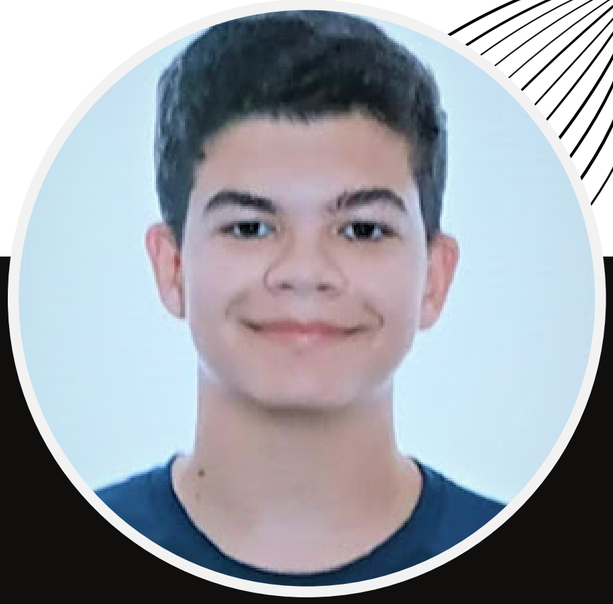
11/03/2024

DESIGN PATTERNS

Esther
Medina
Quintero

esther.quintero.33@ull.edu.es



Hugo
Hernández
Martín

hugo.hernandez.14@ull.edu.es

CONTENT

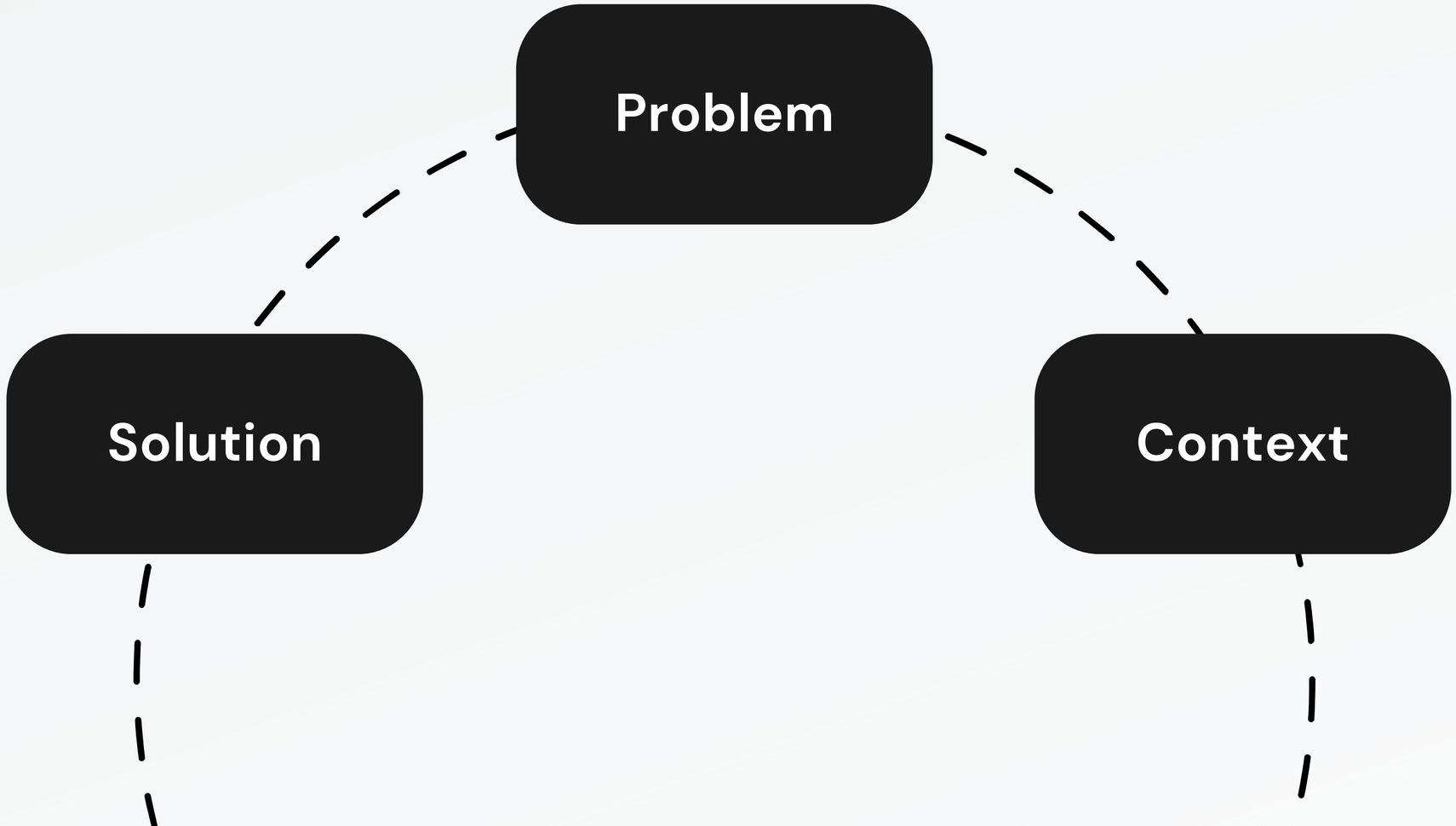
What are Design Patterns?

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?

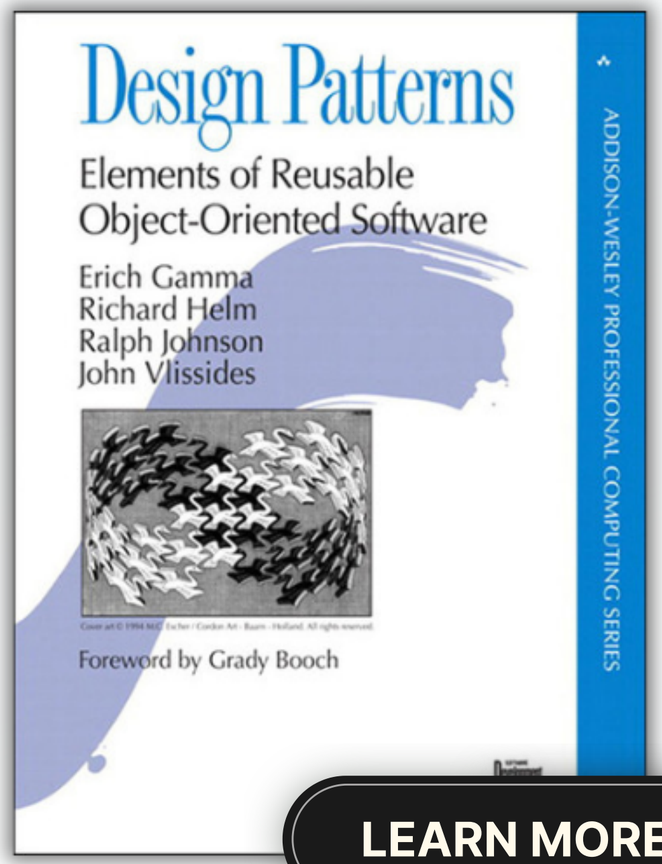
“Reusable solution to a commonly occurring problem within a given context in software design.”

A little bit of History

1977 — 1987 — 1994

Kent Beck

Ward Cunningham



LEARN MORE

Types of Design Patterns

01 Creational

02 Structural

03 Behavioral

LEARN MORE

Creational Patterns

Creational Patterns

- Provides various object creation mechanisms.

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

Builder

Builder

Builder

Extract the object construction code out of its own class and move it to separate objects called *builders*.

HouseBuilder

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

```
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

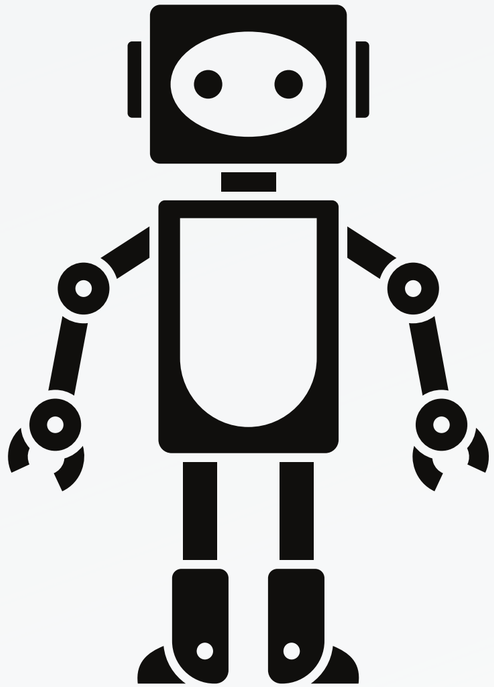
```ts
interface Builder {
  setWalls(walls: number): this;
  setDoors(doors: number): this;
  setRooms(rooms: number): this;
  setPool(pool: boolean): this;
  setGarage(garage: boolean): this;
  setGarden(garden: boolean): this;
  build(): House;
}
```

builder.ts

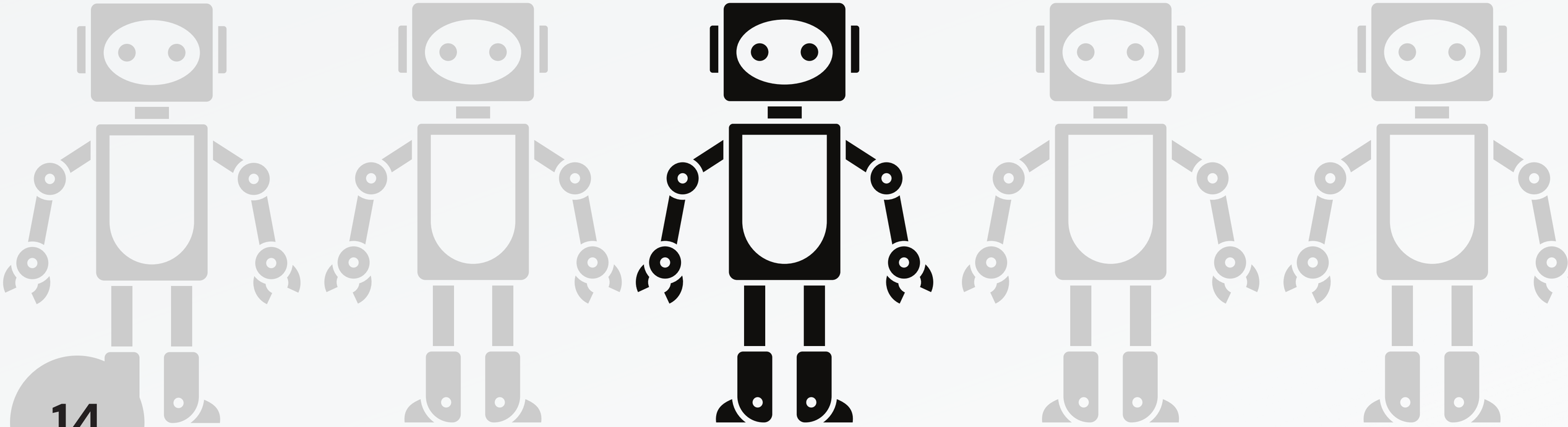
```
class Director {
  public buildHouseWithPool(): House {
    return new HouseBuilder().setWalls(4)
                             .setDoors(2)
                             .setRooms(4)
                             .setPool(true)
                             .setGarage(false)
                             .setGarden(false)
                             .build();
  }
  // public buildHouseWithGarden(): House {
  //   ...
  // }
}
```

builder.ts

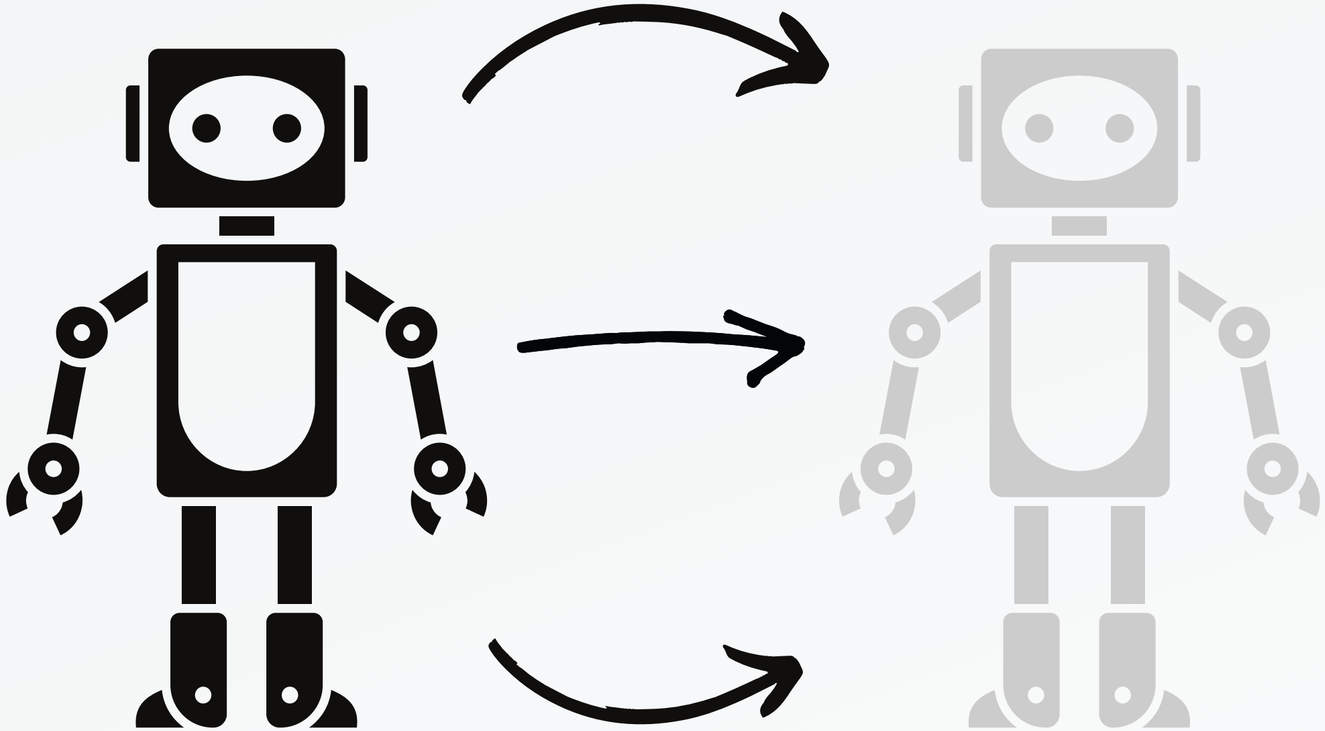
Prototype



Prototype



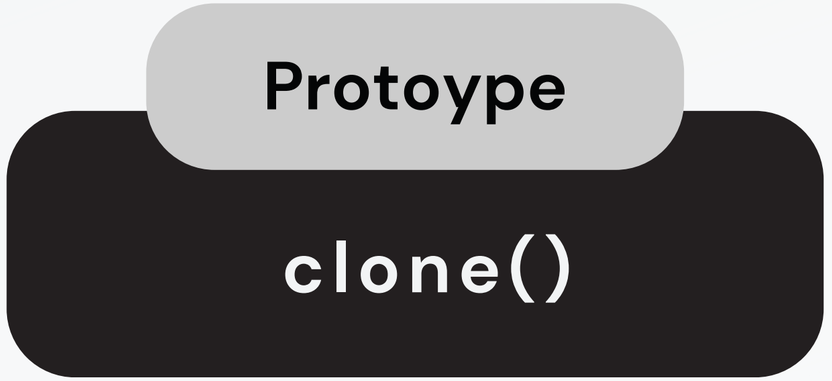
Prototype



Prototype

Delegate the cloning process to the actual objects that are being cloned.

Common interface for all objects that support cloning.



```
// 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

```
class Circle implements Shape {
  private radius: number;
  // constructor goes here
  clone(): Circle { return new Circle(this.radius); }
  draw(): void {
    console.log(`Drawing a circle of radius ${this.radius}`);
  }
  // Setter and getter
  setRadius(radius: number): void { this.radius = radius; }
  getRadius(): number { return this.radius; }
}
```

prototype.ts

```
class Rectangle implements Shape {
  private width: number;
  private height: number;
  // constructor goes here
  clone(): Rectangle { return new Rectangle(this.width, this.height); }
  draw(): void {
    console.log(
    `Drawing a rectangle of width ${this.width} and height ${this.height}`
    );
  }
  // Setters and Getters
}
```


prototype.ts

```
function mainPrototype(): void {
  let circle: Circle = new Circle(10);
  let rectangle: Rectangle = new Rectangle(10, 20);
  let clonedCircle: Circle = circle.clone();
  let clonedRectangle: Rectangle = rectangle.clone();
}
```


prototype.ts

Singleton

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

Singleton

Singleton

Singleton

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

```
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

Structural Patterns

Structural Patterns

- They deal with the relationships between objects.

Structural Patterns

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

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

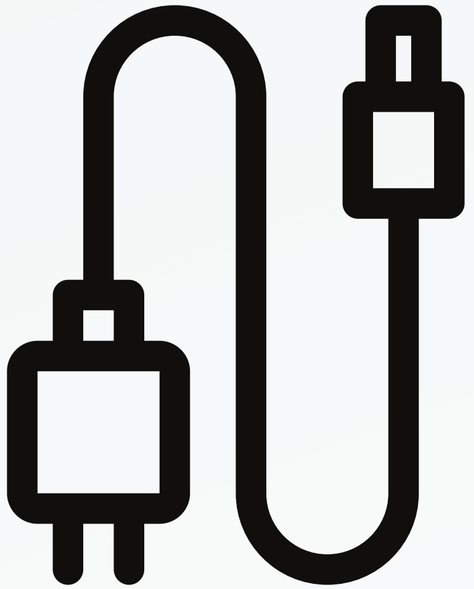
01 Adapter

03 Composite

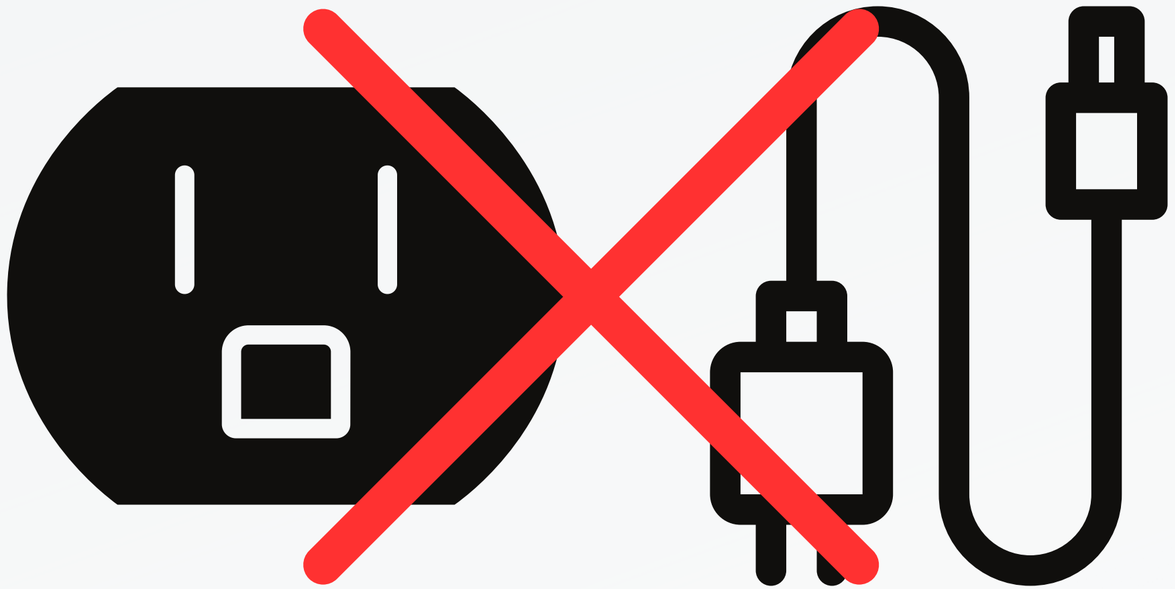
05 Facade

07 Proxy

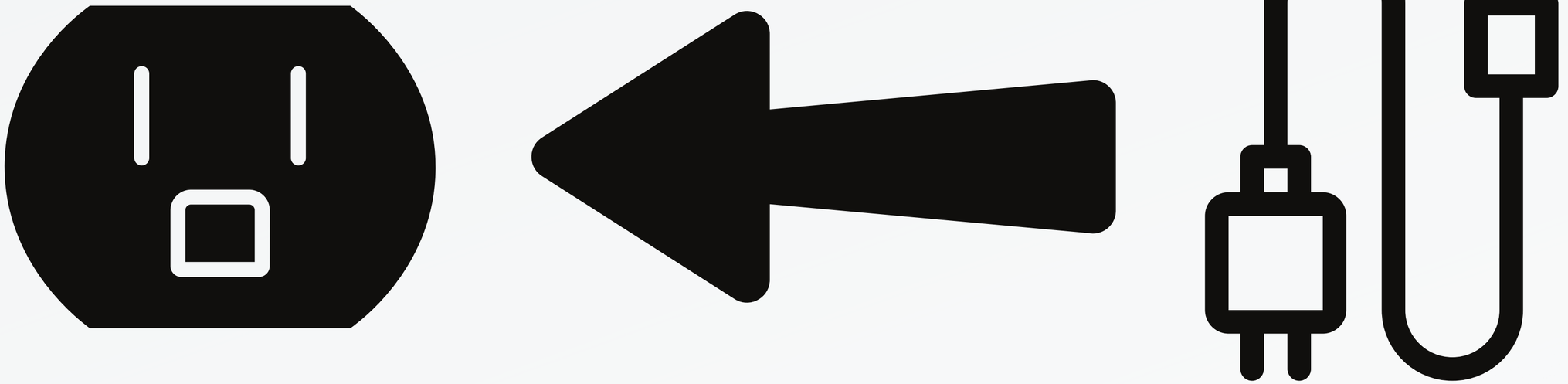
Adapter



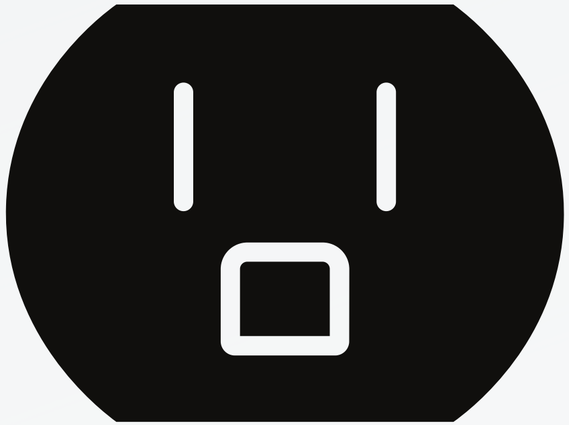
Adapter



Adapter

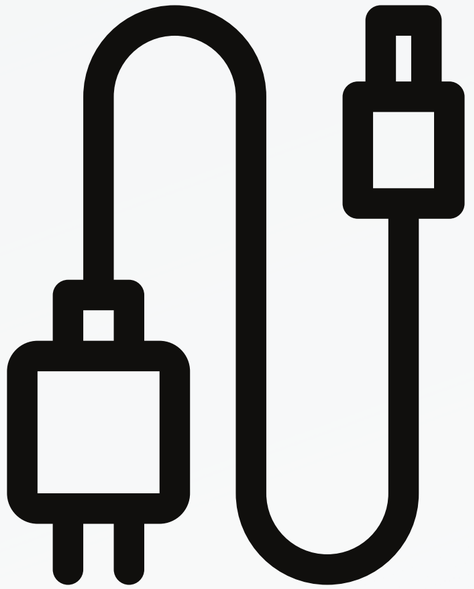


Adapter



Adapter

(...)

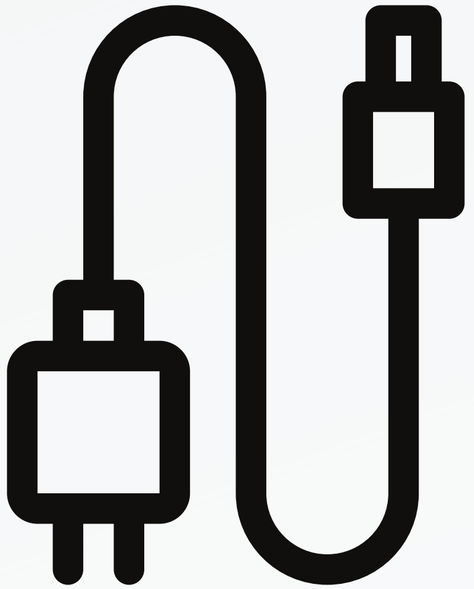


Adapter

Service

Adapter

(...)



Adapter

Service

Adapter

(...)

Interface

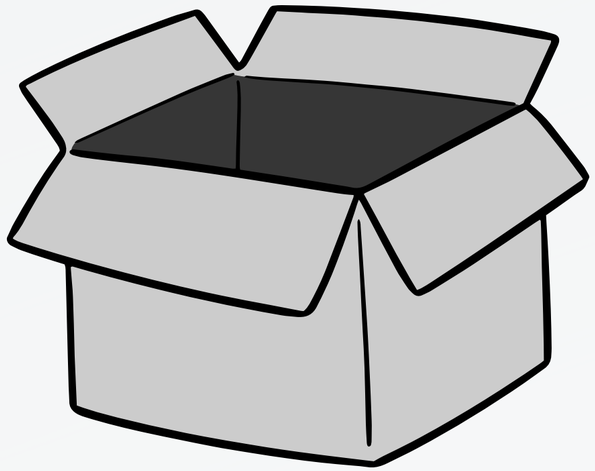
```ts
interface Printer {
  print(): void;
}

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


adapter.ts

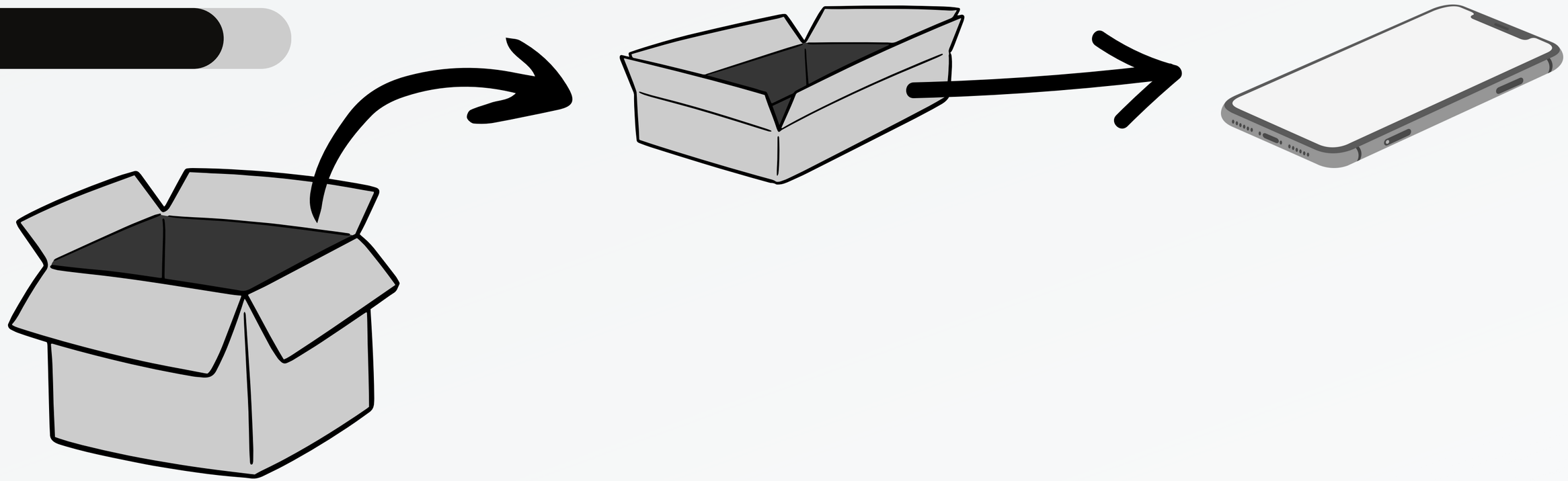
```
class PrinterAdapter implements Printer {
  private legacyPrinter: LegacyPrinter = new LegacyPrinter();
  print(): void {
    this.legacyPrinter.printString();
  }
}

function mainAdapter(): void {
  let printerAdapter: PrinterAdapter = new PrinterAdapter();
  printerAdapter.print();
```


adapter.ts

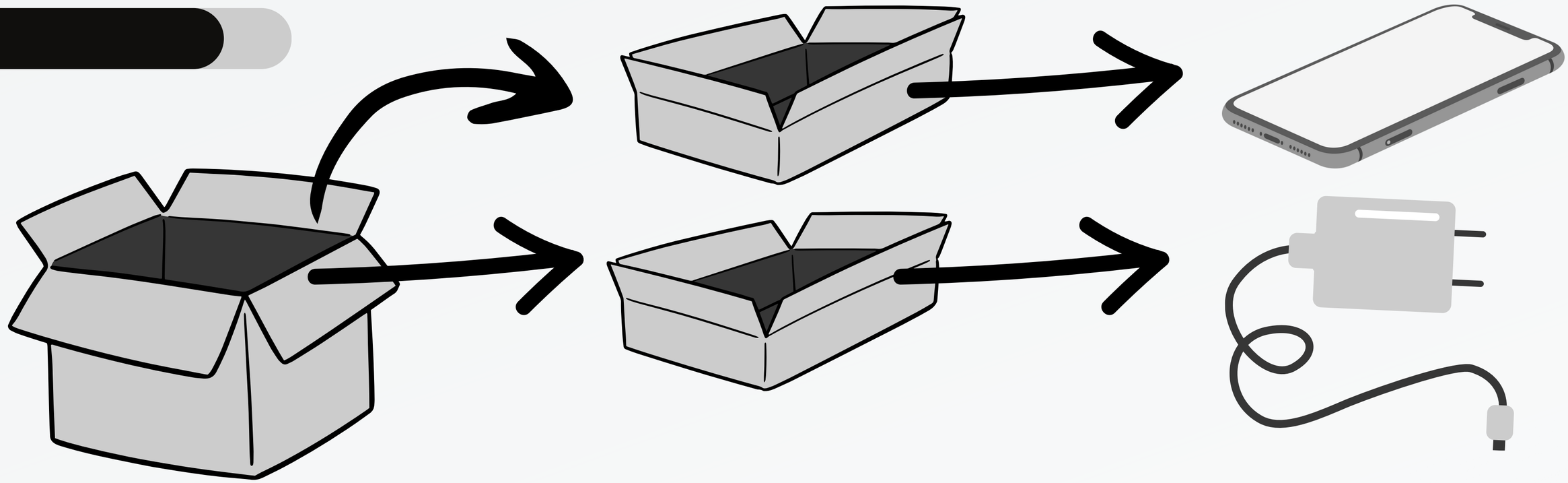
Composite



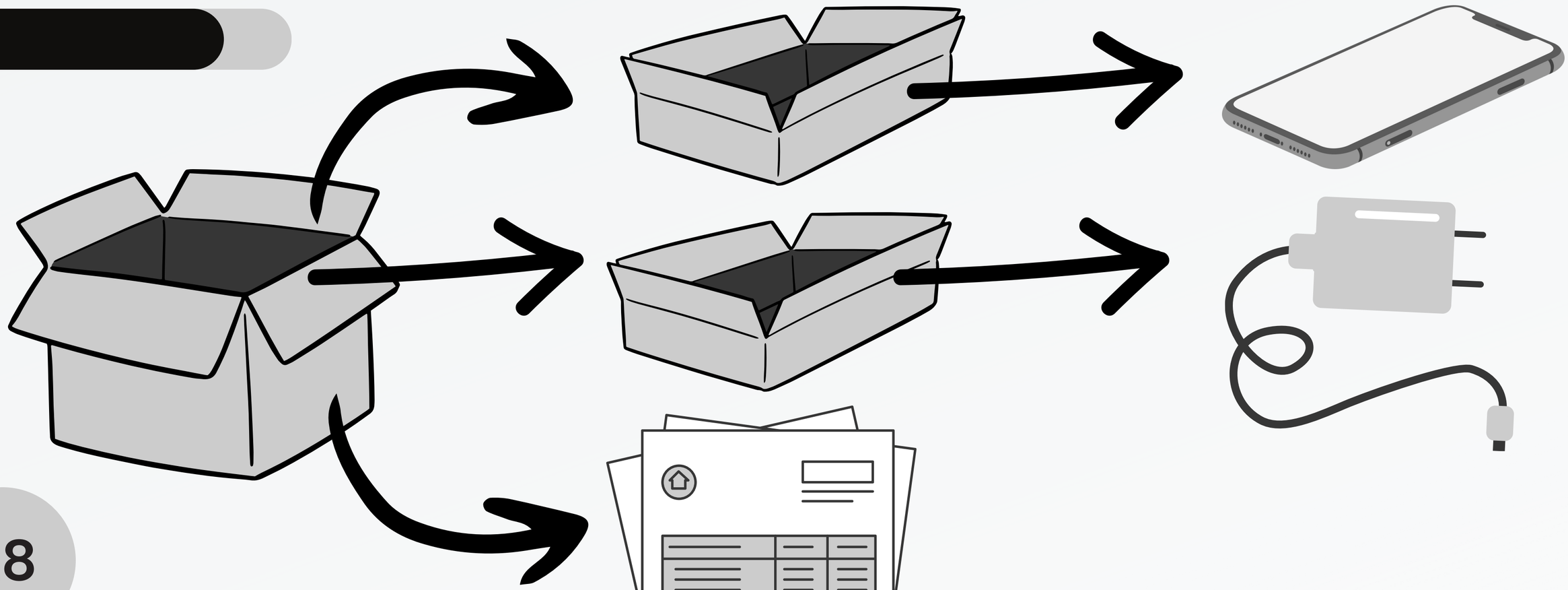
Composite



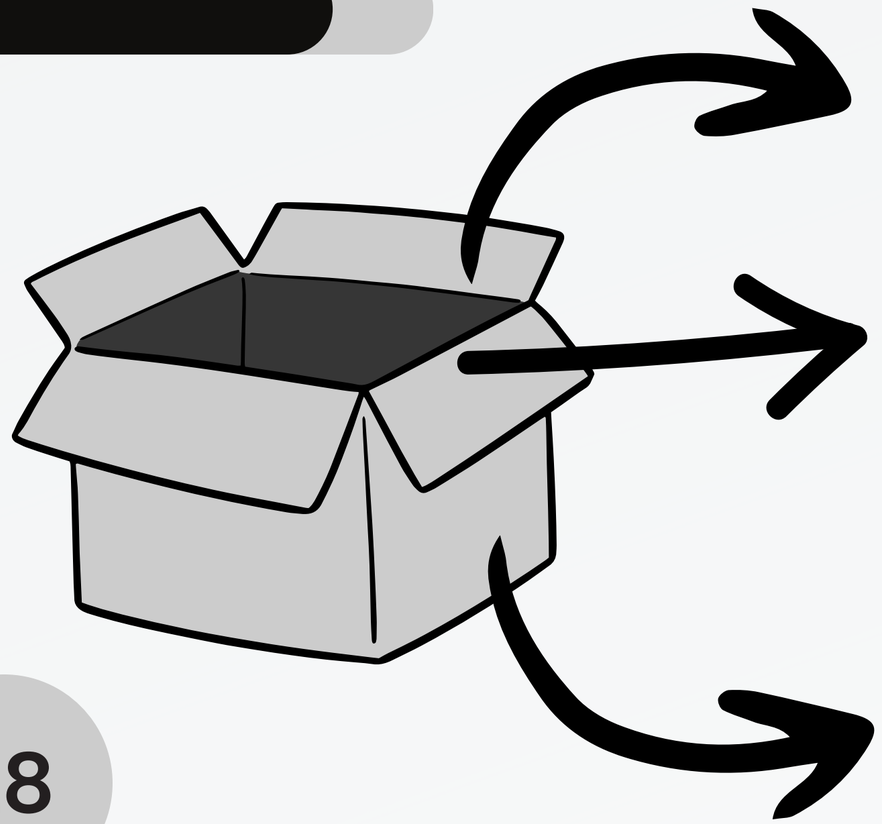
Composite



Composite



Composite

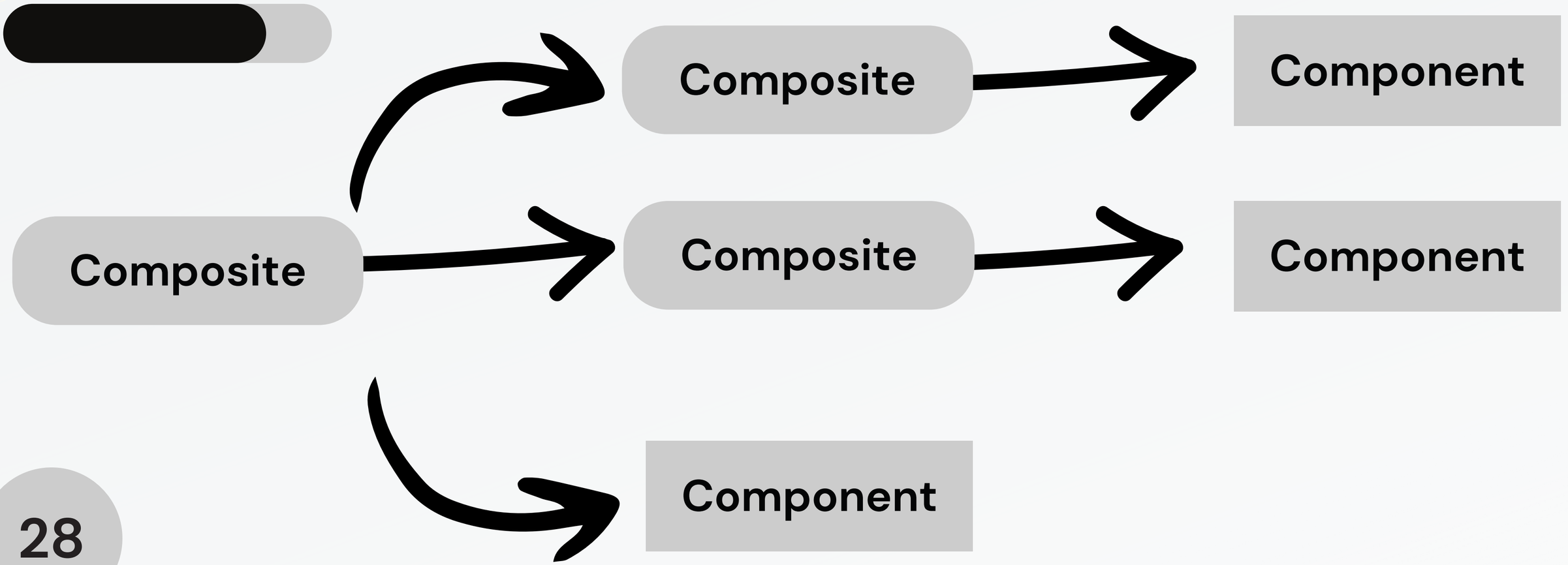


Component

Component

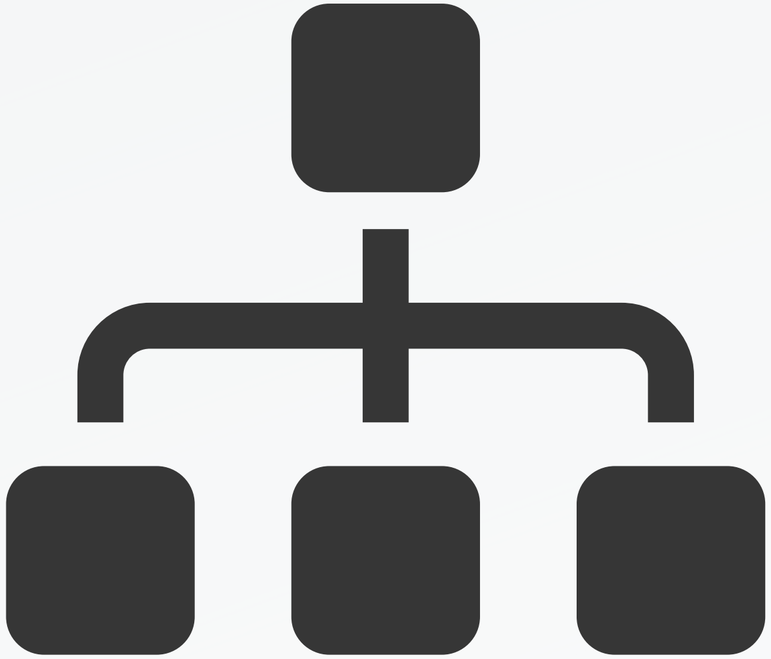
Component

Composite


Composite
Composite
Component
Composite
Composite
Component
Component

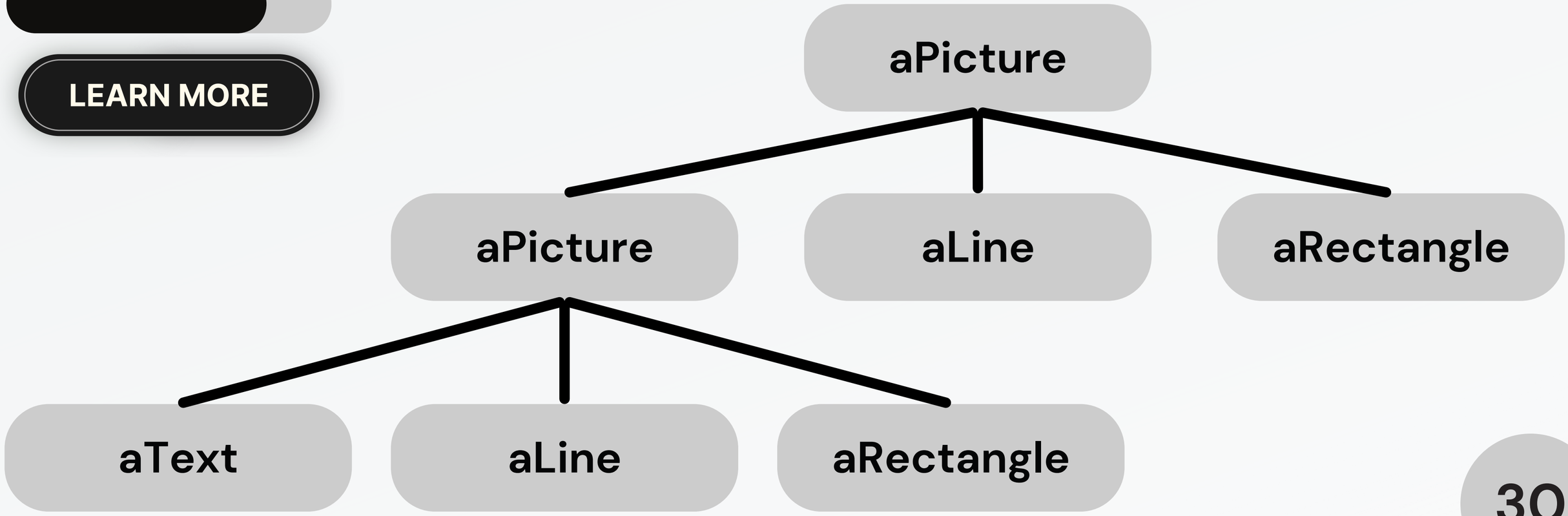
Composite

Work with *Products* and *Boxes* through a common interface.



Composite

LEARN MORE

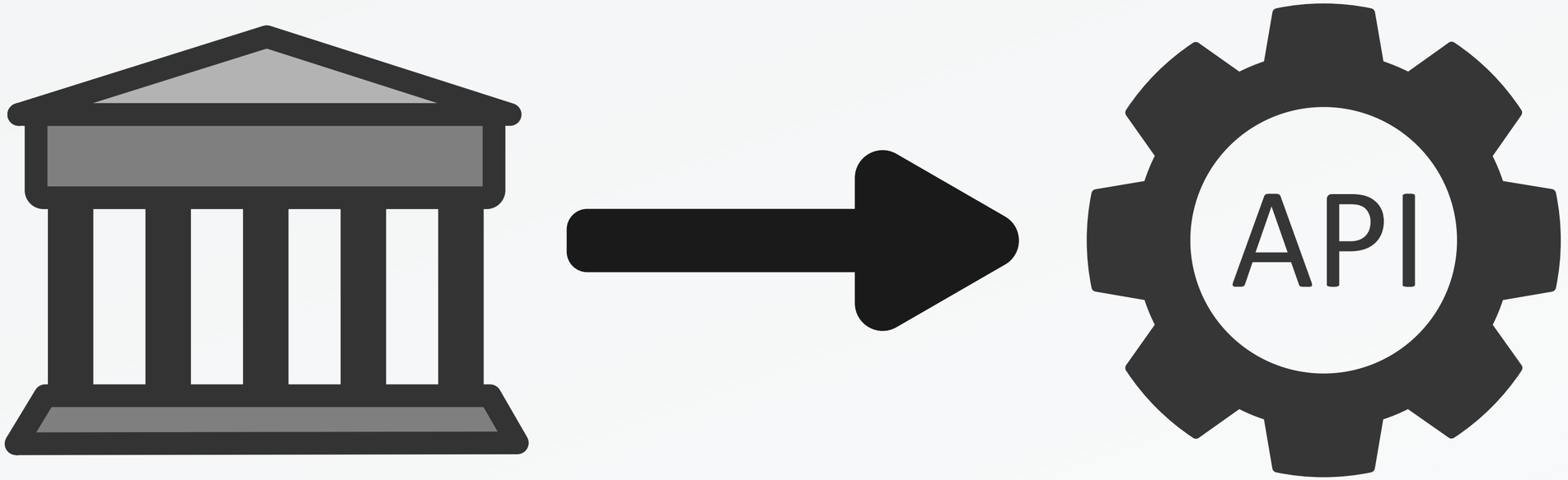


Facade

Facade

Complex Subsystem

Facade



Facade

- It provides a *unified interface* to a set of interfaces in a subsystem.

Facade

- It provides a *unified interface* to a set of interfaces in a subsystem.
  - Facade defines a *higher-level* interface that makes the subsystem easier to use.

```
// 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

```
// Facade Provides only an upload method
class CloudProviderFacade {
  private service: CloudProviderService;
  constructor() { this.service = new CloudProviderService(); }
  public uploadFile(file: string): string {
    if (!this.service.isLoggedIn()) this.service.logIn();
    const convertedFile = this.service.convertFile(file);
    this.service.uploadFile(convertedFile);
    return this.service.getFileLink(convertedFile);
  }
}
```

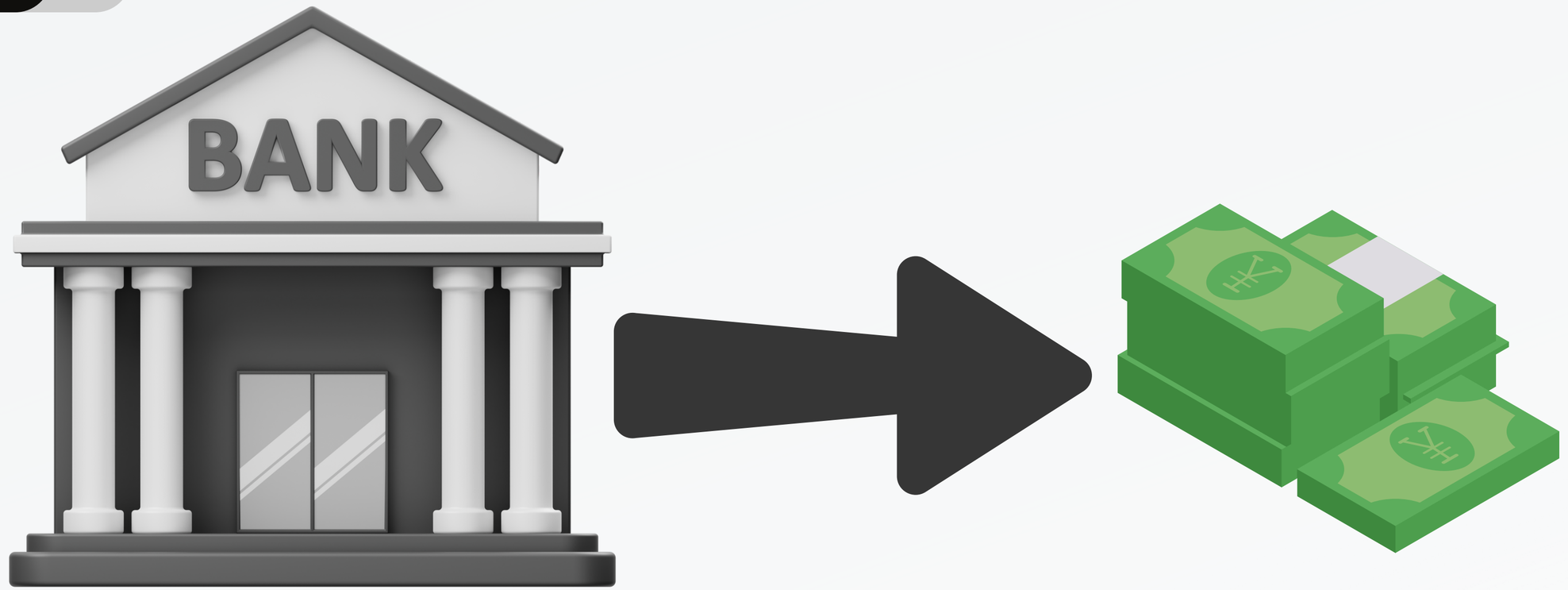
facade.ts

```
// On the client code we can now use the facade to upload
  // a file without having to worry about the complex system.
  const facade = new CloudProviderFacade();
  const fileLink = facade.uploadFile("file.txt");
  console.log("File link:", fileLink);
```


facade.ts

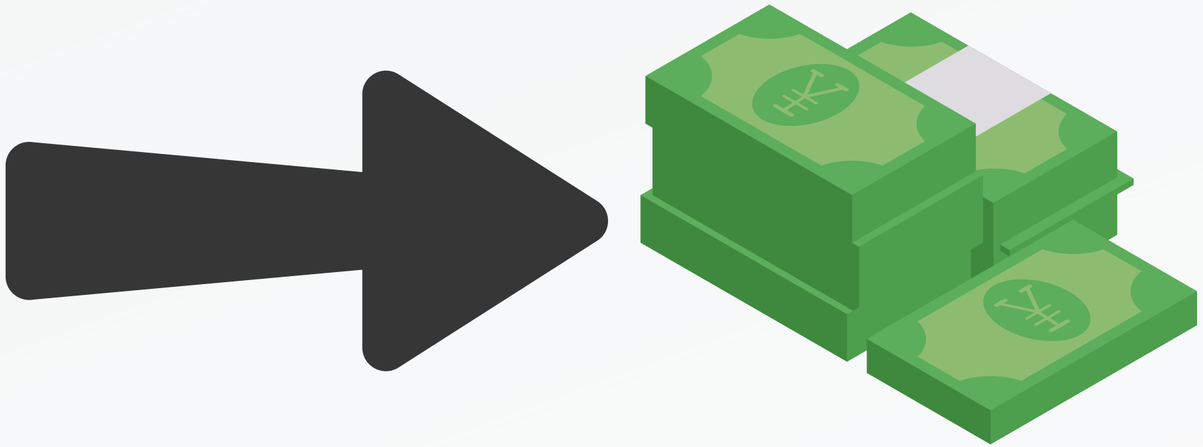
Proxy

Proxy

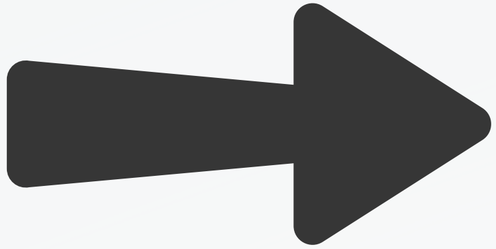


Proxy

Proxy

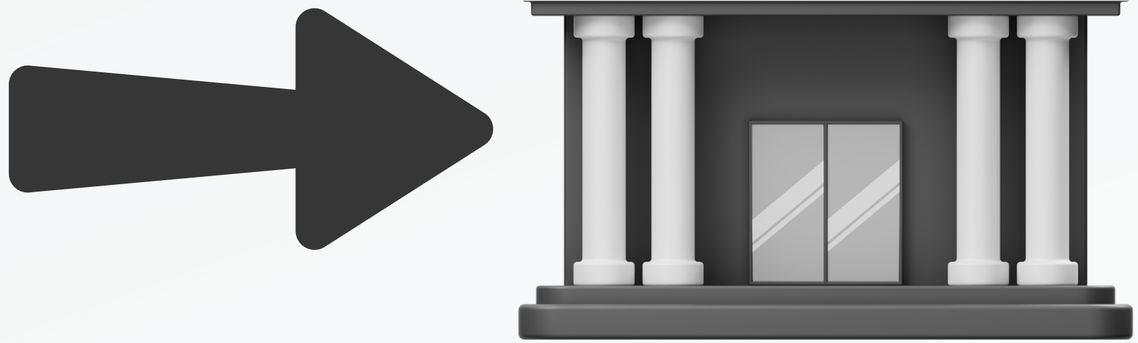


Proxy



Proxy

Proxy



Proxy

- Substitute or placeholder for another object.

Proxy

- Substitute or placeholder for another object.
  - Controls access to the original object.
  - Perform something before the request.

```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

```ts
class Person implements PersonInfo {
  private name: string = 'Unknown';
  private age: number = -1;
  constructor(name?: string, age?: number) {
    if (name) this.name = name;
    if (age) this.age = age;
  }
  // Getters and setters
  getName(): string { return this.name; }
  getAge(): number { return this.age; }
  setName(name: string): void { this.name = name; }
  setAge(age: number): void { this.age = age; }
}
```

proxy.ts

```
class PersonProxy implements PersonInfo {
  constructor(private person: Person) {}
  getName(): string { this.person.getName(); }
  setName(name: string): void {
    if (/[A-Z][a-z]+/.test(name)) {
      this.person.setName(name);
    } else {
      throw new Error('Invalid name');
    }
  }
  (...)
}
```

proxy.ts

```
// Client program uses Proxy
const proxy = new PersonProxy(new Person());
proxy.setName('Hugo');
proxy.setAge(20);
proxy.setName('esther'); // Error: Invalid name
proxy.setAge(20.25); // Error: Invalid age
```

proxy.ts

Behavioral Patterns

Behavioral Patterns

- It deals with communication between objects.

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

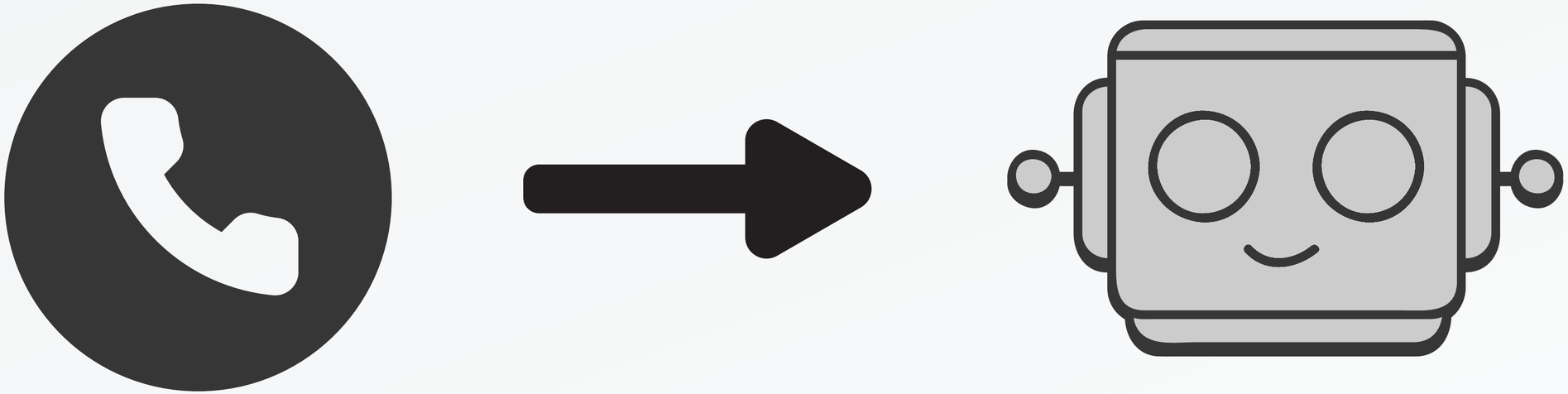
01 Chain of responsibility

06 State

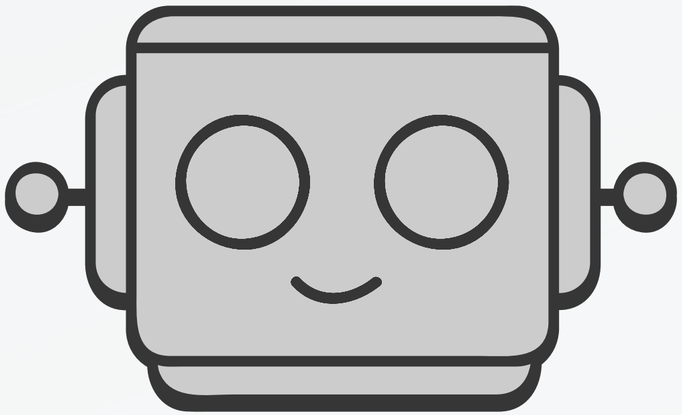
03 Iterator

05 Mediator

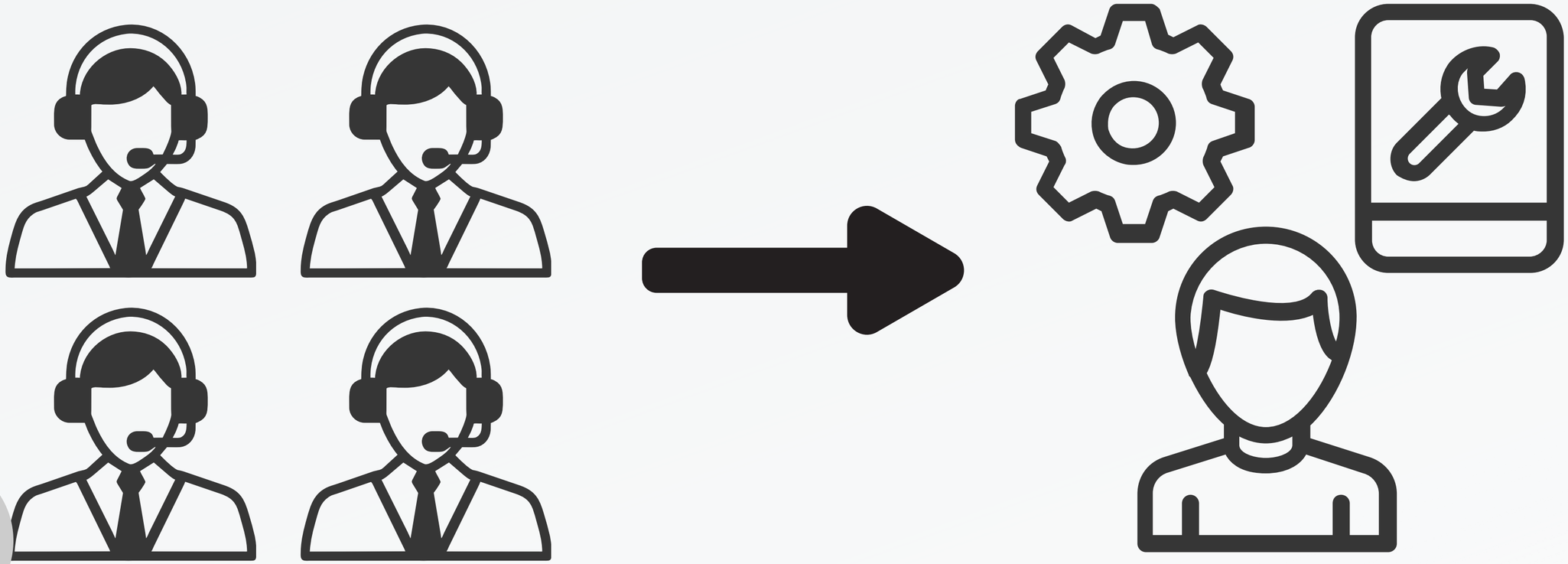
Chain of responsability



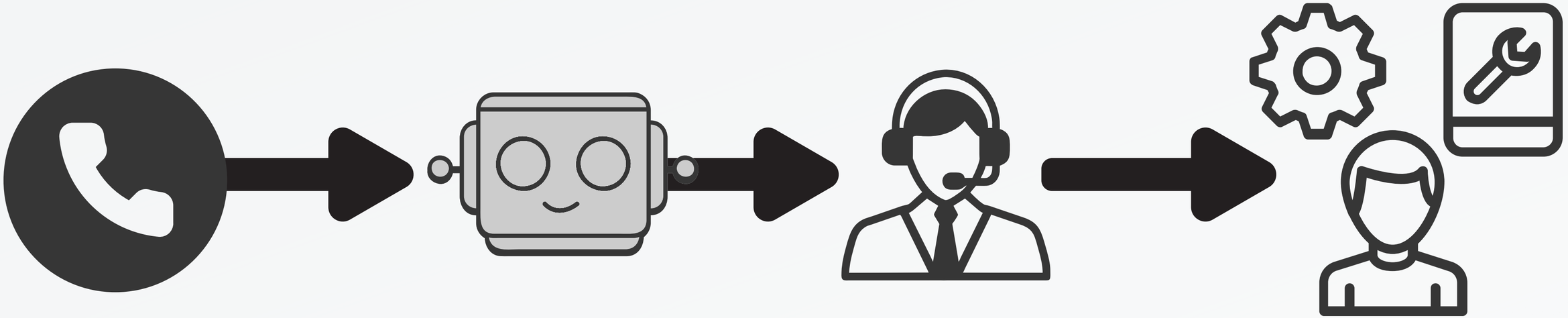
Chain of responsability



Chain of responsability

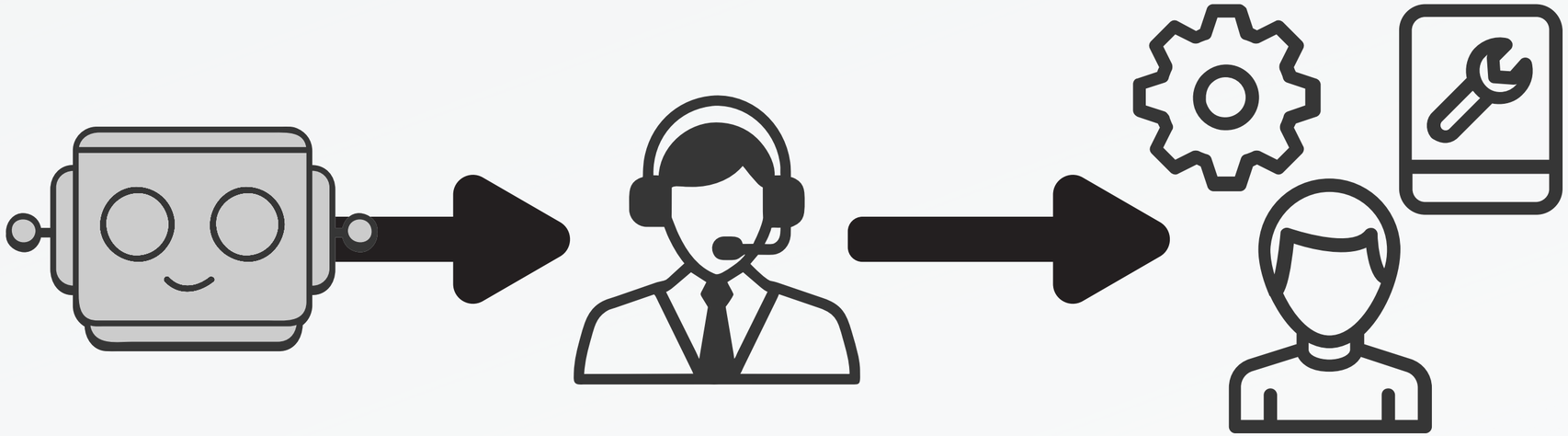


Chain of responsability



Chain of responsability

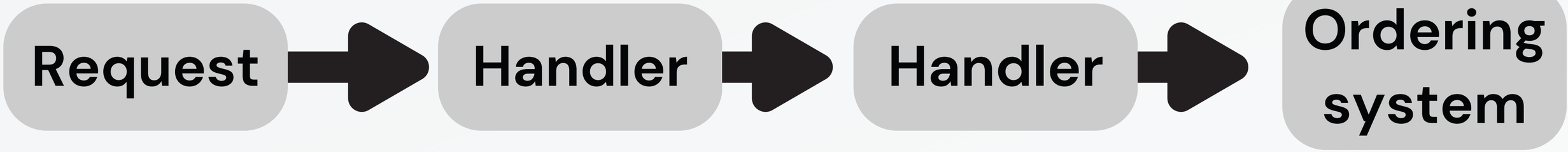
Request



Chain of responsability

Request → Handler → Handler → 

Chain of responsability



Chain of responsability

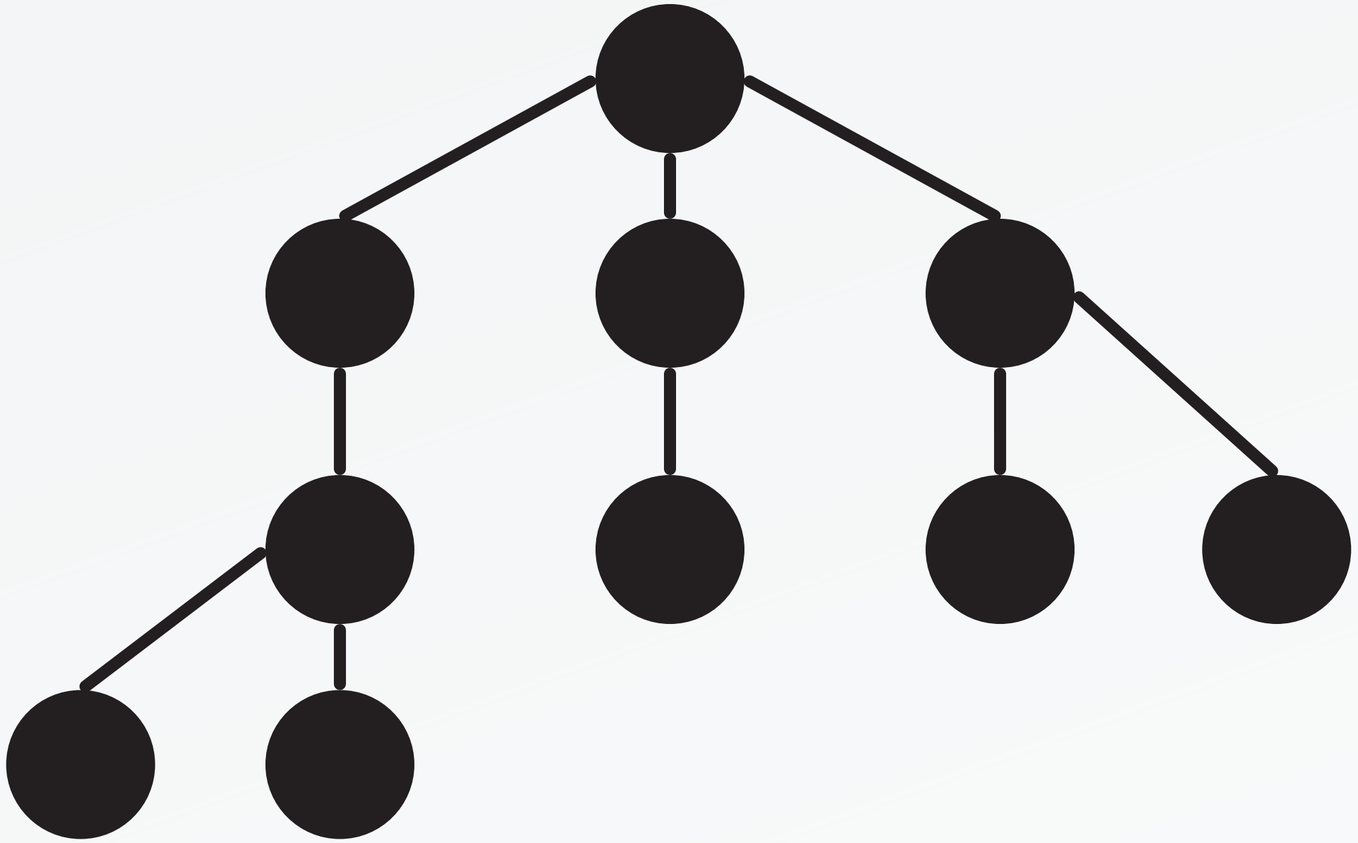
- It lets you pass requests along a chain of handlers.

Chain of responsability

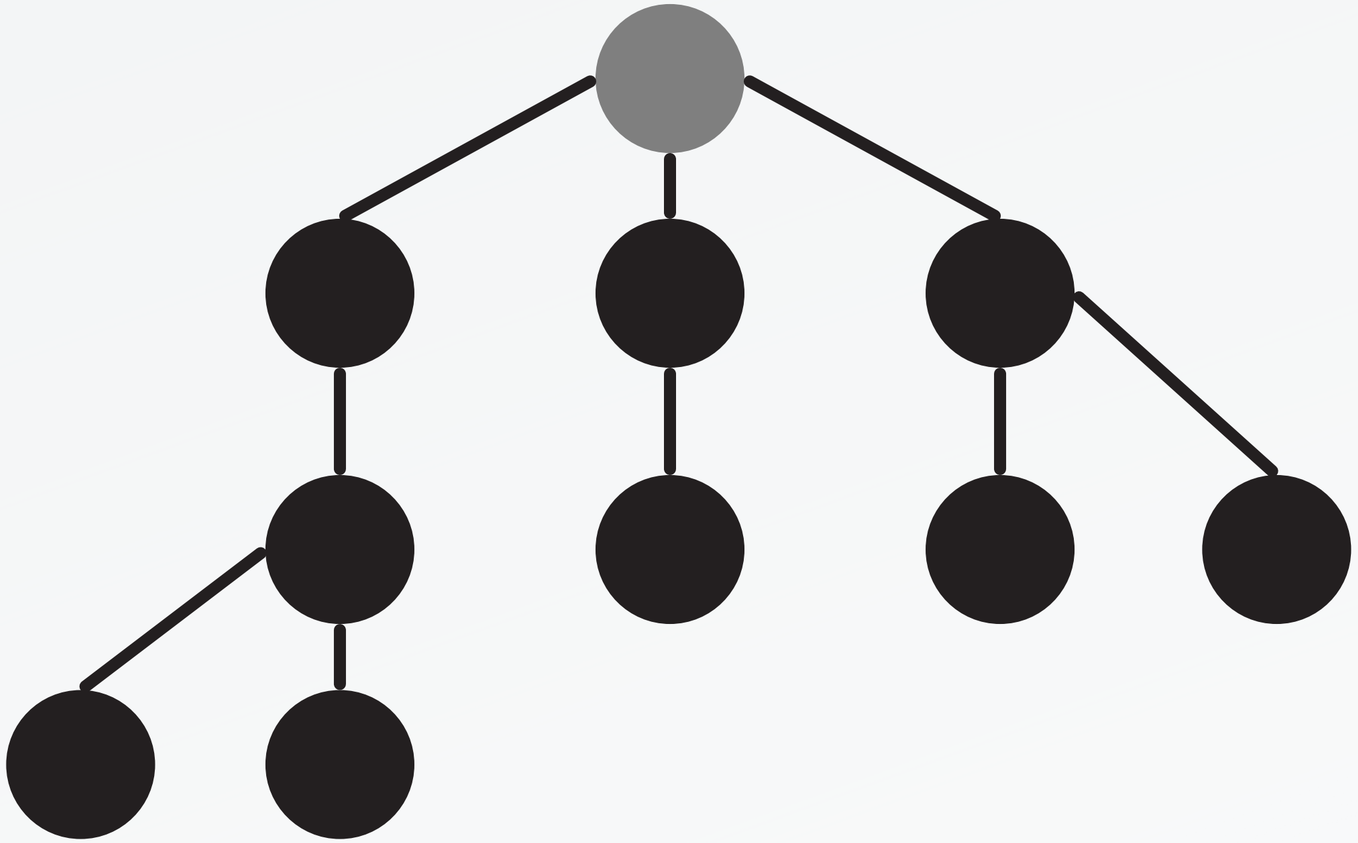
- It lets you pass requests along a chain of handlers.
  - Each handler decides:
    - Process the request
    - Pass it to the next handler.

chain.ts

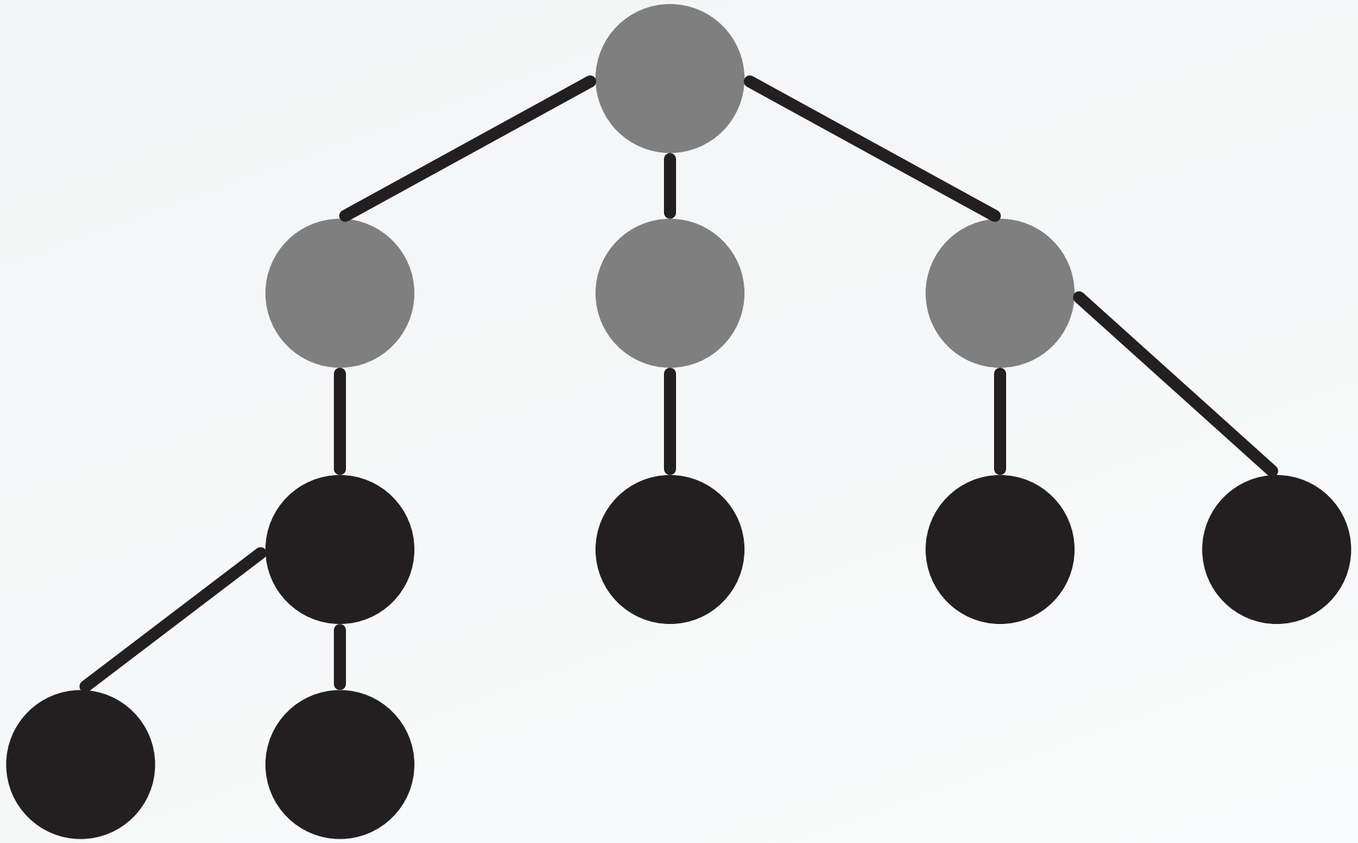
Iterator



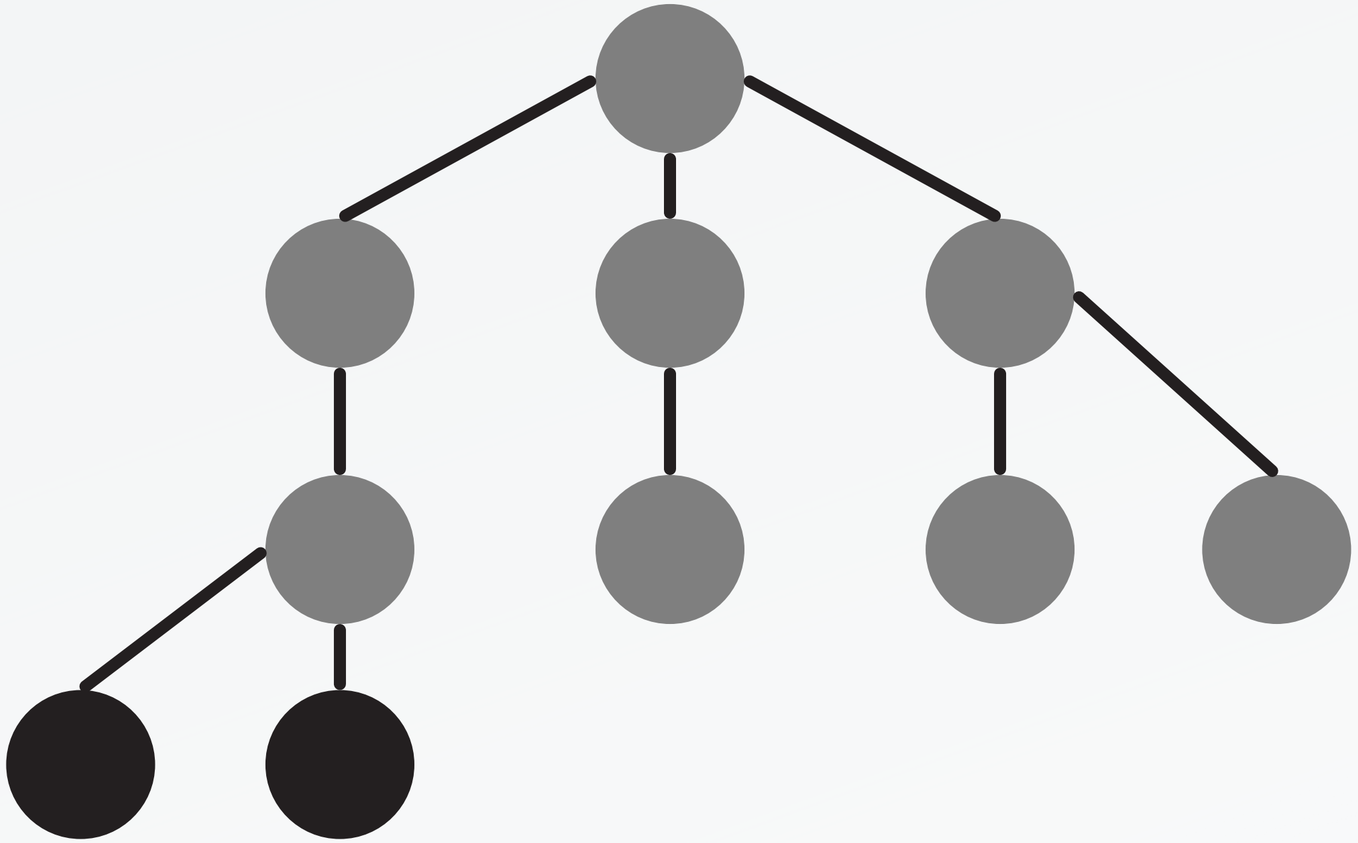
Iterator



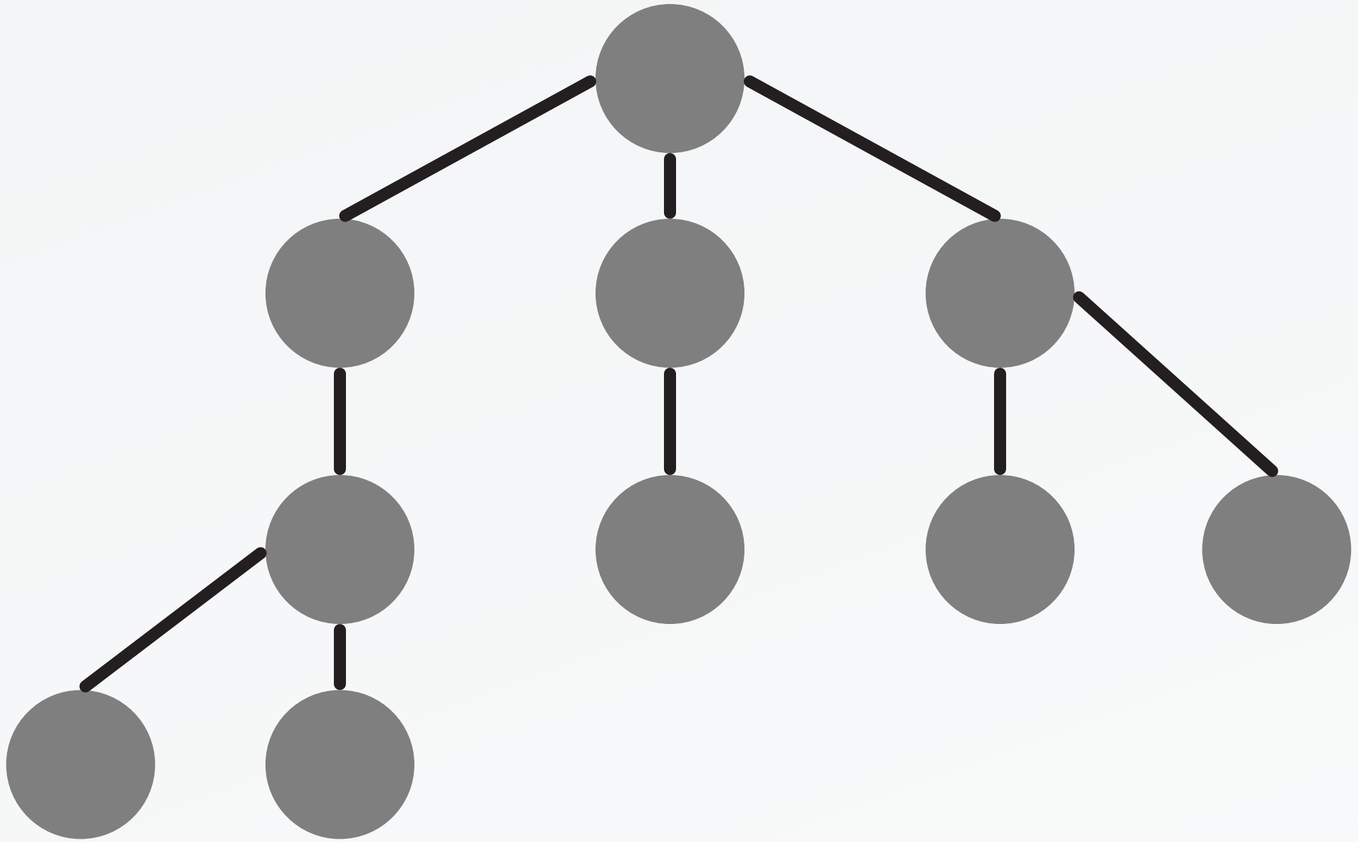
Iterator



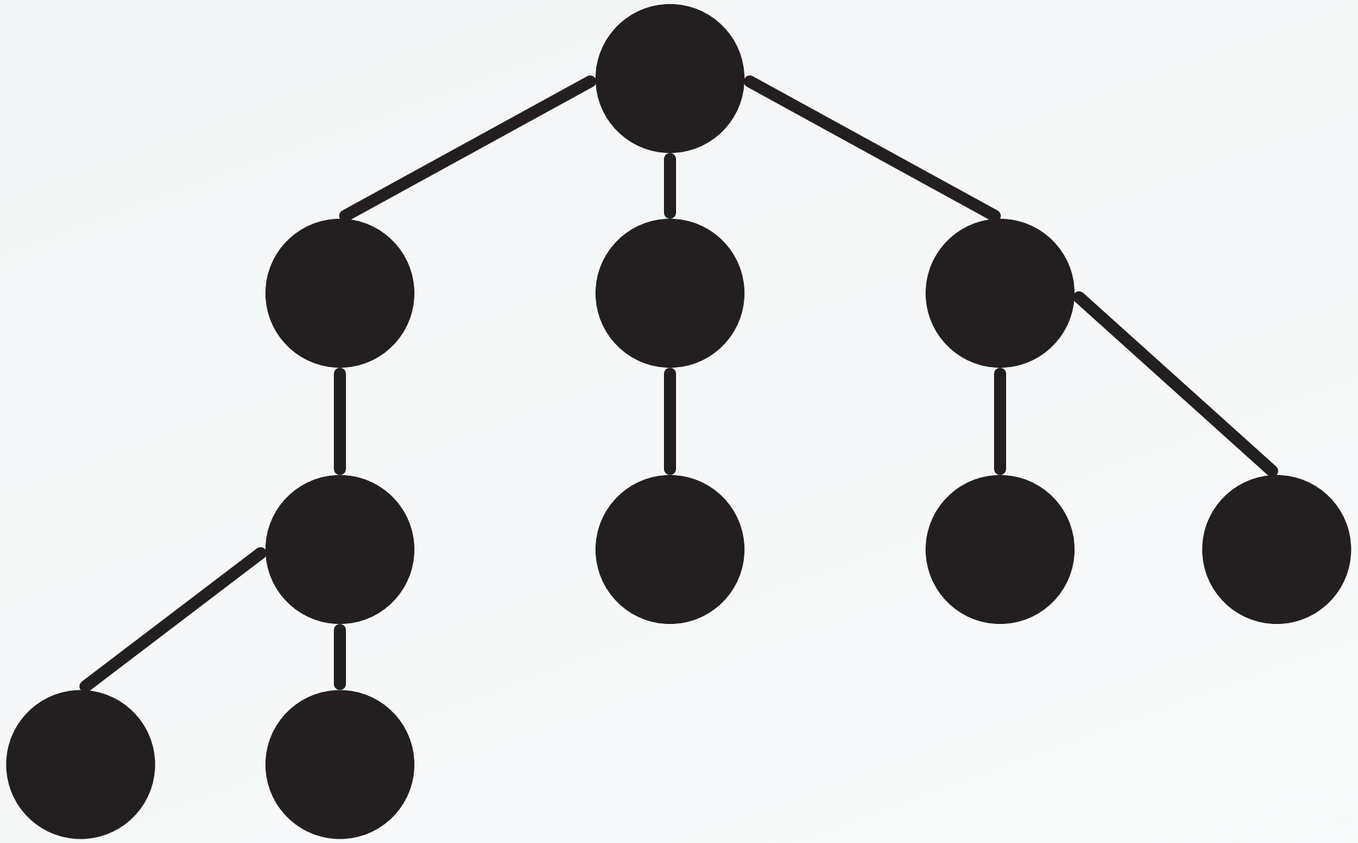
Iterator



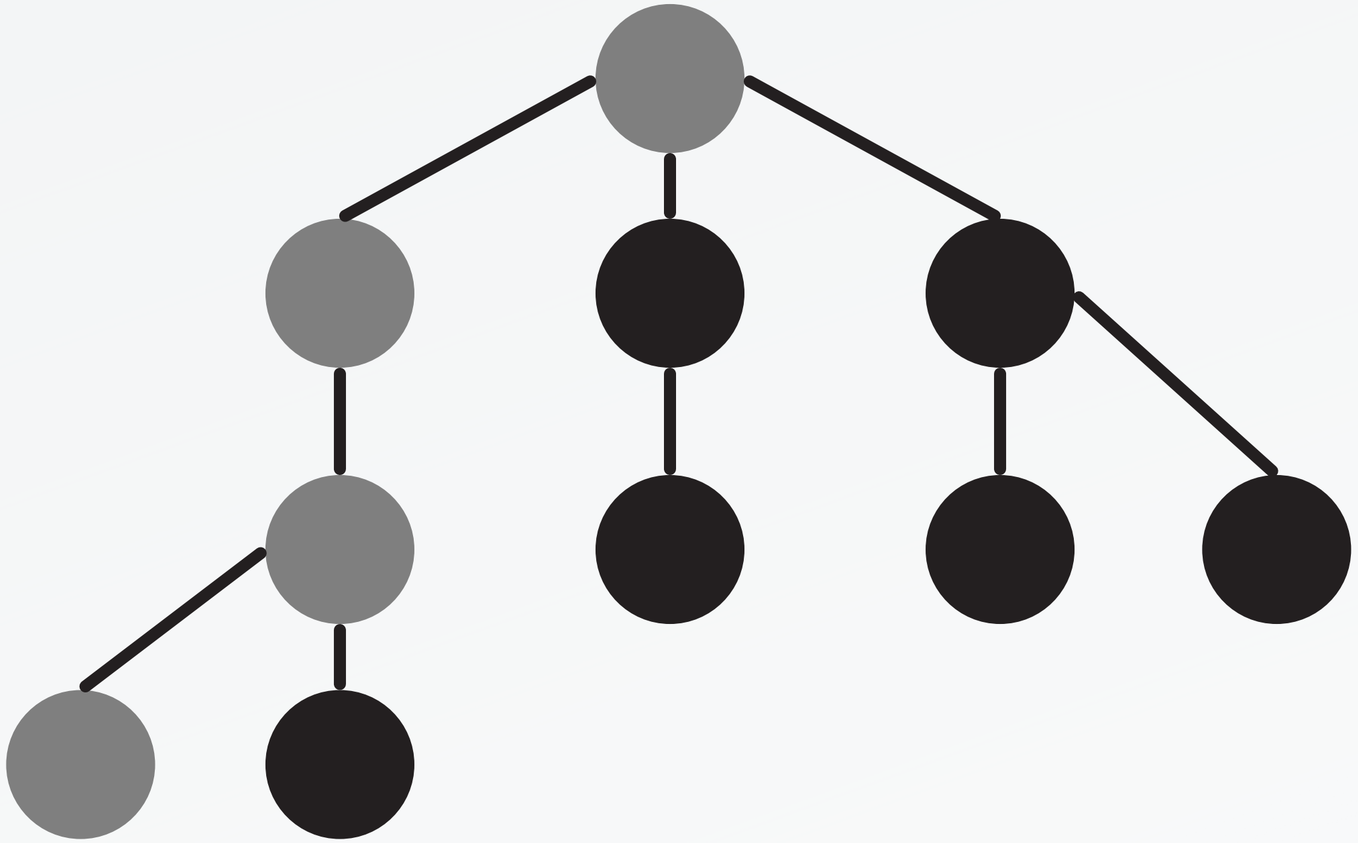
Iterator



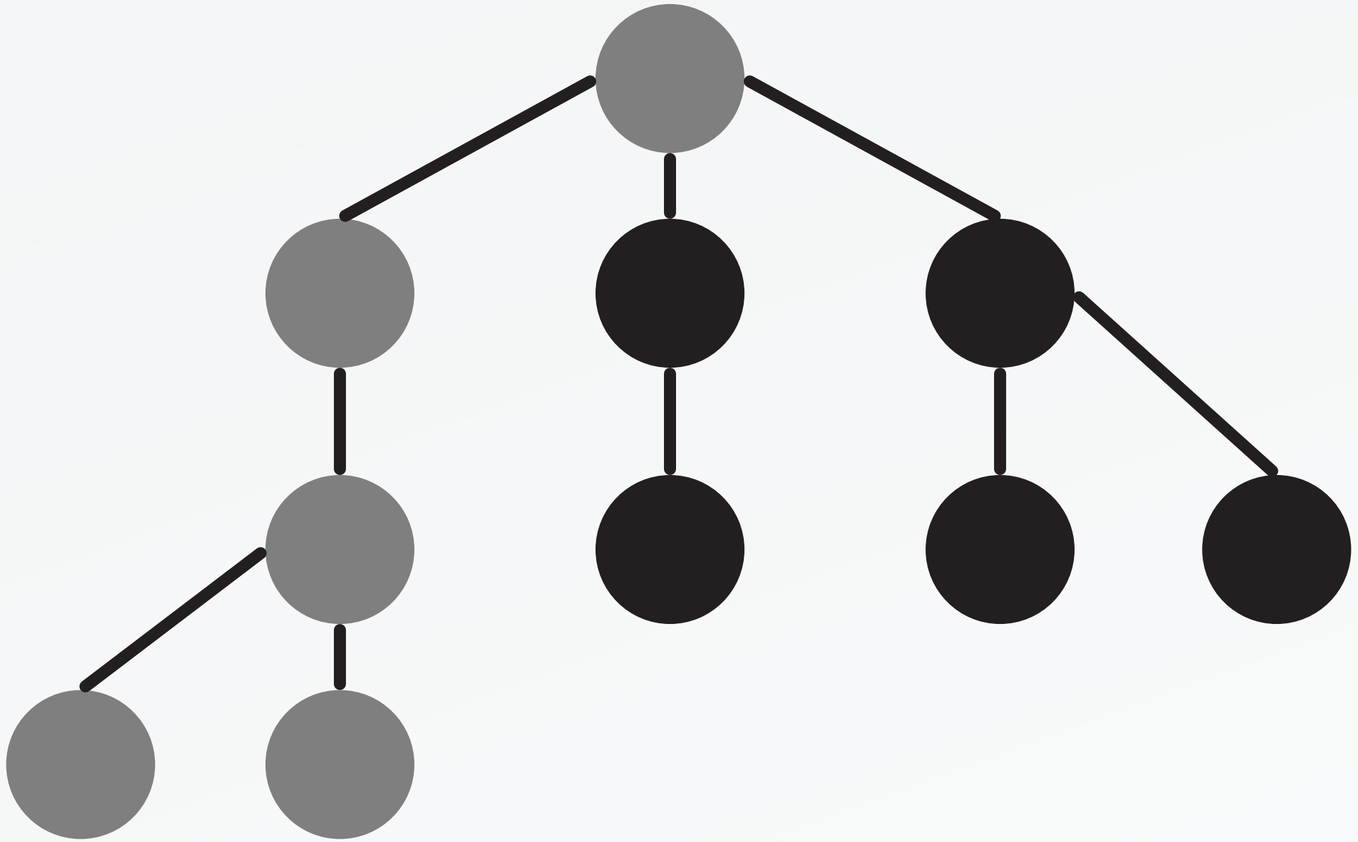
Iterator



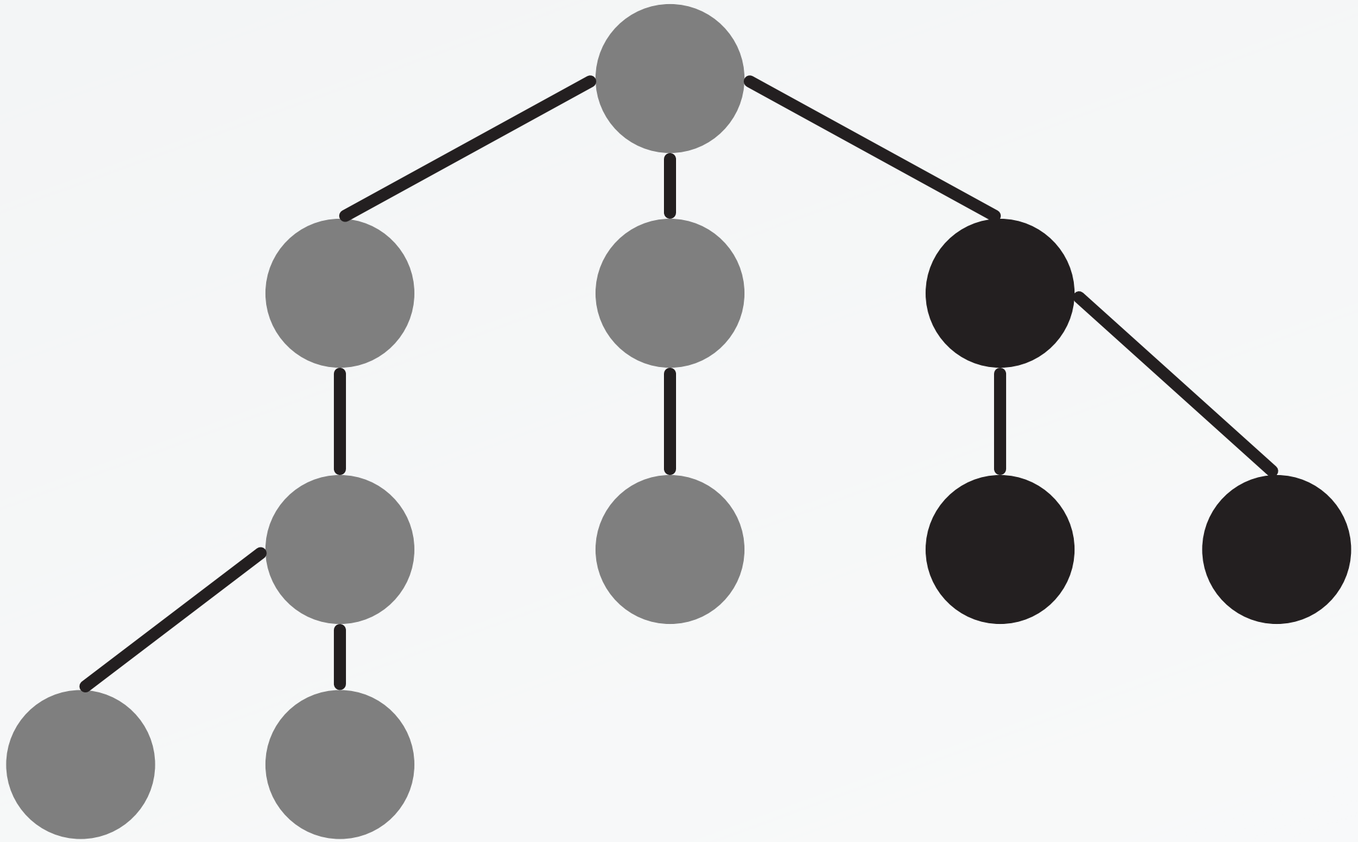
Iterator



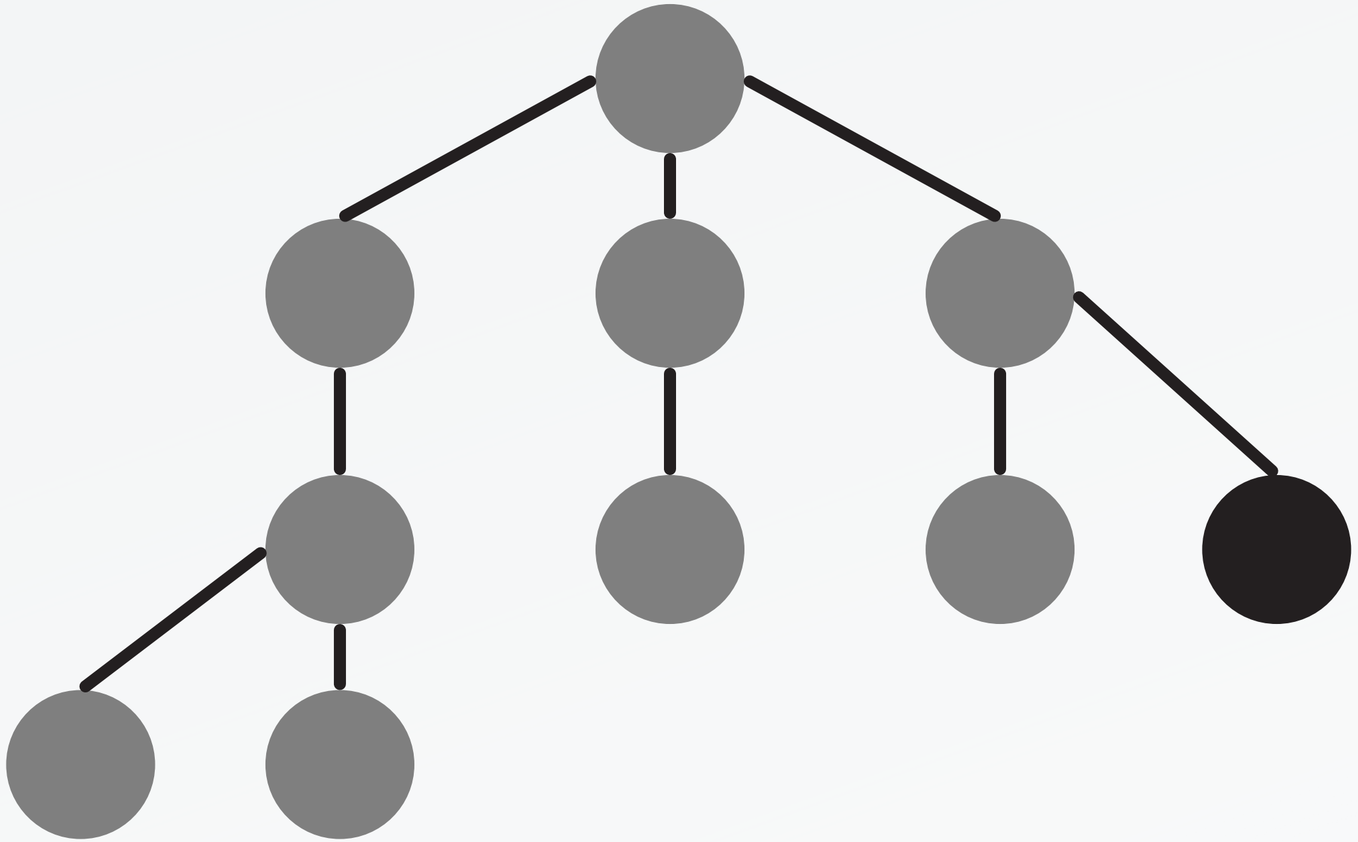
Iterator



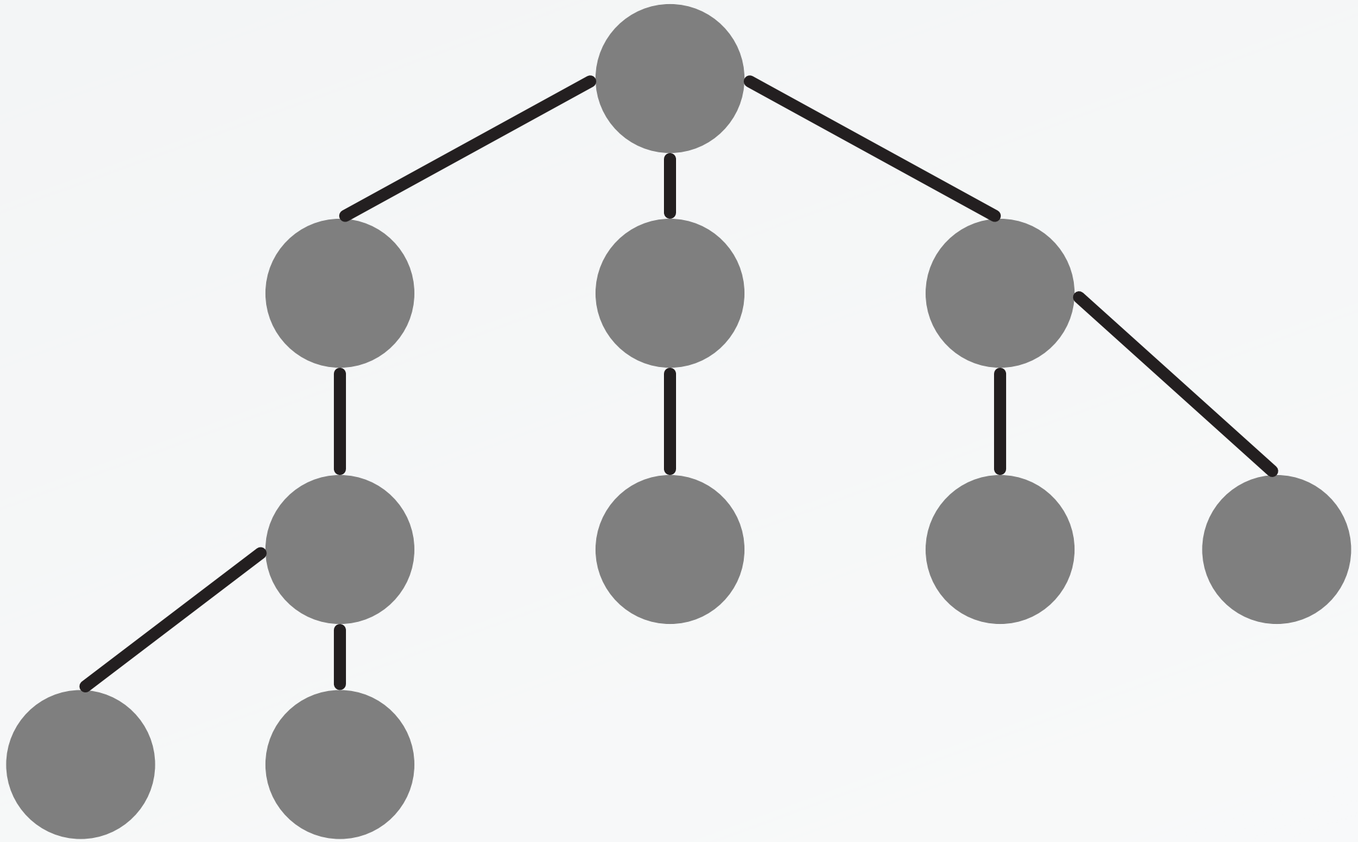
Iterator



Iterator



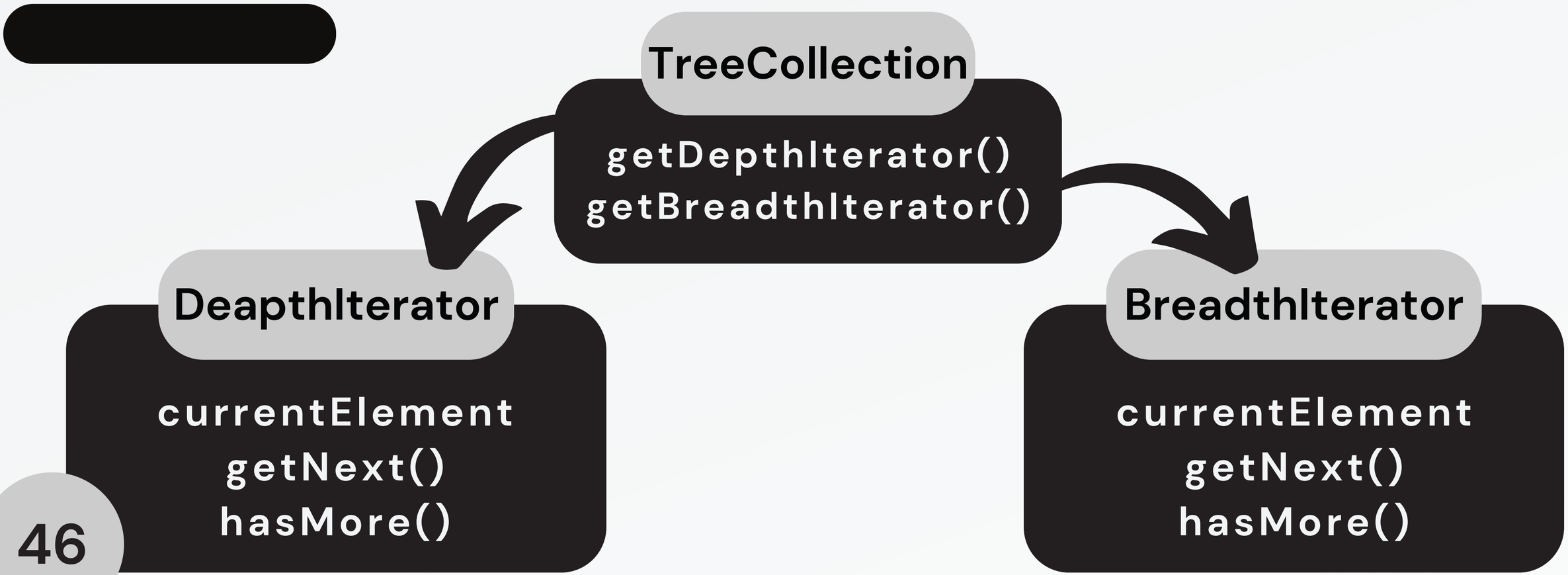
Iterator



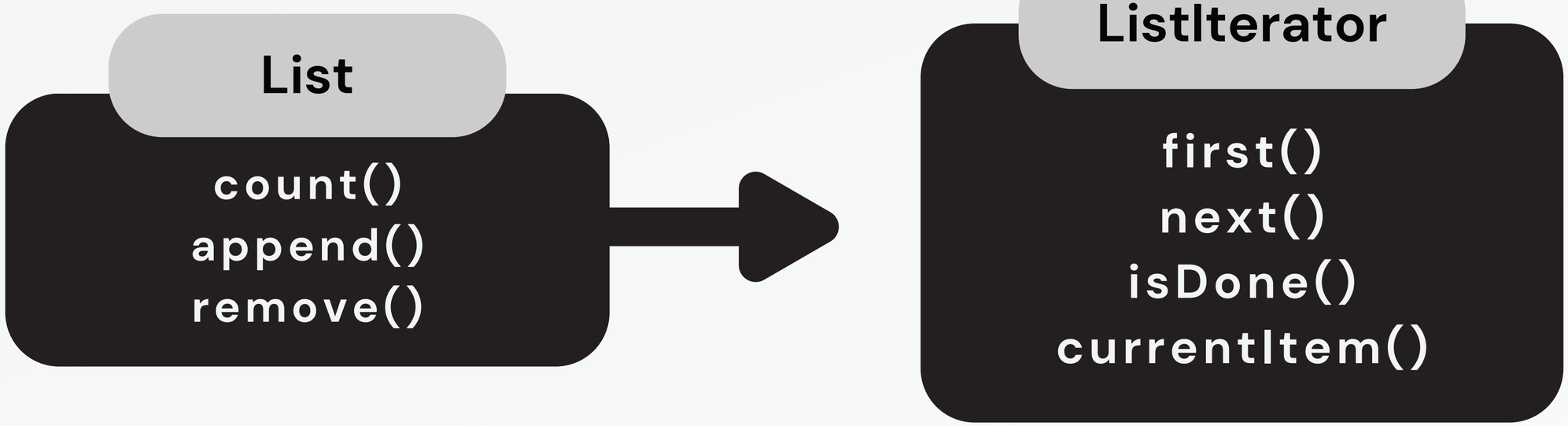
Iterator

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

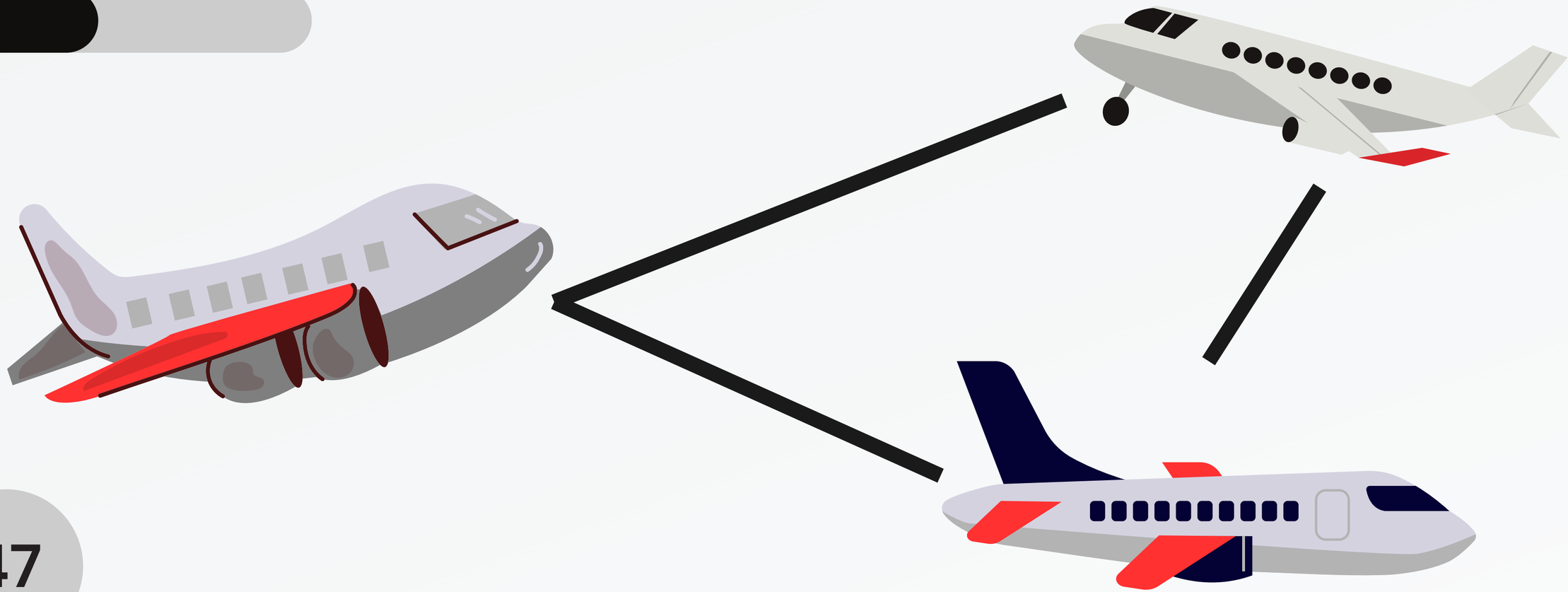
Iterator


TreeCollection
getDepthIterator()
getBreadthIterator()
DeapthIterator
currentElement
getNext()
hasMore()
BreadthIterator
currentElement
getNext()
hasMore()

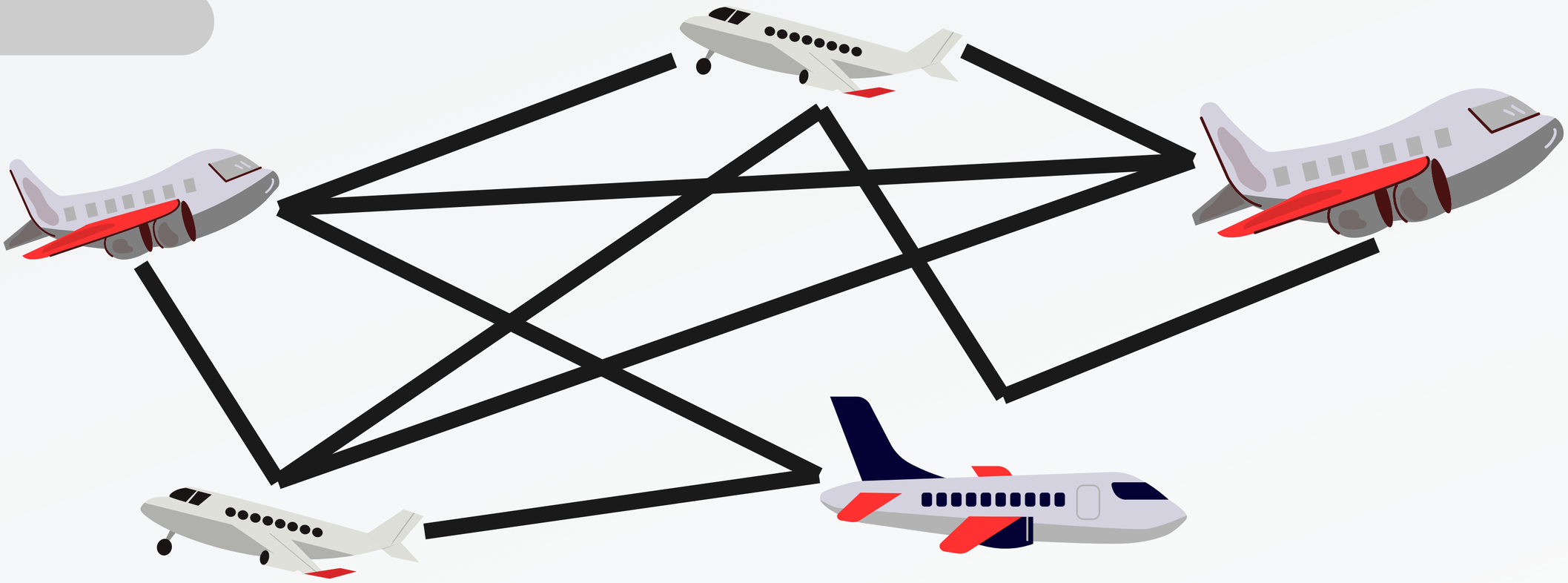
Iterator


List
count()
append()
remove()
ListIterator
first()
next()
isDone()
currentItem()

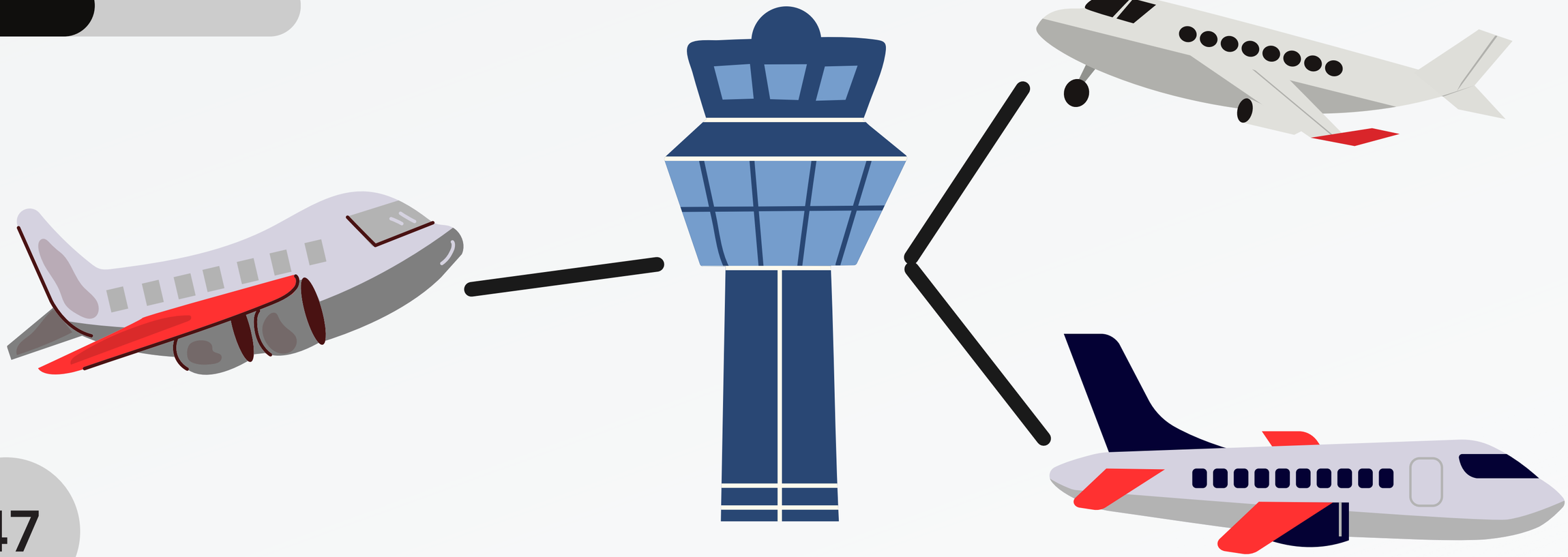
Mediator



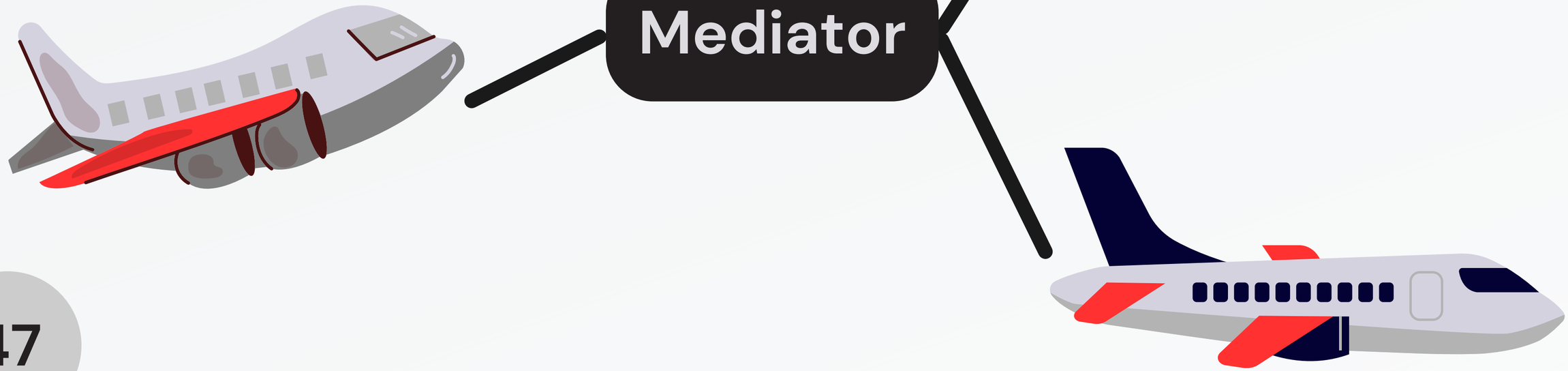
Mediator



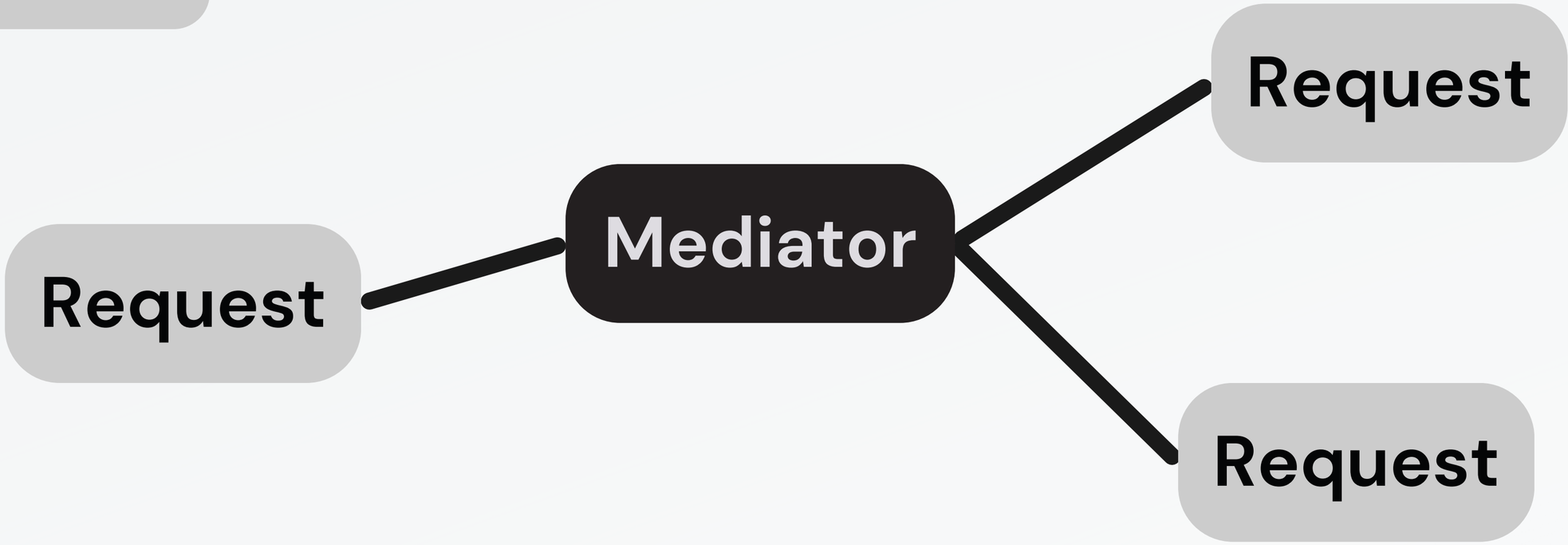
Mediator



Mediator



Mediator



```
// 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

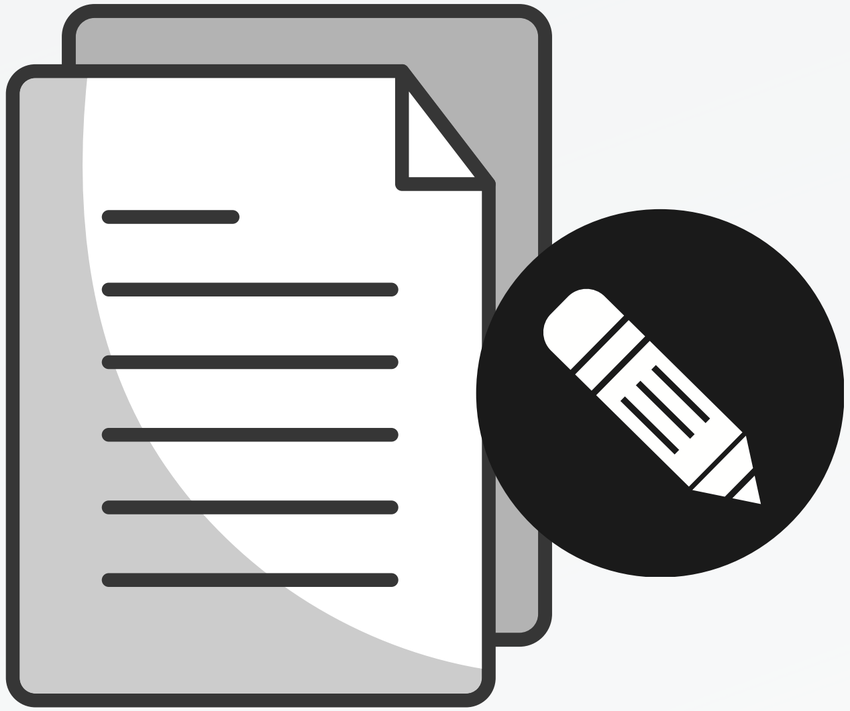
```
// Component (interacts with mediator)
class User {
  constructor(private name: string,
              private chatroom: ChatRoom) {}
  getName() { return this.name; }
  send(message: string): void {
    this.chatroom.logMessage(this, message);
  }
}
```


mediator.ts

```ts
// Client program
const CHATROOM: ChatRoom = new ChatRoom();
const USER_HUGO: User = new User('Hugo', CHATROOM);
const USER_ESTHER: User = new User('Esther', CHATROOM);
USER_HUGO.send("Hi there!");
USER_ESTHER.send("Hey!");
```


mediator.ts

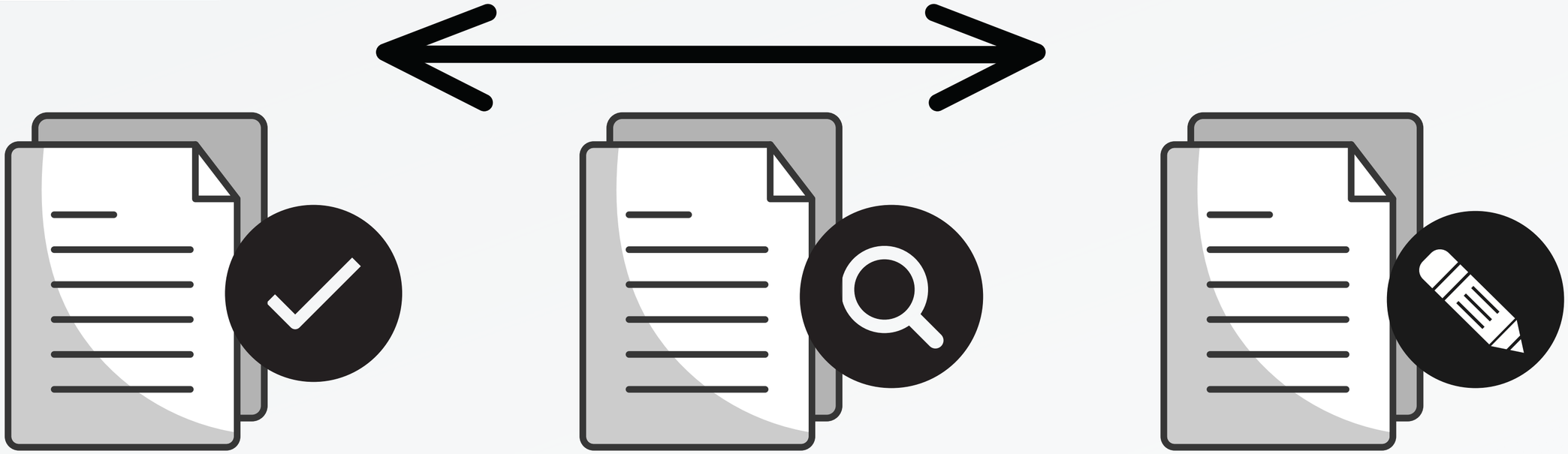
State



State

State

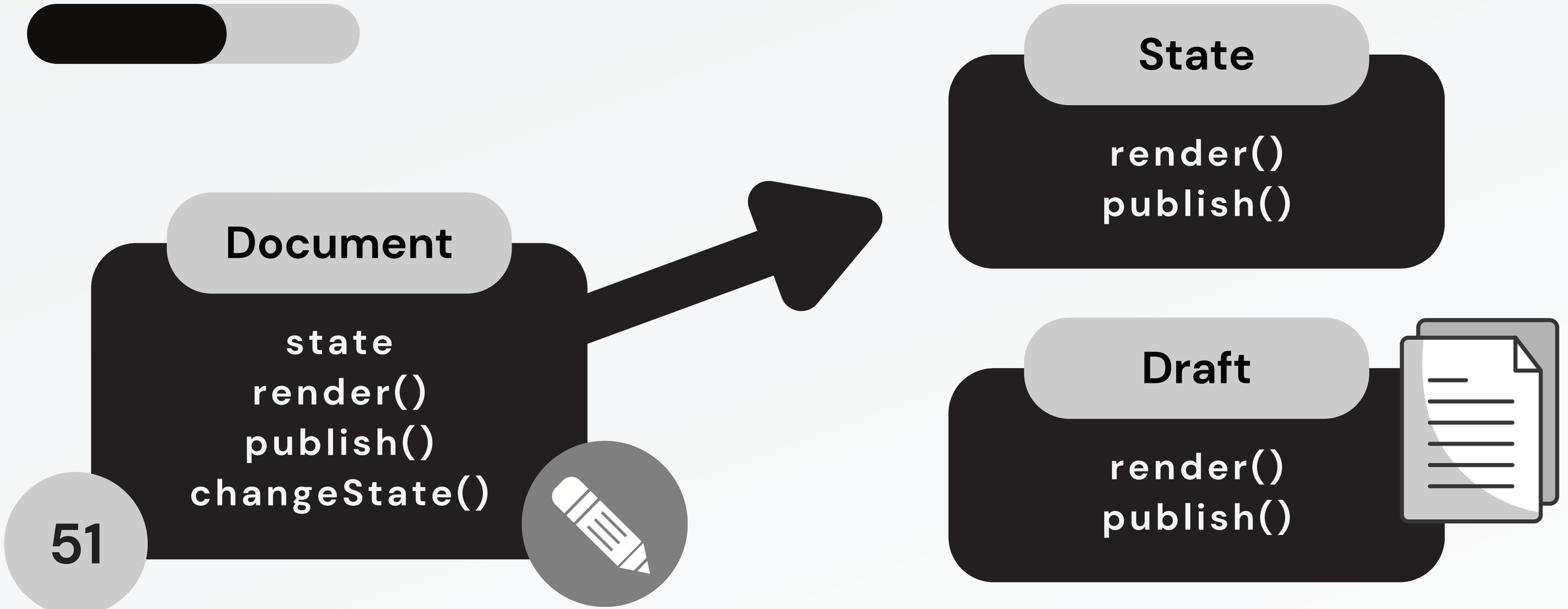
State



State

It lets an object alter its behavior when its internal state changes.

State


Document
state
render()
publish()
changeState()
State
render()
publish()
Draft
render()
publish()

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


state.ts

```
class HappyState implements State {
  think() {
    return 'I am happy';
  }
}
```


state.ts

```
class SadState implements State {
  think() {
    return 'I am sad';
  }
}
```


state.ts

```
class Human {
  state: State;
  constructor() { this.state = new HappyState(); }
  changeState(state) { this.state = state; }
  think() { return this.state.think(); }
}
```


state.ts

Why use Design Patterns?

Better code

Better code

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

Proven solutions

Proven solutions

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

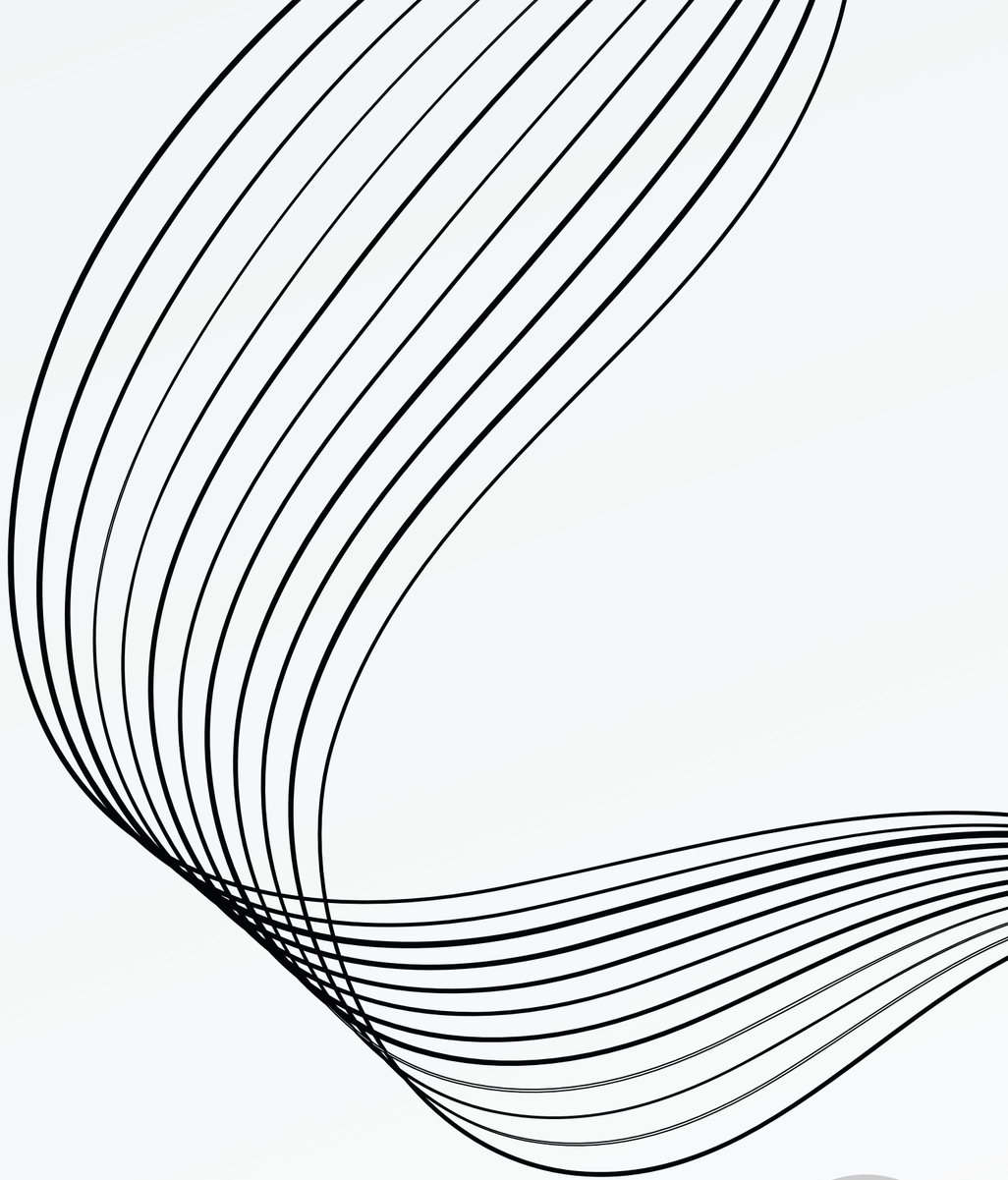
Refactoring

Refactoring

- You won’t need it.
- Optimal solution.

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/

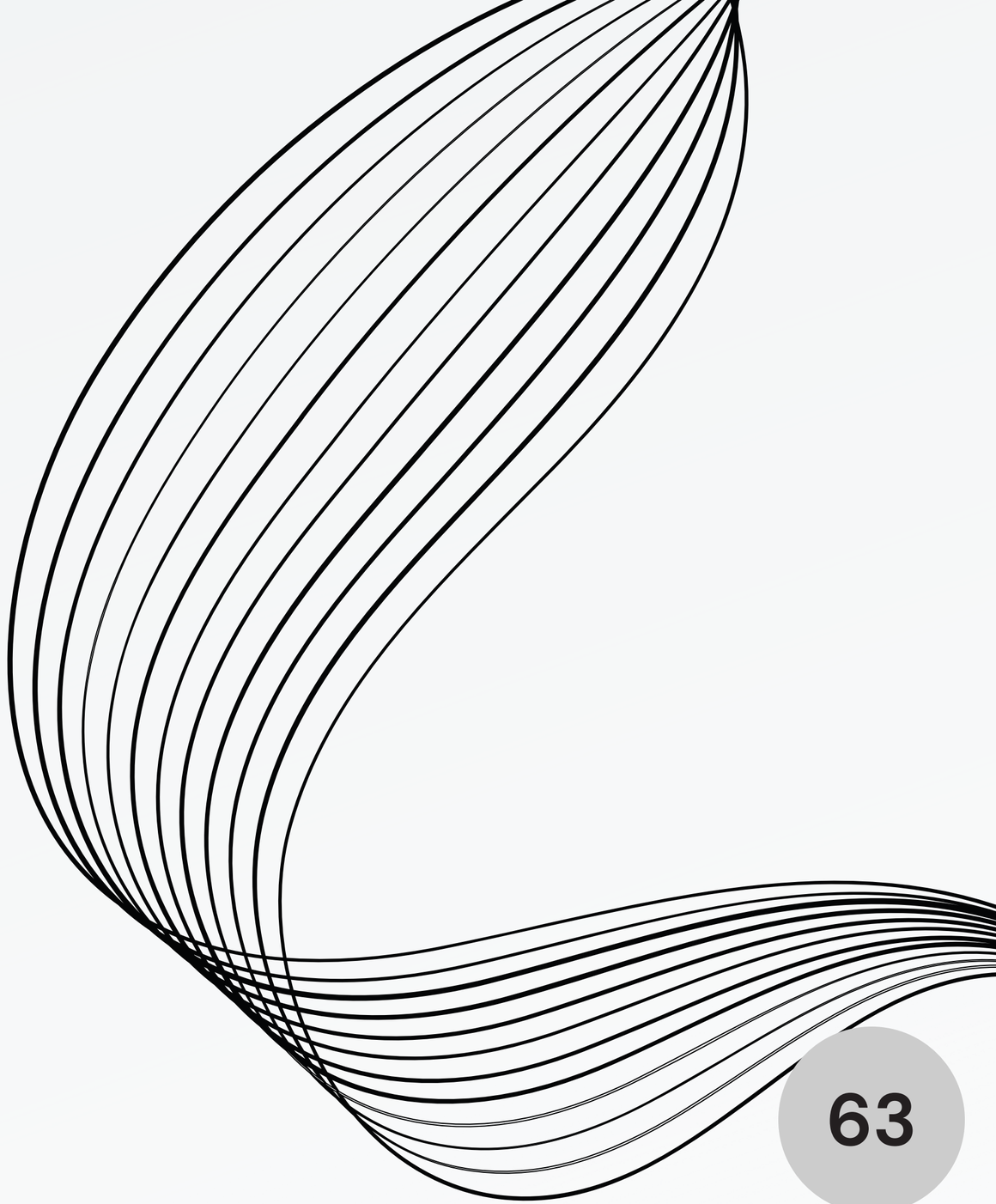
Go4 - Design patterns

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/

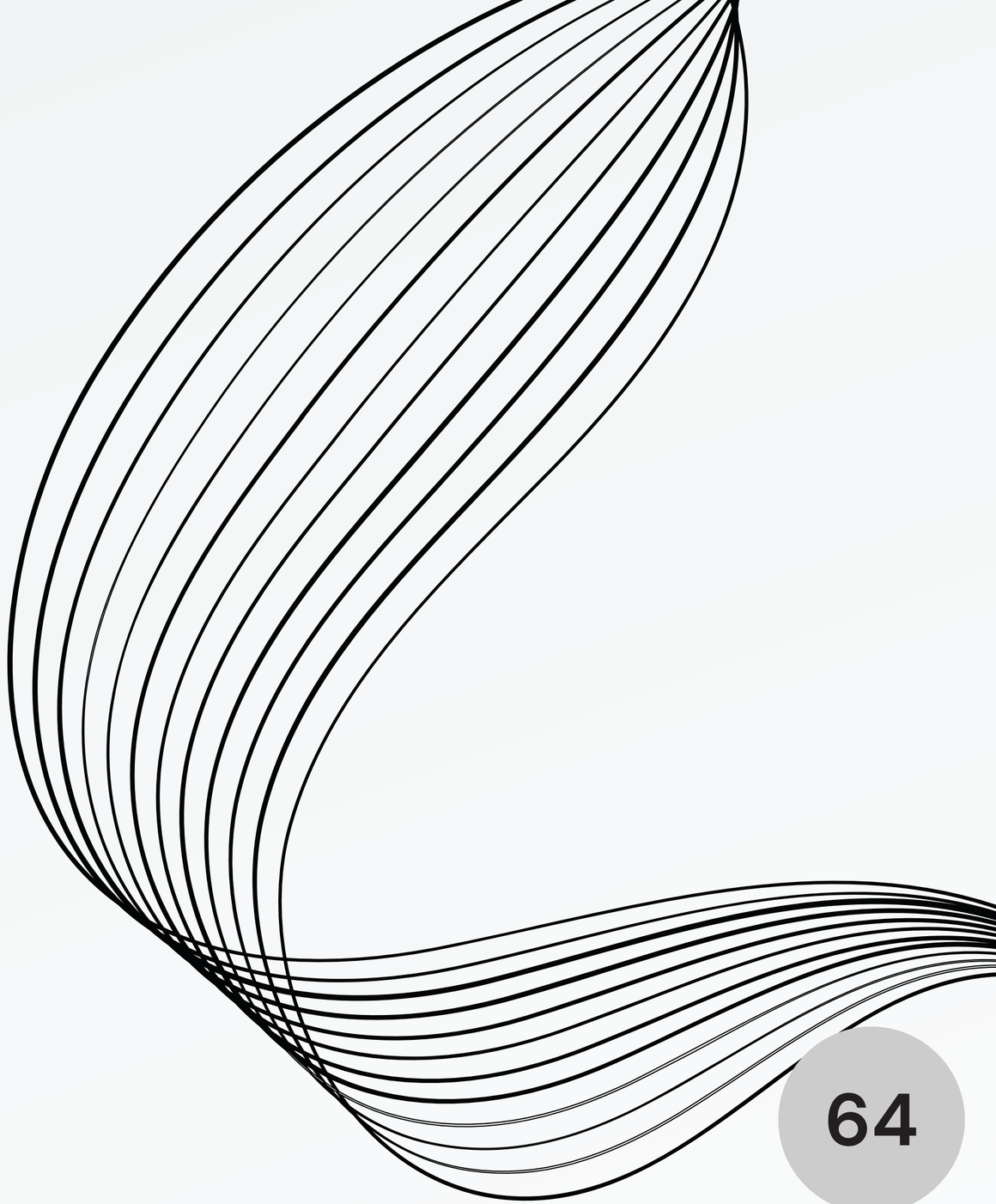
THANK'S FOR WATCHING!



esther.quintero.33@ull.edu.es

hugo.hernandez.14@ull.edu.es

Any questions?



Universidad de La Laguna