Proxy Trinity: Design Patterns

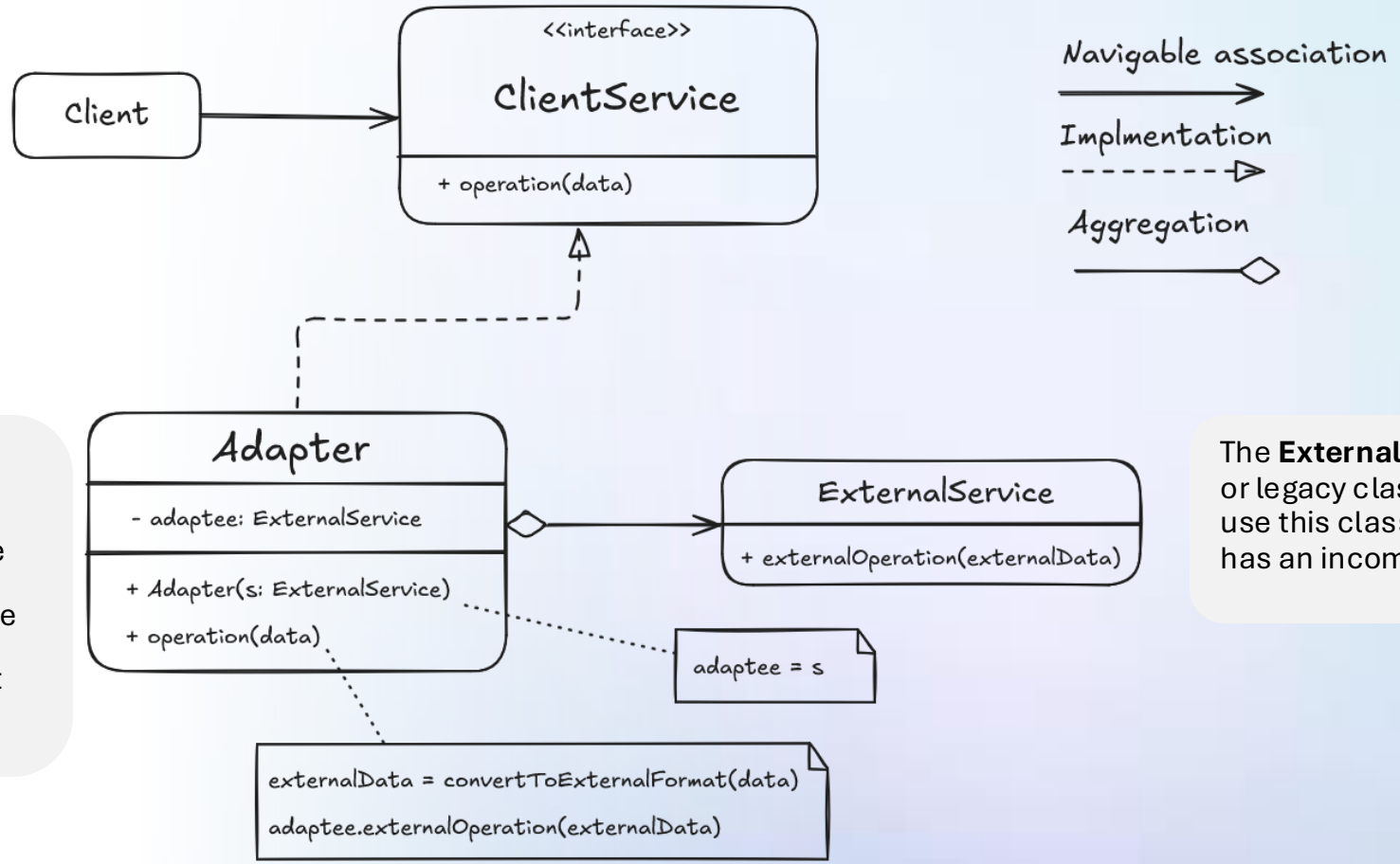
Content

1. Adapter
2. Proxy
3. Decorator

Adapter

The **ClientService** describes an interface that other classes must follow to be able to collaborate with the client code.

The **Client** is a class that contains the existing business logic of the program.



The **Adapter** is a class that's able to work with both the client and the service: it implements the client interface, while wrapping the service object. The adapter receives calls from the client via the client interface and translates them into calls to the wrapped service object in a format it can understand.

The **ExternalService** - a 3rd-party or legacy class. The client can't use this class directly because it has an incompatible interface.

Adapter

Applicability: when you need to change the interface of a class without changing its functionality.

Pros

- unified interface adapters and services
- simplifies client code and tests
- simplifies external libs integration
- reduce dependency on external libraries interfaces
- follows *Single Responsibility* and *Open/Close* Principles

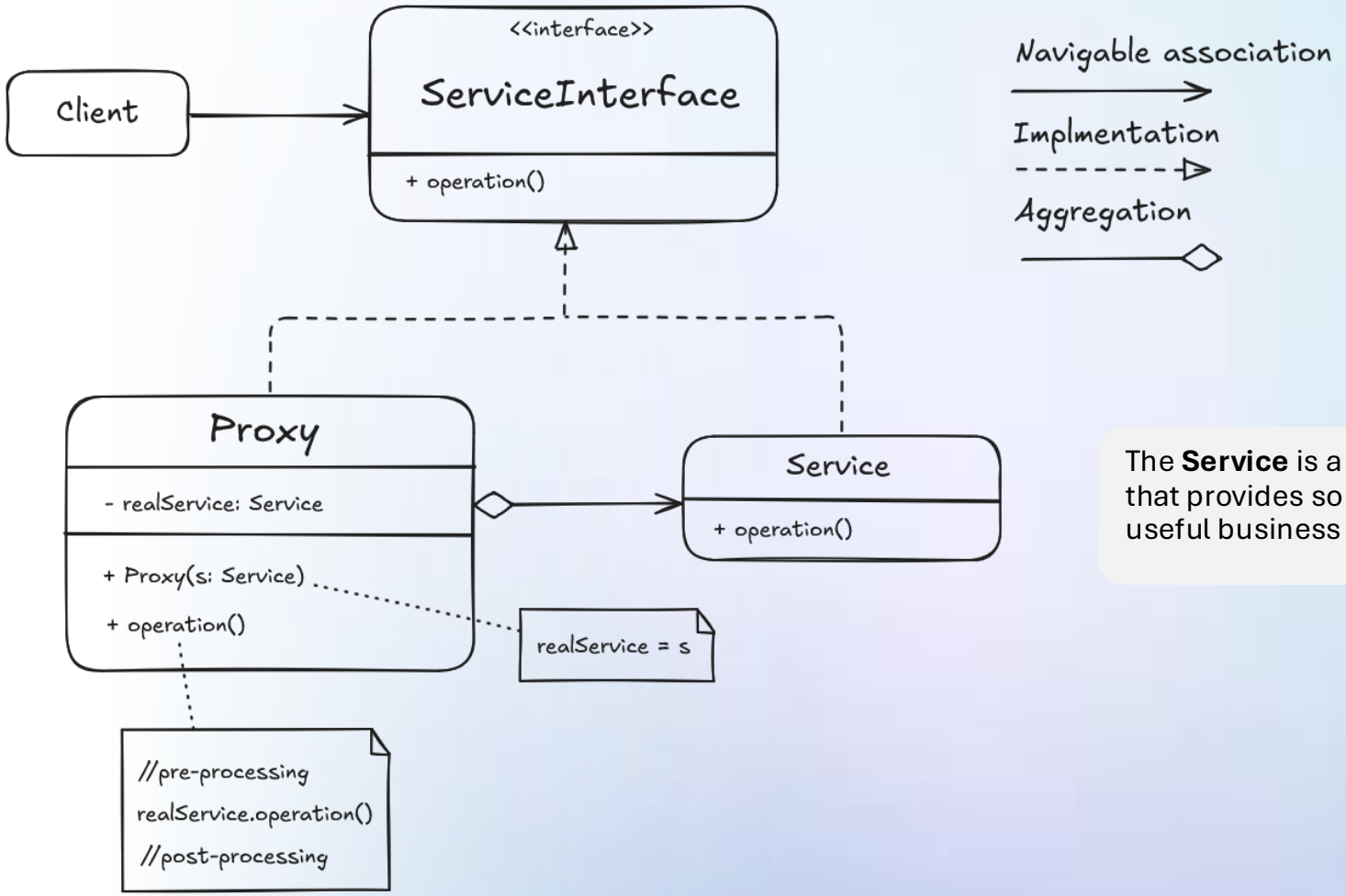
Cons

- increased code complexity (extra classes and interfaces)
- a lot of manual work for huge external classes
- additional maintenance efforts

Proxy

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

The **Client** works with services via interfaces. This way you can pass a proxy into any code that expects a service object.



The **Proxy** class has a reference field that points to a service object or its interface.

The proxy can provide a pre-processing (e.g. access control, caching, etc.) and post-processing steps (e.g. resource cleaning, sending an event, etc.) before and after delegating client request to the service object.

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

Proxy

Applicability: when you need to change the functionality of a class without changing its interface.

Pros

- the ability to add new logic without changing existing classes
- extra control over wrapped objects without clients knowing about it
- the ability to manage the lifecycle of wrapped objects
- follows *Single Responsibility*

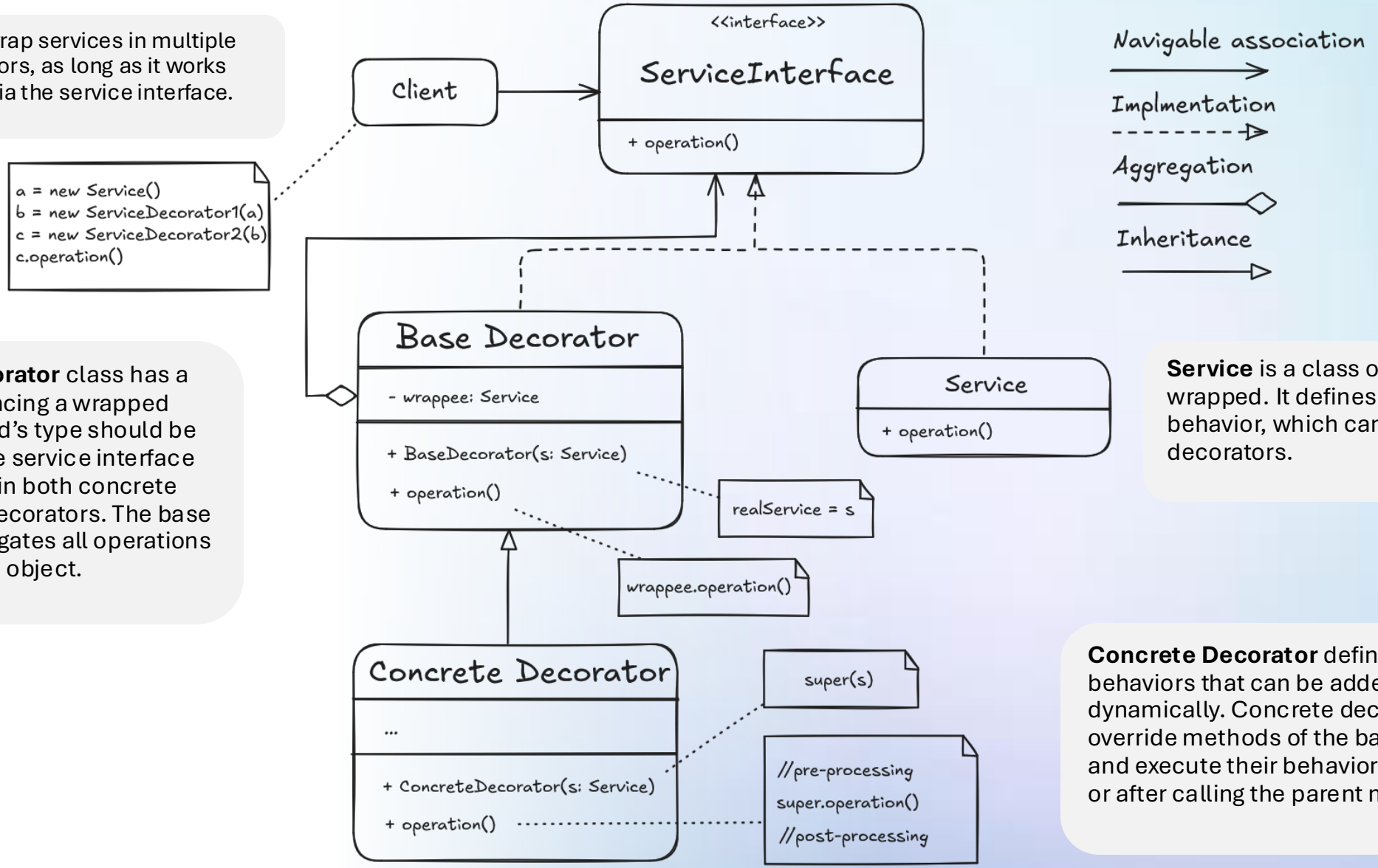
Cons

- increased code complexity (extra classes and interfaces)
- increased response time because of more classes in the request chain

Decorator

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

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



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

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

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

CODEUS_

Decorator

Applicability: when you need to dynamically change the functionality of a class without changing its interface.

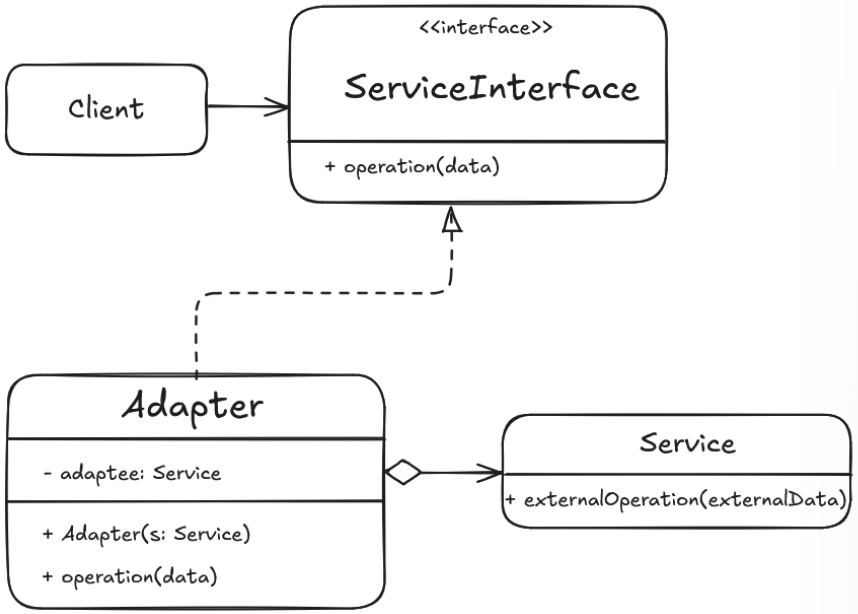
Pros

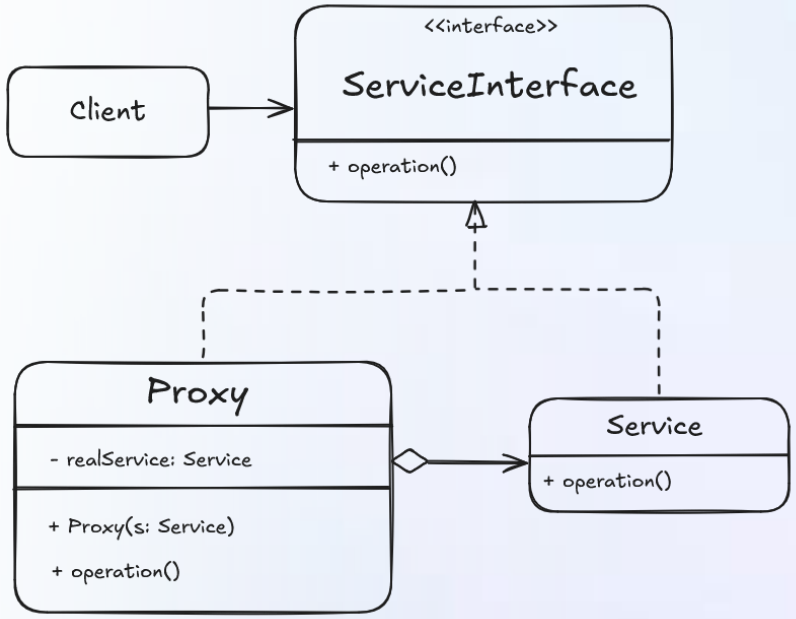
- the ability to add new logic without changing existing classes
- the ability to add new behavior to objects at runtime
- the ability to combine different behaviors
- follows *Single Responsibility*

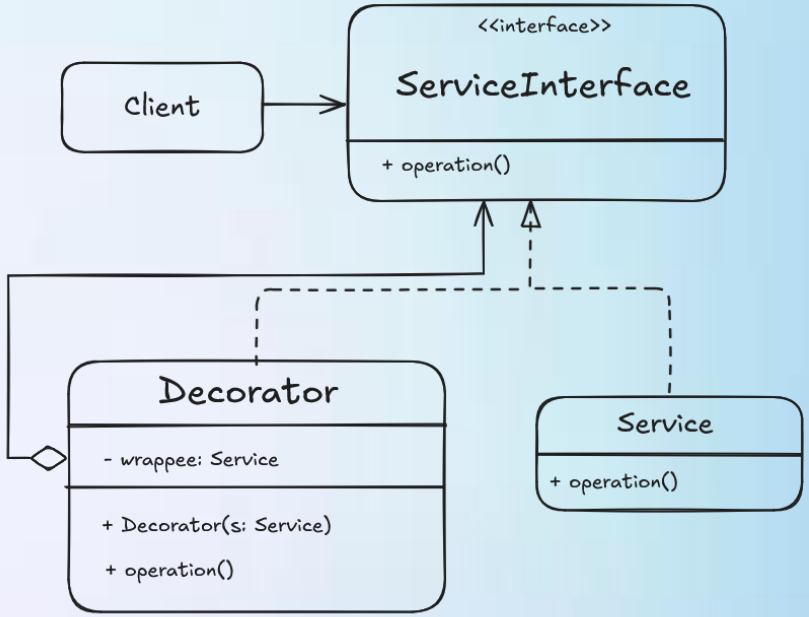
Cons

- increased code complexity (extra classes and interfaces)
- increased response time because of more classes in the request chain
- initial configuration can be complex and cumbersome
- it can be tricky to make decorators independent

Proxy Trinity







Clearly, all three patterns share similarities. Each subsequent pattern builds upon and enhances the one before it, creating a cohesive progression. Still, each of them has its own place and use cases.

Resources
and inspiration

1. Sergey Nemchinskiy: 3 Popular GoF patterns: YouTube
2. Refactoring Guru: The Catalog of Design Patterns: WebSite

Thank you

- Author: Serhii Kravchuk
- My LinkedIn
- Date: 10 December 2024
- Join Codeus community in Discord
- Join Codeus community in LinkedIn