The 7th International Scientific Conference
eLearning and Software for Education
Bucharest, April 28-29, 2011

IDENTIFICATION OF COMMON ELEMENTS AND PARAMETERS FOR CREATIONAL DESIGN PATTERNS IN ORDER TO CREATE A FRAMEWORK CORE

Valentin Corneliu PAU, Marius Iulian MIHAILESCU, Octavian STANESCU
Computer Science Department, ,,Titu Maiorescu'' University, Dambovnicului Street No. 22, sector 4, Bucharest, Romania
v_pau@utm.ro, mihmariusiulian@gmail.com, octavian_stan@yahoo.com

***Abstract**: This paper present our latest researches and opinions regarding the solution for creating a framework (as a library) which integrate all categories of design patterns into one single library, who can be invoked in development environments (e.g. Microsoft Visual Studio, NetBeans, Eclipse, Python, Ruby). To achieve this goal, it is necessary to study every design pattern and to extract the general and common elements and put them together and transform them into parameters.*

***Keywords:** design patterns, creational patterns, abstract factory, builder, singleton, prototype*.

I. INTRODUCTION

The concept of *patterns* comes from architecture, is introduced by Christopher Alexander [2] in 1977-1979. In 1987, Kent Beck [3] and Ward Cunningham [4] have start the idea of applying patterns in programming and they have present their researches at the Object-Oriented Programming, System Languages and Applications conference [5] in the same year.

In software engineering, a design pattern is a reusable solution to a common problem in software design process. A design pattern cannot be transformed immediately into code, because he is not a finished design. It presents a description, a template or a how to for resolving a problem that can be used in many different situations.

In the last twenty years an increase number of articles and books have been published in the architecture category, but few of them has tried to find general and common parameters and to put them together for creating a library especially for design patterns.

This work paper is an effort to provide accurate information for creating a framework core for creational design patterns, to give a pool for developers, focusing on Object Oriented Programming (OOP). *We will present a generalisation under many forms of all creational patterns, and finding for them general elements which can build a framework library to be used for different types of problems from real life.*

To get this goal done, we have to start to analyse all creational patterns, to find general aspects, common parameters, and one of the most important aspects - to understand how each of design pattern work, and bringing them together into a OOP framework, which can be used by IT engineers and software developers in developing their applications in different developing environments, such as Microsoft Visual Studio (with C#, C++, Visual Basic programming languages), NetBeans and Eclipse with Java, Python and Ruby. In this way, the life of programmers will be much easier.

Another important problem that we have to take into considerations is how we can adapt this design patterns framework for different kind of software and web applications from different fields. This is a difficult task, but not impossible, actually, here is the secret of this framework, how we can adapt each design pattern to be applied for different real life applications.

At the beginning it was struggle to understand a precise definition of a new concept, because it is always a new and hence unfamiliar idea. The few articles that exist put developers into difficult situations, becoming very awful to understand the basic concepts, and the most important thing, the way to apply them correctly. Employers want experienced employees, so you will get a feedback from them that you need to have experience to get a specific job. In this domain you will also have to read and think a lot, many of us are afraid but keep in mind the following phrase: *„The knowledge of the actions of great men, acquired by long experience in contemporary affairs, and a continual study of antiquity''* – The Art of War, seems applicable here, isn't it?

In the work of Erich Gramma et al [1], he present how some elements from design patterns can be used in Object Oriented Software. The process of designing object-oriented software is very tricky and hard.

II. THE MYTH CALLED DESIGN PATTERN

Christopher Alexander said one time, ''Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice". Also Alexander was talking about patterns in constructions such as buildings and towns, but what he said is true about object-oriented design patterns. The solutions expressed in this work paper are in terms of objects and interfaces instead of walls and doors, but if we look at the core of both types of patterns is a solution to situation (problem) in a context.

2.1 Essential Elements of Pattern

A pattern has four essential elements: *pattern name, problem, solution, consequences (see Figure 1).*

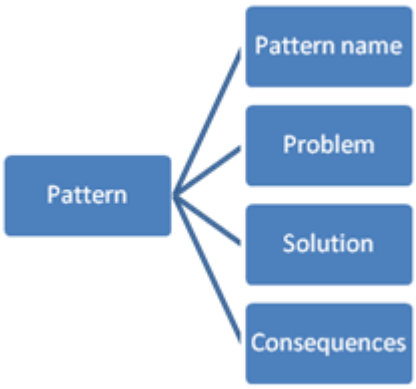

Figure 1. Elements of a pattern

1. The *pattern name* represents a way to use to describe a design problem, to present its solutions, and consequences in few words (maximum two). One important aspect that we need to have in mind is vocabulary adequate for patterns. Giving a name to the pattern immediately increase design vocabulary. It allows us to design at a high level of abstraction.
2. The *problem* describes when to apply the pattern. The problem explains with simple and concrete words the problem and its context. The accent will be on trying to describe specific design problems such as how to represent algorithms as objects. The *problem* element also describes class or object structures that are symptomatic of an inflexible design.
3. The *solution* illustrates the elements that build the design, their relationships, responsibilities, and collaborations. The *solution* doesn't describe a particular concrete design or implementation, because a pattern is like a template or layout that can be applied in many different situations.

4. The *consequence* represents trade-offs of applying the pattern. By consequences are unvoiced when we describe design decisions, they are critical for evaluating design alternatives and for understanding the costs and benefits of applying the pattern.

A simple UML diagram can show us how all this four elements can be generalized and used as a framework library.

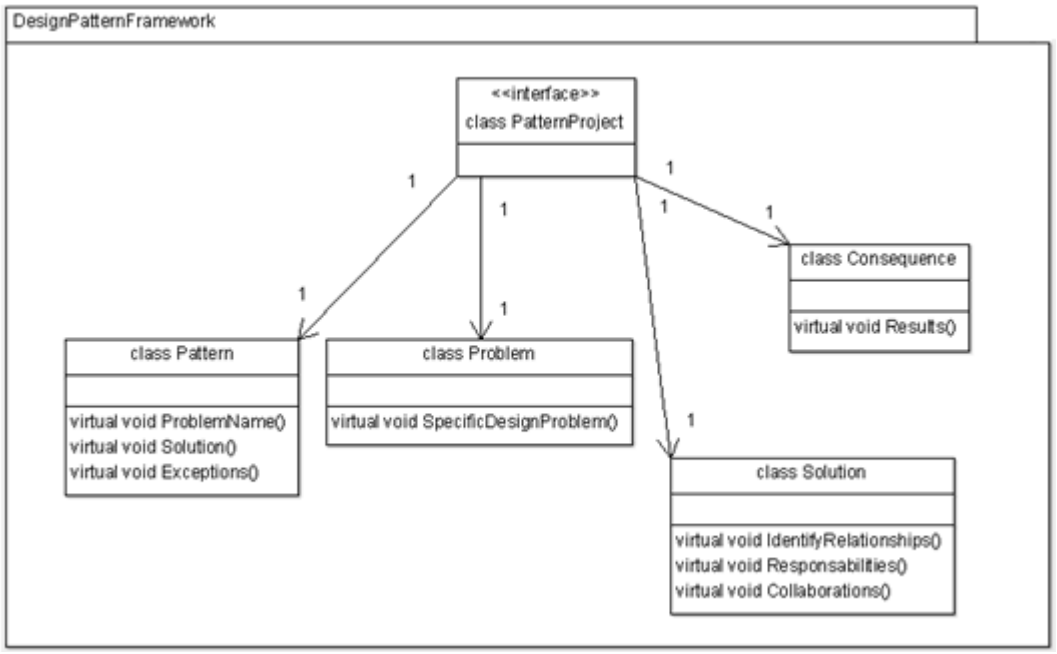

Figure 2. A view at a glance of the elements of an pattern

III. CREATIONAL DESIGN PATTERNS SPECIFICATIONS AND ORGANISING THEM UNDER AN CATALOG

Creational design patterns abstract the instantiation process. They help make a system independent of how its objects are created, composed, and represented. A class creational pattern uses inheritance to vary the class that's instantiated, whereas an object creational pattern will delegate instantiation to another object.

Creational design patterns have become very important as system evolve to depend more on object composition than class inheritance. As that happens, emphasis shifts away from hard-coding a fixed set of behaviors toward 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 [1].

Five creational patterns have been document by Gang of Four:

- **Abstract Factory** (Figure 3) which provides an interface for creating families of related objects, without specifying concrete classes;
- **Factory Method** which draw an interface for creating objects, but let's subclasses decide which classes to instantiate;
- **Builder** separates the construction of a complex object from its representation, so that the same construction process can create different representation;
- **Prototype** defines the kind of objects to create using a prototypical instance;
- **Singleton** ensures that a class has only one instance, and gives a global point of access to that instance.

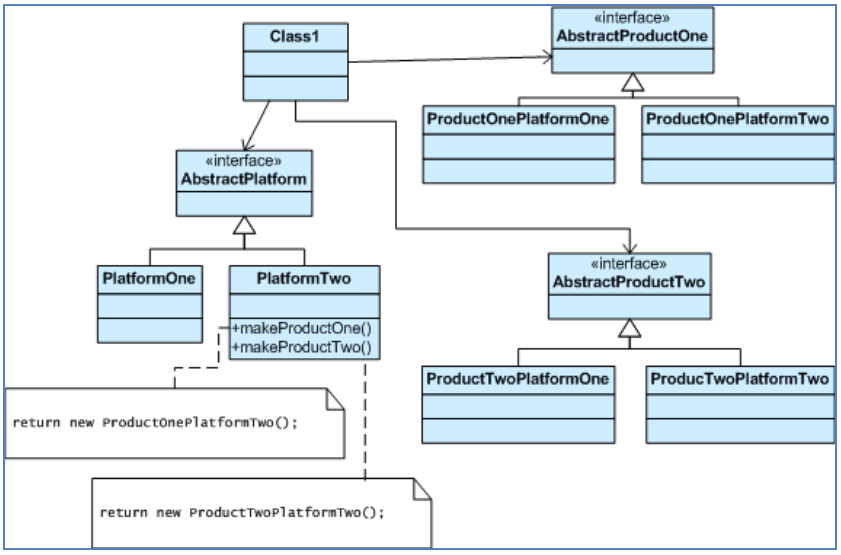

Figure 3. Abstract Factory Design Pattern

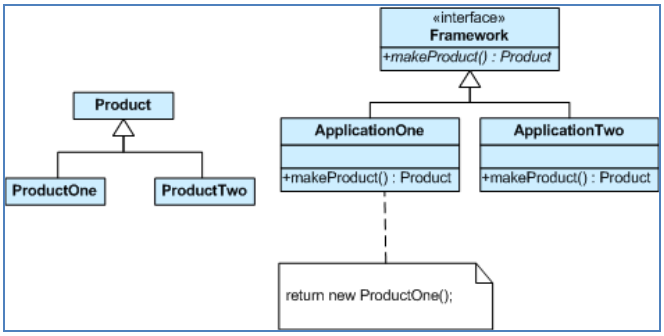

Figure 4. Factory Method Design Pattern

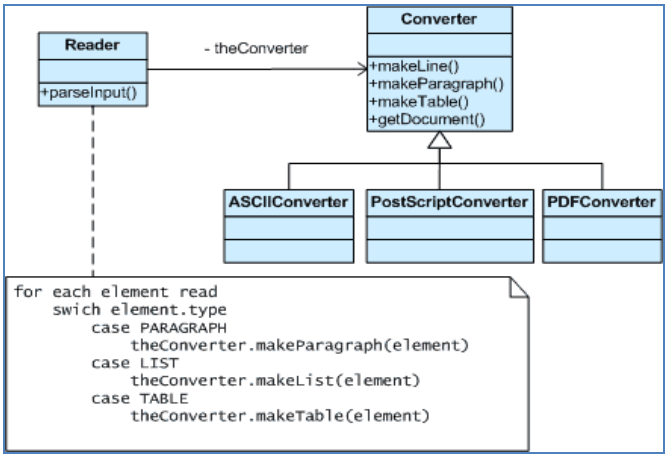

Figure 5. Builder Design Pattern

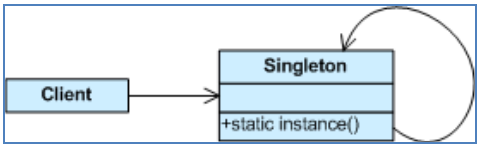

Figure 6. Singleton Design Pattern

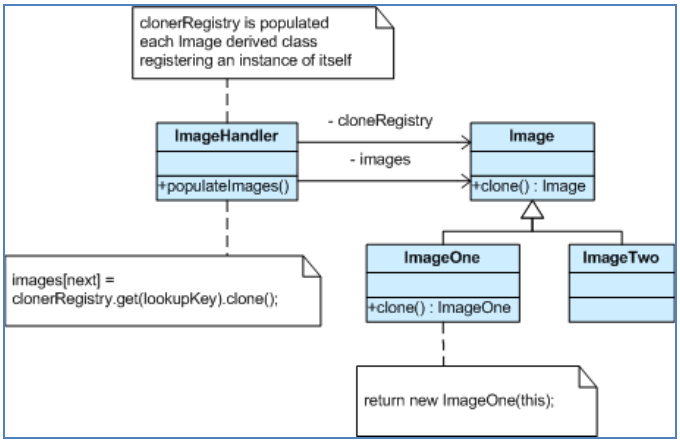

Figure 7. Prototype Design Pattern

At first sight when we are looking at those UML diagrams which represents each design pattern from creational category, we say, that is impossible to find general elements which can be used for parameterization of design patterns. The UML diagrams presented above are very precise and well defined for each design pattern [7].

IV. GENERALIZATION AND PARAMETERS FOR CREATIONAL DESIGN PATTERNS

The next step is to find the general parameters and to create a parameterization for creational design patterns. This chapter is a functional programming approach. Regarding the use of functional programming, we have a faith that functional programming languages are good for test environments of language features.

We have start in the beginning with a quick review of creational design patterns. We can look at design patterns as *high-order datatype-generic programs (HODGPs)* when we want to build a general common way for them.

In chapter three we have study each pattern and we have captured and demonstrate that each of the patterns can be put in terms of higher-order datatype-generic programs (HODGPs).

In this moment we are ready to demonstrate with concrete examples how general elements and parameters are extracted for different cases from real life.

The goal of the *Abstract factory* is to give an interface for creating different families of related or dependent objects without specifying their concrete classes.

This pattern is very handy in systems where we want to expose a library of different products to be used by a client.

In the illustration, which is in C#, we want to demonstrate and to create similar helicopters by using an example from real life.

Both manufactures sell one single car engine and multiple engine cars, with similar design, but with different drive characteristics. We start by defining an interface for the Abstract factory as well as the cars in their own namespace.

```
1  public interface ISingleEngineCar
2  {
3      void Drive();
4  }
5
6  public interface IMultipleEngineCar
7  {
8      void Drive();
9  }
10
11 public interface ICarFactory
12 {
13     ISingleEngineCar CreateSingleEngineCar();
14     IMultipleEngineCar CreateMultipleEnginerCar();
15 }
```

We implement ISingleEnginerCar and IMultipleEnginerCar for the most important manufactures Chrysler and Mercedes as private classes. When a client wants a car, they just invoked the interface library and use the specific concrete factory:

```
public class TheClient
{
    static ChryslerFactory  m_oCF = new ChryslerFactory();
    static MercedesFactory  m_oMF = new MercedesFactory();

    public static void Main(string[] args)
    {
        ISingleEngineCar    oSE;
        IMultipleEngineCar  oME;

        Console.WriteLine("Creating Chrysler SE, Mercedes ME...");

        oSE = m_oCF.CreateSingleEngineCar();
        oME = m_oMF.CreateMultipleEngineCar();

        oSE.Drive();
        oME.Drive();

        Console.WriteLine("Creating Chrisler SE, Mercedes ME...");

        oSE = m_oMF.CreateSingleEngineCar();
        oME = m_oCF.CreateMultipleEngineCar();

        oSE.Drive();
        oME.Drive();
    }
}
```

In *Factory Method* we have to extend the factories and all the subclasses. To make this, is to use Factory Method pattern.

In our example, the real factories define numeric constants for the different objects that can create:

```
public class ChryslerFactory : ICarFactory
{
    public const int SEL = 1;
    public const int MEL = 2;

    public virtual ICar Create(int t)
    {
        switch(t)
        {
            case SEL:
                return new ChryslerSingleEngineCar();
            case MEL:
                return new ChryslerMultiEngineCar();
            default:
                throw new Exception("Unknown Car Type!");
        }
    }
}
```

The client, would use the factory in the following way:

```
ICar o = m_oCF.Create(ChryslerFactory.SEL);
```

If we wish to extend the factory to build other objects, the single way to do it is to derive from the Factory, and override the Create method, by invoking the principal class Create method:

```
public class OtherChryslerFactory : ChrylerFactory
{
    public const int CIT = MEL + 1;

    override public ICar Create(int t)
    {
        try
        {
            return base.Create(t);
        }
        catch
        {
            switch(t)
            {
            case CIT:
                return new ChryslerCitation();
            default:
                throw new Exception("Unknown Car Type!");
            }
        }
    }
}
```

In *Build* design pattern. We begin with the parts, defining an abstract section or part builder, abstract car part, and abstract director as interfaces which we will make real by implementing them later for other specific parts:

```
public interface ICarPart
{
    string Producer { get; }
}

public interface IPartBuilder
{
    bool BuildPart()
}

public interface ICar
{
    ICarPart Carframe { get; }
    ICarPart Engine { get; }
}

public interface ICarAssembler
{
    bool Construct();
}
```

The next step is to create concrete implementations for different various parts, for example a TurboPlus engine:

```
public class ContinentalEngine : ICarPart, IPartBuilder
{
    private const string MANUFACTURER = "TurboPlus";

    public string Manufacturer { get { return MANUFACTURER; } }

    public bool BuildPart()
    {
        Console.WriteLine("Assembling a {0} car engine...",
            MANUFACTURER);

        return true;
    }
}
```

We've combined the GetResult() in the structure of the pattern into the actual BuildPart() for brevity.

V. CONCLUSIONS

Creational design patterns allow great flexibility in the way of how software resolves the problem with creating objects.

Design patterns are traditionally expressed informally, using few words as a short sentence, pictures as UML diagrams. In this paper we have present and argued that, give the right design pattern, at least could be expressed more usefully as reusable library code.

Using an Abstract Factory allows you to control the types of classes that you create during the runtime. He encapsulates all the functionalities of object creation, isolating the implementation from clients.

By using Factory Method pattern, we can see that we have one potential disadvantage of factory methods, and this is the fact that clients might have to subclass the Creator class just to create a specific ConcreteProduct object.

In Builder creational pattern we can see that this pattern allows you to create a various of product's internal structure, as well as how it gets assembled.

Our results come as an encourage for software engineers to employ design patterns throughout the entire software development process, the design patterns should be classified logically and represented formally.

We plan to apply our researches, results and approach to additional design patterns and to develop models of complete and powerful applications that keep design patterns. We also have a desire to develop different ways to retrieve design patterns in an easy way and to secure the information between them.

References

[1] Erich Gamma, Richard Helm, Ralph Johnson, John M. Vissides.Design Patterns: Elements of Reusable Object-Oriented Software, Pearson, ISBN 0-201-63361-2, 1994.

[2] Christopher Alexander, http://en.wikipedia.org/wiki/Christopher_Alexander

[3] Kent Beck, http://en.wikipedia.org/wiki/Kent_Beck

[4] Ward Cunningham, http://en.wikipedia.org/wiki/Ward_Cunningham

[5] Object-Priented Programming, System Languages & Applications, http://en.wikipedia.org/wiki/OOPSLA.

[6] Design Patterns and Refactoring, http://sourcemaking.com

[7] M. O Cinneide, P. Nixon, "Automated Software Evolution Towards Design Patterns", Proceedings of the 4th

[8] International Workshop on Principles of Software Evolution, pp. 162-165, 2001.

[9] J. Dong, S. Yang, and K. Zhang, "A Model Transformation Approach for Design Pattern Evolutions", Proceedings of the 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems, pp. 80-92, 2006.

[10] Model Driven Architecture, http://www.omg.org/mda

[11] G. Rozenberg (Ed.), Handbook of Graph Grammars and Computing by Graph Transformation: Foundations, vol. 1, World Scientific, 1997.

[12] L. Baresi and R. Heckel, "Tutorial Introduction to Graph Transformation: A Software Engineering Perspective", International Conference on Graph Transformation, LNCS 2505, Springer, pp. 402-439, 2002.

[13] S. Burmester, H. Giese, J. Niere, M. Tichy, J. P. Wadsack, R. Wagner, L. Wendehals, and A. Zundorf, "Tool Integration at the Meta-Model Level: the Fujaba Approach", International Journal on Software Tools for Technology Transfer, vol. 6, pp. 203-218, 2004.

[14] A. Schurr, A. Winter, A. Zundorf, "Graph Grammar Engineering with PROGRES", Proceedings of Europe Software Engineering Conference, pp. 219-234, 1995.

[15] K. Zhang, D. Q. Zhang, and J. Cao, "Design, Construction, and Application of a Generic Visual Language Generation Environment", IEEE Transaction on Software Engineering, vol. 27, pp. 289-307, 2001.

[16] D. Q. Zhang, K. Zhang, and J. Cao, "A Context-sensitive Graph Grammar Formalism for the Specification of Visual languages", Computer Journal, vol. 44, pp.187-200, 2001.

[17] J. Kong, K. Zhang, J. Dong, and G. Song, "A Graph Grammar Approach to Software Architecture Verification and Transformation", Proceedings of the 27th Annual International Computer Software and Applications Conference, pp. 492-497, 2003.

[18] G. Varro, A. Schurr, and D. Varro, "Benchmarking for Graph Transformation", Proceedings of the 2005 IEEE Symposium on Visual Language and Human-Centric Computing, pp. 79-90, 2005.

[19] [Y. Zhao, Y. Fan, X. Bai, Y. Wang, H. Cai, and W. Ding, "Towards Formal Verification of UML Diagrams Based on Graph Transformation", Proceedings of IEEE International Conference on E-Commerce Technology for Dynamic E-Business, pp. 180-187, 2004.

[20] J. Kong, G. Song, and J. Dong, "Specifying Behavioral Semantics through Graph Transformation", Workshop on Visual Modeling for Software Intensive Systems, pp.51-58, 2005.