Topic 6 - Structural Design Patterns (Gamma et. al)

Structural Patterns

- Structural patterns are concerned with how classes and objects are composed to form larger structures.
- Structural class patterns combines two or more classes. For example, the class form of the Adapter pattern derives a new class from two or more base classes.
- Structural object patterns describe ways to compose objects to realize new functionality.
- Structural patterns covered are:
  - Composite
  - Decorator
  - Flyweight
  - Proxy
  - Adapter
  - Bridge
  - Facade

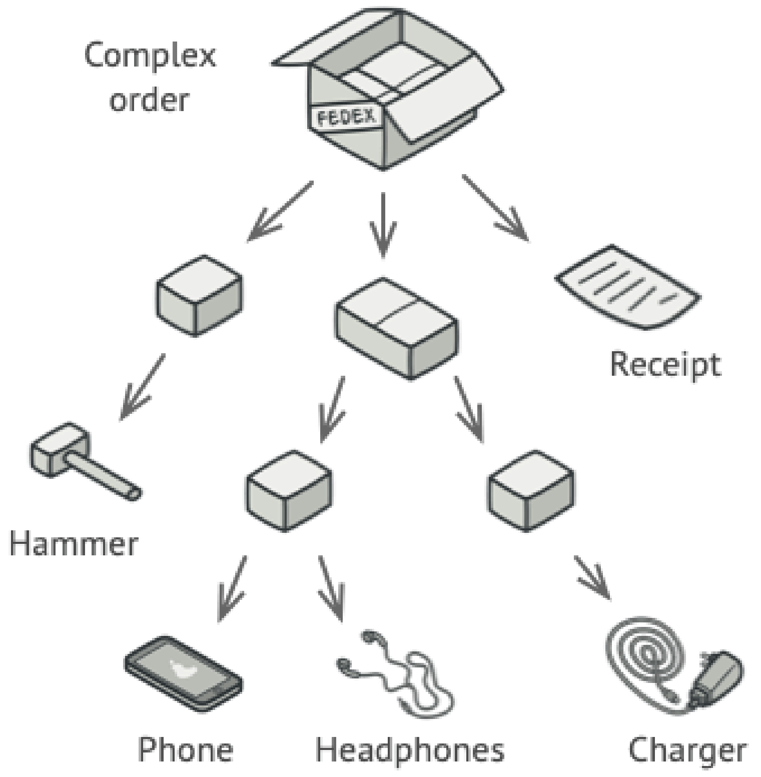
Composite Pattern

Intent

- Compose objects into tree structures to represent part-whole or containment hierarchies
- Composite pattern lets clients treat individual objects and composition of objects uniformly

Motivation

- Building complex objects can be made simpler if an abstract class or interface can be used to represent both primitives and containers

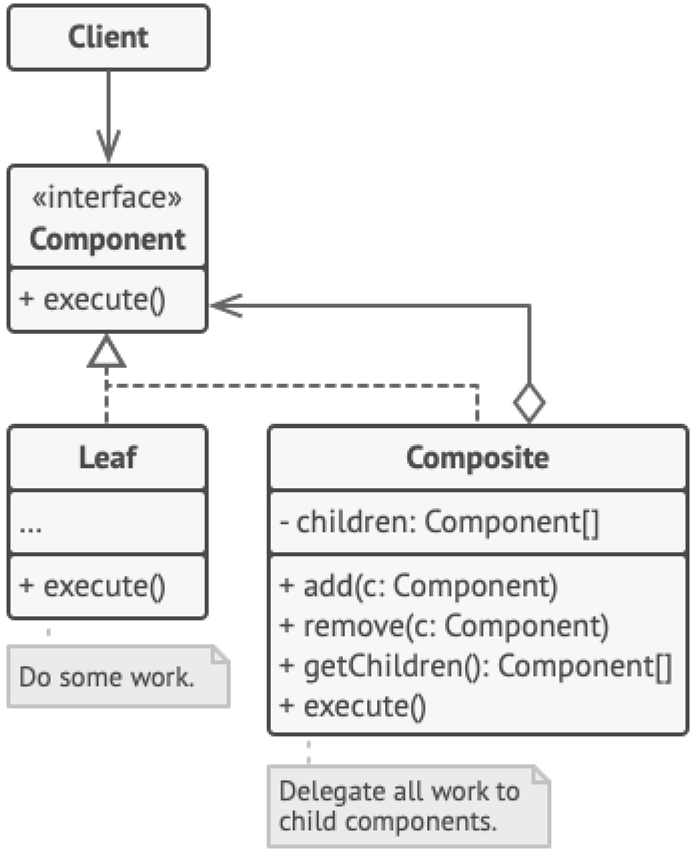


https://refactoring.guru/design-patterns/composite

Composite Example

https://refactoring.guru/design-patterns/composite

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.



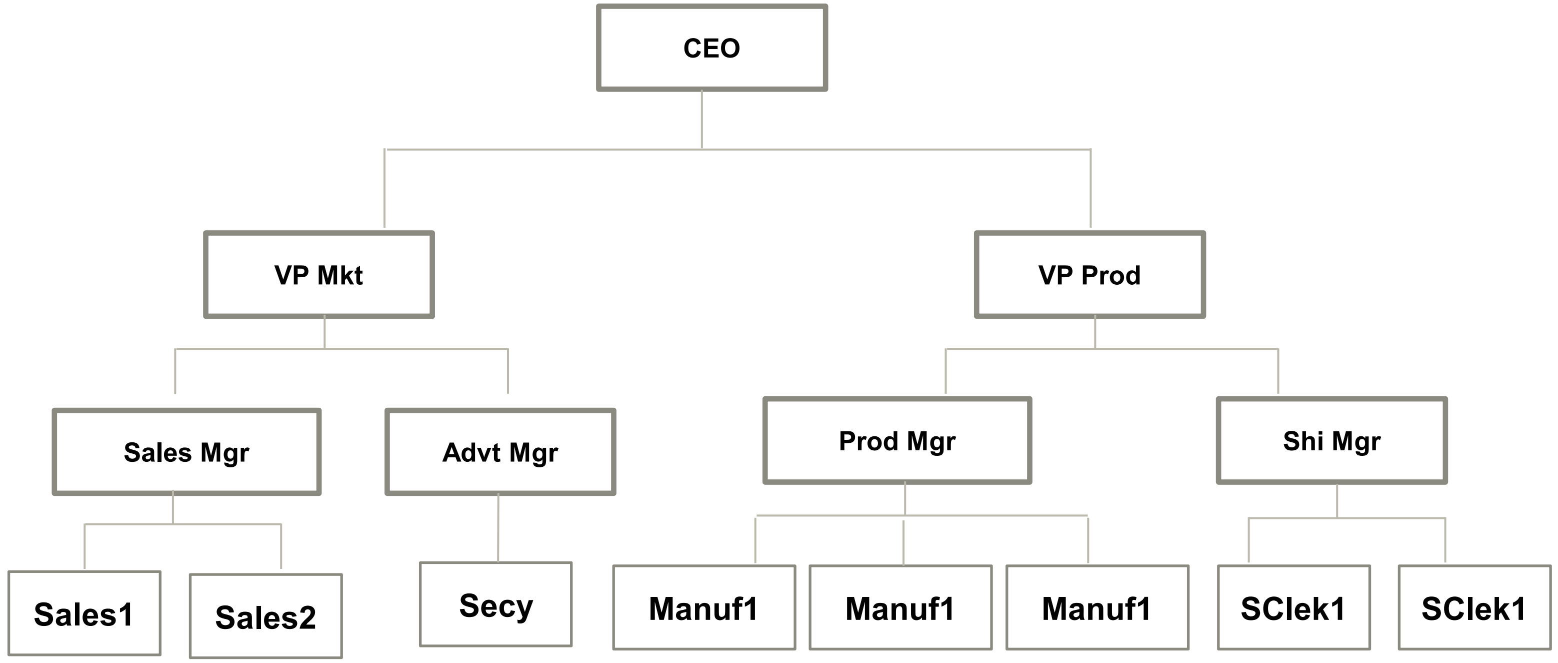
The **Component** interface describes operations that are common to both simple and complex elements of the tree.

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.

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.

Practical Example of a Composite



- Files/Folders/Subfolders are an obvious application of Composite
- See `java.awt.Component`/`java.awt.Container` and `javax.swing.JMenuItem`/`javax.swing.JMenu` for Java API examples

Composite: Consequences

- Defines class hierarchies consisting of primitive objects and composite objects. Complex objects can be created by recursively composing
- Makes the client code simple (without if statements or type checking) since composite structures and primitives can be treated uniformly
- Decisions must be made about where to place operations, especially add/remove, get children/parent etc.
  – see next slide
- placing them in superclass maximises generality/uniformity but reduces safety since what does adding to a non composite leaf actually mean?
- Also an overly general design makes it difficult to restrict the type of components that can be added to any given Composite since the interface specifies that a Composite can contain any Component
  – We can deal with this at the subclass levels using checking and exceptions, boolean return values etc.

Composite: Method placement

- A general principle of inheritance is that subclasses should not inherit methods that they do not need (see Fowler Refused Bequest!)
- According to this principle we would not expect a component to have getChildren, getParent methods etc.
  - rather these would only be methods of Composite
- However we can also view a component/leaf as a special case component with NO children
  - therefore a getChildren method that returns for example an empty List or Set is still semantically meaningful
  - this improves uniformity since the client can always treat Components and Composites in the same way
  - this approach is taken in the provided tree example

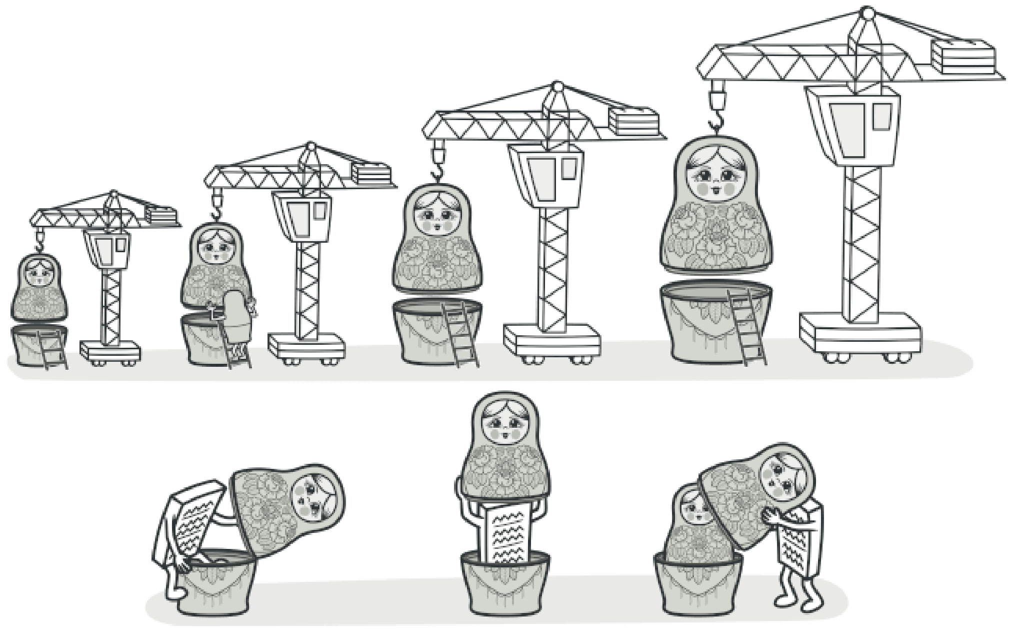
Decorator

Intent

- Attach additional responsibilities to an object dynamically
- Provide a flexible alternative to subclassing for extension
- Object appears to change its dynamic class/type at runtime when it is decorated
- Provides runtime behaviour change without destroying existing and creating new instances
- Decorator does something then forwards to decorated object (just as a subclass can forward to super())

Applicability

- Use to add responsibilities to individual objects dynamically and transparently
- Useful when you have many combinations of behaviour and class hierarchy would get bloated (combinatorial explosion) if subclassing alone is used



https://refactoring.guru/design-patterns/decorator

Decorator Structure

https://refactoring.guru/design-patterns/decorator

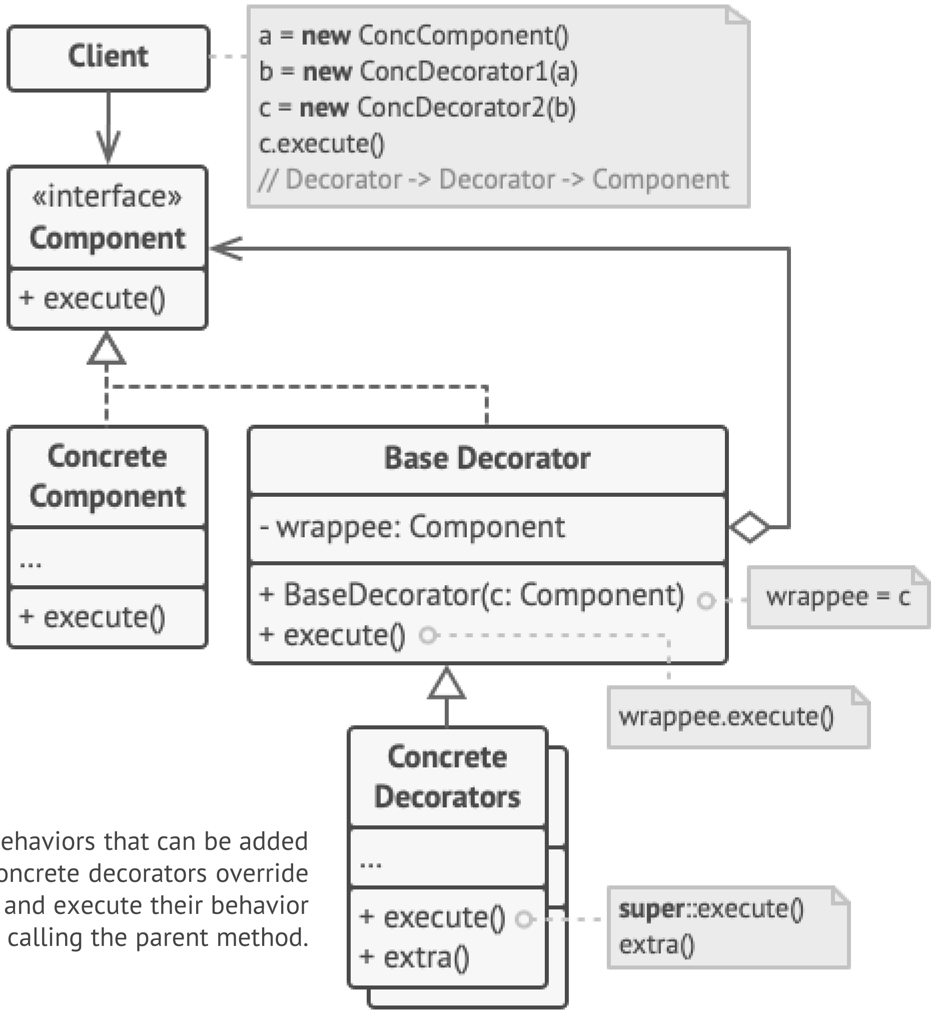
The **Client** can wrap components in multiple layers of decorators, as long as it works with all objects via the component interface.

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

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

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.

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.

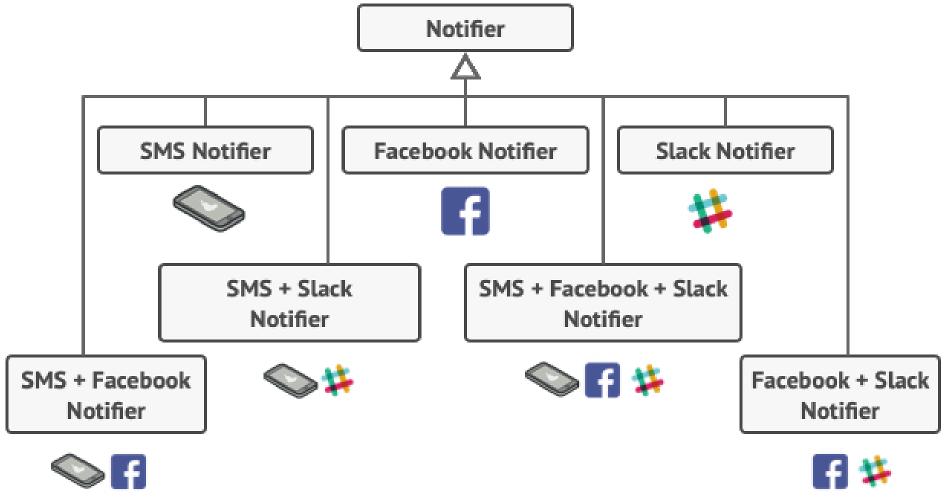


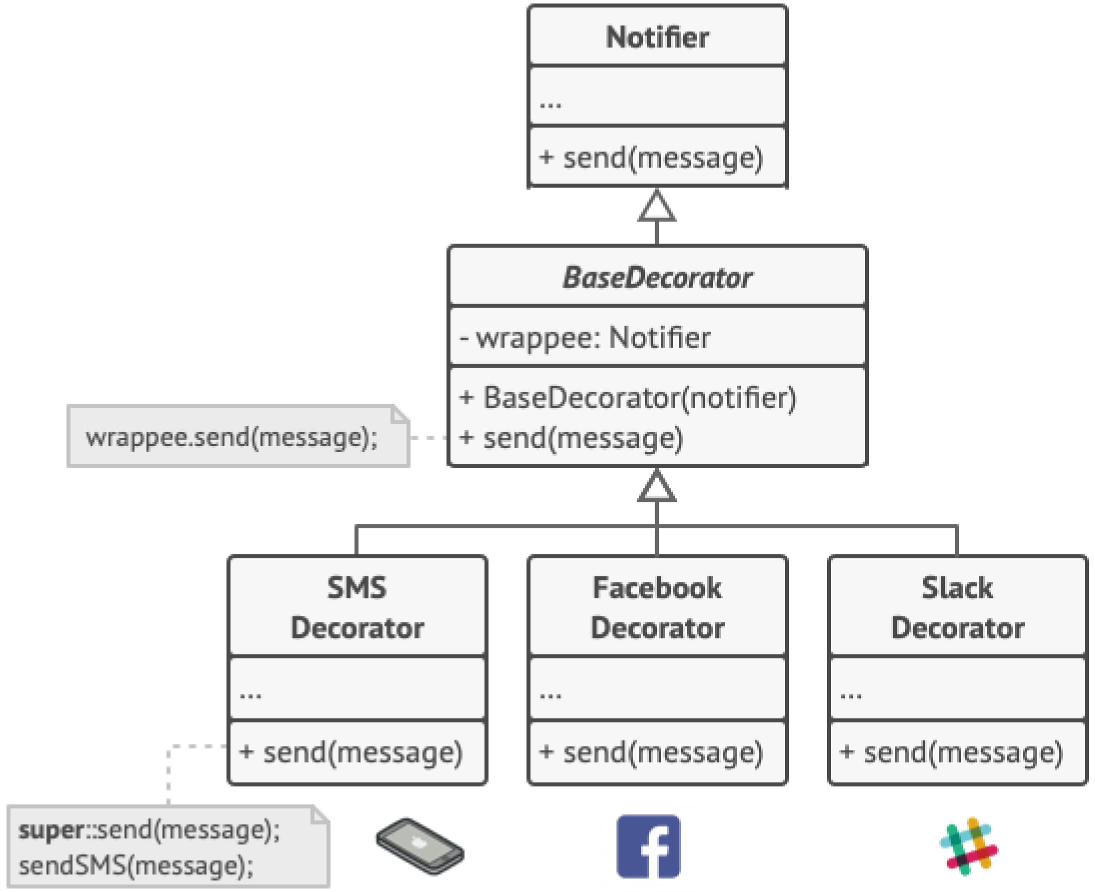
```
a = new ConcComponent()
b = new ConcDecorator1(a)
c = new ConcDecorator2(b)
c.execute()
// Decorator -> Decorator -> Component
```

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

Decorator Example

https://refactoring.guru/design-patterns/decorator





Consequences

- More flexible than inheritance since can be added and removed at runtime without copying or recreating target object
- Avoids feature-heavy classes with many unused features (80/20 rule i.e. Pareto Principle)
- Memory overhead of more runtime objects may be an issue
  - especially if decorators inherit state they don't need
- Complexity (especially when debugging) may be an issue

- *QUESTION: Think about how Decorator and Composite work together? e.g. if you Decorate a Composite?*

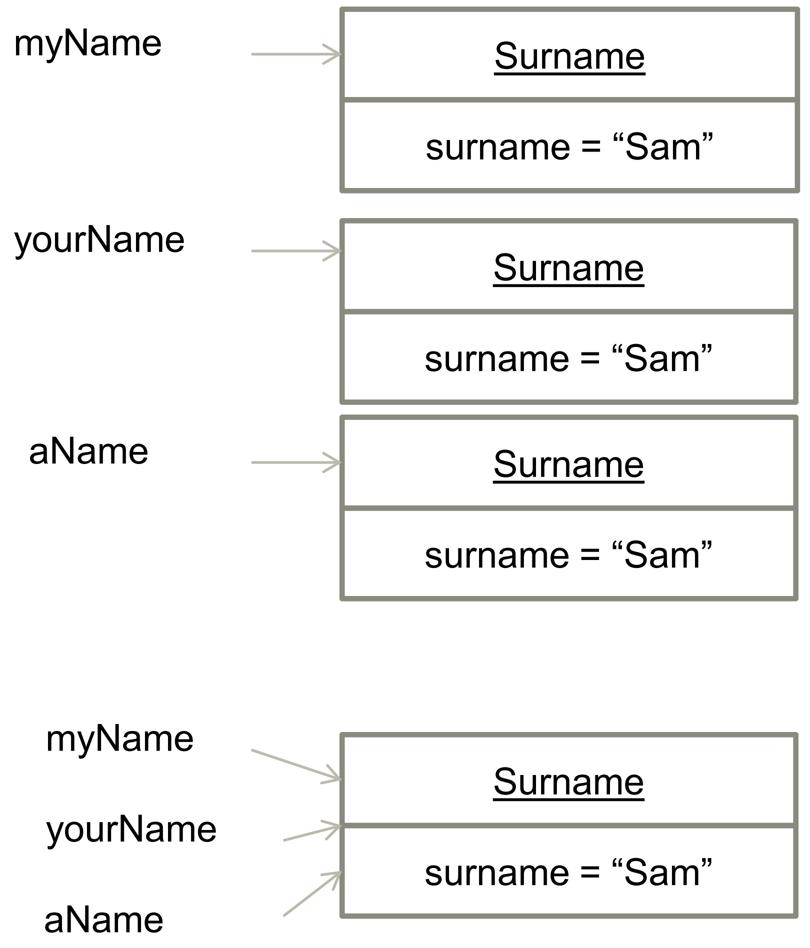
Flyweight

Intent

- The intent of this pattern is to use sharing to support a large number of objects

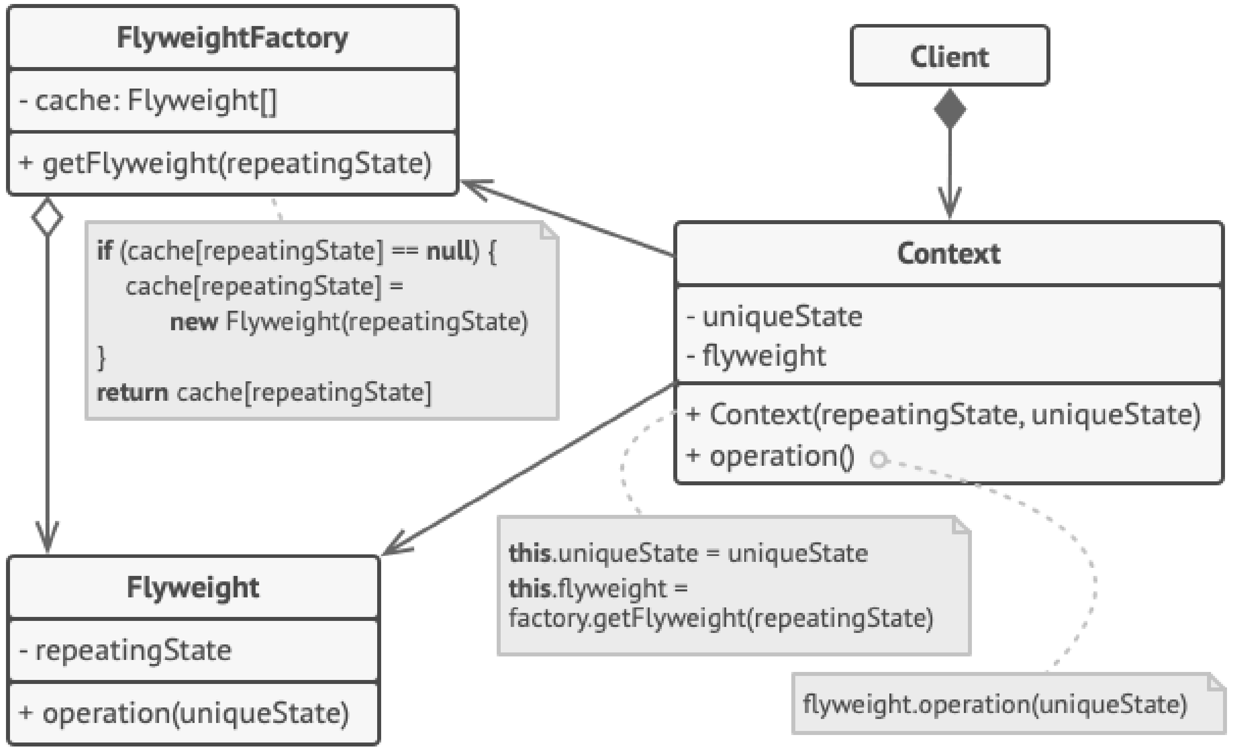
Motivation

- Objects provide useful abstraction however sometimes they may be prohibitive due to number of instances
- e.g. consider a character Class/Object in a large document
- Flyweight pattern allows similar objects to be shared by multiple clients.
- Consider a class used for representing surnames in a phone book



Flyweight

- Flyweights do not have to be 100% shared. The non-shared (***extrinsic***) state can be passed as a parameter to Flyweight methods. For example when a character/glyph is to be printed the position (x,y) can be passed as an argument.
- ConcreteFlyweight Implements the Flyweight interface and stores the shared (***intrinsic***) state
- The Concrete flyweight object must maintain state that it is intrinsic to it, and must be able to manipulate state that is extrinsic.
- **FlyweightFactory** maintains a pool of different flyweight objects and returns the reference of an object from the pool if it is already created. Otherwise it creates and adds one to the pool and returns it's reference.
- Client uses a FlyweightFactory to get a Flyweight (constructor can be private)



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

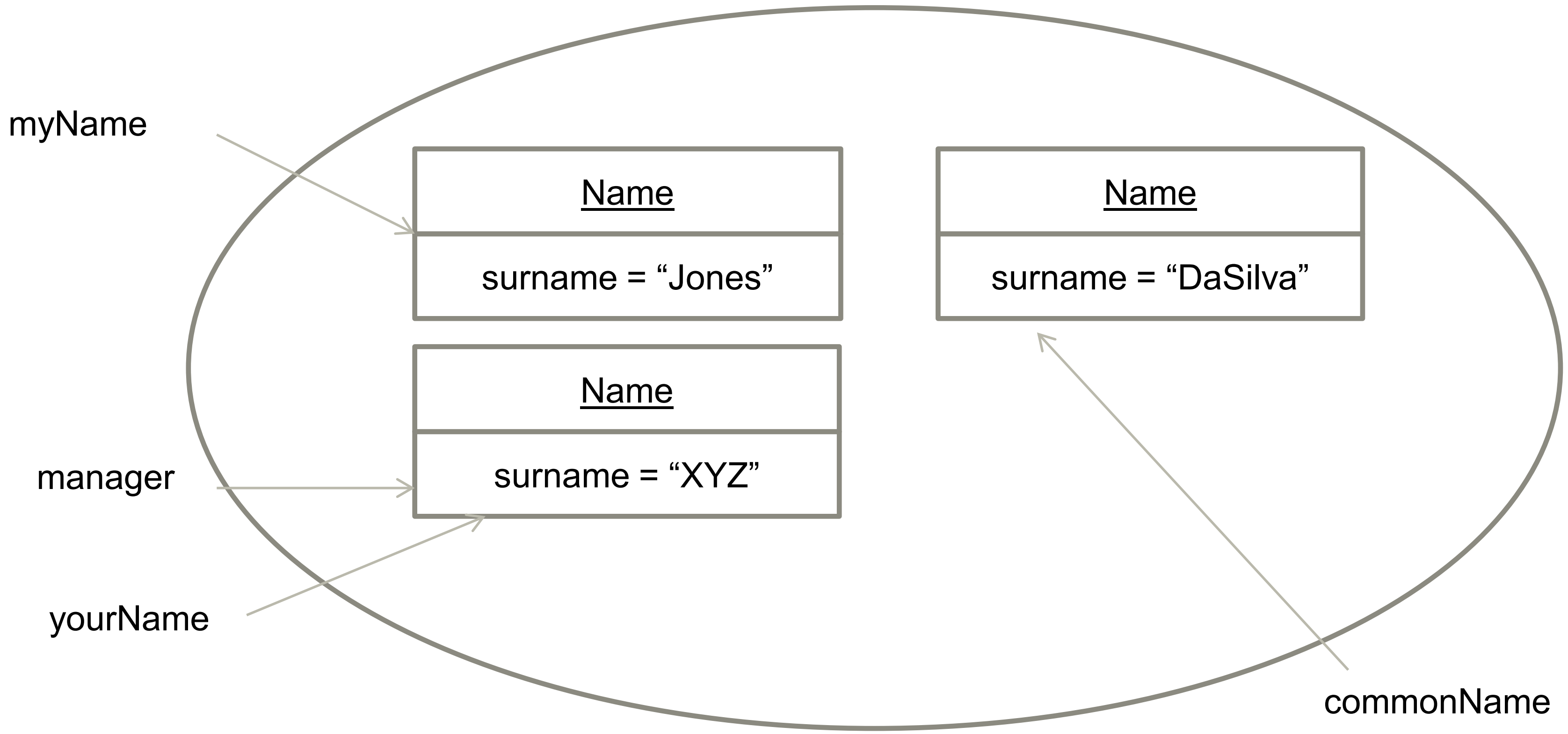
```
this.uniqueState = uniqueState
this.flyweight =
factory.getFlyweight(repeatingState)
```

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

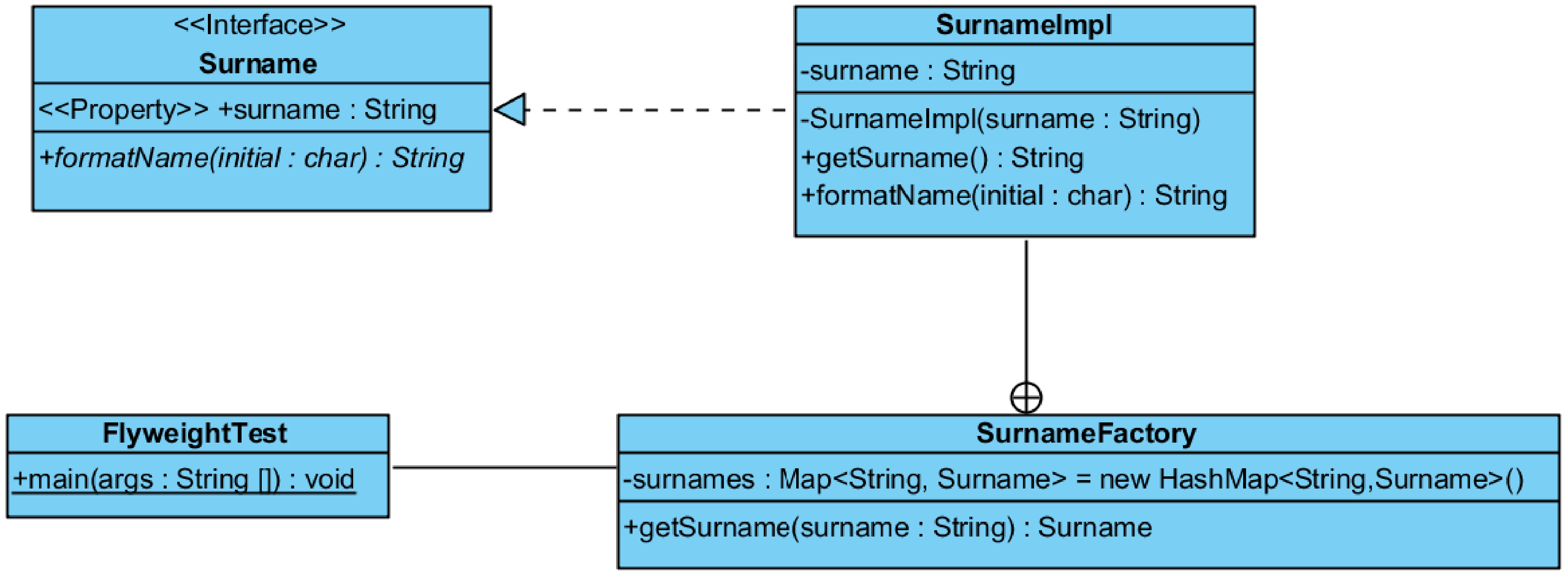
https://refactoring.guru/design-patterns/flyweight

Flyweight Example

To ensure duplicate instances are not created SurnameFactory maintains a pool of instances



Code Example



<<Interface>>
Surname
<<Property>> +surname : String
+formatName(initial : char) : String
SurnameImpl
-surname : String
-SurnameImpl(surname : String)
+getSurname() : String
+formatName(initial : char) : String
FlyweightTest
+main(args : String []) : void
SurnameFactory
-surnames : Map<String, Surname> = new HashMap<String,Surname>()
+getSurname(surname : String) : Surname

`java.lang.Integer.valueOf(int)` uses the flyweight pattern to cache 8 bit ints and also the `Integer` constructor is deprecated in later version of Java (e.g. JDK 11)

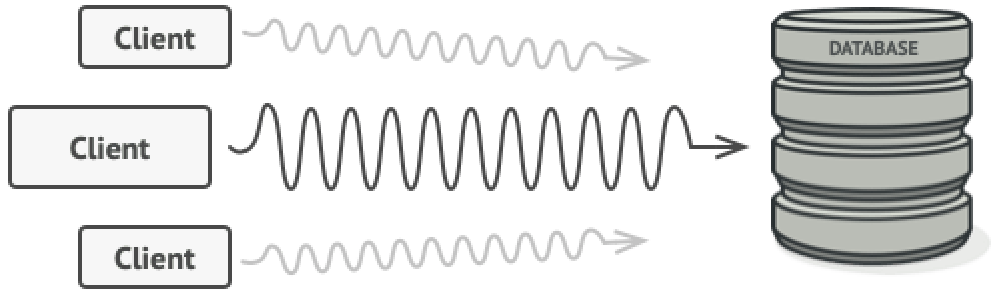
see `flyweight.name.*` example in OOSD source code

Flyweight: Consequences

- May introduce run-time costs associated with finding, computing and transferring extrinsic state
- Such costs however are offset by storage/memory savings which increases as more flyweights are shared
- Is only useful where there is significant intrinsic state and a small number of combinations compared to total instances i.e. only a few hundred different character shapes/glyphs but many thousands of characters in the document
- Especially useful when extrinsic state can be efficiently computed rather than stored
- Is the previous first name example a good practical example of Flyweight?
- Flyweight pattern may be combined with Composite pattern to represent a hierarchical structure
  - has implications for parent relationships which must be extrinsic (why?)

Proxy Pattern

https://refactoring.guru/design-patterns/proxy



- **Intent**
  - To provide a surrogate or placeholder for another object and to control access to it.

- **Motivation**
  - A **remote proxy** provides a local representative for an object in a different address space providing interface for remote resources such as web service or REST resources.
  - A **virtual proxy** creates expensive objects on demand (lazy instantiation) and may pool or cache instances.
  - A **protection proxy** controls access to the original object. Protection proxies are useful when objects should have different access rights.
  - **Thread safety** useful if not all access needs to be thread safe. Use a proxy when necessary but more efficient with direct access when not required.
  - Differs from factories in that you usually have one proxy for each instance unless you are doing sharing or pooling

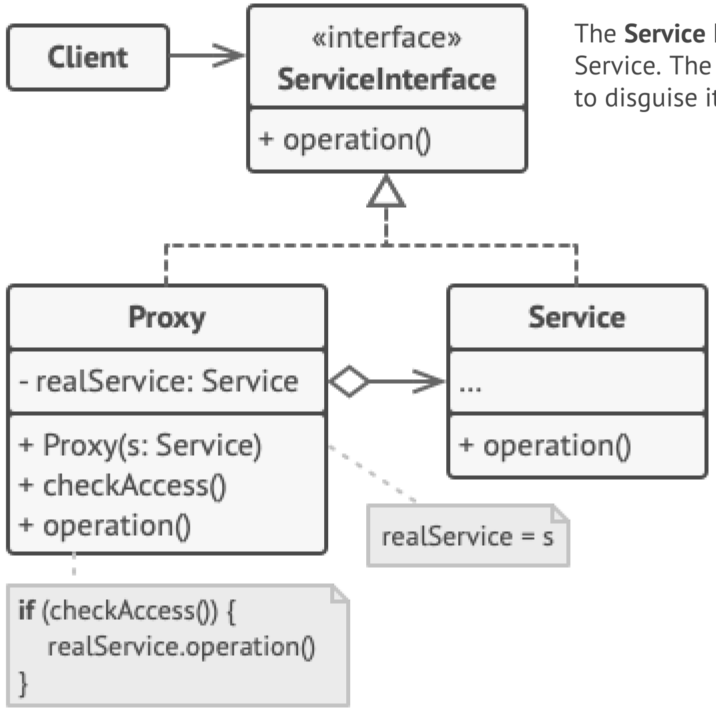
Elements

- **Proxy**
  - Maintains a reference that lets the proxy access the real subject.
  - Provides an interface identical to Subject's so that proxy can be substituted for the real subject.
  - Controls access to the real subject and may be responsible for creating and deleting it.
- **Subject**
  - Defines the common interface for RealSubject and proxy so that a Proxy can be used anywhere a RealSubject is expected.
- **RealSubject**
  - Defines the real object that proxy represents.

Structure

https://refactoring.guru/design-patterns/proxy

The **Client** should work with both services and proxies via the same interface. This way you can pass a proxy into any code that expects a service object.



The **Service Interface** declares the interface of the Service. The proxy must follow this interface to be able to disguise itself as a service object.

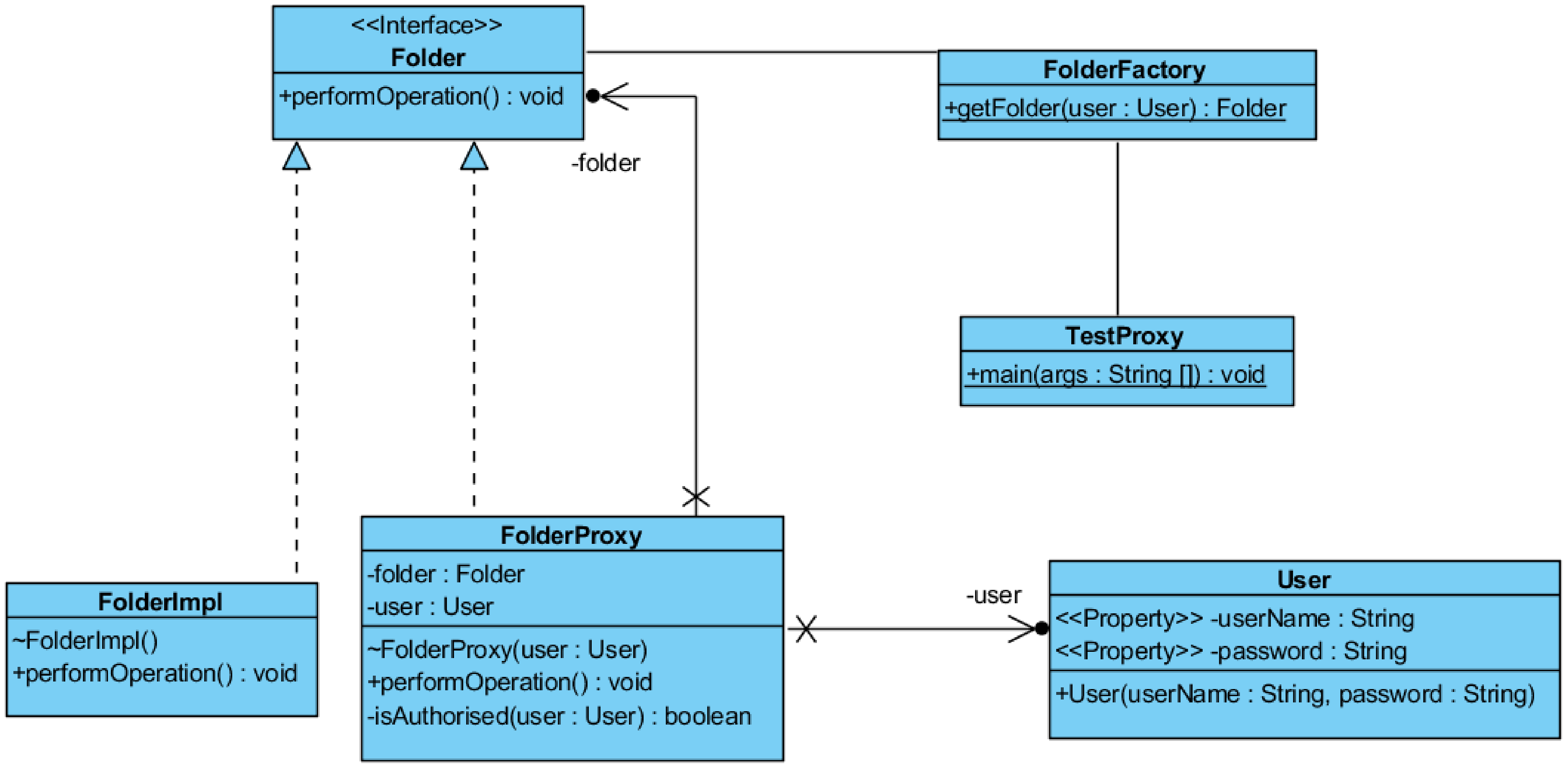
The **Proxy** class has a reference field that points to a service object. After the proxy finishes its processing (e.g., lazy initialization, logging, access control, caching, etc.), it passes the request to the service object.
Usually, proxies manage the full lifecycle of their service objects.

The **Service** is a class that provides some useful business logic.

Example: Security Proxy

- Suppose we have a folder in which you we perform various operation such as copy, paste file or subfolder.
- An existing `Folder` interface and `FolderImpl` class which provides a `performOperation()` method (which we cannot change)
- We want to further specify that only a user with authorization has access to perform operations such as cut or delete files.
- Creating a proxy to the existing `Folder/FolderImpl` allows us to leverage existing code without modification
- This solution is also better for cohesion and ‘separation of concerns’ since we implement security functionality in a different class to the original `FolderImpl` class (i.e. implement authentication/authorisation in a new `FolderProxy` class)
- See class diagram on next slide

Class Diagram



<<Interface>>
Folder
+performOperation() : void
FolderFactory
+getFolder(user : User) : Folder
-folder
TestProxy
+main(args : String []) : void
FolderProxy
-folder : Folder
-user : User
~FolderProxy(user : User)
+performOperation() : void
-isAuthorised(user : User) : boolean
FolderImpl
~FolderImpl()
+performOperation() : void
-user
User
<<Property>> -userName : String
<<Property>> -password : String
+User(userName : String, password : String)

See `proxy.folder.*` example
from http://www.codeproject.com/Tips/468254/Proxy-Design-Pattern-in-Java