

# Best Practices in Programming

## Design Patterns meet Design Principles

**Lubomír Bulej**

KDSS MFF UK

# Software design

## Finding solutions to complex problems

- Many sources of complexity
  - Functionality, cost constraints, performance, security, backwards compatibility, reliability, scalability, ...

## Reusing code makes our life easier

- Imagine not having the Java Class Library...

## Can we reuse design knowledge as well?

- Design concerns structure and trade-offs.
- Design is not code, but art and experience.

# Good software design

## What is good software design?

- Flexible, reusable, maintainable...
- Naturally fits the requirements.
- Robust in the face of changes.

## How to arrive at good design?

- Start with a simple solutions.
- Stimulate changes to force design changes.
- Apply OO principles when necessary.

**Do we have to always reinvent the wheel?**

# Meet design patterns

## **Someone has already solved your problem!**

- Many design problems occur repeatedly in different situations.
- Many people have solved them over the years.
- Patterns can be found in the structure of the successful solutions.

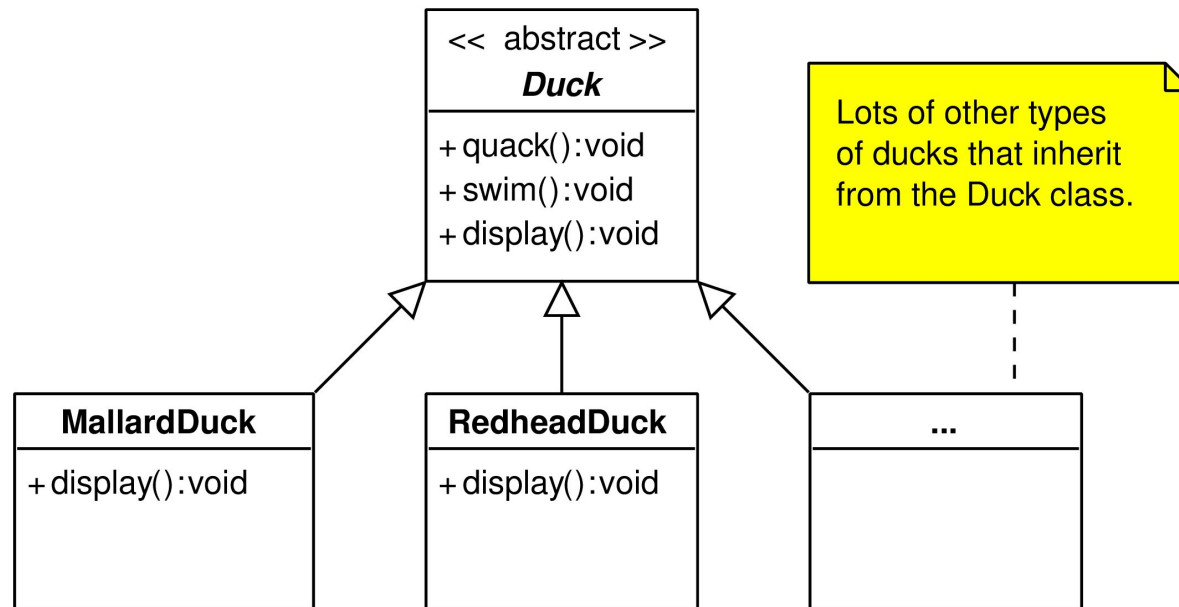
## **Design patterns = reusable design knowledge**

- Exploit the wisdom and lessons learned by other developers working on similar problems.
- Often the result of applying OO principles.

# It all started with SimUDuck

## Duck pond simulation game

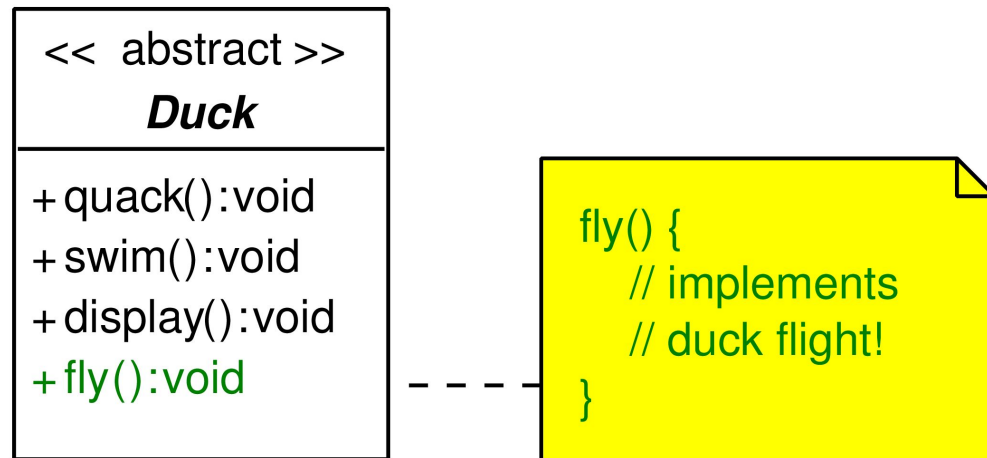
- Variety of duck species swimming around and making quacking noises.
- Initial design created using standard OO techniques.



# But now the ducks need to fly!

## Competitive advantage in duck simulation market

- Luckily, we are using inheritance...
- We can add a fly() method in the Duck class that all Duck subclasses will inherit.

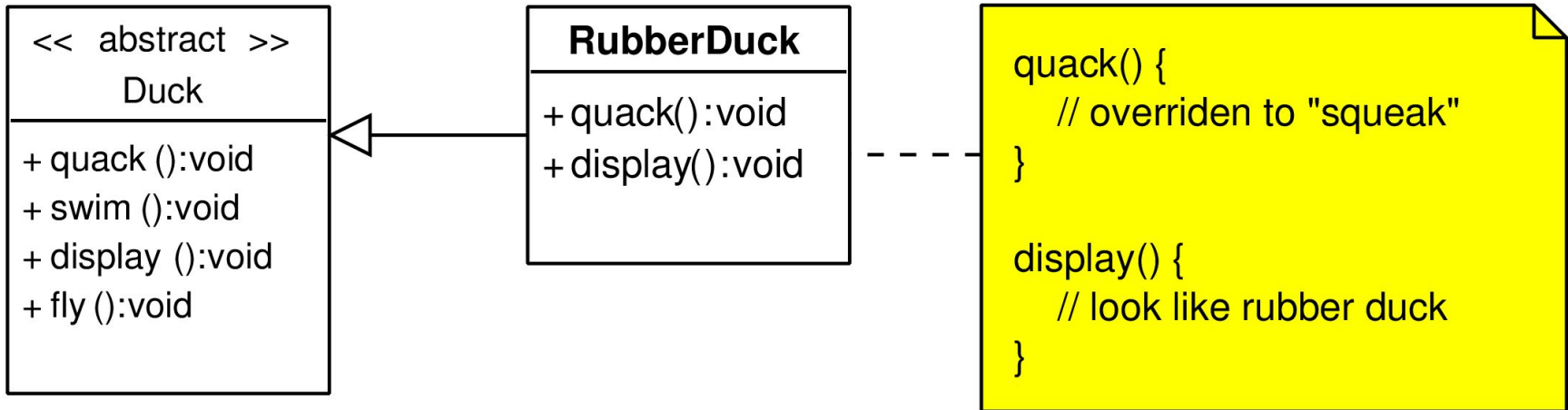


What could possibly go wrong?

# Something did go wrong...

## The rubber ducks are flying too!

- The boss (giving a demo to shareholders) is unhappy!



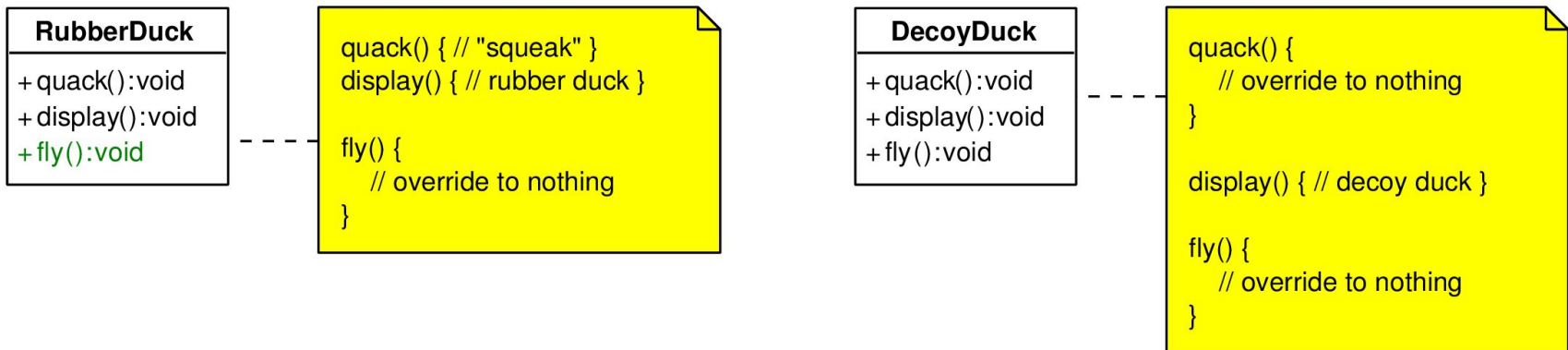
## What happened?

- Non-local side effect (flying rubber ducks) caused by local code change!
- Not all Duck subclasses are supposed to fly.

# Thinking about inheritance...

## The problem is easy to fix

- We could just override the fly() method in RubberDuck.
- But what if we want to add a DecoyDuck, which is not supposed to fly or quack?
  - Let's add it too for good measure!

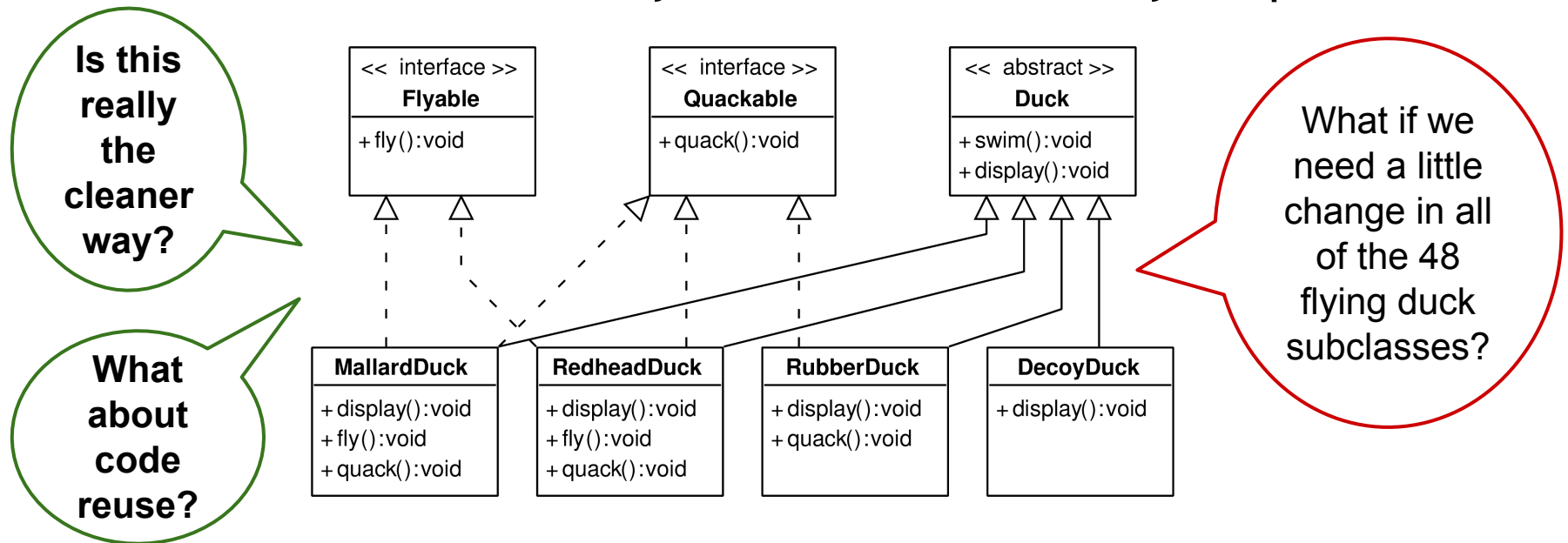




# Inheritance is not the answer

## How about an interface?

- Executives want to update products every 6 months.
  - But they have not yet decided in which ways...
- Chances are, we'll be overriding the fly() and quack() in every new Duck subclass added do SimUDuck.
  - Need cleaner way for some ducks to fly or quack.



# There will always be change...

## Zeroing in on the problem...

- Inheritance did not work out very well.
  - Behavior keeps changing across subclasses.
  - Behavior is not appropriate for all subclasses.
- Interfaces seemed promising.
  - Only some ducks will be Flyable or Quackable.
  - But not much code reuse (prior to Java 8).
    - Java 8 introduced default methods in interfaces.
  - Modifying behavior means tracking it down in all the classes that implement it.

# Design principle

## Encapsulate what changes

- Identify the aspects of application that vary and separate them from what stays the same.
- In other words, encapsulate what varies so that it does not affect the rest of the code.

## The result

- Parts that vary can be altered or extended without affecting code that does not change.
- Fewer unintended consequences from code changes and more flexibility!
- Most patterns provide a way to let some part of a system vary independently of all other parts.

# How does it concern SimUDuck?

## Separating what changes ...

- The Duck class is working well.
- Problems with fly() and quack().
  - These are the parts that vary across ducks.

## Pulling the methods out of Duck

- Create classes to represent duck behaviors.
  - Quacking behaviors: quacking, squeaking, silence.
  - Flying behaviors: with wings, not flying at all.
- Assign behaviors to instances of Duck.

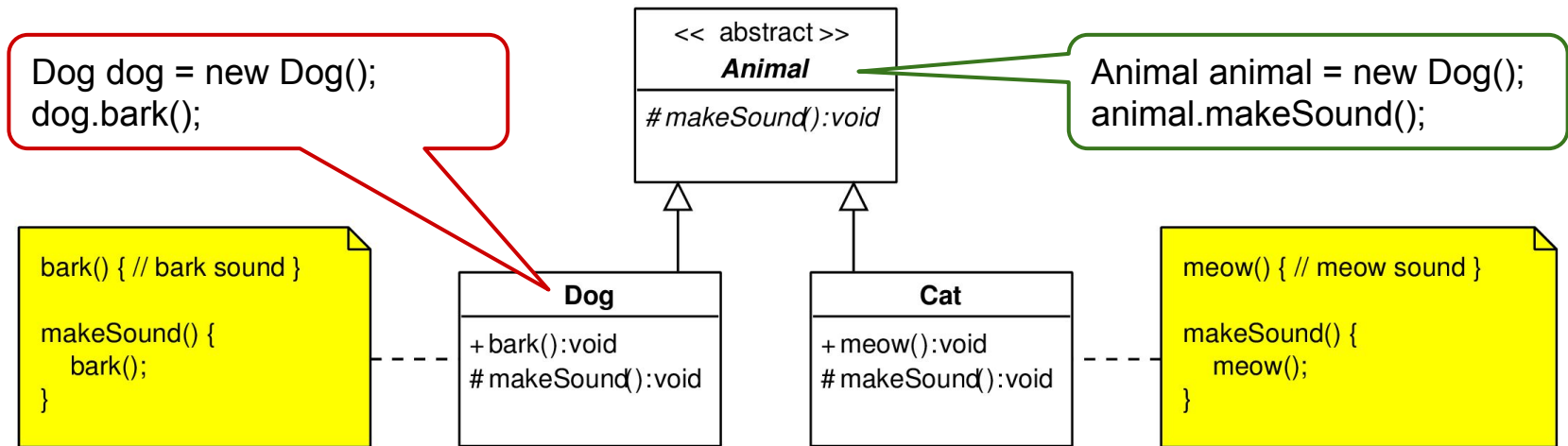
## We could even change it dynamically!

- Avoid depending on a particular behavior.

# Design principle

## Program to an **interface**, not **implementation**

- Program to an interface == program to a supertype.
- Weaker form of *dependency inversion principle*.



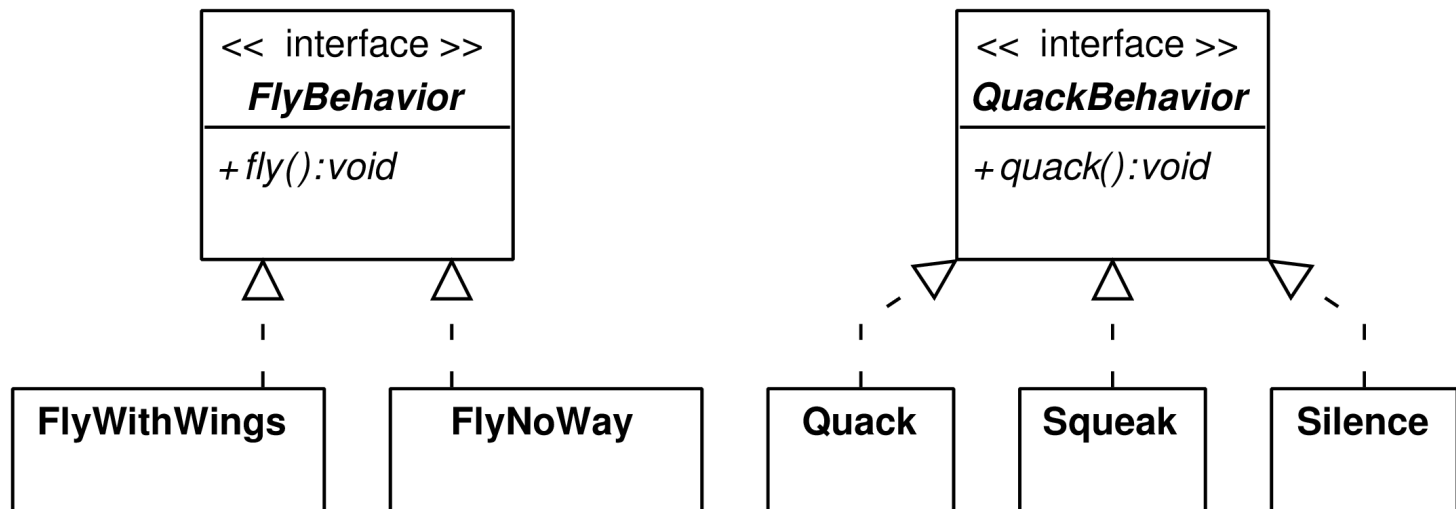
## The result

- The actual type of objects is not locked into code.

# Designing the Duck behaviors

## Interfaces for FlyBehavior and QuackBehavior

- Corresponding classes implement concrete behavior.



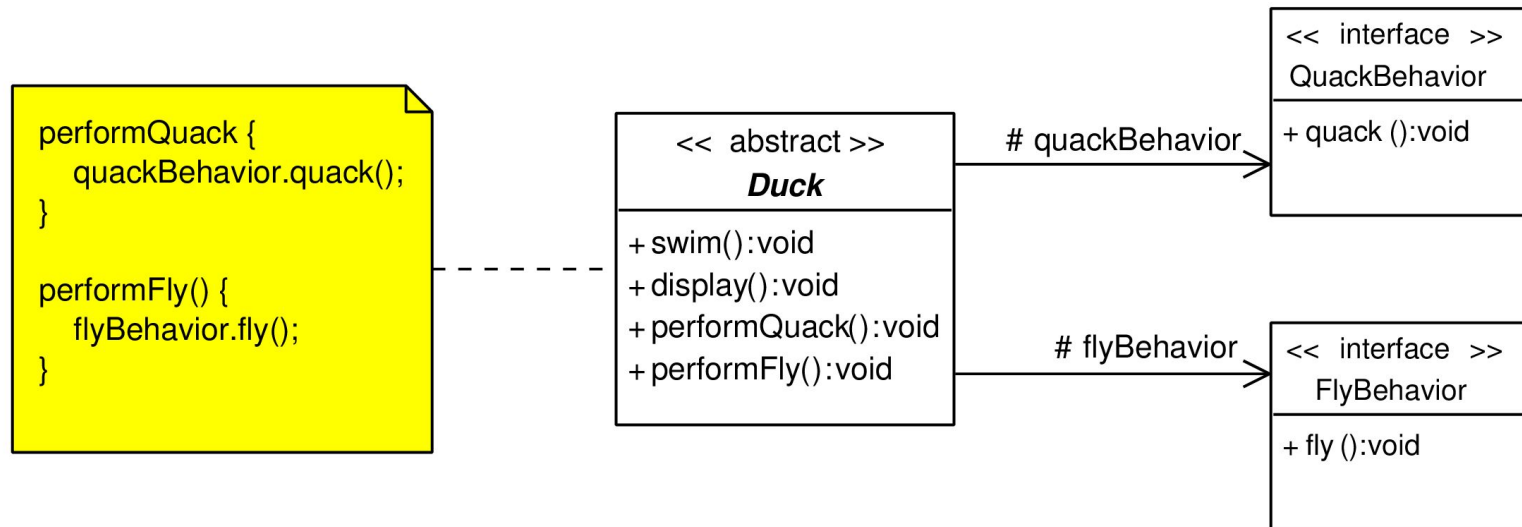
## The result

- Behaviors can be reused elsewhere.
- New behavior can be added without modifying the existing behaviors or any of the Duck classes.

# Integrating the Duck behavior

## Delegate flying and quacking behavior

- Remember to program to the interface.



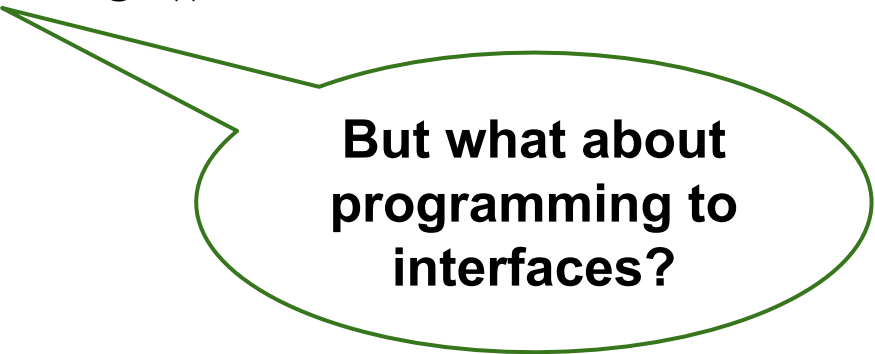
## The result

- Behaviors can be reused elsewhere.
- New behavior can be added without modifying the existing behaviors or any of the Duck classes.

# Integration with Duck subclasses

## Configure behavior in the constructor

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



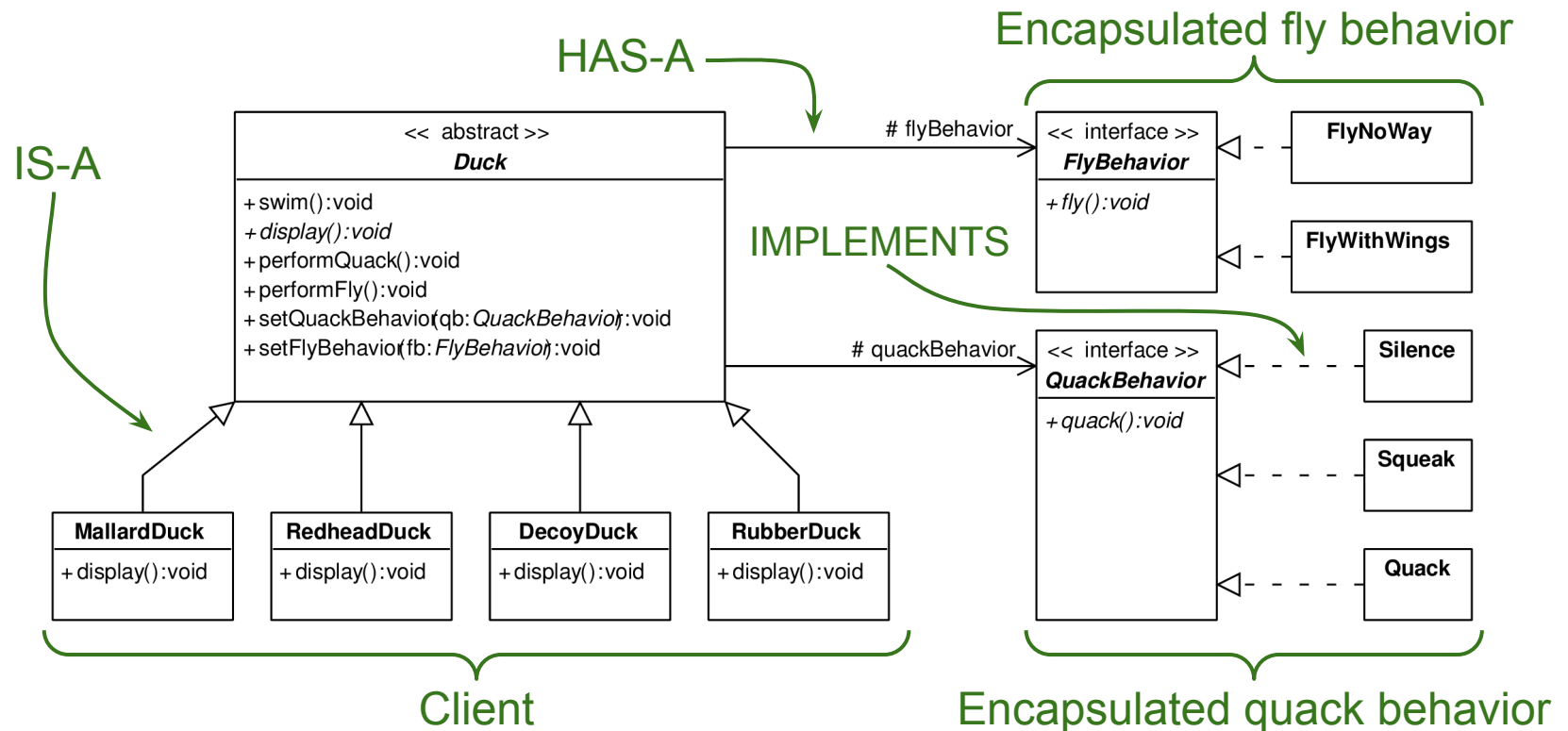
**But what about  
programming to  
interfaces?**

## Change behavior on the fly

```
public abstract class Duck {  
    protected FlyBehavior flyBehavior;  
    public void setFlyBehavior (FlyBehavior flyBehavior) {  
        this.flyBehavior = flyBehavior;  
    }  
    ...  
}
```



# The big picture



Set of behaviors == family of algorithms

- Algorithms are interchangeable.
- Client uses an encapsulated family of algorithms.

Note the relationships between the classes

# Design principle

## Favor composition over inheritance

- Inheritance represents IS-A relationship.
  - Represents the static properties of a type.
- Composition represents HAS-A relationship.
  - Delegates behavior to other classes.

## The result

- Composition provides more flexibility.
  - Enables encapsulating a family of algorithms into their own set of classes.
- Allows changing behavior at runtime!
- Composition used in many design patterns.

# Speaking of design patterns...

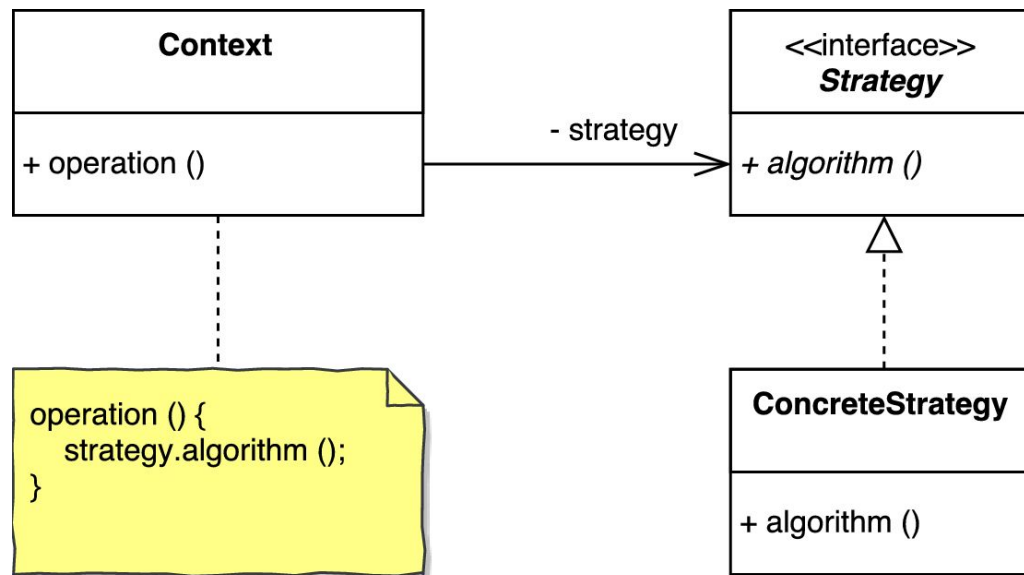
## **We have just applied the Strategy pattern!**

- Just by following a few design principles
  - Encapsulating what varies.
  - Programming to the interface.
  - Favoring composition over inheritance.
- The SimUDuck app is now ready for “any” changes that the executives come up with!
  - For some “strategic definition” of any...

# The Strategy pattern

## Intent and structure

- Defines a family of algorithms, encapsulates each one, and makes them interchangeable.
- Lets the key algorithm vary independently from clients that use it.



# Do we really need patterns?

## Knowing OO basics is not enough

- Abstraction, encapsulation, inheritance, and polymorphism are just tools.
  - Tools don't make us good designers.
- Building systems that are flexible, reusable, and maintainable is not obvious.
  - Patterns discovered through hard work.
  - Patterns let you reuse experience.

## OO design principles deal with change

- Underlying many design patterns.
- Fallback if no suitable pattern matches.

# Shared pattern vocabulary

## Vocabulary of powerful concepts

- Pattern name communicates a whole set of qualities, characteristics, and constraints.
- Makes communication with other developers easier.
  - Say more with less.
- Allows staying “in the design” longer.
  - Keep discussion at the design level instead of dealing with implementation details.
- Development team can move more quickly.
  - Less room for misunderstanding.

# Design pattern history

## Inspired by architecture

- Christopher Alexander, 1977
  - A pattern language to describe visually pleasing and practical structures for buildings and towns.

## Introduced to OO design

- Kent Beck and Ward Cunningham, 1987
  - *Using Pattern Languages for Object-Oriented Programs*
  - Applied Alexander's work to user-interface design.
- Research on identifying and describing patterns, 1990s
- E. Gamma, R. Helm, R. Johnson, J. Vlissides, 1995
  - *Design Patterns: Elements of Reusable Object-Oriented Software*
  - Catalogue of 23 patterns in OO design and their solutions.
  - Categorized by purpose and scope, consistent description format.

# Describing design patterns (1)

## **Name and classification**

- Conveys the essence of the pattern.
- Conceptual handle for design discussion.

## **Intent**

- What does it do? What is its rationale and intent?
- What design issue or problem it solves?

## **Also known as**

- Other well-known names for the pattern, if any.

## **Motivation**

- Scenario illustrating the design problem and how the pattern's class and object structures solve it.

## **Applicability**

- Situations in which the pattern can be applied, examples of poor designs it can address.



# Describing design patterns (2)

## Structure

- Graphical representation of classes (now UML), including diagrams illustrating collaboration among objects.

## Participants

- Participating classes/objects and their responsibilities.

## Collaborations

- How the participants collaborate to carry out their responsibilities.

## Consequences

- What are the trade-offs and results of using the pattern.
- What aspect of system lets you vary independently?

## ... and more ...

- Implementation, sample code, known uses, related patterns.

# A word of warning

## Common misconceptions

- Patterns will solve all the problems... No!
  - Patterns should not introduce unnecessary complexity.
  - Goal: make it easier to live with existing complexity.
- Patterns are good implementations... No!
  - Patterns are general solutions to design problems.
  - Important to understand context and forces at play.
  - Resolution: how and why the solution balances forces.
- More patterns lead to better design... No!
  - Context is extremely important, avoid overuse.

# The Observer pattern

## Keeping the objects in the know

- Notify objects about something they care about.
- Objects can decide if they want to be informed.
- One of the most heavily used patterns in JDK.
- Loosely coupled many-to-one relationship.

**This time, it all started with Weather-O-Rama...**

# Weather-O-Rama IWMS-ng

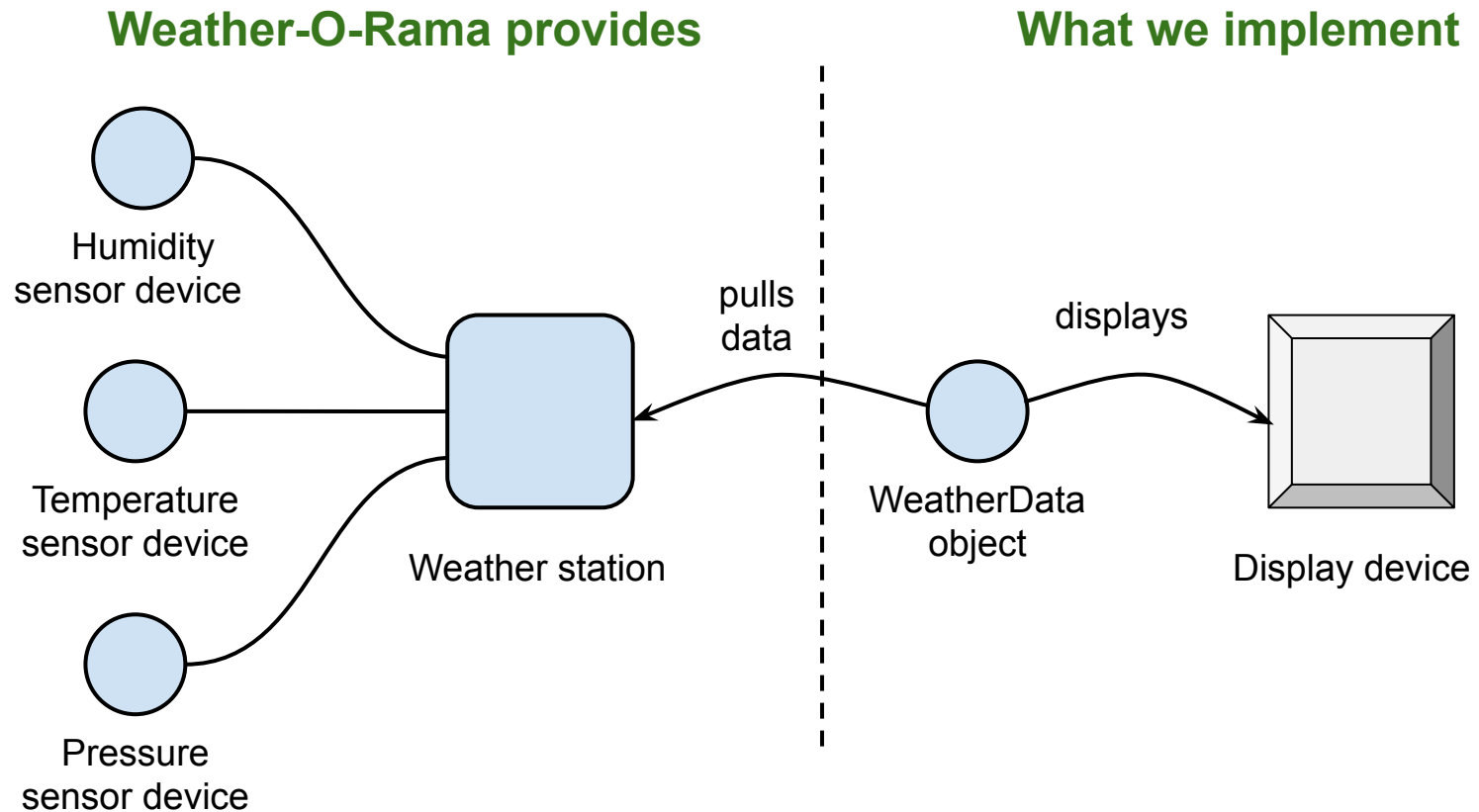
## Internet Weather Monitoring Station (next-gen)

- Need application with three display elements.
  - Current conditions, weather statistics, and a simple forecast, updated in real time.
- Expandable, needs API for developers
  - Allow custom weather displays.

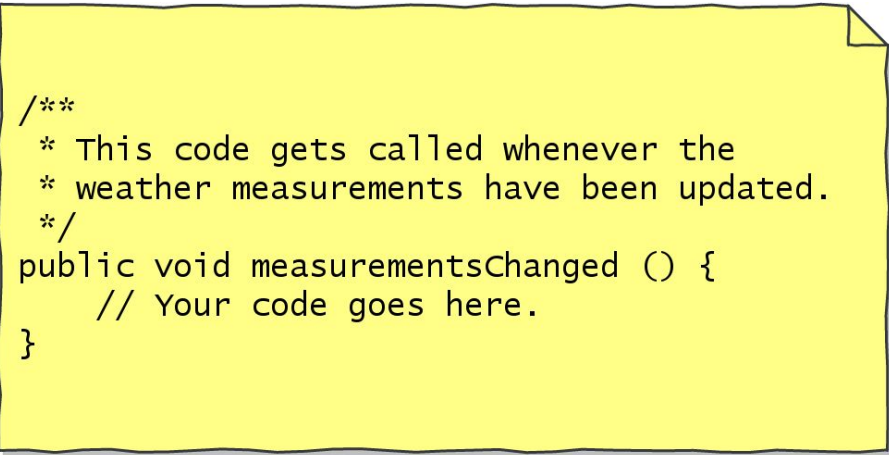
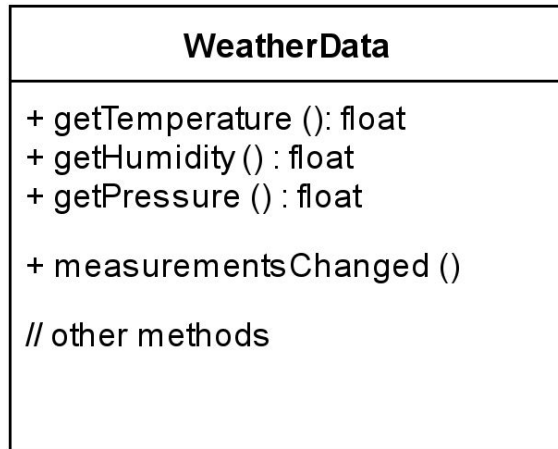
## Based on WeatherData object

- Tracks current weather conditions.
  - Temperature, humidity, barometric pressure.
- Can indicate when data changed.

# Weather monitoring overview



# The WeatherData class



```
/**  
 * This code gets called whenever the  
 * weather measurements have been updated.  
 */  
public void measurementsChanged () {  
    // Your code goes here.  
}
```

## Straightforward...

- Three getters for measurements.
  - WeatherData knows how to talk to the station.
- Developer left us a clue on what we need to add.
- Implement measurementsChanged() so that it updates three displays with current data.

# What do we know so far?

## Specification not entirely clear, but...

- WeatherData has getters for temperature, humidity, and barometric pressure.
  - We don't know or care how it gets data from the station.
- The `measurementsChanged()` method is called any time new measurement data is available.
  - We don't know or care how the method is called.
- Need to implement three display elements updated each time there is new data.
- The system must be expandable.
  - Developers can add/remove display elements.
  - We only know about the *initial* three displays.

# A quick stab at solution

## Adding code to measurementsChanged()

```
public class WeatherData {  
    // instance variable declarations  
  
    public void measurementsChanged() {  
        float temp = getTemperature();  
        float humidity = getHumidity();  
        float pressure = getPressure();  
  
        conditionsDisplay.update(temp, humidity, pressure);  
        statisticsDisplay.update(temp, humidity, pressure);  
        forecastDisplay.update(temp, humidity, pressure);  
    }  
  
    // other WeatherData methods  
}
```



# What is wrong with the code?

## Thinking about OO principles...

```
public class WeatherData {  
    // instance variable declarations  
  
    public void measurementsChanged() {  
        float temp = getTemperature();  
        float humidity = getHumidity();  
        float pressure = getPressure();  
  
        conditionsDisplay.update(temp, humidity, pressure);  
        statisticsDisplay.update(temp, humidity, pressure);  
        forecastDisplay.update(temp, humidity, pressure);  
    }  
  
    // other WeatherData methods  
}
```

Area of change!  
This needs to be encapsulated.

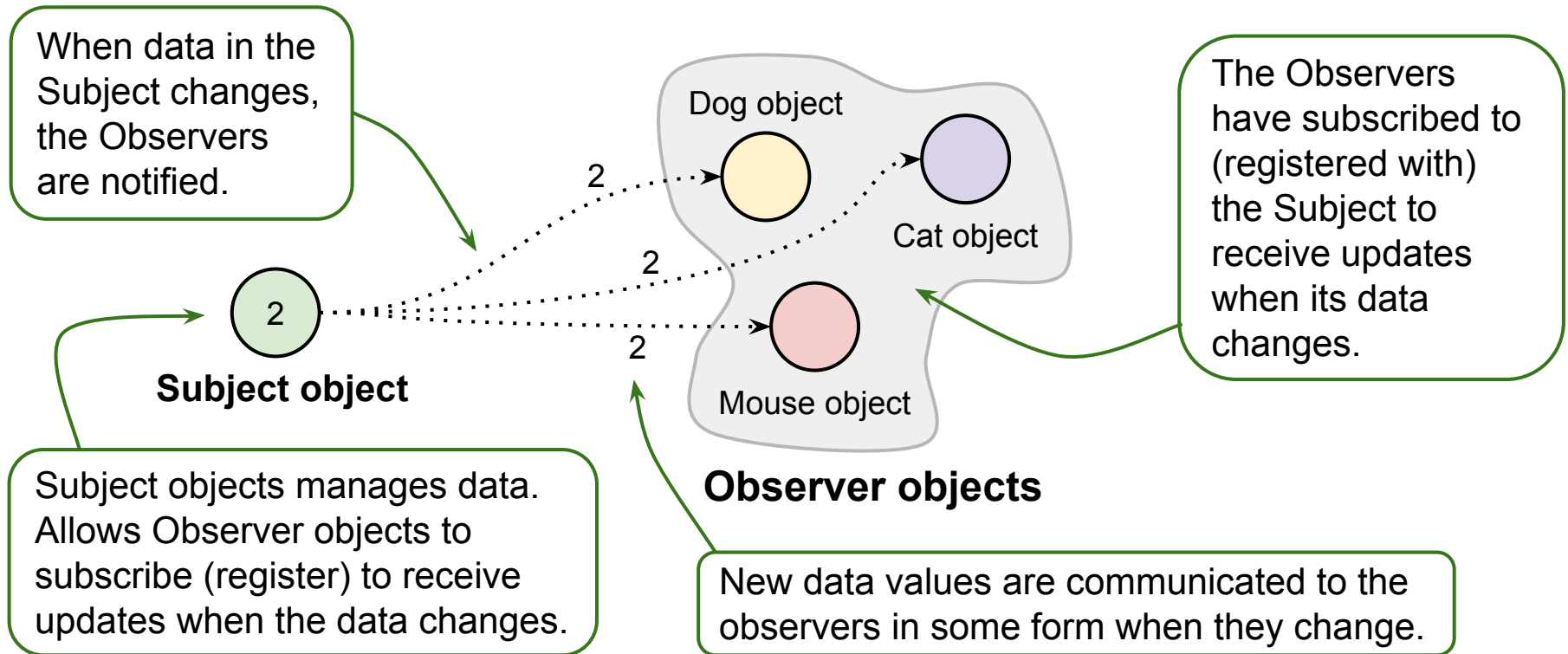
Uh oh! By coding to implementations, we have no way of adding or removing elements without changing WeatherData.

At least we are using common interface to talk to the display elements.

# Meet the Observer pattern

## Publishers + Subscribers = Observer Pattern

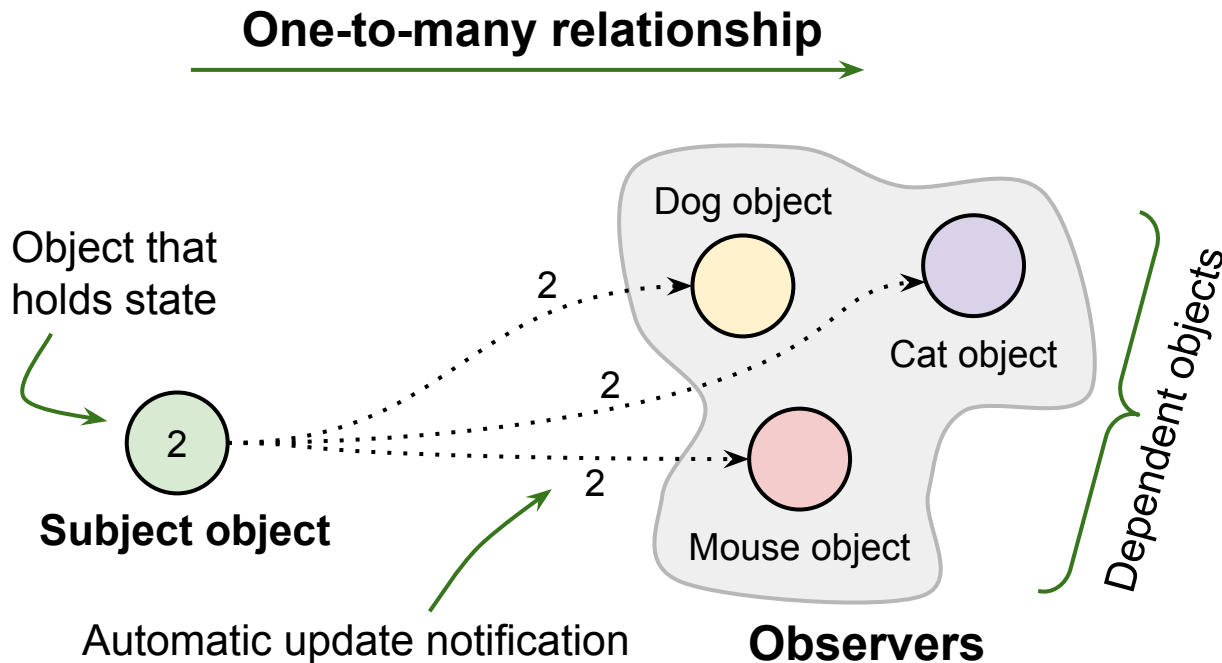
- Just like newspaper, except we call the publisher and the subscribers differently:
  - Publisher = Subject, Subscribers = Observers



# The Observer pattern (1)

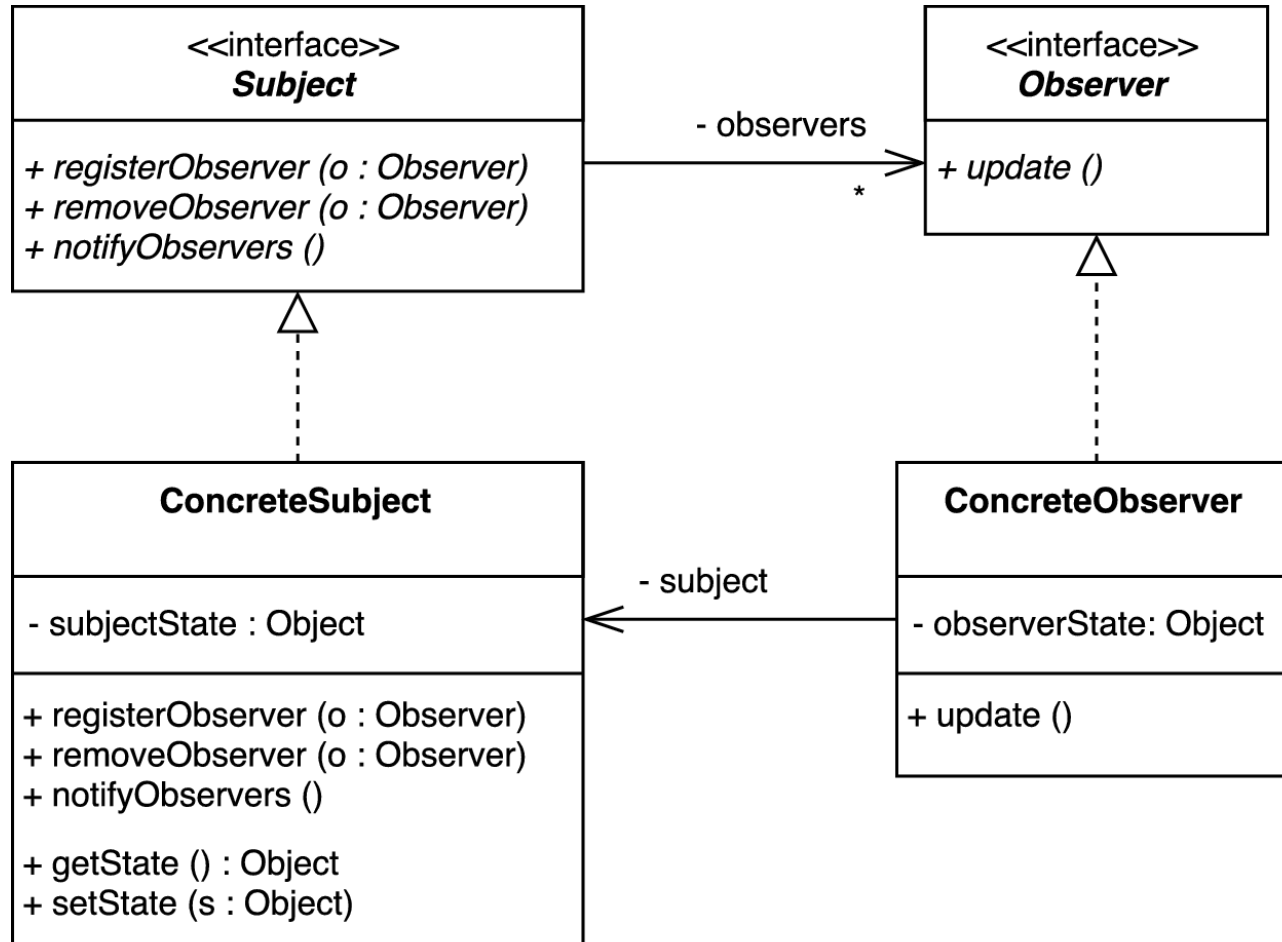
## Intent

- Defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.



# The Observer pattern (2)

## Structure



# The power of loose coupling

## Interaction without entanglement

- Loosely coupled objects can interact, but have very little knowledge of each other.
- Observer pattern provides design where subjects and observers are loosely coupled.
- Subject only knows that an observer implements a certain interface.
- New observers can be added at any time.
- No need to modify subject to add a new type of observers.
- Subjects and observers can be reused independently of each other.
- Changes to either the subject or an observer will not affect the other.

# Design principle

**Strive for loosely coupled designs between objects that interact**

- Minimizes interdependency between objects.

**The result**

- More flexible design that is resilient to change.

# Designing the weather station (1)

## The “one”

- WeatherData
- Contains state that changes

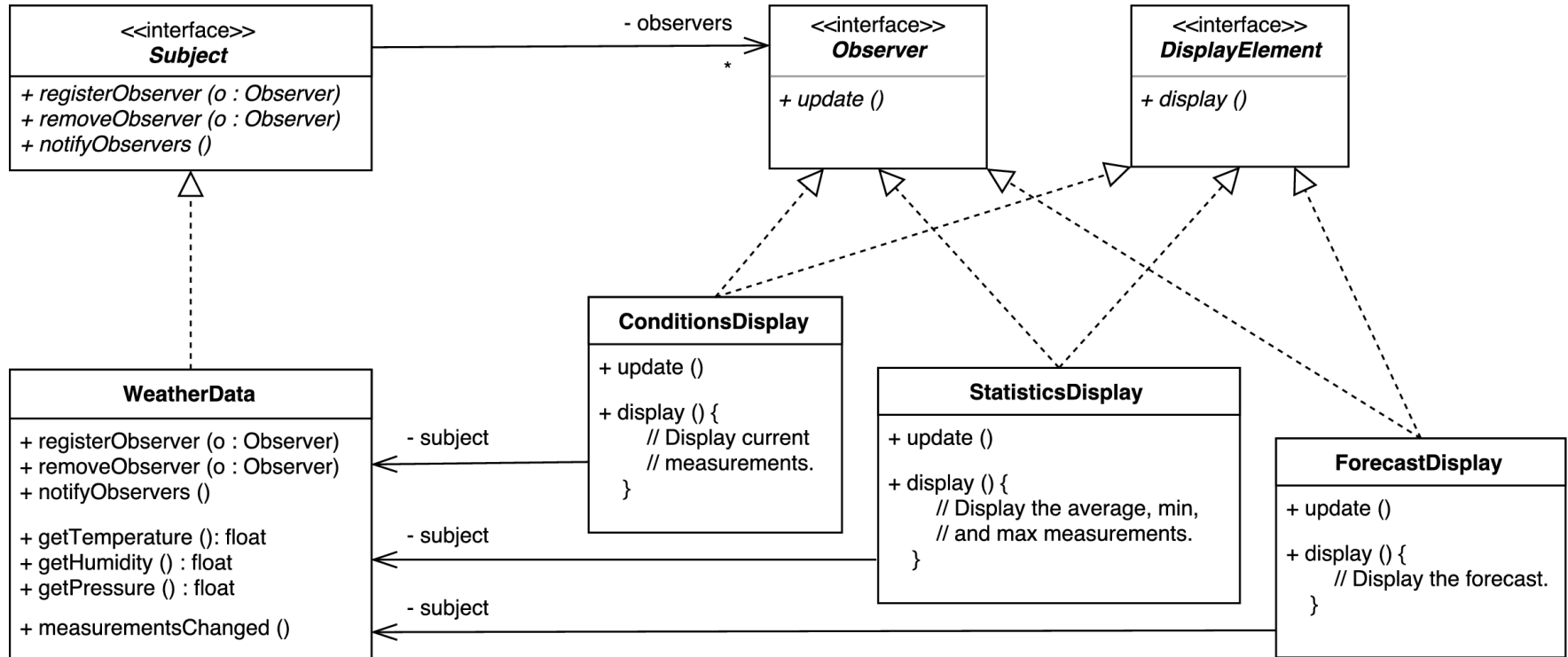
## The “many”

- Display elements using the measurements.
  - ConditionsDisplay
  - StatisticsDisplay
  - ForecastDisplay
  - ThirdPartyDisplay
- Need to be updated when weather data changes.

## Getting the data to the display elements

- Common interface used by WeatherData.

# Designing the weather station (2)





# How to get state to the observer?

## **Subject *pushes* state data to the observer**

- Subject informs observer of change together with the new state information.
  - Observer does not need to query subject.
- Signature of the update method is potentially fragile.
  - How can we make it general?

## **Observer *pulls* state data from the subject**

- Subject informs observer of change.
- Observer queries subject for state information.
  - Observer queries only the information it needs.
  - More flexible in case of a complex subject.
- Observer needs to know the concrete subject.

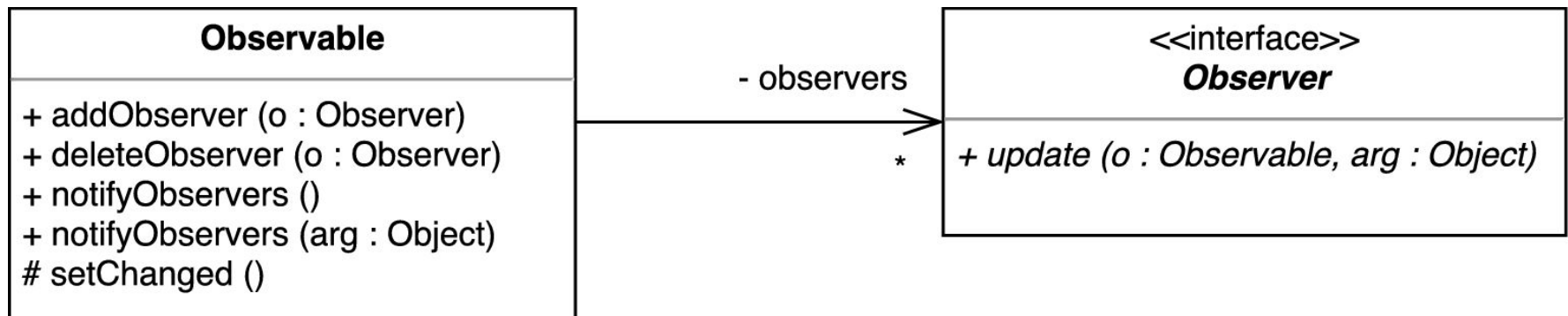
# Observer pattern in Java

## Applied in many places

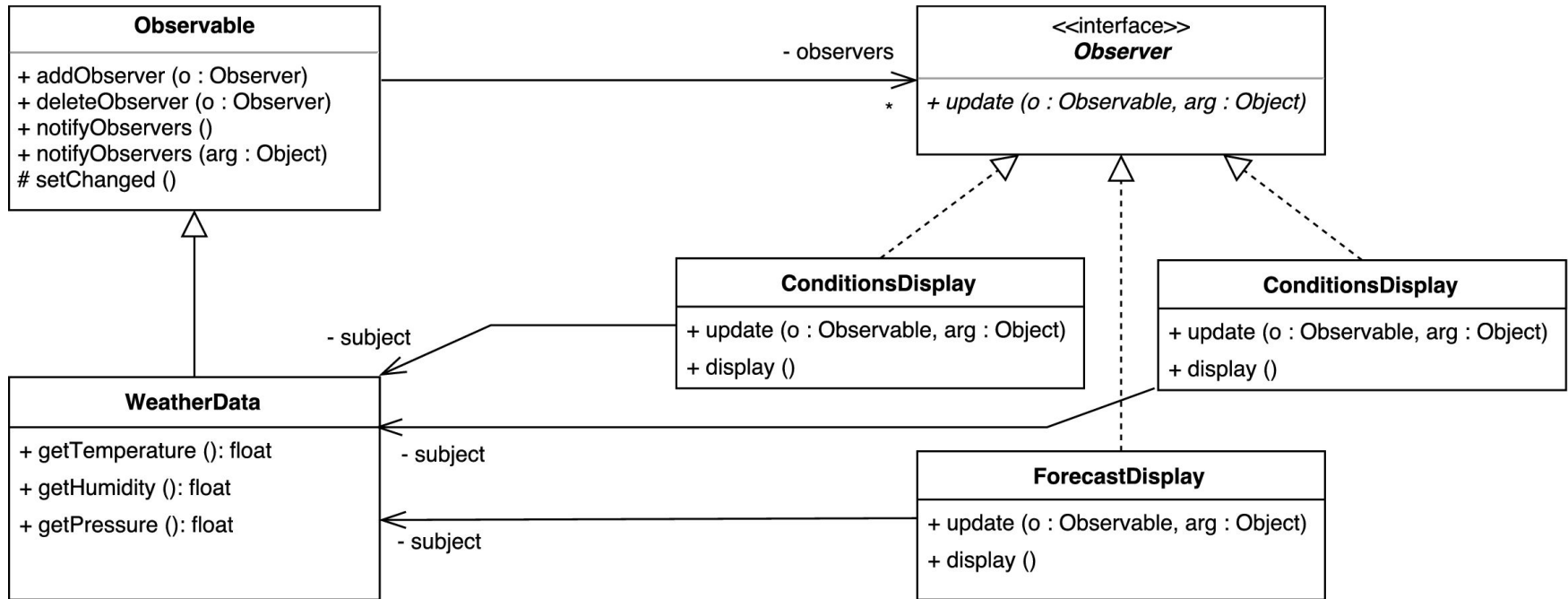
- Swing UI framework, JavaBeans, etc.
- Observers often called listeners.

## General Observer pattern in `java.util`

- Subject is called `Observable`.
- Supports push/pull communication.
- Notifications sent only if `Observable` changed.



# Using Java's Observer pattern



## Pull model

- Cast `Observable` argument to subject type and query.

## Push model (encapsulated state)

- Cast `Object` argument to the type of the type.

# Remarks on Java's Observer

## **Observable is a class, not an interface**

- Implementation basis rather than abstraction.
- But required by the Observer interface.
- Swapping implementations not possible.

## **Subject class must extend Observable**

- Observable behavior cannot be added to a class that already extends another class.
- The `setChanged()` method is protected.
  - Something needs to subclass `Observable`, if only to make `setChanged()` visible to enable composition.

**NB: never depend on notification ordering!**

# Principles found in Observer

## Encapsulate what changes

- What changes is the state of the subject and the number and type of observers.
- We can vary the objects that depend on the state of the subject without changing the subject.

## Program to interface, not implementation

- Subject knows about dependent objects only through the Observer interface.
- Observers register using Subject interface.

## Favor composition over inheritance

- Relationship between Subject and Observers is set up at runtime through composition, not inheritance.

# The Decorator pattern

## Decorating objects at runtime

- Giving objects new responsibilities at runtime without changing the underlying classes.
- Using composition to avoid class hierarchy explosion due to overuse of inheritance.

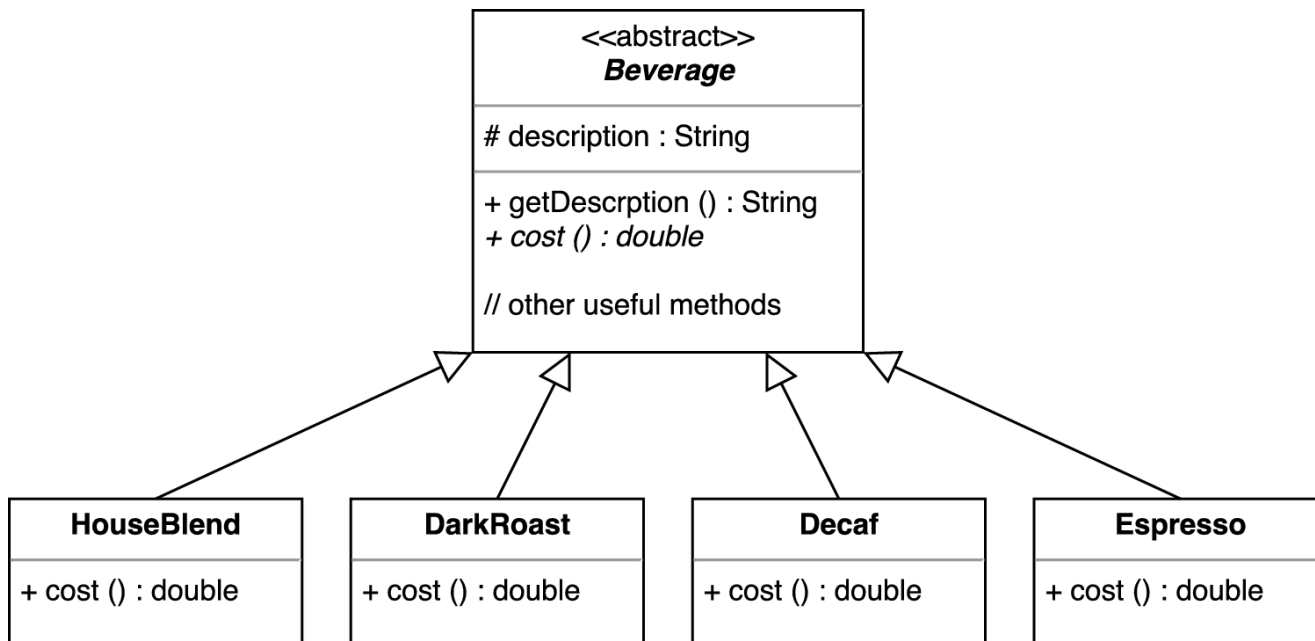
**Let's have some coffee first...**

# Welcome to Starbuzz Coffee

## Quickly expanding beverage offering

- Ordering system needs to match it.
- Frequent updates are needed.

## Straightforward initial design



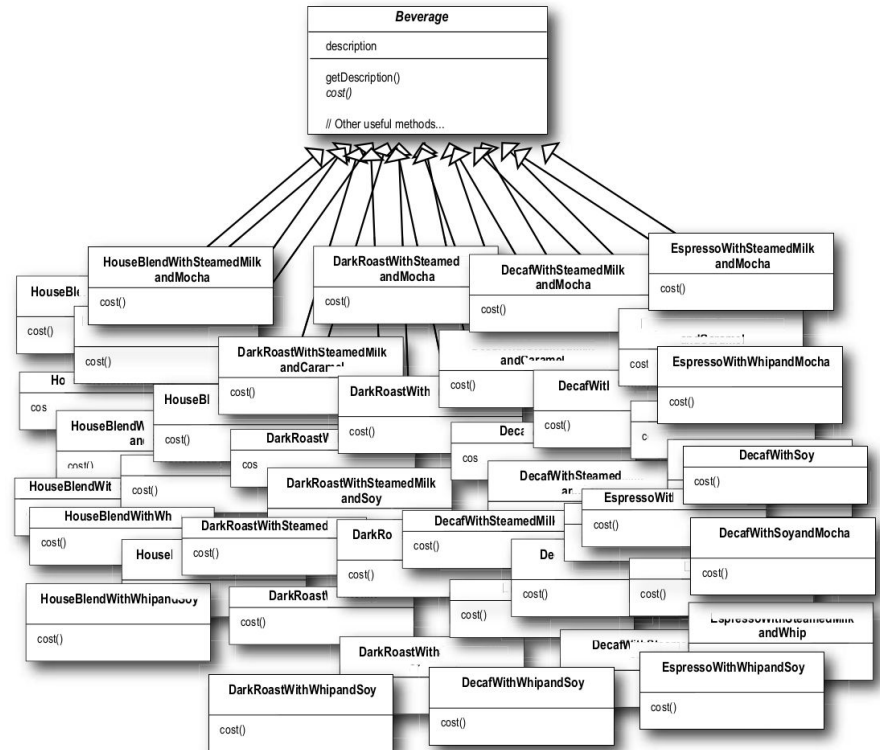
# What about few condiments?

## Starbuzz charges for each condiment

- Needs to be supported by the ordering system.
- Steamed milk, soy, mocha, whipped milk...



**The class hierarchy  
just exploded!**





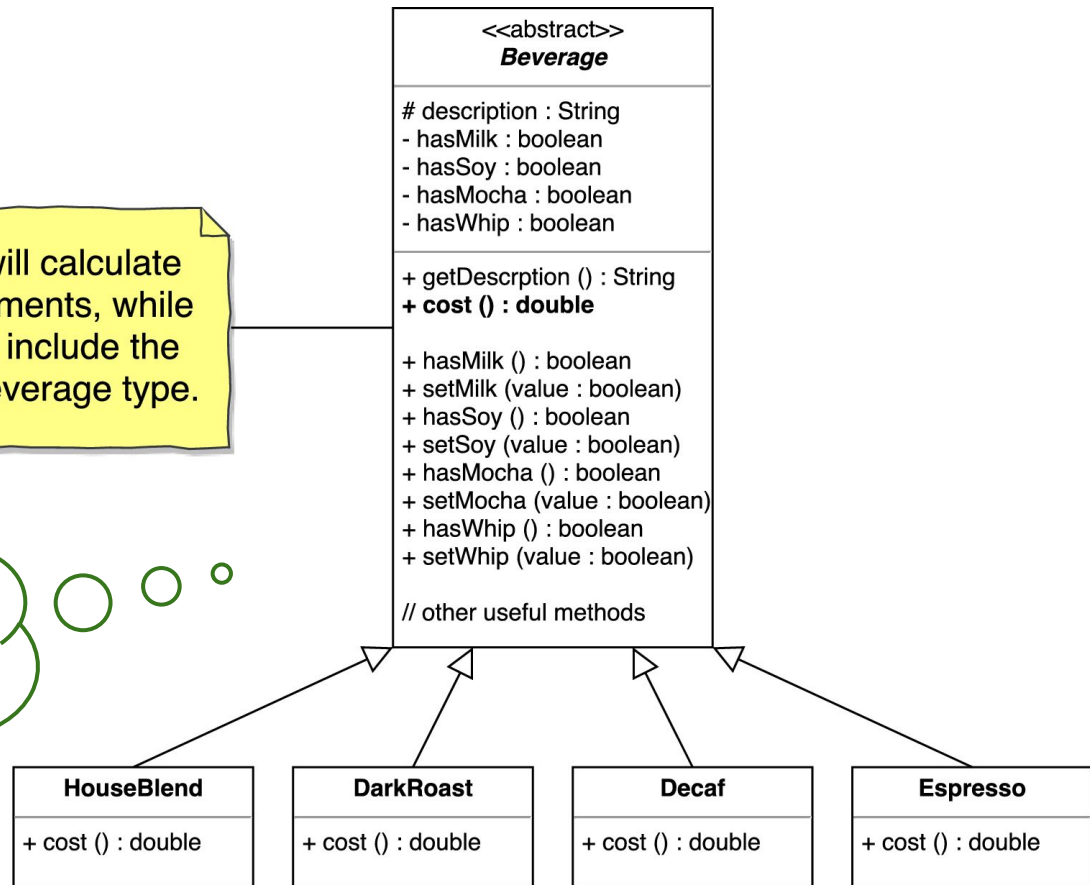
# Why all the classes?

## Just use instance variables

- Calculate the cost of condiments in one place.

The superclass `cost()` will calculate the cost of all the condiments, while the overridden `cost()` will include the costs for the specific beverage type.

Back to five classes!



# Is it a good idea?

Remember the ducks!

## What changes will impact this design?

- Price changes for condiments will force us to alter existing code.
- New condiments will force us to add new fields and methods, and to alter the `cost()` method in the superclass.
- The condiments may not be appropriate for some of the beverages.
- What if a customer wants double mocha?

**Avoid encoding dynamic properties into type!**

# Design principle

## Classes should be open for extension, but closed for modification

- *The Open/Closed Principle.*
- Allow classes to be easily extended to incorporate new behavior without modifying existing code.

## The result

- Design that is resilient to change and flexible enough to take on new functionality to meet changing requirements.

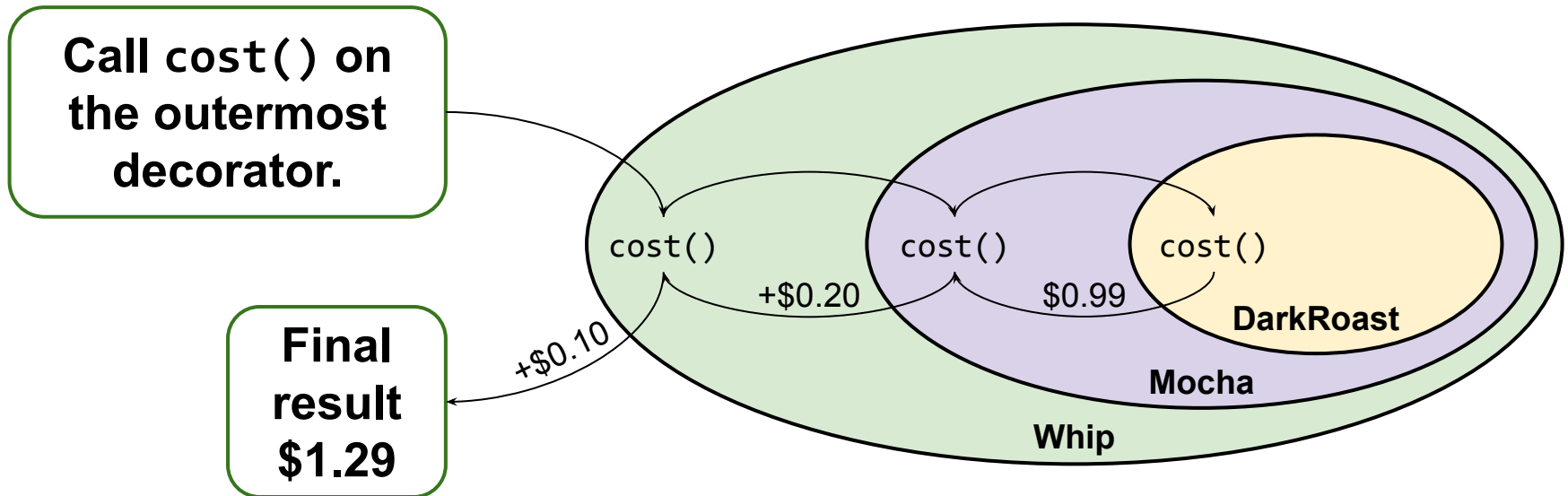
# Meet the Decorator pattern

## Decorate beverage with condiments

- Rely on delegation to add on the condiment costs.

## Dark Roast with Mocha and Whip?

- Start with DarkRoast object...



# Decorator characteristics

## What we know so far...

- Decorators have the same supertype as the objects they decorate.
- Decorated objects can be passed around in place of the original objects.
- Object can be wrapped by one or more decorators.
- Decorator adds its behavior either before and/or after delegating to the object it decorates.
- Objects can be decorated at any time.

# The Decorator pattern (1)

## Intent

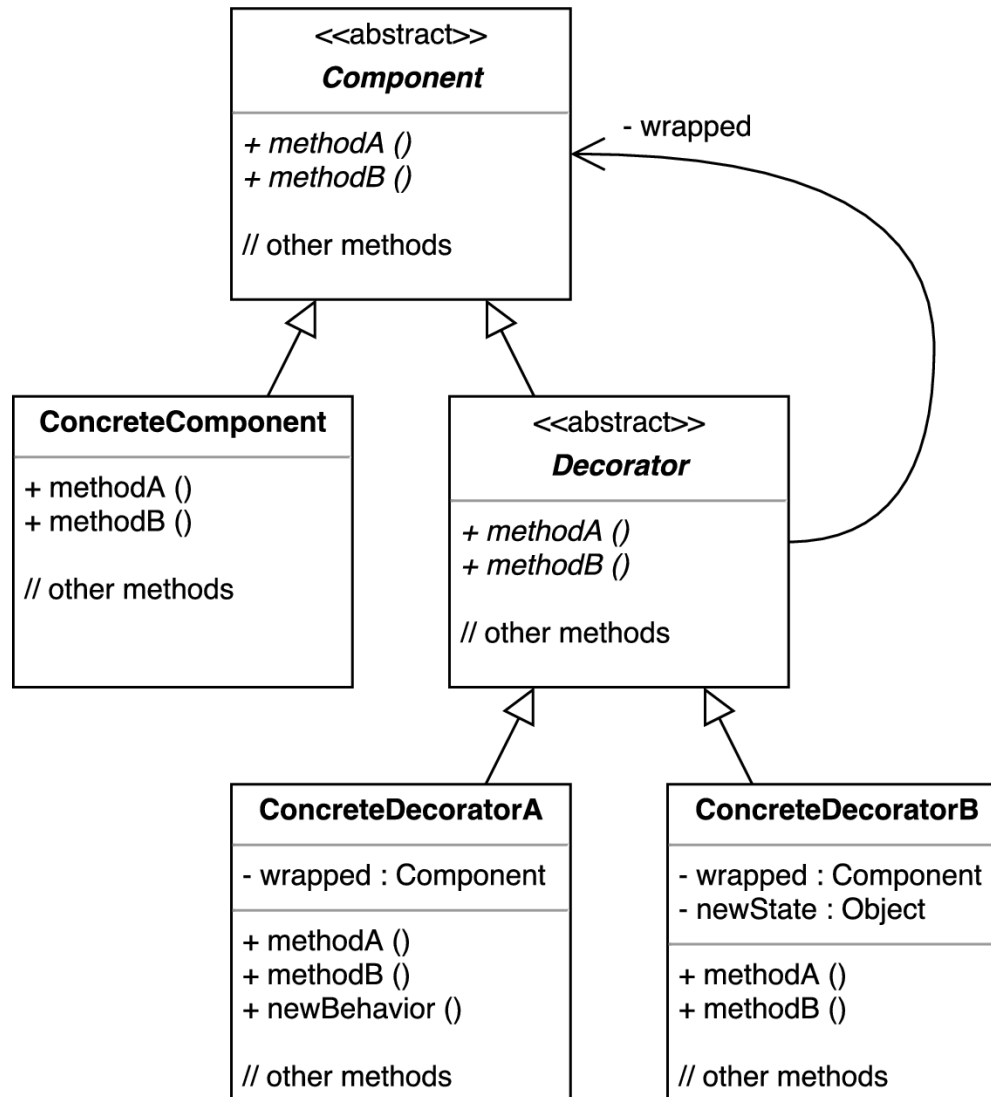
- Attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

## Benefits

- Add/remove responsibilities at runtime.
- Add properties multiple times.
- Pay-as-you go approach to adding responsibilities.
  - Avoids feature-laden classes high up in the hierarchy.

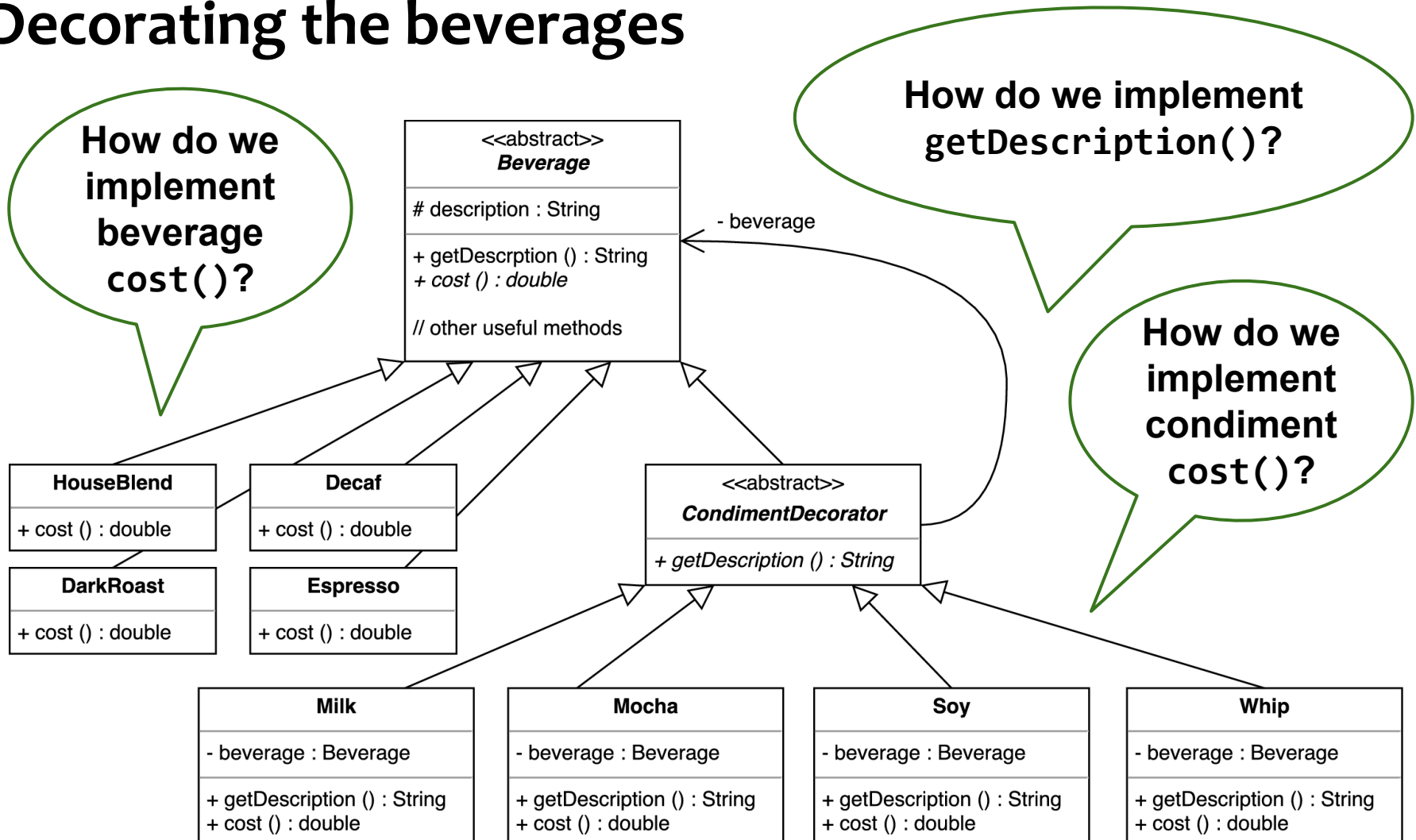
# The Decorator pattern (2)

## Structure



# Back to Starbuzz Coffee

## Decorating the beverages





# Decorator drawbacks

## **Lots of small objects that look alike**

- Differences mainly in the “wiring” of objects.
- Easy to customize, difficult to understand.

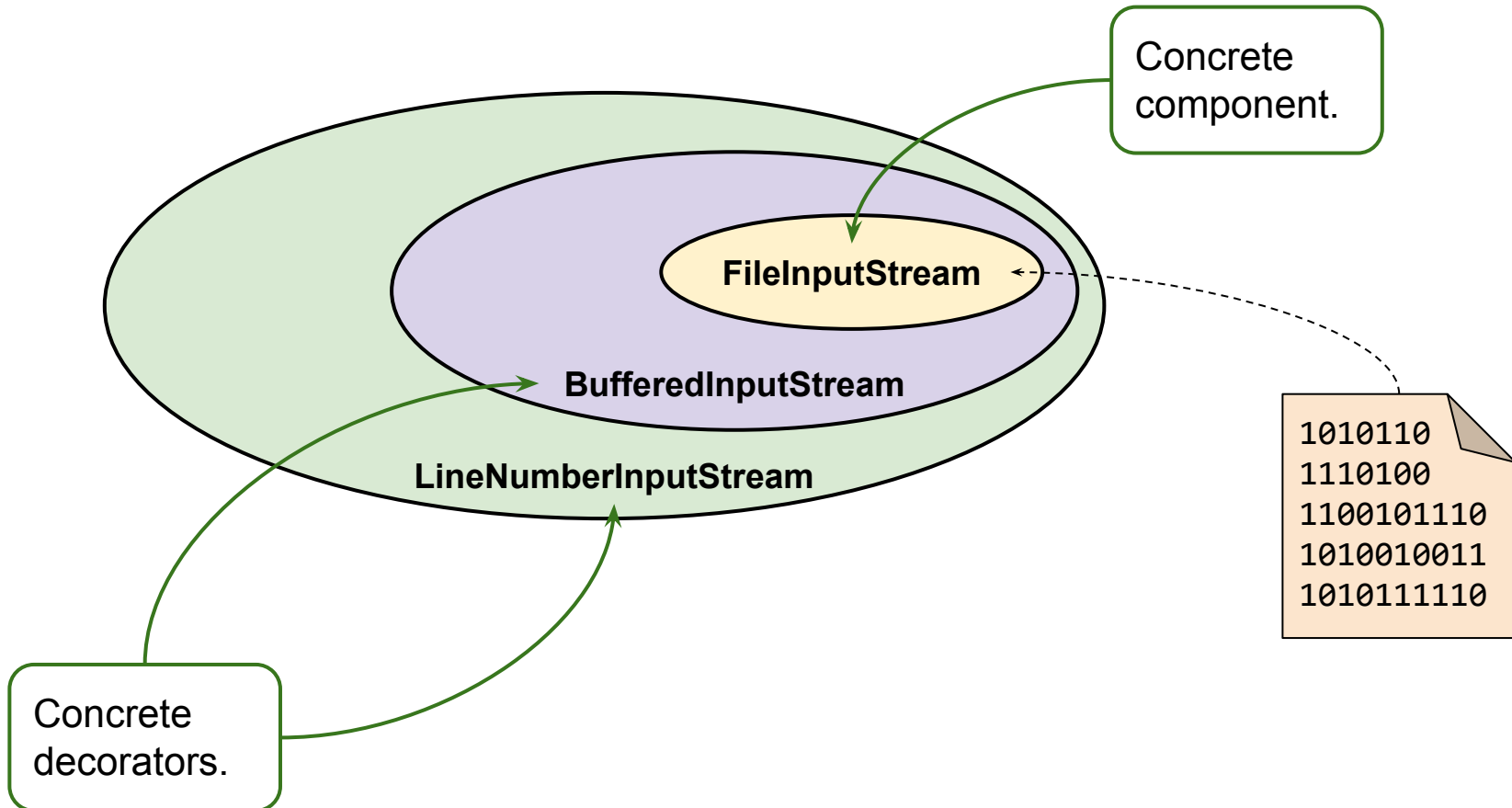
## **Decorated component != component**

- A decorator should act as a transparent enclosure, but the object identities are different.
  - Avoid relying on object identity when using decorators.
- Identities of intermediate decorators can leak.
  - Managing more objects may increase chance of errors.
  - Decorated objects often created using Factory.

# Real-world decorators (1)

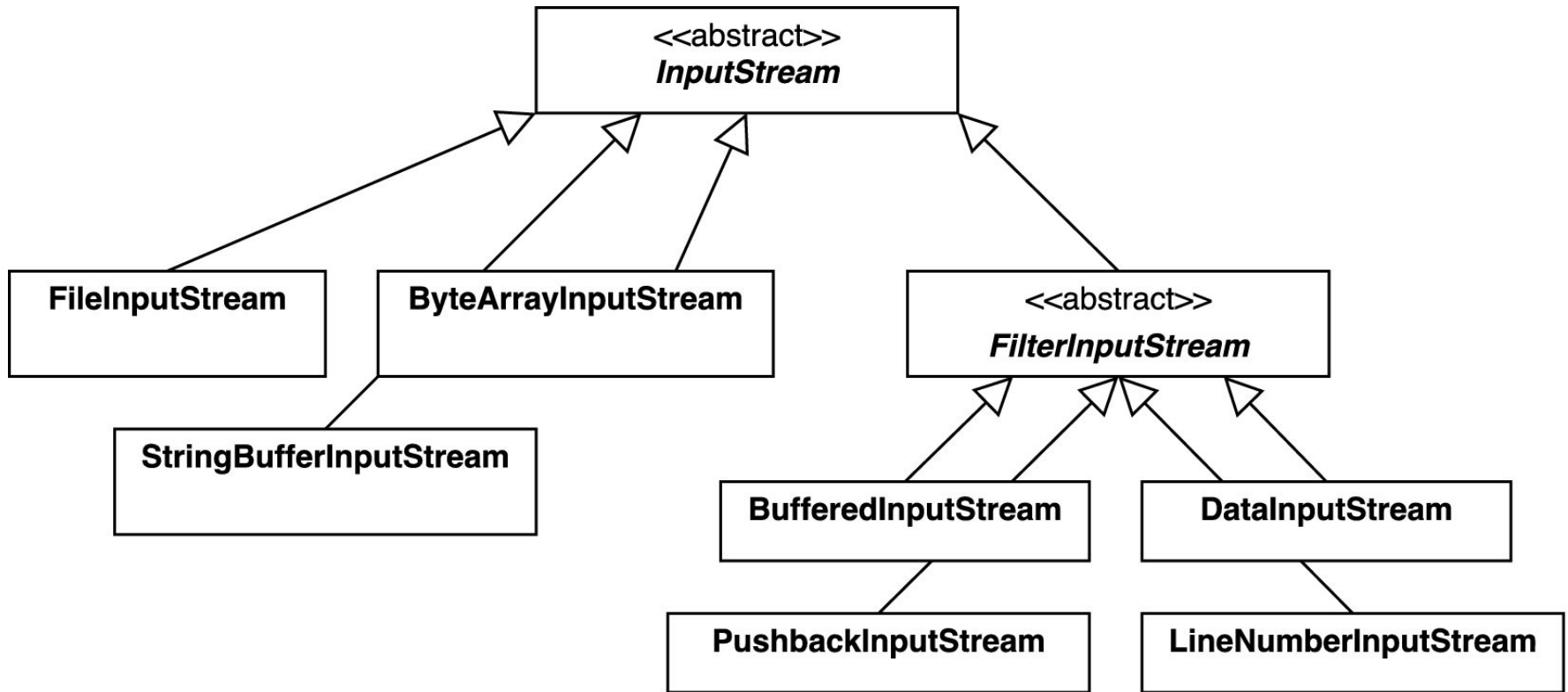
## Java I/O

- Input/output streams, reader/writer streams.



# Real-world decorators (2)

## Decorating the java.io classes



# The Factory pattern

## There is more to creating objects

- The **new** operator is not the only way.
- Instantiation should not always be done in the public, as it can lead to coupling problems.
- Object creation may benefit from encapsulation.

# Thinking about “new”

## Seeing “new” means “concrete”

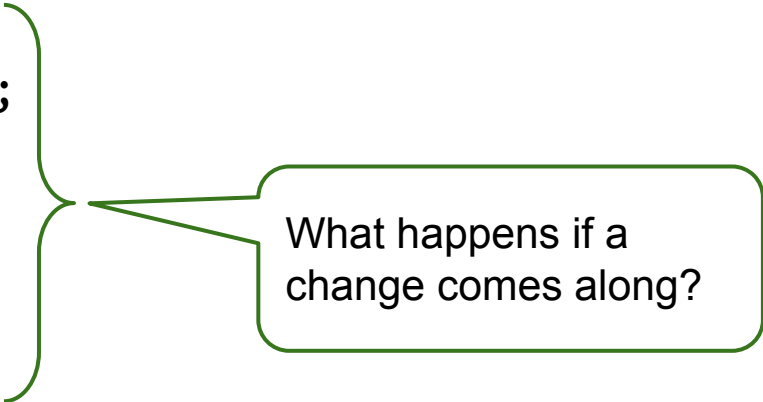
- Even when using abstract supertype in variable and field declarations.

```
Duck duck = new MallardDuck();
```

- The decision which class to instantiate is sometimes made at runtime.

```
Duck duck;
```

```
if (picnic) {  
    duck = new MallardDuck();  
} else if (hunting) {  
    duck = new DecoyDuck();  
} else if (inBathTub) {  
    duck = new RubberDuck();  
}
```



What happens if a change comes along?

# Thinking about “new”

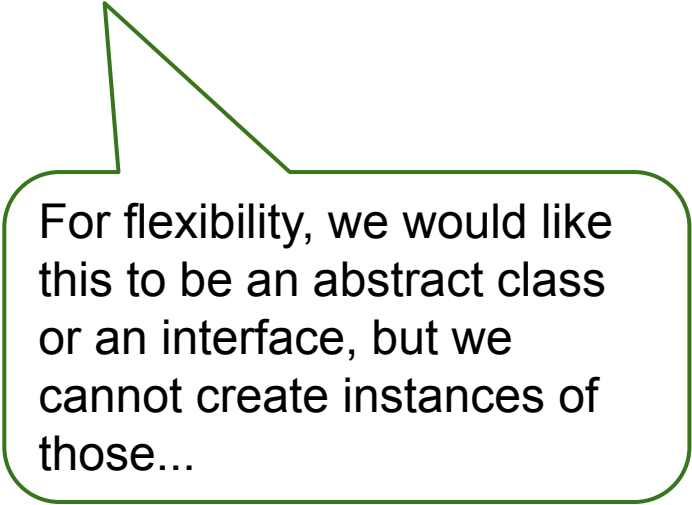
## What is wrong with “new”?

- Technically nothing.
  - We do need a way to create class instances.
- Recall: *program to an interface*.
  - Shields us from changes in the implementation details.
- Using concrete classes in part of a program may require that part to change when adding new classes.
  - Code will not be “closed for modification”.
- Recall: *encapsulate what changes*.
  - Even if it concerns object creation.

# Identify what varies (1)

## A bit of code from pizza store

```
Pizza orderPizza() {  
    Pizza pizza = new Pizza();  
    pizza.prepare();  
    pizza.bake();  
    pizza.cut();  
    pizza.box();  
    return pizza;  
}
```



For flexibility, we would like this to be an abstract class or an interface, but we cannot create instances of those...

# Identify what varies (2)

## We need more than one kind pizza

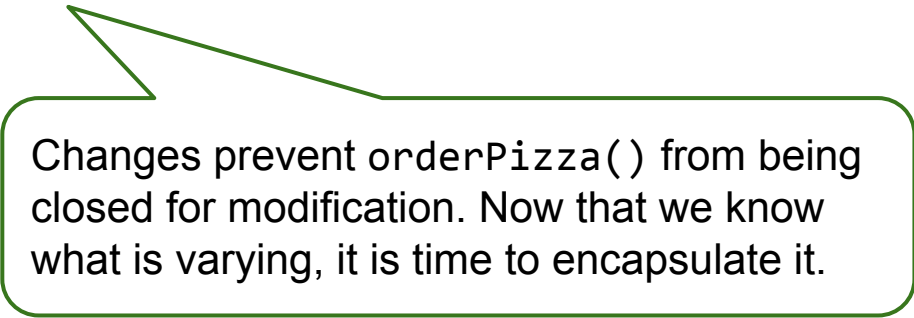
```
Pizza orderPizza(String type) {  
    Pizza pizza;  
  
    if (type.equals ("cheese")) {  
        pizza = new CheesePizza();  
    } else if (type.equals ("greek")) {  
        pizza = new GreekPizza();  
    } else {  
        pizza = new MargheritaPizza();  
    }  
  
    pizza.prepare();  
    pizza.bake();  
    pizza.cut();  
    pizza.box();  
    return pizza;  
}
```



# Identify what varies (3)

## There is pressure to change kinds

```
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 ("clam")) {  
        pizza = new ClamPizza();  
    } else {  
        pizza = new MargheritaPizza();  
    }  
  
    pizza.prepare();  
    pizza.bake();  
    pizza.cut();  
    pizza.box();  
    return pizza;  
}
```

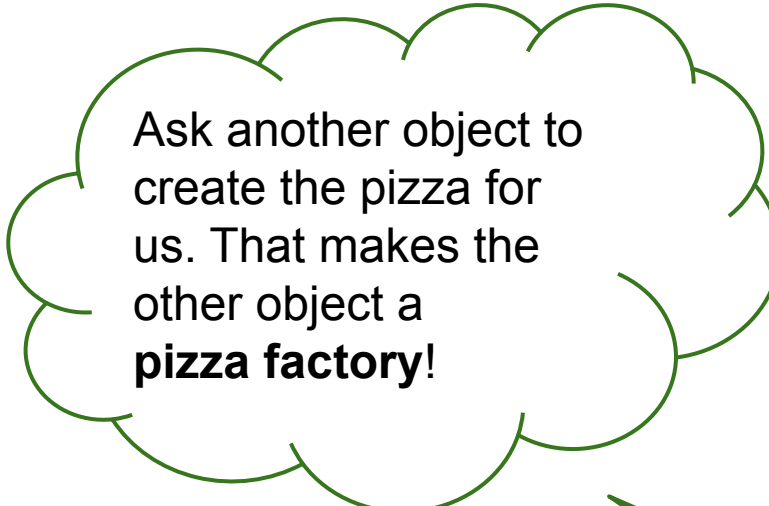


Changes prevent orderPizza() from being closed for modification. Now that we know what is varying, it is time to encapsulate it.

# Encapsulating object creation (1)

## Delegate object creation to another object

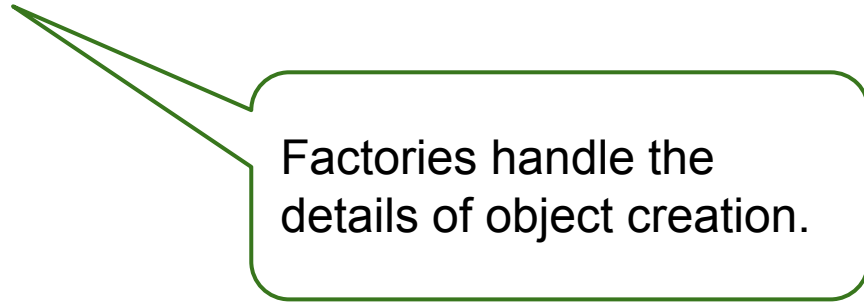
```
Pizza orderPizza(String type) {  
    Pizza pizza;
```



Ask another object to  
create the pizza for  
us. That makes the  
other object a  
**pizza factory!**

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

```
}
```



Factories handle the  
details of object creation.

# Encapsulating object creation (2)

## Building a simple pizza factory

```
public class SimplePizzaFactory {  
    public Pizza createPizza (String type) {  
        if (type.equals ("cheese")) {  
            return new CheesePizza();  
        } else if (type.equals ("pepperoni")) {  
            return new PepperoniPizza();  
        } else if (type.equals ("clam")) {  
            return new ClamPizza();  
        } else if (type.equals ("veggie")) {  
            return new VeggiePizza();  
        } else {  
            // Alternatively, throw an exception here.  
            return new MargheritaPizza();  
        }  
    }  
}
```

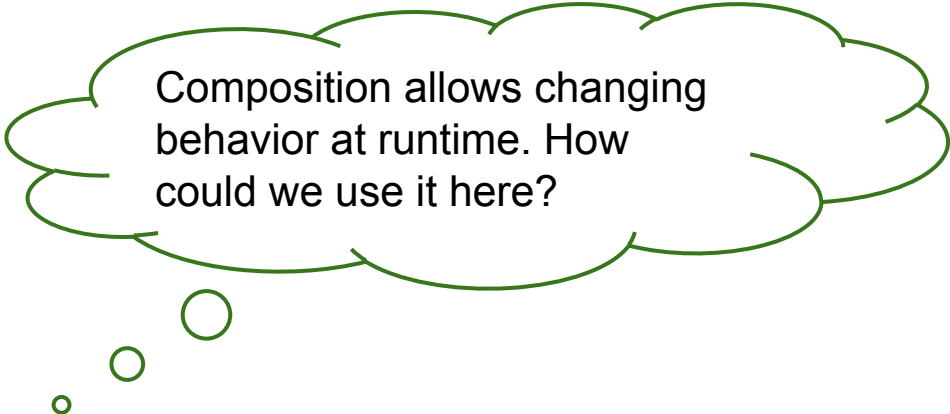
The factory may have multiple clients. Now we only need to make changes in one place.

Did not we just push the problem to another object?

# Encapsulating object creation (3)

## Reworking the PizzaStore class

```
public class PizzaStore {  
    SimplePizzaFactory factory;  
  
    public PizzaStore (SimplePizzaFactory factory) {  
        this.factory = factory;  
    }  
  
    public Pizza orderPizza (String type) {  
        Pizza pizza = factory.createPizza (type);  
        pizza.prepare();  
        pizza.bake();  
        pizza.cut();  
        pizza.box();  
        return pizza;  
    }  
}
```

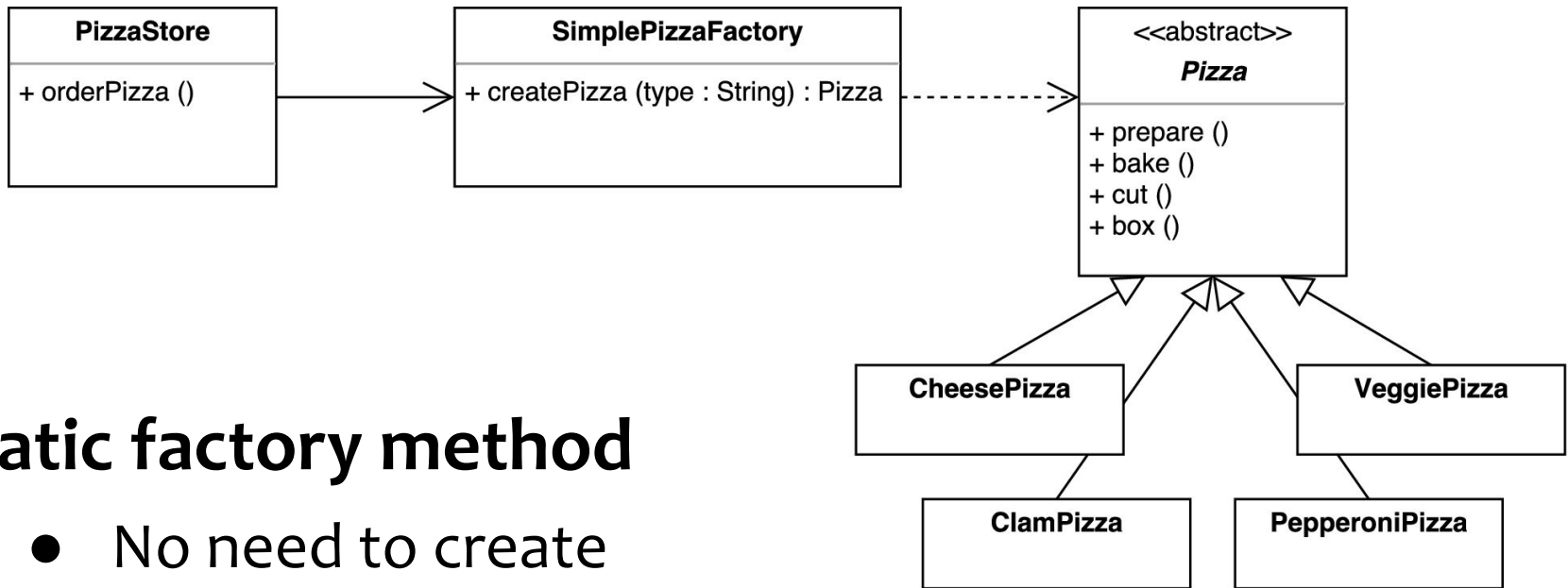


Composition allows changing behavior at runtime. How could we use it here?

# The Simple Factory defined

## Not really a pattern (yet)

- More of a programming idiom.



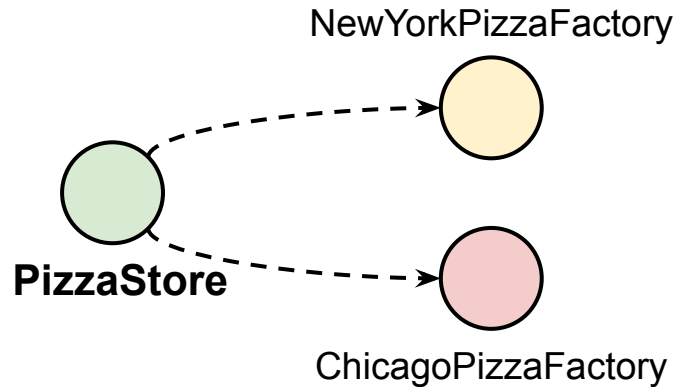
## Static factory method

- No need to create instance of the factory.
- Not possible to plug in *different creation behavior*.

# Franchising the pizza store (1)

Use time-tested pizza-store code...

... but support regional differences!



## Possible approach

- Move code from SimplePizzaFactory to another class and make SimplePizzaFactory abstract.

# Franchising the pizza store (2)

## Using different factories in different stores

```
NYPizzaFactory nyFactory = new NYPizzaFactory ();  
PizzaStore nyStore = new PizzaStore (nyFactory);  
nyStore.orderPizza ("veggie");
```

```
ChicagoPizzaFactory chicagoFactory = new ChicagoPizzaFactory ();  
PizzaStore chicagoStore = new PizzaStore (chicagoFactory);  
chicagoStore.orderPizza ("veggie");
```

## But there are differences in preparation!

- At the moment, we have a unified store receiving region-specific pre-made pizzas.
- We want a region-specific store to be responsible for the pizza preparation.
- We need a framework that ties store and pizza creation, but allows things to remain flexible.

# Franchising the pizza store (3)

## A framework for the pizza store

- The createPizza() factory method needs to go back to the PizzaStore, but as an *abstract method*.

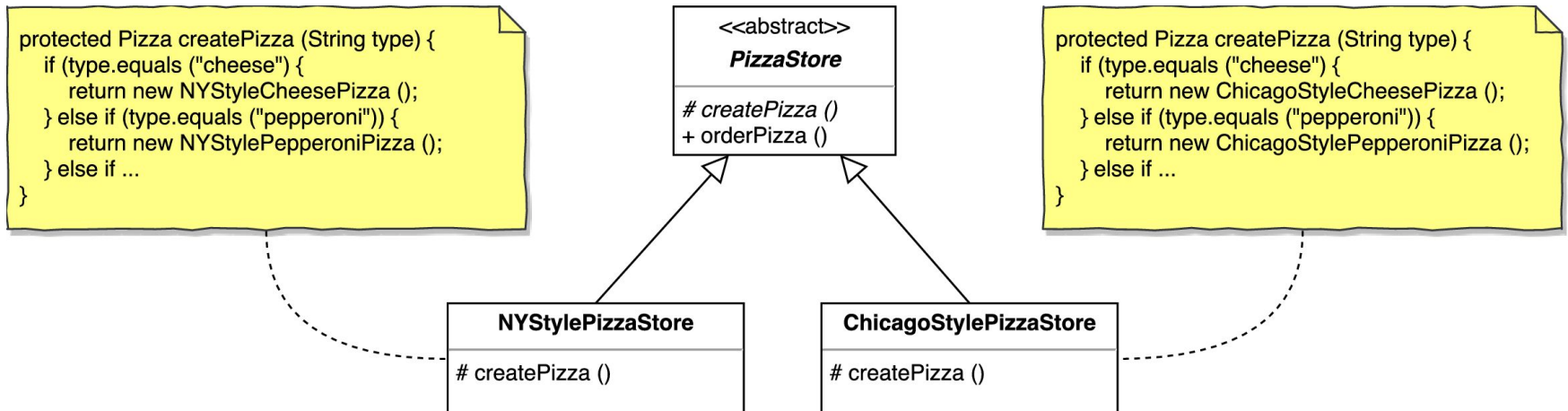
```
public abstract class PizzaStore {  
    public Pizza orderPizza (String type) {  
        Pizza pizza = createPizza (type);  
  
        pizza.prepare ();  
        pizza.bake ();  
        pizza.cut ();  
        pizza.box ();  
        return pizza;  
    }  
  
    protected abstract Pizza createPizza (String type);  
}
```



# Franchising the pizza store (4)

## Allowing the subclasses to decide...

- By choosing a specific subclass of PizzaStore, we choose how pizzas are made.
  - The orderPizza() method just uses a factory method which the subclasses MUST implement.
  - Instead of an object handling instantiation of concrete classes, subclasses now take on the responsibility.



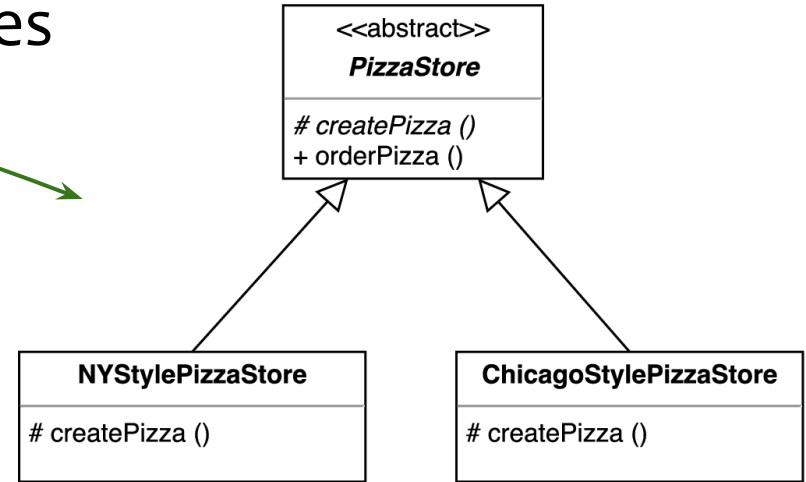
# Meet the Factory Method pattern

## Subclasses decide what objects to create

- Involves two kinds of classes

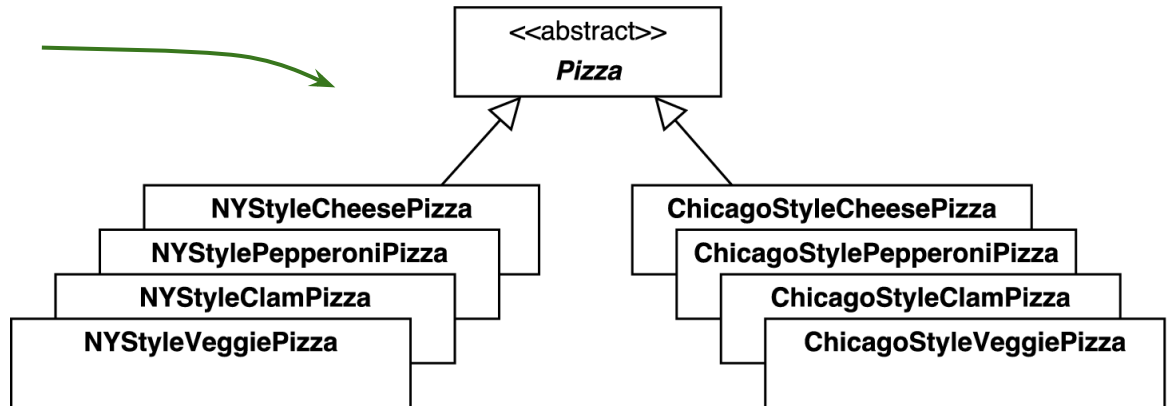
### Creator classes

- Abstract creator defines abstract factory method
- Concrete creators produce products



### Product classes

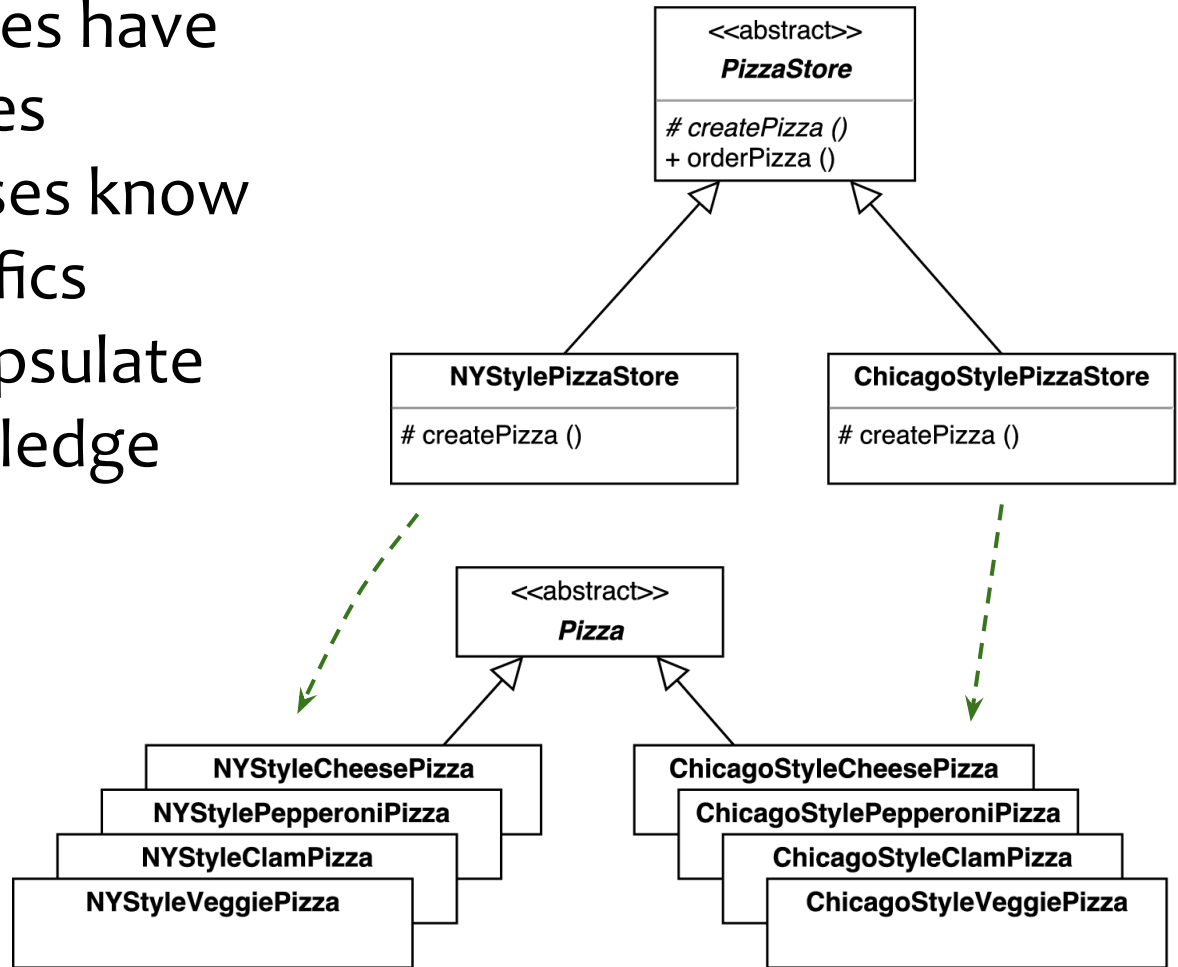
- Abstract and concrete products



# Another perspective

## Parallel class hierarchies

- Both hierarchies have abstract classes
- Concrete classes know regional specifics
- Creators encapsulate product knowledge



# The Factory Method pattern (1)

## Intent

- Defines an interface for creating an object, but lets subclasses decide which class to instantiate.  
Factory Method lets a class defer instantiation to subclasses.

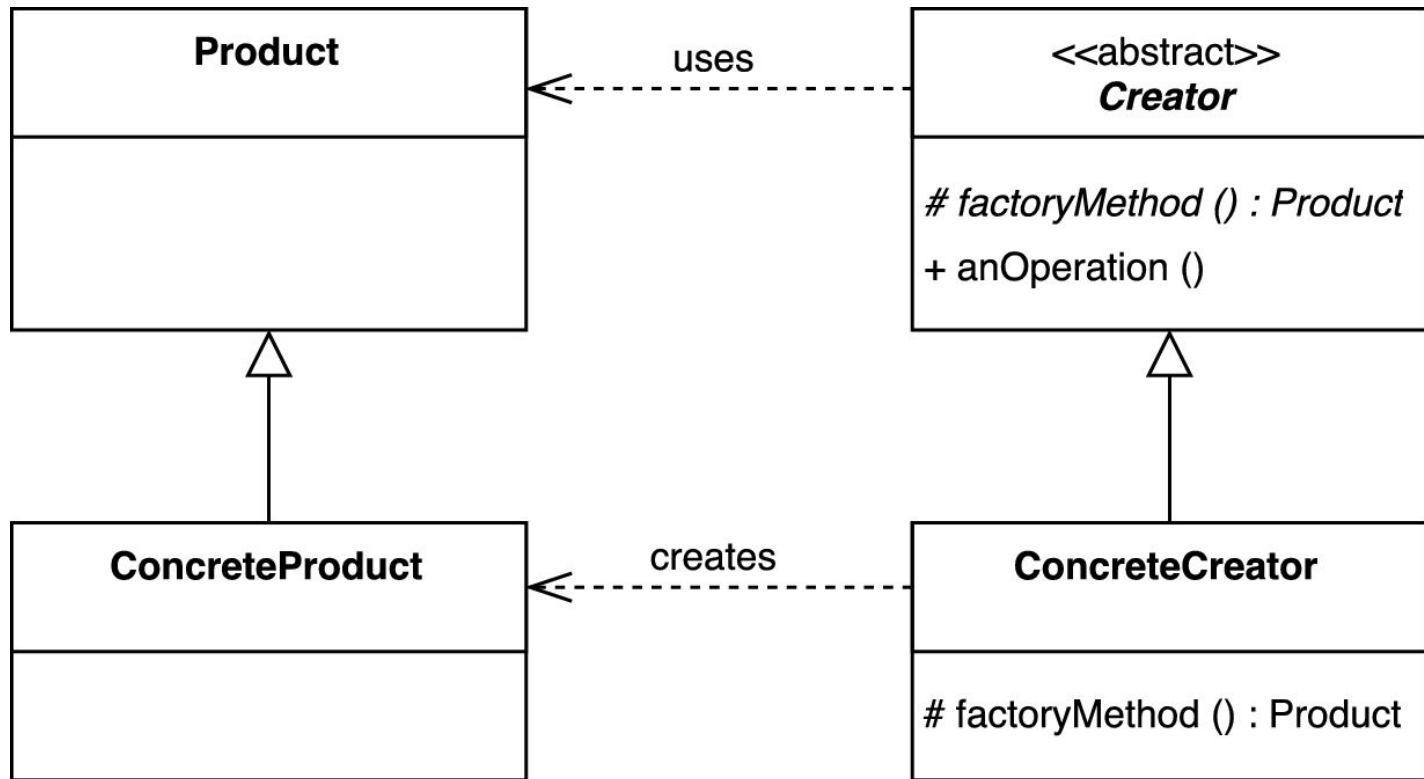
## Benefits

- Encapsulates object creation.
- Decouples product implementation from its use.
- Avoids client dependency on concrete classes.

# The Factory Method pattern (2)

## Structure

Manipulates products, but does not create them.



Produces products, but does not manipulate them.

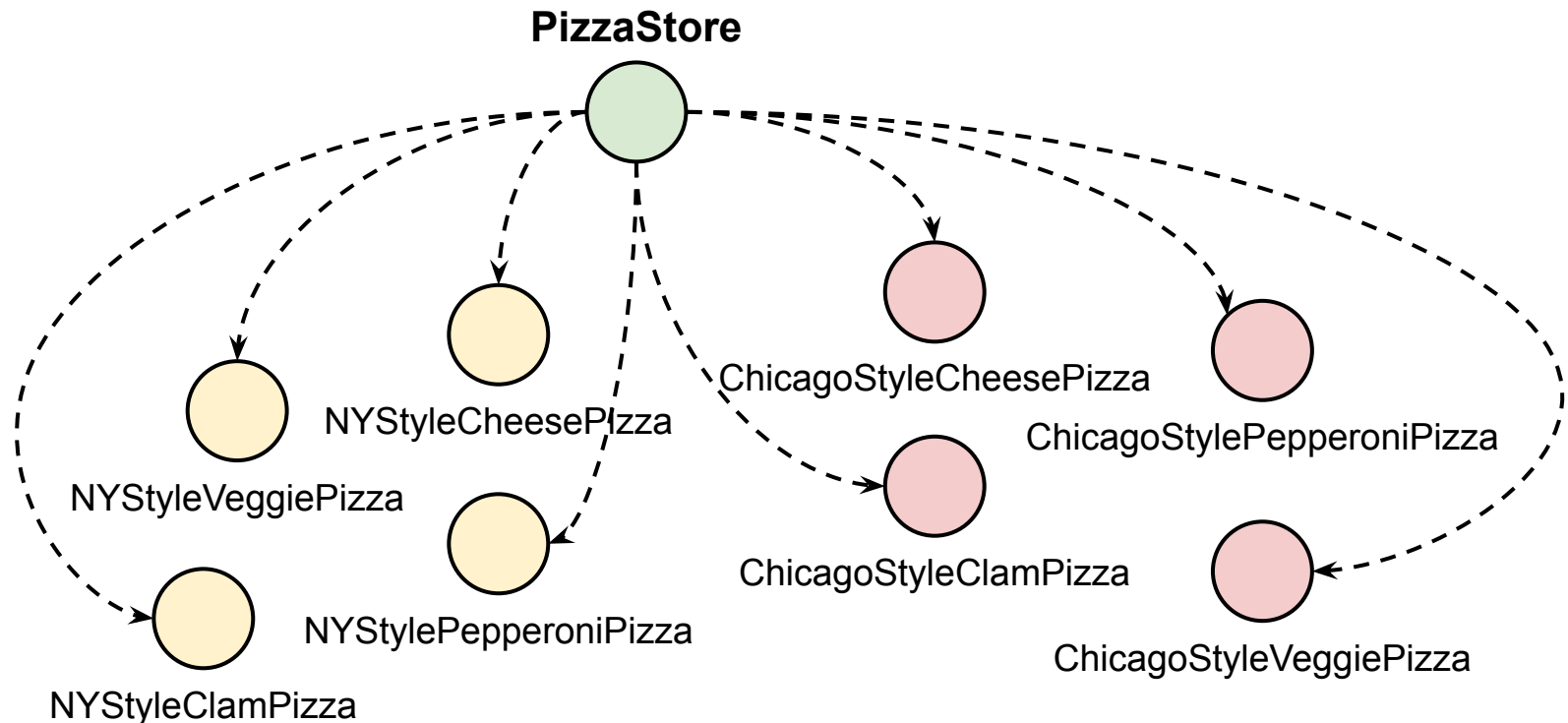
# A very dependent PizzaStore

## If we did not know about factory method...

```
public class VeryDependentPizzaStore {  
    public Pizza orderPizza (String style, String type) {  
        Pizza pizza;  
        if (style.equals ("NY")) {  
            if (type.equals ("cheese")) {  
                pizza = new NYStylePizza ();  
            } else if (...) {  
                ...  
            }  
        } else if (style.equals ("Chicago")) {  
            ...  
        } else {  
            System.out.println ("error: invalid pizza type");  
            return null;  
        }  
  
        pizza.prepare ();  
        pizza.bake ();  
        pizza.cut ();  
        pizza.box ();  
        return pizza;  
    }  
}
```

# Looking at object dependencies

## A very dependent pizza store



# Design principle

**Depend on abstractions. Do not depend on concrete classes.**

- *The Dependency Inversion Principle.*
- Similar to “program to an interface, not an implementation”, but stronger.
- High-level components should not depend on low-level components.
- Clients define and own the interfaces.

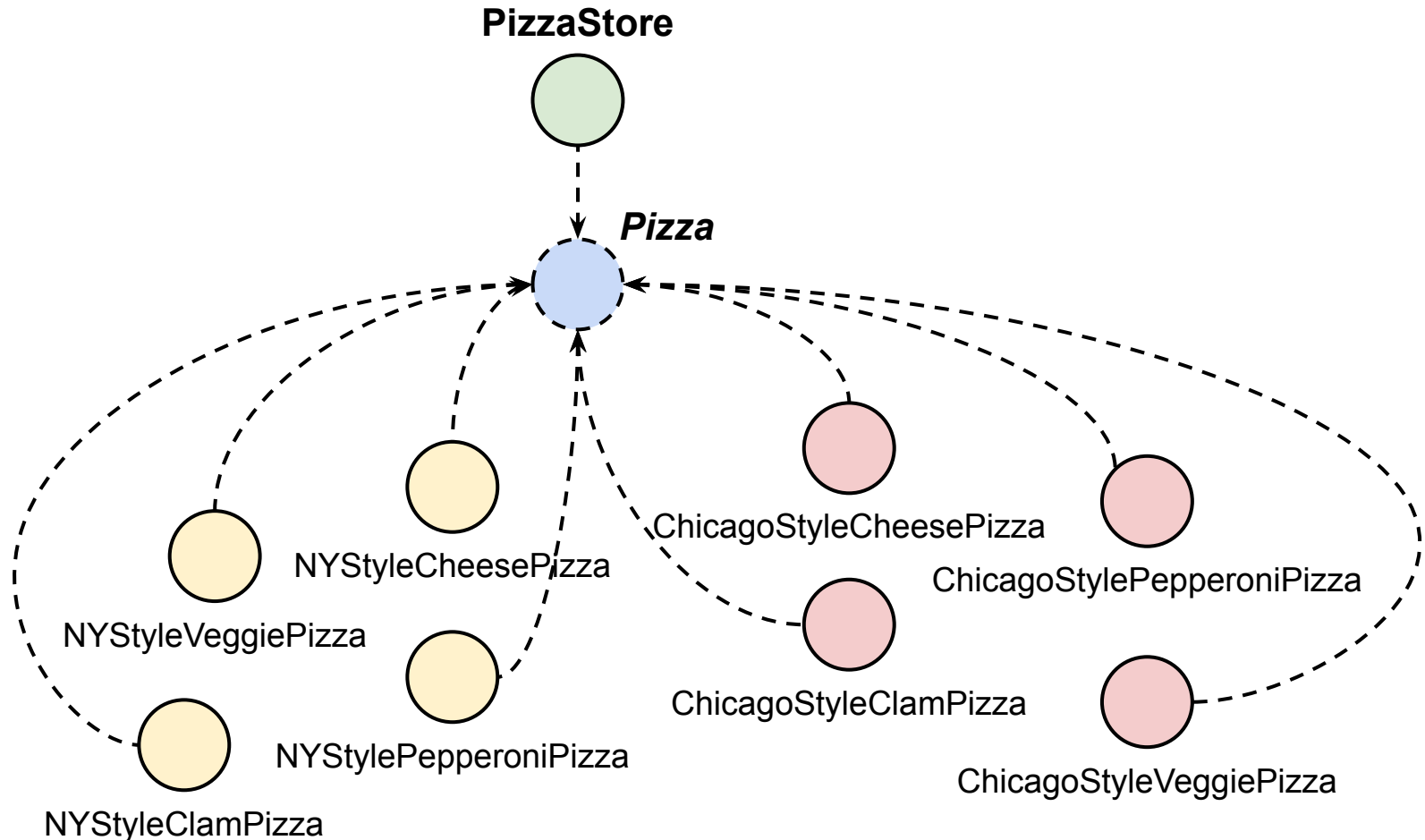
## The result

- Both high-level and low-level components depend on abstractions.



# Applying the principle

## Dependencies inverted



# Avoiding DIP violations

## Basic guidelines

- No variable should refer to a concrete class.
  - Using “new” causes dependency on concrete class.
  - Use factory to get around that.
- No class should derive from a concrete class.
  - Dependency on concrete class.
  - Derive from an abstraction.
- No method should override an implemented method in the superclass.
  - Otherwise the base class was not really an abstraction.

## Impossible to follow literally!

- Depending on concrete classes is not a problem if they are very unlikely to change, e.g., String.

# Back to the pizza store

## **Pizza stores follow the procedures...**

- Embodied in the `orderPizza()` method.

## **... but they cheat on the ingredients**

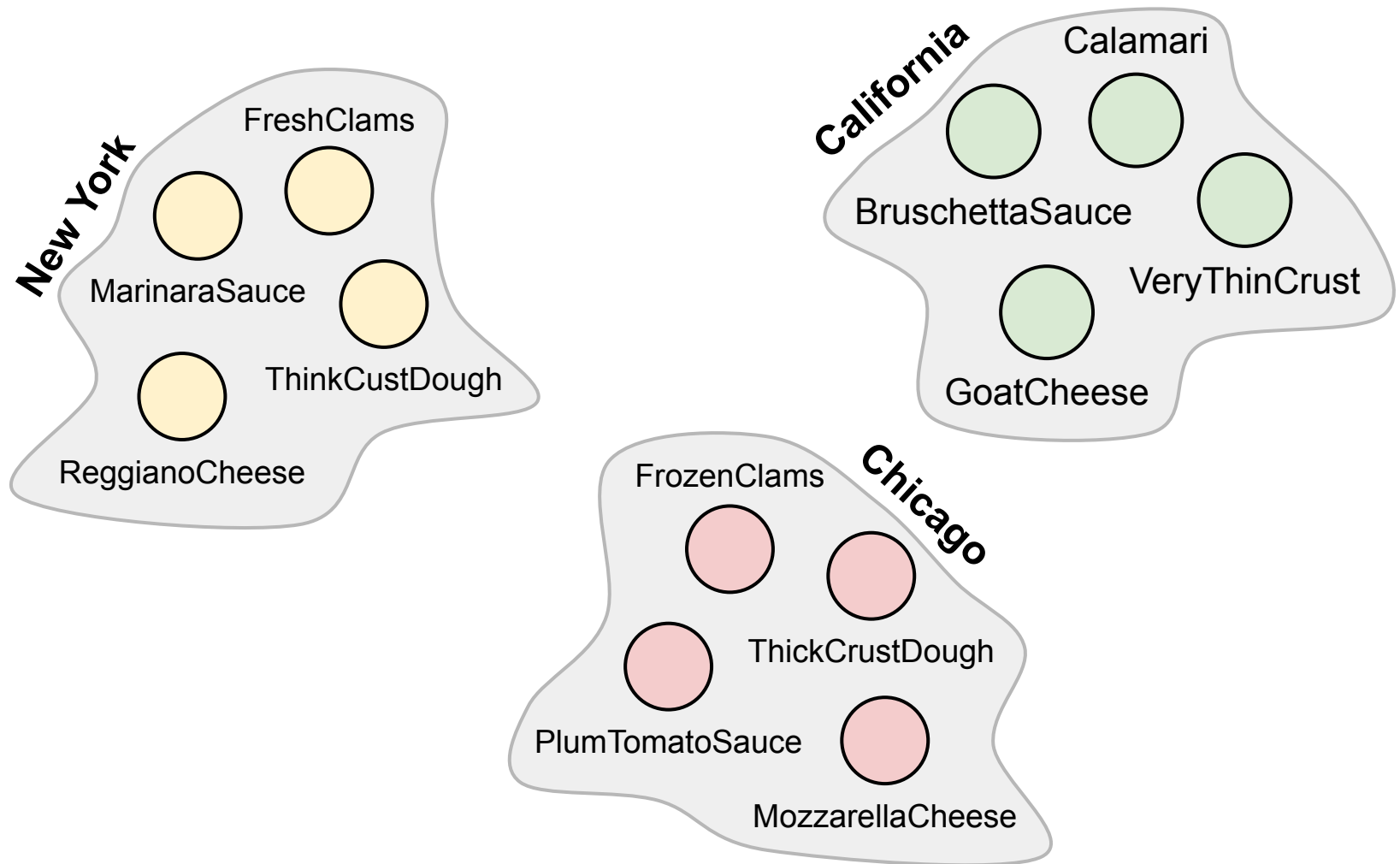
- They are responsible for preparing pizzas.

## **Ensuring consistency in ingredients**

- Provide stores with a factory for ingredients.
- But “red sauce” in NY != “red sauce” in Chicago.
  - The same applies to “cheese”, “clams”, ...
- We need regional factories for ingredients.

# Families of ingredients

So different, yet still the same



# Building ingredient factories

## Creating ingredients from ingredient families

- Concepts are the same, specifics differ.

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

# New York ingredient factory

## Capturing NY specifics

```
public class NYPizzaIngredientFactory implements PizzaIngredientFactory {  
  
    public Dough createDough () { return new ThinCrustDough (); }  
  
    public Sauce createSauce () { return new MarinaraSauce (); }  
  
    public Cheese createCheese () { return new ReggianoCheese (); }  
  
    public Collection <Veggie> createVeggies () {  
        return Arrays.asList (  
            new Garlic (), new Onion (),  
            new Mushroom (), new RedPepper ()  
        );  
    }  
  
    public Pepperoni createPepperoni () { return new SlicedPepperoni (); }  
  
    public Clams createClams () { return new FreshClams (); }  
}
```

# Reworking the pizzas (1)

## Delegate preparation to subclasses

```
public abstract class Pizza {  
    String name;  
    Dough dough;  
    Sauce sauce;  
    Collection <Veggie> veggies;  
    Pepperoni pepperoni;  
    Clams clams;  
  
    abstract void prepare ();  
  
    void bake () { ... };  
    void cut () { ... };  
    void box () { ... };  
  
    void setName (String name) { ... };  
    String getName () { ... };  
  
    public String toString () { ... };  
}
```

The prepare() method is now abstract, and the subclasses are responsible for collecting the appropriate ingredients. These will come from the ingredient factory.

# Reworking the pizzas (2)

## We need less pizza classes

- NY and Chicago cheese pizzas are the same, but use different ingredients.

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

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

    void prepare () {
        dough = ingredientFactory.createDough ();
        sauce = ingredientFactory.createSauce ();
        cheese = ingredientFactory.createCheese ();
    }
}
```



# Revisiting the pizza store

## Tying stores and ingredient factories

```
public class NYPizzaStore extends PizzaStore {  
  
    protected Pizza createPizza (String type) {  
        PizzaIngredientFactory ingredientFactory =  
            new NYPizzaIngredientFactory ();  
  
        Pizza pizza;  
        if (type.equals ("cheese")) {  
            pizza = new CheesePizza (ingredientFactory);  
            pizza.setName ("New York Style Cheese Pizza");  
        } else if (type.equals ("veggie")) {  
            ...  
        } else if (...) {  
            ...  
        }  
  
        return pizza;  
    }  
}
```

# What exactly did we do?

## Enabled creating family of pizza ingredients

- Using a new type of factory: Abstract Factory

## Interface for creating family of products

- Code using the interface is decoupled from actual factory creating the products.
- Allows implementing different factories producing products meant for different contexts.
  - Regions, operating systems, look & feel, ...

## Decoupled code = flexible design

- Substitute factories to get different behaviors.

# The Abstract Factory pattern (1)

## Intent

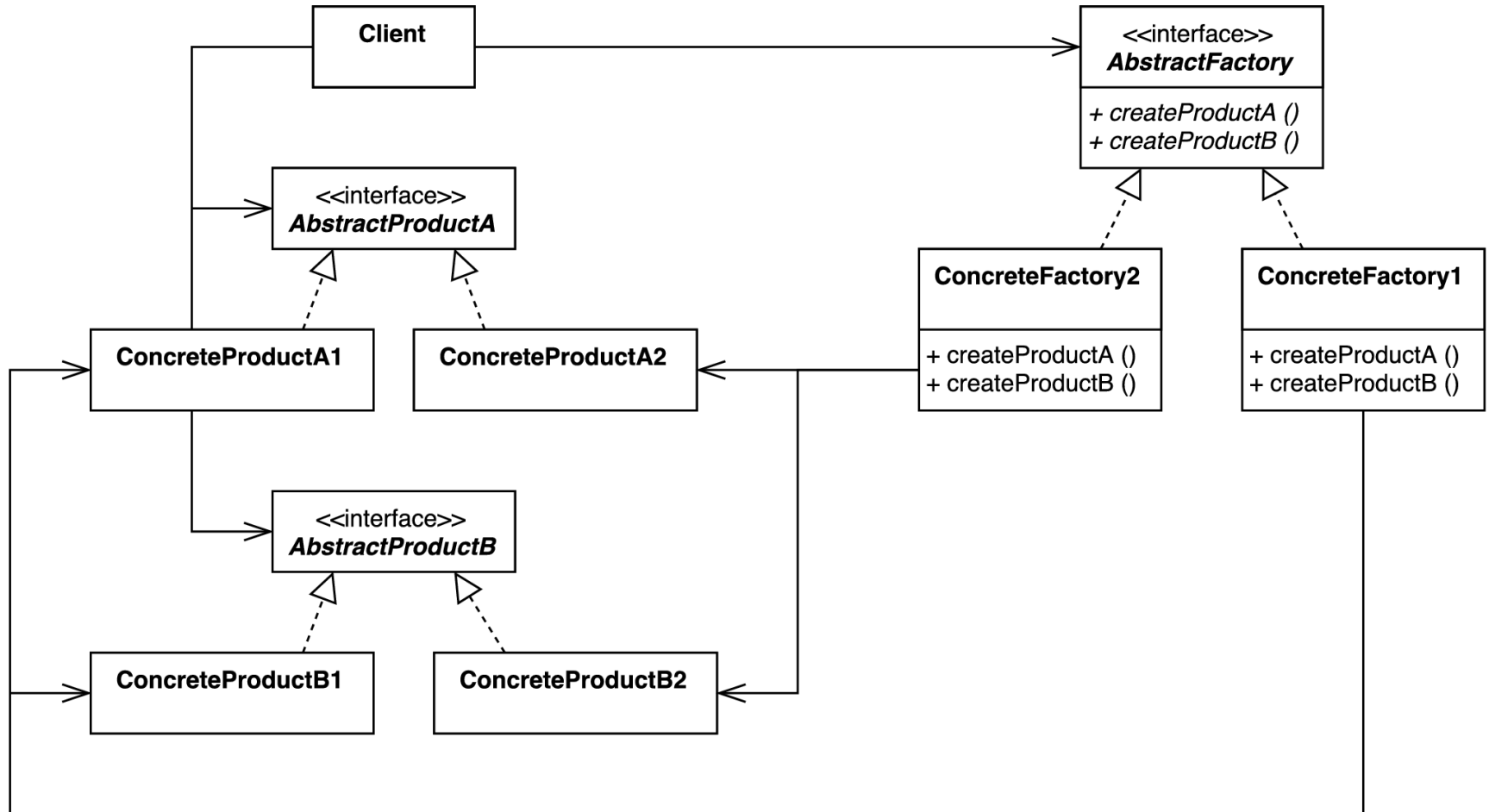
- Provides an interface for creating families of related or dependent objects without specifying their concrete classes.

## Benefits

- Decouples client from specifics of concrete products.

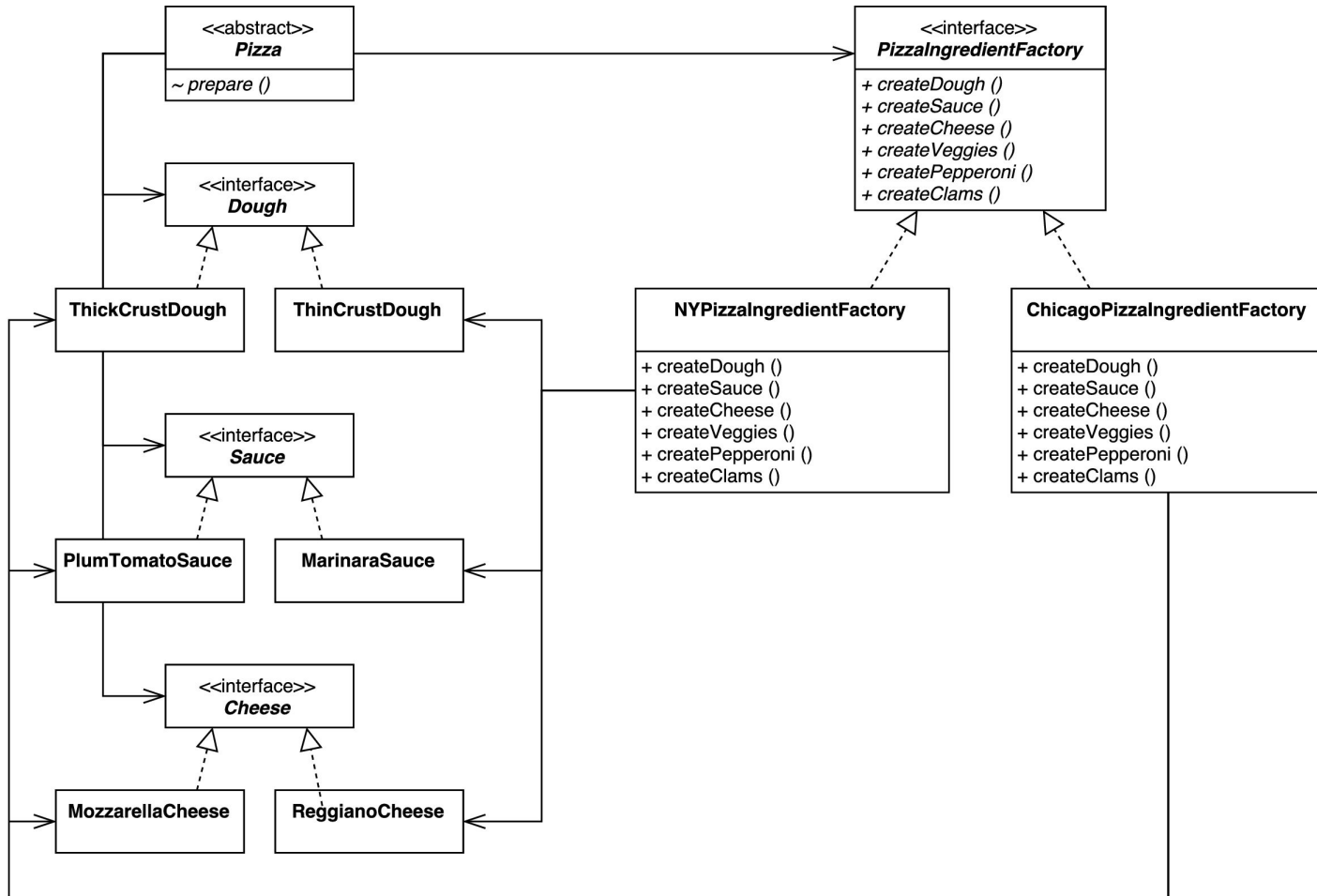
# The Abstract Factory pattern (2)

## Structure



# Abstract Factory applied

## The Abstract Factory in pizza store



# Factory patterns compared (1)

## Common traits

- All factories encapsulate object creation.
- All factory patterns promote loose coupling by reducing dependency on concrete classes.
- Technique for coding to abstractions, not concrete classes.

## Simple Factory

- Not really a pattern, just a simple way to decouple clients from concrete classes.

# Factory patterns compared (2)

## **Factory Method implementation**

- Relies on inheritance.
- Object creation is delegated to subclasses which implement the factory method to create objects.

## **Abstract Factory implementation**

- Relies on composition.
- Object creation is implemented in methods exposed in the factory interface.

# Factory patterns compared (3)

## **Factory Method intent**

- Allow a class to defer instantiation to its subclasses.

## **Abstract Factory intent**

- Create families of related objects without having to depend on their concrete classes.



# The Adapter pattern

## Being adaptive

- Need to put a square peg in a round hole?
  - Design patterns to the rescue!
- Recall the Decorator pattern: objects were wrapped to give them responsibilities.
- Here we wrap objects to make their interface look like something they are not.
  - Or to make their interface simpler.

# Adapters all around us

## The real world is full of adapters

- AC power adapters, MacBook video adapters, ...

## Some adapters are simple

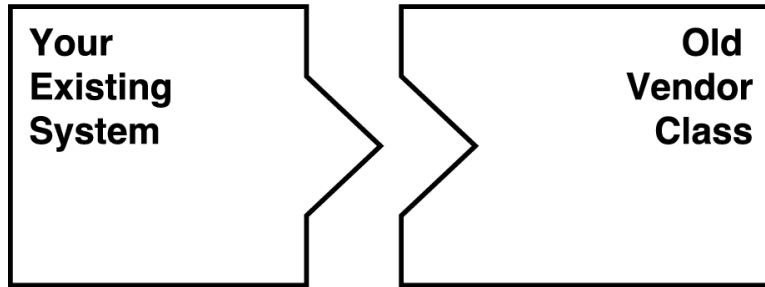
- Only change the external shape to provide a matching interface
  - Socket already provides 220V expected by a device

## Some adapters need to be more complex

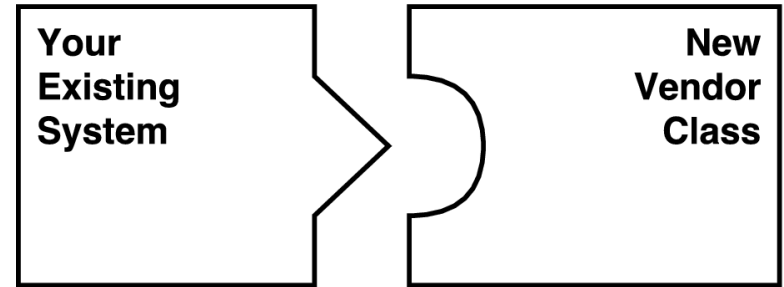
- Matching interface is not enough, more adaptation work is needed
  - Socket provides 220V but device requires 110V
  - Socket provides 110V but device requires 220V

# Object oriented adapters

**Using classes from  
3rd party vendor**

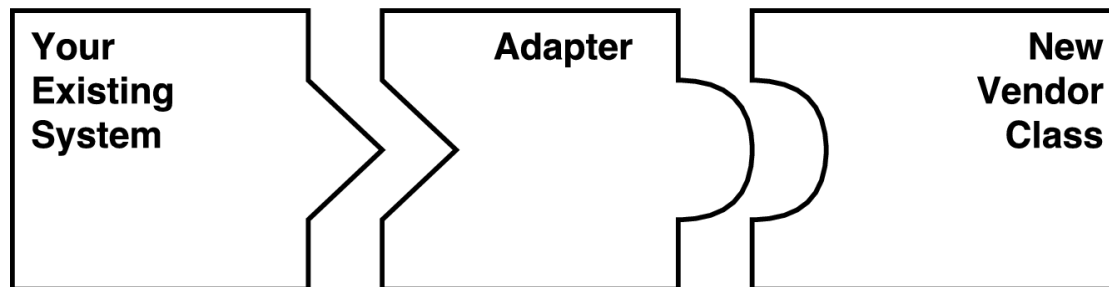


**Changing the  
3rd party vendor**



**Using adapter to integrate the new class**

- Without modifying the existing system.



# Adapter in action (1)

## A simplified Duck interface...

```
public interface Duck {  
    public void quack();  
    public void fly();  
}  
  
public class MallardDuck implements Duck {  
    public void quack() {  
        System.out.println("Quack");  
    }  
  
    public void fly() {  
        System.out.println("I'm flying!");  
    }  
}
```

# Adapter in action (2)

## Now consider a new fowl on the block...

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

Turkeys can fly, but only for a very short distance.

```
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!");  
    }  
}
```

# Adapter in action (3)

## Using turkeys in place of ducks

- Not directly possible, the interfaces are different.

```
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 ();  
        }  
    }  
}
```

Adapt the Turkey interface to the Duck interface.

Make up for the difference between a duck flying and a turkey flying.

# Adapter in action (4)

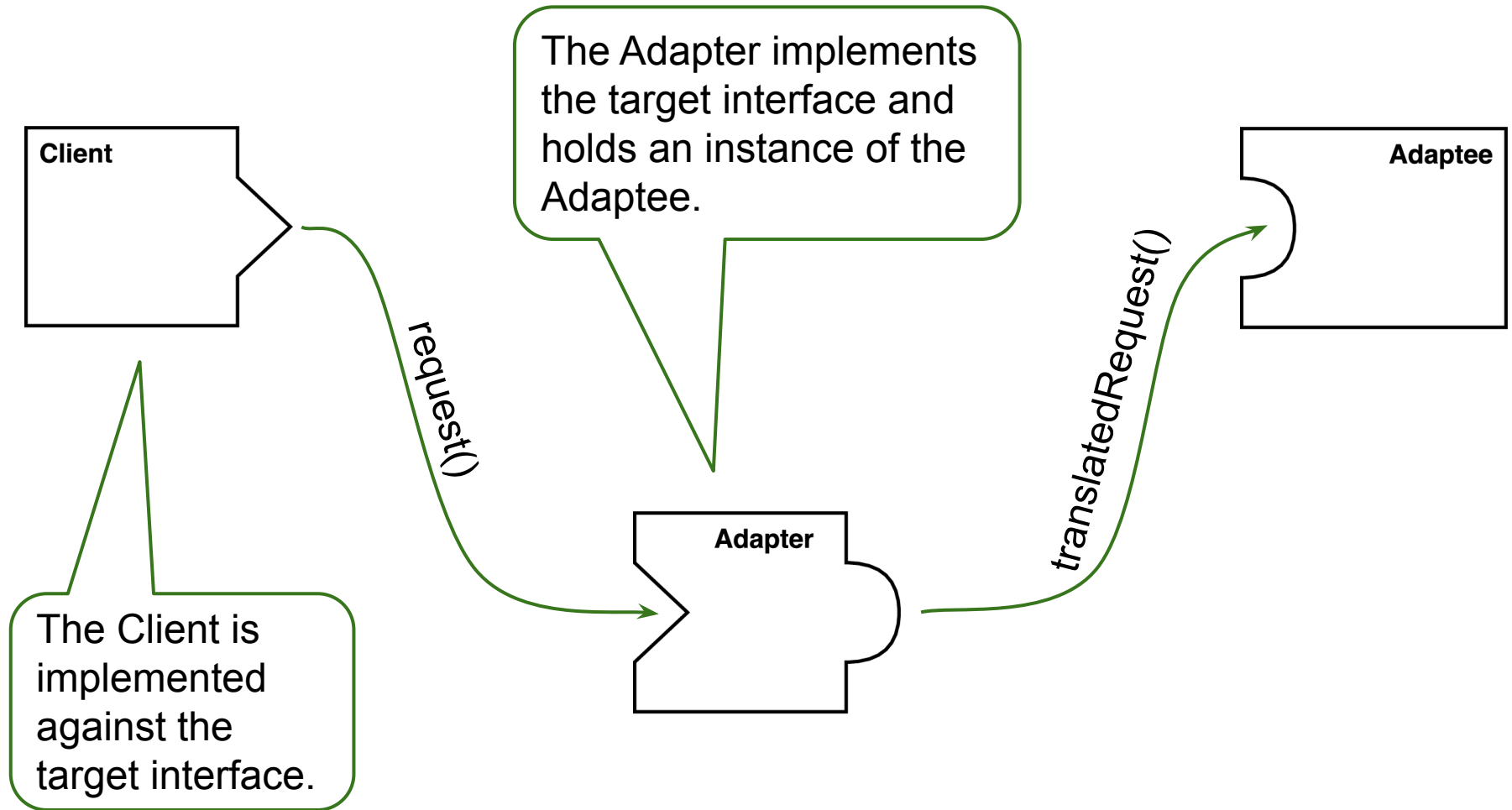
## Testing the adapter

```
public static void main (String [] args) {  
    MallardDuck duck = new MallardDuck ();  
    WildTurkey turkey = new WildTurkey ();  
    Duck turkeyAdapter = new TurkeyAdapter (turkey);  
  
    System.out.println ("The Turkey says...");  
    turkey.gobble ();  
    turkey.fly ();  
  
    System.out.println ("The Duck says...");  
    exerciseDuck (duck);  
  
    System.out.println ("The TurkeyAdapter says...");  
    exerciseDuck (turkeyAdapter);  
}  
  
static void exerciseDuck (Duck duck) {  
    duck.quack ();  
    duck.fly ();  
}
```

Wrap Turkey in a TurkeyAdapter to make it look like a Duck.

Try to pass off the turkey as a duck.

# Reviewing all the pieces





# The Adapter pattern (1)

## Intent

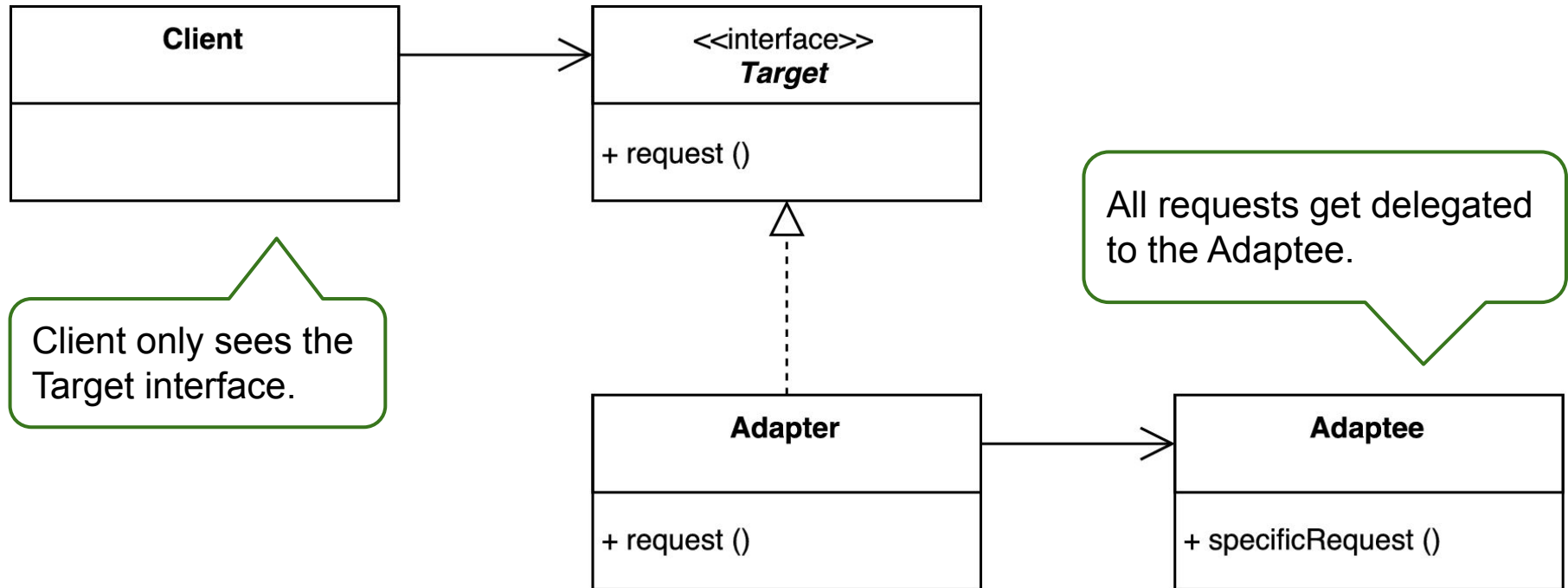
- Converts the interface of a class into another interface the clients expect. Adapter lets classes work together that could not otherwise because of incompatible interfaces.

## Benefits

- Encapsulates change in the interface.
- Decouples client from the implemented interface.
- Client does not have to be modified each time it needs to operate against a different interface.

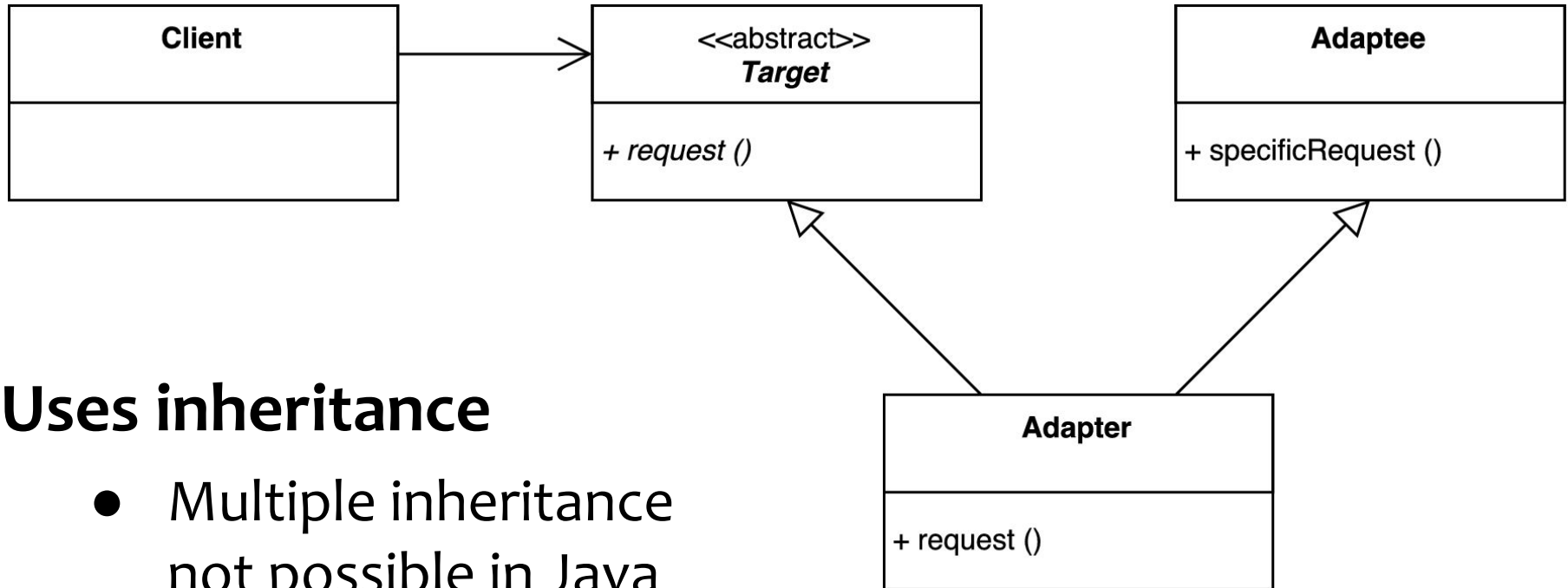
# The Adapter pattern (2)

## Structure



# Object vs Class Adapter

## Class Adapter structure



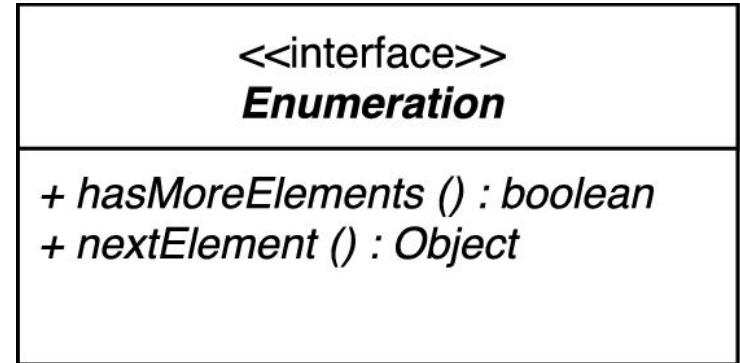
## Uses inheritance

- Multiple inheritance not possible in Java
- May make implementation a bit easier.
- Avoids potential trouble with object identity.
- Less flexible compared to composition.

# Real world adapters

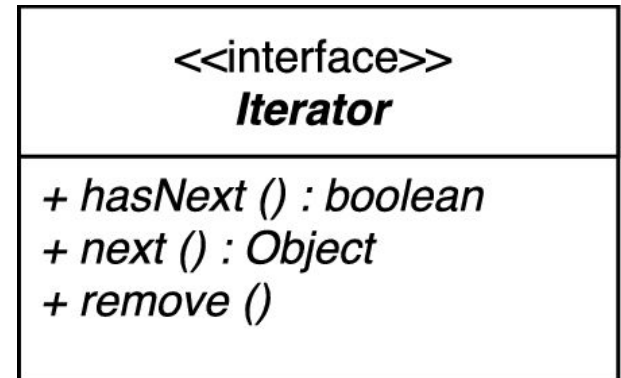
## Old-style Enumerators

- Early collection types.
  - Vector, Stack, Hashtable, ...
- The elements() method.



## New-style Iterators

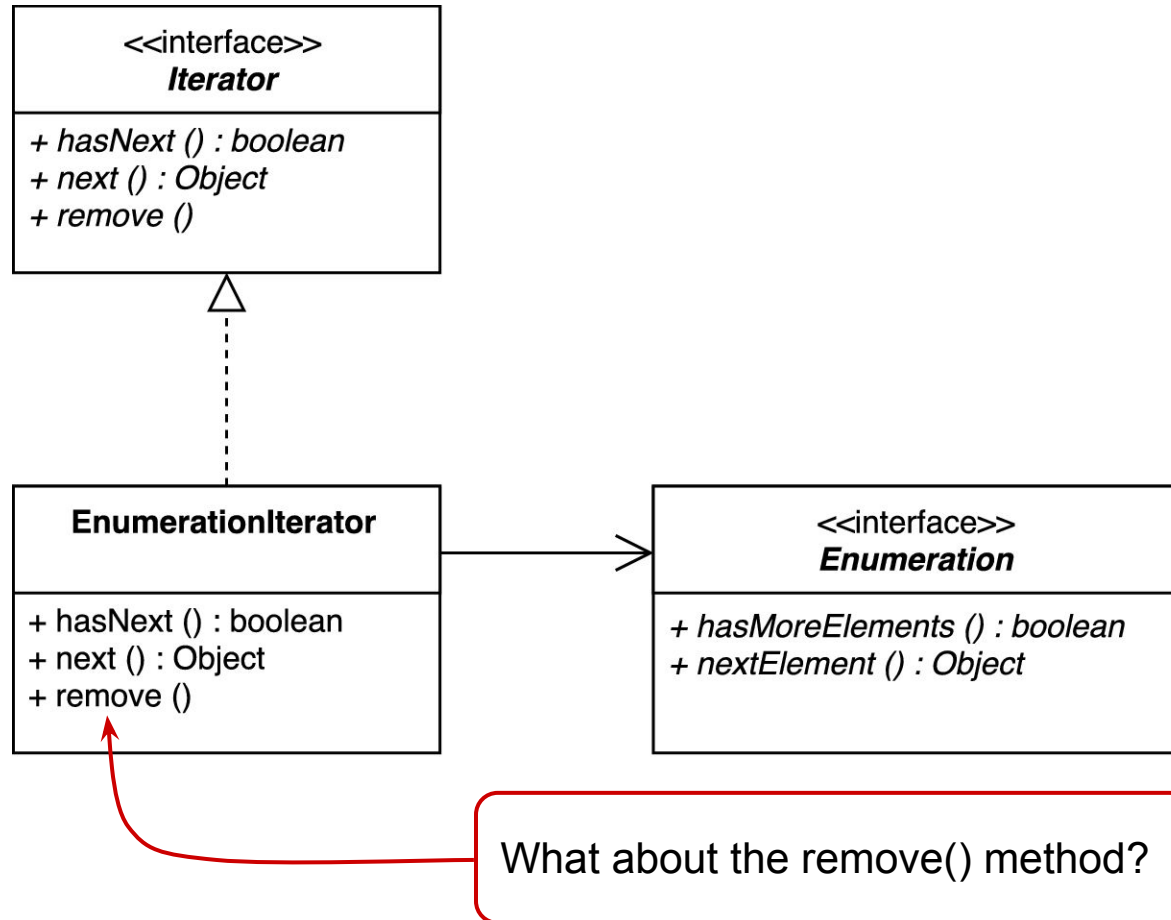
- Recent collection types.
- Allows removing items.
- Less verbose.



What if legacy code exposes Enumeration?

# Adapting Enumeration to Iterator

## Designing an adapter



# EnumerationIterator adapter

## Writing the EnumerationIterator adapter

```
public class EnumerationIterator <E> implements Iterator <E> {  
    private final Enumeration <E> enum;  
  
    public EnumerationIterator (Enumeration <E> enum) {  
        this.enum = enum;  
    }  
  
    public boolean hasNext () {  
        return enum.hasMoreElements ();  
    }  
  
    public E next () {  
        return enum.nextElement ();  
    }  
  
    public void remove () {  
        throw new UnsupportedOperationException ();  
    }  
}
```

We cannot support the Iterator's remove() method, so we have to give up and throw an exception.

**Important: The Iterator's contract allows it!**

# Decorator vs Adapter

## Technically similar, different in intent

- Both wrap objects and delegate methods.
  - Apart from Class Adapter, which does not wrap.
- Both help avoid changing the existing code.

## Decorator intent

- Wrap to add new behaviors and responsibilities.
- The interface does not change.

## Adapter intent

- Wrap one or more objects to provide an interface that the client expects.
  - An uncoupled client is a happy client.

# The Facade pattern

## Making things simple

- Providing simpler interface to a subsystem.
  - By hiding the complexity of one or more classes.
- Similar to Adapter, but with different intent!

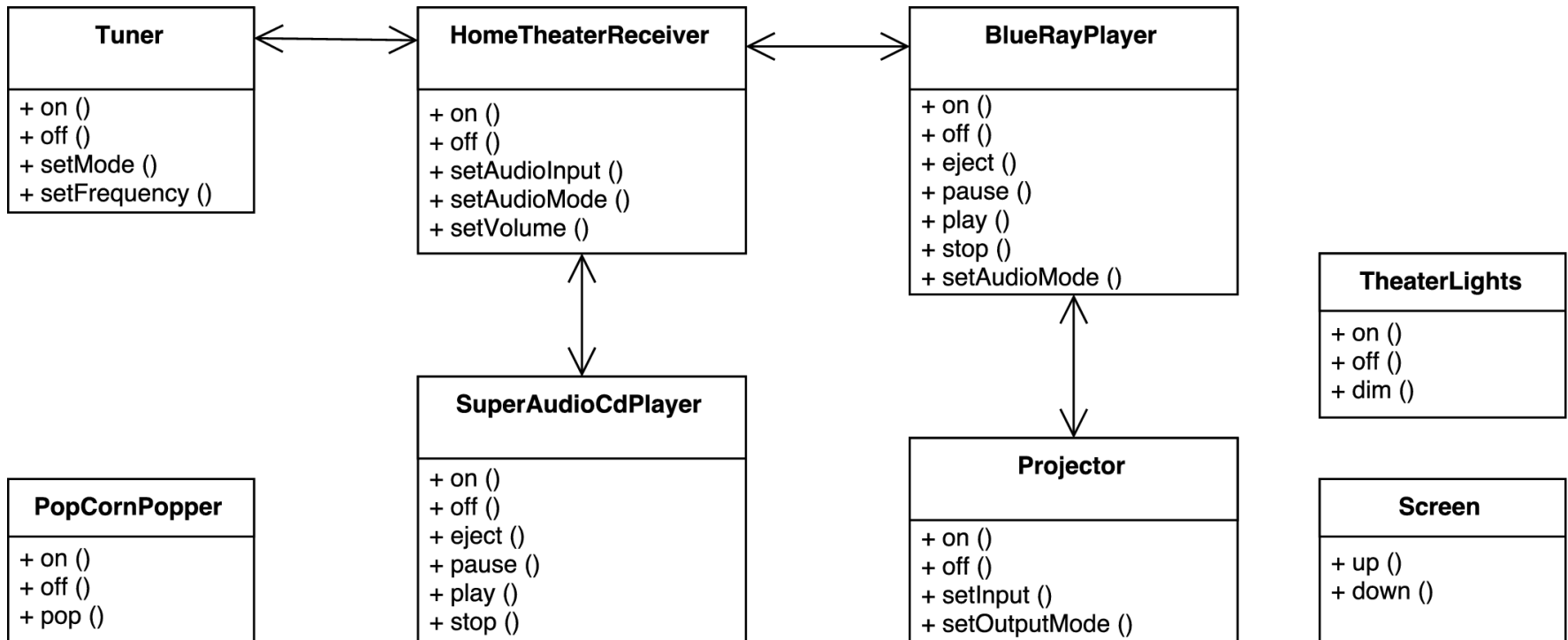
**But let's build a little home theater first!**



# Home Sweet Home Theater

## We have assembled a killer system

- With a lot of classes, interactions, and interfaces to learn, though.



# Watching a movie (1)

## The tasks to watch a movie

1. Turn on the popcorn popper
2. Start the popper popping
3. Put the screen down
4. Turn the projector on
5. Set the projector input to BluRay
6. Put the projector on wide-screen mode
7. Turn the HT receiver on
8. Set receiver input to BluRay
9. Set the receiver to surround sound
10. Set the receiver volume to medium (5)
11. Turn the BluRay player on
12. Start the BluRay player
13. Dim the lights

# Watching a movie (2)

## Tasks in terms of method calls

```
popper.on ();  
popper.pop ();
```

```
screen.down ();
```

```
projector.on ();  
projector.setInput (BluRay);  
projector.setOutputMode (WideScreen);
```

```
receiver.on ();  
receiver.setAudioInput (BluRay);  
receiver.setAudioMode (SurroundSound);  
receiver.setVolume (5);
```

```
bdPlayer.on ();  
bdPlayer.play (movie);
```

```
lights.dim (0.1);
```

# Watching a movie (3)

## And there is more...

- How to turn things off when the movie ends?
  - Repeat in reverse order?
- What about listening to CD or radio?
  - Would it be also as complex?
- What if something... changes?
  - Maybe the procedure would need to be adjusted?

## How can we just enjoy a movie?

- Without messing with all the complexity?

# Lights, Camera, Facade!

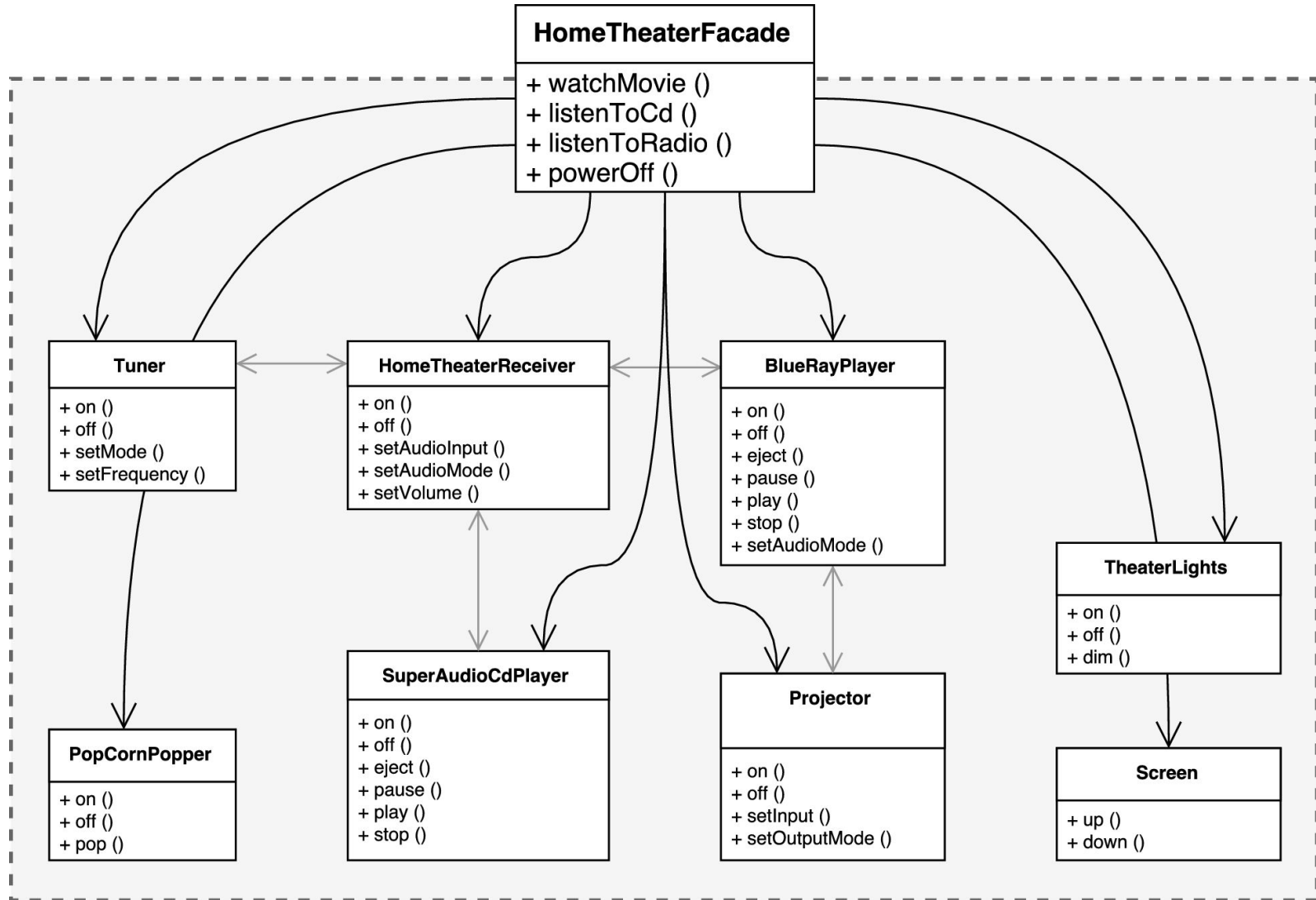
## The Facade pattern

- Take a complex subsystem and make it easier to use by implementing a Facade class.
  - Provides more reasonable interface.
- The power of the complex subsystem is not lost.
  - If necessary, it can still be used.
- But simple tasks should be simple.
  - Difficult tasks should be possible.

## ... just what the doctor ordered!

- We need a HomeTheaterFacade!
  - With few simple methods for basic tasks.

# The simplified home theater



# The Facade pattern (1)

## Intent

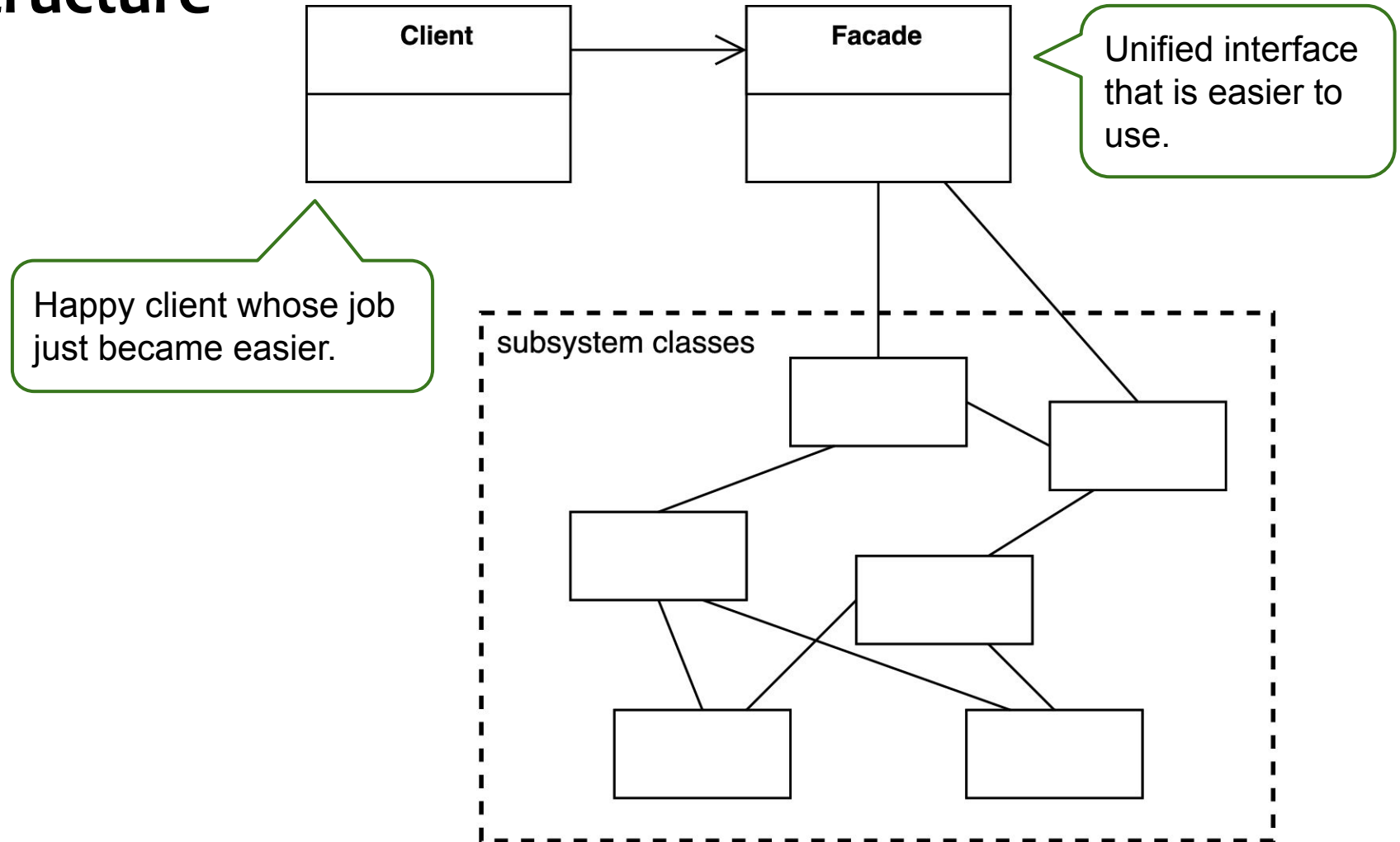
- Provides a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

## Benefits

- Decouples clients from particular subsystems.
- Helps adhering to the principle of least knowledge.

# The Facade pattern (2)

## Structure





# Design principle

## Principle of Least Knowledge: talk only to our immediate friends

- *Also known as The Law of Demeter.*
  - No principle should be considered a law.
  - Principles should be applied when they are helpful.
- Prevents creating designs with a large number of classes coupled together, where changes in one part cascade to the other parts.

## The result

- Design that is less fragile, easier to understand, and less costly to maintain.

# What does it mean?

## How many classes is the code coupled to?

```
public float getTemp () {  
    return station.getThermometer().getTemperature ();  
}
```

## What should we do?

- From any method in an object, we should only invoke methods that belong to:
  - The object itself.
  - Any components of the object.
  - Objects passed in as method parameters.
  - Any object the method creates/instantiates.

**Notice that we are not supposed to call methods on objects returned by calls to other methods.**

# Let us try again...

## Without the principle

- We make a request of another object's subpart.
- Increasing the number of objects we know directly.

```
public float getTemp () {  
    Thermometer thermometer = station.getThermometer();  
    return thermometer.getTemperature ();  
}
```

## With the principle

- Ask the object to make the request for us.
- Keep the circle of friends (known objects) small.

```
public float getTemp () {  
    return station.getTemperature ();  
}
```

# Keeping method calls in bounds

## Calling methods without violating the principle

```
public class Car {  
    Engine engine;  
  
    public Car () { /* initialize engine, etc. */ }  
  
    public void start (Key key) {  
        Doors doors = new Doors ();  
  
        boolean authorized = key.turns ();  
        if (authorized) {  
            engine.start ();  
            updateDashboardDisplay ();  
            doors.lock ();  
        }  
    }  
}  
  
void updateDashboardDisplay () { /* update display */ }  
}
```

The diagram illustrates the relationships between the code elements and their components:

- Method parameter.** Points to `key.turns ();` in the `start` method.
- Component of Car.** Points to `engine.start ();` in the `start` method.
- Local method.** Points to `updateDashboardDisplay ();` in the `start` method.
- Instantiated object.** Points to `doors.lock ();` in the `start` method.

# The Facade and the PLK

## Considering the home theater

- The client only has to know one class.
- The facade manages subsystem components.
  - Keeps the client simple and flexible.
- Subsystems should adhere to PLK as well.
  - If too many friends are intermingling, additional facades can be introduced to form layers of subsystems.

## Disadvantages of PLK

- Results in more “wrapper” classes that handle method calls to other components.
  - Increases complexity and development time.

# Other useful patterns

## Template method

- Outline of an algorithm, subclasses provide details.
- Similar to Strategy (which relies on composition).

## Iterator

- Enables traversal of the elements of an aggregate object (e.g., a collection) without exposing the underlying representation.
- Iterator object responsible for traversal.

## Composite

- Enables building tree object structures to represent part-whole hierarchies and allow the clients to treat them uniformly.