Design Issue #8
User Operations

- Many possible operations
  - Document operations: create, open, save, print a document
  - Editing operations: select, copy, cut, paste, undo, redo
  - Formatting operations: text formatting, character formatting etc
  - Miscellaneous operations: context sensitive help
- Different interfaces for these operations
  - Different Look-and-Feel
  - Different Windowing Systems
  - Different Access Points (menu, shortcut key, context menu)
- Independence from the UI
  - UI is a possible trigger, but not the only one;
  - *What* is done should not depend on the UI

Design Issue #8

User Operations

- Other remarks:
  - The operations are implemented in many different classes
  - De-coupling: we want to access the functionality without adding dependency between the UI classes (i.e. object that issues the request) and all of the different classes involved (i.e. object that will perform the request)
  - We also want to support undo and redo for some functionality

Design Issue #8

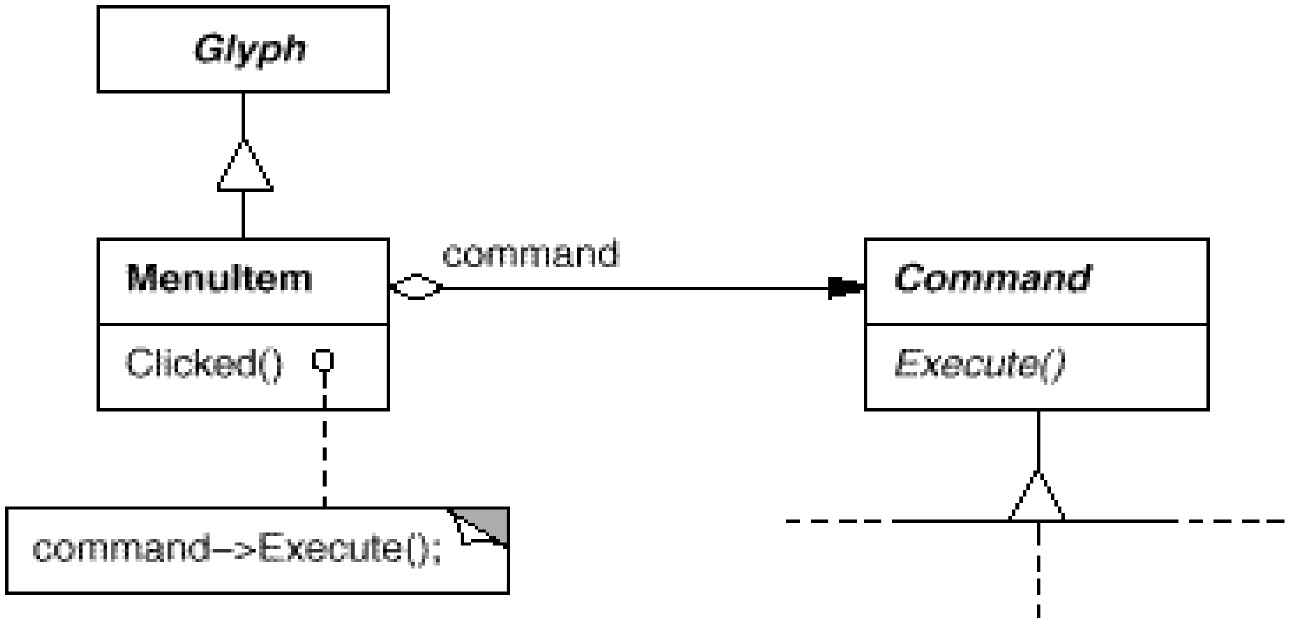
User Operations

Team exercise: Design the system to handle user requests; de-coupling from UI is key!

Design Issue #8

User Operations

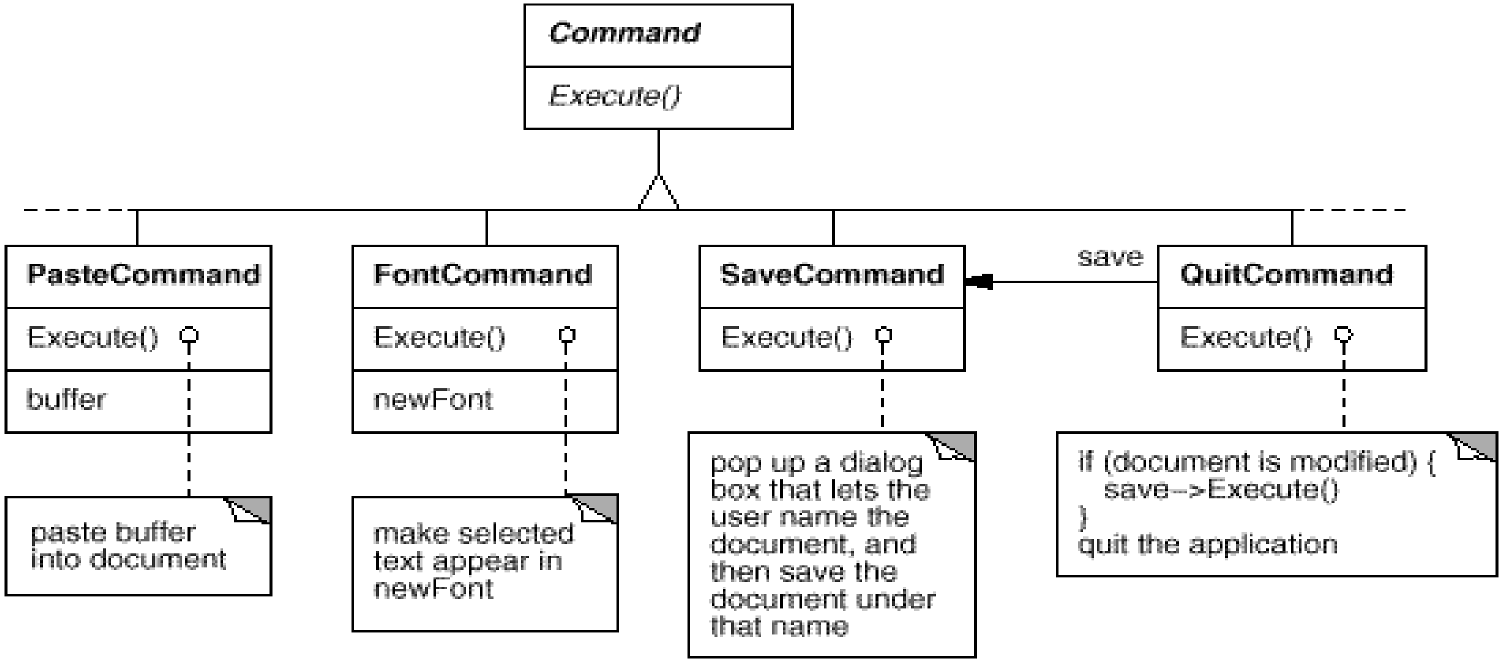
- We need to *encapsulate the request* using the **Command** Pattern
- Each MenuItem, ToolbarButton or KeyboardShortcut refers to an appropriate command object



Design Issue #8

User Operations

…and there is an entire hierarchy of specific commands, each class (command) being in charge with a specific user operation.



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

Naturally, the same command object will be pointed out (shared) by different triggers, like menu items, toolbar buttons etc. Hence, no matter what triggers an operation, the behavior is the same (consistency).

Design Issue #8

User Operations

Team exercise:

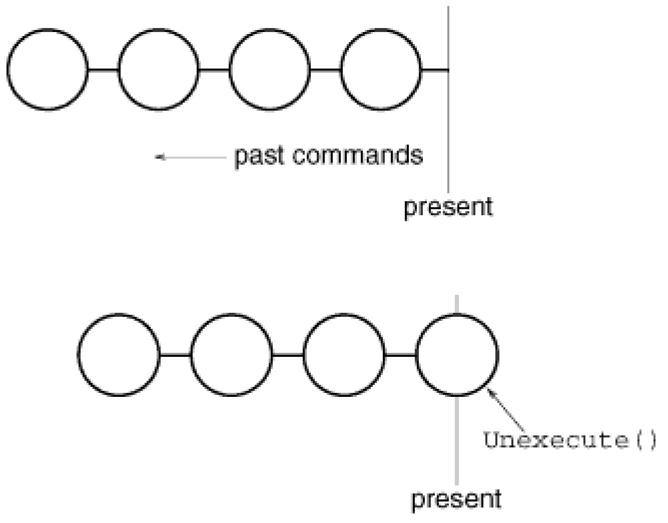
- Modify your design using Command pattern
- Think how to add Undo / Redo functionality to your system

Design Issue #8

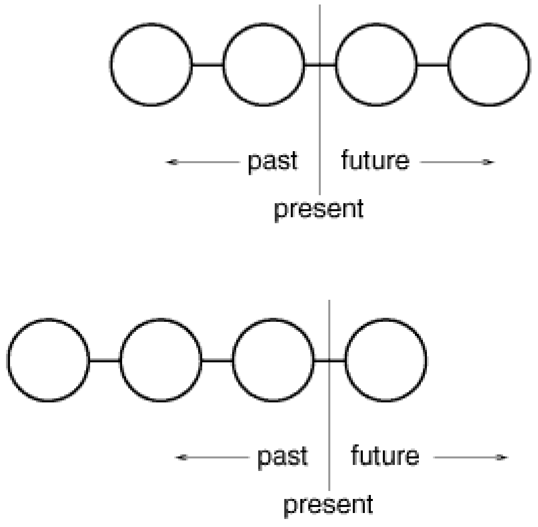
User Operations

- What about Undo? We add an Unexecute() method and keep a command history…
- …and a **Memento** will keep the state the command requires to undo (Unexecute) its effect.

UNDO



REDO



Design Issue #8
User Operations

- Tips for designing an Undo/Redo system?
  - Carefully choose what user operations are undo/redo-able; focus on the most important ones (e.g. delete, formatting)
  - Try to minimize the amount of data saved in each memento to avoid memory exhaustion
  - Decide upon undo buffer size: last action vs. limited N actions vs. unlimited number of actions
  - Display details about the Undo/Redo operation in UI
    - change the menu item name to ‘Undo delete’, ‘Undo paragraph formatting’ etc.
    - display the last M names of actions from the Undo buffer
  - Care must be taken to update the UI consequently

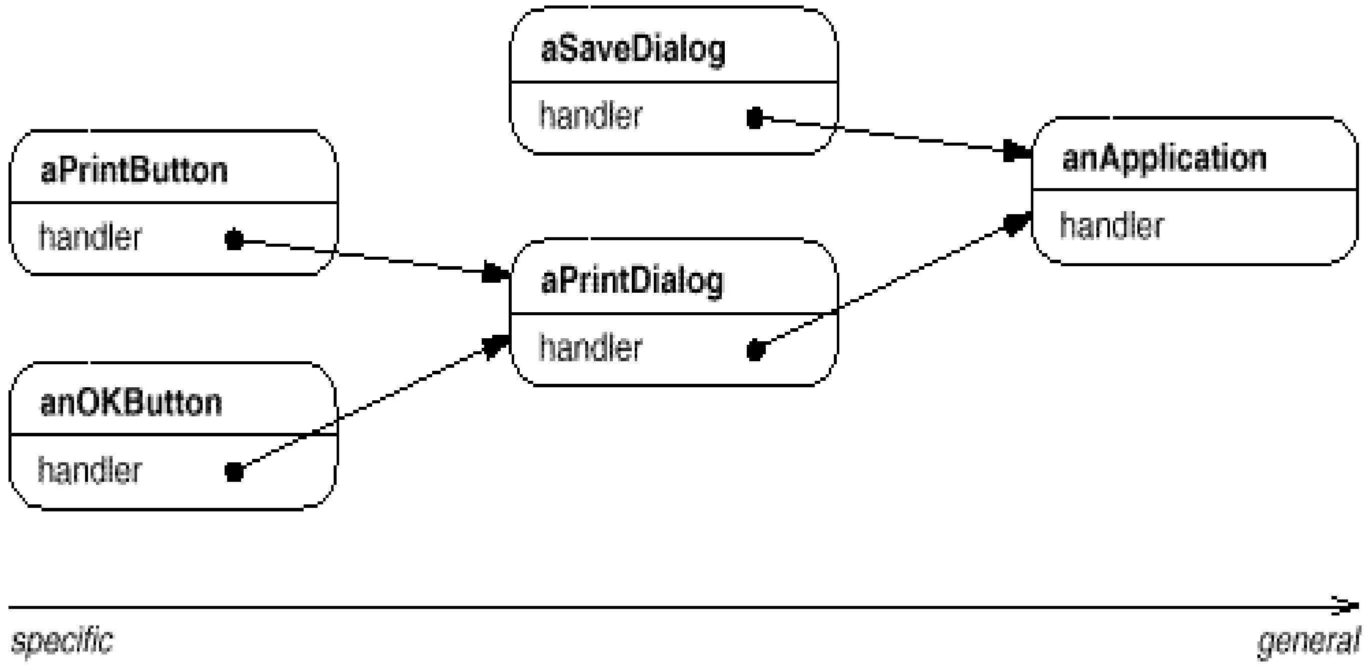
Design Issue #8

User Operations

Team exercise: Handling context-sensitive help

Design Issue #8
User Operations

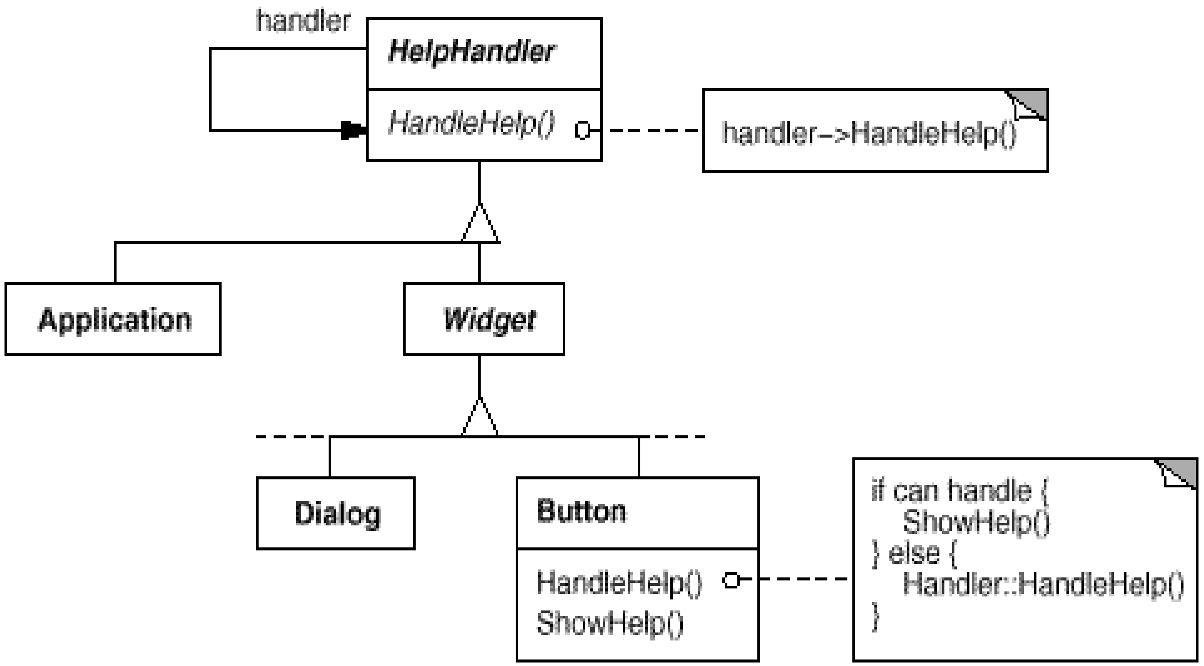
- How to handle context-sensitive Help?



Design Issue #8

User Operations

- **Chain of Responsibility** pattern allows to handle a request by multiple hadlers.
  - **Intent**: Avoids coupling the sender of a request to its receiver by giving more than one object a chance to handle the request.


handler
HelpHandler
HandleHelp()
handler->HandleHelp()
Application
Widget
Dialog
Button
HandleHelp()
ShowHelp()
if can handle {
ShowHelp()
} else {
Handler::HandleHelp()
}