Achieving Agility Through Architecture Visibility

Carl Hinsman[1], Neeraj Sangal[2], and Judith Stafford[3]

[1] L.L.Bean, Inc.
Freeport, Maine USA
chinsman@llbean.com

[2] Lattix, Inc.
Andover, Massachusetts USA
neeraj.sangal@lattix.com

[3] Tufts University
Medford, Massachusetts USA
jas@cs.tufts.edu

Abstract. L.L.Bean is a large retail organization whose development processes must be agile in order to allow rapid enhancement and maintenance of its technology infrastructure. Over the past decade L.L.Bean's software code-base had become brittle and difficult to evolve. An effort was launched to identify and develop new approaches to software development that would enable ongoing agility to support the ever-increasing demands of a successful business. This paper recounts L.L.Bean's effort in restructuring its code-base and adoption of process improvements that support an architecture-based agile approach to development, governance, and maintenance. Unlike traditional refactoring, this effort was guided by an architectural blueprint that was created in a Dependency Structure Matrix where the refactoring was first prototyped before being applied to the actual code base.

Keywords: architecture, dependency, agility.

1 Introduction

This paper reports on L.L.Bean, Inc.'s experience in infusing new life to its evolving software systems through the increased visibility into its system's architecture through the extraction and abstraction of code dependencies. Over years of software development the information technology infrastructure at L.L.Bean had become difficult to maintain and evolve. It has long been claimed that visibility of architectural dependencies could help an organization in L.L.Bean's position [9][12][17]. In this paper we provide support for these claims and demonstrate the value of applying these emerging technologies to a large, commercial code-base. We explore the strengths and weaknesses of the application of the Dependence Structure Matrix (DSM) as implemented in the Lattix LDM [16], to improve the agility of the L.L.Bean code base, and propose avenues for follow-on research to further improve tool support for architecture-based refactoring in support of software agility.

L.L.Bean has been a trusted source for quality apparel, reliable outdoor equipment and expert advice for nearly 100 years[1]. L.L.Bean's software is used to manage its sales, which include retail, mail-order catalog, as well as on-line sales, inventory, and human resources. More than a 100 architects, engineers, and developers work on continual improvement and enhancement of the company's information technology infrastructure, which for the last 8 years, has suffered the typical problems associated with rapid evolution such as increased fragility and decreased intellectual control resulting in increased difficulty in building the system. While the company's software development processes have long included a number of good practices and coding rules to help avoid these issues, in the end the speed of evolution overwhelmed the development teams and maintenance and evolution of the software infrastructure were recognized as chief concerns by upper management of the company. Investigation into the core cause of the problems pointed to the fact that the code had become a complex entanglement of interdependencies. It was decided that the code must be restructured and software engineering process must be enhanced to prevent the web of dependencies from recurring in the future.

Refactoring, as described by Fowler et al. [4] and others in the object community is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior. Refactoring is generally practiced as a series of small changes to classes and collections of classes that modify the structure in a way that improves the code or moves it in an intended direction so that the code is better structured for future enhancements, improved comprehensibility, easier unit testing etc. There are a number of tools that provide support for refactoring (e.g. Eclipse and Intellij). These tools provide a variety of capabilities such as generating 'getters/setters' and 'constructors' etc that simplify code modifications. However, the approach of localized modifications was too limited and L.L.Bean recognized the need to approach solving the problem from a global perspective.

Because the software architecture of a system is a model of software elements and their interconnections that provides a global view of the system it allows an organization to maintain intellectual control over the software and provides support for communication among stakeholders [2][18]. As such, it seemed an architectural approach to "refactoring" L.L.Bean's code base would be appropriate. A variety of approaches for exploring architectural visibility were explored. Ultimately, an approach based on a Dependency Structure Matrix (DSM) [15] representation was selected because of its innate ability to scale and the ease with which alternative architectural organizations could be explored.

L.L.Bean's strategic approach to refactoring required few code changes but rather a code consolidation followed by a series of structural modifications. Unlike current approaches to refactoring, L.L.Bean's method is driven by overall visibility of the architecture and includes five steps: define the problem, visualize the current architecture, model the desired architecture in terms of

---

[1] http://www.llbean.com

current elements, consolidate and repackage the code base, and automate governance of the architecture through continuous integration.

This approach to software development employs the Lattix Architecture Management System as the primary tool for architectural analysis and management. It also uses custom tools developed at L.L.Bean for automating changes to the code organization, and for maintaining visibility of the state of evolving dependencies on a continuing basis.

The remainder of the paper recounts the L.L.Bean experience in "refactoring" and describes the architecture-based approach to software development that has been created and adopted as an organizational standard at L.L.Bean. The new standard was welcomed by all development teams and provides a mechanism for continuous improvement as the technology infrastructure evolves to meet ever-growing business demands of this increasingly popular brand.

We begin our report with a description of the problem facing L.L.Bean's software developers. This is followed by a recounting of research toward identifying a viable solution and the basis for the decision to apply an approach based on a Dependency Structure Matrix. We then provide an overview of this approach in enough detail to support the reader's understanding of this report, and follow that with description of our experience using and extending the Lattix tools at L.L.Bean. We then summarize lessons learned through this experience and propose avenues for future research in providing additional mechanisms to maintain architecture-based agility.

2 Background

2.1 IT Infrastructure

A significant part of L.L.Bean's information technology infrastructure is written in Java and runs on Windows, UNIX, or Linux based servers. The system has gone through a rapid evolution over the last eight years due to several massive development efforts undertaken in response to increased demand from multiple business units. New front-end systems, strategic web site updates, regular infrastructure improvements, external product integration, and security have been among the key drivers.

L.L.Bean develops software primarily in Java and follows object oriented programming principles and patterns [5]. Development teams normally consist of ten or fewer developers grouped by business domains such as product, order capture, human resources, and IT infrastructure. Package names are chosen to represent behavior and/or responsibility of groups of Java classes within the package. Aligning development structure with naming conventions facilitates reuse and helps avoid duplication of effort by increasing visibility. Although it does not address interdependencies among modules, this alignment was an important contributor to the success of this project.

The current system has more than one million lines of code assembled into more than a 100 jar files. In turn, the actual code is organized into nearly 1,000 Java packages and more than 3,000 Java classes. Despite the use of good software development practices and standards, normal code evolution created a complex entanglement of interdependencies, increasing software development and maintenance costs, and decreasing reuse potential. Multiple code bases diverged over time, which increased complexity significantly.

Initially, ad-hoc approaches were tried to deal with these problems. Interdependency issues were mainly identified by configuration managers while attempting to compile and assemble applications for deployment. These were then fixed one dependency entanglement at a time. The process was slow and correcting one problem often led to a different set of problems. One significant effort for resolving core dependency entanglements consumed three man weeks of effort and was not entirely successful. Moreover, there was nothing to prevent entanglements from recurring.

Business needs continued to drive new development, and interdependency entanglements continued to grow. Software development and configuration management costs increased in stride. IT management understood the economic significance of reuse and created a small team of software engineers focused on creating and implementing a comprehensive packaging and reuse strategy. This team quickly identified the following key issues:

- Too many interdependencies increased testing and maintenance costs
- Multiple Code Bases (segmented somewhat by channel) resulted from the rapid evolution and could not be merged. A goal of the effort was to consolidate into a single code base and transition the development life cycle to a producer/consumer paradigm.
- Architecture was not visible and no person or group in the organization maintained intellectual control over the software
- There was no mechanism to govern system evolution

A solution was needed that supported immediate needs while providing the framework for refactoring the architecture to prevent costly entanglements from recurring.

2.2 Preliminary Research and Tool Selection

There were two key tasks. First, research the abstract nature of software packaging from various viewpoints. Second, create a clear and detailed understanding of the existing static dependencies in L.L.Bean's Java code base. What dependencies actually existed? Were there patterns to these dependencies?

In addition to the major goals of eliminating undesirable dependencies and governing packaging of future software development, the resulting packaging structure needed to accomplish other goals. First, provide a single, consolidated code base to support a producer/consumer paradigm (where development teams

consume compiled code instead of merging source code into their development streams) while helping to define code ownership and responsibility. Next, dynamically generate a view of the interdependencies of deliverable software assets. Last, minimize the cost and effort required to compile and assemble deliverable software assets. An unstated goal held by the team was to increase the level of reuse by fostering a Java development community and increase communication between development teams.

It was important to build confidence in the new strategy. The business, and more specifically the development teams supporting the various business domains, would not entertain undertaking a restructuring effort without evidence of the soundness of the strategy. The team understood that the way to earn this trust was through research, communication and prototyping.

Literature Search

As a starting point, the team sought articles and academic papers primarily through the Internet. Managing dependencies is not a new problem, and considerable research and analysis on a wide range of concepts and approaches was available to the strategy development team [3][6][7][10][11][14][15][17]. Another effort was underway at L.L.Bean to create a strategy for reuse metrics; there was overlap between these two efforts [7][8][13][14]. Much of the research suggested that code packaging in domain-oriented software could promote reuse and facilitate metrics. Exploring these metrics, and tools to support them, provided additional focus. Transition of research into practice would happen more quickly with increased effort on both sides to bridge the researcher/practitioner communication chasm.

Analysis Tools

There are many tools available for detecting and modeling dependencies in Java. The team wanted to find the most comprehensive and easy to understand tool. Initially, open-source tools were selected for evaluation. These included Dependency Finder[2] and JDepend[3] (output visualized through Graphviz[4]), among others[5,6]. Each of these tools were useful in understanding the state of dependencies, but none of them offered the comprehensive, easy to understand, global view needed nor did they provide support for restructuring or communication among various stakeholders, which included IT managers, architects, and developers.

---

[2] http://depfind.sourceforge.net/
[3] http://clarkware.com/software/JDepend.html
[4] http://www.graphviz.org/
[5] http://java-source.net/open-source/code-analyzers/byecycle
[6] http://java-source.net/open-source/code-analyzers/classycle

Graphing the analysis was cumbersome, requiring the use of a combination of tools to produce illustrations of problem spaces. One solution was to use JDepend to analyze the code base, which outputs XML. This output was then transformed into the format required by Graphviz for generating directed graphs. The process was computationally intensive, and there was a limit to the amount of code that could be analyzed collectively in this fashion. Furthermore, when this view was generated it was nearly incomprehensible and of little practical value in either communicating or managing the architecture. Using these tools was inefficient and less effective than had been anticipated. After a problem was identified, it was necessary to code or compile a potential solution and then repeat the entire analysis to illustrate the real impact. Given the extent of the interdependency entanglement, identifying and fixing problems through this approach was found to be too cumbersome to be practical.

L.L.Bean's research identified the Lattix matrix-based dependency analysis tool as promising and, through experience, found it to be effective in that it offered a comprehensive easy to understand interface as well as mechanisms for prototyping and applying architecture rules, and supporting "what if" analysis without code modification.

The Lattix Architecture Management System

Lattix has pioneered an approach using system interdependencies to create an accurate blueprint of software applications, databases and systems. To build the initial Lattix model, the LDM tool is pointed at a set of Java jar files. Within minutes, the tool creates a "dependency structure matrix" (DSM)[7] that shows the static dependencies in the code base. Lattix generated DSMs have a hierarchical structure, where the default hierarchy reflects the jar and the package structure.

This approach to visualization also overcomes the scaling problems that L.L.Bean encountered with directed graphs. Furthermore, Lattix allows users to edit system structures to run what-if scenarios and to specify design rules to formalize, communicate, and enforce architectural constraints. This means that an alternate structure, which represents the desired architectural intent, can be manipulated and maintained even if the code structure is not immediately a true reflection. Once an architecture is specified Lattix allows that architecture be monitored in batch mode and key stakeholders are notified of the results.

The Lattix DSM also offers partitioning algorithms to group and re-order subsystems. The result of this analysis shows the layering of the subsystems as well as the grouping of subsystems that are coupled through cyclic dependencies.

[7] http://www.dsmweb.org

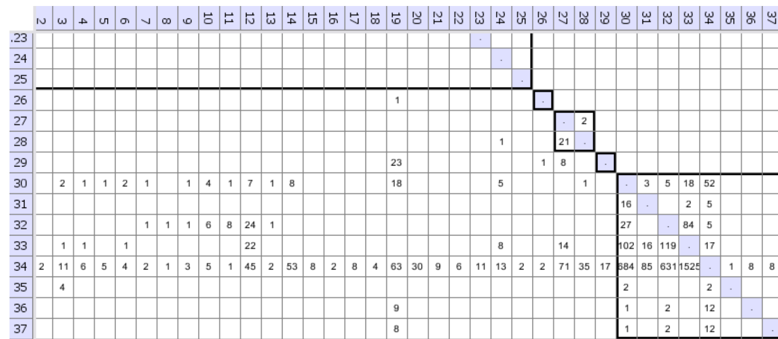

Fig. 1. Using Lattix LDM to Reveal Layering

3 Refactoring the Architecture

With tool support and good development practices in place, L.L.Bean created a five-step approach to architecture-based maintenance that increased the agility of our software development process.

STEP 1: Mapping the Initial State

The first step in the architecture-based refactoring process was to illuminate the state of the code base. An initial DSM was created by loading all Java jars into a Lattix LDM. Then the subsystems in the DSM were organized into layers [1]. The magnitude of the problem became apparent once this DSM was created. It was possible to see numerous undesirable dependencies where application code was being referenced by frameworks and utility functions. The example shown in Fig. 1 illustrates a highly complex and cyclic dependency grouping.

A DSM is a square matrix with each subsystem being represented by a row and column. The rows and columns are numbered for ease of reference and to reduce clutter. The results of DSM partitioning, the goal of which is to group subsystems together in layers, can be evidenced by the lower triangular nature of the upper left-hand portion of the matrix shown in Fig. 1. Each layer in turn is composed of subsystems that are either strongly connected or independent of each other. In this figure, the presence of dependencies above the diagonal in the lower right-hand grouping shows us that subsystems 30..37 are circularly connected. For instance, if you look down column 31, you will see that subsystem 31 depends on subsystem 30 with strength of ‘3’. Going further down column 31, we also note that subsystem 31 depends on subsystems 33 and 34 with strengths of ‘16’ and ‘85’, respectively. By clicking on an individual cell one can see the complete list of dependencies in an attached context sensitive display panel. The DSM view of the L.L.Bean system immediately shed light on the sources of maintenance problems.

STEP 2: Modeling the Desired Architecture

L.L.Bean's Java packaging strategy is a layered architecture [1] that leverages the earlier successes of the package naming approach, adding high-level organizational constructs and rules to govern static dependencies. The Java classes could be grouped into three categories: domain independent, domain specific, and application specific. These classes can be organized into common layers according to their specificity with the most generalized layers at the bottom and the most specific layers at the top.

A layered packaging architecture has several benefits. It is intuitive enough to be understood by a wide range of stakeholders, from software developers and engineers to those with only a limited technical understanding of software development, facilitating discussions between IT managers, project leaders, domain architects, configuration engineers, and software developers.

As the L.L.Bean development organization grew larger and more diverse, increasing communication and coordination across software development efforts also became more difficult. It was hoped that a cohesive and layered architecture would simplify development and improve communication. Clearly communicated and implemented architectural intent would allow teams to develop software components, services and applications without creating undesirable dependencies. Moreover, it would allow development teams to focus on their problem domain and areas of expertise.

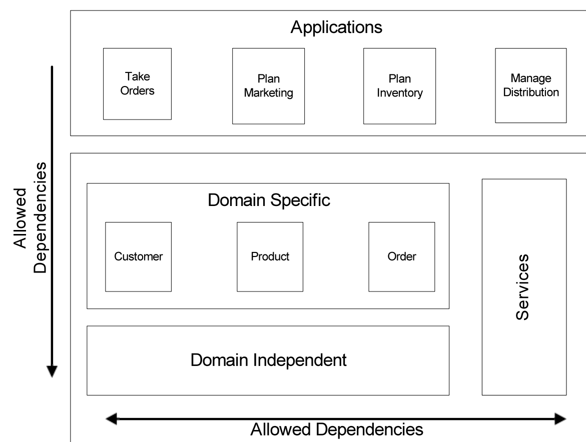

Fig. 2. Example Layered Architecture

A layered architecture such as that shown Fig. 2, governed by rules, minimizes the development of complex dependencies and allows for simplified configuration and assembly strategies. Each layer in L.L.Bean's strategy has well-defined responsibility. Organized in a hierarchy of generality/specificity, each layer is governed by the principle that members of a given layer can only depend on other members in the same level, or in layers below it. Each layer, or smaller subset within a layer, is assembled in a cohesive unit, often referred to

as a program library or subsystem. In L.L.Bean's case, these cohesive units are Java jar files. This approach produces a set of independent consumable components that are not coupled by complex dependencies, and creates a solid foundation for the producer/consumer paradigm.

STEP 3: Validating the Architecture

The next step was prototyping and transforming the current state into the intended architecture.

First, subsystem containers were created at the root level of the model for each of the high-level organizational layers defined in the architecture; initially these were the familiar domain-independent, domain-specific, and application-specific layers. The next step was to examine each jar file, and move them into one of the defined layers.

Here, the benefits of L.L.Bean's early package naming approach became clear; behavior and responsibility were built into package naming clarifying the appropriate layer in most cases. In a small set of specialized cases, developers with in-depth knowledge of the code were consulted. Here again, the well defined and documented layered architecture facilitated communication with software engineers and simplified the process of deciding on the appropriate layer. With the architecture already well understood, it took only two working days to successfully transform the initial model into the intended architecture, simply by prototypically moving Java classes to their appropriate package according to both their generality/specificity and their behavior/responsibility. At the end of that time, we were surprised to observe that nearly all undesirable dependencies at the top level had been eliminated. The DSM shown in Fig. 3 captures the state of the prototyping model near the end of the two-day session. The lower triangular nature of the DSM shows the absence of any top-level cycle.

$root		application 1	domain 2	commons 3
application	1	26%		
domain	2	469	35%	
commons	3	1418	2360	38%

Fig. 3. DSM of layered architecture, no top-level cycles present

STEP 4: Identifying Sources of Architecture Violation

Three key packaging anti-patterns were identified that were at the core of the interdependency entanglement. This is illustrated by the following examples (note: the arrows in the figures show "uses" dependencies [2]):

Misplaced Common Types: Many types (i.e. Value Object Pattern, Data Transfer Object Pattern, etc.) were packaged at the same hierarchical level as the session layer (Session Façade) to which they related. This approach widely scattered dependencies creating a high degree of complexity, and a high number of cyclic dependencies. This issue was resolved as shown in Fig. 4, by moving many of these common types from their current package to an appropriate lower layer. This resolved an astounding 75% of existing circular dependencies.

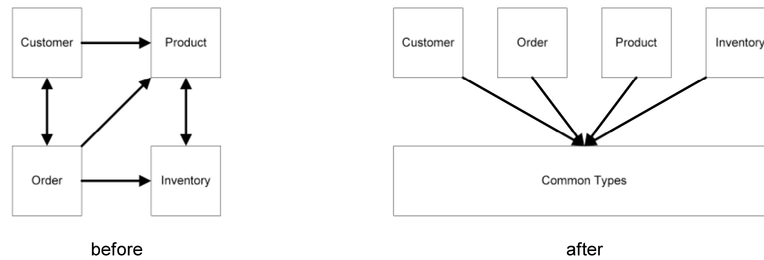

Fig. 4. Repackaging Common Types

Misplaced Inheritable Concrete Class: When a concrete class is created by extending an abstract class, it is packaged according to its behavior. However, when a new concrete class is created by extending this class it then creates

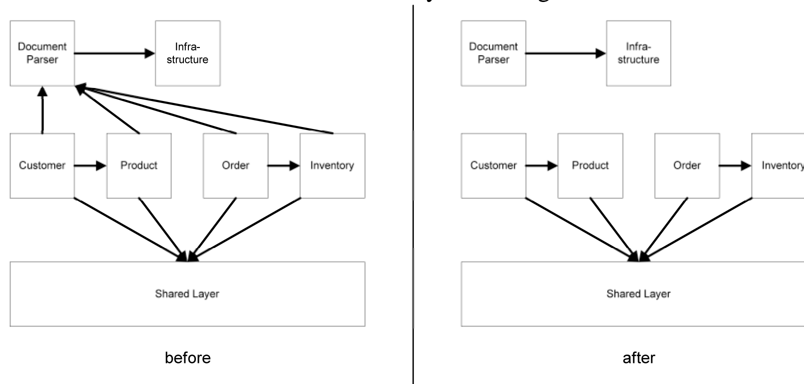

Fig. 5. Moving Descendants into Ancestor's Package

a coupling between components that were normally expected to be independent. Moving the concrete class to the shared layer where its parent was located solved the problem as shown in Fig. 5. This also supports the notion that concrete classes should not be extended [19]. Instead, whenever the need arises to extend a concrete class, the code should be refactored to create a new abstract class, which is then used as a base class for the different concrete classes. This problem also illustrates that as code bases evolve it is necessary to continuously analyze and restructure the code.

Catchall Subsystems: The behavior/responsibility focus of the early package naming approach produced a subsystem dedicated to IT infrastructure. This became a disproportionately large "catch-all" subsystem that represented a broad array of mostly unrelated concepts. It also included a small set of highly used classes supporting L.L.Bean exception handling. It generated a large number of dependencies making it brittle and costly to maintain. To manage this problem, the exception classes were moved into their own subsystem and the remaining parts were reorganized into multiple subsystems of related concepts as shown in Fig. 6. This problem also illustrates how analyzing usage can be used to identify and group reusable assets.

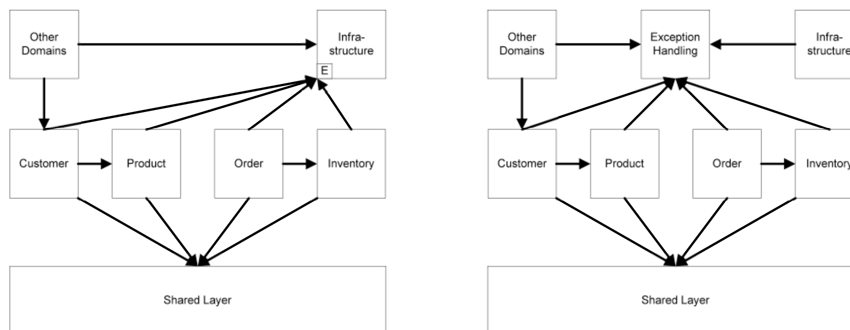

Fig. 6. Breaking up Catchall Subsystems

STEP 5: Refactoring the Code

With the right tools and a well-defined architecture, prototyping packaging change was relatively simple. Fortunately, L.L.Bean's code restructuring effort was primarily limited to changing packages (e.g. Java package and import statements), and did not affect code at a method level.

The initial goal was to take a "big bang" approach by re-architecting the entire system at once. The large commitment to cut over to a consolidated and restructured code base in one step proved costly and did not mesh with the various iterative development cycles across development teams. Instead, an incremental approach is being used where new development and refactored code are packaged according to the principles of the layered architecture. Development teams and configuration engineers use DSM models to analyze static dependencies as well as to prototype new packages and package changes during the normal development cycle. This has proven to be a sustainable approach for continuous improvement of the code base.

A few key standards in place at L.L.Bean have helped facilitate this approach. First, using a standard IDE, developers can easily organize import statements such that fully qualified class names are not embedded within methods. Second, wildcards are not allowed in import statements. Automated

configuration management processes enforce standards[8]. As a result, there existed a relatively consistent state of package and import statements. The last important standard here is unit tests. L.L.Bean standards require a unit tests for every class, and most software development teams employ test-first development methodologies[9]. After restructuring, each unit test is exercised, providing an immediate window into the impact of the changes.

4 Evolving & Improving the Architecture

A software engineering process was needed that would prevent architectural drift and the need for large scale refactoring in the future. A set of rules were created that could be applied to any of L.L.Bean's DSM models, and visibility of maintenance processes was increased.

4.1 Rules

Design rules are the cornerstone of architecture management. L.L.Bean developed a simple set of architecture enforcement rules. These rules enforce a layered architecture and essentially state that members of a given layer may only depend on other members in the same level, or in layers below it.

Rules also help software engineers identify reuse candidates. When violations occur, the nature of the dependencies and the specific behavior of the Java code are closely analyzed. If there are multiple dependencies on a single resource that break an allowed dependency rule, then the target resource is a candidate for repackaging. The analysis is followed by a discussion with the appropriate project manager, architect or software developer.

Governance reduces software maintenance cost, improves quality, and increases agility, by enabling architectural remediation during ongoing development.

4.2 Maintaining Visibility

Architectural governance offers several benefits. A DSM model provides consistent visibility and supports on-going communication between development teams, configuration engineers and project leaders. It also facilitates change impact analysis. L.L.Bean creates DSM models at different organizational levels from application-specific to a comprehensive "master model". Application modeling during normal development cycles enables configuration engineers to determine what dependencies are missing, what dependencies are using an outdated version, whether unused component

---

[8] http://pmd.sourceforge.net/
[9] http://ww.junit.org/index.htm

libraries that are included should be removed, and to report on changes between development iterations. As of the writing of this paper, this analysis and ongoing communication have resulted in a 10% reduction in the number of Java jar files being versioned and dramatically improved understanding about the true dependencies of the applications and the jars they consume.

L.L.Bean creates multiple dependency structure matrices for various purposes. One is designated as “master model”, which maintains visibility to the current state of the overall architecture as new development continually introduces new dependencies. The master model is integrated with and updated through automated configuration management processes, and is designed to support dependency governance. Each time a new version of a software element is created, the master model is updated, design rules are applied and when violations are detected, they are reported to the appropriate stakeholders (project managers, configuration managers, architects, and reuse engineers) who determine whether each violation is a programming error or reflect change in architectural intent. Violations also “break the build”, forcing software development teams to correct problems before the new version is permitted to move forward in its lifecycle.

For additional analysis, L.L.Bean created an analysis and configuration tool leveraging DSM metadata designed to address two long-standing questions. First, given a class, which jar file contains that class? Second, given a jar file which other jar files does it depend upon? This information is then stored in query optimized database tables that are refreshed with each update. The query operation is exposed as a service for use with automated configuration management processes. For example, dependent components defined in build scripts are updated with new dependencies, including the order of those dependencies as code is modified during the course of normal development.

5 Lessons Learned

L.L.Bean’s experience has been successful in demonstrating the value of using an architecture dependency analysis based approach to improve software agility. There were several lessons learned along the way that should be kept in mind.

While good tool support is essential, without good development practices, use of established coding standards, active configuration management, and consistent unit testing, tool use would be much more time-consuming and less effective.

Dependency information must be visible to be managed, but that alone is not enough to reduce maintenance costs and effort. It must be supported by the ability to try “what if” scenarios and creating prototypes to explore change impact.

Business drives requirements and ultimately the need to be agile. The “big bang” approach wasn’t viable in an environment with multiple development

teams that had various and, often, conflicting development cycles, each with different business drivers. Moreover, it became evident that the difficult and costly attempt at using a “big bang” approach was not necessary. Following the incremental approach described in Section 3 development teams remain agile and refactor to use the layered architecture as part of their normal development cycles.

Beyond consolidating and repackaging code, there are often implications with respect to external (non-Java) component coupling. In some cases fully qualified Java packages were specified in scripts and property files. In other cases, Java classes referenced external components, which presented issues when consolidating code. The lesson learned was that change impact is often greater than what is originally estimated.

6 Limitations and Future Work

While it is believed that DSMs can be applied to systems implemented in other languages and databases, the L.L.Bean experience is only with Java based software. Therefore, the results of this experience may not generalize to other types of systems.

While the experience reported here has made a substantial impact on L.L. Bean’s ability to maintain its code-base, we believe this is just one of many benefits that architecture analysis can provide. This report validates the DSM’s support for evolvability, we are continuing to explore the potential for extracting other relationships from code, in particular run-time relationships, which can be used to identify the existence of Component and Connector (run-time) architectural styles and the application of the DSM to support analysis of a variety of run-time quality attributes.

7 Summary

The key to L.L.Bean’s code restructuring success was increasing visibility of both system’s architecture and the process. L.L.Bean has found that increasing the visibility of architecture greatly reduces architectural drift as the system evolves and at the same time reduces ongoing maintenance costs. Architectural visibility provides guidance for large-scale refactoring.

L.L.Bean also discovered that changing the structure of the system can sometimes be achieved without substantial code modifications and that large scale re-organization is a complex process that, when done with proper tool support and in a disciplined software development environment, can be effective.

The results of this experience demonstrate that architecture-based analysis can improve the productivity of software development. It is hoped that future research and practice will produce continued advancement in architectural support for improved software quality.

Acknowledgements. Sincere thanks to Doug Leland of L.L.Bean for his skillful mentorship and guidance, to David Smith and Tom Gould of L.L.Bean for their many insightful contributions to the Reuse Team and code dependency analysis.

References

1. Buschman, F., Meunier, R., Rohnert, H., Sommerlad, P. and Stal, M. *Pattern-Oriented Software Architecture: A System of Patterns*, Wiley, 1996.
2. Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., and Stafford, J. *Documenting Software Architectures: Views and Beyond*, Addison Wesley, 2003.
3. Ducasse, S., Ponisio, L., and Lanza, M. “Butterflies: A Visual Approach to Characterize Packages”. In *Proceedings of the 11th International Software Metrics Symposium (METRICS ’05)*, Como, Italy, September 2005.
4. Fowler, M., Beck, K., Brant, J., Opdyke, W., and Roberts, D., *Refactoring: Improving the Design of Existing Code*, Addison Wesley, 1999.
5. Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns*, Addison Wesley, 1995.
6. Hautus, E.. “Improving Java Software Through Package Structure Analysis”. *Proceedings of the 6th IASTED International Conference Software Engineering and Applications (SEA 2002)*, Cambridge, Massachusetts, September *2002*.
7. Jacobson, I., Booch, G., and Rumbaugh, J. *The Unified Software Development Process*. Addison Wesley, 1999.
8. Jacobson, I., Griss, M., and Jonsson, P. *Software Reuse: Architecture, Process and Organization for Business Success*. Addison Wesley, 1997.
9. Kruchten, P. “Architectural Blueprints: The “4+1” View Model of Software Architecture”. *IEEE Software*, 12(6):42-50, November 1995.
10. Melton, H. and Tempero, E. An Empirical Study of Cycles among Classes in Java, Research, Report UoA-SE-2006-1. Department of Computer Science, University of Auckland, Auckland, New Zealand, 2006.
11. Melton, H and Tempero, E. “The CRSS Metric for Package Design Quality”. *Proceedings of the thirtieth Australasian conference on Computer science*, Ballarat, Victoria, Australia, Pages 201 – 210, 2007 .
12. Perry, D. and Wolf, A., ``Foundations for the Study of Software Architecture''. *ACM SIGSOFT Software Engineering Notes*, 17:4 (October 1992
13. Poulin, J. S. *Measuring Software Reuse*. Addison Wesley, 1997.
14. Poulin, J. S. “Measurements and Metrics for Software Components”. *Component-Based Software Engineering: Putting the Pieces Together*, Heineman, G. T. and Councill, W. T. (Eds), Pages 435-452, Addison Wesley, 2001.
15. Sangal, N. and Waldman, F. “Dependency Models to Manage Software Architecture”. *The Journal of Defense Software Engineering*, November 2005.
16. Sangal, N., Jordan, E., Sinha, V. and Jackson, D. “Using Dependency Models to Manage Complex Software Architecture”. *Proceedings of the 20th annual ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications*, Pages 167-176, San Diego, California, October 2005.
17. Stafford, J. Richardson, D., and Wolf, A., “Architecture-level dependence analysis in support of software maintenance”. *Proceedings of the Third International Conference on Software Architecture*, Pages 129-132, Orlando, Florida, 1998.

18. Stafford, J. and Wolf, A., “Software Architecture” in *Component-Based Software Engineering: Putting the Pieces together*, Heineman, G. T. and Councill, W. T. (Eds), Pages 371-388, Addison Wesley, 2001.
19. Lieherherr, K. J., Holland, I.M. AND Riel, A.J. 1988. Object-oriented programming: an objective sense of style. In OOPSLA’88 Conference Proceedings (San Diego, California, Sept. 25–30). ACM SIGPLAN Not. 23, 11 (Nov.) 323–334.