4 Steps to Persuade a Product Owner to Prioritize Refactoring

by Mike Cohn • 4 Comments

Teams often struggle to convince their product owners to allow time for refactoring, especially significant refactoring efforts. In this post I want to share a four-step framework teams can use to justify refactoring and persuade their product owners to allow it.

What Is Refactoring?

First, let's be clear on what *refactoring* means. Refactoring is changing the *structure* but not the *behavior* of code. This means a team would not undertake refactoring to fix a bug. Fixing a bug involves changing the behavior of the code.

A team could, however, refactor to prevent further bugs from occurring. In fact, preventing future problems is one of the most common reasons to refactor.

For example, a team has experienced a disproportionate number of bugs in a portion of the product's code. The team would like to "clean up" (refactor) that code so that when they revisit that code in the future, the code is not as fragile.

You can see why, then, that in changing the structure but not the behavior of code, refactoring can be understandably hard for product owners to prioritize highly. The "if it's not broke, don't fix it" mentality prevails.

Assessing the Cost of Not Refactoring

When a product owner is reluctant to allow a given refactoring, I've found it best for the

development team to present their argument in the language of the product owner. That is, to present an economic justification of the refactoring.

To do that, I suggest the following four-step process:

- Estimate the impact of the status quo
- Estimate the effort to perform the refactoring
- Estimate how much time will be saved after refactoring
- Estimate the payback period

Following these steps enables a team to put their case to the product owner economically. For example, they may be able to present a refactoring by saying, “We think this will take a third of an iteration to do, but it will pay back that investment in only five sprints.” When the case for a refactoring is made this way, it is much more straightforward for a product owner to decide whether the refactoring is worth pursuing.

Step 1: Estimate the Impact of the Status Quo

The first step is to estimate the impact of leaving the code as is; that is, the cost of not refactoring.

Try to do this by gathering actual data on time spent investigating, fixing, and validating defects in the code that you want to refactor. When data is unavailable, it’s OK to take an educated guess.

If you do need to guess, I recommend guessing conservatively so as to not overstate the case in favor of refactoring. In this way the team can be more confident they are truly recommending an advisable course rather than merely the course they’d prefer.

When a team gets a reputation for overpromising the benefits of technical work, such as refactoring, they make it harder for themselves to sell the benefits of similar work in the future.

An Example

Let’s consider an example that uses a mix of real data combined with some educated guessing by a team, since I find that is the most common.

In this case, our team digs into its product backlog or defect tracking software to determine how many bugs have been reported against a particular functional area within their system. They decide to look at data over the preceding six two-week iterations, approximately three months. I find three months a good compromise between using more data and spending more time on the analysis.

By looking at the work performed in the prior six iterations, the team determines that 12 bugs were fixed during that period. Eight were categorized as high severity. Four were medium severity. There were additionally three low-severity defects that were not fixed.

This is all data. If the team has tracked actual effort to fix these bugs, use that. However, many teams do not track that data and may need to guess at how long the defects took to fix. That’s OK. As you’ll see, this doesn’t need to be perfect.

Let’s suppose our team estimates that each high-severity bug took 12 hours to fix. That includes investigating, coding and testing plus any time that was spent discussing or documenting the solution.

Next, the team guesses that the medium-severity bugs each took half as long, so six hours each.

Putting all this together, our team spent 8 * 12 + 4 * 6 = 120 hours over the previous three months.

What About the Unfixed Bugs?

Remember, there were also three low-severity bugs that were reported but were not fixed. Handling those requires a judgment call.

If the product owner really, really, really would have liked those fixed but just couldn’t justify it, include an estimate for fixing them. On the other hand, if the team’s definition of low severity is

such that those defects are so minor, they don't need to fixed, leave them out of the calculation.

Why I Recommend Hours Rather than Story Points

I recommend using hours for these calculations because most teams do not put story points on their defects. However, if your team does estimate defects in story points, you can use points for these calculations. I'll proceed from here, though, sticking with hours. Nothing in the analysis changes except the units.

Step 2: Estimate the Effort of Refactoring

The second step in preparing to persuade a product owner to prioritize a refactoring is to estimate the effort of performing that refactoring.

To do this, team members should estimate the refactoring just like they would any other product backlog item. That means the estimate could be in story points or in hours.

The estimate will then need to be converted into the same unit used in step one when the team estimated the impact of not doing the refactoring. Since the team in that example used hours, I'll continue with hours for this step

Let's suppose our team has a product backlog item to refactor the troublesome code. I suggest they have the equivalent of a miniature iteration planning meeting during which they discuss what work they anticipate as part of the refactoring.

They might, for example, determine that only coding and testing are needed. But to better understand the work they split that out into three coding and two testing tasks. Each task is estimated quickly and roughly--again, this doesn't need to be perfect. In our case, let's say that is a total of 40 hours.

Step 3: Estimate How Much Time Will Be Saved

In the third of our four steps, the team estimates how much time will be saved after the refactoring is complete. This again will usually be a combination of data and educated guessing.

To see how this is done, let's return again to our example. In step one, our team looked at data and determined they had fixed twelve defects (eight high-severity, four medium-severity) during the previous six iterations. They estimated they spent 120 hours correcting those defects.

This means the team was spending 20 hours per iteration (120 ÷ 6 = 20) addressing defects in the code they want to refactor.

In most cases, it would be unrealistic to think that refactoring will eliminate all defects and all need to revisit old code. So let's have our team make an assumption here that the refactoring they propose will reduce by half the time spent on defects in that area of the system.

In other words, instead of spending twenty hours per iteration on defects in that part of the system, refactoring will reduce that to only ten hours needed per iteration.

It's quite possible the team could improve things further than that. But, again I recommend being conservative when estimating improvements.

Step 4: Estimate the Payback Period

It's time to see if the refactoring will be worth doing. To do that, divide the effort to perform the refactoring (as determined in step two) by the time saved (as determined in step four).

In this example, our team estimated that the desired refactoring would take 40 hours. And they estimated that it would save them 10 hours per iteration.

The payback period is calculated by dividing the number of hours needed to refactor by the number of hours saved per iteration as shown here.

Hours to Refactor

Hours Saved per Iteration

Since the team here plans 40 hours of refactoring and will save 10 hours per iteration, that is a payback period of four iterations.

A payback period of four iterations should be looked upon quite favorably by most product owners. This means that after four iterations the team's time spent on refactoring will be repaid and each subsequent iteration will have additional hours available compared to what the team has before refactoring.

How Short Does a Payback Period Need to Be?

A team and product owner should look for refactoring opportunities that have the shortest possible payback periods. But there's no strict guidance along the lines of "all refactorings have to be repaid in *n* iterations."

The suitable payback period is influenced by many factors on the project, including the urgency of other features and the projected life of the product. It would be hard to justify any refactoring on a product set to be retired in six months.

Putting the Argument in the Product Owner's Terms

By assessing the refactoring using these four steps, a team is able to present an economic case for the refactoring. Their rough estimates of the investment (the time to perform the refactoring) divided by the per-iteration savings gives the product owner a general idea of how long until the investment is recouped.

When a product owner is presented with an argument for refactoring in these terms, the product owner can make a rational decision based on the economic merits of the refactoring. That benefits the team, the product owner, and the product's users and customers.

What’s Your Experience?

Is your product owner receptive to requests to refactor? Or, if you are a product owner what helps persuade you to approve time spent refactoring? Please share your thoughts in the comments section below.

Posted: February 4, 2020

Tagged: product owner, prioritizing, refactoring

About the Author

Mike Cohn specializes in helping companies adopt and improve their use of agile processes and techniques to build extremely high-performance teams. He is the author of User Stories Applied for Agile Software Development, Agile Estimating and Planning, and Succeeding with Agile as well as the Better User Stories video course. Mike is a founding member of the Agile Alliance and Scrum Alliance and can be reached at hello@mountaingoatsoftware.com. If you want to succeed with agile, you can also have Mike email you a short tip each week.