Wrappers

Adapter vs Proxy vs Facade vs Decorator vs … 😱

Adapter

Adapter

- allows two classes with incompatible interfaces to work together

Adapter

- allows two classes with incompatible interfaces to work together
    - converts one interface to another

Adapter

- allows two classes with incompatible interfaces to work together
  - converts one interface to another
  - but to NOT change/add/remove behaviour

Adapter

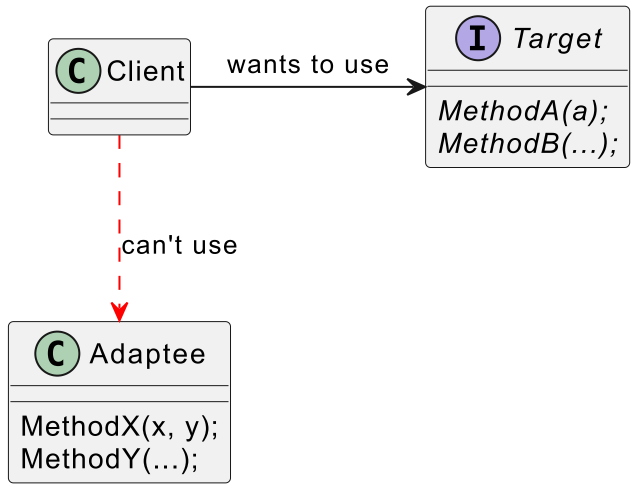
- allows two classes with incompatible interfaces to work together
  - converts one interface to another
  - but to NOT change/add/remove behaviour
  - and (rather) no logic

Adapter

- allows two classes with incompatible interfaces to work together
  - converts one interface to another
  - but to NOT change/add/remove behaviour
  - and (rather) no logic

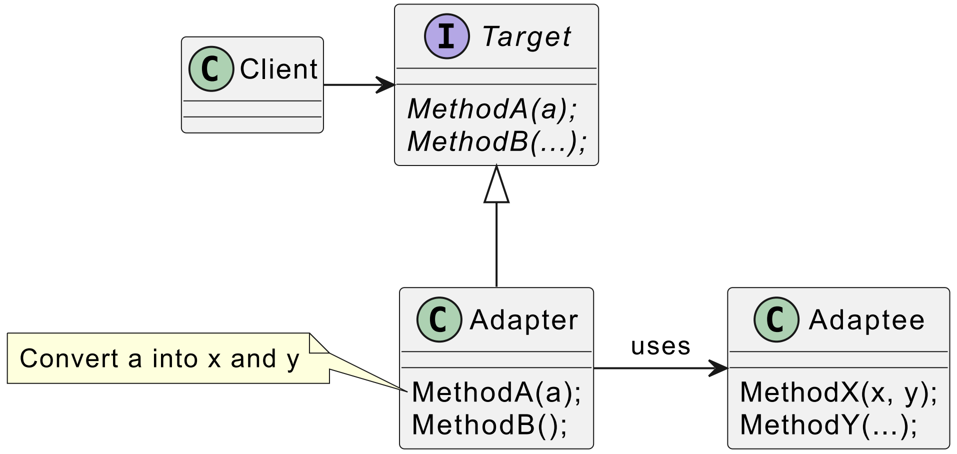
Adapter

- an adapter is not a creator of new objects (like a *Mapper* pattern) but an interface translator



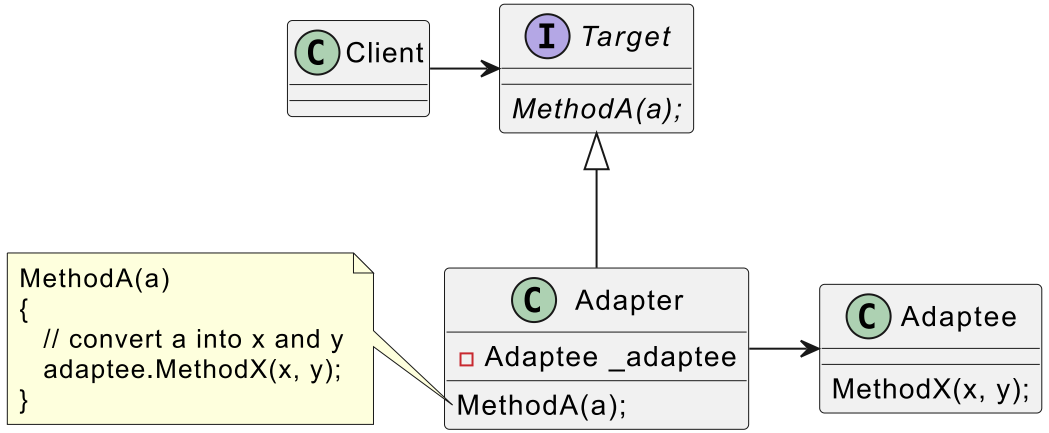
Adapter

- an adapter is not a creator of new objects (like a *Mapper* pattern) but an interface translator



Adapter - implementation #1

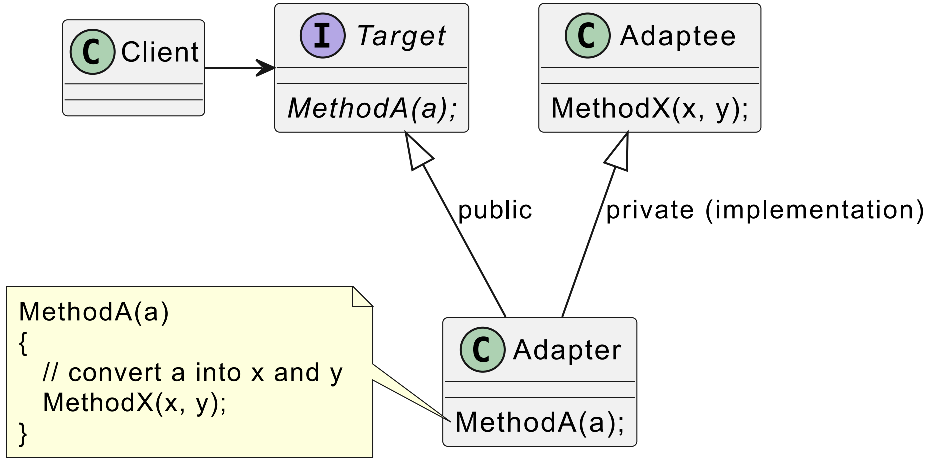
- *Adapter* **has** reference to an *Adaptee* and delegates to its methods:



```
MethodA(a)
{
   // convert a into x and y
   adaptee.MethodX(x, y);
}
```

Adapter - implementation #2

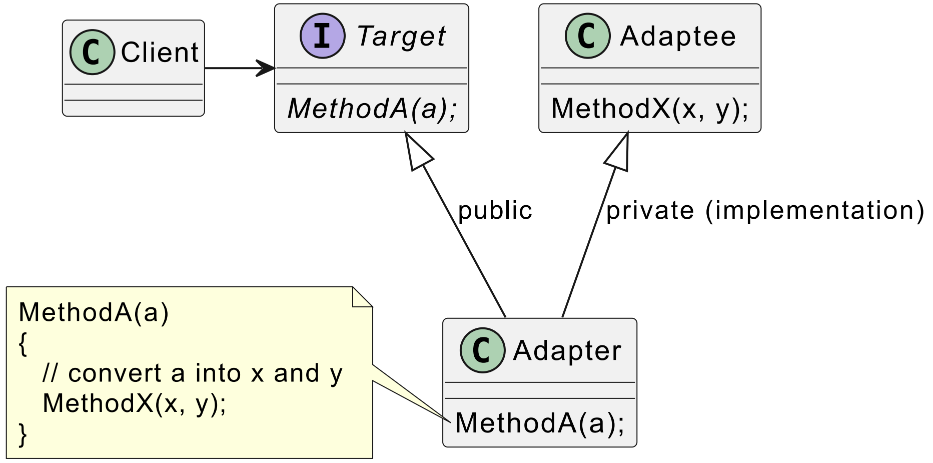
- using multiple inheritance by implementing both target interface and deriving from adaptee - `Adapter` becomes a *subtype* of `Target` but not of `Adaptee`(😱)



```
MethodA(a)
{
   // convert a into x and y
   MethodX(x, y);
}
```

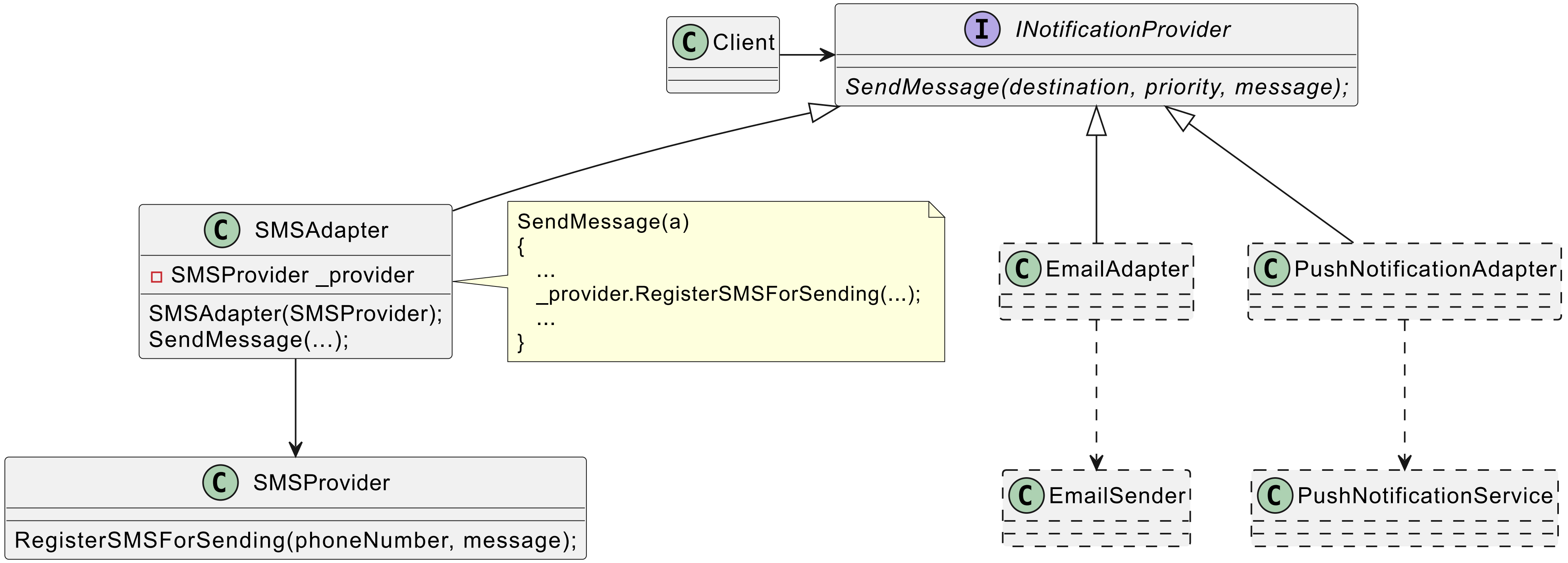
Adapter - implementation #2

- using multiple inheritance by implementing both target interface and deriving from adaptee - **`Adapter`** becomes a *subtype* of **`Target`** but not of **`Adaptee`**(😱)



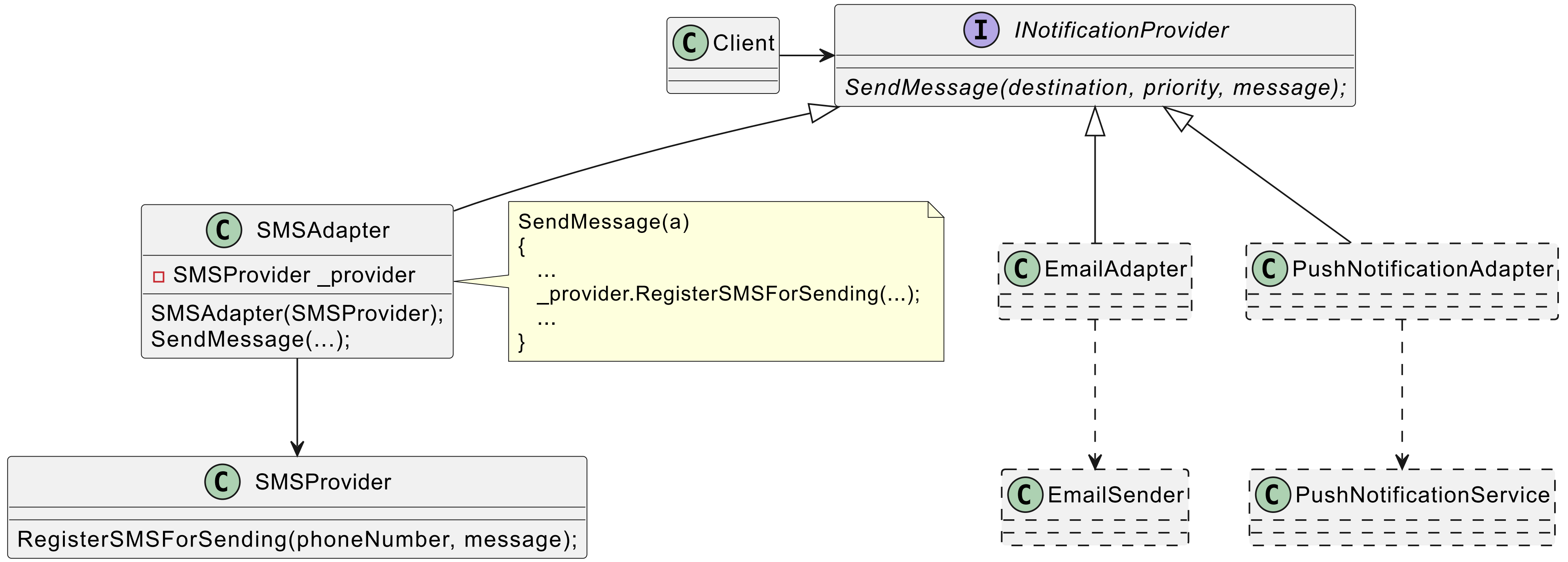
```
class Adapter : public Target, public Adaptee {
 public:
  Adapter() {}
  std::string Request() const override {
    std::string to_reverse = /*Adaptee.*/ SpecificRequest();
    std::reverse(to_reverse.begin(), to_reverse.end());
    return "Adapter: (TRANSLATED) " + to_reverse;
  }
};
```

Adapter - example #1



```
SendMessage(a)
{
   ...
   _provider.RegisterSMSForSending(...);
   ...
}
```

Adapter - example #1



```
SendMessage(a)
{
  ...
  _provider.RegisterSMSForSending(...);
  ...
}
```

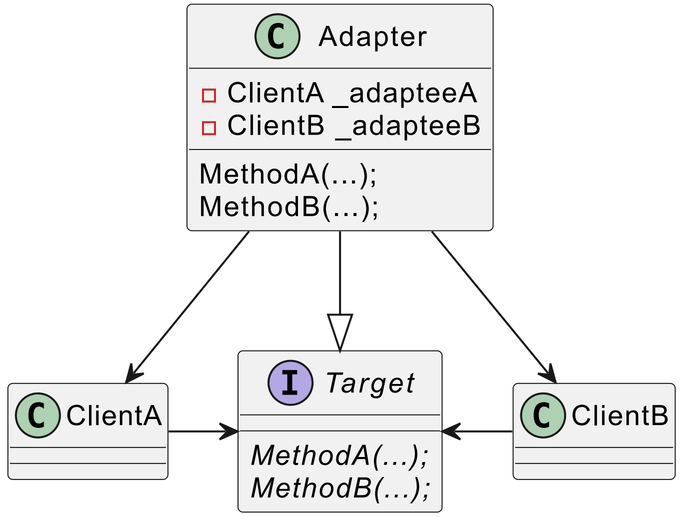
```
services.AddSingleton<SMSProvider>();
services.AddSingleton<INotificationProvider, SMSAdapter>();
...
public class NotificationService(IEnumerable<INotificationProvider> providers) {
   foreach (var provider in providers) provider.SendMessage(...);
}
```

Adapter - summary

- how much logic in adapter? - no more than needed for conversion
- adapter is meant to change the interface of an *existing* object - there is no "abstraction" or "implementation" abstraction like in Strategy or Bridge
- typically used as reactive, not proactive design pattern - to satisfy incompatible interfaces
- is *Mapper* an Adapter? 🤔
- typically it adapts *single* object - while Facade "adapts" multiple

Adapter - PS

We could create two-way adapter - for two-way communication

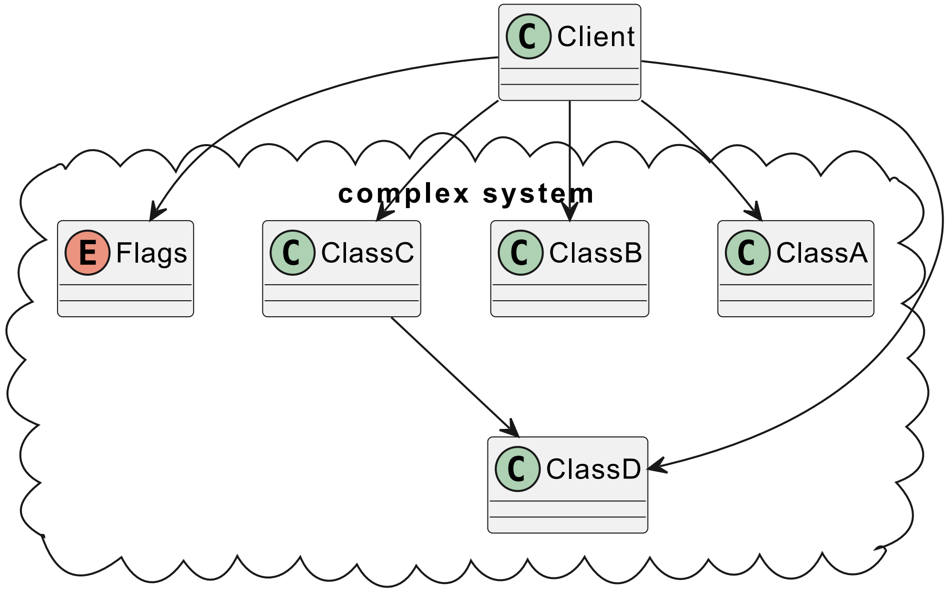


Adapter demo

Facade

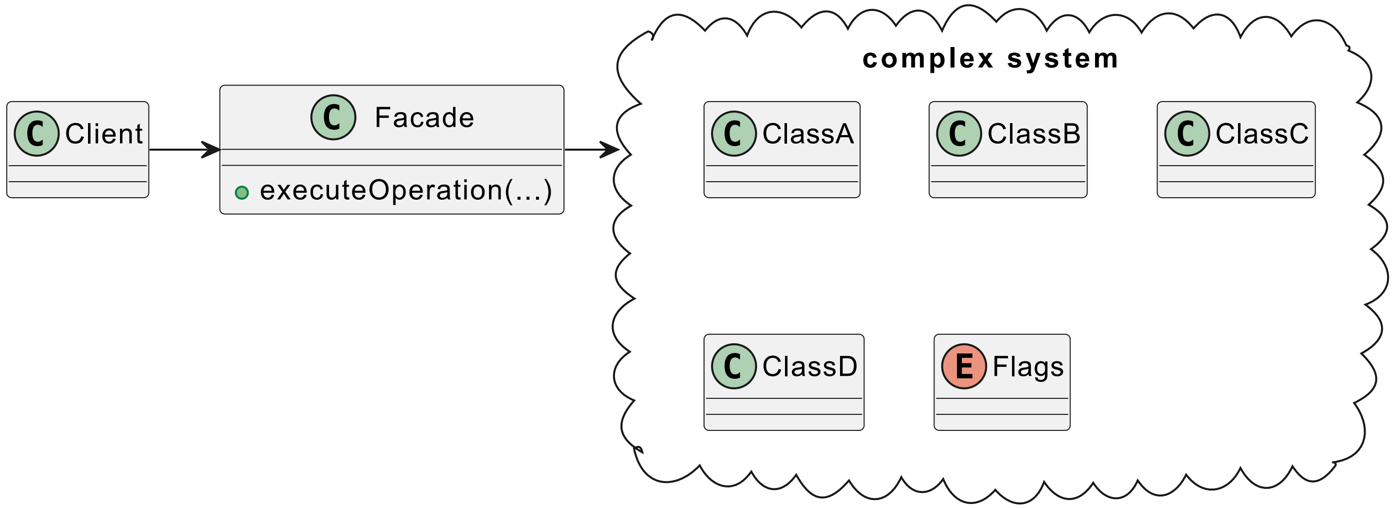
Facade

- converts a complex "class system" into a simplified interface
- a facade reduces the overall complexity of an application and helps move unwanted dependencies to a single place in the program
- helps to avoid *Law of Demeter* (**`classC.ClassD.ClassX.Flags`** ❌)



Facade

- converts a complex "class system" into a simplified interface
- a facade reduces the overall complexity of an application and helps move unwanted dependencies to a single place in the program
- helps to avoid *Law of Demeter* (**`classC.ClassD.ClassX.Flags`** ❌)



Facade - example #1

VulkanSharp - open source .NET binding for the Vulkan API

```
public override void DrawFrame ()
{
   if (!initialized) return;

   uint nextIndex = device.AcquireNextImageKHR (swapchain, ulong.MaxValue, semaphore);
   device.ResetFence (fence);
   var submitInfo = new SubmitInfo {
      WaitSemaphores = new Semaphore [] { semaphore },
      WaitDstStageMask = new PipelineStageFlags [] { PipelineStageFlags.AllGraphics },
      CommandBuffers = new CommandBuffer [] { commandBuffers [nextIndex] }
   };
   queue.Submit (submitInfo, fence);
   device.WaitForFence (fence, true, 100000000);
   var presentInfo = new PresentInfoKhr {
      Swapchains = new SwapchainKhr [] { swapchain },
      ImageIndices = new uint [] { nextIndex }
   };
   queue.PresentKHR (presentInfo);
}
```

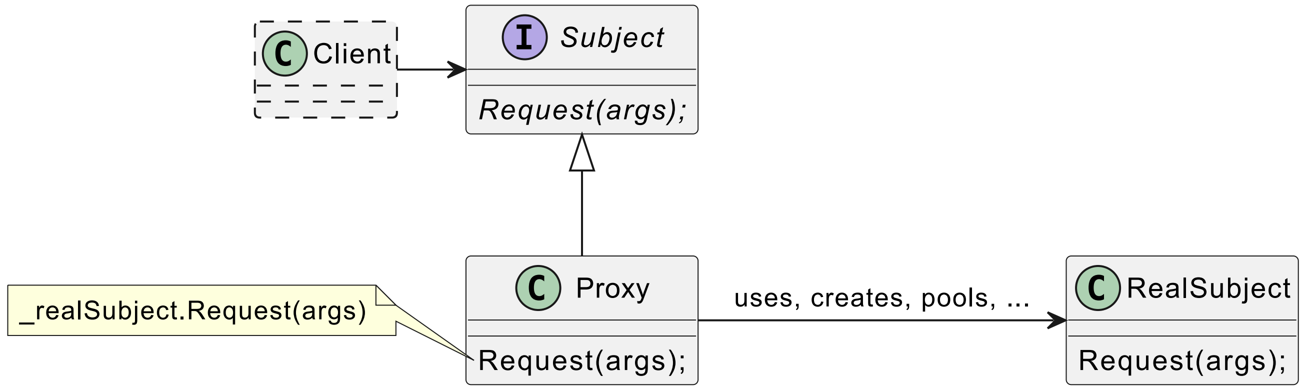
Facade - summary

- Adapter tries to make the existing interface compatible, Facade defines a new, simpler interface for existing objects
- the intent is a real difference:
  - Adapter pattern makes interface (of sth) compatible with a client expectations
  - Facade pattern provides a simplified interface (of sth)
- Adapter usually wraps one object, Facade usually works with a complex system of objects
- it decouples client from the (concrete) complex system
- like in Adapter there is no "abstraction" or "implementation" abstraction like in Strategy or Bridge
- pretty often they become Singletons

Proxy

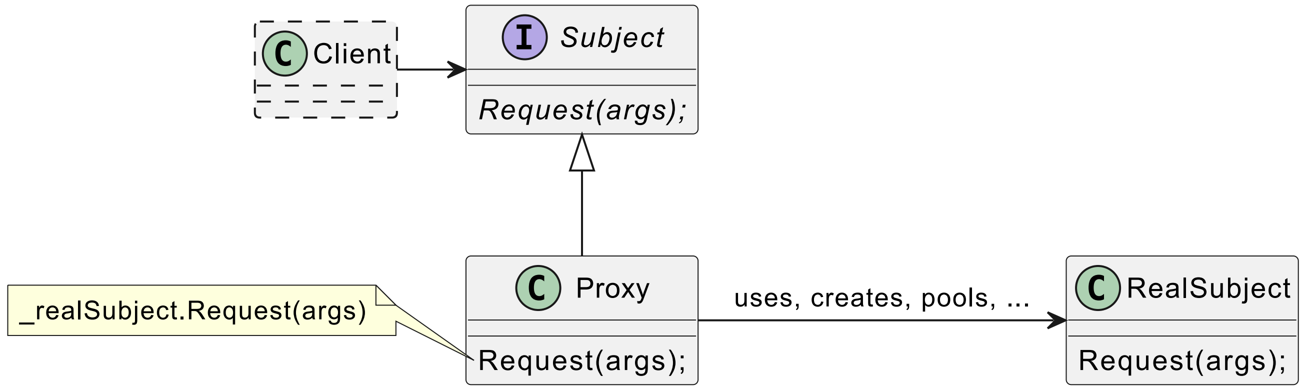
Proxy

- provide a placeholder for another object to control access to it
- does not change the interface (unlike Adapter pattern)



Proxy

- provide a placeholder for another object to control access to it
- does not change the interface (unlike Adapter pattern)



- yes, very similar to Adapter's *Adapter*/*Target*/*Adaptee* but interface is not changed

Proxy

- **remote proxy** - using to access remote resources. Like WCF proxy
- **virtual proxy** (aka "lazy initialization") - control access to resource that is expensive to create using *pooling*, *lazy initialization*, *copy-on-write* etc.
- **protection proxy** - rights access management
- (legacy) smart reference proxy - aka "smart pointers"

Proxy - remote proxy

.NET Remoting (😍😬)

```
IHello obj = (IHello)Activator.GetObject(
   typeof(IHello),
   "tcp://localhost:8080/HelloService");
string result = obj.SayHello("World");
```

Proxy - remote proxy

.NET Remoting (😍😬)

```
IHello obj = (IHello)Activator.GetObject(
   typeof(IHello),
   "tcp://localhost:8080/HelloService");
string result = obj.SayHello("World");
```

gRPC

```
syntax = "proto3";
service Greeter {
  rpc SayHello (HelloRequest) returns (HelloReply);
}
message HelloRequest {
  string name = 1;
}
message HelloReply {
  string message = 1;
}
```

```
var channel = GrpcChannel.ForAddress("https://localhost:5001");
var client = new Greeter.GreeterClient(channel);
```

Proxy - generic

Castle DynamicProxy a lightweight, lightning fast framework for generating proxies on the fly, used extensively by multiple projects within Castle (Windsor) and outside of it (Moq, NSubstitute, FakeItEasy, Rhino Mocks)

```
void Main()
{
   var generator = new Castle.DynamicProxy.ProxyGenerator();
   Calculator c = generator.CreateClassProxy<Calculator>(
      new CalculatorInterceptor());
   c.Add(11, 22);
}

public class Calculator
{
   public virtual void Add(int a, int b) => Console.WriteLine(a + b);
}

public class CalculatorInterceptor : IInterceptor
{
   public void Intercept(IInvocation invocation)
   {
      Console.WriteLine("Before!");
      invocation.Proceed();
   }
}
```

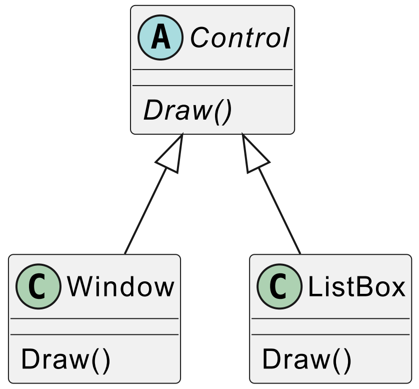
💡 Question

Problem: How to model the drawing of a graphic element (control) so that it can be extended with:

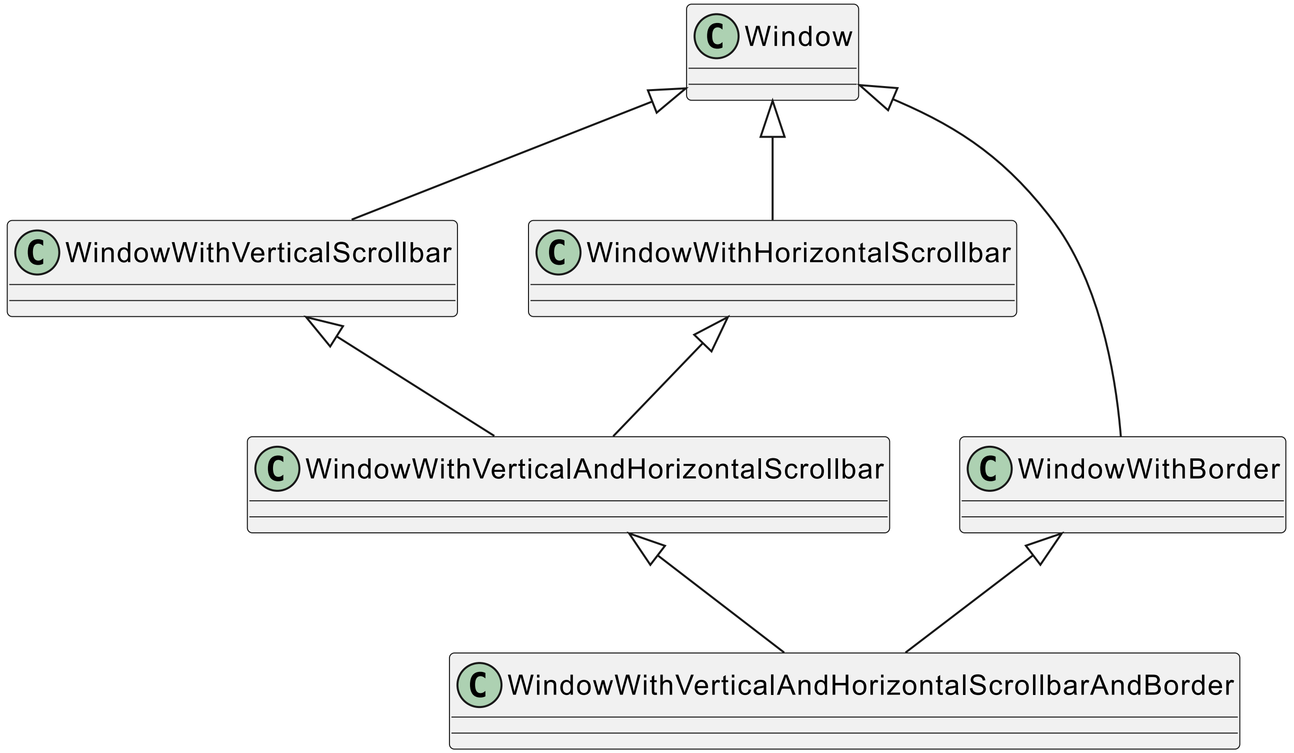
- Vertical and/or horizontal bar
- Frame

The change should be additive - we are not changing the existing code

Currently, we have two types of controls.

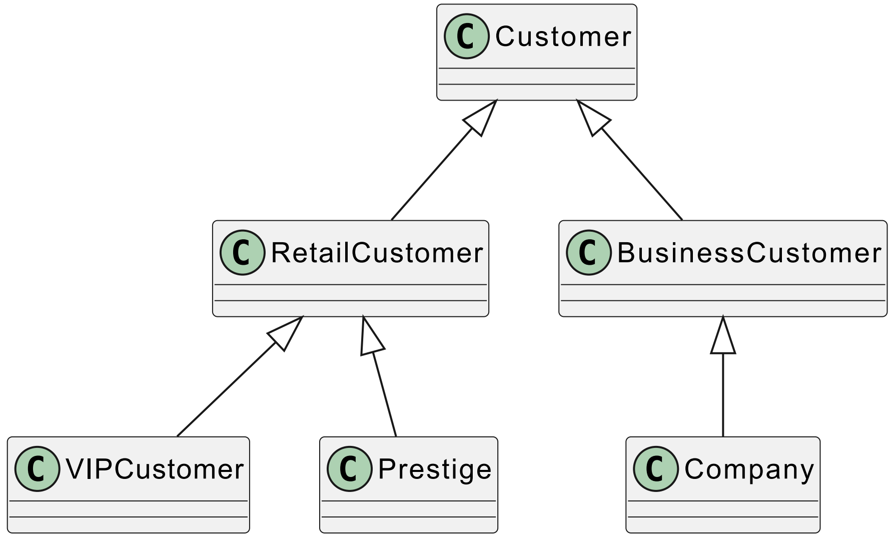


💡 Question



Window
WindowWithVerticalScrollbar
WindowWithHorizontalScrollbar
WindowWithVerticalAndHorizontalScrollbar
WindowWithBorder
WindowWithVerticalAndHorizontalScrollbarAndBorder

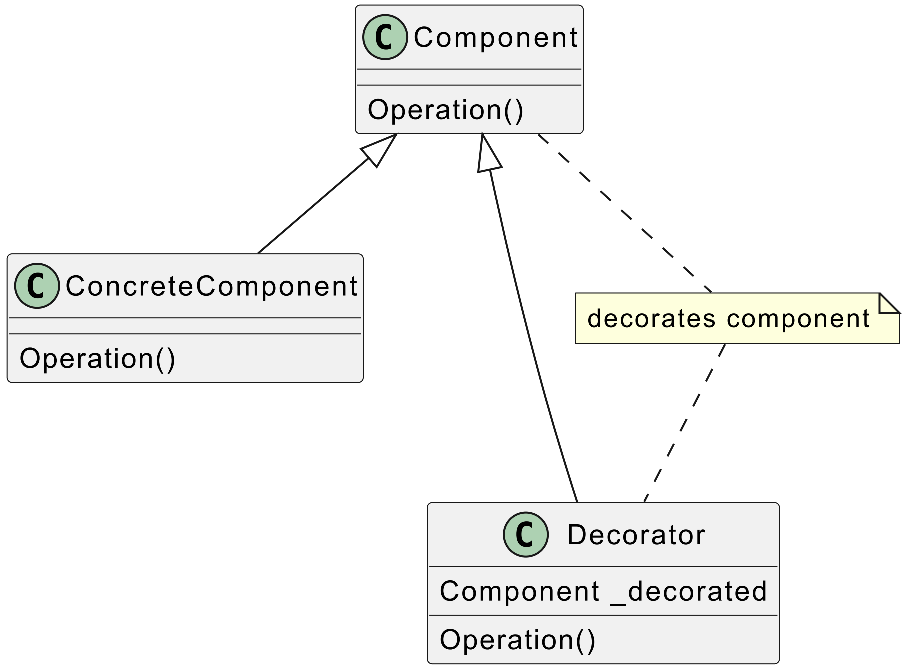
💡 Question


Customer
RetailCustomer
BusinessCustomer
VIPCustomer
Prestige
Company

Decorator

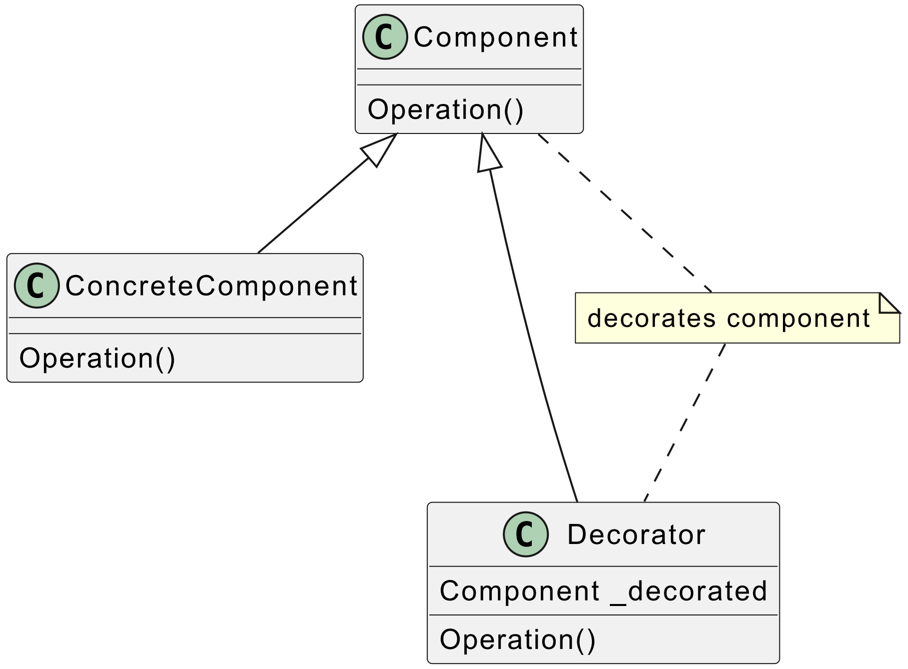
Decorator

- decorator allows to add new responsibilities to objects by placing these objects in wrapper objects that contain the appropriate behaviours
- decorator is effectively a **matrioshka** - the next delegate can wrap the next one and so on.
- decorator **is** a component and **contains** a component



Decorator

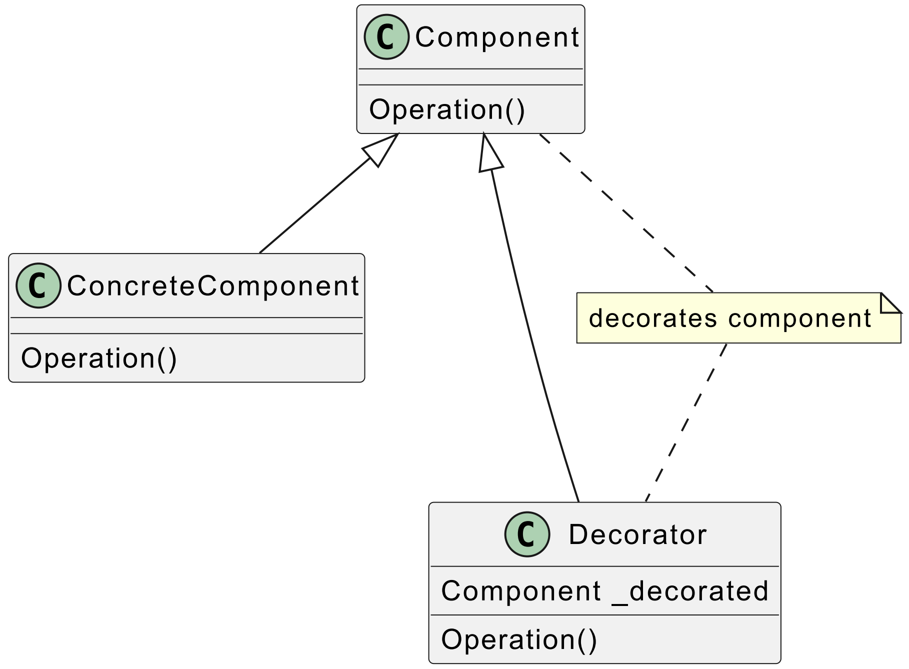
- decorator allows to add new responsibilities to objects by placing these objects in wrapper objects that contain the appropriate behaviours
- decorator is effectively a **matrioshka** - the next delegate can wrap the next one and so on.
- decorator **is** a component and **contains** a component

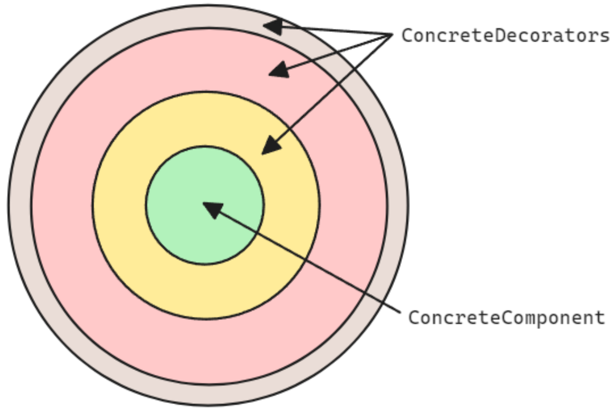


- **`ConcreteComponent`**(s) define an object to which additional responsibilities can be attached

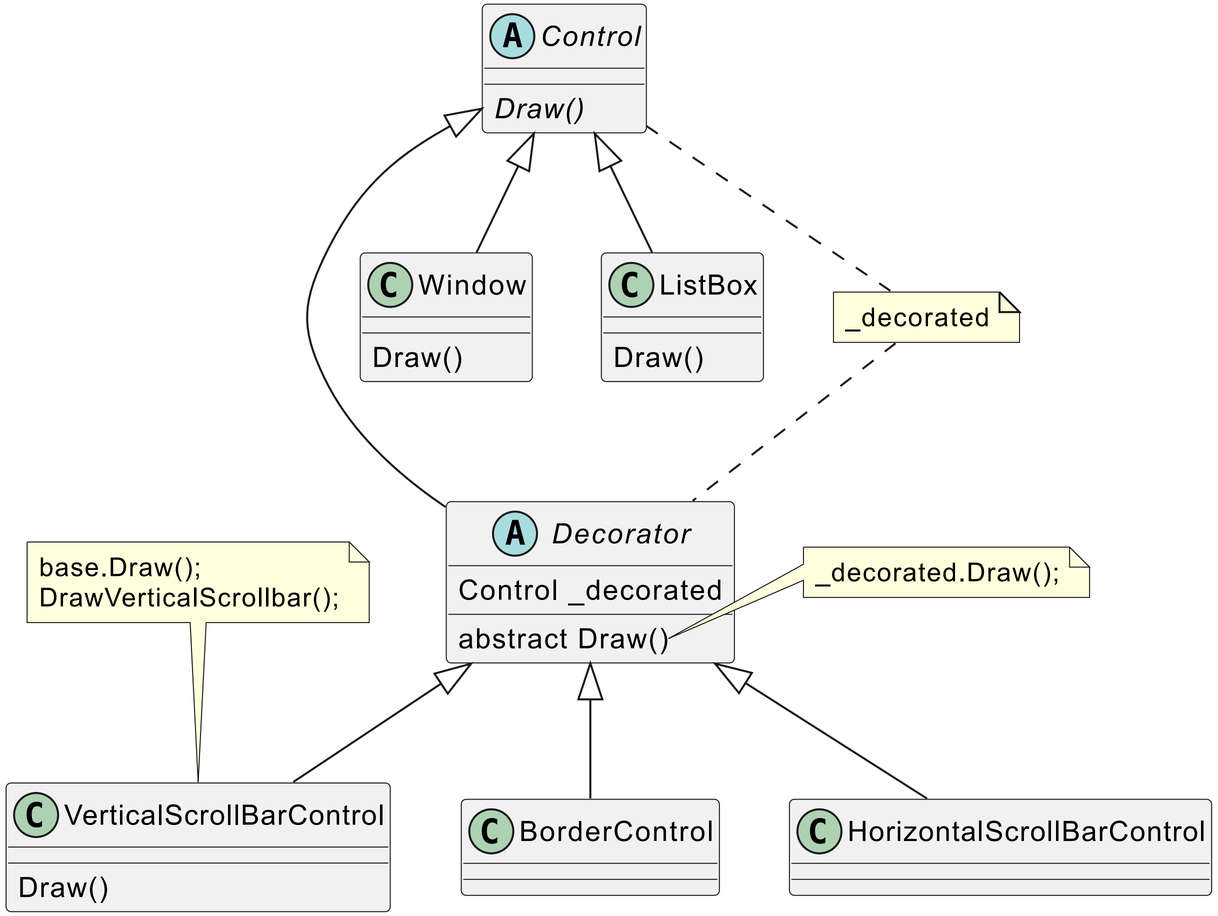
Decorator

- decorator allows to add new responsibilities to objects by placing these objects in wrapper objects that contain the appropriate behaviours
- decorator is effectively a **matrioshka** - the next delegate can wrap the next one and so on.
- decorator **is** a component and **contains** a component



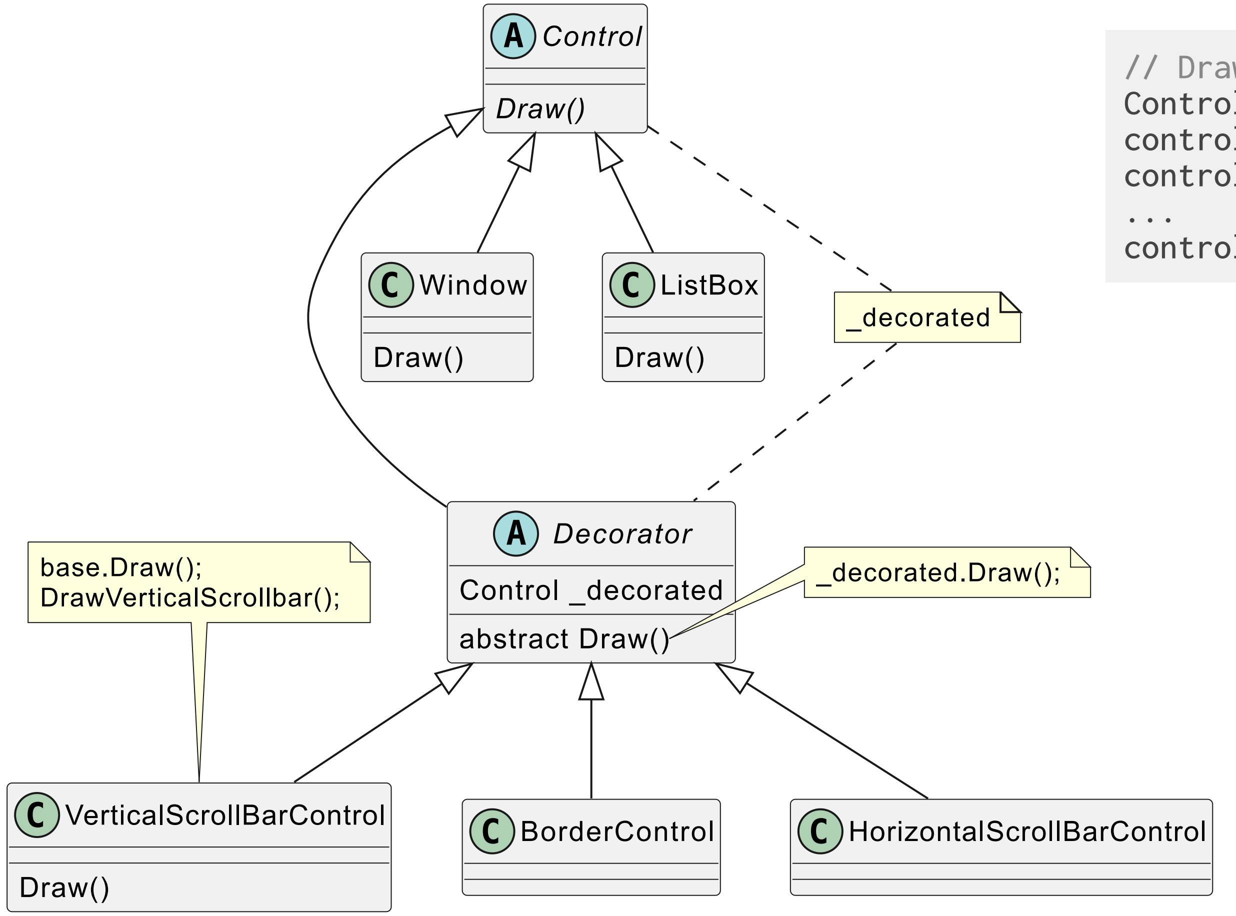


💡 Question



A Control
Draw()
C Window
Draw()
C ListBox
Draw()
_decorated
A Decorator
Control _decorated
abstract Draw()
_decorated.Draw();
base.Draw();
DrawVerticalScrollbar();
C VerticalScrollBarControl
Draw()
C BorderControl
C HorizontalScrollBarControl

Question



```
// Draw 'WindowWithVerticalAndHorizontalScrollbar'
Control control = new Window();
control = new HorizontalScrollBar(control);
control = new VerticalScrollBar(control);
...
control.Draw();
```

Decorator - example #1

- configuring many various configuration getters in an application:

```
new LoggingConfigPathProvider(
   new AppendSubdirectoryConfigPathProvider("section",
       new DecryptingConfigPathProvider(decrypter,
           new AppSettingsConfigPathProvider())));
```

instead of, for example:

```
new AppSettingsConfigPathProvider("section", isLogged: true, decrypt: decrypter);
```

- make sense if there are **many** such variations configured and/or we expect to

Decorator - example #2

- Scrutor library - assembly scanning and decoration extensions for `Microsoft.Extensions.DependencyInjection`
- e.g. for creating *deprecated* objects - gradually, for example, for specific methods?

```
var collection = new ServiceCollection();

collection.Scan(scan => scan
    .FromAssemblyOf<IDiscountsService>()
        .AddClasses(classes => classes.AssignableTo<IDiscountsService>())
            .AsImplementedInterfaces()
            .WithTransientLifetime()

collection.Decorate<IDiscountsService, DeprecatedDiscountsService>();
```

Decorator demo

Decorator

- select **`Component`**, **`ConcreteComponent`** and **`Decorator`** concepts wisely
  - what concrete component do we have?
  - are decorators really components? Or some... decorators

Decorator

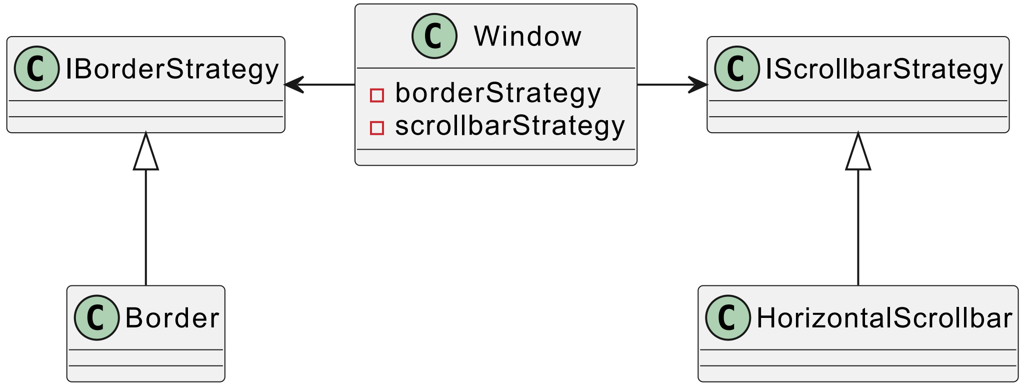
- select **`Component`**, **`ConcreteComponent`** and **`Decorator`** concepts wisely
  - what concrete component do we have?
  - are decorators really components? Or some... decorators
- used more often for "simple" infrastructure packaging and *cross cutting concerns* than for defining business logic (although... you can)

Decorator

- select `Component`, `ConcreteComponent` and `Decorator` concepts wisely
  - what concrete component do we have?
  - are decorators really components? Or some... decorators
- used more often for "simple" infrastructure packaging and *cross cutting concerns* than for defining business logic (although... you can)
- its main strength is its "recursive composition" because the wrapped element has the same interface as the decorator itself, so we can combine logics - the "stack" of commitments seen above

Decorator

- select **`Component`**, **`ConcreteComponent`** and **`Decorator`** concepts wisely
  - what concrete component do we have?
  - are decorators really components? Or some… decorators
- used more often for "simple" infrastructure packaging and *cross cutting concerns* than for defining business logic (although… you can)
- its main strength is its "recursive composition" because the wrapped element has the same interface as the decorator itself, so we can combine logics - the "stack" of commitments seen above
- Decorator ⚔ Strategy - GoF compares it to *"Changing the skin of an object versus changing its guts"*



Mapper

Wrap/unwrap DTO adapters

```
var customer = new Customer
{
    CustomerID = customerDTO.ID,
    Name = customerDTO.FirstName + " " + customerDTO.LastName,
    Address = customerDTO.Address,
    City = customerDTO.City,
    State = customerDTO.State,
    Zip = customerDTO.PostalCode
}
```

Wrap/unwrap DTO adapters

```
var customer = new Customer
{
    CustomerID = customerDTO.ID,
    Name = customerDTO.FirstName + " " + customerDTO.LastName,
    Address = customerDTO.Address,
    City = customerDTO.City,
    State = customerDTO.State,
    Zip = customerDTO.PostalCode
}
```

We can use, for example, AutoMapper:

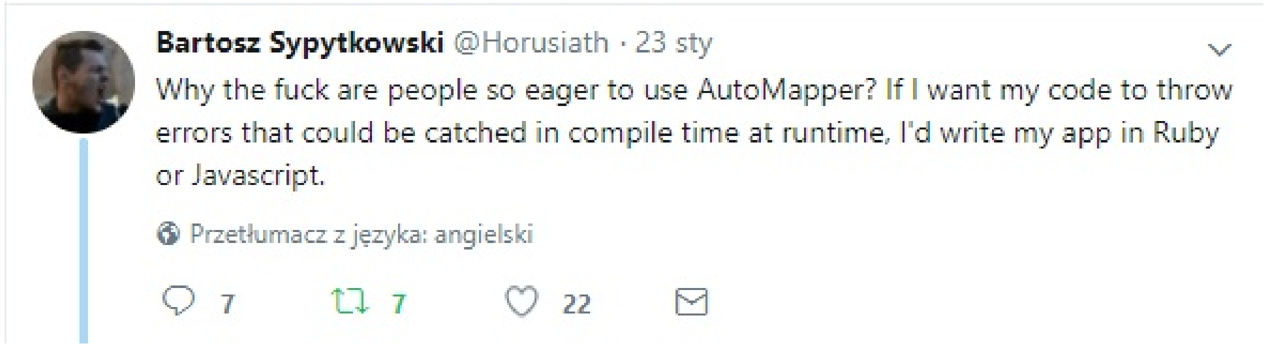
```
var config = new MapperConfiguration(cfg => cfg.CreateMap<CustomerDto, Customer>());

var mapper = new Mapper(config);
Customer customer = mapper.Map<Customer>(customerDTO);
```

AutoMapper controversy

```
Mapper.Initialize(cfg => {
  cfg.CreateMap<UserEntity, UserDTO>()
     .ForMember(x => x.FullName,
                opt =>
                opt.MapFrom(x=> $"{x.FirstName} {x.LastName} ({x.Address.City})"));
});
var userEntity = new UserEntity()
{
    FirstName = "Cezary",
    LastName = "Piątek",
    Address = null
};
var userDto = Mapper.Map<UserDTO>(userEntity);
var serialized = JsonConvert.SerializeObject(userDto, Formatting.Indented);
```

AutoMapper controversy

```
Mapper.Initialize(cfg => {
  cfg.CreateMap<UserEntity, UserDTO>()
     .ForMember(x => x.FullName,
                opt =>
                opt.MapFrom(x=> $"{x.FirstName} {x.LastName} ({x.Address.City})"));
});
var userEntity = new UserEntity()
{
    FirstName = "Cezary",
    LastName = "Piątek",
    Address = null
};
var userDto = Mapper.Map<UserDTO>(userEntity);
var serialized = JsonConvert.SerializeObject(userDto, Formatting.Indented);
```



Bartosz Sypytkowski @Horusiath · 23 sty

Why the fuck are people so eager to use AutoMapper? If I want my code to throw errors that could be catched in compile time at runtime, I'd write my app in Ruby or Javascript.

Przetłumacz z języka: angielski

7 7 22

AutoMapper controversy

- misleads static analysis - some/all fields appear to be unused
- misleads "show usage" - mapping is "out of the box", we will not find it after using the fields
- hard to debug - declarative transformations (as on the previous slide)
- mixing the logic of complex tranformations into the code of the infrastructure (which is AutoMapper) + i.a.
- it is not possible to find/check the use of a particular mapping
- performance issues

AutoMapper controversy

- misleads static analysis - some/all fields appear to be unused
- misleads "show usage" - mapping is "out of the box", we will not find it after using the fields
- hard to debug - declarative transformations (as on the previous slide)
- mixing the logic of complex tranformations into the code of the infrastructure (which is AutoMapper) + i.a.
- it is not possible to find/check the use of a particular mapping
- performance issues

Solutions:

- do not use AutoMapper - write boring manual tranformations
- use only in VERY simple 1-1 mappings (when there are hundreds of them?)
- use very deliberately - AutoMapper Usage Guidelines by Jimmy Bogard
- use alternatives Mapster or Mapperly (source generated!)

Adapter vs Facade vs Proxy vs Decorator

- technically may be very similar, but the **intent** is different:
    - Adapter - makes interfaces compatible
    - Facade - hides some complex logic/complex set of objects
    - Proxy - intercepts call and controls accept to another object
    - Decorator - adds behaviour to something, and is **composable**, so especially if we want to combine one+ behaviours