Understanding Object-Oriented Framework Engineering

Marcus E. Markiewicz
e-mail: mem@acm.org

Carlos J. P. de Lucena
e-mail: lucena@inf.puc-rio.br

PUC-RioInf.MCC38/00 October, 2000

Abstract: Framework engineering is an emerging area within the software engineering discipline. The framework development approach must be considered when one must meet requirements that are prone to fast change as it occurs in new areas such as e-commerce and Web-based education applications. The present paper discusses the pros and cons of using the framework development approach as well as pitfalls and open problems associated with the framework technology.

Keywords: frameworks, framework engineering, object-oriented framework

Resumo: A engenharia de frameworks é uma área nova dentro da engenharia de software. O desenvolvimento de frameworks deve ser considerado quando se deve lidar com requisitos que estão sujeitos a mudanças rápidas, como ocorre nas áreas de e-commerce e de educação a distância via Web. Este artigo discute os prós e contras de abordagens para o desenvolvimento de frameworks, assim como armadilhas e problemas em aberto associados à tecnologia de frameworks.

Palavras-chave: frameworks, engenharia de framework, framework orientado à objetos

Introduction

Perhaps one of the cornerstones of modern software engineering is object-oriented frameworks, addressed here simply as frameworks. Frameworks are application generators that are directly related to a specific domain, i.e. class of problems. For example, let’s imagine we wish build a Graphical User Interface (GUI) toolkit. Building the toolkit is the problem at hand, and GUIs are the domain. By designing the toolkit as a framework we are able to generate not only one but also a collection of toolkits for specific applications within the GUI domain. Since frameworks are created to generate applications for a whole domain, there must be flexibility points, which are customized according to the specific application solving a particular problem.

The points of flexibility of a framework are called hot spots. Due to these hot spots, frameworks are not systems, in a sense that they are not executable (i.e., do not run). In order to generate a system, a framework must be instantiated; that is, appropriate code for a particular application in the framework domain must be implemented in the framework. Most of the time, hot spots are abstract classes that have no implementation and must be customized in order to be instantiated. Once they are instantiated, the framework will use these newly added classes using the callback mechanism. For this reason, the framework approach is sometimes characterized as "old code calls new code."

Even though a framework is composed of hot spots, some aspects are impossible to be made flexible. Thus, some features of the framework will not be mutable and cannot be easily altered. These points of immutability constitute the kernel of a framework, also called the frozen spots of the framework. The kernel will be the constant and always present part of each instance of the framework.

Let's illustrate these concepts imagining that a framework is an engine. This engine must be powered in order to work, and it has many power outlets. Each of these power outlets is a hot spot of the framework. This way, each hot spot must be powered on (implemented) in order for the engine (framework) to work. The power generators are the application specific code that must be plugged to the hot spots. The application code that is added will be used by the kernel code of the framework, and the engine will not run until all plugs are connected. This metaphor is illustrated in Figure 1.

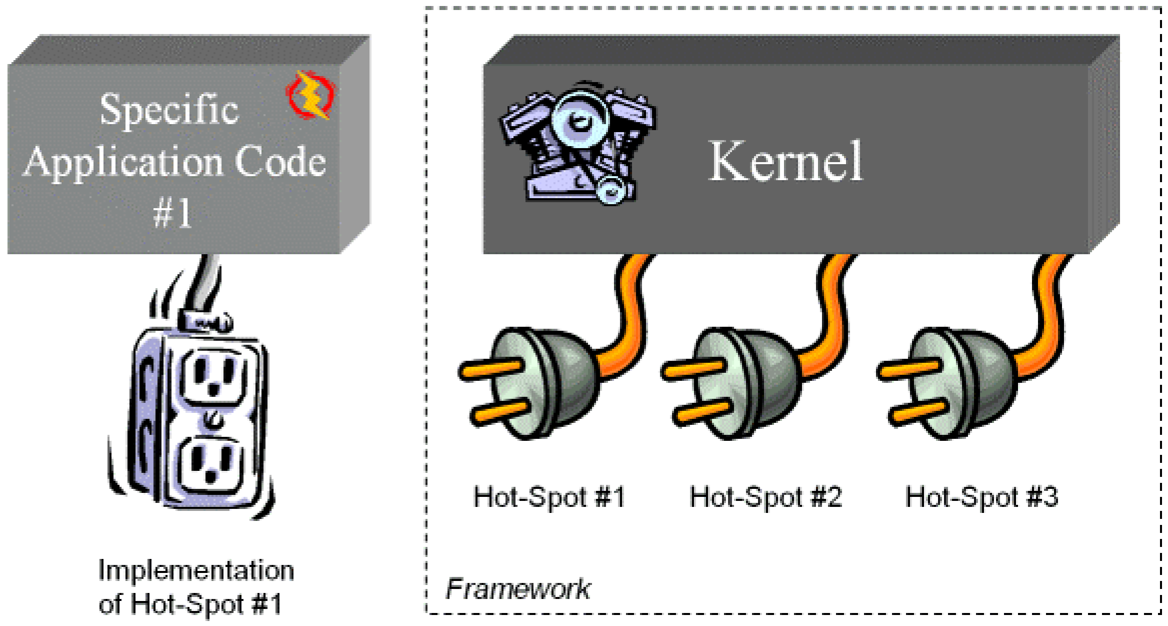


Figure 1. Frameworks

The assessment of the framework technology presented here is based on observations compiled by the authors about the development and instantiation of several frameworks for the e-commerce area in our laboratory. One of these frameworks, called V-Market, is presented in [9].

Frameworks Issues

The process of developing frameworks consists of three major phases: domain analysis, framework design and framework instantiation. The domain analysis is an extensive analysis of a complete domain. It is an attempt to catch the domain's requirements, with a special focus on the anticipation of possible future requirements. In order to capture these requirements, previously published experiences, existing similar software systems, personal experiences and standards are taken into account. In this phase the hot spots and frozen spots begin to be found.

The framework design phase concentrates the efforts on the creation of the framework's abstractions. In this phase the hot spots and frozen spots will be modeled (for example, using the Unified Modeling Language [1] diagrams), and the extensibility and flexibility proposed in the domain analysis is outlined. In order to express the hot spots, architectural solutions that emphasize extensibility are used. Design patterns [3], which are reusable architectural solutions that can be found in catalogs such as [3], are useful for this purpose.

Finally, in the instantiation phase, the framework hot spots are implemented and a software system is generated. It is important to notice that each of these applications will have in common at least the framework's frozen spots. The framework development process phases are compared to the traditional object-oriented design phases in Figure 2.

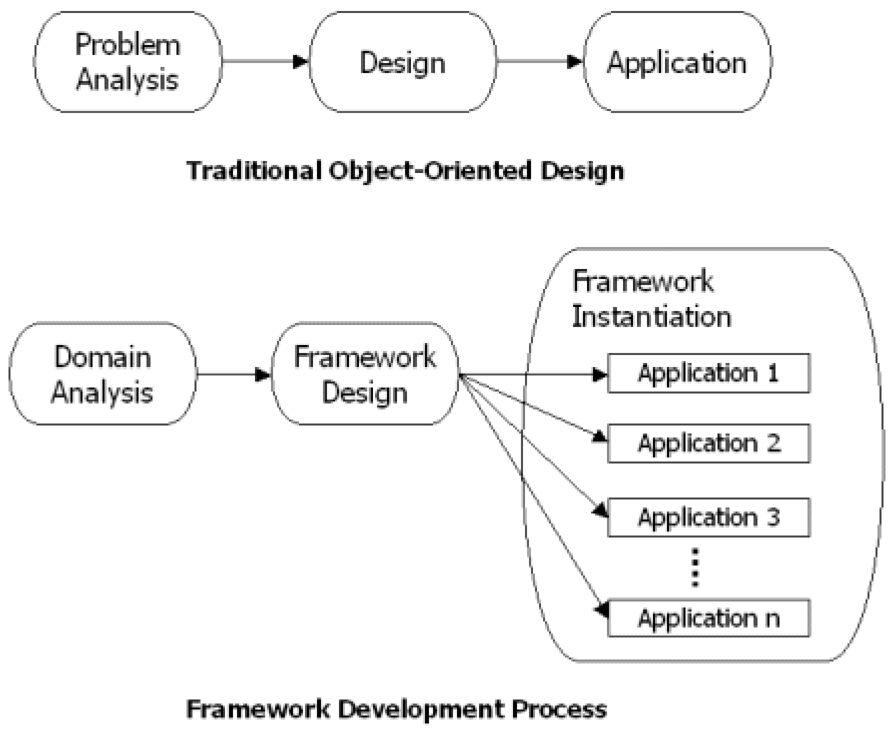


Figure 2. Development Process of Frameworks

Object-oriented frameworks promise higher productivity and shorter time-to-market of application development through code and design reuse when compared to traditional software systems development. This is achieved by the flexible architecture of frameworks, which allow for its kernel to be reused in its many instantiations, thus promoting design and code reuse. Even though these promises are sweet, there are several issues that should be discussed. Every time a method such as framework engineering is proven successful in many domains, there is a tendency to proclaim it "The Silver Bullet Solution" of all-times, a universal panacea. So, skepticism and caution are always welcome, and we must provide space for discussions of both the pros and cons of this approach.

In this spirit, in the following subsections we will review seven issues that must be considered when choosing a framework model. Notice that these points should be carefully considered: they are neither good nor bad, just tradeoffs. Nonetheless it is important to keep in mind that object-oriented framework engineering is a relatively recent approach. Also one must remember that object-oriented design and implementation practice are themselves recent developments. It is our belief that framework engineering will evolve and prove itself the rule of thumb for many domains, but surely not for all.

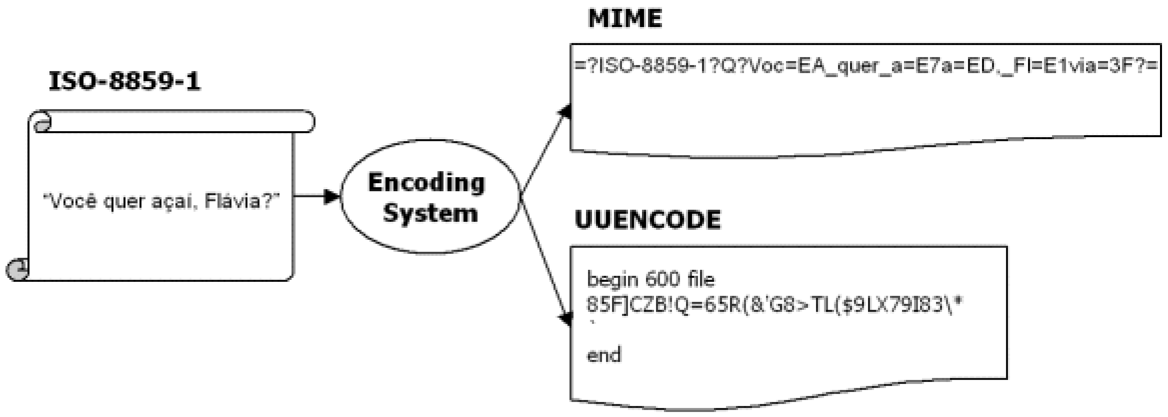
Application Generator Development vs. Application Development

As stated before, frameworks generate applications by means some sort of customization. They are not the applications themselves but, rather, are more complex constructs. It is important to keep in mind that the development of a framework will be at least as expensive as the application, and generally much higher.

One must carefully weight the need for the flexibility of a framework when assessing requirements that must be met for a client or future user. As obvious as it may seem, sometimes this point is entirely missed, allowing for the creation of behemoths that will be used one time only.

On the other hand, sometimes the effort of building application generators will pay itself off through the repeated generation of applications within the proposed domain. So, one question that must be asked before the framework model is chosen is: "Will I be creating applications of this same domain more than once ?" If the answer is yes, than it is important to assess if the work of creating more than one application will pay off the work of creating an application generator. In brief, *be aware of the cost versus benefit of choosing to develop a framework instead of a custom-made software system*.

As an example, let's imagine someone is hired to design a system that will transform text files written with non-ASCII characters, such as the ISO-8859-1 Latin character set, to an e-mail text encoding format like the Multi-purpose Internet Mail Extension (MIME) [7]. This operation comprises the encoding from one character set (ISO-8859-1) into another set such as MIME. Not considering that this is a small application, should this system be built as a framework? The answer is yes if there are plans to convert ISO-8859-1 into other formats, such as UUENCODE or even ASCII to MIME, UUENCODE or possible future formats. In the first case, one hot spot would be the type of the output text. In the second case, the type of the input text will also be a framework. For example, in Figure 3 a text written using ISO-8859-1 characters like "ç" and "á", which do not exist in ASCII, are encoded in MIME and UUENCODE.



```
=?ISO-8859-1?Q?Voc=EA_quer_a=E7a=ED._Fl=E1via=3F?=
```

```
begin 600 file
85F]CZB!Q=65R(&'G8>TL($9LX79I83\*
`
end
```

Figure 3. A sample encoding system at work

But what if this system will only convert ASCII to MIME and there are no plans of further development? In this case, a framework might be an over-elaborate approach to the problem. Sadly, most times the choice is not so clear. However, there is only one thing certain: you will find yourself in twilight zones more often than you would like, so it is always a good idea to check how similar systems have met the client's requirements. You might even discover that each similar system would be an instance of your framework, but is it still worth building?

Composition Issues

It is common for frameworks to be integrated in order to fulfill application requirements. However, Michael Mattson argues in [4], that there are at least six common problems that application and framework developers encounter when integrating two or more frameworks. He classifies four of them as related to architectural design, and the other two result from problems at the architectural design level. All of these problems derive from a set of five common causes: cohesive behavior, domain coverage, design intention, lack of access to source code, and lack of standards for the framework. These problems are detailed thoroughly in [4], where they propose many solutions to each.

If one develops a framework and expects it to be used, framework integration is an inevitable reality. These composition issues must not be taken lightly as they might cause the failure of a framework. By failure, we mean frameworks that are abandoned or aborted after proven "nice but useless." Making one framework work with another is not at all an easy task, and *composition must be considered seriously during development*.

Instantiation and Framework Documentation Issues

A framework can also be classified according to its extensibility: it can be used as a white box or a black box [2]. In white box frameworks, also called architecture-driven frameworks, instantiation is only possible through actual codification and creation of new classes. These classes and code can be introduced in the framework by inheritance or composition. One must program the framework and understand it very well in order to produce an instance.

In black box frameworks, on the other hand, instances are produced by means of scripts or some type of configuration. The classes and code are then created by an instantiation automation tool or instantiation tool of some sort, which hides the complex details from the application developer. This approach will allow those who are instantiating the framework to have less knowledge of the framework internals. For this reason, these frameworks are also called data-driven frameworks [2]. This type of framework is generally easier to be use, since little or no knowledge of the framework internals is needed. In Figure 4 we have an illustration of the concept of white box and black box frameworks. All frameworks that have both white box and black box characteristics but are not entirely white box or black box are called gray box frameworks.

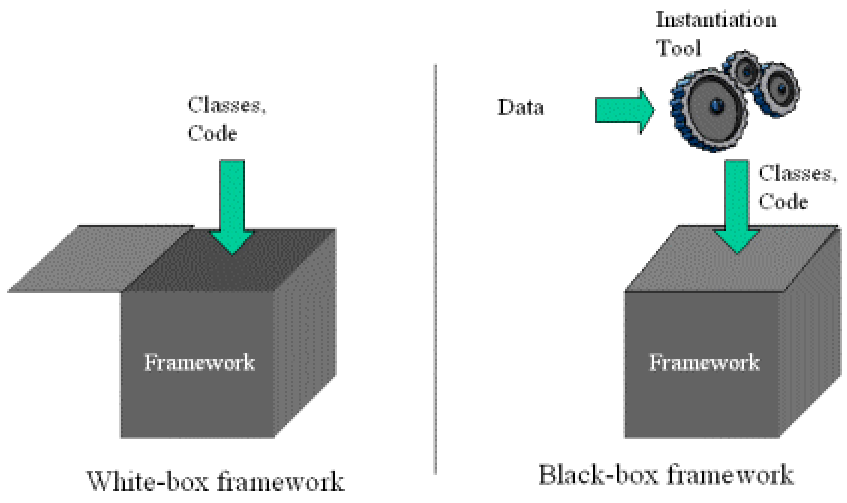


Figure 4. White box frameworks vs. black box frameworks

Either by choosing a white box, black box or mixed (gray box) approach, a learning curve is unavoidable. Thus, the ability to create executable systems from a framework will also be dependent of introduced problems, like the usability of the instantiation mechanisms or the documentation of the framework. Like any software system, if a framework is ill documented, it is most likely that few people will be able to use it or maintain it over time. However, a new type of documentation must be present with frameworks: a "how-to extend" guide and/or instantiation tools.

Many documentation guidelines have been proposed for frameworks, such as documenting its hot spots using hot spots cards [8]. This approach tries to focus on the framework's flexible points, which possibly will be implemented and altered by people who are unaware of the framework design process. Another approach is to create cookbooks [5,8] that will discuss how the framework should be implemented and the steps required. These cookbooks contains many recipes, which describe informally how to solve specific problems when instantiating the framework. A third approach is to map the architectural solutions used throughout the design of the framework, but since this is clearly a documentation of the framework's architecture it does not account for all of the framework's facets.

As said before, framework engineering frequently uses design patterns. Even though these architecture fragments constitute a limited view of a framework, they are well known patterns that can help the framework instantiation process to be more comprehensible. For example, let's consider the previous example as illustrated in Figure 3. In this problem domain, one wishes to convert characters between formats using different algorithms (for example, ISO-8849-1 to MIME conversion algorithm). An effective solution would be the use of a flexibilization point (hot spot) that allows for the conversion algorithm to be altered in a "plug and play" fashion. For this purpose, one can use the strategy design pattern [3] that allows different conversion algorithms to be "plugged-in" into the framework without altering previously written code. This example is modeled in UML [1] in Figure 5. It is important to notice that Figure 5 can also serve as part of the documentation of the design of the framework.

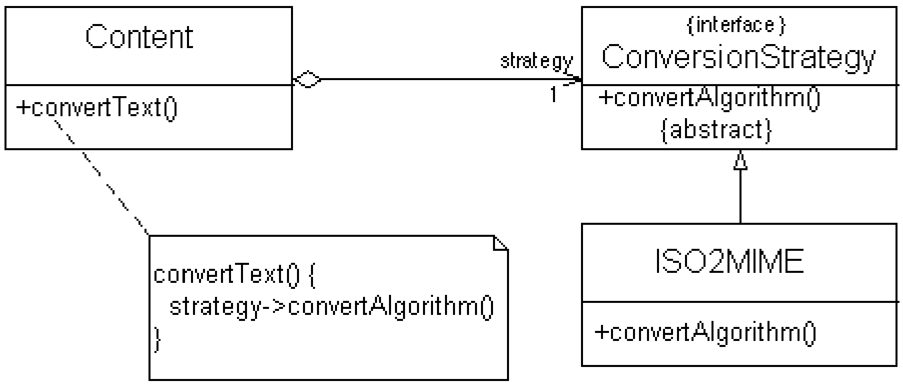


Figure 5. An example of an Architectural Solution

Even though there are many possible and feasible approaches to documenting a framework, there is no clear standard or acclaimed best. So, we believe the safer approach is to make the best of both worlds by using two or more of the approaches discussed above, illustrated by a solid and useful example. By using more than one kind or style of documentation that is well written, application developers are more likely to understand its use and purpose, making the learning curve less steep.

Domain Analysis Cost and Experience

As stated above, frameworks are created with the intention of generating applications within a specific domain. For this purpose, one of the development phases of frameworks is domain analysis. The domain analysis consists of the requirements analysis of a specific domain, i.e., a class of problems. Unlike the requirement phase of software systems, framework requirements cover whole domains, being more complex and expensive to track and maintain.

The central issue here is the size and complexity of the domain chosen. If the domain is too large, a long time is needed to assess information and gather the necessary resources. Furthermore, the development time and cost of the framework will be excessive and there is the need for some source of experience, such as individuals who are familiar with the domain, prototypes or similar software systems. Finding sources of experience that cover a large domain is very difficult or even impossible, in some cases.

On the other hand, if a too-narrow domain is chosen, its applicability is reduced. This means there will be the hot spots, and the generated applications will be too similar to justify the effort of building a framework. It is important to keep in mind what hot spots are needed and made necessary by the requirements and those that are unnecessary or superfluous.

Parallel Evolution of Instances and Frameworks

As time passes and a framework becomes more mature, it changes and evolves accordingly. This process can represent the alteration of its architecture, new requirements being met or unsupported and many other sources of change. Meanwhile, applications generated using the framework will also evolve and change.

Therefore, how it is possible to deal with both application and framework evolution? This problem is very similar to the use of third party function libraries: once they are changed or even discontinued, the applications that use them are left orphaned. There is no clear solution, and certainly most people will suffer the "not made here syndrome," refusing to use any framework not built by themselves. The only possible advice for this situation is to carefully study the framework to be used: a well-designed and implemented framework will not be as volatile as an ill-created one.

Flexibility vs. Complexity and Performance

As stated above, frameworks are built for flexibility and generality, trying to cover a whole domain instead of particular problems. This approach will produce an application generator that is more complex and with more points of flexibilization (hot spots) than traditional software systems. This flexibilization is achieved using inheritance and dynamic binding, common features in object-oriented languages. For this reason, a tradeoff between flexibility and performance is present, since dynamic binding introduces an overhead, and its use throughout the system will make it impossible to be a negligible performance hindrance.

This tradeoff makes it necessary for the framework designer to choose the hot spots carefully, neither exaggerating nor creating a far too generic framework. Even thought flexibility is important and useful, it should only be used where it is needed. Otherwise one could devise a "universal deterministic problem solver framework," illustrated in Figure 6. In this incredible framework, the hot spots are the problem_not_solved(), try_to_solve_problem() and return_solution() methods. It can be instantiated to find the solution of any problem in the deterministic problems domain, but is of course useless.

```
while ( problem_not_solved() != true )
{
        try_to_solve_problem();
}
return_solution();
```

Figure 6. The Universal Deterministic Problem Solver Framework

Along with performance issues, the abusive use of hot spots in a framework design will inevitably lead to complex software systems. By using hot spots to introduce generic solutions instead of specific ones, the complexity of the framework will grow. And since there are no requirements for the "extra" hot spots, the added complexity will add nothing in terms of functionality. It is important to notice that it is common for developers to introduce improper hot spots thinking they will make the framework "more powerful." However, this approach will lead to complexity and performance issues, as shown above, and sometimes the extra-functionality might be an inconvenient addition.

Problems with Debugging Framework Instances

Applications generated by a framework have an intertwining of application specific code and frozen spots code, being part of a same executable software system. This way, any trace of an application code will probably lead to the framework code. Using single-step methods for debugging will not work because of this blend, and frozen spot call must be separated from hot spot calls.

Automatic distinction between frozen spots code and hot spots code is impossible to any debugger. One possible solution is the use of pre- and post-conditions in every method of the frozen spots code, serving as executable assertions ensuring that these calls are valid. This way, the assertions will alert for any corrupted input or feedback given to the frozen spots code, and tracing will be much easier. However, this solution will not deal with the complexity introduced by the mix of hot spot and frozen spot code.

Conclusions

The framework approach definitely must be considered when one must meet requirements that are prone to fast change, as occurs, for instance, in the e-commerce and Web education domains. Frameworks can also be used so that incremental development is achieved by creating simple hot spot codes at first and then replacing them with incremental versions. A good reference for reading more about frameworks is [2], which makes a good assessment of the framework methodology and latest advances.

Since frameworks are a recent research and development topic, there were many problems that are still open in this technology. One such problem is framework documentation. When this article was being written, there were no official or de facto standards for documenting frameworks. This lack of common ground creates a gap between frameworks developers and its extenders and users. We can say that framework engineering is still an emerging area within software engineering. Another topic that is still incipient is framework economics, which estimates the cost of building frameworks vs. applications, and the return on investment. There currently are few industrial frameworks, i.e., frameworks being used in commercial and industrial environments even though the application of framework methodology in this context is promising and assessment of these efforts is still in its preliminary stages [6].

Finally, we believe that the issues exposed here should be looked at carefully by newcomers to the framework development scenario. In a nutshell, consider carefully what you need and what you are doing, and be aware that frameworks, as everything else, have their pros and cons and strong and weak points and they are not a solution to all problems.

References

1

Booch, G., Jacobson, I., Rumbaugh, J., and Rumbaugh, J. *The Unified Modeling Language User Guide* Addison-Wesley Pub Co, 1998.

2

Fayad, M. E., Schmidt, D. C., and Johnson, R. E. *Building Application Frameworks* Addison-Wesley Pub Co, 1st edition, 1999.

3

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns : Elements of Reusable Object-Oriented Software* Addison-Wesley Pub Co, 1st edition, January 1995.

4

Mattsson, M., Bosch, J., and Fayad, M.E. *Framework Integration Problems, Causes, Solutions* Communication of the ACM October 1999/Vol.42, No.10

5

Mattsson, M. *Object-Oriented Frameworks: A Survey of Methodological Issues* Technical Report 96-167, Dept. of Software Eng. and Computer Science, University of Karlskrona/Ronneby

6

Mattsson, M. and Bosch, J. *Observations on the Evolution of an Industrial OO Framework* Proceedings of ICSM'99, International Conference on Software Maintenance, Oxford, UK, 1999

7

--, *MIME (Multipurpose Internet Mail Extensions) Part Three: Message Header Extensions for Non-ASCII Text* Request for Comments (RFC) 2047, url: http://www.rfc-editor.org/rfc/rfc2047.txt

8

Pree, W. *Design Patterns for Object-Oriented Software Development* Addison-Wesley Pub Co, March 1995

9

Ripper, P., Fontoura, M. F., Neto, A. M., and Lucena, C. J. *V-Market: A Framework for e-Commerce Agent Systems* World Wide Web, Baltzer Science Publishers, 3(1), 2000.

Biography

Marcus Eduardo Markiewicz is a M.Sc. Computer Science graduate student at PUC-Rio, Brazil. He is a full time researcher at the TecComm e-Commerce group of the Software Engineering Laboratory (LES) at PUC-Rio.

Carlos J.P. Lucena is a Full-Professor in the Department of Computer Science at PUC-Rio since 1982. He received the B.Sc degree from the Pontifical Catholic University of Rio de Janeiro (PUC-Rio), Brazil, in 1965, the M.Math. degree in computer science from the University of Waterloo, Canada, in 1969, and the Ph.D. degree in computer science from the University of California at Los Angeles in 1974. He is also a member of the Editorial Board of International Journal on Formal Aspects of Computing (New York: Springer-Verlag).