Technical Debt: The Cost of Doing Nothing

by

Austin M. Page

B.S. Electrical Engineering, University of Maryland, College Park, 2006
M.S. Electrical Engineering, Wright State University, 2010

Submitted to the System Design and Management (SDM) program in
Partial Fulfillment of the Requirements for the Degree of

Master of Science in Engineering and Management

at the

Massachusetts Institute of Technology

February 2019

Signature of Author. . . . . .
Austin Page
Graduate Student, System Design and Management Program
January 18, 2019

Certified by . . . . . .
Alan MacCormack
Adjunct Professor of Business Administration, Harvard Business School
Lead Advisor

Certified by . . . . . .
Steven D. Eppinger
Professor of Management Science and Innovation
MIT Advisor

Accepted by . . . . . .
Joan Rubin
Executive Director, System Design and Management Program

THIS PAGE INTENTIONALLY LEFT BLANK

Disclaimer

The views expressed in this document are those of the author and do not reflect the official position or policies of the United States Air Force, Department of Defense, or Government.

THIS PAGE INTENTIONALLY LEFT BLANK

Technical Debt: The Cost of Doing Nothing

by

Austin M. Page

B.S. Electrical Engineering, University of Maryland, College Park, 2006
M.S Electrical Engineering, Wright State University, 2010

Submitted to the System Design and Management (SDM) program in
Partial Fulfillment of the Requirements for the Degree of

Master of Science in Engineering and Management

at the

Massachusetts Institute of Technology

February 2019

Abstract

The Air Force is currently paying a cost for the mismanagement of its software development activities. Software-intensive systems are consistently plagued with cost, schedule, and performance issues, which in the current fiscal environment is unsustainable. There has been much research on the benefits of process improvement, yet the concept of product health is largely ignored. Technical debt – the consequence of making short-term design decisions at the expense of long-term health – has been accumulating within code bases as developers and managers struggle to identify, quantify, and manage it properly. In this thesis, an extensive literature search is performed to define technical debt, explain its implications, and highlight methods to quantify and visualize it so organizations can address it explicitly. Through the use of architectural health analysis tools, a set of metrics is defined and used in case studies to highlight the extent to which the Air Force has lost control of its software and the price it has to pay because of it. Ultimately, eleven recommendations are given on how to incorporate architectural health analysis tools into software development activities to prevent, identify, manage, and reduce the amount of technical debt across product lifecycles.

Thesis Supervisor: Alan MacCormack
Title: Adjunct Professor of Business Administration, Harvard Business School

THIS PAGE INTENTIONALLY LEFT BLANK

About the Author

Major Austin M. Page is currently an Air Force Fellow attending the Massachusetts Institute of Technology in pursuit of his Master's degree in System Engineering. Prior to this assignment, he served as the Deputy Chief of Weapons assigned to the F-35 Joint Program Office in Arlington, VA. He led a 55-member tri-service team across 5 sites and 8 international countries overseeing all weapons activity in the F-35 program. He was directly responsible for the development, integration, test, verification and closeout of $700M in F-35 weapons integration requirements.

Major Page graduated from University of Maryland in College Park, MD and was commissioned in the Air Force in May 2006 as a Developmental Engineer assigned to Air Force Research Laboratory in Wright Patterson AFB, OH. After two years in a lab setting he moved to the Aeronautical Systems Center where he served as an Avionics Engineer in the C-17 Program Office. Here, Maj Page oversaw the development, testing, fielding and maintenance of over $400M in avionics equipment. In 2010, Maj Page assumed multiple leadership roles in the 513th Electronic Warfare Squadron at Eglin AFB, culminating as a Flight Commander. In this role he commanded an 11-member flight responsible for the development, testing and fielding of F-35 mission data, a critical path component to the fielding of ACC's #1 acquisition program. While stationed at Eglin AFB, he attended Squadron Officer School in Maxwell AFB, AL where he was bestowed Distinguished Graduate honors. Following his tour at Eglin AFB he was selected to participate in the AFIT sponsored Education with Industry program where he spent 10 months learning best business practices in private industry before returning to government acquisitions.

THIS PAGE INTENTIONALLY LEFT BLANK

Acknowledgements

Thanks to Dan Sturtevant, Sean Gilliland, Sunny Ahn, and Carol Ann McDevitt from Silverthread Inc., for providing their tools, their data, and their expertise for this thesis.

Thanks to my advisors, Alan MacCormack and Steve Eppinger, for their feedback and guidance throughout this research effort.

Thanks to Jim Reilly, Joe Besselman, Maria Hallett, Chris Froude, and Conner Van Fossen for providing the wealth of data that went into the case study section of this research.

Thanks to my wife, Jamie, for her support and patience along the way.

Finally, thanks to the men and women that serve in our armed forces. This thesis is intended to benefit you and what you do for our country.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

List of Figures

THIS PAGE INTENTIONALLY LEFT BLANK

1 Introduction

1.1 Background

> *"Software has become one of the most important components of our Nation's weapons systems, and it continues to grow in importance. Software defines the way our systems see, communicate, and operate in combat. Design and acquisition decisions at the beginning of the software development process frequently have far-reaching and long-term effects that impact the weapon system's efficacy on the battlefield and its ability to adapt to changing requirements." – Defense Science Board Report on Design and Acquisition of Software for Defense Systems, February 2018* [1]

The current state of software acquisition is a concern within the Department of Defense. [1] [2] [3] [4] [5] [6] [7] This has been highlighted through high-profile programs such as the F-35 Lightning II, Air Operations Center, and Next-Generation Operational Control System programs breaching cost, schedule or performance constraints on one or more occasions. [8] As weapon systems have become more reliant on software, program managers and developers alike must understand how their decisions today will affect both current performance and future capability. This includes the ability to understand the source code and its structure, the ability to discover and fix deficiencies, the ability to effectively and efficiently integrate new capabilities, and the ability to perform maintenance and sustainment of the code as it evolves, including re-factorization efforts to manage the size and complexity of core files.

While the ability to understand the source code is of paramount importance, it is equally critical that developers and managers use that knowledge to manage both short-term and long-term objectives. Too often, long-term objectives are sacrificed to achieve short-term gains. Ward Cunningham coined the term "technical debt" as a metaphor for the trade-off between writing clean code at higher cost and delayed delivery, and writing messy code cheap and fast at the cost of higher maintenance efforts once it's shipped. [9] Joshua Kerievsky extended the metaphor to architecture and design. [10]

Unfortunately, the accumulation of technical debt is a common occurrence in the Air Force. [6] Over time, developers have found areas of their code bases that have become difficult to modify as technical issues are neglected to keep product timelines on track. High levels of interdependency between files and/or modules form within the code base and cause unintended ripple effects which developers are not able to quantify objectively. As such, developers are reduced to performing trial and error to produce the performance results they're aiming to achieve. This trial and error process takes more time, pushing timelines out even further and causing the backlog of technical debt to increase, forming a negative reinforcing cycle. To quantify these programmatic impacts, literature has been published linking the financial impacts of technical debt with the level of architectural erosion of time. [11]

To aid the efforts of the program managers and developers tasked with coding and sustaining highly complex, software-intensive Air Force systems, this thesis explores methods to reduce technical debt across the spectrum of Air Force software development activities. Using data from select Air Force R&D and operational programs, this thesis analyzes the architectural health of five code bases to assess the level of technical debt that has accumulated over time. The core case study tracks the evolution of a software development program over a 10-year period, including an organic re-factoring effort that has been underway since September 2017. It shows how the results from the static scans have been able to guide engineering and management teams to significantly reduce the backlog of technical debt that had previously overwhelmed the program. Three case studies will focus on cases where Air Force code bases have already become unstable. In these cases, this thesis will examine the metrics, actions, and process changes that facilitated management's "re-factor" vs. "re-write" decisions. The last case study provides insight into a program that has avoided taking on technical debt, providing lessons learned for other software development efforts.

Ultimately, this thesis offers eleven recommendations on how to reduce technical debt across the Air Force software acquisition enterprise through the use of new teaming arrangements and quantitative architecture analysis tools. These recommendations include contextual factors on when the tools should be used along with specific action plans on how to overcome organizational inertia to implement these measures.

1.2 Research Questions

There are two research questions that will be addressed in this thesis. First, this thesis seeks to explore the connection between the Air Force's current software development practices and the commercial implementation of technical debt management by asking:

1) *Do the architectural health analysis tools on the market provide relevant, objective, actionable data to help quantify, visualize, and resolve technical debt within a sample of Air Force code bases?*

This connection will be explored with a combination of interviews and technical assessments on five different code bases within one Air Force portfolio. The interviews will provide a qualitative assessment on management's awareness and proficiency in managing technical debt, while the use of architectural health analysis tools will provide a quantitative assessment on the level of technical debt in each code base. By obtaining both qualitative and quantitative results through several case studies, a correlation can be made between the management of each program and the health of its resulting products.

The second question addresses how the Air Force can better manage technical debt within the software acquisition cycle by asking:

2) *What actions can be taken to reduce technical debt across the Air Force, thereby increasing development efficiency while reducing cost, schedule and performance issues across the full spectrum of Air Force software development activities?*

This question will be addressed through the synthesis of case study results in combination with key leader interviews that explore the technical, logistical, structural, procedural, and contractual changes that could aid in the reduction of technical debt.

1.3 Thesis Structure

This thesis is broken into seven chapters. The first chapter gives background and context to the research that follows. It outlines why the research topic was chosen, what specific questions are being addressed, and how the thesis is structured.

The second chapter provides background on the topic of technical debt. This includes its definition, it's prevalence in software development, and its impact on Air Force acquisition. The bulk of the chapter presents results from an extensive literature search that has been done on the subject.

The third chapter discusses ways to measure technical debt. It examines the components of technical debt and how architectural health can be used as a proxy for the accumulation of technical debt within a system. An overview of architectural health tools will be given along with a section on how to interpret the results of the tools. This section outlines key concepts required to understand the case studies that follow.

The fourth chapter discusses how the case studies were chosen and how they relate to the overall objectives of this thesis. It also discusses the process associated with performing the architectural health scans.

The fifth chapter presents the five case studies as examples of how various Air Force programs have avoided, discovered, managed, and fallen victim to technical debt. In cases where architectural health analysis tools were used in the development process, the results show positive results. In cases where technical debt or architectural health were not tracked, the results show the implications of this oversight. Finally, in Case D the case study reflects how architectural health tools can be used to guide a re-factoring effort over time, which underscores one of the recommendations of this thesis.

The sixth chapter synthesizes the results of the case study section. It highlights common themes amongst the cases and provides insight into how architectural health tools could be used in Air Force acquisitions. It concludes with a discussion on opportunities and barriers to implementation and senior leader perspectives on their current and desired decision-making processes, and how architectural health tools could help them realize that vision.

The seventh and final chapter provides recommendations to Air Force acquisition professionals for how to change the business landscape to facilitate the use of architectural health analysis tools, along with specific recommendations on where these tools could be integrated in the product lifecycle. The chapter concludes with a brief discussion on areas of potential future research.

2 Literature Review

Over the past several years, there has been an increasing amount of literature published on the subject of technical debt. The sources of this research span from universities, to industry consortiums, to institutions like the Software Engineering Institute (SEI). This section consolidates ideas from relevant publications to define technical debt, highlight root causes of technical debt within software development efforts, discuss the implications of technical debt within the Air Force, and give a summary of methods that can be used to reduce technical debt.

2.1 What is "technical debt" and why do I care?

Understanding technical debt and the basis for its accumulation is important as it can have significant impacts on the cost, schedule and performance of a software system. According to SEI, "all software developers understand intuitively what technical debt is, [however] they lack clear guidance and proven techniques on how to identify it, how to concretely describe it, and how to account for it within the software-development life cycle." [12]

The following definitions provide insight into the meaning of technical debt. Each one is slightly different, however the theme of sacrificing long-term health for short-term gains are common in each definition.

According to the notes from the Dagsthul Seminar on Managing Technical Debt in Software Engineering, "technical debt is the collection of design or implementation constructs that are expedient in the short term, but set up a technical context that can make future changes more costly or impossible. Technical debt presents an actual or contingent liability that impacts internal system qualities, primarily maintainability and evolvability." [13]

According to DevIQ, "Technical debt is a metaphor for all of the shortcuts, hacks, and poor design choices made for a given software system that compromised its quality, usually in an attempt to meet a deadline." [14]

According to MacCormack et al., "Technical debt is created when design decisions that are expedient in the short term increase the costs of maintaining and adapting this system in future." [11]

According to Ozkaya, Deputy Lead of the SEI Architecture Practices Initiative, "Technical debt communicates the tradeoff between the short-term benefits of rapid delivery and the long-term value of developing a software system that is easy to evolve, modify, repair, and sustain." [12]

For the sake of this research, the paper will utilize the definition from MacCormack et al. [11] for its clear and concise interpretation.

Technical debt can manifest itself in various ways, but is ultimately a non-ideal property of a technical system that leads to non-ideal business outcomes. Figure 1 shows a characterization from SEI on how technical debt can impact both maintainability and evolvability of a code base. On the left-hand side, the figure represents how poor architectural health could slow down the delivery time lines of new features. On the right-hand side, the figure represents how poor code health could influence the lifecycle cost associated with maintaining the code base. In both cases, the accumulation of technical debt can levy programmatic impacts on software development efforts.

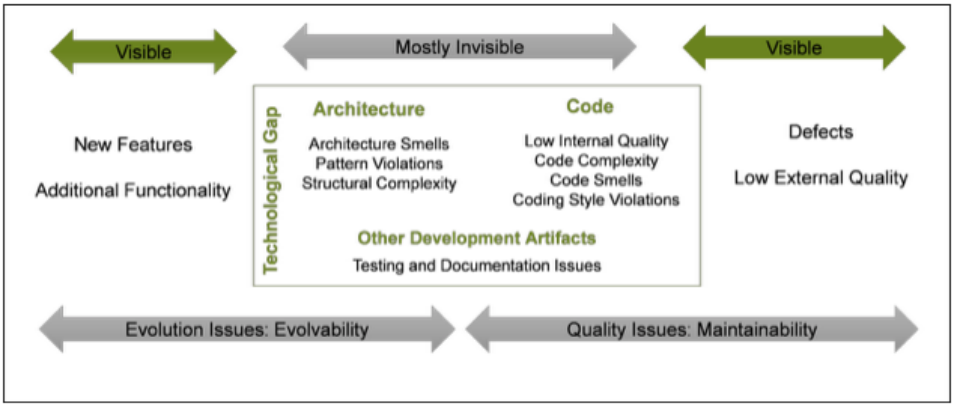


Figure 1 - Technical Debt Landscape [13]

Additionally, research has shown that different categories of technical debt arise during a development effort. Figure 2 shows four categories of technical debt that fall along two axes pertaining to how deliberate and thoughtful the developers were in their effort.

	Reckless	Prudent
Deliberate	"We don't have time for design"	"We must ship now and deal with consequences"
Inadvertent	"What's layering?"	"Now we know how we should have done it"

Figure 2 – Visualization of Technical Debt Quadrants [15]

Of the four quadrants shown in the figure, there is a hierarchy of severity based on how the technical debt is generated and how it is handled after the fact. The bottom left quadrant is by far the worst of the four because the developers don't realize that they've accumulated technical debt, and thus, have no recourse to fix it. This could be caused by lack of knowledge, training, or coding standards. The top left quadrant reflects an understanding that technical debt is being accumulated, but a rejection of the premise that it needs to be managed. This practice could be satisfactory for throw-away systems, but not systems that will require long-term maintenance. The bottom right quadrant reflects when developers don't realize they've accumulated technical debt until after the system has been released, however there is a desire to correct the issue moving forward. This represents "learning" in an organization and can help guide coding standards, development processes, and training moving forward. Finally, the top right quadrant reflects when developers willingly accept technical debt but also develop a plan to reduce it over time. An example of this situation would be releasing a product to meet a deadline but following the release with a focused period of "clean-up" afterward. [14]

2.2 Is all Technical Debt Bad?

Not all technical debt is bad. According to SEI, "Technical debt can be thought of as a design strategy. The financial metaphor is apt because, as with finances, getting into debt in software development by optimizing short-term goals creates tangible value. It represents a tradeoff, a choice of one option over another. But selection of this option must be managed through the duration of the project, with a clear understanding of the penalties that result. The danger lies in choosing the option and making the tradeoff, but then forgetting about, ignoring, or underestimating the consequences." [12] In an interview with the Chief Technology Officer (CTO) for one of the case studies, he stated they made a conscious decision to allocate chunks of time for feature development interspersed with periods dedicated strictly to tackling technical debt. As such, they are able to move quickly and stay engaged with the customer in the feature development phase as a direct result of their ability to "catch up" during their tech debt phase.

While there are many ways to address technical debt, it cannot be ignored altogether. Whether technical debt is addressed in batches as described above, or addressed as the code is being developed, left untouched for too long, it will degrade the software system. Ultimately it will be up to the owner of the software system to determine how to handle the technical and economic implications of this systemic issue.

2.3 Sources of Technical Debt Within the Air Force

There are several sources of technical debt within the Air Force acquisition system. The sources include contractual, cultural and technical factors. Each of the factors will be discussed below, highlighting specific root causes that could be addressed through policy, process, or product management changes.

2.3.1 Contractual Sources of Technical Debt

> *"The utility of defense hardware is beholden to its software, and data rights hold the key to life-cycle affordability… services need to avoid vendor lock, where proprietary rights are restricted to the original system vendor…the military must make the most out of finite budgets, and owning all the data rights and associated intellectual property is one way to create a cost competition." - senior military analyst Dave Deptula* [16]

From a contractual perspective, lack of data rights and improper incentives are two of the largest drivers of technical debt in software systems. At the time of contract award, the government must make decisions on how to handle contractor data rights. In the past, fiscal constraints have forced the government to make short-sighted decisions that precluded them from procuring full rights to the source code. Without data-rights, the government has little or no visibility into the contractor-developed source code, forcing them to rely largely on functional testing as their sole source of evaluation. While functional testing does verify the current performance of a system, it does not provide insight into the maintainability, sustainability, or scalability of the code base. As such, relying on functional testing could easily mask technical debt or architectural deficiencies within a system, increasing overall lifecycle cost. Unfortunately, passing functional test is typically the major milestone required for deliverable acceptance in both fixed price and cost reimbursable contracts.[1]

In addition to the lack of transparency, the government's lack of data rights requires them to return to the original contractor for future changes regardless of programmatic implications. This concept is commonly referred to as "vendor lock" [16] or "intellectual property lock in." [17] By forcing the government to return to the same vendor, contractors are not incentivized to address their technical debt. Instead, the contractor's technical debt is passed through to the government in the form of inflated costs and decreased velocity. Since the government is "locked in", they are forced to pay the bill.

With that said, even if data rights are acquired up front, in some they are not sufficient to enable fair and open follow-on competition to avoid future vendor lock. Technical debt is directly tied to the learning curve for a new company to take over a code base, and should be factored into any cost proposal. Maintaining a high technical debt creates a "burden of entry" for any other company into the code base. This potentially creates a corporate strategy to justify high cost employees while maintaining a vendor lock on the code base.

---

[1] Cost-Reimbursement types of contracts (FAR Subpart 16.3) provide for payment of allowable incurred costs, to the extent prescribed in the contract. These contracts establish an estimate of total cost for the purpose of obligating funds and establishing a ceiling that the contractor may not exceed (except at its own risk) without the approval of the contracting officer.

2.3.2 Cultural Sources of Technical Debt

From a cultural perspective, there are two aspects to consider: the culture of the Department of Defense (DoD) and the culture of the development organizations. At the DoD level, the culture surrounding development and sustainment activities is one of risk-aversion. As such, the acquisition process has become bloated with rules and regulations, largely reactive measures from failures of years past. The documents that guide acquisitions efforts, the FAR, DFARS, and DoD 5000 series documents have grown to the point where they have become unmanageable, and while the size of the documents is largely independent from the concept of technical debt, the behavior it drives is not. Instead of focusing on product health, government officials focus on check-list compliance. Instead of focusing on capability delivery, government officials focus on whether the correct forms were filed. While this depiction is slightly exaggerated, it does give a picture of the cultural landscape in which the DoD operates. A culture that is less focused on improving the core infrastructure of systems (i.e. reducing technical debt), and more concerned with meeting deadlines, completing milestone events, and meeting short-term budget numbers.

To address the bureaucracy within the DoD acquisition system, Ellen Lord, the head of the DoD's Acquisition and Logistics office states that, "In 2019…We are going to invert [the tailoring] approach and take a clean sheet of paper and write the absolute bare minimum to be compliant in 5000.02, and encourage program managers and contracting officers to add to that as they need for specific programs." [3] Lord and others recognize the need to change the acquisition process. They understand that cultural factors are partly to blame for driving cost and risk into DoD systems through the emphasis on checklist compliance and short-term cost savings. While her ambitions are admirable, there will continue to be significant pressure to incur technical debt until these changes take hold.

At the development organization level, program managers often do not emphasize the importance of long-term health in software systems, leading to poorly defined, poorly developed, or poorly tested software that is fielded solely to meet timelines. In the context of operational programs, combatant commanders are continually asking for systems with new functionality. While this mentality is understandable given their perspective of the military landscape, acquisition professionals must articulate the fact that it is much harder, and more expensive, to add new capability in a future iteration if the code base is in disarray. According to Joe Besselman, a technical SME for PEO Digital[2]:

> *"The warfighter or customer practice of "capability-only investments" and acquisition officer compliance is truly insidious. It leads to a succession of releases over time*

[2] Battle Management PEO re-organized into PEO Digital in 2018

accompanied by one or more runtime libraries of different generations. The more egregious aspects of this situation are that 1) we considerably increase the Cyber threat to our capabilities by operating with expired libraries, which to the uninitiated means we're operating software with published exploits and readily available patterns or recipes on the Internet to attack those exploits and 2) in many cases we have sustainment contracts based on lines of code, so as we add these duplicative libraries to a system the size of the code balloons and many program managers fail to understand they are paying to sustain multiple generations of the same library along with their custom code." [18]

A balance must be achieved between delivering "bright, shiny objects" and sustaining the underlying foundation of the code, both for maintenance and security reasons. This balanced approach must be incorporated in the culture of acquisition organizations or else the pressure to perform in the short-term will create overwhelming amounts of technical debt.

In the context of R&D programs, program managers are incentivized by tech transition and fielding rates. Often, irrational optimism can inflate transition rates to the detriment of the products and engineers, which can be compounded by the use of subjective measures for "readiness." Investment strategies focus on new starts and ongoing efforts, but have minimal review of completing programs. There is little review of actual experimental results and actual transition success/failure. All of these factors lead to an emphasis on getting new capabilities out the door, often at the expense of the operators that will be using it or the maintenance organization that will be sustaining it. This short-sighted approach, along with a lack of quantitative data to track the programs after they're fielded, facilitates the accumulation of technical debt in our R&D portfolio.

2.3.3 Programmatic Sources of Technical Debt

"DoD and other government acquisition managers must be able to assess what kind(s) of technical debt their developers and software contractors are creating when they make decisions." – Ipek Ozkaya, SEI Architecture Practices Initiative [12]

From a programmatic perspective, technical debt is largely left undetected due insufficient metrics and a lack of standardized tools to assess quality or maturity in software systems. While a software development plan is required for every new acquisition program, the process, tools, and metrics to be used within the effort are largely subject to program manager discretion. According to the USAF Weapon System Software Management Guidebook, [7] software metrics should:

- Be integral to the developer's processes.
- Clearly portray variances between planned and actual performance.
- Provide early detection or prediction of situations that require management attention.

- Support the assessment of the impact of proposed changes on the program.

This overarching guidance is vague and subject to interpretation. While in general it is favorable to keep high-level guidance devoid of specific tools, processes, and metrics, given the current state of Air Force acquisition, it could be helpful to have a list of accepted, standardized tools and metrics that are kept up to date with state-of-the-art industry practices. [1]

In contractor-led development efforts, metrics should be defined during the source selection process and adhered to throughout the development effort. [7] While most programs track software metrics, the metrics themselves are largely out-of-sync with current industry practices. Typical metrics included in Air Force acquisition programs include: [7]

- Software size
- Software development effort
- Software development schedule
- Software defects
- Software requirements definition and stability
- Software development staffing
- Software progress (design, coding, and testing)
- Computer resources utilization

Most of these metrics are based on legacy measures that don't capture architectural health, technical debt, or any aspect of developer productivity. By incentivizing contractors to work to these metrics, the government is ignoring a major component of cost both in development and sustainment. By selecting metrics focused on system complexity, cyclicality, and modularity, contractors would shift their efforts from short-term metrics to long-term health, saving money over the course of the product's lifecycle.

Similarly, government-led efforts should choose metrics that provide insight into health of their product. The main difference between government-led efforts and contractor-led efforts is the amount of control the government has during the development process. Contractor-led efforts are typically performance-based, precluding the government's involvement in pre-delivery assessments. Put another way, the government keeps control of "what" the contractor is delivering (i.e. executable) but has less control over "how" they're developing it. The lack of standardized tools, useful metrics, and insight into contractor-led product assessments all contribute to the accumulation of technical debt.

2.4 The Costs of Technical Debt

The costs of technical debt are significant. According to SEI, "For systems on which software architectural quality has been allowed to degrade, especially in its modifiability and maintainability dimensions, dealing with the stream of continual changes becomes increasingly less cost-effective, as more and more effort is required for comprehending and sustaining the system, leaving fewer resources for implementing new capabilities. This results in cost and schedule slippage or a diminished ability to field new capabilities." [19]

From a 2016 Information Technology House of Representatives sub-committee report, "The federal government spends the majority of its $80 billion technology budget on maintaining and operating legacy systems." [20] While legacy programs are not inherently bad, there is significant technical debt that is inherited in those programs that must be dealt with just to keep the programs operating. "Money, time and manpower that are devoted [to these efforts]… are unavailable for other efforts, and this crippling debt can impact agency performance and jeopardize the success of IT modernization. The key to successful modernization is paying off technical debt by automating outdated workflows and processes…" [21]

To quantify the direct manpower costs associated with maintaining a software system, figure 3 shows a portfolio analysis from a $1B firm with over 1000 developers. Code quality is shown on the y-axis, with file hygiene improving as you near the origin. In this research, code quality relates to how well the individual parts within the system are constructed. [22] This is specifically focused on how complex the *internal* structure of the file is. Design quality is shown on the x-axis, with architecturally simple code being towards the origin and architecturally complex code moving away from the origin. In this research, design quality is defined as how well individual parts are assembled architecturally, or whether attributes such as modularity or hierarchy have been degraded. [22]

Figure 3 - The Cost of Operating in Various States of Code Health [23]

The empirical data shows that the manpower required to develop, de-bug, and deliver a new feature within a system with healthy code and architectural characteristics is significantly less resource intensive than doing so in a system with poor code and architectural health. [23] While this chart only captures the software systems within one firm, it is effective in characterizing the differences in development time between healthy systems and those with degraded code and architectural health. For systems that have both poor code and design quality, developers may spend upwards of 69% of their time de-bugging. This is in stark contrast to the amount of time developers spend de-bugging in a healthy architecture, which is estimated at around 20%. These differences in productivity can be attributed to the interdependency and complexity inherent in the code, forcing developers to chase the errors that have been propagated through the system.

2.5 Implications of Technical Debt within the Air Force

> *"The Defense Department's approach to software acquisition trails industry standards. Of major Air Force acquisitions exceeding their original cost baselines, the majority (five of nine) are software developments… There are specialized technology cells… that are brought in to apply modern software development to struggling programs. But that is not a sustainable solution. Reforming software acquisition is a top priority for me and the Air Force." - Dr. William Roper, Assistant Secretary of the Air Force for Acquisition, Technology and Logistics* [24]

There are multiple factors contributing to the Air Force's software woes, including overly burdensome regulations, outdated processes, lack of appropriate tools, and poor cultural practices. With all of those factors considered, it is difficult to isolate the cost of technical debt from other sources of mismanagement. In an attempt to separate this cost, it is useful to examine a program's sustainment costs relative to its development costs. Since technical debt drives long-term maintenance costs higher to reduce short-term development costs, this metric may be useful in identifying "at-risk" programs.

Using the F-35 program as an example, as of December 2017 the program had spent roughly $59.8 billion in development costs versus roughly $620 billion on operations and sustainment.[3] [25] These figures capture much more than software, however this 10:1 ratio could be indicative of systemic problems, specifically the plagued automated logistics information system and mission systems software. To make matters worse, the lifecycle sustainment cost of operating the fleet of Joint Strike Fighters has been estimated at over $1 trillion, largely deemed unaffordable by critics. To meet Congress' 10% year-over-year cost reduction goals, a focus on technical debt and architectural health would provide significant return on investment.

[3] Calculated in 2012 dollars

The quotes that follow are direct excerpts from senior leader interviews with the media that speak to the dramatic effect of technical debt and other software mismanagement practices have had on large weapon systems. From the F-35 Program Executive Officer (PEO):

> *"…Air Force Lt. Gen. Chris Bogdan discussed a range of issues affecting the Pentagon's biggest weapons program at nearly $400 billion, including the hundreds of lingering deficiency reports, or DRs, known as "technical debt" in acquisition parlance. There are 419 things that we have yet to decide with the war fighters how we're going to fix them, whether we're going to fix them and when we're going to fix them. The figure was three times higher a few years ago and "we think the technical debt that we have -- the deficiencies that we have -- are things that we can handle."* [26]

It should be noted that in this instance, the term technical debt is being used to describe as a backlog of deficiencies that were discovered in lab and flight testing rather than technical decisions that were made in the development the F-35 source code. From the context of this research, this alternate definition highlights the misuse of the term in Air Force circles. Technical debt may be a causal factor in the deficiencies being present, however, it is likely not the deficiencies themselves.

From Elizabeth McGrath, the Department of Defense Deputy Chief Management Officer, on the cancellation of the Expeditionary Combat Support System (ECSS):

> *"For the United States Air Force, installing a new software system has certainly proved to be a wicked problem. Last month, it canceled a six-year-old modernization effort that had eaten up more than $1 billion. When the Air Force realized that it would cost another $1 billion just to achieve one-quarter of the capabilities originally planned — and that even then the system would not be fully ready before 2020 — it decided to decamp…. ECSS was restructured many times, including three separate times in the last three years. Each time, we chunked it down, breaking it into smaller pieces, focusing on specific capabilities. But this was not enough to save the system, because program managers did not succeed in imposing the short deadlines of 18 to 24 months that the department now requires for similar projects. Tight deadlines will certainly go a long way toward avoiding future billion-dollar fiascos. But much more needs to change before the department's older software systems can be replaced."* [5]

And finally, from a US Air Force spokeswoman speaking about the recent cancellation of the AOC 10.2 program:

> *"The U.S. Air Force has terminated a contract with Northrop Grumman for a network upgrade for the Air Operations Center, a key tool used by the service to plan and conduct air operations, and instead will partner with the Defense Department's innovation unit to find a quicker way to field the update. … Estimated development costs had ballooned from $374 million to $745 million and the program had slipped to more than three years*

late. Going forward, the service intends to start an "AOC Pathfinder" project that will use an agile software development technique called DevOps – short for development operations – to help the service continuously upgrade the system's capabilities… The AOC Pathfinder approach implements industry best practices … shrinking release cycles from years to weeks." [27]

These three examples highlight the cost, schedule, and performance issues that the Air Force has been having with software-intensive systems. In each case, the theme of schedule pressure has exacerbated the poor performance of the program, with long-term health getting deferred until the effort had become untenable.

In addition to schedule pressure, there is Congressional pressure for DoD software costs to go down in conjunction with commercial software development costs. According to Besselman, "You have a Congress that expects IT costs to decline because the average cost of an IT/Software capability is declining commercially as we increasingly use libraries and commoditized cloud-based services. The problem is we never see this decline in the DoD, because the warfighter's demand for new IT/software capabilities is increasing at a higher rate and they want only new capabilities. The acquisition offices exacerbate the situation further with their declining rate of IT/software expertise, leading ultimately to the most harmful practice of buying only new features and not allocating some percentage of an investment to system or weapon system hygiene and the reduction of technical debt over time." [18]

While the high-profile failures within the Air Force garner a majority of media attention, there are exponentially more programs which have flown below the radar that have just as much technical debt, and are performing just as poorly, only on a smaller scale. The case study section provides additional data to substantiate this claim. This research shows that the Air Force must do something to address the crippling amounts of technical debt within the acquisition system. Whether the root cause is technical, contractual, or cultural in nature, it is costing taxpayer's millions of dollars in development and sustainment costs.

2.6 Methods of Resolving Technical Debt

There are several ways in which technical debt can be dealt with. This thesis does not address each method in detail, however this section will provide a brief overview of each concept. It should be noted that the first step in achieving any of these techniques is to first identify and quantify technical debt within a software system. [28] As such, the research presented in this paper is pertinent to achieving any of the methods in this section.

At a high level, there are four methods to address technical debt within a development environment. Figure 4 shows these methods in terms of the cost of debt versus the need

for new features. By characterizing the techniques this way, the matrix describes when each course of action would be applicable for certain business objectives.

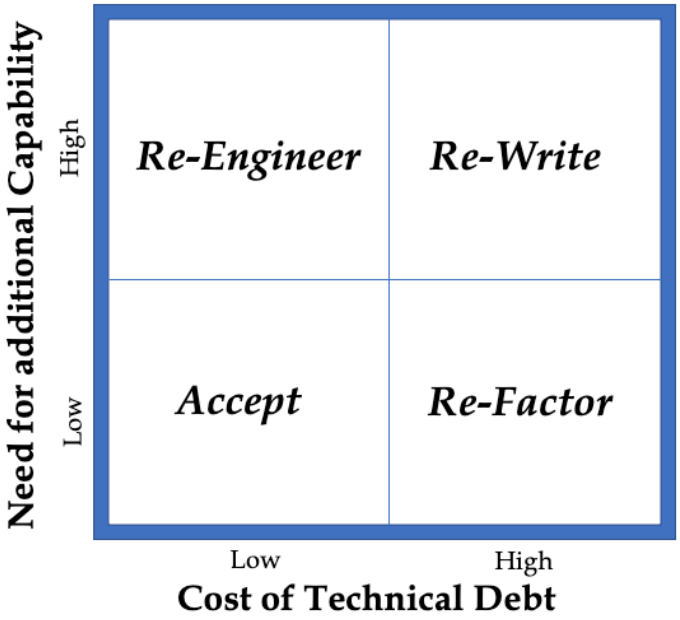


Figure 4 – Cost vs. Capability Matrix for Resolving Technical Debt

2.6.1 Re-Factoring

Re-factoring is defined as the "process of changing a software system in such a way that it does not alter behavior of the code yet improves its internal structure." [29] Re-factoring can take place at the code level or higher (architectural) levels depending on the need. From Buschmann, [30] there are three direct implications to re-factoring:

- can improve "only" developmental qualities, such as maintainability and comprehensibility of code and design;
- doesn't allow changing published contracts; and
- isn't an activity or substitute for bug fixing.

Any change that doesn't adhere to this definition is considered out of scope for a re-factoring effort. Types of changes that are not considered re-factoring include cleaning up interfaces, adding new features, fixing bugs, etc. While the scope of re-factoring is limited, Buschmann argues it holds tremendous value as, "Improper developmental quality has a direct and measurably negative impact on a system's life cycle costs: it takes longer to understand and maintain its code, and architecture drift is harder to discover. It's more laborious to test the system and chances are higher that modifications introduce bugs that are costly to fix." [30]

Re-factoring is a powerful and agile approach to maintain a system's high developmental quality. If regularly practiced, re-factoring has a positive effect on

developer habitability and system life cycle costs. [31] Yet, the light weight of re-factoring is no excuse for a quick-and-dirty development style. If too much of an iteration's duration is spent on structural gardening without adding or improving features, refactoring is hard to sell to project management. Pragmatic architects thus try to minimize the need for refactoring, which is possible by advocating a development culture that values thoughtful design decisions based on requirements and an economic, expressive coding style. [32] [33]

The power and agility of re-factoring is fueled by a strict focus on small, structural improvements limited to single system entities, paired with a rigid but lightweight quality assurance process. Unleashing the power of refactoring requires a solid understanding of its specific focus paired with disciplined practice. Otherwise, [it] can do more harm than good to a system's quality. [30]

2.6.2 Re-Engineering

Re-engineering is defined as "a systematic activity to evolve existing software to exhibit new behavior, features, and operational quality." [34] Re-engineering is distinctly different from re-factoring based on the ability to add new functionality to the software. While re-engineering is usually associated with new feature development, there are still several cases where this approach would be appropriate for the reduction of technical debt. One such example would be when short-term architectural decisions have created hard limitations on throughput, utilization, or scalability of the system. Since re-factoring is not intended to change system functionality, re-engineering would be more appropriate in this case.

Re-engineering alters the design and realization of software through a series of system-level disassembly and reassembly activities. [35] This activity seeks to evolve and maintain software assets that have provided a visible contribution to a system's business case or have a proven, significant role in an organization's portfolio. [34] Because these undertakings can be fairly significant in both scope and implications, re-engineering efforts are typically performed as standalone projects with cost, schedule and performance controls. This is in contrast to re-factoring efforts which are typically executed by the programs existing developers within the context of their normal duties. As a heavyweight gardening activity, reengineering requires strong indicators for considering its application. There are four main reasons to contemplate reengineering: [35]

- Re-factoring is insufficient for achieving the required qualities.
- Bug fixes in one place repeatedly cause bugs in other places.
- New operational or functional requirements can't be realized appropriately within the given architecture.
- The business case for the system changed.

2.6.3 Re-Writing

Often considered as the most drastic option in addressing technical debt, re-writing is defined as "replacing an existing system or component with an entirely new implementation." [34] There are several reasons why re-writing an entire system may be the best course of action, including cases where it is too costly to re-factor or re-engineer a system, or the case where an old system has reached end-of-life and a new system has already been planned to take its place. In a re-writing effort it is important to design the system using sound architectural principles (modular, hierarchical, etc.) to avoid inappropriate dependencies that burden developers with unnecessary complexity. It is equally important to track and manage technical debt during the development effort to avoid costly re-engineering efforts in the future.

A disadvantage of re-writing is that functional capabilities, both documented and undocumented, can be lost.

2.6.4 Accepting Technical Debt

Managing technical debt is ultimately a business decision. As such, there are instances where the cost of a re-factoring, re-engineering, or re-writing activity would be cost prohibitive. In those cases, leadership may make the decision not to do anything with the technical debt within the software system. This could be the case when a system is close to reaching end-of-life or when an organization does not have the requisite resources to undertake a new development effort. While this technique doesn't present the best engineering solution, it may provide the best financial solution. If this path is chosen, it is extremely important to identify and measure the amount of technical debt in the system to manage its implications properly. As such, it is just as important to understand the tools and techniques presented in this research in order to aid in the identification and quantification process.

3 Measuring Technical Debt

> *"In the acquisition of new systems, software drives program risk for approximately 60 percent of programs... Unexpected complications can arise from unanticipated interdependencies within the software itself, often driven by the underlying architecture." - Defense Science Board Report on Design and Acquisition of Software for Defense Systems, February 2018* [1]

Technical debt can be present from the beginning of a design or may accumulate over time as software functionality changes, coding practices degrade, or market pressure forces systems to be fielded without regard to long-term health concerns. In cases where system objectives change over time, developers may find themselves using areas of the code base in ways that are different from their intended use, or scaling modules in a manner that were never considered at the initial design. These types of practices result in issues within modules in the form of increased code complexity, and in the relationships between modules in the form of degraded architectural health. This section focuses on both aspects as they relate to the accumulation of technical debt. Specifically, this section explains how architectural health can serve as a proxy for technical debt, examines the architectural health assessment tools on the market, and explains how to interpret the results of the architectural health analysis tools used for this research.

3.1 Relationship between Technical Debt and Architectural Health

> *"Without a robust underlying architecture, someone working on a low-level function will be unable to understand all the end applications in which a function might be used. Therefore, the architect must try to define modules in a way that avoids cross-couplings, whereby changes in one module impact and require changes to other modules." - Defense Science Board Report on Design and Acquisition of Software for Defense Systems, February 2018* [1]

Technical debt can take many forms within a code base. It can describe a lack of documentation, lack of appropriate architectural structure, poor code quality, or lack of test coverage. These program characteristics can be caused by a number of factors, including poor coding practices, poor requirements definition, poor processes, schedule pressure, or lack of management attention. Of these root causes, this research focuses on architectural health as a primary component of technical debt and examines the relationship between architectural health, technical debt, and business outcomes.

Architectural health is a concept that is derived from an analysis of the relationships between entities in a code base. A code base that is said to have good architectural health is both modular and hierarchical in nature. If technical debt is the generic term for sacrificing long-term health in favor of short-term value, architectural debt can be thought of as sacrificing the modularity and hierarchy of the code in order to deliver

capability faster. In the sections that follow, modularity and hierarchy will be discussed in the context of how they relate to technical debt and architectural health.

3.1.1 Role of Modularity in Architectural Health

Modularity is an assessment of the degree of coupling internal to a code base. According to Baldwin, et al., "The concept of modularity is used primarily to reduce complexity by breaking a system into varying degrees of interdependence and independence across and hide the complexity of each part behind an abstraction and interface." [36] A well-designed API can mask internal complexity such that external developers need only be concerned with the inputs and outputs of the module, effectively treating it as a black box.

Within a code base, it is desirable to have loosely coupled architectures that are easy to modify or replace any individual component or service without making corresponding changes to [those] that depend on it. [37] Because the code is composed of different modules, there must be an integrative framework that allows for the independence of structure while integrating its functions. [36] Modularity presents several advantages, including:

- Increasing the range of manageable complexity
- Allowing different parts of a large design to be worked on concurrently, and
- Accommodating uncertainty

[36]

With certain architectural health analysis tools, modularity can be measured through fan-in and fan-out counts to determine how files relate to one another. In circumstances where files are both depending on and being depended on by other files, we define the interdependent relationship as a cyclical "core."

MacCormack et al. suggests that "cyclical groupings can proliferate over time as modularity and hierarchy erode, causing hundreds or thousands of source files to become mutually interdependent. In these cores, changes have strong, reinforcing ripple effects. A single change to a file can impact thousands of others, in distant parts of the system. Critically, this complexity can't be detected through inspection or code reviews. It's made visible only by tracing relationships between files across the system and its associated organizational groups… From a business outcome perspective, it is suggested that tightly-coupled core or central components cost significantly more to maintain then loosely-coupled peripheral components." [11]

3.1.2 Role of Hierarchy in Architectural Health

From MacCormack et al., "design hierarchy refers to a specific ordering of components in a system, such that dependencies flow in a uniform direction." [11] The concept of

hierarchy is useful in the "building-block" approach to system architecture, as it supports software re-use at lower levels. There are multiple approaches in designing hierarchical systems, including layering, main-subroutine, master-slave, and virtual machine. [38] [39]

If a system is thought about in terms of layers, a module that is used by every other module but is dependent on nothing would be considered at the bottom layer of the hierarchy. This means that if a change occurs to this module (i.e. a utility), then it could potentially affect every module either directly or indirectly connected to it. If a module is dependent on other modules, yet nothing is dependent on it, it is considered to be at the top of the hierarchy. If something in this module is changed, other modules are not affected.

Preserving the hierarchy of a software architecture is important to control unwanted coupling and propagation effects, topics that will be discussed in the metrics section of this research.

3.2 Architectural Health Assessment Tools on the Market

As discussed above, technical debt and architectural health are closely linked to business outcomes. As such, there are several commercial packages on the market today to assess the architectural health of code bases. The four main vendors in this space are Silverthread Inc., Lattix, SonarQube, and CAST. While each vendor analyzes architectural health in some capacity, they each have a slightly different approach with different analysis methods, plug-in tools, visualization techniques, and health dashboards.

CAST Application Intelligence Platform analyzes source code by categorizing each business function into a measurable unit. This allows for faster identification of reduced software quality, system vulnerabilities, and areas where productivity can be improved in a complex, multi-tiered infrastructure. [40] Like the other three tools, CAST works to find deficiencies at the architectural level, focusing on identifying the relationships between elements, layers, and functions. Similarly, SonarQube uses code smells[4] and other methods to detect vulnerabilities, bugs architectural deficiencies, and violations of business rules. [41] While both of these tools are powerful and useful, they do not use the visualization methods this research seeks to explore further. As such, these tools should be included in future research to assess the validity of their metrics in Air Force applications, but are excluded from this research effort.

Lattix was founded in 2004 by Neeraj Sangal. Its core product, Lattix Architect, is in use by development teams, systems architects, designers, and quality personnel in 29

[4] A code smell is a surface indication that usually corresponds to a deeper problem in the system. [70]

countries around the world. According to Lattix, their core product, Lattix Architect, is a desktop application that enables [customers] to create dependency models of [their] systems, including applications, databases, services, and configuration files. It enables customers to understand detailed dependencies of low-level elements, decompose and understand hierarchical relationships, re-engineer systems and generate work lists, create design rules allow precise specification of layering and componentization, control how 3rd party libraries are used, utilize metrics to measure complexity, stability, cyclicality, coupling and other measures, and features an open API to extend, customize and integrate into tool chain. [42]

Silverthread, Inc was founded in 2015 by researchers from MIT and Harvard Business School. Its core product offering is CodeMRI, which is further broken into CodeMRI Diagnostics and CodeMRI Analytics. According to Silverthread, CodeMRI Diagnostics provides an assessment of the design health and 'cost of ownership' of a codebase by visualizing design quality, capturing architectural health metrics (e.g., modularity, cyclicality, complexity), benchmarking against comparable systems, targeting design degradation and improvement opportunities, and quantifying business impact in terms of risk, schedule, cost, quality, agility. CodeMRI Analytics provides a customized deep dive analysis into a software codebase. It provides the ability to attack root cause of software project problems, build strategic ROI-based cases for design & quality improvement, plan technology changes & monitor refactoring progress, keep system healthy and prevent new design problems, and measure resulting maintainability, agility, & cost outcomes. [43]

In all cases, architectural health analysis tools help customers visualize the structure of, and relationships within their code base using graphs and metrics. In the Lattix and Silverthread tool sets, a specific type of graph called a Design Structure Matrix (DSM), highlights the dependencies of each software module, file, class, or entity within a code base. The metrics being calculated include assessments of modularity, complexity, and hierarchy. While nomenclature, conventions, and presentation may differ between software packages, the underlying principles are similar. The following section describes the visualization techniques (DSM) and architectural metrics used in the Lattix and Silverthread packages in more detail.

3.3 How to Interpret the Architectural Health Analysis Tool Results

Both Lattix and Silverthread use graphical and metric based outputs to convey the architectural health of a software system. This section will be broken into two parts to discuss each type of output in detail.

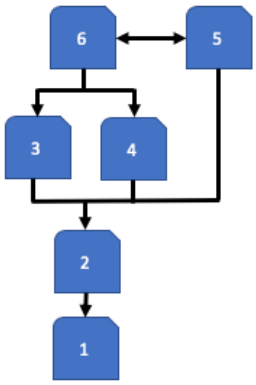
3.3.1 Representing Interdependencies Using a Design Structure Matrix

Design Structure Matrices (DSM) are visual tools that help analyze the degree of coupling within a system. This concept was pioneered by Don Steward in 1981 and

advanced by Steve Eppinger and Tyson Browning in their 2012 publication, *Design Structure Matrix Methods and Applications*. [44]

A DSM is "a network modeling tool used to represent the elements comprising a system and their interactions, thereby highlighting the system's architecture... The DSM is represented as a square NxN matrix, mapping the interactions among the set of N system elements." [44] Each entity is listed 1 through N on a row and in the same location on its corresponding column. The diagonal elements represent the entities relationship to itself and will always be marked with an X. Off-diagonal cells represent the relationships between entities and should be read as "row depends on column." As such, each column represents the output of an element and each row represents the input of an element. If a mark is present in the intersecting cell, it represents the fact that a dependency exists (Row A depends on Column B, etc.).

In Figure 5, a DSM is created from a simple graph to illustrate how it is constructed. In the graph on the left, elements 1 through 6 represent the nodes of the system and the lines represent the edges. The directionality of the arrows describes the flow of the dependency.



	1	2	3	4	5	6
Element 1	■					
Element 2	X	■				
Element 3		X	■			
Element 4		X		■		
Element 5		X			■	X
Element 6			X	X	X	■

Figure 5 - Mapping a System to DSM Form

The graph should be interpreted as follows:

- Element 6 depends on elements 3, 4, and 5 while element 5 is dependent on it
- Element 5 depends on elements 2 and 6 while element 6 is dependent on it
- Elements 3 and 4 depend on element 2 while element 6 is dependent on it
- Element 2 depends on element 1 while elements 3, 4, and 5 depend on it

The relationship between elements 1 and 2 is serial, meaning that task 1 needs to be completed before task 2 starts. This is easy to understand and is a characteristic of a hierarchical architecture. The input of element 2 is shown in the DSM as an X in row 2, column 1. The outputs of element 2 are shown in the DSM as three X's in column 2.

Elements 3 and 4 are in parallel, meaning that they can both be accomplished at the same time assuming the proper input is provided. There is no dependency between the two modules. This is easy to understand and is a characteristic of both modular and hierarchical architectures. The inputs of element 3 and 4 shown in the DSM on rows 3 and 4 respectively. The outputs of elements 3 and 4 are shown in columns 3 and 4 respectively.

The relationships between elements 5 and 6 are of concern. Because each element passes output to the other, this is representative of an interdependency. In the DSM, this interdependency is highlighted graphically by the X present above the diagonal line (row 5, column 6). In practice, if an interdependency is small enough and easy to comprehend, this may be appropriate. In the case of this example, one person may be able to manage the complexity associated with this one interdependency, however, if it grows over time it may become unwieldy to manage. These interdependencies are called "cores" and may be representative of technical debt that has accumulated over time.

The graphic in Figure 6 shows how architectural integrity can degrade over time, forming cyclicality in a code base. While anecdotal in nature, this graphic largely represents what happens in a development organization when shortcuts are taken and attention is not paid to long-term code health.

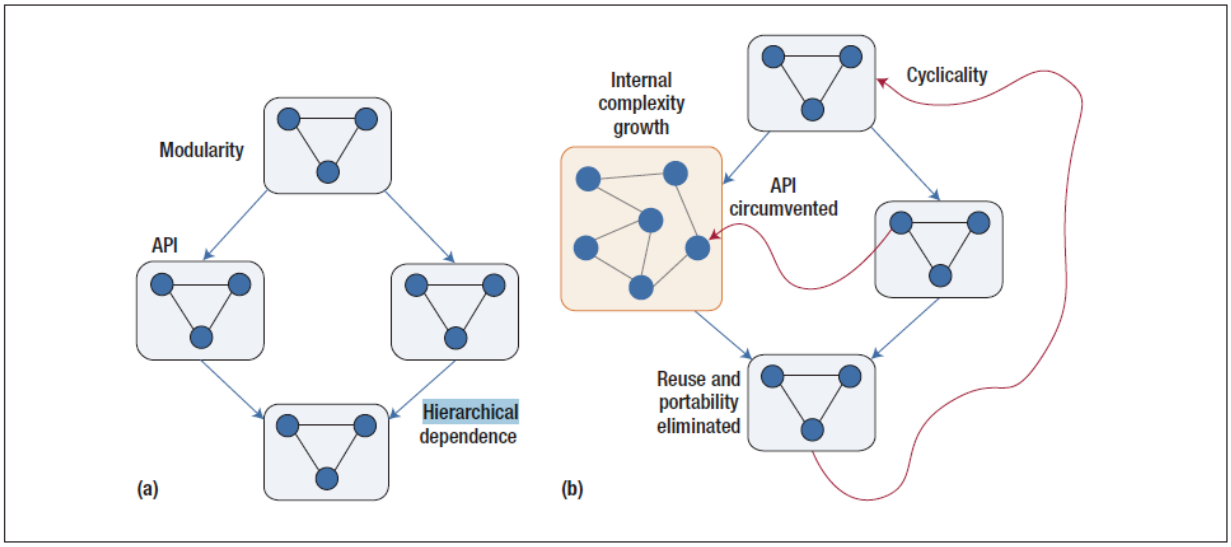


Figure 6 - Proper Architectural Health vs. Eroded Architecture Health [45]

The left-hand side of the graphic shows a modular system with proper hierarchy. Notice the dependencies flow in one direction with each module depending only on the module below it. In contrast, the right-hand side of the graphic shows the same system, but with shortcuts that have been implemented. As architectural decisions were made to call functions within other modules, the architecture has become cyclical and the modularity and hierarchy has been eroded.

While the examples given so far have been largely theoretical, cyclicality and complexity are real issues in practice. In a well-structured program, there may be interdependencies within sub-groups, but with the proper use of interfaces, API's, and control structure, a modular, hierarchical system can be preserved. Figure 7 shows a system that uses proper design principles to establish modular sub-groups, utility layers, control elements, high levels of internal cohesion, and low levels of external coupling. It should be noted that in complex systems such as this, algorithms can be used to re-arrange rows and columns based on their level of coupling and position within the system hierarchy, forming clusters of highly interdependent elements. [11] [45] While none of the data is changed in the process, the algorithms present a more human-readable visualization of a core's size.

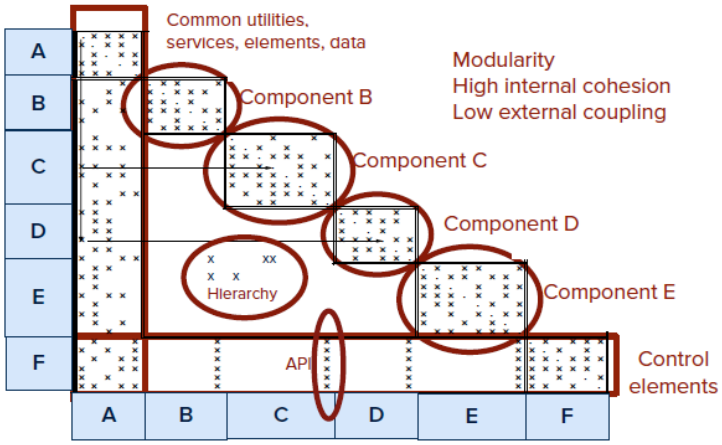


Figure 7 - Example of a Modular, Hierarchical Architecture [23] [36]

In this graphic there is high intra-modular cohesion, yet no cyclical dependencies present between components B through E, which is a sign of low coupling. Component A serves as the utility layer with common services that are used by the other components in the system. This is evident through the large population of dependencies in column A. Row F is where control elements reside with strong API layers to preserve the abstraction and information hiding principles of modularity. Finally, from the circle marked "hierarchy" it is evident that component E uses portions of component B, however there is no corresponding dependence back to component E. This proves that the system is properly re-using features that are present at lower levels in the code.

In contrast to the graphic shown above, Figure 8 shows three examples of large cores found in real-world software systems. In these figures, cores are visible due to their characteristic dependencies above the diagonal line, representing cyclical interactions between modules.

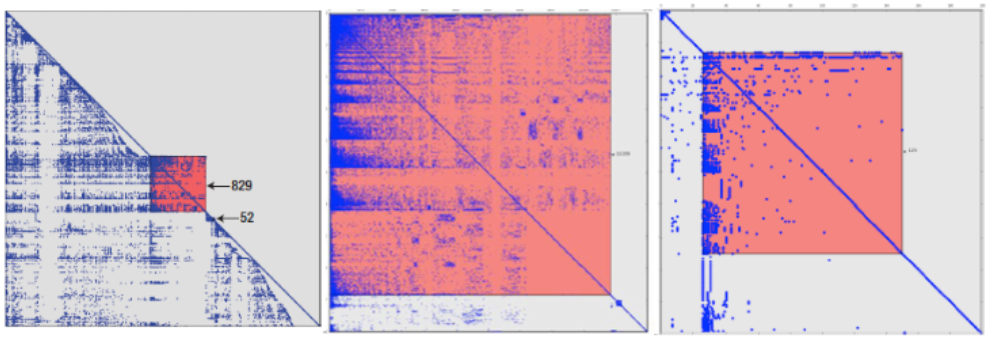


Figure 8 - Example of Large Cores in Software Systems [45]

The core shown in the center of the figure contains over 10,000 files, while the other two cores contain over 800 and 1,000 respectively. The sheer size of these cores makes it impossible for developers to comprehend the interaction between elements, much less manage them. These interdependencies inject complexity into the system which in turn manifests itself through increased cost, decreased velocity, and decreased productivity amongst project delivery teams. In essence, large cores are highly correlated with poor business outcomes.

DSM's are useful to show how a system is architected, the level of hierarchy and modularity in a system, and where technical debt may be accumulating. While the presence of a core doesn't inherently mean there are issues in a code base, there is a high likelihood that a large number of interconnected modules could reduce developer's productivity thereby increasing the cost of system maintenance. Using DSM's to visualize a system's architecture can be a powerful tool for developers and managers alike, especially when advocating for additional resources to undertake re-factoring, re-engineering, or re-writing efforts.

The architecture of a software program is where its behavior is derived, and governs its performance and value. [44] For more experienced practitioners, advanced DSM optimization techniques, clustering algorithms, tearing and banding techniques, and numerical overlays provide different visual depictions and analysis methods of complex systems. [44] [46]

3.3.2 Using Metrics to Convey the Health of a Code Base

> *"No set of metrics can provide an overall absolute quality assessment of the system; however, when analyzed together they can provide invaluable information for decision making by acquisition professional." – Software Engineering Institute Report on Recommended Practice for Application of Quantitative Software Architecture Analysis in Sustainment* [19]

Metrics are helpful in assessing the health of a code base, however, they must be chosen appropriately to ensure the right characteristics are being measured. If chosen properly, they can assess the quality of a software deliverable and project the funding and manpower allocation to complete a software development activity. [47] If chosen poorly, metrics can encourage inappropriate behavior for developers and managers alike. [47] As such, this thesis will utilize three metrics to measure the coupling, cohesion, and complexity of a code base. The merits of these three metrics are discussed in detail in the work of Selbi and Basili [48], Dhama [49], McCabe [50], and MacCormack. [11] [51] For the sake of this research, basic definitions and overviews are given below.

'Complexity' is a function of the number and nature of elements in a system. [52] It is said that the more elements a system has, the more complex it is. From a software perspective, much work has been done to quantify complexity, [50] [53] however each method has its critics on how representative these metrics are.

To quantify complexity in this research, we will use the metric of *cyclomatic complexity*. It was developed by McCabe and considers the program as a directed graph in which the edges are lines of control flow and the nodes are straight line segments of code. [50] [53] The cyclomatic number represents the number of linearly independent execution paths through the program. Figure 9 shows a graphical depiction of this concept.

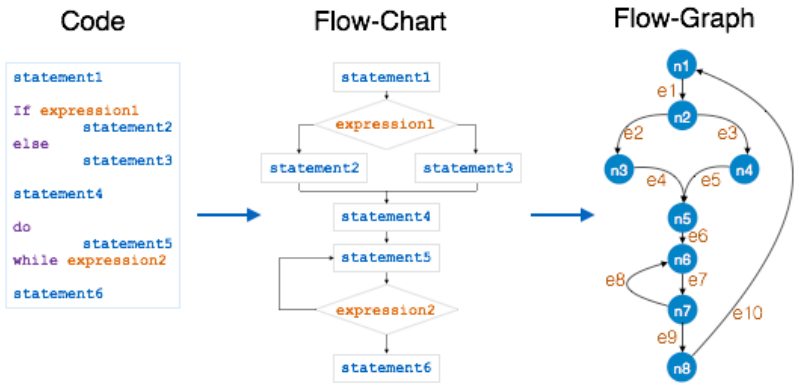


Figure 9 - Graphical Depiction of McCabe's Cyclomatic Complexity [53]

The equation for McCabe complexity is defined as:

$$McCabe\ score = \#\ edges\ (E) - \#\ nodes\ (N) + 2$$

According to SEI, "the appropriate values for many key metrics will be program dependent and context sensitive… The objective is not to set arbitrary limits for not accepting code, but to set 'trip wires' describing codebases where additional investigation and analysis is necessary." [19] As such, there is no definitive threshold for the amount of complexity in a system. As a rule of thumb, Silverthread and SEI have

set the McCabe "trip wire" threshold equal to 50, but recommend developers to take a closer look at modules that contain more than 10 independent paths. [19] [54] Files with a complexity score over this trip wire value of 50 are at risk of having insufficient test coverage to test each path in the code, possibly leading to latent defects in the system along with their hidden costs.

In the context of software systems, 'coupling' is defined as the level of interdependence between modules. [55] In systems with low levels of coupling, a system is determined to be modular, allowing modules to be removed and replaced easily. High-levels of coupling, or coupling with very strong connections is less modular, usually resulting in higher levels of complexity.

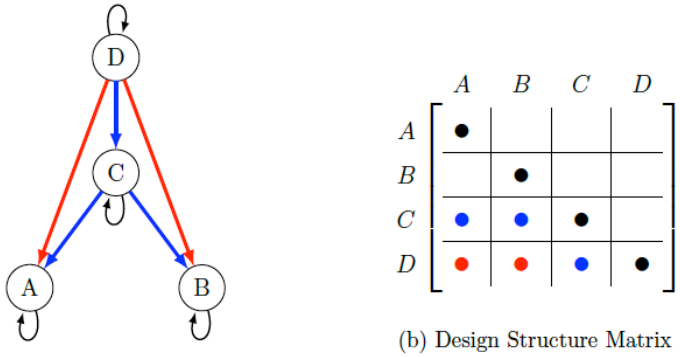
'Cohesion' is defined as the degree to which the elements inside a module belong together. [56] In a software system, it is advantageous to have highly cohesive modules due to their understandability, reusability, and maintainability. [57]

Architectural Cyclicality is a metric that builds off the concept of coupling. It represents the circular or bi-directional dependencies among modules that form tightly interconnected areas. [19] In the context of this research, these interconnected areas are referred to as "cores." In general, the larger the core, the harder it will be to manage. This will in turn increase the cost of developing code within the core and increase the manpower required to develop new features. In this research, both the number of cores and size of each core will be used as metrics for the health of a code base.

Propagation cost is another metric that builds off of the concept of coupling. It measures how many files in a system will be affected with a change to one file, and is therefore a proxy for modularity. This metric is calculated using the visibility fan-in and fan-out counts [5] from individual elements by dividing the number of direct and indirect links in the graph by the total number of possible links. [11] [58]

Figure 10 shows a simple system with both direct and indirect dependencies, both of which are used in this calculation. [11] [51] [17]

[5] Fan-in is a measure of how many other nodes depend upon it directly. This metric is computed by counting the number of links down the column (including the diagonal link) of the first-order dependency DSM. Visibility Fan-In counts the indirect links as well. Fan-out is a measure of how many other nodes it directly depends upon. This metric is computed by counting the number of links across the row (including the diagonal link) of the first-order dependency DSM. Visibility Fan-out counts the indirect links as well [11] [17].



(a) Simple Network

(b) Design Structure Matrix

Figure 10 - How Direct and Indirect Dependencies are used to Calculate Propagation Cost [17]

In this graph, the blue dots represent direct dependencies while the red dots represent the indirect dependencies. It is evident that a change to node A directly affects node C and indirectly affects node D. In this example, every node depends on nodes A and B. The graph, called a visibility or transitive closure matrix, is created by summing the direct dependency DSMs of all elements into one DSM. [17] As such, the matrix shows the level of connectivity throughout the entire system. The intent of the metric is to translate the level of connectivity of a code base into economic outcomes associated with future development efforts. The range of the *propagation cost* metric is between 0 and 1 (0 and 100%), with higher numbers representing a more tightly coupled system.

These are definitions critical to understanding the case studies presented in this research. Additionally, *cyclomatic complexity, architectural cyclicality,* and *propagation cost* will be the primary metrics used to assess the quantitative health of a code base. This in turn will help determine the "cost of doing nothing" with respect to the technical and architectural debt that has accumulated over time. It should be noted that these metrics are being derived from a single set of tools and therefore subject to systemic bias, [59] however, for the sake of this research this issue will be neglected.

3.3.3 Projecting the Costs of Technical Debt

One of the objectives of this thesis is to quantify the programmatic and economic impacts of technical debt within the Air Force software enterprise. To this point, research has been centered on the definition of technical debt, driving factors in how technical debt is accumulated, and metrics on how to measure technical debt. This section will focus on how the output from architectural health analysis tools is mapped to its associated financial outcomes, ultimately quantifying the "cost" of technical debt.

As described in US Patent 0235569, published by Dan Sturtevant, CEO of Silverthread Inc., methods can be used to link technical characteristics for either single or multiple code bases to business outcomes such as *productivity, defect density, staff turnover, growth rates, cost performance, and schedule performance.* [22] At a high level, the process of

mapping technical health metrics to business outcomes is a function of the relationship between product, process, and personnel characteristics. It has been asserted that the more complex a code base becomes; the worse the corresponding business outcomes will be. [22] [45] [58] The basic steps in this process are as follows:

- Perform a scan of the code base to determine the technical debt metrics of the system
- Capture the development activity metrics from issue tracking and version control systems
- Run statistical analyses to determine the significance of the technical metrics on the associated business outcomes of interest. This results in a calibrated model of the system
- Develop predictive analytics projecting the cost of future development efforts using the current health of the system as a baseline

To determine the existence of this correlation, millions of files from over 20 commercial code bases were analyzed to compute their *architectural cyclicality, cyclomatic complexity,* and *propagation cost*. In parallel, version control and issue tracking systems were analyzed to determine the regions of the code base had the most activity, the time associated with each commit, the amount of time developing new features versus the time spent fixing bugs, and which commits had the most downstream impacts. At the same time, the teams developing these code bases were analyzed to determine the amount of time spent per software commit, how many commits were completed per period of time (day/week/month), and how many commits were completed by file. This ultimately resulted in an assessment of how productive each developer was in each region of the code. By performing statistical analyses on these three characteristics, it was shown that a high degree of correlation existed between the complexity in the code and the associated business outcomes. [58]

For this thesis, the most relevant programmatic metrics relate to cost and schedule performance of the system. They include:

- Cost to develop a new feature [6]
- Time required to develop a new feature [7]
- Money wasted per $1M spent

These three metrics were chosen because they represent the most relevant aspects of programmatic concern within the Air Force. Cost and schedule breaches above a certain threshold are reported to Congress, therefore they are they metrics that are tracked at

[6] Defined as the cost associated with developing roughly 1,000 new lines of code, including the associated cost of debugging

[7] Defined as the time required to develop roughly 1,000 new lines of code, including the associated schedule associated with debugging

senior DoD levels. Metrics such as turnover rates are important for team leads and first line supervisors, but of less concern at higher levels.

For this research effort, access was not granted to the relevant Air Force software issue tracking, version control, and HR systems required for proper calibration. To overcome this limitation, costs were projected using calibrated models from Silverthread's database of commercial studies. As such, the data presented in the case studies is directionally correct, but not 100% accurate. Future research should make efforts to tap into Air Force issue tracking and HR systems to tune these results to each organization.

THIS PAGE INTENTIONALLY LEFT BLANK

4 Research Methods & Sample Design

4.1 Identification of Select Code Bases

For this research, an existing relationship between the Air Force and Silverthread was leveraged to gather data on the state of architectural health, and by proxy, technical debt across a specific Air Force software portfolio. Through their early pilots with the Air Force, Silverthread has successfully scanned, analyzed, and presented results on over 96 code bases. As a result, several scans for programs with significant technical debt have been presented to key senior leaders influencing decisions about the future of their programs. As presented in the case study section, some of these programs have led to successful re-factoring efforts while others have led to program cancellation or re-writes. Since this research is centered on the accumulation of technical debt within the Air Force, the collection of Air Force data in Silverthread's portfolio was influential in which cases were selected for further analysis. The five cases ultimately selected represent a cross-section of high-value opportunities for the Air Force moving forward.

Of the programs selected, senior leadership attested to the fact that 3 of the 5 were consistently over budget and behind schedule. These programmatic challenges represented deeper structural and technical issues that had not been explored previously. The first case study (Case A) was selected to showcase a program that has accumulated significant technical debt, however is advocating an organic re-factoring effort to proactively address the architectural erosion issues. By being proactive in their approach, Case A demonstrates ways in which management can interject technical tools at appropriate times in the development process to influence business outcomes prior to system failure.

Cases B and C were selected as cautionary tales of what happens when technical debt is ignored over time, ultimately forcing program re-writes and cancellations. These cases demonstrate the utility of incorporating architectural health analysis tools into the development process early (i.e. source selection/design reviews) prior to architectural failure.

Case D was selected as an example of a successful organic re-factoring effort using architectural health analysis tools within the program's development process. In situations where technical debt has already been accumulated, these types of tools can be used to improve architectural health over time, saving program managers the expense of complete re-writes.

Finally, Case E was selected as an example of a code base with proper architectural health. This case was representative of how proper controls can be implemented in a continuous development pipeline to prevent technical debt from accumulating in the first place. Case E and Kessel Run are both examples of agile-type development processes, the model the Air Force is currently striving to replicate.

While Silverthread scans have been ordered for many different reasons, the results have been useful in every case. They have influenced the re-direction of resources between management teams, the re-writing of at least four programs due to unmanageable technical debt, and through a different contractual vehicle, established a long-term partnership to monitor the health of the code during an organic re-factoring effort. Overall, the contractual vehicles between Silverthread and the Air Force have resulted in over 96 scans with several more in-work at the time this thesis is being written.

In addition to Silverthread's existing database, the author sought out opportunities to scan additional high-profile software programs that have recently been in the news. Specifically, efforts were made to establish relationships with the F-35 program and the Kessel Run program. The F-35 program has been consistently plagued with development issues, hence the interest in assessing the architectural health of their mission systems software along with their automated logistics information system (ALIS) software. Unfortunately, contractual talks stalled when we discovered that the government did not own data rights to the source code for either system.

A relationship with Kessel Run seemed promising as Hanscom AFB leadership has generally been more receptive to the use of architectural health analysis tools in the evaluation of their programs. Long-term partnerships are currently being explored, however no access to data was provided for this thesis effort.

4.2 Extraction of Data from Select Code Bases

To properly assess the health of the selected code bases, the research team first met with the customer to understand the history of the program. In some cases, functional overlays were not provided so the code was assessed without context to whether the architectural structure was consistent with the desired function. In other cases, in-depth interviews were conducted to gather qualitative feedback on the health of the code base, program performance, and delivery success rates. Statements such as "our developers have trouble working in this file," or "we've been struggling to meet our timelines" were potential symptoms for deeper underlying issues, which in conjunction with the report produced by the architectural health analysis tools, helped tell the full story of the software.

After the subject interviews, a proprietary tool[8] was run on each code base to parse the source code and the associated extract metadata. This output captured the direct dependencies between files, classes, and other objects within the code. These relationships, along with Silverthread's network analysis, form the basis for the *architectural complexity, cyclomatic complexity,* and *propagation cost* metrics. [58]

[8] *Understand* is one, albeit a core, component of this proprietary software package. It is a commercially available software package that compiles information on how functions, classes, and variables are called, creating call trees and relational metrics for further analysis. [69]

After the analysis was completed and the report compiled, the research team held a follow up meeting with the customer to review the results. In some cases, senior management was present for this de-brief. In other cases, only the PM and development teams were present. In all cases, the team reviewed the code quality metrics, design quality metrics, and the prognostic financial forecasts, articulating to the customer where the code base was healthy and where improvement was required. In some cases, leadership asked if it made sense to re-factor the code base, in other cases, decisions were made fairly quickly to re-write the code base from scratch.

THIS PAGE INTENTIONALLY LEFT BLANK

5 Case Study Findings

> *"All large-scale software-intensive systems have technical debt; whether or not technical debt is being actively managed can be a key differentiator between the success and failure of a project or system." – Data Driven Management of Technical Debt* [12]

Using *cyclomatic complexity, architectural cyclicality,* and *propagation* cost metrics, analyses were performed across Silverthread's entire portfolio of software development efforts. The full results of these scans can be found in Appendix A. To interpret these results, each result is color coded with green indicating good performance, red meaning poor performance, and yellow and orange falling in the middle of the spectrum. Figure 11 shows a subset of these programs, specifically the ones that have had systemic technical debt issues.

	Cyclomatic Complexity: Number of untestable files	**Architectural Cyclicality:** Size of largest file-file cycle	**Connectedness:** Likelihood of unintended downstream effects
Case F (Java)	101	6134	0.29
Case K (Java)	535	26526	0.92
Case M (Web)	5	258	0.15
Case N (Java)	24	211	0.02
Case O (Java)	1	588	0.42
Case U (C++)	29	243	0.54
Case V (C++)	0	0	0.27
Case X (C#)	1	73	0.17
Case X (Web)	1	11	0.25
Case Y (C++)	30	230	0.56
Case Z (Ada)	26	10898	0.87
Case Z (C++)	47	174	0.04
Case AC (C#)	0	81	0.13
Case AD (C#)	0	4	0.31
Case AD (C++)	25	260	0.17
Case AD (Web)	0	42	0.40
Case AE (C++)	146	662	0.09
Case AF (C#)	6	38	0.01
Case AF (C++)	146	671	0.09
Case AG (C#)	7	41	0.01
Case AG (C++)	145	695	0.09
Case AH (C#)	11	203	0.01
Case AH (C++)	295	2868	0.15
Case AI (Java)	21	166	0.03
Case AJ (Java)	20	166	0.03
Case AK (Web)	2	143	0.33
Case AK (Java)	0	8	0.02
Case AN (C#)	5	690	0.19
Case AN (C++)	0	34	0.05

Figure 11 – Subset of Poor Performing Scans on Select Air Force Code Bases

For the *cyclomatic complexity* metric, a program is of concern if it has 20 or more files with a McCabe score of over 50. For the *architectural cyclicality* metric, a program is of concern if it has a core of over 100 interconnected files. For the *propagation cost* metric (labeled here as "connectedness"), a program is of concern if over 10% of its files have direct or indirect connections to each other. While the root causes differ between efforts, it is evident based on this small sample that the Air Force has a problem with technical debt in at least some portions of its portfolio.

The case studies that follow are grouped into three sections:

- Using architectural health analysis tools to influence programmatic decisions
- Using architectural health analysis tools to guide a re-factoring process
- Using architectural health analysis tools within continuous development pipelines

The first section will highlight three examples of systems that have accumulated significant technical debt and now face a decision of their viability moving forward (Cases A, B, C). The second section contains an example of a program that had accumulated significant technical debt but has since undergone a successful re-factoring effort guided by data from architectural health analysis scans (Case D). Finally, the third section contains an example of a successful development effort that has maintained low levels of technical debt since its inception (Case E).

5.1 Using Architectural Health Analysis Tools to Influence Programmatic Decisions

This section highlights three cases where programs have accumulated significant technical debt over time and are now faced with decisions on whether the existing system should be sustained and re-factored or whether it should be re-written. In all three of the cases, results of the architectural health analysis scans were presented to senior leaders. In one case, there is an ongoing discussion on how to properly address the technical debt present in the system. In another case, the development process was modified to place increased emphasis on technical debt reduction (re-factor). In the last case, the project was cancelled and responsibility for a new development effort was given to a different organization (re-write).

5.1.1 Case A

Case A: The Cost of Doing Nothing		
Java	Actual	Industry Baseline
Cost to develop a new feature	$16,977	$6,195
Days required to develop a new feature	24	11
Money wasted per $1M Spent	$579,856	-
C++	Actual	Industry Baseline
Cost to develop a new feature	$7,534	$10,008
Days required to develop a new feature	12	15
Money wasted per $1M Spent	$53,322	$278,144
C#	Actual	Industry Baseline
Cost to develop a new feature	$17,136	$8,457
Days required to develop a new feature	26	15
Money wasted per $1M Spent	$583,758	$128,377

Figure 12 - Case A: The Cost of Doing Nothing

5.1.1.1 Background

Case A analyzes a program funded by two different stakeholders within the Air Force. The program started in 1999 under the oversight of a management team at Hanscom Air Force Base (AFB) in Lexington, MA. While the oversight function is inherently governmental, the development and maintenance aspects of the program executed by contractor personnel. The current development team consists of 36 developers, allocated to 3 different teams.

In 2017, a third-party contractor performed a Software Quality Assessment (SQA) that uncovered the program had significant technical debt. After the assessment, program management developed a 4-year plan to improve upon this metric. Since that time, management had started the process of improving the existing performance by baselining the health of the code base, meeting with both management and funding authorities, and deciding on whether to make changes in the development process or management team. As such, an architectural health analysis scan was ordered to assess the health of the code. The scan revealed issues in multiple areas of the code base, to include its C++, C#, and Java components.

5.1.1.2 Discussion on Technical Health

Within the Java portions of the code base a significant core was discovered (*architectural cyclicality*), containing 670 individual files. Figure 13 shows the DSM for the Java portion of Case A.

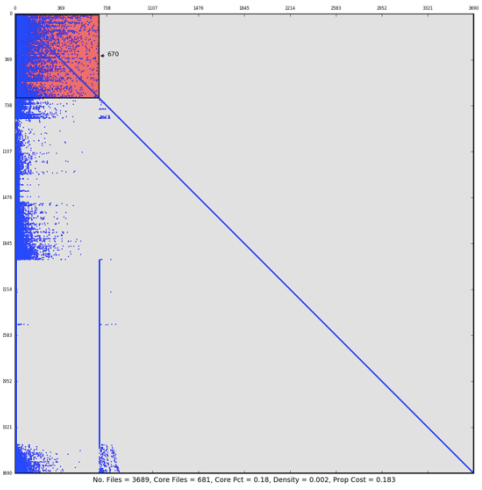


Figure 13 - Case A: DSM (Java)

In addition to the large core, there was also evidence of a severe lack of code quality control, as shown in the high *cyclomatic complexity* scores.

Complexity

System metrics

	Metric	% of Total
Files with High McCabe Complexity (> 20)	137	4%
Lines of code in these files	210524	19%
Files with McCabe Complexity > 7	1033	28%
Lines of code in these files	683346	62%

Figure 14 - Case A: Cyclomatic Complexity (Java)

Notice that over 137 files have a McCabe complexity score of over 20, and that even though those 137 files represent 4% of the code base, they make up over 19% of the lines of code. Based on these technical metrics, the "cost" of the architecture can be seen in Figure 15.

Java	Actual	Industry Baseline
Cost to develop a new feature	$16,977	$6,195
Days required to develop a new feature	24	11
Money wasted per $1M Spent	$579,856	-

Figure 15 - Case A: The Cost of Doing Nothing (Java)

To interpret this dashboard, the predictive analytics for the cost to develop 1000 lines of code, days required to develop 1000 lines of code, and money wasted per $1M are shown under the 'Actual' column. The industry baseline calculation is based on the average levels of overhead, productivity, and cost codes base with similar attributes (size, language, etc.) For the Java-based portion of Case A, it is evident that actual costs are significantly higher than industry average.

Within the C++ portion of the code base, the same lack of code quality is exhibited, specifically highlighted by one file with a *cyclomatic complexity* score of 239, which is significantly higher than the SEI threshold recommendation of 50.

Complexity

System metrics

	Metric	% of Total
Files with High McCabe Complexity (> 20)	250	13%
Lines of code in these files	511314	52%
Files with McCabe Complexity > 7	539	27%
Lines of code in these files	770961	79%

Figure 16 - Case A: Cyclomatic Complexity (C++)

Despite the lack of code quality in this portion of the code base, the architectural health is positive, displaying a hierarchical structure. Figure 14 shows the DSM for the C++ portion of this code base. It should be noted that no cores have formed, a characteristic that could predict high velocity feature development moving forward.

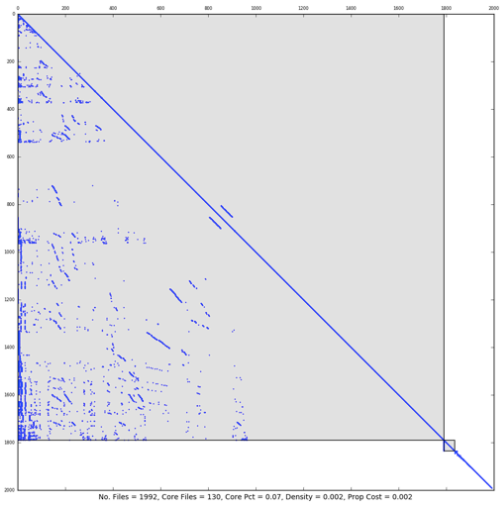


Figure 17 - Case A: DSM (C++)

Due to the degree of modularity and hierarchy in the code base, the 'cost' of the C++ portion is better than the industry average. The associated business outcomes are shown in Figure 18.

C++	Actual	Industry Baseline
Cost to develop a new feature	$7,534	$10,008
Days required to develop a new feature	12	15
Money wasted per $1M Spent	$53,322	$278,144

Figure 18 - Case A: The Cost of Doing Nothing (C++)

In the final area of the code base, the C# portion contained over 15,000 files and 2.8M lines of code. Within the C# source code there were five discrete cores of over 150 files, which is higher than most systems Silverthread has analyzed. In addition to the five discrete cores, there were 11 emerging cores of between 30 and 150 files and 42 files that had *cyclomatic complexity* scores of over 50. Figure 19 shows the DSM for this system. It clearly shows the presence of the 5 critical and 11 emerging cores, all of which have the potential to degrade developer productivity.

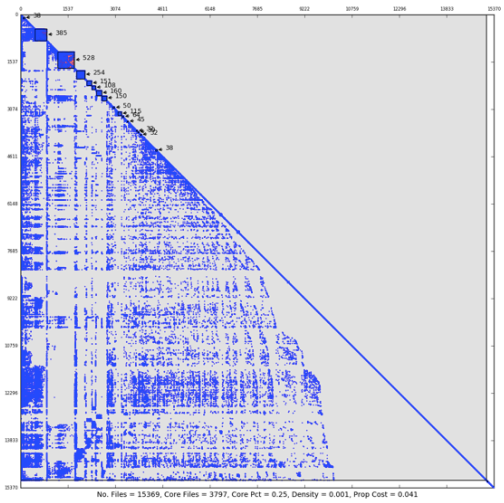


Figure 19 - Case A: DSM (C#)

For the C# portion of the code, the architectural health analysis tools were able to uncover the most problematic areas of this code base and make recommendations on how to re-factor the code to reduce both the size and number of cores in the code. It should be noted that due to the unwieldy nature of the code, the re-factoring tool took upwards of 98 hours of continuous running to converge on a solution, an amount of time that is unprecedented in Silverthread's previous work.

Based on the current state of the system, Figure 20 shows the projected penalties associated with "doing nothing" to address this technical debt.

C#	Actual	Industry Baseline
Cost to develop a new feature	$17,136	$8,457
Days required to develop a new feature	26	15
Money wasted per $1M Spent	$583,758	$128,377

Figure 20 - Case A: The Cost of Doing Nothing (C#)

It should be noted that based on the prognostics, the management team is currently wasting $583,000 for every million dollars spent. Additionally, the prognostics predict a cost of over $17,000 to develop 1000 new lines of code, more than double what a healthy system should cost. Finally, prognostics predict a new feature could take upwards 26 days to code based on the level of connectedness in the system. In this case, the cost of doing nothing is substantial.

In an interview with the PM, a desire to move forward with an organic re-factoring effort was expressed. In order to concurrently satisfy customer requirements while performing the requisite infrastructure improvements, a plan to balance the two was devised. In this plan, 20-30% of the development team's time would be dedicated to reducing technical debt in the system with the remainder working to deliver new capability. The percentage would have been higher, however the idea of pausing new capability delivery was not taken well by the operational user. It was recommended that the customer be made aware of the increased velocity and reduced cost with which he will be able to field new capabilities in the future with investments in the infrastructure now. Additionally, with a highly modular code base, the PM would be able to migrate his application to the cloud more readily, one of his primary objectives for the future of the program.

5.1.1.3 Takeaways

Case A serves as an example of how architectural health analysis tools can be used to baseline the architectural health & code quality of an existing code base. In the future, it will serve as an example on how to provide developers actionable information on where to focus their efforts, provide return on investment (ROI) metrics for new feature development, influence programmatic decisions to re-factor versus re-write problematic portions, and assess readiness for cloud migration. Case A would be a great candidate to follow over time to determine how the architectural health analysis tools have helped turn a program riddled with technical debt into a cloud-native, modular, streamlined product suitable for its own CI/CD pipeline activity.

5.1.2 Case B

Case B: The Cost of Doing Nothing		
Java	Actual	Industry Baseline
Cost to develop a new feature	$37,499	$9,597
Days required to develop a new feature	55	16
Money wasted per $1M Spent	$809,792	$192,565
Java	Actual	Industry Baseline
Cost to develop a new feature	$19,054	$8,457
Days required to develop a new feature	30	15
Money wasted per $1M Spent	$625,659	$128,377

Figure 21 - Case B: The Cost of Doing Nothing

5.1.2.1 Background

Case B highlights two components that have a prominent role in a much larger, higher profile system. The larger program was at risk of termination due to its breach of Congressionally mandated funding levels. As such, portions of the system had already been discarded in favor of new code being developed in an agile environment with

flexible requirements, shorter delivery cycles, and highly modularized "apps." For the two components being studied, the contractor had not made an operational delivery of software over the course of the programs five-year development effort. As such, people were already beginning to speak about it as a 'legacy' system even thought it had not made it to the field. Management finally realized that something needed to be done to address these programmatic issues. In 2017, they brought in Silverthread to perform and architectural health analysis for both components.

5.1.2.2 Discussion on Technical Health

The results of the scans are shown below. Figure 22 shows a core of 6346 files, the largest discovered within this body of research. A core size this large makes it virtually impossible to track bugs as downstream implications propagate through the source code. It is evident why developers were having such difficulty getting releases to the field. In this architecture, neither modularity or hierarchy is established and developers were forced to deal with the technical debt of the system during almost every commit.

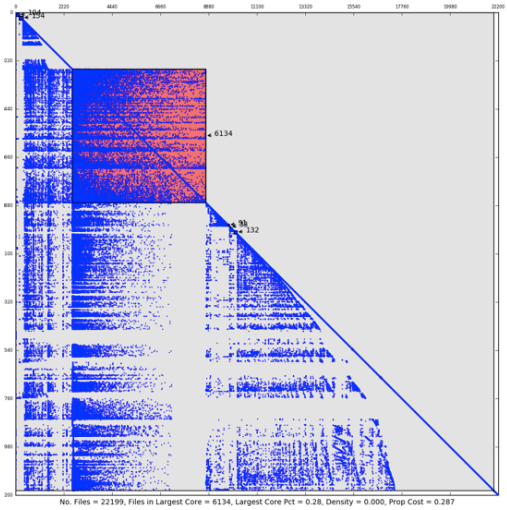


Figure 22 - Case B: DSM (Component 1)

This high degree of *architectural cyclicality* paired with the cyclomatic complexity issues shown in the dashboard below (101 files with *cyclomatic complexity* scores of over 50; 800 files with *cyclomatic complexity* scores of over 20) had serious implications on the business outcomes for the system. For component 1, the "cost of doing nothing" to address the technical debt in the system was a throwaway cost of over $800,000 for each $1M spent. In addition, comparing the costs to develop 1000 new LOC we find that this system projected a cost of $37,499 versus the industry baseline of around $10,000.

Figure 23 - Case B: Technical Dashboard (Component 1)

Component 2 did not fare much better than its companion. In this portion of the code, predictive analytics suggested that $625,000 was being wasted for every million dollars spent. Additionally, the predictive analytics forecasted a cost of over $19,000 to develop 1000 new lines of code, more than double what a healthy system should cost. Finally, a new feature could take upwards 30 days to code based on the level of connectedness in the system. In this case, the cost of doing nothing was so substantial that leadership decided to cancel the current program and re-write it with another team of developers.

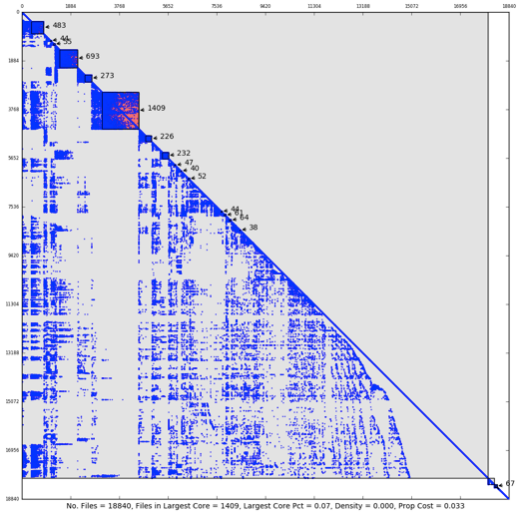


Figure 24 - Case B: DSM (Component 2)

Java	Actual	Industry Baseline
Cost to develop a new feature	$19,054	$8,457
Days required to develop a new feature	30	15
Money wasted per $1M Spent	$625,659	$128,377

Figure 25 - Case B: The Cost of Doing Nothing (Component 2)

5.1.2.3 Takeaways

Case B highlights the importance of detecting and addressing technical debt early in the lifecycle of a development program. Prior to the decision to re-write both components, this software effort had been going on for over five years. Had architectural health analyses been performed at planning and milestone events, the government would have not had to rely solely on functional test results as their primary source of performance feedback.

5.1.3 Case C

Case C: The Cost of Doing Nothing		
Ada	Actual	Industry Baseline
Cost to develop a new feature	$11,762	$10,381
Days required to develop a new feature	19	16
Money wasted per $1M Spent	$393,565	$290,623

Figure 26 - Case C: The Cost of Doing Nothing

5.1.3.1 Background

Case C analyzes a program that developed and fielded a mission critical piece of software for the Air Force. While functional overlays will not be presented in this case for various reasons, it is thought-provoking to see how a system of this importance could be allowed to erode over time.

The system was first developed in the 1970's in the programming language, Ada. In the early 2000's, leadership began reporting issues with the duration, complexity, and frustration levels associated with system maintenance and capability upgrades. These challenges were initially blamed on the contractor, then the process, then finally the development team. Changes were made over time, yet the software remained problematic. When the operational dates of the software got extended, the Air Force was forced to decide whether they wanted to live with this product for another 10-15 years, or whether they needed to do something about it. Through the contractual vehicle with Silverthread, management requested that architectural health scans be performed on the code base to support their assertions that there were underlying issues with the code base.

5.1.3.2 Discussion on Technical Health

Upon inspection of the *architectural cyclicality*, the scan revealed a core with 261 interconnected files, well above the threshold recommended by SEI. Figure 27 shows the DSM that visualizes the core size.

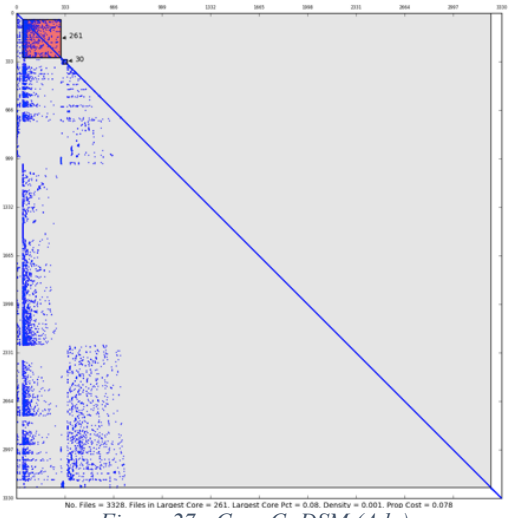


Figure 27 - Case C: DSM (Ada)

Based on the level of interconnectedness of the source code, the predictive analytics projected a cost of $11,762 to develop a new 1000 LOC feature. Additionally, the analytics projected that for every $1M spent on this code base, just under $400,000 would be wasted chasing propagation errors. There were 296 files with a *cyclomatic complexity* score of over 20, and 54 files with a *cyclomatic complexity* score of over 50. Given the legacy language the software was written in, combined with the technical debt present in the system, a decision was made to undergo a complete re-write while maintaining the existing system until the new system was fielded.

Along with the decision to re-write the software, the development agency decided to bring in one of the Air Force's rapid acquisition cells to aid in the development of more agile, architecturally sound prototypes. Starting in January 2018, the team developed a strategy to deliver working prototypes in under 2 years. In addition to the re-write of the traditional Ada code, the project took on the task of transforming two additional hardware systems into software defined systems. With the first major review upcoming in March 2019, it would be interesting to see how the architectural health of the new product compares to the legacy product.

5.1.3.3 Takeaways

This case study highlights the impact the visualization tools within an architectural health scan can have on decision-makers. This system was fielded for over 40 years, incurring cost and schedule penalties both during system maintenance and new capability development. It wasn't until 2017 when management saw these results that they decided their current course was unsustainable. In that instant, the data from the architectural health analysis tools did what 40 years of developer feedback couldn't do, convince management that the erosion of architectural modularity and hierarchy was a root cause of poor developer productivity. While speculative, it would be interesting to

see if the program would have shifted courses sooner had these types of tools been used earlier in the program's lifecycle.

5.2 Using Architectural Health Analysis Tools to Guide a Re-Factoring Process

> *"Unlike hardware, software never dies. Laying the groundwork to allow software improvement over the life of a weapons system is a strategic imperative. Utilizing development practices that enable continuous upgrade of capability ensures software can be adapted to threats and opportunities unanticipated during the specification of the system. The DoD must lay the groundwork now for software to meet the demands of the future." - Defense Science Board Report on Design and Acquisition of Software for Defense Systems, February 2018* [1]

5.2.1 Case D

Case D: The Cost Before Re-Factoring (2014)		
Java	Actual	Industry Baseline
Cost to develop a new feature	$19,501	$10,780
Days required to develop a new feature	27	17
Money wasted per $1M Spent	$634,246	$599,048

Figure 28 - Case D: The Cost of Doing Nothing at the Start of Formal Re-Factoring Effort

Case D: The Cost After Re-Factoring		
Java	Actual	Industry Baseline
Cost to develop a new feature	$9,939	$10,780
Days required to develop a new feature	15	14
Money wasted per $1M Spent	$282,367	$132,873

Figure 29 - Case D: The Cost After Re-Factoring

5.2.1.1 Background

Case D is a rich example of how to identify, quantify, and manage technical debt due to the amount of data available from its organic re-factoring effort. Due to the contractual relationship between the Air Force and Silverthread, researchers and developers have been able to collect over 10 years of architectural health data which have been used to navigate the program past the "un-managed" phase and into the "re-factoring" phase of execution. A large degree of background information was collected from interviews with the program manager, Jim Reilly, to overlay the functional, cultural, and contractual factors of the re-factoring effort with the technical metrics that have resulted. [60]

.

As opposed to the previous case studies which are all snapshots of operational software programs, Case D analyzes a program that began as an Air Force Research Laboratory (AFRL) Research and development (R&D) effort in 2000. In about 2008, transition to an AF Program of Record (PoR) began. The software was operationally fielded in 2010 and

replaced in 2015. Beginning in 2015, the same team that originally developed the code decided to use it as a demonstration of a representative AF code base that is cyber resilient and survivable under cyber conditions.

The program is Java-based with a modular cell structure that contains an underlying Oracle database. From 2000 through 2008, the capability was built through rapid prototyping with operational users in Korea and South West Asia. Initially, there were no written requirements for the system other than a single four bullet PowerPoint page. The software was used in operational exercises every three to six months, and feedback was rapidly incorporated for the next operational exercise. Developers expressed frustration at the lack of time to refactor, but the R&D budgets did not include a specific refactoring task. Refactoring and code clean-up had to be "hidden" in adding new capabilities.

In 2009 and 2010, the development team was primarily focused on transition and integration of the capability through web services into a PoR. Again, there was no budget for refactoring or code clean-up. As the product was in Test and Evaluation (T&E) through this time, refactoring could only take place under the cover of "bug fixes." Integration was further complicated by the quality of the PoR code base. It appeared that the prime contractor had been maintaining a vendor lock on that code base, in part by using high complexity to drive competitor costs higher.

In 2011 the AFRL program manager was replaced with another AFRL program manager. During the period of 2011 through 2014, little capability was added and bug fixes were limited to just patching the specific bug. As it was perceived as increasing risk of increasing additional bugs, fixing the underlying problem was not addressed. Much of this philosophy was driven by a need to get through the T&E phase as quickly as possible.

In 2015 the original AFRL program manager was assigned to develop, demonstrate, and transition cyber resiliency and survivability capabilities. The legacy code base was chosen for the demonstration vehicle. However, the high complexity of the code base limited the team's ability to develop effectively and efficiently. It also seemed intuitive that highly complex code bases could not be cyber resilient and survivable. As a result, the team returned to addressing code clean-up and re-factoring activities.

Initial refactoring efforts were guided by the developers focusing on areas of the code that were the most frustrating to work in, but not so frustrating that the developers were overwhelmed. Lattix was used to provide a quantitative assessment.

As part of a defensive cyber technical exchange between parent organizations, an acquisition agency within the Air Force offered to fund Silverthread scans of the historical code base. The results of that collaboration are shown in the data sets through 2017 and provide a quantitative measurement that matches the development team's

qualitative experiences. If it had been available in 2010, this data would have greatly helped convey the fundamental issues impacting the program and all parties could have made better decisions. While the data was useful, it became apparent that understanding how to make the code better was more valuable than accurately understanding how bad it was.

In 2018, AFRL funded Silverthread to recommend re-factoring improvements over multiple builds. The AFRL use of this new methodology led to integration of the capability in their DevOps environment. The theory was that this funding would reduce the maintenance and sustainment costs of the software. An integrated automated regression testing capability was also funded. The team used the Silverthread recommendations and the automated testing to achieve a rapid improvement in the code base. The results of this work are shown in the data sets from 2018 and on.

5.2.1.2 Discussion on Technical Debt

Figure 30 below shows the DSM for the first scan that was completed on the 2008 version of the code base. This scan was completed to baseline the state of the software when the R&D team inherited the software. The significant core size of 415 files reflects a lack of architectural health in the initial product.

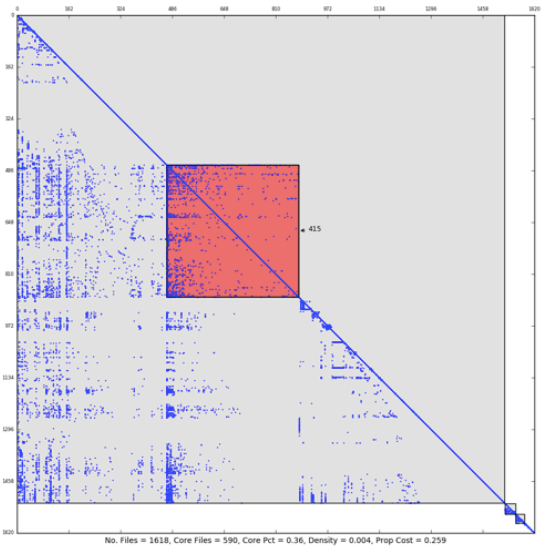


Figure 30 - Case D: DSM (2008)

If the R&D agency had chosen to do nothing at this point, the cost associated with the accumulated technical debt would have been as shown in Figure 31 below:

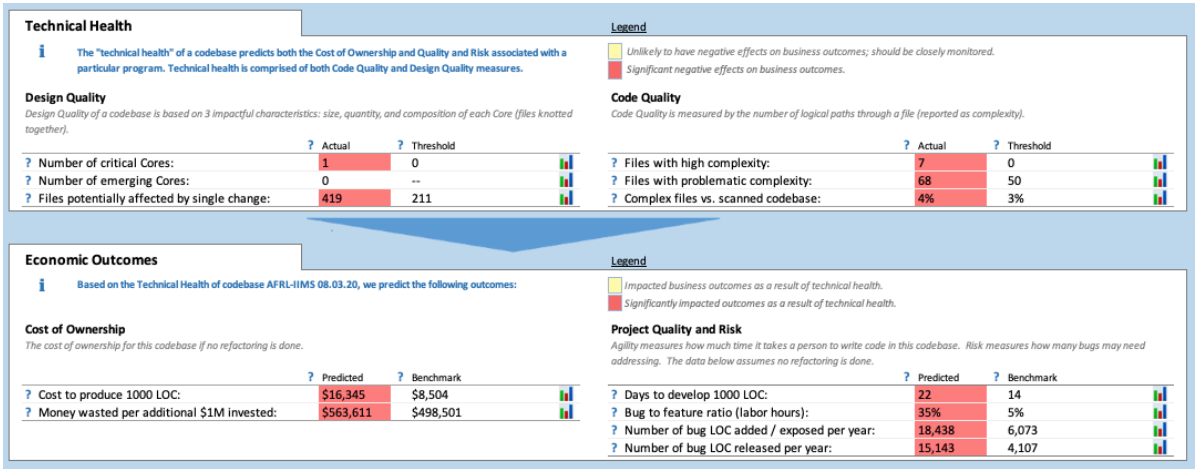


Figure 31 - Case D: Technical Health Dashboard (2008)

The economic prognostics predicted that in 2008, the management team was wasting $563,000 for every million dollars spent. Additionally, the prognostics predict a cost of over $16,000 to develop 1000 new lines of code, more than double what a healthy system should cost. Finally, prognostics predicted a new feature could take upwards 22 days to code based on the level of connectedness in the system.

As the R&D agency started their re-factoring process, they worked based on qualitative data, asking "where do we think we have the most problems?" Without the use of architectural health scans or re-factoring tools, the problem got much worse. Figure 32 shows how the core had grown during the re-factoring efforts without proper controls in place. In fact, a second core had developed during the 2013 scans. These cores were 768 files and 631 files in size respectively, which were significantly larger than the SEI thresholds.

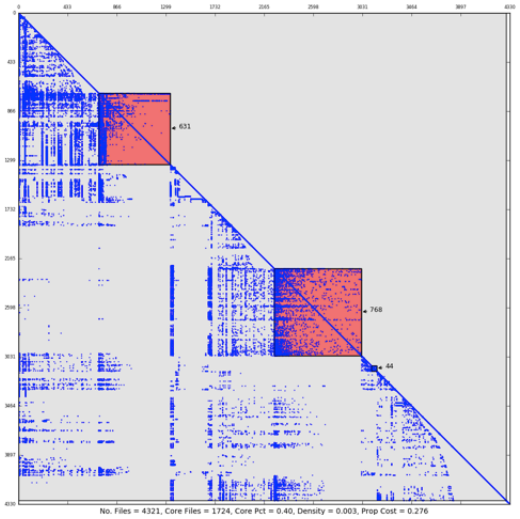


Figure 32 - Case D: DSM (2013)

Figure 33 shows the DSM representation of *architectural cyclicality* from the 2015 scans. It is evident that the cyclicality metric got incrementally better as the R&D team started

using the feedback from the Lattix scans to guide their refactoring process. The core sizes decreased from 768 & 631 files to 717 & 51 files respectively. It should be noted that while the "red core" appears bigger in Figure 33, it is actually less dense, making it easier to break apart into smaller cores using the algorithms found in Silverthread's re-factoring tools.

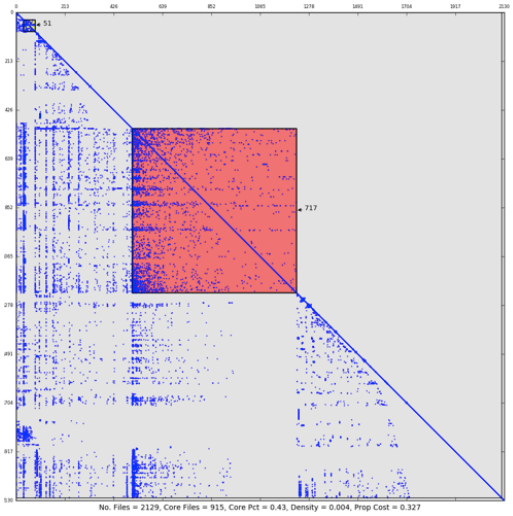


Figure 33 - Case D: DSM (2015)

This improvement attests to the fact that having quantitative feedback to guide a development process is better than not having it. With that said, Lattix did not provide the level of fidelity required to make significant improvements. When the R&D team contracted with Silverthread in 2017, major breakthroughs in architectural health began emerging within the code base. The difference was that Silverthread provided specific information in the DSM's, file lists, and re-factoring tools to show the developers exactly which linkages needed to be broken to reduce the size of the core. Figure 34 shows the state of the code base in 2017 at the start of the re-factoring effort with Silverthread. At this point the core sizes are 566 and 219 respectively.

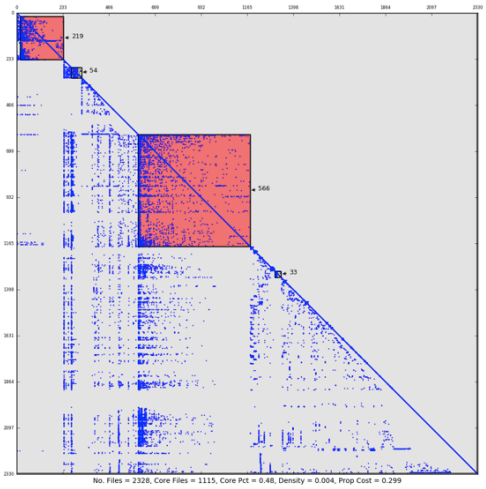


Figure 34 - Case D: DSM (June 2017)

As AFRL continued their re-factoring effort, the quantity of cores and the size of the main core continued to decrease. With the help of the architectural health analysis tools, developers were able to break interdependencies and develop new structures within the code to increase modularity and restore hierarchy to the architectural construct. In January 2019, the development team was able to eliminate all cores from the system. Figure 35 shows the final product of the re-factoring effort.

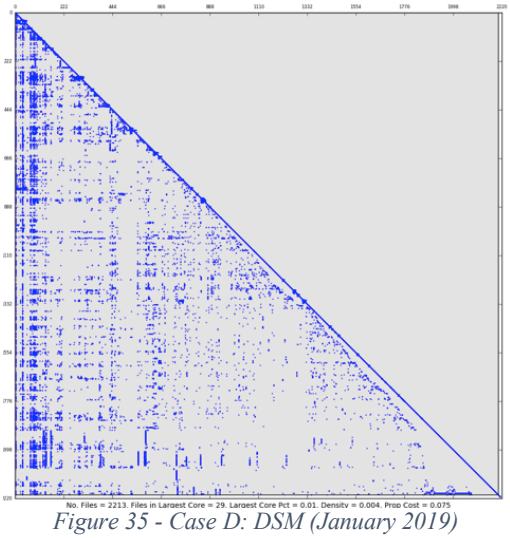


Figure 35 - Case D: DSM (January 2019)

As you can see from the DSMs, from June 2017 to January 2019, the software has reduced the number of critical cores from two to zero. The only grouping of cyclic dependencies now only contains 29 files, a drastic departure from the 400+ files in 2008, and 700+ files in 2013. This dramatic improvement is a direct reflection on what developers can do when provided with actionable information on the architectural health of their code base. Figure 36 shows the cost of sustaining the code base in its current, re-factored form.

Technical Health

Technical Health predicts both the Cost of Ownership and Quality and Risk associated with a particular program. Technical health is comprised of both Code Quality and Design Quality measures.

Legend

Likely to have moderate negative effects on Economic Outcomes; monitor closely.

Likely to have significant negative effects on Economic Outcomes; action is advised.

Design Quality

Design Quality of a codebase is based on 3 impactful characteristics: size, quantity, and composition of each Core (files knotted together).

	Actual	Threshold
Number of critical Cores:	0	0
Number of emerging Cores:	0	--
Files potentially affected by single change:	165	309

Code Quality

Code Quality is measured by the number of logical paths through a file (reported as complexity).

	Actual	Threshold
Files with high complexity:	0	0
Files with problematic complexity:	38	50
Complex files vs. scanned codebase:	2%	2%

Economic Outcomes

Based on the Technical Health of codebase CodeMRI_CTMF build-115, we predict the following outcomes:

Legend

Starting to be negatively impacted by Technical Health measures; monitor closely.

Significantly impacted by Technical Health measures; action is advised.

Cost of Ownership

The cost of ownership for this codebase if no refactoring is done.

	Predicted	Benchmark
Cost to produce 1000 LOC:	$9,939	$8,426
Money wasted per additional $1M invested:	$282,367	$132,873

Project Quality and Risk

Agility measures how much time it takes a person to write code in this codebase. Risk measures how many bugs may need addressing. The data below assumes no refactoring is done.

	Predicted	Benchmark
Days to develop 1000 LOC:	15	14
Bug to feature ratio (labor hours):	21%	5%
Number of bug LOC added / exposed per year:	6,892	4,377
Number of bug LOC released per year:	5,802	3,318

Figure 36 - Case D: Technical Health Dashboard (January 2019)

It should be noted that the cost to develop a new feature has dropped under $10,000 and the amount of money "wasted" per $1 million spent is down to $282,367, some of which is overhead that must be maintained.

5.2.1.3 Takeaways

This case study reveals several important points. First, it reveals that in one example, high level AF program management decisions made the problem worse. The ability to use cost-based metrics (e.g., cost to develop a new feature) would have helped all parties develop an accurate understanding of the problems and solutions.

Second, it reveals the fact that performing a re-factoring effort without quantifiable, actionable information is very difficult. Prior to using Lattix, the software program showed minimal change in its architectural health. In the time that the development team was using Lattix software, improvement was made, but only in the sense that the developers got information on where interdependencies existed. It was not until the Silverthread tools were used in 2018 that the developers were able to show dramatic progress in architectural health.

Third, this case study reveals that not all poorly architected code bases need to be re-written. Too often management teams have the urge to walk away from problematic code bases and start over. While this course of action may be prudent in some cases, in others it may be a better use of organizational resources to undergo an organic re-factoring effort guided by developers who are eager to use the proper feedback tools. For example, note the spike in core size that occurred on 1 Nov 2018. This low-density core was promptly detected and corrected with two changes in the code.

Fourth, the AFRL and Silverthread teams have recognized this is a "knots in your fishing line" problem. The development team made a conscious decision to only solve the easy problems. Difficult refactoring steps were skipped. Easy steps were applied throughout the code base. Like untangling your fishing line, you can't start with the worst part of the knot. Only do the easy part and the worst part gets easier with each step.

Fifth, the AFRL team has integrated the Silverthread capability into their DevOps environment. The team is allocating about 10% of the manpower in every sprint to architectural health improvements based on the recommended Silverthread steps. This ongoing investment has an immediate return in overall productivity.

Figures 37 and 38 show the macro-level trends in the technical health metrics across the entire 10-year re-factoring effort. It is evident that marked improvement was shown towards the end of the effort as developers where able to target specific dependencies and re-factor them in a more modular, hierarchical fashion. Additionally, it is evident that the cost of ownership has decreased as the architectural health metrics improved.

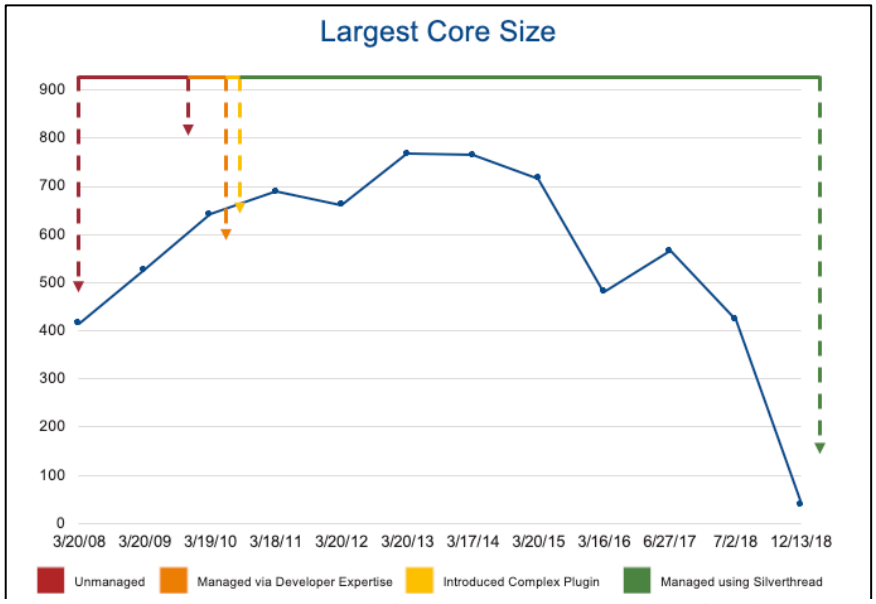


Figure 37 - Case D: Trend of Largest Core Size Over Full Lifecycle

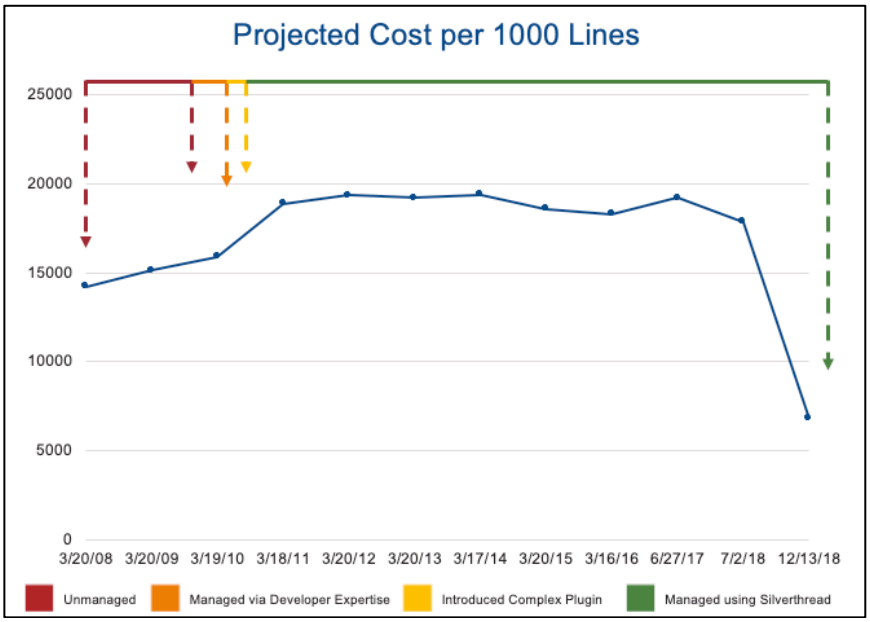


Figure 38 - Case D: Cost Trends to add 1000 LOC over Full Lifecycle

Figure 39 focuses on the significant improvement in core size and cost metrics over the last year. While the macro-level view shows how the re-factoring effort improved using one-year samples, this zoomed in view shows how the core size fluctuated monthly, or even daily, with each new attempt to break the core apart. In some cases, the re-factoring effort worked. In other cases, the re-factoring effort formed new interdependencies that needed to be broken. Ultimately, the developers were able to reduce the core size to under 30 files using a piece-meal approach and the actionable information provided by the Silverthread tools.

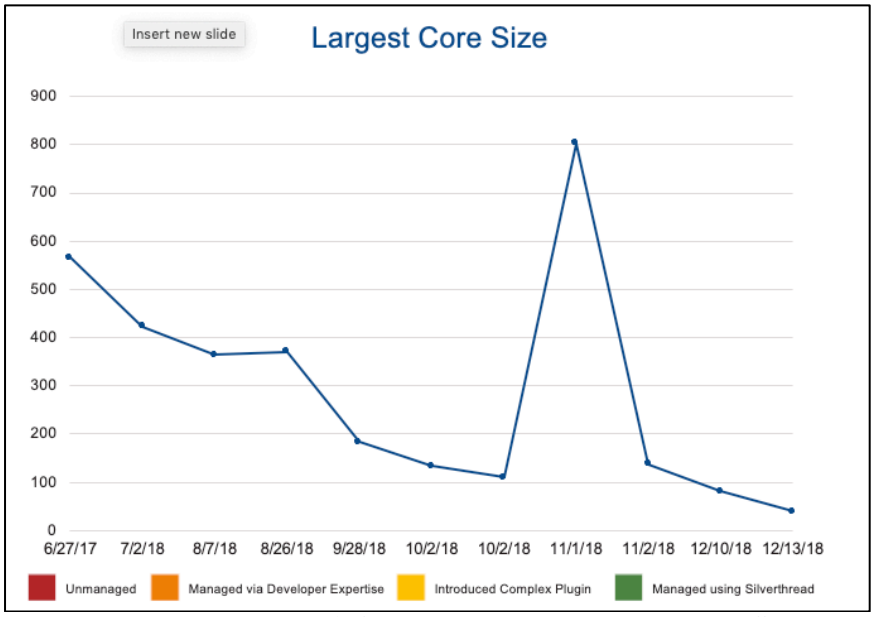


Figure 39 - Case D: Trend of Largest Core Size Over Re-Factoring Effort

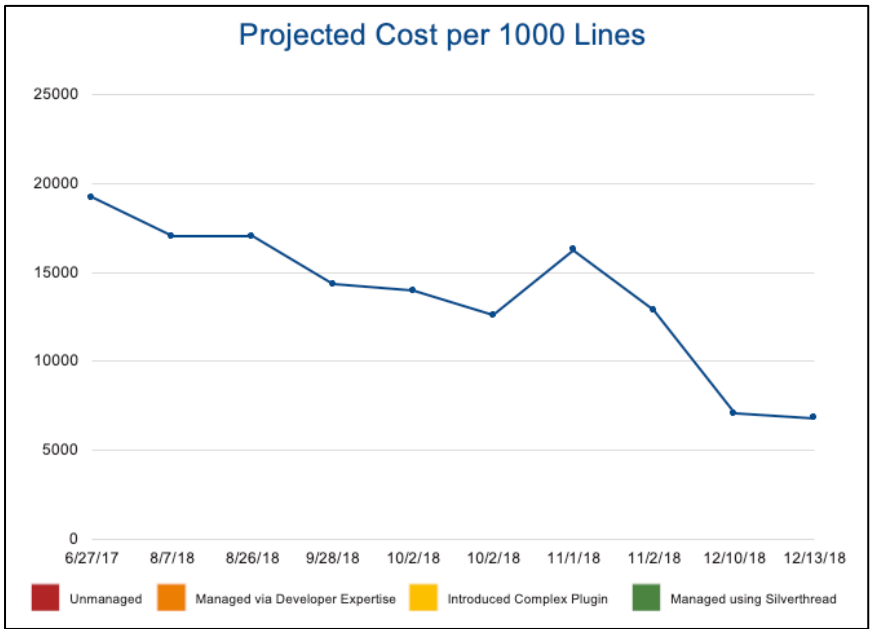


Figure 40 - Case D: Cost Trends to add 1000 LOC over Re-Factoring Effort

In addition to the metrics shown above, this effort also produced tangible evidence of a reduction in technical debt. Over the 10-year re-factoring effort, the PM was forced to keep large amounts of documentation to provide developers insight into the intricacies of the software program. While documentation is a foundational part of programming, a separate book should not be required to capture the unique propagation effects of modifying code in certain areas. As a result of this effort, the code became less coupled and more modular, reducing the total page count of this design handbook by roughly 75%. The labor mix of the team has also changed. When the code was highly complex,

it was very difficult to bring on a new, less experienced member of the team. With the improvements, new team members can quickly and effectively operate in all areas of the code base.

Based on the improvement in architectural metrics, cost of ownership data, and reduction in documentation, it is evident that the R&D organization invested wisely to reduce the amount of technical debt in its system.

5.3 Architectural Health Analysis Tools Within Continuous Development Pipelines

> *"The United States must have the ability to quickly respond to adversary advancements and update our systems accordingly. Rapid and continuous software development will be essential to achieving this outcome." - Defense Science Board Report on Design and Acquisition of Software for Defense Systems, February 2018* [1]

5.3.1 Case E

Case E: The Cost of Well-Constructed Code		
Java	Actual	Industry Baseline
Cost to develop a new feature	$5,852	$8,646
Days required to develop a new feature	11	15
Money wasted per $1M Spent	-	$158,972

Figure 41 - Case E: The Cost of Well-Constructed Code

5.3.1.1 Background

Case E analyzes a program used by the intelligence community. The program started over 10 years ago under the oversight of a management team at Hanscom Air Force Base. While the oversight function was inherently governmental, the development and maintenance aspects of the program have been executed by three separate contractors since the inception of the project. Contractor A released 3 major versions of the code base in addition to many minor version updates. Contractor B took over the development effort around 2010 and eventually transitioned the code base to Contractor C, a startup comprised of members from Contractor B. Contractor C has since released one major version with several minor revisions. The transition from Contractor A to Contractor B (and then Contractor C) was made to eliminate the technical lock-in resulting from the proprietary nature of the code base. As one of the requirements of their contract, Contractor C was required to develop this code base using open source, non-proprietary sources and methods. This case study will analyze the architectural and code health of all 4 major revisions (v1.3, v2.0, v3.0, v4.5) in an effort to understand how the code base evolved over time, especially during the transition between contractors.

5.3.1.2 Discussion on Technical Health

The series of DSM's in Figure 42 show how the architecture has evolved over the course of the four major versions. Of the case studies in this thesis, this is the only example of a program that has demonstrated positive architectural health from start to finish. While personnel, processes, and functionality have evolved over time, this is a good example of how architectural controls can be used within a continuous development pipeline to deliver more functionality to the customer, at a faster pace, with lower cost to the government.

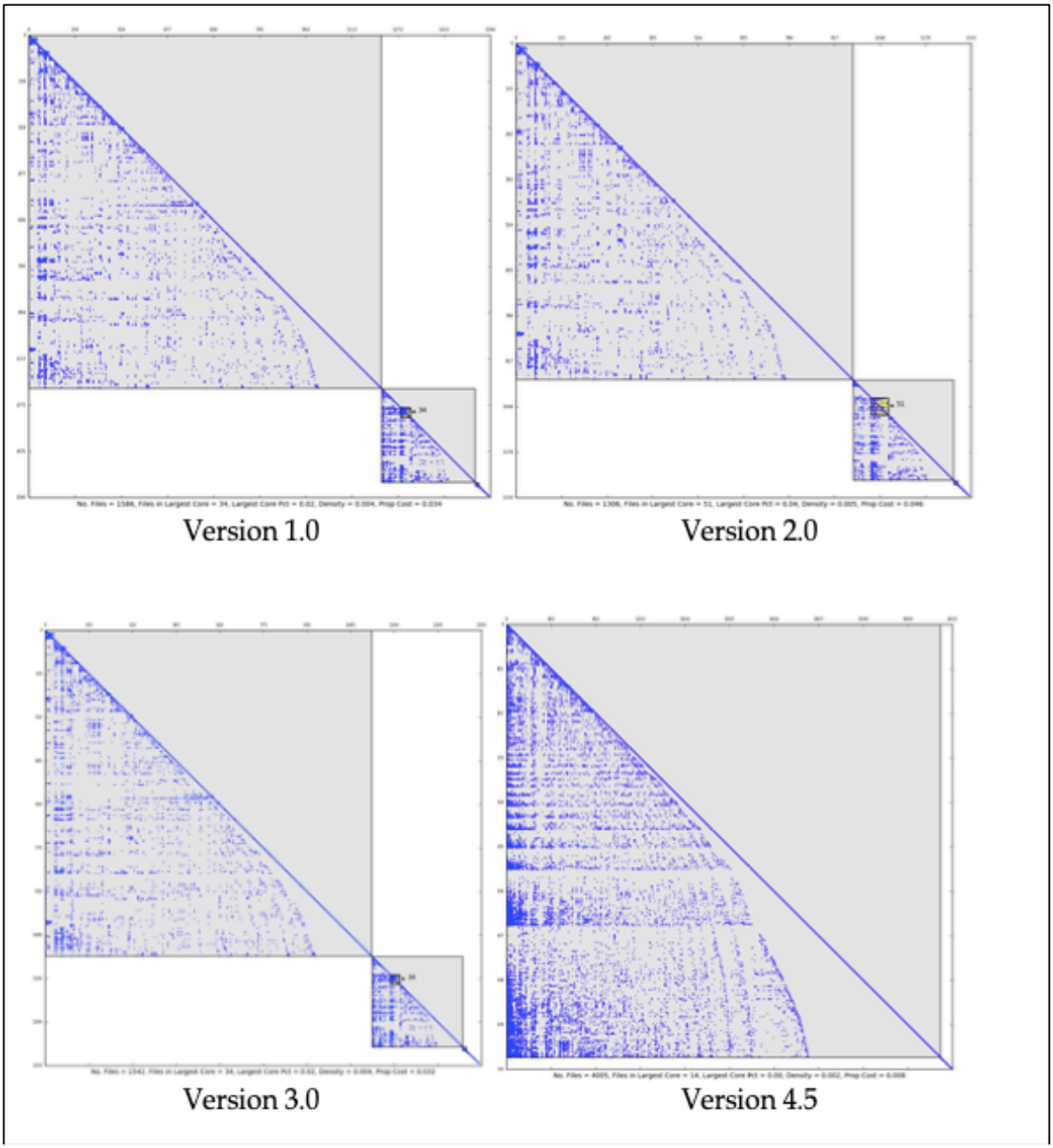


Figure 42 - Case E: DSMs across 4 Version Releases

It should be noted that there are no major changes in architectural health between v1.3, v2.0, and v3.0 with the exception of a small core shrinking from 51 files to 34 files in the auxiliary code base. After discussion with the development team it was discovered that this second code base was a modified COTS installer product that was included with the early releases. In versions 4.0 and beyond, this installer was not used. It should also be noted that there are no major changes in *architectural cyclicality* between v3.0 and v4.5, despite a considerable increase in size of the code base. This indicates that developers placed a high emphasis on modularity in their development process.

The remainder of the discussion on Case E will be focused on the transition between v3.0 and v4.5, as this was where the contract changed along with the associated development team and management components. As stated above, the government's rationale for the change of contract was not due to technical incompetency. Rather, the objective was to move from a proprietary code base to one based in open-source code. By moving away from a proprietary code base, the effects of "technical lock-in" would be removed, hence saving the government money.

During this transition, personnel, processes and product health were assessed in an effort to categorize what should be changed and what should be kept. From a personnel standpoint, several of the existing developers remained on the project, while new management and oversight were brought in. Due to the developer's familiarity with the code base, assessments on which portions to keep and which portions to re-write were fairly straightforward. While the architectural foundation was sound, with the exception of one parser and the Core API's, the proprietary nature of the code base forced a re-write from the ground up.

In the new build effort, it is evident from the architectural scans of v4.5 that the new, open-source code base was designed to be modular. The need for modularity was underscored in discussions with developers when they stated their goal is to produce an enterprise product with a core functionality and customizable plug-ins based on user affiliation, functionality, and standard operating procedure. The highly modular structure enabled the developers to customize the desired the plug-ins to meet each site's individual needs. In this sense, the structure of the code directly mirrors its need for tailoring. It is no surprise that the developers have placed so much emphasis on architectural health.

During the development process, the team was also able to build security into the architecture and focus on a flexible design that allowed for the scaling they knew they would need. This solid foundation allowed them to develop new capability at an impressive pace, resulting in a much larger and much more capable code base. It is this growth in functionality that has allowed the 300% increase in size from v3.0 to v4.5 while still enabling the code base to maintain positive architectural health metrics.

Given the current health of the system, the financial implications of developing new content are positive. The figure below shows the financial prognostics:

Technical Health

Technical Health predicts both the Cost of Ownership and Quality and Risk associated with a particular program. Technical health is comprised of both Code Quality and Design Quality measures.

Legend

Likely to have moderate negative effects on Economic Outcomes; monitor closely.

Likely to have significant negative effects on Economic Outcomes; action is advised.

Design Quality

Design Quality of a codebase is based on 3 impactful characteristics: size, quantity, and composition of each Core (files knotted together).

	Actual	Threshold
Number of critical Cores:	0	0
Number of emerging Cores:	0	--
Files potentially affected by single change:	32	565

Code Quality

Code Quality is measured by the number of logical paths through a file (reported as complexity).

	Actual	Threshold
Files with high complexity:	2	0
Files with problematic complexity:	17	50
Complex files vs. scanned codebase:	0%	1%

Economic Outcomes

Based on the Technical Health of codebase DIB 4.5, we predict the following outcomes:

Legend

Starting to be negatively impacted by Technical Health measures; monitor closely.

Significantly impacted by Technical Health measures; action is advised.

Cost of Ownership

The cost of ownership for this codebase if no refactoring is done.

	Predicted	Benchmark
Cost to produce 1000 LOC:	$5,852	$8,646
Money wasted per additional $1M invested:	$0	$158,972

Project Quality and Risk

Agility measures how much time it takes a person to write code in this codebase. Risk measures how many bugs may need addressing. The data below assumes no refactoring is done.

	Predicted	Benchmark
Days to develop 1000 LOC:	11	15
Bug to feature ratio (labor hours):	8%	6%
Number of bug LOC added / exposed per year:	2,853	5,347
Number of bug LOC released per year:	2,196	4,115

It is evident that having an architecturally sound code base has positive financial implications, as these prognostics predict no wasted money in the development effort, along with shorter than expected timelines and costs to deliver 1000 lines of code.

From a process standpoint, Contractor A had already shifted from a more traditional waterfall model[9] to more of an agile approach prior to the transition. This agile approach continued under the new management team with several new controls being implemented to reduce technical debt, code defects, and security vulnerabilities to the maximum extent possible. These new controls included the implementation of OSGI, which enforces the better parts of Object-Oriented Design, including an emphasis on modularity and compartmentalization. Additionally, the new contractor team implemented some of the more standard controls in today's development environment, including running builds on changes prior to integrating them into the baseline, 2-personal manual code reviews, automated unit and integration testing, and vulnerability, security and code-quality scans. From a test coverage perspective, they were looking for >75% coverage. This metric is achievable due to their low complexity scores from the architectural health analysis tool scans. If files scored poorly on the *cyclomatic complexity* scale, it would be highly unlikely that they would be able to get the threshold amount of test coverage on their unit or integration testing.

[9] In legacy programs, waterfall methodology is an abstraction for defining all of the requirements in the beginning of the program, undergoing a lengthy decomposition process, developing the parts of the system, integrating the system, testing the system, and fielding the system. This process takes years to transition from requirement to a final deliverable, which in our current software environment is unacceptable.

5.3.1.3 Takeaways

Case E serves as an example of how architectural health analysis tools can be used within a continuous development pipeline to baseline the architectural health of a code base, monitor it over time, and set proper bounds on programming practices that allow for the integrity of the architectural health as functionality expands over time. While tools were not injected into the development pipeline for this specific software application, they did serve to provide the development teams with quantitative feedback on the health of their product after the version had already been released. Instead of limiting the role of architectural health analysis tools to post-mortem analysis, the goal would be to insert these tools into the build and test cycle to get immediate feedback back to the development teams. The sooner the teams get feedback, the sooner they can make the appropriate changes to preserve the integrity of the code base, reducing downstream implications.

THIS PAGE INTENTIONALLY LEFT BLANK

6 Discussion and Synthesis

6.1 Summary of Results

Figure 43 consolidates the architectural health metrics from each case study into one chart. Cases A, B, and C were one-time scans while Cases D and E were scanned multiple times throughout a re-factoring effort. For the latter cases, the most recent set of technical metrics is shown.

		Cyclomatic Complexity: Number of untestable files	**Architectural Cyclicality:** Size of largest file-file cycle	**Connectedness:** Likelihood of unintended downstream effects
Case A				
	Case A (Java)	13	670	0.18
	Case A (C#)	42	528	0.04
	Case A (C++)	15	11	0.00
Case B				
	Case B (Java)	64	1409	0.03
	Case B (C#)	4	4	0.03
	Case B (Web)	28	527	0.04
Case C				
	Case C: 16 Nov 17 (Ada)	24	261	0.08
Case D				
	Case D: 10 Jan 19 (Java)	0	29	0.07
Case E				
	Case E: Version 4.5 (Web)	2	14	0.01

Figure 43 - Summary of Technical Health Metrics

The figure shows the varying magnitudes in which technical debt has affected each software development effort. Both *cyclomatic complexity* and *architectural cyclicality* values ranged from 'negligible' to 'severe'. As such, the methods used to address technical debt varied from 'acceptance' to 're-write'. Despite the contextual differences between programs, there are several common trends, factors, and characteristics that should be highlighted. First, out of the five programs examined, none incorporated architectural health analysis tools within their build environments to assess the code base's health during development. In some cases, a focus on process health overshadowed the need for product health. By focusing solely on process metrics (build times, milestone events, etc.), functional code was developed at the expense of maintainability and sustainability considerations. In other cases, lack of market awareness prevented managers and developers from integrating the most effective tools for the job. Yet in other cases, contractual limitations precluded knowledgeable

government officials from accessing or examining the source code directly. In all cases, having an objective source of data to show the erosion of architectural health within a code base could have prevented the government from discarding a system and starting over.

Second, in each case there was a desire for developers and managers to produce a quality product that met customer requirements. In cases A, B, C, and D, the developers knew there was technical debt in the system, and in all cases, they attempted to address it. However, without a way to measure or locate the technical debt within the system they were relegated to qualitative metrics and trial and error to find linkages, errors, and degraded architectural health characteristics within the code. Using the graphical output from architectural health analysis tools would help developers visualize the relationships between elements and quickly locate dependencies that need to be broken.

Third, in each case study resources were a driving factor in how the program accumulated technical debt, and also in how they addressed it. Ultimately, dealing with technical debt is a business decision that factors in the severity of the debt, the software design life, the importance of the software system, the amount of resources available, and the amount of resources being wasted. In case A, the program manager had the manpower and funding to recommend undergoing a re-factoring effort. In cases B and C, the amount of technical debt had risen to a level where a re-factoring effort was not worth the time or money required to restore proper operation. As such, re-writes were required in both cases. In case D, the organization had the time, money, and expertise to perform an organic re-factoring effort. Finally, in case E there was very little technical debt to be addressed, therefore the PM is decided to maintain the status quo of accepting the technical debt in each new release under the condition that clean-up actions were performed immediately after.

Finally, in each case study, the managers were able to make decisions quickly based on the results from the software scan. Too often, developers are not able to convince leadership to allocate resources to infrastructure improvement efforts within a code base. In an environment where customers demand new functionality and managers quickly oblige, developers rarely get to spend time performing the very functions that would ultimately increase velocity and reduce lifecycle cost. Using the output of the architectural health analysis tools, developers and managers are able to visualize the health of the code, speak the same language, and allocate resources accordingly.

It is clear that the Air Force must make changes in its acquisition processes to reduce the amount of technical debt across its software portfolio. At a minimum, technical debt reduces developer productivity and leads to degraded business outcomes. At worst, systems can degrade to the point where complete re-writes are required. All of these decisions cost money, hence the title of this research. With billions of dollars being spent on software acquisition within the Air Force, even a small percentage of technical debt will cost taxpayers millions of dollars. The longer technical debt flies under the

radar, the more money it will cost in the long run. By addressing the debt up front, it can be prevented and managed early in the development lifecycle, reducing overall cost.

In the section that follows, opportunities for change will be discussed. This includes a discussion on the overall climate of software acquisition and the perspective of senior leaders within the field. In the final chapter, recommendations will be given on how to implement technical debt reduction measures across the Air Force software enterprise.

6.2 Opportunities to Reduce Technical Debt in the Air Force

There are significant opportunities for integrating architectural health analysis tools into the Air Force acquisition process. First, the timing is right for pursuing new, innovative ideas within the DoD. Dr. Roper, the Assistant Secretary of the Air Force for Acquisitions, Technology and Logistics, is focused on innovation, rapid fielding, and agile development. [2] Like Ash Carter and Frank Kendall before him, he has laid out several policies that strive to improve the performance of the systems we buy and reduce the number of cost overruns we've experienced in recent years. Organizations like AFWerx and Defense Innovation Unit (DIU) have been formed to leverage high performing Air Force talent, the venture capital community, and non-traditional industry partners to develop creative solutions to complex problems. The DoD is starting to recognize where it is falling short, and attempting to integrate new ideas, processes, and tools into legacy programs. [2] [4] Congressional Authorizations have pushed aggressive reforms, and some appear to be gaining traction. [24] According to Dr. Roper, "program managers are taking advantage of new authorities to experiment with commercial technologies and use expedited contracting arrangements. [24] All of these factors together have started to create new capability delivery models which are being lauded by the end-users.

Additionally, the Air Force's newest software factory, Kessel Run, is getting significant attention in the media. [27] [61] This underscores the Air Force's interest in moving towards agile software development practices and also highlights the specific use case of where tools could be integrated into the most notorious DevOps program the Air Force currently has. According to the Air Force Chief Technology Officer, "The Air Force has been pushing for broad organizational change when it comes to adopting new technology, as evidenced by the newly launched digital program executive office to handle agile software development…you want to have [authority to operate] within three or four weeks, not six years." [61] These types of statements show that there is momentum behind the idea of creating new types of teams, governed by new types of contracts, with more agile software development processes. These three areas, in conjunction with getting the right tools into those agile pipelines, have the opportunity to significantly reduce the amounts of technical debt in software development activities.

6.3 Air Force Senior Leader Perspective

To develop the full picture of how technical debt has manifested itself in the Air Force, it is important to understand the leadership, managerial and cultural trends within the environment in which software is being developed and acquired. As stated in the introductory section, technical debt has contractual components, cultural components and technical components. In the previous section, architectural health analysis tools were used to provide insight into how developers could use a technical tool to uncover sources of technical debt, and in Case D, to guide their re-factoring efforts. In this section, an interview with the PEO Digital Chief Engineer is included to shed light on the cultural and managerial trends that have influenced the accumulation of technical debt to date, and what he intends to do to reverse those trends.

6.3.1 Interview with Steve Falcone, Chief Engineer, PEO Digital [62]

Steve Falcone has been the Chief Engineer for the PEO Digital portfolio since 2015, prior to which he was a division chief in the same organization. He likes to think of technical debt not as "the cost of doing nothing," but more as "the cost of delay." This distinction may seem trivial, however, it emphasizes the idea that it is the system itself that injects technical debt into the product through the creation of unnecessary handoff points, lengthy approval cycles, and black box deliverables from defense contractors. This idea refutes the assertion that we're doing nothing about technical debt and instead suggests that we are aware that we have technical debt but are too slow in fixing the acquisition process to remove it.

Mr. Falcone acknowledges that technical debt has been an issue in his portfolio for many years, highlighting several key factors that have led them down this path. Specifically, he highlighted the software re-use policy that was Congressionally mandated several years ago, requiring programs to re-use existing source code to the maximum extent possible. By forcing this mandate on Air Force acquisition programs and using "code re-use" as a metric during source selection, government and contractor personnel alike were encouraged to re-purpose code that may or may not be suitable for the new application it was being acquired for. This re-use policy meant that very few systems were built from the ground up, and therefore, inherited the technical debt that was present in the re-used code.

On the opposite side of the spectrum, Mr. Falcone mentioned a colleague that took a more aggressive approach to fighting technical debt, adopting a policy that any software system over 10 years old must be discarded and a new system must be developed to take its place. While this policy addressed the accumulation of technical debt directly, it is likely that many good systems (or good portions of bad systems) were discarded in the process.

Instead of instituting blanket policies on program age, percentage of re-use, or any other congressionally mandated metric, Mr. Falcone suggested that we analyze the teaming arrangements, both contractual and governmental, for how the Air Force develops software systems. During the recent transformation from Battle Management to PEO Digital, Mr. Falcone has fostered an environment that put government management, developers, and end users on the same teams to incentivize continuous development, rapid feedback, and decreased overhead. Kessel Run is the most recent and high-profile example of this DevOps style. In this environment, technical debt is reduced as users are incorporated into the process to provide immediate feedback and reduce false starts, while also putting less emphasis on meeting obsolete contractual requirements. This teaming arrangement also allows for a greater risk tolerance for program managers as they understand they can recover quickly from false starts. Teams are encouraged to develop multiple prototypes knowing full well that one or more may fail.

One of the main components that enables these teaming arrangements is a change in contractual arrangement between the government and defense contractors. By restructuring the contractual relationship between the parties, more collaboration and transparency around the product has emerged. Instead of legacy cost-based or fixed-price contracts that contain performance incentives based on high level, and often ineffective metrics, PEO Digital utilizes time and material (T&M) contracts that pay developers for their direct labor costs instead of their functional output. While this may sound like a poor incentive structure, it works based on the fact that there are embedded government controls in each team to maximize the developer's productivity and provide guidance to their daily, weekly and monthly tasks. Additionally, the shift from event-based milestones to schedule-based milestones have kept the DevOps teams on track to maximize their capability delivery in each sprint.

It is evident through discussion with Mr. Falcone that traditional contract arrangements foster the accumulation of technical debt due to their lengthy requirement development cycles, long decomposition timelines, burdensome development milestones, and testing phases. By the time a product is delivered it is often obsolete, triggering an immediate upgrade at the cost of millions of dollars to the American taxpayers. In a teaming framework as described above, software timelines are reduced dramatically, in some cases moving from a requirement, to development, to test, to fielding within weeks. While moving fast may cause more deficiencies to be discovered in the field, the timeline to get those problems fixed is reduced significantly.

It should be noted that this project delivery methodology has its critics. In one exchange with a senior leader in the test community, the individual was lamenting that a product shouldn't be fielded because his teams' report identified too many deficiencies. In response, Mr. Falcone's colleague commented that the software had already been through six iterations since the report was written, and that he could "write software faster than [the T&E team] could write [their] test and evaluation report."

While new contractual arrangements, DevOps teaming, and reduced fielding timelines cannot solve the problem of technical debt on their own, Mr. Falcone believes these process changes are a step in the right direction. In fact, he does not see any barriers to implementation from senior leaders in the Air Force. While every change will have its doubters, Mr. Falcone will push on with this vision and continue to advocate for software development that operates on a schedule-driven cadence, have automated testing capabilities, and have user centered design in conjunction with feedback from the operational community. This cultural and contractual shift, in conjunction with the architectural health analysis tools mentioned above, will increase transparency in the product and reduce technical debt in the majority of the products the Air Force acquires moving forward.

6.4 Barriers to Implementation

In contrast to the opportunities listed above, there are also barriers to the integration of architectural health tools into existing processes and organizations.

6.4.1 Discovery of Technical Debt Leading to Program Termination

One concern about the incorporation of architectural health tools into the software development lifecycle is that program managers may not want their code base to be scanned for fear it may uncover latent defects or infrastructure issues that have gone undetected over the course of the program, leading to termination. There is validity to this concern, as several of the case studies resulted in program cancellation or restructuring. With that said, using architectural health tools to provide accurate assessments of the technical debt that has accumulated over time will help senior leaders make more informed decisions using objective data, ultimately resulting in better products in the field and more efficient use of taxpayer dollars.

6.4.2 DoD and Defense Industrial Base Inertia Opposing Rapid Acquisition Principles

While the Congress is trying to encourage more rapid innovation and experimentation with new approaches to acquisition in the form of Other Transaction Agreements (OTAs), the inertia of the DoD and defense industrial base poses barriers to high velocity exploitation of these tools. Some question the allowable types of appropriated funds that can be used for rapid prototyping, effectively blocking an entire sector of development activities from using lean contracting methods. The root cause for this resistance is speculative, however several anecdotes suggest that the entrenched base wants to ensure their business remains with the large defense contractors and not the Silicon-Valley startups that have been disrupting the commercial sector.

While this discussion is not directly related to the accumulation of technical debt, it does relate closely to the contractual changes that need to take place in order to foster

an environment for innovation. This innovative culture is what will drive new processes, products, and business rules that drive technical debt out of both developmental and fielded systems. As such, cultural resistance is indirectly related to technical debt.

6.4.3 Qualified Personnel Shortages

Personnel shortages are being addressed at the DoD level, as leaders are recognizing that, "the human element puts a kink in long-term success of agile software development." The DoD is considering bringing back software development as a career field, however this only addresses part of the problem. [61]

6.4.4 Increased Cycle Time and Up-Front Resources

From the perspective of programmatic investment, it does cost money, take time, and add a step in the process to run an architectural health scan, however, research is starting to support the assertion that the return-on-investment of these scans greatly exceed the up-front costs of incorporating them in any development effort.

THIS PAGE INTENTIONALLY LEFT BLANK

7 Recommendations and Policy Guidance for DoD Software Acquisition

To address technical debt within the Air Force, the majority of the literature has been focused on changing software development processes. While a focus on process health addresses one of the root causes for the poor performance, it is equally important to address product health. This can be achieved by integrating architectural health analysis tools into product development cycles. By using tools like those outlined in this research, program managers, developers and senior leaders will begin to trust their product as they make fielding decisions to put the software in the hands of the operational community. Architectural assessments provide significant insight into the health of a product, providing high return-on-investment metrics on stakeholder's up-front investments. The intent is to catch technical debt prior to or during its creation, rather than after it has been incurred.

In the recommendations that follow, solutions will be offered to identify and manage technical debt throughout a project's lifecycle. The first set of recommendations focus on establishing the right business climate for architectural health analysis tools to be integrated within the software acquisition process. These recommendations include contractual, process, and teaming factors. The other set of recommendations address specific use cases where software acquisition programs should leverage the architectural health analysis tools presented in this research.

7.1 Improving Business Practices to Reduce Technical Debt

Business conditions and processes can be influential in the way software is developed, purchased and maintained. Throughout the course of this research, architectural health analysis tools are highlighted as a way to prevent, quantify, and address technical debt within the lifecycle of a software development effort. In some cases, these types of tools can be quickly integrated into a development effort. In other cases, contractual limitations, workforce restrictions, and process factors impede the inclusion of these types of tools. In this section, recommendations are given on how to establish the appropriate business climate to facilitate the five use cases that follow.

Recommendations to Reduce Technical Debt Through Improved Business Practices

- Train a cadre of personnel to specialize in data rights and licensing to ensure government has access to source code when appropriate
- Utilize OTA, T&M, and other innovative contract vehicles to properly incentivize developers
- Move towards agile product delivery methods in conjunction with proper education, training, and cultural changes
- Balance architectural health hygiene with new capability delivery
- Educate customers on basics of software development so they understand what they are buying
- Demand that the acquisition organizations dramatically improve the Software/IT expertise of its workforce

7.1.1 Utilizing Appropriate Contract Vehicles

> *"The DoD develops software and associated contracting based on upfront detailed systems requirements and specification for the entire completed system, an approach that is inadequate to meet today's challenges. The Department must change the structure of its contracts to incentivize best practices in its contractor base in order to take advantage of these modern software development practices." - Defense Science Board Report on Design and Acquisition of Software for Defense Systems, February 2018* [1]

From a contractual standpoint, the DoD must change the way it incentivizes contractors, how it oversees and evaluates software deliverables, how it staffs software development teams, and how it handles the procurement of source code and data rights.

From a data rights perspective, a balance must be struck between mandating full-government ownership and allowing contractors retain comprehensive data rights. While full-government ownership may be conducive from a cost perspective, this model would dampen industry's desire to innovate, thereby eroding the industrial base. As such, *the Air Force should train a cadre of personnel to specialize in data requirements and licensing*. These personnel would facilitate dialogue with industry to develop overarching IP valuation methodology, as well as data requirements and licensing processes for use in creating requests for proposals (RFPs) and crafting specially negotiated licenses. [16]

From a contract vehicle perspective, Kessel Run has set an example for how to properly use time and materiel contracts to develop, manage and deliver capability to their users. By removing contractual barriers, developers, managers, and users are integrated into highly cohesive yet loosely coupled teams (i.e. flat organizations with strong team presence). In this construct, source code is accessible by the entire team which increases transparency and allows developers to quickly identify and remediate technical debt before long-term cost is incurred. For software applications, *this model should be copied to the maximum extent possible.*

7.1.2 Improving Software Development Processes

From a process standpoint, *the DoD must continue to move towards more agile product delivery methods in conjunction with proper education, training, and cultural changes.* "The main benefit of iterative development — the ability to catch errors quickly and continuously, integrate new code with ease, and obtain user feedback throughout the development of the application — will help the DoD to operate in today's dynamic security environment, where threats are changing faster than Waterfall development can handle." [1] By pursuing iterative development practices, there is potential for technical debt to be reduced, as non-value added code is identified and removed at each iteration.

Some critics note that while agile processes have the potential to reduce technical debt, they also have potential to incur technical debt faster, as new features are released without regard for clean-up cycles. [63] This pitfall can be avoided through proper education and training along with the establishment of design rules to ignore technical debt on throwaway systems while addressing technical debt on systems that provide customer value. As such, Dr. Roper, Steve Falcone, the Defense Innovation Board, and the Defense Digital Service all support a shift towards agile processes, as does the 115th Congress. The 2018 National Defense Authorization Act mandates between 4 and 8 programs be selected as pilots to test agile processes. [64] The Air Force needs to capitalize on this movement by *utilizing agile practices whenever appropriate,* in conjunction with proper tools, training, management support, and cultural changes.

Regardless of whether a program successfully transitions to an agile development cycle, *acquisition professionals need to demand that for every capability they seek that enhances an existing weapon system some percentage of that funding must go towards the hygiene of the existing weapon system*.

7.1.3 Education and Training Reform

It should be noted that changing contract delivery mechanisms and using agile processes will not decrease technical debt without the appropriate cultural, education, and training reforms. According to Besselman, "…agile will not rescue DoD acquisition, because like so many previous innovations, it is being done in isolation, without restructure and reshaping of acquisition organizations, new processes, and genuine Software/IT education." [18] As such, there are two aspects that need to be addressed from an education and training perspective, warfighter training and acquisition officer training.

From an operational warfighting perspective, leaders need to recognize that their future capabilities rest on increasing volumes of software/IT, so *they need to learn the basics so they can be informed, sophisticated customers*. [18] From an acquisition perspective, *the DoD needs to demand that the acquisition organizations dramatically improve the software/IT expertise of its workforce*. We don't let nonpilots fly fighter jets or command fighter wings, but the DoD remains content to let the incurious attempt to acquire and sustain technically sophisticated weapon systems. No commercial company operates this way. [18]

7.2 Utilizing Architectural Health Analysis Tools to Reduce Technical Debt

Long-term agility is possible only if you're employing an agile product architecture. If humans can't easily understand or modify their code, teams might be using the best agile practices, but their ability to respond to market demands will be far from agile. [45] In the section above, business process recommendations were given on how to clear the way for the technical recommendations that follow. True reform cannot be achieved without both business and technical reforms.

From a technical perspective, this thesis has identified five potential use cases that need to be considered by Air Force leadership for inclusion in the acquisition process. Each use case is presented with details on when the Air Force could use it, how it could be employed, and what value the Air Force could garner from its inclusion. Since nearly 70 percent of all program costs are life-cycle sustainment and maintenance costs, [3] any investment that could reduce that cost would be well worth their up-front costs.

As a pre-requisite to the recommendations, an effort must be undertaken to track the architectural health metrics for each development program over time with the results being stored in a centralized database. Without appropriate history, it is difficult to identify trends across the Air Force's software portfolio. Instead, leaders are forced to rely on snapshots in time, often becoming reactive to unforeseen issues. According to the Defense Innovation Board, "the DoD keeps very little data about its own software projects. And since that's exactly the kind of information one would need to pinpoint where the problems lie, the board has made collecting it its first order of business." [4] It is evident that leadership has identified a deficiency in the way the Air Force operates its software acquisition system. With that said, the time is right to establish data repositories to track both government and contractor-led software development efforts over time. This pre-requisite sets the stage for all of the recommendations that follow.

Recommendations to Integrate Architectural Health Tools into Software Acquisition Process

- Include architectural health data in the evaluation criteria for source selection
- No software baseline should be accepted from industry without a sell-off of the capability's features based on quantitative architectural, code quality, component composition, and cyber security evidence
- Mandate architectural health assessments as a condition for contractual ownership changes
- No software baseline should be passed from development agency to sustainment agency within the government without a sell-off of the capability's features based on quantitative architectural, code quality, component composition, and cyber security evidence
- Architectural health analysis tools should be incorporated into the build cycle of CI/CD pipelines

7.2.1 Source Selection

The most effective way to address technical debt is to prevent it up front. As such, *architectural health data should be included in the evaluation criteria for source selection*. This includes architectural health data from previous efforts that is maintained in the central repository referenced above, as well as plans for how technical debt will be managed in the development effort being contracted. A contractor that performs quantitative assessments on their code base has the ability to develop modular, streamlined infrastructure that reduces long-term development and sustainment costs significantly.

In new source selections, government boards could reference this data to understand which contractors have the ability to develop well-structured code minimal cost built into the systems from the accumulation of technical debt. The DoD needs to focus contract awards solely on companies possessing genuine core competencies. For too long too much of the DoD's acquisition dollars goes to unqualified companies, where we not only pay them for a capability, but the first real phase of the respective acquisition is to pay industry to first learn how to use the technologies we seek to exploit. [18] Since nearly 70 percent of all program costs are life-cycle sustainment and maintenance costs, [3] any investment that could reduce that cost would be well worth their up-front costs.

7.2.2 Traditional Waterfall Programs

Despite the momentum towards Agile and DevOps style project delivery styles within the Air Force, it is likely that a large number of traditional waterfall-style programs will remain unchanged near-term. As such, traditional program managers and software developers should, through their contract vehicles, be granted authority to use the requisite tools to assess architectural health and technical debt within their portfolios. *No software baseline should be accepted from industry without a sell-off of the capability's features based on quantitative architectural, code quality, component composition, and cyber security evidence*. By allowing and encouraging the use of architectural health analysis tools in the entry criteria for successful design review completing, the government will be able to gain more insight into the deliverable that the contractor is required to deliver. It should be noted that the Procuring Contracting Officer (PCO) should be consulted prior to mandating this change as it may have contractual implications.

7.2.3 Contractor Handoff

As the government starts focusing on data rights issues, specifically shifting towards acquiring and owning product rights up front, there are likely to be more ownership changes over the course of a software program's lifecycle. In an effort to ensure incoming contractors understand the health of the code base their inheriting, *architectural health assessments should be mandated as a condition in contractual ownership changes*. This includes handoffs between contractors, handoffs from contractor to the

government, and handoffs within contractor teams. To take this one step further, results of the architectural scans could be included in the Request for Proposal (RFP) for contract re-competes to reduce technical uncertainty associated with the incoming contractor. The more they know about the status of the code their inheriting, the lower the risk to the contractor, therefore the lower the overall cost of the effort.

7.2.4 Intra-Government Handoff

In addition to the Air Force organizations tasked with the development of software systems, there are also organizations dedicated to sustaining software systems. In some situations, the same program office will manage both aspects of a product's lifecycle, while in other cases the code may change ownership from a program office to a software sustainment center. Similar to the recommendation above, new government owners must be aware of the health of the code their inheriting. As such, *no software baseline should be passed from program office to government sustainment agency without a sell-off of the capability's features based on quantitative architectural, code quality, component composition, and cyber security evidence*.

These intra-governmental handoffs most often occur between a system program office (SPO) and an Air Logistics Center, however could also occur between a SPO and various R&D organizations as shown in Case D. Having data that baselines the architectural health of the code helps scope the resources the government needs to allocate to the re-factoring and/or long-term sustainment of the code. These resources include both manpower and budget, which have to be included in Program Objective Memorandum or unit manning document requests, which are updated annually. It would make sense to get resourcing requirements correct on the first try, as opposed to undergoing a multi-year effort to rectify a situation that could have been avoided with appropriate information on the health of the product.

7.2.5 Continuous Development Pipelines

Continuous integration and continuous deployment (CI/CD) pipelines such as Kessel Run are prime candidates for incorporation of architectural health tools. With their frequent build and test cycles, these tools could provide developers immediate feedback on the structure of their code base. As such, *architectural health analysis tools should be incorporated into the build cycle* to be employed prior to the release of every new release or patch to ensure the underlying architecture of the software is appropriate for both the current execution of the system and future growth objectives.

Additionally, re-factoring tools could be useful in scoping sprints alongside new capability development. In an effort to deliver capability to the warfighter quickly, architectural health tools will ensure that software is both effective and efficient, and that the code base can be maintained over time to continuously allow for new capability development.

7.3 Future Research

The concept of technical debt has been receiving more attention as the world moves towards more software intensive systems. Journal articles in IEEE have increased dramatically since 2012. The SEI has several threads on the subject along with an annual conference dedicated to furthering exploration in this area. While this thesis explores technical debt within specific areas of Air Force software acquisition and development pipeline, future research could improve on the breadth of the case studies, including examples from the commercial market to compare and contrast the levels of technical debt between government and industry.

Second, future research should be dedicated to fine-tuning the business outcomes for how technical debt impacts future cost and schedule metrics. While the tools used in this thesis provide predictive analytics, they were not calibrated for the use cases that were explored. Any improvement in this area would be extremely useful for program managers in requesting the resources needed for program execution.

Third, to support the identification of problematic programs within the Air Force, cost-threshold metrics that indicate an increase in lifecycle cost expenditures for the sake of short-term savings should be investigated. This thesis highlighted a potential correlation between a program's sustainment cost in relation to its development cost as a marker for technical debt. This relationship should be examined further to see if there are threshold values that could alert senior leaders to looming programmatic issues.

Finally, research on the uncertainty surrounding measurements of technical debt should be further explored.

THIS PAGE INTENTIONALLY LEFT BLANK

8 Works Cited

[1] "Design and Acquisition of Software for Defense Systems," February 2018. [Online]. Available: https://www.acq.osd.mil/dsb/reports/2010s/DSB_SWA_Report_FINALdelivered2-21-2018.pdf.

[2] S. Maucione, "Software Is Air Force Acquisition's Biggest Problen, Roper Says," 30 April 2018. [Online]. Available: https://federalnewsnetwork.com/air-force/2018/04/software-is-air-force-acquisitions-biggest-problem-roper-says/.

[3] A. Mehta, "Six Things on the Pentagon's 2019 Acquisition Reform Checklist," Defense News, 27 December 2018. [Online]. Available: nearly 70 percent of all program costs are life-cycle sustainment and maintenance costs. [Accessed 27 December 2018].

[4] J. Serbu, "Innovation Board has a Dozen Ideas to Help Fix DoD's Software Acquistion Woes," Federal News Network, 12 July 2018. [Online]. Available: https://federalnewsnetwork.com/acquisition/2018/07/innovation-board-has-a-dozen-ideas-to-help-fix-dods-software-acquisition-woes/. [Accessed 22 December 2018].

[5] R. Stross, "Billion-Dollar Flop: Air Force Stumbles on Software Plan," 8 December 2012. [Online]. Available: https://www.nytimes.com/2012/12/09/technology/air-force-stumbles-over-software-modernization-project.html.

[6] S. Verch, "How Technical debt makes government software crap, and what we can do about it," FCW, 29 May 2018. [Online]. Available: https://fcw.com/articles/2018/05/29/technical-debt-usds-verch.aspx. [Accessed 31 December 2018].

[7] S. o. t. A. F. f. Acquisition, "Weapon Systems Software Management Guidebook," Department of Defense, Washington DC, 2008.

[8] S. Maucione, "Air Force Digital Service will Fundamentally Change Acquisition," Federal News Network, 17 March 2017. [Online]. Available: https://federalnewsnetwork.com/air-force/2017/03/air-force-digital-service-will-fundamentally-change-acquisition/. [Accessed 28 December 2018].

[9] W. Cunningham, "The WyCash Portfolio Management System," *Addemdum to Proc. Object-Oriented Programming Systems, Languages, and Applications (OOPSLA),* pp. 29-30, 1992.

[10] J. Kerievsky, Refactoring to Patterns, Addison-Wesley, 2004.

[11] A. MacCormack and D. Sturtevant, "Technical Debt and System Architecture: The Impact of Coupling on Defect-Related Activity," *Journal of Systems and Software,* 2016.

[12] I. Ozkaya, "Data-Driven Management of Technical Debt," 29 October 2018. [Online]. Available: https://insights.sei.cmu.edu/sei_blog/2018/10/data-driven-management-of-technical-debt.html.

[13] R. Nord, "The Future of Managing Techincal Debt," 29 August 2016. [Online]. Available: https://insights.sei.cmu.edu/sei_blog/2016/08/the-future-of-managing-technical-debt.html.

[14] "Technical Debt: All Things in Moderation," [Online]. Available: https://deviq.com/technical-debt/.

[15] M. Fowler, "TechnicalDebtQuadrant," MartinFowler.com, 14 October 2009. [Online]. Available: http://martinfowler.com/bliki/TechnicalDebtQuadrant.html. [Accessed 28 December 2018].

[16] D. Deptula, "The Growing Importance of Data Rights in Defense Acquisition," Forbes, 16 October 2018. [Online]. Available: https://www.forbes.com/sites/davedeptula/2018/10/16/the-growing-importance-of-data-rights-in-defense-acquisition/#42f6d2b02a04. [Accessed 29 December 2018].

[17] C. Berardi, *Intellectual Property and Architecture: How Architecture Influences Intellectual Property Lock-In,* Massachusetts Institute of Technology, 2017.

[18] J. Besselman, Interviewee, *Technical SME; PEO Digital.* [Interview]. 9 January 2019.

[19] F. Schull and I. Ozkaya, Recommended Practice for Application of Quantitative Software Architecture Analysis in Sustainment, Carnegie Mellon University: Software Engineering Institute, 2018.

[20] House of Representatives Information Technology Subcommittee , *Federal Agencies' Reliance on Outdated and Unsupported Information Technology: A Ticking Time-Bomb,* Washington DC: Committee on Oversight and Government Reform, 2016.

[21] Red Hat, "Paying Off Technical Debt for Successful IT Modernization," Federal News Network, 18 December 2018. [Online]. Available: https://federalnewsnetwork.com/open-first/2018/12/paying-off-technical-debt-for-successful-it-modernization/. [Accessed 28 December 2018].

[22] D. Sturtevant, "Computer-Implemented Methods and Systems for Measuring, Estimating, and Managing Economic Outcomes and Technical Debt in Software Systems and Projects". United States Patent US 2017/0235569A1, 17 August 2017.

[23] Silverthread Inc., *Gaining Control of Your Software,* Cambridge: Silverthread Inc., 2018.

[24] S. Erwin, "Air Force changing how it buys weapons and satellites, but software still a headache," Space News, 7 March 2018. [Online]. Available: https://spacenews.com/air-force-changing-how-it-buys-weapons-and-satellites-but-software-still-a-headache/. [Accessed 1 January 2019].

[25] "Selected Acquisition Report (SAR): F-35," Office of the Secretary of Defense, Washington DC, 2018.

[26] B. McGarry, "F-35 Deficiencies Decreasing, but Hundreds Remain: Program Manager," 2018. [Online]. Available: https://www.military.com/daily-news/2016/02/17/f35-deficiencies-decreasing-hundreds-remain-program-manager.html.

[27] V. Insinna, "Air Force Cancels Air Operations Center 10.2 Contract, Starts New Pathfinder Effort," 13 July 2017. [Online]. Available: https://www.defensenews.com/air/2017/07/13/air-force-cancels-air-operations-center-10-2-contract-starts-new-pathfinder-effort/.

[28] P. Kruchten, R. Nord and I. Ozkaya, "Technical Debt: From Metaphor to Theory and Practice," *IEEE Software,* vol. 29, no. 6, pp. 18-21, 2012.

[29] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999.

[30] F. Buschmann, "Gardening Your Architecture, Part 1: Refactoring," *IEEE Software,* vol. 28, no. 4, pp. 92-94, 2011.

[31] F. Buschmann, "Unusable Software Is Useless, Part 2," *IEEE Software,* vol. 28, no. 2, pp. 92-94, 2011.

[32] F. Buschmann and K. Henny, "Five Considerations for Software Architecture, Part 2," *IEEE Software,* vol. 27, no. 4, pp. 12-14, 2010.

[33] B. F., "Learning From Failure, Part 3: On Hammers and Nails, and Falling in Love with Technology and Design," *IEEE Software,* vol. 27, no. 2, pp. 49-51, 2010.

[34] F. Bushmann, "Gardening Your Architecture, Part 2: Reengineering and Rewriting," *IEEE Software,* pp. 21-23, 2011.

[35] S. Demeyer, S. Ducasse and O. Nierstrasz, Object-Oriented Reengineering Patterns, Morgan-Kaufmann, 2002.

[36] C. Baldwin and K. Clark, Design Rules: Volume 1. The Power of Modularity, Cambridge, MA: The MIT Press, 2000.

[37] N. Fosgren, "2017 State of DevOps Report," Puppet, 2017.

[38] "Hierarchical Architecture," Tutorials Point, [Online]. Available: https://www.tutorialspoint.com/software_architecture_design/hierarchical_architecture.htm. [Accessed 21 December 2018].

[39] D. Batory and S. O'Malley, "The Design and Implementation of Hierarchical Software Systems with Reusable Components," *ACM Transactions on Software Engineering and Methodology,* vol. 1, no. 4, pp. 355-398, 1992.

[40] "Code Analysis Tools," Cast Software, [Online]. Available: https://www.castsoftware.com/products/code-analysis-tools. [Accessed 12 January 2019].

[41] "Detect Tricky Issues," SonarQube, [Online]. Available: https://www.sonarqube.org/features/issues-tracking/. [Accessed 12 January 2019].

[42] "Lattix Architect," 2018. [Online]. Available: https://lattix.com/lattix-architect. [Accessed 22 December 2018].

[43] "Silverthread," 2018. [Online]. Available: http://www.silverthreadinc.com/what-we-do/our-products/. [Accessed 22 December 2018].

[44] S. Eppinger and T. Browning, Design Structure Matrix Methods and Applications, Cambridge, MA: The MIT Press, 2012.

[45] D. Sturtevant, "Modular Architectures Make You Agile in the Long Run," *IEEE Software,* pp. 104-108, 2018.

[46] "DSM Tutorials: Overview," DSMWeb.org, [Online]. Available: http://www.dsmweb.org/en/understand-dsm/tutorials-overview.html. [Accessed 28 December 2018].

[47] J. Kearney, R. Sedlmeyer, W. Thompson, M. Gray and M. Adler, "Software Complexity Measurement," *Communications of the ACM,* vol. 29, no. 11, pp. 1044-1050, 1986.

[48] R. Selby and V. Basili, "Error Localization During Software Maintenance: Generating Hierarchical System Descriptions from the Source Code Alone," in *Software Maintenance*, 1988.

[49] H. Dhama, "Quantitative Models of Cohesion and Coupling in Software," *J. Systems Software,* vol. 29, pp. 65-74, 1995.

[50] T. McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering,* Vols. SE-2, no. 4, pp. 309-320, 1976.

[51] C. Baldwin, A. MacCormack and J. Rusnak, "Hidden Structure: Using Network Methods to Map System Architecture," *Research Policy,* vol. 43, no. 8, pp. 1381-1397, 2014.

[52] A. MacCormack, C. Baldwin and J. Rusnak, "Exploring the Duality between Product and Organizational Architecture: A Test of the "Mirroring" Hypothesis," *Research Policy,* vol. 41, no. 8, pp. 1309-1324, 2012.

[53] "Software Design Complexity," Tutorials Point, [Online]. Available: https://www.tutorialspoint.com/software_engineering/software_design_complexity.htm. [Accessed 28 December 2018].

[54] P. Jorgensen, in *Software Testing: A Craftsman's Approach, Second Edition*, CRC Press, 2002, pp. 150-153.

[55] *ISO/IEC/IEEE 24765:2010 Systems and software engineering*.

[56] E. Yourdon and L. Constantine, Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design, Yourdon Press, 1979.

[57] I. Chowdhury and M. Zulkernine, "Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities," *Journal of Systems Architecture,* vol. 57, no. 3, pp. 294-313, 2011.

[58] D. Sturtevant, *System Design and the Cost of Architectural Complexity,* Cambridge: MIT, 2013.

[59] C. Izurieta, I. Griffith, D. Reimanis and R. Luhr, "On the Uncertainty of Technical Debt Measurement," *IEEE,* 2013.

[60] J. Reilly, Interviewee, *Program Manager; AFRL.* [Interview]. Multiple Multiple 2018-2019.

[61] L. Williams, "Air Force wants to make 'Kessel Run' standard in tech acquisition," FCW, 3 October 2018. [Online]. Available: https://fcw.com/articles/2018/10/03/usaf-kessel-run-standard.aspx. [Accessed 15 December 2018].

[62] S. Falcone, Interviewee, *Chief Engineer, PEO Digital.* [Interview]. 5 November 2018.

[63] F. Arcelli, W. Trumler, C. Izurieta and R. Nord, "Ninth International Workshop on Managing Technical Debt," in *Report on the MTD 2017 Workshop*, Cologne, 2017.

[64] *H.R. 2810: National Defense Authorization Act for Fiscal Year 2018,* Washington DC: 115th Congress, 2017.

[65] O. de Weck, "Design Structure Matrix," 2012. [Online]. Available: https://ocw.mit.edu/courses/engineering-systems-division/esd-36-system-project-management-fall-2012/lecture-notes/MITESD_36F12_Lec04.pdf.

[66] P. Clements, R. Kazman and M. Klein, Evaluating Software Architectures, Addison-Wesley, 2002.

[67] J. Reilly, Interviewee, *Program Manager.* [Interview]. 16 August 2018.

[68] "Cost-Reimbursement Contracts," AcqNotes, [Online]. Available: http://acqnotes.com/acqnote/careerfields/cost-reimbursement-contracts. [Accessed 31 December 2018].

[69] SciTools, SciTools, [Online]. Available: https://scitools.com/features/. [Accessed 12 January 2019].

[70] M. Fowler, "Code Smell," MartinFowler.com, 9 February 2006. [Online]. Available: https://martinfowler.com/bliki/CodeSmell.html. [Accessed 12 January 2019].

Appendix A: List of Silverthread Scans on Air Force Systems

This table is included for two reasons. First, it shows the *magnitude* of technical debt that has been found in some of the Air Force programs that Silverthread has scanned to date. The worst offender has been Case K with a core of over 26,000 files, a propagation cost with over 96% of the files having direct or indirect dependencies to each other, and a cyclomatic complexity metric of 535 files having a McCabe score of over 50.

Second, this table shows the *pervasiveness* of technical debt across the entire software enterprise. Of the 49 different cases shown in this table, 24 of them have at least one metric in the "red" category (49%).

		Technical Health			Details	
		Cyclomatic Complexity: # of files with McCabe >50	**Architectural Cyclicality:** Size of largest file-file cycle	**Propagation Cost:** % of code that is linked together	Lines of Code	Number of Files
Case A						
	Case A (Java)	13	670	0.18	1097347	3689
	Case A (C#)	42	528	0.04	2888104	15639
	Case A (C++)	15	11	0.00	975776	1992
Case B						
	Case B (Java)	64	1409	0.03	3857487	18840
	Case B (C#)	4	4	0.03	61259	465
	Case B (Web)	28	527	0.04	1767930	7573
Case C						
	Case C: 16 Nov 17 (Ada)	24	261	0.08	1150299	3328
Case D						
	Case D: 20 Mar 08 (Java)	7	415	0.26	307282	1618
	Case D: 20 Mar 09 (Java)	6	526	0.30	332138	1790
	Case D: 19 Mar 10 (Java)	8	642	0.34	373762	1937
	Case D: 18 Mar 11 (Java)	27	689	0.22	1202122	4914
	Case D: 20 Mar 12 (Java)	26	662	0.21	1205413	4865
	Case D: 20 Mar 13 (Java)	24	768	0.28	1095985	4321
	Case D: 17 Mar 14 (Java)	24	765	0.27	1112800	4369
	Case D: 20 Mar 15 (Java)	3	717	0.33	395105	2129
	Case D: 16 Mar 16 (Java)	1	481	0.30	374173	2067
	Case D: 27 Jun 17 (Java)	1	566	0.30	402021	2328
	Case D: 7 Aug 18 (Java)	1	364	0.23	314967	1959
	Case D: 28 Sep 18 (Java)	0	368	0.22	323422	2048
	Case D: 29 Sep 18 (Java)	0	184	0.15	323480	2049
	Case D: 2 Oct 18 (Java)	0	133	0.14	323648	2048
	Case D: 3 Oct 18 (Java)	0	110	0.11	323707	2048
Case E						
	Case E: Version 1.3 (Web)	4	51	0.05	353100	1306
	Case E: Version 2.0 (Web)	4	34	0.03	390,148	1,586
	Case E: Version 3.0 (Web)	4	34	0.03	375,854	1,542
	Case E: Version 4.5 (Web)	2	14	0.01	763047	4005
Add'l Cases						
	Case F (Java)	101	6134	0.29	5268326	22199
	Case G (Java)	4	11	0.02	270889	2034
	Case H (C#)	0	5	0.09	102269	493

Case H (Web)	0	41	0.26	121218	161
Case I (Web)	12	5	0.09	65936	142
Case J (Java)	1	11	0.12	85567	331
Case K (Java)	535	26526	0.92	5030193	28270
Case L (Java)	2	14	0.01	763047	4005
Case M (Web)	5	258	0.15	472313	1730
Case N (Java)	24	211	0.02	1204270	7465
Case O (Java)	1	588	0.42	260723	1461
Case P (C++)	0	16	0.06	64787	155
Case Q (C#)	0	9	0.03	181983	1091
Case R (C#)	5	12	0.02	216478	459
Case S (C#)	1	13	0.01	274015	1456
Case T (C#)	4	19	0.07	185065	225
Case T (C++)	5	127	0.13	405577	953
Case U (C++)	29	243	0.54	383826	447
Case V (C++)	0	0	0.27	2791	9
Case W (C#)	0	2	0.04	67646	334
Case X (C#)	1	73	0.17	93624	677
Case X (Web)	1	11	0.25	49643	52
Case Y (C++)	30	230	0.56	366443	410
Case Z (Ada)	26	10898	0.87	3189043	12502
Case Z (C++)	47	174	0.04	2238362	1719
Case Z (Java)	7	38	0.01	1043497	5949
Case AA (C#)	12	25	0.02	1029648	1842
Case AA (C++)	0	10	0.02	217781	432
Case AA (Web)	4	10	0.36	16719	32
Case AB (C#)	7	10	0.04	717555	2475
Case AB (C++)	1	2	0.11	6621	15
Case AC (C#)	0	81	0.13	167207	428
Case AD (C#)	0	4	0.31	4792	24
Case AD (C++)	25	260	0.17	469434	1880
Case AD (Web)	0	42	0.40	131844	109
Case AD (Java)	0	6	0.12	18019	84
Case AE (C#)	6	29	0.01	1565537	3429
Case AE (C++)	146	662	0.09	4300133	9139
Case AF (C#)	6	38	0.01	1943526	5220
Case AF (C++)	146	671	0.09	4328610	9201
Case AG (C#)	7	41	0.01	2079661	5439
Case AG (C++)	145	695	0.09	4332000	9184
Case AH (C#)	11	203	0.01	3175193	7115
Case AH (C++)	295	2868	0.15	7935260	18145
Case AI (Java)	21	166	0.03	1025136	6338
Case AJ (Java)	20	166	0.03	1034954	6376
Case AK (Web)	2	143	0.33	84137	479
Case AK (Java)	0	8	0.02	148251	1135
Case AL (C#)	0	12	0.04	269121	401
Case AL (Web)	0	3	0.04	53222	84
Case AM (C#)	0	5	0.03	41388	376
Case AN (C#)	5	690	0.19	610348	3725
Case AN (C++)	0	34	0.05	314250	430
Case AO (C++)	42	427	0.04	1457073	6090
Case AO (Python)	0	0	0.18	4266	31
Case AO (Java)	1	2	0.00	64001	651
Case AP (C++)	19	145	0.17	513915	719
Case AQ (C++)	23	161	0.27	509484	570
Case AR (C++)	24	162	0.27	516391	582
Case AS (C#)	13	32	0.03	470968	2798
Case AT (C#)	8	86	0.08	660345	1382
Case AT (C++)	0	0	0.19	1862	15
Case AU (Web)	5	7	0.02	191827	587
Case AV (Java)	0	2	0.04	17193	73
Case AW (C++)	2	12	0.05	96969	380