7c. Design Patterns – Behavioral Patterns

Three types of design patterns

- **Creational** patterns: How do we create objects?
- **Structural** patterns: How do we compose large objects out of small objects?
- **Behavioral** patterns: How do objects work with each other to achieve desired behavior?

Mediator

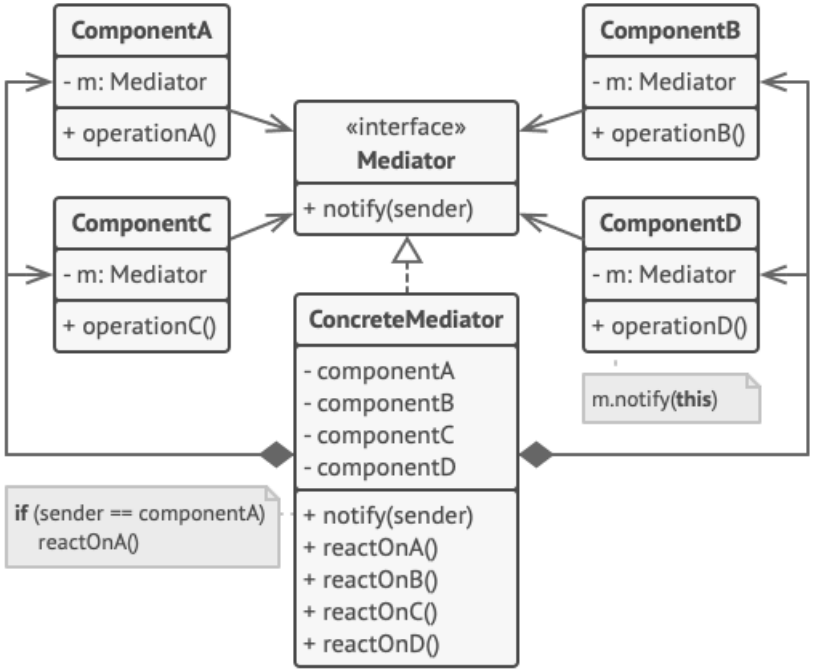
- Whenever an object refers to another, coupling is increased
  - Rat contains a copy of Cat to run from it, Cat refers to Walls to check collision, Cat checks Rats to see if an attack succeeds, etc…
- Coupling can prevent object reuse: If I want a copy of the Cat code for a different game, I may have to remove all the code about Rats
- We should instead program a *Mediator* object that handles inter-object communication

Mediator

- Motivating example: A dialog box for font selection
- Objects: Buttons, Checkboxes, TextFields…
- They are closely coupled:
  - Choosing a font may disable certain weight and slant
  - Clicking the “condensed” checkbox may disable the “bold” radio selection
  - Choosing a different font may reset the weight and slant to default

Mediator

- Inter-object communication should pass through a Mediator:



Mediator: Terminology

- Mediator: one object that handles all inter-object communication (of a certain type)
- Components: communicate through the Mediator

Mediator

- The Mediator contains objects of each component type
- Each object will also contain the Mediator but no other objects
- The Mediator only need a single notify() method
  - In the text box example, whenever any dialog changes, it notifies the Mediator
- Each object can notify the Mediator

Mediator: Example

```
public class LoginDialog implements Mediator {
    void notify(Object sender, String event) {
        if (sender instanceof LoginButton) {
            if (SQL_check(loginUsername, loginPassword)) {
                if (rememberPasswordBox.checked) //…
            }
            else {
                time.sleep(1);
                warningBox.setText("Incorrect password.");
            }
        }
    }
}
```

Mediator: Benefits and Downsides

- Reducing coupling
  - Allow you to write better classes that do not need to rely on other classes (though they need to depend on a Mediator)
  - We can subclass the Mediator to change behavior instead
    - RegistrationDialog can be a different type of mediator using the same objects
- Centralize communication operations
  - Easier to understand object interactions – one-to-many instead of one-to-one
  - Easier to find/modify interaction code
- Downside: one complicated, monolithic class
  - A “god class” is a code smell…

Command

- If object interaction becomes complicated, direct method calling may be too clumsy
- Document editor example: pasting text from the clipboard
  - Several ways to do so: shortcut key, right click buttons, menu buttons…
  - Pasting into a table and pasting into a text box may be slightly different
  - Pasting as pure text and pasting with formatting

Command

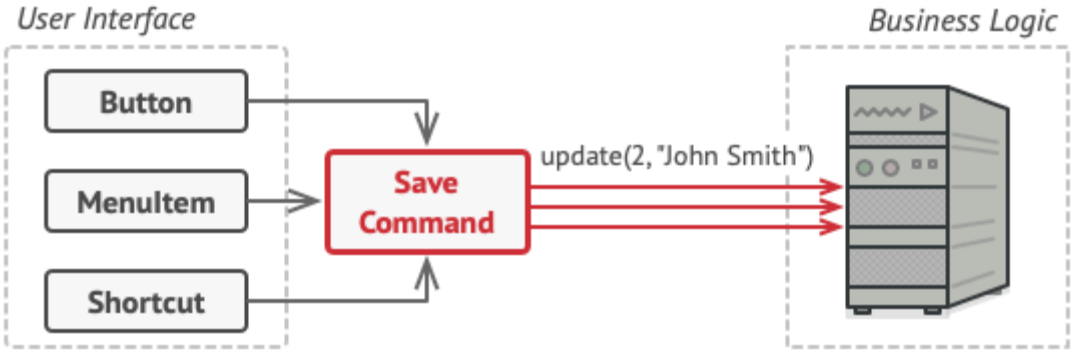
- Method calls:

```
void onKey(Key key) {
    if (key.StringEquals("Ctrl+V")) {
        if (!settings.formatDisabled) {
            paste(clipboard, format=null);
        }
        if (context instanceof TableCell) {
            context.expand(clipboard.length());
            paste(clipboard);
        }
        //more possibilities…
    }
}
void UnformatPasteButton.onClick() {//…}
```

- Bad for *Separation of Concerns*: Shortcut function needs to understand paste

Command

- Unify object requests into a *Command* design pattern



refactoring.guru

- Caller creates a Command object and delegates command execution to it
  - Different implementations of Command can be used
- Similar to callbacks for functional programming

Command: Example

- Client code becomes very simple

```
void onKey(Key key) {
    if (key.StringEquals("Ctrl+V")) {
        if (PasteCommand == null) {
            PasteCommand = new PasteCommand();
        }
        PasteCommand.execute(this);
    }
}
```

- PasteCommand.execute() can grab all necessary information from the calling object

Command: Example

```
class PasteCommand extends Command {
    PasteCommand(Object context) {
        if (context instanceof UnformatPasteButton) {
            this.disableFormat = true;
        }

    }
    void execute(Object context) {
        if (context instanceof MainWindow) {
            if (context.textSelected() != null) {
                editor.removeText(context.textSelected());
            }
            editor.addText(clipboard, format=null);
        }
        //...
    }
}
```

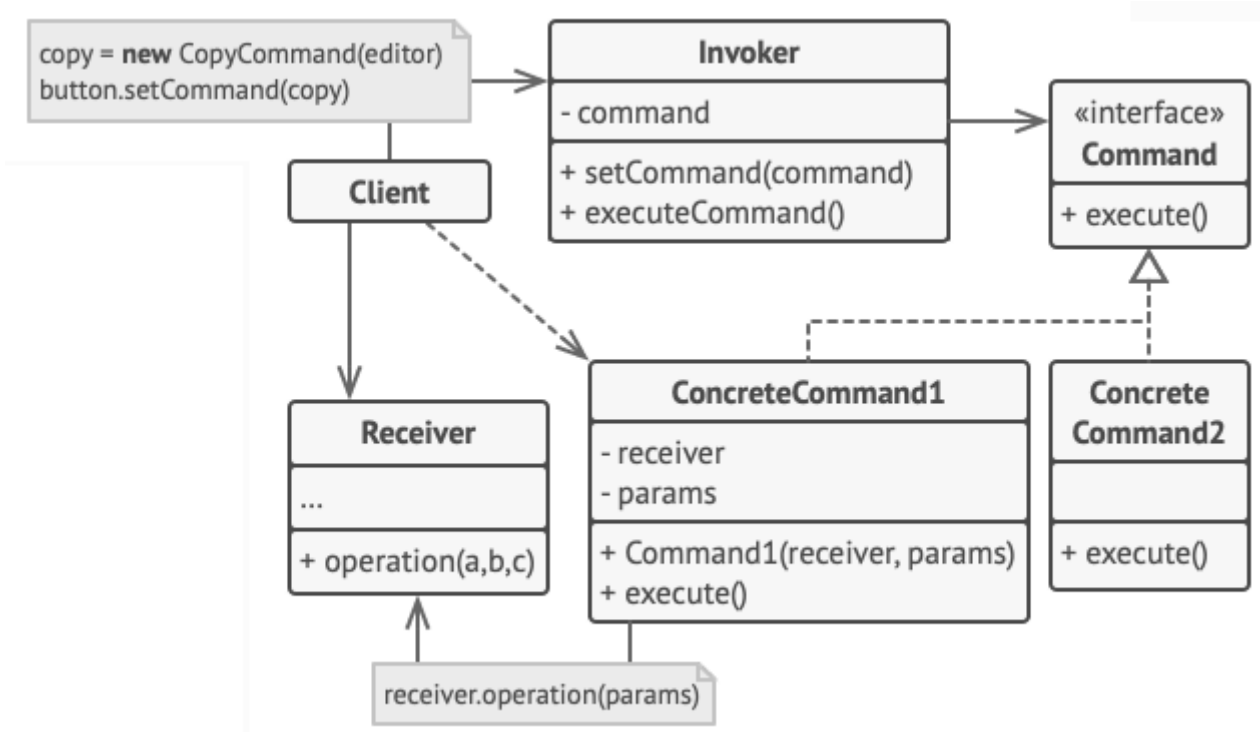
Command: Benefits

- Single Responsibility: All Paste code can be found in one obvious location
  - Avoids unexpected bugs if code is copied multiple times into multiple locations, possibly with modifications
  - Easier to read and understand
- Open/Closed Principle: Easy to add new commands
- Command object can easily handle more complicated functionality
  - Queue an action: command object has access to a command queue, adds call to the queue with internal logic; allows deferring execution if it would be helpful
  - Undo/redo: command object can store history that allows reversion

Command: Terminology

- Command: implements an execute() method that is called by Invokers
  - Concrete Command: implements Command as an interface
- Invoker: calls the Command's execute() method
- Receiver: called by Command's execute() method to perform the required actions
- Client: creates and correctly sets the Command

Command


copy = new CopyCommand(editor)
button.setCommand(copy)
Invoker
- command
+ setCommand(command)
+ executeCommand()
«interface»
Command
+ execute()
Client
Receiver
...
+ operation(a,b,c)
ConcreteCommand1
- receiver
- params
+ Command1(receiver, params)
+ execute()
Concrete
Command2
+ execute()
receiver.operation(params)

Iterator

- Many different ways to implement “lists” of objects:
  - Array, linked list, tree (B-tree), matrix, …
- Often, caller just wants to traverse all elements of an object
  - Caller does not care which implementation is being used
- An Iterator object handles this with only two method calls:
  - next() – returns the next object
  - boolean hasNext()

Iterator: Java

- Java's Collection extends from Iterable
  - e.g. List, ArrayList, …
  - e.g. HashMap can return a Set, which extends Collection
- Iterable has an iterator() method call that returns an Iterator
- Several options for looping over all elements:
  - Using Iterator's hasNext() and next()
  - Using forEach on the Iterable
  - Using a for loop with indexing on a List

Iterator: Java

```
List suits = ...;
List ranks = ...;
List sortedDeck = new ArrayList();

// BROKEN - throws NoSuchElementException!
for (Iterator i = suits.iterator(); i.hasNext(); )
    for (Iterator j = ranks.iterator(); j.hasNext(); )
        sortedDeck.add(new Card(i.next(), j.next()));
```

```
for (Suit suit : suits)
    for (Rank rank : ranks)
        sortedDeck.add(new Card(suit, rank));
```

Iterator

- If you're writing a Tree class, you should make it implement Iterable
  - next() would get the next element according to depth-first or breadth-first
- Other uses:
  - Find a car over roads on a city's map
  - Check all elements of a complex shopping order for validity
  - Check for updates from all channels on a messaging app
  - Composites

Iterator

- You can also implement a custom Iterator object
- e.g. use API calls to iterate over all friends on Facebook and Discord, send them a message
  - next() and hasNext() would implement the API calls
  - FacebookIterator and DiscordIterator would be implementations of a SocialIterator interface
- Advantages of Iterator:
  - Client code is written for general iterators, allowing you to substitute different iterators
  - Client cannot access or change iterated objects directly
  - If traversal is complicated, we achieve Single Responsibility Principle
  - Simplifies iterating over multiple objects

Memento

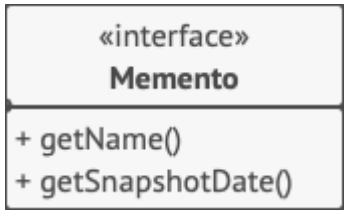
- How do you implement a **save** function?
  - Similarly, how do you undo/redo?
- Naive solution: save function visits all objects and records their state
  - Not all states are public or have getters
  - This makes the save function dependent on *all* objects
  - It breaks encapsulation

Memento

- Instead, delegate the work to each saveable object
- Each saveable object is able to make a “Memento” – a snapshot that contains its saved state
- Each saveable object should implement two public functions:
  - Memento makeSnapshot()
    - Creates the Memento
  - void restore(Memento memento)
    - Restores the object’s state to that of the Memento

Memento: Interface

- The Memento interface can be intentionally restrictive:



- This means that other objects cannot set field values in a Memento
- Only the original object can use the Memento in the restore() function
- Alternative implementation: Nested class

Memento: Caretaker

- The Caretaker interacts with Mementos to support functionality
  - e.g. undo, redo, save, load
- Undo/redo: Caretaker has a History object that saves all Mementos
  - Every command adds a snapshot to the History
  - If user undo's, restore the snapshot's object
  - getName() is used to determine which object is being restored
  - getSnapshotDate() is used to determine which is the most recent object
- Save/load: Saved file is parsed as object state from all Mementos

Memento: Example

- In Settlers of Catan, the saveable objects are:
  - Board states: robber location, buildings, yields
  - Player states: resources, cards, achievements
  - Game states: whose turn
- Did we forget anything?
- Using save states to cheat randomizer?

Memento: Example (Nested class)

```
class Board {
    private List<Building> buildings;
    private List<Player> players;
    private int robberLocation;
    private List<int> yields;

    Memento makeSnapshot() {
        return new Memento("Board", buildings, robberLocation, yields);
    }
    void restore(Memento boardMemento) {//setters}

    private class Memento {
        String memName;
        List<Building> memBuildings;
        List<int> memYields;
        public Memento(...) {//constructor is also setters}
    }
}
```

Memento: Example

```
public class SaveLoad {
        public void saveFile(File f, Board board) {
            Object boardMemento = board.makeSnapshot();
            //serialize boardMemento, write to file
        }
        public void restoreFile(File f, Board board) {
            //obtain mementos from file, then
            board.restore(boardMemento);
        }
}
```

Memento: Terminology

- Originator: Board – the object that makes the Memento
- Caretaker: SaveLoad – the object that uses Mementos to support undo/redo/save/load

Memento: Caveats

- Take care to store Mementos correctly especially if state regards interaction of two objects

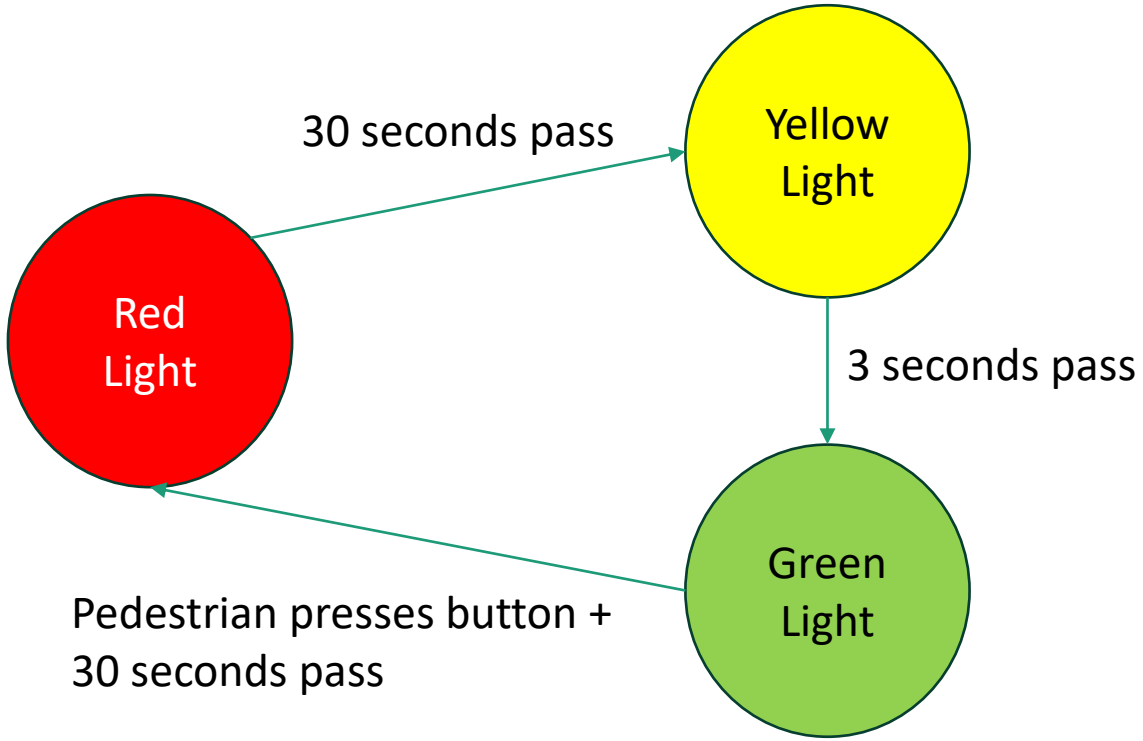
1. User drags a box…

2. User presses undo…

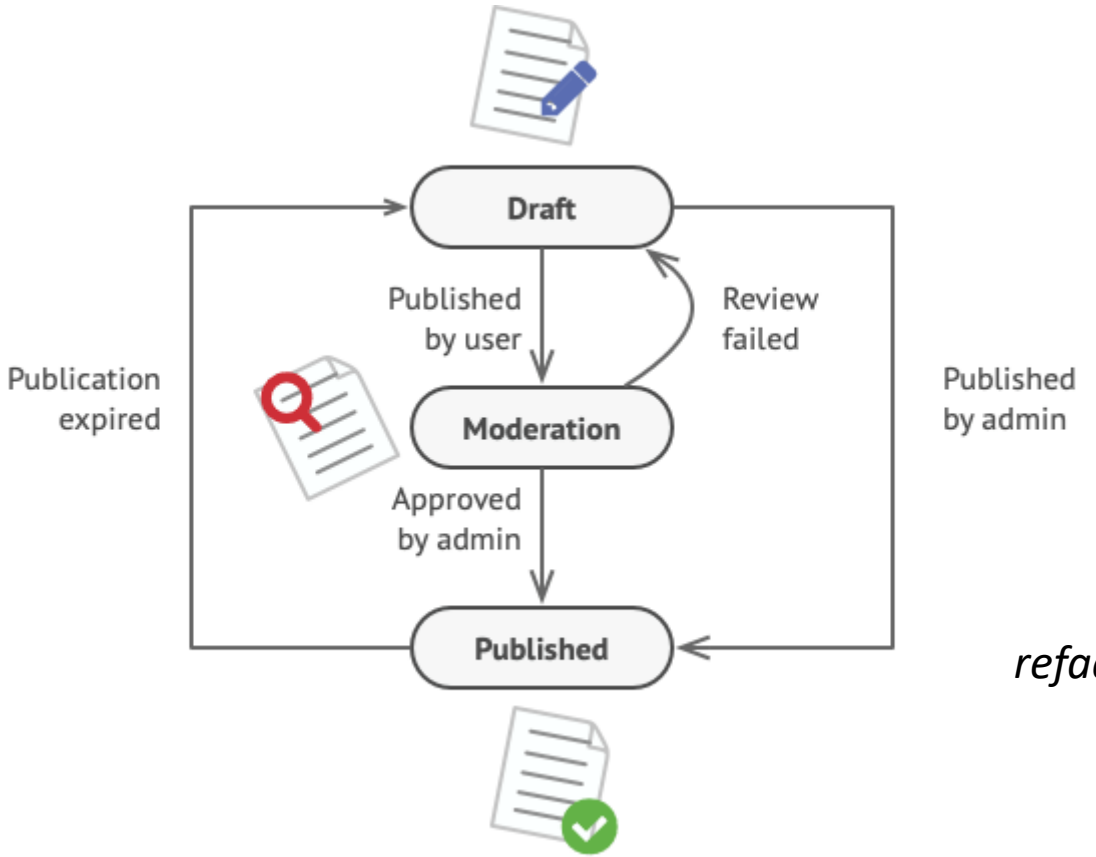
- Manage storage size: Mementos are stored in memory too
- Command and Memento:
  - Command changes the state of an object
  - Memento saves state before each Command

State

- Many objects can be implemented as a finite-state machine:



State



refactoring.guru

State

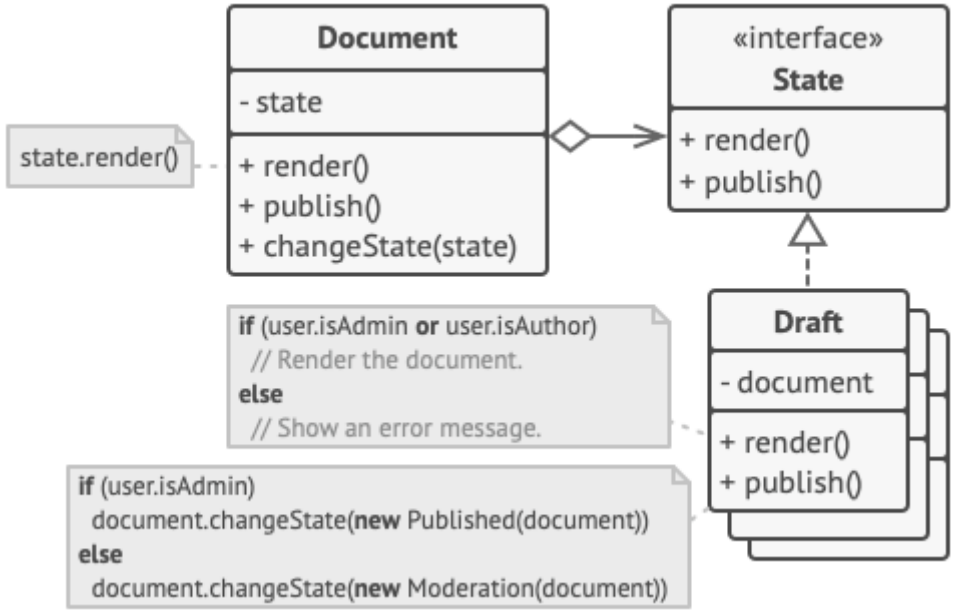
- Naive implementation could include a lot of conditionals:

```
enum DocState {
    DRAFT,
    MODERATION,
    PUBLISHED
}
```

```
DocState myDocState;
void onPublish(String userType) {
    if (myDocState == DocState.DRAFT) {
        if (userType.equals("user")) {
            myDocState = DocState.MODERATION;
            return;
        }
        if (userType.equals("admin")) {
            myDocState = DocState.PUBLISHED;
            return;
        }
    }
    if (myDocState == DocState.MODERATION) ...
}
```

State

- If object states become complicated, conditionals can become spaghetti code
- To avoid this, we can implement states themselves as objects:



State: Advantages

- Single Responsibility: Each State object is responsible for exactly its own behavior
- State transitions and possible states are explicit and clear
  - Adding a new state is easy
- Avoids large conditional statements
- Uses object composition like Bridge