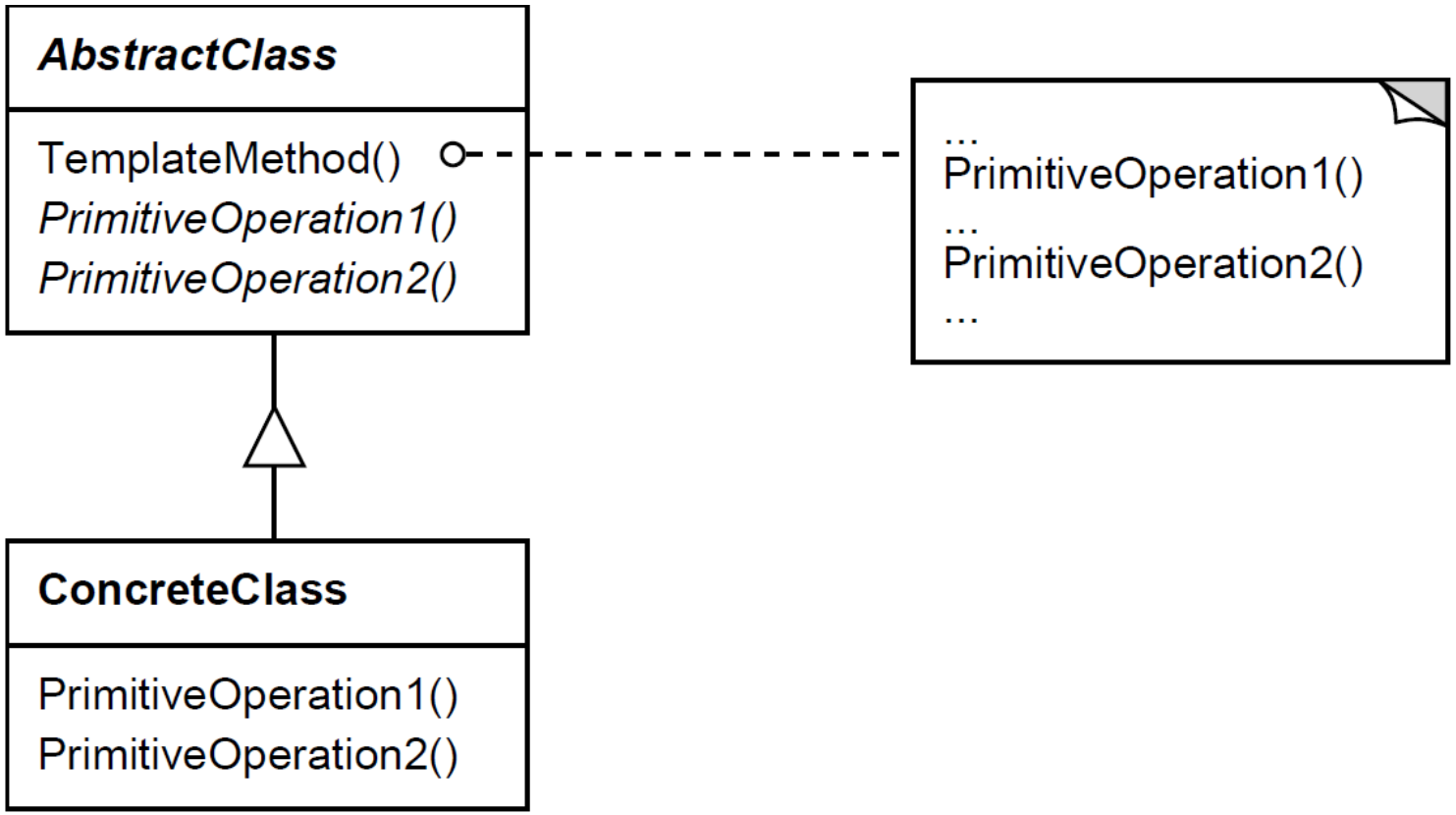
The Template Method Pattern

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.



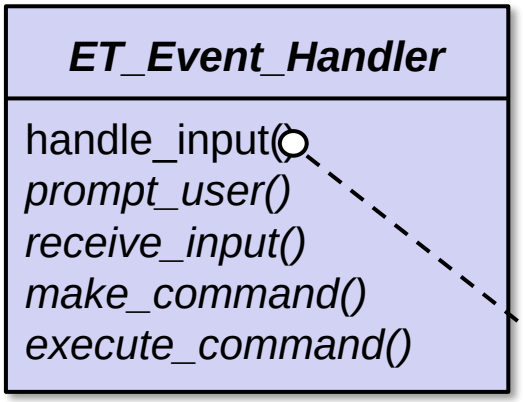
Douglas C. Schmidt

Structure & Functionality of the Template Method Pattern

Template Method — GoF Class Behavioral

Intent

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

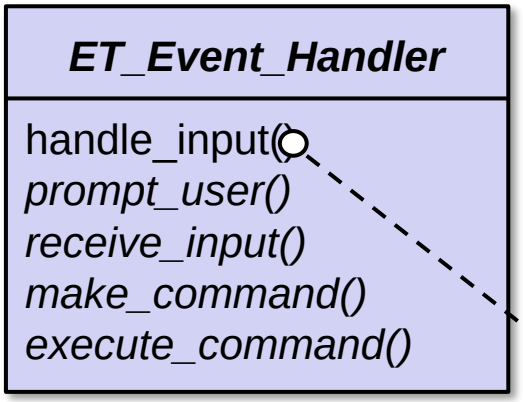



```
void handle_input(){
  prompt_user();
  string input = receive_input();
  User_Command command =
                    make_command(input);
  execute_command(command))
}
```

See en.wikipedia.org/wiki/Template_method_pattern

Applicability

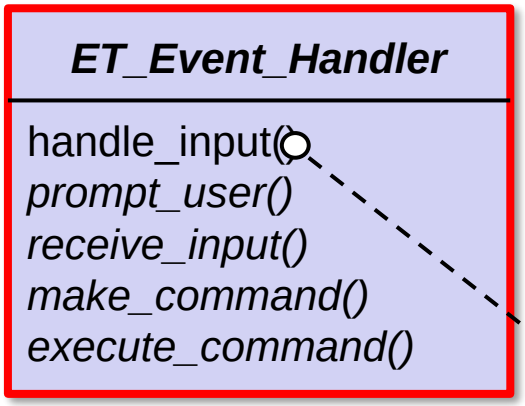
- Implement invariant aspects of an algorithm *once* & let subclasses define variant parts




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

Applicability

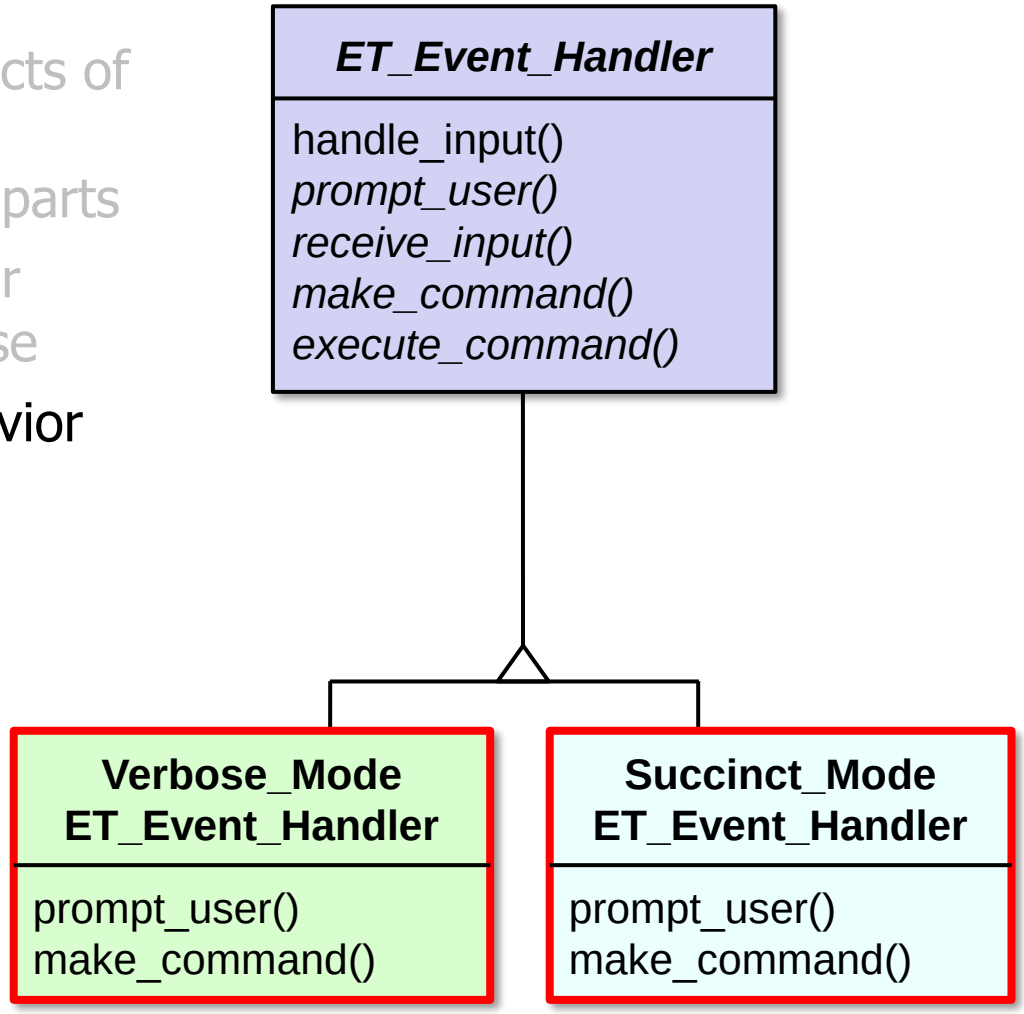
- Implement invariant aspects of an algorithm *once* & let subclasses define variant parts
- Localize common behavior in a class to enhance reuse



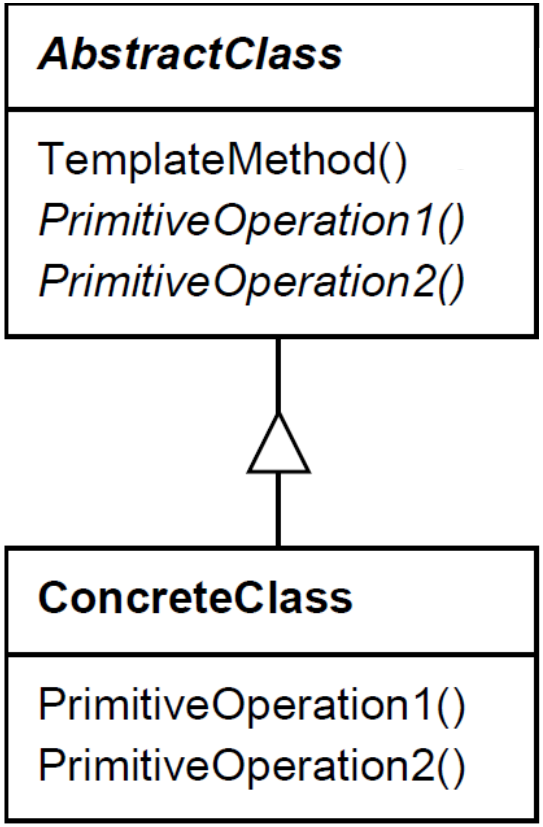

```
void handle_input(){
  prompt_user();
  string input = receive_input();
  User_Command command =
                    make_command(input);
  execute_command(command))
}
```

Applicability

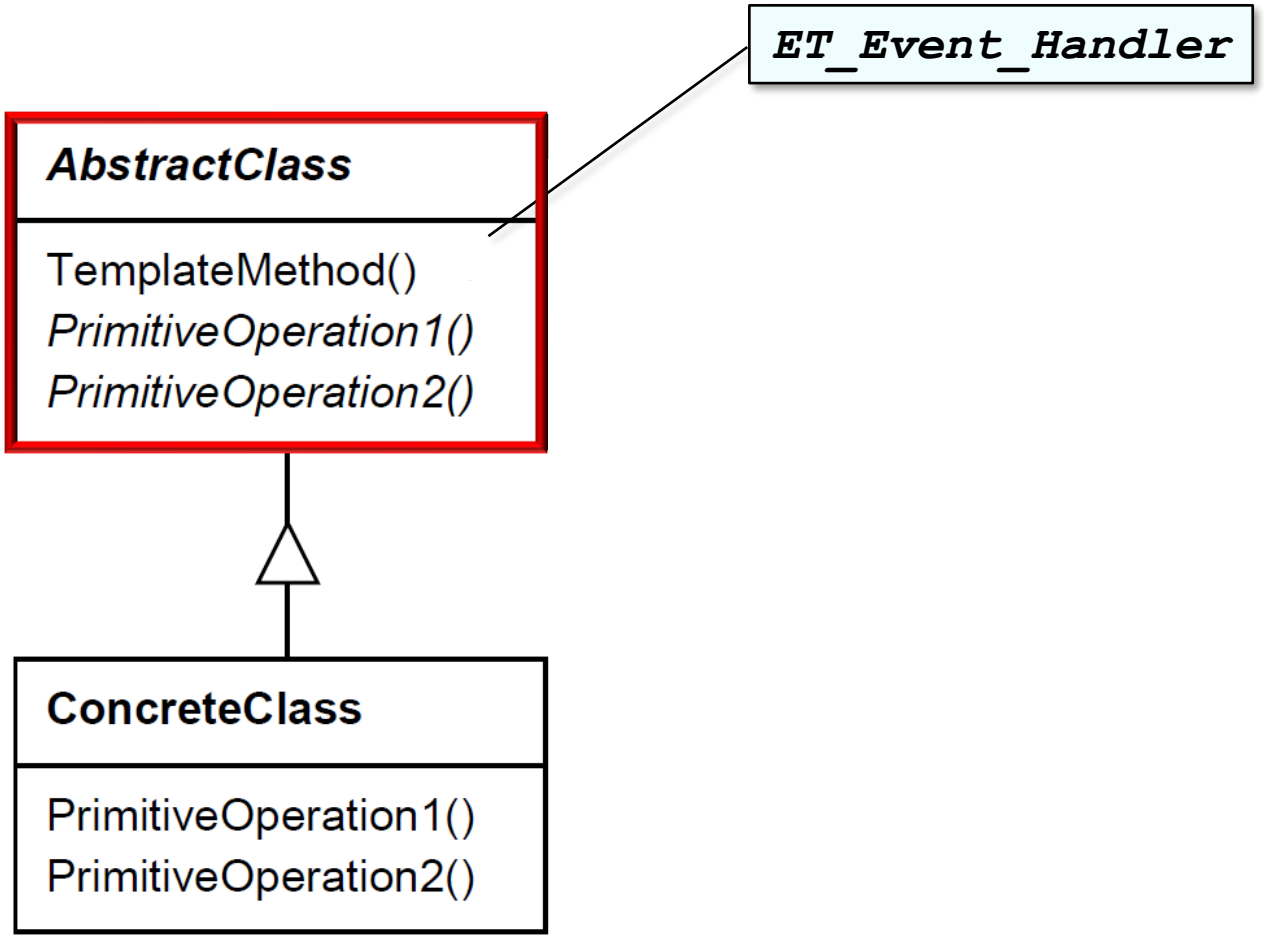
- Implement invariant aspects of an algorithm *once* & let subclasses define variant parts
- Localize common behavior in a class to enhance reuse
- Handle variations in behavior via subclassing



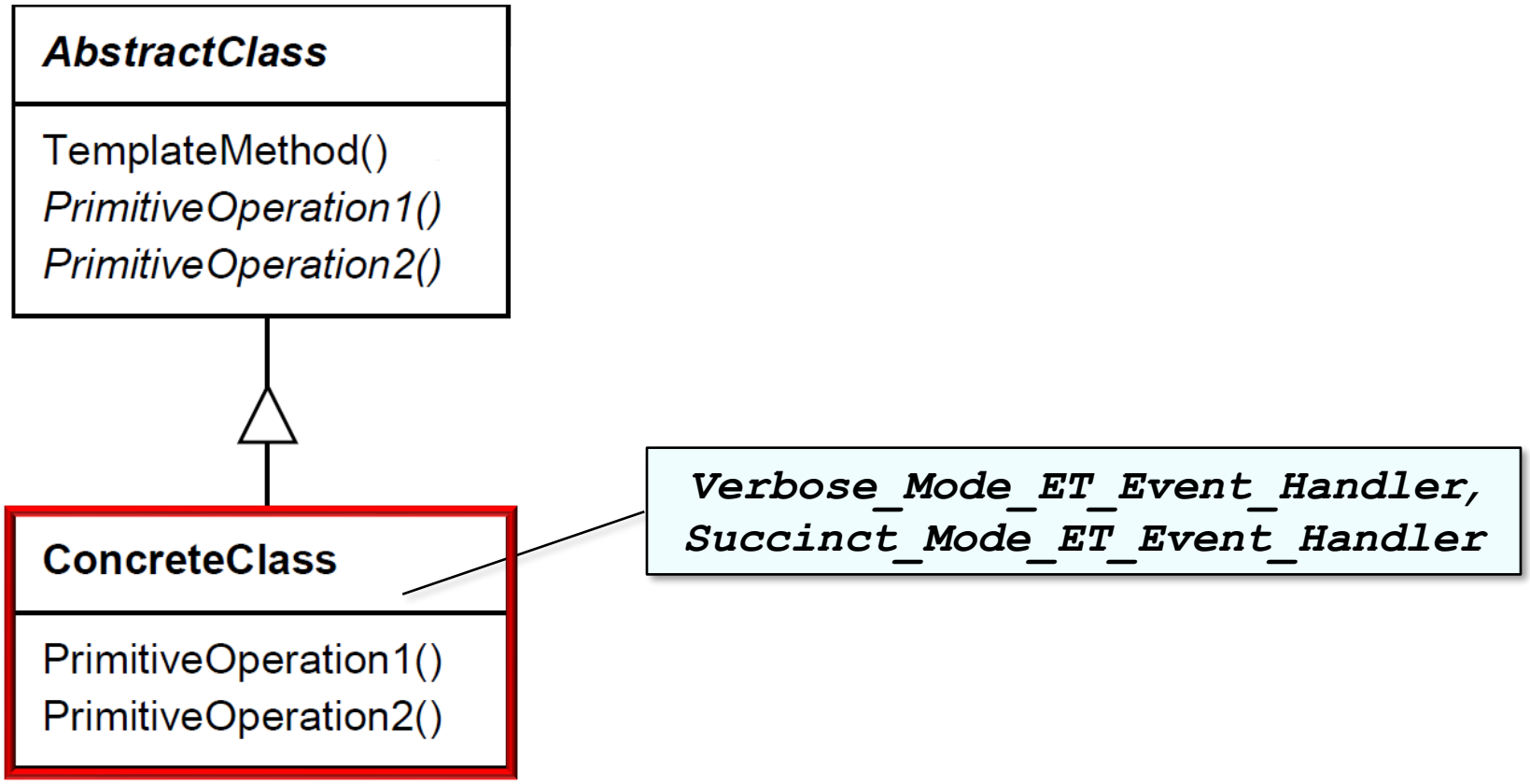
Structure & participants



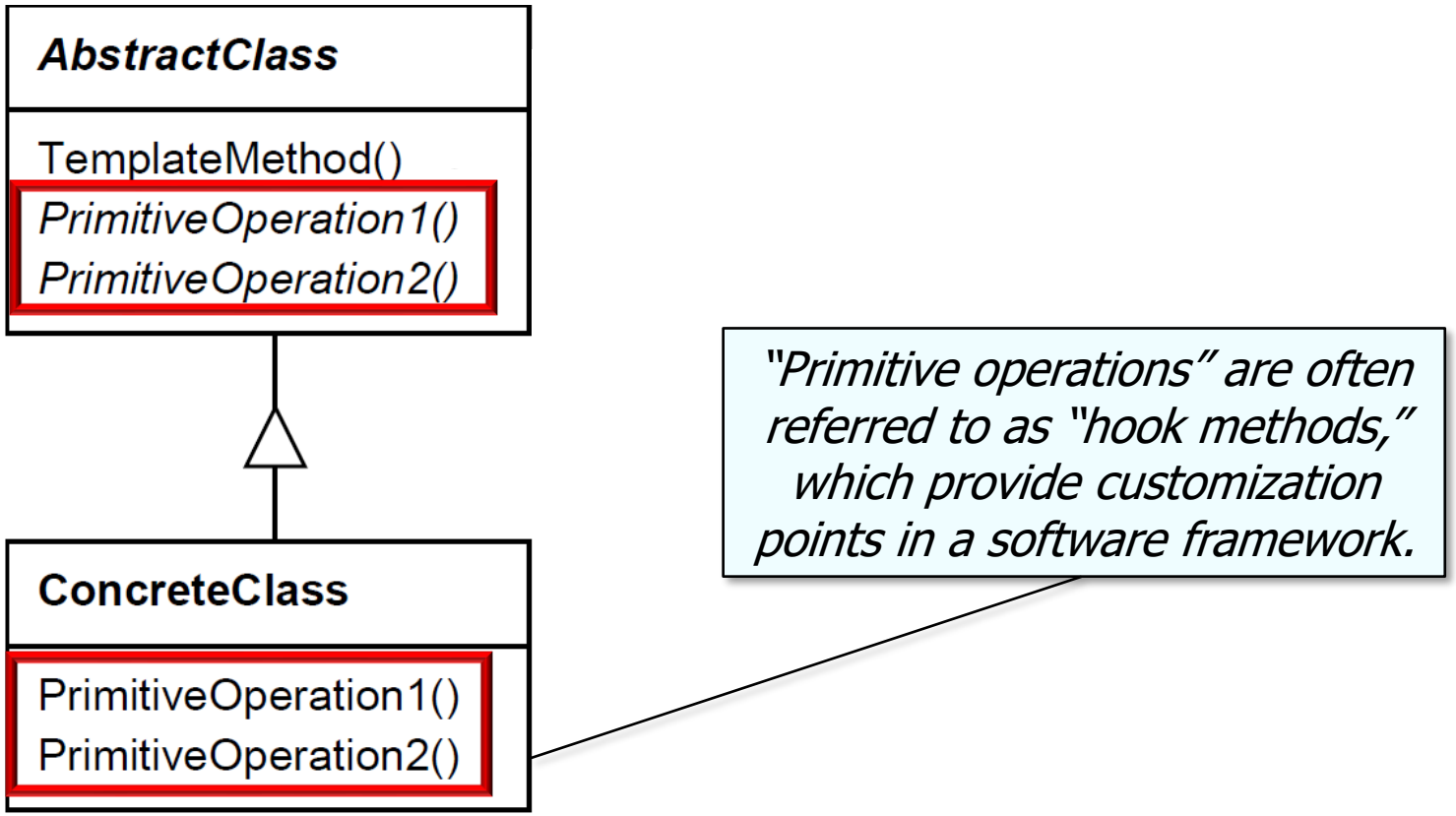
Structure & participants



Structure & participants

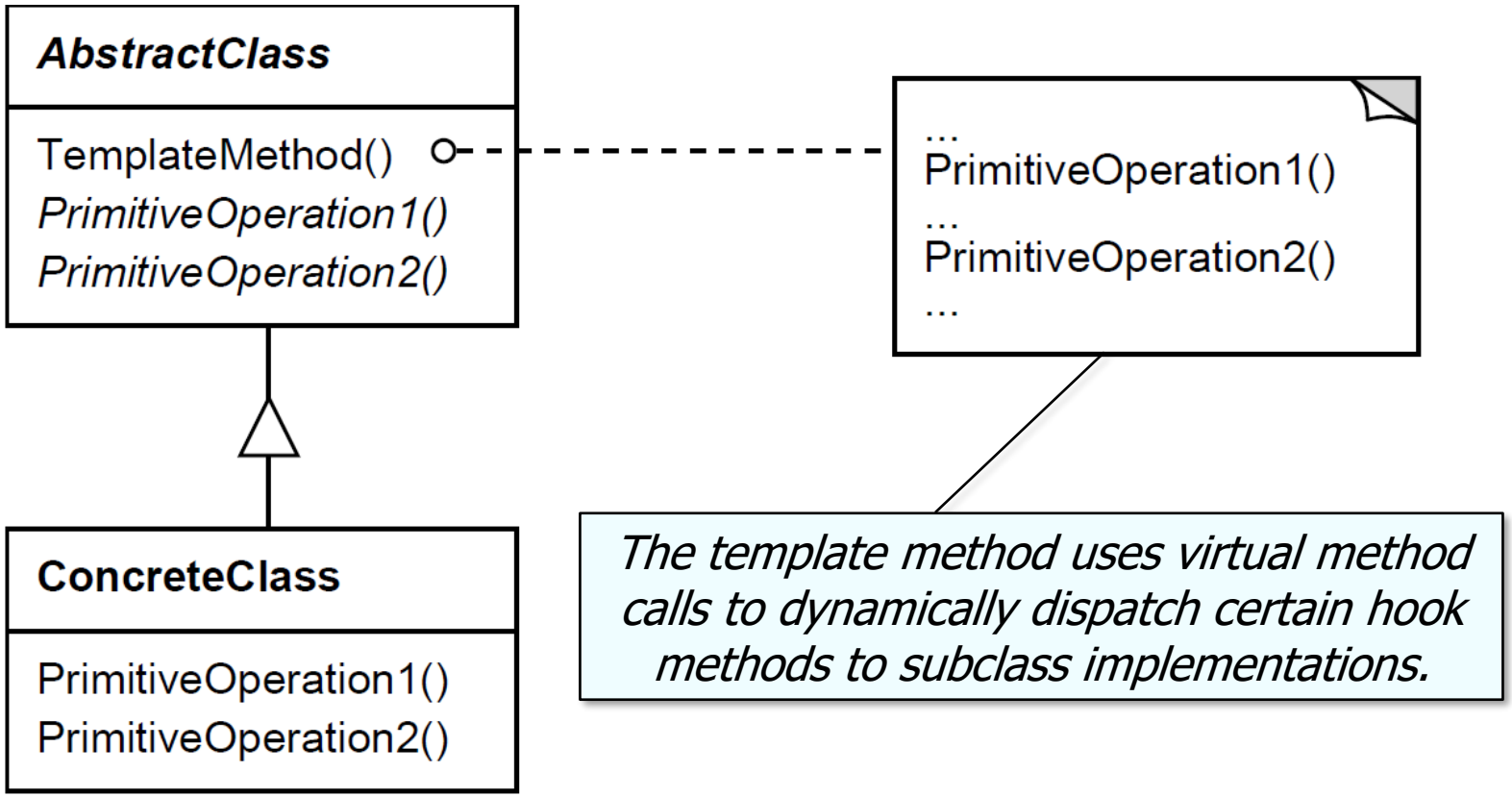


Structure & participants



See wiki.c2.com/?HookMethod

Dynamics



This pattern is "OOP 101" www.stroustrup.com/whatis.pdf

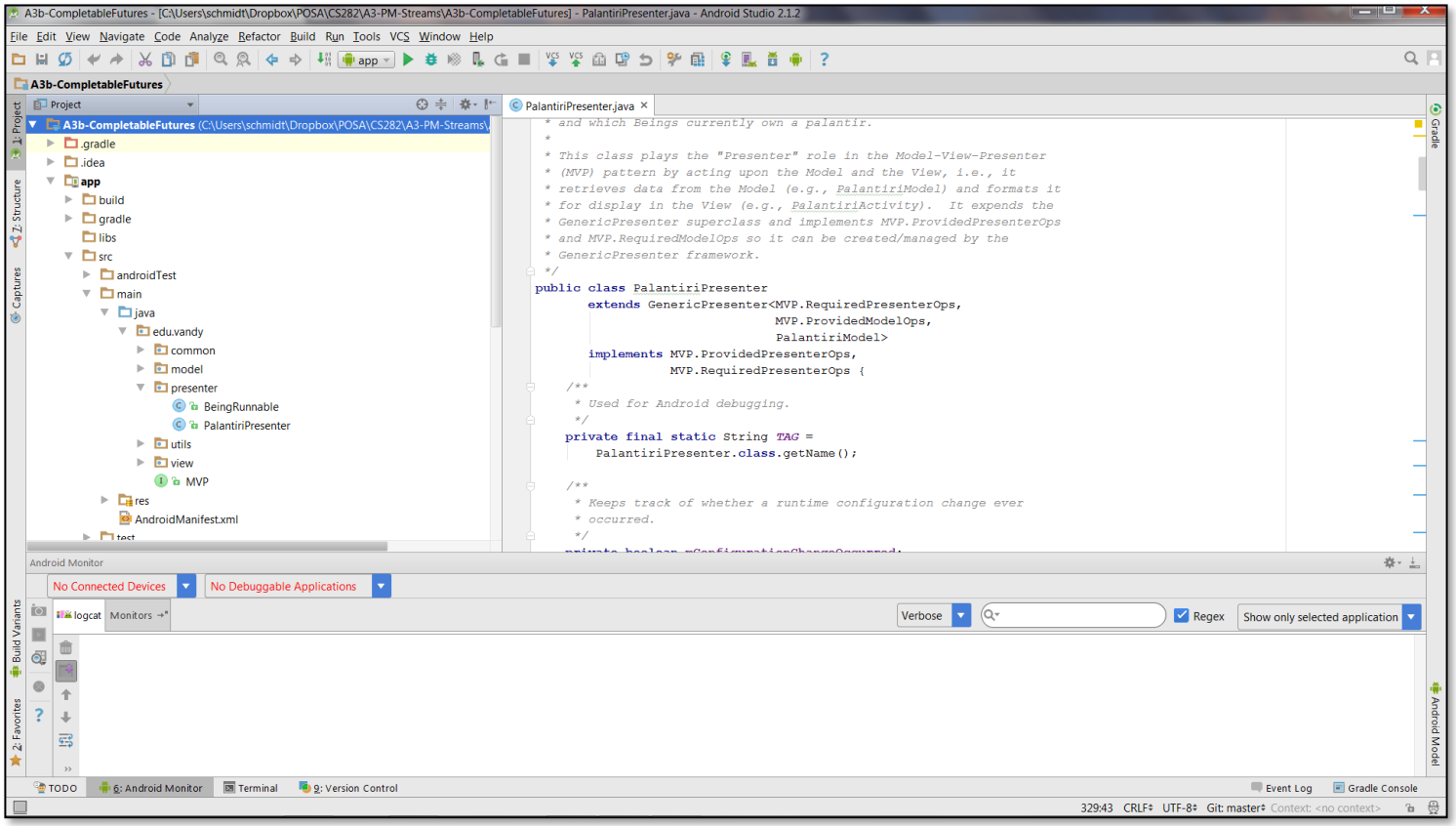
The Template Method Pattern

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.

```
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

Template Method example in C++

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

```
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);
  }
```

Hook methods

Template Method example in C++

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

```
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.

```
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);
  }
```

Specialized hook method

Template Method example in C++

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

```
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.

```
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

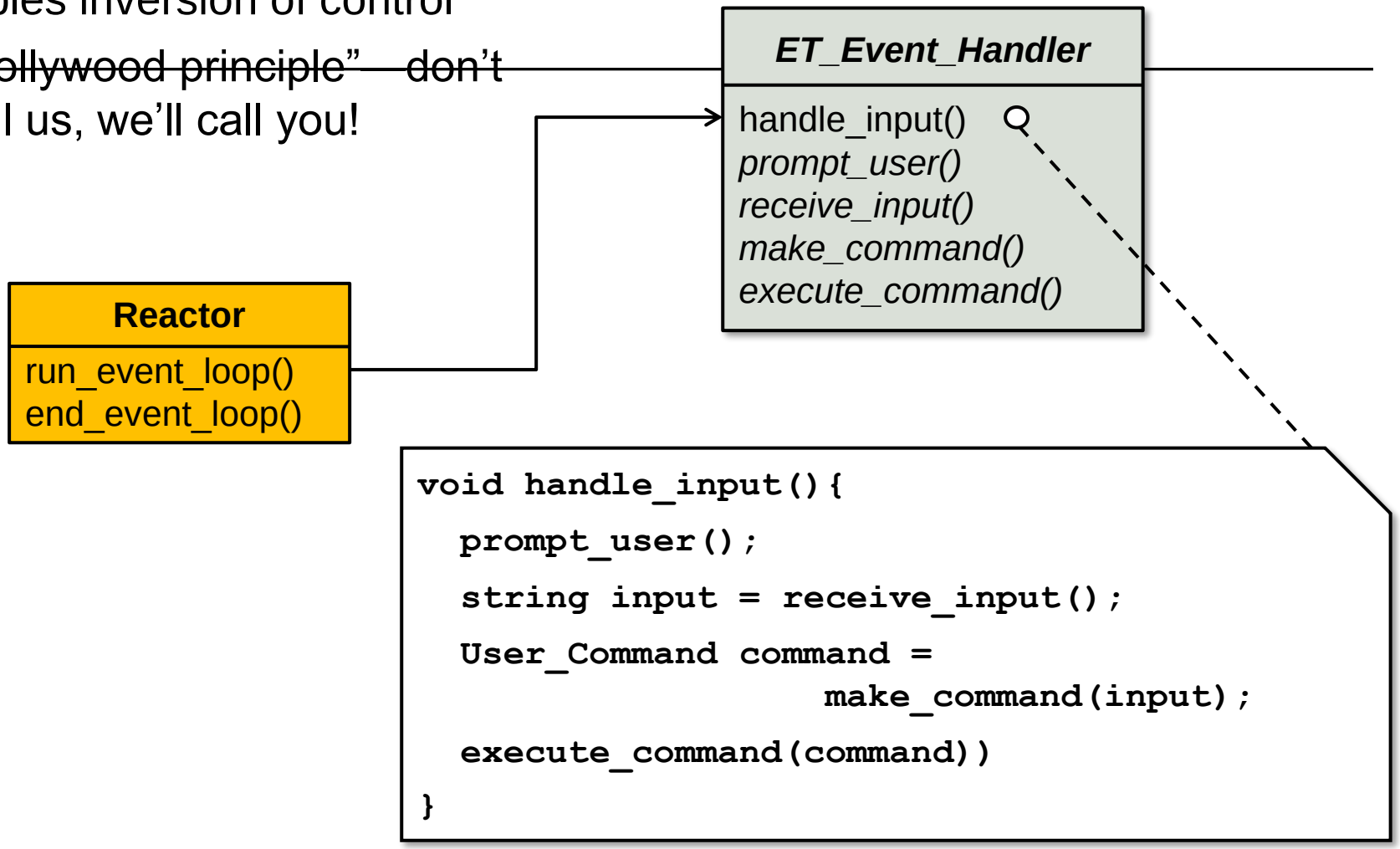
Douglas C. Schmidt

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

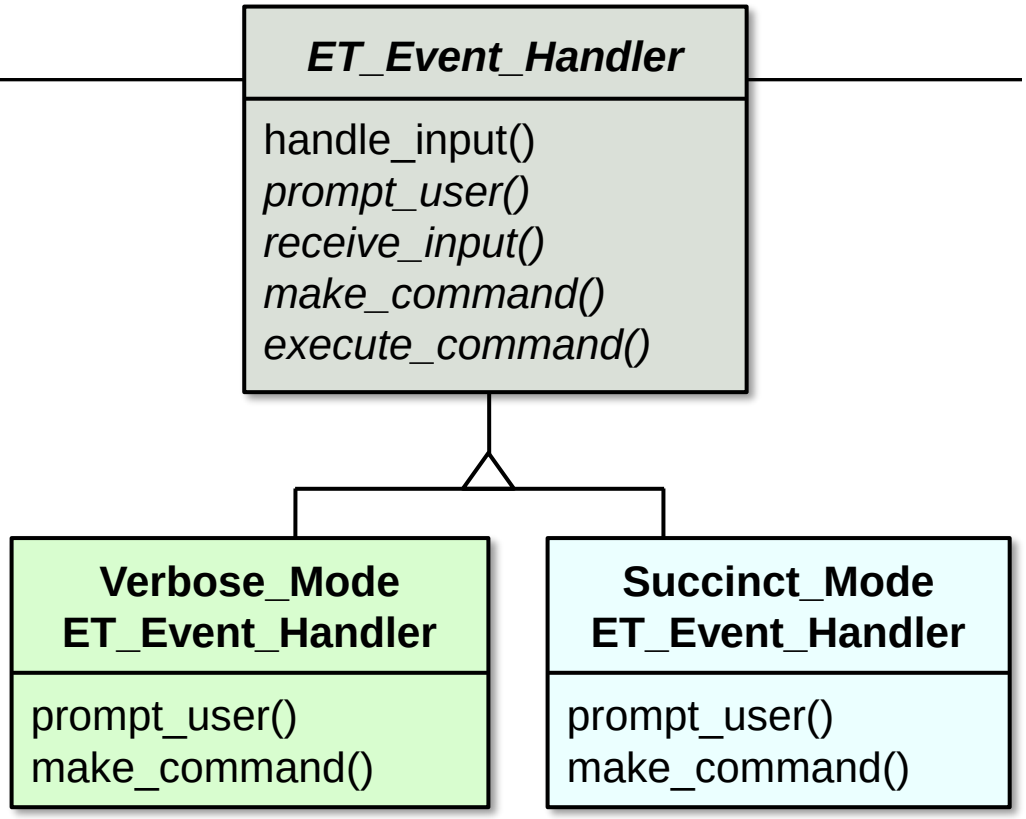
+ Enables inversion of control
  - ~~"Hollywood principle" – don't~~ call us, we'll call you!



See www.dre.vanderbilt.edu/~schmidt/Coursera/articles/hollywood-principle.txt

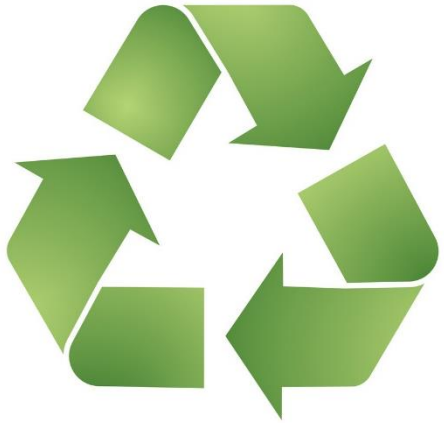
Consequences

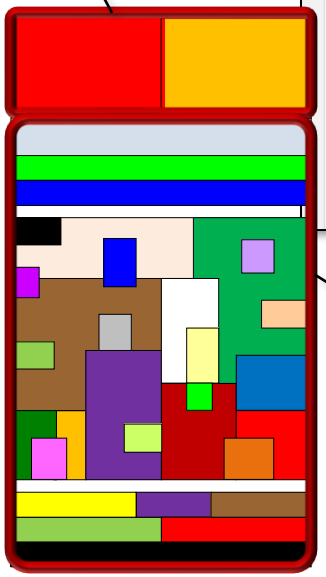
+ Overriding rules are enforced via subclassing



Consequences

+ Promotes systematic reuse by collapsing stovepipes





Variant (non-reusable) code

Common (reusable) code

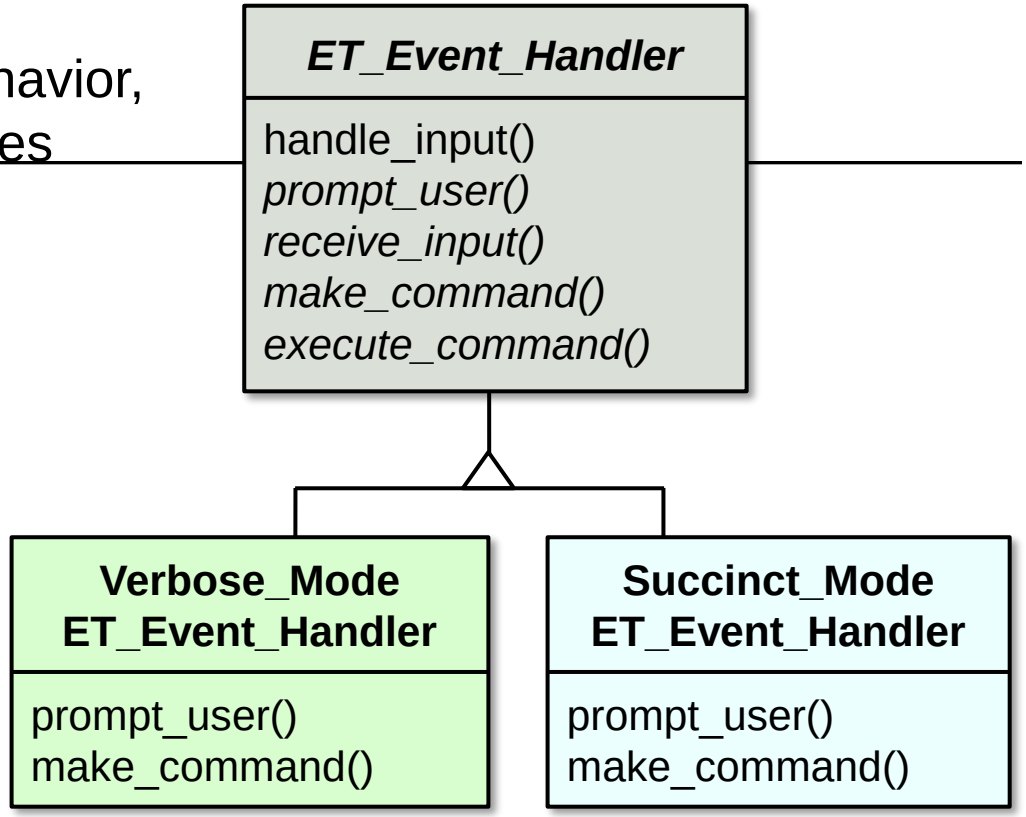
```
"D:\Douglas Schmidt\Dropbox\Documents\Vandy\cs251\CPlusPlus\ex
1a. format [in-order]
1b. set [variable=value]
2. expr [expression]
3a. eval [post-order]
3b. print [in-order | pre-order | post-order | level-order]
0. quit
>format in-order

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)
```

```
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
```

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

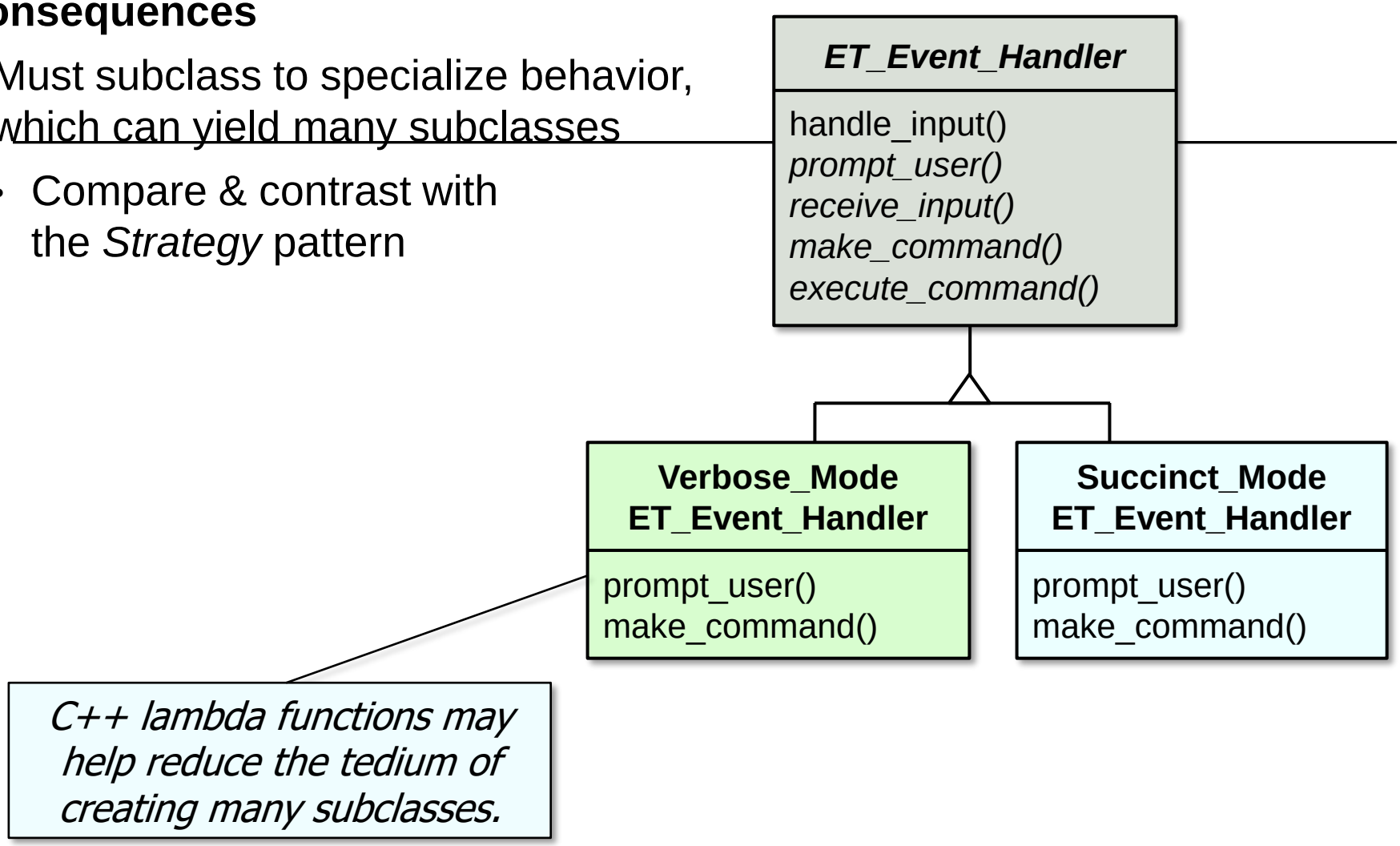
Consequences

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



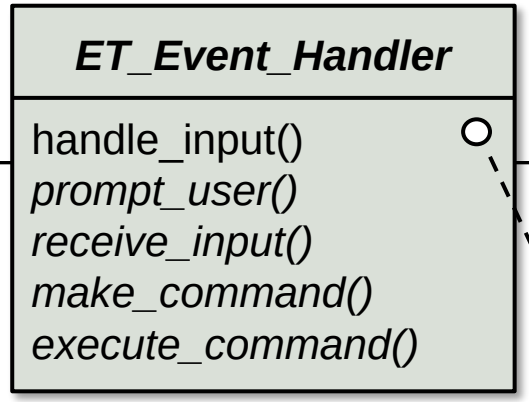
Consequences

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



Implementation considerations

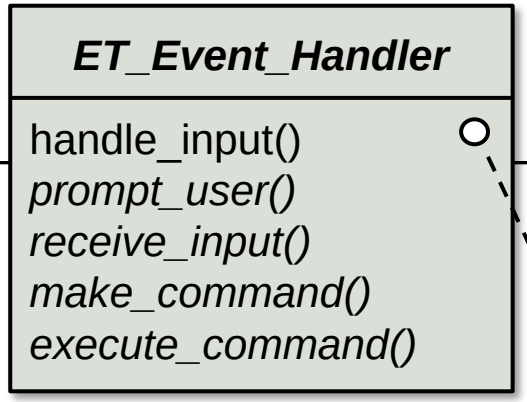
- Virtual vs. non-virtual (final) template method
  - Depends on whether the algorithm embodied by the template method itself may need to change




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

Implementation considerations

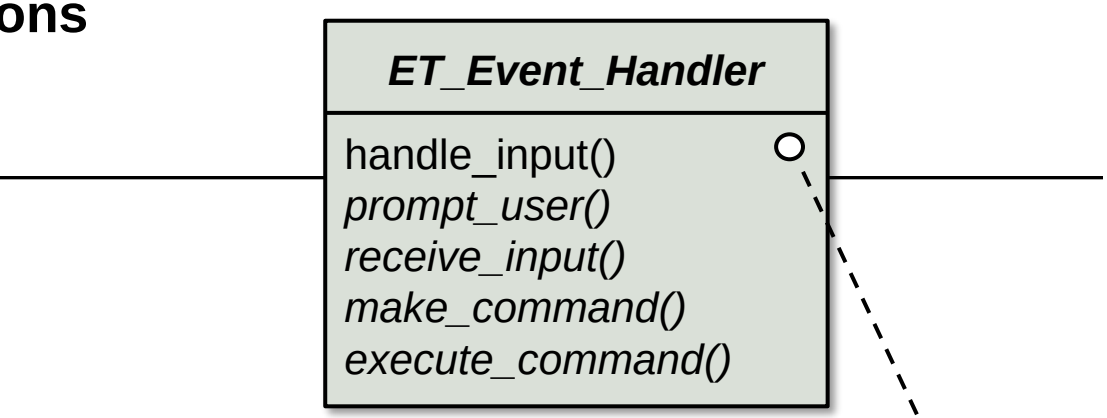
- Few vs. many primitive operations (hook methods)
  - e.g., how much variability's needed in the template method's algorithm?




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

Implementation considerations

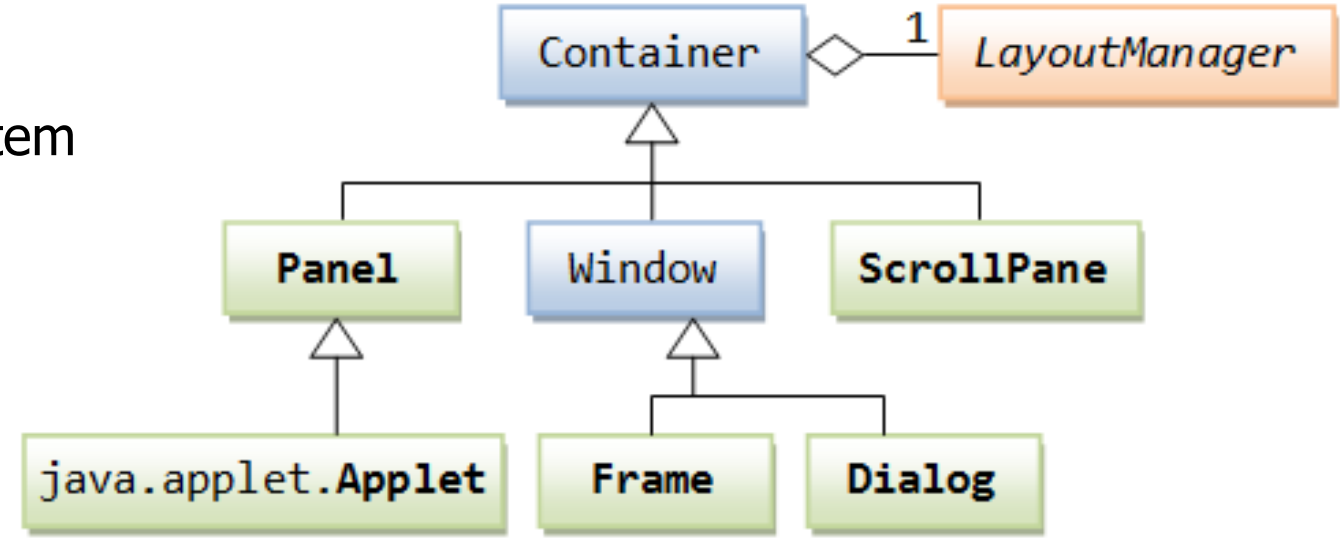
- Naming conventions
  - ~~e.g., do*() vs. make*()~~ vs. on*() prefixes




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

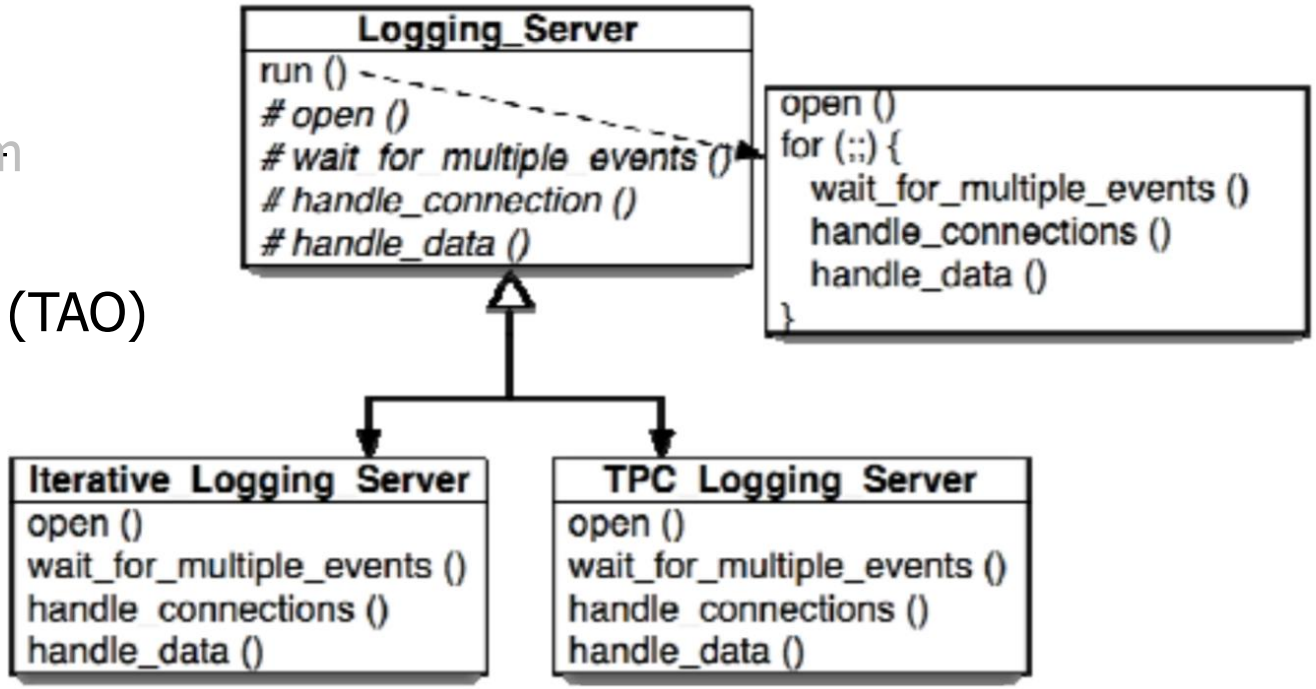
Known uses

- InterViews Kits
- ~~ET++ WindowSystem~~
- AWT Toolkit



Known uses

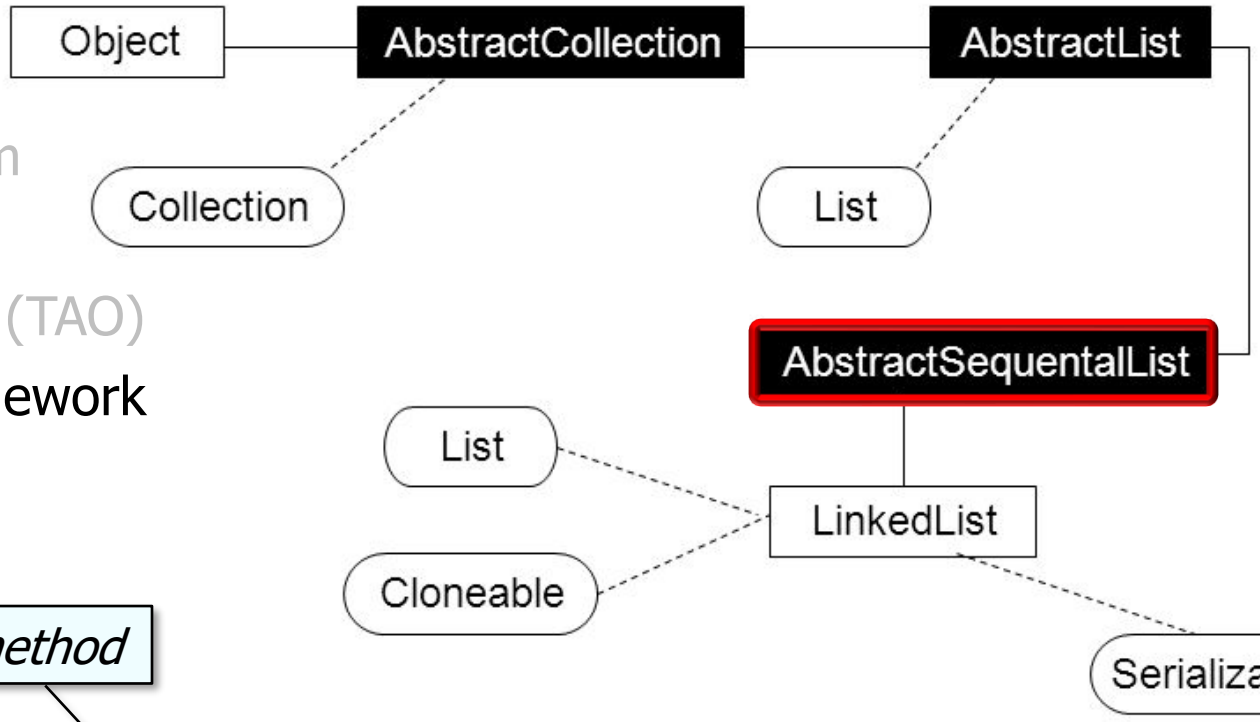
- InterViews Kits
- ET++ WindowSystem
- AWT Toolkit
- ACE & The ACE ORB (TAO)



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



Template method

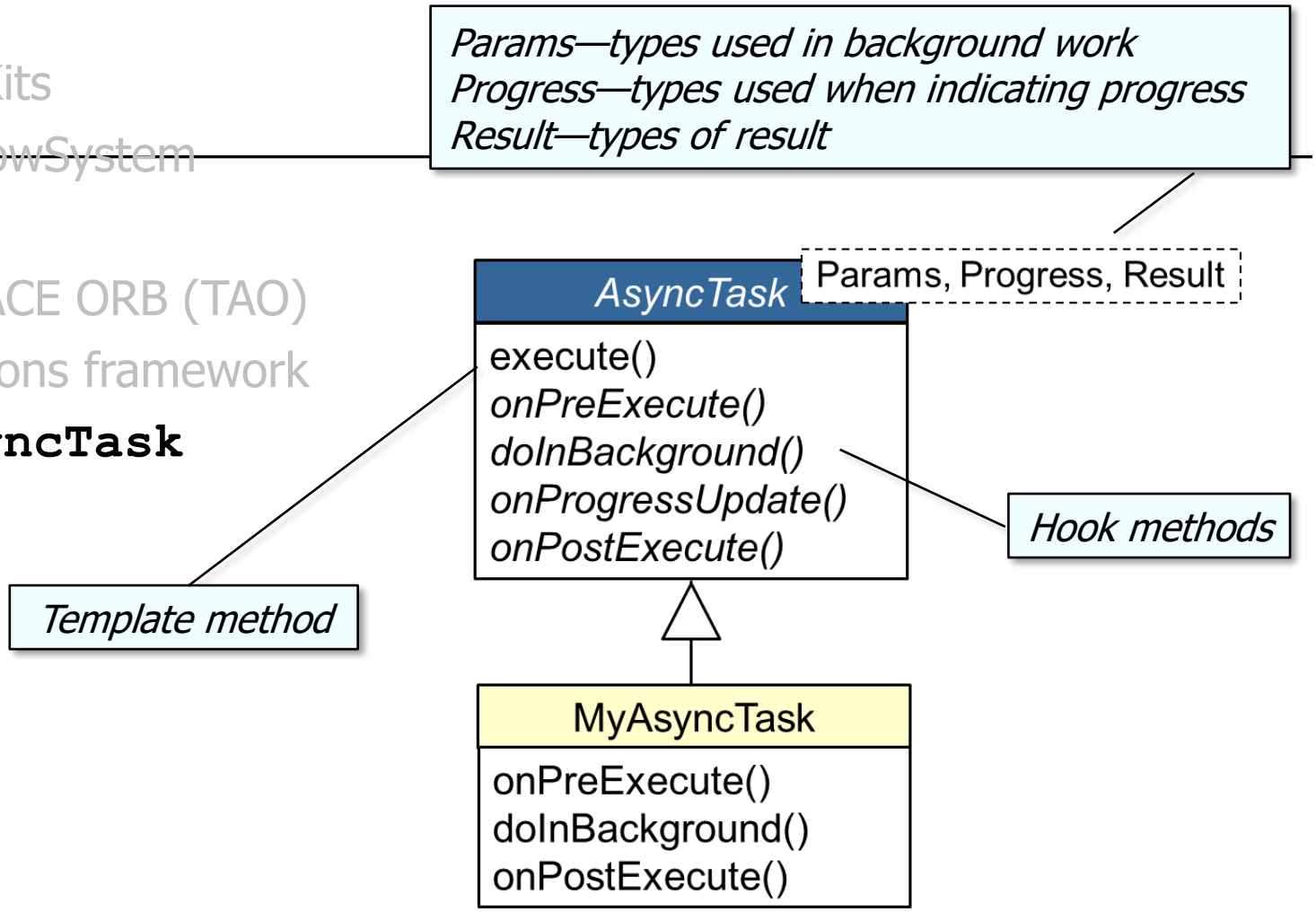
```
public E set(int index, E element) {
    try {
      ListIterator<E> e = listIterator(index);
      ...
```

Hook method

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



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

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

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.

We selected *Template Method* for our case study since it's a simple use case.

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

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

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.