# ALUBERI - A DESIGN PATTERN FRAMEWORK

OCTAVIAN PAUL ROTARU

*University Politehnica Bucharest*
*Octavian.Rotaru@ACM.org*

MARIAN DOBRE

*University Politehnica Bucharest*
*Marian.Dobre@Amdocs.com*

## ABSTRACT

*Design patterns are an experience encapsulation mechanism, such that good designs can be applied again in similar situations. Design patterns are also a common vocabulary that facilitates and raises the abstraction level of the communication between designers and developers of object-oriented software. The availability of a tool for automatically generating the code of design patterns will be beneficial both for novice and experienced developers, helping them to overcome the inherent difficulties of a design pattern implementation.*

*This paper describes the architecture of Aluberi, a design pattern generation framework composed of a stand-alone application, a Visual Studio add-in, a pattern repository and a pattern template library. Named after the Great Spirit of the Arawak tribe, Aluberi provides two modalities of pattern generation: a complete customized generation based on a pattern skeleton taken from the pattern repository and a light generation, based on the instantiation of the pattern template classes from the pattern template library.*

*Also, the paper presents a case study based on the Pluggable Factory, indicating that the non-intrusive design pattern implementations can be generalized using templates into a pattern template library.*

***Key words:*** *code generation, design patterns, design pattern generation framework, pattern repository, pattern template, pluggable factory*

## 1. DESIGN PATTERNS – AN INTRODUCTION

Patterns are solutions based on experience to recurrent problems, describing best practices and good design. They are ways to capture experience and make it available for others.

The origin of patterns lies on Christopher Alexander's work on architectural design [2]. Christopher Alexander considers that "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" [2].

Even if Alexander was referring to patterns in architecture and urbanism, his view is also valid for object-oriented design.

Experience is an intangible but for sure valuable commodity, which distinguishes a novice from an expert. People acquire it slowly, through hard work and perseverance, and communicating it to the other is a challenge. Design patterns are a promising step towards capturing and communicating expertise in building object-oriented software. [5]

The "Design Patterns: Elements of reusable Object-Oriented Software" book of Erich Gamma, Richard Helm, Ralph Johnson and John Vlissidess, also known as The Gang of Four book [4], had a decisive role in the popularization of the patterns in software engineering. It presents a catalog of 23 software design patterns taken from numerous object-oriented systems.

Design patterns provide a common design lexicon, and communicate both the structure of a design and the reasoning behind it. [10] They allow people to understand object-oriented software applications in terms of stylized relationships between program entities. A pattern identifies the roles of the participating entities, the responsibilities of each participant and the connections between them. The use of patterns also raises the abstraction level at which designers and developers communicate, by providing a high-level shared vocabulary of solutions.

## 2. INTENT

Design patterns are to great extent work-around solutions for deficiencies in programming languages and technologies. For example, the Visitor pattern was created to overcome the lack of support for double-dispatch in nowadays object-oriented programming languages (OOPL). Design patterns offer a way to improve OOPL by reusing proven solution and to tame complexity. They resolve “misfits”, as Christopher Alexander calls them [1, 2, 3].

Even if the design patterns popularity is increasing, there is still lack of support for patterns in the development environments.

This paper addresses the problem of automatic generation of code for design patterns. A design pattern describes a solution which will most of the times lead to similar implementations. For most of the developers it becomes a burden to write the same design pattern skeleton over and over again, and then add to it only minimal customization.

We believe that every design pattern has an intrinsic skeleton or meta-pattern in it that can be automatically generated in the same way the class wizard of Microsoft Visual Studio generates the skeleton of an application or of a dialog or view class.

Our aim is to create an extension for the development environment able to generate skeletons of design patterns and a desktop application able to visualize, edit, create, generate and export design patterns.

The main advantage of this is the uniformity of the design pattern implementations that will provide an easy way of pattern recognition in the code.

## 3. CASE STUDY

If every pattern has an intrinsic skeleton then a wizard or an automated tool can be used to generate the pattern skeletons, or the patterns inside patterns.

Presuming that the above statement is true, how easy it is to detect the pattern skeletons and how useful is to automate its creation? Is the pattern skeleton generic enough as to be able to automate it?

Each pattern must be adapted every time when applying it. The general context remains the same, but the pattern should be adapted to be specific to the sub-context.

We will utilize one of the most frequently used patterns as example for detecting the pattern “heart” and eventually answering the questions above.

Dynamic Pluggable Factory is the most generic type of factory and also the most powerful creational design pattern and therefore it will be used as example in our case study. Its scope is broader than the scope of the Factory Method and Abstract Factory. Vlissides [8, 9] considers the Pluggable Factory as applicable when Abstract Factory is applicable and any of the following are true:

- Products may vary independently during the factory’s lifetime.
- Ad-hoc parameterization techniques are not flexible or extensible enough.
- The avoidance of the ConcreteFactory subclasses proliferation is required.

The Pluggable Factory has one creation method that is able to create all necessary artifacts based on artifact’s key. The factory searches in the pool for the creator that corresponds to the respective key and invokes it. The creator will return the abstract base type of the artifact. The creators register with the factory during their construction and un-register at destruction time. Figure 1 presents the object model of the Pluggable Factory pattern.

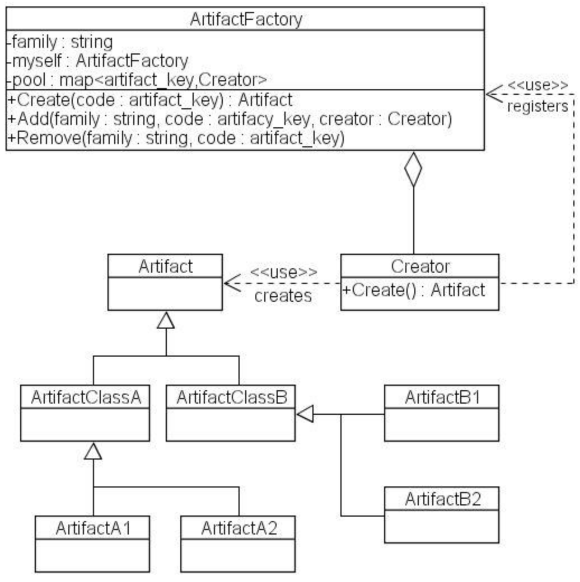
ArtifactFactory
-family : string
-myself : ArtifactFactory
-pool : map<artifact_key,Creator>
+Create(code : artifact_key) : Artifact
+Add(family : string, code : artifacy_key, creator : Creator)
+Remove(family : string, code : artifact_key)
<<use>>
registers
Artifact
<<use>>
creates
Creator
+Create() : Artifact
ArtifactClassA
ArtifactClassB
ArtifactB1
ArtifactB2
ArtifactA1
ArtifactA2

Figure 1. Pluggable Factory – Object Model

Apart from the artifacts that it instantiates, the Pluggable Factory pattern has two major players: ArtifactFactory and Creator. We will discuss about their implementations in order to determine their skeletons and see how generative they are.

A typical implementation of the Factory class of a Pluggable Factory will look like:

```
// Factory implementation
class ArtifactFactory {
   // static member for myself (singleton)
   static ArtifactFactory m_myself;

   // creators pool
   std::map <string, Creator *> m_pool;

   // private default constructor
   ArtifactFactory () {}
public:
   static ArtifactFactory * Instance()
   { return &m_myself; }

   // register creator with the factory
   void Add ( string key, Creator * c)
   { m_pool[key] = c;}

   // unregister
   void Remove (string key)
   {
      if ( m_pool.find(key) != m_pool.end())
         m_pool.erase(key);
   }

   // create artifact
   Artifact * Create (string key)
   {
      if ( m_pool.find(key) != m_pool.end())
         return m_pool[key]->Create();
      else return NULL;
   }
};
```

The family information specified in the object model presented in Figure 1 was removed for simplicity reasons. The Pluggable Factory implementation presented above constructs only objects from one family and its key is a string.

The implementation of the ArtifactFactory class presented above has only two elements that can vary: the type of the key (italic in the code example) and the Artifact (underlined in the code example). The Creator can be considered as being the same because it can be generated every time with same name.

Reintroducing the family information that was removed earlier brings the total number of variable parameters to three.

A typical implementation of the Creator as a self-registering object is the following:

```
struct Creator
{
   string m_code;
   virtual Artifact* Create() const = 0;

   Creator( string & code)
      : m_code(code)
   {
      ArtifactFactory::instance()->Add(
         code, this);
   }

   ~Creator()
   {
      ArtifactFactory::instance()->Remove(
         m_code);
   }
};

template <class SpecArtifact>
struct SpecializedCreator : public Creator
{
   virtual Artifact * Create() const
   {
      return new SpecArtifact;
   }

   SpecializedCreator(
      string code)
      : Creator( code) {}
};
```

The implementation of the Creator presented above has the same variables as the implementation of the Factory.

To conclude the discussion, the Pluggable Factory design pattern can be generated using an automated tool, its variable parameters being:

- Family type
- Key type
- Artifact base type

The registration of <key, Creator> pairs to the factory by instantiation of the SpecializedCreator can also be generated. The extra parameter that must be specified by the user is a list of <code, Concrete Artifact> pairs. However, the user can manually do the registration at the factory, since it requires only the construction of a SpecializedCreator object, persistent as long as the factory is necessary.

A template is a parameterized class. The Pluggable Factory design pattern can also be generalized using templates. In this way the amount of code generated for every specific implementation will drastically decrease. The pattern generation process will only create an instance of the pattern template, based on the instantiation parameters supplied by user.

The Pluggable Factory design pattern implementation can be generalized using templates in the following way:

```
// Creator class
template < class KEY, class ARTIFACT>
struct Creator {
   KEY m_code;
   virtual ARTIFACT * Create() const = 0;

   Creator( KEY & code)
      : m_code(code)
   {
      ArtifactFactory::instance()->Add(
         code, this);
   }

   ~Creator()
   {
      ArtifactFactory::instance()->Remove(
         m_code);
   }
};

// SpecializedCreator class
template <class SPEC_ARTIFACT,
   class KEY, class ARTIFACT>
struct SpecializedCreator
   : public Creator< KEY, ARTIFACT>
{
   virtual ARTIFACT * Create() const {
      return new SPEC_ARTIFACT;
   }

   SpecializedCreator( KEY code)
      : Creator< KEY, ARTIFACT>( code) {}
};

// Inner ArtifactFactory class
template < class KEY, class ARTIFACT>
class InnerArtifactFactory
{
protected:
   std::map<KEY, Creator< KEY, ARTIFACT> *>
      m_pool;
public:
   // Adds a KEY / Creator pair
   void Add ( KEY key,
      Creator<KEY, ARTIFACT> *creator) {
      m_pool[key] = creator;
   }

   // Removes a KEY / Creator pair
   void Remove (KEY key) {
      if ( m_pool.find(key) != m_map.end())
         m_pool.erase(key);
   }

   // Constructs an ARTIFACT based on a KEY
   ARTIFACT * Create (KEY key) {
      if ( m_pool.find(key) != m_pool.end())
         return m_pool[key]->Create();
      else return NULL;
   }
};
```

The ArtifactFactory class will become now a singleton façade for the instantiation of InnerArtifactFactory:

**Example Parameters:**

```
        KEY = string
        ARTIFACT = Product
```

**Implementation:**

```
// Factory implementation
typedef InnerArtifactFactory< string,
      Product> MyInnerFactory;

class ArtifactFactory
{
   // static member for myself (singleton)
   static ArtifactFactory m_myself;
   // InnerArtifactFactory instantiation
   MyInnerFactory m_InnerFactory;
public:
   static MyInnerFactory * Instance()
   { return &(m_myself.m_InnerFactory); }
};
```

**Registration:**

```
SpecializedCreator< ProductA,
   string, Product> a("a");
SpecializedCreator< ProductB,
   string, Product> b("b");
SpecializedCreator< ProductC,
   string, Product> c("c");
```

The InnerArtifactFactory object handles the operations of registration, un-registration and creation. The ArtifactFactory is a singleton object that transfers the control to the InnerArtifactFactory contained whenever its Instance method is invoked. The same singleton Factory class can now contain multiple InnerArtifactFactory instantiations, parameterized differently. An application can use a single Factory class, that instantiates InnerArtifactFactory objects for all required artifact base types and provides access methods to them.

Most of the design patterns contained in the GoF book can be parameterized in the same manner described above for Pluggable Factory. Starting from this idea, we created a template library of design patterns, which will be presented in a future paper.

As demonstrated by the case study presented in this section, design patterns have skeletons, which are generic enough as to be automatically generated or implemented using templates

## 4. ARCHITECTURE

A design pattern gives a language independent solution to a particular problem. The implementation of a design pattern at a given location and in a specific programming language is developer's responsibility. Implementing a design pattern requires usually experience. It is quite complicated for inexperienced programmers to jump from a pattern description to a particular implementation.

Our Design Pattern Generation Framework was developed to address these requirements. It creates design pattern instances for specific contexts by customizing their skeleton, as presented in the previous section. We named it Aluberi, after the name of the remote supreme god of the Arawak mythology, a Native American tribe from Guyana. Some also considers Aluberi the creator or the supreme spirit and he is distantly aloof.

It is worth mentioning that Aluberi is not the first tool of this kind and certainly not the last. Frameworks and tools addressing the same requirements are described in [5, 11, and 12]. We tried to learn as much as possible from their experience in order to make our framework better.

Aluberi provides two possible ways of generation for a design pattern: Light and Heavy.

The light generation will consist of an instantiation of the template skeleton of the pattern, contained by the pattern template library. The amount of code generated will be minimal, but the project in which the pattern instance will be used will depend on the pattern template library. On the other hand, the heavy generation will produce all the code required for a specific context.

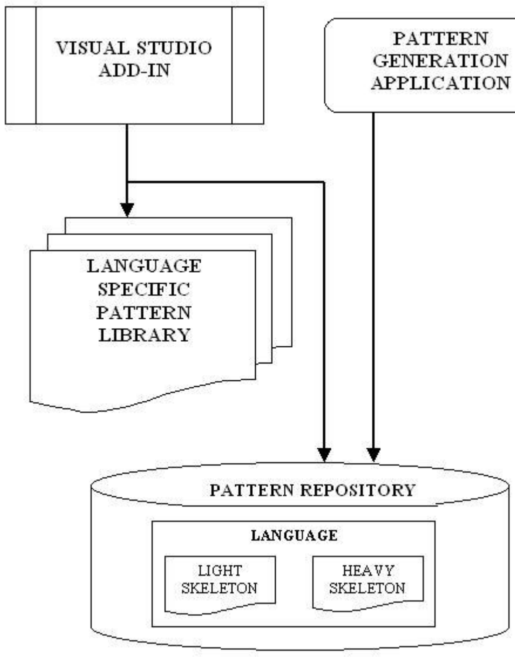

Figure 2. Aluberi's Architecture

As shown in Figure 2, Aluberi uses a pattern repository, which is shared by two different clients: a Microsoft Visual Studio (VS) add-in able to generate the pattern skeleton directly into a VS project and a stand-alone application, able to display information about each pattern, manage the pattern repository and generate the patterns independent of the IDE.

Aluberi is able to generate code for multiple languages, as long as the pattern repository contains corresponding pattern skeleton files.

Each design pattern skeleton has a correspondent file in the pattern repository for each programming language. The pattern skeleton file contains details of the code that will be generated for each pattern, its customization parameters and hooks. Also, the generated code contains comments detailing about what remains to be done and is implementation specific.

Adding a new pattern to the repository is a simple task that can be done using the pattern editor provided by the pattern management stand-alone application included in our framework. Also, new programming languages can be defined at user's discretion.

The skeleton is divided into sections and the customization parameters are defined accordingly. Each section will be generated once or multiple times, depending on its parameters. In case a section should be generated multiple times a list of values should be provided for at least one of its parameters.

For example, in case of a Pluggable Factory the registration is a repetitive section initialized with the list of types that it can construct.

Aluberi provides the user interface necessary to visualize, create and edit the brief descriptions of the intent, motivation and applicability for each of the patterns contained in the repository.

A pattern skeleton created on a computer can be easily transferred to another, since it is encapsulated in a file. The only operation required in order to register a pattern file with application is to copy it into the folder from which the application reads the patterns.

## 5. CONCLUSIONS

Reusing software, tools, design, experience accumulated in creating any software artifacts avoids investing again and again effort in re-crating them or in creating similar artifacts. Besides shortening the time to market of various products this leaves to designers and/or implementers more time and freedom to concentrate on the creative part of their work, adding more value to their products. Our paper deals with design patterns, one of the techniques for encapsulating experience in design and reusing it.

Based on the assumption that every pattern has a skeleton, a tool can be created to generate automatically the pattern skeletons or the patterns inside patterns. Such a tool is described in the paper, developed as part of a design pattern framework named by the authors Aluberi. The architecture consists in a pattern generation application, a Visual-

Studio add-in, a pattern repository and a language specific library.

Two possible ways of generation for design patterns are implemented: the light one will consists only in instantiation of the pattern template classes from the language specific library and the generated code will be minimal, while the heavy generation will produce all the code required for a specific context, based on the pattern skeleton taken from the repository.

Along with implementation details, the advantages and flexibility of the implemented framework are described: ability to generate code for multiple languages, capability to easy add new patterns to the repository, easiness in porting pattern skeletons from one platform to another.

We consider Aluberi a valuable helper when implementing industrial software applications using design patterns that brings design pattern implementations just one click away. It also induces uniformity to the design pattern implementations, allowing fast, easy and accurate design pattern recognition from the code.

## 6. REFERENCES

[1] Christopher Alexander, "Notes on the Synthesis of Form", *Harvard University Press*, 1964.

[2] Christopher Alexander, S. Ishikawa, M. Silverstein, "A Pattern Language", *Oxford University Press*, 1977.

[3] Christopher Alexander, "The Timeless Way of Building", *Oxford University Press*, 1979.

[4] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, "*Design Patterns – Elements of Reusable Object-Oriented Software*", Addison-Wesley, 1995.

[5] F. J. Budinsky, M. A. Finnie. J. M. Vlissides, and P. S. Yu, "Automatic Code Generation from Design Patterns", *IBM Systems Journal, Object Technology*, Volume 35, Number 2, 1996.

[6] Kent Beck, Ron Crocker, James O. Coplien, Lutz Dominick, John Vlissides, "Industrial Experience with Design Patterns", *Proceedings of The 18th International Conference on Software Engineering (ICSE-18)*, Berlin, Germany, 1996.

[7] Ellen Agerbo, Aino Cornils, "How to preserve the benefits of Design Patterns", *Proceedings of OOPSLA'98*, Vancouver, B.C., Canada, 1998.

[8] John Vlissides, "Pattern Hatching – Pluggable Factory, Part I", *C++ Report*, November – December 1998.

[9] John Vlissides, "Pattern Hatching – Pluggable Factory, Part II", *C++ Report*, February 1999.

[10] S. MacDonald, D. Szafron, J. Schaeffer, J. Anvik, S. Bromling and K. Tan, "Generative Design Patterns", *Proceedings of IEEE Automated Software Engineering 2002 (ASE2002)*, Edinburgh, United Kingdoms, September 2002.

[11] T.Tung Do, Manuel Kolp, T. T. Hang Hoang and Alain Pirotte, "A Framework for Design Patterns in TROPOS", *Proceedings of the 17th Brasilian Symposium on Software Engineering (SBES'2003)*, Manaus, Amazonas, Brasil, 2003.

[12] Somsak Phattarasukol and Daisy Sang, "Design Pattern Integrated Tool", *Proceeding of OOPSLA'04*, Vancouver, B.C., Canada, October 24-28, 2004.