CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Lecture 7: Design Patterns for Variable and Evolving Systems

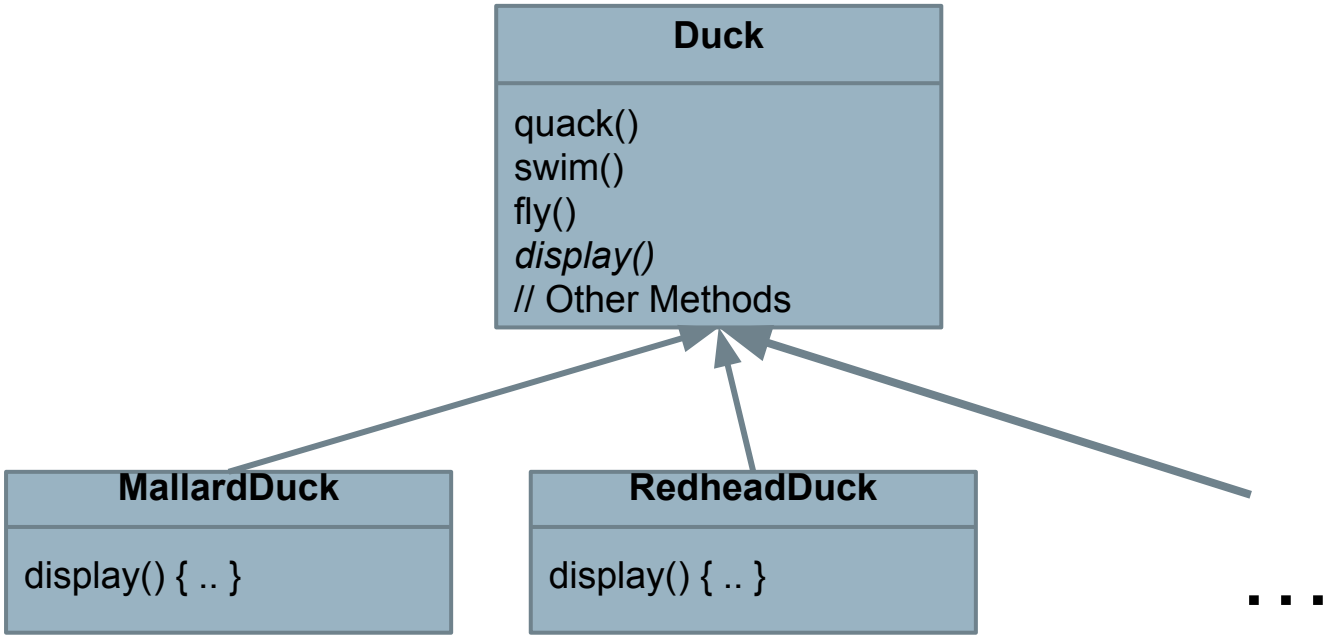
Gregory Gay
TDA/ DIT 594 - November 24, 2020

Today's Goals

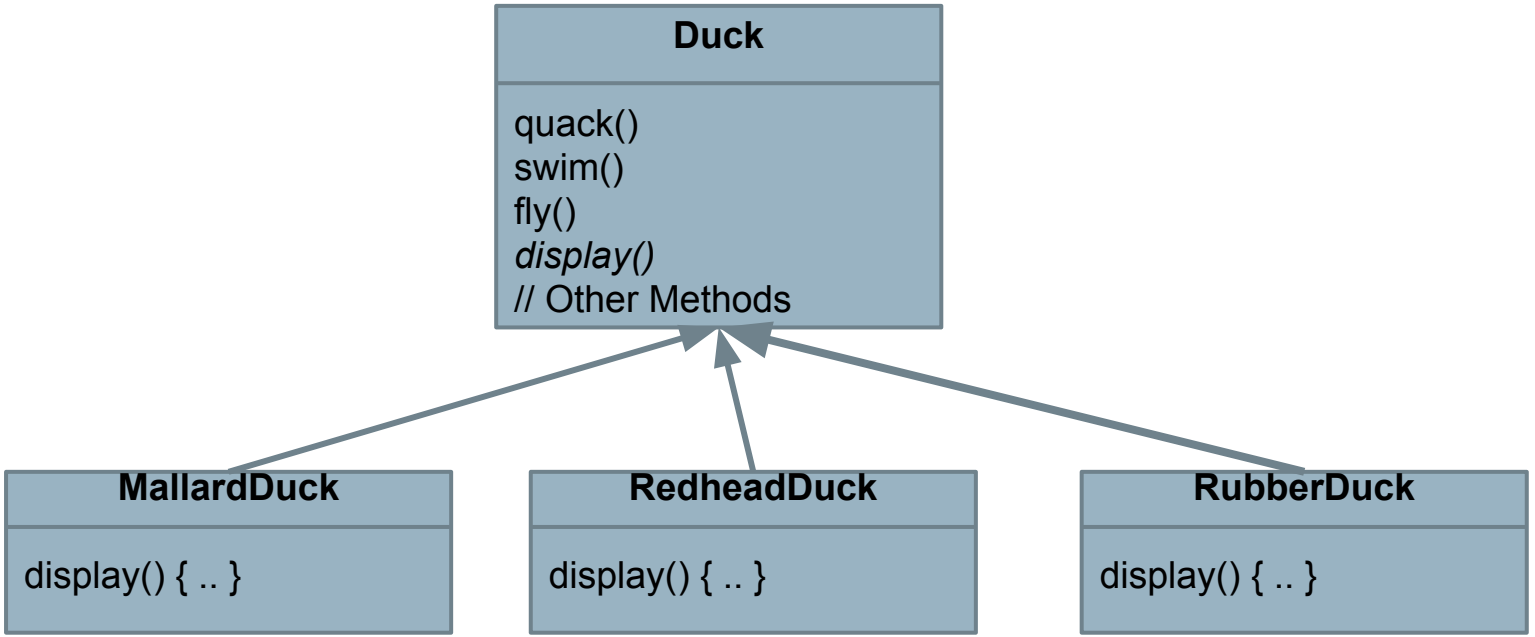
- Using design patterns to implement variability.
  - Strategy Pattern
  - Factory Pattern
  - Decorator Pattern
  - Adapter Pattern
  - Facade Pattern
  - Template Method Pattern

OO Design Exercise:

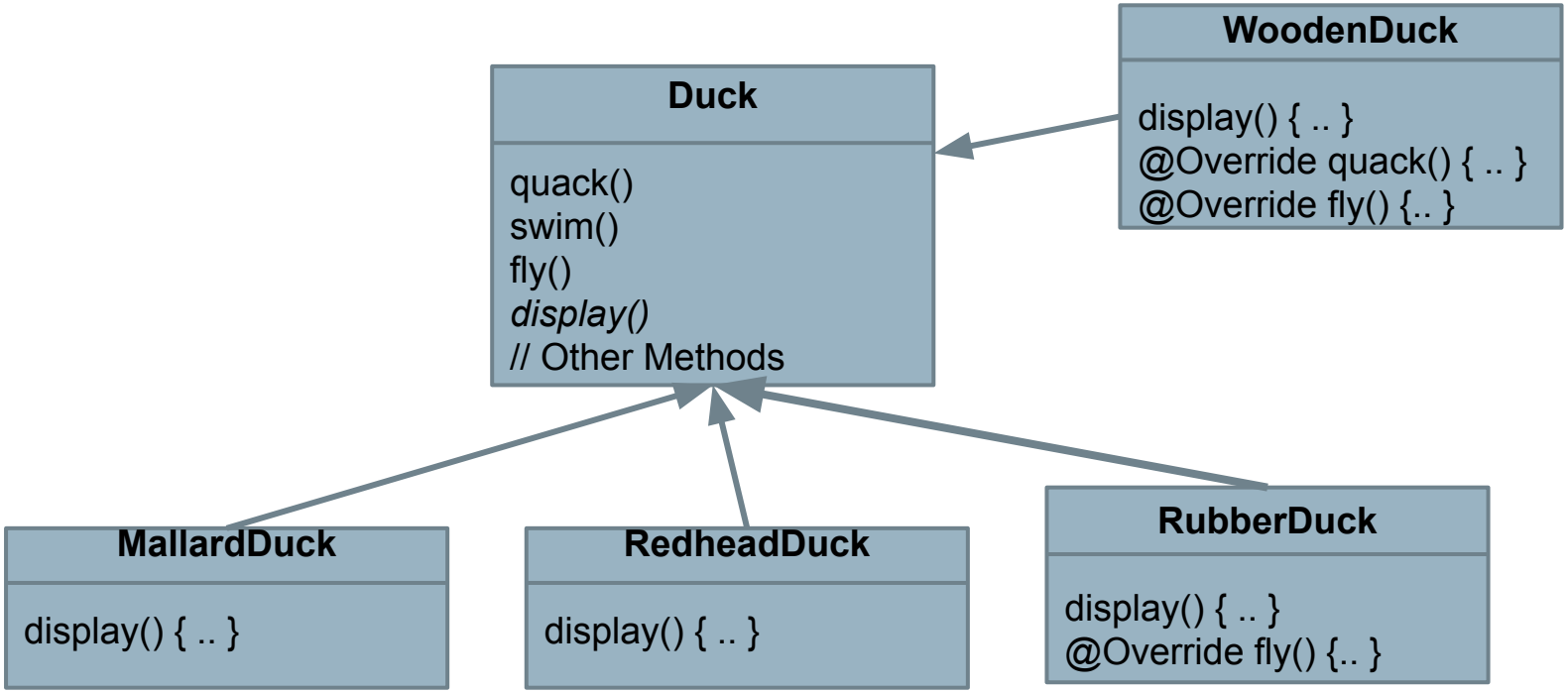
Building a Better Duck



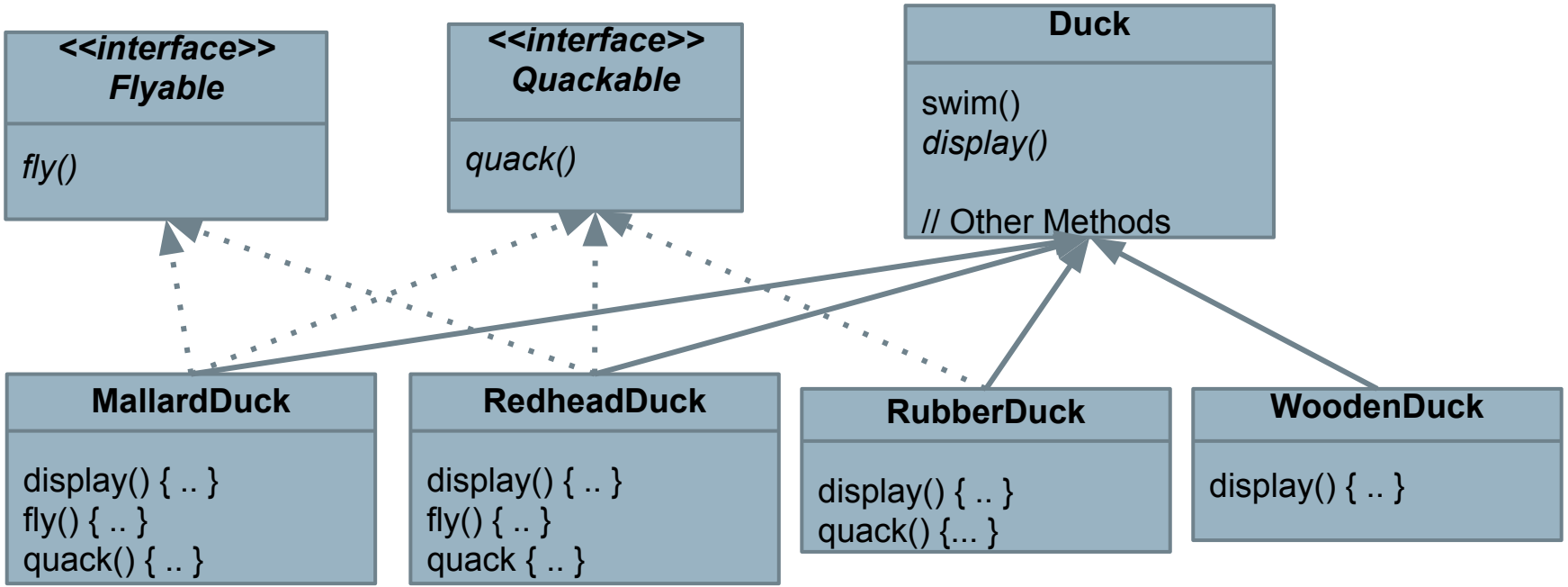
Adding new ducks



Why not override?



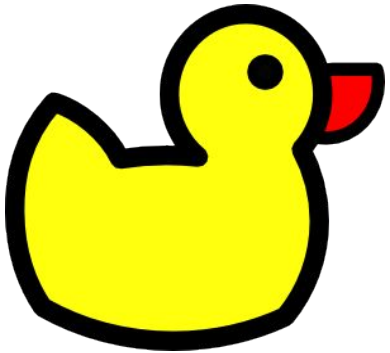
Why not interfaces?



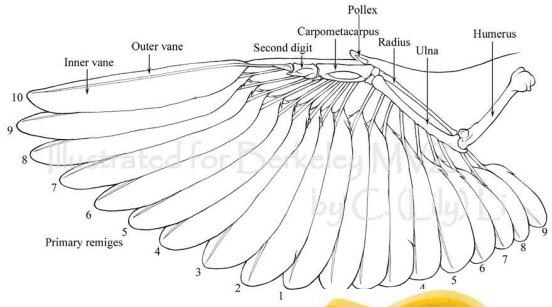
How do we fix this mess?

Apply good OO design principles!

Step 1: Identify the aspects that vary and encapsulate them.

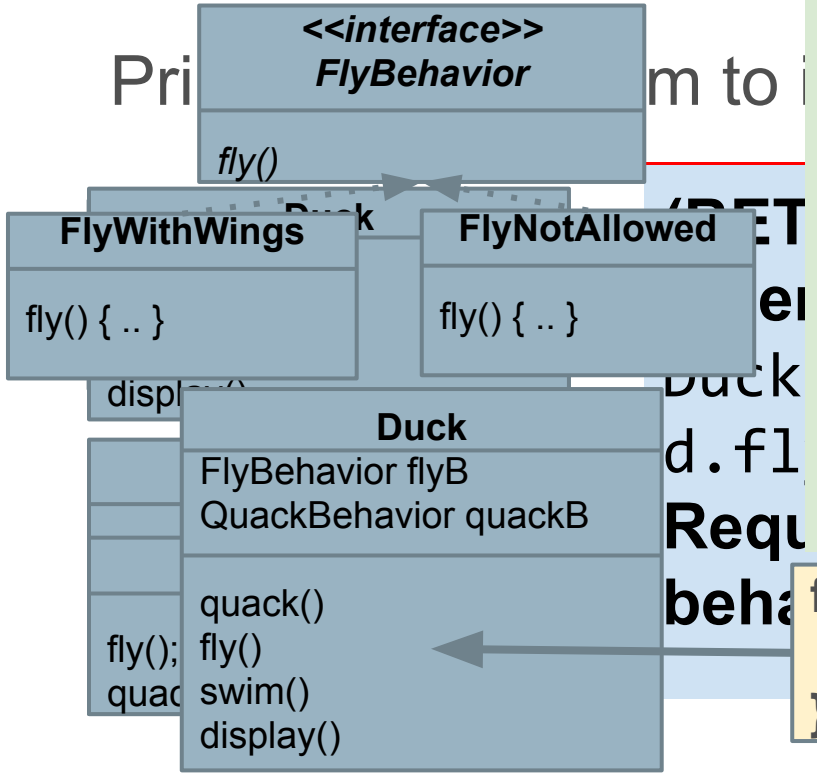


Duck class



Flying behaviors

Step 2: Implement behaviors as classes



(GOOD)
Programming to an interface:

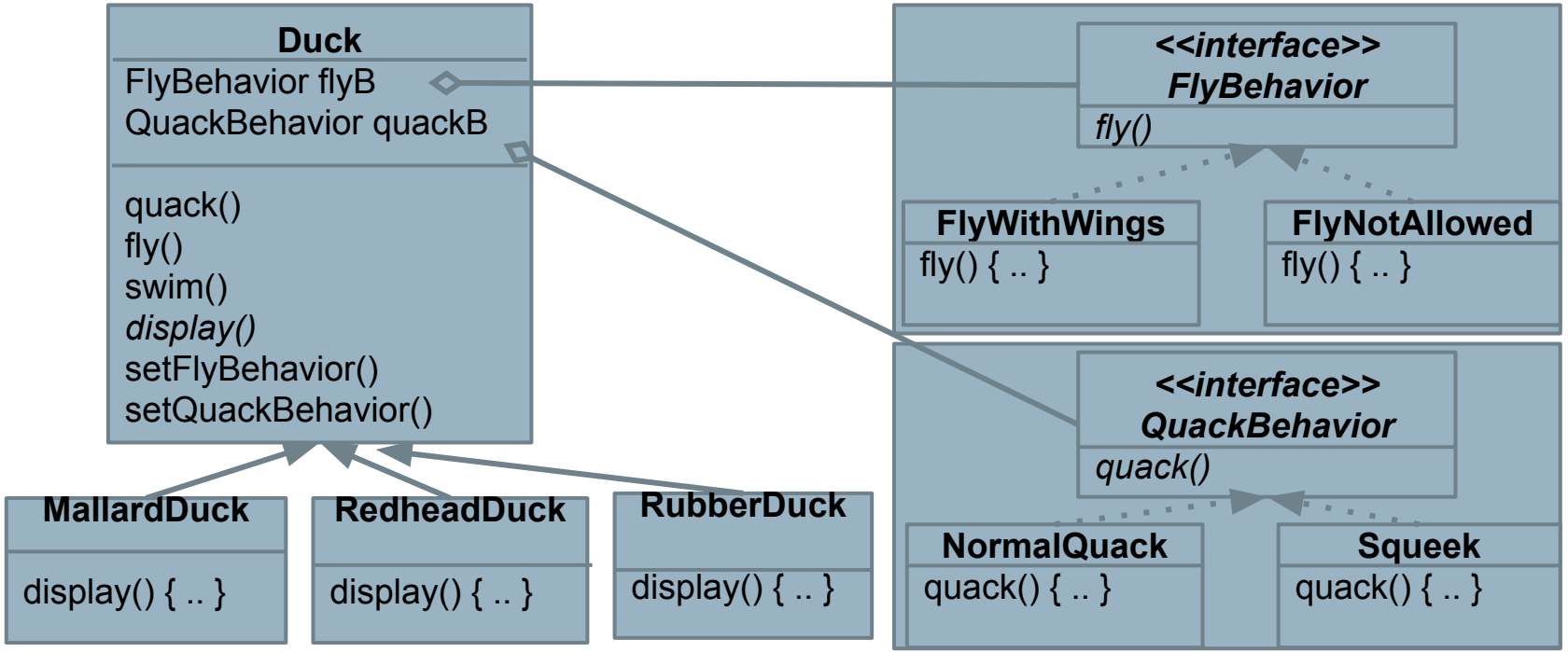
```
Duck d = new MallardDuck();
d.performFly();
```

Behavior called in same way for all ducks. Only implement behavior once.

```
fly() {
      flyB.fly();
}
```

HAS-A can be better than IS-A

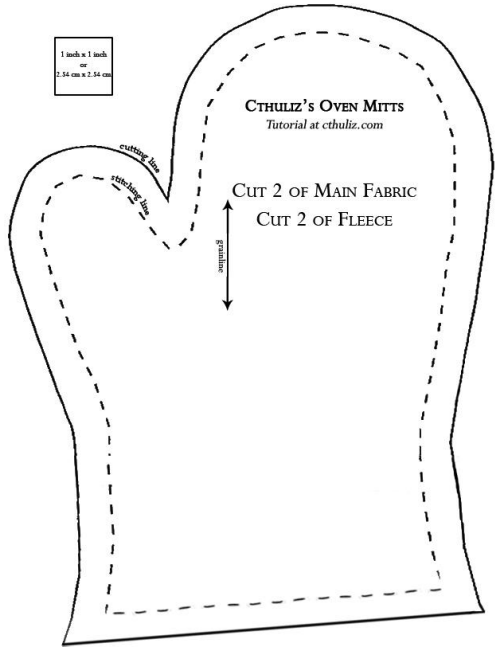
Principle: Favor composition over inheritance.



Enter... Design patterns

Don’t just describe *classes*, describe ***problems***.

Patterns prescribe design guidelines for common problem types.



Guidelines, not solutions

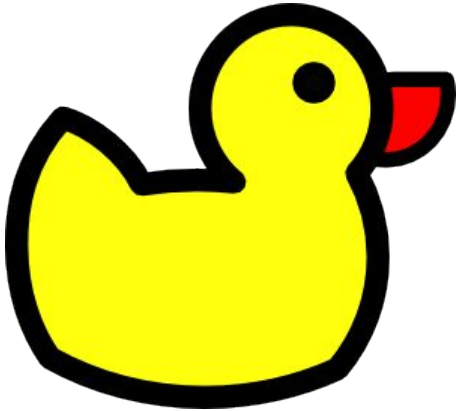
“Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem in such a way that you can use this solution a million times over, without ever doing it the same way twice.”

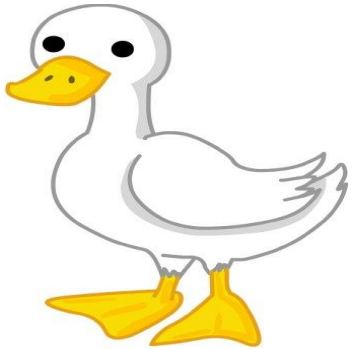
- Christopher Alexander

You already applied one pattern

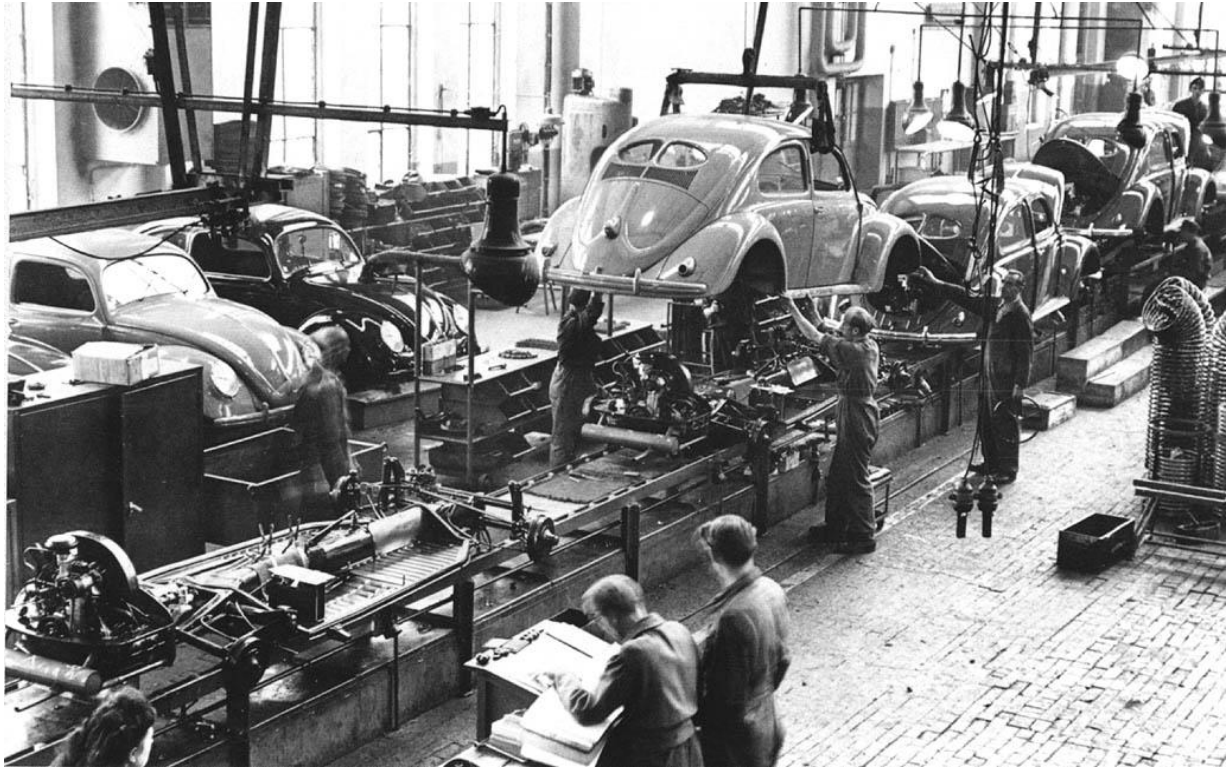
Strategy Pattern

Defines family of algorithms, encapsulates them, makes them interchangeable.





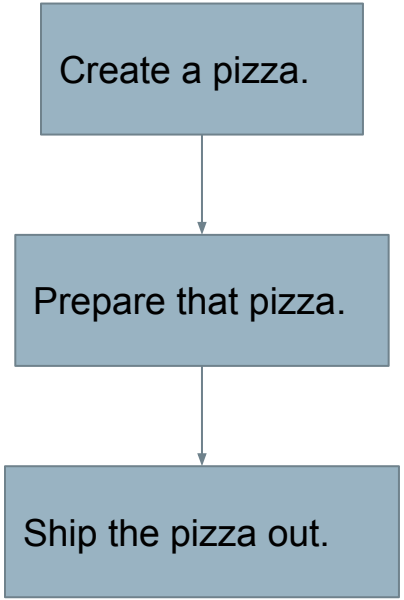
Factory Pattern - Motivation



Factory Pattern - Motivation

```
Pizza orderPizza(){
    Pizza pizza = new Pizza();

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



First Try

```
Pizza orderPizza(String type){
    Pizza pizza;
    if (type.equals(“cheese”)){
        pizza = new CheesePizza();
    else if(type.equals(“pepperoni”)){
        pizza = new PepperoniPizza();
    }
    // Prep methods
}
```

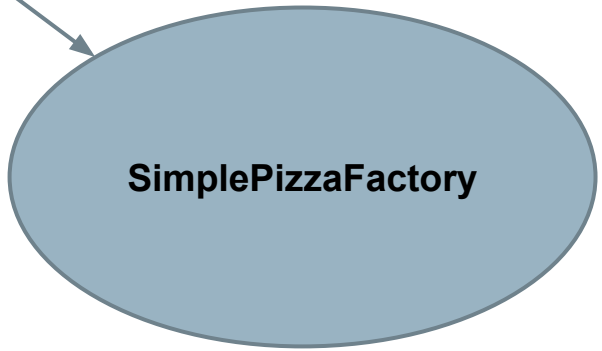
Factory Pattern - Motivation

```
Pizza orderPizza(String type){
     Pizza pizza;
     if (type.equals(“cheese”)){
          pizza = new CheesePizza();
     else if(type.equals(“pepperoni”)){
          pizza = new PepperoniPizza();
     } else if(type.equals(“kebab”)){
          pizza = new KebabPizza();
     }
     // Prep methods
}
```

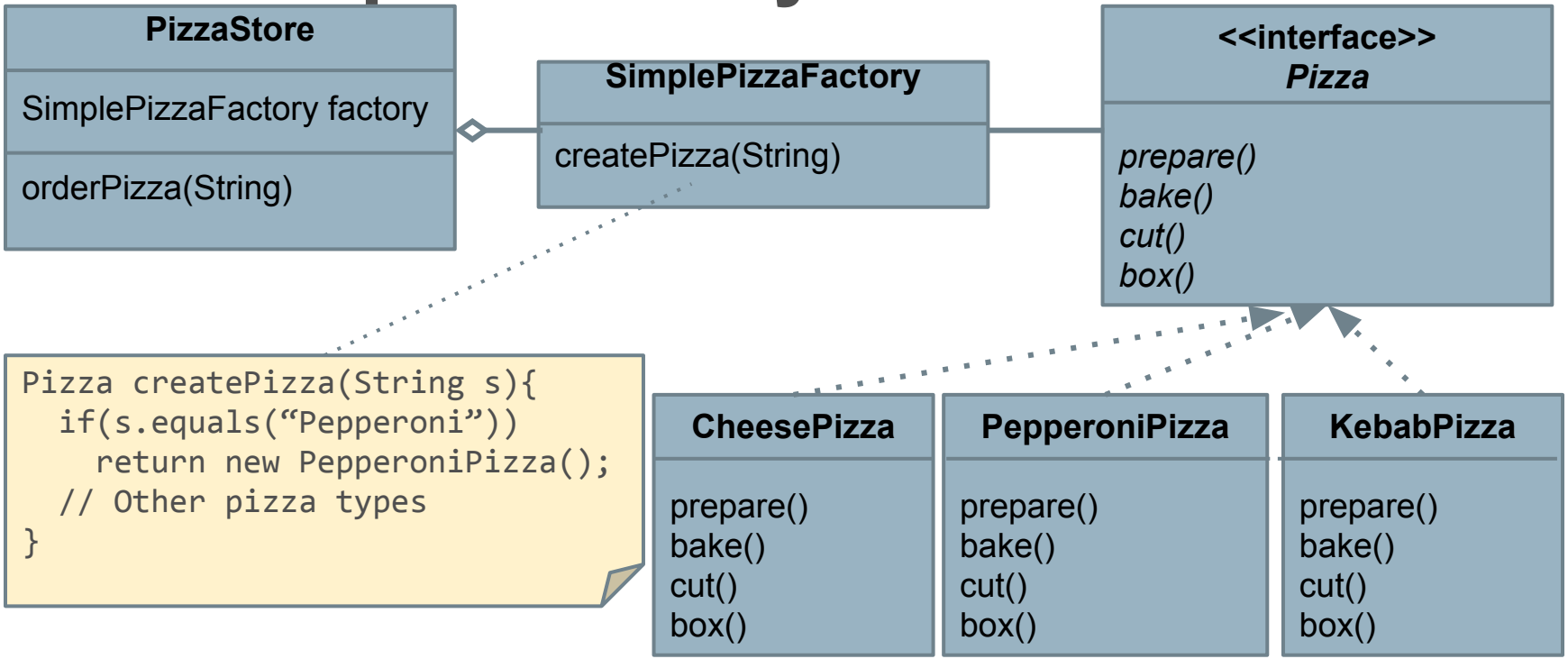
Factory Pattern - Motivation

```
Pizza orderPizza(String type){
    Pizza pizza;

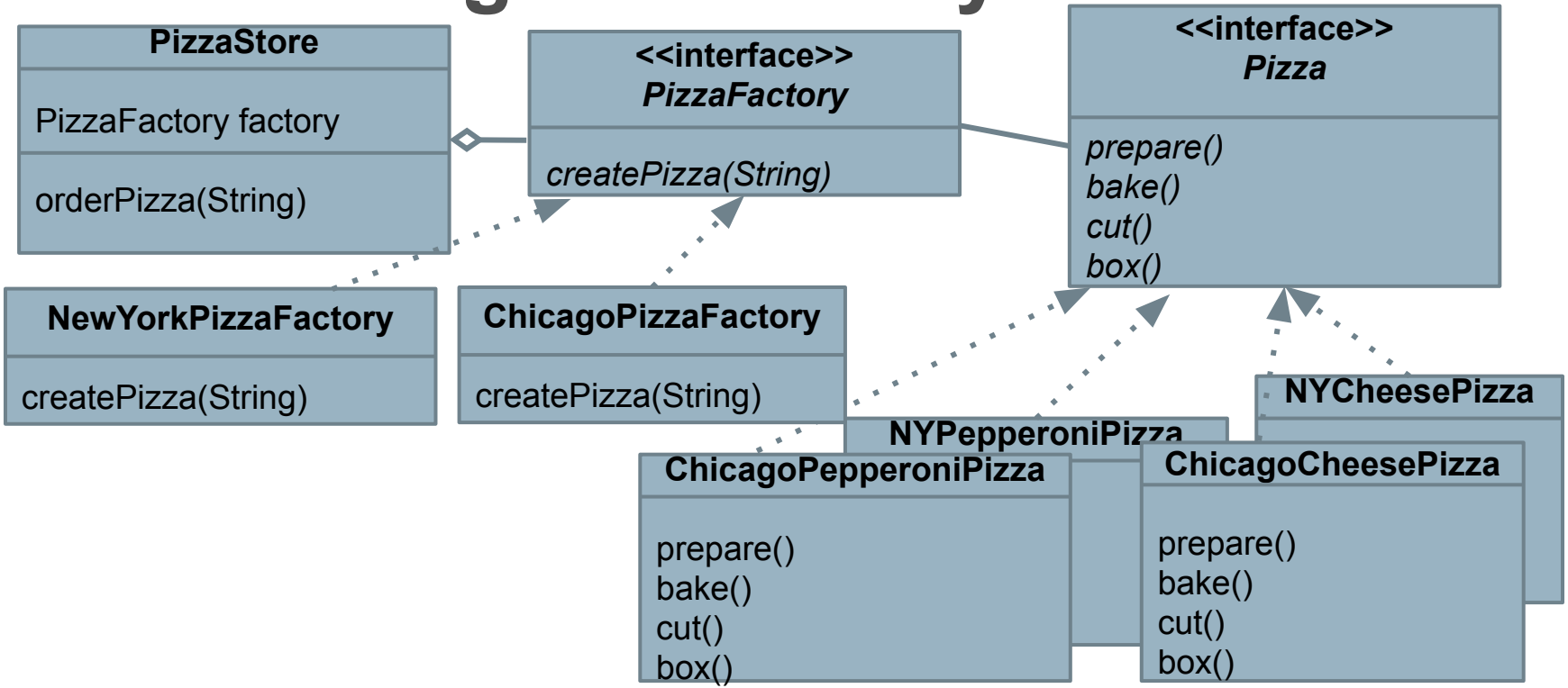
    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();
    return pizza;
}
```



The Simple Factory

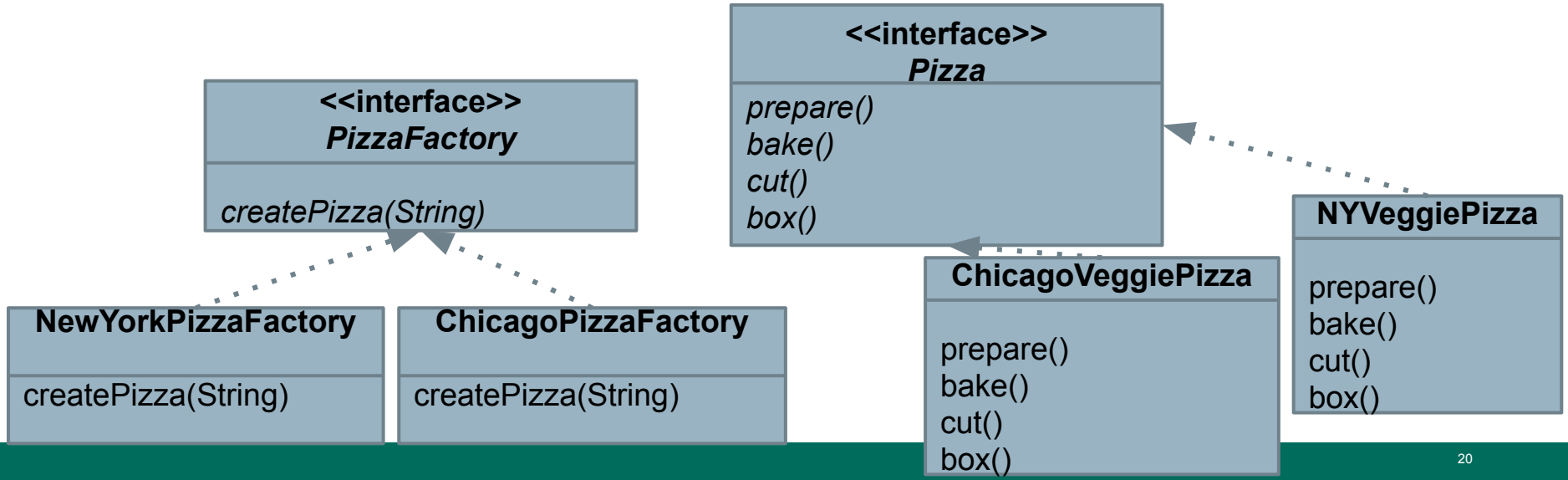


Franchising the Factory



Factory Pattern - Definition

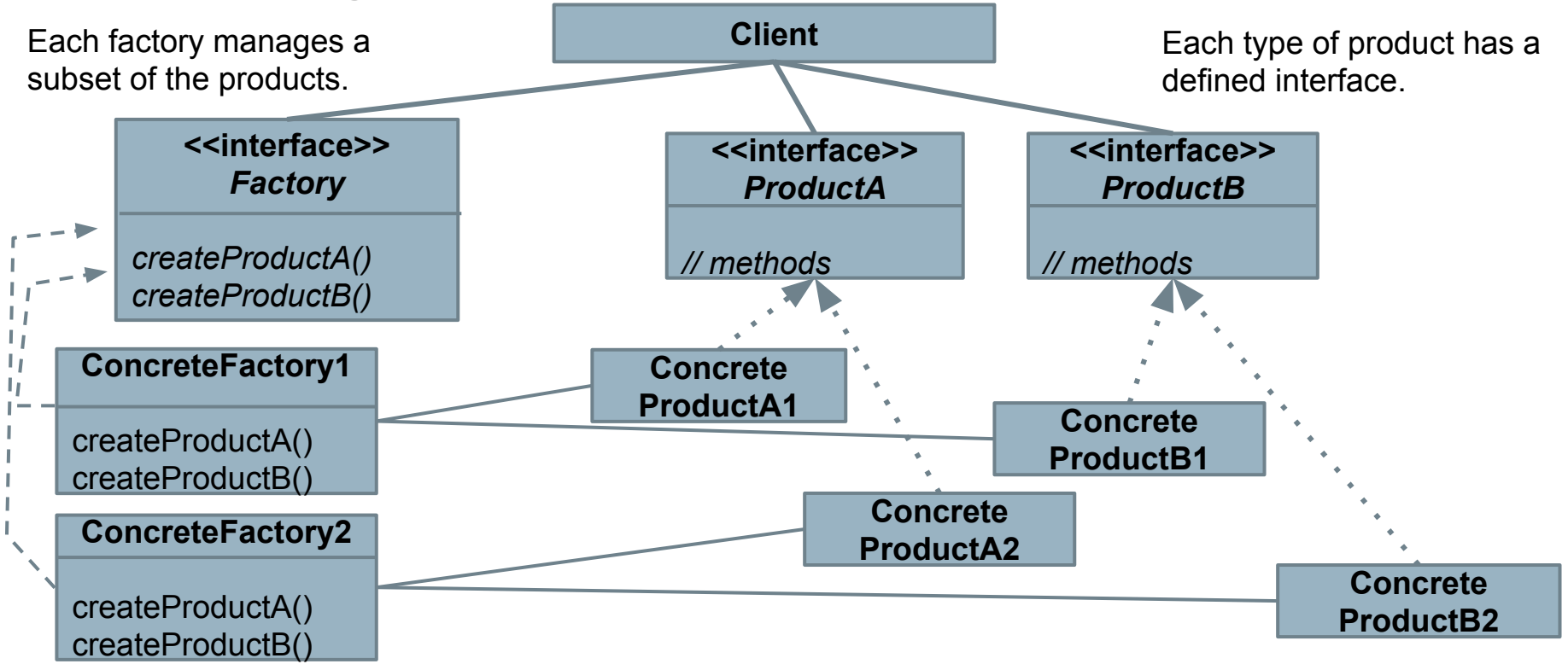
Defines interface for creating an object, lets subclasses decide which object to instantiate. Allows reasoning about **creators** and **products**.



Factory Pattern - In Practice

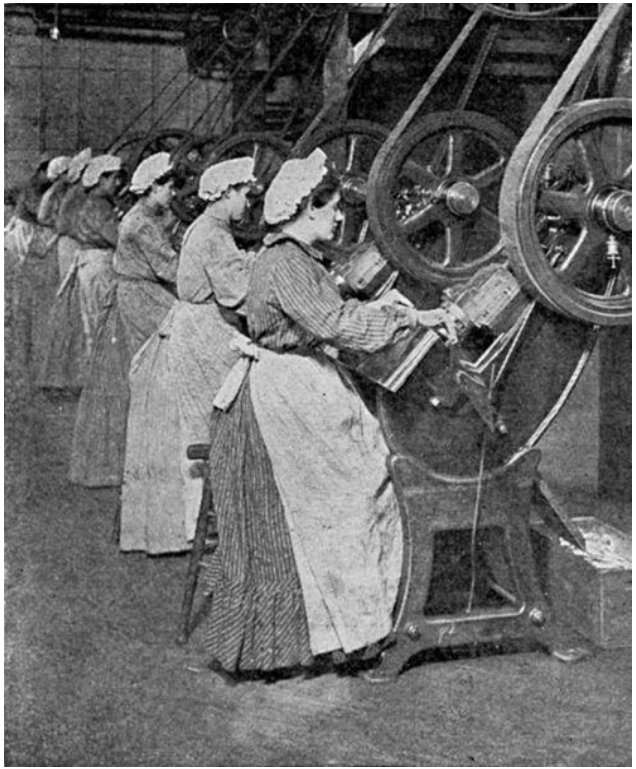
Each factory manages a subset of the products.

Each type of product has a defined interface.



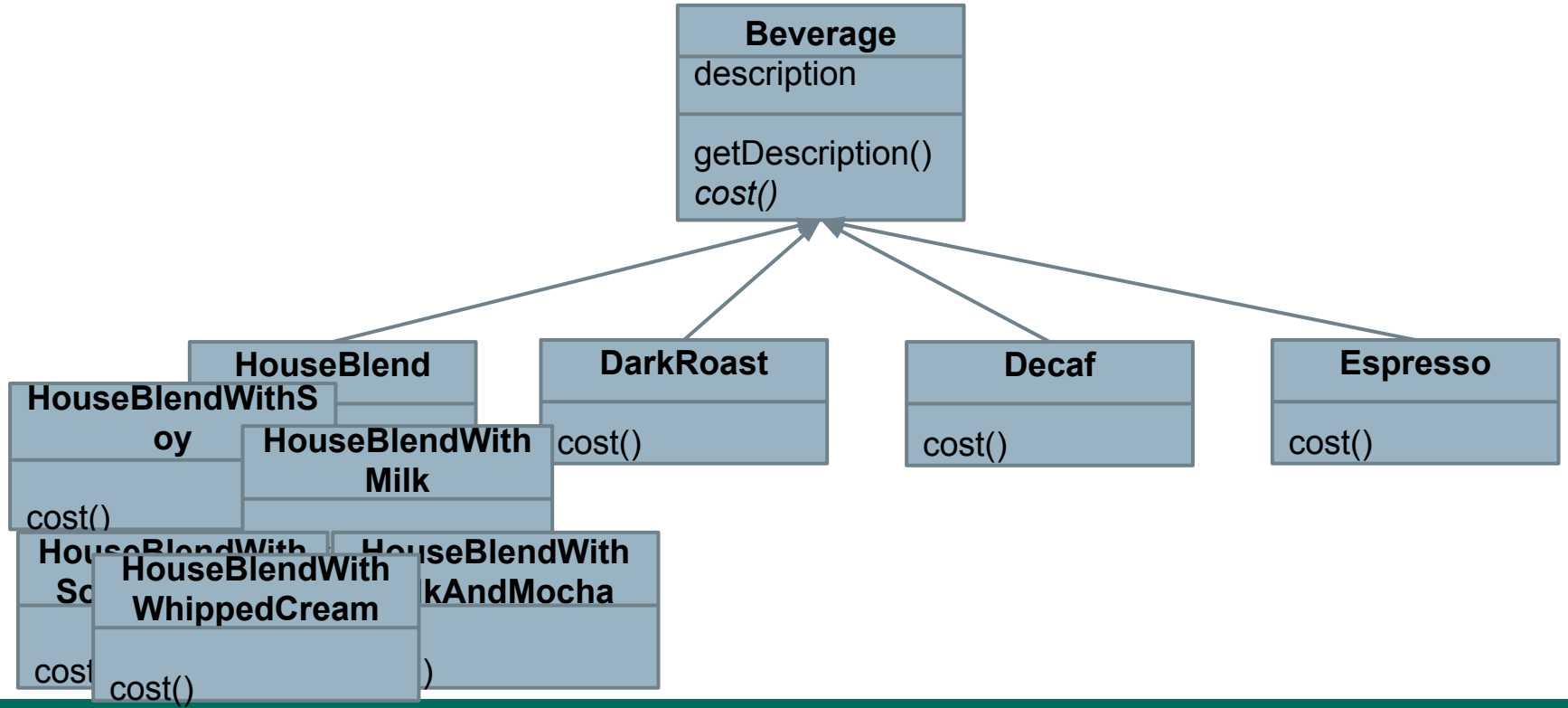
Benefits of Factory Pattern

1. Loose coupling.
2. Creation code is centralized.
3. Easy to add new products.
4. Lowered class dependency (depend on abstractions, not concrete classes).

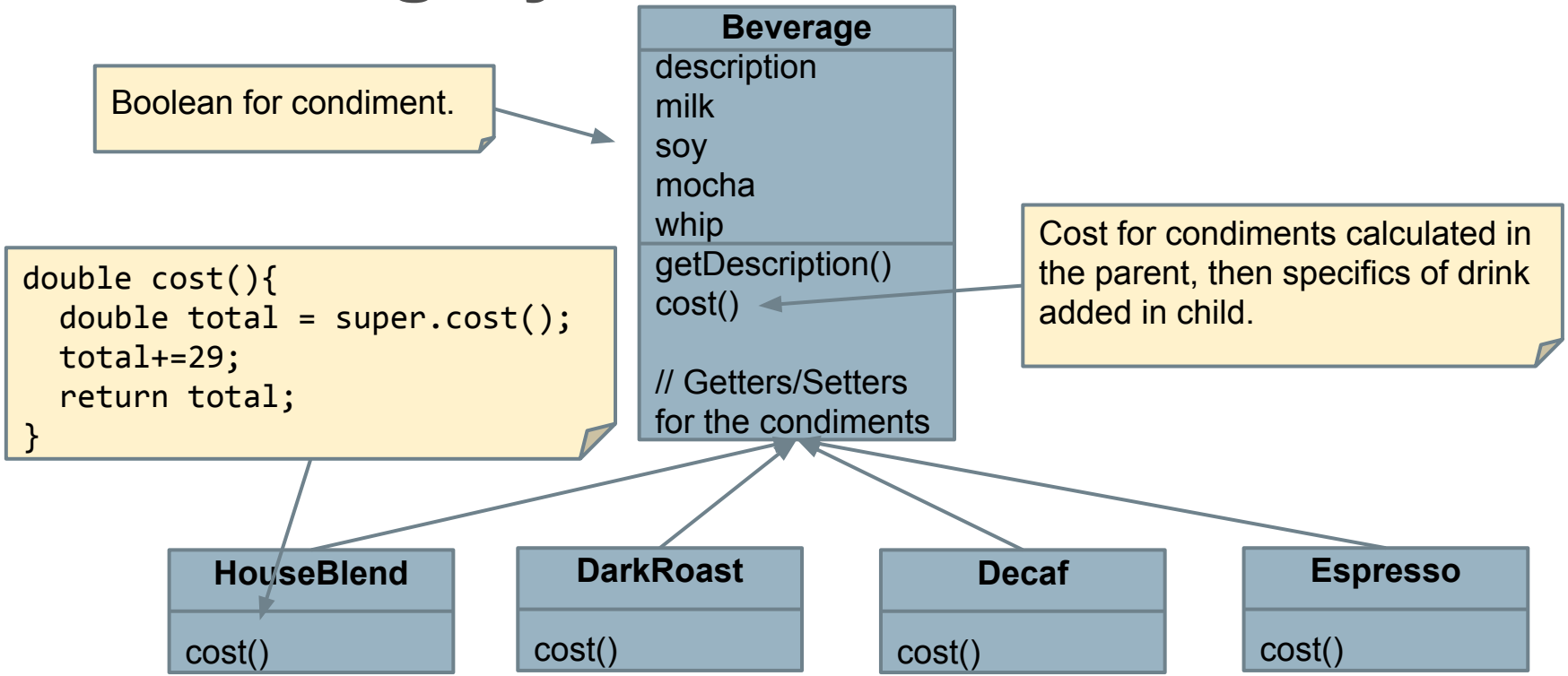


Let’s take a break!

The Coffee Shop Ordering System



Ordering System - Take 2



How Code Reuse is Achieved

- Inheritance allows writing code once and reusing in the children.
  - Good - changes only made once (in theory).
  - Bad - maintenance issues and inflexible design.
    - Inherit all behaviors of the parent. Might have to work around inherited features in child.
- Code can also be reused through composition.

Composition

- “Attach” an object to another object to add behaviors and attributes.
  - All Ducks have some form of flying behavior.
  - Implement behavior as a class, attach at object creation.
- Behavior extension done at runtime.
  - Dynamically change abilities of objects as system runs.
- Change a class without changing code of the class.

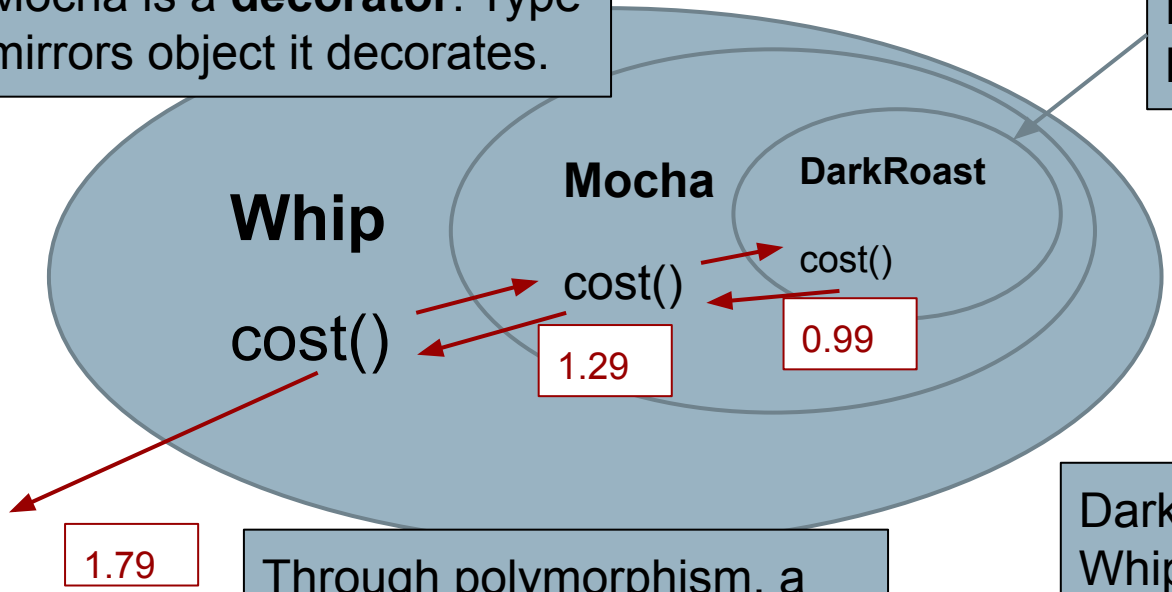
The Open-Closed Principle

- **Classes should be open for extension, closed for modification.**
  - Add new behavior without changing existing code.
  - Create class with new data and operators, attach class it is intended to extend.
  - Allow extension without direct modification.
- **Do not try to apply this everywhere**.
  - Focus on areas likely to change.

The Decorator Pattern

Mocha is a **decorator**. Type mirrors object it decorates.

DarkRoast inherits from Beverage, has cost() method.



Whip is a **decorator**. Type mirrors object it decorates (and anything that object decorates).

Through polymorphism, a Beverage wrapped in Mocha is still a Beverage.

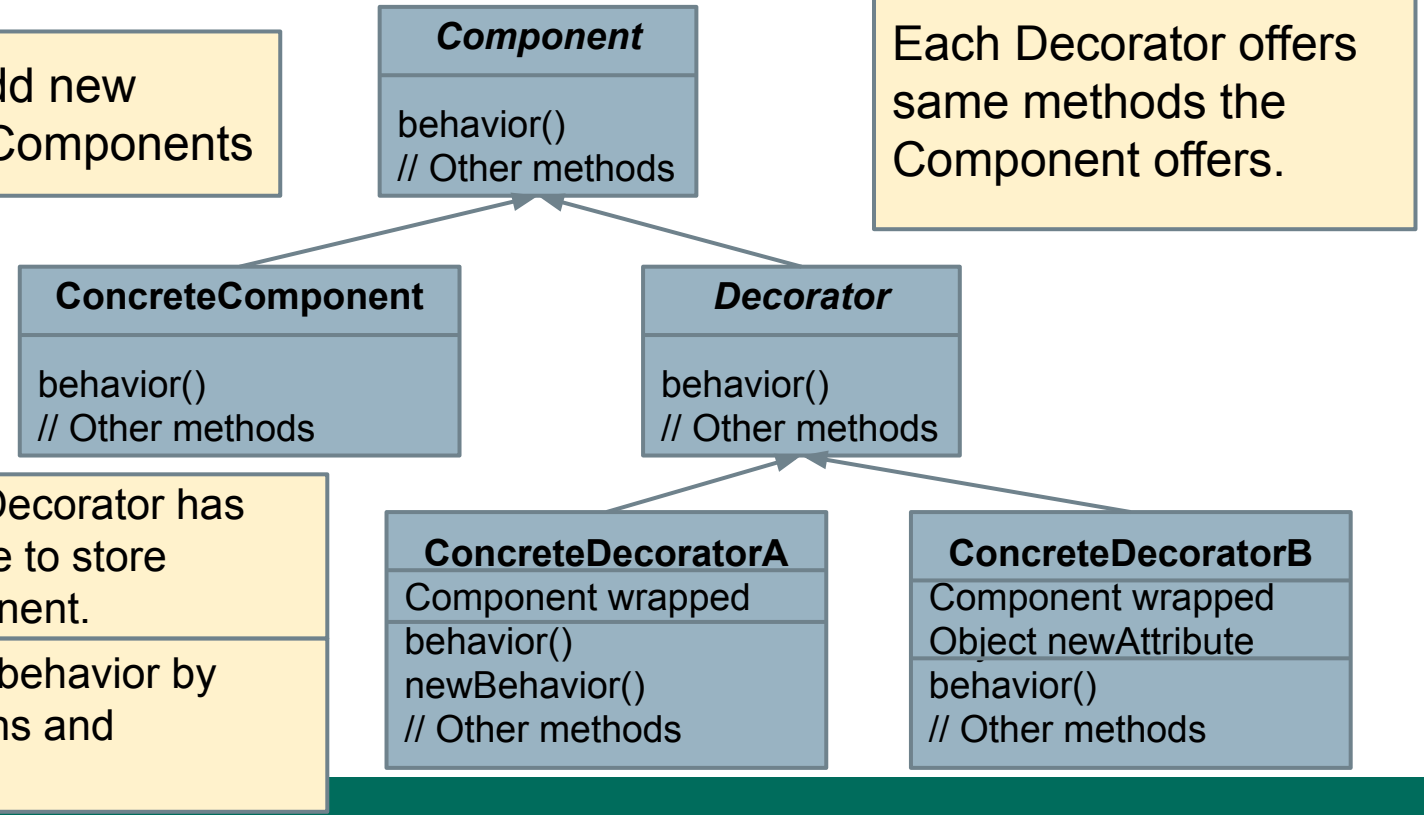
DarkRoast wrapped in Mocha and Whip is a Beverage, and can perform any Beverage function.

The Decorator Pattern Defined

- Attaches responsibilities to an object dynamically.
- Flexible alternative to subclassing.
  - Decorators have same supertype as decorated object.
  - One or more decorators can wrap an object.
  - Can pass decorated object in place of the original.
  - Decorator adds its own behavior before or after calling wrapped object.

The Decorator Pattern

Decorators add new behaviors to Components

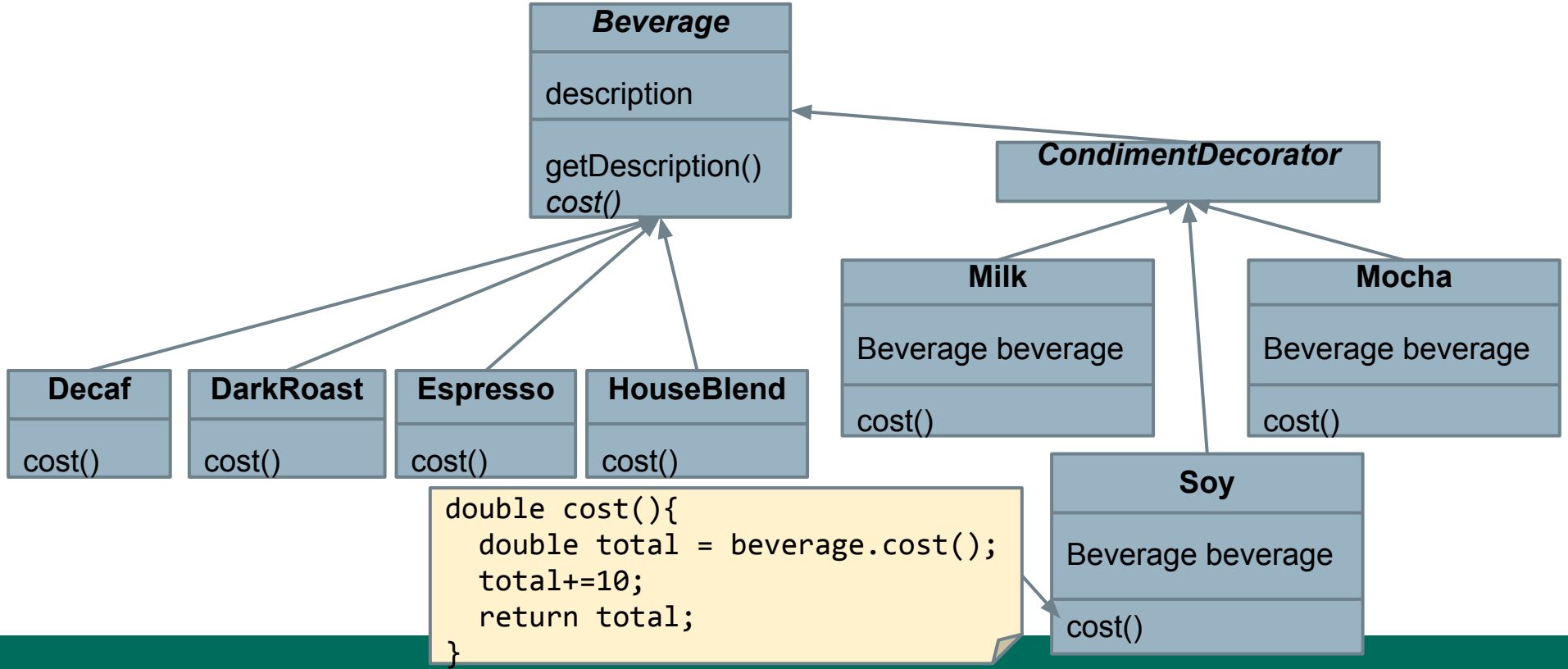


Each Decorator offers same methods the Component offers.

Each concrete Decorator has instance variable to store wrapped component.

Decorators add behavior by adding operations and attributes.

Ordering System - Decorator Pattern



The Decorator Pattern

- Uses inheritance to achieve *type matching*, but not to inherit behavior.
- By composing decorator with a component, we add new behavior to component.
  - Adds flexibility to how we mix and match behaviors.
  - Can reassign decorators at runtime.
  - Can add new behavior by writing new decorator.

Decorator Pattern Negatives

- Often results in explosion of small classes.
  - Results in hard to understand design.
- Potential type issues.
  - If code does not need specific type, decorators can be used transparently.
    - Everything is a Beverage.
  - Problems if we need to know type.
    - DarkRoast gets a discount.

Working With Other Systems

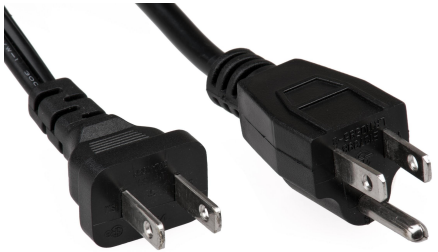
- We want to incorporate services or code from another system.
  - Their interface may be compatible with your interface.

Adapters

Your Existing System

Adapter

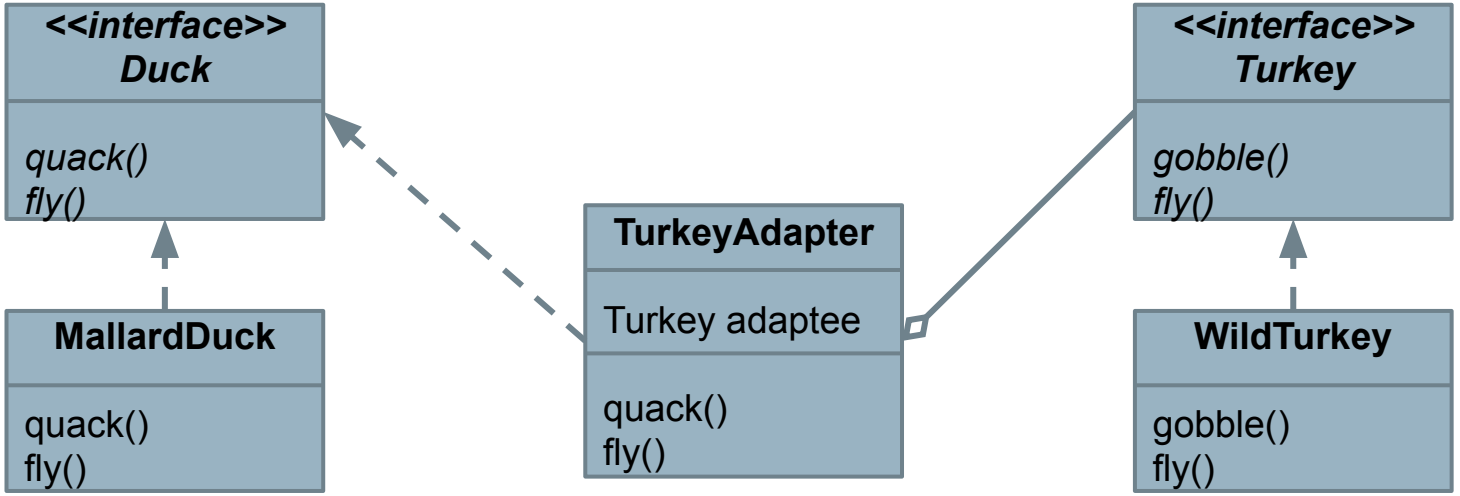
External System



Adapter implements the interface your class expects.

And talks to the external class to service requests.

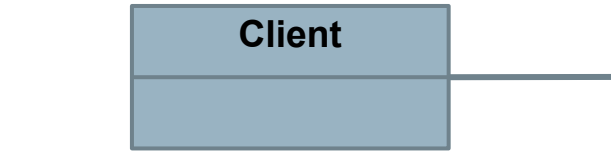
Adapter Example

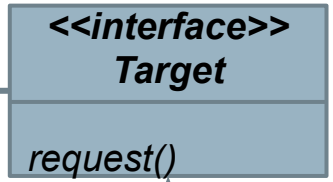


The Adapter Pattern Defined

- Converts an interface into interface client expects.
  - **Adapter**'s methods call corresponding methods from **adaptee**.
  - If adaptee changes, only the adapter needs to change.
    - No changes needed to classes that call adapter.

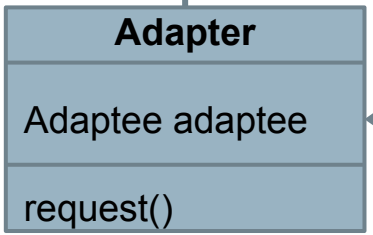
The Adapter Pattern

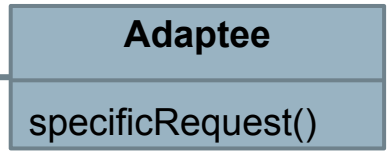




Adapter implements target interface.

The client sees only the target interface.





Adapter composed with Adaptee.

Requests get delegated to Adaptee.

Let’s take a break!

Watching a Movie

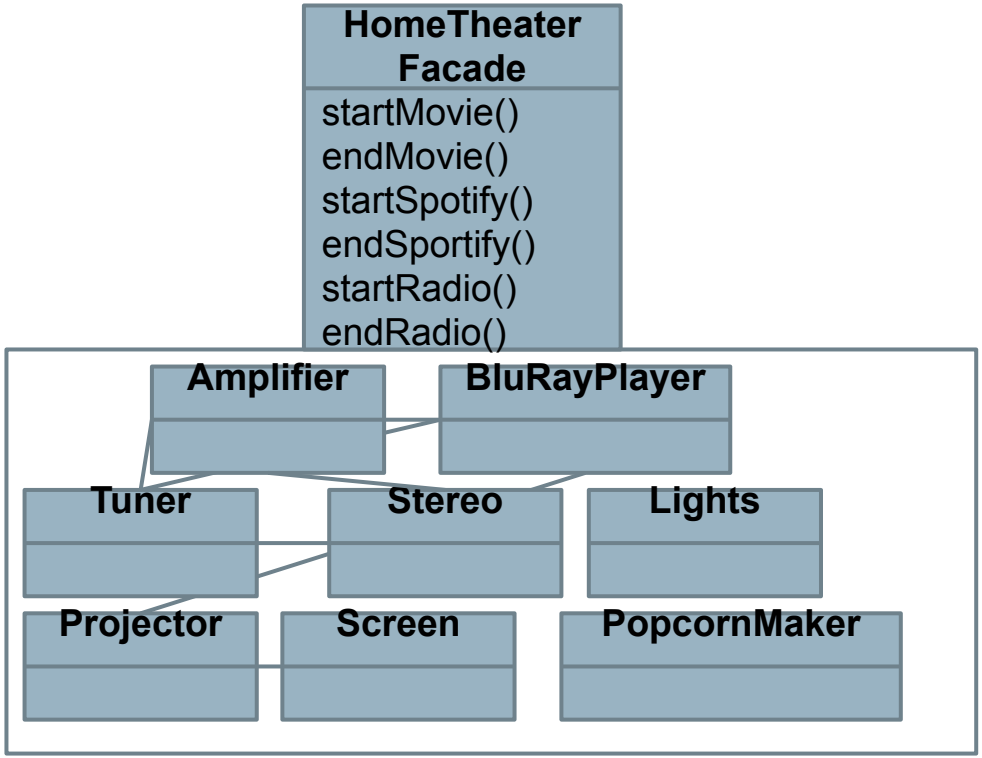
To watch a movie, we need to perform a few tasks:

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

Wrapping Classes

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

The Facade Pattern

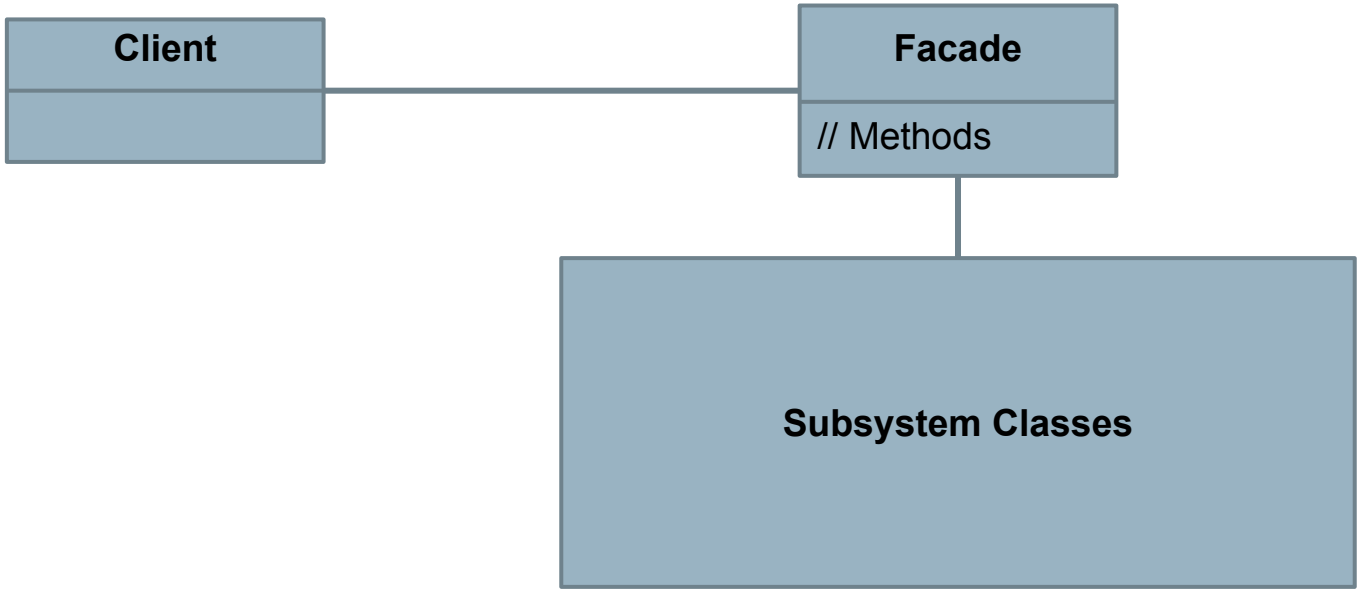


- Create a new class that exposes simple methods (the **facade**).
- Facade calls on classes to implement high-level methods.
- Client calls facade instead of classes.
- Classes still accessible.

The Facade Pattern Defined

- Provides a unified interface to a set of classes.
- Facade defines a high-level interface that makes a subsystem easier to use.
  - Provides an additional method of access.
  - Multiple facades may provide situational functions.
  - Decouples client from any one subsystem.

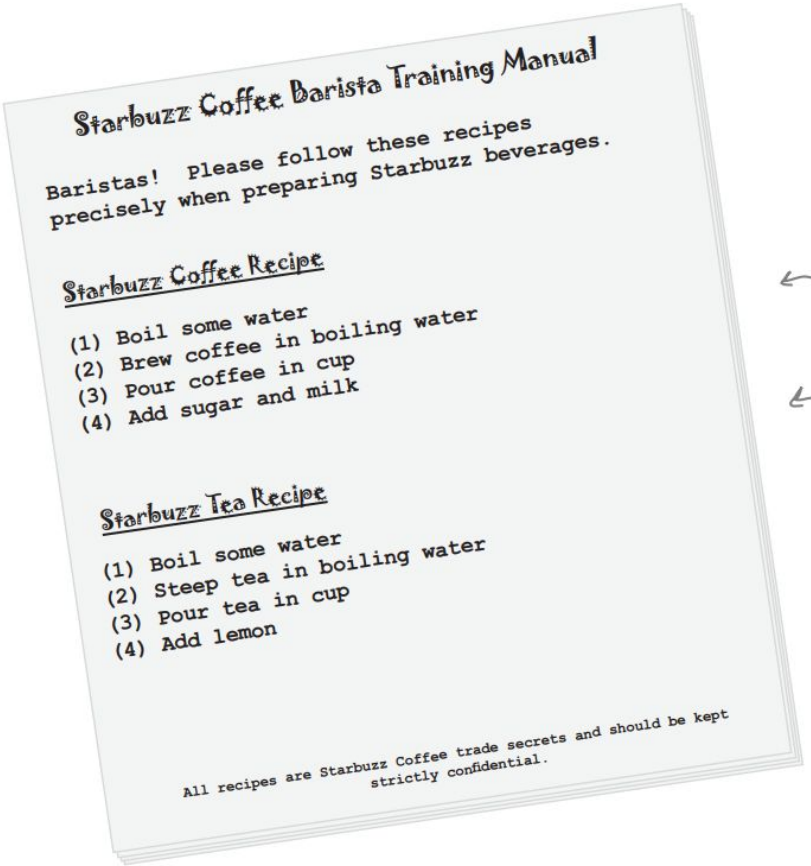
The Facade Pattern



The Principle of Least Knowledge

- **Talk only to your immediate friends**.
- Be careful of the number of classes your class interacts with and how it interacts with them.
- Only invoke methods that belong to the object, objects passed as parameters, objects created or instantiated, and attached objects.

Coffee and Tea



Starbuzz Coffee Barista Training Manual

Baristas! Please follow these recipes precisely when preparing Starbuzz beverages.

Starbuzz Coffee Recipe

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

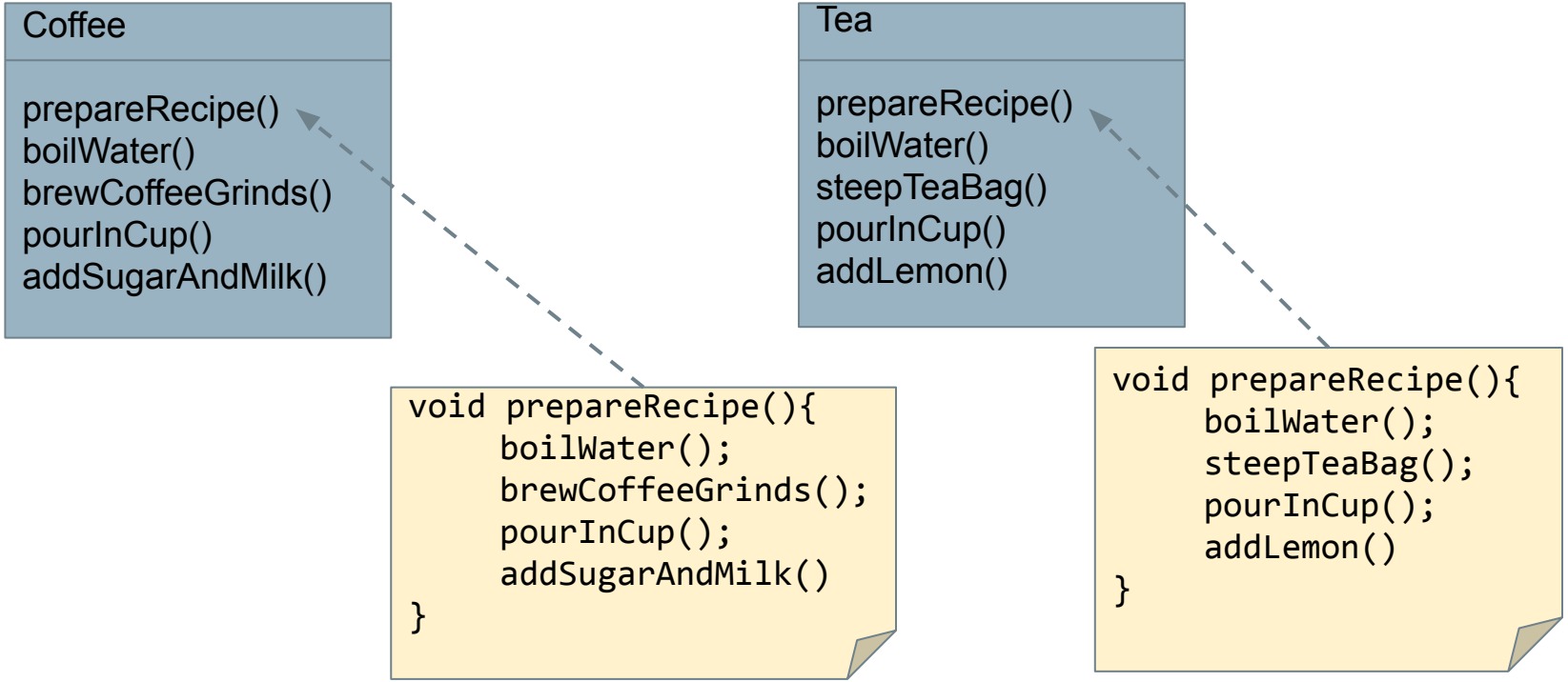
Starbuzz Tea Recipe

(1) Boil some water
(2) Steep tea in boiling water
(3) Pour tea in cup
(4) Add lemon

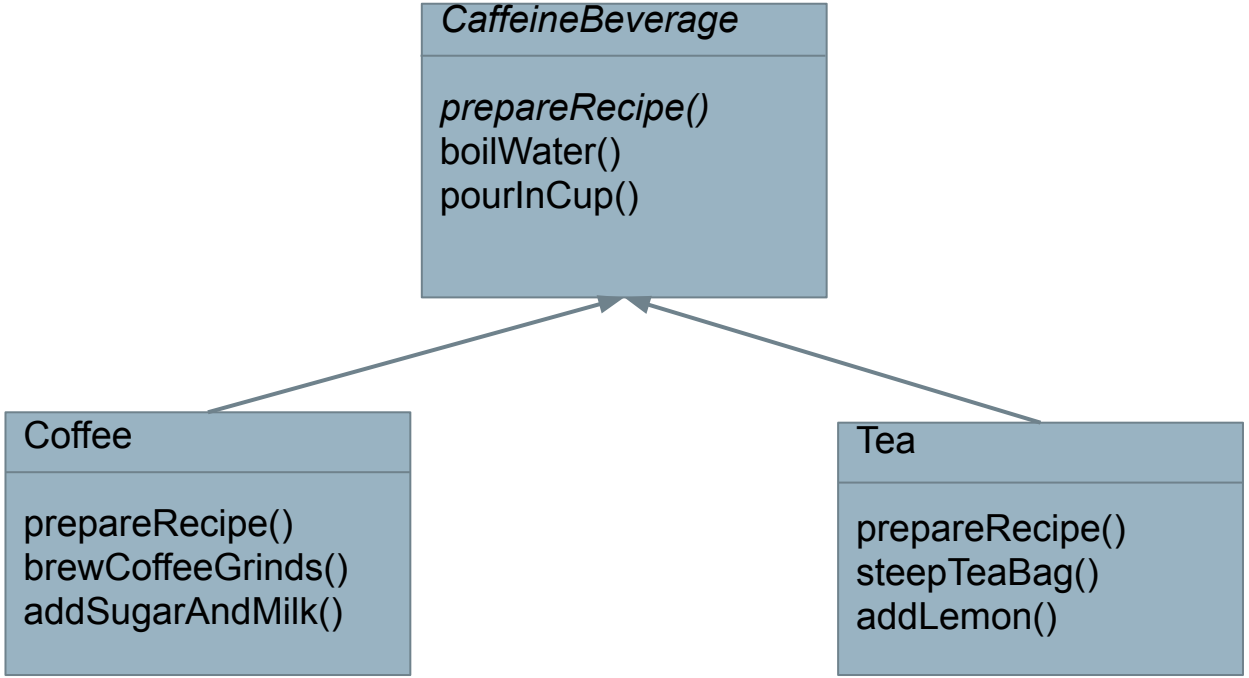
All recipes are Starbuzz Coffee trade secrets and should be kept strictly confidential.

The recipe for coffee looks a lot like the recipe for tea, doesn't it?

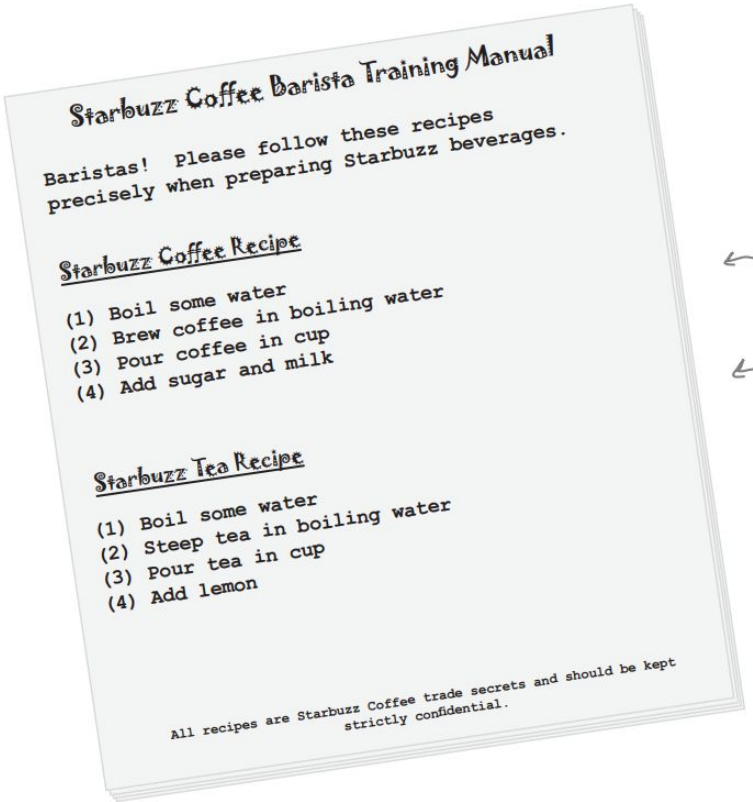
Coffee and Tea (In Code)



Coffee and Tea (In Code) - Take 2



Back to the Recipes



Starbuzz Coffee Barista Training Manual

Baristas! Please follow these recipes precisely when preparing Starbuzz beverages.

Starbuzz Coffee Recipe

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

Starbuzz Tea Recipe

(1) Boil some water
(2) Steep tea in boiling water
(3) Pour tea in cup
(4) Add lemon

All recipes are Starbuzz Coffee trade secrets and should be kept strictly confidential.

The recipe for coffee looks a lot like the recipe for tea, doesn't it?

Algorithm

1) Boil some water.
2) Use hot water to extract the beverage from a solid form.
3) Pour the beverage into a cup.
4) Add appropriate condiments to the beverage.

- Steps 1 and 3 already abstracted into base class.
- Steps 2 and 4 not abstracted, but are the same concept applied to different beverages.

Abstracting `prepareRecipe()`

- Coffee: `brewCoffeeGrinds()`, `addSugarAndMilk()`
- Tea: `steepTeaBag()`, `addLemon()`.
  - Steeping and brewing aren't all that different (`brew()`).
  - Adding sugar is like adding lemon (`addCondiments()`).
- ```
  void prepareRecipe() {
      boilWater();
      brew();
      pourInCup();
      addCondiments();
  }
  ```
  ```

Our Redesigned Code

CaffeineBeverage is abstract, just like in the class design.

```
public abstract class CaffeineBeverage {

    final void prepareRecipe() {
        boilWater();
        brew();
        pourInCup();
        addCondiments();
    }

    abstract void brew();

    abstract void addCondiments();

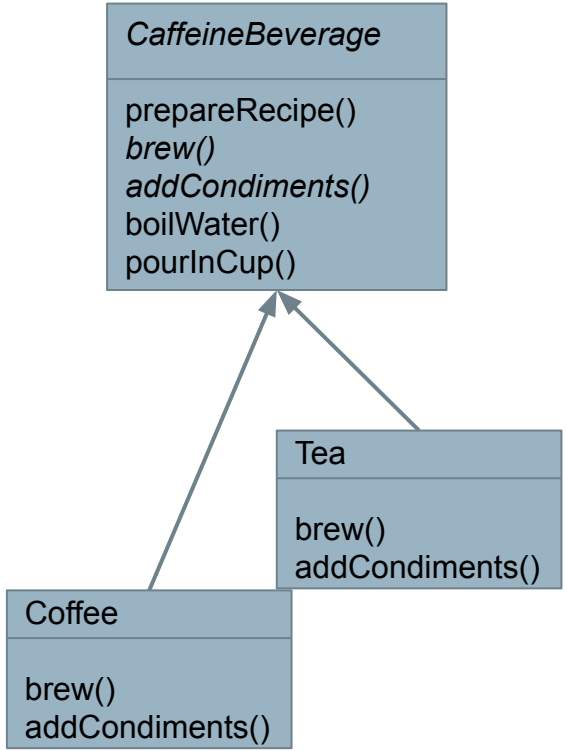
    void boilWater() {
        System.out.println("Boiling water");
    }

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

Now, the same prepareRecipe() method will be used to make both Tea and Coffee. prepareRecipe() is declared final because we don't want our subclasses to be able to override this method and change the recipe! We've generalized steps 2 and 4 to brew() the beverage and addCondiments().

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!

Remember, we moved these into the CaffeineBeverage class (back in our class diagram).



What Have We Done?

- Two recipes are the same, although some steps require *different implementations*.
- Generalized recipe into a base class.
  - CaffeineBeverage controls the steps of the recipe. It performs common steps itself.
    - *(encapsulating what does not change...)*
  - It relies on subclasses to implement unique steps.
    - *(... from what does change)*

The Template Method Pattern

- `prepareRecipe()` is our **template method**.
  - Serves as a template for an algorithm.
- Each step is represented by a method.
- Some methods handled by the base class, others by the subclasses.
  - The supplied methods are declared abstract.

What Does the Template Method Get Us?

Original Implementation

- Coffee and Tea control algorithm.
- Code duplicated in Coffee and Tea.
- Changes to algorithm require changes to the subclasses.
- Classes are organized in a structure that requires more work to add a new beverage.
- Knowledge of algorithm and how to implement it distributed over multiple classes.

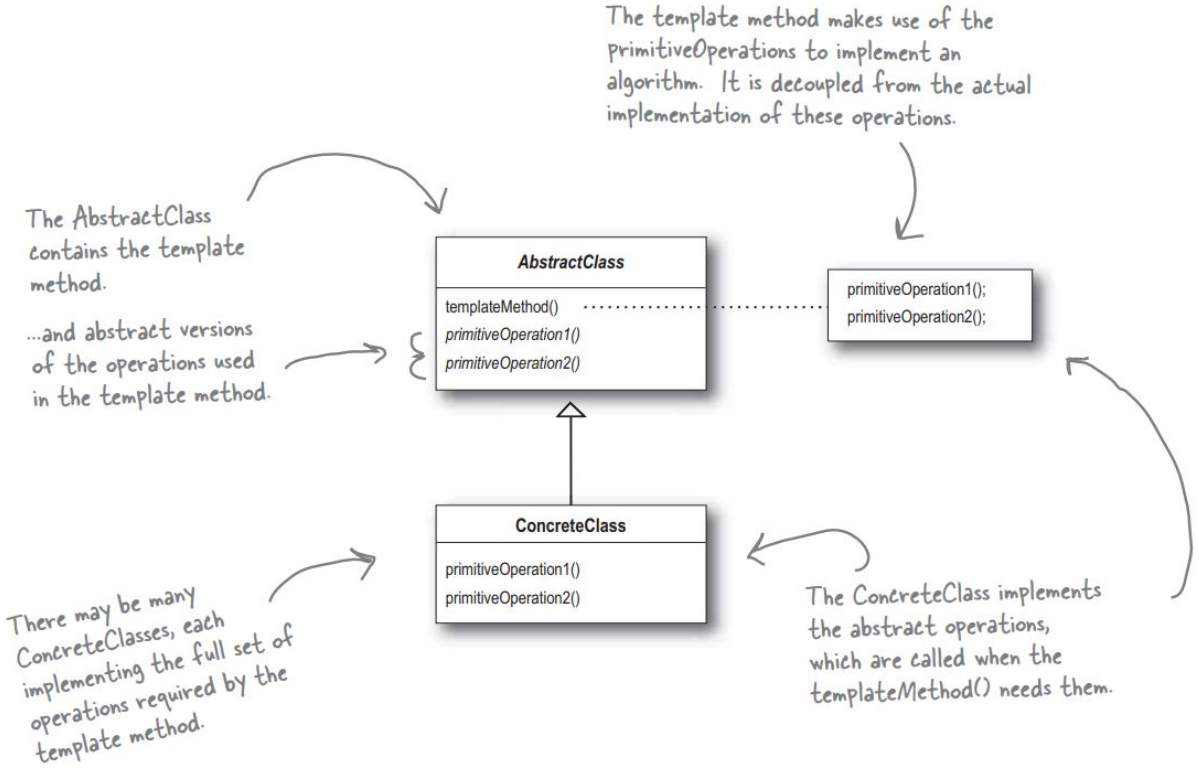
Template Method:

- CaffeineBeverage controls and protects the algorithm, implements common code.
- Algorithm lives in one place and changes only made there.
- Template Method allows new beverages to be added. They implement specialized methods.
- CaffeineBeverage class contains all knowledge about algorithm, relies on subclasses to provide implementations.

The Template Method Pattern

- Defines skeleton of an algorithm in a method, deferring some steps to subclasses.
- Lets subclasses redefine steps of an algorithm without changing the algorithm's structure.
- A template defines an algorithm as a set of steps.
  - Abstract steps are implemented by subclasses.
  - Ensures algorithm's structure stays unchanged.

Template Method Pattern



Looking Inside the Code


```
abstract class AbstractClass {

    final void templateMethod() {
        primitiveOperation1();
        primitiveOperation2();
        concreteOperation();
    }

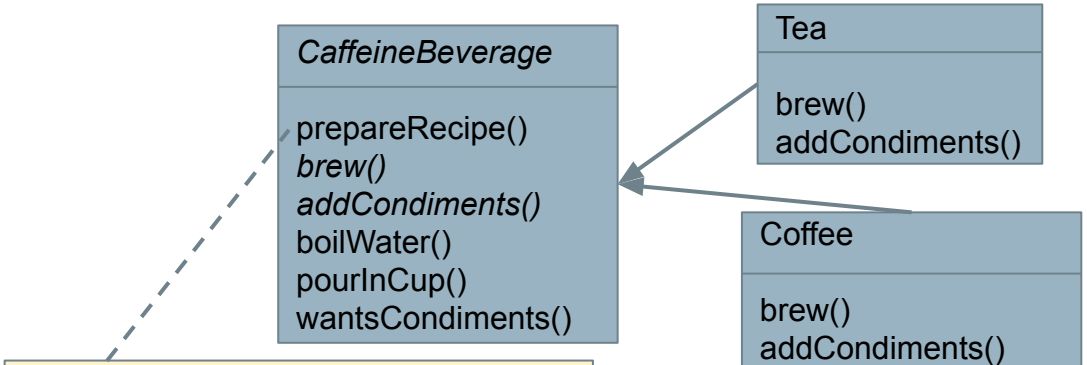
    abstract void primitiveOperation1();

    abstract void primitiveOperation2();

    void concreteOperation() {
        // implementation here
    }
}
```

Adding Hooks

- Parent defines concrete default implementations (**hooks**).
  - Subclasses can override, but do not have to.
  - Gives subclasses ability to “hook into” the algorithm.




```
void prepareRecipe(){
  boilWater();
  brew()
  pourInCup();
  if(wantsCondiments()){
    addCondiments();
  }
}
boolean wantsCondiments(){
  return true;
}
```

```
boolean wantsCondiments(){
  String answer = getUserInput();
  if answer.equals(“yes”)
    return true;
  else
  return false;
}
```

The Hollywood Principle

- **Don’t call us, we’ll call you.**
- Prevents “dependency rot”.
  - High-level components depend on low-level components, low-level depend on high level.
- Allows low-level components to hook into a system, but high-level components decide when and how they are needed.

Principles of Design

1. Identify aspects that vary and encapsulate them away from what doesn't.
2. Program to interface rather than implementation.
3. Favor composition over inheritance.
4. Open for extension, but closed for modification.
5. Talk only to your immediate friends.
6. Don't call us, we'll call you.

Why not use a design pattern?

What are the drawbacks to using patterns?

- Potentially over-engineered solution.
- Increased system complexity.
- Design inefficiency.

How can we avoid these pitfalls?

We Have Learned

- Design patterns allow implementation and management of variability in code.
  - Strategy Pattern encapsulates interchangeable behaviors and uses delegation to decide which to use.
  - Factory Pattern encapsulates object creation so system doesn’t need to know what type of object was created.
  - Decorator Pattern wraps an object in another to provide new behavior without code changes.

We Have Learned

- Design patterns allow implementation and management of variability in code.
  - Adapter Pattern wraps object in a new interface.
  - Facade Pattern wraps a set of classes in simplified interfaces.
  - Template Method Pattern encapsulates pieces of algorithms so that subclasses can hook into a computation.

Next Time

- Modularity

- Assignment 2 due Sunday. Any questions?

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY