Design Patterns (Part 3)

CSCE 247 - Lecture 20 - 04/03/2019

Design Patterns

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

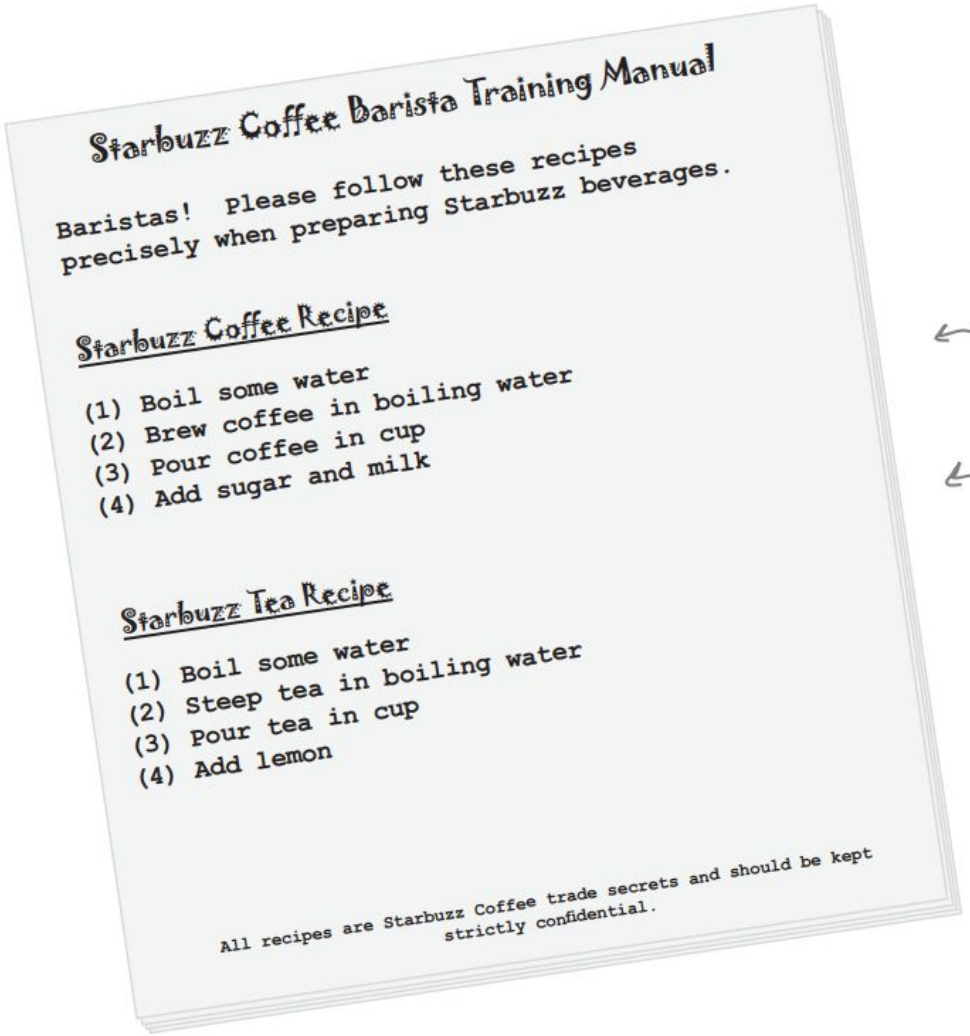
Design Patterns

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

Today

- **Template Method Pattern** encapsulates pieces of algorithms so that subclasses can hook into a computation.
- **Iterator Pattern** encapsulates the details of iterating through collections of items.
- **Composite Pattern** allows transparent treatment of collections and items.
- How we can bring patterns together in a complex system.

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

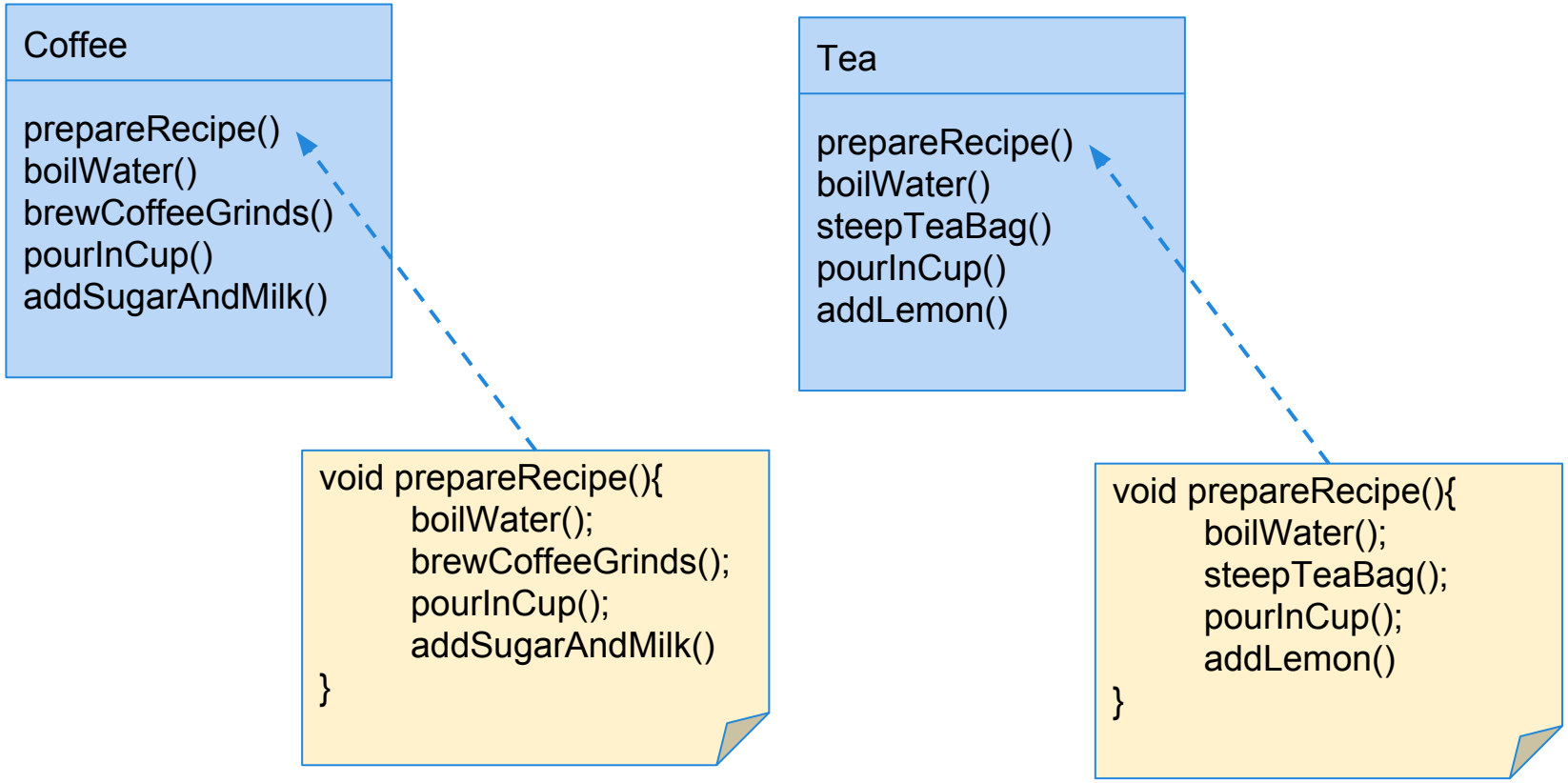
Starbuzz Tea Recipe

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

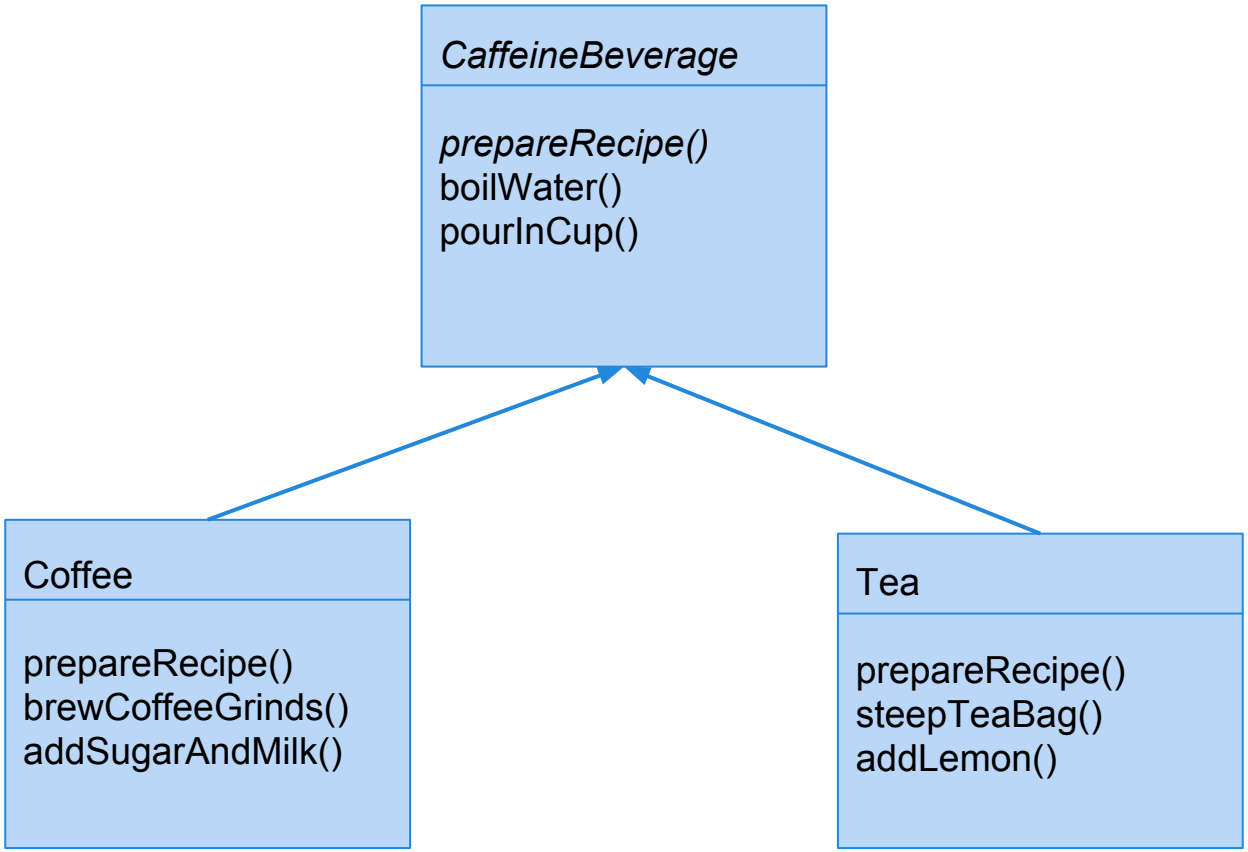
All recipes are Starbuzz Coffee trade secrets and should be kept strictly confidential.

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

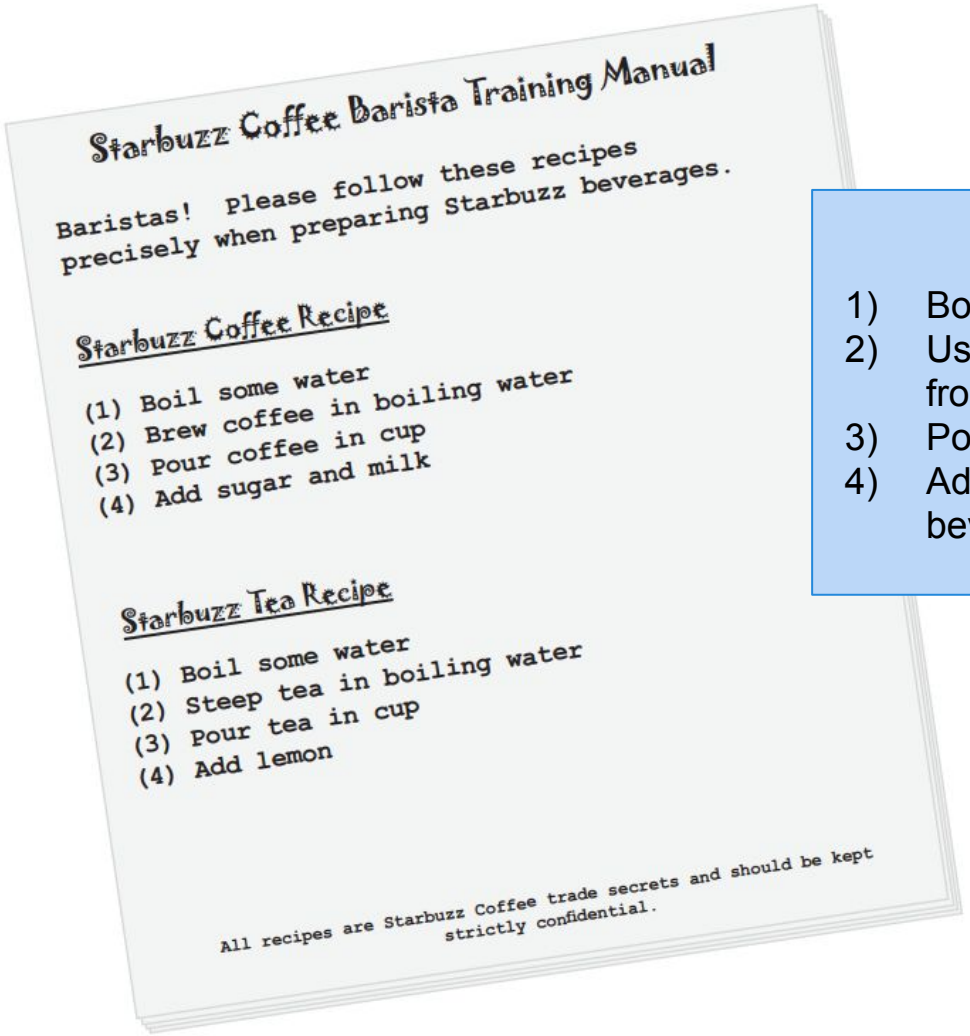
Coffee and Tea (In Code)



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.

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

Abstracting `prepareRecipe()`

- Coffee uses `brewCoffeeGrinds()` and `addSugarAndMilk()`, Tea uses `steepTeaBag()` and `addLemon()`.
  - But steeping and brewing aren't all that different.
    - Rename both to `brew()`.
  - Adding sugar is just like adding lemon.
    - Rename both to `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();

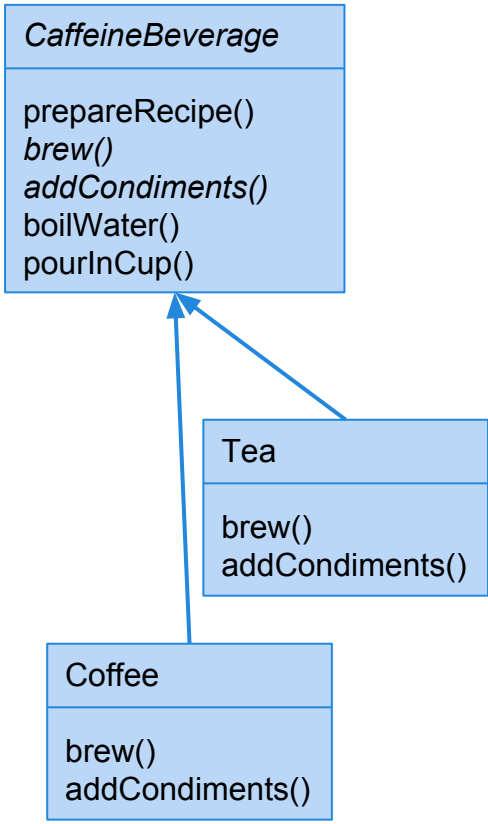
    void boilWater() {
        System.out.println("Boiling water");
    }

    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?

- We've recognized that two recipes are essentially the same, although some of the steps require *different implementations*.
- We have generalized the recipe and placed it in a base class.
  - CaffeineBeverage knows and 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**.
  - It is a method.
  - It serves as a template for an algorithm.
- In the template, each step of the algorithm is represented by a method.
- Some methods are handled by the base class, others are handled by the subclasses.
  - The methods that need to be supplied by a subclass are declared abstract.

What Does the Template Method Get Us?

Original Implementation

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

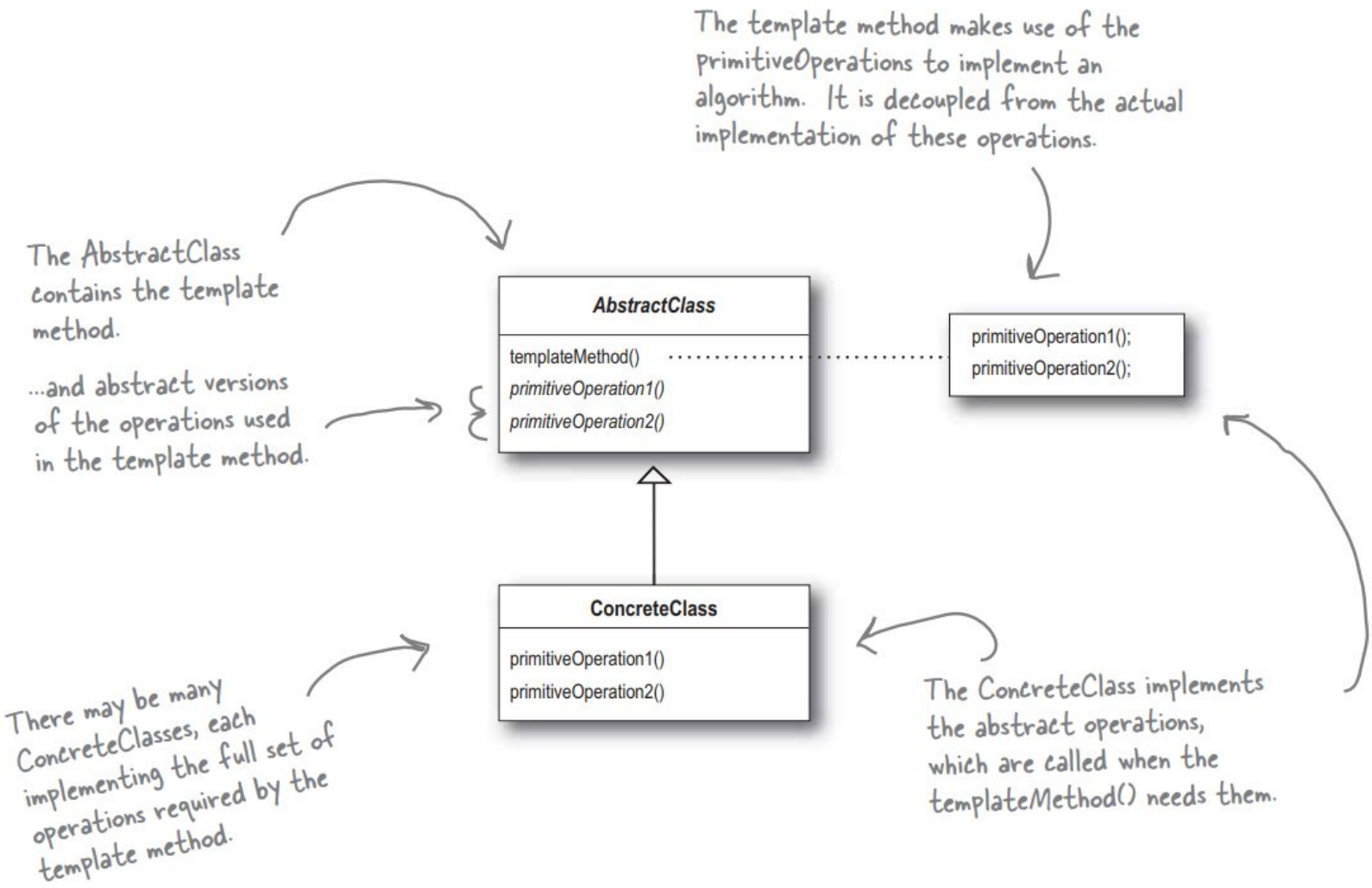
Template Method:

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

The 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.
- A template is a method that defines an algorithm as a set of steps.
  - Abstract steps are implemented by subclasses.
  - Ensures the algorithm's structure stays unchanged.

Template Method Pattern



Looking Inside the Code


```
abstract class AbstractClass {

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

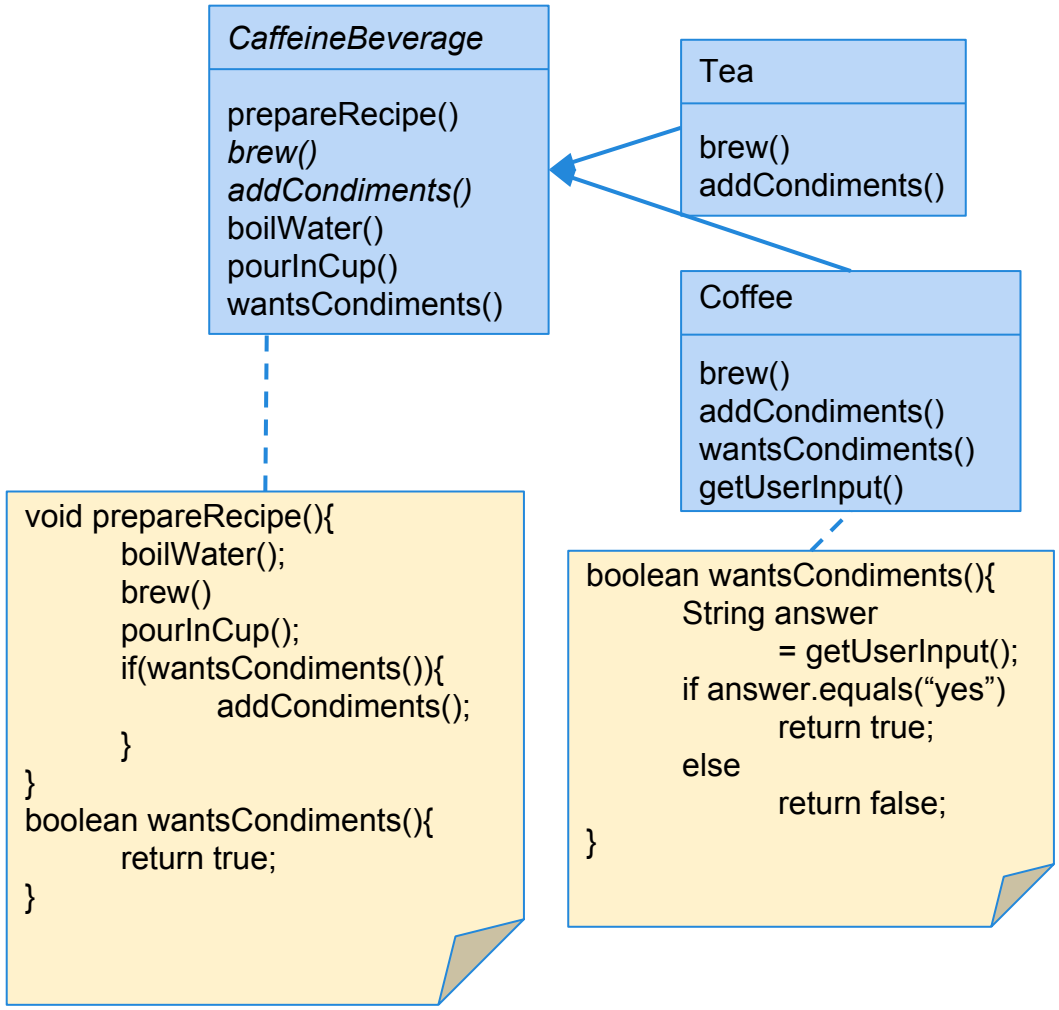
    abstract void primitiveOperation1();

    abstract void primitiveOperation2();

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

Adding Hooks

- The parent class can define concrete methods that are empty or have a default implementation (called **hooks**).
  - Subclasses can override these, but do not have to.
  - Gives subclasses the ability to “hook into” the algorithm if they wish.



The Hollywood Principle

- Don’t call us, we’ll call you.
- Prevents “dependency rot”.
  - When high-level components depend on low-level components, and those components depend on high-level components, etc, etc.
  - When you have dependency rot, it is hard to understand how a system is designed.
- The Hollywood Principle allows low-level components to hook into a system, but the high-level components decide when and how they are needed.

The Diner and Pancake House Merge

- The owners of the diner and pancake house have agreed on an implementation for the menu items…
- But they can't agree on how to implement the menus themselves.
  - Both have invested time in writing their own code.
  - Pancake house uses an ArrayList to hold items.
  - Diner uses an Array.

Menu Implementations

- Pancake House
  - Items stored in an ArrayList.
    - Allows easy menu expansion.
  - Each item is added using `addItem(name, description, vegetarian, price)`
    - Creates a new instance of MenuItem, passing in each argument.
  - `getMenuItems()` returns the list of items.
  - There are several other methods that depend on the ArrayList implementation.
- Diner
  - Items are stored in an Array.
    - Allows control over the maximum size of the menu.
  - `addItem(...)` creates a MenuItem and checks whether the array is full.
  - `getMenuItems()` returns the array.
  - There are also several methods that depend on Array implementation.

Why is this a Problem?

- Waitress class should be able to print the full, breakfast, lunch, and vegetarian menu, and check whether an item is vegetarian.
- How would we implement this?
  - getMenuItems() returns different data types.
  - If we iterate over both menus, we need two loops.
  - Every method requires custom code for both implementations. If we add another restaurant, we need three loops.
- Implementation will be hard to evolve.
  - If both implemented the same interface, we could minimize concrete references and only use one loop to iterate.

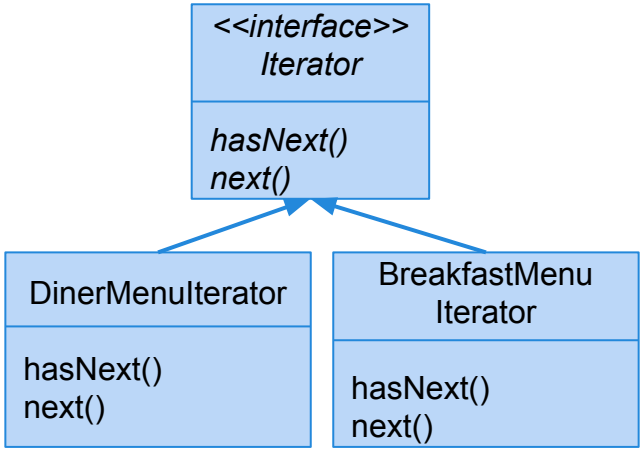
Encapsulating the Iteration

- What changes here is how we iterate over different collections of objects.
  - To iterate over an ArrayList, we use `size()` and `get()` methods on the collection.
  - Over the Array, we use `.length` and `array[i]`.
- Create an “Iterator” that encapsulates how we walk through a collection:

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

The Iterator Pattern

- Relies on an interface called Iterator.
  - hasNext() tells us if there are more elements.
  - next() returns the next item.
- Implement concrete Iterators for any type of collection that we can make use of.
  - Each contains concrete details for ArrayList, Array, etc.



Iterators for the Restaurants

- DinerMenuIterator
  - Maintains an index for the current array position.
  - Constructor takes in the array and sets position to 0.
  - `next()` gets the item at the current position, increments position, and returns the item.
  - `hasNext()` checks to see if the position is at the end, or if the next element is null (the menu isn't full, but we've seen everything).
- The DinerMenu class must add a method `createIterator()` that returns a new Iterator.
  - The object returned is an instance of DinerMenu Iterator, but the return type is the generic Iterator.
  - The client does not need to know what type of Iterator it is working with - all have the same methods.
- We can get rid of `getMenuItems()`. The Iterator replaces it in a generic manner.
- The Waiter class can be rewritten to remove redundant loops.

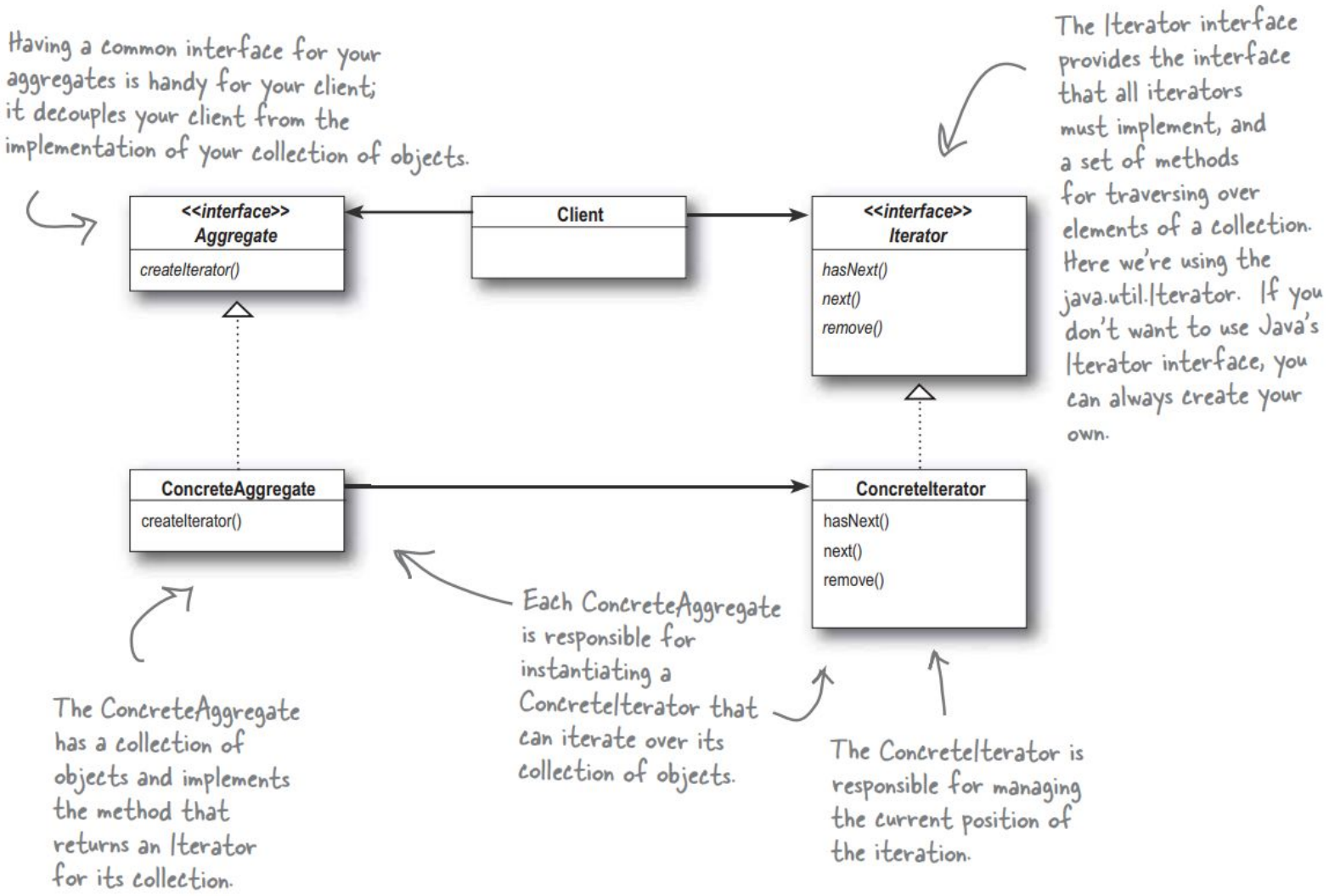
Before and After

- Before Iterators
  - Menus are not well encapsulated. One uses ArrayList and the other uses Array.
  - We need two loops to iterate over all MenuItems.
  - The Waitress is bound to concrete classes MenuItem[] and ArrayList.
  - The Waiter is bound to two concrete Menu classes, with near-identical interfaces.
- After Iterators
  - Menu implementations are encapsulated. The Waitress does not know how Menus hold their collections.
  - We need one loop that polymorphically handles any collection of items.
  - The Waitress now uses an interface (Iterator).
  - We can define a Menu interface that has method `createIterator()`.

Iterator Pattern Defined

- The **Iterator Pattern** provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
- Once you have a uniform way to access elements of all aggregate objects, your code will work with *any* of these aggregates.
- Iterator Pattern takes the responsibility of traversing collections from the collection to the Iterator itself.

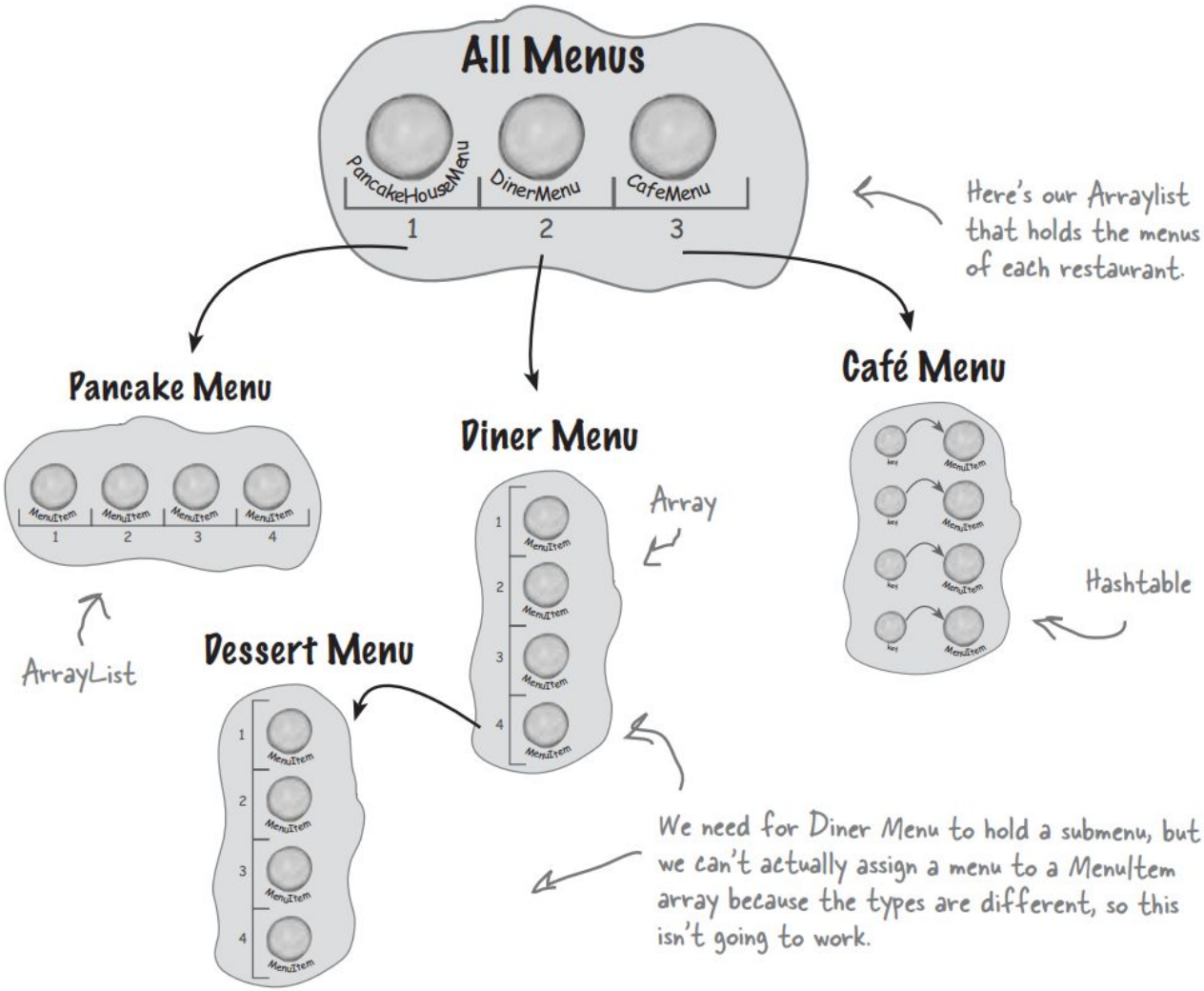
Iterator Pattern



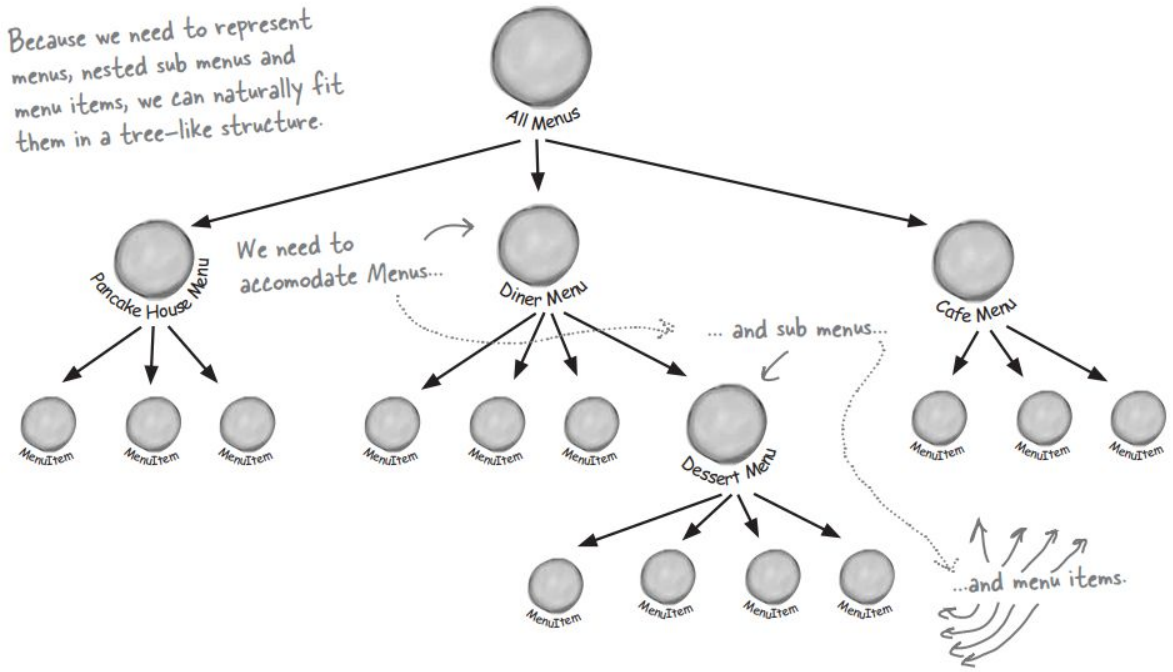
The Principle of Single Responsibility

- Why do we not let collections manage their own iteration?
  - Why is it bad to increase the number of methods?
- This would give the collection two reasons to change if we were to evolve the class.
  - **A class should have only one reason to change.**
  - Protect what changes from what might not (or from other aspects that change).
  - Assign each responsibility to one class alone.
    - This is hard to ensure. Look for signals that a class is changing in multiple ways as the system grows.

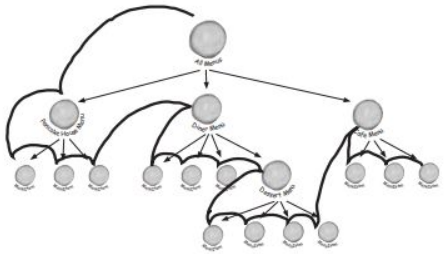
Adding a Dessert Submenu



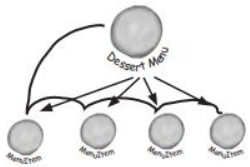
What Do We Need?



We still need to be able to traverse the all the items in the tree.



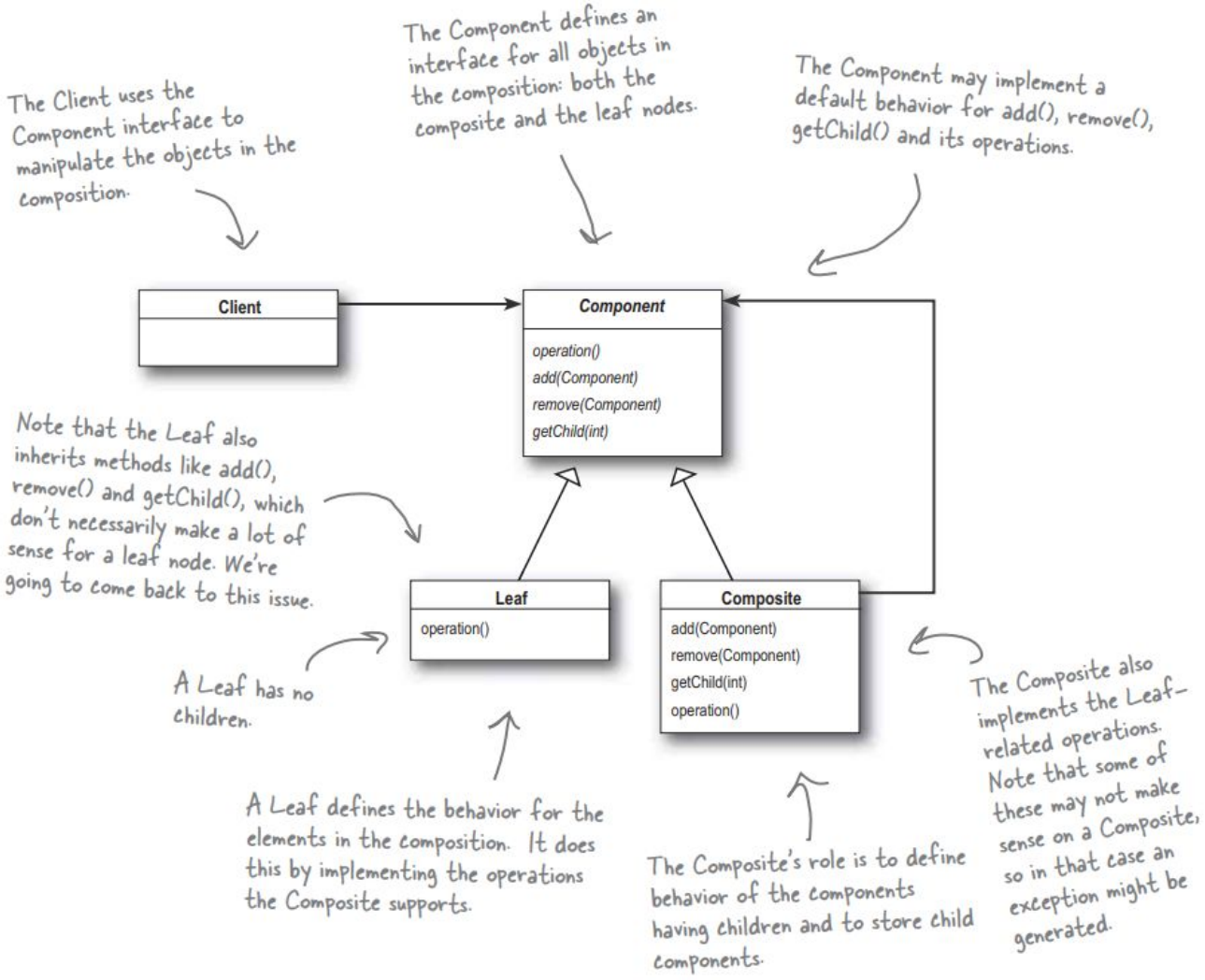
We also need to be able to traverse more flexibly, for instance over one menu.



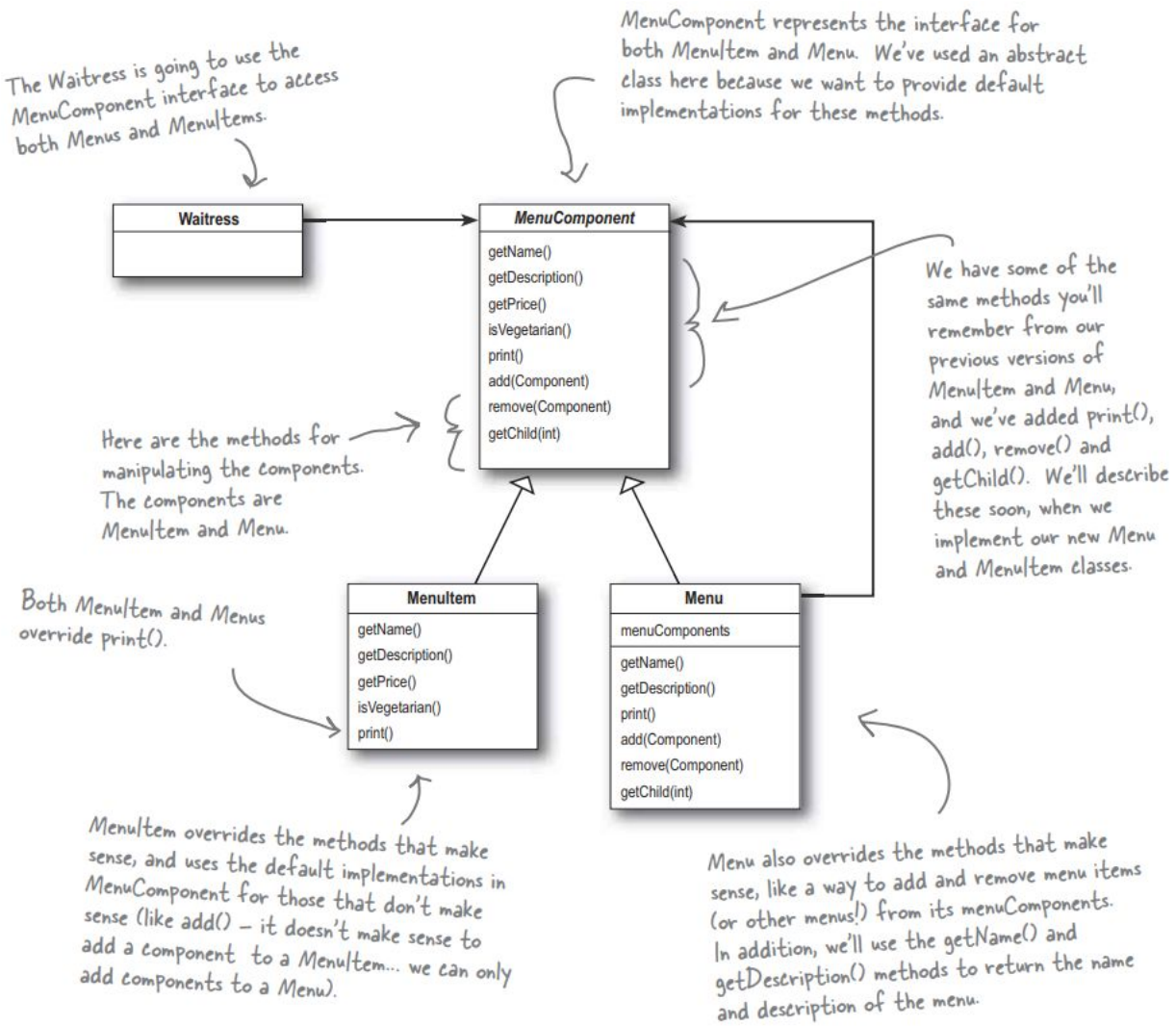
The Composite Pattern

- **The Composite Pattern** allows you to compose objects into tree structures to represent hierarchies.
- Composite lets clients treat individual objects and compositions of objects uniformly.
  - Elements with child elements are called **nodes**.
  - Elements without children are called **leaves**.
  - Menus are nodes, MenuItems are leaves.
    - Menus are compositions of other Menus and MenuItems.
  - The Composite Pattern allows us to write code that can apply the same operation (i.e., printing) over the entire Menu structure.

The Composite Pattern



Designing Menus with Composite



Implementing MenuComponent

MenuComponent provides default implementations for every method.

```
public abstract class MenuComponent {

    public void add(MenuComponent menuComponent) {
        throw new UnsupportedOperationException();
    }
    public void remove(MenuComponent menuComponent) {
        throw new UnsupportedOperationException();
    }
    public MenuComponent getChild(int i) {
        throw new UnsupportedOperationException();
    }

    public String getName() {
        throw new UnsupportedOperationException();
    }
    public String getDescription() {
        throw new UnsupportedOperationException();
    }
    public double getPrice() {
        throw new UnsupportedOperationException();
    }
    public boolean isVegetarian() {
        throw new UnsupportedOperationException();
    }

    public void print() {
        throw new UnsupportedOperationException();
    }
}
```

Because some of these methods only make sense for MenuItems, and some only make sense for Menus, the default implementation is UnsupportedOperationException. That way, if MenuItem or Menu doesn't support an operation, they don't have to do anything, they can just inherit the default implementation.

We've grouped together the "composite" methods – that is, methods to add, remove and get MenuComponents.

Here are the "operation" methods; these are used by the MenuItems. It turns out we can also use a couple of them in Menu too, as you'll see in a couple of pages when we show the Menu code.

print() is an "operation" method that both our Menus and MenuItems will implement, but we provide a default operation here.

Implementing Menu Printing

```
public class Menu extends MenuComponent {
    ArrayList menuComponents = new ArrayList();
    String name;
    String description;

    // constructor code here

    // other methods here

    public void print() {
        System.out.print("\n" + getName());
        System.out.println(", " + getDescription());
        System.out.println("---------------------");

        Iterator iterator = menuComponents.iterator();
        while (iterator.hasNext()) {
            MenuComponent menuComponent =
                (MenuComponent)iterator.next();
            menuComponent.print();
        }
    }
}
```

All we need to do is change the print() method to make it print not only the information about this Menu, but all of this Menu's components: other Menus and MenuItems.

Look! We get to use an Iterator. We use it to iterate through all the Menu's components... those could be other Menus, or they could be MenuItems. Since both Menus and MenuItems implement print(), we just call print() and the rest is up to them.

NOTE: If, during this iteration, we encounter another Menu object, its print() method will start another iteration, and so on.

A Design Trade-Off

- The Composite Pattern violates the Single Responsibility principle.
  - Composite Pattern manages a hierarchy and performs operations on items in the hierarchy.
- Composite Pattern trades *safety* for *transparency*.
  - A client can treat composite and leaf nodes uniformly. The type of node (composite or leaf) is transparent to the client.
  - We lose safety because the client might try to apply inappropriate operations to an element.

Bringing Back the Ducks

- First, we need a Quackable interface.

```
public interface Quackable
    public void quack();
}
```

- Now, some Ducks that implement Quackable

```
public class DuckCall implements Quackable {
    public void quack() {
        System.out.println("Kwak");
    }
}
```

A DuckCall that quacks but doesn't sound quite like the real thing.

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

A RubberDuck that makes a squeak when it quacks.

Your standard Mallard duck.

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

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

We've got to have some variation of species if we want this to be an interesting simulator.

… And Our Duck Simulator

```
public class DuckSimulator {
    public static void main(String[] args) {
        DuckSimulator simulator = new DuckSimulator();
        simulator.simulate();
    }

    void simulate() {
        Quackable mallardDuck = new MallardDuck();
        Quackable redheadDuck = new RedheadDuck();
        Quackable duckCall = new DuckCall();
        Quackable rubberDuck = new RubberDuck();

        System.out.println("\nDuck Simulator");

        simulate(mallardDuck);
        simulate(redheadDuck);
        simulate(duckCall);
        simulate(rubberDuck);
    }

    void simulate(Quackable duck) {
        duck.quack();
    }
}
```

Here's our main method to get everything going.

We create a simulator and then call its simulate() method.

We need some ducks, so here we create one of each Quackable...

... then we simulate each one.

Here we overload the simulate method to simulate just one duck.

Here we let polymorphism do its magic: no matter what kind of Quackable gets passed in, the simulate() method asks it to quack.

Challenge 1: Geese!

- Geese are not Ducks, but we can make them Ducks using the **Adapter** Pattern

```
public class Goose {
    public void honk() {
        System.out.println("Honk");
    }
}
```

A Goose is a honker, not a quacker.

Remember, an Adapter implements the target interface, which in this case is Quackable.

```
public class GooseAdapter implements Quackable {
    Goose goose;

    public GooseAdapter(Goose goose) {
        this.goose = goose;
    }

    public void quack() {
        goose.honk();
    }
}
```

The constructor takes the goose we are going to adapt.

When quack is called, the call is delegated to the goose's honk() method.

Integrating Geese into the Simulator

```
public class DuckSimulator {
    public static void main(String[] args) {
        DuckSimulator simulator = new DuckSimulator();
        simulator.simulate();
    }
    void simulate() {
        Quackable mallardDuck = new MallardDuck();
        Quackable redheadDuck = new RedheadDuck();
        Quackable duckCall = new DuckCall();
        Quackable rubberDuck = new RubberDuck();
        Quackable gooseDuck = new GooseAdapter(new Goose());

        System.out.println("\nDuck Simulator: With Goose Adapter");

        simulate(mallardDuck);
        simulate(redheadDuck);
        simulate(duckCall);
        simulate(rubberDuck);
        simulate(gooseDuck);
    }

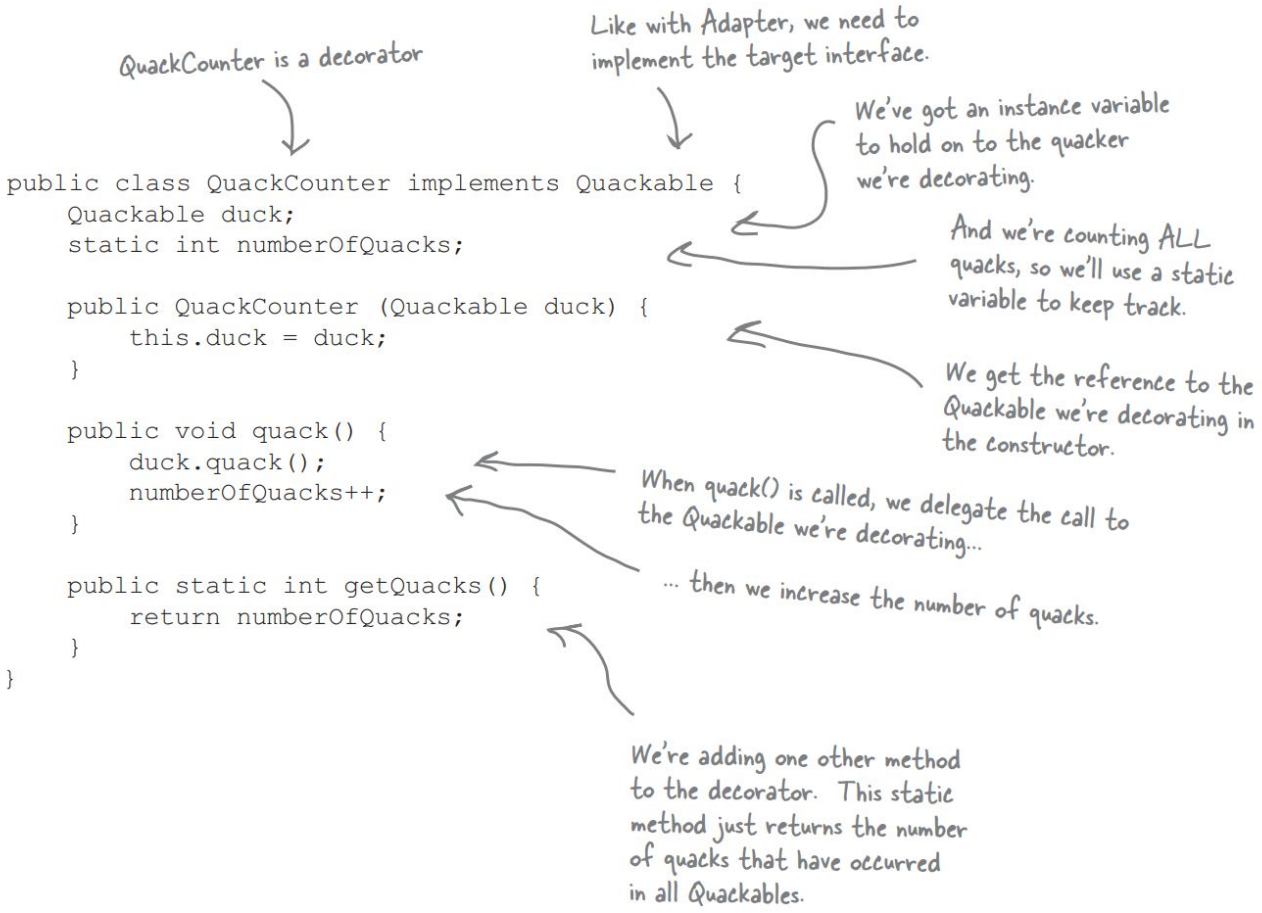
    void simulate(Quackable duck) {
        duck.quack();
    }
}
```

We make a Goose that acts like a Duck by wrapping the Goose in the GooseAdapter.

Once the Goose is wrapped, we can treat it just like other duck Quackables.

Challenge 2: Counting Quacks

- A biologist wants us to count the quacks, for some reason…
- We can create a **Decorator** that gives Ducks a new behavior (counting) by wrapping a Duck with a decorator object.
  - We do not need to change Duck behavior at all.



Integrating Counting into the Sim

```
public class DuckSimulator {
    public static void main(String[] args) {
        DuckSimulator simulator = new DuckSimulator();
        simulator.simulate();
    }
    void simulate() {
        Quackable mallardDuck = new QuackCounter(new MallardDuck());
        Quackable redheadDuck = new QuackCounter(new RedheadDuck());
        Quackable duckCall = new QuackCounter(new DuckCall());
        Quackable rubberDuck = new QuackCounter(new RubberDuck());
        Quackable gooseDuck = new GooseAdapter(new Goose());

        System.out.println("\nDuck Simulator: With Decorator");

        simulate(mallardDuck);
        simulate(redheadDuck);
        simulate(duckCall);
        simulate(rubberDuck);
        simulate(gooseDuck);

        System.out.println("The ducks quacked " +
                           QuackCounter.getQuacks() + " times");
    }

    void simulate(Quackable duck) {
        duck.quack();
    }
}
```

Each time we create a Quackable, we wrap it with a new decorator.

The park ranger told us he didn't want to count geese honks, so we don't decorate it.

Here's where we gather the quacking behavior for the Quackologists.

Nothing changes here; the decorated objects are still Quackables.

Challenge 3: Easy Duck Creation

- We might find that too many Ducks are being created without the decorator.
  - You have to remember to decorate objects to get decorated behavior.
- We should take Duck creation and localize it to one code location.
  - Take Duck creation and decorating and encapsulate it in one spot.
- We can create a Duck **Factory**.

Duck Factory

```
public abstract class AbstractDuckFactory {

    public abstract Quackable createMallardDuck();
    public abstract Quackable createRedheadDuck();
    public abstract Quackable createDuckCall();
    public abstract Quackable createRubberDuck();
}
```

We're defining an abstract factory that subclasses will implement to create different families.

creates one kind of duck.

```
public class DuckFactory extends AbstractDuckFactory {

    public Quackable createMallardDuck() {
        return new MallardDuck();
    }

    public Quackable createRedheadDuck() {
        return new RedheadDuck();
    }

    public Quackable createDuckCall() {
        return new DuckCall();
    }

    public Quackable createRubberDuck() {
        return new RubberDuck();
    }
}
```

DuckFactory extends the abstract factory.

Each method creates a product: a particular kind of Quackable. The actual product is unknown to the simulator – it just knows it's getting a Quackable.

Decorated Duck Factory

CountingDuckFactory also extends the abstract factory.

```
public class CountingDuckFactory extends AbstractDuckFactory {

    public Quackable createMallardDuck() {
        return new QuackCounter(new MallardDuck());
    }

    public Quackable createRedheadDuck() {
        return new QuackCounter(new RedheadDuck());
    }

    public Quackable createDuckCall() {
        return new QuackCounter(new DuckCall());
    }

    public Quackable createRubberDuck() {
        return new QuackCounter(new RubberDuck());
    }
}
```

Each method wraps the Quackable with the quack counting decorator. The simulator will never know the difference; it just gets back a Quackable. But now our rangers can be sure that all quacks ar being counted.

Integrating the Factory

```
public class DuckSimulator {
    public static void main(String[] args) {
        DuckSimulator simulator = new DuckSimulator();
        AbstractDuckFactory duckFactory = new CountingDuckFactory();

        simulator.simulate(duckFactory);
    }

    void simulate(AbstractDuckFactory duckFactory) {
        Quackable mallardDuck = duckFactory.createMallardDuck();
        Quackable redheadDuck = duckFactory.createRedheadDuck();
        Quackable duckCall = duckFactory.createDuckCall();
        Quackable rubberDuck = duckFactory.createRubberDuck();
        Quackable gooseDuck = new GooseAdapter(new Goose());

        System.out.println("\nDuck Simulator: With Abstract Factory");

        simulate(mallardDuck);
        simulate(redheadDuck);
        simulate(duckCall);
        simulate(rubberDuck);
        simulate(gooseDuck);

        System.out.println("The ducks quacked " +
                           QuackCounter.getQuacks() +
                           " times");
    }

    void simulate(Quackable duck) {
        duck.quack();
    }
}
```

First we create the factory that we're going to pass into the simulate() method.

The simulate() method takes an AbstractDuckFactory and uses it to create ducks rather than instantiating them directly.

Nothing changes here! Same ol' code.

Challenge 4: Managing a Flock

- This is not very manageable.

```
Quackable mallardDuck = duckFactory.createMallardDuck();
Quackable redheadDuck = duckFactory.createRedheadDuck();
Quackable duckCall = duckFactory.createDuckCall();
Quackable rubberDuck = duckFactory.createRubberDuck();
Quackable gooseDuck = new GooseAdapter(new Goose());

simulate(mallardDuck);
simulate(redheadDuck);
simulate(duckCall);
simulate(rubberDuck);
simulate(gooseDuck);
```

- Why are we managing ducks individually?
  - We need a way to talk about collections of Ducks (flocks?) or subcollections of Ducks (to please our biologist).
  - The **Composite** pattern allows us to treat a group as we would an individual.

Creating a Flock

Remember, the composite needs to implement the same interface as the leaf elements. Our leaf elements are Quackables.

```
public class Flock implements Quackable {
    ArrayList quackers = new ArrayList();

    public void add(Quackable quacker) {
        quackers.add(quacker);
    }

    public void quack() {
        Iterator iterator = quackers.iterator();
        while (iterator.hasNext()) {
            Quackable quacker = (Quackable)iterator.next();
            quacker.quack();
        }
    }
}
```

We're using an ArrayList inside each Flock to hold the Quackables that belong to the Flock.

The add() method adds a Quackable to the Flock.

(Also - spot the **Iterator**)

Now for the quack() method – after all, the Flock is a Quackable too. The quack() method in Flock needs to work over the entire Flock. Here we iterate through the ArrayList and call quack() on each element.

Integrating the Flock

```
public class DuckSimulator {
    // main method here

    void simulate(AbstractDuckFactory duckFactory) {
        Quackable redheadDuck = duckFactory.createRedheadDuck();
        Quackable duckCall = duckFactory.createDuckCall();
        Quackable rubberDuck = duckFactory.createRubberDuck();
        Quackable gooseDuck = new GooseAdapter(new Goose());
        System.out.println("\nDuck Simulator: With Composite - Flocks");

        Flock flockOfDucks = new Flock();

        flockOfDucks.add(redheadDuck);
        flockOfDucks.add(duckCall);
        flockOfDucks.add(rubberDuck);
        flockOfDucks.add(gooseDuck);

        Flock flockOfMallards = new Flock();

        Quackable mallardOne = duckFactory.createMallardDuck();
        Quackable mallardTwo = duckFactory.createMallardDuck();
        Quackable mallardThree = duckFactory.createMallardDuck();
        Quackable mallardFour = duckFactory.createMallardDuck();

        flockOfMallards.add(mallardOne);
        flockOfMallards.add(mallardTwo);
        flockOfMallards.add(mallardThree);
        flockOfMallards.add(mallardFour);

        flockOfDucks.add(flockOfMallards);

        System.out.println("\nDuck Simulator: Whole Flock Simulation");
        simulate(flockOfDucks);

        System.out.println("\nDuck Simulator: Mallard Flock Simulation");
        simulate(flockOfMallards);

        System.out.println("\nThe ducks quacked " +
                           QuackCounter.getQuacks() +
                           " times");
    }

    void simulate(Quackable duck) {
        duck.quack();
    }
}
```

Create all the Quackables, just like before.

First we create a Flock, and load it up with Quackables.

Then we create a new Flock of Mallards.

Here we're creating a little family of mallards...

...and adding them to the Flock of mallards.

Then we add the Flock of mallards to the main flock.

Let's test out the entire Flock!

Then let's just test out the mallard's Flock.

Finally, let's give the Quackologist the data.

Nothing needs to change here, a Flock is a Quackable!

Wrapping Up

- **Template Method Pattern** encapsulates pieces of algorithms so that subclasses can hook into a computation.
- **Iterator Pattern** encapsulates the details of iterating through collections of items.
- **Composite Pattern** allows transparent treatment of collections and items.

Next Time

- From Design to Implementation
  - Modeling dynamic behavior of objects.
  - UML sequence diagrams
  - Implementation practices
  - Reading
    - Sommerville, chapter 5, 7
    - Fowler, chapter 4
- Homework 3
  - Due April 7