The Command Pattern

Other Considerations

Douglas C. Schmidt

Learning Objectives in This Lesson

- Recognize how the *Command* pattern can be applied to perform user-requested commands consistently & extensibly in the expression tree processing app.
- Understand the structure & functionality of the *Command* pattern.
- Know how to implement the *Command* pattern in C++.
- Be aware of other considerations when applying the *Command* pattern.

Douglas C. Schmidt

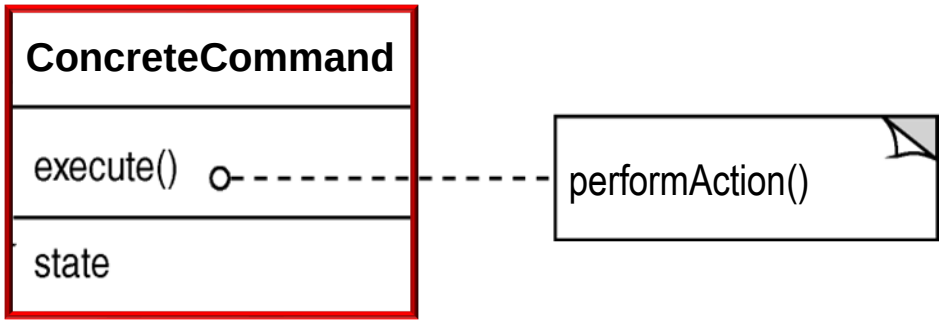
Other Considerations of the Command Pattern

Consequences

+ Abstracts the executor of a service
  - Makes programs more modular & flexible

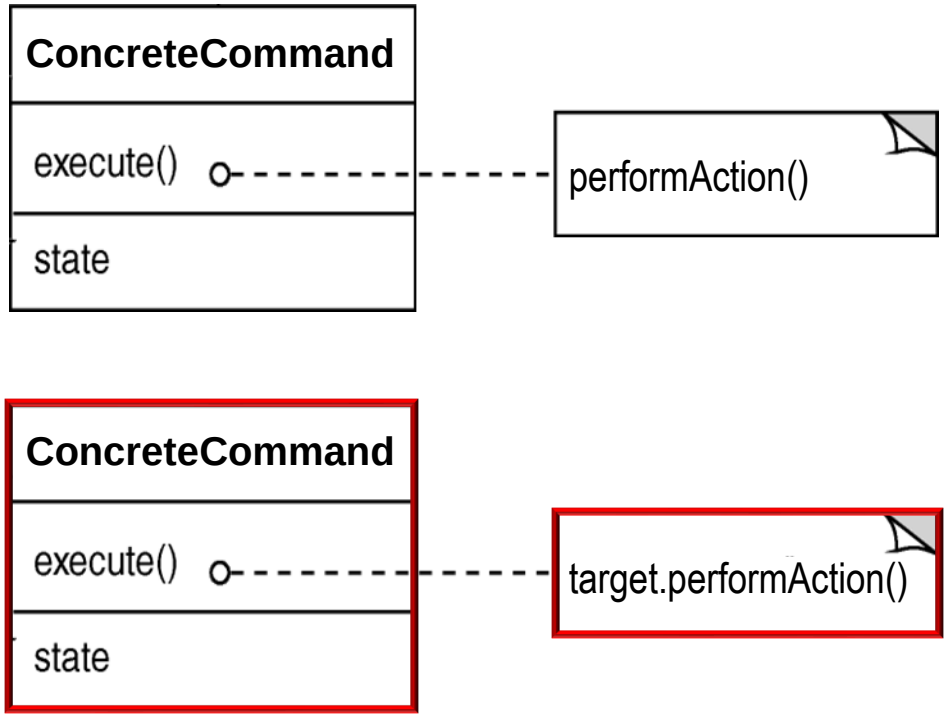
Consequences

+ Abstracts the executor of a service
  - Makes programs more modular & flexible, e.g.,
    - Can bundle state & behavior into an object



Consequences

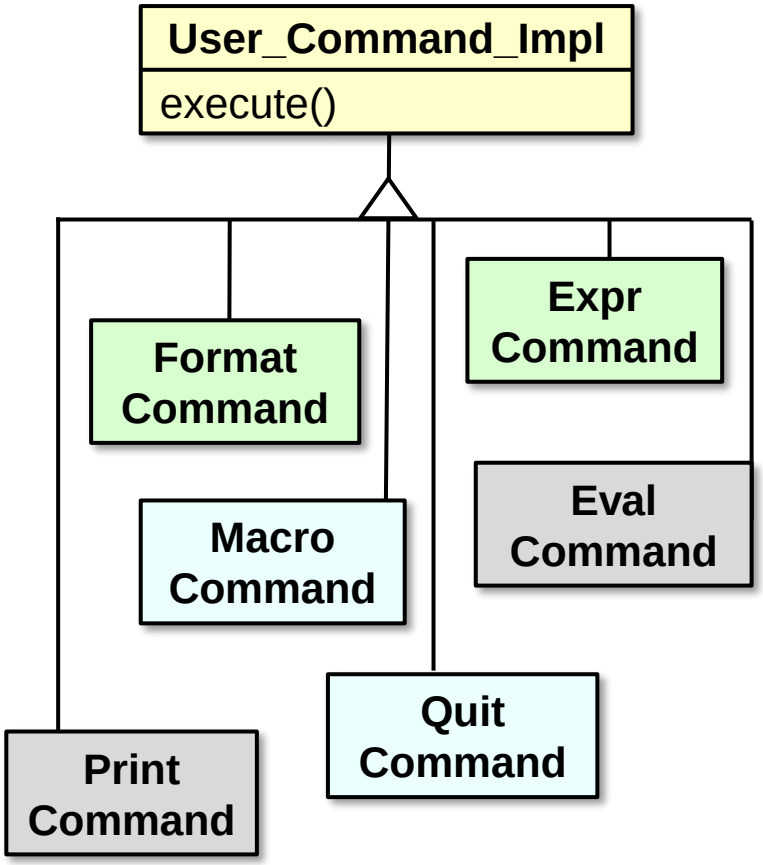
+ Abstracts the executor of a service
  - Makes programs more modular & flexible, e.g.,
    - Can bundle state & behavior into an object
    - Can forward behavior to other objects



See upcoming lesson on the *State* pattern for an example of forwarding.

Consequences

+ Abstracts the executor of a service
  - Makes programs more modular & flexible, e.g.,
    - Can bundle state & behavior into an object
    - Can forward behavior to other objects
    - Can extend behavior via derived classing



Consequences

+ Abstracts the executor of a service
  - Makes programs more modular & flexible, e.g.,
    - Can bundle state & behavior into an object
    - Can forward behavior to other objects
    - Can extend behavior via derived classing
    - Can pass a command object as a parameter

```
void handle_input() {
  ...
  User_Command command =
    make_command(input);


  execute_command(command);
```

The handle_input() method in Input_Handler plays the role of "invoker."

Consequences

+ Abstracts the executor of a service
  - Makes programs more modular & flexible, e.g.,
    - Can bundle state & behavior into an object
    - Can forward behavior to other objects
    - Can extend behavior via derived classing
    - Can pass a command object as a parameter

```
void handle_input() {
  ...
  User_Command command =
    make_command(input);

  execute_command(command);
```

Call a hook (factory) method to make a command based on user input

Consequences

+ Abstracts the executor of a service
  - Makes programs more modular & flexible, e.g.,
    - Can bundle state & behavior into an object
    - Can forward behavior to other objects
    - Can extend behavior via derived classing
    - Can pass a command object as a parameter

```
void handle_input() {
  ...
  User_Command command =
    make_command(input);
```

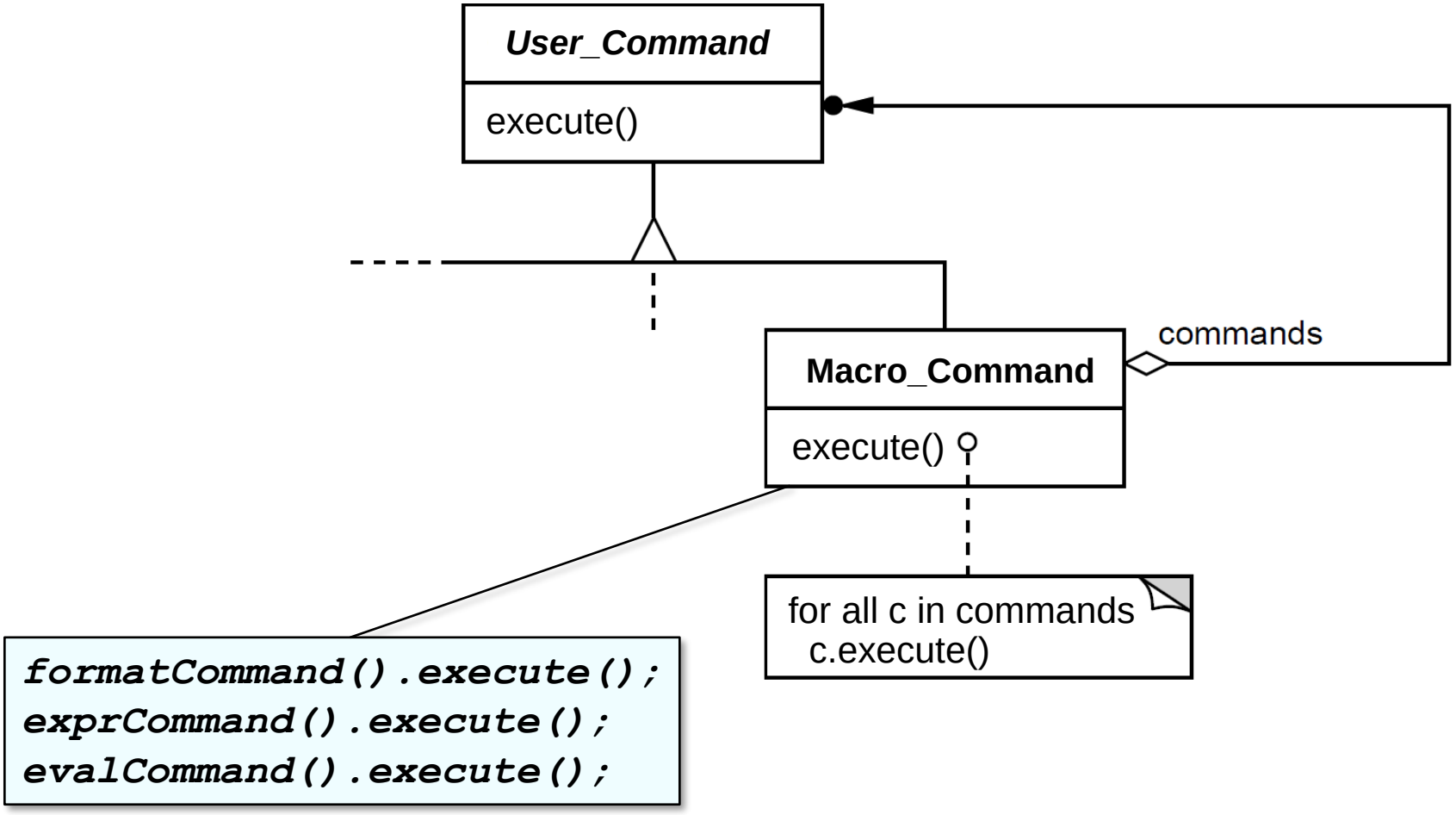
```
execute_command(command);
```

Call a hook method & pass a command to execute

See upcoming lesson on "*The Template Method Pattern*"

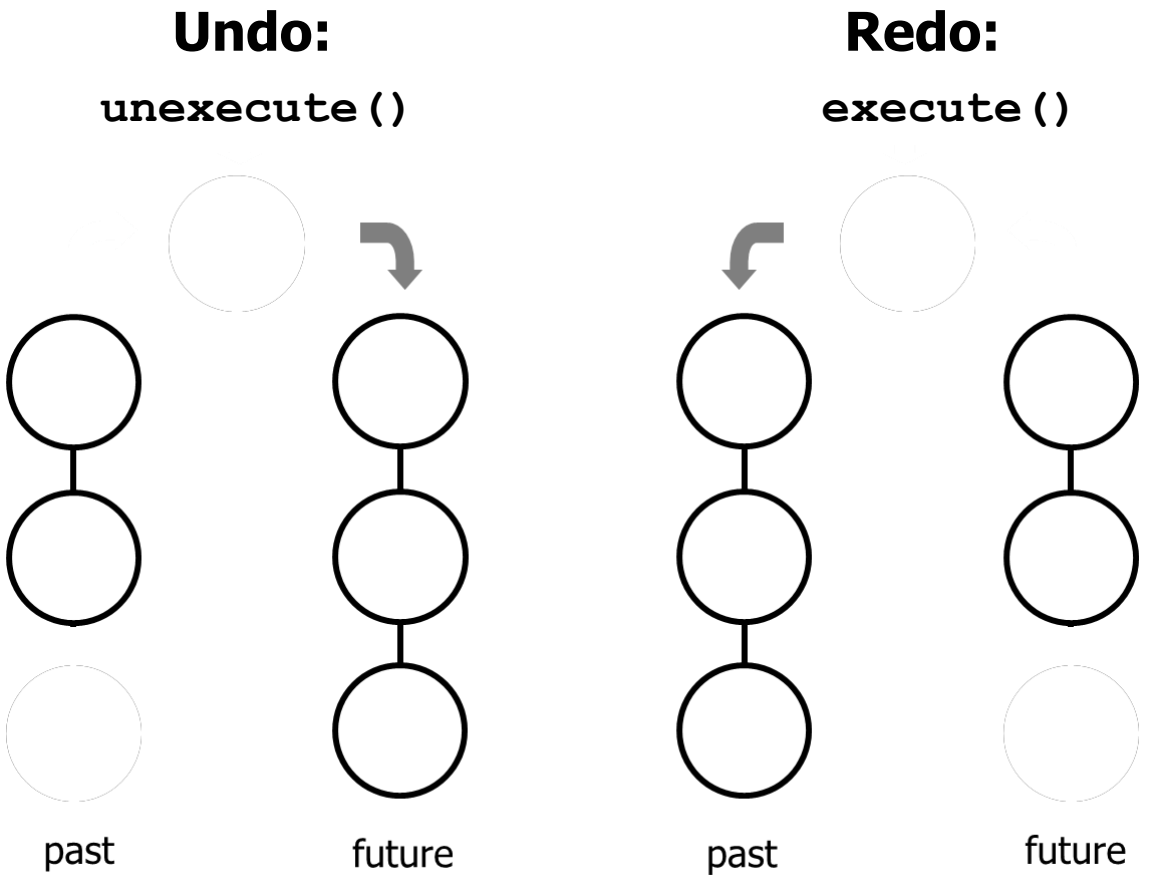
Consequences

+ Composition yields macro commands



Consequences

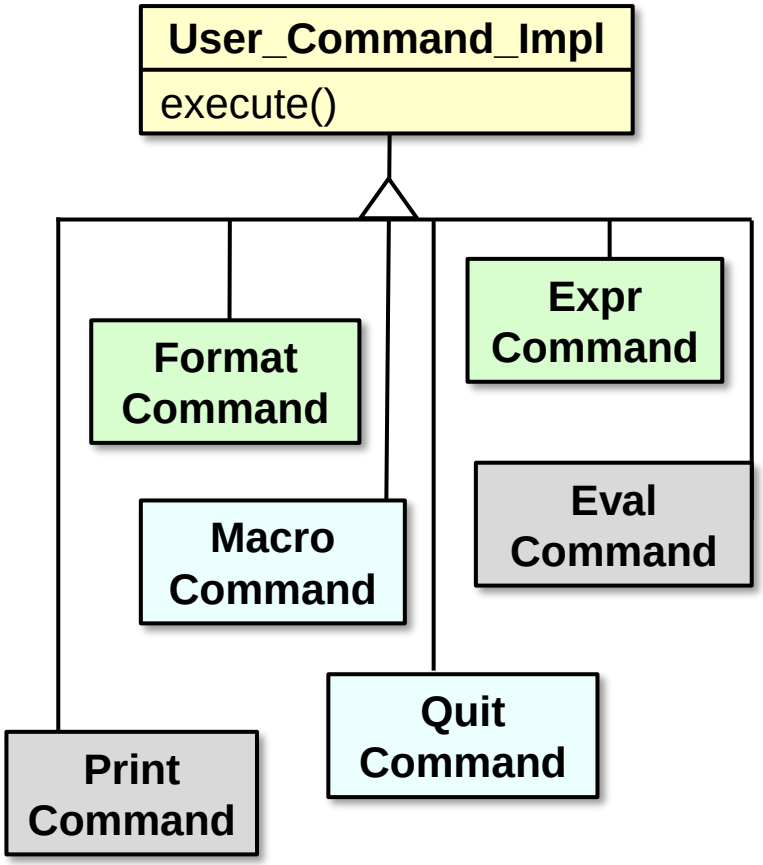
+ Supports arbitrary-level undo-redo



Case study doesn't use **`unexecute()`**, but it's a common *Command* feature.

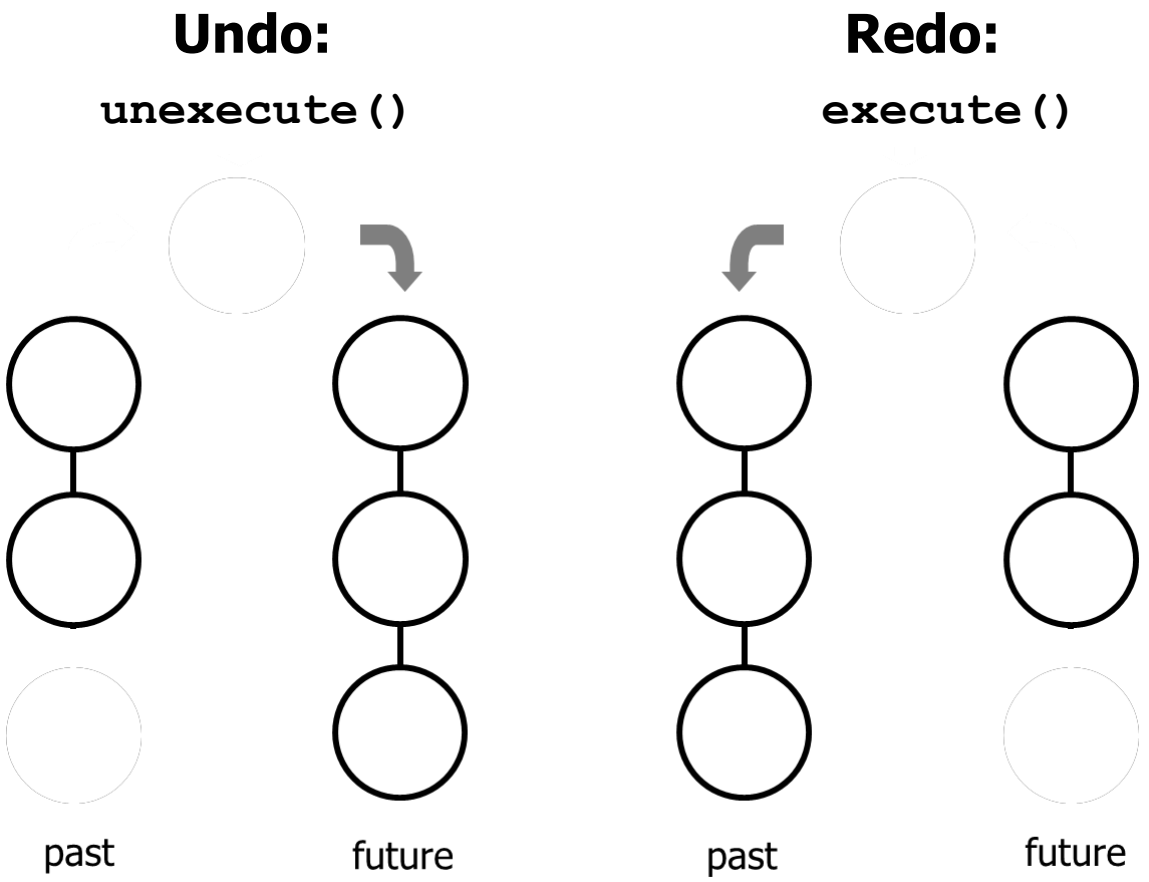
Consequences

– Might result in lots of trivial command derived classes



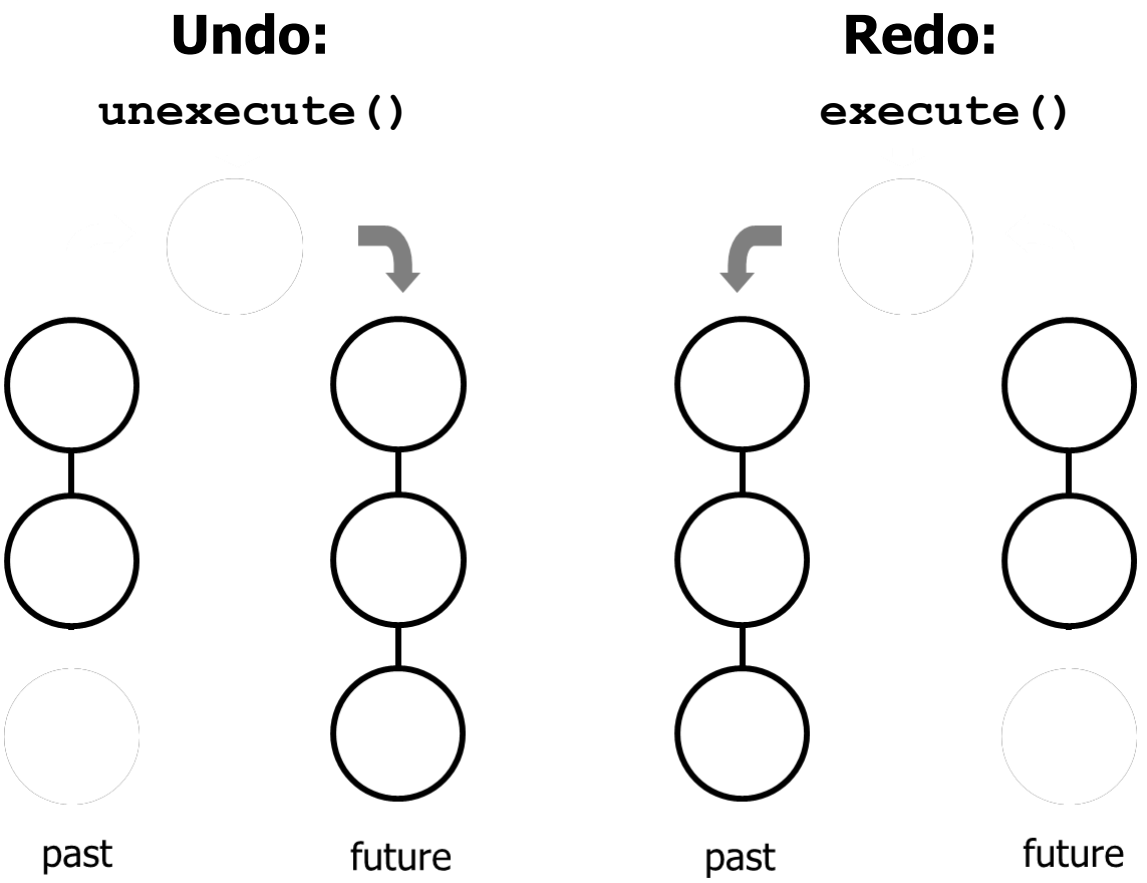
Consequences

– Excessive memory may be needed to support undo/redo operations



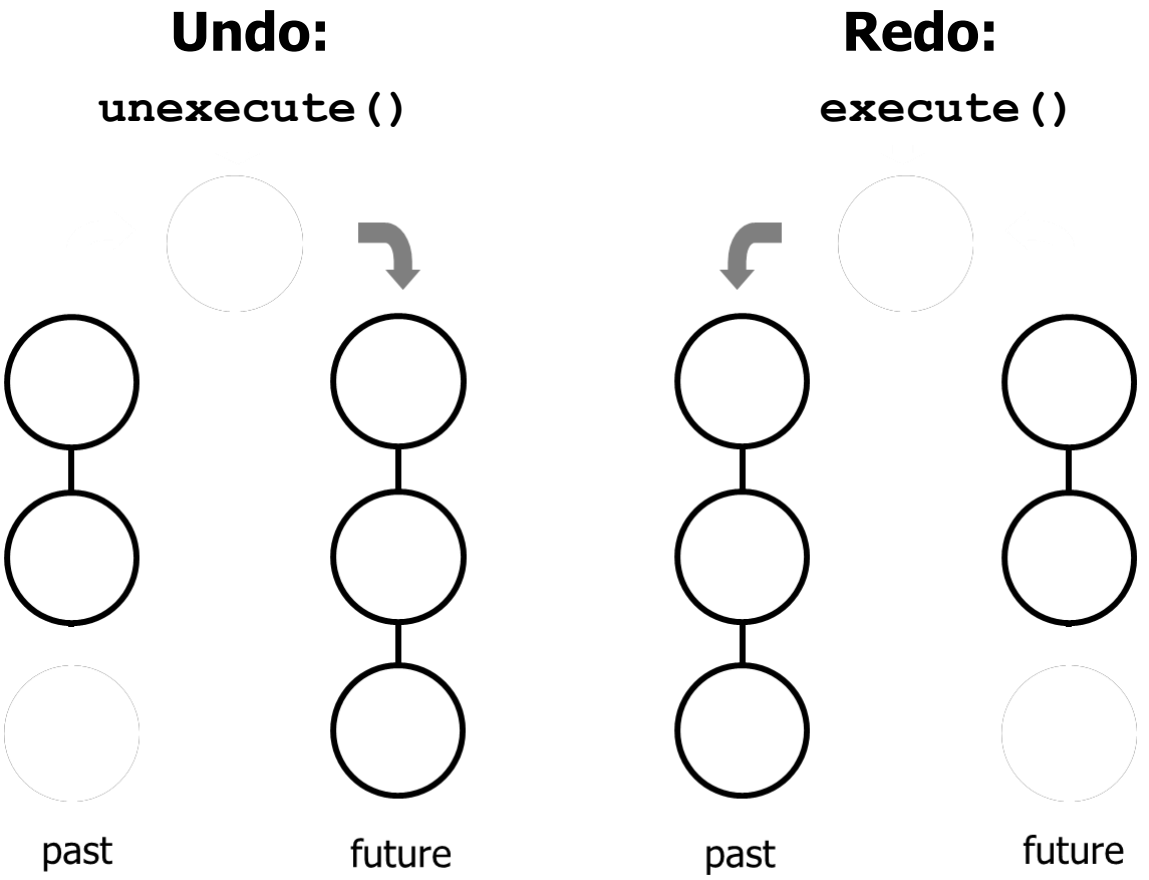
Implementation considerations

- Copying a command before putting it on a history list



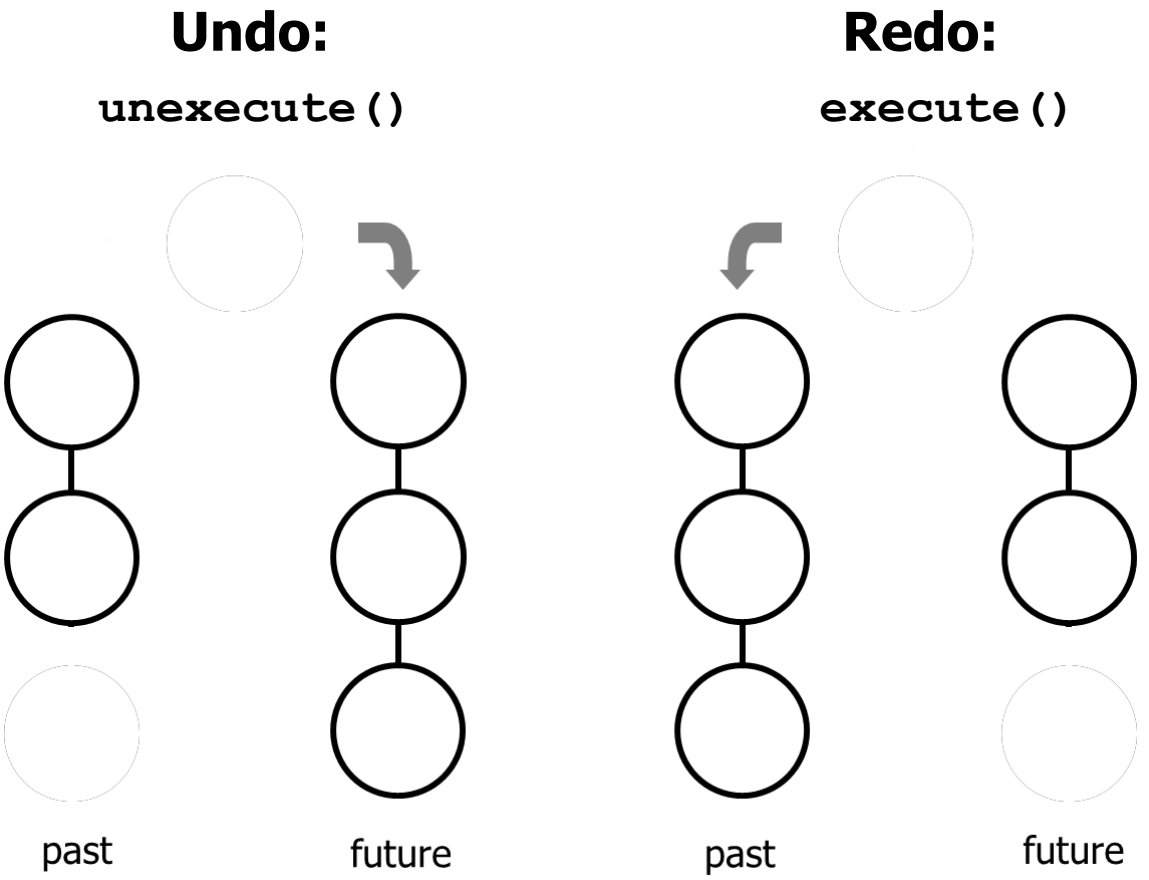
Implementation considerations

- Avoiding error accumulation during undo/redo



Implementation considerations

- Supporting transactions



Known uses

- InterViews Actions
- MacApp, Unidraw Commands
- JDK's UndoableEdit, AccessibleAction
- GNU Emacs
- Microsoft Office tools
- Java **`Runnable`** interface

java.lang

Interface Runnable

All Known Subinterfaces:

RunnableFuture<V>, RunnableScheduledFuture<V>

All Known Implementing Classes:

AsyncBoxView.ChildState, FutureTask, RenderableImageProducer, SwingWorker, Thread, TimerTask

---

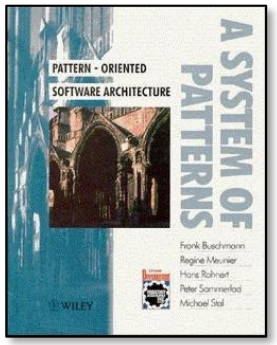
```
public interface Runnable
```

The `Runnable` interface should be implemented by any class whose instances are intended to be executed by a thread. The class must define a method of no arguments called `run`.

See docs.oracle.com/javase/8/docs/api/java/lang/Runnable.html

Known uses

- InterViews Actions
- MacApp, Unidraw Commands
- JDK's UndoableEdit, AccessibleAction
- GNU Emacs
- Microsoft Office tools
- Java **`Runnable`** interface
  - **`Runnable`** can also be used to implement the *Command Processor* pattern



java.lang

Interface Runnable

All Known Subinterfaces:

RunnableFuture<V>, RunnableScheduledFuture<V>

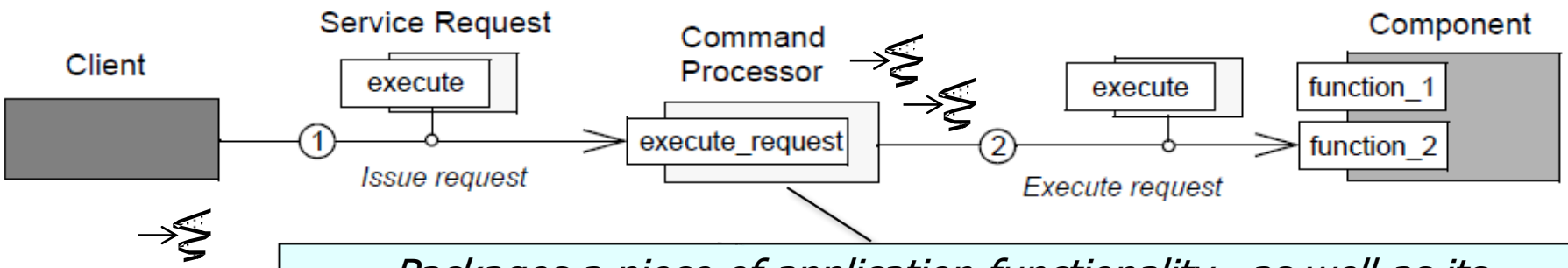
All Known Implementing Classes:

AsyncBoxView.ChildState, FutureTask, RenderableImageProducer, SwingWorker, Thread, TimerTask

---

`public interface` **`Runnable`**

The `Runnable` interface should be implemented by any class whose instances are intended to be executed by a thread. The class must define a method of no arguments called `run`.

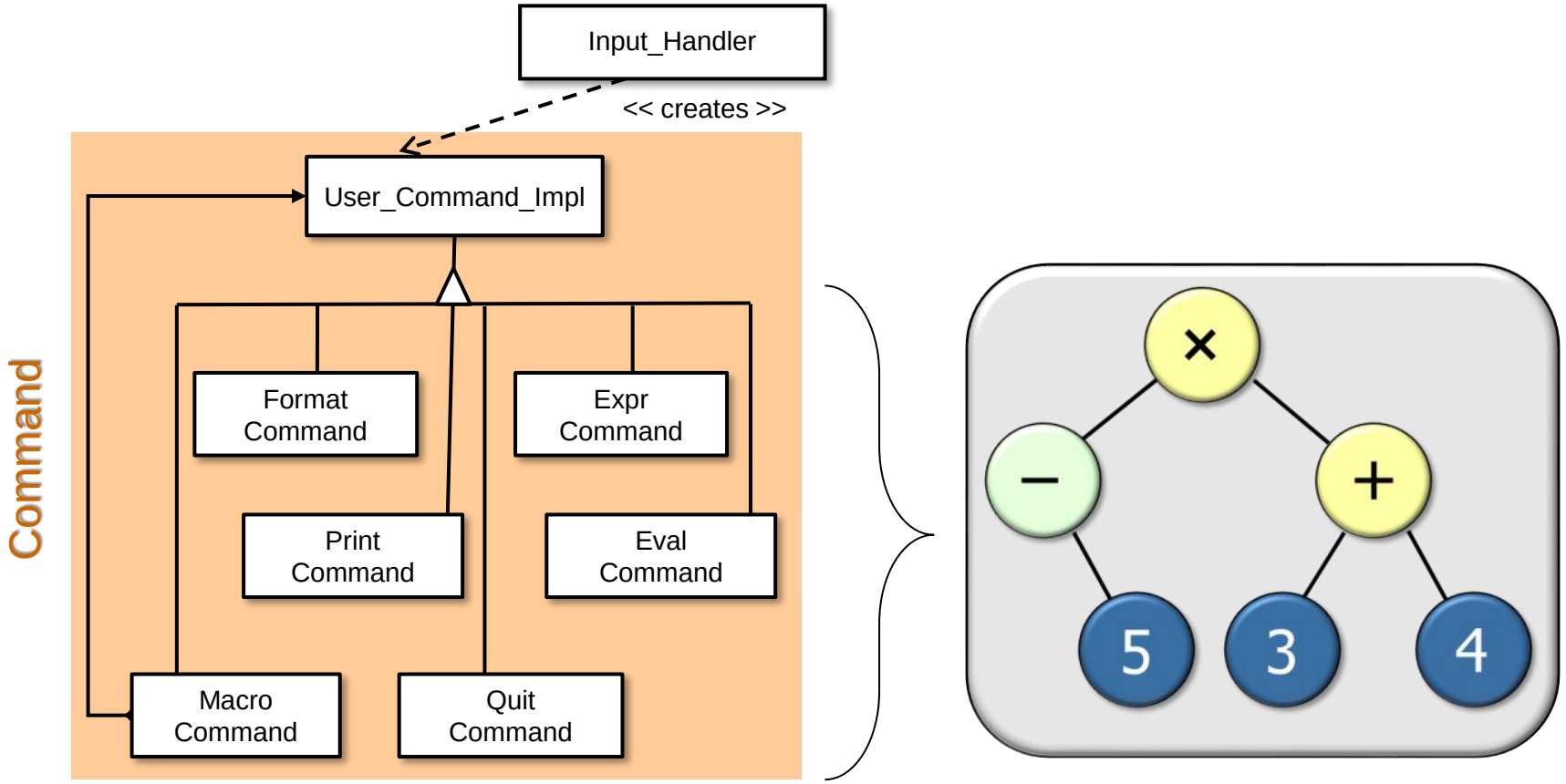


Packages a piece of application functionality—as well as its parameterization in an object—to make it usable in another context

See www.dre.vanderbilt.edu/~schmidt/CommandProcessor.pdf

Summary of the Command Pattern

- *Command* ensures users interact with the expression tree processing app in a consistent & extensible manner.



Command provides a uniform means to process all user-requested operations.