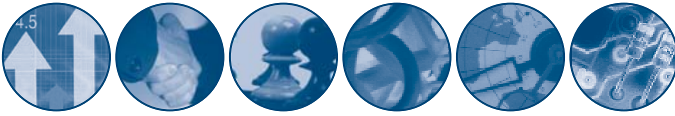

Update

Agile Product & Project Management Advisory Service
Executive Update Vol. 11, No. 20

Validating Legacy Code:

Modernization Strategies Through Technical Debt Assessments

by John Heintz, Technical Consultant, Cutter Consortium

What strategies do you apply to modernizing a product code base? What results do you get with those strategies? This *Executive Update* takes a retrospective look at a past project, both to describe the strategies my colleagues and I used to rearchitect the product and to validate the effectiveness of those strategies with two technical debt assessments via Cutter's Technical Debt Assessment and Valuation practice.[1] The six strategies we used are presented here. The two assessments are used to evaluate the measured impact on the system from the team's efforts and compare it to the actual time spent modernizing the code.

This is the story of the DeLorean[2] system, a client's longtime production setup. This client had successfully developed, evolved, and sustained this system, and its business, for more than a handful of years. This Java Struts[3] Web application embodied the cumulative business and technical experience of the whole company. So far, so good.

A PREDICTABLE STORY

It's easy to predict the next part of the story: slowing down, inflexibility, and brittleness. After some point, enough technical debt had settled into the system to cause problems. In this case, the primary causes were multiple inconsistent design approaches over time, significant code duplication, no test automation or manual test plans, and lack of discipline managing technical debt in the code. New features took increasingly longer to complete. Bugs were more difficult to track down and really fix. Changes made to the system needed to be carefully tested by many individuals (manually) and often had nonlocal failures. Getting an accurate estimate of effort for planned changes became difficult. Finally, each new release became an all-hands-on-deck event in which many individuals were prepared to deal with production failures.

All of this made it more difficult to capitalize on business opportunities with both new customer features and partner integrations — the situation was recognized to be affecting the business.

BUY VS. REWRITE VS. REARCHITECT

Given a consensus view that something must be done, the choices simply boiled down to three options:

1. Buy a commercial replacement system, and customize as needed.
2. Rewrite from scratch.
3. Rearchitect the current production system in place.

When the time came to choose among options, the company had no empirical data to base a decision on, but relied on evaluations and judgment based on the following:

- Suitability of commercial offerings for the business
- Failure rates for big rewrite projects
- Perceived effort to rearchitect the existing system

The management team chose option 3: rearchitect the application. This choice was based primarily on (1) the low suitability of commercial offerings and high risk of a rewrite failure and (2) that proceeding with a rearchitecture cleanup would reduce the risks of possibly implementing options 1 and 2 in the future by providing a cleaner and better-defined platform to move from.

THE CHARTER

The DeLorean project was explicitly chartered with cleaning up the architecture, removing duplication, improving code quality, building in testing, and improving reliability; in short, to remove technical debt. The

team also needed to support new business features and integrating with a customer relationship management (CRM) appliance. The team's efforts were guided by judgment and experience to balance the progress on all of these goals; no tools were used at the time for measuring the reduction of technical debt or complexity.

DeLorean lasted more than a year, with three to four people at a time working concurrently. Team members subjectively considered "at least half" of their time to have been spent on making improvement to the code and reducing the technical debt. The remaining time focused on developing and integrating new features.

THE REARCHITECTURE APPROACH

The team's strategies for reducing technical debt and improving the system evolved over time. The six strategies used on the DeLorean project were as follows:

1. **"Fix it if you see it's broken."** This rule gave the team liberty to make the numerous small changes to improve the code by reducing complexity, documenting, refactoring, and adding unit tests. In particular, this meant fixing all bugs in any duplicated copy-and-pasted code.

   The team was very careful to keep the scope of "fixes" balanced, neither too small nor too large. It was particularly easy to bite off too large a change when each fix would make visible more "broken" code. Both timeboxes and ad hoc team reviews of scope were used to maintain this balance, serving as metaphorical brake pedals.

2. **Flexible schedule commitments.** The scheduled delivery of new features was not cast in stone, and often the date and features of a production release were altered to support ongoing cleanup activities. The two primary factors that triggered this were (1) wrapping up refactoring of a significant subsystem or (2) taking the time to clean up and test a region of code newly recognized as high risk for failure.

3. **Technical management.** The manager of the DeLorean team was also a senior technical contributor. This enabled decisions to be made that accurately balanced customer demand/value, technical risk/reward, and schedule. There was little conflict or justification needed when balancing these forces.

4. **Technically competent and experienced team members.** The team had no junior members. The risks associated with modifying untested production code required that each member of the team be able to judge (1) the likelihood of nonlocal effects, (2) what refactoring and test strategies to apply, and (3) when to call in help from the rest of the team.

5. **Tracking lines of code removed.** Every few days the team would calculate the total lines of code and celebrate when dozens or hundreds of line were removed. Code was removed by refactoring to better design abstractions, through the use of application frameworks, and by deleting copy-and-pasted code.

6. **Defining unit test and coverage rules for different parts of the code base.** New code was written into a new package namespace. This new namespace was the focus of unit testing, held to high craftsmanship standards. Generally, the team code reviewed new modules. A best-effort attempt would be made to refactor legacy code at hand into this new package namespace while adhering to the code, test, and design standards. Any legacy code not moved was given much less attention and focus until the next opportunity for refactoring appeared.

   Examples of the standards applied to the new code namespace include: (1) greater than 80% unit test coverage, (2) design modules with DI[4] into a layered code model, and (3) a preference for clear readable code, a subjective judgment. These standards evolved over time. At the beginning of the project, there were no such rules, and it took some time to understand the testing and refactoring options for the legacy code. Initially, we'd assumed that a single global percent measure of unit test coverage would be desirable, but after creating the new package namespace, the team realized only that code should have testing standards.

   Both management and the DeLorean project team maintained a disciplined balance between the priorities of business features and continuing to apply these strategies for reducing technical debt. The day-to-day focus of the team was either blended

The *Executive Update* is a publication of the Agile Product & Project Management Advisory Service. ©2010 by Cutter Consortium. All rights reserved. Unauthorized reproduction in any form, including photocopying, downloading electronic copies, posting on the Internet, image scanning, and faxing is against the law. Reprints make an excellent training tool. For information about reprints and/or back issues of Cutter Consortium publications, call +1 781 648 8700 or e-mail service@cutter.com. Print ISSN: 1946-7338 (*Executive Report*, *Executive Summary*, and *Executive Update*); online/electronic ISSN: 1554-706X.

across those goals or would soon return to a balance after a necessary focus of fixing a bug or releasing an important feature. Where possible, the team applied the strategies for reducing technical debt while implementing new features or fixing bugs.

WHAT IS A TECHNICAL DEBT ASSESSMENT?

Cutter's Technical Debt Assessment is two distinct things: (1) an automaton-assisted analysis of code[5] and (2) an ongoing input to project governance and management. The automated analysis is both dynamic (unit testing and code coverage) and static (rule conformance, code complexity, duplication of code, documentation, and design characteristics). Every deficit identified in the analysis is added to a total time effort, measured both in person-days and dollars.

TECHNICAL DEBT ASSESSMENT: HISTORIC ANALYSIS

The Technical Debt Assessment was conducted, post hoc, on a 12-month period during the DeLorean project. Both the accounting records and source code repository were examined during that time frame to enable a comparison of the actual effort to the assessed reduction in technical debt.

Assessments of the source code repository at the start and end of the 12-month period indicate how much technical debt had existed when the team started working on the DeLorean project. As shown in Figure 1, the total starting technical debt was 740 person-days, consisting mostly of lack of coverage, code duplications, and code violations.

How did the DeLorean project fare at the end of the period? As Figure 2 shows, the system was significantly improved, reducing total assessed technical debt to 415 days, down 325 from the original 740 days. This was a 44% reduction from the year before.

In addition to the aggregated technical debt, other project data changed in significant ways. Figure 3 visually shows the relative changes from project start to finish for coverage (shown as untested code percentage), complexity, code duplication percentage, lines of code in

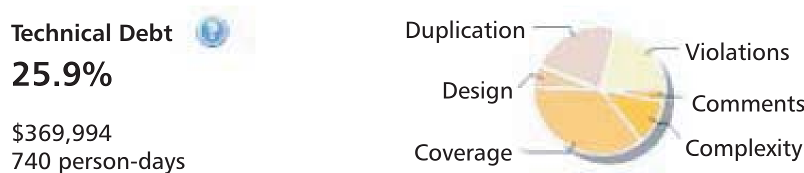

Figure 1 — Technical debt at start.

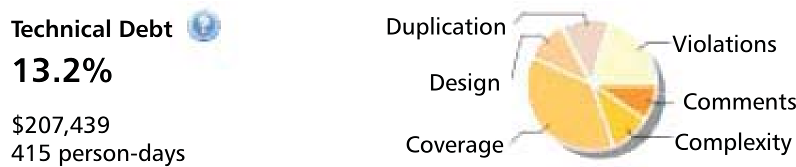

Figure 2 — Technical debt at finish.

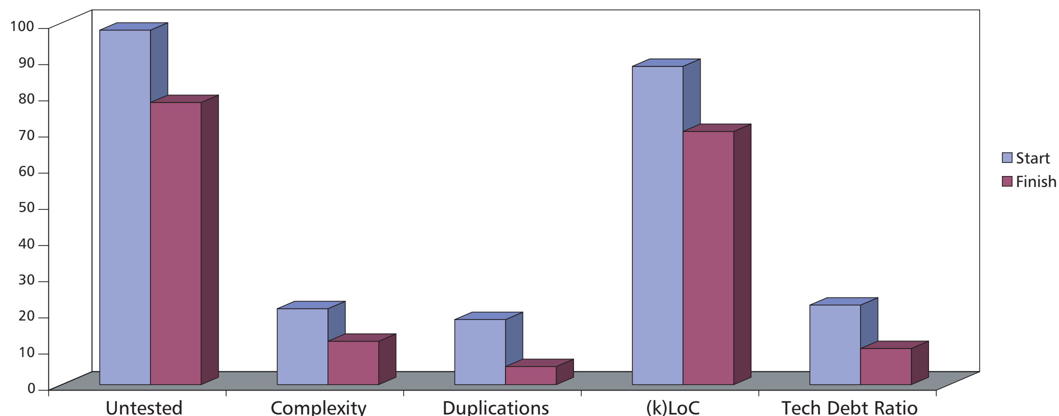

Figure 3 — DeLorean project changes.

the thousands, and the technical debt ratio.[6] In each of these measures, the DeLorean project made visibly significant progress.

The measured changes to the size of the project, complexity, amount of code duplication, and test coverage are shown in Table 1.

The accounting records for the DeLorean project were used to assess the total effort as 337 person-days spent reducing technical debt. This was based on the subjective evaluation that "at least half" of the team's time was spent on cleaning up the system and an accounting of 674 total days.

CONCLUSION

The strategies applied helped the team rearchitect a large portion of the product and maintain forward momentum for the entire DeLorean project. I am currently applying these strategies to modernize code in other projects.

The actual recorded effort (337 days) corresponds to the analyzed improvement (325 days). This correspondence is accurate within 4%, an astonishingly close result. This accuracy is interpreted as success for both the DeLorean team's legacy modernization strategies as well as the tools and process of the Technical Debt Assessments.

Table 1 — DeLorean Summary at Finish

	Lines of Code	Complexity/Class	Duplications	Test Coverage
At Start	91,969	23.4	22.1%	0%
At Finish	73,610	15.4	7.6%	19.9%
Improved by	20%	34%	66%	N/A

ENDNOTES

[1]See "Technical Debt Assessment and Valuation." (www.cutter.com/consulting-and-training/technical-debt-assessment.html).

[2]While not the project's real name, this reference to the time-machine car from the 1985 movie *Back to the Future* provides an analogy to the jumps from today to two points in the past for each Technical Debt Assessment.

[3]Java Struts is a popular Web application framework for the Java language (http://struts.apache.orgi).

[4]DI is dependency injection, a style of composing objects to reduce coupling.

[5]Many tools can accomplish this. The Sonar tool is used here. (http://sonarsource.org).

[6]This is a ratio of current technical debt compared with total possible technical debt.

ABOUT THE AUTHOR

John Heintz is a Technical Consultant with Cutter Consortium's Agile Product & Project Management practice. He is an experienced agile manager, particularly in lean and Kanban. In 2008, Mr. Heintz founded Gist Labs to further focus on the essential criteria for innovative success. This focus has led to Concrete Reflective Tools, which are immediately useful, generate feedback information, and are backed by their own guiding principles. On a recent project, he coached a 100-person agile/lean game studio, helping the organization increase its throughput of game features per month while coordinating cross-team communication paths, resulting in a doubling of features in one year.

Mr. Heintz's approach to systems and team building emerged in 1999 as he led his first Scrum team, coaching XP and test-driven development. He has consulted with clients on enterprise architecture, development, and TDD practices; XP and Scrum leadership; Kanban coaching; and RESTful/messaging architecture. Mr. Heintz is a regular speaker at industry events, including No Fluff Just Stuff (NFJS), Architecture and Design World, and Dallas JavaMUG; he is the Program Chair for Agile Austin. Mr. Heintz holds a BS in electrical engineering from the University of Michigan. He can be reached at jheintz@cutter.com.

 ©2010 Cutter Consortium