Template Method

Intent

- Define the skeleton of an algorithm in an operation, deferring some steps to subclasses.
- Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

Motivation

- The abstract Application class defines the algorithm for opening and reading a document in its OpenDocument operation:

  ```c++
  void Application::OpenDocument
  (const char* name) {
    if (!CanOpenDocument(name)) {
      // cannot handle this document
      return;
    }

    Document* doc = DoCreateDocument();

    if (doc) {
      _docs->AddDocument(doc);
      AboutToOpenDocument(doc);
      doc->Open();
      doc->DoRead();
    }
  }
  ```

Applicability

- to implement the invariant parts of an algorithm once and leave it up to subclasses to implement the behavior that can vary.
- when common behavior among subclasses should be factored and localized in a common class to avoid code duplication.
- to control subclasses extensions by "hook" operations at specific points.
Structure

![Diagram of AbstractClass and ConcreteClass classes]

Participants

- **AbstractClass** (Application)
  o defines abstract **primitive operations** that concrete subclasses define to implement steps of an algorithm.
  o implements a template method defining the skeleton of an algorithm.
  o The template method calls primitive operations as well as operations defined in AbstractClass or those of other objects.

- **ConcreteClass** (MyApplication)
  o implements the primitive operations to carry out subclass-specific steps of the algorithm.

Collaborations

- ConcreteClass relies on AbstractClass to implement the invariant steps of the algorithm.

Consequences

- Template methods are a fundamental technique for code reuse by factoring out common behavior.
- Template methods lead to an inverted control structure
- Template methods call the following kinds of operations:
  o concrete operations (either on the ConcreteClass or on client classes);
  o concrete AbstractClass operations (i.e., operations that are generally useful to subclasses);
  o primitive operations (i.e., abstract operations);
  o factory methods and
  o hook operations

```cpp
void ParentClass::Operation () {
  // ParentClass behavior
  HookOperation();
}
void ParentClass::HookOperation () { }
void DerivedClass::HookOperation () { }
  // derived class extension
```
• Must specify which operations are hooks (may be overridden) and which are abstract operations (must be overridden).

**Implementation**

1. *Using C++ access control.*
   - The primitive operations that a template method calls can be declared protected members. This ensures that they are only called by the template method.
   - Primitive operations that *must* be overridden are declared pure virtual.
   - The template method itself should not be overridden.

2. *Minimizing primitive operations*
   - The more operations that need overriding, the more tedious things get for clients.

3. *Naming conventions.*
   - You can identify the operations that should be overridden by adding a prefix to their names such as "DoCreateDocument", "DoRead", and so forth.

**Sample Code**

```cpp
void View::Display () {
    SetFocus();
    DoDisplay();
    ResetFocus();
}

void View::DoDisplay () { }

void MyView::DoDisplay () {
    // render the view's contents
}
```

**Known Uses**

• Template methods are so fundamental that they can be found in almost every abstract class.

**Related Patterns**

• Factory Methods are often called by template methods.
• Strategy: Template methods use inheritance to vary part of an algorithm. Strategies use delegation to vary the entire algorithm.