Object-Oriented Programming Using Java

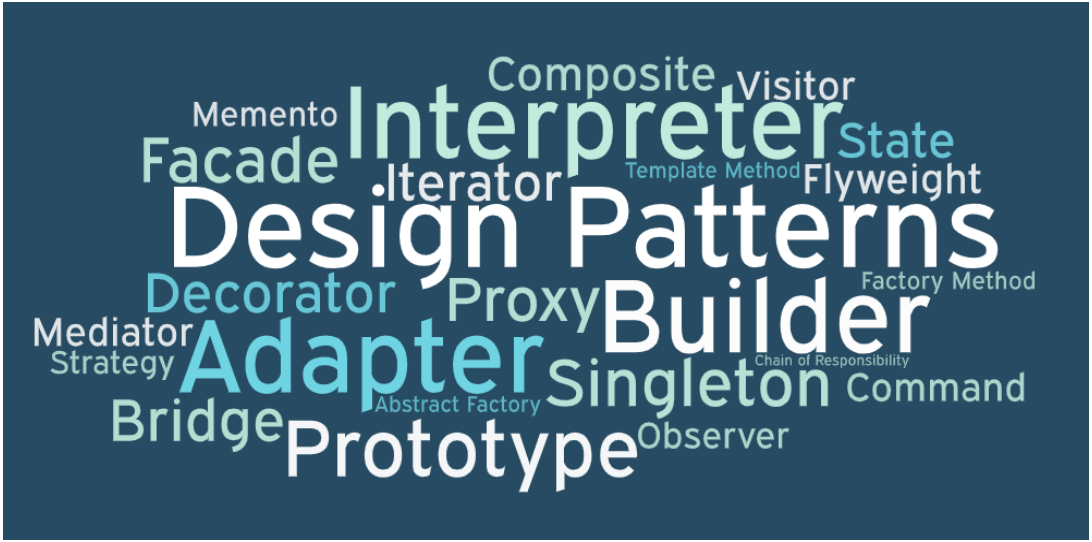
Design Patterns

Quan Thai Ha

HUS

2024


DESIGN PATTERNS
A MUST HAVE SKILL FOR
SOFTWARE DEVELOPERS


Composite
Visitor
Memento
Interpreter
State
Facade
Iterator
Template Method
Flyweight
Design Patterns
Decorator
Proxy
Builder
Factory Method
Mediator
Strategy
Adapter
Chain of Responsibility
Singleton
Command
Abstract Factory
Bridge
Prototype
Observer

Design Patterns are Everywhere

- In 1995, a book was published by the “Gang of Four” called Design Patterns.
  - It applied the concept of patterns (discussed next) to software design and described 23 of them.
    - The authors did not invent these patterns. Instead, they included patterns they found in at least 3 “real” software systems.
- Since that time lots of Design Patterns books have been published.
  - And more patterns have been cataloged.
- Unfortunately, many people feel like they should become experts in object-oriented analysis and design before they learn about patterns.
  - The book takes a different stance: learning about design patterns will help you become an expert in object-oriented analysis and design.

Christopher Alexander (I)

- Design patterns in software design traces its intellectual roots to work performed in the 1970s by an architect named Christopher Alexander.
  - His 1979 book called "The Timeless Way of Building" that asks the question "Is quality objective?".
    - In particular, "What makes us know when an architectural design is good? Is there an objective basis for such a judgement?".
  - His answer was "yes" that it was possible to objectively define "high quality" or "beautiful" buildings.
- He studied the problem of identifying what makes a good architectural design by observing all sorts of built structures.
  - Buildings, towns, streets, homes, community centers, etc.
- When he found an example of a high quality design, he would compare that object to other objects of high quality and look for commonalties.
  - Especially if both objects were used to solve the same type of problem.

- By studying high quality structures that solve similar problems, he could discover similarities between the designs and these similarities where what he called patterns.
  - "Each pattern describes a problem which occurs over and over again in our environment and then 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 way twice".
  - The pattern provides an approach that can be used to achieve a high quality solution to its problem.
- Alexander identified four elements to describe a pattern.
  - The name of the pattern.
  - The purpose of the pattern: what problem it solves.
  - How to solve the problem.
  - The constraints we have to consider in our solution.
- He also felt that multiple patterns applied together can help to solve complex architectural problems.

- Work on design patterns got started when people asked:
  - Are there problems in software that occur all the time that can be solved in somewhat the same manner?
  - Was it possible to design software in terms of patterns?
- Many people felt the answer to these questions was “yes” and this initial work influenced the creation of the Design Patterns book by the Gang of Four.
  - It catalogued 23 patterns: successful solutions to common problems that occur in software design.
- Design patterns, then, assert that the quality of software systems can be measured objectively.
  - What is present in a good quality design (X’s) that is not present in a poor quality design?
  - What is present in a poor quality design (Y’s) that is not present in a good quality design?
- We would then want to maximize the X’s while minimizing the Y’s in our own designs.

- As a developer, it is our responsibility to design the code in such a way which allow our code to be flexible, maintainable, and re-usable.
  - The code we have written is flexible enough that we can make any changes to it with less or any pain. Our code is re-usable so that we can re-use it anywhere without any trouble. We can maintain our code easily and any changes to a part of the code will not affect any other part of the code.
- Designing is an art and it comes with the experience. But there are some set of solutions already written by some of the advanced and experienced developers while facing and solving similar designing problems. These solutions are known as Design Patterns.
- The Design Patterns is the experience in designing the object oriented code.
- Design Patterns are general reusable solution to commonly occurring problems. These are the best practices, used by the experienced developers. Patterns are not complete code, but it can use as a template which can be applied to a problem. Patterns are re-usable; they can be applied to similar kind of design problem regardless to any domain. In other words, we can think of patterns as a formal document which contains recurring design problems and its solutions.

- "Descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context."

  [Gamma, Helm, Johnson, Vlissides 1995]

  - **Name**.
  - **Intent**: The purpose of the pattern.
  - **Problem**: What problem does it solve?
  - **Solution**: The approach to take to solve the problem.
  - **Participants**: The entities involved in the pattern.
  - **Consequences**: The effect the pattern has on your system.
  - **Structure**: Class Diagram.
  - **Implementation**: Example ways to implement the pattern.
  - **Sample code**.
  - **Related patterns**.

- **Flexibility**
  - Using design patterns your code becomes flexible. It helps to provide the correct level of abstraction due to which objects become loosely coupled to each other which makes your code easy to change.
- **Reusability**
  - Loosely coupled and cohesive objects and classes can make your code more reusable. This kind of code becomes easy to be tested as compared to the highly coupled code.
- **Shared Vocabulary**
  - Shared vocabulary makes it easy to share your code and thought with other team members. It creates more understanding between the team members related to the code.
- **Capture best practices**
  - Design patterns capture solutions which have been successfully applied to problems. By learning these patterns and the related problem, an inexperienced developer learns a lot about software design.

- Design patterns provide you not with code reuse but with experience reuse.
  - Knowing concepts such as abstraction, inheritance and polymorphism will NOT make you a good designer, unless you use those concepts to create flexible designs that are maintainable and that can cope with change.
- Design patterns can show you how to apply those concepts to achieve those goals.
- Design Patterns give you a higher-level perspective on:
  - The problems that come up in object-oriented analysis and design work.
  - The process of design itself.
  - The use of object orientation to solve problems.
- You'll be able to think more abstractly and not get bogged down in implementation details too early in the process.

- Improved Motivation of Individual Learning in Team Environments.
  - Junior developers see that the design patterns discussed by more senior developers are valuable and are motivated to learn them.
- Improved maintainability.
  - Many design patterns make systems easy to extend, leading to increased maintainability.
- **Design patterns lead to a deeper understanding of core OO principles**.
- They reinforce useful design heuristics such as:
  - Code to an interface.
  - Favor delegation over inheritance.
  - Find what varies and encapsulate it.
- Since they favor delegation, they help you avoid the creation of large inheritance hierarchies, reducing complexity.

- Modularity
  - Ease the management (object technology).
- Cohesion
  - Measure of relatedness or consistency in the functionality of a software unit.
  - Cohesion is the degree to which the elements inside a module belong together.
  - Strong cohesion is good quality.
- Coupling
  - Degree with which methods of different modules are dependent on each other.
  - A loose coupling is good quality.
- Reusability
  - Libraries, frameworks.

- Guidelines that, when followed, can dramatically enhance the maintainability of software.
  - **Single Responsibility Principle (SRP)**. The SRP states that each class or similar unit of code should have one responsibility only and, therefore, only one reason to change.
  - **Open / Closed Principle (OCP)**. The OCP states that all classes and similar units of source code should be open for extension but closed for modification.
  - **Liskov Substitution Principle (LSP)**. The LSP specifies that functions that use references to base classes must be able to use objects of derived classes without knowing it.
  - **Interface Segregation Principle (ISP)**. The ISP specifies that clients should not be forced to depend upon interfaces that they do not use. Instead, those interfaces should be minimised.
  - **Dependency Inversion Principle (DIP)**. The DIP states that high level modules should not depend upon low level modules and that abstractions should not depend upon details.

Original Catalogue of Patterns

Creational Design Patterns	Structural Design Patterns	Behavioural Design Patterns
• Singleton Pattern	• Adapter Pattern	• Template Method Pattern
• Factory Pattern	• Composite Pattern	• Mediator Pattern
• Abstract Factory Pattern	• Proxy Pattern	• Chain of Responsibility Pattern
• Builder Pattern	• Flyweight Pattern	• Observer Pattern
• Prototype Pattern	• Façade Pattern	• Strategy Pattern
	• Bridge Pattern	• Command Pattern
	• Decorator Pattern	• State Pattern
		• Visitor Pattern
		• Interpreter Pattern
		• Iterator Pattern
		• Memento Pattern

- Creational design patterns are used to design the instantiation process of objects. The creational pattern uses the inheritance to vary the object creation.
- There are two recurring themes in these patterns. First, they all encapsulate knowledge about which concrete classes the system uses. Second, they hide how instances of these classes are created and put together.
  - Consequently, the creational patterns give you a lot of flexibility in what gets created, who creates it, how it gets created, and when.
- There can be some cases when two or more patterns looks fit as a solution to a problem. At other times, the two patterns complement each other, for example, Builder can be used with other pattern to implements which components to get built.

- **Factory Method** define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
- **Abstract Factory** provide an interface for creating families of related or dependent objects without specifying their concrete classes.
- **Builder** separate the construction of a complex object from its representation so that the same construction process can create different representations.
- **Prototype** specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
- **Singleton** ensure a class only has one instance, and provide a global point of access to it.

- Structural patterns are concerned with how classes and objects are composed to form larger structures. Structural class patterns use inheritance to compose interfaces or implementations.
  - As a simple example, consider how multiple inheritance mixes two or more classes into one. The result is a class that combines the properties of its parent classes. This pattern is particularly useful for making independently developed class libraries work together.
- Rather than composing interfaces or implementations, structural object patterns describe ways to compose objects to realize new functionality.
  - The added flexibility of object composition comes from the ability to change the composition at run-time, which is impossible with static class composition.

- **Adapter** convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- **Bridge** decouple an abstraction from its implementation so that the two can vary independently.
- **Composite** compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
- **Decorator** attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
- **Facade** provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
- **Flyweight** use sharing to support large numbers of fine-grained objects efficiently.
- **Proxy** provide a surrogate or placeholder for another object to control access to it.

- Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects. Behavioral patterns describe not just patterns of objects or classes but also the patterns of communication between them.
  - These patterns characterize complex control flow that's difficult to follow at run-time. They shift your focus away from flow of control to let you concentrate just on the way objects are interconnected.
- Behavioral object patterns use object composition rather than inheritance. Some describe how a group of peer objects cooperate to perform a task that no single object can carry out by itself.
  - An important issue here is how peer objects know about each other. Peers could maintain explicit references to each other, but that would increase their coupling. In the extreme, every object would know about every other. The Mediator pattern avoids this by introducing a mediator object between peers. The mediator provides the indirection needed for loose coupling.

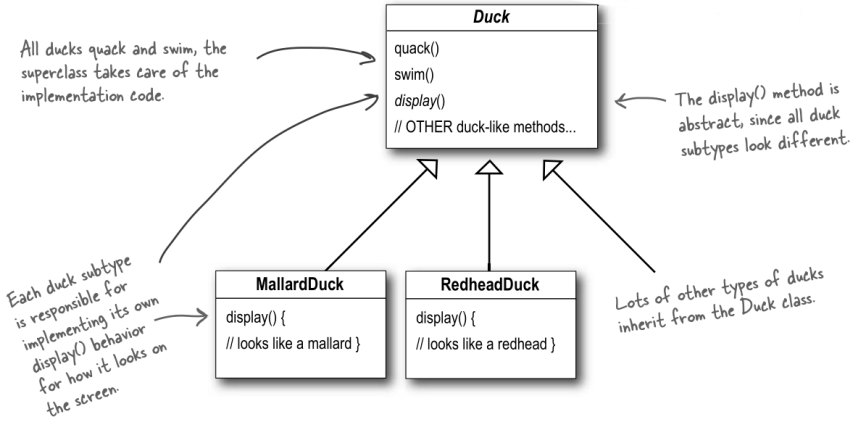
- **Chain of Responsibility** avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
- **Command** encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
- **Interpreter** given a language, define a represention for its grammar along with an interpreter that uses the representation to interpret sentences in the language.
- **Iterator** provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
- **Mediator** define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.
- **Memento** without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

- **Observer** define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- **State** allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
- **Strategy** define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
- **Template Method** define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
- **Visitor** represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

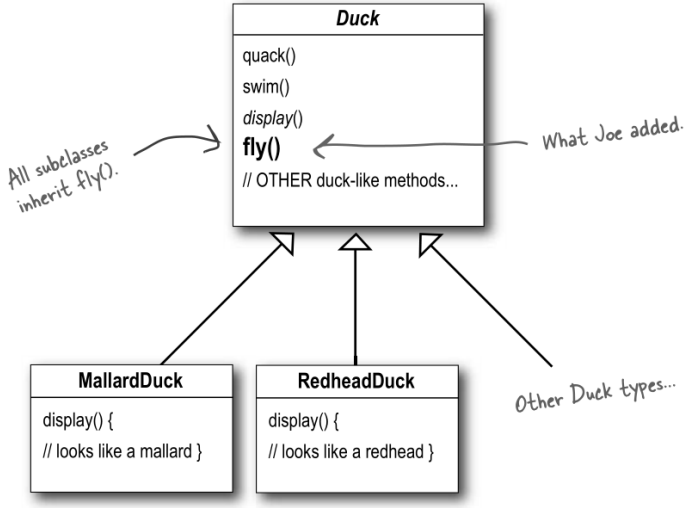
- There are a number of design patterns to choose from, to choose one, you must have good knowledge of each one of them.
  - There are many design patterns which look very similar to one another. They solve almost a similar type of design problem and also have similar implementation.
  - One must have a very deep understanding of them in order to implement the correct design pattern for the specific design problem.
- A design pattern can be used to solve more than one design problem, and one design problem can be solved by more than one design patterns.
  - There could be plenty of design problems and solutions for them, but, to choose the pattern which fits exactly is depends on your knowledge and understanding about the design patterns.
  - It also depends on the code you already have in place.

- You need to identify the kind of design problem you are facing.
  - A design problem can be categorized into creational, structural, or behavioral. Based to this category you can filter the patterns and selects the appropriate one.
- For example:
  - **There are too many instances of a class which represent only a single thing, the value in the properties of the objects are same, and they are only used as read-only**: you can select the **Singleton** pattern for this design problem which ensures only a single instance for the entire application. It also helps to decrease the memory size.
  - **Classes are too much dependent on each other. A Change in one class affects all other dependent classes**: you can use **Bridge**, **Mediator**, or **Command** to solve this design problem.
  - **There are two different incompatible interfaces in two different parts of the code, and your need is to convert one interface into another which is used by the client code to make the entire code work**: the **Adapter** pattern fits into this problem.

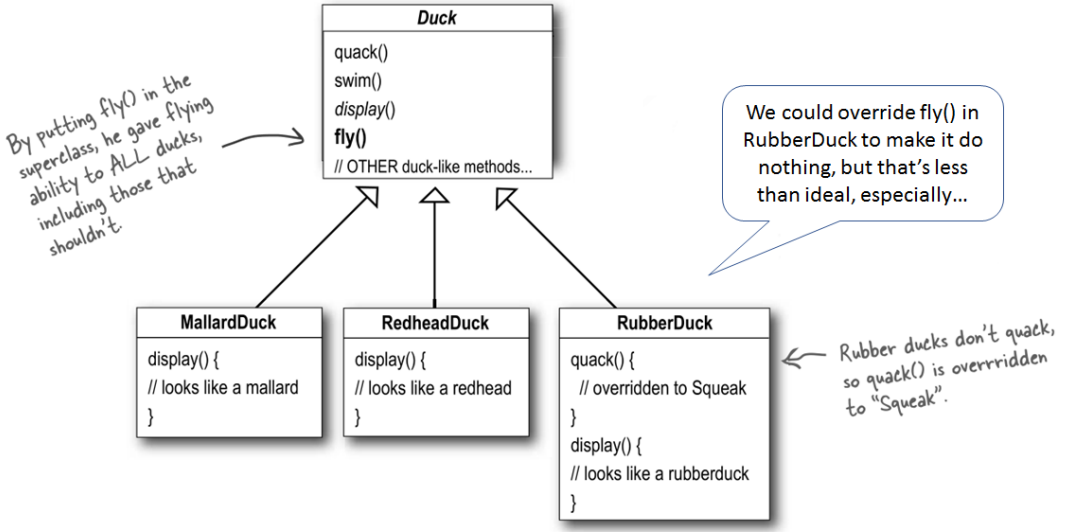
- DuckSimulator: a "Duck Pond Simulator" that can show a wide variety of duck species swimming and quacking in a pond.
- The initial designers of the system used standard OO techniques and created one Duck superclass from which all other duck types inherit.



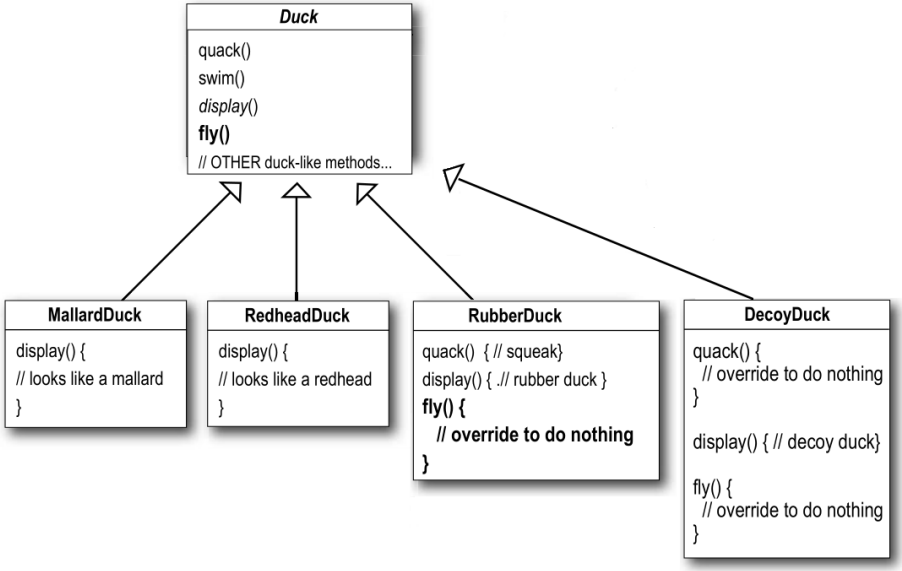
- But a request has arrived to allow ducks to also fly.

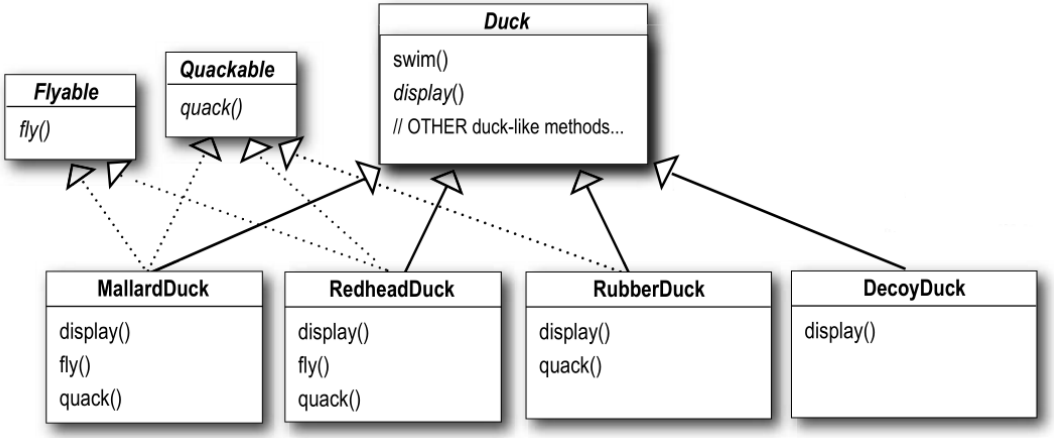


Whoops!


Duck
quack()
swim()
display()
fly()
// OTHER duck-like methods...
By putting fly() in the superclass, he gave flying ability to ALL ducks, including those that shouldn't.
We could override fly() in RubberDuck to make it do nothing, but that's less than ideal, especially...
MallardDuck
display() {
// looks like a mallard
}
RedheadDuck
display() {
// looks like a redhead
}
RubberDuck
quack() {
// overridden to Squeak
}
display() {
// looks like a rubberduck
}
Rubber ducks don't quack, so quack() is overrridden to "Squeak".

Double Whoops!





- Hear we define two interfaces and allow subclasses to implement the interfaces thay need. What are the trade-offs?

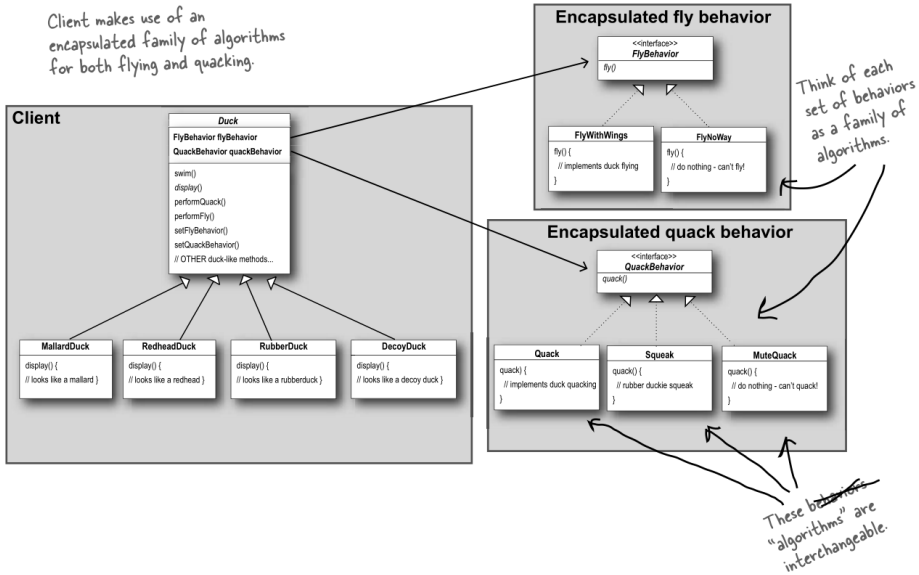
- With inheritance, we get:
  - Code reuse, only one fly() and quack() method vs. multiple (pro).
  - Common behavior in root class, not so common after all (con).
- With interfaces, we get:
  - Specificity: only those subclasses that need a fly() method get it (pro).
  - No code re-use: since interfaces only define signatures (con)
- Use of abstract base class over an interface? Could do it, but only in languages that support multiple inheritance.
  - In this approach, you implement Flyable and Quackable as abstract base classes and then have Duck subclasses use multiple inheritance.

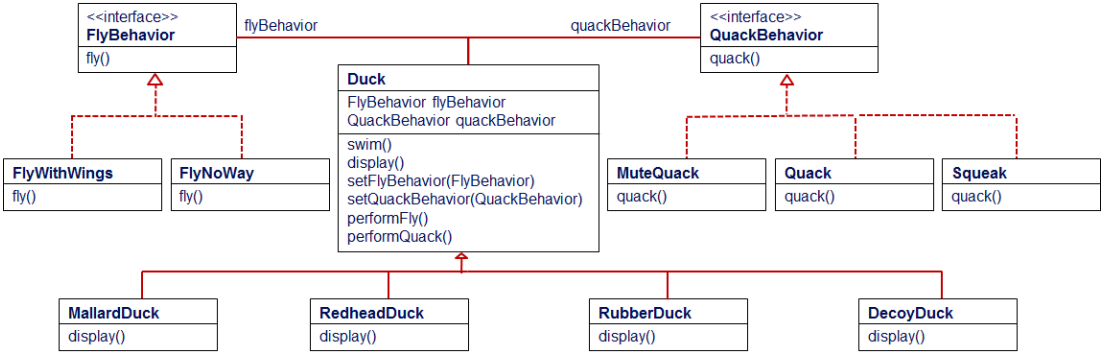
- **Encapsulate what varies**
  - For this particular problem, the "what varies" is the behaviors between Duck subclasses. We need to pull out behaviors that vary across the subclasses and put them in their own classes (i.e., encapsulate them.)
    - Duck will no longer have fly() and quack() methods directly.
    - Create two sets of classes, one that implements fly behaviors and one that implements quack behaviors.
  - Benefits: fewer unintended consequences from code changes (such as when we added fly() to Duck) and more flexible code.
- **Code to an interface**
  - We'll make use of the "code to an interface" principle and make sure that each member of the two sets implements a particular interface.
    - For QuackBehavior, we'll have Quack, Squeak, MuteQuack...
    - For FlyBehavior, we'll have FlyWithWings, FlyNoWay, FlyWhenThrown, ...
  - Additional Benefits: Other classes can gain access to these behaviors and we can add additional behaviors without impacting other classes.

"Code to an interface" does NOT imply Java Interface

- We are overloading the word "interface" when we say "code to an interface":
  - We can implement "code to an interface" by defining a Java interface and then have various classes implement that interface.
  - Or, we can "code to a supertype" and instead define an abstract base class which classes can access via inheritance.
- When we say "code to an interface" it implies that the object that is using the interface will have a variable whose type is the supertype (whether it is an **interface** or an **abstract base class**) and thus
  - can point at any implementation of that supertype,
  - and is shielded from their specific class names.
    - A Duck will point to a fly behavior with a variable of type FlyBehavior NOT FlyWithWings; the code will be more loosely coupled as a result.

- To take advantage of these new behaviors, we must modify Duck to delegate its flying and quacking behaviors to these other classes
  - rather than implementing this behavior internally.
- We'll add two attributes that store the desired behavior and we'll rename fly() and quack() to performFly() and performQuack().
  - This last step is meant to address the issue of it not making sense for a DecoyDuck to have methods like fly() and quack() directly as part of its interface.
    - Instead, it inherits these methods and plugs-in CantFly and Silence behaviors to make sure that it does the right things if those methods are invoked.
- This is an instance of the principle "**Favor delegation over inheritance**".


Client makes use of an encapsulated family of algorithms for both flying and quacking.
Client
Duck
FlyBehavior flyBehavior
QuackBehavior quackBehavior
swim()
display()
performQuack()
performFly()
setFlyBehavior()
setQuackBehavior()
// OTHER duck-like methods...
MallardDuck
display() { // looks like a mallard }
RedheadDuck
display() { // looks like a redhead }
RubberDuck
display() { // looks like a rubberduck }
DecoyDuck
display() { // looks like a decoy duck }
Encapsulated fly behavior
<<interface>>
FlyBehavior
fly()
FlyWithWings
fly() { // implements duck flying }
FlyNoWay
fly() { // do nothing - can't fly! }
Think of each set of behaviors as a family of algorithms.
Encapsulated quack behavior
<<interface>>
QuackBehavior
quack()
Quack
quack() { // implements duck quacking }
Squeak
quack() { // rubber duckie squeak }
MuteQuack
quack() { // do nothing - can't quack! }
These behaviors "algorithms" are interchangeable.



- FlyBehavior and QuackBehavior define a set of behaviors that provide behavior to Duck.
- Duck delegates to each set of behaviors and can switch among them dynamically, if needed.
- While each subclass now has a performFly() and performQuack() method, at least the user interface is uniform and those methods can point to null behaviors when required.

```java
public interface FlyBehavior {
  void fly();
}

public class FlyWithWings implements FlyBehavior {
  public void fly() {
    System.out.println("I'm flying!!");
  }
}

public class FlyRocketPowered implements FlyBehavior {
  public void fly() {
    System.out.println("I'm flying with a rocket");
  }
}

public class FlyNoWay implements FlyBehavior {
  public void fly() {
    System.out.println("I can't fly");
  }
}
```

```java
public interface QuackBehavior {
  void quack();
}

public class Squeak implements QuackBehavior {
  public void quack() {
    System.out.println("Squeak");
  }
}

public class Quack implements QuackBehavior {
  public void quack() {
    System.out.println("Quack");
  }
}

public class MuteQuack implements QuackBehavior {
  public void quack() {
    System.out.println("<< Silence >>");
  }
}
```

```java
public abstract class Duck {
  protected FlyBehavior flyBehavior;
  protected QuackBehavior quackBehavior;

  public Duck() { ... }

  public void setFlyBehavior(FlyBehavior flyBehavior) { ... }

  public void setQuackBehavior(QuackBehavior quackBehaviow) { ... }

  abstract void display();

  public void performFly() {
    flyBehavior.fly();
  }

  public void performQuack() {
    quackBehavior.quack();
  }

  public void swim() {
    System.out.println("All ducks float, even decoys!");
  }
}
```

```java
public class MallardDuck extends Duck {
  public MallardDuck() {
    quackBehavior = new Quack();
    flyBehavior = new FlyWithWings();
  }

  public void display() {
    System.out.println("I'm a real Mallard duck");
  }
}


public class RedHeadDuck extends Duck {
  public RedHeadDuck() {
    flyBehavior = new FlyWithWings();
    quackBehavior = new Quack();
  }

  public void display() {
    System.out.println("I'm a real Red Headed duck");
  }
}
```

```java
public class RubberDuck extends Duck {
  public RubberDuck() {
    flyBehavior = new FlyNoWay();
    quackBehavior = new Squeak();
  }

  public RubberDuck(FlyBehavior flyBehavior, QuackBehavior quackBehavior) { ... }

  public void display() {
    System.out.println("I'm a rubber duckie");
  }
}

public class DecoyDuck extends Duck {
  public DecoyDuck() {
    setFlyBehavior(new FlyNoWay());
    setQuackBehavior(new MuteQuack());
  }

  public void display() {
    System.out.println("I'm a duck Decoy");
  }
}
```

```java
public class DuckSimulator {
  public static void main(String[] args) {
    List<Duck> ducks = new LinkedList<>();
    ducks.add(new MallardDuck());
    ducks.add(new DecoyDuck());
    ducks.add(new RedheadDuck());
    Duck myDuck = new RubberDuck();  // Note: variable is type Duck, not the
                                     // specific subtypes (Code to  interface).
    ducks.add(myDuck);

    processDucks(ducks);

    // Change behaviors of ducks dynamically.
    // Here we see the power of delegation, we can change behaviors at run-time
    myDuck.setFlyBehavior(new FlyRocketPowered());
    myDuck.setQuackBehavior(new Speak());

    processDucks(ducks);
  }
```

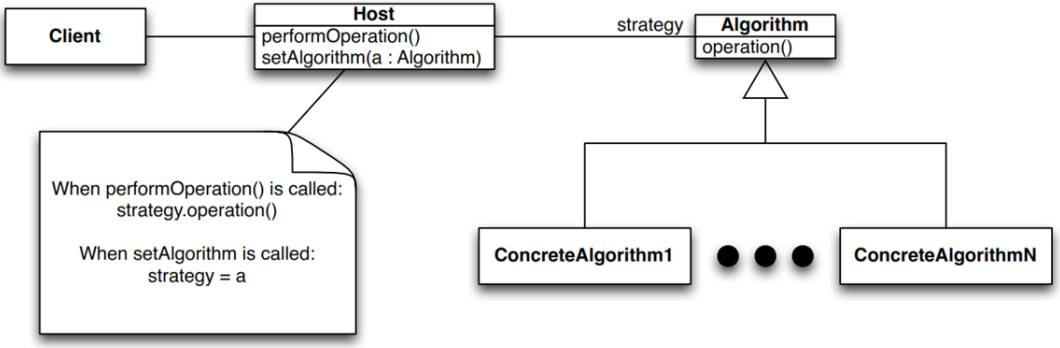
```java
/*
 * Because of abstraction and polymorphism, processDucks()
 *   consists of nice, clean, robust, and extensible code!
 */
public static void processDucks(List<Duck> ducks) {
  for (Duck duck : ducks) {
    System.out.println("--------------------");
    System.out.println("Name: " + duck.getClass().getName());
    duck.display();
    duck.performQuack();
    duck.performFly();
    duck.swim();
  }

  System.out.println("Done processing ducks!");
}
}
```

- The solution that we applied to this design problem is known as the **Strategy Design Pattern**. It features the following design concepts/principles:
  - **Encapsulate what varies**
  - **Code to an Interface, not implementations**
  - **Favor delegation over inheritance**

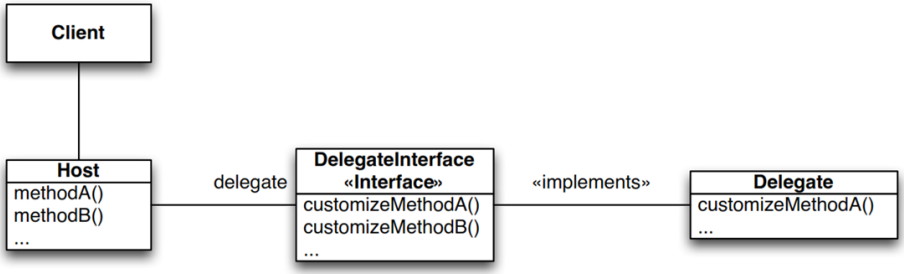
Strategy Pattern

The **Strategy Pattern** defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

- Creating systems using delegation gives you a lot more flexibility. Not only does it let you encapsulate a family of algorithms into their own set of classes, but it also lets you change behavior at runtime as long as the object you're composing with implements the correct behavior interface.



- Algorithm is pulled out of Host. Client only makes use of the public interface of Algorithm and is not tied to concrete subclasses.
- Client can change its behavior by switching among the various concrete algorithms.



- Purpose of Delegate:
  - Allow an object's behavior to be customized without forcing a developer to create a subclass that overrides default behavior.
- Client invokes method on Host; Host checks to see if Delegate handles this method; if so, it routes the call to the Delegate; if not, it provides default behavior for the method.
  - Here, if Client invokes methodA() on Host, the Delegate's customizeMethodA() will be invoked at some point to help customize Host's behavior for methodA().

- Let's derive this pattern by starting with a class that has no restrictions on who can create it.

```java
public class Ball {
  private String color;

  public Ball(String color) {
    this.color = color;
  }

  public void bounce() {
    System.out.println("Boing!");
  }
}

Ball ball1 = new Ball("Red");
Ball ball2 = new Ball("Blue");

// As long as a client object knows about the name of class Ball, it can create
//   instances of Ball. This is because the constructor is public.
// We can stop unauthorized creation of Ball instances by making the constructor private.
```

```java
public class Ball {
  private String color;

  private Ball(String color) {
  this.color = color;
}

  public void bounce() {
    System.out.println("Boing!");
  }
}


public class TestBall {
  public static void main(String[] args) {
    Ball ball = new Ball("Red");  // ERROR
    // Now Instantiation of Ball is impossible by any method outside of Ball.
  }
}
```

- Now that the constructor is private, no class can gain access to instances of Ball.
  - But our requirements were that there would be at least one way to get access to an instance of Ball.
- We need a method to return an instance of Ball.
  - But since there is no way to get access to an instance of Ball, the method CANNOT be an instance method.
    - This means it needs to be a class method, aka a static method.

```java
public class Ball {
  private String color;

  private Ball(String color) {
    this.color = color;
  }

  public void bounce() {
    System.out.println("Boing!");
  }

  public static Ball getInstance(String color) {
    return new Ball(color);
  }
}


public class TestBall {
  public static void main(String[] args) {
    Ball ball = new Ball("Red");  // ERROR
    // Now Instantiation of Ball is impossible by any method outside of Ball.
    Ball ball = Ball.getInstance("Blue");  // OK
  }
}
```

```java
public class Ball {
  private static Ball ball;
  private String color;

  private Ball(String color) {
    this.color = color;
  }

  public void bounce() {
    System.out.println("Boing!");
  }

  public static Ball getInstance(String color) {
    return ball;
  }
}
```

- To ensure only one instance is ever created,
  - need a static variable to store that instance.
    - No instance variables are available in static methods.
- Now the getInstance() method returns null each time it is called.
  - Need to check the static variable to see if it is null.
    - If so, create an instance.
  - Otherwise return the single instance.

```java
public class Ball {
  private static Ball ball;
  private String color;

  private Ball(String color) {
    this.color = color;
  }

  public void bounce() {
    System.out.println("Boing!");
  }

  public static Ball getInstance(String color) {
    // Lazy instantiation
    if (ball == null) {
      ball = new Ball(color);
    }
    return ball;
  }
}
```

- The code shows the Singleton pattern:
  - private constructor.
  - private static variable to store the single instance.
  - public static method to gain access to that instance.
    - This method creates object if needed; returns it.
- But this code ignores the fact that a parameter is being passed in.
  - so if a “Red” ball is created all subsequent requests for a "Green" ball are ignored.

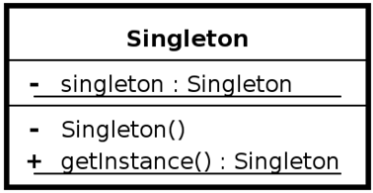
- The solution to the final problem is to change the private static instance variable to a Map.
- Then check if the map contains an instance for a given value of the parameter.
  - This ensures that only one ball of a given color is ever created.
  - This is a very acceptable variation of the Singleton pattern.

```
...
private static Map<String, Ball> ballRecord = new HashMap...
...

public static Ball getInstance(String color) {
  if (!ballRecord.containsKey(color)) {
    ballRecord.put(color, new Ball(color));
  }
  return ballRecord.get(color);
}
```

Singleton Pattern

Singleton Pattern used to ensure that only one instance of a particular class ever gets created and that there is just one (global) way to gain access to that instance.

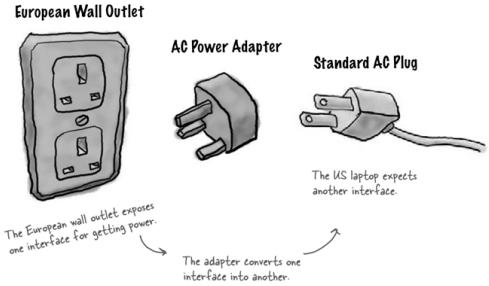
- What's really going on here?
  - We're taking a class and letting it manage a single instance of itself.
  - We're also preventing any other class from creating a new instance on its own.
  - To get an instance, you've got to go through the class itself.
- We're also providing a global access point to the instance:
  - Whenever you need an instance, just query the class and it will hand you back the single instance.
  - We can implement this so that the Singleton is created in a lazy manner, which is especially important for resource-intensive objects.



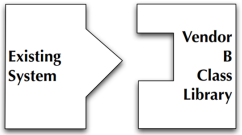
- Singleton involves only a single class (not typically called Singleton). That class is a full-fledged class with other attributes and methods (not shown).
- The class has a static variable that points at a single instance of the class.
- The class has a private constructor (to prevent other code from instantiating the class) and a static method that provides access to the single instance.

- Centralized manager of resources:
  - Window manager
  - File system manager
  - ...
- Logger classes
- Factories
  - Especially those that issue IDs.
  - **Singleton** is often combined with **Factory Method** and **Abstract Factory** patterns.

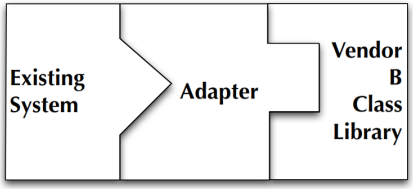
- Adapter pattern provides steps for converting an incompatible interface with an existing system into a different interface that is compatible.
- Real world example: AC power adapters.
  - Electronic products made for the USA cannot be used directly with outlets found in most other parts of the world.
  - To use these products outside the US, you need an AC power adapter.



- **Pre-condition**: You are maintaining an existing system that makes use of a third-party library from vendor A.
- **Stimulus**: Vendor A goes belly up and corporate policy does not allow you to make use of an unsupported class library.
- **Response**: Vendor B provides a similar library but its interface is completely different from the interface provided by vendor A.
- **Assumptions**: You don't want to change your code, and you can't change vendor B's code.
- **Solution**: Write new code that adapts vendor B's interface to the interface expected by your original code.



...plug it in



A slightly different duck model.

```java
public interface Duck {
  void quack();
  void fly();
}
```

```java
public class MallardDuck implements Duck {
  public void quack() {
    System.out.println("Quack");
  }

  public void fly() {
    System.out.println("I'm flying");
  }
}
```

An interloper wants to invade the simulator.

```java
public interface Turkey {
  public void gobble();
  public void fly();
}
```

```java
public class WildTurkey implements Turkey {
  public void gobble() {
    System.out.println("Gobble Gobble");
  }

  public void fly() {
    System.out.println("I'm flying a short distance");
  }
}
```

Example: A Turkey Amongst Ducks!

- But the duck simulator doesn't know how to handle turkeys, only ducks!
- Solution

If it walks like a duck and quacks like a duck, then it must be a duck!

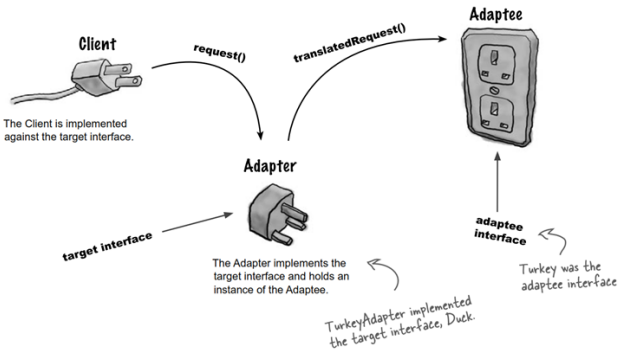
Or…

It might be a **turkey wrapped with a duck adapter**!

```java
public class TurkeyAdapter implements Duck {
  private Turkey turkey;

  public TurkeyAdapter(Turkey turkey) {
    this.turkey = turkey;
  }

  public void quack() {
    turkey.gobble();
  }

  public void fly() {
    for (int i = 0; i < 5; i++) {
      turkey.fly();
    }
  }
}
```

1. Adapter implements target interface (Duck).
2. Adaptee (turkey) is passed via constructor and stored internally.
3. Calls by client code are delegated to the appropriate methods in the adaptee.
4. Adapter is full-fledged class, could contains additional variables and methods to get its job done; can be used polymorphically as a Duck.

```
public class DuckSimulator {
  public static void main(String[] args) {
    MallardDuck mallardDuck = new MallardDuck();

    WildTurkey wildTurkey = new WildTurkey();
    // Swap the turkey in a TurkeyAdapter, which makes it look like a Duck
    Duck turkeyAdapter = new TurkeyAdapter(wildTurkey);

    List<Duck> ducks = new LinkedList<>();
    ducks.add(mallardDuck);
    ducks.add(turkeyAdapter);

    for (Duck duck : ducks) {
      duck.quack();
      duck.fly();
    }
  }
}
```



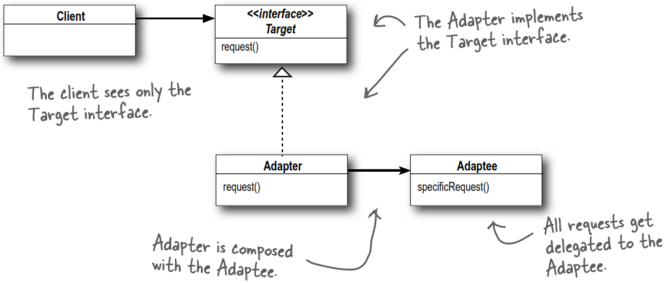
Here's how the Client uses the Adapter

1. The client makes a request to the adapter by calling a method on it using the target interface.
2. The adapter translates the request into one or more calls on the adaptee using the adaptee interface.
3. The client receives the results of the call and never knows there is an adapter doing the translation.

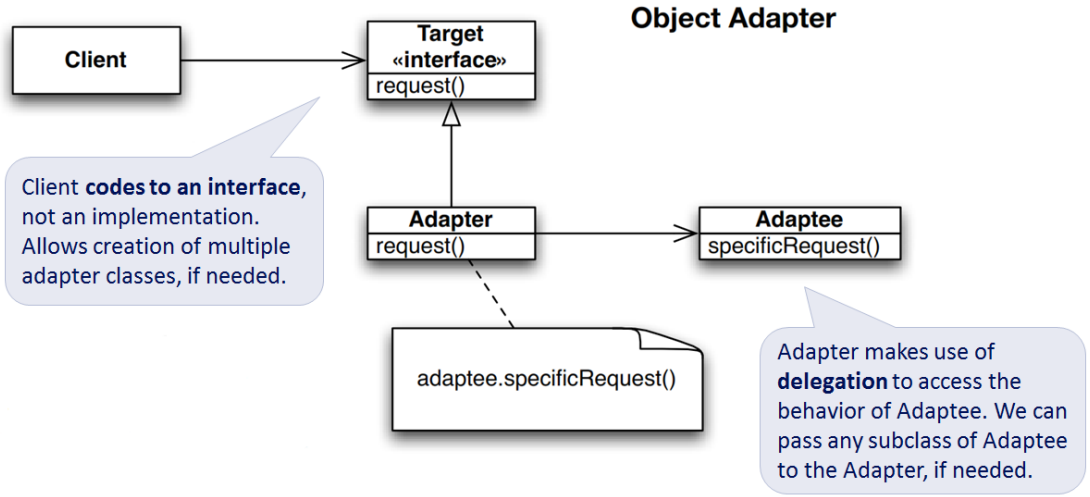
Note that the Client and Adaptee are decoupled – neither knows about the other.

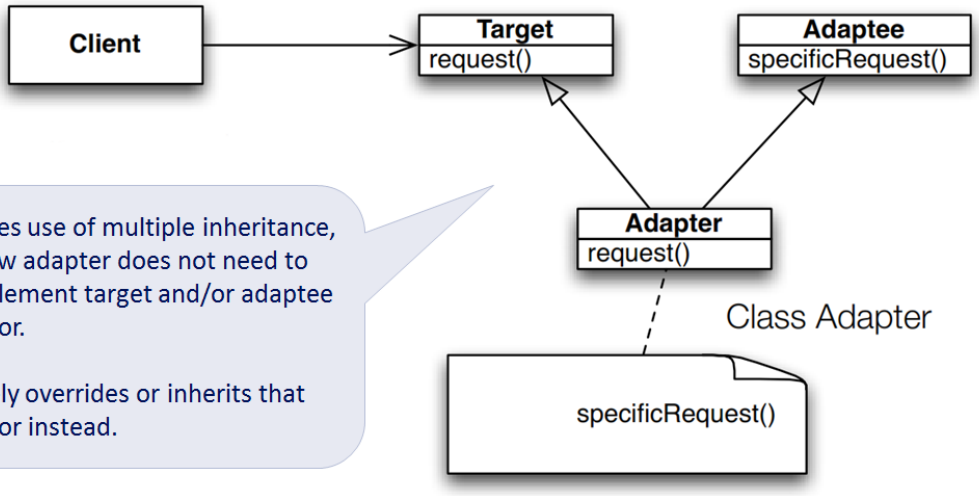
Adapter Pattern

The Adapter pattern converts the interface of a class into another interface that clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.



- **Adapter Pattern** allows us to use a client with an incompatible interface by creating an Adapter that does the conversion.
- This acts to decouple the client from the implemented interface.
- If we expect the interface to change over time, the adapter encapsulates that change so that the client doesn't have to be modifi ed each time it needs to operate against a different interface.


Object Adapter
Client
Target
«interface»
request()
Client codes to an interface, not an implementation. Allows creation of multiple adapter classes, if needed.
Adapter
request()
Adaptee
specificRequest()
adaptee.specificRequest()
Adapter makes use of delegation to access the behavior of Adaptee. We can pass any subclass of Adaptee to the Adapter, if needed.



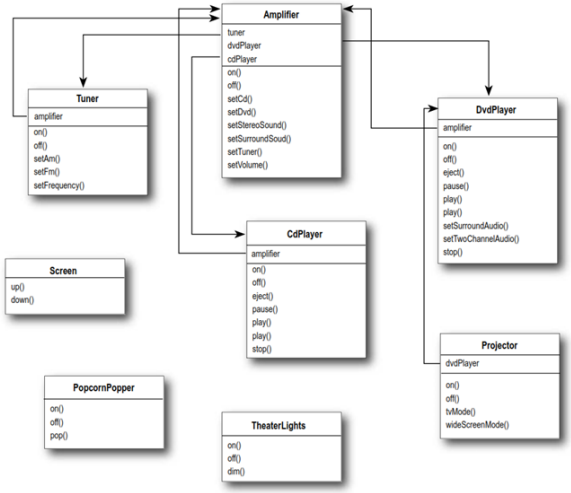
Requires use of multiple inheritance, but now adapter does not need to re-implement target and/or adaptee behavior.

It simply overrides or inherits that behavior instead.

- Before Java's new collection classes, iteration over a collection occurred via java.util.Enumeration
  - hasMoreElements() : boolean
  - nextElement() : Object
- With the collection classes, iteration was moved to a new interface: java.util.Iterator
  - hasNext(): boolean
  - next(): Object
  - remove(): void
- There's a lot of code out there that makes use of the Enumeration interface
  - New code can still make use of that code by creating an adapter that converts from the Enumeration interface to the Iteration interface.

- Imagine a library of classes with a complex interface and/or complex interrelationships.
- Home Theater System
  - Amplifier, DvdPlayer, Projector, CdPlayer, Tuner, Screen, PopcornPopper, and TheaterLights.
  - Each with its own interface and interclass dependencies.

That's a lot of classes, a lot of interactions, and a big set of interfaces to learn and use

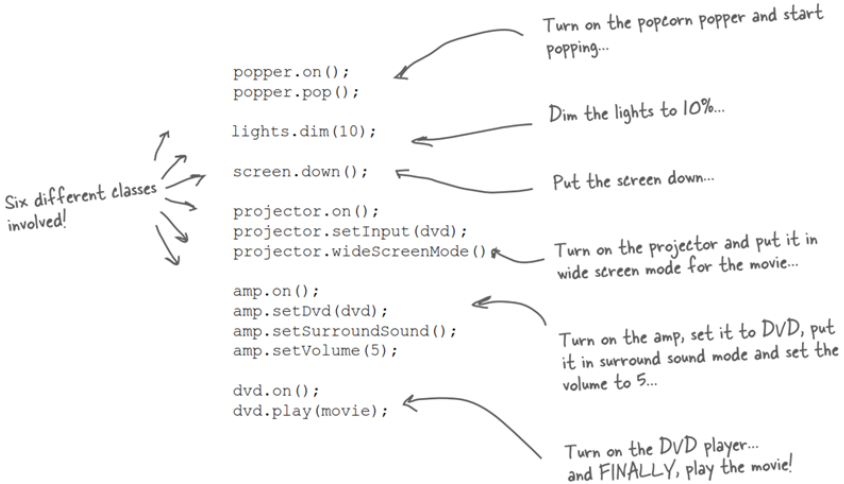


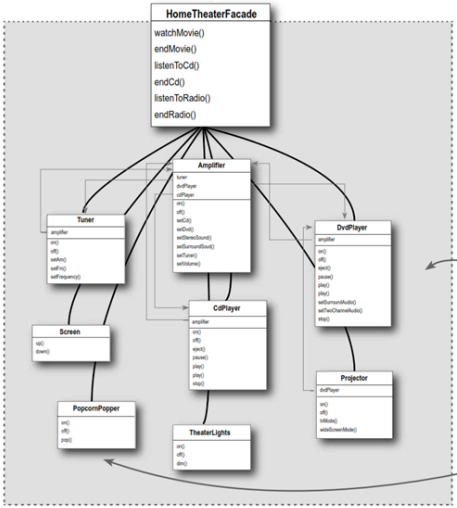
Watching a movie (the hard way)

There's just one thing – to watch the movie, you need to perform a few tasks:

1. Turn on the popcorn popper
2. Start the popper popping
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 wide-screen 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 (5)
12. Turn the DVD Player on
13. Start the DVD Player playing

Let's check out those same tasks in terms of the classes and the method calls needed to perform them:






```
public class HomeTheaterFacade {
    Amplifier amp;
    Tuner tuner;
    DvdPlayer dvd;
    CdPlayer cd;
    Projector projector;
    TheaterLights lights;
    Screen screen;
    PopcornPopper popper;

    public HomeTheaterFacade(Amplifier amp,
                             Tuner tuner,
                             DvdPlayer dvd,
                             CdPlayer cd,
                             Projector projector,
                             Screen screen,
                             TheaterLights lights,
                             PopcornPopper popper) {

        this.amp = amp;
        this.tuner = tuner;
        this.dvd = dvd;
        this.cd = cd;
        this.projector = projector;
        this.screen = screen;
        this.lights = lights;
        this.popper = popper;
    }

    // other methods here

}
```

Here's the composition; these are all the components of the subsystem we are going to use.

The facade is passed a reference to each component of the subsystem in its constructor. The facade then assigns each to the corresponding instance variable.

We're just about to fill these in...

Time to watch a movie (the easy way)

It's SHOWTIME!

```
public class HomeTheaterTestDrive {
    public static void main(String[] args) {
        // instantiate components here

        HomeTheaterFacade homeTheater =
                new HomeTheaterFacade(amp, tuner, dvd, cd,
                        projector, screen, lights, popper);

        homeTheater.watchMovie("Raiders of the Lost Ark");
        homeTheater.endMovie();
    }
}
```

Here we're creating the components right in the test drive. Normally the client is given a facade, it doesn't have to construct one itself.

First you instantiate the Facade with all the components in the subsystem.

Use the simplified interface to first start the movie up, and then shut it down.

```java
public class PopcornPopper {
  private String description;

  public PopcornPopper(String description) {
    this.description = description;
  }

  public void on() {
    System.out.println(description + " on");
  }

  public void off() {
    System.out.println(description + " off");
  }

  public void pop() {
    System.out.println(description + " popping popcorn!");
  }

  public String toString() {
    return description;
  }
}
```

```java
public class Screen {
  private String description;

  public Screen(String description) {
    this.description = description;
  }

  public void up() {
    System.out.println(description + " going up");
  }

  public void down() {
    System.out.println(description + " going down");
  }

  public String toString() {
    return description;
  }
}
```

```java
public class Projector {
  private String description;
  private DvdPlayer dvdPlayer;

  public Projector(String description, DvdPlayer dvdPlayer) {
    this.description = description;
    this.dvdPlayer = dvdPlayer;
  }

  public void on() {
    System.out.println(description + " on");
  }

  public void off() {
    System.out.println(description + " off");
  }

  public void wideScreenMode() {
    System.out.println(description + " in widescreen mode (16x9 aspect ratio)");
  }

  ...
}
```

```java
public class TheaterLights {
  String description;

  public TheaterLights(String description) {
    this.description = description;
  }

  public void on() {
    System.out.println(description + " on");
  }

  public void off() {
    System.out.println(description + " off");
  }

  public void dim(int level) {
    System.out.println(description + " dimming to " + level + "%");
  }

  public String toString() {
    return description;
  }
}
```

```
public class HomeTheaterFacade {
  private Amplifier amp;
  private Tuner tuner;
  private DvdPlayer dvd;
  private CdPlayer cd;
  private Projector projector;
  private TheaterLights lights;
  private Screen screen;
  private PopcornPopper popper;

  public HomeTheaterFacade(Amplifier amp,
                           Tuner tuner,
                           DvdPlayer dvd,
                           CdPlayer cd,
                           Projector projector,
                           Screen screen,
                           TheaterLights lights,
                           PopcornPopper popper) {
    this.amp = amp;
    this.tuner = tuner;
    this.dvd = dvd;
    ...
  }
```

```java
public void watchMovie(String movie) {
    System.out.println("Get ready to
        ↪ watch a movie...");
    popper.on();
    popper.pop();
    lights.dim(10);
    screen.down();
    projector.on();
    projector.wideScreenMode();
    amp.on();
    amp.setDvd(dvd);
    amp.setSurroundSound();
    amp.setVolume(5);
    dvd.on();
    dvd.play(movie);
}
```

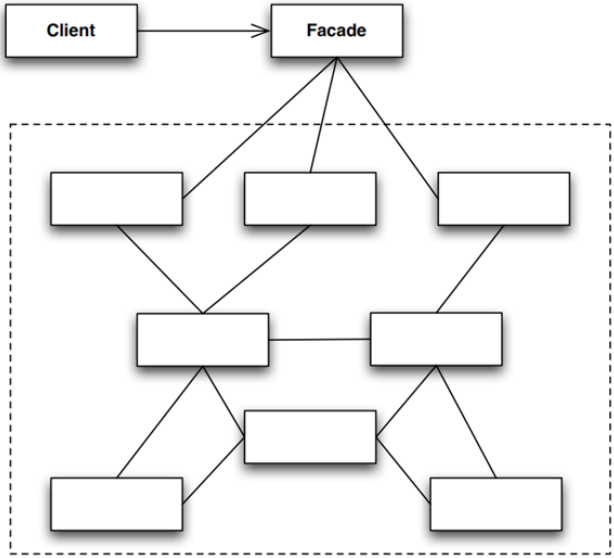
```java
public void endMovie() {
    System.out.println("Shutting
        ↪ movie theater down...");
    popper.off();
    lights.on();
    screen.up();
    projector.off();
    amp.off();
    dvd.stop();
    dvd.eject();
    dvd.off();
}
```

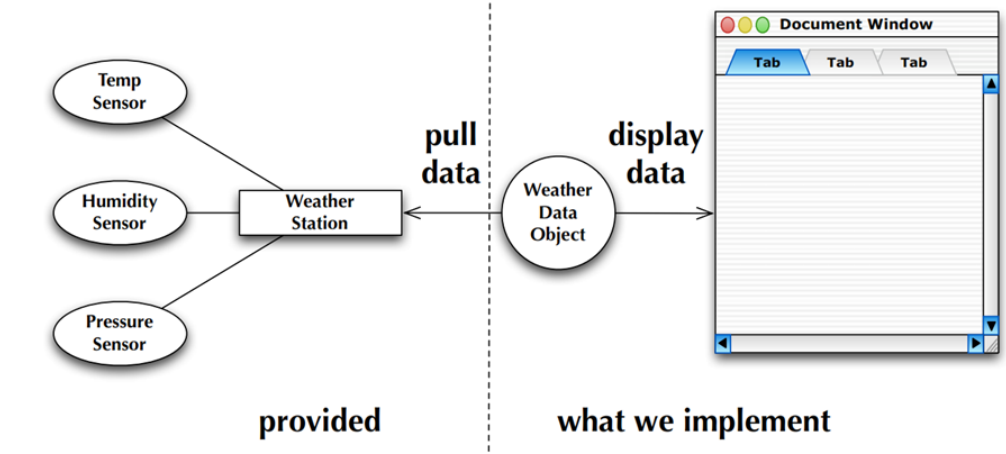
```java
public class HomeTheaterTestDrive {
  public static void main(String[] args) {
    // Initialize subsystems
    Amplifier amp = new Amplifier("Top-O-Line Amplifier");
    Tuner tuner = new Tuner("Top-O-Line AM/FM Tuner", amp);
    DvdPlayer dvd = new DvdPlayer("Top-O-Line DVD Player", amp);
    CdPlayer cd = new CdPlayer("Top-O-Line CD Player", amp);
    Projector projector = new Projector("Top-O-Line Projector", dvd);
    TheaterLights lights = new TheaterLights("Theater Ceiling Lights");
    Screen screen = new Screen("Theater Screen");
    PopcornPopper popper = new PopcornPopper("Popcorn Popper");

    HomeTheaterFacade homeTheater = new HomeTheaterFacade(amp, tuner, dvd, cd,
    ↪ projector, screen, lights, popper);

    homeTheater.watchMovie("Raiders of the Lost Ark");
    homeTheater.endMovie();
  }
}
```

Facade Pattern

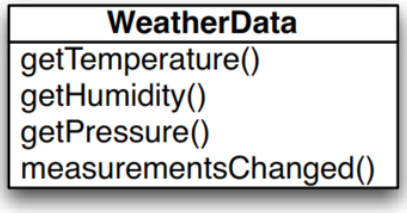
Facade Pattern provide a unified interface to a set of interfaces in a subsystem. **Façade** defines a higher-level interface that makes the subsystem easier to use.

- There can be significant benefit in wrapping a complex subsystem with a simplified interface.
  - If you don't need the advanced functionality or fine-grained control of the former, the latter makes life easy.
- **Façade** works best when you are accessing a subset of the system's functionality.
  - You can add new features by adding them to the **Façade** (not the subsystem); you still get a simpler interface.
- Client code is simplified and dependencies are reduced.
  - A **Façade** not only simplifies an interface, it decouples a client from a subsystem of components
- Indeed, **Façade** lets us encapsulate subsystems, hiding them from the rest of the system.


Client
Facade



- We need to pull information from the station and then generate "Current Conditions, Weather Statistics, and a Weather Forecast".



- We receive a partial implementation of the WeatherData class from our client.
  - They provide three getter methods for the sensor values and an empty measurementsChanged() method that is guaranteed to be called whenever a sensor provides a new value.
  - We need to pass these values to our three displays… so that's simple!

```
pulic class WeatherData {
  // instance variable declarations

  public void measurementsChanged() {
    float temp = getTemperature();
    float humidity = getHumidity();
    float pressure = getPressure();

    currentConditionsDisplay.update(temp,
                                    humidity,
                                    pressure);
    statisticsDisplay.update(temp,
                             humidity,
                             pressure);
    forecastDisplay.update(temp,
                           humidity,
                           pressure);
  }

  // Other WeatherData methods here
}
```

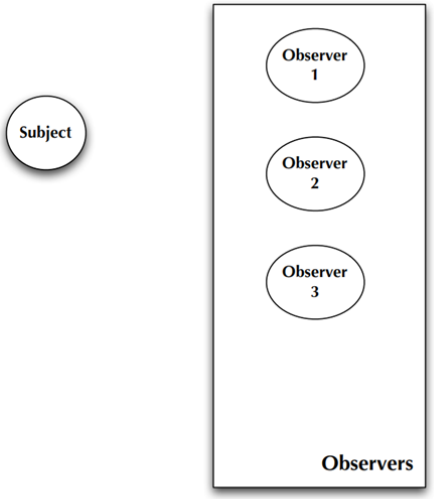
Problems?

- The number and type of displays may vary.
  - These three displays are hard coded with no easy way to update them.
- Coding to implementations, not an interface!
  - Each implementation has adopted the same interface, so this will make translation easy!

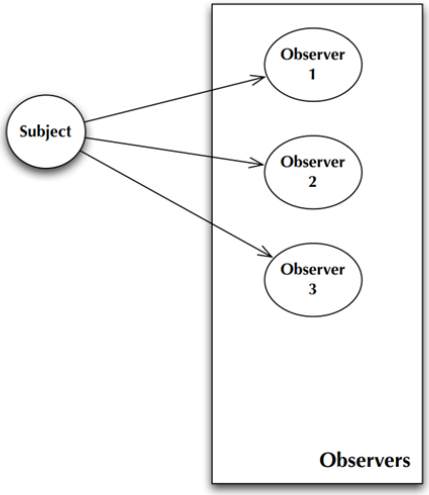
Solution:

- This situation can benefit from use of the observer pattern.

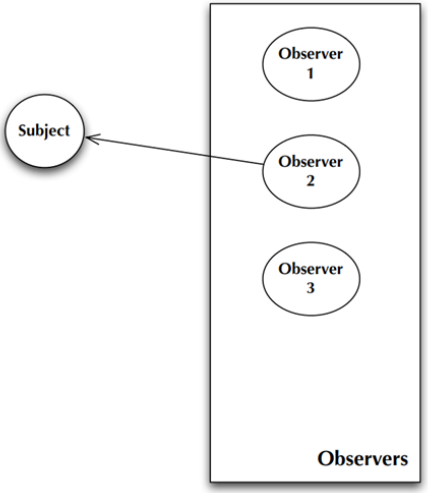
- This pattern is similar to subscribing to a hard copy newspaper.
  - A newspaper comes into existence and starts publishing editions.
  - You become interested in the newspaper and subscribe to it.
  - Any time an edition becomes available, you are notified (by the fact that it is delivered to you).
  - When you don't want the paper anymore, you unsubscribe.
  - The newspaper's current set of subscribers can change at any time.
- Observer is just like this but we call the publisher the "subject" and we refer to subscribers as "observers".



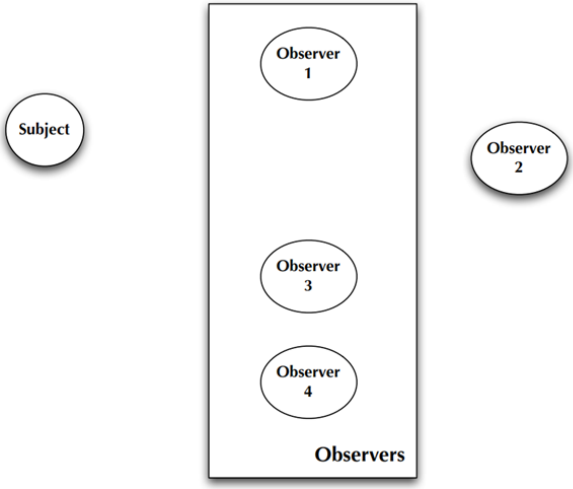
Subject maintains a list of observers



If the Subject changes, it notifies its observers



If needed, an observer may query its subject for more information

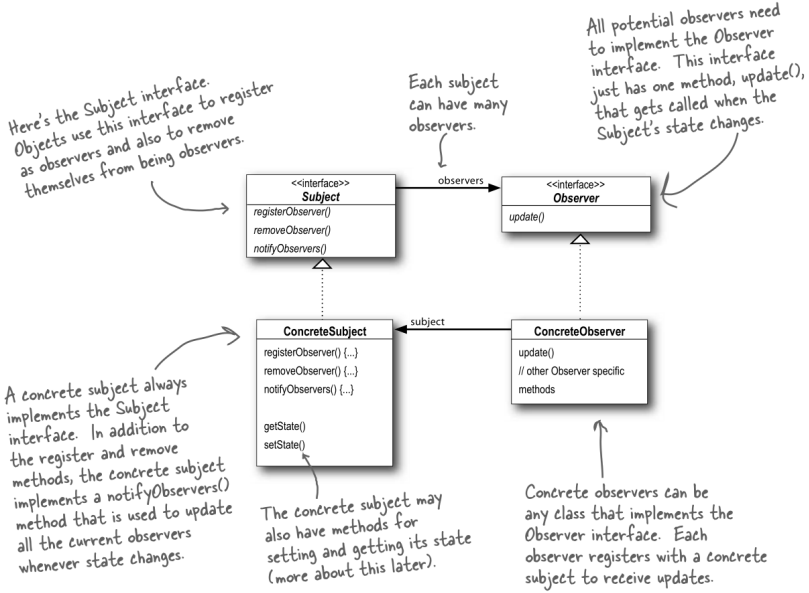


At any point, an observer may join or leave the set of observers

Observer Pattern

The Observer Pattern defines a one-to-many dependency between a set of objects, such that when one object (the subject) changes all of its dependents (observers) are notified and updated automatically.

- Don't miss out when something interesting (in your system) happens!.
  - The **Observer Pattern** allows objects to keep other objects informed about events occurring within a software system (or across multiple systems).
  - It's dynamic in that an object can choose to receive or not receive notifications at runtime.
  - Observer happens to be one of the most heavily used patterns in the Java Development Kit.
    - And indeed is present in many frameworks.


Here's the Subject interface. Objects use this interface to register as observers and also to remove themselves from being observers.
Each subject can have many observers.
All potential observers need to implement the Observer interface. This interface just has one method, update(), that gets called when the Subject's state changes.
<<interface>>
Subject
registerObserver()
removeObserver()
notifyObservers()
observers
<<interface>>
Observer
update()
ConcreteSubject
registerObserver() {...}
removeObserver() {...}
notifyObservers() {...}
getState()
setState()
subject
ConcreteObserver
update()
// other Observer specific methods
A concrete subject always implements the Subject interface. In addition to the register and remove methods, the concrete subject implements a notifyObservers() method that is used to update all the current observers whenever state changes.
The concrete subject may also have methods for setting and getting its state (more about this later).
Concrete observers can be any class that implements the Observer interface. Each observer registers with a concrete subject to receive updates.

- Observer affords a loosely coupled interaction between subject and observer.
  - This means they can interact with very little knowledge about each other.
- The subject only knows that observers implement the Observer interface.
  - We can add/remove observers of any type at any time.
  - We never have to modify subject to add a new type of observer.
- We can reuse subjects and observers in other contexts.
  - The interfaces plug-and-play anywhere observer is used.
- Observers may have to know about the ConcreteSubject class if it provides many different state-related methods.
  - Otherwise, data can be passed to observers via the update() method.

```java
public interface Subject {
  public void registerObserver(Observer observer);
  public void removeObserver(Observer observer);
  public void notifyObservers();
}


public interface Observer {
  public void update(float temp, float humidity, float pressure);
}


public interface DisplayElement {
  public void display();
}
```

```java
public class WeatherData implements Subject {
  private List<Observer> observers;
  private float temperature;
  private float humidity;
  private float pressure;

  public WeatherData() {
    observers = new ArrayList<>();
  }

  public void registerObserver(Observer observer) {
    observers.add(observer);
  }

  public void removeObserver(Observer observer) {
    observers.remove(observer);
  }

  public void notifyObservers() {
    for (Observer observer: observers) {
      observer.update(temperature, humidity, pressure);
    }
  }
```

```
  public void measurementsChanged() {
    notifyObservers();
  }

  public void setMeasurements(float temperature, float humidity, float pressure) {
    this.temperature = temperature;
    this.humidity = humidity;
    this.pressure = pressure;
    measurementsChanged();
  }

  public float getTemperature() {
    return temperature;
  }

  public float getHumidity() {
    return humidity;
  }

  public float getPressure() {
    return pressure;
  }
}
```

```java
public class ForecastDisplay implements Observer, DisplayElement {
  private float currentPressure;
  private float lastPressure;
  private WeatherData weatherData;

  public ForecastDisplay(WeatherData weatherData) {
    this.weatherData = weatherData;
    weatherData.registerObserver(this);
  }

  public void update(float temp, float humidity, float pressure) {
    lastPressure = currentPressure;
    currentPressure = pressure;
    display();
  }

  public void display() {
    if (Float.compare(currentPressure, lastPressure) > 0) {
      System.out.println("Improving weather on the way!");
    } else if (Float.compare(currentPressure, lastPressure) < 0) {
      System.out.println("Watch out for cooler, rainy weather");
    }
  }
}
```

```java
public class StatisticsDisplay implements Observer, DisplayElement {
  private float maxTemp;
  private float minTemp;
  private float tempSum;
  private int numReadings;
  private WeatherData weatherData;

  public StatisticsDisplay(WeatherData weatherData) { ... }

  public void update(float temp, float humidity, float pressure) {
    tempSum += temp;
    numReadings++;
    if (temp > maxTemp) maxTemp = temp;
    if (temp < minTemp) minTemp = temp;
    display();
  }

  public void display() {
    System.out.println("Avg/Max/Min temperature = " + (tempSum / numReadings)
      + "/" + maxTemp + "/" + minTemp);
  }
}
```

```java
public class WeatherStation {
  public static void main(String[] args) {
    WeatherData weatherData = new WeatherData();

    CurrentConditionsDisplay currentDisplay = new CurrentConditionsDisplay(weatherData);
    StatisticsDisplay statisticsDisplay = new StatisticsDisplay(weatherData);
    ForecastDisplay forecastDisplay = new ForecastDisplay(weatherData);

    System.out.println("----------------------------------------");
    weatherData.setMeasurements(80, 65, 30.4f);

    System.out.println("----------------------------------------");
    weatherData.setMeasurements(82, 70, 29.2f);

    System.out.println("----------------------------------------");
    weatherData.setMeasurements(78, 90, 29.2f);
    System.out.println("----------------------------------------");
  }
}
```

- Consider an example in which we have recipes for making tea and coffee at a coffee shop.

Coffee

1. Boil water
2. Brew coffee in boiling water
3. Pour coffee in cup
4. Add sugar and milk

Tea

1. Boil water
2. Steep tea in boiling water
3. Pour tea in cup
4. Add lemon

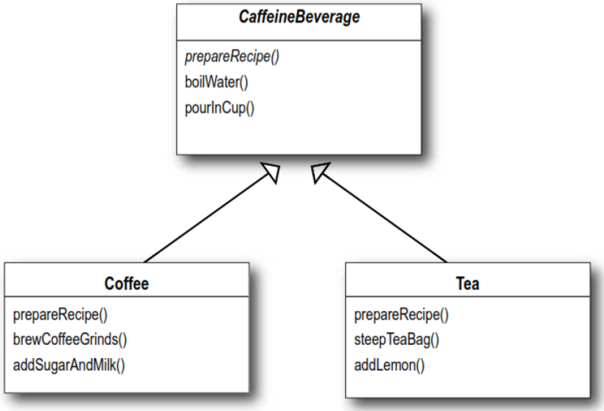
```java
public class Coffee {
  public void prepareRecipe() {
    boilWater();
    brewCoffeeGrinds();
    pourInCup();
    addSugarAndMilk();
  }

  public void boilWater() {
    System.out.println("Boiling water");
  }

  public void brewCoffeeGrinds() {
    System.out.println("Dripping Coffee through filter");
  }

  public void pourInCup() {
    System.out.println("Pouring into cup");
  }

  public void addSugarAndMilk() {
    System.out.println("Adding Sugar and Milk");
  }
}
```

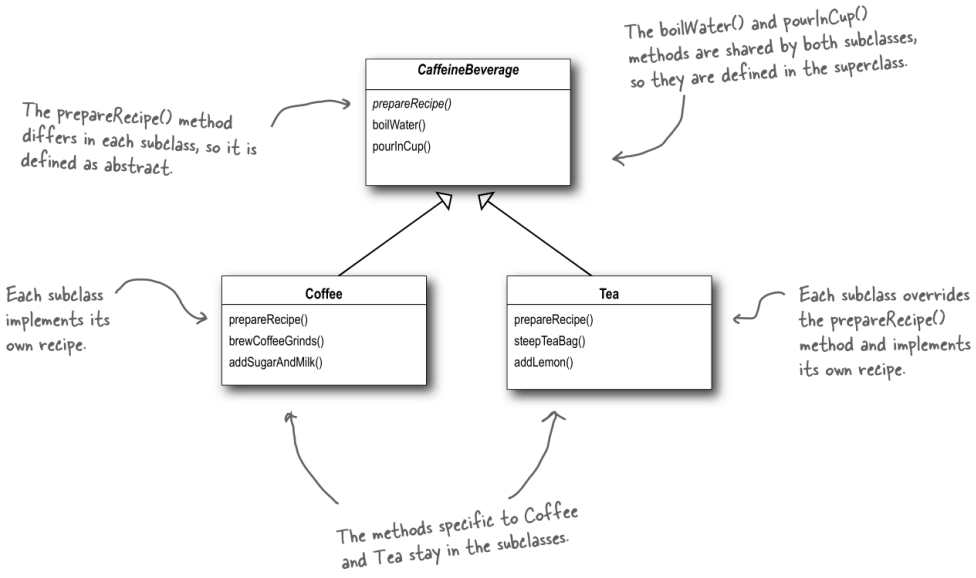
- The prepareRecipe() method is our recipe for coffee, straight out of the training manual.
  - Each of the steps is implemented as a separate method.
- Each of these methods implements one step of the algorithm.
  - There's a method to boil water, brew the coffee, pour the coffee in a cup and add sugar and milk.

```
public class Tea {
  public void prepareRecipe() {
    boilWater();
    steepTeaBag();
    pourInCup();
    addLemon();
  }

  public void boilWater() {
    System.out.println("Boiling water");
  }

  public void steepTeaBag() {
    System.out.println("Steeping the tea");
  }

  public void addLemon() {
    System.out.println("Adding Lemon");
  }

  public void pourInCup() {
    System.out.println("Pouring into cup");
  }
}
```

- The prepareRecipe() method looks very similar to the one we just implemented in Coffee; the second and forth steps are different, but it's basically the same recipe.
  - These two methods are specialized to Tea.
- Notice that the first and third methods are exactly the same as they are in Coffee! So we definitely have some code duplication going on here.

- We have code duplication occurring in these two classes.
  - boilWater() and pourInCup() are exactly the same.
- Lets get rid of the duplication.




The boilWater() and pourInCup() methods are shared by both subclasses, so they are defined in the superclass.
The prepareRecipe() method differs in each subclass, so it is defined as abstract.
CaffeineBeverage
prepareRecipe()
boilWater()
pourInCup()
Each subclass implements its own recipe.
Coffee
prepareRecipe()
brewCoffeeGrinds()
addSugarAndMilk()
Tea
prepareRecipe()
steepTeaBag()
addLemon()
Each subclass overrides the prepareRecipe() method and implements its own recipe.
The methods specific to Coffee and Tea stay in the subclasses.

- The structure of the algorithms in prepareRecipe() is similar for Tea and Coffee.
  - We can improve our code further by making the code in prepareRecipe() more abstract.

    brewCoffeeGrinds() and steepTea() -> brew()

    addSugarAndMilk() and addLemon() -> addCondiments()

- Now all we need to do is specify this structure in CaffeineBeverage.prepareRecipe() and make sure we do it in such a way so that subclasses can't change the structure.
  - By using the word "final".

```
public abstract class CaffeineBeverage {
  public final void prepareRecipe() {
    boilWater();
    brew();
    pourInCup();
    addCondiments();
  }

  public abstract void brew();
  public abstract void addCondiments();

  public void boilWater() {
    System.out.println("Boiling water");
  }

  public void pourInCup() {
    System.out.println("Pouring into cup");
  }
}
```

- Because Coffee and Tea handle these methods in different ways, they're going to have to be declared as abstract. Let the subclasses worry about that stuff!
- brew() and addCondiments() are abstract and must be supplied by subclasses.
- boilWater() and pourInCup() are specified and shared across all subclasses.

```java
public class Coffee extends CaffeineBeverage {
  public void brew() {
    System.out.println("Dripping Coffee through filter");
  }

  public void addCondiments() {
    System.out.println("Adding Sugar and Milk");
  }
}


public class Tea extends CaffeineBeverage {
  public void brew() {
    System.out.println("Steeping the tea");
  }

  public void addCondiments() {
    System.out.println("Adding Lemon");
  }
}
```

- Coffee and Tea now extend CaffeineBeverage.
- Coffee and Tea need to define brew() and addCondiments() - the two abstract methods from CaffeineBeverage.
- boilWater() and pourInCup() are specified and shared across all subclasses.

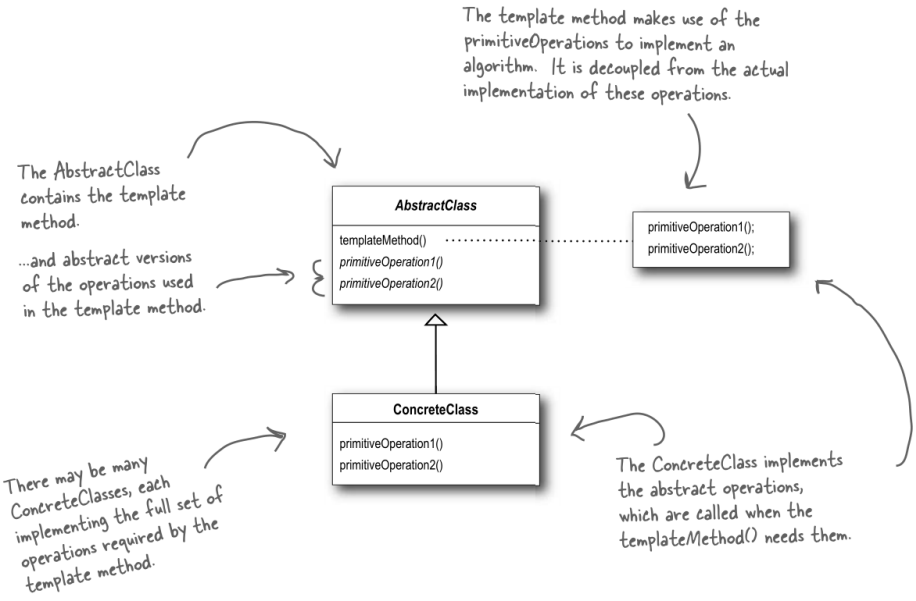
What have we done?

- Took two separate classes with separate but similar algorithms.
- Noticed duplication and eliminated it by adding a superclass.
- Made steps of algorithm more abstract and specified its structure in the superclass.
  - Thereby eliminating another "implicit" duplication between the two classes.
- Revised subclasses to implement the abstract (unspecified) portions of the algorithm… in a way that made sense for them.

Template Method Pattern

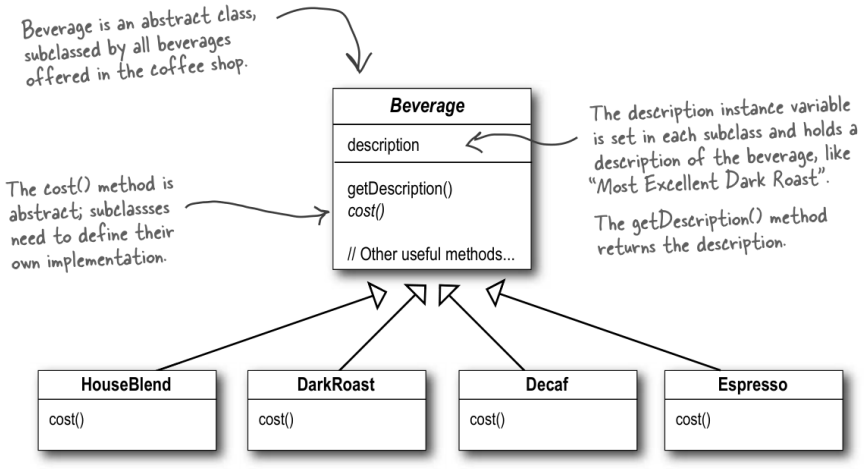
The **Template Method** pattern defines the skeleton of an algorithm in a method, deferring some steps to subclasses. **Template Method** lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

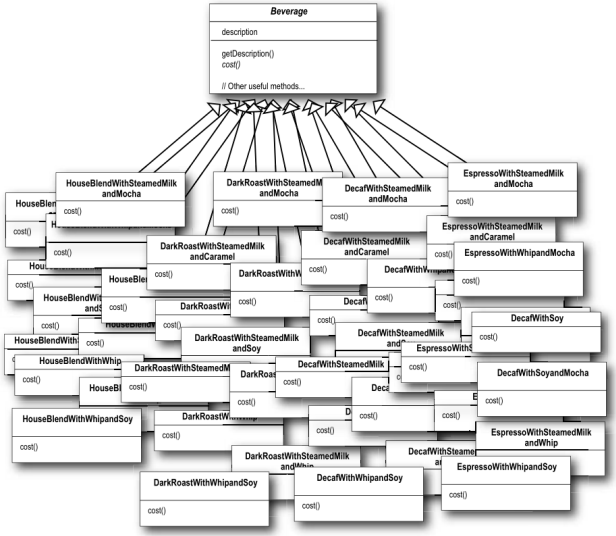
- **Template Method** defines the steps of an algorithm and allows subclasses to provide the implementation for one or more steps.
  - Makes the algorithm abstract.
    - Each step of the algorithm is represented by a method.
  - Encapsulates the details of most steps.
    - Steps (methods) handled by subclasses are declared abstract.
    - Shared steps (concrete methods) are placed in the same class that has the template method, allowing for code re-use among the various subclasses.
- This ensures the algorithm's structure stays unchanged, while subclasses provide some part of the implementation.


The template method makes use of the primitiveOperations to implement an algorithm. It is decoupled from the actual implementation of these operations.
The AbstractClass contains the template method.
...and abstract versions of the operations used in the template method.
AbstractClass
templateMethod()
primitiveOperation1()
primitiveOperation2()
primitiveOperation1();
primitiveOperation2();
ConcreteClass
primitiveOperation1()
primitiveOperation2()
There may be many ConcreteClasses, each implementing the full set of operations required by the template method.
The ConcreteClass implements the abstract operations, which are called when the templateMethod() needs them.

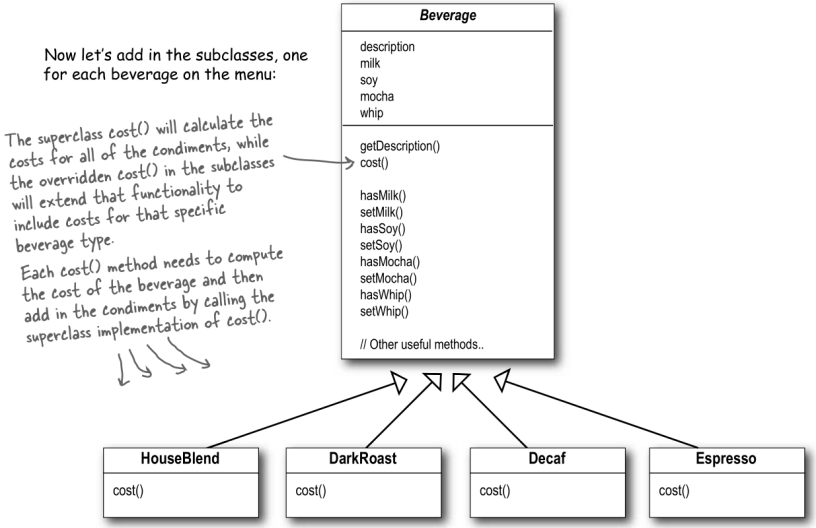
- Under pressure to update their "point of sale" system to keep up with their expanding set of beverage products.
  - Started with a Beverage abstract base class and four implementations: HouseBlend, DarkRoast, Decaf, and Espresso.
    - Each beverage can provide a description and compute its cost.
  - But they also offer a range of condiments including: steamed milk, soy, and mocha.
    - These condiments alter a beverage's description and cost.
    - The use of the word "alter" here is key since it provides a hint that we might be able to use the **Decorator Pattern**.
- With inheritance on your brain, you may add condiments to this design in one of two ways.
  - One subclass per combination of condiment.
  - Add condiment handling to the Beverage superclass.

- When they first went into business they designed their classes like this...



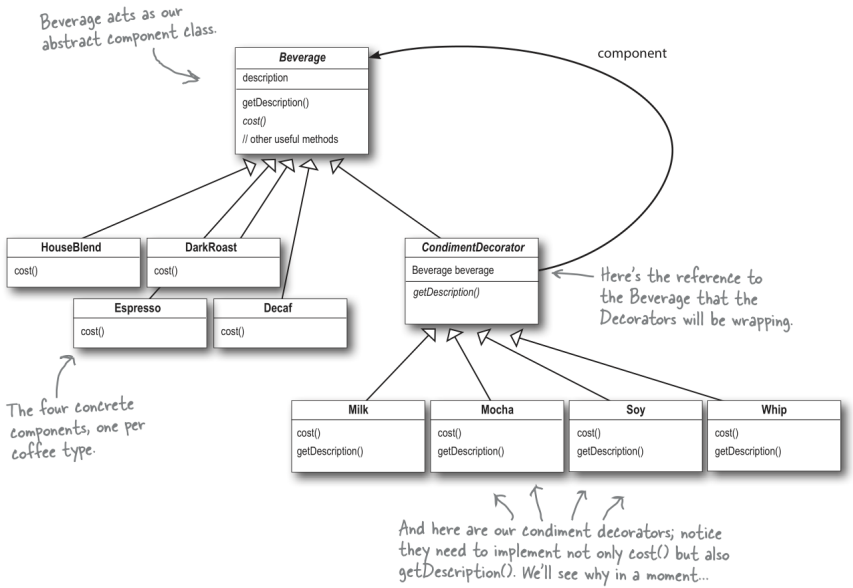


- This is incomplete, but you can see the problem ...



We have a problem

- This assumes that all concrete Beverage classes need these condiments.
- Condiments may vary (old ones go, new ones are added, price changes occur, etc.), shouldn't Beverage be encapsulated from this some how?
- What if a customer wants a double mocha?


Beverage acts as our abstract component class.
Beverage
description
getDescription()
cost()
// other useful methods
component
HouseBlend
cost()
DarkRoast
cost()
Espresso
cost()
Decaf
cost()
CondimentDecorator
Beverage beverage
getDescription()
Here's the reference to the Beverage that the Decorators will be wrapping.
The four concrete components, one per coffee type.
Milk
cost()
getDescription()
Mocha
cost()
getDescription()
Soy
cost()
getDescription()
Whip
cost()
getDescription()
And here are our condiment decorators; notice they need to implement not only cost() but also getDescription(). We'll see why in a moment...

```java
public abstract class Beverage {
  protected String description = "Unknown Beverage";

  public String getDescription() {
    return description;
  }

  public abstract double cost();
}

public abstract class CondimentDecorator extends Beverage {
  protected Beverage beverage;

  public CondimentDecorator(Beverage beverage) {
    this.beverage = beverage;
  }

  public abstract String getDescription();
}
```

```java
public class Espresso extends Beverage {
  public Espresso() {
    this.description = "Espresso";
  }

  public double cost() {
    return 1.99;
  }
}

public class HouseBlend extends Beverage {
  public HouseBlend() {
    this.description = "House Blend Coffee";
  }

  public double cost() {
    return 0.89;
  }
}
```

```java
public class DarkRoast extends Beverage {
  public DarkRoast() {
    this.description = "Dark Roast Coffee";
  }

  public double cost() {
    return 0.99;
  }
}

public class Decaf extends Beverage {
  public Decaf() {
    this.description = "Decaf Coffee";
  }

  public double cost() {
    return 1.05;
  }
}
```

```java
public class Mocha extends CondimentDecorator {
  public Mocha(Beverage beverage) {
    super(beverage};
  }

  public String getDescription() {
    return beverage.getDescription() + ", Mocha";
  }

  public double cost() {
    return 0.20 + beverage.cost();
  }
}
```

```java
public class Soy extends CondimentDecorator {
  public Soy(Beverage beverage) {
    super(beverage};
  }

  public String getDescription() {
    return beverage.getDescription() + ", Soy";
  }

  public double cost() {
    return 0.15 + beverage.cost();
  }
}
```

```java
public class Whip extends CondimentDecorator {
  public Whip(Beverage beverage) {
    super(beverage};
  }

  public String getDescription() {
    return beverage.getDescription() + ", Whip";
  }

  public double cost() {
    return 0.10 + beverage.cost();
  }
}
```

```java
public class Milk extends CondimentDecorator {
  public Milk(Beverage beverage) {
    super(beverage};
  }

  public String getDescription() {
    return beverage.getDescription() + ", Milk";
  }

  public double cost() {
    return 0.10 + beverage.cost();
  }
}
```

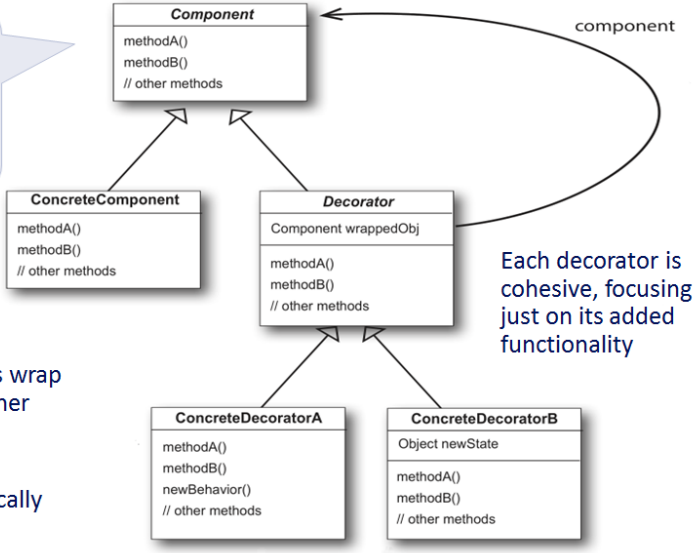
```java
public class StarbuzzCoffee {
  public static void main(String args[]) {
    Beverage beverage = new Espresso();
    System.out.println(beverage.getDescription() + " $" + beverage.cost());

    // Make a DarkRoast object
    Beverage beverage2 = new DarkRoast();
    // Wrap it with a Mocha.
    beverage2 = new Mocha(beverage2);
    // Wrap it in a second Mocha
    beverage2 = new Mocha(beverage2);
    // Wrap it in a Whip
    beverage2 = new Whip(beverage2);
    System.out.println(beverage2.getDescription() + " $" + beverage2.cost());

    Beverage beverage3 = new HouseBlend();
    beverage3 = new Soy(beverage3);
    beverage3 = new Mocha(beverage3);
    beverage3 = new Whip(beverage3);
    System.out.println(beverage3.getDescription() + " $" + beverage3.cost());
  }
}
```

Decorator Pattern

The **Decorator Pattern** is a design pattern that allows behavior to be added to an individual object, dynamically, without affecting the behavior of other objects from the same class.

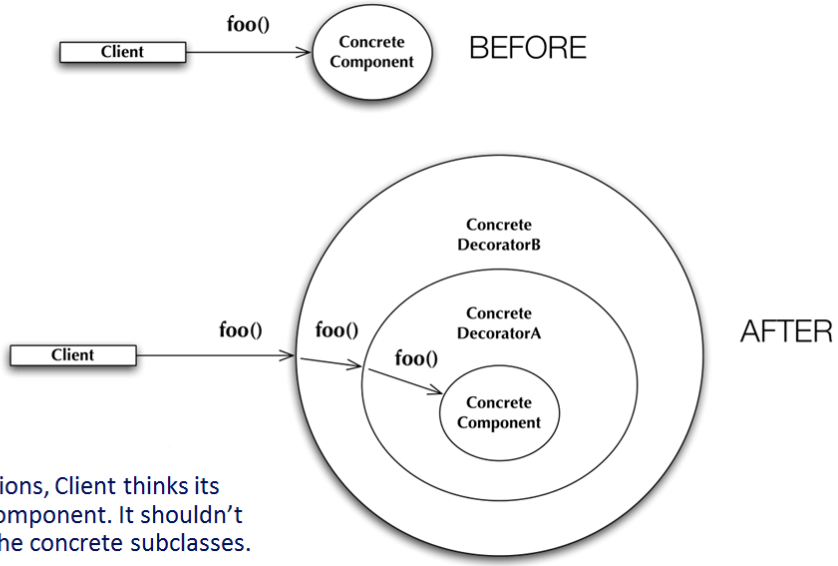
- The **Decorator Pattern** provides a powerful mechanism for adding new behaviors to an object at runtime.
  - The mechanism is based on the notion of "wrapping" which is just a fancy way of saying "delegation" but with the added twist that the delegator and the delegate both implement the same interface.
- The decorator pattern provides yet another way in which a class's runtime behavior can be extended without requiring modification to the class.
  - This supports the goal of the **Open-Closed Principle**: Classes should be open for extension but closed to modification.
    - Inheritance is one way to do this, but composition and delegation are more flexible (and **Decorator** takes advantage of delegation).
    - As the Gang of Four put it: "**Decorator** lets you attach additional responsibilities to an object dynamically. **Decorators** provide a flexible alternative to subclassing for extending functionality."

Inheritance is used to make sure that components and decorators share the same interface: namely the public interface of Component which is either an abstract class or an interface

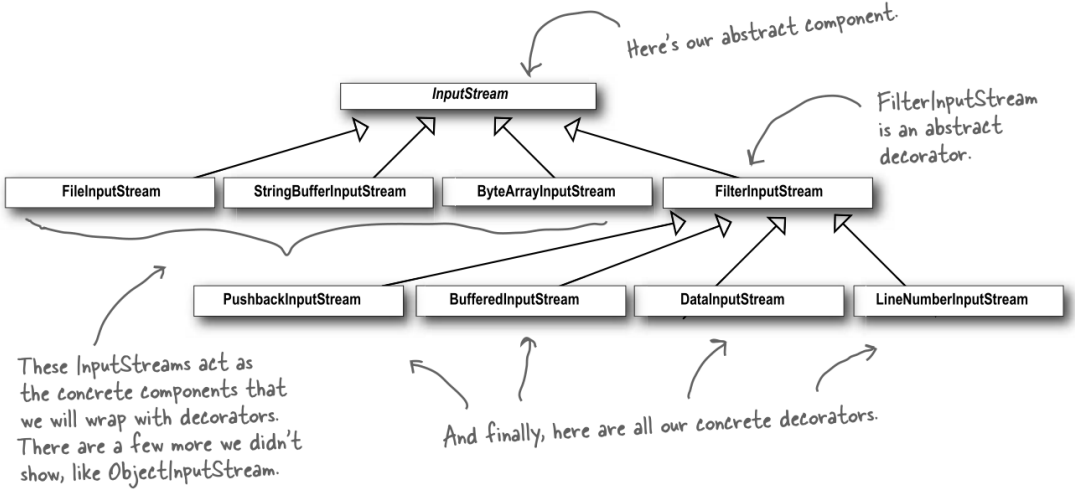


Each decorator is cohesive, focusing just on its added functionality

- At run-time, concrete decorators wrap concrete components and/or other concrete decorators
- The object to be wrapped is typically passed in via the constructor



- In both situations, Client thinks its talking to a Component. It shouldn't know about the concrete subclasses.


Here's our abstract component.
InputStream
FilterInputStream is an abstract decorator.
FileInputStream
StringBufferInputStream
ByteArrayInputStream
FilterInputStream
PushbackInputStream
BufferedInputStream
DataInputStream
LineNumberInputStream
These InputStreams act as the concrete components that we will wrap with decorators. There are a few more we didn't show, like ObjectInputStream.
And finally, here are all our concrete decorators.

- As we saw, **Decorator** offers another solution to the problem of rapidly multiplying combinations of subclasses.
  - We saw examples of other solutions when we made use of the Strategy Pattern and the Bridge Pattern.
- The **Decorator Pattern** provides a means for creating different combinations of functionality by creating chains in which each member of the chain can augment or "decorate" the output of the previous member.
  - Plus, it separates the step of building these chains from the use of these chains.
- The **Decorator Pattern** comes into play when there are a variety of optional functions that can precede or follow another function that is always executed.
- This is a very powerful idea that can be implemented in a variety of ways.
  - The fact that all of the classes in the decorator pattern hide behind the abstraction of Component enables all of the good benefits of OO design discussed previously.

- When you use new you are certainly instantiating a concrete class, so that's definitely an implementation, not an interface.

```
Duck duck;
if (picnic) {
  duck = new MallarDuck();
} else if (huntin) {
  duck = new DecoyDuck();
} else {
  duck = new RubberDuck();
}
```

- When you see code like this, you know that when it comes time for changes or extensions, you'll have to reopen this code and examine what needs to be added (or deleted). Often this kind of code ends up in several parts of the application making maintenance and updates more difficult and error-prone.

```
public class PizzaStore {
  public Pizza orderPizza(String type) {
    Pizza pizza;
    if (type.equals("cheese")) {
      pizza = new CheesePizza();
    } else if (type.equals("greek")) {
      pizza = new GreekPizza();
    } else if (type.equals("pepperoni")) {
      pizza = new PepperoniPizza();
    }

    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();
    return pizza;
  }
}
```

- We have a pizza store program that wants to separate the process of creating a pizza from the process of preparing / ordering a pizza.
  - This code is an excellent example of "coding to an interface",
  - But, it is NOT closed for modification. If the Pizza store changes its pizza offerings, we have to open this code and modify it.

```
public class PizzaStore {
  // A reference to a SimplePizzaFactory.
  private SimplePizzaFactory factory;

  public PizzaStore(SimplePizzaFactory factory) {
    // The factory passed to it in the constructor.
    this.factory = factory;
  }

  public Pizza orderPizza(String type) {
    // Using the factory to create its pizzas by
    //  simply passing on the type of the order.
    Pizza pizza = factory.createPizza(type);

    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();

    return pizza;
  }
}
```

- A simple way to encapsulate the code of pizza creation and move it out into another object that is only going to be concerned with creating pizzas.
  - That new class depends on the concrete classes, but those dependencies no longer impact the preparation code.

```
public class SimplePizzaFactory {
  public Pizza createPizza(String type) {
    if (type.equals("cheese"))
      return new CheesePizza();

    if (type.equals("greek")) {
      return new GreekPizza();

    if (type.equals("pepperoni")) {
      return new PepperoniPizza();

    if (type.equals("clam")) {
      return new ClamPizza();

    return null;
  }
}
```

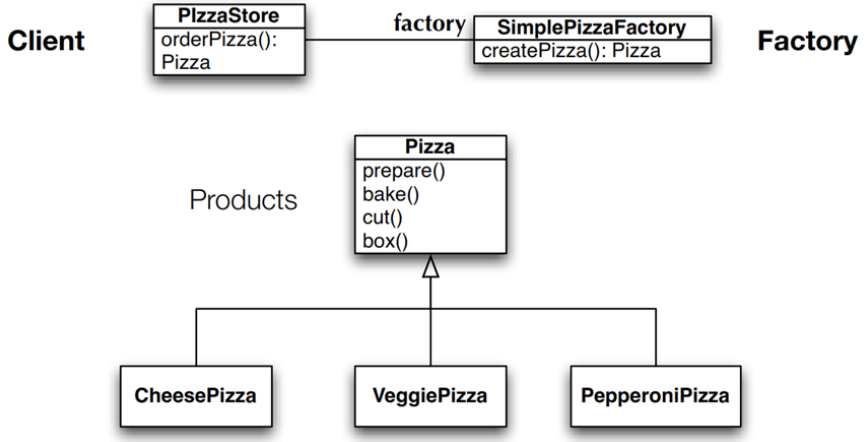
- The new object is called a **Factory**.
- Factories handle the details of object creation. Once we have a SimplePizzaFactory, our orderPizza() method becomes a client of that object. Anytime it needs a pizza, it asks the pizza factory to make one.
- Now the orderPizza() method just cares that it gets a pizza that implements the Pizza interface so that it can call prepare(), bake(), cut(), and box().

```java
public abstract class Pizza {
  protected String name;
  protected String dough;
  protected String sauce;
  protected List<String> toppings;

  public Pizza() {
    this.toppings = new ArrayList<>();
  }

  public String getName() {
    return name;
  }

  public void prepare() {
    System.out.println("Preparing " + name);
  }

  public void bake() {
    System.out.println("Baking " + name);
  }
```

```
  public void cut() {
    System.out.println("Cutting " + name);
  }

  public void box() {
    System.out.println("Boxing " + name);
  }

  public String toString() {
    // Code to display pizza name and ingredients
    StringBuilder display = new StringBuilder();
    display.append("---- " + name + " ----\n");
    display.append(dough + "\n");
    display.append(sauce + "\n");
    for (String topping : toppings) {
      display.append(topping + "\n");
    }

    return display.toString();
  }
}
```

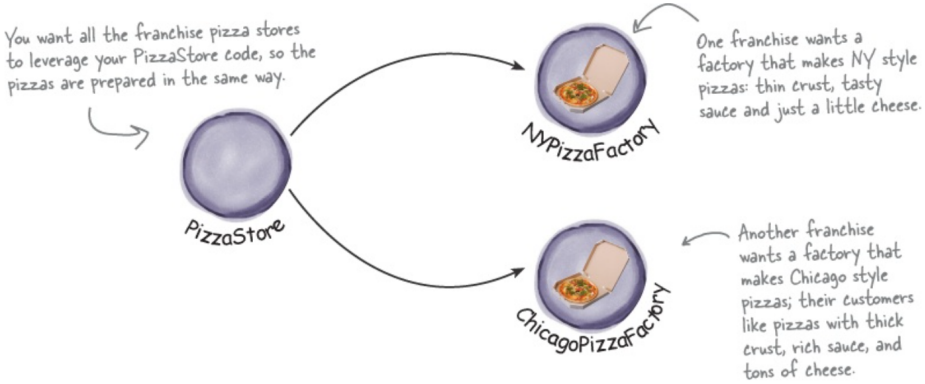
```java
public class CheesePizza extends Pizza {
  public CheesePizza() {
    this.name = "Cheese Pizza";
    this.dough = "Regular Crust";
    this.sauce = "Marinara Pizza Sauce";
    this.toppings.add("Fresh Mozzarella");
    this.toppings.add("Parmesan");
  }
}


public class PepperoniPizza extends Pizza {
  public PepperoniPizza() {
    this.name = "Pepperoni Pizza";
    this.dough = "Crust";
    this.sauce = "Marinara sauce";
    this.toppings.add("Sliced Pepperoni");
    this.toppings.add("Sliced Onion");
    this.toppings.add("Grated parmesan cheese");
  }
}

...
```

```java
public class SimplePizzaTestDrive {
  public static void main(String[] args) {
    SimplePizzaFactory factory = new SimplePizzaFactory();
    PizzaStore store = new PizzaStore(factory);

    Pizza pizza = store.orderPizza("cheese");
    System.out.println("We ordered a " + pizza.getName() + "\n");

    pizza = store.orderPizza("veggie");
    System.out.println("We ordered a " + pizza.getName() + "\n");
  }
}
```

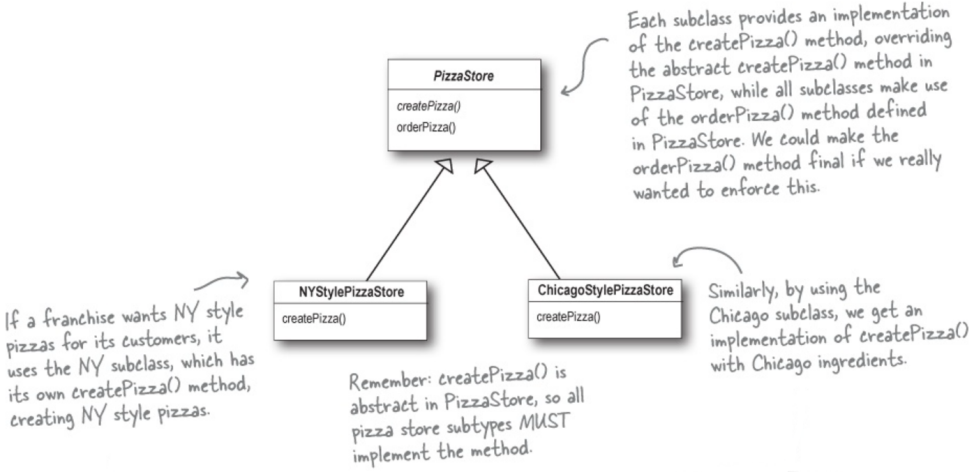


- While this is nice, it is not as flexible as it can be. To increase flexibility we need to look at two design patterns: **Factory Method** and **Abstract Factory**.

- What about regional differences? Each franchise might want to offer different styles of pizzas (New York, Chicago, and California, to name a few), depending on where the franchise store is located and the tastes of the local pizza connoisseurs.



- What varies among the regional PizzaStores is the style of pizzas they make. We can push all these variations into the createPizza() method and make it responsible for creating the right kind of pizza.



```java
public abstract class PizzaStore {
  protected abstract Pizza createPizza(String type);

  public Pizza orderPizza(String type) {
    Pizza pizza = createPizza(type);

    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();

    return pizza;
  }
}
```

- This class is a (very simple) OO framework. The framework provides one service: "order Pizza".
- The framework invokes the createPizza() factory method to create a pizza that it can then prepare using a well-defined, consistent process.
- A "client" of the framework will subclass this class and provide an implementation of the createPizza() method.

```
public class NYPizzaStore extends PizzaStore {
  public Pizza createPizza(String style) {
    if (style.equals("cheese"))
      return new NYStyleCheesePizza();

    if (style.equals("veggie"))
      return new NYStyleVeggiePizza();

    if (style.equals("clam"))
      return new NYStyleClamPizza();

    if (style.equals("pepperoni"))
      return new NYStylePepperoniPizza();
    }

    return null;
  }
}
```

- Nice and simple. If you want a NY-style Pizza, you create an instance of this class and call orderPizza() passing in the type. The subclass makes sure that the pizza is created using the correct style.

```
public class ChicagoPizzaStore extends PizzaStore {
  public Pizza createPizza(String style) {
    if (style.equals("cheese"))
      return new ChicagoStyleCheesePizza();

    if (style.equals("veggie"))
      return new ChicagoStyleVeggiePizza();

    if (style.equals("clam"))
      return new ChicagoStyleClamPizza();

    if (style.equals("pepperoni"))
      return new ChicagoStylePepperoniPizza();

    return null;
  }
}
```

- If you need a different style, create a new subclass.

```java
public class NYStylePepperoniPizza extends Pizza {
  public NYStylePepperoniPizza() {
    this.name = "NY Style Pepperoni Pizza";
    this.dough = "Thin Crust Dough";
    this.sauce = "Marinara Sauce";
    this.toppings.add("Grated Reggiano Cheese");
    this.toppings.add("Sliced Pepperoni");
    this.toppings.add("Garlic");
    this.toppings.add("Onion");
    this.toppings.add("Mushrooms");
    this.toppings.add("Red Pepper");
  }

  void cut() {
    System.out.println("Cutting the pizza into 8 slices");
  }
}
```

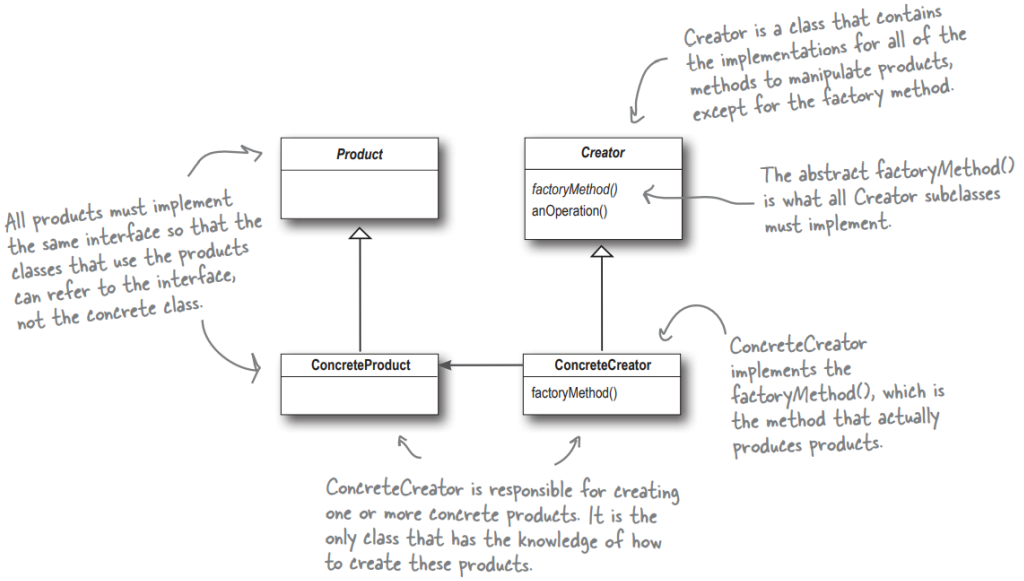
```java
public class ChicagoStylePepperoniPizza extends Pizza {
  public ChicagoStylePepperoniPizza() {
    this.name = "Chicago Style Pepperoni Pizza";
    this.dough = "Extra Thick Crust Dough";
    this.sauce = "Plum Tomato Sauce";
    this.toppings.add("Shredded Mozzarella Cheese");
    this.toppings.add("Black Olives");
    this.toppings.add("Spinach");
    this.toppings.add("Eggplant");
    this.toppings.add("Sliced Pepperoni");
  }

  void cut() {
    System.out.println("Cutting the pizza into square slices");
  }
}
```

```java
public class FactoryMethodPizzaTestDrive {
  public static void main(String[] args) {
    PizzaStore nyStore = new NYPizzaStore();
    PizzaStore chicagoStore = new ChicagoPizzaStore();

    Pizza pizza = nyStore.orderPizza("pepperoni");
    System.out.println("Ethan ordered a " + pizza.getName() + "\n");

    Pizza pizza = chicagoStore.orderPizza("pepperoni");
    System.out.println("Joel ordered a " + pizza.getName() + "\n");
  }
}
```

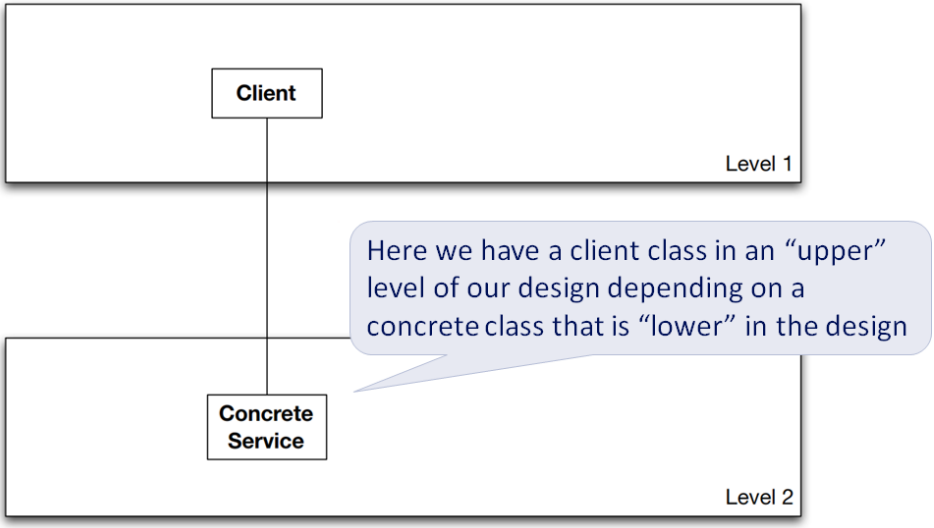
Factory Method

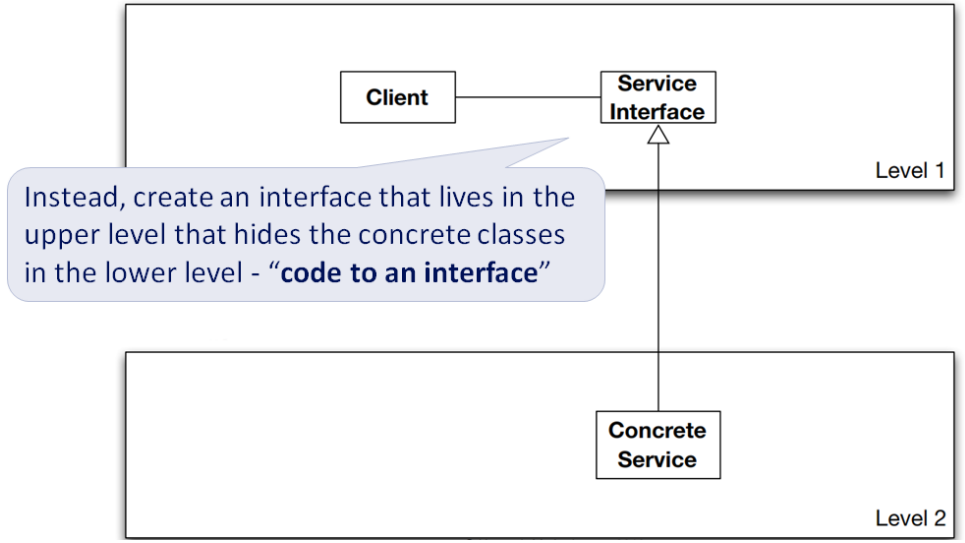
The **Factory Method Pattern** defines an interface for creating an object, but lets subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

- The **Factory Method Pattern** gives us a way to encapsulate the instantiations of concrete types.
- The Pattern gives you an interface with a method for creating objects, also known as the "factory method".
- The subclasses actually implement the factory method and create products.


Creator is a class that contains the implementations for all of the methods to manipulate products, except for the factory method.
Product
Creator
factoryMethod()
anOperation()
The abstract factoryMethod() is what all Creator subclasses must implement.
All products must implement the same interface so that the classes that use the products can refer to the interface, not the concrete class.
ConcreteProduct
ConcreteCreator
factoryMethod()
ConcreteCreator implements the factoryMethod(), which is the method that actually produces products.
ConcreteCreator is responsible for creating one or more concrete products. It is the only class that has the knowledge of how to create these products.

- **Factory Method** is one way of following the **Dependency Inversion Principle**.
  - "Depend upon abstractions. Do not depend upon concrete classes."
- Normally "high-level" classes depend on "low-level" classes.
  - Instead, they BOTH should depend on an abstract interface.


Client
Level 1
Here we have a client class in an “upper” level of our design depending on a concrete class that is “lower” in the design
Concrete Service
Level 2


Client
Service Interface
Level 1
Instead, create an interface that lives in the upper level that hides the concrete classes in the lower level - "code to an interface"
Concrete Service
Level 2

- To achieve the **Dependency Inversion Principle** in your own designs, follow these GUIDELINES.
  - No variable should hold a reference to a concrete class.
    - If you use new, you'll be holding a reference to a concrete class. Use a factory to get around that!
  - No class should derive from a concrete class.
    - If you derive from a concrete class, you're depending on a concrete class. Derive from an abstraction, like an interface or an abstract class.
  - No method should override an implemented method of its base classes.
    - If you override an implemented method, then your base class wasn't really an abstraction to start with. Those methods implemented in the base class are meant to be shared by all your subclasses.
- These are "guidelines" because if you were to blindly follow these instructions, you would never produce a system that could be compiled or executed.
  - Instead use them as instructions to help optimize your design.
- And remember, not only should low-level classes depend on abstractions, but high-level classes should to … This is the very embodiment of "code to an interface".

- The **Factory Method** approach to the pizza store is a big success, allowing our company to create multiple franchises across the country quickly and easily.
  - But (bad news): we have learned that some of the franchises.
    - while following our procedures (the abstract code in PizzaStore forces them to)
    - are skimping on ingredients in order to lower costs and increase margins.
  - Our company's success has always been dependent on the use of fresh, quality ingredients.
    - So, something must be done!

- We will alter our design such that a factory is used to supply the ingredients that are needed during the pizza creation process.
  - Since different regions use different types of ingredients, we'll create region-specific subclasses of the ingredient factory to ensure that the right ingredients are used.
  - But, even with region-specific requirements, since we are supplying the factories, we'll make sure that ingredients that meet our quality standards are used by all franchises.
    - They'll have to come up with some other way to lower costs.

- Now there is only one problem with this plan: the franchises are located in different regions and what is red sauce in New York is not red sauce in Chicago. So, you have one set of ingredients that needs to be shipped to New York and a different set that needs to be shipped to Chicago.

- Now we're going to build a factory to create our ingredients; the factory will be responsible for creating each ingredient in the ingredient family. In other words, the factory will need to create dough, sauce, cheese, and so on...
- Let's start by defining an interface for the factory that is going to create all our ingredients:

```
public interface PizzaIngredientFactory {
  Dough createDough();
  Sauce createSauce();
  Cheese createCheese();
  Veggie[] createVeggies();
  Pepperoni createPepperoni();
  Clams createClams();
}
```

```
public class ChicagoPizzaIngredientFactory implements PizzaIngredientFactory {
  public Dough createDough() {
    return new ThickCrustDough();
  }

  public Sauce createSauce() {
    return new PlumTomatoSauce();
  }

  public Cheese createCheese() {
    return new MozzarellaCheese();
  }

  public Veggies[] createVeggies() {
    Veggies veggies[] = {new BlackOlives(), new Spinach(), new Eggplant()};
    return veggies;
  }
  ...
}
```

```
1  public abstract class Pizza {
     protected String name;
3    protected Dough dough;
     protected Sauce sauce;
5    protected Veggies veggies[];
     protected Cheese cheese;
7    protected Pepperoni pepperoni;
     protected Clams clam;
9
     public abstract void prepare();
11
     void bake() {
13     System.out.println("Bake for 25 minutes at 350");
     }
15
     void cut() {
17     System.out.println("Cutting the pizza into diagonal slices");
     }
```

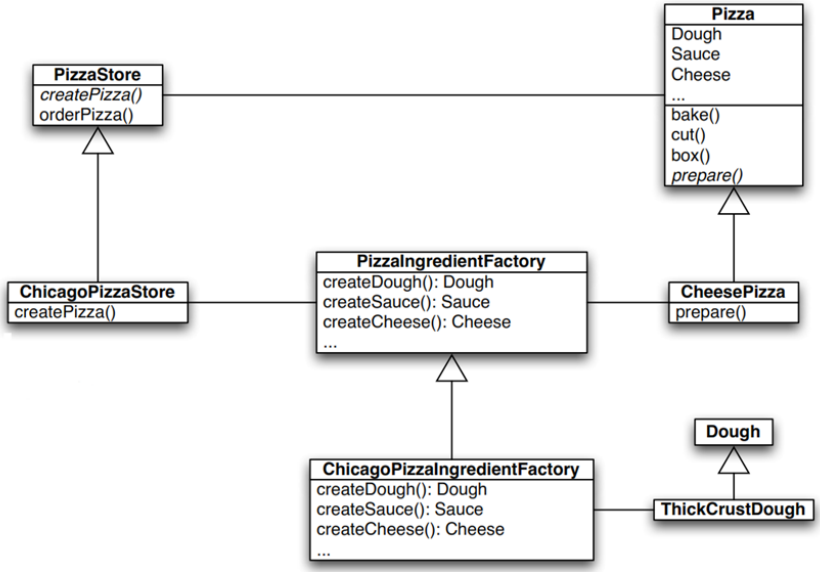
```java
public class CheesePizza extends Pizza {
  private PizzaIngredientFactory ingredientFactory;

  public CheesePizza(PizzaIngredientFactory ingredientFactory) {
    this.ingredientFactory = ingredientFactory;
  }

  public void prepare() {
    System.out.println("Preparing " + name);
    this.dough = ingredientFactory.createDough();
    this.sauce = ingredientFactory.createSauce();
    this.cheese = ingredientFactory.createCheese();
  }
}
```

- We no longer need subclasses like NYCheesePizza and ChicagoCheesePizza because the ingredient factory now handles regional differences.

```
public class ChicagoPizzaStore extends PizzaStore {
  pulic Pizza createPizza(String style) {
    Pizza pizza = null;
    PizzaIngredientFactory ingredientFactory =
      new ChicagoPizzaIngredientFactory();

    if (style.equals("cheese")) {
      pizza = new CheesePizza(ingredientFactory);
      pizza.setName("Chicago Style Cheese Pizza");
    } else if (style.equals("veggie")) {
      pizza = new VeggiePizza(ingredientFactory);
      pizza.setName("Chicago Style Veggie Pizza");
    }
    ...

    return pizza;
  }
}
```

Summary: What did we just do?

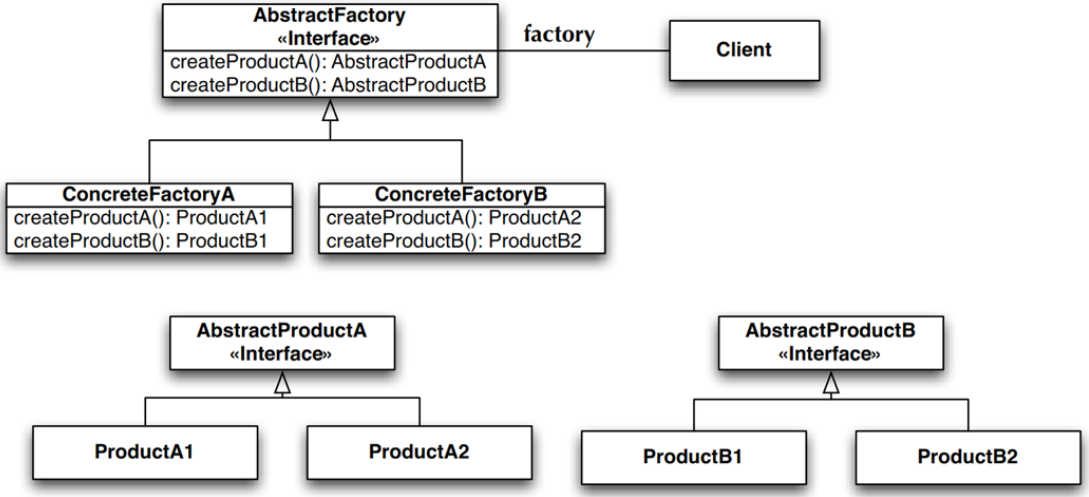
- We created an ingredient factory interface to allow for the creation of a family of ingredients for a particular pizza.
- This abstract factory gives us an interface for creating a family of products (e.g., NY pizzas, Chicago pizzas).
  - The factory interface decouples the client code from the actual factory implementations that produce context-specific sets of products.
- Our client code (PizzaStore) can then pick the factory appropriate to its region, plug it in, and get the correct style of pizza (**Factory Method**) with the correct set of ingredients (**Abstract Factory**).


PizzaStore
createPizza()
orderPizza()
Pizza
Dough
Sauce
Cheese
...
bake()
cut()
box()
prepare()
ChicagoPizzaStore
createPizza()
PizzaIngredientFactory
createDough(): Dough
createSauce(): Sauce
createCheese(): Cheese
...
CheesePizza
prepare()
Dough
ChicagoPizzaIngredientFactory
createDough(): Dough
createSauce(): Sauce
createCheese(): Cheese
...
ThickCrustDough

Abstract Factory

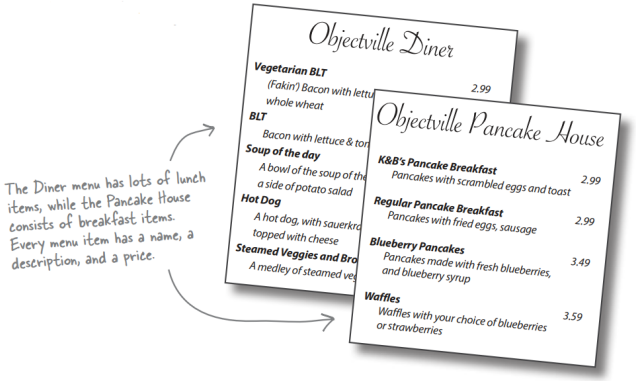
The Abstract Factory Pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes.

- We've certainly seen that **Abstract Factory** allows a client to use an abstract interface to create a set of related products without knowing (or caring) about the concrete products that are actually produced.
- In this way, the client is decoupled from any of the specifics of the concrete products.


AbstractFactory
«Interface»
createProductA(): AbstractProductA
createProductB(): AbstractProductB
factory
Client
ConcreteFactoryA
createProductA(): ProductA1
createProductB(): ProductB1
ConcreteFactoryB
createProductA(): ProductA2
createProductB(): ProductB2
AbstractProductA
«Interface»
ProductA1
ProductA2
AbstractProductB
«Interface»
ProductB1
ProductB2

- All factories encapsulate object creation.
- **Simple Factory**, while not a bona fide design pattern, is a simple way to decouple your clients from concrete classes.
- **Factory Method** relies on inheritance: object creation is delegated to subclasses, which implement the factory method to create objects.
- **Abstract Factory** relies on object composition: object creation is implemented in methods exposed in the factory interface.
- All factory patterns promote loose coupling by reducing the dependency of your application on concrete classes.
- The intent of Factory Method is to allow a class to defer instantiation to its subclasses.
- The intent of **Abstract Factory** is to create families of related objects without having to depend on their concrete classes.
- The **Dependency Inversion Principle** guides us to avoid dependencies on concrete types and to strive for abstractions.
- Factories are a powerful technique for coding to abstractions, not concrete classes.

- There are lots of ways to stuff objects into a collection.
- Put them into an Array, a Stack, a List, a Hash Map - take your pick. Each has its own advantages and tradeoffs.
- How you can iterate through your objects without ever getting a peek at how you store your objects.



```
public class MenuItem {
  private String name;
  private String description;
  private double price;

  public MenuItem(String name,
                  String description,
                  double price) {
    this.name = name;
    this.description = description;
    this.price = price;
  }

  public String getName() {
    return name;
  }

  ...
```

```
public class PancakeHouseMenu {
  private List<MenuItem> menuItems;

  public PancakeHouseMenu() {
    menuItems = new ArrayList<MenuItem>();
    addItem("K&B's Pancake Breakfast",
      "Pancakes with scrambled eggs and toast",
      2.99);
    addItem("Regular Pancake Breakfast",
      "Pancakes with fried eggs, sausage",
      2.99);
  }

  public void addItem(String name, String description, double price) {
    MenuItem menuItem = new MenuItem(name, description, price);
    menuItems.add(menuItem);
  }
  ...
```

```
public class DinerMenu {
  private static final int MAX_ITEMS = 6;
  private int numberOfItems = 0;
  private MenuItem[] menuItems;

  public DinerMenu() {
    menuItems = new MenuItem[MAX_ITEMS];
    addItem("Vegetarian BLT",
      "(Fakin') Bacon with lettuce & tomato on whole wheat", true, 2.99);
    addItem("BLT",
      "Bacon with lettuce & tomato on whole wheat", false, 2.99);

  public void addItem(String name, String description, double price) {
    MenuItem menuItem = new MenuItem(name, description, price);
    menuItems[numberOfItems] = menuItem;
    numberOfItems = numberOfItems + 1;
  }
  ...
```

- Now, to print out the items from the PancakeHouseMenu, we'll loop through the items on the breakfastItems ArrayList. And to print out the Diner items, we'll loop through the Array.

```
for (int i = 0; i < breakfastItems.size(); i++) {
    MenuItem menuItem = breakfastItems.get(i);
    System.out.print(menuItem.getName() + " ");
    System.out.println(menuItem.getPrice() + " ");
    System.out.println(menuItem.getDescription());
}

for (int i = 0; i < lunchItems.length; i++) {
    MenuItem menuItem = lunchItems[i];
    System.out.print(menuItem.getName() + " ");
    System.out.println(menuItem.getPrice() + " ");
    System.out.println(menuItem.getDescription());
}
```

- Now what if we create an object, let's call it an Iterator, that encapsulates the way we iterate through a collection of objects?

```
public interface Iterator {
  boolean hasNext();
  MenuItem next();
}


Iterator iterator = breakfastMenu.createIterator();
while (iterator.hasNext()) {
  MenuItem menuItem = iterator.next();
}

Iterator iterator = lunchMenu.createIterator();
while (iterator.hasNext()) {
  MenuItem menuItem = iterator.next();
}
```

```java
public class DinerMenuIterator implements Iterator {
  private MenuItem[] items;
  private int position = 0;

  public DinerMenuIterator(MenuItem[] items) {
    this.items = items;
  }

  public MenuItem next() {
    MenuItem menuItem = items[position];
    position++;
    return menuItem;
  }

  public boolean hasNext() {
    if (position >= items.length || items[position] == null) {
      return false;
    }

    return true;
  }
}
```

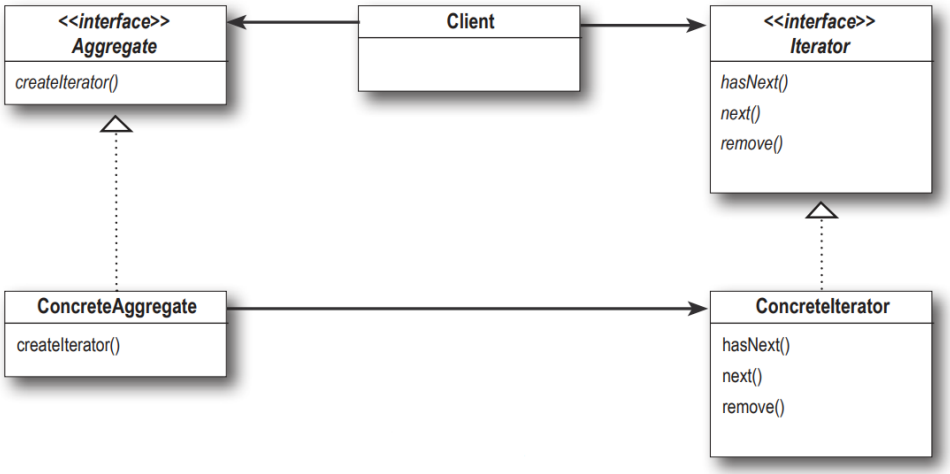
```
public class DinerMenu {
  private static final int MAX_ITEMS = 6;
  private int numberOfItems = 0;
  private MenuItem[] menuItems;

  // constructor here

  // addItem here

  /* We are not going to need the getMenuItems() method anymore; in fact,
     we do not want it because it exposes our internal implementation! */
  public MenuItem[] getMenuItems() {
    return menuItems;
  }

  public Iterator createIterator() {
    return new DinerMenuIterator(menuItems);
  }
}
```

```
public class MenuTestDrive {
  public static void main(String[] args) {
    Iterator pancakeIterator = pancakeHouseMenu.createIterator();
    printMenu(pancakeIterator);

    Iterator dinerIterator = dinerMenu.createIterator();
    printMenu(dinerIterator);
  }

  private static void printMenu(Iterator iterator) {
    while (iterator.hasNext()) {
      MenuItem menuItem = iterator.next();
      System.out.print(menuItem.getName() + ", ");
      System.out.print(menuItem.getPrice() + " -- ");
      System.out.println(menuItem.getDescription());
    }
  }
}
```

Iterator Pattern

The Iterator Pattern provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

- The Iterator Pattern allows traversal of the elements of an aggregate without exposing the underlying implementation
- It also places the task of traversal on the iterator object, not on the aggregate, which simplifies the aggregate interface and implementation, and places the responsibility where it should be.


<<interface>>
Aggregate
createIterator()
Client
<<interface>>
Iterator
hasNext()
next()
remove()
ConcreteAggregate
createIterator()
ConcreteIterator
hasNext()
next()
remove()

References

- Allen B. Downey, Chris Mayfield, ***Think Java***, (2016).
- Graham Mitchell, ***Learn Java the Hard Way***, 2nd Edition, (2016).
- Cay S. Horstmann, ***Big Java - Early Objects***, 7e-Wiley, (2019).
- James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, ***The Java Language Specification - Java SE 8 Edition***, (2015).
- Martin Fowler, ***UML Distilled - A Brief Guide To The Standard Object Modeling Language***, (2004).
- Richard Warburton, ***Object-Oriented vs. Functional Programming - Bridging the Divide Between Opposing Paradigms***, (2016).
- Naftalin, Maurice Wadler, Philip ***Java Generics and Collections***, O'Reilly Media, (2009).
- Eric Freeman, Elisabeth Robson ***Head First Design Patterns - Building Extensible and Maintainable Object***, Oriented Software-O'Reilly Media, (2020).
- Alexander Shvets ***Dive Into Design Patterns***, (2019).

Thank You!