STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **XLVIII**, Number 1, 2003

A DESIGN PROPOSAL FOR AN OBJECT ORIENTED ALGEBRAIC LIBRARY

VIRGINIA NICULESCU

Abstract. Object oriented programming and design patterns introduce a high level of abstraction that allows us to implement and work with mathematical abstractions. Classic algebraic libraries, based on imperative programming, contain subalgorithms for working with polynomials, matrices, vectors, etc. Their big inconvenience is the dependency on types. For example, a polynomial can be built over any kind of algebraic unitary commutative ring $(R, +, *)$, and we have to define a different set of procedures that implement the common operations with polynomials, for every such ring.

We propose here an object oriented approach for designing an algebraic library, based on design patterns, which remove this inconvenient. The big advantage of this approach is given by the creational design patterns, specifically Abstract Factory and Singleton. They introduce significant flexibility and abstractness. Thus, we may work with abstract algebraic structures, such as: groups, rings, fields, etc., like mathematicians do.

Keywords: OOP, design, patterns, algebraic structures, abstracness

1. Introduction

During the time, many algebraic libraries, which contain subalgorithms for working with polynomials, matrices, vectors, etc., have been built [7, 8, 2].

The big inconvenience of classic imperative algebraic libraries is their dependency on the types. For example, an polynomial can be built over any kind of algebraic unitary commutative ring $(R, +, *)$, and we have to define a different set of procedures that implement the common operations with polynomials, for every such ring.

Some other approaches are based on generic programming[8]. This may represent a solution but it has some inconveniences:

Received by the editors: May 20, 2003.

2000 *Mathematics Subject Classification.* 98A70,11C08,11C20.

1998 *CR Categories and Descriptors.* D.1.5 [**Software**]: Programming Techniques – *Object-oriented Programming*; D.3.3 [**Software**]: Programming Languages – *Language Constructs and Features*; E.2 [**Data**]: Data Storage Representations.

- Not all object oriented languages have mechanisms for genericity, and those having these mechanisms – like STL of C++ – don't offer the possibility to constrain the parameterized types to any explicit conditions. (Still, in the "Generic Java" proposals, the parametric types may be constrained to some conditions[10], but GJ is yet not used.)
- A parameterized matrix multiplication routine could be written and instantiated for matrices over integers, rationals, maybe real and complex numbers, numbers in Z/mZ and so on. But, this will produce the complete multiplication code for each type, in the executable.

Object oriented programming and design patterns form a very good framework for implementing a general algebraic library. We analyze here an object oriented approach for designing an algebraic library, based on design patterns, which remove the inconvenient of type dependency. We will create abstract classes that implement general abstract algebraic structures, and we will use Abstract Factory design pattern for building the special values.

Object oriented programming has been used before for designing some algebraic libraries [8], but the difference is given by the usage of creational design patterns. In this way, we can build not only a flexible numerical algebraic library, but a general abstract algebraic library.

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 [5].

We need, for our library, to use some special values, such as null and unity elements, in some classes where we don't know the concrete type of the special values. So, we have to create them by using special methods.

For example, building a general class for a polynomial does not have to be dependent on the coefficient types. So, we will use a general abstract type for the coefficients. But for the implementation of the class polynomial, we need to work with the special values 0 and 1.

Abstract Factory, Factory Method, and Prototype design pattern may be used for our purpose, and we analyse which one is more appropriate.

Factory Method is not appropriate for building the library, because it imposes the derivation of new classes for the main algebraic structures. For example, for a polynomial, the purpose is to define in our library a completely defined class, and let the user to use it for any appropriate coefficient type.

The Prototype pattern may be used with some advantages. The Prototype pattern imposes only that every type defines a method **clone**, that allow an element to be copied. The advantage is that using the Prototype pattern leads to fewer classes than using the Abstract Factory pattern, but the structuring of the library would not be so good.

So, we have chosen the Abstract Factory design pattern. Different factory classes that define creational methods for the special values are defined, such as: `GroupFactory, FieldFactory`, etc.

Factory classes may use Singleton pattern, in order to allow a single factory instance for each type.

3. The Design Scheme

We start from a general algebraic element: `AlgElem`, which is implemented as an interface with no methods. We define the interfaces corresponding to algebraic elements, starting from their definition.

3.1. **Basic structures.** A very well known and used basic algebraic structure is the group. We define consequently an interface `GroupElem` that extends `AlgElem` (Figure 1). This interface contains a method `isZero()` that allows us to verify if an element is the identity element or not, a method for the computation of the opposite for an element, and a method for the operation $+$. The method `isAddCommutative()` will be defined to return `true` or `false`, depending on the concrete case. If it is possible, this method should be defined static, and implicitly returning `false`.

Rings are other basic algebraic structures, and corresponding to them, we define three interfaces: `RingElem`, `UnitaryRingElem`, `DivisionRingElem`, and `FieldElem` (Figure 1). For fields, which are commutative division rings the corresponding interface is `FieldElem`.

The method `isMultiplyCommutative()` returns `true` or `false` in the ring classes, and always `true` for the field classes.

The method `isInvertible()` returns always `true` for `DivisonRingElem` and `FieldElem` objects. Similarly, the method `isMultiplyCommutative()` always return `true`, in the classes implementing the `FieldElem` interface.

3.2. **Polynomials.** Now we consider the set of all polynomials – $R[X]$, over a commutative ring with identity $(R, +, \cdot)$. $R[X]$ is a subring with identity of the ring $(R^{\mathbb{N}}, +, \cdot)$, where $R^{\mathbb{N}}$ is the set of all functions with the domain $\mathbb{N}$ and codomain R.

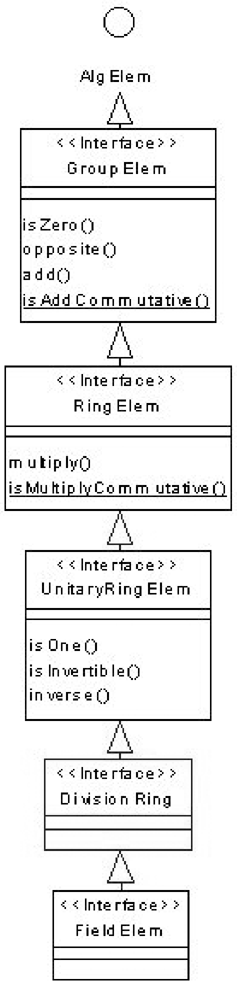

FIGURE 1. The class diagram for the basic structures

Corresponding to these, we consider two interfaces derived from the interface `UnitaryRingElem`: `Polynomial` and `DivisionPolynomial` (Figure 2).

The method `division` has some strong preconditions, assuring that the necessary conditions for polynomials division are satisfied.

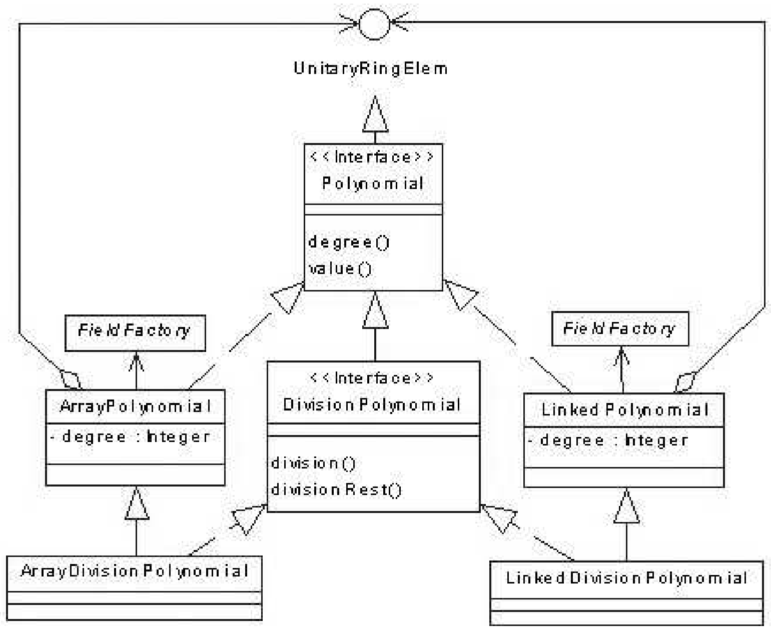

FIGURE 2. The class diagram for polynomials

To implement a concrete class for the ADT Polynomial, we have to choose the representation of the data. There are two classic ways for representation:

- using an array of coefficients;
- using a list of monoms.

We may also consider other storage formats, and, as well, we may introduce *storage format abstraction level* like in [8]. This would mean creating another abstract class `StorageFormat`, from which the specialized storage format classes are derived. Bridge and Iterator patterns have to be used in this case.

In order to better illustrate the associated factory classes, we will consider in this presentation only these two kinds of storage, and we will use simple inheritance.

Using the first representation, the operation of coefficient selection is very fast, because of the direct access. The second one is very useful for the implementation of sparse polynomials.

Corresponding to each of them we define two concrete implementation classes – `ArrayPolynomial`,
`ArrayDivisionPolynomial` and `LinkedPolynomial`, `LinkedDivisionPolynomial`.

The coefficients type is also `UnitaryRingElem`, and so the Composition pattern is used here.

3.3. **Matrices and Vectors.** We start from the definition of the vector space.

Definition 1 (Vector Space)**.** *A vector space over a field K is an abelian group $(V, +)$ together with a so-called external operation*

$$\cdot : K \times V, (k, v) \mapsto k \cdot v,$$

satisfing the following axioms:

(1) $k \cdot (v_1 + v_2) = k \cdot v_1 + k \cdot v_2$*;*
(2) $(k_1 + k_2) \cdot v = k_1 \cdot v + k_2 \cdot v$*;*
(3) $(k_1 \cdot k_2) \cdot v = k_1 \cdot (k_2 \cdot v)$*;*
(4) $1 \cdot v = v$

Theorem 1. *Let V be a vector space over K and $n \in N^*$. Then V^n has a structure of a vector space over K, where the operations are defined by*

$$(v_1, \ldots, v_n) + (v_1^{'}, \ldots, v_n^{'}) = (v_1 + v_1^{'}, \ldots, v_n + v_n^{'}),$$
$$k(v_1, \ldots, v_n) = (kv_1, \ldots, kv_n),$$

where $k \in K$ and $(v_1, \ldots, v_n), (v_1^{'}, \ldots, v_n^{'}) \in V^n$.

For vectors implementation, we will consider the vector space, for which $V = W^n, n \in \mathbb{N}^*$, where $(W, +)$ is an abelian group. Corresponding to it, we build a class `Vector` (Figure 3). The method
`scalarProd(FieldElem)` corresponds to the external operation.

We may consider the matrices to be elements of the vector space $W^{nm}(K) = M_{mn}(K)$. The corresponding class is `SimpleMatrix`.

If the elements of the matrices are unitary ring elements ($(W, +, \cdot)$ forms a unitary ring), we can define a product operation between two matrices A and B that respect the following property: $cols(A) = rows(B) = n$. The product matrix C is defined by:

$$c(i, j) = \sum_{k=0}^{n} a(i, k) \cdot b(k, j).$$

For this case, we defined a class `Matrix` derived from the class `SimpleMatrix` (Figure 3).

Some specific operations can be defined for matrices, such as computation of the rank.

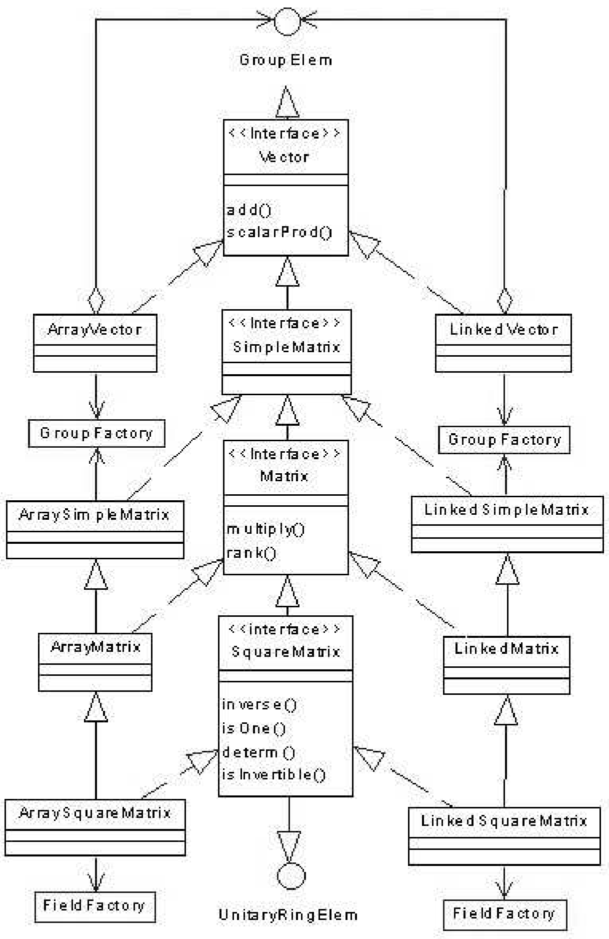

FIGURE 3. The class diagram for Vectors and Matrices

Square matrices with elements from a unitary ring $(R, +, \cdot)$ form also a unitary ring (Figure 3). On square matrices we may introduce many specific operations, such as computation of the determinant, and of the inverse matrix, etc. The class `SquareMatrix` is derived from the class `Matrix`, and it has specific methods.

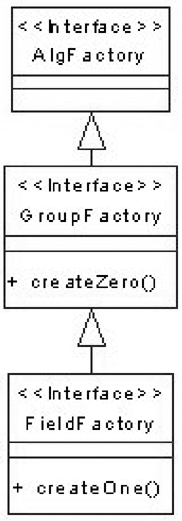

FIGURE 4. The class diagram for the basic factories

For matrices, we may also consider two classic ways for data representation, corresponding to dense and to sparse matrices:

- using a bi-dimensional array of elements;
- using a list of triples.

Corresponding to these representation, we have two kinds of concrete classes for matrices (Figure 3). Like for the polynomials case, Composition design pattern is also used for vectors and matrices.

3.4. **Other Structures and Classes.** The library may be extended with many other structures and classes.

For example, we can add a general abstract class named `AlgebraicSystem`, which allows the user to solve a $n \times n$ algebraic system. The concrete classes derived from it, will implement some concrete solving methods.

3.5. **Factories.** For simetry, we define an empty interface `AlgFactory`, which is the root of factories hierarchy.

The interface `GroupFactory` declare only one method: `createZero():AlgElem`. The interface `FieldFactory` extends the interface GroupFactory, and adds a new

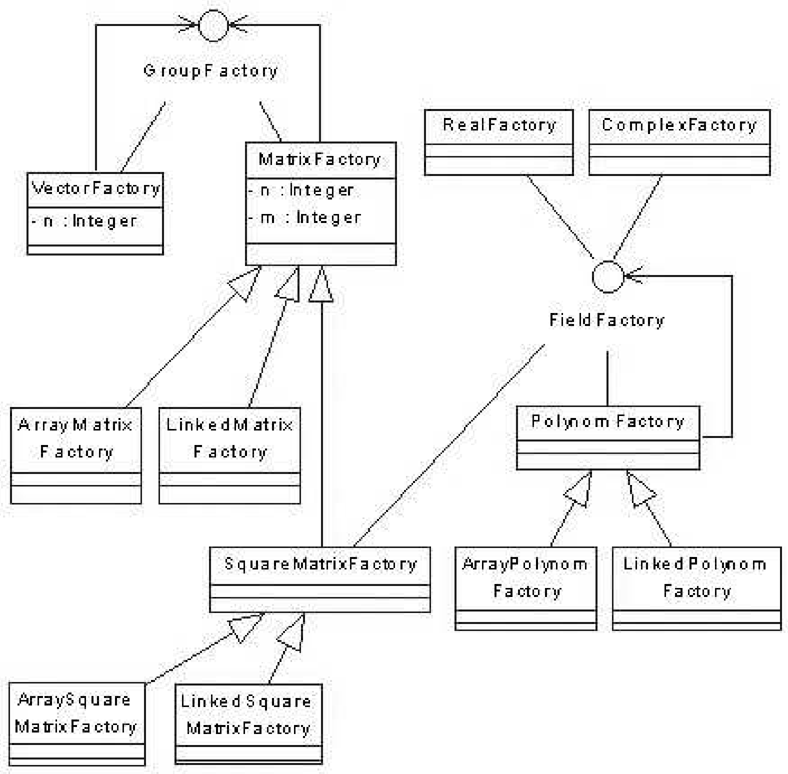

Figure 5. The class diagram for some concrete factories

method: `createOne():AlgElem`. We have chosen the name `FieldFactory`, because fields are much more used, but this interface will be also used for the unitary and division rings.

In order to solve the problem of creation for the special value 0 and 1, the implementations of the class `Polynomial` has to use a `FieldFactory` instance. For example, one constructor of the class Polynomial creates an null polynomial. To create a null polynomial, we have to create a null coefficient, and we can do this by using the method `createZero()` of a class that implements `FieldFactory`.

The concrete classes for matrices also have to use a `GroupFactory`. Square matrices classes need for the creation of identity matrices, both methods of a `FieldFactory`.

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

FIGURE 6. The creation of null polynomials over $\mathbb{R}$ and over $\mathbb{C}$

```
Matrix m1 = new LinkedMatrix(10,10, RealFactory.getInstance());
Matrix m2 = new ArrayMatrix(10,10, ComplexFactory.getInstance());
```

FIGURE 7. The creation of two 10×10 matrices over $\mathbb{R}$ and over $\mathbb{C}$

```
Matrix m =  new ArraySquareMatrix(10,
            new ArrayPolynomFactory(RealFactory.getInstance()))
```

FIGURE 8. The creation of a null 10×10 matrix over $\mathbb{R}[X]$

`GroupFactory` and `FieldFactory` are basic factories. We may define some other specialized factories, such as: `PolynomFactory, MatrixFactory` (Figure 5). These classes create the null and the unity values for polynomials, and matrices. These factories extend `GroupFactory` and `FieldFactory`, but in the same time they use these interfaces for creation of their basic elements. Their constructors receive a `GroupFactory` or a `FieldFactory` instance, which is used in the functions `createZero()`, or `createOne()`.

For concrete examples, concrete factories have to be built, and examples are given for the real and complex numbers: `RealFactory` and `ComplexFactory`. These concrete factories may implement Singleton[5] pattern.

4. IMPLEMENTATION

Any object oriented language can be chosen.

We have chosen Java because is an almost pure object oriented language, which offers many advantages. We have been very interested in having a good and simple mechanism for exception handling. We need it because there are many preconditions that impose different types of verification – for example, the necessity of checking if a parameter instance has a specific type. Our implementation in Java, define a main package named `Algebra`, which has some subpackages: `Factories, BasicStructures`, and `Structures`.

5. Concrete Examples

Let's consider that we need to work with polynomials with coefficients' types: real and complex. First, we define two classes `Real` and `Complex`, which implement `FieldElem` interface. Also, two factory classes has to be built: `RealFactory` and `ComplexFactory`, which implement the interface `FieldFactory`. The code showed in the Figure 6 illustrates the creation of two null polynomials: one over the $\mathbb{R}$ and another over the $\mathbb{C}$.

Similarly, we can work with matrices with elements of type real and complex, without creating any other class (Figure 7).

But, we can also work with matrices that have polynomial elements, like it is showed in the Figure 8.

6. Conclusions and Future Work

We have defined here a design scheme for a general algebraic library. The design scheme is based on object oriented programming, and it offers generality and flexibility. The Abstract Factory and Singleton creational design patterns, and some other design patterns, such as Composition, have been used.

The library is designed in a way that offers the user the possibility of working with general algebraic structures without concerning about the types dependencies.

Some general abstract algebraic structures and also some basic concrete algebraic structures have been designed. The users may define, without any problems, other algebraic structures based on the abstract ones. Also, existing algebraic structures may be combined and used in different ways. These are possible mainly because of using creational design patterns.

The fact that OOP allows us to describe problems in the terms of the problem space, rather than the terms of the solution space[3], is very well emphasized here. We can work and reasoning with mathematical abstractions as: groups, fields, rings, ...

This design scheme represents the first step for developing a general algebraic library; in the next step we intend to introduce the storage format abstraction level, which is based on Bridge and Iterator patterns. We intend to combine the storage format abstraction level with Abstract Factory pattern, in a way in which the factory will be also responsible for the creation of the specialized storage.

References

[1] Arnold, K., Holmes, D., Gosling, J., *The Java Programming Language*, Addison-Wesley, 2000.

[2] BLAS Technical Forum, Document for the BAsic Linear Algebra Subprograms, http://www.netlib.org/blas/blast-forum/

[3] Eckel, B., *Thinking in Java*, http://www.EckelObjects.com, 2000.

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

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

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

[7] LAPACK++: A Design Overview of Object Oriented Extensions for High Performance Linear Algebra, In Proceeedings of SuperComputing'93, pg. 162-171, IEEE Computer Society Press, 1993.

[8] Luján, M., Freeman, T. L., Gurd, J. R., *OoLaLa: an Object Oriented Analysis and Design of Numerical Linear Algebra*, In the Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications – OOPSLA 2000.

[9] Musser, D.R., Scine A., *STL Tutorial and Reference Guide: C++ Programming with Standard Template Library*, Addison-Wesley, 1995.

[10] Myers A.C., Bank J.A., Liskov Barbara, *Parameterized Types for Java*, Proceadings of the 24th ACM Symposium on the Principles of Programming Languages, Paris, 1997.

Department of Computer Science, Faculty of Mathematics and Computer Science, Babeş-Bolyai University, Cluj-Napoca, Romania

E-mail address: `vniculescu@cs.ubbcluj.ro`