Efficient Decorator Pattern Variants through C++ Policies

Virginia Niculescu
Faculty of Mathematics and Computer Science, Babeş-Bolyai University, 1 M. Kogalniceanu, Cluj-Napoca, Romania
vniculescu@cs.ubbcluj.ro

Keywords: Decorator, Patterns, Templates, Policy, Accessibility, Extensibility, Interfaces, C++.

Abstract: C++ Policies represent an interesting metaprogramming mechanism that allows behavior infusion in a class. In this paper, we will investigate the possibility of using them for the implementation of some extended Decorator pattern variants. For the MixDecorator variant, policies are used to simulate extension methods, which are needed for implementation. Beside this, other two alternatives for Decorator pattern are proposed: one purely based on inheritance, and another that is a hybrid one; the last one wraps an object with decorations defined as a linear hierarchy of classes – introduced using policies. This is possible since a policy introduces a kind of recursive definition for inheritance. The advantages and disadvantages of these variants are analyzed on concrete examples. The hybrid variant presents many important advantages that qualify it as a valid and efficient Decorator pattern variant – HybridDecorator.

1 INTRODUCTION

C++ Policies could be considered a very interesting and useful metaprogramming mechanism that allows behavior infusion in a class through templates and inheritance (Alexandrescu, 2001; Abrahams and Gurtovoy, 2003). They have been also described as a compile-time variant of the Strategy pattern. Instead of parameterizing the class through a strategy object able to solve a required problem, the solving strategy is introduced through a template parameter from which the class also is inherited from. In addition, policies could be very useful as a mean for specifying implementation of extended interfaces. We intend to introduce here another interesting usage of the policies, related to static implementations of Decorator pattern and its variants.

Decorator and Strategy patterns provide both alternative designs to address problems that involve changing or extending the objects' functionalities, possibly dynamically (Gamma et al., 1994; Pikus, 2019; Shalloway and Trott, 2004). Strategy is a behavioral pattern that encapsulates a family of algorithms, and make them interchangeable; it changes an object from inside. On the other side, Decorator, classified as fundamental design pattern in (Zimmer, 1995), is considered a structural design pattern; but since it is used to dynamically attach additional responsibilities to an object it also affects the behaviour of that object.

The reason for this investigation is given also by the fact that static definitions are considered in general more optimal since the number of operations to be executed at the runtime is reduced, but the main advantage is given by the enhanced functionality. In the classical definition of Decorator pattern, the initial object is wrapped with several "decorations" that not only could change the functionality of the initial object interface but also, they could enlarge its interface with new methods. This extension of the interface brings some problems related to the accessibility of the new introduced methods. This have been analysed in (Niculescu, 2015) and the solution for this was expressed as a new enhanced variant of the Decorator pattern named MixDecorator. The implementation of MixDecorator requires (if we want to allow extensibility in time, which means adding new decorations with the same accesibility properties) a form of extension methods mechanism to be existent into the implementation language. Implementation solutions were given for Java – through interface default methods (Oracle, 2018), and in C# – through extension methods (Microsoft Contributors, 2015). Since in C++ there is not such an extension methods mechanism (Stroustrup, 2018), we proposed here a C++ solution based on policies.

The paper also presents other two variants for Decorator implementation, which could be considered as compile-time variants of the Decorator pattern. One purely based on inheritance, and another that is a hybrid one, which wraps an object with decorations that are defined as a linear hierarchy of classes.

The paper is structured as follows: Section 2 succinctly describes what policies means in C++ together with their applicability. Section 3 presents the classical Decorator pattern with the associated challenges, and the MixDecorator solution. In section 3.1 a C++ implementation of the MixDecorator pattern is explained, and Section 4 analyses the two new proposed variants of the Decorator pattern. Conclusions and future work are presented in section 5.

2 C++ POLICIES

Generally, a policy is defined as a class or class template that defines an interface as a service to other classes (Alexandrescu, 2001). A key feature of the policy idiom is that, usually, a template class will derive from (make itself a child class of) each of its policy classes (template parameters).

More specifically, policy based design implies the creation a template class, taking several type parameters as input, which are instantiated with types selected by the user – the policy classes, each implementing a particular implicit interface – called policy, and encapsulating some orthogonal (or mostly orthogonal) aspect of the behavior of the template class.

Through policies the code is minimized, while still allowing solutions for specific class related functionalities. In (Alexandrescu, 2001) it is stated that policy-based class design fosters assembling a class with complex behavior out of many little classes (called policies), each of which taking care of only one behavioral or structural aspect. As the name suggests, a policy establishes an interface pertaining to a specific issue. It is possible to implement policies in various ways as long as the policy interface is respected. Policies have been also described as a compile-time(static) variant of the Strategy pattern.

For example, if we need to enlarge the capacity of a container when a new element is added, we can increase it with one or with half of the existing size (or other strategies could be used, too). The strategy will be chosen depending on the specific usage of the container – Listing 1.

```
template < class T >
class incrementEnlarger {
    protected:
        void enlarge(int &capacity) {
            capacity++;
        }
}
template < class T >
class halfEnlarger {
    protected:
        void enlarge(int &capacity) {
            capacity += capacity / 2;
        }
}
template < typename EnlargePolicy >
class Container :
            protected EnlargePolicy{
    ...
    protected:
        void doEnlarge() const {
            enlarge(capacity);
            ...
        }
    ...
};
```

Listing 1: An example of using policies.

Policies have direct connections with C++ template metaprogramming and they have been used in many examples; classical examples are: `string` class, I/O streams, the standard containers, and iterators (Abrahams and Gurtovoy, 2003).

3 DECORATOR PATTERN

In (Gamma et al., 1994) book the Decorator pattern is described as a pattern that provides a flexible alternative to using inheritance for modifying behavior. By using it we may add additional functionalities or change the functionality of an object. This is a structural design pattern used to extend or alter the functionality of objects by wrapping them into instances of some decorator classes. Figure 1 presents the structure of its solution. It is generally agreed that decorators and the original class object share a common interface, and different decorators can be stacked on top of each other, each time adding new functionality to the overridden method(s).

Still, there are many situations when the decorators need to add also new methods (i.e. methods that are not in the initial object interface) that bring additional behavior; a very well known example is represented by the Java decorator classes for input and output streams.

Since the interface `IComponent` defines only the initial set of operations, the new added methods are accessible only if they belongs to the last added decorator, and the object is used through a reference of that decorator type.

Also, removing and adding decorations at the running time is a requirement stated into the classical pattern definition. Adding decorations is not difficult, since they could be added on top. If we understand by removing decorations that we can remove the top decoration, then this is only possible if a method that re-

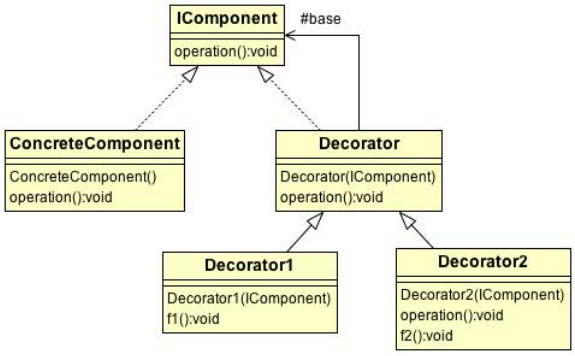

Figure 1: The Class Diagram of the Classical decorator Pattern.

turns the wrapped object is defined in the `IDecorator` interface (e.g. `getBase()`). But if we would like to remove a decoration, which is in the middle of the decoration chain, this is not a trivial task.

3.1 MixDecorator Pattern with Policies

MixDecorator is an enhanced version of Decorator pattern that eliminates some constraints of the classical pattern – most importantly the limitation to initial interface. In the same time, it could also be seen as a general extension mechanism (Niculescu, 2015).

The structure of the MixDecorator is similar to that of the classical Decorator pattern but with some important differences. Figure 2 shows the UML class diagram of the MixDecorator solution.

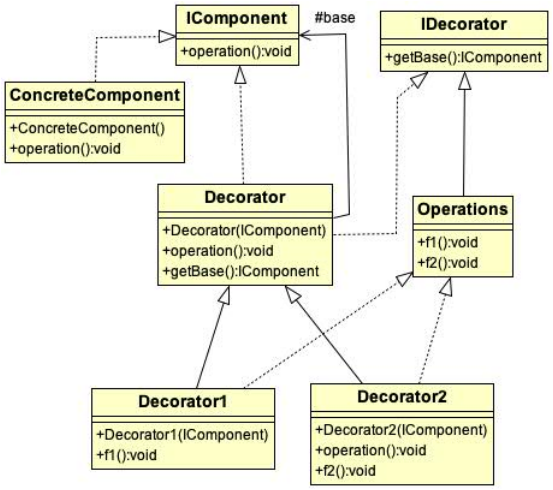

Figure 2: The Class Diagram of the MixDecorator Pattern.

As for the classical decorators, the subject (a concrete component) is enclosed into another object – a decorator object, but the decorator has an interface that could extend the general component interface. This means that the decorators could define new public methods, which could be refered to as interface-responsibilities.

In addition to the classical Decorator structure, there is a special class `Operations`, which defines methods that correspond to all new interface-responsibilities defined in all decorators. As it can be seen from Figure 2, the concrete decorator classes `Decorator1, Decorator2` are derived from `Decorator` but also from `Operations`. The implementation of an `Operations` method hides a recursion that tries to call the method with the same name from the enclosed object, and if this is not available, it goes further to the previous decoration. The recursion stops either when the concrete method is found or when it arrives to an undecorated component.

Another possible variant would be to consider `Operations` as a class derived from `Decorator`, and then all concrete decorators extend only `Operations` class – the problem with this approach is related to the difficulties in extending the set of newly added interface-responsibilities.

For a particular application/framework, after the new interface-responsibilities are inventoried, then a particular class `Operations` could be defined. But, in time, it is possible that new decorators with new methods are required to be defined. For example, considering the diagram presented in Figure 2 it would be possible to introduce a new decorator – `Decorator3` that defines a new method `f3`; this requires an extension of the class `Operations` – `OperationsExtended` that defines the method `f3()`, as well. The integration of this kind of extension of the `Operations` class represents the implementation challenge of the MixDecorator pattern.

```
1  template <typename B>
2  class Decorator: public B
3  {
4   protected:
5     IComponent* c;
6   public:
7     Decorator(IComponent* c) {
8        this->c = c;
9     }
10    IComponent* getBase() {
11       return c;
12    }
13    void operation() {
14       c->operation();
15    }
16 };
```

Listing 2: `Decorator` class definition based on a policy.

A simple extension through inheritance of the `Operations` interface that will be implemented by the

new decorators is not enough since we also have to allow direct access to all responsibilities independently of the order in which decorators are added. If, for example, a previously defined decorator is added on top, the new added responsibilities would not be accessible. In Java, or C# we may use default interface methods, respectively extension methods, to overcome this requirement.

Because in C++ it is not possible to define extension methods or other similar mechanisms, a solution is to use policies in order to postpone the specification of the parent class for decorators.
The `Decorator` class is defined as a template class, and the template parameter is also used as a parent class for the `Decorator` class – Listing 2. The class `Operations` is needed to be defined as a parent class of the `Decorator` class, but this relation will be defined through the template parameter. The policy that is introduced this way specifies which kind of `Operations` to be used. So, we may postpone the specification of the base class, and allow this base to be either `Operations` or `OperationsExtended`.

```
template <typename B>
class Decorator1: public Decorator<B>
{
 public:
    Decorator1(IComponent* c) :
    Decorator<B>(c) {    }
    virtual void f1() {
        cout<<"f1"<<endl;
    }
};
```

Listing 3: An example of a concrete `Decorator` implementation.

The concrete decorators are also defined as template classes, since they are derived from the `Decorator` class – Listing 3 presents the implementation for `Decorator1`.

```
class Operations : public IComponent,
                   public IDecorator
{
 public:
    Operations() { }
    virtual void f1() {
        IComponent* c = getBase();
        Operations* dc =
           static_cast<Operations*>(c);
    if (dc != NULL) dc->f3();
    }
    virtual void f2() {
        // similar to f1
 }
};
```

Listing 4: `Operations` class in C++ implementation.

The class `Operations` implements the `IDecorator` interface and defines the corresponding searching methods for the newly defined methods in the concrete decorators (Listing 4).

The `Decorator` class inherits `IComponent` and `IDecorator` through `Operations` class. (By letting `Decorator` to specify directly the inheritance from `IComponent` and `IDecorator` then virtual inheritance would be needed, and in this case `static_cast` should be replaced with `dynamic_cast`.)
The class `Operations` could be extended with classes that define new methods that corresponds to the new responsibilities added into additional decorators. For example, if we have a new decorator `Decorator3` that defines a new method `f3()`, than a new class `OperationsExtended` that extends `Operations` is defined, and when using this new decorator we need to specify `OperationsExtended` as a parameter for `Decorator` (Listing 5).

```
class OperationsExtended:
              public Operations
{
public:
 OperationsExtended() { }
 virtual void f3() {
  IComponent* c = getBase();
  OperationsExtended* dc =
 static_cast<OperationsExtended*>(c);
  if (dc !=NULL)  dc->f3();
 }
};
```

Listing 5: Extending the `Operations` class in C++ implementation.

```
void main() {
 IComponent* c=
  new ConcreteComponent();
 Decorator<Operations> *dc =
   new Decorator1<Operations>(c);
 dc->f1();
 Decorator<OperationsExtended>*
 dc13 =
 new Decorator1<OperationsExtended>(
  new Decorator3<OperationsExtended> ↩
                          (c)   );
 dc13->f3();
 dc13->f1();
 dc13->operation()
}
```

Listing 6: Testing different methods' calls in C++ implementation.

The example in Listing 6 shows first a decoration with `Decorator1` (for which the class `Operations` is enough to be used), and then `Decorator3` is added (for which we have the correspondent class `OperationsExtended`). Since `OperationsExtended` ex-

tends `Operations` the function `f3()` could be called even after `Decorator1` was added.

Because the template instantiation leads to the creation of a new class this is a statical efficient approach.

4 DECORATOR VARIANTS BASED ON POLICIES

The principal intent of the Decorator pattern is to avoid the class explosion when there are many combination of a set of functionalities/responsibilities.

In the classical Decorator definition, composition is used instead of inheritance as a mean of avoiding the class explosion; but if we can avoid this explosion while still using inheritance, this should not be excluded.

In order to better explain the proposed solutions, and also to link to a real example, we will consider a concrete example on which we will analyze the possibilities to define a kind of Decorator based on template inheritance.

Case Study: [ReaderDecorator]. We consider a case when we intend to define *text analyzer* decorations, which decorate a `Reader` (an object that could retrieve from a text stream, a single character, an entire line, or a specified number of characters). There are examples of such readers in Java and also in C# - (Java: the class `Reader`; C# - the class `StreamReader`). The basic method of the Reader is `readChar()` that extracts a character from the associated stream and returns it. Some basic decorations that could be provided are:

1. `CharDecorator` – a decoration that is able to count the number of already read characters; this defines an attribute `no_of_chars` that will be updated by the overridden `readChar()` method, and defines a method `getNoChars()` that returns the value of `no_of_chars`.
2. `WordDecorator` – a decoration oriented to words that provides a method `getNoWords()` that returns the number of already read words) (optionally the class could stores all the read words into an internal list, which could be returned).
3. `SentenceDecorator` – a decoration oriented to sentences that provides a method `getNoSentence()` (optionally the class could stores all the read sentences into an internal list, which could be returned).

We will consider a general `IReader` interface that defines a method `readChar()`, and a concrete class `StringReader` that uses strings as sources. For other types of sources, correspondent `IReader` specializations could be defined. The use of decorations is appropriate since if we do not need to read or count sentences, we don't have to add the `SentenceDecorator`, and similarly for `WordDecorator`, or `CharDecorator`.

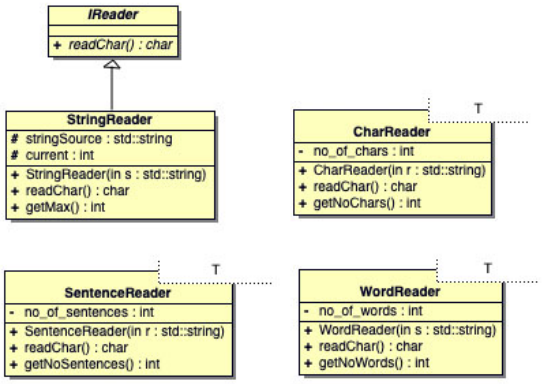

Figure 3: The Class Diagram of the Template Inheritance Solution Applied to *Reader* Problem.

When needed, other decorations could be added: counting the number of proper nouns, the number of non-blank characters, the number of sections, chapters, etc.

4.1 Inheritance based Solution

The simplest way to assure the possibility to create an object that offers any combination of the three responsibilities enumerated before is to create three template classes that are derived from their template parameter.

* This means that each decorator is defined using a policy – that should be either a `IReader` or other decorator.

```
1  template<typename T>
2  class CharReader : public T {
3    private: int no_of_chars;
4    public:
5     CharReader(std::string r) : T(r){
6      no_of_chars=0;
7     };
8     char readChar() {
9       char c = T::readChar();
10      if (c!=0) no_of_chars++;
11      return c;
12    }
13   int getNoChars() {
14      return no_of_chars;
15   }
16 }
```

Listing 7: Inheritance based solution – Definition of `CharReader` class.

In this way, a recursive composition of the decorators is possible. Even if there is no explicit constraint regarding the template type, it is implicitly assured that

```
CharReader <
 SentenceReader <
  WordReader <StringReader >>> *wr =
  new
   CharReader <
   SentenceReader <
        WordReader <StringReader >>>
          ("Happy Birthday!");

 while ((c= wr->readChar()) != 0)
   cout << c;

 cout <<wr->getNoWords() <<endl;
 cout <<wr->getNoChars() <<endl;
 cout <<wr->getNoSentences() <<endl;
```

Listing 8: Inheritance based solution – Usage example.

a class of `IReader` type (e.g. `StringReader`) will be the last in the chain, since it is not generic.

The associated diagram, shown in the Figure 3, does not emphasizes any relation between the classes. They will be connected only when concrete examples are created.

Listing 8 shows a usage example. In the usage example, all three decorators are added (the order has no importance), but depending on the needs only one, or a combination of two could be used.

This solution based only on inheritance (without composition) has advantages, but also some disadvantages:

Advantages:

- only classes for the needed combinations are created;
- all new defined methods in different decorators are accessible;
- the solution is very simple,
- static class creation assures efficiency.

Disadvantages:

- for each specialization of `IReader`, new classes should be defined for each particular combination of decorators. If `FileReader` is a basic reader with a text file as a source, in order to use an instance similar to that defined in the previous usage example then another distinct class should be defined as:

```
    CharReader <
        SentenceReader <
            WordReader <FileReader >>>
```

- the combination of functionalities could not be changed during the source traversal;
- the decorations could not be dynamically add or retrived.

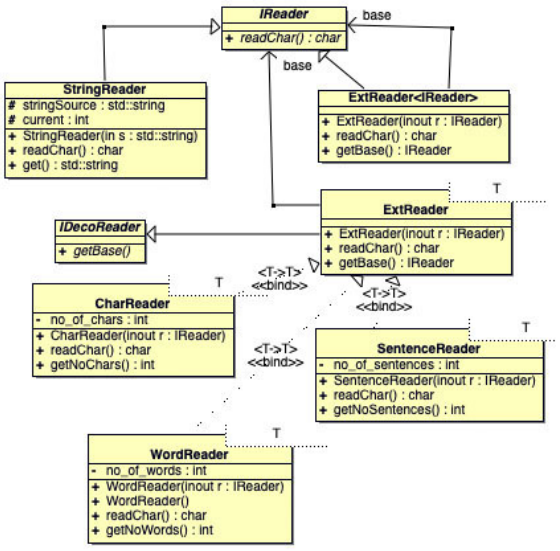

Figure 4: The Class Diagram of the HybridDecorator Pattern Applied to the *Reader* Problem.

4.2 Hybrid Solution – HybridDecorator

A hybrid solution that uses both inheritance and composition could be defined. This preserves the basic object wrapping, but the decorations will be defined as a single class obtained through a chain of inheritance derivation. In order to assure this, we need to define a class `ExtReader` that provides the support for composition, but in the same time, intermediates the decorations inheritance.

```
template <typename T=IReader>
class ExtReader: public T,
public IDecoReader, public IReader
{
protected:
IReader * base;
public:
 ExtReader(IReader* r): T(r) {
  this->base = r;
 }
 char readChar() {
  return T::readChar();
 }
 IReader * getBase() {
  return T::getBase();
 }
};
```

Listing 9: The definition of the class `ExtReader` for the general parameter value.

The class is defined for the following two cases:

- the basic case with the template parameter equal to `IReader`,
- the general case with a general template parameter(`T`)

The definitions for these two cases are different, and the difference is related to the call of the overridden method `readChar()`:

- for the basic case the call is sent to the wrapped object (`base`);
- for the general case the call is sent up to the superclass – `T`.

Listing 9 presents the implementation the class `ExtReader` for the general template parameter case, and Listing 10 presents the implementation of the `ExtReader` for the implicit case when the template parameter is `IReader`.

```
template <>
class ExtReader<IReader>:
    public IReader, public IDecoReader
 protected:
  IReader* base;
 public:
  ExtReader(IReader * r) {
    this->base = r;
  }
  char readChar() {
   return base->readChar();
  }
  IReader* getBase() {
   return base;
  }
};
```

Listing 10: The definition of the class `ExtReader` for the implicit parameter value (`IReader`).

```
  template <typename T=IReader>
class CharReader: public ExtReader<T>
{
 private:
  int no_of_chars;
 public:
  CharReader(IReader* r):
                ExtReader<T>(r){
   ExtReader<T>::base = r;
   no_of_chars=0;
  }
  char readChar() {
   char c = ExtReader<T>::readChar();
   if (c!=0) no_of_chars++;
   return c;
  }
  int getNoChars() {
   return no_of_chars;
  }
};
```

Listing 11: `CharReader` class definition.

For the concrete decorators, which are extended from `ExtReader`, there is no need to explicitly define the class for the implicit value of the parameter; this is solved through the `ExtReader` definition. Listing 11 shows the definition of the `CharReader` class. `WordReader` and `SentenceReader` have similar definitions.

The associated UML diagram for this hybrid implementation is shown in Figure 4. The interface `IDecoReader` is similar to the interface `IDecorator`, and it just defines the method `getBase()`.

A usage example based on this solution is given in the Listing 12: a reader associated to a string source is decorated with `CharReader`, `WordReader`, and `SentenceReader` in order to allow the counting of the read characters, words and sentences. It can be noticed that the usage is very similar to the classical Decorator implementation, but composition is replaced with the template parameter specification.

```
StringReader *r =
 new StringReader("Happy Birthday!");

SentenceReader<
 CharReader<
  WordReader<>>> *wr =
       new SentenceReader<
            CharReader<
              WordReader<>>>(r);

 while ((c= wr->readChar()) != 0)
   cout << c;

 cout<<wr->getNoWords()<<endl;
 cout<<wr->getNoChars()<<endl;
 cout<<wr->getNoSentences()<<endl;
```

Listing 12: Hybrid solution – Usage example.

Advantages of the hybrid solution:

- only the classes for the needed combinations are created;
- all new defined methods in different decorators are accessible;
- the combination of functionalities could be changed during the source traversal – the basic object is retrieved (through the method `getBase()`), and then it could be passed to another combination of functionalities.
- for all the specializations of `IReader` only one particular class that corresponds to a particular functionality combination could be used;
- in order to add a new decoration a new object wrapping (similar to the classical Decorator) could be done – the wrapping will specify a new class with the desired decoration.
- static class creation assures efficiency.

Disadvantages:

- The decorations could not be dynamically retrieved;

Based on the advantages that this solution offers, we may consider that it represents a valid and efficient

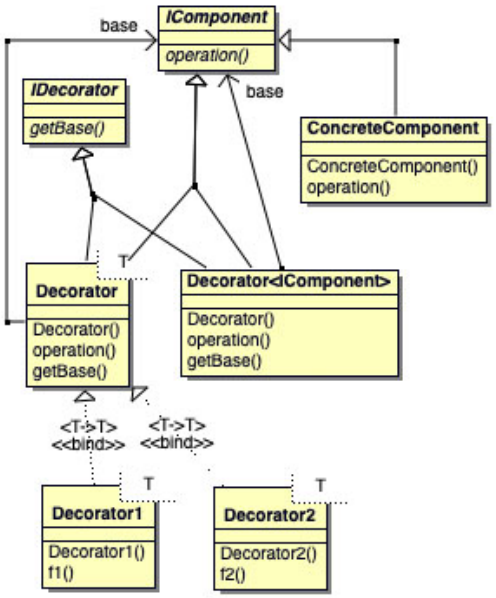

Figure 5: The Class Diagram of the HybridDecorator Pattern - The General Solution.

Decorator variant – HybridDecorator. The general structure of this is presented in the Figure 5. If we compare the structure diagram of the MixDecorator (Figure 2) and the HybridDecorator structure diagram, we may notice that even HybridDecorator assures full accesibility of all new interface-responsibilities, it is not necessary to define another special class `Operations` to assure this.

5 CONCLUSIONS

The paper analyzes policy based C++ implementations of the Decorator pattern. Since the classical Decorator has an accessibility problem if new responsibilities are added to the interface, MixDecorator is an alternative that solves this problem. But its implementation (in the extensible mode) requires some language mechanisms that allow interface extension (or extension methods). For MixDecorator C++ implementation, we have shown how the template policies could be used as a mean for defining new interfaces, and in fact to cover the need for extension methods.

Template policies are proposed to be also used for defining other alternative solutions of the Decorator pattern. Both of the two proposed variants assure complete accessibility over new defined methods (since inheritance is used for decorations). The HybridDecorator has a higher degree of reusability, and it also offers the advantages of the MixDecorator related to accessibility (so more than the classical Decorator) but with a simpler structure.

The implementation of the MixDecorator pattern preserves the characteristics of the classical Decorator pattern regarding the possibility to dynamically remove or add decorations. Also, for the HybridDecorator we have the possibility to add decoration, since the initial decorated object could be wrapped into a new decoration, similarly defined.

In the Decorator pattern definition is it stated that the composition is a good alternative to inheritance. The presented solutions that are based on policies are implicitly based on inheritance, but the idea that inheritance should be avoided was in the context of using inheritance for creating all possible combinations - and this is not the case for the policy solution. The recursion implicitly involved by the templates policies fits very well to the recursion implied by the Decorator definition.

REFERENCES

Abrahams, D. and Gurtovoy, A. (2003). *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. Addison-Wesley.

Alexandrescu, A. (2001). *Modern C++ design: generic programming and design patterns applied*.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object Oriented Software*. Addison Wesley.

Microsoft Contributors (2015). Extension methods (C# programming guide). [online: https://msdn.microsoft.com/en-us/library/bb383977.aspx]. [Accessed: 07.20.19].

Niculescu, V. (2015). Mixdecorator: An enhanced version of decorator pattern. In *Proceedings of the 20th European Conference on Languages of Programs*, EuroPLoP '15, pages 36:1–36:12, New York, USA. ACM.

Oracle (2018). JAVA SE 8: Default methods. [online: https://docs.oracle.com/javase/tutorial/java/IandI/]. [Accessed:07.20.19].

Pikus, F. G. (2019). *Hands-On Design Patterns with C++: Solve common C++ problems with modern design patterns and build robust applications*. Packt Publ.

Shalloway, A. and Trott, J. R. (2004). *Design Patterns Explained: A New Perspective on Object Oriented Design, 2nd Edition*. Addison Wesley.

Stroustrup, B. (2018). *A Tour of C++ (2nd Edition) (C++ In-Depth Series)*. Addison-Wesley Professional.

Zimmer, W. (1995). *Relationships between Design Patterns*, page 345–364. ACM Press/Addison-Wesley Publishing Co., USA.