Contents

Unit Testing: Good Vs. Bad Tests

Bugs or flaws in production code are a given. They simply aren't going to go away. Which is why any developer should know there is no proper coding without unit testing. However, some developers don't realize that unit testing is one of the essential parts of any software development cycle or process.

And it's the reason why getting it right is so critical. Everything from when to test, to whether to mock or not is essential. These and a few other factors we will discuss will help determine what constitutes a good versus a poor unit test.

Unit Testing: The Fundamentals

A unit is essentially the smallest collection of code which can be tested usefully. Roy Osherove, the author of *The Art of Unit Testing* said, "A unit test is an automated piece of code that invokes a unit of work in the system and then checks a single assumption about the behavior of that unit of work."

It is the initial level of software testing. There are two important aspects of unit testing that must be considered. The unit test case should, first and foremost, focus specifically on functionality. Secondly, it should do so in isolation. Externalities of any type should not be part of the unit test.

Unit Testing: How Much Is Needed?

The amount of testing undertaken can be an important contributing factor to how good or poor a unit test will be. The ratio between the production code and the test code could be anywhere between 1:1 and 1:3. For example, one to three lines of test code per every line of production code. However, the ratio can sometimes get as high as 1:10.

The mark of a good unit test is that it should offer invaluable feedback on the modularity and design of the code being developed.

When To Mock

When to mock may depend on what "school" a developer prefers. The London school, which is also know as the "mockist" school believes all mutable dependencies or collaborators should be replaced with mocks. The classical school, also known as the "Detroit" school believes only shared (mutable out-of-process) dependencies should be mocked.

Mocking should be done sparingly since mocks can:

- Result in violations of the DRY (Don't Repeat Yourself) principle
- Make refactoring more difficult
- Reduce the simplicity of the code design

When Not To Mock

Knowing when not to mock requires asking a simple question: Will the mocking replace a dependency in a unit under test with a stand-in for that dependency?

Examples of when not to mock could include the following scenarios:

- When the mock might override the logic of the mocked class
- When the mock has attributes, methods, or arguments that the real object does not
- When the mock has side-effects and behavior that differ to those of the real object
- When it's the business domain logic that is being mocked
- When it's mocking any infrastructure dependency that is directly attributable to the given business domain

When To Write Tests

When to write a unit test can be answered by asking this question: What is it that needs to be achieved for the code to be maximized? Here are three typical possibilities:

1. The number of bugs needs to be reduced early, which is the most important reason why unit testing may be needed
2. Code design needs to be improved, which is especially true in test-driven developments (TDDs)
3. Proof of the coding process is required, and a unit test provides documentation regarding the system and its functionality

Furthermore, unit testing can help improve teamwork, particularly within an Agile context. Collaboration benefits from insight, which happens when team members can review the logic behind code.

Are Unit Tests For GUI Interactions A Good Idea?

GUIs are increasingly critical components of software, and these days their testing is imperative. Unit testing can ensure that testing a GUI is more thorough. Here's how that is accomplished:

- Unit testing is more robust than GUI testing, which tends to be slower and more fragile
- Unit testing reduces lost time, both in the writing and execution of test cases
- Unit tests focus solely on functionality, meaning defects thereof are easily detected, whereas GUI tests focus more on the integration of functionalities

There is one important caveat however, and that is unit tests are not viable for testing how an application behaves under real-world conditions. For that, functional testing via the UI is the more effective testing methodology.

What Is A Good Unit Test?

A good unit test encompasses many different attributes or practices, including that it should:

- Be small and isolated
- Be about something highly specific regarding the code

- Have a ratio of testing to assertion near 1, thereby making it easier to identify any failed assertion
- Be singular and therefore with its own build-up and tear-down
- Focus on the behavior of code, making any failure thereof easier to test and assess
- Keep mocking to a minimum, since too many fakes can break when the code is altered
- Ensure there is a clear naming convention at all times, so the code is kept comprehensible.

More Attributes Of A Good Unit Test

A good, value-adding unit test should encompass even more, meaning the test should:

- Run in memory, for example with no DB or file access
- Consistently return the same result
- Full control is needed over all tested units; and
- Mocks or stubs should be used in isolation when needed.

Furthermore, a unit test should be fully automated, fast, readable, maintainable and trustworthy. A unit test is ultimately about catching bugs in the code. A test suite that never goes red equates to unit testing that is actually not working and is therefore of limited or no value.

An example of good unit testing can be seen with this partial extract of test code:

```
public interface IFormatter {
    String formatMoney(int money);
    String formatDate(Date date);
}

public class StandardFormatter implements IFormatter
{

    @Override
    public String formatMoney(int money)
    {
        int dollars = money / 100;
        int cents = money % 100;
        String centsString = (cents < 10 ? "0" : "") + cents;
        return "$" + dollars + "." + centsString;
    }

    @Override
    public String formatDate(Date date)
    {
        return new SimpleDateFormat("MM/dd/yyyy").format(date);
    }
```

This unit test will work for a variety of reasons, including:

- This class is testable
- It has two public methods
- It has no complex dependencies
- It is highly flexible, for example, testing can be done for different jurisdictions or countries where date or currency formats are altered

In its best practices for writing unit tests, Microsoft suggests that minimally passing tests are best. Microsoft claims that behavior being verified (tested) should be as simple as possible, thereby making the unit test more resilient to future alterations in the codebase and more aligned to actual behavior.

So, What Is A Poor Unit Test?

Likewise, a poor unit test can arise due to different attributes or practices, which should be avoided and may include:

- Non-deterministic factors in the code-base are problematic, since they are difficult to test; for example, time as an authentication factor in code can fail due to different time zones
- Side-effecting methods are also difficult and problematic to test, resulting in unwarranted complexities in the code
- Anti-patterns, secret dependencies, and other "back-door" scenarios are also highly problematic

An example of poor unit testing can be seen with this code:

```
@Test
public void test1()
{
    String[] countries = { "Canada", "AUSTRALIA", "spain", "Chad", "JaPaN" };
    Sorter sorter = new Sorter();
    CaseModifier caseModifier = new CaseModifier();
    String[] expected = { "AUSTRALIA", "CANADA", "CHAD", "JAPAN", "SPAIN" };

    String[] result = sorter.sort(caseModifier.toUpper(countries));

    Assert.assertArrayEquals(expected, result);
}
```

There are three problems with this unit testing:

1. The name is too vague – what is the unit test actually about?
2. What specifically is being tested – the sorter or the case modifier?
3. Is it to be assumed that two issues are being tested simultaneously?

A simple example of poor unit testing practice is that which doesn't consider future code. Adrian Bolboacă discussed this at the 2017 European Testing Conference. Business domain language can be critically important, including making a clear test name. He cited how vague test names like "ExceptionOnOverflow" or "CustomerTest" could be highly problematic for

future testing, collaborations, and clients. Hyper-specific names are needed, such as “WhenTooManyPlayersAreAddedAnErrorIsReturned” or “InvalidCustomerIsRejectedByOrder”.

Writing a viable, value-adding unit test is as difficult as writing good production code. Done right they should be both effective in discovering defects as well as being inexpensive to maintain.

Getting The Most From Remote Pair Programming

Extreme Programming (XP) is a methodology used in Agile software development that has two exceedingly worthwhile aims: enabling the production of higher-quality software and fostering a better quality of work life for the development teams. Remote pair programming is an example of XP, which unsurprisingly continues to gain a lot of traction during the ongoing lock-downs and social distancing of the global COVID-19 pandemic.

But remote pair programming is not without its quirks and uncertainties. For one thing, it's not easy to sync with a development partner when not in the same physical space. In this article we'll go through some of the benefits of pair programming, and then we'll offer some tips on how to achieve this form of programming in the best way possible.

The Benefits of Remote Pair Programming

As you can imagine, remote pair programming delivers the same benefits as that of in-person pair programming, including:

- Better code:
    - Collaboration means simultaneous code-testing, more timely fixes and improved code
- Knowledge sharing
    - The is the real essence of any collaborative process
- Focus
    - Being in constant contact with a peer ensures more focused and less distracted developers

Additionally, remote pair programming also has its own distinct benefits:

- Enabling technology

    - 5G and other developments will make collaborative technology more reliable and sophisticated in the near future
- Inevitability
    - Remote work won't simply disappear after COVID-19. The "new normal" is a global zeitgeist with remote work at its center

Common Attributes of Remote Pair Programming

Remote pair programming does have certain fundamentals which must be followed, and the four most important of those fundamentals are:

1. Collaborative code editing
2. Common sprints and sprint planning
3. Terminal sharing
4. Screen sharing

In addition, the approach is also key. For example, a pair may decide that a free-form collaboration works best for them. Or a more structured process may be needed. A programming pair may opt for a Driver/Navigator set-up, whereby the driver works with the code and implementation thereof, while the navigator brainstorms and reviews the code. A ping-pong approach can also be used to test-drive code and update it to the required standard.

Either way, a good idea is to swap roles during any sprint. It keeps the collaboration flexible, even-handed, and more experimental.

Basic Real-Time Collaboration Tools

Screen sharing tools are essential for any remotely-paired work, and some of the more generalist examples offering remote pairs some functionality:

- Zoom
    - The remote collaboration 'darling' of the COVID-19 pandemic has decent screen-sharing attributes, including viewing multiple screens simultaneously and its whiteboard annotation tool

- TeamViewer
    - A cross-platform and encrypted screen-sharing tool that doesn't require a VPN. It also has a built-in interactive whiteboard for real-time annotations

- USE Together
    - The most exciting feature may be the ability to use multiple mouse cursors on a shared screen

Other examples of popular, more generic-use platforms include:

- Screenleap
- Slack
- Surfly
- join.me
- Apache OpenMeetings

Cloud-Based IDE's

Intense software development work may necessitate advanced screen-sharing capabilities. A cloud-based integrated development environment (IDE) offers a fully functional development platform embedded in a browser that saves on development environment set-up, maintenance, and updating.

Two popular examples:

- Amazon Cloud9
    - This option provides a code editor, terminal and debugger directly from a user's browser, allowing for coding with the same editor using two separate cursors. It also has pre-packaged tools for over 40 programming languages, including C++, JavaScript, PHP, Python and Ruby.

- Codenvy
    - For greater security, Codenvy offers a self-hosted option. It also offers workspace permission management, automated workspace creation and integrated version control

Other notable examples include:

- CodePen
- Microsoft Azure Notebooks
- Observable
- Google Cloud Shell
- Codeanywhere

Live Share Extensions For Code Editors

For even greater security, some developers still prefer working with their own code editor, for example live share extensions.

Two examples include:

- Teletype for Atom
    - created by GitHub's core team. It is very secure for remote sessions since there is no centralized server; furthermore, Teletype encrypts all communication.

- Microsoft's Visual Studio Live Share
    - Files can be edited synchronously, with changes appearing immediately in the editor window. Real-time debugging sessions are also possible as breakpoints can be set up and the code worked on simultaneously in real-time

Measuring Remote Pair Programming

Agile is about consistently measuring performance. Pre-determined metrics during sprints help assess whether the remote paired collaboration is working optimally.

Metrics can include:

- Development time
  - How efficient is the development process as measured in time?
- Resolution time
  - How long does it take for software issues to be resolved?
- Resource usage
  - How efficient is the development with regard to the programming pair?
- Employee satisfaction
  - Are the developers enjoying their work and being paired?
- Error counts
  - These can determine software quality, so are all software errors being captured at all times?

Making It Work

Remote pair programming has its challenges, primarily because of the remote aspect. Mechanisms that can improve this type of collaboration include:

- Deciding on a collaboration approach from the outset
- Setting boundaries and expectation from the beginning
- Respecting each other's work schedules and "peak" times
- Setting aside "alone" time for research and singular work
- Being flexible, including with the collaboration tools used
- Having continuous software design as the key ethos
- Remaining open to learning from and teaching others
- Communicating clearly at all times

Pair Programming Is Essential

Ultimately, a developer should remember that the most fundamental benefits of remote pair programming are those of pair programming itself. Pair programming means continuous code reviews, simultaneously by two pairs of critical eyes.

This should equate to more stringent-quality code and fewer bugs. After all, remote pair programming should still be the epitome of real-time synchronous collaboration. This is as relevant to pair programming done remotely as it is to pair programming done in person. The technology is available to enable it – the rest is up to the given remote pair.

The words of Joseph Moore are worth noting here: "Some of the greatest achievements in history were produced by pairs of great minds working side by side on the same task. Software is no exception."

Furthermore, developers that embrace remote pair programming will be better prepared for the future.

How to Measure Quality in Agile Projects?

Agile requires a change in mindset from traditional methods to more quick and agile ways of working. In an Agile project we want to provide business value early and often to customers. But how do shorter and quicker cycles impact quality? How does Agile handle and measure quality?

Agile recommends using velocity and burn-downs. But they both predict the progress of the project not the quality.

This post will try to answer some of these questions-

1. How does Agile ensure quality is built in the software?
2. How can we measure software quality in agile projects?
3. How can we make sure metrics are used for good and not evil?

How Agile ensures Quality is in-built into the system?

The goal at the end of any sprint should be to provide a working application with minimal or no defects. We do this by adopting the following best practices.

- Each user story has a list of acceptance criteria and all user stories meet the criteria
- Each sprint team member has the same understanding of the user stories
- We use the 3-amigos practice to make sure we have clear requirements and design
- We use code reviews and have good code coverage

However no process is ever perfect and even if all of the above are in place we are still going to have defects. So let's look at some of the ways we can measure how good or bad we're doing and also if it's getting better or getting worse.

How to measure software quality in Agile projects?

Quality of software can be measured by the number of defects that are pushed to the client or found in production and the number of users it impacts. The key to good quality software is to ensure that critical defects are not released to production

Identifying the right metric is not an easy job. There are many to choose from but depending on your circumstances you need to decide which one is relevant and will best help you obtain the overall objective for your project. Some example agile quality metrics are as follows:

- User Story Acceptance= No of user story accepted by the customer/number of stories *100
- Review Effectiveness = (No. Of Defects found in Review)/ Total No. Of Defects found before Delivery (both Reviews and Testing) * 100
- Defect Leakage= (E/ I+E) *100
- Defect Removal Efficiency = (I/ I+E) *100
  where

I = Pre-delivery (Internal) errors including review defects as well as testing defects and
E = Post-delivery defects (External) or defects found after the release.

- Defect Density =Defects found/Size (actual in Story Points)
- Testing Effectiveness =(No. of Testing defects detected internally / No. of Testing defects detected internally + No. of Testing defects detected externally) * 100
- Test Coverage = % of code covered in automated testing
- And many more

These metrics, when compared with velocity, can give you important insight into the project. You need to choose the right metric or combination of metrics to capture based on your project complexity and type.

You can also include non-functional metrics that indicate the quality of the product. Some examples are:

- Portability(usability of the same software in different environments)
- Performance(software responsiveness and stability under a particular workload)
- Functionality(how well the software meets the user's needs)
- Usability(degree of ease of use and learnability, even for people with disabilities)
- Reliability(the software design and functionality can be depended on to be accurate)
- Compatibility(the ability of a piece of software or system to work with another)
- Maintainability(ease with which code can be amended or maintained)
- Security(software is secure from malicious attacks and hackers)

Good vs. Evil?

Agile metrics work best when they are used to serve as a lead indicator to any future problems. The best way to select a good metric is a 3-step process:

1. Consider the goals that you are trying to achieve
2. Develop metrics that would help indicate that you are moving in the right direction
3. Decide how best to capture these metrics

For example, if the goal of the project is to reduce the number of defects in production, use *defect leakage* to show if you are moving in the right direction or not. This is pretty simple to capture and gives you important insights in to the quality of your deliverable.

From the definition above we can see defect leakage is the number of pre-delivery defects divided by the number of pre-delivery defects and post-delivery defects. You should analyze any downward trends, identify the root cause behind the drop in the metrics and implement the right process

improvements to get you back on track. Good metrics should always – give you an opportunity to improve.

Metrics can become an important indicator of your project's progress and can help you identify process improvements, but at the same time, they can be easily misused or manipulated.

To ensure that you use the metrics for good:

1. Just enough metrics. You would not like your teams to spend a substantial time capturing data leaving development work aside and metrics becoming an overhead for the team.
2. Never measure an individual. Metrics are there to indicate patterns in your project. They should be defined carefully to measure your goals and should not focus on measuring individual's performance.
3. Never measure something because it is easily available. Just because a measure is easy to generate should not be criteria for capturing these metrics. The criteria should be based on what we described how it aligns with your project's goals.
4. Use metrics to track the progress of your project and improve the team's results. Metrics serve no purpose if they simply exist on your dashboard and are never analyzed to gain insights into the project. Analyze trends to help improve future deliveries

Let's take a scenario where a manager saw drop in defect removal efficiency from 95% to 89% while velocity increased from 10 to 12 story points. There were less defects reported by the sprint team before delivery than the last sprint. Higher velocity at the same time suggests that the team has spent more time in ensuring that more stories are delivered which could be compromising quality. It is likely that the effort that the team should have spent on review and testing was instead used for delivering more story points. The Manager needs to correct course at this time otherwise defects will just accumulate and need to be dealt with at the end of the project

Metrics are used for evil when:

1. You start using it for intra-team comparisons. Many factors define team dynamics and if you start using metrics for comparison, this may lead to low team motivation and further performance deterioration.
2. You manipulate data to show better metrics: Many teams know the goals they need to achieve and in fear of comparison or poor performance, tries to show better results by manipulating the data. This defeats the whole purpose of having metrics in place. This can be avoided simply by having just enough metrics, not treating it as a mode of comparisons between the teams and not tagging individuals with these measurements

It is important to have metrics in a project to understand if we are meeting the goals set and take corrective actions if goals are not being met. Metrics give us an opportunity to analyze our performance and help to reduce defect leakage to the customer. While they are vital for a project, it is important that we don't get obsessed with metrics and they become a burden for the team. Plan just enough metrics which gives you the right information to make sound planning decisions.

Agile Velocity and Capacity

Note: Velocity and Capacity are extensions to the Scrum Guide. Although it's not a formal part of Scrum, their use is quite widespread and recommended as a best practice.

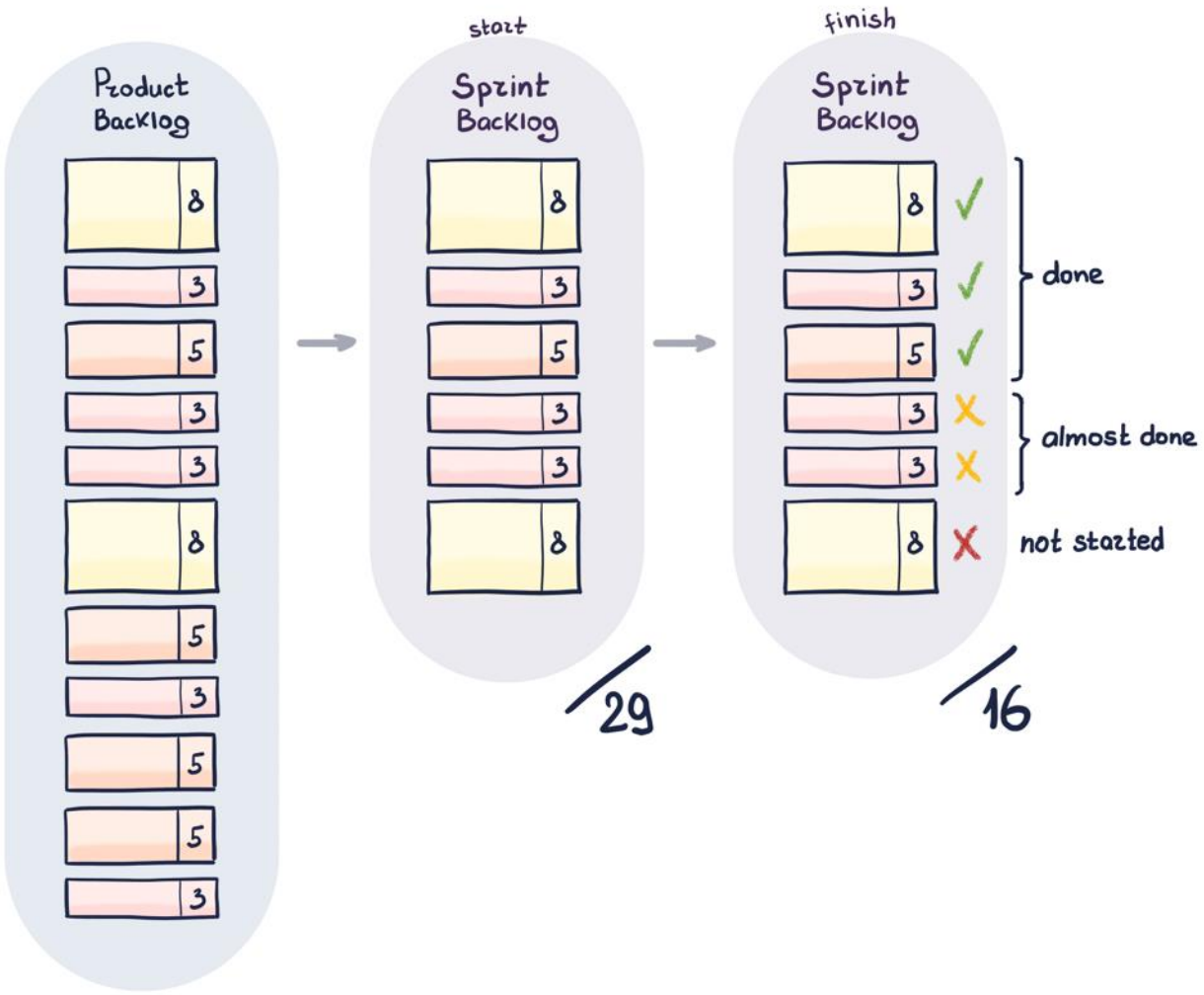
Velocity

Like a speedometer in a car Velocity helps a Team to forecast how many Sprints they need to develop new features. Following its ups and downs also allows a Team to analyze the progress they make as a result of any recently implemented improvements.

What is Velocity

Velocity is a measure of the amount of work a Team can complete per Sprint. A Sprint varies from team to team but is typically a week or two weeks, with a two-week Sprint consisting of 80 working hours. Just to keep things clear we can use an "hour" to show our speed on the speedometer. But what about measuring the amount of work? There are three basic approaches we can use:

1. Points. The most widely used and recommended as a best practice.
2. Man hours or man-days — a traditional waterfall approach.
3. Count by the completed Product Backlog items — the most recent approach also known as #NoEstimates. This works well for the mature Teams with stable Product Backlogs where the items are more or less similar to each other.



We can measure the instantaneous speed of a car at any time by just looking at the speedometer. Using that to predict how long the journey is not going to be accurate, as the speed constantly changes. So we use average speed for forecasting. The same logic applies to Velocity. After every Sprint, we can easily get the actual Velocity. But for predictions, we should use the average Velocity and re-calculate it at the end of each Sprint.

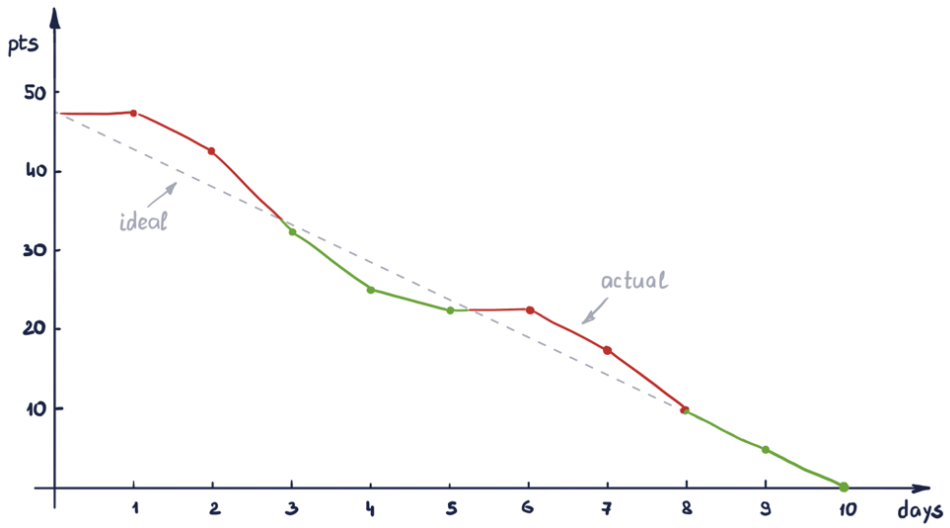
And while average speed might be helpful in getting an ETA to the end of the journey, it won't work so well for short distances. The quality of the road may vary, the traffic might increase. Everyone knows that feeling when you run into congestion and realize that the next 10 miles is going to take the same time as the previous 100. And if you try to calculate how many hours to your destination based on the overall average speed, it will almost certainly result in

unrealistic expectations. To get a better prediction, we should measure the average speed based on shorter sections of the whole journey. For Velocity we usually consider the last three Sprints, in what we call the "Yesterday's Weather" pattern.

Velocity in a Sprint

During a Sprint, you might want to know if your Team is able to complete the planned scope or if any corrections are required. The most common approach is to use a burn-down chart. There are three basic burn-down strategies:

1. Burn-down points by the completed Sprint Backlog items. The items must be small enough to be completed within one-two days. Otherwise, bigger items would result in more stair-stepping and less clear burndown line.
2. Burn-down points by the completed end tasks. This implies that Planning Poker is used to estimate the tasks in points. This approach takes more time during planning but doesn't require small backlog items to have a clear picture for Velocity.
3. Burn-down hours by the completed end tasks. This implies that the end tasks are estimated in ideal hours during Sprint Planning. This is similar to the previous point. Even though this approach is still widely used *Scrum Foundation and Scrum Inc. no longer view this as a best practice*.



Starting Velocity

For the most accurate forecasting we take the average of the last three Velocity measurements using the Yesterday's Weather concept. This approach is quite reliable, but isn't possible when a Team is starting their very first Sprint. There is no historical data which could be taken as an input, so one of the main goals during this time frame is to get that data.

On the first Sprint Planning session, the Team starts by using Planning Poker and coming up with points for each item. Team members often rely on a gut feeling to come up with the total number of the Product Backlog items that they can complete within the first Sprint. During the second Sprint Planning, the Team can use their actual "instantaneous" Velocity for the first Sprint, which increases the accuracy of estimations. On the third — the average of the two passed, which again is more accurate than the previous. Finally, starting from the fourth Sprint, the Team has all the data to make the most precise estimations.

Non Functional Items

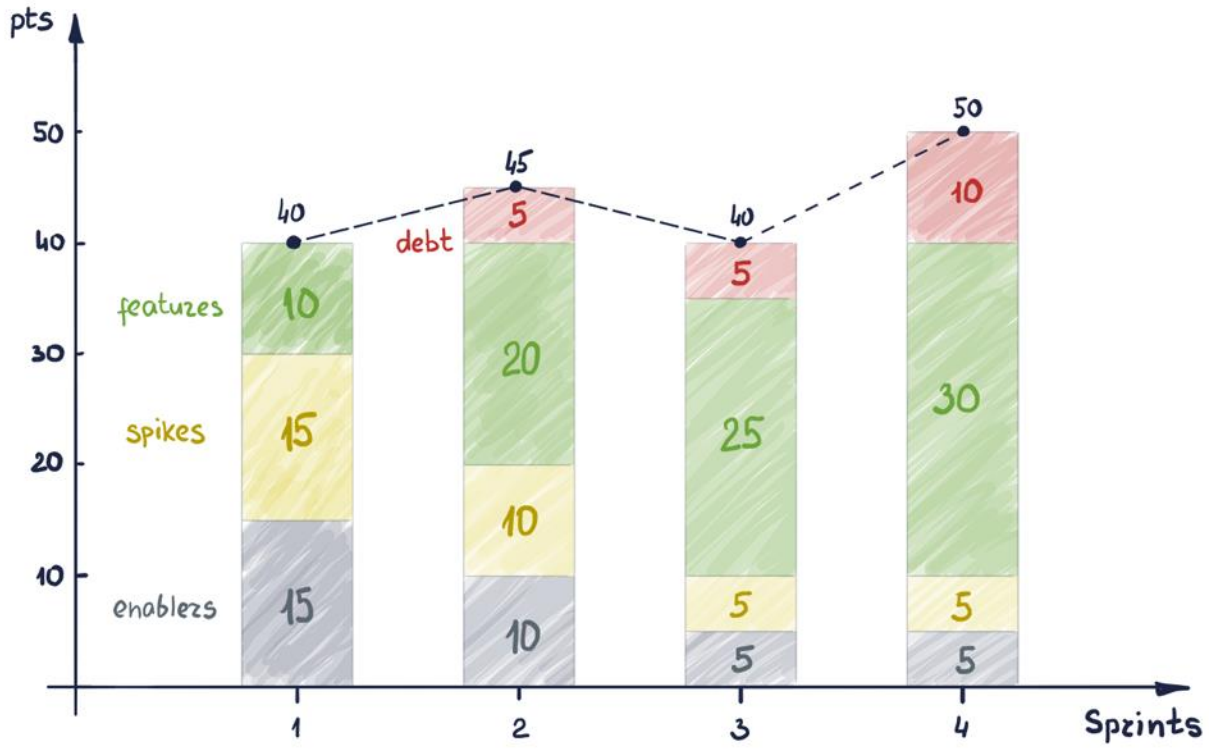
Velocity is often mis-interpreted and can create the wrong expectations. At first glance, the Velocity calculation is quite clear and straightforward. When someone hears "the completed backlog items", in most cases, they imagine functional items. In Agile we care about adding business value by primarily adding new functionality. But that's not the complete story. Every Development Team always has "non-value" items in their Sprint Backlogs such as:

- technical enablers
- exploration spikes
- refactoring and other technical debt

Not doing such investments means that a product will get blocked or stuck at some point of time. Technical enablers do not give anything tangible to the business right now, but they unblock future development. Exploration spikes are not features that can be demonstrated during the Sprint Review. However, they give a Team vital knowledge of how to do the right things in the right way.

Refactoring and technical debt deal with already delivered functional parts are much less interesting than the new features. However, only high code quality makes frequent delivery possible.

Since it's not something Product Owners can proudly present during a Sprint Review session these non-functional items get much less attention. To ensure a healthy balance between these non-value and value items, make the Velocity calculation as transparent as possible. Knowing the Sprint's Velocity is not enough; show the proportion between these basic non-functional categories. This aligns expectations and helps allocate some part of the Velocity for "technical stuff" in future Sprints.



Alternatively the Team could explicitly agree that Velocity is calculated based on functional items only. Don't consider any non-value ones in the formula. The Product Owner can then fully use Velocity on just functional features. The Team then runs the risk of losing clarity on why Velocity goes up or down as non-functional items are added to a Sprint.

Velocity and bug-fixing

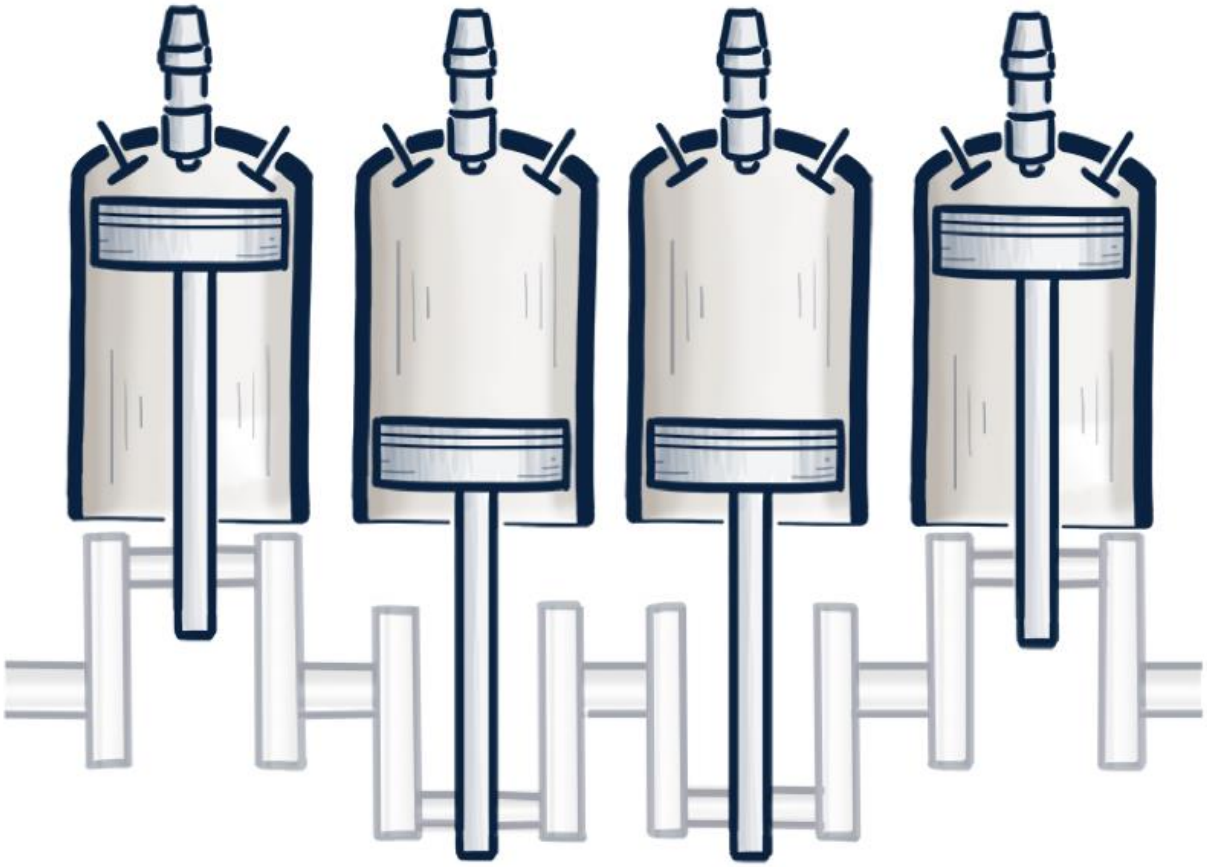
While bug-fixing can be viewed as part of technical debt discussed above, many treat it as an integral part of the development process. No one is immune to mistakes, especially in a complex software system developed by multiple Teams.

Bug-fixing has an impact on Team's Velocity. The more defects discovered in a Sprint, the less new functionality the Team is able to deliver. In order to mitigate useless frustrations in this case, it is better to have a clearly defined place for bug-fixing in Velocity. There are three basic ways to achieve that:

1. Include estimations for bug-fixing into every Product Backlog item during Sprint Planning. In case of more complex items, where there is greater risk, allocate more points for bug-fixing.
2. Allocate some fixed number of Velocity points for bug-fixing for all the Product Backlog items in a Sprint (like for the technical spikes or the technical debt).
3. Don't allocate anything for bug-fixing, so Velocity will be just lower and its oscillations will be less clear. In other words, you do the required bug-fixing every Sprint without reflecting anywhere on the effort spent. This impacts the Team's Velocity as the Velocity "automatically adjusts".

Number 3 doesn't require any overheads,but it isn't as transparent as the 1 and 2. Let the Team choose the most appropriate option to find their ideal balance between extra overhead vs decreased transparency.

Capacity



Capacity is a measure of the Team's availability during the upcoming Sprint.

What defines the speed of a car? The most basic is engine displacement. The higher the total volume the engine has, the higher speed a car can reach and the more operating load it can carry. The same is for a Team with its Velocity and number of the Product Backlog items.

Engine displacement is determined from the bore and stroke of an engine's cylinders and the number of the cylinders. Translating the metaphor back to the Development Team: the cylinders are the team members, the bore reflects the breadth, and the stroke — each team member's depth of knowledge and experience.

But the Development Team is not an engine, which has the same number of cylinders all the time and can run without stops. Capacity is not a constant for

the Team, it varies over time. Yes, the Team has its maximum Capacity when all the "cylinders" are in operation but various factors and conditions are permanently affecting it. And we need to consider these to help make better forecasts.

Variable impact

Basic factors which impact the Team's Capacity:

1. Number of workdays available in a Sprint for all the Team. The number of days in 2 two-week Sprints may differ depending on:
    1. Common public holidays.
    2. Common company's events (e.g. celebrations, teambuilding).

2. Number of workdays available in a Sprint for individuals. It depends on:
    1. Public holidays for different countries if you have a distributed Team.
    2. Vacations or time off for each team member.
    3. Time to attend meetings, events, activities outside the Team's interest (e.g. conferences, self-training, training for other team members, support of other teams).
3. Ramp-up and ramp-down of your Team.
4. Part-time membership of the Team.

Some Teams try to account these variable overheads by adding the corresponding items to the Sprint Backlogs. This approach seems extremely transparent, but it's not recommended. It stuffs the Product Backlog with items that little or no relation to the Product. Also, it adds useless bureaucracy which are simply not worth the level of transparency. Finally, it can't be applied to variable overheads such as ramp-up and ramp-down.

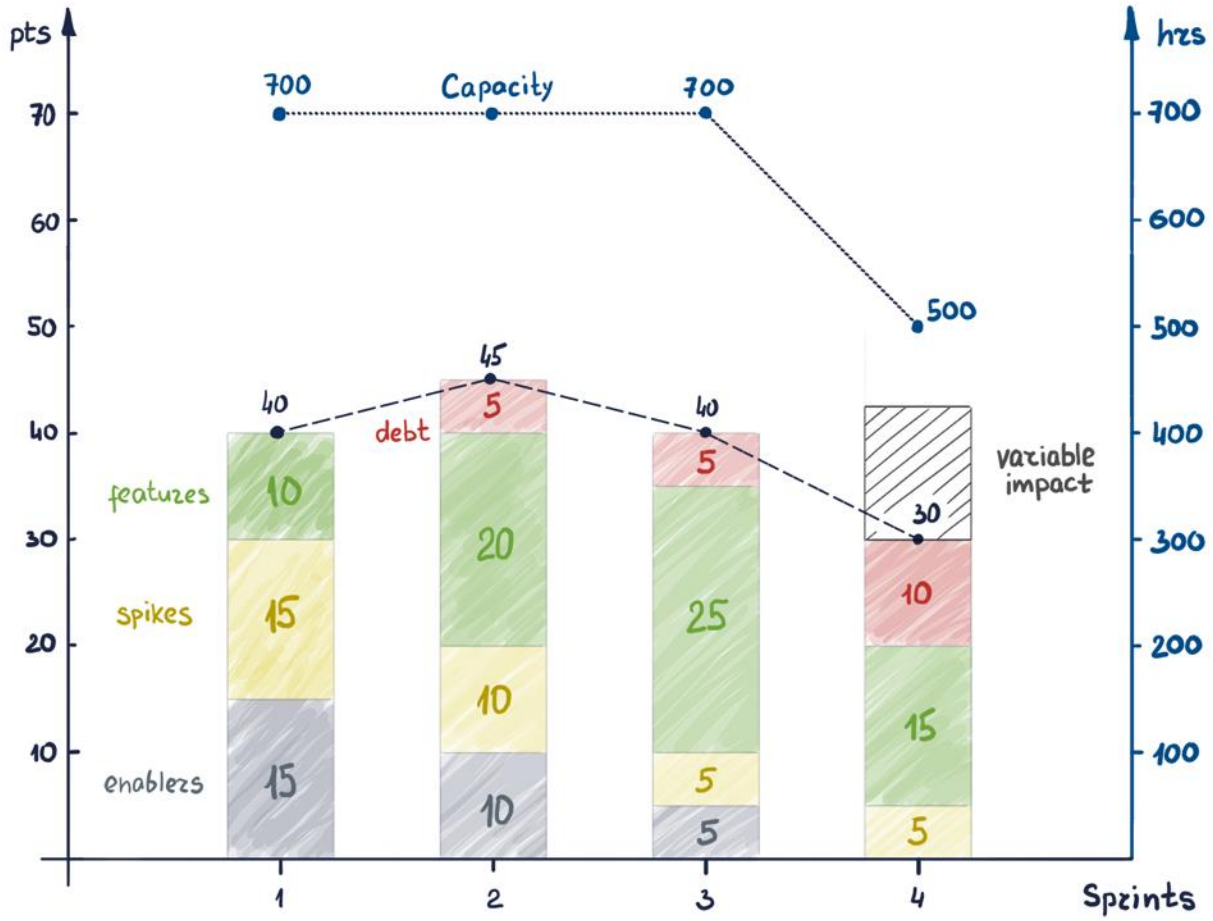
There is another more efficient way to account for changes in Capacity and get a refined Velocity forecast. The steps are as follows:

1. Calculate the maximum Capacity of your Team in man-hours, assuming that you have ten working days in a two-week Sprint and all the team members are available.
2. Calculate the adjusted Capacity, considering all the Capacity loss you will have in the upcoming Sprint.
3. Get the ratio of the adjusted Capacity to the maximum Capacity.
4. Apply this ratio to your current average Velocity.

Here is an example:

5. Imagine we have 9 team members, who work within 2-week Sprints 40 hours a week. The maximum Team's Capacity is: 9 * 2 * 40 = 720 hours.
6. Now imagine, 1 software engineer and 1 business analyst are going to have their 2-week vacations during the next Sprint and all the team will be away on holiday. So, the adjusted Capacity is: 720 – (2 * 40 * 2 + 9 * 8) = 720 – 232 = 488 hours.
7. The ratio is: 488 / 720 = 0.68
8. If the average Velocity for the previous three Sprints was 160 points. Then the refined forecast for the upcoming Sprint is: 60 * 0.68 = 109 points.

Putting Capacity together with Velocity provides transparency and visibility to your Product Owner and Business Stakeholders. Diagrams similar to the one below answer the questions like "why we did so little in the last Sprint" and "why we are able to take so few backlog items to the upcoming Sprint" even before they are asked. Such a clear picture also prevents demotivation of teammates, who may be frustrated for the same reasons. Clarity aligns expectations.

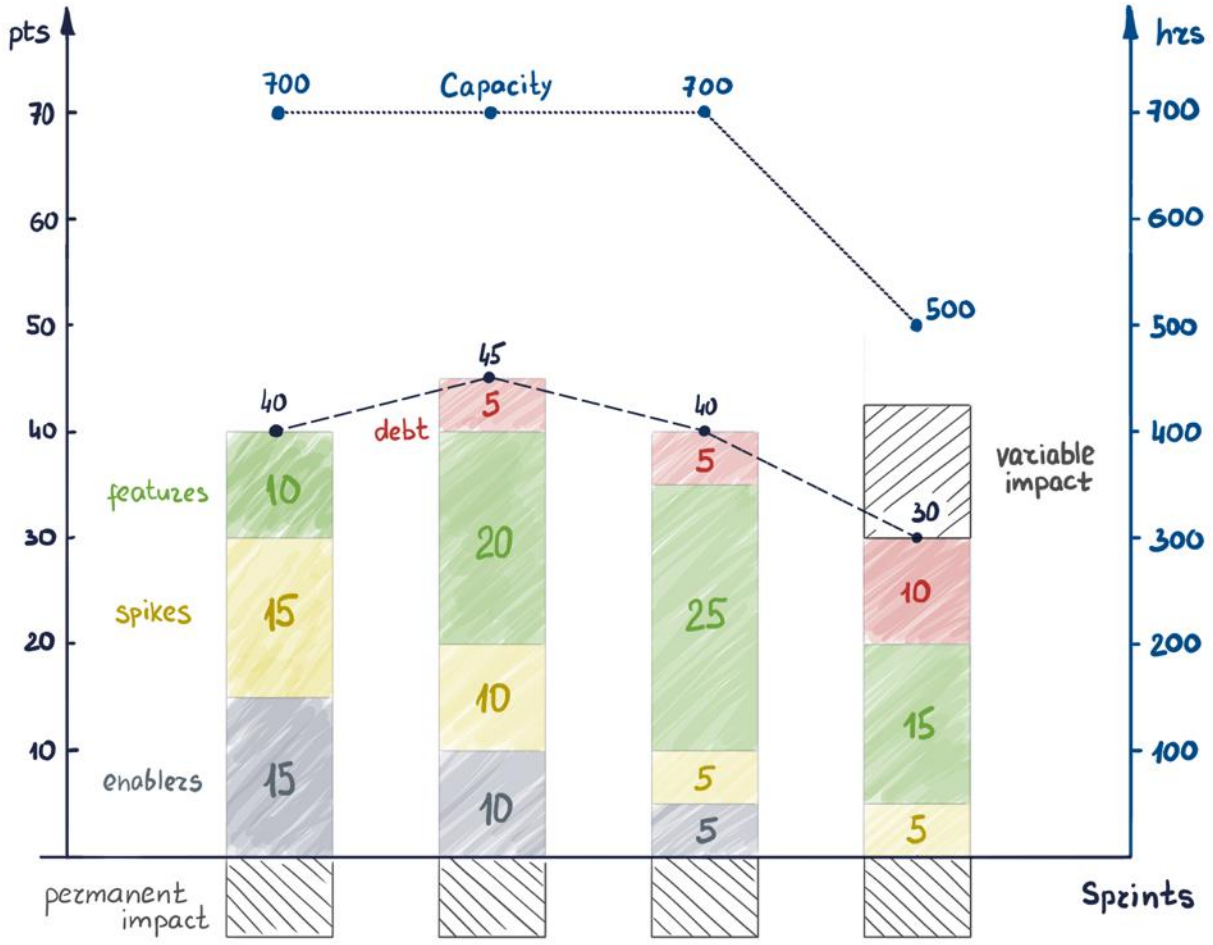


Be careful with this ratio as it may fail. It assumes that all the team members have the same level of knowledge and experience. Rough approximation works fine for a well balanced team. However, if 3 team members (out of 9) decide to have a vacation during the same Sprint, and they are the only software engineers in the Team, it will not adjust the Velocity by 1/3, but will drop the Velocity to zero making the Sprint a complete disaster. The same logic is applied to the balance between junior and senior team members. So, the Team should always think about the impact on Capacity when planning vacations, time off, conferences and other non-project events.

Permanent impact

Examples of permanent impact are regular Scrum events (e.g. Daily Scrum, Sprint Planning, Sprint Retrospective, etc.) and the time needed to prepare them. Like cylinders in an engine which all the time have to overcome the

engine's resistance, similarly, we have the fixed overhead caused by the regular events every Sprint. It doesn't change, it doesn't impact the Capacity and the Velocity over time, it is forever. So, there is nothing to do with the fixed overhead. Calculate it once, let everyone be aware of it and then just forget it.



Summary

So now, knowing what Velocity is with all its dependencies. Feel free to use it in the way your Team feels is best.

1. Start with a clear definition of Velocity. Agree with your team on which activities are included in the calculation.
2. Use the instantaneous Velocity to celebrate breakthroughs and adjust the course during a Sprint.

3. Use the average Velocity for planning purposes and evaluation of improvements. Play with the number of Sprints to get more accurate predictions. Remember all Teams have a different Velocity.
4. Consider the impact of variable Capacity. Find your own factors which impact Velocity and define your own approach for planning vacations and other days off.
5. Finally please remember, Velocity is not a KPI for managers; it is a powerful and helpful instrument in the arsenal of your Team.

Backlog Refinement

Product Backlog Refinement is a crucial task that makes Sprint Planning more predictable.

Imagine you are in a Sprint Planning meeting, discussing one of the Product Backlog items with your Team. It turns out that you need to get some clarification which could significantly impact estimations. However the Product Owner can't provide it right now as they need to talk to the Business Stakeholders. So what do you do? You could postpone the backlog item until the next Sprint? Or you could pull it into the upcoming Sprint and hope to get clarifications as soon as possible? Or maybe you could brainstorm with the Development Team to try to cover all the possible scenarios?

Alternatively imagine that the Product Owner is able to provide answers, but the Development Team has so many questions, that you have to extend Sprint Planning to four, six, or even eight hours to cover everyone's questions thoroughly. Throw in some off track debates and you can quickly see a whole day of Sprint Planning with people totally drained by the end of the meeting.

To prevent the sudden appearance of questions and disagreements when you should be actually starting your Sprint, you might want to think about being proactive by having a Backlog Refinement meeting.

Goals and activities

The main goal of a Backlog Refinement meeting is to make relevant items in the Product Backlog crystal clear for the Development Team. So that everybody understands all the items, realizes for what purpose each and every item exists and how they support product vision. Backlog Refinement should also be the meeting where the whole Team re-sorts items according to the latest knowledge and feedback and clarifies any issues to the level of detail required for the upcoming Sprint Planning meeting.

The term “Grooming” is often used instead of “Refinement” to help us think of the Product Backlog as a plant which requires weeding, trimming, watering and cleaning to make the product grow well. Watering keeps the focus on the initial vision and goals for the whole product. Weeding and trimming ensures that the Product Backlog only consists of relevant and valuable items. Cleaning (of leaves) means clarification of backlog items at a sufficient level. For example, you might break some of them into smaller pieces, add acceptance criteria, identify dependencies on other Teams, define architectural, technical and discovery spikes etc..

We can summarize this grooming metaphor into the following physical actions:

- Remove items that are no longer relevant.
- Sort the items to reflect the new priorities.
- Split the priority items which cannot be done in a single Sprint into smaller ones.
- Create new items based on new needs, circumstances, knowledge, and feedback.
- Estimate or adjust the estimates for the items on top of the list.

Best practices

How can you decide that your Backlog Refinement meetings are going well? Try focusing on the main goal — to make the Sprint Planning sessions easier to conduct. A good practice is to have two upcoming Sprints packed with ready-to-go items. In this case, Sprint Planning becomes relatively simple because everyone is familiar with most of the items, vital questions have been addressed, and rough estimations are already provided. Just a few finishing touches might be required. Sure brand new items can also still appear in the Sprint Backlog, but normally these should be in the minority.

How can you measure the readiness of backlog items for two upcoming Sprints? From the Scrum Guide, we understand the term Definition of Done (DoD). We use this like a “gate” where every backlog item needs to pass before it can get to the Sprint Review. Similarly, we can build a “gate” for the Sprint Planning.

We call this Definition of Ready (DoR). It is an agreement between the Product Owner and the Development Team of how thoroughly backlog items should be defined before they can be included into the Sprint; this can be as simple as a list of conditions which need to be met before moving forward.

Items from the top of the backlog go to upcoming Sprints first. However, we need to take care of the whole list of items in the backlog. Many Teams fall into a trap here. If the Product Backlog grows faster than the Team's capacity, you might find yourself in a situation where the bottom half of your Product Backlog is a "black hole" which you never want to get to. In this case, the Team can lose its perspective and becomes unable to suggest a better path for the product, which in turn increases demotivation.

You may find time to put things in order, but then there you also run the risk that these items will never be developed. "Black holes" can eat up the time with no return on the investment. Remember it is perfectly okay to close issues the Team are unlikely to tackle. Be brave enough to "trim the tail" on a regular basis of these "black hole" items.

Regularity

The Scrum Guide says that Backlog Refinement can take up to 10% of the Team's capacity. But what could be this 10% in reality? The right answer is "it depends". Each and every Team finds their own pace. However, I recommend to conduct it at least once a week, ideally, twice. It forces everyone to be in shape and work on clarifications instead of postponing the actual resolution until Sprint Planning. See our example of a 2-week Sprint calendar above.

Agenda

Finally lets look at how to run a Backlog Refinement meeting.

1. Reiterate the vision of the product. You might find this boring to repeat what everyone should know anyway. However, our brain functions in a way that we do need recycling to refresh the right focus in mind. The vision might sometimes change according to changes in the business

world. You will not be able to discover such shifts without checking new backlog items against your current vision.

2. Level down: take a look on your product roadmap for the next time period, say, one or two releases ahead. Identify the pain points you want to address and the gains you want to target.
3. Move no longer relevant backlog items to the "tail" and take everyone's confirmation to trim it.
4. Identify new backlog items together with their value hypotheses. Attach the basic outcomes of discussion and capture questions, and, assign action items.
5. Re-sort the list. Use the approaches to prioritization negotiated with the Team. Consider the expected value, uncertainty, risks, and complexity. Weighted Short Job First (WSJF) method might be helpful.
6. Break down big backlog items (Epics) to smaller pieces (User Stories). Attach details, wireframes and architectural agreements together with any design decisions.
7. Identify dependencies with other products and Teams.
8. Re-sort the list again.
9. Ensure that you have the Product Backlog items sufficiently clarified according to your DoR to pack two upcoming Sprints.

Backing Into Estimates

If you've been involved in software development for any period of time you already know that estimating projects is a difficult problem. Of all the projects I've worked on over the years I can only think of a handful where we came close to getting the timing, the cost and the scope close to the original estimates. There are plenty of reasons for this, scope creep, team dynamics, planning, domain knowledge, poor understanding of the requirements ….. the list is practically endless. Humans are just not biologically predisposed to work in absolute estimates. Some are better than others – especially if they understand the problem – but most of the time we're just wildly guessing how long the work is going to take and then making the team stick to that. Which is often very painful for everyone involved. This approach is also not a great way of keeping and growing your talent.

Time, Cost and Scope

We have three constraints, namely time, cost and scope. In a fixed estimation world, one of these has to give when it looks like the project is going off the rails. Either we change the time we take to get it done, or change the cost by adding more resources or we reduce scope. This unsolvable problem has led to all sorts of attempts to fix it, such as months of requirements analysis or prototyping or endless change request forms to make sure the consultants get paid for the work. But in the end someone is left holding the bag. Maybe it's the developers or the testers or simply that the customer just doesn't get what they want even after months of requirements sessions.

MVP

But there is a better way. Just assume that you don't really know what you want and start working on the *simplest possible* minimum viable product or MVP. Work with the customer or product owner to list what features you need in your MVP trying all the time to keep it really is simple or lean. And it turns out that we humans are much better at relative estimates. We can tell how tall

a building is relative to another building and we can tell how long something is probably going to take relative to the other tasks or features in our MVP. Use T-Shirt sizes, e.g. small (2 points), medium (4 points) and large (8 points) in your estimates to define how many story points each feature is going to take. Then add them to the sprint. Show the customer your work and ask what would they like to do next. Wash, rinse and repeat. Over time as the sprints are completed the application evolves into something the customer actually wants. The team has much less fear about meeting the project deadlines and there is typically lower stress and tension during the development process.

Agile Transformation

If you work in a small organization then moving from Waterfall to this Agile process is a relatively straightforward transition. It works especially well for a series of short projects as over time a well coached team figures out what will make them more efficient. Unfortunately it's not as straightforward for larger organizations. The mindset is and has always been, how much is a project going to cost and how long is it going to take. How on earth do you change that? We've seen many attempts where the company starts with the right idea but inevitably falls into the Agilefall trap where messaging is Agile but in reality the process is Waterfall. There are several approaches that can help break this vicious cycle.

Timeboxing

The first is timeboxing. Agree on a budget and based on the number of resources divide the budget into a fixed number of sprints. Push the app to production after each sprint and adjust the features based on the feedback from your internal and external customers. This continues until the last sprint. This is a paradigm shift in most companies and inevitably we hear the question – but what if I don't get what we want when the time runs out? The answer is that it's not working now, you're not getting what you want at the end of a project and if you take this approach you're going to closer to what you need.

Plateaus

The second approach is based on the Agile estimation we mentioned earlier and only works after you've gone through one or two timeboxing projects and everyone is working together as a team. Before each sprint the team chooses how many t-shirt size points they're going to try to finish. New teams don't always complete the estimated number of points, sometimes less work is completed and sometimes more points are completed. The goal is to get to a constant velocity, so that they're completing about the same number of story points each sprint. Initial sprints will probably vary wildly and if and when the team jells it'll start to plateau. Every team will be different but every team has a number of story points where they plateau. We use this velocity to get estimates for the stakeholders. So at the end of the day by not doing hard estimates we find a way to back into an estimate. It won't be exact but it will be an estimate of the relative time it will take a team to complete a project.

Predictability and Reliability

As you take the increasingly well traveled Agile road, it's worth asking yourself the question – why are you Agile? Everyone will probably have a different answer. If you don't have an obvious answer then I can tell you ours. We want to be Predictable and Reliable. Simple as that. No more, no less.

We Want Our Teams to be Predictable

In classic waterfall, teams make an effort to estimate the work, the deadline is set and we all run headlong into a death march to hopefully make the date. There is no predictability. There are too many things that can go wrong. Missed requirements, bad estimates, scope creep, changes in priority, changes in staff are just a few of the things that can wrong in a the lifecycle of a waterfall project.

Agile teams are much more predictable, especially after a couple of sprints. Typically the number of features or story points that the team can accomplish goes up and down in the earlier sprints. But after a few sprints the number of story points starts to be the around the same for each sprint. In the earlier sprints the team fixes the most obvious problems or blockers that are making them inefficient. If they get the right support the number of points stops varying wildly and reaches a plateau. This gives you a predictable number of story points each sprint at a team level. If you're not getting to a plateau then something else is going on. Talk to us, we can help.

We Want Our Teams to be Reliable

The Agile Testing Pyramid is a great place to start when you're looking to make your software more reliable. The majority of our tests will be unit tests that are written at the method level. If most of your code has complimentary tests then we're said to have good code coverage. Adding new features then becomes much easier and you don't tend to get weird side effect bugs anymore – assuming all your tests still pass.

We do a lot of mobile apps which tend to have backend APIs. Now some of the APIs are written by our teams and some are not. To stop any finger pointing we use Postman to create a suite of API tests. If the API tests fail then the backend developers fix their code. If the API tests all pass we know it's the mobile developer who needs to investigate the defect.

Websites and mobile apps have user interfaces that also need to be tested. So we write GUI tests in tools like Selenium or Cucumber. We write less GUI tests than unit tests or API tests but they're still essential for making our code reliable. The Agile Pyramid – especially when it's automatically run in a Continuous Integration server – provides you with an insurance policy of regression tests. This extra effort will increase the reliability and extend the lifetime of your code. Any new features added will not destroy any other functionality if all your tests pass during the build.

Radiating Information By Building An Agile Project Wall

Information radiators are valuable Agile tools in building transparency from the project team level to stakeholders. In short, information radiators are large and visible graphical representations of critical information about an Agile project or team. They can come in the form of a sprint task board showing the team's progress, a release burndown chart, or metrics dashboards, just to name a few. At its core, an information radiator seeks to foster communication and provide awareness of the current state of a project.One of the most effective ways to embed the act of radiating information into your team's culture and practice is by building and maintaining an agile project wall. A project wall is enough wall space in the team's workspace to display key information that communicates the current state of a project. Although a project wall can be built at any time, the most ideal scenario is constructing at least a first iteration of it as soon as the team is assembled for a project.

The Scrum Master or Agile Coach isn't the only one that provides input as to what should be on the project wall. The entire team should be asked to contribute what pieces of information would be helpful or vital to share to communicate project health and progress.

What would you post on a project wall?

A project wall might include a team's current sprint task board, story map illustrating the minimum viable product to be built, sprint burndown chart, release burndown chart, product backlog, team calendar with team member vacation schedules and key project dates, and a list of actionable outcomes from the last retrospective. We'll dig more specifically into some of these deliverables shortly.

Where should the project wall be constructed?

The project wall should be in close proximity to the team's workspace since the team is one of the primary consumers of the information being shared. This not only keeps the information easily accessible but also highly visible to the team to encourage keeping it current. Ideally, the team's workspace is also in a public and accessible area to stakeholders and those who want to remain informed about the team's progress and the project's health.

Who maintains the project wall?

The goal is for the project wall and the items published is to be simple enough that anyone on the team could update them. The team is responsible for maintaining the project wall and for keeping it up to date.

How often should the project wall be updated?

While every item won't follow the same cadence for being updated, the team should review frequently changing artifacts more often than those that will only change at the end of a sprint cycle. For example, the team's sprint task board should be constantly updated to reflect the tasks and their accurate status, while the release burndown chart may only be updated at the end of a sprint.

Let's build a sample project wall

In an effort to illustrate a more tangible example of what a project wall might look like, next we'll pull together some of the common artifacts you might use to build a project wall. The key thing to remember is that the example below is not a prescription for exactly what a project wall should look like, but rather an idea of what you might see on a wall. There are many possibilities and options for what might be posted on a project wall depending on the size and type of the project, level of rigor required, reporting expectations from your organization or client, etc.

Sprint task board

If you are working on a Scrum project, you are likely using some type of task board to visualize and track all of the work that must be completed for each user story. Including a sprint task board on your project wall is a great way of keeping the team's progress in a sprint visible to help team members coordinate their work, foster accountability, and serve as a visual during the daily stand-up.

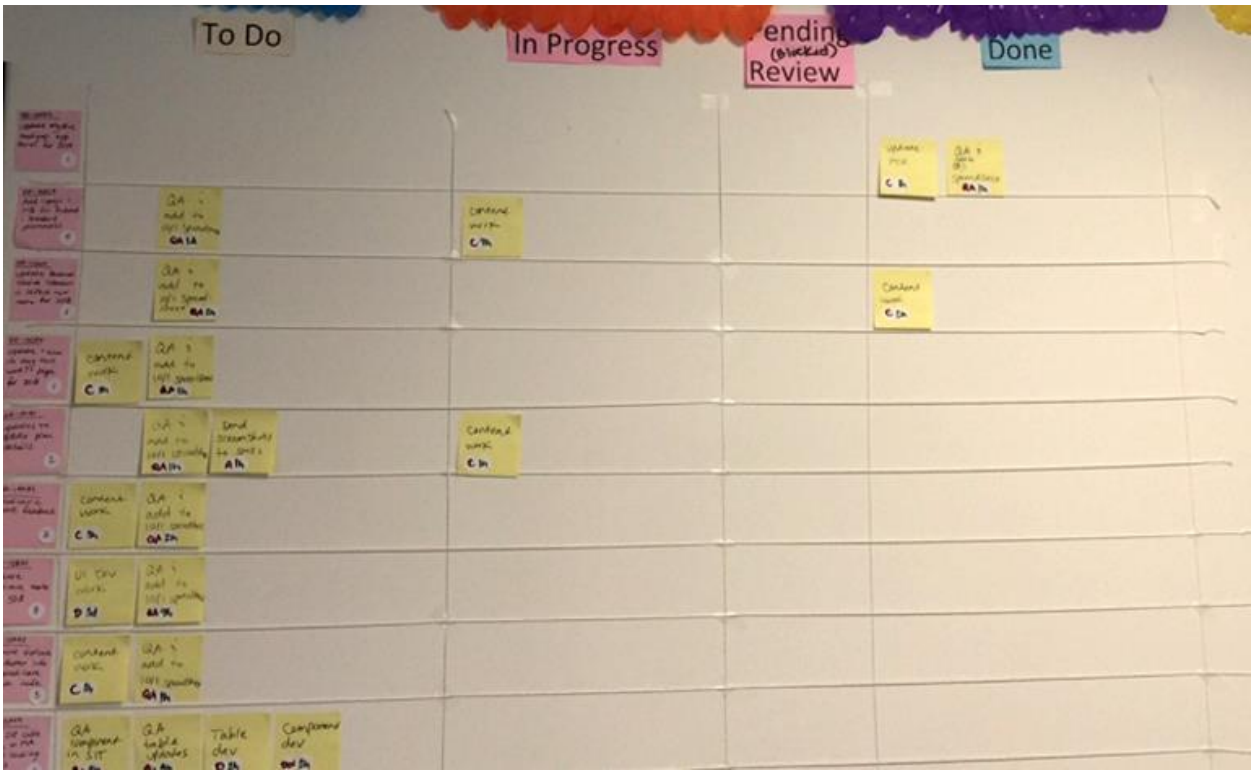


Figure 1.1 sprint task board example

Sprint burndown chart

One extremely helpful graphical representation to publish on the project wall and keep updated is the sprint burndown charts. The sprint burndown shows the rate at which the team is completing or burning through the user stories committed to in the sprint, as well as whether they are on track for completing their commitment by the end of the sprint.

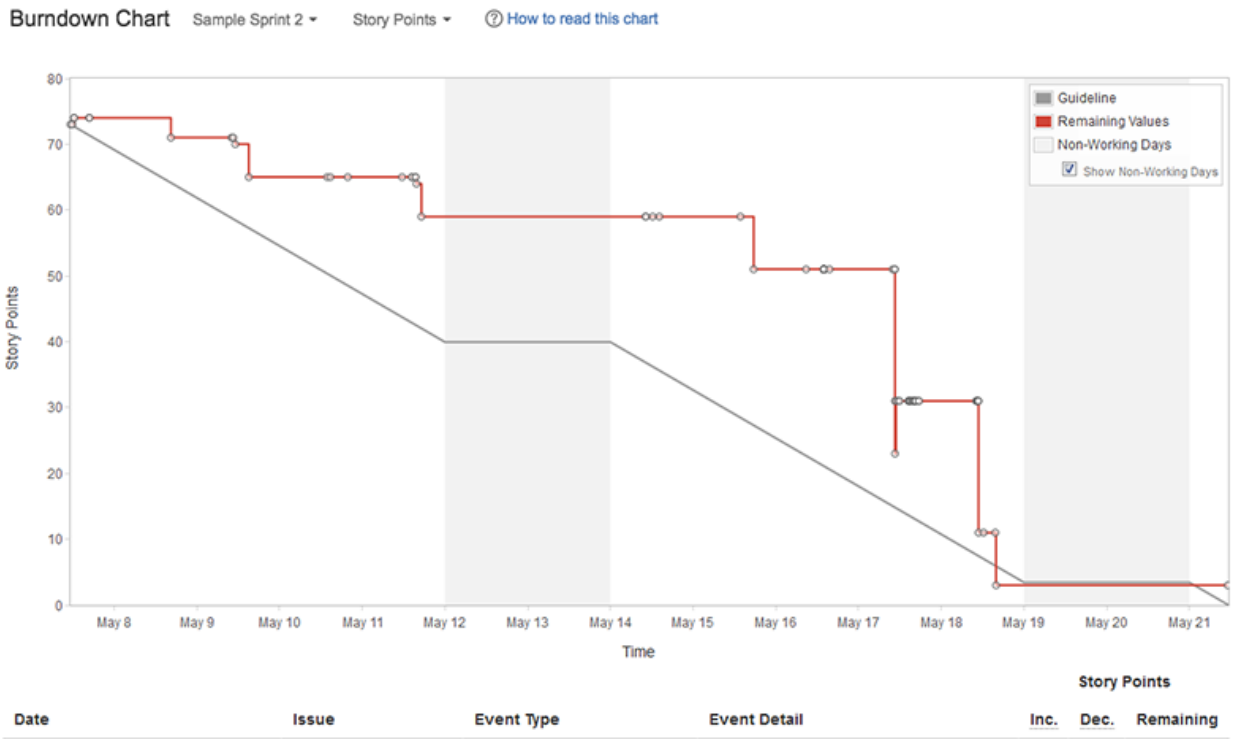


Figure 1.2 sprint burndown chart example

Team calendar

A team calendar is particularly useful in keeping not only key project dates such as release dates top of mind, but also team member schedules. By maintaining a low-tech version of the team calendar, the artifact remains constantly visible, easy to update, and aids in planning ahead for when a critical team member might be on vacation or supporting another project team.

Release plan

Something that is easy to lose sight of, but critical to keep at the forefront is the release plan. This is especially important in projects that span several months or more than a year, during which it is easy to lose sight of the end goal. These artifacts can come in a multitude of formats. At the most simplistic level, it should display known releases that are planned for the project, along with any target dates.

Milestone	Sprint	Release Date
Release 1	19 – 24	6/1/17
Release 2	24 – 27	8/29/17
Release 3	28 – 32	12/19/17

Figure 1.3 release plan example

Conclusion

An Agile project wall is essentially a collection of information radiators that communicate key information about a project's health and status. There isn't one correct way of building a project wall, which allows for adapting it to suit the needs of your team, organization, project, or stakeholder needs. In the end, the project wall is only as useful as its information is up to date, easy to consume and understand, and reflective of key project information.

Refactoring: The What, Why And When

During and throughout the development process, code can go wrong. Unfortunately, code can develop glitches or simply not function in the way the developer needs it to function. There are different methods available to software developers to work around and solve these coding issues. One popular method is refactoring. Refactoring seeks to improve code at its simplest, smallest iteration.

This article is specifically aimed at those developers who have are struggling with their code and have no idea where to start with refactoring. Additionally, developers who have limited experience with refactoring and need some guidance on how better to proceed with this aspect of programming will benefit from an exploration of this topic.

Refactoring Defined

Martin Fowler is one of the leading refactoring gurus. He authored a seminal book, along with Kent Black, that was first published in 1999 and titled *Refactoring.* Additionally, he launched an indispensable website: refactoring.com. Fowler defines refactoring as, *"A change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior."*

Sydney Stone, a leading software development writer, sees refactoring as a software development approach used in DevOps that *"involves editing and cleaning up previously written software code without changing the function of the code at all."* Conversely, refactoring does **not** mean rewriting code or "fixing" bugs.

Refactoring Fundamentals

Refactoring has some key aspects worth knowing:

- Crucially, it alters the internal structure of code without changing its external behavior
- Each iteration or refactoring should be a tiny transformation
- A single refactoring does very little by itself but can affect a significant change to the code's behavior as a sequence of iterations or refactorings
- It helps developers better understand code that may not be working and eases any future work with the code
- It should not be a "special event" but a normal, built-in part of programming
- It is a disciplined technique, not an ad-hoc "nice-to-have"

Refactoring Is Simplicity Personified

- There's a very simple reason why each refactoring should be as small as possible. By doing so, you ensure that no single refactoring can seriously disrupt or harm any coding system during restructuring. A small refactoring is also less likely to go wrong. Furthermore, it keeps the process less complicated.
- Taking the small steps inherent in refactoring can enable frequent testing. This should help a developer detect mistakes in code in a more timely manner. This 'keeping it simple' aspect of refactoring can help to resolve the 'technical debt' that exists due to flawed or dirty code.
- There are even ways in which refactoring can be objectively shown to improve the over-complexity of code, with one example being the use of cyclomatic complexity.This approach measures the complexity of a program, including a program's over-reliance on branches, loops, or switch/cases. Refactoring can help identify those factors that may require redesign.

Refactoring Helps Improve Code

Essentially, refactoring builds on that which already is, for example, existing code, and more specifically, its improvement. After all, the full title of Martin Fowler's book is *Refactoring: Improving the Design of Existing Code*. But Fowler doesn't see it as a means of cleaning up code. Rather, it's one specific technique a developer can use to improve the health of the existing code-base.

Fowler prefers to use the word "restructuring," which is a deft means of reorganizing code to make it more efficient and by extension, better. Ultimately, intelligent refactoring should make code more efficient and maintainable. You can see how Java code can be cleaned up and improved using refactoring in this example using part of the Test Task class:

This is how a part of the `TestTask` class looked before the refactoring:

```
TestEngine.select(project).ifPresent(testEngine -> {
  args.addAll(List.of("--add-reads", moduleName + "=" + testEngine.moduleName));

  Set<File> testDirs = testSourceSet.getOutput().getClassesDirs().getFiles();
  getPackages(testDirs).forEach(p -> {
    args.add("--add-opens");
    args.add(String.format("%s/%s=%s", moduleName, p, testEngine.addOpens));
  });
});
```

and here's how it looks afterwards:

```
TestEngine.select(project).ifPresent(testEngine -> Stream.concat(
    buildAddReadsStream(testEngine),
    buildAddOpensStream(testEngine)
).forEach(jvmArgs::add));
```

Another example in Java shows how factoring can help to improve extracting a variable:

Problem

You have an expression that is hard to understand.

Solution

Place the result of the expression or its parts in separate variables that are self-explanatory.

```
void renderBanner() {
  if ((platform.toUpperCase().indexOf("MAC
       (browser.toUpperCase().indexOf("IE"
        wasInitialized() && resize > 0 )
  {
    // do something
  }
}
```

```
void renderBanner() {
  final boolean isMacOs = platform.toUpper
  final boolean isIE = browser.toUpperCase
  final boolean wasResized = resize > 0;

  if (isMacOs && isIE && wasInitialized() {
    // do something
  }
}
```

Java | C# | PHP | Python | TypeScript

Further Refactoring Benefits

Refactoring is not only about helping improve or clean code. Its other benefits can include:

- Reducing technical cost to help prevent more costly errors from arising later
- Improving a code's readability, especially important for other or future developers who may work on the code
- Making the quality assurance (QA) and debugging process run more smoothly and efficiently
- Aid in preventing bugs from arising.
- It favors reusable design elements, including design patterns and code modules

- It favors low coupling, whereby different modules are highly independent from each other
- Helps adherence to the single-responsibility principle (SRP), ensuring a class in a software program has sole responsibility over a single part of the program's functionality

Further Refactoring Considerations

Refactoring is tailor-made for Agile software development: it encompasses doing each improvement one step at a time, followed by testing – in essence, an Agile approach. No wonder so many developers who use Agile methodology are huge fans of code refactoring. One Agile expert goes as far as to proclaim: "Refactor Or Die".

When considering refactoring, it is worth noting Martin Fowler's four principal reasons to refractor:

- It improves the design of software
- It makes software easier to comprehend
- It helps detect bugs
- It aids in executing the program faster

When To Do Refactoring?

When to undertake refactoring can be intimidating for the unseasoned or unknowing developer. Key reasons or times to do refactoring could include:

- Prior to key changes. Refactoring can be an invaluable exercise before adding any updates or new features to existing code
- To improve attributes of code. These can include objective attributes such as length, duplication, or coupling and cohesion of code
- To prevent code rot. Duplicate code, myriad patches, bad classifications and other programming flaws can seriously hinder code, which refactoring can address before they arise or get worse.

- To improve post-launch code. Refactoring code already released to market may sound bizarre, but can be beneficial to the given software and help detect or remedy flaws before the market does.

When Not To Do Refactoring

Refactoring is not a 'cure-all' during development. There are instances when refactoring is not fit for purpose and worth remembering:

- It should never be used if it could affect the performance of an application
- There are times when issues with code are so far gone that it would be better and more efficient to simply re-do it all from scratch
- Refactoring can be very time-consuming and become a convoluted process, often referred to as a 'a rabbit hole') and is thus best avoided when development time is tight or deadlines are looming
- If the software is working, leave it alone because refactoring could cause undue time delays and issues

The good thing about clean code is that it means design elements and code modules can be reused as the basis for code elsewhere. Refactoring, though not a solution for dirty code, can help achieve and maintain clean code.

Code should be as efficient to modify as possible, both in terms of time and cost. It should also be easy to understand. As Martin Fowler states, *"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."*