SWEN 262
Engineering of Software Subsystems

Command Pattern

Clipboard Coding*

1. The word processing application shall allow users to copy the currently selected text to the system clipboard.
   a. Using a *Copy* option in the *Edit* menu.
   b. Using a keyboard shortcut: *CTRL-C*
   c. Using the *Copy* option in the context menu (right click).
   d. Using the *Copy* icon in the toolbar.
2. The application shall allow users to paste the contents of the system clipboard into the current document.
   a. Using a *Paste* option in the *Edit* menu.
   b. Using a keyboard shortcut: *CTRL-V*
   c. Using the *Paste* option in the context menu (right click).
   d. Using the *Paste* icon in the toolbar.

Copy

Paste

Q: How might you go about implementing this requirement?

* See what we did there?

Embedded Code

```
public void menuClicked() {
  Clipboard clipboard = application.getClipboard();
  String contents = clipboard.getContents();
  Document document = application.getDocument();
  document.paste(contents);
}
```

```
public void keyboardShortcutUsed() {
  Clipboard clipboard = application.getClipboard();
  String contents = clipboard.getContents();
  Document document = application.getDocument();
  document.paste(contents);
}
```

A: Embed the code into each of the widgets (buttons, menus, etc.) that can be used to perform the paste function.

Q: What are the drawbacks to this solution?

A: The most obvious is code duplication. What else?

A number of custom widgets must be created by extending buttons, menu items, etc. This leads to class explosion and lower cohesion (how?).

There is also a high degree of coupling between the application, document, clipboard, and various widgets.

A Paste Method

```
public void paste() {
  Clipboard clipboard = getClipboard();
  String contents = clipboard.getContents();
  Document document = getDocument();
  document.paste(contents);
}
```

```
public void menuClicked() {
  application.paste();
}
```

```
public void keyboardShortcutUsed() {
  application.paste();
}
```

A: Use *extract method* to encapsulate the code in a method, and call that method from each of the appropriate widgets.

Q: What are the drawbacks to this solution?

A: Again, a number of custom widgets must be created and coupled with the main application. This causes class explosion and violates *single responsibility* (why?).

A Command Interface

Begin by defining an `interface` to represent a ***command*** that can be executed by the user in the word processing application.

```
public interface Action {
  public void performAction();
}
```

Next, create a ***concrete command*** to implement the copy action.

```
public class Copy implements Action {
  private WordProc application;

  public void performAction() {
    application.copy();
  }
}
```

Create a separate ***concrete command*** for each user action, e.g. paste, save, open, etc. Each will call some method(s) on a specific ***receiver***, e.g. the main application class that defines the copy and past methods.

Generic Widgets

```
public class Button {
  private Action action;

  public Button(Action action) {
    this.action = action;
  }

  public void buttonClicked() {
    action.performAction();
  }
}
```

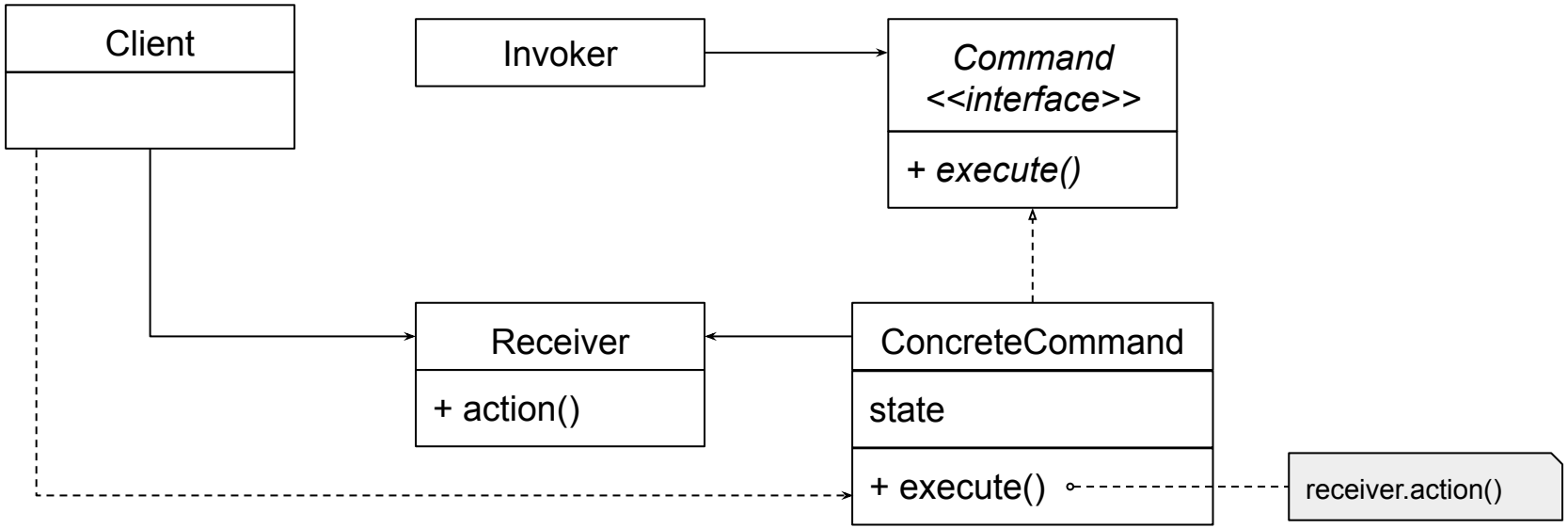
Modify each of your widgets (buttons, menu items, etc.) so that it can be created with an instance of your ***command*** interface.

When the user interacts with the widget, it ***invokes*** the method on its ***command***.

The widget does not need any information about what the ***command*** actually *does*. It just needs to ***invoke*** its command.

This means that the same, generic widget can be reused many times in the application with different ***commands***.

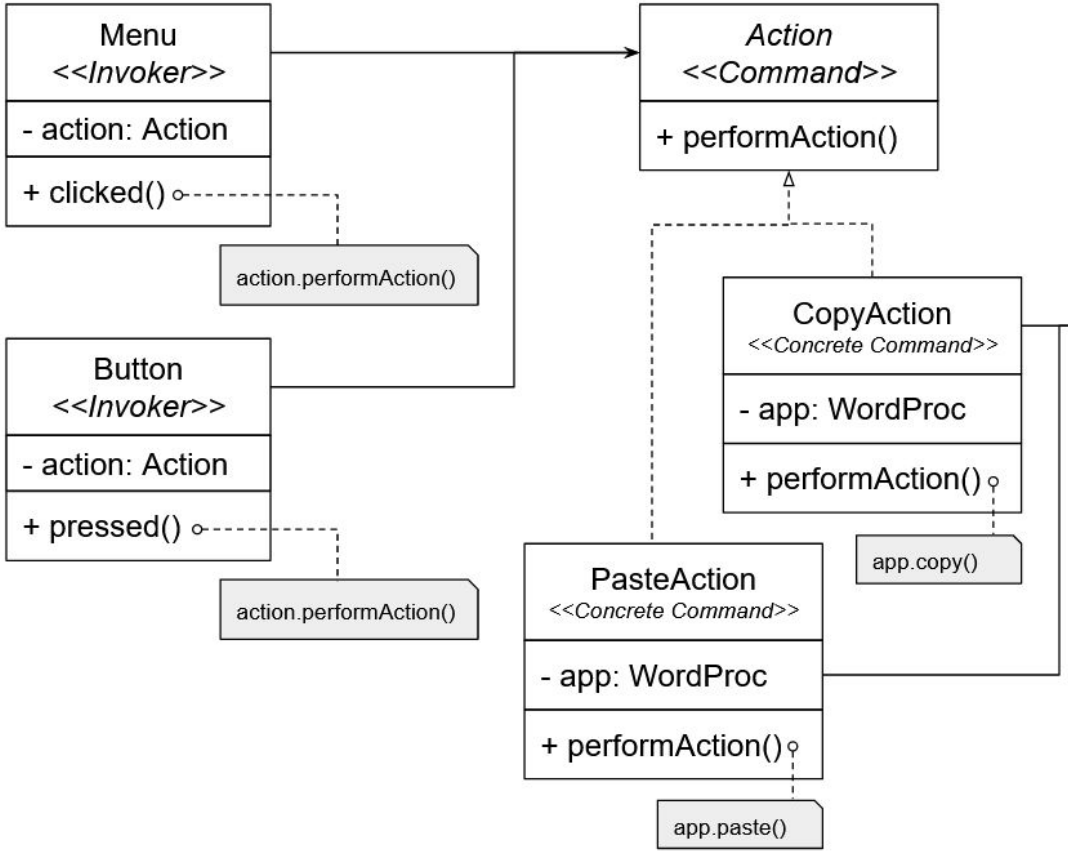
GoF Command Structure Diagram



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

(Behavioral)

Copy/Paste Command Design



Each widget (button, menu, etc.) is the ***invoker*** of some ***command***.

Each ***concrete command*** executes one or more methods on some ***receiver***, e.g. the main application.

The ***invokers*** do not need to know anything about the specific ***command*** - only when to ***invoke*** it.

GoF Pattern Card

Sorry about the eye chart, but this is a lot of information to pack into one slide!

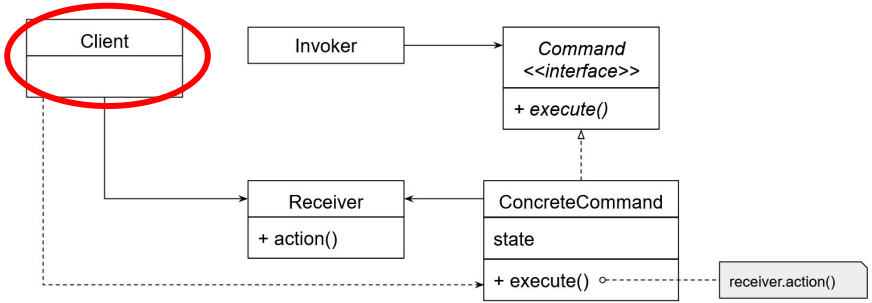
Note that each ***concrete command*** is documented ***separately*** - they are ***not*** combined into a single row.

One of ***the most common*** errors in documentation of the Command Pattern is combining multiple commands (up to an including all of them) into the same row.

Remember, your documentation should be written as though a separate engineering team is going to implement the system. Include sufficient detail!

Name: *Copy Paste Subsystem*		GoF Pattern: *Command*
Participants		
Class	Role in Pattern	Participant's Contribution in the context of the application
WordProc	*Client, Receiver*	*The main word processing application. The application is responsible for creating concrete commands and binding them to their receivers when the application loads. As the information expert for both the clipboard and the currently opened document, it is also the receive for the copy and paste commands.*
Action	*Command*	*Defines the interface for a user action in the word processing application. The actionPerformed method is invoked each time that the command should perform its related task.*
Menu	*Invoker*	*One of many generic menu items. Each menu item is associated with a specific action. When the menu item is used, it invokes the action.*
Button	*Invoker*	*One of many generic buttons. Each button is associated with a specific action. When the button is clicked, it invokes the action.*
Copy	*ConcreteCommand*	*A concrete command that performs a copy operation by calling the copy method on the word processing application. This copies the currently selected text to the system clipboard.*
Paste	*ConcreteCommand*	*A concrete command that performs a paste operation by calling the paste method on the word processing application. This pastes the contents of the system clipboard into the currently opened document.*
Deviations from the standard pattern: *The main application is both the Client and the Receiver in this implementation.*		
Requirements being covered: *1.a-1.d - Users shall be able to copy selected text using menus, buttons, etc. 2.a.-2.d. Users shall be able to paste copied text using menus, buttons, etc.*		

Where is the Client?

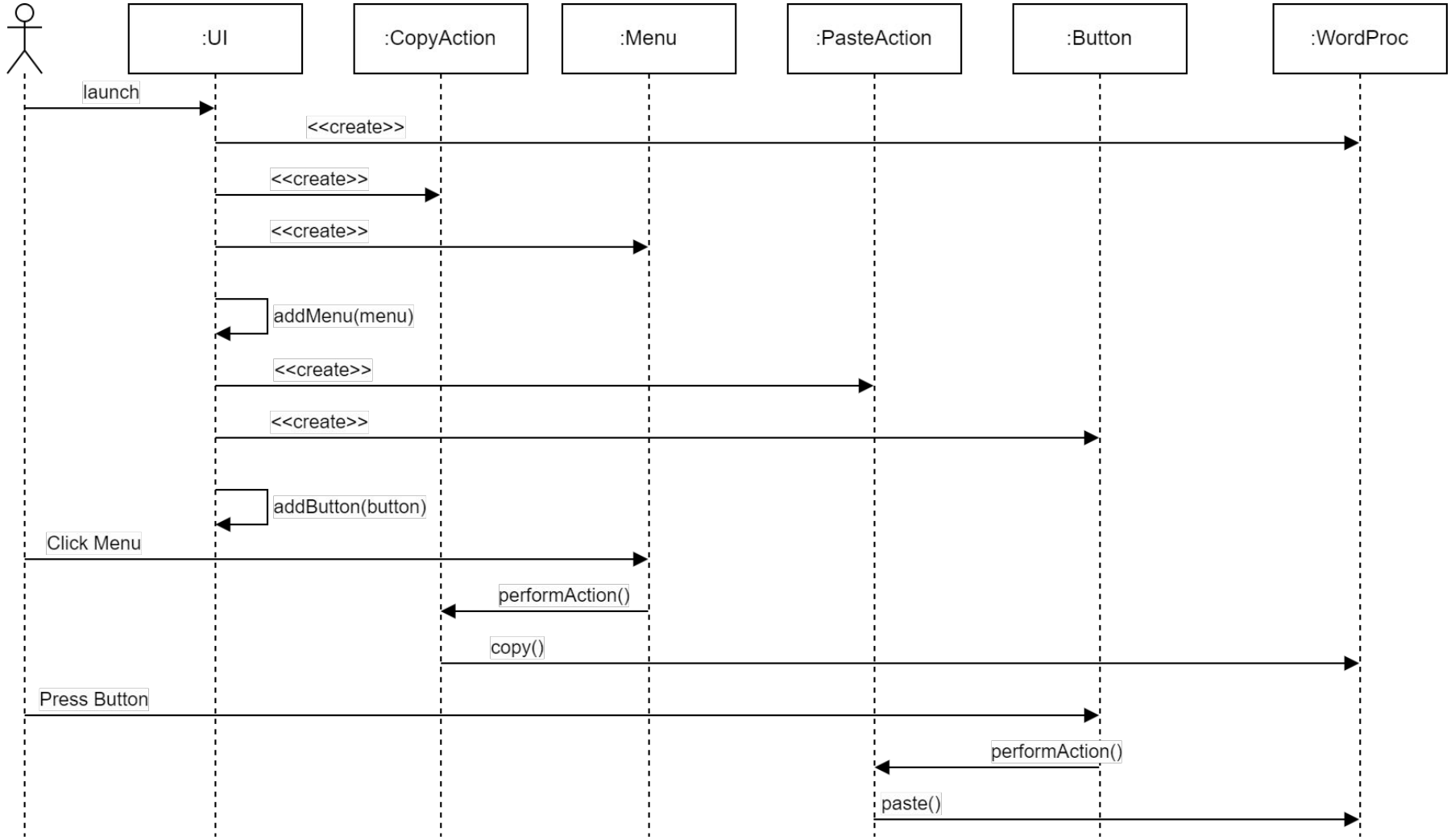


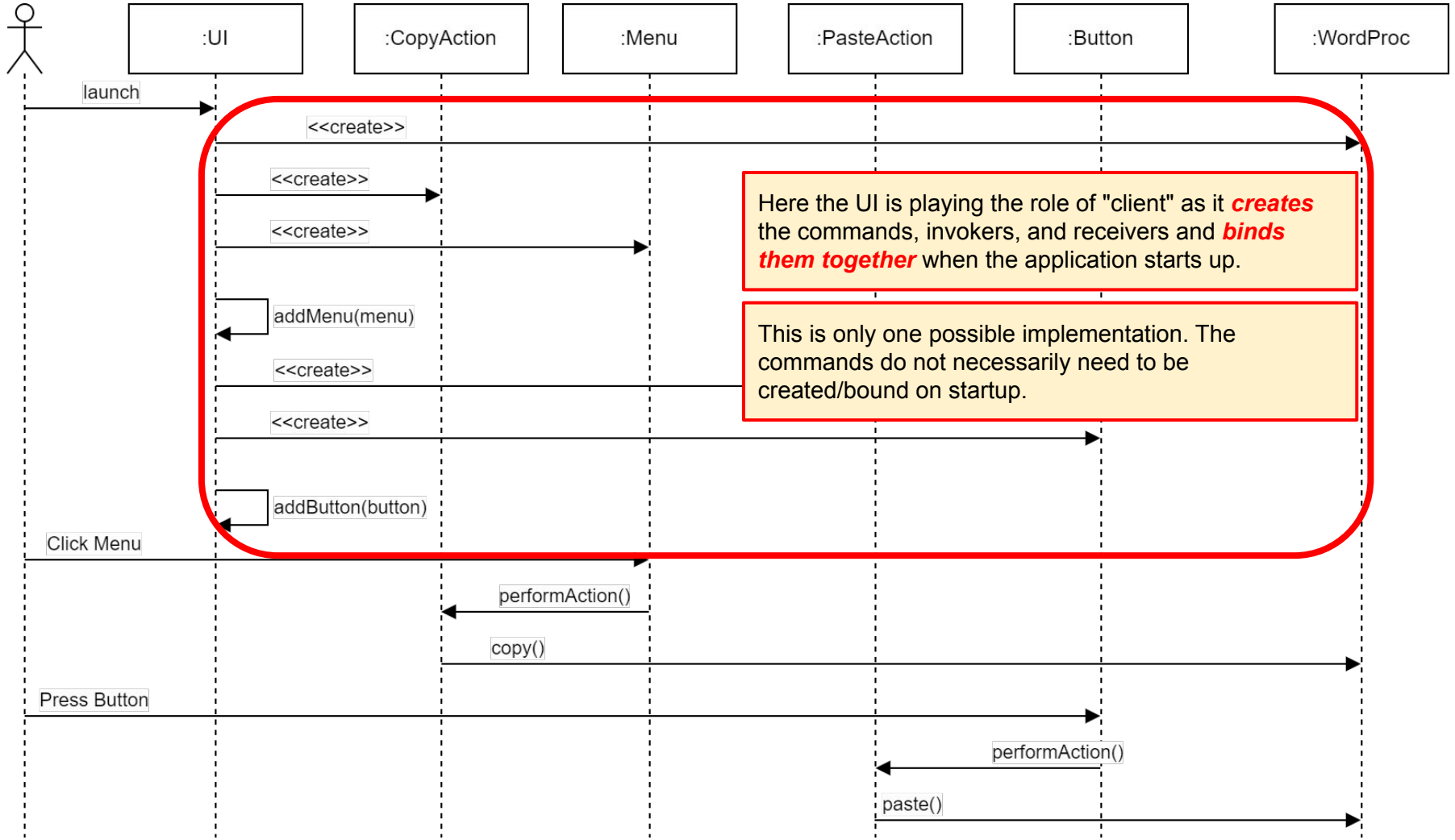
- The ***client*** in the Command Pattern plays a somewhat different role than the client in other patterns that we have studied; it is the part of the system that instantiates a ***concrete command*** and binds it to its ***receiver***.
- Depending on the implementation, this may occur only once when the system starts up.
  - The ***client*** may not be involved after that.

Some implementations of the Command Pattern will create all of the ***concrete commands*** at startup and reuse them.

In this case, the ***client's*** role is brief (but important).

In other implementations, ***concrete commands*** are created on demand - the ***client*** will be involved throughout the lifetime of the application.


:UI
:CopyAction
:Menu
:PasteAction
:Button
:WordProc
launch
<<create>>
<<create>>
<<create>>
addMenu(menu)
<<create>>
<<create>>
addButton(button)
Click Menu
performAction()
copy()
Press Button
performAction()
paste()


:UI
:CopyAction
:Menu
:PasteAction
:Button
:WordProc
launch
<<create>>
<<create>>
<<create>>
addMenu(menu)
<<create>>
<<create>>
addButton(button)
Click Menu
performAction()
copy()
Press Button
performAction()
paste()
Here the UI is playing the role of "client" as it creates the commands, invokers, and receivers and binds them together when the application starts up.
This is only one possible implementation. The commands do not necessarily need to be created/bound on startup.

Command

Encapsulating the details of performing an operation allows separation of concerns in *space* and *time*.

- Invocation is decoupled from execution.
  - *For example, in MVC the invoker (the view) is decoupled from the executor (the controller and/or model).*
- Execution can happen at a different time than invocation.
  - *How?*
- You can create sequences of commands for later execution.
  - *Commands are objects. They can be bundled together (e.g. in a collection) like any other objects.*
  - *How can this support macro commands?*
  - *How can this support undo/redo?*

Command Decisions

The Command pattern provides the designer with several choices:

- How smart is the command object?
  - *Only binds the command to the receiver and action.*
  - *Performs the operation itself.*
- When is a command instantiated?
  - *Prior to invocation.*
  - *Upon invocation.*
- When is the receiver bound to the command?
  - *When the command is instantiated.*
  - *When the command is invoked.*

Consider that part of the intent of the Command Pattern is to *"...support undoable operations...."*

Q: How might such a requirement impact how "smart" the commands need to be? Or when the commands are instantiated?

Command

There are several *consequences* to implementing the command pattern:

- *Decouples the object that invokes an operation from the object that knows how to perform it.*
- *Commands are first-class objects that can be manipulated and extended like any other object.*
- *Commands can be assembled into composite commands (e.g. macros).*
- *New commands can be easily added because existing classes do not need to be changed (similar to Strategy).*
- *Lots of little command classes.*

Things to Consider

1. How does Command affect the overall cohesion in the system?
2. The coupling?
3. How does it support the Open/Closed Principle?
4. What other design principles might command make better or worse?
5. How would you decide when to instantiate concrete commands?