

WHAT TO LOOK FOR — IN A — CODE REVIEW

BY TRISHA GEE



What to Look for in a Code Review

Effective tips for reviewing code

Trisha Gee

This book is for sale at <http://leanpub.com/whattolookforinacodereview>

This version was published on 2016-01-25



JetBrains - World's Leading Vendor of Professional Development Tools

We help developers work faster by automating common, repetitive tasks to enable them to stay focused on code design and the big picture. We provide tools to explore and familiarize with code bases faster. Our products make it easy for you to take care of quality during all stages of development and spend less time on maintenance tasks.

Our JetBrains Technical Series of books focuses on general computer science topics as well as tooling.

© 2015 - 2016 Copyright 2000-2016, JetBrains

Contents

JetBrains Technical Series	1
About this book	1
About the author	2
Introduction	3
What do you look for when reviewing someone else's code?	3
What should you look for	4
Tests	6
Ask yourself these questions	6
Reviewers can write tests too	10
Summary	10
Performance	11
Performance Requirements	11
Calls outside of the service/application are expensive	12
Using resources efficiently and effectively	13
Warning signs a reviewer can easily spot	15
Correctness	17
Code-level optimisations	18
Summary	20
Data Structures	21
Lists	21
Maps	23
Sets	26
Stacks	27
Queues	27
Why select the right data structure?	28
Summary	29
SOLID Principles	30
What is SOLID?	30
Single Responsibility Principle (SRP)	30

CONTENTS

Open-Closed Principle (OCP)	31
Liskov Substitution Principle (LSP)	32
Interface Segregation Principle (ISP)	33
Mino Dependency Inversion Principle (DIP)	34
Summary	35
Security	36
Automation is your friend	36
Sometimes “It Depends”	37
Understand your Dependencies	38
Summary	40
Upsource Quick Wins	42
Navigation	42
Inspections	43
Exception Handling Issues	43
Probable Bugs	44
Code can be simplified	45
Unused Code	46
Summary	48

JetBrains Technical Series

This book is a part of the **JetBrains Technical Series** of books covering a wide range of topics related to Software Development. For more information on JetBrains, please visit <https://www.jetbrains.com>¹

About this book

This book is mostly a compilation of blog posts written over a period of time, around code reviews, all of which are available on the [Upsource blog](http://blog.jetbrains.com/upsources/category/practices/)².

The contents have been reformatted and made available in booklet format for offline reading.

Source Code

The source code for the examples in this book can be found on the corresponding [JetBrains GitHub repository](https://github.com/jetbrains/jetbrains-books-examples/whattolookforinacodereview)³

¹JetBrains' website

²<http://blog.jetbrains.com/upsources/category/practices/>

³<https://github.com/jetbrains/jetbrains-books-examples/whattolookforinacodereview>

About the author



Trisha Gee, Developer Advocate, JetBrains

Trisha has developed Java applications for a range of industries, including finance, manufacturing, software and non-profit, for companies of all sizes. She has expertise in Java high-performance systems, is passionate about enabling developer productivity, and dabbles with Open Source development. Trisha blogs regularly on subjects that she thinks developers and other humans should care about. She's a leader of the Sevilla Java User Group, a key member of the London Java Community, a MongoDB Master, and a Java Champion. She believes we shouldn't all have to make the same mistakes again and again, and as a Developer Advocate for JetBrains she can share all the cool stuff she's discovered so far.

Introduction

Let's talk about code reviews. If you take only a few seconds to search for information about code reviews, you'll see a lot of articles about why code reviews are a Good Thing (for example, this post by [Jeff Atwood](#)⁴).

You also see a lot of documentation on how to use Code Review tools like our very own Upsource.

What you don't see so much of is a guide to things to look for when you're reviewing someone else's code.

Probably the reason there's no definitive article on what to be looking for is: there are a lot of different things to consider. And, like any other set of requirements (functional or non-functional), individual organizations will have different priorities for each aspect.

Since this is a big topic to cover, the aim of this chapter is to outline just some of the things a reviewer could be looking out for when performing a code review. Deciding on the priority of each aspect and checking them consistently is a sufficiently complex subject to be a chapter in its own right.

What do you look for when reviewing someone else's code?

Whether you're reviewing code via a tool like Upsource or during a colleague's walkthrough of their code, whatever the situation, some things are easier to comment on than others. Some examples:

- Formatting: Where are the spaces and line breaks? Are they using tabs or spaces? How are the curly braces laid out?
- Style: Are the variables/parameters declared as final? Are method variables defined close to the code where they're used or at the start of the method?
- Naming: Do the field/constant/variable/param/class names conform to standards? Are the names overly short?
- Test coverage: Is there a test for this code? These are all valid things to check – you want to minimize context switching between different areas of code and reduce [cognitive load](#)⁵, so the more consistent your code looks, the better.

However, having humans looking for these is probably not the best use of time and resources in your organization, as many of these checks can be automated. There are plenty of tools that can

⁴<http://blog.codinghorror.com/code-reviews-just-do-it/>

⁵https://en.wikipedia.org/wiki/Cognitive_load

ensure that your code is consistently formatted, that standards around naming and the use of the final keyword are followed, and that common bugs caused by simple programming errors are found. For example, you can run [IntelliJ IDEA's inspections from the command line](#)⁶ so you don't have to rely on all team members having the same inspections running in their IDE.

What should you look for

What sort of things are humans really good for? What can we spot in a code review that we can't delegate to a tool?

It turns out there's a surprisingly large number of things. This is certainly not an exhaustive list, nor will we go into any one of them in great detail here. Instead, this should be the start of a conversation in your organization about which things you currently look for in a code review, and what, perhaps, you **should** be looking for.

Design

- How does the new code fit with the overall architecture?
- Does the code follow [SOLID principles](#)⁷, [Domain Driven Design](#)⁸ and/or other design paradigms the team favors?
- What [design patterns](#)⁹ are used in the new code? Are these appropriate?
- If the codebase has a mix of standards or design styles, does this new code follow the current practices? Is the code migrating in the correct direction, or does it follow the example of older code that is due to be phased out?
- Is the code in the right place? For example, if the code is related to Orders, is it in the Order Service?
- Could the new code have reused something in the existing code? Does the new code provide something we can reuse in the existing code? Does the new code introduce duplication? If so, should it be refactored to a more reusable pattern, or is this acceptable at this stage?
- Is the code over-engineered? Does it build for reusability that isn't required now? How does the team balance considerations of reusability with [YAGNI](#)¹⁰?

Readability & Maintainability

- Do the names (of fields, variables, parameters, methods and classes) actually reflect the things they represent?

⁶<https://www.jetbrains.com/idea/help/running-inspections-offline.html>

⁷[https://en.wikipedia.org/wiki/SOLID_\(object-oriented_design\)](https://en.wikipedia.org/wiki/SOLID_(object-oriented_design))

⁸https://en.wikipedia.org/wiki/Domain-driven_design

⁹https://en.wikipedia.org/wiki/Software_design_pattern

¹⁰https://en.wikipedia.org/wiki/You_aren%27t_gonna_need_it

- Can I understand what the code does by reading it?
- Can I understand what the tests do?
- Do the tests cover a good subset of cases? Do they cover happy paths and exceptional cases? Are there cases that haven't been considered?
- Are the exception error messages understandable?
- Are confusing sections of code either documented, commented, or covered by understandable tests (according to team preference)?

Functionality

- Does the code actually do what it was supposed to do? If there are automated tests to ensure correctness of the code, do the tests really test that the code meets the agreed requirements?
- Does the code look like it contains subtle bugs, like using the wrong variable for a check, or accidentally using an *and* instead of an *or*?

Have you thought about...?

- Are there potential security problems with the code?
- Are there regulatory requirements that need to be met?
- For areas that are not covered with automated performance tests, does the new code introduce avoidable performance issues, like unnecessary calls to a database or remote service?
- Does the author need to create public documentation, or change existing help files?
- Have user-facing messages been checked for correctness?
- Are there obvious errors that will stop this working in production? Is the code going to accidentally point at the test database, or is there a hardcoded stub that should be swapped out for a real service?

Tests

In the previous chapter we talked about a wide variety of things you could look for in a code review. Now we'll focus on one area: what to look for in the test code.

We're assuming that “ * Your team already writes automated tests for code. * The tests are regularly run in a Continuous Integration (CI) environment. * The code under review has been through an automated compile/test process and passed.

Ask yourself these questions

In this chapter we'll cover some of the things a reviewer could be thinking about when looking at the tests in a code review.

Are there tests for this new/amended code?

It would be rare that new code, whether a bug fix or new feature, wouldn't need a new or updated test to cover it. Even changes for “non-functional” reasons like performance can frequently be proved via a test. If there are no tests included in the code review, as a reviewer the first question you should ask is “why not?”.

Do the tests at least cover confusing or complicated sections of code?

One step beyond simply “is there a test?” is to answer the question, “is the important code actually covered by at least one test?”.

Checking test coverage is certainly something we should be automating. But we can do more than just check for specific percentages in our coverage: we can use coverage tools to ensure the correct areas of code are covered.

Consider this for example:

Open MRPH-CR-1: Changed getParameterizedClass so that if it's passed an interface without generic params that extends an interface with generic params, it will return the super-interface's generic param. Fixes #784.

ReflectionUtils.java morphia/src/main/java/org/mongodb/morphia/utils

```

300 300     }
301 301     } else {
302 302     *   final Type superclass = c.getGenericSuperclass();
303 303     +   if (superclass == null && c.isInterface()) {
304 304     +       Type[] interfaces = c.getGenericInterfaces();
305 305     +       if (interfaces.length > 0) {
306 306     +           superclass = interfaces[index];
307 307     +       }
308 308     +   }
309 309     if (superclass instanceof ParameterizedType) {
310 310     final Type[] actualTypeArguments = ((ParameterizedType) superclass).getActualTypeArguments();

```

Code Review Changes

You can check the coverage report for the new lines of code (which should be easy to identify, especially if you're using a tool like Upsource) to make sure it's adequately covered.

```

300 300     }
301 301     } else {
302 302     Type superclass = c.getGenericSuperclass();
303 303     if (superclass == null && c.isInterface()) {
304 304     Type[] interfaces = c.getGenericInterfaces();
305 305     if (interfaces.length > 0) {
306 306     superclass = interfaces[index];
307 307     }
308 308     }
309 309     if (superclass instanceof ParameterizedType) {
310 310     final Type[] actualTypeArguments = ((ParameterizedType) supercla

```

Code Review Coverage

In this example above, the reviewer may ask the author to add a test to cover the case where the if on line 303 evaluates to true, as the coverage tool has marked lines 304-306 with red to show they aren't tested.

100% test coverage is an unrealistic goal for pretty much any team, so the numbers coming out of your coverage tool might not be as valuable as insights into which specific areas are covered.

In particular, you want to check that all logic branches are covered, and that complex areas of code are covered.

```

658 658 public static Class<?> getClass(final Type type) {
659 659     if (type instanceof Class) {
660 660     return (Class) type;
661 661     } else if (type instanceof ParameterizedType) {
662 662     return getClass(((ParameterizedType) type).getRawType());
663 663     } else if (type instanceof GenericArrayType) {
664 664     final Type componentType = ((GenericArrayType) type).getGenericComponentType();
665 665     final Class<?> componentClass = getClass(componentType);
666 666     if (componentClass != null) {
667 667     return Array.newInstance(componentClass, 0).getClass();
668 668     } else {
669 669     return null;
670 670     }
671 671     } else {
672 672     return null;
673 673     }

```

Code Review Logic

Can I understand the tests?

Having tests that provide adequate coverage is one thing, but if I, as a human, can't understand the tests, they have limited use. What happens when they break? It'll be hard to know how to fix them.

Consider the following:

```
▼ IdTest.java morphia/src/test/java/org/mongodb/morphia/utils +35 Side-by-side diff View file
13 + @Test
14 + public void testId() throws Exception {
15 +     MyEntity obj = new MyEntity(getDs());
16 +     getDs().save(obj);
17 +     assertEquals(FIRST_ID, obj.getMyLongId());
18 +     obj = new MyEntity(getDs());
19 +     getDs().save(obj);
20 +     assertEquals(SECOND_ID, obj.getMyLongId());
21 + }
22 +
```

Code that is hard to test

It's a fairly simple test, but I'm not entirely sure what it's testing. Is it testing the `save` method? Or `getMyLongId`? And why does it need to do the same thing twice?

The intent behind the test might be clearer as:

```
▼ IdTest.java morphia/src/test/java/org/mongodb/morphia/utils +40 Side-by-side diff View file
11 + @Test
12 + public void shouldIncrementTheEntityIdByOneOnEverySave() throws Exception {
13 +     // when
14 +     MyEntity entity = new MyEntity(getDs());
15 +     getDs().save(entity);
16 +
17 +     // then
18 +     assertEquals(1L, entity.getMyLongId());
19 +
20 +     // when
21 +     entity = new MyEntity(getDs());
22 +     getDs().save(entity);
23 +
24 +     // then
25 +     assertEquals(2L, entity.getMyLongId());
26 + }
```

Easier code to test

The specific steps you take in order to clarify a test's purpose will depend upon your language, libraries, team and personal preferences. This example demonstrates that by choosing clearer names, inlining constants and even adding comments, an author can make a test more readable by developers other than him- or herself.

Do the tests match the requirements?

Here's an area that really requires human expertise. Whether the requirements being met by the code under review are encoded in some formal document, on a piece of card in a user story, or contained in a bug raised by a user, the code being reviewed should relate to some initial requirement.

The reviewer should locate the original requirements and see if:

- The tests, whether they're unit, end-to-end, or something else, match the requirements. For example, if the requirements are "should allow the special characters '#', '!' and '&' in the password field", there should be a test using these values in the password field. If the test uses different special characters then it's not proving the code meets the criteria.
- The tests cover all the criteria mentioned. In our example of special characters, the requirements might go on to say "...and give the user an error message if other special characters are used". Here, the reviewer should be checking there's a test for what happens when an invalid character is used.

Can I think of cases that are not covered by the existing tests?

Often our requirements aren't clearly specified. Under these circumstances, the reviewer should think of edge cases that weren't covered in the original bug/issue/story.

If our new feature is, for example, "Give the user the ability to log on to the system", the reviewer could be thinking, "What happens if the user enters null for the username?" or "What sort of error occurs if the user doesn't exist in the system?". If these tests exist in the code being reviewed, then the reviewer has increased confidence that the code itself handles these circumstances. If the tests for these exceptional cases don't exist, then the reviewer has to go through the code to see if they have been handled.

If the code exists but the tests don't, it's up to your team to decide what your policies are – do you make the author add those tests? Or are you satisfied that the code review proved the edge cases were covered?

Are there tests to document the limitations of the code?

As a reviewer, it's often possible to see limitations in the code being reviewed. These limitations are sometimes intentional – for example, a batch process that can only handle a maximum of 1000 items per batch.

One approach to documenting these intentional limitations would be to explicitly test them. In our example above, we might have a test that proves that some sort of exception is thrown if your batch size is bigger than 1000.

It's not mandatory to express these limitations in an automated test, but if an author has written a test that shows the limits of what they've implemented, having a test implies these limits are intentional (and documented) and not merely an oversight.

Are the tests in the code review the right type/level?

For example, is the author doing expensive integration tests where a unit test might suffice? Have they written performance micro-benchmarks that will not run effectively or in a consistent fashion in the CI environment?

Ideally your automated tests will run as quickly as possible, which means that expensive end-to-end tests may not be required to check all types of features. A method that performs some mathematical function, or boolean logic check, seems like a good candidate for a method-level unit test.

Are there tests for security aspects?

Security is one area that code reviews can really benefit. We'll do a whole post on security later, but on the testing topic, we can write tests for a number of common problems. For example, if we were writing the log-in code above, we might want to also write a test that shows that we cannot enter the protected area of the site (or call protected API methods) without first authenticating.

Performance Tests

In the previous chapter I talked about performance as being an area a reviewer might be examining. Automated performance tests are obviously another type of test that I could have explored in this chapter, but I will leave discussion of these types of tests for a later chapter about looking specifically at performance aspects in a code review.

Reviewers can write tests too

Different organizations have different approaches to code reviews. Sometimes it's very clear that the author is responsible for making all the code changes required; sometimes it's more collaborative with the reviewer committing suggestions to the code themselves.

Whichever approach you take, as a reviewer you may find that writing some additional tests to poke at the code in the review can be very valuable for understanding that code, in the same way that firing up the UI and playing with a new feature is valuable. Some methods and code review tools make it easier to experiment with the code than others. It's in the team's interest to make it as easy as possible to view and play with the code in a code review.

It may be valuable to submit the additional tests as part of the review, but equally it may not be necessary, for example if experimentation has given me, the reviewer, satisfactory answers to my questions.

Summary

There are many advantages to performing a code review, no matter how you approach the process in your organization. It's possible to use code reviews to find potential problems with the code before it is integrated into the main code base, while it's still inexpensive to fix and the context is still in the developer's head.

As a code reviewer, you should be checking that the original developer has put some thought into the ways his or her code could be used, under which conditions it might break, and dealt with edge cases, possibly "documenting" the expected behavior (both under normal use and exceptional circumstances) with automated tests.

If the reviewer looks for the existence of tests and checks the correctness of the tests, as a team you can have pretty high confidence that the code works. Moreover, if these tests are run regularly in a CI environment, you can see that the code continues to work – they provide automated regression checking. If code reviewers place a high value on having good quality tests for the code they are reviewing, the value of this code review continues long after the reviewer clicks the "Accept" button.



Ability to Accept or Reject Changes

Performance

In this chapter we're going to cover what you can look for in terms of the performance of the code under review.

As with all architecture/design areas, the non-functional requirements for the performance of a system should have been set upfront. Whether you're working on a low-latency trading system which has to respond in nanoseconds, you're creating an online shopping site which needs to be responsive to the user, or you're writing a phone app to manage a "To Do" list, you should have some idea about what's considered "too slow."

Let's cover some of the things that affect performance that a reviewer can look for during a code review.

Performance Requirements

Before deciding on whether we need to undertake code reviews based on performance, we should ask ourselves a few questions.

Does this piece of functionality have hard performance requirements?

Does the piece of code under review fall under an area that had a previously published SLA? Or do the requirements state the performance characteristics required?

If the original requirements were a bug along the lines of "the login screen is too slow to load," the original developer should have clarified what would be a suitable loading time – otherwise how can the reviewer or the author be confident that the speed has been sufficiently improved?

If so, is there a test that proves it meets those?

Any performance critical system should have automated performance tests which ensure that published SLAs ("all order requests serviced in less than 10 ms") are met. Without these, you're relying on your users to inform you of failures to meet your SLAs. Not only is this a poor user experience, but could lead to avoidable fines and fees. The last post in this series covered code-reviewing tests in detail.

Has the fix/new functionality negatively impacted the results of any existing performance tests?

If you are regularly running performance tests (or if you have a suite that can be run on demand), check that new code in performance-critical areas hasn't introduced a decrease in system performance. This might be an automated process, but since running performance tests as part of a CI environment is much less common than running unit tests, it is worth specifically mentioning it as a possible step during review.

What if there are no hard performance requirements for this code review?

There's limited value spending hours agonising over optimisations that will save you a few CPU cycles. But there are things a reviewer can check for in order to ensure that the code doesn't suffer from common performance pitfalls that are completely avoidable. Check out the rest of the list to see if there are easy wins to prevent future performance problems.

Calls outside of the service/application are expensive

Any use of systems outside your own application that require a network hop are generally going to cost you *much* more than a poorly optimised `equals()` method. Consider:

Calls to the database

The worst offenders might be hiding behind abstractions like ORMs. But in a code review you should be able to catch common causes of performance problems, like individual calls to a database inside a loop – for example, loading a list of IDs, then querying the database for each individual item that corresponds to that ID.



```
11 11 private final Database database = new Database();
12 12
13 13 public List<Customer> getAllCustomers() {
14 14     List<Integer> customerIds = getAllCustomerIds();
15 15     ArrayList<Customer> customers = new ArrayList<>();
16 16     try (Connection connection = database.getConnection();
17 17         Statement statement = connection.createStatement()) {
18 18         for (Integer customerId : customerIds) {
19 19             try (ResultSet rs = statement.executeQuery("SELECT * FROM Customers WHERE id = " + customerId)) {
20 20                 rs.next();
21 21                 customers.add(new Customer(
22 22                     customerId,
23 23                     rs.getString("first"),
24 24                     rs.getString("last")
25 25                 ));
26 26             }
27 27         } catch (SQLException e) {
28 28             doDatabaseErrorHandling(e);
29 29         }
30 30     }
31 31     return customers;
32 32 }

13 33
14 34 public List<Integer> getAllCustomerIds() {
```

Calls to a database

For example, line 19 above might look fairly innocent, but it's inside a **for** loop – you have no idea how many calls to the database this code might result in.

Unnecessary network calls

Like databases, remote services can sometimes be over-used, with multiple remote calls being made where a single one might suffice, or where batching or caching might prevent expensive network calls. Again, like databases, sometimes an abstraction can hide that a method call is actually calling a remote API.

Mobile / wearable apps calling the back end too much

This is basically the same as “unnecessary network calls”, but with the added problem that on mobile devices, not only will unnecessary calls to the back-end cost you performance, it will also cost you battery life.

Using resources efficiently and effectively

Following on from how we use our network resources, a reviewer can look at the use of other resources to identify possible performance problems.

Does the code use locks to access shared resources? Could this result in poor performance or deadlocks?

[Locks are a performance killer¹¹](#) and very hard to reason about in a multi-threaded environment. Consider patterns like; having only a single thread that writes/changes values while all other threads are free to read; or using [lock free algorithms¹²](#).

Is there something in the code which could lead to a memory leak?

In Java, some common causes can be: mutable static fields, using `ThreadLocal` and using a [ClassLoader¹³](#).

Is there a possibility the memory footprint of the application could grow infinitely?

This is not the same as a memory leak – a memory leak is where unused objects cannot be collected by the garbage collector. But any language, even non-garbage-collected ones, can create

¹¹<http://mechanical-sympathy.blogspot.com.es/2013/08/lock-based-vs-lock-free-concurrent.html>

¹²https://en.wikipedia.org/wiki/Non-blocking_algorithm

¹³<http://docs.oracle.com/javase/8/docs/api/java/lang/ClassLoader.html>

data structures that grow indefinitely. If, as a reviewer, you see new values constantly being added to a list or map, question if and when the list or map is discarded or trimmed.



```
7 7 private final Map<String, TwitterUser> allTwitterUsers = new HashMap<>();
8 8
9 + public void onMessage(String twitterHandle, String tweet) {
10 +     TwitterUser twitterUser = allTwitterUsers.computeIfAbsent(twitterHandle, TwitterUser::new);
11 +     twitterUser.addMessage(tweet);
12 + }
13 +
9 14 public TwitterUser getTwitterUser(String twitterHandle) {
10 15     return allTwitterUsers.get(twitterHandle);
11 16 }
```

Infinite Memory

In the code review above, we can see all messages from Twitter being added to a map. If we examine the class more fully, we see that the `allTwitterUsers` map is never trimmed, nor is the list of tweets in a `TwitterUser`. Depending upon how many users we're monitoring and how often we add tweets, this map could get very big, very fast.

Does the code close connections/streams?

It's easy to forget to close connections or file/network streams. When you're reviewing someone else's code, whichever language it happens to be in, if a file, network or database connection is in use, make sure it is correctly closed.



```
47 47 }
48 48
49 + public Customer getCustomerById(final int customerId) {
50 +     Customer customer = null;
51 +     try {
52 +         Connection connection = database.getConnection();
53 +         Statement statement = connection.createStatement();
54 +
55 +         ResultSet rs = statement.executeQuery("SELECT * FROM Customers WHERE id = " + customerId);
56 +         rs.next();
57 +         customer = new Customer(
58 +             customerId,
59 +             rs.getString("first"),
60 +             rs.getString("last")
61 +         );
62 +     } catch (SQLException e) {
63 +         doDatabaseErrorHandling(e);
64 +     }
65 +     return customer;
66 + }
49 69 private void doDatabaseErrorHandling(Exception e) {
50 70     e.printStackTrace();
51 71 }
```

Close Resources

It's very easy for the original code author to miss this problem, as the above code will compile happily. As the reviewer, you should spot that the connection, statement and result set all need closing before the method exits. In Java 7, this has become much easier to manage thanks to [try-with-resources](https://docs.oracle.com/javase/7/tutorial/essential/exceptions/tryResourceClose.html)¹⁴. The screenshot below shows the result of a code review where the author has changed the code to use try-with-resources.

¹⁴<https://docs.oracle.com/javase/7/tutorial/essential/exceptions/tryResourceClose.html>



try-with-resources

Are resource pools correctly configured?

The optimal configuration for an environment is going to depend on a number of factors, so it's unlikely that as a reviewer you'll know immediately if, for example, a database connection pool is correctly sized. But there are a few things you can tell at a glance, for example is the pool too small (e.g. sized at one) or too big (millions of threads). Fine tuning these values requires testing in an environment as close to the real environment as possible, but a common problem that can be identified at code review is when a pool (thread pool or connection pool, for example) is really far too large. Logic dictates that larger is better, but of course each of these objects takes up resources, and the overhead of managing thousands of them is usually much higher than the benefits of having many of them available. If in doubt, the defaults are usually a good start. Code that deviates from default settings should prove the value with some sort of test or calculation.

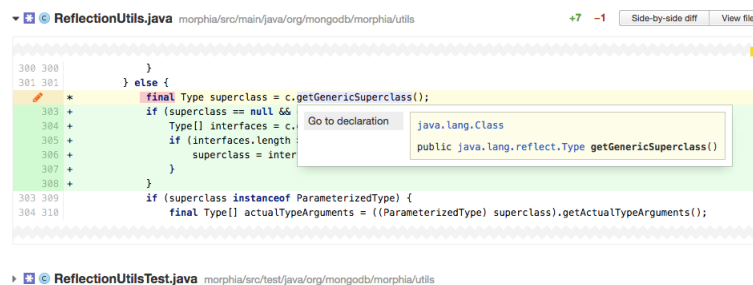
Warning signs a reviewer can easily spot

Some types of code suggests immediately a potential performance problem. This will depend upon the language and libraries used (please let us know in the comments of "code smells" in your environment).

Reflection

Reflection in Java is [slower than doing things without reflection](#)¹⁵. If you're reviewing code that contains reflection, question whether this is absolutely required.

¹⁵<http://docs.oracle.com/javase/tutorial/reflect/index.html>



Reflection

The screenshot above shows a reviewer clicking on a method in Upsource to check where it comes from, and you can see that this method is returning something from the `java.lang.reflect` package, which should be a warning sign.

Timeouts

When you're reviewing code, you might not know what the correct timeout for an operation is, but you should be thinking "what's the impact on the rest of the system while this timeout is ticking down?". As the reviewer you should consider the worst case – is the application blocking while a 5 minute timeout is ticking down? What's the worst that would happen if this was set to one second? If the author of the code can't justify the length of a timeout, and you, the reviewer, don't know the pros and cons of a selected value, then it's a good time to get someone involved who does understand the implications. Don't wait for your users to tell you about performance problems.

Parallelism

Does the code use multiple threads to perform a simple operation? Does this add more time and complexity rather than improving performance? With modern Java, this might be more subtle than creating new threads explicitly: does the code use Java 8's shiny new parallel streams but not benefit from the parallelism? For example, using a parallel stream on a small number of elements, or on a stream which does very simple operations, might be slower than performing the operations on a sequential stream.



Parallel

In the code above, the added use of **parallel** is unlikely to give us anything – the stream is acting upon a Tweet, therefore a string no longer than 140 characters. Parallelising an operation that's

going to work on so few words will probably not give a performance improvement, and the cost of splitting this up over parallel threads will almost certainly be higher than any gain.

Correctness

These things are not necessarily going to impact the performance of your system, but since they're largely related to running in a multi-threaded environment, they are related to the topic.

Is the code using the correct data structure for a multi-threaded environment?



```

1 package com.mechanitis.blog.upsourse.infrastructure;
2
11 public class BroadcastingServerEndpoint<T> extends Endpoint {
12     private final List<Session> sessions = new ArrayList<>();
13
14     @Override
15     public void onOpen(Session session, EndpointConfig config) {
16         sessions.add(session);
17     }
18
19     public void onMessage(T message) {
20         sessions.stream()
21             .filter(Session::isOpen)
22             .forEach(session -> sendMessageToClient(message.toString(), session));
23     }
24 }

```

Threadsafe

In the code above, the author is using an **ArrayList** on line 12 to store all the sessions. However, this code, specifically the **onOpen** method, is called by multiple threads, so the sessions field needs to be a thread safe data structure. For this case, we have a number of options: we could use a [Vector](http://docs.oracle.com/javase/8/docs/api/java/util/Vector.html)¹⁶, we could use [Collections.synchronizedList\(\)](http://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#synchronizedList-java.util.List)¹⁷ to create a thread safe **List**, but probably the best selection for this case is to use [CopyOnWriteArrayList](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CopyOnWriteArrayList.html)¹⁸, since the list will change far less often than it will be read.



```

4 import javax.websocket.EndpointConfig;
5 import javax.websocket.Session;
6 - import java.util.ArrayList;
7 + import java.util.List;
8 + import java.util.concurrent.CopyOnWriteArrayList;
9
10 import static com.mechanitis.blog.upsourse.infrastructure.MessageSender.sendMessageToClient;
11
12 public class BroadcastingServerEndpoint<T> extends Endpoint {
13     * private final List<Session> sessions = new ArrayListCopyOnWriteArrayList<>();
14     @Override

```

ThreadSafe 2

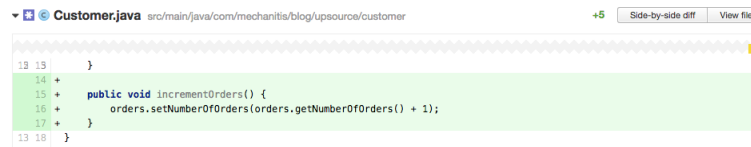
¹⁶<http://docs.oracle.com/javase/8/docs/api/java/util/Vector.html>

¹⁷<http://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#synchronizedList-java.util.List>

¹⁸<http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CopyOnWriteArrayList.html>

Is the code susceptible to race conditions?

It's surprisingly easy to write code that can cause subtle race conditions when used in a multi-threaded environment. For example:

A screenshot of a code editor window titled 'Customer.java'. The code is as follows:

```
13 15 }  
14 +  
15 + public void incrementOrders() {  
16 +     orders.setNumberOfOrders(orders.getNumberOfOrders() + 1);  
17 + }  
13 18 }
```

The code is highlighted in green. The editor interface includes a file explorer on the left, a toolbar with 'Side-by-side diff' and 'View file' buttons, and a line number indicator on the left.

Race Conditions

Although the increment code is on a single line (line 16), it's possible for another thread to increment the orders between this code getting it and this code setting the new value. As a reviewer, look out for get and set combos that are not [atomic](#)¹⁹.

Is the code using locks correctly?

Related to race conditions, as a reviewer you should be checking that the code being reviewed is not allowing multiple threads to modify values in a way that could clash. The code might need [synchronization](#)²⁰, [locks](#)²¹, or [atomic variables](#)²² to control changes to blocks of code.

Is the performance test for the code valuable?

It's easy to write [poor micro-benchmarks](#)²³, for example. Or if the test uses data that's not at all like production data, it might be giving misleading results.

Caching

While caching might be a way to prevent making too many external requests, it comes with its own challenges. If the code under review uses caching, you should look for some common problems, for example, incorrect invalidation of cache items.

Code-level optimisations

If you're reviewing code, and you're a developer, this following section may have optimisations you'd love to suggest. As a team, you need to know up-front just how important performance is to you, and whether these sorts of optimisations are beneficial to your code.

¹⁹<https://en.wikipedia.org/wiki/Linearizability>

²⁰<https://docs.oracle.com/javase/tutorial/essential/concurrency/locksinc.html>

²¹<https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/package-summary.html>

²²<https://docs.oracle.com/javase/tutorial/essential/concurrency/atomicvars.html>

²³<http://www.developertutorials.com/anatomy-flawed-microbenchmark-050426/>

For most organisations that aren't building a low-latency application, these optimisations are probably fall under the category of **premature optimisations**²⁴.

- Does the code use synchronization/locks when they're not required? If the code is always run on a single thread, locks are unnecessary overhead.
- Is the code using a thread-safe data structure where it's not required? For example, can Vector be replaced with ArrayList?
- Is the code using a data structure with poor performance for the common operations? For example, using a linked list but needing to regularly search for a single item in it.
- Is the code using locks or synchronization when it could use atomic variables instead?
- Could the code benefit from lazy loading?
- Can if statements or other logic be short-circuited by placing the fastest evaluation first?
- Is there a lot of string formatting? Could this be more efficient?
- Are the logging statements using string formatting? Are they either protected by an if to check logging level, or using a supplier which is evaluated lazily?




```

19 23 public void onMessage(T message) {
24 +   LOGGER.log(Level.FINEST, String.format("Received message %s", message));
20 25   sessions.stream()
21 26       .filter(Session::isOpen)
  
```

Logging

The code above only logs the message when the logger is set to **FINEST**. However, the expensive string format will happen every time, regardless of whether the message is actually logged.



```

23 23 public void onMessage(T message) {
24 +   if (LOGGER.isLoggable(Level.FINEST)) {
24 25       LOGGER.log(Level.FINEST, String.format("Received message %s", message));
25 26   }
25 27   sessions.stream()
26 28       .filter(Session::isOpen)
  
```

Logging

Performance can be improved by ensuring this code is only run when the log level is set to a value where the message will be written to the log, like in the code above.



```

23 23 public void onMessage(T message) {
24 24 *   LOGGER.log(Level.FINEST, () -> String.format("Received message %s", message));
25 25   sessions.stream()
26 26       .filter(Session::isOpen)
  
```

Logging

²⁴<http://c2.com/cgi/wiki?PrematureOptimization>

In Java 8, these performance gains can be obtained without the boilerplate if, by using a lambda expression. Because of the way the lambda is used, the string format will not be done unless the message is actually logged. This should be the preferred approach in Java 8.

Summary

As with my original list of things to look for in code review, this chapter highlights some areas that your team might want to consistently check for during reviews. This will depend upon the performance requirements of your project.

Although this chapter is aimed at all developers, many of the examples are Java / JVM specific. I'd like to finish with some easy things for reviewers of Java code to look for, that will give the the JVM a good chance of optimising your code so that you don't have to:

- Write small methods and classes
- Keep the logic simple – no deeply nested ifs or loops

The more readable the code is to a human, the more chance the [JIT compiler](#)²⁵ has of understanding your code enough to optimise it. This should be easy to spot during code review – if the code looks understandable and clean, it also has a good chance of performing well.

When it comes to performance, understand that there are some areas that you may be able to get quick wins in (for example, unnecessary calls to a database) that can be identified during code review, and some areas that will be tempting to comment on (like the code-level optimisations) that might not gain enough value for the needs of your system.

²⁵<http://www.oracle.com/technetwork/articles/java/architect-evans-pt1-2266278.html>

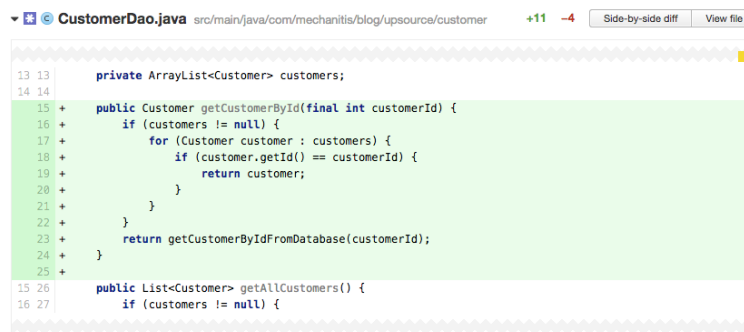
Data Structures

Data structures are a fundamental part of programming – so much so it’s actually one of the areas that’s consistently taught in Computer Science courses. And yet it’s surprisingly easy to misuse them or select the wrong one. In this post, we’re going to guide you, the code reviewer, on what to look out for – we’re going to look at examples of code and talk about “smells” that might indicate the wrong data structure was chosen or that it’s being used in an incorrect fashion.

Lists

Probably the most common choice for a data structure. Because it is the most common choice, it’s sometimes used in situations it shouldn’t be.

Anti-Pattern: Too Much Searching



```
13 13 private ArrayList<Customer> customers;
14 14
15 + public Customer getCustomerById(final int customerId) {
16 +     if (customers != null) {
17 +         for (Customer customer : customers) {
18 +             if (customer.getId() == customerId) {
19 +                 return customer;
20 +             }
21 +         }
22 +     }
23 +     return getCustomerByIdFromDatabase(customerId);
24 + }
25 +
15 26 public List<Customer> getAllCustomers() {
16 27     if (customers != null) {
```

Iterating over a list

Iterating over a list Iterating over a list is not, in itself, a bad thing of course. But if iteration is required for a very common operation (like the example above of finding a customer by ID), there might be a better data structure to use. In our case, because we always want to find a particular item by ID, it might be better to create a map of ID to Customer.

Remember that in Java 8, and languages which support more expressive searches, this might not be as obvious as a for-loop, but the problem still remains.

```

15 16     public Customer getCustomerById(final int customerId) {
16 17         if (customers != null) {
17 -             for (Customer customer : customers) {
18 -                 if (customer.getId() == customerId) {
19 -                     return customer;
20 -                 }
21 -             }
18 +             Optional<Customer> customerWithId = customers.stream()
19 +                 .filter(customer -> customer.getId() == cust
20 +                 .findFirst();
21 +             return customerWithId.orElseGet(() -> getCustomerByIdFromDatabase(customerId));
22 22     }
23 23     return getCustomerByIdFromDatabase(customerId);

```

Iterating over a list Java 8

Anti-Pattern: Frequent Reordering

```

TwitterUserRepository.java src/main/java/com/mechanitis/blog/upsourse/social +25 Side-by-side diff View file
1 + package com.mechanitis.blog.upsourse.social;
2 +
3 + import java.util.ArrayList;
4 + import java.util.Collections;
5 + import java.util.Comparator;
6 + import java.util.List;
7 +
8 + public class TwitterUserRepository {
9 +     private final List<TwitterUser> twitterUsers = new ArrayList<>();
10 +
11 +     public void addTwitterUser(TwitterUser twitterUser) {
12 +         twitterUsers.add(twitterUser);
13 +     }
14 +
15 +     public List<TwitterUser> getTwitterUsers() {
16 +         Collections.sort(twitterUsers, new Comparator<TwitterUser>() {
17 +             public int compare(TwitterUser o1, TwitterUser o2) {
18 +                 return o1.getTwitterHandle().compareTo(o2.getTwitterHandle());
19 +             }
20 +         });
21 +
22 +         return twitterUsers;
23 +     }
24 + }

```

Frequent Reordering

Lists are great if you want to stick to their default order, but if as a reviewer you see code that's re-sorting the list, question whether a list is the correct choice. In the code above, on line 16 the `twitterUsers` list is always re-sorted before being returned. Once again, Java 8 makes this operation look so easy it might be tempting to ignore the signs:

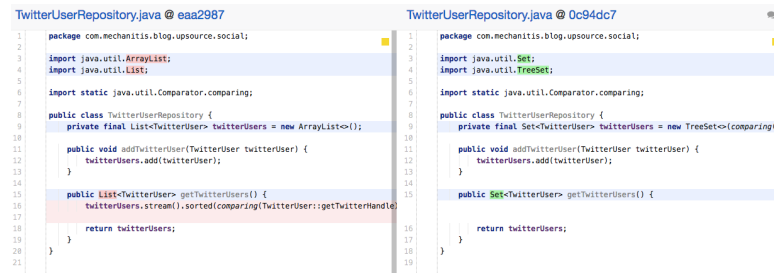
```

15 15     public List<TwitterUser> getTwitterUsers() {
16 -         Collections.sort(twitterUsers, new Comparator<TwitterUser>() {
17 -             public int compare(TwitterUser o1, TwitterUser o2) {
18 -                 return o1.getTwitterHandle().compareTo(o2.getTwitterHandle());
19 -             }
20 -         });
16 +         twitterUsers.stream().sorted(comparing(TwitterUser::getTwitterHandle));
21 17
22 18     return twitterUsers;

```

List Sorting of Java 8

In this case, given that a `TwitterUser` is unique and it looks like you want a collection that's sorted by default, you probably want something like a `TreeSet`.



Twitter User Diff

Maps

A versatile data structure that provide $O(1)^{26}$ access to individual elements, if you've picked the right key.

Anti-Pattern: Map as global constant

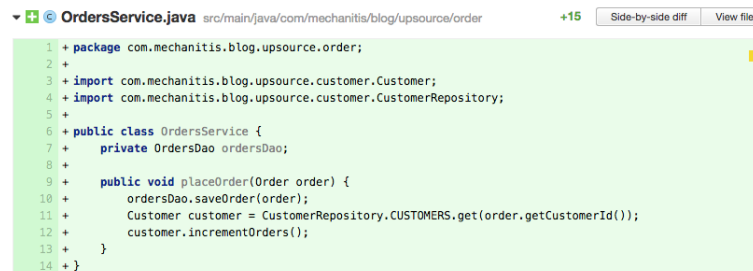
The map is such a good general purpose data structure that it can be tempting to use globally accessible maps to let any class get to the data.



Global Map

In the above code, the author has chosen to simply expose the `CUSTOMERS` map as a global constant. The `CustomerUpdateService` therefore uses this map directly when adding or updating customers. This might not seem too terrible, since the `CustomerUpdateService` is responsible for add and update operations, and these have to ultimately change the map. The issue comes when other classes, particularly those from other parts of the system, need access to the data.

²⁶https://en.wikipedia.org/wiki/Big_O_notation#Orders_of_common_functions

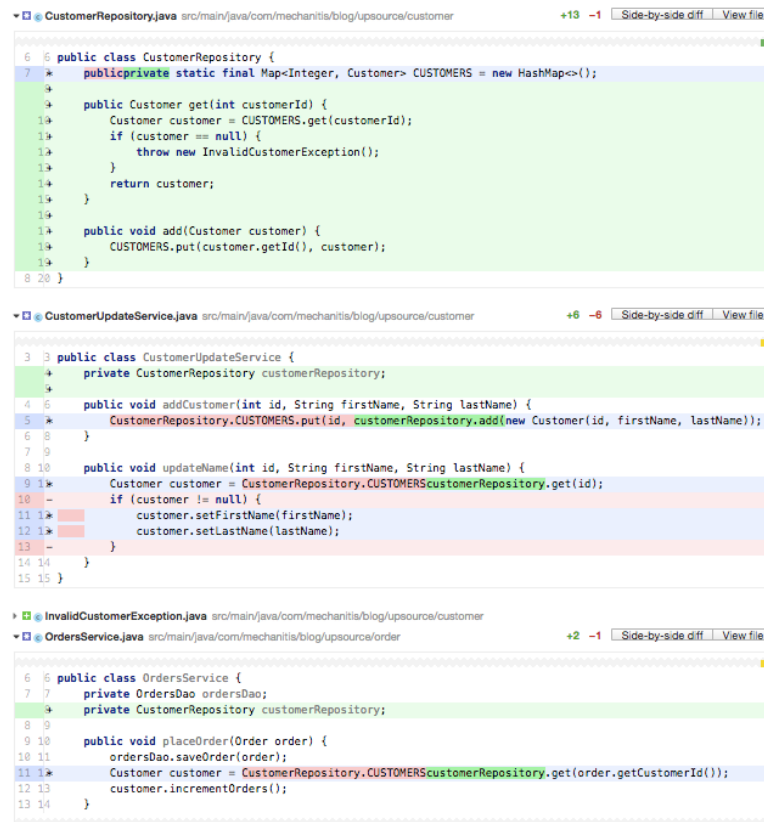


```
1 + package com.mechanitis.blog.upsources.order;
2 +
3 + import com.mechanitis.blog.upsources.customer.Customer;
4 + import com.mechanitis.blog.upsources.customer.CustomerRepository;
5 +
6 + public class OrdersService {
7 +     private OrdersDao ordersDao;
8 +
9 +     public void placeOrder(Order order) {
10 +         ordersDao.saveOrder(order);
11 +         Customer customer = CustomerRepository.CUSTOMERS.get(order.getCustomerId());
12 +         customer.incrementOrders();
13 +     }
14 + }
```

Orders Service

Here, the order service is aware of the data structure used to store customers. In fact, in the code above, the author has made an error – they don't check to see if the customer is null, so line 12 could cause a **NullPointerException**. As the reviewer of this code, you'll want to suggest hiding this data structure away and providing suitable access methods. That will make these other classes easier to understand, and hide any complexity of managing the map in the **CustomerRepository**, where it belongs. In addition, if later you change the customers data structure, or you move to using a distributed cache or some other technology, the changes associated with that will be restricted to the **CustomerRepository** class and not ripple throughout the system. This is the principle of [Information Hiding](https://en.wikipedia.org/wiki/Information_hiding)²⁷.

²⁷https://en.wikipedia.org/wiki/Information_hiding



```

6 public class CustomerRepository {
7     * publicprivate static final Map<Integer, Customer> CUSTOMERS = new HashMap<>();
8
9     public Customer get(int customerId) {
10         Customer customer = CUSTOMERS.get(customerId);
11         if (customer == null) {
12             throw new InvalidCustomerException();
13         }
14         return customer;
15     }
16
17     public void add(Customer customer) {
18         CUSTOMERS.put(customer.getId(), customer);
19     }
20 }

```

```

3 public class CustomerUpdateService {
4     private CustomerRepository customerRepository;
5
6     public void addCustomer(int id, String firstName, String lastName) {
7         * CUSTOMERS.put(id, customerRepository.add(new Customer(id, firstName, lastName));
8     }
9
10    public void updateName(int id, String firstName, String lastName) {
11        * Customer customer = CUSTOMERScustomerRepository.get(id);
12        - if (customer != null) {
13            * customer.setFirstName(firstName);
14            * customer.setLastName(lastName);
15        }
16    }
17 }

```

```

6 public class InvalidCustomerException {
7     *
8 }

```

```

6 public class OrdersService {
7     private OrdersDao ordersDao;
8     * private CustomerRepository customerRepository;
9
10    public void placeOrder(Order order) {
11        ordersDao.saveOrder(order);
12        * Customer customer = CUSTOMERScustomerRepository.get(order.getCustomerId());
13        * customer.incrementOrders();
14    }
15 }

```

Hidden

Although the updated code isn't much shorter, you have standardised and centralised core functions – for example, you know that getting a customer who doesn't exist is always going to give you an Exception. Or you can choose to have this method return the new Optional type.

Note that this is exactly the sort of issue that should be found during a code review – hiding global constants is hard to do once their use has propagated throughout the system, but it's easy to catch this when they're first introduced.

Anti-Pattern: Iteration and Reordering

As with lists, if a code author has introduced a lot of sorting of, or iterating over, a map, you might want to suggest an alternative data structure.

Java-specific things to be aware of

In Java, map behaviour usually relies on your implementation of equals and hashCode for the key and the value. As a reviewer, you should check these methods on the key and value classes to ensure you're getting the expected behaviour.

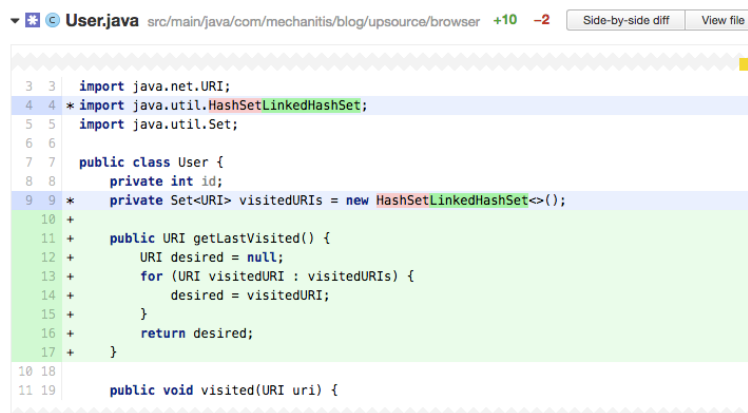
Java 8 has added a number of very useful methods to the Map interface. The getOrDefault method, for example, could simplify the CustomerRepository code at line 11 in the example above.

Sets

An often-underused data structure, its strength is that it does not contain duplicate elements.

Anti-pattern: Sometimes you really do want duplicates

Let's assume you had a user class that used a set to track which website they had visited. Now, the new feature is to return the most recently visited of these websites.



```
3 3 import java.net.URI;
4 4 * import java.util.HashSetLinkedHashSet;
5 5 import java.util.Set;
6 6
7 7 public class User {
8 8     private int id;
9 9 * private Set<URI> visitedURIs = new HashSetLinkedHashSet<>();
10 +
11 + public URI getLastlyVisited() {
12 +     URI desired = null;
13 +     for (URI visitedURI : visitedURIs) {
14 +         desired = visitedURI;
15 +     }
16 +     return desired;
17 + }
10 18
11 19 public void visited(URI url) {
```

Sets

The author of this code has changed the initial set that tracks the sites a user has visited from `HashSet` to `LinkedHashSet`²⁸ – this latter implementation preserves insertion order, so now our set tracks every URI in the order in which they were visited.

There are a number of signs in this code that this is wrong though. Firstly, the author has had to do a costly full iteration of the whole set to reach the last element (lines 13-15) – sets are not designed for accessing elements by position, something that lists are perfect for. Secondly, because sets do not contain duplicate values, if the last page they visited had been visited previously, it will not be in the last position in the set. Instead, it will be where it was first added to the set.

In this case, a list, a stack (see below), or even just a single field, might give us better access to the last page visited.

Java-specific things to be aware of

Because one of the key operations of a set is contains, as a reviewer you should be checking the implementation of equals on the type contained in the set.

²⁸<http://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashSet.html>

Stacks

[Stacks](#)²⁹ are a favourite of Computer Science classes, and yet in the real world are often overlooked – in Java, maybe this is because Stack is an extension of Vector and therefore somewhat old-fashioned. Rather than going into a lot of detail here I'll just cover key points:

- Stacks support [LIFO](#)³⁰, and should ideally be used with push/pop operations, it's not really for iterating over.
- The class you want for a stack implementation in Java (since version 1.6) is [Deque](#)³¹. This can act as both a queue and a stack, so reviewers should check that dequeues are used in a consistent fashion in the code.

Queues

Another Computer Science favourite. [Queues](#)³² are often spoken about in relation to concurrency (indeed, most of the Java implementations live in [java.util.concurrent](#)³³), as it's a common way to pass data between threads or modules.

- Queues are [FIFO](#)³⁴ data structures, generally working well when you want to add elements to the tail of the queue, or remove things from the front of the queue. If you're reviewing code that shows iteration over a queue (in particularly accessing elements in the middle of the queue), question if this is the correct data type.
- Queues can be bounded or unbounded. Unbounded queues could potentially grow forever, so if reviewing code with this type of data structure, check out the earlier post on performance. Bounded queues can come with their own problems too – when reviewing code, you should look for the conditions under which the queue might become full, and ask what happens to the system under these circumstances.

A general note for Java developers

As a reviewer, you should be aware not only of the characteristics of general data structures, but you should also be aware of the strengths and weaknesses of each of the implementations all of which are documented in the Javadoc:

- [Javadoc for List](#)³⁵

²⁹[https://en.wikipedia.org/wiki/Stack_\(abstract_data_type\)](https://en.wikipedia.org/wiki/Stack_(abstract_data_type))

³⁰<https://en.wikipedia.org/wiki/LIFO>

³¹<http://docs.oracle.com/javase/8/docs/api/java/util/Deque.html>

³²[https://en.wikipedia.org/wiki/Queue_\(abstract_data_type\)](https://en.wikipedia.org/wiki/Queue_(abstract_data_type))

³³<http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html>

³⁴[https://en.wikipedia.org/wiki/FIFO_\(computing_and_electronics\)](https://en.wikipedia.org/wiki/FIFO_(computing_and_electronics))

³⁵<http://docs.oracle.com/javase/8/docs/api/java/util/List.html>

- [Javadoc for Map](#)³⁶
- [Javadoc for Set](#)³⁷

If you're using Java 8, remember that many of the collections classes have new methods. As a reviewer you should be aware of these – you should be able to suggest places where the new methods can be used in place of older, more complex code.

Why select the right data structure?

We've spent this chapter looking at data structures – how to tell if the code under review might be using the wrong data structures, and some pointers for the pros and cons of various data structure so not only can you, as the reviewer, identify when they might be being used incorrectly, but you can also suggest better alternatives. But let's talk about why using the right data structure is important.

Performance

If you've studied data structures in computer science, you'll often learn about the performance implications of picking one over another. Indeed, we even mentioned “[Big O Notation](#)”³⁸ in this chapter, to highlight some of the strengths of particular structures. Using the right data structure in your code can definitely help performance, but this is not the only reason to pick the right tool for the job.

Stating Expected Behaviour

Developers who come to the code later, or who use any API exposed by your system, will make certain assumptions based on data structures. If the data returned from a method call is in a list, a developer will assume it is ordered in some fashion. If data is stored in a map, a developer can assume that there is a frequent need to look up individual elements by the key. If data is in a set, then a developer can assume it's intentional that this only stores an element once and not multiple times. It's a good idea to [work within these assumptions rather than break them](#)³⁹.

Reducing Complexity

The overall goal of any developer, and especially of a reviewer, should be to ensure the code does what it's supposed to do with the minimal amount of complexity – this makes the code easier to read, easier to reason about, and easier to change and maintain in the future. In some of the anti-patterns above, for example the misuse of Set, we can see that picking the wrong data structure forced the author to write a lot more code. Selecting the right data structure should, generally, simplify the code.

³⁶<https://docs.oracle.com/javase/8/docs/api/java/util/Map.html>

³⁷<http://docs.oracle.com/javase/8/docs/api/java/util/Set.html>

³⁸https://en.wikipedia.org/wiki/Big_O_notation

³⁹https://en.wikipedia.org/wiki/Principle_of_least_astonishment

Summary

Picking the right data structure is not simply about gaining performance or looking clever in front of your peers. It also leads to more understandable, maintainable code. Common signs that the code author has picked the wrong data structure:

- Lots of iterating over the data structure to find some value or values
- Frequent reordering of the data
- Not using methods that provide key features – e.g. push or pop on a stack
- Complex code either reading from or writing to the data structure

In addition, exposing the details of selected data structures, either by providing global access to the structure itself, or by tightly coupling your class's interface to the operation of an underlying data structure, leads to a brittleness of design, and will be hard to undo later. It's better to catch these problems early on, for example during a code review, than incur avoidable technical debt.

SOLID Principles

In this chapter we'll look more closely at the design of the code itself, specifically checking to see if it follows good practice Object Oriented Design. As with all the other areas we've covered, not all teams will prioritise this as the highest value area to check, but if you are trying to follow SOLID Principles, or trying to move your code in that direction, here are some pointers that might help.

What is SOLID?

The SOLID Principles are five core principles of Object Oriented design and programming. The purpose of this post is not to educate you on what these principles are or go into depth about why you might follow them, but instead to point those performing code reviews to code smells that might be a result of not following these principles.

SOLID stands for:

- [Single Responsibility Principle](#)⁴⁰
- [Open-Closed Principle](#)⁴¹
- [Liskov Substitution Principle](#)⁴²
- [Interface Segregation Principle](#)⁴³
- [Dependency Inversion Principle](#)⁴⁴

Single Responsibility Principle (SRP)

There should never be more than one reason for a class to change.

This can sometimes be hard to spot from a single code review. By definition, the author is (or should be) applying a single reason to change the code base – a bug fix, a new feature, a focussed refactoring.

You want to look at which methods in a class are likely to change at the same time, and which clusters of methods are unlikely to ever be changed by a change to the other methods. For example:

⁴⁰<http://www.objectmentor.com/resources/articles/srp.pdf>

⁴¹<http://www.objectmentor.com/resources/articles/ocp.pdf>

⁴²<http://www.objectmentor.com/resources/articles/lsp.pdf>

⁴³<http://www.objectmentor.com/resources/articles/isp.pdf>

⁴⁴<http://www.objectmentor.com/resources/articles/dip.pdf>



It would be better to use polymorphism to remove this if:

```

EventInterceptor.java @ 91c499d
3 public class EventInterceptor {
4     public void preLoad(Object entity) {
5
6     }
7
8     public void postLoad(Object entity) {
9     }
10
11     public void prePersist(Object entity) {
12     }
13
14     public void preSave(Object entity) {
15     }
16
17     public void postPersist(Object entity) {
18     }
19
20 }
21
22
23
24

EventInterceptor.java @ d0e6048
10 public class EventInterceptor {
11     public void interceptEvent(PreLoad event, Object entity) {
12         //do pre-load stuff here
13     }
14
15     public void interceptEvent(PostLoad event, Object entity) {
16         //do post-load stuff here
17     }
18
19     public void interceptEvent(PrePersist event, Object entity) {
20         //do pre-persist stuff here
21     }
22
23     public void interceptEvent(PreSave event, Object entity) {
24         //do pre-save stuff here
25     }
26
27     public void interceptEvent(PostPersist event, Object entity) {
28         //do post-persist stuff here
29     }
30
31     public void interceptEvent(Event event, Object entity) {
32         //do some logic to routing events to the events
33     }
34
35 }

```

Another Diff

As always, there's more than one solution to this problem, but the key will be removing the complex if/else*8 and the **instanceof checks.

Liskov Substitution Principle (LSP)

Functions that use references to base classes must be able to use objects of derived classes without knowing it.

One easy way to spot violations of this principle is to look for explicit casting. If you have to cast a object to some type, you are not using the base class without knowledge of the derived classes.

More subtle violations can be found when checking:

- Preconditions cannot be strengthened in a subtype⁴⁵
- Postconditions cannot be weakened in a subtype⁴⁶

Imagine, for example, we have an abstract **Order** with a number of subclasses – **BookOrder**, **ElectronicsOrder** and so on. The **placeOrder** method could take a **Warehouse**, and could use this to change the inventory levels of the physical items in the warehouse:

```

3 ↓ public abstract class Order {
4 ↓   public void placeOrder(Warehouse warehouse) {
5     warehouse.itemSold(getItemId());
6     // do other order-related activities...
7   }
8
9   protected abstract int getItemId();
10 }

```

Order

⁴⁵https://en.wikipedia.org/wiki/Liskov_substitution_principle

⁴⁶https://en.wikipedia.org/wiki/Liskov_substitution_principle

Now imagine we introduce the idea of electronic gift cards, which simply add balance to a wallet but do not require physical inventory. If implemented as a **GiftCardOrder**, the **placeOrder** method would not have to use the warehouse parameter:

```

7 + public class GiftCardOrder extends Order {
8 +     private BigDecimal price;
9 +     private Customer customer;
10 +
11 +     public GiftCardOrder(Customer customer, BigDecimal price) {
12 +         this.customer = customer;
13 +         this.price = price;
14 +     }
15 +
16 +     @Override
17 +     public void placeOrder(Warehouse warehouse) {
18 +         customer.addBalanceToWallet(price);
19 +     }

```

Stuff

This might seem like a logical use of inheritance, but in fact you could argue that code that uses **GiftCardOrder** could expect similar behaviour from this class as the other classes, i.e. you could expect this to pass for all subtypes:

```

18 |
19 | @Test
20 | public void shouldRemoveItemFromInventory() {
21 |     List<Order> allOrderTypes = asList(new BookOrder(ITEM_ID),
22 |                                       new ElectronicsOrder(ITEM_ID),
23 |                                       new GiftCardOrder(CUSTOMER, PRICE));
24 |
25 |     for (Order order : allOrderTypes) {
26 |         Warehouse warehouse = mock(Warehouse.class);
27 |         order.placeOrder(warehouse);
28 |
29 |         verify(warehouse).itemSold(ITEM_ID);
30 |     }

```

Liskov Review 2

But this will not pass, as GiftCardOrders have a different type of order behaviour. If you're reviewing this sort of code, question the use of inheritance here – maybe the order behaviour can be plugged in using [composition instead of inheritance](#)⁴⁷.

Interface Segregation Principle (ISP)

Many client specific interfaces are better than one general purpose interface.

Some code that violates this principle will be easy to identify due to having interfaces with a lot of methods on. This principle compliments SRP, as you may see that an interface with many methods is actually responsible for more than one area of functionality.

But sometimes even an interface with just two methods could be split into two interfaces:

⁴⁷https://en.wikipedia.org/wiki/Composition_over_inheritance

```

6 + public class StringCodec implements SimpleCodec<String> {
7 +
8 +     @Override
9 +     public String decode(Reader reader) {
10 +         throw new UnsupportedOperationException("Should never have to decode a String object");
11 +     }
12 +
13 +     @Override
14 +     public void encode(final String value, Writer writer) {
15 +         writer.writeString(value);
16 +     }
17 + }

```

ISP

In this example, given that there are times when the **decode** method might not be needed, and also that a codec can probably be treated as either an encoder or a decoder depending upon where it's used, it may be better to split the **SimpleCodec** interface into an **Encoder** and a **Decoder**. Some classes may choose to implement both, but it will not be necessary for implementations to override methods they do not need, or for classes that only need an **Encoder** to be aware that their **Encoder** instance also implements **decode**.

Mino Dependency Inversion Principle (DIP)

Depend upon Abstractions. Do not depend upon concretions.

While it may be tempting to look for simple cases that violate this, like liberal use of the **new** keyword (instead of using [Dependency Injection](#)⁴⁸ or [factories](#)⁴⁹, for example) and overfamiliarity with your collection types (e.g. declaring **ArrayList** variables or parameters instead of **List**), as a reviewer you should be looking to make sure the code author has used or created the correct abstractions in the code under review.

For example, service-level code that uses a direct connection to a database to read and write data:

```

6 12 public class CustomerService {
13 +     private static final String INSERT_CUSTOMER_STATEMENT = "INSERT INTO Customers"
14 +         + "(id, first_name, last_name)"
15 +         + "VALUES"
16 +         + "( ? , ? , ? )";
17 +     private Database database;
18 +
19 +     public CustomerService(Database database) {
20 +         this.database = database;
21 +     }
22 +
23 +     public void addCustomer(int id, String firstName, String lastName) {
24 +         try (Connection connection = database.getConnection();
25 +             PreparedStatement statement = connection.prepareStatement(INSERT_CUSTOMER_STATEMENT)) {
26 +             statement.setInt(1, id);
27 +             statement.setString(2, firstName);
28 +             statement.setString(3, lastName);
29 +
30 +             statement.executeUpdate();
31 +         } catch (SQLException e) {
32 +             doDatabaseErrorHandling(e);
33 +         }
34 +     }
35 +
36 +     public void placeOrder(Customer customer, Order order) {
37 +         customer.incrementOrders();
38 +         order.placeOrder(getWarehouse());
39 +     }

```

DI

⁴⁸<http://www.martinfowler.com/articles/injection.html>

⁴⁹https://en.wikipedia.org/wiki/Abstract_factory_pattern

This code is dependent on a lot of specific implementation details: JDBC as a connection to a (relational) database; database-specific SQL; knowledge of the database structure; and so on. This does belong somewhere in your system, but not here where there are other methods that don't need to know about databases. Better to extract a [DAO](#)⁵⁰ or use the [Repository pattern](#)⁵¹, and inject the DAO or repository into this service.

Summary

Some code smells that might indicate one or more of the SOLID Principles have been violated:

- Long **if/else** statements
- Casting to a subtype
- Many public methods
- Implemented methods that throw **UnsupportedOperationException**

As with all design questions, finding a balance between following these principles and knowingly bending the rules is down to your team's preferences. But if you see complex code in a code review, you might find that applying one of these principles will provide a simpler, more understandable, solution.

⁵⁰https://en.wikipedia.org/wiki/Data_access_object

⁵¹<http://martinfowler.com/eaCatalog/repository.html>

Security

How much work you do building a secure, robust system is like anything else on your project – it depends upon the project itself, where it's running, who's using it, what data it has access to, etc. Often, if our team doesn't have access to security experts, we go too far in one direction or the other: either we don't pay enough attention to security issues; or we go through some compliance checklist and try to address everything in some 20 page document filled with potential issues.

This chapter aims to highlight some areas you might like to look at when reviewing code, but mostly it aims to get you having discussions within your team or organisation to figure out what it is you do need to care about in a code review.

Automation is your friend

A surprising number of security checks can be automated, and therefore should not need a human. Security tests don't necessarily have to be full-blown penetration testing with the whole system up and running, some problems can be found at code-level.

Common problems like [SQL Injection](#)⁵² or [Cross-site Scripting](#)⁵³ can be found via tools running in your Continuous Integration environment. You can also automate checking for known vulnerabilities in your dependencies via the [OWASP Dependency](#)⁵⁴ Check tool.

Of course, Upsource also provides numerous security inspections. These can inform a reviewer of potential security problems in the code. For example, this code executes a dynamically generated SQL string, which might be susceptible to SQL Injection:

```
private Customer getCustomerByIdFromDatabase(int customerId) {
    Customer customer = null;
    try (Connection connection = database.getConnection();
        Statement statement = connection.createStatement()) {
        try (ResultSet rs = statement.executeQuery("SELECT * FROM Customers WHERE id = " + customerId)) {
            rs.next();
            customer = new Customer(
                customerId,
                rs.getString("first"),
                rs.getString("last")
            );
        }
    } catch (SQLException e) {
        doDatabaseErrorHandling(e);
    }
    return customer;
}
```

Code Inspection

⁵²[https://www.owasp.org/index.php/Testing_for_SQL_Injection_\(OTG-INPVAL-005\)](https://www.owasp.org/index.php/Testing_for_SQL_Injection_(OTG-INPVAL-005))

⁵³https://www.owasp.org/index.php/Testing_for_Cross_site_scripting

⁵⁴https://www.owasp.org/index.php/OWASP_Dependency_Check

Sometimes “It Depends”

While there are checks that you can feel comfortable with a “yes” or “no” answer, sometimes you want a tool to point out potential problems and then have a human make the decision as to whether this needs to be addressed or not. This is an area where Upsource can really shine. Upsource [displays code inspections](#)⁵⁵ that a reviewer can use to decide if the code needs to be changed or is acceptable under the current situation.

For example, suppose you’re generating a random number. If all your security checks are enabled, you’ll see the following warning in Upsource:



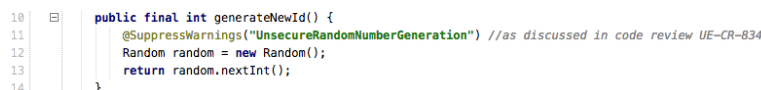
```
10  public final int generateNewId() {
11      Random random = new Random();
12      return random.nextInt();
13  }
```

For security purposes, use 'java.security.SecureRandom' instead of 'java.util.Random'

Security Warning - Random Number

The JavaDoc for **java.util.Random** specifically states “Instances of **java.util.Random** are not cryptographically secure”. This may be fine for many of the occasions when you need an arbitrary random number. But if you’re using it for something like session IDs, password reset links, [nonces](#)⁵⁶ or [salts](#)⁵⁷, as a reviewer you might suggest replacing **Random** with **java.util.SecureRandom**.

If you and the code author decide that **Random** is appropriate for this situation, then it’s a good idea to suppress this inspection for this line of code, and document why it’s OK or point to any discussion on the subject – this way future developers looking at the code can understand this is a deliberate decision.



```
10  public final int generateNewId() {
11      @SuppressWarnings("UnsecureRandomNumberGeneration") //as discussed in code review UE-CR-834
12      Random random = new Random();
13      return random.nextInt();
14  }
```

Suppress Warning

So while tools can definitely point you at potential issues, part of your job as a code reviewer is to investigate the results of any automated checks and decide which action to take.

⁵⁵https://www.jetbrains.com/upsource/help/2.0/display_code_inspection.html

⁵⁶https://en.wikipedia.org/wiki/Cryptographic_nonce

⁵⁷[https://en.wikipedia.org/wiki/Salt_\(cryptography\)](https://en.wikipedia.org/wiki/Salt_(cryptography))



Upsource

If you are using Upsource to review your code, you can customise your inspection settings, including selecting security settings. Do this by opening your project in IntelliJ IDEA and navigating to the Inspections settings. Select the settings you want and save them to the Project Default profile. Make sure Project_Default.xml is checked in with your project code, and Upsource will use this to determine which inspections to run.

At the time of writing, these are the available security inspections:

▼ Java	■	✓
▼ Security issues	■	✓
Access of system properties	■	✓
Call to 'Runtime.exec()'	■	✓
Call to 'System.loadLibrary()' with non-constant string	■	✓
Call to 'System.setSecurityManager()'	■	✓
ClassLoader instantiation	■	✓
Cloneable class in secure context	■	✓
'Connection.prepare*()' call with non-constant string	■	✓
Custom ClassLoader	■	✓
Custom SecurityManager	■	✓
Deserializable class in secure context	■	✓
Design for extension	■	✓
Insecure random number generation	■	✓
Non-'static' inner class in secure context	■	✓
Non-final 'clone()' in secure context	■	✓
'public static' array field	■	✓
'public static' collection field	■	✓
Serializable class in secure context	■	✓
'Statement.execute()' call with non-constant string	■	✓
▼ JavaScript	■	✓
▼ Potentially confusing code constructs	■	✓
Execution of dynamically generated JavaScript	■	✓
▼ Spring		✓
▼ Spring Security		✓
Debug activated	■	✓
Spring Security Model	■	✓
Web.xml Filter Configuration	■	✓

Security Inspections

Understand your Dependencies

Let's move on to other areas that need a human reviewer. One of the areas where security vulnerabilities can creep into your system or code base is via third party libraries. When reviewing code, at the very least you want to check if any new dependencies (e.g. third party libraries) have

been introduced. If you aren't already automating the check for vulnerabilities, you should check for known issues in newly-introduced libraries.

You should also try to minimise the number of versions of each library – not always possible if other dependencies are pulling in additional transitive dependencies. But one of the simplest way to minimise your exposure to security problems in other people's code (via libraries or services) is to

- Use a few as sources as possible and understand how trustworthy they are
- Use the highest quality library you can
- Track what you use and where, so if new vulnerabilities do become apparent, you can check your exposure.

This means:

- Understanding your sources (e.g. maven central or your own repo vs arbitrarily downloaded jar files)
- Trying not to use 5 different versions of 3 different logging frameworks (for example)
- Being able to view your dependency tree, even if it's simply through Gradle/Maven

Check if new paths & services need to be authenticated

Whether you're working on a web application, or providing web services or some other API which requires authentication, when you add a new URI or service, you should ensure that this cannot be accessed without authentication (assuming authentication is a requirement of your system). You may simply need to check that the developer of the code wrote an appropriate test to show that authentication has been applied.

You should also consider that authentication isn't just for human users with a username and password. Identity might need to be defined for other systems or automated processes accessing your application or services. This may impact your concept of "user" in your system.

Does your data need to be encrypted?

When you're storing something on disk or sending things over the wire, you need to know whether that data should be encrypted. Obviously passwords should never be in plain text, but there are plenty other times when data needs to be encrypted. If the code under review is sending data on the wire, saving it somewhere, or it is in some way leaving your system, if you don't know whether it should be encrypted or not, try and locate someone in your organisation who can answer that question.

Are secrets being managed correctly?

Secrets are things like passwords (user passwords, or passwords to databases or other systems), encryption keys, tokens and so forth. These should never be stored in code, or in configuration files that get checked into the source control system. There are other ways of managing these secrets, for example via a secrets server. When reviewing code, make sure these things don't accidentally sneak into your VCS.

Should the code be logging/auditing behaviour? Is it doing so correctly?

Logging and auditing requirements vary by project, with some systems requiring compliance with stricter rules for logging actions and events than others. If you do have guidelines on what needs logging, when and how, then as a code reviewer you should be checking the submitted code meets these requirements. If you do not have a firm set of rules, consider:

- Is the code making any data changes (e.g. add/update/remove)? Should it make a note of the change that was made, by whom, and when?
- Is this code on some performance-critical path? Should it be making a note of start-time and end-time in some sort of performance monitoring system?
- Is the logging level of any logged messages appropriate? A good rule of thumb is that “ERROR” is likely to cause an alert to go off somewhere, possibly on some poor on-call person's pager – if you do not need this message to wake someone up at 3am, consider downgrading to “INFO” or “DEBUG”. Messages inside loops, or other places that are likely to be output more than once in a row, probably don't need to be spamming your production log files, therefore are likely to be “DEBUG” level.

Summary

This is just a tiny subset of the sorts of security issues you can be checking in a code review. Security is a very big topic, big enough that your company may hire technical security experts, or at least devote some time or resources to this area. However, like other non-coding activities such as getting to know the business and having a decent grasp of how to test the system, understanding the security requirements of our application, or at least of the feature or defect we're working on right now, is another facet of our job as a developer.

We can enlist the help of security experts if we have them, for example inviting them to the code review, or inviting them to pair with us while we review. Or if this isn't an option, we can learn enough about the environment of our system to understand what sort of security requirements we have (internal-facing enterprise apps will have a different profile to customer-facing web applications, for example), so we can get a better understanding of what we should be looking for in a code review.

And like many other things we're tempted to look for in code reviews, many security checks can also be automated, and should be run in our continuous integration environment. As a team, you need to discuss which things are important to you, whether checking these can be automated, and which things you should be looking for in a code review.

This post has barely scratched the surface of potential issues. We'd love to hear from you in the comments – let us know of other security gotchas we should be looking for in our code.

Upsource Quick Wins

We've had two themes running through this book on what to look for in a code review:

- Lots of what we're tempted to look for can (and should) be automated
- Humans are good at checking things computers can not.

In this chapter, we're going to look at the grey area in between – we're going to look at how the features in Upsource can make your job as a human code reviewer easier.

Upsource is an the only code review tool that gives developers **IDE-level code insight** including code navigation, Find Usages, static code analysis, and powerful search. It understands your code and makes code reviews easier, faster and much more efficient.

As a developer, code reviews can sometimes be a frustrating process – you're used to all the power of your IDE, yet when you open up a set of changes in your code review tool you can't leverage any of that. The cognitive load of having to navigate through the code in an unfamiliar way and losing all the context your IDE provides is one of the things that makes developers less keen to perform code reviews.

Let's look at some of the features that Upsource provides that overcome these issues.

Navigation

It might seem like a trivial thing, but the ability to navigate through the code via the code is something we simply take for granted when we use an IDE like IntelliJ IDEA. Yet the simple Cmd + click feature to navigate to a given class, or to find the declaration of a field, is often missing when we open code in a tool that is not our IDE. Upsource lets you click on a class name, method, field or variable to navigate to the declaration.



Symbol actions

While this is very useful, something I find myself doing a even more in my IDE is pressing Alt + F7 to find usages of a class, method or field. This is especially useful during code review, because if a method or class has been changed in some way, you as the reviewer want to see what the impact of this change is – which means locating all the places it's used. You can see from the screenshot above that this is easily done in Upsource – clicking on a symbol gives you the option to highlight the usages in the file, or to find usages in the project.

Inspections

Intuitive navigation is great for a reviewer as it lets you browse through the code in a way that's natural for you, rather than having some arbitrary order imposed on you – it makes it easier to see the context of the changes under review.

But there's another IDE feature that would be extremely useful during code review – inspections. If you're already using IntelliJ IDEA, for example, you're probably used to the IDE giving you pointers on where the code could be simpler, clearer, and generally a bit better. If your code review tool offered the same kind of advice, you could easily check that all new/updated code doesn't introduce new obvious issues, and possibly even cleans up long-standing problems.

Upsource uses the [IntelliJ IDEA inspections](#)⁵⁸ – we actually covered how to enable them for Upsource in the previous chapter. There are rather a lot of inspections available in IntelliJ IDEA, so we're just going to give a taste of what's possible – we're going to cover some of the default ones that you may find useful during code review.

Exception Handling Issues

Inspections can catch potential problems around how error conditions are handled. For example, empty catch blocks.

```
try {
    Connection connection = database.getConnection();
    PreparedStatement statement = connection.prepareStatement("SELECT * FROM Customers ORDER BY id") {
        try {
            ResultSet rs = statement.executeQuery();
            rs.next();
            customers.add(extractCustomer(rs));
        }
    }
} catch (SQLException e) {
    // Empty 'catch' block
}
```

Empty catch block

Solutions

It's difficult to think of a time when catching and ignoring an Exception is the right thing to do. A code reviewer should be suggesting:

⁵⁸<https://www.jetbrains.com/idea/help/code-inspection.html>

- Catching the Exception and wrapping it in a more appropriate one, possibly a RuntimeException, that can be handled at the right level.
- Logging the Exception (we also touched on appropriate logging in the chapter on security).
- At the very least, documenting why this is OK. If there's a comment in the catch block, it's no longer flagged by the inspection.

```
@Test
public void shouldThrowAnExceptionWhenCustomerNotFound() {
    // given
    CustomerDao customerDao = new CustomerDao();

    // expect
    try {
        customerDao.getCustomerById(ID_OF_CUSTOMER_THAT_DOES_NOT_EXIST);
        Assert.fail("Should have thrown an Exception");
    } catch (CustomerNotFoundException e) {
        // this is expected
    }
}
```

ExpectedException

Configuring

“Empty ‘catch’ block” is enabled in the default set of inspections. This and other related inspections can be found in [IntelliJ IDEA's inspections](#)⁵⁹ settings under Java > Error Handling.

Probable Bugs

There are a number of inspections available for “probable bugs”. These inspections highlight things that the compiler allows, as they're valid syntax, but are probably not what the author intended.

```
} catch (SQLException e) {
    LOGGER.log(Level.WARNING, String.format("Error querying the database %s"));
}
return customers;
```

Too few arguments for format string (found: 0, expected: 1)

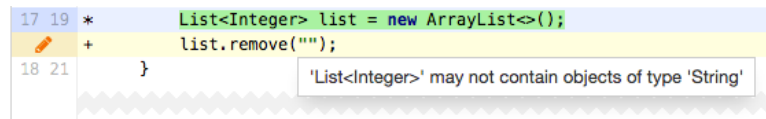
String format potential bug

Examples

- `String.format()` issues like the one above.
- Comparing Strings using `==` not `.equals()`.
- Querying Collections before putting anything in them (or vice versa).

⁵⁹<https://www.jetbrains.com/idea/help/accessing-inspection-settings.html>

- Accessing Collections as if they have elements of a different type (sadly possible due to the way Java implemented generics on collections).



Collections probable bugs

Solution

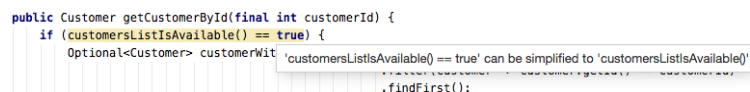
Not all of these problems are automatically bugs, but they do look suspicious. They'll usually require you, the code reviewer, to point them out to the author and have a discussion about whether this code is intentional.

Configuring

Inspections to highlight all the potential problems listed are already selected by default. To find more inspections in this category, look under Java > Probable Bugs in the [inspections settings](#)⁶⁰.

Code can be simplified

It's easy as you evolve code to end up with statements and methods that are more complicated than they need to be – it just takes one more bit of boolean logic or an additional `if` statement. As code reviewers, we're in a fortunate position of being one step back from the coal-face of the code, so we can call out areas ripe for simplification. Fortunately, we don't have to do this alone – Upsource shows us some of these things automatically.



Boolean expression can be simplified

Examples

- Using explicit `true` and `false` in a boolean expression (in the example above this is unnecessarily verbose).
- Boolean expressions that can be simplified, or re-phrased to be simpler to understand.
- `if` or `while` expressions that always evaluate to the same value:

⁶⁰<https://www.jetbrains.com/idea/help/accessing-inspection-settings.html>

```
if (this == o) {  
    return true;  
}  
if (o == null || getClass() != o.getClass()) {  
    return false;  
}  
  
SerializableObject that = (SerializableObject) o;  
  
if (someValue != that.someValue) {  
    return false;  
}  
  
return true;
```

Condition 'someValue != that.someValue' is always 'false'

Condition can be simplified

Solutions

- As with the other examples above, you may simply want to flag them in the code review so the author can use IntelliJ IDEA's inspections to apply the recommended fix.
- In some cases, like if statements that can be simplified in `equals()` methods, the simplified code is not always easier to read. If this is the case, you may want to suggest the code author suppresses the inspection for this code so it is no longer flagged.
- In other cases, the inspection might be pointing to a different smell. In the if statement above, the inspection shows this code (which is in a private class) is always called with a particular set of values so this if statement is redundant. It may be viable to remove the statement, but as this specific example is only used in test code it implies there's a missing test to show what happens when the two objects are equal. The code reviewer should suggest the additional test, or at least have the author document why it wasn't needed.

Configuring

These types of inspections can be found in **Java > Control flow issues** and **Java > Data flow issues**.

Unused Code

Upsource highlights all unused code (classes, methods, fields, parameters, variables) in a grey colour, so you don't even need to click or hover over the areas to figure out what's wrong with it – grey should automatically be a smell to a code reviewer.

```
public class OrdersService {  
    private OrdersDao ordersDao;  
    private CustomerRepository customerRepository;  
  
    public void placeOrder(BookOrder order) {  
        ordersDao.saveOrder(order);  
        Customer customer = customerRepository.get(order.getCustomerId());  
    }  
}
```

Variable 'customer' is never used

Unused code

Examples

There are a number of reasons a code review might contain unused code:

- It's an existing class/method/field/variable that has been unused for some time.
- It's an existing class/method/field/variable that is now unused due to the changes introduced in the code review.
- It's new / changed code that is not currently being called from anywhere.

Solutions

As a reviewer, you can check which category the code falls into and suggest steps to take:

- Delete the unused code. In the case of 1) or 2) above, this should usually be safe at the field/variable level, or private classes and methods. At the class and method level, if these are public they might be used by code outside your project. If you have control over all the code that would call these and you know the code is genuinely unused, you can safely remove them. In case 3) above, it's possible that some code is work-in-progress, or that the author changed direction during development and needs to clean up left over code – either way, flag the code and check if it can be deleted.
- Unused code could be a sign the author forgot to wire up some dependencies or call the new features from the appropriate place. If this is the case, the code author will need to fix the unused code by, well, using it.
- If the code is not safe to delete and is not ready to be used, then “unused code” is at the very least telling you that your test coverage is not sufficient. Methods and classes that are used by other systems, or will be used in the very near future, should have tests that show their expected behaviour. Granted, test coverage can hide genuinely unused code, but it's better to have code that looks used because it's tested than have code that is used that is not tested. As the reviewer, you need to flag the lack of tests. For code that existed before this code review, you might want to raise a task/story to create tests for the code rather than to slow down the current feature/bug being worked on with unrelated work. If the unused code is new code, then you can suggest suitable tests. New code that's untested **should not be let off lightly**⁶¹.

⁶¹<http://blog.jetbrains.com/idea/2015/08/why-write-automated-tests/>

- If you and the code author decide not to address the unused code immediately by deleting it, using it or writing tests for it, then at least document somehow why this code is unused. If there's a ticket/issue somewhere to address it later, refer to that.

```
3 + @SuppressWarnings("unused") //this is phase 1 of development. Code should be used when ISSUE-2574 is done.
4 public class CustomerOrders {
5     private int numberOfOrders;
6
7     public int getNumberOfOrders() {
8         return numberOfOrders;
9     }
10
11     public void setNumberOfOrders(int numberOfOrders) {
12         this.numberOfOrders = numberOfOrders;
13     }
14 }
```

Suppress unused warnings

Gotchas

Inspections are not infallible, hence why they're useful pointers for reviewers but not a fully automated check with a yes/no answer. Code might be incorrectly flagged as unused if:

- It's used via reflection
- It's used magically by a framework or code generation
- You're writing library code or APIs that are used by other systems

Configuring

These types of inspections can be found in **Java > Declaration redundancy**, **Java > Imports** and **Java > Probable bugs**. Or you can search for the string “unused” in the IntelliJ IDEA [inspection settings](#)⁶².

Summary

The navigation and inspection features are all available in the Upsource application. While it would be great if the app could provide everything we as developers want, sometimes we just feel more comfortable in the IDE. So that's why there's also an [Upsource plugin](#)⁶³ for IntelliJ IDEA and other JetBrains IDEs, so we can do the whole code review from within our IDE. There's also a new Open in IDE feature in [Upsource 2.5](#)⁶⁴ which, well, lets you open a code review in your IDE.

While many checks can and should be automated, and while humans are required to think about bigger-picture issues like design and “but did it actually fix the problem?”, there's also a grey area between the two. In this grey area, what we as code reviewers could benefit from is some guidance

⁶²<https://www.jetbrains.com/idea/help/accessing-inspection-settings.html>

⁶³<https://www.jetbrains.com/upsource/whatsnew/#plugin>

⁶⁴<http://blog.jetbrains.com/upsource/2015/09/30/upsource-2-5-early-access-is-open/>

about code that looks dodgy but might be OK. It seems logical that a code review tool should provide this guidance. Not only this, but we should also expect our code review tool to allow us to navigate through the code as naturally as we would in our IDE.

Upsource aims to make code review not only as painless as possible, but also provide as much help as a tool can, freeing you up to worry about the things that humans are really good at.

Upsource is free for an unlimited number of projects for teams with up to 10 developers, and provides a 60-day evaluation period for unlimited number of users. An evaluation key is available [upon request](#)⁶⁵. If you'd like to learn more about Upsource and how it can help you perform useful code reviews, visit our [page](#)⁶⁶.

⁶⁵<https://www.jetbrains.com/upsource/download/>

⁶⁶<http://jetbrains.com/upsource/>