Software Engineering

Lesson Design Pattern 10
Command, Memento
v1.0

Uwe Gühl

Fall 2007/ 2008

Contents

- Command
- Memento

Used sources:

- [GHJ04] Gamma, Helm, Johnson, Vlissides: Design Pattern, Addison Wesley, 2004
- [Hus08] Vince Huston: Design Pattern, www.vincehuston.org/dp/, 2008

Command

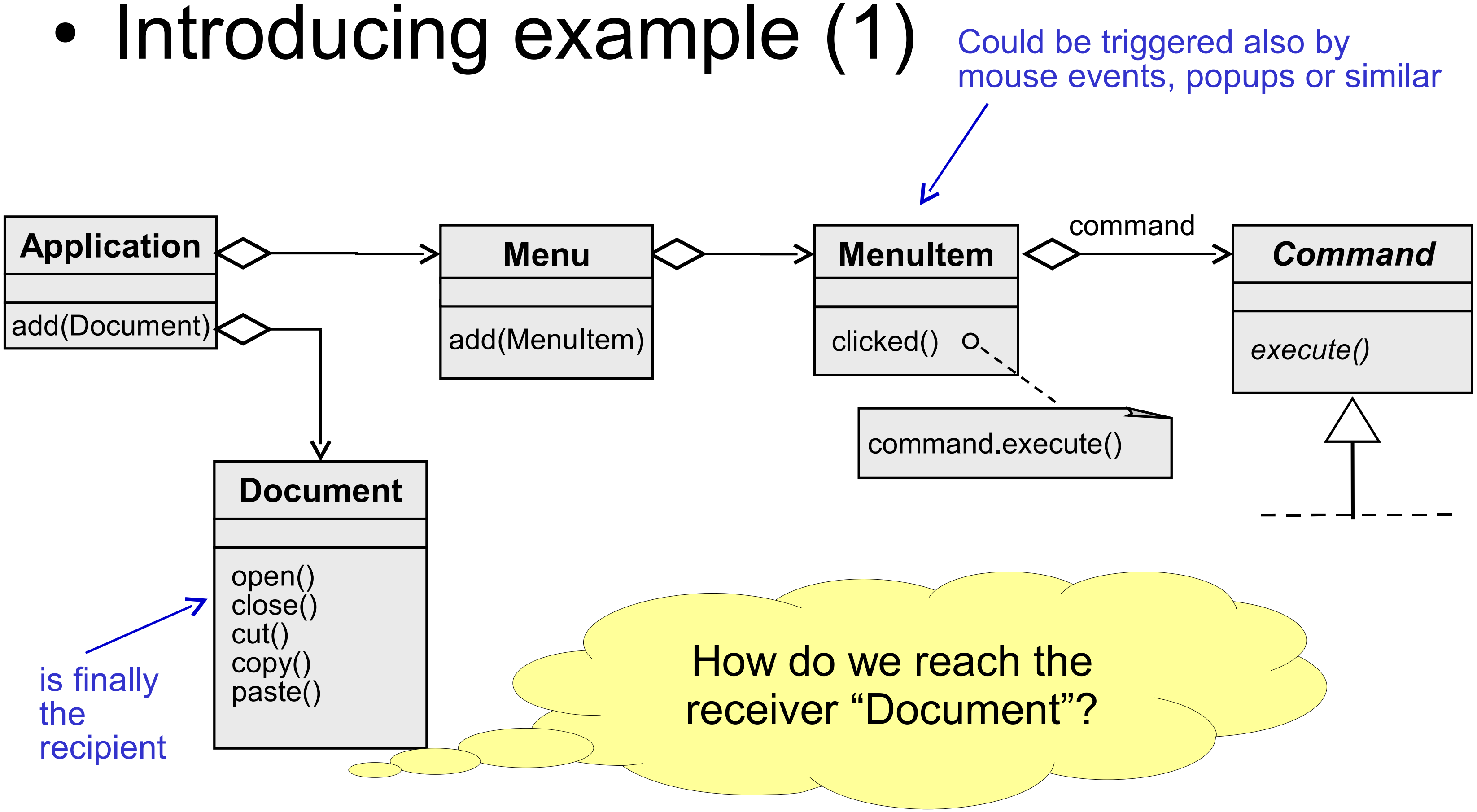
- Intent:
  - Encapsulates a request as an object
  - Allows the parametrization of a client with different
    - requests
    - queues or
    - log requests
  - support undoable operations
  - ... also known as Action, Transaction
  - ... is a Behavioral Pattern

Command

- Motivation
  - Problem: Treating of requests of an object without knowing something about the kind of request or the target object of the request
  - Example: GUI builder have buttons and menus, but the kind of requests could not be implemented explicitly there
  - Goal: Generic implementation of buttons or menus in a graphical user interface, so that no dependencies exist to actions of an application
  - Idea: Treating of a request as an object: An abstract ***Command*** class declaring execute operations – in the simplest case offering an abstract *`execute()`*

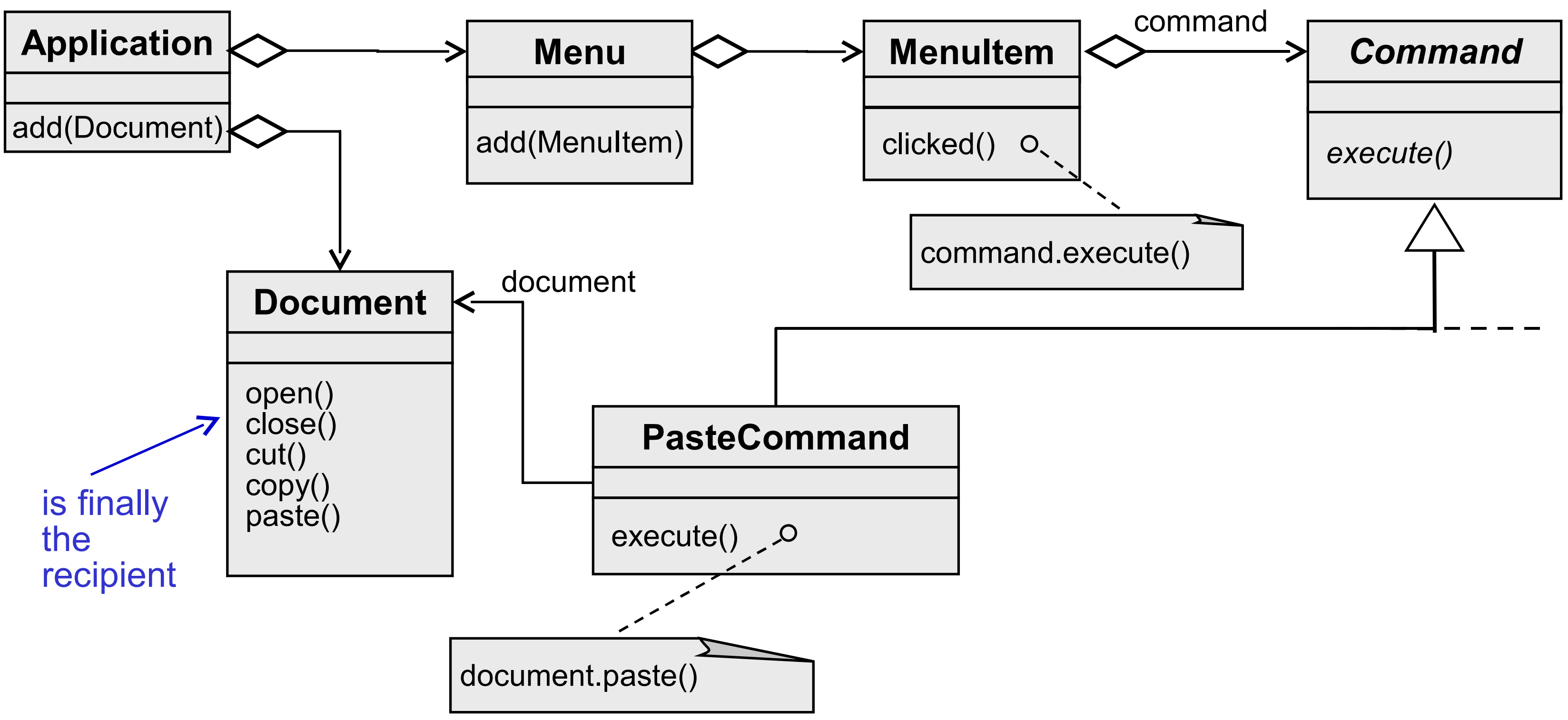
Command

- Introducing example (1)



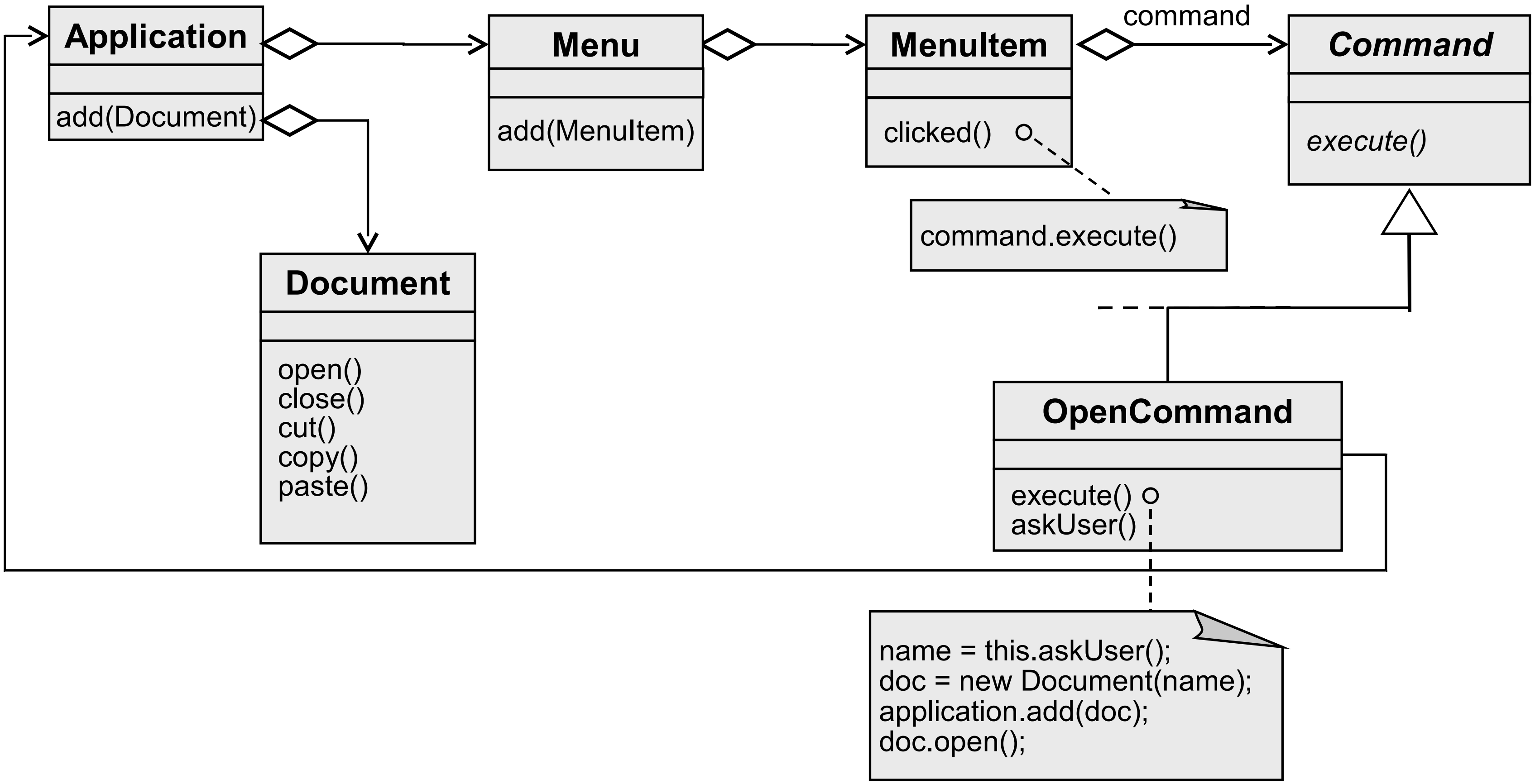
Command

- Introducing example (2)



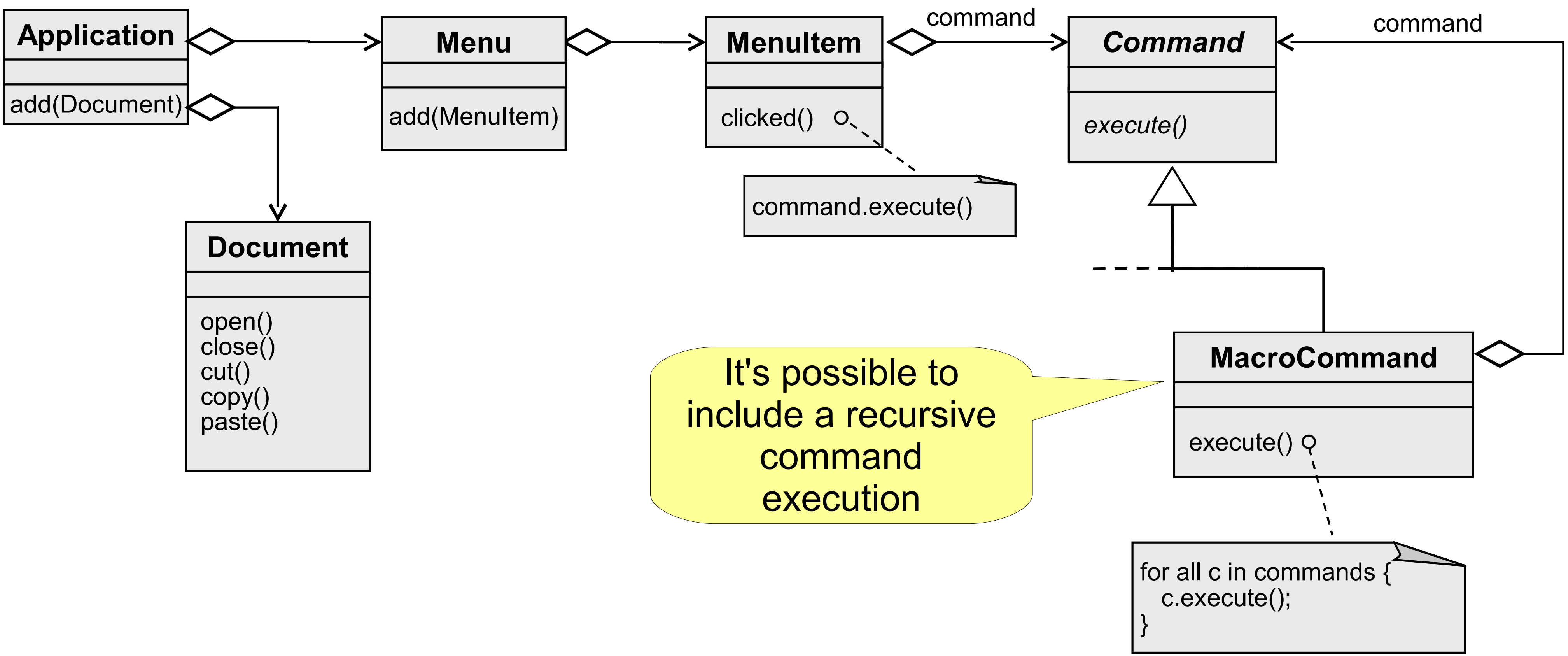
Command

- Introducing example (3)



Command

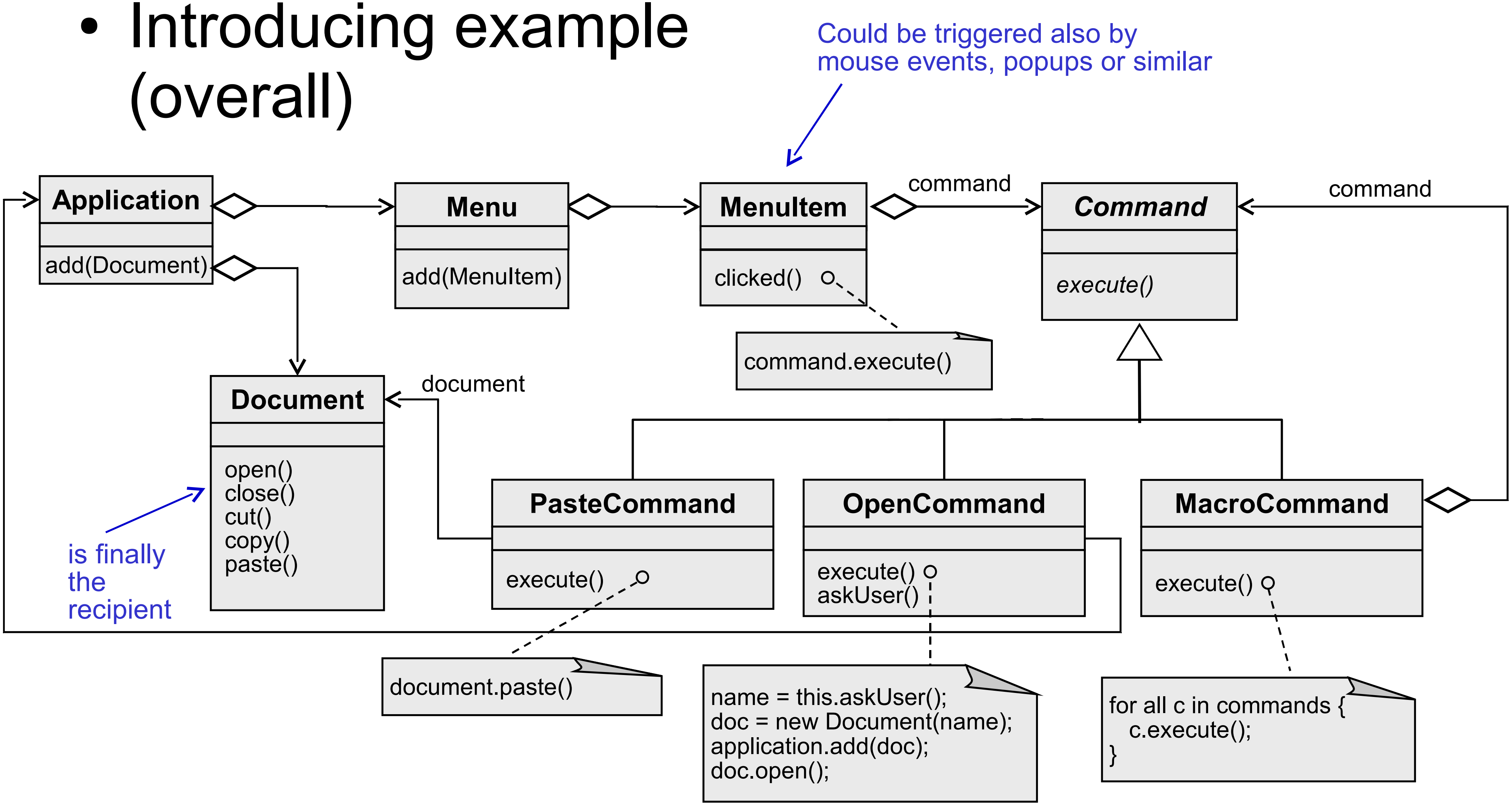
- Introducing example (4)



It's possible to include a recursive command execution

Command

- Introducing example (overall)



Command

- Motivation
  Solution:
  - Implementation of the execution of a command as independent object, that could be saved and given to different other objects
  - Important is the definition of an abstract interface for the call of an operation
  - A command object knows the receiver object and the action to be executed

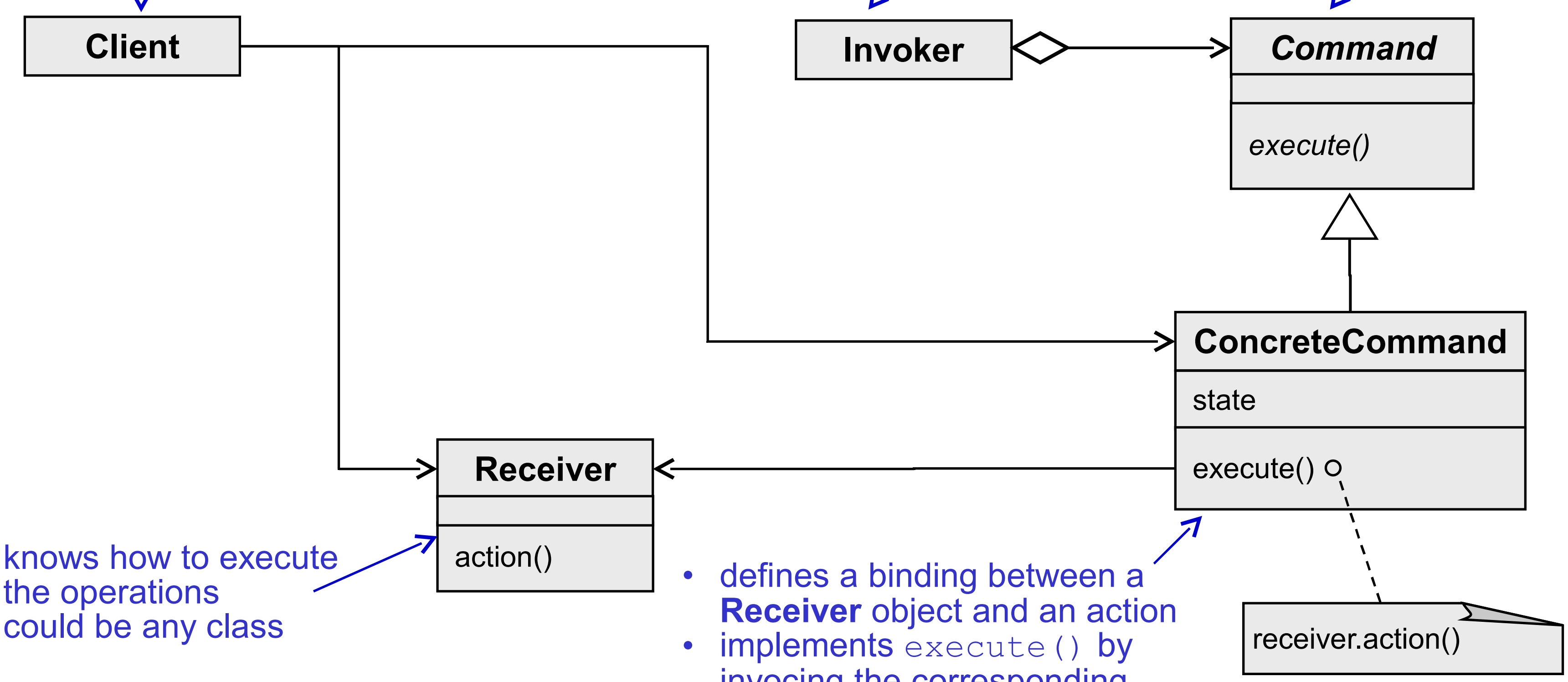
Command

- Structure

- creates a **ConcreteCommand** and sets its **Receiver**

- holds a ***Command***
- asks at some time ***Command*** to execute a request

- declares the interface for executing an operation



- knows how to execute the operations
- could be any class

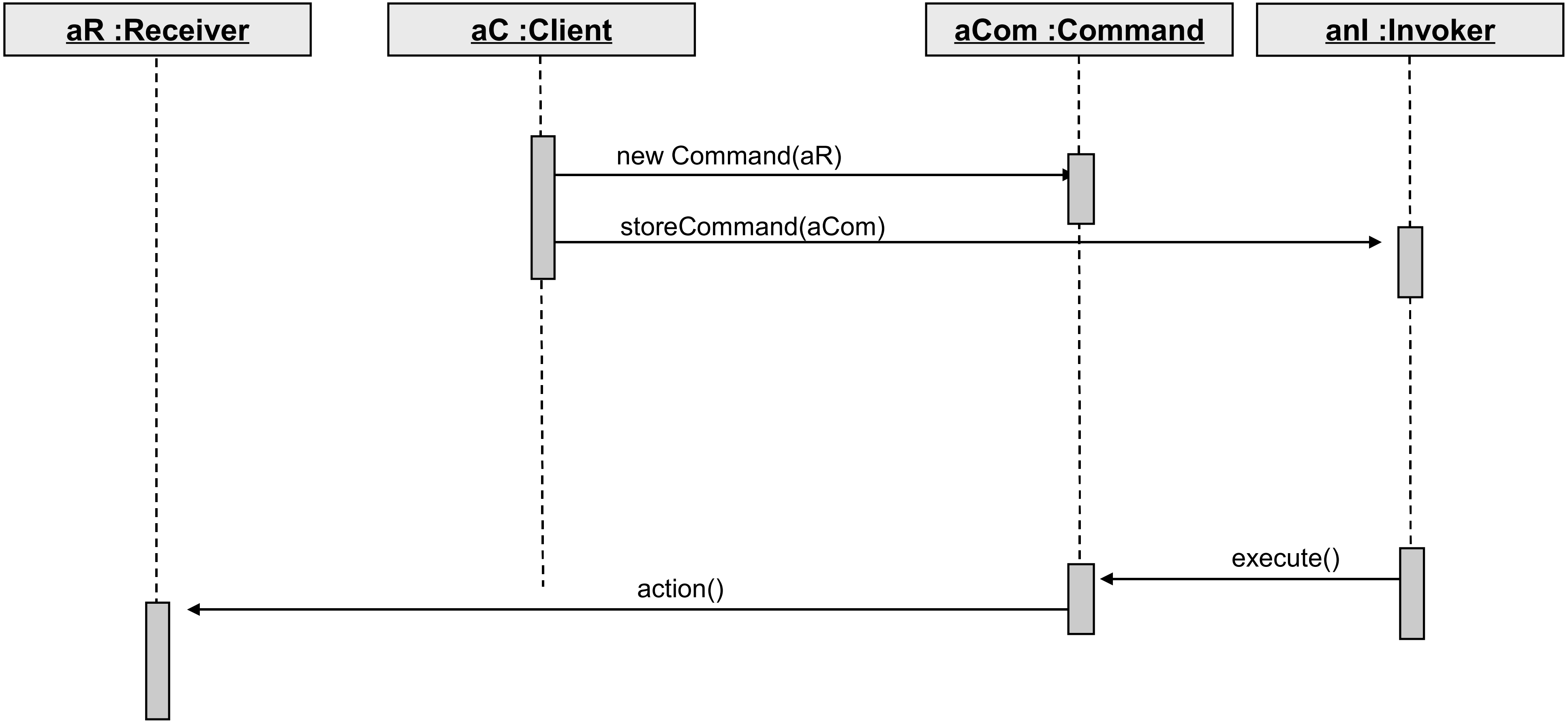
- defines a binding between a **Receiver** object and an action
- implements `execute()` by invocing the corresponding operations on **Receiver**

Command

- Collaboration
  - The **Client** creates a **ConcreteCommand** object and specifies the **Receiver**
  - An **Invoker** object stores the **ConcreteCommand** object
  - The **Invoker** issues a request by calling `execute()` on the ***Command*** object
  - The **ConcreteCommand** object invokes operations on its **Receiver**

Command

- Collaboration



Command

- Applicability
  Use the Command Pattern if you like to do
  - Parameterization of objects by the executing event (compare MenuItem objects in the example)
  - Execution and specification of requests at different times
  - Supporting of Undo (with history list)
  - Logging of commands (e. g. for recovery after system crash)
  - Structuring of a system with complex operations, built out of primitive ones (transactions)

Command

- Consequences

+ ***Command*** decouples the **Invoker** from the operation, which is called in the **Receiver**

+ ***Command***s as normal objects could be manipulated and extended

Command

- Consequences

+ ***Command*** could be combined as a Composites (as MacroCommand, executing a sequence of commands)

+ It's easy to add new **ConcreteCommands**, as given classes don't have to be changed

Command

- Implementation
  - Separating Command and Receiver
    - Passing all information to the receiver or implementing everything itself?
    - How to find a receiver if necessary?
      Enough knowledge necessary to find receiver dynamically

Command

- Implementation
  - Supporting Undo and Redo
    - To support Undo and Redo, a command must memorize the corresponding status
    - Attention in using semantic „Undo/Redo“-techniques – repeated often could lead to inconsistencies
    - Commands could be copied into a history list for any number of Undo steps

Command

- Known Uses (see [GHJ+95])
  - VisualSmalltalk (MenuItems)
  - WindowBuilder (Undo-List)
  - MacApp
  - ET++
  - InterViews

Command

- Related Patterns [Hus08] [p. 349, GHJ+95]
  - Chain of Responsibility, Command, Mediator, and Observer, address how you can decouple senders and receivers, but with different trade-offs
  - Command normally specifies a sender-receiver connection with a subclass
  - Chain of Responsibility can use Command to represent requests as objects

Command

- Related Patterns [Hus08], [p. 242, 346, GHJ+95]
  - Command and Memento act as magic tokens to be passed around and invoked at a later time.
    - In Command, the token represents a request;
    - in Memento, it represents the internal state of an object at a particular time.
    - Polymorphism is important to Command, but not to Memento because its interface is so narrow that a memento can only be passed as a value
  - Command can use Memento to maintain the state required for an undo operation

Command

- Related Patterns [Hus08], [p. 242, GHJ+95]
  - MacroCommands can be implemented with Composite
  - A Command that must be copied before being placed on a history list acts as a Prototype

Memento

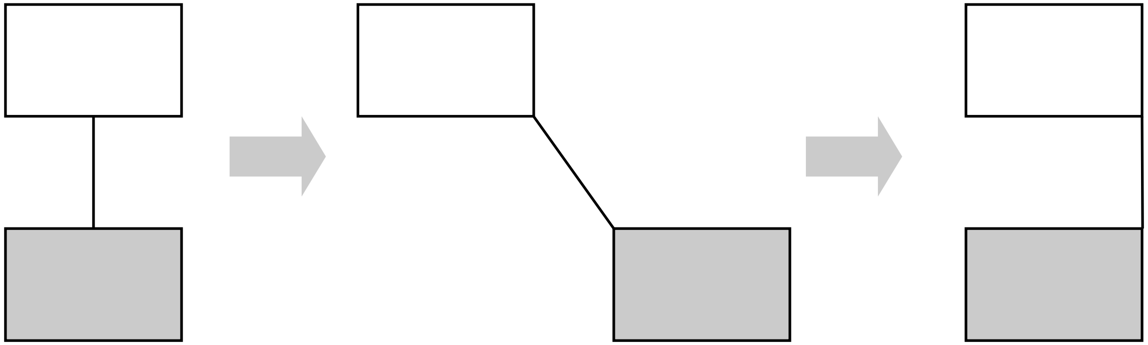
- Intent:
  - Extract the status of an object, without violating object encapsulation
  - ... is a Behavioral Pattern

Memento

- Motivation
  - The internal state of an object should be recorded to be recalled later, for example to be taken to an earlier status
  - Information about earlier states are important for Undo mechanisms
  - A direct access on the object internal states should be avoided – one possible good reason: to safeguard consistency

Memento

- Example
  How to realize Undo in a graphical editor?
  - Idea: Bundling of the functionality in an object „CreateSolver“
    - Management of the connections when created
    - Description with mathematical equations
  - Possible effects, if you manage connected objects and you store only distances:



Memento

- Example – Proposal
  - As side-effect of a „Move“ operation the editor asks the ConstraintSolver for Memento
  - The ConstraintSolver creates and hands over a SolverState Memento, storing the current internal state as a snapshot
  - If there is an „Undo“ operation the editor commits the ConstraintSolver the SolverState Memento
  - Based on the information in the SolverState Memento the ConstraintSolver changes its internal structures and restores the original state

Memento

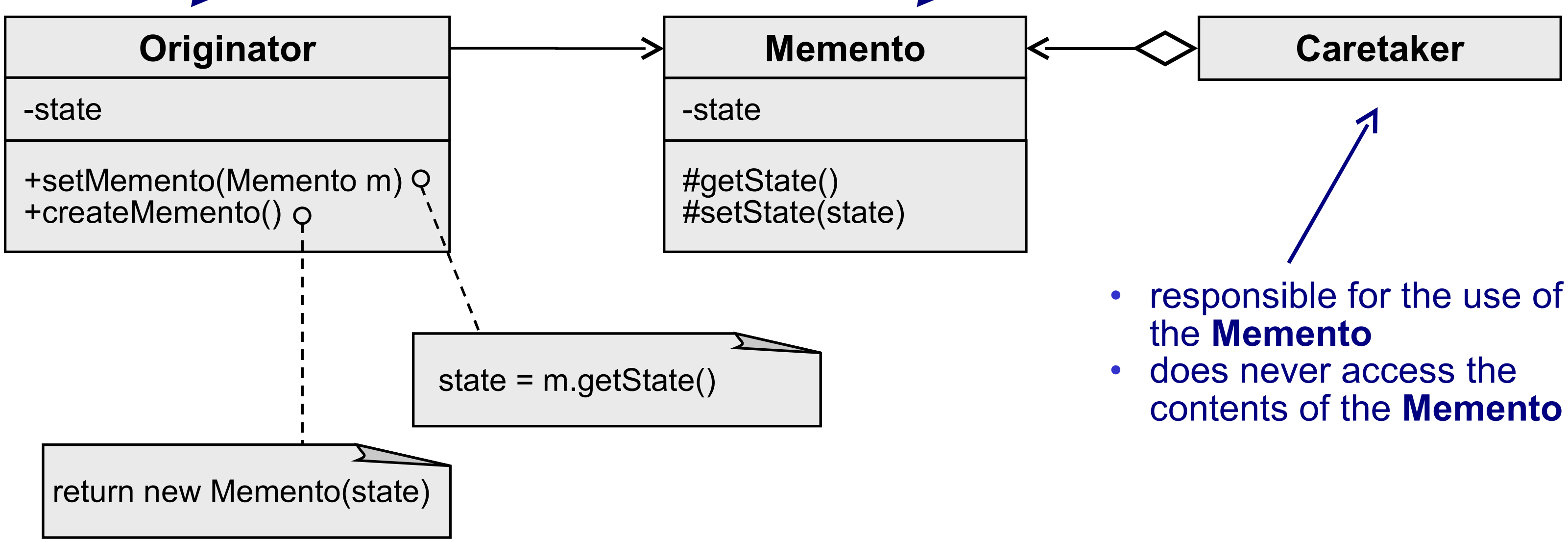
- Applicability
    - A current state of an object should be stored, so that it could be restored later
    - A direct interface to access the state would disclose implementation details and violate the encapsulation principle

Memento

- Structure

- creates a **Memento**, that stores the current internal status as a snapshot
- uses the **Memento** to restore its internal state on demand

- stores an internal state – uses therefore as much details as necessary to be able to restore it
- Has two interfaces
  - **Caretaker** has a narrow interface to pass the **Memento** to other objects
  - **Originator** sees a wide interface, to be able to restore itself to its previous state. Ideally only the **Originator** that produced the **Memento** has access to the internal state.



Originator

-state

+setMemento(Memento m)
+createMemento()

Memento

-state

#getState()
#setState(state)

Caretaker

state = m.getState()

return new Memento(state)

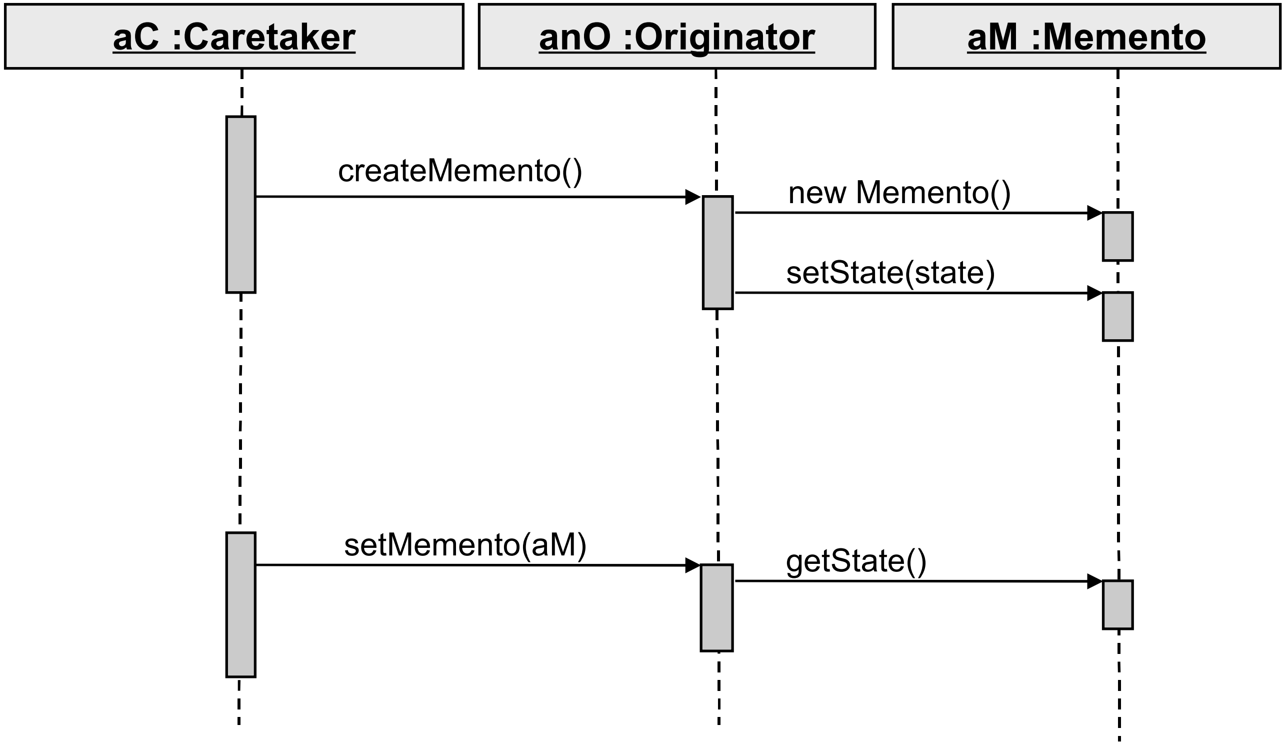
- responsible for the use of the **Memento**
- does never access the contents of the **Memento**

Memento

- Collaboration
  - A **Caretaker** requests a **Memento** from an **Originator**
  - The **Caretaker** holds the **Memento**
    - If the **Originato**r requests, the **Caretaker** passes the **Memento** back
    - If the **Originator** does not need to restore an earlier state, the **Caretaker** never pass the **Memento** back
  - **Memento**s are passive. Only the **Originator** that created a **Memento** will assign to retrieve its state

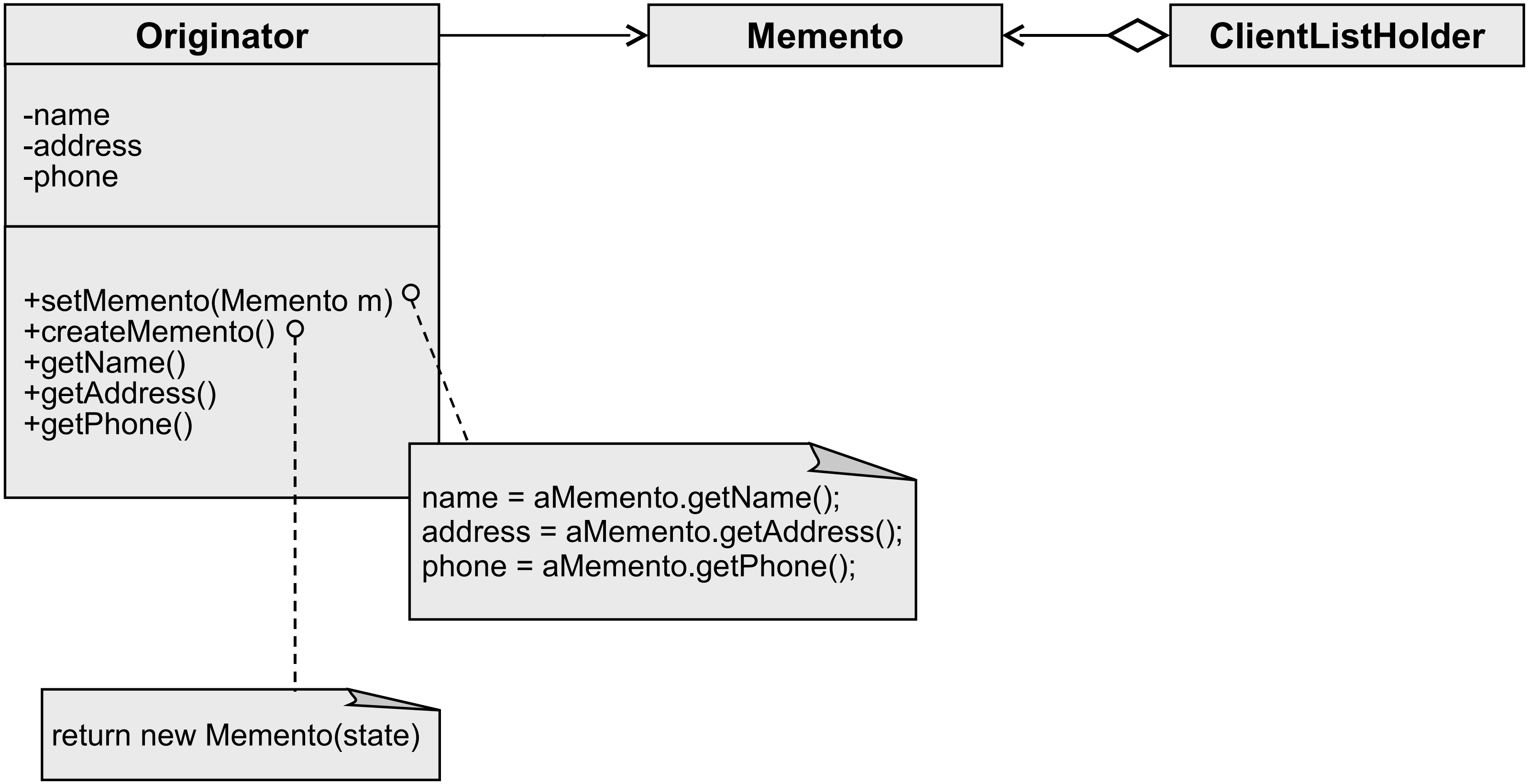
Memento

- Collaboration



Memento

- Example



Memento

- Consequences

  - + Encapsulation principle preserved: **Memento** encapsulates information, which is only known and managed by the **Originator**
  - + Simplification of the **Originator**
    - Outsourcing of the status management
  - – Use of the **Memento** might be expensive
    - **Memento**s might have too much overhead
    - If the encapsulation and restoring of a state of an Originator is cheap, the pattern might not be appropriate

Memento

- Consequences

○ Definition of narrow and wide interfaces

- It may be difficult is some programming languages to ensure that only the **Originator** may access the **Memento**'s state

– Hided costs in administration of a Memento

- **Caretaker** is responsible for deleting **Mementos** it cares for
- A **Caretaker** does not know how much state is in the **Memento** – could result in large storage costs

Memento

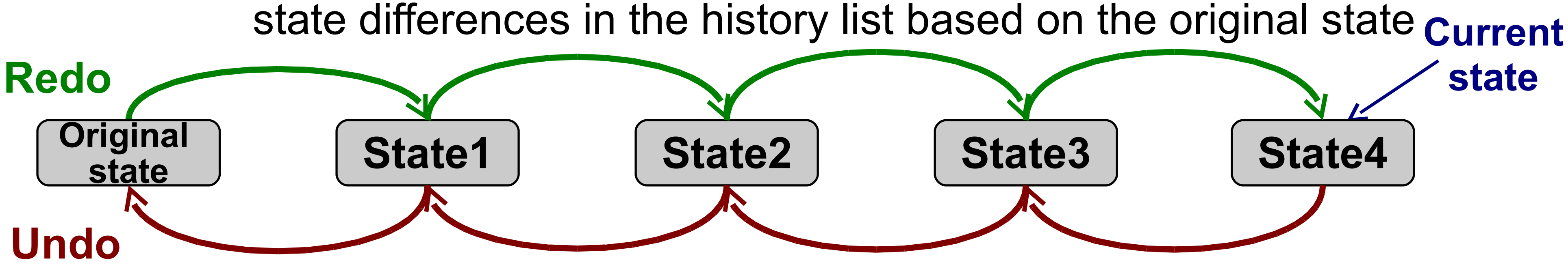
- Implementation
  - Language support
    - Mementos have two interfaces
      - a wide one for the **Originator**s
        - Setting and reading of the variables
      - a small one for other objects, especially the **Caretaker**
        - Creating and setting of the **Memento**
    - C++ supports this ideas with “`friend`“
      - So **Memento** is accessible for the **Originator**
      - For all other objects it acts like “`private`“
    - Smalltalk and Java don't offer such a construct

Memento

- Implementation
  - In simple cases the **Memento** object could be a copy of the **Originator** (this means an object of the same class)
  - It must be decided, how deep a state must be stored (or copied), so that it could be recovered completely

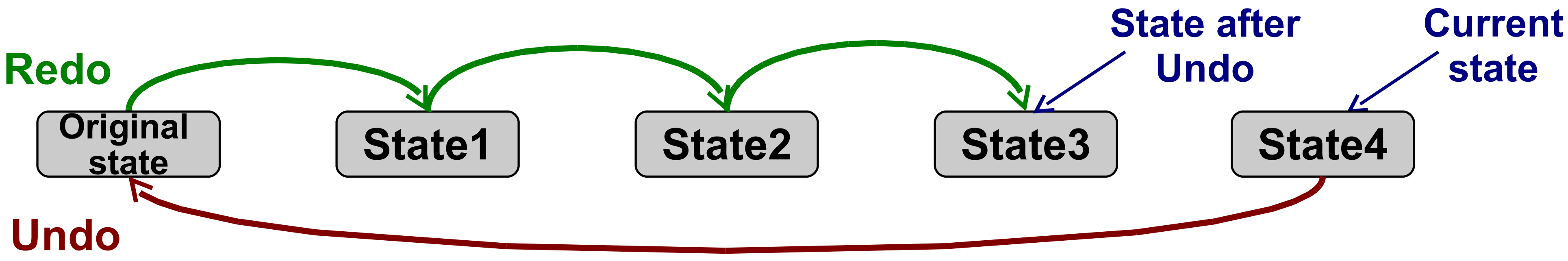
Memento

- Implementation
  - Storage of incremental changes
    - Depending on what really changed in a whole structure you don't save the complete state but the state differences – just the incremental change
    - Example:
      - Managing of undo: Instead of saving all the different states you only save the incremental changes as Mementos
      - Undo / Redo is then “simple” an execution of the corresponding state differences in the history list based on the original state



Memento

- Implementation
  - Example:
    - Doing an Undo in State4 to get back to State3



Memento

- Known Uses [GHJ+95]
  - Unidraw's support for connectivity through its CSolver class
  - Collections in the programming language „Dylan“ use an iteration interface reflecting the Memento Pattern
  - QOCA constraint-solving toolkit
  - Data base connectivities
    - The state of data gets stored to restore the original state, if a transaction fails

Memento

- Related Patterns
  - Command [p. 346, GHJ+95]
    Command objects are often Mementos to maintain state for undoable operations – They act as magic tokens to be passed around and invoked at a later time
  - Iterator [p. 271, GHJ+95]
    Mementos can be used for iteration
    - An Iterator can use a Memento to capture the state of an iteration
    - The Iterator stores the Memento internally