SOFTWARE ENGINEERING

CO3001

CHAPTER 8 – SOFTWARE IMPLEMENTATION

Anh Nguyen-Duc
Tho Quan-Thanh

WEEK 8

Adapted from https://iansommerville.com/software-engineering-book/slides/

TOPICS COVERED

- ✓ Implementation meaning
- ✓ Coding style & standards
- ✓ Code with correctness justification
- ✓ Code smell and some good practices
- ✓ Code inspection
- ✓ Exercise

IMPLEMENTATION

smallest part that will be separately maintained

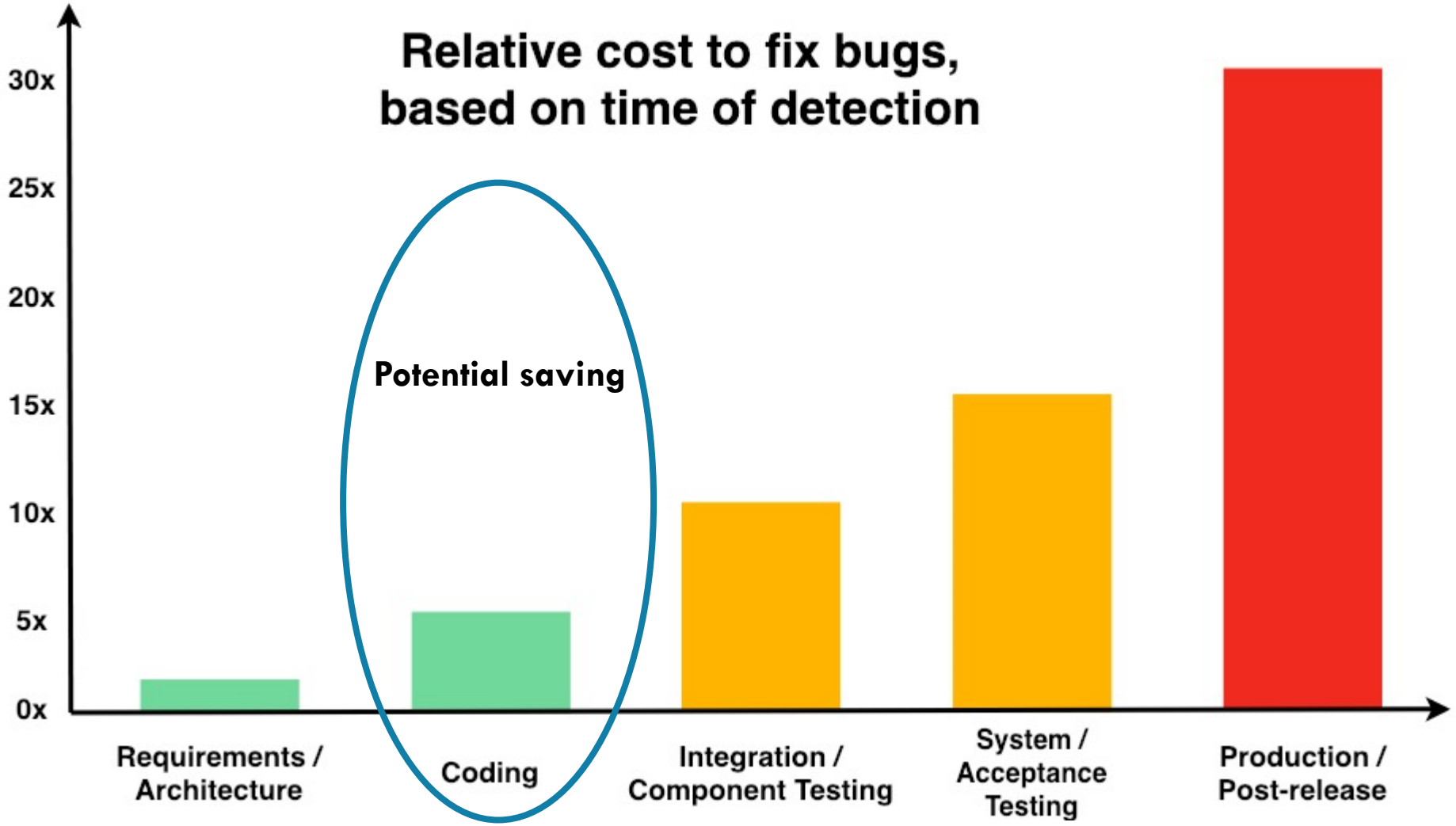
- ✓ Implementation = Unit Implementation + Integration

put them all together

GOLDEN RULE (!?)

- ✓ Requirements to satisfy Customers
- ✓ Design again requirements only
- ✓ Implement again design only
- ✓ Test again design and requirements

CODE QUALITY



IMPLEMENT CODE

- ✓ 1. Plan the structure and residual design for your code
- ✓ 2. Self-inspect your design and/or structure
- ✓ 3. Type your code
- ✓ 4. Self-inspect your code
- ✓ 5. Compile your code
- ✓ 6. Test your code

GENERAL PRINCIPLES IN PROGRAMMING PRACTICE

- ✓ 1. Try to re-use first
- ✓ 2. Enforce intentions
  - If your code is intended to be used in particular ways only, write it so that the code cannot be used in any other way.
  - If a member is not intended to be used by other functions, enforce this by making it private or protected etc.
  - Use qualifiers such as final and abstract etc. to enforce intentions

“THINK GLOBALLY, PROGRAM LOCALLY”

- ✓ Make all members
  - as local as possible
  - as invisible as possible
    - attributes private:
      - access them through more public accessor functions if required.
      - (Making attributes protected gives objects of subclasses access to members of their base classes -- not usually what you want)

EXCEPTIONS HANDLING (ADDITIONAL READING)

"If you must choice between throwing an exception and continuing the computation, continue if you can"
(Cay Horstmann)

- ✓ Catch only those exceptions that you know how to handle
- ✓ Be reasonable about exceptions callers must handle
- ✓ Don't substitute the use of exceptions for issue that should be the subject of testing

NAMING CONVENTIONS (ADDITIONAL READING)

- ✓ Use concatenated words
  - e.g., cylinderLength
- ✓ Begin class names with capitals
- ✓ Variable names begin lower case
- ✓ Constants with capitals
  - as in MAX_N or use static final
- ✓ Data members of classes with an underscore
  - as in _timeOfDay
- ✓ Use get…, set…., and is… for accessor methods
- ✓ Additional getters and setters of collections
- ✓ And/or distinguish between instance variables, local variables and parameters

DOCUMENTING METHODS (ADDITIONAL READING)

- ✓ what the method does
- ✓ why it does so
- ✓ what parameters it must be passed (use @param tag)
- ✓ exceptions it throws (use @exception tag)
- ✓ reason for choice of visibility
- ✓ known bugs
- ✓ test description, describing whether the method has been tested, and the location of its test script
- ✓ history of changes if you are not using a sub-version system
- ✓ example of how the method works
- ✓ pre- and post-conditions
- ✓ special documentation on threaded and synchronized methods

```
/* Class Name          : EncounterCast
* Version information  : Version 0.1
* Date                 : 6/19/1999
* Copyright Notice     : see below
* Edit history:
*   11 Feb
*    8 Feb
*   08 Jan
*/

/*
   Copyrig

   This pro
   "Softwa
   by Eric
```

```
/** Facade class/object for the EncounterCharacters package. Used to
* reference all characters of the Encounter game.
* <p> Design: SDD 3.1 Module decomposition
* <br> SDD 5.1.2 Interface to the EncounterCharacters package
*
* <p>Design issues:<ul>
* <li> SDD 5.1.2.4 method engagePlayerWithForeignCharacter was
*   not implemented, since engagements are handled more directly
*   from the Engaging state object.
* </u
*
* @a
* @v
*/
public
{
   /** Name for human player */
   private static final String MAIN_PLAYER_NAME = "Elena";
```

```
/** Gets encounterCast, the one and only instance of EncounterCast.
   * <p> Requirement: SDD 5.1.2
   *
   * @return      The EncounterCast singleton.
   */
   public static EncounterCast getEncounterCast()
      { return encounterCastS; }
```

DOCUMENTING ATTRIBUTES

- ✓ Description -- what it's used for
- ✓ All applicable invariants
  - quantitative facts about the attribute,
    - such as "1 < _age < 130"
    - or " 36 < _length * _width < 193".

CONSTANTS

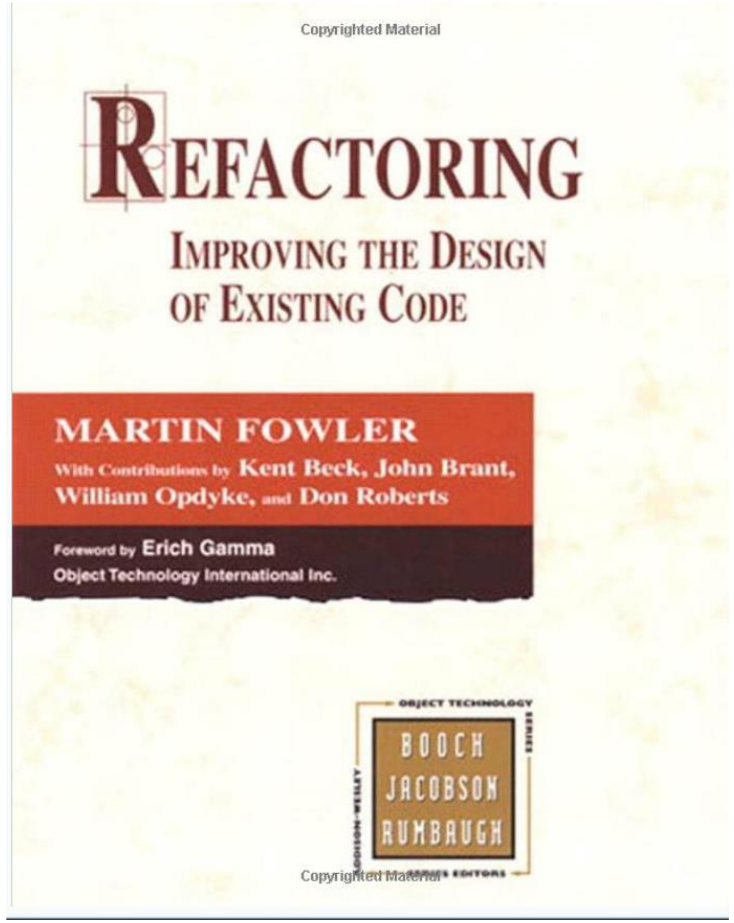
- Before designating a final variable, be sure that it is, indeed, final. You're going to want to change "final" quantities in most cases. Consider using method instead.
- Ex:
  - instead of ...
  - protected static final MAX_CHARS_IN_NAME;

  - consider using ...
  - protected final static int getMaxCharsInName()
  - { return 20;
  - }

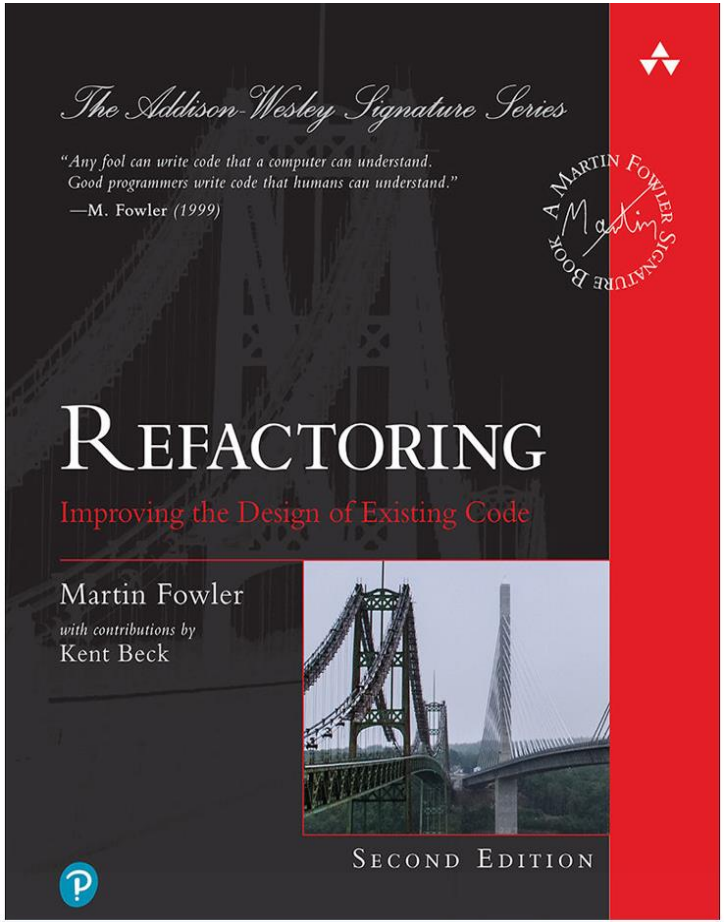
INITIALIZING ATTRIBUTES

- ✓ Attributes should be always be initialized, think of
  - private float _balance = 0;
- ✓ Attribute may be an object of another class, as in
  - private Customer _customer;
- ✓ Traditionally done using the constructor, as in
  - private Customer _customer = new Customer("Edward", "Jones");
- ✓ Problem is maintainability. When new attributes added to Customer, all have to be updated. Also accessing persistent storage unnecessarily.

RELEVANT BOOKS (ADDITIONAL READING)



Old edition, available online at NTNU lib.



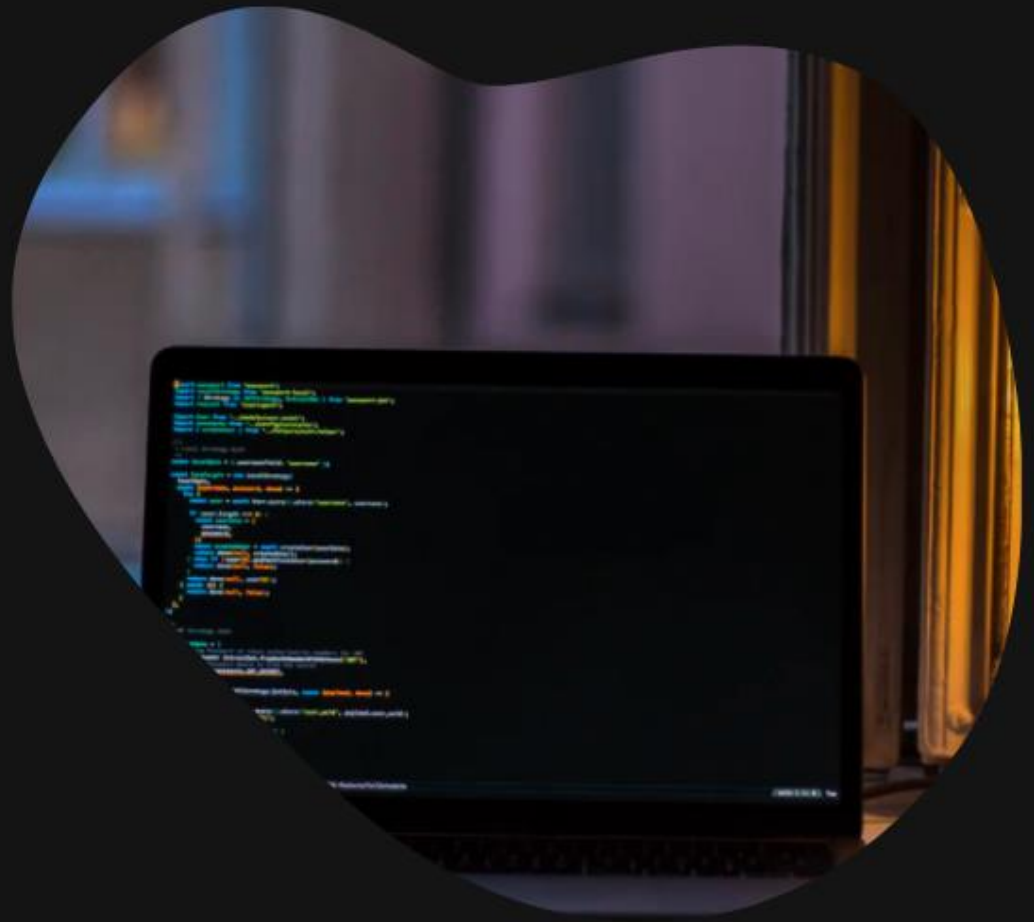
New edition

BAD SMELLS IN CODE (CODE SMELLS)

- ✓ Code smells are any violation of fundamental design principles that decrease the overall quality of the code.
- ✓ Not bugs or errors
- ✓ Can certainly add to the chance of bugs and failures down the line.

COD REFACTORING GOALS AND PROPERTIES

- ✓ Change the internal structure without changing external behavior
- ✓ Eliminate code smells for
  - Readability
  - Consistency
  - Maintainability
- ✓ Properties
  - Preserve correctness
  - One step at a time
  - Frequent testing



Code Smells

That Are Found The Most

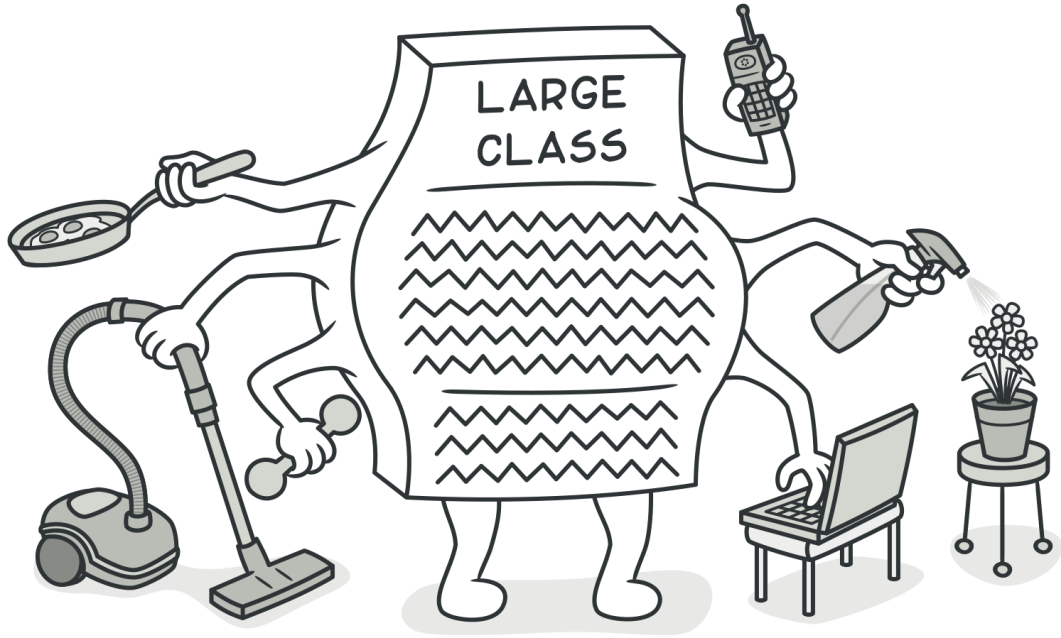
[redacted]

NTNU

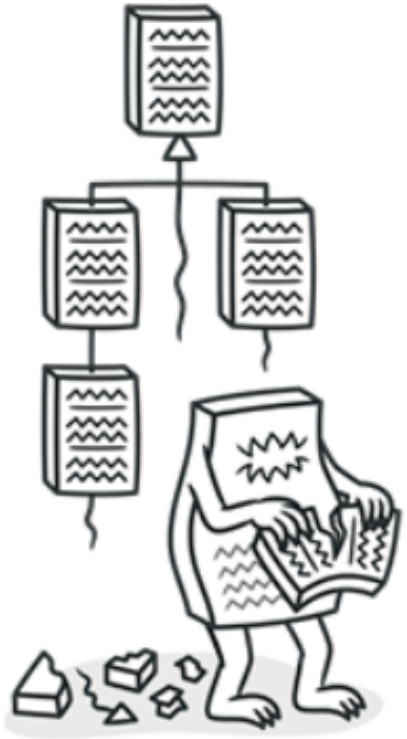
Bloaters

Bloaters are code, methods and classes that have increased to such gargantuan proportions that they are hard to work with. Usually these smells do not crop up right away, rather they accumulate over time as the program evolves (and especially when nobody makes an effort to eradicate them).

- § Long Method
- § Large Class
- § Primitive Obsession
- § Long Parameter List
- § Data Clumps



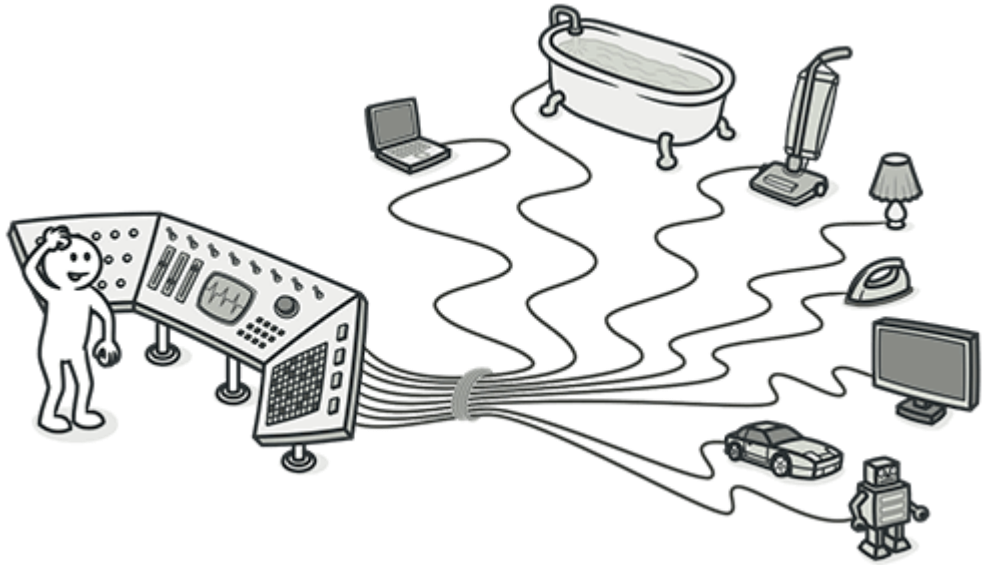
NTNU



Object-Orientation Abusers

All these smells are incomplete or incorrect application of object-oriented programming principles.

- Alternative Classes with Different Interfaces
- Refused Bequest
- Switch Statements
- Temporary Field



NTNU

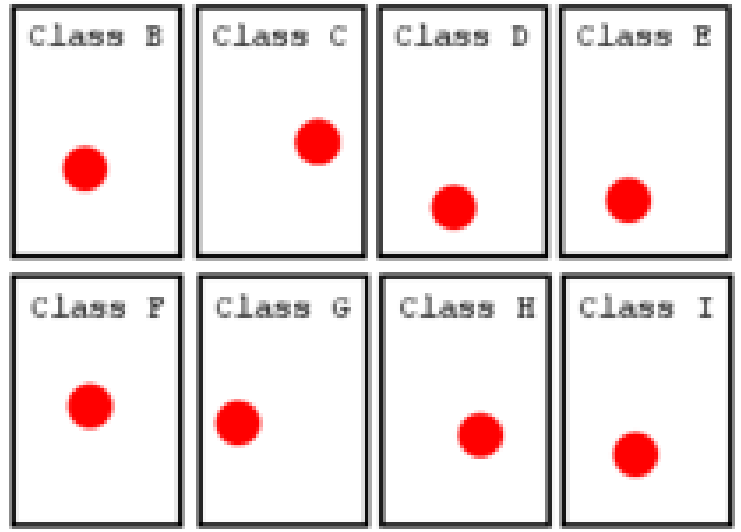
Change Preventers

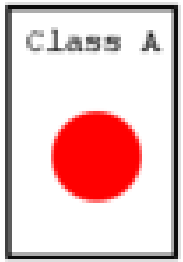
These smells mean that if you need to change something in one place in your code, you have to make many changes in other places too. Program development becomes much more complicated and expensive as a result.

§ **Divergent Change**

§ **Parallel Inheritance Hierarchies**

§ **Shotgun Surgery**



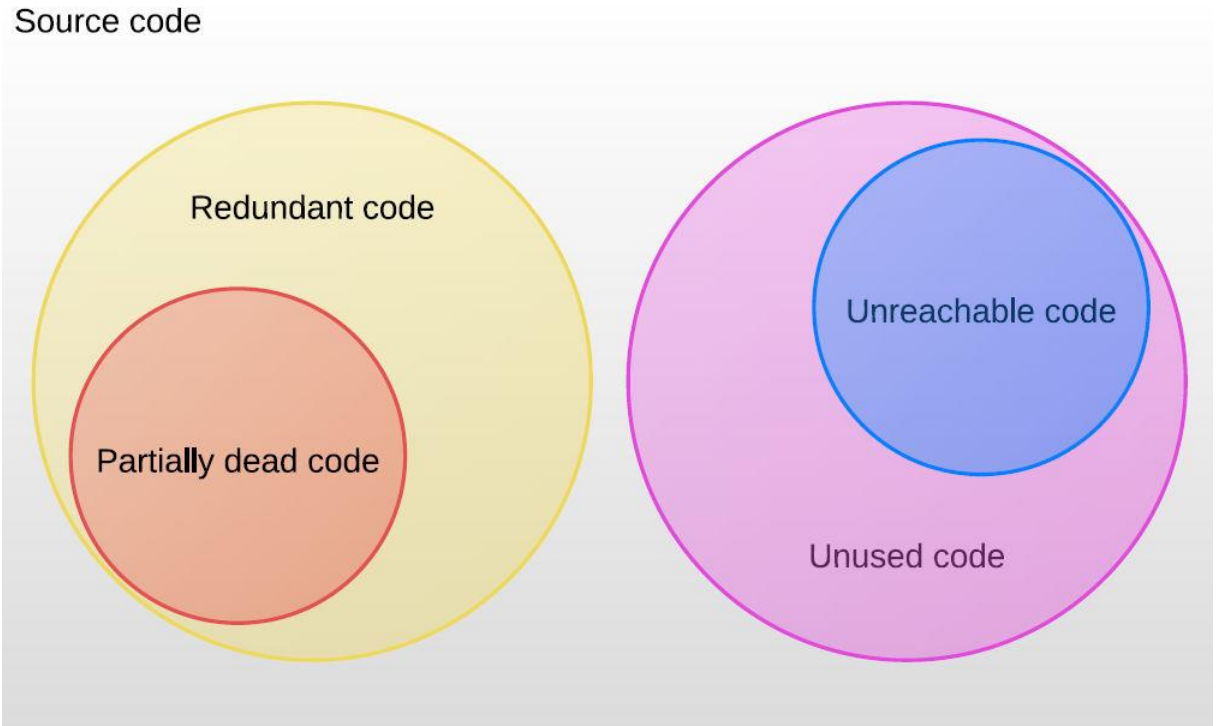


NTNU

Dispensables

A dispensable is something pointless and unneeded whose absence would make the code cleaner, more efficient and easier to understand.

- § **Comments**
- § **Duplicate Code**
- § **Data Class**
- § **Dead Code**
- § **Lazy Class**
- § **Speculative Generality**

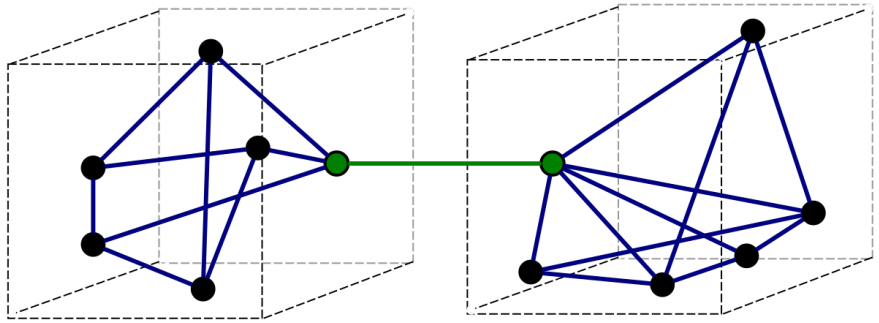


NTNU

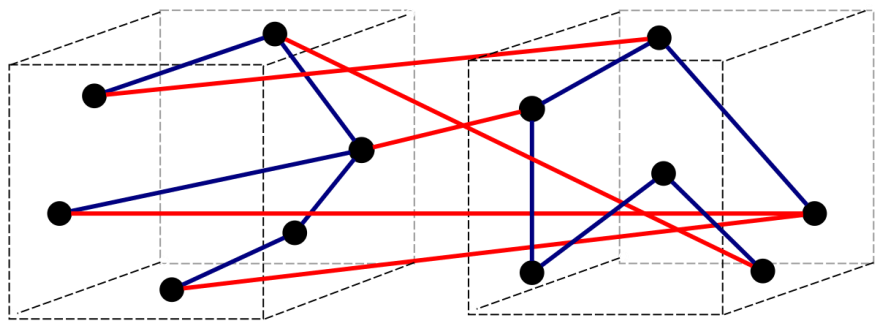
Couplers

All the smells in this group contribute to excessive coupling between classes or show what happens if coupling is replaced by excessive delegation.

- Feature Envy
- Inappropriate Intimacy
- Incomplete Library Class
- Message Chains
- Middle Man



a) Good (loose coupling, high cohesion)



b) Bad (high coupling, low cohesion)

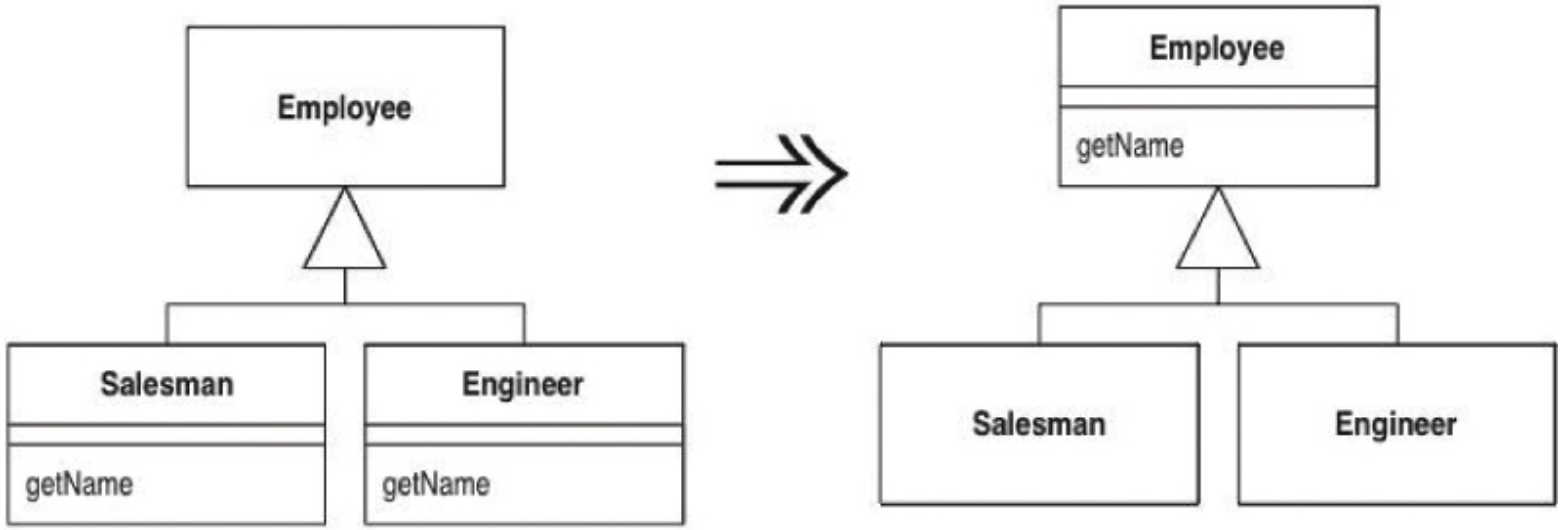
NTNU

CODE REFACTORING STEPS

- ✓ Designing solid tests for the section to be refactored
- ✓ Reviewing the code to identify bad smells of code
- ✓ Introducing refactoring and running tests (One step at a time)

REMOVE DUPLICATED CODE – PULL UP METHOD (DON'T REPEAT YOURSELF (DRY)) (ADDITIONAL READING)

- ✓ You have methods with identical results on subclasses.



REMOVE DUPLICATED CODE - SUBSTITUTE ALGORITHM (ADDITIONAL READING)

- ✓ You want to replace an algorithm with one that is clearer.

```
String foundPerson(String[] people){

  for (int i = 0; i < people.length; i++) {
      if (people[i].equals ("Don"))
      { return "Don"; }

       if (people[i].equals ("John"))
       { return "John"; }

       if (people[i].equals ("Kent"))
       { return "Kent";  }
    }
    return "";
}
```

```
String foundPerson(String[] people){
     List candidates = Arrays.asList(new
String[] {"Don", "John", "Kent"});

  for (int i=0; i<people.length; i++)
       if (candidates.contains(people[i]))
              return people[i];

    return "";

}
```

REDUCE SIZE – SHORTEN METHOD/CLASS

- ✓ **Long Method**
  - E.g., Extract method
- ✓ **Large Class**
  - E.g., Extract class, subclass, interface

EXTRACT METHOD EXAMPLE

If you have to spend effort looking at a fragment of code and figure out what it is doing, then you should extract it into a function/method and name it after "*what*."

```
void printOwing()
{
  printBanner(); //print details
  System.out.println ("name: " +
_name);
  System.out.println ("amount      "
+ getOutstanding());
}
```

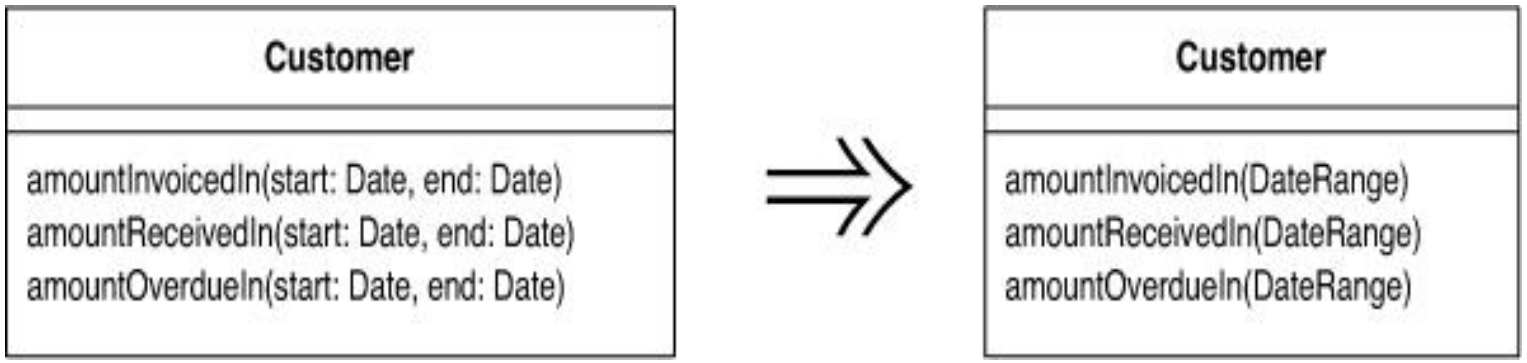
```
void printOwing() {        printBanner();
      printDetails(getOutstanding());}

void printDetails (double outstanding) {
  System.out.println ("name:" + _name);
  System.out.println ("amount:" +
outstanding);
}
```

REDUCE SIZE – SHORTEN PARAMETER LIST

✓ Long Parameter List

- E.g., Introduce Parameter Object



DIVERGENT CHANGE

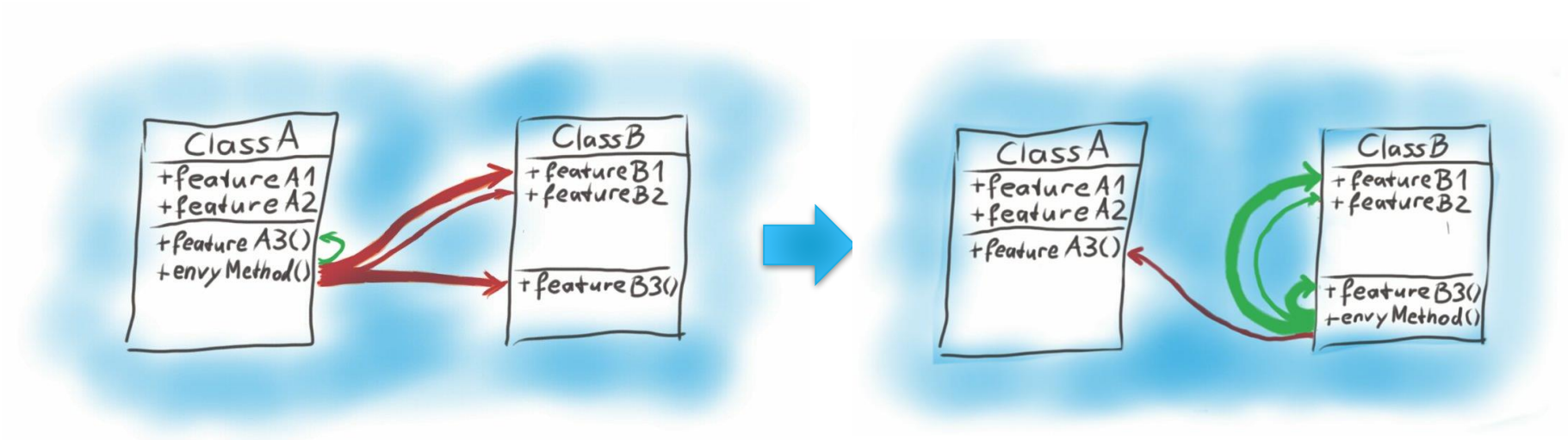
- ✓ **Code smell**
  - One module is often changed in different ways for different reasons.
  - Classes have more than distinct responsibilities that it has more than one reason to change
  - Violation of single responsibility design principle

- ✓ **Refactoring**
  - You identify everything that changes for a particular cause and put them all together

INCREASE COHESION

✓ Feature envy

- A function in one module spends more time communicating with functions or data inside another module than it does within its own module.
- Move function to give it a dream home



https://waog.wordpress.com/2014/08/25/code-smell-feature-envy/

INCREASE COHESION (CONT')

- **Data clumps**
  - Bunches of data often hang around together
  - Consolidate the data together, e.g., Introduce Parameter Object or Preserve Whole Object

```
int low = daysTempRange().getLow();
int high = daysTempRange().getHigh();
withinPlan = plan.withinRange(low, high);
```

```
withinPlan =
plan.withinRange(daysTempRange());
```

PRIMITIVE OBSESSION

- ✓ Primitive fields are basic built-in building blocks of a language, e.g., int, string, or constants

- ✓ Primitive Obsession is when the code relies too much on primitives and when uses primitive types to represent an object in a domain

```php
class contactUs
{
    public function addressUsa()
    {
        $address = new Array();
        $address['streetNo'] = 2074;
        $address['streetName'] = 'JFK street';
        $address['zipCode'] = '507874';

        return $address['streetName']. ' '. $address['streetNo']. ', '. $address['zipCode'];
    }

    public function addressGermany()
    {
        $address = new Array();
        $address['streetNo'] = '25';
        $address['streetName'] = 'Frankfurter str.';
        $address['zipCode'] = '80256';

        return $address['streetName']. ' '. $address['streetNo']. ', '. $address['zipCode'];
    }

    public function hotLine(){
        return '+49 01687 000 000';
    }
}
```

The address is defined as an array. Every time we need the address we will have to hard code it

https://codeburst.io/write-clean-code-and-get-rid-of-code-smells-aea271f30318

```php
class address{

    private $streetNo;
    private $streetName;
    private $zipCode;

    public function addressUsa()
    {
        $this->streetNo = 2074;
        $this->streetName = 'JFK street';
        $this->zipCode = '507874';

        return $this->streetName. ' '. $this->streetNo. ', '. $this->zipCode;
    }
    public function addressGermany()
    {
        $this->streetNo = 25;
        $this->streetName = 'Frankfurter str.';
        $this->zipCode = '80256';

        return $this->streetName. ' '. $this->streetNo. ', '. $this->zipCode;
    }
}
```

We create a new class called Address
Every time we need to add/edit an address we hit the Address class

https://codeburst.io/write-clean-code-and-get-rid-of-code-smells-aea271f30318

EXERCISE

- ✓ https://github.com/anhn/co3001_code_refactoring_lecture

WHAT IS INSPECTION?

- ✓ Visual examination of software product

- ✓ Identify software anomalies
  - Errors
  - Code smells
  - Deviations from specifications
  - Deviations from standards
    - E.g., Java code conventions

      www.oracle.com/technetwork/java/codeconventions-150003.pdf

WHY DO WE NEED INSPECTION?

- ✓ Many software artifacts cannot be verified by running tests, e.g.,
  - Requirement specification
  - Design
  - Pseudocode
  - User manual

- ✓ Inspection reduces defect rates
- ✓ Inspection complements testing
- ✓ Inspection identifies code smells
- ✓ Inspection provides additional benefits

INSPECTION FINDS TYPES OF DEFECTS DIFFERENT FROM TESTING

Number of different types of defects detected by testing vs. inspection [1]

Some defect types	Testing	Inspection
Uninitialized variables	1	4
Illegal behavior, e.g., division by zero	49	20
Incorrectly formulated branch conditions	2	13
Missing branches, including both conditionals and their embedded statements	4	10

INSPECTION REDUCES DEFECT RATES

Removal Step	Lowest Rate	Modal Rate	Highest Rate
Informal design reviews	25%	35%	40%
Formal design inspections	45%	55%	65%
Informal code reviews	20%	25%	35%
Formal code inspections	45%	60%	70%
Modeling or prototyping	35%	65%	80%
Personal desk-checking of code	20%	40%	60%
Unit test	15%	30%	50%
New function (component) test	20%	30%	35%
Integration test	25%	35%	40%
Regression test	15%	25%	30%
System test	25%	40%	55%
Low-volume beta test (<10 sites)	25%	35%	40%
High-volume beta test (>1,000 sites)	60%	75%	85%

Source: Adapted from *Programming Productivity* (Jones 1986a), "Software Defect-Removal Efficiency" (Jones 1996), and "What We Have Learned About Fighting Defects" (Shull et al. 2002).

TESTING VS. INSPECTION

	Testing	Inspection
Pros	• Can, at least partly, be automated • Can consistently repeat several actions • Fast and high volume	• Can be used on all types of documents, not just code • Can see the complete picture, not only a spot check • Can uses all information in the team • Can be innovative and inductive
Cons	• Is only a spot check • Can only be used for code • May need several stubs and drivers	• Is difficult to use on complex, dynamic situations • Unreliable (people can get tired) • Slow and low volume

CODE REVIEW AT GOOGLE (ADDITIONAL READING)

- ✓ "All code that gets submitted needs to be reviewed by at least one other person, and either the code writer or the reviewer needs to have readability in that language"
- ✓ "Most people use Mondrian to do code reviews, and obviously, we spend a good chunk of our time reviewing code"

--Amanda Camp, Software Engineer, Google

Modern Code Review: A Case Study at Google

Caitlin Sadowski, Emma Söderberg,
Luke Church, Michal Sipko
Google, Inc.
{supertri,emso,lukechurch,sipkom}@google.com

Alberto Bacchelli
University of Zurich
bacchelli@ifi.uzh.ch

ABSTRACT

Employing lightweight, tool-based code review of code changes (aka *modern code review*) has become the norm for a wide variety of open-source and industrial systems. In this paper, we make an exploratory investigation of modern code review at Google. Google introduced code review early on and evolved it over the years; our study sheds light on why Google introduced this practice and analyzes its current status, after the process has been refined through decades of code changes and millions of code reviews. By means of 12 interviews, a survey with 44 respondents, and the analysis of review logs for 9 million reviewed changes, we investigate motivations behind code review at Google, current practices, and developers' satisfaction and challenges.

CCS CONCEPTS

• **Software and its engineering** → **Software maintenance tools**;

ACM Reference format:
Caitlin Sadowski, Emma Söderberg,
Luke Church, Michal Sipko and Alberto Bacchelli. 2018. Modern Code Review: A Case Study at Google. In *Proceedings of 40th International Conference on Software Engineering: Software Engineering in Practice Track, Gothenburg, Sweden, May 27-June 3, 2018 (ICSE-SEIP '18)*, 10 pages.
DOI: 10.1145/3183519.3183525

1 INTRODUCTION

Peer code review, a manual inspection of source code by developers other than the author, is recognized as a valuable tool for improving the quality of software projects [2, 3]. In 1976, Fagan formalized a highly structured process for code reviewing—code inspections [16]. Over the years, researchers provided evidence on the benefits of code inspection, especially for defect finding, but the cumbersome, time-consuming, and synchronous nature of this approach hindered its universal adoption in practice [37]. Nowadays, most organizations adopt more lightweight code review practices to limit the inefficiencies of inspections [33]. Modern code review is (1) informal (in contrast to Fagan-style), (2) tool-based [32], (3) asynchronous, and (4) focused on reviewing code changes.

An open research challenge is understanding which practices represent valuable and effective methods of review in this novel context. Rigby and Bird quantitatively analyzed code review data from software projects spanning varying domains as well as organizations and found five strongly convergent aspects [33], which they conjectured can be prescriptive to other projects. The analysis of Rigby and Bird is based on the value of a *broad* perspective (that analyzes multiple projects from different contexts). For the development of an empirical body of knowledge, championed by Basili [7], it is essential to also consider a *focused and longitudinal* perspective that analyzes a single case. This paper expands on work by Rigby and Bird to focus on the review practices and characteristics established at Google, *i.e.*, a company with a multi-decade history of code review and a high-volume of daily reviews to learn from. This paper can be (1) prescriptive to practitioners performing code review and (2) compelling for researchers who want to understand and support this novel process.

Code review has been a required part of software development at Google since very early on in the company's history; because it was introduced so early on, it has become a core part of Google culture. The process and tooling for code review at Google have been iteratively refined for more than a decade and is applied by more than 25,000 developers making more than 20,000 source code changes each workday, in dozens of offices around the world [30].

We conduct our analysis in the form of an exploratory investigation focusing on three aspects of code review, in line with and expanding on the work by Rigby and Bird [33]: (1) The motivations driving code review, (2) the current practices, and (3) the perception of developers on code review, focusing on challenges encountered with a specific review (*breakdowns* in the review process) and satisfaction. Our research method combines input from multiple data sources: 12 semi-structured interviews with Google developers, an internal survey sent to engineers who recently sent changes to review with 44 responses, and log data from Google's code review tool pertaining to 9 million reviews over two years.

We find that the process at Google is markedly lighter weight than in other contexts, based on a single reviewer, quick iterations, small changes, and a tight integration with the code review tool. Breakdowns still exist, however, due to the complexity of the interactions that occur around code review. Nevertheless, developers consider this process valuable, confirm that it works well at scale, and conduct it for several reasons that also depend on the relationship between author and reviewers. Finally, we find evidence on the use of the code

ICSE-SEIP '18, Gothenburg, Sweden

INSPECT CODE 1 OF 5: CLASSES OVERALL

- ✓ C1. Is its (the class') name appropriate?
- ✓ C2. Could it be abstract (to be used only as a base)?
- ✓ C3. Does its header describe its purpose?
- ✓ C4. Does its header reference the requirements and/or design element to which it corresponds?
- ✓ C5. Does it state the package to which it belongs?
- ✓ C6. Is it as private as it can be?
- ✓ C7. Should it be final (Java)
- ✓ C8. Have the documentation standards been applied?

INSPECT CODE 2 OF 5 : ATTRIBUTES

- ✓ A1. Is it (the attribute) necessary?
- ✓ A2. Could it be static?
- ✓ A3. Should it be final?
- ✓ A4. Are the naming conventions properly applied?
- ✓ A5. Is it as private as possible?
- ✓ A6. Are the attributes as independent as possible?
- ✓ A7. Is there a comprehensive initialization strategy?

INSPECT CODE 3 OF 5 : CONSTRUCTORS

- ✓ CO1. Is it (the constructor) necessary?
- ✓ CO2. Does it leverage existing constructors?
- ✓ CO3. Does it initialize of all the attributes?
- ✓ CO4. Is it as private as possible?
- ✓ CO5. Does it execute the inherited constructor(s) where necessary?

INSPECT CODE 4 OF 5: METHOD HEADERS

- ✓ MH1. Is the method appropriately named?
- ✓ MH2. Is it as private as possible?
- ✓ MH3. Could it be static?
- ✓ MH4. Should it be final?
- ✓ MH5. Does the header describe method's purpose?
- ✓ MH6. Does the method header reference the requirements and/or design section that it satisfies?
- ✓ MH7. Does it state all necessary invariants?
- ✓ MH8. Does it state all pre-conditions?
- ✓ MH9. Does it state all post-conditions?
- ✓ MH10.Does it apply documentation standards?
- ✓ MH11.Are the parameter types restricted?

INSPECT CODE 5 OF 5: METHOD BODIES

- ✓ MB1. Is the algorithm consistent with the detailed design pseudocode and/or flowchart?
- ✓ MB2. Does the code assume no more than the stated preconditions?
- ✓ MB3. Does the code produce every one of the postconditions?
- ✓ MB4. Does the code respect the required invariant?
- ✓ MB5. Does every loop terminate?
- ✓ MB6. Are required notational standards observed?
- ✓ MB7. Has every line been thoroughly checked?
- ✓ MB8. Are all braces balanced?
- ✓ MB9. Are illegal parameters considered?
- ✓ MB10. Does the code return the correct type?
- ✓ MB11. Is the code thoroughly commented?