Design Patterns

MSc in Communications Software

Produced by

Eamonn de Leastar (edeleastar@wit.ie)

Department of Computing, Maths & Physics
Waterford Institute of Technology

http://www.wit.ie

http://elearning.wit.ie

Waterford Institute *of* Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

Command

Design Pattern

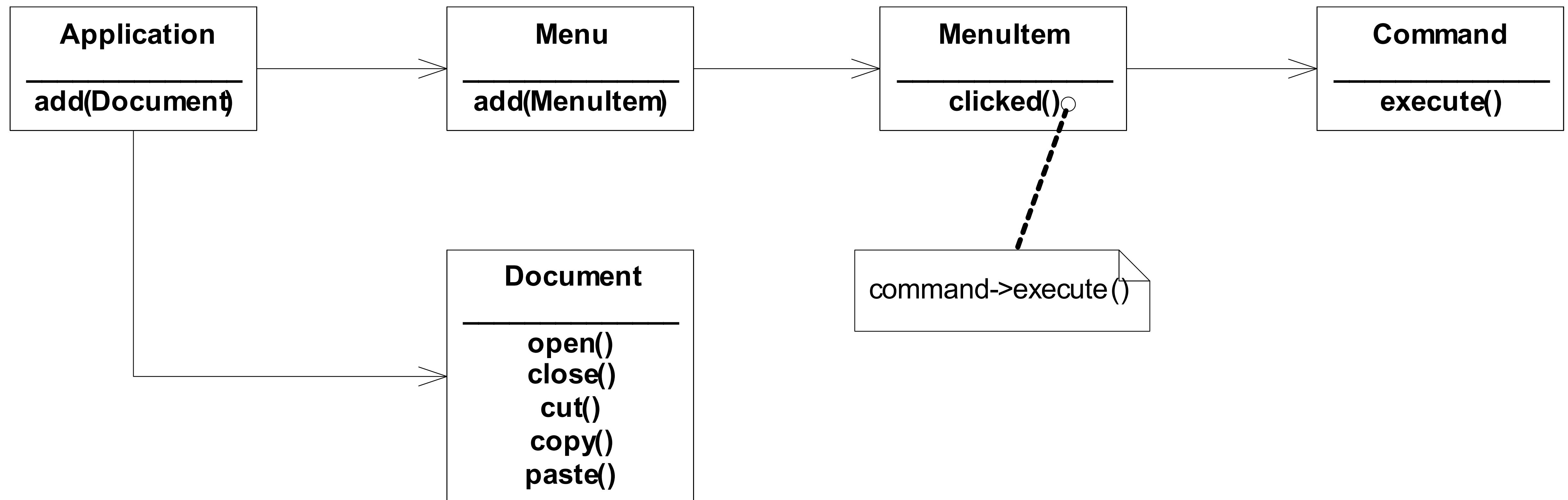
Intent

- Encapsulate a request as an object facilitating:
  - parameterize clients with different requests
  - queue or log requests
  - and support undoable operations.

Motivation(1)

- Be able to issue requests to objects without knowing anything about the operation (the command) or the receiver:
    - user interface toolkits include buttons & menus that carry out a request in response to user input.
    - toolkit can't implement the request explicitly in the button or menu, because only applications have this knowledge.
- The Command pattern turns the request itself into an object.
    - This object can be stored and passed around like other objects.
    - The key to this pattern is an abstract Command class, which declares an interface for executing operations.
- In the simplest form this interface includes an abstract Execute operation.

Motivation(2)

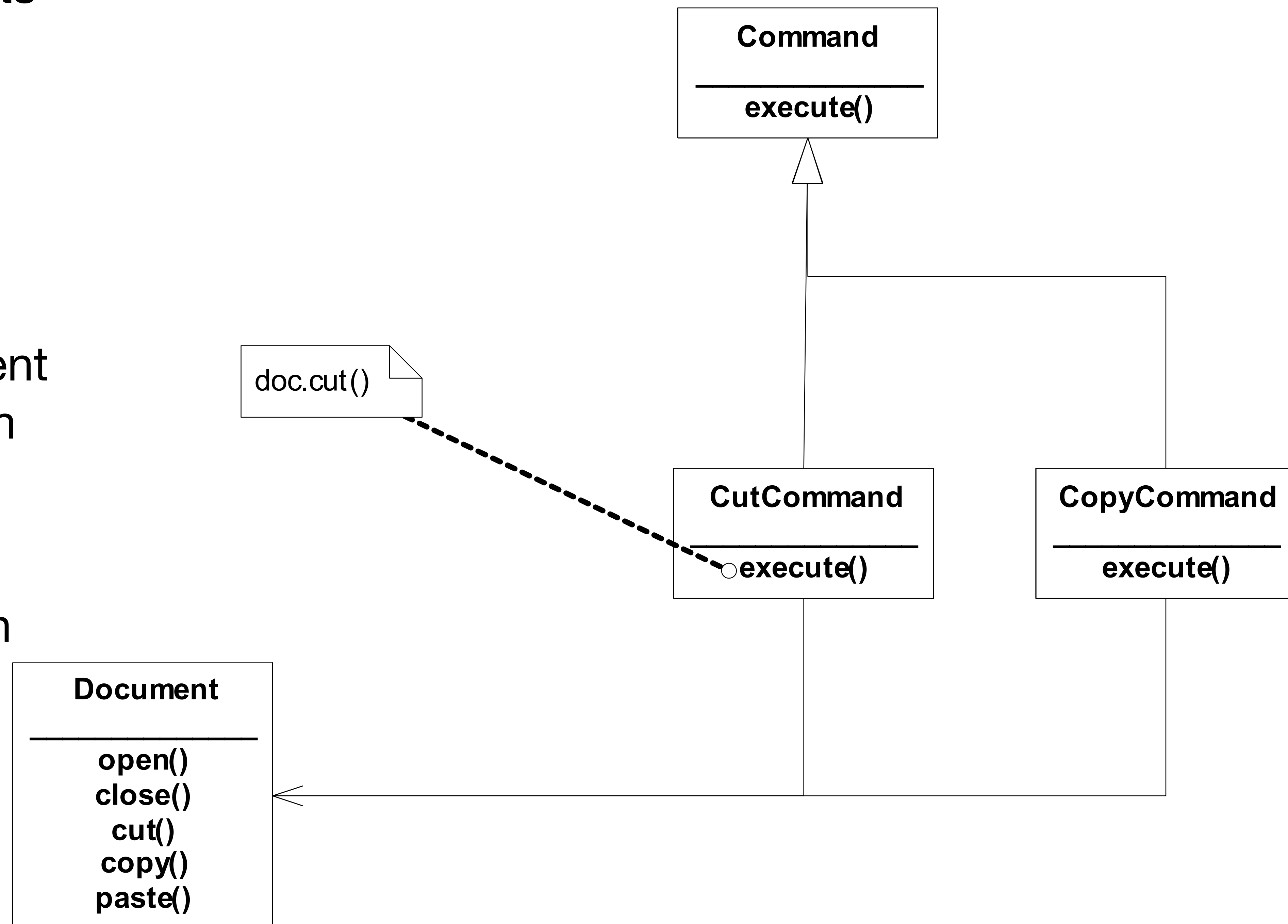


Motivation(3)

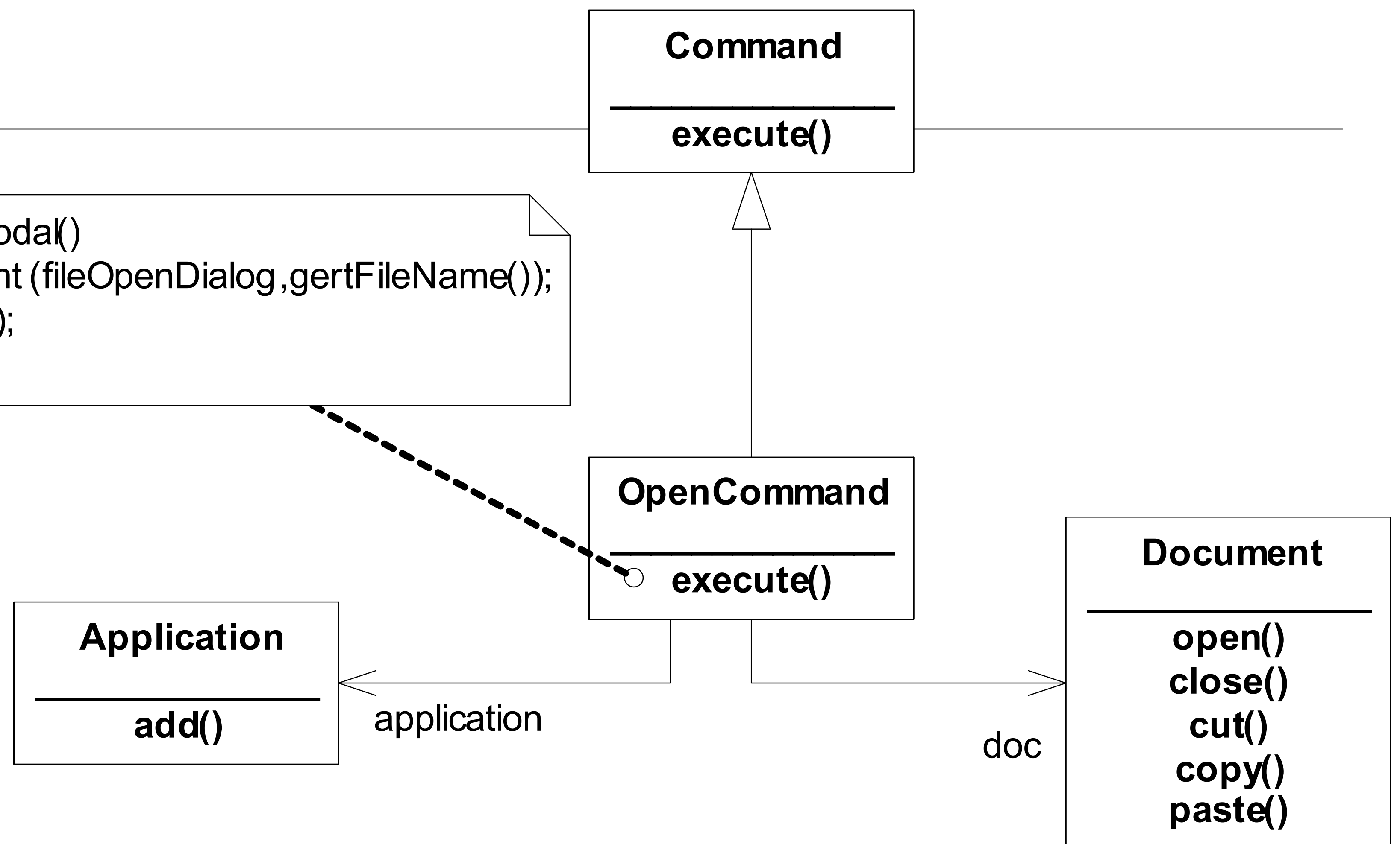
- Menus can be implemented with Command objects.
  - Each choice in a Menu is an instance of a MenuItem class.
  - An Application class creates these menus and their menu items along with the rest of the user interface.
  - The Application class also keeps track of Document objects that a user has opened.
- 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 may store the receiver of the request and invoke one or more operations on the receiver.

Motivation(4)

- CutCommand supports cutting text from the document to the clipboard
- CutCommand has a receiver - the Document object - supplied upon instantiation.
- The execute operation invokes cut() on the receiving Document

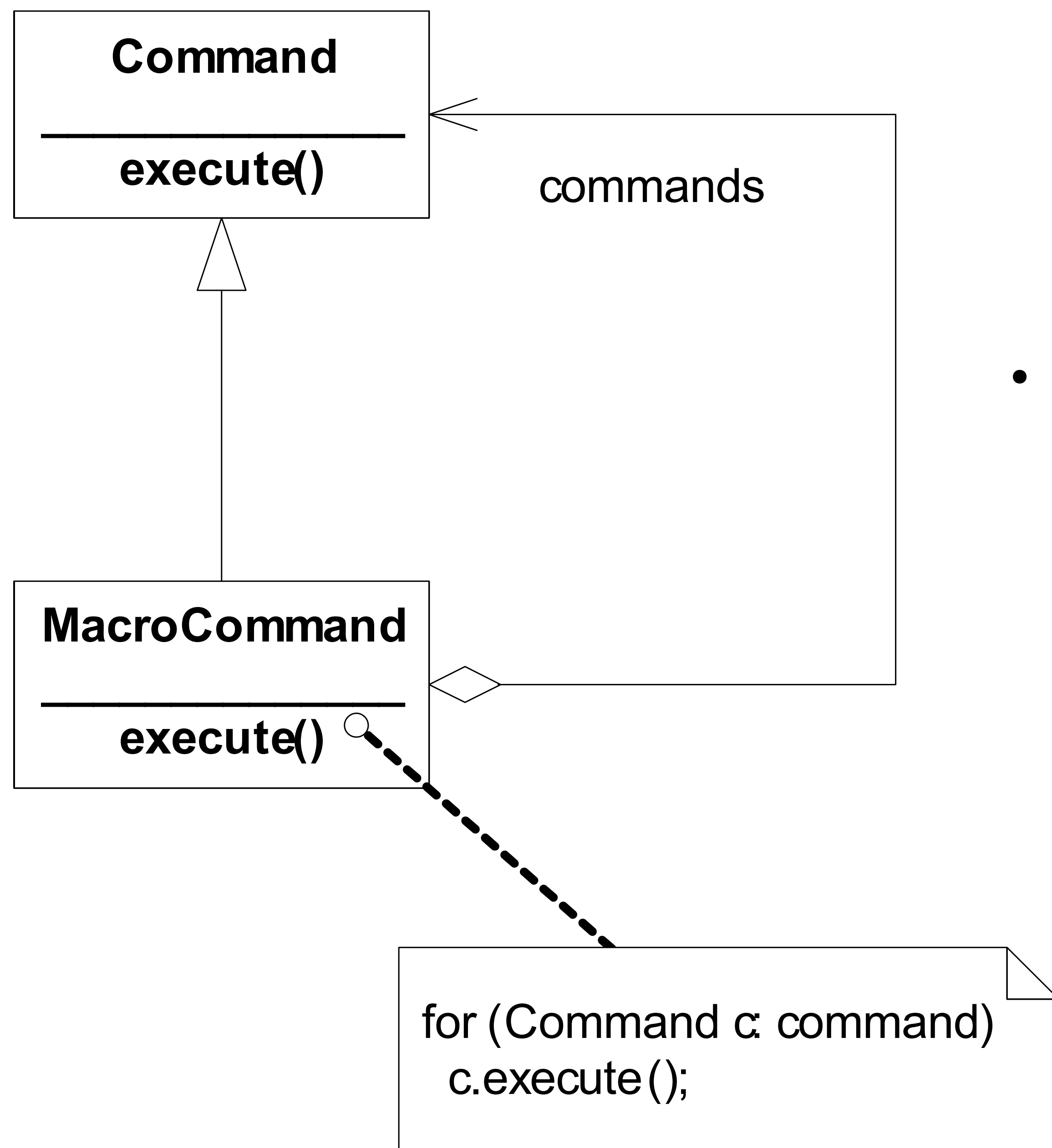


Motivation(5)



- OpenCommand's Execute operation could be different:
  - it prompts the user for a document name,
  - creates a corresponding Document object,
  - adds the document to the receiving application,
  - and opens the document.

Motivation(6)



- A MenuItem may need to execute a sequence of commands.
  - For example, a MenuItem for centering a page at normal size could be constructed from a CenterDocumentCommand object and a NormalSizeCommand object.
- To facilitate this , we can define a MacroCommand class to allow a MenuItem to execute an open-ended number of commands.
  - MacroCommand is a concrete Command subclass that simply executes a sequence of Commands.
  - MacroCommand has no explicit receiver, because the commands it sequences define their own receiver

Motivation(7)

- The Command pattern decouples the object that invokes the operation from the one having the knowledge to perform it – producing significant flexibility:
  - An application can provide both a menu and a push button interface to a feature just by making the menu and the push button share an instance of the same concrete Command subclass.
  - We can replace commands dynamically, which would be useful for implementing context-sensitive menus.
  - We can also support command scripting by composing commands into larger ones.

Applicability(1)

- To parameterize objects by an action to perform:
  - Parameterization is expressed in a procedural language with a callback function, that is, a function that's registered somewhere to be called at a later point.
  - Commands are an object-oriented replacement for callbacks.
- Specify, queue, and execute requests at different times:
  - A Command object can have a lifetime independent of the original request.
  - It may be possible transfer a command object for the request to a different process and fulfill the request there.

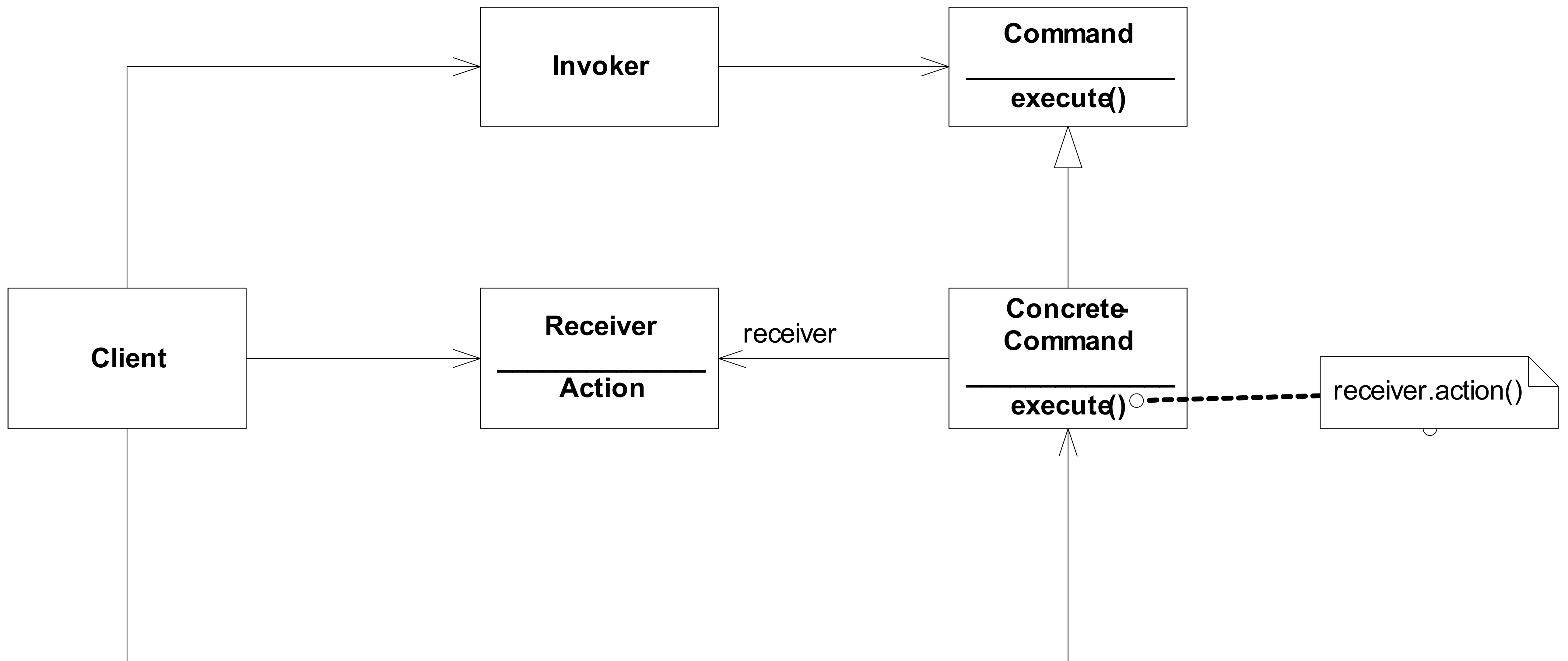
Applicability(2)

- Support undo/redo
  - The Command's Execute operation can store state for reversing its effects in the command itself.
  - The Command interface may have an added Unexecute operation that reverses the effects of a previous call to Execute.
  - Executed commands are stored in a history list.
  - Undo/redo is achieved by traversing this list backwards and forwards calling Unexecute and Execute, respectively.

Applicability(3)

- Support logging changes so that they can be reapplied in case of a system crash.
  - By augmenting the Command interface with load and store operations, a persistent log of changes can be maintained
  - Recovering from a crash involves reloading logged commands from disk and reexecuting them with the Execute operation.
- Structure a system around high-level operations built on primitives operations.
  - Such a structure is common in information systems that support transactions.
  - A transaction encapsulates a set of changes to data. The Command pattern offers a way to model transactions.
  - Commands have a common interface, letting you invoke all transactions the same way. The pattern also makes it easy to extend the system with new transactions

Structure

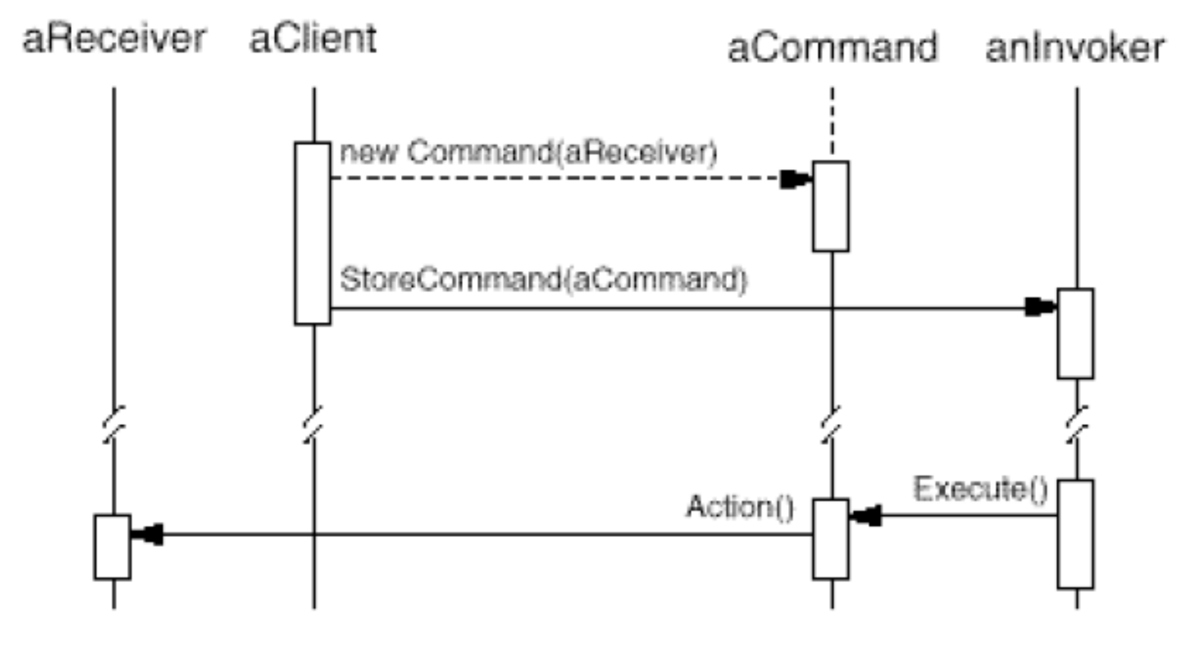


Participants

- Command
    - declares an interface for executing an operation.
- ConcreteCommand (PasteCommand, OpenCommand)
    - defines a binding between a Receiver object and an action.
    - implements Execute by invoking the corresponding operation(s) on Receiver.
- Client (Application)
    - creates a ConcreteCommand object and sets its receiver.
- Invoker (MenuItem)
    - asks the command to carry out the request.
- Receiver (Document, Application)
    - knows how to perform the operations associated with carrying out a request. Any class may serve as a Receiver.

Collaborations

- The client creates a ConcreteCommand object and specifies its receiver.
- An Invoker object stores the ConcreteCommand object.
- The invoker issues a request by calling Execute on the command. When commands are undoable, ConcreteCommand stores state for undoing the command prior to invoking Execute.
- The ConcreteCommand object invokes operations on its receiver to carry out the request.



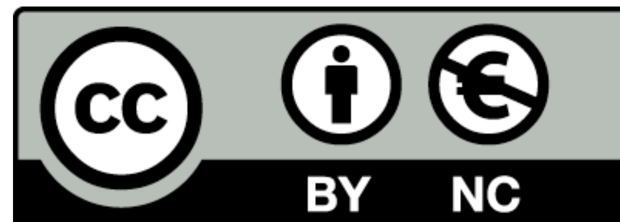
Consequences

- Command decouples the object that invokes the operation from the one that knows how to perform it.

- Commands are first-class objects. They can be manipulated and extended like any other object.

- You can assemble commands into a composite command. An example is the MacroCommand class described earlier. In general, composite commands are an instance of the Composite pattern.

- It's easy to add new Commands, because you don't have to change existing classes.



Except where otherwise noted, this content is licensed under a Creative Commons Attribution-NonCommercial 3.0 License.

For more information, please see http://creativecommons.org/licenses/by-nc/3.0/

Waterford Institute *of* Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE