The Factory Method Pattern

Motivating Example

Douglas C. Schmidt
Learning Objectives in This Lesson

- Recognize how the *Factory Method* pattern can be applied to extensibly create variabilities in the expression tree processing app.

![Diagram of the Factory Method pattern in action]
Motivating the Need for the Factory Method Pattern in the Expression Tree App

Douglas C. Schmidt
A Pattern for Abstracting Object Creation

**Purpose:** Enable the extensible creation of variabilities, such as commands, iterators, & visitors.

*Factory Method* decouples the creation of objects from their subsequent use.
There are many points of variability in the expression tree processing app.
- e.g., user commands, traversal strategies, & visitor operations applied on an expression tree
There are many points of variability in the expression tree processing app.
- e.g., user commands, traversal strategies, & visitor operations applied on an expression tree.

Adding new variants should not affect existing client code.
Problem: Inflexible Creation of Variabilities

- Tightly coupling the creation of variabilities with client code is problematic.
- e.g., hard-coding lexical dependencies on specific derived classes can complicate maintenance & impede extensibility

User_Command *command = new Print_Command();

Visitor *visitor = new Evaluation_Visitor();

ET_Iter_Impl *it = new Pre_Order_ET_Iter_Impl();
• Define a `User_Command_Factory` class whose `make_command()` factory method creates a `User_Command` object.

![Diagram]

**Diagram:**
- **User_Command_Factory**
- `make_command()` method
- **User_Command** object

**Solution:** Abstract Creation of Objects
Solution: Abstract Creation of Objects

• Have the `make_command()` factory method implement the appropriate derived class of `User_Command`
Solution: Abstract Creation of Objects

- Have the `make_command()` factory method implement the appropriate derived class of `User_Command_Impl`, e.g.,
  - Subclass `User_Command.Factory` & override the factory method `make_command()`
Solution: Abstract Creation of Objects

- Have the `make_command()` factory method implement the appropriate derived class of `User_Command_Impl`, e.g.,
- Subclass `User_Command_Factory` & override the factory method `make_command()`
- Or pass a parameter to the `make_command()` factory method & use it to create the appropriate `User_Command_Impl` derived class objects
User_Command_Factory Class Overview

• Create the command corresponding to the user input.

Class methods

User_Command make_command(string inputstring)
...
User_Command.Factory Class Overview

- Create the command corresponding to the user input.

Class methods

User_Command  make_command(string inputstring)

This is a factory method
User_Command.Factory Class Overview

• Create the command corresponding to the user input.

Class methods

User_Command.make_command(string inputstring)

• **Commonality**: provides a common API to create commands
• **Variability**: implementations of expression tree command factory methods can vary depending on the requested commands
User_Command.Factory Class Overview

- Create the command corresponding to the user input.

FACTORY_PTMF

evaluate()

Each factory command object conforms to the FACTORY_PTMF typedef & creates a different type of User_Command_Impl.

std::map<string, FACTORY_PTMF>

<table>
<thead>
<tr>
<th>Command Name</th>
<th>Factory Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;expr&quot;</td>
<td>execute()</td>
</tr>
<tr>
<td>&quot;format&quot;</td>
<td>execute()</td>
</tr>
<tr>
<td>&quot;eval&quot;</td>
<td>execute()</td>
</tr>
<tr>
<td>&quot;macro&quot;</td>
<td>execute()</td>
</tr>
<tr>
<td>&quot;quit&quot;</td>
<td>execute()</td>
</tr>
<tr>
<td>&quot;print&quot;</td>
<td>execute()</td>
</tr>
</tbody>
</table>

User_Command_Impl

evaluate()

- Expr Command
- Eval Command
- Macro Command
- Print Command
- Quit Command