CEN206 Object-Oriented Programming

Behavioral Design Patterns

Author: Asst. Prof. Dr. Ugur CORUH

List of Figures

List of Tables

CEN206 Object-Oriented Programming

Week-11 (Behavioral Design Patterns)

Spring Semester, 2025-2026 Download DOC-PDF[1], DOC-DOCX[2], SLIDE[3]

Behavioral Design Patterns

Weekly Outline

- **Module A:** Introduction to Behavioral Patterns
- **Module B:** Chain of Responsibility Pattern
- **Module C:** Command Pattern
- **Module D:** Iterator Pattern

[1] ce204-week-11.en.md_doc.pdf

[2] ce204-week-11.en.md_word.docx

[3] ce204-week-11.en.md_slide.pdf

- **Module E:** Mediator Pattern
- **Module F:** Memento Pattern
- **Module G:** Observer Pattern
- **Module H:** State Pattern
- **Module I:** Strategy Pattern
- **Module J:** Template Method Pattern
- **Module K:** Visitor Pattern

Module A: Introduction to Behavioral Patterns

Module A: Outline

- What Are Behavioral Design Patterns?
- Why Are They Important?
- Overview of All 10 Behavioral Patterns
- Classification and Relationships

What Are Behavioral Design Patterns?

Behavioral design patterns are concerned with **algorithms** and the **assignment of responsibilities** between objects.

They describe not just patterns of objects or classes but also the **patterns of communication** between them. These patterns characterize complex control flow that is difficult to follow at run-time.

They shift your focus away from flow of control to let you concentrate on the way objects are interconnected.

Source: refactoring.guru[4]

Why Are Behavioral Patterns Important?

- They help manage **complex control flows** in object-oriented systems
- They define clear **communication protocols** between objects
- They promote **loose coupling** between collaborating objects
- They make it easier to **add new behaviors** without modifying existing code
- They encapsulate **varying behavior** behind stable interfaces

---

The 10 Behavioral Design Patterns

Pattern	Purpose
Chain of Responsibility	Pass requests along a chain of handlers
Command	Turn requests into stand-alone objects
Iterator	Traverse collections without exposing internals
Mediator	Reduce chaotic dependencies between objects
Memento	Save and restore previous states

---

[4] https://refactoring.guru/design-patterns/behavioral-patterns

The 10 Behavioral Design Patterns (cont.)

Pattern	Purpose
Observer	Notify multiple objects about state changes
State	Alter behavior when internal state changes
Strategy	Define interchangeable algorithm families
Template Method	Define algorithm skeleton, defer steps to subclasses
Visitor	Separate algorithms from the objects they operate on

Source: refactoring.guru[5]

Classification of Behavioral Patterns

Class-based patterns use inheritance to distribute behavior: - Template Method

Object-based patterns use object composition: - Chain of Responsibility, Command, Iterator, Mediator, Memento, Observer, State, Strategy, Visitor

---

Relationships Between Behavioral Patterns

- **Chain of Responsibility, Command, Mediator, Observer** all address different ways of connecting senders and receivers of requests
- **Command and Memento** often work together for undo functionality
- **Iterator and Visitor** combine for traversing complex structures
- **State and Strategy** share similar structure but differ in intent
- **Template Method and Strategy** represent class vs. object alternatives

---

Behavioral Patterns Overview Diagram

Module A: Takeaway

Behavioral patterns focus on how objects **communicate** and **distribute responsibilities**. They are essential for building flexible, maintainable systems where behaviors can be changed or extended without modifying existing code. In the following modules, we will study each of the 10 behavioral patterns in depth.

Module B: Chain of Responsibility Pattern

Module B: Outline

- Intent
- Problem
- Solution
- Real-World Analogy

[5] https://refactoring.guru/design-patterns/behavioral-patterns

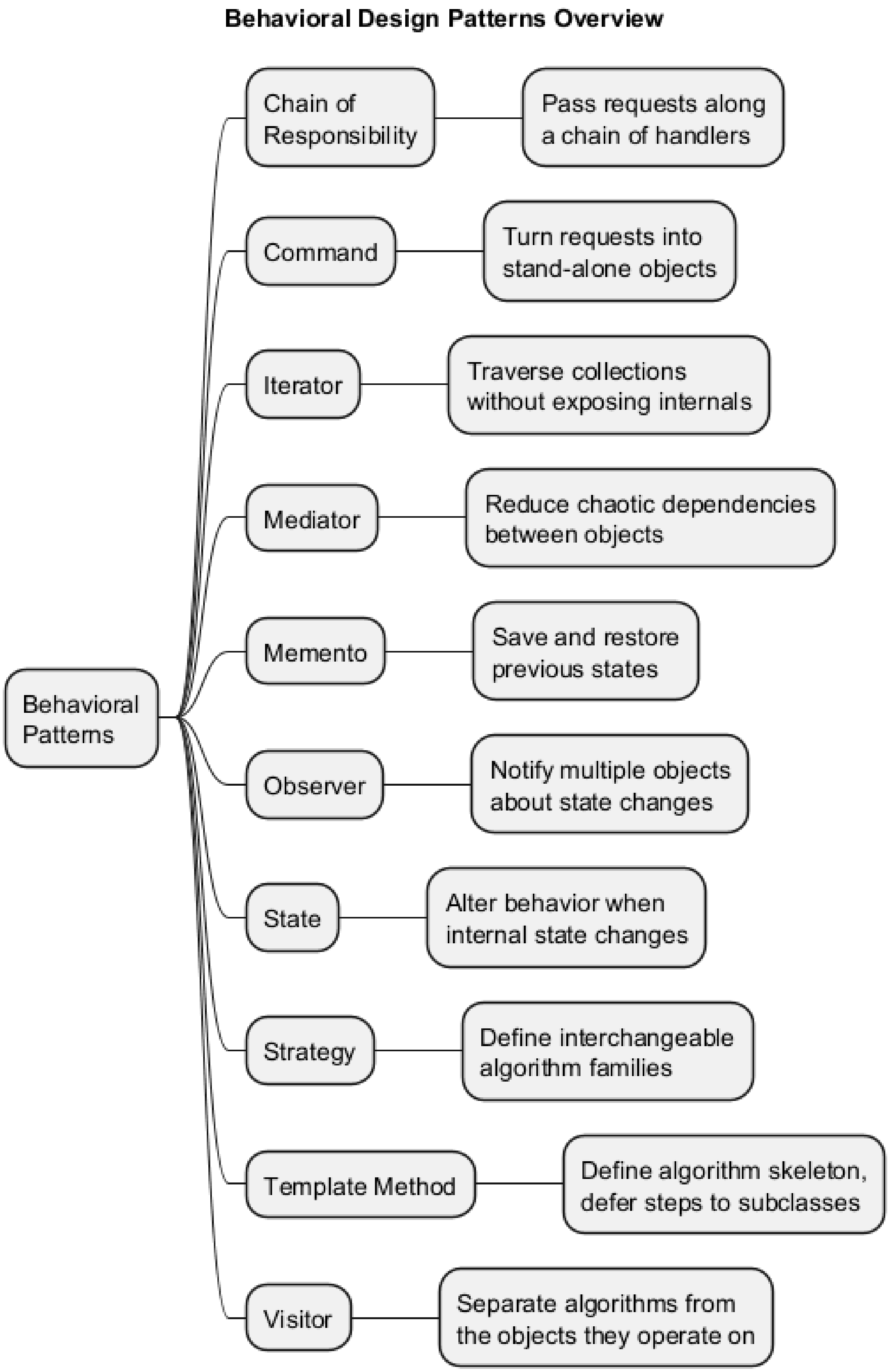
Behavioral Design Patterns Overview
Behavioral Patterns
Chain of Responsibility
Pass requests along a chain of handlers
Command
Turn requests into stand-alone objects
Iterator
Traverse collections without exposing internals
Mediator
Reduce chaotic dependencies between objects
Memento
Save and restore previous states
Observer
Notify multiple objects about state changes
State
Alter behavior when internal state changes
Strategy
Define interchangeable algorithm families
Template Method
Define algorithm skeleton, defer steps to subclasses
Visitor
Separate algorithms from the objects they operate on

Figure 1: center

- Structure
- Java Pseudocode Example
- Applicability
- How to Implement
- Pros and Cons
- Relations with Other Patterns

Chain of Responsibility: Intent

Chain of Responsibility is a behavioral design pattern that lets you pass requests along a chain of handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain.

Also known as: **CoR**, **Chain of Command**

Source: refactoring.guru[6]

Chain of Responsibility: Problem

Imagine you are working on an online ordering system. You want to restrict access so only authenticated users can create orders. Also, users with admin permissions should have full access to all orders.

After some planning, you realize these checks must be performed sequentially:

1. **Authentication** – Is the user logged in?
2. **Authorization** – Does the user have permissions?
3. **Validation** – Is the data valid?
4. **Caching** – Is the result already cached?

Chain of Responsibility: Problem (cont.)

Over time, you add more sequential checks:

- **Rate limiting** to prevent brute-force password attacks
- **Sanitization** to filter raw data in requests
- **Cross-site scripting** protection

The code of the checks, which had been already looking like a mess, became more and more bloated as you added each new feature. Changing one check sometimes affected the others. Reusing checks in other components required code duplication.

Chain of Responsibility: Solution

The Chain of Responsibility pattern suggests that you link handlers into a chain. Each handler has a field for storing a reference to the next handler. In addition to processing a request, handlers pass the request further along the chain.

The request travels along the chain until all handlers have had a chance to process it. A handler can decide **not to pass** the request further down the chain, effectively stopping any further processing.

---

[6] https://refactoring.guru/design-patterns/chain-of-responsibility

Chain of Responsibility: Solution (cont.)

There is a slightly different approach where, upon receiving a request, a handler decides whether it **can** process it. If it can, it does not pass the request any further. So it is either only one handler that processes the request or none at all.

This approach is very common when dealing with events in stacks of elements within a graphical user interface. For example, when a user clicks a button, the event propagates through the chain of GUI elements, starting from the button, going along its containers, and ending with the application window.

Chain of Responsibility: Real-World Analogy

You have just bought and installed a new piece of hardware on your computer. You call tech support:

1. The **automated system** suggests several standard solutions – none work for you.
2. A **human operator** tries scripted troubleshooting steps – still not resolved.
3. The operator **escalates** your call to an **engineer** who finally fixes the problem.

Each tier either resolves the issue or escalates it to the next, more capable, tier.

Chain of Responsibility: Structure

The structure consists of the following participants:

1. **Handler** (interface): Declares the interface for handling requests. It usually contains a single method for handling requests, and sometimes another for setting the next handler.

2. **Base Handler** (abstract class): Optional class where you can put boilerplate code common to all handlers. Usually stores a reference to the next handler and implements default chaining behavior.

Chain of Responsibility: Structure (cont.)

3. **Concrete Handlers**: Contain the actual code for processing requests. Upon receiving a request, each handler must decide whether to process it and whether to pass it along the chain. Handlers are usually self-contained and immutable, accepting all necessary data via the constructor.

4. **Client**: May compose chains just once or compose them dynamically. A request can be sent to any handler in the chain – it does not have to be the first one.

Chain of Responsibility: Structure Diagram

Chain of Responsibility: Java Pseudocode

```java
// Handler interface
interface Handler {
    Handler setNext(Handler handler);
    String handle(String request);
}
```

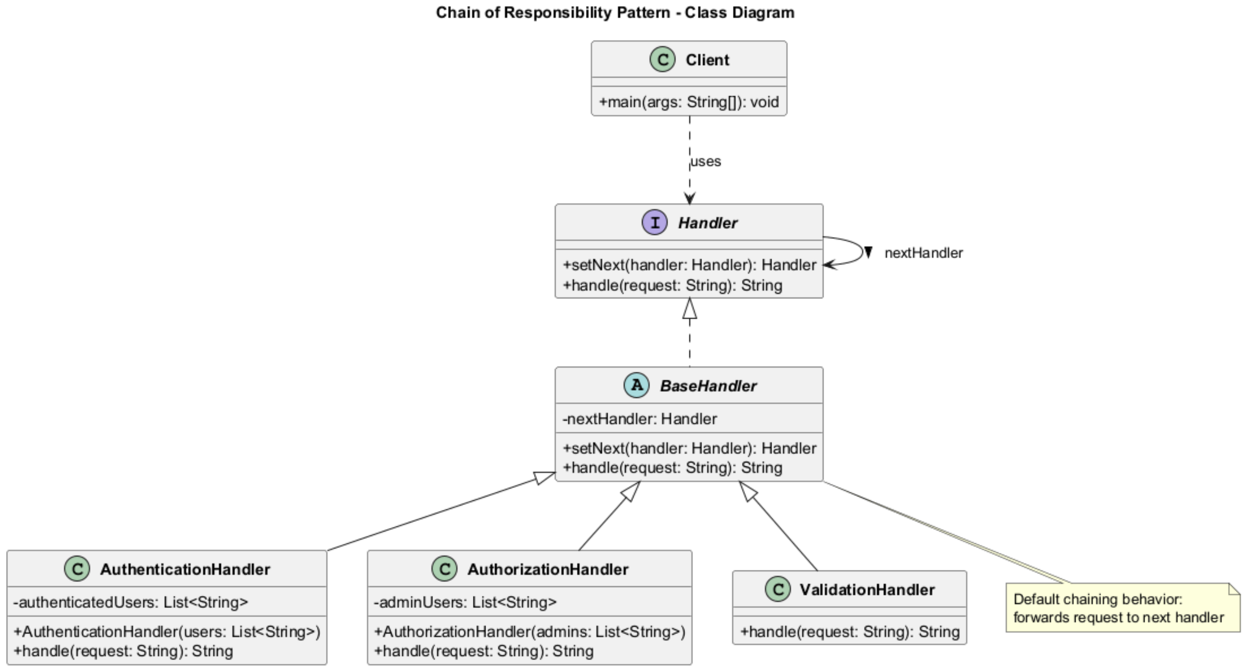
Chain of Responsibility Pattern - Class Diagram
C Client
+main(args: String[]): void
uses
I Handler
+setNext(handler: Handler): Handler
+handle(request: String): String
nextHandler
A BaseHandler
-nextHandler: Handler
+setNext(handler: Handler): Handler
+handle(request: String): String
C AuthenticationHandler
-authenticatedUsers: List<String>
+AuthenticationHandler(users: List<String>)
+handle(request: String): String
C AuthorizationHandler
-adminUsers: List<String>
+AuthorizationHandler(admins: List<String>)
+handle(request: String): String
C ValidationHandler
+handle(request: String): String
Default chaining behavior:
forwards request to next handler

Figure 2: center

Chain of Responsibility: Java Pseudocode (cont.)

```java
// Base handler with default chaining behavior
abstract class BaseHandler implements Handler {
    private Handler nextHandler;

    @Override
    public Handler setNext(Handler handler) {
        this.nextHandler = handler;
        // Returning handler allows chaining:
        // monkey.setNext(squirrel).setNext(dog);
        return handler;
    }

    @Override
    public String handle(String request) {
        if (this.nextHandler != null) {
            return this.nextHandler.handle(request);
        }
        return null;
    }
}
```

Chain of Responsibility: Java Pseudocode (cont.)

```java
import java.util.List;

// Concrete handler: Authentication
class AuthenticationHandler extends BaseHandler {
    private List<String> authenticatedUsers;
    public AuthenticationHandler(List<String> users) {
        this.authenticatedUsers = users;
    }

    @Override
    public String handle(String request) {
        if (!authenticatedUsers.contains(request)) {
            return "AuthenticationHandler: "
                + request + " is NOT authenticated.";
        }
        System.out.println("AuthenticationHandler: "
            + request + " is authenticated.");
        return super.handle(request);
    }
}
```

Chain of Responsibility: Java Pseudocode (cont.)

```
// Concrete handler: Authorization
class AuthorizationHandler extends BaseHandler {
    private List<String> adminUsers;

    public AuthorizationHandler(List<String> admins) {
        this.adminUsers = admins;
    }

    @Override
    public String handle(String request) {
        if (!adminUsers.contains(request)) {
            return "AuthorizationHandler: "
                + request + " is NOT authorized.";
        }
        System.out.println("AuthorizationHandler: "
            + request + " is authorized.");
        return super.handle(request);
    }
}
```

Chain of Responsibility: Java Pseudocode (cont.)

```
// Concrete handler: Validation
class ValidationHandler extends BaseHandler {
    @Override
    public String handle(String request) {
        if (request == null || request.isEmpty()) {
            return "ValidationHandler: "
                + "Request is invalid.";
        }
        System.out.println("ValidationHandler: "
            + request + " is valid.");
        return super.handle(request);
    }
}
```

Chain of Responsibility: Java Pseudocode (cont.)

```
import java.util.Arrays;
import java.util.List;

// Client code
public class ChainOfResponsibilityDemo {
    public static void main(String[] args) {
        // Build the chain
        Handler auth = new AuthenticationHandler(
            Arrays.asList("alice", "bob", "charlie"));
        Handler authz = new AuthorizationHandler(
            Arrays.asList("alice", "charlie"));
        Handler validation = new ValidationHandler();

        auth.setNext(authz).setNext(validation);

        // Send requests through the chain
        for (String user : Arrays.asList(
                "alice", "bob", "dave")) {
            String result = auth.handle(user);
            if (result != null) {
                System.out.println(result);
            } else {
                System.out.println(user
                    + ": Request passed all checks.");
            }
        }

        // Expected Output:
        // AuthenticationHandler: alice is authenticated.
        // AuthorizationHandler: alice is authorized.
        // ValidationHandler: alice is valid.
        // alice: Request passed all checks.
        // AuthenticationHandler: bob is authenticated.
        // AuthorizationHandler: bob is NOT authorized.
        // AuthenticationHandler: dave is NOT authenticated.
    }
}
```

Chain of Responsibility: Sequence Diagram

Chain of Responsibility: Applicability

Use the Chain of Responsibility pattern when:

- Your program is expected to process different kinds of requests in various ways, but the **exact types of requests** and their sequences are **unknown beforehand**.
- It is essential to **execute several handlers** in a particular order.
- The set of handlers and their order are supposed to **change at runtime**. If you provide setters for a reference field inside handler classes, you will be able to insert, remove, or reorder handlers dynamically.

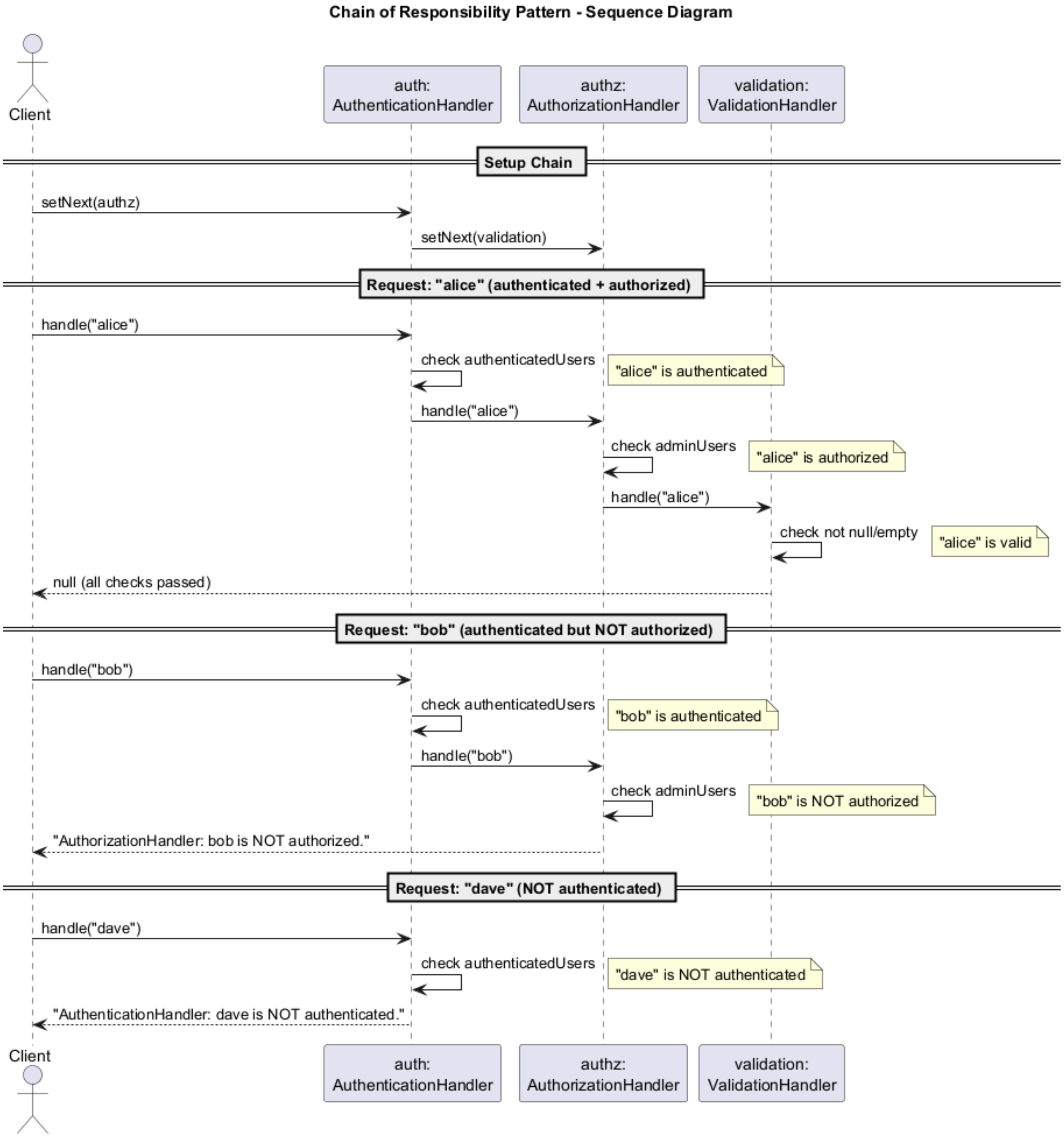

Figure 3: center

Chain of Responsibility: How to Implement

1. Declare the handler interface and describe the signature of a method for handling requests.
2. To eliminate duplicate boilerplate code in concrete handlers, create an **abstract base handler** class derived from the handler interface. Add a field for storing a reference to the next handler and implement default forwarding behavior.
3. Create **concrete handler** subclasses and implement their handling methods. Each handler must make two decisions:
   - Whether it will process the request
   - Whether it will pass the request along the chain

Chain of Responsibility: How to Implement (cont.)

4. The **client** may either assemble chains on its own or receive pre-built chains from other objects. In the latter case, you must implement some factory classes to build chains according to configuration or environment settings.
5. The client may trigger any handler in the chain, not just the first one. The request will be passed along the chain until some handler refuses to pass it further or until it reaches the end.
6. Due to the dynamic nature of the chain, the client should be ready to handle these scenarios:
   - The chain may consist of a **single link**
   - Some requests may **not reach the end** of the chain
   - Others may reach the end of the chain **unhandled**

Chain of Responsibility: Pros and Cons

Pros:

- You can **control the order** of request handling
- **Single Responsibility Principle:** You can decouple classes that invoke operations from classes that perform operations
- **Open/Closed Principle:** You can introduce new handlers into the app without breaking the existing client code

Cons:

- Some requests may end up **unhandled** if no handler processes them

---

Chain of Responsibility: Relations with Other Patterns

- **Chain of Responsibility, Command, Mediator, and Observer** are alternative ways to connect senders and receivers. CoR passes a request sequentially along a dynamic chain.
- **CoR and Composite:** Leaf components can pass requests through the chain of all parent components down to the root of the object tree.
- **CoR and Decorator:** They are very similar in structure. Both rely on recursive composition to pass execution through a series of objects. However, CoR handlers can execute arbitrary operations independently of each other and can stop passing the request at any point. Decorators extend behavior while keeping the base interface consistent.

Module B: Takeaway

The **Chain of Responsibility** pattern decouples senders from receivers by giving multiple objects a chance to handle a request. The request is passed along a chain until an object handles it or the chain ends.

Module C: Command Pattern

Module C: Outline

- Intent
- Problem
- Solution
- Real-World Analogy
- Structure
- Java Pseudocode Example
- Applicability
- How to Implement
- Pros and Cons
- Relations with Other Patterns

Command: Intent

Command is a behavioral design pattern that turns a request into a stand-alone object that contains all information about the request. This transformation lets you pass requests as method arguments, delay or queue a request's execution, and support undoable operations.

Also known as: **Action**, **Transaction**

Source: refactoring.guru[7]

Command: Problem

Imagine you are working on a new text editor app. Your current task is to create a toolbar with a bunch of buttons for various operations. You created a `Button` class that can be used for buttons on the toolbar, as well as for generic buttons in dialogs.

While all of these buttons look similar, they are supposed to do different things. Where would you put the code for the various click handlers of these buttons?

Command: Problem (cont.)

The simplest solution is to create tons of subclasses for each place where the button is used. These subclasses would contain the code to execute on a button click.

Problems with this approach:

- You have an **enormous number of subclasses** that break any time you modify the base `Button` class
- GUI code becomes **dependent** on volatile business logic code
- Some operations (e.g., Copy/Paste) need to be invoked from **multiple places** (buttons, menus, keyboard shortcuts), leading to **code duplication**

---

[7] https://refactoring.guru/design-patterns/command

Command: Solution

Good software design is often based on the **principle of separation of concerns**. The Command pattern suggests that GUI objects should not send requests directly. Instead, you extract all of the request details into a separate **command** class with a single method that triggers this request.

Command objects serve as links between various GUI and business logic objects. The GUI object does not need to know what business logic object will receive the request and how it will be processed.

Command: Solution (cont.)

The next step is to make your commands implement the **same interface**. Usually it has just a single execution method that takes no parameters. This interface lets you use various commands with the same request sender, without coupling it to concrete classes.

As a bonus, you can now switch command objects linked to the sender, effectively changing the sender's behavior at runtime.

Command: Real-World Analogy

After a long walk through the city, you get to a nice restaurant and sit at the table by the window. A friendly waiter approaches you and quickly takes your **order**, writing it down on a piece of paper.

The waiter goes to the kitchen and sticks the order on the wall. After a while, the order gets to the **chef**, who reads it and cooks the meal accordingly.

The cook places the meal on a tray along with the order. The waiter discovers the tray, checks the order to make sure everything is as you wanted it, and brings everything to your table.

The paper order serves as a **command** – it remains in a queue until the chef is ready.

Command: Structure

The Command pattern has five key participants:

1. **Sender (Invoker)**: Responsible for initiating requests. Has a field for storing a reference to a command object. Triggers that command instead of sending the request directly to the receiver.
2. **Command Interface**: Usually declares just a single method for executing the command.

Command: Structure (cont.)

3. **Concrete Commands**: Implement various kinds of requests. A concrete command is not supposed to perform the work on its own, but rather to pass the call to one of the business logic objects (receiver). Parameters needed to execute a method on a receiver object can be declared as fields in the concrete command.
4. **Receiver**: Contains some business logic. Almost any object may act as a receiver. Most commands only handle the details of how a request is passed to the receiver.
5. **Client**: Creates and configures concrete command objects. The client must pass all request parameters, including a receiver instance, into the command constructor.

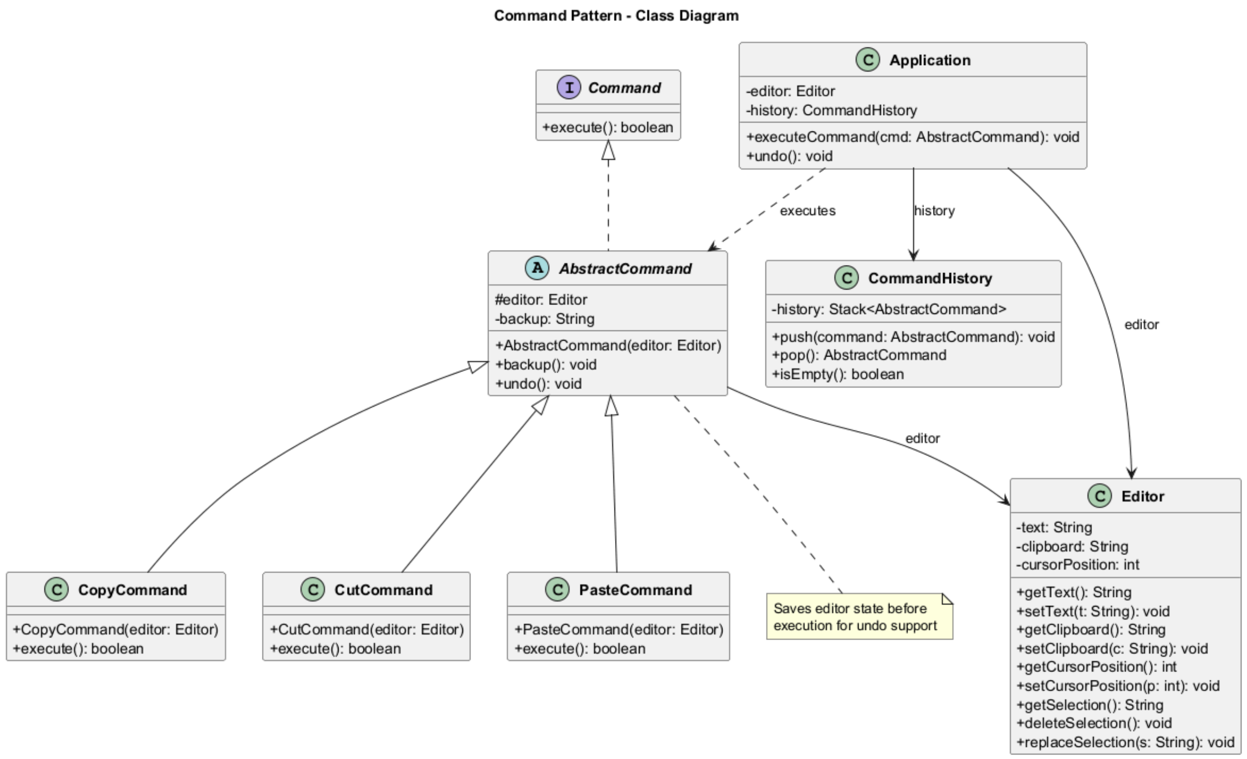

Figure 4: center

Command: Structure Diagram

Command: Java Pseudocode

```java
// Command interface
interface Command {
    boolean execute();
}
```

Command: Java Pseudocode (cont.)

```java
// Receiver: the actual text editor
class Editor {
    private String text = "";
    private String clipboard = "";
    private int cursorPosition = 0;

    public String getText() { return text; }
    public void setText(String t) { this.text = t; }
    public String getClipboard() { return clipboard; }
    public void setClipboard(String c) {
        clipboard = c;
    }
    public int getCursorPosition() {
        return cursorPosition;
    }
    public void setCursorPosition(int p) {
        cursorPosition = p;
    }

    public String getSelection() {
        // Simplified: returns part of text
        return text.substring(
            Math.min(cursorPosition, text.length()));
    }

    public void deleteSelection() {
        text = text.substring(0, cursorPosition);
    }

    public void replaceSelection(String s) {
        text = text.substring(0, cursorPosition) + s;
    }
}
```

Command: Java Pseudocode (cont.)

```
// Abstract command with undo support
abstract class AbstractCommand implements Command {
    protected Editor editor;
    private String backup;

    public AbstractCommand(Editor editor) {
        this.editor = editor;
    }

    // Save editor state before execution
    public void backup() {
        this.backup = editor.getText();
    }

    // Restore editor state
    public void undo() {
        editor.setText(backup);
    }
}
```

Command: Java Pseudocode (cont.)

```
// Concrete Command: Copy
class CopyCommand extends AbstractCommand {
    public CopyCommand(Editor editor) {
        super(editor);
    }

    @Override
    public boolean execute() {
        editor.setClipboard(editor.getSelection());
        // Copy does not change state,
        // so no undo needed
        return false;
    }
}
```

Command: Java Pseudocode (cont.)

```
// Concrete Command: Cut
class CutCommand extends AbstractCommand {
    public CutCommand(Editor editor) {
        super(editor);
    }

    @Override
    public boolean execute() {
        backup();
        editor.setClipboard(editor.getSelection());
        editor.deleteSelection();
        return true;
    }
}
```

Command: Java Pseudocode (cont.)

```
// Concrete Command: Paste
class PasteCommand extends AbstractCommand {
    public PasteCommand(Editor editor) {
        super(editor);
    }

    @Override
    public boolean execute() {
        backup();
        editor.replaceSelection(
            editor.getClipboard());
        return true;
    }
}
```

Command: Java Pseudocode (cont.)

```
import java.util.Stack;

// Command History for undo support
class CommandHistory {
    private Stack<AbstractCommand> history
        = new Stack<>();

    public void push(AbstractCommand command) {
        history.push(command);
    }

    public AbstractCommand pop() {
        return history.pop();
    }
    public boolean isEmpty() {
        return history.isEmpty();
    }
}
```

Command: Java Pseudocode (cont.)

```
// Invoker: Application
class Application {
    private Editor editor = new Editor();
    private CommandHistory history
        = new CommandHistory();

    public void executeCommand(
            AbstractCommand cmd) {
        if (cmd.execute()) {
            // Only save commands that modify state
            history.push(cmd);
        }
    }

    public void undo() {
        if (!history.isEmpty()) {
            AbstractCommand cmd = history.pop();
            cmd.undo();
        }
    }

    public static void main(String[] args) {
        Application app = new Application();
        app.editor.setText("Hello, World!");
        app.editor.setCursorPosition(5);

        // Execute cut (modifies state -> saved)
        app.executeCommand(
            new CutCommand(app.editor));
        System.out.println("After cut: "
            + app.editor.getText());

        // Undo
        app.undo();
        System.out.println("After undo: "
            + app.editor.getText());

        // Expected Output:
        // After cut: Hello
        // After undo: Hello, World!
    }
}
```

Command: Sequence Diagram

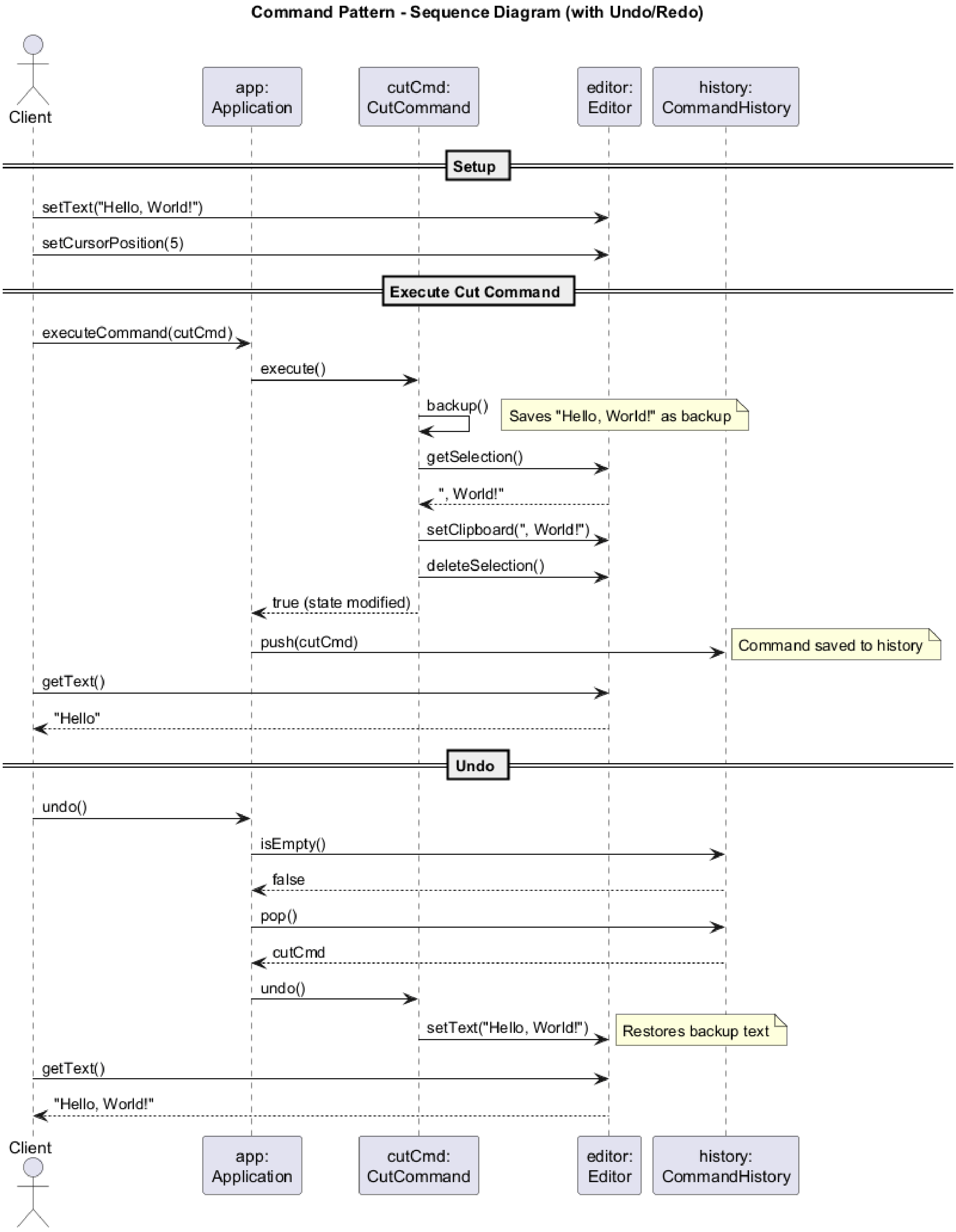

Figure 5: center

Command: Applicability

Use the Command pattern when you want to:

- **Parameterize objects with operations.** The Command pattern can turn a specific method call into a stand-alone object. You can pass commands as method arguments, store them inside other objects, switch linked commands at runtime, etc.
- **Queue operations, schedule their execution, or execute them remotely.** You can serialize commands, send them over the network, and execute them on a different machine.
- **Implement reversible operations.** The most common use. To implement undo, you need to maintain a history of commands with the ability to restore the previous state.

Command: How to Implement

1. Declare the **command interface** with a single execution method.
2. Start extracting requests into **concrete command classes** that implement the command interface. Each class must have fields for storing the request arguments and a reference to the receiver object.
3. Identify classes that will act as **senders** (invokers). Add fields for storing commands. Senders should communicate with commands only via the command interface.
4. Change the senders so they **execute the command** instead of sending a request directly to the receiver.
5. The **client** should initialize objects in the following order: receivers, commands (with receiver references), senders (with command references).

Command: Pros and Cons

Pros:

- **Single Responsibility Principle:** Decouples classes that invoke operations from classes that perform them
- **Open/Closed Principle:** New commands can be introduced without modifying existing client code
- Implement **undo/redo** functionality
- Implement **deferred execution** of operations
- Assemble a set of simple commands into a **complex** one (macro commands)

Cons:

- The code may become **more complicated** since you are introducing a whole new layer between senders and receivers

---

Command: Relations with Other Patterns

- **Chain of Responsibility, Command, Mediator, Observer** handle sender-receiver connections differently. Command establishes unidirectional connections.
- **Command + Memento:** Use Memento to save command state before execution for undo support.
- **Command vs. Strategy:** Both parameterize objects, but Command converts operations to objects (for deferred execution, queueing, history), while Strategy describes different ways of doing the same thing.
- **Prototype** helps when you need to save copies of commands into history.
- **Visitor** can be treated as a powerful version of Command, capable of executing operations on objects of different classes.

Module C: Takeaway

The **Command** pattern encapsulates a request as an object, allowing you to parameterize clients, queue requests, log them, and support undoable operations. It decouples the object that invokes the operation from the one that performs it.

Module D: Iterator Pattern

Module D: Outline

- Intent
- Problem
- Solution
- Real-World Analogy
- Structure
- Java Pseudocode Example
- Applicability
- How to Implement
- Pros and Cons
- Relations with Other Patterns

Iterator: Intent

Iterator is a behavioral design pattern that lets you traverse elements of a collection without exposing its underlying representation (list, stack, tree, etc.).

Source: refactoring.guru[8]

Iterator: Problem

Collections are one of the most used data types in programming. A collection is just a container for a group of objects.

Most collections store their elements in simple lists. However, some of them are based on stacks, trees, graphs, and other complex data structures.

No matter how a collection is structured, it must provide some way of **accessing its elements** so that other code can use these elements. There should be a way to go through each element without accessing the same elements over and over.

Iterator: Problem (cont.)

This may sound like an easy job if your collection is based on a list. You just loop over all of the elements. But how do you **sequentially traverse** elements of a complex data structure, such as a tree?

For example, one day you might need **depth-first** traversal of a tree. The next day you might need **breadth-first**. And the following week, you might need **random access**.

Adding more and more traversal algorithms to the collection gradually **blurs its primary responsibility**: efficient data storage.

[8] https://refactoring.guru/design-patterns/iterator

Iterator: Solution

The main idea of the Iterator pattern is to extract the traversal behavior of a collection into a separate object called an **iterator**.

In addition to implementing the algorithm itself, an iterator object encapsulates all of the traversal details, such as the current position and how many elements are left till the end.

Several iterators can go through the **same collection** at the same time, independently of each other.

Iterator: Solution (cont.)

Usually, iterators provide one primary method for fetching elements of the collection. The client can keep running this method until it returns nothing, which means the iterator has traversed all of the elements.

All iterators must implement the **same interface**. This makes the client code compatible with any collection type or any traversal algorithm as long as there is a proper iterator.

If you need a special way to traverse a collection, you just create a **new iterator class**, without having to change the collection or the client.

Iterator: Real-World Analogy

You plan to visit Rome for a few days and visit all of its sights and attractions. But once there, you could waste a lot of time walking in circles, unable to find even the Colosseum.

On the other hand, you could buy a **virtual guide app** for your smartphone and use it for navigation. Or you could hire a **local guide** who knows the city.

All three options – random walk, smartphone navigator, and human guide – act as **iterators** over the vast collection of sights and attractions in Rome.

Iterator: Structure

The structure consists of the following participants:

1. **Iterator Interface**: Declares the operations required for traversing a collection: fetching the next element, retrieving the current position, restarting iteration, etc.
2. **Concrete Iterators**: Implement specific algorithms for traversing a collection. The iterator object should track the traversal progress on its own. This allows several iterators to traverse the same collection independently.

Iterator: Structure (cont.)

3. **Collection Interface (Iterable)**: Declares one or more methods for getting iterators compatible with the collection. The return type should be declared as the iterator interface.
4. **Concrete Collections**: Return new instances of a particular concrete iterator class each time the client requests one. The rest of the collection's code should be in the same class.
5. **Client**: Works with both collections and iterators via their interfaces, so it is not coupled to concrete classes, allowing the use of various collections and iterators with the same client code.

Iterator: Structure Diagram

Iterator: Java Pseudocode

```java
// Iterator interface
interface ProfileIterator {
    boolean hasNext();
    Profile getNext();
    void reset();
}
```

Iterator: Java Pseudocode (cont.)

```java
// Profile class
class Profile {
    private String name;
    private String email;

    public Profile(String name, String email) {
        this.name = name;
        this.email = email;
    }

    public String getName() { return name; }
    public String getEmail() { return email; }

    @Override
    public String toString() {
        return name + " <" + email + ">";
    }
}
```

Iterator: Java Pseudocode (cont.)

```java
// Collection interface
interface SocialNetwork {
    ProfileIterator createFriendsIterator(
        String profileId);
    ProfileIterator createCoworkersIterator(
        String profileId);
}
```

Iterator: Java Pseudocode (cont.)

```java
import java.util.List;

// Concrete iterator for Facebook friends
class FacebookIterator implements ProfileIterator {
    private Facebook facebook;
    private String profileId;
    private String type;
```

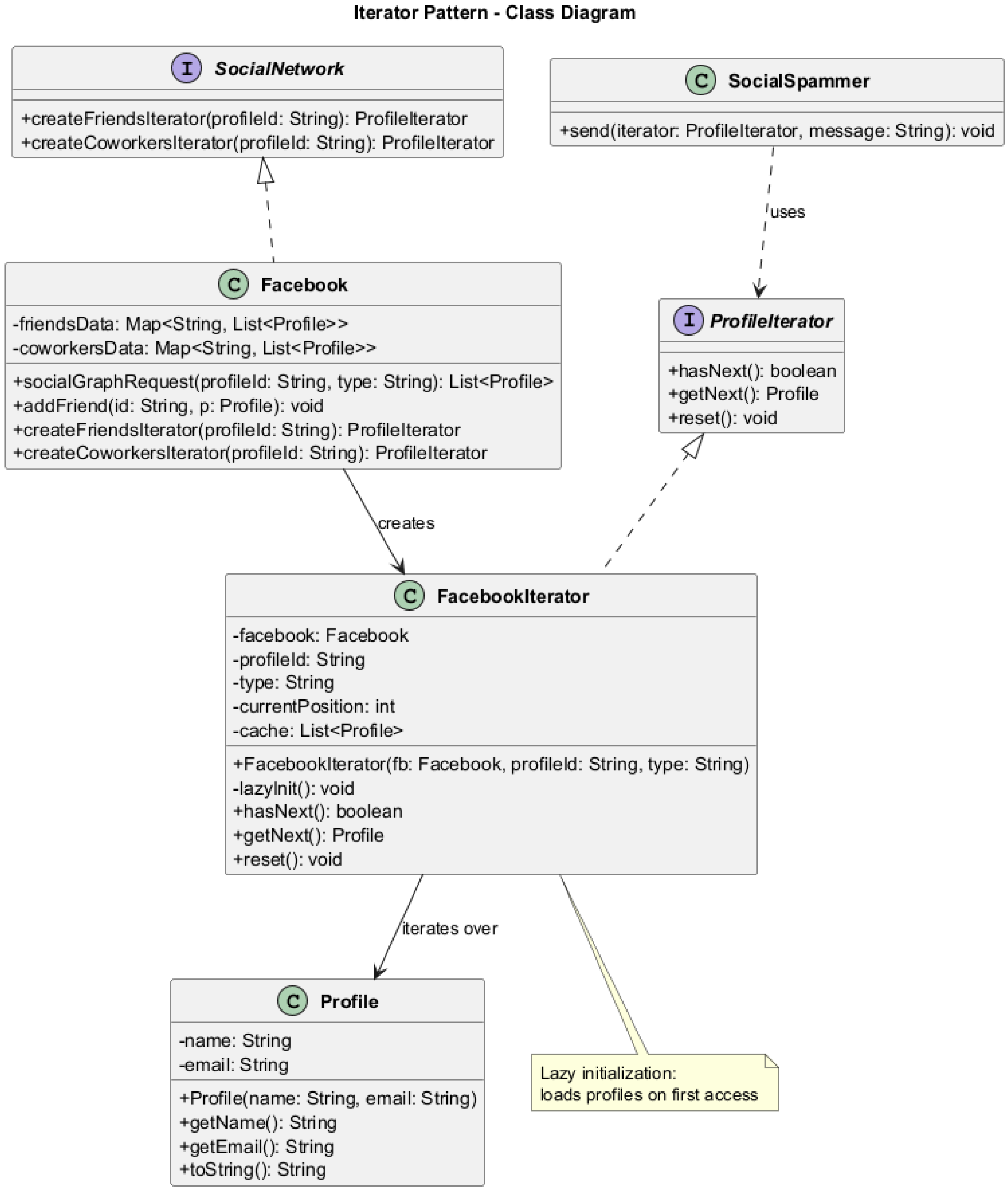
Iterator Pattern - Class Diagram
I SocialNetwork
+createFriendsIterator(profileId: String): ProfileIterator
+createCoworkersIterator(profileId: String): ProfileIterator
C SocialSpammer
+send(iterator: ProfileIterator, message: String): void
uses
C Facebook
-friendsData: Map<String, List<Profile>>
-coworkersData: Map<String, List<Profile>>
+socialGraphRequest(profileId: String, type: String): List<Profile>
+addFriend(id: String, p: Profile): void
+createFriendsIterator(profileId: String): ProfileIterator
+createCoworkersIterator(profileId: String): ProfileIterator
I ProfileIterator
+hasNext(): boolean
+getNext(): Profile
+reset(): void
creates
C FacebookIterator
-facebook: Facebook
-profileId: String
-type: String
-currentPosition: int
-cache: List<Profile>
+FacebookIterator(fb: Facebook, profileId: String, type: String)
-lazyInit(): void
+hasNext(): boolean
+getNext(): Profile
+reset(): void
iterates over
C Profile
-name: String
-email: String
+Profile(name: String, email: String)
+getName(): String
+getEmail(): String
+toString(): String
Lazy initialization:
loads profiles on first access

Figure 6: center

```
    private int currentPosition = 0;
    private List<Profile> cache;

    public FacebookIterator(Facebook fb,
            String profileId, String type) {
        this.facebook = fb;
        this.profileId = profileId;
        this.type = type;
    }

    private void lazyInit() {
        if (cache == null) {
            cache = facebook.socialGraphRequest(
                profileId, type);
        }
    }

    @Override
    public boolean hasNext() {
        lazyInit();
        return currentPosition < cache.size();
    }

    @Override
    public Profile getNext() {
        if (hasNext()) {
            return cache.get(currentPosition++);
        }
        return null;
    }

    @Override
    public void reset() { currentPosition = 0; }
}
```

Iterator: Java Pseudocode (cont.)

```
import java.util.*;

// Concrete collection
class Facebook implements SocialNetwork {
    private Map<String, List<Profile>> friendsData
        = new HashMap<>();
    private Map<String, List<Profile>> coworkersData
        = new HashMap<>();

    // Simulated social graph request
    public List<Profile> socialGraphRequest(
            String profileId, String type) {
        if ("friends".equals(type)) {
            return friendsData.getOrDefault(
                profileId, new ArrayList<>());
        }
        return coworkersData.getOrDefault(
            profileId, new ArrayList<>());
    }
    public void addFriend(String id, Profile p) {
        friendsData.computeIfAbsent(
            id, k -> new ArrayList<>()).add(p);
    }

    @Override
    public ProfileIterator createFriendsIterator(
            String profileId) {
        return new FacebookIterator(
            this, profileId, "friends");
    }

    @Override
    public ProfileIterator createCoworkersIterator(
            String profileId) {
        return new FacebookIterator(
            this, profileId, "coworkers");
    }
}
```

Iterator: Java Pseudocode (cont.)

```
// Client code: SocialSpammer
class SocialSpammer {
    public void send(ProfileIterator iterator,
                     String message) {
        while (iterator.hasNext()) {
            Profile profile = iterator.getNext();
            System.out.println("Sending '"
                + message + "' to "
                + profile.toString());
        }
    }
}

// Usage
public class IteratorDemo {
    public static void main(String[] args) {
        Facebook facebook = new Facebook();
        facebook.addFriend("user1",
            new Profile("Alice", "alice@mail.com"));
        facebook.addFriend("user1",
            new Profile("Bob", "bob@mail.com"));

        ProfileIterator iterator =
            facebook.createFriendsIterator("user1");

        SocialSpammer spammer = new SocialSpammer();
        spammer.send(iterator,
            "Check out our new product!");

        // Expected Output:
        // Sending 'Check out our new product!' to Alice (alice@mail.com)
        // Sending 'Check out our new product!' to Bob (bob@mail.com)
    }
}
```

Iterator: Sequence Diagram

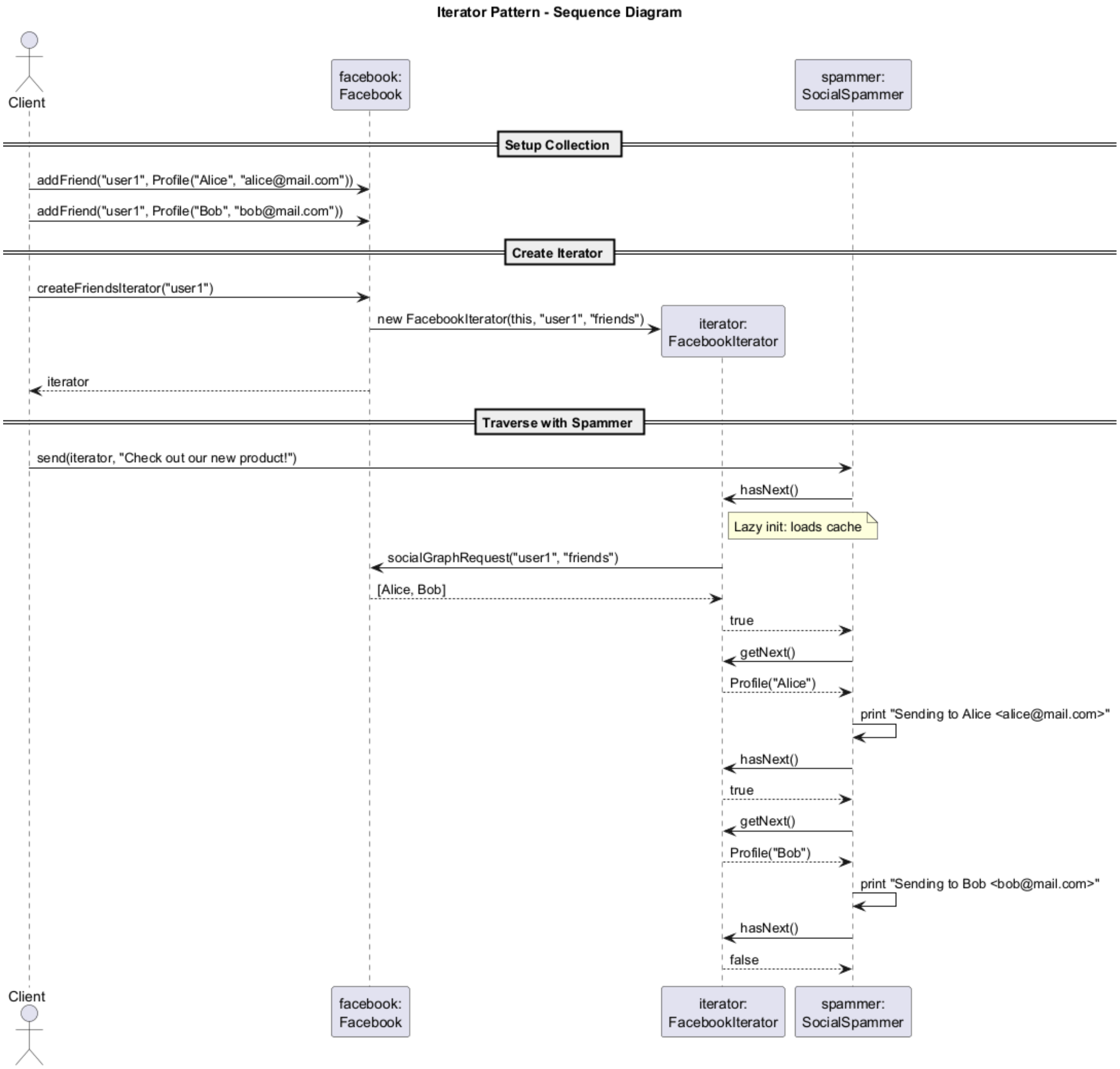
Iterator Pattern - Sequence Diagram
Client
facebook: Facebook
spammer: SocialSpammer
Setup Collection
addFriend("user1", Profile("Alice", "alice@mail.com"))
addFriend("user1", Profile("Bob", "bob@mail.com"))
Create Iterator
createFriendsIterator("user1")
new FacebookIterator(this, "user1", "friends")
iterator: FacebookIterator
iterator
Traverse with Spammer
send(iterator, "Check out our new product!")
hasNext()
Lazy init: loads cache
socialGraphRequest("user1", "friends")
[Alice, Bob]
true
getNext()
Profile("Alice")
print "Sending to Alice <alice@mail.com>"
hasNext()
true
getNext()
Profile("Bob")
print "Sending to Bob <bob@mail.com>"
hasNext()
false
Client
facebook: Facebook
iterator: FacebookIterator
spammer: SocialSpammer

Figure 7: center

Iterator: Applicability

Use the Iterator pattern when:

- Your collection has a complex data structure under the hood, but you want to **hide its complexity** from clients (for convenience or security reasons).
- You want to **reduce duplication** of traversal code across your app.
- You want your code to be able to **traverse different data structures** or when types of these structures are unknown beforehand.

- You need to support **multiple simultaneous traversals** over the same collection.

Iterator: How to Implement

1. Declare the **iterator interface**. At the very least, it must have a method for fetching the next element. But for the sake of convenience, you can add a couple of other methods.
2. Declare the **collection interface** and describe a method for fetching iterators. The return type should be equal to that of the iterator interface.
3. Implement **concrete iterator** classes for the collections that you want to be traversable with iterators.
4. Implement the **collection interface** in your collection classes. The main idea is to provide the client with a shortcut for creating iterators, tailored for a particular collection class.
5. Go over the **client code** to replace all of the collection traversal code with the use of iterators.

Iterator: Pros and Cons

Pros:

- **Single Responsibility Principle:** You can clean up the client code and the collections by extracting bulky traversal algorithms into separate classes
- **Open/Closed Principle:** You can implement new types of collections and iterators and pass them to existing code without breaking anything
- You can iterate over the **same collection in parallel** because each iterator object contains its own iteration state
- For the same reason, you can **delay an iteration** and continue it when needed

Cons:

- Applying the pattern can be **overkill** if your app only works with simple collections
- Using an iterator may be **less efficient** than going through elements of some specialized collections directly

---

Iterator: Relations with Other Patterns

- You can use **Iterators** to traverse **Composite** trees.
- You can use **Factory Method** along with Iterator to let collection subclasses return different types of iterators that are compatible with the collections.
- You can use **Memento** along with Iterator to capture the current iteration state and roll it back if necessary.
- You can use **Visitor** along with Iterator to traverse a complex data structure and execute some operation over its elements even if they all have different classes.

Module D: Takeaway

The **Iterator** pattern provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation. It promotes the Single Responsibility Principle by extracting traversal logic into dedicated iterator objects.

Module E: Mediator Pattern

Module E: Outline

- Intent
- Problem
- Solution
- Real-World Analogy
- Structure
- Java Pseudocode Example
- Applicability
- How to Implement
- Pros and Cons
- Relations with Other Patterns

Mediator: Intent

Mediator is a behavioral design pattern that lets you reduce chaotic dependencies between objects. The pattern restricts direct communications between the objects and forces them to collaborate only via a mediator object.

Also known as: **Intermediary**, **Controller**

Source: refactoring.guru[9]

Mediator: Problem

Say you have a dialog for creating and editing customer profiles. It consists of various form controls such as text fields, checkboxes, buttons, etc.

Some of the form elements may interact with others. For instance, selecting the “I have a dog” checkbox may reveal a hidden text field for entering the dog’s name. Another example is the submit button that has to validate values of all fields before saving the data.

Mediator: Problem (cont.)

By having this logic implemented directly inside the code of the form elements, you make these elements’ classes **much harder to reuse** in other forms of the app. For example, you would not be able to use that checkbox class inside another form because it is coupled to the dog’s text field.

The more elements you add to the form, the more **tangled** the dependencies become. Changing one element may affect many others.

Mediator: Solution

The Mediator pattern suggests that you should cease all direct communication between the components which you want to make independent of each other. Instead, these components must collaborate indirectly by calling a special **mediator object** that redirects the calls to appropriate components.

As a result, the components depend only on a single mediator class instead of being coupled to dozens of their colleagues.

---

[9] https://refactoring.guru/design-patterns/mediator

Mediator: Solution (cont.)

The most significant change happens to the actual form elements. In our dialog example, the **dialog class itself** may act as the mediator. Most likely, the dialog class is already aware of all of its sub-elements, so you would not even need to introduce new dependencies.

Each component must store a reference to the mediator and notify it about events instead of directly calling other components. The mediator then determines which component should handle the event and redirects the call accordingly.

Mediator: Real-World Analogy

Pilots of aircraft that approach or depart the airport control area do not communicate directly with each other. Instead, they speak to an **air traffic controller**, who sits in a tall tower somewhere near the airstrip.

Without the air traffic controller, every pilot would need to be aware of every other plane in the vicinity of the airport, discussing landing priorities with a committee of dozens of other pilots. That would probably skyrocket the airplane crash statistics.

The tower does not need to control the whole flight. It exists only to enforce constraints in the terminal area because the number of involved actors there is too large.

Mediator: Structure

The structure consists of the following participants:

1. **Components**: Various classes that contain some business logic. Each component has a reference to a mediator, declared with the type of the mediator interface. The component is not aware of the actual class of the mediator.
2. **Mediator Interface**: Declares methods of communication with components, which usually include just a single notification method.

Mediator: Structure (cont.)

3. **Concrete Mediators**: Encapsulate the relations between various components. Concrete mediators often keep references to all components they manage and sometimes even manage their lifecycle.
4. **Communication Rule**: Components must not be aware of other components. If something important happens within or to a component, it must only notify the mediator. When the mediator receives the notification, it can easily identify the sender, which might be just enough to decide what component should be triggered in return.

Mediator: Structure Diagram

Mediator: Java Pseudocode

```java
// Mediator interface
interface Mediator {
    void notify(Component sender, String event);
}
```

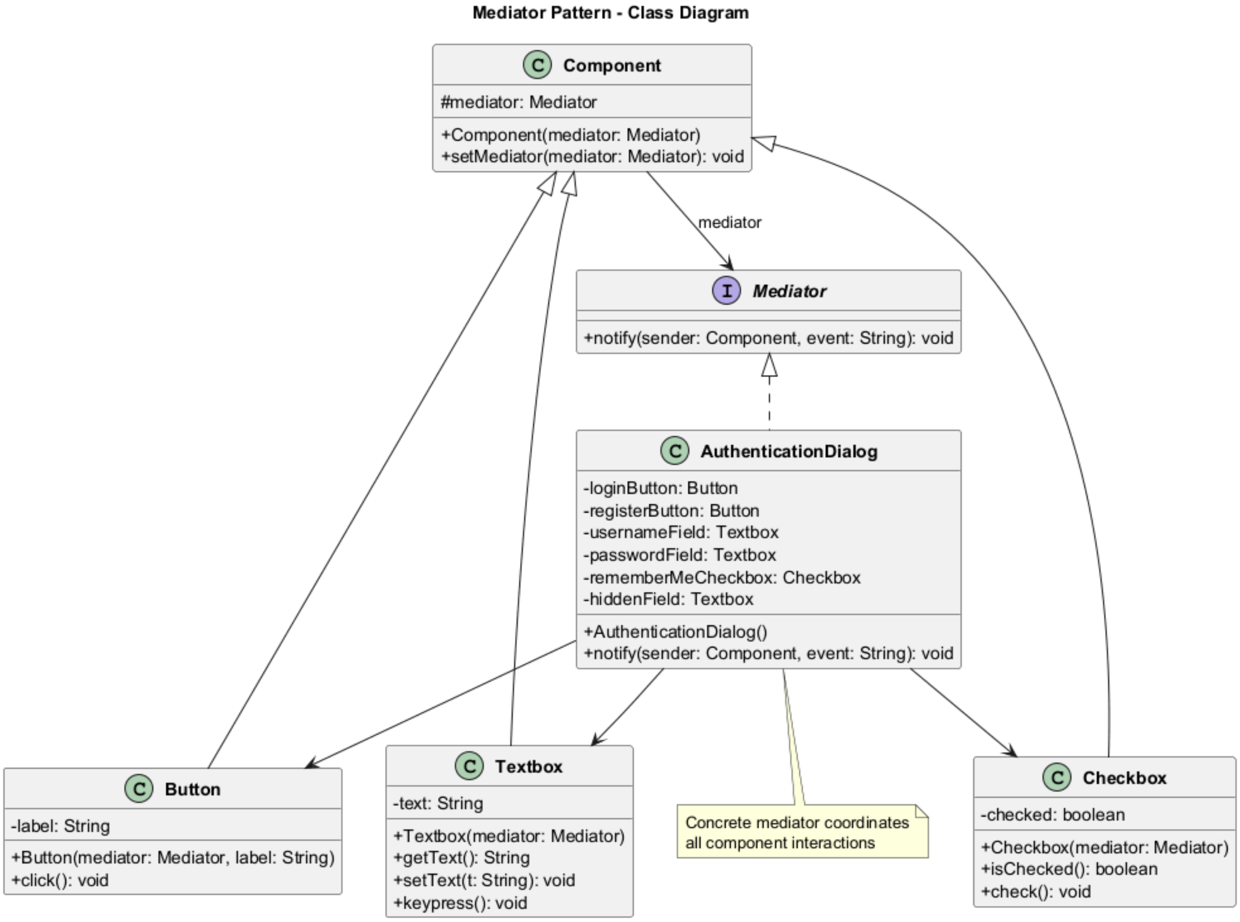
Mediator Pattern - Class Diagram
C Component
#mediator: Mediator
+Component(mediator: Mediator)
+setMediator(mediator: Mediator): void
mediator
I Mediator
+notify(sender: Component, event: String): void
C AuthenticationDialog
-loginButton: Button
-registerButton: Button
-usernameField: Textbox
-passwordField: Textbox
-rememberMeCheckbox: Checkbox
-hiddenField: Textbox
+AuthenticationDialog()
+notify(sender: Component, event: String): void
C Button
-label: String
+Button(mediator: Mediator, label: String)
+click(): void
C Textbox
-text: String
+Textbox(mediator: Mediator)
+getText(): String
+setText(t: String): void
+keypress(): void
Concrete mediator coordinates
all component interactions
C Checkbox
-checked: boolean
+Checkbox(mediator: Mediator)
+isChecked(): boolean
+check(): void

Figure 8: center

Mediator: Java Pseudocode (cont.)

```java
// Base component
class Component {
    protected Mediator mediator;

    public Component(Mediator mediator) {
        this.mediator = mediator;
    }

    public void setMediator(Mediator mediator) {
        this.mediator = mediator;
    }
}
```

Mediator: Java Pseudocode (cont.)

```java
// Concrete component: Button
class Button extends Component {
    private String label;

    public Button(Mediator mediator, String label) {
        super(mediator);
        this.label = label;
    }

    public void click() {
        System.out.println("Button '" + label
            + "' clicked.");
        mediator.notify(this, "click");
    }
}

// Concrete component: Textbox
class Textbox extends Component {
    private String text = "";

    public Textbox(Mediator mediator) {
        super(mediator);
    }

    public String getText() { return text; }
    public void setText(String t) { this.text = t; }

    public void keypress() {
        mediator.notify(this, "keypress");
    }
}
```

Mediator: Java Pseudocode (cont.)

```java
// Concrete component: Checkbox
class Checkbox extends Component {
    private boolean checked = false;
    public Checkbox(Mediator mediator) {
        super(mediator);
    }

    public boolean isChecked() { return checked; }

    public void check() {
        checked = !checked;
        System.out.println("Checkbox is now "
            + (checked ? "checked" : "unchecked"));
        mediator.notify(this, "check");
    }
}
```

Mediator: Java Pseudocode (cont.)

```java
// Concrete mediator: AuthenticationDialog
class AuthenticationDialog implements Mediator {
    private Button loginButton;
    private Button registerButton;
    private Textbox usernameField;
    private Textbox passwordField;
    private Checkbox rememberMeCheckbox;
    private Textbox hiddenField;

    public AuthenticationDialog() {
        loginButton =
            new Button(this, "Login");
        registerButton =
            new Button(this, "Register");
        usernameField = new Textbox(this);
        passwordField = new Textbox(this);
        rememberMeCheckbox = new Checkbox(this);
        hiddenField = new Textbox(this);
    }

    @Override
    public void notify(Component sender,
                       String event) {
        if (sender == loginButton
                && "click".equals(event)) {
            System.out.println(
                "Mediator: Validating"
                + " credentials...");
        } else if (sender == rememberMeCheckbox
                && "check".equals(event)) {
            boolean show =
                rememberMeCheckbox.isChecked();
            System.out.println("Mediator: "
                + (show ? "Showing" : "Hiding")
                + " hidden field.");
        }
    }
}
```

Mediator: Java Pseudocode (cont.)

```
// Client code
public class MediatorDemo {
    public static void main(String[] args) {
        AuthenticationDialog dialog =
            new AuthenticationDialog();

        // Simulate user interactions
        dialog.rememberMeCheckbox.check();
        dialog.loginButton.click();
    }
}
```

Output:

```
Checkbox is now checked
Mediator: Showing hidden field.
Button 'Login' clicked.
Mediator: Validating credentials...
```

Mediator: Sequence Diagram

Mediator: Applicability

Use the Mediator pattern when:

- It is hard to change some of the classes because they are **tightly coupled** to a bunch of other classes.
- You cannot **reuse a component** in a different program because it is too dependent on other components.
- You find yourself creating tons of **component subclasses** just to reuse some basic behavior in various contexts.

Mediator: How to Implement

1. Identify a group of **tightly coupled classes** which would benefit from being more independent.
2. Declare the **mediator interface** and describe the desired communication protocol between mediators and various components.
3. Implement the **concrete mediator** class. Consider storing references to all components inside the mediator.
4. You can go even further and make the mediator responsible for the **creation and destruction** of component objects.
5. Components should store a reference to the mediator object. The connection is usually established in the **component's constructor**.
6. Change the components' code so that they **call the mediator's notification** method instead of methods on other components.

Mediator: Pros and Cons

Pros:

- **Single Responsibility Principle:** You can extract the communications between various components into a single place, making it easier to comprehend and maintain

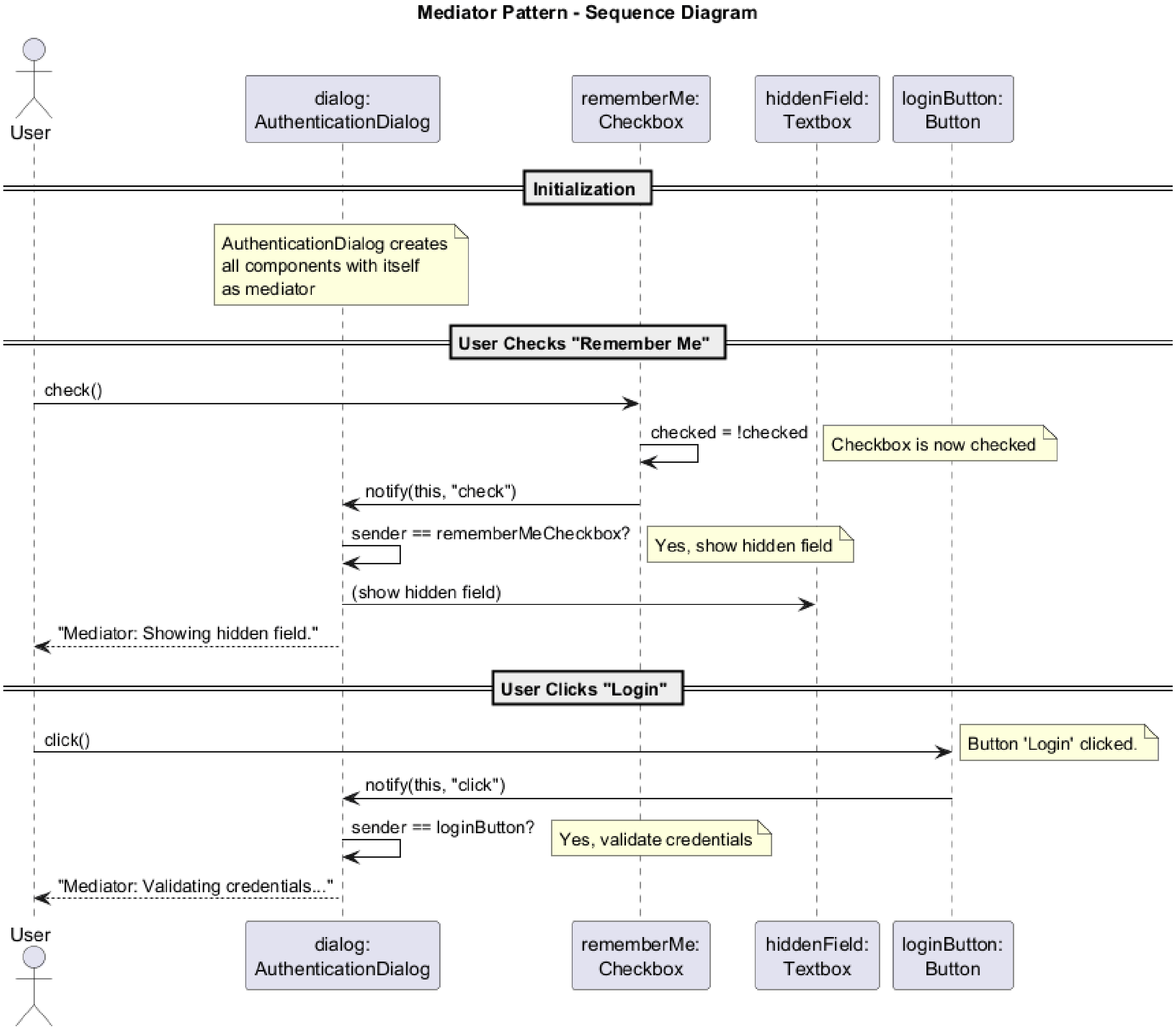

Figure 9: center

- **Open/Closed Principle:** You can introduce new mediators without having to change the actual components
- You can **reduce coupling** between various components of a program
- You can **reuse individual components** more easily

Cons:

- Over time a mediator can evolve into a **God Object** – an object that knows too much or does too much

---

Mediator: Relations with Other Patterns

- **Chain of Responsibility, Command, Mediator, and Observer** address various ways of connecting senders and receivers. Mediator eliminates direct connections between senders and receivers, forcing them to communicate indirectly.
- **Facade vs. Mediator:** Facade defines a simplified interface to a subsystem; it does not add new functionality and subsystem objects are not aware of the facade. Mediator centralizes communication between components.
- **Mediator and Observer:** The distinction is often subtle. Mediator eliminates mutual dependencies via a central object. Observer allows establishing dynamic one-way subscription connections. A Mediator can be implemented using Observer, where the mediator acts as publisher and components are subscribers.

Module E: Takeaway

The **Mediator** pattern defines an object that encapsulates how a set of objects interact. It promotes loose coupling by keeping objects from referring to each other explicitly and lets you vary their interaction independently.

Module F: Memento Pattern

Module F: Outline

- Intent
- Problem
- Solution
- Real-World Analogy
- Structure
- Java Pseudocode Example
- Applicability
- How to Implement
- Pros and Cons
- Relations with Other Patterns

Memento: Intent

Memento is a behavioral design pattern that lets you save and restore the previous state of an object without revealing the details of its implementation.

Also known as: **Snapshot**

Source: refactoring.guru[10]

Memento: Problem

Imagine that you are creating a text editor app. In addition to simple text editing, your editor can format text, insert inline images, etc.

At some point, you decided to let users **undo** any operations carried out on the text. This feature has become so common that people expect to have it in every app.

For the implementation, you chose to take the direct approach: before performing any operation, the app records the **state of all objects** and saves it in some storage. Later, when a user decides to undo an action, the app fetches the most recent snapshot and uses it to restore the state.

Memento: Problem (cont.)

How exactly would you produce the snapshots of the object's state? You would probably need to go over all the fields in an object and copy their values into storage. However, this would only work if the object had quite **relaxed access restrictions** to its contents. Unfortunately, most real objects will not let others peek inside them that easily, hiding all significant data in **private fields**.

Attempting to directly copy private fields either breaks encapsulation or couples snapshot logic to the object's internal structure, making the code fragile.

Memento: Solution

The Memento pattern delegates creating the state snapshots to the actual owner of that state, the **originator** object. Hence, instead of other objects trying to copy the editor's state from the "outside," the editor class itself can make the snapshot since it has full access to its own state.

The pattern suggests storing the copy of the object's state in a special object called **memento**. The contents of the memento are not accessible to any other object except the one that produced it.

Memento: Solution (cont.)

Other objects must communicate with mementos using a **limited interface** which may allow fetching the snapshot's metadata (creation time, name of operation, etc.), but not the original object's state contained in the snapshot.

A **caretaker** object (e.g., the command history) knows "when" and "why" to capture the originator's state, as well as when the state should be restored. The caretaker can store a stack of mementos, and when it is time to travel back in time, it fetches the topmost memento and passes it to the originator's restore method.

Memento: Real-World Analogy

Think of **saving a game**. The game (originator) has a complex internal state: current level, character positions, inventory, health points, etc.

Before a dangerous boss fight, you **save the game** (create a memento). The save file stores all the relevant game state. If you fail, you can **load the save** (restore from memento) and try again.

The save file is the memento. The player (or the game menu) acts as the caretaker.

[10] https://refactoring.guru/design-patterns/memento

Memento: Structure

The classic implementation uses nested classes:

1. **Originator**: Can produce snapshots of its own state, as well as restore its state from snapshots when needed.
2. **Memento**: A value object that acts as a snapshot of the originator's state. It is a common practice to make the memento immutable and pass the data to it only once, via the constructor.

Memento: Structure (cont.)

3. **Caretaker**: Knows not only "when" and "why" to capture the originator's state, but also when the state should be restored. A caretaker can keep track of the originator's history by storing a stack of mementos. When the originator has to travel back in time, the caretaker fetches the topmost memento from the stack and passes it to the originator's restoration method.
4. In the nested class implementation, the **Memento class is nested inside the Originator**. This lets the originator access the fields and methods of the memento, even though they are declared private.

Memento: Structure Diagram

Memento: Java Pseudocode

```java
// Memento: stores editor state
class EditorMemento {
    private final String text;
    private final int cursorPosition;
    private final int scrollPosition;
    private final Date timestamp;

    public EditorMemento(String text,
            int cursorPosition,
            int scrollPosition) {
        this.text = text;
        this.cursorPosition = cursorPosition;
        this.scrollPosition = scrollPosition;
        this.timestamp = new Date();
    }

    // Only originator can access these
    String getText() { return text; }
    int getCursorPosition() {
        return cursorPosition;
    }
    int getScrollPosition() {
        return scrollPosition;
    }

    // Public metadata
    public Date getTimestamp() { return timestamp; }
    public String getName() {
```

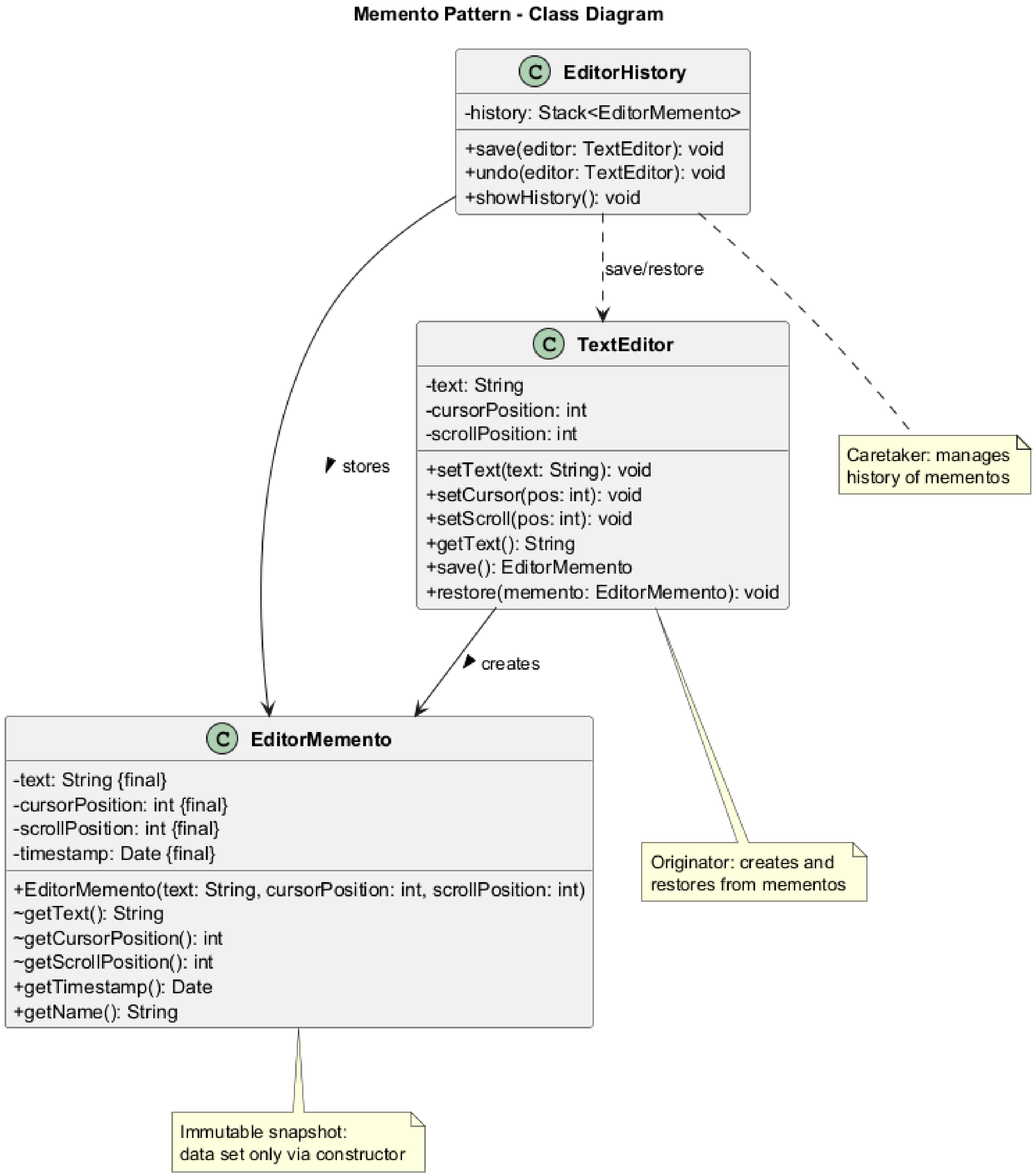
Memento Pattern - Class Diagram
C
EditorHistory
-history: Stack<EditorMemento>
+save(editor: TextEditor): void
+undo(editor: TextEditor): void
+showHistory(): void
save/restore
stores
C
TextEditor
-text: String
-cursorPosition: int
-scrollPosition: int
+setText(text: String): void
+setCursor(pos: int): void
+setScroll(pos: int): void
+getText(): String
+save(): EditorMemento
+restore(memento: EditorMemento): void
Caretaker: manages
history of mementos
creates
C
EditorMemento
-text: String {final}
-cursorPosition: int {final}
-scrollPosition: int {final}
-timestamp: Date {final}
+EditorMemento(text: String, cursorPosition: int, scrollPosition: int)
~getText(): String
~getCursorPosition(): int
~getScrollPosition(): int
+getTimestamp(): Date
+getName(): String
Originator: creates and
restores from mementos
Immutable snapshot:
data set only via constructor

Figure 10: center

```
        return timestamp + " / "
            + text.substring(0,
                Math.min(10, text.length()))
            + "...";
    }
}
```

Memento: Java Pseudocode (cont.)

```
// Originator: the text editor
class TextEditor {
    private String text;
    private int cursorPosition;
    private int scrollPosition;

    public void setText(String text) {
        this.text = text;
    }

    public void setCursor(int pos) {
        this.cursorPosition = pos;
    }

    public void setScroll(int pos) {
        this.scrollPosition = pos;
    }

    public String getText() { return text; }

    // Create snapshot
    public EditorMemento save() {
        return new EditorMemento(
            text, cursorPosition, scrollPosition);
    }

    // Restore from snapshot
    public void restore(EditorMemento memento) {
        this.text = memento.getText();
        this.cursorPosition =
            memento.getCursorPosition();
        this.scrollPosition =
            memento.getScrollPosition();
    }
}
```

Memento: Java Pseudocode (cont.)

```
import java.util.Stack;

// Caretaker: manages history
class EditorHistory {
    private Stack<EditorMemento> history
        = new Stack<>();
    public void save(TextEditor editor) {
        history.push(editor.save());
    }

    public void undo(TextEditor editor) {
        if (!history.isEmpty()) {
            EditorMemento memento = history.pop();
            editor.restore(memento);
            System.out.println("Restored to: "
                + memento.getName());
        }
    }

    public void showHistory() {
        System.out.println("History ("
            + history.size() + " states):");
        for (EditorMemento m : history) {
            System.out.println(
                "  - " + m.getName());
        }
    }
}
```

Memento: Java Pseudocode (cont.)

```
// Client code
public class MementoDemo {
    public static void main(String[] args) {
        TextEditor editor = new TextEditor();
        EditorHistory history = new EditorHistory();

        editor.setText("Hello");
        editor.setCursor(5);
        history.save(editor);

        editor.setText("Hello, World!");
        editor.setCursor(13);
        history.save(editor);

        editor.setText("Hello, World! Bye!");
        editor.setCursor(18);

        System.out.println("Current: "
            + editor.getText());
        history.showHistory();

        // Undo
        history.undo(editor);
        System.out.println("After undo: "
            + editor.getText());

        history.undo(editor);
        System.out.println("After 2nd undo: "
            + editor.getText());

        // Expected Output:
```

```
        // Current: Hello, World! Bye!
        // History (2 states):
        //   - 2026-..._Hello
        //   - 2026-..._Hello, World!
        // Restored to: 2026-..._Hello, World!
        // After undo: Hello, World!
        // Restored to: 2026-..._Hello
        // After 2nd undo: Hello
    }
}
```

Memento: Sequence Diagram

Memento: Applicability

Use the Memento pattern when:

- You want to produce **snapshots of the object's state** to be able to restore a previous state.
- Direct access to the object's fields/getters/setters **violates its encapsulation**. The Memento makes the object itself responsible for creating a snapshot of its state.
- You need to implement **undo/redo**, **transaction rollbacks**, or **checkpointing** functionality.

Memento: How to Implement

1. Determine what class will play the role of the **originator**.
2. Create the **memento class**. One by one, declare a set of fields that mirror the fields declared inside the originator class.
3. Make the memento class **immutable**. A memento should accept the data just once, via the constructor. The class should have no setters.
4. If your language supports **nested classes**, nest the memento inside the originator. If not, extract a blank interface from the memento class and make everyone else use it.
5. Add a method for producing mementos to the **originator** class.
6. Add a method for restoring the originator's state to the **originator** class.
7. The **caretaker** should know when to request new mementos from the originator, how to store them, and when to restore the originator with a particular memento.

Memento: Pros and Cons

Pros:

- You can produce snapshots of the object's state **without violating its encapsulation**
- You can simplify the originator's code by letting the caretaker maintain the **history of the originator's state**

Cons:

- The app might consume **lots of RAM** if clients create mementos too often
- **Caretakers** should track the originator's lifecycle to be able to destroy obsolete mementos
- Most dynamic programming languages (PHP, Python, JavaScript) cannot guarantee that the state within the memento **stays untouched**

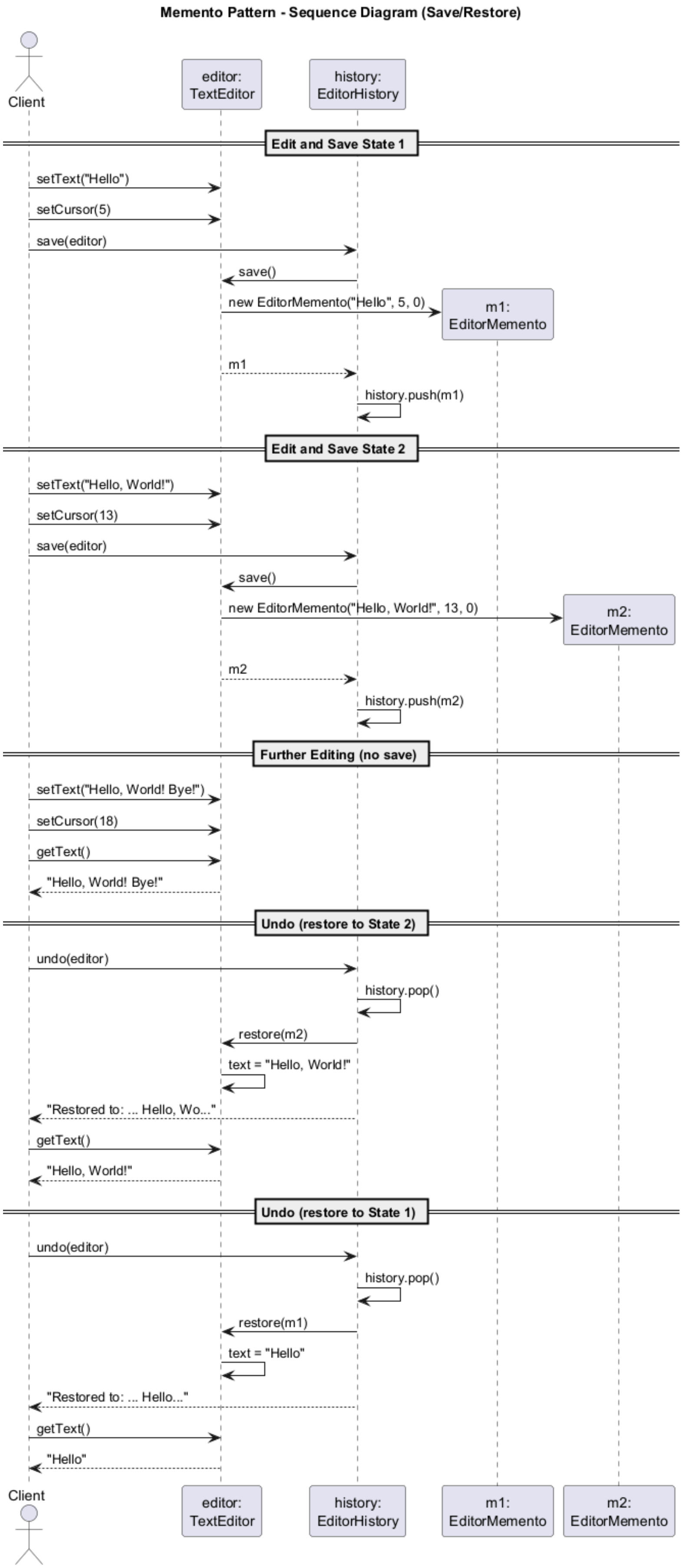

Figure 11: center

Memento: Relations with Other Patterns

- You can use **Command and Memento** together for implementing undo. Commands are responsible for performing various operations over a target object, while mementos save the state of that object just before a command gets executed.
- You can use **Memento along with Iterator** to capture the current iteration state and roll it back if necessary.
- Sometimes **Prototype** can be a simpler alternative to Memento. This works if the object whose state you want to store in the history is fairly straightforward and does not have links to external resources.

Module F: Takeaway

The **Memento** pattern captures and externalizes an object's internal state so that the object can be restored to that state later, all without violating encapsulation.

Module G: Observer Pattern

Module G: Outline

- Intent
- Problem
- Solution
- Real-World Analogy
- Structure
- Java Pseudocode Example
- Applicability
- How to Implement
- Pros and Cons
- Relations with Other Patterns

Observer: Intent

Observer is a behavioral design pattern that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they are observing.

Also known as: **Event-Subscriber**, **Listener**

Source: refactoring.guru[11]

Observer: Problem

Imagine that you have two types of objects: a `Customer` and a `Store`. The customer is very interested in a particular brand of product which should become available in the store very soon.

The customer could visit the store every day and check product availability. But while the product is still en route, most of these trips would be **pointless**.

On the other hand, the store could send tons of emails to all customers each time a new product becomes available. This would **save some customers** from endless trips but would **upset other customers** who are not interested in new products.

[11] https://refactoring.guru/design-patterns/observer

Observer: Solution

The object that has some interesting state is often called **subject** (publisher). All other objects that want to track changes to the publisher's state are called **subscribers**.

The Observer pattern suggests that you add a **subscription mechanism** to the publisher class so individual objects can subscribe to or unsubscribe from a stream of events.

Observer: Solution (cont.)

In reality, this mechanism consists of:

1. An **array field** for storing a list of references to subscriber objects
2. Several **public methods** which allow adding subscribers to and removing them from that list
3. A **notification method** that goes over the list of subscribers and calls their specific notification method

Now whenever an important event happens to the publisher, it goes over its subscribers and calls the specific notification method on their objects.

Observer: Solution (cont.)

It is crucial that all subscribers implement the **same interface** and that the publisher communicates with them only via that interface. This interface should declare the notification method along with a set of parameters that the publisher can use to pass some contextual data along with the notification.

If your app has several different types of publishers, you can make all of them follow the same interface, allowing subscribers to observe all of them with a single subscription.

Observer: Real-World Analogy

If you subscribe to a newspaper or magazine, you no longer need to go to the store to check if the next issue is available. Instead, the **publisher** sends new issues directly to your mailbox right after publication.

The publisher maintains a list of subscribers and knows which magazines they are interested in. Subscribers can leave the list at any time when they wish to stop the publisher from sending new magazine issues to them.

Observer: Structure

The structure consists of the following participants:

1. **Publisher (Subject)**: Issues events of interest to other objects. These events occur when the publisher changes its state or executes some behaviors. Publishers contain a subscription infrastructure that lets new subscribers join and current subscribers leave the list.

2. **Subscriber Interface**: Declares the notification interface. In most cases, it consists of a single `update` method. The method may have several parameters that let the publisher pass some event details along with the update.

Observer: Structure (cont.)

3. **Concrete Subscribers**: Perform some actions in response to notifications issued by the publisher. All of these classes must implement the same interface so the publisher is not coupled to concrete classes.
4. **Client**: Creates publisher and subscriber objects separately and then registers subscribers for publisher updates.

Usually, subscribers need some contextual information to handle the update correctly. For this reason, publishers often pass some context data as arguments of the notification method.

Observer: Structure Diagram

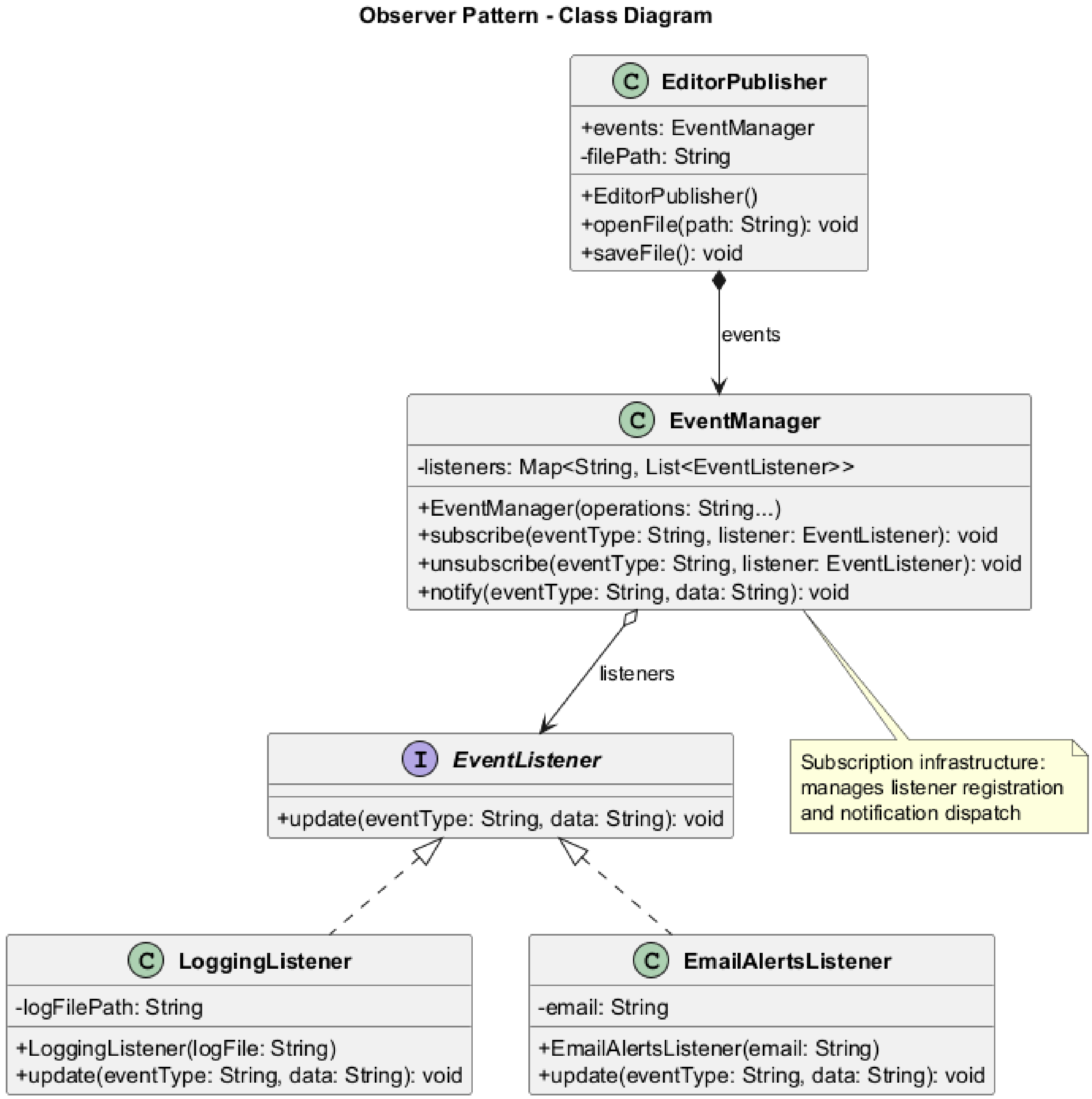
Observer Pattern - Class Diagram
C EditorPublisher
+events: EventManager
-filePath: String
+EditorPublisher()
+openFile(path: String): void
+saveFile(): void
events
C EventManager
-listeners: Map<String, List<EventListener>>
+EventManager(operations: String...)
+subscribe(eventType: String, listener: EventListener): void
+unsubscribe(eventType: String, listener: EventListener): void
+notify(eventType: String, data: String): void
listeners
I EventListener
+update(eventType: String, data: String): void
Subscription infrastructure:
manages listener registration
and notification dispatch
C LoggingListener
-logFilePath: String
+LoggingListener(logFile: String)
+update(eventType: String, data: String): void
C EmailAlertsListener
-email: String
+EmailAlertsListener(email: String)
+update(eventType: String, data: String): void

Figure 12: center

Observer: Java Pseudocode

```
// Subscriber interface
interface EventListener {
    void update(String eventType, String data);
}
```

Observer: Java Pseudocode (cont.)

```
import java.util.*;

// Event manager (subscription infrastructure)
class EventManager {
    private Map<String, List<EventListener>> listeners
        = new HashMap<>();

    public EventManager(String... operations) {
        for (String op : operations) {
            listeners.put(op, new ArrayList<>());
        }
    }

    public void subscribe(String eventType,
                          EventListener listener) {
        listeners.get(eventType).add(listener);
    }

    public void unsubscribe(String eventType,
                            EventListener listener) {
        listeners.get(eventType).remove(listener);
    }

    public void notify(String eventType,
                       String data) {
        for (EventListener listener :
                listeners.get(eventType)) {
            listener.update(eventType, data);
        }
    }
}
```

Observer: Java Pseudocode (cont.)

```
// Publisher: Editor
class EditorPublisher {
    public EventManager events;
    private String filePath;

    public EditorPublisher() {
        this.events = new EventManager(
            "open", "save", "close");
    }
```

```java
    public void openFile(String path) {
        this.filePath = path;
        System.out.println(
            "Editor: opened file " + path);
        events.notify("open", path);
    }

    public void saveFile() {
        System.out.println(
            "Editor: saved file " + filePath);
        events.notify("save", filePath);
    }
}
```

Observer: Java Pseudocode (cont.)

```java
// Concrete subscriber: Logging
class LoggingListener implements EventListener {
    private String logFilePath;

    public LoggingListener(String logFile) {
        this.logFilePath = logFile;
    }

    @Override
    public void update(String eventType,
                       String data) {
        System.out.println("LoggingListener: "
            + "Writing to log " + logFilePath
            + ": Event '" + eventType
            + "' with data '" + data + "'");
    }
}
```

Observer: Java Pseudocode (cont.)

```java
// Concrete subscriber: Email alerts
class EmailAlertsListener
        implements EventListener {
    private String email;

    public EmailAlertsListener(String email) {
        this.email = email;
    }

    @Override
    public void update(String eventType,
                       String data) {
        System.out.println("EmailAlertsListener: "
            + "Sending email to " + email
            + ": Event '" + eventType
            + "' with data '" + data + "'");
    }
}
```

Observer: Java Pseudocode (cont.)

```java
// Client code
public class ObserverDemo {
    public static void main(String[] args) {
        EditorPublisher editor =
            new EditorPublisher();

        LoggingListener logger =
            new LoggingListener(
                "/var/log/app.log");
        EmailAlertsListener emailAlert =
            new EmailAlertsListener(
                "admin@site.com");

        editor.events.subscribe("open", logger);
        editor.events.subscribe("save", emailAlert);
        editor.events.subscribe("save", logger);

        editor.openFile("/home/user/doc.txt");
        editor.saveFile();
    }
}
```

Observer: Java Pseudocode Output

```
Editor: opened file /home/user/doc.txt
LoggingListener: Writing to log /var/log/app.log:
    Event 'open' with data '/home/user/doc.txt'
Editor: saved file /home/user/doc.txt
EmailAlertsListener: Sending email to admin@site.com:
    Event 'save' with data '/home/user/doc.txt'
LoggingListener: Writing to log /var/log/app.log:
    Event 'save' with data '/home/user/doc.txt'
```

Observer: Sequence Diagram

Observer: Applicability

Use the Observer pattern when:

- Changes to the state of one object may require changing **other objects**, and the actual set of objects is **unknown beforehand** or changes dynamically.
- Some objects in your app must observe others, but only for a **limited time** or in **specific cases**.
- You need to **decouple** the publisher from its subscribers so they can be developed, tested, and maintained independently.

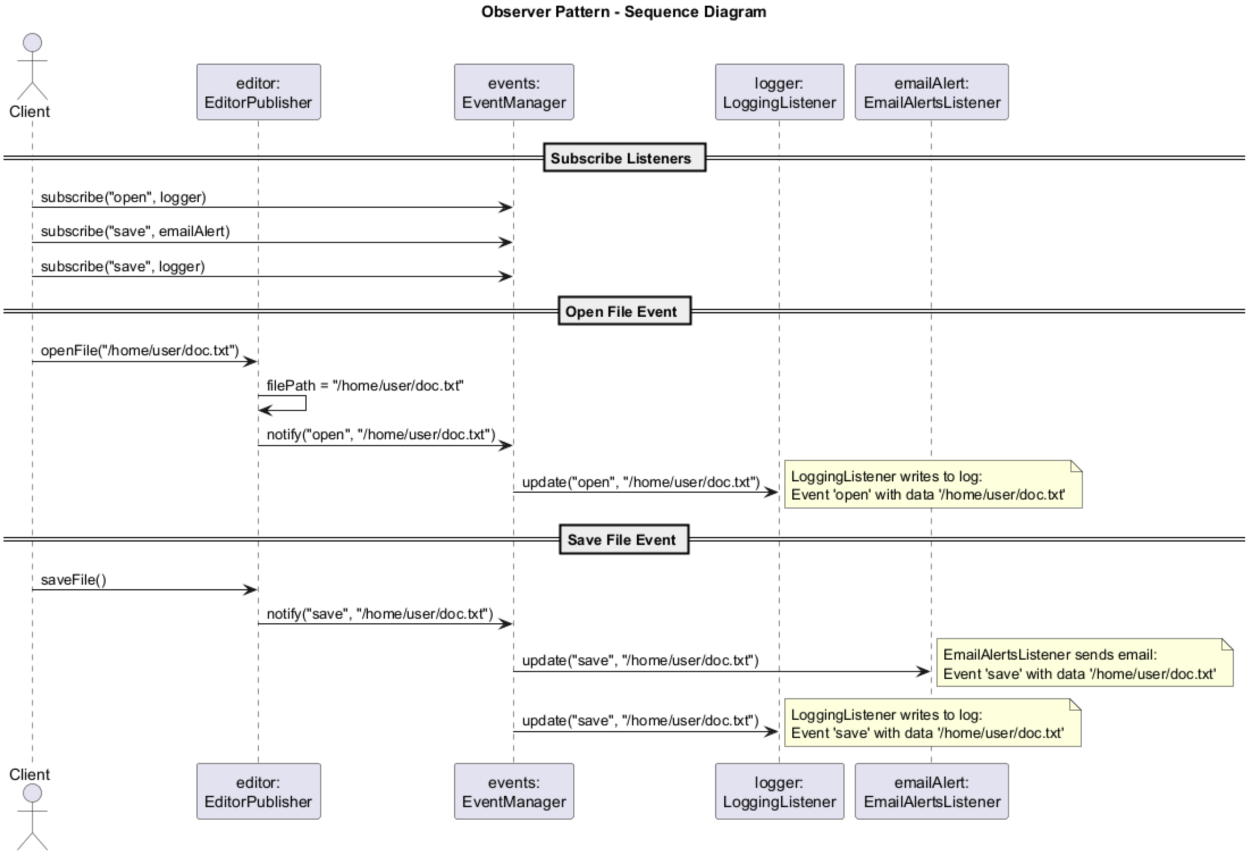
Observer Pattern - Sequence Diagram
Client
editor: EditorPublisher
events: EventManager
logger: LoggingListener
emailAlert: EmailAlertsListener
Subscribe Listeners
subscribe("open", logger)
subscribe("save", emailAlert)
subscribe("save", logger)
Open File Event
openFile("/home/user/doc.txt")
filePath = "/home/user/doc.txt"
notify("open", "/home/user/doc.txt")
update("open", "/home/user/doc.txt")
LoggingListener writes to log:
Event 'open' with data '/home/user/doc.txt'
Save File Event
saveFile()
notify("save", "/home/user/doc.txt")
update("save", "/home/user/doc.txt")
EmailAlertsListener sends email:
Event 'save' with data '/home/user/doc.txt'
update("save", "/home/user/doc.txt")
LoggingListener writes to log:
Event 'save' with data '/home/user/doc.txt'
Client
editor: EditorPublisher
events: EventManager
logger: LoggingListener
emailAlert: EmailAlertsListener

Figure 13: center

Observer: How to Implement

1. Look through your business logic and try to break it down into two parts: the **core functionality** (publisher) independent from other code, and the **rest** that will act as subscriber classes.
2. Declare the **subscriber interface**. At the bare minimum, it should declare a single `update` method.
3. Declare the **publisher interface** and describe a pair of methods for adding and removing a subscriber object from the list.
4. Decide where to put the actual subscription list and the implementation of subscription methods. Usually this code looks the same for all publishers, so create an **abstract class or use composition**.
5. Create **concrete publisher** classes. Each time something important happens inside a publisher, it must notify all its subscribers.
6. Implement the **update notification methods** in concrete subscriber classes.
7. The **client** must create all necessary subscribers and register them with proper publishers.

Observer: Pros and Cons

Pros:

- **Open/Closed Principle:** You can introduce new subscriber classes without having to change the publisher's code (and vice versa if there is a publisher interface)
- You can establish **relations between objects at runtime**
- Publishers are **decoupled** from concrete subscriber implementations

Cons:

- Subscribers are notified in **random order** (no guaranteed ordering)
- If not properly managed, can lead to **memory leaks** (subscribers that forget to unsubscribe)

Observer: Relations with Other Patterns

- **Chain of Responsibility, Command, Mediator, and Observer** are various ways of connecting senders and receivers:
  - **CoR** passes sequentially along a chain
  - **Command** establishes unidirectional connections
  - **Mediator** eliminates direct connections via a central object
  - **Observer** lets receivers dynamically subscribe/unsubscribe
- **Observer vs. Mediator:** Mediator eliminates mutual dependencies; Observer creates one-way dynamic connections. You can implement Mediator using Observer, where the mediator acts as publisher and components as subscribers.

Module G: Takeaway

The **Observer** pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. It is the foundation of event-driven programming.

Module H: State Pattern

Module H: Outline

- Intent

- Problem
- Solution
- Real-World Analogy
- Structure
- Java Pseudocode Example
- Applicability
- How to Implement
- Pros and Cons
- Relations with Other Patterns

State: Intent

State is a behavioral design pattern that lets an object alter its behavior when its internal state changes. It appears as if the object changed its class.

Source: refactoring.guru[12]

State: Problem

The State pattern is closely related to the concept of a **Finite-State Machine**.

The main idea is that, at any given moment, there is a **finite number of states** which a program can be in. Within any unique state, the program behaves differently, and the program can be **switched from one state to another** instantaneously.

However, depending on a current state, the program may or may not switch to certain other states. These switching rules, called **transitions**, are also finite and predetermined.

State: Problem (cont.)

Imagine a `Document` class. A document can be in one of three states: `Draft`, `Moderation`, and `Published`. The `publish` method of the document works a little bit differently in each state:

- In **Draft**, it moves the document to **Moderation**
- In **Moderation**, it makes the document **Published**, but only if the current user is an administrator
- In **Published**, it does **nothing**

State: Problem (cont.)

The biggest weakness of a state machine based on conditionals reveals itself once we start adding more and more states and state-dependent behaviors. Most methods will contain **monstrous conditionals** that pick the proper behavior according to the current state.

Code like this is very **difficult to maintain** because any change to the transition logic may require changing state conditionals in every method. The problem tends to get bigger as the project evolves.

State: Solution

The State pattern suggests that you create **new classes for all possible states** of an object and extract all state-specific behaviors into these classes.

[12] https://refactoring.guru/design-patterns/state

Instead of implementing all behaviors on its own, the original object, called **context**, stores a reference to one of the state objects that represents its current state and **delegates all the state-related work** to that object.

To transition the context into another state, replace the active state object with another object that represents that new state. This is possible only if all state classes follow the **same interface** and the context itself works with these objects through that interface.

State: Solution (cont.)

This structure may look similar to the **Strategy** pattern, but there is one key difference:

In the State pattern, the particular states may be **aware of each other** and initiate transitions from one state to another, whereas strategies almost **never know** about each other.

State: Real-World Analogy

The buttons and switches in your smartphone behave differently depending on the current state of the device:

- When the phone is **unlocked**, pressing buttons leads to executing various functions
- When the phone is **locked**, pressing any button leads to the unlock screen
- When the phone's **charge is low**, pressing any button shows the charging screen

---

State: Structure

The structure consists of the following participants:

1. **Context**: Stores a reference to one of the concrete state objects and delegates to it all state-specific work. The context communicates with the state object via the state interface. The context exposes a setter for passing it a new state object.

2. **State Interface**: Declares the state-specific methods. These methods should make sense for all concrete states because you do not want some of your states to have useless methods that will never be called.

State: Structure (cont.)

3. **Concrete States**: Provide their own implementations for the state-specific methods. To avoid duplication of similar code across multiple states, you may provide intermediate abstract classes.

4. **State Transitions**: Both context and concrete state classes can set the next state of the context and perform the actual state transition by replacing the state object linked to the context.

State: Structure Diagram

State: Java Pseudocode

```
// State interface
interface PlayerState {
    void clickLock(AudioPlayer player);
    void clickPlay(AudioPlayer player);
```

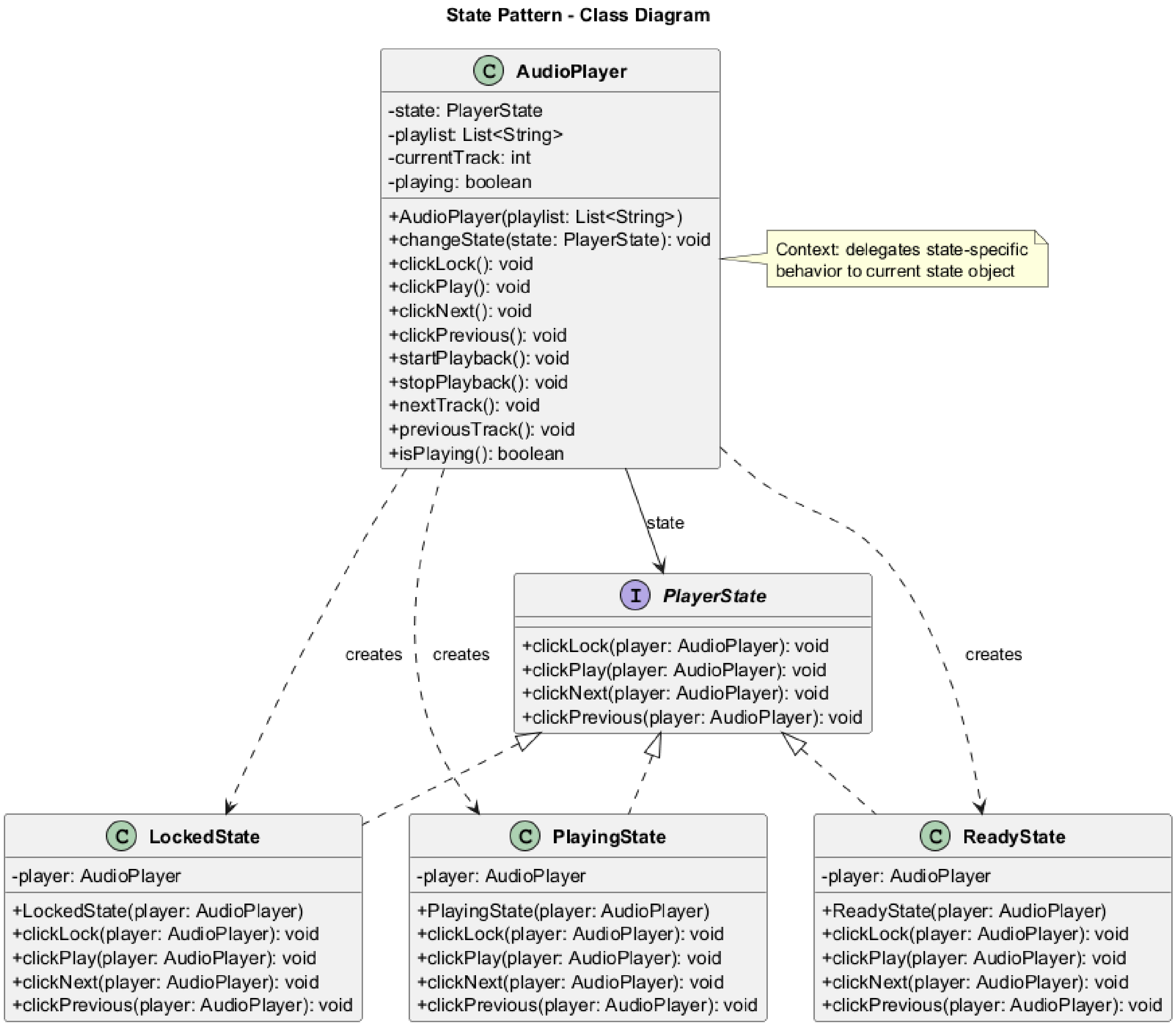
State Pattern - Class Diagram
C AudioPlayer
-state: PlayerState
-playlist: List<String>
-currentTrack: int
-playing: boolean
+AudioPlayer(playlist: List<String>)
+changeState(state: PlayerState): void
+clickLock(): void
+clickPlay(): void
+clickNext(): void
+clickPrevious(): void
+startPlayback(): void
+stopPlayback(): void
+nextTrack(): void
+previousTrack(): void
+isPlaying(): boolean
Context: delegates state-specific
behavior to current state object
state
creates
creates
creates
I PlayerState
+clickLock(player: AudioPlayer): void
+clickPlay(player: AudioPlayer): void
+clickNext(player: AudioPlayer): void
+clickPrevious(player: AudioPlayer): void
C LockedState
-player: AudioPlayer
+LockedState(player: AudioPlayer)
+clickLock(player: AudioPlayer): void
+clickPlay(player: AudioPlayer): void
+clickNext(player: AudioPlayer): void
+clickPrevious(player: AudioPlayer): void
C PlayingState
-player: AudioPlayer
+PlayingState(player: AudioPlayer)
+clickLock(player: AudioPlayer): void
+clickPlay(player: AudioPlayer): void
+clickNext(player: AudioPlayer): void
+clickPrevious(player: AudioPlayer): void
C ReadyState
-player: AudioPlayer
+ReadyState(player: AudioPlayer)
+clickLock(player: AudioPlayer): void
+clickPlay(player: AudioPlayer): void
+clickNext(player: AudioPlayer): void
+clickPrevious(player: AudioPlayer): void

Figure 14: center

```
    void clickNext(AudioPlayer player);
    void clickPrevious(AudioPlayer player);
}
```

State: Java Pseudocode (cont.)

```
import java.util.List;

// Context: Audio Player
class AudioPlayer {
    private PlayerState state;
    private List<String> playlist;
    private int currentTrack = 0;
    private boolean playing = false;

    public AudioPlayer(List<String> playlist) {
        this.playlist = playlist;
        this.state = new ReadyState(this);
    }

    public void changeState(PlayerState state) {
        this.state = state;
    }

    // Delegate to state
    public void clickLock() {
        state.clickLock(this);
    }
    public void clickPlay() {
        state.clickPlay(this);
    }
    public void clickNext() {
        state.clickNext(this);
    }
    public void clickPrevious() {
        state.clickPrevious(this);
    }

    // Player methods
    public void startPlayback() {
        playing = true;
        System.out.println("Playing: "
            + playlist.get(currentTrack));
    }

    public void stopPlayback() {
        playing = false;
        System.out.println("Playback stopped.");
    }

    public void nextTrack() {
        currentTrack = (currentTrack + 1)
            % playlist.size();
        System.out.println("Next: "
            + playlist.get(currentTrack));
    }
```

```java
    public void previousTrack() {
        currentTrack = (currentTrack - 1
            + playlist.size()) % playlist.size();
        System.out.println("Previous: "
            + playlist.get(currentTrack));
    }

    public boolean isPlaying() {
        return playing;
    }
}
```

State: Java Pseudocode (cont.)

```java
// Concrete State: Locked
class LockedState implements PlayerState {
    private AudioPlayer player;

    public LockedState(AudioPlayer player) {
        this.player = player;
    }

    @Override
    public void clickLock(AudioPlayer player) {
        if (player.isPlaying()) {
            player.changeState(
                new PlayingState(player));
        } else {
            player.changeState(
                new ReadyState(player));
        }
        System.out.println("Player unlocked.");
    }

    @Override
    public void clickPlay(AudioPlayer player) {
        System.out.println("Player is locked.");
    }

    @Override
    public void clickNext(AudioPlayer player) {
        System.out.println("Player is locked.");
    }

    @Override
    public void clickPrevious(
            AudioPlayer player) {
        System.out.println("Player is locked.");
    }
}
```

State: Java Pseudocode (cont.)

```java
// Concrete State: Ready
class ReadyState implements PlayerState {
    private AudioPlayer player;

    public ReadyState(AudioPlayer player) {
        this.player = player;
    }

    @Override
    public void clickLock(AudioPlayer player) {
        player.changeState(
            new LockedState(player));
        System.out.println("Player locked.");
    }

    @Override
    public void clickPlay(AudioPlayer player) {
        player.startPlayback();
        player.changeState(
            new PlayingState(player));
    }

    @Override
    public void clickNext(AudioPlayer player) {
        player.nextTrack();
    }

    @Override
    public void clickPrevious(
            AudioPlayer player) {
        player.previousTrack();
    }
}
```

State: Java Pseudocode (cont.)

```java
// Concrete State: Playing
class PlayingState implements PlayerState {
    private AudioPlayer player;

    public PlayingState(AudioPlayer player) {
        this.player = player;
    }

    @Override
    public void clickLock(AudioPlayer player) {
        player.changeState(
            new LockedState(player));
        System.out.println("Player locked.");
    }

    @Override
    public void clickPlay(AudioPlayer player) {
        player.stopPlayback();
        player.changeState(
            new ReadyState(player));
    }

    @Override
    public void clickNext(AudioPlayer player) {
        player.nextTrack();
    }

    @Override
    public void clickPrevious(
            AudioPlayer player) {
        player.previousTrack();
    }
}
```

State: Java Pseudocode (cont.)

```
import java.util.Arrays;

// Client code
public class StateDemo {
    public static void main(String[] args) {
        AudioPlayer player = new AudioPlayer(
            Arrays.asList(
                "Track 1", "Track 2", "Track 3"));

        // Ready state -> play
        player.clickPlay();
        // Playing state -> next
        player.clickNext();
        // Playing state -> lock
        player.clickLock();
        // Locked state -> try play
        player.clickPlay();
        // Locked state -> unlock
        player.clickLock();
    }
}
```

Output:

```
Playing: Track 1
Next: Track 2
Player locked.
Player is locked.
Player unlocked.
```

State: Sequence Diagram

State: Applicability

Use the State pattern when:

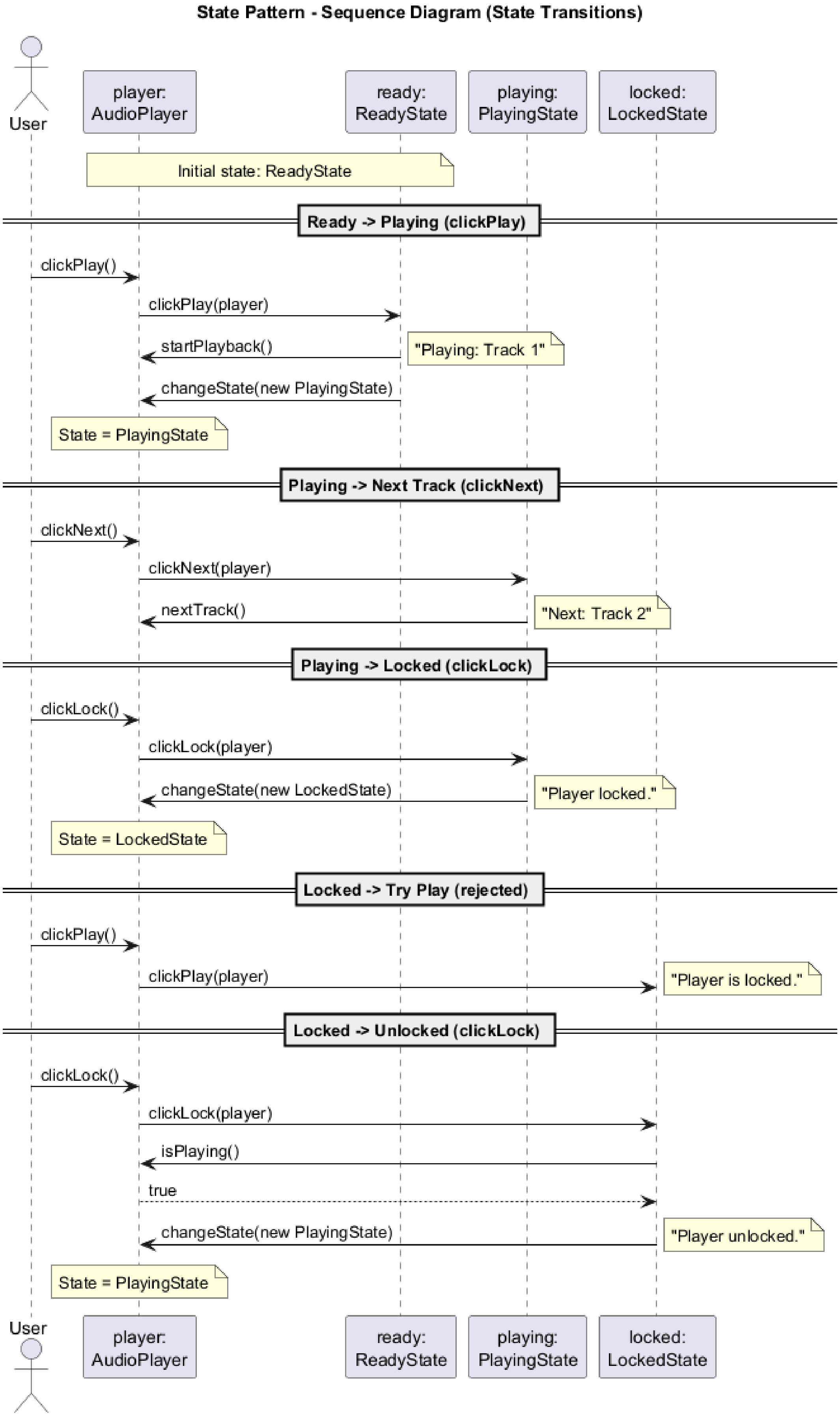

Figure 15: center

- You have an object that behaves differently depending on its current state, the **number of states is enormous**, and the state-specific code changes frequently.
- You have a class polluted with massive conditionals that alter how the class behaves according to **current values of the class's fields**.
- You have a lot of **duplicate code** across similar states and transitions of a condition-based state machine.

State: How to Implement

1. Decide what class will act as the **context**. It could be an existing class with state-dependent code, or a new class.
2. Declare the **state interface**. Although it may mirror all the methods declared in the context, aim only for those that may contain state-specific behavior.
3. For every actual state, create a class that derives from the state interface. Go through the context's methods and **extract all code related to that state** into the newly created class.
4. In the context class, add a reference field of the **state interface** type and a public setter that allows overriding the value of that field.
5. Go over the method of the context again and **replace empty state conditionals** with calls to corresponding methods of the state object.
6. To switch the state of the context, create an instance of one of the state classes and **pass it to the context**.

State: Pros and Cons

Pros:

- **Single Responsibility Principle:** Organize the code related to particular states into separate classes
- **Open/Closed Principle:** Introduce new states without changing existing state classes or the context
- Simplify the code of the context by **eliminating bulky state machine conditionals**

Cons:

- Applying the pattern can be **overkill** if a state machine has only a few states or rarely changes

---

State: Relations with Other Patterns

- **Bridge, State, Strategy (and to some degree Adapter)** have very similar structures. All are based on composition. However, they all solve different problems.
- **State vs. Strategy:** State can be considered an extension of Strategy. Both patterns are based on composition. However, in State, the concrete states may be **aware of each other** and initiate transitions, whereas strategies are **completely independent**.
- **State vs. Finite-State Machine:** State is an alternative to large switch/case statements or if/else chains that implement finite-state machines within a class.

Module H: Takeaway

The **State** pattern allows an object to change its behavior when its internal state changes, as if the object changed its class. It replaces complex conditional logic with polymorphic state objects.

Module I: Strategy Pattern

Module I: Outline

- Intent
- Problem
- Solution
- Real-World Analogy
- Structure
- Java Pseudocode Example
- Applicability
- How to Implement
- Pros and Cons
- Relations with Other Patterns

Strategy: Intent

Strategy is a behavioral design pattern that lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.

Source: refactoring.guru[13]

Strategy: Problem

One day you decided to create a navigation app for casual travelers. The app was centered around a beautiful map which helped users quickly orient themselves in any city.

One of the most requested features for the app was automatic **route planning**. A user should be able to enter an address and see the fastest route to that destination displayed on the map.

Strategy: Problem (cont.)

The first version of the app could only build routes over **roads** for car travel. Then you added **walking** routes. Then **public transport** routing. And it did not stop there – later you planned to add routes for **cyclists**.

From a business perspective, the app was a success. But the technical part caused many headaches. Each time you added a new routing algorithm, the main class of the navigator **doubled in size**. Any change to one algorithm affected the whole class, increasing the chance of creating a bug in already-working code.

Strategy: Solution

The Strategy pattern suggests that you take a class that does something specific in a lot of different ways and **extract all of these algorithms into separate classes** called strategies.

The original class, called **context**, must have a field for storing a reference to one of the strategies. The context delegates the work to a linked strategy object instead of executing it on its own.

The context is not responsible for selecting an appropriate algorithm. Instead, the **client** passes the desired strategy to the context.

[13] https://refactoring.guru/design-patterns/strategy

Strategy: Solution (cont.)

In fact, the context does not know much about strategies. It works with all strategies through the same generic **interface**, which only exposes a single method for triggering the algorithm encapsulated within the selected strategy.

This way the context becomes independent of concrete strategies, so you can add new algorithms or modify existing ones **without changing** the code of the context or other strategies.

Strategy: Real-World Analogy

Imagine that you have to get to the airport. You can catch a **bus**, order a **cab**, or get on your **bicycle**. These are your transportation strategies.

You can pick one of the strategies depending on factors such as budget, time constraints, or personal preference. Each strategy gets you to the same destination but in a different way.

Strategy: Structure

The structure consists of the following participants:

1. **Context**: Maintains a reference to one of the concrete strategies and communicates with this object only via the strategy interface.
2. **Strategy Interface**: Common to all concrete strategies. It declares a method the context uses to execute a strategy.
3. **Concrete Strategies**: Implement different variations of an algorithm the context uses.

Strategy: Structure (cont.)

4. **Execution Flow**: The context calls the execution method on the linked strategy object each time it needs to run the algorithm. The context does not know what type of strategy it works with or how the algorithm is executed.
5. **Client**: Creates a specific strategy object and passes it to the context. The context exposes a setter which lets clients replace the strategy associated with the context at runtime.

Strategy: Structure Diagram

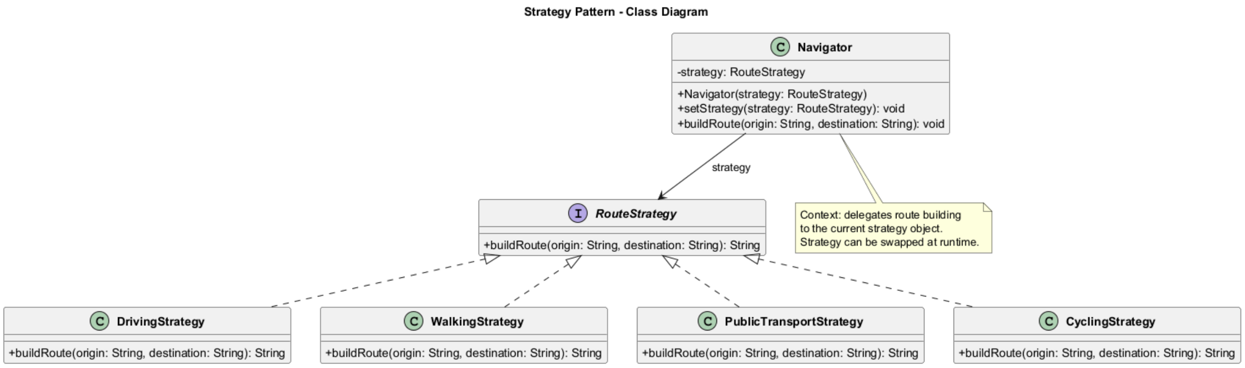
Strategy Pattern - Class Diagram
C Navigator
-strategy: RouteStrategy
+Navigator(strategy: RouteStrategy)
+setStrategy(strategy: RouteStrategy): void
+buildRoute(origin: String, destination: String): void
strategy
I RouteStrategy
+buildRoute(origin: String, destination: String): String
Context: delegates route building
to the current strategy object.
Strategy can be swapped at runtime.
C DrivingStrategy
+buildRoute(origin: String, destination: String): String
C WalkingStrategy
+buildRoute(origin: String, destination: String): String
C PublicTransportStrategy
+buildRoute(origin: String, destination: String): String
C CyclingStrategy
+buildRoute(origin: String, destination: String): String

Figure 16: center

Strategy: Java Pseudocode

```
// Strategy interface
interface RouteStrategy {
    String buildRoute(String origin,
                      String destination);
}
```

Strategy: Java Pseudocode (cont.)

```
// Concrete Strategy: Driving
class DrivingStrategy implements RouteStrategy {
    @Override
    public String buildRoute(String origin,
                             String destination) {
        return "Driving route from " + origin
            + " to " + destination
            + ": Take highway, 25 min, 15 km";
    }
}

// Concrete Strategy: Walking
class WalkingStrategy implements RouteStrategy {
    @Override
    public String buildRoute(String origin,
                             String destination) {
        return "Walking route from " + origin
            + " to " + destination
            + ": Through park, 45 min, 3 km";
    }
}

// Concrete Strategy: Public Transport
class PublicTransportStrategy
        implements RouteStrategy {
    @Override
    public String buildRoute(String origin,
                             String destination) {
        return "Transit route from " + origin
            + " to " + destination
            + ": Bus 42 then Metro, 35 min";
    }
}
```

Strategy: Java Pseudocode (cont.)

```
// Concrete Strategy: Cycling
class CyclingStrategy implements RouteStrategy {
    @Override
    public String buildRoute(String origin,
                             String destination) {
        return "Cycling route from " + origin
```

```
            + " to " + destination
            + ": Bike lane, 20 min, 8 km";
    }
}
```

Strategy: Java Pseudocode (cont.)

```
// Context: Navigator
class Navigator {
    private RouteStrategy strategy;

    public Navigator(RouteStrategy strategy) {
        this.strategy = strategy;
    }

    public void setStrategy(
            RouteStrategy strategy) {
        this.strategy = strategy;
    }

    public void buildRoute(String origin,
                           String destination) {
        String route = strategy.buildRoute(
            origin, destination);
        System.out.println(route);
    }
}
```

Strategy: Java Pseudocode (cont.)

```
// Client code
public class StrategyDemo {
    public static void main(String[] args) {
        Navigator navigator;

        // Use driving strategy
        navigator = new Navigator(
            new DrivingStrategy());
        navigator.buildRoute("Home", "Airport");

        // Switch to walking strategy
        navigator.setStrategy(
            new WalkingStrategy());
        navigator.buildRoute("Home", "Park");

        // Switch to public transport
        navigator.setStrategy(
            new PublicTransportStrategy());
        navigator.buildRoute("Home", "Office");

        // Switch to cycling
        navigator.setStrategy(
            new CyclingStrategy());
        navigator.buildRoute("Home", "Gym");
    }
}
```

Output:

```
Driving route from Home to Airport: ...
Walking route from Home to Park: ...
Transit route from Home to Office: ...
Cycling route from Home to Gym: ...
```

Strategy: Sequence Diagram

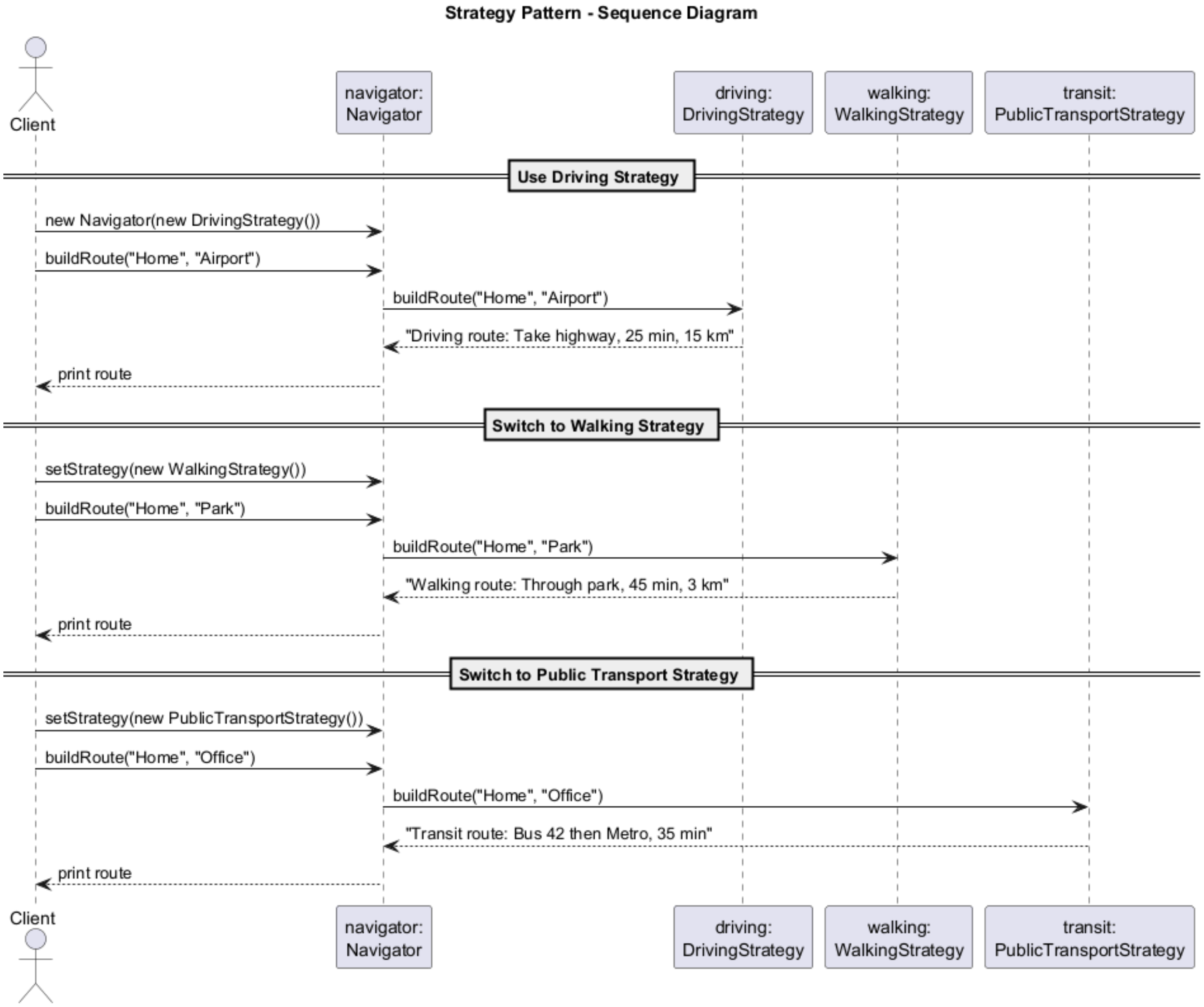
Strategy Pattern - Sequence Diagram
Client
navigator: Navigator
driving: DrivingStrategy
walking: WalkingStrategy
transit: PublicTransportStrategy
Use Driving Strategy
new Navigator(new DrivingStrategy())
buildRoute("Home", "Airport")
buildRoute("Home", "Airport")
"Driving route: Take highway, 25 min, 15 km"
print route
Switch to Walking Strategy
setStrategy(new WalkingStrategy())
buildRoute("Home", "Park")
buildRoute("Home", "Park")
"Walking route: Through park, 45 min, 3 km"
print route
Switch to Public Transport Strategy
setStrategy(new PublicTransportStrategy())
buildRoute("Home", "Office")
buildRoute("Home", "Office")
"Transit route: Bus 42 then Metro, 35 min"
print route
Client
navigator: Navigator
driving: DrivingStrategy
walking: WalkingStrategy
transit: PublicTransportStrategy

Figure 17: center

Strategy: Applicability

Use the Strategy pattern when:

- You want to use **different variants of an algorithm** within an object and be able to switch from one algorithm to another during runtime.

- You have a lot of similar classes that only differ in the way they **execute some behavior**.
- You want to isolate the **business logic** of a class from the implementation details of algorithms that may not be as important in the context of that logic.
- Your class has a massive **conditional operator** that switches between different variants of the same algorithm.

Strategy: How to Implement

1. In the context class, identify an **algorithm** that is prone to frequent changes. It may also be a massive conditional that selects and executes a variant of the same algorithm at runtime.
2. Declare the **strategy interface** common to all variants of the algorithm.
3. One by one, extract all algorithms into their own classes. They should all **implement the strategy interface**.
4. In the context class, add a **field for storing a reference** to a strategy object. Provide a setter for replacing values of that field. The context should work with the strategy object only through the strategy interface.
5. **Clients** of the context must associate it with a suitable strategy that matches the way they expect the context to perform its primary job.

Strategy: Pros and Cons

Pros:

- You can **swap algorithms** used inside an object at runtime
- You can **isolate the implementation details** of an algorithm from the code that uses it
- You can **replace inheritance with composition**
- **Open/Closed Principle:** You can introduce new strategies without having to change the context

Cons:

- If you only have a **couple of algorithms** and they rarely change, there is no real reason to overcomplicate the program
- **Clients must be aware** of the differences between strategies to be able to select a proper one
- A lot of modern programming languages have functional type support that lets you implement different versions of an algorithm inside a set of **anonymous functions**, making the pattern somewhat redundant

---

Strategy: Relations with Other Patterns

- **Bridge, State, Strategy (and to some degree Adapter)** have very similar structures based on composition, but they solve different problems.
- **Command vs. Strategy:** Both parameterize objects. Command converts any operation into an object (for deferred execution, queueing, history of operations). Strategy describes different ways of doing the same thing, letting you swap algorithms within a single context class.
- **Decorator vs. Strategy:** Decorator lets you change the skin of an object (external appearance). Strategy lets you change the guts (internal behavior).
- **Template Method vs. Strategy:** Template Method is based on inheritance at the class level (static). Strategy is based on composition at the object level (dynamic, runtime switching).
- **State vs. Strategy:** State can be considered an extension of Strategy. In State, concrete states may be aware of each other. In Strategy, implementations are completely independent.

Module I: Takeaway

The **Strategy** pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. It lets the algorithm vary independently from clients that use it, promoting composition over inheritance.

Module J: Template Method Pattern

Module J: Outline

- Intent
- Problem
- Solution
- Real-World Analogy
- Structure
- Java Pseudocode Example
- Applicability
- How to Implement
- Pros and Cons
- Relations with Other Patterns

Template Method: Intent

Template Method is a behavioral design pattern that defines the skeleton of an algorithm in the superclass but lets subclasses override specific steps of the algorithm without changing its structure.

Source: refactoring.guru[14]

Template Method: Problem

Imagine you are creating a data mining application that analyzes corporate documents. Users feed the app documents in various formats (PDF, DOC, CSV), and the app tries to extract meaningful data from these documents in a uniform format.

The first version of the app could only work with DOC files. In the following version, it was able to support CSV files. A month later, you “taught” it to extract data from PDF files.

Template Method: Problem (cont.)

At some point, you noticed that all three classes have **a lot of similar code**. While the code for dealing with various data formats was entirely different in all classes, the code for data processing and analysis was almost identical.

It would be nice to get rid of the code duplication, leaving the algorithm structure intact. There was another problem related to **client code** that used these classes. It had lots of conditionals that picked a proper course of action depending on the class of the processing object.

---

[14] https://refactoring.guru/design-patterns/template-method

Template Method: Solution

The Template Method pattern suggests that you break down an algorithm into a series of steps, turn these steps into methods, and put a series of calls to these methods inside a single **template method**.

The steps may either be `abstract` (every subclass must provide its own implementation), or have some **default implementation**.

Template Method: Solution (cont.)

To use the algorithm, the client is supposed to provide its own subclass, implement all abstract steps, and override some of the optional ones if needed (but not the template method itself).

There is another type of step called **hooks**. A hook is an optional step with an empty body. A template method would work even if a hook is not overridden. Usually, hooks are placed before and after crucial steps of algorithms, providing subclasses with additional extension points.

Template Method: Real-World Analogy

The template method approach can be used in **mass housing construction**. The architectural plan for building a standard house may contain several extension points that would let a potential owner adjust some details of the resulting house.

Each building step, such as laying the foundation, framing, building walls, installing plumbing and wiring, etc., can be slightly altered to make the resulting house a little bit different from others. But the overall structure and order of construction remain the same.

Template Method: Structure

The structure consists of the following participants:

1. **Abstract Class**: Declares methods that act as steps of an algorithm, as well as the actual template method which calls these methods in a specific order. The steps may either be declared `abstract` or have some default implementation.
2. **Concrete Classes**: Can override all of the steps, but **not the template method itself**. Concrete classes provide implementations for the abstract steps and may optionally override some steps that have default implementations.

Template Method: Structure (cont.)

Types of steps:

- **Abstract steps** (must be implemented by every subclass)
- **Optional steps** (have default implementation but can be overridden)
- **Hooks** (optional steps with empty body, placed before/after crucial algorithm steps)

Template Method: Structure Diagram

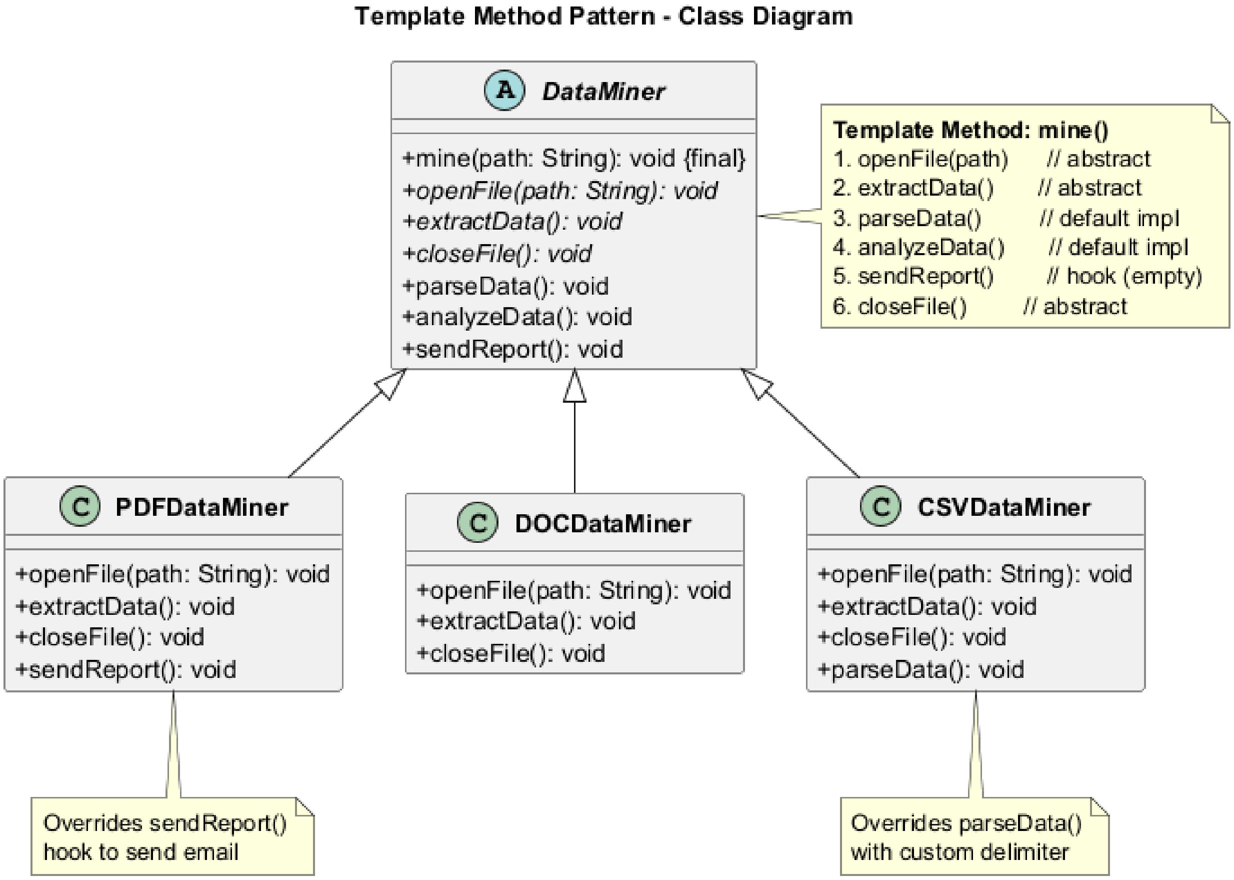

Figure 18: center

Template Method: Java Pseudocode

```java
// Abstract class with template method
abstract class DataMiner {

    // Template method - defines algorithm skeleton
    // This method should NOT be overridden!
    public final void mine(String path) {
        openFile(path);
        extractData();
        parseData();
        analyzeData();
        sendReport();
        closeFile();
    }

    // Abstract steps - must be implemented
    abstract void openFile(String path);
    abstract void extractData();
    abstract void closeFile();

    // Default implementation steps
    void parseData() {
        System.out.println("Parsing raw data"
            + " into structured format...");
    }

    void analyzeData() {
        System.out.println("Analyzing data"
            + " for patterns...");
    }

    // Hook - optional step with
    // default (empty) body
    void sendReport() {
        // Subclasses may override this
    }
}
```

Template Method: Java Pseudocode (cont.)

```java
// Concrete class: PDF Data Miner
class PDFDataMiner extends DataMiner {
    @Override
    void openFile(String path) {
        System.out.println(
            "Opening PDF file: " + path);
    }

    @Override
    void extractData() {
        System.out.println("Extracting data"
            + " from PDF using PDF parser...");
    }

    @Override
    void closeFile() {
        System.out.println("Closing PDF file.");
    }

    @Override
    void sendReport() {
        System.out.println("Sending PDF"
            + " mining report via email...");
    }
}
```

Template Method: Java Pseudocode (cont.)

```
// Concrete class: DOC Data Miner
class DOCDataMiner extends DataMiner {
    @Override
    void openFile(String path) {
        System.out.println(
            "Opening DOC file: " + path);
    }

    @Override
    void extractData() {
        System.out.println("Extracting data"
            + " from DOC using Word API...");
    }

    @Override
    void closeFile() {
        System.out.println("Closing DOC file.");
    }
}
```

Template Method: Java Pseudocode (cont.)

```
// Concrete class: CSV Data Miner
class CSVDataMiner extends DataMiner {
    @Override
    void openFile(String path) {
        System.out.println(
            "Opening CSV file: " + path);
    }

    @Override
    void extractData() {
        System.out.println("Extracting data"
            + " from CSV line by line...");
    }

    @Override
    void closeFile() {
        System.out.println("Closing CSV file.");
    }
    @Override
    void parseData() {
        System.out.println("Parsing CSV data"
            + " with custom delimiter"
            + " handler...");
    }
}
```

Template Method: Java Pseudocode (cont.)

```
// Game AI example
abstract class GameAI {
    // Template method
    public final void turn() {
        collectResources();
        buildStructures();
        buildUnits();
        attack();
    }

    // Default implementation
    void collectResources() {
        System.out.println(
            "Collecting resources"
            + " from controlled buildings...");
    }

    abstract void buildStructures();
    abstract void buildUnits();

    void attack() {
        System.out.println(
            "Sending all units"
            + " to the closest enemy...");
    }
}
```

Template Method: Java Pseudocode (cont.)

```
class OrcsAI extends GameAI {
    @Override
    void buildStructures() {
        System.out.println("Building: Barracks,"
            + " War Camp, Stronghold");
    }

    @Override
    void buildUnits() {
        System.out.println("Training: Grunts,"
            + " Raiders, Wyverns");
    }

    @Override
    void attack() {
```

```java
        System.out.println("Orcs: Charging with"
            + " berserker rage!");
    }
}

class MonstersAI extends GameAI {
    @Override
    void buildStructures() {
        // Monsters don't build
    }

    @Override
    void buildUnits() {
        // Monsters don't train units
    }

    @Override
    void collectResources() {
        // Monsters don't collect resources
    }

    @Override
    void attack() {
        System.out.println("Monsters: Swarming"
            + " the nearest village!");
    }
}
```

Template Method: Java Pseudocode (cont.)

```java
// Client code
public class TemplateMethodDemo {
    public static void main(String[] args) {
        System.out.println("=== PDF Mining ===");
        DataMiner pdfMiner = new PDFDataMiner();
        pdfMiner.mine("/data/report.pdf");

        System.out.println(
            "\n=== DOC Mining ===");
        DataMiner docMiner = new DOCDataMiner();
        docMiner.mine("/data/report.doc");

        System.out.println(
            "\n=== CSV Mining ===");
        DataMiner csvMiner = new CSVDataMiner();
        csvMiner.mine("/data/report.csv");

        System.out.println(
            "\n=== Game AI ===");
        GameAI orc = new OrcsAI();
        orc.turn();

        GameAI monster = new MonstersAI();
        monster.turn();

        // Expected Output:
```

```
        // === PDF Mining ===
        // Opening PDF file: /data/report.pdf
        // Extracting text from PDF...
        // Parsing data...
        // Analyzing data...
        // Sending report...
        //
        // === DOC Mining ===
        // Opening DOC file: /data/report.doc
        // Extracting text from DOC...
        // Parsing data...
        // Analyzing data...
        // Sending report...
        //
        // === CSV Mining ===
        // Opening CSV file: /data/report.csv
        // Extracting text from CSV...
        // Parsing data...
        // Analyzing data...
        // Sending report...
        //
        // === Game AI ===
        // Orcs: Building structures...
        // Orcs: Collecting gold and wood...
        // Orcs: Sending scouts to attack!
        // Monsters: Swarming the nearest village!
    }
}
```

Template Method: Sequence Diagram

Template Method: Applicability

Use the Template Method pattern when:

- You want to let clients extend only **particular steps** of an algorithm, but not the whole algorithm or its structure.
- You have several classes that contain **nearly identical algorithms** with some minor differences. When the algorithm changes, you might need to modify all of the classes.
- You want to **eliminate code duplication** by pulling up the common behavior into a superclass.

Template Method: How to Implement

1. **Analyze** the target algorithm to see whether you can break it into steps. Consider which steps are common to all subclasses and which ones will always be unique.
2. Create the **abstract base class** and declare the template method and a set of abstract methods representing the algorithm's steps. Outline the algorithm's structure in the template method by executing corresponding steps. Consider making the template method `final` to prevent subclasses from overriding it.
3. It is fine if all the steps end up being abstract. However, some steps might benefit from having a **default implementation**. Subclasses do not have to implement those methods.
4. Think of adding **hooks** between the crucial steps of the algorithm.

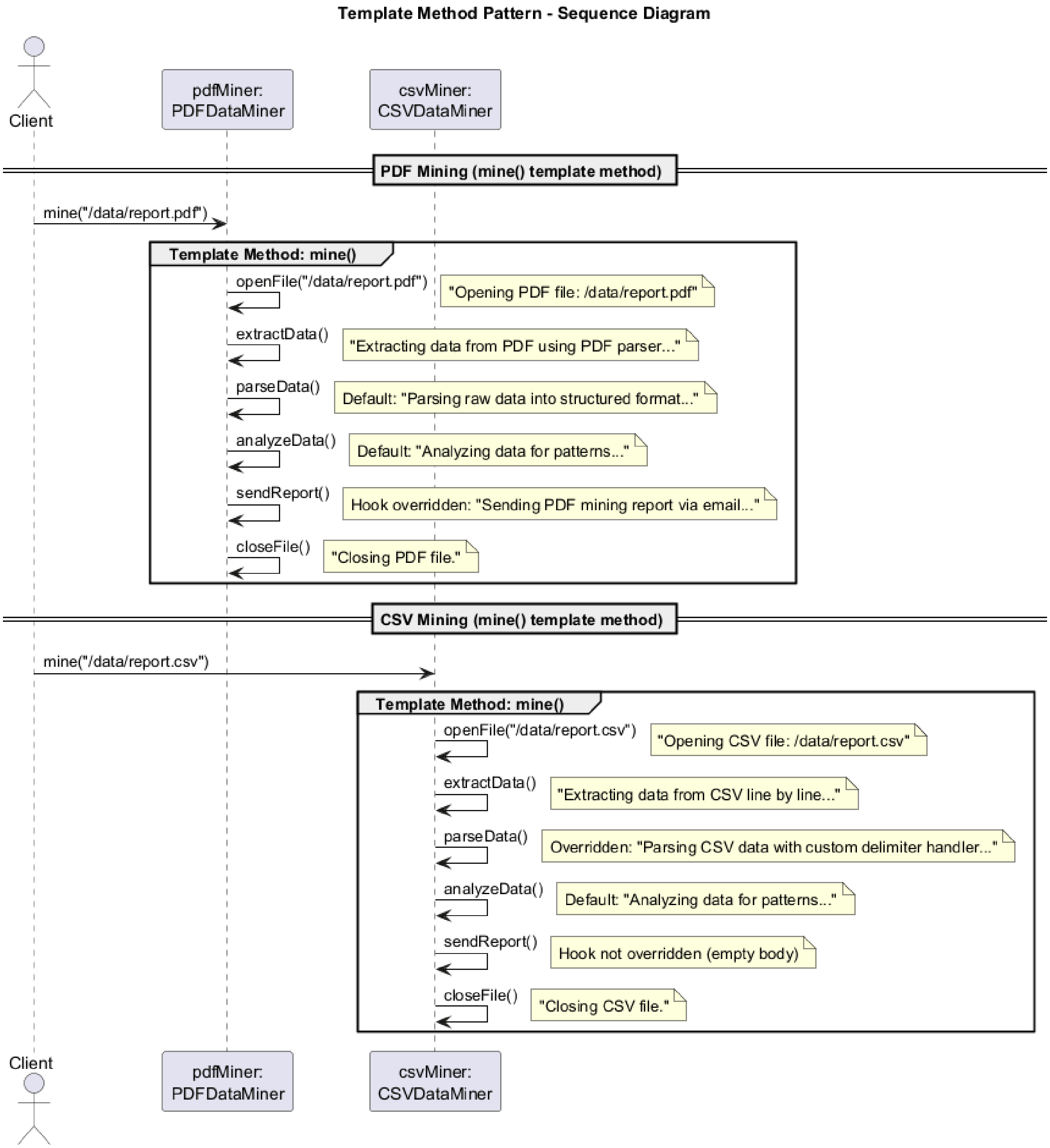
Template Method Pattern - Sequence Diagram
Client
pdfMiner: PDFDataMiner
csvMiner: CSVDataMiner
PDF Mining (mine() template method)
mine("/data/report.pdf")
Template Method: mine()
openFile("/data/report.pdf")
"Opening PDF file: /data/report.pdf"
extractData()
"Extracting data from PDF using PDF parser..."
parseData()
Default: "Parsing raw data into structured format..."
analyzeData()
Default: "Analyzing data for patterns..."
sendReport()
Hook overridden: "Sending PDF mining report via email..."
closeFile()
"Closing PDF file."
CSV Mining (mine() template method)
mine("/data/report.csv")
Template Method: mine()
openFile("/data/report.csv")
"Opening CSV file: /data/report.csv"
extractData()
"Extracting data from CSV line by line..."
parseData()
Overridden: "Parsing CSV data with custom delimiter handler..."
analyzeData()
Default: "Analyzing data for patterns..."
sendReport()
Hook not overridden (empty body)
closeFile()
"Closing CSV file."
Client
pdfMiner: PDFDataMiner
csvMiner: CSVDataMiner

Figure 19: center

5. For each variation of the algorithm, create a new **concrete subclass**. It must implement all of the abstract steps, but may also override some of the optional ones.

Template Method: Pros and Cons

Pros:

- You can let clients **override only certain parts** of a large algorithm, making them less affected by changes that happen to other parts
- You can pull the **duplicate code** into a superclass

Cons:

- Some clients may be **limited by the provided skeleton** of an algorithm
- You might violate the **Liskov Substitution Principle** by suppressing a default step implementation via a subclass
- Template methods tend to be **harder to maintain** the more steps they have

---

Template Method: Relations with Other Patterns

- **Factory Method** is a specialization of Template Method. At the same time, a Factory Method may serve as a step in a large Template Method.
- **Template Method vs. Strategy:** Template Method is based on **inheritance**: it lets you alter parts of an algorithm by extending those parts in subclasses. Strategy is based on **composition**: you can alter parts of the object's behavior by supplying it with different strategies. Template Method works at the class level, so it is static. Strategy works on the object level, letting you switch behaviors at runtime.

Module J: Takeaway

The **Template Method** pattern defines the skeleton of an algorithm in an operation, deferring some steps to subclasses. It lets subclasses redefine certain steps of an algorithm without changing its structure, promoting code reuse through inheritance.

Module K: Visitor Pattern

Module K: Outline

- Intent
- Problem
- Solution
- Real-World Analogy
- Structure
- Java Pseudocode Example
- Applicability
- How to Implement
- Pros and Cons
- Relations with Other Patterns

Visitor: Intent

Visitor is a behavioral design pattern that lets you separate algorithms from the objects on which they operate.

Source: refactoring.guru[15]

Visitor: Problem

Imagine that your team develops an app which works with geographic information structured as one colossal graph. Each node of the graph may represent a complex entity such as a city, an industry area, a sightseeing location, etc.

At some point, you got a task to implement **exporting the graph into XML format**. The job seemed pretty straightforward. You planned to add an export method to each node class.

Visitor: Problem (cont.)

But the system architect refused to let you alter existing node classes. He said that the code was already in production and he did not want to risk breaking it because of a potential bug in your changes.

Besides, he questioned whether it makes sense to have the **XML export code inside the node classes**. The primary job of these classes was to work with geodata. The XML export behavior would look alien there.

There was another reason for the refusal. It was highly likely that after this feature was implemented, someone from marketing would ask you to provide the ability to export into a **different format**, or request some other weird thing, leading to constant changes in these precious node classes.

Visitor: Solution

The Visitor pattern suggests that you place the new behavior into a separate class called **visitor**, instead of trying to integrate it into existing classes. The original object that had to perform the behavior is now passed to one of the visitor's methods as an argument, providing the method access to all necessary data contained within the object.

Visitor: Solution (cont.)

Now, the behavior can be executed over objects of different classes. The visitor class may define a set of methods, each taking arguments of different types:

```
visitor.visit(city)
visitor.visit(industry)
visitor.visit(sightseeing)
```

But how exactly would we call these methods, especially when dealing with the whole graph? These methods have **different signatures**, so we cannot use polymorphism directly.

---

[15] https://refactoring.guru/design-patterns/visitor

Visitor: Solution – Double Dispatch

The Visitor pattern uses a technique called **Double Dispatch**. Instead of letting the client select a proper version of the method to call, we delegate this choice to objects we are passing to the visitor as an argument:

```
// Each element "accepts" the visitor
// and calls the appropriate visitor method
foreach (Node node : graph)
    node.accept(exportVisitor)

// Inside each concrete element:
class City {
    void accept(Visitor v) {
        v.visitCity(this);
    }
}
```

Since the object knows its own class, it can pick the proper method on the visitor itself.

Visitor: Real-World Analogy

Imagine a seasoned **insurance agent** who is eager to get new customers. He can visit every building in a neighborhood, trying to sell insurance to everyone he meets:

- To a **residential building**, he sells medical insurance
- To a **bank**, he sells theft insurance
- To a **coffee shop**, he sells fire and flood insurance

The agent represents the visitor, and the buildings represent the elements. The agent tailors his pitch (algorithm) based on the type of building (element) he visits.

Visitor: Structure

The structure consists of the following participants:

1. **Visitor Interface**: Declares a set of visiting methods that can take concrete elements of an object structure as arguments. These methods may have the same names if the language supports overloading, but the type of their parameters must be different.

2. **Concrete Visitors**: Implement several versions of the same behaviors, tailored for different concrete element classes.

Visitor: Structure (cont.)

3. **Element Interface**: Declares a method for “accepting” visitors. This method should have one parameter declared with the type of the visitor interface.

4. **Concrete Elements**: Must implement the acceptance method. The purpose of this method is to redirect the call to the proper visitor’s method corresponding to the current element class.

5. **Client**: Usually represents a collection or some other complex object. The client creates visitor objects and passes them to elements via the `accept` method.

Visitor: Structure Diagram

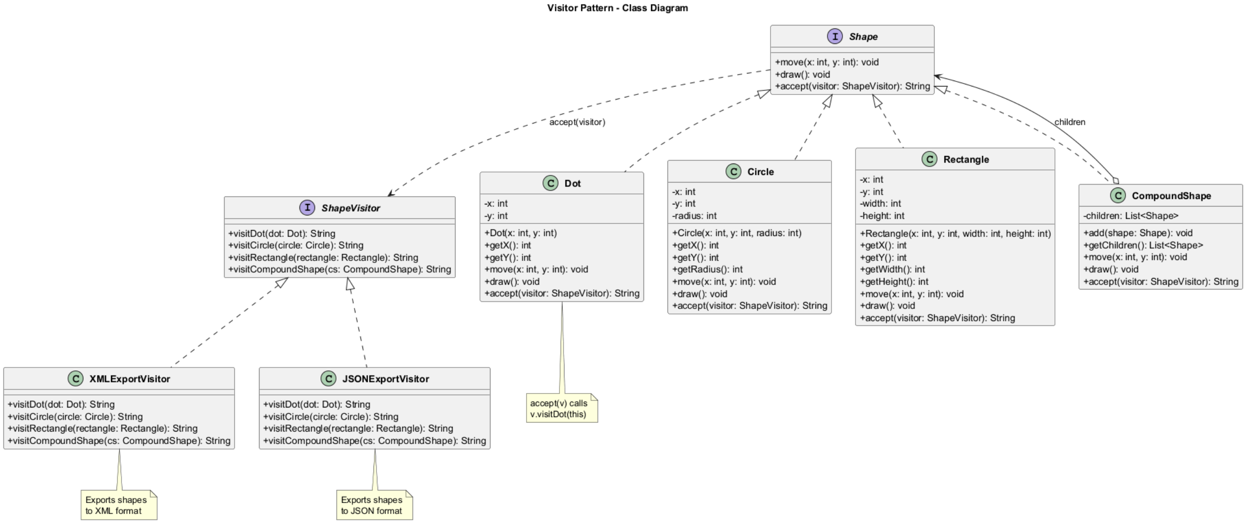
Visitor Pattern - Class Diagram
Shape
+move(x: int, y: int): void
+draw(): void
+accept(visitor: ShapeVisitor): String
accept(visitor)
children
ShapeVisitor
+visitDot(dot: Dot): String
+visitCircle(circle: Circle): String
+visitRectangle(rectangle: Rectangle): String
+visitCompoundShape(cs: CompoundShape): String
Dot
-x: int
-y: int
+Dot(x: int, y: int)
+getX(): int
+getY(): int
+move(x: int, y: int): void
+draw(): void
+accept(visitor: ShapeVisitor): String
Circle
-x: int
-y: int
-radius: int
+Circle(x: int, y: int, radius: int)
+getX(): int
+getY(): int
+getRadius(): int
+move(x: int, y: int): void
+draw(): void
+accept(visitor: ShapeVisitor): String
Rectangle
-x: int
-y: int
-width: int
-height: int
+Rectangle(x: int, y: int, width: int, height: int)
+getX(): int
+getY(): int
+getWidth(): int
+getHeight(): int
+move(x: int, y: int): void
+draw(): void
+accept(visitor: ShapeVisitor): String
CompoundShape
-children: List<Shape>
+add(shape: Shape): void
+getChildren(): List<Shape>
+move(x: int, y: int): void
+draw(): void
+accept(visitor: ShapeVisitor): String
XMLExportVisitor
+visitDot(dot: Dot): String
+visitCircle(circle: Circle): String
+visitRectangle(rectangle: Rectangle): String
+visitCompoundShape(cs: CompoundShape): String
JSONExportVisitor
+visitDot(dot: Dot): String
+visitCircle(circle: Circle): String
+visitRectangle(rectangle: Rectangle): String
+visitCompoundShape(cs: CompoundShape): String
accept(v) calls
v.visitDot(this)
Exports shapes
to XML format
Exports shapes
to JSON format

Figure 20: center

Visitor: Java Pseudocode

```java
// Visitor interface
interface ShapeVisitor {
    String visitDot(Dot dot);
    String visitCircle(Circle circle);
    String visitRectangle(
        Rectangle rectangle);
    String visitCompoundShape(
        CompoundShape compoundShape);
}
```

Visitor: Java Pseudocode (cont.)

```java
// Element interface
interface Shape {
    void move(int x, int y);
    void draw();
    String accept(ShapeVisitor visitor);
}
```

Visitor: Java Pseudocode (cont.)

```java
// Concrete Element: Dot
class Dot implements Shape {
    private int x, y;

    public Dot(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() { return x; }
    public int getY() { return y; }
    @Override
    public void move(int x, int y) {
        this.x += x;
        this.y += y;
    }

    @Override
    public void draw() {
        System.out.println("Drawing dot at ("
            + x + "," + y + ")");
    }

    @Override
    public String accept(ShapeVisitor visitor) {
        return visitor.visitDot(this);
    }
}
```

Visitor: Java Pseudocode (cont.)

```
// Concrete Element: Circle
class Circle implements Shape {
    private int x, y, radius;

    public Circle(int x, int y, int radius) {
        this.x = x;
        this.y = y;
        this.radius = radius;
    }

    public int getX() { return x; }
    public int getY() { return y; }
    public int getRadius() { return radius; }

    @Override
    public void move(int x, int y) {
        this.x += x;
        this.y += y;
    }

    @Override
    public void draw() {
        System.out.println(
            "Drawing circle at ("
            + x + "," + y + ") r=" + radius);
    }

    @Override
    public String accept(ShapeVisitor visitor) {
        return visitor.visitCircle(this);
    }
}
```

Visitor: Java Pseudocode (cont.)

```java
// Concrete Element: Rectangle
class Rectangle implements Shape {
    private int x, y, width, height;

    public Rectangle(int x, int y,
                     int width, int height) {
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
    }

    public int getX() { return x; }
    public int getY() { return y; }
    public int getWidth() { return width; }
    public int getHeight() { return height; }

    @Override
    public void move(int x, int y) {
        this.x += x;
        this.y += y;
    }

    @Override
    public void draw() {
        System.out.println(
            "Drawing rectangle at ("
            + x + "," + y + ") "
            + width + "x" + height);
    }

    @Override
    public String accept(ShapeVisitor visitor) {
        return visitor.visitRectangle(this);
    }
}
```

Visitor: Java Pseudocode (cont.)

```java
import java.util.ArrayList;
import java.util.List;

// Concrete Element: Compound Shape
class CompoundShape implements Shape {
    private List<Shape> children
        = new ArrayList<>();

    public void add(Shape shape) {
        children.add(shape);
    }

    public List<Shape> getChildren() {
        return children;
    }
    @Override
    public void move(int x, int y) {
        for (Shape child : children) {
            child.move(x, y);
        }
    }

    @Override
    public void draw() {
        System.out.println(
            "Drawing compound shape:");
        for (Shape child : children) {
            child.draw();
        }
    }

    @Override
    public String accept(ShapeVisitor visitor) {
        return visitor.visitCompoundShape(this);
    }
}
```

Visitor: Java Pseudocode (cont.)

```
// Concrete Visitor: XML Export
class XMLExportVisitor
        implements ShapeVisitor {
    @Override
    public String visitDot(Dot dot) {
        return "<dot>\n"
            + "  <x>" + dot.getX() + "</x>\n"
            + "  <y>" + dot.getY() + "</y>\n"
            + "</dot>\n";
    }

    @Override
    public String visitCircle(Circle circle) {
        return "<circle>\n"
            + "  <x>" + circle.getX() + "</x>\n"
            + "  <y>" + circle.getY() + "</y>\n"
            + "  <radius>" + circle.getRadius()
            + "</radius>\n"
            + "</circle>\n";
    }

    @Override
    public String visitRectangle(
            Rectangle rect) {
        return "<rectangle>\n"
            + "  <x>" + rect.getX() + "</x>\n"
            + "  <y>" + rect.getY() + "</y>\n"
            + "  <width>" + rect.getWidth()
            + "</width>\n"
            + "  <height>" + rect.getHeight()
            + "</height>\n"
            + "</rectangle>\n";
    }

    @Override
    public String visitCompoundShape(
            CompoundShape cs) {
        StringBuilder sb = new StringBuilder();
        sb.append("<compound>\n");
        for (Shape child : cs.getChildren()) {
            sb.append(child.accept(this));
        }
        sb.append("</compound>\n");
        return sb.toString();
    }
}
```

Visitor: Java Pseudocode (cont.)

```
// Concrete Visitor: JSON Export
class JSONExportVisitor
        implements ShapeVisitor {
    @Override
    public String visitDot(Dot dot) {
        return "{\"type\":\"dot\","
            + "\"x\":" + dot.getX()
            + ",\"y\":" + dot.getY() + "}";
    }

    @Override
    public String visitCircle(Circle circle) {
        return "{\"type\":\"circle\","
            + "\"x\":" + circle.getX()
            + ",\"y\":" + circle.getY()
            + ",\"radius\":"
            + circle.getRadius() + "}";
    }

    @Override
    public String visitRectangle(
            Rectangle rect) {
        return "{\"type\":\"rectangle\","
            + "\"x\":" + rect.getX()
            + ",\"y\":" + rect.getY()
            + ",\"width\":" + rect.getWidth()
            + ",\"height\":"
            + rect.getHeight() + "}";
    }

    @Override
    public String visitCompoundShape(
            CompoundShape cs) {
        StringBuilder sb = new StringBuilder();
        sb.append("{\"type\":\"compound\","
            + "\"children\":[");
        List<Shape> children = cs.getChildren();
        for (int i = 0;
```

```java
                i < children.size(); i++) {
            sb.append(
                children.get(i).accept(this));
            if (i < children.size() - 1)
                sb.append(",");
        }
        sb.append("]}");
        return sb.toString();
    }
}
```

Visitor: Java Pseudocode (cont.)

```java
// Client code
public class VisitorDemo {
    public static void main(String[] args) {
        // Build shape structure
        CompoundShape allShapes =
            new CompoundShape();
        allShapes.add(new Dot(1, 2));
        allShapes.add(new Circle(5, 5, 10));
        allShapes.add(
            new Rectangle(10, 10, 100, 50));

        CompoundShape nested =
            new CompoundShape();
        nested.add(new Dot(20, 30));
        nested.add(new Circle(25, 25, 5));
        allShapes.add(nested);

        // Export to XML
        XMLExportVisitor xmlExporter =
            new XMLExportVisitor();
        System.out.println("XML Export:");
        System.out.println(
            allShapes.accept(xmlExporter));

        // Export to JSON
        JSONExportVisitor jsonExporter =
            new JSONExportVisitor();
        System.out.println("\nJSON Export:");
        System.out.println(
            allShapes.accept(jsonExporter));

        // Expected Output:
        // XML Export:
        // <compound>
        //   <dot x="1" y="2"/>
        //   <circle x="5" y="5" radius="10"/>
        //   <rectangle x="10" y="10" width="100" height="50"/>
        //   <compound>
        //     <dot x="20" y="30"/>
        //     <circle x="25" y="25" radius="5"/>
        //   </compound>
        // </compound>
        //
        // JSON Export:
        // {"type":"compound","children":[
        //   {"type":"dot","x":1,"y":2},
        //   {"type":"circle","x":5,"y":5,"radius":10},
        //   {"type":"rectangle","x":10,"y":10,"width":100,"height":50},
        //   {"type":"compound","children":[
        //     {"type":"dot","x":20,"y":30},
        //     {"type":"circle","x":25,"y":25,"radius":5}
        //   ]}
        // ]}
    }
}
```

Visitor: Sequence Diagram

Visitor: Applicability

Use the Visitor pattern when:

- You need to perform an operation on all elements of a **complex object structure** (for example, an object tree) that have **different classes**.
- You want to **clean up the business logic** of auxiliary behaviors. The pattern lets you make the primary classes of your app more focused on their main jobs by extracting all other behaviors into a set of visitor classes.
- A behavior makes sense only in some classes of a class hierarchy, **but not in others**. You can extract this behavior into a separate visitor class and implement only those visiting methods that accept objects of relevant classes, leaving the rest empty.

Visitor: How to Implement

1. Declare the **visitor interface** with a set of "visiting" methods, one per each concrete element class that exists in the program.
2. Declare the **element interface**. If you are working with an existing element class hierarchy, add the abstract "acceptance" method to the base class of the hierarchy.
3. Implement the **acceptance methods** in all concrete element classes. These methods must simply redirect the call to a visiting method on the incoming visitor object which matches the class of the current element.
4. The element classes should only work with visitors via the visitor interface. Visitors, however, must be **aware of all concrete element classes**, referenced as parameter types of the visiting methods.

Visitor: How to Implement (cont.)

5. For each behavior that cannot be put inside the element hierarchy, create a **new concrete visitor class** and implement all of the visiting methods.
6. You might encounter a situation where the visitor needs **access to some private** members of the element class. In this case, you can either make these fields or methods public, violating the element's encapsulation, or nest the visitor class in the element class (if your language supports nested classes).
7. The **client** must create visitor objects and pass them into elements via "acceptance" methods.

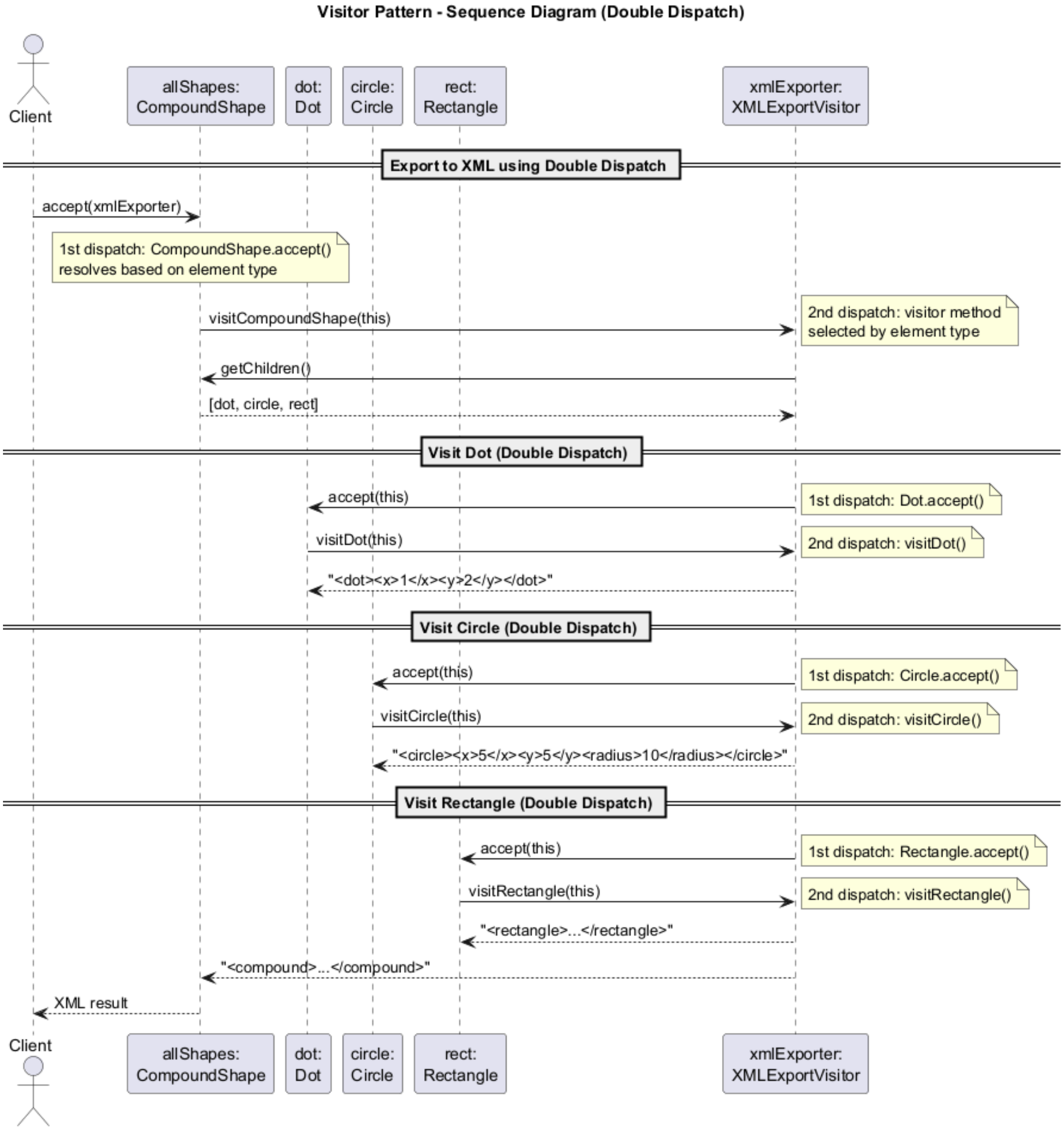
Visitor Pattern - Sequence Diagram (Double Dispatch)
Client
allShapes:
CompoundShape
dot:
Dot
circle:
Circle
rect:
Rectangle
xmlExporter:
XMLExportVisitor
Export to XML using Double Dispatch
accept(xmlExporter)
1st dispatch: CompoundShape.accept()
resolves based on element type
visitCompoundShape(this)
2nd dispatch: visitor method
selected by element type
getChildren()
[dot, circle, rect]
Visit Dot (Double Dispatch)
accept(this)
1st dispatch: Dot.accept()
visitDot(this)
2nd dispatch: visitDot()
"<dot><x>1</x><y>2</y></dot>"
Visit Circle (Double Dispatch)
accept(this)
1st dispatch: Circle.accept()
visitCircle(this)
2nd dispatch: visitCircle()
"<circle><x>5</x><y>5</y><radius>10</radius></circle>"
Visit Rectangle (Double Dispatch)
accept(this)
1st dispatch: Rectangle.accept()
visitRectangle(this)
2nd dispatch: visitRectangle()
"<rectangle>...</rectangle>"
"<compound>...</compound>"
XML result
Client
allShapes:
CompoundShape
dot:
Dot
circle:
Circle
rect:
Rectangle
xmlExporter:
XMLExportVisitor

Figure 21: center

Visitor: Pros and Cons

Pros:

- **Open/Closed Principle:** You can introduce a new behavior that can work with objects of different classes without changing these classes
- **Single Responsibility Principle:** You can move multiple versions of the same behavior into the same class
- A visitor object can **accumulate** some useful information while working with various objects, which is handy when traversing complex structures like object trees

Cons:

- You need to **update all visitors** each time a class gets added to or removed from the element hierarchy
- Visitors might lack the necessary **access to the private fields** and methods of the elements that they are supposed to work with
- The pattern **increases complexity** of the codebase

Visitor: Relations with Other Patterns

- You can treat **Visitor** as a powerful version of the **Command** pattern. Its objects can execute operations over various objects of different classes.
- You can use **Visitor** to execute an operation over an entire **Composite** tree.
- You can use **Visitor** along with **Iterator** to traverse a complex data structure and execute some operation over its elements even if they all have different classes.

Visitor: Double Dispatch Explained

The Visitor pattern solves the **double dispatch** problem:

```java
// Without Visitor - type checking needed:
for (Shape s : shapes) {
    if (s instanceof Dot)
        exportDot((Dot) s);
    else if (s instanceof Circle)
        exportCircle((Circle) s);
    // ... fragile, violates OCP
}

// With Visitor - double dispatch:
for (Shape s : shapes) {
    s.accept(visitor);
    // 1st dispatch: s.accept() resolves
    //   to the correct accept() based on s type
    // 2nd dispatch: accept() calls correct
    //   visit*() based on visitor type
}
```

The element knows its own class, so it picks the proper method on the visitor. No type checking needed.

Module K: Takeaway

The **Visitor** pattern represents an operation to be performed on elements of an object structure. It lets you define a new operation without changing the classes of the elements on which it operates, using the double dispatch technique.

Summary and Comparison

Behavioral Patterns: Summary Table

Pattern	Key Mechanism	When to Use
Chain of Responsibility	Handler chain	Multiple handlers, unknown sequence
Command	Request as object	Undo/redo, queuing, logging
Iterator	Traversal abstraction	Collection traversal
Mediator	Central coordinator	Complex inter-object communication
Memento	State snapshot	Undo, checkpointing

Behavioral Patterns: Summary Table (cont.)

Pattern	Key Mechanism	When to Use
Observer	Pub/sub notification	Event-driven systems
State	Polymorphic state	State-dependent behavior
Strategy	Interchangeable algorithms	Algorithm selection at runtime
Template Method	Algorithm skeleton	Common algorithm, varying steps
Visitor	Double dispatch	New operations on fixed structures

Sender-Receiver Connection Patterns

Four behavioral patterns address connecting senders of requests with their receivers:

Pattern	Connection Style
Chain of Responsibility	Sequential along dynamic chain
Command	Unidirectional sender-to-receiver
Mediator	Indirect via central object
Observer	Dynamic subscription-based

Each gives senders and receivers different trade-offs in terms of coupling and flexibility.

Composition vs. Inheritance Patterns

Composition-Based	Inheritance-Based
Strategy (runtime switching) State (state-aware switching) Command (request as object) Observer (dynamic subscription)	Template Method (compile-time)

Strategy vs. Template Method is a classic composition-vs-inheritance comparison.

State vs. Strategy: State allows inter-state awareness; Strategy keeps implementations independent.

Pattern Combinations

Common pattern combinations in practice:

- **Command + Memento** = Undo/redo support
- **Iterator + Visitor** = Traverse and process complex structures
- **Iterator + Memento** = Checkpointed iteration
- **Observer + Mediator** = Central event coordination
- **Composite + Iterator + Visitor** = Tree traversal with operations
- **Strategy + Factory Method** = Runtime algorithm selection

---

References

- Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley.
- Refactoring.Guru - Behavioral Design Patterns[16]
- Refactoring.Guru - Chain of Responsibility[17]
- Refactoring.Guru - Command[18]
- Refactoring.Guru - Iterator[19]
- Refactoring.Guru - Mediator[20]
- Refactoring.Guru - Memento[21]

References (cont.)

- Refactoring.Guru - Observer[22]
- Refactoring.Guru - State[23]
- Refactoring.Guru - Strategy[24]
- Refactoring.Guru - Template Method[25]
- Refactoring.Guru - Visitor[26]
- Freeman, E., Robson, E. (2020). *Head First Design Patterns.* O'Reilly Media.
- Bloch, J. (2018). *Effective Java.* Addison-Wesley.

Thank You

Questions?

CEN206 Object-Oriented Programming

Week-11: Behavioral Design Patterns

[16] https://refactoring.guru/design-patterns/behavioral-patterns
[17] https://refactoring.guru/design-patterns/chain-of-responsibility
[18] https://refactoring.guru/design-patterns/command
[19] https://refactoring.guru/design-patterns/iterator
[20] https://refactoring.guru/design-patterns/mediator
[21] https://refactoring.guru/design-patterns/memento
[22] https://refactoring.guru/design-patterns/observer
[23] https://refactoring.guru/design-patterns/state
[24] https://refactoring.guru/design-patterns/strategy
[25] https://refactoring.guru/design-patterns/template-method
[26] https://refactoring.guru/design-patterns/visitor