A Case Study of “Gang of Four”
(GoF) Patterns : Part 8

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

Professor of Computer Science
Institute for Software Integrated Systems
Vanderbilt University
Nashville, Tennessee, USA
Topics Covered in this Part of the Module

- Describe the object-oriented (OO) expression tree case study
- Evaluate the limitations with algorithmic design techniques
- Present an OO design for the expression tree processing app
- Summarize the patterns in the expression tree design
- Explore patterns for
  - Tree structure & access
  - Tree creation
  - Tree traversal
  - Commands & factories
Overview of Command & Factory Patterns

**Purpose:** Define operations that can users can perform on an expression tree processing app & centralize extensible creation of these operations

These patterns decouple creation from use & provide a uniform command API
Problem: Consolidating User Operations

Goals

• Support execution of user operations

% tree-traversal -v
format [in-order]
expr [expression]
print [in-order|pre-order|post-order|level-order]
eval [post-order]
quit

> format in-order
> expr 1+4*3/2
> eval post-order
    7
> quit

% tree-traversal
> 1+4*3/2
    7
Problem: Consolidating User Operations

**Goals**

- Support execution of user operations
- Support macro operations

```
% tree-traversal -v
format [in-order]
expr [expression]
print [in-order|pre-order|post-order|level-order]
eval [post-order]
quit

> format in-order
> expr 1+4*3/2
> eval post-order
7
> quit

% tree-traversal
> 1+4*3/2
7
```

*Succinct mode*
Problem: Consolidating User Operations

Goals
• Support execution of user operations
• Support macro operations

Constraints/forces
• Avoid scattering the implementation of operations throughout the source code
• Ensure consistent memory management
Solution: Encapsulate an Operation w/Command

- A **Command** encapsulates
  - An operation method (**execute()**)
  - An inverse operation method (**unexecute()**)
  - A test for reversibility (**boolean reversible()**)
  - State for (un)doing the operation
Solution: Encapsulate an Operation w/ Command

- A **Command** encapsulates
  - An operation method (`execute()`)
  - An inverse operation method (`unexecute()`)
  - A test for reversibility (`boolean reversible()`)
  - State for (un)doing the operation
- A **Command** may
  - Implement the operation itself or
  - Forward the operation implementation to other object(s)

*Bridge pattern encapsulates variability & simplifies memory management*
**ET_Command Class Interface**

- Interface for defining a command that—when executed—performs an operation on an expression tree

```cpp
ET_Command(ET_Command_Impl *=0)
ET_Command(const ET_Command &)
ET_Command & operator=(const ET_Command &)
~ET_Command()
bool execute()
bool unexecute()
```

This class plays the role of the abstraction in the *Bridge* pattern

- **Commonality:** Provides common interface for expression tree commands
- **Variability:** Implementations of expression tree commands can vary depending on the operations requested by user input

These methods forward to the implementor subclass

We don’t use this method but it’s a common command feature
ET_Command_Impl Class Interface

- Base class of an implementor hierarchy used to define commands that perform operations on an expression tree when they are executed.

**Interface**

This class is the base class of the implementor hierarchy in the *Bridge* pattern.

```c++
ET_Command_Impl(ET_Context &)
~ET_Command_Impl() = 0

virtual bool execute() = 0
virtual bool unexecute() = 0
```

- **Commonality**: Provides a common base class for implementations of expression tree commands.
- **Variability**: Subclasses of this base class implement different expression tree commands depending on the operations requested by user input.

The bulk of the work is typically done here by subclasses.
**Intent**

- Encapsulate the request for a service as an object

**Applicability**

- Want to parameterize objects with an action to perform
- Want to specify, queue, & execute requests at different times
- For multilevel undo/redo

**Structure**

```
Client --> Invoker --> Command
    |            |           |
    |            | execute() |
    |            |           |
    V            V           |
Target --> Target
    |         |
    | action()|
    |         |

ConcreteCommand
    execute()  target
    state

```

e.g., ET_Command_Impl

e.g., Eval_Command, Format_Command, Print_Command, Quit_Command, Macro_Command, etc.
Command example in C++

- Encapsulate execution of a sequence of commands as an object

```cpp
class Macro_Command : public ET_Command_Impl {
public:
...
bool execute() {
    std::for_each (macro_commands_.begin(),
                   macro_commands_.end(),
                   std::mem_fun_ref(&ET_Command::execute));
    return true;
}
}

private:
std::vector <ET_Command> macro_commands_;  
...  
```

Executes a sequence of commands (used to implement “succinct mode”)

Vector of commands to execute as a macro
Command example in C++

- Encapsulate execution of a sequence of commands as an object

```cpp
class Macro_Command : public ET_Command_Impl {
public:
    ...
    bool execute() {
        std::for_each (macro_commands_.begin(),
                       macro_commands_.end(),
                       std::mem_fun_ref(&ET_Command::execute));
        return true;
    }
}
```

Application of the Adapter pattern

- Vector of commands to execute as a macro
Command example in C++

- Encapsulate execution of a sequence of subcommands as an object

```cpp
class Macro_Command : public ET_Command_Impl {
public:

    // ...

    bool execute() {
        std::for_each (macro_commands_.begin(),
                       macro_commands_.end(),
                       [](ET_Command &c){ c.execute(); });

        return true;
    }

private:

    std::vector <ET_Command> macro_commands_;

    // ...
};
```

C++11 lambda expression
Command example in C++

- Encapsulate execution of a sequence of subcommands as an object

```cpp
class Macro_Command : public ET_Command_Impl {
public:
    ...
    bool execute() {
        for (auto &iter = macro_commands_)
            iter.execute();

        return true;
    }

private:
    std::vector<ET_Command> macro_commands_; 
    ...
```

C++11 range-based for loop
Command | GoF Object Behavioral

**Consequences**

+ Abstracts executor of a service
+ Supports arbitrary-level undo-redo
+ Composition yields macro-commands
  - Might result in lots of trivial command subclasses
  - Excessive memory may be needed to support undo/redo operations
## Command

<table>
<thead>
<tr>
<th>GoF Patterns Expression Tree Case Study</th>
<th>GoF Object Behavioral</th>
</tr>
</thead>
</table>

### Consequences

+ Abstracts executor of a service
+ Supports arbitrary-level undo-redo
+ Composition yields macro-commands
  - Might result in lots of trivial command subclasses
  - Excessive memory may be needed to support undo/redo operations

### Implementation

- Copying a command before putting it on a history list
- Avoiding error accumulation during undo/redo
- Supporting transactions
## Command

### Consequences
- Abstracts executor of a service
- Supports arbitrary-level undo-redo
- Composition yields macro-commands
  - Might result in lots of trivial command subclasses
  - Excessive memory may be needed to support undo/redo operations

### Implementation
- Copying a command before putting it on a history list
- Avoiding error accumulation during undo/redo
- Supporting transactions

## Known Uses
- InterViews Actions
- MacApp, Unidraw Commands
- JDK’s UndoableEdit, AccessibleAction
- Emacs
- Microsoft Office tools

## See Also
- *Command Processor* pattern in POSA1
Problem: Consolidating Creation of Variabilities

Goals

• Simplify & centralize the creation of all variabilities in the expression tree application to ensure semantic compatibility
• Be extensible for future variabilities
Problem: Consolidating Creation of Variabilities

Goals

- Simplify & centralize the creation of all variabilities in the expression tree application to ensure semantic compatibility
- Be extensible for future variabilities

Constraints/forces

- Don’t recode existing clients
- Add new variabilities without recompiling

Diagram:

```
ET_Command -> ET_Command_Impl

<table>
<thead>
<tr>
<th>Format_Command</th>
<th>Print_Command</th>
<th>Eval_Command</th>
<th>Quit_Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expr_Command</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ET_Iterator -> ET_Iterator_Impl

<table>
<thead>
<tr>
<th>In_Order_ET_Iterator_Impl</th>
<th>Post_Order_ET_Iterator_Impl</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level_Order_ET_Iterator_Impl</td>
<td>Pre_Order_ET_Iterator_Impl</td>
</tr>
</tbody>
</table>
```
Solution: Abstract Object Creation

Factory

• Instead of

    `ET_Command command(new Print_Command());`

Use

    `ET_Command command (command_factory.make_command("print"));`

where `command_factory` is an instance of `ET_Command_Factory`
Solution: Abstract Object Creation

Factory

- Instead of

```cpp
ET_Command command(new Print_Command());
```

Use

```cpp
ET_Command command
    (command_factory.make_command("print"));
```

where `command_factory` is an instance of `ET_Command_Factory`

Factory structure

- **Bridge pattern** encapsulates variability & simplifies memory management
**ET_Command_Factory Class Interface**

- Interface used to create appropriate command based on string supplied by caller

**Interface**

- `ET_Command_Factory(ET_Context &tree_context)`
- `ET_Command_Factory(ET_Command_Factory_Impl *)`

**ET_Command**

- `make_command(const std::string &s)`
- `make_format_command(const std::string &)`
- `make_expr_command(const std::string &)`
- `make_print_command(const std::string &)`
- `make_eval_command(const std::string &)`
- `make_quit_command(const std::string &)`
- `make_macro_command(const std::string &)`

**Commonality**: Provides a common interface to create commands

**Variability**: Implementations of expression tree command factory methods can vary depending on the requested commands

This class plays the role of the abstraction in the **Bridge** pattern
ET_Command_Factory_Impl Class Interface

- Base class of an implementor hierarchy used to create appropriate commands based on string supplied by caller

**Interface**

```
ET_Command_Factory_Impl (ET_Context &context)
~ET_Command_Factory_Impl()
```

**Virtual Methods**

- `virtual ET_Command make_command (const std::string &s)=0`
- `virtual ET_Command make_format_command (const std::string &)=0`
- `virtual ET_Command make_expr_command (const std::string &)=0`
- `virtual ET_Command make_print_command (const std::string &)=0`
- `virtual ET_Command make_eval_command (const std::string &)=0`
- `virtual ET_Command make_quit_command (const std::string &)=0`
- `virtual ET_Command make_macro_command (const std::string &)=0`

**Commonality:** Provides a common interface to create commands

**Variability:** Subclasses of this base class define expression tree command factory methods vary depending on the requested commands
Each factory method creates a different type of concrete command

*Bridge* pattern encapsulates variability & simplifies memory management
**Intent**
- Provide an interface for creating an object, but leave choice of object’s concrete type to a subclass

**Applicability**
- When a class cannot anticipate the objects it must create or a class wants its subclasses to specify the objects it creates

**Structure**
- **Product**
  - **Creator**
    - `FactoryMethod()`
    - `AnOperation()`
    - `... product = FactoryMethod()`
    - `...`
  - **ConcreteCreator**
    - `FactoryMethod()`
    - `return new ConcreteProduct`
  - **ConcreteProduct**

- **e.g., ET_Command_Impl**
- **e.g., ET_Command_Factory_Impl**
- **e.g., Concrete_ET_Command_Factory_Impl**
- **e.g., Eval_Command, Print_Command, Macro_Command, etc.**
Factory Method example in C++

- An interface for creating a command, letting subclass choose concrete type

```cpp
class ET_Command.Factory_Impl {
public:
    virtual ET_Command make_macro_command(const std::string &) = 0;
    ...
```
Factory Method example in C++

- An interface for creating a command, letting subclass chose concrete type

```cpp
class ET_Command_Factory_Impl {
public:
    virtual ET_Command make_macro_command(const std::string &) = 0;
    ...

class Concrete_ET_Command_Factory_Impl : public ET_Command_Factory_Impl {
public:
    virtual ET_Command make_macro_command(const std::string &expr) {
        std::vector<ET_Command> commands;
        commands.push_back(make_format_command("in-order"));
        commands.push_back(make_expr_command(expr));
        commands.push_back(make_eval_command("post-order"));
        return ET_Command(new Macro_Command(tree_context_, commands));
    }
}
```

Create vector of commands that are executed as a macro

Encapsulates command within Bridge abstraction object
<table>
<thead>
<tr>
<th>Factory Method</th>
<th>GoF Class Creational</th>
</tr>
</thead>
</table>

**Consequences**

+ *Flexibility*: The client becomes more flexible by not specifying the class name of the concrete class & the details of its creation

+ *Decoupling*: The client only depends on the interface

- *More classes*: Construction of objects requires an additional class in some cases
**Factory Method**

**GoF Class Creational**

**Consequences**

+ *Flexibility*: The client becomes more flexible by not specifying the class name of the concrete class & the details of its creation

+ *Decoupling*: The client only depends on the interface

- *More classes*: Construction of objects requires an additional class in some cases

**Implementation**

• There are two choices
  
  • The creator class is abstract & does not implement creation methods (then it must be subclassed)
  
  • The creator class is concrete & provides a default implementation (then it can be subclassed)

• If a factory method can create different variants the method should be passed a parameter to designate the variant
**Factory Method**

**GoF Class Creational**

### Consequences

+ **Flexibility**: The client becomes more flexible by not specifying the class name of the concrete class & the details of its creation
+ **Decoupling**: The client only depends on the interface
+ **More classes**: Construction of objects requires an additional class in some cases

### Implementation

- There are two choices
  - The creator class is abstract & does not implement creation methods (then it must be subclassed)
  - The creator class is concrete & provides a default implementation (then it can be subclassed)
- If a factory method can create different variants the method must be provided with a parameter

### Known Uses

- InterViews Kits
- ET++ WindowSystem
- AWT Toolkit
- The ACE ORB (TAO)
- BREW feature phone frameworks