Mixin based Adaptation of Design Patterns

Virginia Niculescu[a]
Faculty of Mathematics and Computer Science, Babeş-Bolyai University, 1 M. Kogălniceanu, Cluj-Napoca, Romania

Keywords: Patterns, Mixins, Composition, Inheritance, Static Binding, Performance, Templates, C++.

Abstract: Design patterns represent important mechanisms in software development, since a design pattern describes the core of the solution for a recurrent problem. This core solution emphasizes the participants with their responsibilities, and their possible relationships and collaborations. In the classical form these solutions are based on inheritance and composition relationships with a favor on composition. Mixins represents another important reuse mechanism that increases the level of flexibility of the resulted structure. In this paper, we investigate the possibility of using mixin for design pattern implementation, providing interesting adaptations. This approach represents a complementary realization of the design patterns, that brings the advantages of increased flexibility and statically defined relations between pattern participants.

1 INTRODUCTION

Design patterns are important mechanisms that capture design expertise and in the same time facilitate structuring of software systems. A design pattern defines a core solution for a problem which occurs over and over again in software development. Design patterns are valuable for emphasizing these recurrent problems, and of course, for providing the appropriate core solutions. In the classical form these core solutions are based on inheritance and composition relationships with a favor on composition.

In (Gamma et al., 1994) design pattern book, it is stated that the (object) composition is a good alternative to inheritance, and so there, an argumentation is given to favor composition when both relations are possible. But composition is not always the perfect solution from all points of view. Method calls are forwarded to some objects and the responsibility of managing these objects could become a problem (creation, deletion, updating, isolating). Also, if composition is used, isolated testing is difficult or even impossible in some cases.

Since design patterns could be considered building blocks in code development, it is of interest to analyze their possible alternative implementations while preserving their purposes.

Mixins represents another important reuse mechanism that increases the level of flexibility of the resulted structure. We intend to prove that mixins represent a good alternative to be used when defining design patterns. Our approach for realizing patterns promotes the benefits of static knowledge within patterns by moving such information to mixin units. This approach represents a complementary realization of the design patterns, that brings the advantage of statically defined relations between pattern participants.

Mixin implementation is different from one language to another. Most of the implementations are based on multiple inheritance, but there are also other specific implementations. We have chosen to use C++ for exemplification, but the design solutions remain the same for other languages.

The reason for this investigation is given also by the fact that static definitions are considered in general more efficient since the number of operations to be executed at the runtime is reduced.
If the component instantiation and interconnection is done statically, when the system is built, then we may expect a better performance from the time execution point of view.

The paper is structured as follows: Next section introduces mixins, and their C++ realization mechanisms, and the next presents a comparative analysis of static versus dynamic code composition. In Section 4 we describe possible mixin based adaptations for several design patterns, with the advantages and the implied challenges. Conclusions and planned future work are presented in Section 5.

[a] https://orcid.org/0000-0002-9981-0139

Niculescu, V.
Mixin based Adaptation of Design Patterns.
DOI: 10.5220/0010444702610268
In *Proceedings of the 16th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2021)*, pages 261-268
ISBN: 978-989-758-508-1

2 MIXINS

In object-oriented programming languages, a mixin is a class that contains methods to be used by other classes without having to be the parent class of those other classes. How those other classes gain access to the mixin's methods depends on the language (Bracha and Cook, 1990; Flatt et al., 1998).

Mixins can use the inheritance relation to extend the functionality of various base classes with the same set of features, but without inducing an "is-a" semantic relation. This is why, when mixins are used, the term is rather "included" rather than "inherited". Using a mixin is not a form of specialization but is rather a mean to collect functionality, even if the implementation mechanism is based on inheritance. A subclass could inherit most or all of its functionality by inheriting from one or more mixins through multiple inheritance.

A mixin can also be viewed as an interface with implemented methods. It is what has been introduced in Java and C# through default interface methods. There are also opinions that consider that the default interface methods represent traits implementation (Bono et al., 2014).

In C++ mixins could be defined using templates (Smaragdakis and Batory, 2000), by using so called policies.

2.1 C++ Policies

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). C++ template policies represent an interesting tool that could also be used to create classes based on ad-hoc hierarchies, even if the policies' initial intent was different. In the same time, they represent a solid mechanism to implement mixins.

A policy defines a class template interface; it is 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, the 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 of 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.

Policies have direct connections with C++ template metaprogramming and C++ traits which have been used in many language libraries; classical examples are: `string` class, I/O streams, the standard containers, and iterators. As emphasized in (Alexandrescu, 2001), in comparison with C++ traits they are more oriented on behavior instead of types. They are also considered a template implementation of the *Strategy* design pattern.

2.2 CRTP Pattern

The CRTP (Curiously Recurring Template Pattern) is in a way similar to policy mechanism being encountered when we derive a class from a template class given the class itself as a template parameter for the base class (Abrahams and Gurtovoy, 2003; Nesteruk, 2018). It is a parameterization of the base class with the derived class. In a concrete usage of these classes, the invocation of method `aMethod` through a `Base<Child>` object will lead to the execution of the variant defined in the `Child` class (Listing 1).

```
1   template <typename T>
2   class Base{
3   public:
4      void aMethod(){
5         T& child = static_cast<T&>(*this);
6         \\ use the 'child' object...
7      }
8   }
9   class Child : public Based<Child> {
10  public:
11     void aMethod(){
12     \\ specific definition of the method
13     \\  in the Child class
14      }
15  };
```

Listing 1: An example of using CRTP.

So, CRTP can be used in order to avoid the necessity for virtual methods, which come with a runtime overhead. In addition, CRTP can be used for adding functionality, as the mixins, too.

3 ARGUMENTATION: STATIC VERSUS DYNAMIC

The inheritance based development comes with several well-known problems, as the following two:

- *The Fragile Base Class Problem* – changes to a base class can potentially affect a large number of descendant classes and also the code that uses them.
- *The Inflexible Hierarchy Problem* – a class taxonomy created based on inheritance is not easy to

maintain, the new use-cases could impose adaptation and transformation that are not easy to obtain. A hybrid taxonomy that is based also on mixins is much more flexible and adaptable; changes could be better isolated.

Polymorphism is one of the most important object-oriented programming advantage. Virtual methods represent a mechanism through which the dynamic subtype polymorphism can be achieved. Still, they come with a performance cost since for their implementation a virtual methods table should be used for each class that defines virtual methods. Beside the memory overhead there is also an important execution time overhead, due to the indirection needed in calling virtual methods.

In their classical form, almost all of the fundamental patterns are based on subtype polymorphism. This is a good feature that assures flexibility, but it is not so efficient from the running time point of view. If we can provide more information that could be evaluated at the compile time we may increase the performance.

The proposed approach for patterns' definition promotes the benefits of static knowledge by moving information to components that could be evaluated at the compile time.

It can be argued that, in this way, we may lose the advantage of the possible dynamic reconfiguration; this is not exactly true since we may still dynamically choose between several configurations that are statically defined.

Separation between the static parts of the software design from the dynamic parts of the system behavior were proposed before, also specifically to patterns (Eide et al., 2002; Alexandrescu, 2001; Nesteruk, 2018). This separation makes the software design more amenable to analysis and effective optimization. In (Eide et al., 2002) the adaptation is done by identifying the parts of the pattern that correspond to static (compile-time or link-time) knowledge about the pattern and its participants, but this process was specific to individual uses of a pattern: each time a pattern is applied, the situation dictates whether certain parts of the pattern correspond to static or dynamic knowledge.

Typically, mixins are defined based on classes and they represent static mechanisms of functionality composition. Still, there are also some variations called dynamic mixins which are applied to objects at run-time, extending the object with new fields and methods. Dynamic mixins represent modular means of developing features or roles that can be composed with objects at run-time. Usually, dynamically typed languages are well suited to support dynamic mixins. In (Burton and Sekerinski, 2014) patterns adaptation using dynamic mixins is discussed, but just in the context of a teoretically defined language.

Instead of trying to extend the objects, we intend to provide general patterns realizations that increase the static part of them by using clasical, static mixins.

In order to demonstrate the proposed adaption, we use C++ language with the corresponding mixin mechanism based on templates. The pattern adaptation is not necessarily connected to C++ language, but it could be applied for any other language that accept mixins.

4 PATTERNS' ADAPTATION

As a general rule, the adaptation tries to transform an object composition relation that exists in the classical pattern definition into a mixin static composition. For an increased flexibility, a specialization relation could be also replaced to a mixin composition – i.e. when an abstraction could be specialized through a mixin inclusion. This is going to be seen in the adaptation of the *Template Method* pattern in Section 4.1. The replacement involves a transformation of at least one class from the pattern structure into a mixin class. Usually, only one relation can be replaced by a mixin composition.

In order to identify the possible candidates, we may follow a general strategy that contains the following steps:

- identify all the composition relations;
  - exclude "one-to-many" compositions;
- identify specialization relations for which semantically there is not (or can be excluded) an "is-a" relation.

We will present in what is follows adaptations for several patterns from all the three categories: structural, behavioral, and creational. For each pattern we will present the transformation that includes mixins, based on the structural diagram taken from Design Pattern book (Gamma et al., 1994): we emphasize what is excluded (using red color) and what is added (using blue color). In addition, for clarifying the solutions, we will present code snippets of the corresponding concrete C++ implementation.

4.1 Template Method

For the *Template Method* pattern, we use an inversion of specialization direction: instead of specialize an abstraction by deriving a new class from it, we allow it to be specialized through a mixin (Figure 1). The specialization inheritance relation is replaced with a mixin inheritance.

```
template <typename T>
class TMClass: public T{
 public:
   void template_method(){
      T::primitiveOperation1();
      T::primitiveOperation2();
   }
};
class ConcreteClass{
 public:
   void primitiveOperation1(){    ... }
   void primitiveOperation2(){    ... }
};
// /////// usage ////////
TMClass<ConcreteClass> a;
a.template_method();
```

Listing 2: Adaptation of Template Method pattern.

More concretely, the class `TMClass` that defines the `template_method` is not longer abstract and it is allowed to receive a mixin for specialization. The code in Listing 2 emphasizes this.

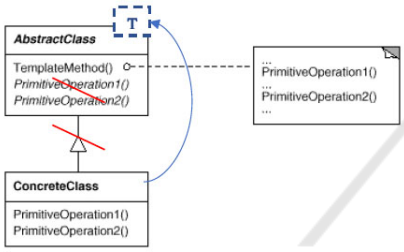

Figure 1: The adaptation of Template Method pattern.

Compared to the classical *Template Method* pattern, the specialization is obtained from a mixin that implements the primitive operations, instead of defining subclassing for specialization; this preserves the focus on the class that defines the algorithm, and lets the classes that implement primitive operation to be independent. This independence could come with a cost related to the state sharing – they will not inherit the state of the `TMClass` if this exists. But, this could be solved by giving this state as a parameter for the primitive methods. In addition, the type of this state could be parameterized and make the pattern more generic.

The independence of the concrete classes from the class that defines the template method, allows a mixin class to be used by more other classes (that defines template methods), the constraint being defined only by the condition of defining all primitive operations.

Concluding, we may say that instead of having many specializations of a class, we have a parameterized class, that it is specialized through the concrete(mixin) class parameters. The coupling between classes is much lower, leading to only one relation for a concrete execution.

Remark: For *Factory Method* pattern a similar approach could be followed – instead of letting the subclasses to specialize the factory methods, mixins could be given.

4.2 Memento

Memento is a very simple pattern that intend to provide a way to capture the internal state of an object at some moment of time and offer the possibility to restore that state in the future.

The classical implementation is based again on composition – the state is store into an object which is an instance of a class created only for this purpose. This composition is replaced with a mixin composition.

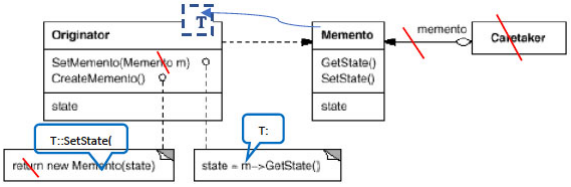

Figure 2: The adaptation of Memento pattern.

The class `Memento` could be defined as a mixin, and the `Originator` could use it (include it) in order to store and restore a specific state. The `Caretaker` is not longer necessary since the memento object is not stored outside the originator object. The client just has to invoke the originator object to preserve or restore its state.

In addition, the classes could be parameterized with the type of the state for a more general definition of `Memento` mixin. In this way the mixin could be used for differnt originator classes.

The encapsulation is still preserved since inside `Memento` the state is stored and managed in the `protected` section.

```
template < class S>
class Memento  {
 protected:
   S state;
   void setState(S s) { state = s; }
   S getState()  { return state; }
};
template < class S, template <class> class T>
class Originator : public T<S>{
   S state;
 public:
   Originator(S state){ this->state=state; }
   void createMemento() {
       T<S>::setState(this->state); }
   void setMemento(){ state = T<S>::getState();}
   void update(){
   // the state is updated
   }
};
// /////// usage ////////
Originator<State, Memento> object();
object.createMemento();
object.update();
object.setMemento();
```

Listing 3: Adaptation of Memento pattern.

Figure 2 emphasizes the transformation of the pattern structure, and the Listing 3 gives details about the implementation. The `Originator` class is parameterized with the type of the state (`S`) and with a `Memento` that depends on that state type. The `Memento` is included as a mixin.

4.3 Bridge

Bridge pattern connects an abstraction with possible concrete implementations of its parts. The adaptation changes the composition relation between the abstraction object and the corresponding implementation object into a class relation – the abstraction "is made concrete" by infusion into it a concrete implementation through a mixin.

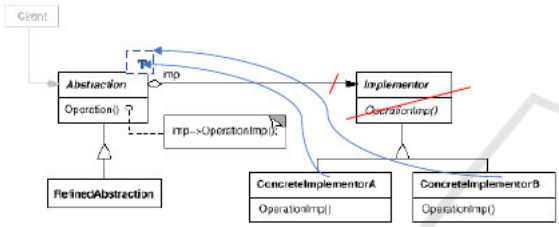

Figure 3: The adaptation of Bridge pattern.

Figure 3 emphasizes the transformation: the abstract superclass `Implementator` is no longer necessary since a concrete implementor class could be directly given to the `Abstraction` class, and there is no need for virtual methods.

As was specified, mixin inheritance doesn't have to be seen as implementing an "is a" relationship. Otherwise the proposed adaptation would lead to the conclusion that 'an abstraction is a kind of implementation' which of course would be wrong. But as the mixin definition says - the relation between the class and a mixin means that a functionality is included. In *Bridge* pattern, the `Implementor` provides a concrete implementation for the abstract part of the class.

```
template <typename T>
class Abstraction: public T{
 public:
   void operation(int params){
      T::operation_imp(params);}
};
class ConcreteImplementorA{
 public:
   void operation_imp(int params) { params++; }
};
// /////// usage ////////
Abstraction<ConcreteImplementorA> entity;
entity.operation(0);
```

Listing 4: Adaptation of the Bridge pattern.

For *Bridge* pattern we may also provide a realization based on CRTP C++ pattern that changes the relation between "abstraction" and "implementor" – this alternative variant is presented in Listing 5. This variant has similar characteristics.

```
template <typename T>
class CRTP_Abstraction{
 public:
   void operation(int params){
      static_cast<T*>(this)->operation_imp(params);
   }
};
class CRTP_ImplementorA:
   public CRTP_Abstraction<CRTP_ImplementorA>{
 public:
   void operation_imp(int params) { params++;}
};
// /////// usage ////////
CRTP_Abstraction<CRTP_ImplementorA> entity;
entity.operation(0)
```

Listing 5: Adaptation of the Bridge pattern – CRTP variant.

It can be argued that using this solution the client doesn't have the possibility to move dynamically from one implementor to another. This is not true, because if the all possible variants are known, the change means only to choose between several predefined variants – factory methods could be used. The drawback comes more from the extensibility point of view, when other new concrete implementors are intended to be added to the system – but since the new implementors should be statically defined, this leads to a problem with classical implementation, too.

4.4 Mediator

The *Mediator* pattern is used when we need to have an object that encapsulates the interaction between a set of other objects. The pattern promotes low coupling by avoiding objects to refer each other explicitly.

For the *Mediator* pattern the adaptation is a little bit more complex as it can be seen in the Figure 4. The adaptation imposes also a class factorization: `Mediator` class should be split and so `StateMediator` class is created in order to store the existing objects. The mediator classes, as ConcreteMediator, will define only the interaction between objects (colleagues) and not their creation and storage. A ConcreteMediator class will be used as a mixin to be included into `StateMediator` class.

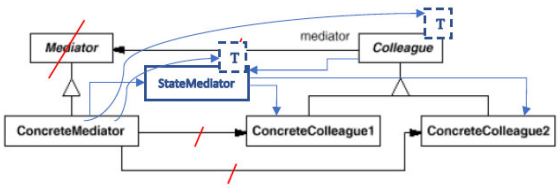

Figure 4: The adaptation of Mediator pattern.

```
template <template <class S > class T>
class Colleague{
 protected:
   StateMediator<T> *state;
 public:
   Colleague(StateMediator<T>* state){
         this->state = state; }
   StateMediator<T> * getState(){ return state;}
   void changed(){
      state->reaction_to_change(this); }
};
template <template <class S > class T>
class ConcreteColleague1: public Colleague<T>{
 public:
   ConcreteColleague1(StateMediator<T>* state )
         : Colleague<T>(state){}
   void update(){
// update something for 'this' colleague
      Colleague<T>::changed(); }
};
template <template <class S > class T>
class StateMediator: public T<StateMediator<T>>
{ protected:
   ConcreteColleague1* c1;
   ConcreteColleague2* c2;
 public:
  void setConcreteColleague1(
      ConcreteColleague1<T>* c){ v1=c; }
  void setConcreteColleague2(
      ConcreteColleague2<T>* c){ v2=c; }
  ConcreteColleague1<T>* getConcreteColleague1(){
        return c1;}
  ConcreteColleague2<T>* getConcreteColleague2(){
        return c2;}
};
```

Listing 6: Adaptation of the Mediator pattern – Colleague, ConcreteColleague1, and StateMediator classes.

An object composition is preserved for storing the set of objects that need to interact; so, the object composition is used for an object of type `StateMediator`, which is used in order to store the team of participants. The behavior is split out of this class and it is included through a mixin class – `ConcreteMediator`.

```
template  <typename T>
class ConcreteMediatorA{
 public:
   void reaction_to_change(Colleague<ConcreteMediatorA
        > * s){
      StateMediator<ConcreteMediatorA>* state =
         s->getState();
      ConcreteColleague1<ConcreteMediatorA>* c1 =
         state->getConcreteColleague1();
      ConcreteColleague2<ConcreteMediatorA>* c2 =
         state->getConcreteColleague2();
   //update based on specific mediator strategy
   }
};
//////// usage /////////
StateMediator<ConcreteMediatorA> ds ;
//create the team
ds.setConcreteColleague1(new ConcreteColleague1<
      ConcreteMediatorA>(&ds);
ds.setConcreteColleague2(new ConcreteColleague2<
      ConcreteMediatorA>(&ds);
//updating one determine a change for the others
ds.getConcreteColleague1()->update();
```

Listing 7: Adaptation of the Mediator pattern – ConcreteMediator class and usage.

If we consider the example given in (Gamma et al., 1994) with graphic widgets (fields, boxes, ...) this will translate into a state mediator that store the graphical objects and several behavior mediators that establish the particular reaction after a specific widget selection.

Mediator is another example (beside of *Decorator*) that combines mixin composition with object composition: for the state of the team, object composition is used, but the specific behavior mediator is specified/infused as a mixin.

4.5 Builder

Creational design patterns could be adapted, too. *Builder* has a nice solution, in which object composition between the director and the builder is changed into a mixin composition.

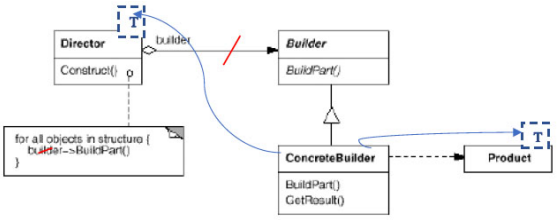

Figure 5: The adaptation of Builder pattern.

From the Figure 5 it can be seen that the class `Director` is infused with `Builder` specific behavior, being able to construct, but also to get the result that is specific to a particular builder – e.g. `ConcreteBuilder`. Implementation details are give in Listing 8.

```
template <typename T,
   typename S = typename T::part_type>
class Product{
   S part1, part2;
 public:
   Product(S p1, S p2){
      this->part1 = p1;
      this->part2 = p2; }
};
class ConcreteBuilder{
 public:
  typedef int part_type;
  void BuildPart1(){ part1 = 10; }
  void BuildPart2(){ part2 = 20; }
  Product<ConcreteBuilder> getResult(){
   return Product<ConcreteBuilder>(part1,part2);
  }
 private:
   part_type part1, part2;
};
template <typename T>
class Director: public T{
 public:
  void construct(){
      T::BuildPart1();
      T::BuildPart2(); }
  Product<T> getResult(){return T::getResult();}
};
//////// usage /////////
Director<ConcreteBuilder> a;
a.construct();
Product<ConcreteBuilder> pa = a.getResult();
```

Listing 8: Adaptation of Builder pattern.

The link between the builder and its specific product is done through the parameter of `Product` class; the result will have the type `Product<ConcreteBuilder>`. In this case, the builder specializes also the product. Though this transformation, the classical object com-

position between `aDirector` and `aBuilder` is transformed into a mixin composition.

4.6 Façade

Facade pattern provides a unified interface to a set of interfaces in a subsystem; this higher-level interface makes the subsystem easier to use.

Clients communicate with the subsystems through Façade that has the responsibility to forward the requests from the clients to the appropriate subsystem. If the number of interfaces in that set of interfaces (subsystems) remains fix, then `Facade` could be defined as a combination of mixins that correspond to the aggregated interfaces. Façade controller is one of the most encountered use-cases.

Here, the definition of mixin composition is directly applied: the Façade is created as a composition of several units (classes). Listing 9 presents a simple variant, with three subsystems and two possible requests from the client.

```
template <typename SUBSYSTEM_A,
typename SUBSYSTEM_B, typename SUBSYSTEM_C>
class Facade:
    public SUBSYSTEM_A,SUBSYSTEM_B,SUBSYSTEM_C{
  public:
    void request1(){
        SUBSYSTEM_A::functionA();
        SUBSYSTEM_B::functionB();
    }
    void request2() {
        SUBSYSTEM_C::functionC();
    }
};
class Sub_SystemA {
  public:
    void functionA() {
        cout << "subsystem A function A" << endl;
    }
};
//similar code for Sub_SystemB and Sub_SystemC
//////// usage /////////
Facade<Sub_SystemA, Sub_SystemB, Sub_SystemC> s;
s.request1();
s.request2();
```

Listing 9: Adaptation of Façade pattern.

4.7 Discussion and Counterexamples

Even the transformation could be applied for many patterns, still there are some patterns for which this kind of adaptation is not possible in good conditions. They are generally characterized by the following possible aspects:

- the life time of the associated entities/objects is different;
- the number of associated entities could be dynamically changed;
- a particular state directly determined the behavior of many objects.

If we consider an adaptation of the *Iterator* pattern that makes the associated collection to use an `Iterator` mixin, we restrict the behavior to only one iterator associated to a collection object that has the same life time as the collection object. This mixin implementation would correspond to an older style of implementing collections, that includes a cursor between collection's elements. The classical *Iterator* pattern increases the flexibility in iterating the elements, by separating iteration from collection.

State pattern deals with situations when the actual state controls a 'system' behavior. Including the state into the system by using mixins would make difficult to allow this state, and implicitely the behavior, to change dynamically.

The *Observer* pattern introduces a behavior dependency on the state of an external object. Theoretically, for *Observer* would be possible to go on using a similar approach with that used for *Mediator*, but in this case the benefits are less than the introduced disadvantages; the problem is given by the fact that the list of the observers is dynamically changed, and each observer could react differently and independently.

Considering only the *Composite* pattern structure, we may say that a similar approach to that used for *Decorator* could be taken. But, by semantic, *Composite* is intrinsically related to objects and their compositions. In addition, for *Composite* it is very common to have "one-to-many" composition: an aggregate node could contain one, two or more other nodes, and this number is not fixed for the entire structure.

This highly dynamic behavior makes these examples of patterns unappropriated to be adapted using static mixins.

Still, as emphasized before, there are many patterns for which the mixin based adaptation is both easy and advantageous. In addition we may mention: *Adapter* pattern adaptation is straightforward, the obtained advantage is that a general parameterized Adapter class could be defined for a particular Target interface and specific requests; *Strategy* is often presented as a use case of using C++ policies; variants of *Decorator* was extendly analyzed in (Niculescu, 2020; Niculescu et al., 2020); a similar approach to that used for *Decorator* can be used for *Chain of responsibility* pattern: the handlers could be recursively added as mixins.

5 CONCLUSIONS

Programming based on static instantiation and interconnection (performed when the program is built instead of when the program is executed) is an approach that offers a good potential for performance. We have presented an adaptation of the fundamental design

patterns based on using mixins. In general, the adaptation replaces object composition with mixin composition. The resulted solutions provide the advantages borrowed from mixins: flexibility, low coupling, and static defined compositions.

The adaptations were illustrated using C++ mixins, but this doesn't mean that other languages with their mixins kind of composition cannot be used. Default interfaces are mechanisms that are now present in Java and C# and they are considered to allow mixin composition.

A comprehensive analysis of the possible adaptation of all fundamental design patterns described in (Gamma et al., 1994) would be interesting, but maybe even more interesting is the analysis of the impact of using these adapted patterns in concrete systems development. We may expect improved execution time, and better opportunities for testing and optimization. These represent the subjects of the planned future work.

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*. Addison-Wesley.

Bono, V., Mensa, E., and Naddeo, M. (2014). Trait-oriented programming in java 8. In *Proceedings of the 3rd PPPJ*, pages 181–186. ACM.

Bracha, G. and Cook, W. (1990). Mixin-based inheritance. In *Proceedings of the OOPSLA/ECOOP 1990*, pages 303–311. ACM.

Burton, E. and Sekerinski, E. (2014). Using dynamic mixins to implement design patterns. In *Proceedings of the 19th EuroPLoP*, page 1–19. ACM.

Eide, E., Reid, A., Regehr, J., and Lepreau, J. (2002). Static and dynamic structure in design patterns. In *Proceedings of the 24th ICSE*, ICSE '02, page 208–218. ACM.

Flatt, M., Krishnamurthi, S., and Felleisen, M. (1998). Classes and mixins. Association for Computing Machinery.

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

Nesteruk, D. (2018). *Design Patterns in Modern C++. Reusable Approaches for Object-Oriented Software Design*. Apress.

Niculescu, V. (2020). Efficient decorator pattern variants through c++ policies. In *Proceedings of the 15th ENASE*, pages 281–288. SciTePress.

Niculescu, V., Sterca, A., and Bufnea, D. (2020). Should decorators preserve the component interface? *CoRR*, abs/2009.06414.

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

Smaragdakis, Y. and Batory, D. (2000). Mixin-based programming in c++. In *Proceedings of the 2nd GCSE*, Lecture Notes in Computer Science. Springer.