Paying Down Design Debt with Strategic Refactoring

Colin J. Neill and Phillip A. Laplante

Vol. 39, No. 12
December 2006

This material is presented to ensure timely dissemination of scholarly and technical work. Copyright and all rights therein are retained by authors or by other copyright holders. All persons copying this information are expected to adhere to the terms and constraints invoked by each author's copyright. In most cases, these works may not be reposted without the explicit permission of the copyright holder.
Paying Down Design Debt with Strategic Refactoring

Colin J. Neill and Phillip A. Laplante
Pennsylvania State University

Properly applied, strategic refactoring offers the most effective solution to decaying code.

The software evolution laws postulated two decades ago by Manny Lehman and Laszlo Belady—that software has the tendency to “decay” over time and become less understandable, maintainable, and more complex—have largely proven true. Combating software decay remains a vexing problem, particularly when the decay begins immediately. In such situations, starting from scratch is not an option; minor tweaks are insufficient and can even accelerate the decay.

Our studies indicate that strategic refactoring using design patterns is the most effective way to repair decaying code for object-oriented (OO) systems. However, applying a pattern-based approach to legacy system repair or even post-design pattern injection is often difficult and, in some cases if misapplied, detrimental.

DESIGN DEBT
Complex software systems erode over time. Software system must be extended, adapted, and modified accordingly as new requirements, constraints, and environments emerge. Developers, however, seldom give these efforts the rigorous consideration of the original design. Consequently, the system decays, resulting in decreased usefulness and increased errors.

Belady and Lehman’s “A Model of Large Program Development” (IBM Systems J., vol. 15, no. 3, 1976, pp. 225-252) provided detailed, empirical insight into the dynamics of software evolution in large systems. This seminal work led to the proposal of five laws (M. M. Lehman and L. A. Belady, eds., Program Evolution: Processes of Software Change, Academic Press, 1985) that govern the dynamics of software evolution, as Table 1 shows.

Building on this work, Stephen G. Eick and colleagues proposed the phenomenon of software decay in “Does Code Decay? Assessing the Evidence from Change Management Data” (Trans. Software Eng., vol. 27, no. 1, 2001, pp. 1-12). They showed how decay corresponds to incremental changes in software systems as these systems evolve to accommodate new functionality, support new hardware, or repair defects.

In the absence of rigor in the original design this erosion can occur very quickly, even before completion of the application. Joshua Kerievsky calls this erosion or decay “design debt” (Refactoring to Patterns, Addison-Wesley, 2005); as with all debt, the earlier it can be paid down the better.

STRATEGIC REFACTORIZATION
In Refactoring: Improving the Design of Existing Code (Addison-Wesley, 2000), Martin Fowler suggested that one way to pay down design debt early is through refactoring—the disciplined process of piecemeal modifications of a system’s internal structure without changing its external behavior. Well-known in the Smalltalk development community, refactoring first gained wide popularity with the agile development movement. Kent Beck championed refactoring as a replacement for

Table 1. Lehman and Belady’s laws of software evolution.

<table>
<thead>
<tr>
<th>Action</th>
<th>Law</th>
</tr>
</thead>
<tbody>
<tr>
<td>Continuing change</td>
<td>Software must be continually adapted or its usefulness will become progressively less satisfactory.</td>
</tr>
<tr>
<td>Increasing complexity</td>
<td>As software evolves, its complexity increases unless work is done to maintain or reduce it.</td>
</tr>
<tr>
<td>Self-regulation</td>
<td>Global software evolution processes are self-regulating.</td>
</tr>
<tr>
<td>Conservation of organizational stability</td>
<td>Unless feedback mechanisms are appropriately adjusted, the average effective global activity rate in an evolving software system tends to remain constant over the product lifetime.</td>
</tr>
<tr>
<td>Conservation of familiarity</td>
<td>The incremental growth and long-term growth rate of software systems tend to decline.</td>
</tr>
</tbody>
</table>
the document- and process-heavy design activities prevalent in plan-driven development. In effect, design is performed on the fly and continuously throughout system development.

Strategic refactoring involves design-pattern-based refactoring with architectural awareness. This simple and efficient process consists of:

- analyzing the existing application for "bad smells";
- determining the target macro architecture;
- defining the target domain-layer architecture;
- identifying microarchitectural patterns to implement changes; and
- teasing away, or culling, the refactored structure to improve connections between microarchitectural elements.

Strategic refactoring has played a valuable role in software evolution. OO design patterns from the Gang of Four such as the Strategy and Façade patterns have been particularly useful in the evolution and redesign of aging software systems, though other design pattern languages are similarly helpful.

Dennis Mancl demonstrated in "Refactoring for Software Migration" (IEEE Comm. Magazine, vol. 39, no. 10, 2001, pp. 88-93) that developers can use strategic refactoring techniques to successfully migrate old legacy code into a redesigned OO architecture with an extensible design.

Researchers have used several automated design tools to facilitate this process. For example, Lance Tokuda and Don Batory implemented a tool capable of automating a suite of schema transformations, design patterns, and hot-spot metapatterns to successfully update and improve source code design in evolving, real-world applications (“Evolving Object-Oriented Designs with Refactorings,” Automated Software Eng., vol. 8, no. 1, 2001, pp. 89-120).

Adrian Trifu, Olaf Seng, and Thomas Gansner likewise used automated refactoring to safely transform more than 14,000 lines of code (“Automated Design Flaw Correction in Object-Oriented Systems,” Proc. 8th Euromicro Working Conf. Software Maintenance and Reeng., IEEE CS Press, 2004, pp. 174-183). They based their integrated methodology for the evolution of real-world OO software on the concept of correction strategies, reference descriptions that "enable a human assisted
tool to plan and perform all necessary steps for the safe removal of detected design flaws, with special concern towards the targeted quality goals of the restructuring process.”

CASE STUDY

We recently applied strategic factoring in two iterations to improve Kahindu, a Java-based imaging toolkit created by Douglas Lyon. In the first iteration, we concentrated on fundamental improvements in the system’s understandability and maintainability and uncovered performance overheads and bottlenecks; in the second, we focused on refactored framework evolution for reusability and extendibility.

Another, simpler case study, which clearly illustrates the benefits of strategic factoring, targeted a multitiered, Web-based equity management application that its developers deemed as impossible to reuse (C.J. Neill and B. Gill, “Refactoring Usable Business Components,” IT Professional, vol. 5, no. 1, 2003, pp. 33-38). Although this application contained many obvious bad smells such as long methods, lazy classes, and data clumps, the most malodorous one was the unplanned architecture that had emerged during incremental development.

For this application, the developers made major design and architectural decisions based on the granularity and decomposition of use cases. Figure 1a shows a segment of the application design for a use case; each use case had a corresponding, similar set of classes.

Following the steps of strategic refactoring, the developers employed a two-stage transformation. First they had to rationalize the architecture—that is, introduce a coherent one effectively. To this end, they refactored appropriate Java 2 Enterprise Edition patterns into the application as shown in Figure 1b. These patterns served to separate the presentation tier and business services components and introduce more fine-grained cohesive classes.

Next the developers concentrated on the improvements’ original intent—reusability. Understanding that meaningful reuse in this sense was at the business-domain level, they grouped features commonly reused together across applications as business services.

These business services were based on a domain model for financial instruments, thus reducing the semantic gap between the system and the problem domain the use-case-based decomposition created. Finally, they decoupled these underlying services from the use cases that aggregated them, allowing the services to be shared across multiple use cases which eliminate duplication and redundancy within the application—while simultaneously creating reusable business services available to other applications. Figure 1c illustrates the final structure.
Common understanding of the refactoring practice is to focus on the piecemeal nature of individual refactorings. This influences developers to adopt a tactical perspective in which they focus on syntactic issues, coding standards violations, or simple design heuristics such as that base classes should be abstract and attributes private.

While admirable and often valuable, effective repair of decaying code often requires a more strategic approach that addresses architectural or macro-architectural deficiencies—low-level changes without a plan of attack are insufficient for such systemic problems. Put another way, the philosophy for strategic refactoring can be stated as a play on an old adage—think locally, act globally.

Colin J. Neill is an associate professor of software engineering in the Engineering Division at the Pennsylvania State University. Contact him at cjneill@psu.edu.

Phillip A. Laplante is a professor of software engineering in the School of Graduate Professional Studies at the Pennsylvania State University. Contact him at plaplante@psu.edu.

Editor: Michael G. Hinchey, NASA Software Engineering Laboratory at NASA Goddard Space Flight Center and Loyola College in Maryland; michael.g.hinchey@nasa.gov