A Case Study of “Gang of Four” (GoF) Patterns : Part 10

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

Professor of Computer Science
Institute for Software Integrated Systems
Vanderbilt University
Nashville, Tennessee, USA
Topics Covered in this Part of the Module

- Describe the object-oriented (OO) expression tree case study
- Evaluate the limitations with algorithmic design techniques
- Present an OO design for the expression tree processing app
- Summarize the patterns in the expression tree design
- Explore patterns for
  - Tree structure & access
  - Tree creation
  - Tree traversal
  - Commands & factories
  - Command ordering protocols
  - Application structure
Overview of Application Structure Patterns

**Purpose:** Structure the overall control flow of the event-driven expression tree processing app

These patterns simplify processing of events & user options.
Problem: Structuring Application Event Flow

Goals

- Decouple expression tree processing app from the context in which it runs
  - e.g., command-line vs. various GUI environments

```plaintext
% tree-traversal -v
format [in-order]
expr [expression]
print [in-order|pre-order|post-order|level-order]
eval [post-order]
quit
> format in-order
> expr 1+4*3/2
> eval post-order
7
> quit
```
Problem: Structuring Application Event Flow

Goals

• Decouple expression tree processing app from the context in which it runs
  • e.g., command-line vs. various GUI environments

Constraints/forces

• Don’t hard-code control flow into app logic
• Don’t hard-code event processing logic into app structure
Solution: Separate Event Handling Concerns

- Create a reactor to detect input on various sources of events

<table>
<thead>
<tr>
<th>Reactor</th>
</tr>
</thead>
<tbody>
<tr>
<td>register_handler()</td>
</tr>
<tr>
<td>remove_handler()</td>
</tr>
<tr>
<td>run_event_loop()</td>
</tr>
<tr>
<td>end_event_loop()</td>
</tr>
</tbody>
</table>
Solution: Separate Event Handling Concerns

• Create a reactor to detect input on various sources of events & then demux & dispatch the events to the appropriate event handlers

```
Reactors
register_handler()
remove_handler()
runevent_loop()
end_event_loop()
```

```python
Event_Handler

ET_Event_Handler

Verbose_ET_Event_Handler

Succinct_ET_Event_Handler
```
Solution: Separate Event Handling Concerns

- Create a reactor to detect input on various sources of events & then demux & dispatch the events to the appropriate event handlers
- Create concrete event handlers & register the concrete event handlers with the reactor

Event handlers perform various operation modes of the expression tree processing app
Solution: Separate Event Handling Concerns

- Create a reactor to *detect* input on various sources of events & then *demux* & *dispatch* the events to the appropriate event handlers
- Create concrete event handlers & register the concrete event handlers with the reactor
- Run the reactor’s event loop to drive the application event flow

*Note the inversion of control*
Reactor & Event Handler Class Interfaces

- An object-oriented event demultiplexor & dispatcher of event handler callback methods in response to various types of events

**Interface**

- Enact "inversion of control"

  ```
  ~Reactor()
  void run_event_loop()
  void end_event_loop()
  void register_handler(Event_Handler *event_handler)
  void remove_handler(Event_Handler *event_handler)
  static Reactor * instance()
  ```

  **Singleton access point**

  **Attach/ detach event handlers**
Reactor & Event Handler Class Interfaces

- An object-oriented event demultiplexor & dispatcher of event handler hook methods in response to various types of events

**Interface**

```cpp
virtual void ~Event_Handler()=0
virtual void delete_this()
virtual void handle_input()=0

virtual ~Reactor()
virtual void run_event_loop()
virtual void end_event_loop()
void register_handler(Event_Handler *event_handler)
void remove_handler(Event_Handler *event_handler)
static Reactor * instance()
```

- **Commonality**: Provides a common interface for managing & processing events via callbacks to abstract event handlers

- **Variability**: Concrete implementations of Reactor & event handlers can be tailored to a wide range of OS muxing mechanisms & application-specific concrete event handling behaviors
Reactors POSA2 Architectural Pattern

**Intent**

- Allows event-driven applications to demultiplex & dispatch service requests that are delivered to an application from one or more clients.

**Applicability**

- Need to decouple event handling from event management infrastructure.
- When multiple sources of events must be handled in a single thread.

**Structure**

![UML Diagram](www.dre.vanderbilt.edu/~schmidt/PDF/Reactor.pdf) for the Reactor pattern.

See [www.dre.vanderbilt.edu/~schmidt/PDF/Reactor.pdf](http://www.dre.vanderbilt.edu/~schmidt/PDF/Reactor.pdf) for the Reactor pattern.
Reactor example in C++

- Detect/demux events & dispatch event handler callback methods in response

```cpp
class Reactor {
public:
  static Reactor *instance();

  void run_event_loop() {
    while (run_event_loop_)
      wait_for_next_event()->handle_input();
  }

  void register_input_handler(Event_Handler *event_handler) {
    dispatch_table_.push_back(eh);
  }

private:
  std::vector<Event_Handler *> dispatch_table_;
  ...
};
```

Singleton access point

Run the app event loop

Register an event handler for input events in dispatch table
Reactor POSA2 Architectural Pattern

**Consequences**

+ Separation of concerns & portability
+ Simplify concurrency control
- Non-preemptive
- Scalability issues
Reactor | POSA2 Architectural Pattern

Consequences
+ Separation of concerns & portability
+ Simplify concurrency control
- Non-preemptive
- Scalability issues

Implementation
- Decouple event detection/demuxing mechanisms from event dispatching
  - e.g., via Bridge
- Handle many different types of events
  - e.g., input/output events, signals, timers, etc.
Reactor

Consequences
+ Separation of concerns & portability
+ Simplify concurrency control
- Non-preemptive
- Scalability issues

Implementation
• Decouple event detection/demuxing mechanisms from event dispatching
  • e.g., via Bridge
• Handle many different types of events
  • e.g., input/output events, signals, timers, etc.

Known Uses
• X Windows Xt
• InterViews Dispatcher
• ET++ WindowSystem
• AWT Toolkit
• ACE & The ACE ORB (TAO)
• Java NIO package

See gee.cs.oswego.edu/dl/cpjslides/nio.pdf for Java Reactor info
### Problem: Managing Access to Global Resources

#### Goals

- Centralize access to resources that should be visible globally, e.g.:
  - Command-line options that parameterize the program behavior
  - Reactor that drives the main event loop

```plaintext
% tree-traversal -v
format [in-order]
expr [expression]
print [in-order|pre-order|post-order|level-order]
```

```plaintext
% tree-traversal
> 1+4*3/2
7
```

- **Verbose mode**

```plaintext
% tree-traversal -v
format [in-order]
expr [expression]
print [in-order|pre-order|post-order|level-order]
```

```plaintext
% tree-traversal
> 1+4*3/2
7
```

- **Succinct mode**
Problem: Managing Access to Global Resources

Goals

- Centralize access to resources that should be visible globally, e.g.:
  - Command-line options that parameterize the program behavior
  - Reactor that drives the main event loop

Constraints/forces

- Only need one instance of command-line options & event loop driver
- Global variables are problematic in C++
Solution: Centralize Access to Global Resources

- Rather than using global variables, create a central access point to global instances, e.g.:

  ```c
  int main(int argc, char *argv[]) {
    Parse the command-line options
    if(!Options::instance()->parse_args(argc, argv))
      return 0;
  Dynamically allocate the appropriate event handler based on the command-line options

  ET_Event_Handler *event_handler =
    ET_Event_Handler::make_handler
    (Options::instance()->verbose());

  Register event handler with the event loop driver
  Reactor::instance()->register_input_handler (event_handler);
  // ...
Intent

- Ensure a class only has one instance & provide a global point of access

Applicability

- When there must be exactly one instance of a class & it must be accessible from a well-known access point
- When the sole instance should be extensible by subclassing & clients should be able to use an extended instance without modifying their code

Structure

```java
if (uniqueInstance == 0)
    uniqueInstance = new Singleton;
return uniqueInstance;
```

**Singleton**

- static instance()
- singletonOperation()
- getSingletonData()
- static uniqueInstance
- singletonData
Singleton example in C++

- Define a singleton class to handle command-line option processing

```cpp
class Options {
public:
    static Options *instance();

    // Parse command-line arguments and sets values as follows:
    // 't' - Traversal strategy, i.e., 'P' for pre-order, 'O' for
    // post-order, 'I' for in-order, & 'L' for level-order.
    bool parse_args(int argc, char *argv[]);
    bool verbose() const; // True if program runs in verbose mode.
    char traversal_strategy() // Returns desired traversal strategy
private:
    Options();
    static Options *instance_; // Points to the one & only instance
    ...}
```

if (instance_ == 0)
    instance_ = new Options;
return instance_;
Singleton

Consequences

+ Reduces namespace pollution
+ Makes it easy to change your mind & allow more than one instance
+ Allow extension by subclassing
- Same drawbacks of a global if misused
- Implementation may be less efficient than a global
- Concurrency/cache pitfalls & communication overhead

See [c2.com/cgi/wiki?SingletonsAreEvil](http://c2.com/cgi/wiki?SingletonsAreEvil) for discussion of Singleton issues
**Singleton**

**GoF Object Creational**

**Consequences**

+ Reduces namespace pollution
+ Makes it easy to change your mind & allow more than one instance
+ Allow extension by subclassing
  - Same drawbacks of a global if misused
  - Implementation may be less efficient than a global
  - Concurrency/cache pitfalls & communication overhead

**Implementation**

- Static instance operation
- Registering singleton instance with manager
- Deleting singletons

[source-making.com/design_patterns/to_kill_a Singleton: Singleton deletion](source-making.com/design_patterns/to_kill_a Singleton: Singleton deletion)
Singleton

**Consequences**

+ Reduces namespace pollution
+ Makes it easy to change your mind & allow more than one instance
+ Allow extension by subclassing
  - Same drawbacks of a global if misused
  - Implementation may be less efficient than a global
  - Concurrency/cache pitfalls & communication overhead

**Implementation**

- Static instance operation
- Registering singleton instance with manager
- Deleting singletons

Singleton

Consequences
+ Reduces namespace pollution
+ Makes it easy to change your mind & allow more than one instance
+ Allow extension by subclassing
  - Same drawbacks of a global if misused
  - Implementation may be less efficient than a global
  - Concurrency/cache pitfalls & communication overhead

Implementation
• Static instance operation
• Registering singleton instance with manager
• Deleting singletons

GoF Object Creational

Known Uses
• Unidraw's Unidraw object
• Smalltalk-80 ChangeSet, the set of changes to code
• InterViews Session object
• ACE Singleton

See Also
• Double-Checked Locking Optimization pattern from POSA2 book

en.wikipedia.org/wiki[Double-checked_locking] has more synchronization info
Summary of Application Structure Patterns

The Reactor pattern embodies key characteristics of frameworks. Reactor structures the overall control flow of the event-driven expression tree processing app and Singleton simplifies access to global resources.
A Case Study of “Gang of Four” (GoF) Patterns : Part 11

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

Professor of Computer Science
Institute for Software Integrated Systems
Vanderbilt University
Nashville, Tennessee, USA
Topics Covered in this Part of the Module

- Describe the object-oriented (OO) expression tree case study
- Evaluate the limitations with algorithmic design techniques
- Present an OO design for the expression tree processing app
- Summarize the patterns in the expression tree design
- Explore patterns for
  - Tree structure & access
  - Tree creation
  - Tree traversal
  - Commands & factories
  - Command ordering protocols
  - Application structure
  - Encapsulating algorithm variability
Overview of Algorithm Variability
Encapsulation Patterns

**Purpose:** Simplify processing of multiple operation modes

These patterns support controlled variability of steps in an algorithm.
Problem: Supporting Multiple Operation Modes

Goals

- Minimize effort required to support multiple modes of operation
  - e.g., verbose & succinct modes

```
% tree-traversal -v
format [in-order]
expr [expression]
print [in-order|pre-order|post-order|level-order]
eval [post-order]
quit
> format in-order
> expr 1+4*3/2
> eval post-order
7
> quit
%
```

Verbose mode

```
% tree-traversal
> 1+4*3/2
7
```

Succinct mode
Problem: Supporting Multiple Operation Modes

Goals
- Minimize effort required to support multiple modes of operation
  - e.g., *verbose* & *succinct* mode

Constraints/forces
- Don’t tightly couple the operation modes with the program structure
- Simplify future enhancements
- Avoid limitations of algorithmic decomposition

```c
void print_tree (TreeNode *root)
{
    switch (root->tag)
    {
    case NUM: printf (“%d”, root->num);
    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");
    }
}
```
Solution A: Encapsulate Algorithm Variability

- Implement algorithm once in base class & let subclasses define variant parts

```cpp
void handle_input() {
    prompt_user();
    std::string input;
    if(!get_input(input))
        Reactor::instance()->end_event_loop();

    ET_Command command = make_command(input);
    if(!execute_command(command))
        Reactor::instance()->end_event_loop();
}
```
Solution A: Encapsulate Algorithm Variability

- Implement algorithm once in base class & let subclasses define variant parts

```cpp
void handle_input() {
    prompt_user();
    std::string input;
    if(!get_input(input))
        Reactor::instance()->end_event_loop();
    ET_Command command = make_command(input);
    if(!execute_command(command))
        Reactor::instance()->end_event_loop();
}
```

*handle_input() is a template method*

The other four methods are “hook methods”
Solution A: Encapsulate Algorithm Variability

- Implement algorithm once in base class & let subclasses define variant parts

```cpp
void handle_input() {
    prompt_user();
    std::string input;
    if(!get_input(input))
        Reactor::instance()->end_event_loop();
    ET_Command command = make_command(input);
    if(!execute_command(command))
        Reactor::instance()->end_event_loop();
}
```
Solution A: Encapsulate Algorithm Variability

- Implement algorithm once in base class & let subclasses define variant parts

```cpp
void handle_input() {
    prompt_user();
    std::string input;
    if(!get_input(input))
        Reactor::instance()->end_event_loop();
    ET_Command command = make_command(input);
    if(!execute_command(command))
        Reactor::instance()->end_event_loop();
}
```

```cpp
ET_Command make_command(const std::string &input) {
    return command_factory_.make_macro_command(input);
}
```
**ET_Event_Handler Class Interface**

- Provides an abstract interface for performing the algorithm associated with the expression tree processing app

**Interface**

```cpp
virtual void handle_input()

static ET_Event_Handler * make_handler(bool verbose)

virtual void prompt_user() = 0

virtual bool get_input(std::string &)

virtual ET_Command make_command(const std::string &input) = 0

virtual bool execute_command(ET_Command &)
```

- **Commonality**: Provides a common interface for handling user input events & performing steps in the expression tree processing algorithm
- **Variability**: Subclasses implement various operation modes, e.g., verbose vs. succinct mode
### Intent
- Provide a skeleton of an algorithm in a method, deferring some steps to subclasses

### Applicability
- Implement invariant aspects of an algorithm *once* & let subclasses define variant parts
- Localize common behavior in a class to increase code reuse
- Control subclass extensions

### Structure
- **AbstractClass**
  - TemplateMethod()
  - PrimitiveOperation1()
  - PrimitiveOperation2()

- **ConcreteClass**
  - PrimitiveOperation1()
  - PrimitiveOperation2()

- **e.g., ET_Event_Handler**

- **e.g., Verbose_ET_Event_Handler, Succinct_ET_Event_Handler**
Template Method example in C++

- Allow subclasses to customize certain steps in event handling algorithm

```cpp
void ET_Event_Handler::handle_input() {
    prompt_user();
    std::string input;
    if (!get_input(input)) Reactor::instance()->end_event_loop();
    ET_Command command = make_command(input);
    if (!execute_command(command))
        Reactor::instance()->end_event_loop();
    ...
```
Template Method example in C++

- Allow subclasses to customize certain steps in event handling algorithm

```cpp
void ET_Event_Handler::handle_input() {
    prompt_user();
    std::string input;
    if (!get_input(input)) Reactor::instance()->end_event_loop();

    ET_Command command = make_command(input);
    if (!execute_command(command))
        Reactor::instance()->end_event_loop();

    ...}
```

```cpp
ET_Command Verbose_ET_Event_Handler::make_command(const std::string &input) {
    return command_factory_.make_command(input);
}
```

```cpp
ET_Command Succinct_ET_Event_Handler::make_command(const std::string &input) {
    return command_factory_.make_macro_command(input);
}
```
Template Method example in C++

- Allow subclasses to customize certain steps in event handling algorithm

```c++
void ET_Event_Handler::handle_input() {
    prompt_user();
    std::string input;
    if (!get_input(input)) Reactor::instance()->end_event_loop();

    ET_Command command = make_command(input);
    if (!execute_command(command))
        Reactor::instance()->end_event_loop();

    // ...
}

ET_Event_Handler *ET_Event_Handler::make_handler(bool verbose) {
    return verbose ? new Verbose_ET_Event_Handler
                   : new Succinct_ET_Event_Handler
}
```

*Factory creates appropriate strategy objects*

This is not the only (or best) way of defining a factory
**Template Method**

**GoF Class Behavioral**

**Consequences**

+ Enables inversion of control
  ("Hollywood principle: don't call us - we'll call you!")
+ Promotes code reuse
+ Lets you enforce overriding rules
  - Must subclass to specialize behavior (*cf. Strategy* pattern)
### Template Method

#### Consequences

++ Enables inversion of control

("Hollywood principle: don't call us – we'll call you!")

++ Promotes code reuse

++ Lets you enforce overriding rules

- Must subclass to specialize behavior (*cf. Strategy* pattern)

#### Implementation

- Virtual vs. non-virtual template method
- Few vs. many primitive operations (hook methods)
- Naming conventions (*do_*() vs. *make_*() prefix)
**Consequences**

+ Enables inversion of control  
  (“Hollywood principle: don't call us – we'll call you!”)
+ Promotes code reuse
+ Lets you enforce overriding rules
  - Must subclass to specialize behavior (*cf. Strategy* pattern)

**Implementation**

- Virtual vs. non-virtual template method
- Few vs. many primitive operations (hook methods)
- Naming conventions (*do_*() vs. *make_*() prefix)

**Known Uses**

- InterViews Kits
- ET++ WindowSystem
- AWT Toolkit
- ACE & The ACE ORB (TAO)
- Android Activities
Solution B: Encapsulate Algorithm Variability

- Implement algorithm once in base class & let strategies define variant parts

```cpp
void handle_input() {
    prompt_user();
    std::string input;
    if(!get_input(input))
        Reactor::instance()->end_event_loop();
    ET_Command cmd = make_command(input);
    if(!execute_command(cmd))
        Reactor::instance()->end_event_loop();
}
```

Same template method & hook methods as before
Solution B: Encapsulate Algorithm Variability

- Implement algorithm once in base class & let strategies define variant parts

void handle_input() {
    prompt_user();
    std::string input;
    if(!get_input(input))
        Reactor::instance()->end_event_loop();
    ET_Command cmd = make_command(input);
    if(!execute_command(cmd))
        Reactor::instance()->end_event_loop();
}

PROMPT_USER_STRATEGY
GET_INPUT_STRATEGY
MAKE_COMMAND_STRATEGY
EXECUTE_COMMAND_STRATEGY

Strategy objects can be passed as arguments to Strategy_ET_Event_Handler constructor

Strategy objects can be defined using C++ parameterized types
Solution B: Encapsulate Algorithm Variability

- Implement algorithm once in base class & let strategies define variant parts

```cpp
void handle_input() {
    prompt_user();
    std::string input;
    if(!get_input(input))
        Reactor::instance()->end_event_loop();
    ET_Command cmd = make_command(input);
    if(!execute_command(cmd))
        Reactor::instance()->end_event_loop();
}

ET_Command make_command(const std::string &input) {
    return mc_strategy_.make_command(input);
}

bool execute_command(const std::string &input) {
    return ec_strategy_.execute_command(input);
}
```
Strategy Template Parameters

- Provides template parameters that perform steps in the algorithm associated with the expression tree processing app

Template Parameters

```cpp
MAKE_COMMAND_CONSTRAINSITY
ET_Command make_command(const std::string &input)
```

```cpp
EXECUTE_COMMAND_CONSTRAINTSTRATEGY
bool execute_command(const std::string &input)
```

- **Commonality**: Provides common expected method signatures for performing steps in the expression tree processing algorithm
- **Variability**: Template arguments provided to `Strategy_ET_Event_Handler` implement various operation modes, e.g., verbose vs. succinct
Strategy

**Intent**
- Define a family of algorithms, encapsulate each one, & make them interchangeable to let clients & algorithms vary independently

**Applicability**
- When an object should be configurable with one of many algorithms,
- *and* all algorithms can be encapsulated,
- *and* one interface covers all encapsulations

**Structure**

```
Context
  contextInterface()

Strategy
  algorithmInterface()
```

E.g., `MAKE_COMMAND_STRATEGY`, `EXECUTE_COMMAND_STRATEGY`, etc.

```
ConcreteStrategyA
  algorithmInterface()

ConcreteStrategyB
  algorithmInterface()

ConcreteStrategyC
  algorithmInterface()
```
**Strategy example in C++**

- Customize algorithm behavior by composition/forwarding vs. inheritance

```cpp
template <..., typename MAKE_COMMAND_STRATEGY, typename EXECUTE_COMMAND_STRATEGY>
class Strategy_ET_Event_Handler : public ET_Event_Handler {
public:
    Strategy_ET_Event_Handler(...) {
        const MAKE_COMMAND_STRATEGY &mc_strategy,
        const EXECUTE_COMMAND_STRATEGY &ec_strategy) ...
```

Parameterized type strategies

Reuse handle_input() template method

Pass strategy objects via constructor & assign to corresponding data members

Note combination of Template Method & Strategy patterns
Strategy example in C++

• Customize algorithm behavior by composition/forwarding vs. inheritance

```cpp
template <..., typename MAKE_COMMAND_STRATEGY,
          typename EXECUTE_COMMAND_STRATEGY>
class Strategy_ET_Event_Handler : public ET_Event_Handler {
public:
    Strategy_ET_Event_Handler
        (...,
        const MAKE_COMMAND_STRATEGY &mc_strategy,
        const EXECUTE_COMMAND_STRATEGY &ec_strategy) ...
    ...

    ET_Command make_command(const std::string &user_input)
    { return mc_strategy_.make_command(user_input); }

    bool execute_command(ET_Command &command)
    { return ec_strategy_.execute_command(command); }
```

Hook methods forward to strategy objects
Strategy example in C++

- Customize algorithm behavior by composition/forwarding vs. inheritance

```cpp
template <..., typename MAKE_COMMAND_STRATEGY, typename EXECUTE_COMMAND_STRATEGY>
class Strategy_ET_Event_Handler : public ET_Event_Handler {
public:
    Strategy_ET_Event_Handler(...) {
        const MAKE_COMMAND_STRATEGY &mc_strategy,
        const EXECUTE_COMMAND_STRATEGY &ec_strategy)
    ...
    ...

    class Macro_Command_Strategy {
    public:
        ...
        ET_Command make_command(const std::string &input) {
            return command_factory_.make_macro_command(input);
        }
    }
```
Strategy example in C++

- Customize algorithm behavior by composition/forwarding vs. inheritance

```cpp
template <..., typename MAKE_COMMAND_STRATEGY,
           typename EXECUTE_COMMAND_STRATEGY>
class Strategy_ET_Event_Handler : public ET_Event_Handler {
public:
  Strategy_ET_Event_Handler
  (...,
   const MAKE_COMMAND_STRATEGY &mc_strategy,
   const EXECUTE_COMMAND_STRATEGY &ec_strategy) ...
...

Factory creates appropriate strategy objects

ET_Event_Handler *Strategy_ET_Event_Handler::make_handler
  (bool verbose) {
  return verbose ? new Strategy_ET_Event_Handler
                   <..., Command_Strategy, ...>
    : new Strategy_ET_Event_Handler
      <..., Macro_Command_Strategy, ...>
}
```
**Consequences**

+ Greater flexibility, reuse
+ Can change algorithms dynamically
- Strategy creation & communication overhead
- Inflexible Strategy interface
- Semantic incompatibility of multiple strategies used together
## Consequences

+ Greater flexibility, reuse  
+ Can change algorithms dynamically  
- Strategy creation & communication overhead  
- Inflexible strategy interface  
- Semantic incompatibility of multiple strategies used together

## Implementation

- Exchanging information between a strategy & its context  
  - Context is not always necessary  
- Static binding of strategy selection via parameterized types
GoF Patterns Expression Tree Case Study

Douglas C. Schmidt

Strategy

Consequences
+ Greater flexibility, reuse
+ Can change algorithms dynamically
- Strategy creation & communication overhead
- Inflexible strategy interface
- Semantic incompatibility of multiple strategies used together

Implementation
• Exchanging information between a strategy & its context
  • Context is not always necessary
• Static binding of strategy selection via parameterized types

GoF Object Behavioral

Known Uses
• InterViews text formatting
• RTL register allocation & scheduling strategies
• ET++SwapsManager calculation engines
• The ACE ORB (TAO) real-time object request broker middleware

See Also
• Bridge pattern (object structural)
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
  - Incurs the overhead of forwarding
  - May yield many strategy classes
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
- Incurs the overhead of forwarding
- May yield many strategy classes

**Template Method**
- No explicit forwarding necessary
- May be easier for small use cases
  - Close coupling between subclass(es) & base 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* is commonly used for Black-box frameworks
*Template Method* is commonly used for White-box frameworks
Overview of Algorithm Variability
Encapsulation Patterns

The *Strategy & Template Method* patterns simplify processing of multiple operation modes.

These patterns avoid limitations of algorithmic decomposition.