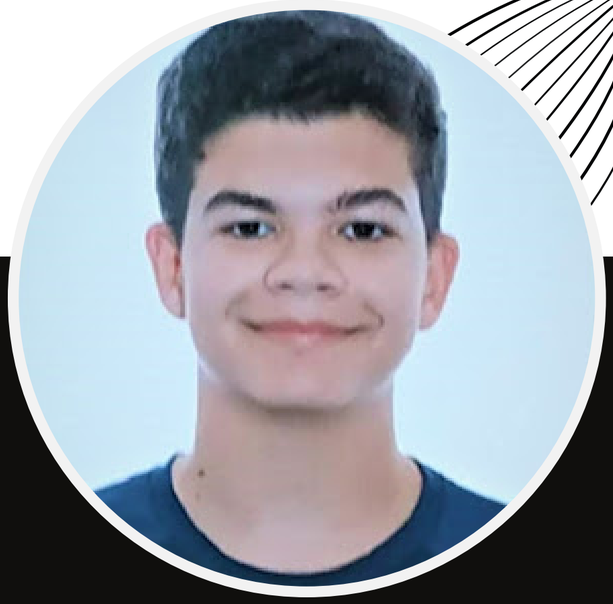
11/03/2024

DESIGN PATTERNS

Esther
Medina
Quintero

esther.quintero.33@ull.edu.es



Hugo
Hernández
Martín

hugo.hernandez.14@ull.edu.es

CONTENT

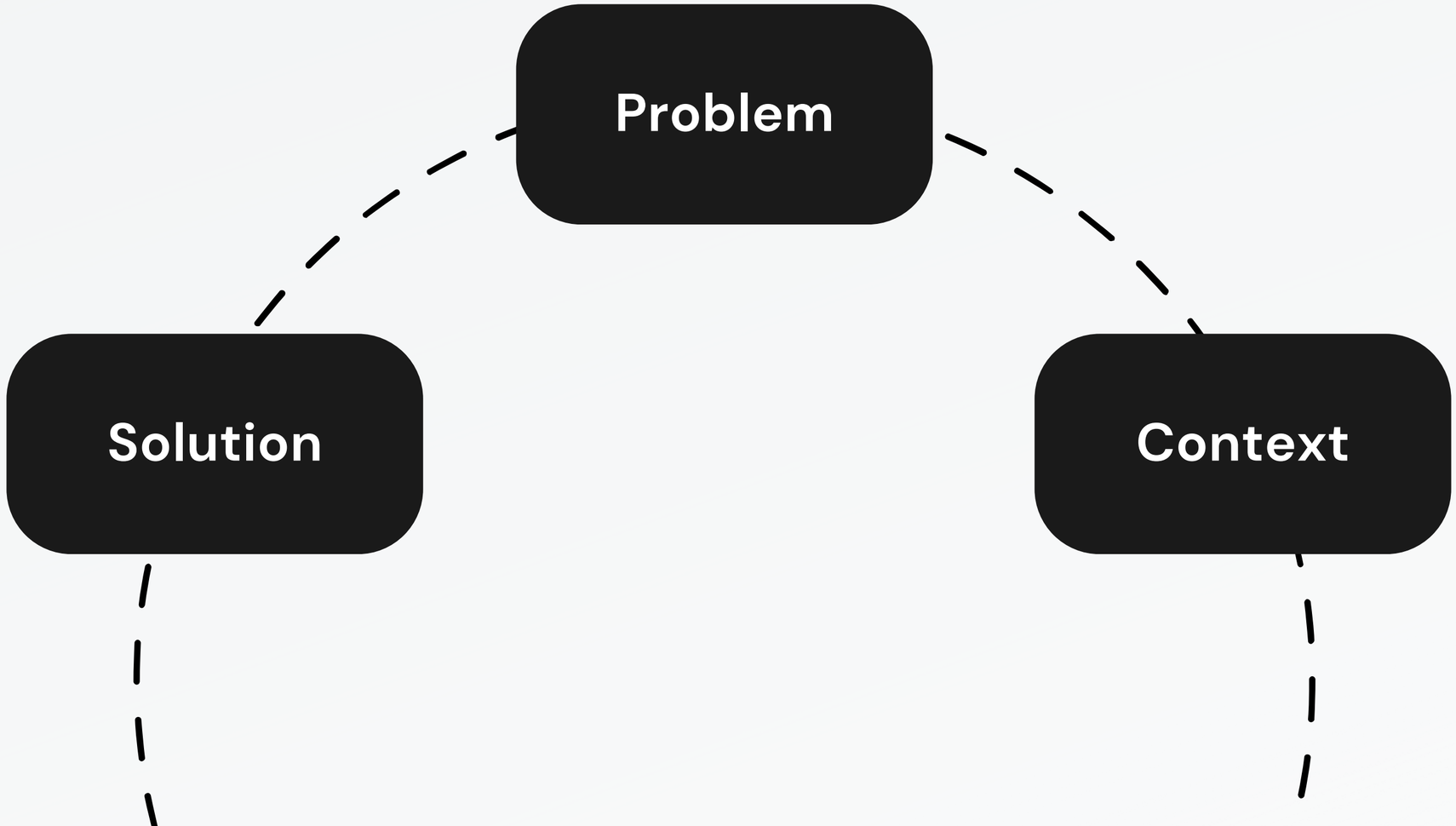
What are Design Patterns?

What is a Pattern?

“Each pattern is a three-part rule, which expresses a relation between a certain context, a problem and a solution.”

Christopher Alexander

What is a Pattern?



What is a Design Pattern?

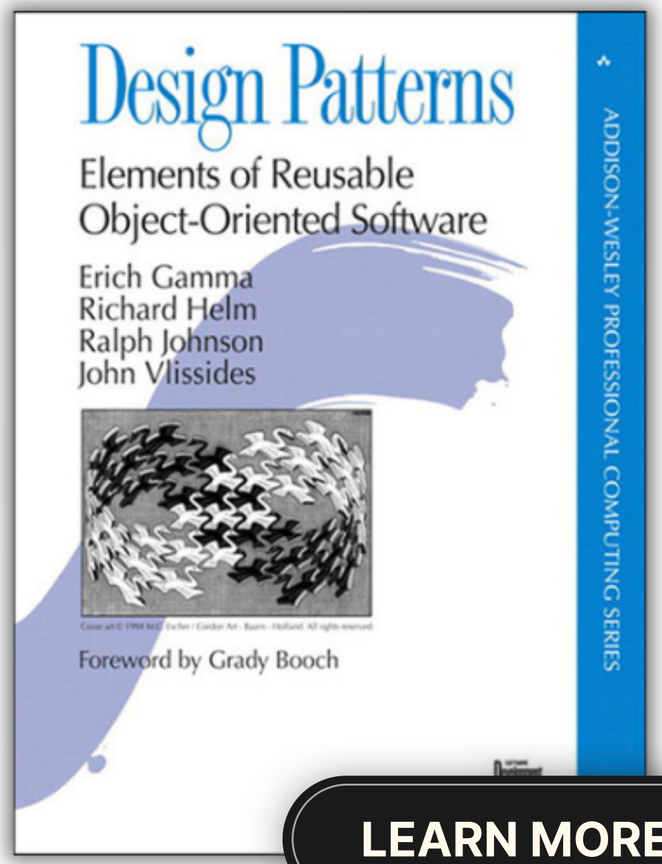
“Reusable solution to a commonly occurring problem within a given context in software design.”

A little bit of History

1977 — 1987 — 1994

Kent Beck

Ward Cunningham



LEARN MORE

Why use Design Patterns?

Better code

- More efficient.
- More reusable.
- More maintainable.

Proven solutions

- Common problems.
- Save time.
- Save effort.

Refactoring

- Less refactoring.
- Optimal solution.

Types of Design Patterns

Types of Design Patterns

01 Creational

02 Structural

03 Behavioral

LEARN MORE

Pattern

Frequency

Creational Patterns

Creational Patterns

- Provides various object creation mechanisms.
    - Easy to test.
    - Easy to reuse.
    - Increase flexibility.

Creational Patterns

01 Factory

02 Abstract factory

03 Builder

04 Dependency injection

05 Lazy initialization

06 Multiton

07 Object pool

08 Protoype

09 RAII

10 Singleton

Creational Patterns

03 Builder

08 Protoype

10 Singleton

Singleton

Singleton

Singleton

Problems

- Ensure that a class has just a single instance.
- Provide a global access point to that instance.

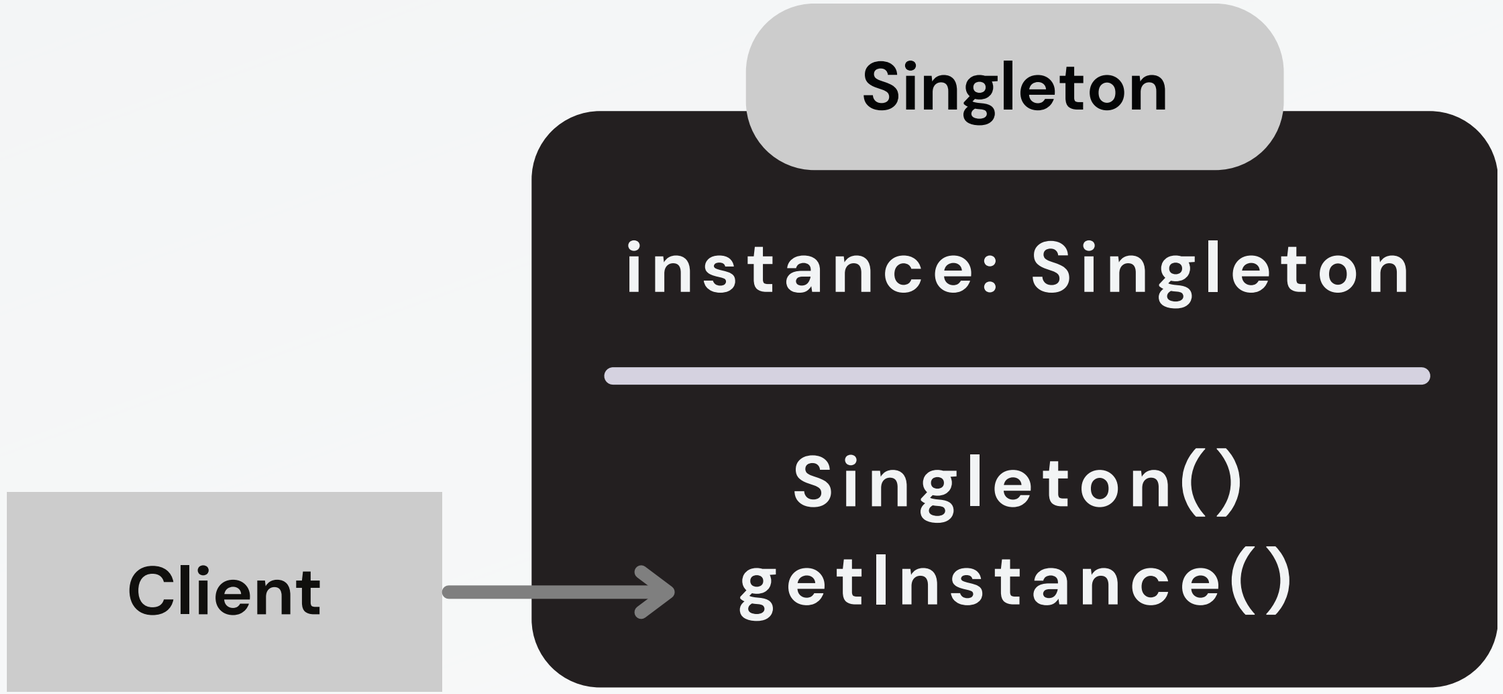
Singleton

Solution

- Make the default constructor private.
- Create a static creation method that acts as a constructor.

Singleton

Solution



```
class SingletonClass {
  private static singletonInstance: SingletonClass;
  static getInstance(exampleParameter: string): SingletonClass {
    if (!this.singletonInstance) {
      this.singletonInstance = new SingletonClass(exampleParameter);
    }
    return this.singletonInstance;
  }
  GetExampleAttribute(): string { return this.exampleAttribute; }
  private constructor(private exampleAttribute: string) {}
}
```


singleton.ts

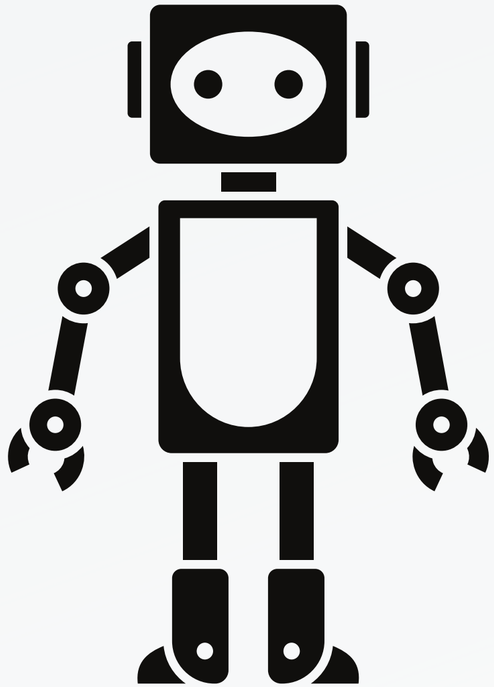
```
function main(): void {
  // You can't do: myInstance = new SingletonClass('my example');
  const MY_FIRST_INSTANCE: SingletonClass =
      SingletonClass.getInstance('My Singleton Class');
  const MY_SECOND_INSTANCE: SingletonClass =
      SingletonClass.getInstance('Nothing done here');
}
```


singleton.ts

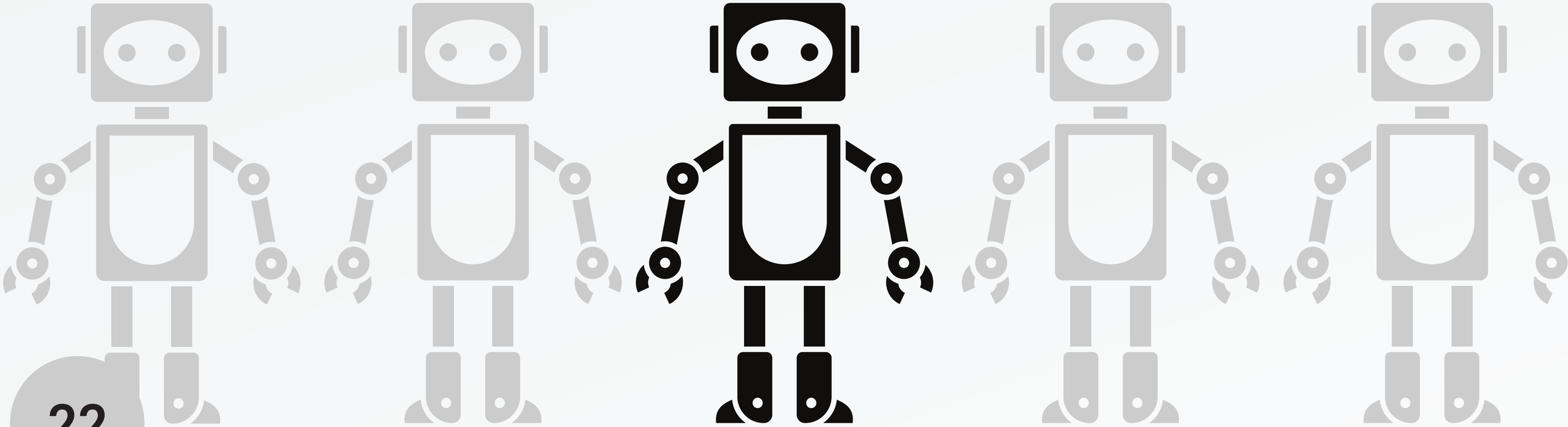
Singleton

This pattern violates the ***Single Responsibility Principle***.

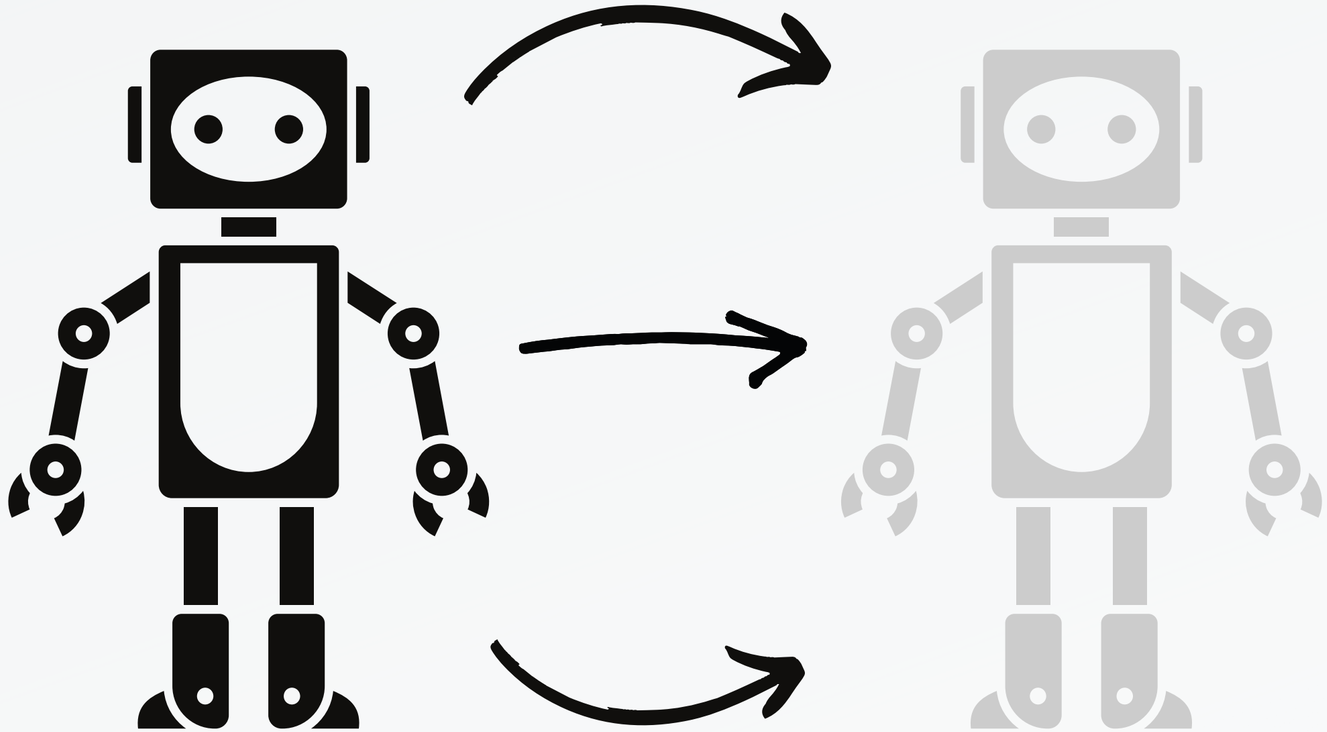
Prototype



Prototype



Prototype



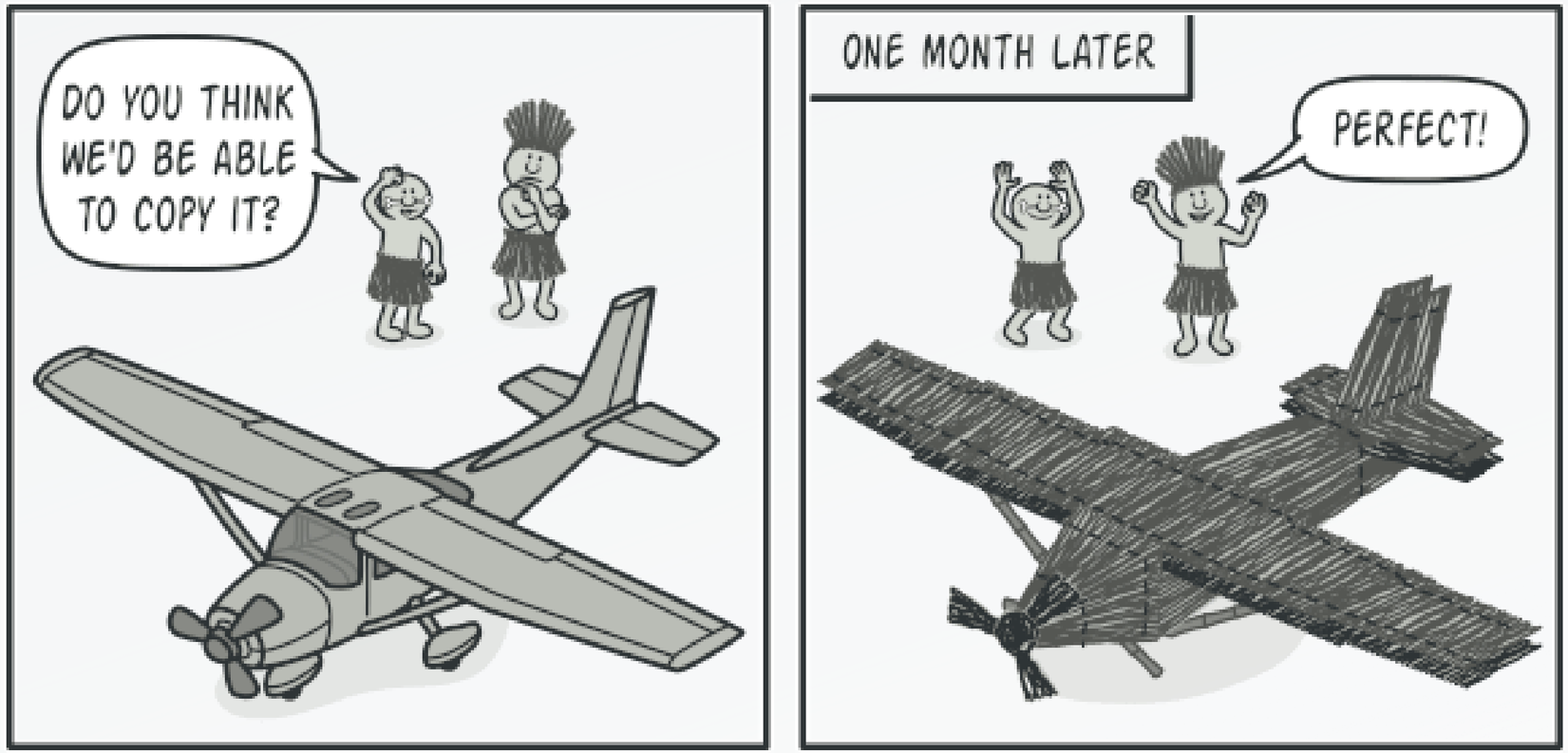
Prototype

Problems

- Create a new object.
- Go through all the fields of the original object.
- Your code becomens dependent on the class.

Prototype

Problems



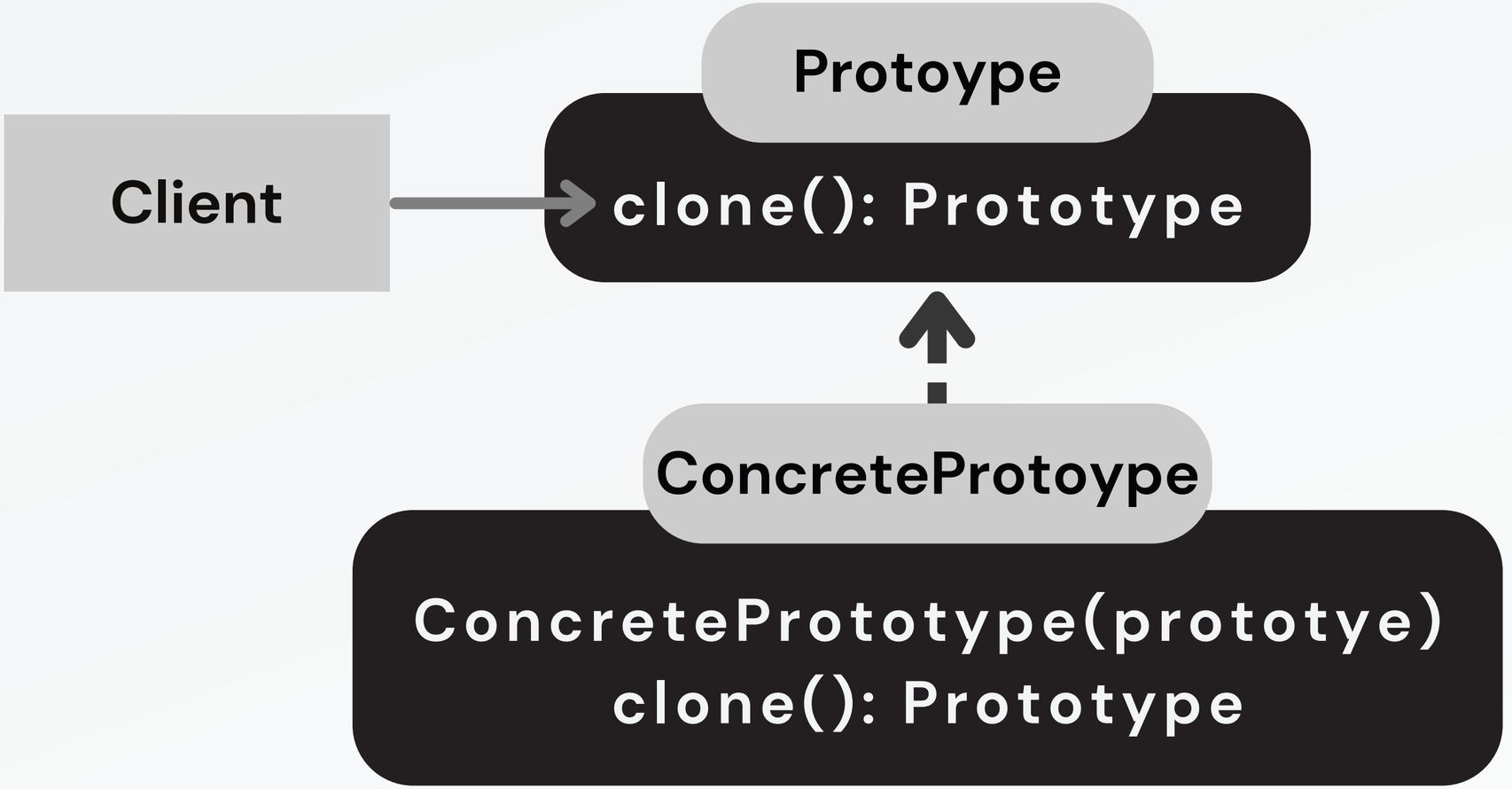
Prototype

Solution

- Delegate the cloning process.
- Common interface for all objects.

Prototype

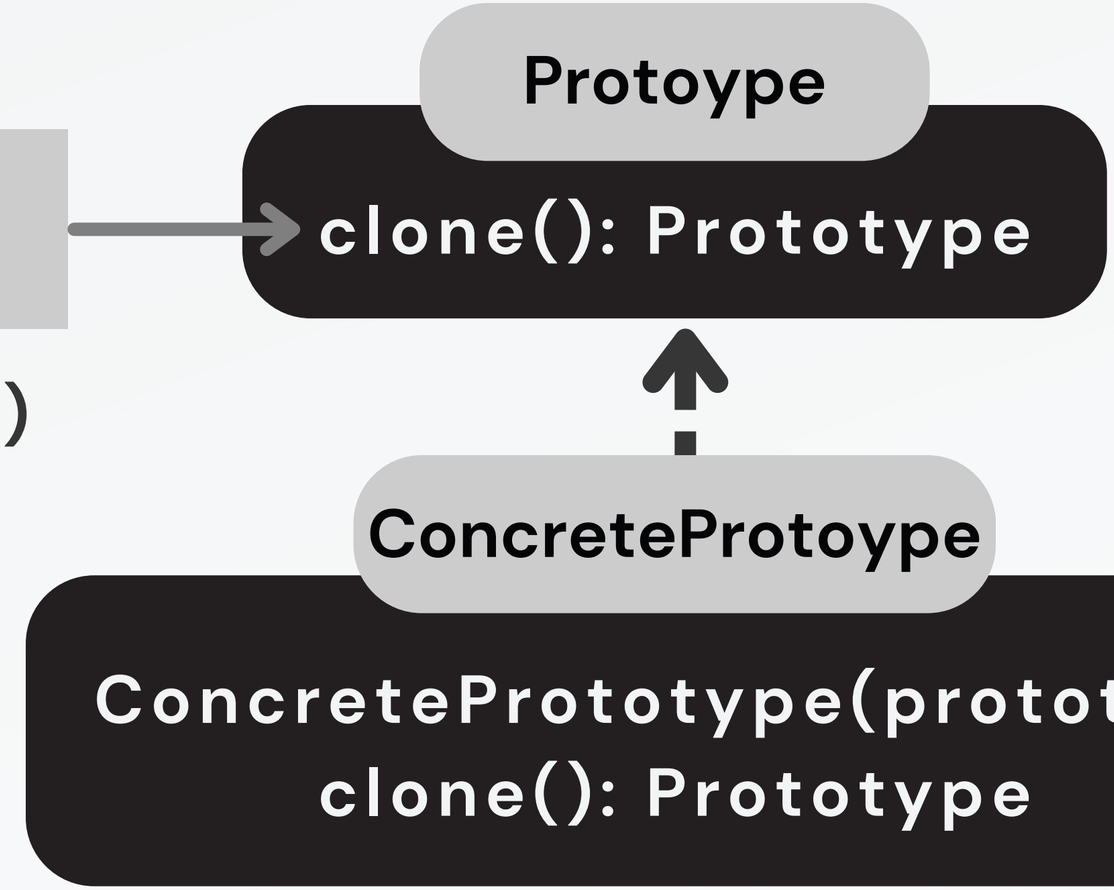
Solution



Prototype

Solution

Client




```
copy = existing.clone()
```

```
// Bad example of cloning objects
function mainBadExample(): void {
  let rectangle: Rectangle = new Rectangle(10, 20);
  let clonedRectangle: Rectangle = new Rectangle();
  clonedRectangle.setHeight(rectangle.getHeight());
  clonedRectangle.setWidth(rectangle.getWidth());
}
```


prototype.ts

```ts
interface Shape {
  // Use any because it will return different types.
  clone(): any;
  draw(): void;
}
```

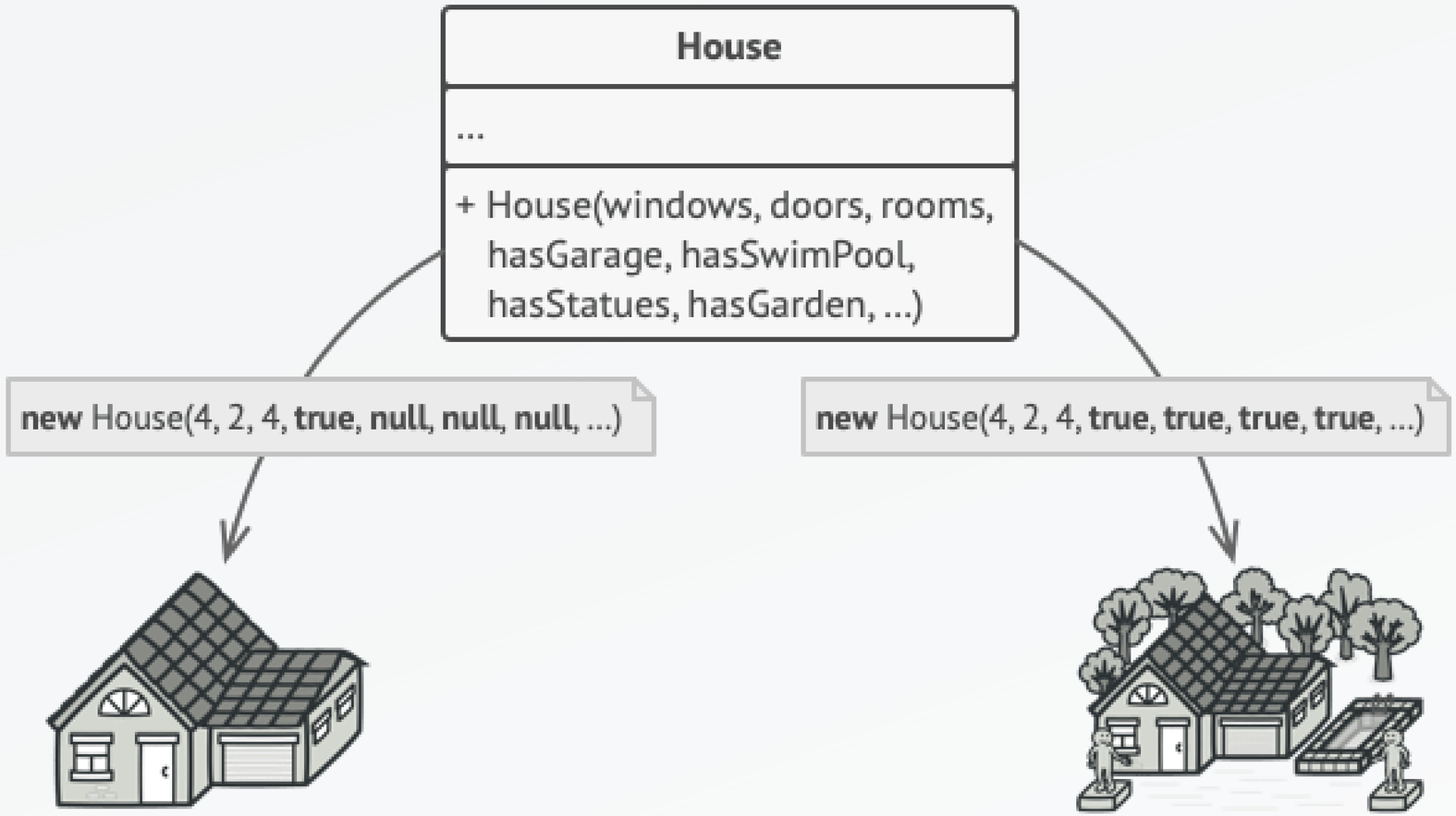

prototype.ts

Builder

Builder

Builder

Problem



Builder

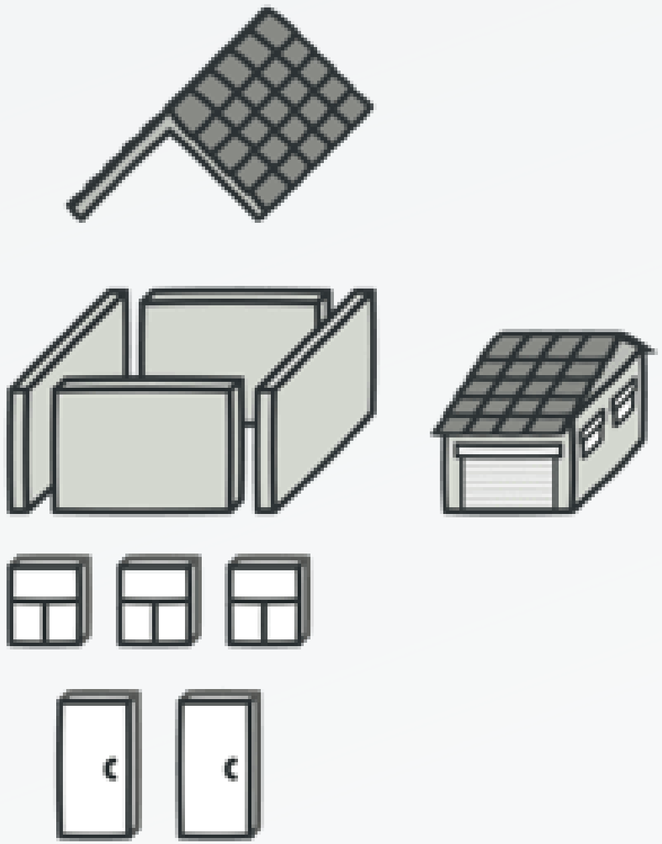
Solution

- Extract the object construction code.
- Organize object construction into steps.

- Extract a series of calls into a separate class called Director.

Builder

Solution



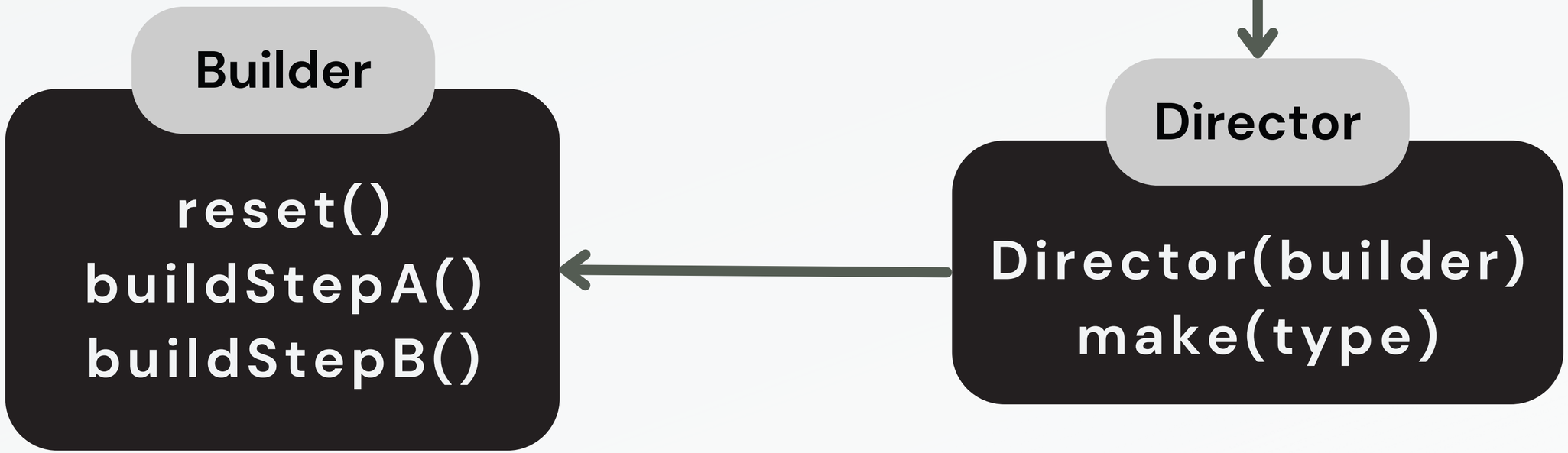
HouseBuilder

buildWalls()
buildDoors()
buildGarage()
buildPool()
getResult(): House

Builder

Solution

```
builder = new WoodBuilder()
director = new Director(builder)
director.make()
```



```
class House {
  // Some private properties declarations
  // Bad constructor with a lot of parameters
  constructor(walls: number, doors: number, rooms: number,
              pool: boolean, garage: boolean, garden: boolean) {
    this.walls = walls;
    this.doors = doors;
    this.rooms = rooms;
    this.pool = pool;
    this.garage = garage;
    this.garden = garden;
  }
}
```


builder.ts

```ts
interface Builder {
  setWalls(walls: number): this;
  setDoors(doors: number): this;
  setRooms(rooms: number): this;
  setPool(pool: boolean): this;
  setGarage(garage: boolean): this;
  setGarden(garden: boolean): this;
  build(): House;
}
```

builder.ts

Structural Patterns

Structural Patterns

- They deal with the relationships between objects.
    - How to assemble objects and classes into larger structures.
    - Keep it flexible.
    - Keep it efficient.

Structural Patterns

01 Adapter

02 Bridge

03 Composite

04 Decorator

05 Facade

06 Flyweight

07 Proxy

08 Delegation

09 Extension object

10 Module

Structural Patterns

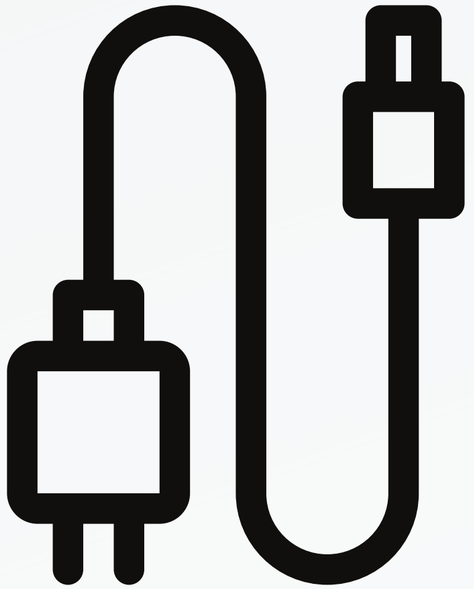
01 Adapter

03 Composite

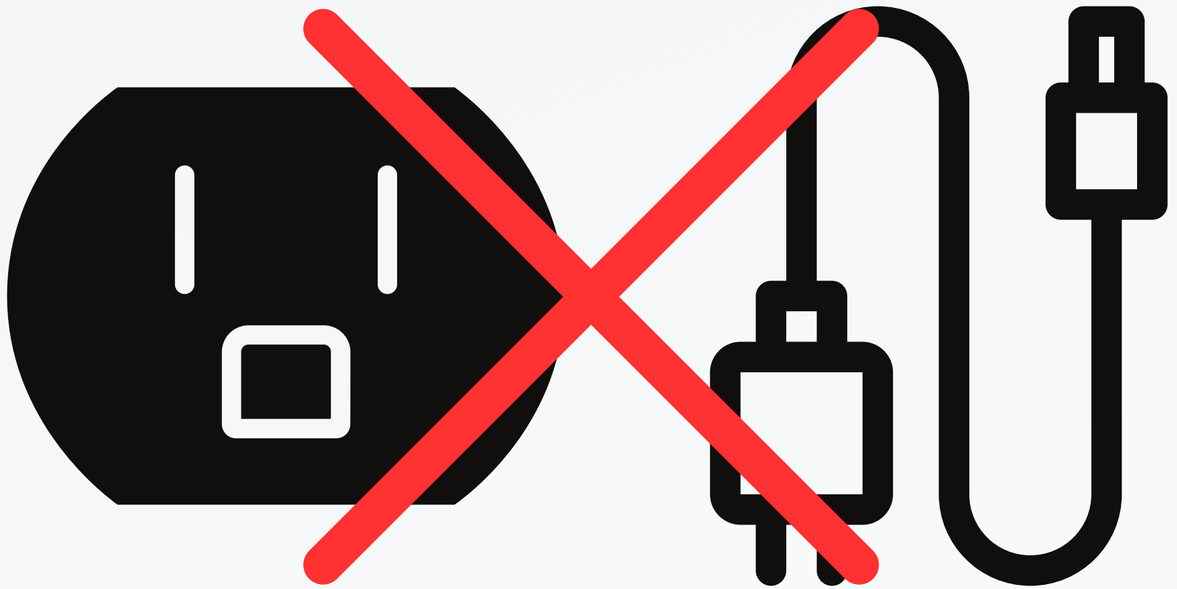
05 Facade

07 Proxy

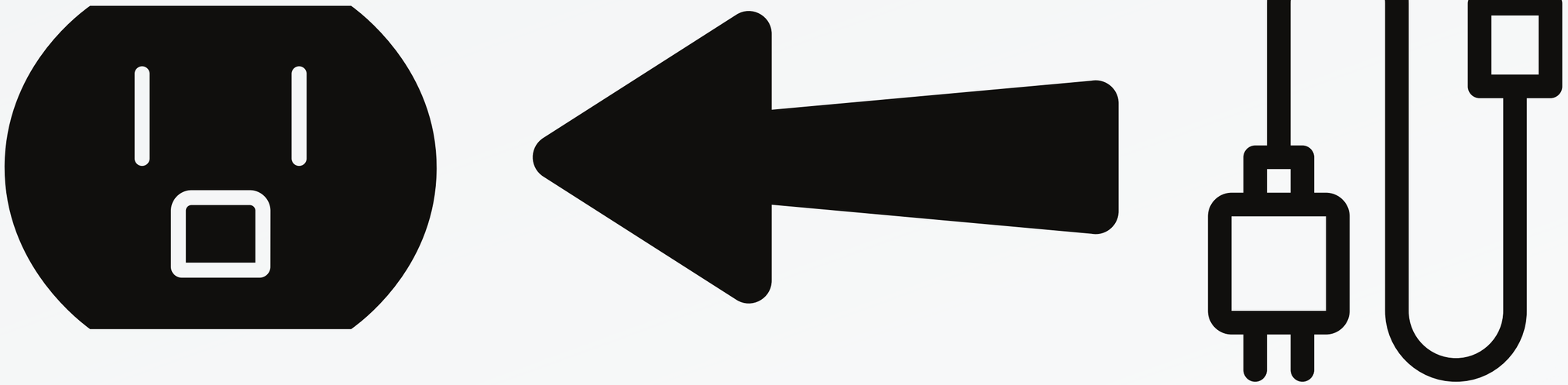
Adapter



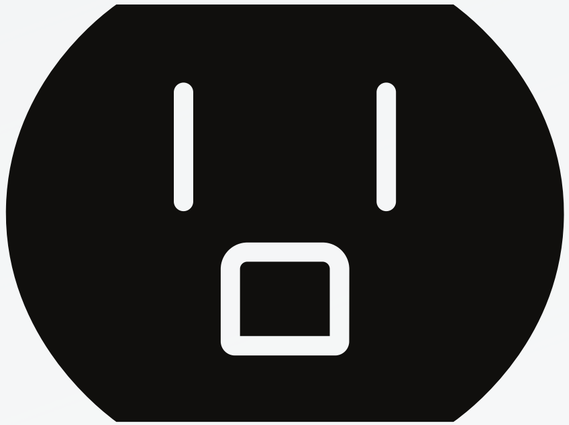
Adapter



Adapter

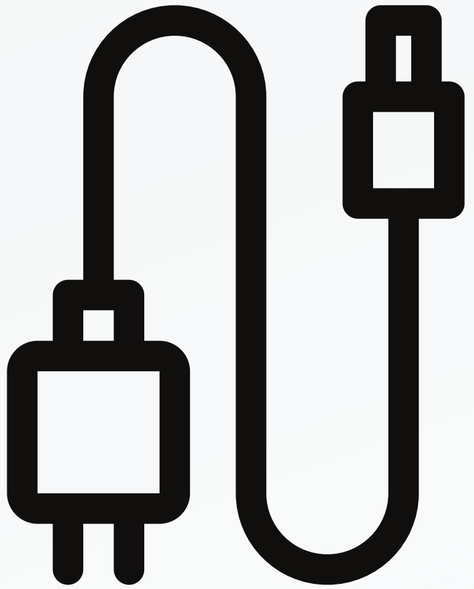


Adapter



Adapter

(...)



Adapter

Problem

Service

Adapter

(...)

Interface

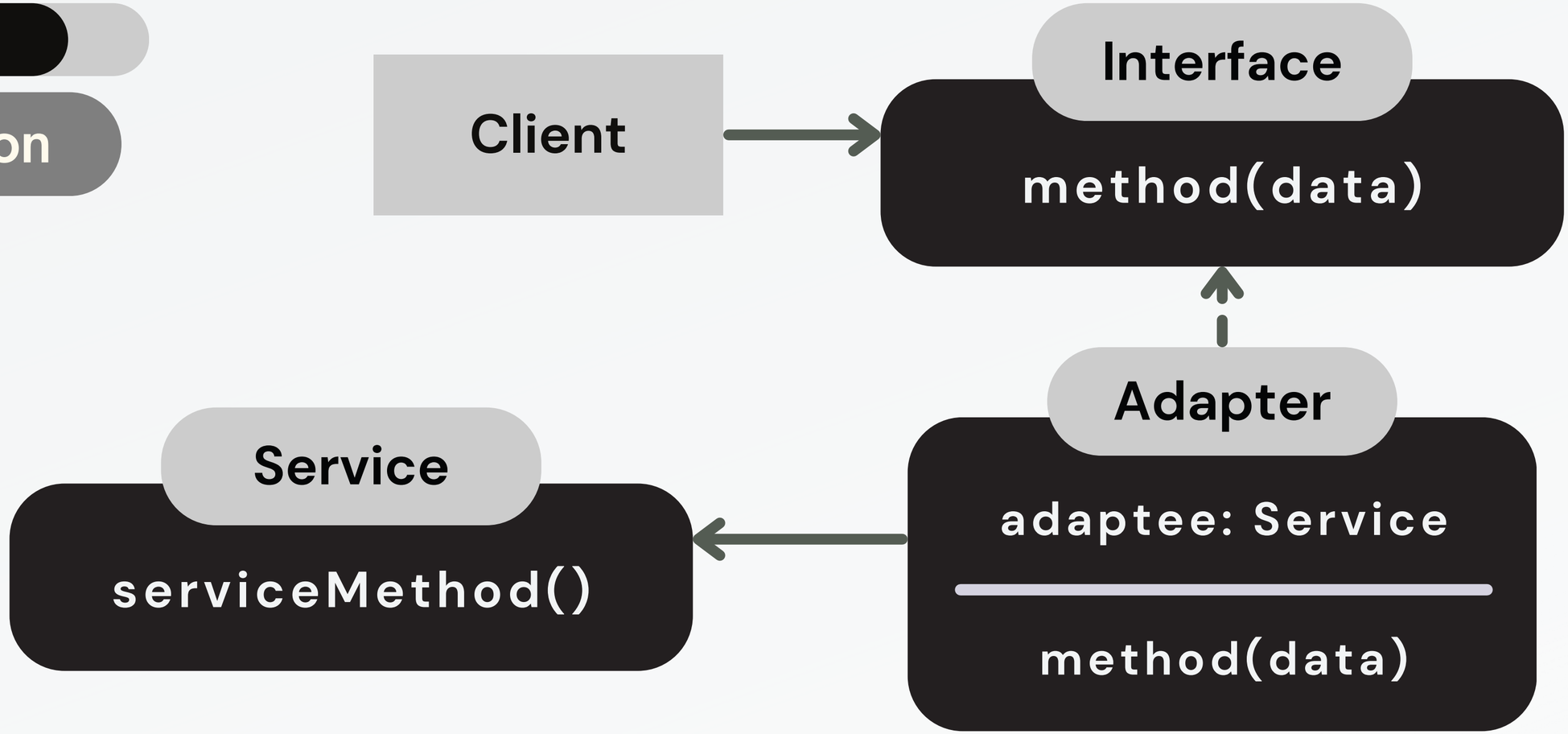
Adapter

Solution

- Wrap one of the objects to hide the comlexity of conversion.

Adapter

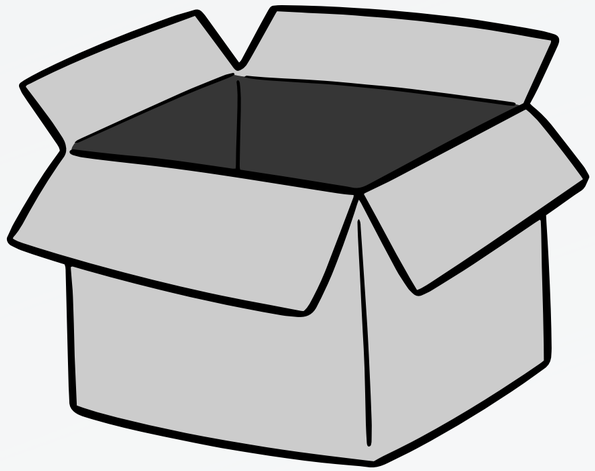
Solution



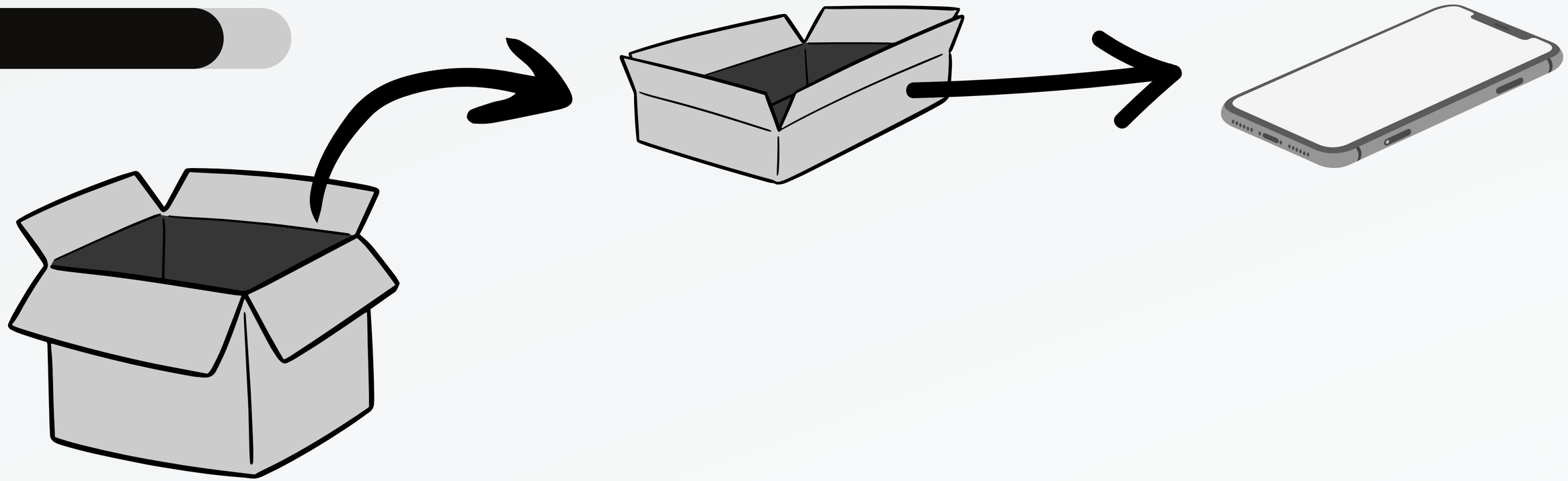
```ts
interface Printer {
  print(): void;
}

class LegacyPrinter {
  printString(): void {
    console.log('Printing with Legacy Printer');
  }
}
```


adapter.ts

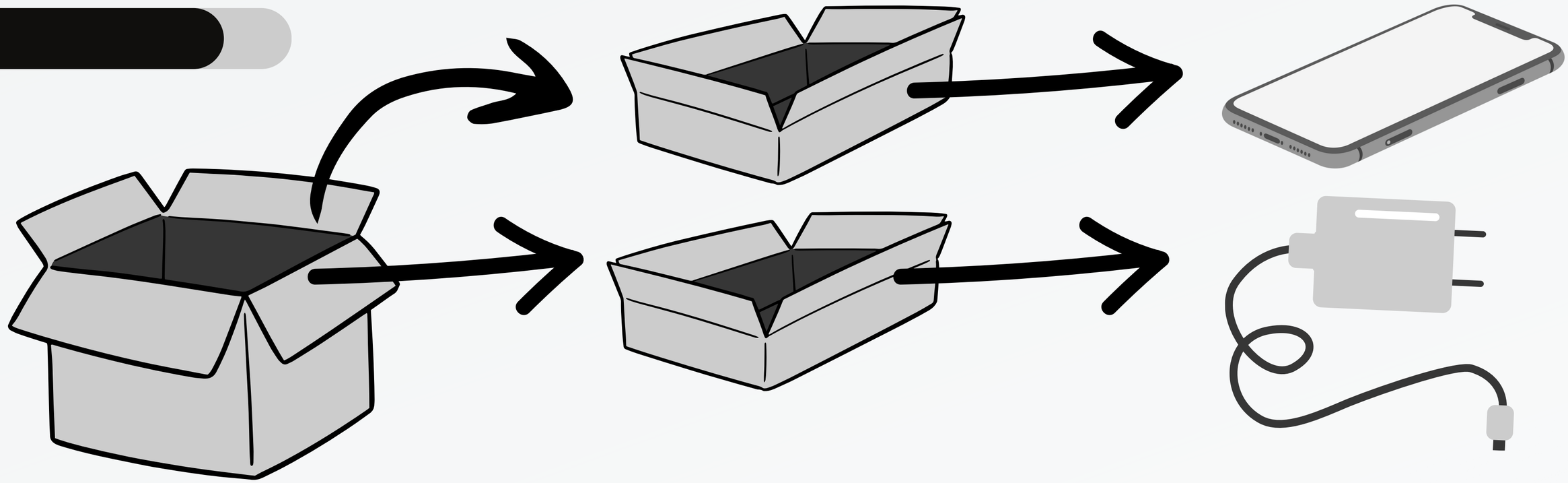
Composite



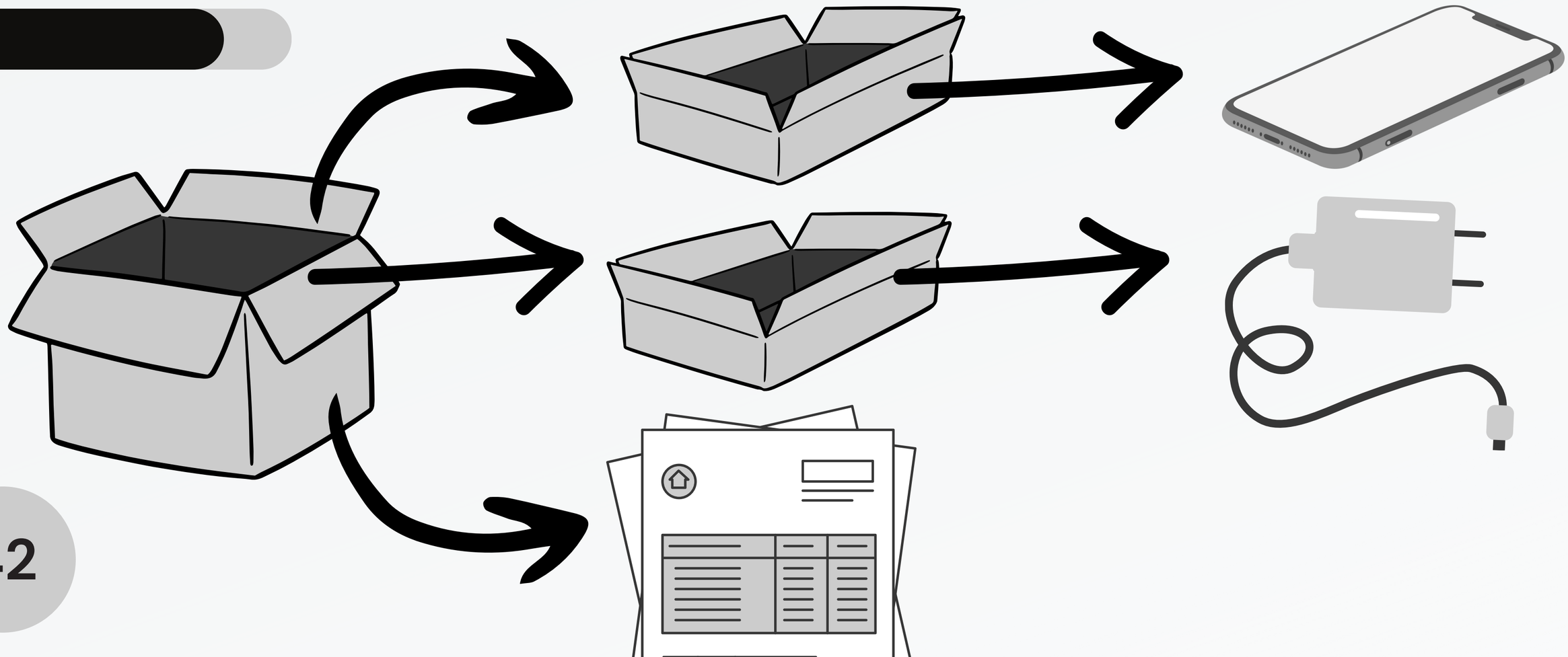
Composite



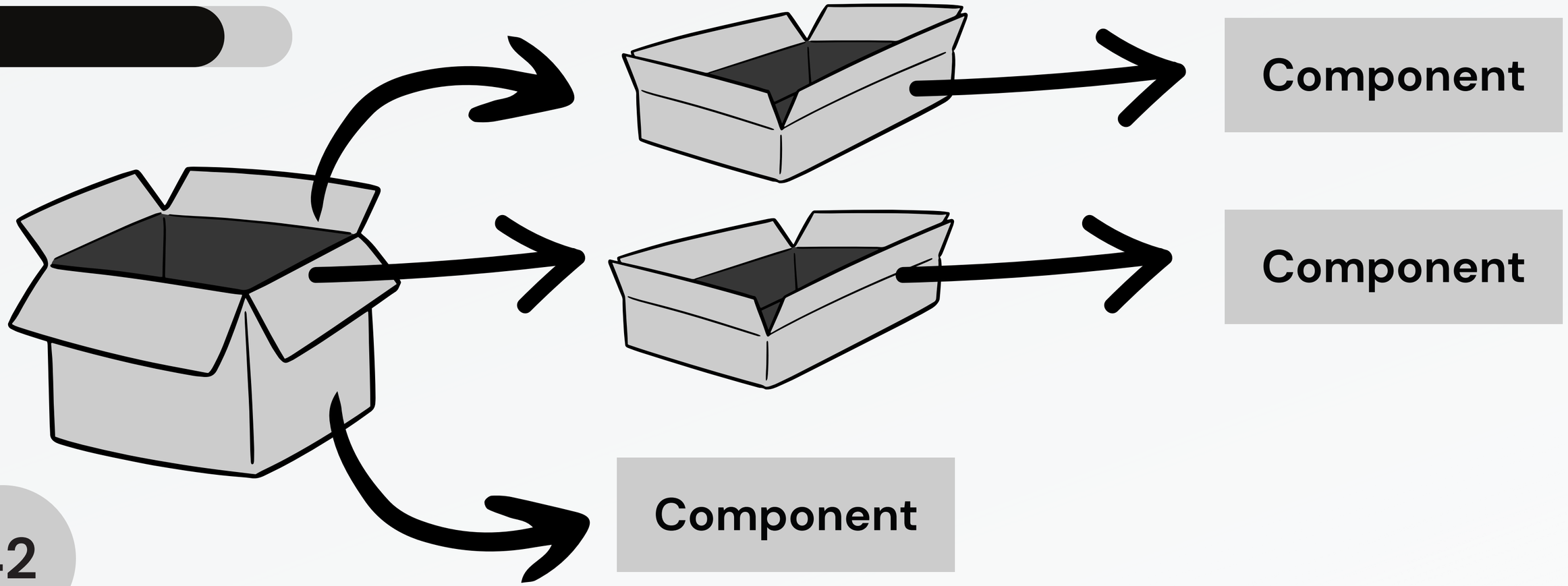
Composite



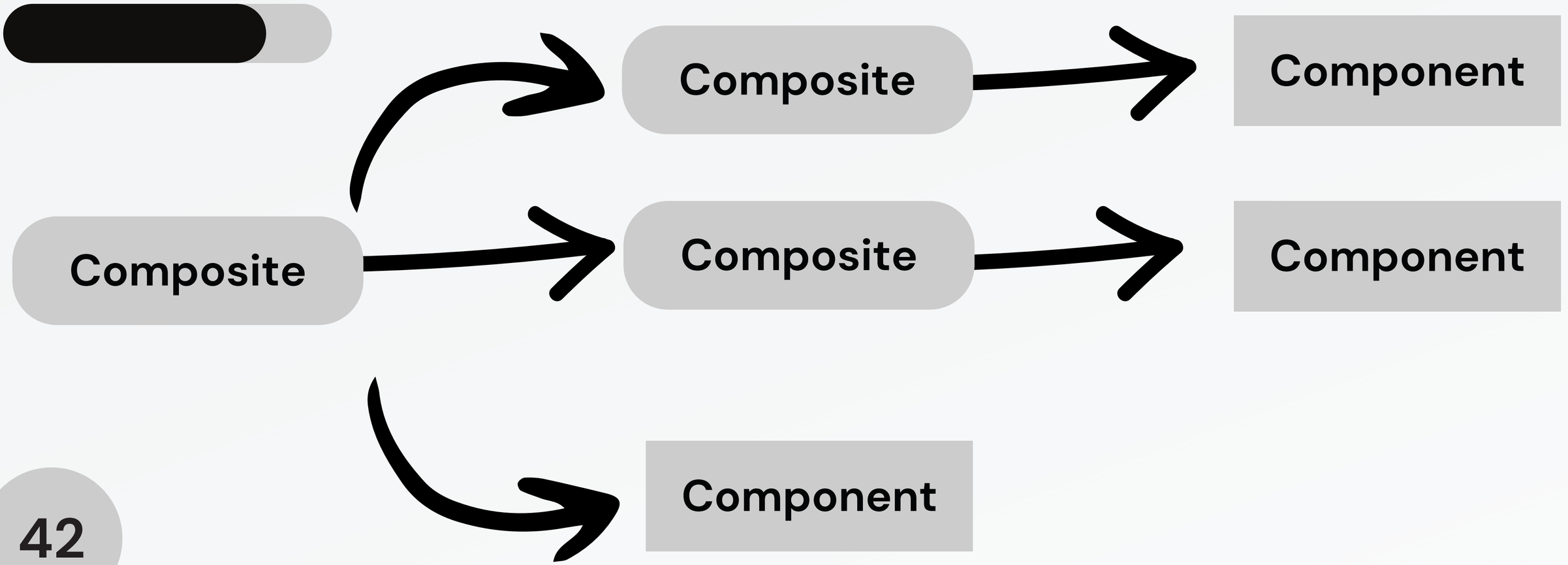
Composite



Composite

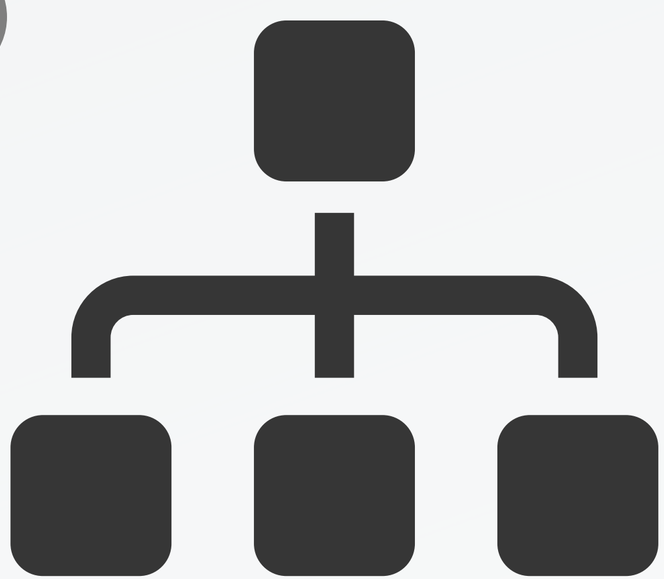


Composite


Composite
Composite
Component
Composite
Composite
Component
Component

Composite

Problems



- You have to know:
  - The classes you're going through.
  - The nesting levels.
  - Details.

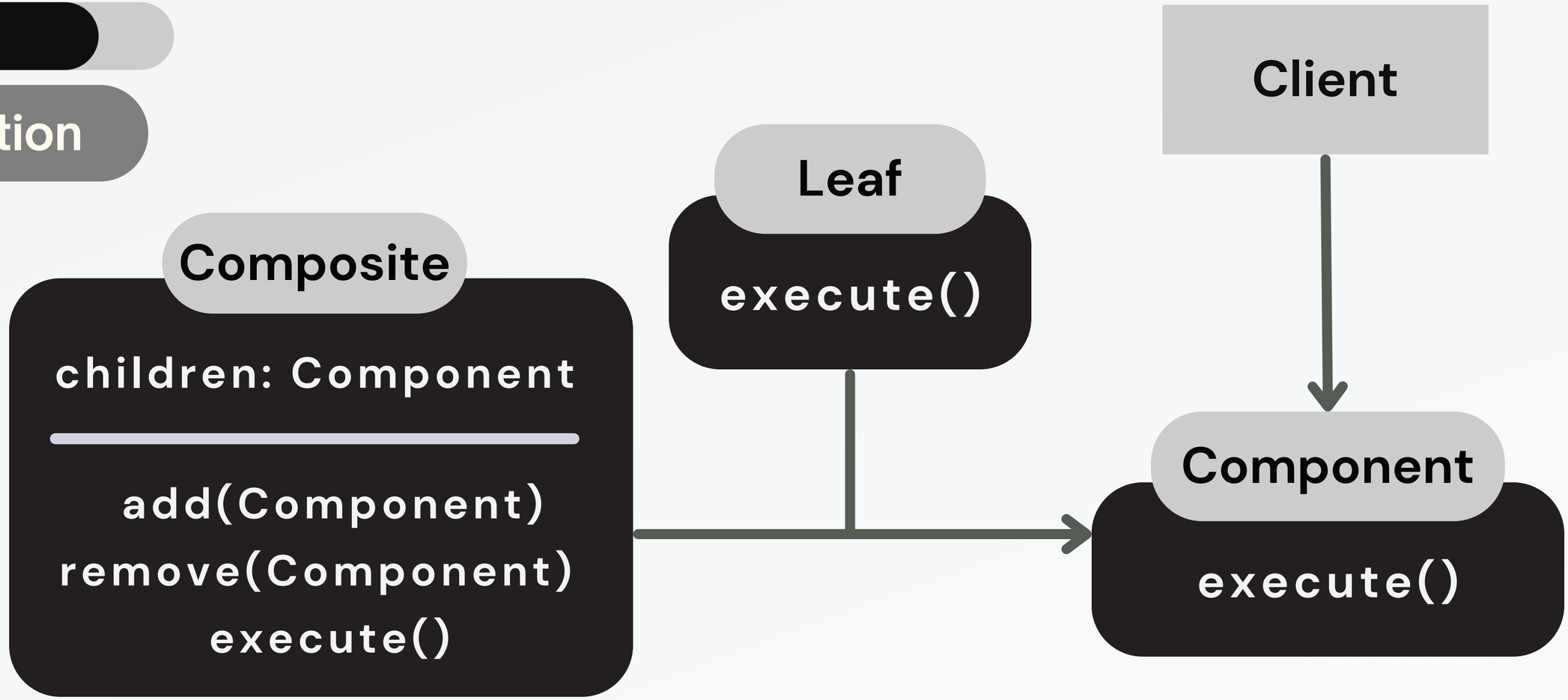
Composite

Solution

Work with
Products and
Boxes through a
common interface.

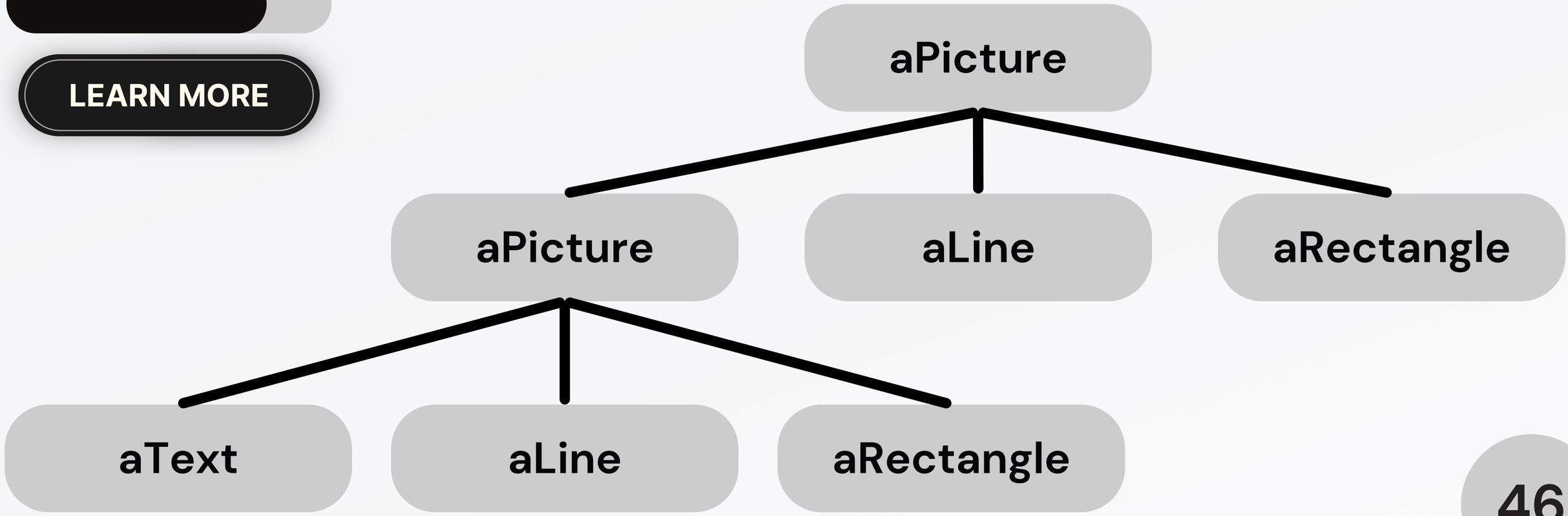
Composite

Solution



Composite

LEARN MORE

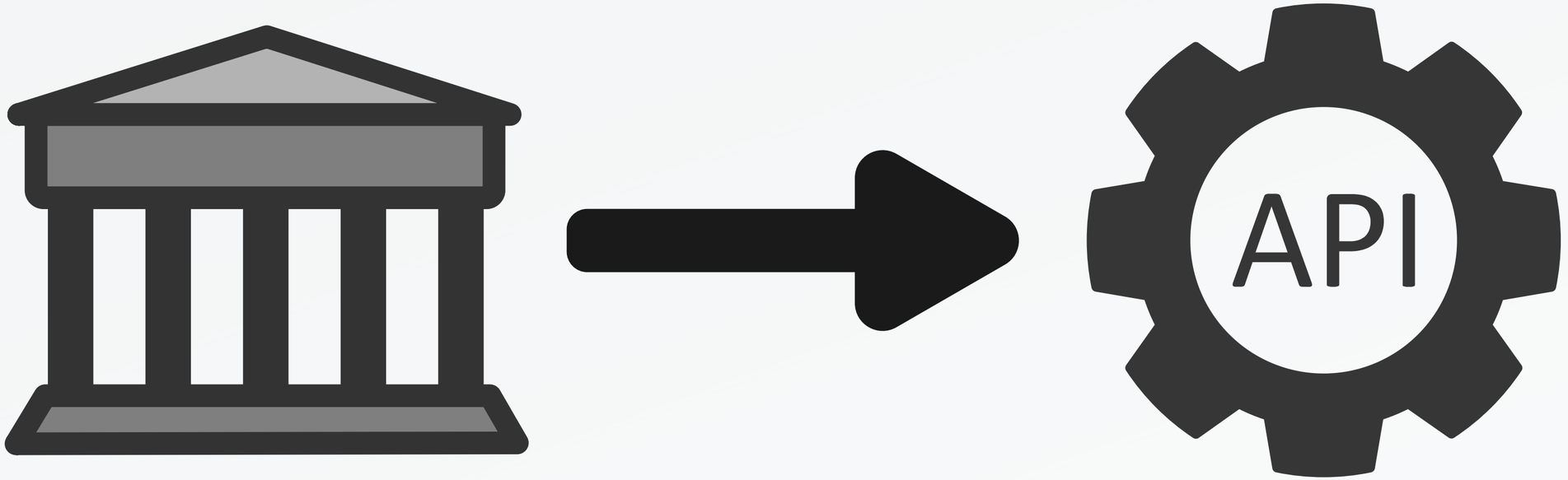


Facade

Complex Subsystem

Facade

Facade



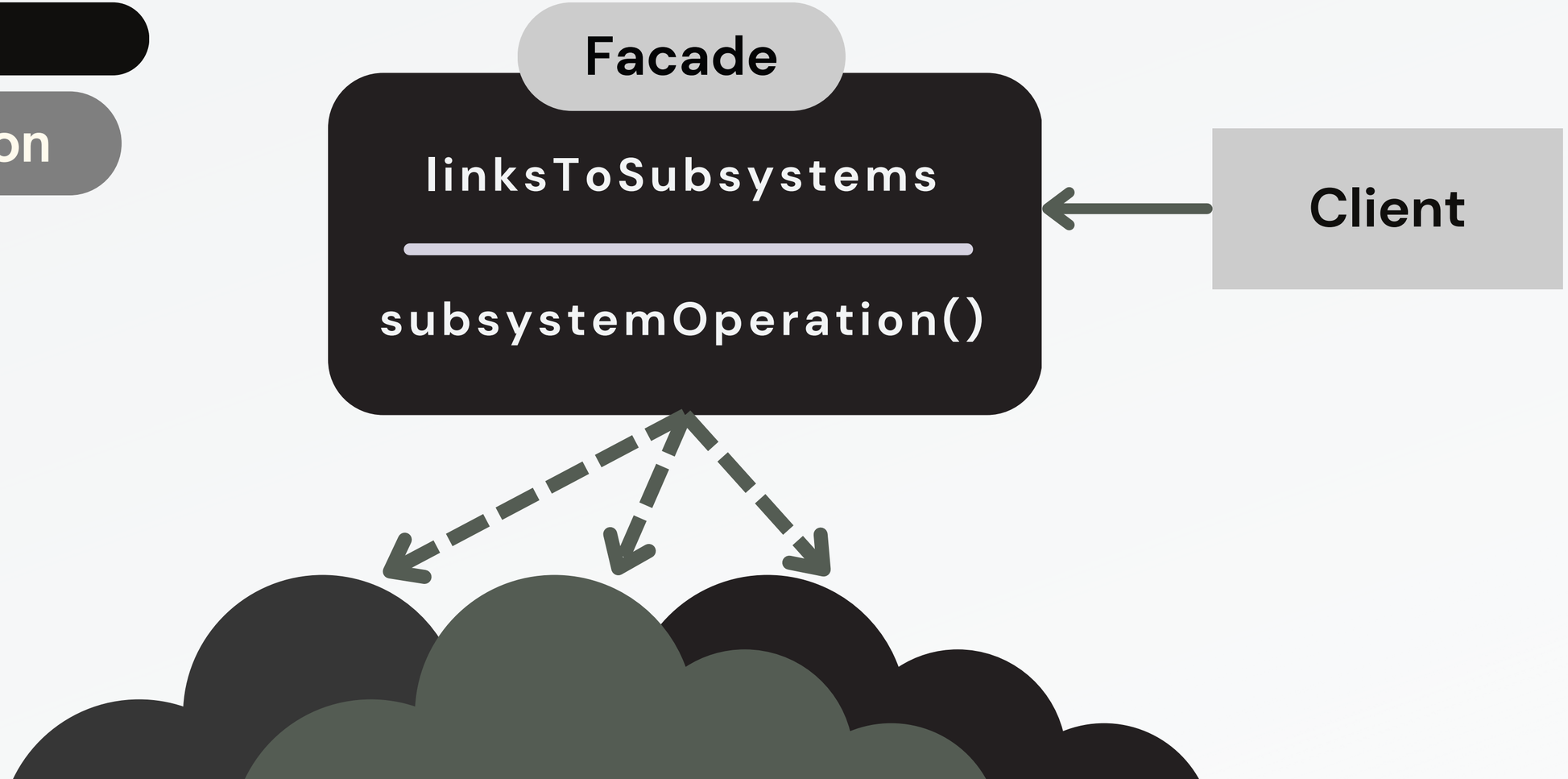
Facade

Solution

- Simple interface to a complex subsystem.
- Limited functionality.
- Features that clients really care about.

Facade

Solution



Facade

- Cloud service system (Complex system).
- Only want to upload files (Simple interface).

```
// Client code that uses the complex system directly.
function main() {
  const CLOUD_SERVICE: CloudProviderService = new CloudProviderService();
  if (!CLOUD_SERVICE.isLoggedIn()) {
    CLOUD_SERVICE.logIn();
  }
  const CONVERTED_FILE: string = CLOUD_SERVICE.convertFile("file.txt");
  CLOUD_SERVICE.uploadFile(CONVERTED_FILE);
  const FILE_LINK: string = CLOUD_SERVICE.getFileLink(CONVERTED_FILE);
}
```

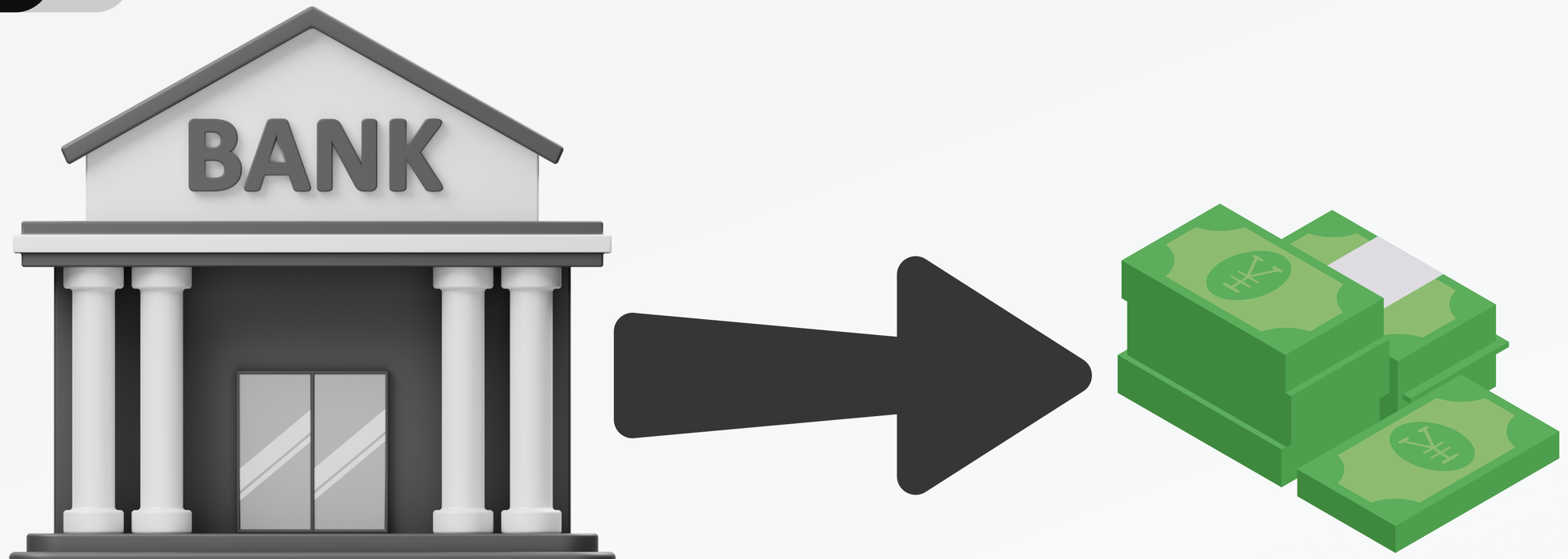

facade.ts

```
// Complex System class that we want to simplify using a facade.
class CloudProviderService {
  public isLoggedIn(): boolean {...}
  public logIn(): void {...}
  public convertFile(file: string): string {...}
  public uploadFile(file: string): void {...}
  public getFileLink(file: string): string {...}
}
```


facade.ts

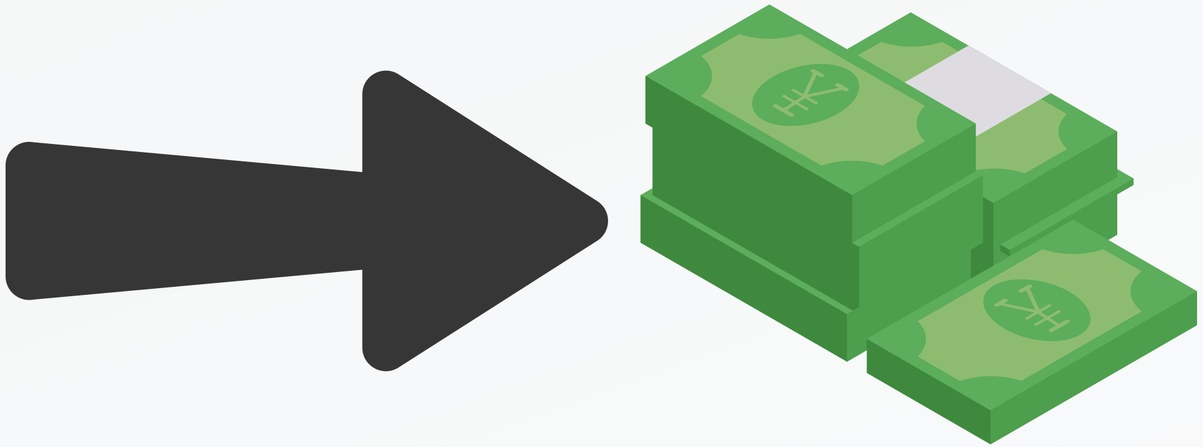
Proxy

Proxy

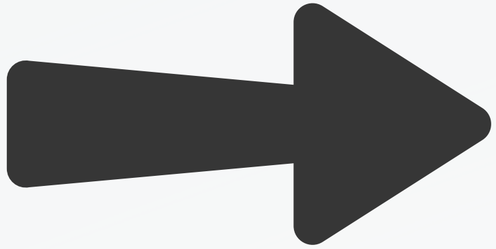


Proxy

Proxy

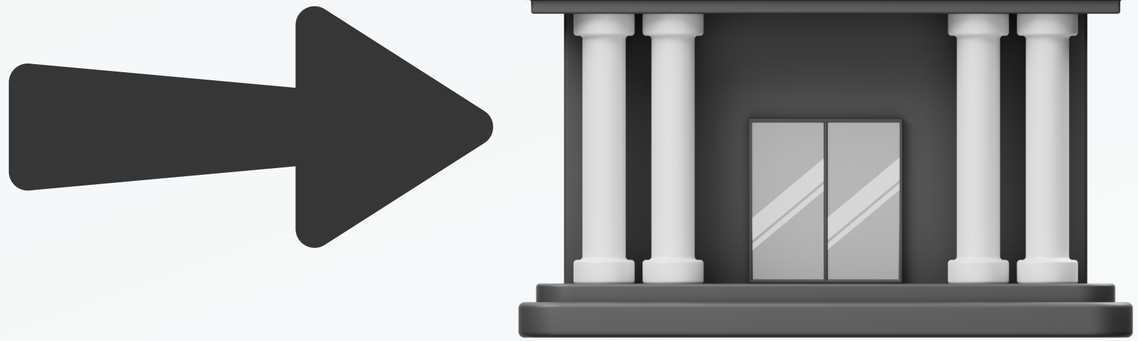


Proxy



Proxy

Proxy



Proxy

Problem

- Control access to an object.
    - ¿How?
        - ¿Lazy initialization?
        - ¿Code directly into the object’s class?

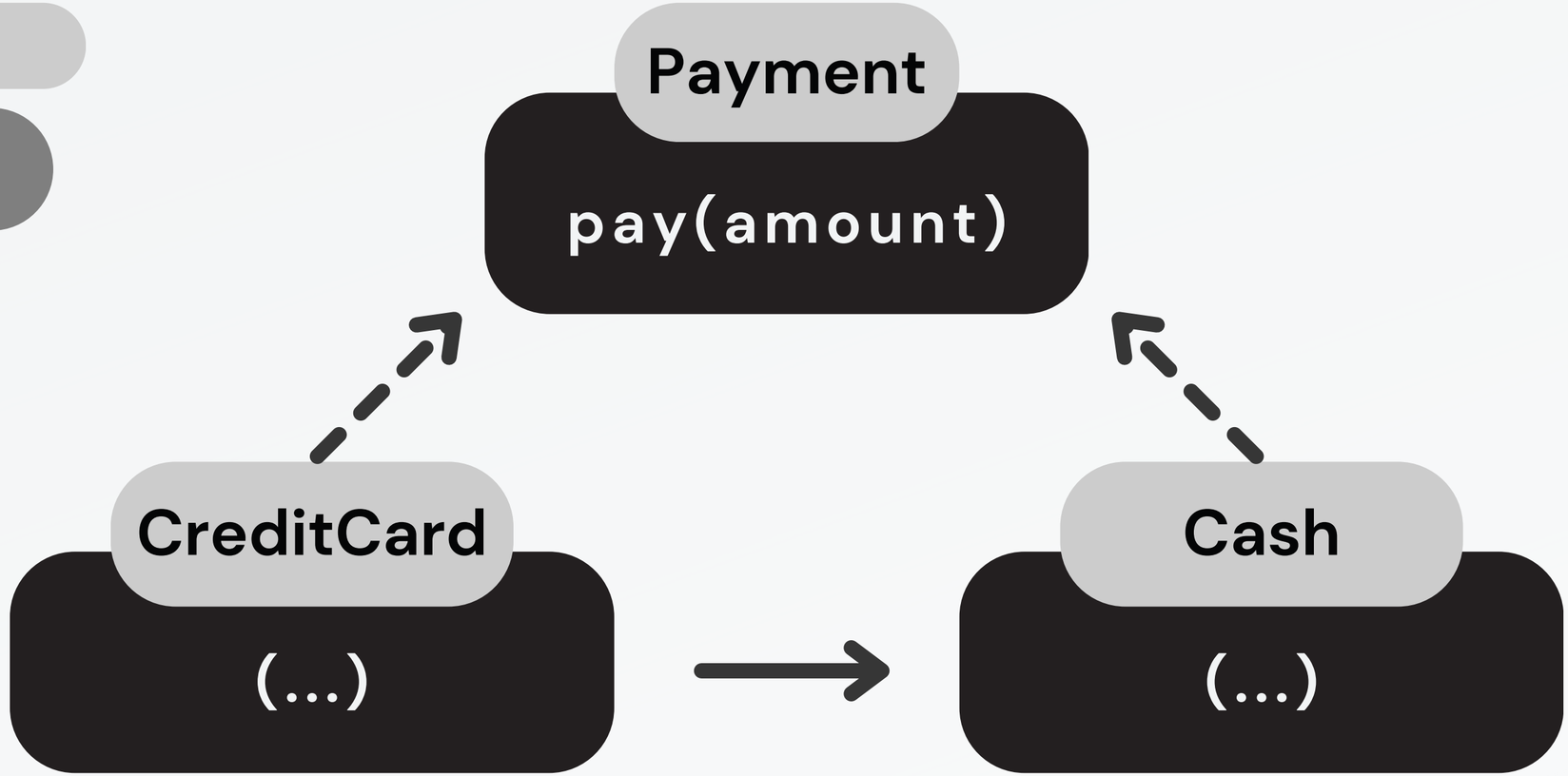
Proxy

Solution

- Create a *proxy* class with the same interface as the original service object.

Proxy

Solution



Proxy


ServiceInterface
operation()
Proxy
realService: Service
Proxy(Service)
checkAccess()
operation()
Service
operation()

Proxy

- Simple Person class (with name and age).
- Proxy to validate values.

```ts
// Defines the common interface for both Proxy and Person.
interface PersonInfo {
  getName(): string;
  getAge(): number;
  setName(name: string): void;
  setAge(age: number): void;
}
```


proxy.ts

Behavioral Patterns

Behavioral Patterns

- It deals with communication between objects.
    - Assignment of responsabilities.

Behavioral Patterns

01 Chain of responsibility

02 Command

03 Iterator

04 Null object

05 Mediator

06 State

07 Visitor

08 Observer

09 Strategy

10 Template method

Behavioral Patterns

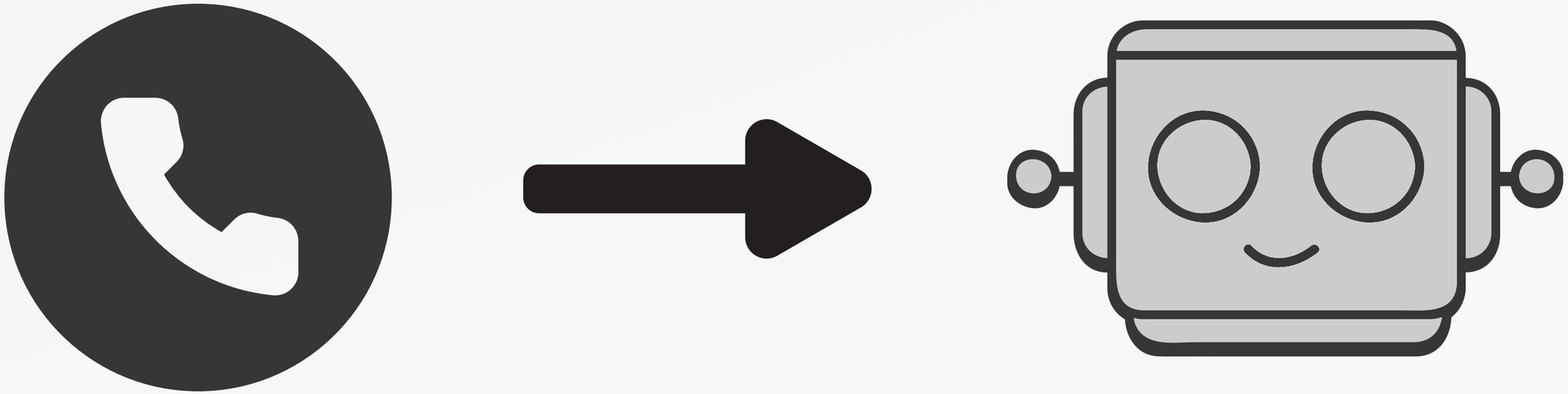
01 Chain of responsibility

06 State

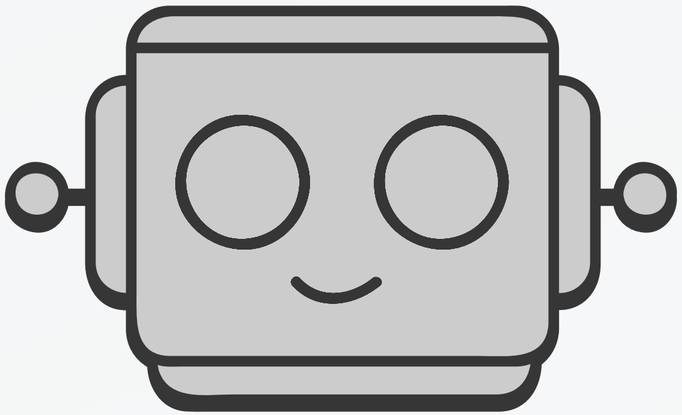
03 Iterator

05 Mediator

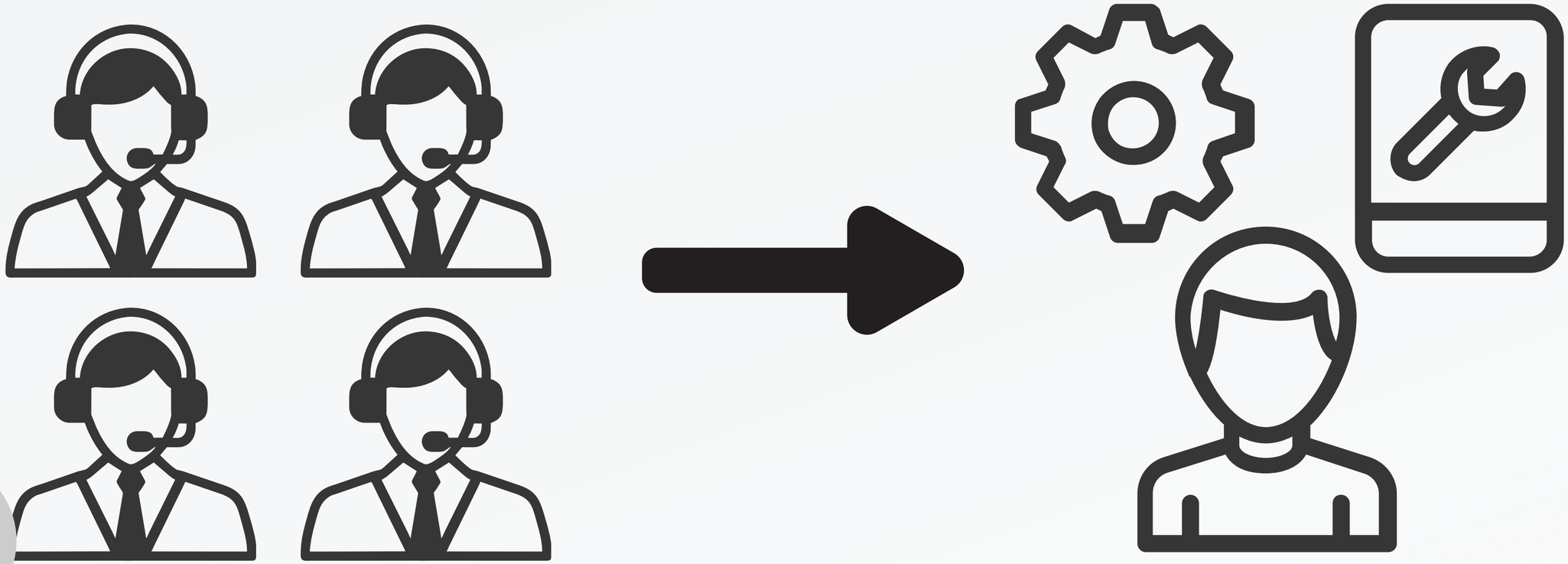
Chain of responsability



Chain of responsability

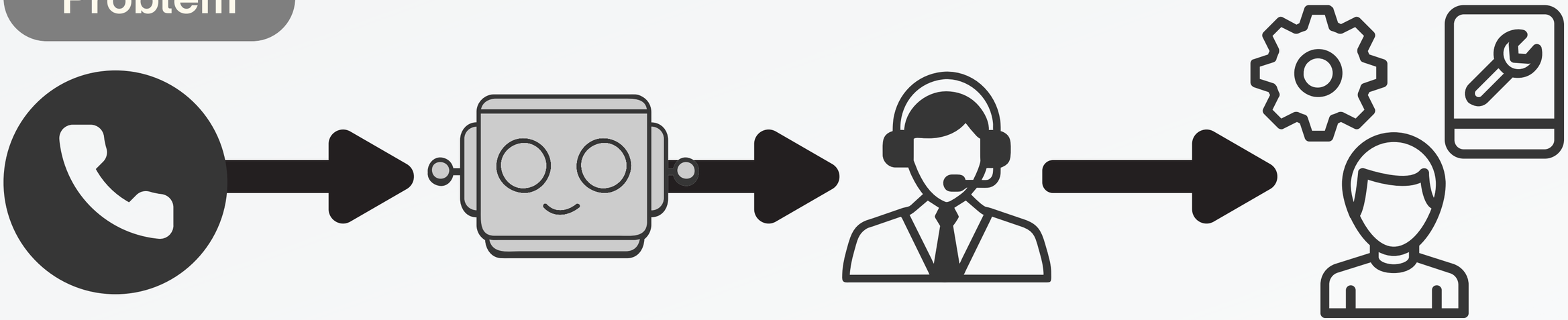


Chain of responsability

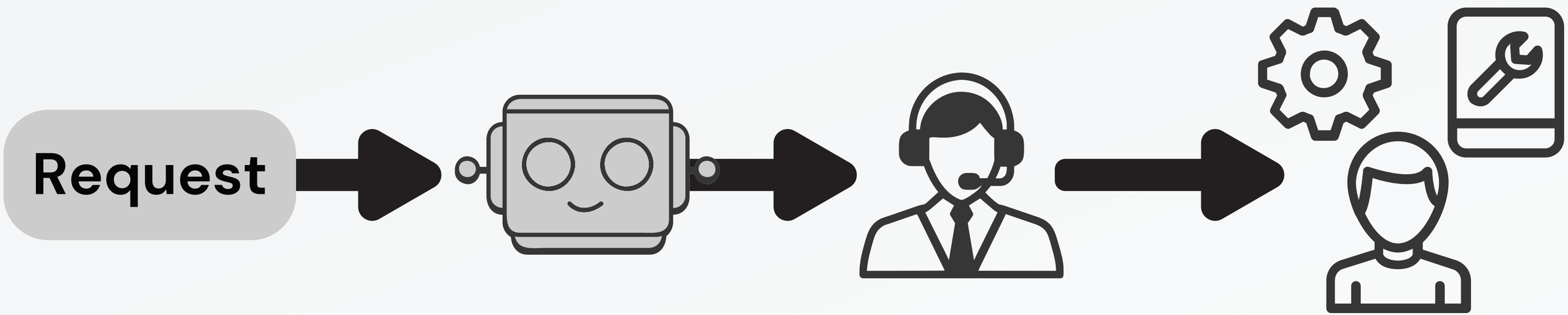


Chain of responsability

Problem



Chain of responsability

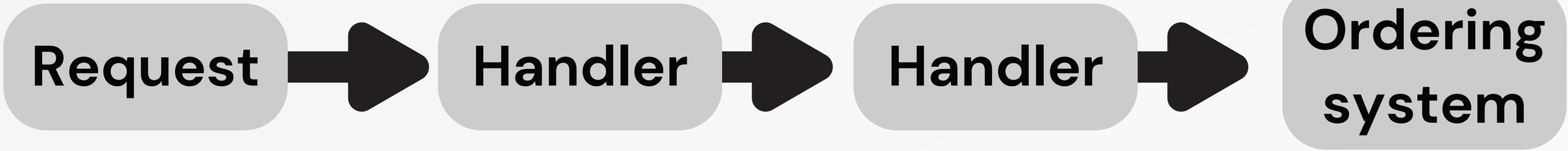


Chain of responsability

Request → Handler → Handler → 

Chain of responsability

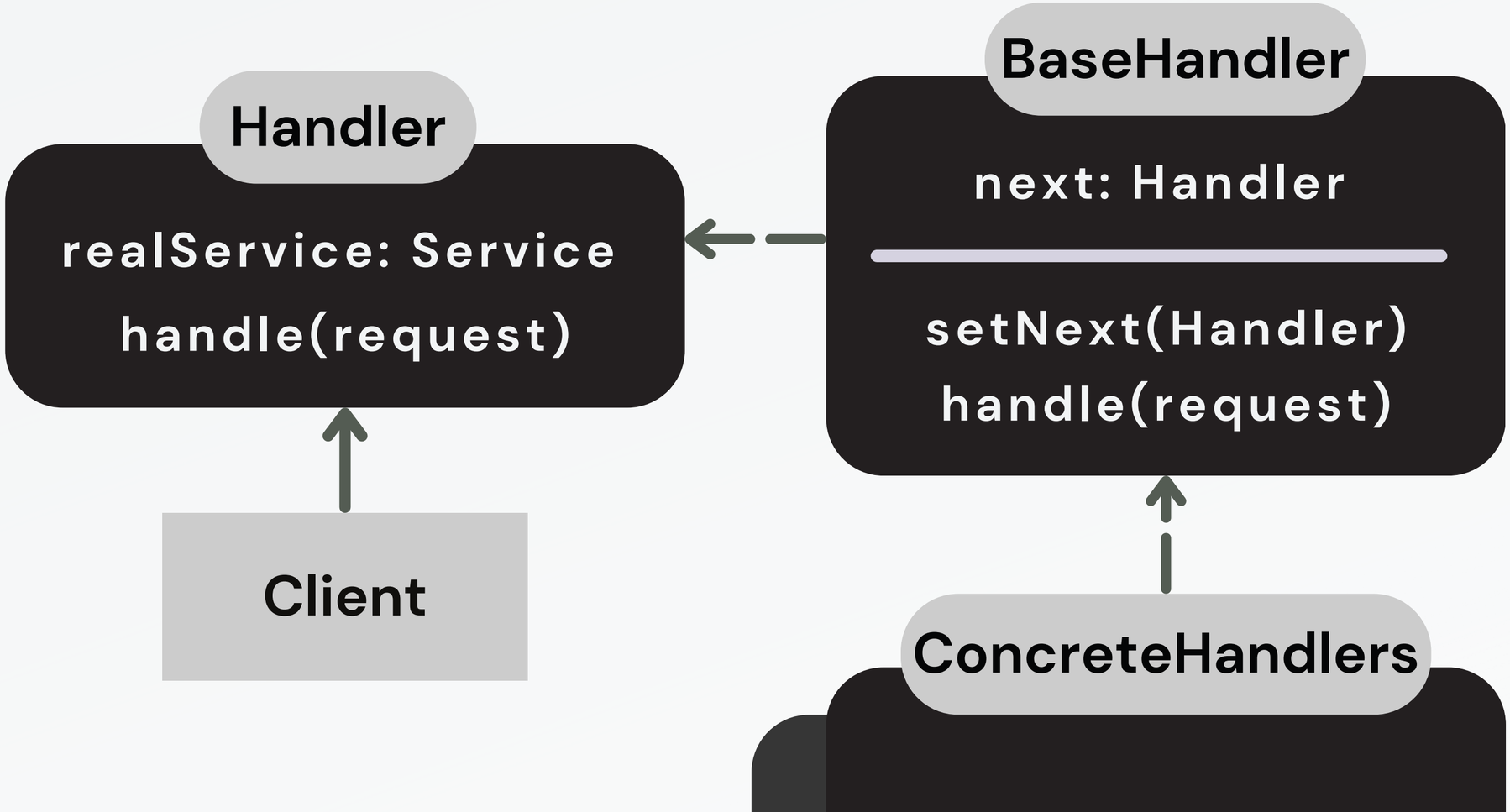
Solution



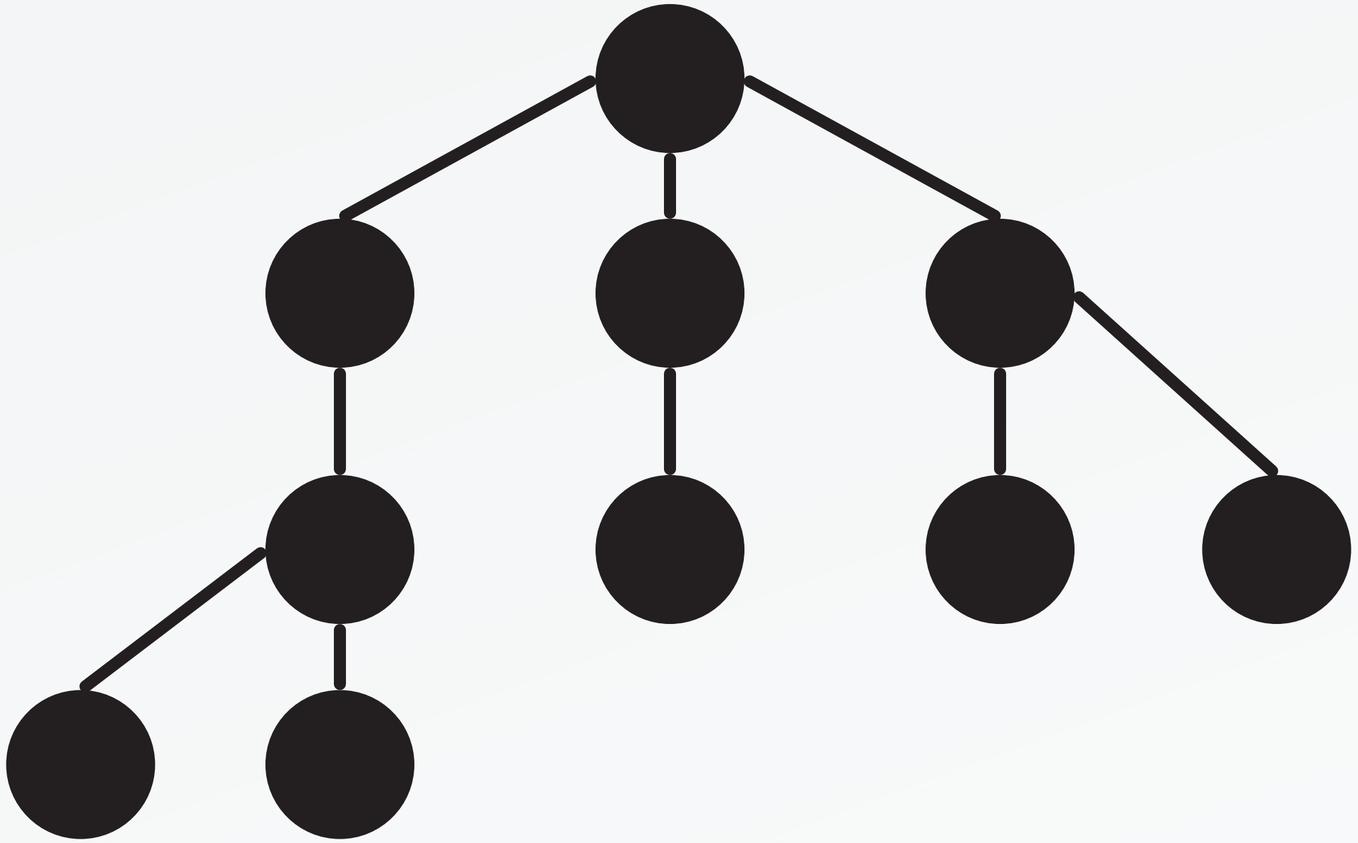
chain.ts

Chain of responsability

Solution

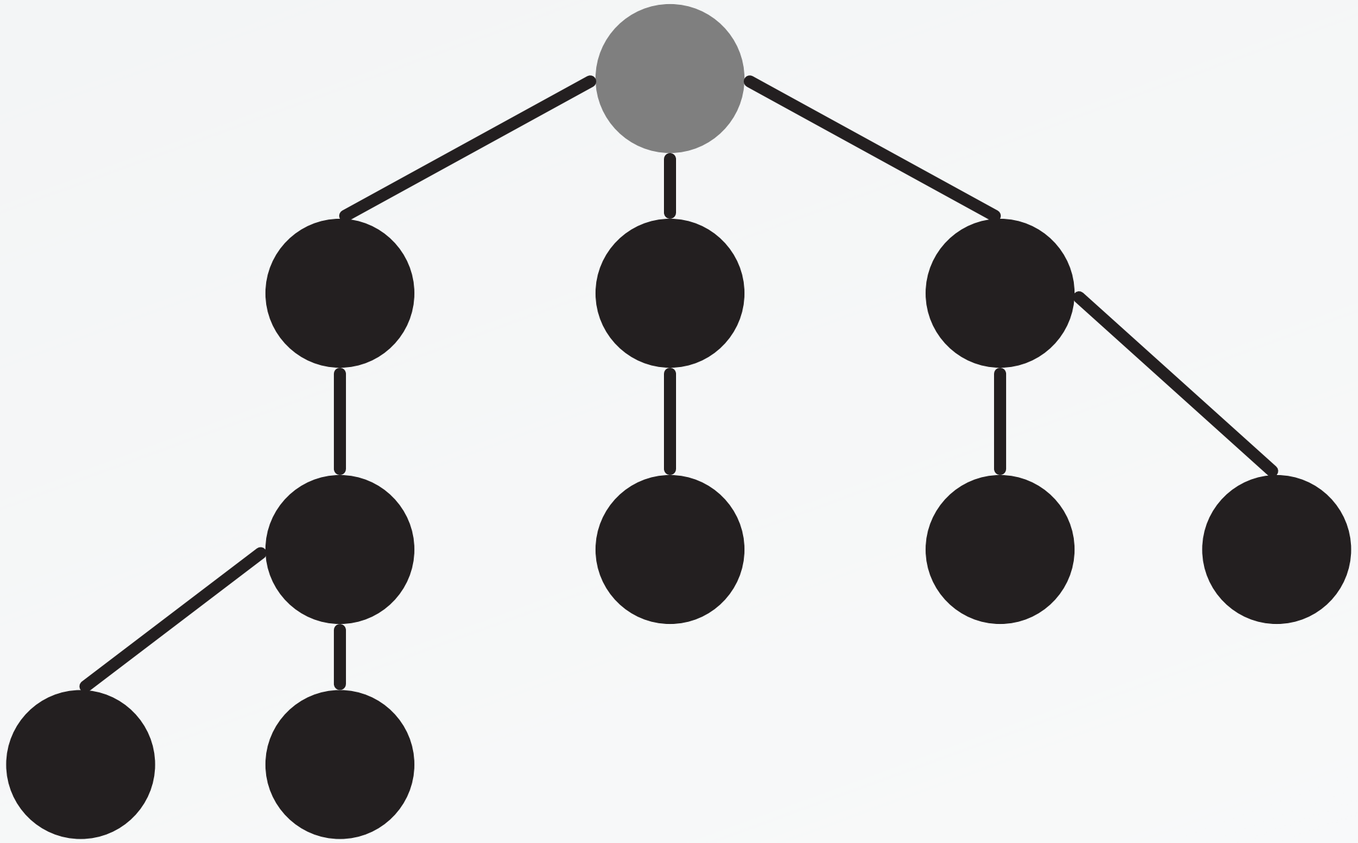


Iterator



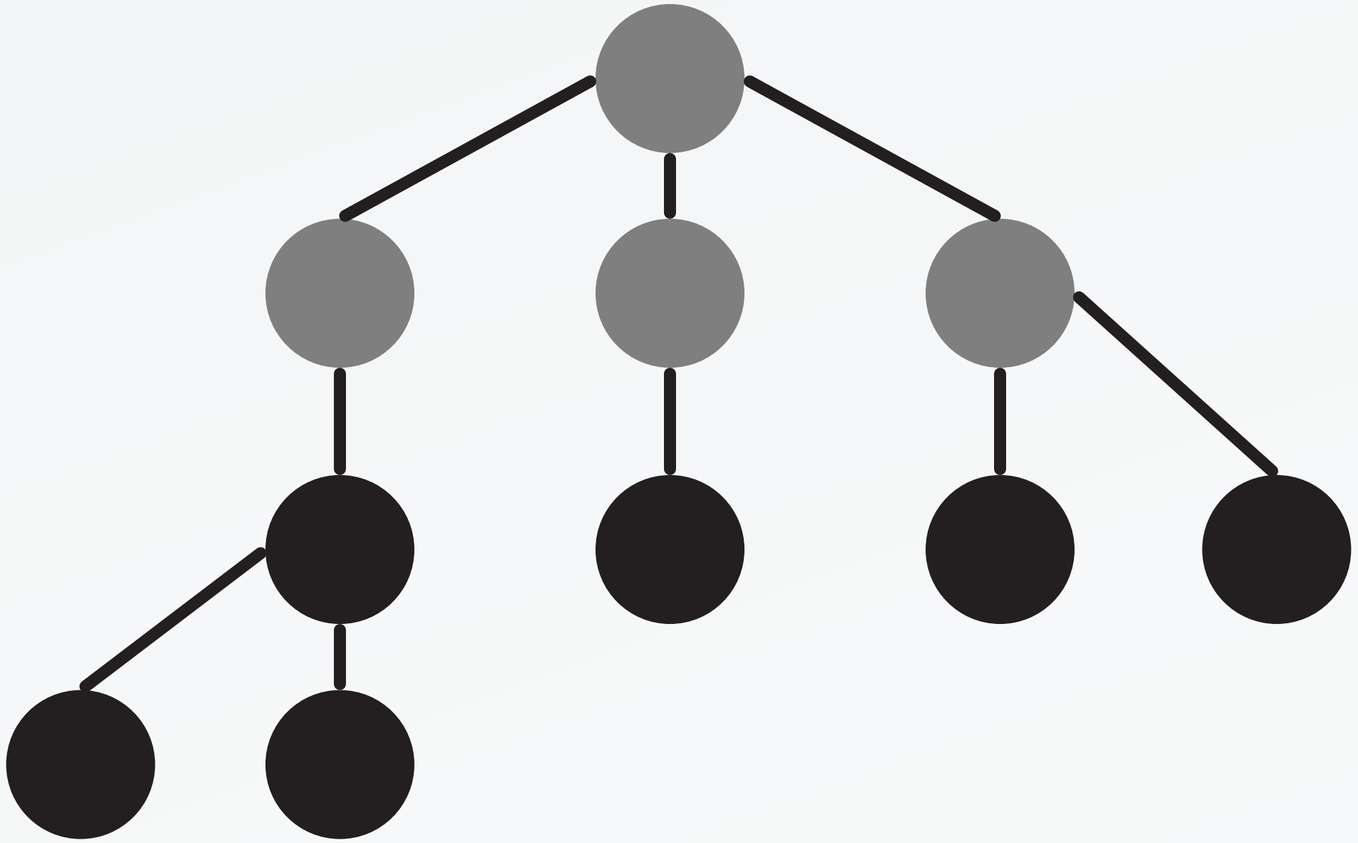
Iterator

Problem



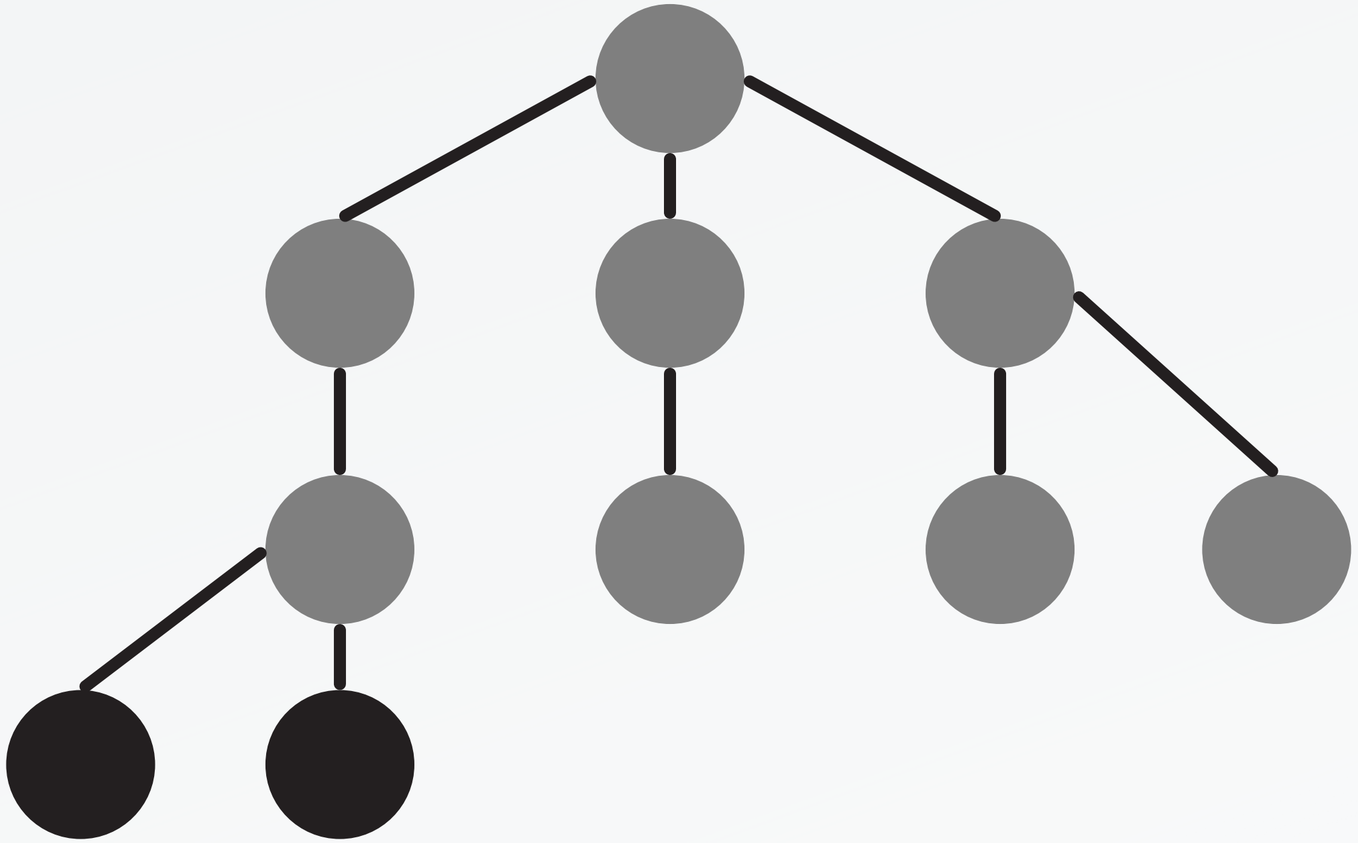
Iterator

Problem



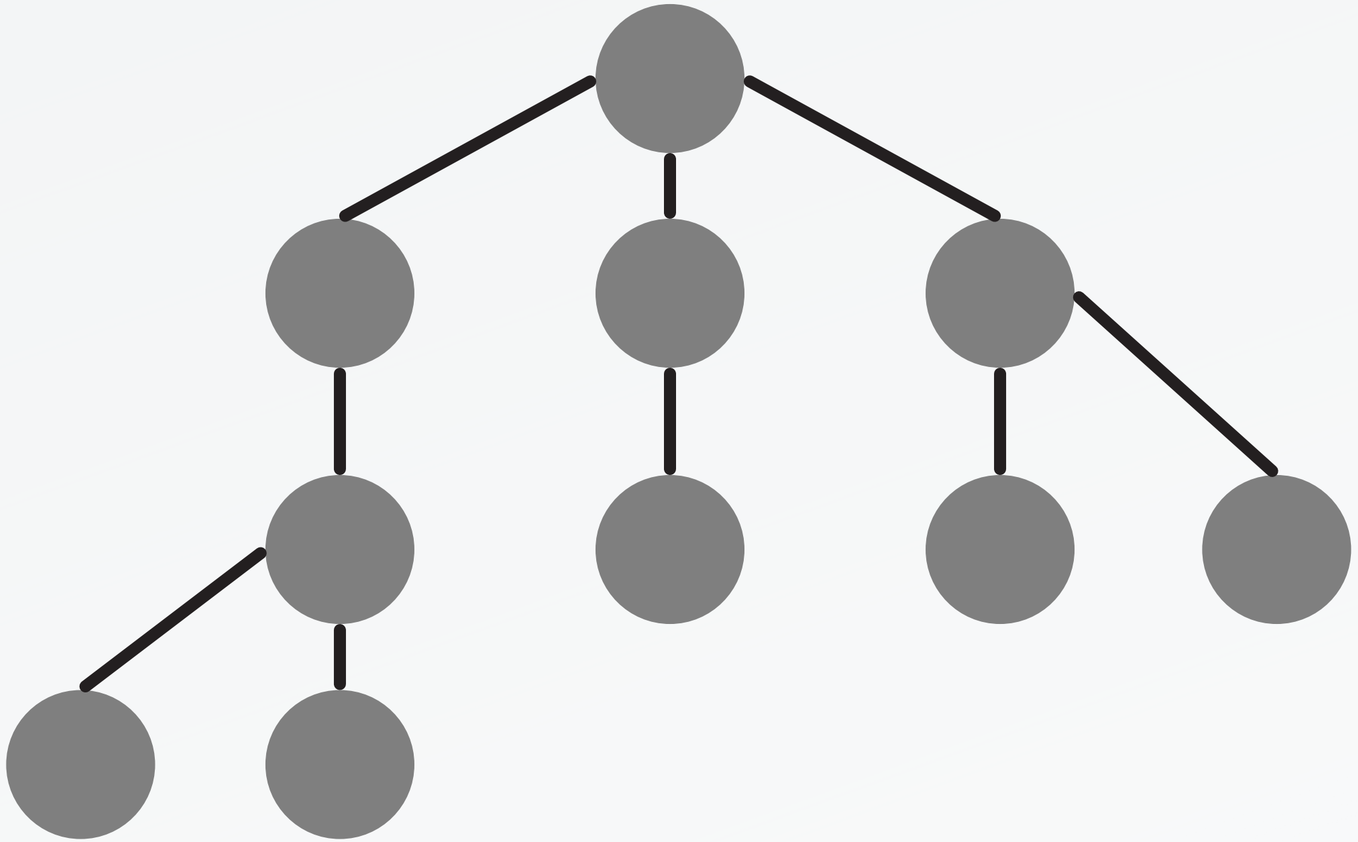
Iterator

Problem



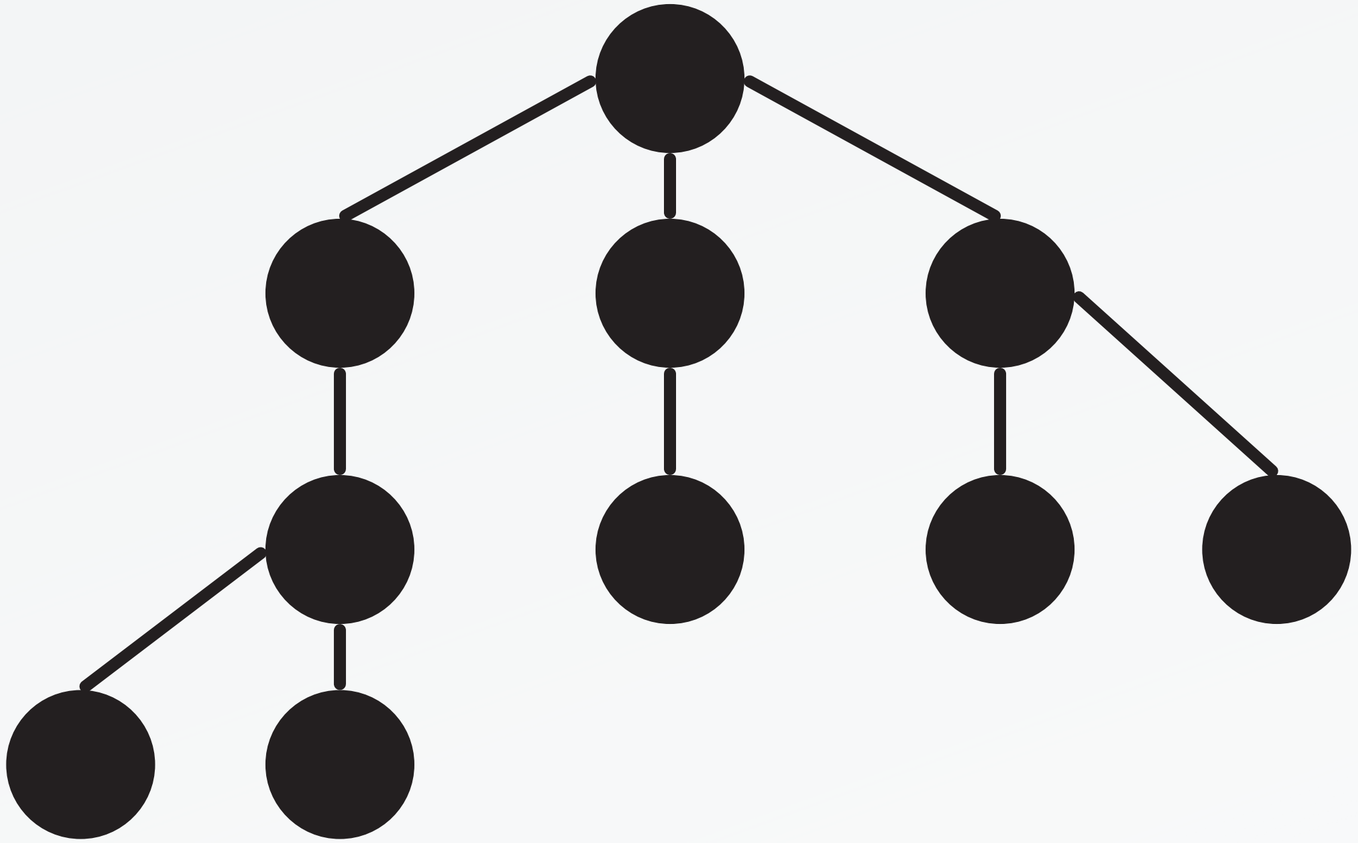
Iterator

Problem



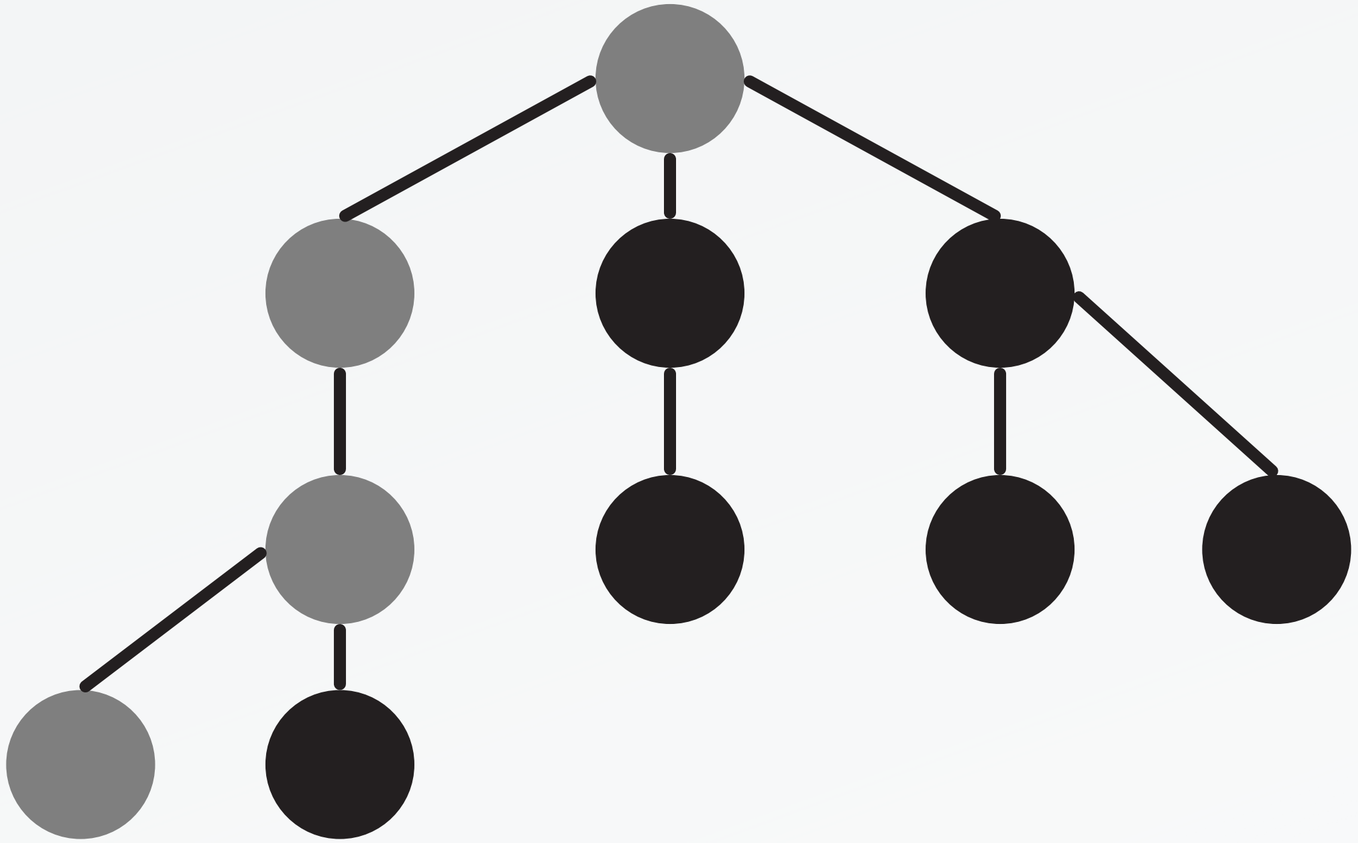
Iterator

Problem



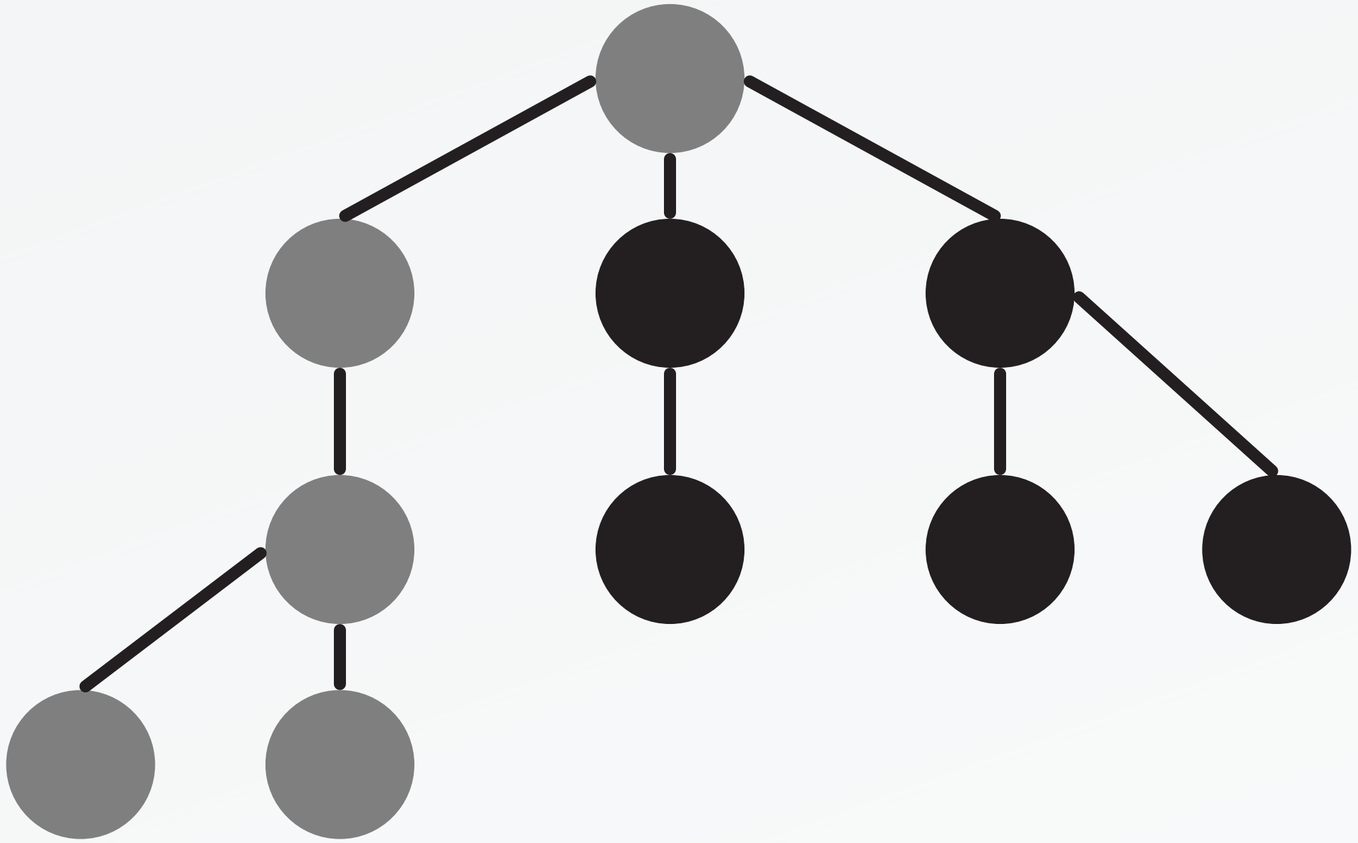
Iterator

Problem



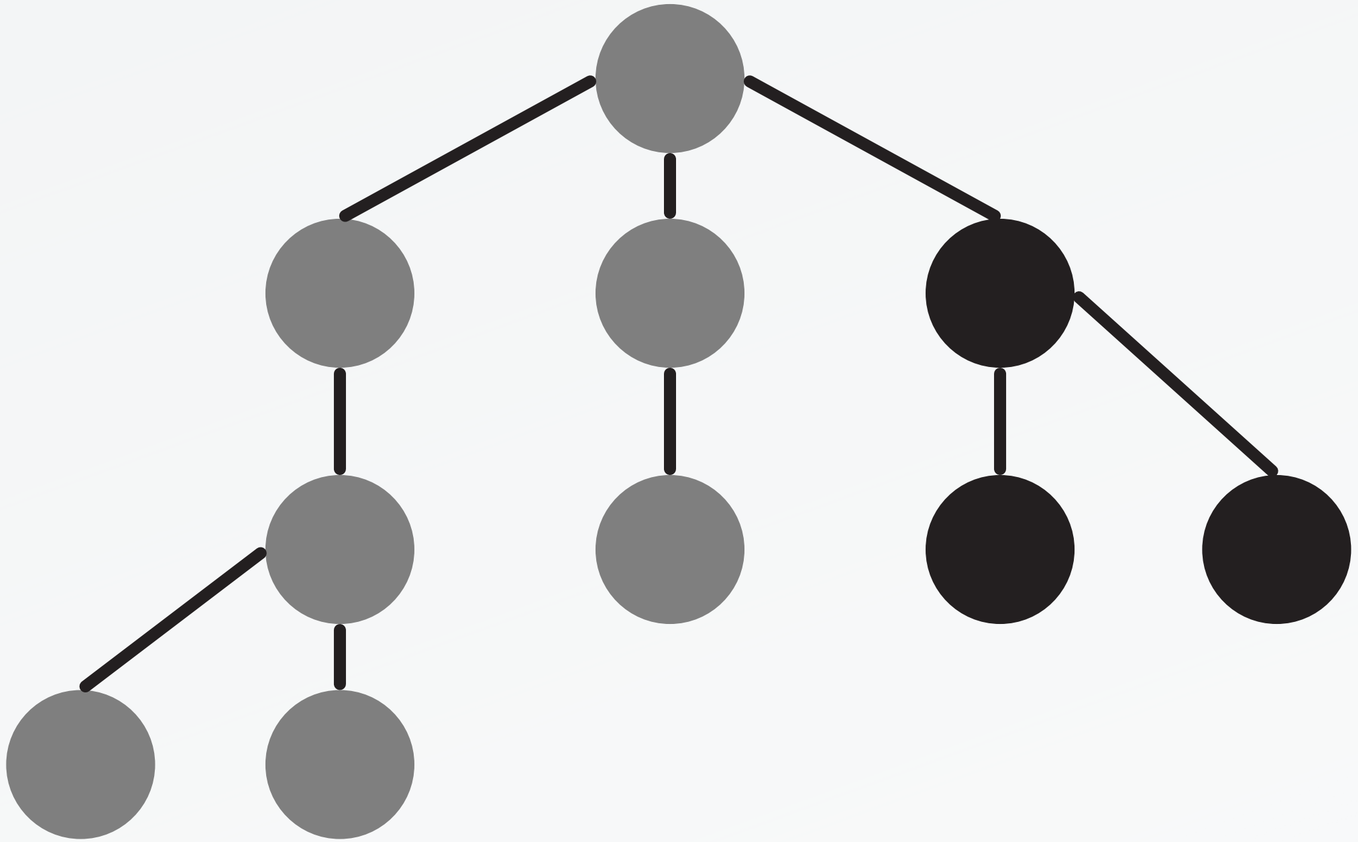
Iterator

Problem



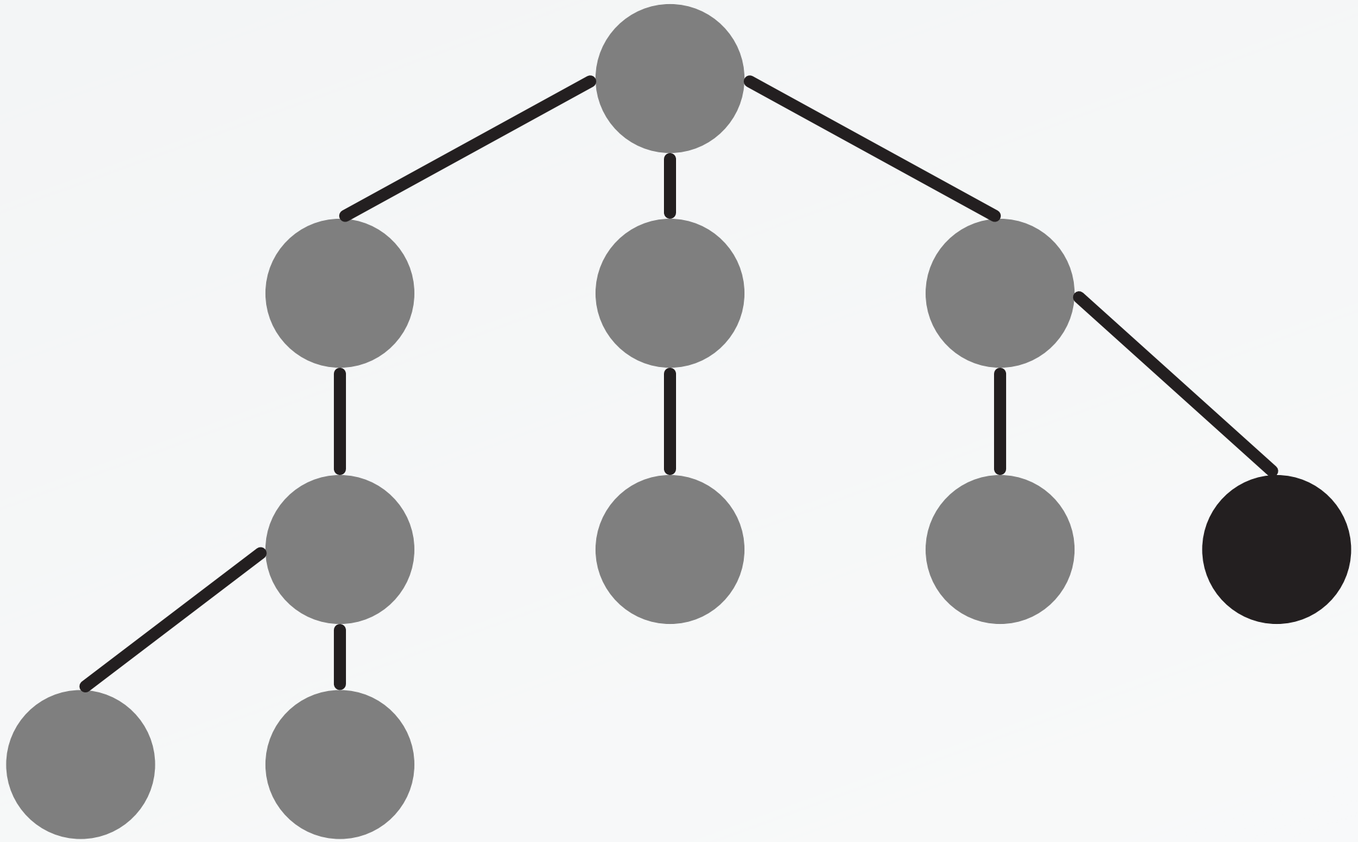
Iterator

Problem



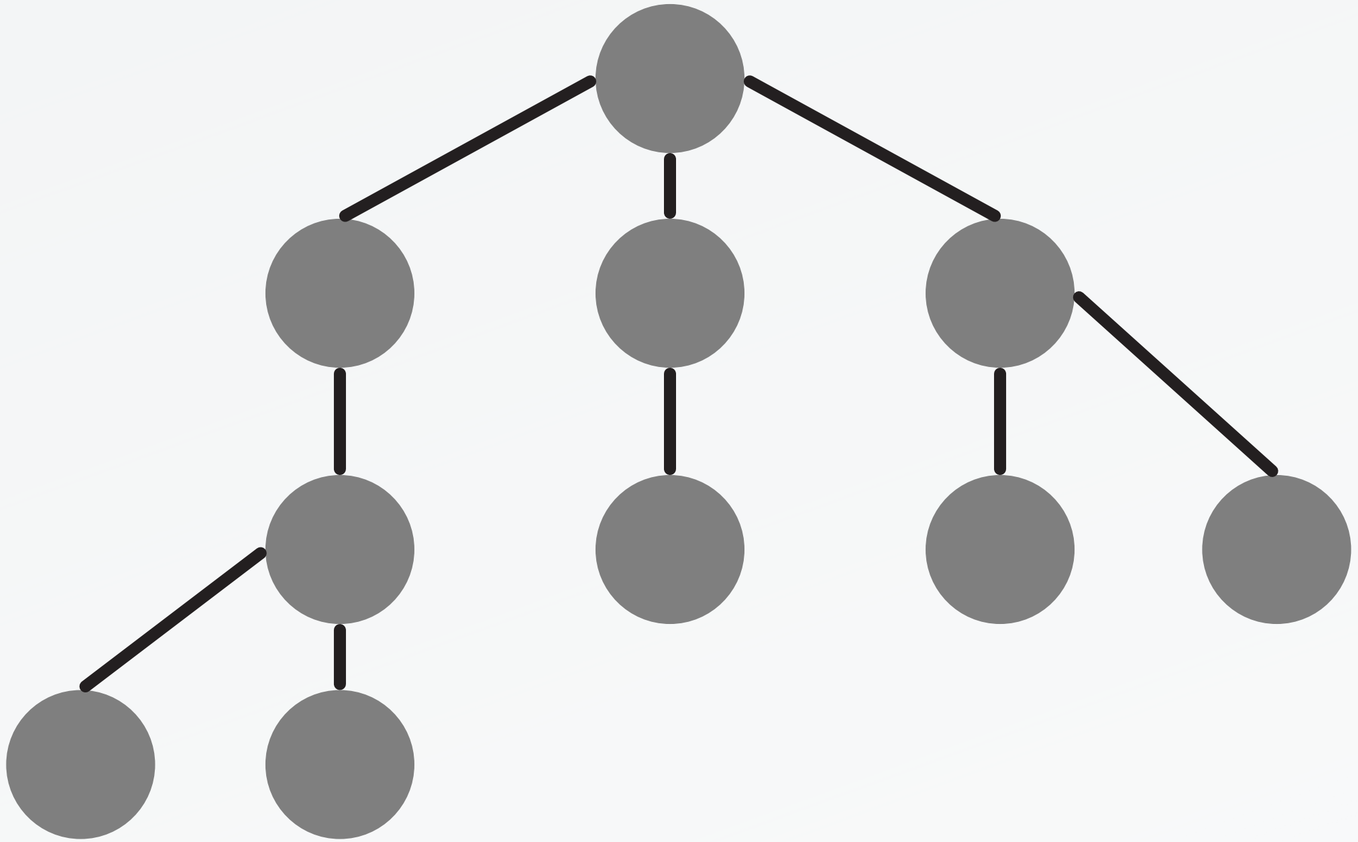
Iterator

Problem



Iterator

Problem



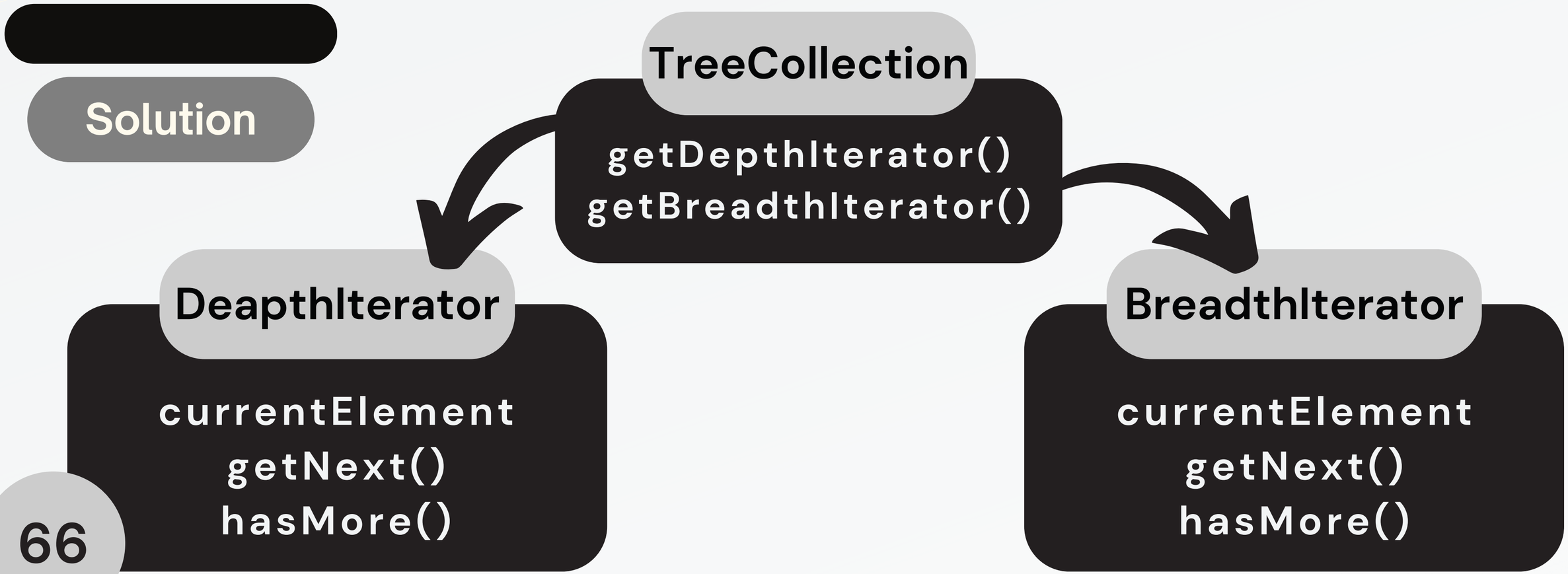
Iterator

Solution

Extract the traversal behavior of a collection into a separate object called an *iterator*.

Iterator

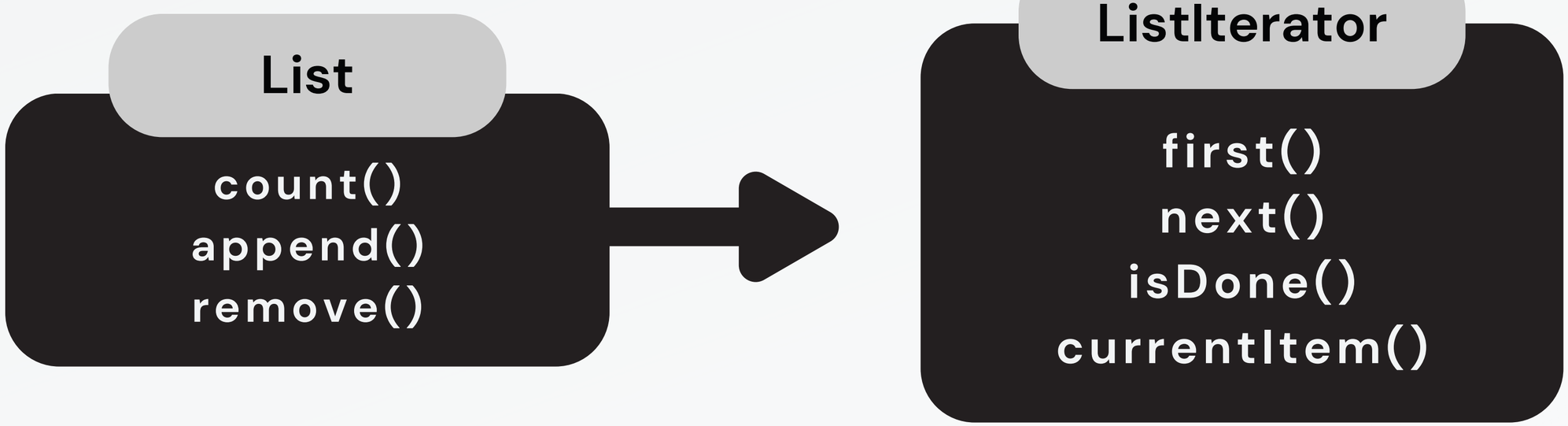
Solution



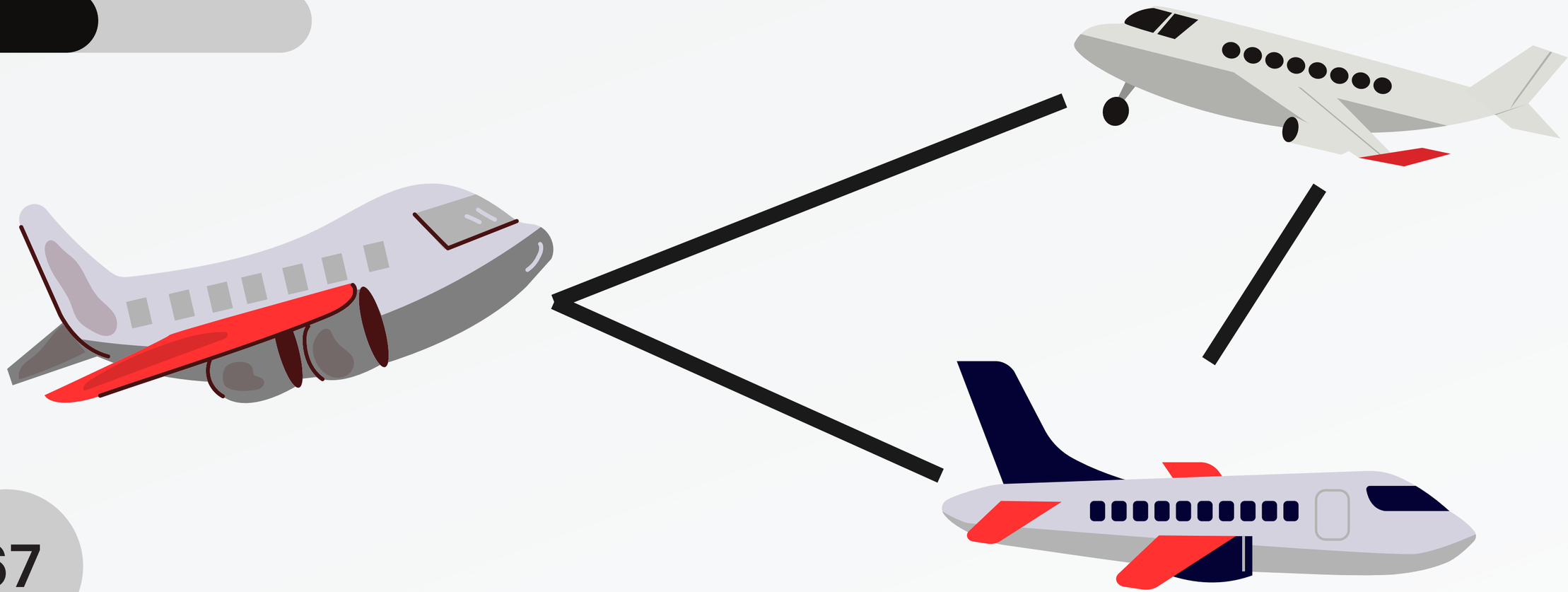
Iterator

LEARN MORE

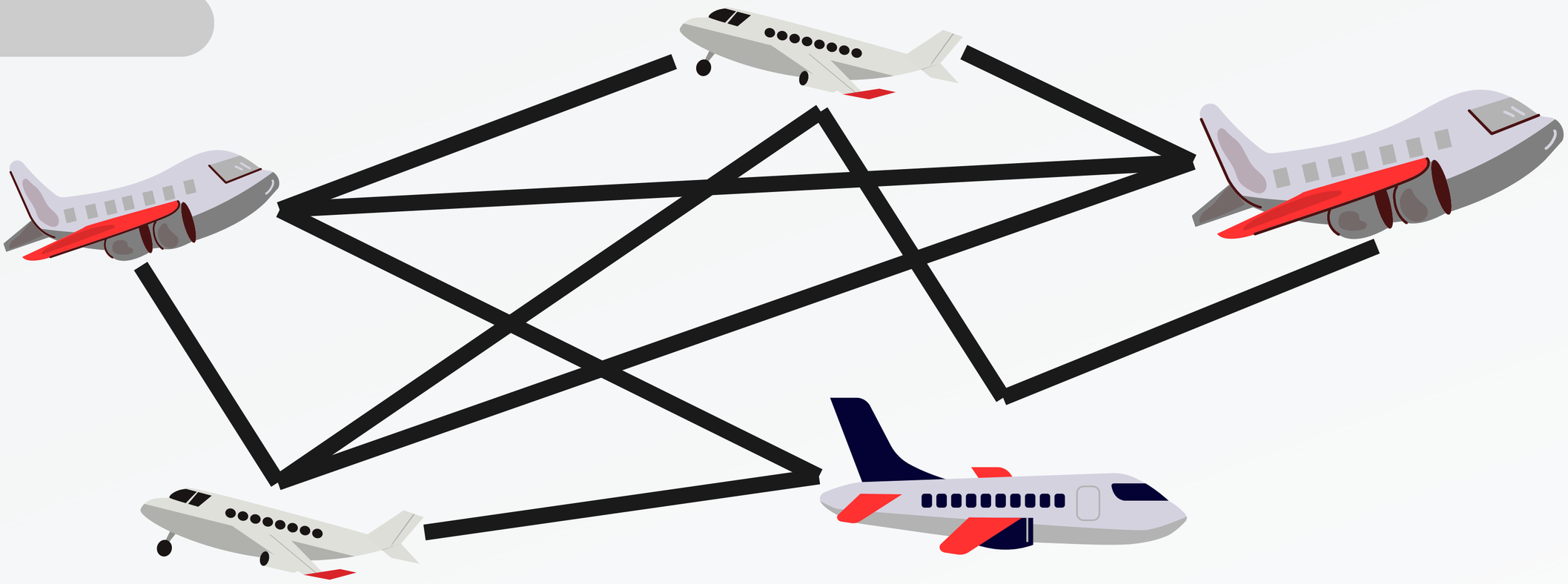
Solution



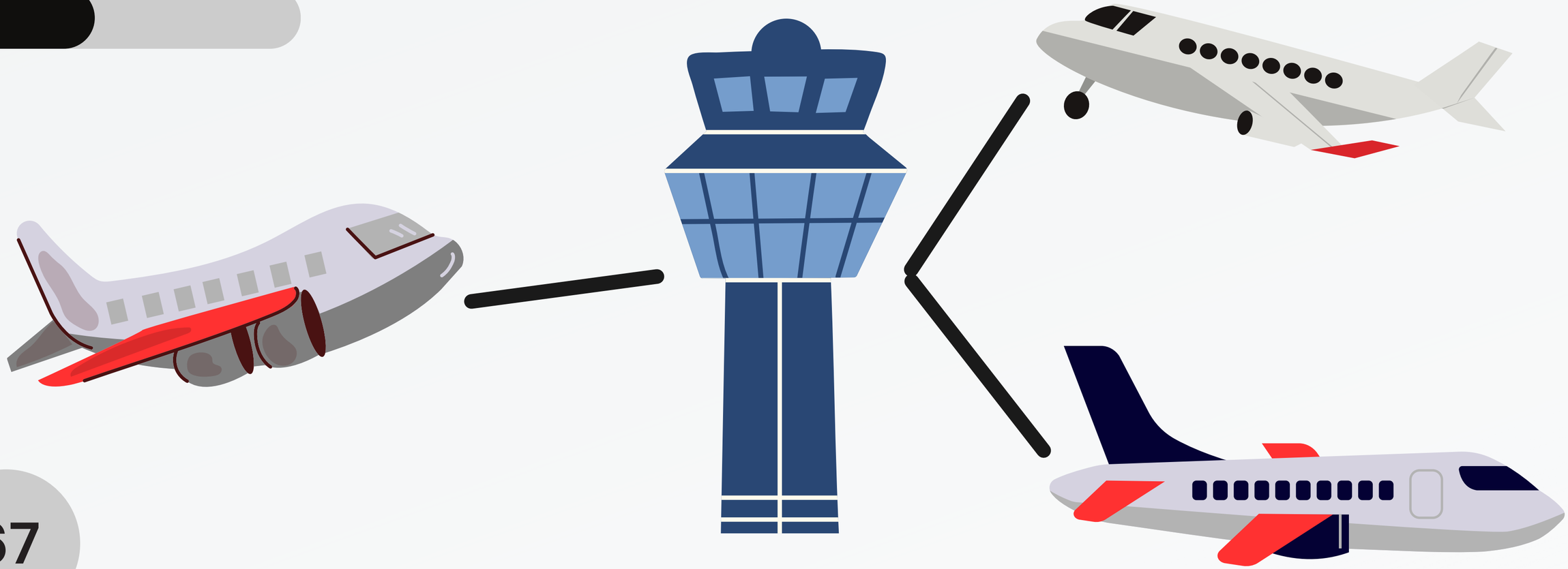
Mediator



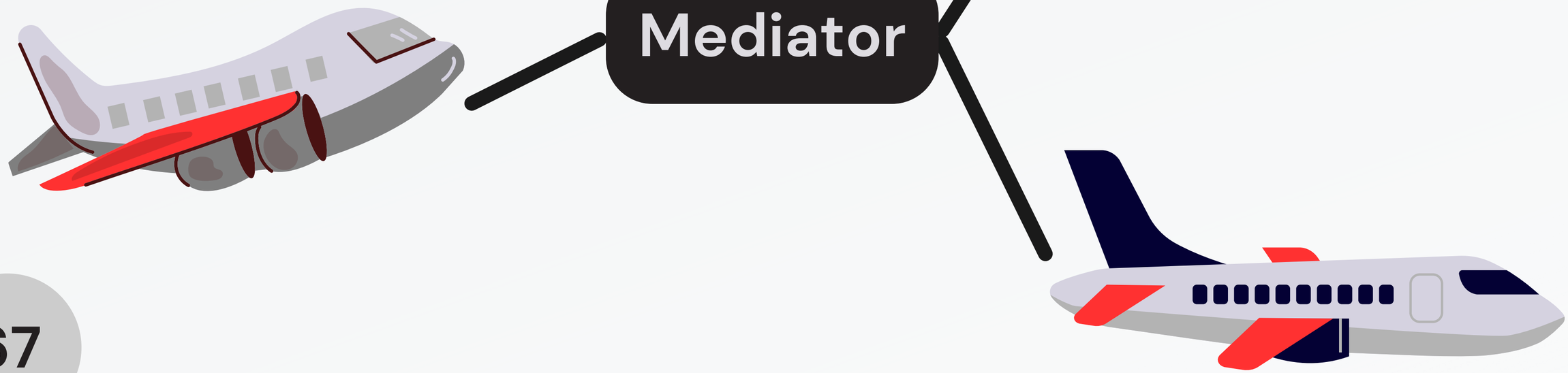
Mediator



Mediator

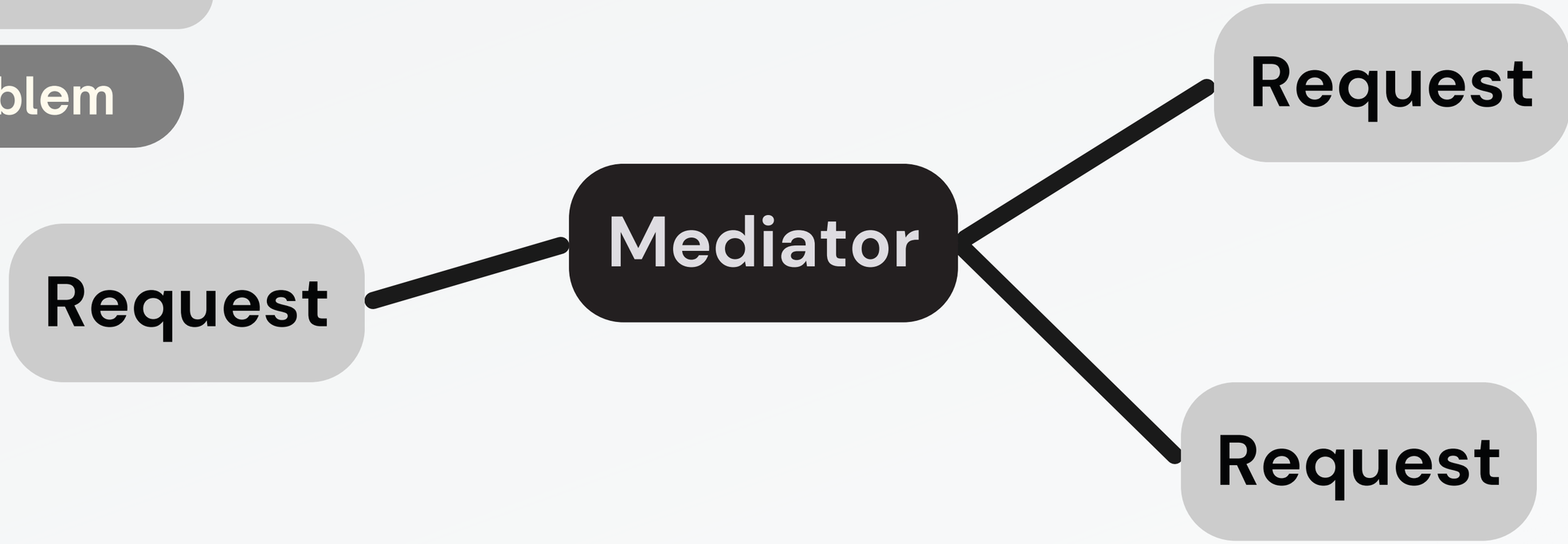


Mediator



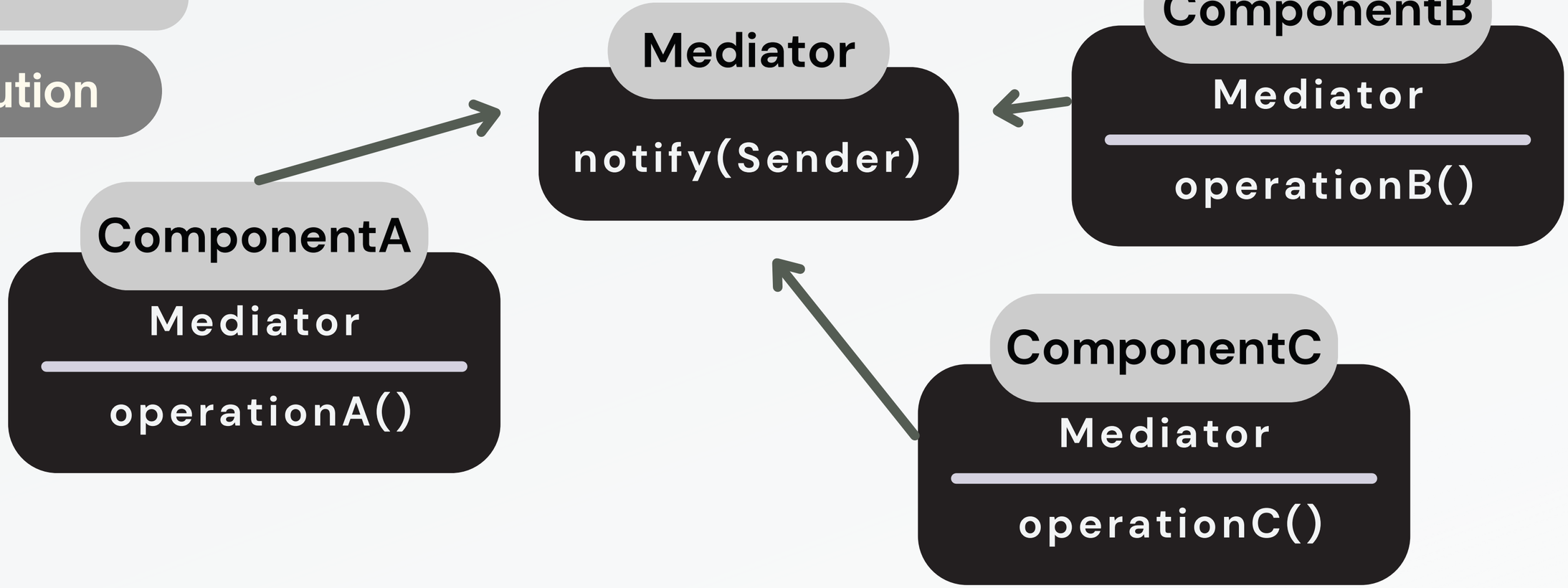
Mediator

Problem



Mediator

Solution



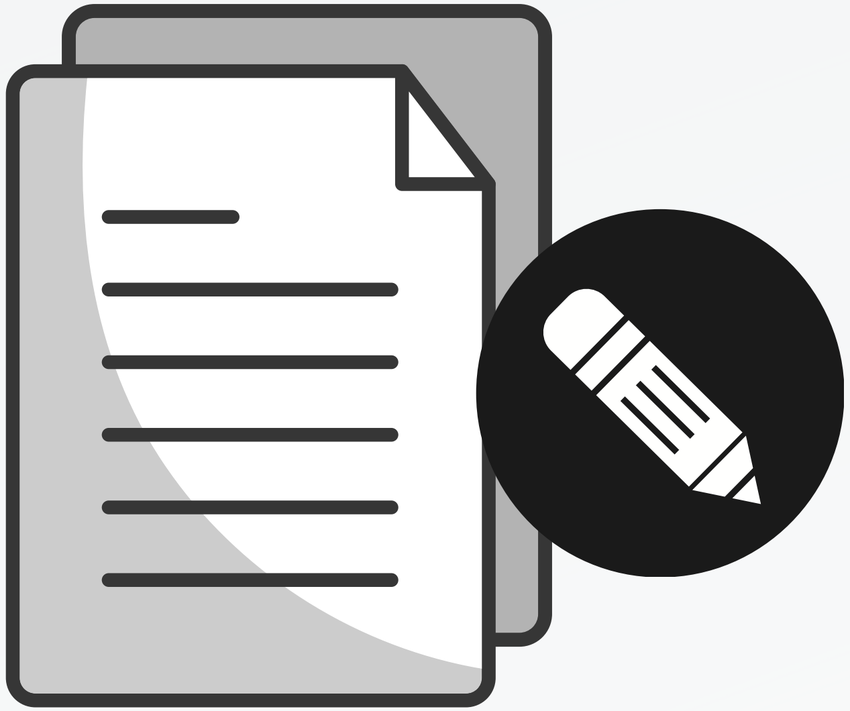
Mediator

- Chat Room (Mediator)
- Users send messages (Requests)

```
// Mediator
class ChatRoom {
  logMessage(user: User, message: string): void {
    const SENDER: string = user.getName();
    let fullMessage: string = `${new Date().toLocaleString()}`;
    fullMessage += ` [${SENDER}]: ${message}`;
    console.log(fullMessage);
  }
}
```


mediator.ts

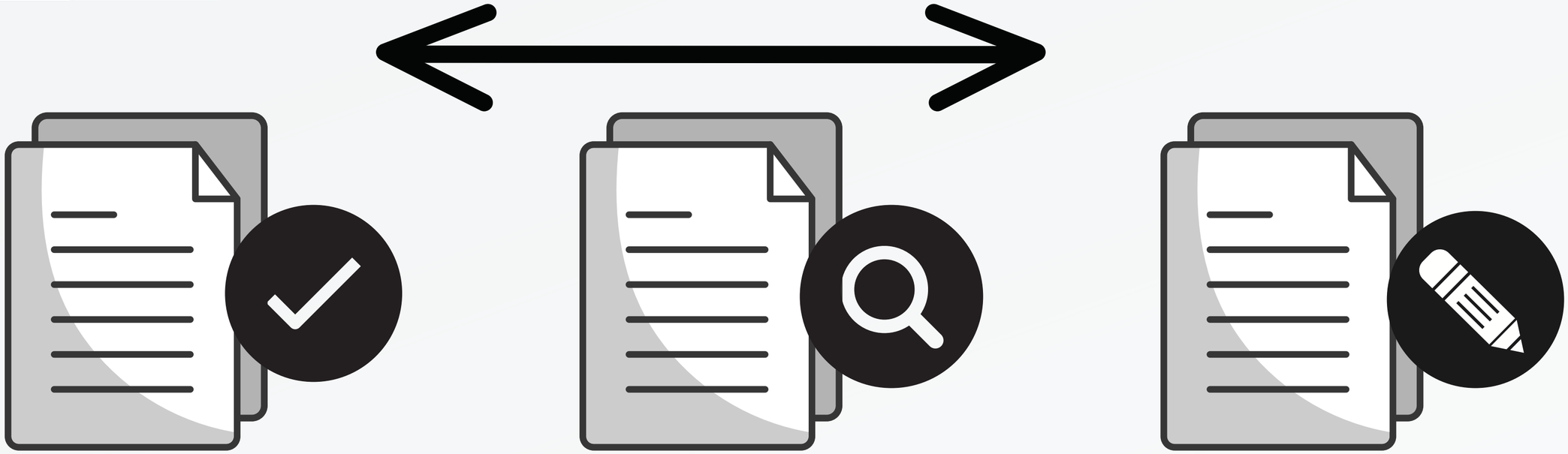
State



State

State

State



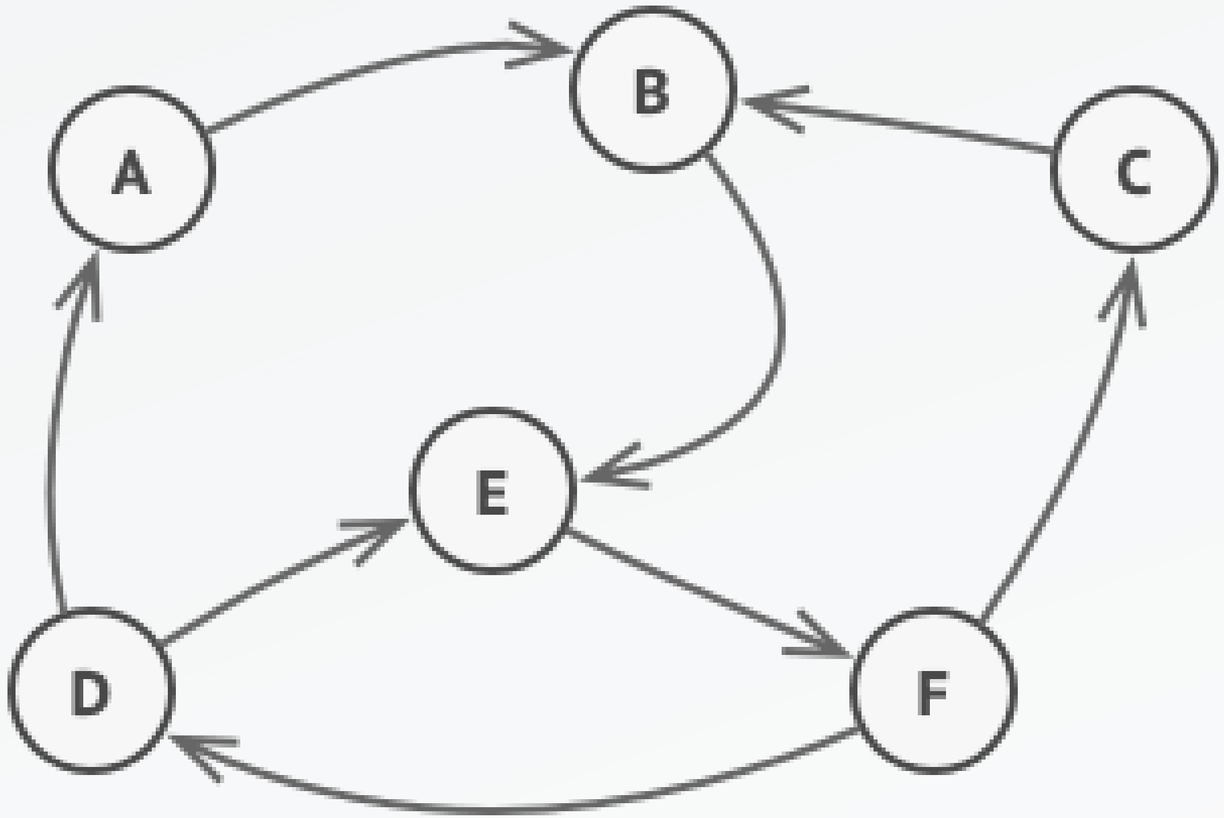
State

Problem

- Finite number of states which a program can be in.
- The program behaves differently.
- It can switch to certain other states.

State

Problem



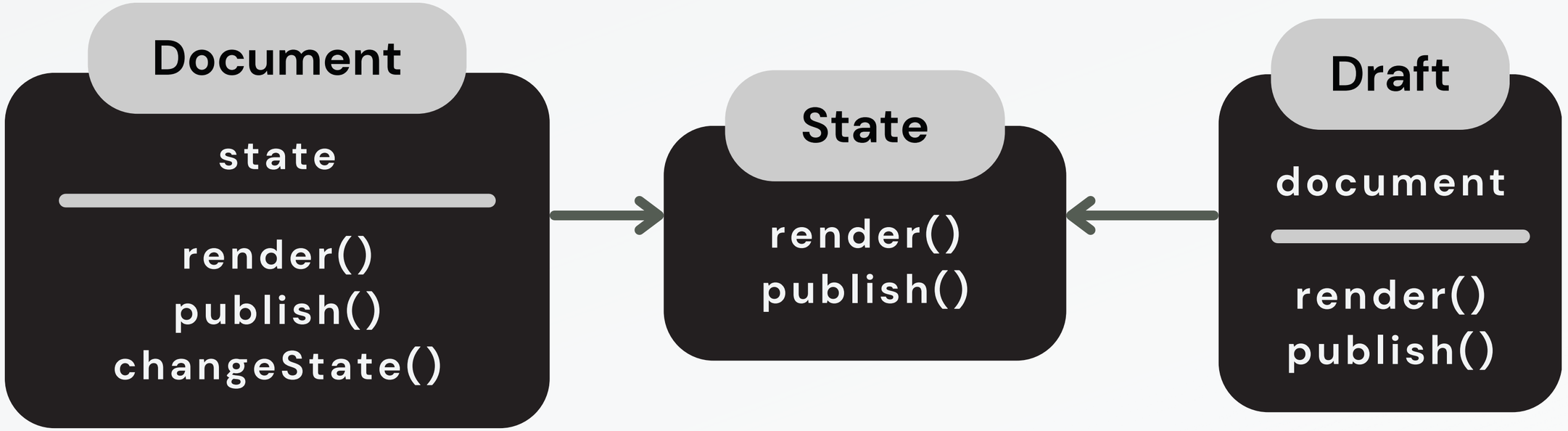
State

Solution

- Create new clases for all possible states.
- Extract all the state specific behaviors into these classes.
- Original object has a context.

State

Solution



State

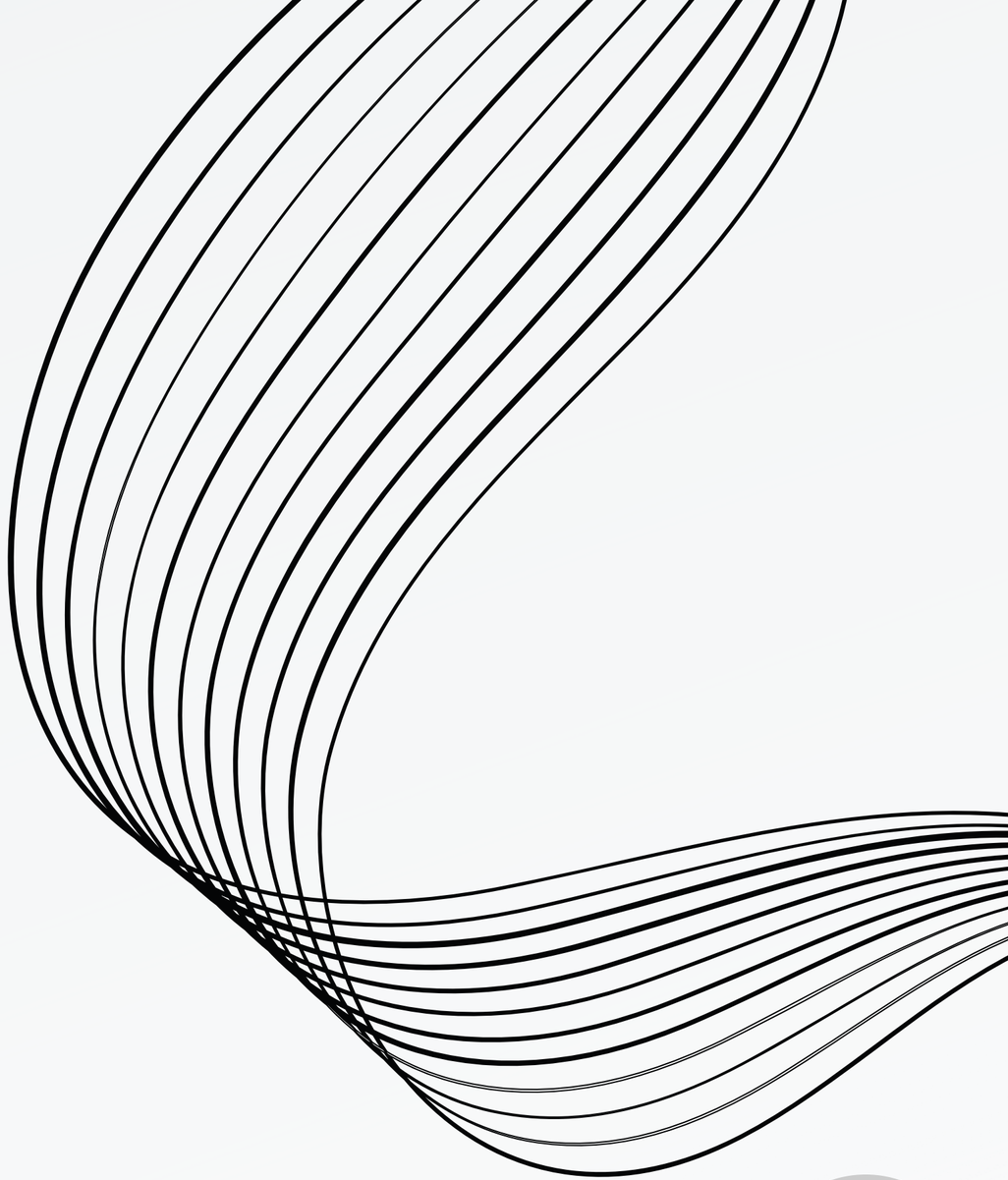
- Human can feel different emotions.
- This emotion can change.

```ts
interface State {
  think(): string;
}
```


state.ts

To sum up...

“Patterns provide a shared language that can maximize the value of your communication with other developers”



Bibliography

Design patterns - Wikipedia

Wikipedia contributors. (2024, 22 febrero). Software design pattern. Wikipedia. https://en.wikipedia.org/wiki/Software_design_pattern

Some examples and explanations of Design Patterns

Index of /254/Patterns. (s. f.). http://www.buyya.com/254/Patterns/

Examples - Javascript

Cocca, G. (2022, 23 junio). JavaScript Design Patterns – Explained with Examples. freeCodeCamp.org. https://www.freecodecamp.org/news/javascript-design-patterns-explained/

Examples - Javascript

Mišura, M. (2018, 29 marzo). The Comprehensive Guide to JavaScript Design Patterns. Toptal Engineering Blog. https://www.toptal.com/javascript/comprehensive-guide-javascript-design-patterns

Examples - Typescript

Refactoring.Guru. (2024, 1 enero). State. https://refactoring.guru/design-patterns/state

Examples - Typescript

10 Design Patterns in TypeScript. (2022, 13 marzo). https://fireship.io/lessons/typescript-design-patterns/

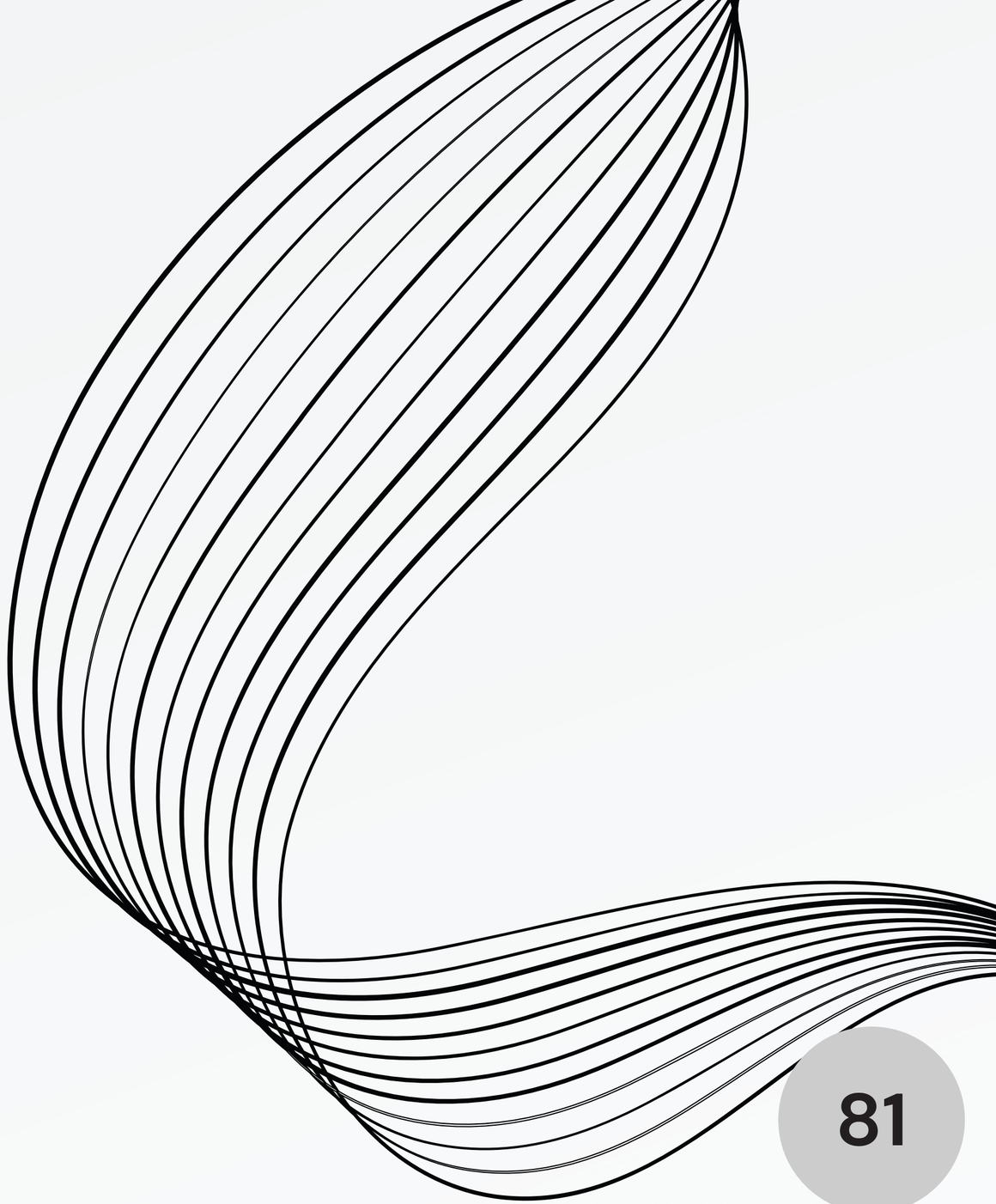
Go4 - Design patterns

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (s. f.). Design Patterns: Elements of Reusable Object-Oriented Software. O'Reilly Online Learning. https://learning.oreilly.com/library/view/design-patterns-elements/0201633612/

Conclusion

Freeman, E., Robson, E., Bates, B., & Sierra, K. (s. f.). Head first design patterns. O'Reilly Online Learning. https://learning.oreilly.com/library/view/head-first-design/0596007124/

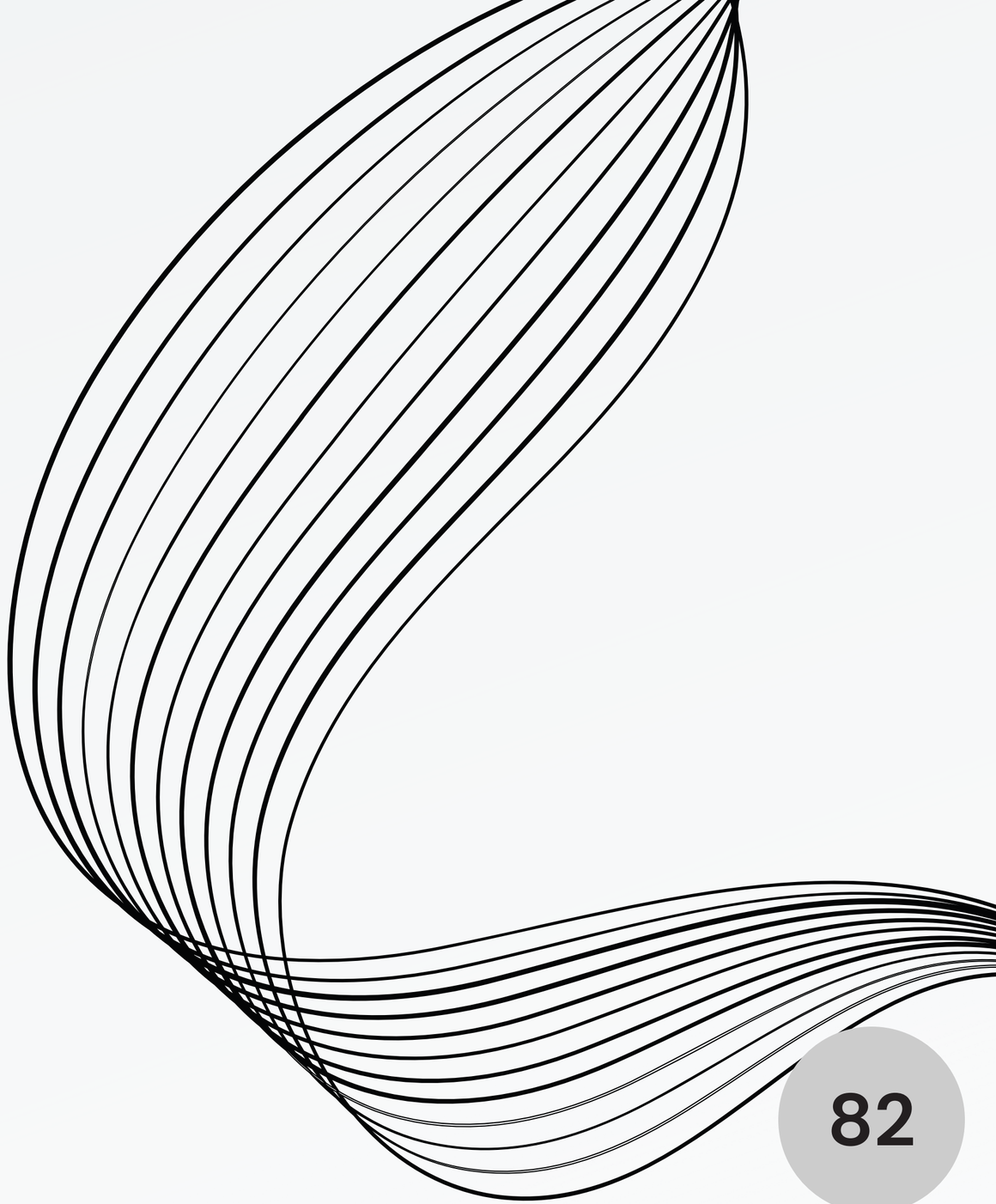
THANK'S FOR WATCHING!



esther.quintero.33@ull.edu.es

hugo.hernandez.14@ull.edu.es

Any questions?



Universidad de La Laguna