



Individual Study

Introduction to Code Review

COMP23311 - Software Engineering I

University of Manchester

Department of Computer Science

2020/2021

taught by

Suzanne M. Embury

written by

Suzanne M. Embury and Markel Vigo

Contents

1	Why Review Code?	1
2	Types of Code Review	2
2.1	Buddy Review	2
2.2	Team-Based Review	3
2.3	Formal Review	3
3	Good Practice for Code Reviewers	4
4	Code Review Facilities in GitLab	9
5	Code Review in COMP23311	11

1 Why Review Code?

Code review is the process by which program code written by one person (or group of people) is inspected by another person (or group of people), to find errors and infelicities. It is one of the primary tools used today to manage the quality of an organisation's code base. Code review comes in lots of different flavours (we will list the main ones later in the document) but the core idea, common to them all, is a very basic one: it is easier to spot problems in code written by someone else than in your own code.

It is easy to see why this might be the case. When we have just written some code, the idea of what the code should be doing is fresh in our minds. It is hard to see the discrepancies between our mental model of what the code should be doing and what the code we have written is actually doing. But when we read code written by someone else, we do not have so many preconditions and assumptions that get in the way of understanding what has been written, as opposed to what was intended.

This applies to general code quality issues as well as bugs. A code reviewer can spot when we have failed to follow the naming and layout standards in use within the code, or when we have used comments in a way that does not follow the conventions of the rest of the code base. It is hard to keep track of all these things, especially when new to a team, as well as making the code do what it should. A code review can point out problems before they leave our feature branch, so that only good quality code reaches our development branch.

We saw earlier in this course unit that there are major advantages to finding bugs earlier after they are introduced rather than later, with costs rising especially dramatically when bugs make it through to code that is used by the customer¹. While automated testing (unit testing, etc.) can go a long way towards finding defects before they reach the customer, it is not by itself a complete solution to the problem. Testing can only find defects that we have thought to check for. And testing cannot help us weed out poor quality code that makes future bugs harder to find and fix.

It turns out that code review is an excellent complement to testing. Studies have shown that code review can find up to 60% of defects [3], with unit testing only finding 25%. The two techniques work well in partnership: automated testing is relatively cheap to run, and can be run repeatedly without needing (much) human intervention. Code review is more expensive, and requires human effort for each time it is performed, but can find a wider range of defects. A good workflow therefore is to make sure that the bugs that are detected by code review are converted into test cases, so that they can be detected cheaply in future versions of the code, and the valuable code review effort can be put towards finding new defects not currently covered by the code base. Of course, for maximum efficiency, this requires that code review is only ever performed on code that already passes the test suite.

¹Refer back to Boehm's Cost of Quality model, presented in lectures in week 1 for details.

Code review has other advantages as well, in helping to homogenise and improve coding styles across teams, and to spread knowledge of the code base more evenly throughout the team. If all code is reviewed by at least one person, then the days when parts of the code are untouchable by anyone except the lone expert who created them are gone and the team's truck factor is increased². And it is human nature to code more carefully and correctly when we know that one of our colleagues will be looking over any code that we push to the team repository.

It is important to be aware of the costs of code review, as well, however. Code review takes time. Typical code review rates are between 100 and 200 lines of code per hour, for experienced professionals [2, 1]. New team members will be slower than this. In a typical team, one can expect to spend between 1 and 5 hours per week reviewing code (more, perhaps, as a release deadline approaches). Time spent reviewing is time not spent coding, and it can sometimes be hard to justify spending the time when deadlines are looming. But, it is exactly when the team is under pressure that code reviews are most needed. Any errors that slip through at this stage will only come back in a more expensive form, when the customer feels their bite.

2 Types of Code Review

There have been many proposals for different ways of doing code reviews, ranging from the very simple and informal to heavyweight and expensive monitoring programmes. Here, we mention a few of the key types, to give you a feel for the different ways in which code review has been implemented in practice.

2.1 Buddy Review

Starting with small and simple, there are several informal kinds of code review. These normally come under the heading of "buddy review". As the name suggests, this kind of review is done informally, between coders with more or less equal status within the team, on an as-needed basis. This could be as simple as asking someone else in your team to look over a particularly tricky piece of code before you commit it ("over the shoulder" review). Or, in some teams, developers have an assigned "buddy" who they talk code through with, when it is ready to be pushed.

In teams that use the agile practice of pair programming, code review will be happening

²The *truck/bus factor* of a team is the number of its members who would need to be run over by a truck/bus for the team to be unable to fulfil its function in some significant way. A team where only one person understands and can safely change the code that interfaces with the database, for example, has a truck factor of 1, and the team should consider itself to be at risk. What do you think the truck factor of your COMP23311 team is? If it is low, what steps might you take to increase the truck factor for later exercises?

all the time, as the pair takes it in turns to act in the “driver” and “navigator” roles. The navigator looks over the code written by the driver, performing what is essentially a code review function, on a continuous basis with a very short feedback time.

2.2 Team-Based Review

Many teams have more formal structures for reviewing code, to ensure that quality is managed evenly across the team, regardless of individual team members’ preferences for or against code review.

The rise of source code repositories like GitHub and GitLab, and the coding workflows that have developed around them give a perfect framework into which to insert code review into the normal day-to-day work of the team. For example, many teams will require the code in a feature branch to be reviewed by another team member before it can be merged with the development branch. This can even be enforced by the use of tools such as Gerrit, which allows code to be held in a “staging area” for code review, and which blocks the integration of code into the main development branch until an authorised user has agreed that it meets the team’s quality standards.

However, other forms of team-based code review are possible, and have been in use for many years. One common team-based review technique is the “walkthrough”. This involves a meeting of affected team members (typically more than 2) in which one team member gives a verbal presentation of some artefact, taking the rest of the team through the way it works and what it is intended to achieve. Walkthroughs can take place early in a development, to sanity check a planned design, for example, or later in a release cycle, to check that correctness of the implementation of some key algorithm or section of the code, by working through it together line-by-line.

2.3 Formal Review

The most formal type of code review involves the work of a team being inspected by an external team. This kind of review is usually only performed in large organisations with very tightly-defined processes for managing software quality across the organisation. They sometimes go by the name of “inspections”, “formal technical reviews” and “formal management reviews”. A formal technical review will involve the work of the team being assessed by an external team of technical experts. Formal management review, on the other hand, is an assessment of the quality of a team’s processes, and will normally involve the work of the team being inspected by more senior (possibly non-technical) staff within the organisation.

These reviews are often linked to the long term future of a team or project. Continued

funding may be dependent on successfully passing through a series of formal review processes.

Unsurprisingly, the most formal type of code review is also the most expensive, with extensive documentation and presentations having to be prepared in advance, as well as hotel and meeting rooms needing to be booked for the participants, sometimes for 2 or 3 days.

Now that teams are taking on responsibility for their own code reviews, using more informal techniques, these large scale formal reviews are less common. (There were always doubts about their cost-effectiveness, and the effects on staff of these stressful reviews was often seen as being counter-productive in the long term.) While funding reviews and presentations to senior management or customers are still a normal part of life as a software engineer, code quality is typically managed through more informal, team-based routes in modern organisations.

3 Good Practice for Code Reviewers

If code review is to be an effective means of discovering and removing defects from software, then it is important that everyone involved (reviewers and reviewees) see the process as a positive one, rather than as a chore to be endured. It is important that criticisms are worded constructively, and that questions of blame are regarded as being of less importance than finding and fixing the errors. Code reviewers should be seen as allies against a common enemy (bugs that reach the customer), but at the same time few people enjoy having their flaws publicly pointed out.

It is therefore important that code reviewers adopt a neutral tone, and balance negative points out with positive comments about aspects of the code that are done well. This is particularly important when performing code review in a new team. Later, when trust and mutual respect have been built up, code reviews can be less carefully worded. But in the early days, it is important to think about how comments can be worded, to ensure that they will be received in a positive way. But regardless of the diplomacy with which the reviewer carries out their work, for the reviewee, code reviews are a powerful mechanism for learning to be challenged, to cope with constructive criticism and to defend your own ideas where you feel the reviewer has overlooked something of importance.

The learning effect of code reviews can be helped by making all reviews accessible to the whole team (as they are in a GitLab repository). That way, the code reviewer is held accountable for their reviews, just as the coder is being held accountable for their code.

Another technique that can help is the practice of “promiscuous reviewing”, in which all team members review the work of all other team members, over time, rather than sticking with rigid and unchanging buddy review pairings. This avoids the risk of “revenge reviews”, and helps to foster an “all-in-it-together” team mentality. The person reviewing your code

will be conscious that, not too far in the future, the roles will be reversed. This can help to keep the review process constructive and considerate.

When performing a code review, the following aspects of the code can be commented on:

- Possible defects: for example, pointing out an off-by-one error in a loop, or improper handling of negative numbers.
- Possible flaws in testing: for example, a missing case that is not covered by tests, or pointing out when two tests seem to be testing the same thing (i.e., redundant tests).
- Design issues: for example, pointing out that a class should be split into two classes, or recommending that a method be moved to a different class.
- Naming issues: for example, noting where a class name is drifting out of line with the changing function of the class, or where legacy variable names need to be updated to follow the team's current practice.
- Coding style issues: for example, pointing out inconsistent code layout, or where naming of constants is inconsistent with constants elsewhere in the code.
- Presence of code or test smells: for example, methods that are too long, or test methods containing two separate tests bundled together.
- Use of Git: for example, noting when a commit message is insufficiently clear or when commits should be squashed together to make a more meaningful representation of the change being made.
- Comments on the Build: for example, noting which platforms a bug-fix is suitable for, or pointing out where a change will require changes in the build process or other configuration elements.
- Effect on Team Metrics: for example, noting where code coverage is negatively or positively affected by a change, or any additional technical debt that the change incurs.
- Examples of good practice: for example, where an elegant solution has been found to a problem, or where code is clear and readable without requiring any comments.

We can see from this list that code review can address more than just defects and errors, covering also code quality, enforcement of conventions, performance issues and code readability aspects. Code reviewers should try not to comment on matters of preference. For example, if the team has not agreed to on a convention for white space in code, reviewers should not criticise team members for using spaces when their own preference is for using tabs, provided the look of the code is the same for both options. The line between preference and best practice is not always clearly defined, so some degree of trial and error is in order. If you find you are continually pointing out that someone is placing their

curly brackets differently from the rest of the team, and that person continually ignores your advice, it is probably time to move on to find bigger fish to fry.

To give an indication of the kinds of conversation that can productively take place using code review, below are some examples of commit comments from the NodeJS project, on GitHub³.

The screenshot shows a GitHub pull request interface for the Node.js repository. At the top, the repository name 'nodejs / node' is displayed along with statistics: 1,933 Watchers, 28,570 Stars, and 4,753 Forks. Below this, navigation tabs include Code, Issues (593), Pull requests (280), Projects (3), Wiki, Pulse, and Graphs. The pull request title is 'http: remove stale timeout listeners #9440', and it is opened by 'karlbohmark' who wants to merge 1 commit into 'nodejs:master' from 'karlbohmark:memory-leak-9268'. The interface shows 2 files changed with a net change of +63 lines and -1 line. A diff view for 'lib/_http_client.js' is shown, with line numbers 562 to 567. The diff indicates a change in the 'tickOnSocket' function, removing a 'socket.once' listener and adding a 'const emitRequestTimeout' and a 'socket.once' listener. Below the diff, there are two comments. The first comment is from 'evanlucas' (Node.js Foundation member) asking a question about the response event. The second comment is from 'karlbohmark' (edited) responding that the socket is destroyed, which solves the cleanup issue.

nodejs / node Watch 1,933 Star 28,570 Fork 4,753

<> Code Issues 593 Pull requests 280 Projects 3 Wiki Pulse Graphs

http: remove stale timeout listeners #9440

Open karlbohmark wants to merge 1 commit into nodejs:master from karlbohmark:memory-leak-9268

Conversation 2 Commits 1 Files changed 2

Changes from all commits 2 files +63 -1

Unified Split Review changes

8 lib/_http_client.js Show comments View

```
@@ -562,7 +562,13 @@ function tickOnSocket(req, socket) {
562 562 socket.on('close', socketCloseListener);
563 563
564 564 if (req.timeout) {
565 - socket.once('timeout', () => req.emit('timeout'));
565 + const emitRequestTimeout = () => req.emit('timeout');
566 + socket.once('timeout', emitRequestTimeout);
567 + req.on('response', (r) => {
```

evanlucas an hour ago Node.js Foundation member
hm, I wonder if the response event is emitted in all cases if the request is aborted...will have to dig into that

karlbohmark an hour ago • edited
No, I don't think it is, but I think the socket is destroyed, which solves the cleanup issue in that case.

³github.com/nodejs

63		- for (let i = 0; i < kNumberOfHeapSpaces; i++) {
	63	+ for (var i = 0; i < kNumberOfHeapSpaces; i++) {
64	64	const propertyOffset = i * kHeapSpaceStatisticsPropertiesCount;
65	65	heapSpaceStatistics[i] = {
66	66	space_name: kHeapSpaces[i],

4 comments on commit 2e568d9



xiangdewei commented on 2e568d9 29 days ago



can you tell me why you change from let to var? I think there are no difference in this case



addaleax commented on 2e568d9 29 days ago

Node.js Foundation member



@xiangdewei You're right, there is no difference in behaviour here. But – I know it sounds weird – using `let` is a bit slower right now, and given how performance-aware Node.js core tries to be, it makes sense to avoid it at least in loops.



TheAlphaNerd commented on 2e568d9 29 days ago

Node.js Foundation member



as @addaleax mentioned this is about how the V8 engine optimizes loops. There is a slight deopt when using `let` in a for loop rather than `var`. This is more noticeable in hot code. The update to `punycode` made a difference of 5%, which is non-trivial



xiangdewei commented on 2e568d9 29 days ago



@addaleax @TheAlphaNerd now I know it's about the performance, thank you

120

```
- let buffer = Buffer.from('');
```

126

```
+ let buffer = new Buffer(0);
```

**mscdex** on Sep 23 • edited contributor

Why this change? We should be using the new `Buffer` methods since the constructor form is deprecated.

/cc @eugeneo

**eugeneo** on Sep 23 contributor

I switched there for consistency with the rest of this test case. Please take a look - [#8748](#)

**jasnell** on Sep 23 Node.js Foundation member

Still, this likely should not have been changed. There are plenty of other places here where `Buffer.from` is used. If anything, this should have been changed to `Buffer.alloc(0)`.

**jasnell** on Sep 23 Node.js Foundation member

That said, woohoo! tests are passing on macos sierra now.


**eugeneo** on Sep 23 contributor

@jasnell I updated [#8748](#) to use `Buffer.alloc`


**jasnell** on Sep 23 Node.js Foundation member

Thank you!


389		-Provides an object enumerating Zlib-related <code>[constants][]</code> .
	389	+Provides an object enumerating Zlib-related constants.
390	390	
391	391	Reset the compressor/decompressor to factory defaults. Only applicable to


cjihrig on Aug 10
 Node.js Foundation member


@ChALkeR I'm in the process of pulling this back to v6.x. It looks like `zlib.constants` was added in the middle of the definition of `zlib.reset()` . I'm fixing it on v6.x, but could you PR master?


jasnell on Aug 10
 Node.js Foundation member


eh? what the heck... this shouldn't have happened :-(...


ChALkeR on Aug 10 • edited
 Node.js Foundation member


@jasnell What shouldn't have happened? I might be missing something.


ChALkeR on Aug 10
 Node.js Foundation member


This specific change removed a broken link. Perhaps it was meant to lead elsewhere?


cjihrig on Aug 10
 Node.js Foundation member


View the fully rendered file and jump down to `zlib.reset()` . Note that `zlib.constants` is now where the description of `zlib.reset()` should be, and the description follows.


ChALkeR on Aug 10 • edited
 Node.js Foundation member

@cjihrig Ah, I missed that. But it wasn't introduced by this commit, right?


ChALkeR on Aug 10
 Node.js Foundation member

@cjihrig I will file a PR with a fix for master, thanks!



jasnell on Aug 10
 Node.js Foundation member

Yeah, I don't think it was introduced by this commit. I believe it was an errant rebase/merge at some point. I haven't looked into it further yet tho

4 Code Review Facilities in GitLab

GitLab, like GitHub, provides facilities for commenting on merge/pull requests, and on individual commits.

When examining a merge request, GitLab allows comments to be placed on the code changes included in the proposed merge. The merge request is assigned to the person who will review the code. After examining the changes, the reviewer can decide to accept the merge request, or to request further changes. Once the changes are made, the developer can request a second round of review. In this way, the code reviewer acts as a gate keeper, preventing poor quality code from reaching the development branch.



This project Search

Project Activity Repository Graphs Issues 0 Merge Requests 0 Wiki

New Merge Request

From `br_merge_req` into `master` [Change branches](#)





Title

my merge request

Start the title with `WIP:` to prevent a **Work In Progress** merge request from being merged before it's ready.

Description

Write Preview

B I    

Write a comment or drag your files here...

Styling with [Markdown](#) and slash commands are supported

Attach a file

Assignee

Select assignee

[Assign to me](#)

Milestone

Select milestone

[Create new milestone](#)

Labels

No labels yet.

[Create new label](#)

Source branch

br_merge_req

Target branch

master

[Change branches](#)

☐ Remove source branch when merge request is accepted.

Submit merge request

Cancel

Or, comments can be added directly to individual commits. Comments on the commit as a whole can be added through the dialogue box at the end of the commit. Or, comments can be added at specific lines. To do this, hover your mouse over the line in the commit where you wish to add the comment. A small speech bubble will appear to the left hand side of the line. Click on this to bring up a comment box, similar to those shown in the NodeJS examples given in the previous section.





src/main/java/uk/ac/man/cs/eventlite/dao/VenueRepository.java
View file @57cea61

```

... @@ -7,4 +7,5 @@ import uk.ac.man.cs.eventlite.entities.Venue;
7 7 public interface VenueRepository extends CrudRepository<Venue, Long> {
8 8     public Iterable<Venue> findAllByNameAsc();
9 9     public Iterable<Venue> findByName(String name);
10 + public Iterable<Venue> findByNameContainingIgnoreCase(String name);

```

Write Preview

B I    

Write a comment or drag your files here...

Styling with [Markdown](#) and slash commands are supported

Attach a file

Comment

Cancel

5 Code Review in COMP23311

For exercises 2 and 3 of the COMP23311 team coursework, you are asked to practice an informal team-based code review system. You should ensure that the code for every feature you add to the Stendhal code base, or change, is reviewed by a separate team member. This includes both test code and production code changes.

You should use GitLab's comment facility to review your team's work. If a feature is short enough to review in one go, and is produced in plenty of time, you can add your code review to the merge request for the feature branch.

If you are attempting a larger feature, or a smaller feature is taking a long time to implement, you may want to give interim code reviews, on partial versions of the feature or its test code. For this, you can use GitLab's comment facility on individual commits. This means you can review any commits that have been pushed to your team's repository, even if the branch they are on has not been merged into the main development branch yet. In this case, you should check with the coder whether they are happy to have the commits reviewed, as it can be quite annoying to be told about problems you are aware of, and are in the middle of fixing.

It is up to you how you organise your team to complete the reviews provided that:

- All code changes made for your new features are reviewed.
- All contributing team members carry out at least one code review.

Code review is very widely used in the software development industry today. You can expect to be subjected to code reviews as soon as you start to write code that becomes visible in your team's repository. Many companies also expect all software engineers, no matter how junior, to review the code of others. As well as helping to keep the quality of your team's code base high, code review is also a great way to learn the conventions and standards used by your team. Look at code reviews provided by your colleagues to learn what is considered good and bad practice, and to get a feel for the degree and style of review comments your team members expect.

In COMP23311, we ask you to carry out some basic buddy reviewing. Through the coursework, you can practice commenting on others code, and responding to comments on your own. This will be useful interviewing material, and will also prepare you to join a software team using modern software quality management techniques.

You can also ask your industry mentor about code reviewing practices used in their organisation, and how they help the work of the team.

Good luck!

References

- [1] David B. Bisant and James R. Lyle. A two-person inspection method to improve programming productivity. *IEEE Transactions on Software Engineering*, 15:1294–1304, 1989.
- [2] Chris F.Kemerer and Mark C. Paulk. The impact of design and code reviews on software quality: An empirical study based on psp data. *IEEE Transactions on Software Engineering*, pages 534–550, 2009.
- [3] Steve McConnell. *Code Complete*. Microsoft Press, 2004.