Templatized Primitive Method Idiom

Bosko Zivaljevic
Cisco Systems
2302 Fox Drive, Suite A
Champaign, IL 61820
bzivalje@cisco.com

ABSTRACT

Template Method Pattern (see [5]) solves the problem of the existence of a generic algorithm for a family of classes that needs specialization in each and every concrete class. It does so by implementing the algorithm in the base class and by forwarding implementation details to (pure) virtual functions. In terms of [5] these forwarding functions are called primitive.

In this paper we propose an implementation of the Template Method Pattern in C++ that addresses the question of stronger insulation of primitive functions, in the sense of [7], and of better conformance to the Open Closed Principle of Bertrand Meyers (see [9]). Mixing virtual functions and templates the Templatized Primitive Method Idiom gives an example of mixing subtype and parametric polymorphism in what we believe is an interesting and powerful idiom.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering—*patterns*

General Terms

Design

Keywords

Design patterns, Template Method Pattern, Generic programming, Templates, C++

1. INTRODUCTION

Suppose that we are building a network management tool that would facilitate the management of large networks. For instance, we might be following the architecture of the Directory Enabled Networks (DEN) and Common Information Model (CIM, see [10]). As a part of the overall design we want to model the CLI (command line interface) of managed devices (firewalls). In DEN terminology these devices are usually called the Policy Enforcement Points (PEPs).

The purpose of modeling commands is to enable manipulation of PEP configurations. A PEP configuration is a sequence of commands (in the native language of the concrete device) that shapes the traffic passing through that device.

An important functionality we want to have is the restoration of the state of a given PEP from an input stream (such as a file or a string stream). The set of commands that have configured a particular PEP can usually be obtained in a text form issuing a single command such as "`show config`", in the case of, for instance, the Cisco Secure Private Internet Exchange (PIX) firewall. The obtained configuration consist of a sequence of records with each record representing a concrete command. Without getting much into the command's semantics, a typical configuration might look like this

```
nameif e0 outside sec0
nameif e1 inside sec100
enable passwd Bob
passwd Bob
hostname lab
no failover
names
name out_addr 10.10.10.10
name in_addr 20.20.20.20
ip address outside out_addr 255.255.255.0
ip address inside in_addr 255.255.255.0
. . .
```

The above commands define hardware characteristics of some of the firewall interfaces (`inside` and `outside`), their security levels, their IP addresses etc.

To achieve our goal we need to be able to restore the state of each command given a record (string). In what is probably a typical approach to a similar problem, we create an interface (abstract) base class `AbstractDeviceCommand` for all device commands. The class might look as follows

```
class AbstractDeviceCommand
{
public :
   AbstractDeviceCommand();
   virtual ~AbstractDeviceCommand() = 0;

   // Some interface.....................
              . . .
```

```
   // Restoring the object from a string
   // stream.
   friend std::istringstream&
   operator>>(std::istringstream& iss,
              AbstractDeviceCommand& cmd);
private :

   // Primitive method for operator>>....
   virtual bool
   In(const std::string& s) = 0;
};
```

The main functionality that we are interested here is the `operator>>` (for the sake of simplicity we consider the string stream only). Given any concrete command `cmd` and a record `s` we want to parse the record and initialize the command. The `operator>>` is a global operator and a friend of `AbstractDeviceCommand`. The friendship is granted as the implementation is delegated to the private method `AbstractDeviceCommand::In`. `In` is declared as a private, pure virtual function that has no implementation in the base class. Rather, the base class mandates that every of its subclasses implements `In`. This is the Template Method pattern from [5]. In terms of [5] the `operator>>` is the template method and the member function `In` is its abstract **primitive** function. Notice that the primitive function is declared **private** (instead of, say, protected or public). Having the function declared pure virtual and private signals the client that the function is strictly used in implementation and that no default implementation is provided for it; the client inherits the obligation to define the primitive function but not its implementation. This schema (in itself a useful rule to follow) goes under the name of the Non-Virtual Interface Idiom (NVI, see [11]; the name Non-Virtual Interface Idiom is mentioned in Sutter's reply to Scott Meyers letter in the December 2001 issue of C/C++ Journal). The idiom stipulates that no public interface of a class should ever be declared virtual (pure or not). Instead, the function is declared as non virtual, thus stating its contract clearly to the clients, and its implementation forwarded to a (pure) virtual and private implementing method, thus encapsulating the implementation details (virtuality) of the concrete method.

The implementation of `operator>>` might look like this.

```
std::istringstream&
operator>>(std::istringstream& iss,
           AbstractDeviceCommand& cmd)
{
   if (command.In(iss.str()))
   {
      // Scan successful.  Clear
      // the stream.
      iss.clear();
   }
   else
   {
      // Scan unsuccessful.
      iss.setstate(std::ios_base::failbit);
   }
   return (iss);
}
```

By forwarding its implementation to the underlying primitive method `In` the `operator>>` obtains, among other things, an extra level of indirection for varying its implementation over the whole population of commands. For instance, we chose to set the state of the stream on failure or success allowing the client to check the state of the stream. In the future we might chose to do something else (throw/catch an exception, introduce more error checking, pay attention to locales etc.).

A concrete command needs to implement the `In` pure virtual function. A typical implementation looks like this.

```
// ConcreteCommand.hpp

class ConcreteDeviceCommand :
   public AbstractDeviceCommand
{
   private :
      bool In(const std::string& s);
 . . .
};

// ConcreteDeviceCommand.cpp

bool
ConcreteDeviceCommand::In(
               const std::string& s)
{
   typedef
   std::vector<std::string> Container;
   typedef
   Container::const_iterator Iterator;

   bool status = false;

   // Break string into tokens.
   Container tokens = ...;

   // The parser.....................
   ConcreteDeviceCommandParser parser;

   for (Iterator c = tokens.begin();
        c != tokens.end();
        ++c)
   {
      parser.Process(*c);
      if (parser.IsInAcceptingState())
      {
         status = true;
         *this = parser.CreateCommand();
      }
      else if (parser.IsInIllegalState())
      {
         // Throw some exception or ... ?
      }
      else { // No-op }
   }
```

```
   return status;
}
```

The implementation of the concrete device command method `In` uses a new component `ConcreteDeviceCommandParser`. The parser can process one token at a time changing its "state". If it reaches an accepting state the string has been processed successfully and we can retrieve the command from the parser. If an error state is reached we can act accordingly either throwing an exception or rising some flags. Finally, the status of success or failure is returned (used by `operator>>` to set the status of the stream).

2. THE PROBLEM

So far, the design is more or less straightforward. To model a new command one creates a class for that command inheriting from the common base class and implements all of the pure virtual functions required. In particular our `In` method would have to be implemented. To implement `In` a new parser has to be created for that command and the above code repeated. Satisfied, we even go further by packaging commands in several different packages grouping them according to some internal logic. A package `A` is responsible for one type of device configuration whereas a package `B` might be responsible for some other device configuration etc. `AbstractDeviceCommand` is placed at the top of dependency hierarchy in a separate package `T`. At least one component from every other package inherits from `AbstractDeviceCommand` and thus depends on `T`.

After a while we realize that the number of commands we have to handle is growing. The framework started with a few commands distributed in a few packages to finish with more than forty commands to model. Furthermore, in order to avoid the complexities of monolithic class interfaces, several C++ classes are used to model a single device command. Now, we are talking of close to hundred classes distributed in about a dozen of packages. For each and every command we have to repeat the boilerplate code in the method `In`. Repeating the code in itself is not such a bad thing, one would argue, if it happens only once. For example, `Clone` method from [5]'s Prototype pattern is hand rolled because it has a fixed, well-defined implementation that will never change in the future. The requirements for the parsing algorithm, however, have already changed a few times and are likely to change (even radically) in the future. Does the string `s` contain only tokens for one command or other tokens are allowed to follow? If so, do we report an error or accept the last successful command? The algorithm does not specify how the string `s` broken into tokens. It does suggest that we create a vector of tokens in one shut. For that we might use the library function `strtok`. Is that the optimal solution? Will the non-reentrancy of `strtok` hurt us at some point? In the future we might even opt for a completely different schema. For example, we might want to use iteration over string tokens based upon the STL string algorithms (see [6]), Boost tokenizer (see [4]), Boost regular expression parsing (see [3]) or an implementation based on classical lex/yac approach. Further, the enclosing framework might maintain a factory of commands creating a command to be initialized from a string on a fly. The framework might require that all parsers start from non-initial state. The other clients, on the other hand, might not have that requirement. How to accommodate them all? How to be ready to satisfy all the above partially specified requirements **after** the code in lower packages `A`, `B`,... has already been written?

3. SOLUTIONS

Being able to state our problem clearly enabled us to come up with a half of the solution, already. We have to implement the method `In` in a generic fashion. Low-level packages and components will implement the method only once without ever modifying it. At the same time we have to be able to change the parsing implementation at any point and be ready to accommodate new client's requirements that may come along painlessly. We want to allow a seamless refactoring.

Bertrand Meyers (see [9]) expressed the ability of a software module to be open for extension and closed for modification as an Open Closed Principle. Our low level components implementing the concrete device commands as well as the base class component have to be closed for modification and yet open for extension and improvement. Looking from that perspective, we have to find a solution to our problem that would conform to the OCP in the strongest possible manner.

Another equally important principle we want to follow is the separation of the implementation and the declaration of our classes, known as the insulation (see [7]). We say that the implementation of a particular feature is insulating if any further change in that feature's implementation will not force the client's recompilation (thus, the insulation is stronger than the encapsulation). Therefore, whatever the future change in the implementation of the `operator>>` is going to be its effect on the rest of the system should be minimal; only the classes that directly depend on the change and only those should be recompiled.

We shall investigate several ways of solving the above problem, starting with sticking to subtype polymorphism, through the use of policy classes to finish with function templates. We shall argue that an idiom we call here Templatized Primitive Method is the only solution that strictly conforms to the above stated principles and that is the right solution to our problem.

3.1 Using Dynamic Template Method Pattern

We have already used Template Method Pattern in implementing the `operator>>`. Let's try to use the same pattern to implement the method `In`.

We can implement the parsing algorithm in the base class delegating all the specific implementation details to the subclasses.

```
virtual bool
AbstractDeviceCommand::In(
                const std::string& s)
{
   typedef
   std::vector<std::string>
   Container;
```

```
   typedef
   Container::const_iterator
   Iterator;

   bool status = false;

   // Break string into tokens.
   Container tokens = ...;

   // The parser.....................
   typedef
   std::auto_ptr<
      AbstractDeviceCommandParser
   >
   ParserHandle;

   ParserHandle parser =
                  this->CreateParser();

   for (Iterator c = tokens.begin();
        c != tokens.end();
        ++c)
   {
      parser->Process(*c);
      if (parser->IsInAcceptingState())
      {
         typedef
         std::auto_ptr<
         AbstractDeviceCommand
         >
         CommandHandle;

         status = true;

         this->AssignFrom(
               *CommandHandle(
                  parser->CreateCommand()
                )
         );
      }
      else if (parser->IsInIllegalState())
      {
         // Throw some exception or ... ?
      }
      else { // No-op }
   }
   return status;
}
```

The new implementation is only slightly different form the one presented in the `ConcreteCommand`. We were able to keep the same logic of feeding tokens into the parser and checking for accepting and illegal states, taking the appropriate actions on both. Only two new pure virtual functions playing the role of primitive methods from Template Method Pattern had to be added to `AbstractDeviceCommand`. First, we need to construct the correct parser using `CreateParser`. Second, we need to polymorphically assign abstract command to a concrete command `AssignFrom`. This leads to the following declaration.

```
class AbstractDeviceCommand
{
public :

   . . .

private :

   // Primitive method for operator>>....
   virtual bool
   In(const std::string& s)          = 0;

   // Primitive method for creating the
   // correct parser.
   virtual AbstractCommandParser*
               CreateParser() const = 0;

   // Primitive method for assigning
   // from an abstract command.
   virtual void
   AssignFrom(
       const AbstractDeviceCommand& cmd
   )                                  = 0;
};
```

Also, we had to create an abstract base class for all parsers, say, `AbstractPixCommandParser`. The class has to have a few methods that will enable us to implement the algorithm in the base class. These are `CreateComamnd`, `IsInAcceptingState`, `IsInIllegalState` and `ProcessToken`. By following Non-Virtual Public Interface idiom all of these forward their implementation to corresponding `Do` methods.

```
class AbstractDeviceCommandParser
{
public :

   AbstractDeviceCommandParser();
   virtual
   ~AbstractDeviceCommandParser()   = 0;

public :

   // Create the command.........
   AbstractDeviceCommand*
       CreateCommand() const;

   // Accepting state query.....
   bool IsInAcceptingState() const;

   // Illegal state query........
   bool IsInIllegalState() const;

   // Process a token............
   void Process(const std::string& s)

public :

   // Primitive methods for the above
   // public interfaces.  We follow NVI.
```

```
   virtual AbstractDeviceCommand*
       DoCreateCommand() const        = 0;

   virtual bool
      DoIsInAcceptingState() const    = 0;

   virtual bool
      DoIsInIllegalState() const      = 0;

   virtual void
      DoProcess(const std::string& s) = 0;
};
```

Creating a new command has to be dynamically dispatched so we needed pure virtual `DoCreateCommand`. The same is true for queries for the accepting and illegal states. In addition, we need a pure virtual method for processing tokens. Finally, our algorithm needed some king of dynamic memory management in the local scope; thus the use of `std::auto_ptr`. The situation does not seem so bad at all. With a little lack our framework could survive a long time.

Looking back at our design, however, we notice a few apparently small details that starts bothering us.

First, the dynamic allocation. In the method `CreateCommand` we had to require the allocation of `AbstractDeviceCommand` on the heap and in the method `CreateParser` we had to do the same with the class `AbstractDeviceCommandParser`. Dynamic allocation is expensive and, (although, as they say, we should not think about the performance so early in the design) we are left we a certain degree of uneasiness. We might address the dynamic allocation of these object later in the design; create some custom-made memory management for the parser; maybe even a `Clone` base factory for it, etc.

Second, we had to assign an abstract command in `AssignFrom`. Let see how we can implement this.

```
// Assign from another concrete
// command.
void
ConcreteDeviceCommand::AssignFrom(
     const AbstractDeviceCommand& cmd)
{
   // Check if the source is of the
   // correct type.
   if (std::typeid(*this)
       ==
       std::typeid(cmd))
   {
      *this =
         dynamic_cast<
            ConcreteDeviceCommand&
         >(cmd);
   }
   else { // Now what?...   }
}
```

We first check if the assignee is of the same type as the assigned command. If it is, we dynamically cast to the concrete command type and call the concrete command assignment operator. But what should we do if the argument command is not of the same type as the caller's command? Do we do nothing or do we need to throw an exception? Assigning from a different type should have been a compilation error in the first place! The dynamic cast is also a weak point. First, we have to pay for it in CPU cycles, and then we have to worry if it fails. One can argue that the cast should not fail; we have just checked that the argument type is the right one. Still, we could have typed in a different `AbstractDeviceCommand`'s child and the compiler would not warn us about that. A little better implementation could just check for the right type once.

```
// Assign from another concrete
// command.
void
ConcreteDeviceCommand::AssignFrom(
     const AbstractDeviceCommand& cmd)
{
   ConcreteDeviceCommand* p
      = dynamic_cast<
            const ConcreteDeviceCommand*
        >(&cmd);

   // Check if the source is of the
   // correct type.
   if (p != 0)
   {
      *this = *p;
   }
   else { // Now what?...}
}
```

We do the dynamic cast first and if we succeed we continue with the assignment. However, we pay the CPU price of casting even in the case when the simple `typeid` would detect mismatch types. Another thing is that the code has to be repeated in each and every of our concrete commands. It would be nice if we did not have to polymorphically assign the object in the first place.

How resilient is our design to the change? Our algorithm is implemented in one place only. It seem that we can accommodate any new requirement easily. For instance, we can chose to process the tokens until we reach the first accepting state instead of the last (command's parser can go through more than one accepting state scanning a single command), or we can decide to process the whole record using the lastly accepted input etc. The big question is what if we decide to improve our parser component? In fact we can decide at some point to eliminate hand rolled parser completely and use some third party solution. Suddenly we realize that our declaration of `AbstractDeviceCommand` has a member function `CreateParser`, albeit private, which returns a pointer to `AbstractDeviceCommandParser`. To change that we have to modify the header file where the command was declared! Finally, we realize that the schema with the parser components

is to be used exclusively in the method `In`'s **implementation** and should not be exposed as a part of the command interface (again, in spite of the fact that that interface is private). Ideally, header files should not contain the information about the implementation details of the classes, at all.

Further, the algorithm in the base class is "do all or nothing". If a concrete command has to parse itself radically differently from the rest of the commands it would have to override the `In` method completely. For example, in the middle of the development we realize that some commands should keep a special environment of maps between IP addresses and their aliases and take those into account when parsing. The change demands radical modification of the parsing algorithm. Yet, the number of commands to be parsed with the environment is about a half of the total number of commands. For one half we have a nice algorithm in the base class and for the other we have to override the method `In` repeating the same code; exactly the problem we wanted to avoid. We can try to continue to play on the card of subtype polymorphism by introducing some more intermediate classes (and/or virtual methods), resolving the new parsing algorithm by applying Template Method Pattern again. Indeed, that sounds plausible, assuming, again, that the change does not happen too often. Alas, soon we discover that some negative forms of commands (every command more or less can be negated by inserting "no" in front of it) have to be parsed differently than their positive counterparts. Now we have to take into account two types of deviations from the general algorithm that can both interact with each other.

The problem is that we are not insulated enough from the implementation details of the primitive method `In` (for the discussion of insulation again see [7]) and that the base class is still trying to do too much without knowing the specifics of the children classes.

3.2 Writing a policy class

Rejecting the Template Method Pattern and dynamic polymorphism we realize that the implementation of the method `In` has to "magically" appear in each and every concrete command with the appropriate parameters, such as the parser and the concrete command, replaced with the correct types. The right way to achieve this is to use templates. Our first attempt is to templetize the class `AbstractDeviceCommand` with the parser and command types. In fact, we could not do exactly that because the parser instances differ from one command to another and we would loose the common base class for all commands. Instead, we need an intermediate templated class `ParsingPolicy`.

```
template <typename COMMAND,
          typename PARSER>
class ParsingPolicy
{
public :

   . . .

private :

   // The implementation for In........
   COMMAND InImpl(const std::string& s);
};
```

`ParsingPolicy` is templetized with the parser and the concrete command. The declaration of a concrete command now looks as follows

```
class ConcereteDeviceCommand :
    public AbstractDeviceCommand,
    private ParsingPolicy<
              ConcreteDeviceCommand,
              ConcereDeviceParser
            >
{
   . . .
};
```

In addition to inheriting publicly from the `AbstractDeviceCommand` we inherit privately from the parsing policy class instantiated with the command parser and the concrete device command class we are declaring. This last technique is a well know parametric inheritance idiom used in several places in the last decade staring with [2] and lately and more extensively by Alexandrescu (see [1]) in his design of policy classes.

The abstract policy base class has a method `InImpl` that implements the method `In`. It is very similar to the original `In` method with the only difference that we do not assign `this` object but merely return the newly created command (otherwise, the ghost of polymorphic assignment would reappear).

```
template <typenema COMMAND,
          typename PARSER>
COMMAND
ParsingPolicy<COMMAND, PARSER>::InImpl(
   const std::string& s)
{
   typedef
   std::vector<std::string> Container;
   typedef
   Container::const_iterator Iterator;

   // Break string into tokens.
   Container tokens = ...;

   // The parser.....................
   PARSER parser;

   // The command....................
   COMMAND cmd;

   for (Iterator c = tokens.begin();
        c != tokens.end();
        ++c)
```

```
   {
      parser.Process(*c);
      if (parser.IsInAcceptingState())
      {
         cmd = parser.CreateCommand();
      }
      else if (parser.IsInIllegalState())
      {
         // Throw some exception or ... ?
      }
      else { // No-op }
   }
   return cmd;
}
```

The implementation of a concrete command method `In` now looks like this.

```
ConcreteDeviceCommand::In(
               const std::string& s)
{
   *this =
      this->InImpl<
         ConcreteDeviceCommandParser,
         ConcreteDeviceCommand
         >;
}
```

Inheriting from a properly instantiated policy base class we inject the proper functionality, namely the method `InImpl` into the host class, and simply use it in the implementation of `In`.

Notice that we could also defined `InImpl` as a template member function of `AbstractDeviceCommand`. However, there are several problems with that decision implied, more or less, by the rift between the current state of standard C++ implementation and our desire to write properly insulated classes. First, not all compilers support the template member functions. Second, even if a particular compiler supports template member functions the implementation is more likely than not, an intrusive one. The C++ standard stipulates that there should be two models of compilation of templates : the inclusion and the separation model. The inclusion model, supported by all of the current compilers, expends templates inline. The separation model allows the separation of the template definitions and template declarations. Now days, the inclusion model is almost the only one supported in practice. Furthermore, if we write a template member function for an otherwise non-templetized class, more likely than not the class itself would had to be written inline (at least this is the case with most popular Microsoft C++ compiler). That means that our base class for all commands would be forced to be inlined thus completely non-insulating. For a large project that we have at hand that's probably not acceptable.

3.3 Writing a function template - The Templatized Primitive Method Idiom

The policy based solution of the previous section is perfectly fine. We have achieved the things that ordinary Template Method Pattern could not. We have insulated the base class of all commands from implementation details of the method `In`. Still, we can do a little better. The problem is that we used **inheritance** to implement a policy. Inheritance, as the strongest dependency relationship we can have between the classes, has its price. What if we decide to change the implementation of our algorithm? As each command inherits from the policy class the command header files are affected. That means recompiling all of the components that depend on the command classes declarations. This might not be acceptable, especially if it can be avoided.

A better solution in our case is not to use the inheritance at all. We can formulate the parsing policy in an ordinary function template.

```
template <tyepnema PARSER,
          typename COMMAND>
COMMAND Scan(const std::string& s)
{
   typedef
   std::vector<std::string> Container;
   typedef
   Container::const_iterator Iterator;

   // Break string into tokens.
   Container tokens = ...;

   // The parser.....................
   PARSER parser;

   // The command....................
   COMMAND cmd;

   for (Iterator c = tokens.begin();
        c != tokens.end();
        ++c)
   {
      parser.Process(*c);
      if (parser.IsInAcceptingState())
      {
         cmd = parser.CreateCommand();
      }
      else if (parser.IsInIllegalState())
      {
         // Throw some exception or ... ?
      }
      else { // No-op }
   }
   return cmd;
}
```

The implementation of every concrete command method `In` is as before.

```
ConcreteDeviceCommand::In(
               const std::string& s)
{
   *this =
      Scan<ConcreteDeviceCommandParser,
           ConcreteDeviceCommand>;
```

```
}
```

We have finally reached a kind of a solution we were looking for. Let's see how it addresses the problems we had with the Template Method Pattern.

First, all the benefits of having the algorithm in one place, found in Template Method Pattern, remain. To implement new requirements we modify only the template `Scan`. Our design is open for extension. Second, we do not need to modify any of our command component to achieve this. Our design is closed. Third, we have a completely insulated solution. The only components that have to be recompiled when a new algorithm arrives are only the ones that implement the commands itself and only those. Fourth, the variety of algorithms painlessly scale in an open-closed fashion. If a number of commands have to implement a radically new algorithm for parsing all we need to do is to come up with a new `Scan` template and use it. For example, earlier we had a problem with commands that need to be implemented using an environment. Assuming that we have created the appropriate algorithm defined in a new templated function `ScanWithEnvironment` we implement the `In` method satisfying the Open-Closed Principle as usual.

```
ConcreteDeviceCommand::In(
               const std::string& s)
{
   *this =
      ScanWithEnvironment<
         ConcreteDeviceCommandParser,
         ConcreteDeviceCommand
      >;
}
```

How do we deal with the problem of negated commands that parse differently than their positive form? All we need a new parser `NoConcreteDeviceCommandParser` for negated commands and some kind of dispatching between the two cases.

```
ConcreteDeviceCommand::In(
               const std::string& s)
{
   if (this->IsNegated())
   {
     *this =
         Scan<
            NoConcreteDeviceCommandParser,
            ConcreteDeviceCommand
         >;
   }
   else
   {
      *this =
         Scan<
            ConcreteDeviceCommandParser,
            ConcreteDeviceCommand
         >;
   }
}
```

Furthermore, a powerful tool of function template specialization is at our disposal. By specializing the implementation of the `Scan` for concrete commands and concrete parser we can change the implementation of the method `In` and ultimately `operator>>` without even touching the source files where the concrete command is implemented (provided that the specialization is "visible" to the concrete command component, e.g., if it resides in a header file that included the definition of the initial algorithm). Everybody would agree that this is an ultimate form of Meyer's Open closed Principle.

4. THE TEMPLATIZED PRIMITIVE METHOD PATTERN

We shall now formulate the Templatized Primitive Method Idiom in a form a pattern formalizing the actors, specifying its applicability and discussing its usage.

4.1 Context

As described in [5], the **Template Method Pattern** defines a skeleton of an algorithm deferring some steps subclasses. The steps are implemented by **primitive** methods.

4.2 Intent

Define the primitive methods of the **Template Method Pattern** using free standing template functions.

4.3 Motivation

The standard implementation of the **primitive** functions in the **Template Method Pattern** as dynamically dispatched methods can sometimes be too rigid and cumbersome. Better compliance to the Open Closed Principle and better insulation is sometimes needed (see **Introduction**).

4.4 Participants

The following are the participants in the Templatized Primitive Method Pattern.

- The base class `Base`.

- The "skeleton" member function `Base::Algorithm` (in our case the algorithm was `operator>>`.

- A set of primitive, member functions `DoWork_1,...,DoWork_n` that implement the `Algorithm` (in our case we have the method `In` as a single primitive functions).

- The concrete class `Concrete` that is a child of the `Base` and that implements the primitive methods `DoWork_i` (the role played by `ConcreteDeviceCommand` in our introduction).

- The templated free functions `DO_WORK_1, ..., DO_WORK_n` that implement the primitive functions `DoWork_1`,..., `DoWork_n` (this corresponds to our `Scan` template from the introduction).

- The parameter classes `P_1,...,P_k` of the templated functions `DO_WORK_i` (in our case `ConcreteDeviceParser`).

4.5 Collaborations

The "skeleton" method `Base::Algorithm` is implemented by combining the primitive functions `DoWork_i` just like in the standard Template Method Pattern.

```
ReturnType
Base::Algorithm(Type_1 p_1,...,Type_k p_k)
{
   this->DoWork_1(p_1,..., p_k);
   ...
   this->DoWork_n(p_1,..., p_k);
}
```

The primitive functions `DoWork_i` are implemented in each and every subclass by instantiating the template functions `DO_WORK_i` with appropriate, concrete types.

```
ReturnType
Concrete::DoWork_i(Type_1 p_1,..., Type_k p_k)
{
   return DO_WORK_i(p_1, ..., p_k);
}
```

4.6 Applicability

One can say that in the case of `i = 1`, i.e., when only one primitive function participates in the algorithm implementation, the Templatized Primitive Method Pattern is just a dynamically dispatched templated function. The template functions `DO_WORK_1` might as well had been used standalone. If one needs a dynamic dispatch, as it was the case in our case, or when more than one primitive function is involved, then the combination of the two was the right thing to do.

The pattern is applicable in any situation where no special private access is needed from both the `Concrete` class and parameter classes. If the algorithm needs access to private interface of either `Concrete` or parameter classes the pattern cannot be applied (it is not possible to grant friendship to template functions). The danger of this is that one might be tempted to declare some of the interfaces of the base, child and/or parameter classes public in order to make them accessible for the algorithm, thus braking the encapsulation.

4.7 Known Uses

Angelika Langer and Klaus Kreft used the pattern (without naming it) to produce a generic implementations of inserters and extractors (see [8], page 170). They isolated the common parts of `operator<<` and `operator>>` in two template functions `g_inserter` and `g_extractor` which are called in the operators implementations. In turn, those call concrete, dynamically dispatched methods that do actually extracting/inserting work. If we apply this approach to our introductory example we would obtain an interesting "nested" mix between the subtype and parametric polymorphism.

The Cisco Secure Policy Manager for PIX used the pattern to define the extractors for PIX commands (as described in the introduction).

5. ACKNOWLEDGMENTS

I am grateful to Toni Marinucci for thorough reading of the paper and for many useful suggestions and remarks.

6. REFERENCES

[1] A. Alexandrescu. *Modern C++ Design : generic programming and design patterns applied*. Addison Wesley, Reading, Massachusetts, 2001.

[2] J. Barton and L. Nackman. *Scientific and Engineering C++*. Addison Wesley, Reading, Massachusetts, 1994.

[3] Boost. *http://www.boost.org/libs/regex/index.htm*.

[4] Boost. *http://www.boost.org/libs/tokenizer/index.htm*.

[5] E. Gamma, R. Helms, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley, Reading, Massachusetts, 1995.

[6] J. Hyslop and H. Sutter. Conversations : Al-go-rithms. *C/C++ Users Journal, Experts Forum*, http://www.cuj.com/experts/, October 2001.

[7] J. Lakos. *Large Scale C++ Software Development*. Addison Wesley, Reading, Massachusetts, 1996.

[8] A. Langer and K. Kreft. *Standard C++ IOStreams and Locales*. Addison Wesley, Reading, Massachusetts, 1999.

[9] B. Meyers. *Object-Oriented Software Construction*. Prentice Hall, Upper Saddle River, NJ, 1998.

[10] J. Strassner. *Directory Enabled Networks*. Macmillan Technical Publishing, Indianapolis, IN, 1999.

[11] H. Sutter. Virtuality. *C/C++ Users Journal*, pages 53–58, September 2001.