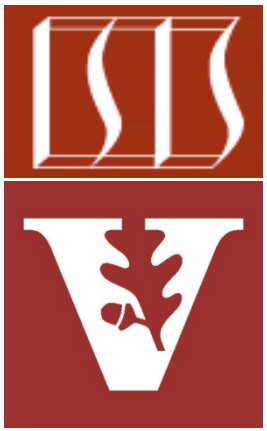
A Case Study of "Gang of Four" (GoF) Patterns : Part 8

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



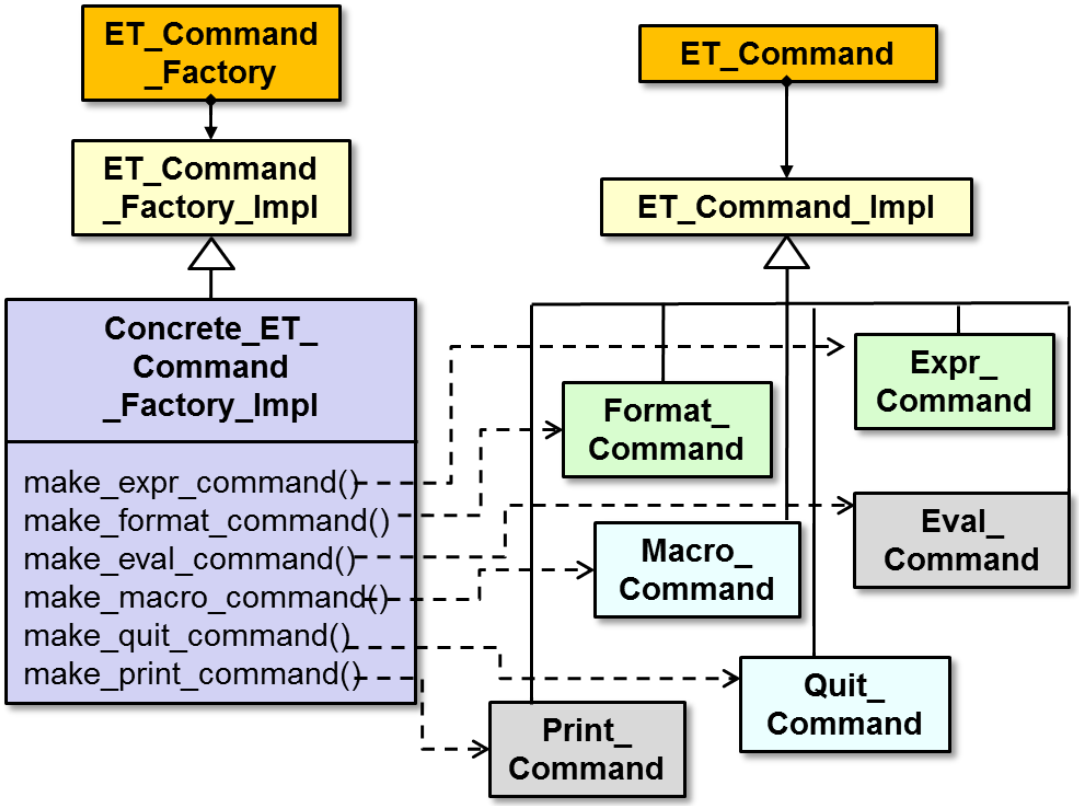
Professor of Computer Science

Institute for Software Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA

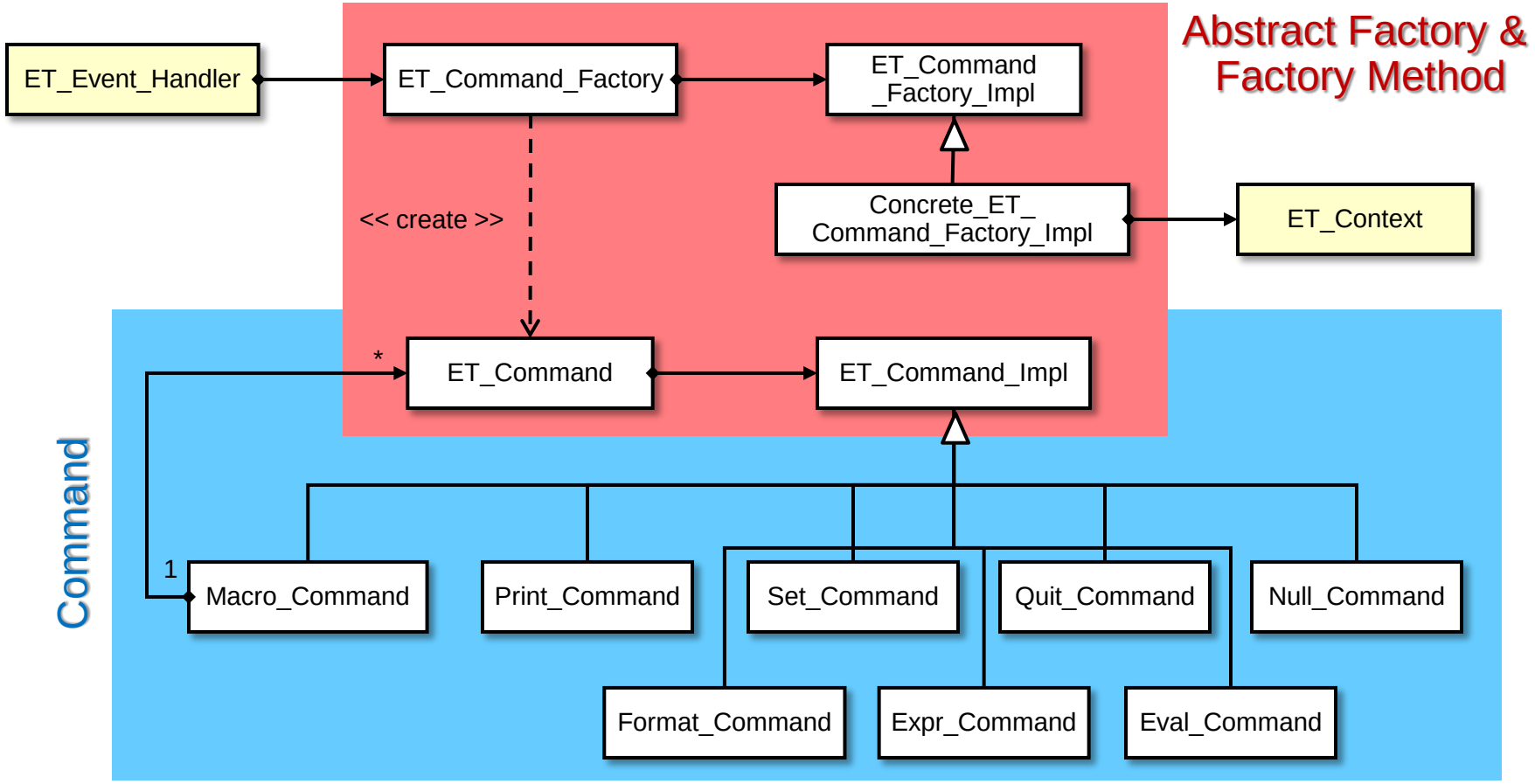
Topics Covered in this Part of the Module

- 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



Overview of Command & Factory Patterns

***Purpose**: Define operations that can users can perform on an expression tree processing app & centralize extensible creation of these operations*



These patterns decouple creation from use & provide a uniform command API

Problem: Consolidating User Operations

Goals

- Support execution of user operations

```
% 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

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

Verbose mode

Problem: Consolidating User Operations

Goals

- Support execution of user operations
- Support macro operations

```
% 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

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

Succinct mode

Problem: Consolidating User Operations

Goals

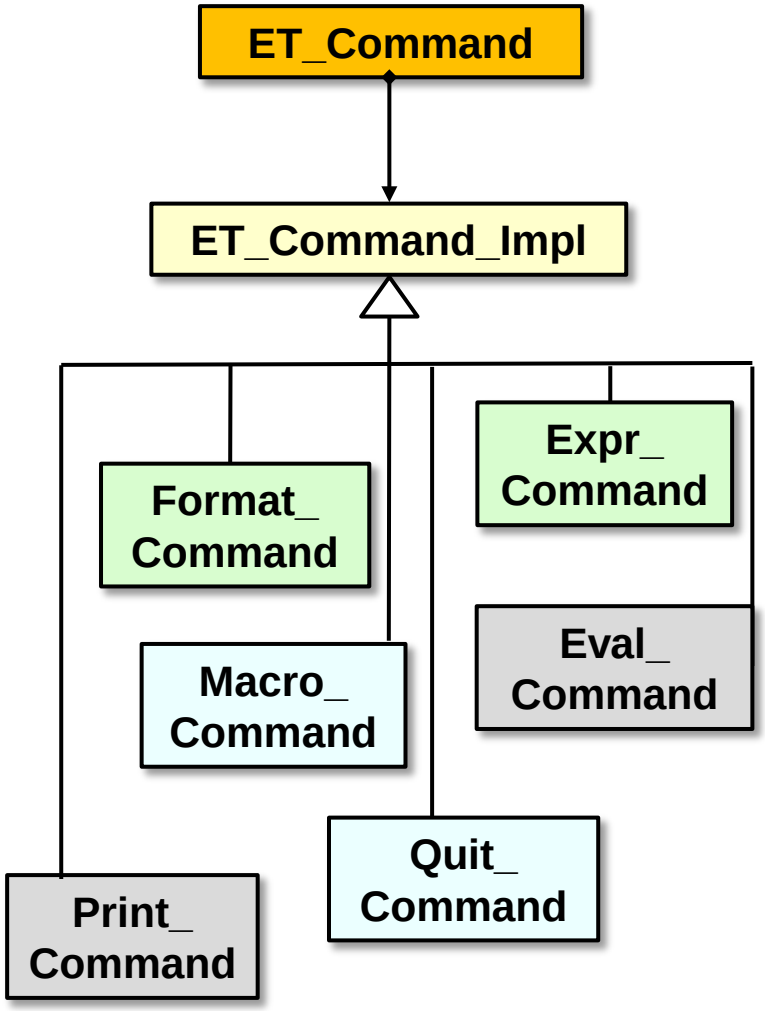
- Support execution of user operations
- Support macro operations

Constraints / forces

- Avoid scattering the implementation of operations throughout the source code
- Ensure consistent memory management

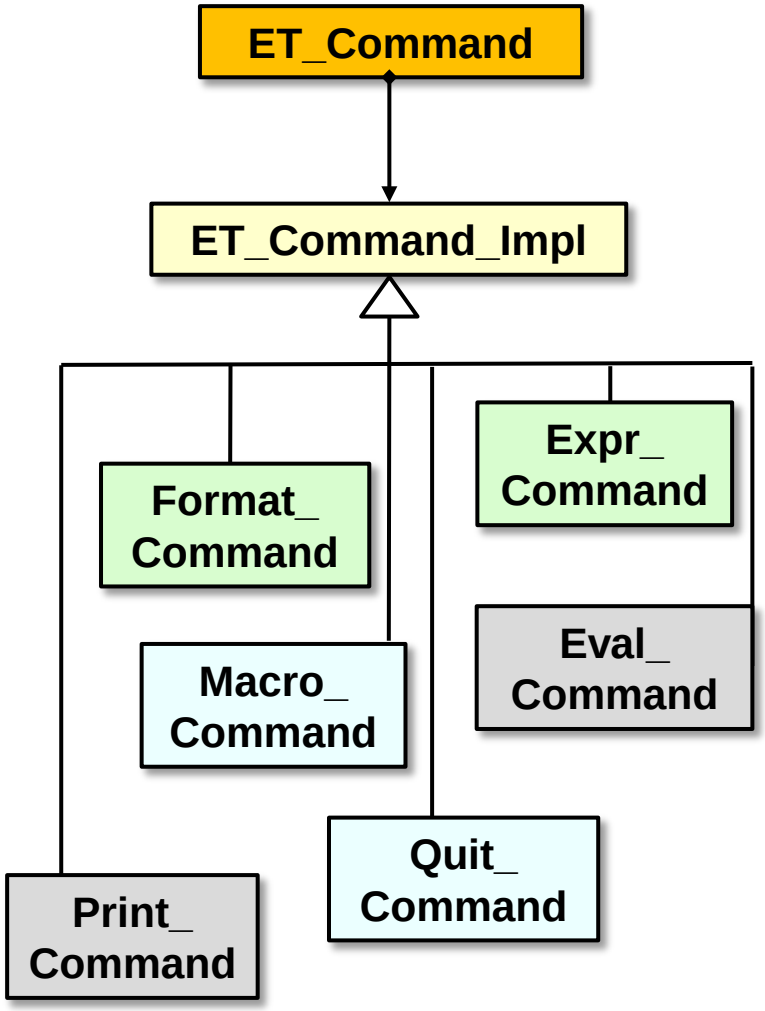
Solution: Encapsulate an Operation w/Command

- A **Command** encapsulates
  - An operation method (**`execute()`**)
  - An inverse operation method (**`unexecute()`**)
  - A test for reversibility (**`boolean reversible()`**)
  - State for (un)doing the operation



Solution: Encapsulate an Operation w/Command

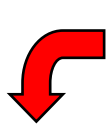
- A **Command** encapsulates
  - An operation method (`execute()`)
  - An inverse operation method (`unexecute()`)
  - A test for reversibility (`boolean reversible()`)
  - State for (un)doing the operation
- A **Command** may
  - Implement the operation itself or
  - Forward the operation implementation to other object(s)



Bridge pattern encapsulates variability & simplifies memory management

ET_Command Class Interface

- Interface for defining a command that—when executed—performs an operation on an expression tree



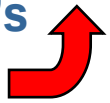
This class plays the role of the abstraction in the *Bridge* pattern

Interface

```
ET_Command(ET_Command_Impl *=0)
ET_Command(const ET_Command &)
ET_Command & operator=(const ET_Command &)
~ET_Command()
bool execute()
bool unexecute()
```

These methods forward to the implementor subclass

We don't use this method but it's a common command feature



- **Commonality**: Provides common interface for expression tree commands
- **Variability**: Implementations of expression tree commands can vary depending on the operations requested by user input

ET_Command_Impl Class Interface

- Base class of an implementor hierarchy used to define commands that perform operations on an expression tree when they are executed

Interface

This class is the base class of the implementor hierarchy in the *Bridge* pattern

```
ET_Command_Impl(ET_Context &)
~ET_Command_Impl()= 0
...
virtual bool execute()=0
virtual bool unexecute()=0
```

The bulk of the work is typically done here by subclasses

- **Commonality**: Provides a common base class for implementations of expression tree commands
- **Variability**: Subclasses of this base class implement different expression tree commands depending on the operations requested by user input

Command GoF Object Behavioral

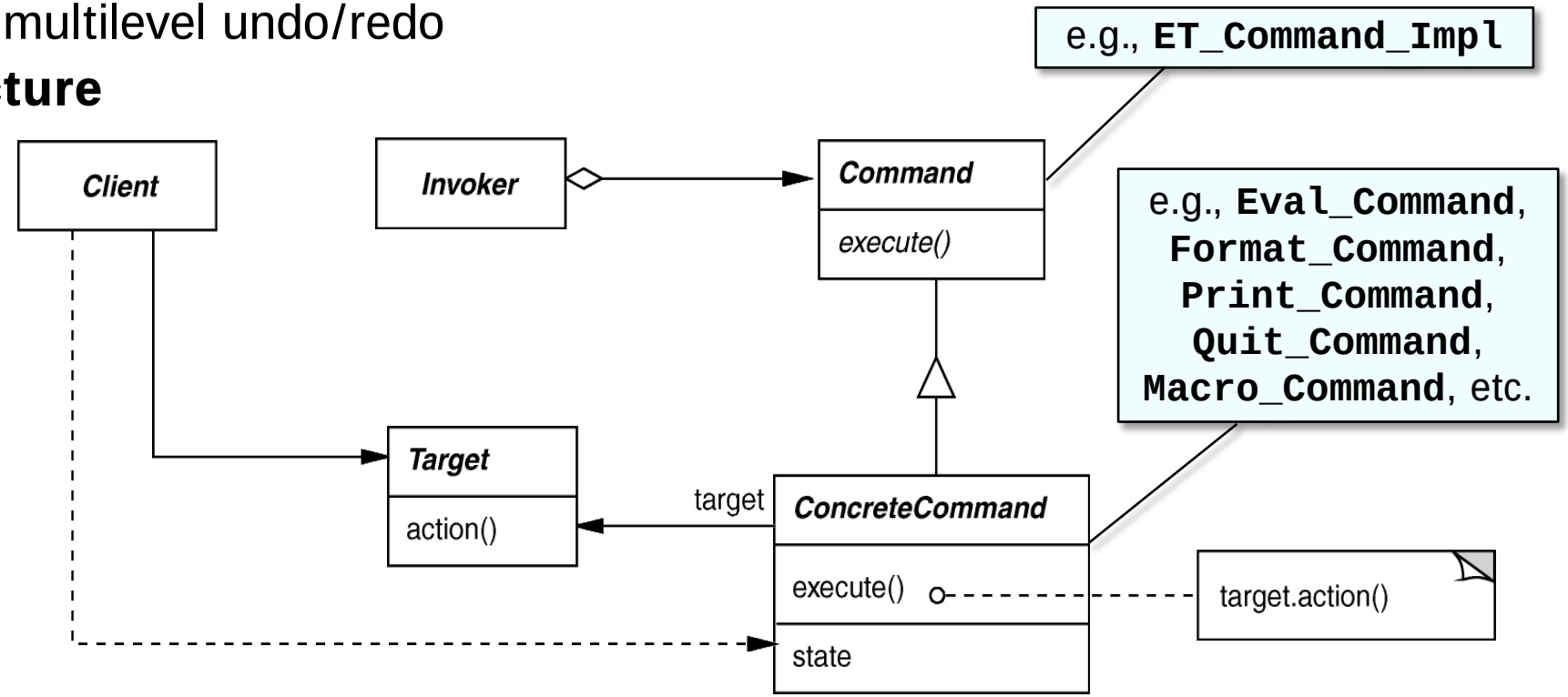
Intent

- Encapsulate the request for a service as an object

Applicability

- Want to parameterize objects with an action to perform
- Want to specify, queue, & execute requests at different times
- For multilevel undo/redo

Structure



Command GoF Object Behavioral

Command example in C++

- Encapsulate execution of a sequence of commands as an object

```
class Macro_Command : public ET_Command_Impl {
public:
  ...
  bool execute() {
    std::for_each (macro_commands_.begin(),
                   macro_commands_.end(),
                   std::mem_fun_ref(&ET_Command::execute));
    return true;
  }

private:
  std::vector <ET_Command> macro_commands_;
  ...
```

Executes a sequence of commands (used to implement "succinct mode")

Vector of commands to execute as a macro

Command GoF Object Behavioral

Command example in C++

- Encapsulate execution of a sequence of commands as an object

```
class Macro_Command : public ET_Command_Impl {
public:
  ...
  bool execute() {
    std::for_each (macro_commands_.begin(),
                   macro_commands_.end(),
                   std::mem_fun_ref(&ET_Command::execute));
    return true;
  }

private:
  std::vector <ET_Command> macro_commands_;
  ...
```

STL `for_each()` algorithm

Application of the *Adapter* pattern

Vector of commands to execute as a macro

Command GoF Object Behavioral

Command example in C++

- Encapsulate execution of a sequence of subcommands as an object

```
class Macro_Command : public ET_Command_Impl {
public:
  ...
  bool execute() {
    std::for_each (macro_commands_.begin(),
                   macro_commands_.end(),
                   [](ET_Command &c){ c.execute(); });
    return true;
  }

private:
  std::vector <ET_Command> macro_commands_;
  ...
```

C++11 lambda expression

Command GoF Object Behavioral

Command example in C++

- Encapsulate execution of a sequence of subcommands as an object

```
class Macro_Command : public ET_Command_Impl {
public:
  ...
  bool execute() {
    for (auto &iter = macro_commands_)
      iter.execute();

    return true;
  }

private:
  std::vector <ET_Command> macro_commands_;
  ...
```

C++11 range-based for loop

Command GoF Object Behavioral

Consequences

+ Abstracts executor of a service
+ Supports arbitrary-level undo-redo
+ Composition yields macro-commands
– Might result in lots of trivial command subclasses
– Excessive memory may be needed to support undo/redo operations

Command GoF Object Behavioral

Consequences

+ Abstracts executor of a service
+ Supports arbitrary-level undo-redo
+ Composition yields macro-commands
– Might result in lots of trivial command subclasses
– Excessive memory may be needed to support undo/redo operations

Implementation

- Copying a command before putting it on a history list
- Avoiding error accumulation during undo/redo
- Supporting transactions

Command GoF Object Behavioral

Consequences

+ Abstracts executor of a service
+ Supports arbitrary-level undo-redo
+ Composition yields macro-commands
– Might result in lots of trivial command subclasses
– Excessive memory may be needed to support undo/redo operations

Implementation

- Copying a command before putting it on a history list
- Avoiding error accumulation during undo/redo
- Supporting transactions

Known Uses

- InterViews Actions
- MacApp, Unidraw Commands
- JDK's UndoableEdit, AccessibleAction
- Emacs
- Microsoft Office tools

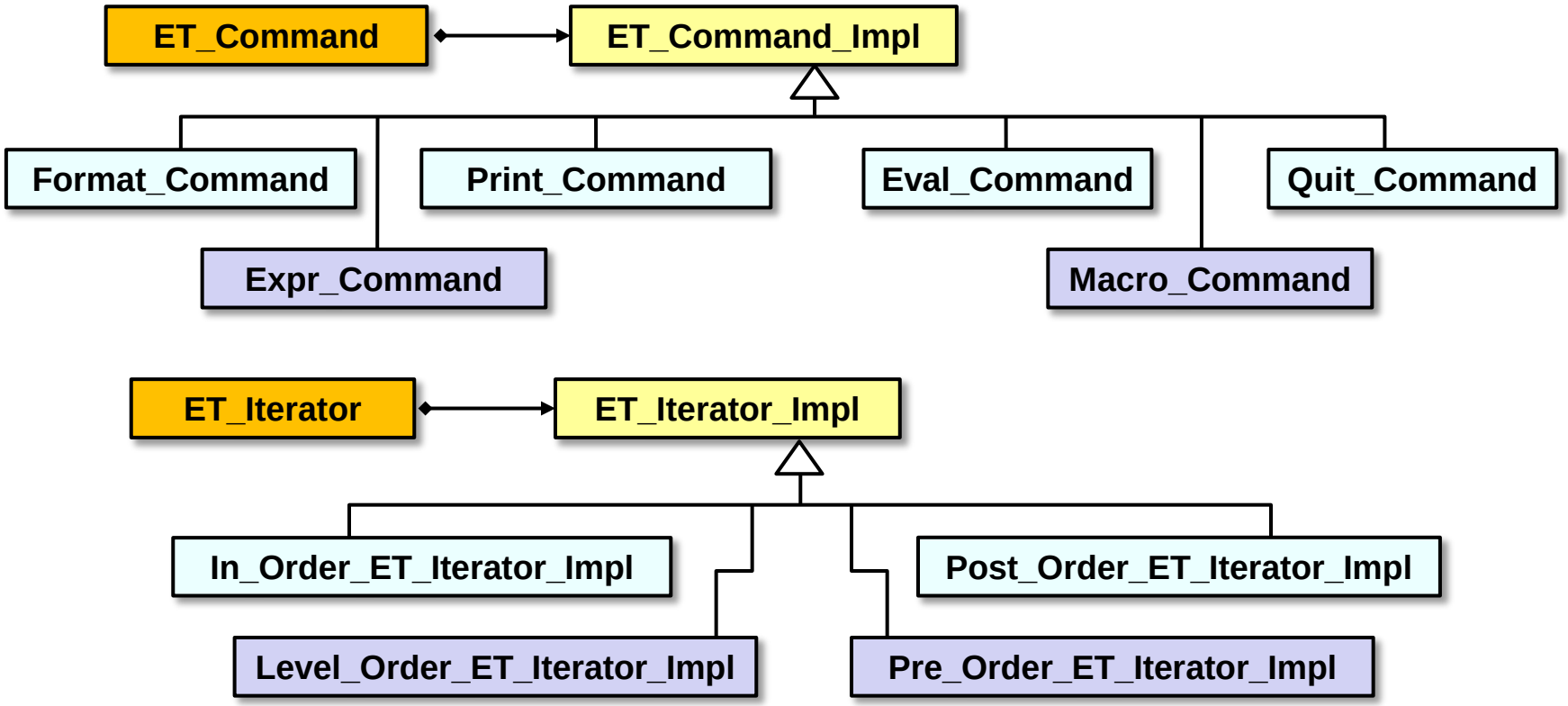
See Also

- *Command Processor* pattern in POSA1

Problem: Consolidating Creation of Variabilities

Goals

- Simplify & centralize the creation of all variabilities in the expression tree application to ensure semantic compatibility
- Be extensible for future variabilities



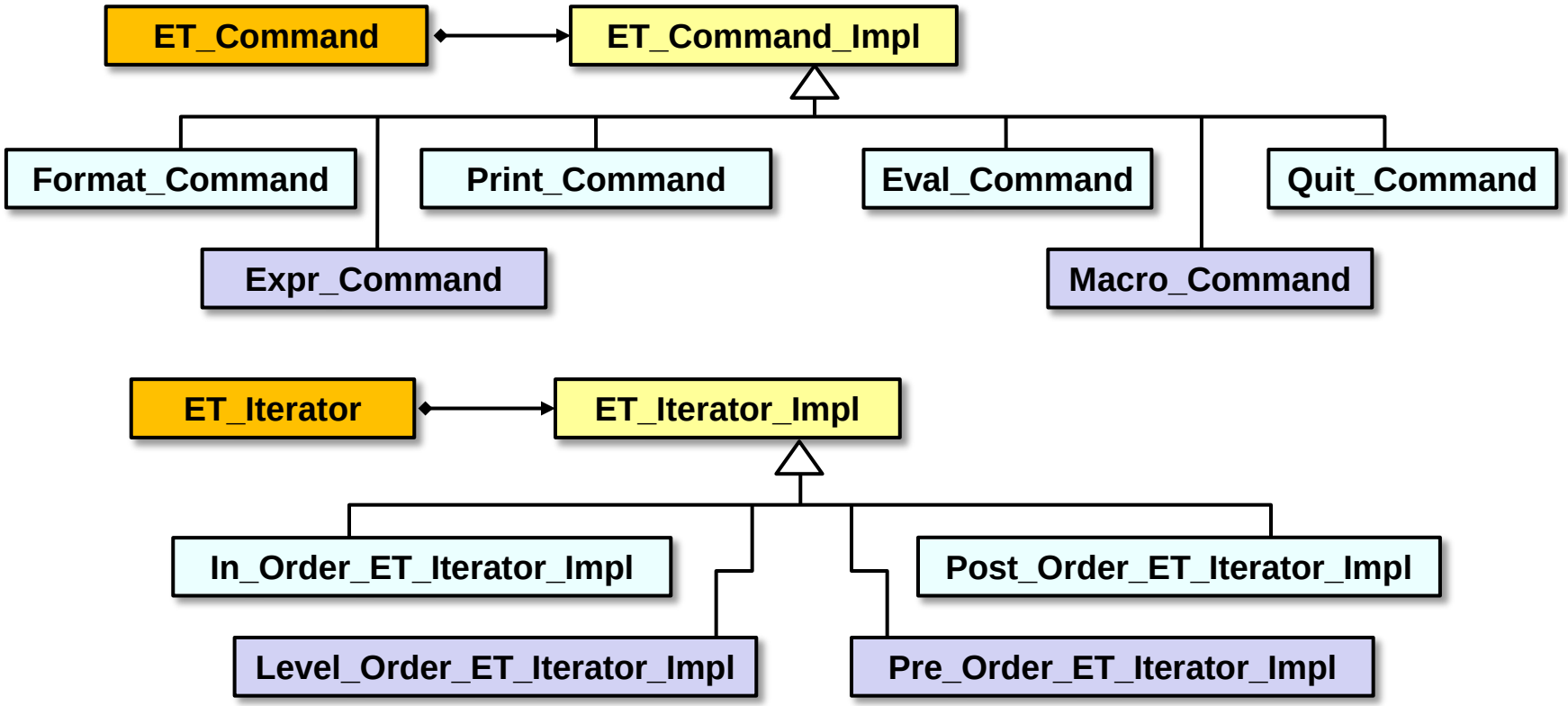
Problem: Consolidating Creation of Variabilities

Goals

- Simplify & centralize the creation of all variabilities in the expression tree application to ensure semantic compatibility
- Be extensible for future variabilities

Constraints / forces

- Don't recode existing clients
- Add new variabilities without recompiling



Solution: Abstract Object Creation

Factory

- Instead of

```
ET_Command command(new Print_Command());
```

Hard-codes a lexical dependency on `Print_Command`

Use

```
ET_Command command
           (command_factory.make_command("print"));
```

No lexical dependency on any concrete class

where **`command_factory`** is an instance of **`ET_Command_Factory`**

Solution: Abstract Object Creation

Factory

- Instead of

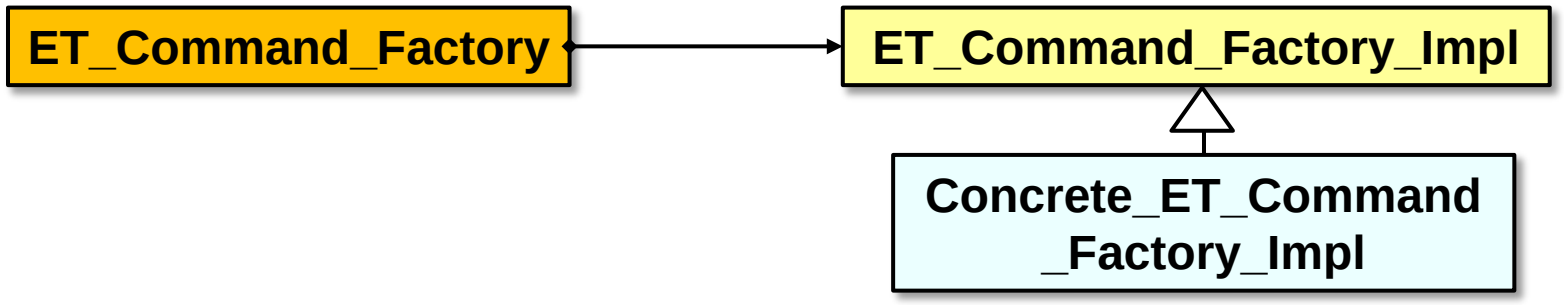
```
ET_Command command(new Print_Command());
```

Use

```
ET_Command command
          (command_factory.make_command("print"));
```

where `command_factory` is an instance of `ET_Command_Factory`

Factory structure



Bridge pattern encapsulates variability & simplifies memory management

ET_Command_Factory Class Interface

- Interface used to create appropriate command based on string supplied by caller

Interface

This class plays the role of the abstraction in the *Bridge* pattern

```
ET_Command_Factory(ET_Context &tree_context)
ET_Command_Factory(ET_Command_Factory_Impl *)
ET_Command make_command(const std::string &s)
...
```

This is the main factory method

These are helper factory methods

```
ET_Command make_format_command(const std::string &)
ET_Command make_expr_command(const std::string &)
ET_Command make_print_command(const std::string &)
ET_Command make_eval_command(const std::string &)
ET_Command make_quit_command(const std::string &)
ET_Command make_macro_command(const std::string &)
```

- **Commonality**: Provides a common interface to create commands
- **Variability**: Implementations of expression tree command factory methods can vary depending on the requested commands

ET_Command_Factory_Impl Class Interface

- Base class of an implementor hierarchy used to create appropriate commands based on string supplied by caller

Interface

This class is base class of implementor hierarchy in the *Bridge* pattern

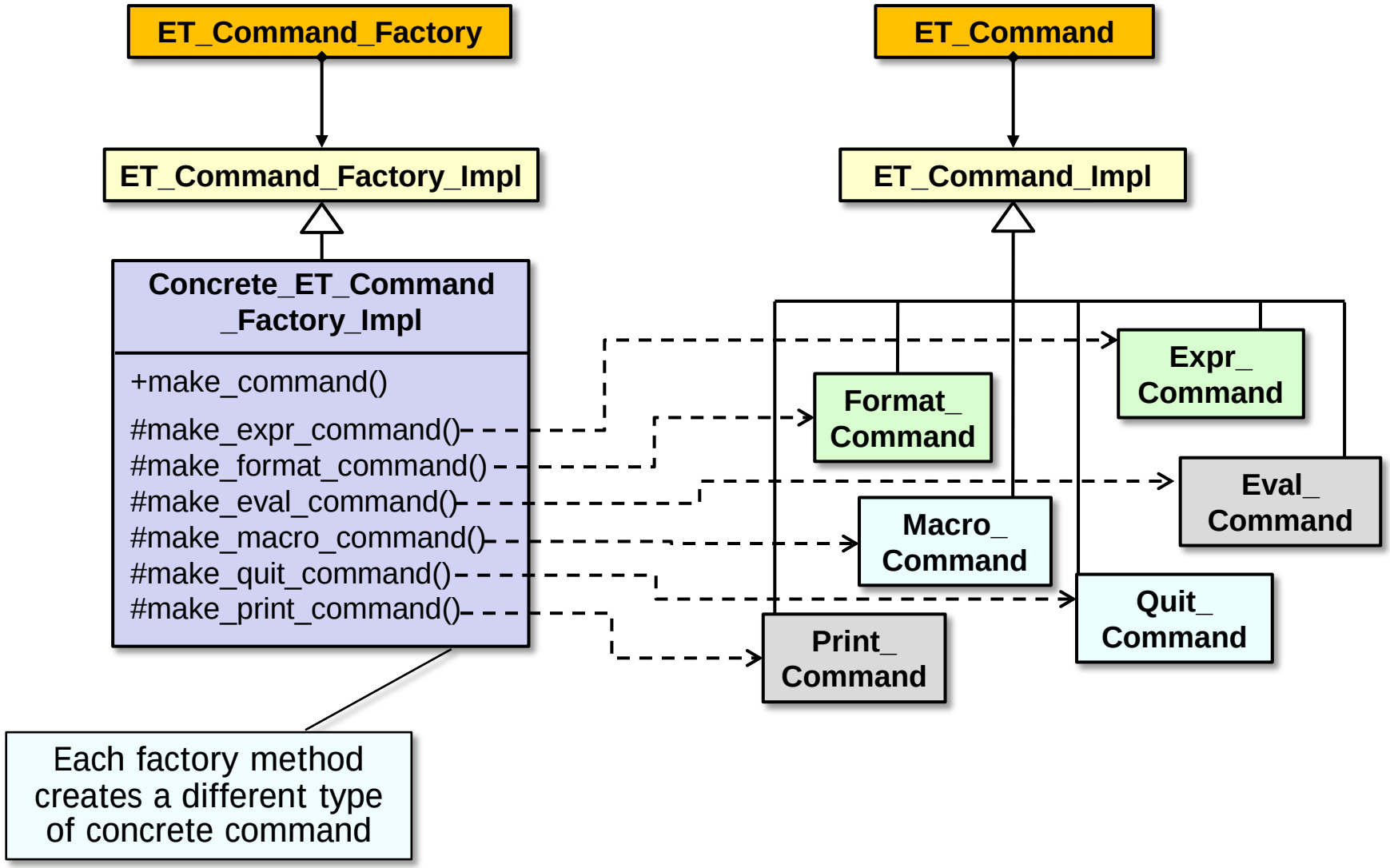
```
ET_Command_Factory_Impl(ET_Context &context)
~ET_Command_Factory_Impl()
virtual ET_Command make_command(const std::string &s)=0
...
```

Pure virtual methods

```
virtual ET_Command make_format_command(const std::string &)=0
virtual ET_Command make_expr_command(const std::string &)=0
virtual ET_Command make_print_command(const std::string &)=0
virtual ET_Command make_eval_command(const std::string &)=0
virtual ET_Command make_quit_command(const std::string &)=0
virtual ET_Command make_macro_command(const std::string &)=0
```

- **Commonality**: Provides a common interface to create commands
- **Variability**: Subclasses of this base class define expression tree command factory methods vary depending on the requested commands

Factory Structure



Bridge pattern encapsulates variability & simplifies memory management

Factory Method GoF Class Creational

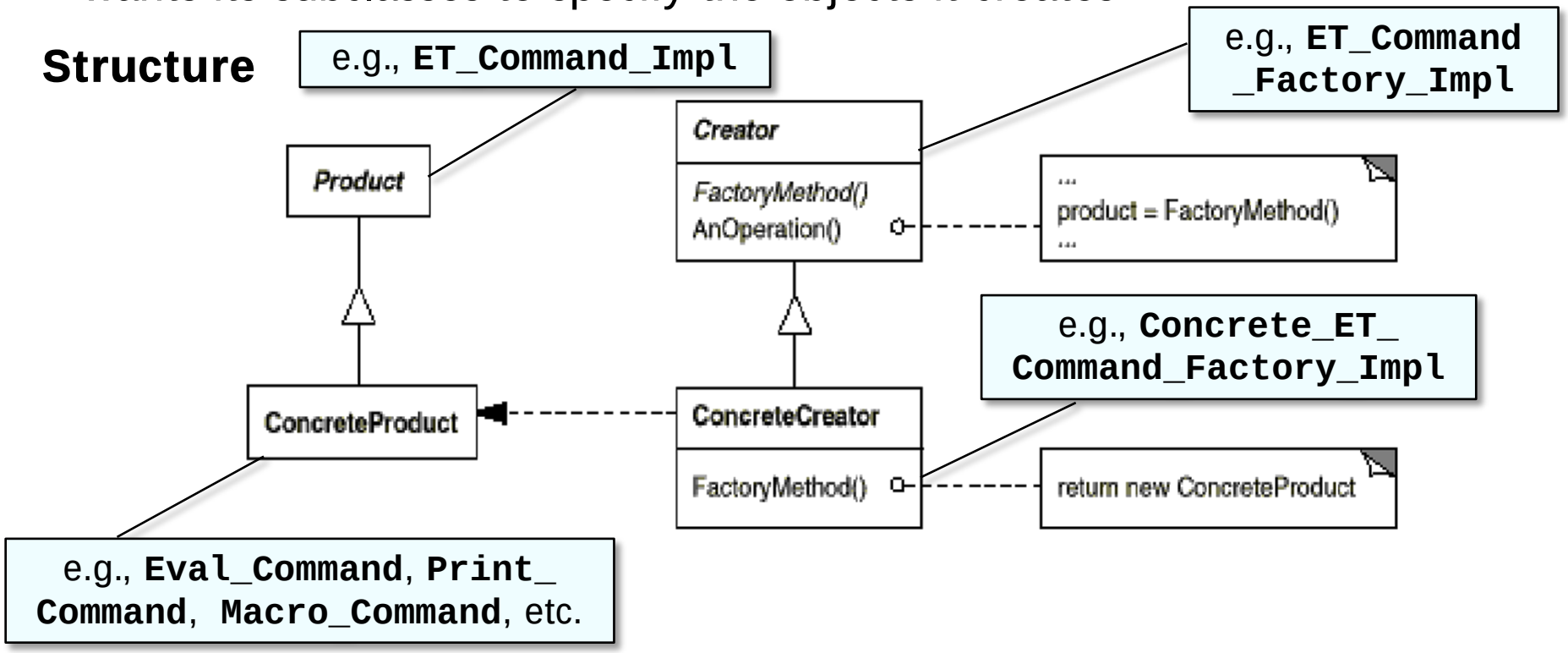
Intent

- Provide an interface for creating an object, but leave choice of object's concrete type to a subclass

Applicability

- When a class cannot anticipate the objects it must create or a class wants its subclasses to specify the objects it creates

Structure



Factory Method GoF Class Creational

Factory Method example in C++

- An interface for creating a command, letting subclass chose concrete type

```
class ET_Command_Factory_Impl {
public:
  virtual ET_Command make_macro_command(const std::string &) = 0;
  ...
```

Pure virtual method

Factory Method GoF Class Creational

Factory Method example in C++

- An interface for creating a command, letting subclass chose concrete type

```
class ET_Command_Factory_Impl {
public:
  virtual ET_Command make_macro_command(const std::string &) = 0;
  ...
class Concrete_ET_Command_Factory_Impl
          : public ET_Command_Factory_Impl {
public:
  virtual ET_Command make_macro_command(const std::string &expr) {
    std::vector <ET_Command> commands;

    commands.push_back(make_format_command("in-order"));
    commands.push_back(make_expr_command(expr));
    commands.push_back(make_eval_command("post-order"));
    return ET_Command(new Macro_Command(tree_context_, commands));
}
```

Used to implement "succinct mode"

Create vector of commands that are executed as a macro

Encapsulates command within *Bridge* abstraction object

Factory Method GoF Class Creational

Consequences

+ *Flexibility*: The client becomes more flexible by not specifying the class name of the concrete class & the details of its creation
+ *Decoupling*: The client only depends on the interface
– *More classes*: Construction of objects requires an additional class in some cases

Factory Method GoF Class Creational

Consequences

+ *Flexibility*: The client becomes more flexible by not specifying the class name of the concrete class & the details of its creation
+ *Decoupling*: The client only depends on the interface
– *More classes*: Construction of objects requires an additional class in some cases

Implementation

- There are two choices
  - The creator class is abstract & does not implement creation methods (then it must be subclassed)
  - The creator class is concrete & provides a default implementation (then it can be subclassed)
- If a factory method can create different variants the method should be passed a parameter to designate the variant

Factory Method GoF Class Creational

Consequences

+ *Flexibility*: The client becomes more flexible by not specifying the class name of the concrete class & the details of its creation
+ *Decoupling*: The client only depends on the interface
– *More classes*: Construction of objects requires an additional class in some cases

Implementation

- There are two choices
  - The creator class is abstract & does not implement creation methods (then it must be subclassed)
  - The creator class is concrete & provides a default implementation (then it can be subclassed)
- If a factory method can create different variants the method must be provided with a parameter

Known Uses

- InterViews Kits
- ET++ WindowSystem
- AWT Toolkit
- The ACE ORB (TAO)
- BREW feature phone frameworks