Design Patterns (part 1 – creational and structural patterns)

Design Patterns – What Are They, When To Use Them

- In software engineering, a design pattern is a general repeatable solution to a commonly occurring problem in software design. A design pattern is not a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations.
- Design patterns can speed up the development process by providing tested, proven development paradigms.
- In addition, patterns allow developers to communicate using well-known, well understood names for software interactions.

Design Patterns – What Are They, When To Use Them

- A pattern use case tells you for which cases the pattern might be a good match
- Pattern consequences can be positive and/or negative: consider implementing a pattern when the advantages outweigh the disadvantages for your use case

Type of Design Patterns

- Creational Design Patterns

  - These design patterns are all about class instantiation.

- Structural Design Patterns

  - These design patterns are all about Class and Object composition.

- Behavioral Design Patterns

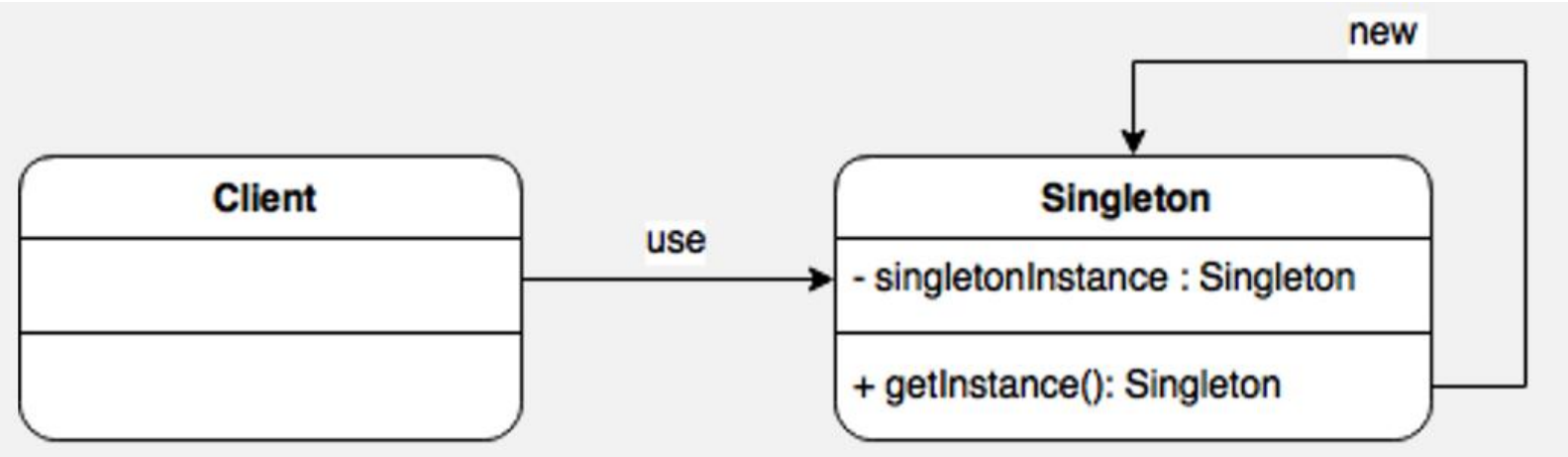
  - Behavioral patterns are those patterns that are most specifically concerned with communication between objects.

Singleton Pattern

Singleton Pattern

- Creational pattern
- Intent
  - ✓ Ensure a class has only one instance and provide a global point of access to it.
  - ✓ Encapsulated "just-in-time initialization" or "initialization on first use".
- Singleton should be considered only if all three of the following criteria are satisfied:
  - ✓ Ownership of the single instance cannot be reasonably assigned
  - ✓ Lazy initialization is desirable
  - ✓ Global access is not otherwise provided for
- Abstract Factory, Builder, and Prototype can use Singleton in their implementation.
- Facade objects are often Singletons because only one Facade object is required.
- State objects are often Singletons.
- Real-life examples: Logger, In-Memory database…

Singleton Pattern Structure



When to Use Singleton Pattern

- When there must be exactly one instance of a class, and it must be accessible to clients from a well-known access point

Singleton Pattern Consequences

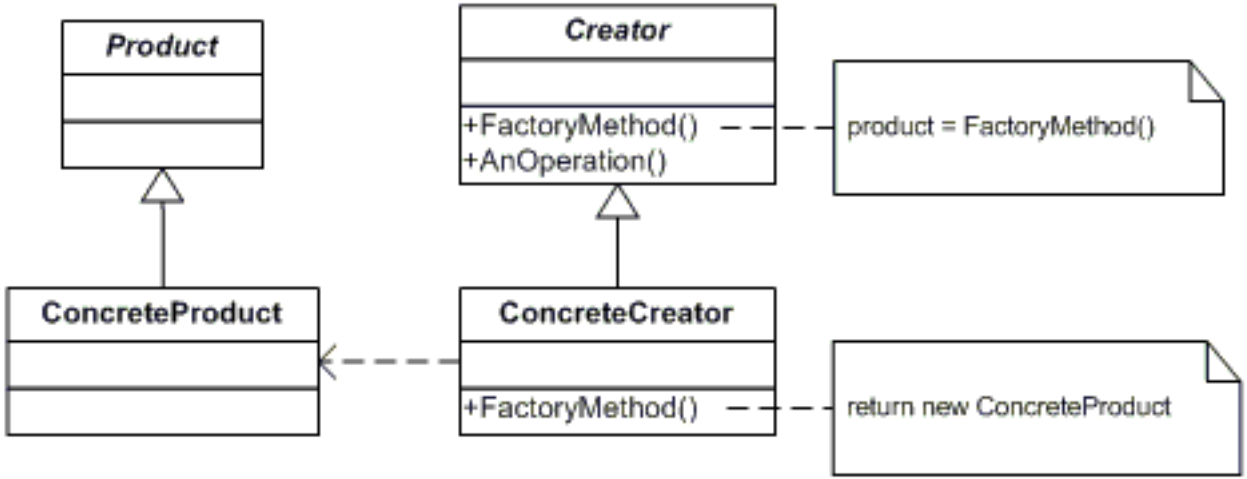
- Strict control over how and when clients access it
- Avoids polluting the namespace with global variables
- Violates the single responsibility principle

Factory Method Pattern

Factory Method Pattern

- Creational pattern
- Intent: defining an interface for creating an object, but to let subclasses decide which class to instantiate. Factory method lets a class defer instantiation to subclasses.

Factory Method Pattern Structure



When to Use Factory Method Pattern

- When a class can't anticipate the class of objects it must create
- When a class wants its subclasses to specify the objects it creates
- When classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate
- As a way to enable reusing of existing objects

Factory Method Pattern Consequences

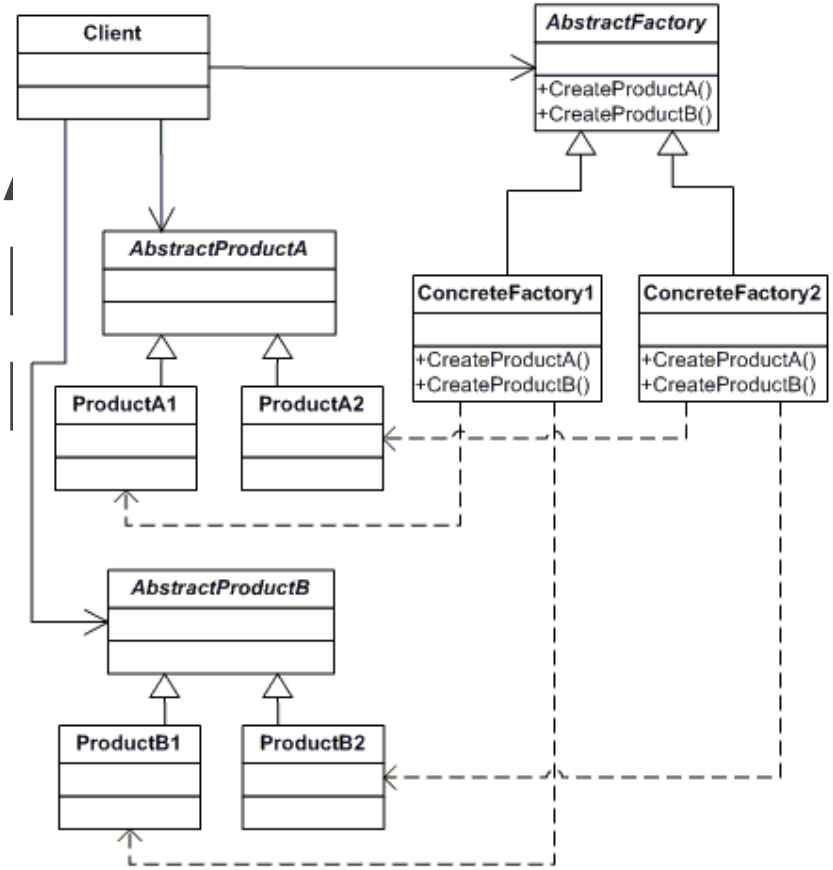
- Factory methods eliminate the need to bind application-specific classes to your code
- New types of products can be added without breaking client code: **open/closed principle**
- Creating products is moved to one specific place in your code, the creator: **single responsibility principle**
- Clients might need to create subclasses of the creator class just to create a particular ConcreteProduct object

Abstract Factory Pattern

Abstract Factory Pattern

- Creational pattern
- Intent
  - ✓ Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
  - ✓ A hierarchy that encapsulates: many possible “factories", and the construction of a suite of "products".
  - ✓ When the *new* operator is considered harmful.

Abstract Factory Pattern Structure



When to Use Abstract Factory Pattern

- When a system should be independent of how its products are created, composed and represented
- When you want to provide a class library of products and you only want to reveal their interfaces, not their implementations
- When a system should be configured with one of multiple families of products
- When a family of related product objects is designed to be used together and you want to enforce this constraint

Abstract Factory Pattern Consequences

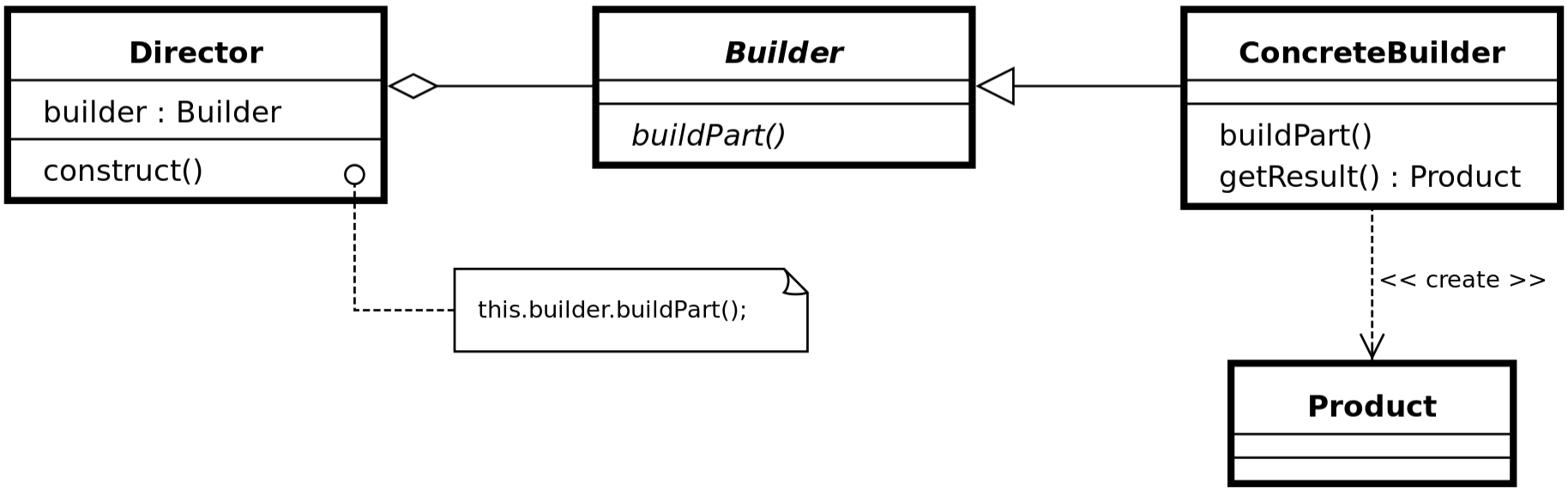
- It isolates concrete classes, because it encapsulates the responsibility and the process of creating product objects
- New products can easily be introduced without breaking client code: **open/closed principle**
- Code to create products is contained in one place: single **responsibility principle**

Builder Pattern

Builder Pattern

- Creational pattern
- Intent: Separate the construction of a complex object from its representation, so the same construction process can create different representations

Builder Pattern Structure



When to Use Builder Pattern

- When you want to make the algorithm for creating a complex object independent of the parts that make up the object and how they're assembled
- When you want to construction process to allow different representations for the object that's constructed

Builder Pattern Consequences

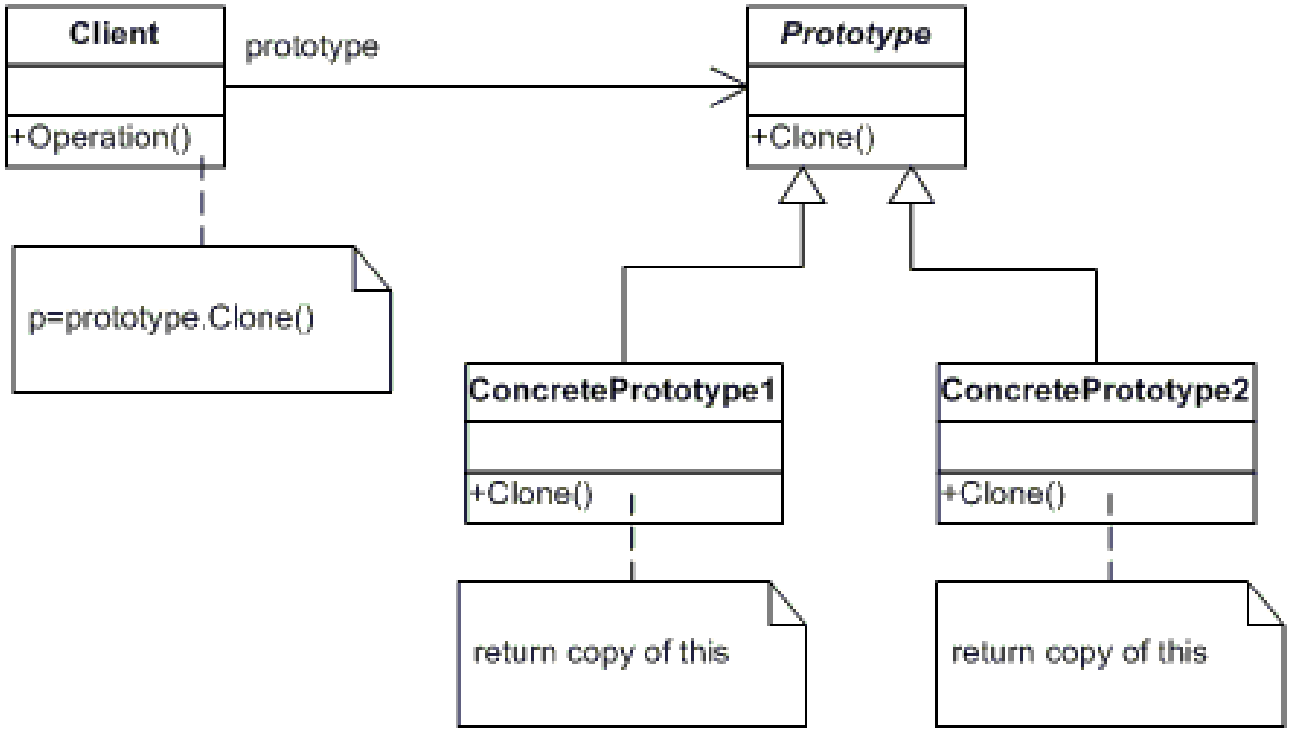
- It lets us vary a products' internal representation
- It isolates code for construction and representation; it thus improves modularity by encapsulating the way a complex object is constructed and represented: **single responsibility principle**
- It gives us finer control over the construction process
- Complexity of your code base increases


Prototype Pattern

Prototype Pattern

- Creational pattern
- Intent: specifying the kind of objects to create using a prototypical instance and create new objects by copying this prototype.

Prototype Pattern Structure



When to Use Prototype Pattern

- When a system should be independent of how its objects are created, and to avoid building a set of factories that mimics the class hierarchy
- When a system should be independent of how its objects are created, and when instances of a class can have one of only a few different combinations of states
- When the *new* operator considered harmful.

Prototype Pattern Consequences

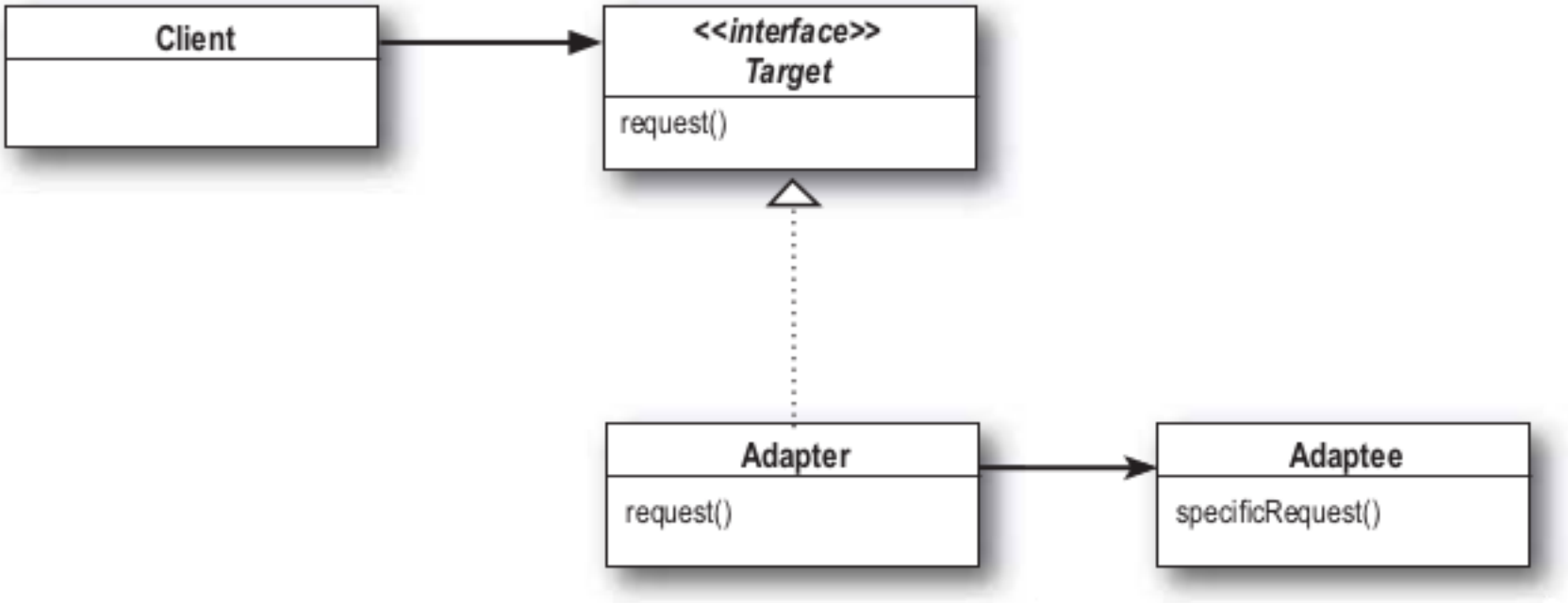
- Prototype hides the ConcreteProduct classes from the client, which reduces what the client needs to know
- Reduced subclassing
- Each implementation of the prototype base class must implement its own clone method


Adapter

Adapter Pattern

- Structural pattern
- Intent
  - ✓ Client collaborates with objects conforming to the Target interface

Adapter Pattern Structure



When to Use Adapter Pattern

- When you want to use an existing class, but the interface does not match the one you need
- When you want to create a reusable class (the adapter) that works with classes that don’t have compatible interfaces
- When you need to use several existing subclasses, don’t want create additional subclasses for each of them, but still need to adapt their interface

Adapter Pattern Consequences

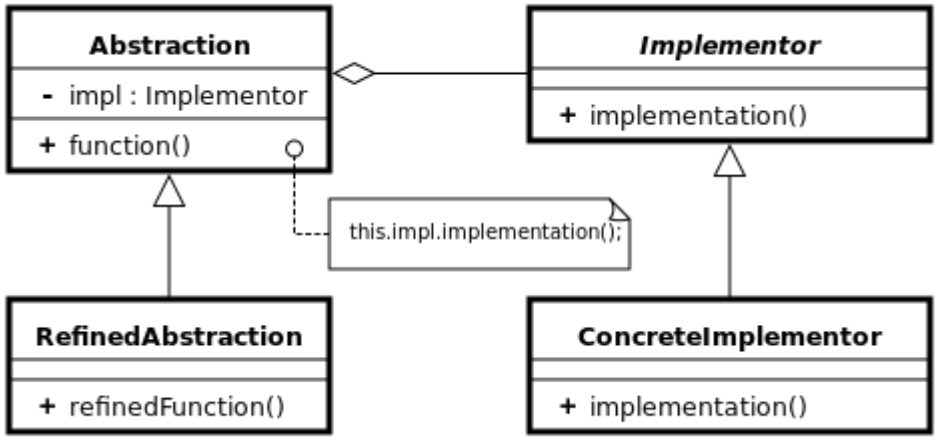
- A single adapter can work with many adaptees, and can add functionality to all adaptees at once
- The interface (adapter code) is separated out from the rest of the code: **single responsibility principle**
- New types of adapters can be introduced without breaking client code: **open/closed principle**
- The object adapter makes it hard to override adaptee behavior
- Additional complexity is introduced

Bridge Pattern

Bridge Pattern

- Structural pattern
- Intent
  - ✓ Decoupling an abstraction from its implementation so the two can vary independently
  - ✓ A means to replace an implementation with another implementation without modifying the abstraction
  - ✓ Abstractions handle complexity by hiding the parts we don’t need to know about

Bridge Pattern Structure



When to Use Bridge Pattern

- When you want to avoid a permanent binding between an abstraction and its implementation (to enable switching implementations at runtime)
- When abstraction and implementations should be extensible by subclassing
- When you don’t want changes in the implementation of an abstraction have an impact on the client

Bridge Pattern Consequences

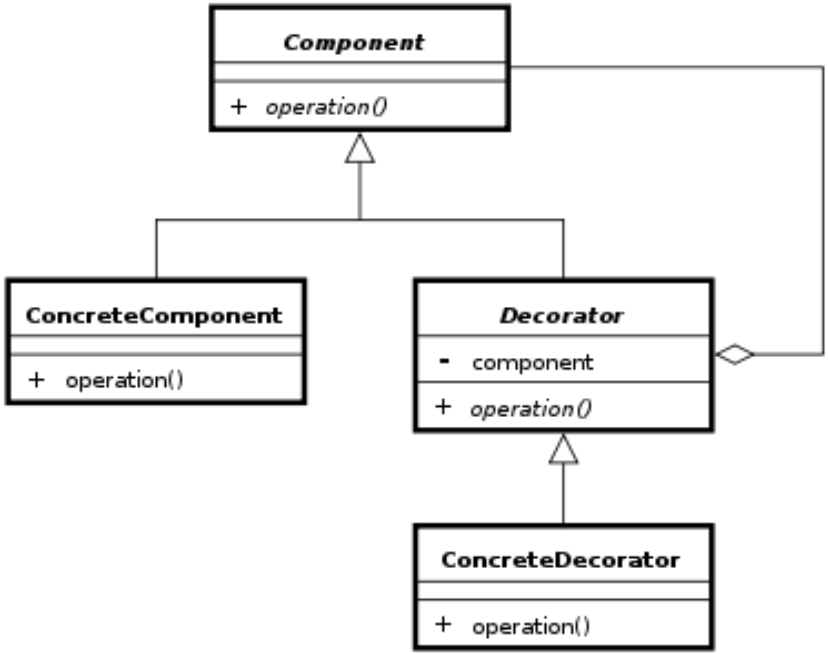
- Decoupling: the implementation isn't permanently bound to the abstraction
- As the abstraction and implementation hierarchies can evolve independently, new ones can be introduced as such: **open/closed principle**
- You can hide implementation details away from clients
- You can focus on high-level logic in the abstraction, and on the details in the implementation: **single responsibility principle**


Decorator Pattern

Decorator Pattern

- Structural pattern
- Intent:
  - ✓ Attach additional responsibilities to an object dynamically.
  - ✓ Provide flexible alternative to subclassing for additional functionality
- Pattern also known as wrapper.

Decorator Pattern Structure



When to Use Decorator Pattern

- When you have a need to add responsibilities to individual objects dynamically (at runtime) without affecting other objects
- When you need to be able to withdraw responsibilities you attached to an object
- When extension by subclassing is impractical or impossible

Decorator Pattern Consequences

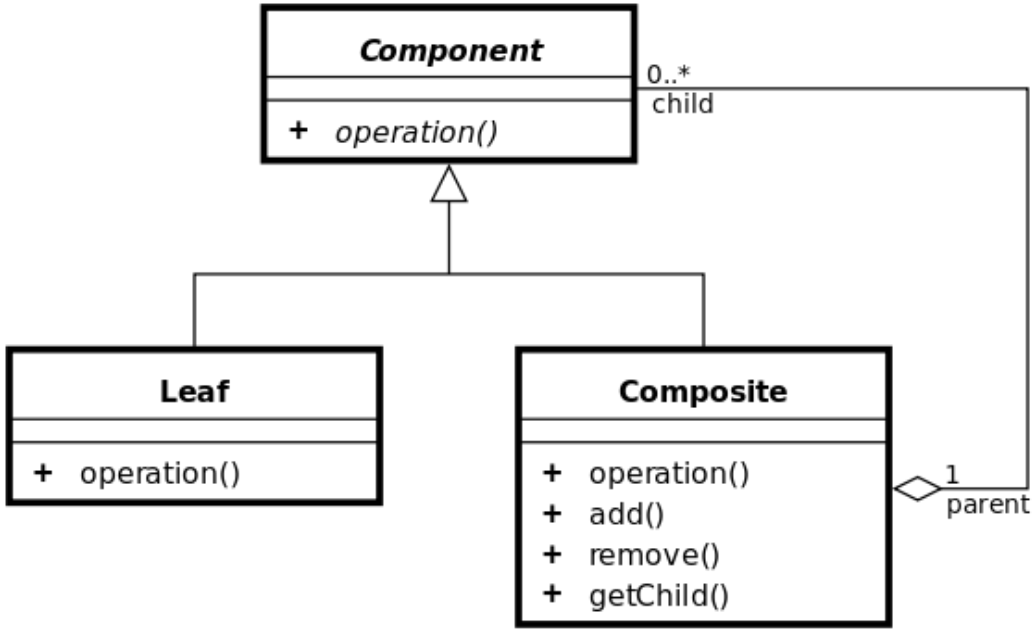
- More flexible than using static inheritance via subclassing: responsibilities can be added and removed at runtime ad hoc
- You can use the pattern to split feature-loaded classes until there's just one responsibility left per class: **single responsibility principle**
- Increased effort is required to learn the system due to the amount of small, simple classes

Composite Pattern

Composite Pattern

- Structural pattern
- Intent:
  - ✓ Represent part-whole hierarchies of objects
  - ✓ When it is needed to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly.
  - ✓ The composite pattern classifies each element in the tree as a composite or a leaf. A composite means that there can be other elements below it, whereas a leaf cannot have any elements below it. Therefore, the leaf must be at the very bottom of the tree

Composite Pattern Structure



When to Use Composite Pattern

- When you want to represent part-whole hierarchies of objects
- When you want to be able to ignore the difference between compositions of objects and individual objects

Composite Pattern Consequences

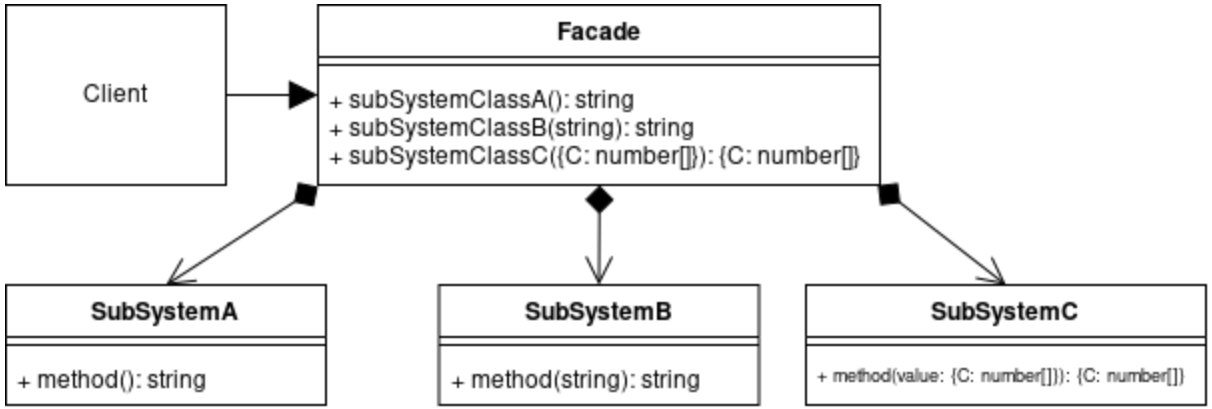
- Makes the client simple
- It’s easy to add new kinds of components: **open/closed principle**
- It can make the overall system too generic

Facade

Facade Pattern

- Structural pattern
- Intent: providing a unified interface to a set of interfaces in a subsystem. It defines a higher-level interface that makes the subsystem easier to use.

Facade Pattern Structure



When to Use Façade Pattern

- When you want to provide a simple interface into a complex subsystem
- When there are many dependencies between a client and the implementation classes of the abstraction

Facade Pattern Consequences

- The number of objects clients have to deal with are reduced
- It promotes weak coupling between the subsystem and its clients, enabling subsystem components to vary without affecting the client: open/closed principle
- Clients are not forbidden to use subsystem classes

Literature

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Design Patterns - Elements of Reusable Object-Oriented Software

John Dooley:

Software Development and Professional Practice (Chapter 11)

Thank You!