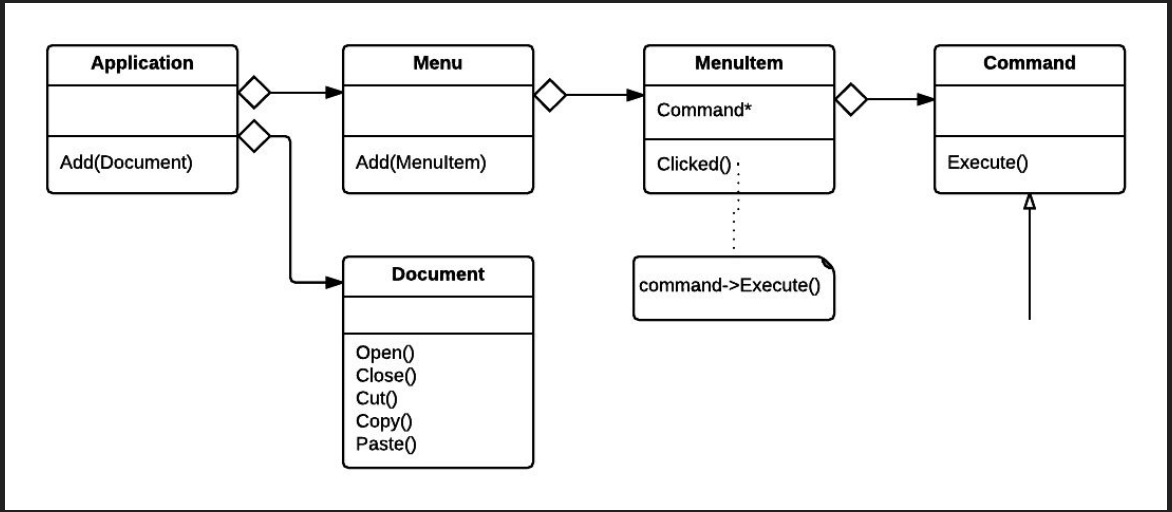
Command Pattern

Behavioral, object focused

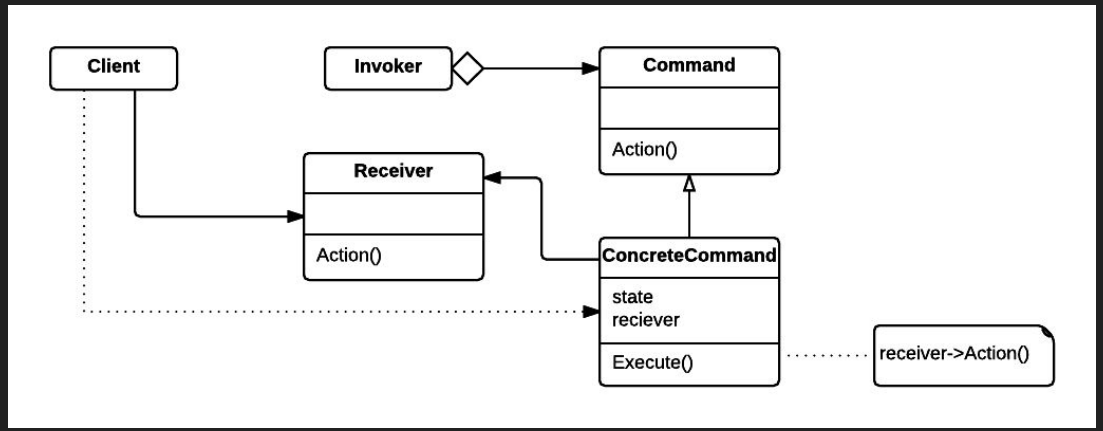
Motivation

Encapsulate a request into an object that is self-contained such that it can be executed at a different time or location from when it is created

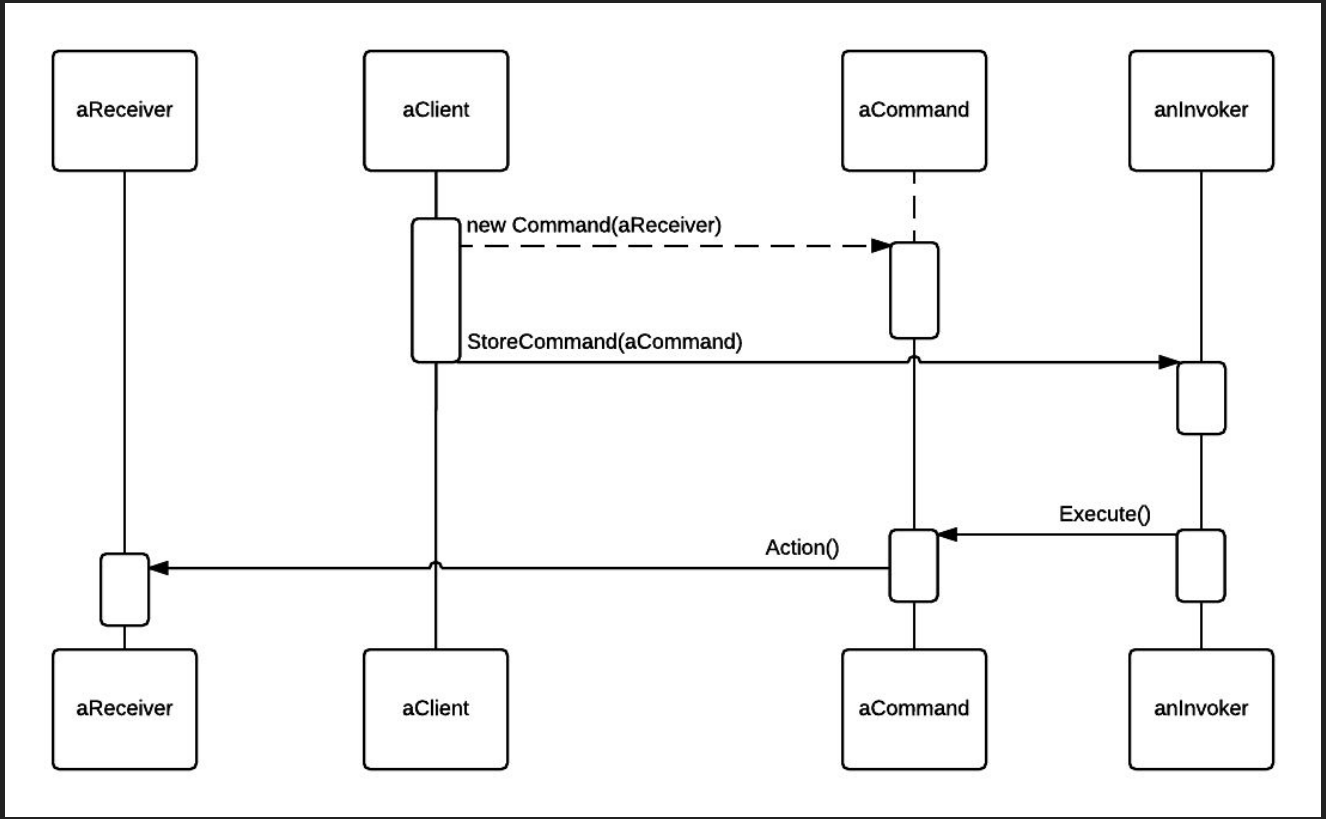


Structure

- **Command -** Declares interface for executing an operation
- **ConcreteCommand -** Binding between `Receiver` and `Action()`
- **Client -** Creates a `ConcreteCommand` object and sets `Receiver`
- **Invoker -** Asks the command to carry out the request
- **Receiver -** Knows how to perform the operation for a request

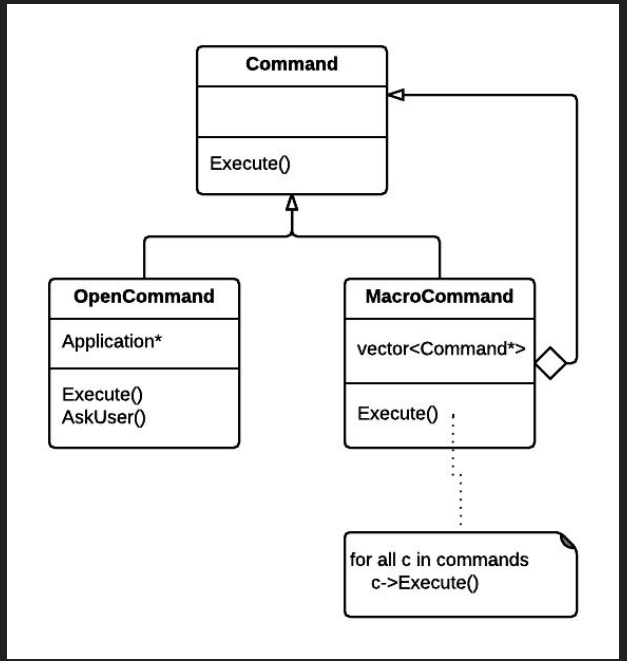


Interactions



Motivation

- Sequences of commands can be executed using the composite pattern
- A `MacroCommand` (composite) can be created
- This composes several commands in sequence

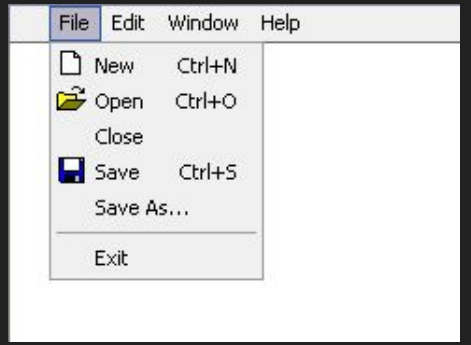


Example from “Design Patterns: Elements of Reusable Object-Oriented Software”

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides

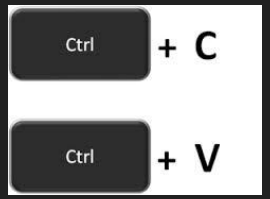
User Operations

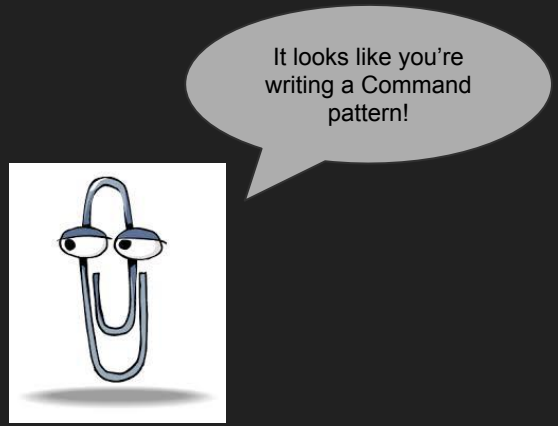
- Some of Lexi's functionality is available through the WYSIWYG interface:
  - Entering and deleting text
  - Moving intersection point
  - Select range of text by pointing
  - Clicking
  - Typing
- Other functionality is available through the pull-down menus, buttons, and keyboard shortcuts



User Operations

- A particular operation shouldn’t associate with a particular user interface
- Think about copy and paste operations
  - Implemented through a menu
  - ... through keyboard shortcuts
  - ... through Clippy ™





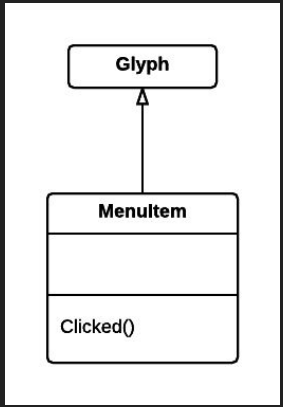
User Operations

- Lexi should also support undo and redo of most but not all of its functionality, undo a copy or paste, but not a save

Encapsulating a Request

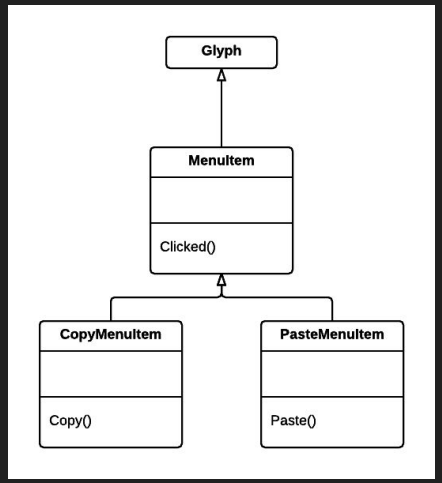
- Pull down menus are just another kind of glyph that contain other glyphs
- The distinguishing factor is menu glyphs do some work in response to an up-click

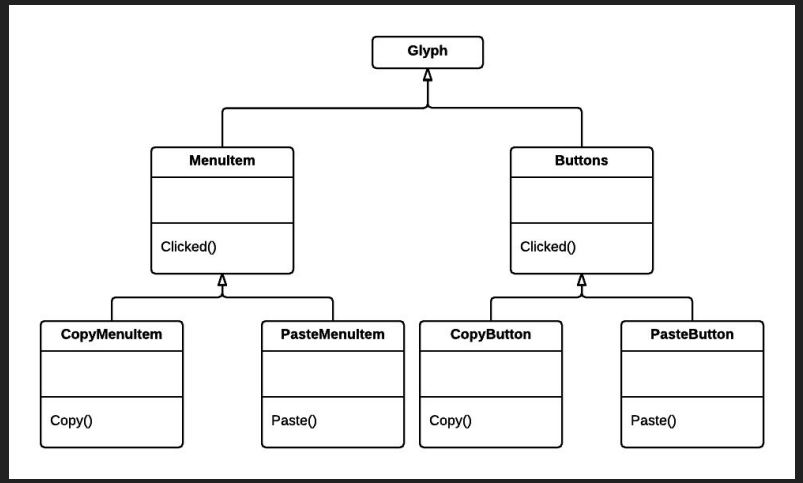
Assume you have a `Glyph` subclass **`MenuItem`**



What not to do

- Define a subclass of MenuItem for every user operation
- Hard code each subclass to carry out the request
- What if we want another interface?





Encapsulating a request

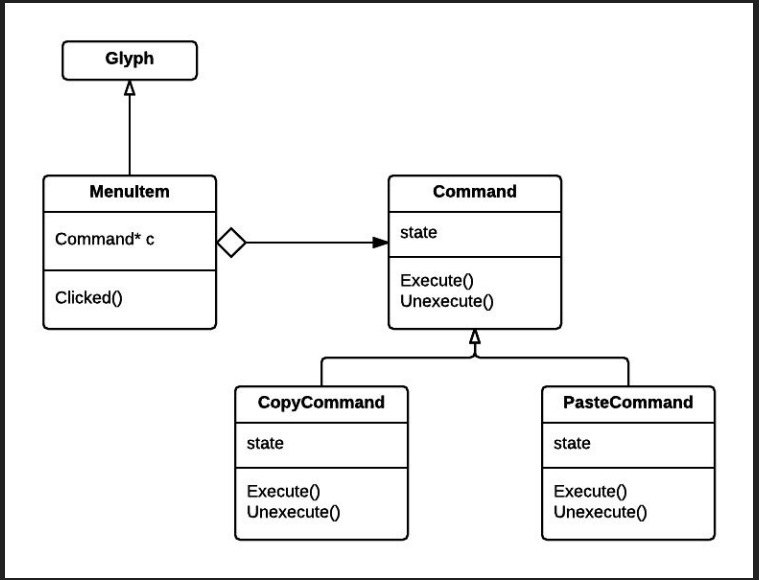
- We are unable to parameterize menu items by the request they should fulfill
- We could update this with a function to call, **but**
  - How do we undo/redo?
  - How would we associate state with the function?
  - How would we extend the functions? How would we reuse part of them?

Let’s parameterize `MenuItems` with an *object* instead

`Commands` as an Object

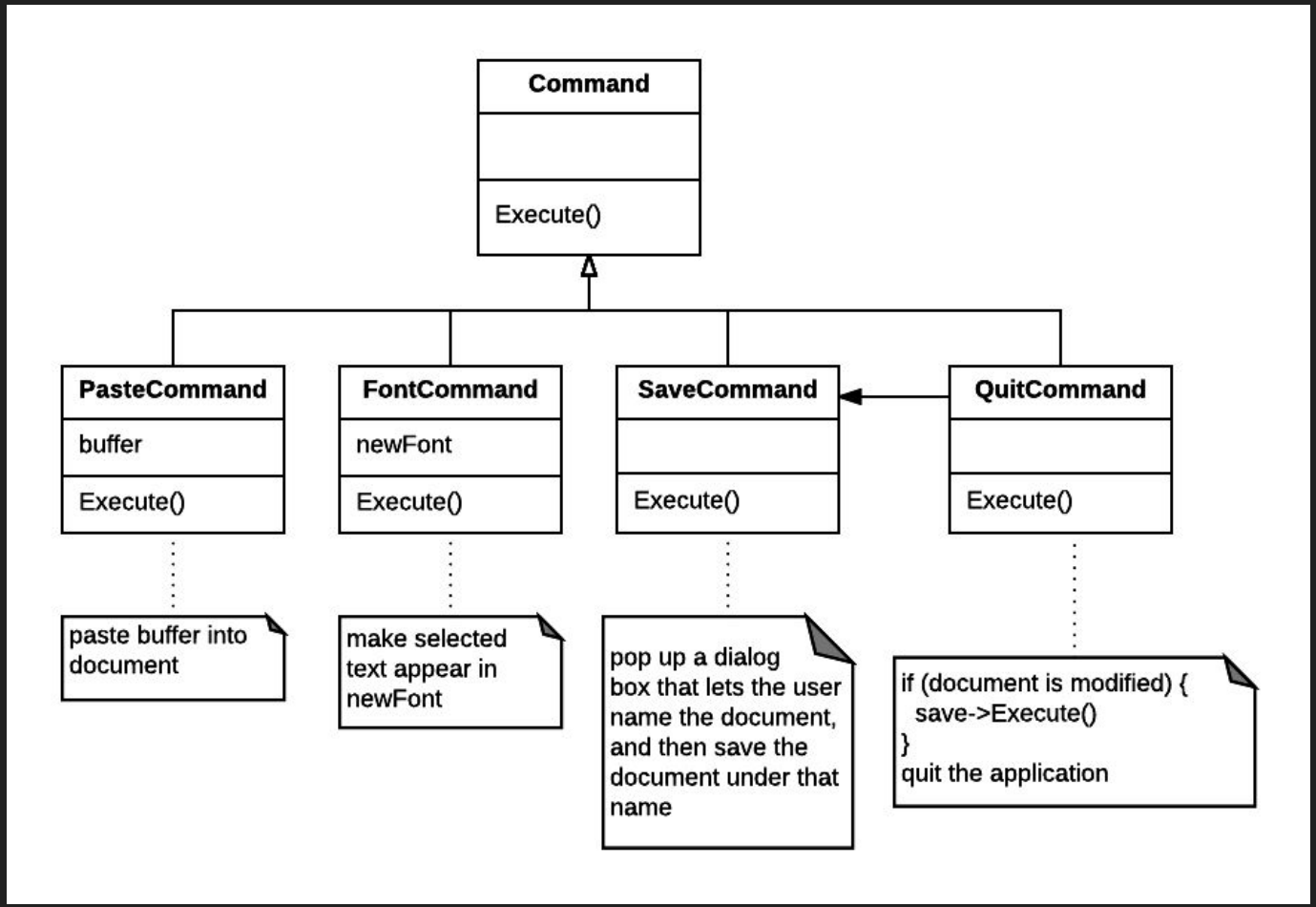
- `MenuItem` can be parameterized with a `Command` *object*
- The Command object can
  - Maintain state
  - Be extended through inheritance
  - Implement undo/redo

`Command` is an **abstract base class** that defines the interface for the inherited **concrete classes**



`Command` Interface and subclasses

1. Define a **`Command`** abstract class to provide the interface for issuing a request
   a. This basic interface is a single abstract operation called "`Execute()`"
2. Particular commands will implement a concrete subclass of **`Command`**
   a. Copy, Paste, Save, Open, etc.
3. Subclasses will define the "`Execute()`" operation in different ways
   a. Some delegate part or all of the work to other objects, `Receiver`
   b. Some do all the work themselves

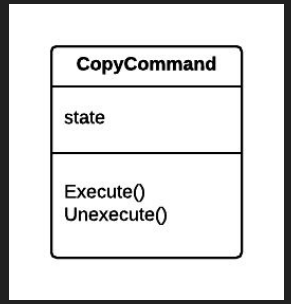

Command
Execute()
PasteCommand
buffer
Execute()
FontCommand
newFont
Execute()
SaveCommand
Execute()
QuitCommand
Execute()
paste buffer into document
make selected text appear in newFont
pop up a dialog box that lets the user name the document, and then save the document under that name
if (document is modified) {
save->Execute()
}
quit the application

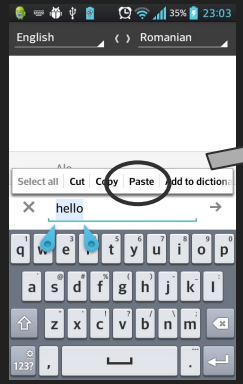
Executing a command

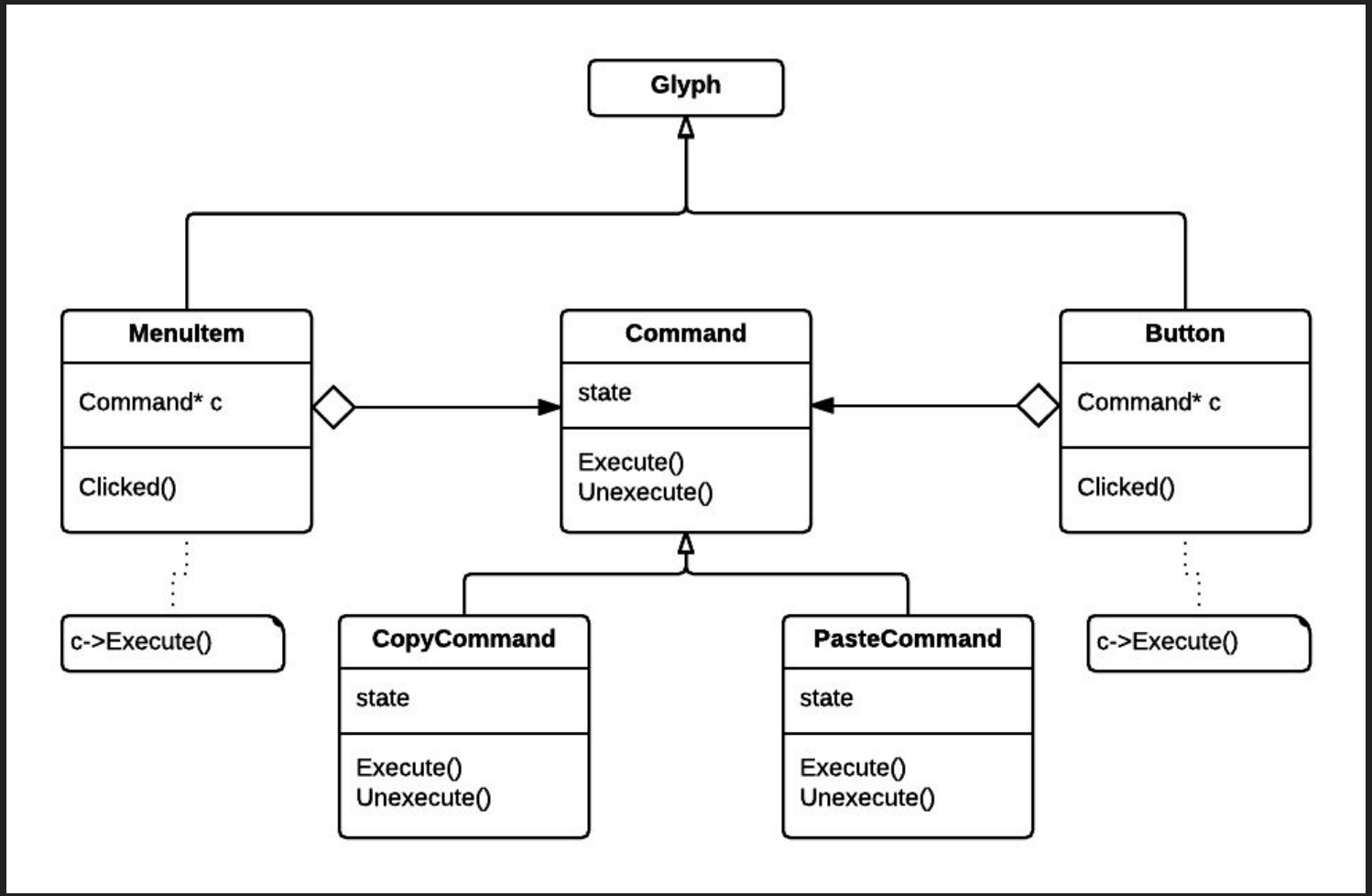
1. `MenuItems` store a `Command` object which encapsulates a **request**
2. When the `MenuItem` is selected/clicked it calls `Execute()` on its `Command` object to carry out the **request**
3. The `Command` object carries out the request through its `concreteCommand` interface

Encapsulated Commands

Buttons and widgets can use commands the same way!

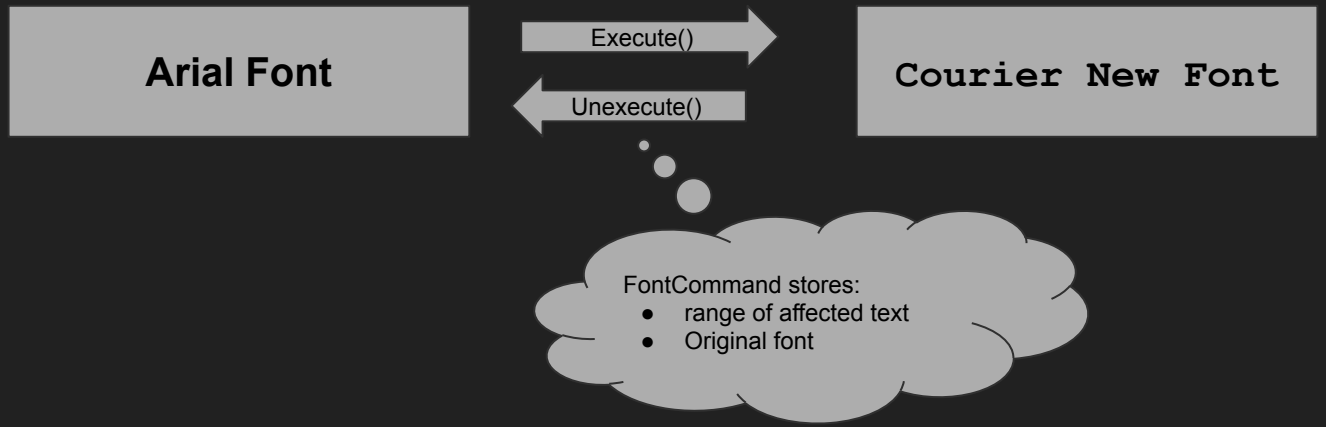





Glyph
MenuItem
Command* c
Clicked()
Command
state
Execute()
Unexecute()
Button
Command* c
Clicked()
c->Execute()
CopyCommand
state
Execute()
Unexecute()
PasteCommand
state
Execute()
Unexecute()
c->Execute()

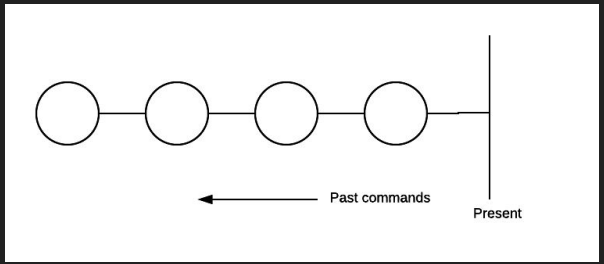
Undo and Redo

- The `Command's` interface is extended with an `Unexecute()` operation
  - `Unexecute()` reverses the effects of the preceding `Execute()` operation
- In the case of `FontCommand`:

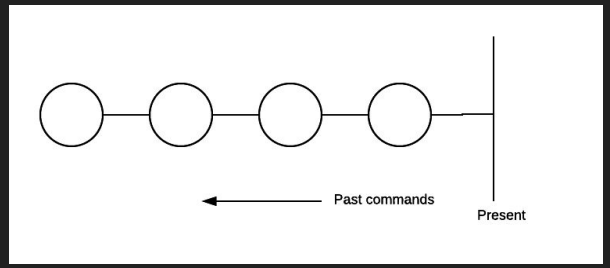


Command History

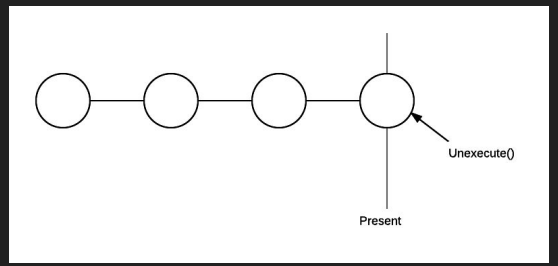
- Arbitrary number of undo and redo commands need to be executed
- a **Command History** is defined to record executed/unexecuted commands



Command History

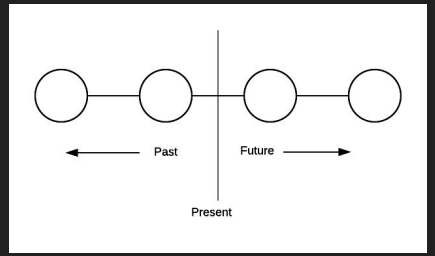


1. Unexecuting() a command moves the “present” line one command to the left



Command History

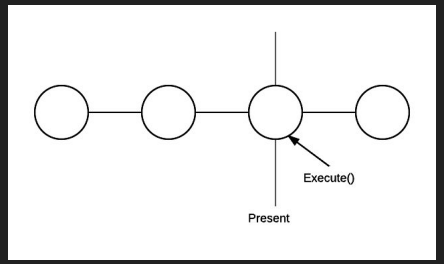
- Unexecute() more commands and you can end up in this state
  - Multiple commands in the “future” and the “past”

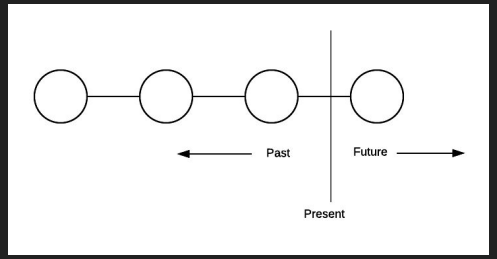


- Repeating this procedure gives us multiple levels of undo
- The length of the **history** gives us the limit on the number of commands we can *undo*

Command History

- We are also able to *redo* a command that has been *undone* by doing the same process but in reverse



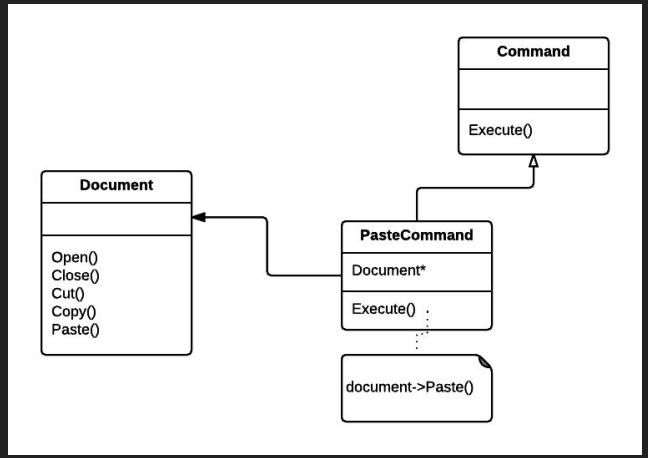


Command Pattern

- Lexi's commands are an application of the **Command Pattern**, which describes how to encapsulate a request
- The command pattern prescribes a uniform interface for issuing requests that lets you configure clients to handle different requests
- The interface shields clients from the request's implementation
- A command may delegate all, part, or none of the request's implementation
- This is perfect for applications like Lexi that must provide centralized access to functionality scattered throughout the application
- The pattern also discusses undo and redo mechanisms built on the basic Command interface

Motivation

- `PasteCommand` has a `Document` as a receiver
- A `MenuItem` can call execute on `PasteCommand` without knowing what it is doing
- `PasteCommand` knows what to do and who to do it to



Motivation

- `OpenCommand` has a slightly different `Execute()` command
  1. Ask user for the name of the doc
  2. Create a new document object
  3. Add the document to the receiving application
  4. Open the document