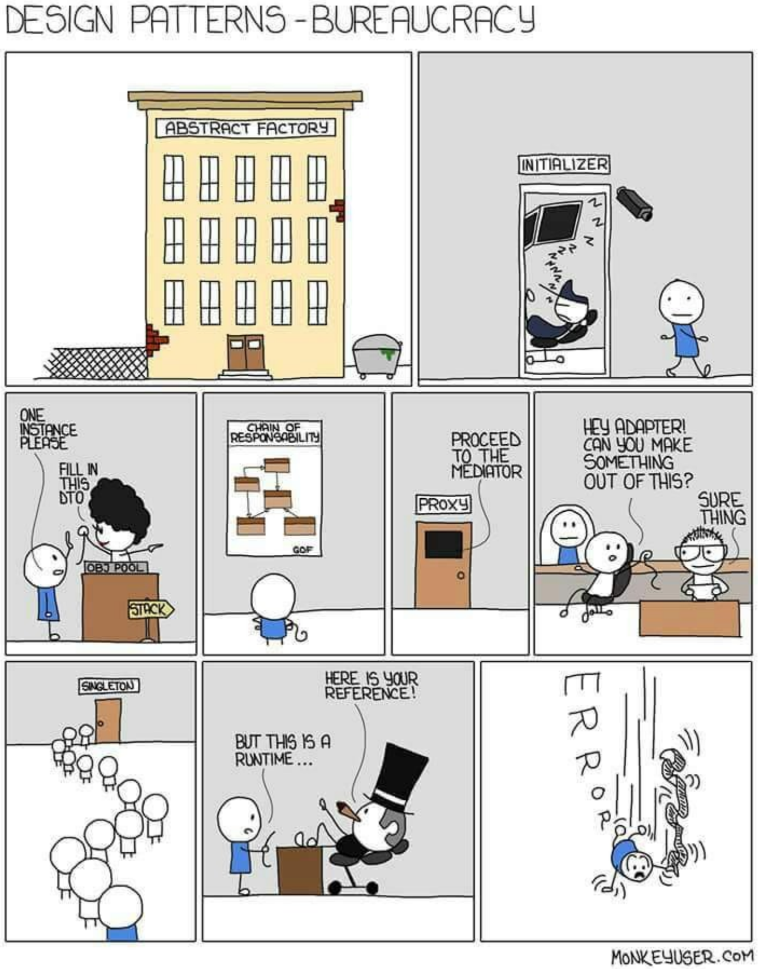
Design Patterns

Part 2

Tamlin Love

Today’s Topics

- Design Patterns II
  - Structural Patterns
    - Adapter
    - Bridge
    - Composite
    - Decorator
    - Facade
    - Flyweight
    - Proxy

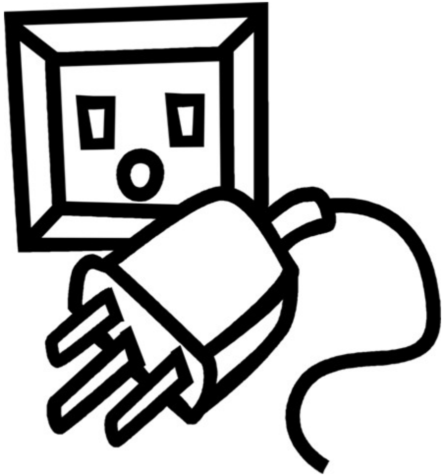


Structural Design Patterns

- **Structural design patterns** deal with organising objects to form larger structures and provide new functionality
  - They can add functionality to objects
  - They can control how objects are accessed
  - They can simplify the process of interacting with objects
  - They can optimise how objects use memory

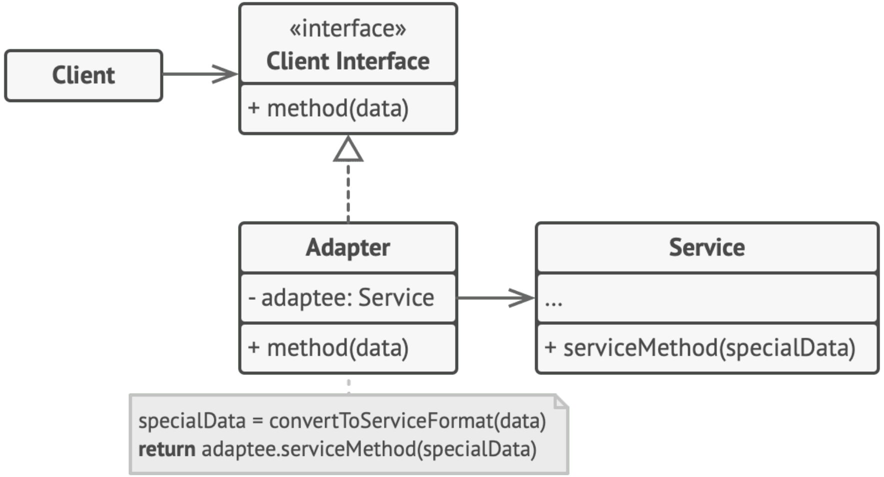
Adapter

- The **adapter pattern**, also called the **wrapper pattern**, allows the interface of a class to be used as another interface for a different class
  - With the adapter pattern, you can ensure that two classes that expect different interfaces can still communicate
- Two variants: **object adapter** and **class adapter**



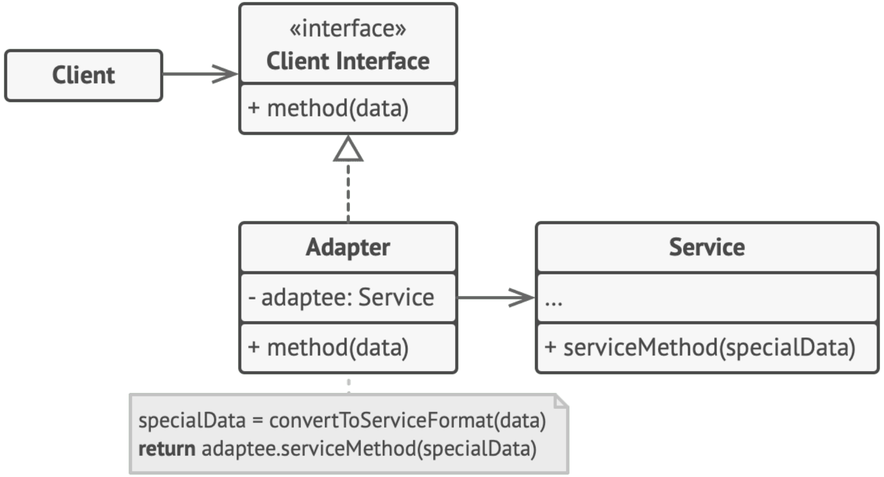
Adapter – Object Adapter

- Client interface (a.k.a. Target) – defines the specific interface the client uses
- Client - collaborates with objects conforming to the client interface
- Service (a.k.a. Adaptee) - defines an existing interface that needs adapting
- Adapter - adapts the interface of Service to the Client interface


Client
«interface»
Client Interface
+ method(data)
Adapter
- adaptee: Service
+ method(data)
Service
...
+ serviceMethod(specialData)
specialData = convertToServiceFormat(data)
return adaptee.serviceMethod(specialData)

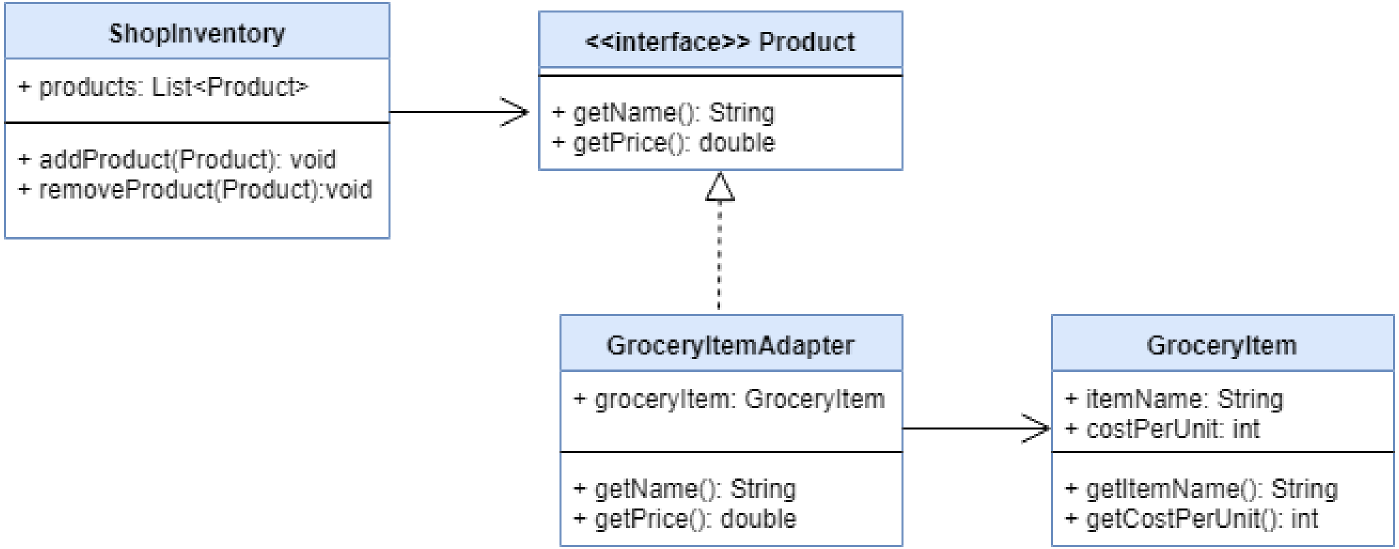
Adapter – Object Adapter

- Adapter inherits and implements the interface provided by Client interface and uses composition to use the interface provided by the Service


Client
«interface»
Client Interface
+ method(data)
Adapter
- adaptee: Service
+ method(data)
Service
...
+ serviceMethod(specialData)
specialData = convertToServiceFormat(data)
return adaptee.serviceMethod(specialData)

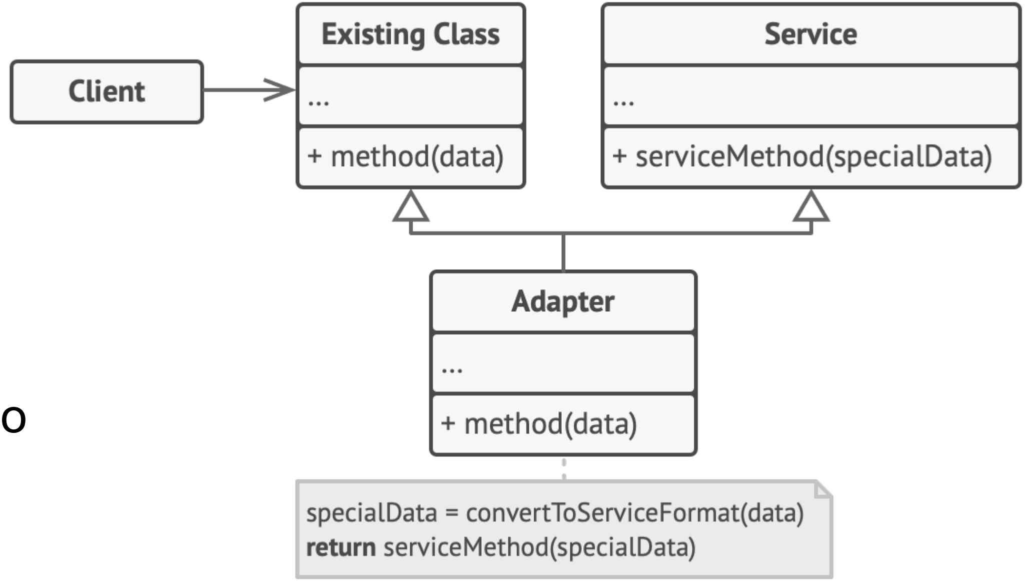
Adapter – Object Adapter

- Example: shopping inventory and grocery items
  - Note: suppose the company implementing ShopInventory is different to the grocery store that implements GroceryItem, thus GroceryItem doesn't implement Product interface



Adapter – Class Adapter

- Class Adapter – same as object adapter, but now the Adapter uses multiple inheritance to implement both interfaces simultaneously
  - Only works in languages that support multiple inheritance (C++, Python, Perl, etc.)
  - Or, you can use clever tricks to mimic: see Twin Pattern


Client
Existing Class
...
+ method(data)
Service
...
+ serviceMethod(specialData)
Adapter
...
+ method(data)
specialData = convertToServiceFormat(data)
return serviceMethod(specialData)

Adapter – Object vs Class

- Object adapter:
  - lets a single Adapter work with many Services—that is, the Service itself and all of its subclasses (if any)
  - makes it harder to override Service behavior. Requires subclassing Service and making Adapter refer to the subclass rather than the Service itself
- Class Adapter
  - Can’t work with many Services
  - lets Adapter override some of Service’s behavior, since Adapter is a subclass of Service

Adapter

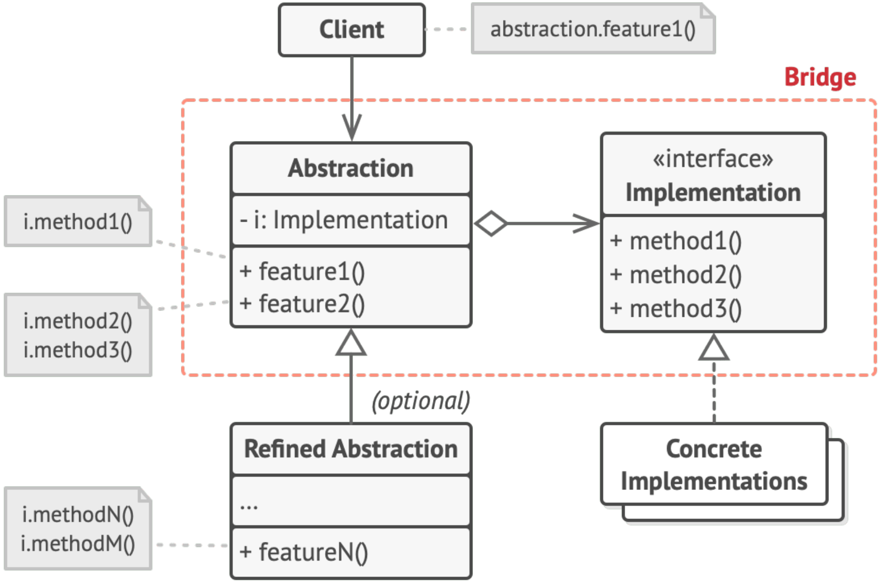
- Use adapter when:
  - you want to use an existing class, and its interface does not match the one you need
  - you want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that don't necessarily have compatible interfaces
  - you need to use several existing subclasses, but it's impractical to adapt their interface by subclassing every one
    - Object adapter only - object adapter can adapt the interface of its parent class

Adapter

- Pros:
  - You can separate the interface or data conversion code from the primary business logic of the program
    - Single responsibility principle
  - You can introduce new types of adapters into the program without breaking the existing client code, as long as they work with the adapters through the client interface
    - Open-closed principle
- Cons:
  - New interfaces and classes can increase complexity and computation
    - Sometimes simpler in the long run to refactor Service to use Client interface

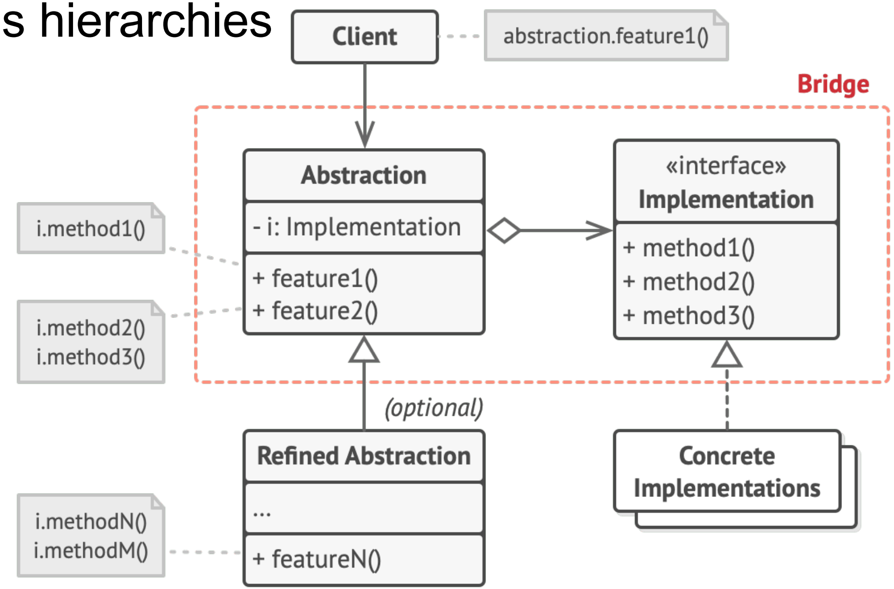
Bridge

- The **bridge pattern** facilitates the decoupling of an abstraction from its implementation, allowing the two to develop separately



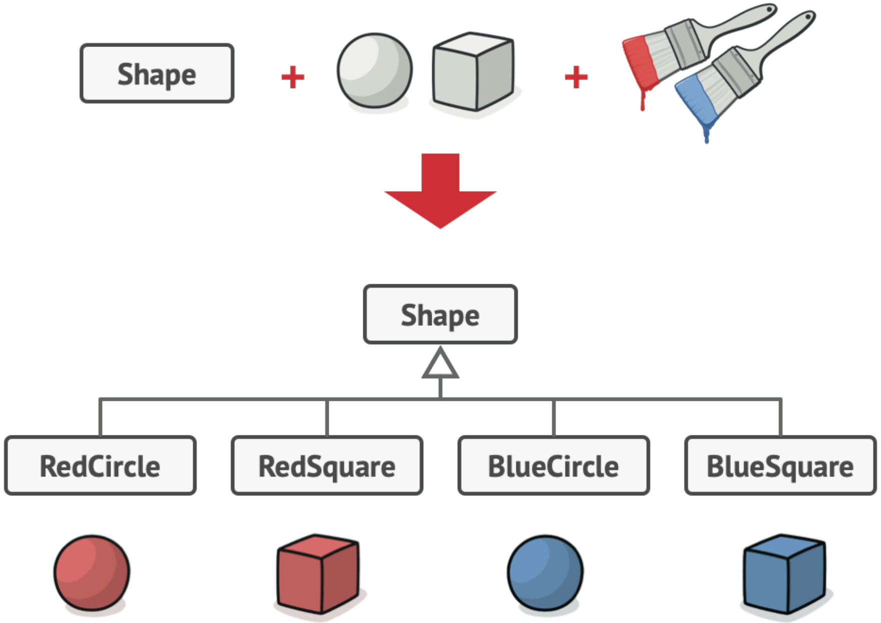
Bridge

- The **bridge pattern** facilitates the decoupling of an abstraction from its implementation, allowing the two to develop separately
  - Bridge allows you to split a large class or collection of classes into two separates hierarchies



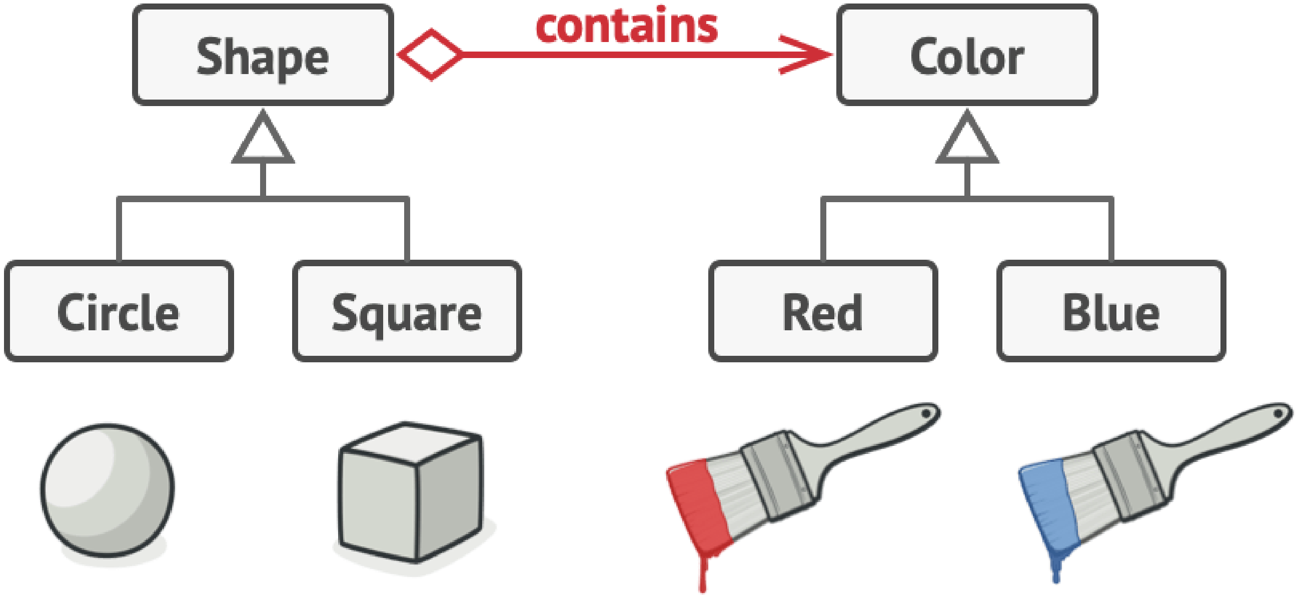
Bridge

- Motivation: adding features to an interface may force you to have multiple subclasses representing configurations of that interface



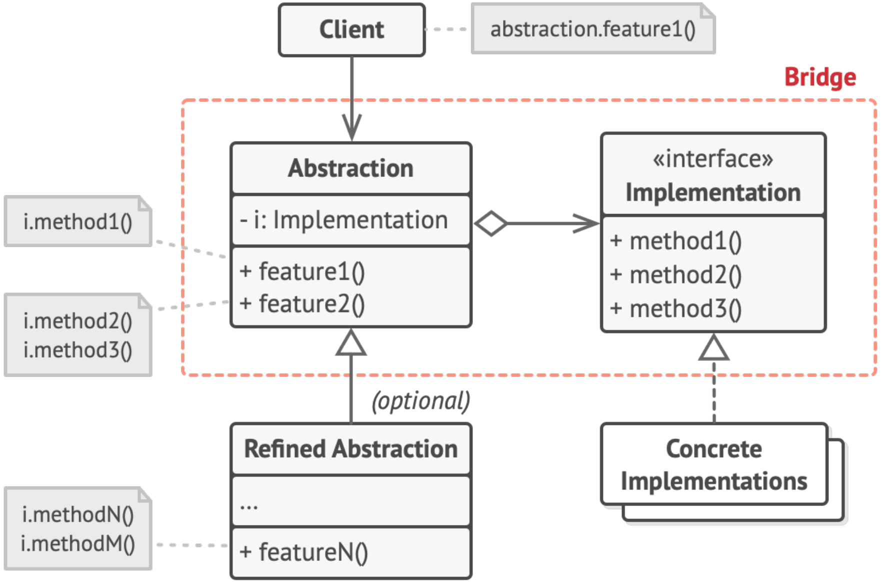
Bridge

- Motivation: with the bridge pattern, we can separate large classes into separate hierarchies of classes



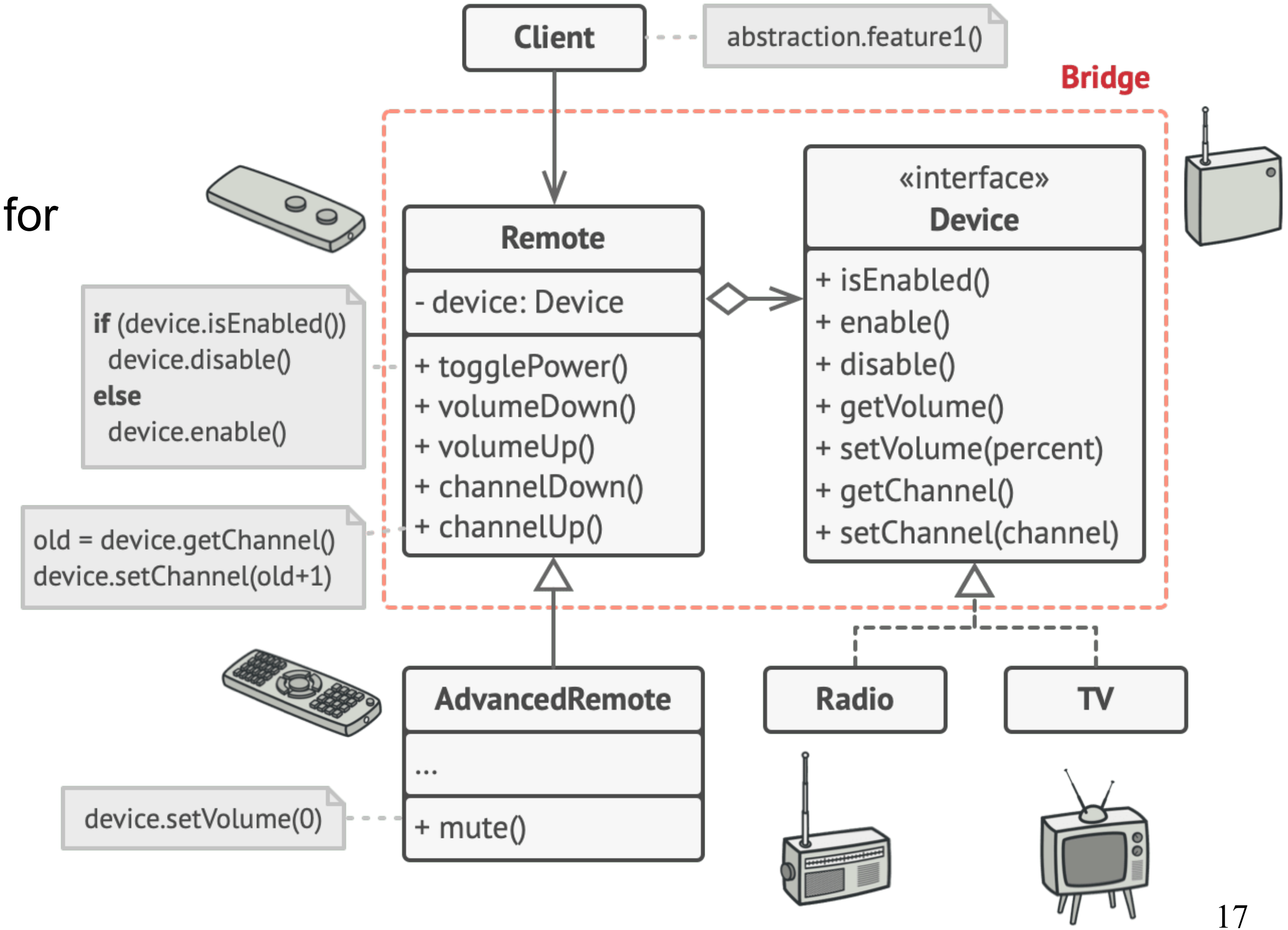
Bridge

- Abstraction – defines an interface, references the Implementor
- RefinedAbstraction – extends the Abstraction
- Implementation – declares the interface used by ConcreteImplementations
- ConcreteImplementation – implements the Implementation interface



Bridge

- Example: remotes for devices



Client

```
abstraction.feature1()
```

Remote

- device: Device

+ togglePower()
+ volumeDown()
+ volumeUp()
+ channelDown()
+ channelUp()

```
if (device.isEnabled())
  device.disable()
else
  device.enable()
```

```
old = device.getChannel()
device.setChannel(old+1)
```

«interface»
Device

+ isEnabled()
+ enable()
+ disable()
+ getVolume()
+ setVolume(percent)
+ getChannel()
+ setChannel(channel)

AdvancedRemote

...

+ mute()

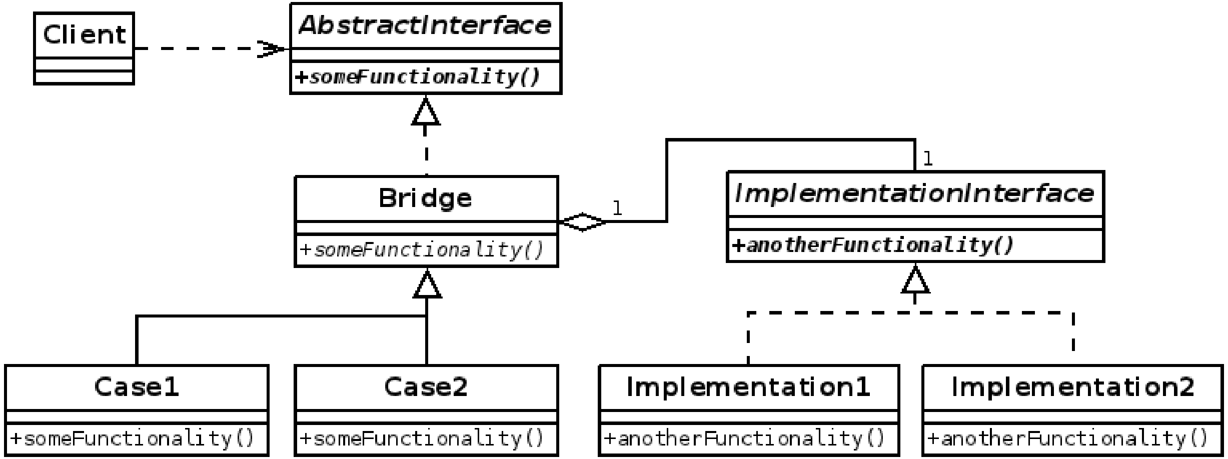
```
device.setVolume(0)
```

Radio

TV

Bridge vs Adapter

- Bridge and adapter often confused as both allow separate classes / class hierarchies to communicate
  - Adapter is often applied to existing systems to get separate components to work together
  - Bridge is often implemented up-front to allow abstractions and implementations to develop separately
- Implementations of the bridge pattern often use an object adapter



Bridge

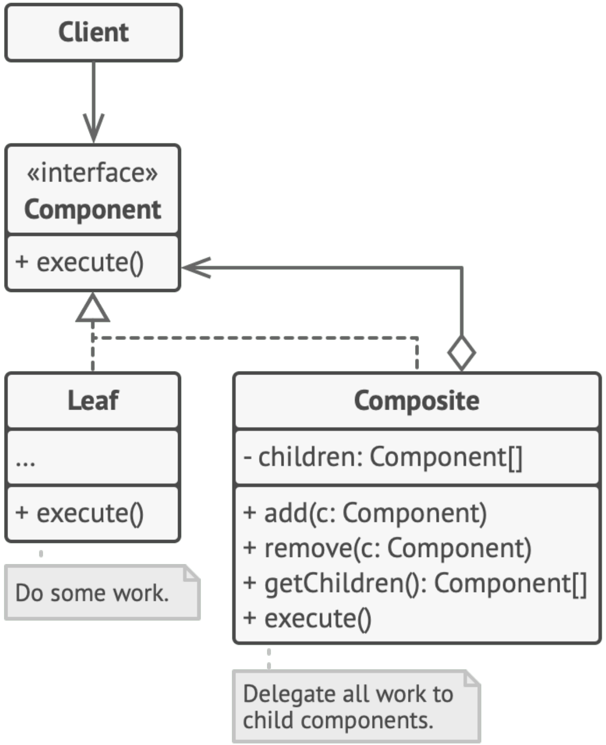
- Use bridge when:
  - you want to avoid a permanent binding between an abstraction and its implementation
    - e.g. when implementation is selected at runtime
  - both the abstractions and their implementations should be extensible by subclassing
  - rewriting an implementation of the abstraction shouldn’t force a rewrite of the client

Bridge

- Pros:
  - You can create platform-independent classes and apps
  - The client code works with high-level abstractions. It isn’t exposed to the platform details
  - You can introduce new abstractions and implementations independently from each other
    - Open-close principle
  - You can focus on high-level logic in the abstraction and on platform details in the implementation
    - Single responsibility principle
- Cons:
  - You might make the code more complicated by applying the pattern to a highly cohesive class

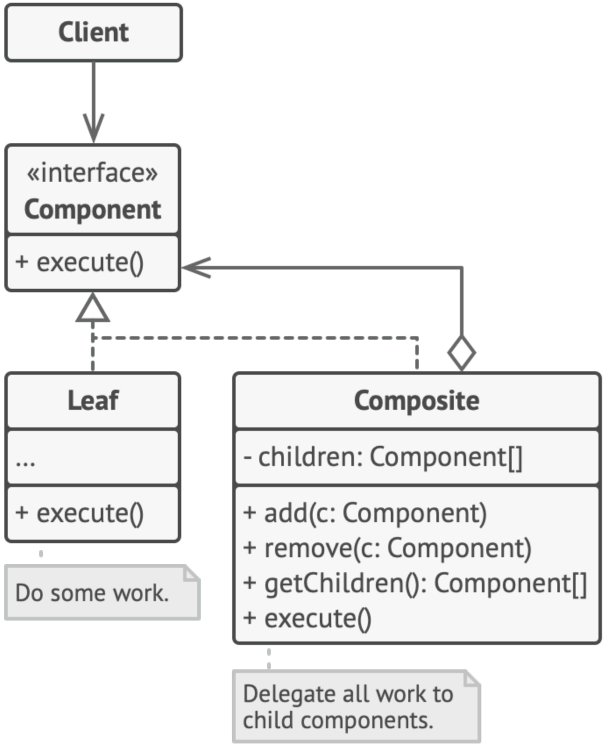
Composite

- The **composite pattern** allows for objects to be composed as tree structures, and then treat individual objects and these tree structures the same way



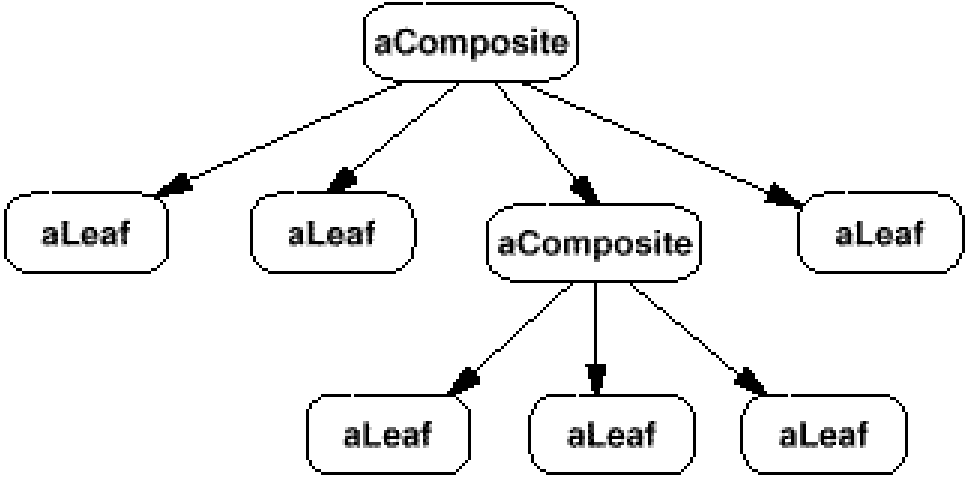
Composite

- Component – declares interface for objects in the composition, as well as methods for accessing children and (optionally) parents
- Leaf – defines primitive behaviour in the composition. Has no children.
- Composite – defines behaviour for having children, stores children and implements child-related methods



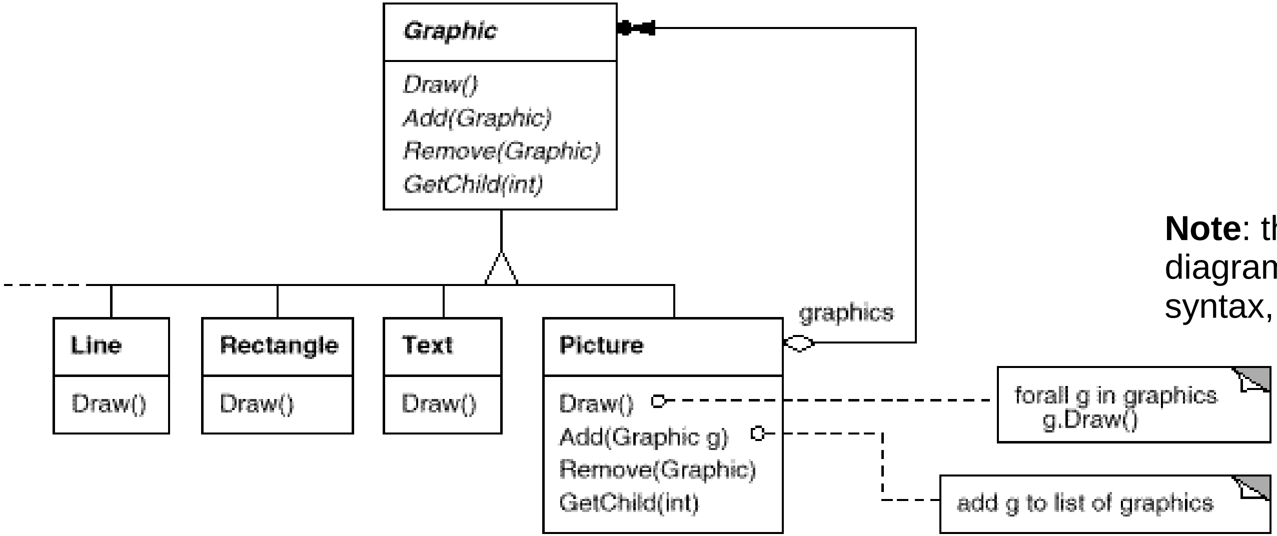
Composite

- Instantiated structure of composite pattern



Composite

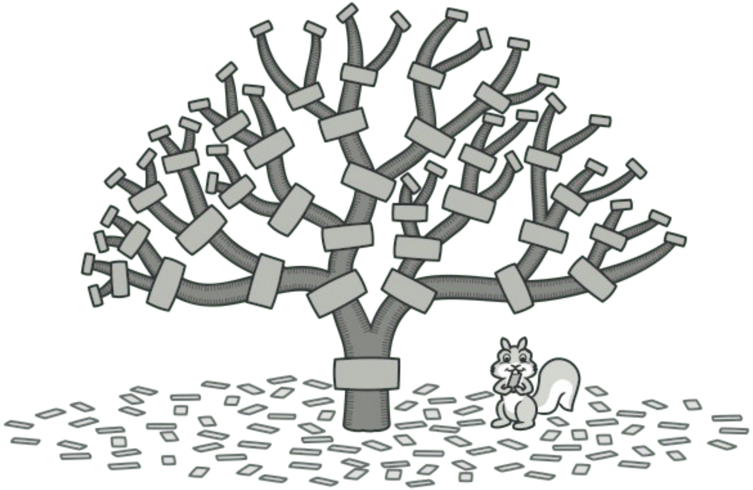
- Example: graphical application



Note: this particular diagram uses older OMT syntax, not UML

Composite

- Use composite when:
  - you want to represent hierarchies of objects in a tree structure
  - you want clients to be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly

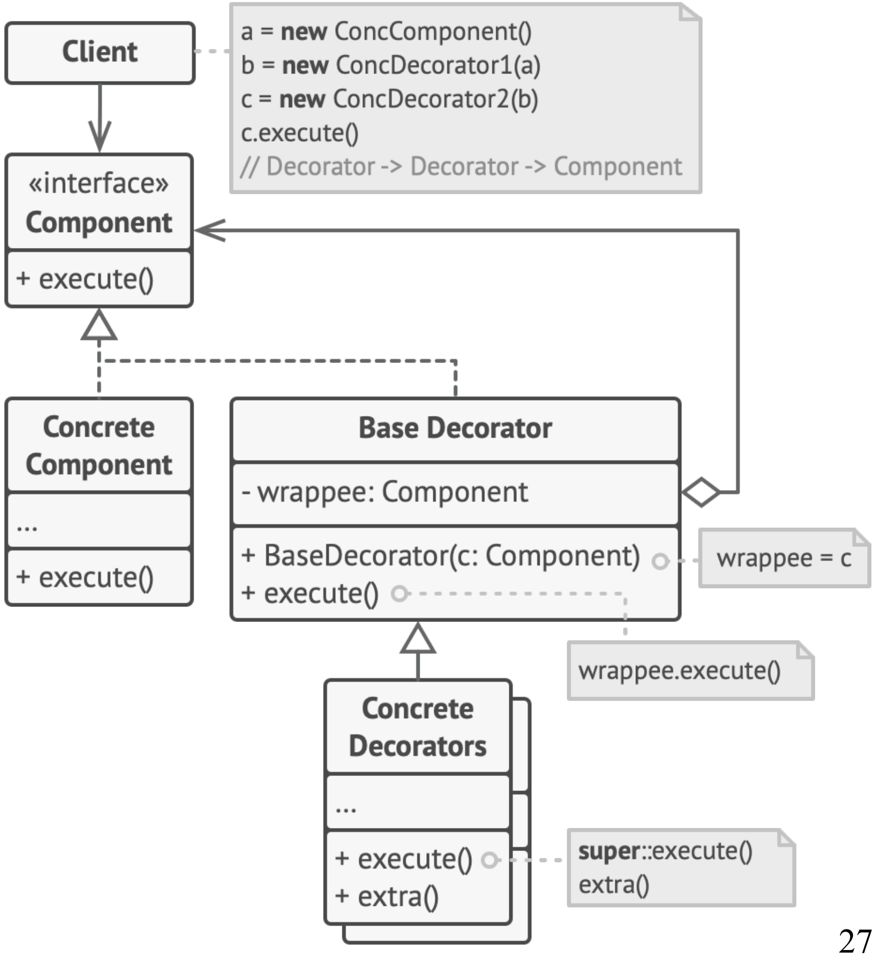


Composite

- Pros:
  - You can work with complex tree structures more conveniently: use polymorphism and recursion to your advantage
  - You can introduce new element types into the app without breaking the existing code, which now works with the object tree
    - Open-closed principle
- Cons:
  - It might be difficult to provide a common interface for classes whose functionality differs too much. In certain scenarios, you’d need to overgeneralize the component interface, making it harder to comprehend

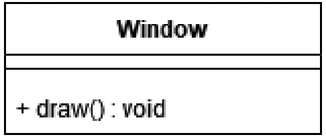
Decorator

- The **decorator pattern**, sometimes called the **wrapper pattern** (not to be confused with the adapter pattern, which shares this alternate name) allows for behaviour to be added to an individual object without affecting other objects of the same class

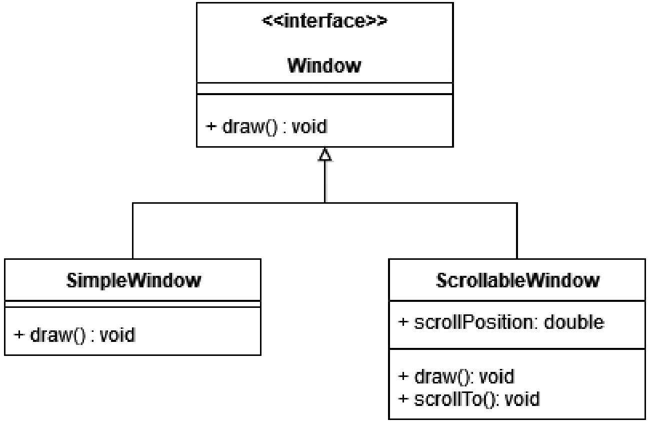


Decorator

- Motivation: suppose we have a editor that let’s us add windows to a GUI

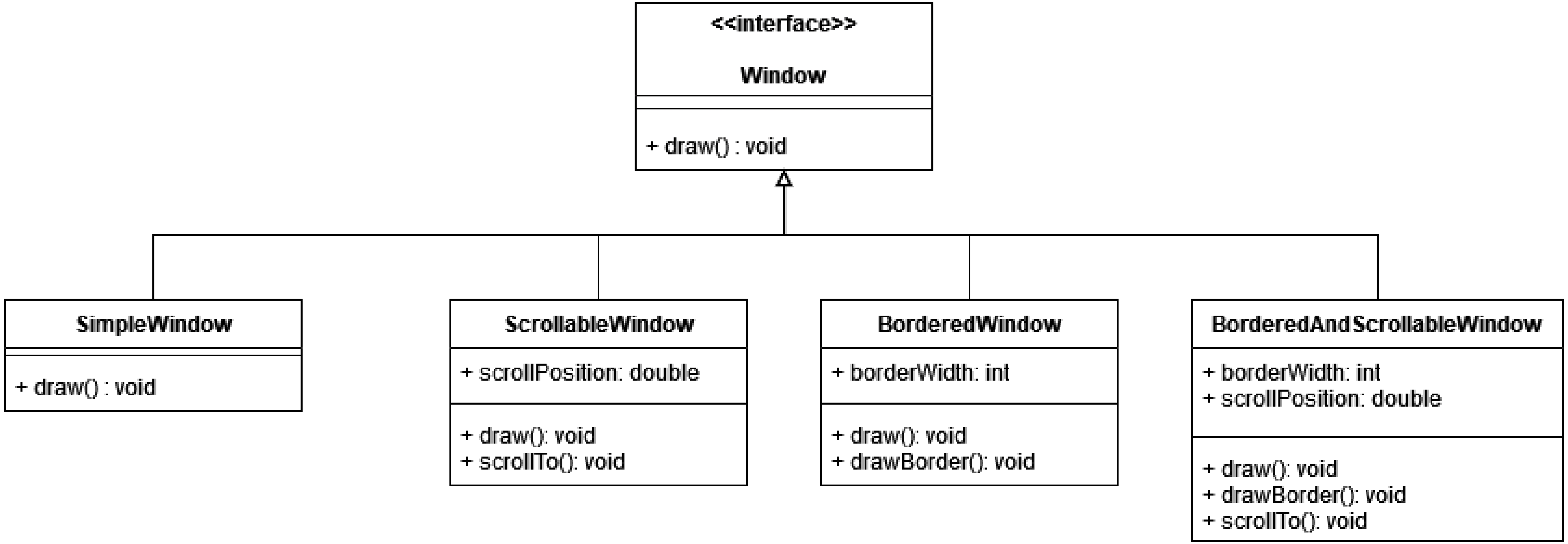


- We now use inheritance to extend functionality to add a scroll bar to the window



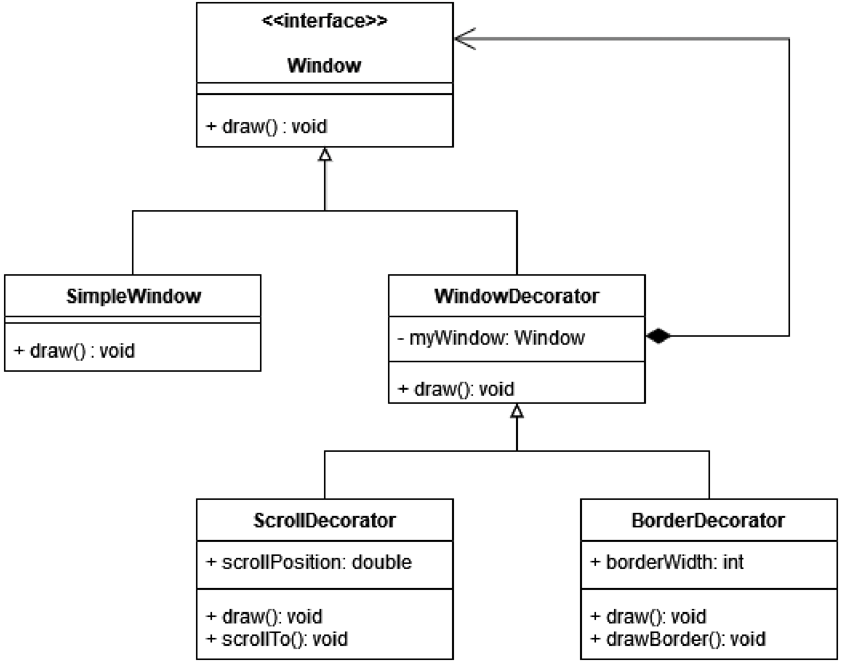
Decorator

- Motivation: suppose we now add a border functionality to a window
  - If we want a class to have a border and a scroll bar, we need an extra class



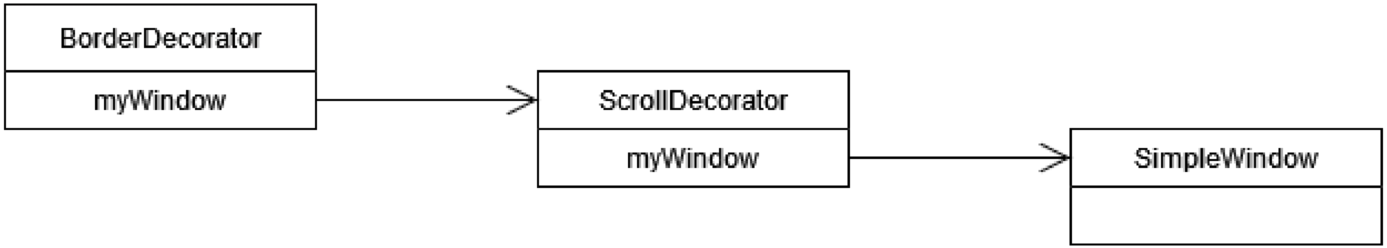
Decorator

- Motivation: if we take advantage of object composition, we can add extra functionality to individual objects using a decorator class



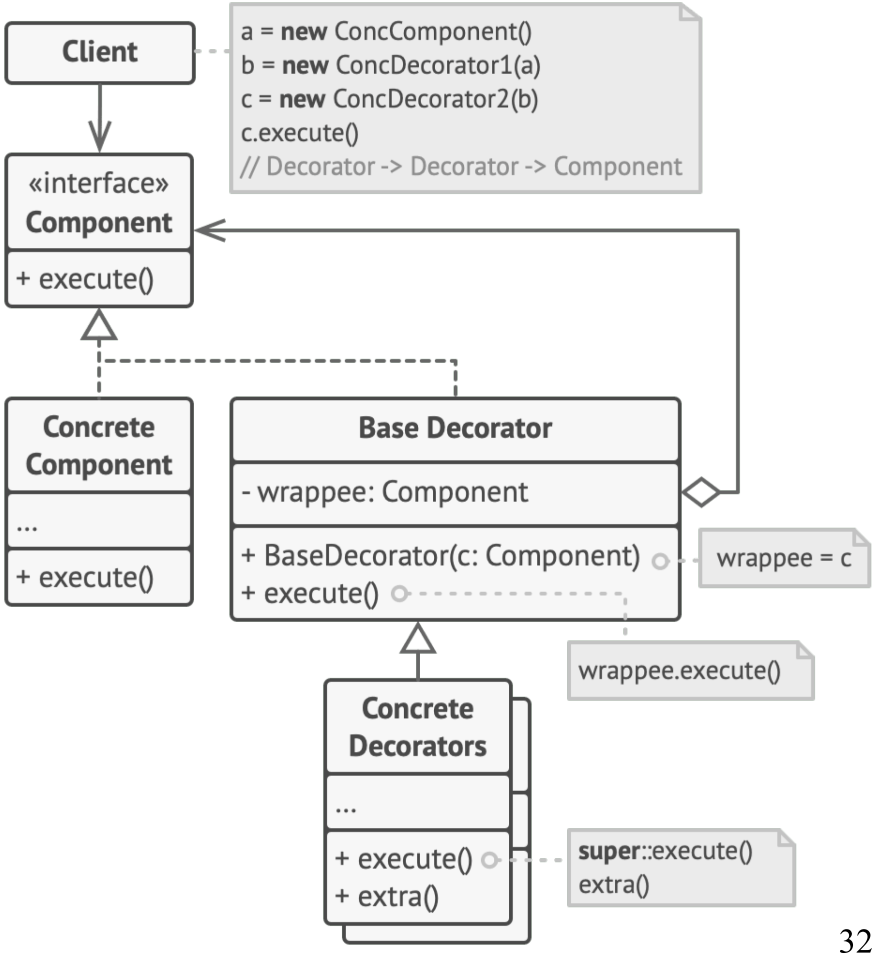
Decorator

- Motivation: view of object references



Decorator

- Component – defines interface for objects that can be decorated
- ConcreteComponent – defines an object to which additional features can be added
- Decorator – references a Component object and conforms to the Component interface
- ConcreteDecorator – adds features to the Component

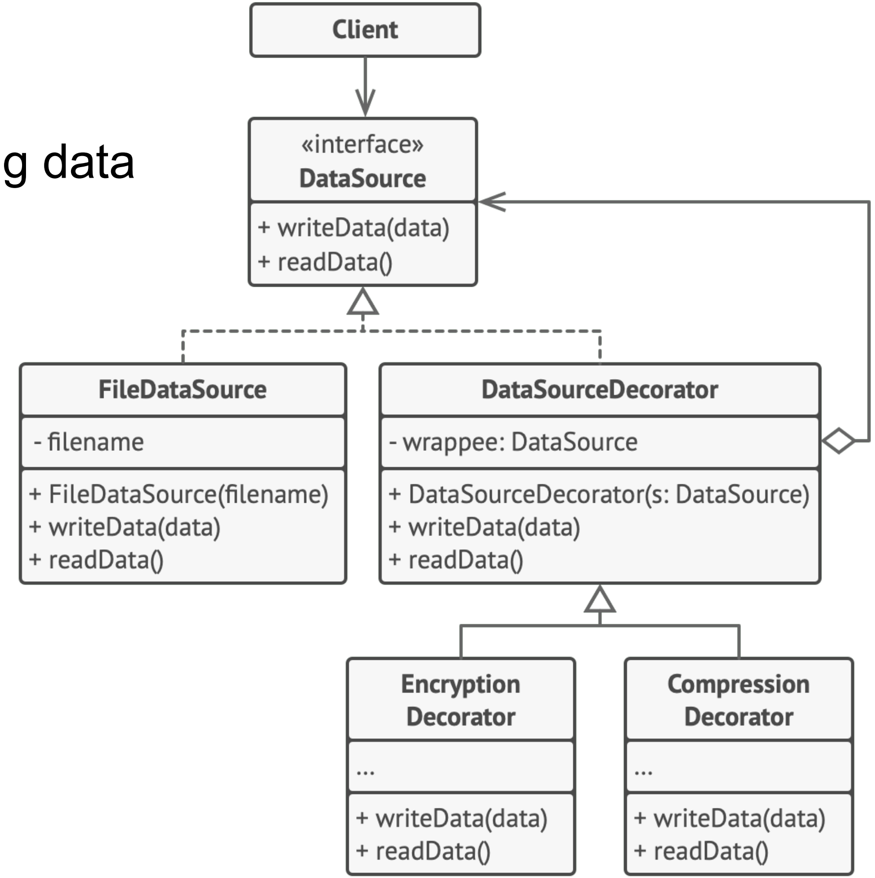


```
a = new ConcComponent()
b = new ConcDecorator1(a)
c = new ConcDecorator2(b)
c.execute()
// Decorator -> Decorator -> Component
```

```
super::execute()
extra()
```

Decorator

- Example: encrypting and compressing data



Decorator vs Adapter

- Adapter:
  - changes the interface of an existing object
- Decorator:
  - enhances an object without changing its interface
  - supports recursive composition

Decorator

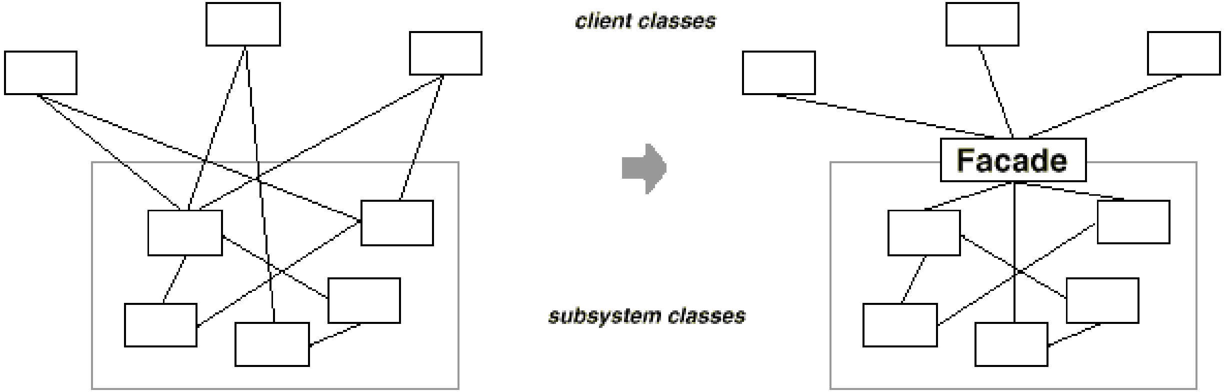
- Use decorator:
  - to add responsibilities to individual objects dynamically and transparently (i.e. without affecting other objects)
  - when extension by subclassing is impractical
    - e.g. large number of extensions results in many subclasses for each combination

Decorator

- Pros:
  - You can extend an object’s behavior without making a new subclass
  - You can add or remove responsibilities from an object at runtime
  - You can combine several behaviors by wrapping an object into multiple decorators
  - You can divide a monolithic class that implements many possible variants of behavior into several smaller classes
    - Single responsibility principle
- Cons:
  - It’s hard to remove a specific wrapper from the wrappers stack
  - It’s hard to implement a decorator in such a way that its behavior doesn’t depend on the order in the decorators stack
  - The initial configuration code of layers might look pretty ugly

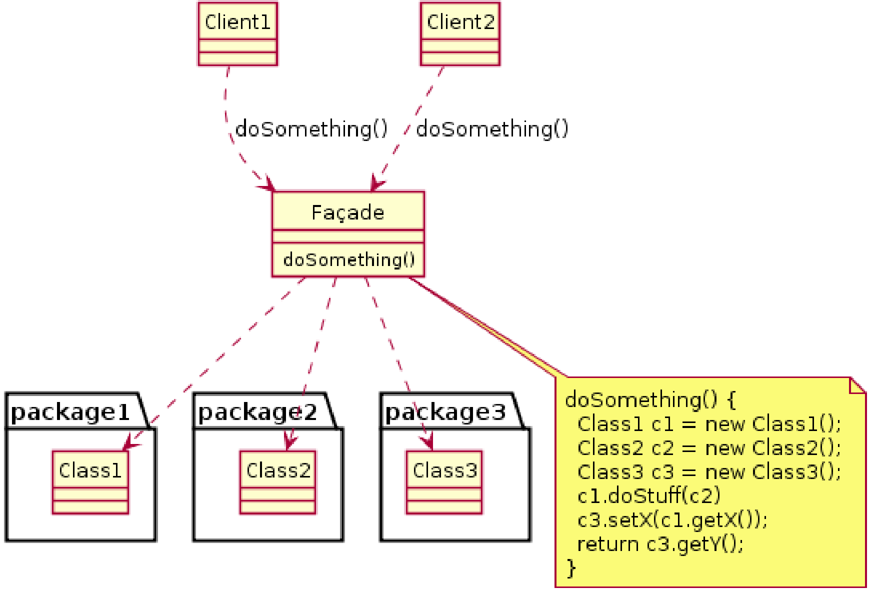
Facade

- The **facade pattern** allows an object to act as a simplified interface for a client to access more complex underlying code



Facade

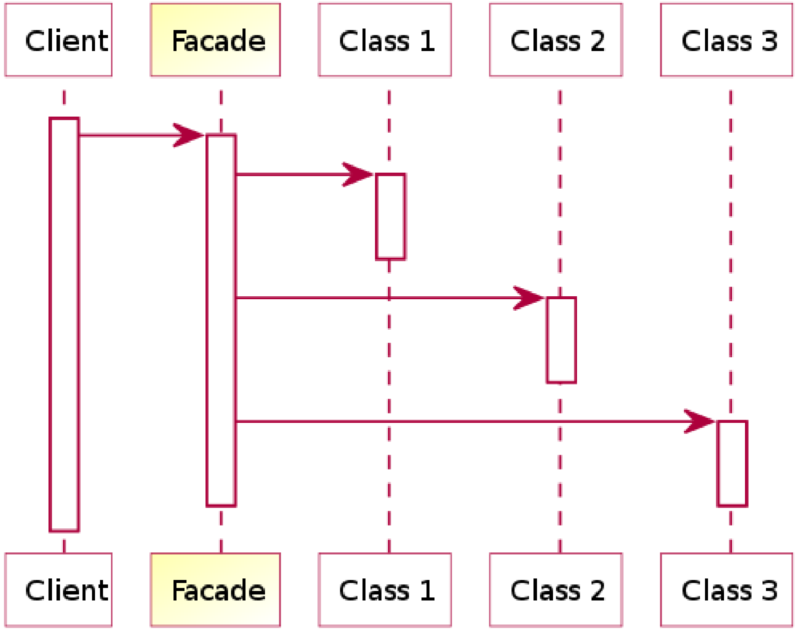
- Facade – provides an interface for and controls access to some underlying subsystem



```
doSomething() {
  Class1 c1 = new Class1();
  Class2 c2 = new Class2();
  Class3 c3 = new Class3();
  c1.doStuff(c2)
  c3.setX(c1.getX());
  return c3.getY();
}
```

Facade

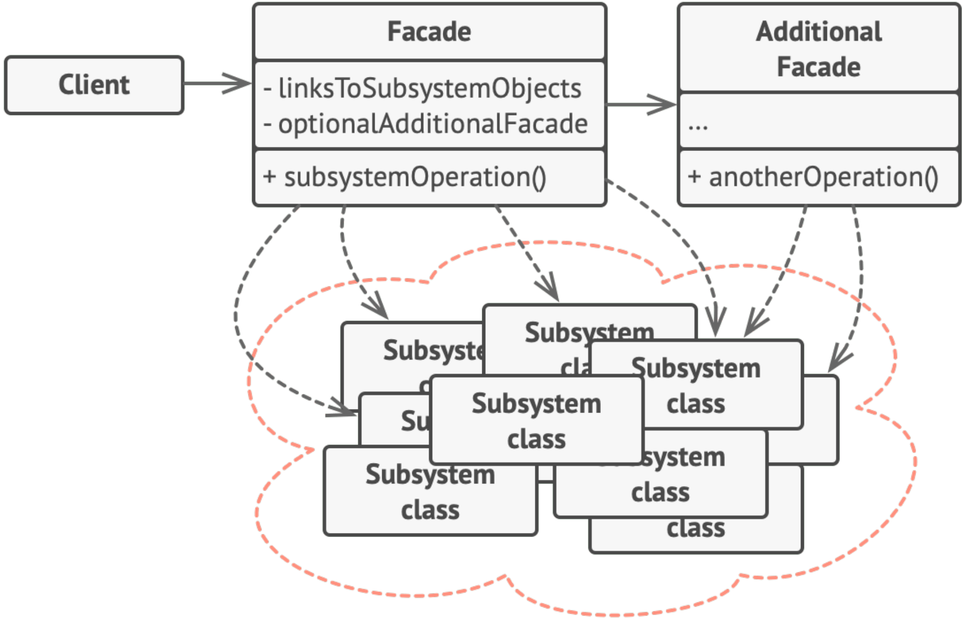
- Example sequence diagram



Sample sequence diagram

Facade

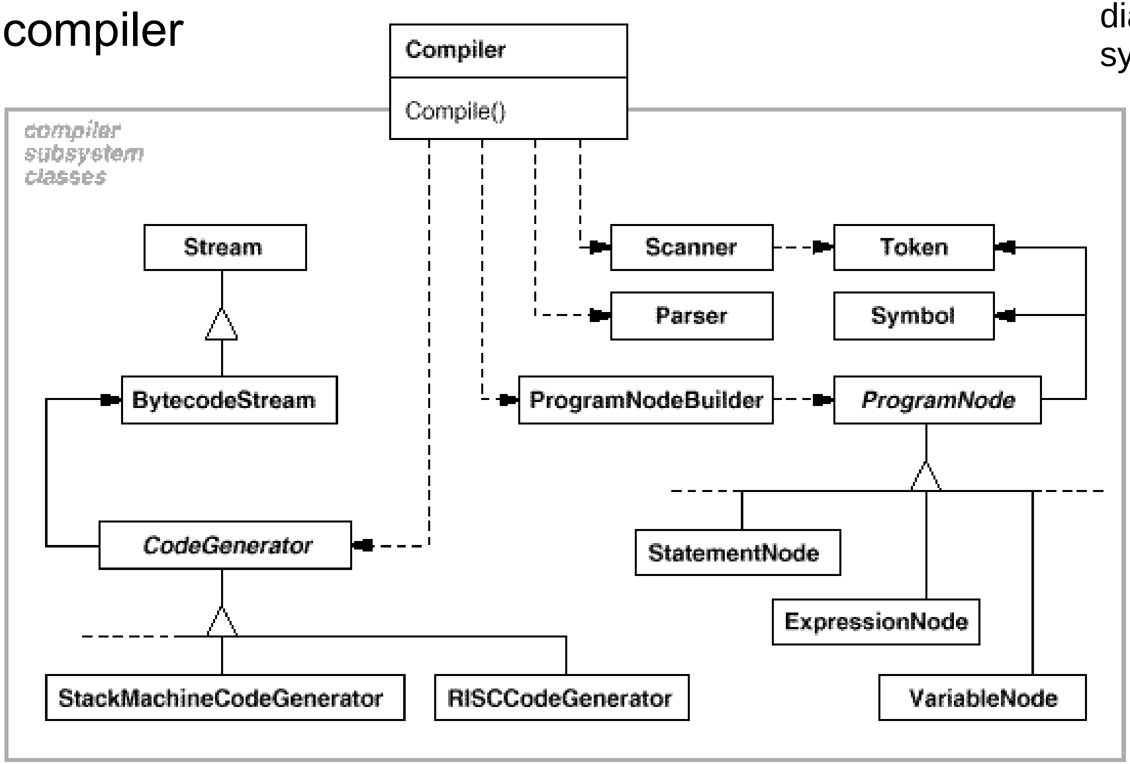
- Additional facades can prevent a single facade from becoming too complex itself
  - Can be called by clients or facades



Facade

- Example: compiler

Note: this particular diagram uses older OMT syntax, not UML



Facade

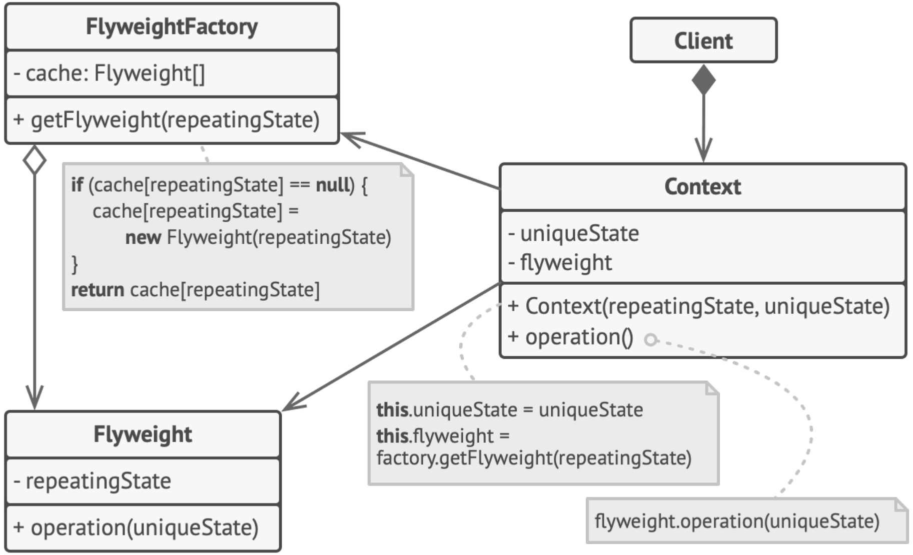
- Use facade when:
  - you want to provide a simple interface to a complex subsystem
  - there are many dependencies between clients and the implementation classes of an abstraction
    - Facades decouple clients from the subsystem
  - you want to layer your subsystems
    - Facades define an entry point to the subsystem

Facade

- Pros:
  - You can isolate your code from the complexity of a subsystem
- Cons:
  - A facade can become a god object coupled to all classes of an app
    - Can mitigate with additional facades, but too many additional facades reintroduces complexity

Flyweight

- The **flyweight pattern** allows objects to reduce their memory usage by sharing resources with other, similar objects.



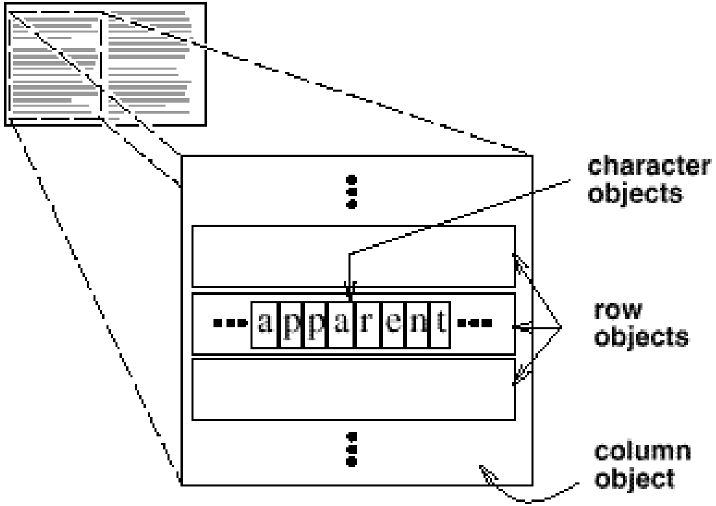
```
if (cache[repeatingState] == null) {
    cache[repeatingState] =
        new Flyweight(repeatingState)
}
return cache[repeatingState]
```

```
this.uniqueState = uniqueState
this.flyweight =
factory.getFlyweight(repeatingState)
```

```
flyweight.operation(uniqueState)
```

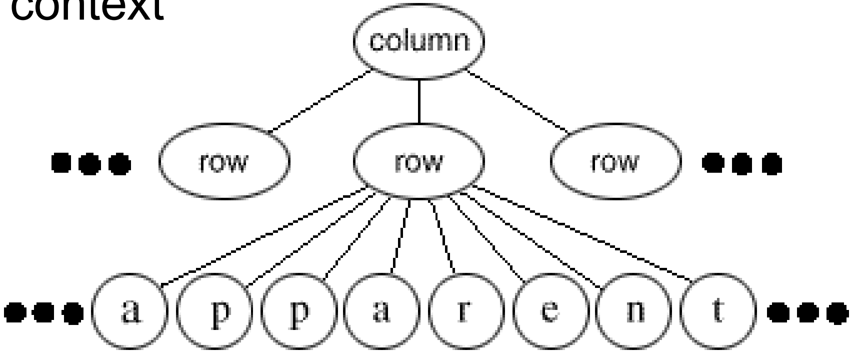
Flyweight

- Motivation: consider a word processor handling text
  - Every character, row and column is a separate object



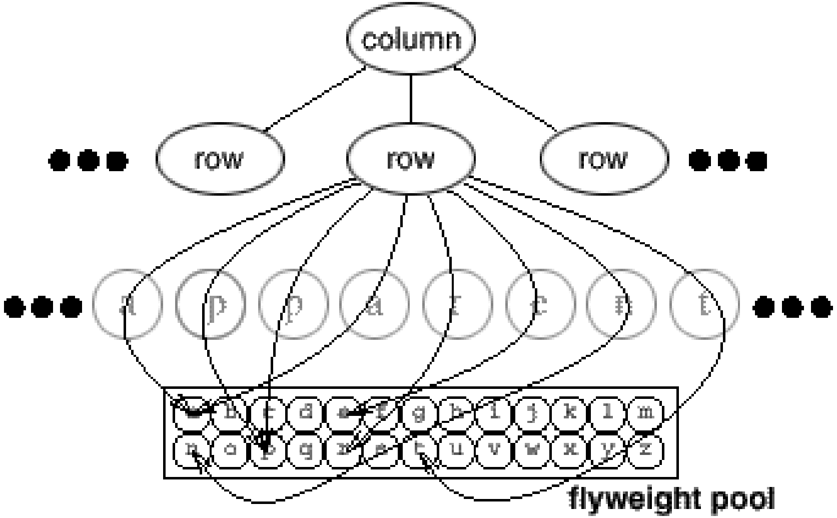
Flyweight

- Motivation: in a document with many thousands of characters, memory usage could grow very large
  - Each character object needs to store character code, font, font weight, font size, colour, position, etc.
  - Typically, a document only uses a few characters, with other information available from context



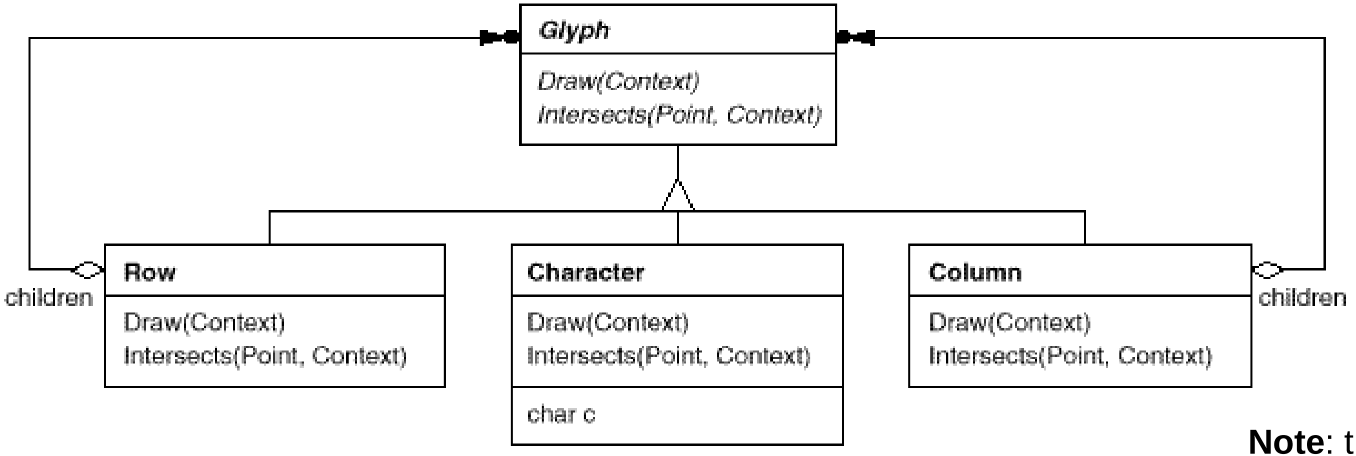
Flyweight

- Motivation: instead, maintain pools of shared objects (called **flyweights**) that store character code, and pass other information (like position, font, colour, etc.) as a context whenever the character is drawn



Flyweight

- Motivation: client is responsible for storing and passing context, flyweight (Character) is responsible only for operations relating to a particular character



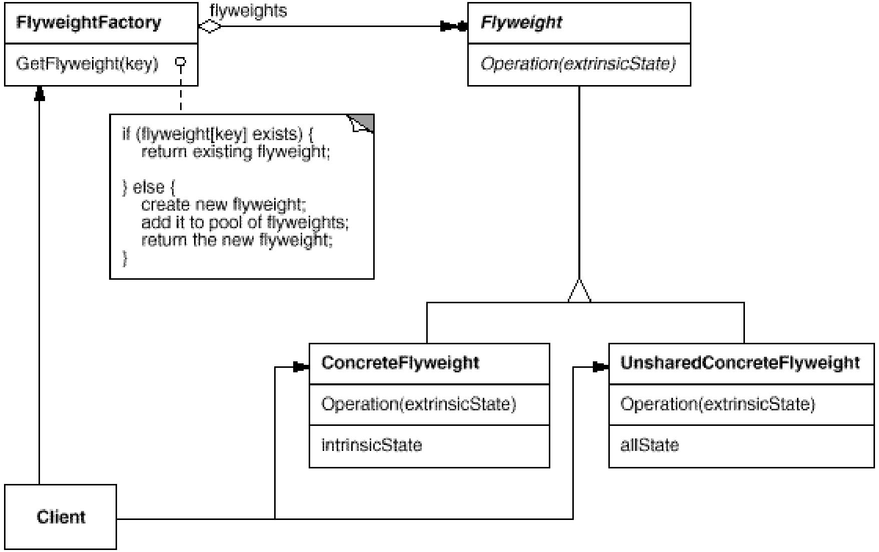
Note: this particular diagram uses older OMT syntax, not UML

Flyweight

- Some useful definitions:
  - A **flyweight** is a shared object that can be used in multiple contexts simultaneously
  - An **intrinsic state** is a state consisting of information that is independent of a flyweight’s context. It is shared and typically immutable.
    - e.g. character code
  - An **extrinsic state** is a state that depends on and varies with a flyweight’s context. It is not shared and is typically mutable.
    - e.g. position, font, font size, colour, etc.

Flyweight

- Flyweight - declares an interface through which flyweights can receive and act on extrinsic state
  - e.g. Glyph
- ConcreteFlyweight - implements the Flyweight interface and adds storage for intrinsic state, if any
  - must be sharable
  - Any state it stores must be intrinsic
    - I.e. independent of the ConcreteFlyweight object's context
  - e.g. Character

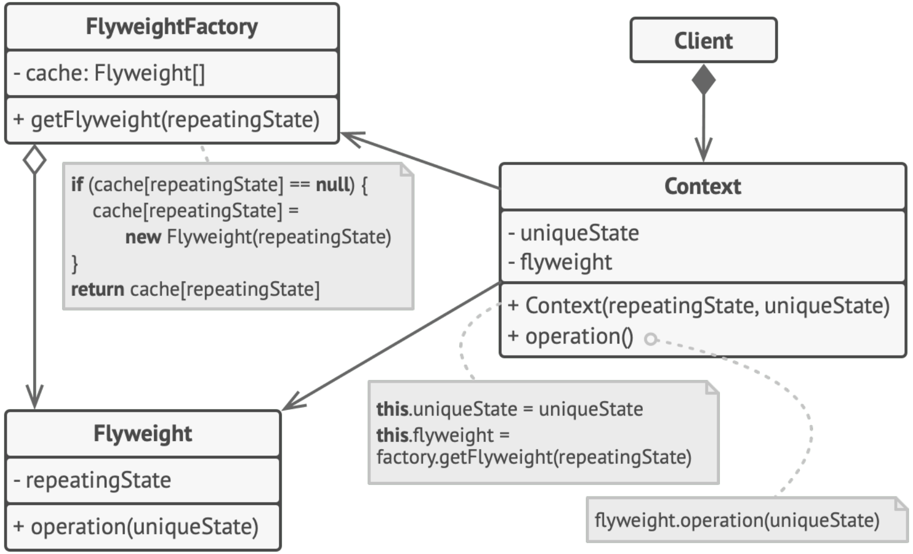

FlyweightFactory
GetFlyweight(key)
flyweights
Flyweight
Operation(extrinsicState)
ConcreteFlyweight
Operation(extrinsicState)
intrinsicState
UnsharedConcreteFlyweight
Operation(extrinsicState)
allState
Client

```
if (flyweight[key] exists) {
    return existing flyweight;

} else {
    create new flyweight;
    add it to pool of flyweights;
    return the new flyweight;
}
```

Flyweight

- FlyweightFactory - creates and manages flyweight objects. Also ensures flyweights are shared properly
- Context – maintains a reference to flyweights and computes and stores extrinsic state



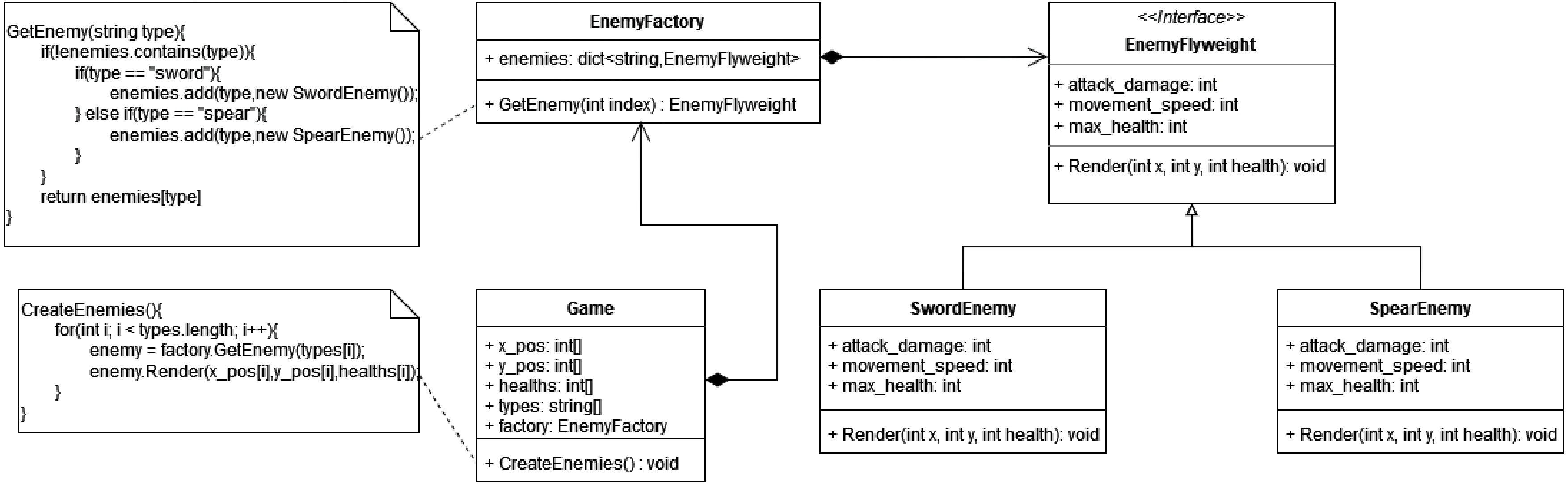
```
if (cache[repeatingState] == null) {
    cache[repeatingState] =
        new Flyweight(repeatingState)
}
return cache[repeatingState]
```

```
this.uniqueState = uniqueState
this.flyweight =
factory.getFlyweight(repeatingState)
```

```
flyweight.operation(uniqueState)
```

Flyweight

- Example: video game enemies
  - Intrinsic – attack_damage, movement_speed, max_health
  - Extrinsic – x_pos, y_pos, health



```
GetEnemy(string type){
    if(!enemies.contains(type)){
        if(type == "sword"){
            enemies.add(type,new SwordEnemy());
        } else if(type == "spear"){
            enemies.add(type,new SpearEnemy());
        }
    }
    return enemies[type]
}
```

```
CreateEnemies(){
    for(int i; i < types.length; i++){
        enemy = factory.GetEnemy(types[i]);
        enemy.Render(x_pos[i],y_pos[i],healths[i]);
    }
}
```

EnemyFactory
+ enemies: dict<string,EnemyFlyweight>
+ GetEnemy(int index) : EnemyFlyweight

<<Interface>>
EnemyFlyweight
+ attack_damage: int
+ movement_speed: int
+ max_health: int
+ Render(int x, int y, int health): void

Game
+ x_pos: int[]
+ y_pos: int[]
+ healths: int[]
+ types: string[]
+ factory: EnemyFactory
+ CreateEnemies() : void

SwordEnemy
+ attack_damage: int
+ movement_speed: int
+ max_health: int
+ Render(int x, int y, int health): void

SpearEnemy
+ attack_damage: int
+ movement_speed: int
+ max_health: int
+ Render(int x, int y, int health): void

Flyweight

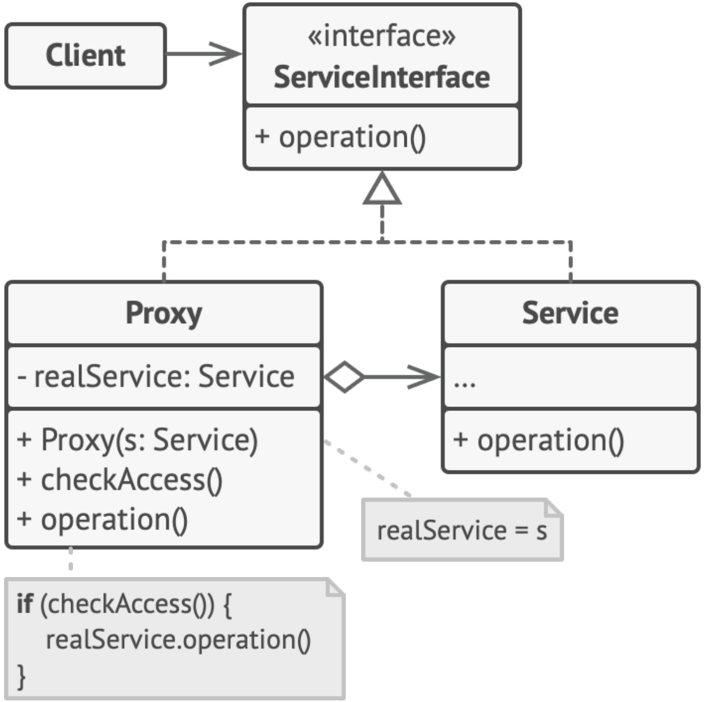
- Use flyweight when:
  - an application uses a large number of objects, **and**
  - memory costs are high because of number of objects, **and**
  - most object state can be made extrinsic, **and**
  - many groups of objects may be replaced by relatively few shared objects once extrinsic state is removed

Flyweight

- Pros:
  - You can save lots of RAM, assuming your program has tons of similar objects
- Cons:
  - You might be trading RAM over CPU cycles when some of the context data needs to be recalculated each time somebody calls a flyweight method
  - The code becomes much more complicated

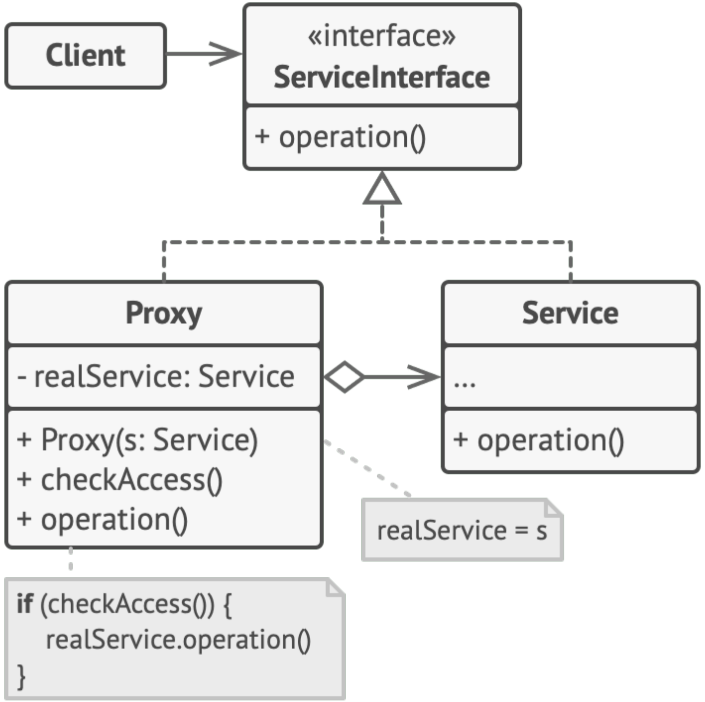
Proxy

- The **proxy pattern**, also called the **surrogate pattern**, allows one to control access to an object via a **proxy object**.
  - The proxy object can perform pre- or postprocessing
  - Often used when creating/initialising an object is expensive



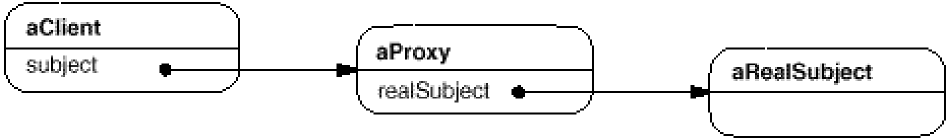
Proxy

- Service – the object we wish to control access to
- Proxy – controls access to Service. May be responsible for creating or destroying Service, or some other processing
- ServiceInterface – an interface for both Service and Proxy, so that they have an identical interface



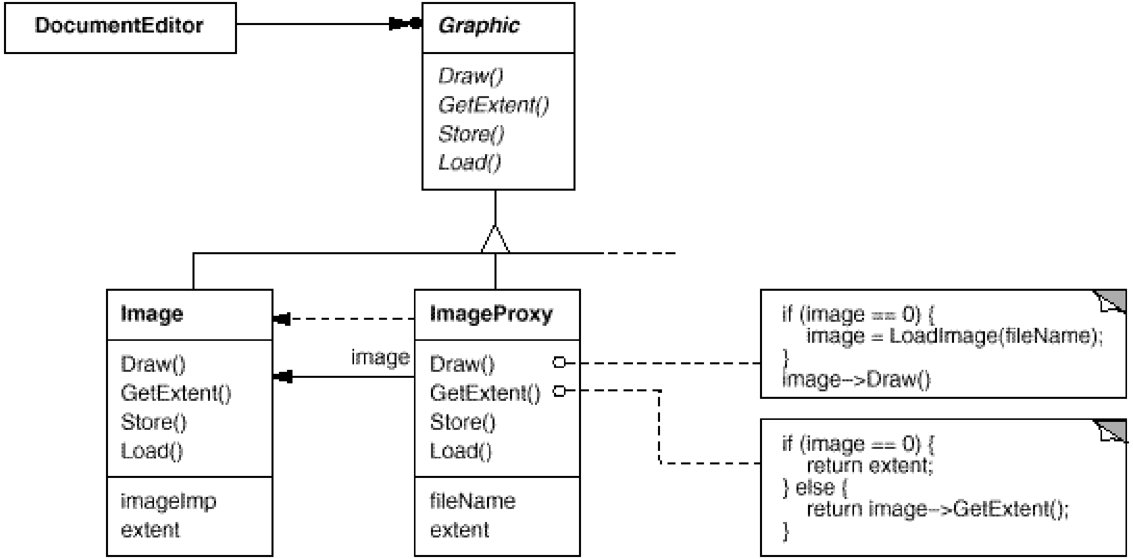
Proxy

- Example object references



Proxy

- Example: image proxy in document editor
  - Proxy stores image extent (width and height) to account for its placement in the document without actually loading the image



```
if (image == 0) {
    image = LoadImage(fileName);
}
image->Draw()
```

```
if (image == 0) {
    return extent;
} else {
    return image->GetExtent();
}
```

Proxy

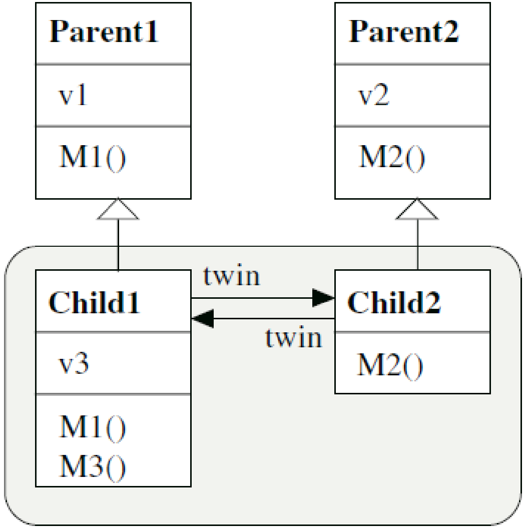
- Typical uses for proxy pattern
  - **Remote proxy** – local proxy represents a remote object
    - e.g. an ATM may have proxy objects for objects in a remote bank server
  - **Virtual proxy** – proxy creates expensive objects on demand
    - e.g. proxy for expensive elements (e.g. images) when loading a document
  - **Protection proxy** – controls access to an object based on access rights
  - **Smart proxy** – performs additional processing when accessing an object
    - e.g. count number of object references so object can be freed when there are no more references (**smart pointer**)
    - e.g. loading a persistent object into memory upon first reference
    - e.g. checking an object is locked before it is accessed so other objects can’t change it

Proxy

- Pros:
  - You can control the real object without clients knowing about it
  - You can manage the lifecycle of the service object when clients don’t care about it
  - The proxy works even if the service object isn’t ready or is not available
  - You can introduce new proxies without changing the service or clients
    - Open-closed principle
- Cons:
  - The code may become more complicated since you need to introduce a lot of new classes
  - The response from the service might get delayed

Other Structural Patterns

- Not in the original GoF list:
  - **Front Controller** – a controller handles all requests directed to a website
  - **Marker Interface** – an interface is used to attach metadata to classes
  - **Twin Pattern** – mimics multiple inheritance by maintaining two closely coupled classes that each inherit from a different class


Parent1
v1
M1()
Parent2
v2
M2()
Child1
v3
M1()
M3()
twin
twin
Child2
M2()