The Template Method Pattern

Motivating Example

Douglas C. Schmidt
Learning Objectives in This Lesson

• Recognize how the Template Method pattern can be applied to flexibly support multiple operating modes in the expression tree processing app.
Motivating the Need for the Template Method Pattern in the Expression Tree App

Douglas C. Schmidt
A Pattern for Encapsulating Algorithm Variability

**Purpose:** Factor out common code to support multiple operating modes (succinct vs. verbose).

---

**Template Method** supports controlled variability of steps in an algorithm.
This app has two primary operating modes: **verbose** & **succinct**.
Problem: Non-Extensible Operating Modes

- Structuring the program in terms of the two operating modes’ algorithms is problematic.
Problem: Non-Extensible Operating Modes

- Structuring the program in terms of the two operating modes’ algorithms is problematic, e.g.,
- Incurs many of the same limitations as algorithmic decomposition
- e.g., complexity will reside in (variable) algorithms rather than (stable) structure

```c
void print_tree (Tree_Node *root)
{
    switch (root->tag_)
    {
    case NUM: printf ("%d", root->num);
    break;
    case UNARY:
        printf ("(%s", root->op_[0]);
        print_tree (root->unary_);
        printf (")"); break;
    case BINARY:
        printf ("(");
        print_tree (root->binary_l_);
        printf ("%s", root->op_[0]);
        print_tree (root->binary_r_);
        printf (")"); break;
    default:
        printf ("error, unknown type\n");
    }
}
```
Problem: Non-Extensible Operating Modes

- Structuring the program in terms of the two operating modes’ algorithms is problematic, e.g.,
  - Incurs many of the same limitations as algorithmic decomposition
  - Impedes maintainability & evolution of the code base due to “silo’ing”
Problem: Non-Extensible Operating Modes

- Structuring the program in terms of the two operating modes’ algorithms is problematic, e.g.,
  - Incurs many of the same limitations as algorithmic decomposition
  - Impedes maintainability & evolution of the code base due to “silo’ing”, e.g.,
    - Verbose mode algorithm bug fixes & improvements won’t be reused by succinct mode algorithms
Problem: Non-Extensible Operating Modes

- Structuring the program in terms of the two operating modes’ algorithms is problematic, e.g.,
  - Incurs many of the same limitations as algorithmic decomposition
  - Impedes maintainability & evolution of the code base due to “silo’ing”, e.g.,
    -Verbose mode algorithm bug fixes & improvements won’t be reused by succinct mode algorithms
  - Violates the “Don’t Repeat Yourself” (DRY) principle

See en.wikipedia.org/wiki/Don’t_repeat_yourself
Solution: Encapsulate Algorithm Variability

- Implement algorithm once in super class

```cpp
ET_Event_Handler

void handle_input()
{
    prompt_user();
    string input = receive_input();
    User_Command command = make_command(input);
    execute_command(command);
}

handle_input() is a template method.
```
Solution: Encapsulate Algorithm Variability

- Implement algorithm once in super class

```cpp
void handle_input()
{
    prompt_user();
    string input = receive_input();
    User_Command command = make_command(input);
    execute_command(command);
}
```

The other four methods are "hook methods."

`handle_input()` is a template method.
Solution: Encapsulate Algorithm Variability

- Implement algorithm once in super class & let subclasses define variants.

```java
void handle_input()
{
    prompt_user();
    string input = receive_input();
    User_Command command = make_command(input);
    execute_command(command);
}
```
Solution: Encapsulate Algorithm Variability

- Implement algorithm once in super class & let subclasses define variants.

```cpp
void handle_input()
{
    prompt_user();
    string input = receive_input();
    User_Command command = make_command(input);
    execute_command(command);
}
```

See earlier lesson on “The Command Pattern: Implementation in C++.”
Solution: Encapsulate Algorithm Variability

- Implement algorithm once in super class & let subclasses define variants.

```cpp
User_Command make_command(string input)
{
    return command_factory_.make_command(input);
}
```

```cpp
void handle_input()
{
    prompt_user();
    string input = receive_input();
    User_Command command = make_command(input);
    execute_command(command);
}
```

See earlier lesson on “The Command Pattern: Implementation in C++.”
Solution: Encapsulate Algorithm Variability

- Implement algorithm once in super class & let subclasses define variants.

```c++
User_Command make_command(string input) {
    return command_factory_.make_command(input);
}

User_Command make_command(string input) {
    return command_factory_.make_macro_command(input);
}

void handle_input() {
    prompt_user();
    string input = receive_input();
    User_Command command =
        make_command(input);
    execute_command(command);
}
```

This solution increases opportunities for systematic software reuse.
ET_Event_Handler Class Overview

- An abstract class that provides the boilerplate algorithm for controlling the operating modes of the expression tree processing app

Class methods

```java
void handle_input()
void prompt_user()
String receive_input()
User_Command make_command(string input)
void execute_command(User_Command command)
static ET_Event_Handler make_handler(bool verbose)
```
ET_Event_Handler Class Overview

- An abstract class that provides the boilerplate algorithm for controlling the operating modes of the expression tree processing app

Class methods

Template method

```java
void handle_input()
void prompt_user()
String receive_input()
User_Command make_command(string input)
void execute_command(User_Command command)
static ET_Event_Handler make_handler(bool verbose)
```
ET_Event_Handler Class Overview

- An abstract class that provides the boilerplate algorithm for controlling the
  operating modes of the expression tree processing app

Class methods

```java
void handle_input()
void prompt_user()
String receive_input()
User_Command make_command(String input)
void execute_command(User_Command command)
static ET_Event_Handler make_handler(bool verbose)
```
ET_Event_Handler Class Overview

- An abstract class that provides the boilerplate algorithm for controlling the operating modes of the expression tree processing app

Class methods

```java
void handle_input()
void prompt_user()
String receive_input()
User_Command make_command(String input)
void execute_command(User_Command command)
static ET_Event_Handler make_handler(bool verbose)
```

Factory method
ET_Event_Handler Class Overview

- An abstract class that provides the boilerplate algorithm for controlling the operating modes of the expression tree processing app

Class methods

```cpp
void handle_input()
void prompt_user()
String receive_input()
User_Command make_command(string input)
void execute_command(User_Command command)
static ET_Event_Handler make_handler(bool verbose)
```

- **Commonality**: provides a common interface for handling user input events & performing steps in the expression tree processing algorithm
- **Variability**: subclasses implement various operating modes, e.g., verbose vs. succinct mode
ET_Event_Handler Class Hierarchy Overview

- The subclasses of **ET_Event_Handler** override several of its hook methods to implement the “verbose” & “succinct” operating modes.

  ```
  ET_Event_Handler
  handle_input()
prompt_user()
receive_input()
make_command()
execute_command()
  
  Verbese_Mode
  ET_Event_Handler
  prompt_user()
  make_command()

  Succinct_Mode
  ET_Event_Handler
  prompt_user()
  make_command()

  Enables reuse of the structure (& some functionality) of ET_Event_Handler
  ```