A Software Architecture Proposal Artistic Engineering Environment -AEE-

Sandro Javier Bolaños Castro
Faculty of Engineering, District University Francisco José de Caldas, Bogotá, Colombia
Rubén González Crespo
College of Engineering and Architecture, Pontificial University of Salamanca, Madrid, Spain

***Abstract*—this article presents the "Artistic Engineering Environment" framework (AEE). It is an environment that supports service-centered components. These components facilitate software development based on a mixture of engineering and art. The aim is to promote the combination of these two disciplines to empower software engineering. Our fundamental purpose is to present the architecture that supports AEE.**

***Index Terms*—Software engineering, art, framework, architecture.**

I. Introduction

Process-centered Software Development Environments (PSEE) [1], tools like CASE (Computer Assisted Software Engineering) [2], and, in general, software intended to support development processes focus on facilitating programming, design, testing or management tasks, neglecting important aspects such as aesthetics, harmony, and software use itself [3]. Environments should not be swiss army knives or golden hammers [4], [5], they should have intersection points between complexity, aesthetics and functionality. The simplicity of a given tool is reflected in its easy use. Functionality must be complemented by style and an aesthetic awareness, which provide a harmonious set of engineering and art that leads to quality [6].

II. The Philosophy of Art-Engineering Environment

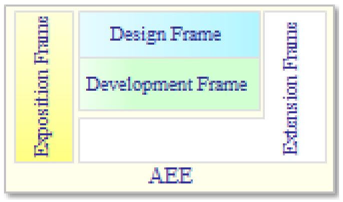


Fig. 1: AEE Scheme.

AEE draws attention to two fundamental pillars, namely engineering and art, Fig. 1.

A. Engineering

From an engineering point of view [7], AEE proposes a set of principles as follows:

1) Simplicity: Inspired by Occam's razor principle [8], simplicity [9] represents the other side of the coin when addressing a problem, not from the standpoint where complexity is frightening right from the start, but from a peaceful counterpoint where the job to be done is suggested to be comfortable and manageable. AEE reflects its simplicity in a set of well-defined principles, even from an implementation perspective when we look at the small set of interfaces to be implemented.

2) The Metaphor: AEE's framework was created using an art gallery as a metaphor [10]. The gallery offers frames for Exposition, Expression and Extension, and the purpose of the frames is to depict the nature of an application from viewpoints that range from general artifact creation to a view of the most specific artifacts. The main concept of this metaphor is the frame; this concept gathers internal semantics according to the environment. The Exposition frame, for example, holds the concept of author reference and also a description of the artifacts arranged in a hierarchical tree. The expression frame, on the other hand, is used to depict the work of art in the environment, highlighting the works that resemble paintings (i.e. modeling) and also those that resemble literature (i.e. development). The extension frame makes it possible to expand expression semantics using two approaches, namely an approach intended to integrate the different views about a problem, and a presentation approach that highlights visuals.

3) Organization: By creating a visual distribution, the aim is to have an impact upon creative development, which focuses on deduction, that is, we start from a framework and go down to the details making a thorough scanning of the concept from left to right and top to bottom. In an intentional fashion, the organization [11] decides that the Exposition frame, from which they reference the "work-of-art" project, be placed on the left, emphasizing that this project should be done first. The center is occupied by the expression frame, while the different views of the extension frame are placed at the bottom and the presentation of this latter frame is placed on the right.

B. Art

From the Artistic viewpoint, two principles stand out, namely aesthetics and style.

1) Aesthetics: This principle refers to aesthetics as the branch of philosophy that studies both the essence and the perception of beauty [12], [13] reflected in the color given to AEE. Yellow, blue and green are the colors chosen to paint frames. Yellow, which is the color chosen for the Exposition frame, reflects creativity and brightness [14]. Blue, which is the color chosen for the expression framework, represents harmony, science and intelligence [14]. Finally, green, which was a color reserved for the extension frame, represents everything that is fresh and functional. [14].

2) Style: This principle promotes customized environments, allowing the adjustment of iconography. At first universally recognized icons are proposed as a component; however, the environmental icon suite can be assembled according to the intentions. By default AEE's style [15] promotes the use of warm colors so that a sense of comfort, simplicity and easiness pervades the environment.

III. Architecture

Software architecture is the structure of all structures within the system. This structure comprises software components and their properties together with the way they relate to each other [16]. Shaw and Garlan consider software architecture as the starting point, where modules are established and relations are defined [17]. Architecture allows configuring a system without having to get into implementation details; moreover, architecture leaves a distinguishing mark on every software solution, similar to conventional architecture and actual buildings. AEE's representation architecture uses graphic representations such as modules, interfaces and components that illustrate the way in which the environment's structure was conceived.

A. Three-tier Model

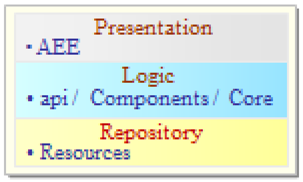


Fig. 2: AEE Multi-tier Architecture.

The most general view of the environment is defined using a layered architectural style [18], which consists of a presentation layer, a logical layer, and a repository layer. Fig 2.

In the presentation layer, the *ArtisticEngineeringFrame* module is defined as the fundamental abstraction, and it represents the ingress point to the platform. In the logical layer, the API (Application Programming Interface) is established. The following elements are condensed into the API: the application semantics, the components together with a set of services required by the environment, and the core, which is the assembly of all the frames that permit the construction of applications over the environment. Finally, the repository layer comprises the application resources, where icons and configuration files can be found.

B. AEE as a Framework

A framework is an architectural pattern that provides an extensible template for applications, facilitates development, and promotes components reuse. Within a framework, an architectural skeleton is specified together with slots, thumb indexes and fragments that are provided for users, who shall adapt the framework for their own context [19], [20].

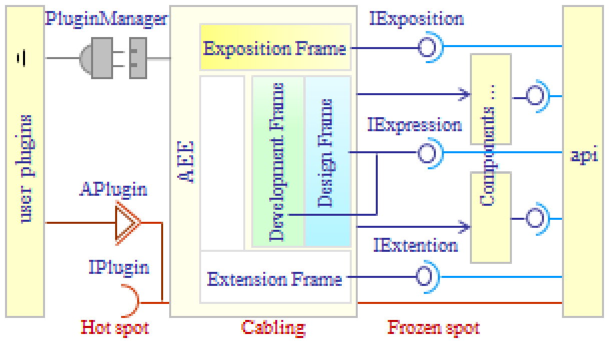


Fig. 3: AEE Framework.

The AEE framework has three abstraction layers, Fig. 3. The first layer comprises the api. This layer is completely abstract and defines the set of primary services from which Frozen Spots and Hot Spots are obtained. A Frozen Spot is the point in an application that has already been implemented and keeps the essential conditions for the framework to work; AEE has three main Frozen Spots that represent the interfaces implementation, namely *IExposition, IExpression* and *IExtension*. These spots appear together with other spots that are implemented with some components that will be used by the environment. A Hot Spot is a point that enables extending and customizing applications. These spots are left for users to implement them. The main Hot Spot is *IPlugin*, which allows creating connectors to the environment; these connectors will be managed using the *PluginManager* module.

The second layer of the framework is the cabling layer. This layer consists of the environment's frames, namely Exposition, Expression and Extension, included in and managed from the *ArtisticEngineeringFrame*, which is the starting point to assemble the user's customized applications. The last layer is the user layer, also known as user plugin. User's applications to be deployed in the environment are found in this last layer.

C. API

API is a mechanism to conceive applications from a higher level of abstraction. APIs facilitate problem understanding from software upper layers; they also improve modularity [21] and constitute a fundamental ingredient for the components assembly.

Application programmable interfaces model the main functionalities that will be provided by the environment. For the Exposition frame implemented by the *IExposition* interface, functionalities of some renown are developed, namely movement, erasure and closing. The Expression frame implemented by the *IExpression* interface develops the following functionalities: storage, erasure and printing. Furthermore, the

Exposition frame implements the special interface called *IPart*, which allows any extension of this concept to be considered as a multiple-piece component of the user's customized jigsaw. Finally, the Extension frame should implement the *IExtension* interface, which is the interface that makes it possible to add special controls to manipulate its content, e.g. copy and erasure controls. Fig. 4.

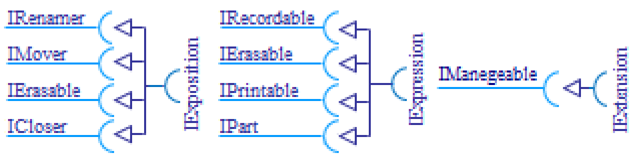


Fig. 4: api AEE.

D. Component

A component represents a modular piece from either a logical or physical system whose external behavior can be more consistently described than its implementation. Components are connected to one another through interfaces, allowing a free exchange and replacement of components as long as the new components are supported by the corresponding interface. Likewise, a component in the inside can be made up of other components so as to constitute more complex modules [20]. In order to promote code reuse, components must be standardize, independent, compose-able, deploy-able and document-able [22]. Fig. 5.

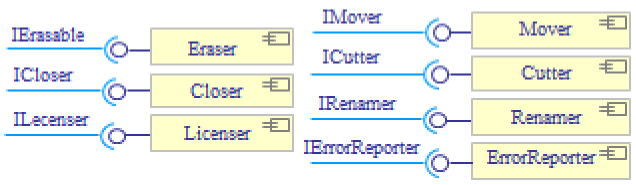


Fig. 5: AEE Components.

AEE Components, Figure 5, allow solving the following functionalities:

- *Eraser*: eliminates a particular node from the Exposition frame tree.
- *Closer*: closes the thumb indexes addressed by the selected node within the expression frame, which in turn also closes its corresponding extension frame.
- *Mover*: allows moving an Exposition frame leaf-node to any package within the same Exposition framework.
- *Renamer*: allows changing the name of a node.
- *Cutter*: is used in the extension frame to take a copy of the corresponding frame as an image and so write reports easily.
- *Licenser*: manages the information that will be displayed by the environment for copyright purposes.
- *ErrorReporter*: was designed to deal with the defects that may appear in the environment. It will automatically send reports to an Internet e-mail, or else to an error log.

E. Frames

A particular type of organization for the art-engineering environment is the one made by using the frames approach. Frames establish the limit and the modular content. Frames are represented as high-level classes and serve to model the Exposition, the Expression and the Extension in an environment.

1) Exposition Frame : The Exposition frame *ExpositionFrame* uses the functionalities implemented by the following components: *Eraser, Closer, Mover and Renamer*. This allows managing the application from this framework. Such functionalities represent the realization commitment as expressed through the interface *IExposition*, but the commitment is delegated to each of the components according to the delegation principle [23], which promotes reuse and flexibility within the application. Fig. 6.

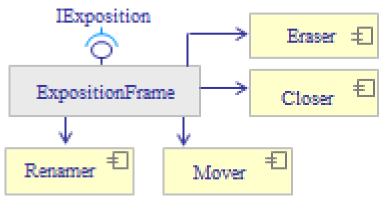


Fig. 6: Exposition Frame.

2) Expression Frame: The expression frame *AExpressionFrame* has two specialized possibilities, namely the development framework and the modeling framework. Fig.7.

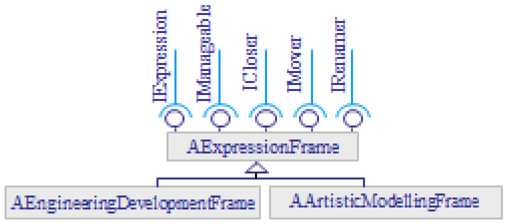


Fig. 7: Expression Frame.

The development framework can be used for the integration of plugins such as text editors, whereas the modeling framework is used for the integration of design plugins. An expression frame is managed trough control services, namely closing, movement, and renaming; moreover, plugins can be printed, stored and deleted.

3) Extension Frame : The Extension frame *ExtensionFrame* allows the expression-frame information to be expanded through both perspective and presentation specializations.

Perspective permits having expression-frame projections, which allows the semantics of an application to be expanded. Some of the projections are, for example, a metrics view or a profile view. The perspective specialization uses the component *Cutter* to take a screenshot that can be cut and then pasted somewhere else, which facilitates making reports. The presentation specialization is primarily used to display the expression-frame information in a hierarchical fashion (tree). It can also be used to display a miniaturized view of the expression frame when the size of such a frame makes it difficult to explore it. An extension frame is controllable, that is, it can have controls like the erasure control. Fig. 8.

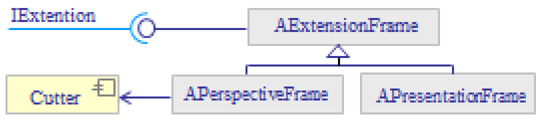


Fig. 8: Extension Frame.

4) Artistic-Engineering Frame: The artistic-engineering frame *ArtisticEngineeringFrame* is the fundamental frame. This frame comprises the exposition, expression and extension frameworks. Plugins are managed from this principal frame by using the *PluginManager* module. The artistic-engineering frame also manages resources, licensing (copyright) and the error reporter by using the corresponding components. This fundamental frame is located in the presentation layer and constitutes the framework's ingress point, Fig. 9.

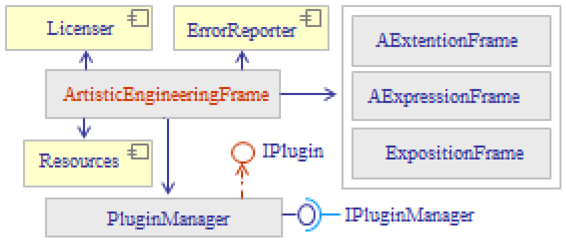


Fig. 9: Artistic-Engineering Frame.

F. Resources

Resources are a basic component of the environment, which is made up of menus, labels, dialogues, tool-tips, and icons. Internationalization is managed through resources [24], which are fundamental to the universalization of an application. Initially, the platform supports two languages, namely English and Spanish, and it has the possibility to be easily extended to more languages without altering the structure of the application.

G. AEE as Plugin Support

A plugin is an application that is linked to other application in order to provide it with more or better functionalities, normally very specific functionalities, Fig. 10.

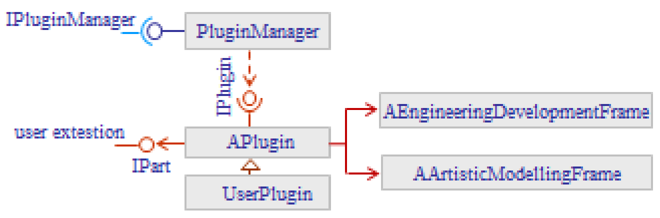


Fig. 10: AEE as Plugin Support.

In AEE the benefits of the plugin concept are exploited. The fundamental idea is to allow new components to be plugged to the existent system. The AEE-plugins architecture focuses its development of plugins on the *APlugin* class, which should specialize in a particular plugin according to the developer (user).

The *APlugin* class has the user's engineering development frame specializations and the artistic modeling specializations, as well as the extensions, by means of the interface *IPart*. *APlugin* is an implementation of *IPlugin*, which is uploaded on the AEE platform using reflection mechanisms, exploiting the dynamic-language characteristics of java. This plugin task is performed by *PluginManager*, the specific role of this module is to read the plugin's route, i.e. the manifesto [25], which indicates the properties of a given plugin.

H. AEE design Patterns

Christopher Alexander put forward the following definition: "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 in the same way twice."[26].

1) GoF Patterns: The GoF design patterns [26], together with other architectural patterns [27], [28], compile a set of design templates whose solutions permit software to be flexible and robust. For the AEE platform, two patterns are used to establish the relationship between the different parts, namely mediator and observer.

a) Mediator: One of the main difficulties is the strong interaction between the Exposition and Expression frames due to the massive information flow between them. The expressions that need to be managed in the environment should pass through the expression frame, which is the place where both the design and development frames are created and managed; therefore these two frames must have a close but decoupled bond in order to avoid stiffness in the application. The mediator pattern has the precise features to model a solution for this scheme. This pattern defines an object that encapsulates the way a set of objects interact. The pattern promotes a coupling loss by concentrating the references and allowing independent interaction.

b) Observer: The extension frames that are created share a common characteristic, namely they display information about the expression frame where they were created; following these ideas, extension frame observe their own behavior continuously in order to update the perspective they exhibit. The observer pattern defines a one-to-many dependence between objects so that whenever an object changes its state, all its dependencies are notified and automatically updated.

2) VRAPS Patterns: VRAPS patterns are a set of patterns gathered according to five principles, namely Vision, Rhythm, Anticipation, Partnering and Simplification [29].

a) Vision: Vision is the mapping of a future value to architectural constraints as a realization measurement of architecture structure and objectives, which should be clear congruent and flexible. Out of the patterns proposed by the Vision principle, for the case of AEE architecture, the chosen pattern was *Generative Vision*, since the architecture was developed to fulfill the requirements of a simple and robust architecture. This requirements phase was conducted with users of platforms such as *netbeans* [30] and *eclipse* [31], which are robust but yet complex.

b) Rhythm: Rhythm is the recurrent and predictable change of work products with respect to an architectural product, through clients and providers. Out of the patterns proposed by the Rhythm principle for the case of AEE architecture, the pattern chosen was *Drop Pass*. This pattern is evident in terms of releases. These releases have kept pace with development, where the most complex functionalities have been solved during the evolution of the environment, without the architecture being affected thanks to the separation of functions on a component-basis. The strategy developed to cope with the development of the architecture is a gradual increase in the implementation, considering a set of minimal functions per release.

c) Anticipation: Anticipation allows the people who build and implement the architecture to predict, validate and adapt the architecture according to the technological changes, competences, and necessities of users. Out of the patterns proposed by the anticipation principle, for the case of AEE architecture, the chosen pattern was *Architecture Review*. This pattern is evident in the permanent reviews of the AEE architecture. This review has followed the evolution of both the java implementation language and the SWT framework. Besides updating the different versions, such monitoring has been useful to correct the mistakes found in previous versions due to the evolution of SWT and the appearance of java version 5.0.

d) Partnering: Partnering clearly defines cooperative roles. Out of the patterns proposed by the partnering principle, for the case of AEE architecture, the pattern chosen was *Know the Stakeholders* because of the importance associated to users who have used Beta versions of the product and whose opinions have been highly valued in order to improve AEE.

e) Simplification: Simplification means intelligent minimization and clarification of both the architecture and the organizational environment where operation occurs. Out of the patterns proposed by the simplification principle, for the case of AEE architecture, the pattern chosen was *Slow down to Speed up*. This pattern allowed acceleration in the architectural development, especially when considering that this pattern required a significant period of time for reflection on how to build a simple, yet robust structure.

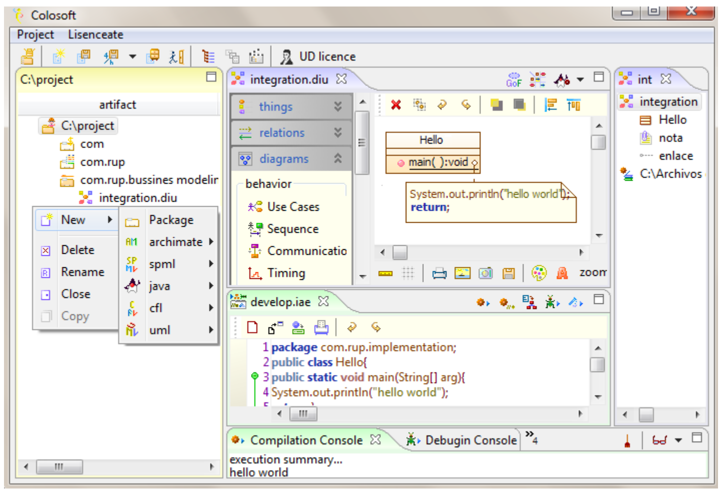
IV. Results

TABLE I: Coloso Platform

Frameworks	plugins	components
Engineering:		
- AEE	- UML diagrams	- GoF patterns
- uml	- archimate viewpoint	- profiles
- archimate	- java compiler	- metrics
- java	- java debugger	- process template
- process	- process viewpoint	- appearance
Artistic:		
- appearance		- icons

AEE architecture represent the structure of the Coloso platform [32], where a set of frameworks, plugins and software design components have been implemented; the artistic approach of the platform, supported by its visual layout (colors) and its icons is provided with a framework and a component that can be customized and/or replaced according to the user's needs Table I.

The Coloso platform has been used for academic and entrepreneurial purposes, Fig. 11.



```
System.out.println("hello world");
return;
```

```
package com.rup.implementation;
public class Hello{
public static void main(String[] arg){
System.out.println("hello world");
```

```
execution summary...
hello world
```

Fig. 11: Coloso Version.

some findings of its use include the following:

- The environment is easy-to-use once its philosophy of use has been explained; the first impression made on most useful scenarios tends to underestimates its functionalities due to the proposed simplicity-based design.
- The implemented languages offer robust support when modeling, and although non-supported mechanisms have been detected, these have been an important input to refine the modules.
- The difference of using the Coloso platform, compared to other environments, was more noticeable when identifying significant contributions of the components associated to metrics, traceability, patterns, profiles, and mostly the development process-and-methodology templates, which make of Coloso a PSEE platform that integrates components and plugins into a single conceptual unit.
- Still unexpected bugs are found, and the repair protocol, triggered from the tool, has allowed gradually debugging and stabilizing the platform.
- The main weakness, explicit from its use, was identified in the lack of modules that serve to integrate other implementation languages like c++, php and sql.

Fig. 12 highlights the most important features of the environment that was developed on the proposed architecture.

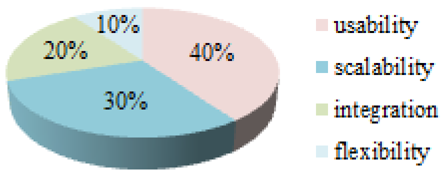


Fig. 12: AEE Main Features.

V. Conclusions

The motivation to develop AEE was to obtain a simple, yet robust and independent architecture to integrate components intended to support software development. This proposal is aimed at developers who wish to integrate and use components to support software development. The combination of engineering concepts and art concepts strengthens software development due to the usability conditions that can be obtained from such integration.

Open platforms with plugin support, such as eclipse and netbeans, have permitted extensive use of modules that enrich software development tasks. However, this type of platforms is instable when running plugins of different versions; this is caused by the change in their basic structures. AEE keeps a stable architectural concept, which is scalable throughout versions without having to sacrifice its fundamental structure.

VI. Future Work

As future work, we propose to open this platform with the purpose of implementing more components in an open-source format through the proposed facility-and-use interface. The environment will soon be available at www.colosoft.com.co

Acknowledgment

The authors would like to thank the MIS (Modelamiento en Ingeniera de Sistemas) research group at Universidad Distrital, and also to students Arturo Arismendi and Edwin Castaeda for their supportive job.

References

[1] J. C. Derniame, B. A. Kaba, and D. Wastell, *Software process: principles, methodology, and technology*. Berlin Heidelberg: Springer-Verlag, 1999.

[2] I. Sommerville, *Ingeniera del Software*. Addison-Wesley, 2005.

[3] N. Tractinsky, A. S. Katz, and D. Ikar, "What is beautiful is usable," *Interacting with Computers*, vol. 13, no. 2, pp. 127–145, 2000.

[4] W. J. Brown, R. C. Malveau, W. H. "Skip" McCormick, S. W. Thomas, and T. H. (ed), *Anti-Patterns in Project Management*. John Wiley & Sons, 2000.

[5] P. A. Laplante and J. N. Colin, *Antipatterns: Identification, Refactoring and Management*. Auerbach Publications, 2005.

[6] J. McCall, P. Richards, and G. Walters, "Factors in software quality," in *NTIS AD-A049-014*, vol. 3, 1997.

[7] J. O'Connor and I. McDermott, *Introduccin al pensamiento Sistemico*. Ediciones Urano, 1998.

[8] J. T. Trevors and M. H. Saier, "Occam's razor and human society," *Water, Air, & Soil Pollution*, vol. 205, no. 1, pp. 69–70, 2010.

[9] J. Maeda, *Las leyes de la simplicidad*. Gedisa, 2008.

[10] G. Lakoff and M. Jonhson, *Metaforas de la Vida Cotidiana*. Catedra, 2004.

[11] M. Overdijk and W. van Diggelen, "Appropriation of a shared workspace: Organizing principles and their application," *International Journal of Computer-Supported Collaborative Learning*, vol. 3, no. 2, pp. 165–192, 2008.

[12] K. Karvonen, "The beauty of simplicity," in *Proceedings on the 2000 conference on Universal Usability*, Arlington, Virginia, United States, ACM New York, NY, USA, 2000, pp. 85–90.

[13] U. K. Yusof, L. K. Khaw, H. Yang, and B. J. Neow, "Balancing between usability and aesthetics of web design," in *Information Technology (ITSim), 2010 International Symposium in*, 2010, pp. 1–6.

[14] E. Heller, *Psicologa del color*. Gustavo Gili, 2004.

[15] H. Cohen, "Style as emergence (from what?)," *The Structure of Style, part 1*, pp. 3–20, 2010.

[16] L. Bass, P. Clements, and R. Kasman, *Software Architecture in Practice*. Addison-Wesley, 1998.

[17] M. Shaw and D. Garlan, *Software Architecture*. Prentice Hall, 1996.

[18] R. Pressman, *Ingeniera del Software Un Enfoque Practico, Quinta Edicion*. Mc Graw Hill, 2002.

[19] J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language User Guide, second edition*. Addison-Wesley, 2005.

[20] ——, *The Unified Modeling Language Reference Manual, second edition*. Addison-Wesley, 2006.

[21] B. Meyer, *Construccin de Software Orientado a Objetos, Segunda Edicion*. Prentice Hall, 1999.

[22] J. Cheesman and J. Daniels, *UML Components*. Addison-Wesley, 2001.

[23] T. Budd, *Introduccion a la Programacion Orientda a Objetos*. Addison-Wesley, 1994.

[24] C. Horsmann and G. Cornell, *Core Java Volumen II- Advanced Features*. Sun Microsystem Press A Prentice Hall Title, 2000.

[25] I. Horton, *Beginning Java 2*. Wrox, 2000.

[26] E. Gamma, R. Helm, R. Jonson, and J. Vlissides, *Design Patterns Elements of Reusable Object-Orienyted Software*. Addison-Wesley, 1995.

[27] F. Buschmann and et al., *Pattern-Oriented Software Architecture a System of Pattern*. Wiley, 1996.

[28] P. Kuchana, *Software Architecture Design Patterns in Java*. Auerbach, 2004.

[29] D. Dikel, D. Kane, and J. Wilson, *Software Architecture Organizational Pinciples and Patterns*. Prentice Hall, 2001.

[30] (2012) Netbeans. [Online]. Available: http://www.netbeans.org/

[31] (2012) Eclipse communit. [Online]. Available: http://www.eclipse.org/

[32] (2012) Coloso community. [Online]. Available: http://www.colosoft.com.co/