

Application Frameworks

Theory and Practice

CIS518

Advanced Software Engineering

Ryan Somma
StudentID: 9989060874
Ryeguy123@yahoo.com

Table of Contents

Introduction.....	3
Common Framework Design Patterns	6
Inversion of Control (IoC)	7
Front Controller.....	7
Model View Controller (MVC)	8
Object-Relational Mapper (ORM)	9
Application Framework Strength and Weaknesses.....	11
Extensibility	11
Integrateability	13
Metrics	14
Maturity	14
Learning Curve.....	15
Documentation and Support	16
Framework “Make or Buy” Analysis Considerations.....	18
Advantages of Adopting an Existing Framework.....	18
Advantages of Developing a Framework In-House	19
Organization-Dependent Considerations.....	19
Conclusions.....	21
References.....	22

Introduction

The evolution of programming languages over the last 70 years has shown a clear trend towards reducing complexity and improving efficiency of software development. From first generation machine languages that used binary strings representing programming instruction being replaced with assembly languages that provided mnemonics representing CPU instructions, to third, fourth, and fifth generation programming languages (3GL, 4GL, 5GL) providing programming statements that implemented increasing ratios of machine instructions to each programming statement, to object-oriented languages which combined programming statements and variables into integrated objects, developers have striven to make programming easier to understand while simultaneously architecting development environments where programmers may write fewer instructions to accomplish tasks (Burd, 2006).

Frameworks represent a natural extension of this evolution, sequestering even more complexity away from the programmer and providing a head start in their development efforts:

A frequently used definition is “a framework is a reusable design of all or part of a system that is represented by a set of abstract classes and the way their instances interact.” Another common definition is “a framework is the skeleton of an application that can be customized by an application developer.” These are not conflicting definitions; the first describes the structure of a framework while the second describes its purpose (Fayad, Schmidt, and Johnson, 1999).

Just as the third through fifth GL’s took chunks of machine instructions and combined them into single programming statements that were easier to understand, frameworks take chunks of scripting and object-oriented programming language functionality and wrap them up into reusable tools for development.

Frameworks can encompass a whole application, providing an entire suite of tools for information systems development, or just a sub-component, providing a specific solution to a related set of problems, environment, or layer-specific programming needs (Fayad, Schmidt, and Johnson, 1999). For example, web templating frameworks, or “template engines,” provide reusable web-design functions and the capability to standardize user interface content through interface-layer objects (Smarty, 2008). Cascading Style-Sheet (CSS) frameworks also provide web-design functionality, but do so strictly through client-side scripts rendering content in a web browser (Stenhouse, 2005). Semantic web frameworks tie together a variety of “bleeding-edge” technologies such as RDF, OWL, and SPARQL so that developers can focus on business functionality rather than the syntactic-specifics of working with these resources (Open Channel, 2008). Similarly, JavaScript frameworks help developers easily implement newer technologies, like AJAX and dynamic HTML, while hiding the complexity of having these solutions work properly in all web browsers (Reindel, 2008).

In many of the above examples, a well-designed framework should insulate the complexity of software development from the programmer, allowing them to focus on programming to the business requirements. Additionally, frameworks encourage best practices in software development, such as the “Don’t Repeat Yourself” (DRY) principle, philosophically stated, “Every piece of knowledge must have a single, unambiguous, authoritative representation within a system (Hunt and Thomas, 1999),” where reusing framework solutions means that modifying the code in the single place it occurs modifies every instance of its implementation throughout the application. Such reuse standardizes coding and development practices.

Combining the traits of reduced complexity with frameworks taking “the tedium out of writing all the program code for an application from scratch (PCMag, 2009),” a properly realized framework can provide the benefit of improving productivity and accelerating development. Maximizing this advantage is the focus of Rapid Application Development frameworks (RADF) (Spatial, 2009)¹, some of which implement RAD software development life cycle (SDLC) best practices and others even go so far as to automate the generation of software code for developers.

¹ It is arguable that the implementation of any properly designed framework is a RAD strategy; therefore, RADF may be a redundancy.

Common Framework Design Patterns

Architect Dr. Christopher Alexander is considered the “father of the Pattern Language movement in computer science,” thanks to his 1977 book on architecture, *Pattern Language: Towns, Buildings, Construction* (PatternLanguage, 2001), which emphasizes the need for implementing standard solutions to common architectural problems. Alexander defined a format for describing patterns that included a statement of the problem, its context, solution pattern, diagram of the pattern, and how the pattern relates to others (Schummer and Lukosch, 2007). Alexander’s “pattern language,” also known as the “Alexandrian pattern form,” was used by the “Gang of Four” software developers, Gamma, Helm, Johnson, and Vlissides, to structure their classic book on software development, *Design Patterns: Elements of Reusable Object-Oriented Software*, which provides 23 object-oriented solutions to common software architectural problems. In this *Design Patterns*, the authors define three major classifications for software design patterns: creational patterns, which “concern the process of object creation,” structural patterns, which “deal with the composition of classes or objects,” and behavioral patterns, which “characterize the ways in which classes or objects interact and distribute responsibility (Gamma, *et al.* 1995).”

One of the questions raised by Fayad, Schmidt, and Johnson’s *Building Application Frameworks* is the conceptual issue of whether frameworks are actually a large-scale pattern or simply another kind of programming component? Similar to developers working in the same framework, “Developers who share a set of patterns have a common vocabulary for describing their designs;” however, “frameworks are more than just ideas—they are also code;” therefore, because of this “different level of abstraction,”

design patterns are distinguished as “the architectural elements of frameworks (Fayad, Schmidt, and Johnson, 1999).”

A framework constitutes a collection of interacting design patterns, and design patterns are based on common solutions implemented in frameworks. While many design patterns apply to solving problems that may occur in a wide variety of programming situations, there are several design patterns for solving challenges faced specifically in application framework implementations:

Inversion of Control (IoC)

While it is true that “Object-oriented application frameworks... are structured as a class library (PCMag, 2009),” conceptualizing frameworks in this manner gives the impression that its classes are meant to be accessed by the developer as if they were checking out books from a library; however, “The major difference between an object-oriented framework and a class library is that the framework calls the application code. Normally the application code calls the class library (Mattsson, 1996).” This is known as the “Inversion of Control” pattern, where programmers write application components that the framework will reference. The framework is in control, and the programmer is simply supplementing its existing functionality with the specific business logic he or she needs it to perform. This “gives frameworks the power to serve as extensible skeletons (Johnson and Foote, 1988).”

Front Controller

When the framework serves as the center of control, accessing components supplied by a programmer, it is useful to have a central point in the application for handling interface layer requests. The Front Controller pattern “defines a single

component that is responsible for processing application requests. A front controller centralizes functions such as view selection, security, and templating, and applies them consistently across all pages or views (Sun Microsystems, 2002).” The front controller handles functionality common throughout the application, or references objects that provide the common functionality, standardizing implementation and discouraging programmers from writing their own nonstandard classes to handle functionality already covered in the framework.

Stephen R. Jones refers to a pattern that shares characteristics of both IoC and Front Controller, which he titles “Application Is a Class,” in 1999, and, although this pattern is so simple that it is rarely mentioned as a pattern at all, failure to include it in a framework can be disastrous, as Jones discovered early in his career when he worked on a framework that allowed for multiple controls:

In these situations, each development team invented its own idea of what shape their application would take and, as a result, there was no consistency among the various approaches. This made introducing new services common to all applications next to impossible and required new developers to learn the individual semantics of each group’s approach (Fayad, Schmidt, and Johnson, 1999).

Model View Controller (MVC)

On one end of the framework is the user accessing the application through a user interface (UI). The UI renders graphically the actions the user can perform in the system and the data from the persistence structures, such as the database or XML file, organized according to the business purposes or domain. Objects that encapsulate how the user should see this information, how the business should logically organize it, and what rules need to be in place for retrieving and updating it, would require dramatic changes should any one of these aspects change; therefore, a solution that loosely couples these three aspects of what the user is seeing into three separate objects, is optimum.

As it was defined originally for the Smalltalk-80 system in 1988, the MVC pattern separates the UI into a set of “View” objects, which are strictly focused on presenting information to the user and accepting request inputs. These inputs are then passed to the “Controller” objects, which encapsulate the logic to handle both the user inputs and responses from the domain objects. The “Model” objects encapsulate the business logic, a “domain-specific simulation or implementation of the application’s central structure (Krasner and Pope, 1988).”

Depending on the development environment in which the framework is operating, what constitutes the objects interacting within the pattern can change quite dramatically. This is because the MVC is a metaphor for how to separate logic and classify functionality. For instance, in the example of a simple web page, the model would be the HTML, the page source, the view the CSS, defining how the model should be displayed, and the web browser serving as the controller, determining how to handle the input of a user clicking on a hyperlink. In a web application, where data is served up to the user dynamically, the model consists of the “classes which are used to store and manipulate state, typically in a database of some kind (Atwood, 2008).”

Object-Relational Mapper (ORM)

At the other end of the framework is the relational database or other persistent data source, like XML, where the application is performing create, read, update, and delete (CRUD) operations. Also, in the interest of application performance, the more recent frameworks also implement caching strategies, where more static data may be stored for quicker access, bypassing the database. Because databases store data in normalized form, using relational tables, while modern programming languages work

with data encapsulated in objects, a framework operating in such an environment should provide functionality for translating data between the two organizational methods. The ORM pattern provides such functionality, which would normally reside in the Model portion of the MVC, but can also be a framework unto itself (Ambler, 2009).

Application Framework Strength and Weaknesses

Adopting a framework is adopting a software development methodology.

Frameworks don't just provide reusable classes, but reusable analysis as well. The objects in the framework provide, "a vocabulary for talking about a problem (Fayad, Schmidt, and Johnson, 1999)." A survey of multiple sources finds a variety of core considerations in evaluating frameworks, including extensibility, integrateability, metrics/benchmarks, maturity, learning curve, documentation and support ((Reindel, 2007), (Fayad, Schmidt, and Johnson, 1999), and (Cliff, 2006)). These characteristics of the framework are characteristics of the software methodology as well:

Extensibility

A framework must provide developers the flexibility to handle a wide range of issues that may arise within the framework's domain. For an applications framework, this includes being flexible enough to allow programmers to develop solutions to problems so unique and wide in possibility that no framework could possibly provide the necessary functionality built-in to account for them all. For this reason, framework extensibility is of crucial importance, not only must the framework be extensible to meet a wide variety of situations, but framework's strategy for extensibility should be taken into consideration as well.

Frameworks may be "classified by the techniques used to extend them, which range along a continuum from *whitebox frameworks* to *graybox frameworks* to *blackbox frameworks* (Fayad, Schmidt, and Johnson, 1999)." In terms of software reuse, black box reuse refers being able to use components "as is," without having to know anything about

their internal workings, only the inputs and outputs expected from them. In this context, white-box reuse refers to modifying the software or component to fit specific needs, which requires some understanding of how it works internally (Pfleeger and Atlee, 2006). Blackbox reuse allows for easier implementation, but whitebox allows for greater customization.

In OOP, black-box versus white-box reuse is classically framed in terms of “Composition vs. Inheritance,” where objects are defined in either “has-a” or “is-a” relationships (Brown, 1998). In this context, black-box reuse refers to composing objects “to achieve more complex functionality,” which “requires that the objects have well-defined interfaces since the internals of the objects are unknown,” while white-box reuse involves class inheritance, allowing “a subclass’ implementation to be defined in terms of the parent class’ implementation (Rajlick, 1998).” In the context of object-oriented frameworks:

Whitebox frameworks rely heavily on OO language features like inheritance and dynamic binding in order to achieve extensibility. Existing functionality is reused and extended by (1) inheriting from framework base classes and (2) overriding predefined hook methods using patterns like the Template Method. Blackbox frameworks support extensibility by defining interfaces for components that can be plugged into the framework via object composition. Existing functionality is reused by (1) defining components that conform to a particular interface and (2) integrating these components into the framework using patterns like Strategy and Functor... a good graybox framework has enough flexibility and extendibility, and also has the ability to hide unnecessary information from the application developers (Fayad, Schmidt, and Johnson, 1999).

Thus, in a blackbox framework, developers create new classes, which they provide to the framework through thoroughly defined interfaces, and in a whitebox framework, developers extend existing classes, overriding the appropriate methods with functionality specific to the instance of the object.

William Wake talks about black and white box solutions in the context of Java applications frameworks, observing that they tend to evolve in their reuse strategies with maturity:

Frameworks tend to change over their lifetime. When a framework is new, it tends to be white-box: you change things by subclassing, and you have to peek at source code to get things done. As it evolves, it becomes more black-box, and you find yourself composing structures of smaller objects. Johnson and Roberts point out that frameworks can evolve beyond black-box, perhaps becoming visual programming environments, where programs can be created by interconnecting components selected from a palette (Wake, 2009).

Integrateability

“Application development will be increasingly based on integration of multiple frameworks (GUIs, communication systems, databases, and so on), together with class libraries, legacy systems, and existing components (Fayad, Schmidt, and Johnson, 1999).” The framework does not operate in a bubble at the organization level, but will operate as one system among a variety of other information systems. As such, it must have the capability or extensibility to integrate with other systems both within and external to the organization. Does the framework “play well with others?”

The ability to have different applications running on different platforms be able to exchange and synchronize data provides “one version of the truth (Schmidt and Lyle, 2005).” The purpose of Service-Oriented Architecture (SOA) is to unify disparate systems operating within different business domains so that they may exchange information or be brought under a single system without losing their individual operations (Channabasavaiah, Holley, and Tuggle, 2003). XML and Web 2.0 innovations provide some strategies for implementing SOA, but there is no specific technology required for integration; therefore, frameworks must provide sufficient decoupling of

presentation and control layers so that the application may interface with either a human user, or another application requesting data.

Metrics

Another consideration when choosing or implementing a framework is the way the architecture of that framework will skew OO size and design measures. For example, applications using the Smalltalk/V system's *ViewManager* class requires subclassing a class that is already three levels deep, so that the classes the developer implements extending it will be subject to the framework's existing nesting levels (Lorenz and Kidd, 1994). The same holds true for other measures such as the application's specialization index (SI), as this is dependent on nesting level and total class methods (Pfleeger and Atlee, 2006).

“Frameworks enhance extensibility by employing additional levels of indirection... However, the resulting generality and flexibility often reduce efficiency (Fayad, Schmidt, and Johnson, 1999).” All application frameworks present a tradeoff between functionality and performance. Benchmarks of a selection of some of the most popular PHP web application frameworks found basic HTML serving up 1,327.9 requests per second, with baseline PHP serving 331.8 requests, and none of the three PHP frameworks serving more than 22 requests per second on average (Ekerete, 2008).

Maturity

Michael Mattsson and Jan Bosch at the University of Karlskrona in Sweden gathered a variety of metrics on Ericsson Software Technology's Mediation framework, comparing the depth of inheritance (DOI), number of ancestors (NOA), number of children (NOC), and numerous others between the framework's four versions:

Framework generally evolve to maturity through a number of iterations due to the incorporation of new requirements and better domain understanding. Since changes to frameworks have a large impact due to the effects on the applications build based on the asset, it is important to assess the maturity of a framework [sic] (Mattson and Bosch, 1999).

They reference J. Bansiya as formulating four statements about frameworks as they

mature:

(1) framework maturity is achieved generally in the third to fifth version, (2) evolving frameworks change between 30% and 50% of the maximum number of changes between version, whereas this figure is below 10% for mature versions, (3) the average number of changes per class in mature frameworks is about 50% less than in evolving frameworks and (4) mature frameworks have an average depth of inheritance above 2.0 and an average width of inheritance below 0.8 (Mattson and Bosch, 1999).

Mattson and Bosch's own research found that changing framework requirements should be taken into consideration in considering at what version it will mature, as the framework they studied continued to have changes of about 20% between versions after reaching maturity. Although the researchers could not support Bansiya's findings, which attributed a much higher level of stability to mature frameworks, they did find that mature frameworks did reach an "absolute low figure" in changes, which indicated maturity (Mattson and Bosch, 1999).

Learning Curve

According to PC Magazine, "although the purpose of a framework is to eliminate a certain amount of programming drudgery, programmers must first learn the structure and peculiarities of the framework in order to use it (PCMag, 2009)." While it is true that adopting a framework involves a great deal time and money invested into learning how to use it, it must also be taken into consideration that, "the suitability of a framework for a particular application may not be apparent until the learning curve has flattened (Fayad, Schmidt, and Johnson, 1999)." This presents a chicken and egg dilemma, as a framework needs to be chosen for how well it can meet the demands placed by the requirements, but

it is nearly impossible to assess the framework's effectiveness at meeting these demands until the developers have made the immense investment of effort into learning the framework. Another way to state this is:

Many developers In order to choose a framework one needs to understand each framework's, conceptual and logical designs in addition to comparing their promised features. Anyone who can't or won't perform this level of analysis before choosing a framework is hoping to get lucky. The catch is that most developers, particularly developers new to OOAD, can't perform this level of analysis [sic] (Miller, 2007).

An interesting example of this was found in a comparison of the PHP frameworks CodeIgniter (CI) and Symfony by DevTrench. The reviewer found that, while Symfony provided far more features, there was a much greater level of effort to get it up and running initially. By contrast, CI was much easier to get up, running, and start developing (DevTrench, 2007). While many of the commenters for the article agreed with the author's assessment and preferred CI for its easy interface, one commenter related their experience spending three months coding in Symfony, and then being forced to use CI. "I am frustrated all the time," the comment read, "I have to do everything myself (dazz, 2009)." While this evidence is anecdotal, it does show that there are tradeoffs between framework sophistication and learning curve so that leaning to either side will always please some developers, yet frustrate others.

Documentation and Support

Finally, documentation is crucial to the success of a framework for use by the general public. In the CI versus Symfony comparison, both frameworks were credited for their extensive documentation. The Zend framework's documentation not only covers the details of developing in its environment, but also thoroughly covers the design features and methodologies at work within the framework (Zend, 2009).

While documentation provides a reference for developers using the framework, no documentation can be thorough enough to meet every possible idiosyncratic problem a programmer might encounter working with it. For this reason, an active community of developers is key to keeping the framework up to date, developing ways around new developmental obstacles and quickly responding to peers who are stumped on how to implement a solution within the framework's methodologies. "As frameworks invariably evolve, the applications that use them must evolve with them (Fayad, Schmidt, and Johnson, 1999)," and an active core of developers provides the user support to ease the pain of this evolution.

Framework “Make or Buy” Analysis Considerations

The word “Buy” in this section’s title is a bit misleading, as most frameworks are free and open-source, which is also why they are so successful. A more accurate title would be whether to develop the framework in-house or use an existing solution. In fact, building a framework from scratch is far more costly, making an in-house solution seem more like the “buy” side of the equation. There are advantages and disadvantages to developing a framework in-house or adopting an existing framework. The following considerations are based on Carl Karsten’s make or buy analysis in the context of Visual Fox Pro (Karsten, 2009); however, the conclusions are based on the research presented in this paper thus far:

Advantages of Adopting an Existing Framework

Existing frameworks, especially mature frameworks with significant mindshare, provide a collection of established practices that have been peer-reviewed and tested in every organization that has implemented it. There is no “reinventing the wheel” and having to program through issues that the framework community has already identified and worked through. Additionally, an established framework comes with that framework’s community of support, which will include forums, FAQs, and documentation. Rather than being limited to the organization’s resources, the adoption of a framework means adopting the community resources that fashioned it. This encourages an “egoless programming” environment, where responsibility is shared by everyone, and the focus is on solutions to problems rather than assigning blame (Pfleeger & Atlee, 2006).

Existing frameworks also provide a much more cost-effective solution, as developing a framework from scratch could take years, which equates to hundreds of thousands of dollars in programmer hours. If the existing framework is open-source and free to use, then the only costs are the amount of time it takes the organization's developers to learn the framework and how to program within it, which should only take months in comparison.

Advantages of Developing a Framework In-House

Developing an applications framework in-house provides the organization with total customization of the solution to fit their needs. They may use existing frameworks as models, choosing the features best-suited to the requirements rather than having to take an all-or-nothing approach to buying into a specific framework. Also, the organization doesn't have to write code for features it does not need, such as interfacing with databases it does not use.

This customization allows for streamlining the framework to just those features the organization will use. As a result, the performance of the in-house framework would be far superior to a framework built to support a wide range of environments and backwards compatibility with older systems. A from-scratch framework has the advantage of being able to adopt all the latest innovations in software development.

Organization-Dependent Considerations

It is important to take into consideration the capabilities of the organization when performing this analysis. For instance, does the organization have the proven capabilities to develop a framework that has the same level of quality as one developed by a community of developers from around the world? Does the organization have a proven

track-record of writing thorough documentation so that new developers can quickly learn how to begin programming in the organization's framework? If not, is the organization willing to accept the need to committing personnel to mentoring new developers in this?

The choice to go with an in-house solution should also consider whether this may be a case of "not invented here" syndrome (NIHS):

Most developers would rather be known as the hero who developed the UI framework that the whole company now relies on than simply the guy who made the suggestion to use Tapestry, for instance (Nash, 2009).

NIHS can occur at the organizational-level, as a matter of the business culture, or it can be the result of personal pride at the individual-level.

When considering a framework of component-sized scope, analysis should ask questions specific to the organization's needs, such as: Does the existing solution solve at least 90% of the problem, allowing, via open source licensing or vendor willingness, customization to solve the remaining 10 percent? Is the existing solution appropriately licensed? Is the component built for reuse or easily customized for reuse? Is the existing solution compatible with existing development standards, platform, and programming languages? If the answer is "yes" to each of these questions, then there is no reason not to adopt the existing framework (Nash, 2009).

Conclusions

Software frameworks represent the next logical step from object-oriented programming to maximize reuse, standardize development, and improve productivity within the scope of producing numerous projects or an information system among multiple teams within an organization. A mature applications framework that is extensible and integrateable, which has an architecture that provides features without overly impacting performance can provide a solid, stable solution to an applications service provider. The decision to construct an applications framework should be taken with care to ensure there are sufficient resources and controls in place to ensure the final product will meet all of the organizations needs, and that the framework implements the standard design patterns that make existing frameworks so successful.

References

- Atwood, Jeff 2008. *Understanding Model-View-Controller*. Coding Horror: Programming and Human Factors. May 5, 2008. Retrieved from codinghorror.com June 20, 2009 at: <http://www.codinghorror.com/blog/archives/001112.html>
- Ambler, Scott W. 2009. *Mapping Objects to Relational Databases: O/R Mapping in Detail*. Agile Data. Retrieved from agiledata.org June 20, 2009 at: <http://www.agiledata.org/essays/mappingObjects.html>
- Brown, Lawrence M. 1998. *Composition vs Inheritance*. Dec 31, 1998. Retrieved from apl.jhu.edu May 20, 2009 at: <http://www.apl.jhu.edu/Notes/LMBrown/resource/Composition.pdf>
- Burd , Stephen D., *Systems Architecture Fifth Edition*, Thomson Course Technology, 2006.
- Channabasavaiah, Kishore; Holley, Kerrie, and Tuggle Jr., Edward 2003. *Migrating to a service-oriented architecture*. IBM Technical Library, Dec 16, 2003. Retrieved from ibm.com May 20, 2009 at: <http://www.ibm.com/developerworks/library/ws-migratesoa/>
- Cliff, 2006. *How Do You Decide Which Framework to Use?* Slashdot. Feb 24, 2006. Retrieved from ask.slashdot.org May 20, 2009 at: <http://ask.slashdot.org/article.pl?sid=06/02/24/0148201>
- DevTrench 2007. *PHP Application Framework Battle Royale: CodeIgniter vs. Symfony*. DevTrench. Sep 5, 2007. Retrieved from devtrench.com Jun 20, 2009 at: <http://www.devtrench.com/codeigniter-vs-symfony/>
- Ekerete 2008. *PHP Framework Comparison Benchmarks*. AvnetLabs Jun 30, 2008. Retrieved from avnetlabs.com Jun 20, 2009 at: <http://avnetlabs.com/php/php-framework-comparison-benchmarks>
- Gamma, Erich; Helm, Richard; Johnson, Ralph; and Vlissides, John 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. Westerford, MA.
- Hunt, Andrew and Thomas, David 1999. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley Professional, Boston, MA.
- Johnson, Ralph E. and Foote, Brian 1988. *Designing Reusable Classes*. Journal of Object-Oriented Programming, June/July 1988, Vol. 1, No. 2, pg 22-35. Department of Computer Science, University of Illinois, Urbana, IL.

- Karsten, Carl 2009. *Frameworks Build or Buy*. Visual Foxpro. Retrieved from fox.wikis.com May 12, 2009 at: <http://fox.wikis.com/wc.dll?Wiki~FrameworkNotInventedHere~SoftwareEng>
- Krasner, Glen E. and Pope, Stephen T. 1988. *A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System*. ParcPlace Systems, Mountain View, CA.
- Lorenz, Mark and Kidd, Jeff, 1994. *Object-Oriented Software Metrics*. Prentice Hall, Englewood Cliffs, NJ.
- Mattsson, Michael, 1996. *Object-Oriented Frameworks: A Survey of Methodological Issues*. Department of Computer Science, Lund University, Lund Institute of Technology, Lund, Sweden.
- Mattsson, Michael and Bosch, Jan 1999. *Assessing Object-Oriented Application Framework Maturity – A Replicated Case Study*. Department of Software Engineering and Computer Science, University of Karlskrona/Ronneby, Ronneby, Sweden.
- Miller, John 2007. *Framework Not Invented Here*. Visual Foxpro. Retrieved from fox.wikis.com May 12, 2009 at: <http://fox.wikis.com/wc.dll?Wiki~FrameworkNotInventedHere~SoftwareEng>
- Nash, Michael 2009. *Overcoming “Not Invented Here” Syndrome*. Developer.com. Retrieved from developer.com May 12, 2009 at: <http://www.developer.com/open/article.php/3338791>
- Open Channel Foundation, 2008. *Semantic Web Framework*. Open Channel Foundation, Mitre Corporation. Retrieved from [openchannelsoftware.com](http://www.openchannelsoftware.com/projects/Semantic_Web_Framework/) June 18, 2009 at: http://www.openchannelsoftware.com/projects/Semantic_Web_Framework/
- PatternLanguage 2001. *Christopher Alexander*. Retrieved from PatternLanguage.com on May 5, 2009 at: <http://www.patternlanguage.com/leveltwo/ca.htm>
- PCMag 2009. *Encyclopedia entry for Applications Framework*. Pcmag.com. Retrieved from pcmag.com May 18, 2009 at: http://www.pcmag.com/encyclopedia_term/0,2542,t=application+framework&i=37907,00.asp
- Pfleeger, S., & Atlee, J. (2006). *Software Engineering: Theory and Practice*. New Jersey: Prentice Hall.
- Rajlick, Paul John 1998. *Object Composition vs. Inheritance*. Retrieved from brighton.ncsa.uiuc.edu May 20, 2009 at: <http://brighton.ncsa.uiuc.edu/~prajlich/T/node3.html>

- Reindel, Brian 2007. *How to Choose a JavaScript Framework*. d'bug, Oct 30, 2007. Retrieved from blog.reindel.com May 20, 2009 at: <http://blog.reindel.com/2007/10/30/how-to-choose-a-javascript-framework/>
- Reindel, Brian 2008. *Will You Need a JavaScript Framework on Your Next Project?*. d'bug, Aug 18, 2009. Retrieved from blog.reindel.com May 20, 2009 at: <http://blog.reindel.com/2008/08/18/will-you-need-a-javascript-framework-on-your-next-project/>
- Schmidt, John and Lyle, David 2005. *Integration Competency Center: An Implementation Methodology*.
- Schummer, Till and Lukosch, Stephan 2007. *Patterns for Computer-Mediated Interaction*. John Wiley and Sons, West Sussex, England.
- Smarty, 2008. *Is Smarty Right for Me?* Smarty Template Engine. Retrieved from [smarty.net](http://www.smarty.net) June 20, 2009 at: <http://www.smarty.net/rightforme.php>
- Spatial, 2009. *Rapid Application Development Framework (RADF)*. Spatial Corporation. Retrieved from [spatial.com](http://www.spatial.com) June 18, 2009 at: <http://www.spatial.com/products/radf-3d-development>
- Sun Microsystems, 2002. *Front Controller*. Sun Microsystems Inc. Retrieved from java.sun.com June 20, 2009 at: <http://java.sun.com/blueprints/patterns/FrontController.html>
- Stenhouse, Mike, 2005. *A CSS Framework*. Content with Style, May 22, 2005. Retrieved from [contentwithstyle.co.uk](http://www.contentwithstyle.co.uk) on June 13, 2009 at: <http://www.contentwithstyle.co.uk/content/a-css-framework>
- Wake, William C. (2009). *Growing Frameworks in Java*. Retrieved from xp123.com June 2, 2009: <http://xp123.com/wwake/fw/ch12-bb.htm>
- Zend 2009. *Zend Framework Quick Start*. Zend Technologies Ltd. Retrieved from framework.zend.com Jun 19, 2009 at: <http://framework.zend.com/docs/quickstart>