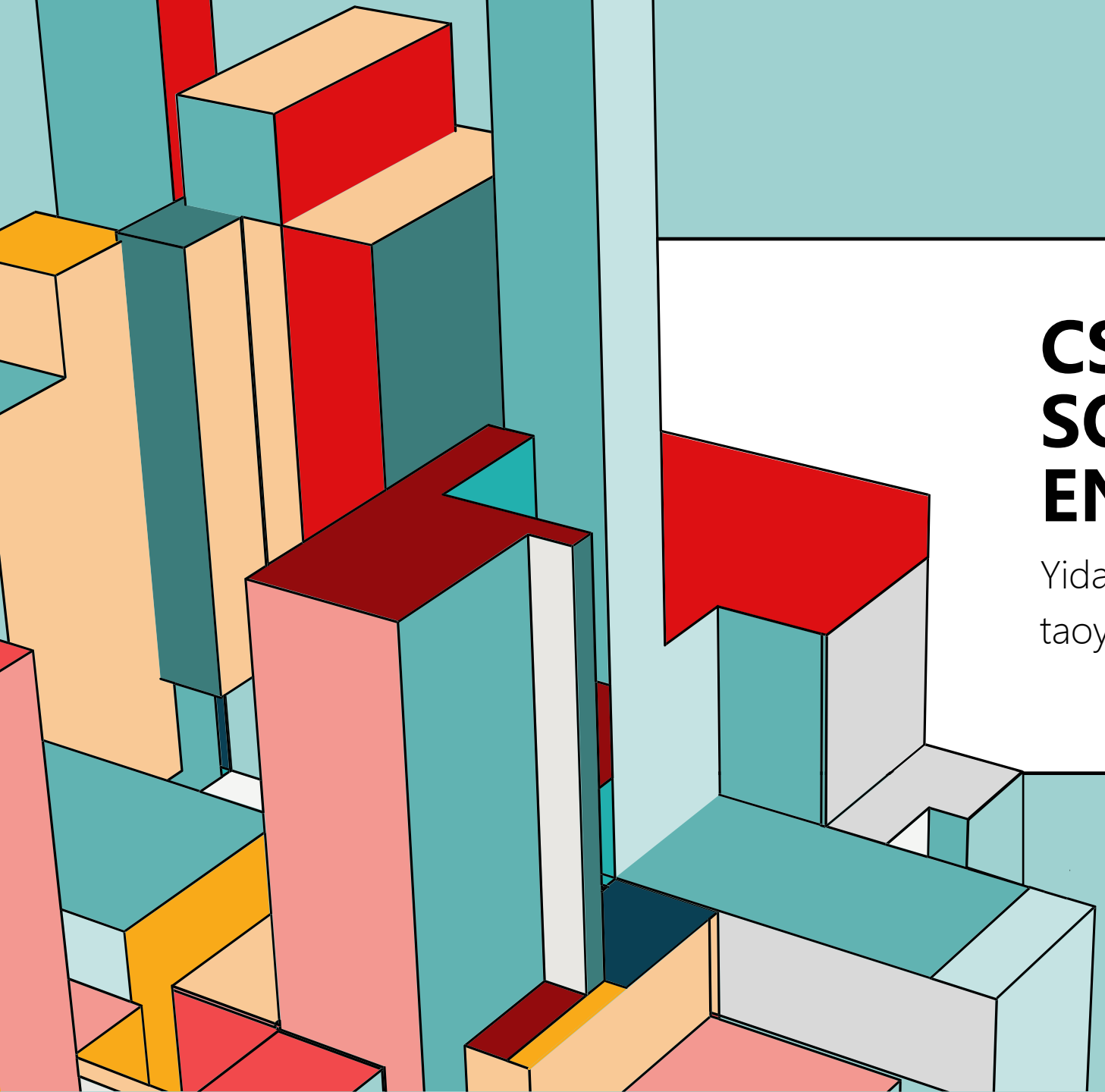


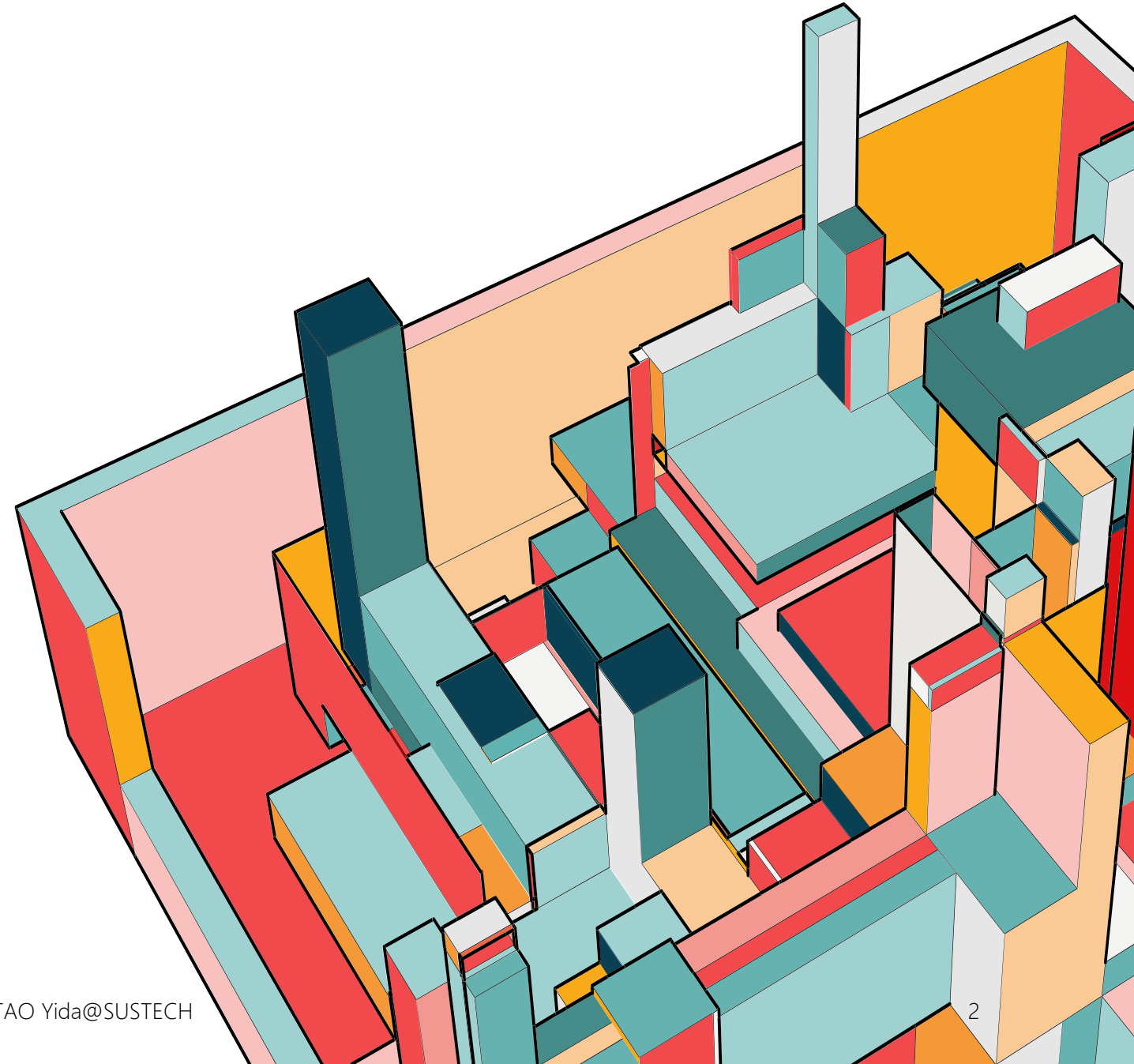
CS304 SOFTWARE ENGINEERING

Yida Tao

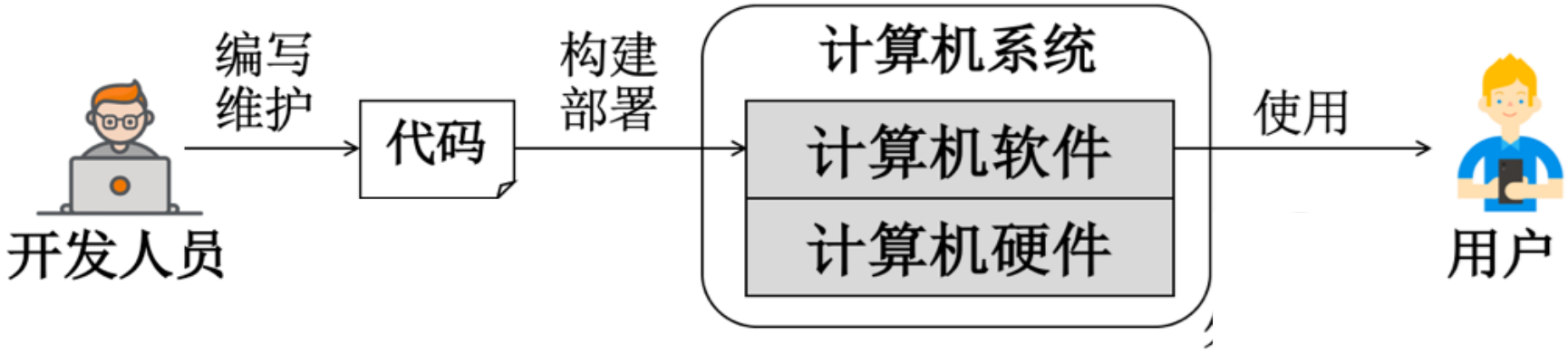
taoyd@sustech.edu.cn

LECTURE 7

- Code quality
- Code review

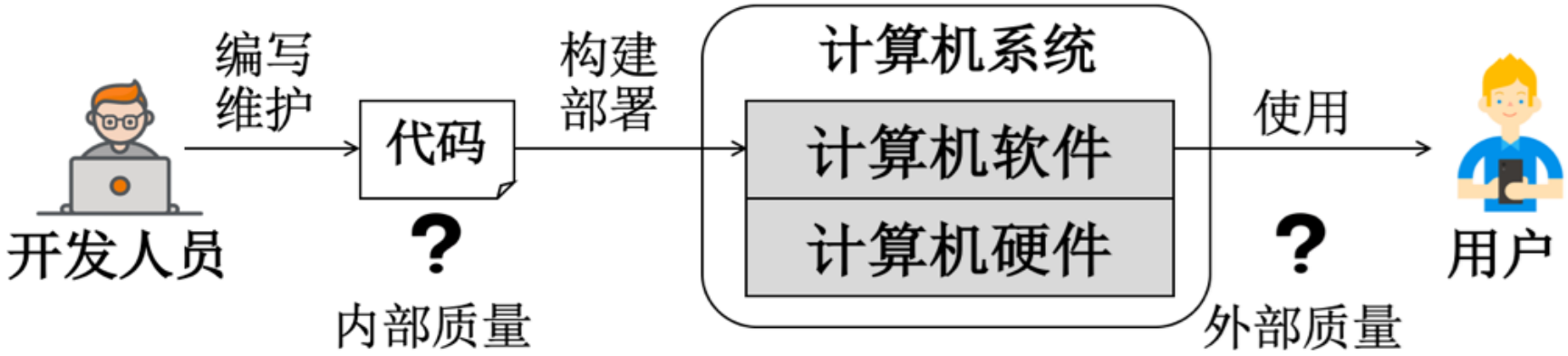


SOFTWARE QUALITY



- External quality is the usefulness of the system as perceived from outside. It provides customer value and meets the product owner's specifications.
- This quality can be measured through feature tests, QA and customer feedback.

SOFTWARE QUALITY

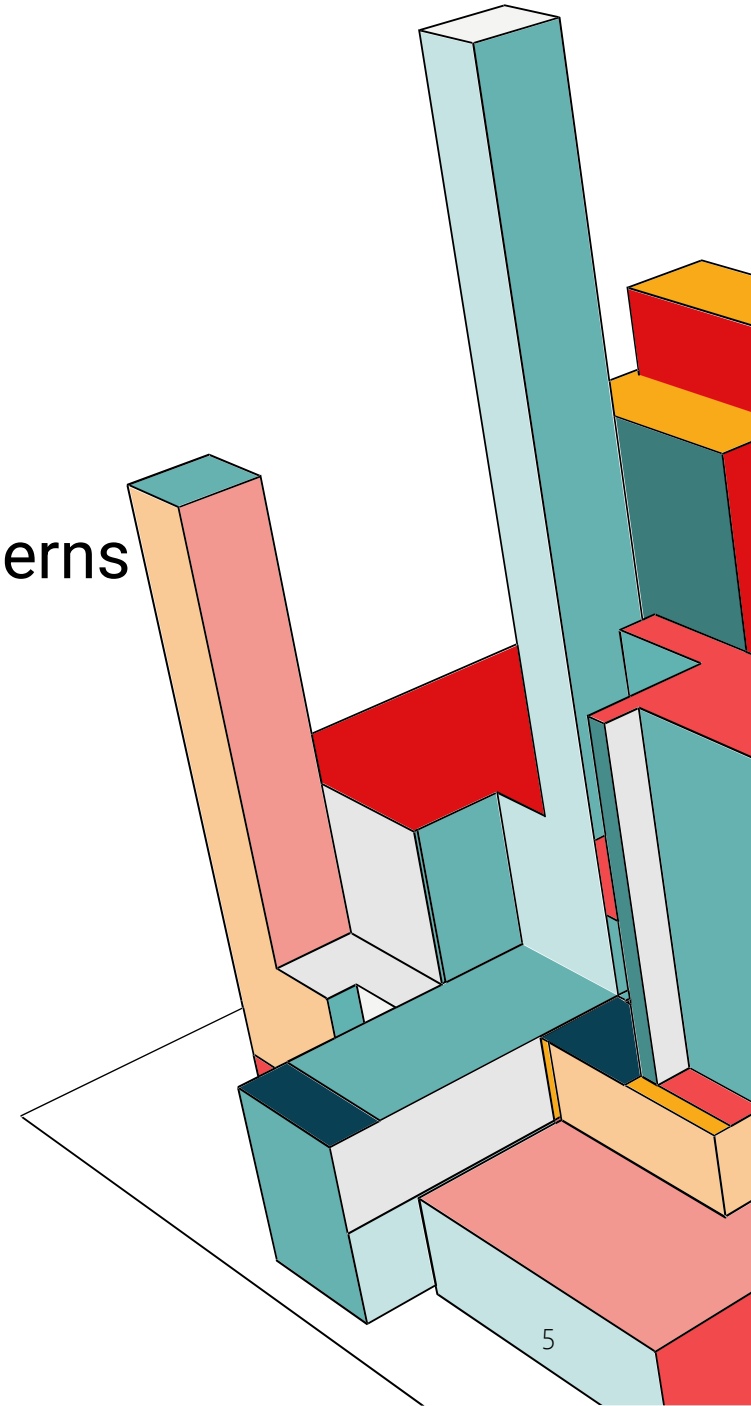


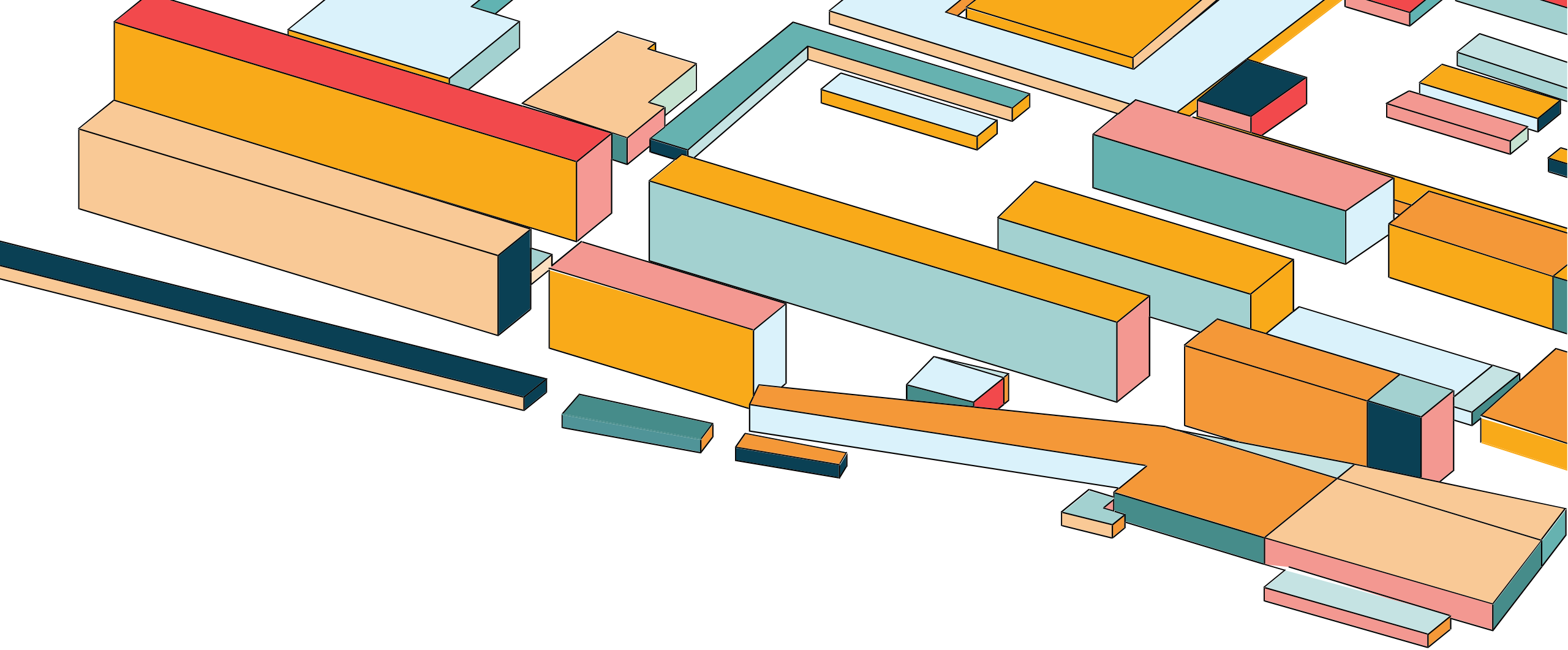
- Internal quality has to do with the way that the system has been constructed. It affects your ability to manage and reason about the program.
- This quality can be measured through linters, unit tests etc.

CODE QUALITY

In addition to **logic correctness**, code quality also concerns

- Understandability (可理解性)
- Maintainability (可维护性)
- Reliability (可靠性)
- Security (安全性)
- Efficiency (高效性)
- Portability (可移植性)





CASE STUDY: BORROW BOOK

EXAMPLE OF BAD CODE: BORROW BOOK

```
1       public void borrowBook(String id1, String id2) {
2           if (! id1.equals(""))) {
3               if (checkBooks(id1) < 5) {
4                   if (!(hasOverdueBooks(id1))) {
5                       if (! id2.equals(""))) {
6                       if (! getBook(id2).isAllOut() && getBook(id2).blocked())
7                           updateStudentStatus(id1, id2); updateBookStatus(id2, id1);
8                       throw new IllegalArgumentException("This book can't be borrowed");
9                       } else
10                          throw new IllegalArgumentException("Book id can't be empty");
11                  }
12                  else
13                      throw new IllegalArgumentException("Overdue books exist");
14              } else
15                  throw new IllegalArgumentException("The number of books borrowed exceeds the limit");
16          } else
17              throw new IllegalArgumentException("Student id can't be empty");
18      }
```

EXAMPLE OF BAD CODE: BORROW BOOK

```
1       public void borrowBook(String id1, String id2) {
2           if (! id1.equals(""))) {
3               if (checkBooks(id1) < 5) {
4                   if (!(hasOverdueBooks(id1))) {
5                       if (! id2.equals(""))) {
6                       if (! getBook(id2).isAllOut() && getBook(id2).blocked())
7                           updateStudentStatus(id1, id2); updateBookStatus(id2, id1);
8                       throw new IllegalArgumentException("This book can't be borrowed");
9                       } else
10                          throw new IllegalArgumentException("Book id can't be empty");
11                  }
12                  else
13                      throw new IllegalArgumentException("Overdue books exist");
14              } else
15                  throw new IllegalArgumentException("The number of books borrowed exceeds the limit");
16          } else
17              throw new IllegalArgumentException("Student id can't be empty");
18      }
```

Confusing identifiers: which is student ID, and which is book ID?

EXAMPLE OF BAD CODE: BORROW BOOK

```
1       public void borrowBook(String id1, String id2) {
2           if (! id1.equals(""))) {
3               if (checkBooks(id1) < 5) {
4                   if (!(hasOverdueBooks(id1))) {
5                       if (! id2.equals(""))) {
6                       if (! getBook(id2).isAllOut() && getBook(id2).blocked())
7                           updateStudentStatus(id1, id2); updateBookStatus(id2, id1);
8                       throw new IllegalArgumentException("This book can't be borrowed");
9                       } else
10                          throw new IllegalArgumentException("Book id can't be empty");
11                  }
12                  else
13                      throw new IllegalArgumentException("Overdue books exist");
14              } else
15                  throw new IllegalArgumentException("The number of books borrowed exceeds the limit");
16          } else
17              throw new IllegalArgumentException("Student id can't be empty");
18      }
```

Misleading indentation: if statements should be nested, instead of the same level

EXAMPLE OF BAD CODE: BORROW BOOK

```
1       public void borrowBook(String id1, String id2) {
2           if (! id1.equals(""))) {
3               if (checkBooks(id1) < 5) {
4                   if (!(hasOverdueBooks(id1))) {
5                       if (! id2.equals(""))) {
6                       if (! getBook(id2).isAllOut() && getBook(id2).blocked())
7                           updateStudentStatus(id1, id2); updateBookStatus(id2, id1);
8                       throw new IllegalArgumentException("This book can't be borrowed");
9                       } else
10                          throw new IllegalArgumentException("Book id can't be empty");
11                  }
12                  else
13                      throw new IllegalArgumentException("Overdue books exist");
14              } else
15                  throw new IllegalArgumentException("The number of books borrowed exceeds the limit");
16          } else
17              throw new IllegalArgumentException("Student id can't be empty");
18      }
```

Deep nesting: 5 nested if statements, hard to understand

EXAMPLE OF BAD CODE: BORROW BOOK

```
1   public void borrowBook(String id1, String id2) {
2     if (! id1.equals(""))) {
3       if (checkBooks(id1) < 5) {
4         if (!(hasOverdueBooks(id1))) {
5           if (! id2.equals(""))) {
6           if (! getBook(id2).isAllOut() && getBook(id2).blocked())
7             updateStudentStatus(id1, id2); updateBookStatus(id2, id1);
8           throw new IllegalArgumentException("This book can't be borrowed");
9           } else
10            throw new IllegalArgumentException("Book id can't be empty");
11        }
12        else
13          throw new IllegalArgumentException("Overdue books exist");
14      } else
15        throw new IllegalArgumentException("The number of books borrowed exceeds the limit");
16    } else
17      throw new IllegalArgumentException("Student id can't be empty");
18  }
```

Improper exception type: only line 10 and line 17 are IllegalArgumentException.

EXAMPLE OF BAD CODE: BORROW BOOK

```
1       public void borrowBook(String id1, String id2) {
2           if (! id1.equals(""))) {
3               if (checkBooks(id1) < 5) {
4                   if (!(hasOverdueBooks(id1))) {
5                       if (! id2.equals(""))) {
6                       if (! getBook(id2).isAllOut() && getBook(id2).blocked())
7                           updateStudentStatus(id1, id2); updateBookStatus(id2, id1);
8                       throw new IllegalArgumentException("This book can't be borrowed");
9                       } else
10                          throw new IllegalArgumentException("Book id can't be empty");
11                  }
12                  else
13                      throw new IllegalArgumentException("Overdue books exist");
14              } else
15                  throw new IllegalArgumentException("The number of books borrowed exceeds the limit");
16          } else
17              throw new IllegalArgumentException("Student id can't be empty");
18      }
```

Potential bugs 1: no null-checking for id1 and id2

EXAMPLE OF BAD CODE: BORROW BOOK

```
1       public void borrowBook(String id1, String id2) {
2           if (! id1.equals(""))) {
3               if (checkBooks(id1) < 5) {
4                   if (!(hasOverdueBooks(id1))) {
5                       if (! id2.equals(""))) {
6                       if (! getBook(id2).isAllOut() && getBook(id2).blocked())
7                           updateStudentStatus(id1, id2); updateBookStatus(id2, id1);
8                       throw new IllegalArgumentException("This book can't be borrowed");
9                       } else
10                          throw new IllegalArgumentException("Book id can't be empty");
11                  }
12                  else
13                      throw new IllegalArgumentException("Overdue books exist");
14              } else
15                  throw new IllegalArgumentException("The number of books borrowed exceeds the limit");
16          } else
17              throw new IllegalArgumentException("Student id can't be empty");
18      }
```

Potential bugs 2: missing ! operator

EXAMPLE OF BAD CODE: BORROW BOOK

```
1       public void borrowBook(String id1, String id2) {
2           if (! id1.equals(""))) {
3               if (checkBooks(id1) < 5) {
4                   if (!(hasOverdueBooks(id1))) {
5                       if (! id2.equals(""))) {
6                       if (! getBook(id2).isAllOut() && getBook(id2).blocked())
7                           updateStudentStatus(id1, id2); updateBookStatus(id2, id1);
8                       throw new IllegalArgumentException("This book can't be borrowed");
9                       } else
10                          throw new IllegalArgumentException("Book id can't be empty");
11                  }
12                  else
13                      throw new IllegalArgumentException("Overdue books exist");
14              } else
15                  throw new IllegalArgumentException("The number of books borrowed exceeds the limit");
16          } else
17              throw new IllegalArgumentException("Student id can't be empty");
18      }
```

Potential bugs 3: missing else block

IMPROVEMENT 1

```
1       public void borrowBook(String studentID, String bookID) throw ServiceException {
2           if (studentID == null || "".equals(studentID))
3               throw new StudentIDEmptyException();
4           if (getBorrowedBookCount(studentID) >= BOOK_BORROW_LIMIT)
6               throw new BookBorrowExeedLimitException();
7           if (hasOverdueBooks(studentID))
8               throw new OverdueBooksException();
9           if (bookID == null || "".equals(bookID))
10              throw new BookIDEmptyException();
11          if (getBook(bookID).isAllOut() || getBook(bookID).blocked())
12              throw new BookUnavailableException();
13          updateStudentStatus(studentID, bookID);
14          updateBookStatus(bookID, studentID);
15      }
```

IMPROVEMENT 2

```
1       public void borrowBook(String studentID, String bookID) throw ServiceException {
2           notEmpty(studentID, "Student ID can't be empty");
3           notEmpty(bookID, "Book ID can't be empty");
4
5           checkStudentCanBorrowBook(studentID);
6           checkBookCanBeBorrowed(bookID);
7           updateBorrowingStatus(studentID, bookID);
…           ……
```

Defining notEmpty, check* and update* methods
Identifiers and logics are self-explanatory

UNDERSTANDABILITY

Developers spend around 70% of their time understanding code*

- Understandability is the concept that a system should be presented so that an engineer can easily comprehend it.
- The more understandable a system is, the easier it will be for engineers to change it in a predictable and safe manner.

* Roberto Minelli, Andrea Mocci, and Michele Lanza. 2015. I know what you did last summer: an investigation of how developers spend their time. In Proc. International Conference on Program Comprehension (ICPC). IEEE, 25–35.

CODE OF POOR UNDERSTANDABILITY

```
1       public List<int[]> getList() {
2           List<int[]> list = new ArrayList<int[]>();
3           for (int[] x: theList) {
4               if (x[0] == 4) {
5                   list.add(x);
6               }
7           }
8           return list;
9       }
```

- What's **theList**? (ambiguous identifier)
- What do **0** and **4** mean? (ambiguous magic number)

CODE OF GOOD UNDERSTANDABILITY

```
public List<int[]> getList() {
    List<int[]> list = new ArrayList<int[]>();
    for (int[] x: theList) {
        if (x[0] == 4) {
            list.add(x);
        }
    }
    return list;
}
```

```
public List<Cell> getFlaggedCells() {
    List<Cell> flaggedCells = new ArrayList<Cell>();
    for (Cell cell: gameBoard) {
        if (cell.isFlagged()) {
            flaggedCells.add(cell);
        }
    }
    return flaggedCells;
}
```

MEANINGFUL IDENTIFIER NAMES

- Use Intention-Revealing Names
- Name Functions as Verbs
- Name Classes as Nouns
- Use Meaningful Distinction
- Use Pronounceable Names
- Use Searchable Names

MEANINGFUL IDENTIFIER NAMES

- Use Intention-Revealing Names
- Name Functions as Verbs
- Name Classes as Nouns
- **Use Meaningful Distinction**
- Use Pronounceable Names
- Use Searchable Names

```
int[] arr1;

int[] arr2;

String status;

String statusValue;
```

Noise words like **Data, Value, Info, Variable, Table, String, Object**, etc which are used as a suffix do not offer any meaningful distinction. Noise words are redundant and should be avoided.

MEANINGFUL IDENTIFIER NAMES

- Use Intention-Revealing Names
- Name Functions as Verbs
- Name Classes as Nouns
- Use Meaningful Distinction
- Use Pronounceable Names
- **Use Searchable Names**

```
1   switch (cardState) {
2       case 1: // valid
3           // ...
4           break;
5       case 2: // freezed
6           // ...
7           break;
8       case 3: // expired
9           // ...
10          break;
11      default: ...
12  }
```

Avoid magic numbers:
Create named constants instead of using numbers or other constant values where it is supposed to denote something.

MEANINGFUL IDENTIFIER NAMES

- Use Intention-Revealing Names
- Name Functions as Verbs
- Name Classes as Nouns
- Use Meaningful Distinction
- Use Pronounceable Names
- **Use Searchable Names**

```
1   switch (cardState) {
2       case CARD_STATE_VALID:
3           // ...
4           break;
5       case CARD_STATE_FREEZED:
6           // ...
7           break;
8       case CARD_STATE_EXPIRED:
9           // ...
10          break;
11      default:
12  }
```

MAINTAINABILITY

Code maintainability simply means that code is easy to modify, extend, search, or reuse

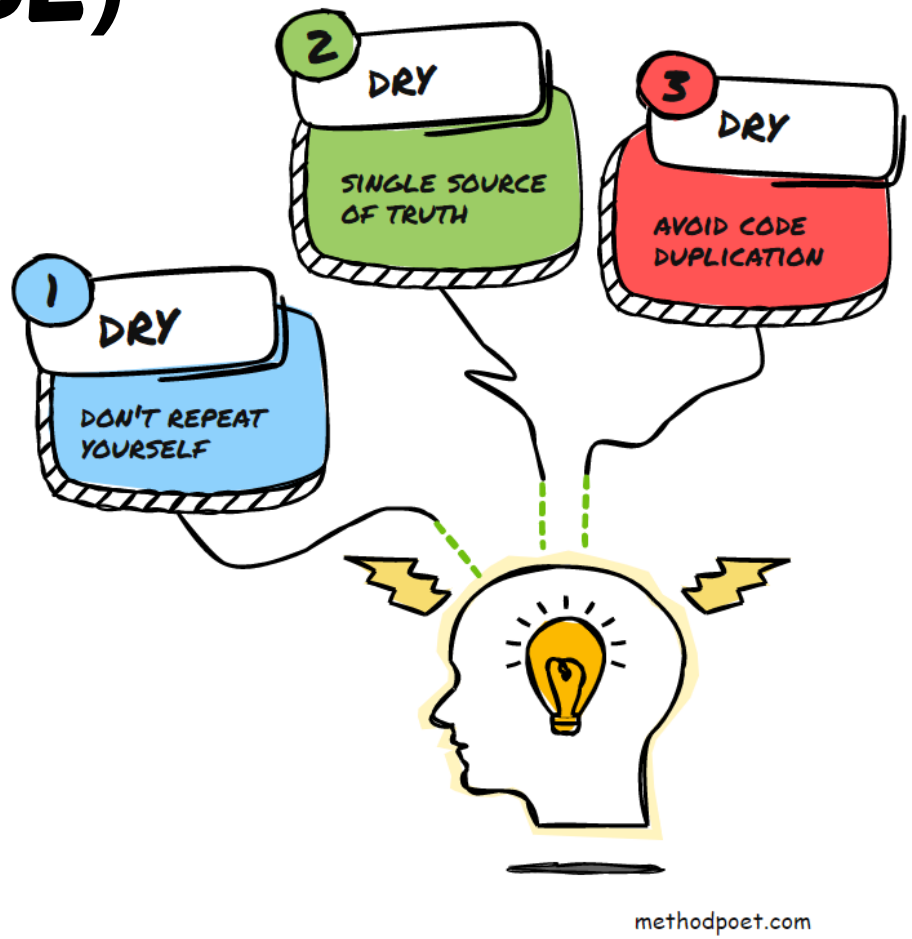
If your team is doing a good job with code maintainability, the following are true:

- (Search) It's easy for the team to find examples in the codebase, reuse other people's code, and change code maintained by other teams if necessary.
- (Extend) It's easy for the team to add new dependencies to their project, and to migrate to a new version of a dependency.
- (Modify & Reuse) The team's dependencies are stable and rarely break the code.

https://cloud.google.com/architecture/devops/devops-tech-code-maintainability

CODE CLONES (DUPLICATE CODE)

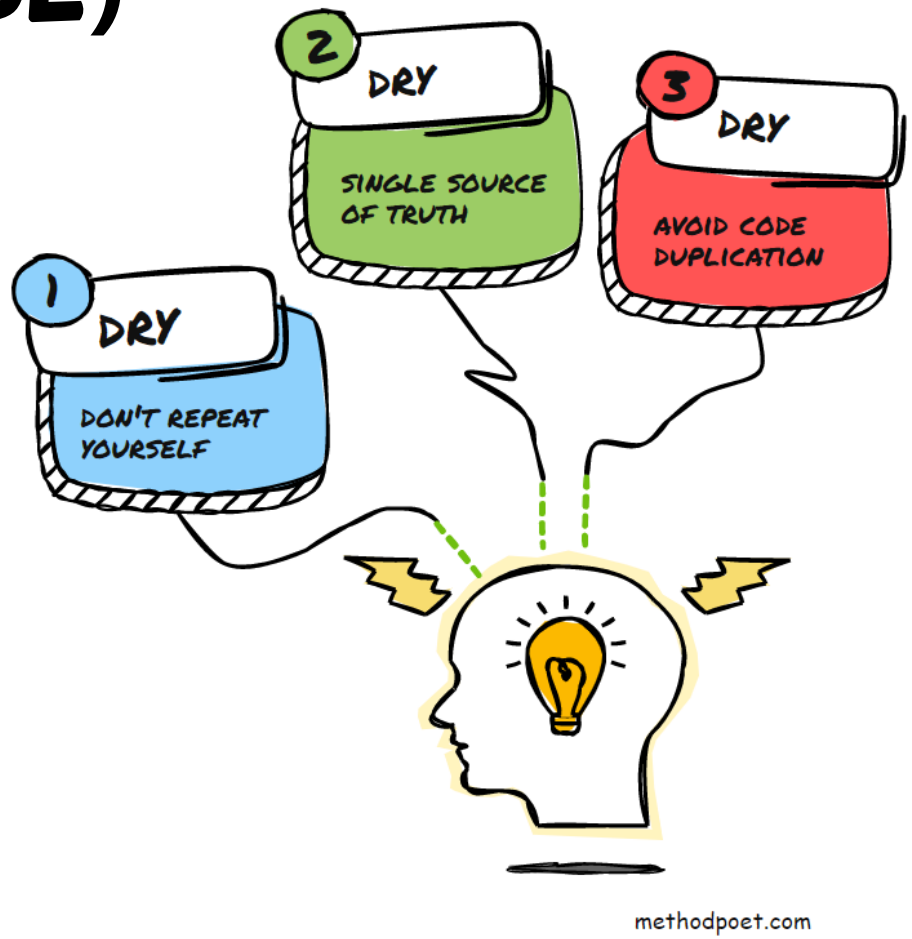
- In a software system, similar or identical fragments of code are known as code clones (or duplicate code)
- Code clones often result from copy/paste programming from within the project or other projects/resources (e.g., online tutorials)



CODE CLONES (DUPLICATE CODE)

Code clones might be necessary in some cases; however, code clones are generally considered "harmful" in terms of maintainability

- Increase code size
- Increase maintenance cost: If one code fragment contains a bug and gets fixed, all its clones should always be fixed in similar ways



TYPES OF CODE CLONES

Type 1: Exact copy, only differences in white space and comments.
Type 2: Same as type 1, but also variable renaming.
Type 3: Same as type 2, but also changing or adding few statements.
Type 4: Semantically identical, but not necessarily same syntax.

Initial Code Fragement CF_0

```
for(i = 0; i < 10; i++)
{
    // foo 2
    if (i % 2 == 0)
        a = b + i;
    else
        // foo 1
        a = b - i;
}
```

CF_1 – Type-1 Clone

```
for(i = 0; i < 10; i++)
{
    if (i % 2 == 0)
      a = b + i; //cmt 1
    else
      a = b - i; //cmt 2
}
```

https://www.researchgate.net/figure/Clone-Types-1-to-Type-4_fig2_335152710

TYPES OF CODE CLONES

Type 1: Exact copy, only differences in white space and comments.
Type 2: Same as type 1, but also variable renaming.
Type 3: Same as type 2, but also changing or adding few statements.
Type 4: Semantically identical, but not necessarily same syntax.

Initial Code Fragement CF_0

```
for(i = 0; i < 10; i++)
{
    // foo 2
    if (i % 2 == 0)
        a = b + i;
    else
        // foo 1
        a = b - i;
}
```

CF_2 – Type-2 Clone

```
for(j = 0; j < 10; j++)
{
    if (j % 2 == 0)
      y = x + j; //cmt 1
    else
      y = x - j; //cmt 2
}
```

https://www.researchgate.net/figure/Clone-Types-1-to-Type-4_fig2_335152710

TYPES OF CODE CLONES

Type 1: Exact copy, only differences in white space and comments.
Type 2: Same as type 1, but also variable renaming.
Type 3: Same as type 2, but also changing or adding few statements.
Type 4: Semantically identical, but not necessarily same syntax.

Initial Code Fragement CF_0

```
for(i = 0; i < 10; i++)
{
    // foo 2
    if (i % 2 == 0)
        a = b + i;
    else
        // foo 1
        a = b - i;
}
```

CF_3 – Type-3 Clone

```
for(i = 0; i < 10; i++)
{
    // new statement
    a = 10 * b;
    if (i % 2 == 0)
      a = b + i; //cmt 1
    else
      a = b - i; //cmt 2
}
```

https://www.researchgate.net/figure/Clone-Types-1-to-Type-4_fig2_335152710

TYPES OF CODE CLONES

Type 1: Exact copy, only differences in white space and comments.
Type 2: Same as type 1, but also variable renaming.
Type 3: Same as type 2, but also changing or adding few statements.
Type 4: Semantically identical, but not necessarily same syntax.

Initial Code Fragement CF_0

```
for(i = 0; i < 10; i++)
{
    // foo 2
    if (i % 2 == 0)
        a = b + i;
    else
        // foo 1
        a = b - i;
}
```

CF_4 – Type-4 Clone

```
while(i < 10)
{
    // a comment
  a = (i % 2 == 0) ?
  b + i : b - i;
  i++;
}
```

https://www.researchgate.net/figure/Clone-Types-1-to-Type-4_fig2_335152710

REDUCING CODE CLONES

Code reuse: turn duplicate code to reusable methods, classes, modules, or libraries

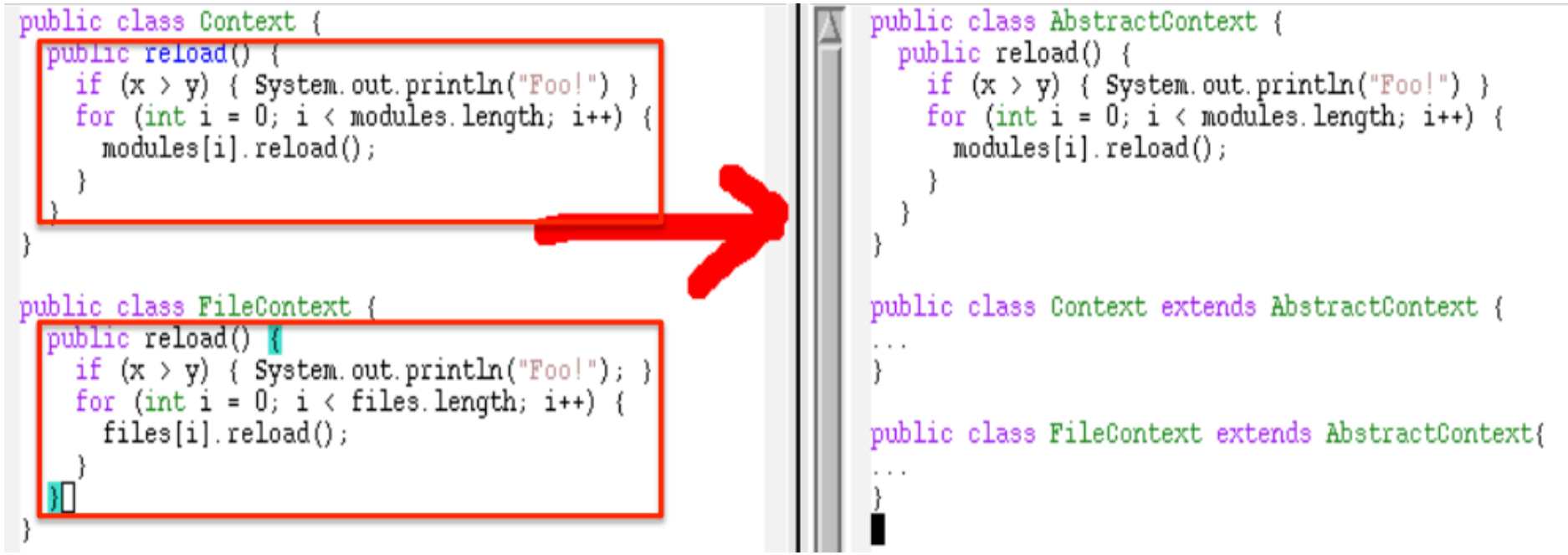
```
int main(int argc char** argv) {
  int x = 3;
  int y = 2;
  Socket s = opensocket(x, y);
  char buf[80] = readsocket(s, 80);
  closesocket(s);

  int xx = 47;
  int yy = 21;
  Socket ss = opensocket(xx, yy);
  char fub[50] = readsocket(ss, 50);
  closesocket(ss);

  return 0;
}
```

```
void getdata(int x, int y, int size, char * buf) {
  Socket s = opensocket(x, y);
  readsocket(s, buf, 80);
  closesocket(s);
}

int main(int argc char** argv) {
  char buf[80];
  getdata(3, 2, 80, &buf);

  char fub[50];
  getdata(47, 21, 50, &fub);

  return 0;
}
```

Extract method:
Extract duplicate code to a new method

REDUCING CODE CLONES

Code reuse: turn duplicate code to reusable methods, classes, modules, or libraries



```
public class Context {
  public reload() {
    if (x > y) { System.out.println("Foo!") }
    for (int i = 0; i < modules.length; i++) {
      modules[i].reload();
    }
  }
}

public class FileContext {
  public reload() {
    if (x > y) { System.out.println("Foo!"); }
    for (int i = 0; i < files.length; i++) {
      files[i].reload();
    }
  }
}
```

```
public class AbstractContext {
  public reload() {
    if (x > y) { System.out.println("Foo!") }
    for (int i = 0; i < modules.length; i++) {
      modules[i].reload();
    }
  }
}

public class Context extends AbstractContext {
...
}

public class FileContext extends AbstractContext{
...
}
```

Pull up method: taking a method and "Pulling" it up in the inheritance chain.
This is used when a method needs to be used by multiple implementers.

CLONE DETECTION TECHNIQUES

Tool	*Proposed By*	*Language Supported*	*Technique*	*Application*
CloneDr	Baxter et. al., [3]	C, C++, Java and Cobol	Abstract Syntax Tree Method	Clone Detection
CCFinder	Kamiya et. al., [4]	C, C++, Java	Token Based Method	Clone Detection
CP-Miner	Li et. al., [5]	C, C++, Java	Frequent Subsequent Mining	Clone Detection and copy-pasted bug identification
Bauhaus	Bellon [6]	C, C++, Java	Abstract Syntax Tree	Clone Detection
Coogle	Sager et. al., [7]	Java	Abstract Syntax Tree	Finding identical java class
Deckard	Jiang et. al. [8]	C, Java	Tree Matching, Euclidean space	Clone Detection
CCFnderX	Kamiya et. al., [9]	C, C++, Java, COBOL, VB, C#	Token Based Approach	Clone Detection
PMD	Sourcefourge community [10]	Java, C, C++, JSP, Ruby, PHP, PLSQL etc	String Matching	Clone Detection
PDG-Dup	Komondor et.al., [11]	C,C++	Program Dependence Graph	Clone Detection
Duplix	Krinke et. al., [12]	C	Program Dependence Graph	Clone Detection

https://www.researchgate.net/publication/275772216_Performance_Evaluation_of_Clone_Detection_Tools

1. Text matching 2. Token sequence matching 3. Graph matching

TEXT MATCHING

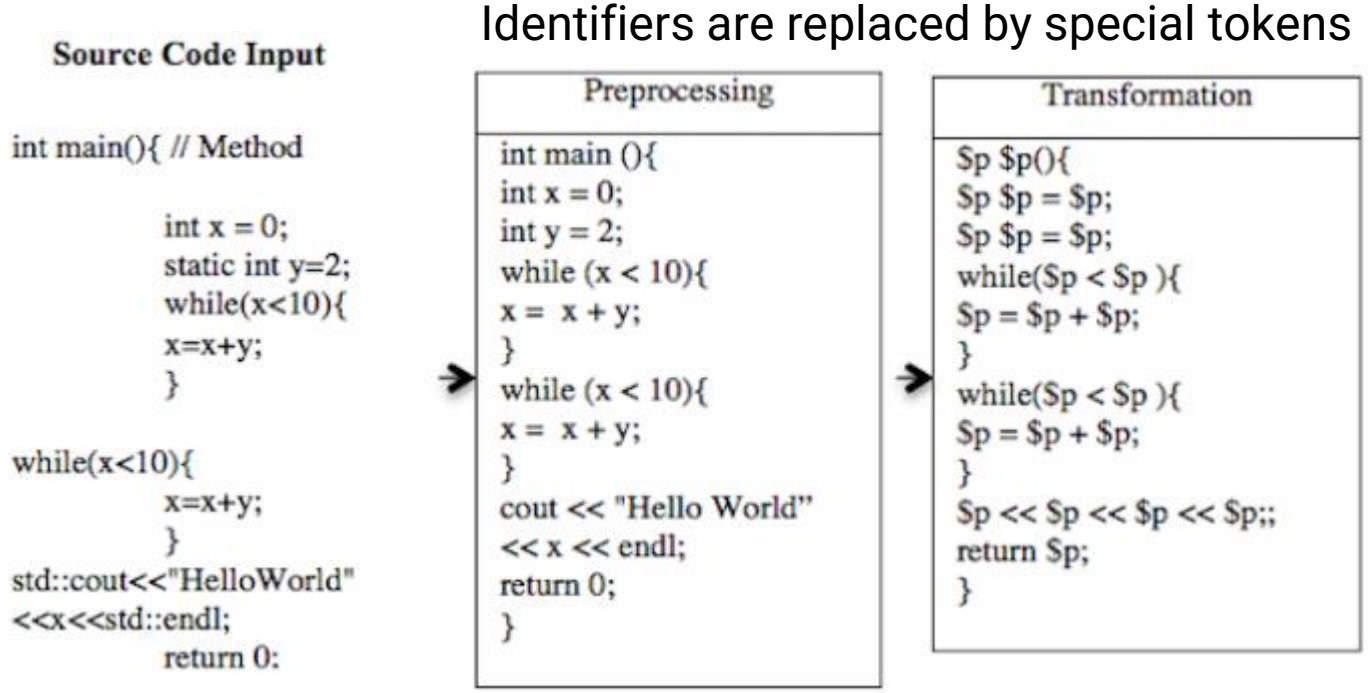
- Older, studied extensively
- Less complex, and most widely used
- No program structure is taken into consideration
- Detect Type-1 clones & some Type-2 clones
- Two types of text matching
  - Exact string match: diff (cvs, svn, git) is based on exact text matching
  - Ambiguous match: LCS, n-gram match

https://courses.cs.vt.edu/cs5704/spring16/handouts/5704-10-CodeClones.pdf

TOKEN SEQUENCE MATCHING

- A little more complex, less widely used
- No program structure is taken into account
- Type-1 and Type-2 clones

Identifiers are replaced by special tokens



http://www.cs.uccs.edu/~jkalita/papers/2016/SheneamerAbdullahIJCA2016.pdf

GRAPH MATCHING

- Newer
- More complex
- Type-1, Type-2, and Type-3 clones

Graphs for modeling source code

- AST (Abstract Syntax Tree)
- CFG (Control Flow Graph)
- PDG (Program Dependency Graph)

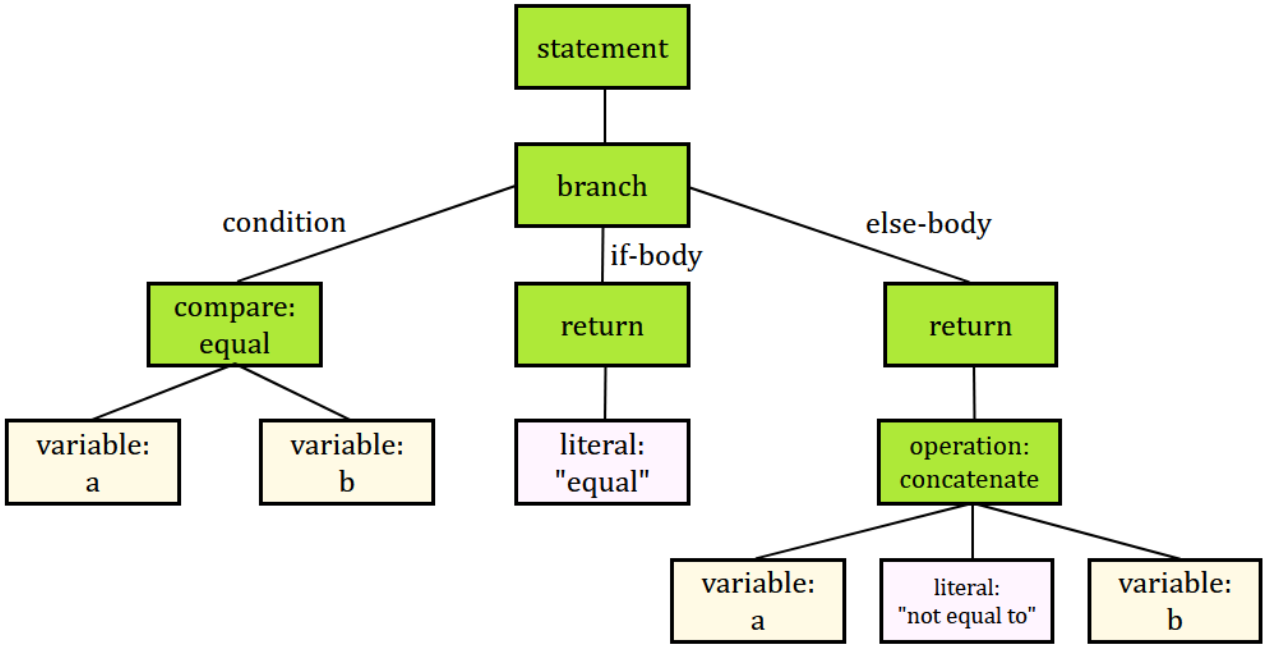
GRAPH MATCHING

- Newer
- More complex
- Type-1, Type-2, and Type-3 clones

Graphs for modeling source code

- AST (Abstract Syntax Tree)
- CFG (Control Flow Graph)
- PDG (Program Dependency Graph)

AST for the code : if a = b then return "equal" else return a + " not equal to " + b

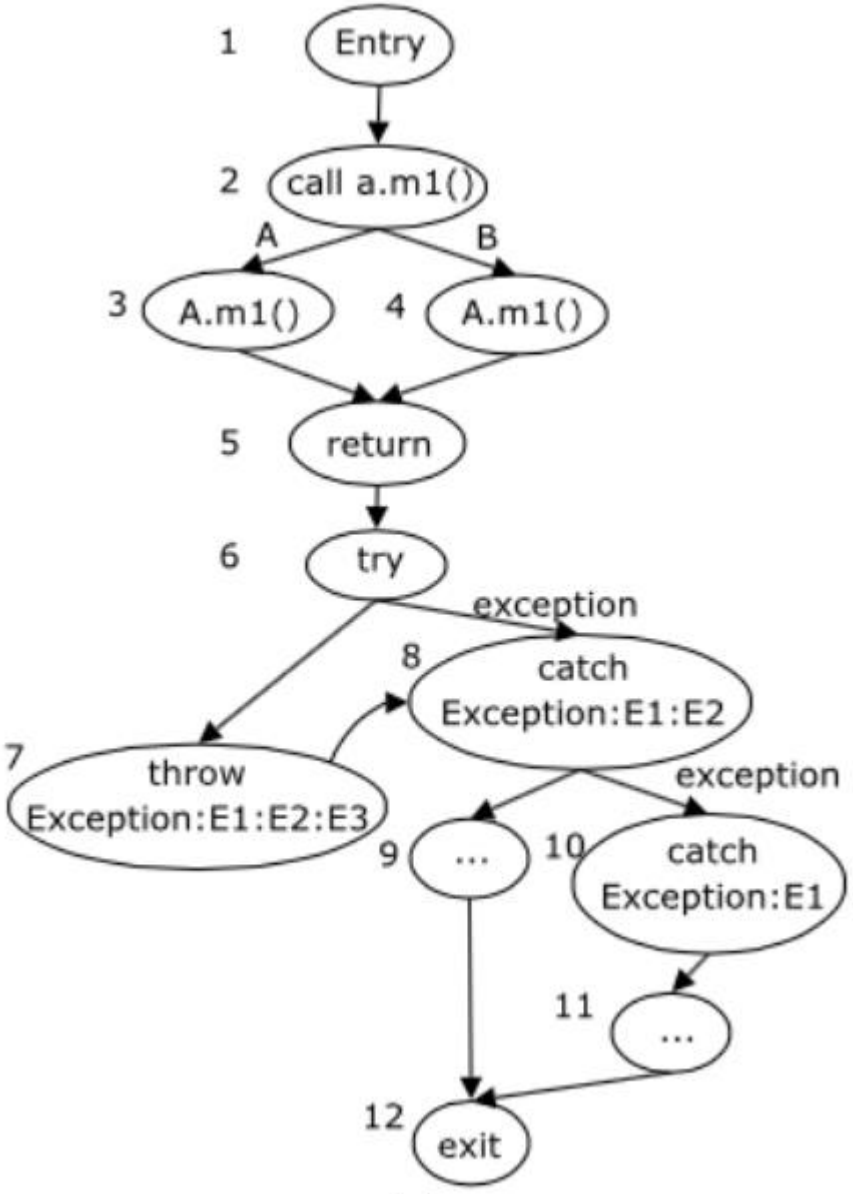


GRAPH MATCHING

- Newer
- More complex
- Type-1, Type-2, and Type-3 clones

Graphs for modeling source code

- AST (Abstract Syntax Tree)
- CFG (Control Flow Graph)
- PDG (Program Dependency Graph)

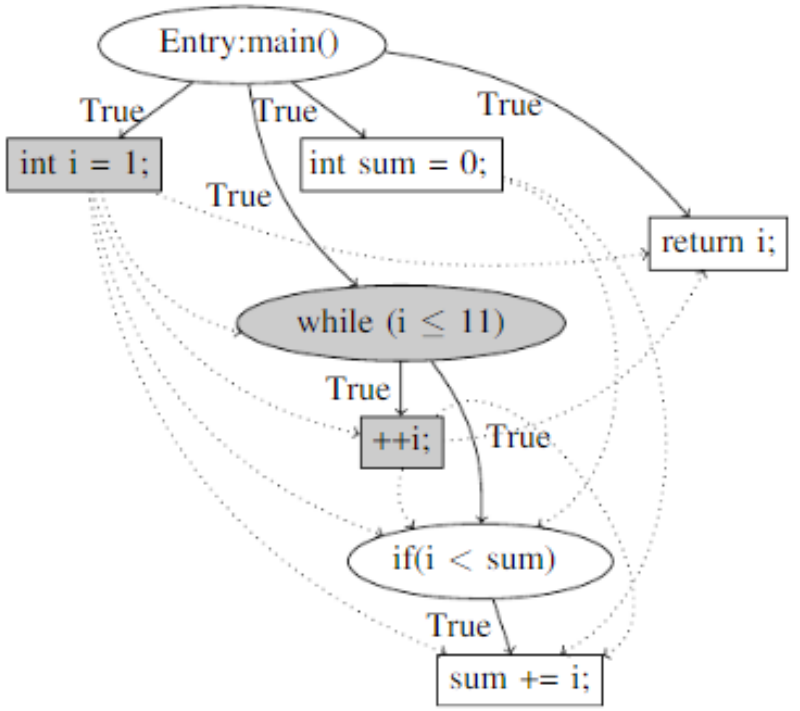


GRAPH MATCHING

- Newer
- More complex
- Type-1, Type-2, and Type-3 clones

Graphs for modeling source code

- AST (Abstract Syntax Tree)
- CFG (Control Flow Graph)
- PDG (Program Dependency Graph)




```
int main()
{
    int i = 1;
    int sum = 0;
    while(i < 11 ){
        if (i < sum)
            sum += 1;

        ++i;
    }
    return i;
}
```

https://www.researchgate.net/figure/Program-Dependence-Graph_fig1_271557302

(TRADITIONAL) CLONE DETECTION TECHNIQUES

Tool	*Proposed By*	*Language Supported*	*Technique*	*Application*
CloneDr	Baxter et. al., [3]	C, C++, Java and Cobol	Abstract Syntax Tree Method	Clone Detection
CCFinder	Kamiya et. al., [4]	C, C++, Java	Token Based Method	Clone Detection
CP-Miner	Li et. al., [5]	C, C++, Java	Frequent Subsequent Mining	Clone Detection and copy-pasted bug identification

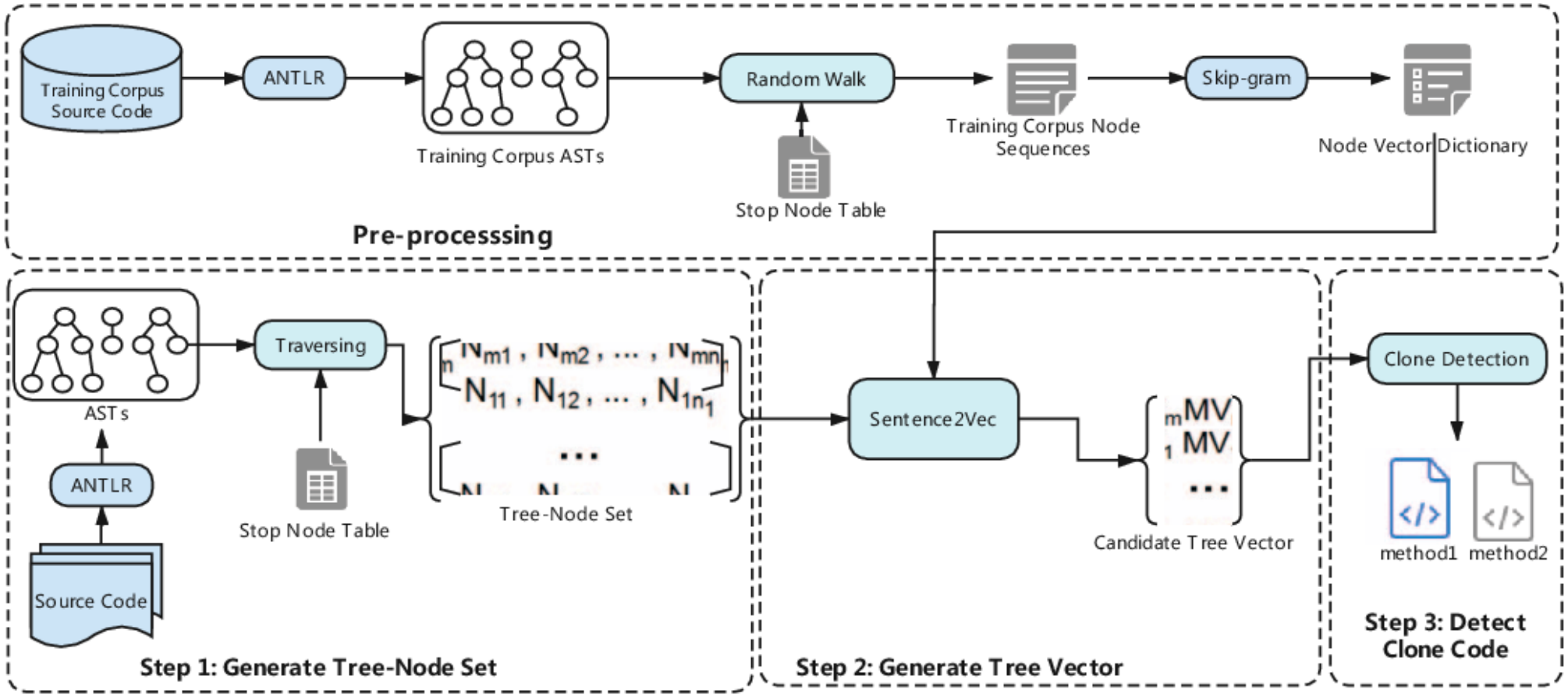
What about Type-4 (semantic) clones?

CCFnderX	Kamiya et. al., [9]	C, C++, Java, COBOL, VB, C#	Token Based Approach	Clone Detection
PMD	Sourcefourge community [10]	Java, C, C++, JSP, Ruby, PHP, PLSQL etc	String Matching	Clone Detection
PDG-Dup	Komondor et.al., [11]	C,C++	Program Dependence Graph	Clone Detection
Duplix	Krinke et. al., [12]	C	Program Dependence Graph	Clone Detection

https://www.researchgate.net/publication/275772216_Performance_Evaluation_of_Clone_Detection_Tools

1. Text matching 2. Token sequence matching 3. Graph matching

(SOTA) CLONE DETECTION TECHNIQUES



Gao, Yi et al. "TECCD: A Tree Embedding Approach for Code Clone Detection." 2019 (ICSME) (2019): 145-156.

RELIABILITY AND SECURITY

- Reliability is the ability to perform the operations consistently without any failures, every time it runs.
  - In reliability, you're identifying what each chunk of code expects to get, and then define how to handle exceptions, unexpected input.

- Security is the ability to defend against malicious threats (e.g., unauthorized use, information leak, attacks) in order to preserve system privacy, integrity and usability
  - In security, you're looking at the attacks, and making sure your code is secure against them.

INPUT VALIDATION

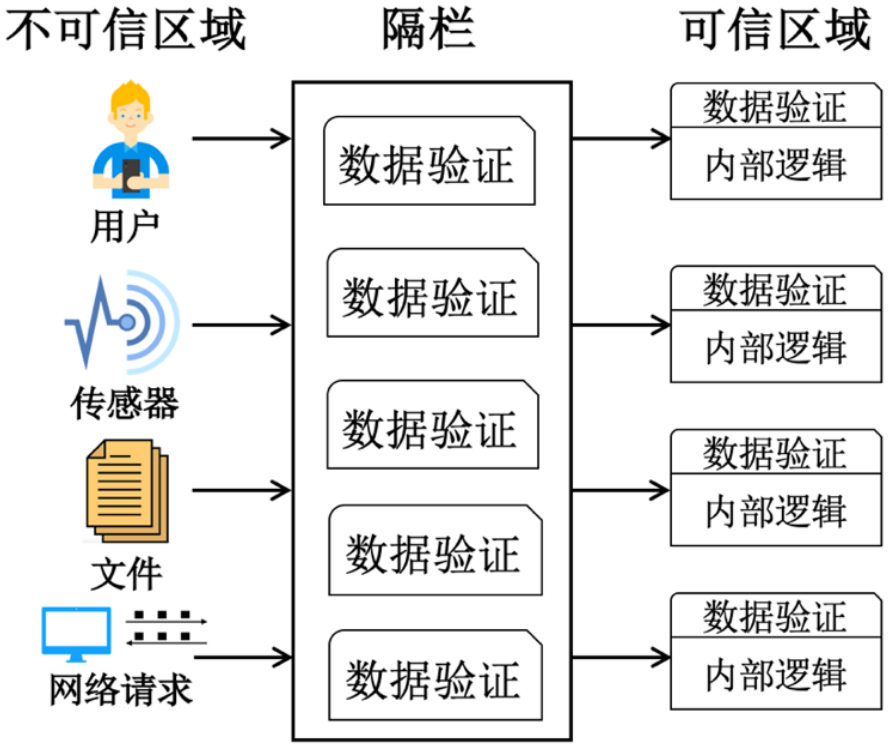
- Input validation is performed to ensure only properly formed data is entering the workflow in an information system, preventing malformed data from persisting in the database and triggering malfunction of various downstream components.

- Input validation should happen as early as possible in the data flow, preferably as soon as the data is received from the external party.

INPUT VALIDATION

Untrusted region (outside)
- Out of developers' control
- Potentially malformed or malicious
- Much more restrict requirements for input validation

Barricade: Establish "trust boundaries" — everything outside of the boundary is dangerous, everything inside of the boundary is safe. In the barricade code, validate all input data.



Trusted region (inside)
- Controlled by developers
- Less likely to be exposed to malformed or malicious input
- May be affected by internal defect or expected exceptions

ERROR HANDLING

- Error handling refers to the response and recovery procedures from error conditions present in a software application.
- Proper error handling improves software reliability and maintainability
  - Reliability: proper error handling ensures the normal execution of the program
  - Maintainability: proper error handling logs error conditions, which facilitate debugging and fixing.

STRATEGIES OF ERROR HANDLING

- Maintaining normal service
  - Retry
  - Replacing invalid input with neutral values (e.g., 0), previous input values, or similar valid values
  - Invoking standard error-handling subroutine

- Logging, informing, and terminating
  - Writing error info to log files
  - Return error code or display error message
  - Terminating the program

CHOOSING PROPER ERROR HANDLING

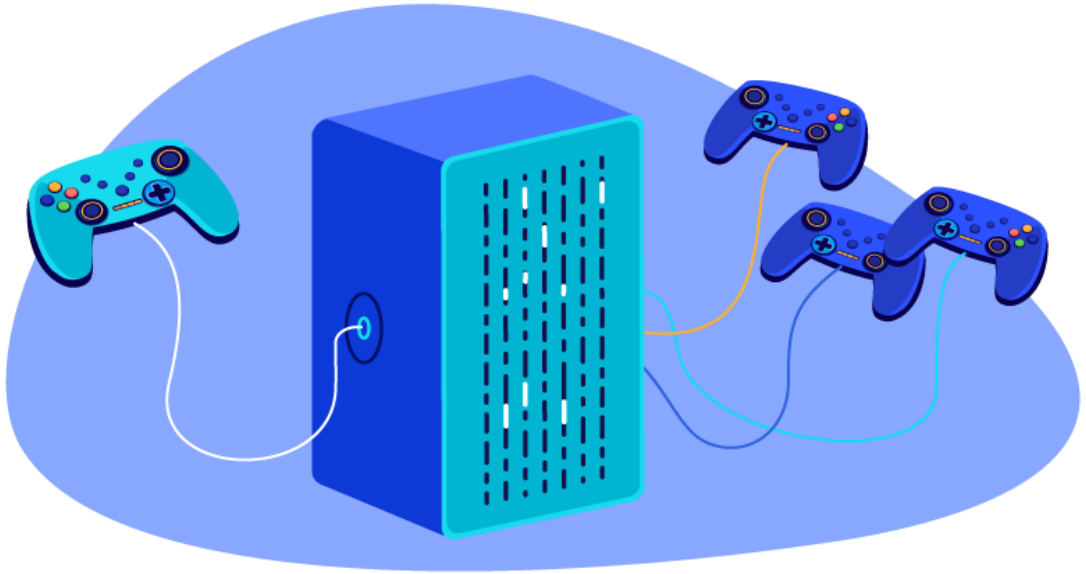
- **A safety-critical system (SCS)** or life-critical system is a system whose failure or malfunction may result in:
  - death or serious injury to people
  - loss or severe damage to equipment/property
  - environmental harm

- For SCS, error handling should value ***software correctness: the software could terminate, but should never return incorrect results.***

CHOOSING PROPER ERROR HANDLING

- **A mission-critical system (MCS)** must function continuously in order for a business to be successful.
- If a mission-critical application experiences even brief downtime, the negative consequences are likely to be financial.

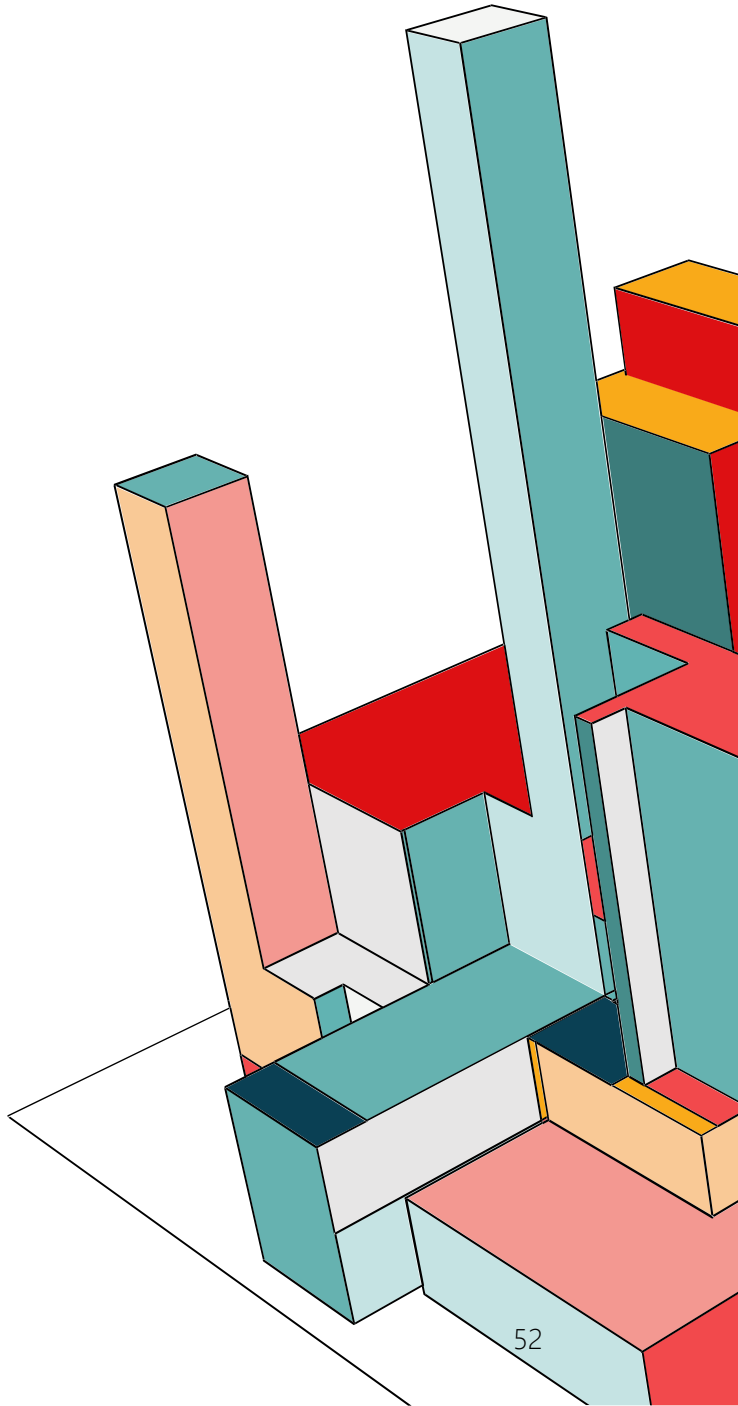
- For MCS, error handling should value ***software robustness: the software could tolerate certain errors, but should not terminate***



EFFICIENCY

- The degree to which the software makes optimal use of system resources

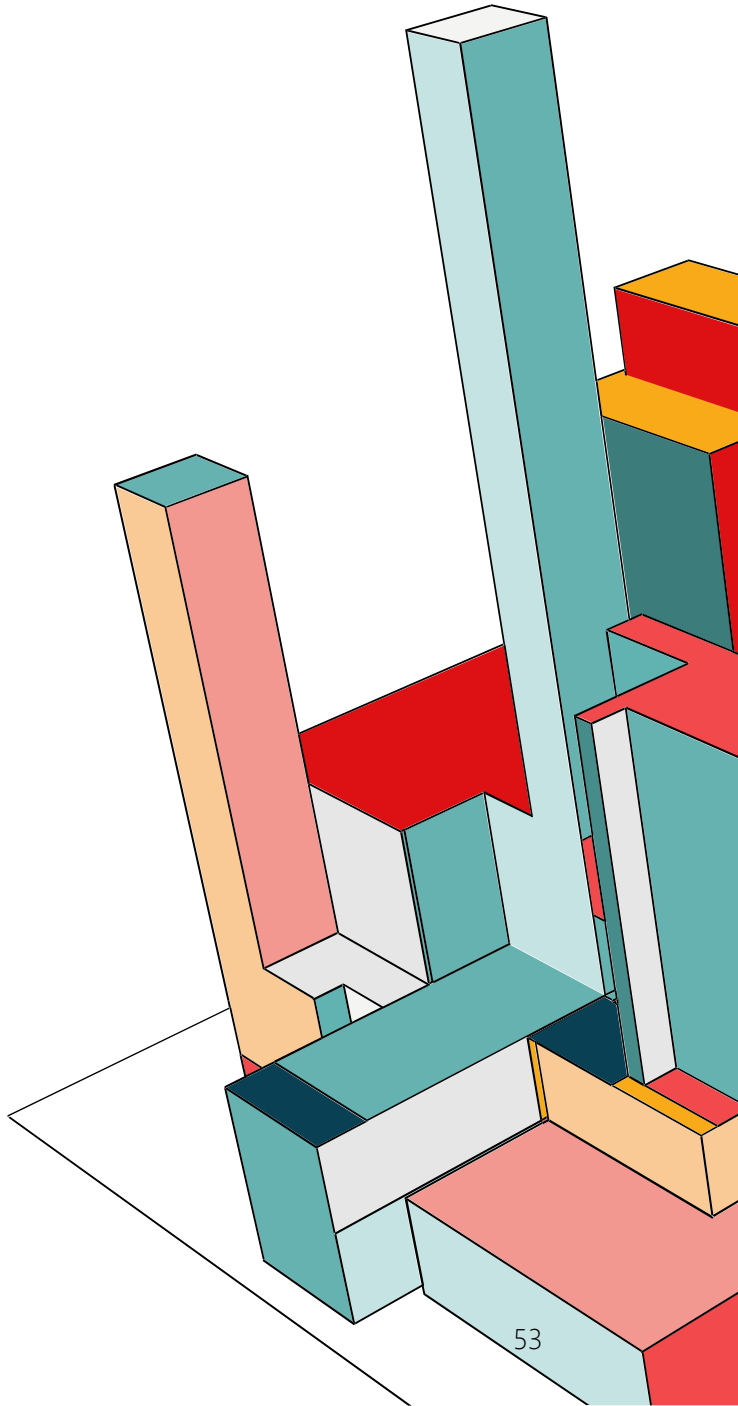
- Efficiency is indicated by the following subattributes:
  - time behavior
  - resource behavior



PORTABILITY

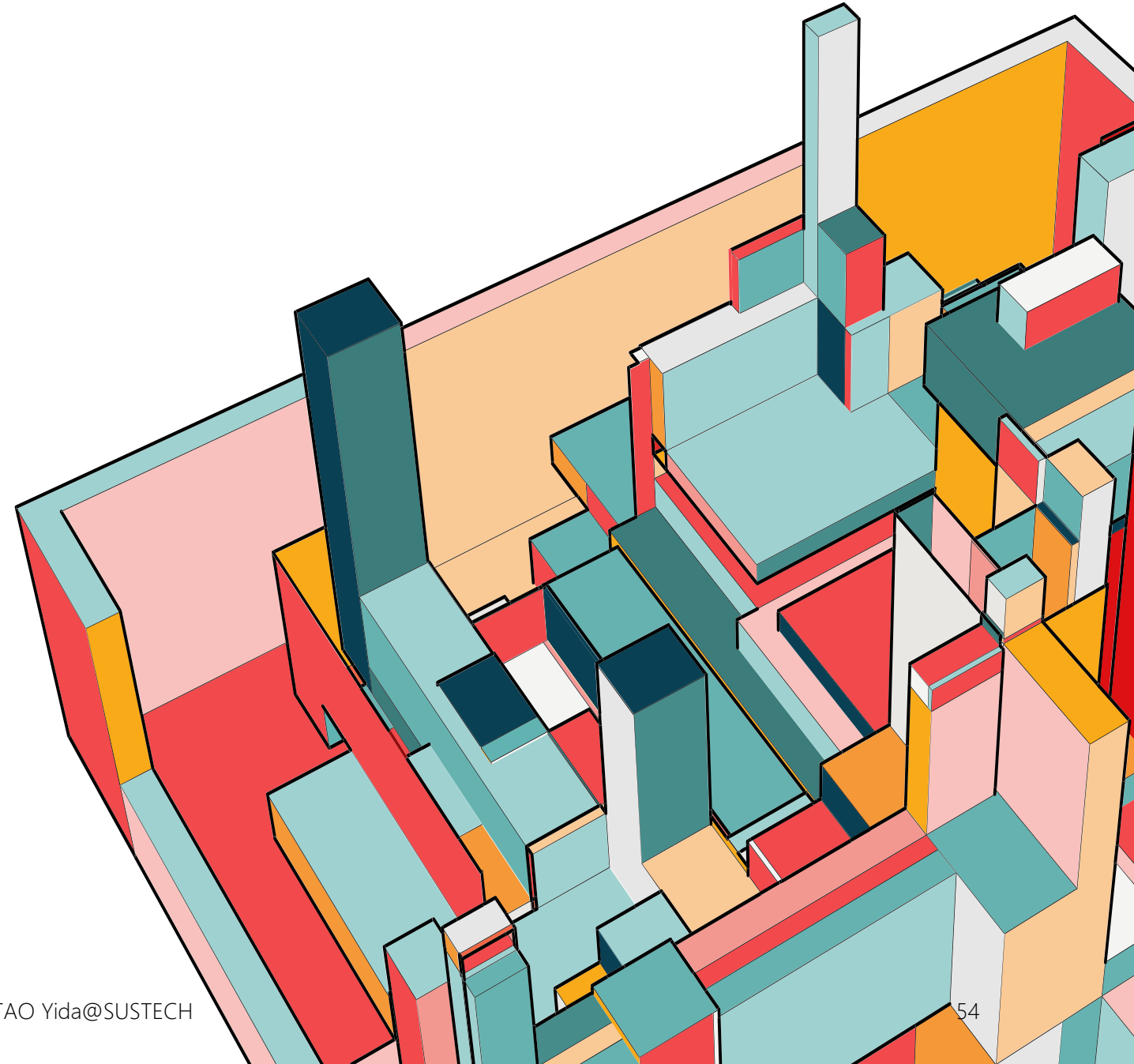
- The ease with which the software can be transposed from one environment to another

- Example
  - (Better portability) A Java program that only uses core java APIs
  - (Decreased portability) A Java program that uses native APIs to invoke certain Windows OS services



LECTURE 7

- Code quality
- Code review



WHAT IS CODE REVIEW?

Code review (sometimes referred to as peer review) is a software quality assurance activity in which one or several people check a program mainly by viewing and reading parts of its source code (changes/commits)

https://smartbear.com/learn/code-review/what-is-code-review/

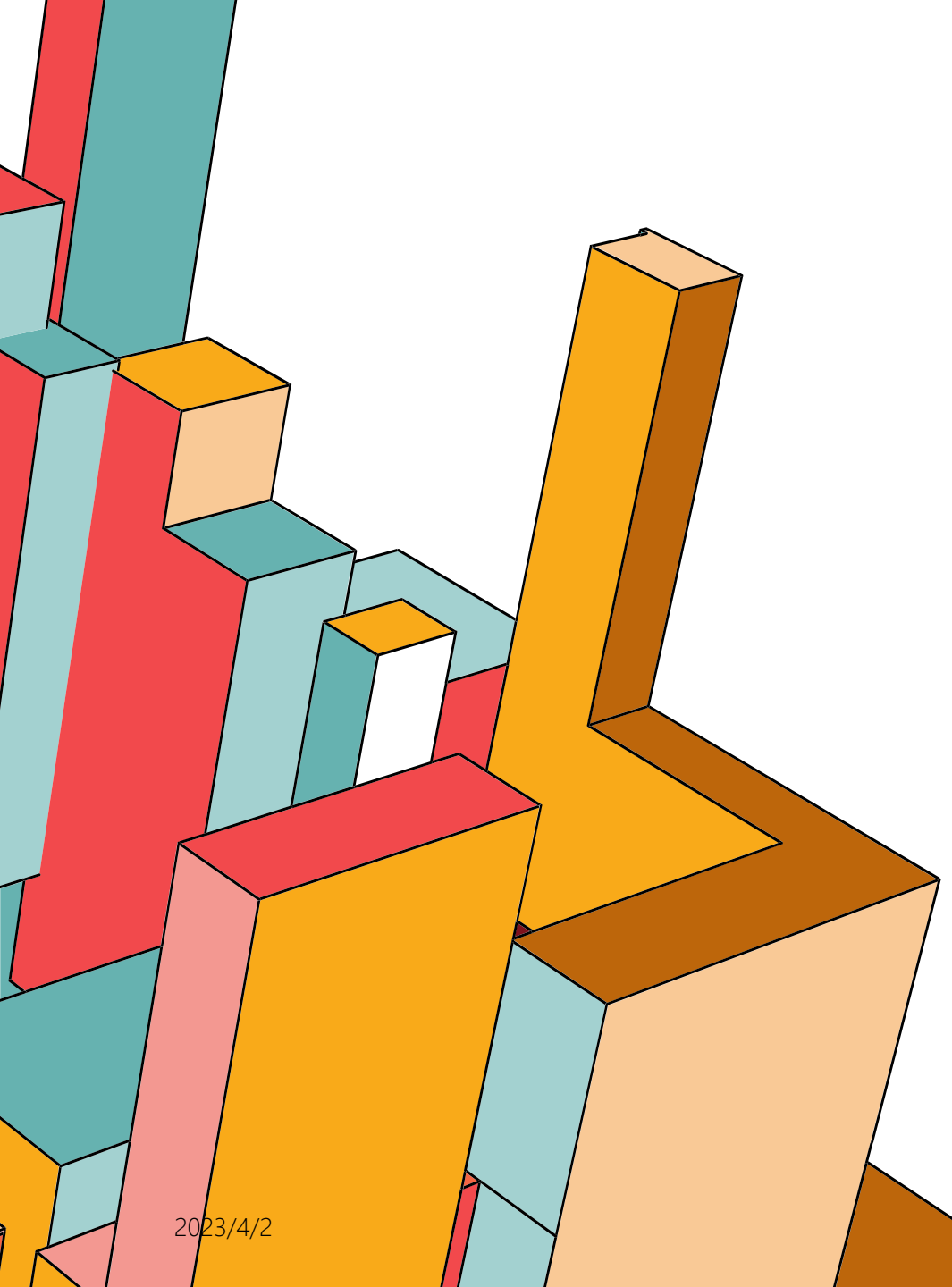
TYPES OF CODE CHANGES TO BE REVIEWED

- Modifications to existing code
  - Behavioral changes, improvements, optimizations
  - Bug fixes and rollbacks
- Entirely new code
  - Greenfield code review: review entirely new code
- Automatically generated code
  - E.g., refactoring tools, intelligent code generators

WHY CODE REVIEW?

- Across the industry, code review is a well recognized, important practice
- Many companies and open-source project have enforced some form of code review
  - Code review is a mandate process in which all software engineers at Google must participate
  - Google engineers spent a large chunk of time in code review

- But code review slows down development and incurs additional cost. So why code review?



CODE REVIEW BENEFITS

A well-designed code review process and a culture of taking code review seriously provides the following benefits

- Code correctness
- Code readability
- Code consistency
- Knowledge sharing
- ……

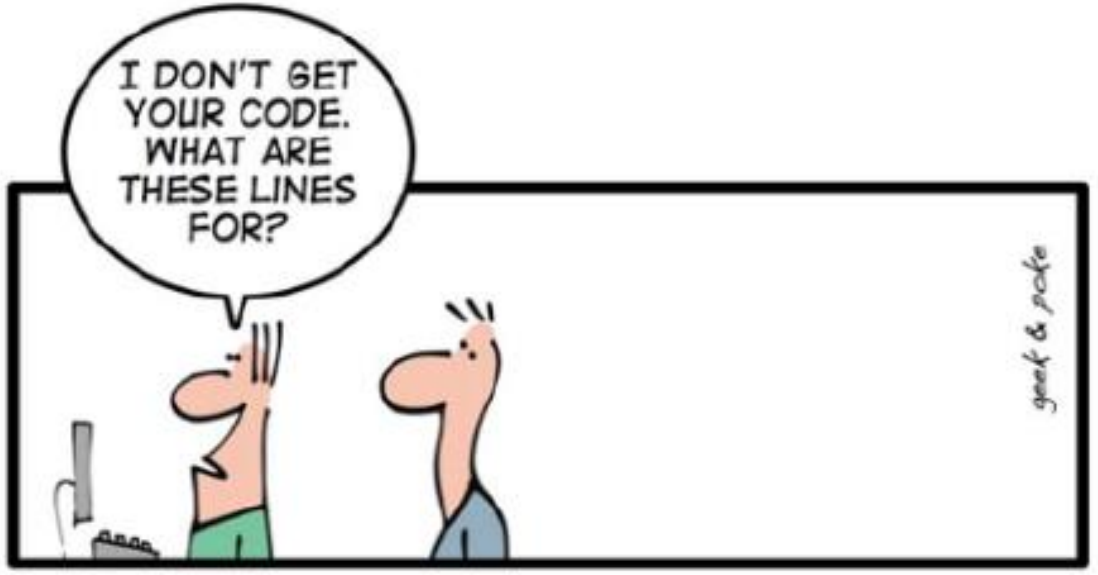
CODE CORRECTNESS

- Reviewers typically look for whether a code change
  - was intended
  - was properly designed
  - functions correctly and efficiently
  - has proper testing
  - introduces bugs into the codebase

Many correctness checks are performed automatically through techniques such as static analysis and automated testing

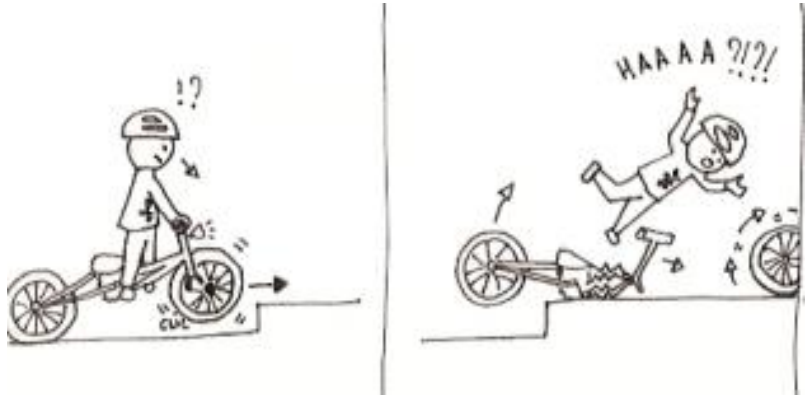
CODE READABILITY

- Code might be written once, but it will be read dozens, hundreds, or even thousands of times
- A code review is often the first test of whether a given code change is understandable to a broader audience



CODE CONSISTENCY

- Code needs to conform to some standards of consistency so that it can be understood and maintained
- Reviewers should also check
  - Whether the code change follows best practices for the particular programming language
  - Whether the code change is consistent with the particular codebase/repository



KNOWLEDGE SHARING

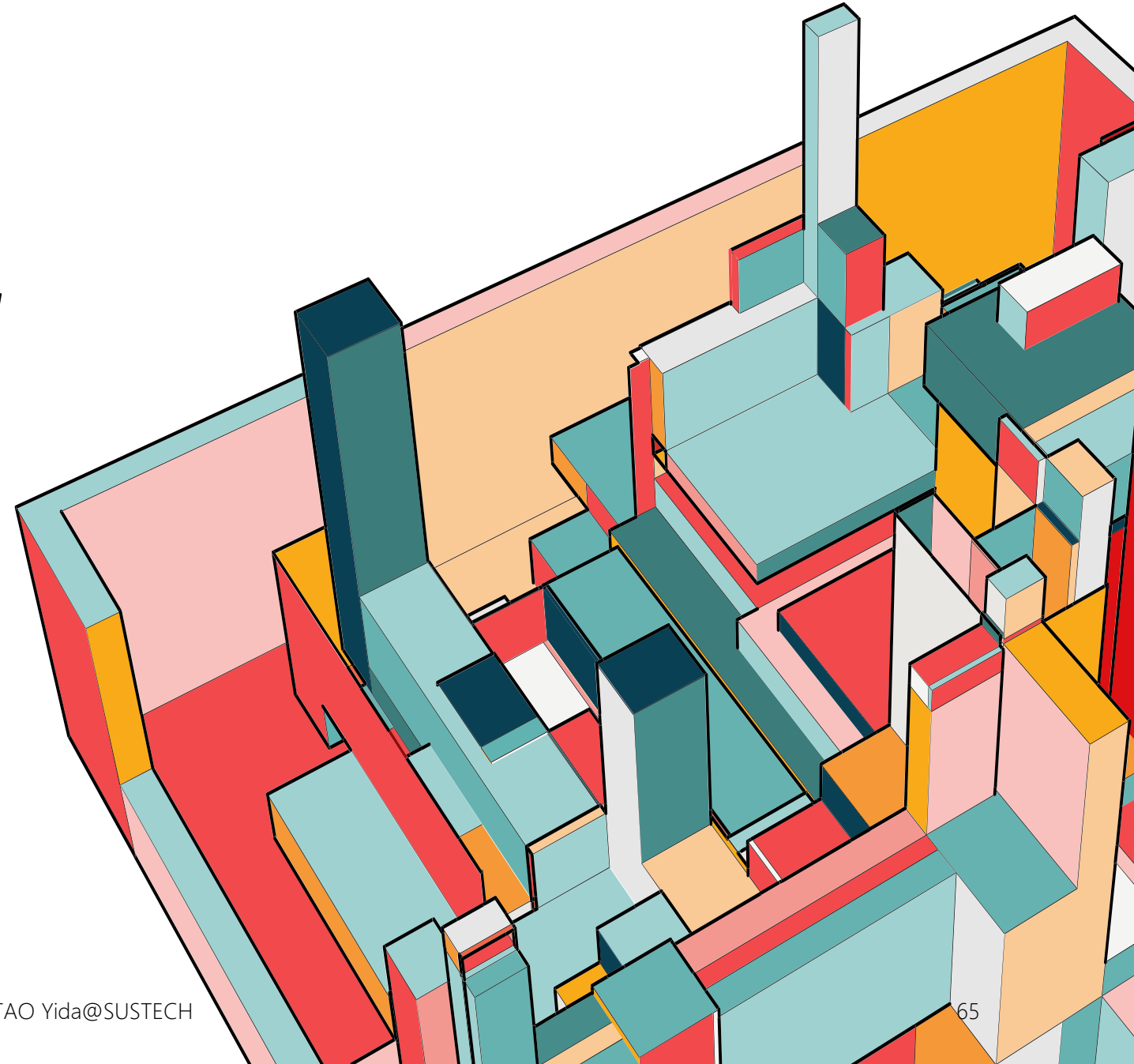
- Part of the code review process of feedback and confirmation involves asking questions on why the change is done in a particular way; this exchange of information facilitates knowledge sharing

- Both the authors and reviewers can learn new techniques and patterns from code review
    - Many engineers at Google “meet” other engineers first through their code reviews!

CODE REVIEW FLOW

- Code reviews can happen at many stages of software development
- Code review processes also differ from organizations.

- We'll look at the code review flow at Google



CODE REVIEW FLOW AT GOOGLE

- At Google, code reviews take place before a change can be committed to the codebase; this stage is also known as a *precommit review*.
- The primary end goal of a code review is to get another engineer to consent to the change, which is denoted by tagging the change as “looks good to me” (LGTM).
- This LGTM is used as a necessary permissions “bit” to allow the change to be committed.

Code Review Flow at Google - Step 1

- A user writes a change to the codebase in their local workspace.
- This *author* then creates a snapshot of the change: a patch and corresponding description that are uploaded to the code review tool (*Critique* at Google).
- This change produces a *diff* against the codebase, which is used to evaluate what code has changed.

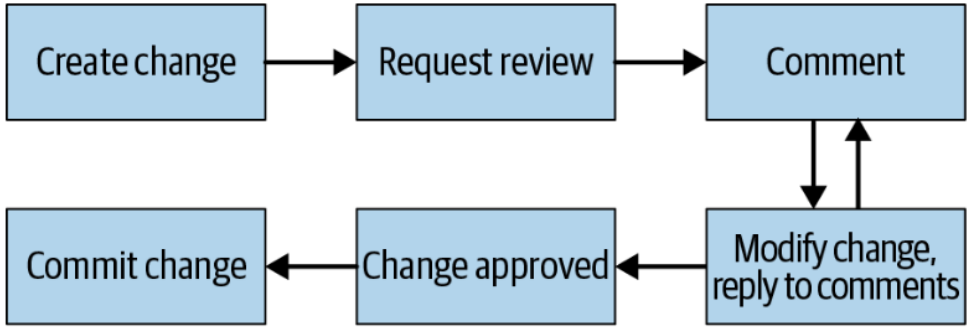


Figure 19-1. The code review flow

CODE REVIEW FLOW AT GOOGLE - STEP 1

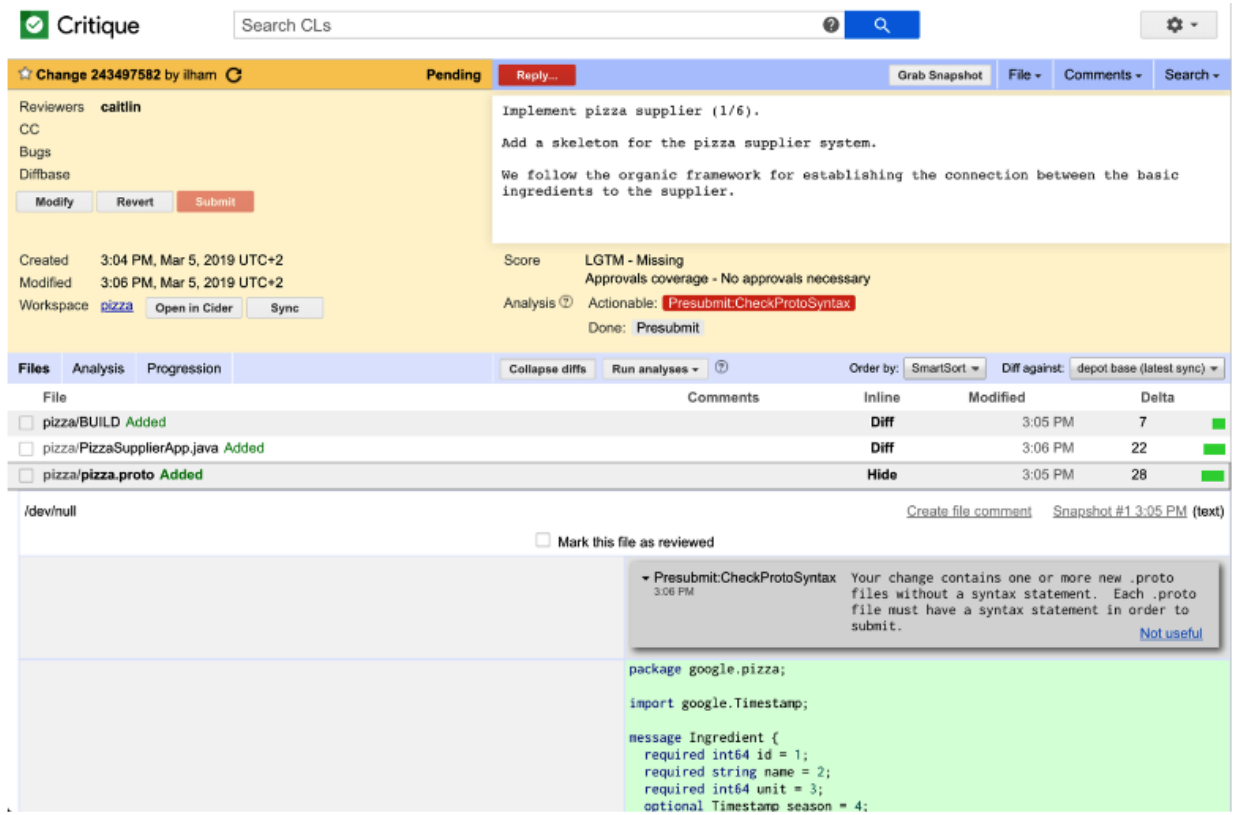
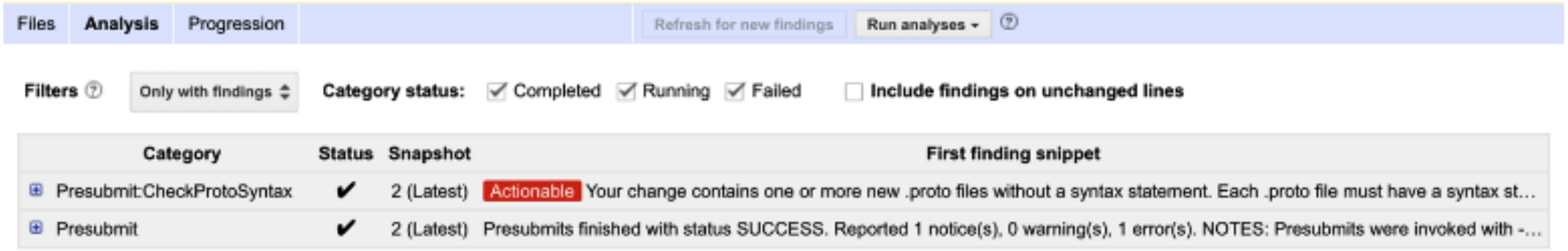


Figure 19-3. Change summary and diff view

CODE REVIEW FLOW AT GOOGLE - STEP 1

- Uploading a snapshot of the change triggers automated code analyzers (e.g., checking code styles)
- The author can preview the fix suggestion (e.g., remove the extra spaces) and apply the fix on the change.



Code Review Flow at Google - Step 2

- When the author is satisfied with the diff of the change, they mail the change to one or more reviewers.
- This process notifies those reviewers, asking them to view and comment on the snapshot.

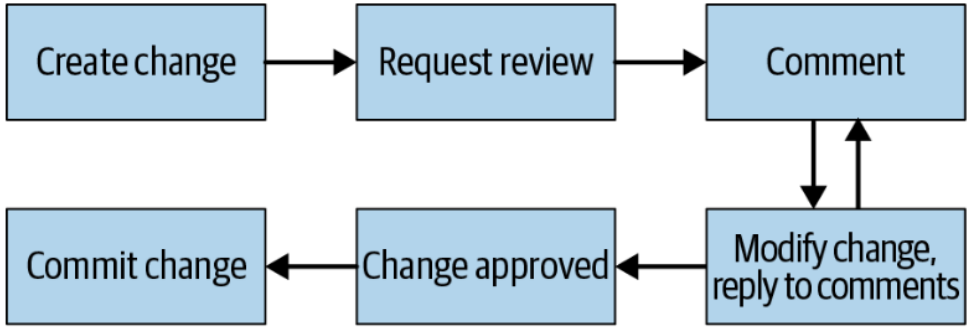
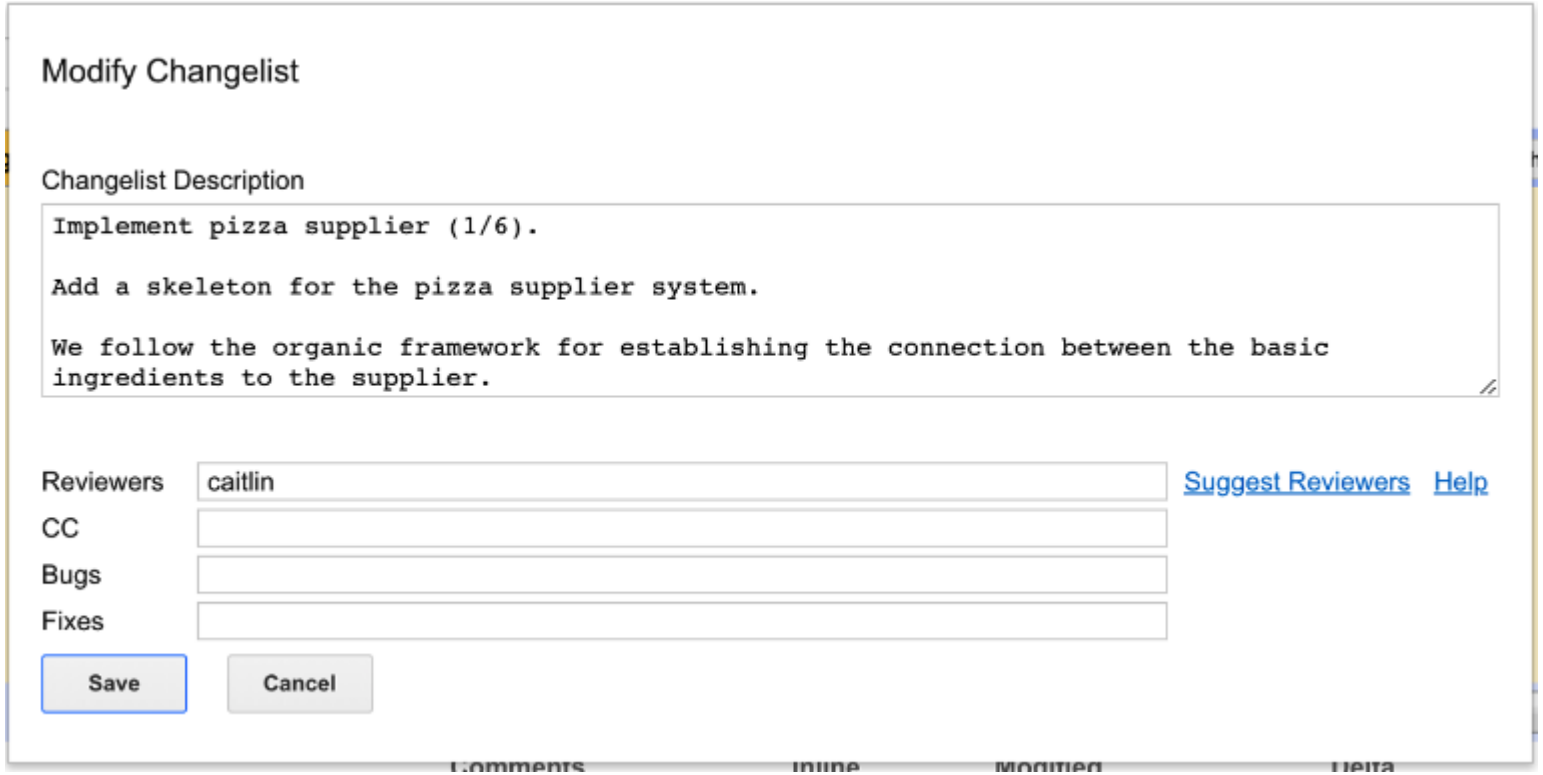


Figure 19-1. The code review flow

CODE REVIEW FLOW AT GOOGLE - STEP 2



Code Review Flow at Google - Step 3

- Reviewers open the change in the code review tool and post comments on the diff.
- Some comments request explicit resolution. Some are merely informational.

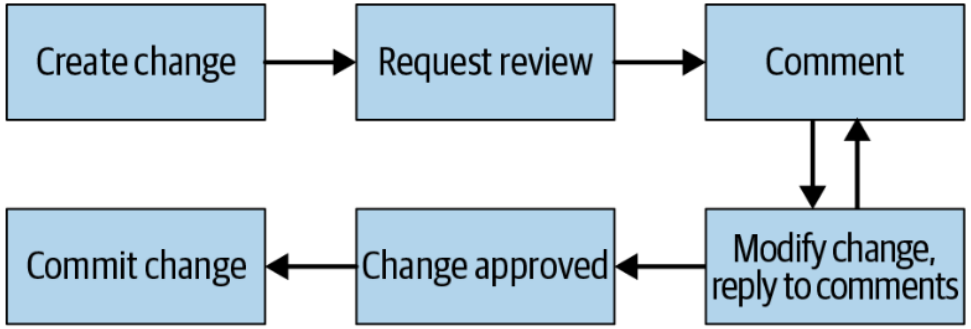


Figure 19-1. The code review flow

CODE REVIEW FLOW AT GOOGLE - STEP 3

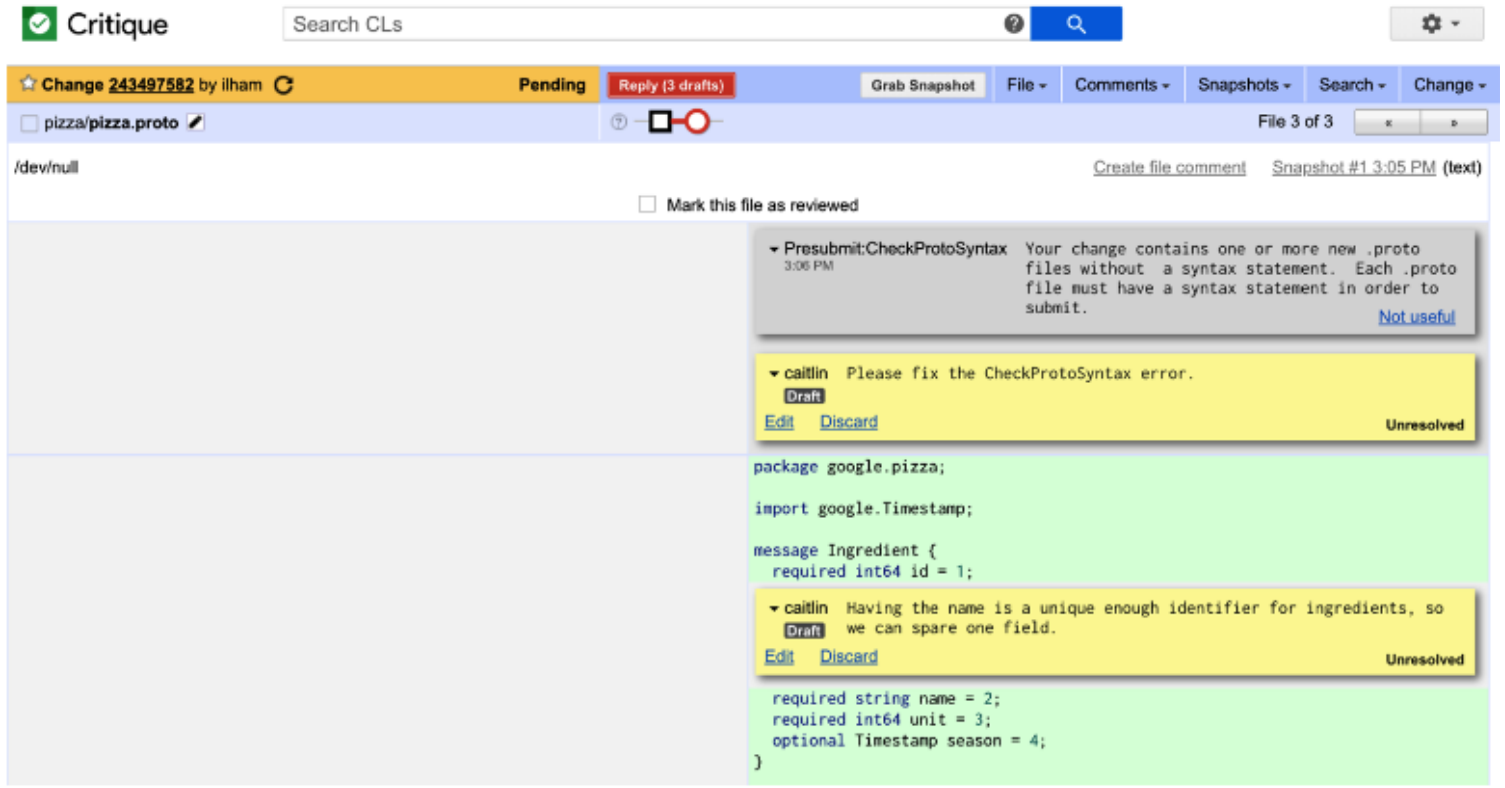


Figure 19-6. Commenting on the diff view

Code Review Flow at Google - Step 4

- The author modifies the change and uploads new snapshots based on the feedback and then replies back to the reviewers. Steps 3 and 4 may be repeated multiple times.

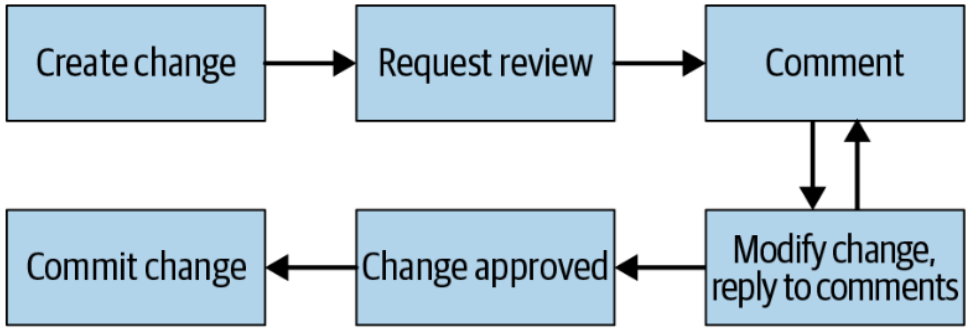


Figure 19-1. The code review flow

Code Review Flow at Google - Step 5

- After the reviewers are happy with the latest state of the change, they agree to the change and accept it by marking it as "looks good to me" (LGTM).
- Only one LGTM is required by default

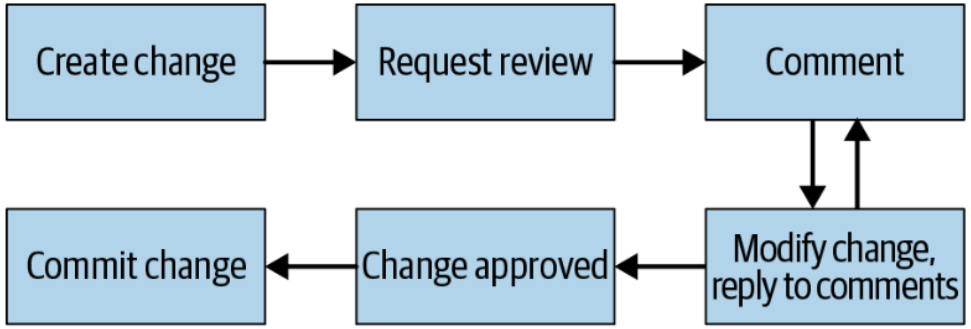


Figure 19-1. The code review flow

Code Review Flow at Google - Step 6

- After a change is marked LGTM, the author is allowed to commit the change to the codebase, provided they resolve all comments and that the change is approved.

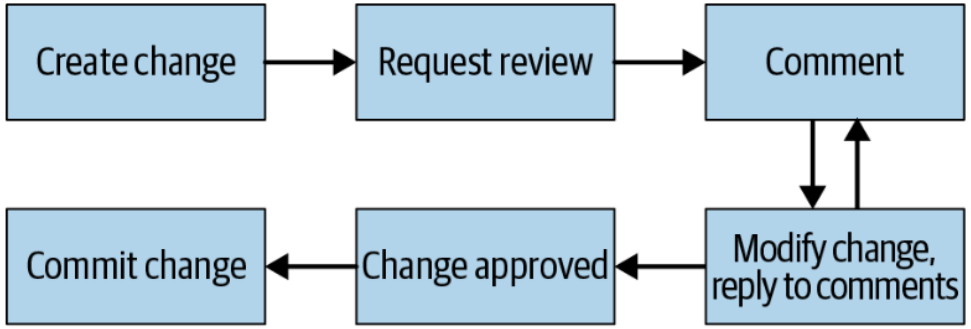
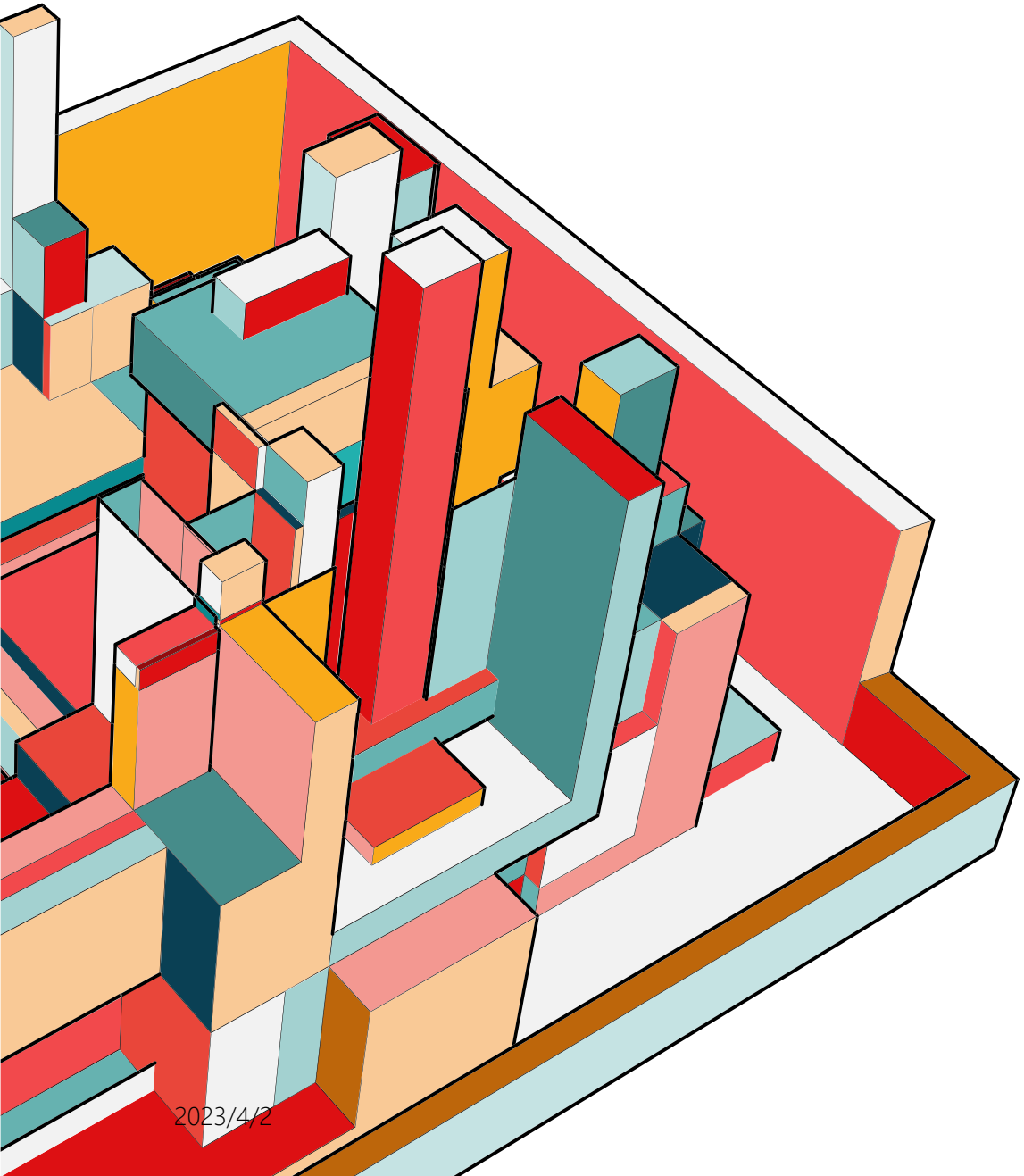


Figure 19-1. The code review flow



CODE REVIEW BEST PRACTICES

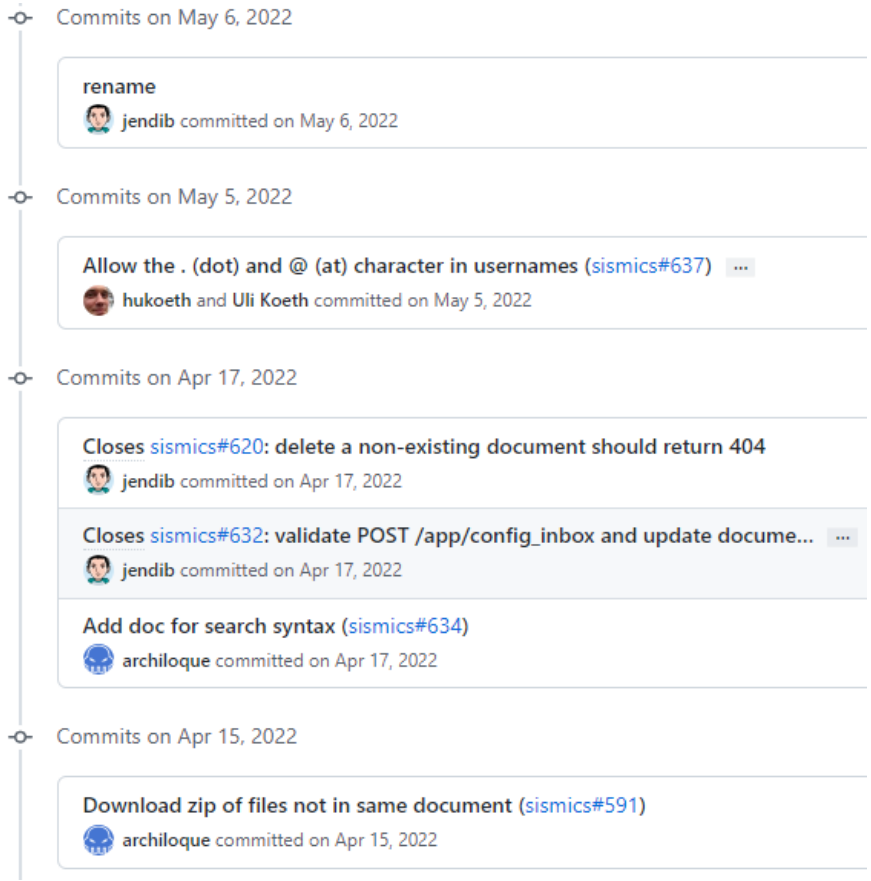
Most of the practices emphasize keep the code review process nimble and quick, so that code review can scale properly

Code Review Best Practices - Write Small Changes

- Small changes should generally be limited to ~200 LoC
- Small code changes are easy to digest and tend to focus on a single issue
  - 35% of the changes at Google are to a single file
- Small code changes allow quick approvals and quicker changes to the codebase
  - Most changes at Google are expected to be reviewed within a day

CODE REVIEW BEST PRACTICES - WRITE GOOD CHANGE DESCRIPTIONS

- A change description should
  - summarize the change
  - Explain what is changed and why
  - E.g., “bug fix” is not a good change description
- A good change description also allows Code Search tools to find changes



CODE REVIEW BEST PRACTICES

- Keep reviewers to a minimum
  - Most code reviews at Google involve only 1 reviewer
  - The cost of additional reviewers quickly outweighs their value (the most important LGTM is the first one)

- Automate where possible
  - Rather than require authors to run tests, static analysis tools, or formatters, modern code review tools typically automate these processes
  - The tools detect a variety of problems with the submitted change, reject the change and ask the author to fix the problems (see CI/CD, discussed later)

NEXT

- Measurements & Metrics