Large Scale Refactoring through Architecture Visibility

Carl Hinsman
L.L.Bean
Freeport, Maine USA
chinsman@llbean.com

Neeraj Sangal
Lattix, Inc.
Andover, Massachusetts USA
neeraj.sangal@lattix.com

Judith Stafford
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 seven years L.L.Bean's Java software code-base had become more difficult to evolve. A corporate-wide 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 current Java code-base and adoption of process improvements that support an architecture-based agile approach to development and maintenance.

1. Introduction

This paper reports on L.L.Bean, Inc.'s experience infusing new life into its evolving software systems. 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, and 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 7 years, has suffered the typical problems associated with rapid evolution such as increased complexity and difficulty in building and maintaining the system. While the company's Java software development processes have long included a number of good practices and coding rules to help avoid these issues, the speed of evolution made development increasingly difficult and costly. Maintenance and evolution of the software infrastructure were recognized as concerns by IS management. 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 Java code base 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. [3] 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 that make changing the code easy such as generating 'getters/setters' and 'constructors' and performing operations such as 'rename' etc. L.L.Bean viewed the concepts as a good approach for solving their problem but the size and complexity of the existing code base was such that attempts to apply refactoring in the normal way, which is a series of localized modifications, failed. L.L.Bean recognized the need to approach solving the problem from a global perspective.

Because the 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 an abstracted view appropriate for communication among stakeholders [17]. As such, it seemed an architectural approach to "refactoring" L.L.Bean's Java code base seemed appropriate. Initial efforts in process improvement at L.L.Bean involved evaluation of several tools; the tools were evaluated for

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

their potential effectiveness to help remove the dependencies, reduce the rate of architectural decay in the future, and to support communication among stakeholders of the system. Of the tools that were evaluated the Lattix Architecture Management System[2] easily rose above competitive approaches. The Lattix system provides an architectural view of source code dependencies, creates an accurate blueprint of the system, and supports analysis of the system structure and as such can be used to direct activities related to improving reliability and maintainability [15]. Lattix's LDM tool supports exposure of an application's software architecture and allows organizations to quickly produce a module view [1] of the application in order to identify interdependencies among code modules. The complimentary Lattix LDC tool supports continuous monitoring and impact analysis while the code base evolves enabling an organization to avoid architectural drift [11]. The Lattix system was adopted by L.L.Bean and the process of "architecture refactoring" began.

L.L.Bean's strategic approach to refactoring required few code changes but rather a series of significant structural modifications. Unlike current approaches to refactoring, L.L.Bean's method is driven by overall visibility of the software architecture and includes five steps: define the problem, visualize the current architecture, model the desired architecture in terms of current elements, automate making large scale changes to code organization, and continuous governance of the architecture

This agile 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 agile 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 every growing business demands of this increasingly popular brand.

We begin our report with a description of the problem facing L.L.Bean's Java software developers. This is followed by a recounting of research toward identifying a viable solution and the basis for the decision to apply the Lattix Architecture Management System. We then provide an overview of the Lattix system 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 lessoned learned through this experience and describe our ongoing efforts.

[2] http://www.lattix.com/products/products.php

2. Background

L.L.Bean has more than 20 retail stores in 10 states, as well as a mail order and ecommerce channels. As a result, L.L.Bean needs a high degree of automation for inventory control, order fulfillment, and for managing suppliers, with a focus on quality and consistency across channels.

A significant part of the information technology infrastructure is written in Java and runs on servers running either Windows or Unix based Operating Systems. The system has gone through a rapid evolution over the last 7 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 has been developing software using Java for nearly seven years, primarily following the model-view-controller pattern. Development teams normally consist of ten or fewer developers grouped by business domains such as product, order capture, human resources, and IS infrastructure. One group has the responsibility for package naming for all software development. Package naming intuitively represents behavior and/or responsibility of the Java class. This approach was intended to facilitate reuse and help avoid duplication of effort by increasing visibility. In that regard, the approach has been successful. However, it is limited in scope and does not address interdependencies among modules.

The current system has more than one million lines of Java code that is roughly organized by business domain and assembled into more than a 100 jar files. In turn, the actual Java code is organized into nearly 1,000 Java packages and more than 3,000 Java classes.

Over time, normal code evolution created a complex entanglement of interdependencies, increasing software development and maintenance costs, and decreasing reuse potential. The number of Java developers and teams increased. Additionally, certain application decisions resulted in multiple diverging Java code bases, increasing complexity and management costs by a factor of three. Software development and configuration management continued for a short period of time before reaching a critical mass. An attempt was made at merging the Java code bases back into a single stream, which failed because the totality of the entanglements was simply too complex.

Tactical approaches were informally attempted by several developers, none of which succeeded. These were essentially "find and fix" scenarios, one dependency entanglement at a time, without considering the impacts of change holistically. Interdependency problems were primarily identified by configuration managers when attempting to compile code and assemble applications for deployment. The process was slow at best, and correcting one problem would often create a different set of problems. Moreover, it did not offer a plan for preventing entanglements from recurring.

Business needs continue to drive new development, and interdependency entanglements continue to grow. Software development and configuration management costs increase in stride. IS management created a task force to analyze the problem and recommend a solution. In their conclusions, the task force emphasized the economic significance of reuse. The primary recommendation of the task force was to create a small team of software engineers focused on creating and implementing a comprehensive packaging strategy. This newly formed 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 create a single Java code base and transition the development life-cycle to a producer/consumer paradigm.
- Software architecture was not visible and no person or group in the organization maintained intellectual control over the software
- There was no mechanism to govern dependency evolution

A solution to support immediate needs and be permanently adopted by the organization must not only support refactoring the software architecture but also help prevent this problem from recurring.

3. Strategy for Restructuring

The team[3] created as a result of the task force chose a strategic approach. At the outset, two tasks were undertaken simultaneously. 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 circular dependencies and governing packaging of future software development, the resulting packaging structure needed to accomplish other goals. First, provide a single Java 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, create a dynamic view of 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 the Java development community and increase communication between development teams.

The hope of the multi-path approach was to create a packaging strategy that defined architectural intent irrespective of the actual problem, while simultaneously challenging the validity of that strategy using real details from the problem space. In other words, create a strategy that proactively manages static dependencies, and prove that it works.

Trust was paramount in the minds of those defining the new strategy. The business, and more specifically the development teams supporting the various business domains, would not entertain undertaking a restructuring effort of any kind without having solid evidence of the soundness of the strategy. The team understood that the way to earn this trust was through research, communication and prototyping.

3.1. Research

3.1.1. Literature Search

As a starting point, the team sought articles and academic papers primarily through the internet, using a wide range of thematic keywords. 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

[3] The team is L.L.Bean's Reuse Team, supporting and consulting all things reuse within the IS organization.

team **[2][5][9][10][13][14][16]**. At times, the team found it challenging to digest the abundant and often overly abstract information.

The timing of this effort was fortuitous in that another effort was underway at L.L.Bean to create a strategy for reuse metrics. Considerable research had already been conducted in this problem space, and it was clear that overlap existed **[7][12][13]**. Much of this research suggested that code packaging in domain-oriented software can promote reuse and facilitate metrics. Exploring these metrics and tools to support them provided additional focus and served as the starting point for the strategy.

3.1.2. 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, without spending valuable time evaluating the various tools. Open-source tools were chosen as a starting point for obvious reasons. These included Dependency Finder[4] and JDepend[5] (output visualized through Graphviz[6]), among others[7][8]. Each of these tools were useful in understanding the state of dependencies, but none of them offered the comprehensive, easy to understand, view needed nor did they provide any support for restructuring or communication among stakeholders; it was crucial to success to be able to clearly communicate to IS leaders, architects, engineers, and developers alike.

Graphing the analysis was most challenging, requiring the use of a combination of tools to produce illustrations of problem spaces. One solution was to use JDepend to analyze the Java code base, which produces XML output. This output was then transformed via XSLT into the dot format required by Graphviz for generating directed graphs. While not particularly difficult, the process was extremely computationally intensive; there was a limit to the amount of code that could be analyzed collectively in this fashion. *Figure 1* shows an example (a small portion of a much larger directed graph) produced using this approach. This view of dependencies was nearly incomprehensible and of little practical use in efforts to manage the code.

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

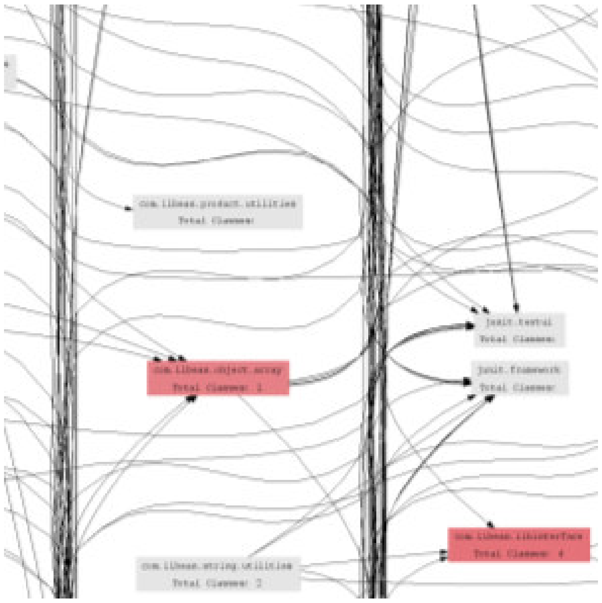


Figure 1: Excerpt from Generated Dependence Graph

A primary issue with the early attempts at analysis was that the available tools only support analysis of the current state of the system. None of the tools allowed for "what if" scenario prototyping, nor did they offer mechanisms for applying architectural rules. For example, after the aforementioned modeling approach was conducted and a problem identified, it would be necessary to determine one or more potential solutions, code and compile those solutions, then perform the analysis and modeling steps again to see the real impact. Given the chaotic state of the interdependency entanglement, even identifying problems through something like a directed graph was nearly impossible and far too time-intensive to be of use. To visualize the impact see *Figure 1*, which contains an excerpt from the generated dependence graph.

The shortcomings of these tools have been researched and discussed by others, and there is evidence of their limited effectiveness. L.L.Bean's research identified only the Lattix matrix-based dependence analysis tool as being promising and, through experience, found it to be highly effective in that it offered a comprehensive easy to understand interface as well as mechanisms for prototyping and applying software architecture rules. The Lattix Architecture Management System is capable of modeling (as a matrix) dependencies for several software languages and databases. The L.L.Bean experience exercised only the system's Java capabilities, however.

3.2. 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 was pointed at the complete base of Java jar files. Within minutes, the tool created a “dependency structure matrix” (DSM)[9] that illustrated all the static dependencies in the Java 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 we encountered with directed graphs. Furthermore, Lattix allows users to edit that structure to try create what-if scenarios and to specify design rules to formalize, communicate and enforce the architecture. This means that an alternate structure which represents the desired design intent can be maintained even if the code structure does not immediately reflect it. Once an architecture is specified Lattix allows the architecture to be monitored in batch mode and key stakeholders are notified of the results.

The Lattix DSM offers two powerful aides to understanding and improving the architecture. They are algorithms purposed for re-ordering of subsystems, and for identifying dependencies and opportunities for better layering. The first is “DSM Partitioning”, which yields virtual partitions containing subsystems that can form a single unit. The second is “Provider Proximity Partitioning”, which attempts to bring the provider subsystem closer to the subsystems that depend on them. L.L.Bean’s Lattix models were created maintaining each Jar file as a root subsystem. By doing so, the partitioning clearly illustrated independently deployable cohesive jar files; at the same time it illustrated which jar files were entangled with which other jar files. This simple step led L.L.Bean to create an additional analysis and configuration tool based on the Lattix model, discussed later in this paper.

Provider Proximity Partitioning has a subtle yet powerful benefit. In organizing provider subsystems close to those upon which it has dependencies, it forces the subsystems with the greatest dependencies to the bottom of the matrix revealing a layered module view **[1]** of the architecture. Reading the list of subsystems from the bottom up defines a natural build order for the jar files. The order in which the jars should be built has long been an issue for our configuration managers. It also illustrates how a well defined software architecture helps configuration management.

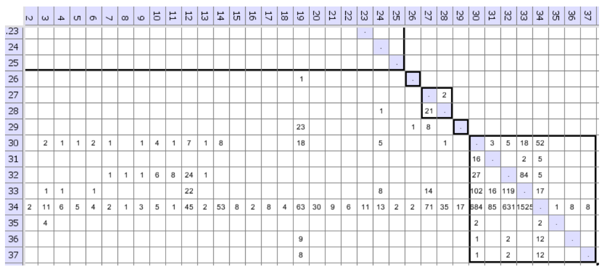
3.3. Iterative Strategy Development

The team consciously approached strategy development in an iterative fashion on two fronts. Each version of the defined architectural intent was used to create a model using the existing Java code base, to identify what worked and what didn’t. The tools used for this prototyping are described later in this paper. The results of the modeling and the updated strategy documentation were discussed with stakeholders to gather their input. There were nearly a dozen iterations. This approach maintained a high degree of communication and visibility in the strategy development process, which facilitated adoption and understanding.

Defining a software architecture is more difficult than developing the software itself, but there are several practices that work well for both. What software architecture is the “right” one? Is there even such a thing? As in most software development, L.L.Bean’s strategy development was ultimately an exercise in compromise, influenced by IS leaders, architects, and engineers.

3.4. The Initial State

The first step in the architecture-based refactoring process was to understand the current state. An initial DSM was created by loading all jars into a Lattix LDM. The subsystems in the DSM were organized to reflect a layered structure. 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 *Figure 2* illustrates a highly complex and cyclic dependency grouping.



	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37
.23																						.														
24																							.													
25																								.												
26																		1							.											
27																										.	2									
28																							1			21	.									
29																		23							1	8		.								
30		2	1	1	2	1		1	4	1	7	1	8					18					5				1		.	3	5	18	52			
31																													16	.		2	5			
32						1	1	1	6	8	24	1																	27		.	84	5			
33		1	1		1						22												8			14			102	16	119	.	17			
34	2	11	6	5	4	2	1	3	5	1	45	2	53	8	2	8	4	63	30	9	6	11	13	2	2	71	35	17	684	85	631	1525	.	1	8	8
35		4																											2				2	.		
36																		9											1		2		12		.	
37																		8											1		2		12			.

Figure 2: Using Lattix LDM to Reveal Layering.

A DSM is a square matrix with each subsystem being represented by a row and column. *Figure 2* shows

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

the results of DSM partitioning which groups subsystems together in layers. 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 grouping shows us that subsystems 30..37 are circularly connected. The rows and columns are numbered. 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 a strength of '119' and '631', respectively. By clicking on an individual cell you can see the complete list of dependencies in an attached display panel.

3.5. Modeling the Desired Architecture

L.L.Bean's Java packaging strategy is a layered architecture that leverages the earlier successes of the package naming approach, adding high-level organizational constructs and rules to govern static dependencies at the appropriate level.

Models of typical software applications are commonly divided into three classes describing the relative specificity of the components that make up the given application: 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. This concept is at the core of L.L.Bean's Java code restructuring strategy.

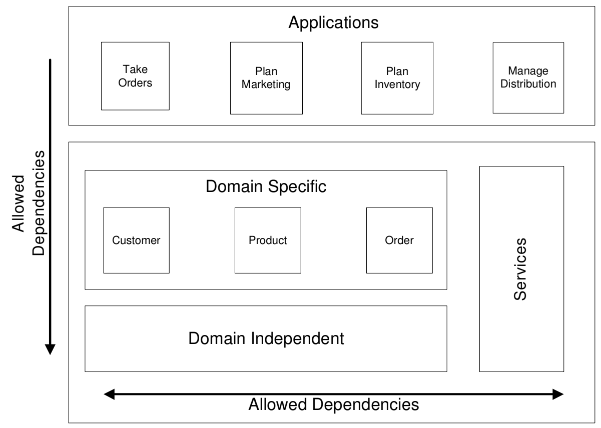


Figure 3: Example Layered Architecture

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. Communication is crucial to success, and the layered architecture facilitated discussions with IS leaders, project leaders, domain architects, configuration engineers, and software developers. For example, a chief architect was hired after the strategy was developed, and it took less than one hour to fully communicate the strategy in all its facets to the complete satisfaction of the new chief.

When well described with concise documentation, a layered packaging architecture can promote healthy change-tolerant evolution. The larger and more diverse an organization, the greater the difficulty in communicating and coordinating software development efforts. If development teams are organized to correlate with the architectural layers then the software development process can be appropriately tailored to simplify process management and maximize effectiveness. Clearly communicated architectural intent will provide leaders and developers with the information needed to develop software components, services and applications without creating undesirable dependencies. More importantly, it will allow development teams to focus on their problem domain and areas of expertise while adhering to the packaging architecture.

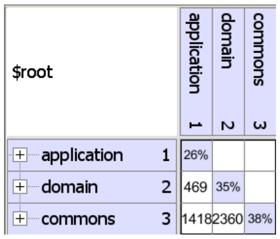
A layered packaging architecture such as that shown in *Figure 3* governed by rules minimizes the development of complex dependencies and allows for simplified configuration and assembly strategies. Each layer in L.L.Bean's strategy has well-defined responsibility. Organized in a hierarchy of generality/specificity, each layer is governed by the principle that members of a given layer can only depend on other members that are in the same level, or in layers below it. Each layer, or smaller subset within a layer, is assembled in a cohesive unit, often called 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 or cyclic dependencies, and creates a solid foundation for the producer/consumer paradigm.

3.6. Validating the Architecture

With the well defined and documented architecture and a powerful dependency analysis tool in hand, the next step was to transform the current state into the intended packaging architecture. Using the base model described above, prototyping change began.

First, subsystem containers were created at the root level of the model for each of the high-level organizational layers defined in the packaging 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, L.L.Bean's early package naming approach was beneficial; behavior and responsibility were built into package naming clarifying the appropriate layer in most cases. In a small set of specialized cases, resources with in-depth knowledge of the code were required. Here again, the well defined layered architecture facilitated communication with software engineers and simplified the process of deciding on the appropriate layer. In total, it took two working days to successfully transform the initial model into the intended packaging architecture, simply by changing Java package statements such that classes were appropriately layered according to both their generality/specificity and their behavior/responsibility. At the end of that time, all undesirable dependencies had been eliminated. The DSM in *Figure 4* captures the state of the prototyping model near the end of the two day session, emphasizing 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%

Figure 4: DSM of layered architecture, no top-level cycles present

3.7. Patterns

A few key packaging patterns were identified that were at the core of the interdependency entanglement. Upon deeper examination of the dependencies, it was discovered that a large number of these entanglements were caused by a small number of what were considered "anti-patterns" related to the code organization. Each example includes an illustration, which represents "uses" dependencies and not necessarily architectural layers:

- Data 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 cycles. L.L.Bean solved this problem, as shown in Figure 7 by extracting all abstract data types from their existing package locations and moving them to the bottom layer in their package hierarchy. This concept made sense in terms of a layered architecture, placing the most generalized layers at the bottom and the most specific layers at the top. An astounding 75% of existing circular dependencies were caused by an abstract class being packaged in the wrong layer.

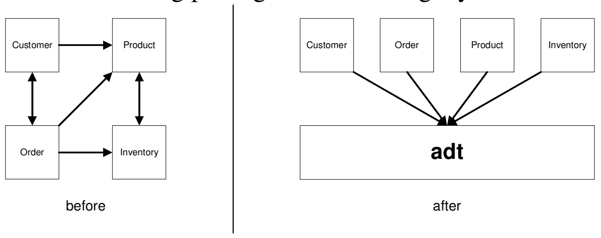


Figure 5: The Affect of Repackaging ADTs

- A single concrete class designed for inheritance was packaged according to its (abstract) nature. Classes extending it added specific behavior and were packaged according to that behavior, which was inherently different from their ancestor. This occurred repeatedly eventually creating a significant number of unrelated interdependent subsystems. L.L.Bean solved this problem by repackaging the subclasses within the same subsystem as their ancestor class as shown in Figure 6. This problem illustrates that as code bases evolve it is necessary to continuously analyze and restructure the code.

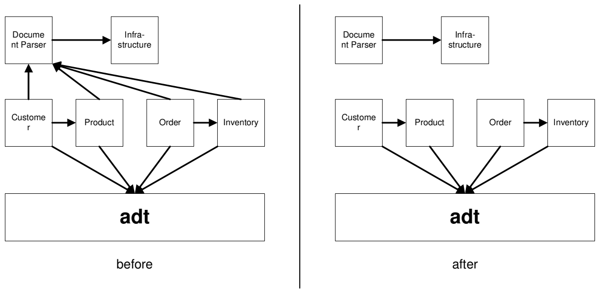


Figure 6: Moving descendents to ancestor's package

- The behavior/responsibility focus of the early package naming approach produced a subsystem dedicated to IS infrastructure. This subsystem was disproportionately large, and included a small set of highly used classes supporting L.L.Bean's exception handling standards. The subsystem represented a broad array of mostly unrelated concepts, and can best be described as a change propagator. It generated a huge number of dependencies making it brittle and costly to maintain. To manage this problem the exception classes were harvested into their own subsystem, and the remaining parts were reorganized into multiple subsystems of related concepts as shown in Figure 7. This problem also illustrates a clear way to identify reusable assets, simply by analyzing usage.

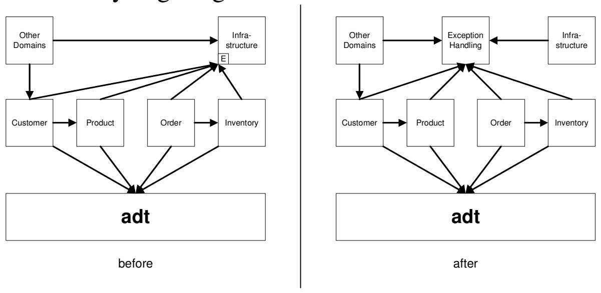


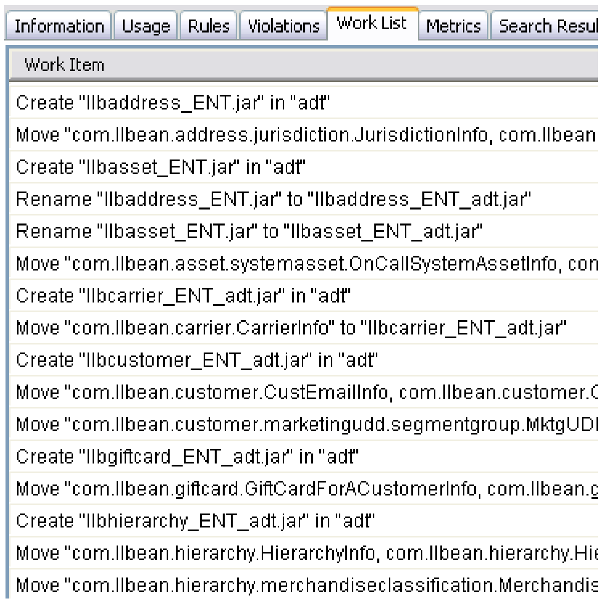
Figure 7: Harvesting a change propagator

The dependency entanglements described in the second and third patterns are similar in how L.L.Bean resolved them, but they differ in their causes. One is an "inherits" dependency, while the other is a combination of "constructs" and "uses" dependencies.

3.8. Refactoring the Code

With the right tools and a well defined software architecture, prototyping packaging change is relatively simple. Actually updating code is a somewhat different story. Fortunately, L.L.Bean's Java code restructuring effort was limited to changing packages (e.g. Java package and import statements), and did not affect code at a function level.

L.L.Bean was able to automate physical repackaging by leveraging another useful feature in Lattix, its work list. The work list keeps track of every step taken during prototyping. Given the iterative nature of the strategy development process, any given class could be moved multiple times. L.L.Bean created a tool (in Java) that processes the work list, building a point to point map reflecting where (which package) a given class started at the beginning of the prototyping, and where it ended up. In batch fashion, the tool updates version controlled source code changing the Java package and import statements appropriately. This tool is called the Latt-o-Mattix.



```
Work Item
Create "llbaddress_ENT.jar" in "adt"
Move "com.llbean.address.jurisdiction.JurisdictionInfo, com.llbean
Create "llbasset_ENT.jar" in "adt"
Rename "llbaddress_ENT.jar" to "llbaddress_ENT_adt.jar"
Rename "llbasset_ENT.jar" to "llbasset_ENT_adt.jar"
Move "com.llbean.asset.systemasset.OnCallSystemAssetInfo, con
Create "llbcarrier_ENT_adt.jar" in "adt"
Move "com.llbean.carrier.CarrierInfo" to "llbcarrier_ENT_adt.jar"
Create "llbcustomer_ENT_adt.jar" in "adt"
Move "com.llbean.customer.CustEmailInfo, com.llbean.customer.C
Move "com.llbean.customer.marketingudd.segmentgroup.MktgUDI
Create "llbgiftcard_ENT_adt.jar" in "adt"
Move "com.llbean.giftcard.GiftCardForACustomerInfo, com.llbean.g
Create "llbhierarchy_ENT_adt.jar" in "adt"
Move "com.llbean.hierarchy.HierarchyInfo, com.llbean.hierarchy.Hie
Move "com.llbean.hierarchy.merchandiseclassification.Merchandis
```

Figure 8: A Lattix work list example.

Figure 8 shows a snippet from a Lattix work list. This grouping captured the creation of several Lattix "subsystems" during the ADT move iteration phase. In each ADT set, a "container" subsystem is created, then each ADT that is being repackaged is "moved" from its existing package location to the newly created container subsystem.

A few key standards in place at L.L.Bean helped enable this approach. First, using a standard IDE, developers are required to organize import statements such that fully qualified class names are not embedded within methods. Second, standards disallow the use of wildcards in import statements. The use of wildcards is discouraged because all the members of the package are imported into the class path, not just the necessary dependencies. It is also discouraged for clarity and maintainability concerns. Both standards are enforced by automated configuration management processes[10]. Through these standards there existed a relatively consistent state of package and import statements as a foundation for the automated restructuring tool. The last important standard here is unit tests. L.L.Bean standards require a unit tests for every class, and several software development teams employ test-first development methodologies[11]. After restructuring, each unit test is exercised. This provided an immediate window into the impact of the automated changes.

There are risks to this approach, although greatly mitigated by the use of tools. Code written before the implementation of standards could be missed by the

[10] http://pmd.sourceforge.net/
[11] http://www.junit.org/index.htm

automated tool, and inadequate unit tests could easily result in false positives. It was assumed that the automated restructuring tool would not provide 100% coverage, nor would it be able to provide 100% certainty of success. There simply had to be a degree of human involvement in the restructuring.

L.L.Bean's restructuring was a "Big Bang" approach vs. an incremental one. There is an incremental aspect to the restructuring, however. Remember that a goal of L.L.Bean's restructuring was to provide a single Java code base to support a producer/consumer paradigm. The Jar files resulting from the restructuring are simply a new version of existing Java code. Software development teams gradually become consumers of the new versions of the Java code (the new set of Jar files) as L.L.Bean metamorphoses to the producer/consumer paradigm. It was understood that there would be two Java code bases for a period of time.

4. Maintaining the Architecture

Although this exercise was less painful than initially anticipated, L.L.Bean was anxious to avoid having to repeat it. Three steps were taken to prevent architectural drift. First, a set of rules were created, second, a "Master Matrix" was created, and third, visibility of maintenance efforts were increased. Each of these is elaborated upon in the following sections.

4.1. Rules

Design rules are the cornerstone of software architecture management. L.L.Bean developed a simple set of software architecture enforcement rules. These rules 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 code are analyzed in detail. If there are several dependencies to a single resource, and each of those dependencies break a permissibility rule, then the target resource is most likely packaged incorrectly. The analysis is followed by a discussion with the appropriate project manager, architect or software developer.

Governance reduces software maintenance cost, improves quality, and makes software development teams more agile, enabling more rapid development. L.L.Bean's rules prevent problematic dependency evolution, which saves the organization money by not having to solve this kind of problem in the future.

4.2. Master Matrix

L.L.Bean's "master matrix" is central to maintaining visibility to the current state of the software architecture as new development continually introduces new dependencies. While L.L.Bean creates multiple dependency structure matrices for various purposes there is one matrix hooked to and updated through automated configuration management processes.

Each time new software development creates a new version of a software element, the master matrix is updated, design rules are applied and violations are reported to the appropriate stakeholders (project managers, configuration managers, architects, and reuse engineers). These violations may be programming errors or reflect changes 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.

4.3. Maintaining Visibility

Architectural governance benefits the organization in several ways. First, the existence of the master matrix provides consistent visibility at all levels in the IS organization; on-going communication of how restructuring has impacted the organization. It also facilitates change impact analysis.

L.L.Bean created an analysis and configuration tool leveraging metadata provided by Lattix that was 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 tool is called the Java Dependency Analysis Radar, or JDAR.

L.L.Bean creates its Lattix models using each Jar file as a root subsystem, and then parses the underlying metadata into query optimized data base tables. The parsed metadata includes Jar files, classes contained within each, and the dependency relationships identified by Lattix. A simple interface allows either a class name or Jar file name argument to specified, which results in either a Jar file or a list of Jar files. The data base tables are refreshed with each update to the master matrix.

The query operation is also exposed as a service for use with automated configuration management process. For example, dependent components defined in build scripts are updated with new dependencies, including the order of those dependencies when necessary.

5. Summary

The key to L.L.Bean's Java code restructuring success was increasing visibility of both software architecture and the process. L.L.Bean has found that increasing the visibility of software 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 and, while substantially changing the structure of the system, may not require substantial code modifications.

Disentangling the chaos of interdependencies required large-scale Java code reorganization, which is a complex process that requires proper tool support. Proper tools allow an organization to adopt an iterative approach to development and to communicate the current state of the software architecture to all stakeholders as progress is made. Scalability of representation has been a historic problem. The hierarchical, matrix-based representation provided a substantially more tractable view of Java code dependencies than provided by the typically used graph based approaches. Lattix proved a powerful and intuitive tool and has allowed L.L.Bean to successfully improve the agility of its software development process.

6. Acknowledgements

Sincere thanks to Doug Leland of L.L.Bean for his skillful mentorship and guidance, and to David Smith of L.L.Bean for his many insightful contributions to the Reuse Team and Java code restructuring.

7. References

[1] 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.

[2] 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.

[3] Fowler, M., Beck, K., Brant, J., Opdyke, W., and Roberts, D., *Refactoring: Improving the Design of Existing Code*, Addison Wesley, 1999.

[4] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns*, Addison Wesley, 1995.

[5] 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*.

[6] Jacobson, I., Booch, G., and Rumbaugh, J. *The Unified Software Development Process*. Addison Wesley, 1999.

[7] Jacobson, I., Griss, M., and Jonsson, P. *Software Reuse: Architecture, Process and Organization for Business Success*. Addison Wesley, 1997.

[8] Kruchten, P. "Architectural Blueprints: The "4+1" View Model of Software Architecture". *IEEE Software*, 12(6):42-50, November 1995.

[9] 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.

[10] 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 .

[11] Perry, D. and Wolf, A., ``Foundations for the Study of Software Architecture''. *ACM SIGSOFT Software Engineering Notes*, 17:4 (October 1992

[12] Poulin, J. S. *Measuring Software Reuse*. Addison Wesley, 1997.

[13] 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.

[14] Sangal, N. and Waldman, F. "Dependency Models to Manage Software Architecture". *The Journal of Defense Software Engineering*, November 2005.

[15] 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.

[16] 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.

[17] Stafford, J. and Wolf, A., "Software Archtiecture" in *Component-Based Software Engineering: Putting the Pieces together*, Heineman, G. T. and Councill, W. T. (Eds), Pages 371-388, Addison Wesley, 2001.