# CptS 487
# Software Design and Architecture

Lesson 14

Design Patterns 5:

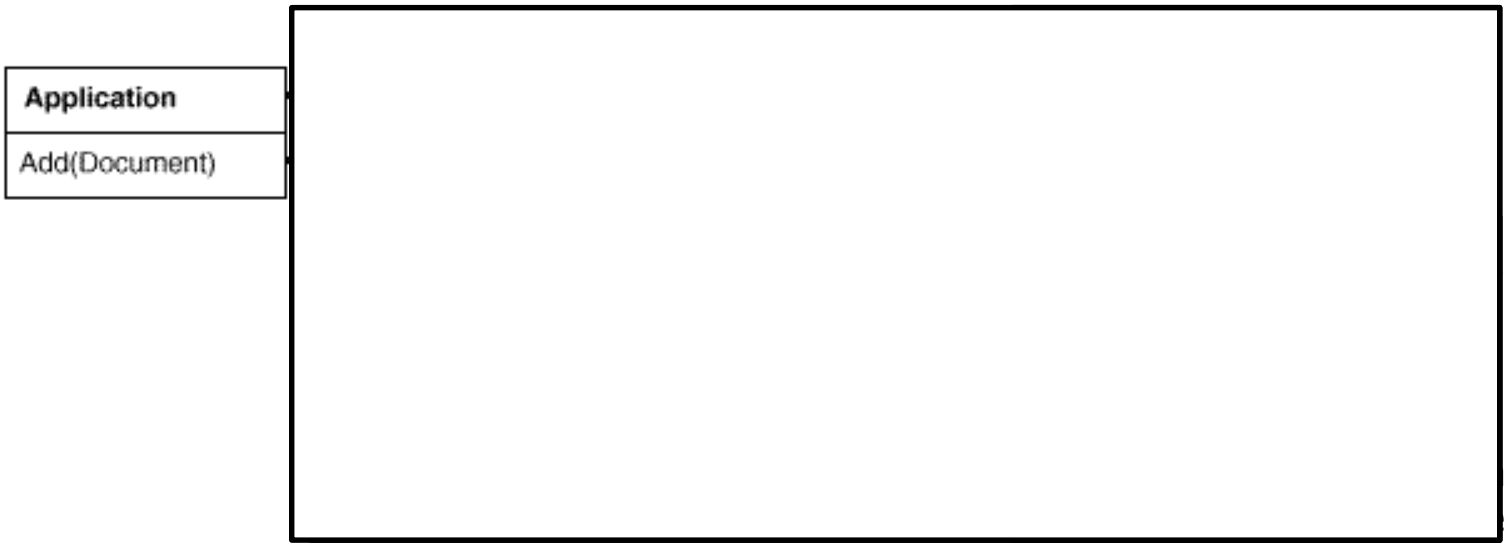
Command and Memento

WASHINGTON STATE UNIVERSITY
World Class. Face to Face.

**Instructor: Bolong Zeng**

# 4. Command (Object behavioral pattern)

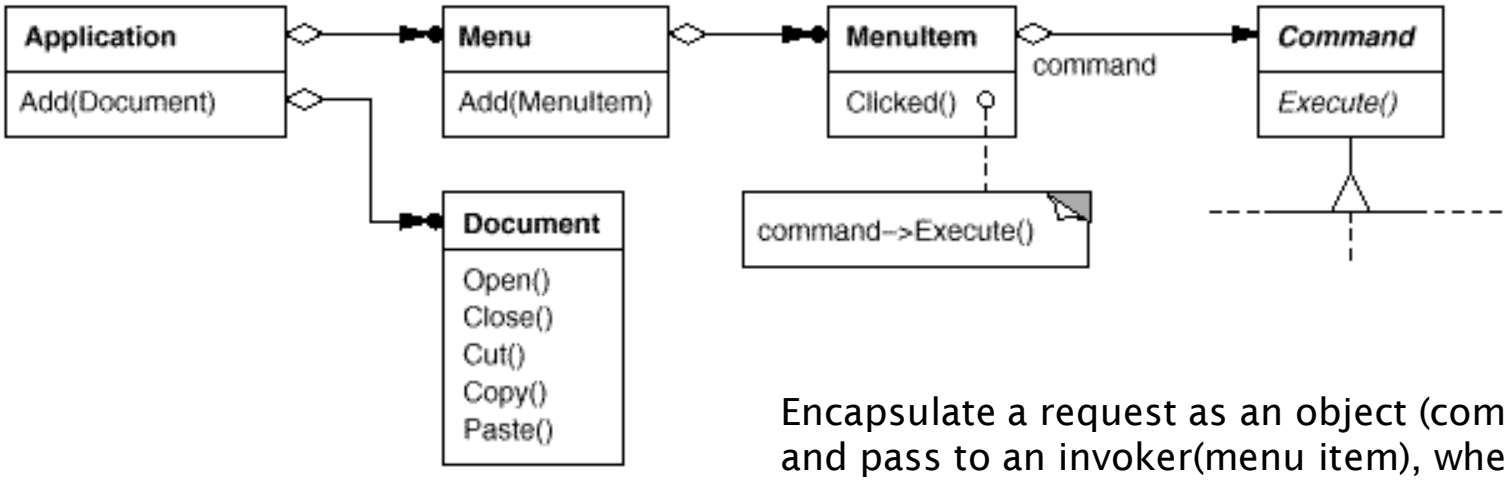
- Intent
  - Encapsulate a request as an object, therefore
    - allow to parameterize clients with different requests, queue or log requests
    - support undoable operations.
- Motivation




mand)
re the
invoker doesn't know how to service the request but uses the encapsulated command to perform an action.

# 4. Command (Object behavioral pattern)

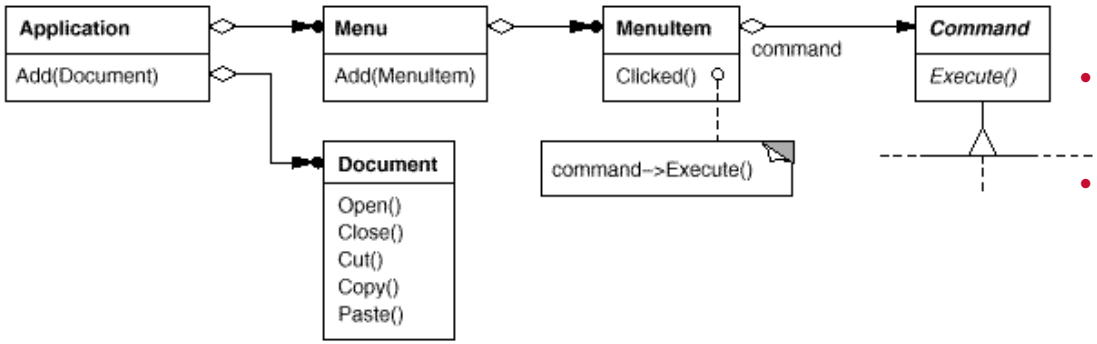
- Intent
  - Encapsulate a request as an object, therefore
    - allow to parameterize clients with different requests, queue or log requests
    - support undoable operations.
- Motivation

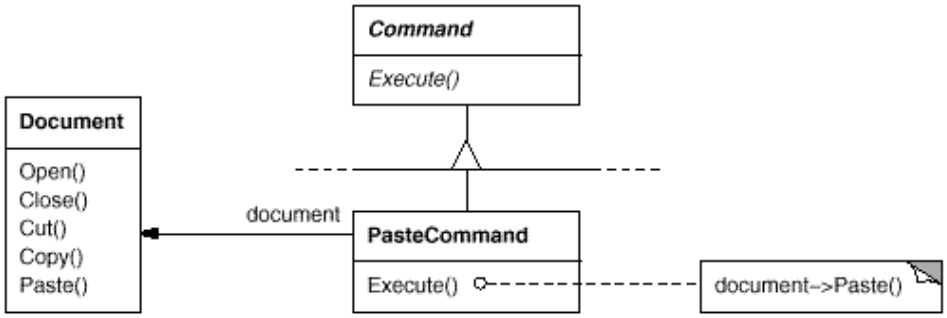



Encapsulate a request as an object (command) and pass to an invoker(menu item), where the invoker doesn't know how to service the request but uses the encapsulated command to perform an action.

# 4. Command

- Motivation



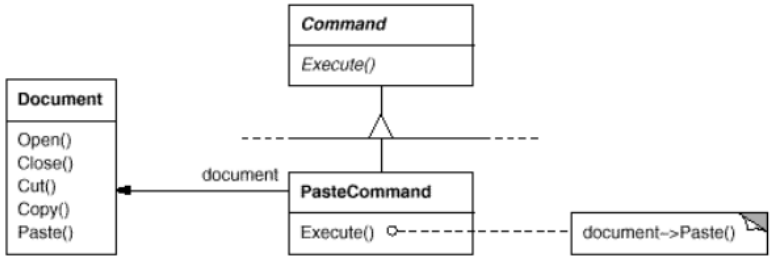

- The application configures each MenuItem with an instance of a concrete Command subclass.
- When the user selects a MenuItem, the MenuItem calls Execute on its command, and Execute carries out the operation.
- MenuItems don't know which subclass of Command they use.
- Command subclasses store the receiver of the request and invoke one or more operations on the receiver.

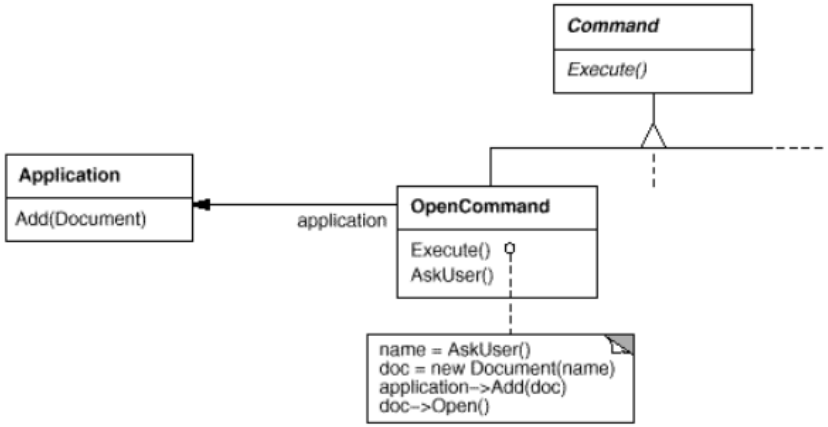
For example:

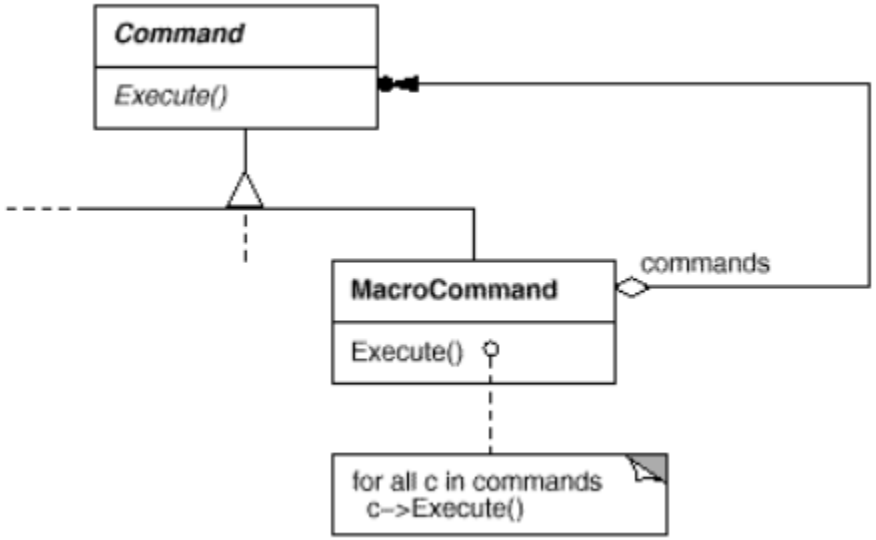
- PasteCommand supports pasting text from the clipboard into a Document.
- PasteCommand's receiver is the Document object it is supplied upon instantiation.
- The Execute operation invokes Paste on the receiving Document.

# 4. Command

- Motivation



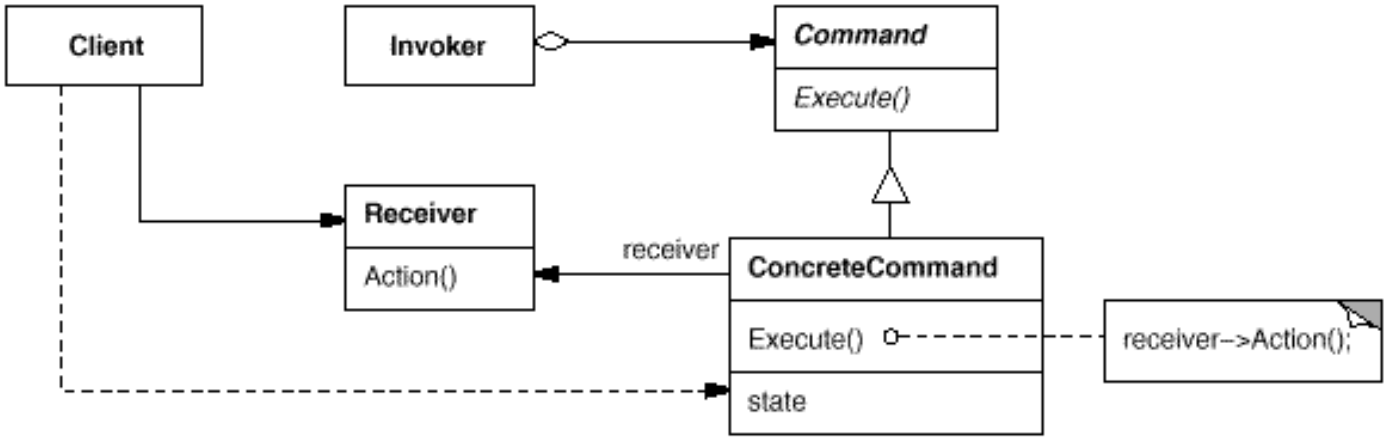



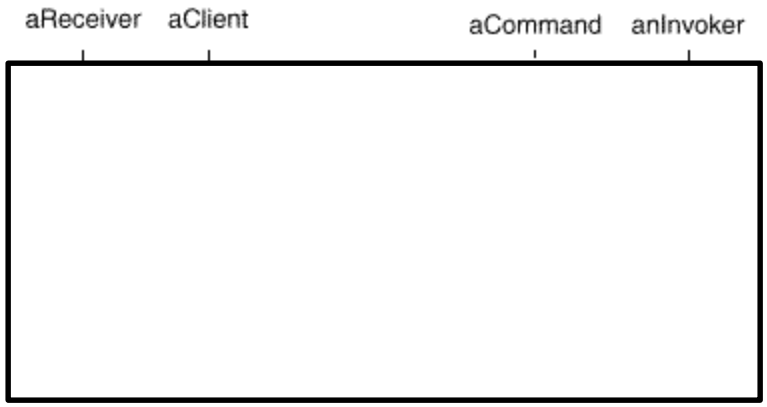

- More commands can be defined, which behaved rather differently.

# 4. Command

- Structure



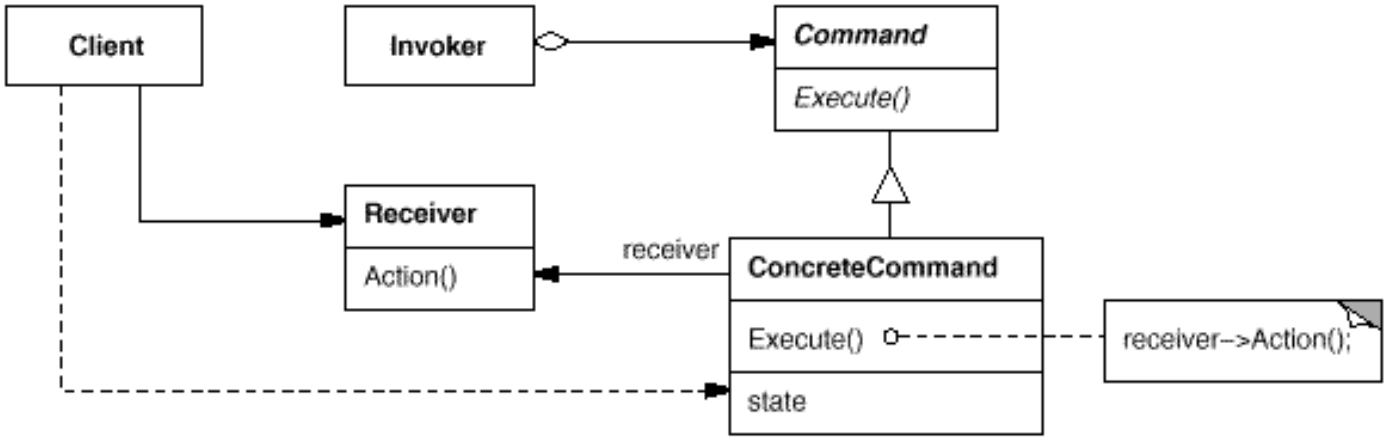

- Collaborations



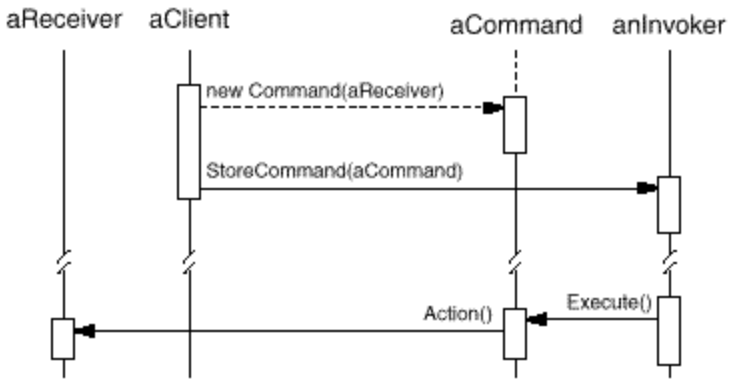

- Command is an interface with execute method. It is the core of contract.
- The client creates an instance of a ConcreteCommand implementation and associates it with a receiver.
- An Invoker object stores the ConcreteCommand object
- The Invoker issues a request by calling Execute on the command.
- The ConcreteCommand object invokes operations on its receiver to carry out the request.

# 4. Command

- Structure




- Collaborations




- Command is an interface with execute method. It is the core of contract.
- The client creates an instance of a ConcreteCommand implementation and associates it with a receiver.
- An Invoker object stores the ConcreteCommand object
- The Invoker issues a request by calling Execute on the command.
- The ConcreteCommand object invokes operations on its receiver to carry out the request.

# 4. Command

- Applicability

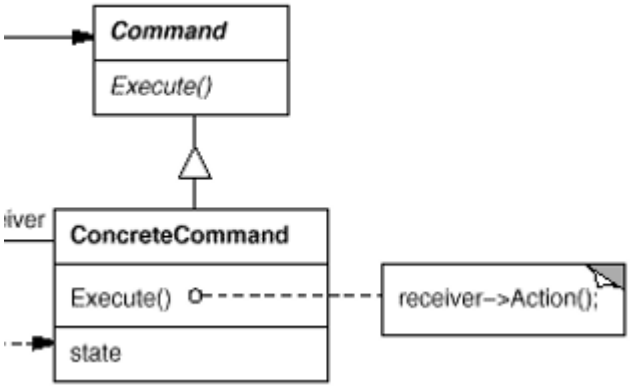
  Use the Command pattern when
  - You want to implement a callback function capability
    - Commands are an object-oriented replacement for callbacks
  - You want to specify, queue, and execute requests at different times
    - A Command object can have a lifetime independent of the original request.
    - You can transfer a command object for the request to a different process and fulfill the request there
  - You need to support undo and change log operations
    - The Command's Execute operation can store state for reversing its effects in the command itself.
    - The Command interface must have an added Unexecute operation that reverses the effects of a previous call to Execute.
    - Executed commands are stored in a history list.

# 4. Command

- Consequences
  - Benefits:
    - Decouples the object that invokes the operation from the one that knows how to perform it.
    - Commands can be manipulated and extended as any other object
    - You can assemble commands into a composite command, using Composite pattern.
    - Easy to add new Commands, since you don’t need to change existing classes.

# Command Example

```
1.  //Command
2.  public interface Command
3.  {
4.     public void execute();
5.  }
```



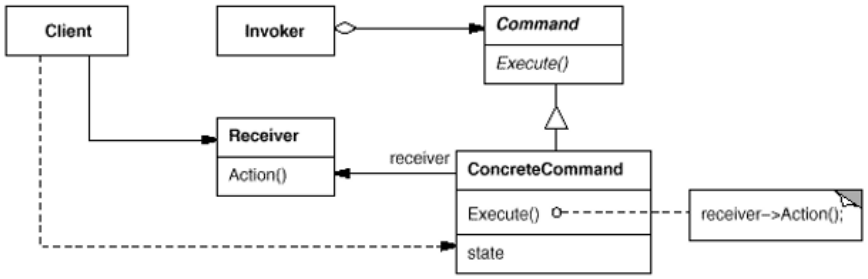

```
01.  //Concrete Command
02.  public class LightOnCommand implementsCommand
03.  {
04.     //reference to the light
05.     Light light;
06.
07.     public LightOnCommand(Light light)
08.     {
09.        this.light = light;
10.     }
11.
12.     public void execute()
13.     {
14.        light.switchOn();
15.     }
16.
17.  }
```

```
01.  //Concrete Command
02.  public class LightOffCommand implementsCommand
03.  {
04.     //reference to the light
05.     Light light;
06.
07.     public LightOffCommand(Light light)
08.     {
09.        this.light = light;
10.     }
11.
12.     public void execute()
13.     {
14.        light.switchOff();
15.     }
16.
17.  }
```

# Command Example

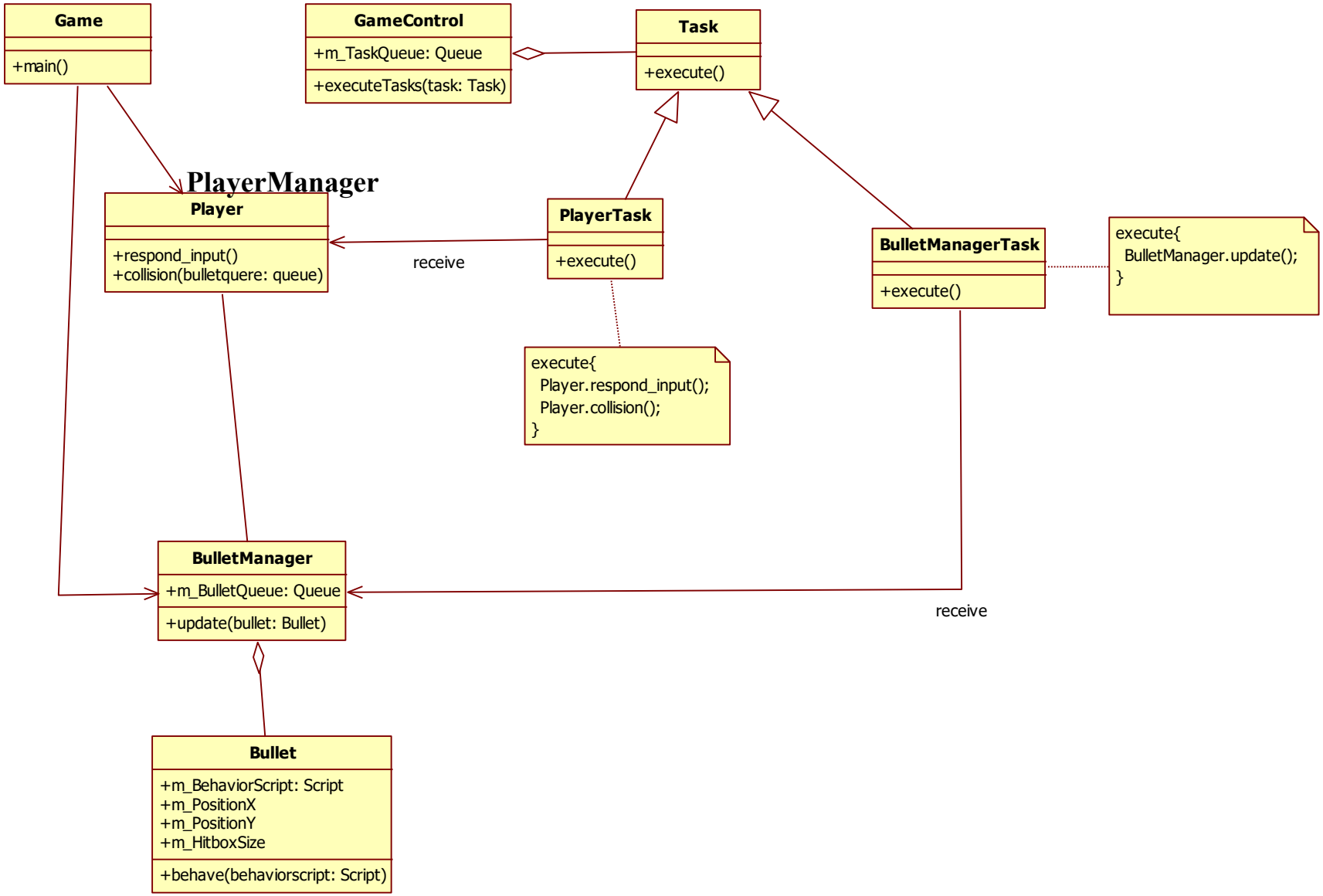
```
01. //Receiver
02. public class Light
03. {
04.    private boolean on;
05.
06.    public void switchOn()
07.    {
08.       on = true;
09.    }
10.
11.    public void switchOff()
12.    {
13.       on = false;
14.    }
15.
16. }
```

```
01. //Invoker
02. public class RemoteControl
03. {
04.     private Command command;
05.
06.     public void setCommand(Command command)
07.     {
08.         this.command = command;
09.     }
10.
11.
12.     public void pressButton()
13.     {
14.         command.execute();
15.     }
16.
17. }
```



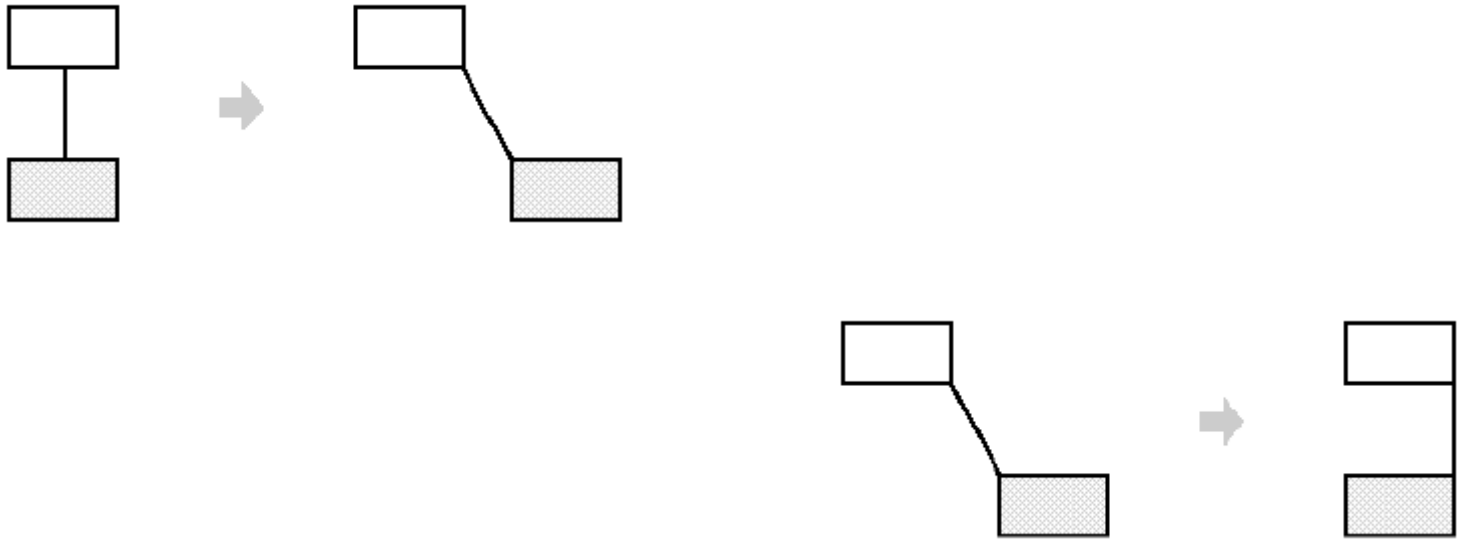

```
01. //Client
02. public class Client
03. {
04.     public static void main(String[] args)
05.     {
06.         RemoteControl control = new RemoteControl();
07.
08.         Light light = new Light();
09.
10.         Command lightsOn = new LightsOnCommand(light);
11.         Command lightsOff = new LightsOffCommand(light);
12.
13.         //switch on
14.         control.setCommand(lightsOn);
15.         control.pressButton();
16.
17.         //switch off
18.         control.setCommand(lightsOff);
19.         control.pressButton();
20.
21.     }
22.
23. }
```

# Command Example: Touhou



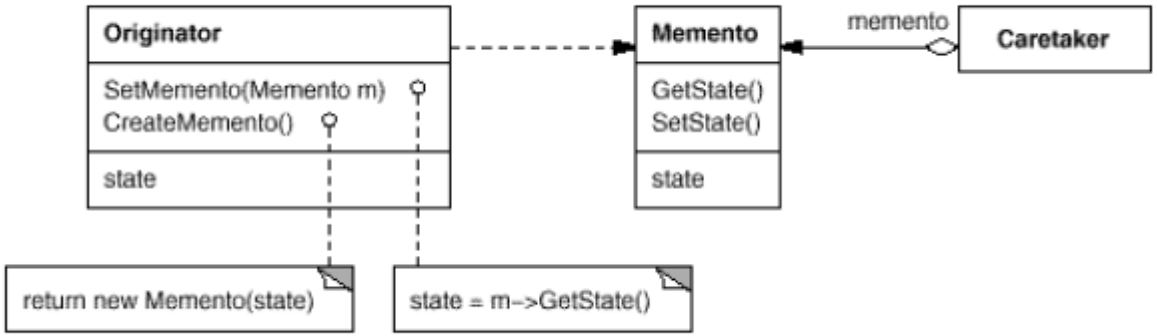
# 9. Memento (Object behavioral pattern)

- Intent
  - Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.
- Motivation



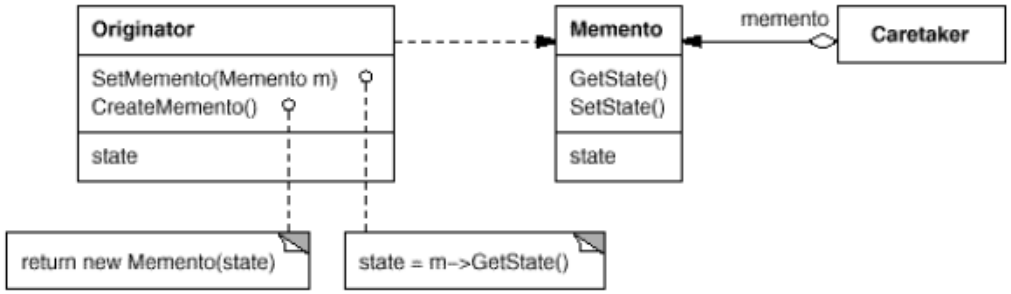
# 9. Memento (Object behavioral pattern)

- Applicability
  - A snapshot of (some portion of) an object's state must be saved so that it can be restored to that state later, *and*
  - A direct interface to obtaining the state would expose implementation details and break the object's encapsulation.
- Structure



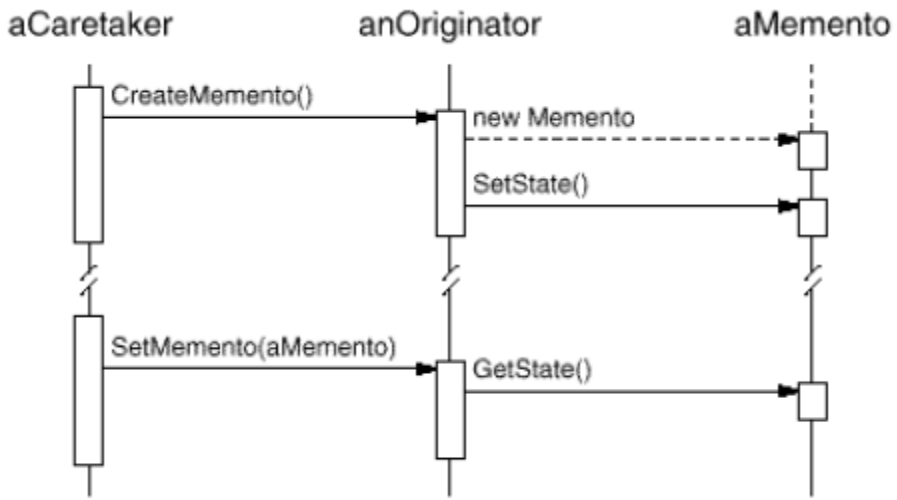
# 9. Memento (Object behavioral pattern)

- Participants
  - Memento
    - Stores internal state of the Originator object. The memento may store as much or as little of the originator's internal state as necessary at its originator's discretion.
    - Protects against access by objects other than the originator.
  - Originator
    - Creates a memento containing a snapshot of its current internal state.
    - Uses the memento to restore its internal state.
  - Caretaker
    - Is responsible for the memento's safekeeping.
    - Never operates on or examines the contents of a memento.



# 9. Memento (Object behavioral pattern)

- Collaborations
  - A caretaker requests a memento from an originator, holds it for a time, and passes it back to the originator, as the following diagram:



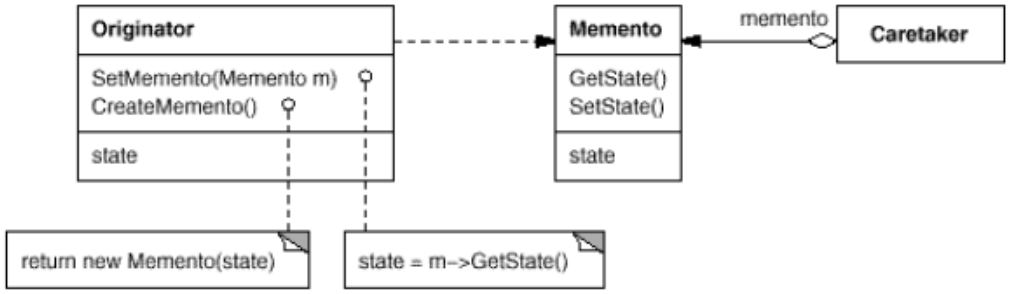
# 9. Memento (Object behavioral pattern)

- Consequences
  - Benefits
    - Preserving encapsulation boundaries
    - Simplifies Originator by stripping the “state” from the Originator.
  - Liabilities
    - Using mementos might be expensive.
    - Defining narrow and wide interfaces.
    - Hidden costs in caring for mementos.

- Implementation Issues
  - Language support
  - Storing incremental changes.

# Memento Example

```
01. //Memento
02. public class EditorMemento
03. {
04.
05.   private final String editorState;
06.
07.   public EditorMemento(String state)
08.   {
09.      editorState = state;
10.   }
11.
12.   public String getSavedState()
13.   {
14.    return editorState;
15.   }
16.
17. }
```




```
01. //Originator
02.
03. public class Editor
04. {
05.
06.   //state
07.   public String editorContents;
08.
09.   public void setState(String contents)
10.   {
11.      this.editorContents = contents;
12.   }
13.
14.   public EditorMemento save()
15.   {
16.     return new EditorMemento(editorContents);
17.   }
18.
19.   public void restoreToState(EditorMemento memento)
20.   {
21.    editorContents = memento.getSavedState();
22.   }
23.
24. }
```