Command Pattern

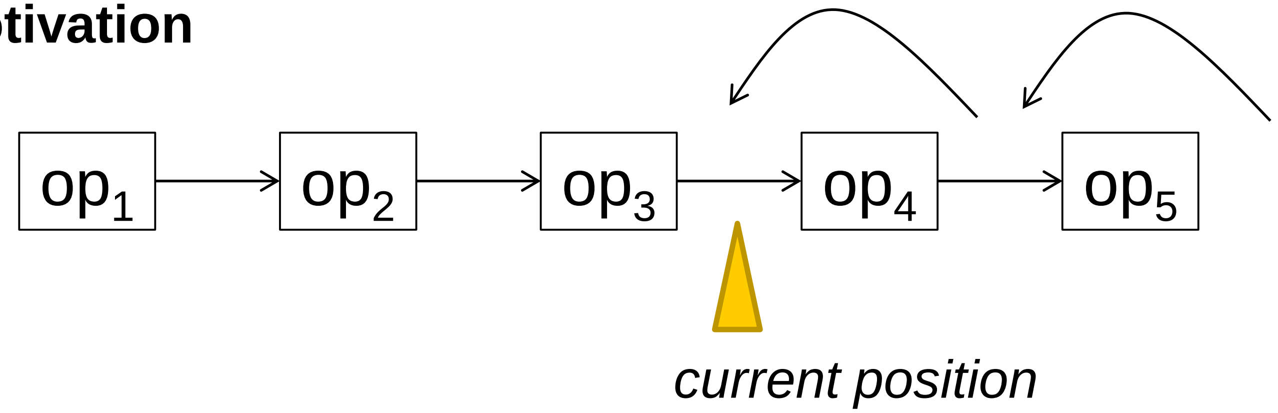
- **Goal**
  - Turn an action into an object (command), i.e. encapsulate all the information needed to call an action at a later time
    - Client instantiates the command and provides the information needed to call the method at a later time (receiver, method name, parameters)
    - Command may be stored and may be executed at a later time
  - Invoker does not need to know the receiver of the method nor the method parameters (they are stored in the command object)
  - Decouple the class that is invoking the action from the action itself

Command Pattern

- **Applications**
  - Transactional behavior
    - Similar to the undo, a database engine keeps a list of operations that have been (or will be) performed. If one of them fails, a rollback is executed
  - Wizards
    - Command object is created when the wizard is first displayed
    - Each wizard page stores its GUI changes in the command object
    - "Finish" executes the command
  - Actions in Swing: an Action is a command object
  - Thread Pool: actions are registered to be executed
  - Macro recording: user interactions can be recorded
  - Multi-level undo
    - User actions are stored in an undo history stack

Undo / Redo

- **Motivation**



  - Remarks:
    - Association of undo/redo list with model or with view?
    - What happens if a new operation is executed at the current position?

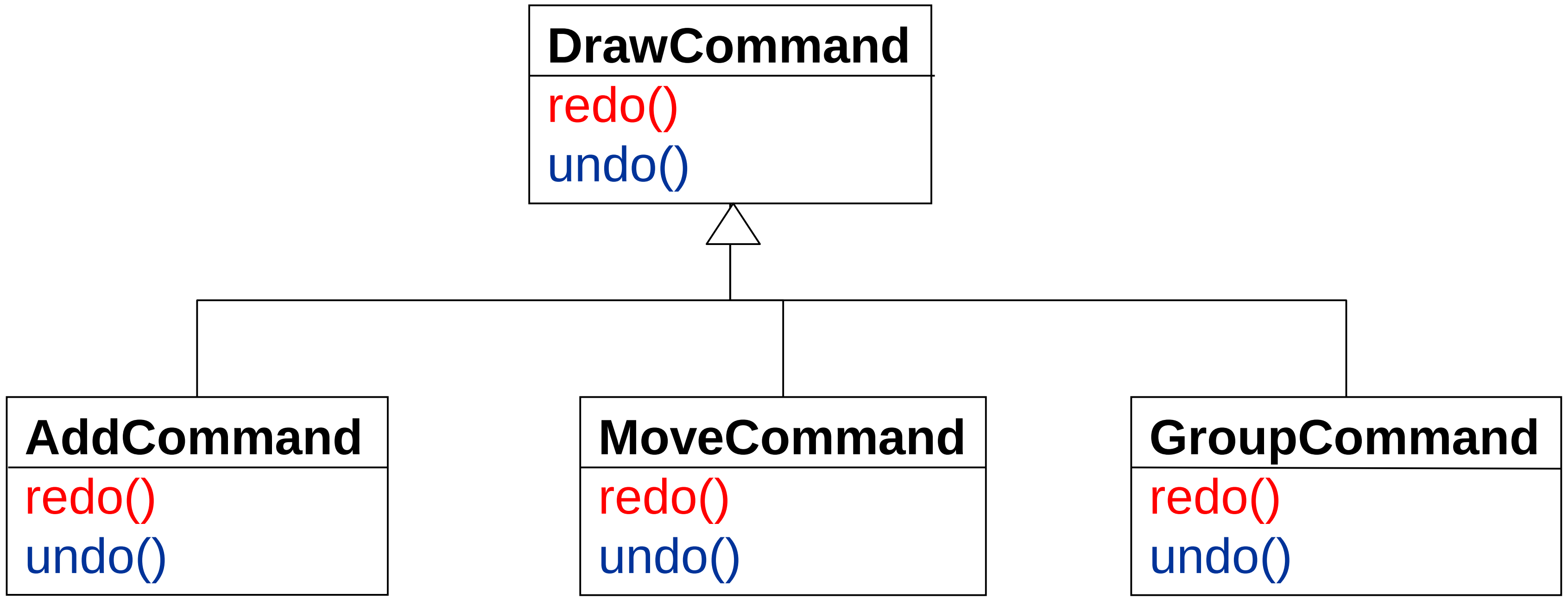
Undo / Redo

- **Implementation**

$$\begin{tikzcd}
\boxed{S_0} \arrow[r, "\Delta_0"] & \boxed{S_1} \arrow[r, "\Delta_1"] & \boxed{S_2} \arrow[r, "\Delta_2"] & \boxed{S_3}
\end{tikzcd}$$

  - Copy and save the states
    - Figure list has to be (deep) cloned upon every operation
  - Copy and save the changes
    - The changes have to be represented as command objects

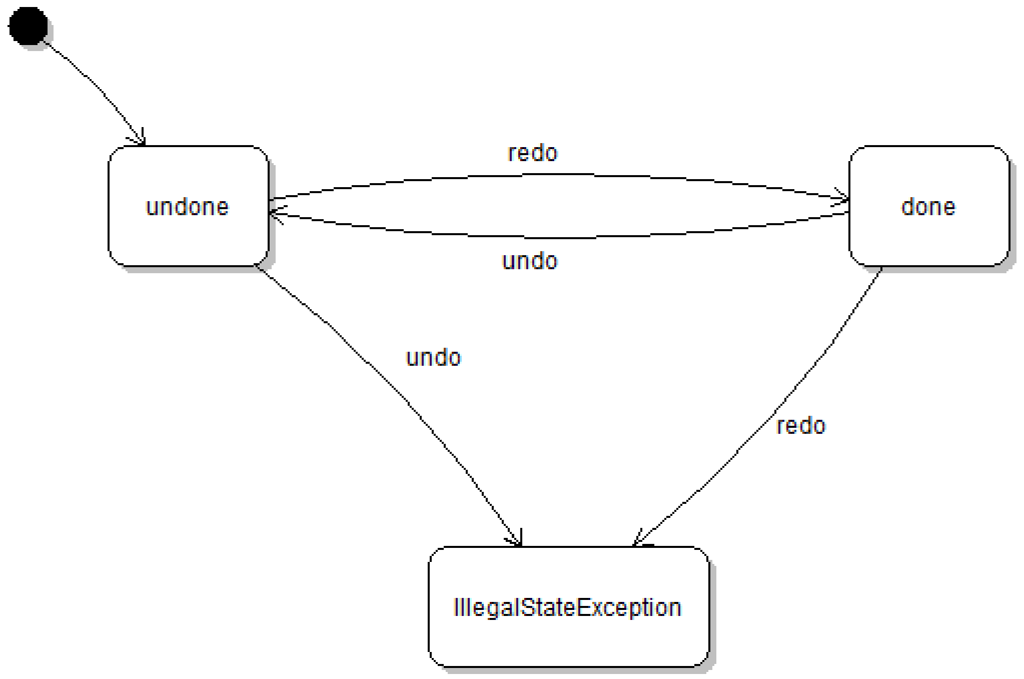
Undo / Redo


DrawCommand
redo()
undo()
AddCommand
redo()
undo()
MoveCommand
redo()
undo()
GroupCommand
redo()
undo()

Undo / Redo

- **State diagram for commands**

From the view of a single command



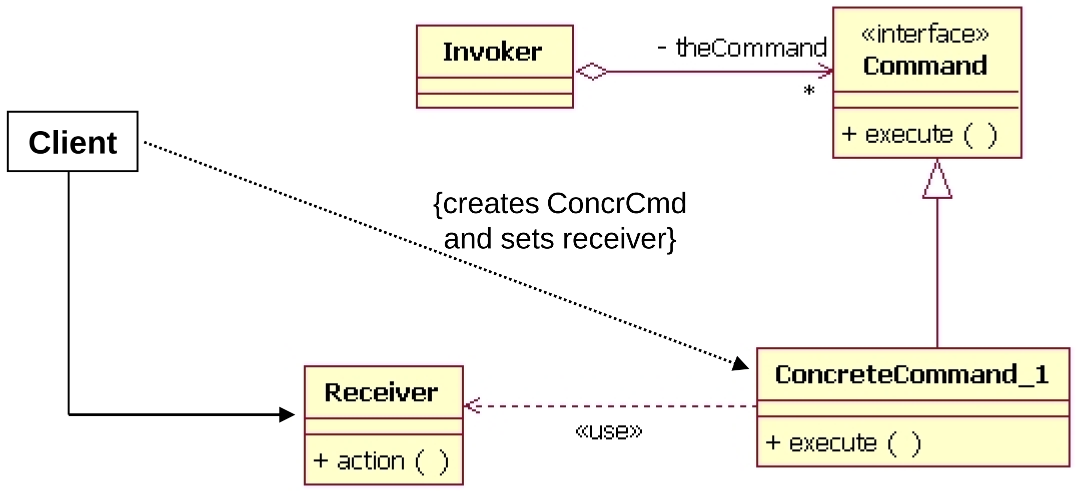
Undo / Redo: DrawCommandHandler

```
public interface DrawCommandHandler {
   void addCommand(DrawCommand cmd);     // clears redo list
                                         // adds cmd to undo commands
   void undo();
   void redo();

   boolean undoPossible();
   boolean redoPossible();

   void clearHistory();
}
```

- **Implementation:**
  - List of commands (ListIterator supports methods `next` and `previous`)
  - Two stacks

Command Pattern

- **Definition**
  - Encapsulate a request as an object
  - Allows to parameterize clients with different requests



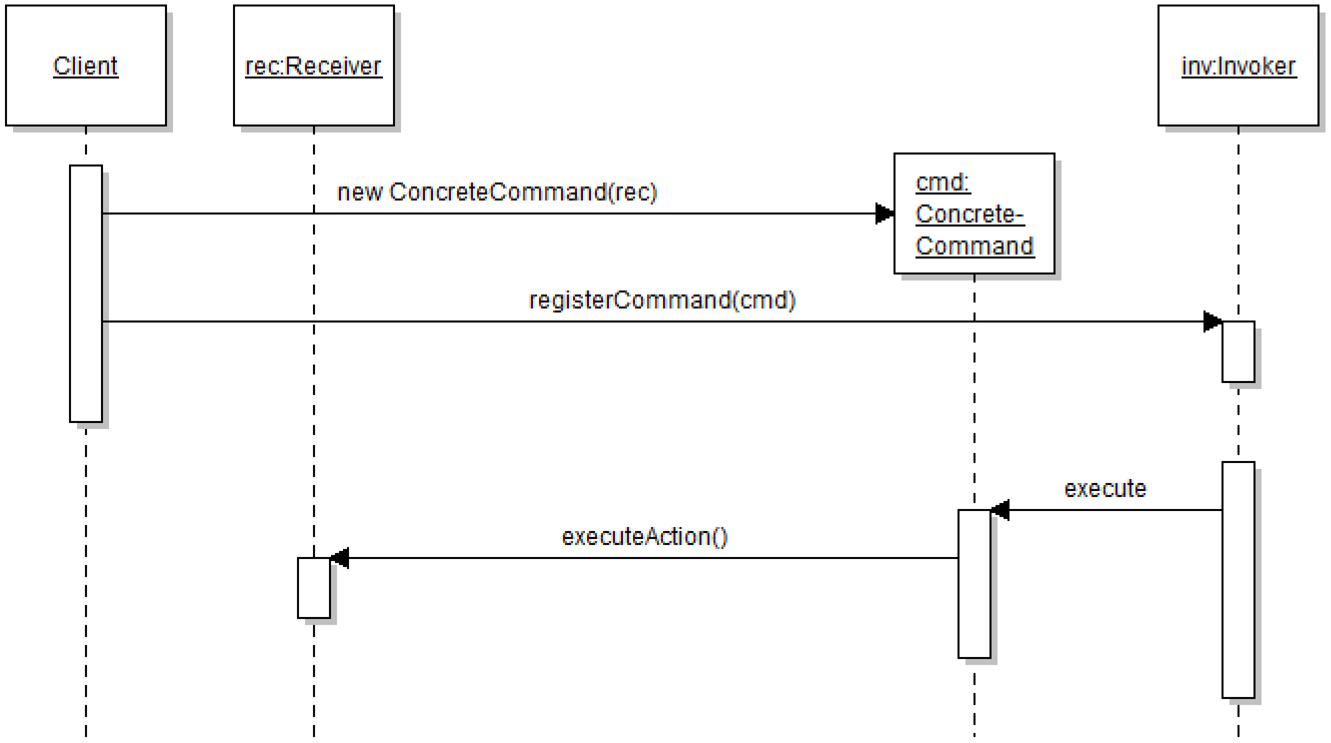
  - Example: ActionListener implementation

Command Pattern

- **Participants**
  - Command
    - Declares interface for executing commands
  - Concrete Command
    - Implements execute by invoking an operation on the receiver
  - Client
    - Creates a concrete command and sets the information needed to call the method at a later time (receiver and parameters)
  - Invoker
    - Decides when the method is called, i.e.
    - Asks the command to carry out the request
  - Receiver
    - Knows how to execute the operation, i.e.
    - Receiver executes the actual work to be done

Command Pattern

- **Sequence Diagram**



Undo / Redo: DrawCommandHandler

- **Problem:**
  - Movement of a figure leads to many operations
    - `move(dx, dy)` typically moves the figure only about 1-2 pixels
    - Guarantees visual feedback
  - All `MoveCommand` objects are stored in the undo-list
  - Undo of such a move operation requires many undo invocations

- **Solution:**

Fachhochschule
Nordwestschweiz

Command Pattern

- **Forces**
  - You have to provide a undo / redo management
  - You have to queue commands
    - Remember Observer Causality Problem
  - You need to maintain a persistent log of commands executed
  - You need to provide transaction support
  - You have to support timed operations