Project Management and Technical Debt

by Tom Grant and Declan Whelan
Contributors: Thierry Coq, Jean-Pierre Fayolle, Jean-Louis Letouzey and Dan Sturtevant

The adage, *"Act in haste, repent at leisure,"* definitely applies to technical debt. People under deadline pressure take shortcuts, which is one of several ways in which project management is a key determinant of technical debt. Project management is also important in other ways. For example, a realistic plan takes into account the impact of technical debt. A superior plan sets aside time to address technical debt.

As discussed in, *Introduction to the Technical Debt Concept*, the concept of technical debt is quite simple. However, when dealing with technical debt, you have a wide array of options for how to proceed. Therefore, it is important to know the full menu of these options, so that you can make informed selections among them. Whichever you choose, the general rule of thumb should always be, *Avoid creating technical debt whenever possible.*

Agile project management provides many opportunities to deal with technical debt. Here are some mechanisms you can consider in your projects.

Factor technical debt into release-level activities.

- ***Alignment between business and technical concerns***. Incurring technical debt allows you to release quickly today at the cost of the slower delivery in the future. Therefore, your approach to technical debt should be clearly articulated and negotiated to balance business outcomes over the near and and long term.
- ***Working agreement.*** The team should have a working agreement that includes technical debt concerns. For example, you could decide to have a list of 10 rules, detected by Static Code Analysis that the team will always adhere to. The agreement could include mentoring, pair programming or other practices for avoiding, measuring or dealing with technical debt.
- ***Short-term investment for long-term improvement***. When doing long-term planning, beyond the iteration, you need to ask the question, "How do we increase long-term productivity?" The answer includes short-term investments in reducing technical debt.
- ***Static code analysis***. You may want to include, in your plans, time needed to run a static code analysis tool, as a one-off effort to establish a baseline if one does not exist. Static code analysis makes technical debt visible in ways that other activities, such as code review, do not.
- ***Technical debt epics***. Some teams encapsulate technical debt reduction efforts into epics to be achieved within a release. This approach simplifies the planning effort, while helping teams think about the business argument for reducing technical debt (the "so that" part of the story).
- ***Technical debt iterations***. If technical debt exceeds the level where some regular refactoring or a few technical debt stories can address it, the team may need to dedicate one or more

iterations to addressing it. This can be particularly valuable if a new team is working with code built by an earlier team.

- ***Experimentation***. Although we have provided some estimated costs and benefits of many technical debt remediation measures, your results may vary. Additionally, some measures, such as code review, will vary widely from team to team. Other activities, such as test automation, won't have a direct impact on technical debt, but may have powerful second-order effects on the team's ability to deal with debt. Therefore, we urge that you plan for experimentation with these measures.
- ***Promotion of successful experiments***. Of course, the whole point of experimentation is to apply the results. If you find, for example, that implementing continuous integration gets team members to pay better attention to the state of the code, then you need to set aside time to implement the CI tools and practices.
- ***Education***. Schedule time to educate people outside of the team about the value of your technical debt-reducing activities. Exercises like the game *Dice Of Debt* can make a powerful impression in a short amount of time.

Include technical debt consideration in iteration-level activities.

- ***Code coverage***. Code coverage is important, and not just for catching defects. The tests provide additional information about the intended behavior of the code, and the testing paths may reveal excessive complexity and other technical debt patterns. Consider tracking code coverage at each iteration and focus on having the coverage trend upwards over time. Avoid having absolute code coverage goals (such as 90%).
- ***Cost***. Measure the amount of overhead that technical debt imposes in your iterations. This could be as simple as, "What percentage of your time do you spend unproductively due to technical debt?" You can trend this measure to visualize the impact of technical debt.
    - **Overhead.** Technical debt makes it harder for teams to address issues, as part of the regular overhead that constrains the amount of time available to implement new stories. You can make a rough estimate of how much harder it is to address defects and other issues, as a percentage of the regular overhead imposed on the team. For example, you might estimate that time spent on addressing defects in production would drop from 30% to 20%, if you took the time to refactor the code.
    - **Velocity reduction.** Technical debt also makes it harder for teams to implement new stories. You can make a similar rough estimate of how much easier it would be to implement a story, if you addressed the major sources of technical debt in the code. For example, you might say that the estimate for a story would be 8 story points instead of 13, if you eliminated the code duplication and poor error handling that plagues the code.

- **Developer happiness**. As "How happy are you working in the code base?" Often developer happiness, and productivity, will drop if they can't work effectively in the code due to technical debt. You can trend this measure.
- ***Refactoring***. When estimating stories be sure to include effort for any refactoring.

- ***Technical debt stories***. You may want to allocate a percentage of the team velocity for technical debt stories.
- ***Slack***. Maintain enough slack that, if time is available, developers can use it for refactoring.
- ***Mentoring***. Set aside time to teach other developers how to prevent technical debt. Consider pair programming or mob programming to promote deeper discussions around good technical practices.
- ***Flow***. Identify, within the iteration, whether technical debt-created impediments to flow exist (for example, only one person on the team can fix the code, because only that person can understand how it works).
- ***Static code analysis***. You may want to include static code analysis to chart overall progress towards reducing technical debt.
- ***Retrospectives***. You may want to facilitate a retrospective on technical debt and explore how it impacts the team and review ideas, perhaps some from this paper, on decreasing the technical debt in your code.

Consider technical debt when implementing user stories.

- ***There is a never a good reason to knowingly write bad code!*** Enough said.
- ***Code review***. Prevent technical debt by and deterring intentional technical debt by having code reviews. You may want to add this to your Definition of Done or add a "Review" column in your task board.
- ***Definition of done.*** Your definition of done can help avoid and reduce technical debt. Elements like required code coverage and code review are examples of injecting the discipline, at a story level, needed to prevent technical debt.
- ***Definition of ready*****.** The impact of technical debt is more severe the larger the code base gets. Having a clear definition of ready for new stories helps the teams build only the code that provides value.
- ***Pair/mob programming***. Consider pair, or mob, programming to share insights about good and bad code and to promote learning about avoiding and fixing technical debt.
- ***Boy Scout rule***. Ensure that you always leave the code cleaner than you found it. Technical debt is addressed one line of code at a time. By applying the Boy Scout rule you ensure that you get the biggest bang for your buck with technical debt reduction by focusing improvements on the code that changes the most. Consider including the Boy Scout rule in your team working agreement. (For a good discussion of opportunistic refactoring, see http://martinfowler.com/bliki/OpportunisticRefactoring.html.)
- ***Coding standards***. Coding standards should contribute to both the prevention and reduction of technical debt. Teams should agree together on a coding standard and hold each other accountable to it. You can use the [spreadsheet] as a starting point for identifying coding standards that have an effect on technical debt. Over time, the team can tune the coding standards to improve their ability to deal with technical debt.
- ***Making technical debt visible***. To keep focused on technical debt while coding it can be helpful to use tools that make the quality of the code visible. To build this kind of micro-feedback loop,

consider using editors that highlight coding standard violations, or tools that automatically run static code analysis and test code coverage.

Every team has to make decisions about where to start addressing technical debt. Ultimately, you need to create a working agreement within the team that covers how you assess and resolve technical debt. Not every team or project is the same, so you must exercise discretion about which clauses to put into this working agreement. And of course, it is prudent to have a pragmatic working agreement that is doable today and evolves over time.

For your convenience, the following chart can be your checklist for deciding which gets immediate attention.

RELEASE-LEVEL ACTIVITIES	
Alignment between business and technical concerns	
Working agreement	
Short-term investment for long-term improvement	
Static code analysis	
Technical debt epics	
Technical debt iterations	
Experimentation	
Promotion of successful experiments	
Education	
ITERATION-LEVEL ACTIVITIES	
Code coverage	
Cost	
Developer happiness	
Refactoring	
Technical debt stories	
Slack	
Mentoring	
Flow	
Static code analysis	
Retrospectives	
STORY-LEVEL ACTIVITIES	
Code review	
Definition of done	
Definition of ready	
Pair/mob programming	
Boy Scout rule	
Coding standards	
Making technical debt visible	