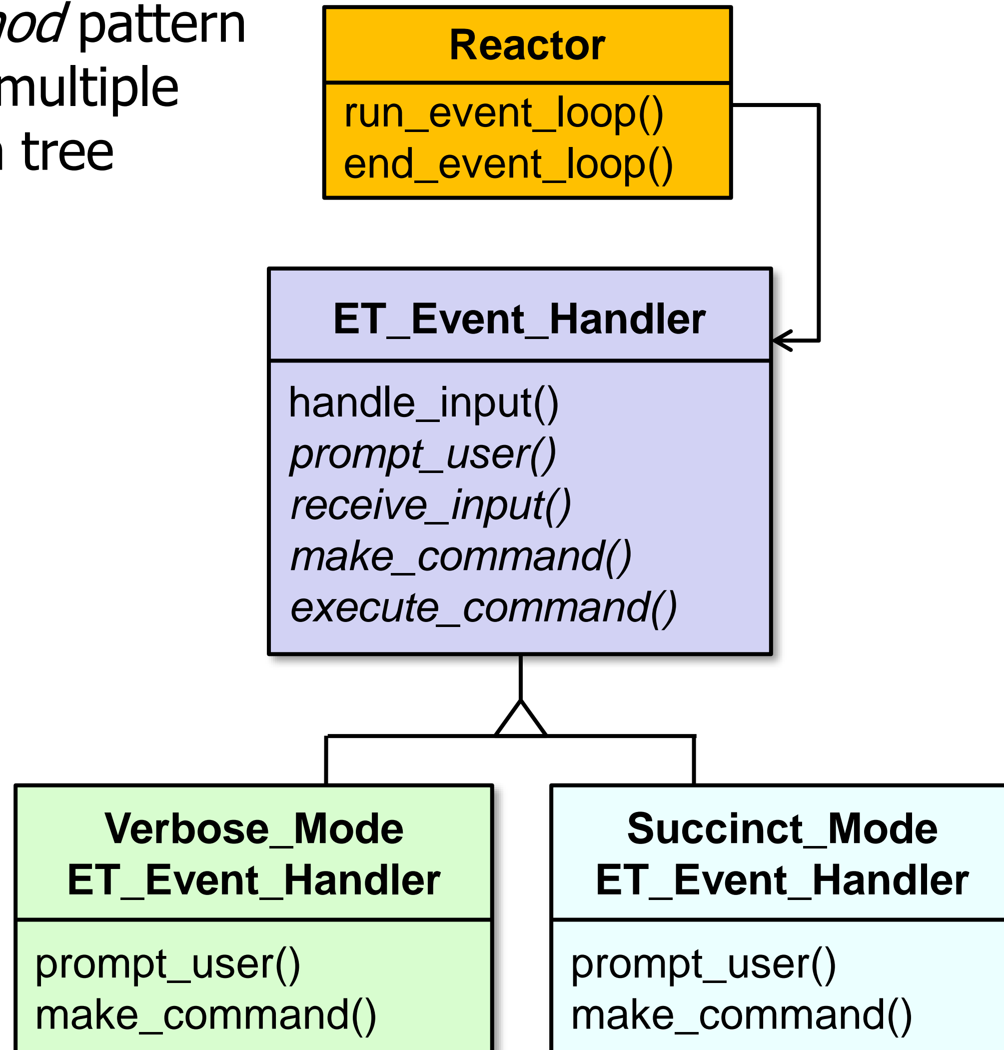
The Template Method Pattern

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.

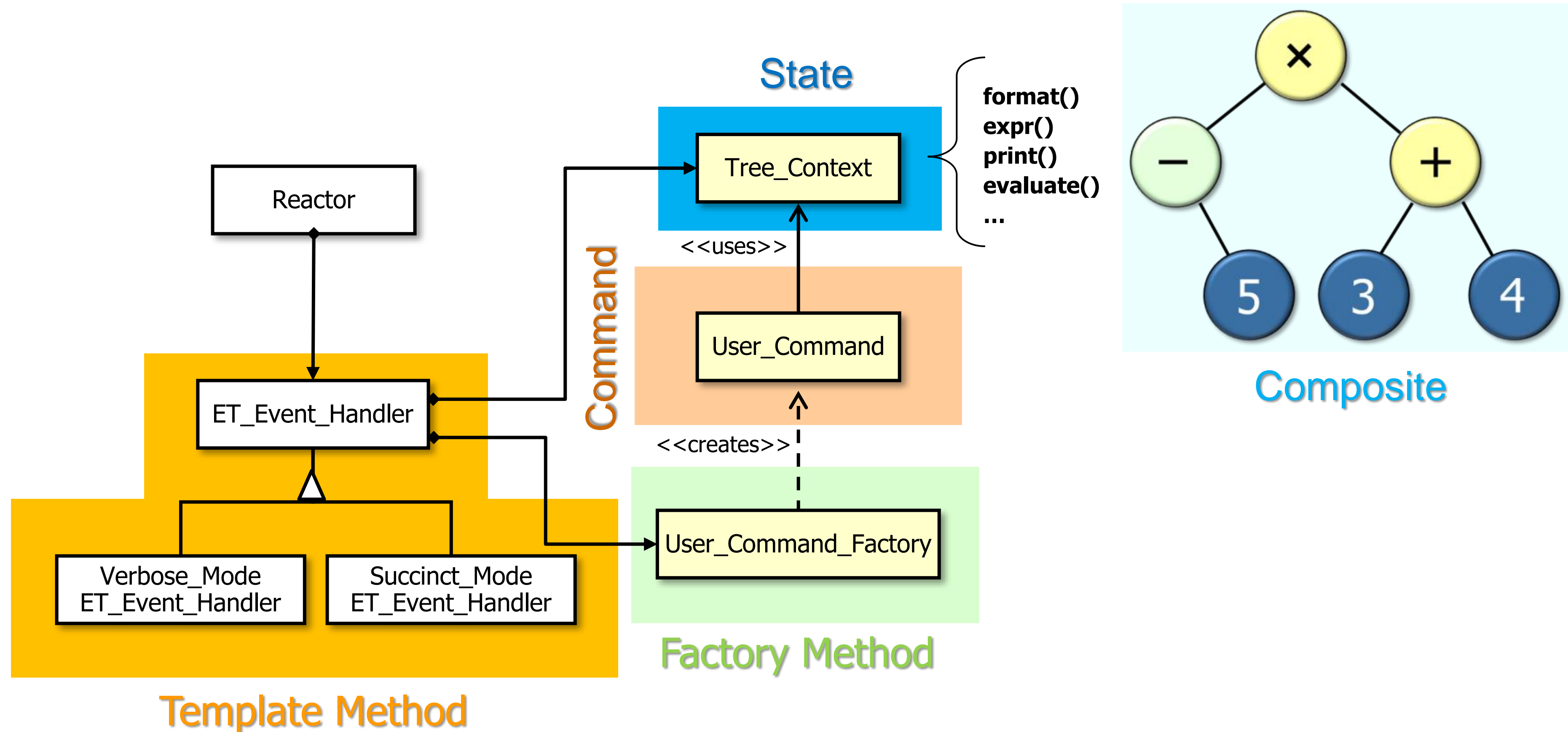


Douglas C. Schmidt

Motivating the Need for the Template Method Pattern in the Expression Tree App

A Pattern for Encapsulating Algorithm Variability

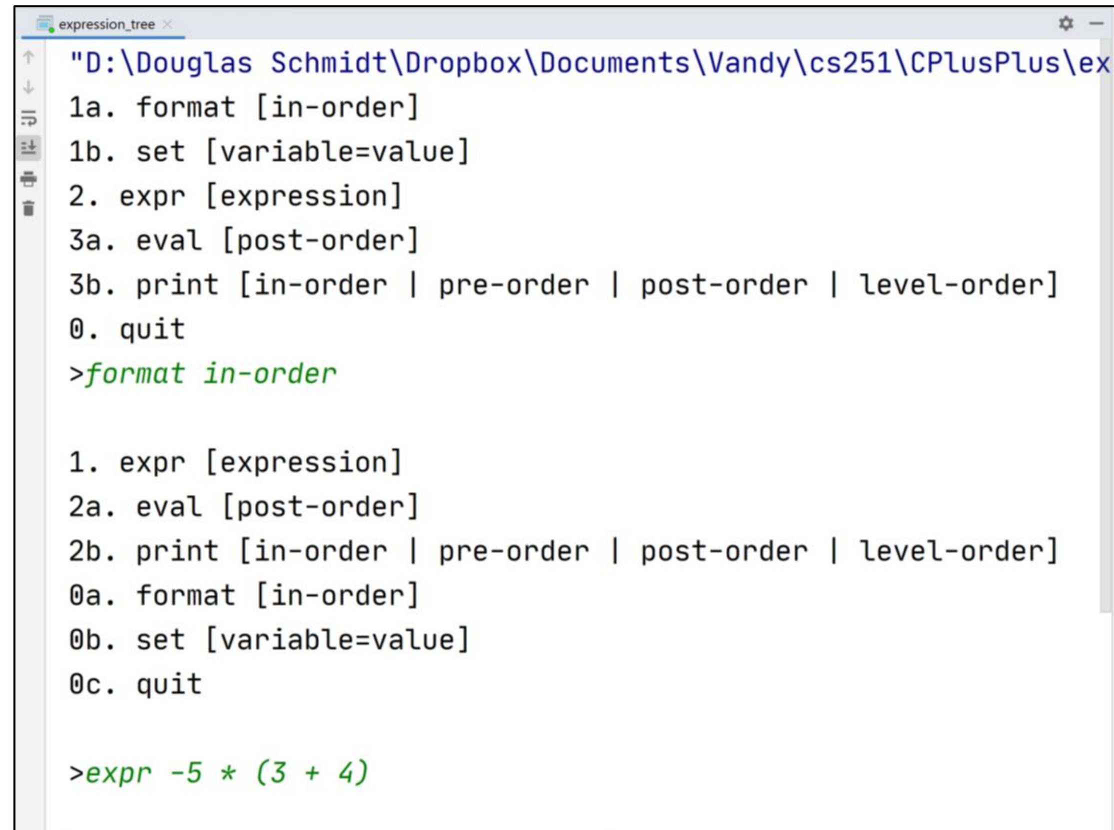
***Purpose**: Factor out common code to support multiple operating modes (succinct vs. verbose).*



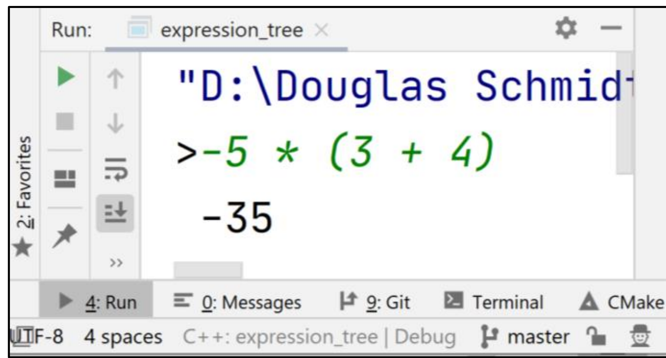
Template Method supports controlled variability of steps in an algorithm.

Context: OO Expression Tree Processing App

- This app has two primary operating modes: *verbose* & *succinct*.



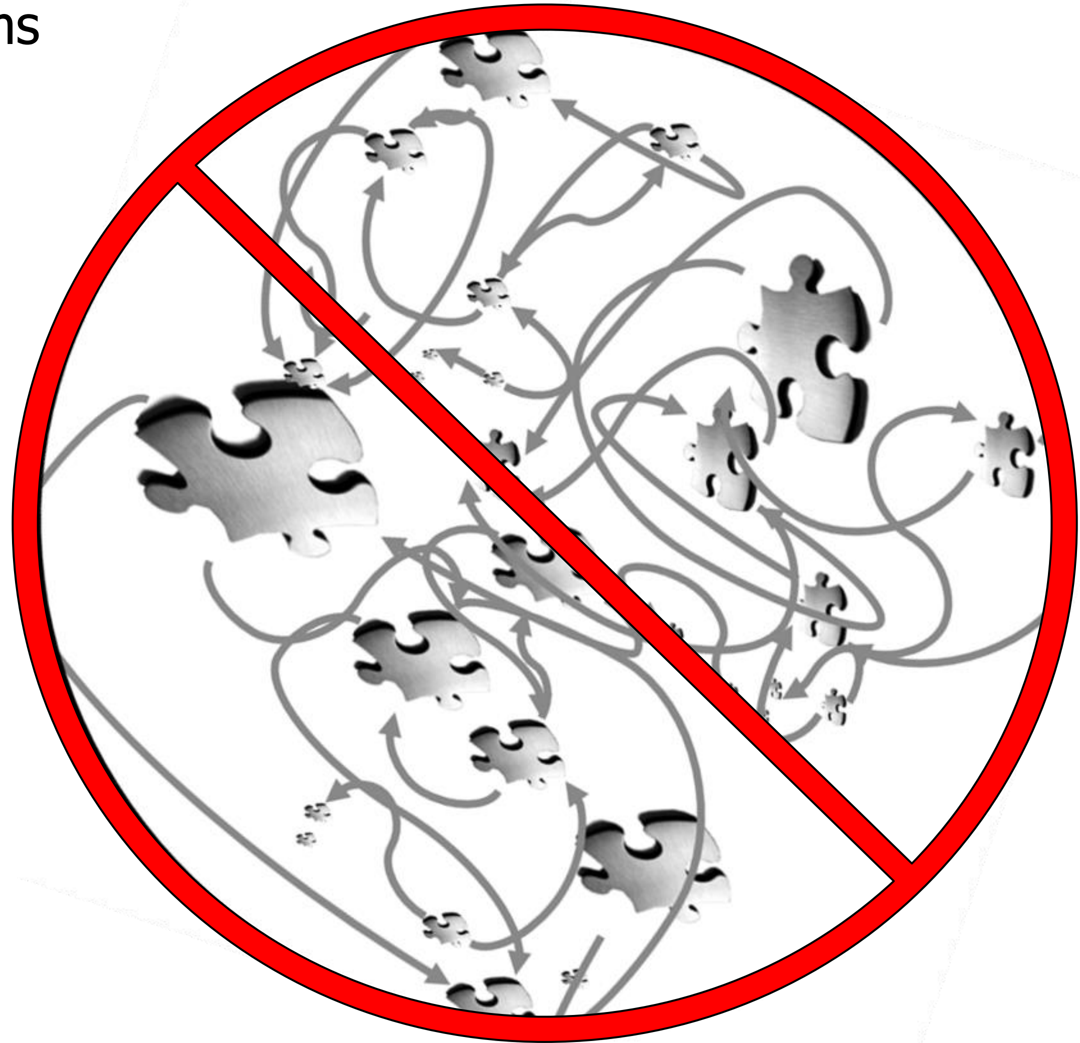
Succinct mode



Verbose mode

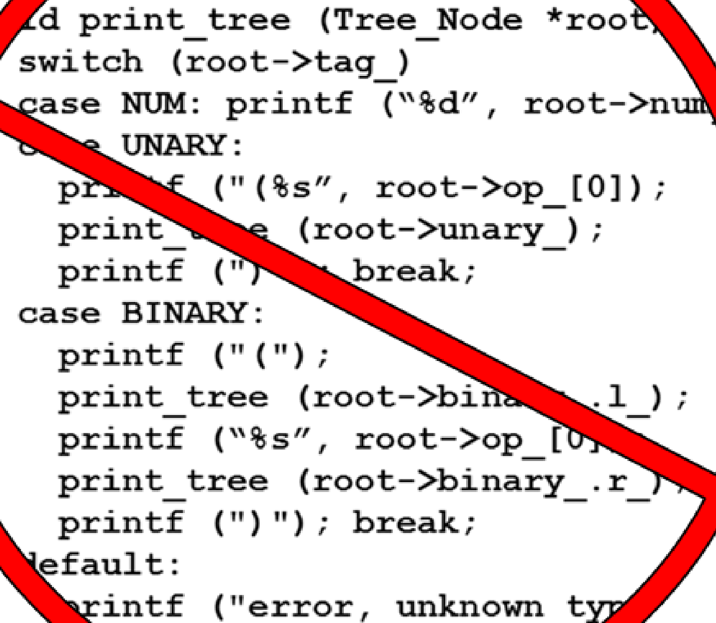
Problem: Non-Extensible Operating Modes

- Structuring the program in terms of the two operating modes' algorithms is problematic.



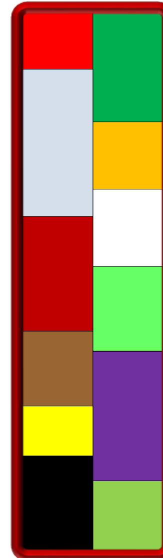
Problem: Non-Extensible Operating Modes

- 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



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"

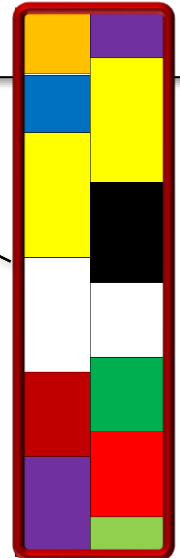


```
expression_tree
"D:\Douglas Schmidt\Dropbox\Documents\Vandy\cs251\CPlusPlus\ex
1a. format [in-order]
1b. set [variable=value]
2. expr [expression]
3a. eval [post-order]
3b. print [in-order | pre-order | post-order | level-order]
0. quit
>format in-order

1. expr [expression]
2a. eval [post-order]
2b. print [in-order | pre-order | post-order | level-order]
0a. format [in-order]
0b. set [variable=value]
0c. quit

>expr -5 * (3 + 4)
```

Silo'ing (non-reusable) code

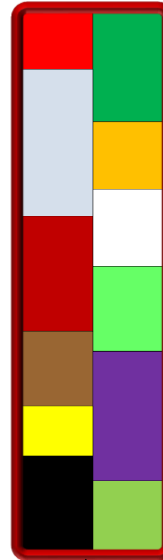
```
Run: expression_tree
"D:\Douglas Schmidt
>-5 * (3 + 4)
 -35
```

4: Run   0: Messages   9: Git   Terminal   CMake
UTF-8   4 spaces   C++: expression_tree | Debug   master



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

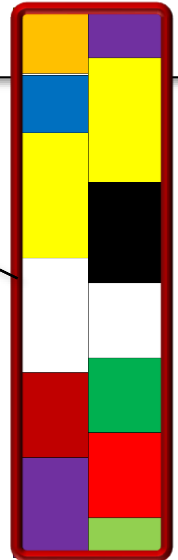


```
expression_tree
"D:\Douglas Schmidt\Dropbox\Documents\Vandy\cs251\CPlusPlus\ex
1a. format [in-order]
1b. set [variable=value]
2. expr [expression]
3a. eval [post-order]
3b. print [in-order | pre-order | post-order | level-order]
0. quit
>format in-order

1. expr [expression]
2a. eval [post-order]
2b. print [in-order | pre-order | post-order | level-order]
0a. format [in-order]
0b. set [variable=value]
0c. quit

>expr -5 * (3 + 4)
```

Stovepiped (non-reusable) code

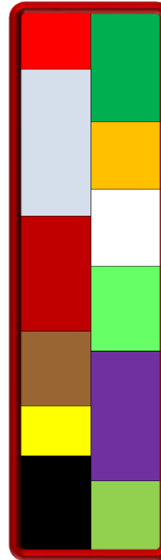
```
Run: expression_tree
"D:\Douglas Schmid
>-5 * (3 + 4)
 -35
```

4: Run   0: Messages   9: Git   Terminal   CMake
UTF-8   4 spaces   C++: expression_tree | Debug   master

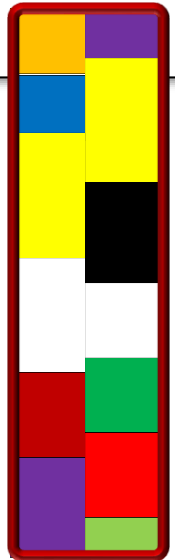


Problem: Non-Extensible Operating Modes

- 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



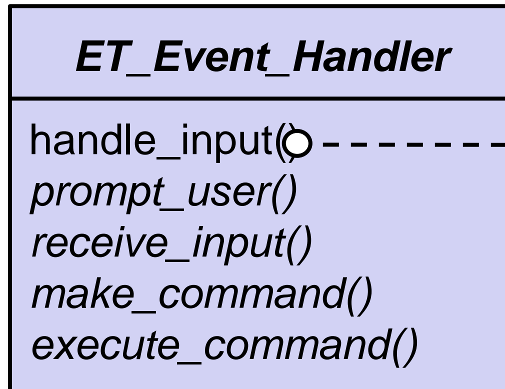
```
expression_tree
"D:\Douglas Schmidt\Dropbox\Documents\Vandy\cs251\CPlusPlus\ex
1a. format [in-order]
1b. set [variable=value]
2. expr [expression]
3a. eval [post-order]
3b. print [in-order | pre-order | post-order | level-order]
0. quit
>format in-order

1. expr [expression]
2a. eval [post-order]
2b. print [in-order | pre-order | post-order | level-order]
0a. format [in-order]
0b. set [variable=value]
0c. quit
```



See en.wikipedia.org/wiki/Don't_repeat_yourself

Solution: Encapsulate Algorithm Variability

- Implement algorithm once in super class

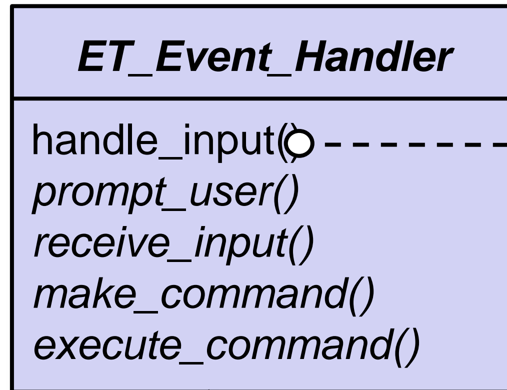



```
void handle_input(){
  prompt_user();
  string input = receive_input();
  User_Command command =
                    make_command(input);
  execute_command(command))
}
```

handle_input() is a template method.

Solution: Encapsulate Algorithm Variability

- Implement algorithm once in super class



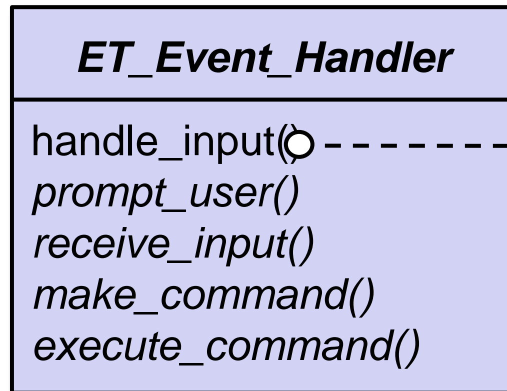

```
void handle_input(){
  prompt_user();
  string input = receive_input();
  User_Command command =
                      make_command(input);
  execute_command(command))
}
```

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.




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

Verbose_Mode ET_Event_Handler

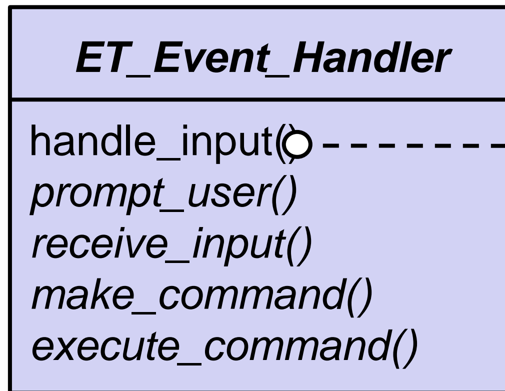
prompt_user()
make_command()

Succinct_Mode ET_Event_Handler

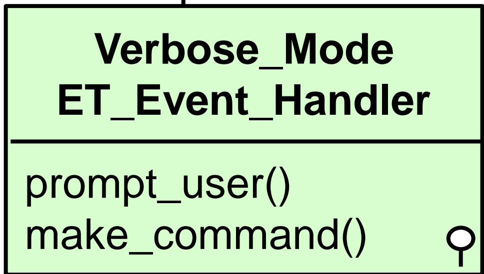
prompt_user()
make_command()

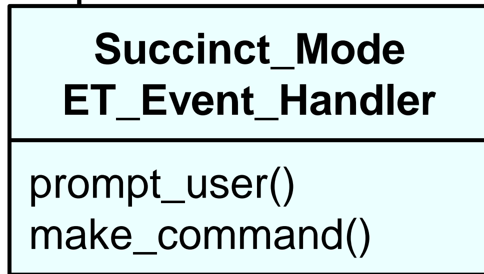
Solution: Encapsulate Algorithm Variability

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




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





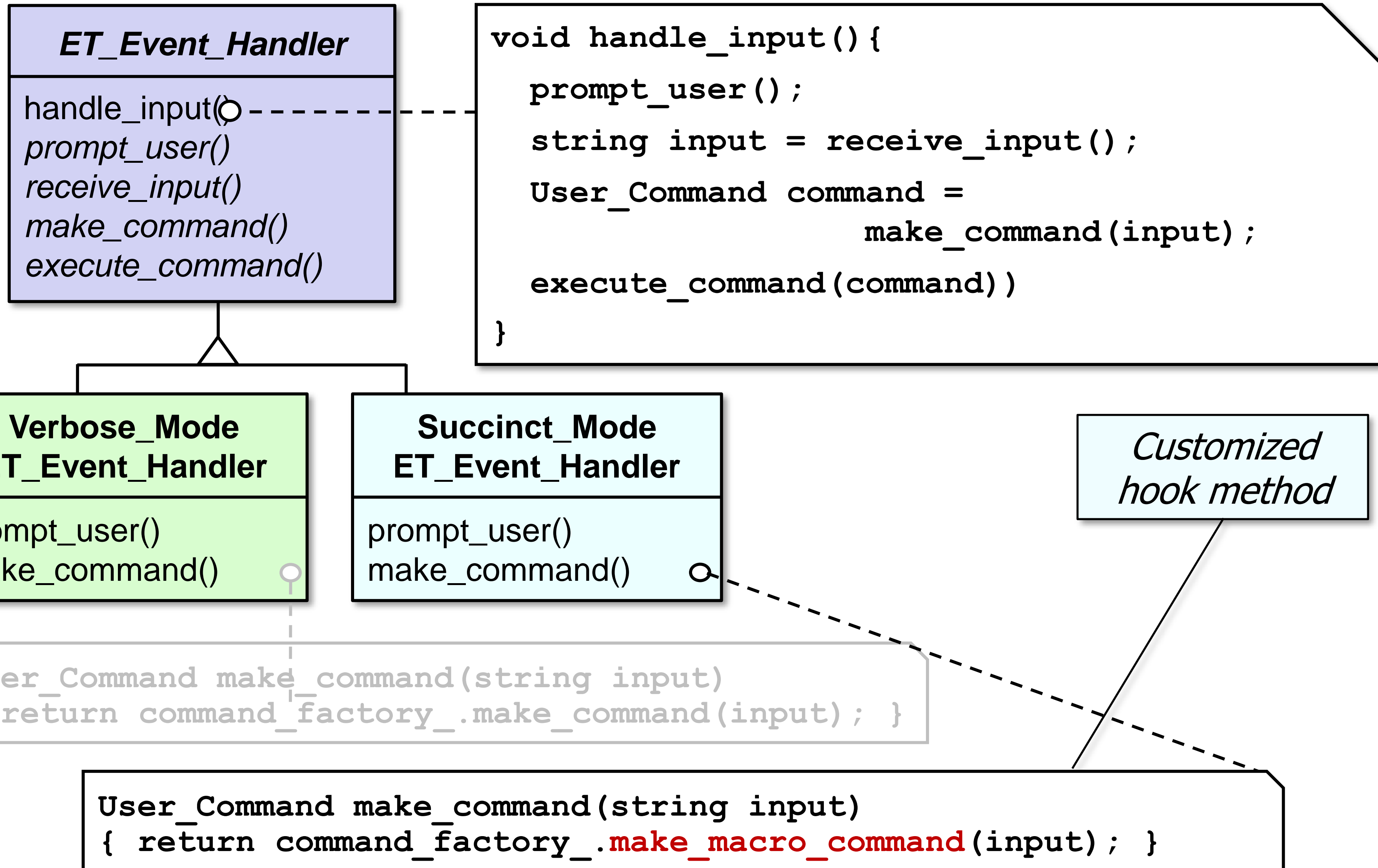
Customized hook method

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

See earlier lesson on "*The Command Pattern: Implementation in C++*."

Solution: Encapsulate Algorithm Variability

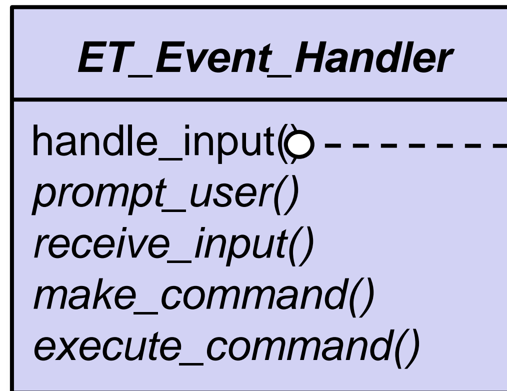
- Implement algorithm once in super class & let subclasses define variants.



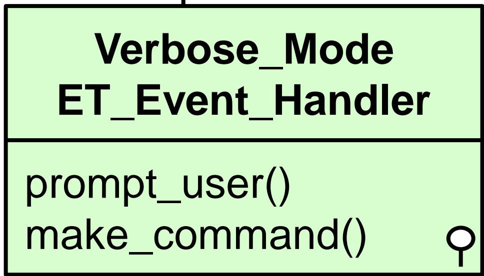
See earlier lesson on "*The Command Pattern: Implementation in C++*."

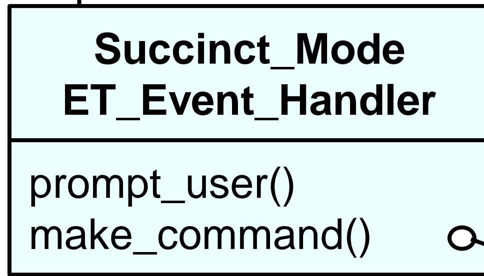
Solution: Encapsulate Algorithm Variability

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




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





```
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); }
```

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

```
                 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

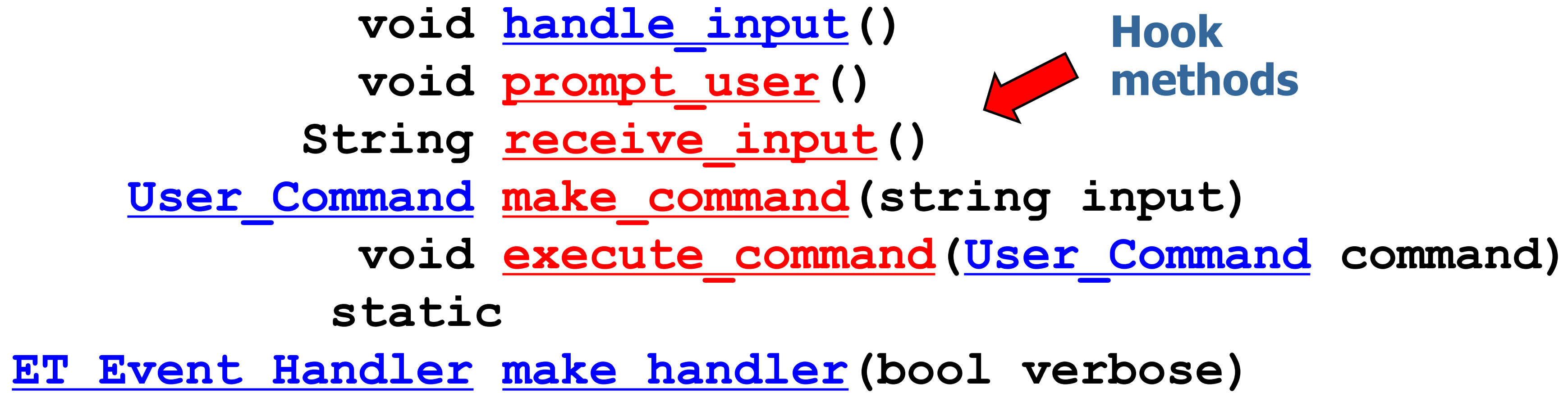
Class methods

```
                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

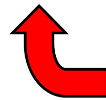


ET_Event_Handler Class Overview

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

Class methods

```
                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

```
                 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.