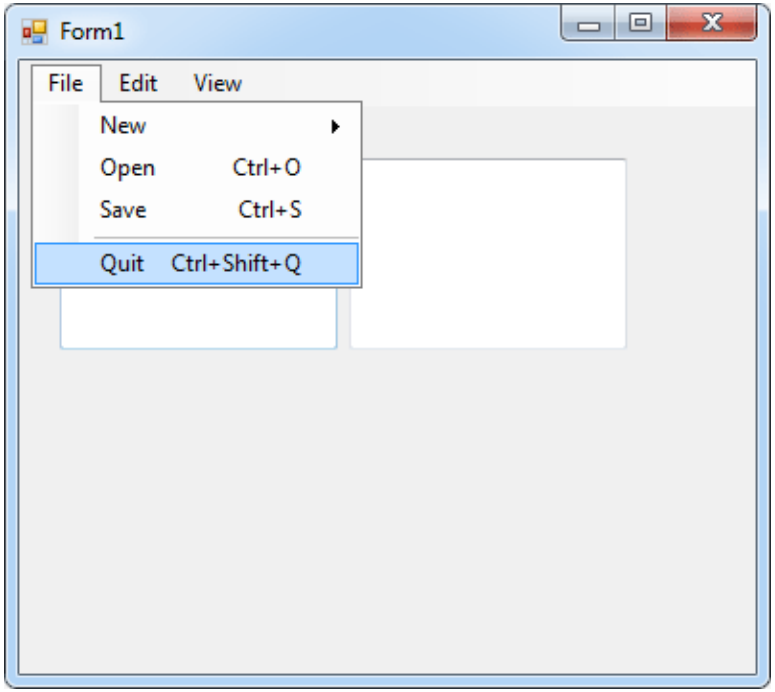
Command

Briefly

Commands are object oriented replacements for callbacks

-GoF

Basic example




```
public void HandleAction(string action) {
      if (action == "New")    {
            // handle action for new command
      }
      else if (action == "Open")    {
            // handle action for open command
      }
      else if (action == "Save")    {
            // handle action for save command
      }
      else (action == "Quit")    {
            // handle action for quit command
      }
}
```

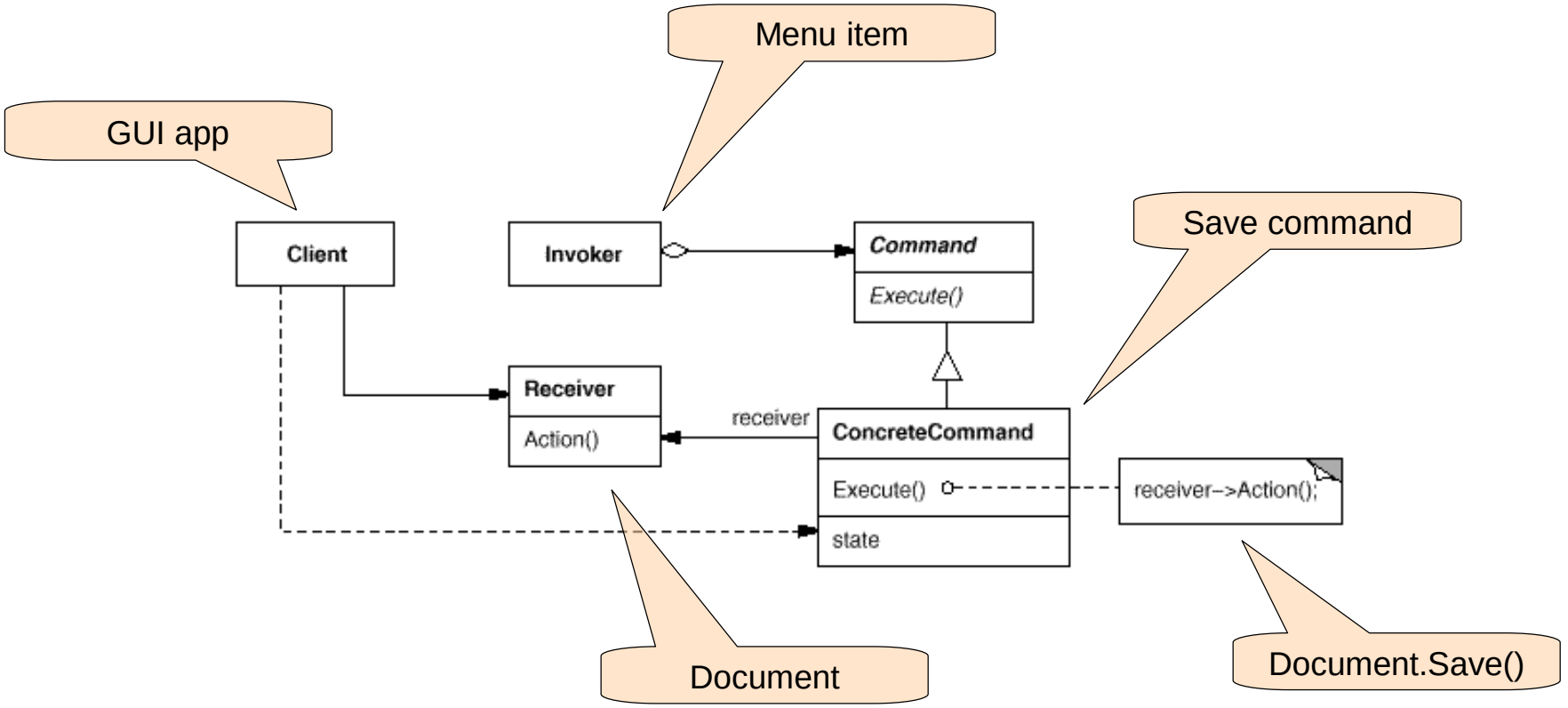
Hard to maintain

Basic example

```
interface ICommand {
     void Execute();
}
```

```
class Menu {
     private ICommand newCmd;
     private ICommand openCmd;

     public Menu(ICommand newCmd,
                 ICommand openCmd
                 /* ... */
     ) {
          this.newCmd = newCmd;
          this.openCmd = openCmd;
          // ...
     }

     public void HandleAction(string action) {
          if (action == "New")    {
               newCmd.Execute();
          }
          else if (action == "Open")    {
               OpenCmd.Execute();
          }
          // etc
     }
}
```

Formal structure



Receiver

- Specified during creation × during invocation

```
class OpenDialog : ICommand
{
    private readonly DialogService service;
    public OpenDialog(DialogService service) {
        this.service = service;
    }

    public void Execute()
    {
        service.Open("Welcome back!");
    }
}
```

Receiver

- Specified during creation × during invocation

```
interface ICharacterCommand
{
    void Execute(Character character);
}

class Jump : ICharacterCommand
{

    public void Execute(Character character)
    {
        character.Jump();
    }
}

static void Main(string[] args)
{
    // ...
    Character activeHero;
    ICharacterCommand command = inputHandler.GetCommand();
    command.Execute(activeHero);
}
```

Undo & Redo

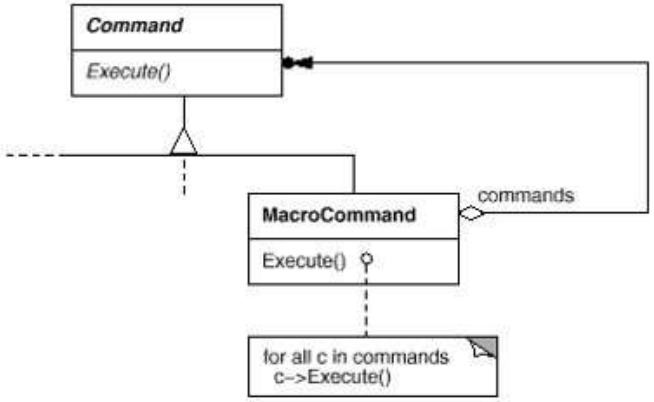
- Preserve receiver state
  - **Memento**
- Copy commands
  - **Prototype**
- Alternatively add inverse operation

Undo & Redo

```
interface ICommand
{
    void Execute();
    void Undo();
}

class DeleteTextCommand : ICommand
{
    private Document document;
    private Selection deleted;

    DeleteTextCommand(Document doc)
    {
        document = doc;
    }

    void Execute()
    {
        deleted = document.GetSelection();
        document.RemoveText(deleted.Start, deleted.Content.Length);
    }

    void Undo()
    {
        document.InsertText(deleted.Start, deleted.Content);
    }
}
```

Macro

- Multiple sequential commands
- **Composite**




```
class MacroCommand : ICommand
{
    private List<ICommand> commands;

    void Execute()
    {
        foreach (ICommand c in commands)
            c.Execute();

        // ...
    }
}
```

Functional

```
type Command = () => void;

function startCarFactory(car: Car) {
  return function () {
    car.start();
  }
}

interface MenuItem {
  text: string;
  command: Command;
}

class Menu {
  private items: MenuItem[];

  constructor(...items: MenuItem[]) {
    this.items = items;
  }
}
```

```
const myCar = new Car();
const carMenu = new Menu(
  {
     text: "start car",
     command: startCarFactory(myCar)
  }
);
```

Known uses

- Multilevel undo & redo
- Macro recording
- GUI toolkits
- Task queue
- Remote execution
- Wizards
- Transactions

Related design patterns

- **Prototype**
- **Composite**
- **Memento**
- **Chain of Responsibility**

Summary

- **When to use:**
  - Parametrize objects with actions
  - Undo
  - Task queue
  - History logging
- **Pros**
  - Caller & callee decoupling
  - Extensibility
- **Cons**
  - Large amount of new classes