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

+ Overiding rules are enforced via subclassing
Consequences

+ Promotes systematic reuse by collapsing stovepipes

Template Method

GoF Class Behavioral

Variant (non-reusable) code

Common (reusable) code

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

---

**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](http://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

Template Method

```
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

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

```
Template method

AsyncTask

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

Hook methods

MyAsyncTask

onPreExecute()
doInBackground()
onPostExecute()

See developer.android.com/reference/android/os/AsyncTask.html
```
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](http://www.dre.vanderbilt.edu/~schmidt/PDF/DRC.pdf)
Comparing Strategy With Template Method

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Template Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ Provides for clean separation between components via “black-box” interfaces</td>
<td>+ No explicit forwarding necessary</td>
</tr>
<tr>
<td>+ Allows for strategy composition at runtime</td>
<td>+ May be easier for small use cases due to “white-box” interfaces</td>
</tr>
<tr>
<td>+ Supports flexible mixing &amp; matching of features</td>
<td>– Close coupling between subclass(es) &amp; super class</td>
</tr>
<tr>
<td>– May yield many strategy classes</td>
<td>– Inheritance hierarchies are static &amp; cannot be reconfigured at runtime</td>
</tr>
<tr>
<td>– Incurs forwarding overhead</td>
<td>– Adding features via inheritance may yield combinatorial subclass explosion</td>
</tr>
<tr>
<td></td>
<td>– Beware overusing inheritance since it’s not always the best choice.</td>
</tr>
<tr>
<td></td>
<td>– Deep inheritance hierarchies in an app are a red flag.</td>
</tr>
</tbody>
</table>

We selected *Template Method* for our case study since it’s a simple use case.
## Comparing Strategy With Template Method

<table>
<thead>
<tr>
<th><strong>Strategy</strong></th>
<th><strong>Template Method</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>+ Provides for clean separation between components via “black-box” interfaces</td>
<td>+ No explicit forwarding necessary</td>
</tr>
<tr>
<td>+ Allows for strategy composition at runtime</td>
<td>+ May be easier for small use cases due to “white-box” interfaces</td>
</tr>
<tr>
<td>+ Supports flexible mixing &amp; matching of features</td>
<td>– Close coupling between subclass(es) &amp; super class</td>
</tr>
<tr>
<td>– May yield many strategy classes</td>
<td>– Inheritance hierarchies are static &amp; cannot be reconfigured at runtime</td>
</tr>
<tr>
<td>– Incurs forwarding overhead</td>
<td>– Adding features via inheritance may yield combinatorial subclass explosion</td>
</tr>
<tr>
<td></td>
<td>– Beware overusing inheritance since it’s not always the best choice.</td>
</tr>
<tr>
<td></td>
<td>– Deep inheritance hierarchies in an app are a red flag.</td>
</tr>
</tbody>
</table>

*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](http://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.