Net Objectives
ESSENTIAL WHITE PAPERS

Avoiding System Bankruptcy

How to Pay Off Your Technical Debt

by Amir Kolsky

Reprinted from
Agile Product & Project Management
Vol. 11, No9, pp. 1-13

Avoiding System Bankruptcy: How to pay off your technical debt

A Net Objectives Essential White Paper

Net Objectives Press, a division of Net Objectives

13619 Mukilteo Speedway, Suite 1205

Lynnwood, WA 98087

404-593-8375

Find us on the Web at: *www.netobjectives.com*

To report errors, please send a note to *info@netobjectives.com*

10 9 8 7 6 5 4 3 2 1

Printed in the United States of America

First Edition

This article was first printed in Agile Product & Project Management, Vol. 11, No. 9, pp. 1-13.

In life we are all familiar with debt, especially financial debt. The longer a monetary debt is left unpaid, the more interest accrues. Eventually bankruptcy may be declared. Similarly, in software development, every time something is executed incorrectly, it may be thought of as technical debt. If the technical debt is not paid up, the system's quality will rapidly deteriorate until it goes "bankrupt"; it then may be decommissioned as the cost of maintaining it will be too high. This Executive Report introduces the concept of technical debt, what practices and attitudes cause it, and what we can do to prevent it or pay it off.

Software development is a complex endeavor with two major challenges: making sure you do the right thing and making sure you do it right. Doing the right thing has to do with meeting the customer's needs and is usually handled successfully. Doing it right, however, concerns how the development team goes about its work. If this is not done properly — if attention is not paid to the right things and if the development practices are not good enough — the quality of the development environment and the code base will deteriorate rapidly.

Debt is a familiar concept. There are many types of debt, including a loan carrying interest that we need to pay off; a penalty on a fine we forgot to pay; a carton of milk we borrowed from a neighbor that we need to replace; or a request that we will not refuse when asked as the individual previously did us a favor. Debt, eventually, has to be paid — with or without interest. As much as we may like to forget the debt, the debtors tend to remind us and at some point come to collect.

If a financial debt is not repaid, it may accrue interest. The longer that debt goes unpaid, the more interest will be added — exponentially; eventually bankruptcy may be declared. Similarly, in software development, every time something is executed incorrectly, it may be thought of as technical debt. If the technical debt is not paid, the system's quality will rapidly deteriorate until it goes "bankrupt"; it then may be decommissioned as the cost of maintaining it will be too high.

The term "technical debt" is a metaphor commonly used by development teams to refer to something that they control and yet know is not as good as it should be. In this metaphor, the team owes it to the system to correct the situation.

Financial debt has an impact on our ability to acquire goods or services because some of our income goes toward covering the debt or dealing with its repercussions. Technical debt is similar in that it limits our ability to maintain the system through modifications or the addition of new features because we have to deal with the technical debt's repercussions. In this report, we will extend the financial metaphor further to introduce the concept of a "technical tax," which is similar to property tax.

A wanting design is a common source of technical debt, but it is not the only one. In this Executive Report, we explore the causes of technical debt and look at the various forms such debt can take, including design-related debt, testing-related debt, defect-related debt, and organizational and administrative-related debt. The report examines the source of the debt, the cost of the debt, and how to pay the debt off or avoid it altogether. Since accruing technical debt may be a conscious decision, the team needs to have a plan in place for paying off the debt before it commits to it.

Developers use many excuses to justify why they should not do their job properly and in a timely manner.

UNDERSTANDING TECHNICAL DEBT

Software development is a complex process. Doing it properly involves keeping in mind immediate needs as well as future maintenance (modifying existing behavior, adding extra behavior, fixing bugs, improving performance, etc.). Accounting for the time spent attending to the immediate needs is easy; if we do not spend this time, we cannot get the work done. Accounting for the time spent on the other activities — those that will

Amir Kolsky is a Senior Consultant, Coach, and Trainer for Net Objectives. He has been in the computer sciences field for almost 30 years. He worked for 10 years in IBM Research and spent nine more years doing chief architect and CTO work in assorted companies big and small. He has been involved with Agile since 2000. He founded MobileSpear and subsequently XP and Software, which provides Agile coaching, software education, and Agile projects in Israel and Europe. He brings his expertise to Net Objectives as coach and trainer in lean and Agile software processes, tools, and practices; Scrum, XP, design patterns, and TDD; and acceptance testing. He still writes code. Working with a development team on real problems is one of his favorite things. He can be reached at *amir.kolsky@netobjectives.com*.

enable easier maintenance — is much harder as those activities do not have any immediate, tangible benefits. This puts pressure on the developers — from managers, customers, peers, and even the developers themselves — to take shortcuts.[1]

Developers employ a plethora of excuses to justify why they should not do their job properly and in a timely manner, including the following:

- "We don't have time now, but we will allocate time for it later."
- "We don't really need it now, but we will do it when we will need it."
- "I worked in another team that never did it, and we did OK."
- "We don't know how to do it."
- "We don't want to do it because it will take too long; if something bad happens, we will address it then."

"It" in the excuses above refers to any activity that should be, but is not, done.

Whenever we take shortcuts we accumulate technical debt. We are indebted to correct the technical actions that were done improperly.

Technical debt, very much like financial debt, accumulates interest — technical interest — and it does so in an accelerated fashion. When we take out a financial loan, we must pay off the principle as well as the interest on the principle. If we don't pay on time, we are forced to pay additional interest — not only on the loan itself but also on the interest we should have paid. If we do not deal with this situation in a timely fashion, things will get much worse very quickly.

When we have technical debt in our system, it is harder not only to make changes to the system but also to make the changes in the right way. The result is that more debt is added; this is compound technical interest.

Intention

It is important to note that sometimes the decision to incur debt is a conscious one. In the financial world we may elect to take a mortgage or a car loan. This is a calculated step designed to allow us to acquire something that we otherwise would have to delay for a period of time. Similarly, the development team may decide to take some shortcuts that will result in technical debt but may have a plan — a payment plan if you wish — to pay off this debt by fixing the actions that caused it. For example, the team, for the purpose of making a specific milestone, may introduce undesired qualities into its design. Since it is aware of these undesirables, it has a plan to fix them, at the price of reduced productivity in the future. This plan is agreed upon when the decision to take the design shortcuts is made.

In the case of legacy code, the team is usually faced with considerable technical debt. At this point a conscious plan must be made to map out that debt and see what should be addressed right away and what can be safely ignored — for now. For example, the lack of unit tests is considered to be a technical debt. We can intentionally ignore this debt if it pertains to an area of the code that has been stable for awhile and is currently not worked on or planned to be worked on in the foreseeable future. Again, this is a conscious, intentional decision to ignore this technical debt.

Responsibility

In most cases, however, it is irresponsible to create technical debt or to ignore it. Unfortunately, there are many aspects of the development process that can result in technical debt. It is the responsibility of the team members and, even more so, of the team's managers to be aware of technical debt and to create a culture where technical debt is not tolerated. This is a marked change from the way many managers behave, where they try to cajole the team into operating in a professionally irresponsible fashion because they only have the short-term delivery schedule in mind and are ignoring the maintenance aspects of the code.

In the remainder of this report, we will describe the various types of technical debt and detail their causes, their impact, and what can be done about them.

CODE-RELATED DEBT

The most common form of technical debt is associated with the code base, specifically with the design of the system. Coming up with the correct design for any system is one of the major challenges facing the development team, and there are many concerns that have to be kept in mind. Recognizing all these concerns and addressing them in the design is not always easy; the team may forget to address a concern or may not have enough time, due to scheduling pressure, to deal with all the concerns it should. Ignoring a concern will introduce a weakness into the design that may — and, in most cases, will — hurt us in the future. This is the design technical debt, and the longer we wait to pay it off, the harder it will become. This is due to two factors:

1. The longer we wait to solve a problem, the less we remember the problem's details. We run the risk of missing the fine points or even forgetting

about the problem altogether. The sooner we solve the problem, the fresher it will be in our mind and the less likely we will be to forget crucial details related to it.

2. The longer we wait to solve a problem, the more likely it becomes that we will build on the foundation of that problem, which will make it harder to undo. Moreover, it is likely that one wrong design decision will lead to a subsequent wrong design decision, creating a ripple effect into the future. The sooner we resolve the design problem, the less likely we will be to make suboptimal design decisions in the future.

There are quite a few types of design problems we can introduce into our code. In this section, we will explore some of them, including redundancy, lack of cohesion and focus, no separation of use from construction, inappropriate coupling, unnecessary complexity, and lack of clarity.

Redundancy

One of the classic design problems is redundancy. The definition of redundancy in software is this: a set of design elements is redundant if changing one element will always necessitate a similar change in the rest. Redundancy is traditionally associated with code and data but may also be manifested in other design elements.

It is the responsibility of the team members and, even more so, of the team's managers to be aware of technical debt and to create a culture where technical debt is not tolerated.

Since the definition of redundancy has to do with change, this is where the risk is. If, for example, we make a change to some redundant code (or design), we will have to change all the other places where that code (or design) exists and change them all in the same way. This introduces two potential risks:

1. We may forget one of the places in which we need to make the change.
2. We may make the change but not in the same way.

To attenuate these risks and reduce the uncertainty, we need to extensively test all the places where the change was made to ensure it is consistent, and we have to test all the other parts of the system to make sure we haven't missed a place we should have changed.

A classic cause of redundancy is the practice of copy, paste, and change. We might have some functionality that is similar to a new functionality we need to introduce. To get the new functionality in quickly, we copy the old functionality over and change the relevant places. This is an example of technical debt. We need to remove the redundancy we have consciously created through the copy-and-paste process.

In order to do this, we need to analyze the new and the original code and see where they are similar. Once the similarity is identified it needs to be extracted from both places and put in a shared location. Of the original and copied code, only the differences will remain.

Note that in a case like this it may well be that the team decided to copy and paste so that it could get the code working, tested, and released quickly in order to satisfy the needs of the customer as soon as possible. It is the team's responsibility in this circumstance to immediately pay off this debt. Delivering the functionality early is not the end of the team's work. The work ends when the technical debt is paid off and the redundancy is removed. This is an example of intentionally going into debt with the intent of clearing it as soon as possible.

The customer should not see the quick delivery as proof that no further work is needed. This type of debt is difficult to get rid of, and once it begins to accumulate, the problem becomes progressively worse. Effort must be invested to identify and eradicate redundancy at its infancy, when it is first created or at least as soon as it is identified.

It is important for the team to record sighting of redundancy in the code base. Each case should be discussed and a decision made on the risk associated with the redundancy and whether it should be resolved immediately or deferred. At a minimum, a comment should be placed in the code in the appropriate place to remind the developers of the redundancy.[2] When the developers happen to work in that area of the code, they should resolve the redundancy and pay off that debt.

A top-down separation of the code into cohesive units is simple and safe. It is the first step in dealing with debt-ridden legacy code.

Lack of Cohesion and Focus

The Single Responsibility Principle is a proven design principle that states that each software element should have a single responsibility; this design quality is called cohesion. Moreover, it is recommended that a single responsibility should not be spread across multiple software elements; this is called focus.

In the course of a code's evolution, there is a tendency for additional responsibility to be added to existing code elements. For example, a piece of code may call a remote database. Over time it may turn out that there are performance issues with that access. The developer may solve the problem with a cache and will implement the cache in the same code unit. From the developer's perspective it may seem this solution was more efficient; the developer doesn't have to jump between several source files, it may be easier to debug, etc.

The problem is that the developer has created more technical debt. By placing both responsibilities in the same code element, the developer introduced the risk of one responsibility influencing the other; this results in having to test all possible combinations of behavior of the two elements. This makes future maintenance of that code harder and slower and makes for larger, more complicated tests. Also, an embedded cache will be difficult or impossible to reuse, and so if a similar behavior is needed elsewhere it will be redundantly implemented there.

The code should not be left in that state. Once the new behavior is proven to work properly, the developers must immediately separate it into its own element and pay off the debt. Fortunately, modern development environments automate most if not all of the extraction process.

A top-down separation of the code into cohesive units is simple and safe and is the recommended first step in dealing with debt-ridden legacy code.

Use vs. Construction

Another classic cause of debt is the failure to separate the usage of objects from their creation. The ability to evolve the system's design depends, in part, on the fact that objects should not know whether they are interacting with other objects directly or polymorphically through an interface. Hiding object creation in factory methods or factory objects places the responsibility of deciding what to create and how it is to be made on the factory and leaves the client object with only the responsibility to use the object correctly.

This is best explained through an example. Assume a client object needs to encrypt some data. Initially, there is just one type of encryption and it is implemented in an encryption object. Later on, more encryption types are introduced, and we would like the client to use them.

If, initially, the client both created and used the encryption object, the client will have to change the encryption object to take into account the fact that more than one kind of encryption exists by implementing the logic to select and create the appropriate encryption. This breaks the Single Responsibility Principle and introduces the risk of errors in the client code.

Also, if more than one client exists to these encryption objects, which could arise later if it is not true now, then this knowledge will be in multiple places, thereby violating the no redundancy rule.

The need to change the creation process is the technical debt. This debt is easily avoided through the separation of use from construction.

If we use a factory to create the encryption object, then initially it would always create and return the same type of encryption. As more encryption types become available, the factory will choose which one to create and return to the client. Through polymorphism, the client will not be aware of the change.

It is fortunate that modern development environments automate the sequence of steps required to encapsulate construction. This means that even if our code does not follow this convention it is easy and safe to enforce it to do so. This is an easy debt to pay off.

Not separating the use from construction also creates a coupling between the client and the objects it directly creates. We will discuss coupling next.

Inappropriate Coupling

Incorrect coupling is another source of design debt. As we've seen earlier, if an object creates and uses another object, it is coupled to it in a way that makes it hard to change later.

Software evolves through the introduction of variation. Initially we have one way of doing things, and over time we discover more methods. The variations may be in behavior (e.g., encryption), process (e.g., payment processing), cardinality (e.g., varying number of participants), formats (e.g., encoding), and technology (e.g., frameworks, architectures, and operating systems), to name but a few.

Whenever a reference is made directly to any of these variations rather than to the concept they represent, debt is created. Eventually, when the variation occurs, we will have to change our code.

We can prevent this debt by making sure that we write the code properly without creating any coupling that is not absolutely necessary. This is captured in the following design rule: design to interfaces. By not committing to any specific implementation in the code we allow polymorphism to emerge. Together with the use of design patterns such as iterators, factories, adapters, and facades, our client code (and, again, there may be or may arise multiple clients) can be made completely oblivious to the nature of the services that it uses and hence not be affected when they change. This considerably simplifies the process of dealing with variation but requires discipline on the developers' behalf.

Note that encapsulation, or hiding, is directly related to coupling and will not, therefore, be discussed here.

Allowing variation to occur is important. Attempting to anticipate all the possible kinds of variation will lead to something completely different: unnecessary complexity. And unlike debt, which is relatively easy to get rid of, it is hard to get rid of unnecessary complexity.

Unnecessary Complexity

Unnecessary complexity is not a technical debt per se; the code may be beautifully designed. Unnecessary complexity is still undesired though, as it is a liability.

As much as we would like to build our system in a way that it is flexible — easy to change and maintain — we do not want this flexibility to be achieved by anticipating all possible changes and thus building the code to deal with every potential change. We would like our system to be flexible in general and to provide our developers with the skills needed to change the system in whichever way is needed.

Unnecessary complexity is still undesired though, as it is a liability.

Building unnecessary complexity into our code means that whatever we do, for the lifetime of our system, we will be burdened with having to deal with that complexity. The system will take longer to build, test, and debug, and it will take more time for anyone new to the system to understand it. The time spent building the unnecessary complexity could be put to better use, such as providing more functionality.

A common example of unnecessary complexity is the focus on building frameworks with high extensibility because of some perceived future need. Good frameworks evolve out of need, not out of design.

The risky way to build a framework is as follows:

1. A requirement comes in: send a specific status by TCP/IP once an hour.
2. We discuss the requirement and conceive potential variations: send status by e-mail or send it on some external event or send any possible status or report.
3. We get excited by all the variations and decide to build a framework that will handle all the variations we can think of: send anything anywhere on any event; the original request is just a special case.

There are several risks associated with this process:

- It takes much longer to implement the original request.
- It assumes that we get it right the first time. If it turns out that some variation (e.g., sending) works differently in reality than expected (e.g., more parameters, synchronous vs. asynchronous operation) then a change to the framework would be required. Making changes to frameworks is undesirable as it often requires changing every use of these frameworks. This means that code that has already been written, tested, and delivered may need to be modified, which may introduce instability into the system that incorporates it. There is also a psychological difficulty in accepting a change to a framework that we thought was complete, both from the morale perspective and also from the sheer mental effort required to ensure that the change in the design is consistent throughout the framework.
- If an anticipated variation does not show up for awhile, then when it finally is requested the working of the framework will have to be relearned, which could be more time-consuming then starting from scratch.
- Finally, what if the anticipated variation is not requested? All that effort would have been wasted.

Unnecessary complexity haunts us not only in the system's design but also in its functionality.

Does that mean that frameworks are evil? Not by a long shot. Frameworks are one way developers deal with redundancy. When a process repeats itself with minor variations, there is great value in pulling the common aspects into a framework and encapsulating the variations. This, however, is better done in retrospective fashion, as outlined in the following four steps:

1. A feature request comes in, and we implement it in the simplest possible way.
2. Another feature request comes in and is also implemented in the simplest possible way.
3. With the functionality delivered, we can now consider the design and change the system by pulling out the commonality between the two features. This is the beginning of our framework's evolution.
4. The framework is always minimal, has no unused features in it, and the psychological impact of changing it is minimal as we are always working

in change mode. Change is not alien to us: we are practicing emergent design.

The unnecessary complexity haunts us not only in our system's design but also in the system's functionality. Whenever we add features that are not absolutely needed, these too need to be implemented, tested, and documented. This results in even the simplest things becoming complex and hard to do. Consider, for example, a cell phone with too many options; even the simple things become difficult to accomplish. Where do these extra features come from? Here we can blame the product development process:

- Whenever our process requires customers to deliver all their requirements up front, the individuals will ask for things that are important to them but also for what is marginal and sometimes frivolous. They do so because they feel they have no other time to communicate their needs.
- Whenever our process allows developers to add features unchecked, we run the risk of adding stuff that is "cool," dealing with unrealistic boundary conditions, or optimizing the wrong parts.
- If our process has no customer feedback, we have no idea if what we implemented is what the customer wanted. If it is not what the customers wanted it will not be used.

Dealing with the unnecessary complexity is not hard but requires discipline. We need to get the process right. The developers need to focus on doing only what they have to do to get the job at hand completed, and for that they need to have a clear vision of the product as a whole and how the current work item fits within it. The practice of acceptance testing goes a long way toward creating this visibility and ensuring that the true value in the work done is always known. Frequent deliveries to the customers will ensure that we are on the right track. Allowing the customers to specify their needs on an ongoing basis will allow them to focus only on what they immediately require, knowing that if something emerges at a later time they will be able to get it implemented.

When unnecessary complexity is recognized, either in the system's design or its functionality, it has to be removed. This is an emotionally charged activity; it evokes fear, but it is crucial to go through with it. This also holds true when a feature is no longer needed for some reason, perhaps support for some obscure format or a system that is forever defunct. Dead code — code that is not used but was left in the system — is another example. Browsing through our code files, we can often observe large swaths of code that is commented out. The habit of commenting out code and leaving it there is a direct result of the fear of doing the wrong thing and the lack of trust that developers have in their skills.

It makes perfect financial sense to sell the big house once the kids move out and to move to a smaller one that is easier, and cheaper, to maintain, with lower property taxes and higher utility. The same holds true for our system. We need to know when to trim off unnecessary complexity.

Lack of Clarity

Not all design qualities are purely technical in nature. Clarity concerns itself with the readability of your code. There are several aspects to clarity, but eventually it all boils down to one thing: how easy it is to understand the code. The easier it is to understand the code, the easier it will be to make changes to it. Conversely, the harder it is to understand, the harder it is to work on it. Hence, not paying attention to readability is another factor that will put us in debt.

The qualities that we have mentioned earlier: cohesion, coupling, and redundancy have a direct impact on the code's clarity. If the code is not cohesive, it will be harder to understand. If the code is excessively coupled, understanding its behavior will require understanding the behavior of all the elements to which it is coupled. Lastly, redundant code is hard to understand because of its repetitive nature.

But there are other things we need to pay attention to when it comes to the clarity of the code, as discussed in this section.

Coding Conventions

Not defining coding conventions or not adhering to defined coding conventions will create inconsistency in the code. This will make the code harder for team members to understand. Issues to consider include variable naming conventions, indentation rules, brace and bracket placement, and line spacing. Not paying attention to conventions will create inconsistency in the code and make it harder for different developers to work on it.

Simple code reviews can identify the coding convention debt. Fortunately it is very easy to address by employing automatic tools that will format the code according to the established convention as part of the build. The team needs to invest in defining the convention and programming the tool to do the formatting. This is a simple debt to pay off.

Comments

Comments are code perfume. If your code is bad, you may slather comments on it to make it readable, but they do nothing more than mask its bad smell; in debt terms, they hide the debt, but it is still there.

Comments serve five different purposes in code. All but two are related to technical debt. The following are the five types of comments:

1. Comments that explain what a function or its parameters do. The need for these comments indicates that the name of the function is not clear enough; it is not intention-revealing. This is sure to cause clarity issues in the future, as the comments associated with the name of the function are not available where it is used. This may cause the function to be used incorrectly or will require the developer to jump back and forth between the code being written and where the function is implemented. The solution is to give the function (or class) a better name and get rid of the comment.
2. Comments that explain how the code works. The need for such comments stems from poorly written code, which may be too long, contain incoherent variable and class names, and introduce unnecessary couplings. The comment serves to explain what the unreadable code does. Therein, however, lies its failure: since the comment describes the code, it will need to change if the code changes. This is a prime example of redundancy, and like all redundancies, eventually results in the comments becoming dated and out of sync with the actual code. This is extremely confusing to developers trying to understand the code.
3. Comments that explain why the code does what it does. These are the good comments; they actually add value to the code.
4. Comments that mark places where the code was changed. There's no need to elaborate on the changes; these are litter. Every decent team has a source control system that should serve the same purpose.
5. Comments that hold meta-commands, for example, for the purpose of IntelliSense or document creation. These are not really comments but are rather commands for other tools.

Naming

The code maintained by the team needs to match the vocabulary of the customer. Not doing so creates a communication gap that is especially pernicious for new team members. Confusion because of a mismatch between the language the customer uses and that which is in the code can cause errors down the line.

Whenever the vocabulary of the customer changes, we must update the names of our variables. With the advent of modern refactoring tools, this is not too difficult and is done mostly automatically. Developers, however, tend to not do so on the grounds that they understand the code and they do not require it to be updated. This, of course, is a fallacy since for any developer, especially a new one, any deviation from the language the customer uses is a cause for misunderstanding and, hence, errors, as well as an inefficient maintenance process.

It is the responsibility of the entire team to weed out any discrepancies between the language of the customer and what is captured in the code. Code reviews are instrumental in identifying a failure to do so. Techniques like Commonality-Variability Analysis are useful in identifying and establishing the customer's vocabulary.

Deviating from the language of the customer is a cause for misunderstanding and, thus, errors.

TESTING-RELATED DEBT

Testing is another area that can get the debt accumulating. The purpose of testing is twofold: In the short term, the purpose is to verify that the task at hand was completed correctly. In the long term, the collection of tests — the test suite — serves to prevent regression as changes are made to the code base either by modifying functionality or working on the system's design.

Missing Tests

Missing tests are the primary source of technical debt vis-à-vis testing. Whenever tests are missing, it is hard to tell whether the feature was implemented correctly or what the actual scope of the feature is. If an exact definition of boundary conditions and failure modes is absent, it is hard for the developers to get it right. They will tend to do too much or too little or come up with their own interpretation of what needs to be done.

The solution to this initial problem lies in a practice called test-driven development (TDD). In a nutshell, no development is allowed to proceed without a test to guide it. The test must initially fail, indicating that development is necessary. The developers will change the system in a way that gets the test to pass, indicating that the mission was accomplished.

Some teams elect not to engage in TDD and prefer to write the tests after the code. This introduces extra debt. Besides losing the advantages of having the test up front, they also run into the risk of testing what the code does rather than testing what the code was supposed to do.

If the tests are not written at all, then the system has no regression-preventing protection. This means that whenever a new feature is released some parts of the system (those with missing tests) will not get exercised until the changes are released, which makes for a very long feedback cycle.

There are several reasons given for why the team shouldn't write a test:

- **"The test is too simple, I will write it later when it has some meat."** Good incremental development will start with very simple features and evolve them. Initially the test will be simple, perhaps even trivial. But there is more to a test than just writing it. We have to create the test, hook it to the code, hook it to the automated testing framework, and see that it runs when it needs to run. This takes some effort but is considerably easier when the test is small and trivial. Moreover, it promotes a critical design quality: testability. Delaying the writing of the test will lead to technical debt because it promotes tests not being written, as in the next excuse.
- **"This test is too complex to write now — tight schedule — I will write it later."** When we work in a system without complete code coverage, it is much harder to write the tests. The code's design may make it harder to connect the tests to the system; writing the test for the new functionality will require writing tests for the old functionality to establish a baseline, and the administrative overhead of creating the test also takes its toll. Preventing this debt is easy: write the tests when they are simple and fast; this will make it easy to modify and extend them when the system grows in complexity.
- **"There are just too many tests to write."** This is a classic legacy problem: we want to work on a legacy code base but it has no tests, and the task of creating tests for the system is daunting. The solution is easy: Do not write tests for the entire system. Identify the small subset of functionality you're modifying and write tests for it. This leaves the rest of the system untested, but we are no worse off than we were to begin with. The other reason for the need to write many tests is a problem in the design; excessive coupling, lack of cohesion, and improper abstractions also cause the test suite to bloat. The only solution is to revisit the design. Trying to use a brute-force approach, such as the usage of automatic test generation tools, will result in a huge amount of tests that test the code based on the current design. This makes these tests useless when we try to change the system's design, which we do all the time. The time spent fixing the design to enable the testing, is what pays off the debt. We have to do it, and the more we delay the more expensive it will be.

Dealing with Failing Tests

A common mistake that we see when dealing with test suites is ignoring failing tests. Every failing test must be analyzed, and the problem it manifests must be fixed.

Unfortunately, with large test suites, if attention is not paid to the design, tests may fail not because they show a defect but because of an addition of new tests or modifications done to existing tests. This indicates a design problem in the system. The quick way to solve it is to fix the test. The problem with doing this is that it addresses the symptom not the cause of the problem. What we need to do is to address the original design problem.

This is a crucial point. Teams that ignore it will very quickly reach a point where they will spend an inordinate amount of time dealing with fixing the tests to get them to pass. The incorrect system design is the debt but instead of paying off the principle by fixing the design, teams ignore it and prefer to pay off the ever-increasing interest by just fixing the tests.

This will cause loss of trust in the test suite; very soon the team members will begin to comment out failing tests just so that they will be able to get their real work done. We often hear the developers say, "I will fix these tests later, when I have time. Right now I will just comment them out." The commented failing tests become the debt.

Test Quality

The system's functionality is always in flux. This means that the tests that test that functionality are also always in flux. Not paying attention to the code qualities of the tests themselves will result in debt. Making changes to the tests will become harder with an adverse effect on the development time

Tests should be made small; if they are too big, it means that they test too much and they are not cohesive. Noncohesive tests are hard to change.

Tests should be made to run faster; if they are slow, they will not be executed as often as they should be. Because more code will be written between test runs, if a defect is introduced, it will be longer before we catch and fix it. The longer it takes to identify a defect, the longer it will take to correct it, so a few minutes of inconvenience can mushroom into hours and days wasted fixing the defect.

Note, also, that the code qualities we mentioned earlier have direct influence over the quality and number of tests, for example:

- Redundancy in code will result in redundancy in tests.
- Coupling in code will result in tests that have a lot of dependencies and are hard to implement and slower to run.
- Lack of cohesion will result in large tests both in length and in the number of permutations that need to be tested. Often, this results in tests that are much larger than the code they test. Also, when classes are not cohesive, tests must be written to guard the against side-effects the code for one responsibility can have on another. These tests have no business value, they are written simply to make sure "bad things don't happen." Cohesive classes remove the need for these unwanted tests.

DEFECT-RELATED DEBT

Defect-related technical debt could merit a report of its own. Defect debt is like a fine that you are slapped with when you weren't expecting it but that you still must pay off. If you procrastinate in paying it off, you will eventually pay a lot more because of penalties.

Every failing test must be analyzed and the problem it manifests must be fixed.

Defects should be considered debt because:

1. Business value is not accrued. In fact, a defect may actually result in negative business value by causing damage.
2. Other functionality may be built on top of the defect, which will also need to be fixed once the defect is corrected.
3. Developers have to stop adding value to the system and correct the defect. This delays the delivery of the new features.

The delay in dealing with a defect is also debt. Not only are the effects above compounded, but, in addition, the longer it takes to deal with a defect, the harder it will be to do so.

The longer it takes to deal with a defect, the more we have to relearn the system.

Fixing defects requires intimate knowledge of the system, what's wrong with it, and how to go about fixing it. This knowledge is often maintained in the minds of the developers and is — unfortunately — short-lived.

This means that the longer we take to deal with a defect, the more we have to relearn the system. This is both expensive and risky as the reacquired knowledge may be incomplete or inaccurate, giving rise to more defects.

Two factors affect defect handling:

1. **Feedback**. The sooner we get the feedback, the sooner we can resolve the defect. Hence, not providing feedback mechanisms will result in defect debt.
2. **Administrative delays**. The process flow in the organization should give priority to defects once they are found. Any queuing or batching of defects will result in a longer period between the introduction of the defect and its resolution; as we have seen, this creates debt.

Root-Cause Analysis

Root-cause analysis is a process whereby we attempt to find the root cause for a problem. It can be summarily described as "ask 'why' five times" and is a basic tool of lean implementations, from manufacturing to software development.

Any defect in our product is a liability — we already said it was akin to a fine — and it must be addressed. The real debt is in the root cause for the defect, and by discovering that, we can prevent similar defects from occurring in the future.

There are two questions that need to be asked to avoid having to pay the defect fine henceforth:

1. Why did we put the defect in to begin with? How did it get in? For example, it could be due to a requirement misunderstanding, a lack of technical proficiency, or a coding error, to name just a few. Once we figure out how the defect got in, we can take appropriate actions to make sure it doesn't happen again.

2. Why was the defect not discovered before the product was released? How did it get out? For example, it could be that the testing process was not thorough enough, the test did not match the requirements, or the testing environment is not the same as the release environment, to name just a few. Again fixing the cause of the defect's escape into production will prevent it from occurring in the future.

Note, however, that finding the root cause of the defects does not necessarily mandate that they be fixed. The decision of fixing or not depends on the cost of the fix relative to the cost of individual defect instances. Spending $10 million to fix a defect that costs the business $10,000 once a month makes no sense in the same way that paying $1,000 a month for a parking spot does not make sense if we get fined $20 a day for illegal parking. So, for example, we may acknowledge the fact that the testing environment and the deployment environment are different, but decide not to do anything about it because creating a duplicate testing environment is prohibitively expensive. In the end, it boils down to a business decision based on ROI.

ORGANIZATIONAL AND ADMINISTRATIVE-RELATED DEBT

Writing code is not the only thing that a development team does. There is the ongoing process of development and a lot of debt is accumulated there.

Code Organization

The way code is organized is important. As code is modified, its location may need to vary over time. Failing to do so, because of laziness or time pressure, will result in a muddied class and file structure that will make finding the places to make changes and then actually making the changes themselves more difficult.

Class cohesion is important. As classes evolve in functionality, methods may need to be moved out. Failing to do so may result in cohesion problems as well as code redundancy, where code is duplicated because it was not in the right place from the start.

Placing all the code in the global scope is also a problem. Utility classes should only be made visible to the classes that need them. Elements that only make sense within a specific context, such as exceptions or constants, should be scoped appropriately by making them inner to the scope or by using namespaces. Failing to do so may result in coupling to these classes outside of their intended context of use. This will prevent them from being removed or changed to suit the context.

Packages and Modules, Directories, and Files

Structural debt is accrued when files are not placed in the correct location. When a file is moved, we need to update its references, update whoever is referencing it, change the build system, and perhaps change the source control system as well. This takes time that the developer may not want to invest.

Putting files into the wrong directory makes future maintenance of the code more difficult. If the files of a specific product are spread around the file system, they may be referenced inconsistently; this means that future name changes or location changes will result in a lot of time-consuming manual tweaking.

Module and package management is also important. Files that change together should be put together. Files that do not change often should be placed separately from files that do. Putting files that tend to change together in the same place will ensure that changes will affect the smallest part of the system possible. If we put interface and implementation in the same file, then a change to the implementation will result in the entire file being marked as changed, which means that everything that uses the interface will change and this could have a huge impact on the system from a build-time and test-time perspective. If we were to separate them into two files, then a change to the implementation would only result in those referencing the specific implementation changing.

Naming

Keeping the names of the files and packages current from the perspective of their content and the customer vocabulary is also important. Not updating the names appropriately will result in confusion with respect to the code location and may result in duplication of work.

Merging

Working off multiple development branches and delaying merging will cause considerable debt: merge debt.

If this debt is not paid off throughout the development cycle, through the process of continuous integration it will build to massive proportions. Some teams can spend months merging their branches and dealing with all the defects that ensue.

The solution is simple: continuous integration coupled with an automated test suite. For a team that employs these methods, merging is a trivial activity. Because of the constant sharing of the code, there is little to no rework due to design changes; they are visible within minutes of their introduction into the code.

Another problem with merging is when we must deal with multiple release branches. Any change done to one branch may need to be manually applied to any of the other branches. This is obvious debt as you need to do the same thing in multiple places, and this means that all the risks in redundancy apply. What we need to do is merge all the variations into a single code base and define multiple products not by multiple release branches but by configuring the main product. The branches are simplified to hold configuration information only.

Merge all the variations into a single code base. Define multiple products by configuring the main product.

Build and Deploy Debt

The build and deployment systems are where we usually find massive technical debt. The build is the series of steps we take to convert the code base into executable format. The deployment process is the series of steps we need to take to make the executable format available to the end users.

Whenever these steps are not automated, they have to be done manually, which slows down everything by orders of magnitude. This increases our rollout time and the time between releases, which has a direct impact on our bottom line. This also increases the feedback time and makes responding to feedback much slower.

Aspects that contribute to the build and deploy debt include:

- Manual steps.
- Intermittent failures that are not solved at the root.
- Incomplete release documentation, including tacit build/deploy knowledge that is not relayed back to the development team.
- Manual testing, which takes longer and is more error-prone.
- Different development, testing, and deployment environments; the build process should be automated across all environments, not just the development environment.

Documentation

Documentation is another source of debt. Some processes have steps that result in nothing more than a document (e.g., requirements, analysis, design). Eventually these documents go out of date; this is because they are redundant — they all describe the same thing in different ways. They all describe the system being built. As the system moves to the coding phase, the developers do not, usually, have the time or the energy to go back and update these documents. It's not uncommon to hear some version of: "Why do I have to go and update the design document? No one reads it anyway."

Once the documents go out of sync with the code, they become useless at best. They can actually be dangerous. If the variation between the content of the document and the system is large, this will be obvious and the document will be ignored. If the variation is subtle, however, we may run into trouble, as someone may incorrectly think that what's in the document is what we actually do in the code and act upon it.

It is common to hear some version of: "Why do I have to go and update the design document? No one reads it anyway."

Paying off this debt or preventing it requires one of two approaches:

1. **Forcing it** — making the team update all the documents all the time. This will pay off the debt but is extremely expensive. But, if anyone will ever need to refer to the documents, they will be up to date. This solution is extremely difficult to enforce, and the teams resist it because they often do not see these documents referred to after their initial use.
2. **Abandoning it** — changing the process to do away with these documents. Find alternative ways of getting the benefits from the documents without the tedium of maintaining them. Possible alternatives include:
   - Acceptance testing — where the tests form the specification and test suite
   - Big visible charts — have the design be maintained on the wall where it can be seen (e.g., on a blueprint poster or a whiteboard; easy to change and has use for the team)
   - Generated automatically — have the documents generated from the code or from other documents where possible

When it comes to user documentation, it is important that this is kept up to date at all times. User documentation includes user guides and references, API descriptions, and online help content. This means that the technical writer or whoever has this responsibility needs to update the document as the code changes. The acceptance criteria of the task at hand includes the correct behavior as well as verified documentation.

Failure to keep the user documentation up to date will result in a mad rush before the final release date, which means that instead of writing or updating just a paragraph based on fresh information, the tech writer will need to write complete tomes based on information that may be many months old, at the same time disrupting the work of the development team with questions. This slows down the entire team and creates substandard documentation.

THE MINDSET THAT FOSTERS TECHNICAL DEBT

When it comes to established teams, which are often working on a legacy code base, their mindset could be a major problem. Granted, the system is not as good as it should be; it is what it is, but individuals must not come to work with a defeatist state of mind.

The team members and especially management must be aware that things will take longer to do. What they all need to realize is that the system, as objectionable as it may be, is working. Code that has survived the long history of the company is still around for a reason; it has value. Had it no value, it would have been abandoned already.

Unless expressly told so, the team must not assume any rewrite is pending. As such the effort must be put into making the existing system better. Common excuses for not making the system better include:

- "We can't do anything about it now, it's too complicated."
- "We are afraid to do anything about it, it's too risky, and we have no tests."
- "We have no idea where to start."
- "We don't know what it actually does."
- "We don't have time now."
- "We'll deal with it when we rewrite the system."
- "This code is terrible; it's not worth fixing."

This state of mind promotes a lack of respect for the existing system, and it will deteriorate rapidly. It is similar to an abandoned building that still has all its windows intact, but, once one window is broken, very soon all the windows will be broken unless the first broken window is quickly replaced. This happens because people lose respect for the building. Many people have no respect for their legacy code.

The fact that a legacy has a lot of debt in it does not mean that we can allow ourselves to make it worse. Unless you decide to abandon it, you need to take care of the old system.

CONCLUSION

There are many reasons for and causes of technical debt. Some are design-related, some are test-related, some are administrative, and some have to do with the build and deployment environments. Others have to do with mindset.

In all cases, we need to be aware of what we do to add debt — and stop it. Determine what the current debt is and then decide whether we are willing to pay the price of the debt through extra work and slower delivery or if we want to pay the one-time cost of solving the debt problem altogether. As long as the decision is conscious, either approach is sensible.

The team may also create debt intentionally, for example, in order to expedite some deadline. In this case, it is important that the debt be identified and that a plan is made to pay it off before development commences. You should not take out a loan if you do not know exactly how and when you'll repay it.

Debt represents risk. Technical debt is no different. It should be accounted for and factored into schedules and quality risk planning. It is also management's responsibility to ensure that debt does not accumulate beyond the team's ability to pay it off or else the development of future products will be jeopardized.

ENDNOTES

1. Part of the problem stems from the fact that there is no professional standard for software developers, but that is the subject of another discussion.
2. This holds true for any of the design flaws discovered.

SCALED AGILE ★ BRONZE ★ PARTNER

Drive from Business Value

BUSINESS-DRIVEN SOFTWARE DEVELOPMENT

Business-Driven Software Development is Net Objectives' proprietary integration of Lean-Thinking with Agile methods across the business, management and development teams to maximize the value delivered from a software development organization. This approach has a consistent track record of delivering higher quality products faster and with lower cost than other methods.

Business-Driven Software Development goes beyond the first generation of Agile methods such as Scrum and XP by viewing the entire value stream of development. Lean-Thinking enables product portfolio management, release planning and critical metrics to create a top-down vision while still promoting a bottom-up implementation.

Our approach integrates business, management and teams. Popular Agile methods, such as Scrum, tend to isolate teams from the business side and seem to have forgotten management's role altogether. These are critical aspects of all successful organizations. Here are some key elements:

- Business provides the vision and direction; properly selecting, sizing and prioritizing those products and enhancements that will maximize your investment.
- Teams self-organize and do the work; consistently delivering value quickly while reducing the risk of developing what is not needed.
- Management bridges the two; providing the right environment for successful development by creating an organizational structure that removes impediments to the production of value. This increases productivity, lowers cost and improves quality.

BECOME A LEAN-AGILE ENTERPRISE

Involve all levels. All levels of your organization will experience impacts and require change management. We help prepare executive, mid-management and the front-line with the competencies required to successfully change the culture to a Lean-Agile enterprise.

Prioritization is only half the problem. Learn how to both prioritize and size your initiatives to enable your teams to implement them quickly.

Learn to come from business need not just system capability. There is a disconnect between the business side and development side in many organizations. Learn how BDSD can bridge this gap by providing the practices for managing the flow of work.

WHY NET OBJECTIVES

While many organizations are having success with Agile methods, many more are not. Much of this is due to organizations either starting in the wrong place, such as focusing on the team when that is not the main problem, or using the wrong method, such as using Scrum or kanban because they are popular.

Net Objectives is experienced in all of the Agile team methods (Scrum, XP, Kanban) and integrates business, management and teams. This lets us help you select the right method for you.

LEARN TO DRIVE DEVELOPMENT FROM THE DELIVERY OF BUSINESS VALUE

What really matters to any organization? The delivery of value to customers. Most development organizations, both large and small, are not organized to optimize the delivery of value. By focusing the system within which your people are working and by aligning your people by giving them clear visibility into the value they are creating, any development organization can deliver far more value, lower friction, and do it with fewer acts of self-destructive heroism on the part of the teams.

THE NET OBJECTIVES TRANSFORMATION MODEL

Our approach is to start where you are and then set out a roadmap to get you to where you want to be, with concrete actionable steps to make immediate progress at a rate your people and organization can absorb. We do this by guiding executive leadership, middle management, and the teams at the working surface. The coordination of all three is required to make change that will stick.

OUR EXPERTS

Net Objectives' consultants are actually a team. Some are well known thought leaders. Most of them are authors. All of them are contributors to our approach.

Al Shalloway

Alan Chedalawada

Guy Beaver

Scott Bain

Max Guernsey

Luniel de Beer

SELECTED COURSES

Executive Leadership and Management

Lean-Agile Executive Briefing

Preparing Leadership for a Lean-Agile/SAFe Transformation

Product Manager & Product Owner

Lean-Agile Product Roadmaps

PM/PO Essentials

Lean-Agile at the Team

Acceptance Test-Driven Development

Implementing Team Agility

Team Agility Coaching Certification

Lean-Agile Story Writing with Tests

Technical Agility

Advanced Software Design

Design Patterns Lab

Effective Object-Oriented Analysis and Design

Emergent Design

Sustainable Test-Driven Development

DevOps

DevOps for Leaders and Managers

DevOps Roadmap Overview

SAFe®-Related

Implementing SAFe with SPC4 Certification

Leading SAFe® 4.0

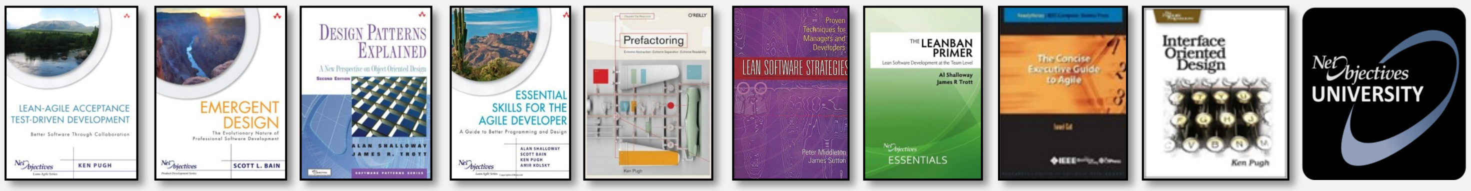
Using ATDD/BDD in the Agile Release Train (workshop)

Architecting in a SAFe Environment

Implement the Built-in Quality of SAFe

Taking Agile at Scale to the Next Level

OUR BOOKS AND RESOURCES


LEAN-AGILE ACCEPTANCE TEST-DRIVEN DEVELOPMENT
KEN PUGH
EMERGENT DESIGN
SCOTT L. BAIN
DESIGN PATTERNS EXPLAINED
ALAN SHALLOWAY
JAMES R. TROTT
ESSENTIAL SKILLS FOR THE AGILE DEVELOPER
Prefactoring
LEAN SOFTWARE STRATEGIES
THE LEANBAN PRIMER
ESSENTIALS
The Concise Executive Guide to Agile
Interface Oriented Design
Ken Pugh
Net Objectives UNIVERSITY

CONTACT US
info@netobjectives.com
1.888.LEAN-244 (1.888.532.6244)

LEARN MORE
www.NetObjectives.com
portal.NetObjectives.com