Many software developers have to deal with legacy code at some point in their careers. Seemingly simple changes can become frustrating endeavors. Code may be hard to read and unnecessarily complex. Test scripts and requirements are often out of sync with the existing system. The build is nearly always cryptic, minimally sufficient, and difficult to successfully configure and execute. It is almost impossible to find the proper place to make a requested change without breaking unexpected portions of the application. And the people who originally worked on the application? Long gone.

How did the software get like this? It is almost certain the people who developed this application did not intend to create such a mess. The following article will explore the multitude of factors involved in the development of software with debt.

**What Contributes to Software Debt?**

Software debt accumulates when a team’s focus remains on immediate completion while neglecting changeability of the system over time. The accumulation of debt does not impact software delivery immediately, and may even create a sense of increased feature delivery. Business responds well to the pace of delivered functionality and the illusion of earlier returns on investment. Team members may complain about the quality of delivered functionality while debt is accumulating, but do not force the issue due to enthusiastic acceptance and false expectations they have set with the business. Debt usually shows itself when the team works on stabilizing the software functionality later in the release cycle. Integration, testing, and bug fixing is unpredictable and does not get resolved adequately before the release.

The following sources constitute what I call software debt:

- **Technical Debt**: Those things that you choose not to do now and will impede future development if left undone
- **Quality Debt**: Diminishing ability to verify functional and technical quality of entire system
- **Configuration Management Debt**: Integration and release management become more risky, complex, and error-prone
• Design Debt: Cost of adding average-sized features becomes greater than writing from scratch
• Platform Experience Debt: Availability and cost of people to work on system features are becoming limited

Software Debt Creeps In

Software projects are complex. They involve balancing requirements and technology in the pursuit of delivering valuable software to the intended users. At the beginning of new software projects, teams focus on the design. This focus on design enables the team to deliver something they are proud of. There is little to no decay in the first feature delivered across a set of architectural components.

![Figure 1.1: Focus on good design in first feature of release with little schedule pressure yet](image1)

As the release progresses, schedule pressure starts to settle in and there is increasing focus on just getting features done. Software debt starts to creep in as team members look for ways to shave off small amounts of time implementing a feature leaving behind small defects in the software.

![Figure 1.2: A relatively new system with little debt accrued](image2)

Figure 1.2 represents a system that has minimal amount of software debt accrued. A few low-priority defects have been logged against the system and the build process may involve some manual configuration. The debt is not enough to significantly prolong implementation of upcoming features.

Business owners expect a sustained pace of feature development and the team attempts to combine both features and bugs into daily activities, which accelerates the accrual of debt. Software debt is accruing faster than it is being removed. This may become apparent with an increase in the number of issues logged against the system.
As a system ages, small increments of software debt are allowed to stay so the team can sustain their velocity of feature delivery. The team may be complaining about insufficient time to fix defects. Figure 1.3 shows a system that has incurred software debt across all functional areas and components.

At this point, delivery slows down noticeably. The team asks for more resources to maintain their delivery momentum, which increases the costs of delivery without increasing the value delivered. Return on investment (ROI) is affected negatively, and management attempts to minimize this by not adding as many resources as the team asks for, if any.

Even if business owners covered the costs of extra resources, it would only reduce the rate of debt accrual and not the overall software debt in the system. Feature development by the team produced artifacts, code, and tests that complicate software debt removal. The cost of fixing the software debt increases exponentially as the system ages and the code base grows.
Managing Software Debt: Continued Delivery of High Values as Systems Age | Chris Sterling | July 2008

![Figure 1.4: The aging system with accrued significant debt in all functional areas and components](image)

Software debt in the system continues to accrue over time, as shown in Figure 1.4. At this point, new feature implementation is affected significantly. Business owners may start to reduce feature development and put the system into “maintenance” mode. These systems usually stay in use until business users complain that the system no longer meets their needs.

### Managing Software Debt

There are no magic potions for managing software debt. Software can accrue debt through unforeseen circumstances and shortsighted planning. There are some basic principles that help minimize software debt over the lifespan of the product:

- Maintain one list of work
- Emphasize quality
- Evolve tools and infrastructure continually
- Always improve system design
- Share knowledge across the organization
- Hire the right people to work on your software!

### Maintain One List of Work

One certain way to increase software debt is to have multiple lists of work. Clear direction is difficult to maintain with separate lists of defects, desired features, and technical infrastructure enhancements. Which list should a team member choose from? If the bug tracker includes high-priority bugs, it seems like an obvious choice.
Influential stakeholders want new features, though, so they can show progress to their management and customers. Also, if organizations don’t enhance their technical infrastructure, future software delivery will be affected.

Deployed software considered valuable to its users is a business asset, and modifications to a business asset should be driven from business needs. Bugs, features, and infrastructure desires for software should be prioritized together on one list. Focus on one prioritized list of work will minimize confusion on direction of product and context switching of team members.

**Emphasize Quality Towards Executable Design**

Going beyond the design/code/test software development flow to building integrity in as we go is the first step to emphasizing quality. Executable Design is an approach involving existing well-known practices, effective principles, and a mindset shift that is heavily influenced by Agile values and principles. The principles of Executable Design are:

- The way we design can always be improved
  
  There is no general way to conduct software design and attempts to do so will reduce value delivered by teams and increase costs to manage.

- You’ll get it “right” the third time
  
  Customers change their mind, technology gets better understood, and teams learn more about the software’s domain as they construct the software. Find ways to incorporate this learning to properly incorporate effective change.

- We will not get it “right” the first time
  
  Unfortunately, as an industry we have not been successful in determining what customers want before they interact with the software. Short iterations allow teams to give customers what they think they want quickly and get valuable feedback on changes the customer identifies once they touch and feel the software.

- Design and construct for change rather than longevity
  
  The purpose of defining the software’s architecture and design is not to create software that lives a long time. It is to make the software’s structure changeable to meet the new customer and business demands. Increased test and deployment automation for repeatable and specific activities in software delivery provides the structure necessary to allow the software to evolve for these new demands.

- Lower the threshold of pain
  
  I participated at a workshop in Grand Rapids, Michigan on Technical Debt where Matt Heusser proposed that technical debt could be an outcome of individual developers not having to deal with consequences of their actions. The decisions that we make each day in developing software lead to technical debt due to this “moral hazard.”

_Moral Hazard is the prospect that a party insulated from risk may behave differently from the way it would behave if it were fully exposed to the risk._
This does not mean that team members act in a malicious or dishonest way. It means that if individuals are insulated from the long-term effects of a decision then we will not take as much care in the short-term. Matt’s example was that a person may take a shortcut in developing a feature since they are not going to be working on the code one year from now when it must be modified to support a new feature.

Immediately following Matt’s discussion on moral hazard, Chet Hendrickson pointed out that a good way to minimize the moral hazard problem is by “lowering the threshold of pain.” For example, Chet brought up how many people approach doing their taxes in the United States. They could incrementally update their taxes two hours each month. Instead many of us wait until two weeks before the deadline to get our tax forms completed. We push to complete our taxes by the deadline because the potential headache of tax evasion is strong enough that it pushes us over a threshold of pain. The approaching threshold causes us to act because the pain is something we do not wish to feel.

Lowering the threshold of pain involves decreasing the duration to get feedback at all levels of software development. Automate programmer tests and execute them in each developer environment. Provide access to automated integration test execution to entire team and stakeholders. Reduce logging of defects on functionality getting worked on in current iteration and increase communication between those that find defects and those that can fix them. The ultimate goal of lowering the threshold of pain is to achieve a “push-button release.” This will give flexibility to the business so they can decide to release whenever it makes the most sense and optimizes their return on investment.

Evolve Tools and Infrastructure Continually

Ignoring the potential for incremental improvements in existing software assets leads to the assets becoming liabilities. Maintenance efforts in most organizations lack budget and necessary attention. The International Organization for Standardization (ISO) standardizes on four basic categories of software maintenance in ISO/IEC 14764:

- Corrective maintenance: Reactive modification of a software product performed after delivery to correct discovered problems
- Adaptive maintenance: Modification of a software product performed after delivery to keep a software product usable in a changed or changing environment
- Perfective maintenance: Modification of a software product after delivery to improve performance or maintainability
- Preventive maintenance: Modification of a software product after delivery to detect and correct latent faults in the software product before they become effective faults

Most maintenance efforts seek to prolong the life of the system rather than increase its maintainability. Maintenance efforts tend to be reactive to end user requests while business evolves and the technology decays.

To prevent this, attention must be given to all four categories of software maintenance. Budgets for software projects frequently ignore costs for adaptive, perfective, and preventive maintenance. Understanding that corrective maintenance is only part of the full maintenance picture can help an organization manage their software assets over its lifespan.
**Improve System Design Always**

Manage visibility of system design issues with the entire team. Create a common etiquette regarding modification of system design attributes. Support the survival of good system design through mentoring, proactive system evolution thinking, and listening to team member ideas. In the end, a whole team being thoughtful of system design issues throughout development will be more effective than an individual driving it top down.

**Share Knowledge Across the Organization**

With some software systems, there is a single person in the organization who owned development for five years or more. Some of these developers may find opportunities to join other companies or are getting close to retirement. The amount of risk these organizations bear due to lack of sharing knowledge on these systems is substantial.

Although that situation may be an extreme case of knowledge silos, a more prevalent occurrence in IT organizations is specialization. Many specialized roles have emerged in the software industry for skills such as usability, data management, and configuration management. The people in these roles are referred to as “shared resources” because they use their specialized skills with multiple teams.

Agile software development teams inherit team members with specialized roles, which initially is a hindrance to the team’s self-organization around the work priorities. Teams who adhere to agile software development values and principles begin to share specialized knowledge across the team, which allows teams to be more flexible in developing software based on priorities set by business. Sharing knowledge also reduces the risk of critical work stoppage from unavailable team members who are temporarily on leave.

**Hire the Right People!**

It is important to have the team involved in the hiring process for potential team members. Teams will know the most relevant skills they need, so allowing them to review and edit the job description is essential. Conducting traditional interview sessions that smother candidates with difficult questions are insufficient for determining if the candidate will be a great fit. Augmenting the interview questions with a process for working with the candidate during a one to two hour session involving multiple team members in a real-world situation adds significant value to the interview process. Before hiring a candidate, teams members should be unanimous in their decision. This will increase the rate of success for incorporation of a new team member because the team is accepting of its addition.

Another significant hiring focus for organizations and teams must be placing more emphasis on soft skills than technical expertise. I am not advocating ignoring technical experience. It is critical in an agile software development organization or team to have people who can collaborate and communicate effectively. Soft skills are more difficult to learn than most technical skills. Look for people who have alignment with the hiring team’s values and culture.
In Summary

As systems age, they can become more difficult to work with. Software assets become liabilities when software debt creeps into systems through technical debt, quality debt, configuration management debt, design debt, and platform experience debt.

Applying the six principles in this article will lead to small changes that over time will add up to significant positive change for teams and organizations. The goal of managing software debt is to optimize the value of software assets for your business and increase the satisfaction of your customers with the software they use.

References


About the Author

Chris Sterling is an Agile Coach, Certified Scrum Trainer, and Technology Consultant for SolutionsIQ. He has been involved in many diverse projects and organizations and has extensive experience with bleeding edge and established technology solutions. He has been a coordinator of multiple Puget Sound area groups including International Association of Software Architects (IASA), Seattle Scrum Users Group, and most recently the Beyond Agile group. He has been a speaker at many conferences and group meetings including Agile 2007 & 2008, SD West, Scrum Gathering, and others. In his consulting and speaking engagements, Chris brings his real world experience and deep passion for software development enabling others to grasp the points and take away something of value. Chris has also contributed to and created multiple open source projects. He is currently teaching the “Advanced Topics in Agile Software Development” class at the University of Washington Agile Developer Certificate extension program and writing a book with publisher Addison-Wesley on software architecture.

About SolutionsIQ

SolutionsIQ offers a full spectrum of services to develop software and fulfill technical talent needs, while improving your Agile knowledge and capabilities. Clients include AT&T (Cingular), Amazon, Corbis, Expedia, Federal Home Loan Bank, InfoSpace, Key Bank, Microsoft, Nordstrom, Safeco, and US Bank. A Microsoft Gold Certified Partner, SolutionsIQ is also a member of Scrum Alliance, Agile Alliance, and Washington Technology Industry Association. Learn more at www.SolutionsIQ.com.