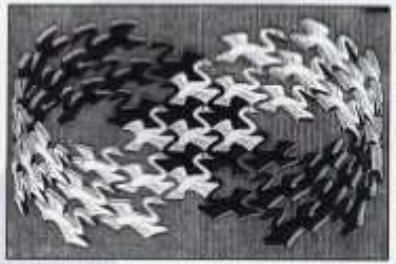
Design Patterns

Michael Heitland
Oct 2015

Design Patterns

Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides



Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

411

4.5 out of 5 stars

5 star	70%
4 star	17%
3 star	8%
2 star	3%
1 star	2%

See all 411 reviews

Creational Patterns

- Abstract Factory
- Builder
- Factory Method
- Object Pool*
- Prototype
- Simple Factory*
- Singleton

(* this pattern got added later by others)

Structural Patterns

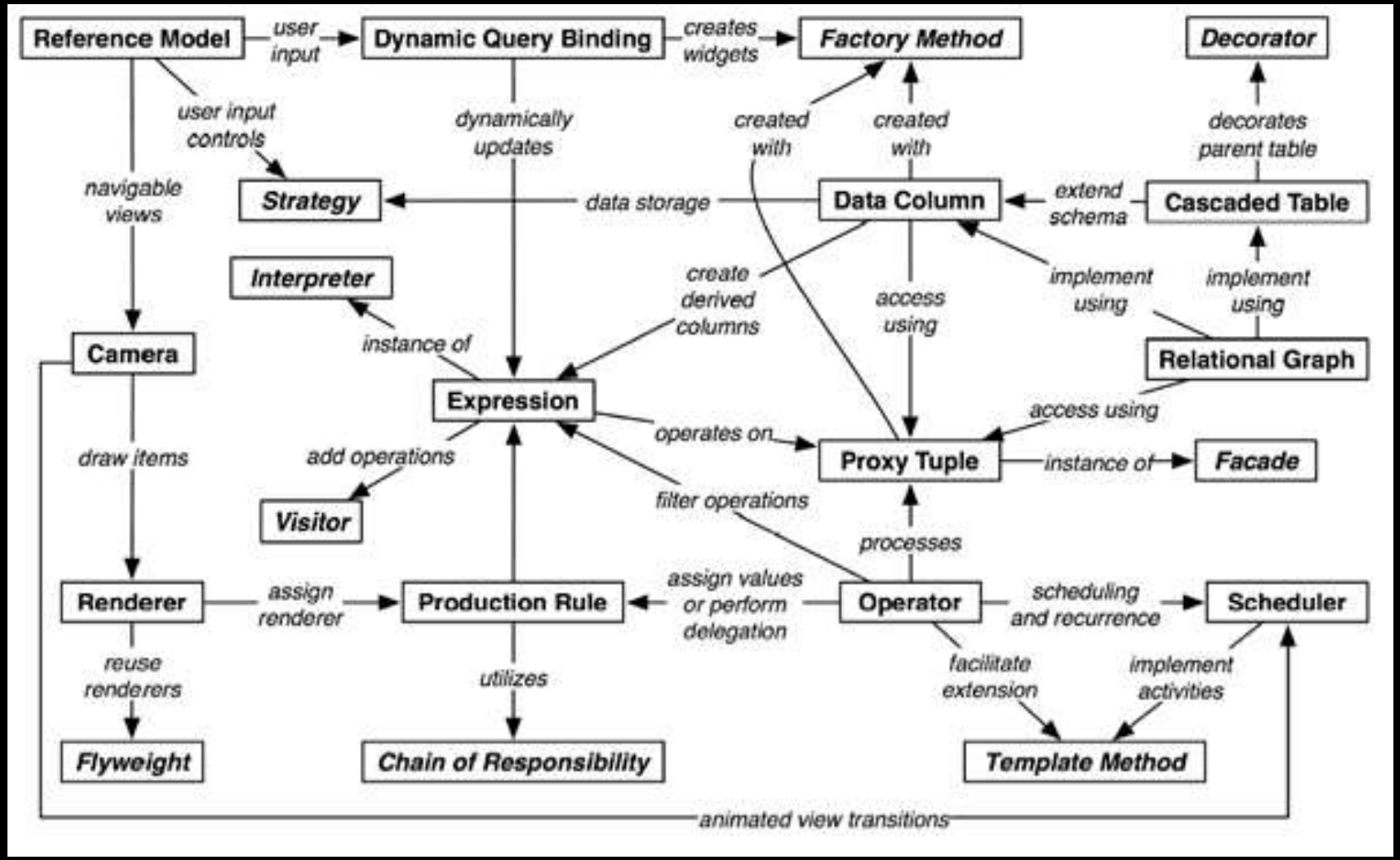
- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy

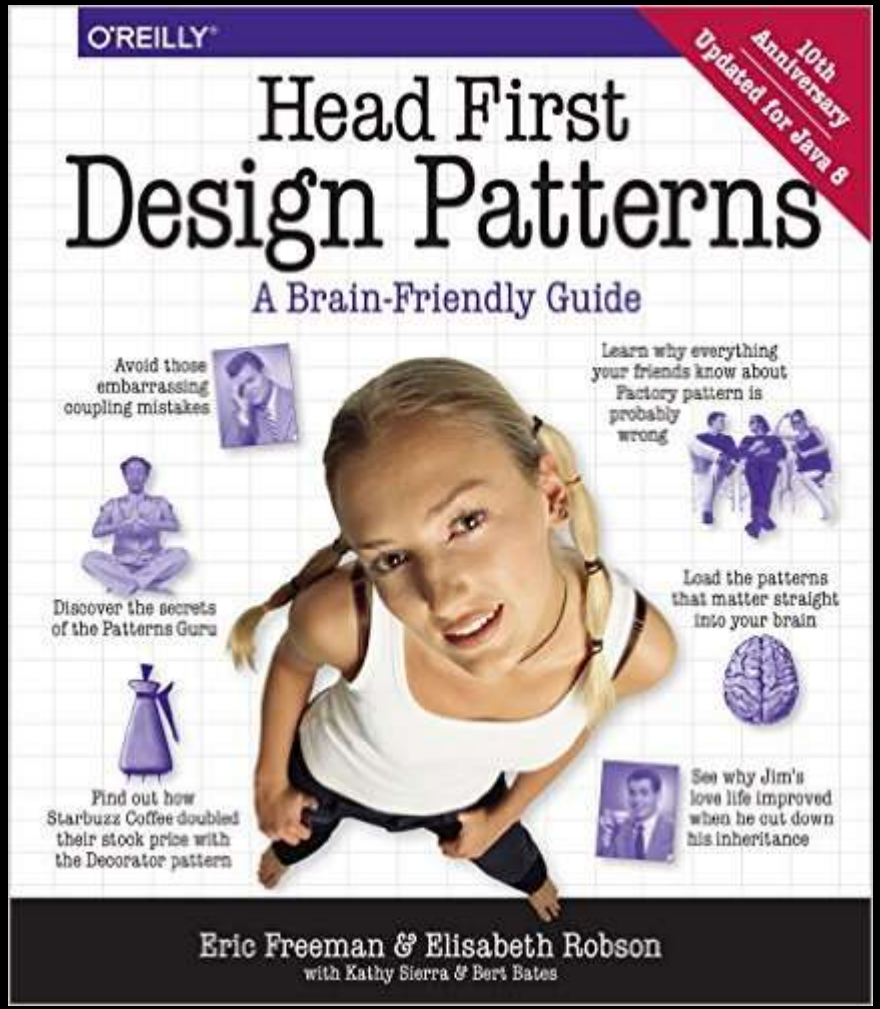
Behavioural Patterns 1

- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento

Behavioural Patterns 2

- Null Object *
- Observer
- State
- Strategy
- Template Method
- Visitor


Reference Model
user input
Dynamic Query Binding
creates widgets
Factory Method
Decorator
user input controls
dynamically updates
created with
created with
decorates parent table
navigable views
Strategy
data storage
Data Column
extend schema
Cascaded Table
Interpreter
create derived columns
access using
implement using
implement using
instance of
Camera
Relational Graph
Expression
access using
operates on
Proxy Tuple
instance of
Facade
draw items
add operations
filter operations
Visitor
processes
Renderer
assign renderer
Production Rule
assign values or perform delegation
Operator
scheduling and recurrence
Scheduler
reuse renderers
utilizes
facilitate extension
implement activities
Flyweight
Chain of Responsibility
Template Method
animated view transitions



Customer Reviews

476

4.5 out of 5 stars

5 star	73%
4 star	16%
3 star	5%
2 star	4%
1 star	2%

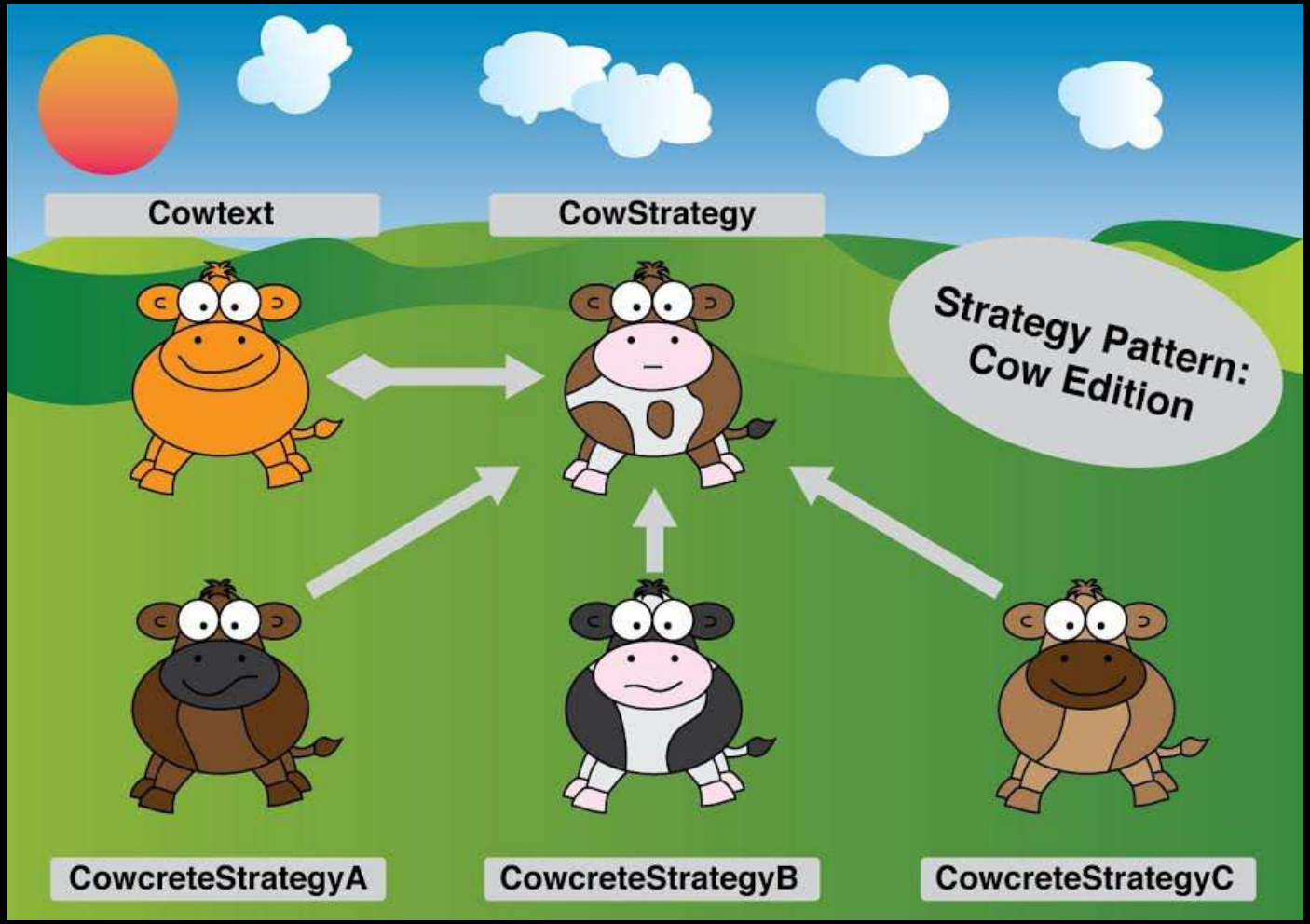
Initial	Acronym	Concept
S	SRP	Single responsibility principle: A class should have only a single responsibility (i.e. only one potential change in the software's specification should be able to affect the specification of the class)
O	OCP	Open/closed principle: “Software entities … should be open for extension, but closed for modification.”
L	LSP	Liskov substitution principle: “Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.” See also design by contract.
I	ISP	Interface segregation principle: “Many client-specific interfaces are better than one general-purpose interface.”[8]
D	DIP	Dependency inversion principle: One should “Depend upon Abstractions. Do not depend upon concretions.”[8]

Creational Patterns

Simple Factory*

Encapsulating object creation.

Clients will use object interfaces.


Cowtext
CowStrategy
Strategy Pattern:
Cow Edition
CowcreteStrategyA
CowcreteStrategyB
CowcreteStrategyC

Here's our new class, the SimplePizzaFactory. It has one job in life: creating pizzas for its clients.

```
public class SimplePizzaFactory {

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

        if (type.equals("cheese")) {
            pizza = new CheesePizza();
        } else if (type.equals("pepperoni")) {
            pizza = new PepperoniPizza();
        } else if (type.equals("clam")) {
            pizza = new ClamPizza();
        } else if (type.equals("veggie")) {
            pizza = new VeggiePizza();
        }
        return pizza;
    }
}
```

First we define a createPizza() method in the factory. This is the method all clients will use to instantiate new objects.

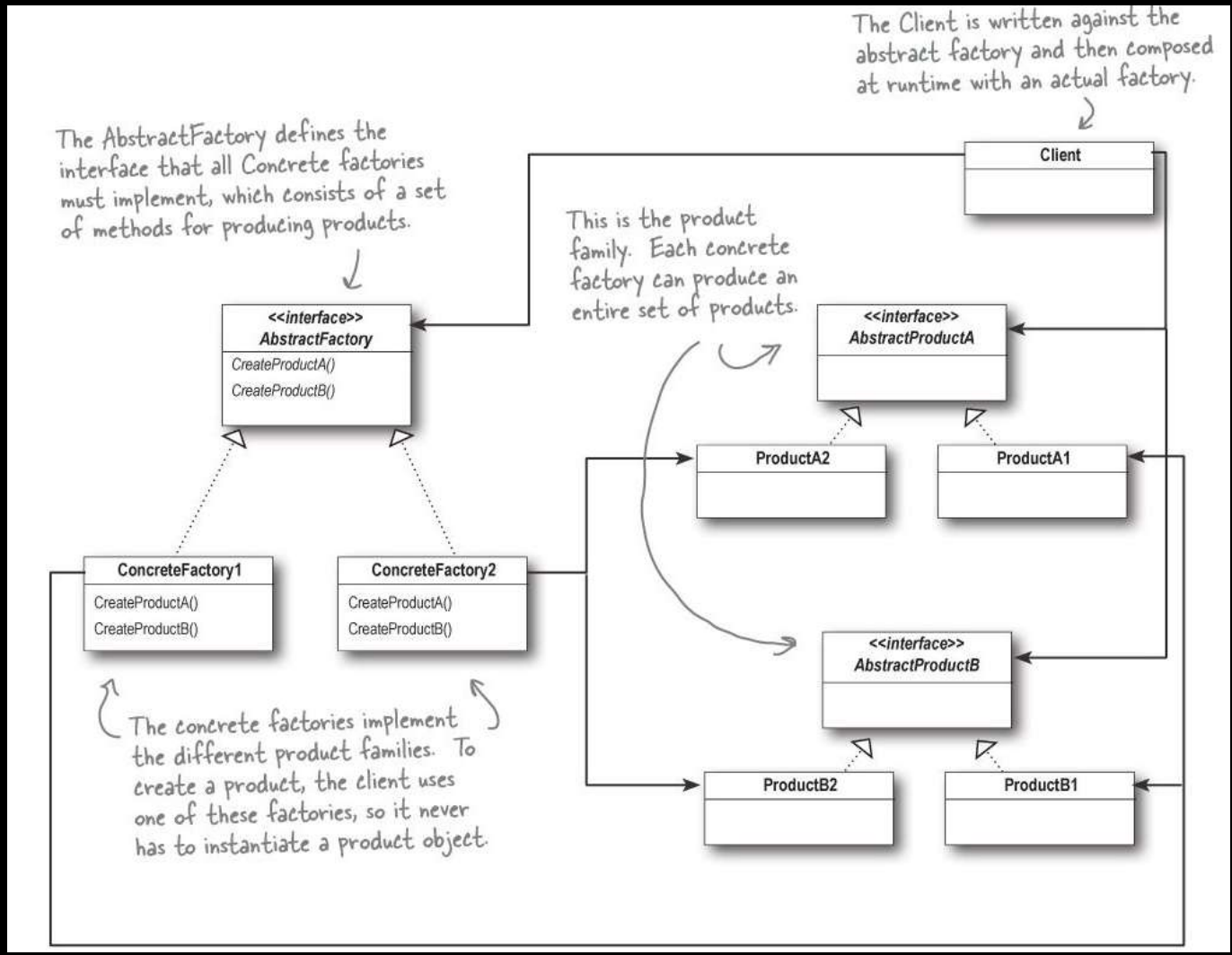
Here's the code we plucked out of the orderPizza() method.

This code is still parameterized by the type of the pizza, just like our original orderPizza() method was.

Abstract Factory

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

Inject the factory into the object.


The Client is written against the abstract factory and then composed at runtime with an actual factory.
The AbstractFactory defines the interface that all Concrete factories must implement, which consists of a set of methods for producing products.
This is the product family. Each concrete factory can produce an entire set of products.
Client
<<interface>>
AbstractFactory
CreateProductA()
CreateProductB()
<<interface>>
AbstractProductA
ProductA2
ProductA1
ConcreteFactory1
CreateProductA()
CreateProductB()
ConcreteFactory2
CreateProductA()
CreateProductB()
<<interface>>
AbstractProductB
ProductB2
ProductB1
The concrete factories implement the different product families. To create a product, the client uses one of these factories, so it never has to instantiate a product object.

```
public class NYPizzaIngredientFactory implements PizzaIngredientFactory {

    public Dough createDough() {
        return new ThinCrustDough();
    }

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

    public Cheese createCheese() {
        return new ReggianoCheese();
    }
```

For each ingredient in the ingredient family, we create the New York version.

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

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

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

To make a pizza now, we need a factory to provide the ingredients. So each Pizza class gets a factory passed into its constructor, and it's stored in an instance variable.

Here's where the magic happens!

The prepare() method steps through creating a cheese pizza, and each time it needs an ingredient, it asks the factory to produce it.

```
public class NYPizzaStore extends PizzaStore {

    protected Pizza createPizza(String item) {
        Pizza pizza = null;
        PizzaIngredientFactory ingredientFactory =
            new NYPizzaIngredientFactory();

        if (item.equals("cheese")) {

            pizza = new CheesePizza(ingredientFactory);
            pizza.setName("New York Style Cheese Pizza");

        } else if (item.equals("veggie")) {

            pizza = new VeggiePizza(ingredientFactory);
            pizza.setName("New York Style Veggie Pizza");

        } else if (item.equals("clam")) {

            pizza = new ClamPizza(ingredientFactory);
            pizza.setName("New York Style Clam Pizza");

        } else if (item.equals("pepperoni")) {

            pizza = new PepperoniPizza(ingredientFactory);
            pizza.setName("New York Style Pepperoni Pizza");

        }
        return pizza;
    }
}
```

The NY Store is composed with a NY pizza ingredient factory. This will be used to produce the ingredients for all NY style pizzas.

We now pass each pizza the factory that should be used to produce its ingredients.

Look back one page and make sure you understand how the pizza and the factory work together!

For each type of Pizza, we instantiate a new Pizza and give it the factory it needs to get its ingredients.

Dependency Inversion Principle

Depend upon abstractions.
Do not depend upon concrete classes.

Our high-level components should not depend on our low-level components; rather, they should *both* depend on abstractions.

Builder

Separate the construction of a complex object from its implementation so that the two can vary independently.

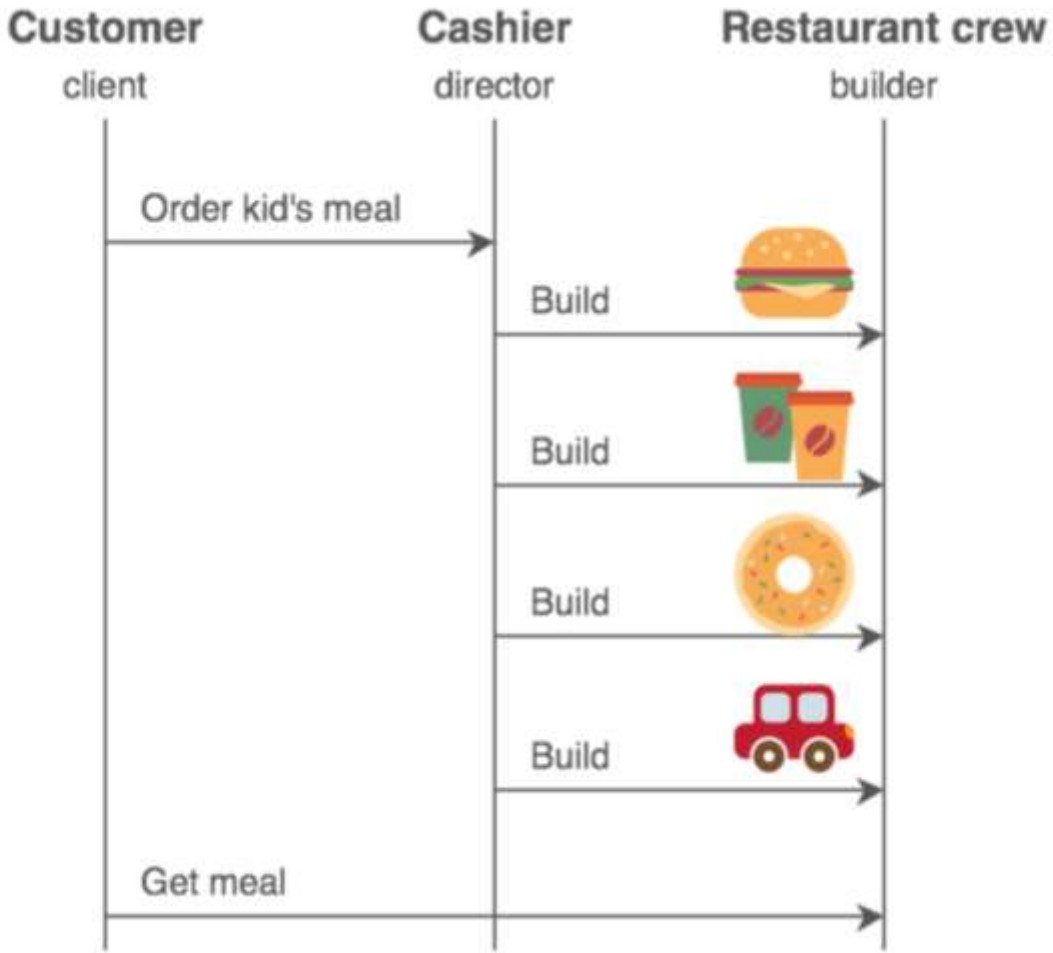
The same construction process can create different representations.

Often used for building composite structures step by step.

The "director" invokes "builder" services as it interprets the external format.

The "builder" creates part of the complex object each time it is called and maintains all intermediate state. When the product is finished, the client retrieves the result from the "builder".

Affords finer control over the construction process. Unlike creational patterns that construct products in one shot, the Builder pattern constructs the product step by step under the control of the "director".


Customer
client
Cashier
director
Restaurant crew
builder
Order kid's meal
Build
Build
Build
Build
Get meal

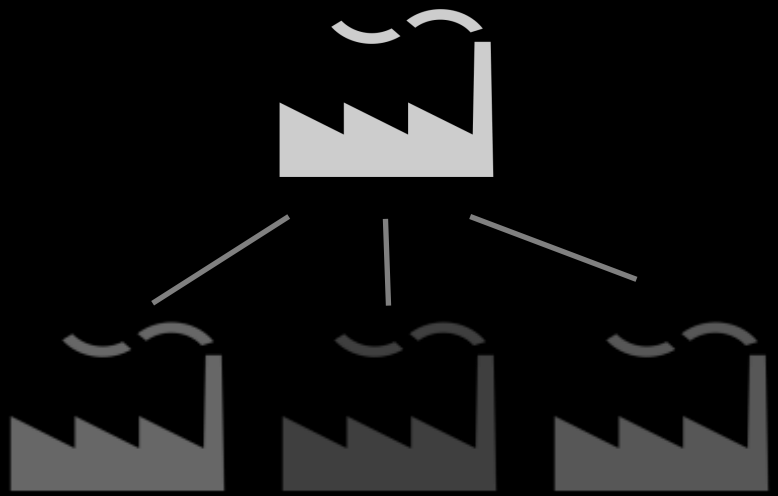
```
public class Pizza
{
    private readonly string _dough;
    private readonly string _sauce;
    private readonly string _topping;

    private Pizza(Builder builder) { _dough = builder.Dough; _sauce = builder.Sauce; _topping = builder.Topping; }

    public override string ToString() { return $"Dough: {_dough}, Sauce: {_sauce}, Topping: {_topping}"; }

    public static void Main()
    {
        var pizza = new Pizza.Builder().SetSauce("Pomodoro").SetTopping("Parmesan").Build();
        Console.WriteLine(pizza);
    }

    public class Builder
    {
        public string Dough { get; set; } = "default dough";
        public string Sauce{ get; set; } = "default sauce";
        public string Topping { get; set; } = "default topping";

        public Builder SetDough(string dough) { Dough = dough; return this; }
        public Builder SetSauce(string sauce) { Sauce = sauce; return this; }
        public Builder SetTopping(string topping) { Topping = topping; return this; }

        public Pizza Build() { return new Pizza(this); }
    }
}
```

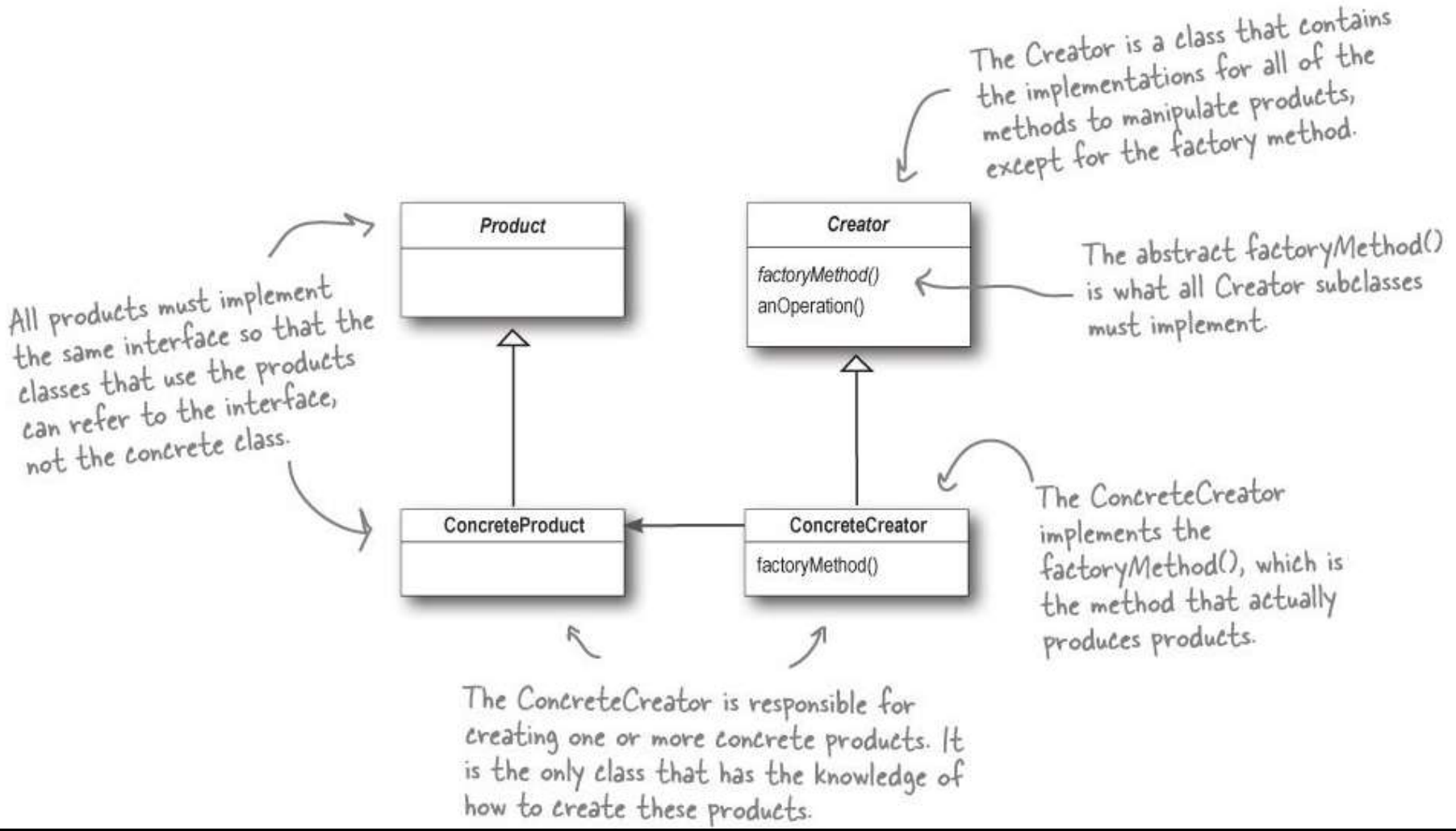
Factory Method

Define an interface for creating an object, but let subclasses decide which class to instantiate.

Factory Method lets a class defer instantiation to subclasses.

This decouples the client code in the base class from the object creation code in the subclasses.




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

```
public abstract class PizzaStore {

    public Pizza orderPizza(String type) {
        Pizza pizza;

        pizza = createPizza(type);

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

        return pizza;
    }

    protected abstract Pizza createPizza(String type);

    // other methods here
}
```

The subclasses of PizzaStore handle object instantiation for us in the createPizza() method.

All the responsibility for instantiating Pizzas has been moved into a **method** that acts as a **factory**.

createPizza() returns a Pizza, and the subclass is fully responsible for which concrete Pizza it instantiates.

The NYPizzaStore extends PizzaStore, so it inherits the orderPizza() method (among others).

```
public class NYPizzaStore extends PizzaStore {

    Pizza createPizza(String item) {
        if (item.equals("cheese")) {
            return new NYStyleCheesePizza();
        } else if (item.equals("veggie")) {
            return new NYStyleVeggiePizza();
        } else if (item.equals("clam")) {
            return new NYStyleClamPizza();
        } else if (item.equals("pepperoni")) {
            return new NYStylePepperoniPizza();
        } else return null;
    }
}
```

We've got to implement createPizza(), since it is abstract in PizzaStore.

Here's where we create our concrete classes. For each type of Pizza we create the NY style.

```
public class PizzaTestDrive {

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

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

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

First we create two different stores.

Then use one one store to make Ethan's order.

And the other for Joel's.

Object Pool*

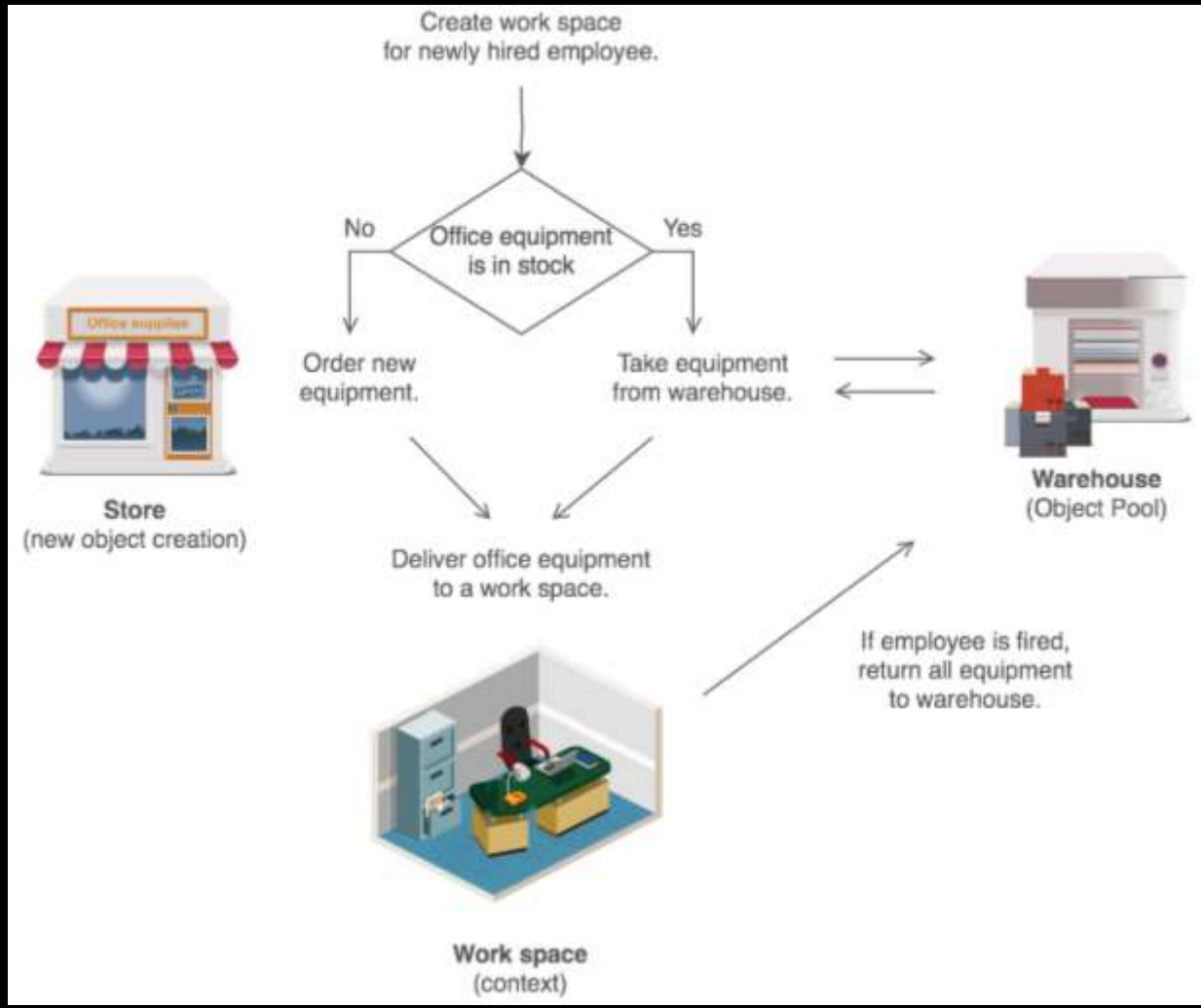
Avoid expensive acquisition and release of resources by recycling objects that are no longer in use.

Biggest performance boost if instantiation cost and rate is high, but number is limited.

A pool of reusable objects manages object caching. The pool is often a Singleton.

It creates new objects if needed, usually there is a limit on the total number of reusable objects (e.g. db connections).

Unused objects get returned to the pool and will be reused. The pool runs a facility to clean them up periodically.


Create work space
for newly hired employee.
Office equipment
is in stock
No
Yes
Order new
equipment.
Take equipment
from warehouse.
Store
(new object creation)
Warehouse
(Object Pool)
Deliver office equipment
to a work space.
If employee is fired,
return all equipment
to warehouse.
Work space
(context)

Prototype

**Specify the kinds of objects to create using a prototypical instance,
and create new objects
by copying this prototype.**

Co-opt one instance of a class for use as a breeder of all future instances. The *new* operator considered harmful.

Prototype can avoid expensive "creation from scratch", and support cheap cloning of a pre-initialized prototype with few variations on the initialization parameters.

Declare an abstract base class that specifies a pure virtual "*clone*" method, and, maintains a dictionary of all "cloneable" concrete derived classes.

Any class that needs a "polymorphic constructor" capability derives itself from the abstract base class, registers its prototypical instance, and implements the *clone()* operation (member-wise clone for shallow copies, deserialization for deep copies).

Singleton

Ensures a class has only one instance and provide a global point of access to it.

The Singleton class is responsible to create the object, not the client!

Example:

- **Shared cached data prevents overuse of resources**
- **Factory**
- **Configuration**
- **Resource Coordinator (thread pool, connection pool …)**

But be aware of problems around having a global state!

Why not just use a simple static class?

- Static classes cannot use interfaces and inheritance and you cannot create objects of static classes and pass them as parameters
- Less namespace cluttering than having global vars
- Allow lazy allocation / initialization

```
public class Singleton
{
    // Problem: C# standard does not tell us when _instance gets created
    // It could be at first usage (lazy) or any time before
    private static readonly Singleton
        _instance = new Singleton();

    private Singleton() { }

    public static Singleton Instance
    { get { return _instance; } }

    // fields, properties, methods ...
}
```

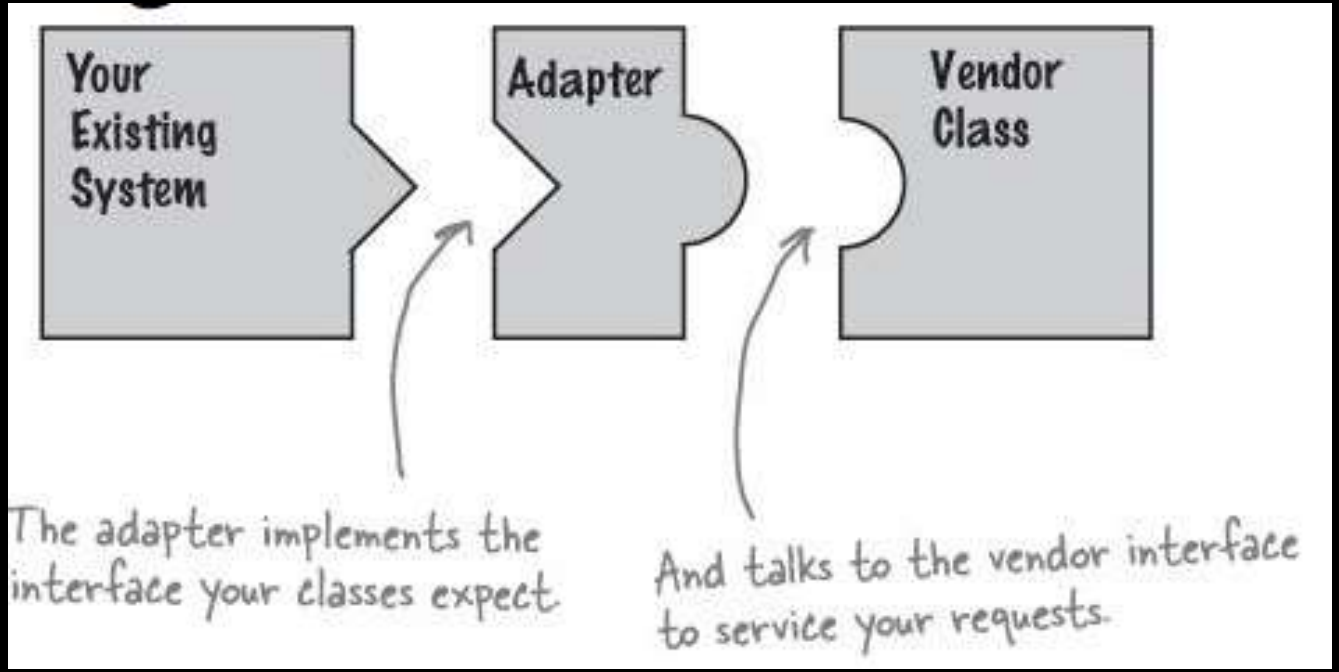
```
public class LazySingleton
{
    // need to use lambda to construct since constructor private
    private static readonly Lazy<LazySingleton> _instance
        = new Lazy<LazySingleton>(() => new LazySingleton());

    // private to prevent direct instantiation.
    private LazySingleton() { }

    // accessor for instance
    public static LazySingleton Instance { get { return _instance.Value; } }

    // fields, properties, methods ...
}
```

Structural Patterns

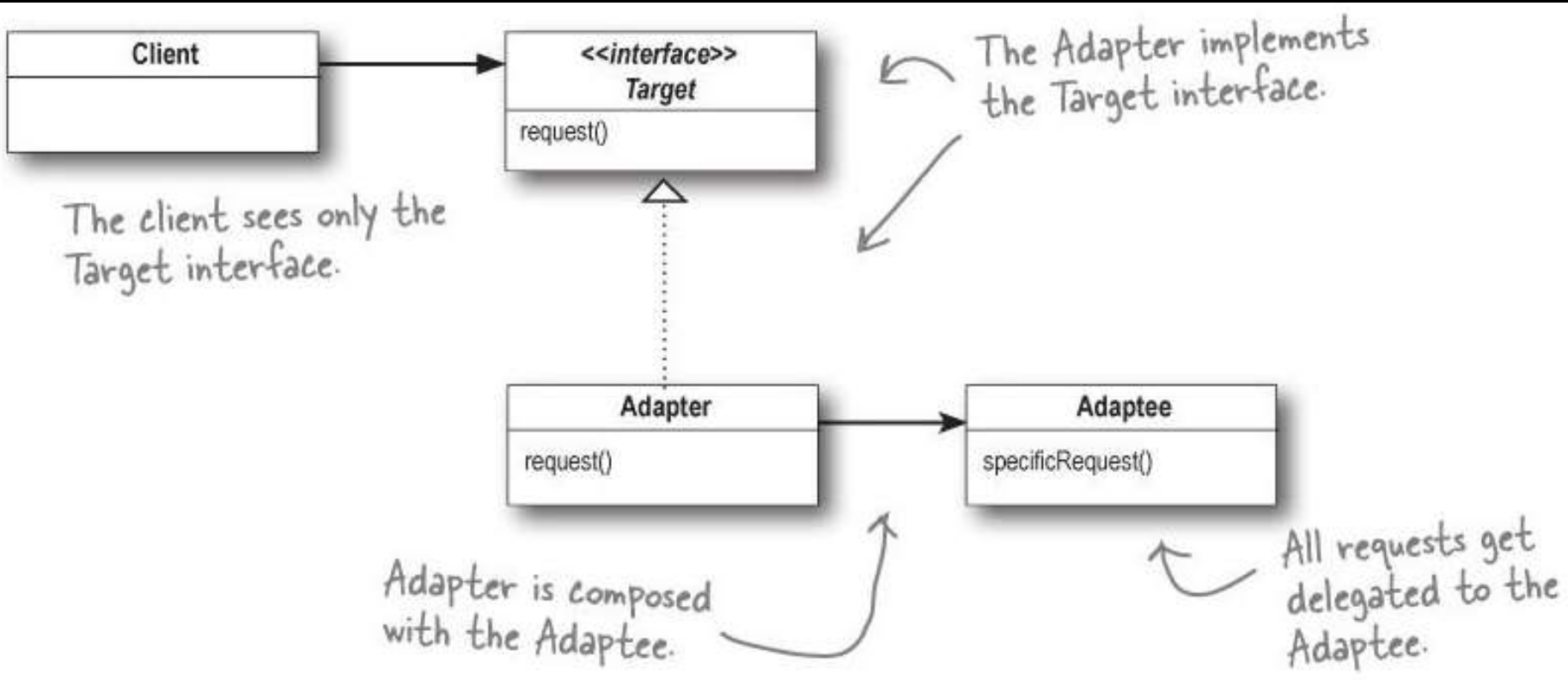
Adapter

Convert the interface of a class into another interface clients expect.

Adapter lets classes work together that couldn’t otherwise because of incompatible interfaces.



Adapter's motto: "An uncoupled client is a happy client!"


Client
<<interface>>
Target
request()
The Adapter implements the Target interface.
The client sees only the Target interface.
Adapter
request()
Adaptee
specificRequest()
Adapter is composed with the Adaptee.
All requests get delegated to the Adaptee.

First, you need to implement the interface of the type you're adapting to. This is the interface your client expects to see.

```
public class TurkeyAdapter implements Duck {
    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();
        }
    }
}
```

Next, we need to get a reference to the object that we are adapting; here we do that through the constructor.

Now we need to implement all the methods in the interface; the quack() translation between classes is easy: just call the gobble() method.

Even though both interfaces have a fly() method, Turkeys fly in short spurts – they can't do long-distance flying like ducks. To map between a Duck's fly() method and a Turkey's, we need to call the Turkey's fly() method five times to make up for it.

1. The client makes a request to the adapter by calling a method on it using the target interface.

2. The adapter translates the request into one or more calls on the adapter using the adaptee interface.

3. The client receives the results of the call and never knows there is an adapter doing the translation.

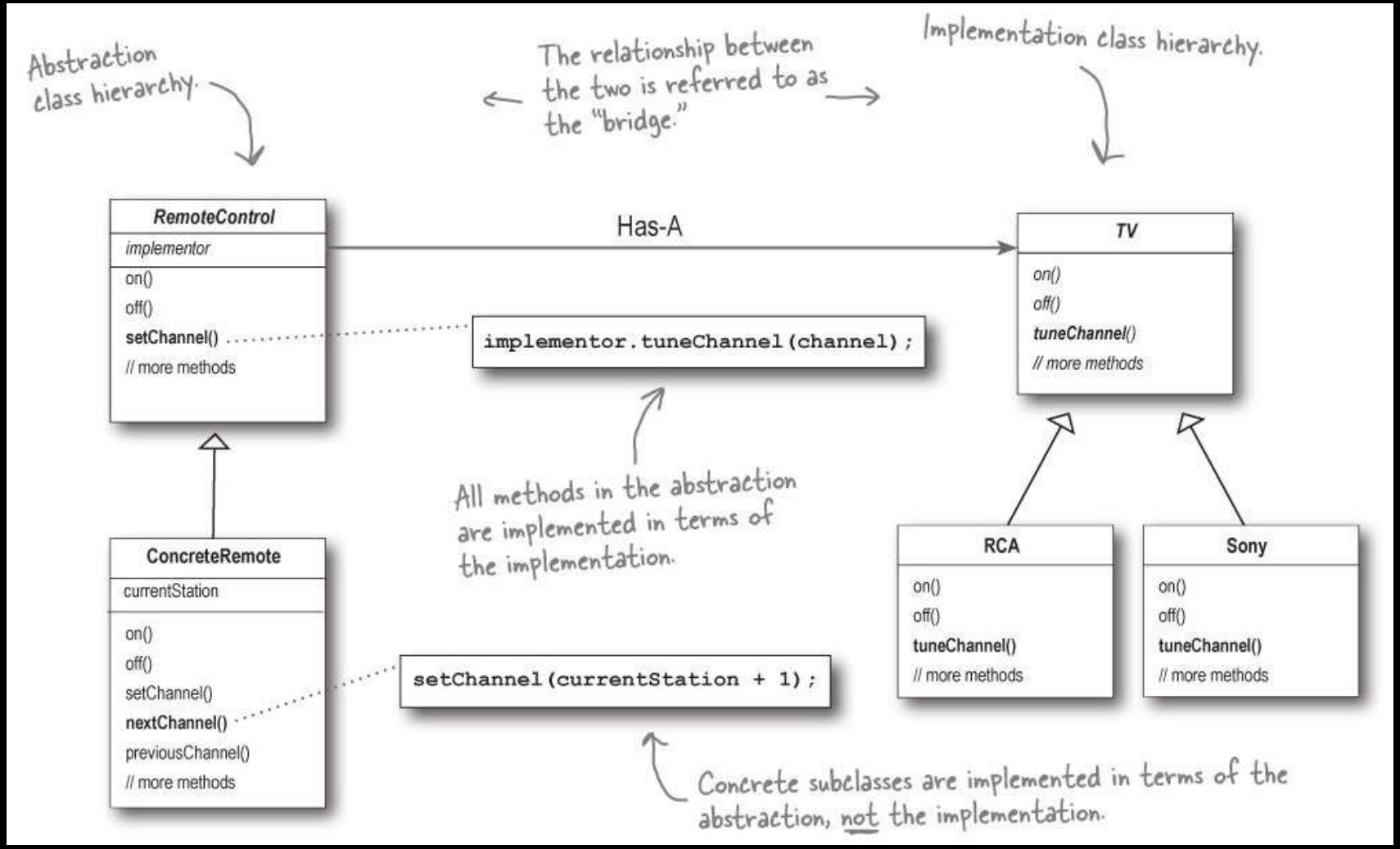
Bridge

Decouple an **abstraction** from its **implementation** so that the two can vary independently.

Changes to the concrete abstraction classes don’t affect the client.
Using composition and inheritance.

Bridge pattern is useful when a new version of software is brought out replacing an existing version, but the older version must still run for its existing client base. The client picks the version.

Example:
Graphics, where different displays have different capabilities and drivers


Abstraction class hierarchy.
The relationship between the two is referred to as the "bridge."
Implementation class hierarchy.
RemoteControl
implementor
on()
off()
setChannel()
// more methods
Has-A
TV
on()
off()
tuneChannel()
// more methods
implementor.tuneChannel(channel);
All methods in the abstraction are implemented in terms of the implementation.
ConcreteRemote
currentStation
on()
off()
setChannel()
nextChannel()
previousChannel()
// more methods
setChannel(currentStation + 1);
RCA
on()
off()
tuneChannel()
// more methods
Sony
on()
off()
tuneChannel()
// more methods
Concrete subclasses are implemented in terms of the abstraction, not the implementation.

Abstraction:
the interface that the client sees

Bridge:
the interface defining those parts of the abstraction that vary

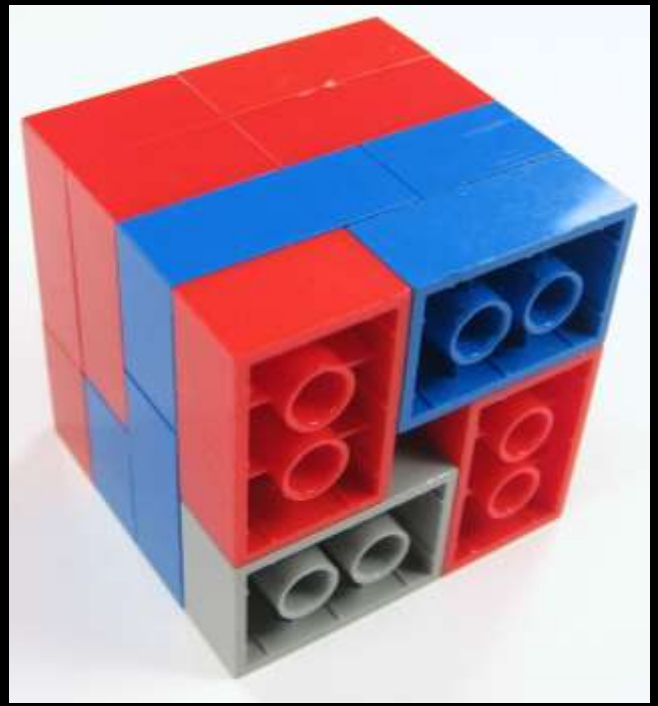
Abstraction *has* an instance of the bridge (composition instead of inheritance)

Composite

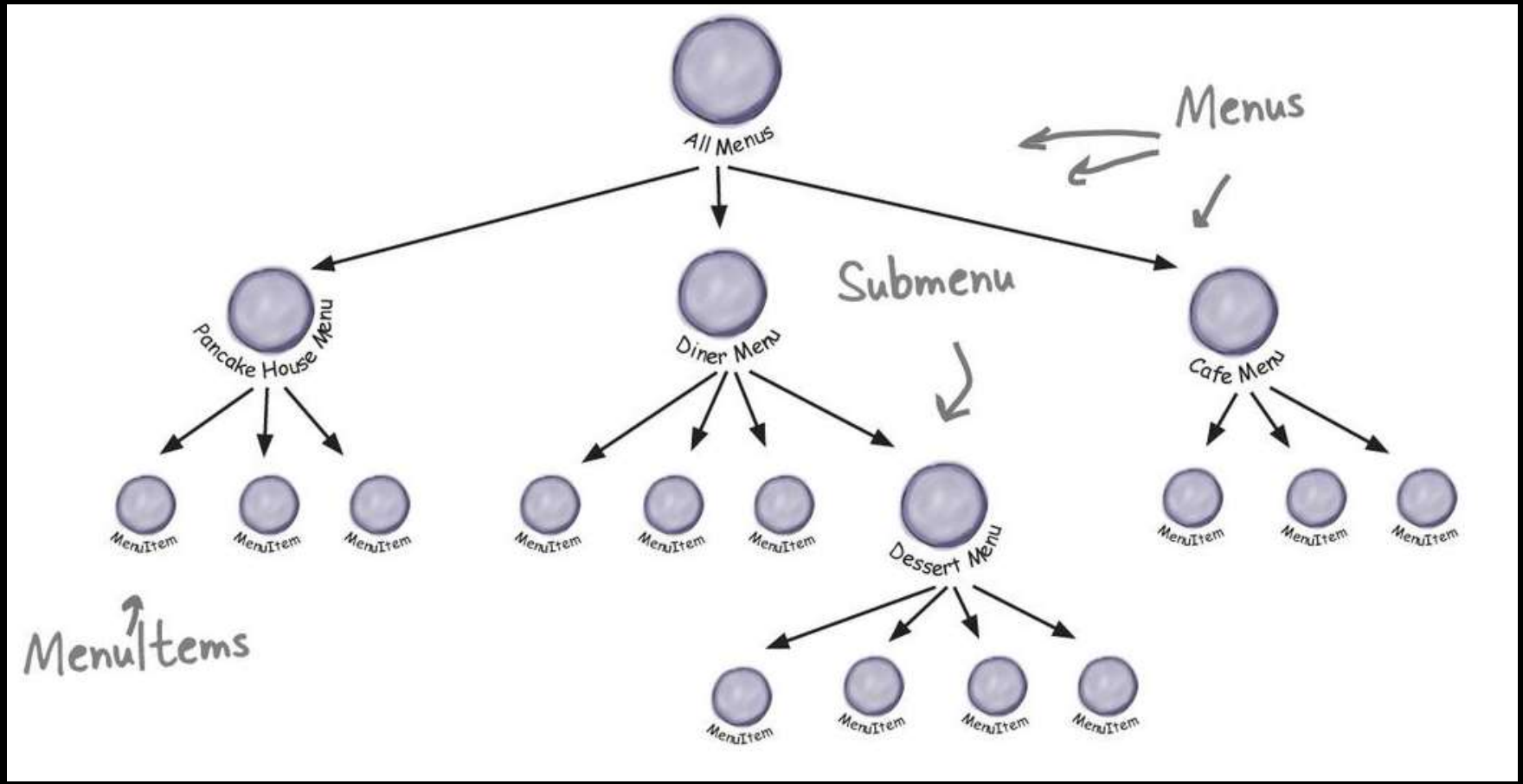
Compose objects into tree structures to represent part-whole hierarchies.

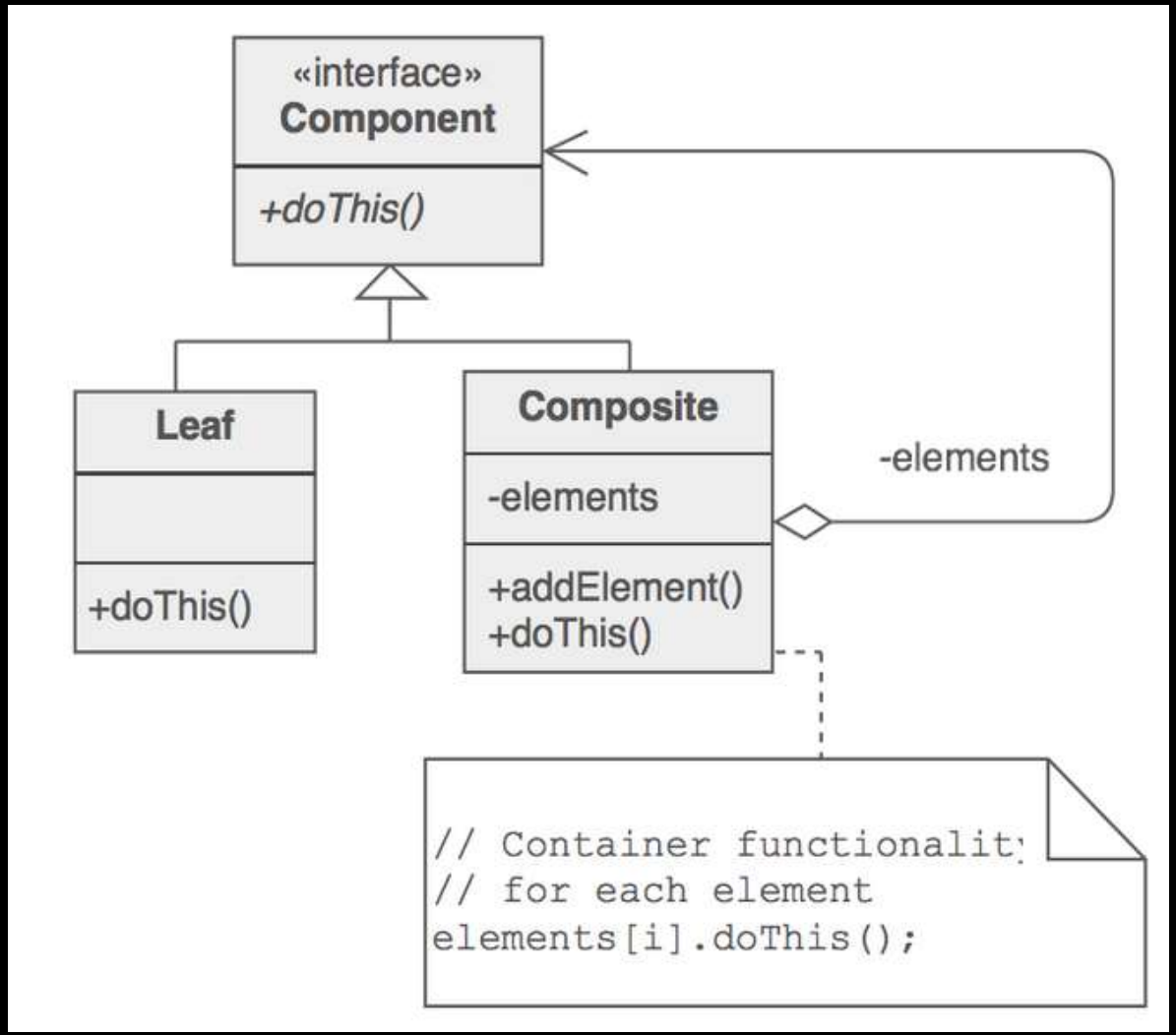
Composite lets clients treat individual objects and compositions of objects uniformly.

The same operations work on the whole or on its parts.



Components may be other composites or leaf nodes.




«interface»
Component
+doThis()
Leaf
+doThis()
Composite
-elements
+addElement()
+doThis()
-elements
// Container functionalit
// for each element
elements[i].doThis();

All container and containee classes declare an "is a" relationship to the interface.

All container classes declare a one-to-many "has a" relationship to the interface.

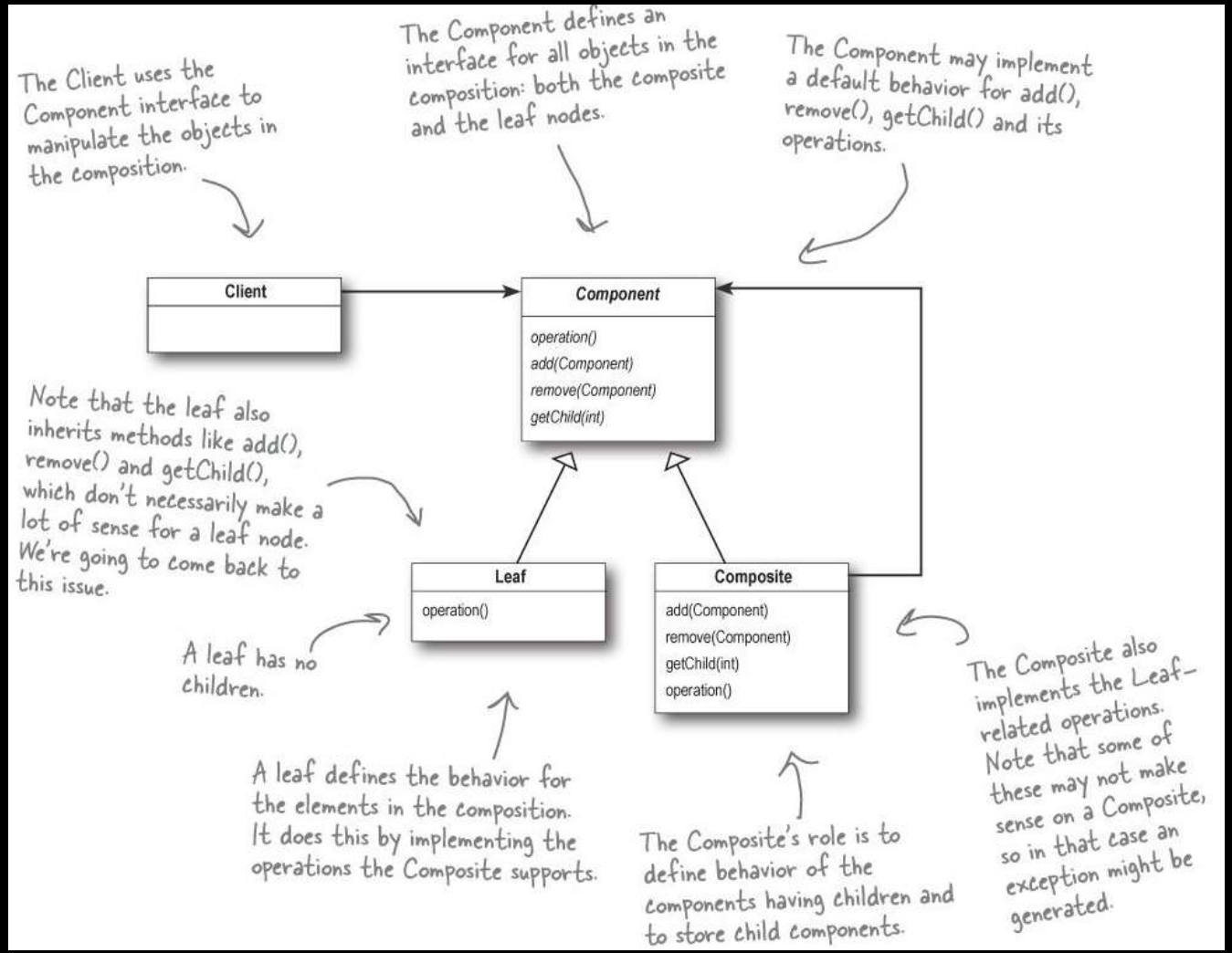
Container classes leverage polymorphism to delegate to their containee objects.

Collections and items implement the same interface (abstract base class).

The client accesses all of them using the interface.

Some of the interface operations only make sense for collections (*add*, *remove* …), some only for items.

Default method implementation: throw UnsupportedEx.
Alternative: don’t put them into the interface.


The Client uses the Component interface to manipulate the objects in the composition.
The Component defines an interface for all objects in the composition: both the composite and the leaf nodes.
The Component may implement a default behavior for add(), remove(), getChild() and its operations.
Client
Component
operation()
add(Component)
remove(Component)
getChild(int)
Note that the leaf also inherits methods like add(), remove() and getChild(), which don't necessarily make a lot of sense for a leaf node. We're going to come back to this issue.
Leaf
operation()
Composite
add(Component)
remove(Component)
getChild(int)
operation()
A leaf has no children.
A leaf defines the behavior for the elements in the composition. It does this by implementing the operations the Composite supports.
The Composite's role is to define behavior of the components having children and to store child components.
The Composite also implements the Leaf-related operations. Note that some of these may not make sense on a Composite, so in that case an exception might be generated.

We trade safety and Single Responsibility for transparency.

Now we do not need conditional code based on the type of the object.

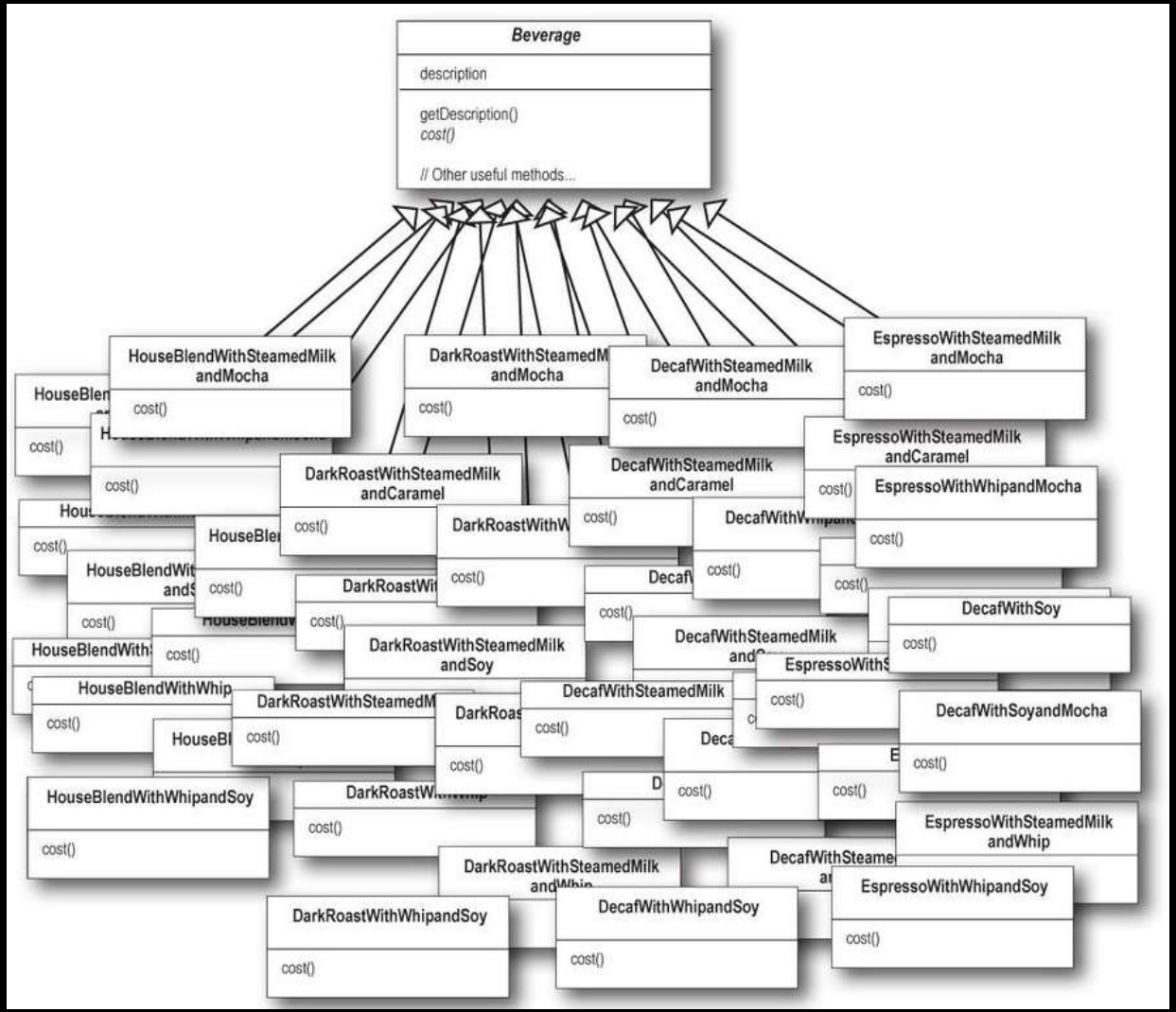
Examples for using the Composite Pattern:

+ Arithmetic expressions and numbers
+ Directories and files
+ Menus and items
+ Assemblies and components

Decorator

Attach additional responsibilities to an object dynamically.

Decorators provide a flexible alternative to subclassing for extending functionality.


Beverage
description
getDescription()
cost()
// Other useful methods...
HouseBlendWithSteamedMilk andMocha
cost()
DarkRoastWithSteamedMilk andMocha
cost()
DecafWithSteamedMilk andMocha
cost()
EspressoWithSteamedMilk andMocha
cost()
EspressoWithSteamedMilk andCaramel
cost()
DarkRoastWithSteamedMilk andCaramel
cost()
DecafWithSteamedMilk andCaramel
cost()
EspressoWithWhipandMocha
cost()
DecafWithSoy
cost()
DarkRoastWithSteamedMilk andSoy
DecafWithSoyandMocha
cost()
HouseBlendWithWhipandSoy
cost()
EspressoWithSteamedMilk andWhip
DarkRoastWithWhipandSoy
cost()
DecafWithWhipandSoy
cost()
EspressoWithWhipandSoy
cost()

Composition adds behaviour dynamically at runtime using composition and delegation instead of adding it statically with inheritance.

But it uses inheritance to have the same type as the wrapped object thus allowing chaining.

Decorating gives your objects new responsibilities without making any code changes to the underlying classes.

Open-Closed Principle:

Classes should be open for extensions,

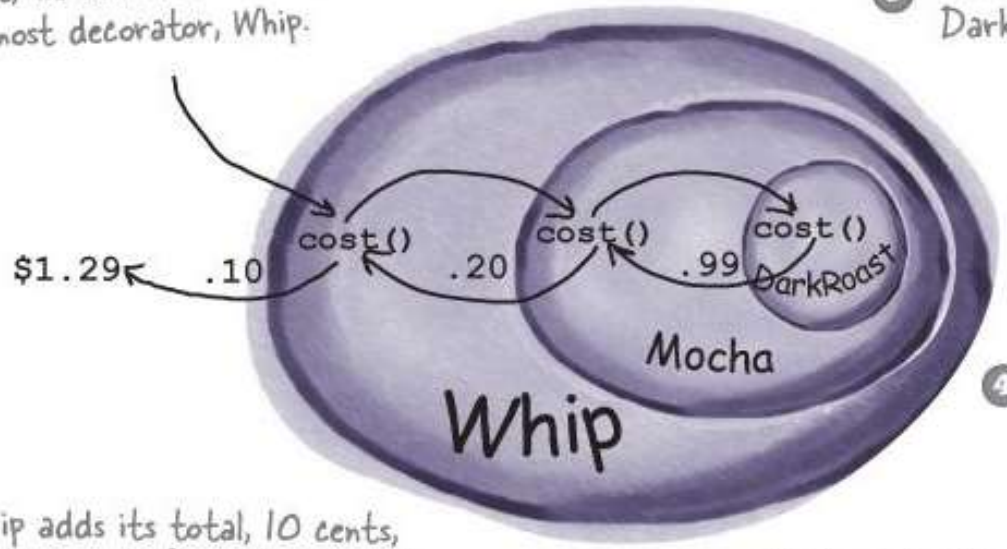
but closed for modifications.

(You'll see how in a few pages.)

2 Whip calls cost() on Mocha.

1 First, we call cost() on the outmost decorator, Whip.

3 Mocha calls cost() on DarkRoast.



4 DarkRoast returns its cost, 99 cents.

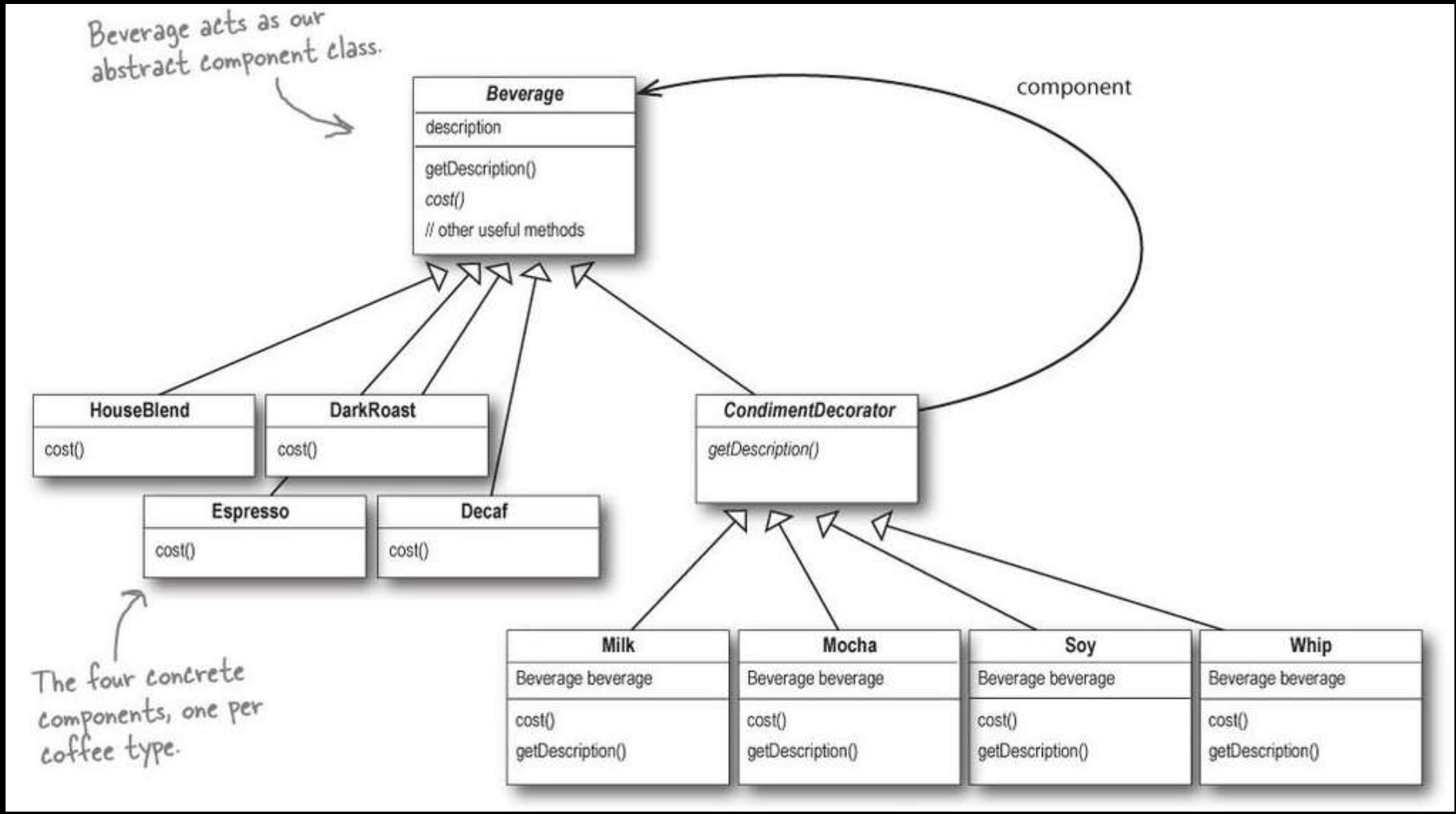
6 Whip adds its total, 10 cents, to the result from Mocha, and returns the final result—$1.29.

5 Mocha adds its cost, 20 cents, to the result from DarkRoast, and returns the new total, $1.19.

Decorators have the same base class as the objects they decorate (allows chaining, giving us a Fluent Interface).

The decorated object can be used in place of the original (wrapped) object (Liskov's Substitution principle).

You can use multiple decorators to wrap an object.


Beverage acts as our abstract component class.
Beverage
description
getDescription()
cost()
// other useful methods
component
HouseBlend
cost()
DarkRoast
cost()
CondimentDecorator
getDescription()
Espresso
cost()
Decaf
cost()
The four concrete components, one per coffee type.
Milk
Beverage beverage
cost()
getDescription()
Mocha
Beverage beverage
cost()
getDescription()
Soy
Beverage beverage
cost()
getDescription()
Whip
Beverage beverage
cost()
getDescription()

Mocha is a decorator, so we extend CondimentDecorator.

Remember, CondimentDecorator extends Beverage.

```
public class Mocha extends CondimentDecorator {
    Beverage beverage;

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

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

    public double cost() {
        return beverage.cost() + .20;
    }
}
```

We're going to instantiate Mocha with a reference to a Beverage using:

(1) An instance variable to hold the beverage we are wrapping.

(2) A way to set this instance variable to the object we are wrapping. Here, we're going to pass the beverage we're wrapping to the decorator's constructor.

We want our description to not only include the beverage – say "Dark Roast" – but also to include each item decorating the beverage (for instance, "Dark Roast, Mocha"). So we first delegate to the object we are decorating to get its description, then append ", Mocha" to that description.

Now we need to compute the cost of our beverage with Mocha. First, we delegate the call to the object we're decorating, so that it can compute the cost; then, we add the cost of Mocha to the result.

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

    Beverage beverage2 = new DarkRoast();
    beverage2 = new Mocha(beverage2);
    beverage2 = new Mocha(beverage2);
    beverage2 = new Whip(beverage2);
    System.out.println(beverage2.getDescription()
            + " $" + beverage2.cost());

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

Order up an espresso, no condiments, and print its description and cost.

Make a DarkRoast object.

Wrap it with a Mocha.

Wrap it in a second Mocha.

Wrap it in a Whip.

Finally, give us a HouseBlend with Soy, Mocha, and Whip.

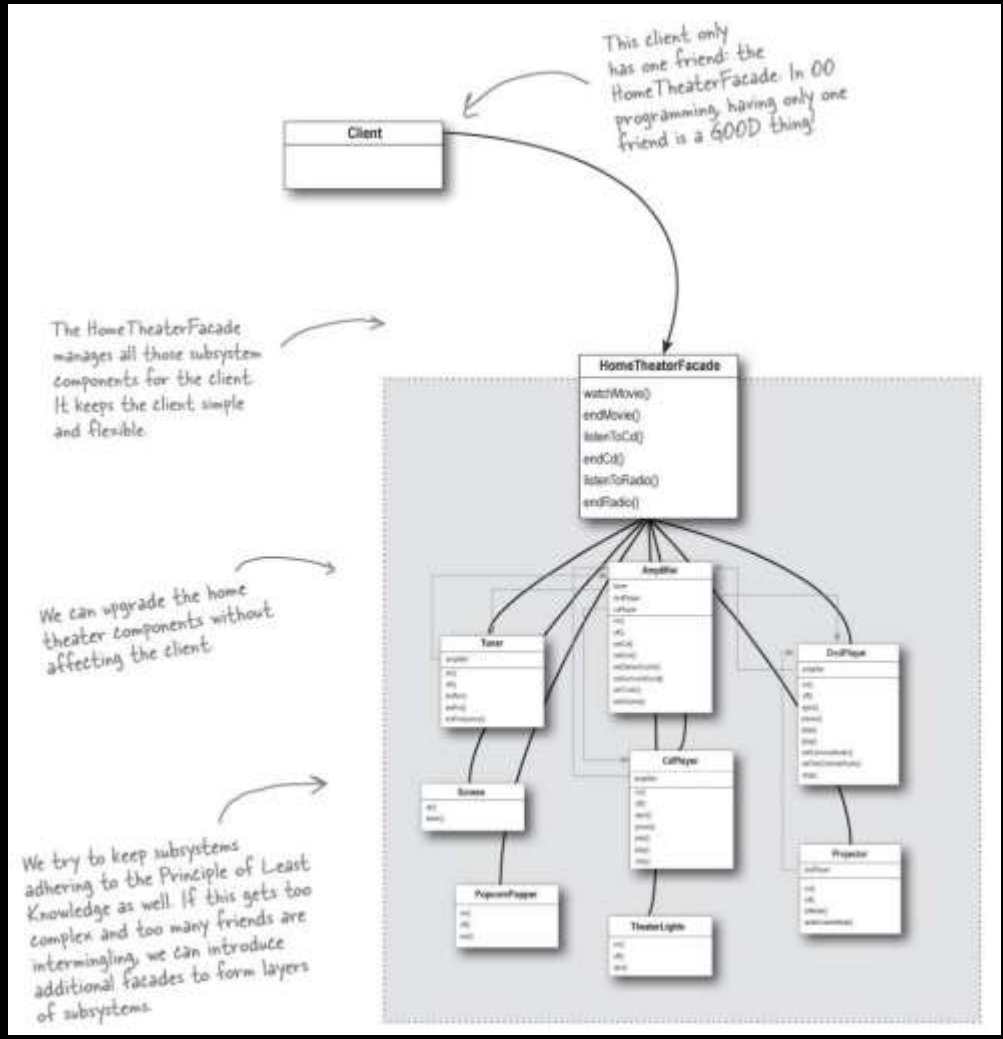
The decorator adds its own behaviour before and / or after delegating to the object it decorates.

Decorator lets you change the skin of an object, Strategy lets you change the guts.

Facade

Provide a unified interface to a set of interfaces in a subsystem.

Façade defines a higher-level interface that makes the subsystem easier to use.


This client only has one friend: the HomeTheaterFacade. In OO programming, having only one friend is a GOOD thing!
Client
The HomeTheaterFacade manages all those subsystem components for the client. It keeps the client simple and flexible.
HomeTheaterFacade
watchMovie()
endMovie()
listenToCd()
endCd()
listenToRadio()
endRadio()
We can upgrade the home theater components without affecting the client.
We try to keep subsystems adhering to the Principle of Least Knowledge as well. If this gets too complex and too many friends are intermingling, we can introduce additional facades to form layers of subsystems.

Principle of Least Knowledge (“Demeter’s Law”)

Reduce interactions between objects to a few “friends”.
Talk only to your immediate friends!

Prevents us from designs that have a large number of coupled classes with cascading changes.

Less fragile => easier to maintain
Less complex => easier to understand

Interface Segregation Principle

Many client-specific interfaces are better than one general-purpose interface. Clients shouldn't be forced to implement interfaces they don't use.

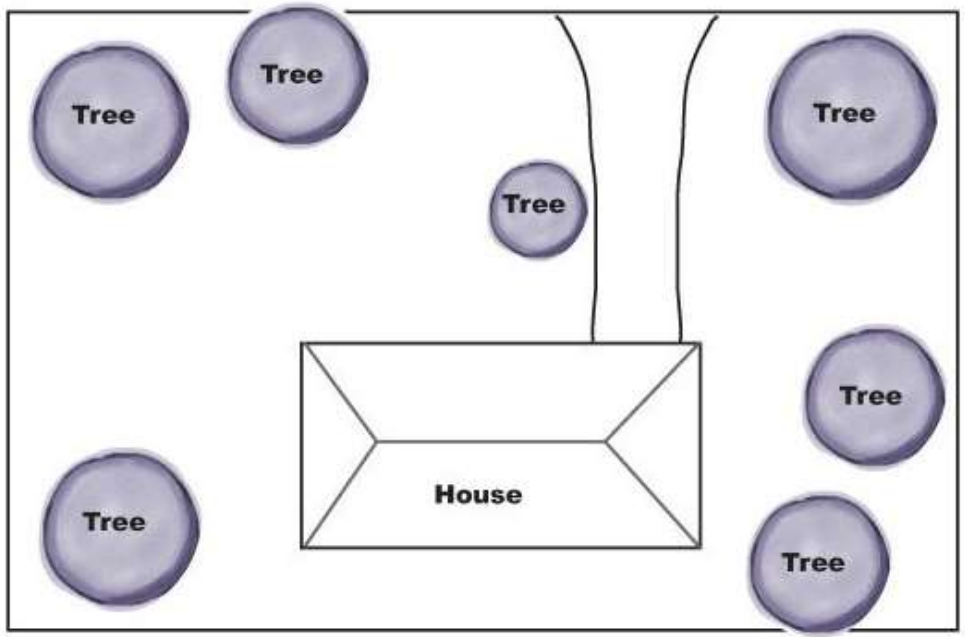
If you find yourself creating interfaces that don't get fully implemented in its clients, then that's a good sign that you're violating the ISP.

Example of ISP violation: ASP.NET Membership Provider

Flyweight

Use sharing to support a large number of fine-grained objects efficiently.

**Store the state-independent (intrinsic) part in the Flyweight object,
the state-dependent part in a collection object at client side.**

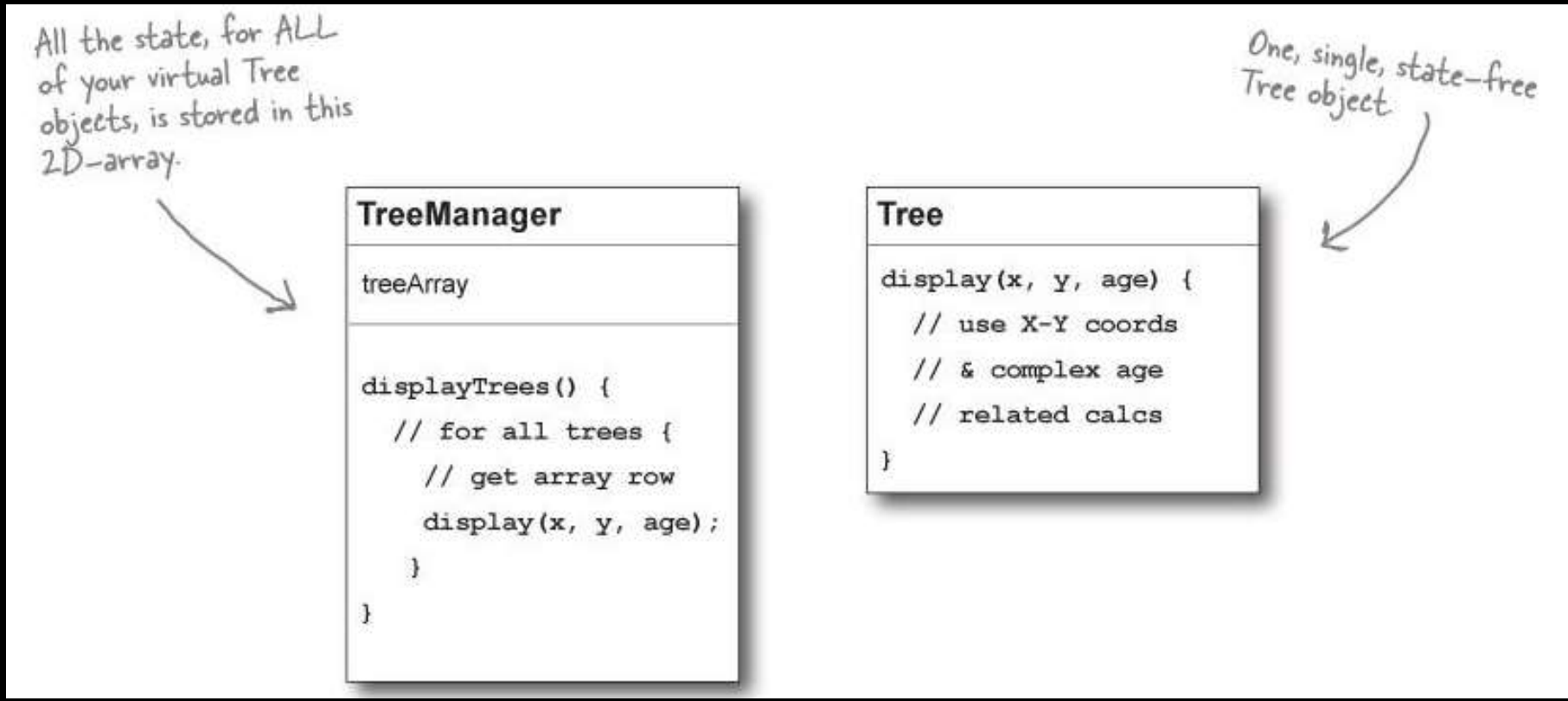


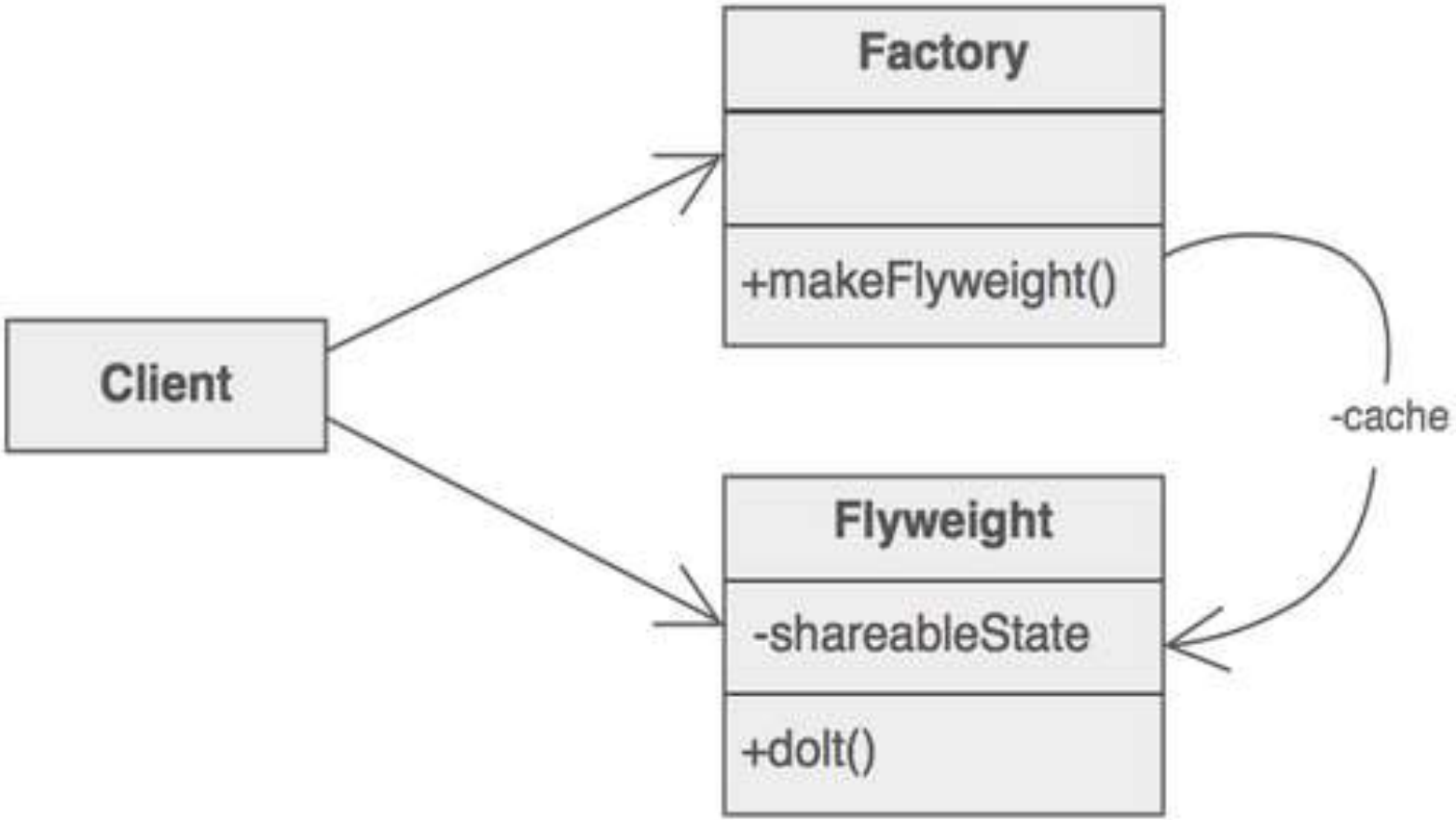
Each Tree instance maintains its own state.

Tree

xCoord
yCoord
age

```
display()  {
  // use X-Y coords
  // & complex age
  // related calcs
}
```


All the state, for ALL of your virtual Tree objects, is stored in this 2D-array.
TreeManager
treeArray
displayTrees() {
// for all trees {
// get array row
display(x, y, age);
}
}
One, single, state-free Tree object
Tree
display(x, y, age) {
// use X-Y coords
// & complex age
// related calcs
}


Factory
+makeFlyweight()
Client
-cache
Flyweight
-shareableState
+dolt()

Reduces the number of objects, saving memory (Example: browser caches shared photos)

Centralises state for many “virtual” objects into a single location.

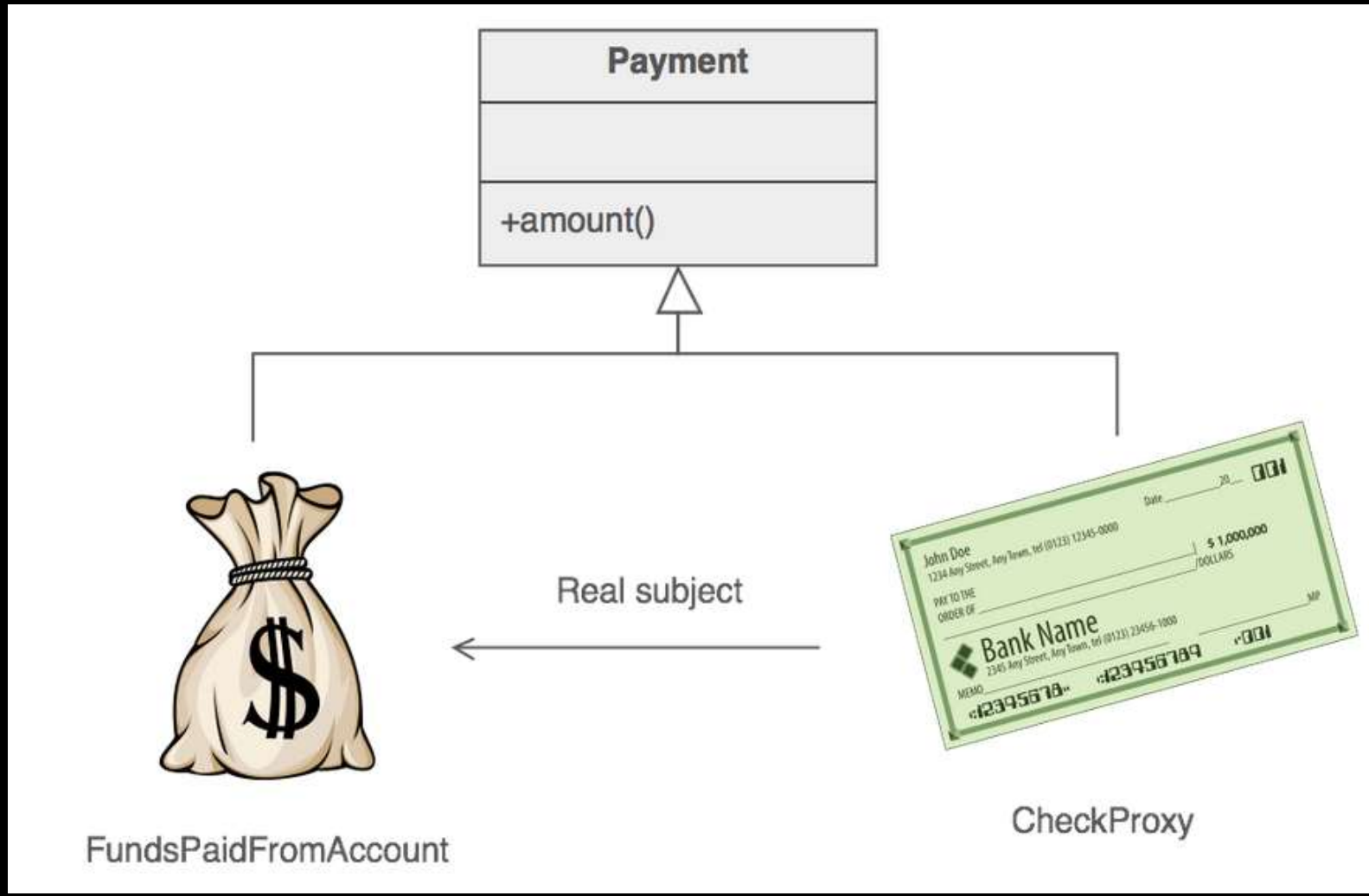
Drawback: objects cannot behave independently

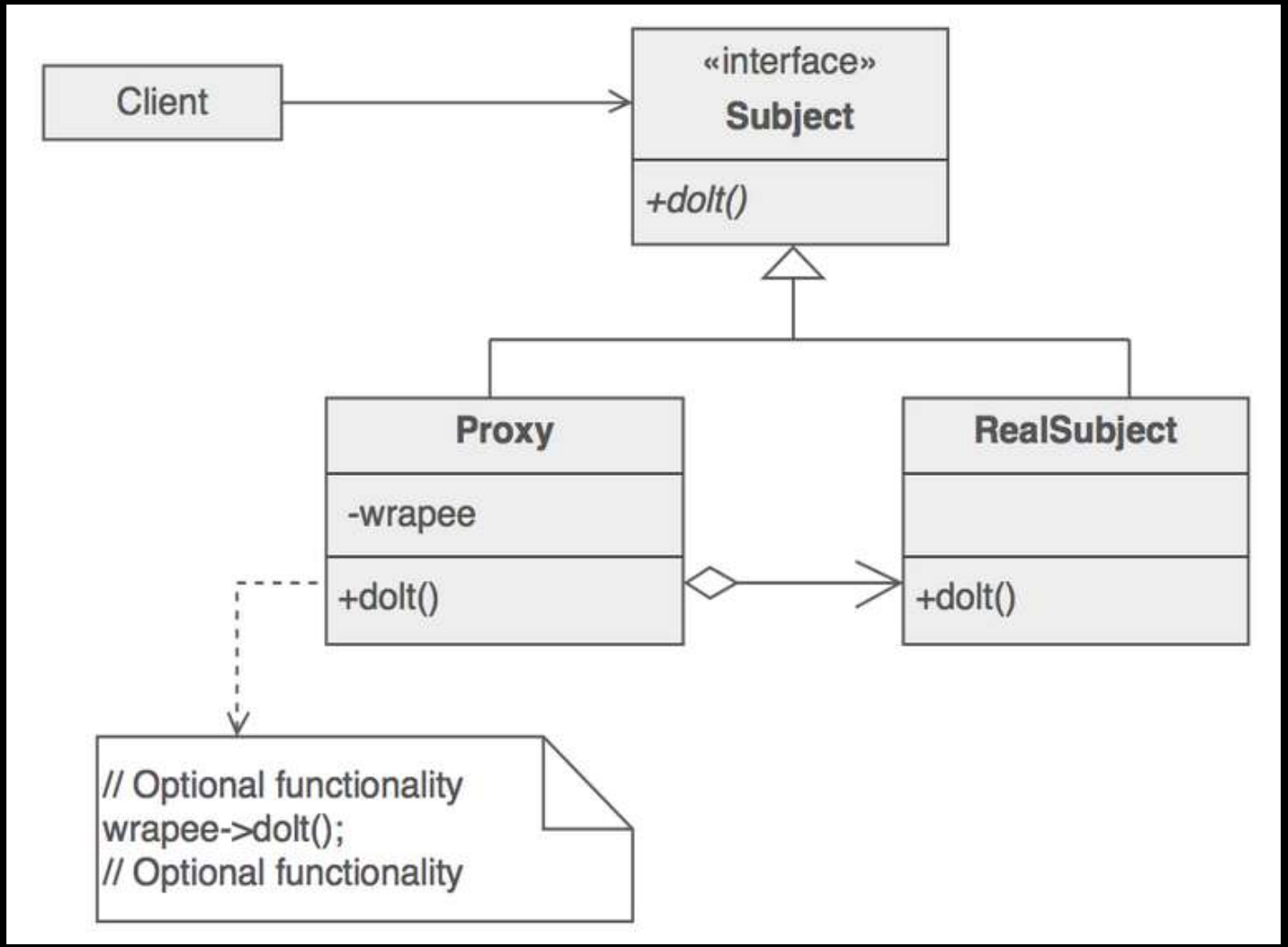
Proxy

Provide a surrogate or placeholder for another object to control access to it.

Uses composition and delegation.

Can also control creation of actual object.


Payment
+amount()
Real subject
John Doe
1234 Any Street, Any Town, tel (0123) 12345-0000
$ 1,000,000
PAY TO THE ORDER OF
Bank Name
2345 Any Street, Any Town, tel (0123) 23456-1000
MEMO
FundsPaidFromAccount
CheckProxy


Client
«interface»
Subject
+dolt()
Proxy
-wrapee
+dolt()
RealSubject
+dolt()
// Optional functionality
wrapee->dolt();
// Optional functionality

Proxy Types

+ Remote Proxy controls access to a remote object
+ Virtual Proxy controls access to an expensive resource that may be created only at first use
+ Protection Proxy controls access to a resource based on access rights
+ Firewall Proxy controls access to network resources
+ Synchronization Proxy provides safe access to a subject across multiple threads
+ Smart Clients change requests (e.g. adding logging)

Proxy Examples

+ WCF Client

+ Cache storing db objects

+ Service monitor

+ UI grid to scroll through images

Structural Pattern Comparison

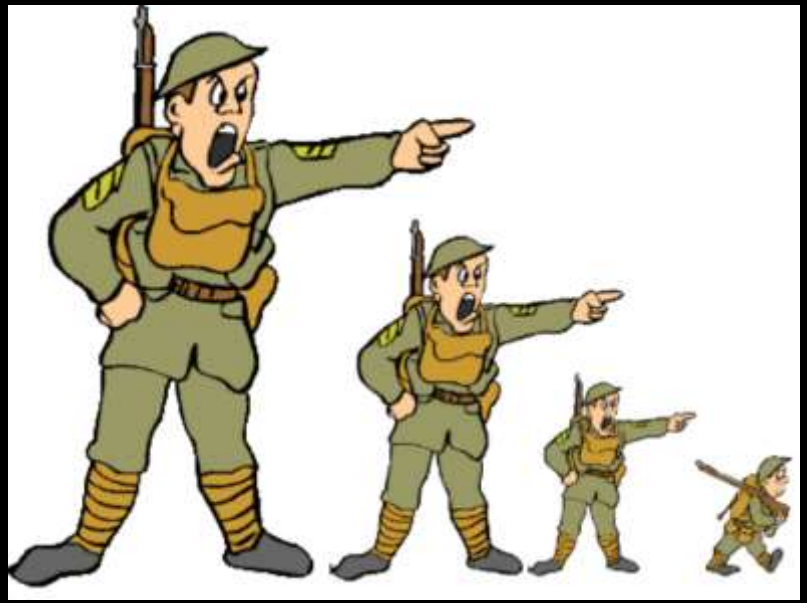
- Adapter wraps an object to change its interface
- Proxy wraps an object to control its access; same interf.
- Decorator wraps an object to add new behaviour dyn.
- Facade wraps a set of objects to simplify its interface
- Bridge abstracts the varying parts of an object
- Composite treats objects and collections uniformly

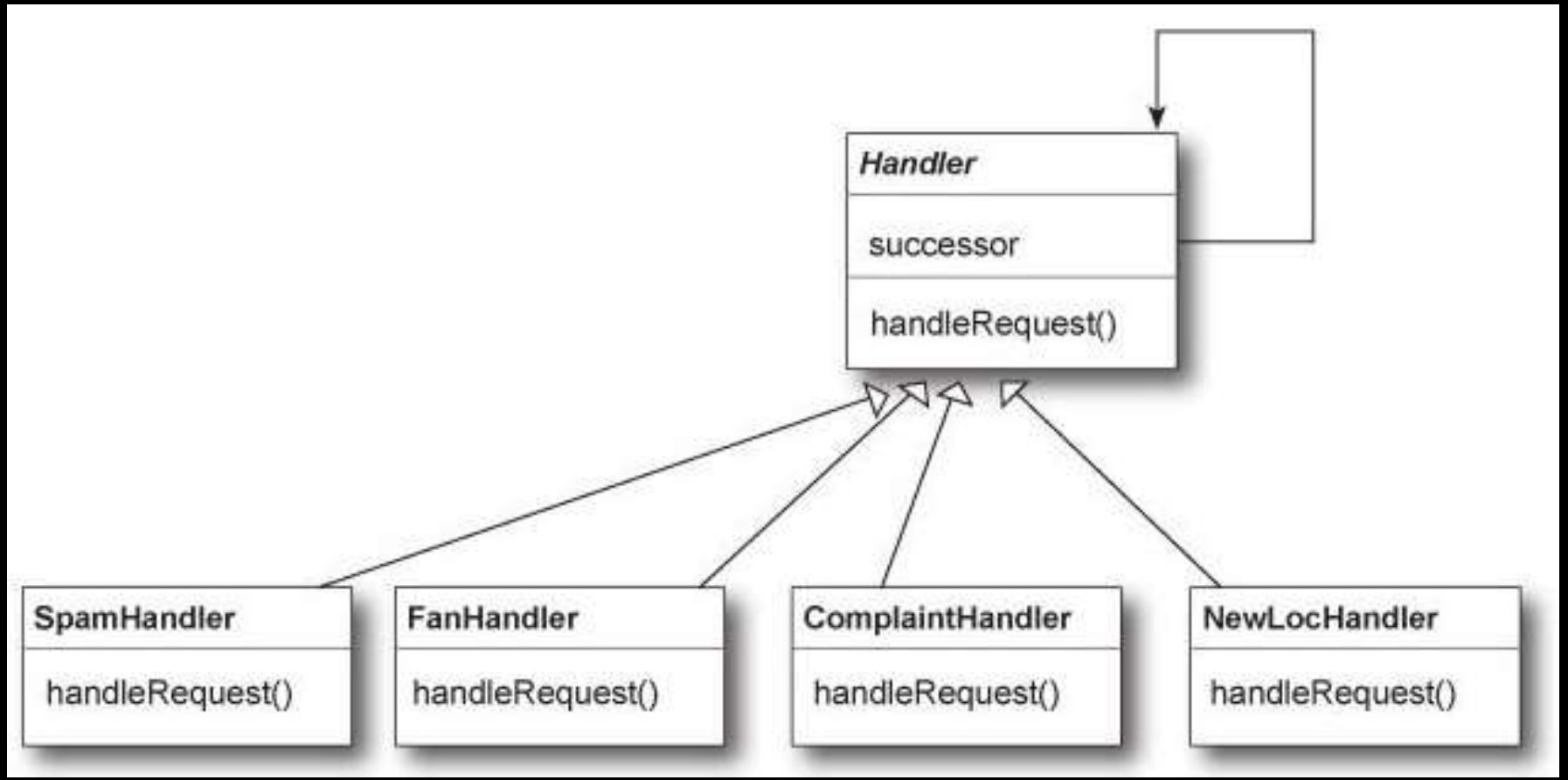
Behavioural Patterns

Chain of Responsibility

Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request.

Chain the receiving objects and pass the request along the chain until an object handles it.




Handler
successor
handleRequest()
SpamHandler
handleRequest()
FanHandler
handleRequest()
ComplaintHandler
handleRequest()
NewLocHandler
handleRequest()

Each object in the chain acts as a handler and has a successor object.

If it can handle the request, it does;
otherwise, it forwards the request to its successor.

A single object doesn’t know the chain structure,
that keeps it simple.

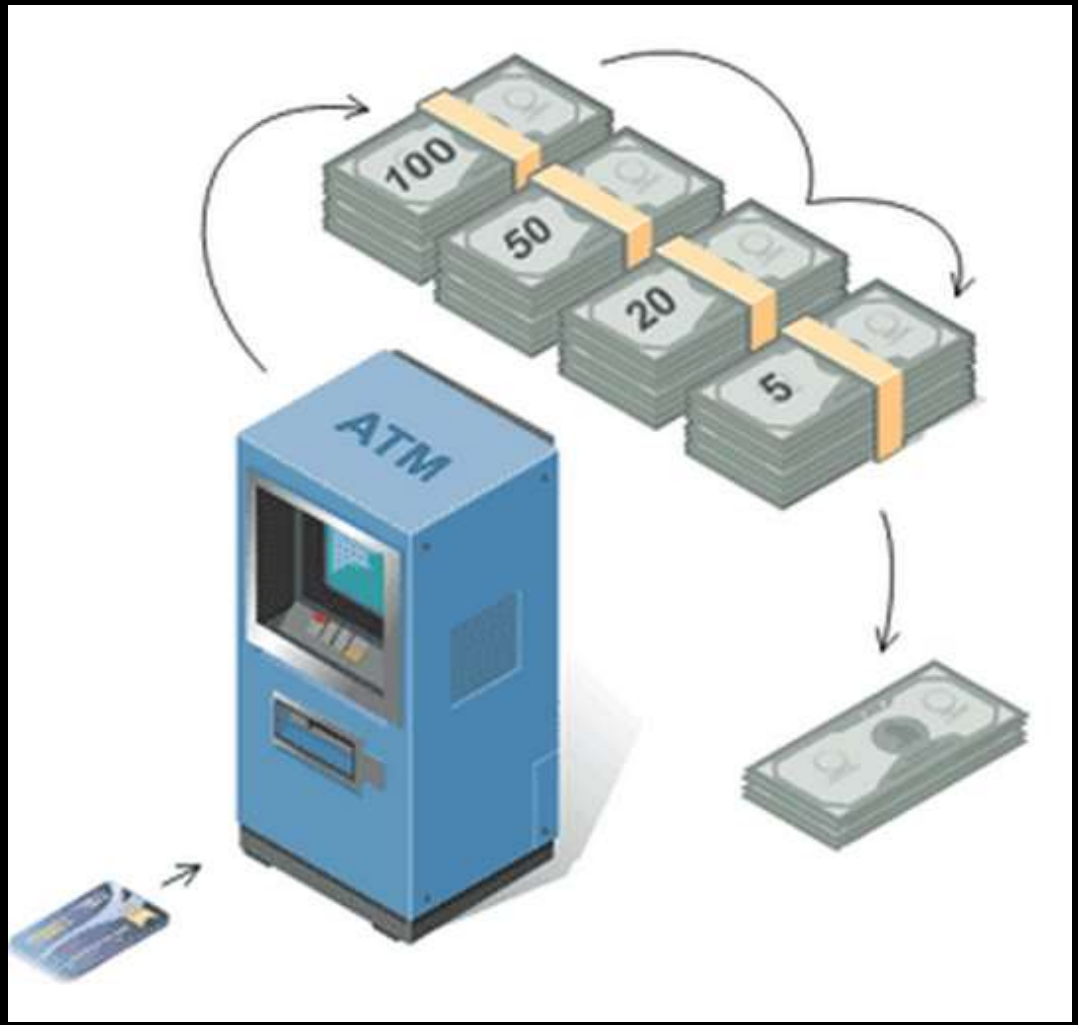
Decouples the sender of a request from its receiver.

Allows you to add or remove responsibilities dynamically by changing the members
or the order of the chain.

Examples:

+ passing mouse clicks or keyboard events to UI controls (Windows Forms, WPF, Cocoa, Cocoa Touch)

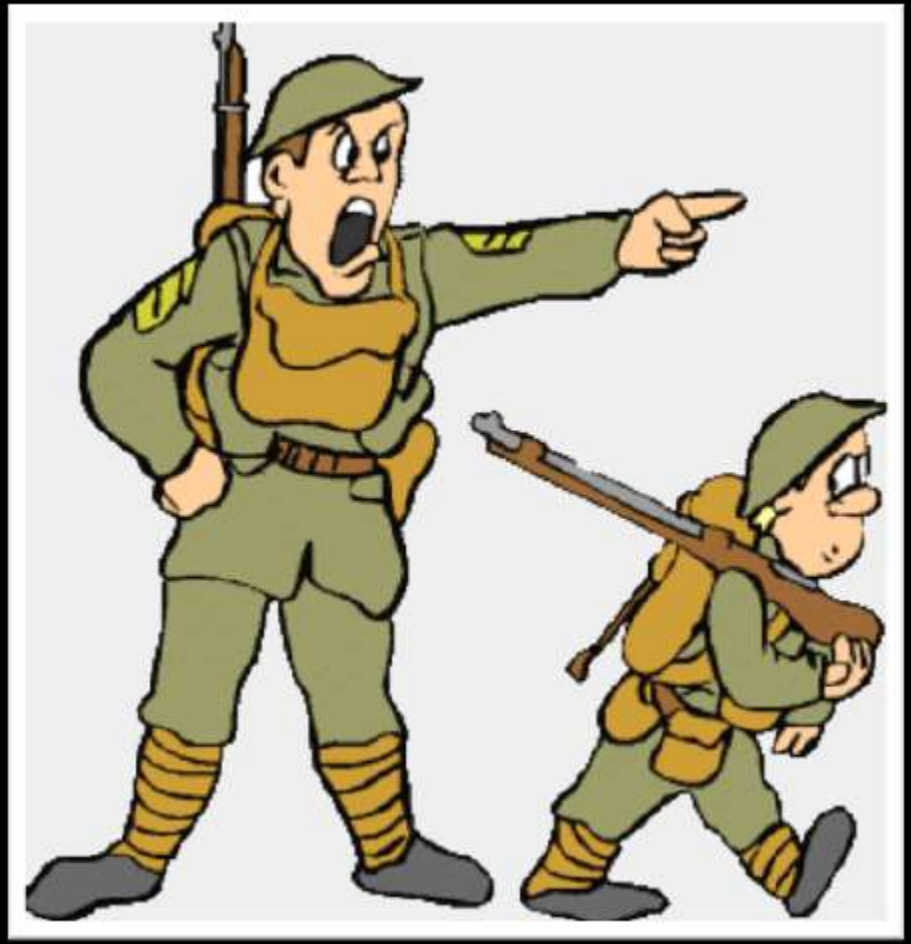
+ Courier Service sending parcels in various sizes


100
50
20
5
ATM

Command

Encapsulate a request as an object letting you parametrize clients with different requests.

Decouple the requestor of an action from the object that knows how to perform it.



The client that creates a command is not the same client that executes it.

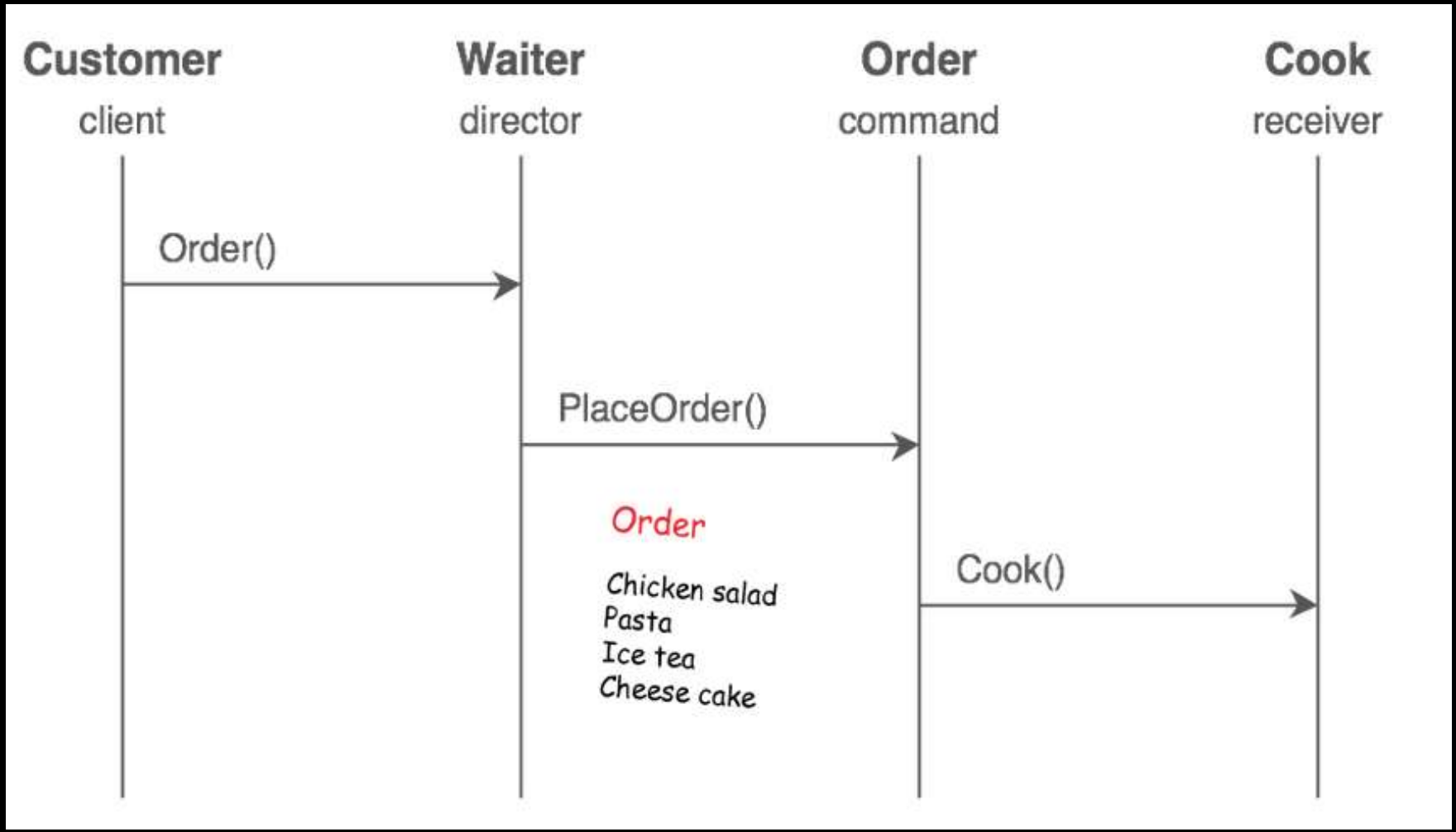
This separation provides flexibility in the timing and sequencing of commands.

Materializing commands as objects means they can be passed, staged, shared, loaded in a table, and otherwise instrumented or manipulated like any other object.

❶ You, the Customer, give the Waitress your Order.

❷ The Waitress takes the Order, places it on the order counter, and says "Order up!"

❸ The Short-Order Cook prepares your meal from the Order.


Customer
client
Waiter
director
Order
command
Cook
receiver
Order()
PlaceOrder()
Order
Chicken salad
Pasta
Ice tea
Cheese cake
Cook()

Command objects can be thought of as "tokens" that are created by one client that knows what needs to be done, and passed to another client that has the resources for doing it.

An action or a set of actions and a receiver are packaged into a command object that implements *Execute().*

The command can support *Undo()* **that mirrors** *Execute().* **It restores the receiver object to its previous state.**

For multiple undos, stack all commands and run their *Undo()* **operations in reverse order.**
Redo() **reruns** *Execute().*

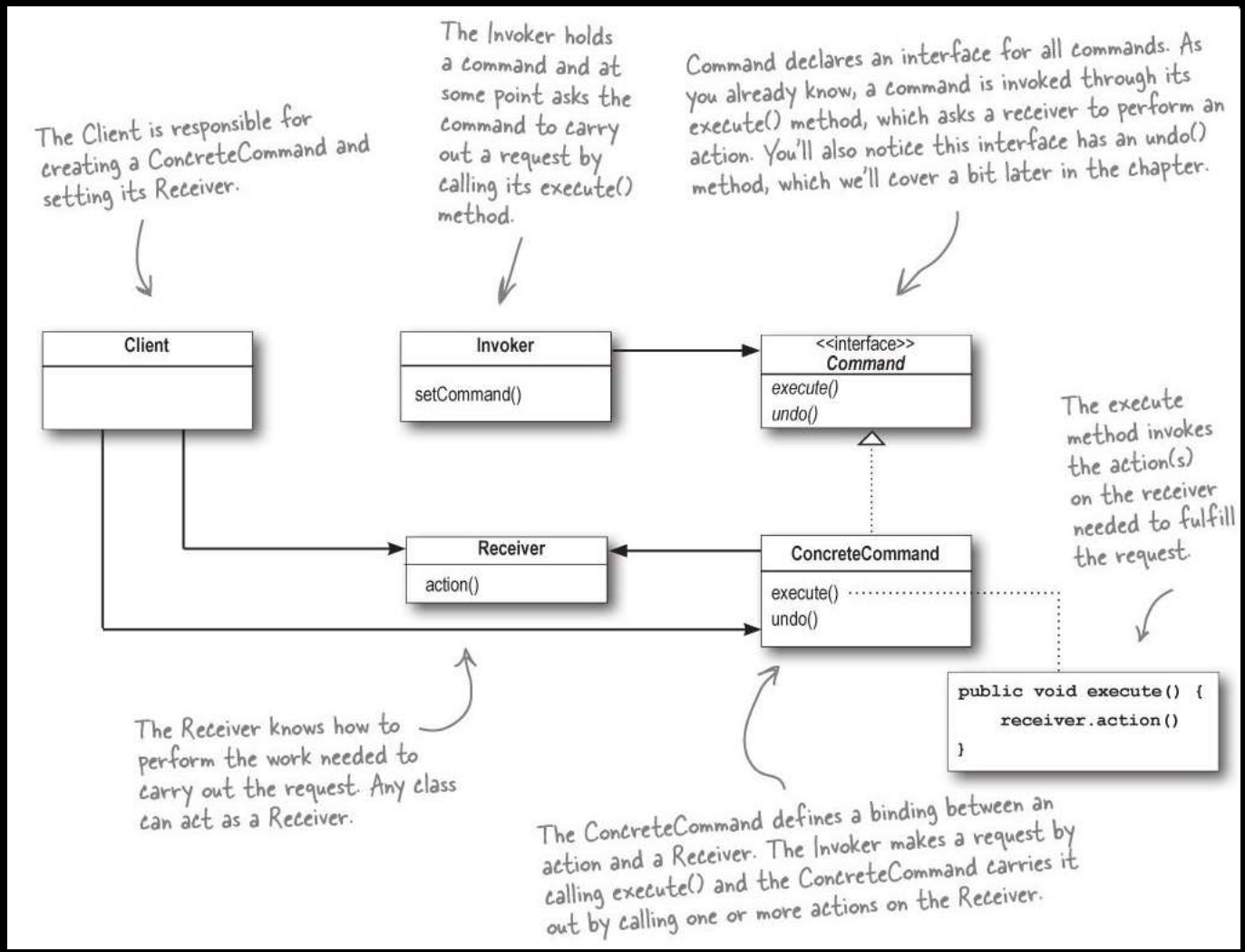
We can use composition to build macro commands.

A common interface for a set of different receivers.

Strife for “dumb” command classes that just invoke an action on a receiver.

Leveraging the command pattern, requests can be queued, logged, measured …

Command pattern is also useful for transactional systems.


The Client is responsible for creating a ConcreteCommand and setting its Receiver.
The Invoker holds a command and at some point asks the command to carry out a request by calling its execute() method.
Command declares an interface for all commands. As you already know, a command is invoked through its execute() method, which asks a receiver to perform an action. You'll also notice this interface has an undo() method, which we'll cover a bit later in the chapter.
Client
Invoker
setCommand()
<<interface>>
Command
execute()
undo()
Receiver
action()
ConcreteCommand
execute()
undo()
The execute method invokes the action(s) on the receiver needed to fulfill the request.
public void execute() {
receiver.action()
}
The Receiver knows how to perform the work needed to carry out the request. Any class can act as a Receiver.
The ConcreteCommand defines a binding between an action and a Receiver. The Invoker makes a request by calling execute() and the ConcreteCommand carries it out by calling one or more actions on the Receiver.

```
public class StereoOnWithCDCommand implements Command {
    Stereo stereo;

    public StereoOnWithCDCommand(Stereo stereo) {
        this.stereo = stereo;
    }

    public void execute() {
        stereo.on();
        stereo.setCD();
        stereo.setVolume(11);
    }
}
```

Just like the LightOnCommand, we get passed the instance of the stereo we are going to be controlling and we store it in a local instance variable.

To carry out this request, we need to call three methods on the stereo: first, turn it on, then set it to play the CD, and finally set the volume to 11. Why 11? Well, it's better than 10, right?

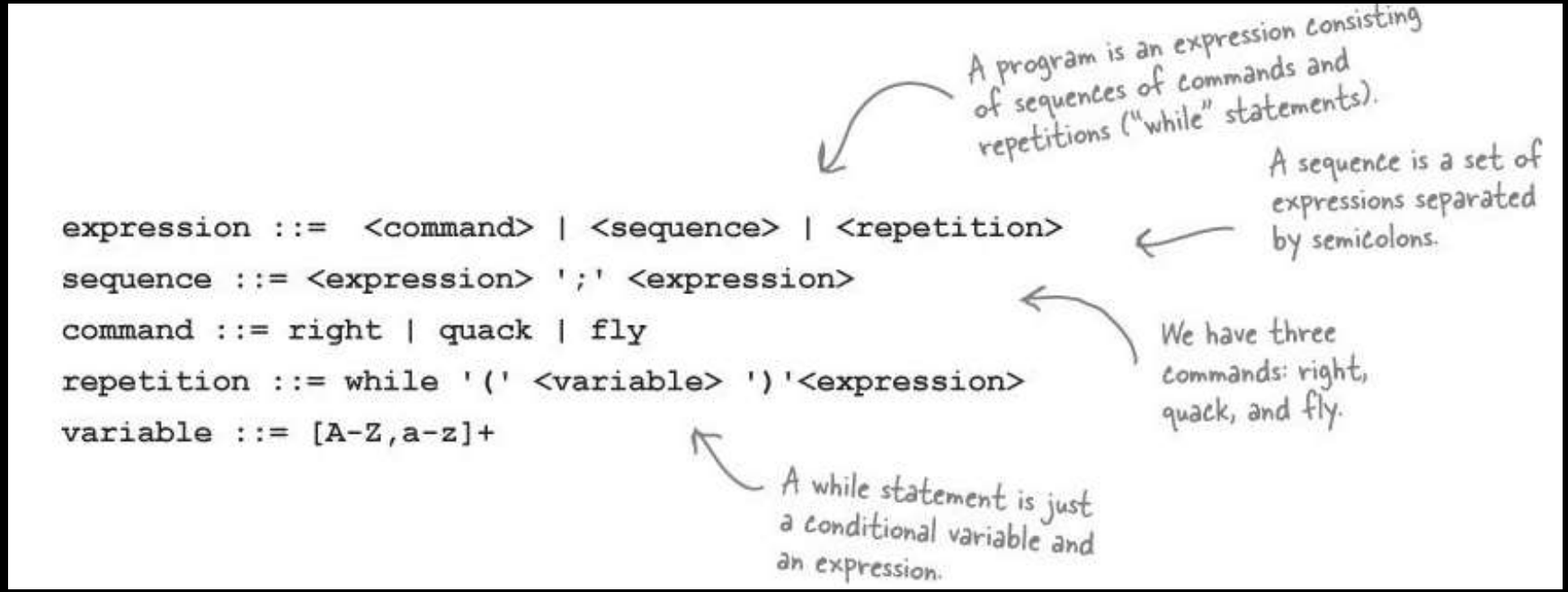
Interpreter

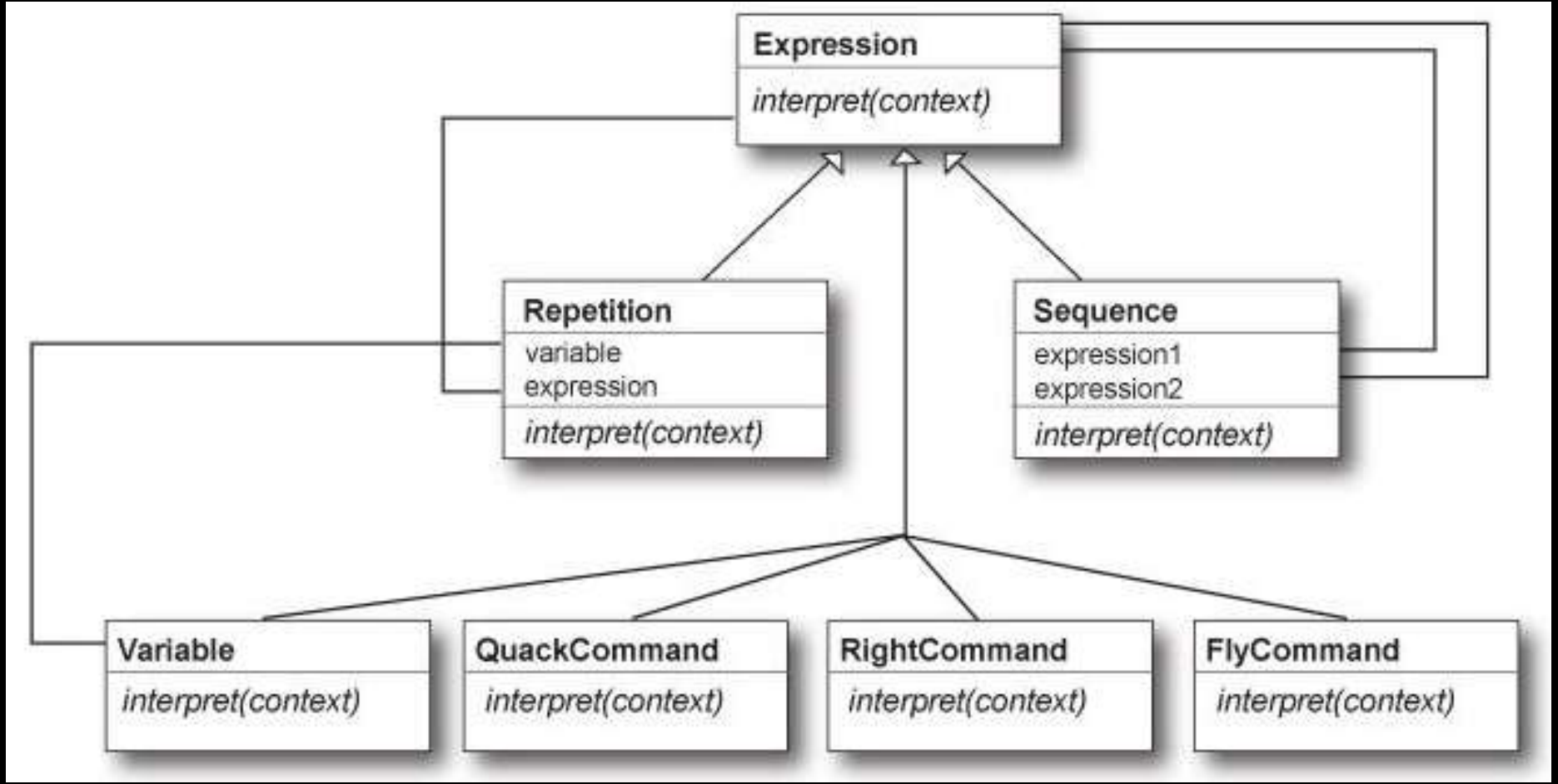
Given a language.

Define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in this language.

Example:

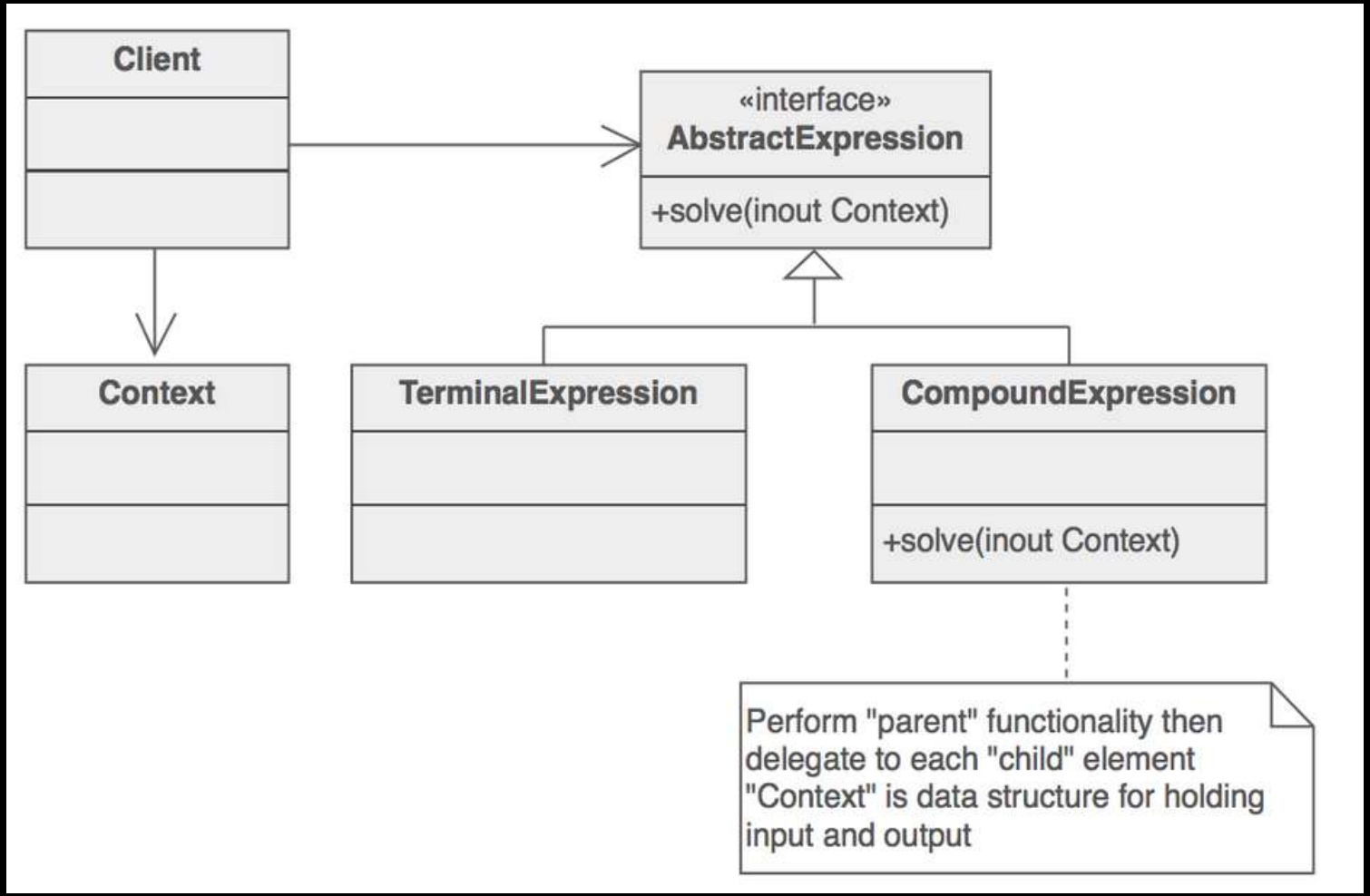
right;
while (daylight) fly;
quack




Expression
interpret(context)
Repetition
variable
expression
interpret(context)
Sequence
expression1
expression2
interpret(context)
Variable
interpret(context)
QuackCommand
interpret(context)
RightCommand
interpret(context)
FlyCommand
interpret(context)

Map a domain to a language,
the language to a grammar, and the
grammar to a hierarchical object-oriented design.

Each class represents one rule in the language.


Client
«interface»
AbstractExpression
+solve(inout Context)
Context
TerminalExpression
CompoundExpression
+solve(inout Context)
Perform "parent" functionality then delegate to each "child" element
"Context" is data structure for holding input and output

Useful for scripting and programming languages.

Use interpreter for a simple grammar;
for a complex grammar with a high number of rules a parser / compiler generator might be more appropriate.

The Context object encapsulates the current state of the input and output as the former is parsed and the latter is accumulated. It is manipulated by each grammar class as the "interpreting" process transforms the input into the output.

Examples:

+ Windows Forms designer file (XML),

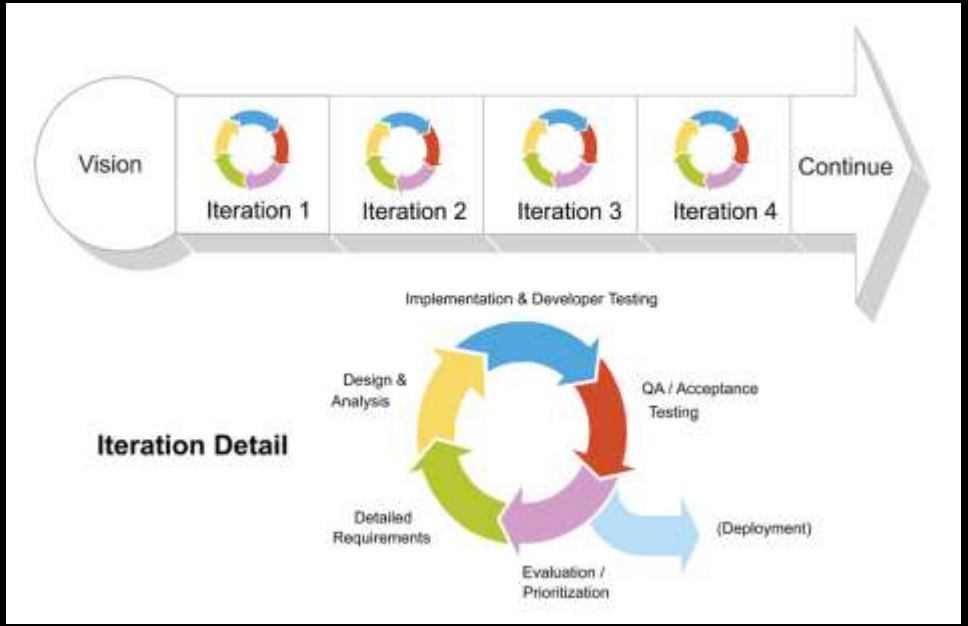
+ WPF (XAML)

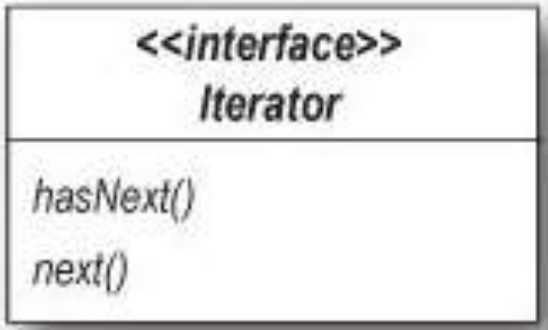
+ WCF configuration (XML)

+ Config files

Iterator

Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

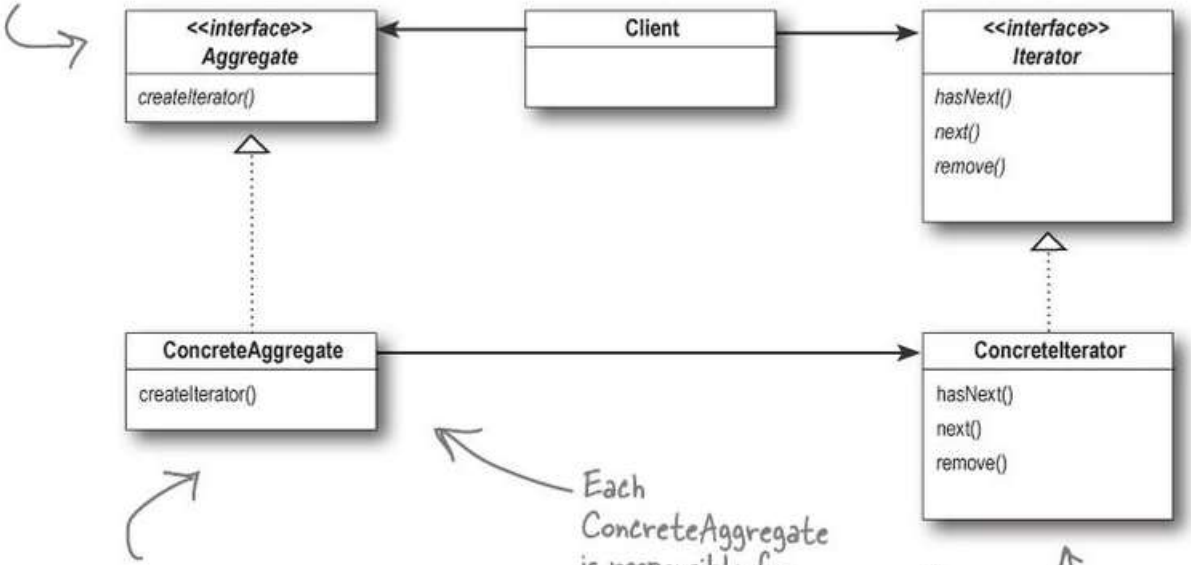




The hasNext() method tells us if there are more elements in the aggregate to iterate through.

The next() method returns the next object in the aggregate.

Having a common interface for your aggregates is handy for your client; it decouples your client from the implementation of your collection of objects.



The Iterator interface provides the interface that all iterators must implement, and a set of methods for traversing over elements of a collection. Here we're using the java.util.Iterator. If you don't want to use Java's Iterator interface, you can always create your own.

The ConcreteAggregate has a collection of objects and implements the method that returns an Iterator for its collection.

Each ConcreteAggregate is responsible for instantiating a ConcreteIterator that can iterate over its collection of objects.

The ConcreteIterator is responsible for managing the current position of the iteration.

C#’s *foreach* **relies on** *IEnumerable*:

+ *IEnumerable: IEnumerator GetEnumerator()*

+ IEnumerator: Reset(), Current, bool MoveNext()

Single Responsibility Principle

Keep each class to a single responsibility.

Every responsibility of a class is an area of potential change. More than one responsibility means more than one area of change.

Use a separate iterator class if the collection class should not have this responsibility (*polymorphic traversal*).
Then we can use different traversals on the same list.

High cohesion = all methods are related

We strife for high cohesion within a class and
for low cohesion between classes.

Example (C++ STL):

- 4 data structures (array, tree, linked list, hash table)

- 3 operations: sort, find, merge

Operations within collection: 4 x 3 = 12 methods

Separately implemented: 4 iterators + 3 methods

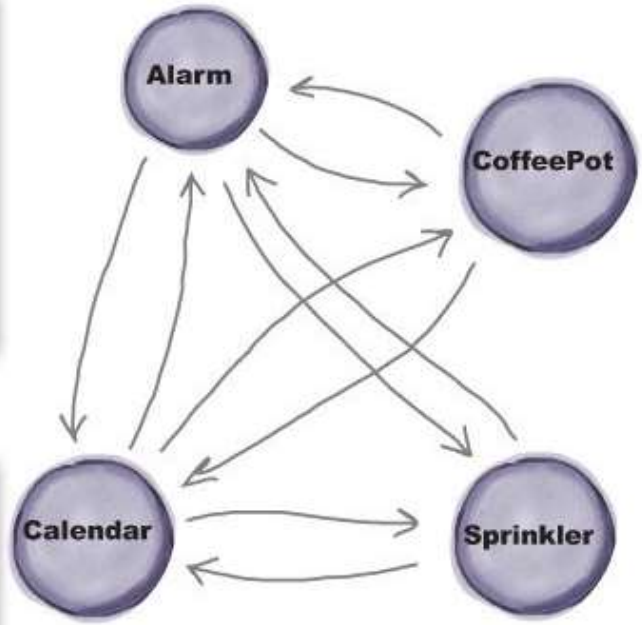
Mediator

Define an object that encapsulates how a set of objects interact.

Mediator promotes loose coupling by preventing objects from referring to each other explicitly, and it lets you vary their interaction independently.

Alarm

```
onEvent() {
  checkCalendar()
  checkSprinkler()
  startCoffee()
  // do more stuff
}
```



CoffeePot

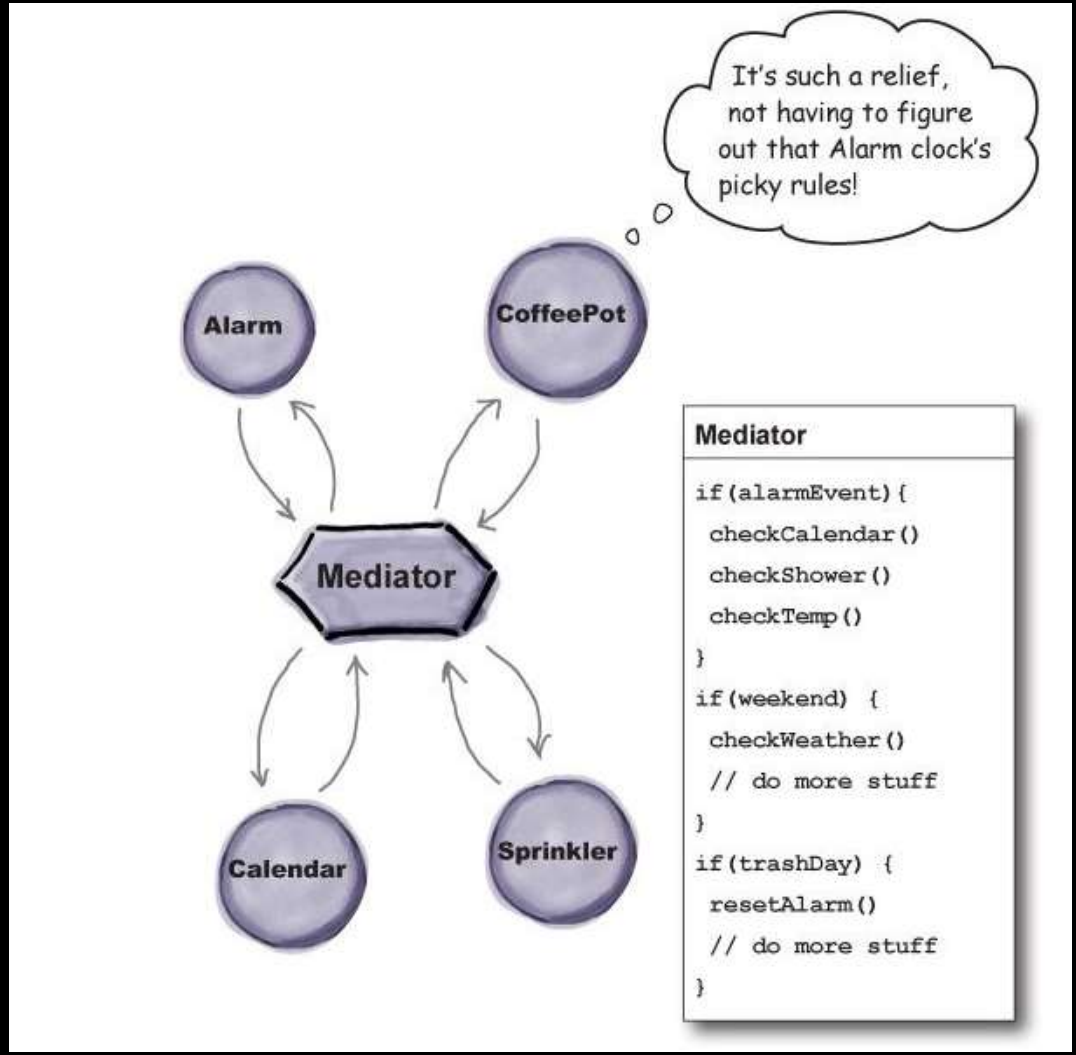
```
onEvent() {
  checkCalendar()
  checkAlarm()
  // do more stuff
}
```

Calendar

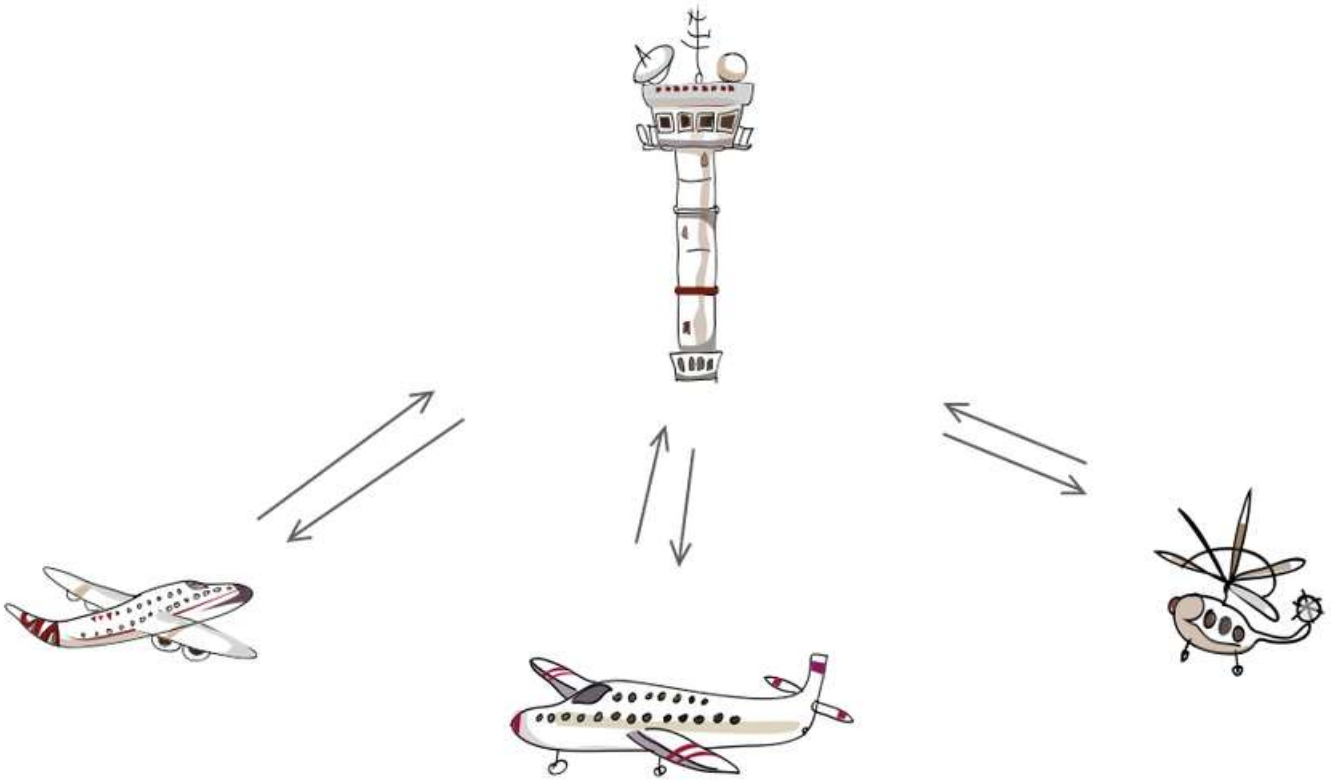
```
onEvent() {
  checkDayOfWeek()
  doSprinkler()
  doCoffee()
  doAlarm()
  // do more stuff
}
```

Sprinkler

```
onEvent() {
  checkCalendar()
  checkShower()
  checkTemp()
  checkWeather()
  // do more stuff
}
```


It's such a relief, not having to figure out that Alarm clock's picky rules!
Alarm
CoffeePot
Mediator
Calendar
Sprinkler
Mediator
if(alarmEvent){
checkCalendar()
checkShower()
checkTemp()
}
if(weekend) {
checkWeather()
// do more stuff
}
if(trashDay) {
resetAlarm()
// do more stuff
}

ATC Mediator



Mediator centralizes complex communication and control between objects.

With a mediator, objects don’t have to know of each other as they only deal with the mediator.

The objects tell the mediator when their state changes and they respond to requests from the mediator.

Simplifies maintenance of the system by centralizing logic.

Reduces variety of messages sent between objects.

But be aware of creating god classes!

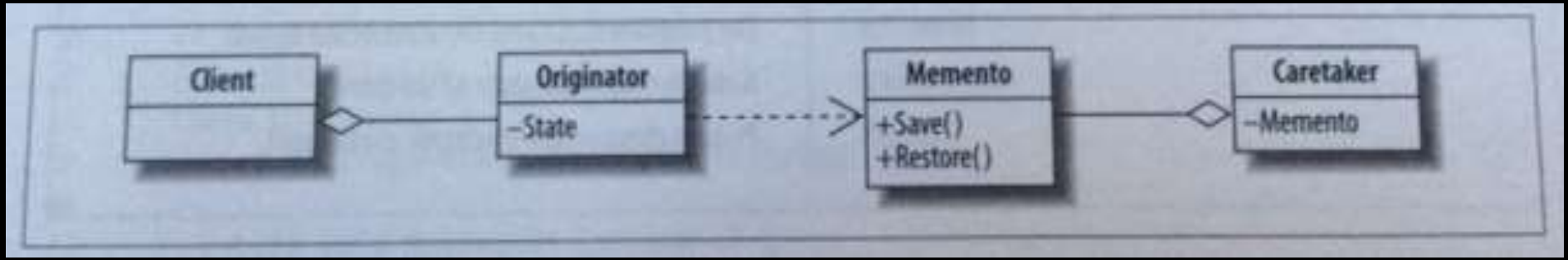
Examples:

+ coordinate related GUI components

+ managing users and groups in UNIX with a mapping

Memento

Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.


Client
Originator
-State
Memento
+Save()
+Restore()
Caretaker
-Memento

+ Memento saves the state of a system’s key object

+ Memento maintains the key object’s encapsulation

+ Provides an easy mechanism to restore objects (Undo)

SRP:

Keeping the saved object’s state separate from object.

This separated state is called “Memento”.

```
[Serializable]
// Serializes by deep copy to memory and back
public class Memento
{
    readonly MemoryStream _stream = new MemoryStream();
    readonly BinaryFormatter _formatter = new BinaryFormatter();

    public Memento Save(Object o)
    {
        _formatter.Serialize(_stream, o);
        return this;
    }

    public object Restore()
    {
        _stream.Seek(0, SeekOrigin.Begin);
        var o = _formatter.Deserialize(_stream);
        _stream.Close();
        return o;
    }
}
```

The client is the "care-taker" of the Memento, but only the source object can store and retrieve information from the Memento (the Memento is "opaque" to the client and all other objects).

If the client subsequently needs to "rollback" the source object's state, it hands the Memento back to the source object for reinstatement.

Examples:

+ save the state in a computer game

+ save the state of long-running computations

+ save the state to allow undo in image processing

Null Object

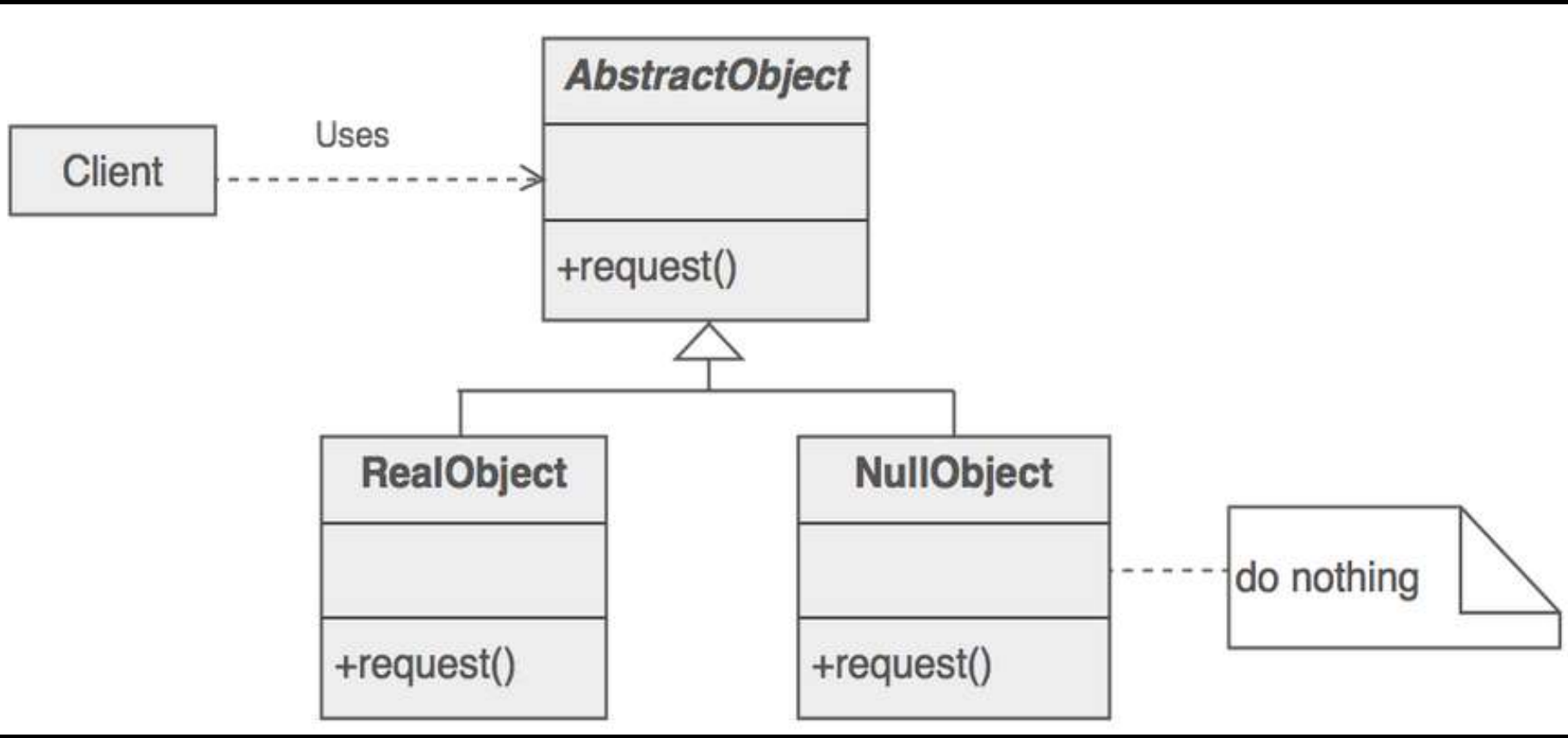
Encapsulate the absence of an object by providing a substitutable alternative that offers suitable default do nothing behaviour.

Rather than using null references.

A null object is useful when you don’t have a meaningful object to return, and yet you want to remove the responsibility from the client to handle null references.

The result of a null check is to do nothing (neutral behaviour) or use some default value.

Example: returning an empty list rather than null


AbstractObject
Client
Uses
+request()
RealObject
+request()
NullObject
+request()
do nothing

```
// Animal interface is the key to compatibility for Animal implementations below.
interface IAnimal
{
    void MakeSound();
}

// Dog is a real animal.
class Dog : IAnimal
{
    public void MakeSound()
    {
        Console.WriteLine("Woof!");
    }
}

// The Null Case: this NullAnimal class should be instantiated and used in place of C# null keyword.
class NullAnimal : IAnimal
{
    public void MakeSound()
    {
        // Purposefully provides no behaviour.
    }
}
```

+ often implemented as a *Singleton*

+ can be a special case of a *Strategy* or a *State* pattern

+ allows a *Visitor* to safely visit a hierarchy

+ Objective-C does nothing when sending a message to *nil*

Extension methods can be used to perform 'null coalescing'. This is because extension methods can be called on null values as if it concerns an 'instance method invocation' while in fact extension methods are static.

Extension methods can be made to check for null values, thereby freeing code that uses them from having to do so.

```
using System;
using System.Linq;
namespace MyExtensionWithExample {
    public static class StringExtensions {
        public static int SafeGetLength(this string valueOrNull) {
            return (valueOrNull ?? string.Empty).Length;
        }
    }
    public static class Program {
        // define some strings
        static readonly string[] strings = new [] { "Mr X.", "Katrien Duck", null, "Q" };
        // write the total length of all the strings in the array
        public static void Main(string[] args) {
            var query = from text in strings select text.SafeGetLength(); // no need to do any checks here
            Console.WriteLine(query.Sum());
        }
    }
}
// The output will be:
// 18
```

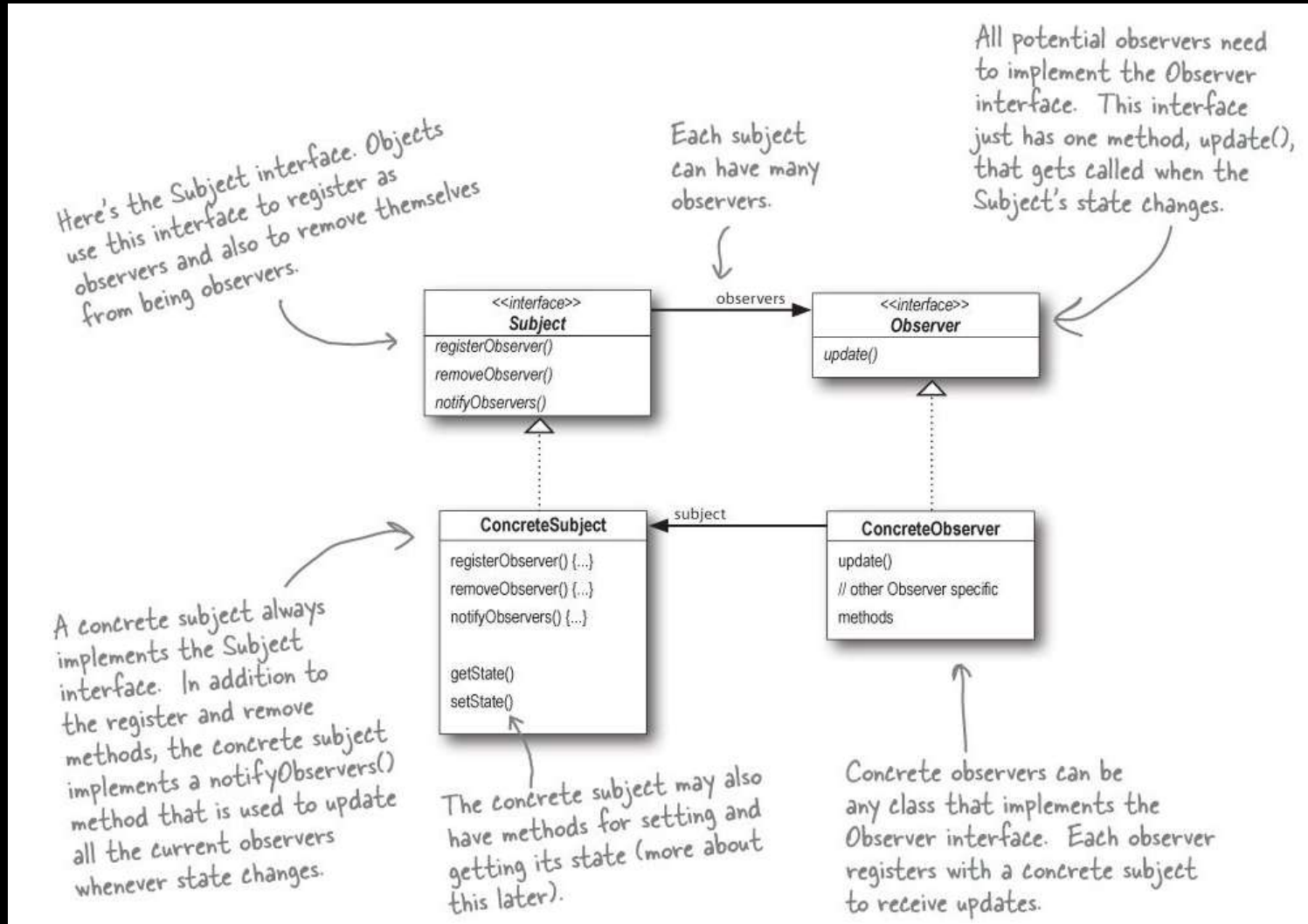
Observer

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

Minimize interdependency.

Observers register themselves at runtime with the Subject as they are created.

Whenever the **Subject changes, it broadcasts** to all registered Observers that it has changed, and each Observer queries the Subject for that subset of the Subject's state that it is responsible for monitoring.


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

Loosely Coupled System: each of its components has little to no knowledge of other components.

Subject and Observers are loosely coupled: both use only a tiny interface of the other and can be reused independently.

We never need to change the Subject when we add new types of Observers. Don’t rely on any notification order!

Examples:

+ the View of a Model-View-Controller

+ WinForms event handling:

```
EventHandler handler = (s, e) => MessageBox.Show("Woho");
button.Click += handler; button.Click -= handler;
```

+ WPF: INotifyPropertyChanged + data binding

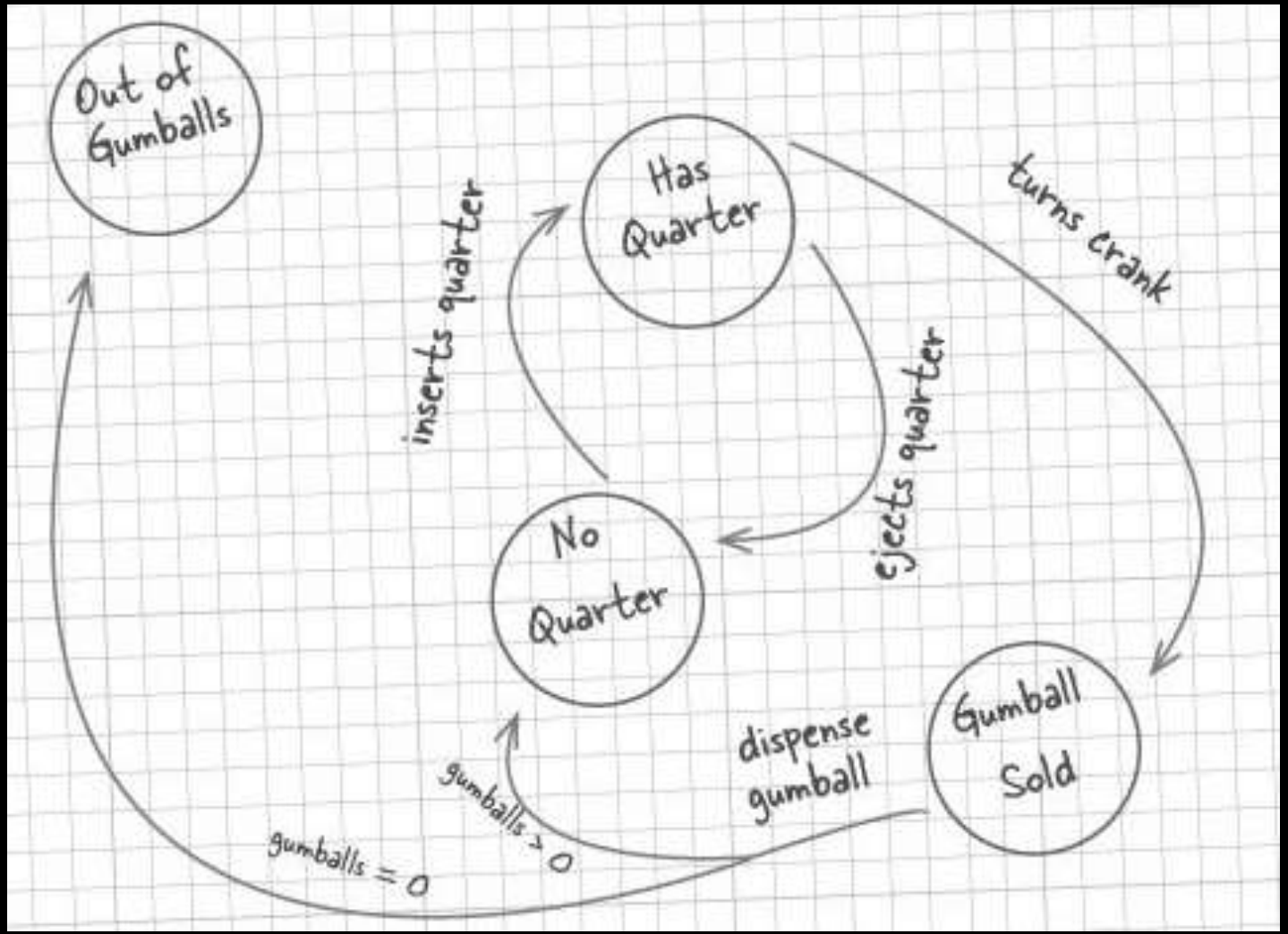
```
public class Book : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    protected void OnPropertyChanged(string name)
    {
        if (PropertyChanged != null)
        {
            PropertyChanged(this, new PropertyChangedEventArgs(name));
        }
    }

    private string bookName;
    public string BookName
    {
        get { return bookName; }
        set
        {
            if (bookName != value)
            {
                bookName = value;
                OnPropertyChanged("BookName");
            }
        }
    }
}
```

State

Allow an object to alter its behaviour when its internal state changes.

The object will appear to change its class (object oriented state machine).

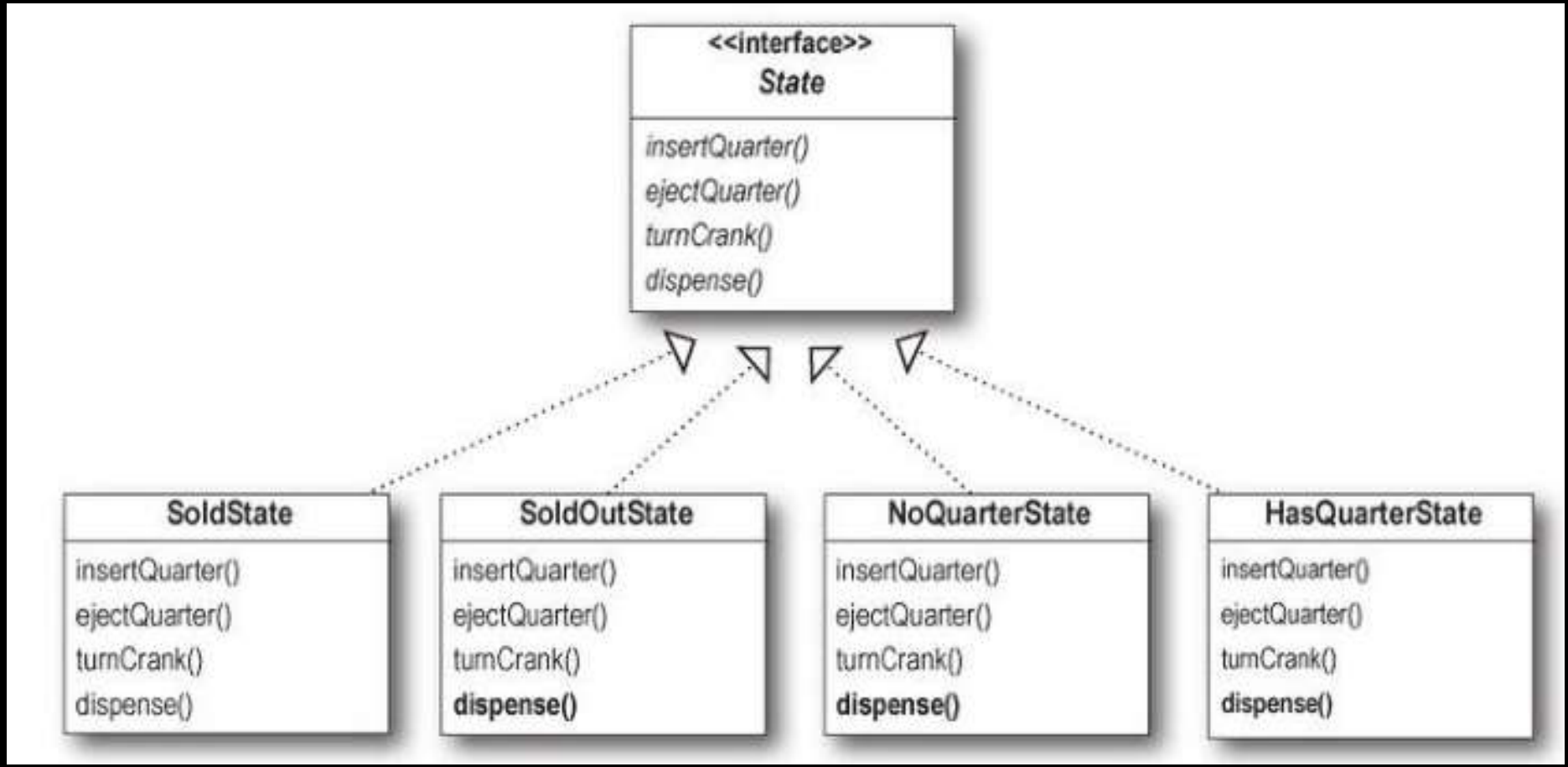
Behaviour depending on state.


Out of Gumballs
Has Quarter
inserts quarter
turns crank
ejects quarter
No Quarter
Gumball Sold
dispense gumball
gumballs > 0
gumballs = 0

State Pattern allows an object to have many different behaviours based on its internal state.

Use one class per state to localize its behaviour.
If we have to change one class, it does not affect the others.

The context class delegates to the state object that represents the current state instead of using *if* statements.


<<interface>>
State
insertQuarter()
ejectQuarter()
turnCrank()
dispense()
SoldState
insertQuarter()
ejectQuarter()
turnCrank()
dispense()
SoldOutState
insertQuarter()
ejectQuarter()
turnCrank()
dispense()
NoQuarterState
insertQuarter()
ejectQuarter()
turnCrank()
dispense()
HasQuarterState
insertQuarter()
ejectQuarter()
turnCrank()
dispense()

First we need to implement the State interface.

We get passed a reference to the Gumball Machine through the constructor. We're just going to stash this in an instance variable.

```
public class NoQuarterState implements State {
    GumballMachine gumballMachine;

    public NoQuarterState(GumballMachine gumballMachine) {
        this.gumballMachine = gumballMachine;
    }

    public void insertQuarter() {
        System.out.println("You inserted a quarter");
        gumballMachine.setState(gumballMachine.getHasQuarterState());
    }

    public void ejectQuarter() {
        System.out.println("You haven't inserted a quarter");
    }

    public void turnCrank() {
        System.out.println("You turned, but there's no quarter");
     }

    public void dispense() {
        System.out.println("You need to pay first");
    }
}
```

If someone inserts a quarter, we print a message saying the quarter was accepted and then change the machine's state to the HasQuarterState.

You'll see how these work in just a sec...

You can't get money back if you never gave it to us!

And you can't get a gumball if you don't pay us.

We can't be dispensing gumballs without payment.

```
public class GumballMachine {

    State soldOutState;
    State noQuarterState;
    State hasQuarterState;
    State soldState;

    State state;
    int count = 0;

    public GumballMachine(int numberGumballs) {
        soldOutState = new SoldOutState(this);
        noQuarterState = new NoQuarterState(this);
        hasQuarterState = new HasQuarterState(this);
        soldState = new SoldState(this);

        this.count = numberGumballs;
        if (numberGumballs > 0) {
            state = noQuarterState;
        } else {
            state = soldOutState;
        }
    }

    public void insertQuarter() {
        state.insertQuarter();
    }
    public void ejectQuarter() {
        state.ejectQuarter();
    }
    public void turnCrank() {
        state.turnCrank();
        state.dispense();
    }
```

Here are all the States again...

...and the State instance variable.

The count instance variable holds the count of gumballs – initially the machine is empty.

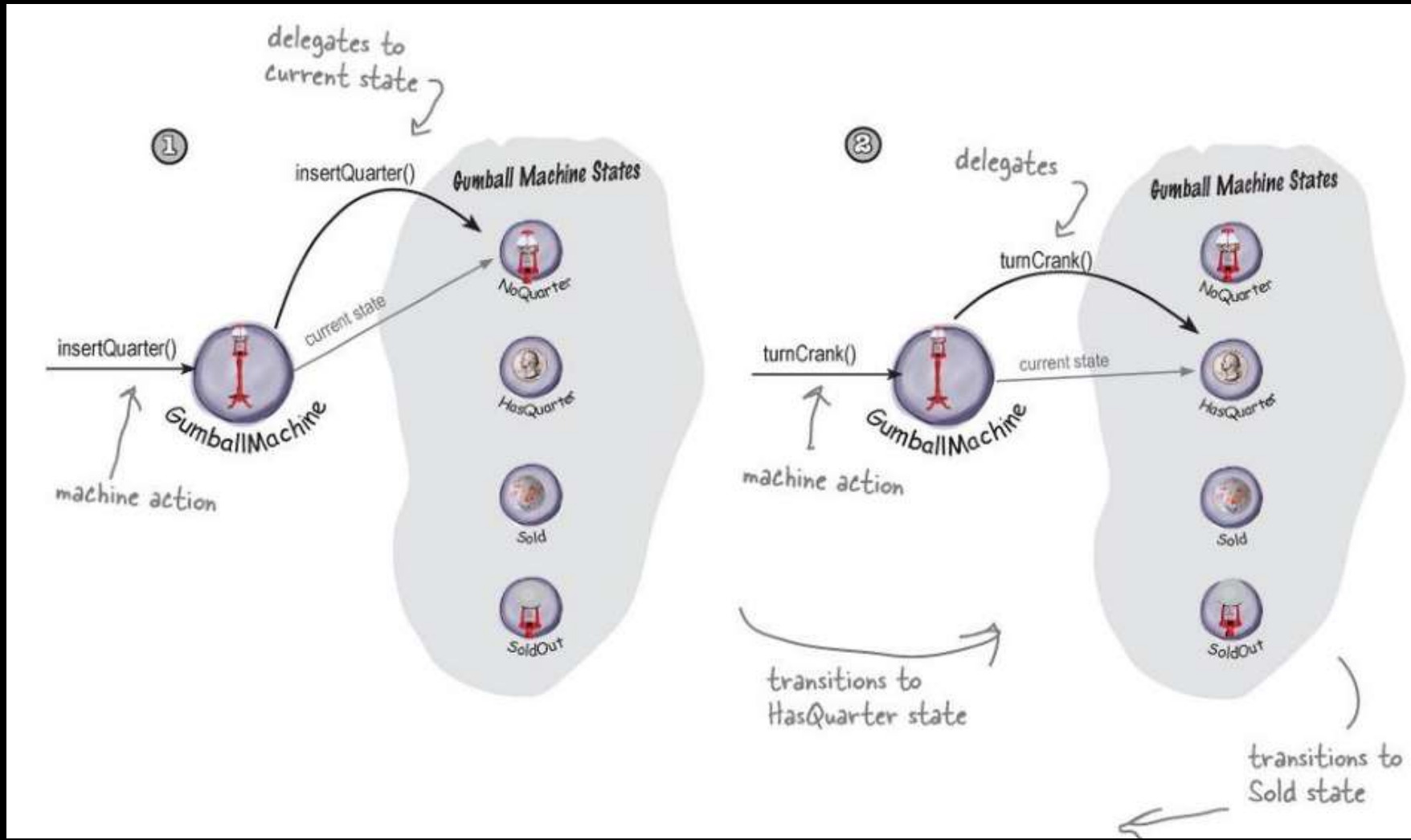
Our constructor takes the initial number of gumballs and stores it in an instance variable.

It also creates the State instances, one of each.

If there are more than 0 gumballs we set the state to the NoQuarterState; otherwise, we start in the SoldOutState.

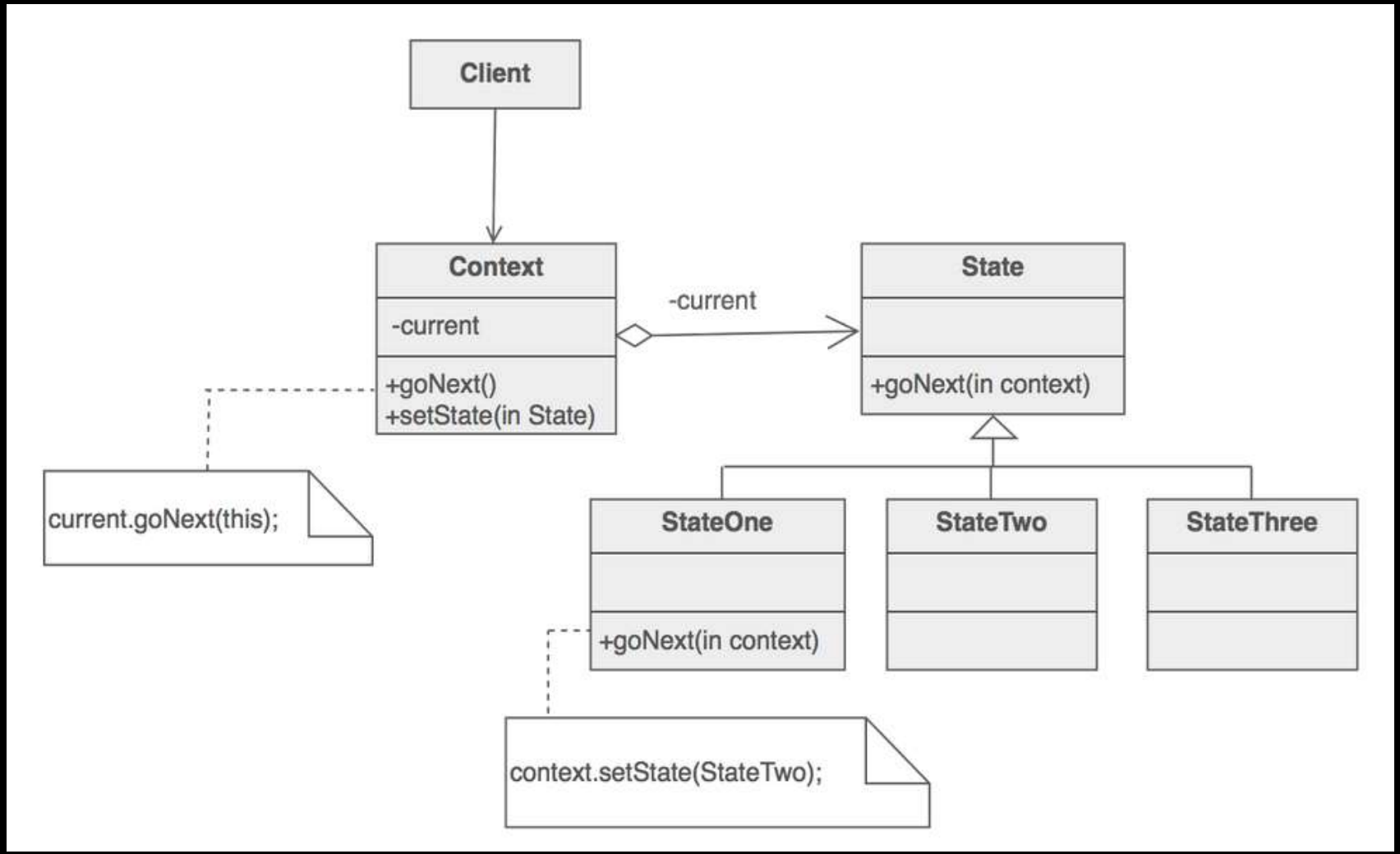
Now for the actions. These are VERY EASY to implement now. We just delegate to the current state.

Note that we don't need an action method for dispense() in GumballMachine because it's just an internal action; a user can't ask the machine to dispense directly. But we do call dispense() on the State object from the turnCrank() method.


delegates to current state
1
insertQuarter()
Gumball Machine States
insertQuarter()
current state
GumballMachine
machine action
NoQuarter
HasQuarter
Sold
SoldOut
2
delegates
turnCrank()
Gumball Machine States
turnCrank()
current state
GumballMachine
machine action
NoQuarter
HasQuarter
Sold
SoldOut
transitions to HasQuarter state
transitions to Sold state

The State pattern does not specify where the state transitions will be defined. The choices are two: the "context" object, or each individual State derived class.

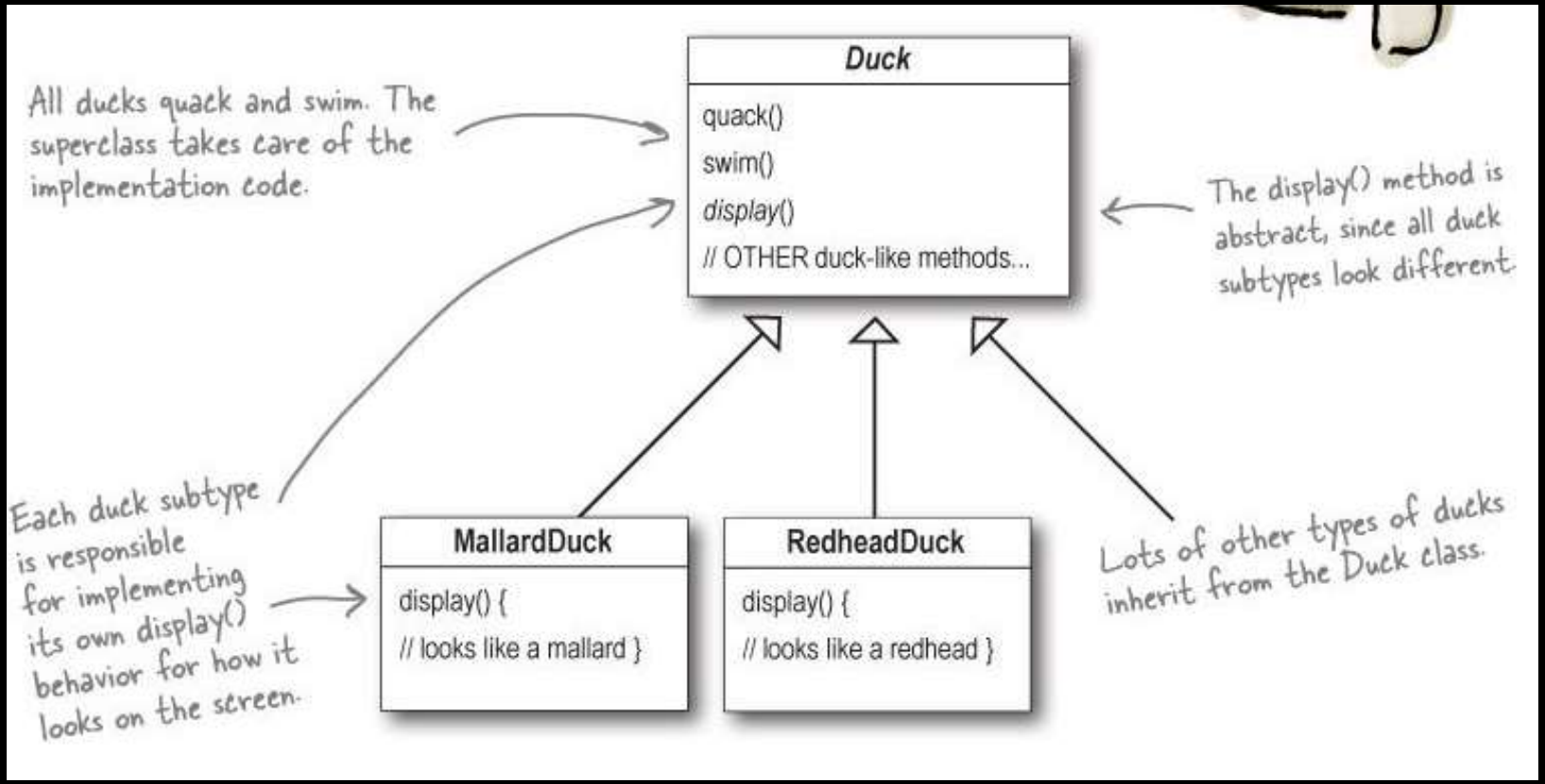
The advantage of the latter option is ease of adding new State derived classes. The disadvantage is each State derived class has knowledge of (coupling to) its siblings, which introduces dependencies between subclasses.


Client
Context
-current
+goNext()
+setState(in State)
-current
State
+goNext(in context)
current.goNext(this);
StateOne
+goNext(in context)
StateTwo
StateThree
context.setState(StateTwo);

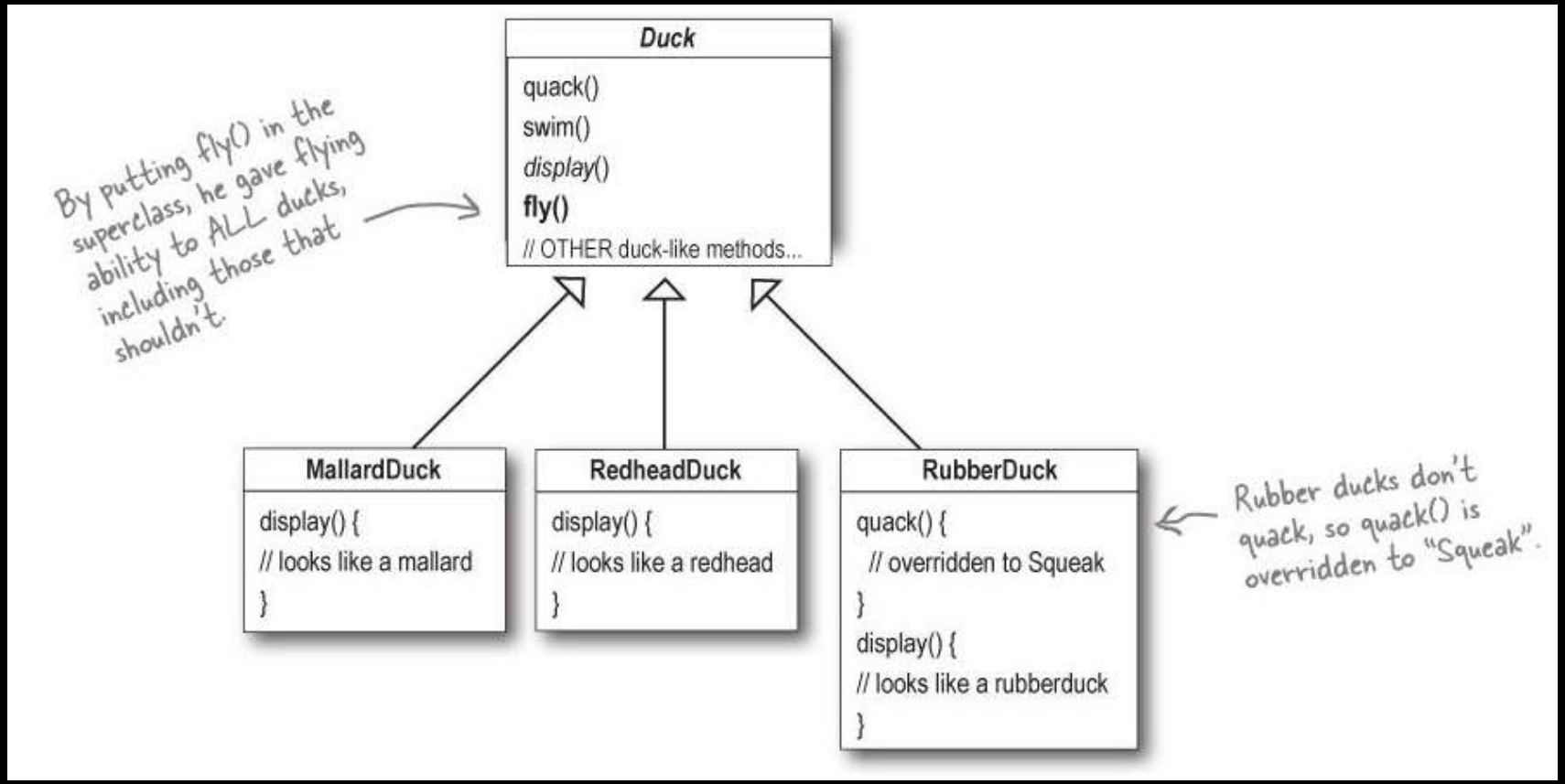
Strategy

Define a family of algorithms, encapsulate each one, and make them interchangeable.

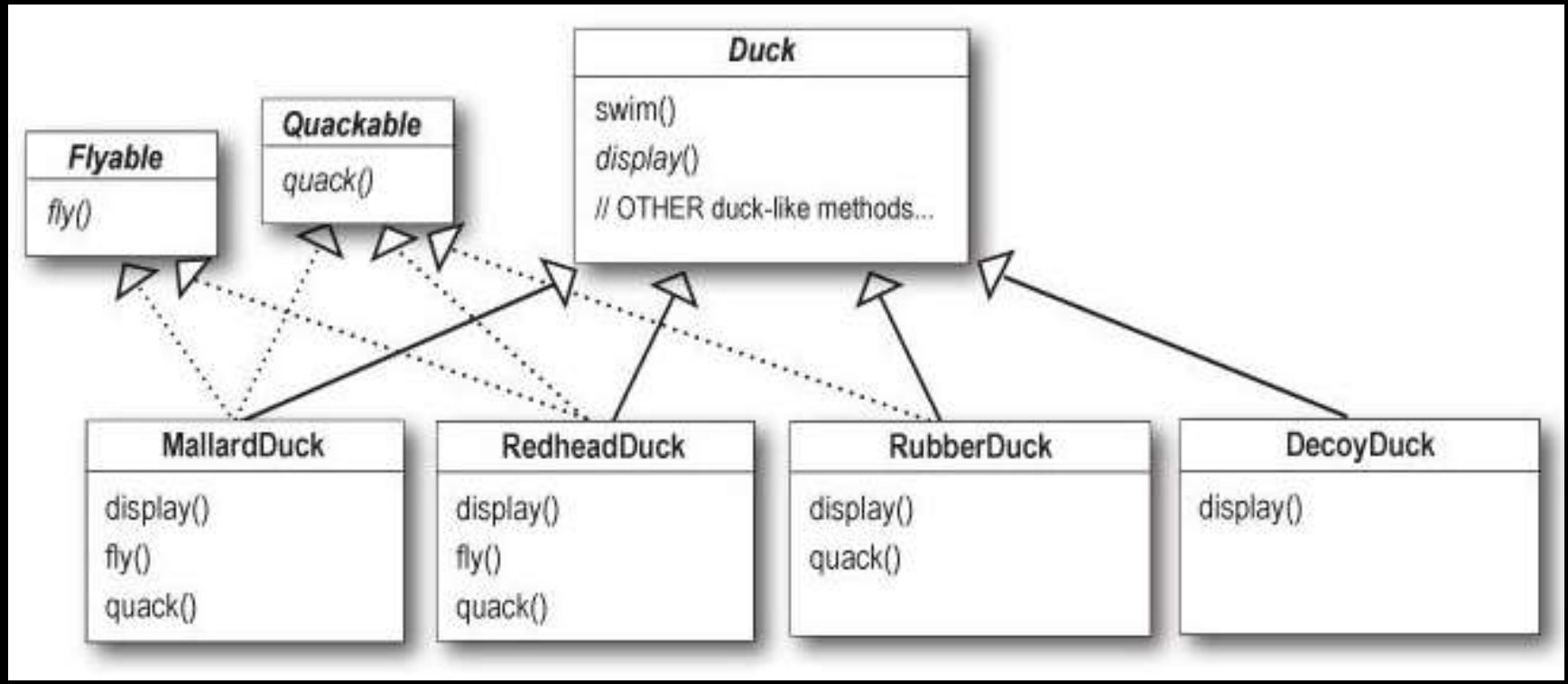
Strategy lets the algorithm vary independently from clients that use it.



Start with base class implementing common behaviour, override specific behaviour in derived classes.



Problem adding *Fly()* to base class: *RubberDuck* cannot fly!

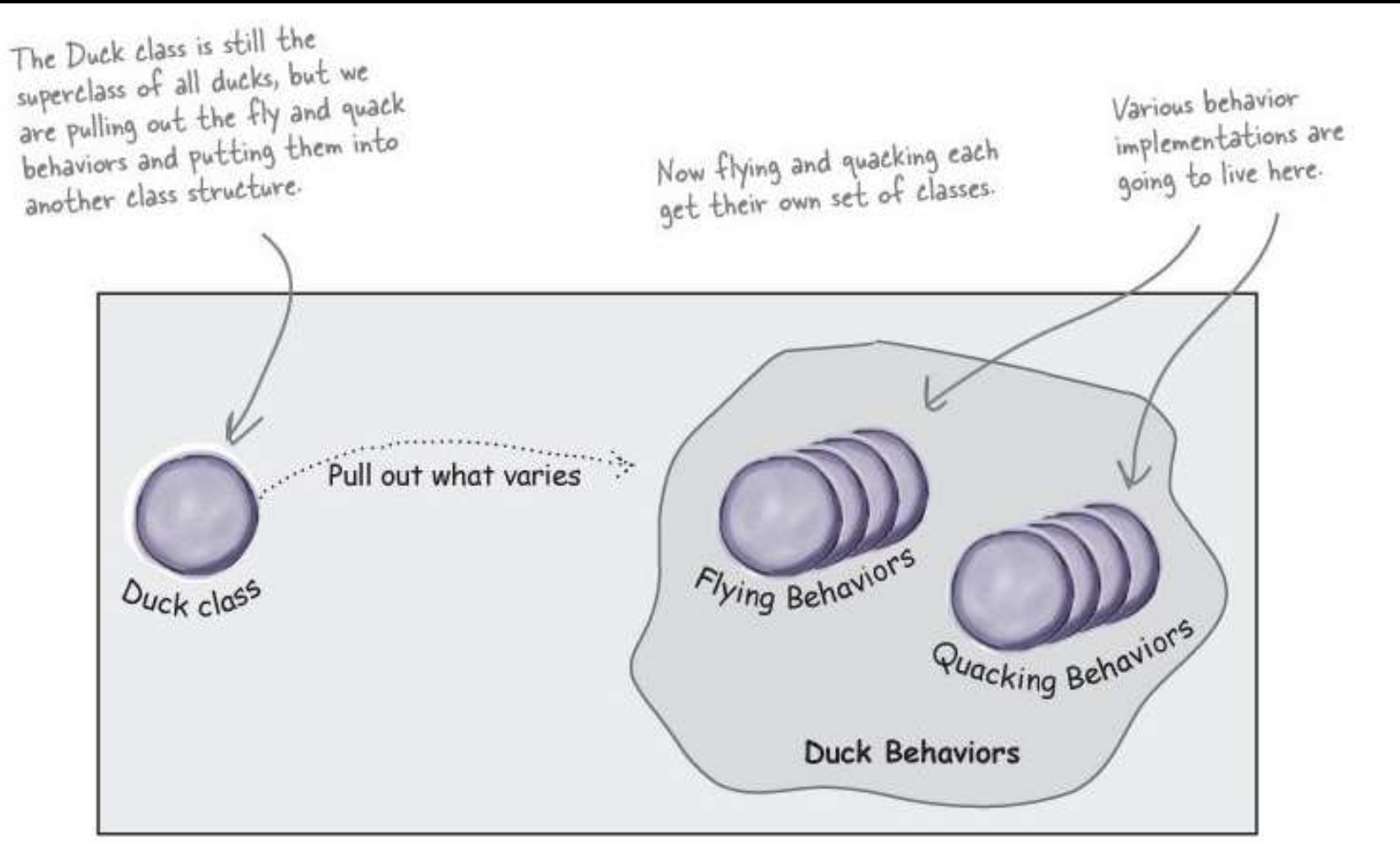


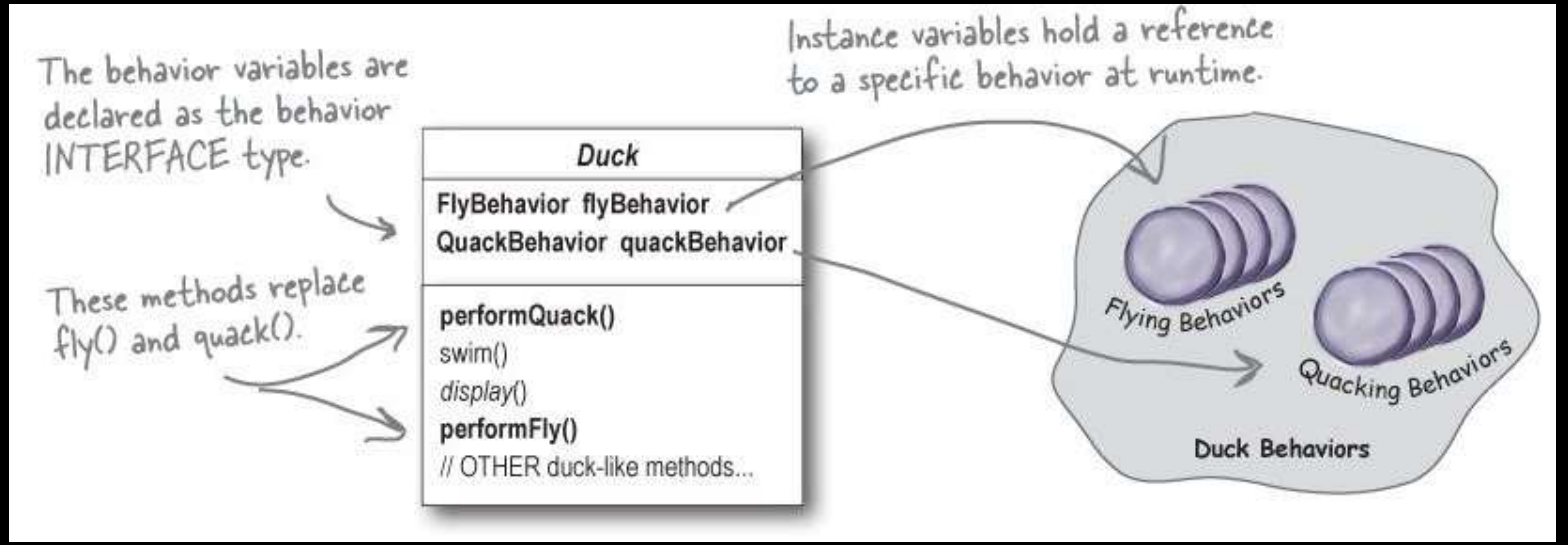
Problem with using Interfaces: duplicate code!`

Extract changing behaviour: Identify the aspects that vary and separate them from what stays the same.

In other words: Take the parts that vary and encapsulate them in an interface, so that later you can alter or extend the parts that vary without affecting those that don’t!

Result: fewer unintended consequences from code changes and more flexibility in your system!

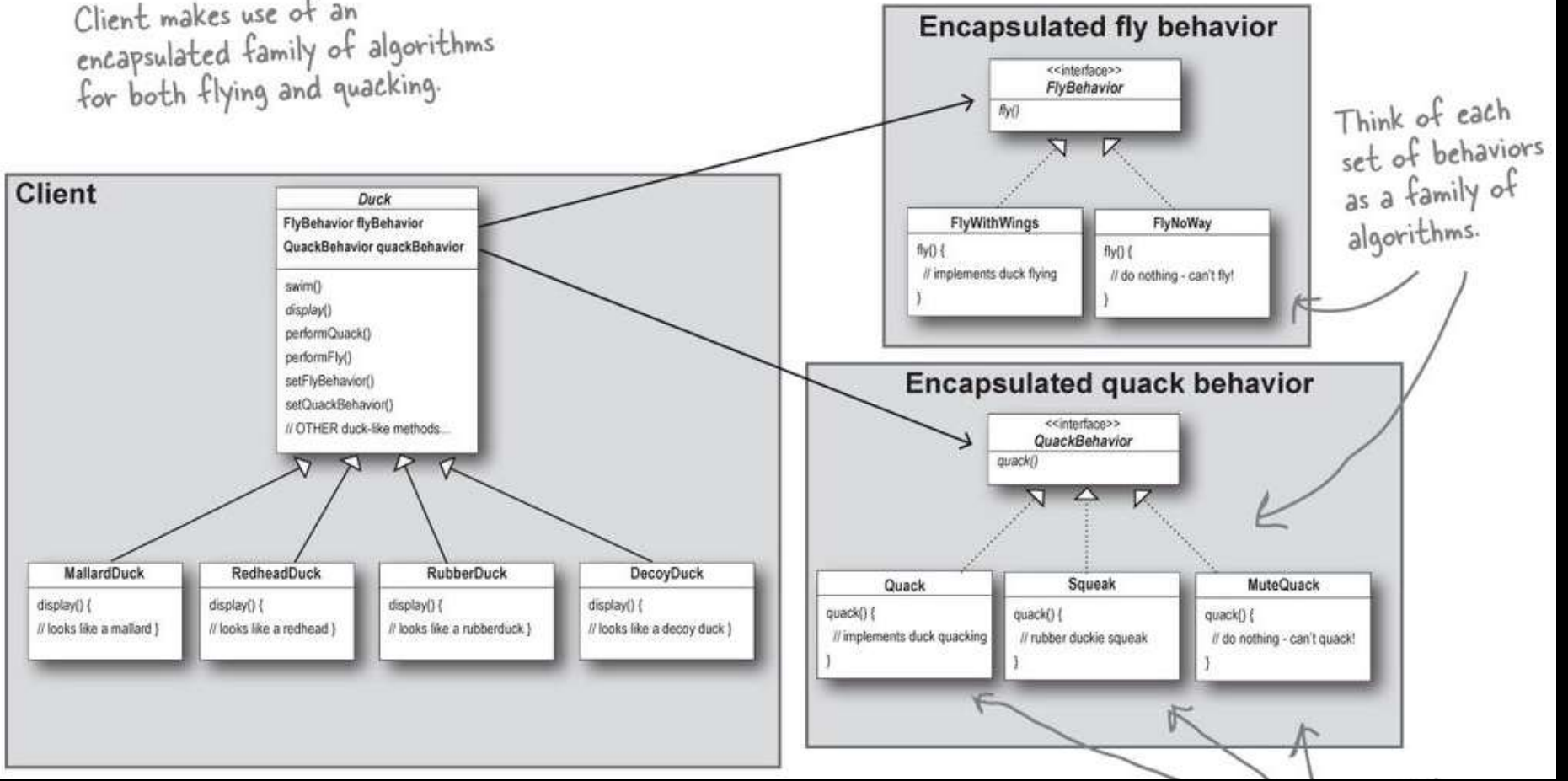

The Duck class is still the superclass of all ducks, but we are pulling out the fly and quack behaviors and putting them into another class structure.
Now flying and quacking each get their own set of classes.
Various behavior implementations are going to live here.
Pull out what varies
Duck class
Flying Behaviors
Quacking Behaviors
Duck Behaviors



Duck **delegates** **behaviour:**

```
public class Duck {
    QuackBehavior quackBehavior;
    // more

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

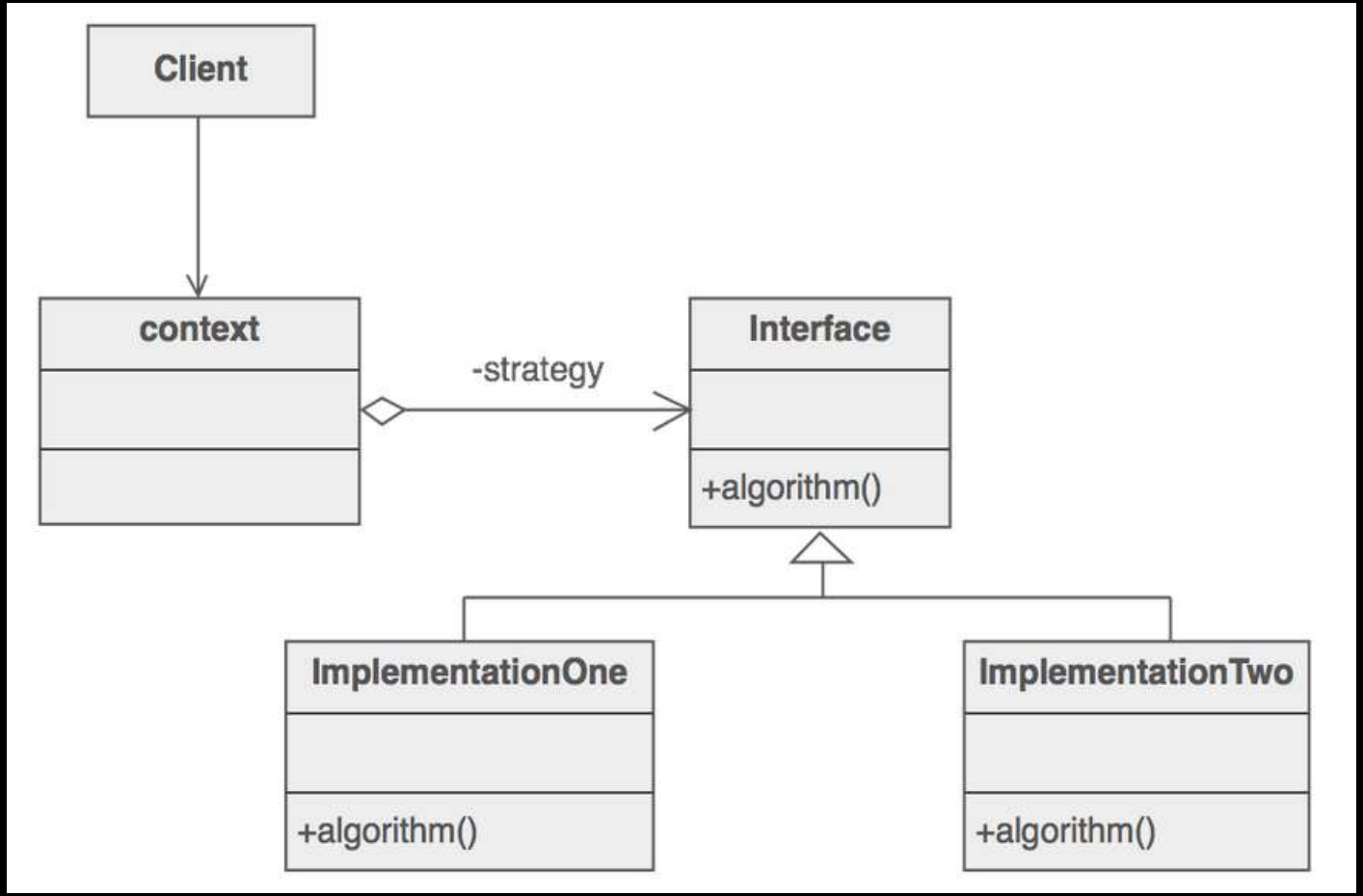
Client makes use of an encapsulated family of algorithms for both flying and quacking.

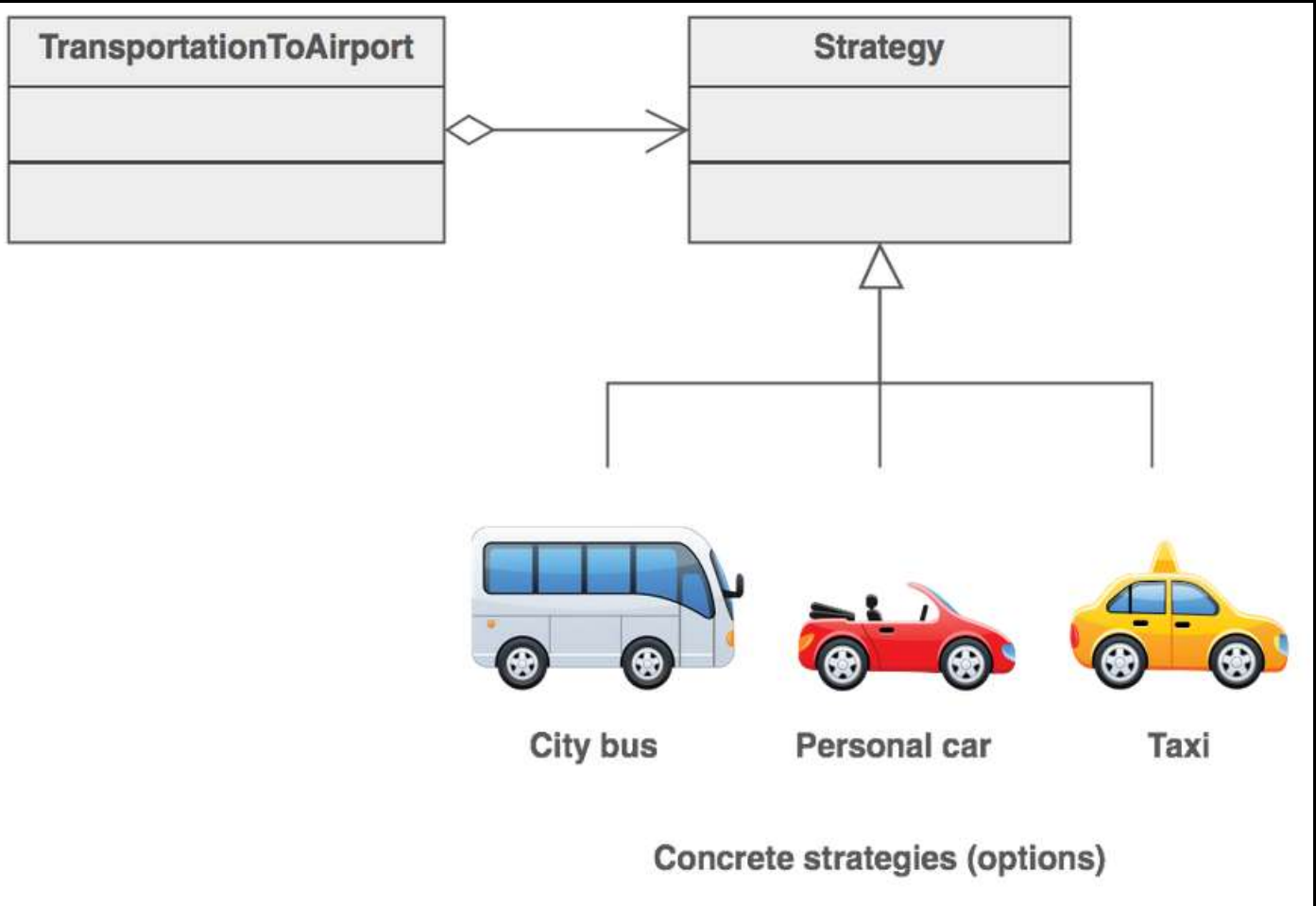


Each *duck* **HAS-A** *FlyBehaviour* **and a** *QuackBehaviour***.**

Using composition & delegation instead of inheritance.

This allows us to encapsulate a family of algorithms into their own set of classes and to change behaviour at runtime.

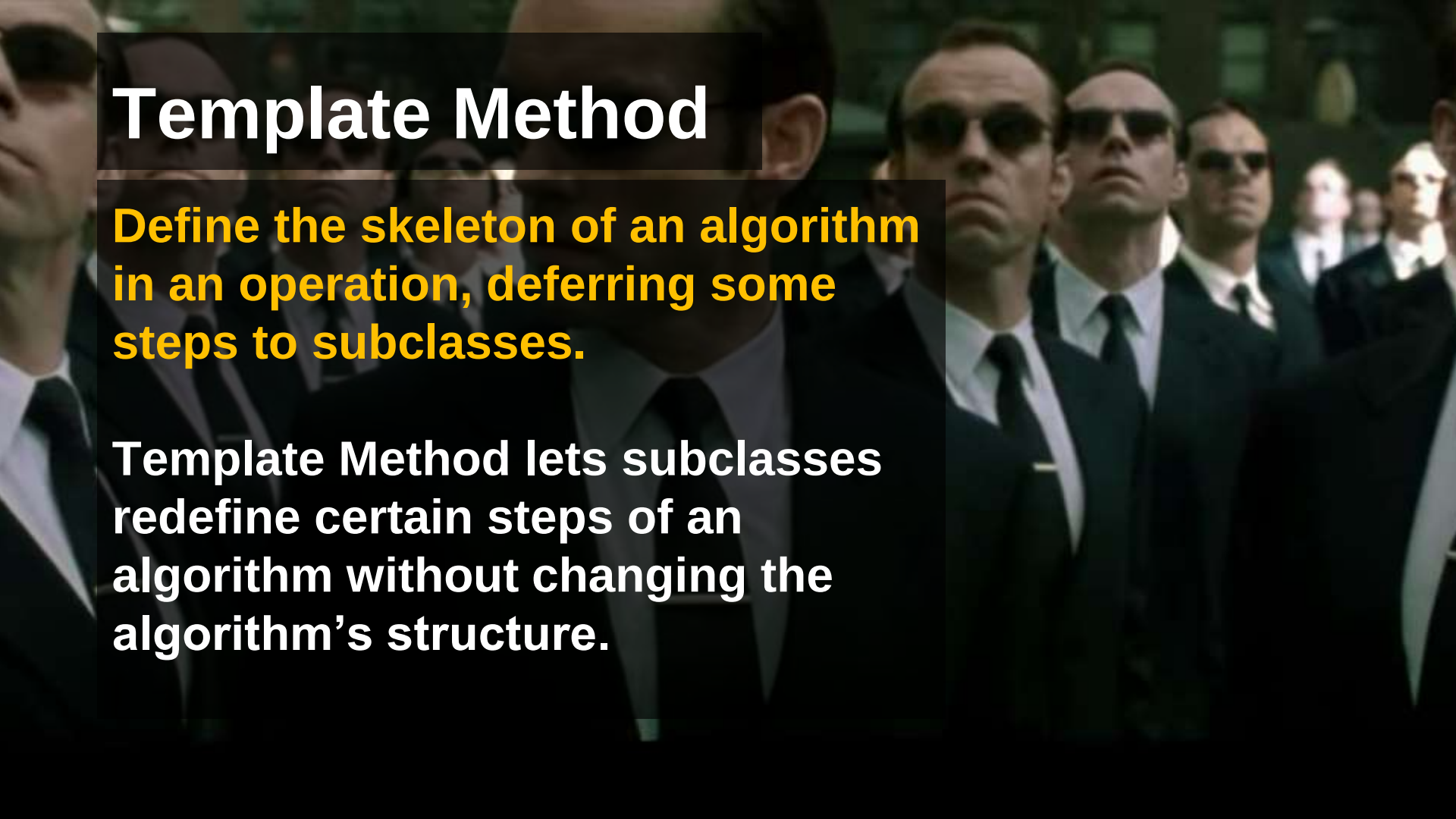

Client
context
-strategy
Interface
+algorithm()
ImplementationOne
+algorithm()
ImplementationTwo
+algorithm()


TransportationToAirport
Strategy
City bus
Personal car
Taxi
Concrete strategies (options)

State and Strategy Pattern have the same class diagram, but they differ in intent.

With Strategy Pattern, the client configures the Context classes with a behaviour or algorithm (*bind-once*).

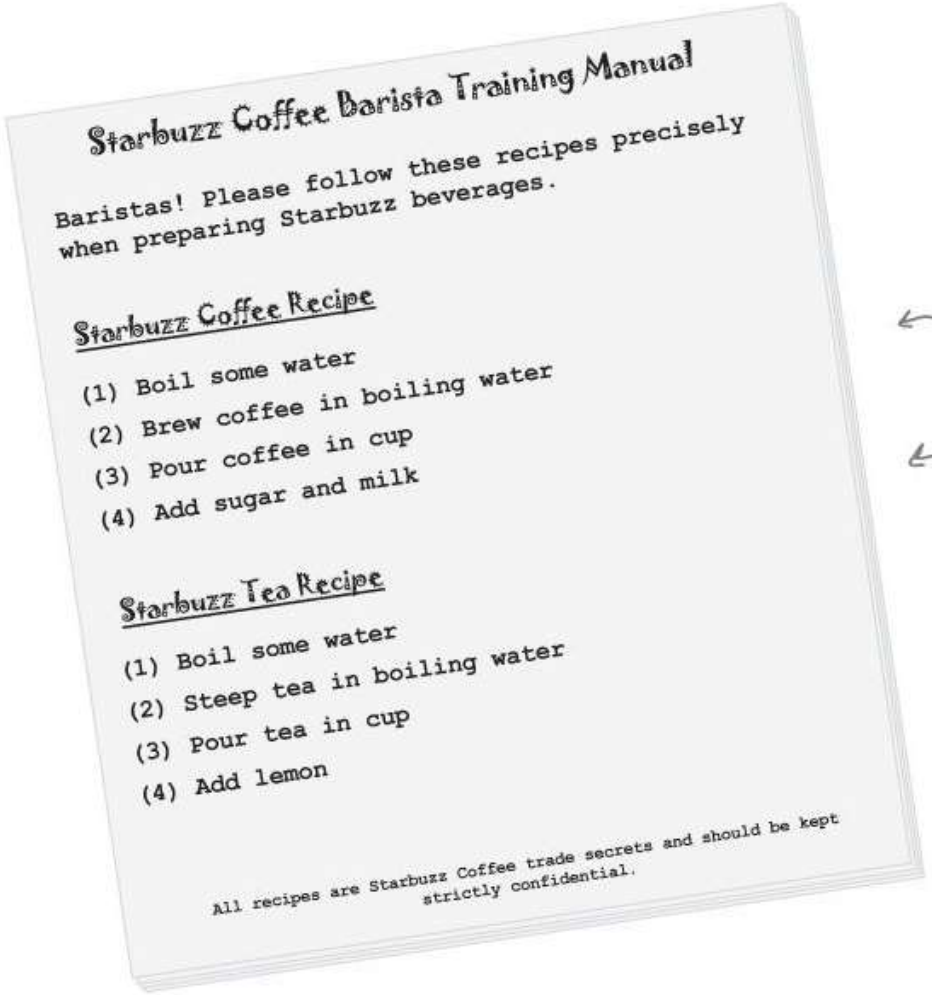
State Pattern allows a Context to change its behaviour as the state of the Context changes (*bind-dynamic*). The client usually knows very little, if anything, about the state objects.



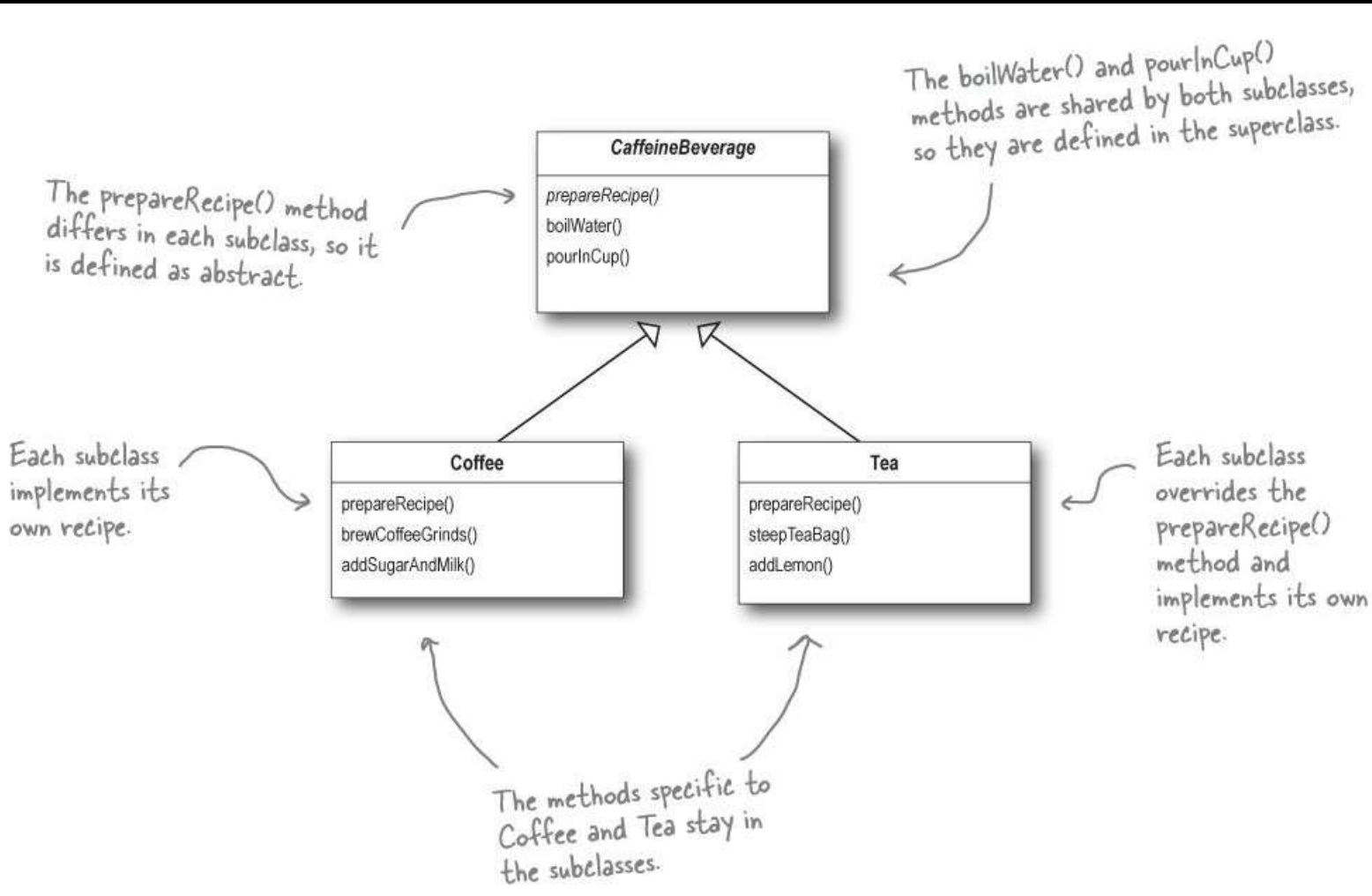
Template Method

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses.

Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.



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


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

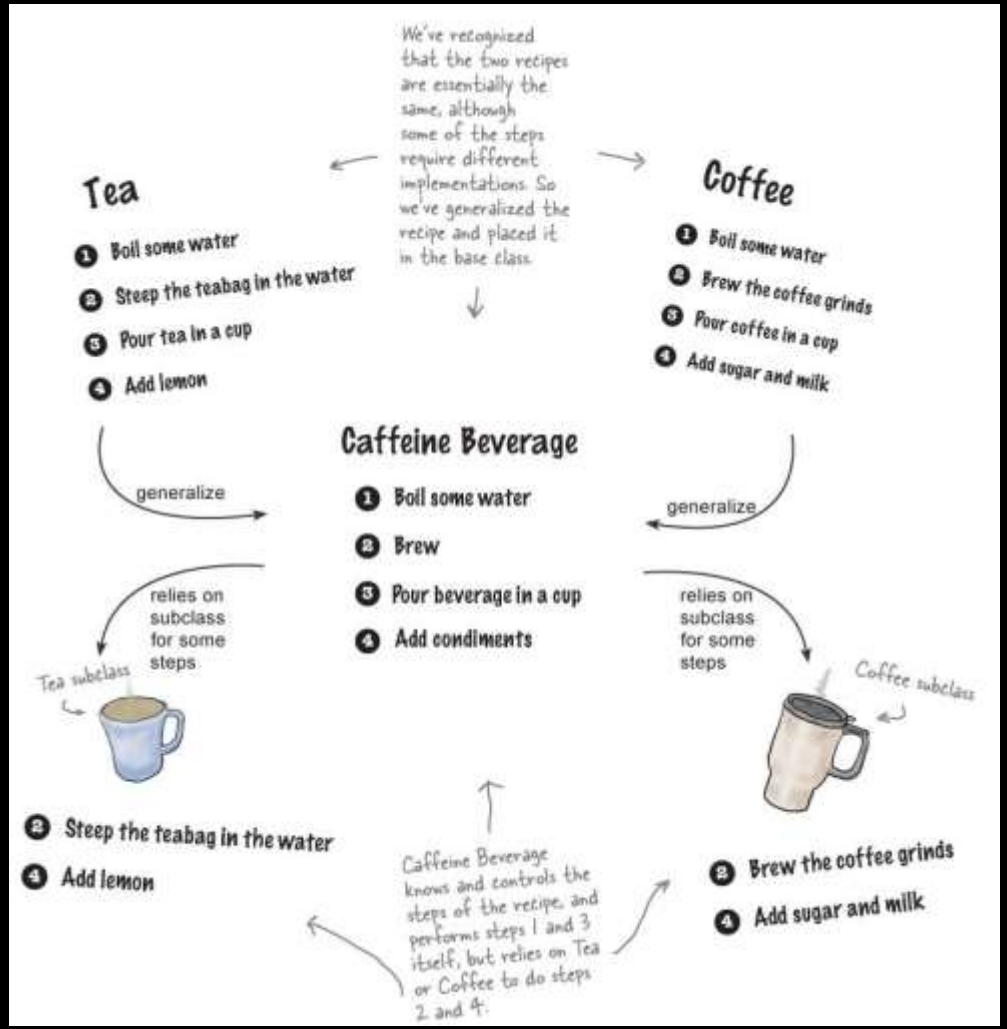
CaffeineBeverage is abstract, just like in the class design.

```
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!

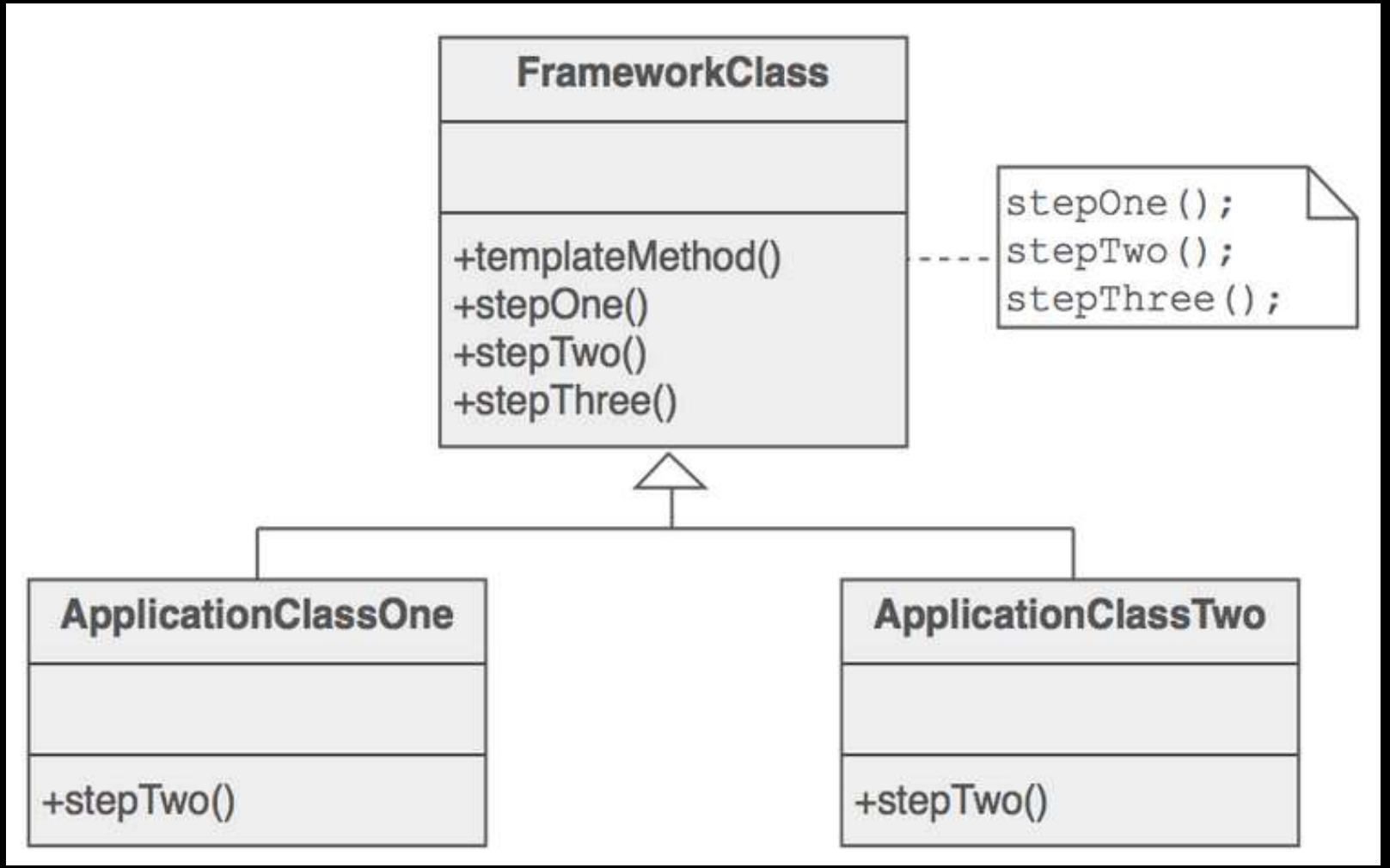
Remember, we moved these into the CaffeineBeverage class (back in our class diagram).


We've recognized that the two recipes are essentially the same, although some of the steps require different implementations. So we've generalized the recipe and placed it in the base class.
Tea
1 Boil some water
2 Steep the teabag in the water
3 Pour tea in a cup
4 Add lemon
Coffee
1 Boil some water
2 Brew the coffee grinds
3 Pour coffee in a cup
4 Add sugar and milk
generalize
generalize
Caffeine Beverage
1 Boil some water
2 Brew
3 Pour beverage in a cup
4 Add condiments
relies on subclass for some steps
relies on subclass for some steps
Tea subclass
Coffee subclass
2 Steep the teabag in the water
4 Add lemon
2 Brew the coffee grinds
4 Add sugar and milk
Caffeine Beverage knows and controls the steps of the recipe, and performs steps 1 and 3 itself, but relies on Tea or Coffee to do steps 2 and 4.

The **template method in the abstract base class** controls the algorithm; at certain points in the algorithm, it lets the **subclass supply the implementation of the steps**.

The algorithm lives in one place; code changes have to be done only in the base class.

Example for a template method: Array.Sort() / IComparer


FrameworkClass
+templateMethod()
+stepOne()
+stepTwo()
+stepThree()
stepOne();
stepTwo();
stepThree();
ApplicationClassOne
+stepTwo()
ApplicationClassTwo
+stepTwo()

The **Template Method Pattern** is all about creating a template for an algorithm. The template is just a method that **defines an algorithm as a set of steps**.

One or more of these steps is defined to be **abstract** and has to be implemented by a subclass.

This ensures the algorithm's structure stays unchanged, while subclasses provide some part of the implementation.

The base class can offer optional hook methods to the subclasses

```
public abstract class CaffeineBeverageWithHook {

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

    abstract void brew();

    abstract void addCondiments();

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

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

    boolean customerWantsCondiments() {
        return true;
    }
}
```

We've added a little conditional statement that bases its success on a concrete method, customerWantsCondiments(). If the customer WANTS condiments, only then do we call addCondiments().

Here we've defined a method with a (mostly) empty default implementation. This method just returns true and does nothing else.

This is a **hook** because the subclass can override this method, but doesn't have to.

Hollywood Principle

The Template Method’s abstract class may define:

+ concrete methods (are not present in subclasses)

+ abstract methods (must be overwritten in subclasses)

+ hooks (can be overwritten in subclasses)

You decide which steps of an algorithm are invariant (or standard), and which are variant (or customizable).

The invariant steps are implemented in an abstract base class, while the variant steps are either given a default or no implementation at all. The variant steps represent "placeholders", that can, or must, be supplied by the component's client in a concrete derived class.

The component designer mandates the required steps of an algorithm, and the ordering of the steps, but allows the component client to extend or replace some of these steps.

Strategy and Template Method both encapsulate algorithms:

the first does it by composition and delegation,

the latter by inheritance.

Strategy modifies the logic of individual objects.

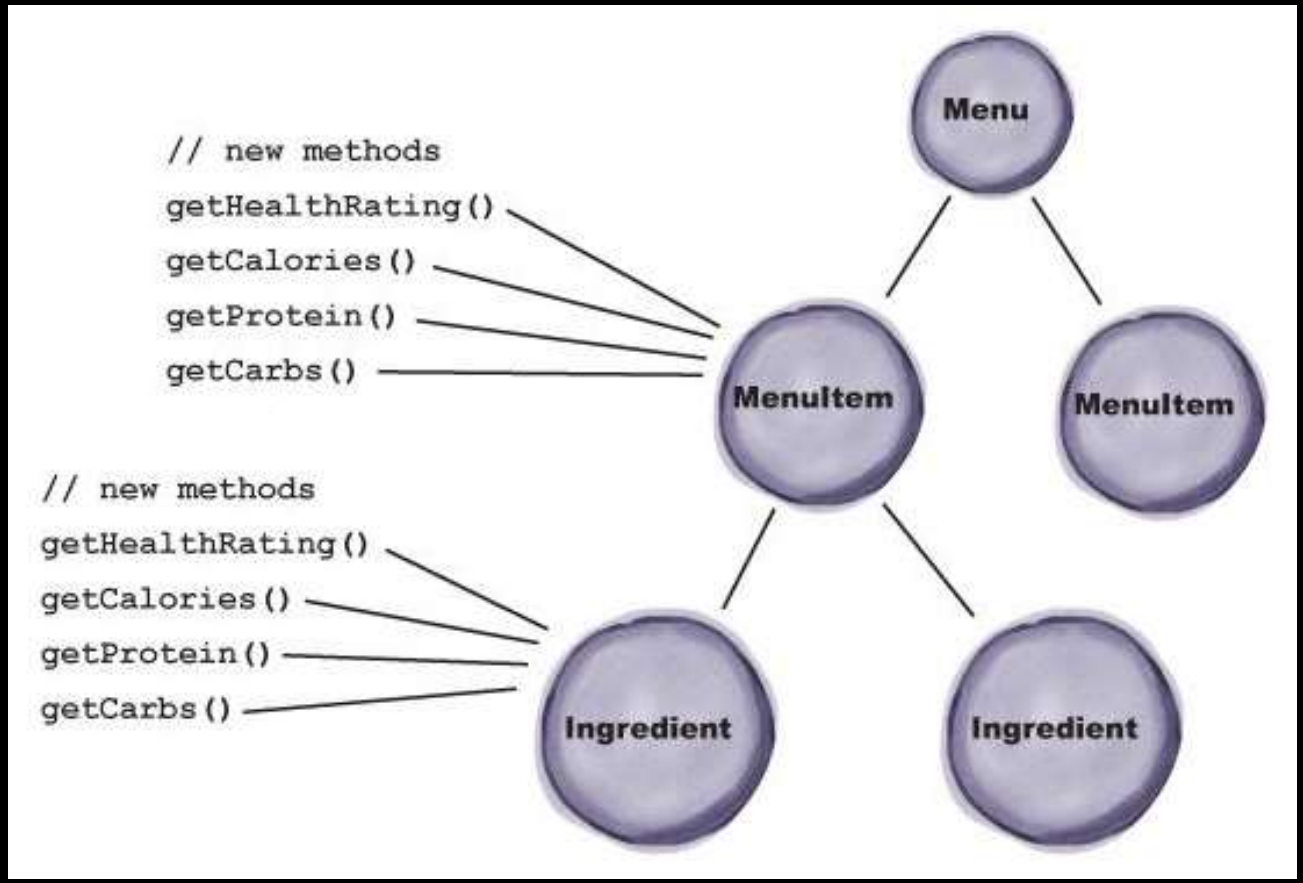
Template Method modifies the logic of an entire class.

The Factory pattern is a specialization of Template Method.

Visitor

Represent an operation to be performed on the elements of an object structure.

Visitor lets you define a new operation without changing the classes of the elements on which it operates.



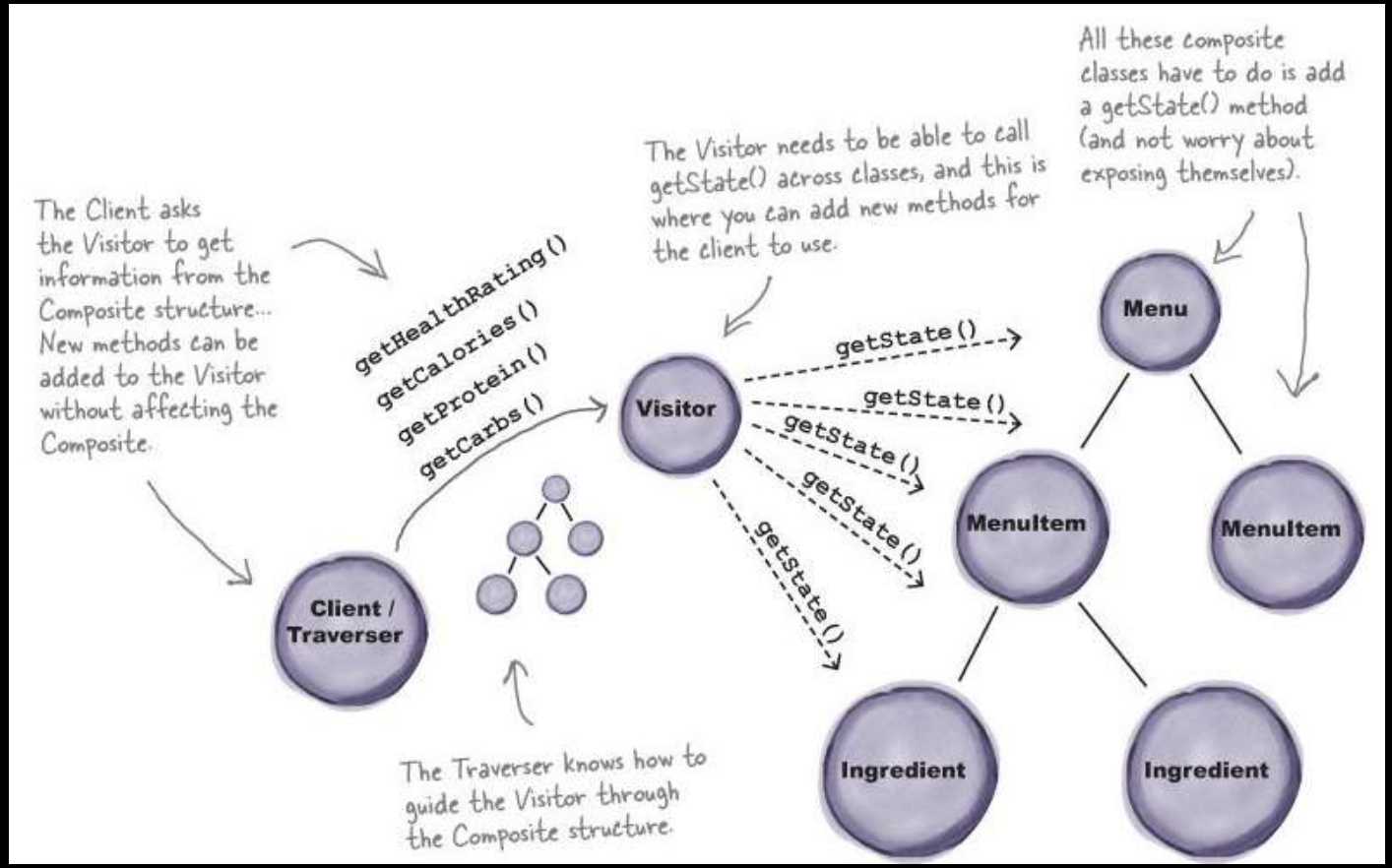
Design without Visitor

Problem:

Many unrelated operations need to be performed on node objects in a heterogeneous aggregate structure.

You don’t want to “pollute” the node classes with these ops.

You don't want to query the type of each node and cast the object to the correct type before performing the operation.



Design with Visitor separating algorithm & data structure

Use the Visitor pattern when you want to add capabilities to a composite of objects and encapsulation is unimportant.

Visitor works hand in hand with an Iterator.

The Iterator navigates to all the objects of a composite.
The visitor performs operations on the object.
To add new operations, only the Visitor changes.

Visitor centralizes the code for operations.

But Visitor breaks the composite classes' encapsulation.

Don't use Visitor if the visited classes are not stable!

The Element hierarchy is coupled only to the Visitor base class, but the Visitor hierarchy is coupled to each Element derived class.

If the stability of the Element hierarchy is low, and the stability of the Visitor hierarchy is high,
consider swapping the 'roles' of the two hierarchies.

Cab company dispatcher

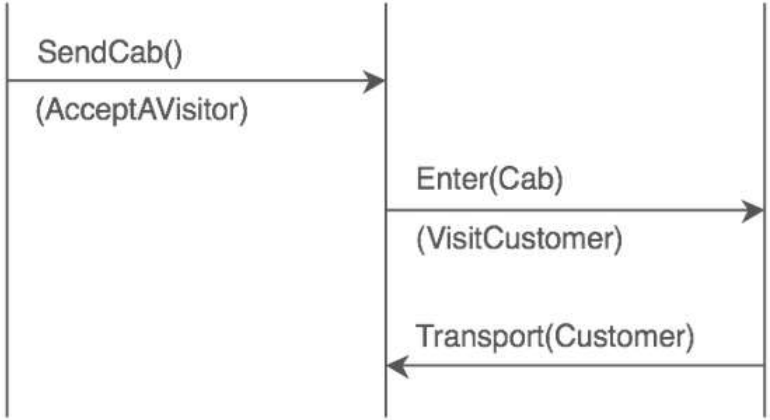
Object structure is list of Customers

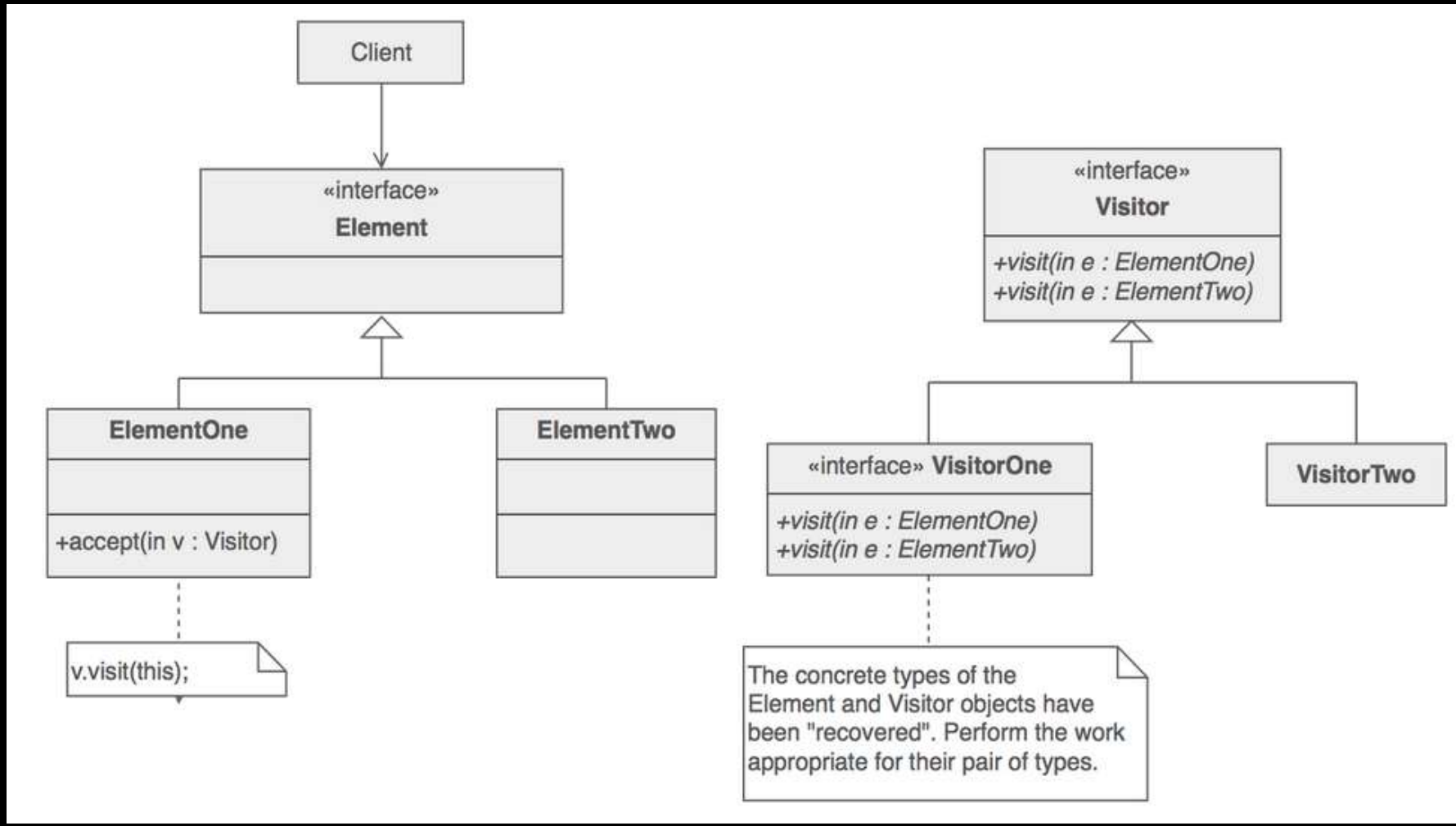
Customer

Concrete element of Customers list

Taxi

Visitor

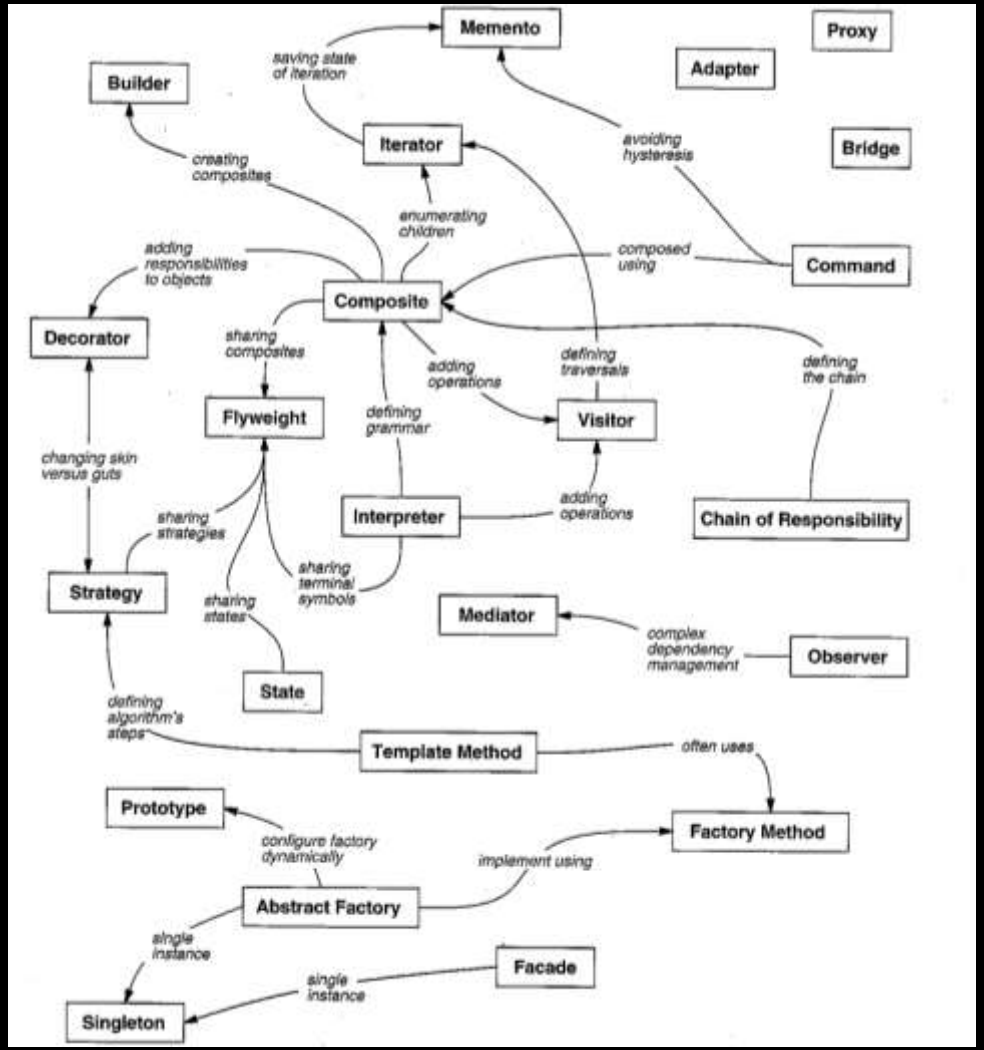



Client
«interface»
Element
ElementOne
+accept(in v : Visitor)
v.visit(this);
ElementTwo
«interface»
Visitor
+visit(in e : ElementOne)
+visit(in e : ElementTwo)
«interface» VisitorOne
+visit(in e : ElementOne)
+visit(in e : ElementTwo)
The concrete types of the Element and Visitor objects have been "recovered". Perform the work appropriate for their pair of types.
VisitorTwo

```
public abstract class Element
{
    public abstract void Accept(BaseVisitorClass visitor);
}
public class ConcreteElementA : Element
{
    public override void Accept(BaseVisitorClass visitor)
    {
        visitor.Visit(this);
    }
    public void OperationA() { }
}
```

```
public abstract class BaseVisitorClass
{
    public abstract void Visit(ConcreteElementA concreteElementA);
    public abstract void Visit(ConcreteElementB concreteElementB);
}

public class VisitorClass1 : BaseVisitorClass
{
    public override void Visit(ConcreteElementA concreteElementA)
    {
    }
    public override void Visit(ConcreteElementB concreteElementB)
    {
    }
}
```

```
public class ObjectStructure
{
    List<Element> _elements = new List<Element>();

    public void Attach(Element element)
    {
        _elements.Add(element);
    }

    public void Detach(Element element)
    {
        _elements.Remove(element);
    }

    public void Accept(BaseVisitorClass visitor)
    {
        foreach (Element element in _elements)
        {
            element.Accept(visitor);
        }
    }
}
```



Pattern Relationships

Reference

- “Design Patterns” by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides (GoF), 1994
  http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612/ref=sr_1_1?ie=UTF8&qid=1442111826&sr=8-1&keywords=design+patterns+gamma
- “Head First Design Patterns” by Eric Freeman et al., 2004
- Judith Bishop: C# 3.0 Design Patterns
- “Design Patterns – Explained Simply”
  https://sourcemaking.com/design_patterns
- “R# Design Patterns – Smart Templates”
  http://www.danylkoweb.com/Blog/using-jetbrains-resharper-9-to-implement-and-learn-design-patterns-AA
- DoFactory - .NET Design Pattern Framework (Code Library)
  http://www.dofactory.com/net/design-patterns

Summary

Speed up your development by using standard proven design patterns. They make your code less error-prone and easier to maintain.

And they give us a common language to communicate design ideas better.

Questions

Slides and code: http://heitland-it.de