The Template Method Pattern

Structure & Functionality

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.
- Understand the structure & functionality of the *Template Method* pattern.
Structure & Functionality of the Template Method Pattern

Douglas C. Schmidt
Template Method

Intent

• Provide an algorithm skeleton in a method, deferring some steps to subclasses

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

See [en.wikipedia.org/wiki/Template_method_pattern](en.wikipedia.org/wiki/Template_method_pattern)
Applicability

- Implement invariant aspects of an algorithm *once* & let subclasses define variant parts

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

- Implement invariant aspects of an algorithm *once* & let subclasses define variant parts
- Localize common behavior in a class to enhance reuse

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

- Implement invariant aspects of an algorithm *once* & let subclasses define variant parts
- Localize common behavior in a class to enhance reuse
- Handle variations in behavior via subclassing
Structure & participants

```
AbstractClass

TemplateMethod()
PrimitiveOperation1()
PrimitiveOperation2()

ConcreteClass

PrimitiveOperation1()
PrimitiveOperation2()
```
Template Method

GoF Class Behavioral

Structure & participants

AbstractClass

TemplateMethod()
PrimitiveOperation1()
PrimitiveOperation2()

ConcreteClass

PrimitiveOperation1()
PrimitiveOperation2()

ET_Event_Handler
Template Method

Class Behavioral

Structure & participants

AbstractClass
  TemplateMethod()
  PrimitiveOperation1()
  PrimitiveOperation2()

ConcreteClass
  PrimitiveOperation1()
  PrimitiveOperation2()

Verbose_Mode_ET_Event_Handler,
Succinct_Mode_ET_Event_Handler
“Primitive operations” are often referred to as “hook methods,” which provide customization points in a software framework.

See wiki.c2.com/?HookMethod
The template method uses virtual method calls to dynamically dispatch certain hook methods to subclass implementations.

This pattern is “OOP 101” www.stroustrup.com/whatis.pdf
The Template Method Pattern

Implementation in C++

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.
- Understand the structure & functionality of the Template Method pattern.
- Know how to implement the Template Method pattern in C++.
Template Method example in C++

• Allow subclasses to customize certain steps in the input handling algorithm.

```cpp
class ET_Event_Handler : public Event_Handler {
    ...
    void handle_input() override {
        prompt_user();
        string input = receive_input();
        User_Command command = make_command(input);
        execute_command(command);
    }
```
Template Method example in C++

- Allow subclasses to customize certain steps in the input handling algorithm.

```cpp
class ET_Event_Handler : public Event_Handler {
    ...

    void handle_input() override {
        prompt_user();
        string input = receive_input();
        User_Command command = make_command(input);
        execute_command(command);
    }
}
```
Template Method example in C++

- Allow subclasses to customize certain steps in the input handling algorithm.

```cpp
class ET_Event_Handler : public Event_Handler {
    ...
    void handle_input() override {
        prompt_user();
        string input = receive_input();
        User_Command command = make_command(input);
        execute_command(command);
    }

    unique_ptr<Event_Handler> make_handler(bool verbose) { 
        return verbose 
            ? make_unique<Verbose_Mode_ET_Event_Handler>() 
            : make_unique<Succint_Mode_ET_Event_Handler>();
    }
}
```

**Factory method creates designated concrete classes**

This is not the only/best way to define a factory since it’s too tightly coupled.
Template Method example in C++

• Allow subclasses to customize certain steps in the input handling algorithm.

```cpp
class Verbose_Mode_ET_Event_Handler : public ET_Event_Handler {
    ...

    public User_Command make_command(string user_input) {
        return command_factory_.make_command
            (user_input);
    }
};
```
Template Method example in C++

- Allow subclasses to customize certain steps in the input handling algorithm.

```cpp
class Verbose_Mode_ET_Event_Handler : public ET_Event_Handler {
    ...
    public User_Command make_command(string user_input) {
        return command_factory_.make_command (user_input);
    }
}

class Succinct_Mode_ET_Event_Handler : public ET_Event_Handler {
    ...
    public User_Command make_command(string user_input) {
        return command_factory_.make_macro_command (user_input);
    }
}
```

Specialized hook method
Template Method example in C++

- Allow subclasses to customize certain steps in the input handling algorithm.

```cpp
class Verbose_Mode_ET_Event_Handler : public ET_Event_Handler {
    ...
    public User_Command make_command(string user_input) {
        return command_factory_.make_command(user_input);
    }
}

class Succinct_Mode_ET_Event_Handler : public ET_Event_Handler {
    ...
    public User_Command make_command(string user_input) {
        return command_factory_.make_macrp+command(user_input);
    }
}
```

See earlier lesson on "The Factory Method Pattern" for User_Command_Factory
The Template Method Pattern

Other Considerations

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.
- Understand the structure & functionality of the *Template Method* pattern.
- Know how to implement the *Template Method* pattern in C++.
- Be aware of other considerations when applying the *Template Method* pattern.
Consequences
+ Enables inversion of control
  • “Hollywood principle” — don’t call us, we’ll call you!

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

See [www.dre.vanderbilt.edu/~schmidt/Coursera/articles/hollywood-principle.txt](http://www.dre.vanderbilt.edu/~schmidt/Coursera/articles/hollywood-principle.txt)
Consequences

+ Overriding rules are enforced via subclassing

**ET_Event_Handler**

- handle_input()
- prompt_user()
- receive_input()
- make_command()
- execute_command()

**Verbose_Mode**

**ET_Event_Handler**

- prompt_user()
- make_command()

**Succinct_Mode**

**ET_Event_Handler**

- prompt_user()
- make_command()
Consequences

+ Promotes systematic reuse by collapsing stovepipes

[Image of recycling symbol]

Variant (non-reusable) code

[Code snippet]

Common (reusable) code

[Code snippet]

www.dre.vanderbilt.edu/~schmidt/reuse-lessons.html has info on systematic reuse.
Consequences

- Must subclass to specialize behavior, which can yield many subclasses
- Compare & contrast with the Strategy pattern

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

```
Verbose_Mode
ET_Event_Handler
prompt_user()
make_command()
```

```
Succinct_Mode
ET_Event_Handler
prompt_user()
make_command()
```
Consequences

- Must subclass to specialize behavior, which can yield many subclasses
- Compare & contrast with the *Strategy* pattern

\[
\text{ET\_Event\_Handler} \\
\text{handle\_input()}
\]

\[
\text{prompt\_user()}
\]

\[
\text{receive\_input()}
\]

\[
\text{make\_command()}
\]

\[
\text{execute\_command()}
\]

---

\[
\text{Verbose\_Mode} \\
\text{ET\_Event\_Handler}
\]

\[
\text{prompt\_user()}
\]

\[
\text{make\_command()}
\]

---

\[
\text{Succinct\_Mode} \\
\text{ET\_Event\_Handler}
\]

\[
\text{prompt\_user()}
\]

\[
\text{make\_command()}
\]

---

*C++ lambda functions may help reduce the tedium of creating many subclasses.*
Implementation considerations

- Virtual vs. non-virtual (final) template method
- Depends on whether the algorithm embodied by the template method itself may need to change

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

- Few vs. many primitive operations (hook methods)
- e.g., how much variability’s needed in the template method’s algorithm?

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

- Naming conventions
  - e.g., `do*()` vs. `make*()` vs. `on*()` prefixes

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

- InterViews Kits
- ET++ WindowSystem
- AWT Toolkit
Known uses

• InterViews Kits
• ET++ WindowSystem
• AWT Toolkit
• ACE & The ACE ORB (TAO)

See www.dre.vanderbilt.edu/~schmidt/PDF/BC-schmidt.pdf
Known uses

- InterViews Kits
- ET++ WindowSystem
- AWT Toolkit
- ACE & The ACE ORB (TAO)
- Java Collections framework

```
public E set(int index, E element) {
    try {
        ListIterator<E> e = listIterator(index);
        ...
    }
}
```

See refactoring.guru/design-patterns/template-method/java/example
Known uses

- InterViews Kits
- ET++ WindowSystem
- AWT Toolkit
- ACE & The ACE ORB (TAO)
- Java Collections framework
- Android `AsyncTask` framework

**Template Method**

```
Template method
```

**Hook methods**

```
Params—types used in background work
Progress—types used when indicating progress
Result—types of result
```

```
AsyncTask

execute()
onPreExecute()
doInBackground()
onProgressUpdate()
onPostExecute()

MyAsyncTask

onPreExecute()
doInBackground()
onPostExecute()
```

Comparing Strategy With Template Method

**Strategy**

+ Provides for clean separation between components via “black-box” interfaces
+ Allows for strategy composition at runtime
+ Supports flexible mixing & matching of features
  - May yield many strategy classes
  - Incurs forwarding overhead

See [www.dre.vanderbilt.edu/~schmidt/PDF/DRC.pdf](www.dre.vanderbilt.edu/~schmidt/PDF/DRC.pdf)
We selected \textit{Template Method} for our case study since it’s a simple use case.

\subsection*{Comparing Strategy With Template Method}

\textbf{Strategy}

\begin{itemize}
  \item Provides for clean separation between components via “black-box” interfaces
  \item Allows for strategy composition at runtime
  \item Supports flexible mixing & matching of features
  \begin{itemize}
    \item May yield many strategy classes
    \item Incurs forwarding overhead
  \end{itemize}
\end{itemize}

\textbf{Template Method}

\begin{itemize}
  \item No explicit forwarding necessary
  \item May be easier for small use cases due to “white-box” interfaces
  \begin{itemize}
    \item Close coupling between subclass(es) & super class
    \item Inheritance hierarchies are static & cannot be reconfigured at runtime
    \item Adding features via inheritance may yield combinatorial subclass explosion
      \begin{itemize}
        \item Beware overusing inheritance since it’s not always the best choice.
        \item Deep inheritance hierarchies in an app are a red flag.
      \end{itemize}
  \end{itemize}
\end{itemize}
Strategy

+ Provides for clean separation between components via “black-box” interfaces
+ Allows for strategy composition at runtime
+ Supports flexible mixing & matching of features
  - May yield many strategy classes
  - Incurs forwarding overhead

Template Method

+ No explicit forwarding necessary
+ May be easier for small use cases due to “white-box” interfaces
  - Close coupling between subclass(es) & super class
  - Inheritance hierarchies are static & cannot be reconfigured at runtime
  - Adding features via inheritance may yield combinatorial subclass explosion
  - Beware overusing inheritance since it’s not always the best choice.
  - Deep inheritance hierarchies in an app are a red flag.

Strategy & Template Method are often treated as “pattern complements” since they provide alternative solutions to related design problems.

See [www.dre.vanderbilt.edu/~schmidt/POSA-tutorial.pdf](www.dre.vanderbilt.edu/~schmidt/POSA-tutorial.pdf)
Summary of the Template Method Pattern

- **Template Method** enables controlled variability of steps in the `ET_Event_Handler` algorithm for processing multiple operating modes, which enhances reuse.