# Definition of Done and Code Reviews

Over the course of this reading we will discuss:

- Establishing a definition of when software changes are ready to be sent for approval “definition of done”.
- Some principles for conducting more pleasant code reviews with less arguments.

In most computer science classes you have most likely submitted programming assignments at the end of a due date with little regard to what happens to that code, receive an unchangeable grade, and never interact with that code again.

In the real world, the code we write and the mistakes we make will show up again and again as we modify a long term project.

Processes need to be in place that attempt to ensure that the code we provide to our teams meet the expected goals, have a reasonable degree of quality, and introduce a minimal amount of bugs into the system. While this seems reasonable, its execution is complicated by the egos of ourselves and our team members, who are often highly sensitive to criticism of their work by their peers.

In this reading we will discuss two processes for ensuring that checked in code meets the team standards, while keeping the process as simple and painless as possible.

These processes are:

- Having team wide “Definition of Done".
- Having members review the code on every pull request.

# Definition of Done

Doing a pull request and doing a code review can be extremely time consuming when pull requests are constantly coming in. In order to make sure that teammate time is not being wasted, code needs to meet some set of standards before being considered ready to submit for review.

This concept, called a “Definition of Done” can greatly vary across organizations and teams but is simply a set of standards that the team has agreed to mean that the individual developer can say they have finished their task and submit it for review.

The Definition of Done that we recommend using for this class is the following:

- **The code builds**
- **Any and all tests are passing**
- **Code matches team style**
- **No unused variables or import statements**
- The task as understood by the developer is completely finished
- Involved Acceptance Criteria met (The tests on the user story)
- No known bugs

The elements in bold are particularly easy to check, and under no circumstances should you submit code that does not pass these checks. Otherwise you are simply wasting your teammates' time.

The top two should each be testable by the click of one button each, one that builds the project, and one that runs all the project's unit tests. You need to learn where to find these in your IDE, most IDEs have them or an equivalent menu action.

The next two, code matches team style and no unused variables or import statements, can be easily automatically accomplished via a programming cool called a linter, which formats your code to a style guide, typically each time you save your code.

To find a linter, just look up the name of the technology you are using and "linter" and you will find many different options. All of your team should use the same linter configured to the same settings.

The three remaining criteria, the task as understood by the developer is completely finished, involved Acceptance Criteria met, and No known bugs, are at least subjective enough to where you might make a mistake. This is what your teammate is supposed to be checking.

While this list is similar to what your teammate will be looking for when they review your code, it is your job to make sure all of these criteria are met to the best of your knowledge when you submit a pull request.

# Code Reviews

At its core, a code review is a process for having your team view, suggest changes to, and/or approve your code.

We do this not because of a lack of trust between teammates, but because even the most experienced developers produce bugs and misinterpret requirements.

This process is intended to reduce the amount of new bugs introduced, as well as ensuring that what is produced lines up with what the team is expecting.

There is no correct definition of what happens during the code review process, it varies between teams and

organizations.

Some teams may review the submitted code all together on the projector, with any anyone offering comments as one person scrolls down the changes. This is probably the most formal form of a code review.

A less formal process might be having the code author and the reviewer sit together and just talk through the changes. While this is probably ideal for this class, it requires a time commitment from at least two team members, and therefore might not be possible all the time.

The least time and availability intensive process would be for the code to simply be reviewed with the reviewer by themselves, viewing the changes on GitHub. While this is not ideal, this is the easiest to do and therefore will probably happen most of the time.

Code reviews seem simple, we look at the submitted code and we suggest changes, but unfortunately it's not, due to the human factor.

A person submitted their code because they think it's great and finished, and sometimes you have to tell them that it isn't. This can be a very hard message to convey, especially because programmers can be very sensitive to criticism of their code.

This can be even further complicated when the review process takes place over writing, where miscommunication happens all of the time. Here it can be extremely easy for constructive criticism to be taken as a personal attack.

How to Do Code Reviews Like a Human (Link below) makes the following recommendations for better code reviews:

1. Let the computer do the boring parts
2. Settle arguments with a style guide
3. Start reviewing immediately
4. Start high level and work your way down
5. Be generous with code examples
6. Never say "you"
7. Frame feedback as requests, not commands
8. Tie notes to principles, not opinions

https://mtlynch.io/human-code-reviews-1/

# Let the computer do the boring parts

Part of the purpose of the tools listed in the definition of done is to eliminate some of the work necessary during

the code reviews.

Having a code linter basically ensures that no one needs to be rejected for having improperly formatted code or unused variables.

Having tools that builds our code and runs our tests further ensures we don’t have to do this for our teammates. Continuous integration tools do this automatically, but require servers, so we have to just settle for clicking the build and test buttons.

However this is only easy and useful as long as we have a comprehensive set of unit tests that can be quickly run with only a few actions.

If your team is having to manually do all of the testing (entering inputs and the like), you need to re-evaluate your testing strategy.

Running a large series of manual tests for each pull request is unmanageable and will lead to lax testing and thereby large parts of your system to go untested. This is what makes Test Driven Development so important.

Ideally our reviewer is walking into a code review and only having to think about the things that only a human could do, make sure there are few hidden bugs, and making sure that the code meets the team's task expectations.

# Settle arguments with a style guide

The code review should not be complicated by team conflicts over how the code should be styled. There does not need to be constant back and forth over tabs versus spaces, function and variable naming, or the placement of curly braces. These kinds of things should be established in a team style guide near when development begins and this document has the final say in all of these arguments, period.

Each member should receive code that matches this style guide, and this style guide should be enforced as much as possible through the linter settings rather than requiring the developers to constantly correct themselves or each other.

## Where do we get a style guide?

There are two main ways of getting a style guide:

- Making your own
- Adopting an industry standard one

### Making your own

Essentially this involves writing down the decisions that your team comes to each time there is an argument over code style. This creates an additional living document.

We recommend this process for more experienced and highly opinionated teams.

### Going with an industry standard:

There are a variety of places you can find these, but we recommend the following source:

https://google.github.io/styleguide/

This repository holds style guides for most of the most popular languages in use today,

It is missing C#, for which we recommend the following source:

https://github.com/raywenderlich/c-sharp-style-guide

Using a style guide from someone else takes the responsibility of making a lot of these style guide decisions away from the team. Arguments are essentially resolved through “the style guide authors know more than us, so they are probably right”, and following the standards in the guide.

There is of course an additional option of starting with another style guide and making changes over time as your team sees fit. This is one reason why all of these guides are on Git, so that they can be forked from.

## Start reviewing immediately

The idea of protecting master and develop and requiring pull requests only works as long as new changes are quickly pulled into the system.

When large amounts of changes sit waiting to be accepted, people are starting work on an old version of the system that doesn’t properly represent the teams understanding of the project.

Testing and accepting or rejecting new pull requests should be one of if not the team’s highest priority when working throughout the sprint.

We want our team members to be able to complete a task, have it either accepted or be able to incorporate feedback, and then move on. Having team members with a bunch of unapproved work can be harmful to morale and productivity.

Furthermore, the need for fast turnaround times on pull requests incentivizes team members submitting frequent small pull requests that do not need much work in order to accept.

## Start high level and work your way down

Have you ever compiled a program and gotten so many errors that you felt overwhelmed about where to start fixing them?

We want to avoid that feeling with code reviews.

It is often not necessary to break down the code review comments into many single line changes.

You can often get a lot more work done with a high level criticism than a large series of low level comments.

This might involve pointing out a reoccurring problem in the code, or the overall behavior in a function.

If there are a couple of small, isolated problems with lines of code, these should be mentioned, but only after the high level problems have been identified.

This will result in a much more manageable change response process.

## Be generous with code examples

Providing a few code examples in your review comment can go a long way in improving the code review process.

Code examples can:

- Clarify meaning of a comment.
- Show a new way of doing something.
- Indicate you are taking the time to try and help rather than just criticize.

This is not for big picture things, and typically is used to illustrate an important concept or reoccurring problem.

You might show them how to break up a big function they have, how to use a more advanced feature of the language, or fill a hole in their unit tests.

In no way are we suggesting that you fix every problem in their code and provide them with a solution. This will only signal that you think they are an inadequate developer that can not fix their own problem. Code examples should be limited to a couple times (1-3) in a review.

## Never say “you”

It is important when doing code reviews to not make the person being reviewed feel that they are being

commanded to do things or that accusatory statements about them are being made.

It is therefore best to have corrections have the subject of the sentence be the code, rather than its author, or to have no sentence subject altogether.

Compare the following two requests

```
You misspelled “connection”
```

To

```
Conection->connection
```

Or

```
Your variable name of “time” is vague, change it to “seconds_remaining”
```

To

```
Suggest renaming “time” to “seconds_remaining”
```

## Frame feedback as requests, not commands

As a follow up on the previous practice, it is often best to phrase your corrections in terms of questions or requests instead of commands.

It is not often in real life that we would command a peer to do something, and yet during code reviews this politeness can fall away under the guise of “helping”.

Having your code review comments from your peer can come off as having the wrong power dynamic, and can be frustrating to the person being reviewed.

Consider phrasing the command:

```
Move Foo to a separate file
```

To

```
Can Foo be moved to a separate file?
```

This also makes it easier for the person being reviewed to defend their decisions by answering these questions

rather than saying no to a command if they have a good reason for the design decisions they made.

## Tie notes to principles, not opinions

It is important to remember that the person you are reviewing submitted their code because they thought it was good and finished, and you are giving them more work that they may not see as necessary right away.

This is why you should try to back up the reasons for your changes with principles when possible, rather than simply seeming to assert an opinion.

This might mean referring to the team style guide, the SOLID principles, Clean Code (later in the semester), or wherever your team has established their coding principles.

This can help relay to your teammate that your edits are for a good reason and not based solely on your opinion.

For more information on conducting pleasant and productive code reviews:

https://mtlynch.io/human-code-reviews-1/

## “Premature Optimization is the root of all evil”

Imagine that you are reviewing some code that belongs to a new feature, and you come across some code that is completely unreadable and has the comment “For performance optimization”.

To a certain degree, it might seem like you should accept this and move on even though you don’t understand it.

However, each time this happens, there is now a new dependency in your system. This code cannot be changed by anyone except the code’s author, because only the author understands it.

Over time, this would produce more and more dependencies on unreviewed code until the system reaches the point where bugs start occurring and only a few people on the project are able to fix them.

It is far more conducive to a healthy development lifecycle to write highly readable and maintainable code than it is to write high performance code.

For the most part, projects do not need high performance code, other than a very small amount of critical functions. In the scope of this class, probably no high performance code is necessary.

Code that is unreadable should be rejected just as if it is incorrect.

## Reviewing the Review

Throughout this process, it is important to consider that we are being reviewed by our peers, which means that they are likely no more experienced than us, and it is possible that they have made wrong assumptions or mistakes. We have not submitted our code to a professional fixer, we are just looking for a second perspective.

For this reason, it is not necessarily the case that you will receive a review and immediately do all of the changes suggested.

Instead, you might do all of the changes you agree with, and provide rationales for the changes that you didn't make.

Because you are working with a peer, they don't have to accept your rationale either. They might make a counter argument to your rebuttal.

This process might continue for several rounds before your reviewer is satisfied that this code is "done".

If two people cannot come to an agreement, then additional team members need to step in as a neutral third party to resolve the conflicts.

# In Summary

In this reading we have discussed:

- Establishing a definition of when software changes are ready to be sent for approval "definition of done".
- Some principles for conducting more pleasant code reviews with less arguments.

In the next reading we will cover effectively demonstrating our software products.