The Command Pattern
Implementation in C++

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++.
Douglas C. Schmidt

Implementing the Command Pattern in C++
Command example in C++

- Plays role of “Command” in the Command pattern
- Defines an API for “Concrete Command” implementations that perform an operation on the expression tree when it's executed

```cpp
class User_Command_Impl {
    Tree_Context &tree_context_; 

    User_Command_Impl (Tree_Context &tree_context) {
        tree_context_ = tree_context;
    }

    virtual void execute() = 0;
}
```
Command example in C++

- Plays role of “Command” in the `Command` pattern
- Defines an API for “Concrete Command” implementations that perform an operation on the expression tree when it's executed

```cpp
class User_Command_Impl {
    Tree_Context &tree_context_;  

    User_Command_Impl(Tree_Context &tree_context) {
        tree_context_ = tree_context;
    }

    virtual void execute() = 0;
};
```

See upcoming lesson on the `State` pattern
Command example in C++

- Plays role of “Command” in the Command pattern
- Defines an API for “Concrete Command” implementations that perform an operation on the expression tree when it's executed

```cpp
class User_Command_Impl {
    Tree_Context &tree_context_; 

    Constructor sets the field

    User_Command_Impl(Tree_Context & tree_context) {
        tree_context_ = tree_context;
    }

    virtual void execute() = 0;
};
```
Command example in C++

• Plays role of “Command” in the *Command* pattern

• Defines an API for “Concrete Command” implementations that perform an operation on the expression tree when it's executed

```cpp
class User_Command_Impl {
    Tree_Context &tree_context_;  

    User_Command_Impl(Tree_Context &tree_context) {
        tree_context_ = tree_context;
    }

    virtual void execute() = 0;

    // Concrete implementations run
    // the command via this method
```
**Command example in C++**

- Encapsulate the execution of a command object that sets the desired input expression.
- e.g., “\(-5 \times (3+4)\)"

```cpp
class Expr_Command
    : public User_Command_Impl {
    string expr_;

    Expr_Command(Tree_Context &context, string newexpr)
        : User_Command_Impl(context), expr_ (std::move(newexpr)) {
    }

    void execute() override {
        tree_context_.expr(expr_);
    }
}
```
Command example in C++

- Encapsulate the execution of a command object that sets the desired input expression.
  - e.g., “−5×(3+4)”

```cpp
class Expr_Command
    : public User_Command_Impl {
  string expr_;

  Expr_Command(Tree_Context &context, string newexpr)
      : User_Command_Impl(context),
        expr_ (std::move(newexpr)) {
  }

  void execute() override {
    tree_context_.expr(expr_);
  }
};
```
Command example in C++

- Encapsulate the execution of a command object that sets the desired input expression.
- e.g., “\(-5 \times (3+4)\)"

```cpp
class Expr_Command :
    public User_Command_Impl {
  string expr_; 

  Expr_Command(Tree_Context &context, string newexpr) :
      User_Command_Impl(context),
      expr_ (std::move(newexpr)) {

  } 

  void execute() override {
    tree_context_.expr(expr_);
  }

  // Provide Tree_Context & requested expression
```

Command example in C++

- Encapsulate the execution of a command object that sets the desired input expression.
- e.g., 

```
−5×(3+4)
```

```cpp
class Expr_Command
    : public User_Command_Impl {
    string expr_; 

    Expr_Command(Tree_Context &context, 
                 string newexpr) 
        : User_Command_Impl(context), 
          expr_ (std::move(newexpr)) 
    
    void execute() override {
        tree_context_.expr(expr_); 
    }
}
```

Forward to Tree_Context to create desired expression tree

See upcoming lesson on the State pattern
Command example in C++

- Encapsulate the execution of a sequence of commands as an object, which is used to implement the “succinct mode.”
- e.g., “−5×(3+4)”

```c++
formatCommand().execute();
exprCommand().execute();
evalCommand().execute();
```

```
User_Command_Impl
execute()

for all c in commands
  c.execute()
```

```
Macro_Command
execute()
```

GoF Object Behavioral
Command example in C++

- Encapsulate the execution of a sequence of commands as an object, which is used to implement the “succinct mode.”

```cpp
class Macro_Command : public User_Command_Impl {
    ...
    vector<User_Command> macro_command_;

    Macro_Command(Tree_Context &context,
        vector<User_Command> macro_command)
        : User_Command_Impl(context),
        macro_command_(std::move(macro_command)) {
    }

    void execute() override {
        for (auto &command : macro_command_) command.execute();
    }

    ...
```

GoF Object Behavioral
Command example in C++

- Encapsulate the execution of a sequence of commands as an object, which is used to implement the “succinct mode.”

```cpp
class Macro_Command : public User_Command_Impl {
    ...
    vector<User_Command> macro_command_;  

    Macro_Command(Tree_Context &context,  
                   vector<User_Command> macro_command)  
        : User_Command_Impl(context),  
          macro_command_(std::move(macro_command)) {
    }

    void execute() {  
        for (auto &command : macro_command_) command.execute();
    }
} ...
```
Command example in C++

- Encapsulate the execution of a sequence of commands as an object, which is used to implement the “succinct mode.”

```cpp
class Macro_Command : public User_Command_Impl {
  ...
  vector<User_Command> macro_command_;  

  Constructor initializes the field

  Macro_Command(Tree_Context &context,  
                vector<User_Command> macro_command)  
    : User_Command_Impl(context),  
      macro_command_(std::move(macro_command));
  }

  void execute() {
    for (auto &command : macro_command_) command.execute();
  }
  ...
```
Command example in C++

- Encapsulate the execution of a sequence of commands as an object, which is used to implement the “succinct mode.”

```cpp
class MacroCommand : public UserCommand_Impl {
    ...
    vector<UserCommand> macro_command_; 

    MacroCommand(Tree_Context &context, 
                  vector<UserCommand> macro_command) :
        UserCommand_Impl(context),
        macro_command_(std::move(macro_command)) {
    }

    void execute() {
        for (auto &command : macro_command_) command.execute();
    }
    ...
}
```

C++ range-based for loop runs all commands to implement “succinct mode”
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.
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

ConcreteCommand

<table>
<thead>
<tr>
<th>execute()</th>
<th>state</th>
</tr>
</thead>
</table>

performAction()
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

```c
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

See the next lesson on “The Factory Method Pattern” for User_Command_Factory.
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

```java
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

```java
formatCommand().execute();
exprCommand().execute();
evalCommand().execute();
```

**User_Command**

```java
execute()
```

**Macro_Command**

```java
for all c in commands
c.execute()
```
Consequences

+ Supports arbitrary-level undo-redo

**Undo:**
- `unexecute()`

**Redo:**
- `execute()`

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

**Undo:**
- `unexecute()`

**Redo:**
- `execute()`
Implementation considerations

- Avoiding error accumulation during undo/redo
Implementation considerations

- Supporting transactions

**Undo:**
unexecute()}

**Redo:**
execute()
**Known uses**

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

---

**Interface Runnable**

**All Known Subinterfaces:**

RunnableFuture&lt;V&gt;, RunnableScheduledFuture&lt;V&gt;

**All Known Implementing Classes:**

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

```java
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](https://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

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.

*Command*