

Patterns in Software Engineering

02 Design Patterns I

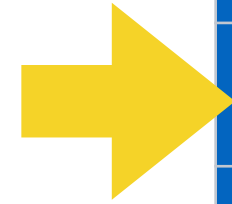
Stephan Krusche



7 November 2022
Technical University of Munich



Course schedule



#	Date	Subject
	17.10.22	No lecture, repetition week (self-study)
1	24.10.22	Introduction
	31.10.22	No lecture, repetition week (self-study)
2	07.11.22	Design Patterns I
3	14.11.22	Design Patterns II
4	21.11.22	Architectural Patterns I
5	28.11.22	Architectural Patterns II
6	05.12.22	Antipatterns I
7	12.12.22	Antipatterns II
	19.12.22	No lecture
8	09.01.23	Testing Patterns I
9	16.01.23	Testing Patterns II
10	23.01.23	Microservice Patterns I
11	30.01.23	Microservice Patterns II
12	08.02.21	Course Review

Roadmap of the lecture



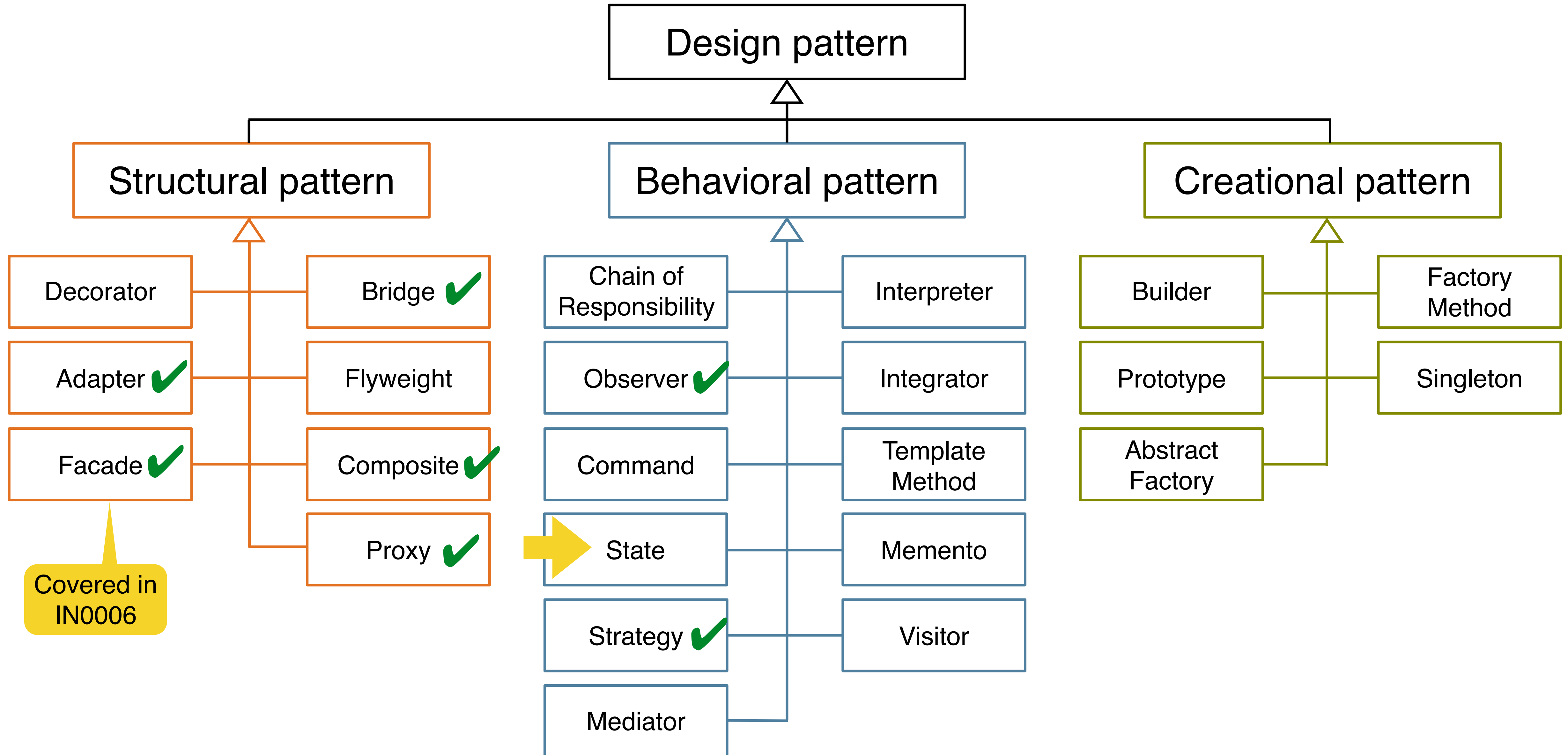
- **Context and assumptions**
 - You have understood the basic concepts of patterns
 - You have made first experiences with design patterns
- **Learning goals: at the end of this lecture you are able to**
 - Differentiate between the different design patterns
 - Choose the right design pattern for a specific problem
 - Model the application of the design pattern (object design)

Outline

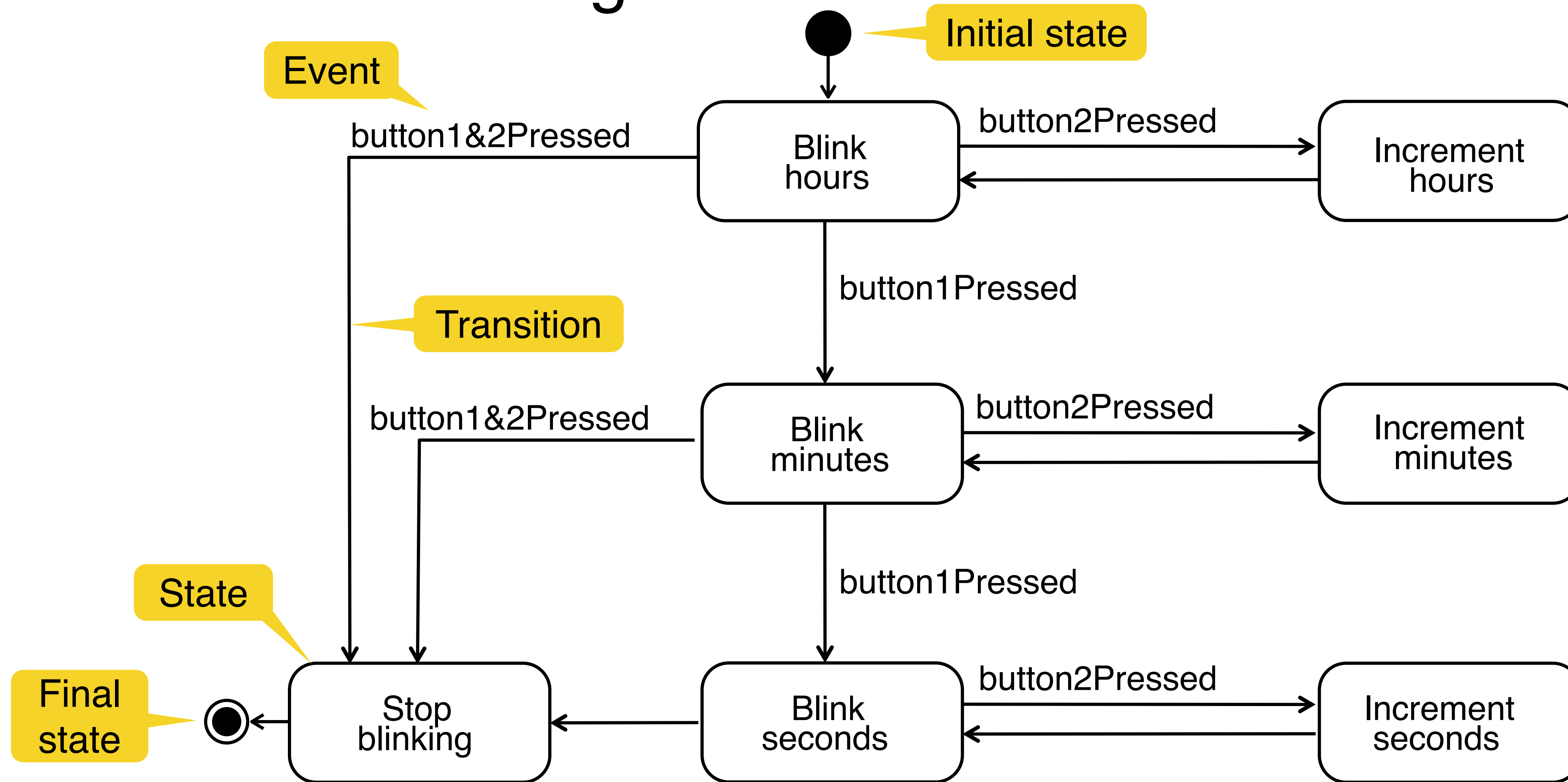
State pattern

- Template method pattern
- Command pattern
- Mediator pattern

Design pattern overview



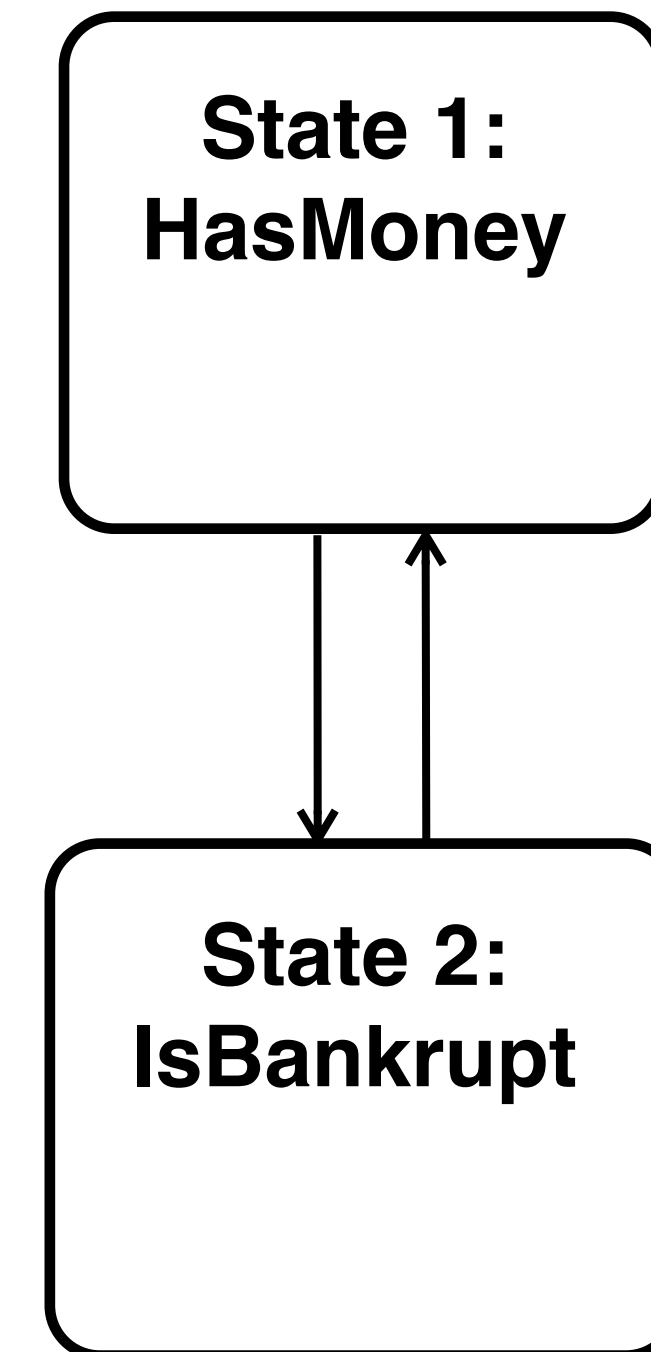
UML state chart diagrams



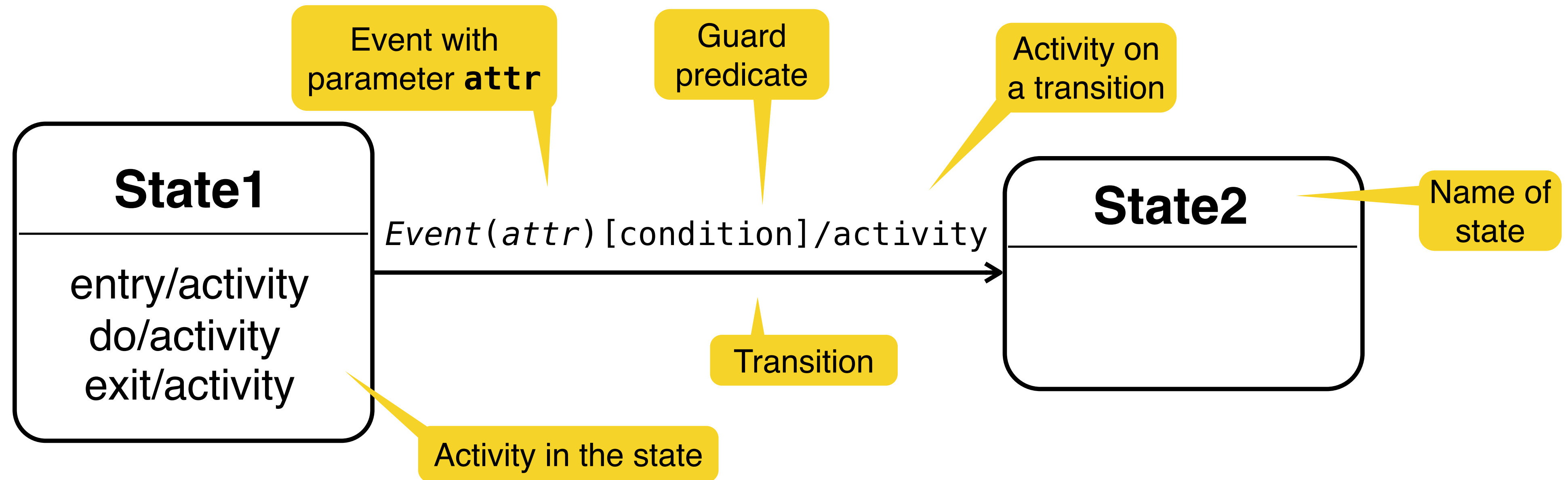
State charts represent the dynamic behavior of **one single object**
(this state chart diagram models a [watch](#) with 2 buttons)

State

- An abstraction of the attributes of a class (aggregation of several attributes of a class)
- An equivalence class of all those attribute values that do not need to be distinguished
- Has a certain duration
- **Example:** state of a bank
 - State 1: bank has money
 - State 2: bank is bankrupt

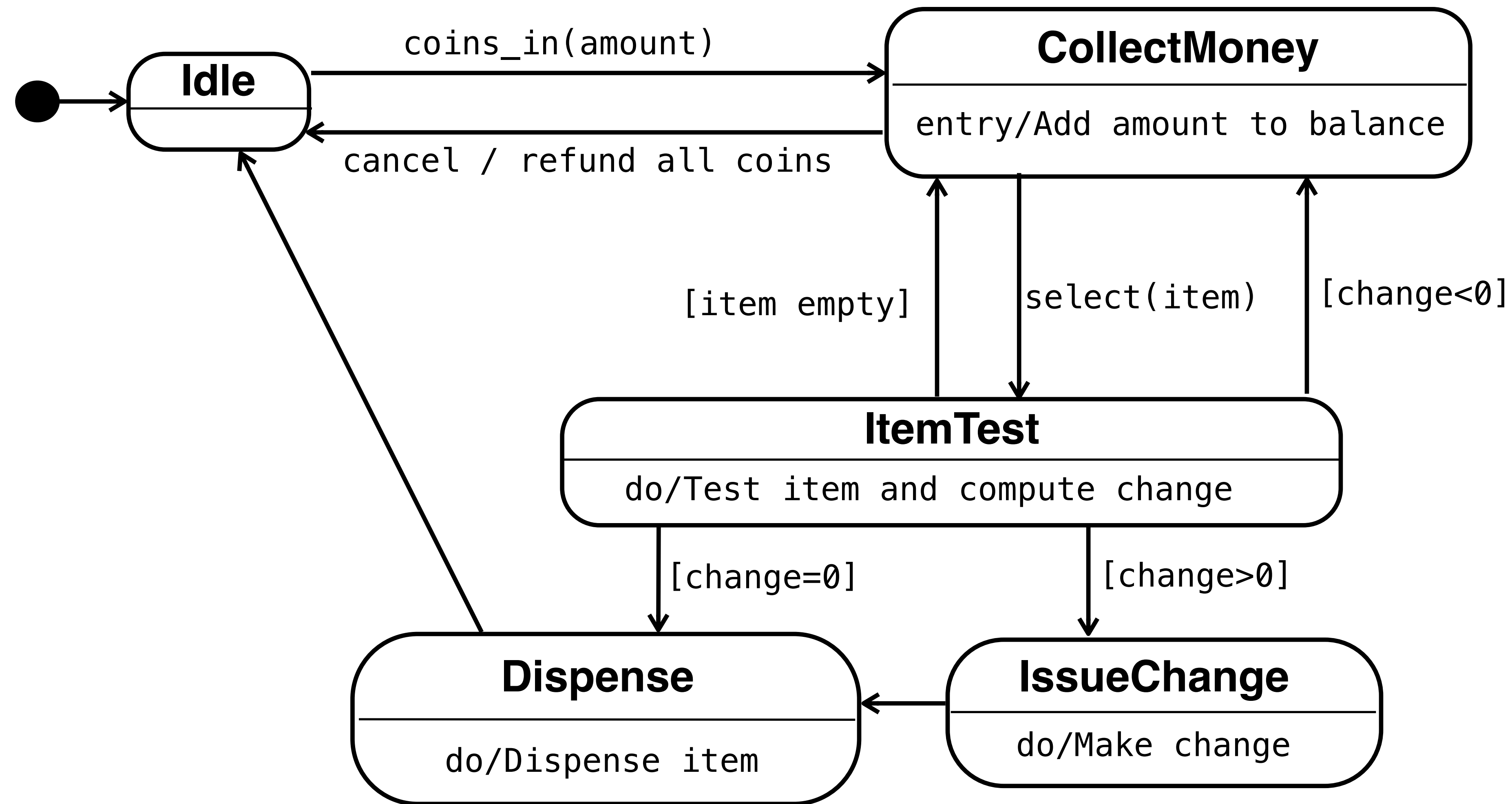


UML state chart diagram notation





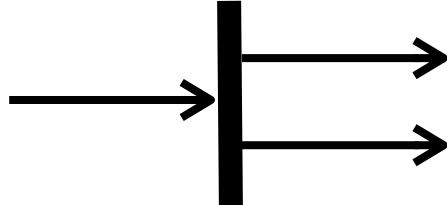
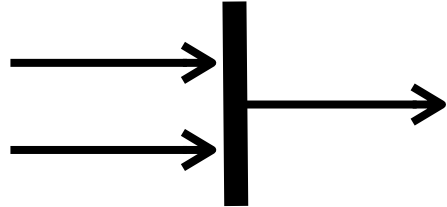

- Modeling conventions
 - Events are written in italics or with a normal type font
 - Guard predicates are enclosed in brackets []
 - Activities on a transition are always prefixed with a slash /

Example: state chart diagram of a vending machine

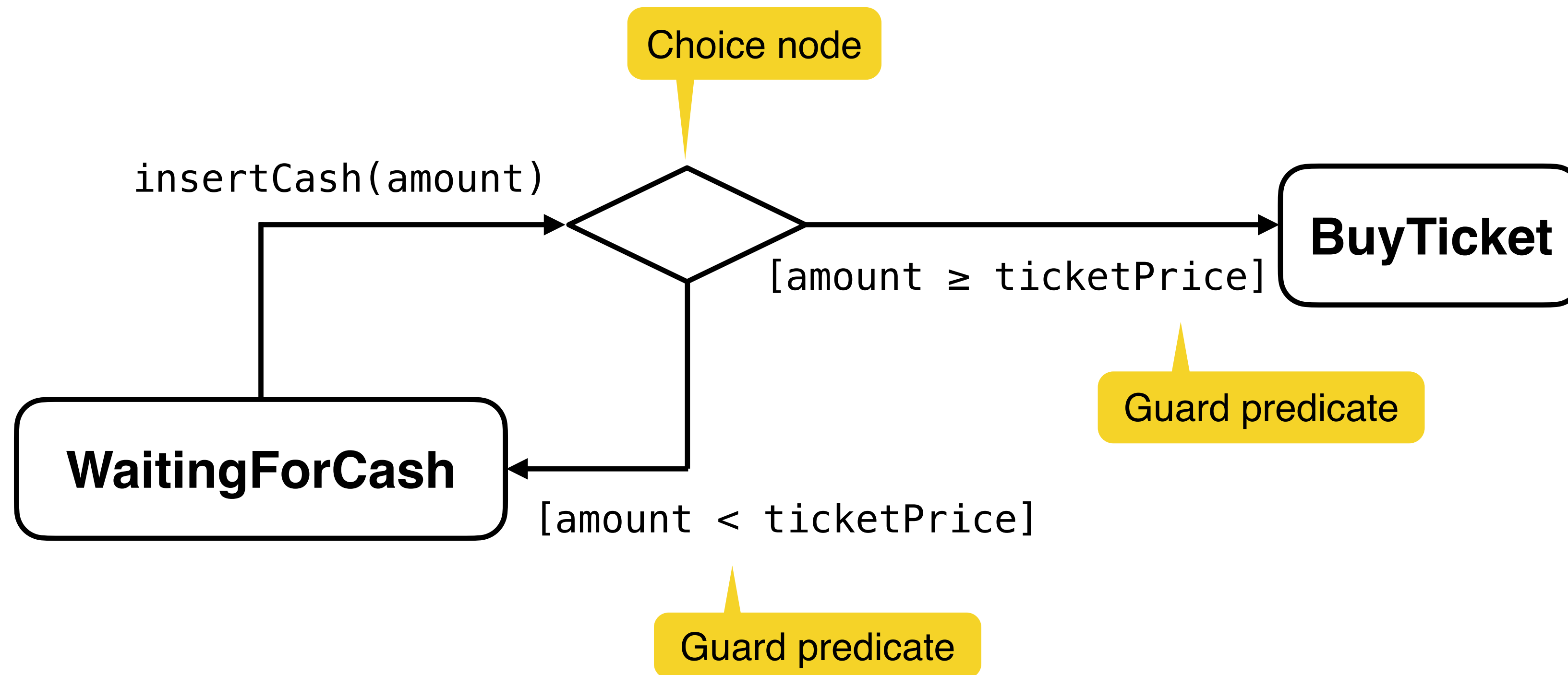


Control nodes in state chart diagrams

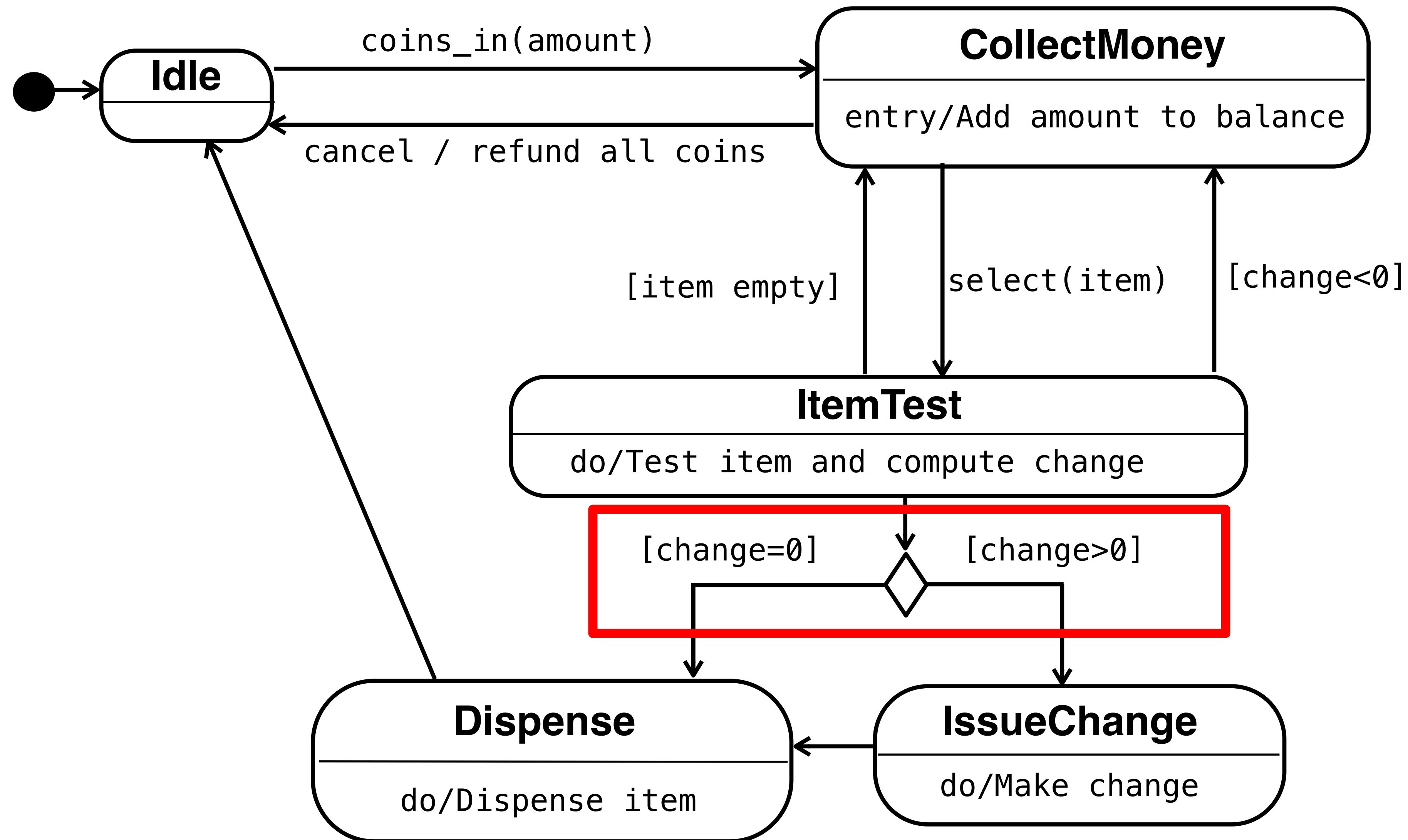
Special node that coordinates the transition between other nodes

Name	Graphical Icon
Initial node	
Final node	
Fork node	
Join node	
Choice node	

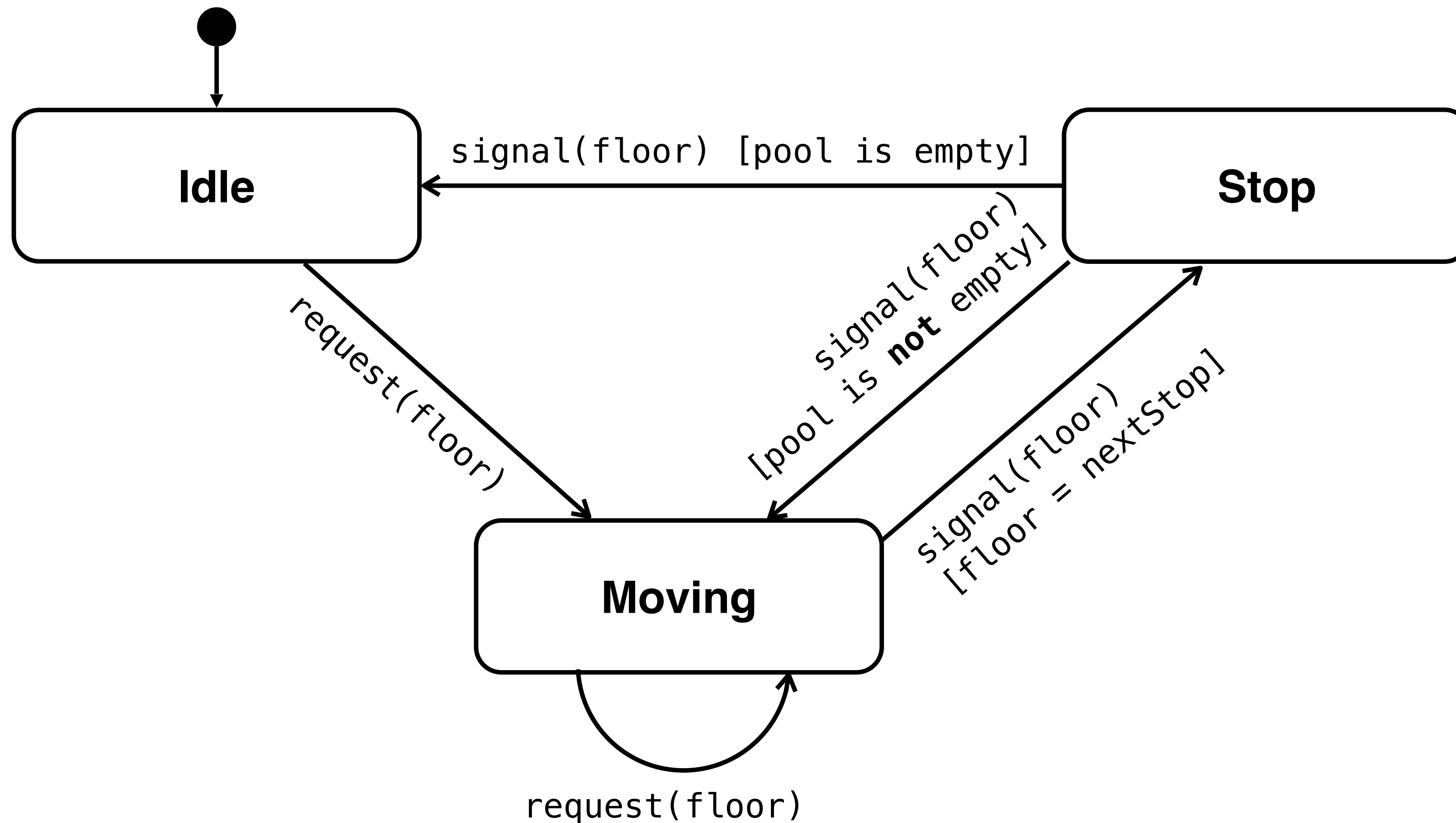
Example: choice node



Example: vending machine with a choice node



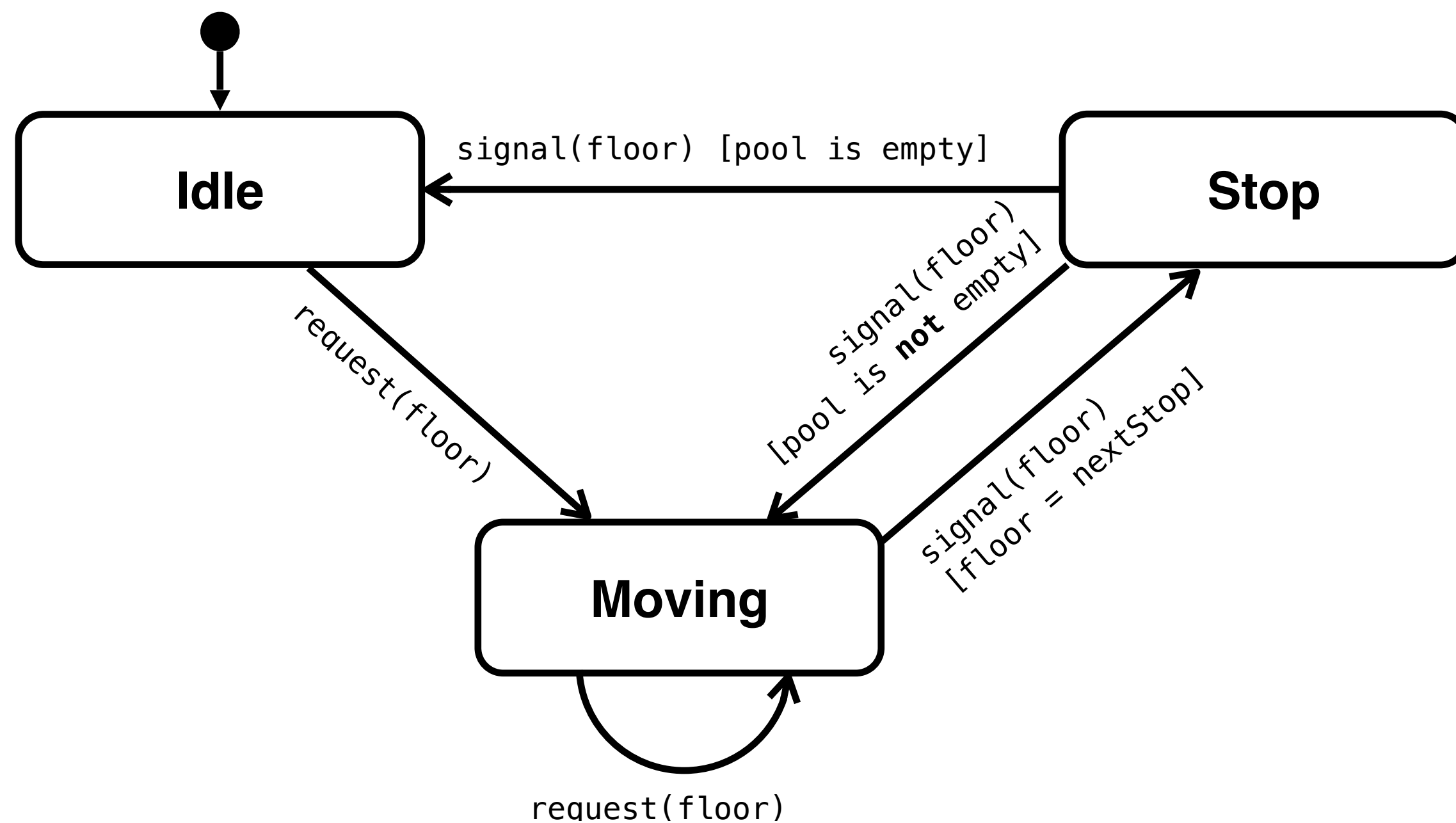
Example: state diagram for an elevator



Mapping UML state chart diagrams to Java source code

1. Write a **public method** for each event
2. Define a variable **state**
3. Write a **switch statement** with cases for each of the states in the UML state diagram
4. Write an **if statement for each guard predicate** on the transitions to check if it has become true
5. Update the **state** variable accordingly

Implementation of the **request** event



```
public void request(int floor) throws ... {
    switch(state) {
        case IDLE:
            state = MOVING;
            break;
        case MOVING:
        case STOP:
            break;
        default: throw new UnexpectedStateException();
    }
}
```

```
public void signal(int floor) throws ... {
    switch(state) {
        case IDLE:
            break;
        case MOVING:
            if (floor == getNextStop()) {
                state = STOP;
            }
            break;
        case STOP:
            if (pool.isEmpty()) {
                state = IDLE;
            } else {
                nextStop = pool.choose();
                state = MOVING;
            }
            break;
        default: throw new UnexpectedStateException();
    }
}
```

Implementation of the **request** event (2)

```
public void signal(int floor) throws ... {  
    switch(state) {  
        case IDLE:  
            break;  
        case MOVING:  
            if (floor == getNextStop()) {  
                state = STOP;  
            }  
            break;  
        case STOP:  
            if (pool.isEmpty()) {  
                state = IDLE;  
            } else {  
                nextStop = pool.choose();  
                state = MOVING;  
            }  
            break;  
        default: throw new UnexpectedStateException();  
    }  
}
```

Introduce **state** variable for current state

Check condition of transition

Transition

Perform action

Another transition

Illegal state or message

Problem

- Functional approach
- Each state must be covered in the switch statement
- If there are many events
 - All logic resides in one switch statement
 - New states need to be handled in different places
 - Developers might forget to handle states

➔ **A more object oriented approach is necessary**

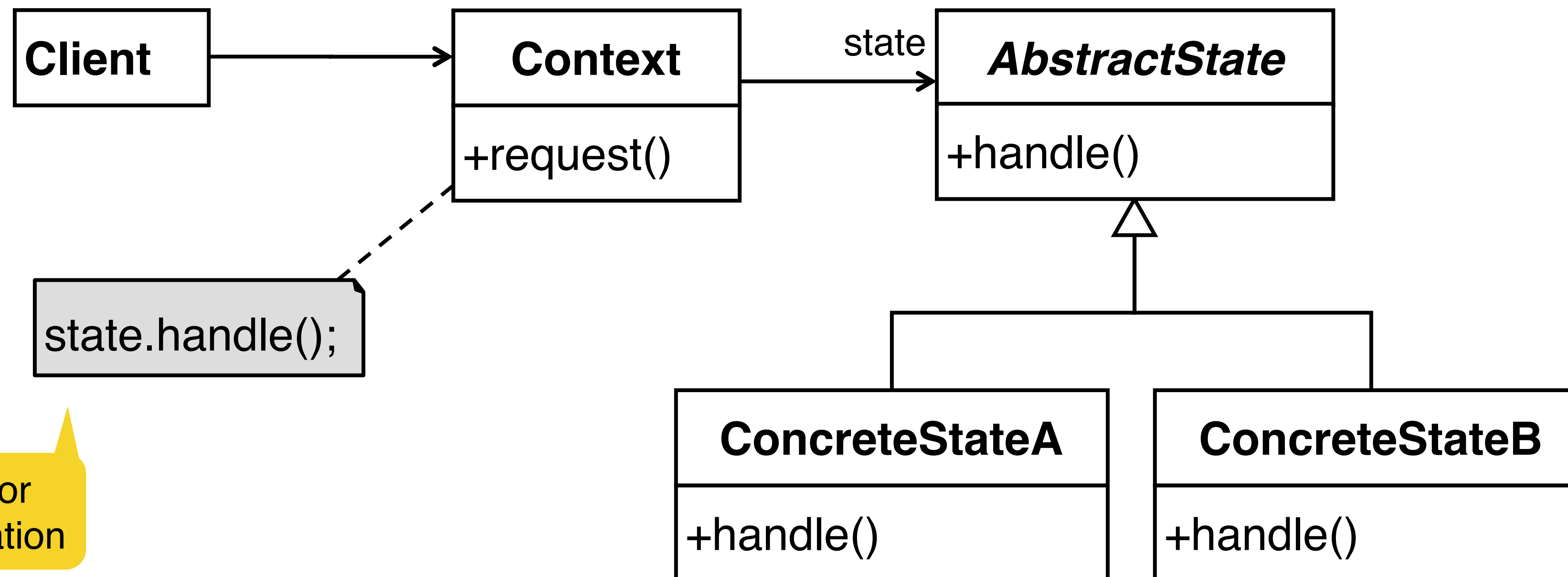
Please note: this is a simplified version

State pattern

- **Motivation**

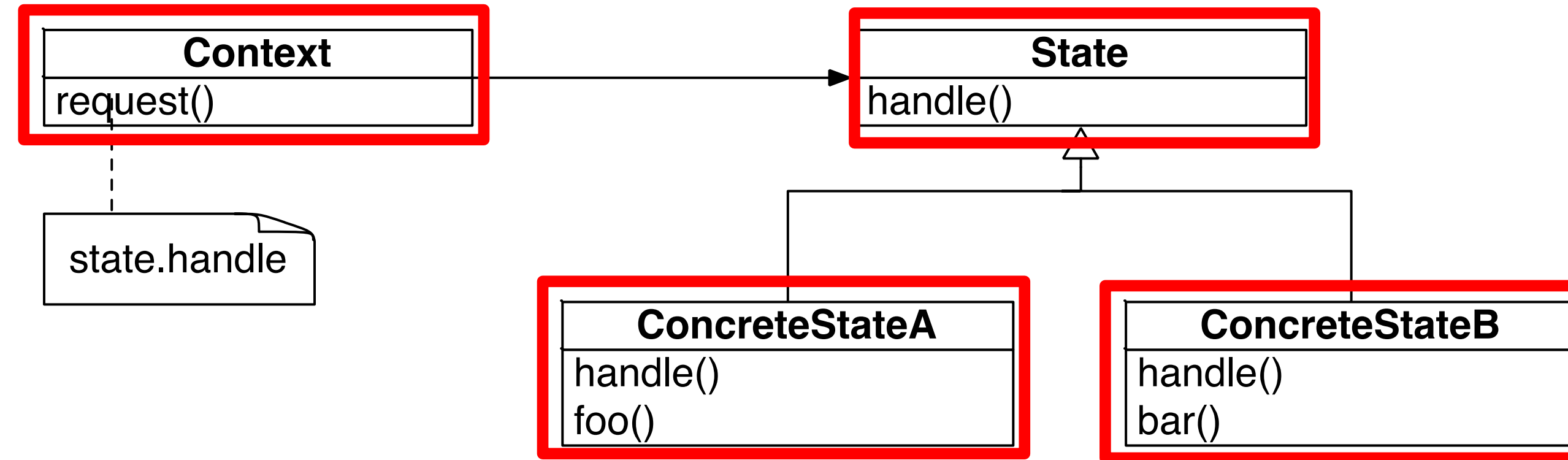
- There is an object with interesting dynamic behavior that can be in several states with different behavior in each of these states
- The object changes its behavior when its state changes at runtime
- The source code contains large, multipart conditional statements that depend on the object's state

- **Structure**



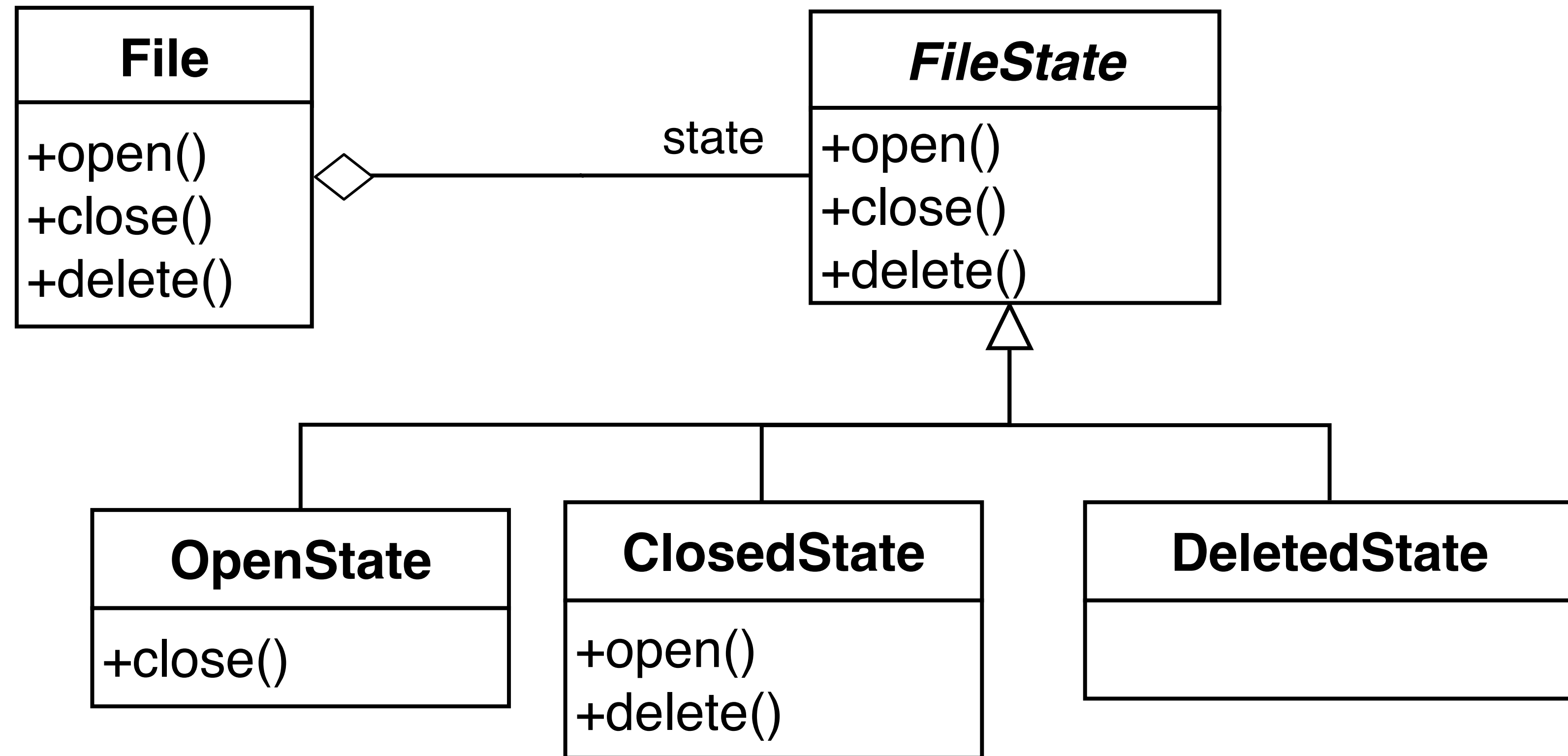
A note in UML for additional information

State pattern



- **Context** can have a number of internal states —> presents a single interface to the outside world
- **State** (interface or abstract class) defines a common interface for all concrete states
- **State** subclasses all implement the same interface so they are interchangeable
- Usually, the number of concrete states is equal to the number of internal states
- **ConcreteStates** handle requests from the **Context** and realize the state-specific behavior
- Each **ConcreteState** provides its own implementations for a request
- When the **Context** changes state, its behavior will change as well

Example: files



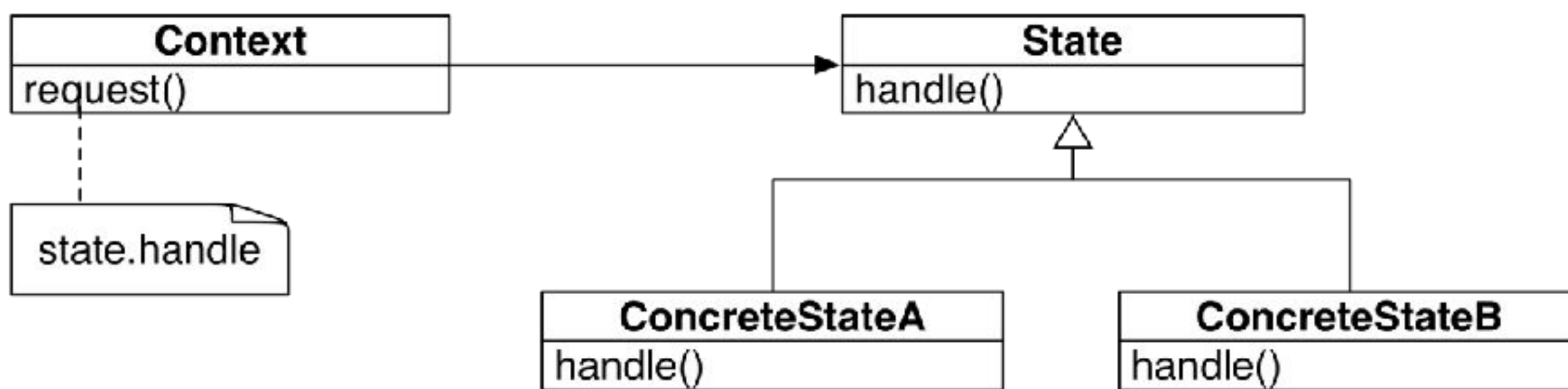
State pattern: properties



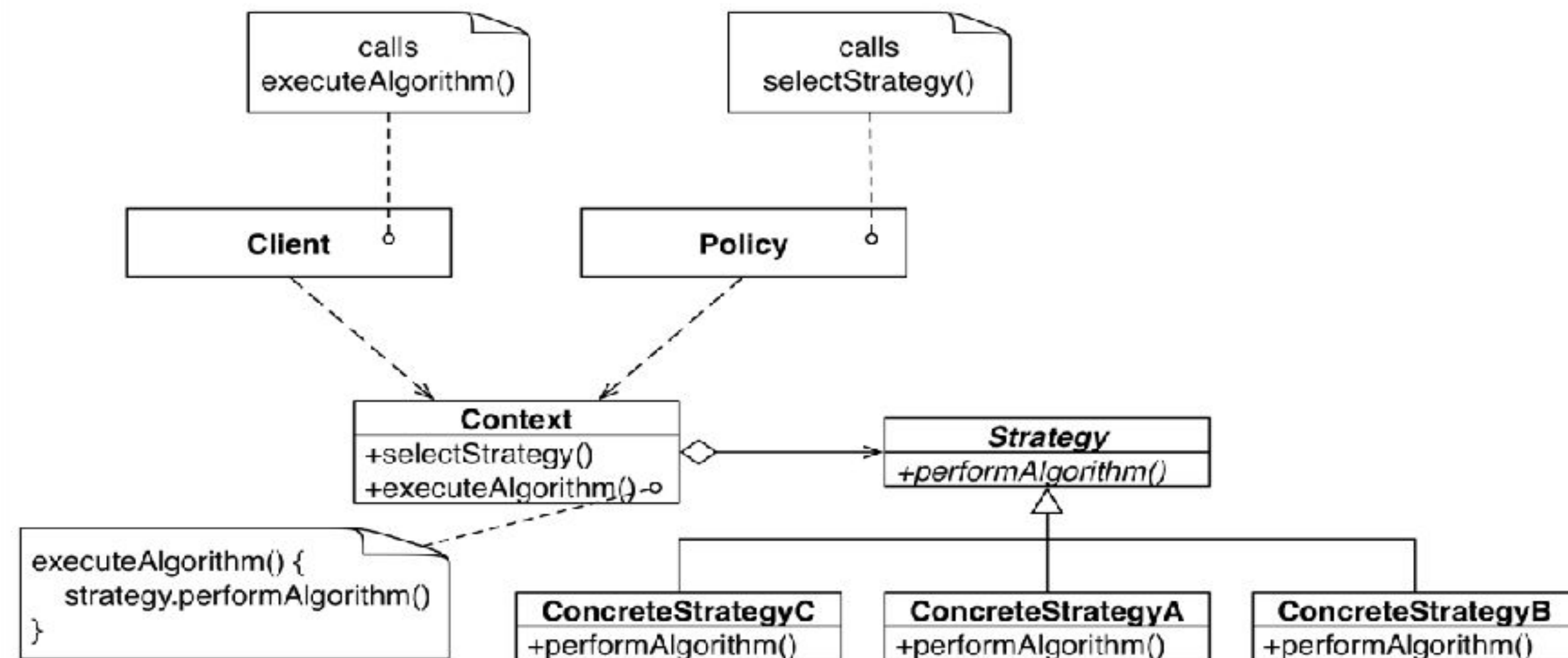
- Useful for objects which change their **state at runtime**
- Uses **polymorphism** to define different behaviors for different states of an object
- The selection of the subclass depends on the state of the object
 - Comparison with **bridge pattern**: the selection of the subclass is done at system initialization time
 - Comparison with **strategy pattern**: the selection of the subclass depends on an external policy

Strategy vs. state pattern

State pattern: helps objects to control their behavior by changing their internal state



Strategy pattern: algorithms can be interchanged at runtime



State pattern



- **Benefits**

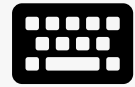
- Localizes state specific behavior
- Extensibility and flexibility: easy to add more states for additional behavior
- Avoids cluttered if-else or switch-case statements
- Makes state transitions explicit

- **Implementation questions**

- Is the state transition defined in the context class or in the concrete states?
- When are the states created and destroyed?



L02E01 State Pattern



Start exercise

Easy

Not started yet.

Due date in 7 days



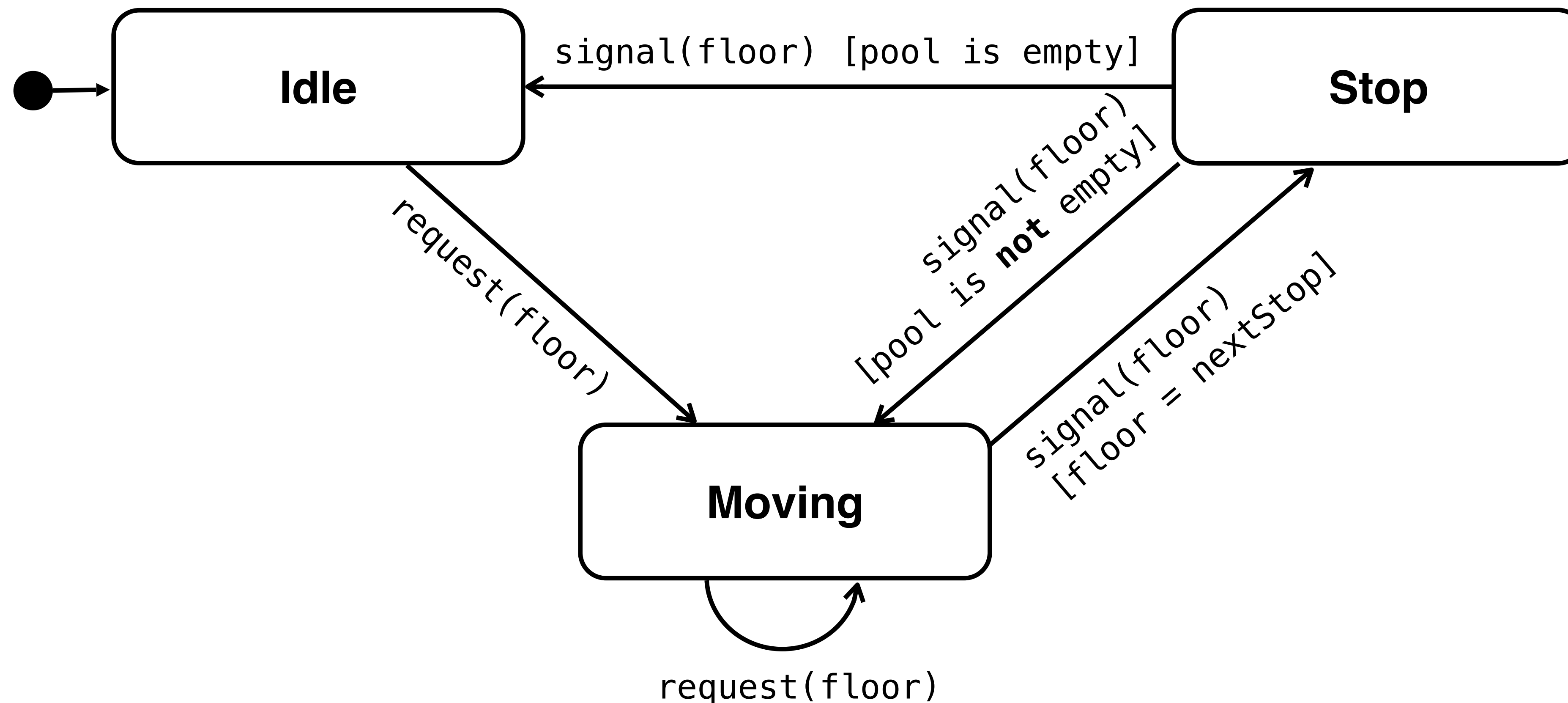
15 min



6 pts



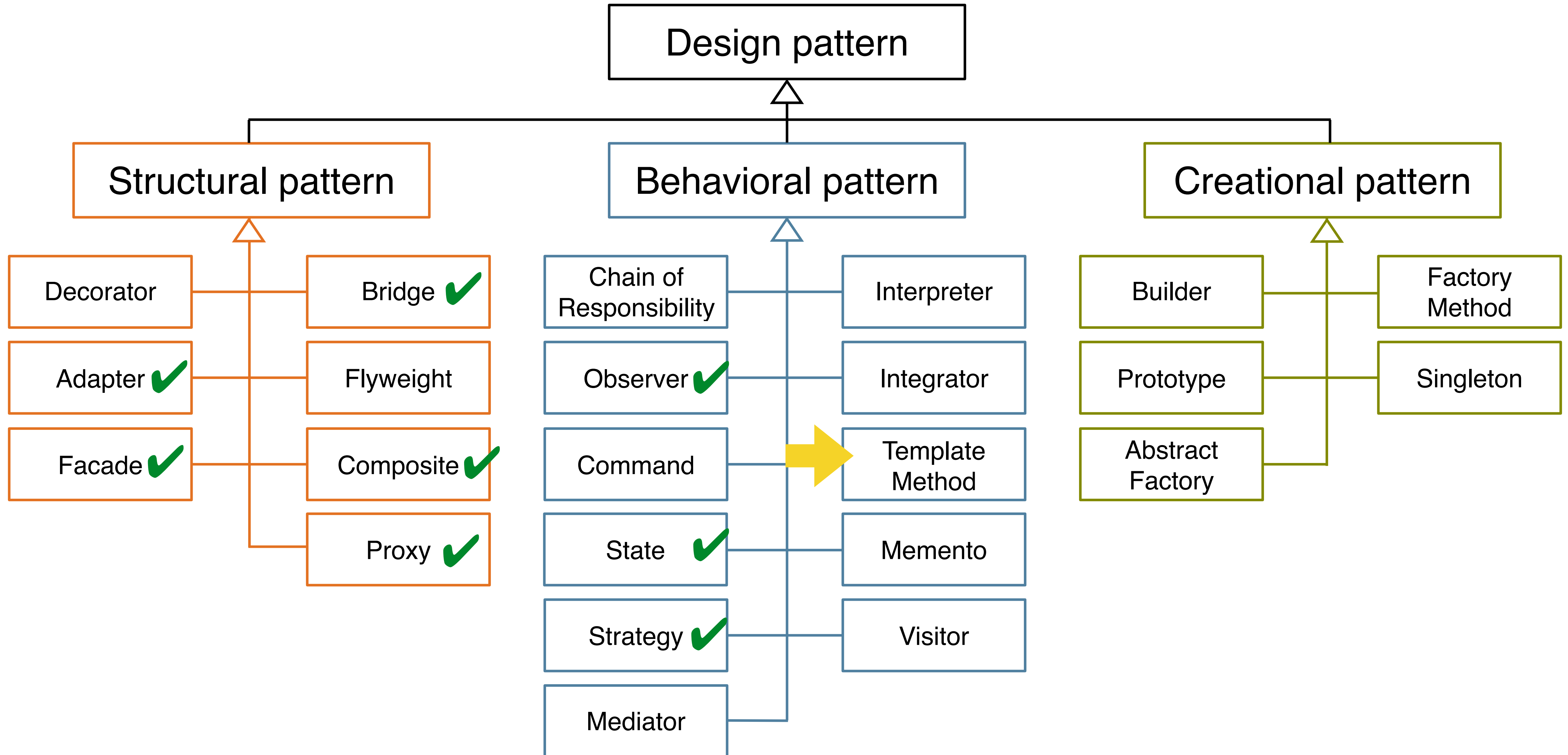
- Problem statement: apply the state pattern for the elevator example



Outline

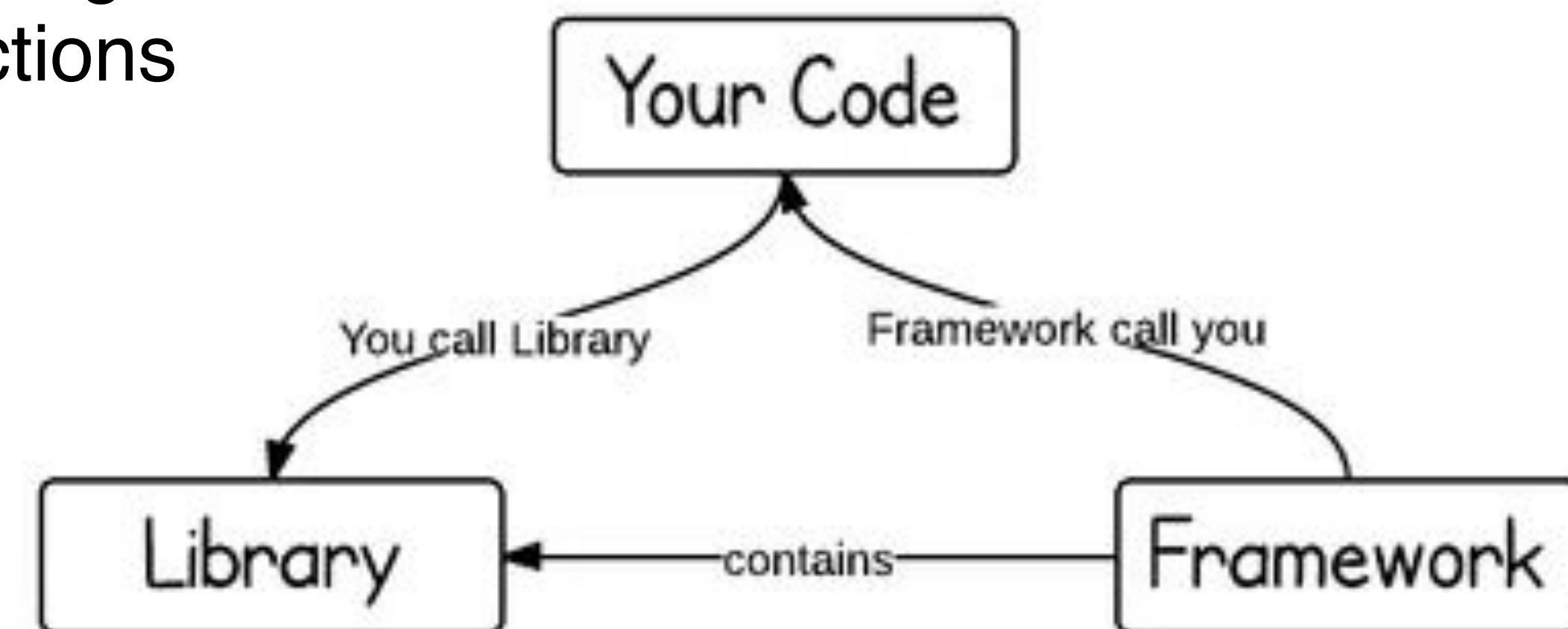
- State pattern
- ➔ Template method pattern
- Command pattern
- Mediator pattern

Design pattern overview



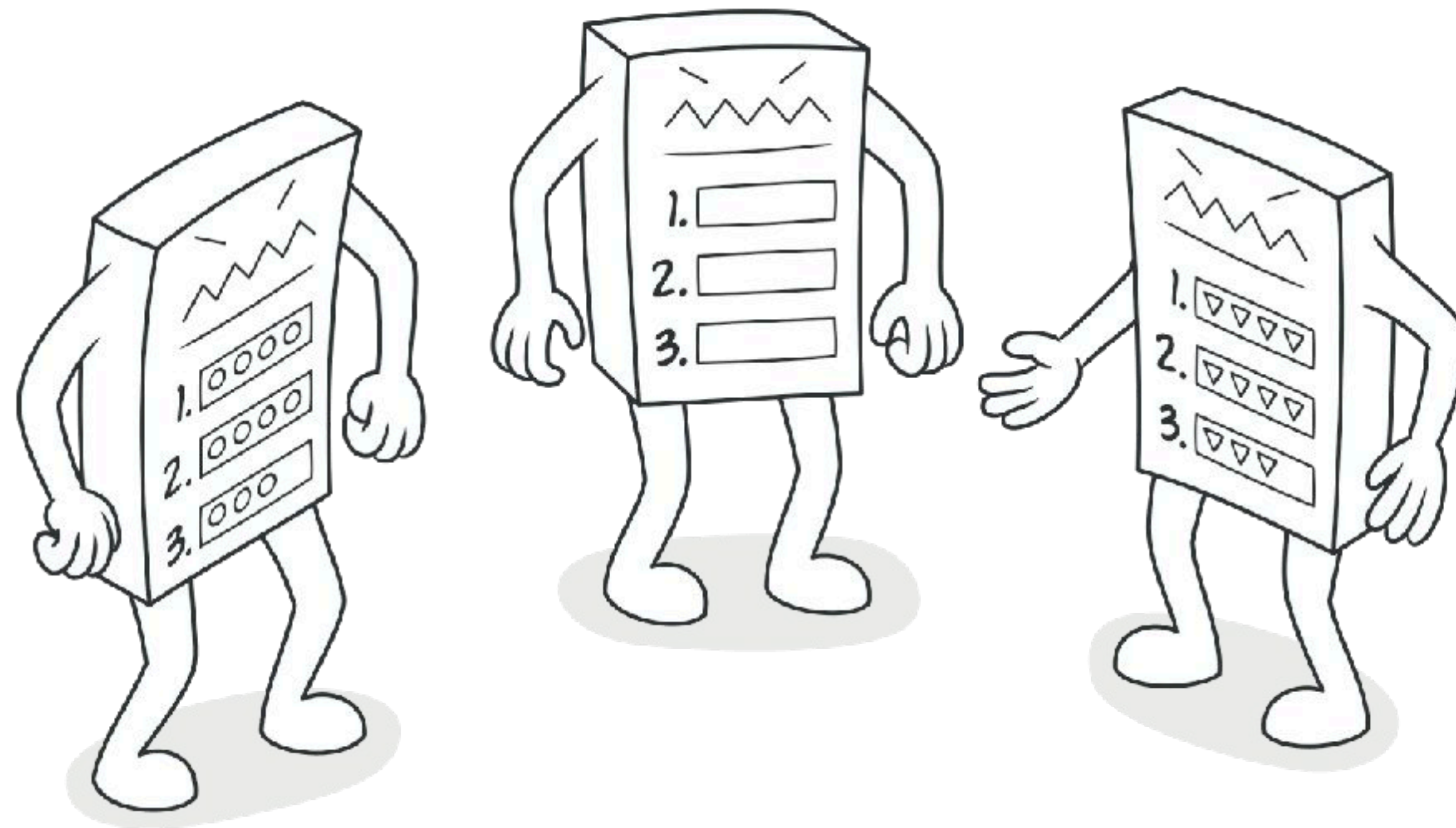
Framework vs. library

- A library is just a collection of class definitions with the purpose of code reuse
 - Classes and methods normally define specific operations in a domain specific area
- A framework is normally more complex
 - Includes control flow
 - Includes predefined white spots that application developers should implement
 - Defines a skeleton where the application implements its own features to fill this skeleton
 - Calls application code when appropriately
 - **Benefit:** developers do not need to worry if a design is good or not, but just about implementing domain specific functions

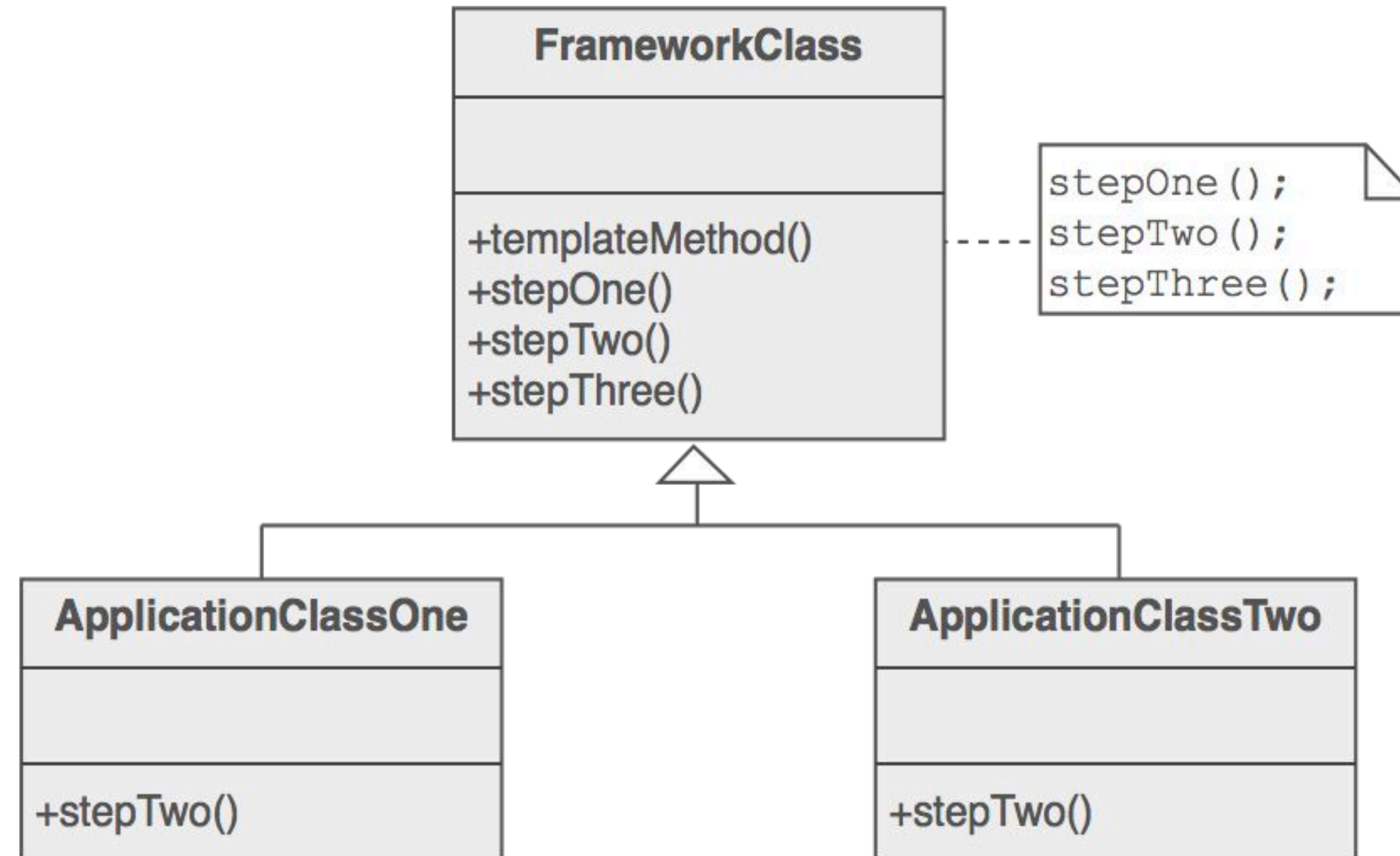


Problem

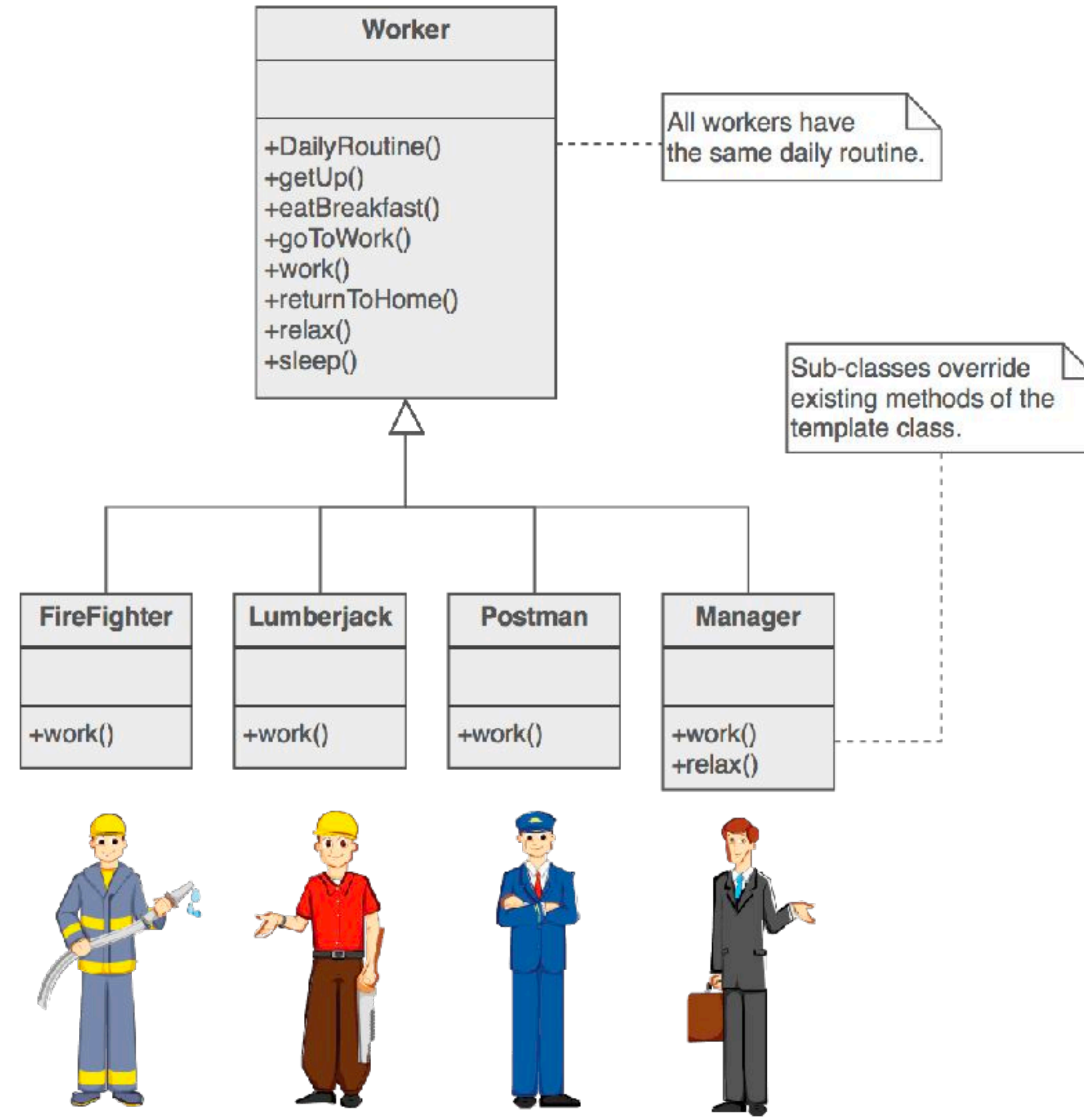
- Two different components (e.g. classes) have **significant similarities**, but demonstrate **no reuse** of a common interface or implementation
- If a change common to both components becomes necessary, **duplicate effort** is necessary



Template method pattern

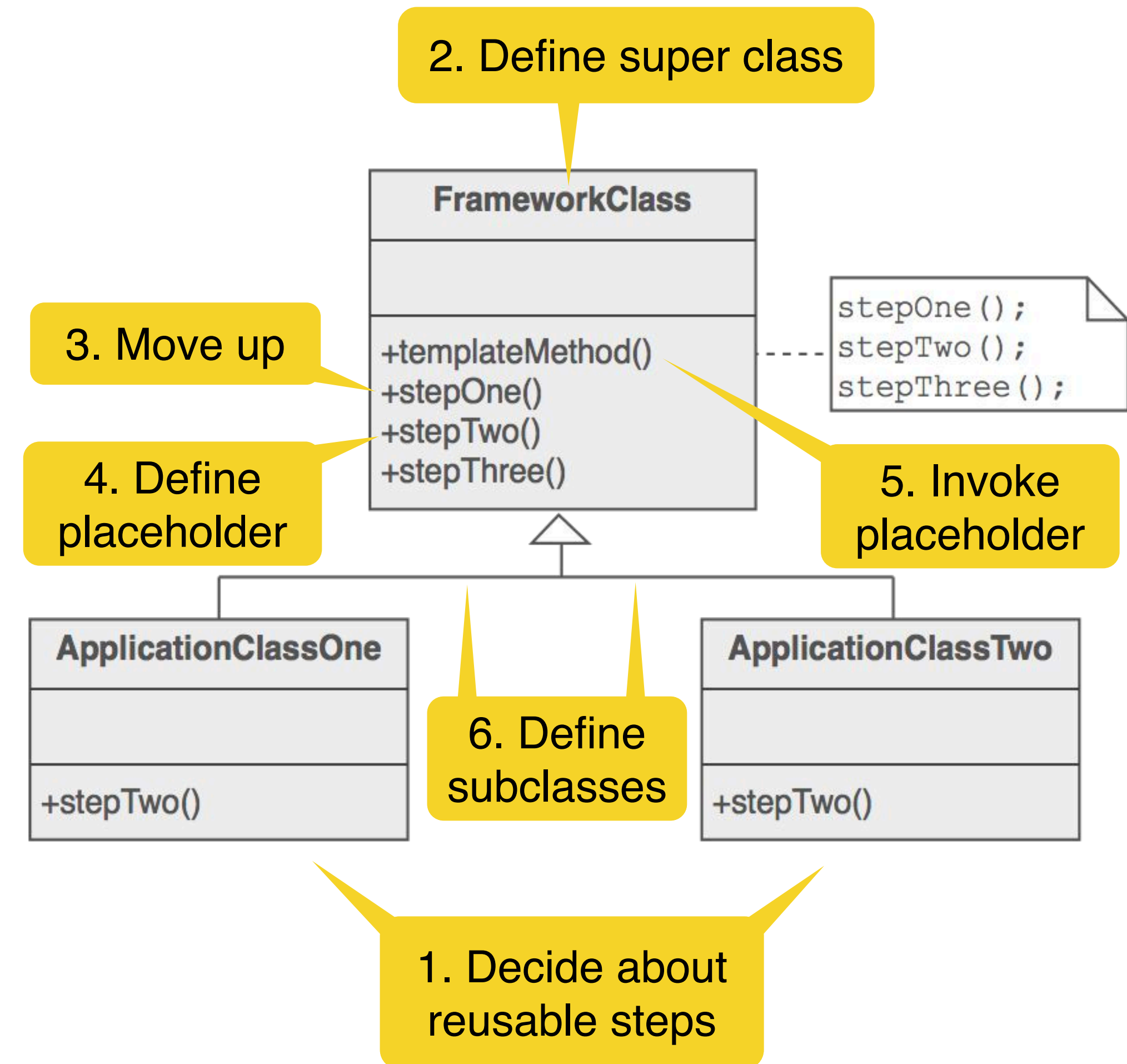


Example



Check list to introduce template method to existing code

1. Decide which steps are standard and which steps are concrete to each of the current classes
2. Define a new abstract **super class** to host the "don't call us, we'll call you" framework
3. Move the body of the algorithm and the definition of all standard steps to the new **super class**
4. Define **placeholder** methods in the **super class** for each step that requires different implementations
5. Invoke the **placeholder** method(s) from the **template** method
6. Define all existing classes as subclasses of the new abstract **super class**

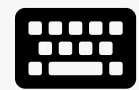


Strategy vs. template method pattern

- Both are similar except the granularity
- **Strategy** uses **delegation** to vary the entire algorithm
- **Template** method uses **inheritance** to vary parts of an algorithm
- **Strategy** modifies the logic of **individual** objects
- **Template method** modifies the logic of an **entire class**



L02E02 Template Method Pattern



Start exercise

Easy

Not started yet.

Due date in 7 days



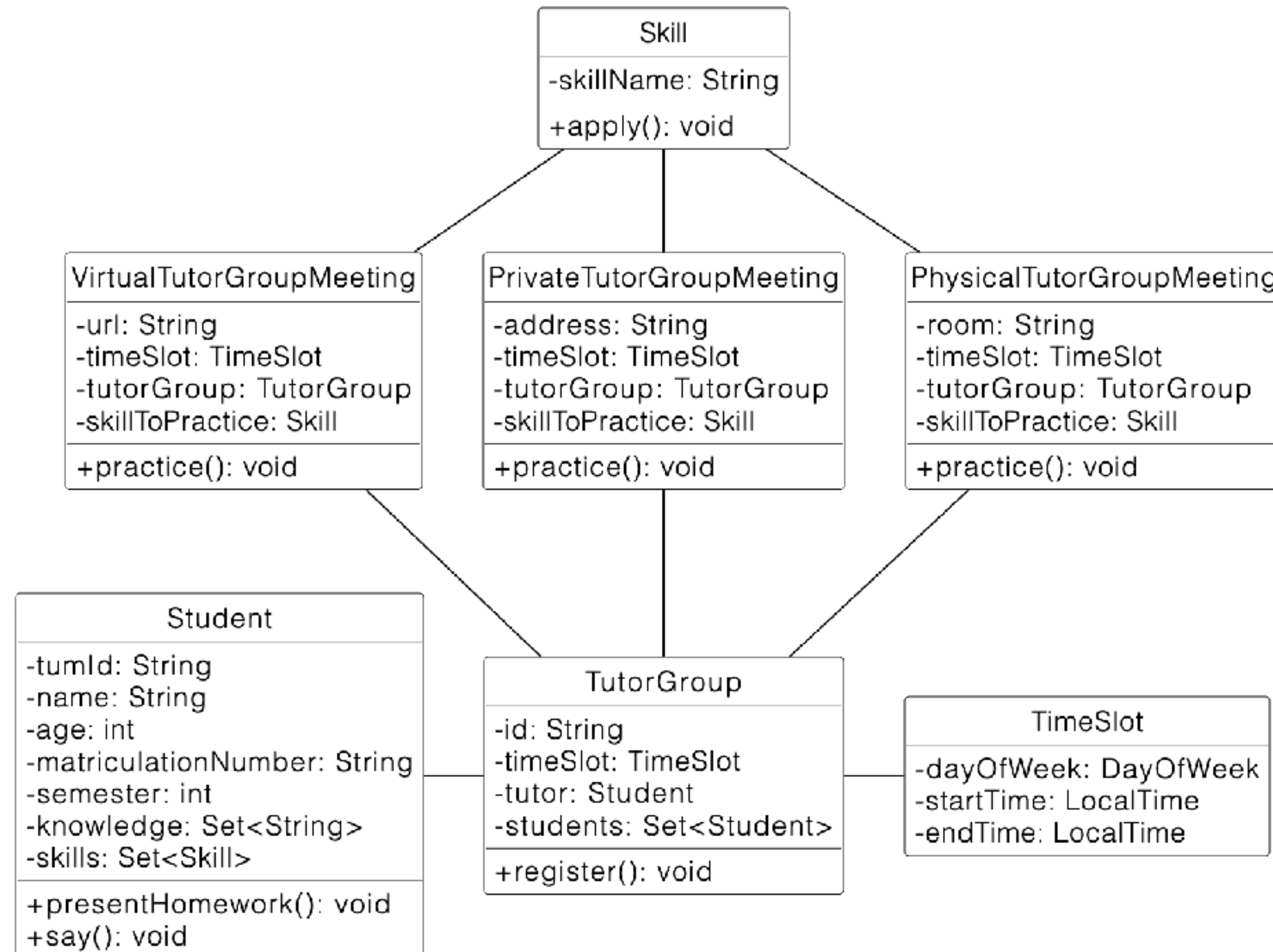
20 min



7 pts



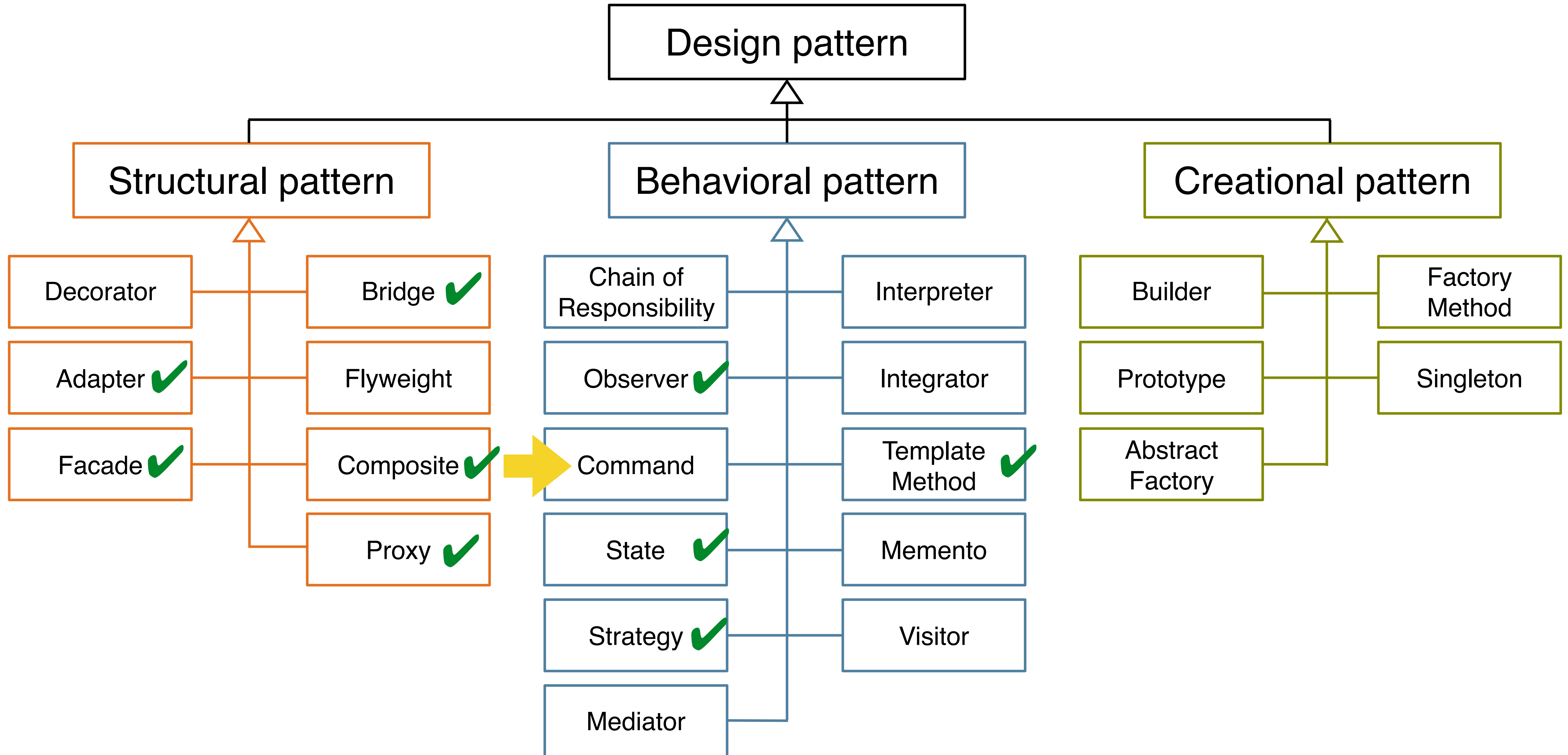
- Problem statement: refactor a tutor group system



Outline

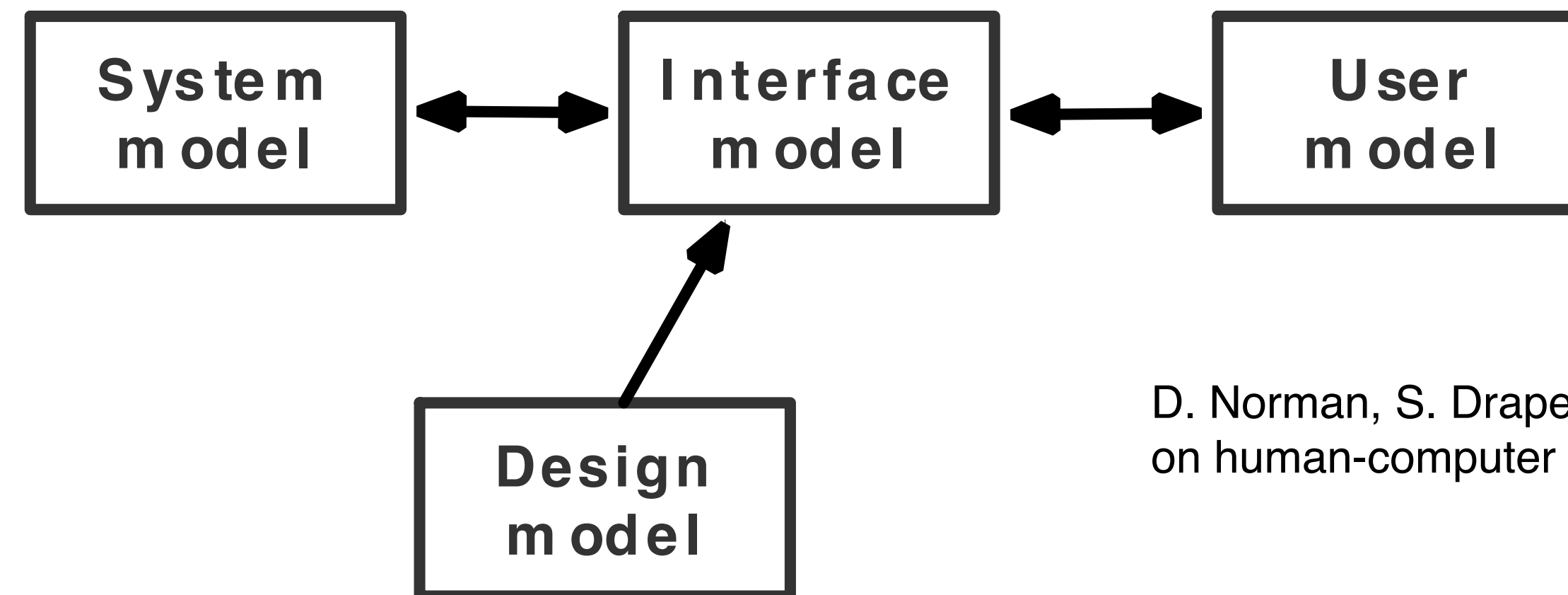
- State pattern
- Template method pattern
- ➔ Command pattern
- Mediator pattern

Design pattern overview



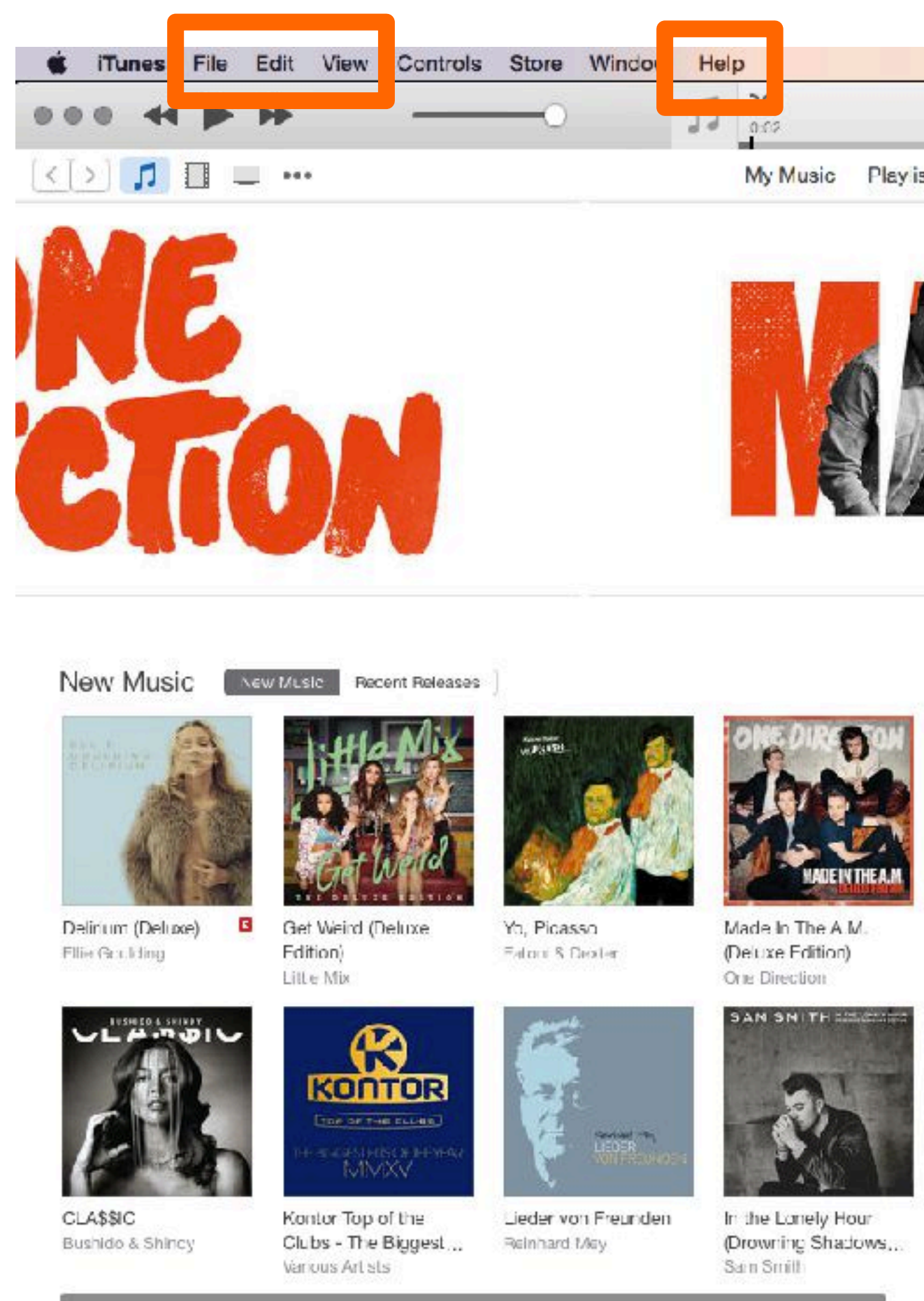
User interface design goals

- **Main issue:** how to design a **good** user interface (UI)?
- **Good** UI design goals (among others)
 - Provide a uniform UI across applications
 - Provide a uniform UI within a platform
 - Develop a good understanding of the user model
 - Provide a natural mapping: map the user model to the design model (without confusing the users)

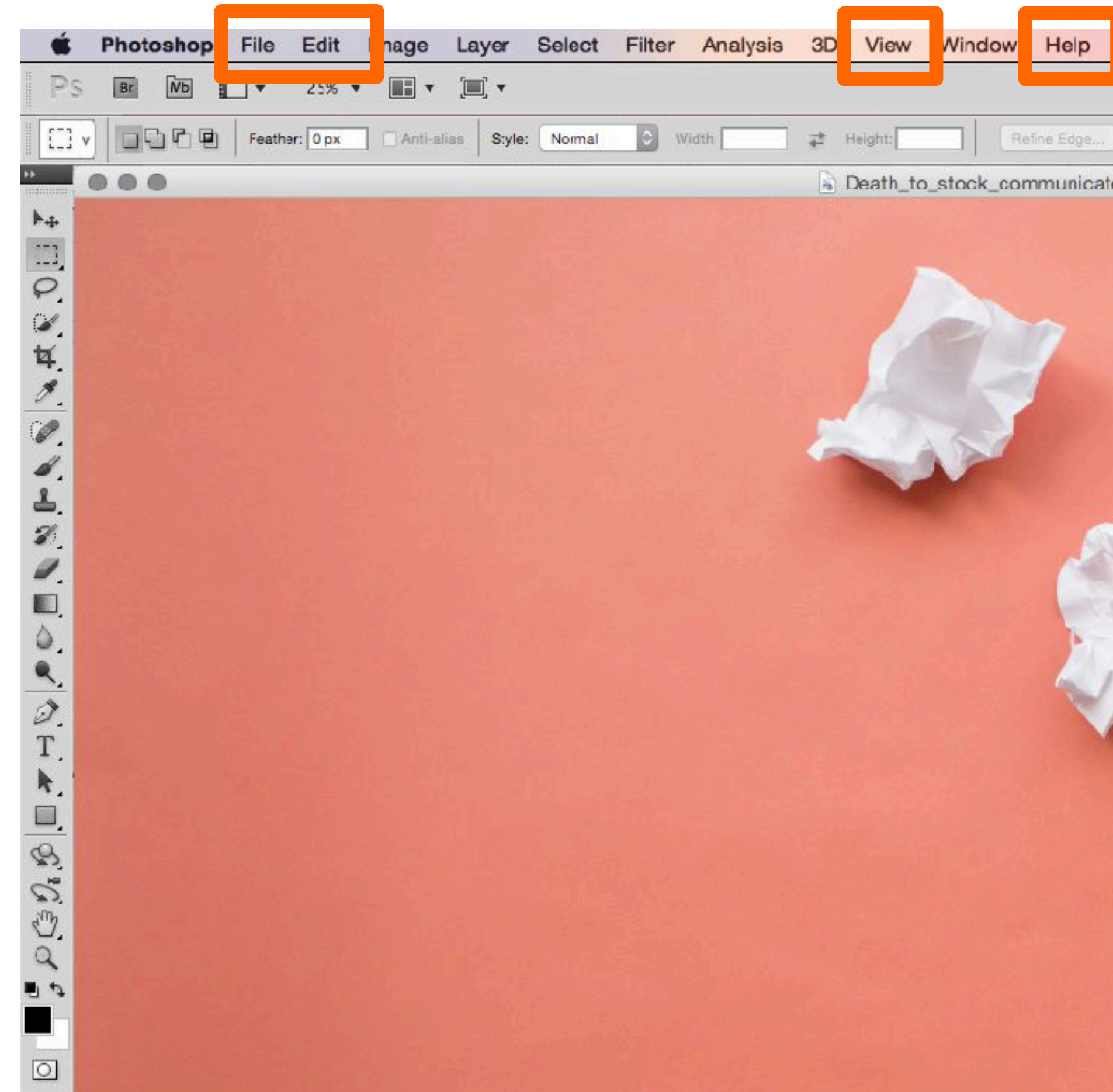


D. Norman, S. Draper. User centered system design; new perspectives on human-computer interaction. L. Erlbaum Associates Inc., 1986.

Uniform user interface across applications (example)

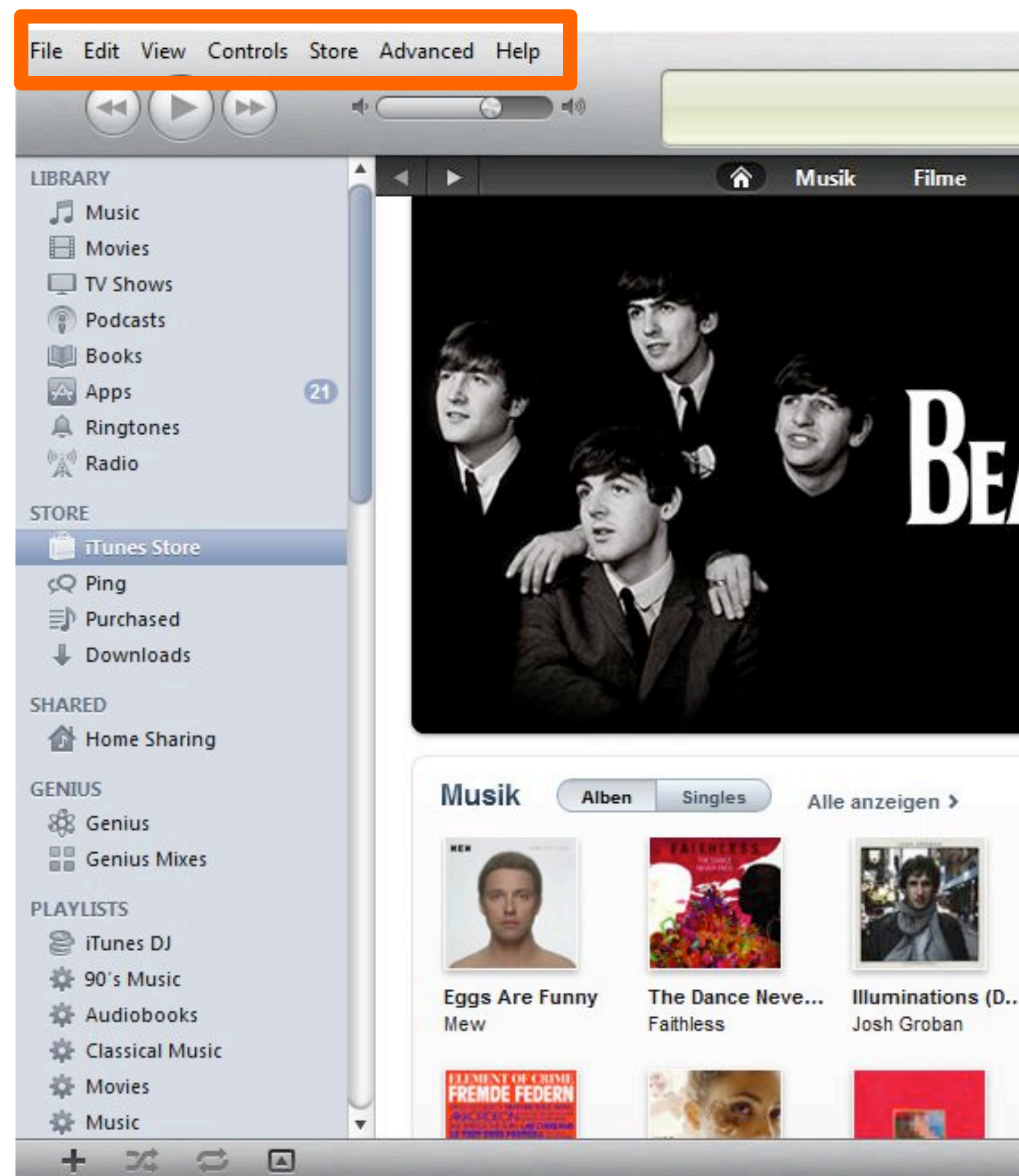


iTunes

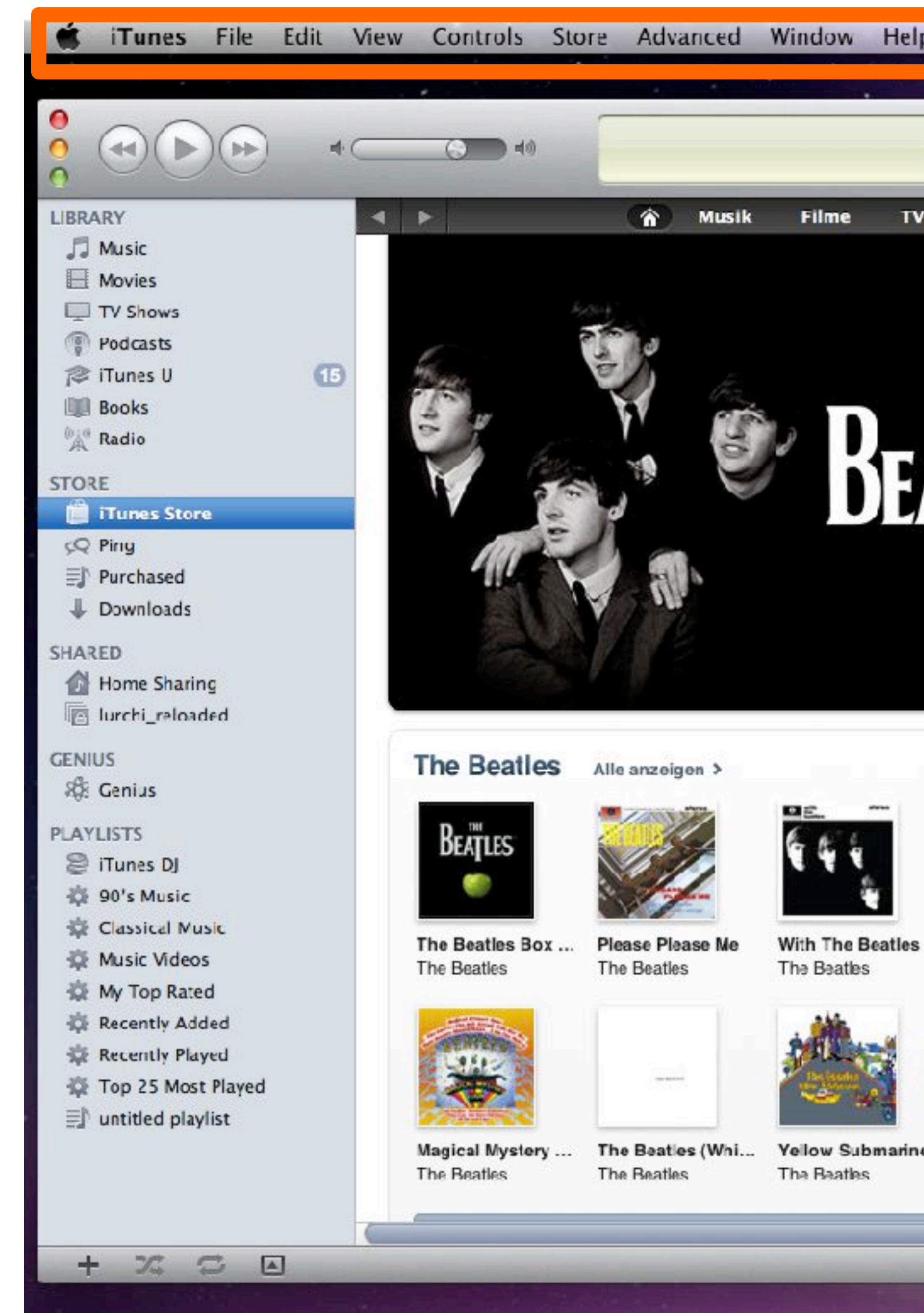


Photoshop

Uniform user interface across applications (example)



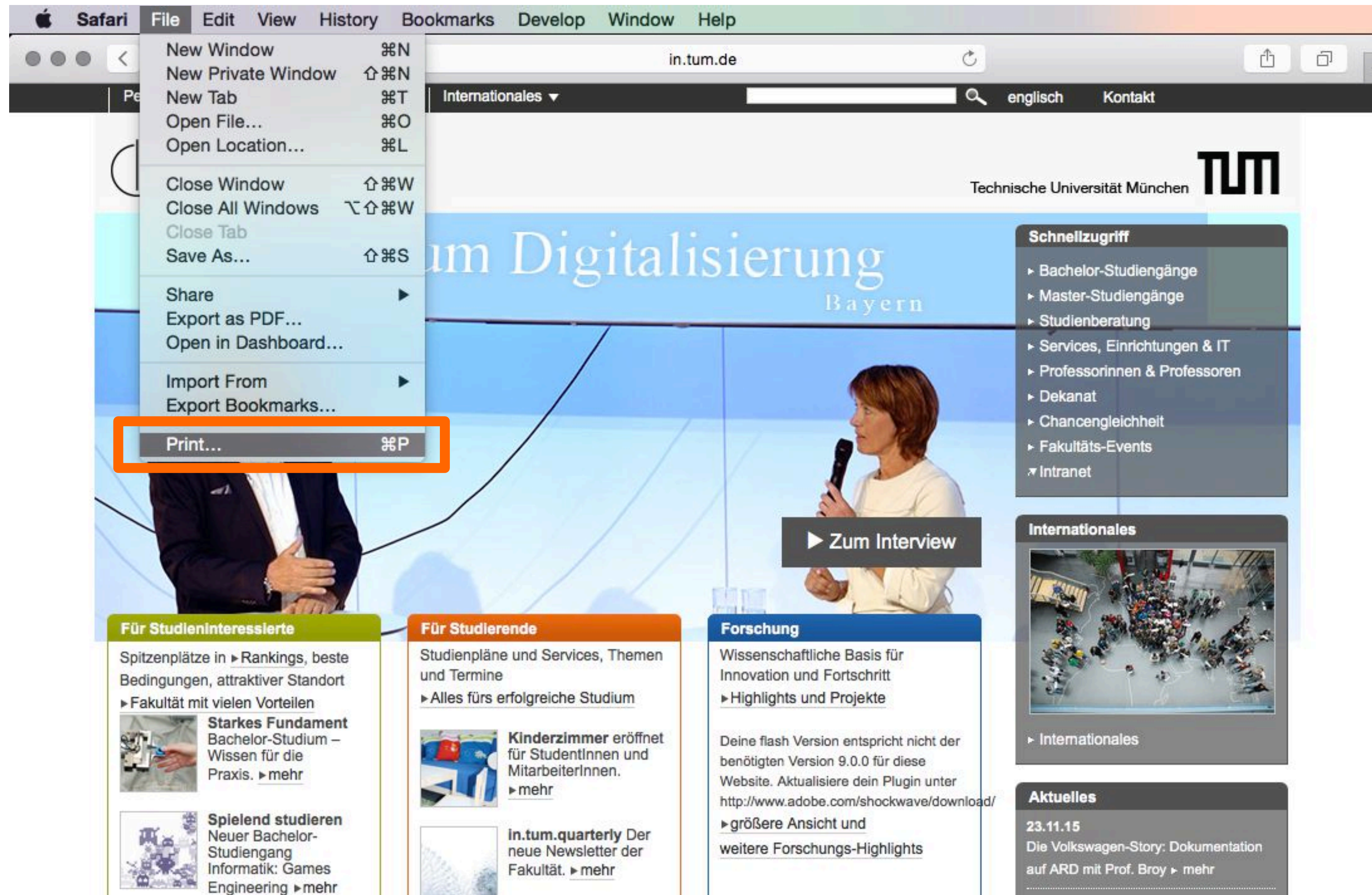
Windows 7



MacOS X

Example: in.tum.de viewed with Safari in English

How do you print this page?



in.tum.de viewed with Safari in Thai



in.tum.de viewed with Safari in Chinese



- The (unconscious) understanding of the user (also called “conceptual model”)
- Helps the user to know and understand the underlying application domain model
- Users apply user models when interacting with devices (especially if they do not read the documentation)
- Depends on
 - Components made visible by the system to the user
 - The mapping between the visible objects of the system and the actions supported by these objects
- A good design enables a user to create a user model in their mind without any problems
 - ➡ Natural mapping

Natural mapping

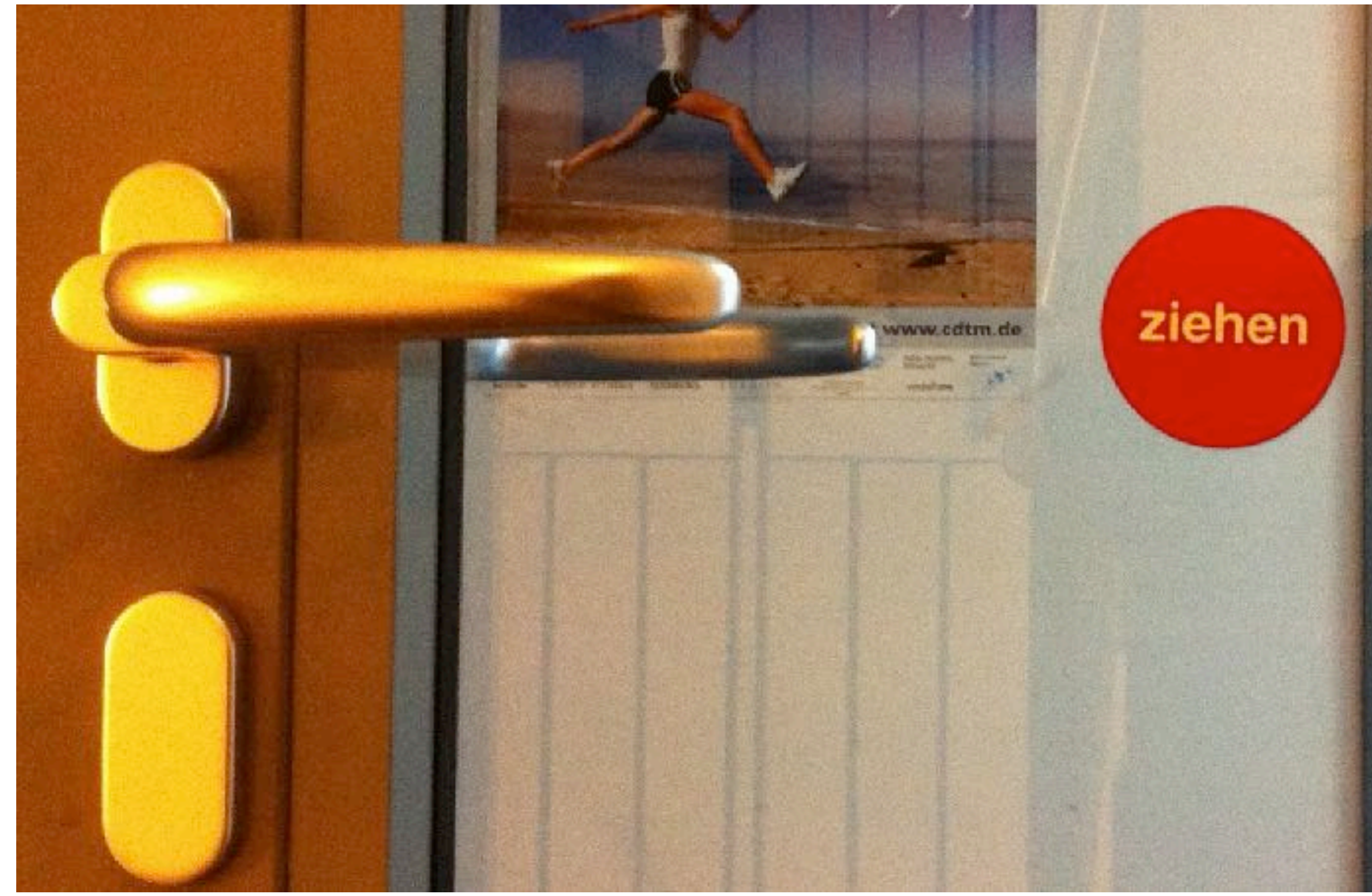
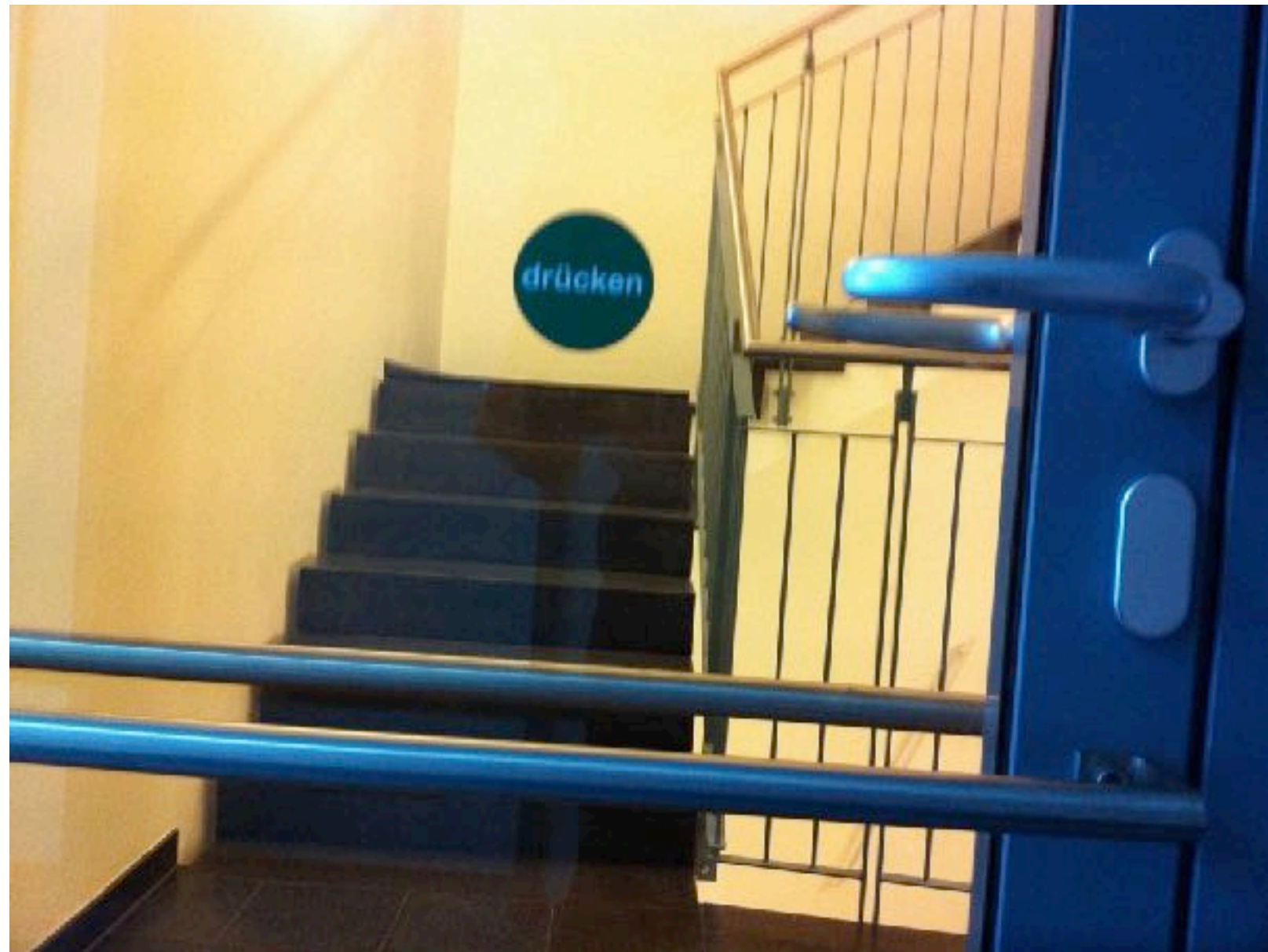
- A mapping between “User interface controls” (boundary objects) of a system and objects in the real world
- The mapping does not tax the user’s memory when performing a task that involves the manipulation of these controls
- Good natural mapping
 - Requires no complicated documentation or no user manual at all
 - Pays attention to the physical layout of boundary objects (e.g. the layout of menu items)
 - **Examples**
 - A stove that can be easily operated when using its knobs
 - A door that can be opened and closed without reading an instruction manual
 - A door that requires a user manual is called a Norman door

Literature: Donald A. Norman, The Design of Everyday Things, Basic Books, 2002.

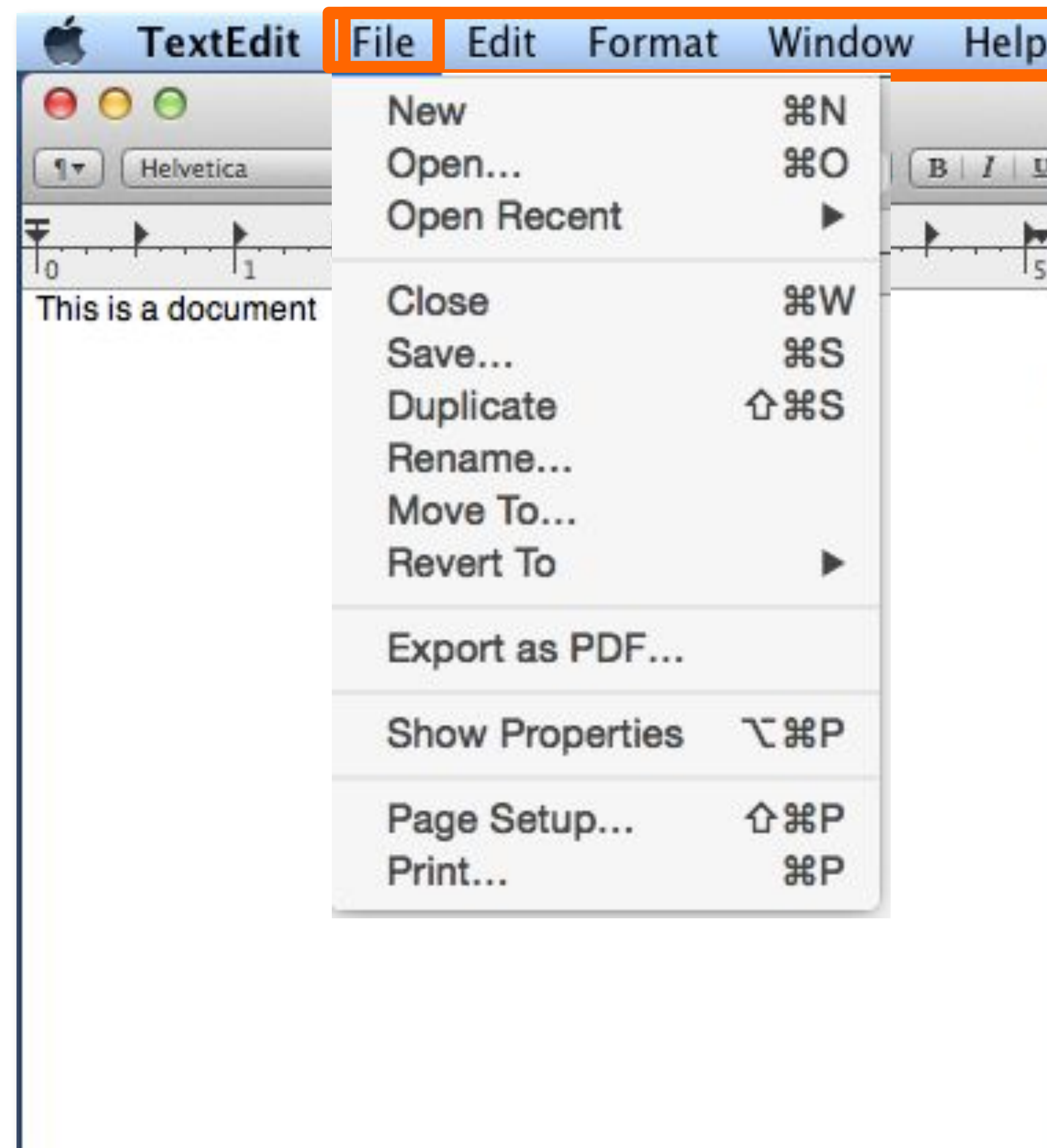
Example of Norman doors



TUM also has Norman doors

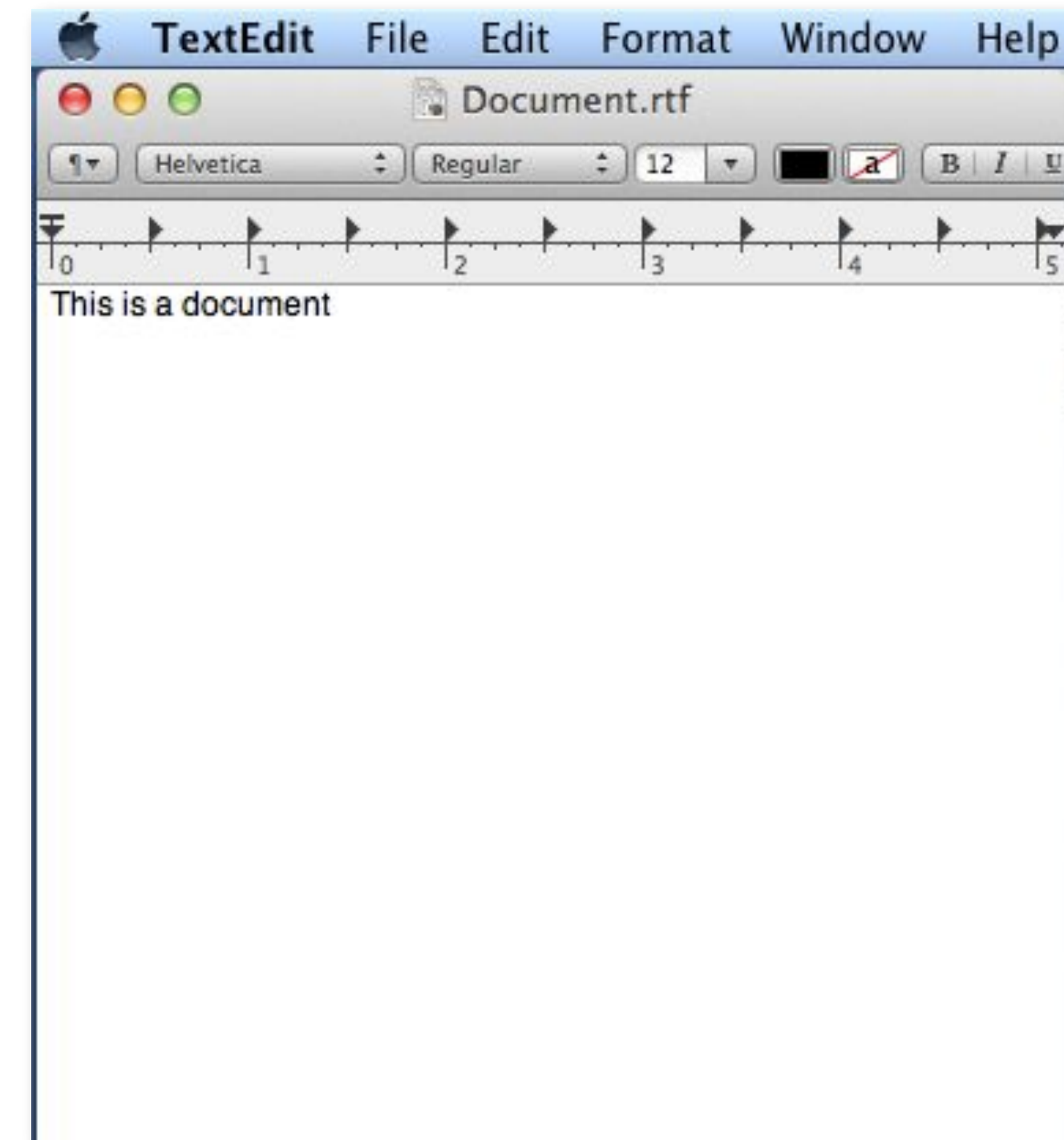


Example: designing the user interface for a text editor



Implementation attempt: the ugly way

```
public static void main() {  
    String command = getUserInput();  
    if(command == "File") {  
        fileCommand();  
    } else if(command == "Edit") {  
        editCommand();  
    } else if(command == "Format") {  
        formatCommand();  
    } else if(command == "Window") {  
        windowCommand();  
    } else if(command == "Help") {  
        helpCommand();  
    }  
}
```

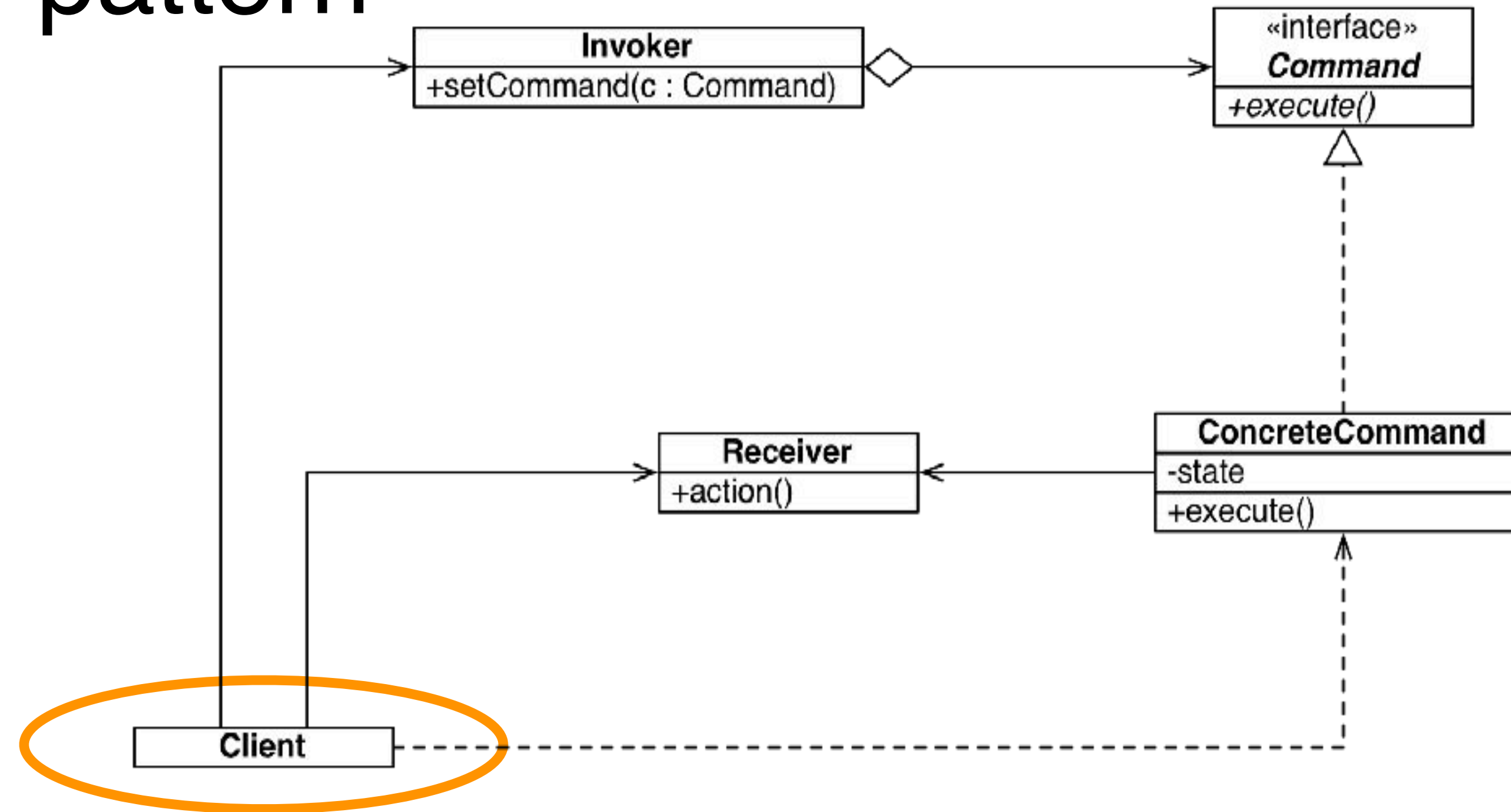


Is there a better way than hardcoding the menu commands?

Implementation attempt: the better way

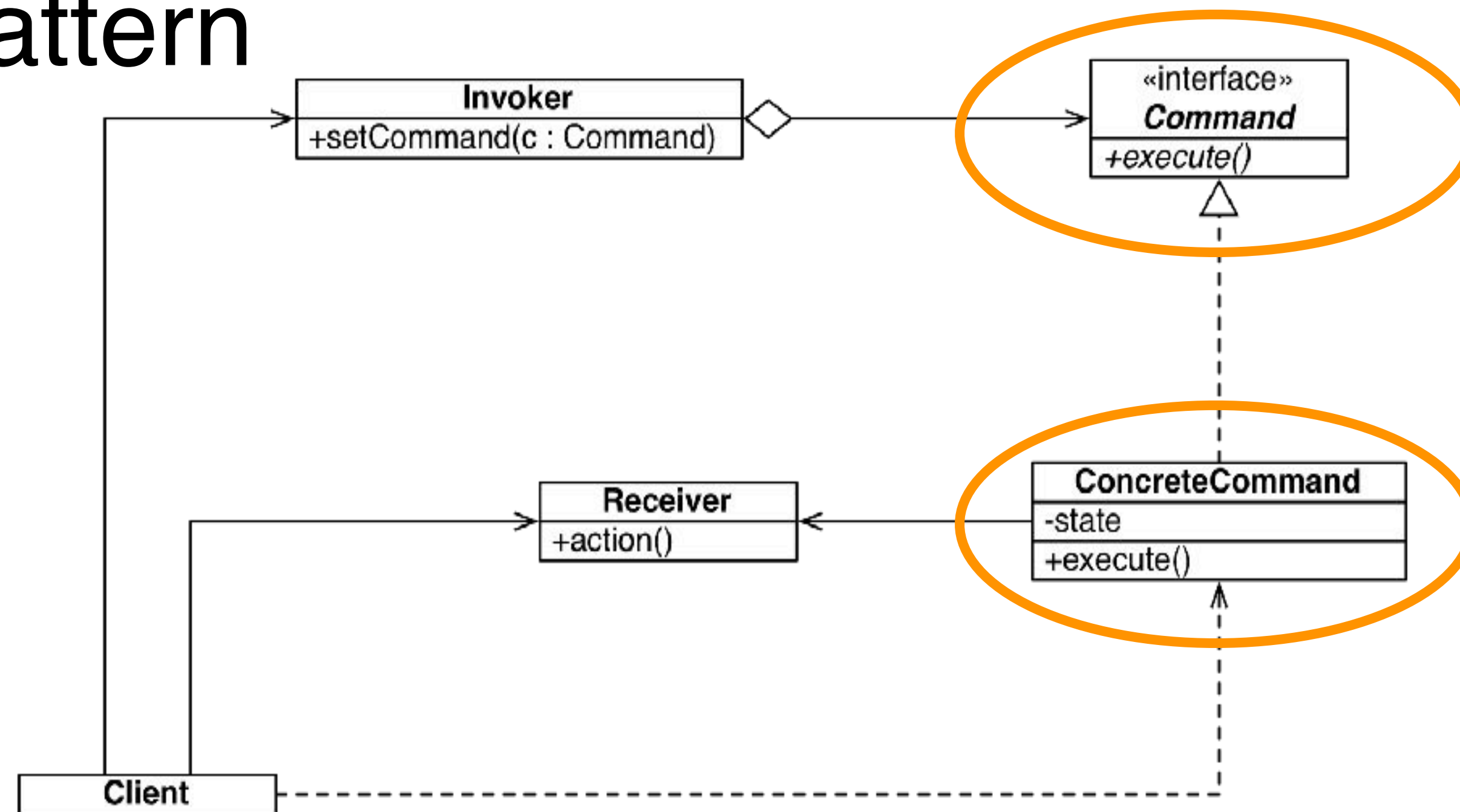
- Start with a command taxonomy
- Map the command taxonomy onto a menu structure
- Let the application know, when a command is selected by the user
- Such a user interface can be implemented with the **command pattern**
 - Can be used to make menus reusable across applications
 - Can be used by user interface builders

Command pattern

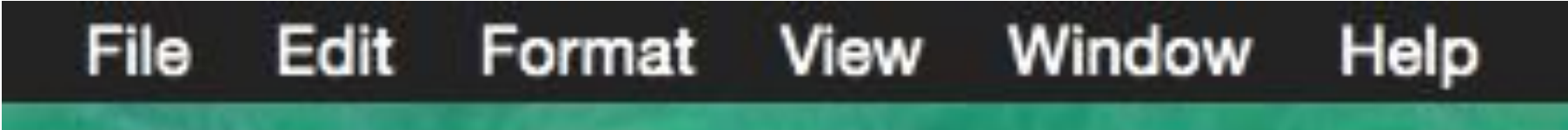


- **Client** is a class in a user interface builder or a class in the application that builds the user interface

Command pattern

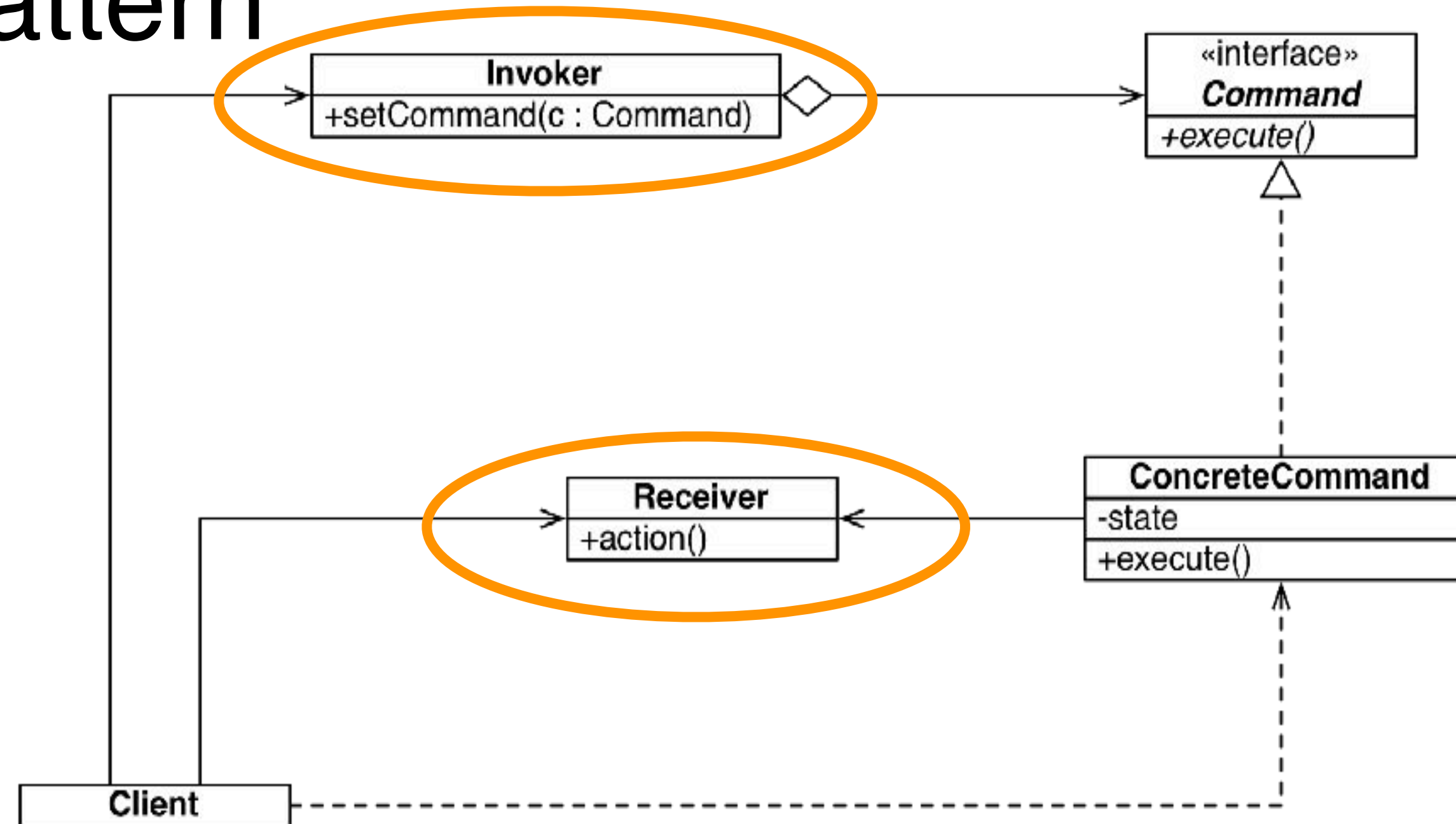


- The **Client** creates instances of **ConcreteCommand**
- All concrete commands implement the interface **Command**
- **Example:** user interface for a text editor



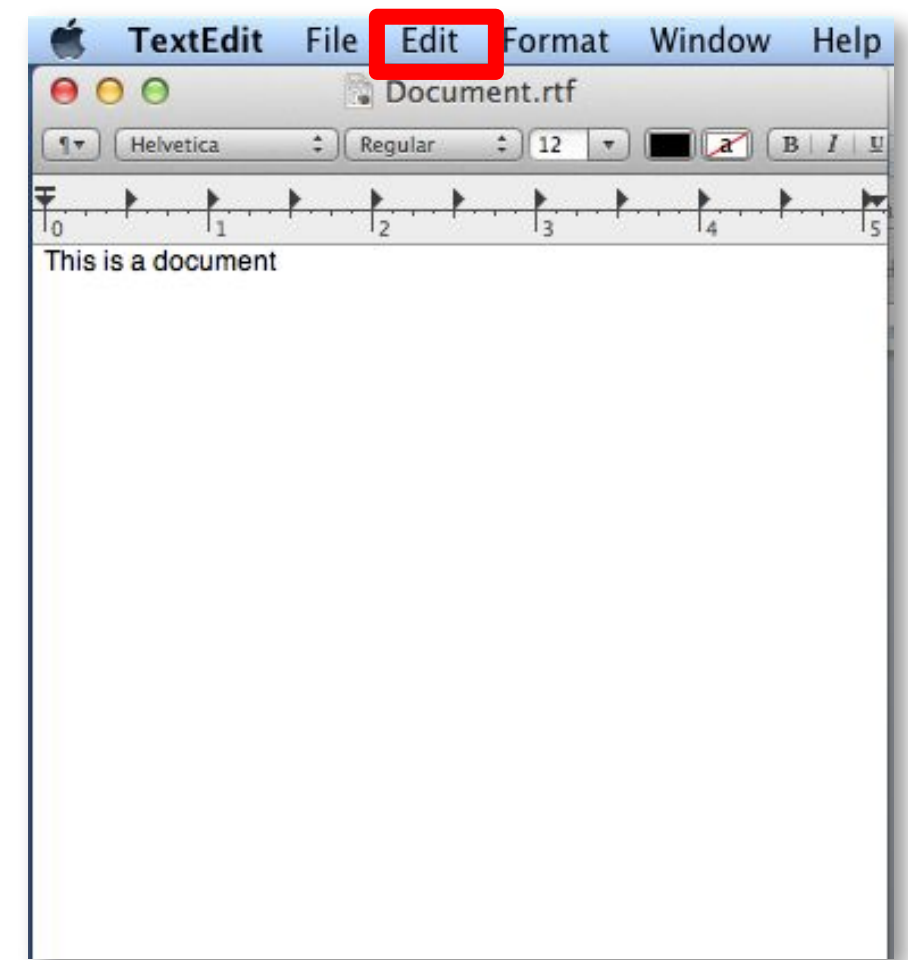
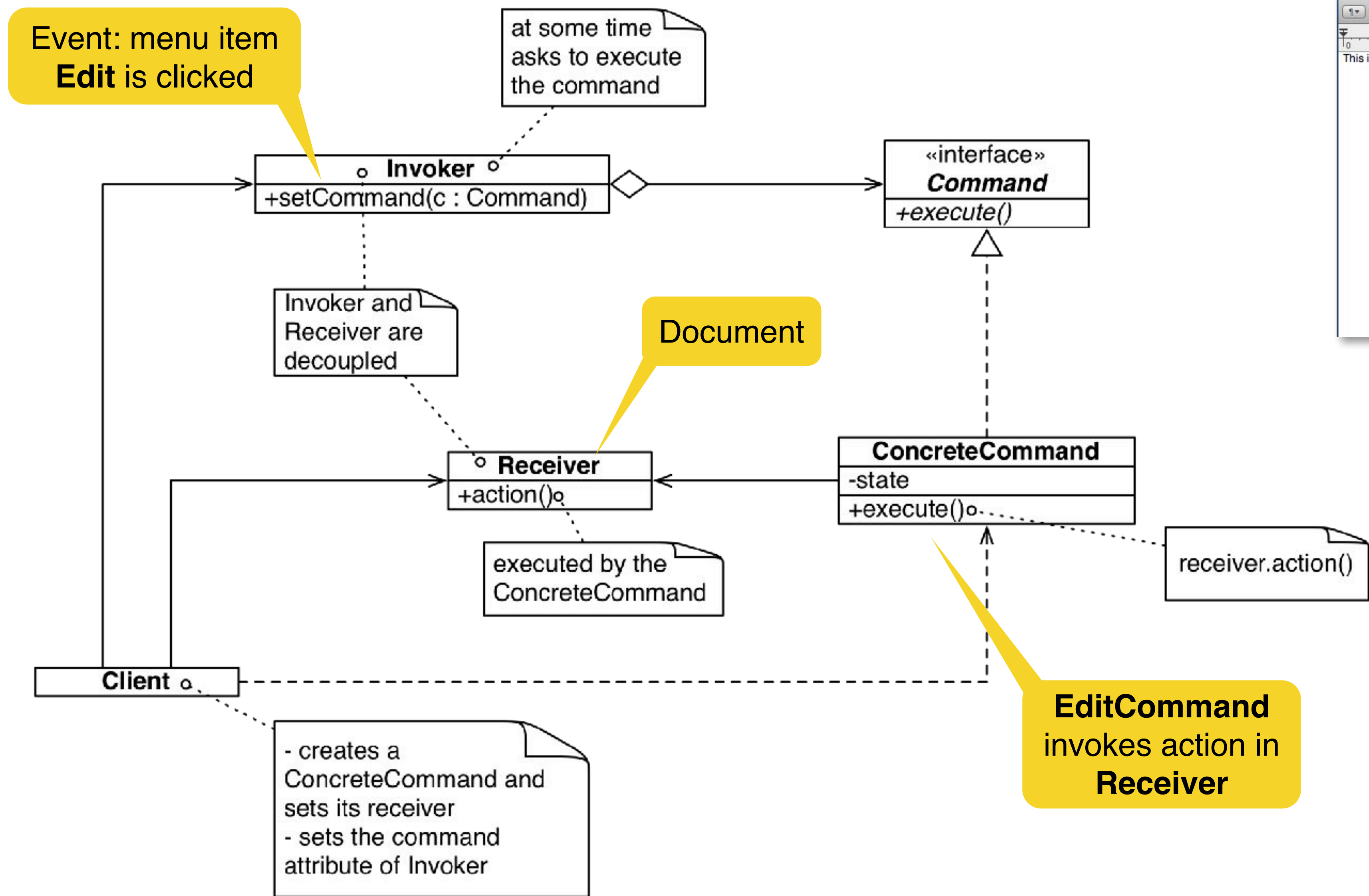
➡ 6 objects of type **ConcreteCommand**: File, Edit, Format, View, Window, Help

Command pattern

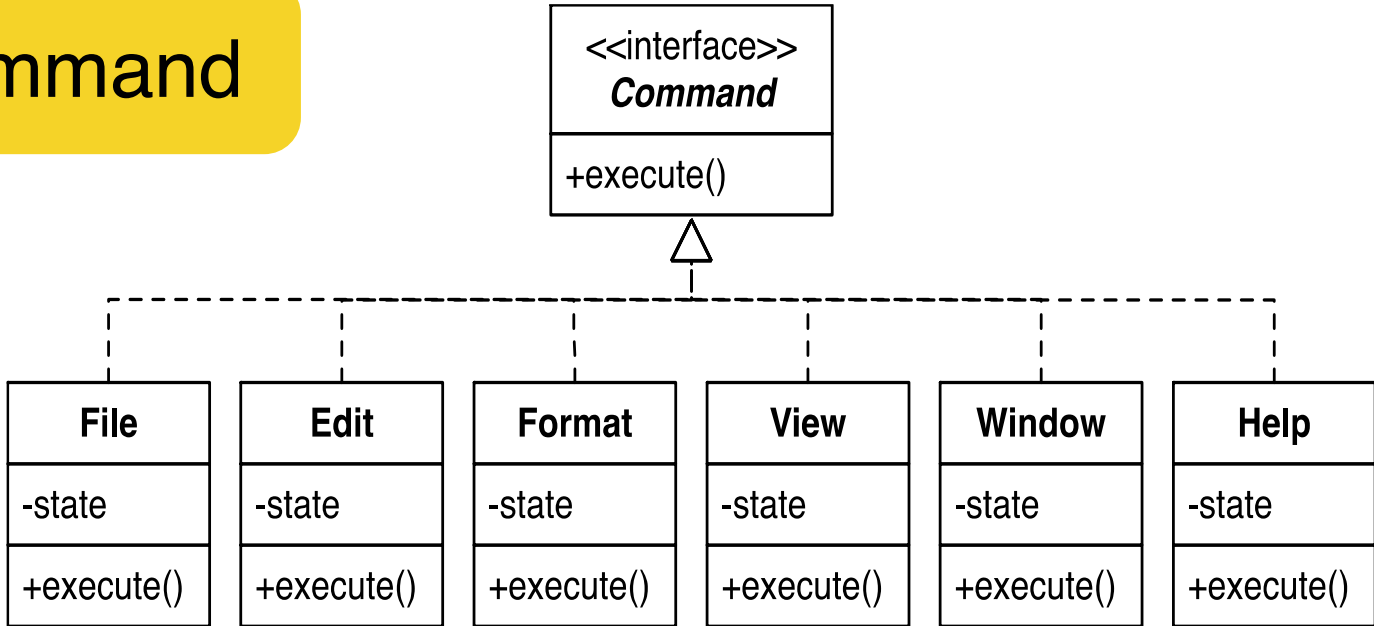
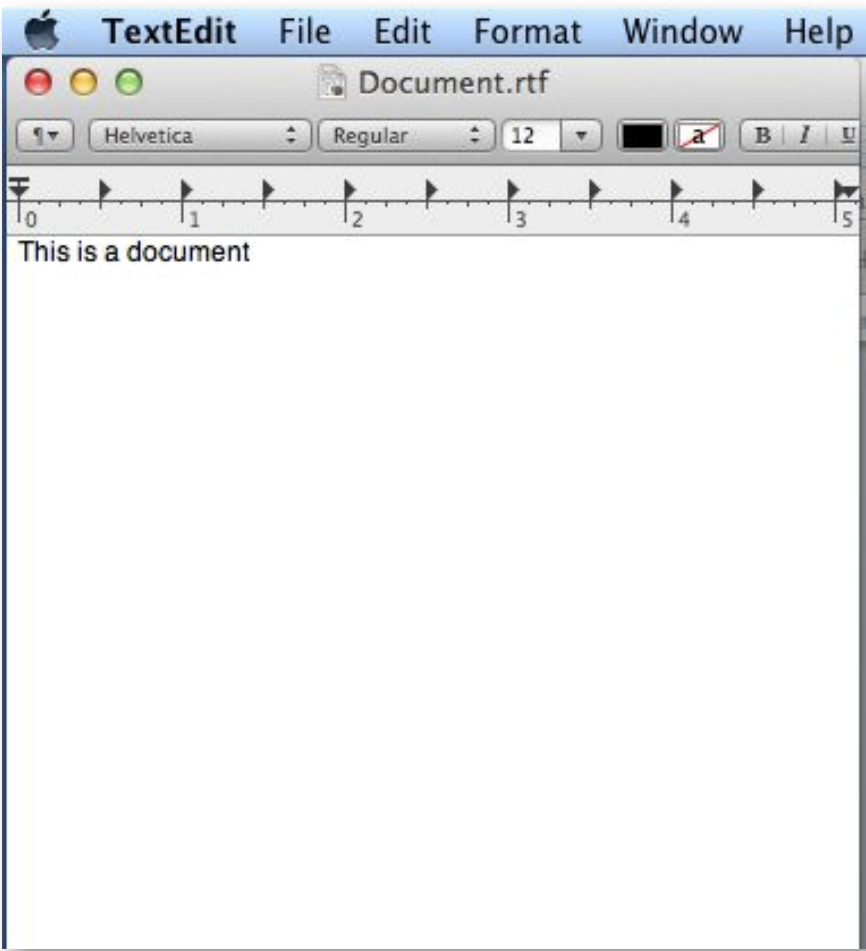
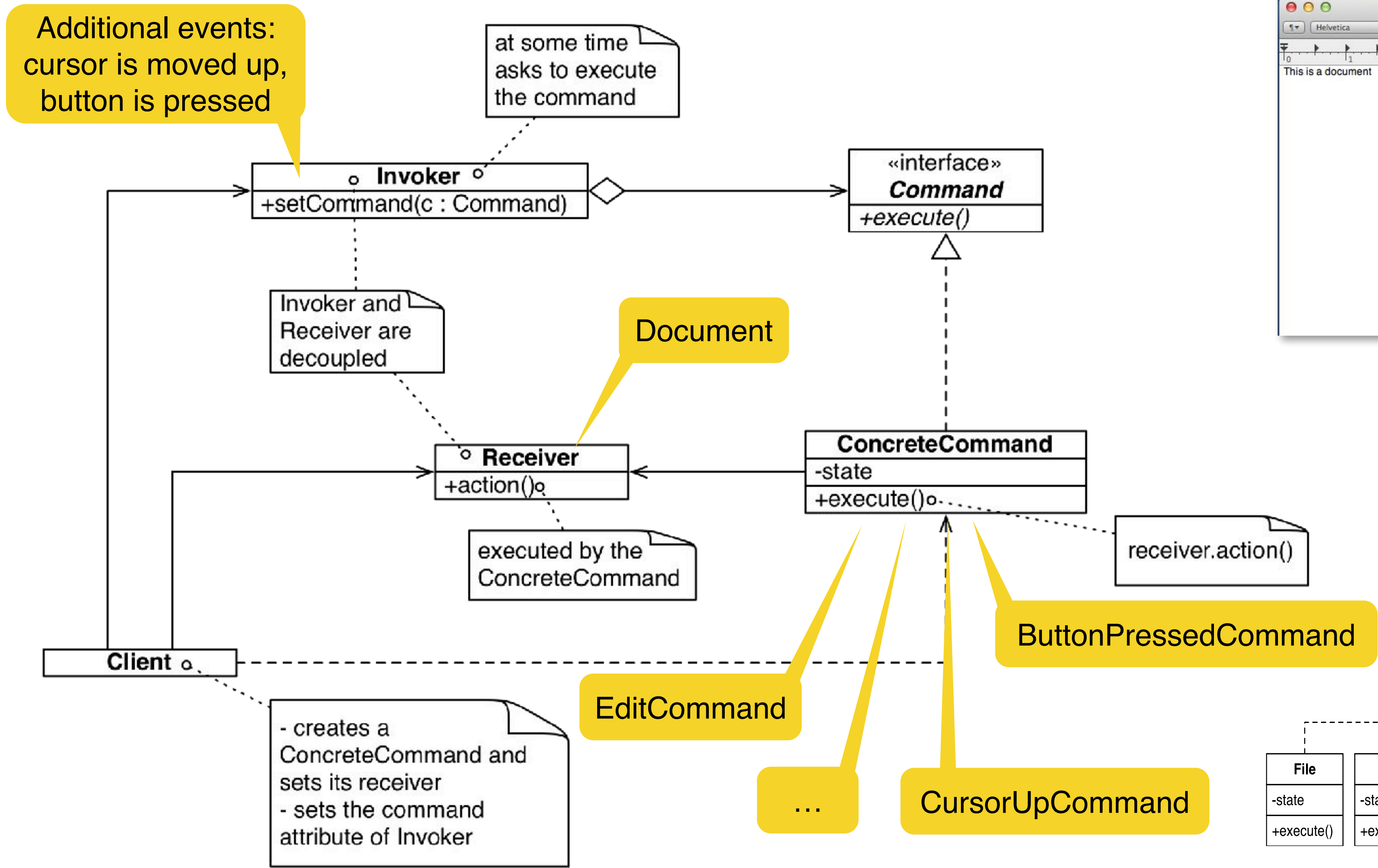


- When a user clicks a button or selects a menu entry or moves the cursor, the associated event is passed from the **Invoker** to the **ConcreteCommand** object by invoking `execute()`
- **ConcreteCommand.execute()** calls the **Receiver.action()** method which does the work (another example of delegation)

Example: command pattern in GUI design



Example: extending the taxonomy



Realizing the command pattern in Java

5 steps to realize a command pattern

1. Create the **Command** interface

The key to the pattern: declares the interface for executing operations, includes an abstract operation **execute()**

2. Create the **ConcreteCommand** classes (control objects)

Each concrete class specifies a receiver-action pair by storing the receiver as an instance variable and provides a different implementation of the **execute()** method to invoke the specific request

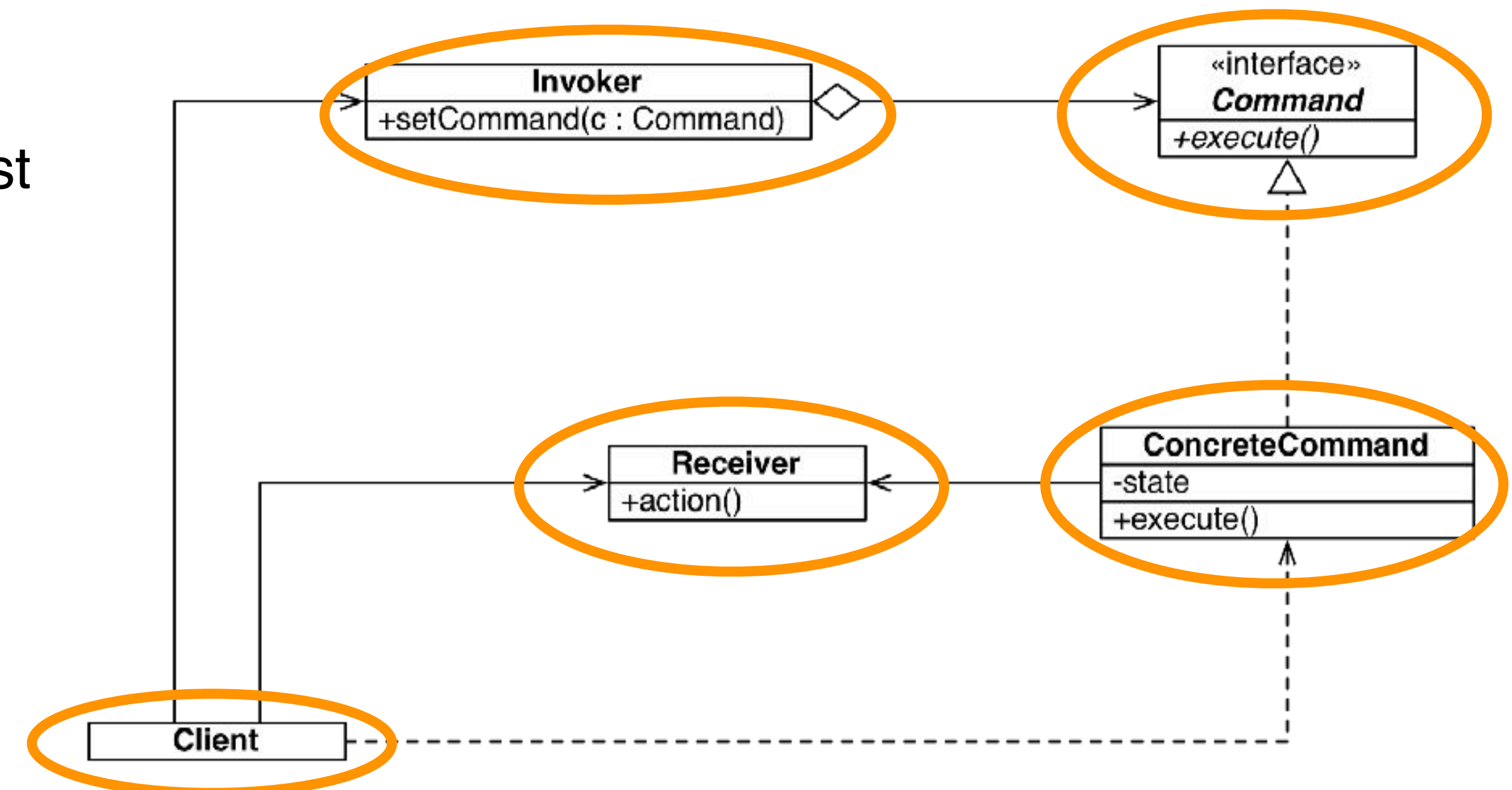
3. Create the **Receiver** (entity objects)

Has the knowledge to carry out the specific request

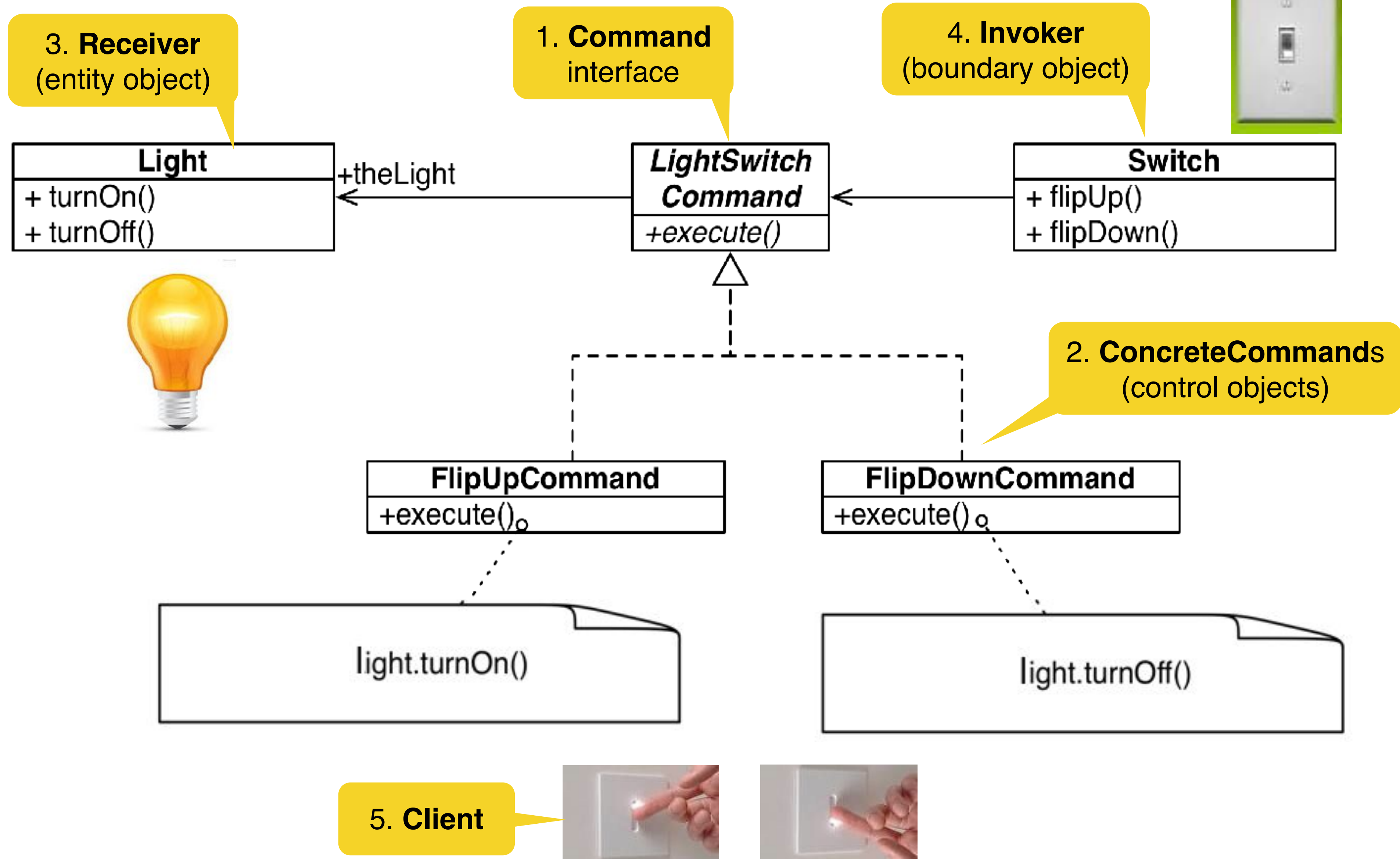
4. Create the **Invoker** (boundary objects)

5. Create the **Client**

➔ **Example:** light switch application



Light switch application



Other applications of the command pattern

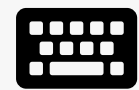


- **Invoker** is an abstraction that can be used in a variety of tasks typical for interactive and real-time systems
- **Command manager**
 - Provides a central repository for all commands available for the client (aka the application)
 - Performs operations based on the commands and responds to command events
 - The client can dynamically add or remove commands from the command manager
- **Redo / undo manager**
 - Pushes the command or events on a stack for possible redo/undo of operations
- **Queue**
 - Holds command or events until other objects are ready to do something with them
 - Useful for schedulers
- **Dispatcher** (also called callback handler)
 - **Example**: keyboard event loop in an interactive system

- **Complexity reduction:** **decouples** boundary objects from control objects and separates them from entity objects
 - **Example:** boundary objects are the menu items and buttons
 - They send messages to the control objects (the **ConcreteCommands**)
 - Only **ConcreteCommand** objects can modify entity objects (the receivers)
- **Dealing with change**
 - When the user interface is changed, only the boundary objects need to be modified
 - The code for the control and entity objects does not have to be touched
 - **Examples**
 - A menu item is added
 - The menu is replaced by a palette
 - The menu is replaced by a speech recognition system



L02E03 Command Pattern



Start exercise

Medium

Not started yet.

Due date in 7 days



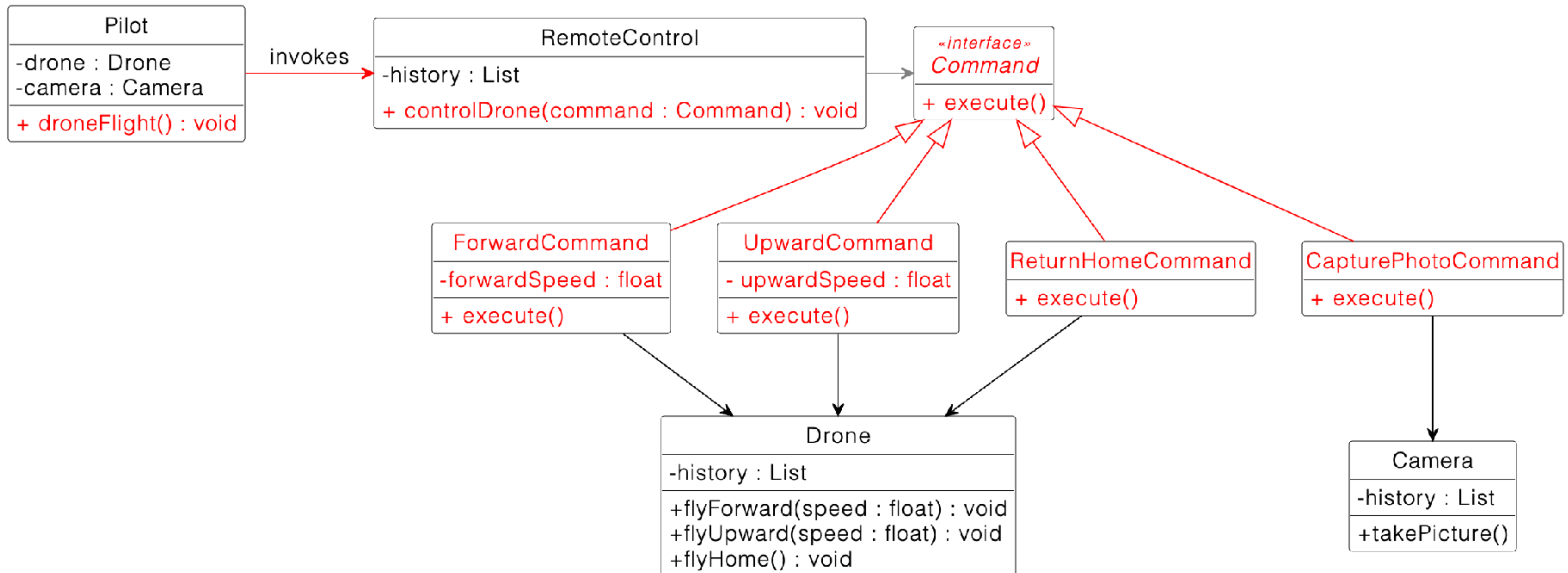
15 min



6 pts



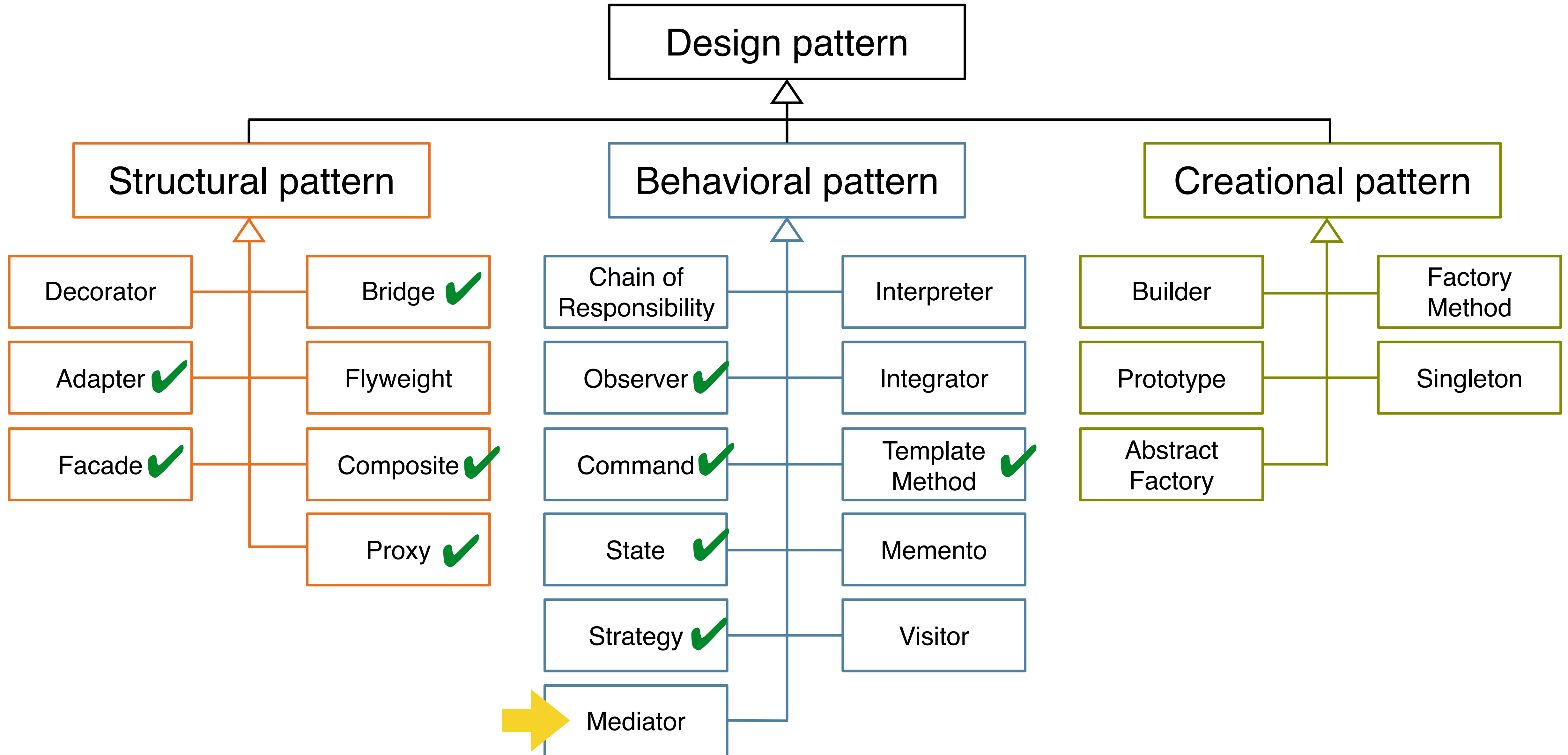
- Problem statement: implement a drone system



Outline

- State pattern
- Template method pattern
- Command pattern
- ➔ Mediator pattern

Design pattern overview

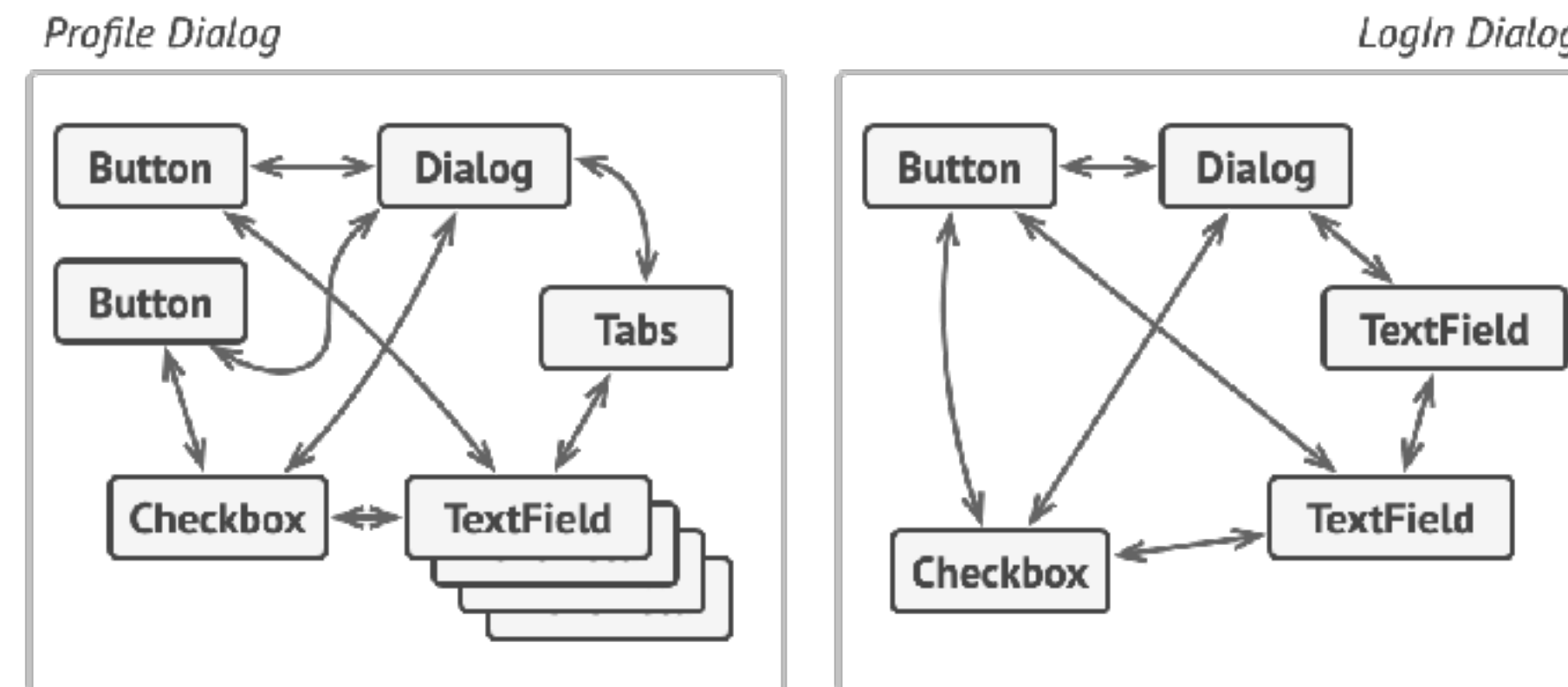


Single responsibility principle

- A class should only have one responsibility
- It should only have one reason to change
- **Benefits**
 - **Easier testing** – a class with one responsibility will have far fewer test cases
 - **Lower coupling** – less functionality in a single class will have fewer dependencies
 - **Better organization** – smaller, well-organized classes are easier to search than monolithic ones

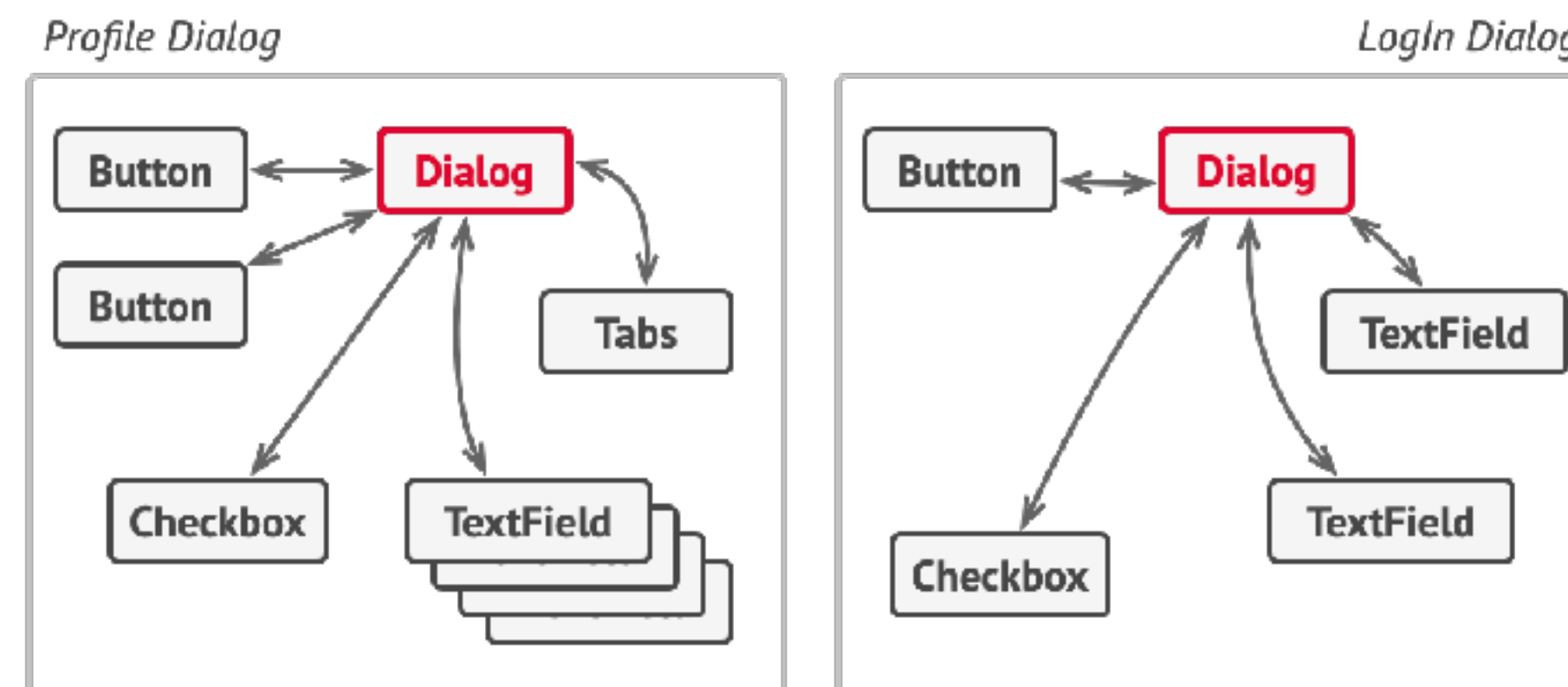
Mediator pattern

- **Problem:** design reusable components, but dependencies between the potentially reusable pieces demonstrate the **spaghetti code** phenomenon



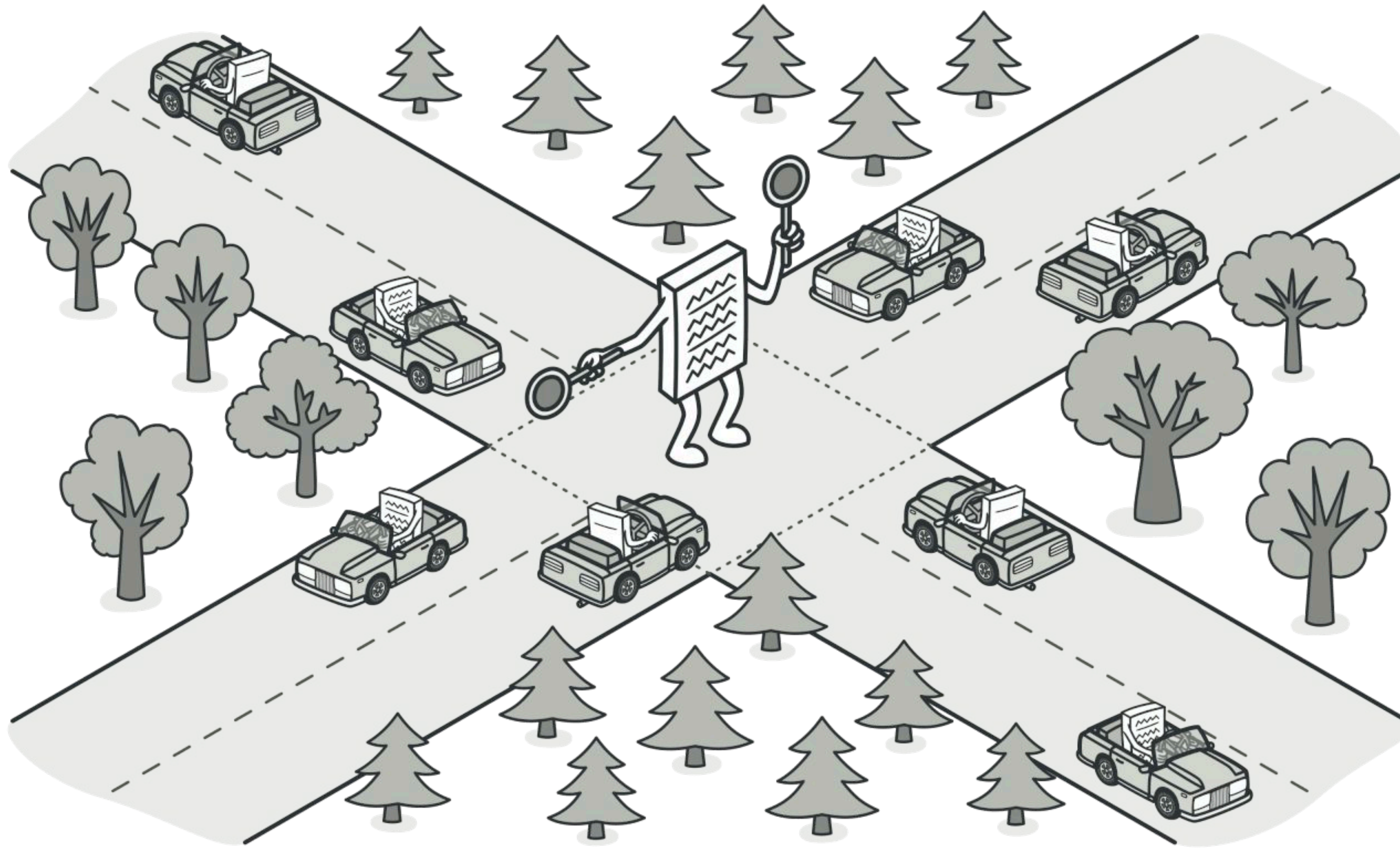
Bad design

- **Motivation:** reduce coupling between classes that communicate with each other



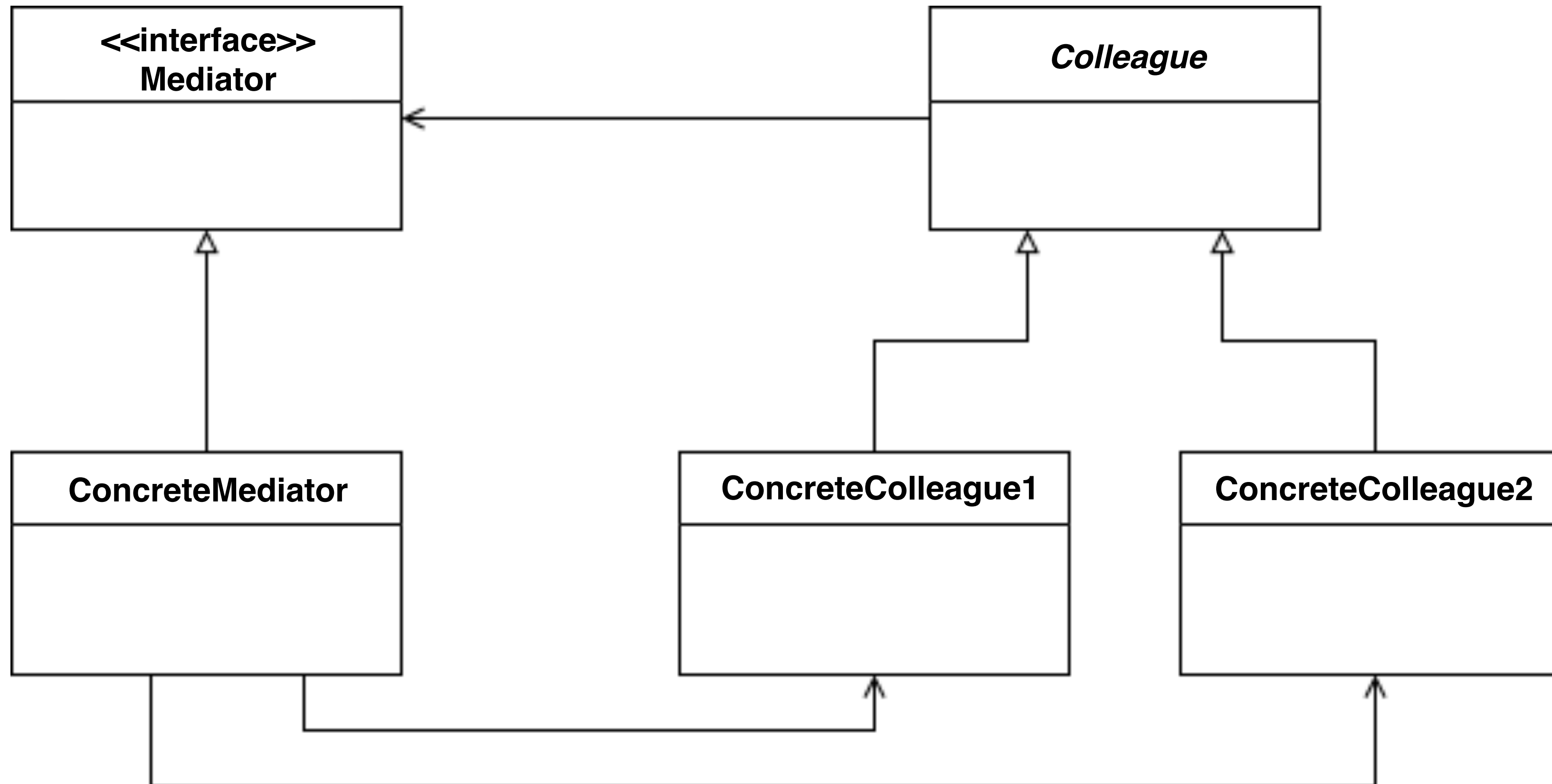
Good design

Example

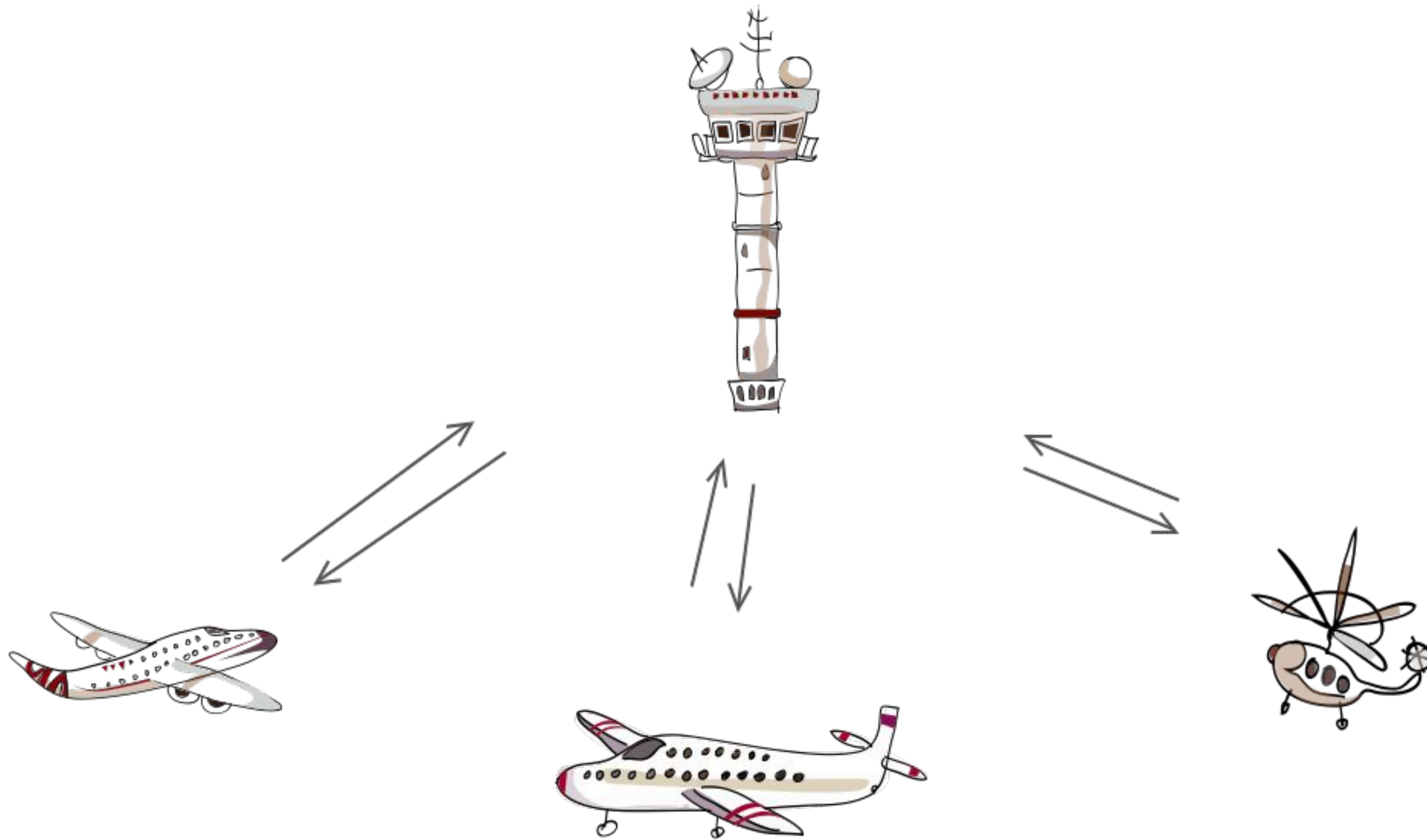


- **Reduce dependencies** between objects
 - Define an object that encapsulates how a set of objects interact
 - Promote loose coupling by keeping objects from referring to each other explicitly
 - Allows to vary their interaction independently
- Design an intermediary object to **decouple** many objects
 - Restrict direct communications between the objects and force them to collaborate only via a **mediator** object
 - Turn many-to-many relationships between interacting peers into an object

Structure



Example



- Good choice if you have to deal with a set of objects that are **tightly coupled** and **hard to maintain**
- Extracts the communication logic to a single object
—> **single responsibility principle**
- You can introduce new mediators with no need to change the remaining parts of the system —> **low coupling**



Open for extension,
closed for modification

- If there are too many tightly coupled objects due to the faulty design of the system, you should **not** apply the mediator pattern
 - Instead take one step back and rethink your system and object design!
- Consider your specific use case before blindly implementing the mediator pattern

Check list



1. Identify a collection of interacting objects that would benefit from mutual **decoupling**
2. **Encapsulate** those interactions in the **abstraction** of a new class
3. Create an instance of that new class and rework all peer objects to interact with the **mediator** only
4. Balance the principle of **decoupling** with the principle of distributing responsibility evenly
5. Be careful **not** to create a controller or god object

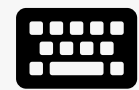
Mediator pattern vs. other patterns



- **Command**, **mediator**, and **observer** address how to decouple senders and receivers, but with different trade-offs
 - **Command** normally specifies a sender-receiver connection with a subclass
 - **Mediator** has senders and receivers reference each other indirectly
 - **Observer** defines a very decoupled interface that allows for multiple receivers to be configured at run-time
- **Mediator vs. observer**
 - **Observer** distributes communication by introducing observer and subject objects
 - **Mediator** encapsulates the communication between other objects
 - ➡ It is easier to create reusable observers and subjects than reusable mediators
 - ➡ **Mediator** can use **observer** for dynamically registering colleagues and communicating with them



L02E04 Mediator Pattern



Start exercise

Medium

Not started yet.

Due date in 7 days



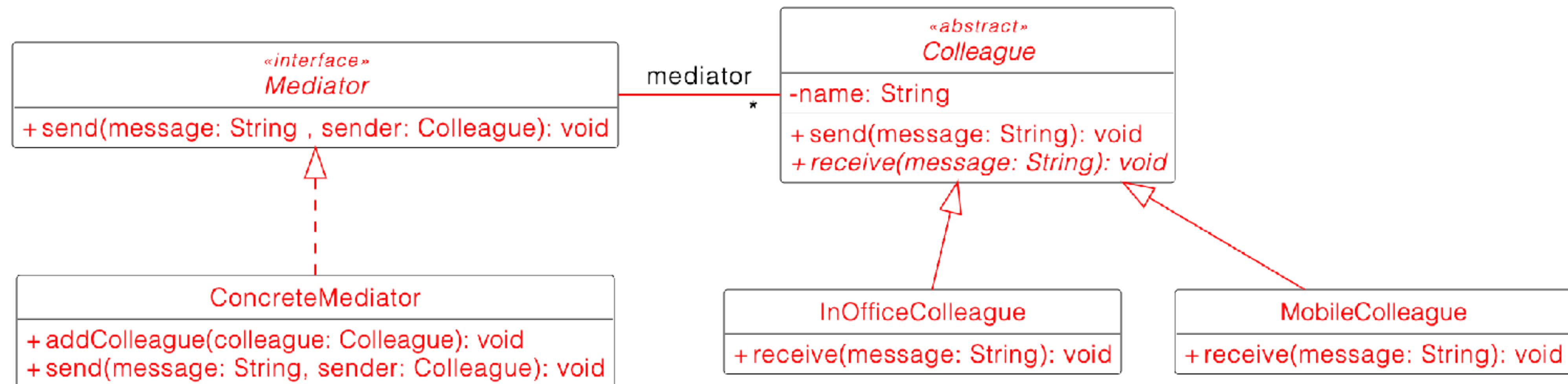
15 min



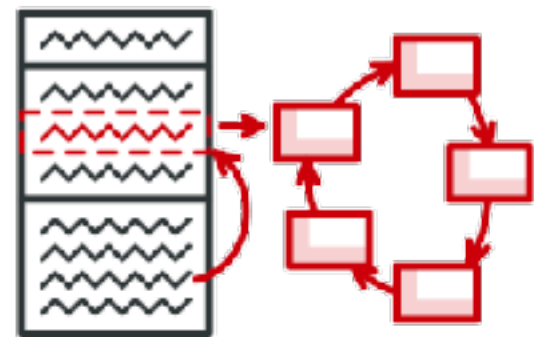
6 pts



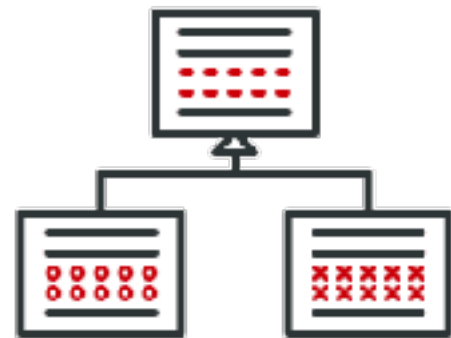
- Problem statement: implement a chat room for colleagues



Summary



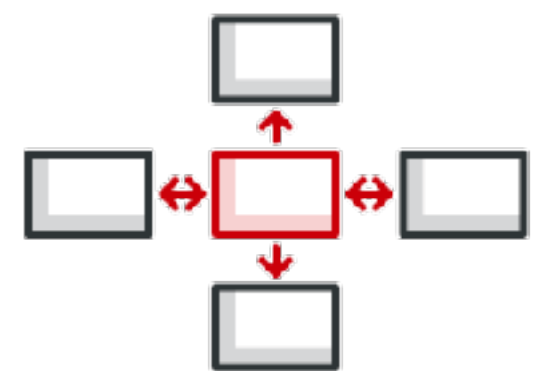
The **state pattern** lets one object alter its behavior when its internal state changes by turning the states into objects



The **template method pattern** defines the skeleton of an algorithm in the superclass and lets subclasses override specific steps of the algorithm without changing its structure



The **command pattern** turns a request into an object containing all information, allows to delay or queue requests, and supports undoable operations



The **mediator pattern** reduces dependencies between objects, restricts the communications between them and forces them to collaborate only via a mediator object

- Eric Gamma et al: Design patterns, Addison Wesley, 1995
- Elisabeth Freeman et al: Head First Design Patterns, O'Reilly 2004
- <https://refactoring.guru/design-patterns>
- https://sourcemaking.com/design_patterns
- Donald A. Norman, The Design of Everyday Things, Basic Books, 2002
- D. Norman, S. Draper. User centered system design; new perspectives on human-computer interaction. L. Erlbaum Associates Inc., 1986