Design Patterns

Ahmed Hesham

Pattern Groups

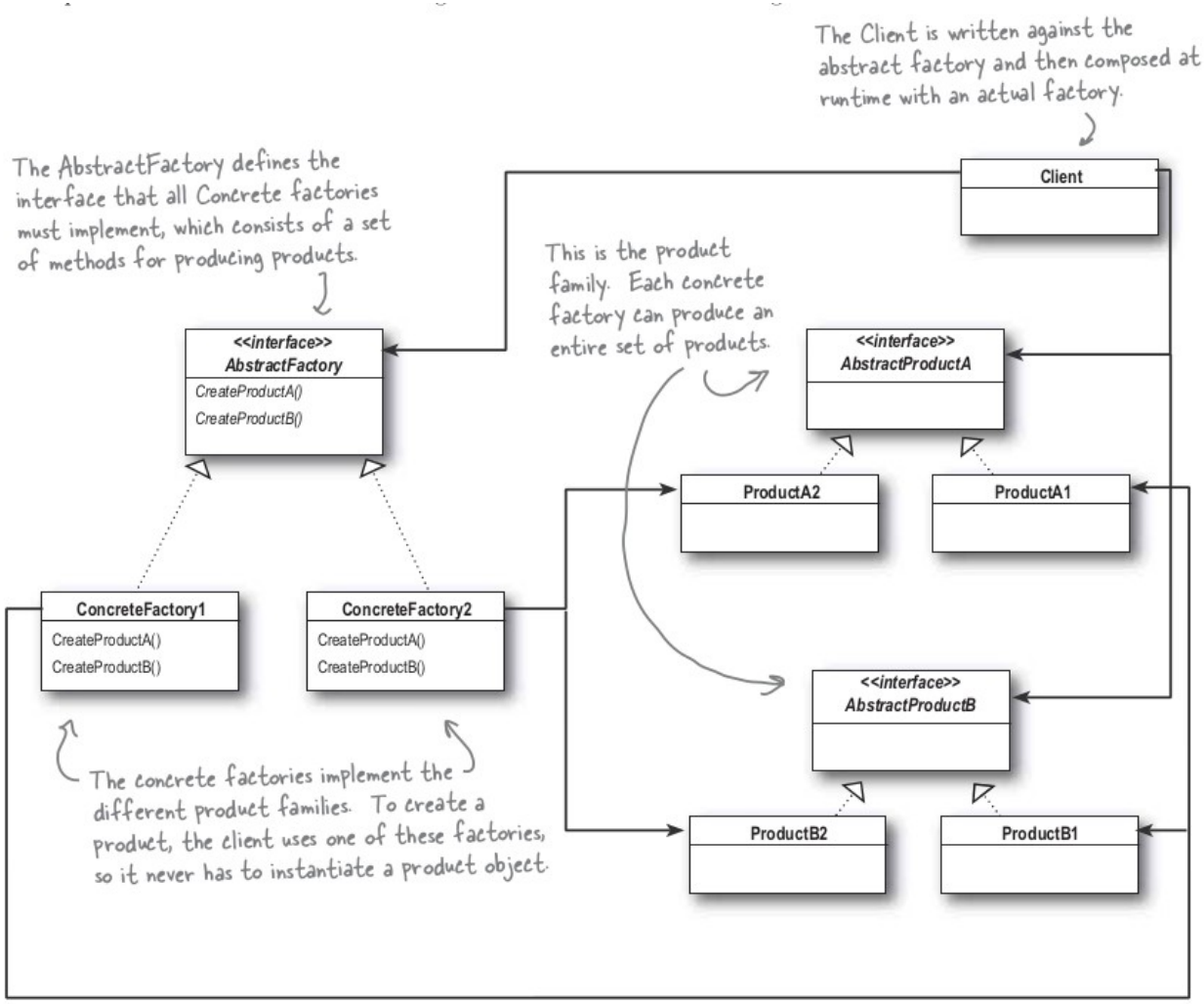
- Creational
- Structural
- Behavioral

Creational Patterns

- **Used to construct objects such that they can be decoupled from their implementing systems**

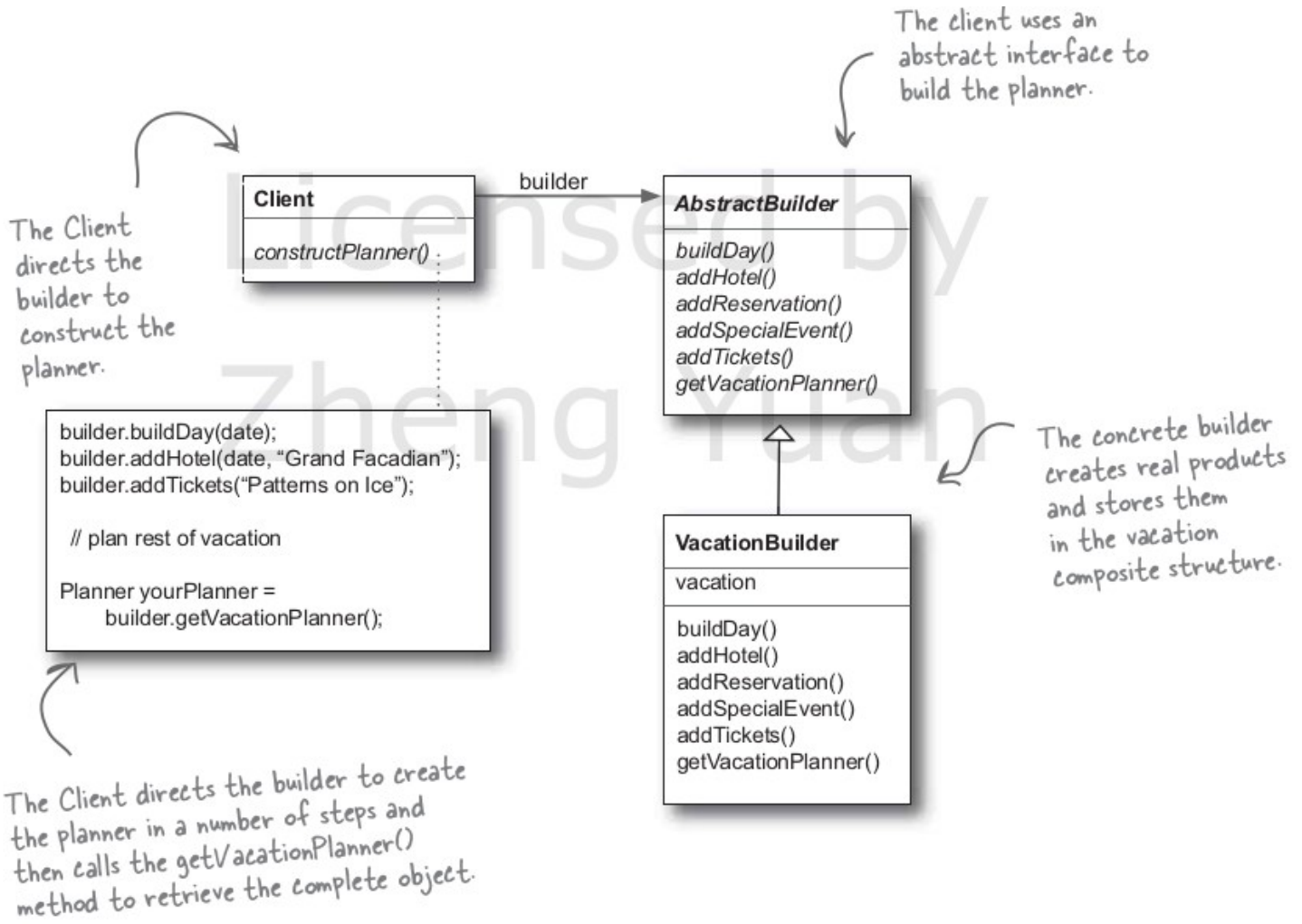
Abstract Factory

- **Provides an interface for creating families of similar/dependent objects without specifying their concrete classes, rather it delegates the creation to the concrete class**


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.

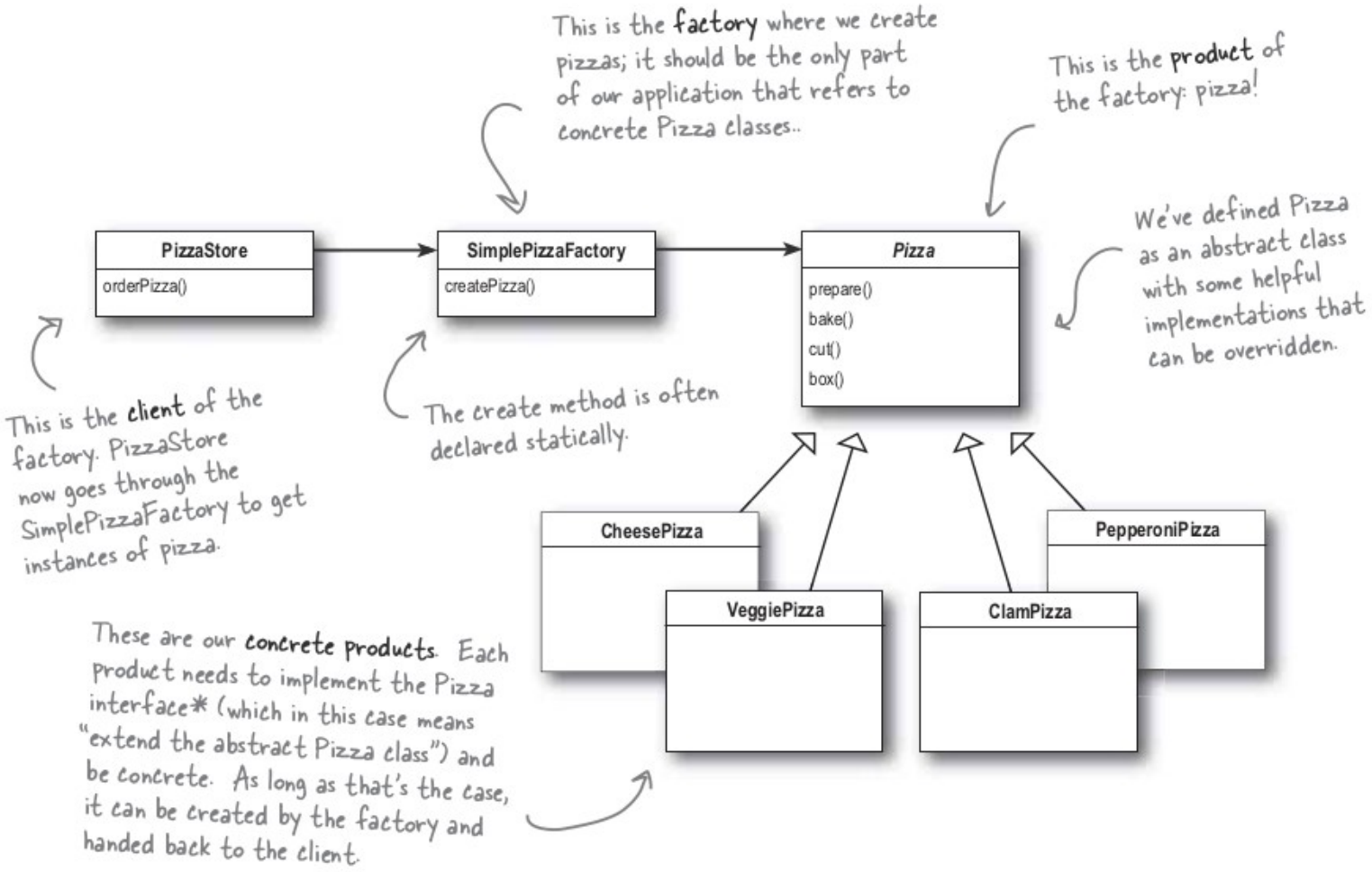
Builder

- **Used to encapsulate the construction of an object and allow it to be constructed in steps**



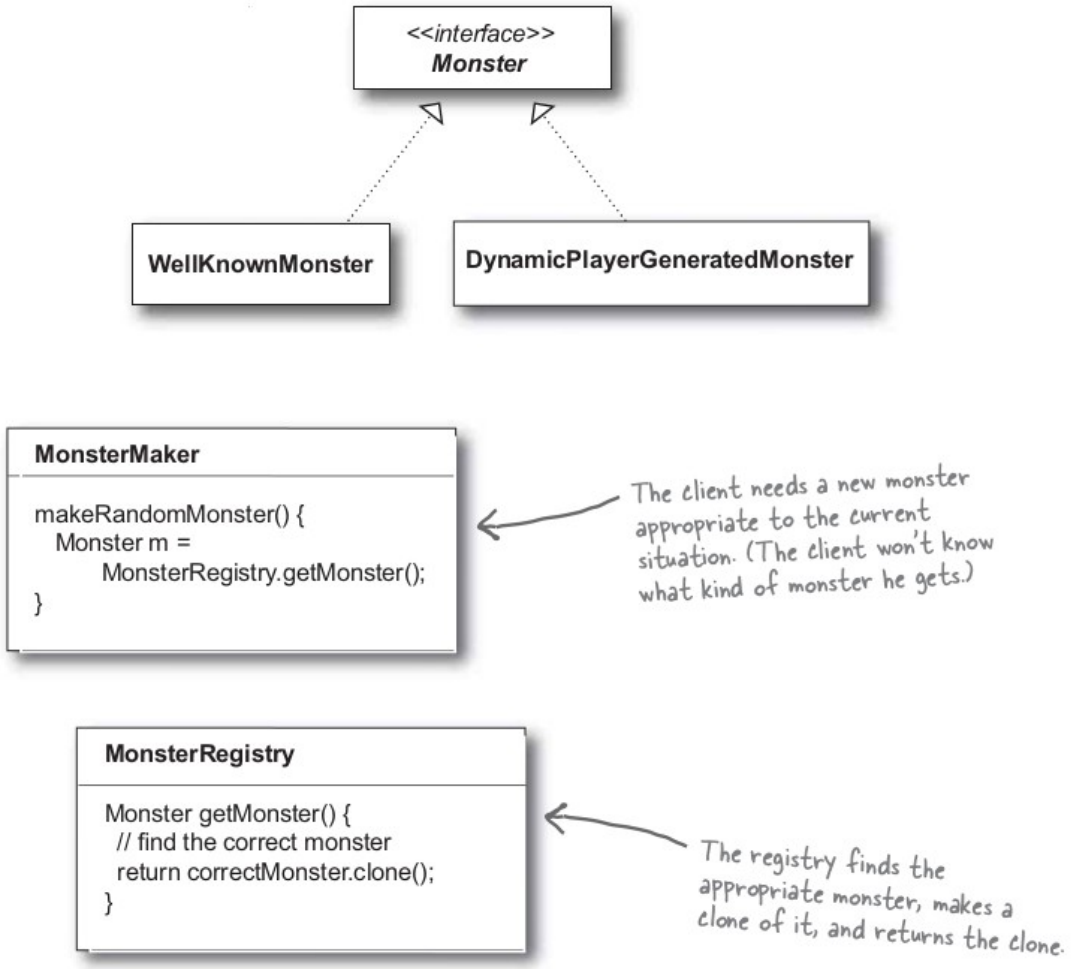
Factory

- **Exposes methods for creating objects but lets subclasses decide on which class to instantiate**


This is the factory where we create pizzas; it should be the only part of our application that refers to concrete Pizza classes..
This is the product of the factory: pizza!
We've defined Pizza as an abstract class with some helpful implementations that can be overridden.
PizzaStore
orderPizza()
SimplePizzaFactory
createPizza()
Pizza
prepare()
bake()
cut()
box()
This is the client of the factory. PizzaStore now goes through the SimplePizzaFactory to get instances of pizza.
The create method is often declared statically.
CheesePizza
VeggiePizza
ClamPizza
PepperoniPizza
These are our concrete products. Each product needs to implement the Pizza interface* (which in this case means "extend the abstract Pizza class") and be concrete. As long as that's the case, it can be created by the factory and handed back to the client.

Prototype

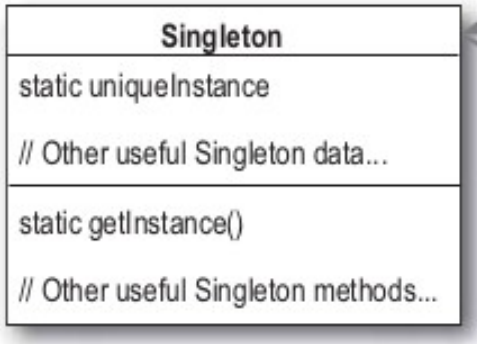
- **Used when creating an instance of a class is complex or expensive. Instances of an object are created by copying other objects**


<<interface>>
Monster
WellKnownMonster
DynamicPlayerGeneratedMonster
MonsterMaker
makeRandomMonster() {
Monster m =
MonsterRegistry.getMonster();
}
The client needs a new monster appropriate to the current situation. (The client won't know what kind of monster he gets.)
MonsterRegistry
Monster getMonster() {
// find the correct monster
return correctMonster.clone();
}
The registry finds the appropriate monster, makes a clone of it, and returns the clone.

Singleton

- **Used when ensuring that one instance of an object exists throughout the system**

The getInstance() method is static, which means it's a class method, so you can conveniently access this method from anywhere in your code using Singleton.getInstance(). That's just as easy as accessing a global variable, but we get benefits like lazy instantiation from the Singleton.



The uniqueInstance class variable holds our one and only instance of Singleton.

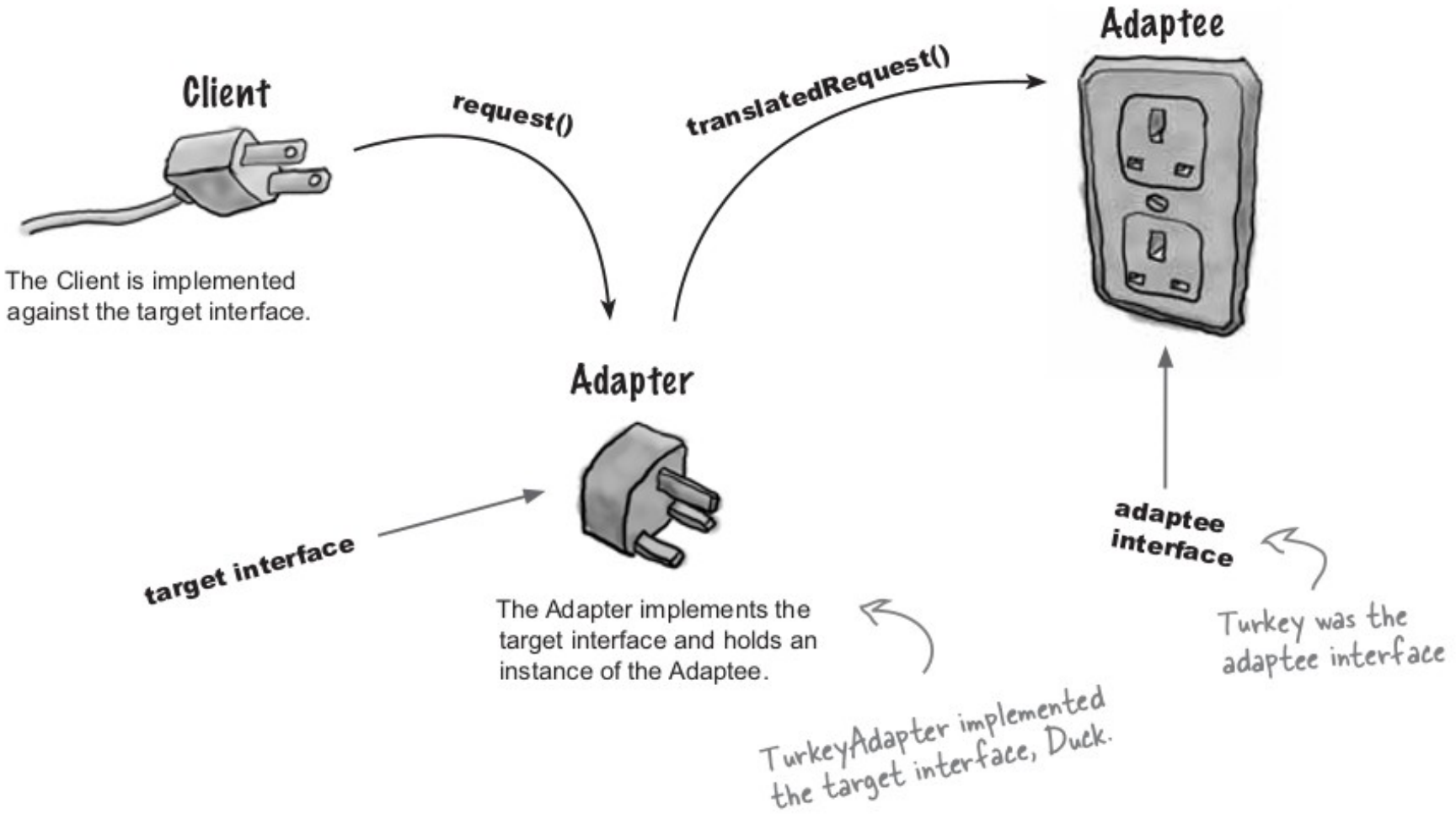
A class implementing the Singleton Pattern is more than a Singleton; it is a general purpose class with its own set of data and methods.

Structural Patterns

- **Used to form large object structures between many different objects**

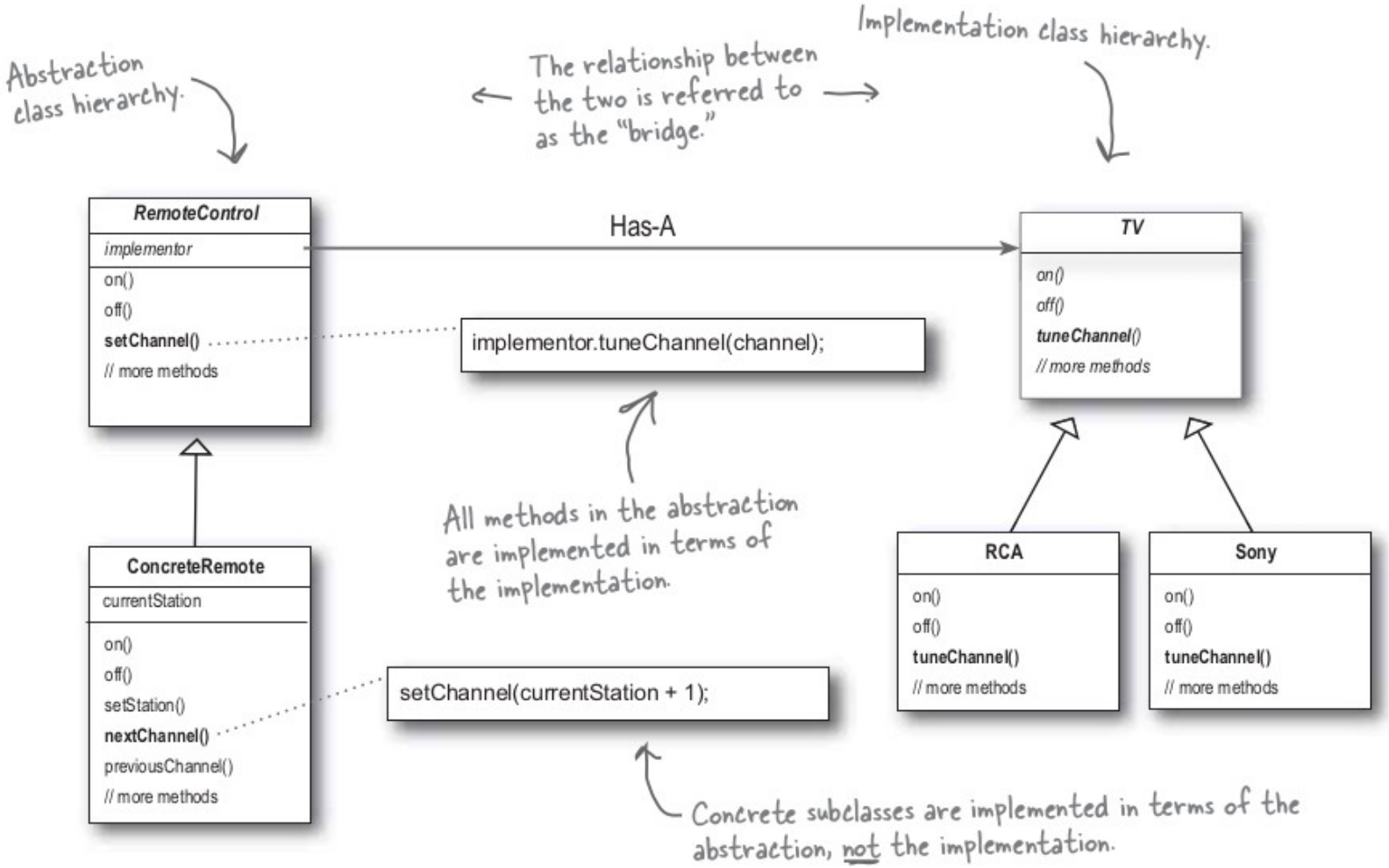
Adapter

- **Allows different classes to work together by acting as a binding glue**


Client
request()
translatedRequest()
Adaptee
The Client is implemented against the target interface.
Adapter
target interface
adaptee interface
The Adapter implements the target interface and holds an instance of the Adaptee.
Turkey was the adaptee interface
TurkeyAdapter implemented the target interface, Duck.

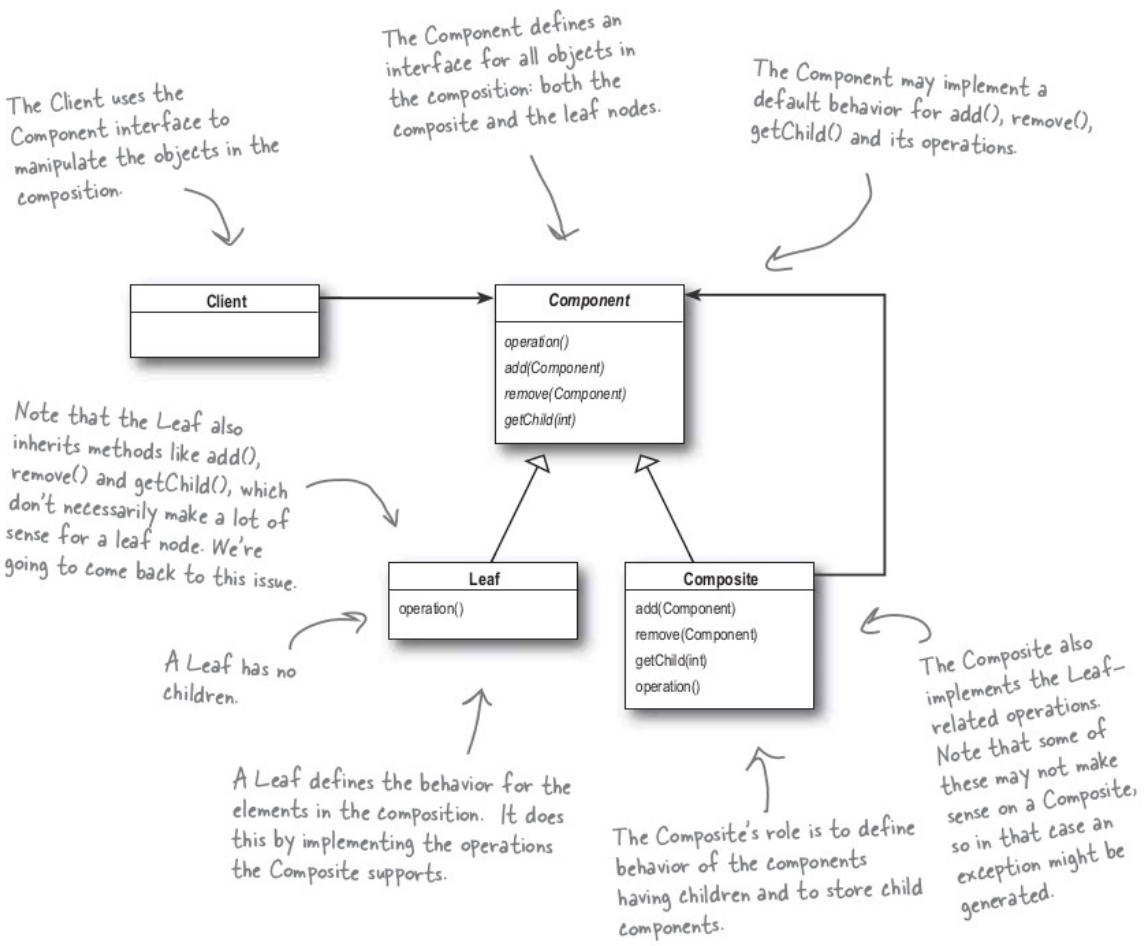
Bridge

- **Allows for variable implementation of an object and the abstraction of it by creating a class hierarchy for the implementation and another for abstraction**


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()
setStation()
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.

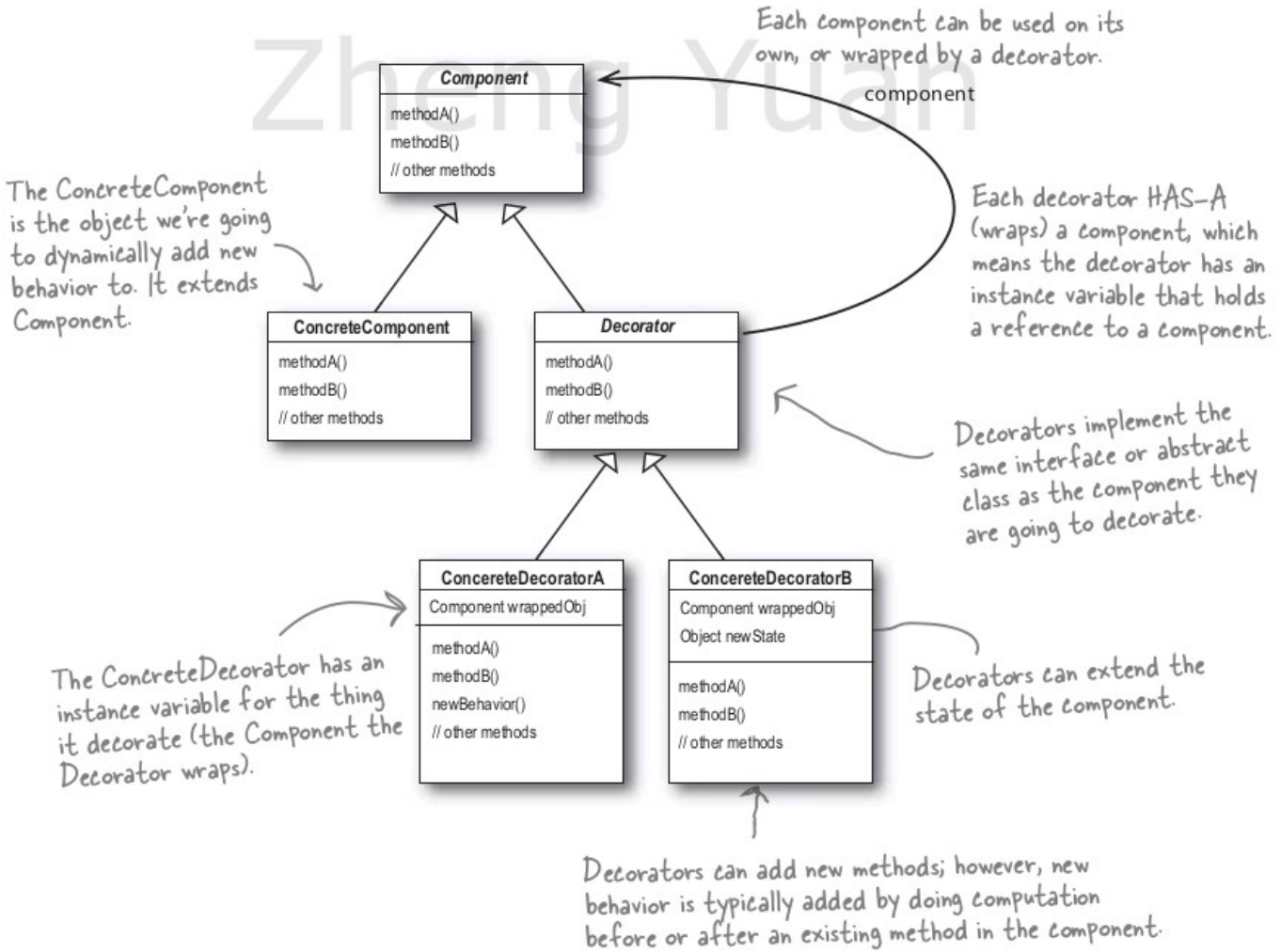
Composite

- **Models the objects into a tree such that operations can be applied to each sub-trees or all objects at once**


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.

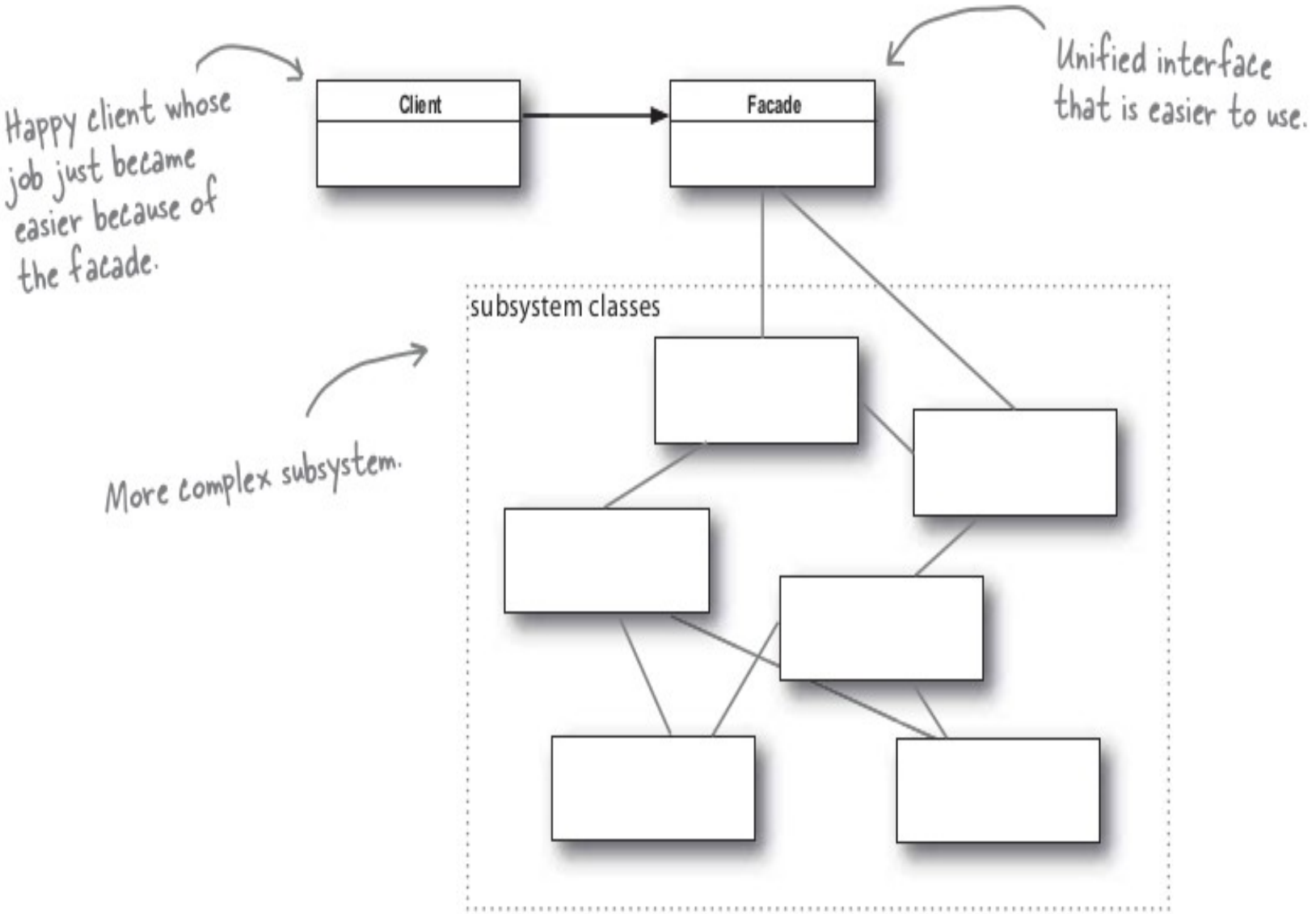
Decorator

- **Allows behavior to be added to an individual object, either statically or dynamically, without affecting the behavior of other objects from the same class**


Each component can be used on its own, or wrapped by a decorator.
component
Component
methodA()
methodB()
// other methods
The ConcreteComponent is the object we're going to dynamically add new behavior to. It extends Component.
Each decorator HAS-A (wraps) a component, which means the decorator has an instance variable that holds a reference to a component.
ConcreteComponent
methodA()
methodB()
// other methods
Decorator
methodA()
methodB()
// other methods
Decorators implement the same interface or abstract class as the component they are going to decorate.
ConcereteDecoratorA
Component wrappedObj
methodA()
methodB()
newBehavior()
// other methods
ConcereteDecoratorB
Component wrappedObj
Object newState
methodA()
methodB()
// other methods
The ConcreteDecorator has an instance variable for the thing it decorate (the Component the Decorator wraps).
Decorators can extend the state of the component.
Decorators can add new methods; however, new behavior is typically added by doing computation before or after an existing method in the component.

Facade

- **Hides the complexities of the larger system and provides a simpler interface to the client. It typically involves a single wrapper class that contains a set of members required by the client. These members access the system on behalf of the facade client and hide the implementation details**

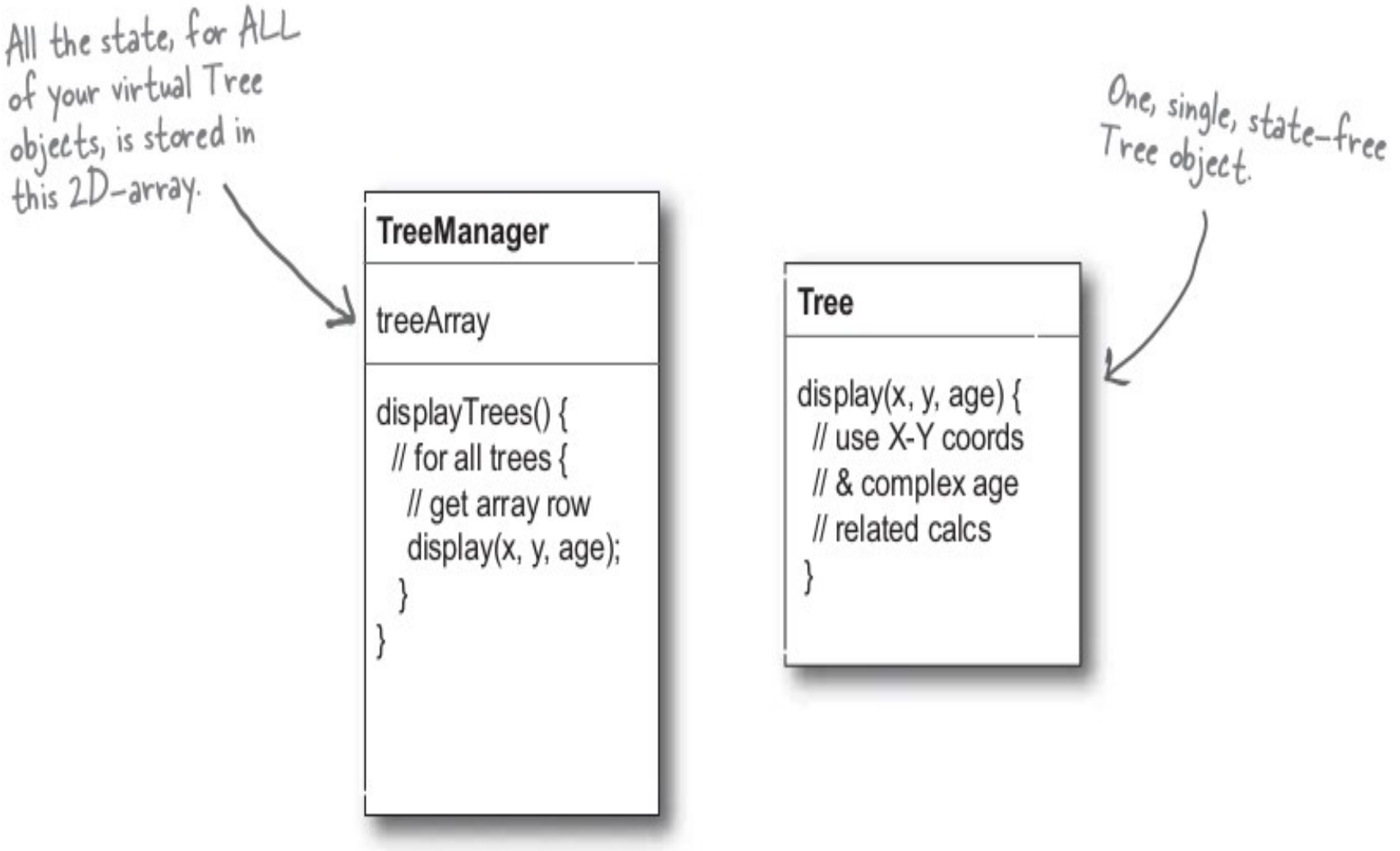

Happy client whose job just became easier because of the facade.
Client
Facade
Unified interface that is easier to use.
subsystem classes
More complex subsystem.

Flyweight

- **Shares data as much as possible with other objects to minimize memory usage**

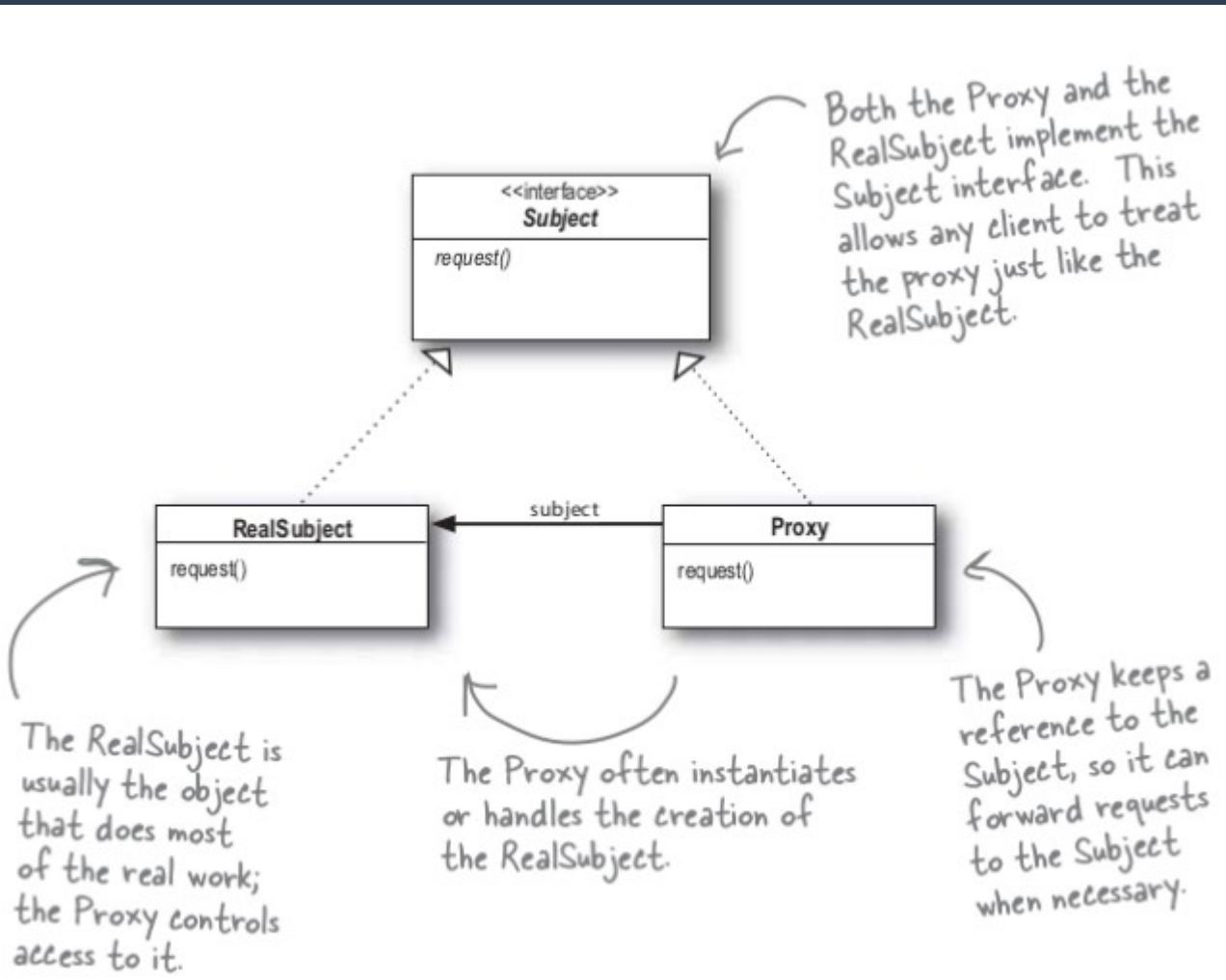
Flyweight – example

- A landscape design program that can render trees as object based on their (x,y) coordinates and age


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
}

Proxy

- **Provides a placeholder for another object to access it**


<<interface>>
Subject
request()
Both the Proxy and the RealSubject implement the Subject interface. This allows any client to treat the proxy just like the RealSubject.
RealSubject
request()
subject
Proxy
request()
The RealSubject is usually the object that does most of the real work; the Proxy controls access to it.
The Proxy often instantiates or handles the creation of the RealSubject.
The Proxy keeps a reference to the Subject, so it can forward requests to the Subject when necessary.

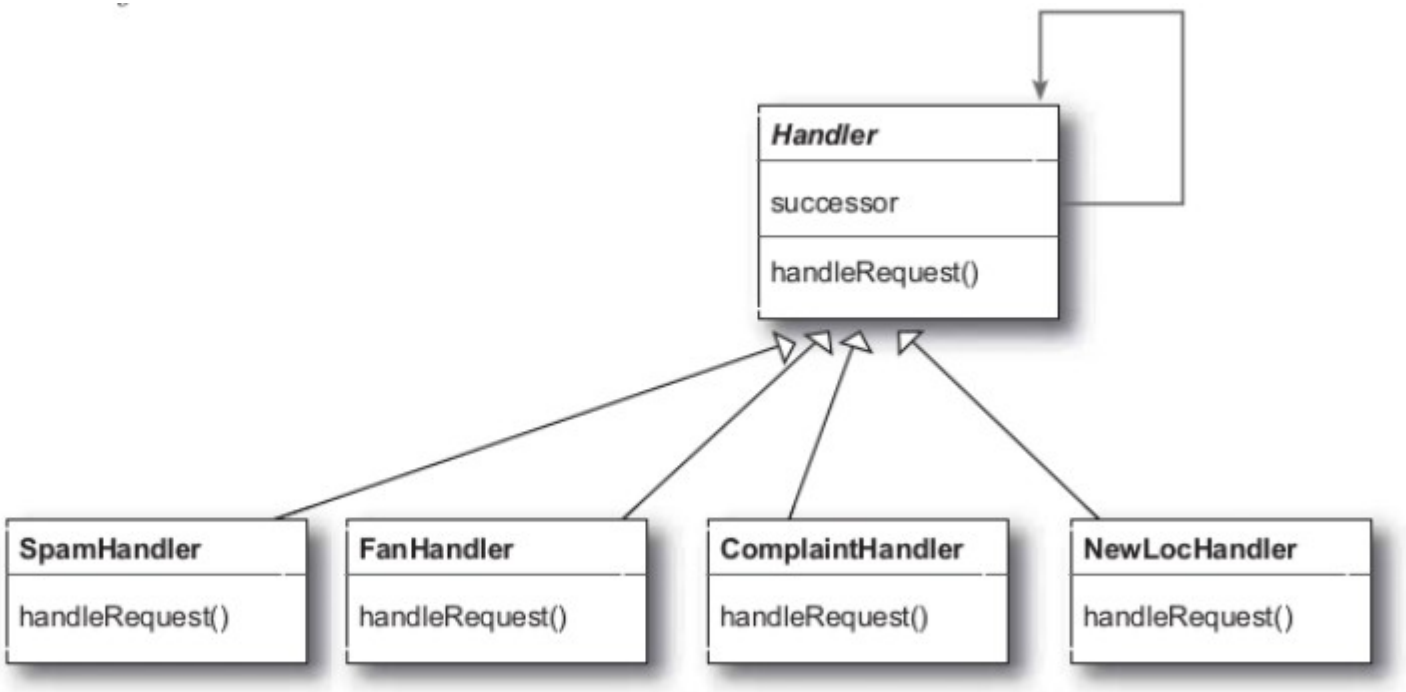
Behavioral

- **Used to manage algorithms, relationships and responsibilities between objects**

Chain Of Responsibility

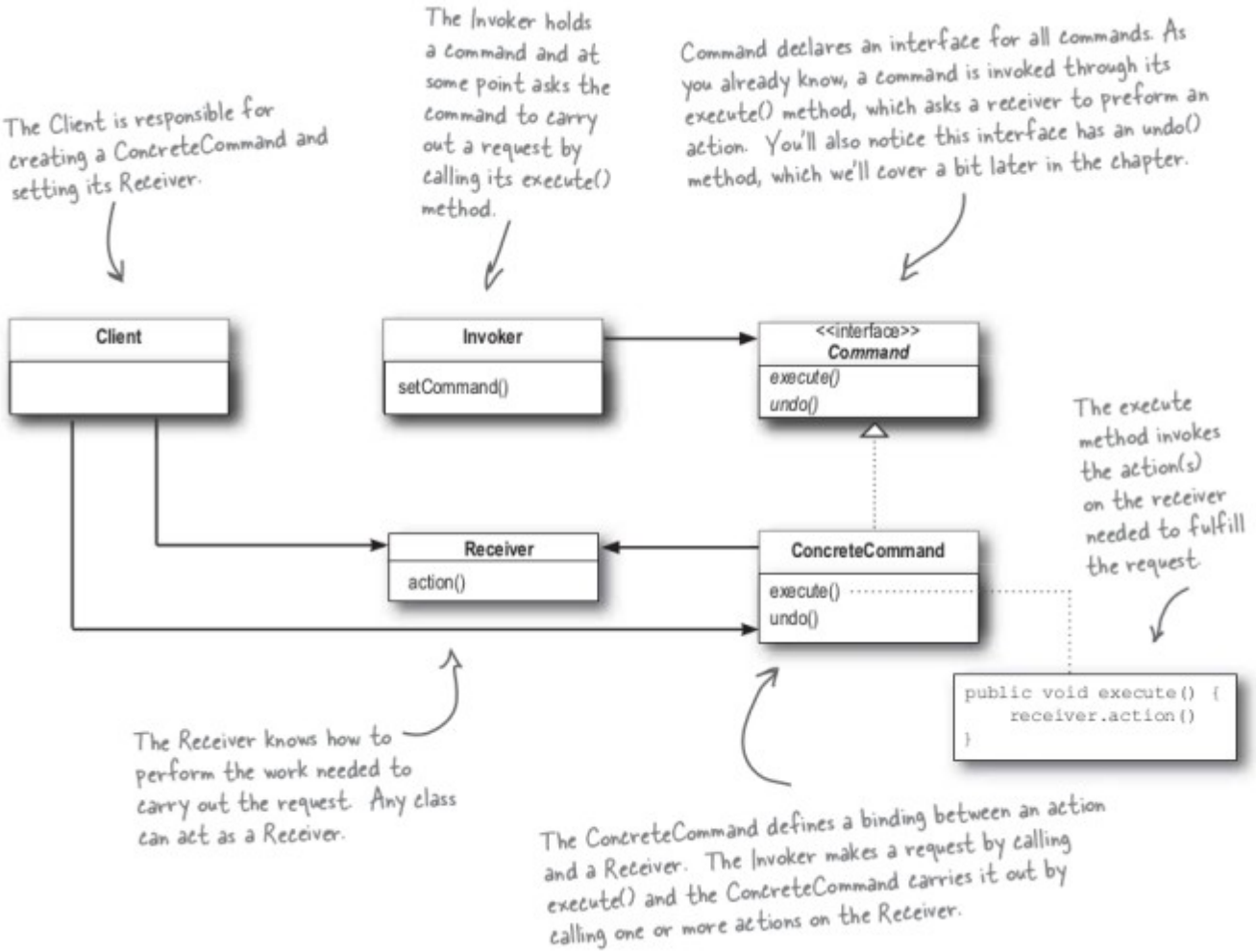
- **Repeatedly passes a request down to objects in a chain and each object can either handle the request or pass it down to the next object (Netty Pipeline)**

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.



Command

- An object is used to encapsulate all information needed to perform an action or trigger an event at a later time. This information includes the method name, the object that owns the method and values for the method parameters, making it easier to construct general components that can delegate method calls when appropriate without the need to know the class of the method or the method parameter

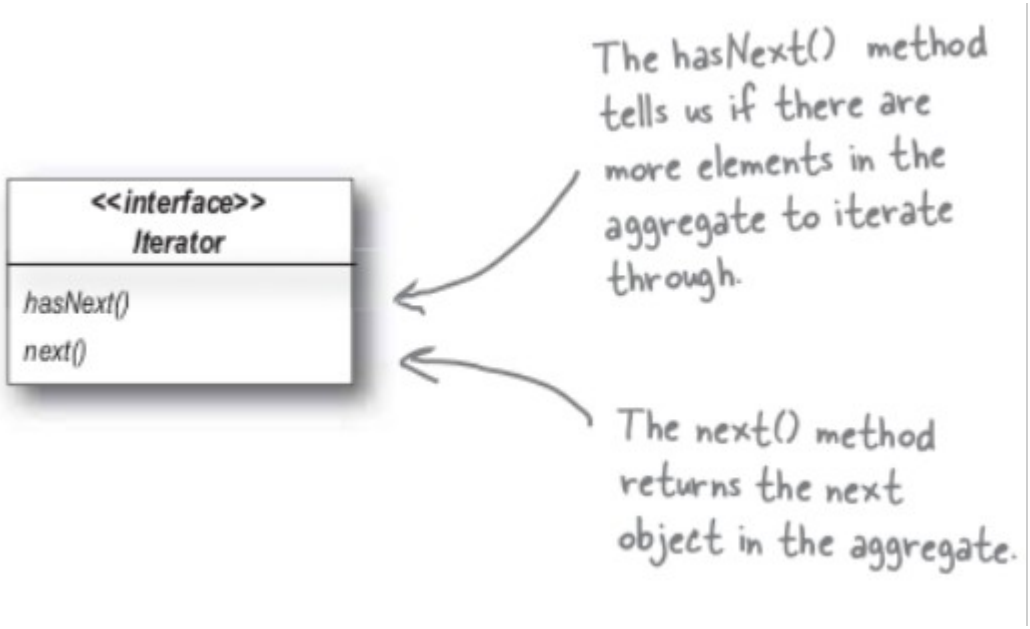

The Client is responsible for creating a ConcreteCommand and setting its Receiver.
The Invoker holds a command and at some point asks the command to carry out a request by calling its execute() method.
Command declares an interface for all commands. As you already know, a command is invoked through its execute() method, which asks a receiver to preform 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.

Interpreter

- Specifies how to evaluate sentences in a language usually through the use of context free grammars (CFGs)
- Example: python is a language written using this pattern

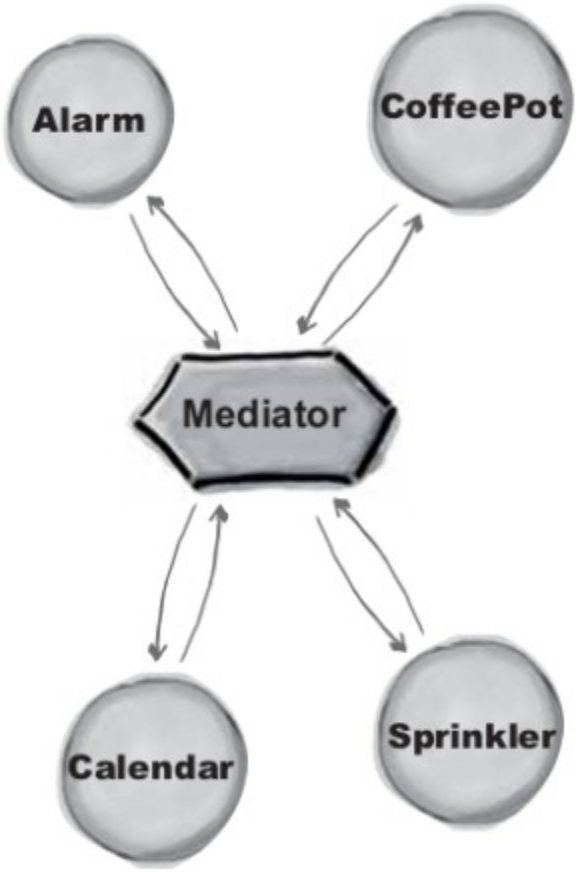
Iterator

- **Utilizes and object (iterator) to traverse the content of another object**


<<interface>>
Iterator
hasNext()
next()
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.

Mediator

- **Provides a centralized communication platform between objects. Objects no longer communicate directly with each other, but instead communicate through the mediator. Thus, reducing dependencies between objects, thereby reducing coupling (Messaging Queues)**



Mediator

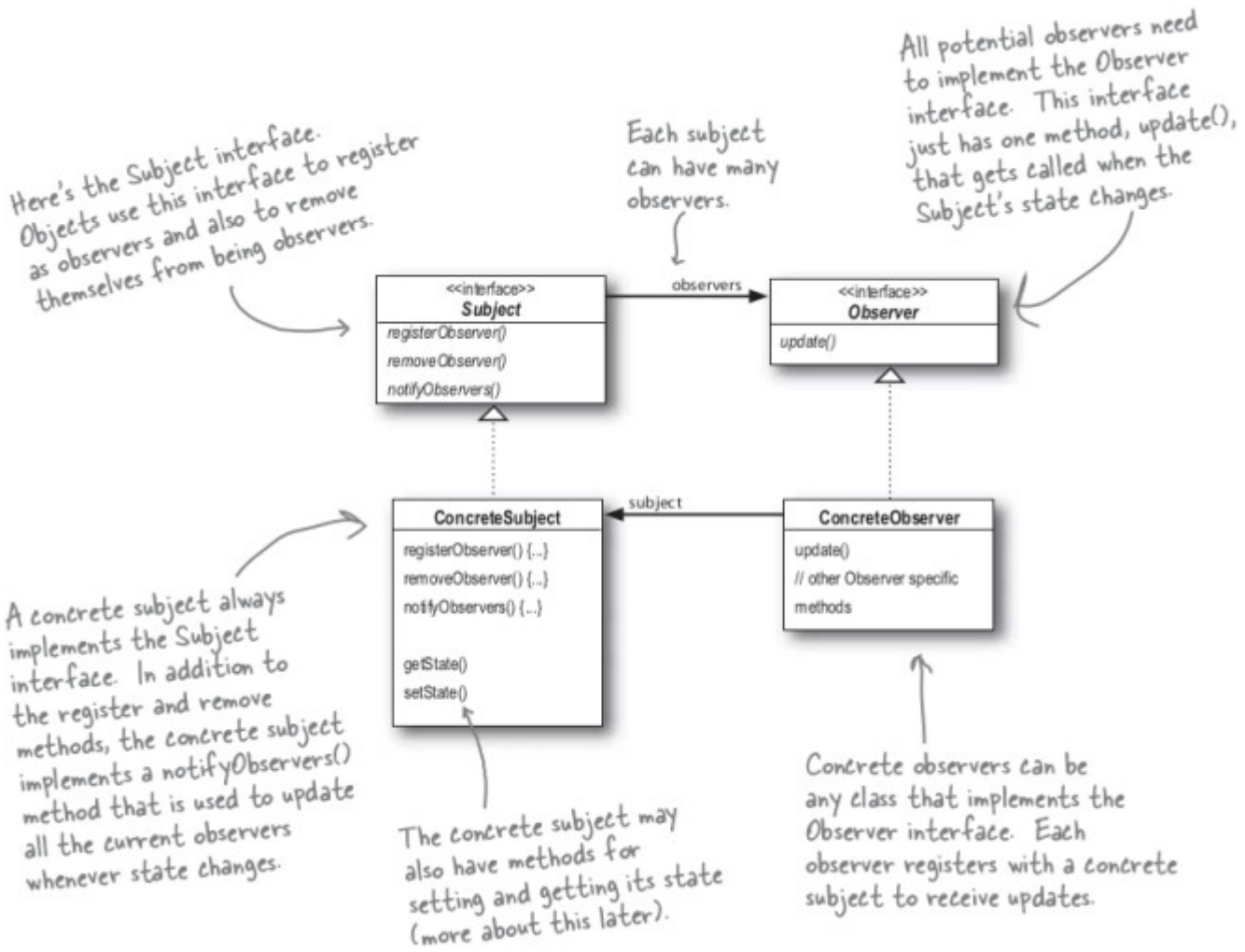
```
if(alarmEvent){
 checkCalendar()
 checkShower()
 checkTemp()
}
if(weekend) {
 checkWeather()
 // do more stuff
}
if(trashDay) {
 resetAlarm()
 // do more stuff
}
```

Memento

- **Provides the ability to restore an object to its previous state**

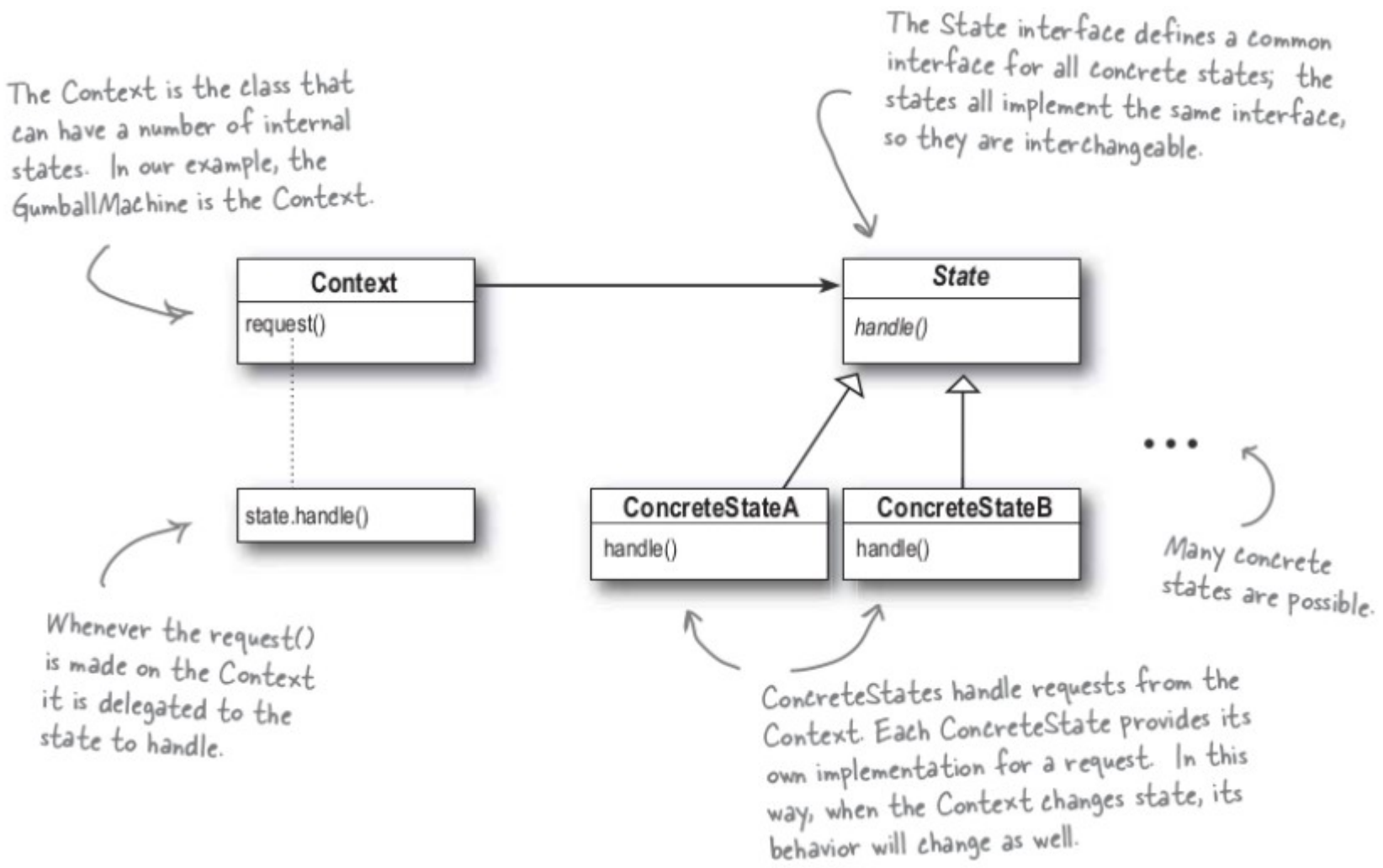
Observer

- **Defines a one to many relation between objects so that when one object changes, all of its dependents are notified**


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.

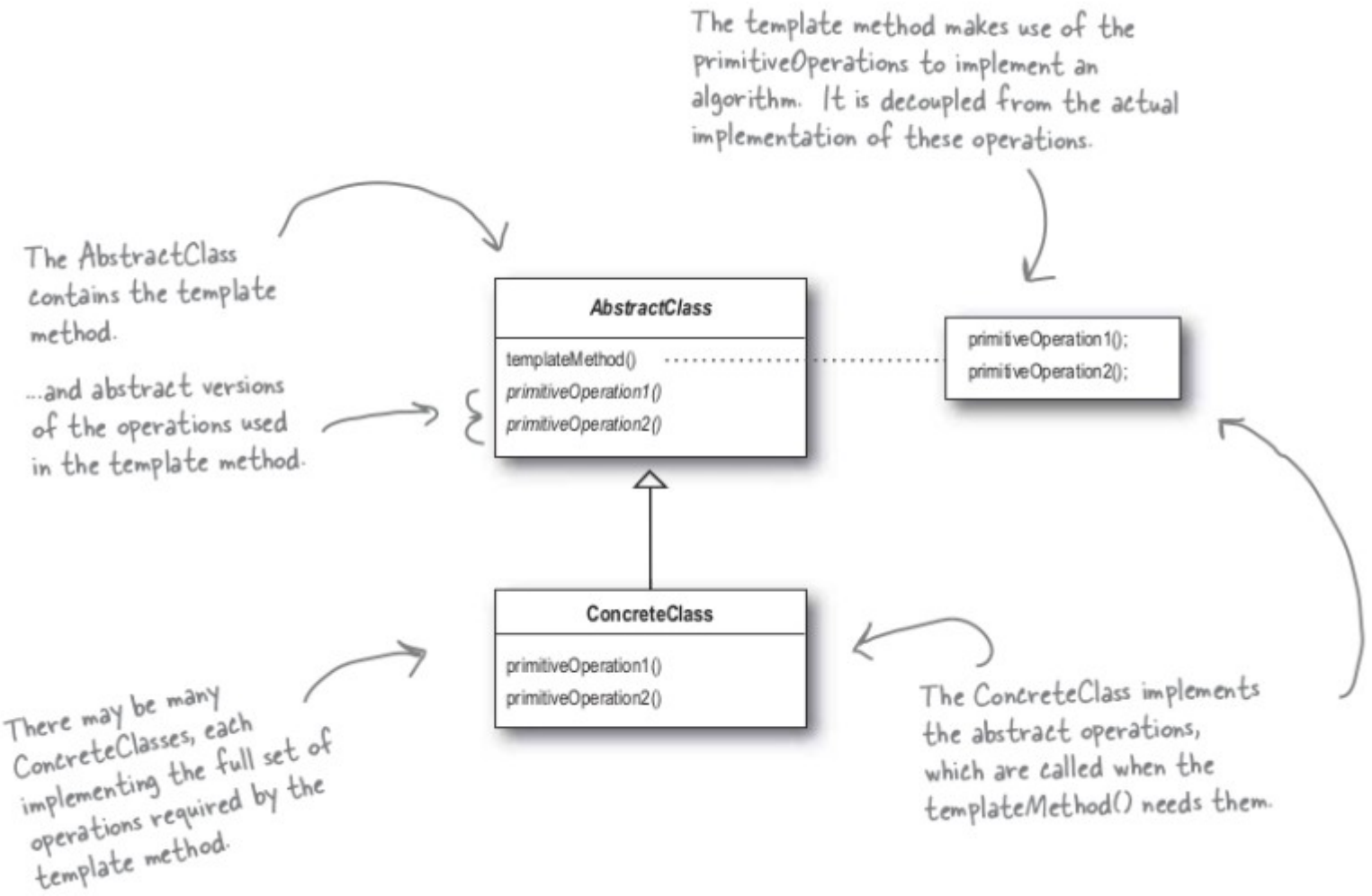
State

- **Allows an object to alter its behavior when its internal state changes**


The Context is the class that can have a number of internal states. In our example, the GumballMachine is the Context.
The State interface defines a common interface for all concrete states; the states all implement the same interface, so they are interchangeable.
Context
request()
State
handle()
state.handle()
ConcreteStateA
handle()
ConcreteStateB
handle()
Many concrete states are possible.
Whenever the request() is made on the Context it is delegated to the state to handle.
ConcreteStates handle requests from the Context. Each ConcreteState provides its own implementation for a request. In this way, when the Context changes state, its behavior will change as well.

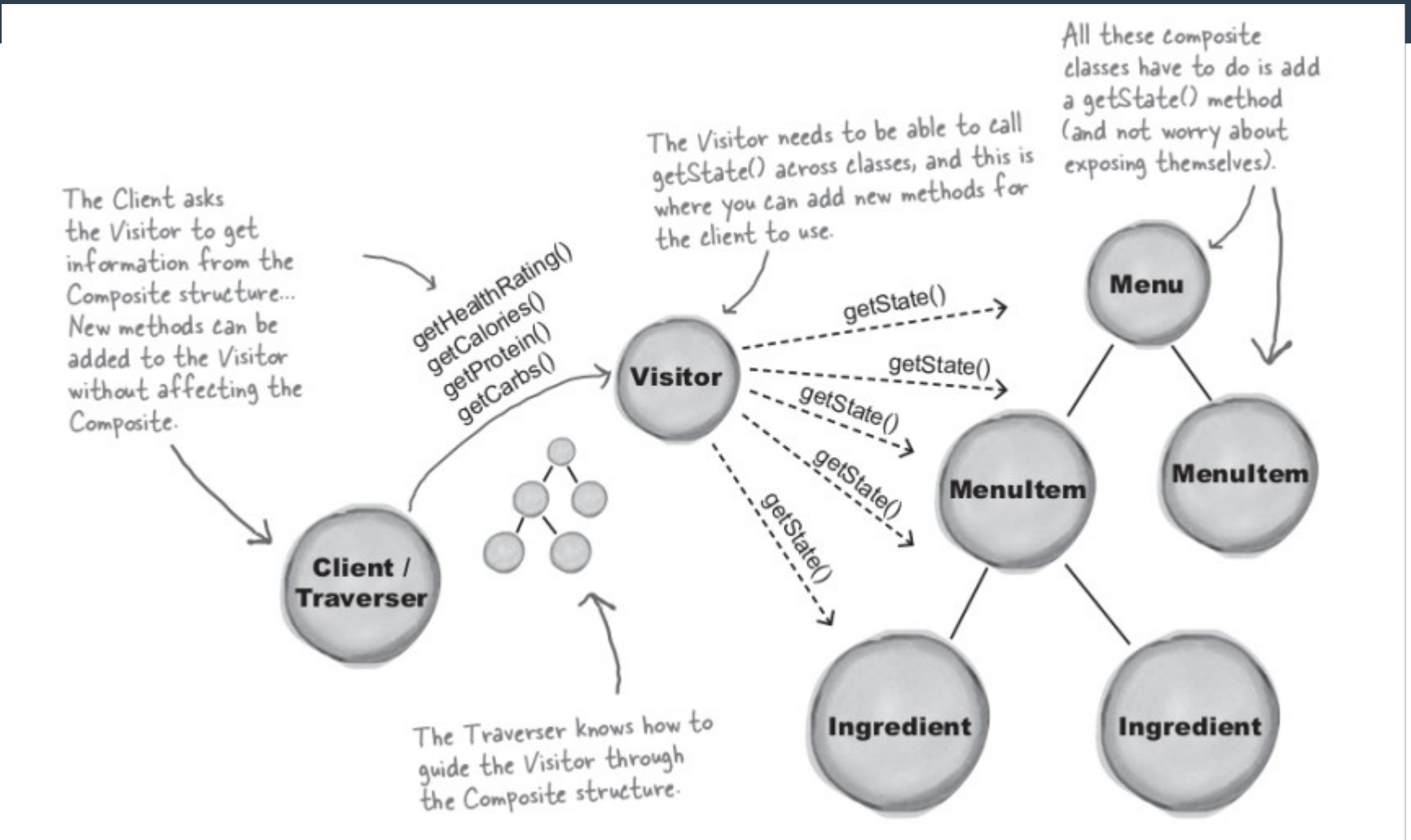
Template Method

- **Defines the skeleton of an algorithm in one step, deferring some steps to the other classes. It additionally allows subclasses to redefine certain steps without changing the algorithm structure**


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

Visitor

- **Allows adding new virtual functions to a family of classes, without modifying the classes**

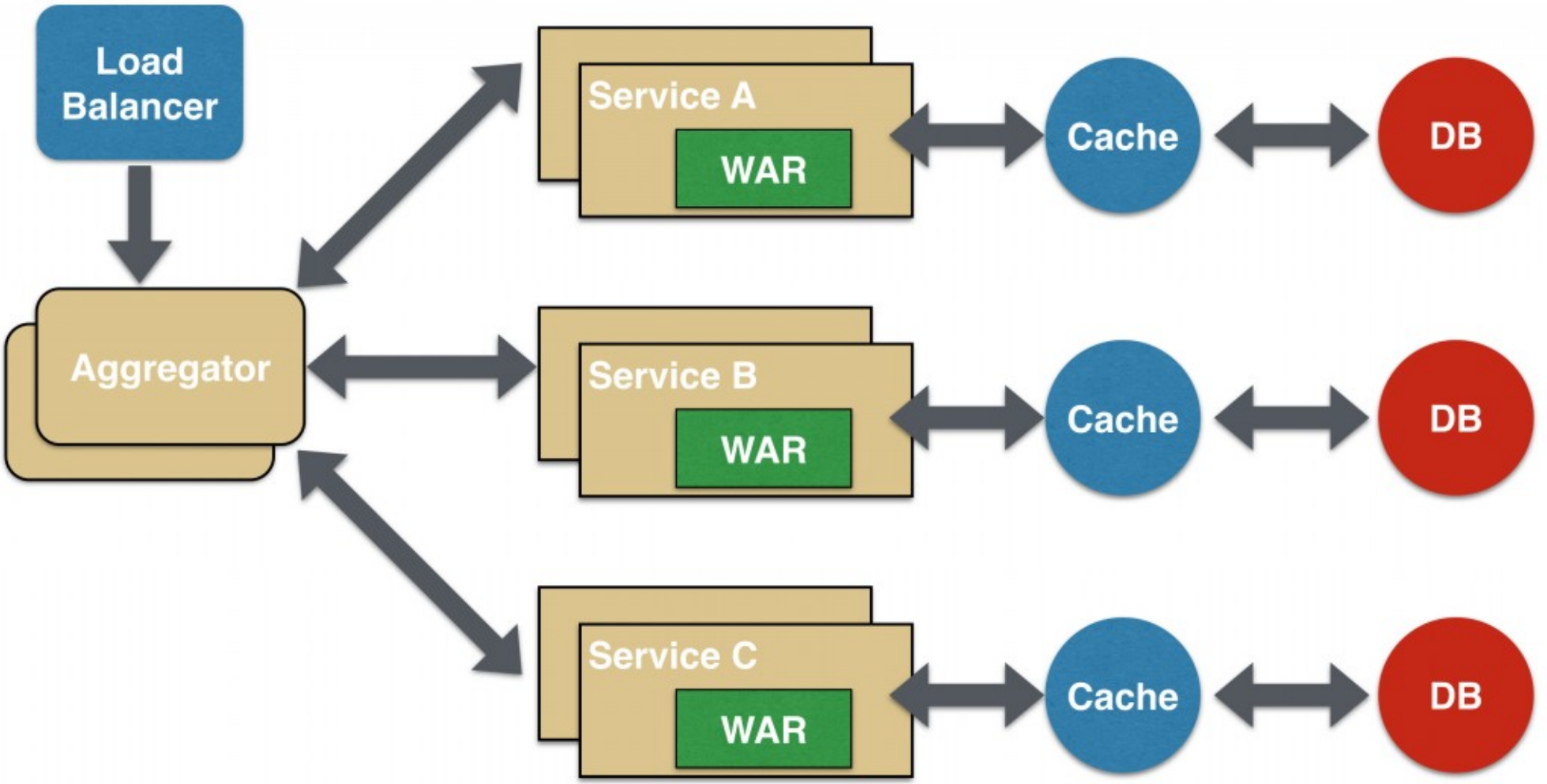

All these composite classes have to do is add a getState() method (and not worry about exposing themselves).
The Visitor needs to be able to call getState() across classes, and this is where you can add new methods for the client to use.
The Client asks the Visitor to get information from the Composite structure... New methods can be added to the Visitor without affecting the Composite.
getHealthRating()
getCalories()
getProtein()
getCarbs()
Visitor
getState()
getState()
getState()
getState()
getState()
Menu
MenuItem
MenuItem
Client / Traverser
Ingredient
Ingredient
The Traverser knows how to guide the Visitor through the Composite structure.

Modern Design Patterns

- **New design patterns have emerged to conform to the rise of microservices**

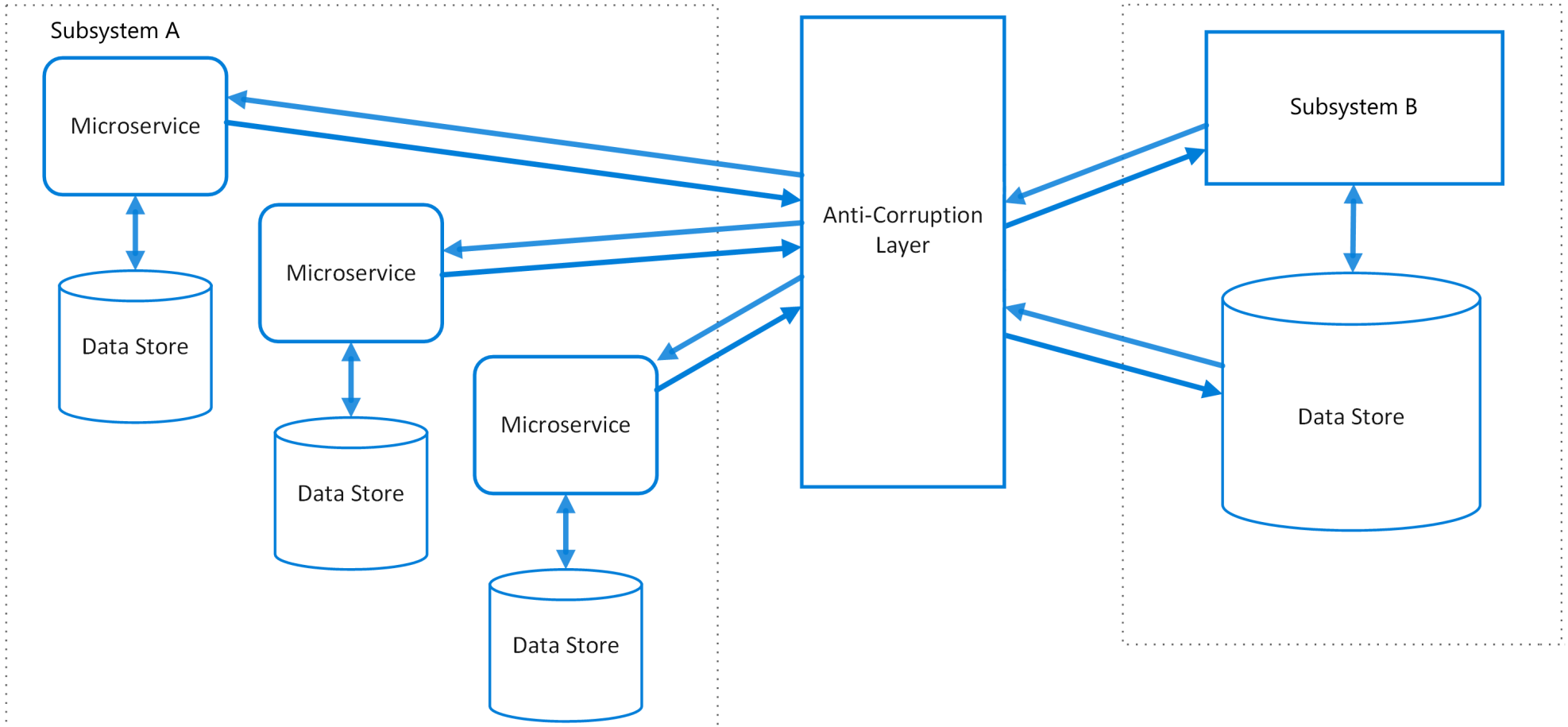
Aggregator

- A common unit named the aggregator requests all the data for the given user request
- Data is generated from different sources and merged at the aggregator


Load Balancer
Aggregator
Service A
WAR
Service B
WAR
Service C
WAR
Cache
DB
Cache
DB
Cache
DB

Anticorruption Layer

- A new layer is added between two subsystems in an application that translate requests from either side
- Usually used when migrating a subsystem to newer communication form
- Adds an overhead for each communication


Subsystem A
Microservice
Data Store
Microservice
Data Store
Microservice
Data Store
Anti-Corruption Layer
Subsystem B
Data Store

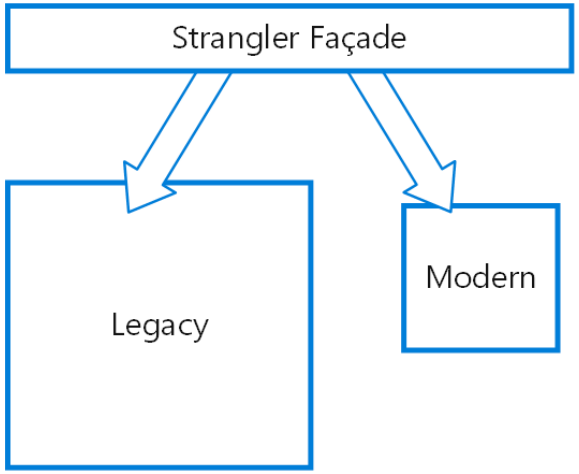
Sidecar

- Separate components of an application into a separate process or container to provide isolation and encapsulation
- Allows applications to be composed of heterogeneous components and technologies

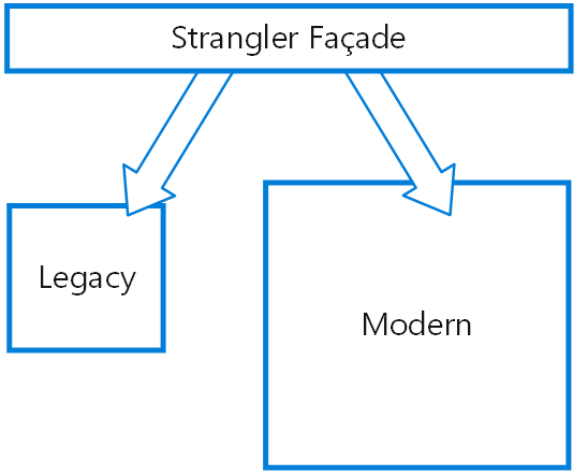
Strangler

- Incrementally migrates a legacy system by replacing specific pieces of functionality with new applications and services
- A new facade layer is developed above the system to route the requests to the correct place

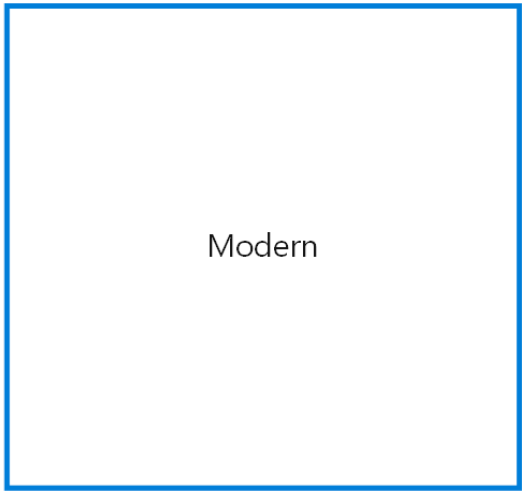
Early migration



Later migration

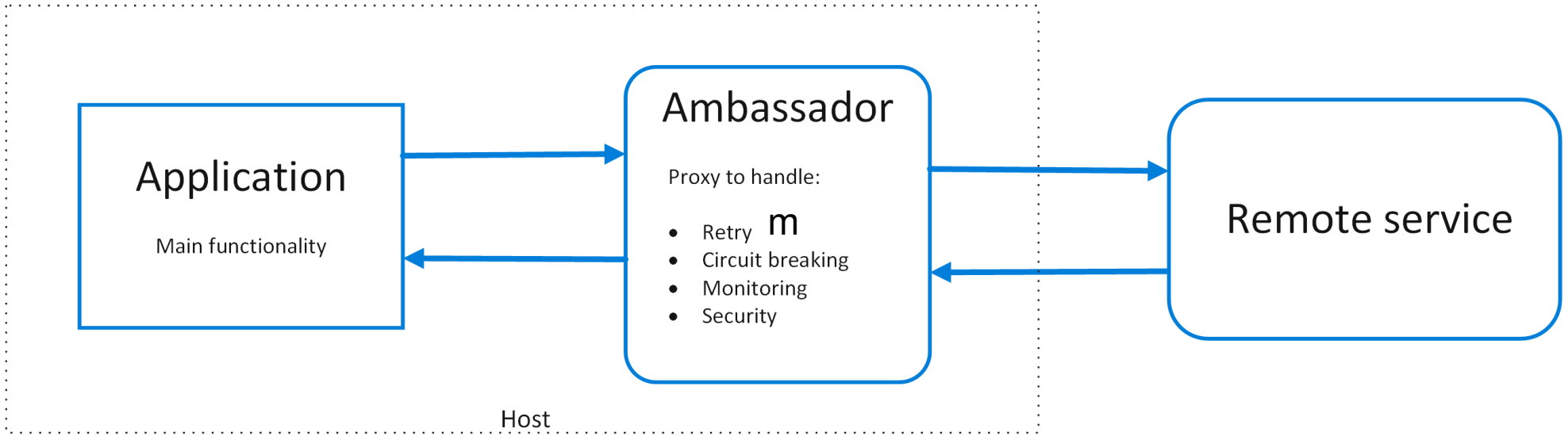


Migration complete



Ambassador

- Create helper services that send network requests on behalf of an application
- Can be thought of as an out-of-process proxy that is co-located with the client
- Handles usually resending requests on failures, monitoring and circuit breaking (disabling all future requests until application is up again from a failure)


Application
Main functionality
Ambassador
Proxy to handle:
Retry m
Circuit breaking
Monitoring
Security
Remote service
Host

Bulkhead

- Isolates critical resources, such as connection pool, memory, and CPU, for each workload or service
- A single service can’t consume all of the resources, starving others

Any questions?