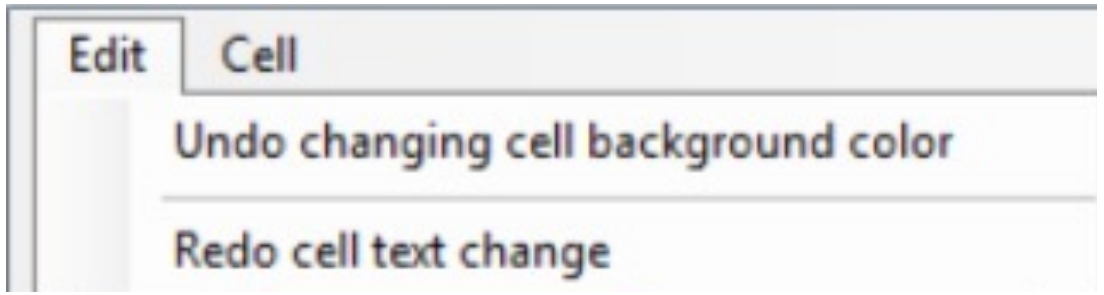
Supporting Undo/Redo in the Spreadsheet Application

Cpt S 321

Washington State University

HW8

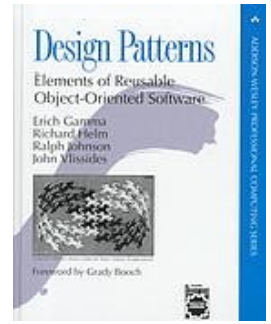
- Two main tasks:
  - Add support for background color for cells
  - Support undo/redo for text and background color changes of cells



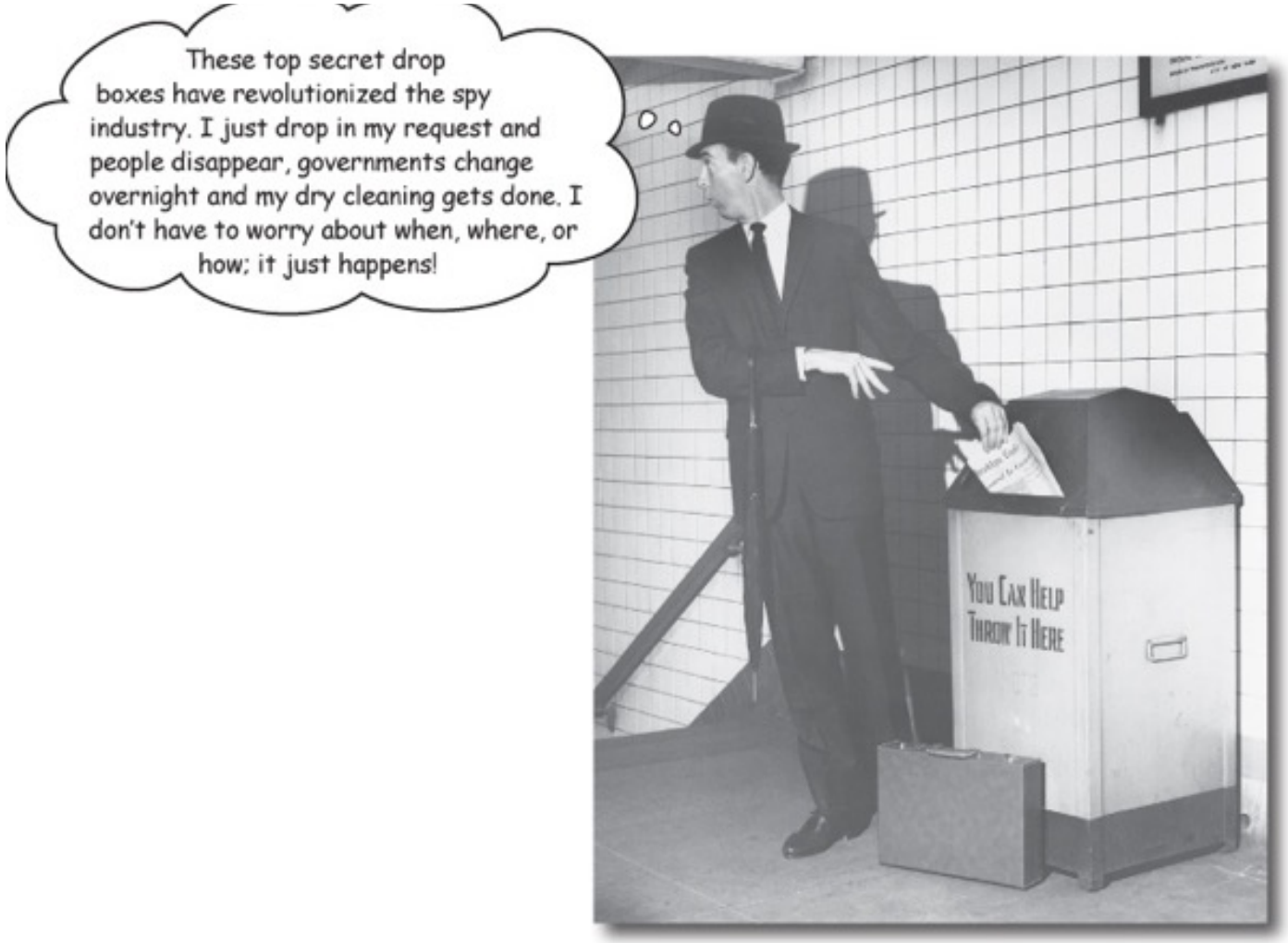
Recall: Software design principles and patterns

- Design principles: general guidelines on how to design applications with a “better” design
- **Design Patterns (DP):** “In software engineering, a **software design pattern** is a general, reusable solution to a commonly occurring problem within a given context in software design.” Wikipedia
- A catalog of Design Patterns was proposed by the “Gang of Four”:

  Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. ISBN 978-0-201-63361-0.



Encapsulating a method invocation?!?



Command Design Pattern (DP) - Intent

- Intent: The **Command** Design Pattern encapsulates a request as an object, thereby letting you parameterize other objects with different requests, queue or log requests, and support undoable operations.

- This kind of operation will be key in building “undo” and “redo” options in an application

- Also known as: Action, Transaction

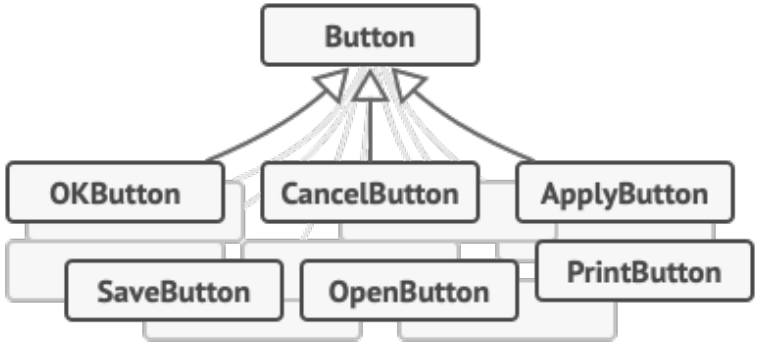
Command DP – Problem

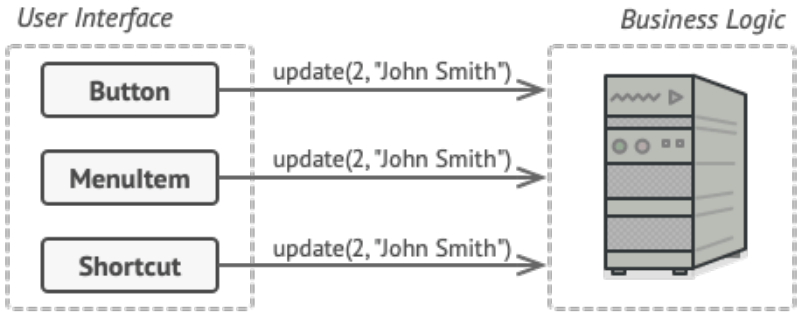
- How to make objects issue requests to other objects without knowing anything about
  1. The operations that will be executed upon the request and
  2. The exact type of the receiver of the request

- Motivational Example:
  - We are designing a GUI library with GUI objects (such Buttons, menus etc.) that will issue requests or carry actions in response to the user input
  - But as designers of this library, we have no way of knowing upfront what those actions are and which objects should will be involved.

Command DP – Problem (cont.)

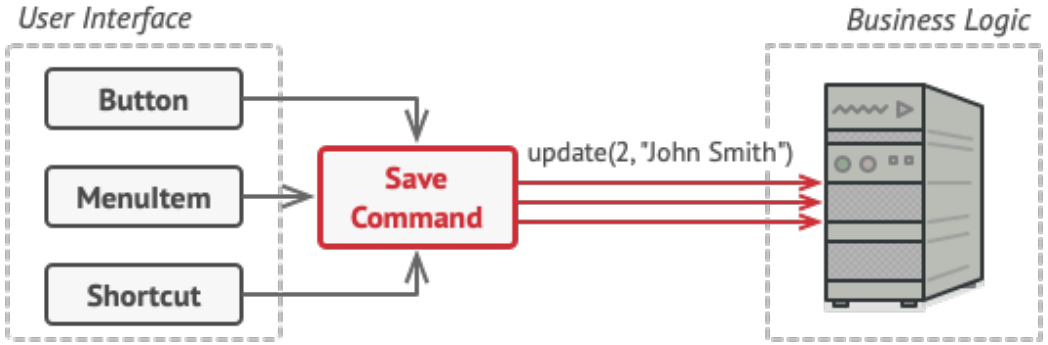
- **Poor solution**: Make tons of classes anticipating what the user would want to do
- Problems with this solution:
  - Cannot foresee all actions
  - The GUI objects are not supposed to implement domain logic
  - We don’t have the knowledge of the application domain this cannot know the objects that will be involved
  - Will end up with duplicated code when the user issues the same request from different places (keyboard and mouse for example)

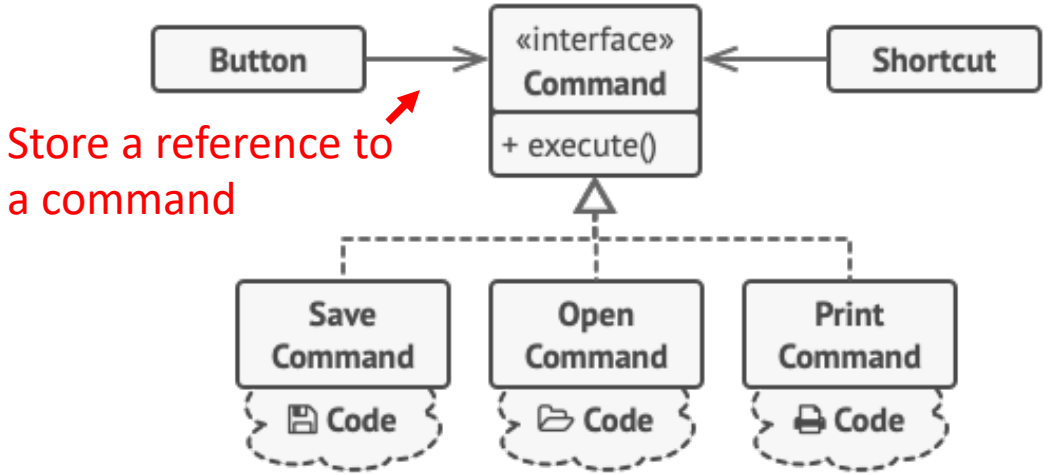




Command DP – Solution

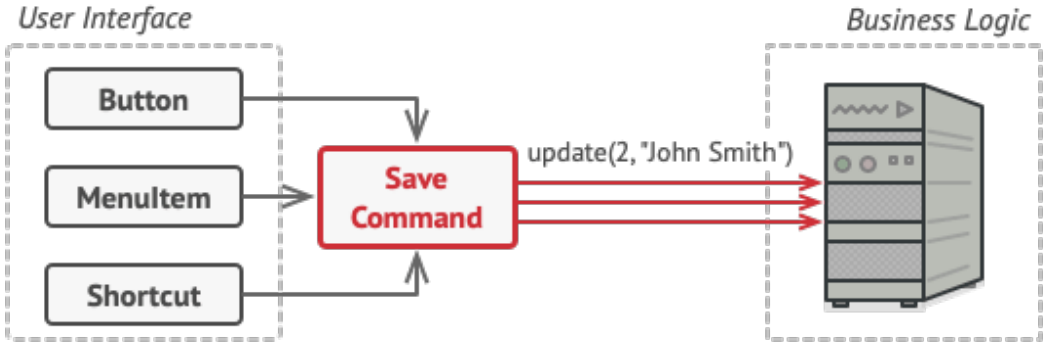
- Good solution:
  - Extract all the necessary information to perform the request in a separate type (*Command*) and make the GUI delegate the work to objects of this type by calling one single method (*execute*) with no parameters
  - Rely on abstraction: let the client code implement the exact way that specific commands should be performed
  - Pre-configure the different commands

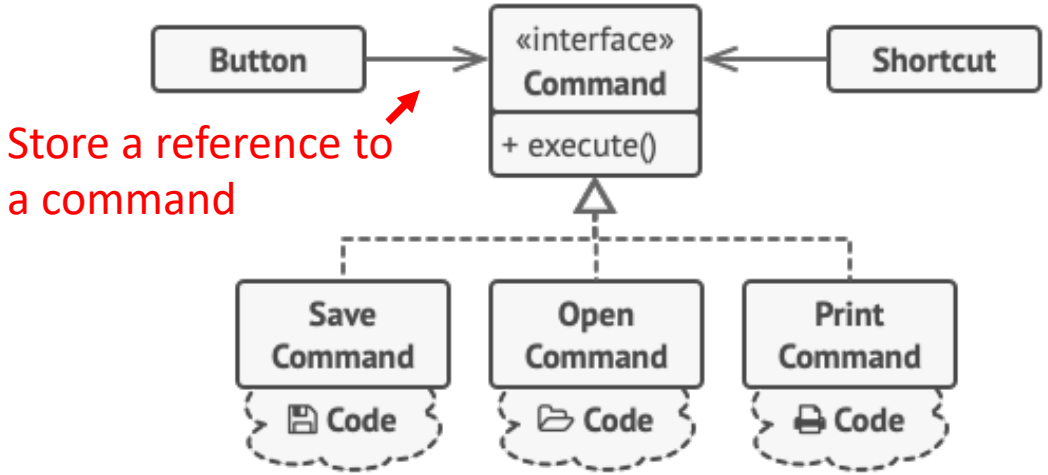




Command DP – Consequences

- No code duplicaton: buttons and shortcuts that issue the same request will execute the same command
- Reduced coupling between the GUI and the Domain Logic
- The GUI does not need the details to the actions and objects that will be involved in the different requests



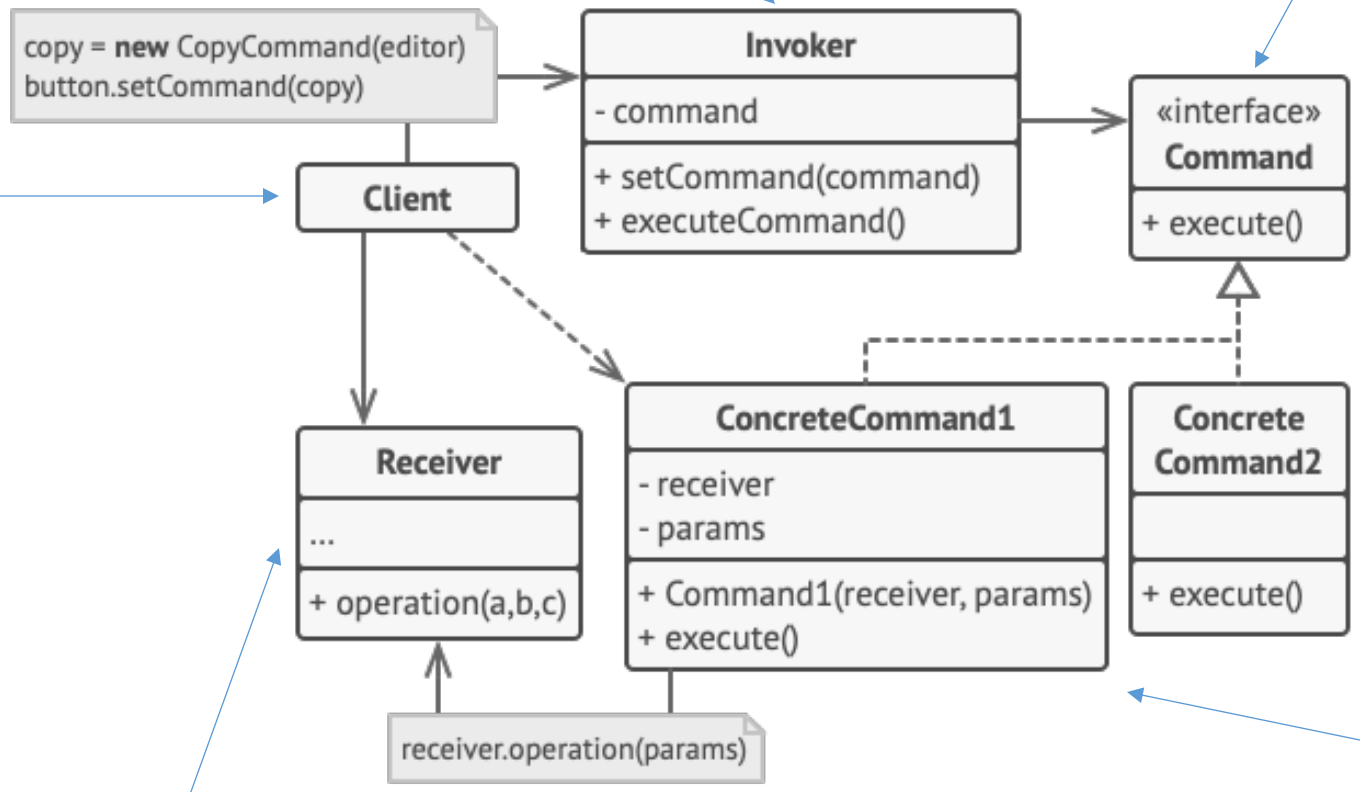


Command DP – Actors

- An object is used to represent and encapsulate all the information needed to call a method at a later time
- This information includes the method name, the object that owns the method, and the values for the method parameters
- Three fundamental Command pattern actors:
  - Client: Instantiates command object and provides information to call the method at a later time
  - Invoker: Executes the Command possibly at a later time. Relies on abstractions.
  - Receiver: The instance of the class that contains the method's code (i.e., the object the command should affect)

1 The **Sender** class (aka *invoker*) is responsible for initiating requests. This class must have a field for storing a reference to a command object. The sender triggers that command instead of sending the request directly to the receiver. Note that the sender isn't responsible for creating the command object. Usually, it gets a pre-created command from the client via the constructor.

2 The **Command** interface usually declares just a single method for executing the command.

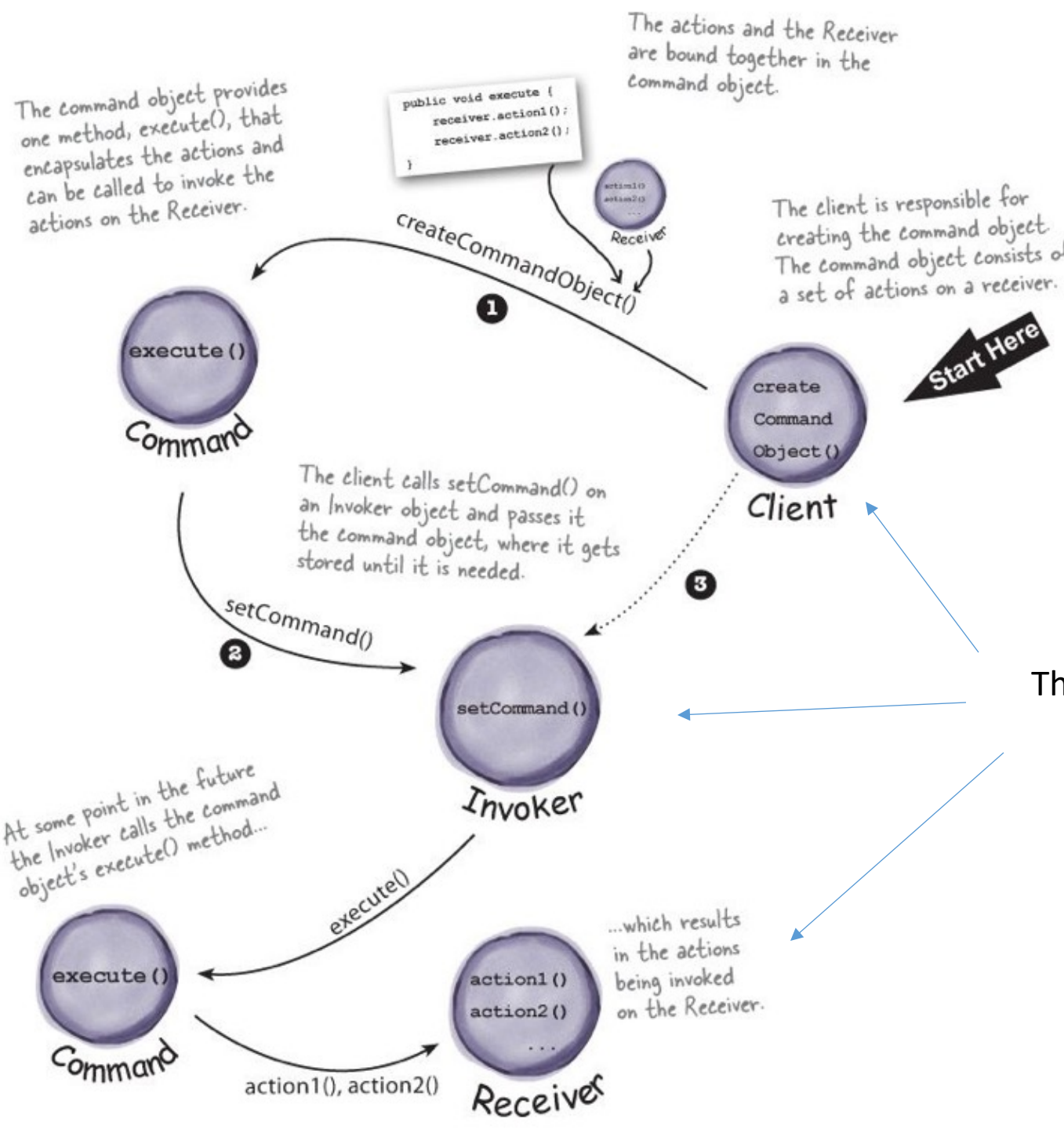


5 The **Client** creates and configures concrete command objects. The client must pass all of the request parameters, including a receiver instance, into the command's constructor. After that, the resulting command may be associated with one or multiple senders.

3 **Concrete Commands** implement various kinds of requests. A concrete command isn't supposed to perform the work on its own, but rather to pass the call to one of the business logic objects. However, for the sake of simplifying the code, these classes can be merged.

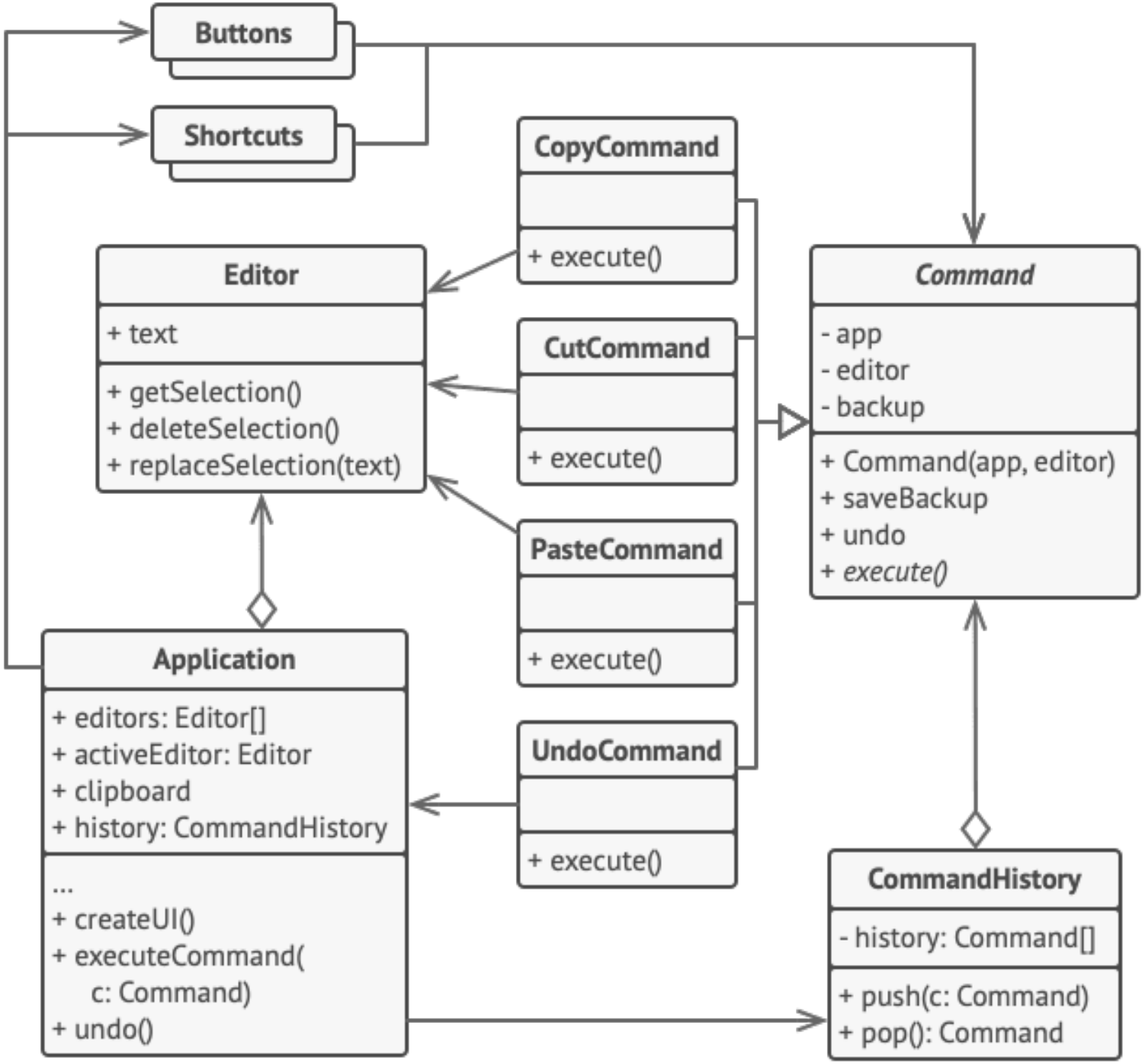
Parameters required to execute a method on a receiving object can be declared as fields in the concrete command. You can make command objects immutable by only allowing the initialization of these fields via the constructor.

4 The **Receiver** class 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, while the receiver itself does the actual work.

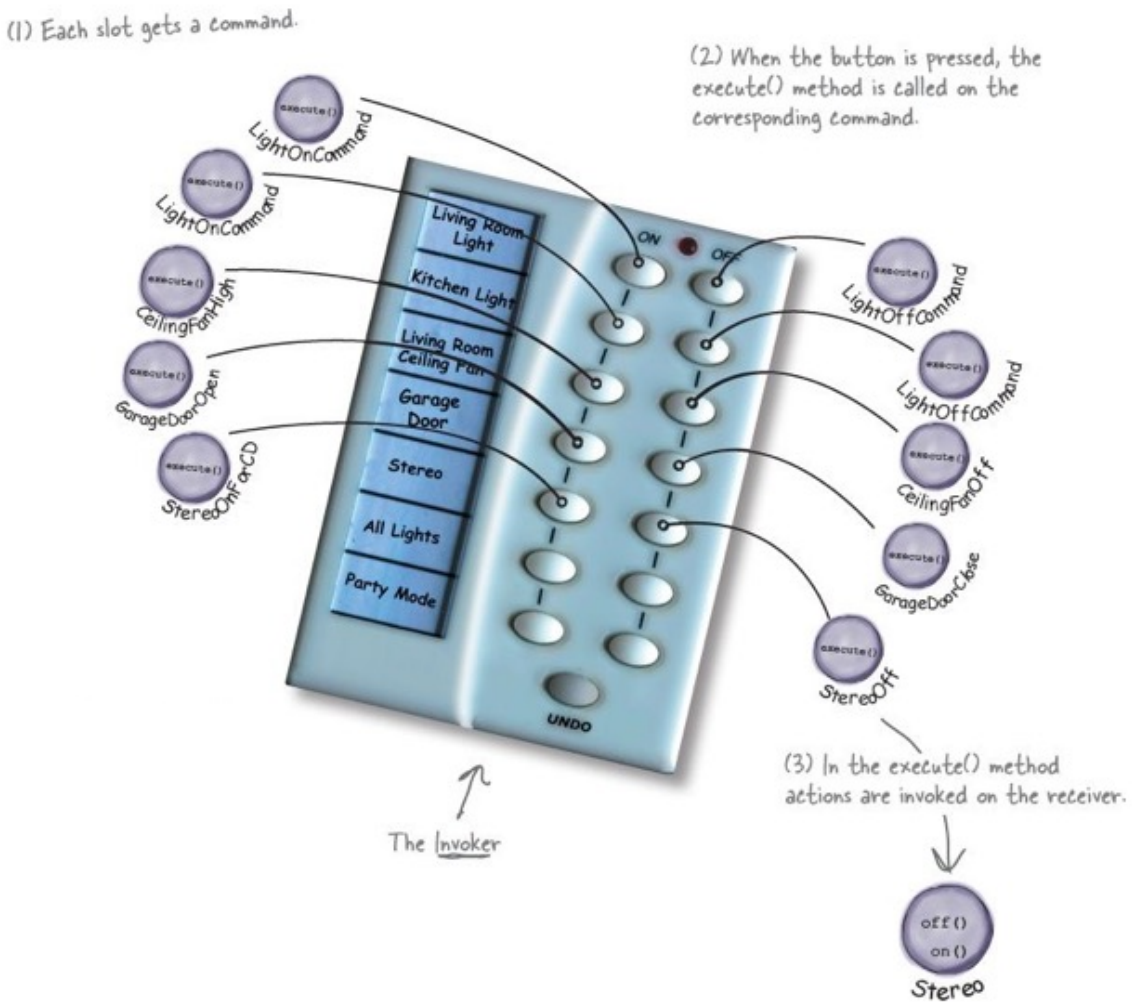


The main actors

Command DP for the motivating example



Coding exercise! (Open laptop and start IDE!)



- Implement a remote control using the Command pattern:
  - RemoteLoader: client
  - RemoteControl: invoker
  - Light, CeilingFan, etc.: receivers
- Each slot is assigned a command (define a controller with 7 slots as shown here)
- **Let's start by implementing only one type of receiver: Light**

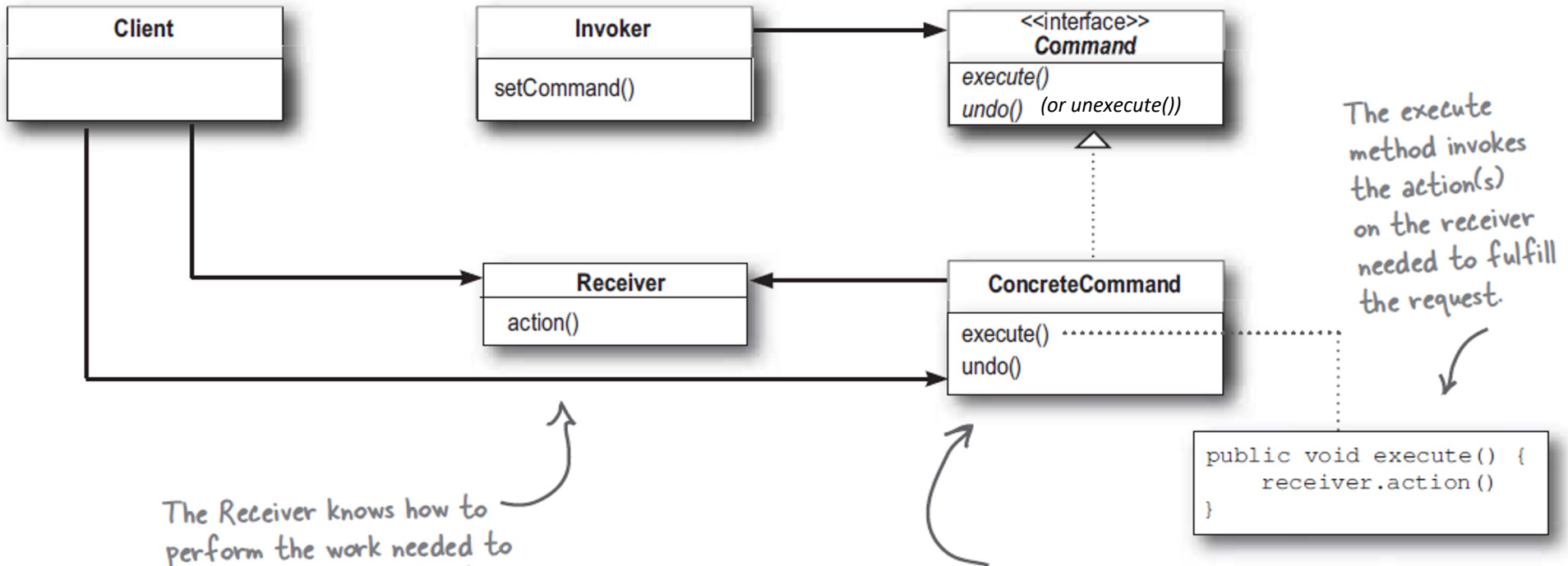
Remember

- The *Client* creates a *ConcreteCommand* object and specifies its *Receiver*
- An *Invoker* object stores the *commands*
- The *Invoker* issues a request by calling execute( ) on the command
- The *ConcreteCommand* object invokes operations on its *Receiver* to carry out the request

The Client is responsible for creating a ConcreteCommand and setting its Receiver.

The Invoker holds a command and at some point asks the command to carry out a request by calling its execute() method.

Command declares an interface for all commands. As you already know, a command is invoked through its execute() method, which asks a receiver to preform an action. You'll also notice this interface has an undo() method.

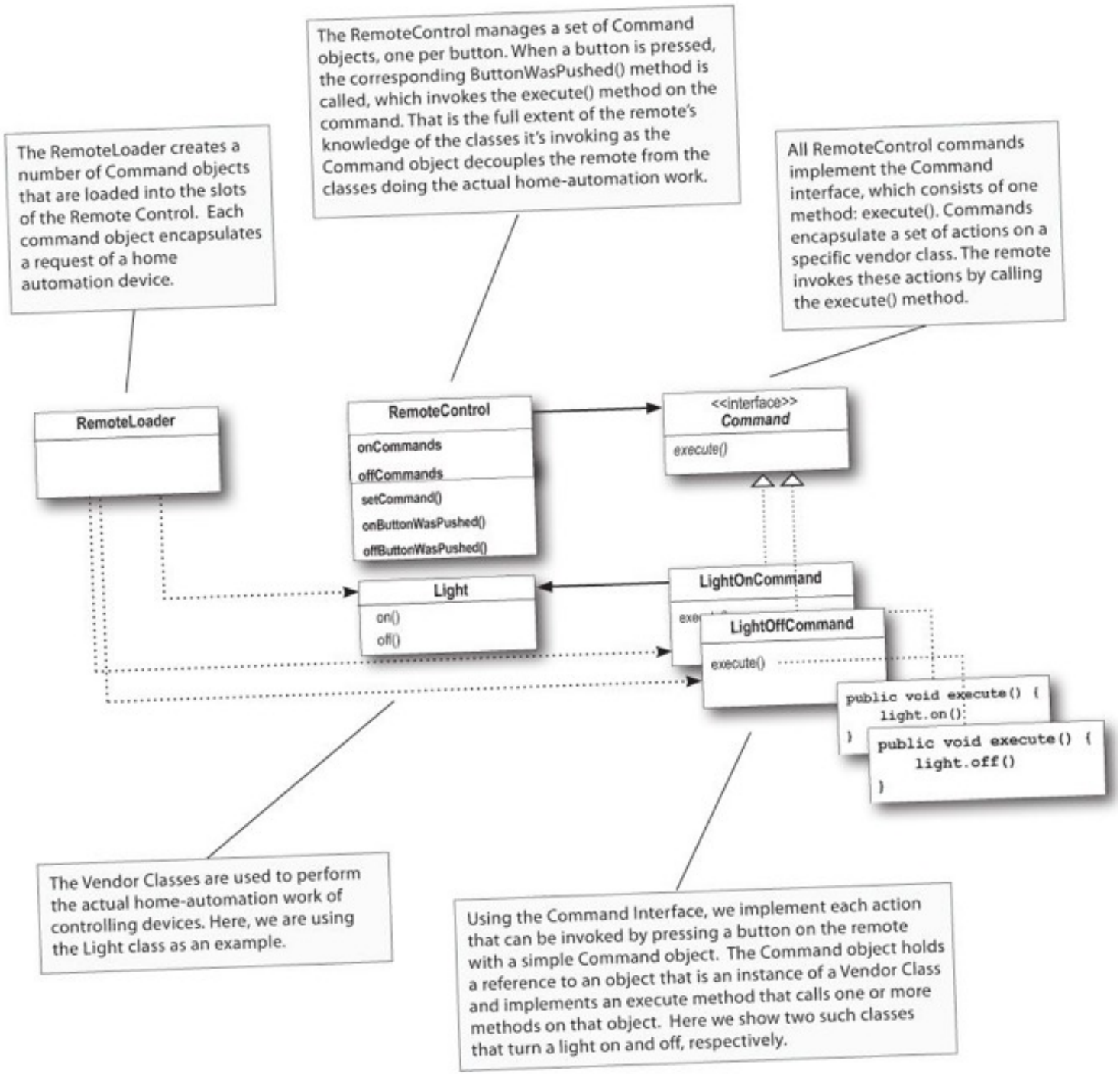


The execute method invokes the action(s) on the receiver needed to fulfill the request.

The Receiver knows how to perform the work needed to carry out the request. Any class can act as a Receiver.

The ConcreteCommand defines a binding between an action and a Receiver. The Invoker makes a request by calling execute() and the ConcreteCommand carries it out by calling one or more actions on the Receiver.

Lights…


The RemoteControl manages a set of Command objects, one per button. When a button is pressed, the corresponding ButtonWasPushed() method is called, which invokes the execute() method on the command. That is the full extent of the remote's knowledge of the classes it's invoking as the Command object decouples the remote from the classes doing the actual home-automation work.
The RemoteLoader creates a number of Command objects that are loaded into the slots of the Remote Control. Each command object encapsulates a request of a home automation device.
All RemoteControl commands implement the Command interface, which consists of one method: execute(). Commands encapsulate a set of actions on a specific vendor class. The remote invokes these actions by calling the execute() method.
RemoteLoader
RemoteControl
onCommands
offCommands
setCommand()
onButtonWasPushed()
offButtonWasPushed()
<<interface>>
Command
execute()
Light
on()
off()
LightOnCommand
LightOffCommand
execute()
public void execute() {
light.on();
}
public void execute() {
light.off()
}
The Vendor Classes are used to perform the actual home-automation work of controlling devices. Here, we are using the Light class as an example.
Using the Command Interface, we implement each action that can be invoked by pressing a button on the remote with a simple Command object. The Command object holds a reference to an object that is an instance of a Vendor Class and implements an execute method that calls one or more methods on that object. Here we show two such classes that turn a light on and off, respectively.

Even further: undo!

- We want to add functionality to support the **undo** button on the remote.

- Suppose that the Living Room Light is off and you press the on button on the remote. Obviously the light turns on. Now if you press the undo button then the last action will be reversed in this case, the light will turn off.

Command pattern and undo

- When commands support undo, they have an *undo()* (or *unexecute()*) method that mirrors the execute() method. Whatever execute() last did, undo() reverses. So, before we can add undo to our commands, we need to add an undo() method to the Command interface.
- Then, implement the undo() (or unexecute()) method in the LightOnCommand and LightOffCommand. The method has to do the exact opposite of the execute() method.
- Add a field undoCommand in the RemoteControl. Don’t forget to set it every time a button is pushed
- Add a method undoButtonPushed() in the RemoteControl

Command pattern summary

- Two important aspects of the Command pattern:
  - interface separation (the invoker is isolated from the receiver)
  - time separation (stores a ready-to-go processing request that's to be stated later)
- Easily change Commands without changing existing classes
- Commands are first-class objects. They can be manipulated and extended like any other object
- Allows for a history of commands to be kept (undo & redo)
- Can assemble multiple Commands into composite commands, like Macros/ Transactions

References

- Freeman, Eric,Robson, Elisabeth,Bates, Bert,Sierra, Kathy. Head First Design Patterns: A Brain-Friendly Guide (Kindle Location 3107). O'Reilly Media.
- Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. ISBN 978-0-201-63361-0.
- Refactoring Guru: https://refactoring.guru/design-patterns/command