Design Patterns with TypeScript

Presented By: Mark Halpin

Who Am I?

- Software Developer at SAS
- Wrangler of rug rats
- Amateur woodworker
- Not a coffee drinker

Twitter- @halpinCodes

Website- www.halpincodes.com

What is your experience with design patterns?

- 1) I have heard of them, but have not used them
- 2) I have heard of them, and use them
- 3) I have not heard of them, but want to use them
- 4) I have not heard of them, but just want information at this point

What We Will Learn

- What are design patterns?
- Why do we need design patterns?
- Selecting and using design patterns
- How to implement 9 of the 23 well known design patterns

Design Patterns

- “Design Patterns: Elements of Reusable Object Oriented Software”. Otherwise known as the Gang of Four (GOF)
    - Eric Gamma
    - Richard Helm
    - Ralph Johnson
    - John Vlissides

	Creational	Structural	Behavioral
Class	Factory Method	Adapter (class)	Interpreter Template Method
Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Momento Observer State
			Strategy Visitor

How Design Patterns are Organized

As seen in “Design Patterns - Elements of Reusable Software”

What makes a design pattern?

- Design patterns typically contain 4 elements:
  - Definition of the pattern
  - Description of a common problem
  - Guidelines on how to solve those problems
  - Effects of pattern application

What Design Patterns Are

- A ‘shortcut’ to creating durable and maintainable code
- A ‘cookbook’ describing common problems and solutions
- A ‘springboard’ of basic concepts and architecture to make individual solution implementation easier
- Architectural concepts used in object oriented languages

What Design Patterns Are Not

- Not a copy and paste solution to any problem
- Not a set of strict standards with exact steps
- Not a silver bullet - Unnecessary use of design patterns creates complex, slow systems
- Not specific to any one object oriented programming language

Why do we need design patterns?

- Use them when you are repeatedly solving the same problem
- To create code that adheres to the object oriented design principles, some of which were created by ‘Uncle Bob’ Martin (found here: https://bit.ly/2W6XAVm)
- To keep code DRY(Don’t Repeat Yourself)
- Communicate ideas with a common set of terms among programmers

Selecting Design Patterns

- Consider the problem you are facing, and how the objects interact with one another

- Understand the purpose of the code you are trying to write

- Consider the ever changing elements in your design (encapsulate what varies)

Buy a book

- 23 design patterns
- More detailed information
- Good to have as a reference

Object Oriented Terms

- Coupling - Loose vs Tight
- Composition
- Encapsulation
- Abstraction

Using Design Patterns

- No set path to follow
- You code varies, as do pattern implementations
- Have a book handy

Understanding Patterns

As seen in “Design Patterns - Elements of Reusable Software”

- Pattern Name
- Intent
- Alias
- Motivation
- Applicability
- Applicability
- Structure
- Participants
- Collaborations
- Consequences
- Implementation
- Sample Code
- Known Uses
- Related Patterns

Questions?

Behavioral Patterns

Behavioral Design Patterns
Interpreter Template Method Chain of Responsibility
Command Iterator Mediator
Momento Observer State
Strategy Visitor

Behavioral Design Patterns

- Handles responsibilities of objects
- Manages how objects interact with one another
- Inheritance vs Composition

3 Common Behavioral Patterns

- Observer Pattern
- State Pattern
- Iterator Pattern

Understanding Behavioral Design Patterns

- Composition
- Loose Coupling / Tight Coupling
- Single Responsibility Principle

How to Choose Behavioral Patterns

- Do you have tightly coupled interfaces, which make adding functionality to existing classes difficult?

- Do you have aspects of your program that change frequently?

Observer Pattern

Observer Pattern

- Definition: Observer Pattern
- Problem: Keeping consistency across tightly coupled classes
- Solution: Pub-Sub Implementation
- Effects: Very loose coupling, broadcasting, unexpected updates

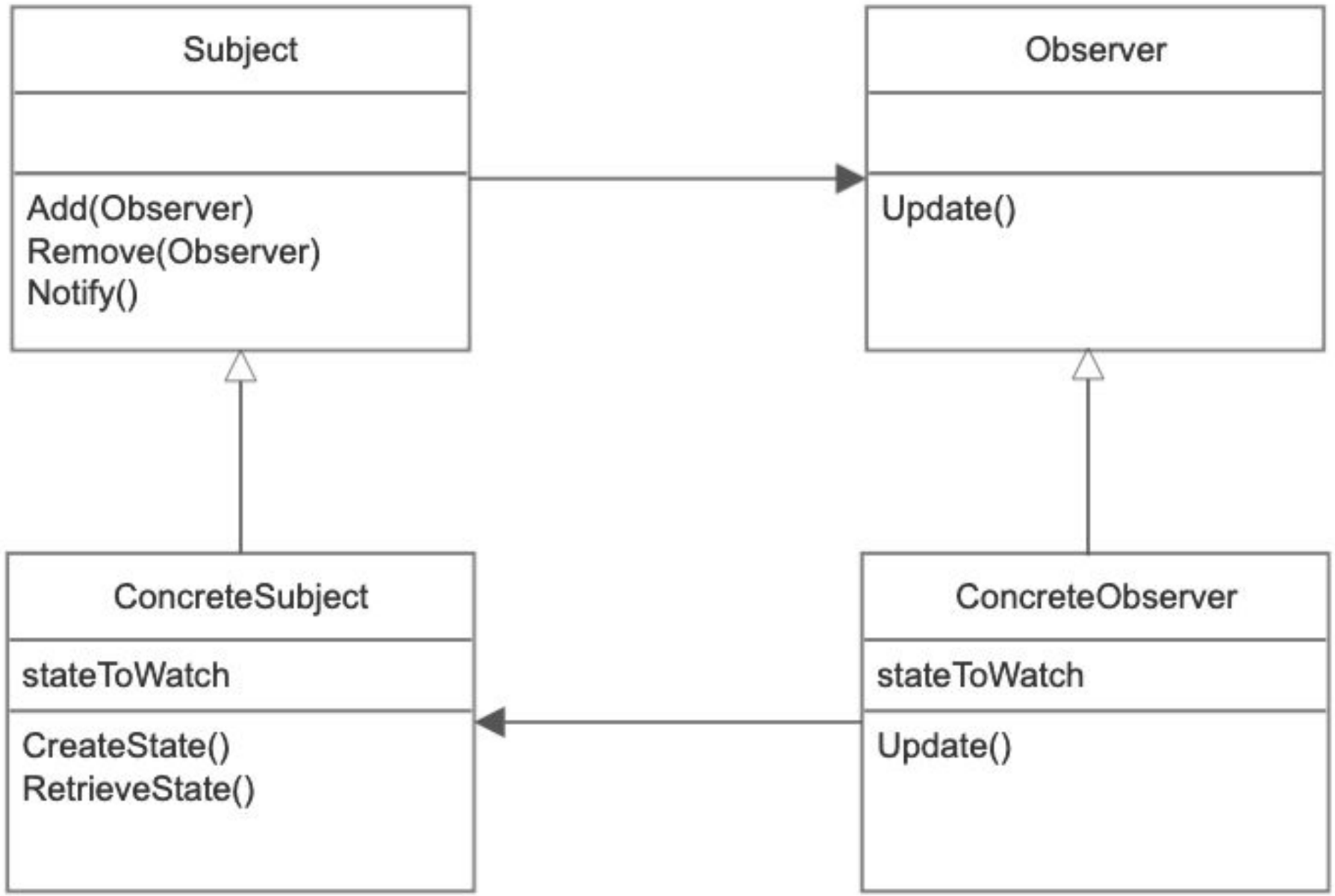

Diagram for Observer Pattern

Demo:

Observer Implementation

State Pattern

State Pattern

- Definition: State Pattern
- Problem: Changing behavior based on state (data), especially within large conditional statements
- Solution: State interface, used to track state between composed objects
- Effects: All state is handled in one object, which localizes state. State can be shared, and transitions are explicit

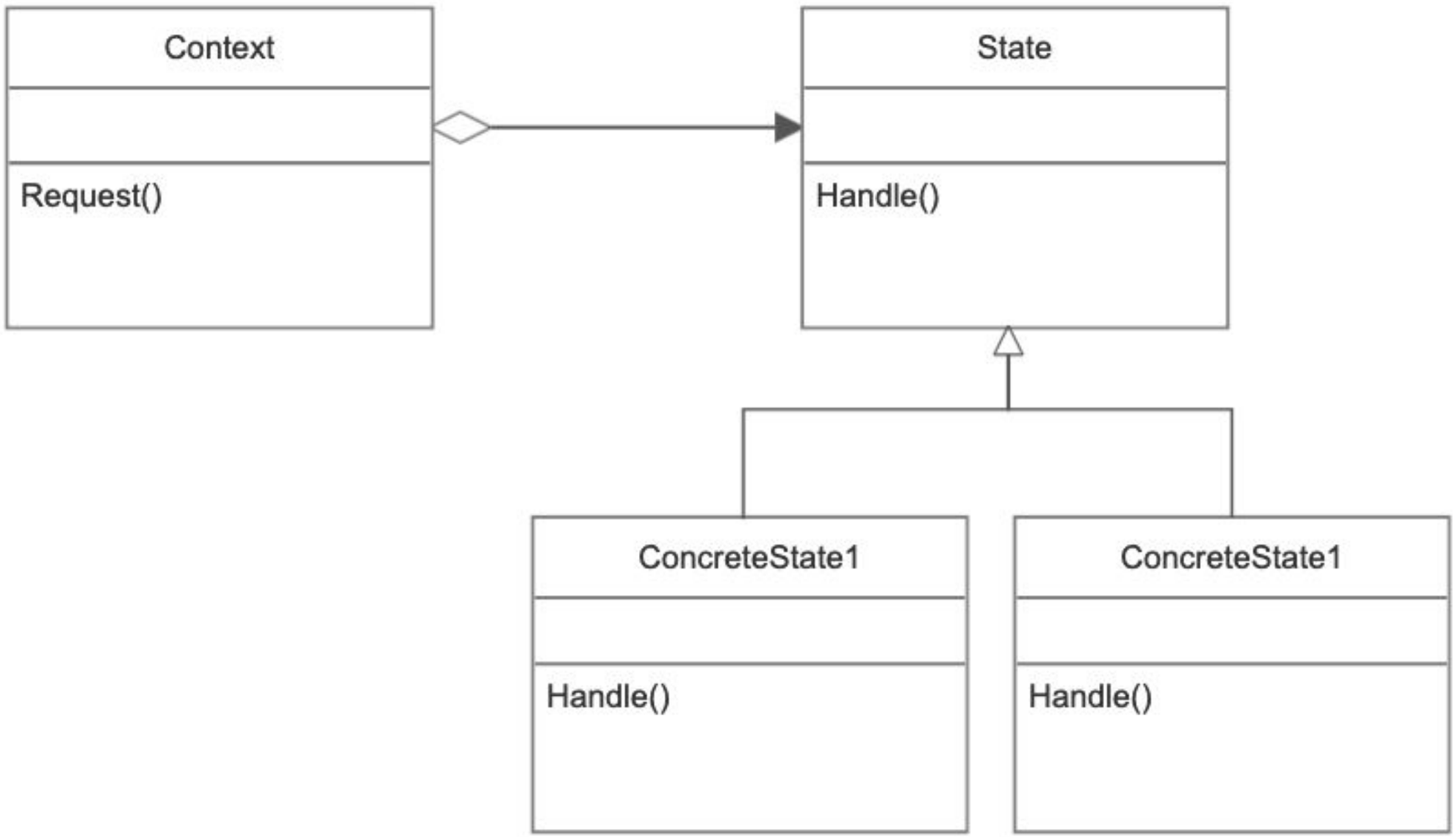

Diagram for State Pattern

Demo:

State Implementation

Iterator Pattern

Iterator Pattern

- Definition: Iterator Pattern
- Problem: Client code coupling with aggregate object code
- Solution: Separate out list aggregation code from client code
- Effects: Varieties of traversal implementations, simplified aggregate interface

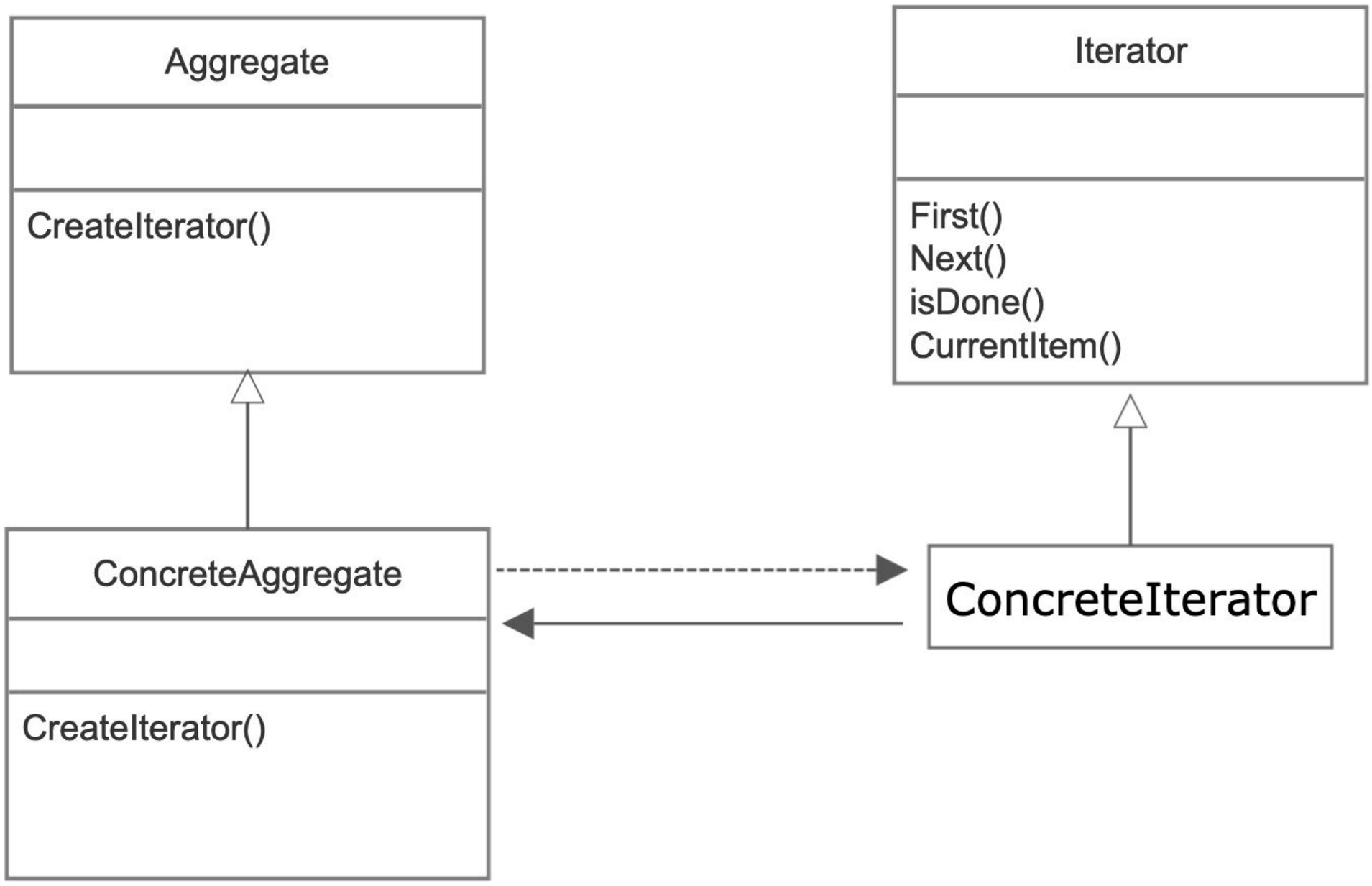

Diagram for Iterator Pattern

Demo:

Iterator Implementation

Can you see the benefit of design patterns?

- 1) Yes, it makes sense where I can use them
- 2) Yes, but I don’t know where to use them
- 3) No, I don’t see the benefit

Questions?

Let's Take a Brake!

Creational Patterns

Creational Design Patterns

Abstract Factory
Builder
Factory

Prototype
Singleton

Creational Design Patterns

- Used to abstract the process of instantiating objects
- Scenario when a system should not depend on how objects are created
- Becomes more important as systems grow in complexity

3 Common Creational Patterns

- Factory Pattern
- Prototype Pattern
- Singleton Pattern

Understanding Creational Design Patterns

- Encapsulation

- Abstraction

How to Choose Creational Patterns

- Do you want flexibility to determine what particular object gets created, which object creates it, how it gets created, and when?

- Do you have complex systems with hard coded data and fixed behaviors?

- Is object creation verbose, interdependent, and tightly coupled?

Factory Method Pattern

Factory Method Pattern

- Definition: Factory Method Pattern
- Problem: Locked into an implementation with concrete types, making it harder to create code that is easily extended
- Solution: Encapsulate the creation code within an object only concerned with creation
- Effects: Hooks for subclasses, parallel class hierarchies

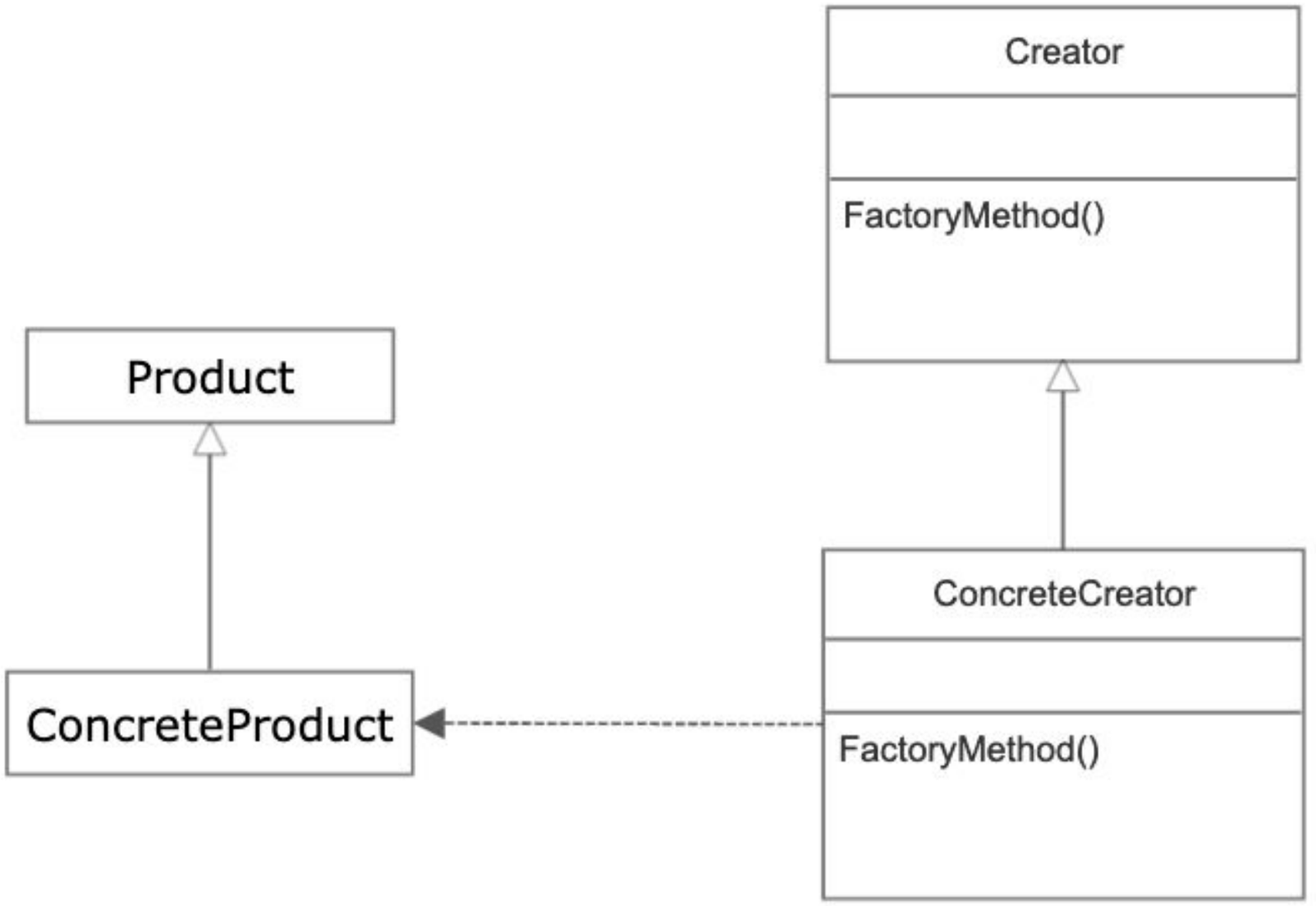

Diagram for Factory Pattern

Demo:

Factory Method Implementation

Prototype Pattern

Prototype Pattern

- Definition: Prototype Pattern
- Problem: Creating instances of other classes is complex
- Solution: Delegate object creation to the new objects being created
- Effects: Ability to add/remove objects at run time, create new objects with different values and structures, reduction of subclassing

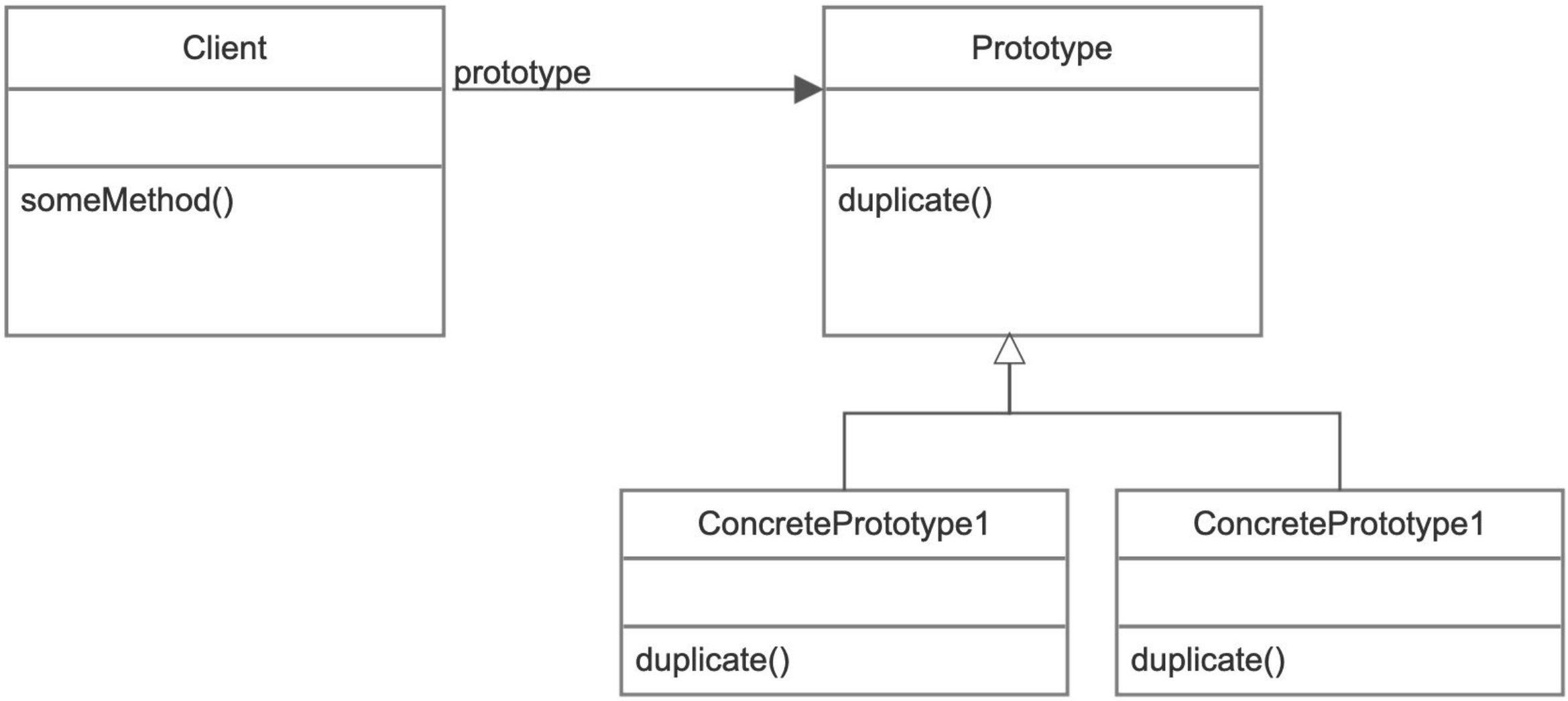

Diagram for Prototype Pattern

Demo:
Prototype Implementation

Singleton Pattern

Singleton Pattern

- Definition: Singleton Pattern
- Problem: Providing global access to a single instance of a class
- Solution: Prevent the ‘new’ operator by creating a static creation function that deals with a private constructor
- Effects: Controlled access to a single class, reduced name space, flexibility in creation at run time

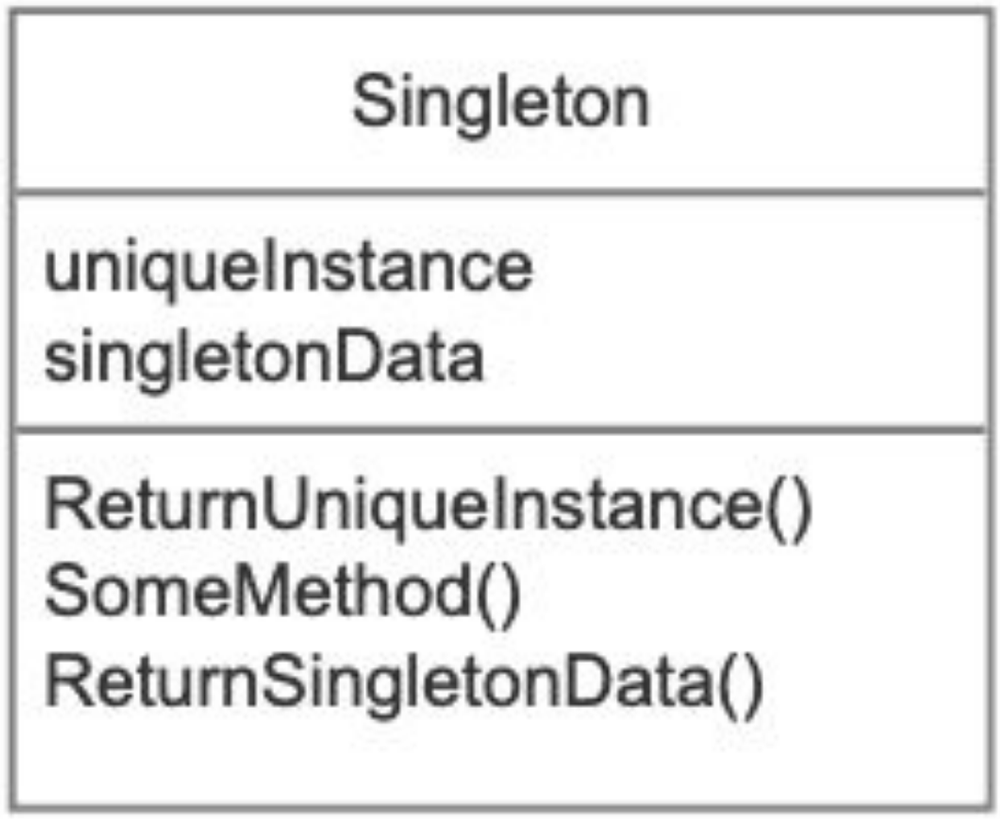

Diagram for Singleton Pattern

Demo:

Singleton Implementation

Questions?

Let's Take a Brake!

Structural Patterns

Structural Design Patterns

Adapter
Bridge
Composite

Decorator
Facade
Flyweight
Proxy

Structural Design Patterns

- Concerned with how objects and classes can be assembled into larger systems

- Keep larger systems flexible, efficient, and easy to change

- Uses inheritance to compose interfaces or implementations

3 Common Structural Patterns

- Adapter Pattern
- Facade Pattern
- Decorator Pattern

How to Choose Structural Patterns

- Do you have two or more complex systems that need to talk to one another?

- Are you trying to re-use code to tie systems together, without re-inventing the wheel?

Understanding Structural Patterns

- Focus on how objects and classes are used to form larger systems

- Used to describe ways of creating new functionality within existing objects

- Patterns use both inheritance and interfaces to extend functionality

Adapter Pattern

Adapter Pattern

- Definition: Adapter Pattern

- Problem: Integrating third party code, with incompatible or unchangeable classes/objects with your own

- Solution: Create an “adapter” that implements the incompatible code via a compatible operation

- Effects: Adapter overrides adapter’s behavior, abstraction over adapter method, adapted object no longer functioning as intended

Adapter Pattern

- The purpose of the adapter pattern is to make incompatible interfaces work together

- Example - Third party vendor code can’t be changed and needs to integrate with your system

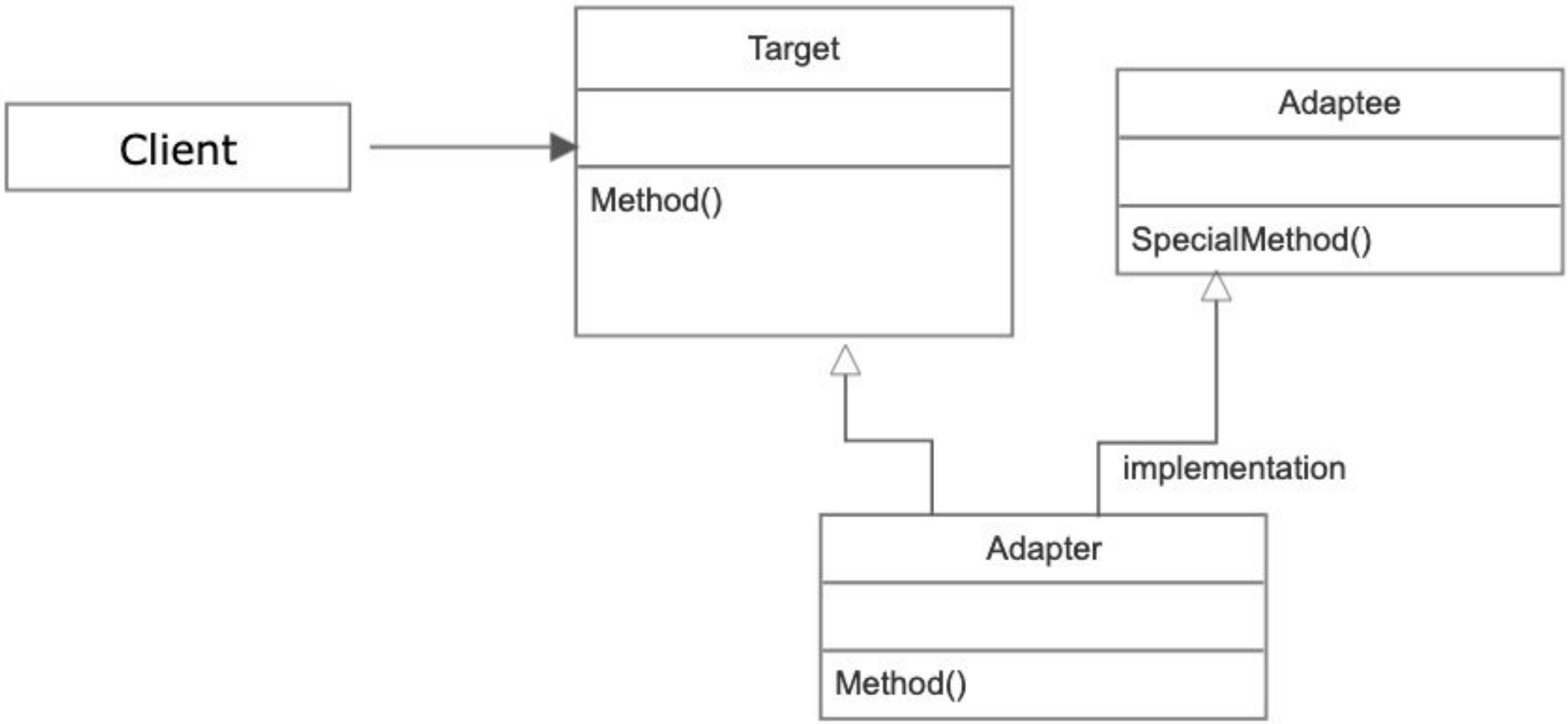

Diagram for Adapter Pattern

Demo:

Adapter Implementation

Facade Pattern

Facade Pattern

- Definition: Facade Pattern
- Problem: Multiple entry points into many complex subsystems that share data
- Solution: Implement a single interface to allow access to a complex system
- Effects: Abstracts complexity from client, loose coupling, allows subclassing

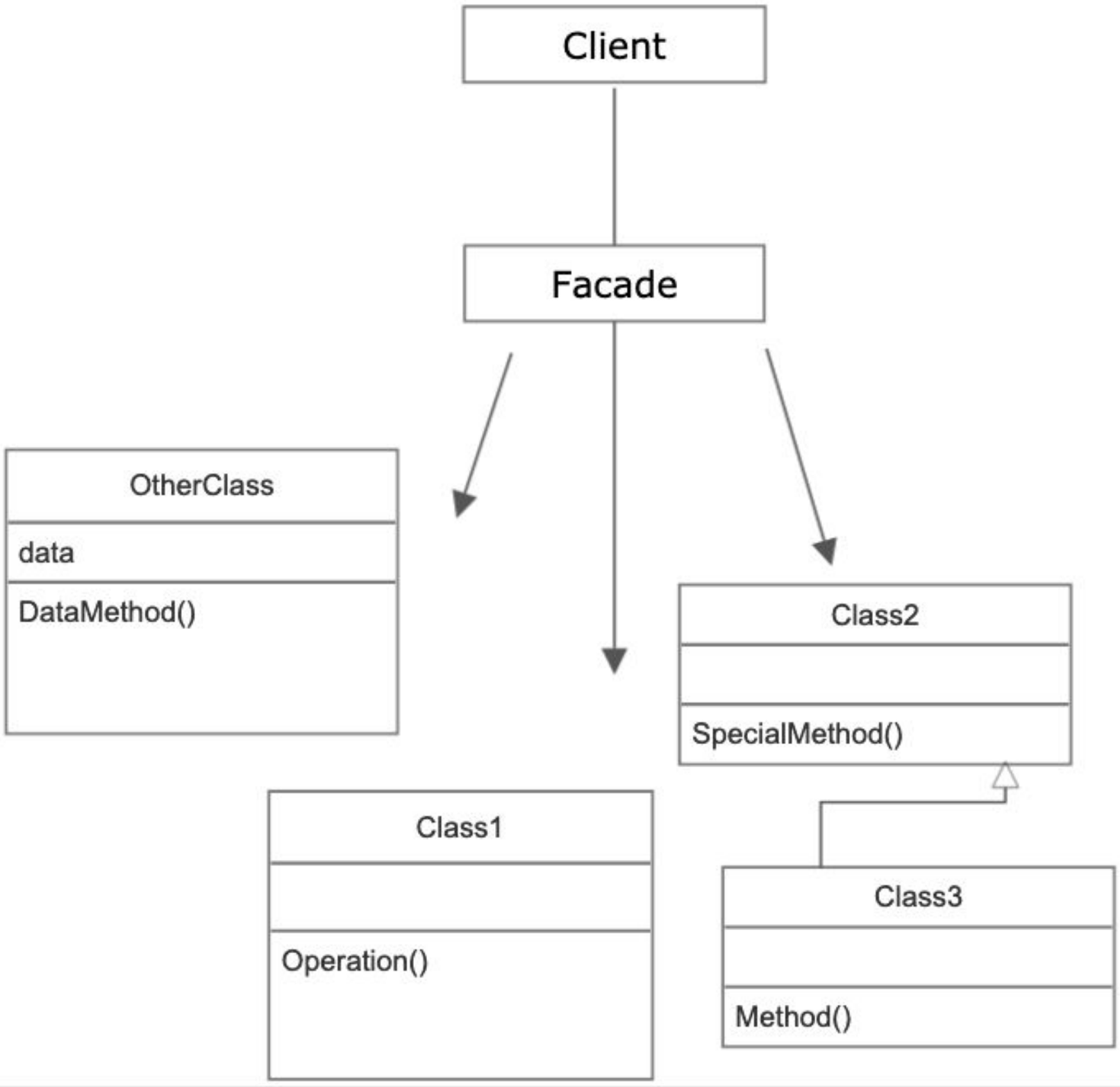

Diagram for Facade Pattern

Demo:

Facade Implementation

Decorator Pattern

Decorator Pattern

- Definition: Decorator Pattern
- Problem: Require the ability to add functionality to an existing object without touching its class
- Solution: Encapsulate the object that needs to change within another object
- Effects: Flexible way of adding responsibilities, re-usable functions that can be applied elsewhere

Types of Decorators

- 4 Types of Decorators
  - Class
  - Method
  - Property
  - Parameter

Decorator Basics

- Called at runtime
- Written as functions
- Multiple decorators can be used

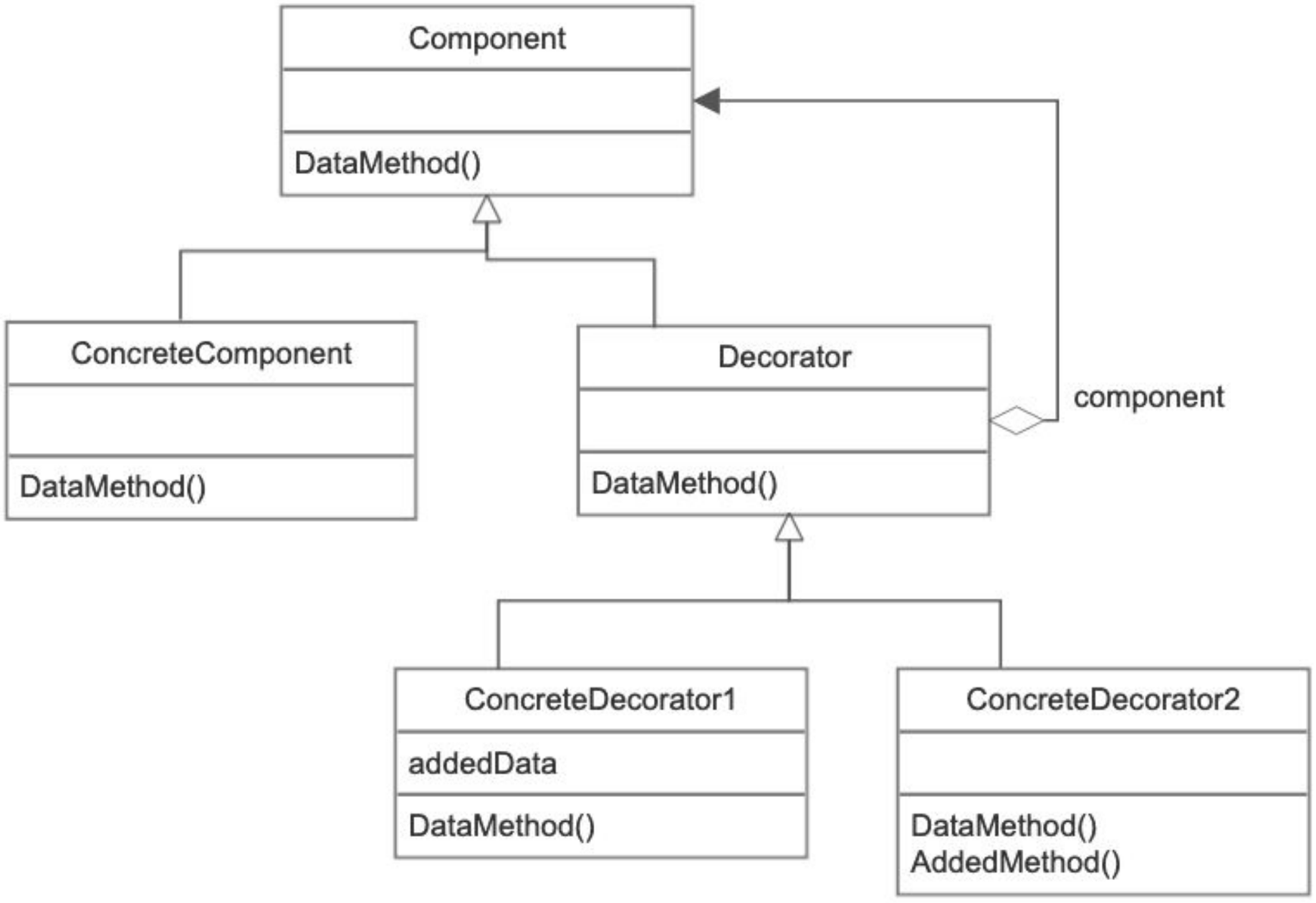

Diagram for Decorator Pattern

Important Notes

- TypeScript decorators are experimental
- All of these decorators, and their implementations are subject to change
- Must use experimental decorator flag, emit decorator metadata flag, and target es5 in your TypeScript config or CLI options
- I will not be going into all of these types

Demo:

Decorator Implementation

Summary

What did we learn?

- 9 Common Design Patterns
- Why design patterns are important
- What design patterns are, and what they are not
- How to choose patterns

Fool me once…

- Refactor in design patterns

- Don’t prematurely optimize

Questions?

Get in Touch!

- Twitter: @HalpinCodes

- Website: www.halpincodes.com

Thank You!