Code Review

Jieung Kim

Aug. 2022

Code review

Careful and systematic study of source code by people who are not the original author of the code

It’s analogous to proofreading a term paper

Code review

Code reviews should look at:

- **Design** - Is the code well-designed and appropriate for your system?
- **Functionality** - Does the code behave as the author likely intended? Is the way the code behaves good for its users?
- **Complexity** - Could the code be made simpler? Would another developer be able to easily understand and use this code when they come across it in the future?
- **Tests** - Does the code have correct and well-designed automated tests?
- **Naming** - Did the developer choose clear names for variables, classes, methods, etc.?
- **Comments** - Are the comments clear and useful?
- **Style** - Does the code follow our style guides?
- **Documentation** - Did the developer also update relevant documentation?

Again, each project or company has its own guideline that developers have to follow (e.g., Google Python style guide)

Motivation for reviews

- Can catch many bugs, design flaws early
- > 1 person has seen every piece of code
  - Insurance against author's disappearance
  - Accountability (both author and reviewers are accountable)
- Forcing function for documentation and code improvements
  - Authors to articulate their decisions
  - Authors participate in the discovery of flaws
  - Prospect of someone reviewing your code raises quality threshold
- Inexperienced personnel get hands-on experience without hurting code quality
  - Pairing them up with experienced developers
  - Can learn by being a reviewer as well
    - Google has a readability reviewer class for Google software engineers

Motivation by the numbers

From Steve McConnel's Code Complete

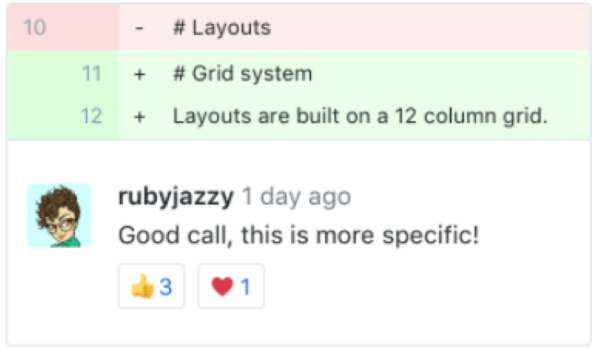
- Average defect detection rates
  - Unit testing: 25%
  - Function testing: 35%
  - Integration testing: 45%
  - Design and code inspections: 55% and 60%
- 11 programs developed by the same group of people
  - First 5 without reviews: average 4.5 errors per 100 lines of code
  - Remaining 6 with reviews: average 0.82 errors per 100 lines of code
  - Errors reduced by > 80%
- After AT&T introduced reviews, 14% increase in productivity and a 90% decrease in defects

Code review in Industry

- Code reviews are a very common industry practice
- Made easier by advanced tools that
  - integrate with configuration management systems
  - highlight changes (i.e., diff function)
  - allow traversing back into history
- Google also provide integrated code review tool with our development tools

Code review in industry

1. Barkeep
2. Cahoots
3. CodeReviewHub
4. CodeReviewer
5. Codebrag
6. Codifferous
7. Collaborator
8. Crucible
9. Differential
10. Exercism
11. Gerrit
12. Gitcolony
13. **Github code review**
14. Kallithea
15. Malevich
16. Redmine + Code Review Plugin
17. Review Board
18. Reviewable
19. Reviewlead
20. Rietveld
21. TeamReview
22. Upsource

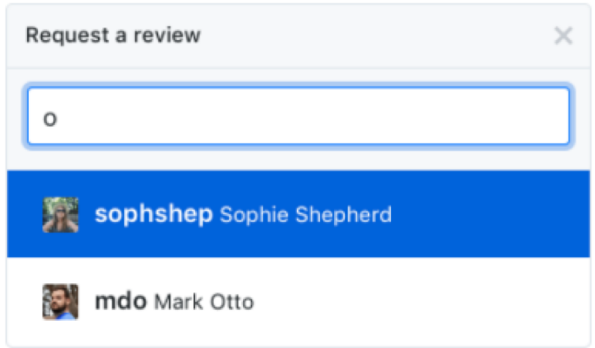


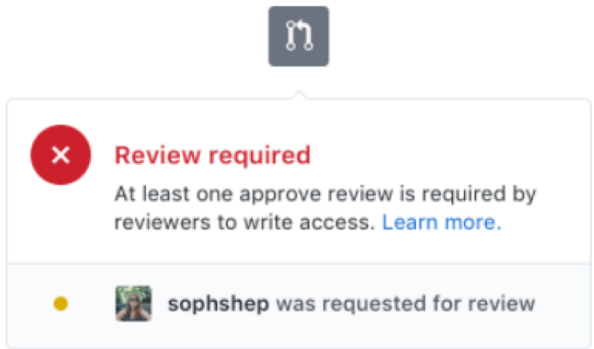
Comments

On GitHub, conversations happen alongside your code. Leave detailed comments on code syntax and ask questions about structure inline.

Review requests

If you're on the other side of the code, requesting peer reviews is easy. Add users to your pull request, and they'll receive a notification letting them know you need their feedback.





Reviews

Save your teammates a few notifications. Bundle your comments into one cohesive review, then specify whether comments are required changes or just suggestions.

Code review at Yelp

"At Yelp we use **a review-board**. An **engineer** works on a branch and **commits** the code to their own branch. The **reviewer** then goes through the diff, **adds inline comments** on the review board and sends them back. The reviews are meant to be a dialogue, so typically comment threads result from the feedback. Once the reviewer's **questions and concerns are all addressed** they'll click '**Ship It**!' and the author will merge it with the main branch for deployment the same day."

- Alan Fineberg, Software Engineer, Yelp

Code review at Facebook

"At Facebook, we have an **internally-developed web-based tool** to **aid the code review process**. Once an **engineer** has **prepared a change**, she submits it to this tool, which will **notify the person or people** she has **asked to review** the change, along with others that may be interested in the change -- such as people who have worked on a function that got changed."

"At this point, **the reviewers** can make **comments, ask questions, request changes, or accept the changes**. If changes are requested, the submitter must submit a new version of the change to be reviewed. All versions submitted are retained, so reviewers can compare the change to the original, or just changes from the last version they reviewed. **Once a change has been submitted**, the **engineer can merge her change into the main source tree** for deployment to the site during the next weekly push, or earlier if the change warrants quicker release."

- Ryan McElroy, Software Engineer, Facebook

Code review – how can we do that?

- We have an official style guide of this sort
- Strictly enforcing personal style in larger projects is prohibited (DON'T DO THAT!)
- There are some rules that are quite sensible, removing "bad s mells"

“Bad smells”

The coding patterns and approaches that lack expandability/flexibility and can hinder future code development

“Bad smells”

There are tons of bad smells. But the following things are frequent and easy to get rid of:

- **Duplicated code** is a maintenance nightmare. Extract it into a function that you can call where needed
- **Magic numbers and strings** don’t tell the reader what they mean. Use named constants instead
- **Dead code** makes your code seem more complex than it is
- **Redundant comments** hurt readability, just remove them

Other things to add “good smells”

- Some common guidelines are as follows.
  - The rule of the three.
    - If a class defines one (or more) of the following, it should explicitly define all three, which are:
      - destructor
      - copy constructor
      - copy assignment operator
  - Do not use #define unless you have to.
  - Try to use **const** member functions and variables.
  - Set up the criteria on class, function, field, and variable names.
  - Locate functions in proper classes.
  - Try to use initializer list.
  - Use iteration over STL containers.
  - Use reference when it is possible.
  - …

```
#define Fresh 1
#define Sophomore 2
#define Junior 3
#define Senior 4

class Student {
    public:
    Student(int id, int year) {
        student_id = id;
        student_year = year;
    };

    ~Student();

    int GetStudentID() { return student_id; }
    int get_student_year() { return student_year; }

    private:
    int student_id;
    int student_year;
};


bool FindStudent(int id, std::vector<Student> students) {
    for (int i = 0; i < students.size(); i++) {
        if (students[i].GetStudentID() == id) {
            return true;
        }
    }
    return false;
}
```

```
#define Fresh 1
#define Sophomore 2
#define Junior 3
#define Senior 4

class Student {
  public:
  Student(int id, int year) {
    student_id = id;
    student_year = year;
  };

  ~Student();

  int GetStudentID() { return student_id; }
  int get_student_year() { return student_year; }

  private:
  int student_id;
  int student_year;
};


bool FindStudent(int id, std::vector<Student> students) {
  for (int i = 0; i < students.size(); i++) {
    if (students[i].GetStudentID() == id) {
      return true;
    }
  }
  return false;
}
```

Do not use #define

Violate the rule of t he three

Initializer list is not used

Inconsistency in fu nction names

Fields are not distinguishable from local variables

References should be used

Not proper functio n location (debatable)

Iterator is not used

```
class Student {
  public:
  enum StudentYear { FRESH = 1, Sophomore, Junior, Senior };

  Student(const int id, const StudentYear year) :
    id_(id), year_(year) {};
  Student(const Student& student) :
    id_(student.id_), year_(student.year_) {};
  Student& operator=(const Student& student) {
    if (this != &student) {
      *this = Student(student);
    }
    return *this;
  };
  ~Student();

  int GetId() { return id_; }
  int GetYear() { return year_; }

  bool FindStudent(const int id,
    const std::vector<Student>& students) const {
    for (const auto& student : students) {
      if (student.id_ == id) {
        return true;
      }
    }
    return false;
  }

  private:
  const int id_;
  StudentYear year_;
};
```

Prep

- Please rewrite the following code during 25 minutes with you r teammates.
- Share the rewritten code with us (5 minutes per one group).

```
int STANDARD=0, BUDGET=1, PREMIUM=2, PREMIUM_PLUS=3;

class Account {
  public:
  double principal,rate; int daysActive,accountType;
};

double calculateFee(std::vector<Account> accounts)
{
  double totalFee = 0.0;
  Account account;
  for (int i=0;i<accounts.size();i++) {
    account=accounts[i];
    if ( account.accountType == PREMIUM ||
      account.accountType == PREMIUM_PLUS )
      totalFee += .0125 * (            // 1.25% broker's fee
        account.principal * pow(account.rate,
        (account.daysActive/365.25))
        - account.principal);   // interest-principal
  }
  return totalFee;
}
```

```
// An individual account.  Also see XXX.
class Account {
  public:
  // Constructor with default value.
  // It is better for us to provide those default values with config files.
  Account() :
  broker_fee_percent_(0.0125),
  days_per_year_(365.25) {};

  // Constructor with user-defined broker fee percent and days per year.
  Account(double broker_fee_percent, double days_per_year) :
  broker_fee_percent_(broker_fee_percent),
  days_per_year_(days_per_year) {};

  // Copy constructor
  Account(const Account& account) :
  broker_fee_percent_(account.broker_fee_percent_),
  days_per_year_(account.days_per_year_),
  principal_(account.principal_),
  rate_(account.rate_),
  days_active_(account.days_active_),
  account_type_(account.account_type_) {};

  Account& operator=(const Account& account) {
    if (this != &account) {
      *this = Account(account);
    }
    return *this;
  };
  ~Account();
```

```
// The varieties of account our bank offers.
enum AccountType {STANDARD, BUDGET, PREMIUM, PREMIUM_PLUS};

// Compute interest.
double GetInterest() {
  double years = days_active_ / days_per_year_;
  double compound_interest = principal_ * pow(rate_, years);
  return compound_interest - principal_;
}

// Return true if this is a premium account.
bool IsPremium() {
  return account_type_ == AccountType::PREMIUM ||
    account_type_ == AccountType::PREMIUM_PLUS;
}

// Return the sum of the broker fees for all the given accounts.
double CalculateFee(std::vector<Account>& accounts) {
  double total_fee = 0.0;
  for (Account account : accounts) {
    if (IsPremium()) {
      total_fee += broker_fee_percent_ * GetInterest();
    }
  }
  return total_fee;
}
```

```
  private:
  // The portion of the interest that goes to the broker.
  const double broker_fee_percent_;
  // The number of days per one year.
  const double days_per_year_;
  double principal_;
  // The yearly, compounded rate (at 365.25 days per year).
  double rate_;
  // Days since last interest payout.
  int days_active_;
  // Account type.
  AccountType account_type_;
};
```

Code review

- One hour talk about C++ code smells (bad anFor more code smells, look at "C++ Code Smells –Jason Turner".
- d good smells).

- The slide is also available at the CppCon2019 repo (slide).
  - CppCon is the annual, week-long online face-to-face gathering for t he entire C++ community.