Design Patterns (Part 2)

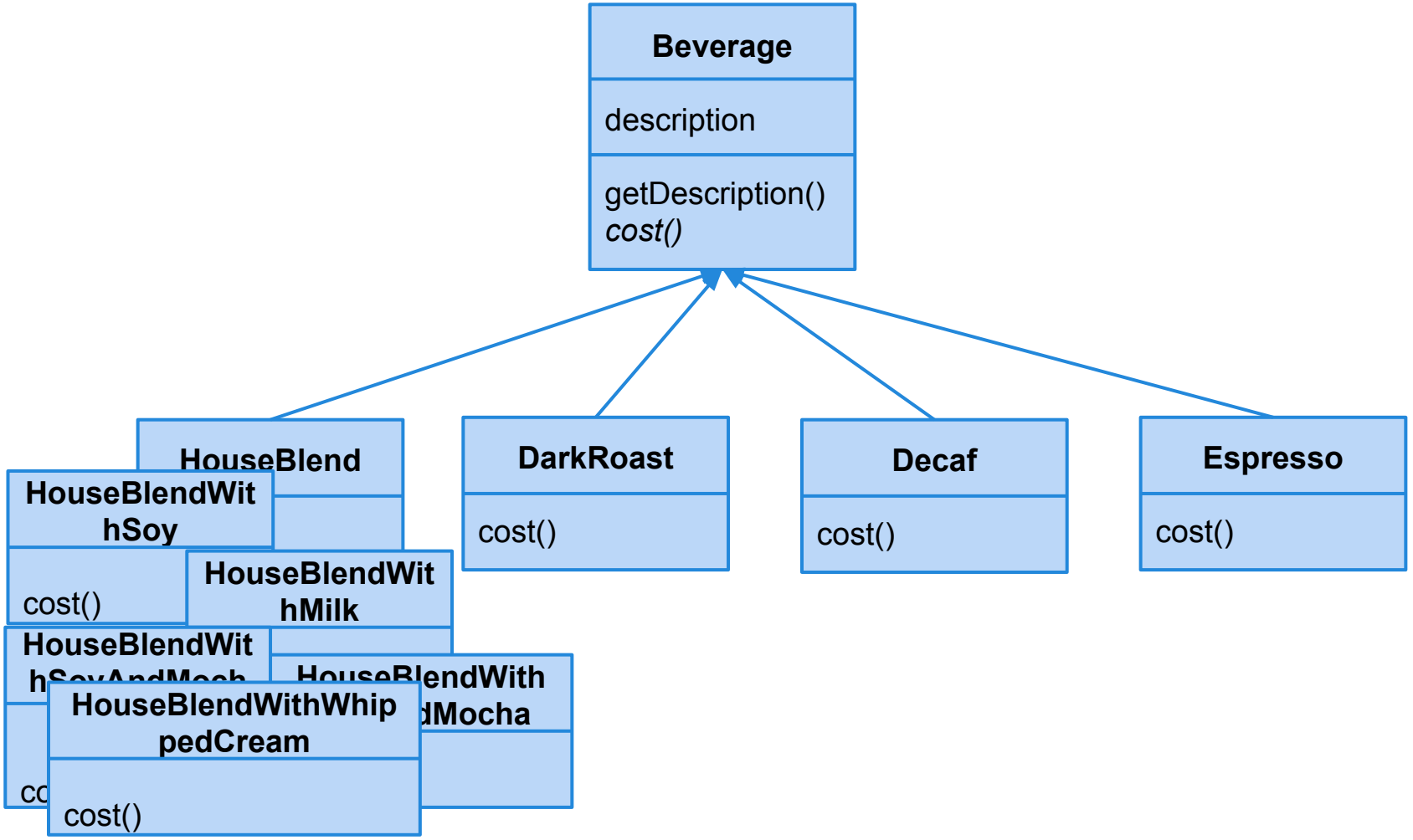
CSCE 740 - Lecture 18 - 11/02/2015

(Partially adapted from Head First Design Patterns by Freeman, Bates, Sierra, and Robson)

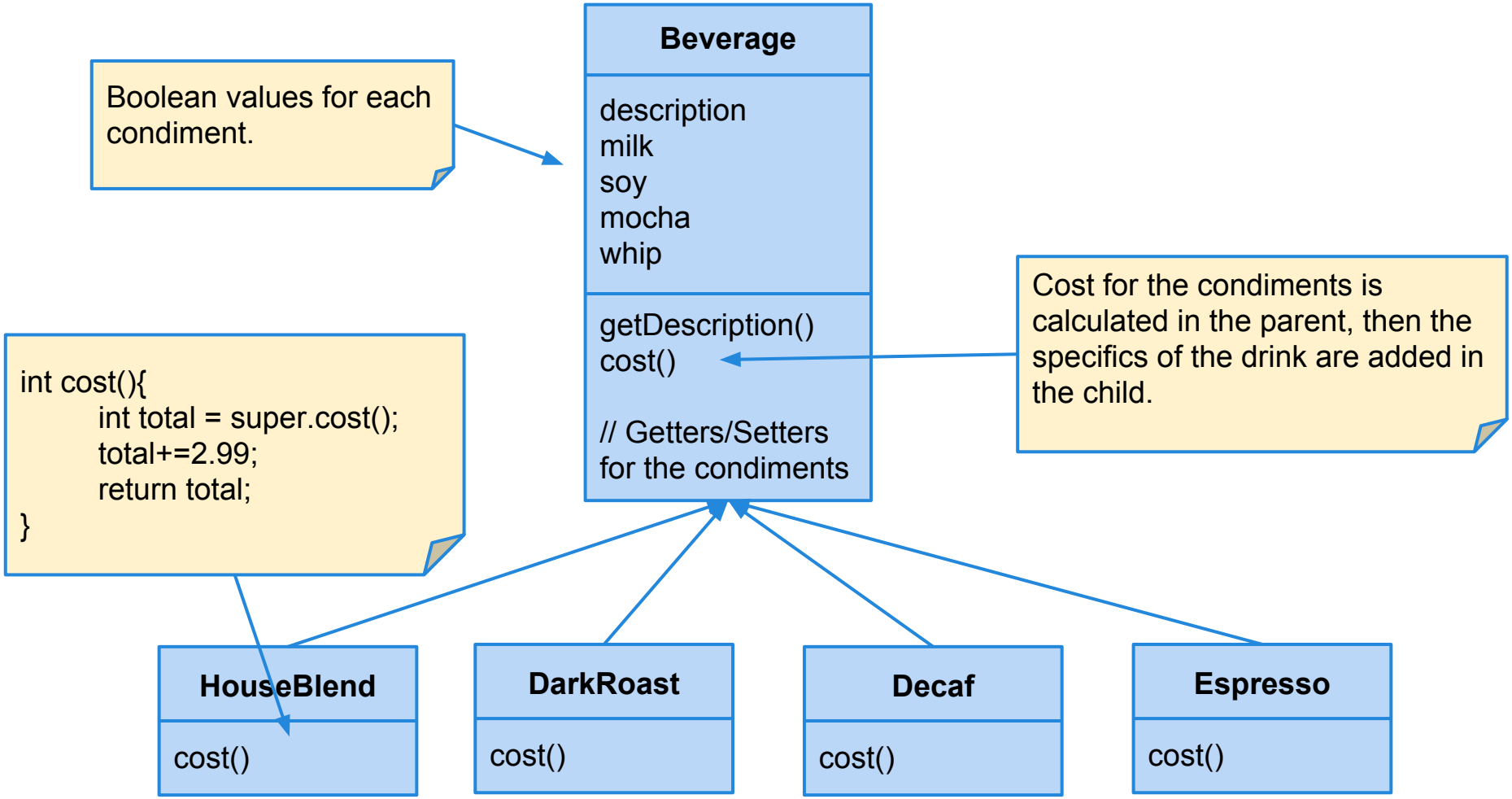
Objectives for Today

- The point of OO: Separate what changes from what doesn't.
  - Easy to say, hard to do.
- Design patterns prescribe ways to structure your design to ensure this separation.
  - Strategy pattern encapsulates behaviors as classes and assign them to the appropriate owner.
  - Visitor pattern enables changes to operations performed over data without modifications to the data classes.
  - Factory pattern encapsulates object creation so that the system doesn't need to know what type of object was created.
- Today - more design patterns.

The Coffee Shop Ordering System



Ordering System - Take 2



How Code Reuse is Achieved

- Inheritance allows us to write code once and reuse it in the children.
  - Good - changes only need to be made once (in theory).
  - Bad - leads to maintenance issues and inflexible design.
    - Must inherit all behaviors of the parent. Might have to write code in the child to work around inherited features.
- Code can also be reused through composition.

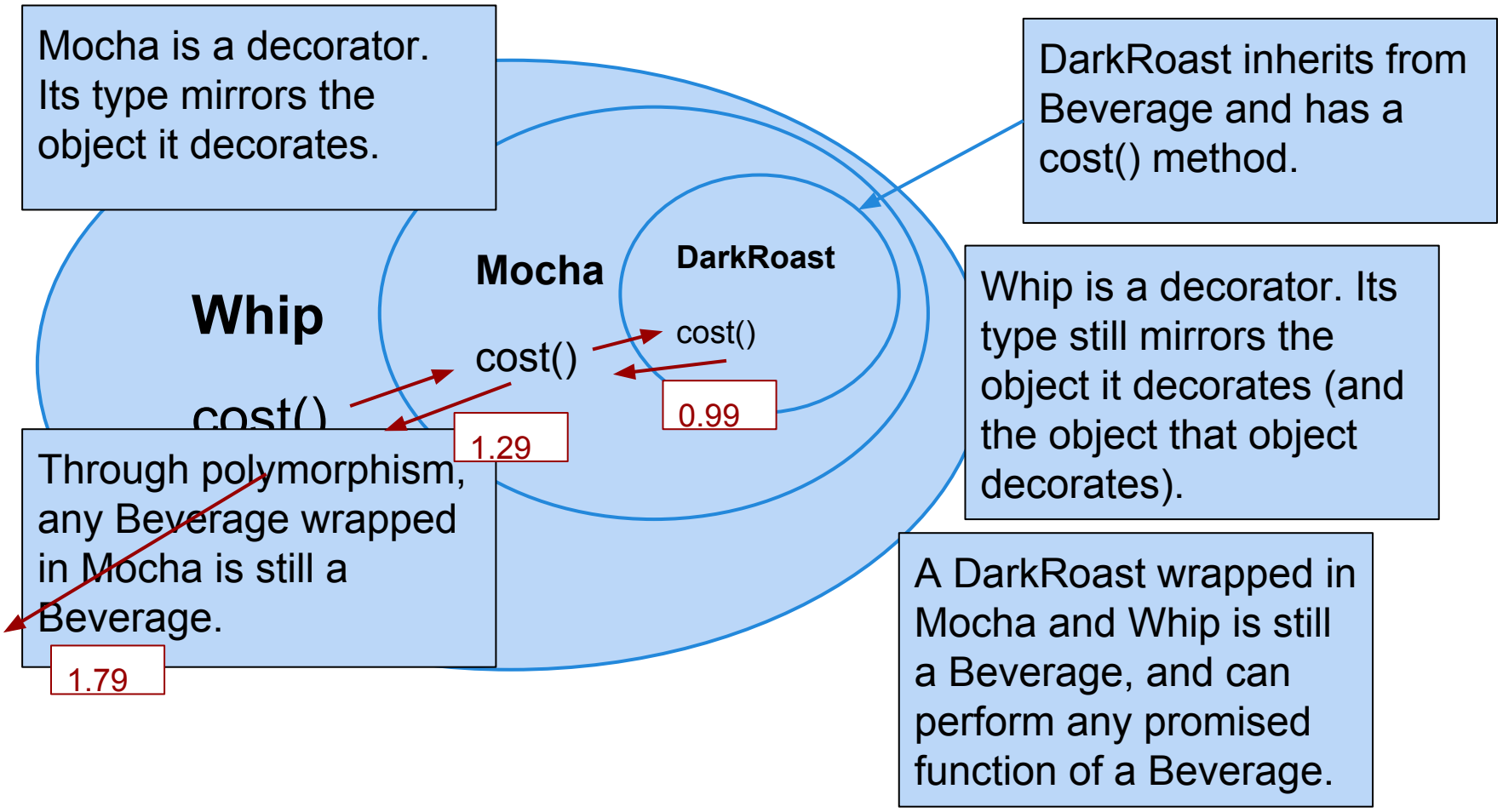
Composition

- We can “attach” an object to another object to add behaviors and attributes to it.
  - All Ducks have some form of flying behavior.
  - We implement each type of flying behavior as a class.
  - We attach the appropriate one at object creation.
- Behavior extension can be done at runtime.
  - We can dynamically change the abilities and responsibilities of objects as the system runs.
- Allows changes to a class while never changing the code of that class.

The Open-Closed Principle

- Classes should be open for extension, but closed for modification.
  - Feel free to extend the class with new behavior, but we spent a lot of time getting the code right, so don't change what it already there.
- Allow change to the system without direct modification.
- **Do not try to apply this everywhere**, but can be important for protecting critical parts of your system.

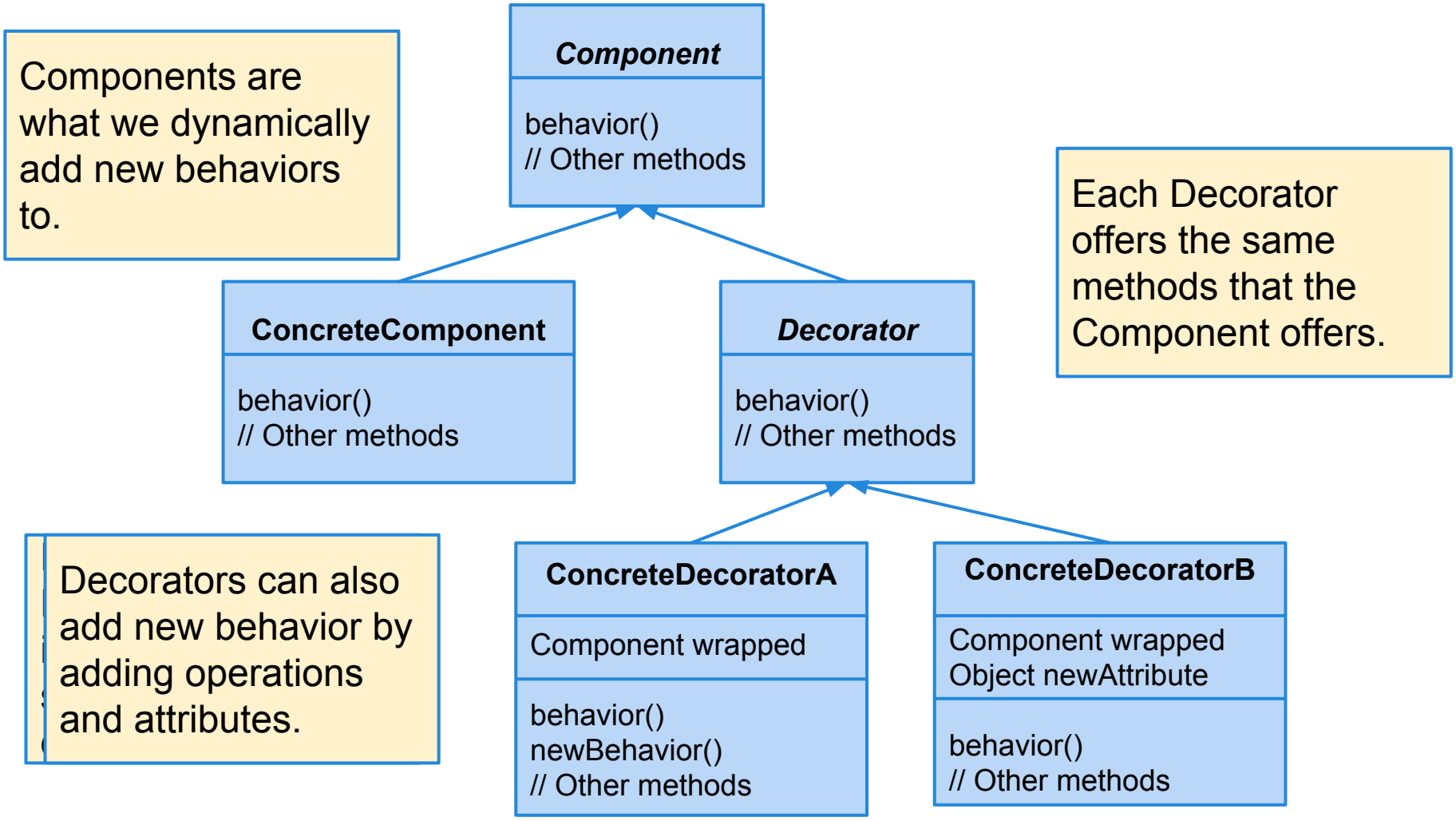
The Decorator Pattern



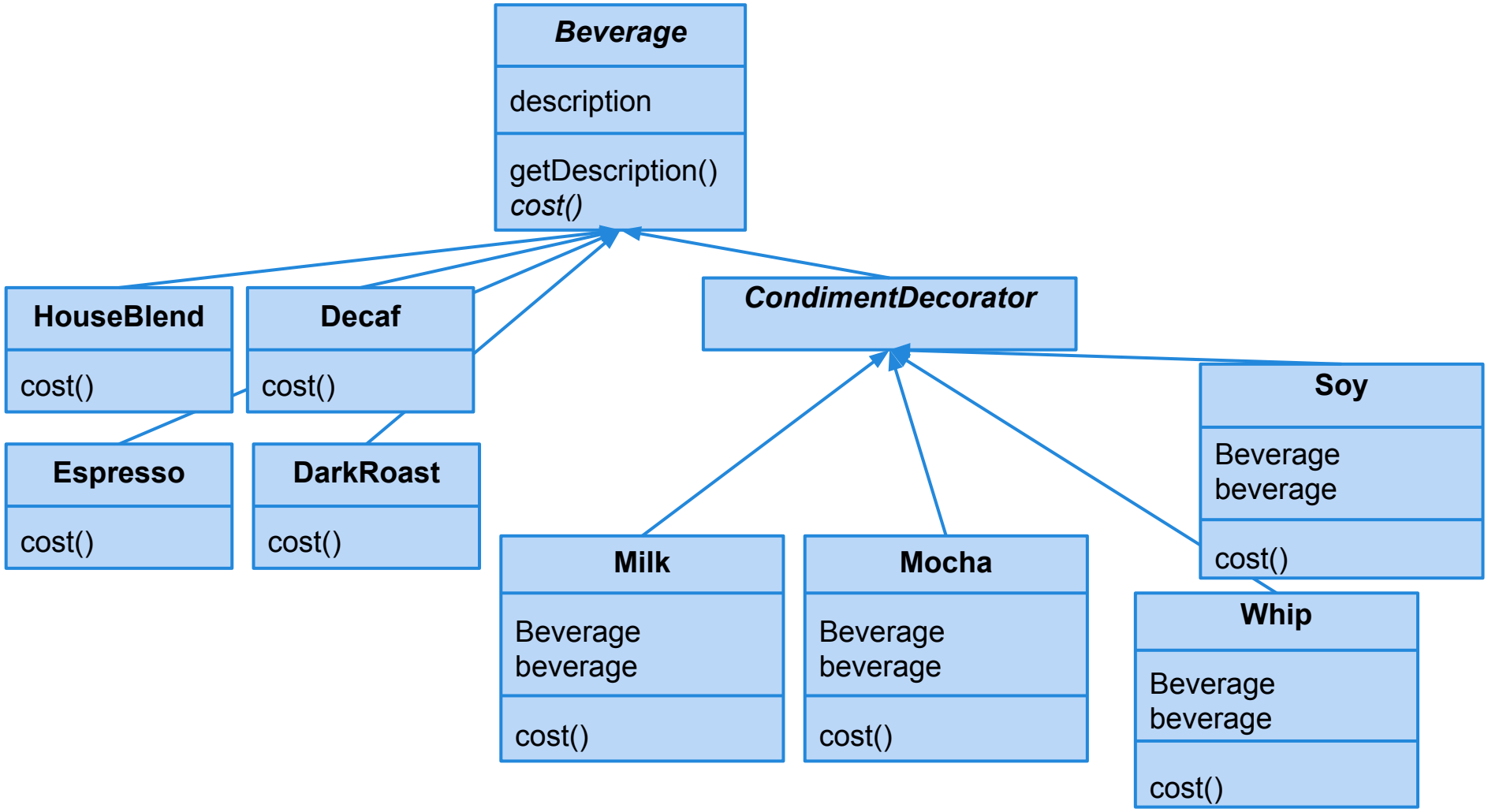
The Decorator Pattern Defined

- The Decorator Pattern attaches additional responsibilities to an object dynamically.
- Decorators provide a flexible alternative to subclassing for extending functionality.
  - Decorators have the same supertype as the objects they decorate.
  - You can use one or more decorators to wrap an object
  - We can pass a decorated object in place of the original object.
  - The decorator adds its own behavior before or after asking the wrapped object to do the rest of the job.

The Decorator Pattern



Ordering System - Take 3



The Decorator Pattern

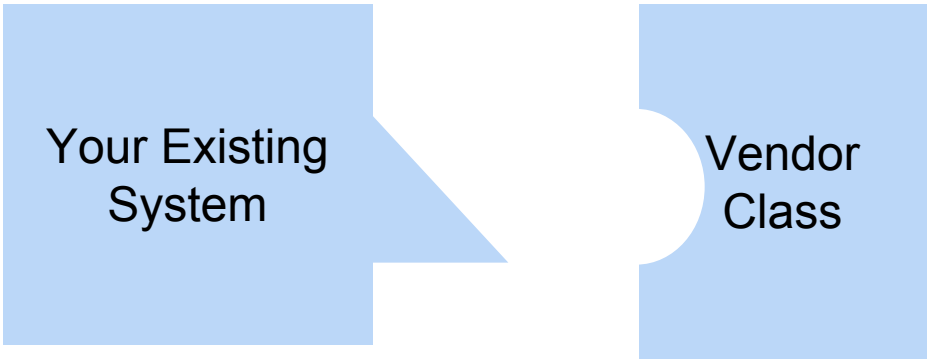
- The Decorator Pattern uses inheritance to achieve *type matching*, but not to inherit behavior.
- By composing a decorator with a component, we add new behavior.
- Composition adds flexibility to how we mix and match behaviors.
  - Can reassign decorators at runtime.
  - Can add new behaviors by writing a new decorator without changing the component.

Decorator Pattern Negatives

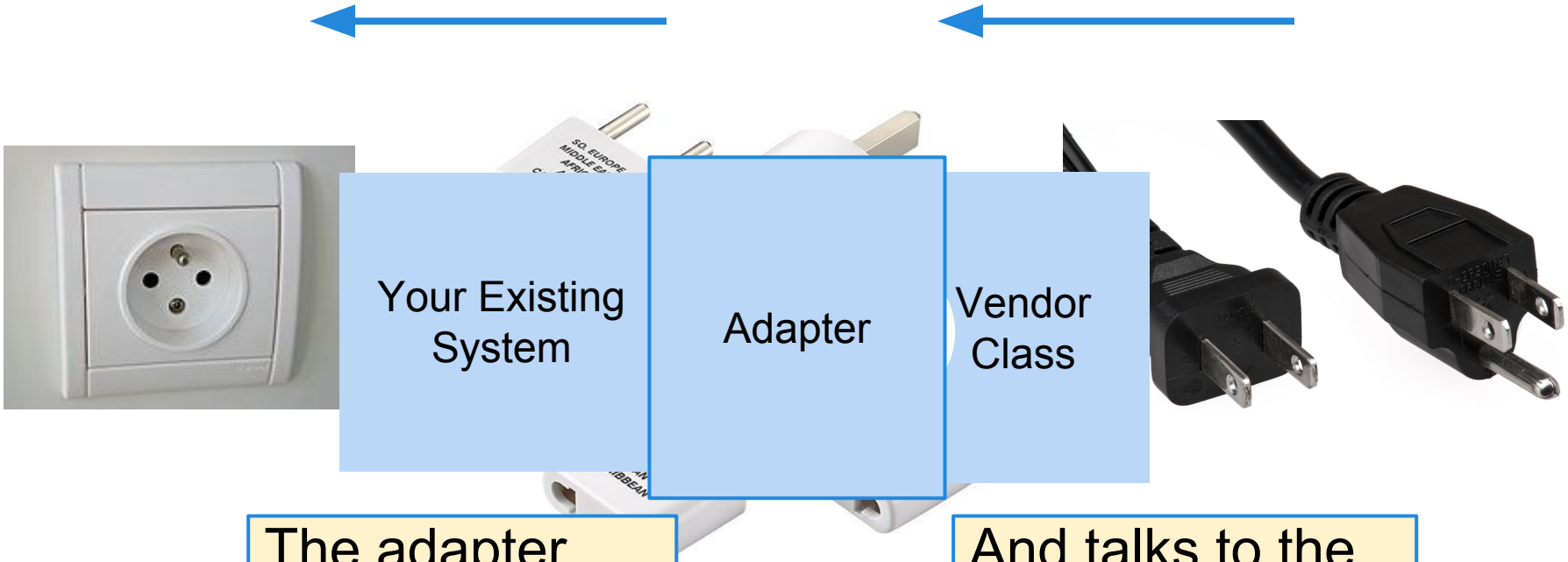
- Decorator Pattern often results in a large number of small classes.
  - Resulting in a design that is harder to understand and find information in.
- Potential type issues.
  - If code does not need to know the specific type, decorators can be used transparently (everything is a Beverage).
  - If code does need the type (any DarkRoast gets a discount), then bad things happen once decorators are applied)

Working With Other Systems

- Often, you will want to bring in services or code from another system so you don't have to write it yourself.
  - However, their interface may not match the one your code uses.



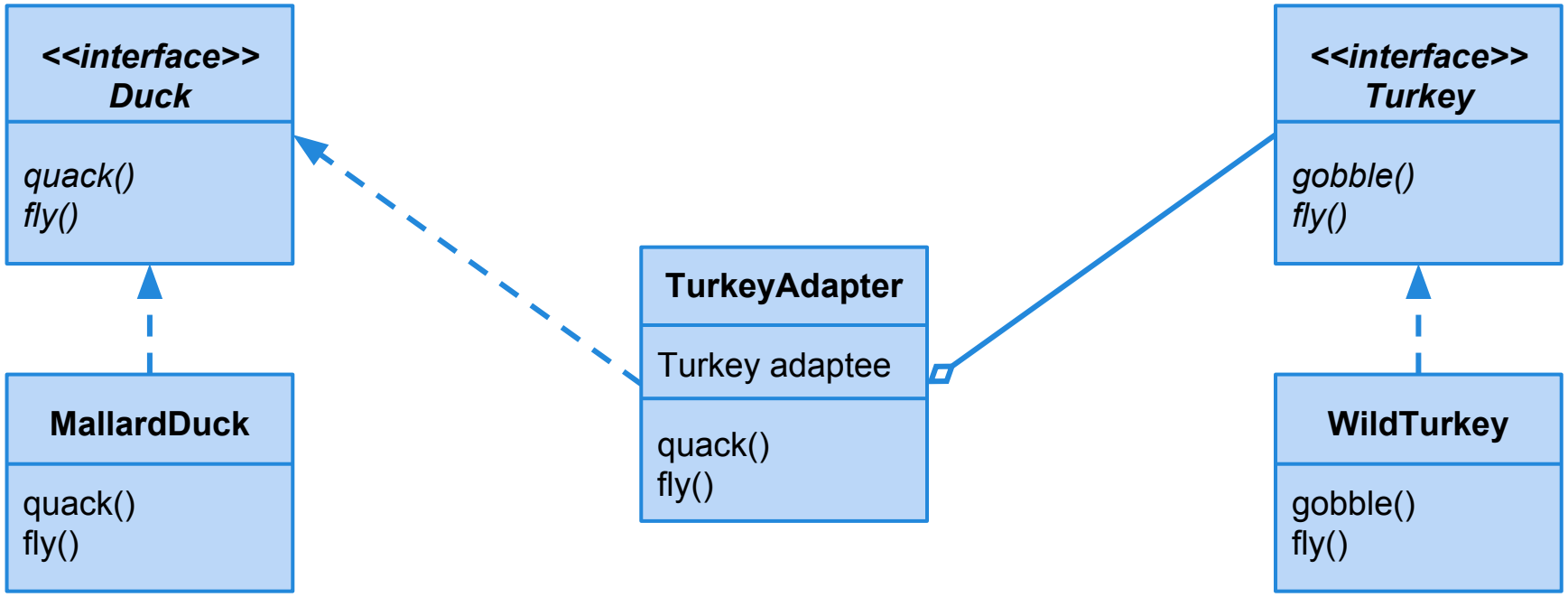
Adapters



The adapter implements the interface your class expects.

And talks to the vendor interface to service requests.

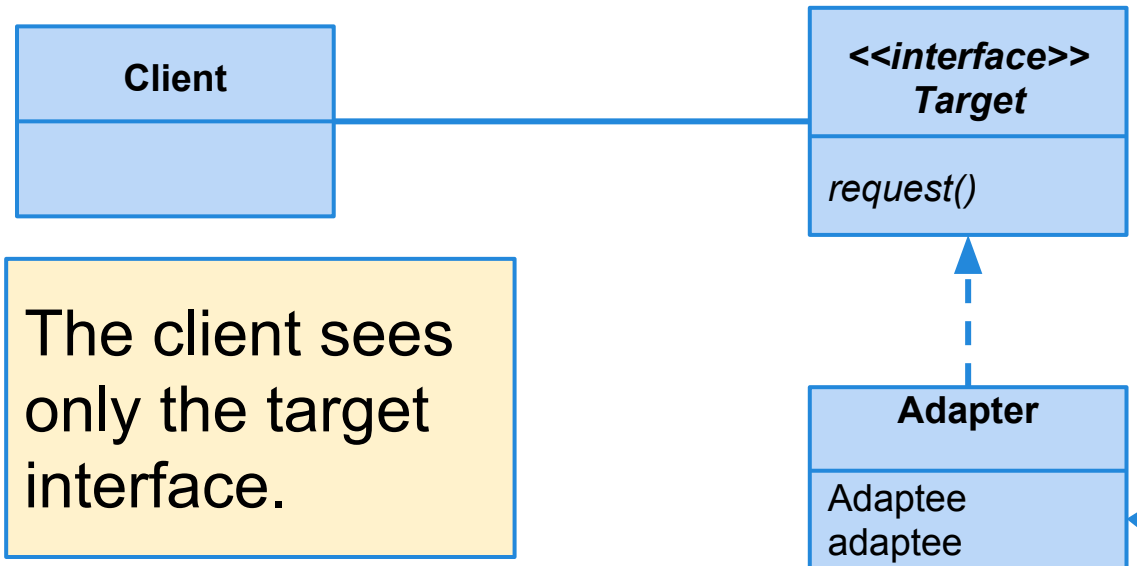
Adapter Example



The Adapter Pattern Defined

- The Adapter Pattern converts the interface of a class into another interface the client expects.
- Adapters let classes work together that couldn't otherwise do so due to incompatible interfaces.
- Adapters can wrap multiple adaptees together when we need multiple objects to provide the services the client requires.

The Adapter Pattern



The client sees only the target interface.

The Adapter implements the target interface.

The Adapter is composed with the Adaptee.

All requests get delegated to the Adaptee.

Watching a Movie

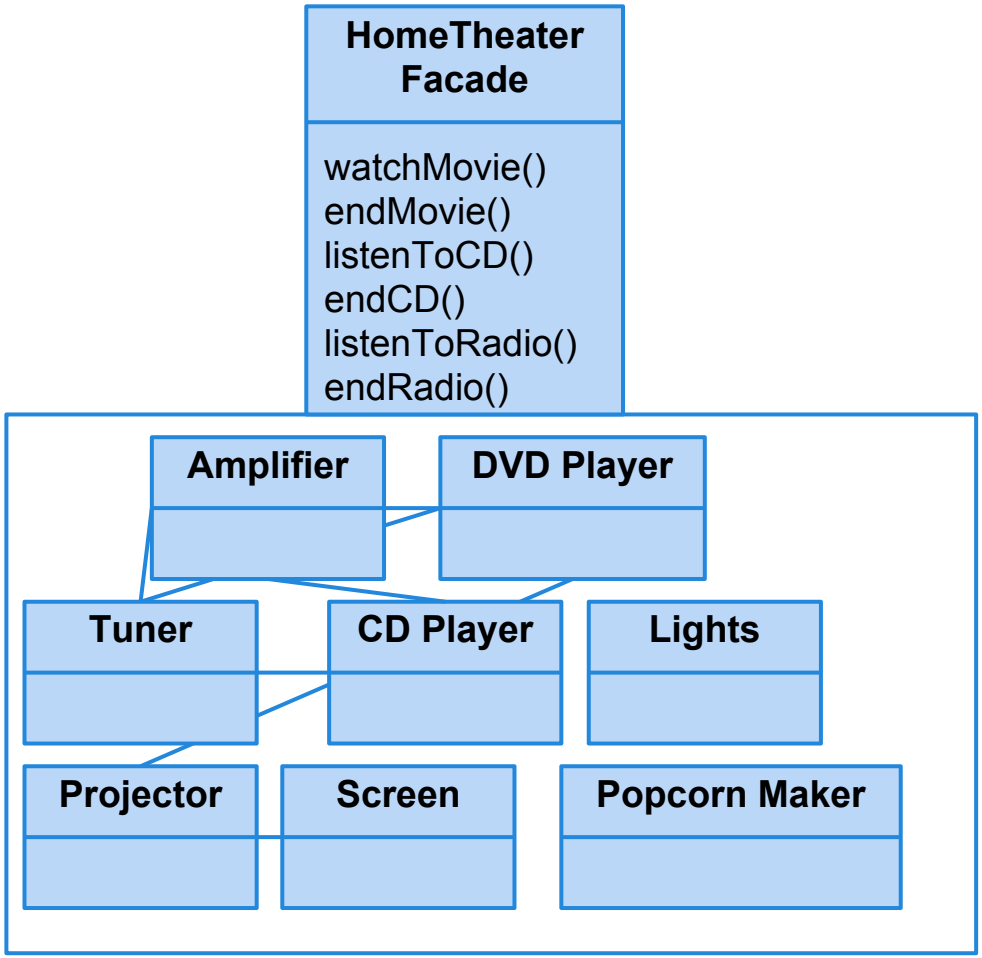
To watch a DVD, we need to perform a few tasks:

1. Turn on the popcorn popper.
2. Start the popper.
3. Dim the lights.
4. Put the screen down.
5. Turn the projector on.
6. Set the projector input to DVD.
7. Put the projector on widescreen mode.
8. Turn the sound amplifier on.
9. Set the amplifier to DVD input.
10. Set the amplifier to surround sound.
11. Set the amplifier volume to medium.
12. Turn the DVD player on.
13. Start the DVD.

Wrapping Classes

- The Adapter Pattern converts the interface of a class into one the client is expecting.
- The Decorator Pattern doesn't alter an interface, but wraps classes in new functionality.
- The Facade Pattern simplifies interactions by hiding complexity behind a clean, easy-to-understand interface.
  - Wrapping classes into a shared interface.

The Facade Pattern

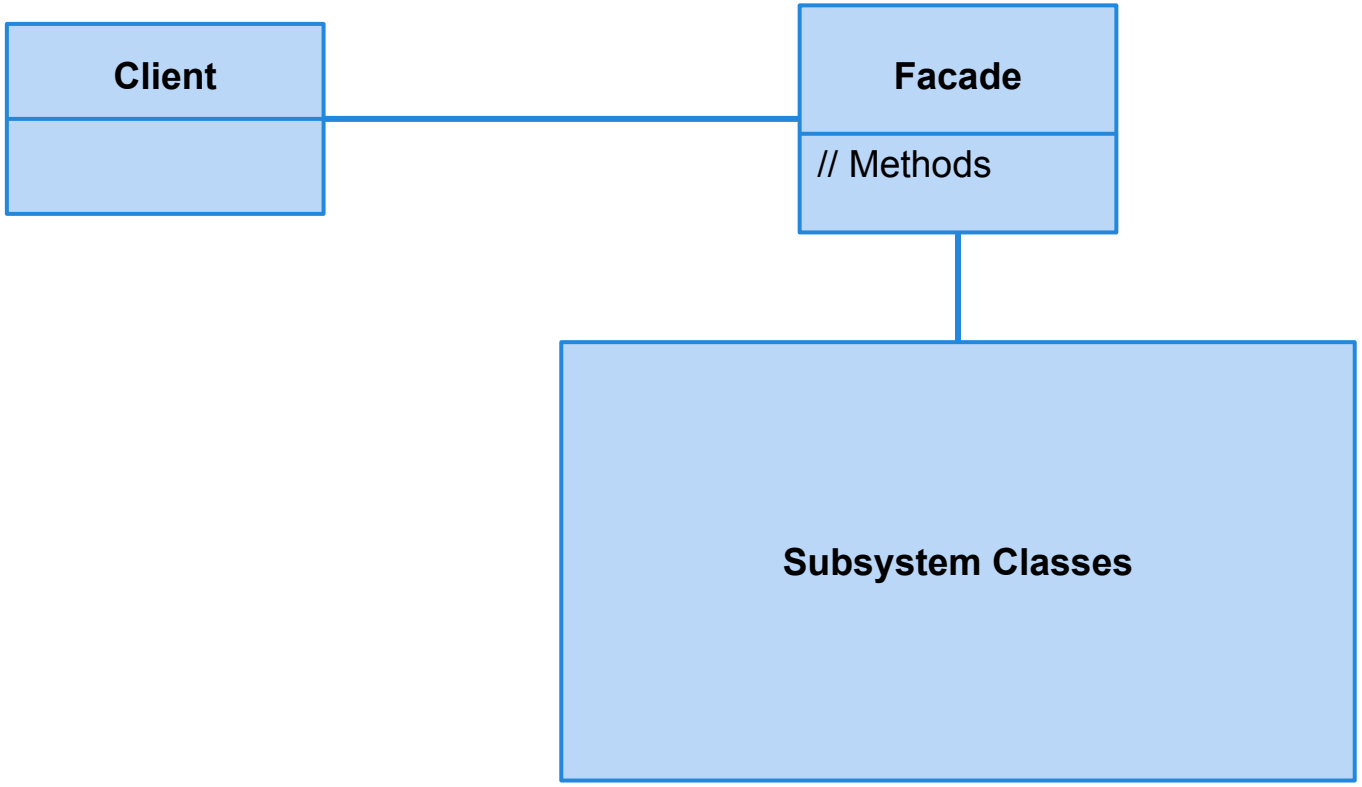


1. Create a new class HomeTheaterFacade that exposes a few simple methods.
2. Facade treats the home theater components as a subsystem and call on the subsystem to implement its methods.
3. Client code calls methods on the facade instead of the subsystem.
4. Facade still leaves the subsystem accessible to use directly.

The Facade Pattern Defined

- The Facade Pattern provides a unified interface to a set of interfaces in a subsystem.
- Facade defines a higher-level interface that makes the subsystem easier to use.
  - No encapsulation - the lower levels are still accessible. Merely provides another method of access.
  - Multiple facades may be defined for the same subsystem to provide different situational functions.
  - Decouples the client from any one subsystem.

The Facade Pattern



The Principle of Least Knowledge

- Principle - **talk only to your immediate friends.**
- When designing a class, be careful of the number of classes it interacts with and how it comes to interact with them.
- Only invoke methods that belong to the object itself, objects passed in as parameters, objects the method creates or instantiates, and components of the object.

The Home Automation Remote

- We want to develop a remote control API for controlling a variety of home appliances.
  - ceiling lights, outdoor lights, fan, TV, garage door, faucets, thermostat, hot tub, security, sprinkler, etc.
  - The remote should be able to turn any device on or off, and there is an undo button to negate the last action taken.
  - The remote has seven slots, and each slot can be replaced at any time with another appliance.
- The classes for each appliance were not developed with the same interface, and are completely independent of each other.
- **Does the Facade Pattern make sense for designing the remote control API?**

Why Not Facade?

- Facade presents a simplified interface for a single subsystem.
- In this case, we are designing an interface for several independent subsystems.
  - Where those subsystems can be replaced at any time.
  - … And we only want to provide access to seven at once.
  - … And we need detailed information on how to use each since they don't provide a common interface.
- We need a different approach. **How should we design this remote?**

The Command Pattern

- The Command Pattern decouples the requester of an action from the object that performs the action.
  - Command objects are introduced into the design that encapsulate a method call on an object.
  - When a remote button is pressed, it asks the command object to do some work.
  - The requester is decoupled from the object doing the work.
- Command objects can be used to add additional functionality to objects.
  - Logging, Action Queueing, Undo

Ordering Food

The Customer creates an Order

createOrder()

The Order consists of an order slip with menu items written on it.

The Waitress takes the Order and sends it to the kitchen.

takeOrder()

orderUp()

The Order has all instructions for making the meal.

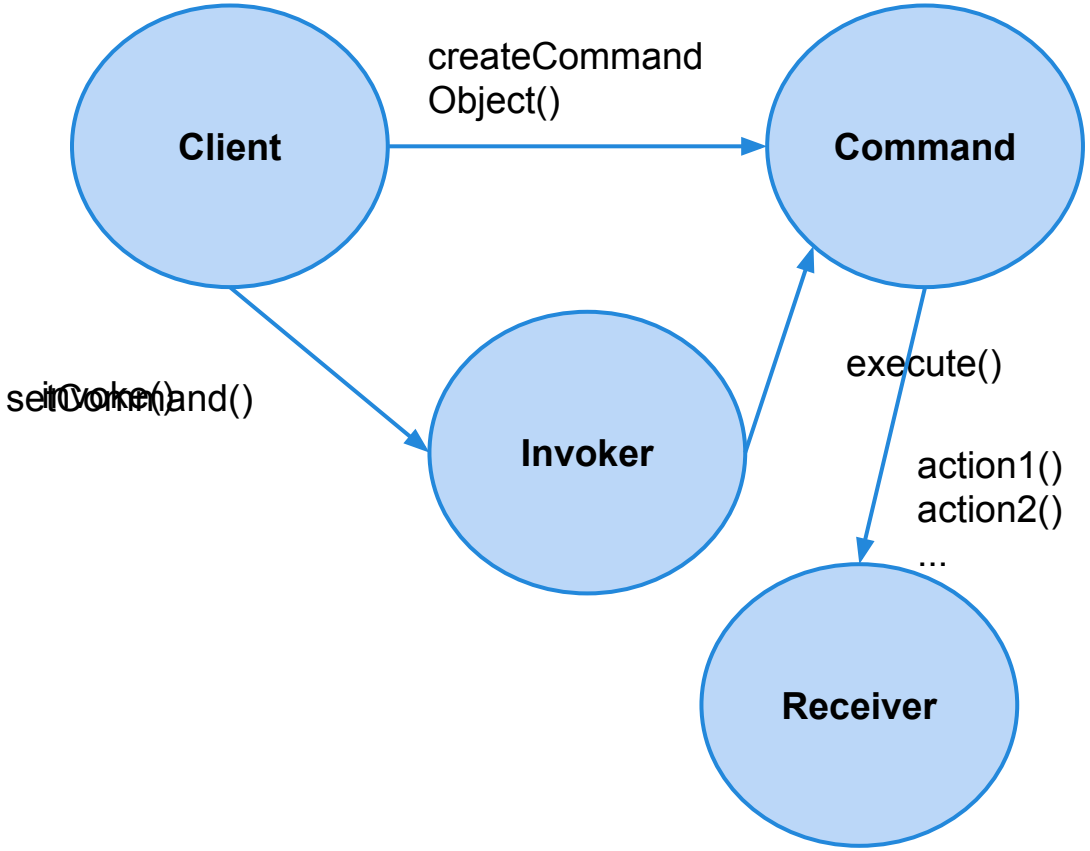
makeBurger(), makeShake()

The Cook follows the instructions from the Order.

Ordering Food

- An Order Slip encapsulates a request to prepare a meal.
  - Object that acts as a request to prepare a meal.
    - Can be passed around like any object.
    - References the object needed to prepare the meal.
  - Encapsulated such that Waitress doesn't need to know what is in the meal or who prepares it.
- The Cook has the knowledge required to prepare the meal.
  - Waitress and Cook are completely decoupled.

The Command Pattern



Command object consists of a set of

Later, the Client asks

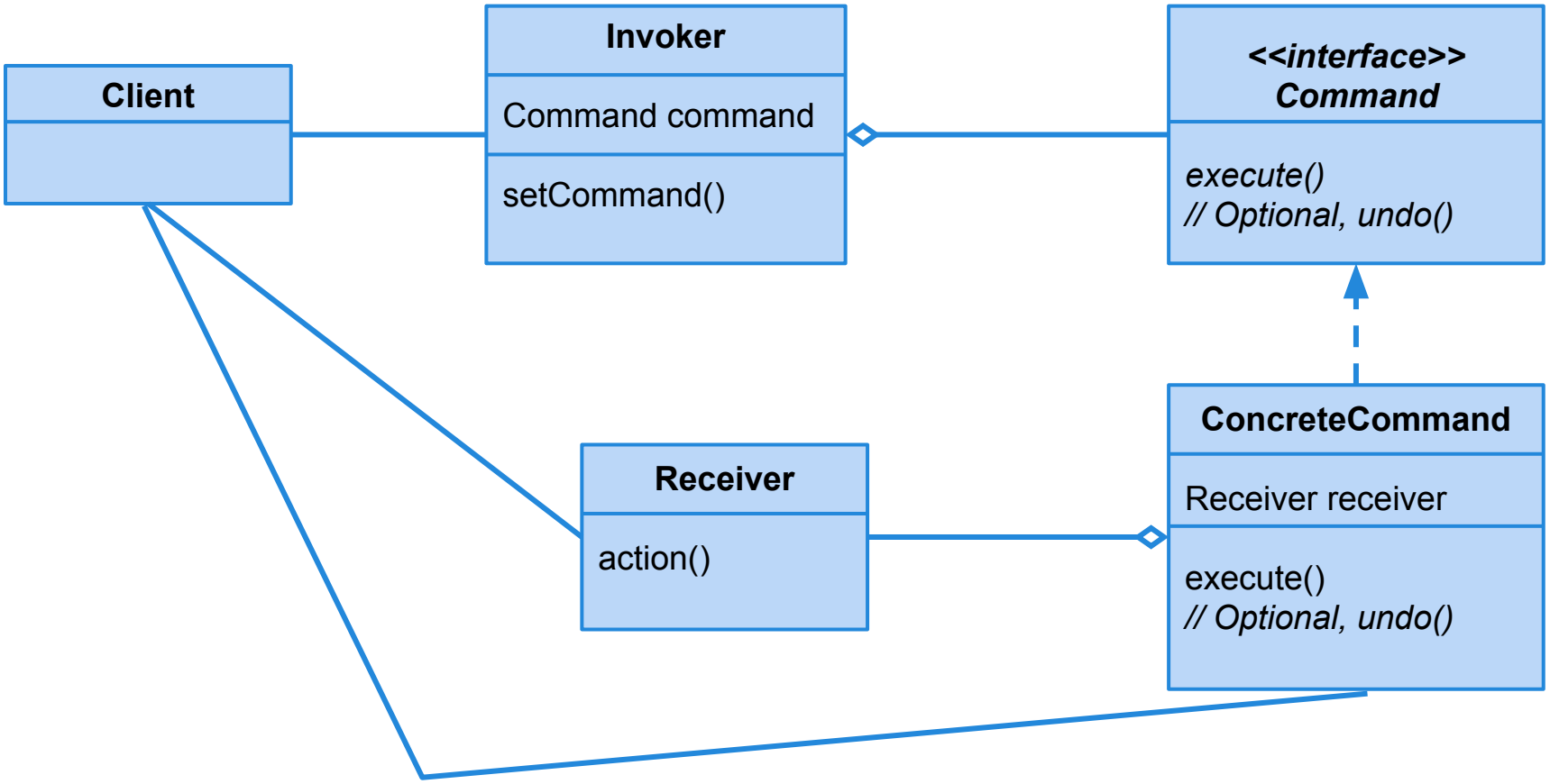
The Command steps through the list of actions and has the Receiver perform them.

object to the Invoker.

The Command Pattern Defined

- The Command Pattern encapsulates a request as an object, allowing control over *how requests are performed*.
  - Command objects encapsulate requests by binding a set of actions on a specific Receiver.
  - Objects being parameterized don't care what commands they have as long as they offer the same interface.
  - This encapsulation can add functionality that the Receiver does not natively support.

The Command Pattern



Tip - NoCommand

What if you don't have enough appliances to fill all seven remote slots?

- Implement a command that does nothing.
- An example of a **null object**.
  - Useful when you don't have anything meaningful to return, but want to avoid having to implement a check for null.

```
public void buttonPushed(int slot){
    if(commands[slot] != null){
        commands[slot].execute();
    }
}
```

```
public class NoCommand implements
Command{
    public void execute() { }
}
```

Using State to Implement Undo

- Undoing an operation requires keeping track of state information in the Command object.
- In the CeilingFan - keep track of previous speed and revert to it.
- Keep a stack of states to enable multiple undo presses.

```
public class CeilingFanHighCommand
implements Command{
     CeilingFan fan;
     int prevSpeed;

     public CeilingFanHighCommand
(CeilingFan cf){ fan = cf;     }

     public void execute(){
          prevSpeed = fan.getSpeed();
          fan.high();
     }

     public void undo(){
          if(prevSpeed == "high")
               fan.high();
          else if ...
     }
}
```

Queueing Requests

Commands give us a way to package a piece of computation and pass it around as an object.

- Computation can be invoked at any time.
- Computation can be invoked by anything with knowledge of the Command object.
- A job queue could store Commands that are processed by threads as earlier work is completed.
  - Job queue decoupled from the work being completed.

Logging Requests

Some applications recover after a crash by reinvoking the actions already performed.

- Commands can enable this with two new methods - store() and load().
  - As we execute commands, store a history of them on disk. When a crash occurs, reload the list of commands and invoke their execute() methods.
- Useful for applications where many actions might be taken between saving a permanent copy of work.

Principles of Design

1. **Identify the aspects that vary and encapsulate them away from what doesn't.**
2. Program to an interface rather than an implementation.
3. Favor composition over inheritance.
4. Classes should be open for extension, but closed for modification.
5. Talk only to your immediate friends.

Design Patterns

- **Strategy Pattern** encapsulates interchangeable behaviors and uses delegation to decide which one to use.
- **Observer Pattern** allows objects to be notified when state changes.
- **Visitor Pattern** provides a way to traverse a collection of objects without exposing its implementation.
- **Factory Pattern** encapsulates object creation so that the system doesn't need to know what type of object was created.

Design Patterns

- **Decorator Pattern** wraps an object to provide new behavior.
- **Adapter Pattern** wraps an object and provides a different interface to it.
- **Facade Pattern** simplifies the interface of a set of classes.
- **Command Pattern** encapsulates a request as an object.

Next Time

- Interaction Diagrams
  - Modeling dynamic behavior of objects.
  - UML sequence diagrams
- Reading
  - Sommerville, chapter 5, 7
  - Fowler, chapter 4
- Homework
  - Questions on class diagrams?