Teaching about Creational Design Patterns – General Implementations of an Algebraic Structure

Virginia Niculescu
Department of Computer Science
Babeş-Bolyai University, Cluj-Napoca
E-mail: vniculescu@cs.ubbcluj.ro

Abstract

Teaching about design patterns is not easy, especially for the students that don't have so much experience in OOP. Hence, finding example from well-known arias is very important. Usually, students in Computer Science also learn a lot of Mathematics, and Algebra is a field that is normally very well treated. This article presents an example that deals with algebraic structures and uses creational design patterns.

The possibility of constructing a general polynomial built over any kind of algebraic field $(K, +, *)$ is discussed. In order to work with polynomials that have different coefficient types: real, complex, etc., the type of a polynomial coefficient is consider to be an abstract type – `FieldElem`. Three creational design patterns are discussed and advantages of using them are also analyzed. Patterns are used for the creation of the special values of the coefficients: zero and one.

The example helps the students to learn about creational design patterns, without creating big applications. This leads to an easy and well understanding of the concepts.

1 Introduction

Teaching about design patterns is not easy, especially for the students that don't have so much experience in OOP. Hence, finding example from well-known arias is very important. Usually, students in Computer Science also learn a lot of Mathematics, and Algebra is a field that is normally very well treated. This article presents an example that deals with algebraic structures and uses creational design patterns.

A polynomial can be built over any kind of algebraic field $(K, +, *)$, and usually we have to define a different class that implements the common operations with polynomials, for every such field.

Our goal is to define a general implementation of a polynomial, which can be used for different applications with polynomials, with different coefficient types.

2 Creational Design Patterns

Creational design patterns abstract the instantiation process. They are based on composition and inheritance. They allow us to make the pass from the hardcoding of a fixed set of behaviors towards

defining a smaller set of fundamental behaviors that can be composed into any number of more complex ones. Thus creating objects with particular behaviors requires more than simply instantiating a class. Five creational design patterns are considered to be classic: Abstract Factory, Prototype, Factory Method, Builder and Singleton [1].

To implement a general algebraic structure over a field, we need to use some special values, such as null and unity elements. Because, we want that our structure to be independent on the specific chosen field, we have to create these special values by using special methods.

Building a general class for a polynomial has not to be dependent on the coefficient types. So, we will use a general abstract type for the coefficients – `FieldElem`. But for the implementation of the polynomial class, we need to work with the special values 0 and 1. For instance, one constructor of the class `Polynomial` has to create a null polynomial. To create a null polynomial, we have to create a null coefficient, and we can do this only if we have the possibility to create a null coefficient, even if we don't know its exact type.

Abstract Factory, Factory Method, and Prototype design pattern may be used for this purpose, and we analyze here all these three approaches.

3 The Three Design Approaches

We start from an interface `FieldElem` (Figure 1), which is the interface for a general field element. The methods correspond to the usual operations defined with field elements: addition, multiplication, computing the opposite and the inverse element, and verifying if an element is equal to 0 or 1.

```
public interface FieldElem {
 FieldElem add(Fieldelem e);
 //PostCond:  -> this = this + e
 FieldElem multiply(FieldElem e);
 //PostCond:  -> this = this * e
 FieldElem opposite();
 //PostCond:  -> this = - this
 FieldElem inverse();
 //PostCond:  -> this = 1 / this
 boolean isZero();
 //PostCond:  -> true, if this = 0; false, otherwise
 boolean isUnu();
 //PostCond:  -> true, if this = 1; false, otherwise
}
```

Figure 1: The interface of the field elements `FieldElem`.

Also, we define a class `Polynomial` corresponding to the type of all polynomials with the coefficients type `FieldElem`. The classic operations are considered: value in a point, addition, and multiplication. If the coefficients are field elements, we may also define the division operation for polynomials.

For exemplification, we discuss, in all the cases, the creation of two null polynomials: one over $\mathbb{R}$, and the other over $\mathbb{C}$.

Note: We have chosen Java for the implementations.

3.1 Factory Method

Pattern Intent: *Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defers instantiation to subclasses [1].*

In this case, the class `Polynomial` is an abstract class, which defines two abstract methods: `createZero():FieldElem` and `createOne():FieldElem`. In order to create a concrete polynomial, it is necessary to derive a new class from the abstract class `Polynomial` (Figure 2).

Figure 3 shows a part of the definition of the abstract class `Polynomial`, and Figure 4 shows the definition code for the class `RealPolynomial`. The definition of the class `ComplexPolynomial` is similar.

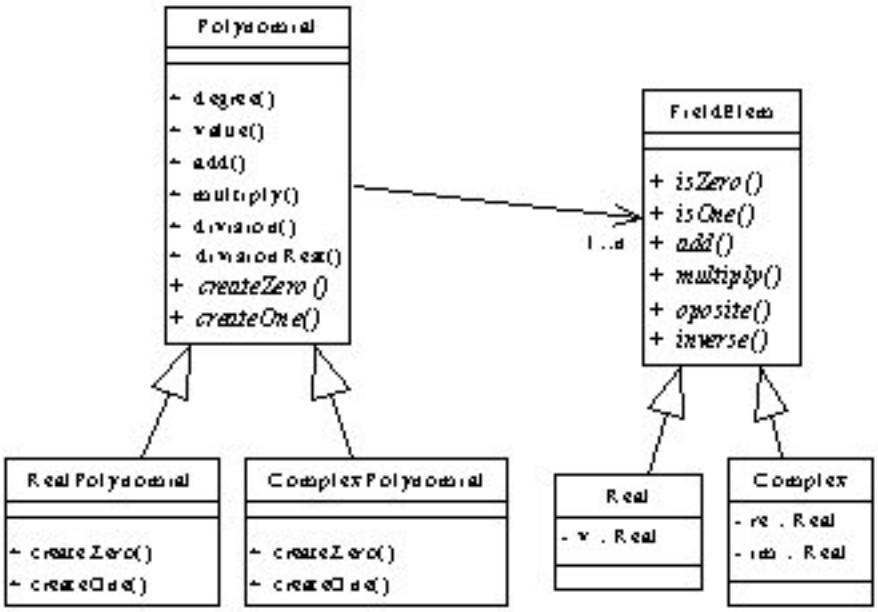

Figure 2: The class diagram for the Factory Method case.

```
public abstract class Polynomial {
 final static int MAX = 50;
 protected int gr;
 protected FieldElem []c;
//////////////////////////////////////
 public Polynomial() {
 //PostCond:  this = null polynomial
  c = new FieldElem[MAX];
  gr = 0;
  c[0] = createZero();
 }
 public FieldElem value(FieldElem x) {
  FieldElem e = createZero();
  for (int i = gr; i>=0; i--) {
   e.multiply(x);
   e.add(c[i]);
  }
  return e;
 }
 // ...
 public abstract FieldElem createZero();
 public abstract FieldElem createOne();
}
```

Figure 3: The definition of the abstract class `Polynomial`.

```
class RealPolynomial extends Polynomial{
 public FieldElem createZero(){
   return new Real(0);
 }
 public FieldElem createOne(){
   return new Real(1);
 }
}
```

Figure 4: The definition of the class `RealPolynomial`.

```
Polynomial p1 = new RealPolynomial();
Polynomial p2 = new ComplexPolynomial();
```

Figure 5: The creation of null polynomials over $\mathbb{R}$ and over $\mathbb{C}$ for the Factory Method case.

The creation of the two null polynomials of our exemplification is very simple and direct(Figure 5).

Because the class `Polynomial` is an abstract class we cannot define static methods as one that returns the sum of two polynomials: `public static Polynomial sum(Polynomial p1, Polynmial p1)` (*PostCond: -> p1 + p2*); and this would have been helpful for the users of this class.

We may conclude that Factory Method is not a very appropriate pattern, especially because it imposes the derivation of new polynomial classes.

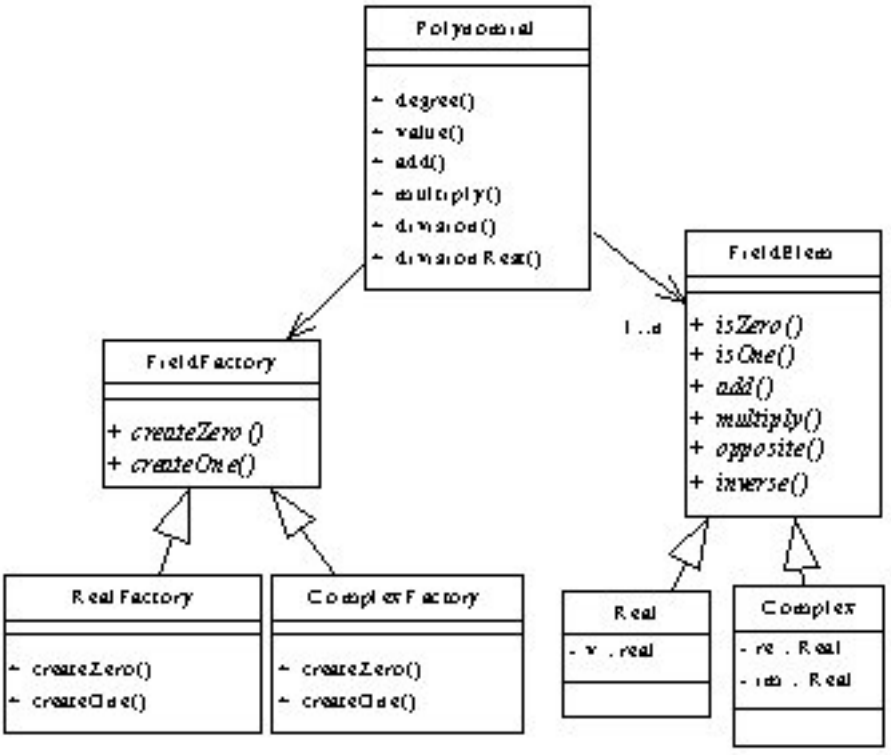

Figure 6: The class diagram for the Abstract Factory case.

3.2 Abstract Factory

Pattern Intent: *Provide an interface for creating families of related or dependent objects without specifying their concrete classes [1].*

```
public class Polynomial {
 final static int MAX = 50;
 protected int gr;
 protected FieldElem []c;
 protected FieldFactory f;
 ////////////////////////////////////////////////
 public Polynomial(FieldFactory f) {
 //PostCond:  this = null polynomial
  this.f = f;
  c = new FieldElem[MAX];
  gr = 0;
  c[0] = f.createZero();
 }
 ////////////////////////////////////////////////
 public FieldElem value(FieldElem x) {
  FieldElem e = f.createZero();
  for (int i = gr; i >= 0; i--) {
   e.multiply(x);
   e.add(c[i]);
  }
  return e;
 }
 ////////////////////////////////////////////////
 public static Polynomial sum(Polynomial p1, Polynomial p2) {
 //PostCond:  -> p1 + p2
  Polynomial a = new Polynomial(p1.f);
  a.add(p1);
  a.add(p2);
  return a;
 }
 // ...
}
```

Figure 7: The class Polynomial of the Abstract Factory case.

In this case, we consider a factory interface named `FieldFactory` that has two methods: `createZero():FieldElem` and `createOne():FieldElem` (Figure 6).

The implementation of the class `Polynomial` has to use a `FieldFactory` instance, and so the constructors of the class will receive an argument of this type. Figure 7 shows a part of the implementation of the class `Polynomial` in this case.

The Abstract Factory design pattern brings some advantages over the Factory Method case. We receive a very good structuring, and the class `Polynomial` does not have to be derived. Also, static methods as `public static Polynomial sum(Polynomial p1, Polynomial p2)` can be defined.

The classes `RealFactory` and `ComplexFactory` may be used for other purposes, too. For example, we may define a class `Matrix`, built in a similar way with the class `Polynomial`. So, we can work with matrices over real and complex numbers.

We may also use Singleton creational design pattern for the creation of the factory objects.

Singleton Pattern Intent: *Ensure a class only has one instance, and provide a global point access to it. [1]*

Figure 8 shows the implementation of the Singleton patterns for the class `ComplexFactory`.

The exemplification is showed in the Figure 9.

```
class ComplexFactory implements FieldFactory {
 private ComplexFactory(){}
 private static ComplexFactory instance;
 //////////////////////////////////////////////////
 public static ComplexFactory getInstance(){
  if (instance == null) instance = new ComplexFactory();
  return instance;
 }
 public FieldElem createZero() {
  return new Complex(0.0,0.0);
 }
 public FieldElem createOne() {
  return new Complex(1.0,0.0);
 }
}
```

Figure 8: The definition of the class `ComplexFactory`.

```
Polynomial p1 = new Polynomial(RealFactory.getInstance());
Polynomial p2 = new Polynomial(ComplexFactory.getInstance());
```

Figure 9: The creation of null polynomials over $\mathbb{R}$ and over $\mathbb{C}$ for the Abstract Factory case.

3.3 Prototype

Pattern Intent: *Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype[1].*

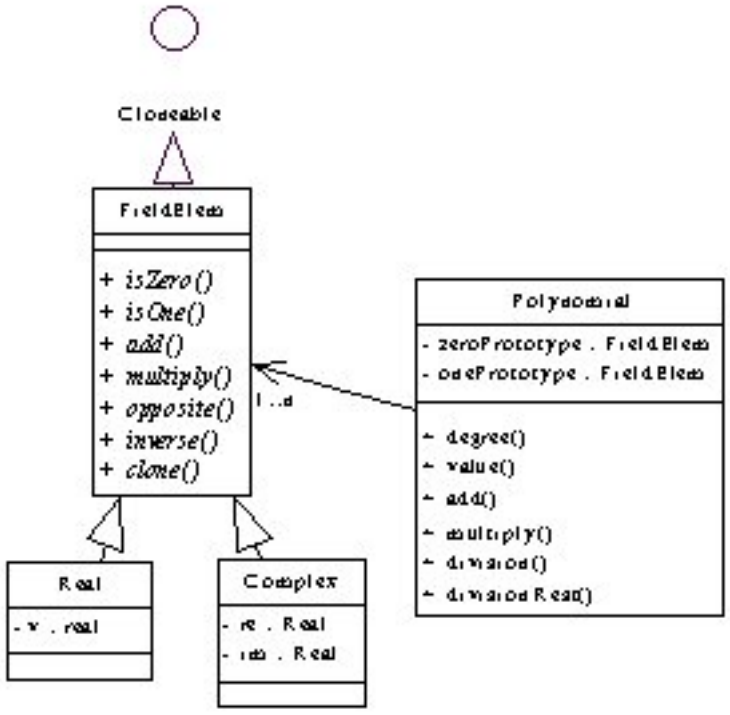

Figure 10: The class diagram for the Prototype case.

```
class Polynomial {
 final static int MAX = 50;
 private int gr;
 private FieldElem []c;
 private FieldElem zeroPrototype;
 private FieldElem onePrototype;
 ////////////////////////////////////////////////////
 public Polynomial(FieldElem z, FieldElem o) {
 //PostCond:  this = null polynomial
  c = new FieldElem[MAX];
  gr = 0;
  this.zeroPrototype = (FieldElem)z.clone();
  this.onePrototype = (FieldElem)o.clone();
  c[0] = (FieldElem)zeroPrototype.clone();
 }
 ////////////////////////////////////////////////////
 public FieldElem val(FieldElem d) {
  FieldElem e = (FieldElem)zeroPrototype.clone();
  for (int i = gr; i >= 0; i--) {
   e.multiply(d);
   e.add(c[i]);
  }
  return e;
 }
 ////////////////////////////////////////////////////
 public static Polynomial sum(Polynomial p1, Polynomial p2) {
  Polynomial a = new Polynomial(p1.zeroPrototype, p1.onePrototype);
  a.add(p1);
  a.add(p2);
  return a;
 }
 // ...
}
```

Figure 11: The definition of class `Polynomial` of the Prototype case.

The Prototype pattern may be used with some advantages. The Prototype pattern imposes only that every field type defines a method **clone** that allows an element to be copied. The advantage is that this leads to fewer classes than using Abstract Factory pattern, but the structuring would not be so good. We need only the class `Polynomial` and the field classes `FieldElem`, `Real`, and `Complex`.

The class `Polynomial` has two new attributes: `zeroPrototype:FieldElem` and `onePrototype:FieldElem`, which are initialized with the arguments received by the constructors (Figure 10). In order to force the field element classes to implement the method `clone()`, the interface `FieldElem` has to extends the interface `Cloneable`.

Figure 12 shows the creation of the two null polynomials for the Prototype case.

The disadvantage of using Prototype over Abstract Factory is evident only if we try to develop a general algebraic library for polynomials, matrices, etc. over any possible types.

```
Polynomial p1 = new Polynomial(new Real(0), new Real(1));
Polynomial p2 = new Polynomial(new Complex(0,0), new Complex(1,0));
```

Figure 12: The creation of null polynomials over $\mathbb{R}$ and over $\mathbb{C}$ for the Prototype case.

4 Conclusions

We have defined a design scheme for the implementation of a general class: `Polynomial`. The design scheme offers generality and flexibility.

The implementation of the class `Polynomial` is done in such a way to offer the user the possibility to work with different types of polynomials without concerning about the coefficient types dependencies. Also, this design offers a very good code reusing.

We have presented three approaches, based on three classic creational design patterns: Factory Method, Abstract Factory, and Prototype. The final analysis shows that the best advantages are given by Abstract Factory. The Singleton design pattern is also discussed. The example allows students to understand these design patterns, and also the differences between them are emphasized.

Students are usually very familiar with the algebraic structures, and on the other hand they usually find design patterns difficult to understand. Applying patterns on something they know very well can help them to understand the concepts. These were confirmed by the concrete experiences.

We also extended this designing experience for other algebraic structures, such as matrices. Now, our purpose is to build a general algebraic library.

References

[1] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns. Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[2] J. Gilbert, *Elements of Modern Algebra*, PWS-Kent, Boston, 1992.

[3] B. Eckel, *Thinking in Patterns*, 2000.

[4] H.E. Eriksson, M. Penker, *UML Toolkit* Wiley Computer Publishing, 1997.