Application Frameworks: How They Become Your Enemy

Martin Mailloux
University of Illinois at Urbana-Champaign
martin.mailloux@gmail.com

Abstract

Application frameworks have become a de-facto standard to implement business systems. In most organizations, when choosing either a development platform or a commercial solution, an application framework is part of the overall solution. This paper reviews my personal experience developing a proprietary application framework, its lifecycle, software engineering practices, successes and mistakes through its releases.

Categories and Subject Descriptors D.3.3 [**Programming Languages**]: Language Contructs and Features – frameworks.

General Terms Management, Documentation, Performance, Design, Economics, Experimentation, Human Factors.

Keywords Application Frameworks, Coupling, Evolution

1. Introduction

Application frameworks are central to most system development; they can either greatly facilitate or impair the implementation. Many in-house application frameworks are solution specific to their business domain, they provide the "silver bullet" [1] to the team, making it more productive and improving the quality by resting on the shoulders of previous releases. In most organizations, mastering the proprietary application framework enables greater developer productivity.

In many situations, an application framework will remain in use, as long as a minimal expertise remains within the team. As both the team's personnel change and the system goes through releases, its usage and longevity may suffer a downturn. Application frameworks can represent challenging design and conceptual work; enticing deep understanding of the technology, complex algorithm, integration with the operating system.

The demise of the American programmer was predicted over a decade ago [10] and with the flattening of the world [6], they are in competition with developers from around the world. Commoditization of IT [3] is an added pressure to increase developer productivity. In this paper I review my personal experience building a proprietary application framework through its releases which spanned many years. The information provided is based on my past experience as a software architect, but it also includes information made available to me by past colleagues. It was a successful development in many aspects for the early releases but the last releases were challenging. I document the successes, failures and challenges that each release brought to the team in terms of software engineering practices, software architecture and organizational structure.

Unlike most research on application frameworks, a study was done independently to evaluate the value of migrating through different releases of the application framework [4]. The study focused on the early releases of the application framework and concluded it was positive in terms of effort versus new functionality.

Application framework just like its counterpart, application system, evolves through time by subsequent chains of minor improvements and a few major (big bang) evolutions. To better understand the architectural changes that occurred through time in the application framework, information regarding the company strategy is provided and the evolution of the software engineering practices.

2. Organization Specificity

The organization was involved in developing website systems for its own usage, and all websites were not only sharing the application framework but some of the business functionality. The websites were Business To Business (B2B) e-marketplace for a specific industry. All websites were online transactional processing (OLTP) systems, requiring high availability, with worldwide clientele (24/7). Specific extension and capability was provided to enable customization of application functionality for each website. The application framework supported the encapsulation of system level functionality but also extensibility features such as business rules specialization found in business oriented application frameworks such as the San Francisco framework [2].

The software development group (SDG) was divided in three distinct teams:

• Software Infrastructure: in charge of the application framework.

• Business Component: in charge of providing generic business components to implement business functionality.

• Website Family: in charge of developing all the websites within a common business domain. Amongst the websites that a website family team was in charge of, many were related to a similar industry in terms of functionality and business requirements. A website family team would implement common functionality into a generic website for its family, which would then be specialized.

3. Releases

3.1 Bare Bones

Description: it represents the initial implementation, with very few websites being developed by the organization; most developers had limited professional experience.

Organization: the overall company had less than 30 employees.

Table 1: Bare Bones Software Engineering Practice

Project planning	None
Project management	• Most specifications could fit on a single page. • Release management was non existent, write code→clean compiled→ship to production→wait for the comeback.
Configuration management	• An excel spreadsheet is used to track who is modifying which file. • Development and production environment only.
Software Quality Assurance	No bug tracking tool (except excel).
Change management	None.
Programmer's Life	No enforcing of the few rules, collegiate environment.

Software Architecture: the implementation was divided in 4 layers: template, controller, entity object, entity accessor.

- Template: an html page template was divided in sections, such as header, body and footer. Within the template, placeholders were used to add content at runtime.
- Controller: process a request and implement the business rules, interacts directly with the entity object and entity accessor to execute the request and generate the response using template.
- Entity: an entity represents a Java implementation of a relationnal table. It did not implement any business logic and was a data object (getter/setter) to interact with its entity accessor.
- Entity accessor: an entity accessor was composed of two sections, one generated by the framework and a user defined part. A generator was used to generate both the entity and the entity accessor based on a create table statement.

No factory was used within the system, therefore to begin the implementation of a website, a set of scaffolding classes was available. The scaffolding classes were composed of classes prefixed with "AB" such as "ABUser". Within the code, instantiation of objects were instructed by "new ABUser()". Each Website class was prefixed with a set of specific characters, such as "TEST". The generator processed the class' sources, and generated the specific code for a website (it would generate the class file and rename all AB to the specific prefix for the website).

Lesson Learned:

- Learning curve: the application framework used very basic techniques; therefore it was assimilated very rapidly. If a programmer could do an SQL statement, they could very rapidly code an entity and its accessor, and write a specific method for a controller. The template system, even though proprietary, was very close to other templating systems of that time.

- Protection: there was no protection against programming errors, (ie., not closing a 'PreparedStatement'). An unclosed PreparedStatement by a Controller will not release the resource at the database level and will exhaust in the long run the available resource.

- Maintainability: the website could easily be maintained and the overall classes/code to implement functionality could very rapidly be found and traced by a developer. The code was using specialization to implement specific business rules. There was no indirection in the code and very few abstractions were used in the system.

- Generator: very easy and powerful way to generate the scaffolding code for a website. The drawback was that many classes were generated only to instantiate specific classes. To implement modifications of the application framework could require regenerating the website code.

3.2 1st Abstraction

Description: It was the first release of the application framework, where the 'whitebox' framework was not a set of generated classes specific for each website. A basic Class factory was implemented which required minimal configuration to instantiate the specialized classes by the business components or the website. Specialization of business rules was still based on overloading the method within the class hierarchy.

Organization: the software development group had grown, but was still comprised of less than 30 employees.

Table 2: 1st Abstraction Software Engineering Practice

Project planning	None
Project management	• No development methodology. • Release management: no change.
Configuration management	• Quality Assurance and Pre-Production environment are added to the Development and Production environment. • A proprietary build system is implemented.
Software Quality Assurance	• No bug tracking tool (except excel). • A Quality Assurance Group is in place: QA is on overdrive to pickup the past releases.
Change management	Informal one-on-one coaching
Programmer's Life	• Programming standard emerged. • Code generation for Entity was available. A java program interprets the SQL Data Definition Language (DDL) and based on its specification, generates the properties and the getters/setters.

Software Architecture: A new business object emerged, the Transactional Object (TXO). It will be used to remove some of the business related processing from the controller. Common interface started to be used to specify class signature.

Lesson Learned:

- Learning curve: initial loss of control due to the introduction of the class factory and its added abstraction was an acceptable solution in relation to its cost/benefit.
- Protection: it introduces a purer whitebox framework without code generation. As loopholes were found in the application framework, the correction did not require anymore re-generating the website.
- Maintainability: the drawback of using a factory to instantiate an object is greatly alleviated by the removal of all the repeated code. The business code is not mixed with framework related code.

3.3 Commercial

Description: This iteration of the application framework was in reaction to the business initiative of the company, to facilitate the commercialization of its solution if given the opportunity. The initial application framework was based solely on proprietary technology, from the templating system all the way to the application server itself. Only Java and its components, such as JDBC were used in the application server. As J2EE was becoming the fad of the day, a new direction was set for the application framework. The business entity became J2EE Entity (the Entity and Entity accessor were merged) and the TXO a stateless Session bean. It was decided to modify the proprietary application server to support the J2EE semantic instead of purchasing and migrating the production infrastructure.

Organization: the software development group was growing, and within a year it was over 100 employees.

Table 3: Commercial Software Engineering Practice

Project planning	Websites begin to implement project plan.
Project management	No change.
Configuration management	A configuration manager is implemented.
Software Quality Assurance:	No change.
Change management	Formal documentation: "How-to".
Programmer's Life	A code generator was implemented, to generate the required Interface, Entity and the J2EE proxy object. First release of the Developer Workbench.

Software Architecture: the overall layered architecture remained the same. The modifications implied changes at each level, but their design responsibility remained the same.

Lesson Learned:

- Organization growth: the rapid growth of the software development group requires documenting and formal training on our technology. The original few are spread amongst the teams in an effort to provide in-team support. The average tenure for team members in the company can almost be counted in weeks.
- Learning curve: this release is a major rewrite of the business object implementation, but the overall learning curve is fast as most of the concepts are very similar.
- Inertia: the rapid growth of the software development group and in the number of websites that it must support, adds to the challenge of implementing through the website teams a new release of the application framework.
- Maintainability: having its proprietary J2EE implementation, no commercial IDE is available to help developer productivity. Projects are started to integrate the continuous build platform into a tool (Developer Workbench) to automate and facilitate the tasks of managing the project configuration.

3.4 Advanced Business Function

Description: The software development group initiates the centralization of common business function development to a central business component team. The more complex business functions are redesigned, to implement a configurable workflow system to facilitate the customization across websites.

Organization: the software development group is stable, with a very low turnover.

Table 4: Advanced Business Function

Software Engineering Practice

Project planning	The application framework is following a release management, with planned release and features set (Priority: Required, High, Medium, Low).
Project management	No change.
Configuration management	Documentation for Major/Minor release and overall release policy
Software Quality Assurance	Automated testing is implemented by the websites, but not for the application framework.
Change management	• A preliminary description of each release is communicated to the website teams. • Formal release notes includes: dependency & compatibility, migration activities, fixes & improvement, business layer modifications.
Programmer's Life	TogetherJ is now used as platform for code generation.

Software Architecture: the layered architecture is reviewed to enforce greater de-coupling between the presentation and the business layer. XML serialization of Entity is added and XSLT template is supported alongside with the previous template system. A new business object, New Entity, is implemented to replace the J2EE entity. The New Entity is implemented to fully support inheritance, unlike its DDL predecessor. It is modeled using TogetherJ. TogetherJ code generation is too generic and not specific to the application framework. A custom code generator is integrated with TogetherJ.

Lesson Learned:

- Decoupling: XML provided a very strong decoupling mechanism between the presentation and the business layer. Within the business layer and through the persistence layer, XML processing was more costly to program and also to execute. It could adversely influence performance due to the cost of XPath and/or serializing/de-serializing the objects. In an effort to decouple all aspects of the application system, all dependencies were transferred into configuration elements. Using XML as a decoupling mechanism moved most of the basic validation from the compilation to the runtime realm. Code quality degradation cannot be tracked by using coupling as a criteria [7] with XML as a integration scheme.
- Layers vs Workflow: most designs of a workflow, start with the state machine and the pre/post condition. The workflow system had to be highly customizable to meet the variety of requirements across all the websites. The current layering implementation between the presentation and the business layers is based on using XML as communication protocol. This requires specific processing for the workflow system and creates a dependency in its configuration and its counterpart at the presentation layer. The workflow system may dictate as it changes state the next user interface to be displayed. If a pre-condition is added and adds a user input, the presentation layer configuration must be modified to enable this use case.
- Role Based Access Control (RBAC) [FK92]: to further improve the granularity and customization of the

business components, the ACL security control was replaced with a RBAC system. The workflow system was integrated with the RBAC system, to provide greater security in its implementation. The configurable element at both the workflow and at the RBAC level made it almost impossible to predict the dependency between adding a condition in the workflow and the required RBAC modifications. Conflicts arise between achieving a configuration of RBAC at a higher level to simplify its configuration and its impact on the workflow's condition.

- Learning curve: as new technology and abstraction concepts are introduced, the short term productivity is lowered. The strategy is through greater decoupling, the cost of customization should be reduced in the long term.
- Maintainability: decoupling between the components using XML, made the current IDE (Eclipse) ill-adapted. We lost the power of live debugging and checking objects/variables states at run time. Properly writing a configuration file became almost a programming language of its own.
- Configuration: the number of errors caused by the high level of configurable elements in the components configuration, required building a configuration browser/editor. Also, as the configuration became a central location for dependency between components, it was also the main element in configuring the components across the environment (development, quality assurance, pre-production, production). From the original configuration file in XML it grew into a XSLT like file. Where specific keywords could be used within the configuration to specify based on the environment/servers specific parameters to the components (such as database username/password).

3.5 User Experience

Description: To improve the User Experience, we initiated a stronger binding, through configuration and naming convention, between the presentation layer and the business layer. One of the caveats of web development, compared to a 4th generation programming platform, is field validation such as maximum input must be duplicated between the layers. The extent to which a business object will be customized across the website could not be predicted by its designer. The ease of modifying a field's attribute at the configuration level did not have its equivalent all the way to the presentation layer. Basic dynamic binding using 'contract' between the layers was initiated, binding the field's attribute of the persistence layer across all the layers.

Organization: the software development group is stable, with very low turnover.

Table 5: User Experience Software Engineering Practice

Project planning	No change.
Project management	No change.
Configuration management	The business component group is challenged in implementing a release management, with planned releases and feature sets. The customization by some websites makes it too costly to migrate them to the newer version.
Software Quality Assurance	A bug tracking tool is implemented (bugzilla).
Change management	• A preliminary description of each release is communicated to the website teams. • Formal release note includes: dependency & compatibility, migration activities, fixes & improvements, business layer modifications.
Programmer's Life	A developer workbench is developed to integrate the proprietary tools (build system, management console).

Software Architecture: to provide a refined user interface, such as highlighting field in error, a mechanism that enables to customize errors between the layers is implemented. The error management by itself is very complex and costly to use effectively.

Lesson Learned:

- Decoupling: To make the components work together we kept adding more configurations. Architects and designers greatly appreciated the advantage, but developer productivity started to be a challenge.
- Learning curve: more abstraction added more proprietary technology, which required more proprietary tools added to Eclipse to alleviate the development effort. Our development teams were no longer Java specialists, but experts (some will say PhD) in our proprietary technology. Error messages and stack trace were not indicative to the source of the problem, but required interpretation to diagnose the issue.
- Maintainability: in most cases, a runtime error generated a stack trace that included calls only to the framework classes. Most business objects became anonymous and were dynamically configured.
- Configuration: it seems that most developers spent more time trying to decrypt their configuration then doing java coding.

- Contract: the contract system provided a mechanism to manage the difference in specification across the layers. As an example, in a business object 'User' the username may be mandatory. In the search function for users, the username must follow the attribute length, but not inherit that it is mandatory. Contracts were dynamically generated for Create/Read/Update/Delete (CRUD) services on an object, but complex business transactions had to build their contract step by step. At the business function level, the contract represented a Service Oriented Architecture (SOAP), but within the system implementation itself, all operations had become services. Developers on top of configuring the business object had become contract experts.

  The standardization of literal values for numbers, strings, dates, was standardized in the earlier release, following the "Whole Value" pattern [5]. The deferred validation pattern was implemented to provide an initial pass of validation for a form. Instead of having each field process all its validation at once, a two phase validation was implemented. The first phase provided basic unit field validation such as: mandatory/optional, length, format. Any error during this phase, produced a feedback to the user about which fields were incorrectly entered. The second phase of validation was related to complex business rules, which included inter-field dependency such as verifying a begin date is before or equal to the end date.

3.6 Going Horizontal

Description: the model of centralizing the development of common business objects, but going through a distributed model for customization became inefficient. The distributed customization incurs a very high cost of training each website team in the specificity of each common business module. Also, the centralized team lacked the opportunity to synchronize the websites' release in accordance to their own schedule. Websites were allowed an 'a la carte' choice, instead of the 'buffet' approach, and therefore would pick and choose to upgrade only a few modules at a time. The cost to manage the diverse version of the common modules, and their inter-dependencies, made it impractical to have an 'agile' response time.

Organization: the software development group was reorganized based on the common business modules that existed. Each team was in charge of a specific set of common business modules and its customization.

Table 6: Going Horizontal Software Engineering Practice

Project planning	Each team plans its activities, with informal inter-dependency scheduling.
Project management	Initial attempt to initiate a Project Management Office (PMO).
Configuration management	No change.
Software Quality Assurance	Quality Assurance was divided across the business modules team.
Change management	An initial training was provided to the development team. The application framework rate of change was so fast, that rapidly the initial training became obsolete. There are no formal meetings/communication channels to funnel the information.
Programmer's Life	The developer workbench/open-source or commercial tools are not providing the proper toolset to enable productive work with this application framework.

Software Architecture: the organization takes this opportunity to expand the Contract system. The primary goals were to leverage emerging open-source technologies and to use a declarative approach. It leveraged the metadata information from the entity as a source to define the contract.

This release of the application framework, unlike the previous release, could not upgrade one website at a time to track the tasks required and fix the problems that arose.

The new organization structure meant it had to upgrade all the business module teams at once, to have one functional website. The major challenges for the infrastructure team were:

- Persistence Layer: major upgrades to the persistence layer are performed to make it more declarative. It is migrated from a proprietary ORM implementation to Hibernate.
- Declaractive programming: very complex processing is added to support the declarative approach across all the layers.
- Website as assembly: websites are not whole anymore. They are an assembly of services. Distributing the workload across servers based on the website must be modified to support the distribution by business module.
- Distribution model: the distribution model is also supported at each layer, distributing the presentation and business/persistence layer across servers.

The overhaul of the application framework is initiated as the functional module teams are created. During the first four months, the functional module teams are involved in requirements gathering across all the websites. Training sessions are organized to bootstrap the team on the technology. As implementation begins, missing functionality or heavy/redundant configurations are modified. No efforts are spared in having an application framework that will require no programming, only declarative configuration. The declarative programming is implemented by specifying all rules in XML. Code walkthroughs are also performed to monitor what is lacking in the declarative programming. The application framework is extended as business modules are developed to support them through declarative programming. The modifications are retrofitted into the current development.

In the first few weeks when the implementation began, developers encountered problems with the application framework that were resolved promptly without many side effects. As the development effort gained momentum, more developers were using the application framework encountering more problems and limitations.

Throughout its lifecycle, the application framework had very limited formal quality assurance performed prior to a website upgrade. Ad-hoc unit tests were performed by each developer on the application framework with each modification. The first website migration was performed by the application framework team, assisted by the website technical lead.

Under the horizontal team structure all the teams had to be affected at once by any changes to the application framework. Any deficiency in quality assurance by the application framework team increased the risk of downturn with all the teams experiencing bugs, stopping them in their tracks. As expected, the downward spiral of having more teams finding more bugs meant tighter deadlines for the application framework team to release fixes, increasing the risk of regression bugs showing up.

The application framework was upgraded every day, with all the teams linking directly to the development version to have access to the latest fixes. One way to minimize the risk was to slow down the rate of releases of the application framework and to stabilize each version prior to release. With increasing pressure to provide the missing functionality, proceeding forward was deemed the best solution.

Lesson Learned:

- Pulling the Carpet: having a whole software development organization developing on an application framework that was modified beneath them was a risky proposition and it was proven to be a costly approach.
- Pilot Project: no pilot project was initiated, or even in this case, a pilot/test website to support future quality assurance activities. The pilot project should have led the way in functionality and been used as a demonstration/teaching platform for the teams.
- Champion: as the application framework team got overwhelmed with the development/bug fixing/support cycle, having a champion in each business module team would have eased the problems. Champions would have been a key asset in communicating the best practices as well as serving as a single point of communication.
- Toolset/IDE: as new technology is released, having the proper toolset/IDE to support it is essential. As an example, the lack of validation in the configuration makes it very costly to track and fix a problem. Poor support of the developer in repetitive tasks or having complex configuration files, creates a barrier to the adoption of the application framework.
- Weekly gathering: no organized communication structure was established to gather either the technical lead on each team or at the management level.
- Moral: developers did not see it in a positive light, to become a XML specifier instead of a Java developer.
- Quality: an application framework initial quality + support of the developers in their task would be proportional to its acceptance. Each issue encountered slowed the curve of acceptance of the new framework.
- Layering: The new application framework implemented two distribution layers for a website. The traditional distribution between presentation and business layer was implemented between two instances of application servers, but also, across servers for each functional module. Each functional module could be leveraged across multiple websites. The double distribution and dependency across functional modules (i.e., each are dependent on the security module) required complex operational process for upgrades or system restart.
- Remote layering: as a means to provide higher scalability and remove all coupling between servers, each invocation across layers went through a queuing system. The queuing system was implemented using a persistent manager. Each layer was conceptually implementing a service oriented approach, publishing its services and guaranteeing the execution of the request.

SOA: As described in [12], the major goals of a service oriented architecture vs an object oriented analysis-design are:

- Increased Business Requirements Fulfillment
- Increased Robustness
- Increased Extensibility
- Increased Flexibility
- Increased Reusability and Productivity.

With the promised gain of a SOA approach, one can only question why almost the opposite occurred. All aspects of SOA were leveraged: service contracts, coupling, abstraction, reusability, autonomy, statelessness, discoverability and composability. The major misstep came in having a very small granularity level for the service definition. The system was not anymore implemented using an object-oriented approach, but instead services built by aggregating atomic services together. No distinction was done between an internal service required only within one business transaction and one by an external module. An internal service would be implemented in a more straightforward manner through a traditional OO approach. Toolsets to support SOA, especially complex service composition, were not available and had to be scripted manually.

A service orientation architectural direction is justifiable if requirements such as a heterogeneous platform [11] is required. The operational criteria under which the system was deployed did not have such requirements.

3.7 Open-Source

Description: after spending much time, energy and a massive capital investment, the organization judged that the current direction was not achieving its goals and could not be sustained. Conversion to the Going Horizontal application framework was suspended; some functional modules remain implemented with the User Experience release. It did a first prototype based on open-source technology (Spring, Hibernate, JSP), which was then standardized for future development. As the open-source release was deployed, efforts are engaged to break the ties between websites still dependent on the previous release.

Organization: the software development group is regrouped around websites, as it was previously.

Software Architecture: the software architecture is now a mix of three releases, User Experience, Going Horizontal and Open-Source.

4. Conclusion

Application frameworks are part of most systems; some are part of a commercial platform and others a proprietary implementation. In all cases they support a set of implementation patterns relevant to their business domain, with the goal to reduce time to market and cost of development. Our business domain was specific and as such, not very likely to attract the attention of a researcher or of a software provider. Some of the challenges we encountered were common across the industry, such as implementing an Object-Relational-Mapping system. Today, commercial and open-source solutions are readily available for many of the requirements we had.

The area where we encountered the most difficulty, contract management, is the one area where both research and the industry have not advanced to answer our requirements.

Acknowledgments

I would like to thank Professor Ralph E. Johnson at the University of Illinois at Urbana-Champaign, who served as my supervisor for an independent study which was the original bases for this paper.

References

[1] Brooks F. P., The mythical man-month, Addison Wesley, 1995.

[2] Carey J., Carlson B., Graser T., San Francisco Design Patterns: blueprints for business software, Addison Wesley, 2000.

[3] Carr N. G., IT Doesn't Matter, May 2003, Vol 81, Issue 5, p. 41-49, Harvard Business Review, 2003.

[4] Corrales Y., Laporte C. Y., Étude de cas : Évaluation de la Migration d'une Architecture Logicielle d'une Société de Commerce Électronique, Génie Logiciel, N. 82, Septembre 2007.

[5] Cunningham & Cunningham Inc., The CHECKS Pattern Language for Information Integrity, Site reviewed on March 11th, 2009.

[6] Friedman T. L., The World is Flat: A Brief History of the Twenty-first Century, Farrar-Straus and Giroux, 2005.

[7] Subramaniam G. V., Object Model Resurrection - An Object Oriented Maintenance Activity, ICSE 2000, ACM, 2000.

[8] Ferraiolo D. F., Kuhn R. D., Role-Based Access Controls, 15th National Computer Security Conference (1992), Baltimore MD, pp. 554 – 563, 1992.

[9] Weinberg G.M., The Psychology of Computer Programming, Silver Anniversary edition, Dorset House, 1998.

[10] Yourdon E., Decline & Fall of the American Programmer, Yourdon Press, 1993.

[11] Mariani R., Bohling B., Smith C. U. Barber S., Improving .Net Application Performance and Scalability, Microsoft Corporation, 2004.

[12] Erl T., SOA Principles of Service Design, Prentice Hall, 2008.