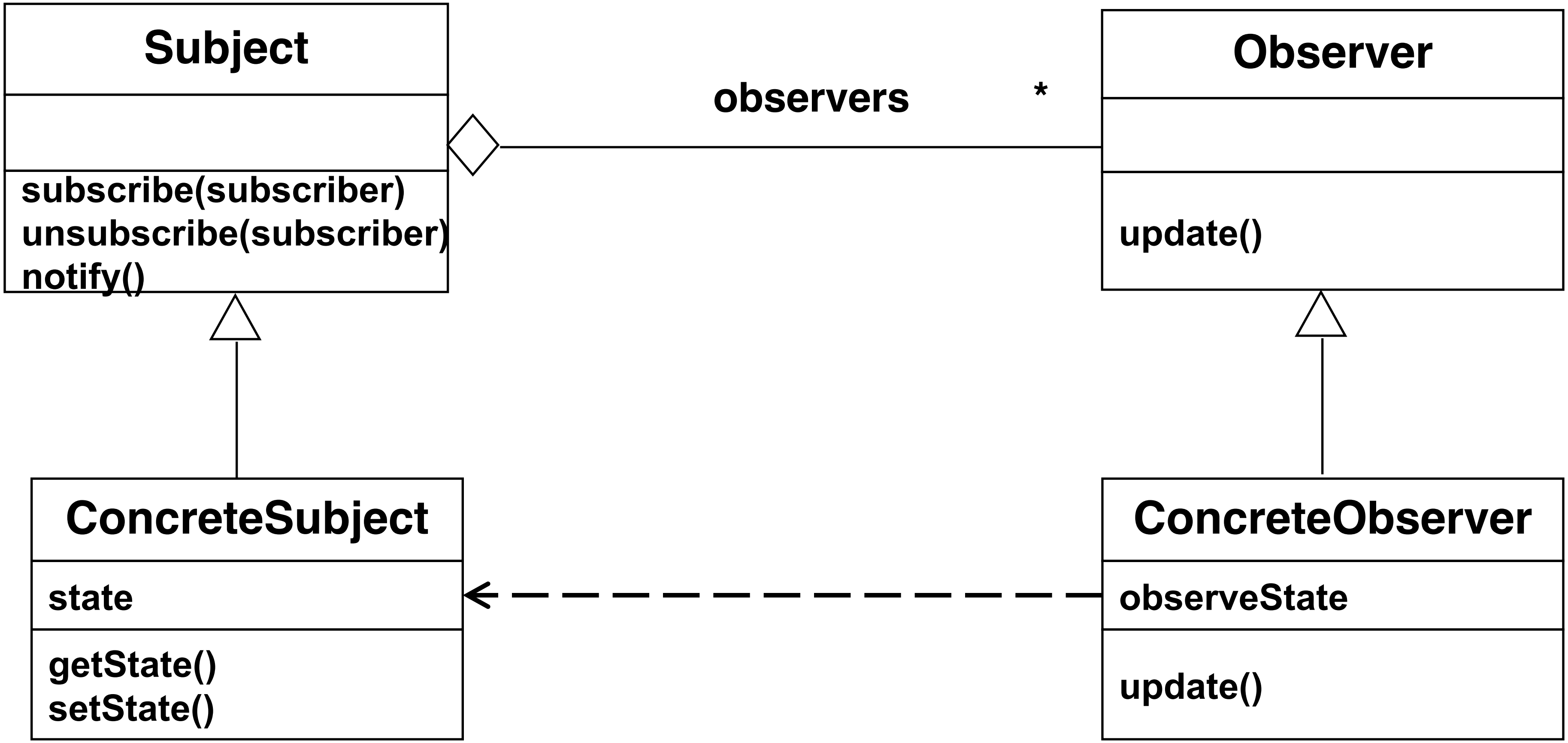
- Problem:
  - We have an object that changes its state quite often
    - Example: A Portfolio of stocks
  - We want to provide multiple views of the current state of the portfolio
    - Example: Histogram view, pie chart view, time line view, alarm
- Requirements:
  - The system should maintain consistency across the (redundant) views, whenever the state of the observed object changes
  - It should be possible to add new views without having to recompile the observed object or the existing views.







Observer Pattern: Decouples an Abstraction from its Views



- The **Subject** ("Publisher") represents the entity object
- **Observers** ("Subscribers") attach to the Subject by calling **subscribe()**
- Each Observer has a different view of the state of the entity object

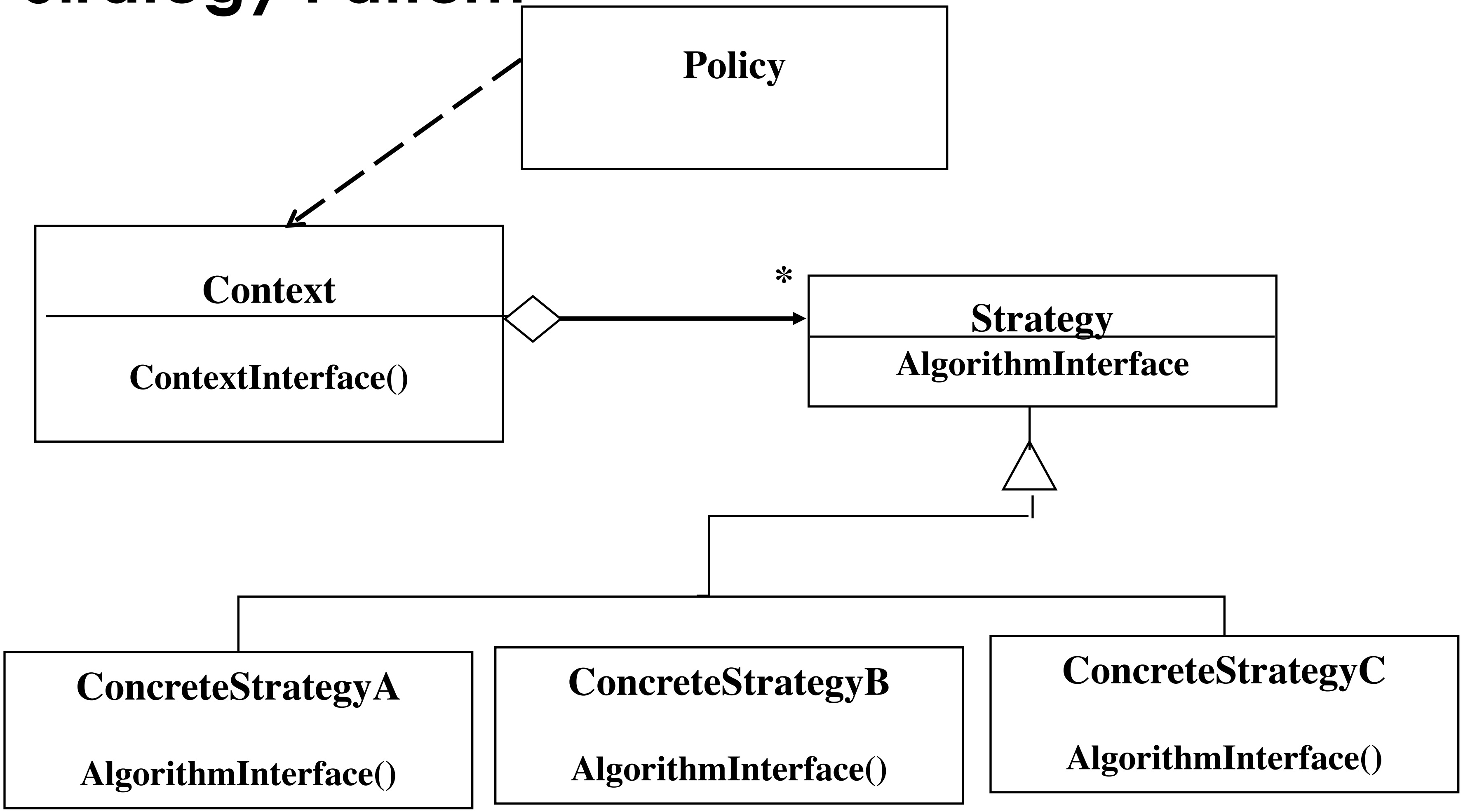
Observer Pattern

- Models a 1-to-many dependency between objects
  - Connects the state of an observed object, the subject with many observing objects, the observers
- Usage:
  - Maintaining consistency across redundant states
  - Optimizing a batch of changes to maintain consistency
- Three variants for maintaining the consistency:
  - Push Notification: Every time the state of the subject changes, *all* the observers are notified of the change
  - Push-Update Notification: The subject also sends the state that has been changed to the observers
  - Pull Notification: An observer inquires about the state the of the subject
- Also called Publish and Subscribe.

Strategy Pattern

- Different algorithms exists for a specific task
  - We can switch between the algorithms at run time
- Examples of tasks:
  - Different collision strategies for objects in video games
  - Parsing a set of tokens into an abstract syntax tree (Bottom up, top down)
  - Sorting a list of customers (Bubble sort, mergesort, quicksort)
- Different algorithms will be appropriate at different times
  - First build, testing the system, delivering the final product
- If we need a new algorithm, we can add it without disturbing the application or the other algorithms.

Strategy Pattern



Policy decides which ConcreteStrategy is best in the current Context.

Using a Strategy Pattern to Decide between Algorithms at Runtime