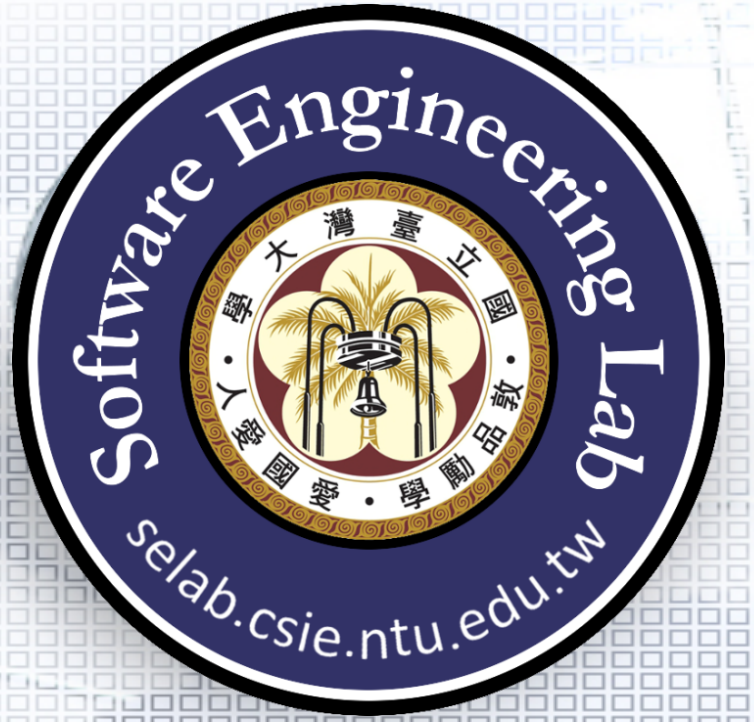


Command Pattern

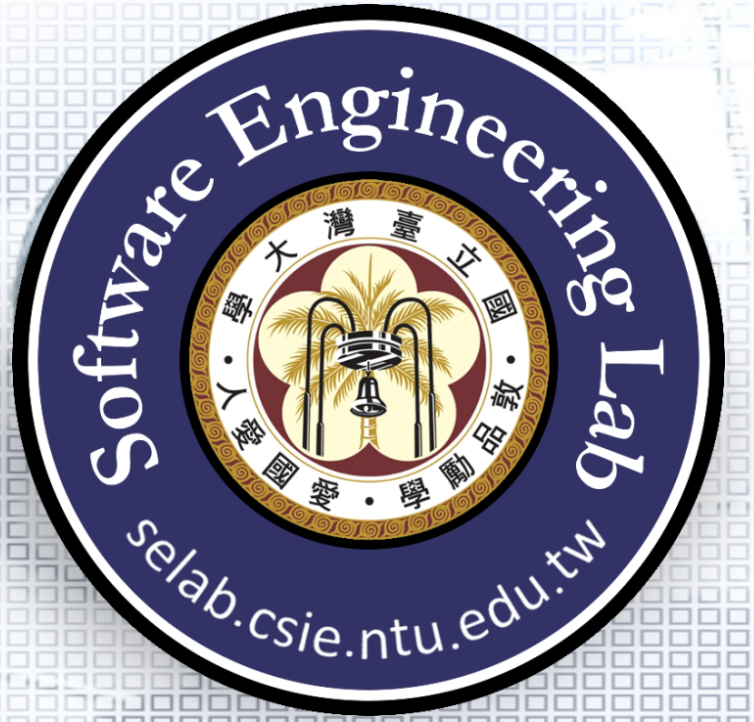
Prof. Jonathan Lee（李允中）
Department of CSIE
National Taiwan University

Design Aspect of Command

When and how a request is fulfilled

Outline

- Cut, Copy, Paste on a Document Requirements Statements
- Initial Design and Its Problems
- Design Process
- Refactored Design after Design Process
- Recurrent Problems
- Intent
- Command Pattern Structure
- Remote Control: Another Example



Cut, Copy, Paste on a Document

Prof. Jonathan Lee (李允中)

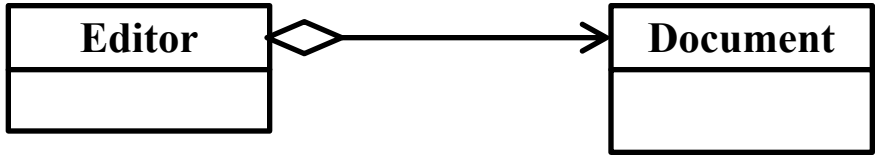
Department of Computer Science and Information Engineering
National Taiwan University

Requirements Statements

- An editor application carries a document.
- A menu in the editor application contains several menu items performing three specific operations: cut, copy and paste on a document when clicked.

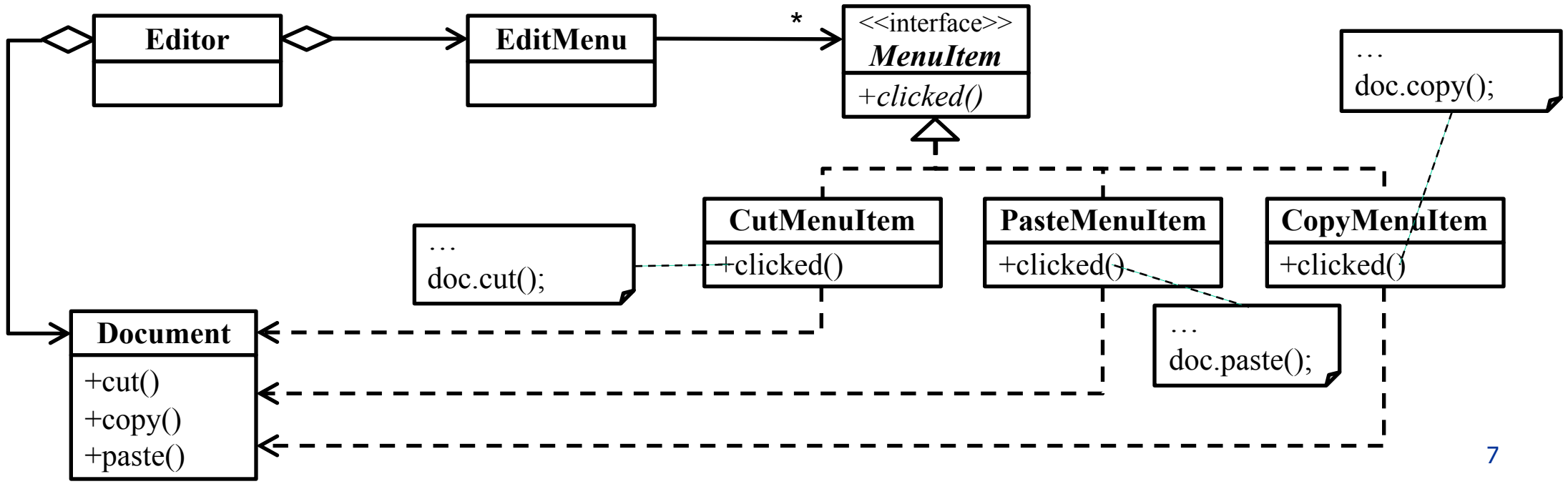
Requirements Statements$_1$

- An editor application carries a document.

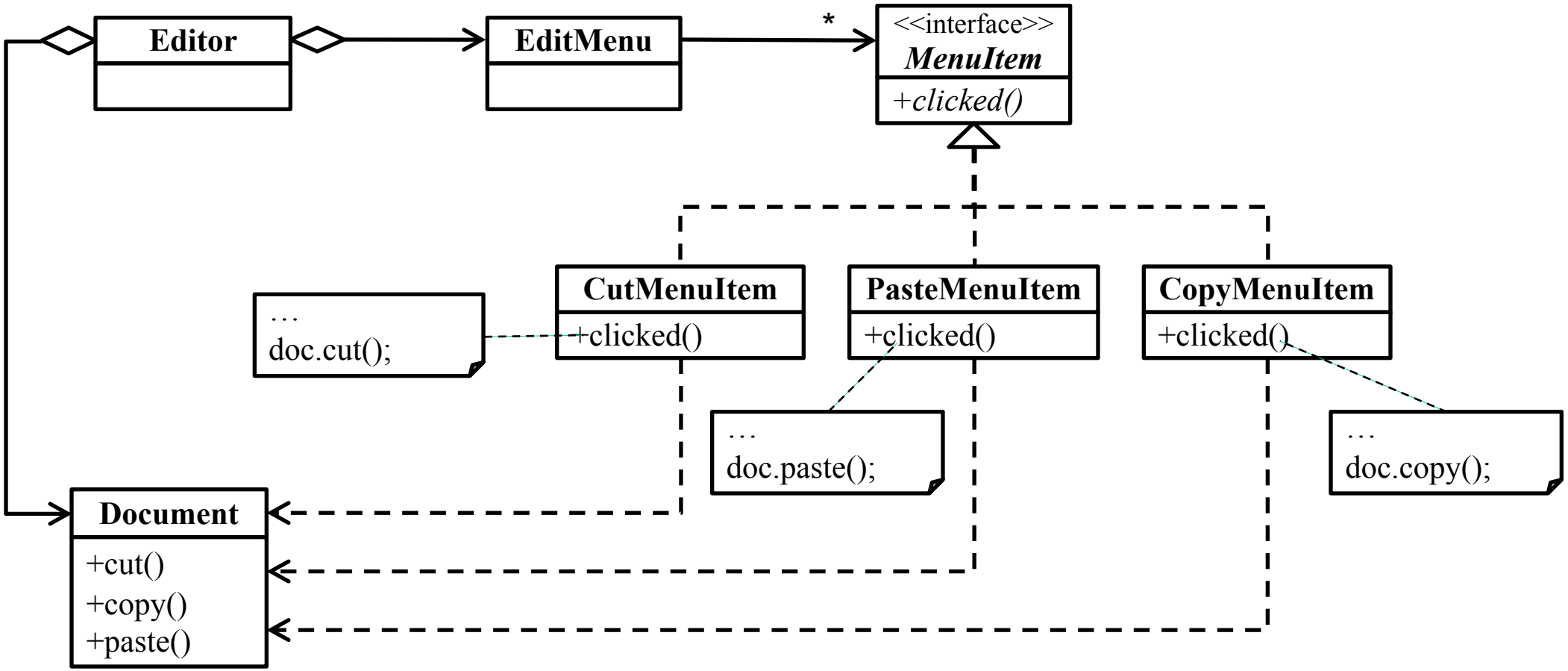


Requirements Statements$_2$

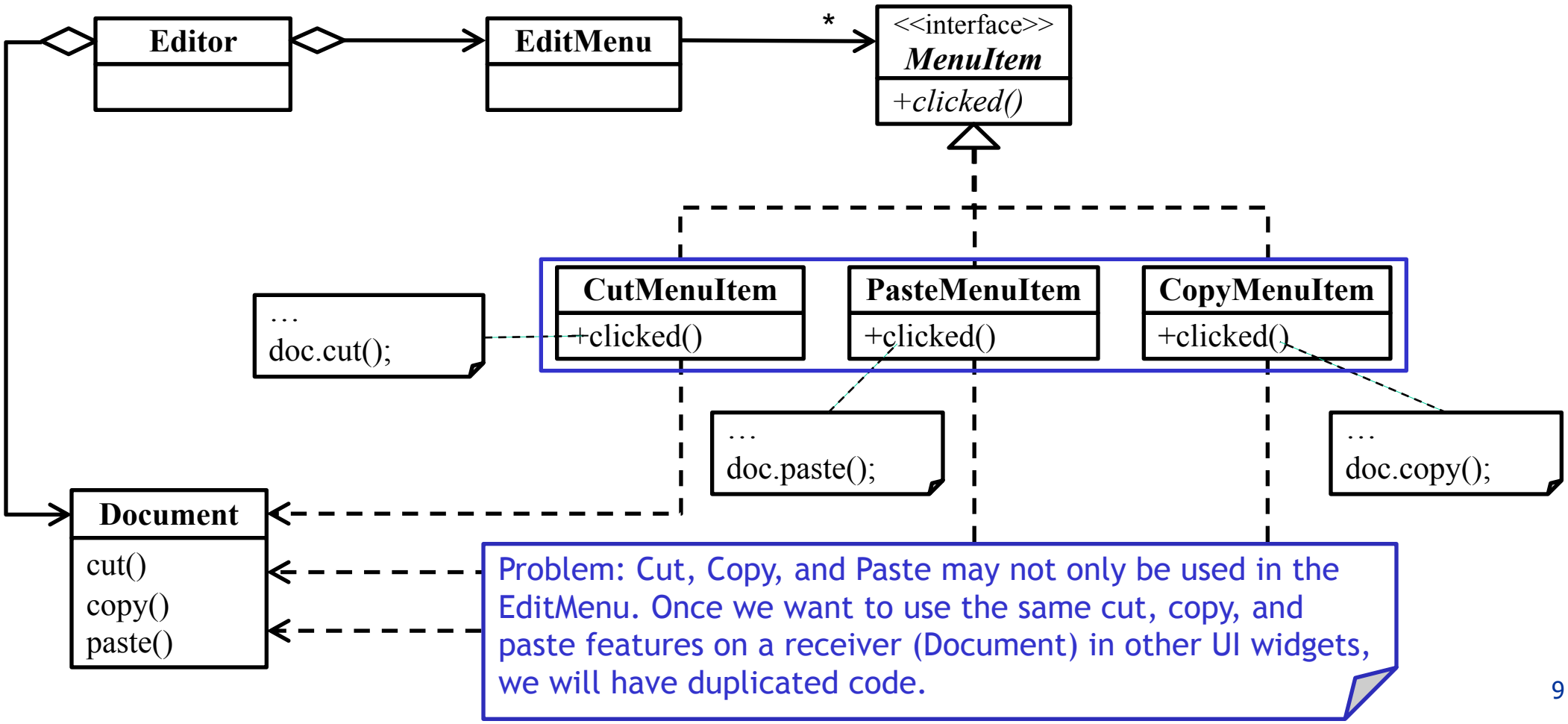
- A menu in the editor application contains several menu items performing three specific operations: cut, copy and paste on a document when clicked.



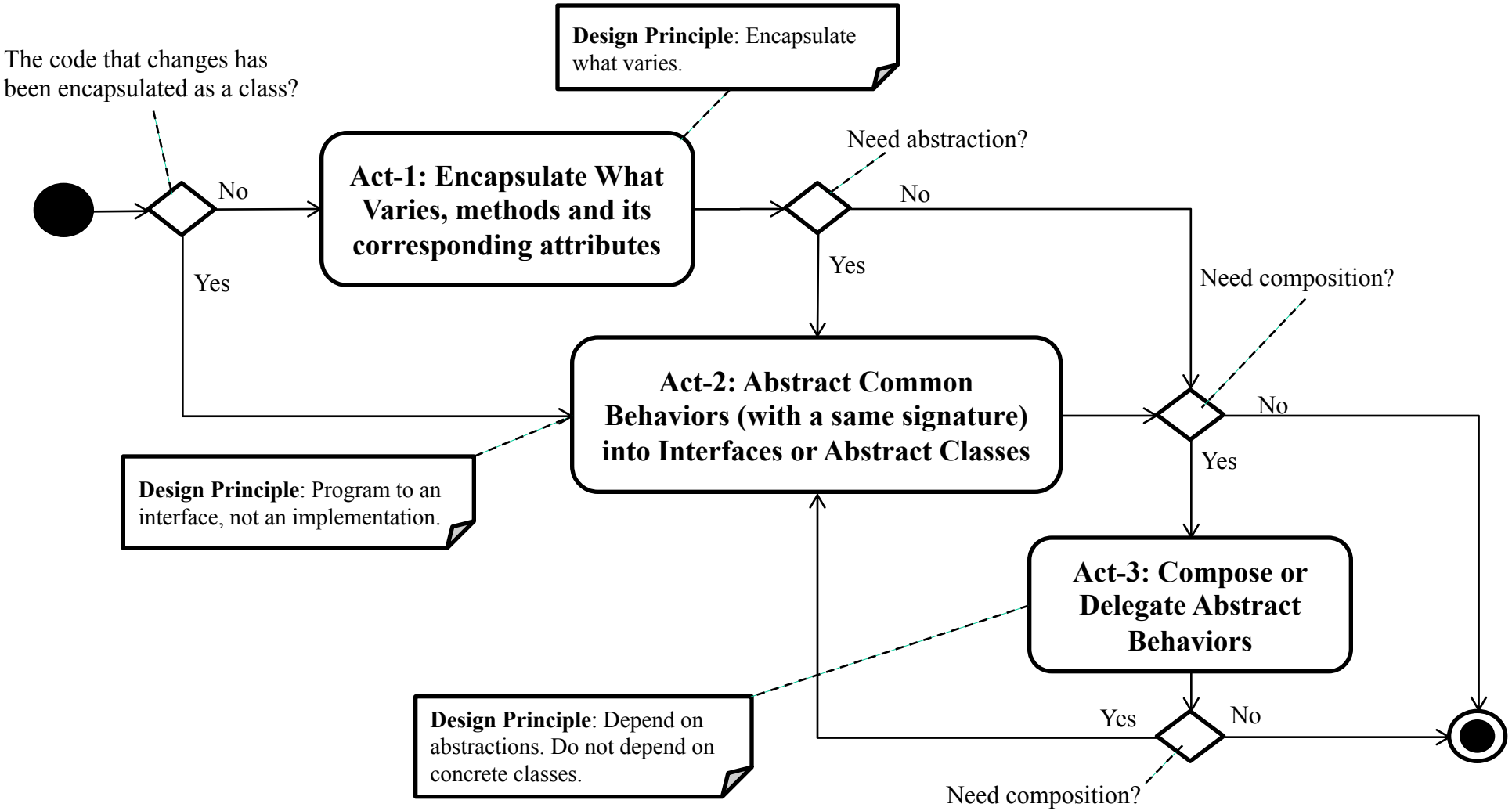
Initial Design



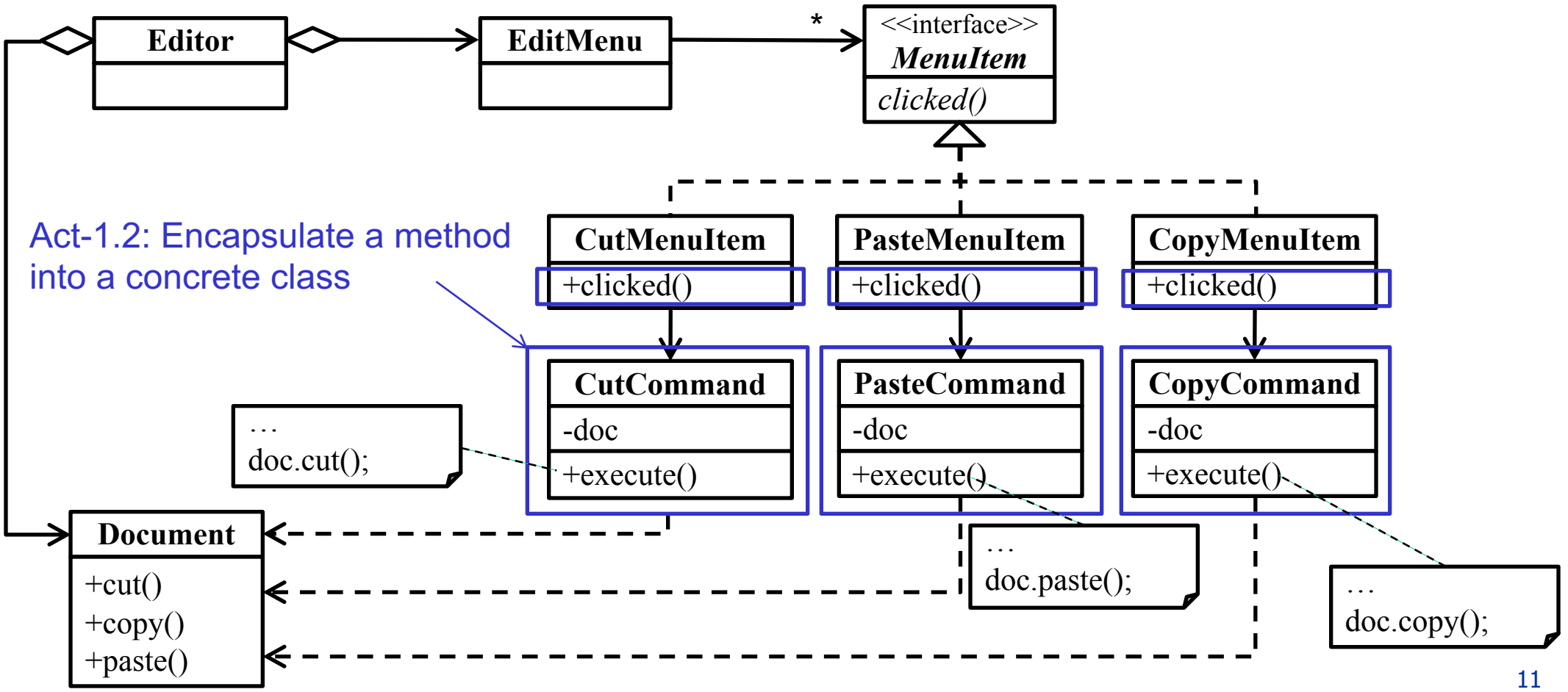
Problems with Initial Design



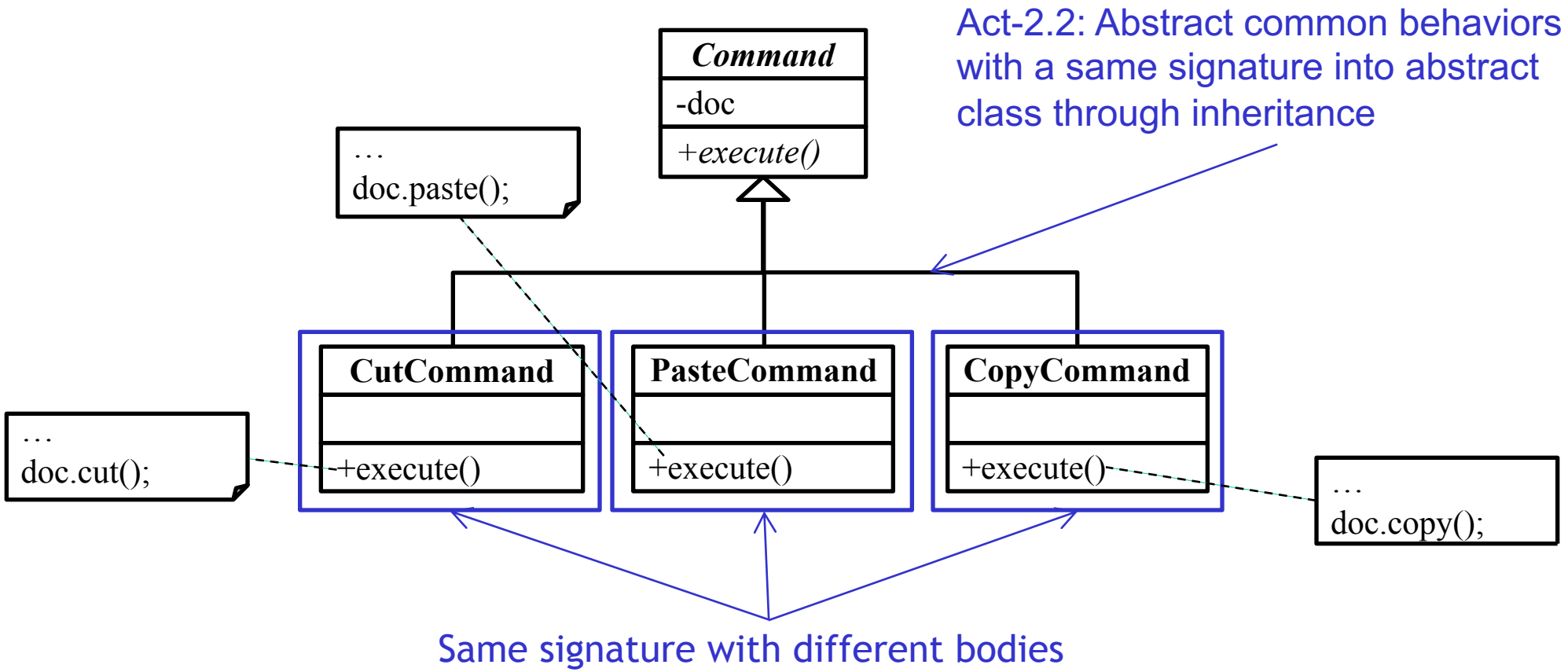
Design Process for Change



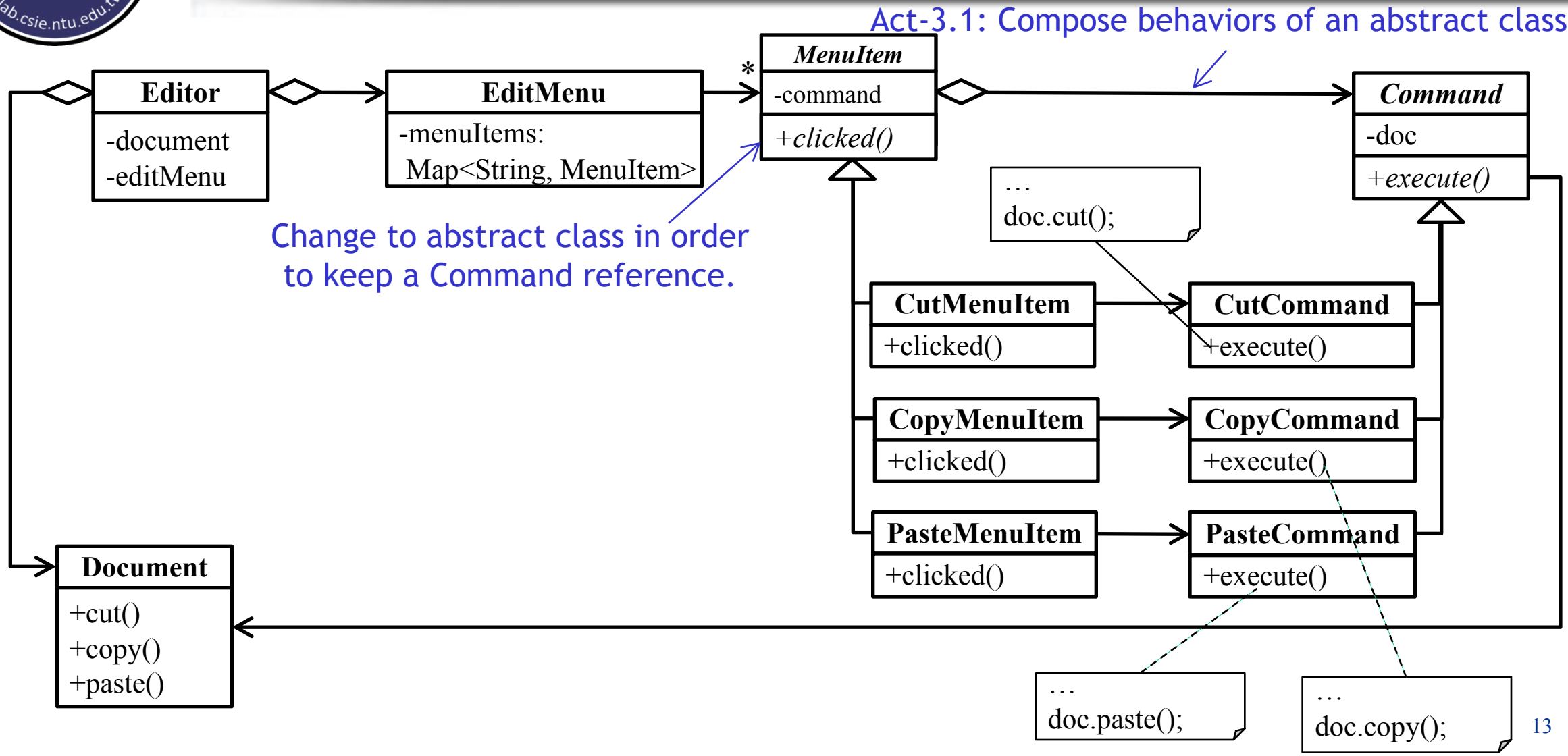
Act-1: Encapsulate What Varies



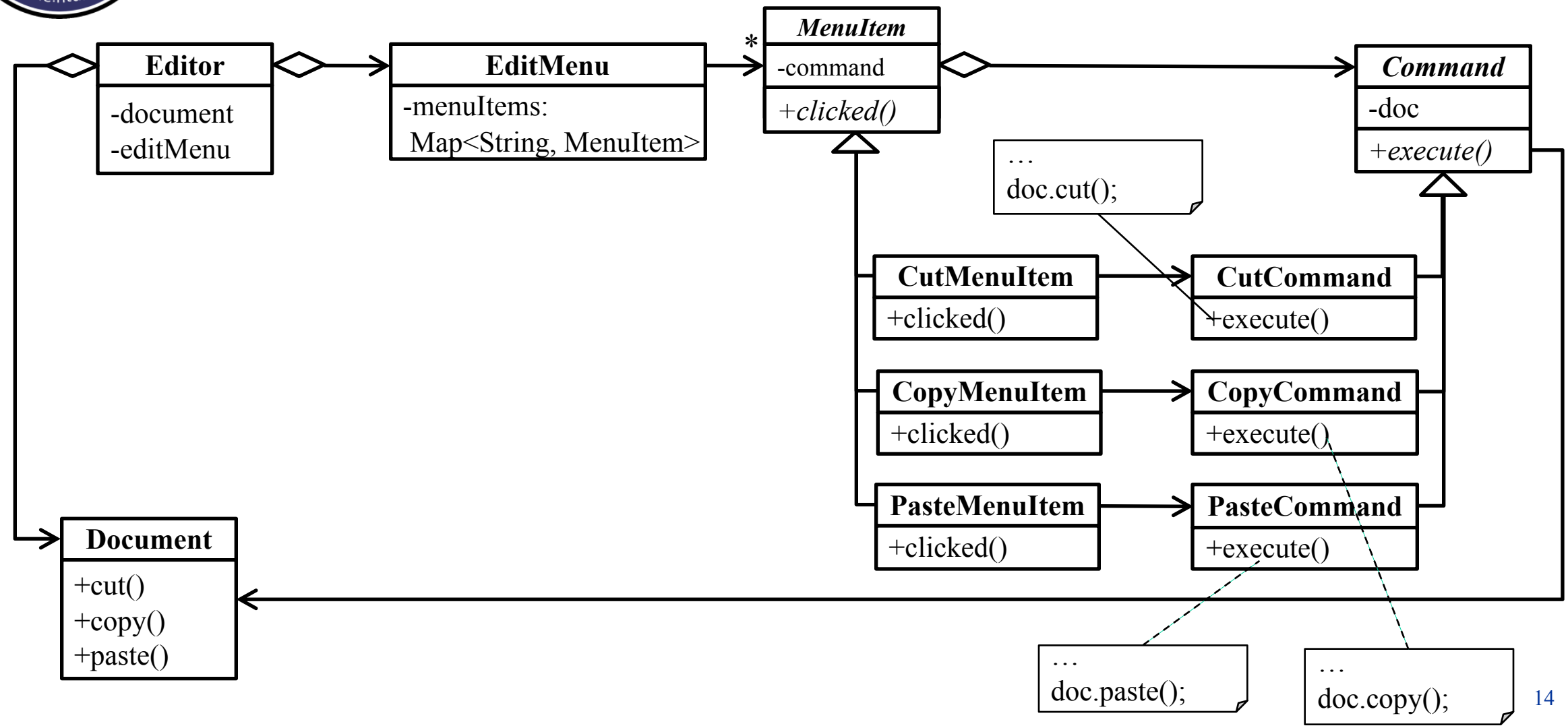
Act-2: Abstract Common Behaviors



Act-3: Compose Abstract Behaviors



Refactored Design after Design Process



Editor

```
public class Editor {
    private Document docuement;
    private EditMenu editMenu;

    public void setDocument(Document document) { editMenu = new EditMenu(document); }

    public EditMenu getEditMenu(){
        return editMenu;
    }
}
```

Document

```
public class Document {

    public void cut() { System.out.println("Cut~"); }

    public void copy() { System.out.println("Copying"); }

    public void paste() { System.out.println("Paste Done"); }
}
```

Editmenu

```
public class EditMenu {
    Map<String, MenuItem> menuItems = new HashMap<>();

    public EditMenu(Document document){
        menuItems.put("Cut", new CutMenuItem(document));
        menuItems.put("Copy", new CopyMenuItem(document));
        menuItems.put("Paste", new PasteMenuItem(document));
    }

    public MenuItem getMenuItem(String str) { return menuItems.get(str); }
}
```

MenuItem

```
public abstract class MenuItem {
    private Command command;

    public MenuItem(Command command){
        this.command = command;
    }

    public Command getCommand() { return command; }

    public abstract void click();
}
```

CopyMenuItem

```
public class CopyMenuItem extends MenuItem{

    public CopyMenuItem(Document document) { super(new CopyCommand(document)); }

    @Override
    public void click() { getCommand().execute(); }

}
```

CutMenuItem

```
public class CutMenuItem extends MenuItem{

    public CutMenuItem(Document document) { super(new CutCommand(document)); }

    @Override
    public void click() { getCommand().execute(); }

}
```

PasteMenuItem

```
public class PasteMenuItem extends MenuItem{

    public PasteMenuItem(Document document) { super(new PasteCommand(document)); }

    @Override
    public void click() { getCommand().execute(); }

}
```

Command

```
public abstract class Command {
    private Document docuement;

    public Document getDocument() { return docuement; }

    public Command(Document docuement) { this.docuement = docuement; }

    public abstract void execute();
}
```

CutCommand

```
public class CutCommand extends Command{

    public CutCommand(Document docuement) { super(docuement); }

    @Override
    public void execute() { getDocument().cut(); }

}
```

CopyCommand

```
public class CopyCommand extends Command{

    public CopyCommand(Document docuement) { super(docuement); }

    @Override
    public void execute() { getDocument().copy(); }

}
```

PasteCommand

```
public class PasteCommand extends Command{

    public PasteCommand(Document docuement) { super(docuement); }

    @Override
    public void execute() { getDocument().paste(); }

}
```

Input / Output

Input:

```
[operation]
...
```

Output:

```
// if [operation] is Cut
     Cut~


// if [operation] is Copy
     Copying


// if [operation] is Paste
     Paste Done
```

Test cases

- TestCase 1: Copy
- TestCase 2: Cuy
- TestCase 3: Paste
- TestCase 4: Complex

Test case1

Sample1.in

```
1 Copy
```

Sample1.out

```
1 Copying
2
```

Test case2

Sample2.in

```
1 Cut
```

Sample2.out

```
1 Cut~
2
```

Test case3

Sample3.in

```
1 Paste
```

Sample3.out

```
1 Paste Done
2
```

Test case4

Sample4.in

```
1  Cut
2  Copy
3  Paste
4  Copy
5  Paste
6  Paste
7  Copy
8  Cut
9  Copy
10 Paste
11 Copy
12 Paste
13 Paste
14 Copy
15 Copy
16 Paste
17 Paste
18 Copy
19 Cut
20 Copy
```

Sample4.out

```
1  Cut~
2  Copying
3  Paste Done
4  Copying
5  Paste Done
6  Paste Done
7  Copying
8  Cut~
9  Copying
10 Paste Done
11 Copying
12 Paste Done
13 Paste Done
14 Copying
15 Copying
16 Paste Done
17 Paste Done
18 Copying
19 Cut~
20 Copying
```

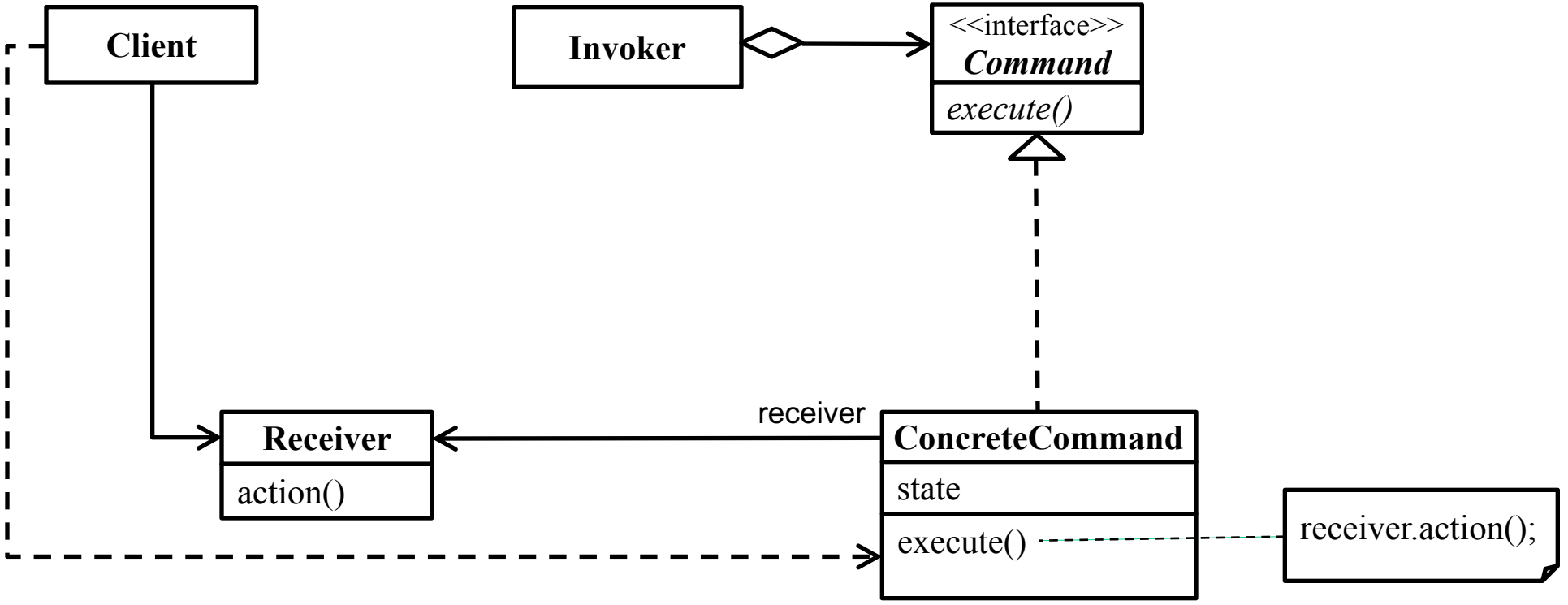
Recurrent Problems

- The invoker object is subject to be modified once the set of the actions on a receiver is changed
  - Sometimes it's necessary to issue requests to objects without knowing anything about the operation being requested or the receiver of the request.

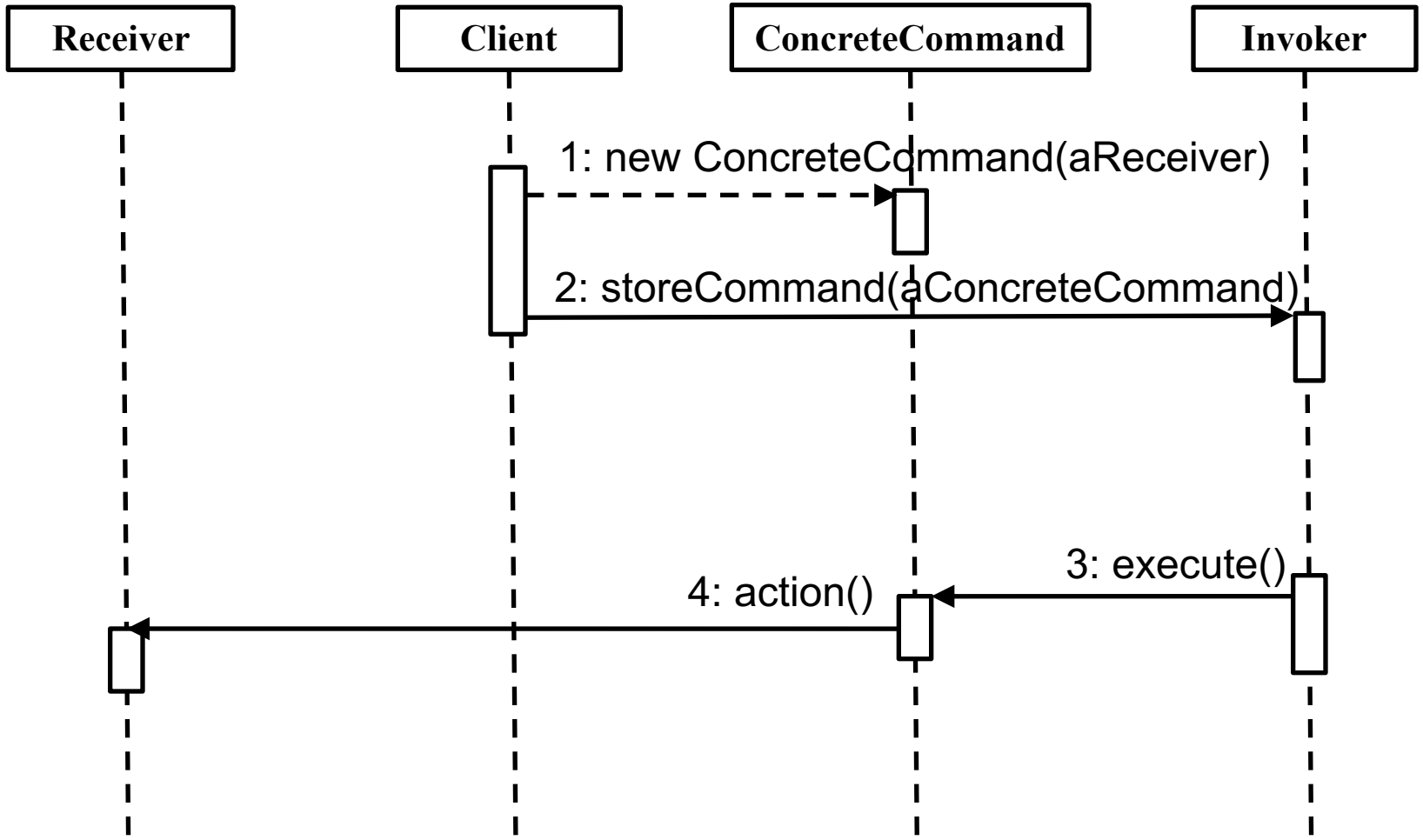
Intent

- Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

Command Pattern Structure$_1$

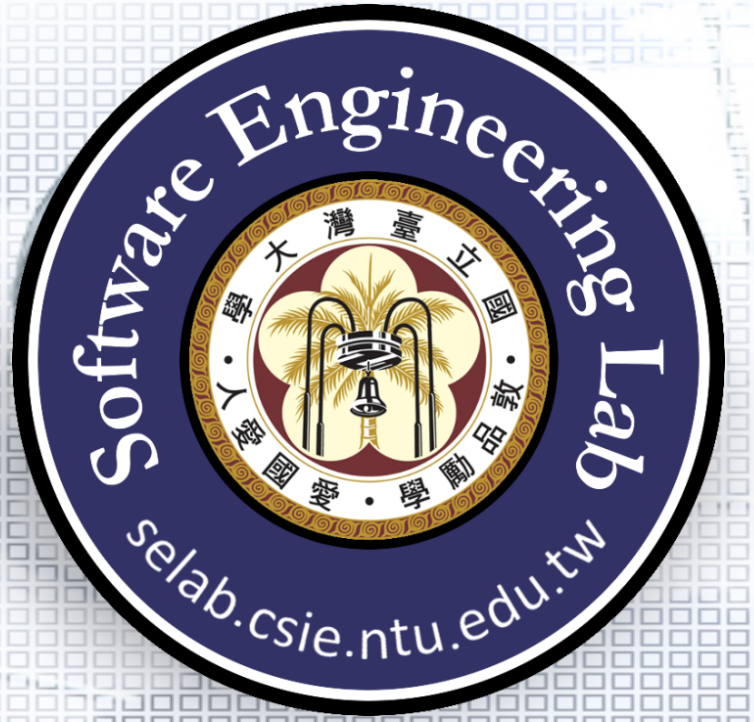


Command Pattern Structure$_2$



Command Pattern Structure$_3$

	Instantiation	Use	Termination
Command	X	Invoker keeps a Command reference, and then delegate to it while Invoker is requested.	X
Concrete Command	Client	Client creates a ConcreteCommand with a Receiver as a parameter, and store it to an Invoker. The Invoker delegates the request to the ConcreteCommand, and it uses the Receiver to complete the request.	ConcreteCommand will be terminated while the Invoker doesn't need it anymore.
Invoker	Don't Care	Invoker holds a Command reference from Client. While Invoker is invoked, it delegates to Command.	Don't Care
Receiver	Don't Care	Receiver is sent to ConcreteCommand as a reference by Client. ConcreteCommand completes its request by using Receiver.	Don't Care



Remote Control

Prof. Jonathan Lee (李允中)

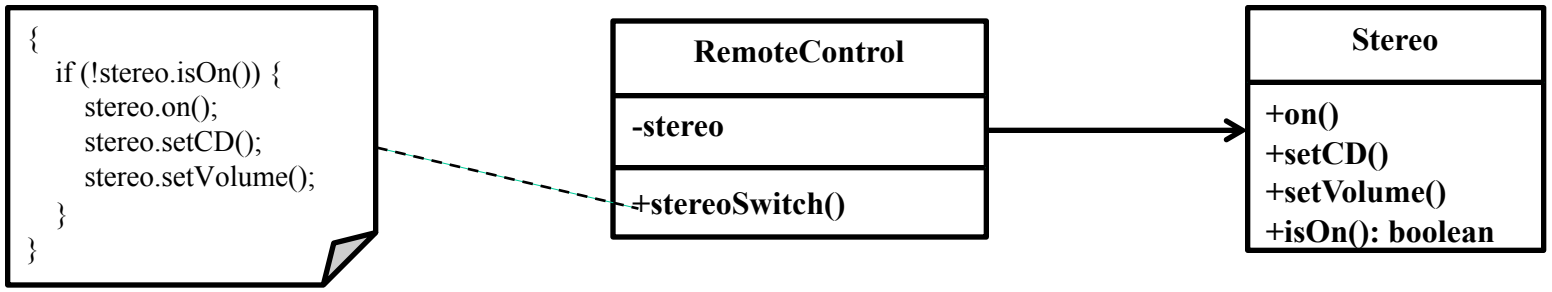
Department of Computer Science and Information Engineering
National Taiwan University

Requirements Statements

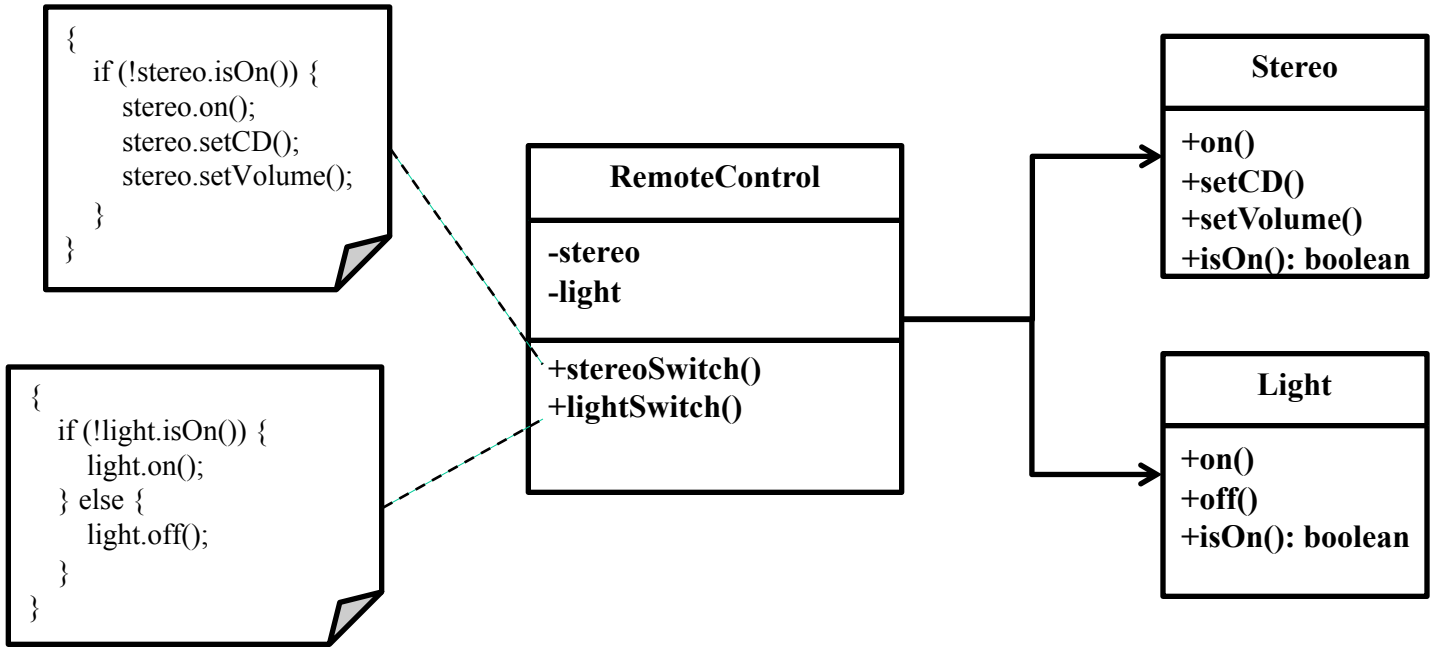
- The remote control can control a Week16InClassHW1 Week16InClassHW1 remotely.
- While a stereo is switched on by the remote control, the CD and volume will be set at the same time.
- Furthermore, the remote control also controls light that can be switched on and off.

Requirements Statements$_1$

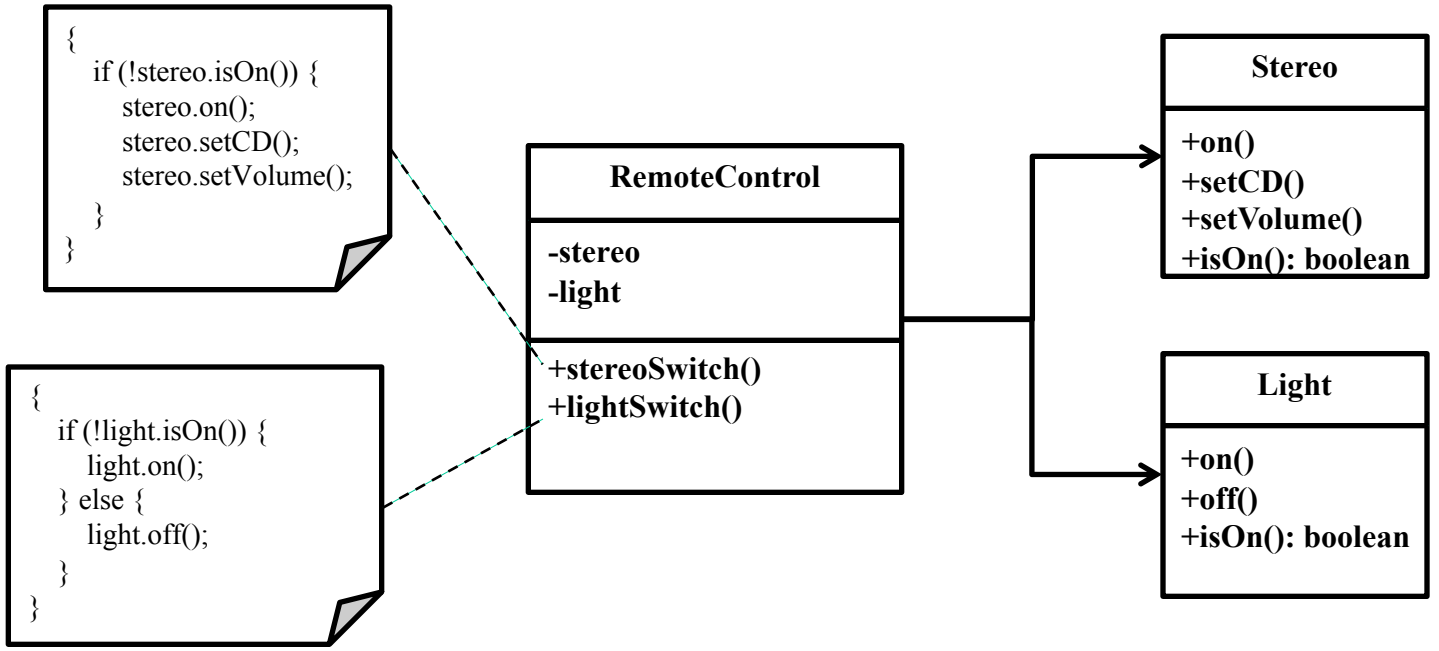
- The remote control can control a stereo remotely.
- While a stereo is switched on by the remote control, the CD and volume will be set at the same time.



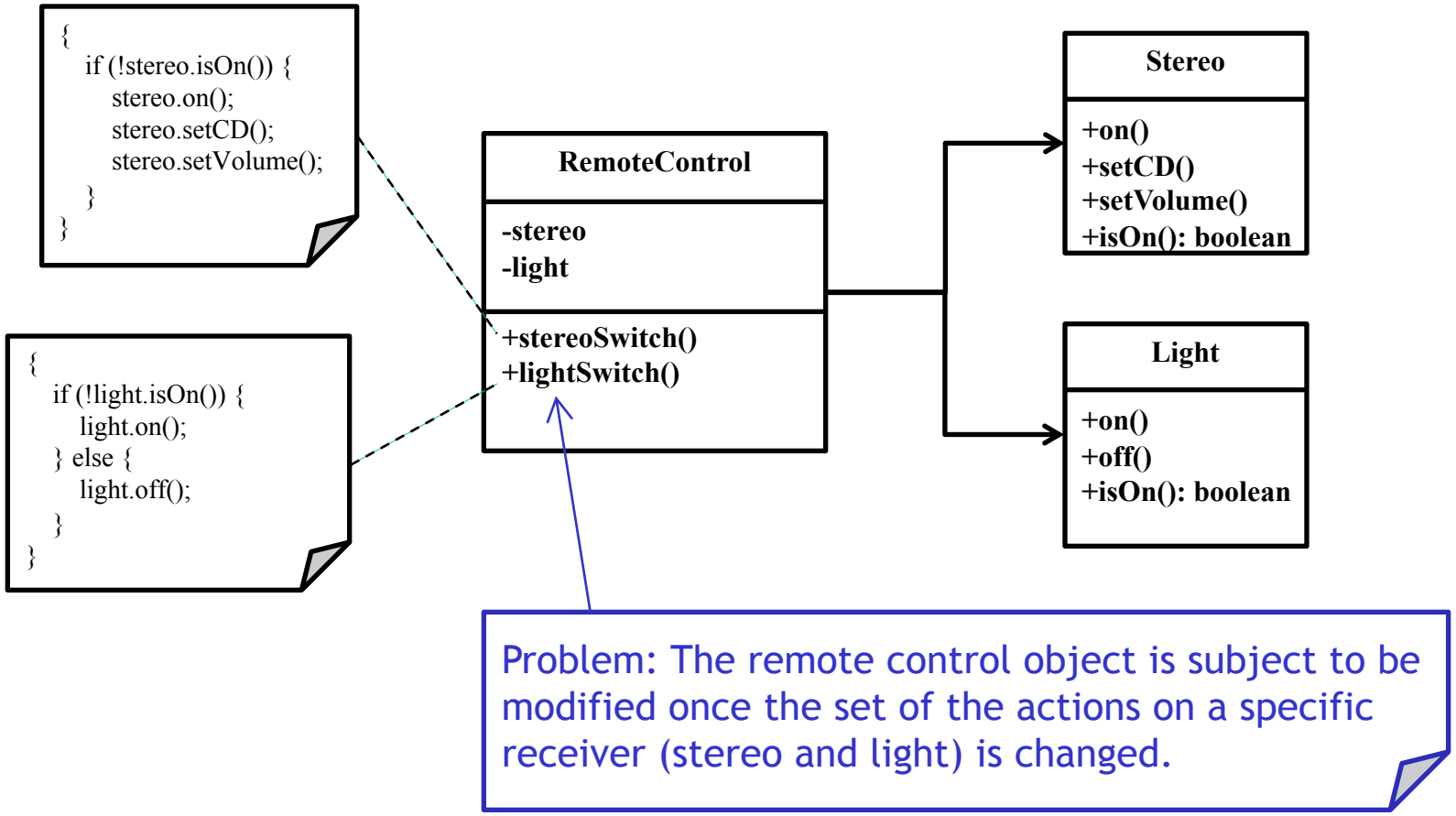
Requirements Statements$_2$



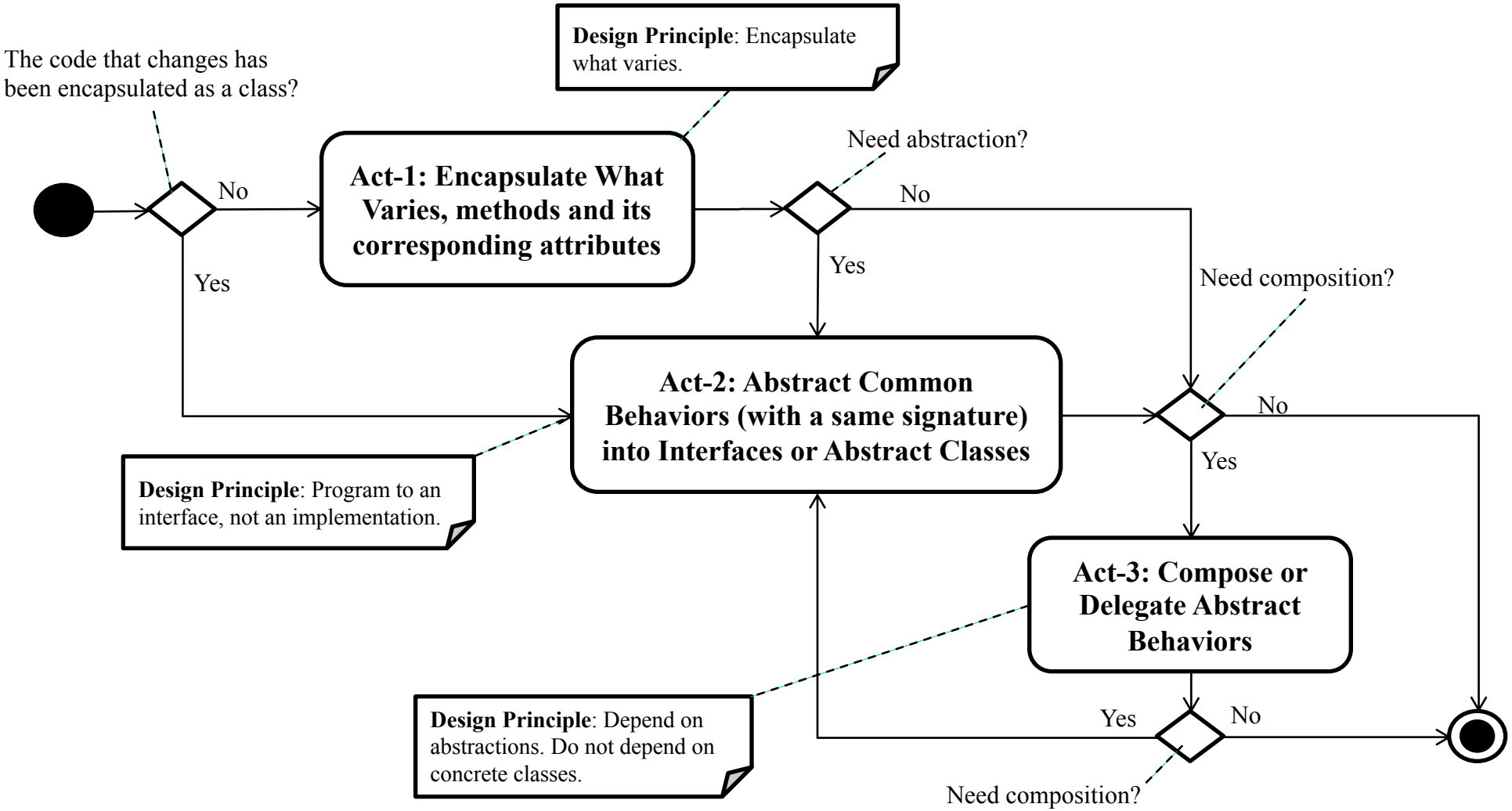
Initial Design - Class Diagram



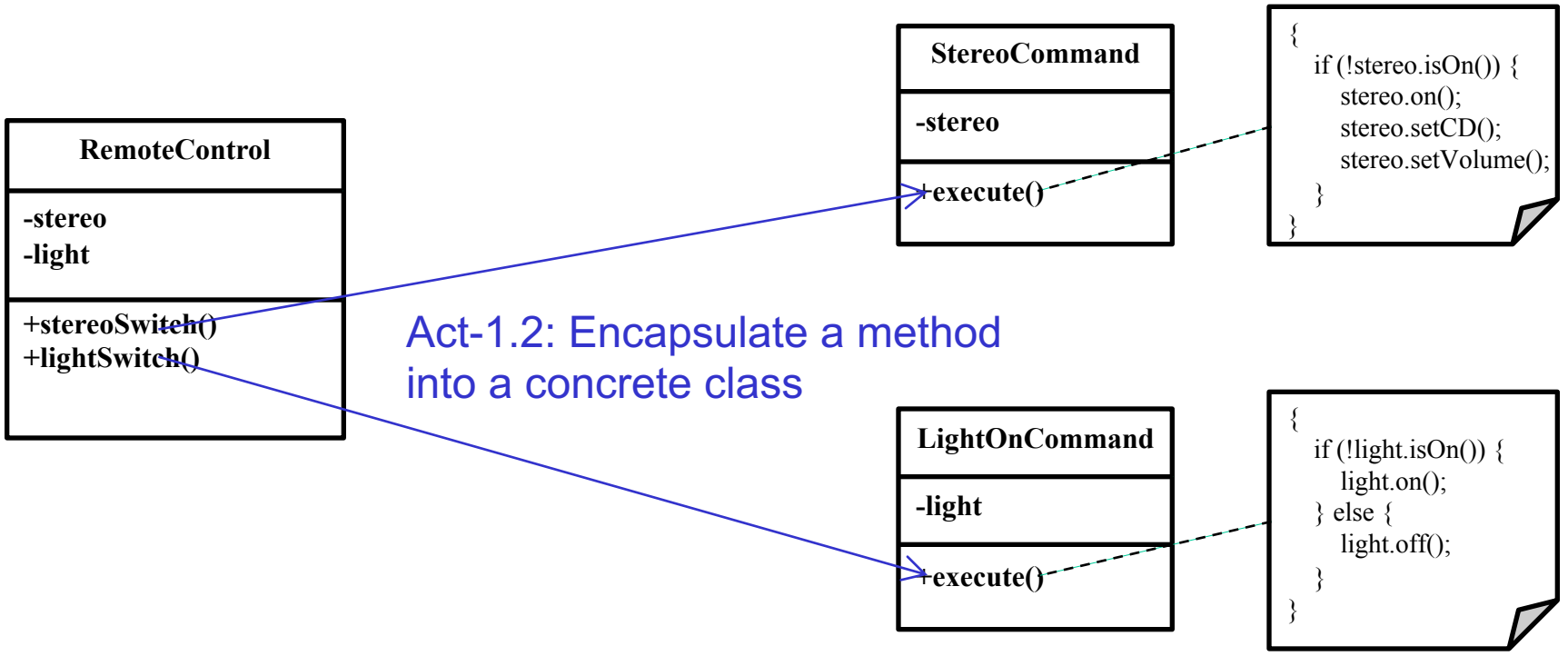
Problems with Initial Design



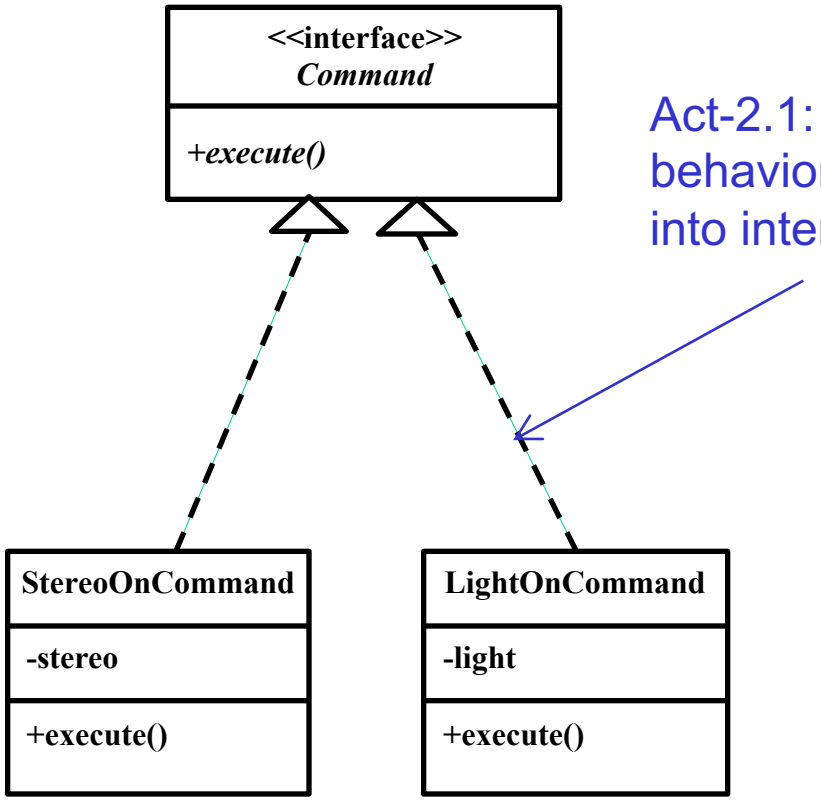
Design Process for Change



Act-1: Encapsulate What Varies



Act-2: Abstract Common Behaviors

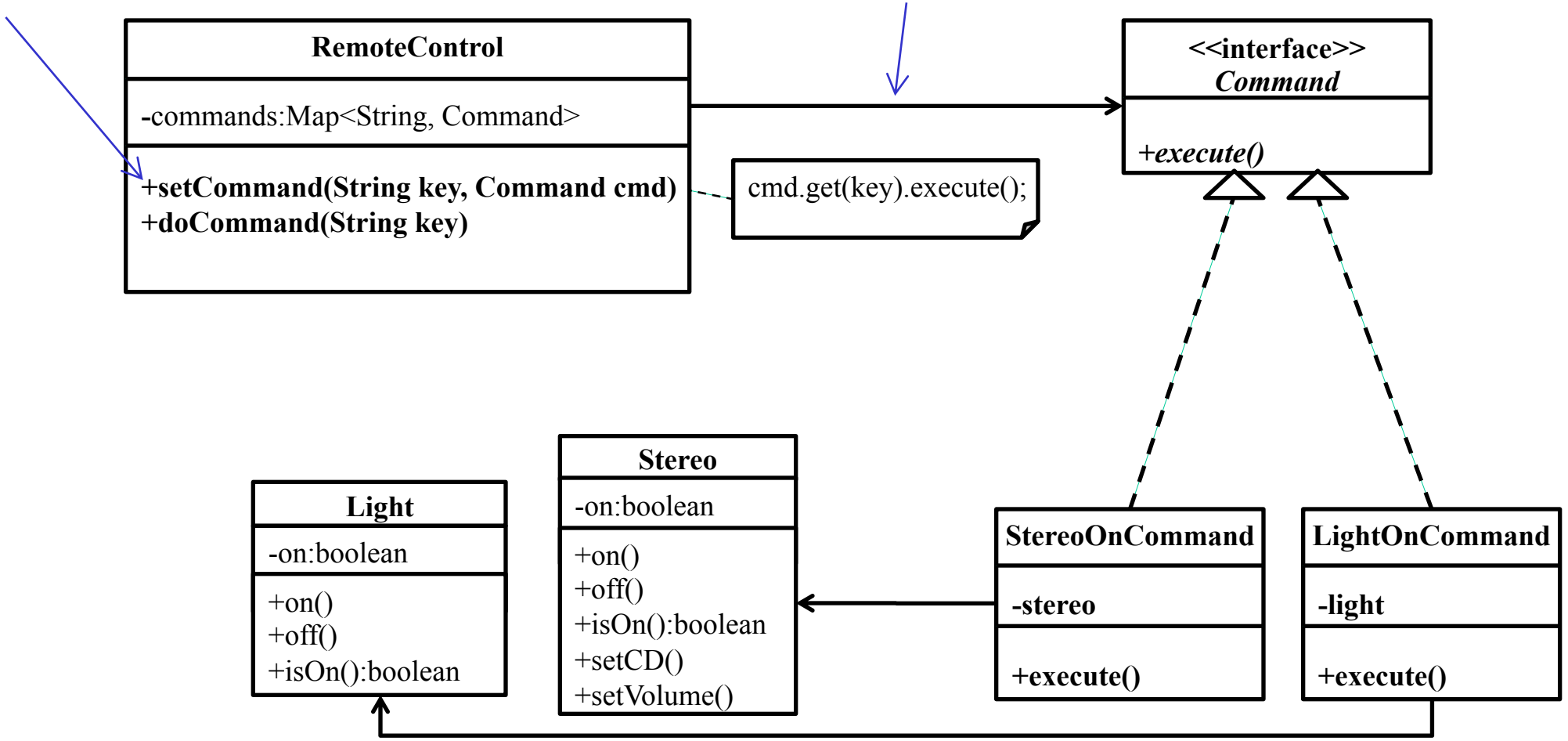


Act-2.1: Abstract common behaviors with a same signature into interface through inheritance

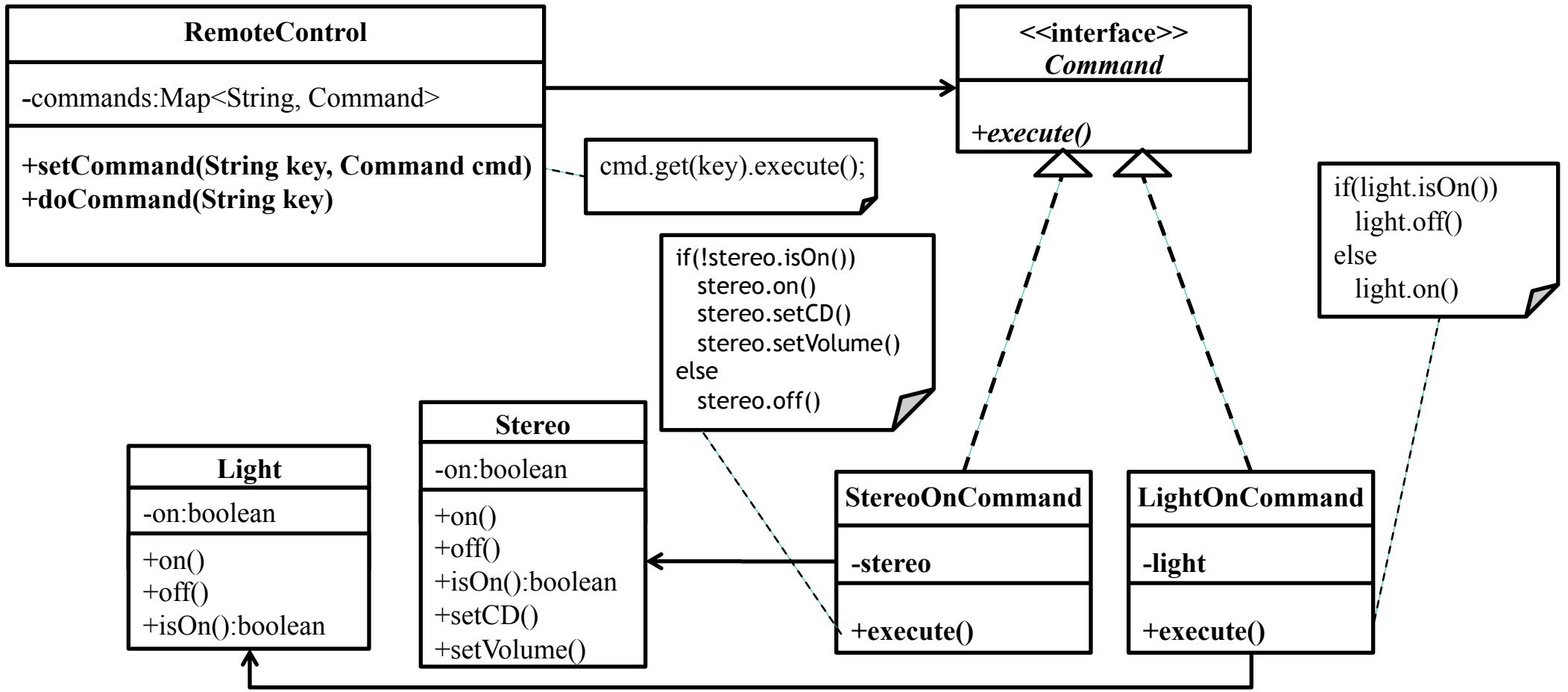
Act-3: Compose Abstract Behaviors

Provide an interface to add commands.

Act-3.1: Compose behaviors of an interface



Refactored Design after Design Process



RemoteControl

```
public class RemoteControl {
    private Map<String, Command> commands = new HashMap<>();

    public void setCommand(String key, Command command){
        commands.put(key, command);
    }

    public void doCommand(String key) { commands.get(key).execute(); }
}
```

Command

```
public interface Command {
    public void execute();
}
```

LightCommand

```
public class LightCommand implements Command{
    private Light light;

    public LightCommand(Light light) { this.light = light; }

    @Override
    public void execute() {
        if(light.isOn()){
            light.off();
        }
        else {
            light.on();
        }
    }

}
```

StereoCommand

```
public class StereCommand implements Command{
    private Stereo stereo;

    public StereCommand(Stereo stereo) { this.stereo = stereo; }

    @Override
    public void execute() {
        if(!stereo.isOn()){
            stereo.on();
            stereo.setCD();
            stereo.setVolume();
        }
        else {
            stereo.off();
        }

    }
}
```

Light

```java
public class Light {
    private boolean on = false;

    public void on(){
        System.out.println("Light turns on~");
        on = true;
    }

    public void off(){
        System.out.println("Light turns off!");
        on = false;
    }

    public boolean isOn() { return on; }
}
```

Stereo

```java
public class Stereo {
    private boolean on = false;

    public void on(){
        System.out.println("Stereo turns on!");
        on = true;
    }

    public void setCD() { System.out.println("Stereo set CD #1"); }

    public void setVolume() { System.out.println("Stereo set Volume 13"); }

    public void off(){
        System.out.println("Stereo turns off~");
        on = false;
    }

    public boolean isOn() { return on; }
}
```

Input / Output format

Input:

```
[device]
...
```

Output:

```
//If state of Light is on
Light turns on~

//If state of Light is off
Light turns off!

//If state of Stereo is on
Stereo turns on!
Stereo set CD #1
Stereo set Volume 13

//If state of Stereo is off
Stereo turns off~
```

Test cases

- TestCase 1: Light
- TestCase 2: Stereo
- TestCase 3: Both

Test case1

Sample1.in

```
1 Light
2 Light
3 Light
```

Sample1.out

```
1 Light turns on~
2 Light turns off!
3 Light turns on~
```

Test case2

Sample2.in

```
1  Stereo
2  Stereo
3  Stereo
4
```

Sample2.out

```
1  Stereo turns on!
2  Stereo set CD #1
3  Stereo set Volume 13
4  Stereo turns off~
5  Stereo turns on!
6  Stereo set CD #1
7  Stereo set Volume 13
```

Test case3

Sample3.in

```
1  Light
2  Stereo
3  Light
4  Stereo
5  Light
6  Stereo
7  Light
8  Stereo
9  Light
10 Stereo
11 Light
12 Stereo
13 Light
14 Stereo
15 Light
16 Stereo
17 Light
18 Stereo
```

Sample3.out

```
1  Light turns on~
2  Stereo turns on!
3  Stereo set CD #1
4  Stereo set Volume 13
5  Light turns off!
6  Stereo turns off~
7  Light turns on~
8  Stereo turns on!
9  Stereo set CD #1
10 Stereo set Volume 13
11 Light turns off!
12 Stereo turns off~
13 Light turns on~
14 Stereo turns on!
15 Stereo set CD #1
16 Stereo set Volume 13
17 Light turns off!
18 Stereo turns off~
19 Light turns on~
20 Stereo turns on!
21 Stereo set CD #1
22 Stereo set Volume 13
23 Light turns off!
24 Stereo turns off~
25 Light turns on~
26 Stereo turns on!
27 Stereo set CD #1
28 Stereo set Volume 13
```