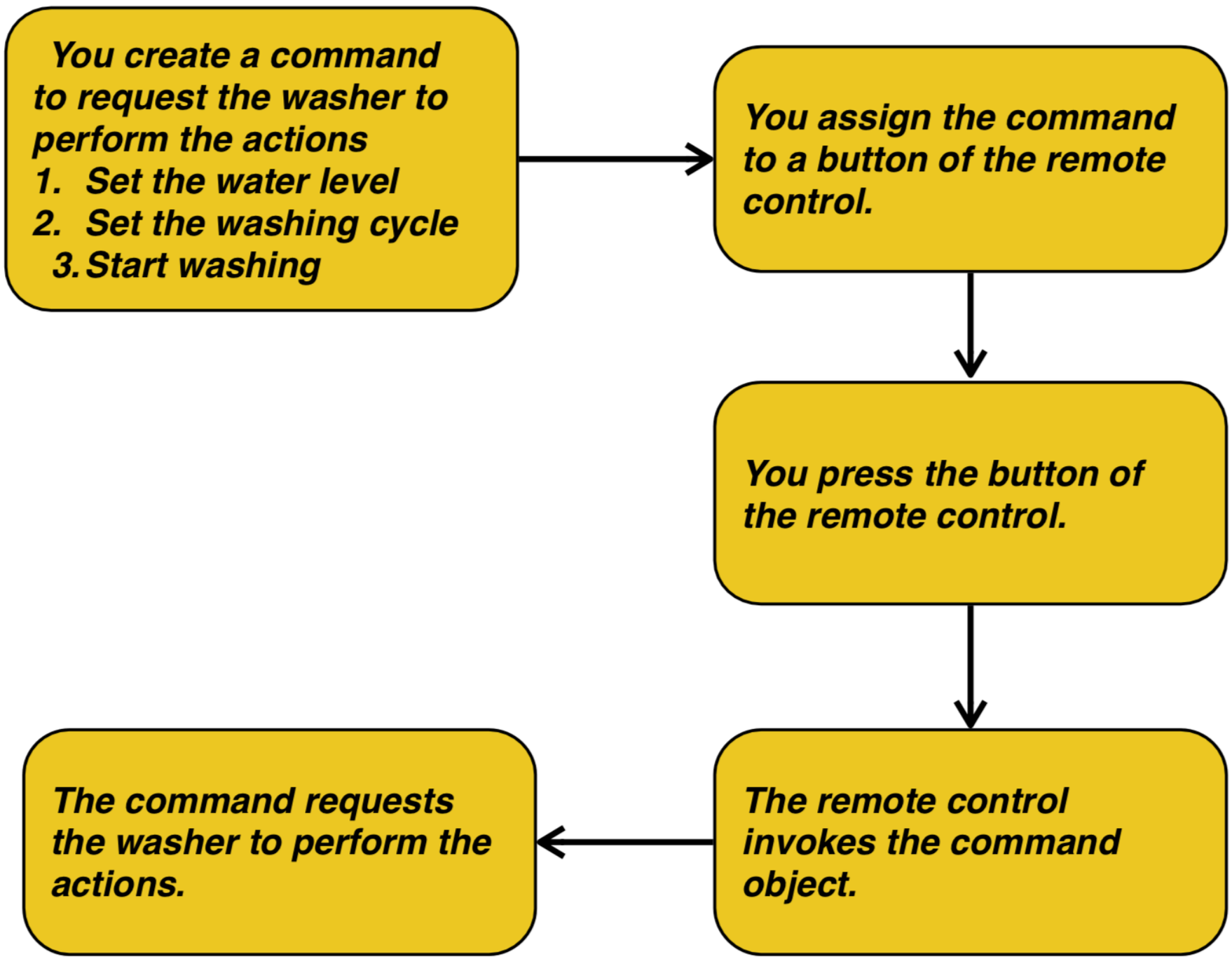
Command Pattern

Introduction

- This will encapsulate the invocation of the method. In doing so, we are able to make the computation clear and accurate. The object invoking the computation does not have to worry about the way things are done, it simply uses our well-defined method to do it.

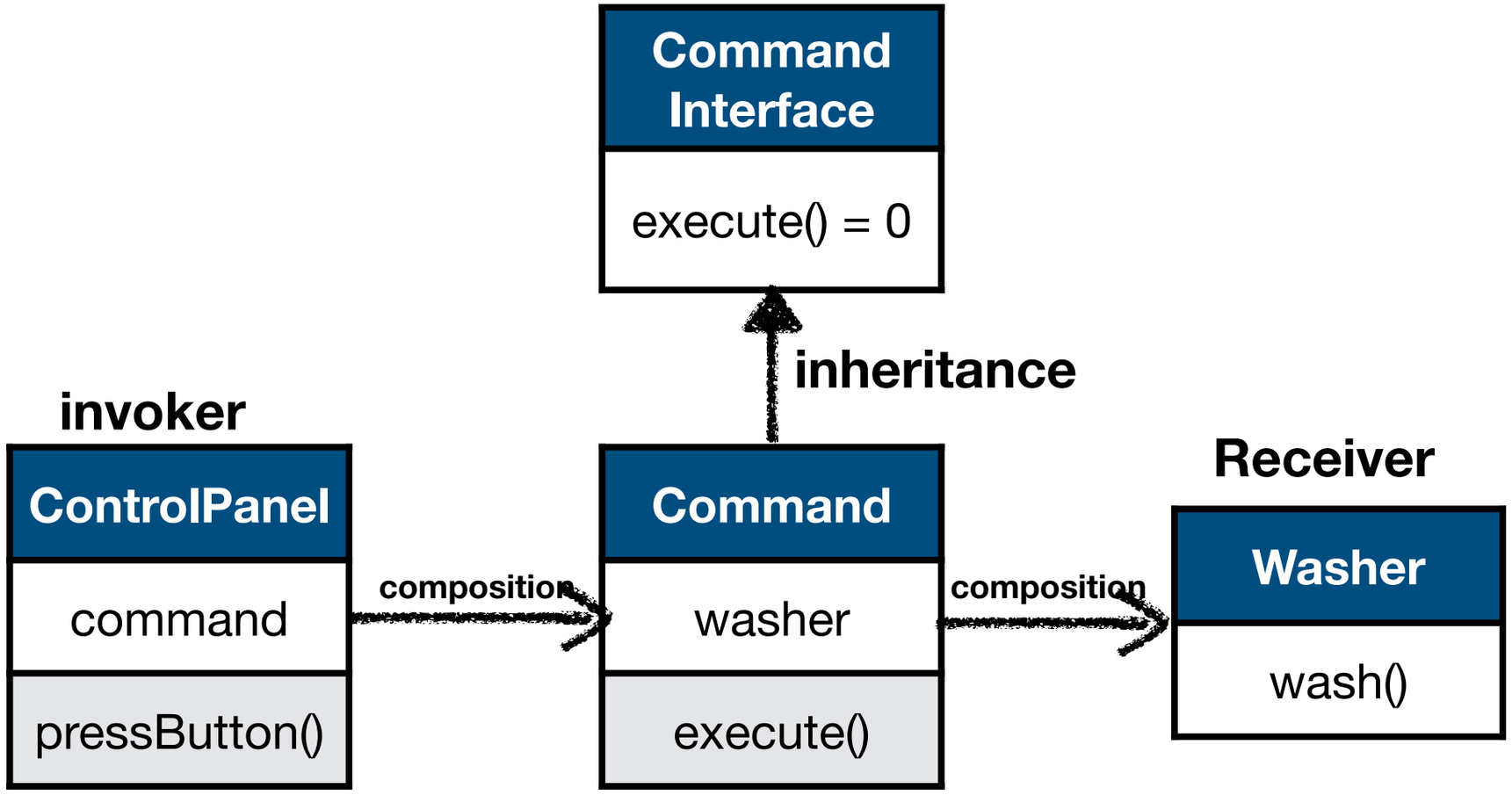
- Let's say you have to design a universal remote control with 5 sets of ON/OFF buttons. You will use it to control all of your household appliances. For example, television, washing machine, dryer, ceiling fan and lights. Each electrical appliance is an instance of different classes. They have their own methods to enable users to use them. For example, a washer class can provide methods to regulate the water level, the washing time, therefore not only on and off operations. On the other hand, the light class is the most straightforward, which has only on and off method. As a result, your job is to design a method for decoupling the remote control and device classes..

The Flow Chart of Command Request



The Command Pattern

- The Command Pattern allows you to decouple the invoker (The Control Panel) of an action from the object (e.g. Washer) that actually performs the action. We use a command object that encapsulates a request to do something on a specific object (receiver). So, we store a command object for each button, when the button is pressed we ask the command object to do some work.



The Workflow of the Command Pattern

1. The client is responsible for creating the command object consists of a set of actions on a receiver. So the actions and the receiver are tied together in the control object.
2. The client does a setCommand() to store the command object in the invoker.
3. The client asks the invoker to execute the command.

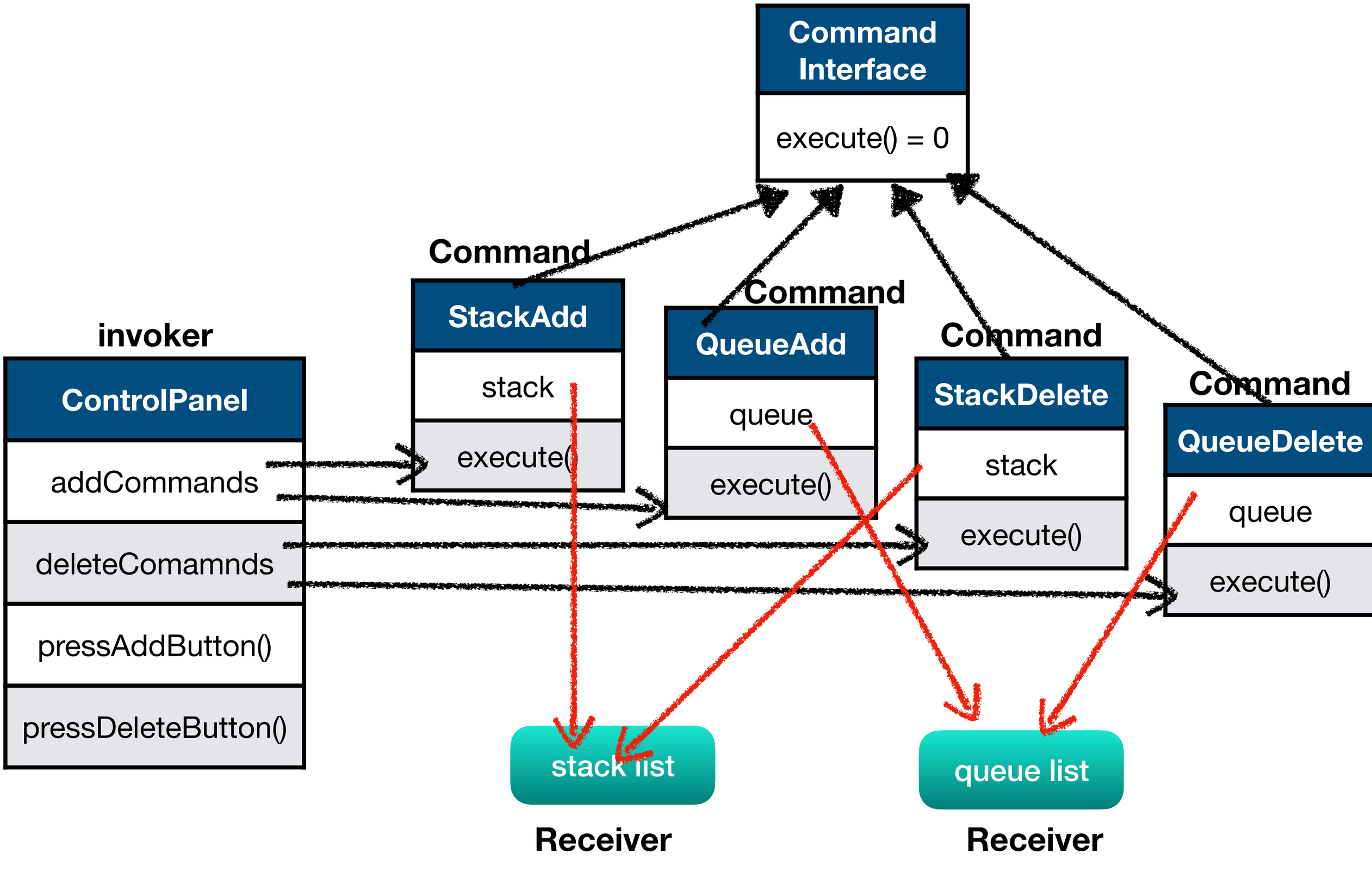
```
class Receiver:
    def doAction1(self):
        print("Action 1 performed")

    def doAction2(self):
        print("Action 2 performed")

class Command:
    def __init__(self, receiver):
        self.receiver = receiver

    def execute(self):
        self.receiver.doAction1()
        self.receiver.doAction2()

class Invoker:
    def __init__(self):
        self.slot = None

    def setCommand(self, command):
        self.slot = command

    def invoke(self):
        self.slot.execute()

def main():
    receiver = Receiver()
    command = Command(receiver)
    invoker = Invoker()
    invoker.setCommand(command)

    # Sometime later the client calls invoker to invoke the
command
    invoker.invoke()

if __name__ == "__main__":
    main()
```

A Control Panel for Adding and Deleting items

- I'll use a similar example. There are many different kinds of data structures. Each data structure has its own method for adding and removing elements. For example, stack has push() and pop() method. And queue has enqueue() and dequeue() method. And we will design a control panel which provides the same interface (Add/Delete button) for the user to perform the Add/Delete on various data structures.

- In Python, we're going to use list to simulate both stack and queue.
  - For stack, it is LIFO order
    - push() - we use append()
    - pop() - we use pop() that removes the last item
  - For queue, it is FIFO order
    - enqueue() - we use append()
    - dequeue() - we use pop(0) that removes the first item

A Control Panel for Adding and Deleting items



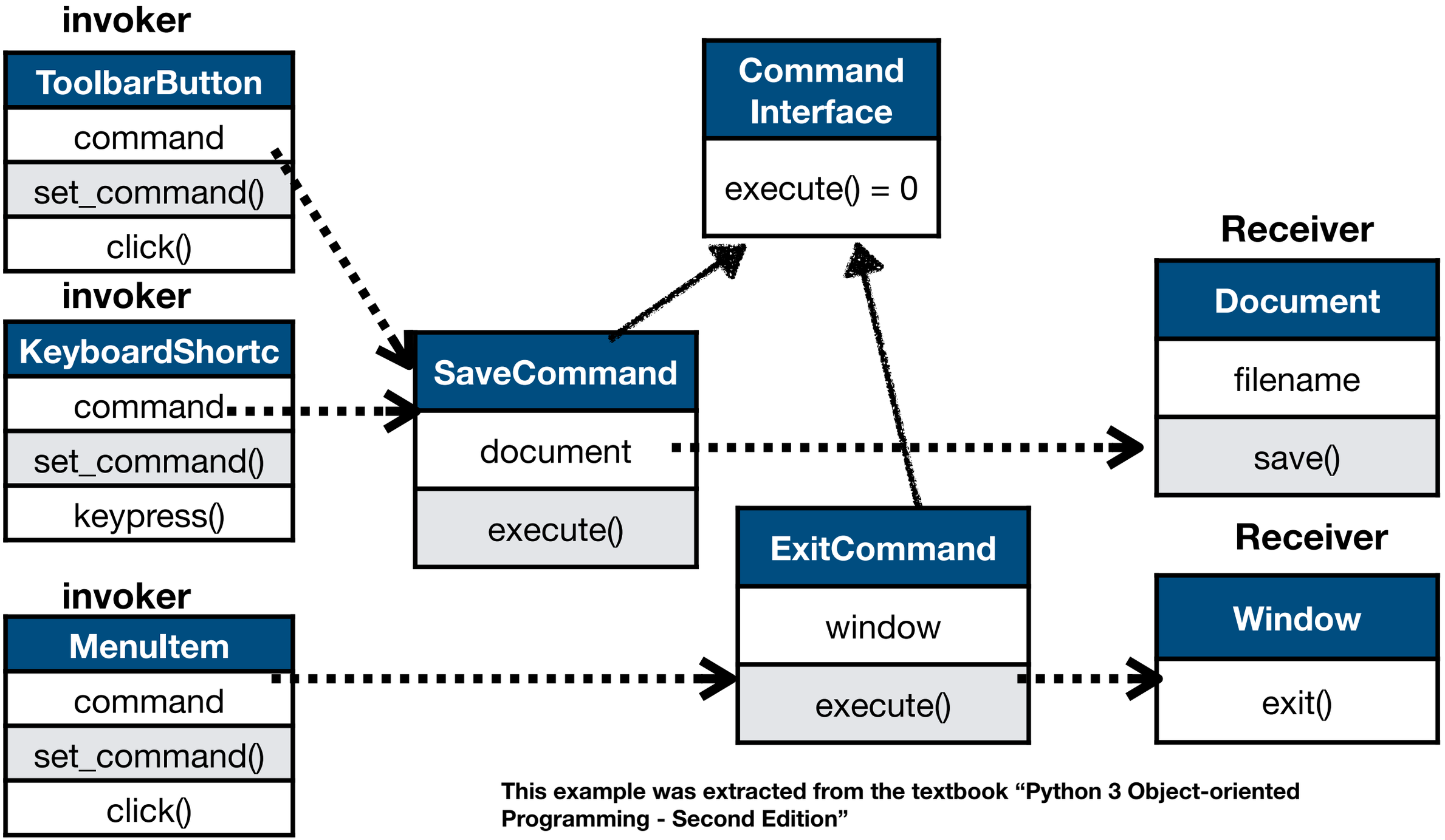
A Control Panel for Adding and Deleting items

```
class Command(ABC):
    @abstractmethod
    def execute(self):
        pass

class StackAddCommand(Command):
    def __init__(self, stack, newItem):
        self.stack = stack
        self.newItem = newItem

    def execute(self):
        self.stack.append(self.newItem)
        print('Pushed', self.newItem)

class StackDeleteCommand(Command):
    def __init__(self, stack):
        self.stack = stack

    def execute(self):
        deletedItem = self.stack.pop()    # delete the last one
        print('Popped', deletedItem)

class QueueAddCommand(Command):
    def __init__(self, queue, newItem):
        self.queue = queue
        self.newItem = newItem

    def execute(self):
        self.queue.append(self.newItem)
        print('Enqeued', self.newItem)

class QueueDeleteCommand(Command):
    def __init__(self, queue):
        self.queue = queue

    def execute(self):
        deletedItem = self.queue.pop(0)   # delete the first one
        print('Dequeued', deletedItem)
```

```
class ControlPanel:
    def __init__(self):
        self.addCommands = []
        self.deleteCommands = []

        noCommand = NoCommand()
        for i in range(5):
            self.addCommands.append(noCommand)
            self.deleteCommands.append(noCommand)

    def setCommand(self, slot, addCommand, deleteCommand):
        self.addCommands[slot] = addCommand
        self.deleteCommands[slot] = deleteCommand

    def pressAddButton(self, slot):
        self.addCommands[slot].execute()

    def pressDeleteButton(self, slot):
        self.deleteCommands[slot].execute()
```

Case Study: Command pattern used in Graphical Window system

- A common example of the command pattern is actions on a graphical window. Often, an action can be invoked by a menu item on the menu bar, a keyboard shortcut, a toolbar icon, or a context menu.



This example was extracted from the textbook "Python 3 Object-oriented Programming - Second Edition"

Case Study: Command pattern used in Graphical Window system

invoker

```
class ToolbarButton:
    def __init__(self, name, iconname):
        self.name = name
        self.iconname = iconname

    def set_command(self, command):
        self.command = command

    def click(self):
        print("The button was clicked")
        self.command.execute()


class MenuItem:
    def __init__(self, menu_name, menuitem_name):
        self.menu = menu_name
        self.item = menuitem_name

    def set_command(self, command):
        self.command = command

    def click(self):
        print("The menu item was clicked")
        self.command.execute()


class KeyboardShortcut:
    def __init__(self, key, modifier):
        self.key = key
        self.modifier = modifier

    def set_command(self, command):
        self.command = command

    def keypress(self):
        print("The key was pressed")
        self.command.execute()
```

Command

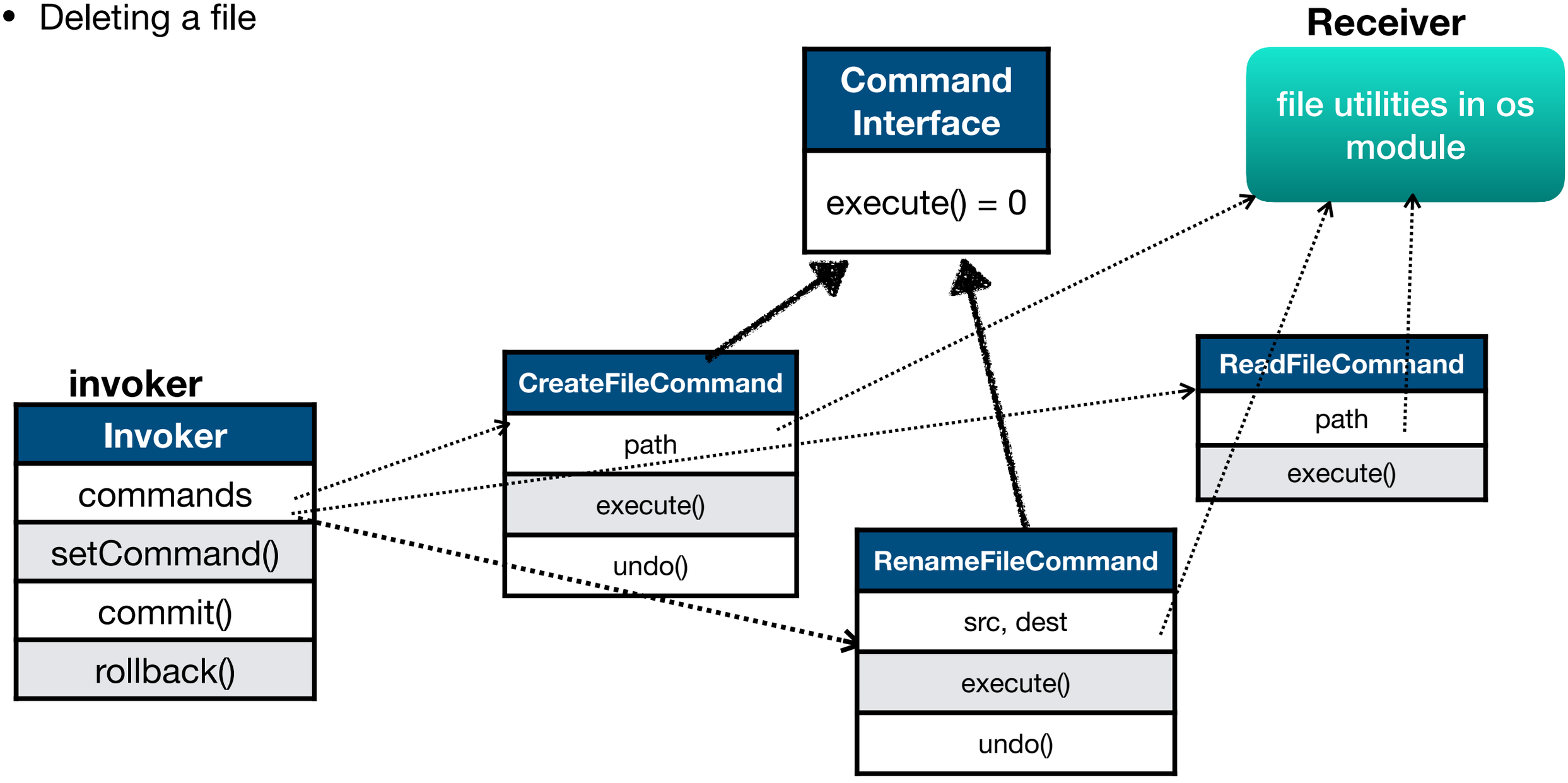
```
class Command(ABC):
    @abstractmethod
    def execute(self):
        pass

class SaveCommand(Command):
    def __init__(self, document):
        self.document = document

    def execute(self):
        self.document.save()


class ExitCommand(Command):
    def __init__(self, window):
        self.window = window

    def execute(self):
        self.window.exit()
```

Receiver

```
class Window:
    def exit(self):
        sys.exit(0)

class Document:
    def __init__(self, filename):
        self.filename = filename
        self.contents = "This file cannot
be modified"

    def save(self):
        with open(self.filename, 'w') as
file:
            file.write(self.contents)
```

Case Study - File Operations

- In this example, we will use the Command pattern to implement the most basic file utilities:
- Creating a file and optionally writing text (a string) to it
- Reading the contents of a file
- Renaming a file
- Deleting a file



This example was extracted from the book "Mastering Python Design Patterns - Second Edition"

Case Study: Command pattern used in Graphical Window system

invoker

```
class Invoker:
    def __init__(self):
        self.commands = []

    def setCommand(self, command):
        self.commands.append(command)

    def commit(self):
        [c.execute() for c in self.commands]

    def rollback(self):
        for c in reversed(self.commands):
            try:
                c.undo()
            except AttributeError as e:
                print("Error", str(e))
```

Command

```
class Command(ABC):
    @abstractmethod
    def execute(self):
        pass

class CreateFileCommand(Command):
    def __init__(self, path, txt='hello world\n'):
        self.path = path
        self.txt = txt

    def execute(self):
        if verbose:
            print(f"[creating file '{self.path}']")
        with open(self.path, mode='w', encoding='utf-8') as out_file:
            out_file.write(self.txt)

class ReadFileCommand(Command):
    def __init__(self, path):
        self.path = path

    def execute(self):
        if verbose:
            print(f"[reading file '{self.path}']")
        with open(self.path, mode='r', encoding='utf-8') as in_file:
            print(in_file.read(), end='')


class RenameFileCommand(Command):
    def __init__(self, src, dest):
        self.src = src
        self.dest = dest

    def execute(self):
        if verbose:
            print(f"[renaming '{self.src}' to '{self.dest}']")
        os.rename(self.src, self.dest)
```

Other use of Command Pattern - Queuing Requests

- The commands give us a way to put together a set of actions and assign them to the Invoker. The actual execution of the command object can be called long after the creation of the command object by the client. In fact, it can even be invoked by some other thread.
- For instance, you can add commands to a job queue on one side, and on the other side sit down a thread pool. Threads can perform the following.:

1. Remove a command from the queue
2. Call the command's execute()
3. Wait for the call to finish
4. Discard the command object and retrieve a new one.

Another use of the Command Pattern - Logging Requests

- Some apps require us to record all actions and be able to receive them after a crash by re-inventing those actions. It is possible to change the Command interface by adding two methods: store() and load().
- As we execute commands, we store historical records on disk. When a crash occurs, we reload command objects and call their execute() methods by batch and in sequence.

Summary

- The Command Pattern decouples an object, by asking the one (receiver) who knows how to run it.

- A command object lies in the center of this decoupling and encapsulates a receiver with an action (or set of actions)

- A Invoker requests a Command object by calling its execute() method that invokes these actions on the receiver.