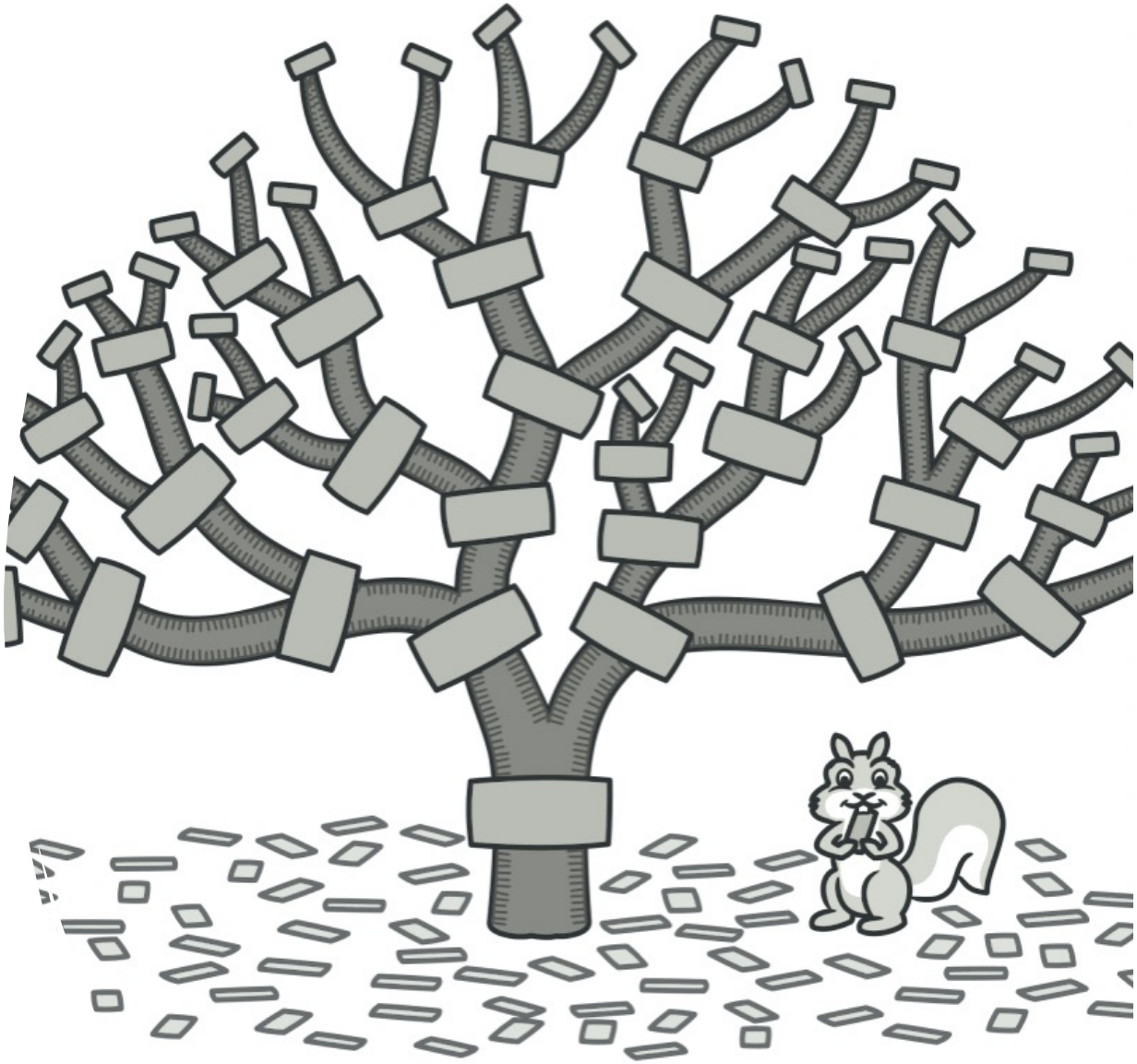
MODELLI STRUTTURALI: COMPOSITE PATTERN

INTENT

- Composite is a structural design pattern that lets you compose objects into tree structures and then work with these structures as if they were individual objects.

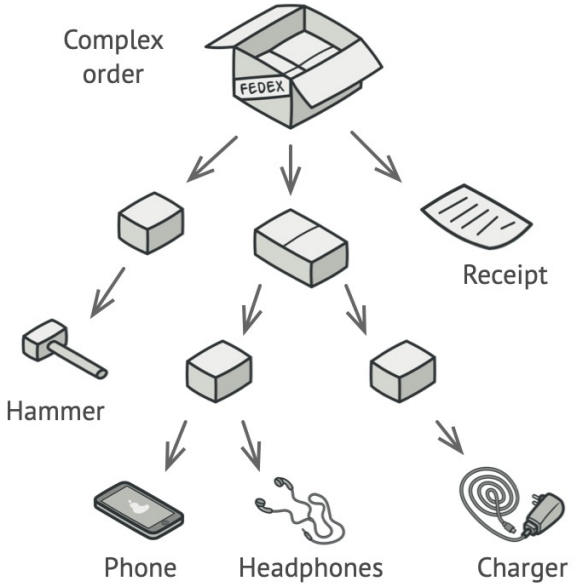


PROBLEM

Using the Composite pattern makes sense only when the core model of your app can be represented as a tree.

- For example, imagine that you have two types of objects: Products and Boxes.
- A Box can contain several Products as well as a number of smaller Boxes.
- These little Boxes can also hold some Products or even smaller Boxes, and so on.
- Say you decide to create an ordering system that uses these classes.
- Orders could contain simple products without any wrapping, as well as boxes stuffed with products...and other boxes.

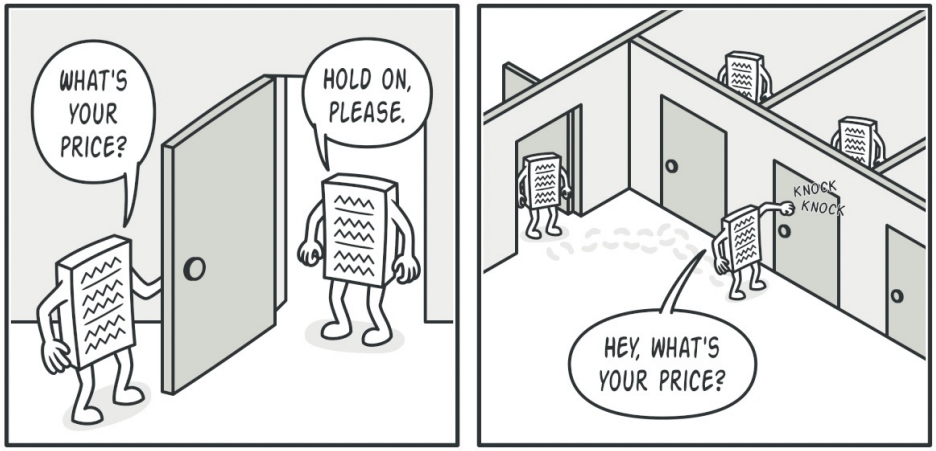
How would you determine the total price of such an order?



An order might comprise various products, packaged in boxes, which are packaged in bigger boxes and so on. The whole structure looks like an upside down tree.

SOLUTION

- The Composite pattern suggests that you work with Products and Boxes through a common interface which declares a method for calculating the total price.

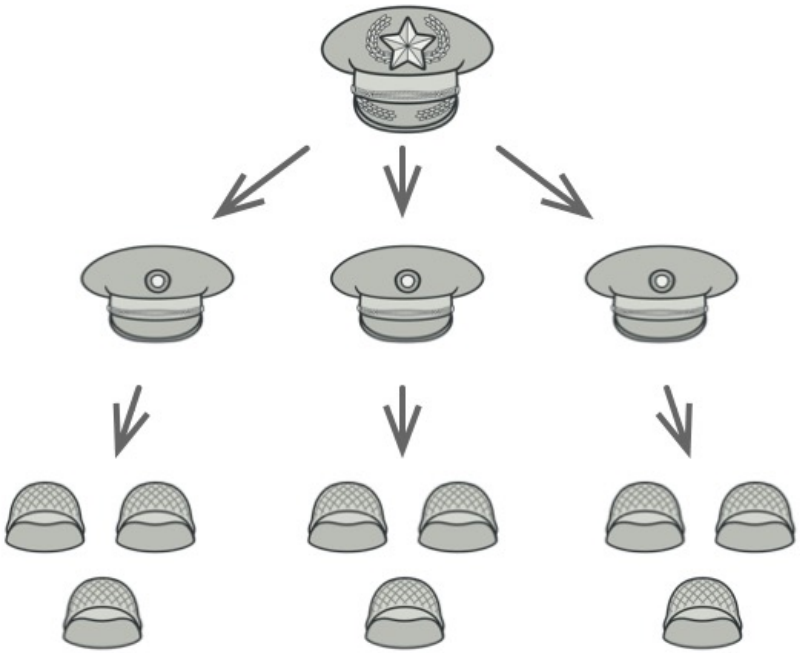


The Composite pattern lets you run a behavior recursively over all components of an object tree.

- The greatest benefit of this approach is that you don't need to care about the concrete classes of objects that compose the tree. You don't need to know whether an object is a simple product or a sophisticated box. You can treat them all the same via the common interface.

REAL-WORLD ANALOGY

- Armies of most countries are structured as hierarchies. An army consists of several divisions; a division is a set of brigades, and a brigade consists of platoons, which can be broken down into squads. Finally, a squad is a small group of real soldiers. Orders are given at the top of the hierarchy and passed down onto each level until every soldier knows what needs to be done.



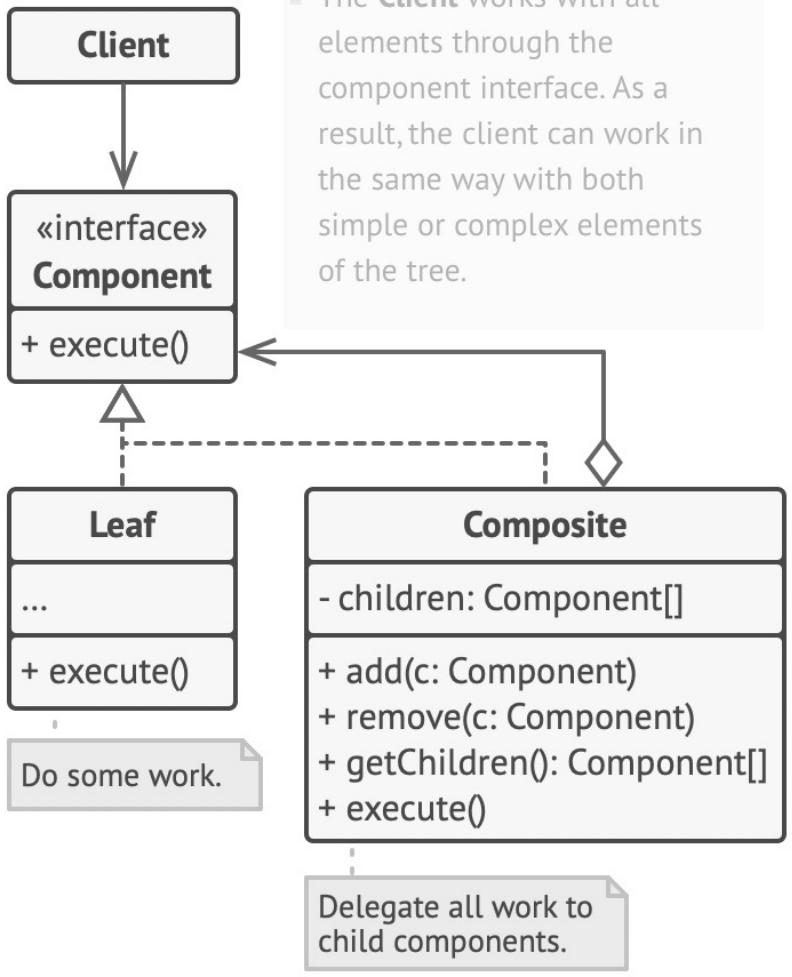
An example of a military structure.

STRUCTURE

1 The **Component** interface describes operations that are common to both simple and complex elements of the tree.

2 The **Leaf** is a basic element of a tree that doesn't have sub-elements.

Usually, leaf components end up doing most of the real work, since they don't have anyone to delegate the work to.



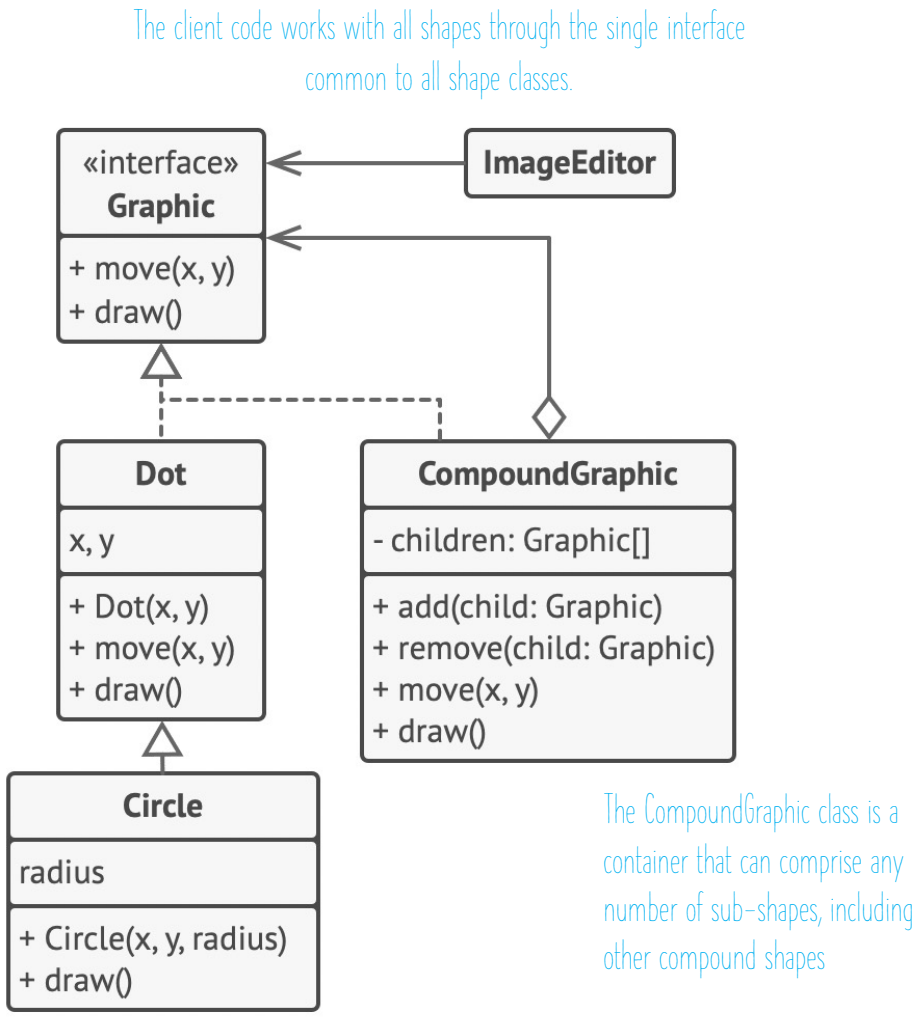
4 The **Client** works with all elements through the component interface. As a result, the client can work in the same way with both simple or complex elements of the tree.

3 The **Container** (aka *composite*) is an element that has sub-elements: leaves or other containers. A container doesn't know the concrete classes of its children. It works with all sub-elements only via the component interface.

Upon receiving a request, a container delegates the work to its sub-elements, processes intermediate results and then returns the final result to the client.

#PSEUDOCODE

- In this example, the Composite pattern lets you implement stacking of geometric shapes in a graphical editor.



The geometric shapes editor example.

CODE

```
// The composite class represents complex components that may
// have children. Composite objects usually delegate the actual
// work to their children and then "sum up" the result.
class CompoundGraphic implements Graphic is
    field children: array of Graphic

    // A composite object can add or remove other components
    // (both simple or complex) to or from its child list.
    method add(child: Graphic) is
        // Add a child to the array of children.

    method remove(child: Graphic) is
        // Remove a child from the array of children.

    method move(x, y) is
        foreach (child in children) do
            child.move(x, y)

    // A composite executes its primary logic in a particular
    // way. It traverses recursively through all its children,
    // collecting and summing up their results. Since the
    // composite's children pass these calls to their own
    // children and so forth, the whole object tree is traversed
    // as a result.
    method draw() is
        // 1. For each child component:
        //     - Draw the component.
        //     - Update the bounding rectangle.
        // 2. Draw a dashed rectangle using the bounding
        // coordinates.
```

```
// The component interface declares common operations for both
// simple and complex objects of a composition.
interface Graphic is
    method move(x, y)
    method draw()

// The leaf class represents end objects of a composition. A
// leaf object can't have any sub-objects. Usually, it's leaf
// objects that do the actual work, while composite objects only
// delegate to their sub-components.
class Dot implements Graphic is
    field x, y

    constructor Dot(x, y) { ... }

    method move(x, y) is
        this.x += x, this.y += y

    method draw() is
        // Draw a dot at X and Y.
```

```
// All component classes can extend other components.
class Circle extends Dot is
    field radius

    constructor Circle(x, y, radius) { ... }

    method draw() is
        // Draw a circle at X and Y with radius R.
```

CODE

```
// The client code works with all the components via their base
// interface. This way the client code can support simple leaf
// components as well as complex composites.
class ImageEditor is
    field all: CompoundGraphic

    method load() is
        all = new CompoundGraphic()
        all.add(new Dot(1, 2))
        all.add(new Circle(5, 3, 10))
        // ...

    // Combine selected components into one complex composite
    // component.
    method groupSelected(components: array of Graphic) is
        group = new CompoundGraphic()
        foreach (component in components) do
            group.add(component)
            all.remove(component)
        all.add(group)
        // All components will be drawn.
        all.draw()
```

APPLICABILITY

Use the Composite pattern when you have to implement a tree-like object structure.

Use the pattern when you want the client code to treat both simple and complex elements uniformly.

HOW TO IMPLEMENT

1. Tree Structure: Represent the app's core model as a tree of simple elements and containers. Containers should hold both element types.
2. Component Interface: Define an interface with methods applicable to both simple and complex components.
3. Leaf Class: Create classes for simple elements (leaves). Multiple leaf types are allowed.
4. Container Class: Create a container class with an array to store sub-elements, declared using the component interface type.
5. Delegation: Implement container methods to delegate most work to sub-elements.
6. Child Management: Add methods for adding and removing child elements in the container.
7. Optional Interface Methods: If child management is declared in the component interface, leaf classes will have empty implementations, enabling uniform tree manipulation at the cost of Interface Segregation Principle adherence.

PROS & CONS

ou can work with complex tree structures more conveniently: use polymorphism and recursion to your advantage.

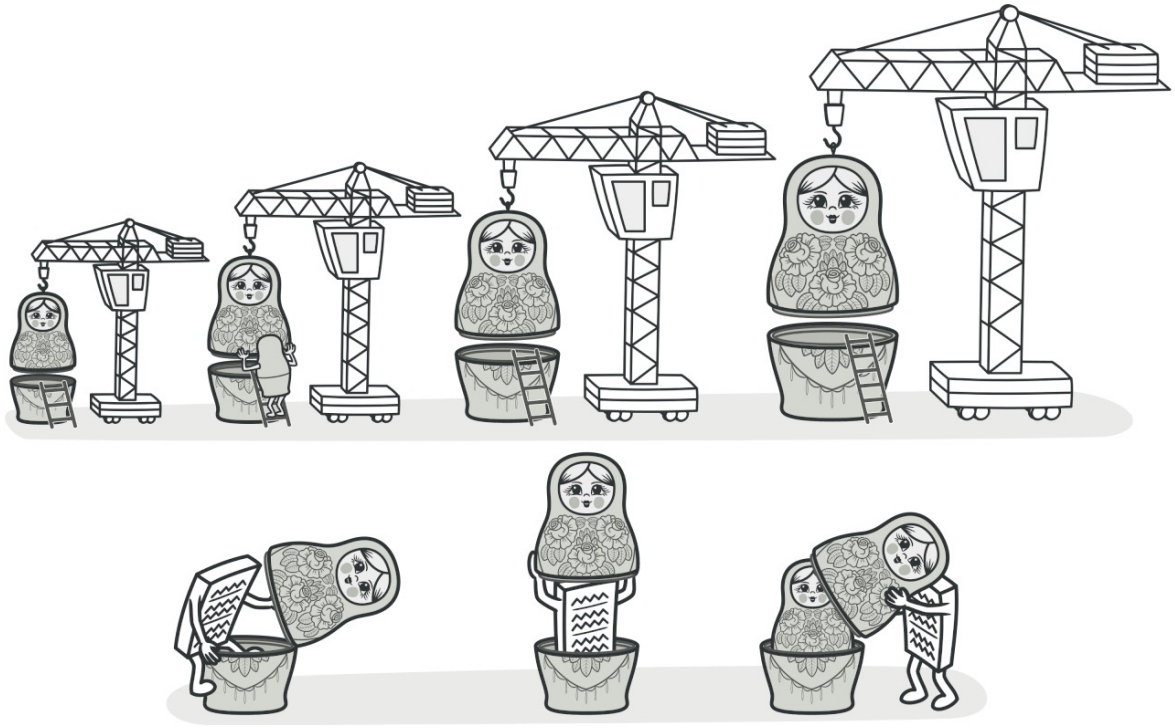
Open/Closed Principle. You can introduce new element types into the app without breaking the existing code, which now works with the object tree.

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.

MODELLI STRUTTURALI:
DECORATOR PATTERN

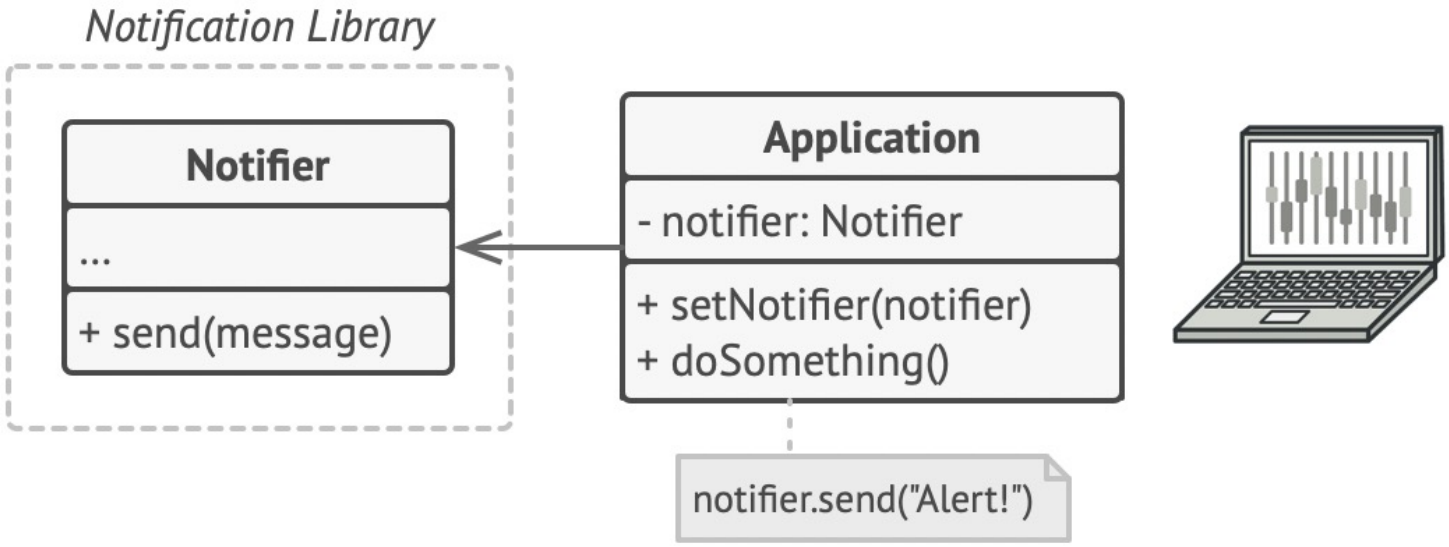
INTENT

- **Decorator** is a structural design pattern that lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.



PROBLEM (1)

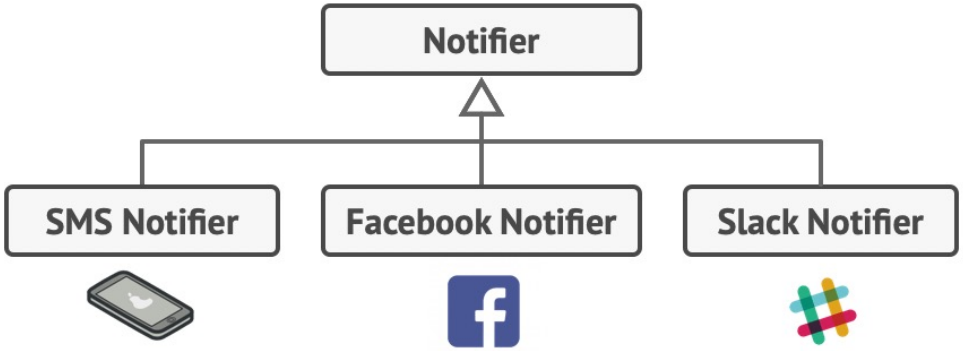
- Imagine that you're working on a notification library which lets other programs notify their users about important events.



A program could use the notifier class to send notifications about important events to a predefined set of emails.

PROBLEM (2)

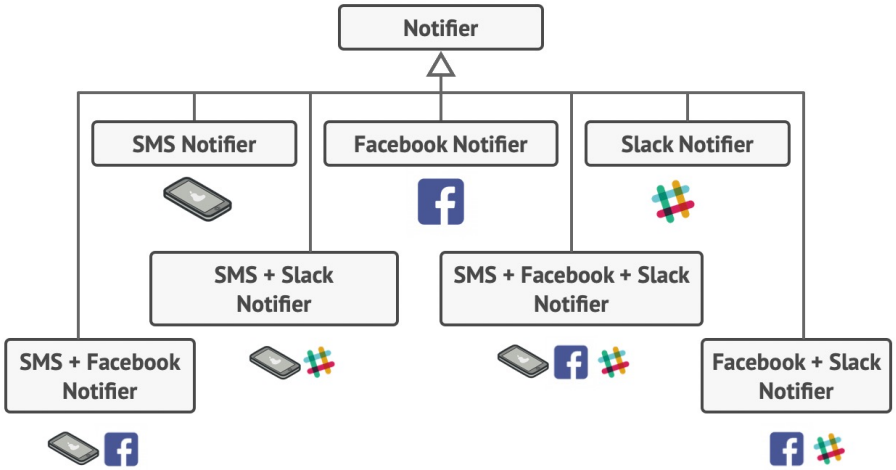
- At some point, you realize that users of the library expect more than just email notifications. Many of them would like to receive an SMS about critical issues. Others would like to be notified on Facebook and, of course, the corporate users would love to get Slack notifications.



Each notification type is implemented as a notifier's subclass.

PROBLEM (3)

- You tried to address that problem by creating special subclasses which combined several notification methods within one class. However, it quickly became apparent that this approach would bloat the code immensely, not only the library code but the client code as well.



Combinatorial explosion of subclasses.

- You have to find some other way to structure notifications classes so that their number won't accidentally break some Guinness record

SOLUTION (1)

Extending a class is the first thing that comes to mind when you need to alter an object's behavior. However, inheritance has several serious caveats that you need to be aware of.

1. Inheritance is static. You can't alter the behavior of an existing object at runtime. You can only replace the whole object with another one that's created from a different subclass.
2. Subclasses can have just one parent class. In most languages, inheritance doesn't let a class inherit behaviors of multiple classes at the same time.

SOLUTION (2)

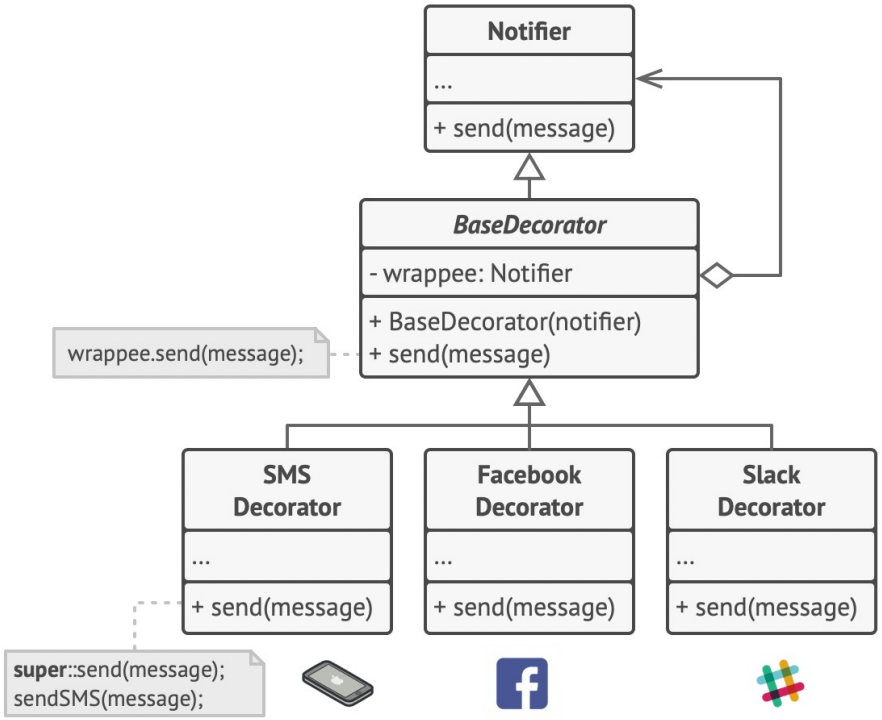
- Using Aggregation or Composition instead of Inheritance. Both of the alternatives work almost the same way: one object has a reference to another and delegates it some work, whereas with inheritance, the object itself is able to do that work, inheriting the behavior from its superclass.

Inheritance vs. Aggregation

SOLUTION (3)

- "Wrapper" is the alternative nickname for the Decorator pattern that clearly expresses the main idea of the pattern. A wrapper is an object that can be linked with some target object. The wrapper contains the same set of methods as the target and delegates to it all requests it receives. However, the wrapper may alter the result by doing something either before or after it passes the request to the target.
- When does a simple wrapper become the real decorator?

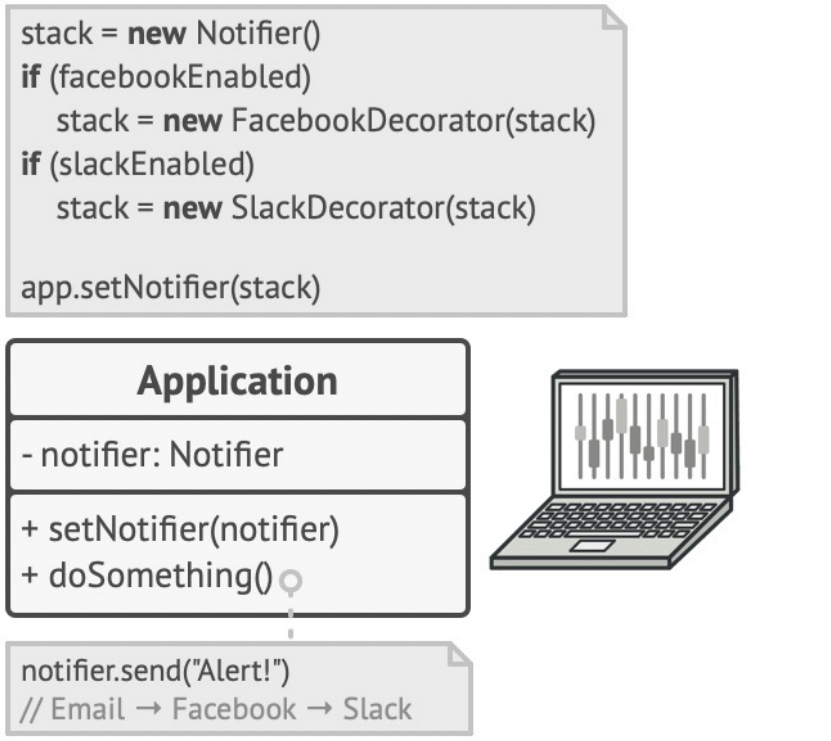
- In our notifications example, let's leave the simple email notification behavior inside the base Notifier class, but turn all other notification methods into decorators.



Various notification methods become decorators.

SOLUTION (4)

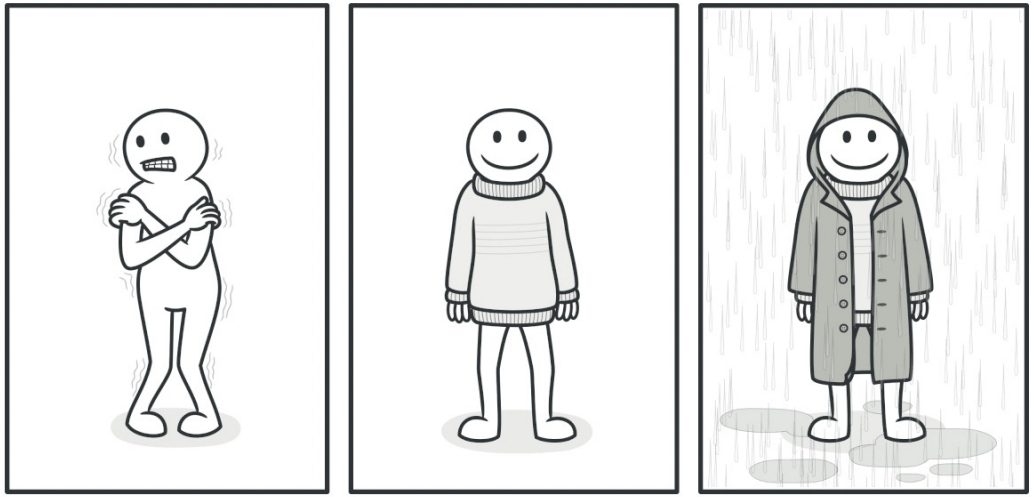
- The client code would need to wrap a basic notifier object into a set of decorators that match the client's preferences. The resulting objects will be structured as a stack.
- The last decorator in the stack would be the object that the client actually works with. Since all decorators implement the same interface as the base notifier, the rest of the client code won't care whether it works with the "pure" notifier object or the decorated one.



Apps might configure complex stacks of notification decorators.

REAL-WORLD ANALOGY

- Wearing clothes is an example of using decorators. When you're cold, you wrap yourself in a sweater. If you're still cold with a sweater, you can wear a jacket on top. If it's raining, you can put on a raincoat. All of these garments "extend" your basic behavior but aren't part of you, and you can easily take off any piece of clothing whenever you don't need it.

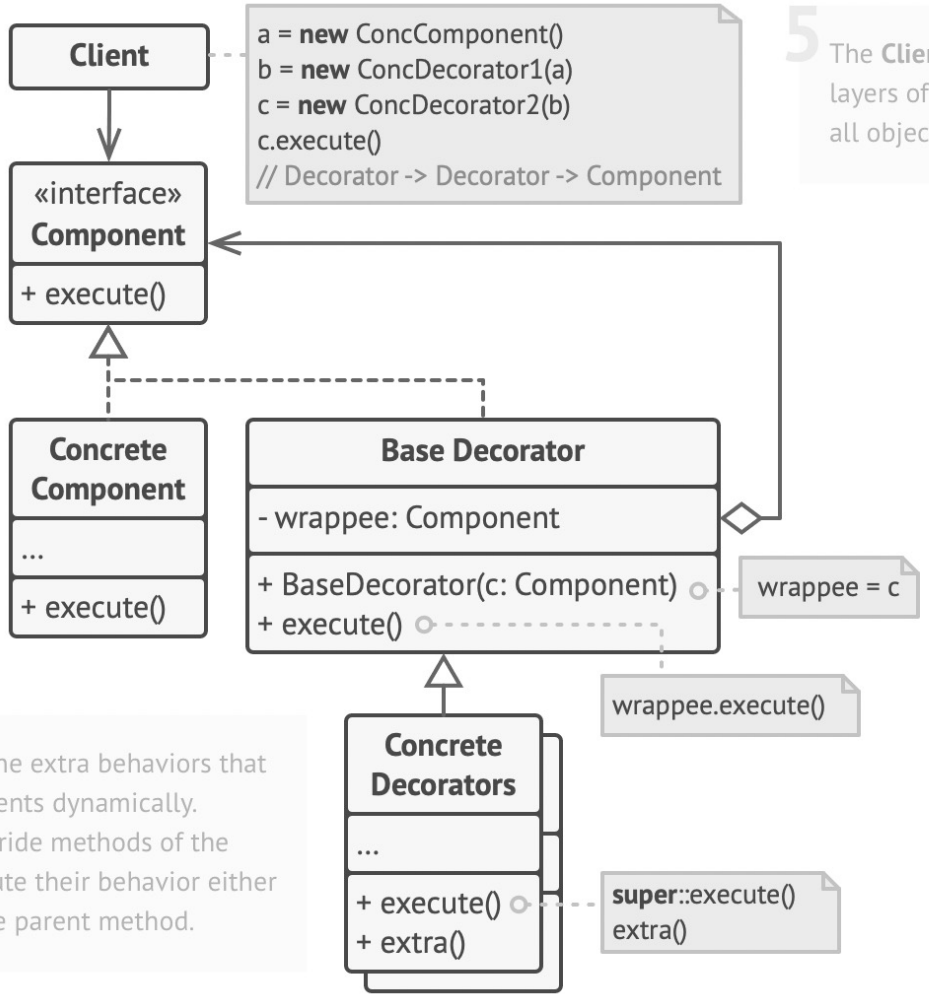


You get a combined effect from wearing multiple pieces of clothing.

STRUCTURE

1 The **Component** declares the common interface for both wrappers and wrapped objects.

2 **Concrete Component** is a class of objects being wrapped. It defines the basic behavior, which can be altered by decorators.



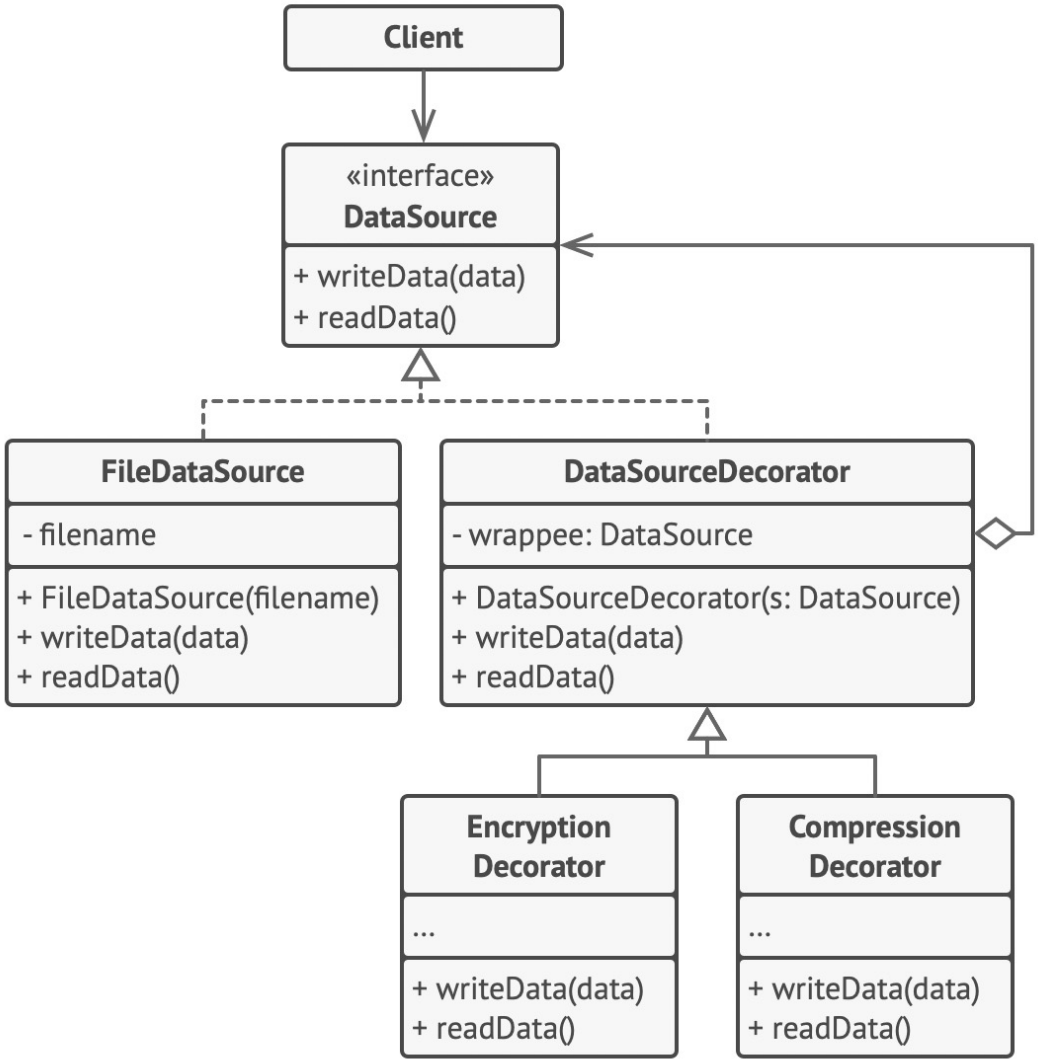
3 The **Base Decorator** class has a field for referencing a wrapped object. The field's type should be declared as the component interface so it can contain both concrete components and decorators. The base decorator delegates all operations to the wrapped object.

4 **Concrete Decorators** define extra behaviors that can be added to components dynamically. Concrete decorators override methods of the base decorator and execute their behavior either before or after calling the parent method.

5 The **Client** can wrap components in multiple layers of decorators, as long as it works with all objects via the component interface.

#PSEUDOCODE

- In this example, the Decorator pattern lets you compress and encrypt sensitive data independently from the code that actually uses this data.



The encryption and compression decorators example.

CODE

```
// The component interface defines operations that can be
// altered by decorators.
interface DataSource is
    method writeData(data)
    method readData():data

// Concrete components provide default implementations for the
// operations. There might be several variations of these
// classes in a program.
class FileDataSource implements DataSource is
    constructor FileDataSource(filename) { ... }

    method writeData(data) is
        // Write data to file.

    method readData():data is
        // Read data from file.
```

```
// The base decorator class follows the same interface as the
// other components. The primary purpose of this class is to
// define the wrapping interface for all concrete decorators.
// The default implementation of the wrapping code might include
// a field for storing a wrapped component and the means to
// initialize it.
class DataSourceDecorator implements DataSource is
    protected field wrappee: DataSource

    constructor DataSourceDecorator(source: DataSource) is
        wrappee = source

    // The base decorator simply delegates all work to the
    // wrapped component. Extra behaviors can be added in
    // concrete decorators.
    method writeData(data) is
        wrappee.writeData(data)

    // Concrete decorators may call the parent implementation of
    // the operation instead of calling the wrapped object
    // directly. This approach simplifies extension of decorator
    // classes.
    method readData():data is
        return wrappee.readData()
```

CODE

```
// Concrete decorators must call methods on the wrapped object,
// but may add something of their own to the result. Decorators
// can execute the added behavior either before or after the
// call to a wrapped object.
class EncryptionDecorator extends DataSourceDecorator is
    method writeData(data) is
        // 1. Encrypt passed data.
        // 2. Pass encrypted data to the wrappee's writeData
        // method.

    method readData():data is
        // 1. Get data from the wrappee's readData method.
        // 2. Try to decrypt it if it's encrypted.
        // 3. Return the result.

// You can wrap objects in several layers of decorators.
class CompressionDecorator extends DataSourceDecorator is
    method writeData(data) is
        // 1. Compress passed data.
        // 2. Pass compressed data to the wrappee's writeData
        // method.

    method readData():data is
        // 1. Get data from the wrappee's readData method.
        // 2. Try to decompress it if it's compressed.
        // 3. Return the result.
```

```
// Option 2. Client code that uses an external data source.
// SalaryManager objects neither know nor care about data
// storage specifics. They work with a pre-configured data
// source received from the app configurator.
class SalaryManager is
    field source: DataSource

    constructor SalaryManager(source: DataSource) { ... }

    method load() is
        return source.readData()

    method save() is
        source.writeData(salaryRecords)
    // ...Other useful methods...


// The app can assemble different stacks of decorators at
// runtime, depending on the configuration or environment.
class ApplicationConfigurator is
    method configurationExample() is
        source = new FileDataSource("salary.dat")
        if (enabledEncryption)
            source = new EncryptionDecorator(source)
        if (enabledCompression)
            source = new CompressionDecorator(source)

        logger = new SalaryManager(source)
        salary = logger.load()
    // ...
```

CODE

```
// Option 1. A simple example of a decorator assembly.
class Application is
    method dumbUsageExample() is
        source = new FileDataSource("somefile.dat")
        source.writeData(salaryRecords)
        // The target file has been written with plain data.

        source = new CompressionDecorator(source)
        source.writeData(salaryRecords)
        // The target file has been written with compressed
        // data.

        source = new EncryptionDecorator(source)
        // The source variable now contains this:
        // Encryption > Compression > FileDataSource
        source.writeData(salaryRecords)
        // The file has been written with compressed and
        // encrypted data.
```

APPLICABILITY

Use the Decorator pattern when you need to be able to assign extra behaviors to objects at runtime without breaking the code that uses these objects.

Use the pattern when it's awkward or not possible to extend an object's behavior using inheritance.

HOW TO IMPLEMENT

1. Domain Representation: Represent the domain as a primary component with optional layers.
2. Component Interface: Define common methods for both the component and layers.
3. Concrete Component: Implement the primary component with base behavior.
4. Base Decorator: Create a decorator class with a component-typed field to wrap components or other decorators, delegating work to the wrapped object.
5. Concrete Decorators: Extend the base decorator and add behavior before or after delegating to the wrapped object.
6. Client Composition: Let the client create and compose decorators as needed.

PROS & CONS

- ✓ 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.
- ✓ *Single Responsibility Principle*. You can divide a monolithic class that implements many possible variants of behavior into several smaller classes.

- ✗ 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.

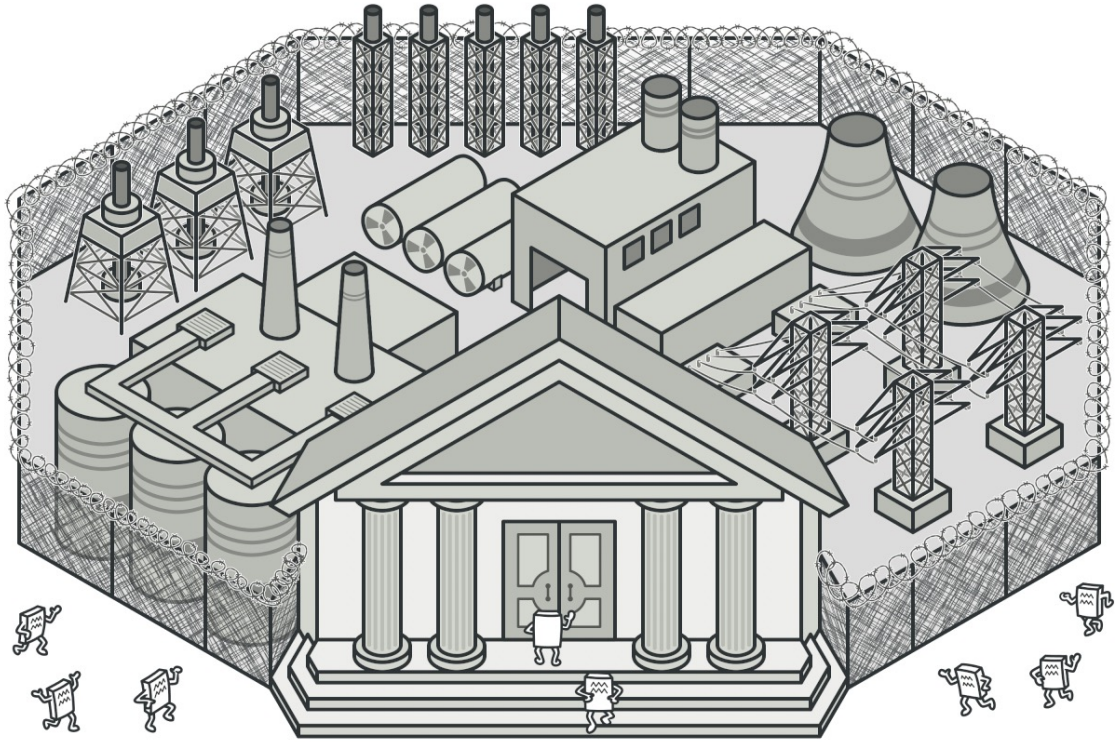
RELATIONS WITH OTHER PATTERNS

- **Adapter** provides a completely different interface for accessing an existing object. On the other hand, with the **Decorator** pattern the interface either stays the same or gets extended. In addition, *Decorator* supports recursive composition, which isn't possible when you use *Adapter*.
- With **Adapter** you access an existing object via different interface. With **Proxy**, the interface stays the same. With **Decorator** you access the object via an enhanced interface.
- **Chain of Responsibility** and **Decorator** have very similar class structures. Both patterns rely on recursive composition to pass the execution through a series of objects. However, there are several crucial differences.
- The *CoR* handlers can execute arbitrary operations independently of each other. They can also stop passing the request further at any point. On the other hand, various *Decorators* can extend the object's behavior while keeping it consistent with the base interface. In addition, decorators aren't allowed to break the flow of the request.
- **Composite** and **Decorator** have similar structure diagrams since both rely on recursive composition to organize an open-ended number of objects.
- A *Decorator* is like a *Composite* but only has one child component. There's another significant difference: *Decorator* adds additional responsibilities to the wrapped object, while *Composite*just "sums up" its children's results.
- However, the patterns can also cooperate: you can use *Decorator* to extend the behavior of a specific object in the *Composite* tree.
- Designs that make heavy use of **Composite** and **Decorator** can often benefit from using **Prototype**. Applying the pattern lets you clone complex structures instead of re-constructing them from scratch.
- **Decorator** lets you change the skin of an object, while **Strategy** lets you change the guts.
- **Decorator** and **Proxy** have similar structures, but very different intents. Both patterns are built on the composition principle, where one object is supposed to delegate some of the work to another. The difference is that a *Proxy* usually manages the life cycle of its service object on its own, whereas the composition of *Decorators* is always controlled by the client.

MODELLI STRUTTURALI:
FACADE PATTERN

INTENT

- **Facade** is a structural design pattern that provides a simplified interface to a library, a framework, or any other complex set of classes.



PROBLEM

- Imagine that you must make your code work with a broad set of objects that belong to a sophisticated library or framework. Ordinarily, you'd need to initialize all of those objects, keep track of dependencies, execute methods in the correct order, and so on.
- As a result, the business logic of your classes would become tightly coupled to the implementation details of 3rd-party classes, making it hard to comprehend and maintain.

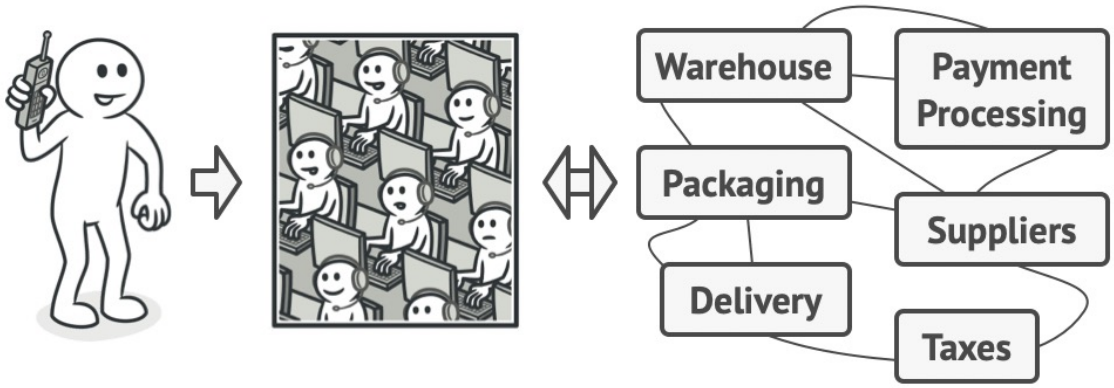
SOLUTION

- A facade is a class that provides a simple interface to a complex subsystem which contains lots of moving parts. A facade might provide limited functionality in comparison to working with the subsystem directly. However, it includes only those features that clients really care about.
- Having a facade is handy when you need to integrate your app with a sophisticated library that has dozens of features, but you just need a tiny bit of its functionality.

- For instance, an app that uploads short funny videos with cats to social media could potentially use a professional video conversion library. However, all that it really needs is a class with the single method encode(filename, format). After creating such a class and connecting it with the video conversion library, you'll have your first facade.

REAL-WORLD ANALOGY

- When you call a shop to place a phone order, an operator is your facade to all services and departments of the shop. The operator provides you with a simple voice interface to the ordering system, payment gateways, and various delivery services.

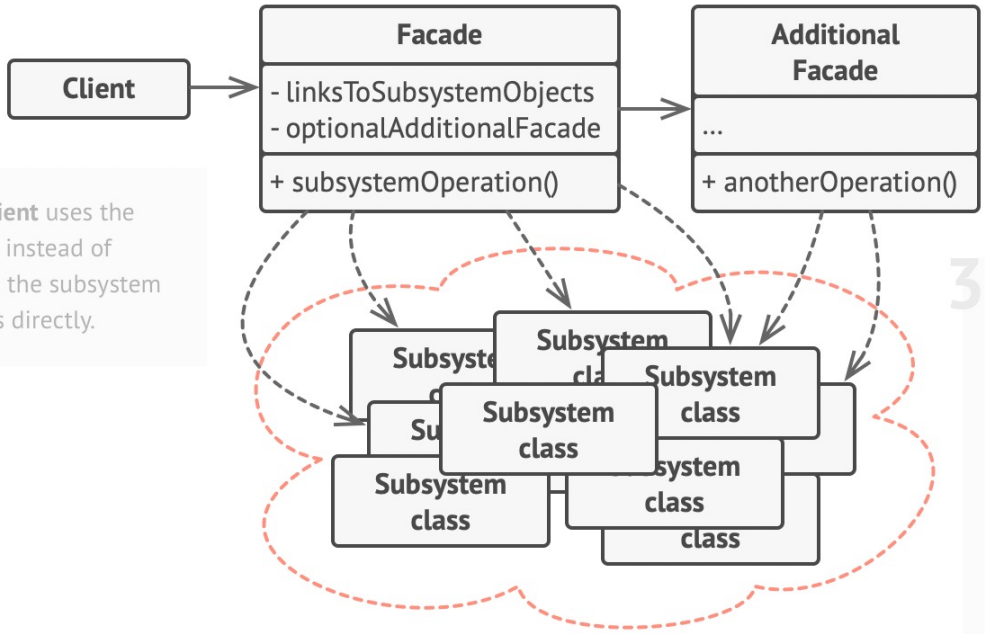


Placing orders by phone.

STRUCTURE

1 The **Facade** provides convenient access to a particular part of the subsystem's functionality. It knows where to direct the client's request and how to operate all the moving parts.

2 An **Additional Facade** class can be created to prevent polluting a single facade with unrelated features that might make it yet another complex structure. Additional facades can be used by both clients and other facades.



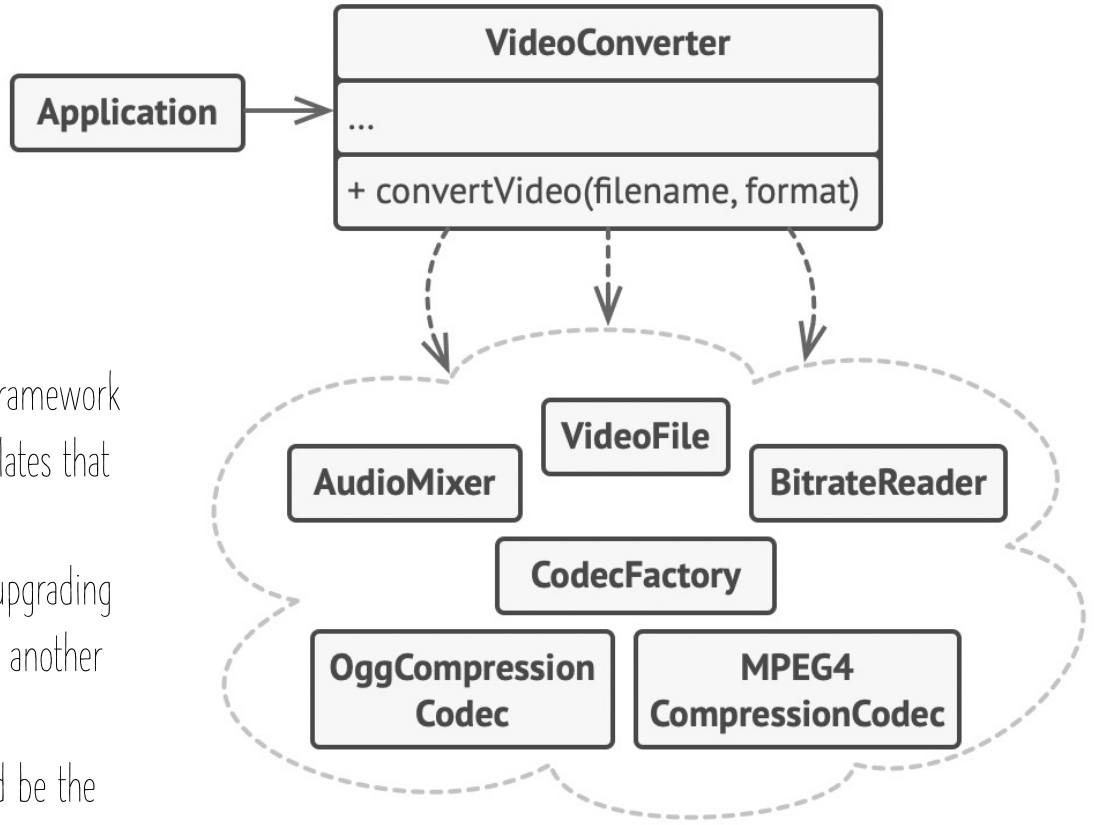
4 The **Client** uses the facade instead of calling the subsystem objects directly.

3 The **Complex Subsystem** consists of dozens of various objects. To make them all do something meaningful, you have to dive deep into the subsystem's implementation details, such as initializing objects in the correct order and supplying them with data in the proper format.

Subsystem classes aren't aware of the facade's existence. They operate within the system and work with each other directly.

#PSEUDOCODE

- Instead of making your code work with dozens of the framework classes directly, you create a facade class which encapsulates that functionality and hides it from the rest of the code.
- This structure also helps you to minimize the effort of upgrading to future versions of the framework or replacing it with another one.
- The only thing you'd need to change in your app would be the implementation of the facade's methods.



An example of isolating multiple dependencies within a single facade class.

CODE

```
// These are some of the classes of a complex 3rd-party video
// conversion framework. We don't control that code, therefore
// can't simplify it.

class VideoFile
// ...

class OggCompressionCodec
// ...

class MPEG4CompressionCodec
// ...

class CodecFactory
// ...

class BitrateReader
// ...

class AudioMixer
// ...
```

```
// We create a facade class to hide the framework's complexity
// behind a simple interface. It's a trade-off between
// functionality and simplicity.
class VideoConverter is
    method convert(filename, format):File is
        file = new VideoFile(filename)
        sourceCodec = (new CodecFactory).extract(file)
        if (format == "mp4")
            destinationCodec = new MPEG4CompressionCodec()
        else
            destinationCodec = new OggCompressionCodec()
        buffer = BitrateReader.read(filename, sourceCodec)
        result = BitrateReader.convert(buffer, destinationCodec)
        result = (new AudioMixer()).fix(result)
        return new File(result)
```

```
// Application classes don't depend on a billion classes
// provided by the complex framework. Also, if you decide to
// switch frameworks, you only need to rewrite the facade class.
class Application is
    method main() is
        convertor = new VideoConverter()
        mp4 = convertor.convert("funny-cats-video.ogg", "mp4")
        mp4.save()
```

APPLICABILITY

Use the Facade pattern when you need to have a limited but straightforward interface to a complex subsystem.

Use the Facade when you want to structure a subsystem into layers. Create facades to define entry points to each level of a subsystem. You can reduce coupling between multiple subsystems by requiring them to communicate only through facades.

HOW TO IMPLEMENT

1. Simplify Interface: Create a simpler interface to decouple client code from subsystem classes.
2. Facade Class: Implement the interface in a new facade that redirects client calls to subsystem objects and manages the subsystem's lifecycle if needed.
3. Exclusive Access: Ensure all client communication with the subsystem goes through the facade to shield clients from subsystem changes.
4. Refine if Needed: Split the facade into smaller classes if it becomes too complex.

PROD & CONS

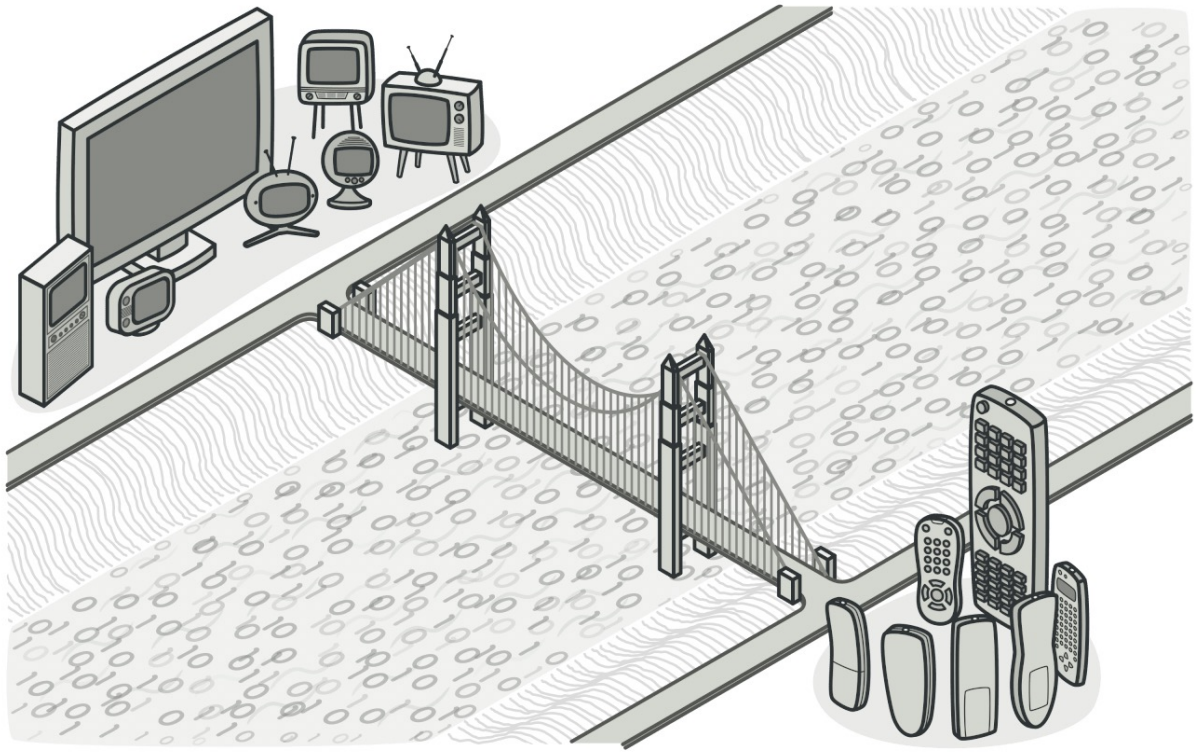
You can isolate your code from the complexity of a subsystem.

A facade can become a god objectcoupled to all classes of an app.

MODELLI STRUTTURALI:
BRIDGE PATTERN

INTENT

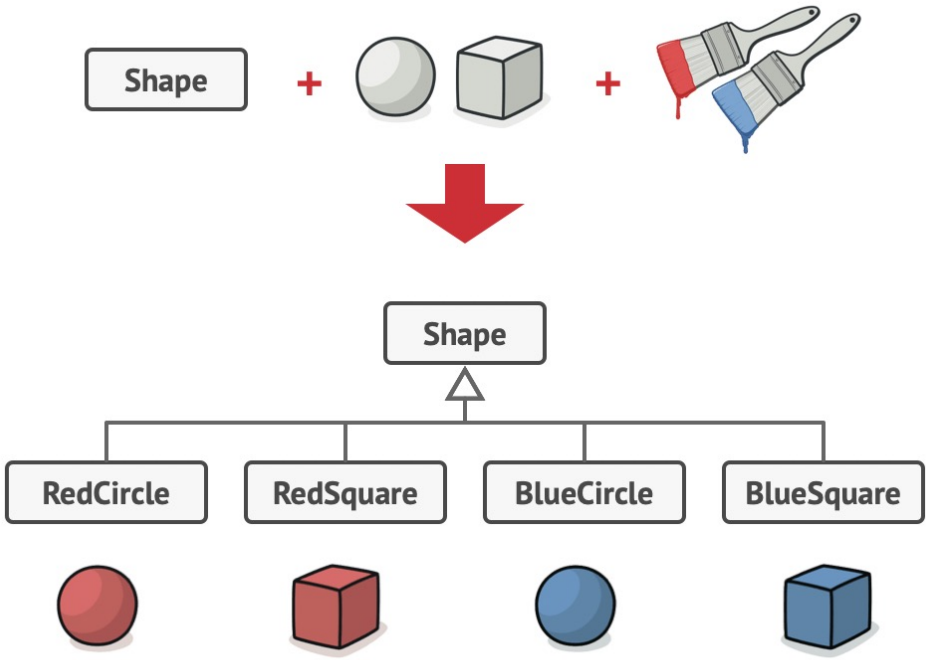
- Bridge is a structural design pattern that lets you split a large class or a set of closely related classes into two separate hierarchies–abstraction and implementation–which can be developed independently of each other.



PROBLEM

Abstraction? Implementation? Sound scary? Stay calm and let's consider a simple example.

- Say you have a geometric Shape class with a pair of subclasses: Circle and Square. You want to extend this class hierarchy to incorporate colors, so you plan to create Red and Blue shape subclasses. However, since you already have two subclasses, you'll need to create four class combinations such as BlueCircle and RedSquare.
- Adding new shape types and colors to the hierarchy will grow it exponentially.

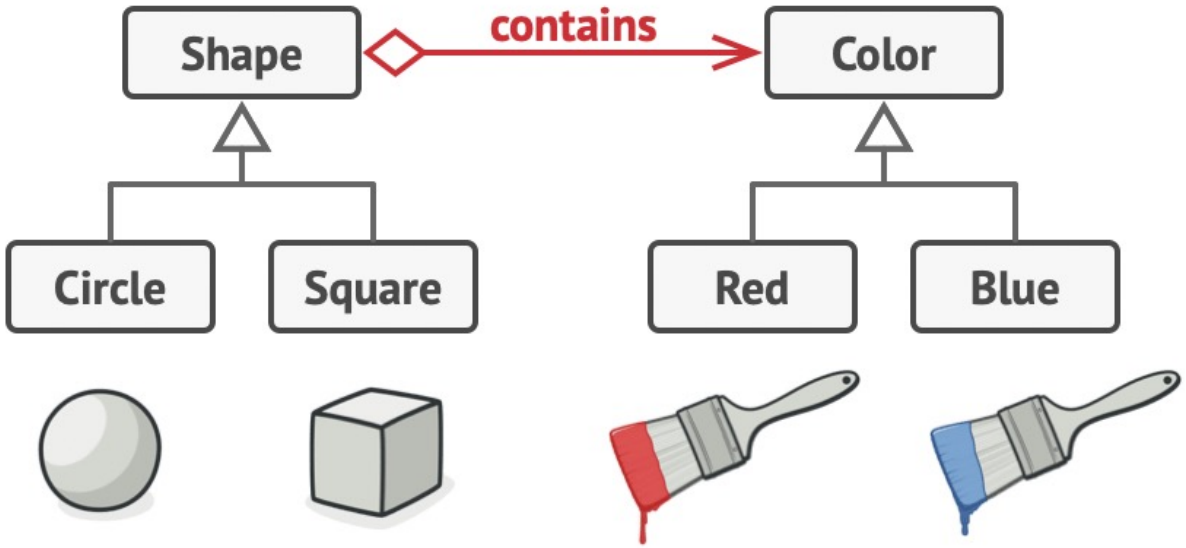


Number of class combinations grows in geometric progression.

This problem occurs because we're trying to extend the shape classes in two independent dimensions: by form and by color. That's a very common issue with class inheritance.

SOLUTION

- The Bridge pattern attempts to solve this problem by switching from inheritance to the object composition. What this means is that you extract one of the dimensions into a separate class hierarchy, so that the original classes will reference an object of the new hierarchy, instead of having all of its state and behaviors within one class.



You can prevent the explosion of a class hierarchy by transforming it into several related hierarchies.

ABSTACTION AND IMPLEMENTATION

- Abstraction (also called interface) is a high-level control layer for some entity.
- This layer isn't supposed to do any real work on its own. It should delegate the work to the implementation layer (also called platform).

- *Note that we're not talking about interfaces or abstract classes from your programming language. These aren't the same things.*

PROBLEM

Generally speaking, you can extend such an app in two independent directions:

- Have several different GUIs (for instance, tailored for regular customers or admins).
- Support several different APIs (for example, to be able to launch the app under Windows, Linux, and macOS).

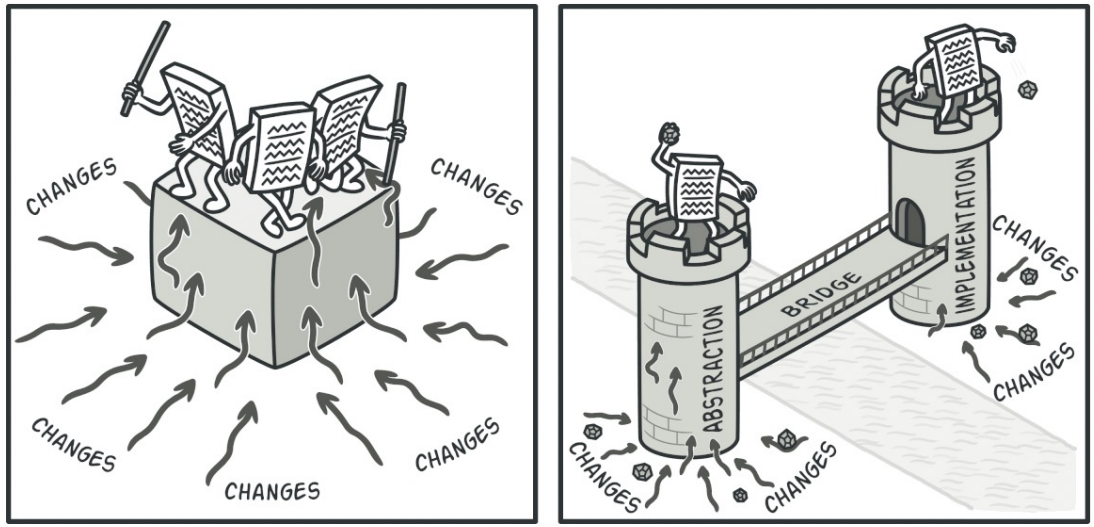
In a worst-case scenario, this app might look like a giant spaghetti bowl, where hundreds of conditionals connect different types of GUI with various APIs all over the code.

- You can bring order to this chaos by extracting the code related to specific interface-platform combinations into separate classes. However, soon you'll discover that there are *lots* of these classes.

The class hierarchy will grow exponentially because adding a new GUI or supporting a different API require creating more and more classes.

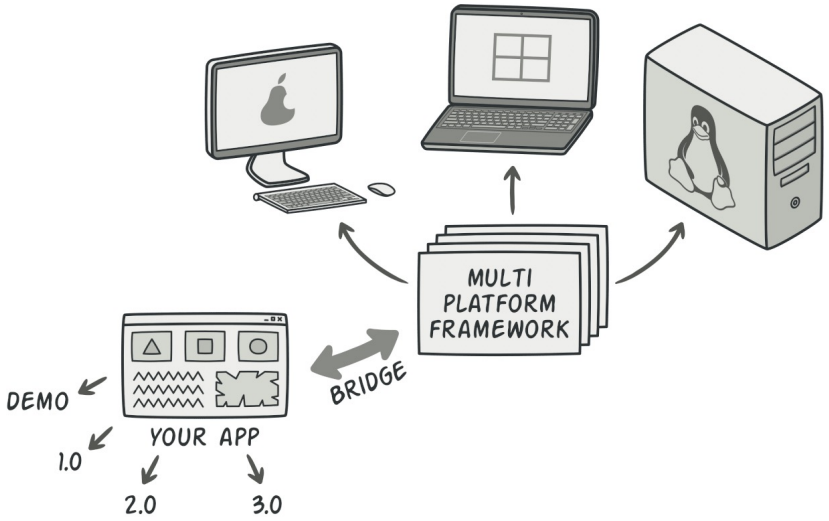
SOLUTION

- Let's try to solve this issue with the Bridge pattern. It suggests that we divide the classes into two hierarchies:
- Abstraction: the GUI layer of the app.
- Implementation: the operating systems' APIs.



Making even a simple change to a monolithic codebase is pretty hard because you must understand the entire thing very well. Making changes to smaller, well-defined modules is much easier.

SOLUTION



One of the ways to structure a cross-platform application.

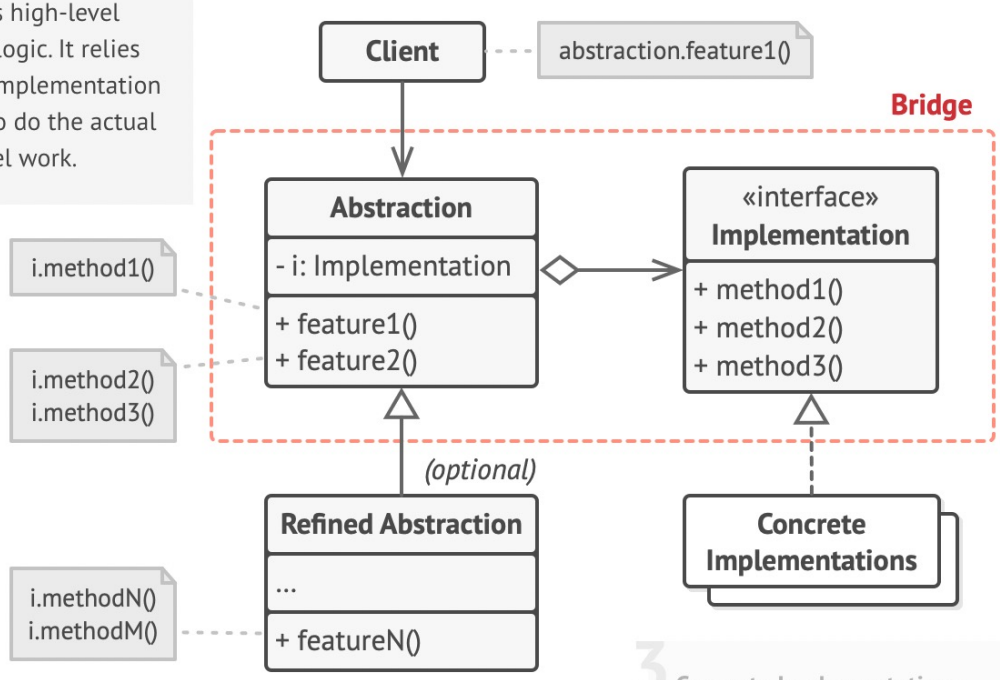
The abstraction object controls the appearance of the app, delegating the actual work to the linked implementation object.

- Different implementations are interchangeable as long as they follow a common interface, enabling the same GUI to work under Windows and Linux.
- As a result, you can change the GUI classes without touching the API-related classes. Moreover, adding support for another operating system only requires creating a subclass in the implementation hierarchy.

STRUCTURE

1 The **Abstraction** provides high-level control logic. It relies on the implementation object to do the actual low-level work.

5 Usually, the **Client** is only interested in working with the abstraction. However, it's the client's job to link the abstraction object with one of the implementation objects.



2 The **Implementation** declares the interface that's common for all concrete implementations. An abstraction can only communicate with an implementation object via methods that are declared here.

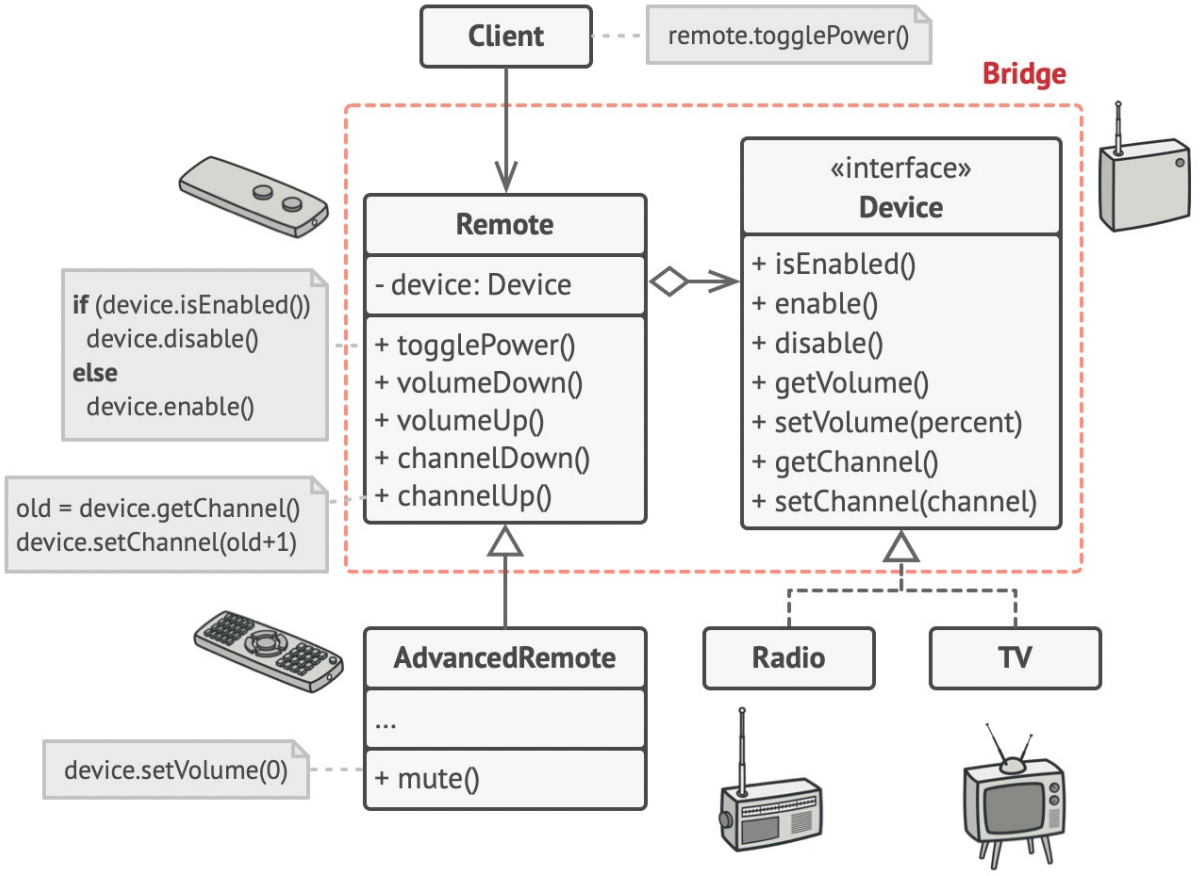
The abstraction may list the same methods as the implementation, but usually the abstraction declares some complex behaviors that rely on a wide variety of primitive operations declared by the implementation.

3 **Concrete Implementations** contain platform-specific code.

4 **Refined Abstractions** provide variants of control logic. Like their parent, they work with different implementations via the general implementation interface.

#PSEUDOCODE

- This example illustrates how the Bridge pattern can help divide the monolithic code of an app that manages devices and their remote controls. The Device classes act as the implementation, whereas the Remotes act as the abstraction.



The original class hierarchy is divided into two parts: devices and remote controls.

CODE

```
// All devices follow the same interface.
class Tv implements Device is
    // ...

class Radio implements Device is
    // ...


// Somewhere in client code.
tv = new Tv()
remote = new RemoteControl(tv)
remote.togglePower()

radio = new Radio()
remote = new AdvancedRemoteControl(radio)
```

```
// The "abstraction" defines the interface for the "control"
// part of the two class hierarchies. It maintains a reference
// to an object of the "implementation" hierarchy and delegates
// all of the real work to this object.
class RemoteControl is
    protected field device: Device
    constructor RemoteControl(device: Device) is
        this.device = device
    method togglePower() is
        if (device.isEnabled()) then
            device.disable()
        else
            device.enable()
    method volumeDown() is
        device.setVolume(device.getVolume() - 10)
    method volumeUp() is
        device.setVolume(device.getVolume() + 10)
    method channelDown() is
        device.setChannel(device.getChannel() - 1)
    method channelUp() is
        device.setChannel(device.getChannel() + 1)
```

```
// You can extend classes from the abstraction hierarchy
// independently from device classes.
class AdvancedRemoteControl extends RemoteControl is
    method mute() is
        device.setVolume(0)


// The "implementation" interface declares methods common to all
// concrete implementation classes. It doesn't have to match the
// abstraction's interface. In fact, the two interfaces can be
// entirely different. Typically the implementation interface
// provides only primitive operations, while the abstraction
// defines higher-level operations based on those primitives.
interface Device is
    method isEnabled()
    method enable()
    method disable()
    method getVolume()
    method setVolume(percent)
    method getChannel()
    method setChannel(channel)
```

APPLICABILITY

Use the Bridge pattern when you want to divide and organize a monolithic class that has several variants of some functionality (for example, if the class can work with various database servers).

Use the pattern when you need to extend a class in several orthogonal (independent) dimensions.

Use the Bridge if you need to be able to switch implementations at runtime.

HOW TO IMPLEMENT

1. Identify Dimensions: Break classes into orthogonal concepts like abstraction/platform or interface/implementation.
2. Base Abstraction: Define client-required operations in the base abstraction class.
3. Implementation Interface: Declare platform-wide operations needed by the abstraction.
4. Concrete Implementations: Create platform-specific classes adhering to the implementation interface.
5. Delegate Work: Add an implementation reference in the abstraction and delegate tasks to it.
6. Refined Abstractions: Extend the base abstraction for different logic variants.
7. Client Interaction: Pass the implementation to the abstraction's constructor, allowing the client to interact only with the abstraction.

PROS & CONS

You can create platform-independent classes and apps.

The client code works with high-level abstractions. It isn't exposed to the platform details.

Open/Closed Principle. You can introduce new abstractions and implementations independently from each other.

Single Responsibility Principle. You can focus on high-level logic in the abstraction and on platform details in the implementation.

You might make the code more complicated by applying the pattern to a highly cohesive class.