

# Code Review #1

Majid Babaei



Code reviews act as *quality assurance* of the code base.

Software developers should be encouraged to have their code reviewed as soon as they've completed coding.

The reviewer can also act as a second step in *identifying bugs, logic problems, or uncovered edge cases*.



*An efficient code review process is very important in DevOps*

*Code Reviews facilitate knowledge sharing across the code base and across the team.*



# Code Review is your best tool!

- A properly conducted code review can do more for the *efficiency of your application* than any other steps in DevOps.
- A large numbers of *bugs can be found* and fixed before the code makes it into an official build or into the hands of the test team.
- The code review process is the way to *share best practices amongst a development team* and it produces 'lessons' that can be learned from in order to prevent future bugs.
- Code review is an *ongoing process* that, ideally, should occur *with every code check-in*.

# Why should we care if code is bad if it does the job?

---

**Poor readability:** A code which is not readable is a code that is more difficult to reuse, extend, and evolve.

**Low productivity:** A peer who is unfamiliar with the code will surely take more time to understand what the code is trying to achieve. Furthermore, you might end up spending more time fixing bugs rather than focusing on the core values added by the software you are implementing.

**Bug pollution:** Bugs can grow in size and number when not promptly addressed.

**Delays in releases:** Deadlines might be delayed because of bugs you ignored so far.

**Poor maintainability:** The code is difficult to read. Thus, the development process is slowed down by bugs. Furthermore, it is difficult to know how long it will take to implement new functionalities because of other defects in the system.

At a certain point, you might be even tempted to throw everything away and start from scratch. But please stop! That's not the right way, and there is a solution for this!

# Improving Quality with Code Reviews

---

**Informal reviews** are the more naïve way to check the code for defects. This kind of review usually involves no particular preparation nor planned metrics to measure the effectiveness of the review. They are usually performed by one or more peers, typically for brainstorming ideas.

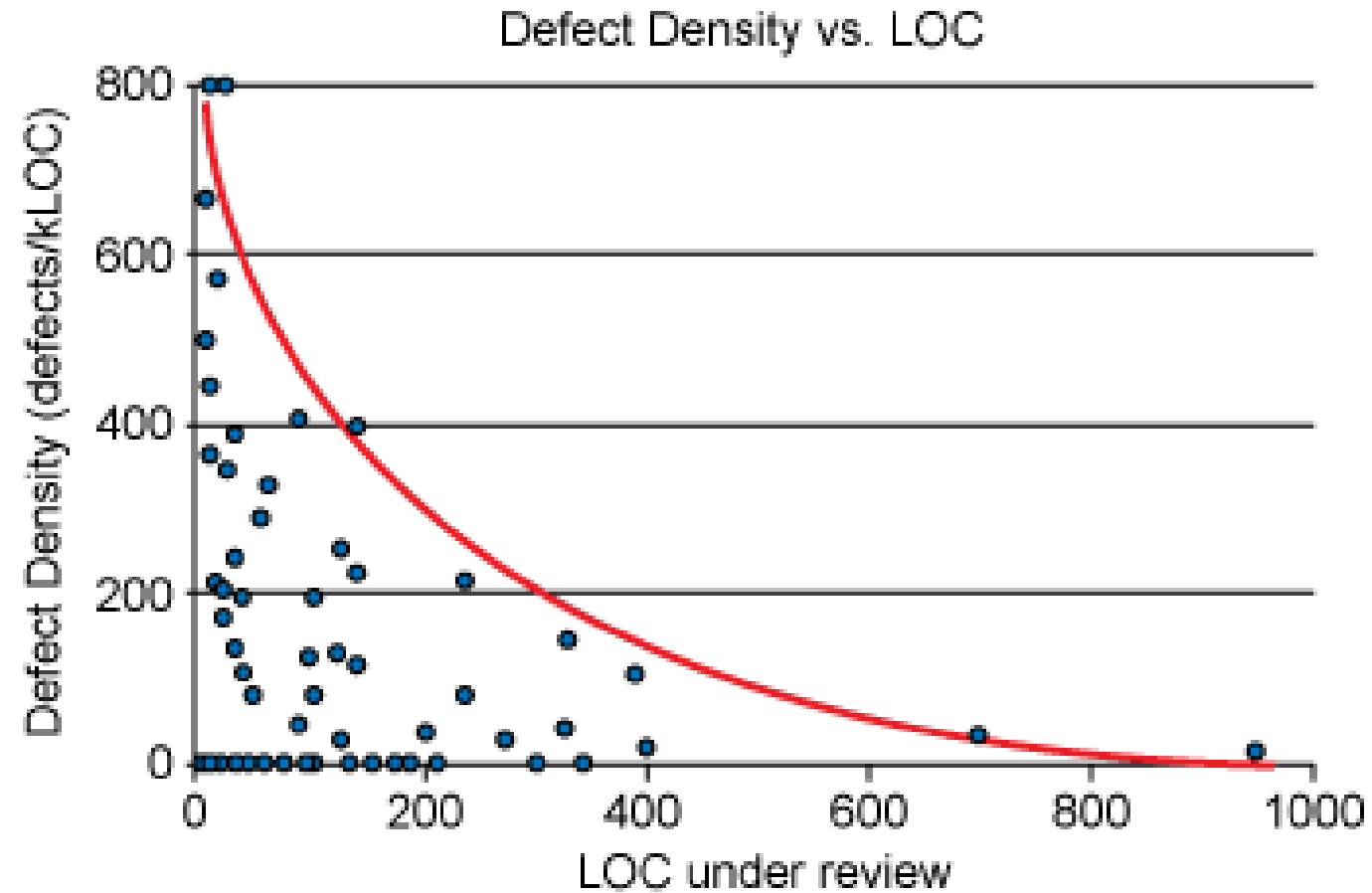
**Walkthrough reviews** are slightly more formal than informal reviews. They can be performed by a single person or by multiple participants. Defects are usually pointed out and discussed. This type of review is more about querying for feedback and comments from participants rather than actually correcting defects.

**Inspection reviews** are well planned and structured. It aims at finding and logging defects, gathering insights, and communicating them within the team. The process is supported by a checklist and metrics to gather the effectiveness of the review process. It is usually not performed by the author.

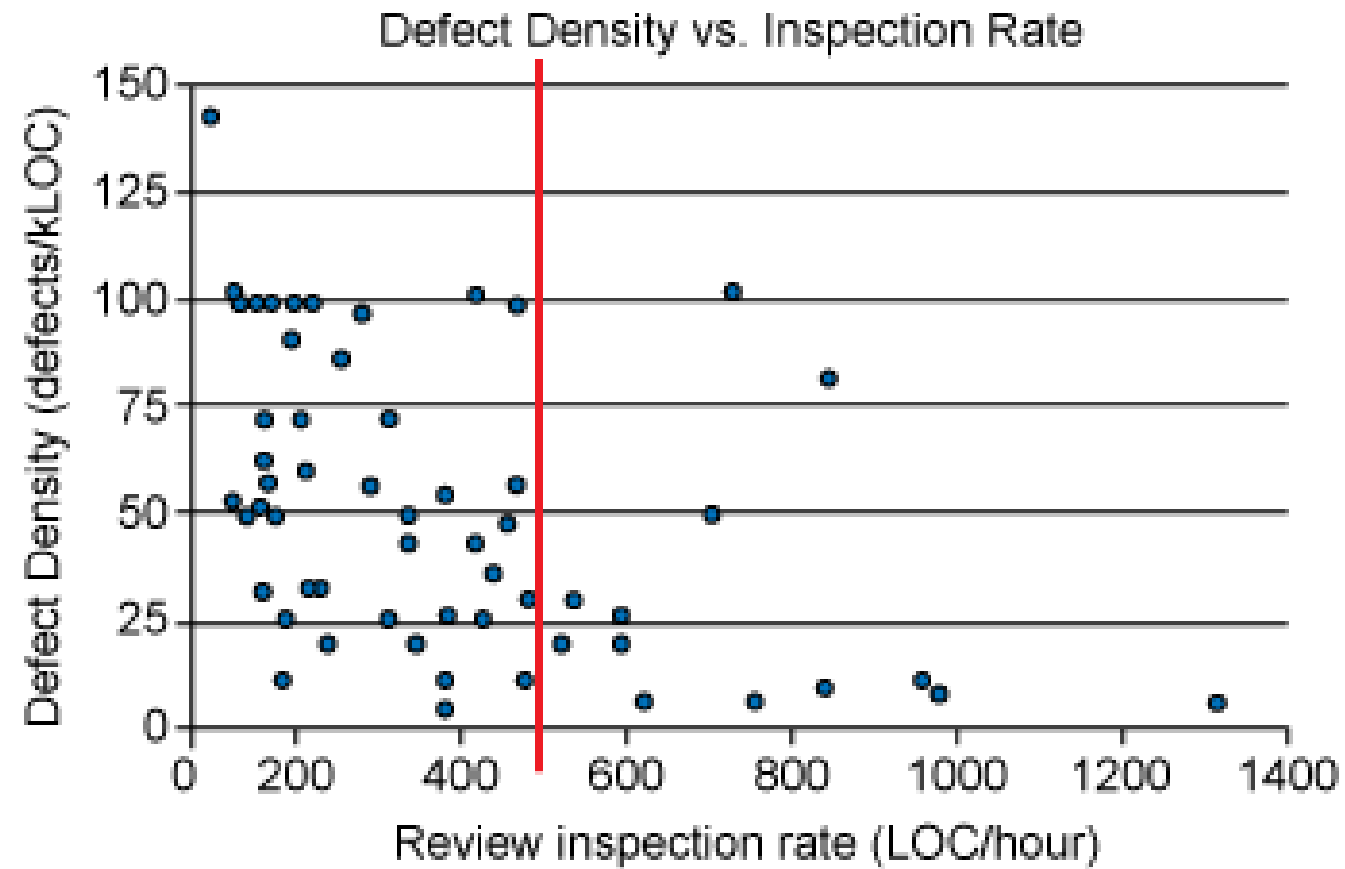
Code inspection is a broader review because it focuses on different aspects of code quality, from obvious logic errors including coding standards and style guidelines.



# Review fewer than 400 lines of code at a time



# Inspection rates should under 500 LOC per hour



# Design Smells

*Architectures that are not well designed and/or not properly maintained over time make new functionalities more difficult to develop.  
Furthermore, technical debt can quickly pile up in such scenario.*

# Cyclic Dependencies: *Description & Detection*

---

- happens when a component (namely, A) depends on another component (namely, B) to perform its duties, which in turn depends on it (i.e., A). The same behavior can be extended to classes, functions, methods, and interfaces.
- The number of dependencies within the code is referred to as *coupling*.
- Cycles of dependencies can span across several components before returning to the starting point. A better approach in this case is to take a look at the class diagram. Inspect it to see if it behaves like a directed acyclic graph (DAG).

```
1  # I am moduleA, nice to meet you
2  import moduleB
3  def print_val():
4      print('I am just an example.')
5  def calling_moduleB():
6      moduleB.module_functionality()
7
8  # I am moduleB, nice to meet you too
9  import moduleA
10 def module_functionality():
11     print ('I am doing nothing at all.')
12     moduleA.print_val()
13
14 # I am moduleC
15 import moduleA
16 def show_dependency():
17     print ('Such an interesting dependency.')
18     moduleA.calling_moduleB()
19
```

# Feature Density: *Description & Detection*

---

- This defect happens when a component implements more than a single functionality.
- Components have a lot of dependencies that are not clearly structured

For example, of an object depending on multiple other objects to perform its duties. However, instead of having structured dependencies, it uses and builds on top of them eventually performing functions that should have been implemented by the dependencies.

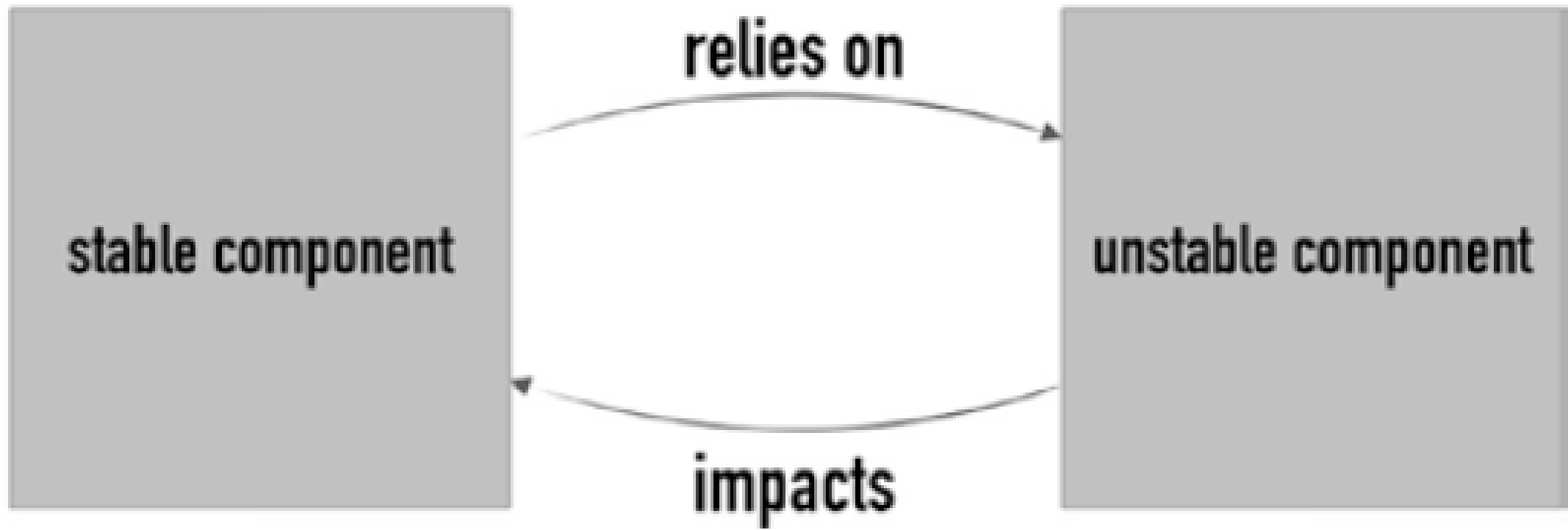
```
1  # This is a very smart virtual coffee machine
2  class CoffeeMachine:
3      def __init__(self, user):
4          self.user = user
5
6  def make_coffee(self, user):
7      self.gather_user_preferences()
8      print ('Work in progress. Wait a minute, please.')
9      ...
10
11 def gather_user_preferences(self):
12     preferred = user._preferred
13     milk = user._milk
14     print ('I am learning the type of coffee you like')
```

# Unstable Dependency: *Description & Detection*

---

- It happens if a component depends on a less stable one. This applies to any type of component (e.g., classes, methods)
- When a component depends on less stable ones, it will be easier to break the code.
- Detection strategy: Some tools and metrics are still under development or tuning.
- Try to find a more stable version of libraries used





# Mashed Components: *Description & Detection*

---

- As the opposite of the feature density defect, a mashed component is one that should logically be a single functionality, while scattered on different places on the code
- Detection strategy: take a look at diagrams (e.g., class diagrams) and inspect for highly coupled behavior. It might be the case of a mashed component smell.
- Once you can clearly state responsibilities for each and every component, think about the problems around the interactions they will have and pick communication patterns accordingly.

```
1  # UserManager - 1
2  class FirstUserManager:
3      def __init__(self, name, surname):
4          self._name = name
5          self._surname = surname
6          self._printer = SecondUserManager()
7
8      def print_user(self),
9          self._printer.printing(self._name, self._surname)
10
11  # UserManager - 2
12  class SecondUserManager:
13      def printing(self, name, surname):
14          print ('Yeah, I just printing on screen.')
15          ...
```

# Ambiguous Interfaces: *Description & Detection*

---

- APIs should be extensible and flexible, not unclear or ambiguous.

Ambiguity scenarios:

1. A component offers from a single entry point: e.g., *class provides a single public method that allows for actually accessing the class behavior.*

In real-world scenario: a website that needs to offer login, logout, and password retrieval functionality, and all these features are modeled with a **single general interface**.

# Ambiguous Interfaces: *Description & Detection*

---

Ambiguity scenarios:

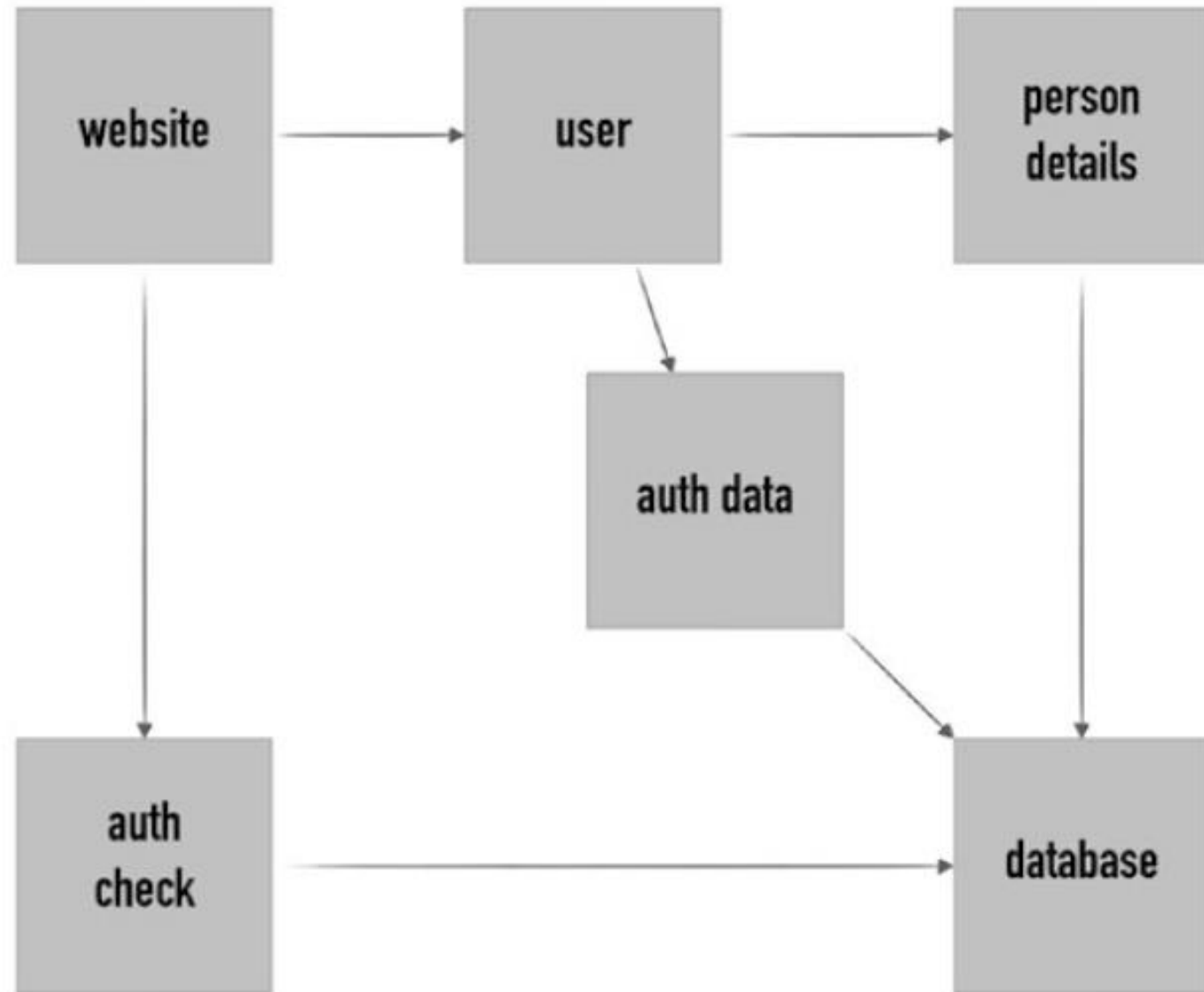
2. When multiple (i.e., more than one) signatures are too similar (e.g., same method name but with slightly different parameters). What can happen is that under certain scenarios, both of them might be applied, causing ambiguity in which one should be used.
- Detection strategy: Scan the code to figure out where interactions between components in the system are not clearly modeled.

```
1  # This is an example containing ambiguous interfaces
2  class Ambiguous:
3      def ambiguity(name:object, surname: str):
4          pass
5
6      @overload
7      def ambiguity(name:str, surname: object):
8          pass
9
10 # This is an example containing ambiguous interfaces
11 class Ambiguous:
12     def ambiguity(name:object, surname: object):
13         pass
14
15     @overload
16     def ambiguity(name:str, surname: str):
17         pass
```

# Mesh Component: *Description & Detection*

---

- This is the case where components, are heavily coupled, with a lot of dependencies and oftentimes without clear and well-defined patterns.
- The root cause might simply be a lack of abstraction and design patterns
- Detection strategy: Similar to the mashed components, to detect this smell, take a look at diagrams (e.g., class diagrams) and inspect for highly coupled behavior.





# First Lady Components (God component): *Description*

---

- This kind of component will do all the possible work by itself. It reaches such a growth that it becomes really expensive to maintain.
- It is similar to the feature density smell, but it is exponential in the lack of abstraction and of the negative impacts it has.
- It introduces possibly unneeded dependencies and well as a single point of failure

# First Lady Components (God component): *Detection*

---

- Classes, modules, and functions with a big number of lines of code (LOC) might be good candidates
- It is also as simple as reading package name, class name, and the methods
- It helps in spotting any behavior that does not belong to the given abstraction

A simple example of single lady is the scenario where there is a class that models the behavior of a user. This class, however, also implements general logging capabilities within the system. As an immediate effect, every other component that needs logging capability will have to depend on the user abstraction to access to logging.

# Bossy Component: *Description & Detection*

---

- At the opposite end of the first lady, this component likes to delegate stuff that it doesn't want to deal with to other components.
- It breaks the single responsibility principle - single means one, not none.
- Detection strategy: Inspect heavily coupled components. Is the caller only delegating logic to other components?
- Suggested refactoring: Rethink the functionalities and refactor accordingly. Check if this abstraction is really needed.

```
1  # I am the boss
2  class BossyComponent:
3      def do(self):
4          Mario.check_architecture()
5          Lisa.check_code()
6          Luigi.test()
7          self.something_left_for_me()
8
9      def something_left_for_me(self):
10         print ('I am the boss, I am doing nothing. Hee Hee. ')
11
12  class Mario:
13      @staticmethod
14      def check_architecture():
15         print ('Ok')
16
17  class Lisa:
18      @staticmethod
19      def check_code():
20         print ('Ok')
21
22  class Luigi:
23      @staticmethod
24      def test():
25         print ('Ok')
```

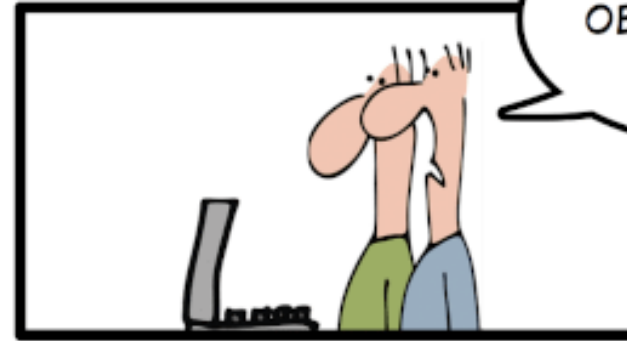
# Principles

---

## *HOW TO MAKE A GOOD CODE REVIEW*



geek & poke



AT LEAST WE  
DON'T NEED TO  
OBFUSCATE IT  
BEFORE  
SHIPPING

*RULE 1: TRY TO FIND  
AT LEAST SOMETHING  
POSITIVE*

- ***CRITIQUE THE CODE (NOT THE PERSON)***
- They are not their code!
- The code that someone submits should not reflect upon them as a person. Code reviews should *never make it personal*.
- It should be about the problem we're trying to solve, and whether this particular version of the change is the best way to do it.

- DON'T ASSUME IT'S OBVIOUS
- CODE CHANGES AND FEEDBACK BOTH NEED EXPLANATION
- Many, many important things get omitted in communication because someone assumed, "this is obvious, I don't need to say it."
- Always *state your assumptions*, and overcommunicate.
- Code authors, *explain what you're doing* and *why*!
- Code reviewers, *explain the feedback* you're giving, and *why* it will help!

# The process

---

- You can approach code reviewing in stages *using “outside-in” mindset*.
- By *starting at the highest level*, I can short-circuit my review if I reveal anything that might *require a major change* to the pull request.
- There's no point in reviewing each line of code if the architecture is all wrong!




# Intention


---

- Let's *start with the intention behind the change*, and make sure we agree on what we want this PR to actually accomplish.
- WHAT IS THE GOAL?
- IS THE EXPLANATION CLEAR?
- Every single pull request summary should explain the goal of the PR. Otherwise, how will you as a reviewer evaluate it?
- *The author of a pull request will have the most context on the problem*, so it's important that they can explain their goal clearly. And if the explanation doesn't make sense to you as a reviewer, *request changes*!

# Here's an informative pull request

## Implement `jest-haste-map` instead of `node-haste` #896

 Closed

cpojer wants to merge 10 commits into `facebook:master` from `cpojer:jest-haste-map` 

 Conversation 93

 Commits 10

 Checks 0

 Files changed 47



cpojer commented on Apr 15, 2016 • edited ▼

Collaborator



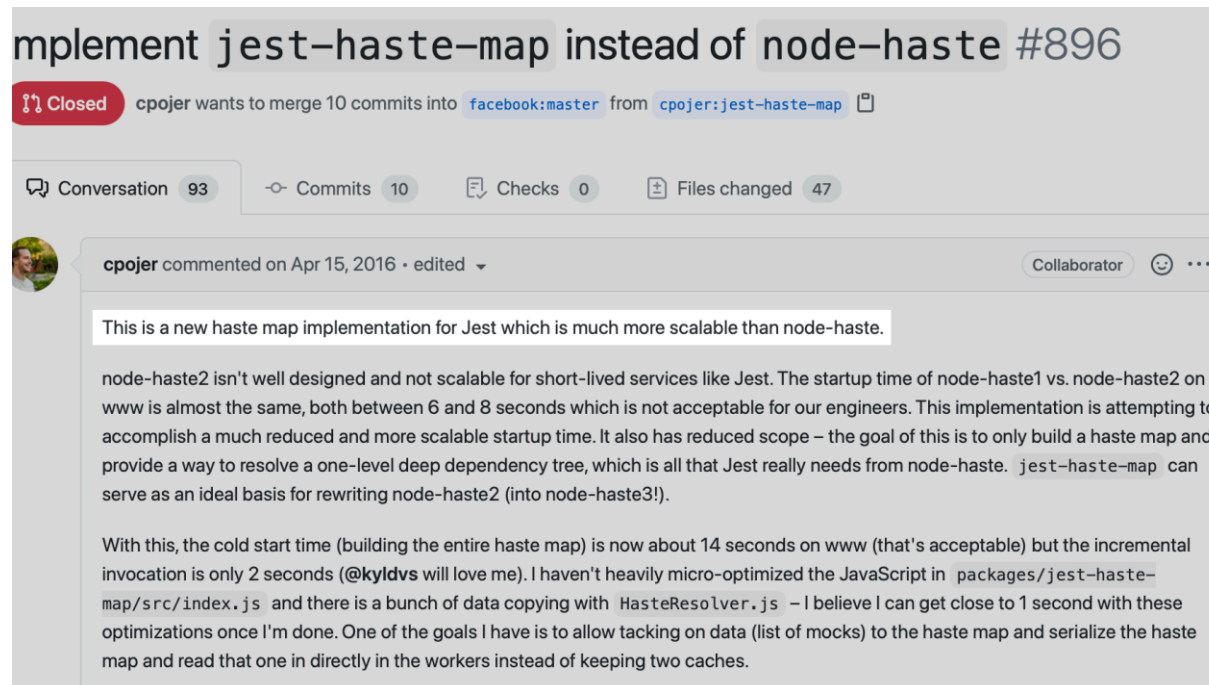
This is a new haste map implementation for Jest which is much more scalable than `node-haste`.

`node-haste2` isn't well designed and not scalable for short-lived services like Jest. The startup time of `node-haste1` vs. `node-haste2` on `www` is almost the same, both between 6 and 8 seconds which is not acceptable for our engineers. This implementation is attempting to accomplish a much reduced and more scalable startup time. It also has reduced scope – the goal of this is to only build a haste map and provide a way to resolve a one-level deep dependency tree, which is all that Jest really needs from `node-haste`. `jest-haste-map` can serve as an ideal basis for rewriting `node-haste2` (into `node-haste3`!).

With this, the cold start time (building the entire haste map) is now about 14 seconds on `www` (that's acceptable) but the incremental invocation is only 2 seconds (@[kyldvs](#) will love me). I haven't heavily micro-optimized the JavaScript in `packages/jest-haste-map/src/index.js` and there is a bunch of data copying with `HasteResolver.js` – I believe I can get close to 1 second with these optimizations once I'm done. One of the goals I have is to allow tacking on data (list of mocks) to the haste map and serialize the haste map and read that one in directly in the workers instead of keeping two caches.

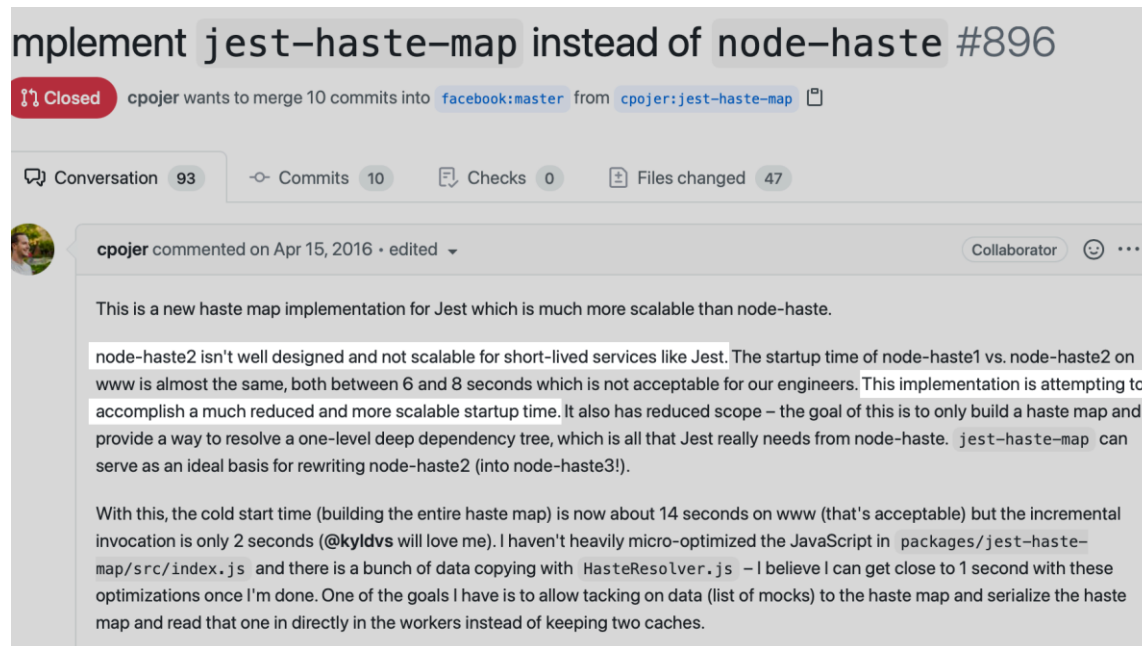
# What are the nice aspects of it?

- The very *first sentence gives you the goal right away*—to introduce "a new haste map implementation ... which is much more scalable than node-haste."
- *There's some assumed context here*, but in this case, it refers to something all of the reviewers will be familiar with.



# What are the nice aspects of it?

- Then, the summary elaborates on *why that goal is important*. The existing implementation "isn't well designed and not scalable." "This implementation is attempting to [reduce] startup time."
- Reviewers now have enough information to *evaluate whether the goal is worth achieving* (i.e., if this is the right problem to solve).



# The next step



# SUCCEED

# Does it succeed?

---

- Presuming you, as the reviewer, understand the goal of the PR, your next responsibility is to determine whether this pull request actually achieves it.
- These are a couple of the questions you (the reviewer) should have, and the *author should have satisfying answers for you* right there in the PR summary.
- For example:
  - Is the bug fix or new feature tested?
  - How will you know if there's a regression?

# Another example

## Initial version of jest-worker #4497



Merged

cpojer merged 8 commits into `facebook:master` from `mjesun:jest-parallel` on Oct 4, 2017



Conversation 85



Commits 8



Checks 0



Files changed 14



mjesun commented on Sep 17, 2017 • edited ▾

Contributor



This PR introduces a new module, `jest-worker`, intended to allow heavy task parallelization over multiple workers.

The module has a few advantages over the currently one used both in `jest` and `metro-bundler`:

- 100% `flow`-ified.
- 100% test coverage on it, all statements, methods and branches.
- Slightly faster than the currently used one.
- Natively provides a `Promise` based interface, which allow us to avoid the extra wrapping layer in order to be used with `async / `await`.
- It only has one single dependency (`merge-stream`), which we could also remove.



# Is there any evidences?

## Performance test

It can be run by doing `node --expose-gc test.js` under `__performance_tests__`. Note that the percentage improvement shown (~ 10%) applies to 10,000 calls, meaning the performance improvement per single call is negligible. The test implements a `Promise` wrapper over the current implementation, so we can equivalently test both implementations as we use them in real scenarios.

```
-----  
jest-worker: { globalTime: 738, processingTime: 707 }  
worker-farm: { globalTime: 885, processingTime: 866 }
```

```
-----  
jest-worker: { globalTime: 738, processingTime: 718 }  
worker-farm: { globalTime: 865, processingTime: 849 }
```





```
-----  
jest-worker: { globalTime: 708, processingTime: 685 }
```

**And we have *evidence* that it succeeds at improving performance, because the PR helpfully includes benchmarking results right in the description—along with the process by which anyone can run their own benchmarks.**



# Any regression has happened?

## Coverage

File ▲		Statements ▾		Branches ▾		Functions ▾		Lines ▾	
<a href="#">child.js</a>		100%	27/27	100%	20/20	100%	4/4	100%	27/27
<a href="#">index.js</a>		100%	67/67	100%	29/29	100%	11/11	100%	67/67
<a href="#">types.js</a>		100%	5/5	100%	0/0	100%	0/0	100%	5/5
<a href="#">worker.js</a>		100%	44/44	100%	13/13	100%	8/8	100%	44/44

**We have evidence that despite the performance improvement, no functionality has regressed.**