Object Oriented Framework Development

by *Marcus Eduardo Markiewicz* and *Carlos J.P. Lucena*

Introduction

Object oriented frameworks (hereafter simply 'frameworks') are a cornerstone of modern software engineering. Framework development is rapidly gaining acceptance due to its ability to promote reuse of design and source code. **Frameworks** are application generators that are directly related to a specific domain, i.e., a family of related problems.

As an example, consider building a Graphical User Interface (GUI) tool kit. We might choose to design and implement a single tool kit. On the other hand, if we design the tool kit as a framework, our single design will enable us to generate a collection of tool kits for a variety of GUI applications. Frameworks must generate applications for an entire domain. Consequently, there must be points of flexibility that can be customized to suit the application. For example, one point of extensibility might be the algorithm used to draw graphical elements.

The points of flexibility of a framework are called **hot spots**. Hot spots are abstract classes or methods that must be implemented. Frameworks are not executable. To generate an executable, one must instantiate the framework by implementing application specific code for each hot spot. Once the hot spots are instantiated, the framework will use these classes using callback. In callback, the service user code declares that it wants to be called on the occurrence of a determined event. Then, the service provider code performs callback on the service user code when the event occurs. For this reason, the framework approach is sometimes characterized as "old code calls new code."

Some features of the framework are not mutable and cannot be easily altered. These points of immutability constitute the kernel of a framework, also called the **frozen spots** of the framework. Frozen spots, unlike hot spots, are pieces of code already implemented within the framework that call one or more hot spots provided by the implementer. The kernel will be the constant and always present part of each instance of the framework.

Think of a framework as an engine. An engine requires power. Unlike a traditional engine, a framework engine has many power inlets. Each of these power inlets is a hot spot of the framework. Each hot spot must be powered (implemented) for the engine (framework) to work. The power generators are the application specific code that must be plugged in to the hot spots. The added application code will be used by the kernel code of the framework. The engine will not run until all plugs are connected. This metaphor is illustrated in Figure 1.

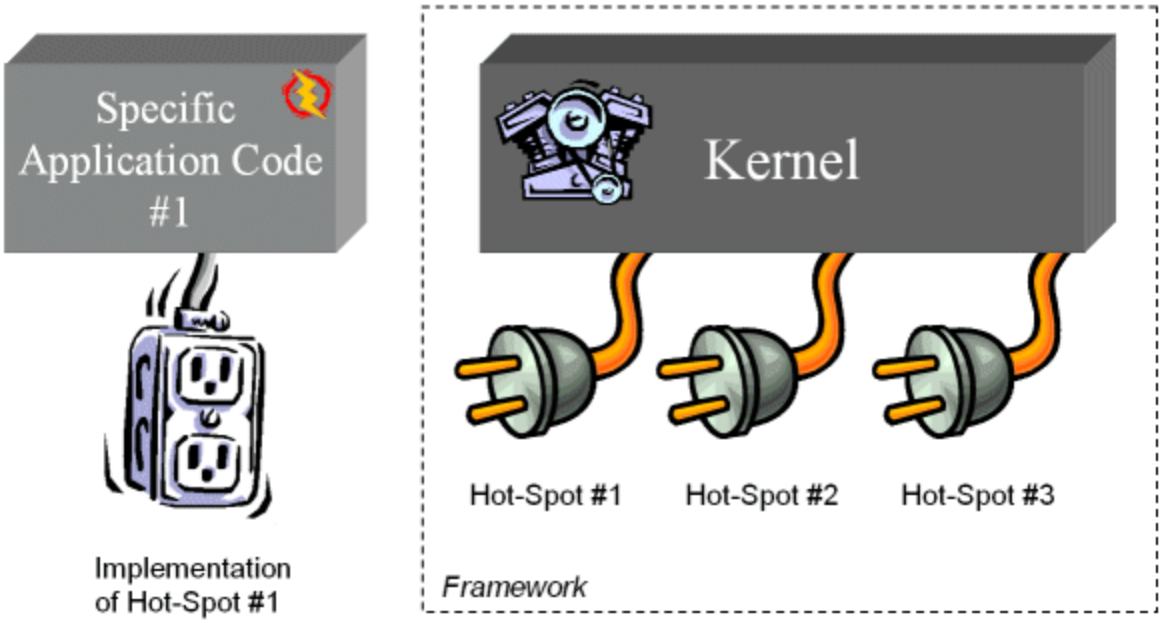

Figure 1. Frameworks

The code and design reuse capability of object oriented frameworks enables higher productivity and shorter time-to-market of application development when compared with traditional software systems development [4]. The flexible architecture of frameworks, allows kernel reuse. Framework development has been successful in many domains. Some examples include the Microsoft Foundation Classes (MFC) framework, the Object Management Group's (OMG) CORBA [3] and Microsoft's DCOM and COM+ [11]. To clarify frameworks, let us look briefly at a testing framework, the JUnit.

Understanding Frameworks: The JUnit Testing Framework

JUnit [7] is a simple, well designed, testing framework for JAVA. The architecture of JUnit is shown as a UML class diagram in Figure 2[1]. Each rectangular box represents a class. The upper section holds its name and the lower holds its methods. Each relationship between these classes is represented by the bars that connect them. The arrow bar connecting the TestCase and Test classes indicates an "is-a" relationship, following the direction of the arrow; thus, the TestCase class inherits from the Test class. The connecting bar between the TestResult and TestCase classes indicates an "association," i.e. one class calls the other's methods. The diamond head bar indicates the "has-a" relationship, where the class nearer to the diamond head is the one that has the other; in this example, the TestSuite class has an instance of one of the derived classes of the Test class (e.g., the TestSuite and TestCase classes).

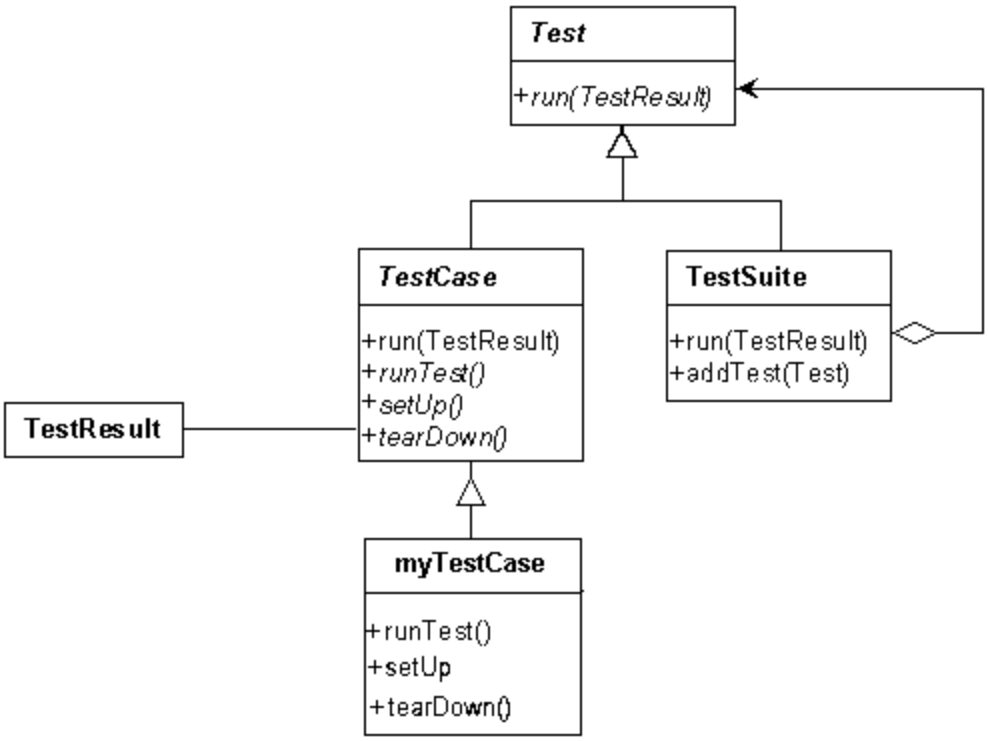

Figure 2. The JUnit Framework Classes

Provided with the framework is a "cookbook," which describes the instantiation process step by step. For example, this framework can be instantiated by implementing the TestCase class. The run() method, inherited from the Test class will run the test case by executing setUp(), runTest(), and tearDown() in sequence. Let us explain the role of these methods carefully.

The runTest() method holds API function calls, method calls, and other code being tested. The JUnit framework also provides infrastructure for running two or more tests on similar or identical sets of objects. This is called test fixture, and it is possible using the setUp() and tearDown() methods. The setUp() method initializes the object's variables. Correspondingly, the tearDown() method clears the resources allocated by setUp(). Every time the run() method is called, the setUp() method sets the test scenario, the test is executed, then tearDown() destroys the scenario created. The application generator instantiates the framework via the myTestCase class which subsequently instantiates the methods discussed above.

Test cases may also be grouped into test suites. A particular test suite might contain several test cases or test suites. When the run() method is invoked for a test suite, all of its tests are recursively executed. The results of the tests are collected into the TestResult classes, which contain details of test failure. Grouping is implemented using the composite design pattern [5].

The hot spots of this framework are the testing code, the testing scenario construction, and the destruction code. All these are implemented by the methods described above. Even further, this framework has a clear and well defined interface, the abstract class Test, which is extended by TestCase and TestSuite.

Framework Issues

The three major stages of framework development are domain analysis, framework design, and framework instantiation.

Domain analysis attempts to discover the domain's requirements and possible future requirements. In order to capture these requirements, previously published experiences, existing similar software systems, personal experiences, and standards are taken into account. During domain analysis, the hot spots and frozen spots are partially uncovered.

The framework design phase defines the framework's abstractions. Hot spots and frozen spots are modeled (perhaps with the Unified Modeling Language [1] diagrams), and the extensibility and flexibility proposed in the domain analysis is outlined. As mentioned above, design patterns are used in this phase.

Finally, in the instantiation phase, the framework hot spots are implemented, generating a software system. It is important to note that each of these applications will have the framework's frozen spots in common. The framework development process phases are compared to the traditional object oriented design phases in Figure 3. In this figure, we named the development phases as described in [6].

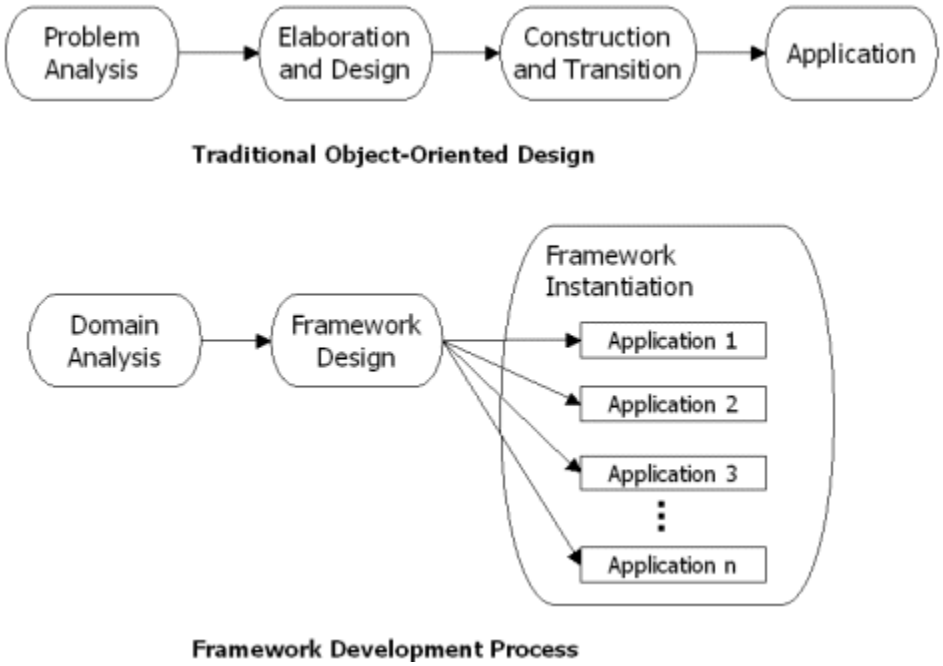

Figure 3. Development Process of Frameworks

As shown in Figure 3, traditional object oriented development differs from framework development. In object oriented development, the problem analysis phase, also called inception, only studies the requirements of a single problem. On the other hand, framework development captures the requirements for an entire domain. Furthermore, the final result of traditional object oriented development is an executable application that is executable, whereas many applications result from the instantiation phase of framework development. The instantiation phase comprises the construction and transition phases of the traditional development. Thus, separate construction and transition phases

are present in each of the framework's instances. For each of the framework's instances there is an implementation effort introduced by these phases.

Even though framework development promises to be very efficient, there are several issues that should be discussed. In the following sections we will review seven issues that one must consider 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 development 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 development will evolve and prove itself the rule of thumb for many domains, but surely not for all.

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, the TecComm/LES. One of these frameworks, called V-Market, is presented in [14], and we encourage the reader to examine this e-commerce framework.

Application Generator Development vs. Application Development

As stated before, frameworks generate applications by customization. They are not applications themselves; they are more complex constructs. It is important to keep in mind that the development of a framework will be at least as expensive as single application development, and generally much more expensive. One must carefully analyze the need for the flexibility of a framework when assessing requirements that must be met for a client or future user, otherwise a single use behemoth will be created unnecessarily.

On the other hand, the effort of building application generators can pay itself off through the repeated generation of applications within the proposed domain. When choosing a framework model one must ask: "Will I be creating applications of this same domain more than once?" If the answer is yes, then 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 costs versus the benefits of choosing to develop a framework instead of a custom-made software system*.

Consider the design of a system to transform text files from one encoding, such as the ISO-8859-1 Latin character set, to an alternate encoding, such as the e-mail text encoding format Multi-purpose Internet Mail Extension (MIME) [12]. Should this system be built as a framework? The answer is yes if there are plans to convert ISO-8859-1 into other formats (e.g. UUENCODE), or even to convert 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 would also be a hot spot. For example, in Figure 4 a text written using ISO-8859-1 characters like "ç" and "í", which do not exist in ASCII, are encoded in MIME and UUENCODE.

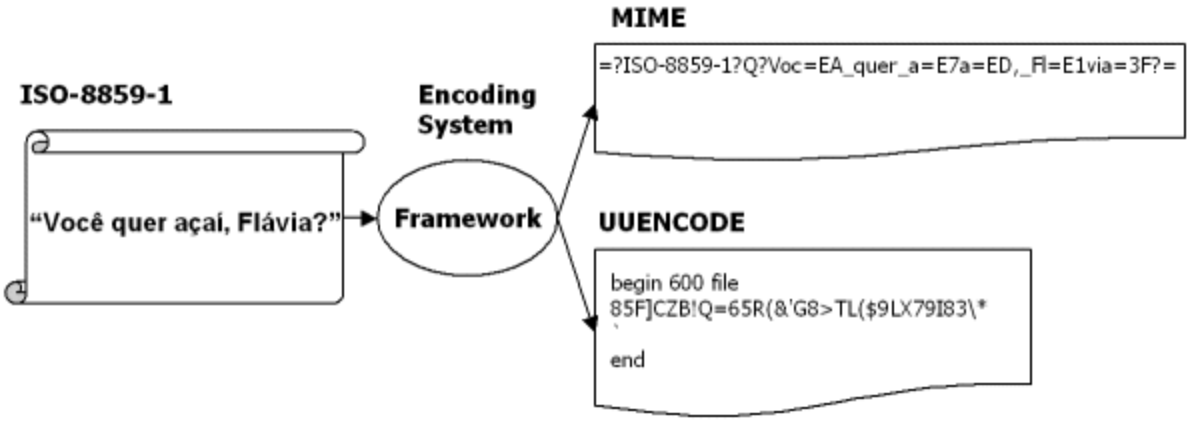

Figure 4. A sample encoding system at work

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. You will find yourself in gray areas more often than you would like. 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 to integrate frameworks to fulfill application requirements. However, Michael Mattson argues that there are at least six common problems that application and framework developers encounter when integrating two or more frameworks [9]. 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 [9], where many solutions are proposed 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. Frameworks are often abandoned or aborted because they cannot be easily integrated with other frameworks. Framework integration is not an easy task, and *composition must be considered seriously during development*.

One way to consider composition when developing frameworks is to maintain a set of APIs that encapsulate the services that the framework provides. When composing with a framework, the application only needs to know some functions and parameters that should be called, ignoring the inner workings of the framework. Another option is to create a mediation layer, to convert framework's requests. However, if many composing frameworks are involved, this approach can prove expensive if one does not choose a unified mediation layer between *all* the composed elements. In this case, the mediation layer will act as a "glue" between the composed elements.

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 [4]. In **white box frameworks**, also called architecture-driven frameworks, instantiation is

only possible through the 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.

Black box frameworks produce instances using configuration scripts. Following configuration, an instantiation automation tool creates the classes and source code. For example, it is possible to use a graphical wizard that guides the user step by step through a framework instantiation process. The black box approach does not require framework users to learn details of the framework internals. Consequently, these frameworks are also called data-driven frameworks [4] and are generally easier to use. Figure 5 conceptually illustrates white box and black box frameworks. Frameworks that contain both white box and black box characteristics are called **gray box frameworks**.

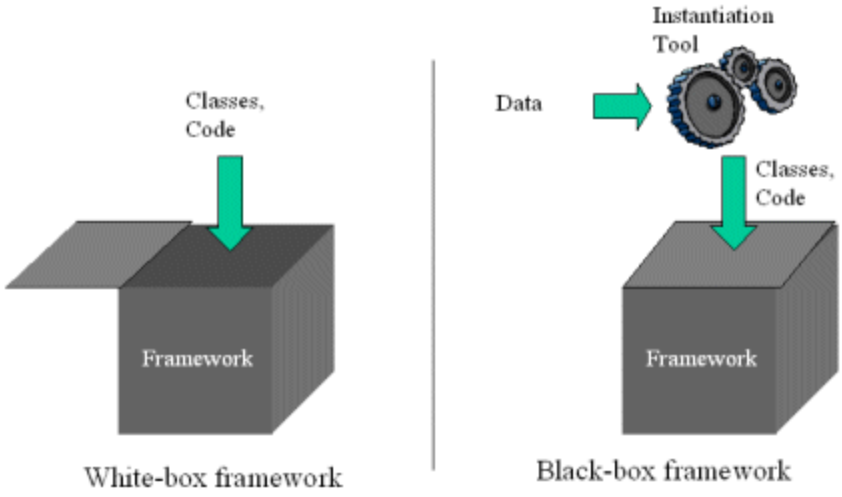

Figure 5. White box frameworks vs. black box frameworks

White box, black box and mixed (gray box) approaches have a steep learning curve. Thus, the ability to create executable systems from a framework will depend on the usability of the instantiation mechanisms and the documentation of the framework. If a framework is ill documented, few people will use 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. The **hot spots cards** [13] approach focuses on the framework's flexible points. Another approach is to create **cookbooks** that discuss how the framework should be implemented and the steps required [8,13]. These cookbooks contain many "recipes," which describe informally how to solve specific problems while instantiating the framework. A third approach is to map the architectural solutions used throughout the design of the framework. However, documentation of the framework's architecture does not account for all of the framework's facets.

As said before, framework development 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 comprehensibility of the framework instantiation process. Consider again the example shown in Figure 4. In this problem domain, one wishes to convert characters between formats using different algorithms (e.g., the ISO-8849-1 to MIME conversion algorithm). An effective solution might create

an extensibility point (hot spot) that allows for the conversion algorithm to be altered in a "plug and play" fashion. The *strategy* design pattern is suited to this application and allows different conversion algorithms to be "plugged-in" to the framework without altering previously written code [5]. This example is modeled in UML in Figure 6. This diagram uses the same notation used in Figure 2, with a few more elements. The folded-corner box is a comment, and the dashed line indicates what element of the diagram the comment refers to. It is important to notice that Figure 6 can also serve as part of the documentation of the design of the framework.

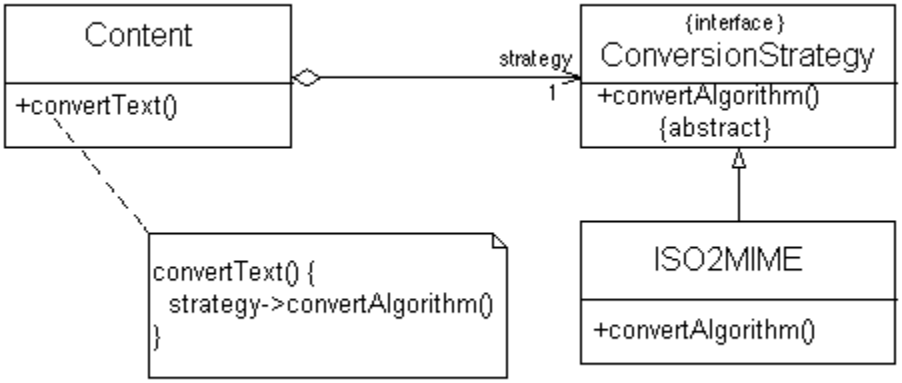

Figure 6. An example of an Architectural Solution

Even though there are many possible and feasible approaches to documenting a framework, there is no clear standard. The safest approach is to use two or more of the approaches discussed above.

Domain Analysis Cost and Experience

As stated above, frameworks are created to generate applications for a specific domain. For this purpose, one of the development phases of frameworks is domain analysis. Unlike the requirement phase of software systems, domain analysis covers an entire class of problems.

Domain analysis attempts to characterize the size and complexity of a chosen domain. If the domain is too large, it is time consuming to gather and assess information and resources. Furthermore, the development time and cost of the framework will be excessive. In addition, individuals familiar with the domain, prototypes or similar software systems will be necessary. Finding sources of experience that cover a large domain is difficult.

On the other hand, if one chooses a domain that is too narrow, the framework's applicability is reduced 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.

How is it possible to deal with both application and framework evolution? Applications based upon frameworks might be orphaned if a framework is changed or discontinued. 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 a poorly created one.

By "well designed" we mean that frameworks should have solid documentation and meet the domain requirements. A framework that maintains the object oriented concept of encapsulation and has a well defined public interface is likely to remain forward compatible at the interface level across upgrades and revisions. The evaluation of a framework's design and/or implementation is not always straightforward; one must consider the experience of the designer, the complexity of the instantiation process, the requirements met and unmet, and the update road map. An update road map contains the plans for updating the framework, whether it will represent a complete redesign at each new version or a backward compatible smooth update.

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 produces an application generator that is more complex and more extensible (hot spots) than traditional software systems. Extensibility 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 a 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 though flexibility is important and useful, it should only be present where it is needed. Otherwise one could devise a "universal deterministic problem solver framework," illustrated in Figure 7. 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 7. 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. Using hot spots to introduce generic solutions adds to the

complexity of the framework. As 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

Frameworks generate applications that have an intertwining of application specific code and frozen spot code. Consequently, a debug trace of the application code often leads to framework code. Using single-step methods for debugging will not work because of this blend. Frozen spot calls must be separated from hot spot calls.

Automatic distinction between frozen spot code and hot spot code is impossible for any debugger. One possible solution is the use of pre- and post-conditions in every method of the frozen spot 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 spot 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. By entangling hot and frozen spots, the framework will be harder to instantiate, as application developers will have to change and introduce code carefully, in order to avoid corrupting frozen spot code that should not be altered. The cost of maintaining confusing code such as this is unavoidably high.

Conclusions

The framework approach should be considered when product requirements mutate rapidly. Frameworks can also be used for incremental development by implementing simple hot spot code at first and subsequently upgrading it. A good reference of frameworks is [4], which makes a good assessment of the framework methodology and latest advances.

Frameworks are a recent research and development topic. Consequently, there are still many open problems. One such problem is framework documentation. At the time of publication of this article, there are no official or de facto standards for documenting frameworks. The lack of common ground creates a gap between framework developers, extenders, and users. Other open questions are in the area of framework economics, which estimates the cost of building frameworks vs. applications, and the return on investment. There currently are few industrial frameworks (i.e., those being used in commercial and industrial environments). Even though the application of framework methodology in this context is promising, assessment of these efforts is still in its preliminary stages [10].

Finally, we believe that newcomers to the framework development scenario should look at the issues exposed here carefully. Consider what you need, what you are doing, and be aware that frameworks have their pros and cons. They are not a solution to all problems. Furthermore, if you are considering using an existing framework, study its documentation and verify if there is a "how to instantiate" explanation of good quality, if it exists. It is also important to observe the applications that were generated and the amount of effort spent in this process.

References

1

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

2

Brooks, Jr., F. P. *No Silver Bullet--Essence and Accidents of Software Engineering*. IEEE Computer 20(4), April, 10-19, 1987.

3

--, *CORBA Website*. url: http://www.corba.org/.

4

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

5

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.

6

Jacobson, I., Booch, G., Rumbaugh, J. *The Unified Software Developement Process*. Addison-Wesley, 1999.

7

--, *JUnit, Testing Resources for Extreme Programming,* url: http://www.junit.org.

8

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.

9

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

10

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.

11

--, *Microsoft COM Technologies*. url: http://www.microsoft.com/com/.

12

--, *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.

13

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

14

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 (mem@rdc.puc-rio.br) 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 (lucena@csg.uwaterloo.ca) 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).

Want more articles about Software Engineering? Go to the index, to the next one, or to the previous one.

Location: www.acm.org/crossroads/xrds7-4/frameworks.html