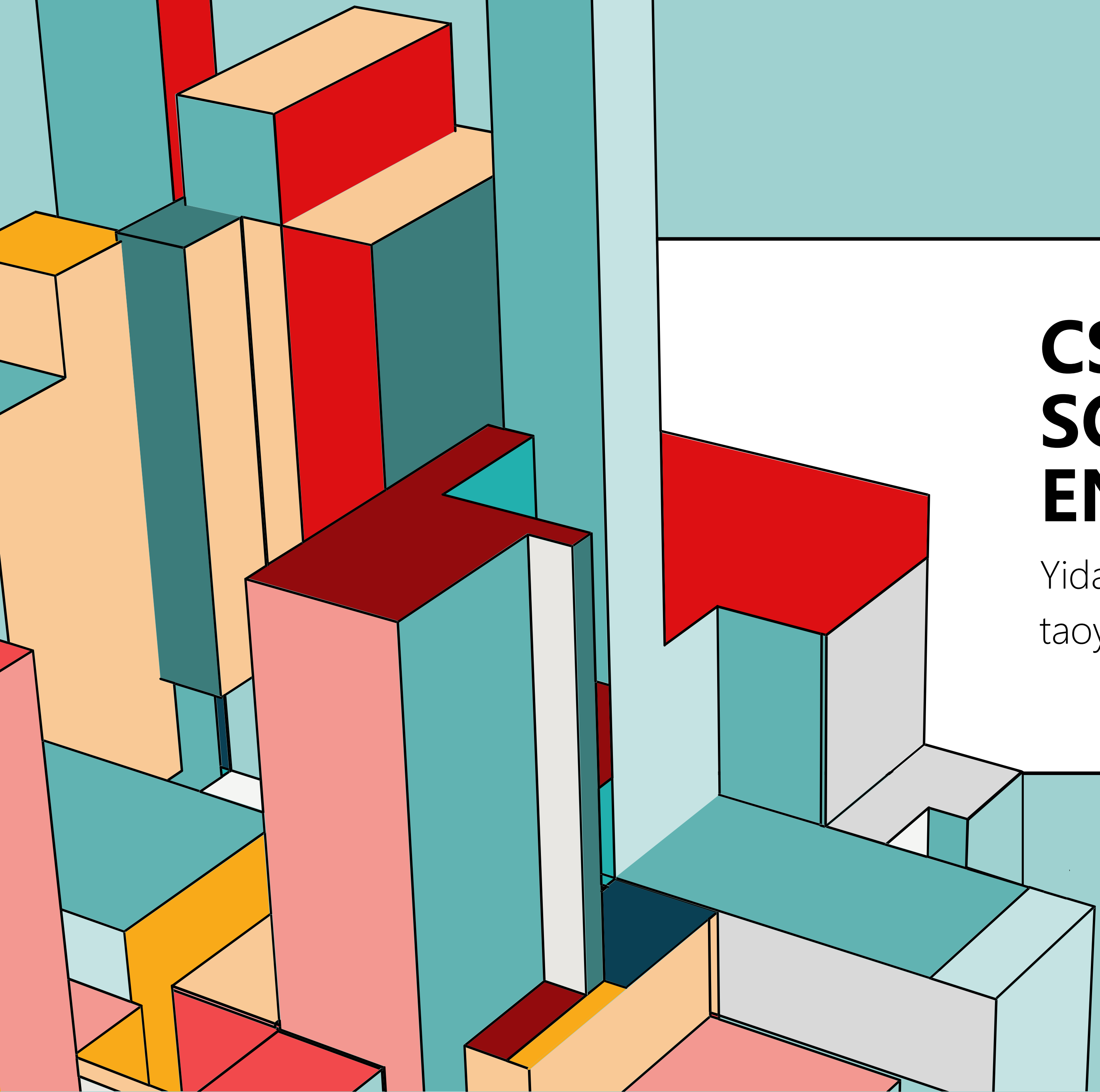


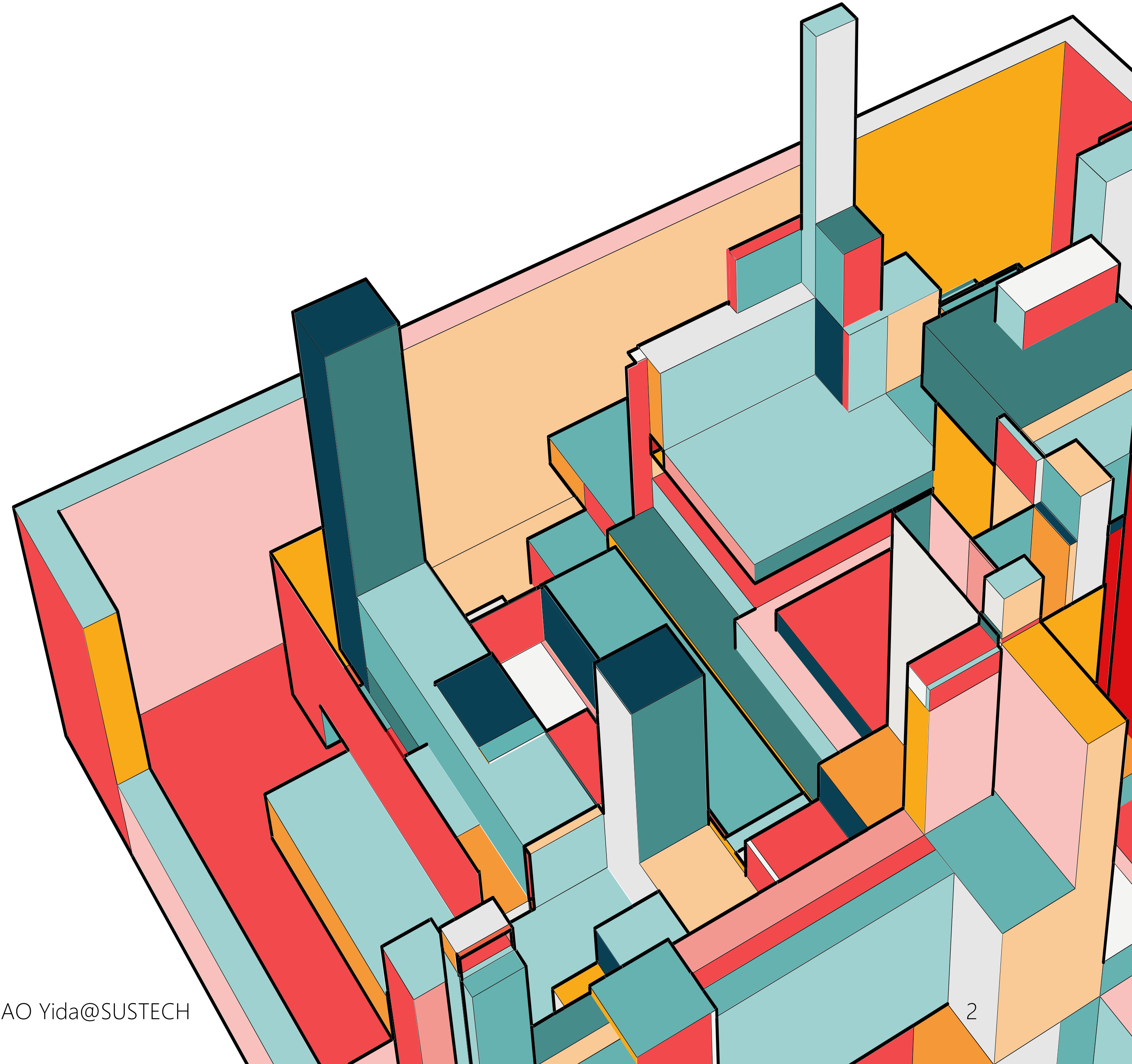
CS304 SOFTWARE ENGINEERING

Yida Tao

taoyd@sustech.edu.cn

LECTURE 9

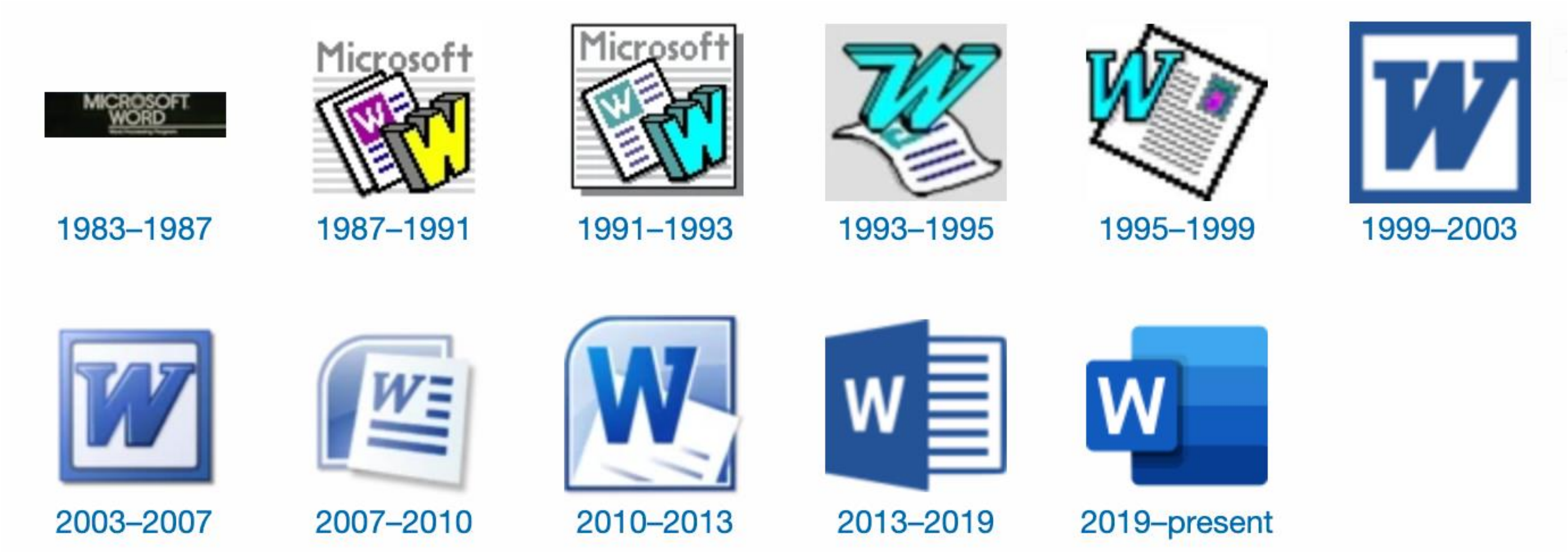
- Software evolution
  - Legacy systems
  - Deprecation
- Software maintenance
  - Reengineering
  - Refactoring



SOFTWARE EVOLUTION AND MAINTENANCE

- Changes drive software evolution (演化) and maintenance (维护)
- Changes occur when
  - Errors and inefficiencies are detected
  - Software is adapted to a new environment
  - Clients request for new features
  - ……

THE EVOLUTION OF MICROSOFT WORD

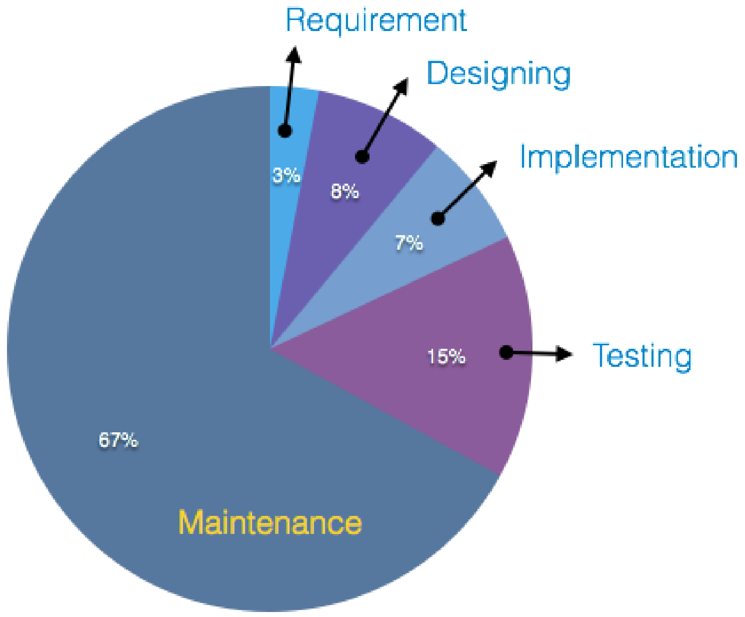


https://www.versionmuseum.com/history-of/microsoft-word

SOFTWARE EVOLUTION AND MAINTENANCE

- Research suggests that 85–90% of organizational software costs are evolution costs.

- Other surveys suggest that about two-thirds of software costs are evolution costs.



https://bcastudyguide.com/unit-5-software-maintenance/

WHY COSTLY?

- Software needs to **migrate** to new platforms, adjust for different machines and OS, and meet new use requirements
- As software grows, **complexity grows**; more changes result in poor designed structures, poor coding logic, and poor documentations
- People come and go as software evolves; costly to get **newcomers** familiar with the software

LEGACY SYSTEM

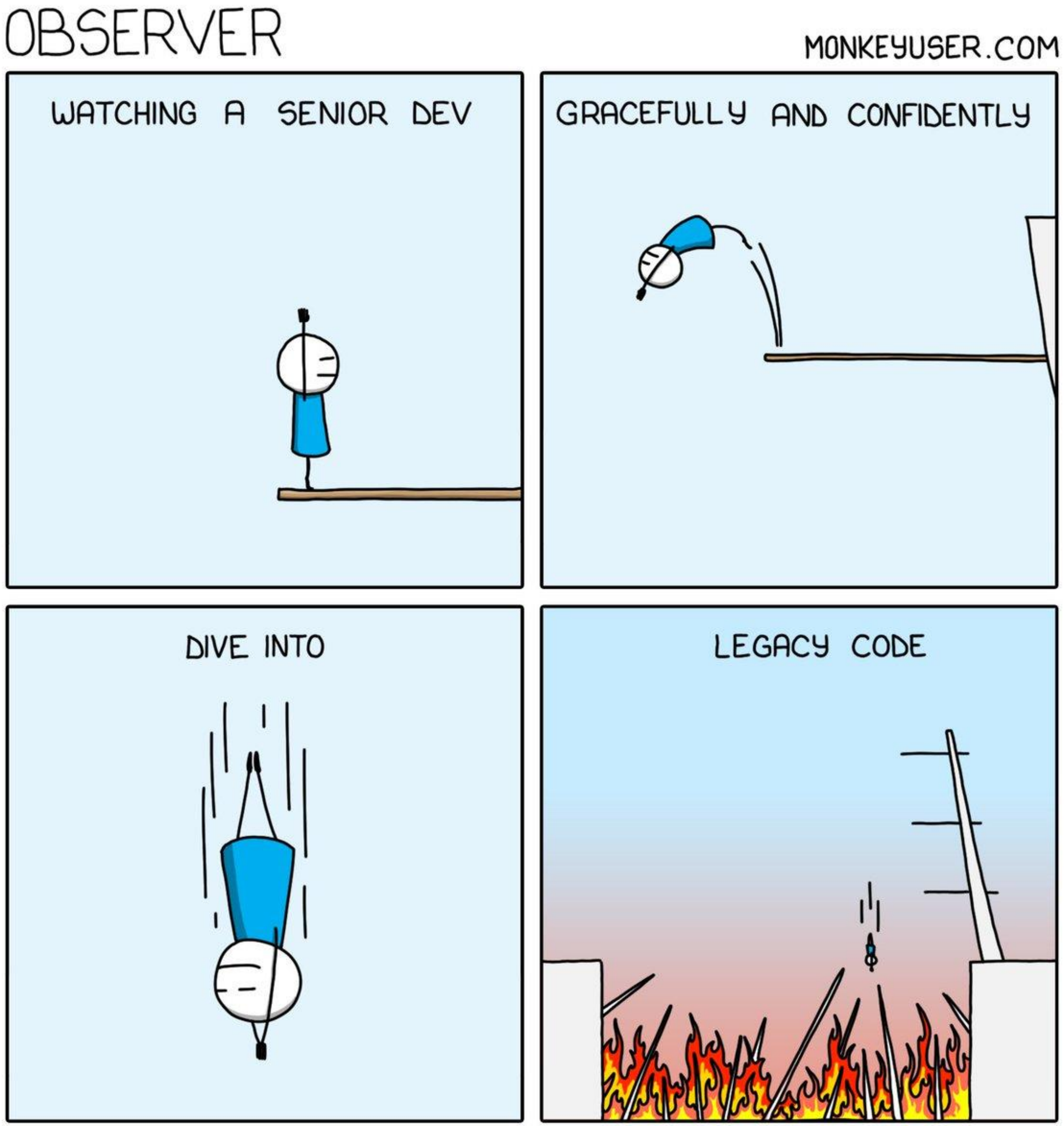
- Legacy systems (遗留系统) are outdated software or hardware that is still in use despite being replaced by newer technology
- Legacy software may depend on outdated hardware that is no longer produced or supported
- It is often difficult and expensive to maintain legacy system, which requires specialized skills that are no longer in demand

Although off-support since April 2014, Windows XP has endured continued use in fields such as ATM operating system software.

https://en.wikipedia.org/wiki/Legacy_system

LEGACY CODE （遗留代码）

- Old code
- Someone else’s code
- Code without tests
- Code without documentation
- Code that you’re afraid to change
- ……



LEGACY CODE （遗留代码）

- COBOL is a programming language designed for writing business systems
- It was the main business development language from the 1960s to the 1990s, particularly in the finance industry
- Industry has estimated that there are still more than 200 billion lines of COBOL code in current business systems.

Why not simply replace the legacy code?

TOO EXPENSIVE, TOO RISKY

- **Lack of** specification or documentation (lost or doesn't even exist)

- Important business rules may be **implicitly embedded** in software without any documentation

- Many years of maintenance **degrades** the system structure, making it increasingly difficult to understand or to extend

TOO EXPENSIVE, TOO RISKY

- System may be implemented using obsolete programming languages, old techniques, and adapted to older, slower hardware
- Hard to find people with required knowledge and expertise
- Data processed by the system may be out of date, inaccurate, incomplete, and depend on different database suppliers

REMEMBER HYRUM'S LAW?

- The more users of a system
  - The higher the probability that users are using it in unexpected and unforeseen ways
    - The harder it will be to change or remove such a system without affecting existing users

Every change breaks someone's workflow

DECISIONS FOR LEGACY SYSTEM

1. **Abandon** the system completely

2. Leave the system **unchanged** and continue with **regular maintenance**

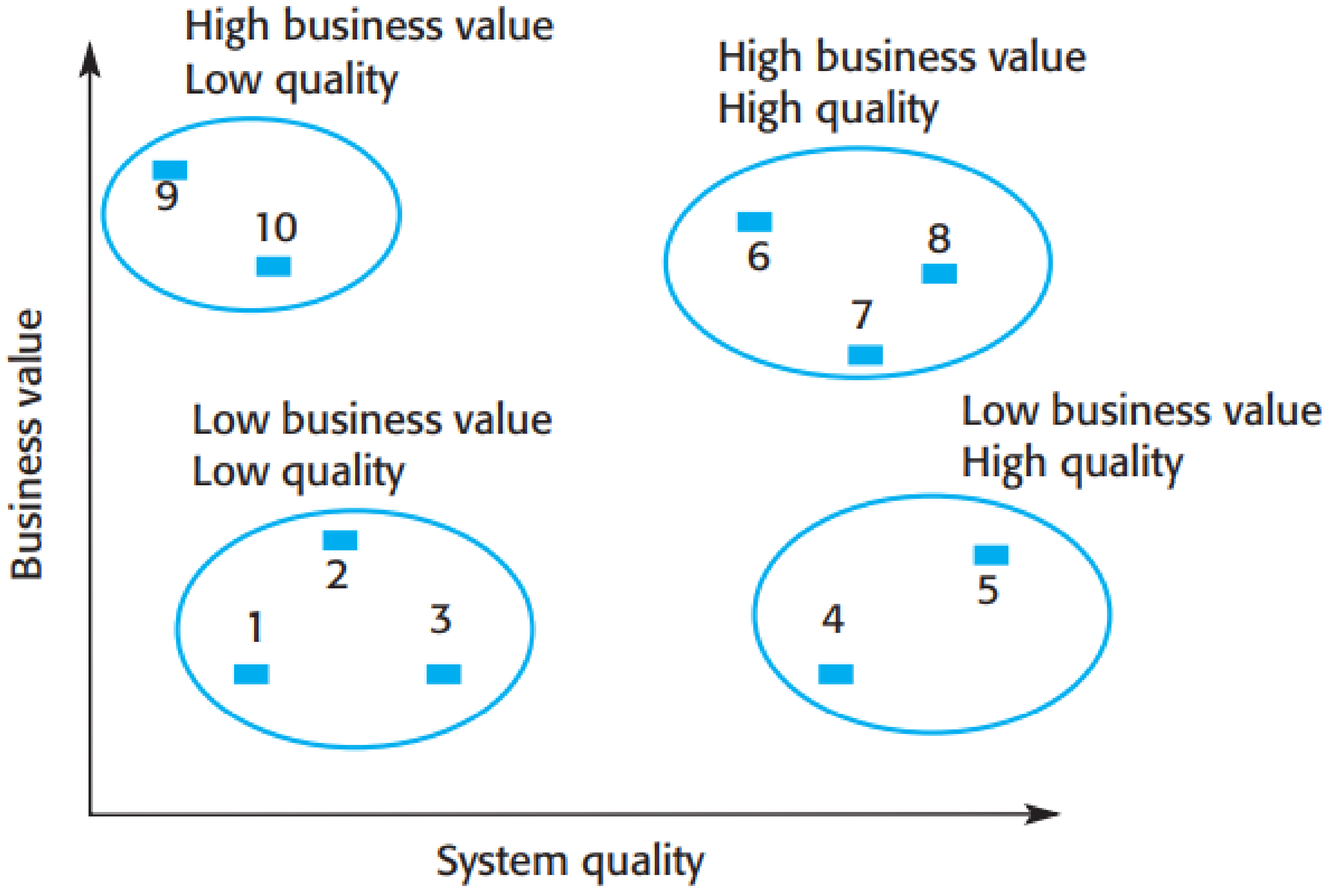
3. **Reengineer** the system to improve its maintainability

4. **Replace** part or all of the system with a new system

WHICH DECISION TO MAKE?

Suppose we have 10 legacy systems.
We'll evaluate them in terms of:

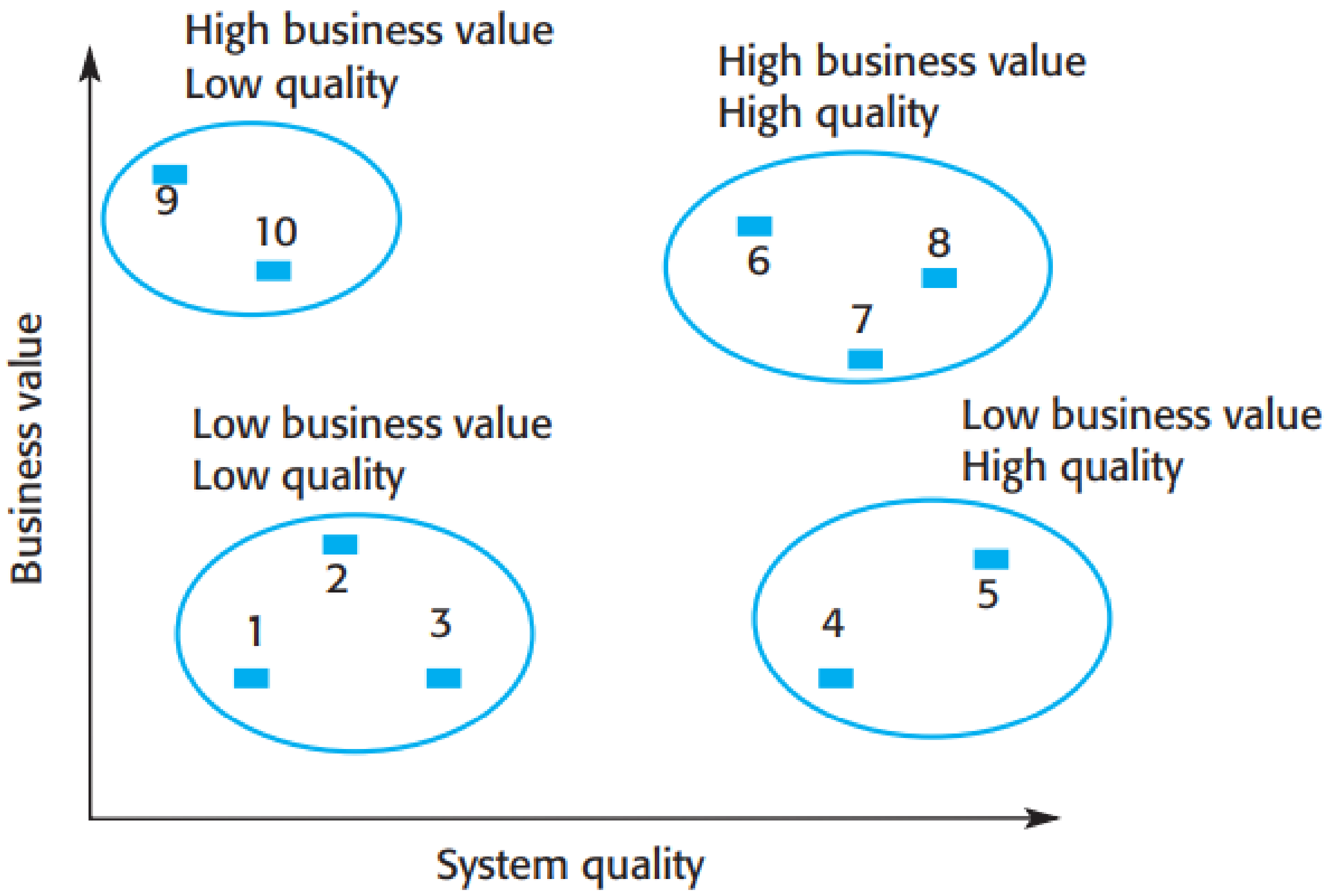
- System quality
- Business value



WHICH DECISION TO MAKE?

Low quality, low business value:

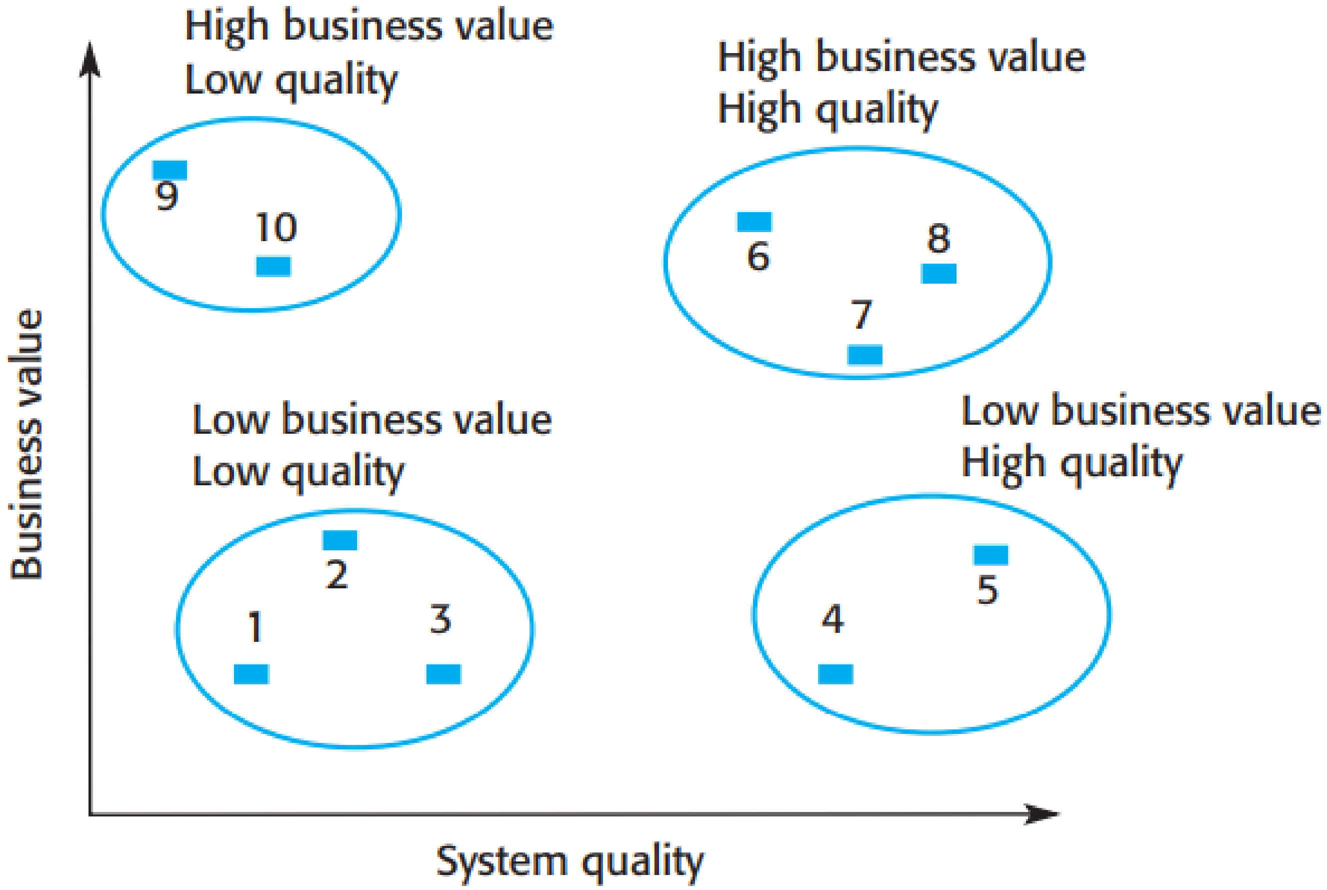
- Keeping these systems in operation will be expensive
- The rate of the return to the business will be fairly small.
- These systems should be **scrapped/abandoned**



WHICH DECISION TO MAKE?

Low quality, high business value:

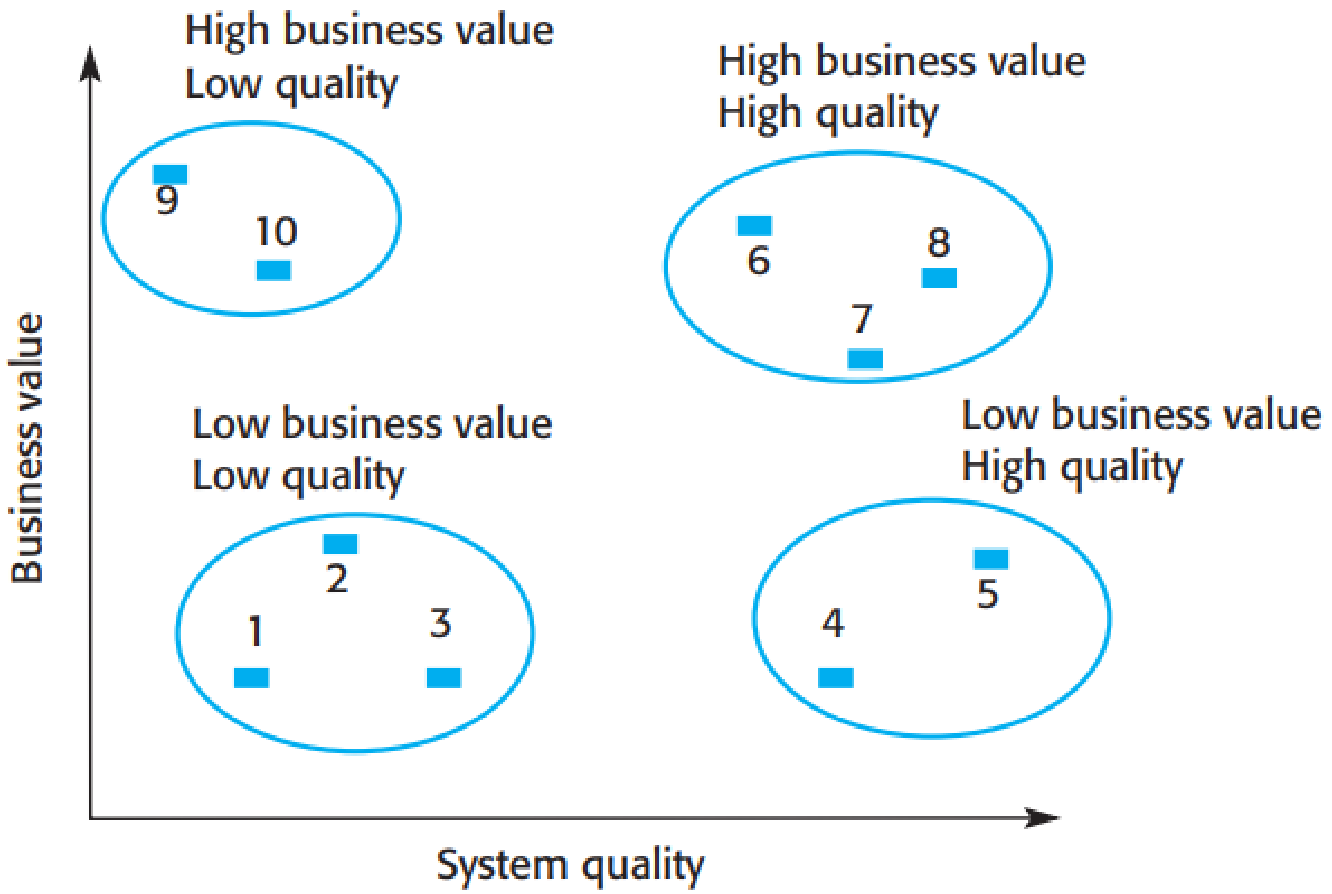
- These systems are making an important business contribution, so they cannot be scrapped.
- However, their low quality means that they are expensive to maintain.
- These systems should be **reengineered** to improve their quality.
- They may also be **replaced**, if suitable off-the-shelf systems are available.



WHICH DECISION TO MAKE?

High quality, low business value:

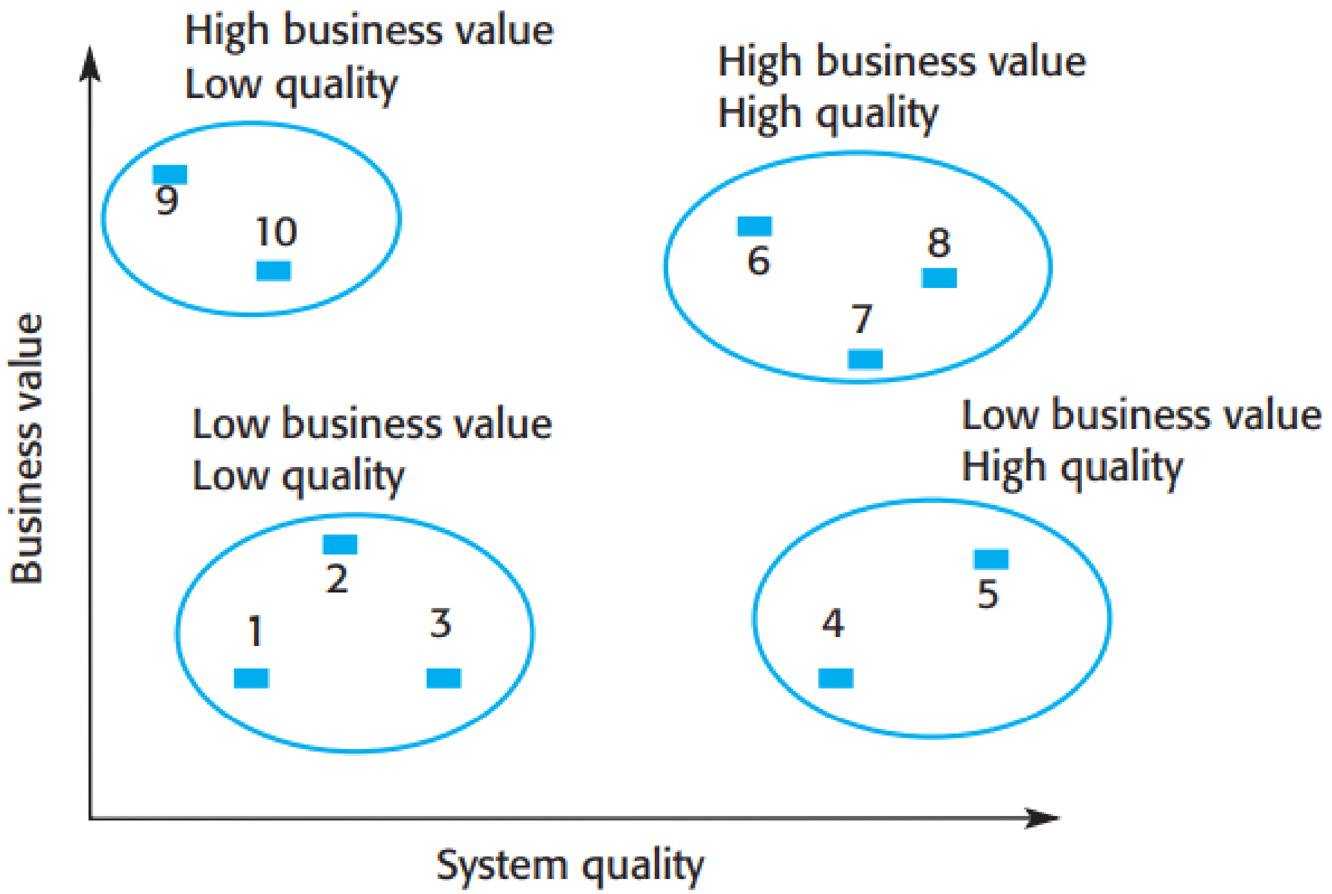
- These systems don't contribute much to the business but may not be very expensive to maintain.
- It is not worth replacing these systems, so **normal system maintenance** may be continued if expensive changes are not required and the system hardware remains in use.
- If expensive changes become necessary, the software should be **scrapped/abandoned**.



WHICH DECISION TO MAKE?

High quality, high business value:

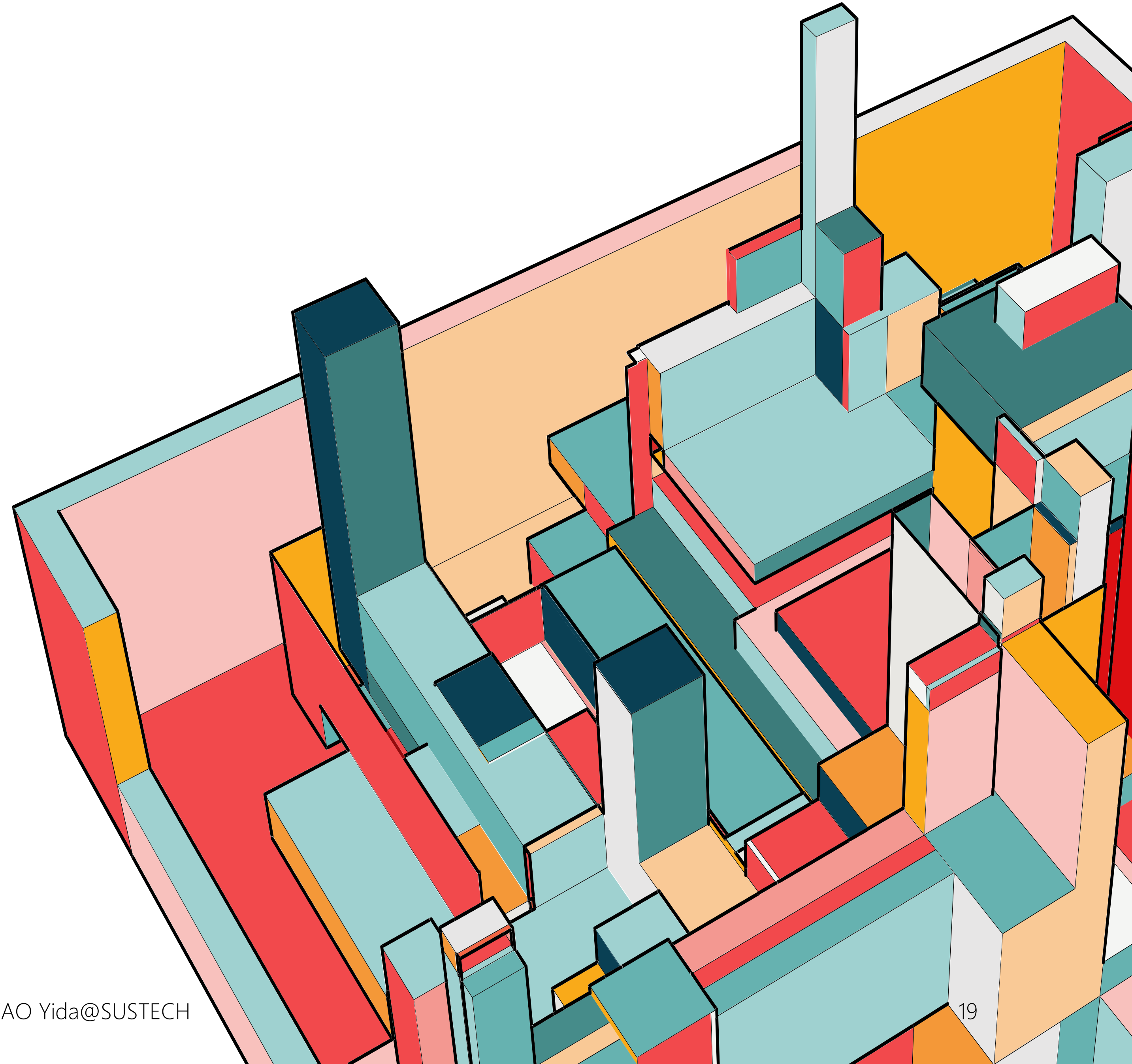
- These systems have to be kept in operation.
- However, their high quality means that you don't have to invest in transformation or system replacement. **Normal system maintenance** should be continued



DEPRECATION REVISITED

- Deprecation (弃用): the process of orderly migration away from and eventual removal of obsolete (过时) systems

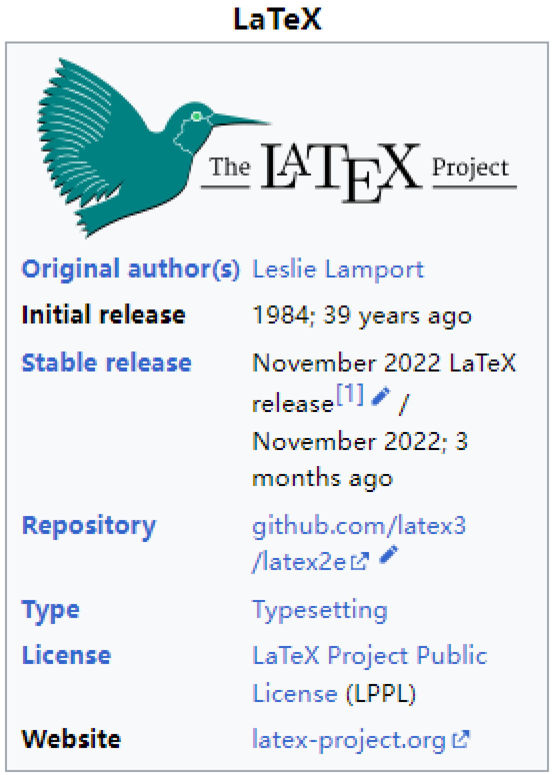
- Deprecation is an important process in software evolution



WHAT SHOULD BE DEPRECATED?

- Age doesn’t justify deprecation: some software systems are old, but still work fine
  - The LaTex typesetting system is old, but it has been finely improved over the course of decades and still functions well

- Old doesn’t mean obsolete



LaTeX

Original author(s)	Leslie Lamport
Initial release	1984; 39 years ago
Stable release	November 2022 LaTeX release[1] / November 2022; 3 months ago
Repository	github.com/latex3/latex2e
Type	Typesetting
License	LaTeX Project Public License (LPPL)
Website	latex-project.org

WHAT SHOULD BE DEPRECATED?

- Deprecation is best suited for systems/modules/code/APIs/features that are demonstrably obsolete and a replacement exists that provides comparable functionality.

- The new replacement might use resources more efficiently, have better security properties, be built in a more sustainable fashion, or just fix bugs.

HOW TO DEPRECATE (ELEGANTLY)?

Dependency Discovery

- To deprecate a system, it is useful to determine:
  - Who is using the old system
  - How the system is being used

- Tools helpful for dependency discovery
  - Static analysis (e.g., which method is calling the deprecated API)
  - Logging

HOW TO DEPRECATE (ELEGANTLY)?

Warning flags

- Owners of deprecated systems add compiler annotations to deprecated symbols (e.g., the `@deprecated` Java annotation)
- Tools check for new usages of these symbols at review time and alerting authors to shy away from the deprecated components

HOW TO DEPRECATE (ELEGANTLY)?

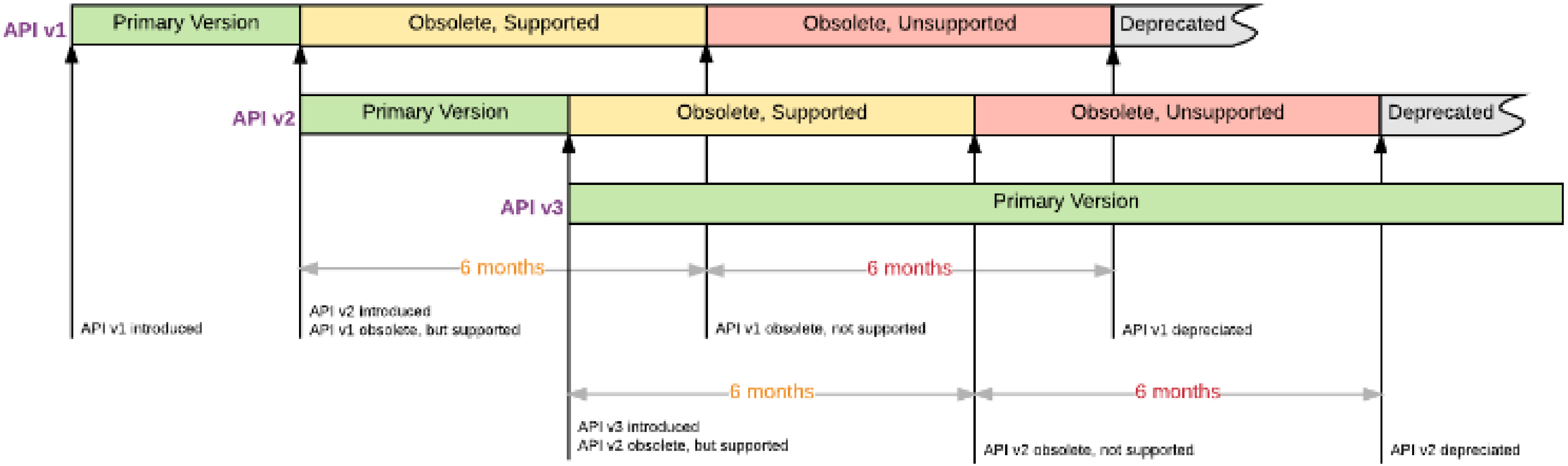
Sunset period

- Sunset period provides API consumers with an adequate time to upgrade to a newer version or retire the functionality before the API stops working.

- To provide a smooth transition for customers and internal developers, some providers defines sunset period as a combination of 6 months of fully functional and supported API and another 6 months of functional API with no additional support.

HOW TO DEPRECATE (ELEGANTLY)?

Sunset period



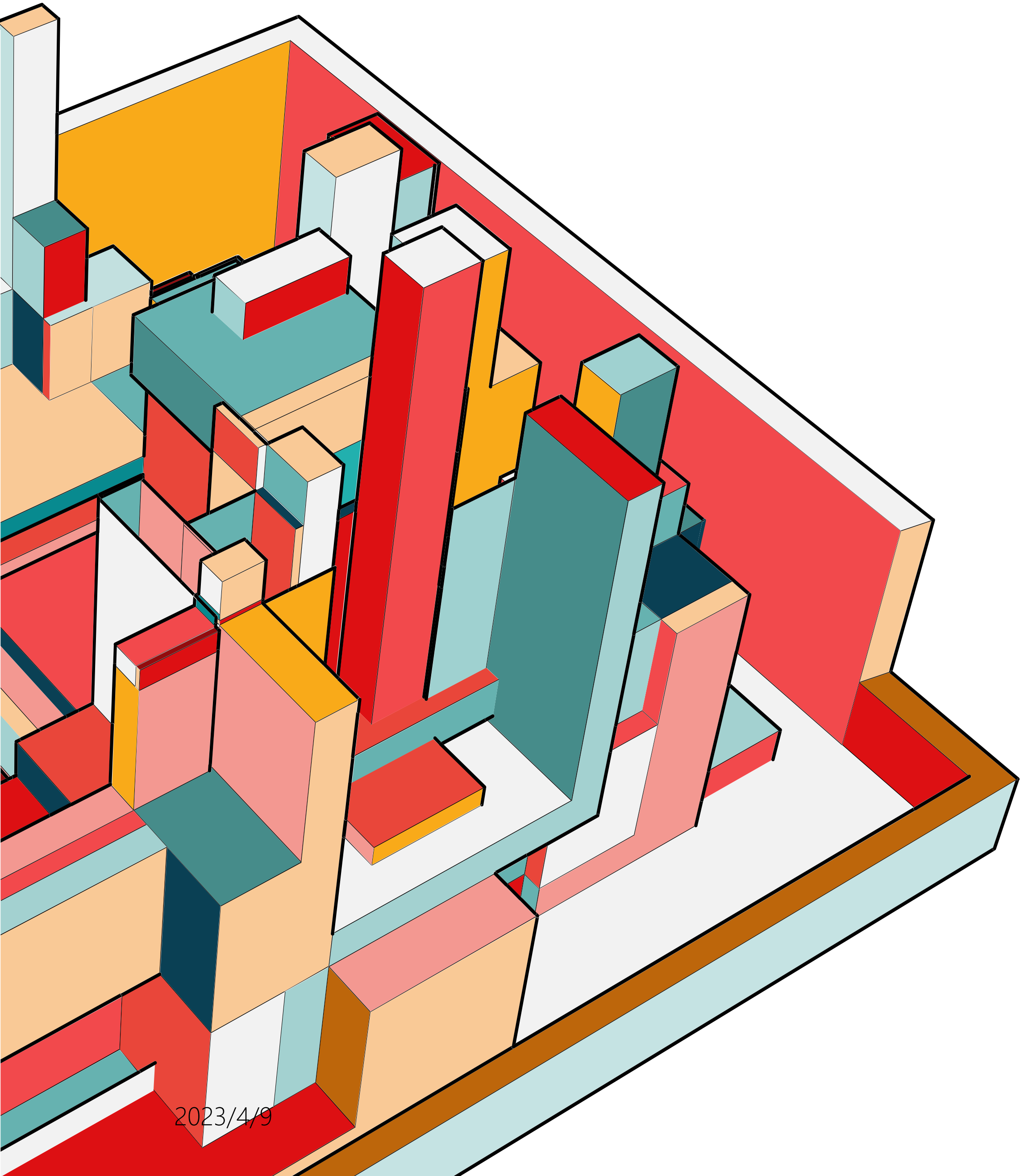
https://connect.ultipro.com/api-deprecation

HOW TO DEPRECATE (ELEGANTLY)?

Migration & Testing

- Using tools to automatically update the codebase to refer to new libraries or runtime services

- Using test suite to automatically determine whether all references to deprecated symbols have been removed without breaking existing functionalities



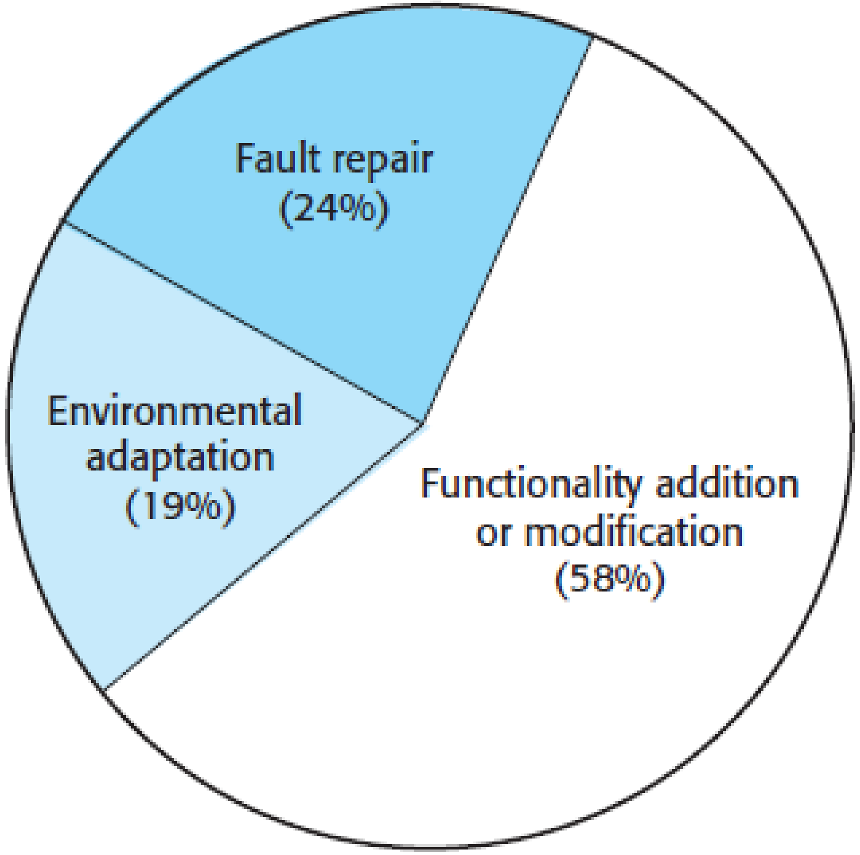
SOFTWARE MAINTENANCE

Software maintenance is the general process of changing a system after it has been delivered.

MAINTENANCE EFFORT DISTRIBUTION

Fault repairs to fix bugs and vulnerabilities

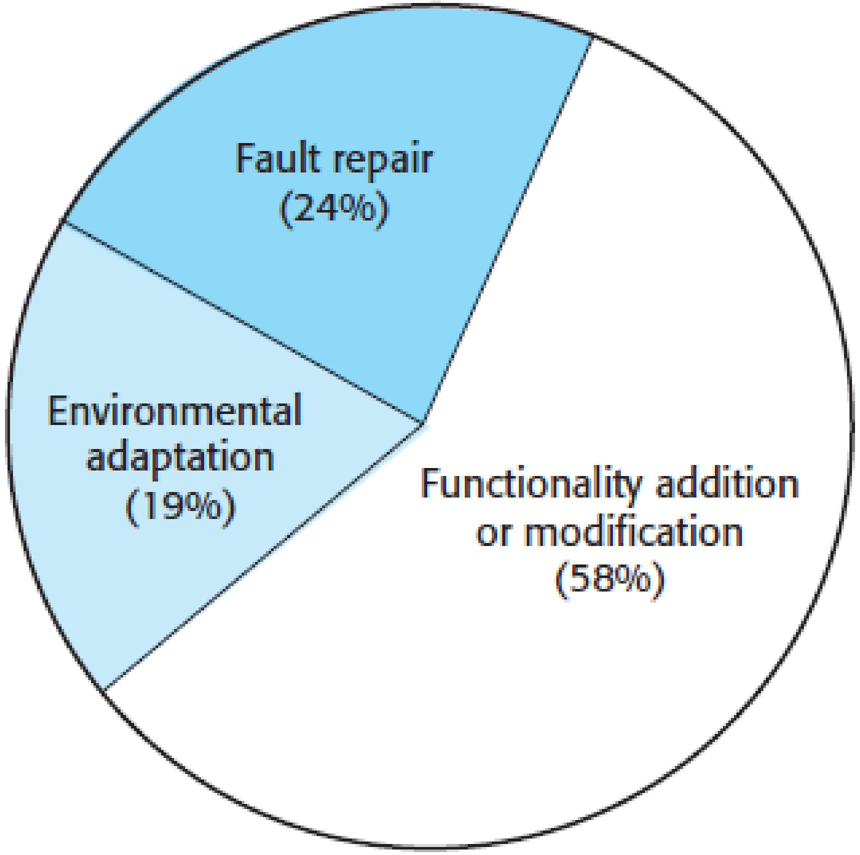
- *Coding errors* are usually relatively cheap to correct;
- *Design errors* are more expensive because they may involve rewriting several program components.
- *Requirements errors* are the most expensive to repair because extensive system redesign may be necessary.



MAINTENANCE EFFORT DISTRIBUTION

Environmental adaptation to adapt the software to new platforms and environments.

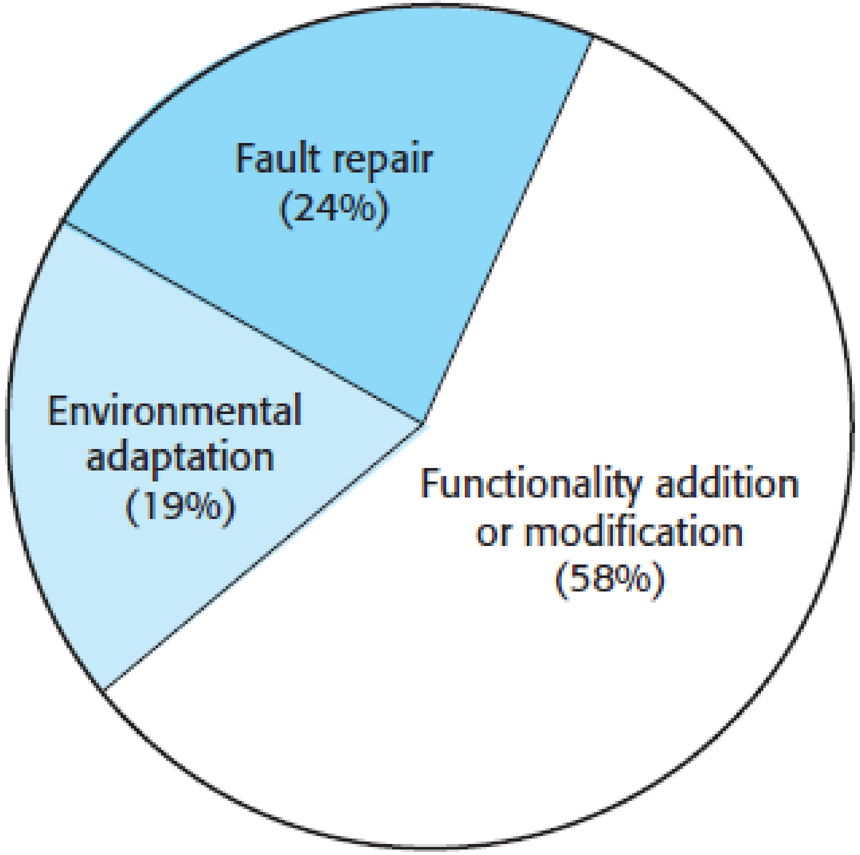
- This type of maintenance is required when some aspect of a system’s environment, such as the hardware, the platform operating system, or other support software, changes.
- Application systems may have to be modified to cope with these environmental changes.

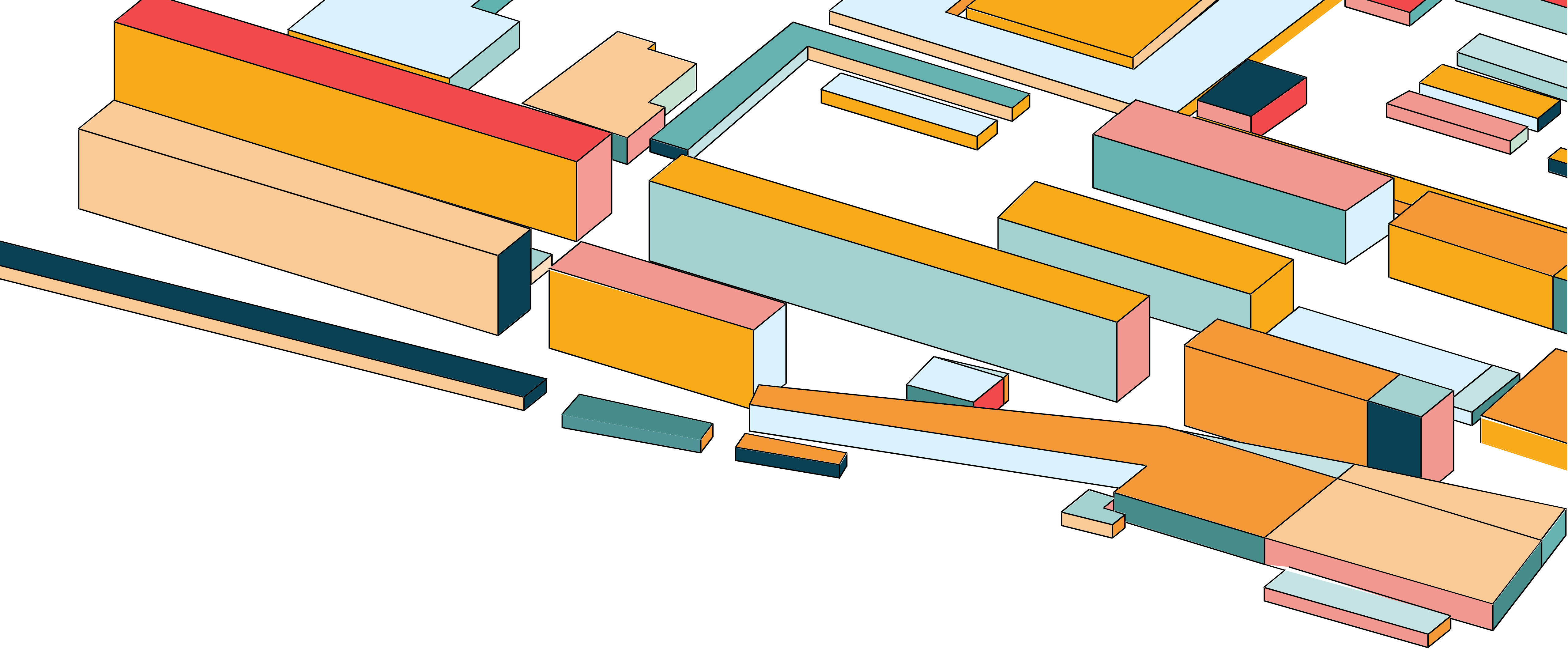


MAINTENANCE EFFORT DISTRIBUTION

Functionality addition to add new features and to support new requirements.

- This type of maintenance is necessary when system requirements change in response to organizational or business change.
- The scale of the changes required to the software is often much greater than for the other types of maintenance.





HOW TO MEASURE MAINTAINABILITY?

EXAMPLE METRICS FOR MAINTAINABILITY

Number of requests for corrective maintenance

An increase in the number of bug and failure reports may indicate that more errors are being introduced into the program than are being repaired during the maintenance process. *This may indicate a decline in maintainability*.

EXAMPLE METRICS FOR MAINTAINABILITY

Average time required for impact analysis

- This is related to the number of program components that are affected by the change request.
- If the time required for impact analysis increases, it implies that more components are affected and *maintainability is decreasing*.

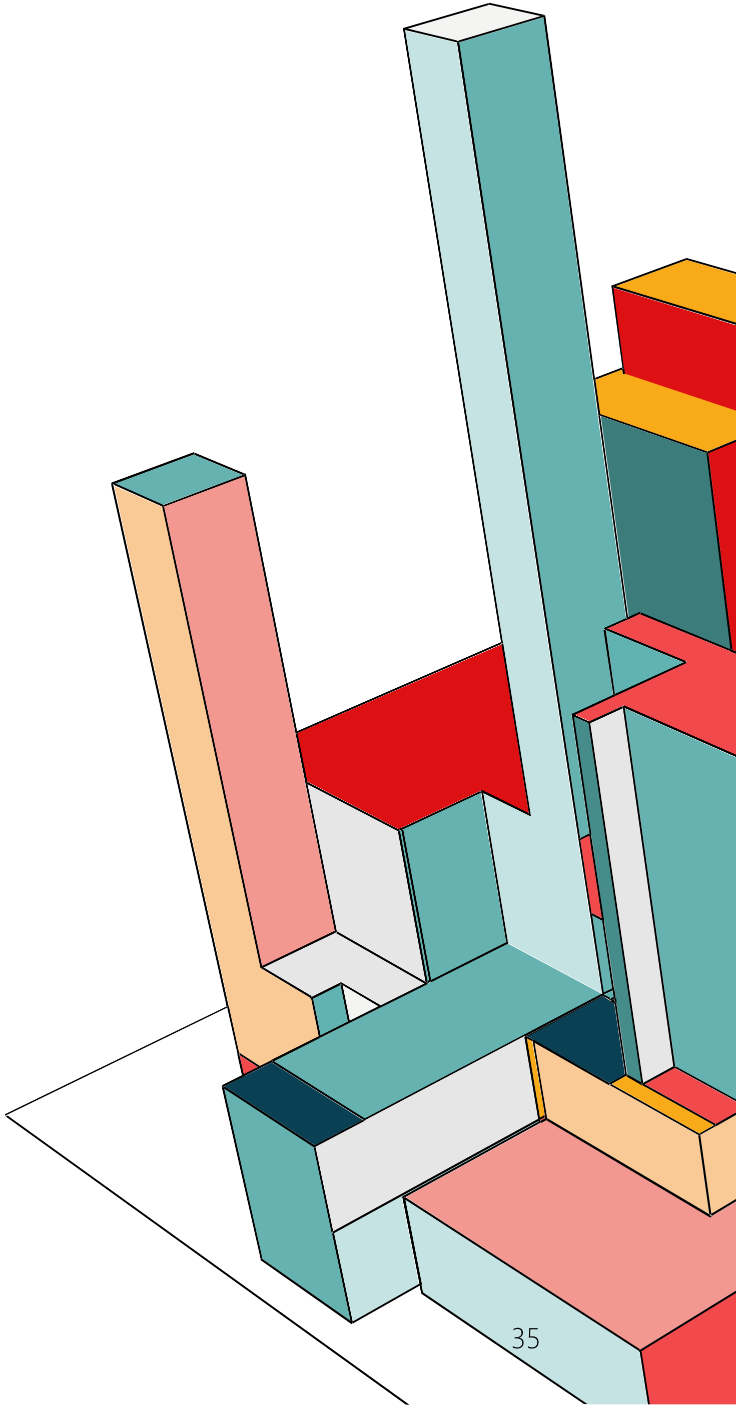
EXAMPLE METRICS FOR MAINTAINABILITY

Average time taken to implement a change request

- This is the amount of time that you need to modify the system and its documentation, after you have assessed which components are affected.
- An increase in the time needed to implement a change *may indicate a decline in maintainability*.

IMAGINING THIS

- An application has served the business needs of a company for 10 or 15 years.
- During that time it has been corrected, adapted, and enhanced many times.
- Now the application is unstable. It still works, but every time a change is attempted, unexpected and serious side effects occur.
- Yet the application must continue to evolve. What should we do?

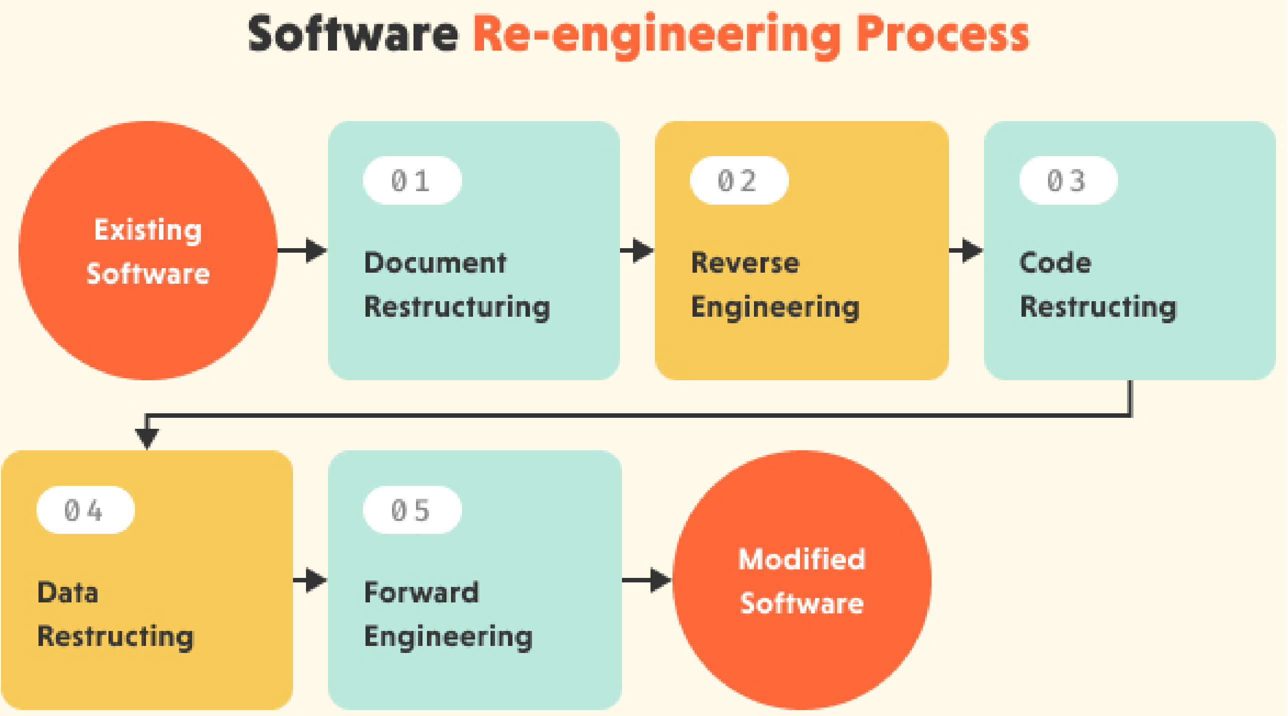


SOFTWARE REENGINEERING

- To make legacy software systems easier to maintain, you can **reengineer** these systems to improve their structure and understandability

- Reengineering may involve:
  - Redocumenting the system
  - Refactoring the system architecture
  - Translating programs to a modern programming language
  - Modifying and updating the structure and values of the system’s data.

SOFTWARE REENGINEERING PROCESS



https://modlogix.com/blog/legacy-software-re-engineering-risks-and-mitigations/

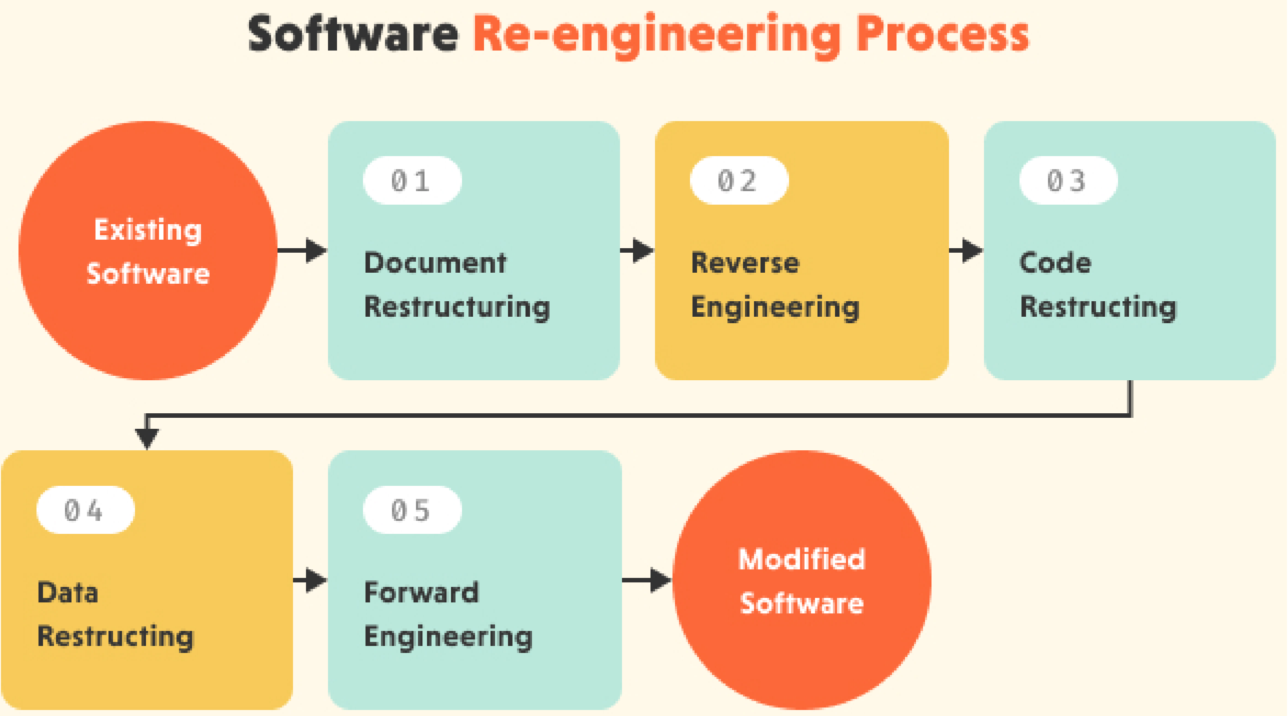
Input:

- The original legacy system

Output:

- An improved and restructured version of the same program
- Program documentation
- Reengineered data

SOFTWARE REENGINEERING PROCESS

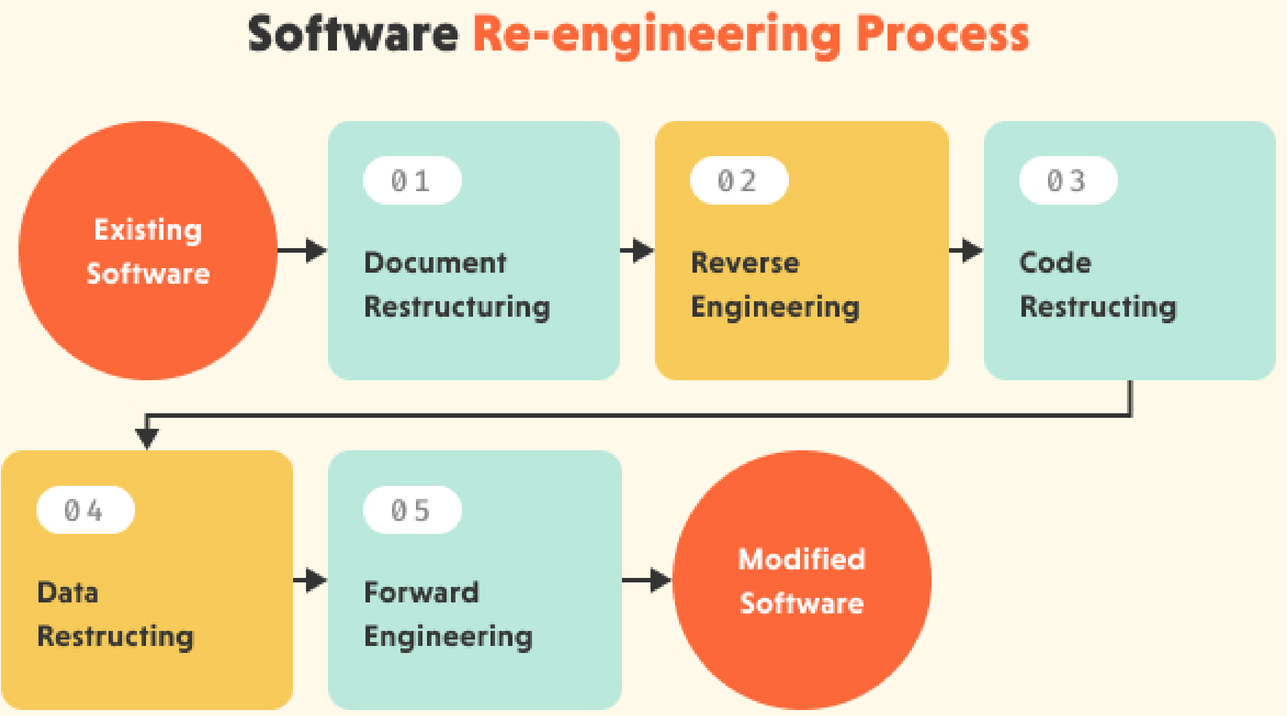


https://modlogix.com/blog/legacy-software-re-engineering-risks-and-mitigations/

3 stages

- Reverse engineering
- System transformation
- Forward engineering

SOFTWARE REENGINEERING PROCESS

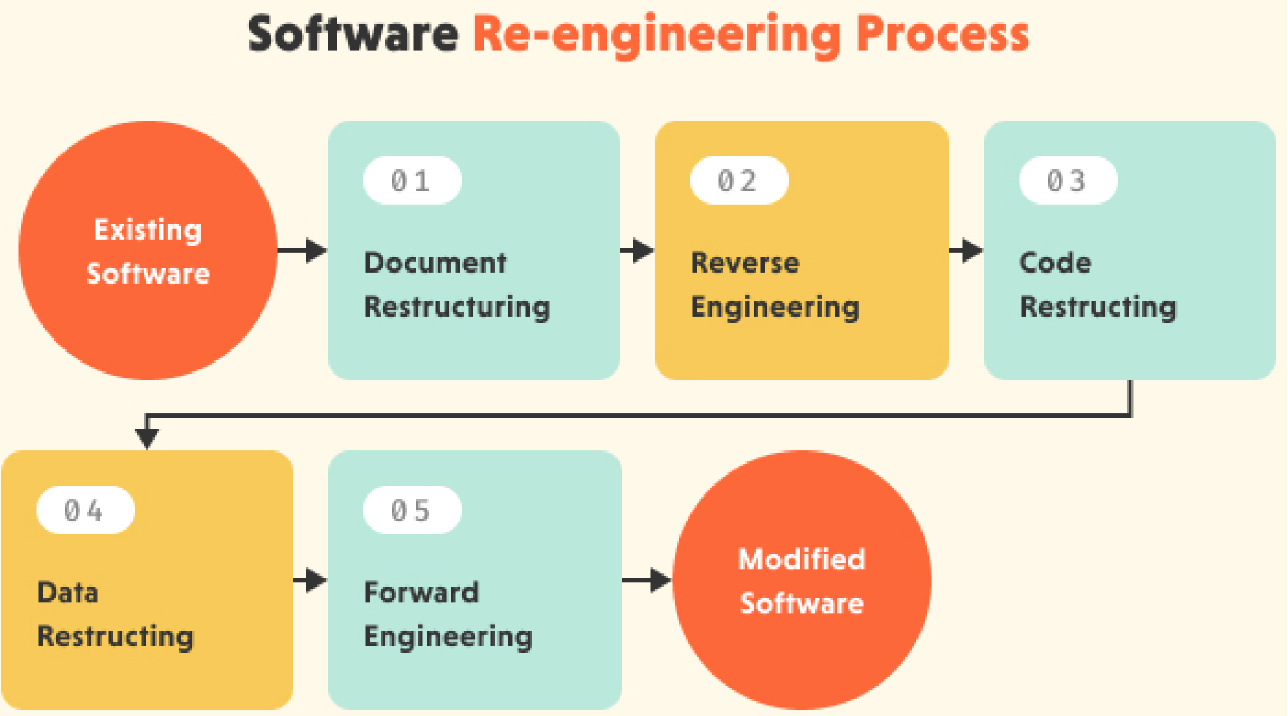


https://modlogix.com/blog/legacy-software-re-engineering-risks-and-mitigations/

Reverse engineering:

- A process of design recovery: analyzing a program to create a representation of it at a higher level of abstraction than source code
- Extract data, architectural, and procedural design information from an existing program
- Goal is to understand how it was built

SOFTWARE REENGINEERING PROCESS

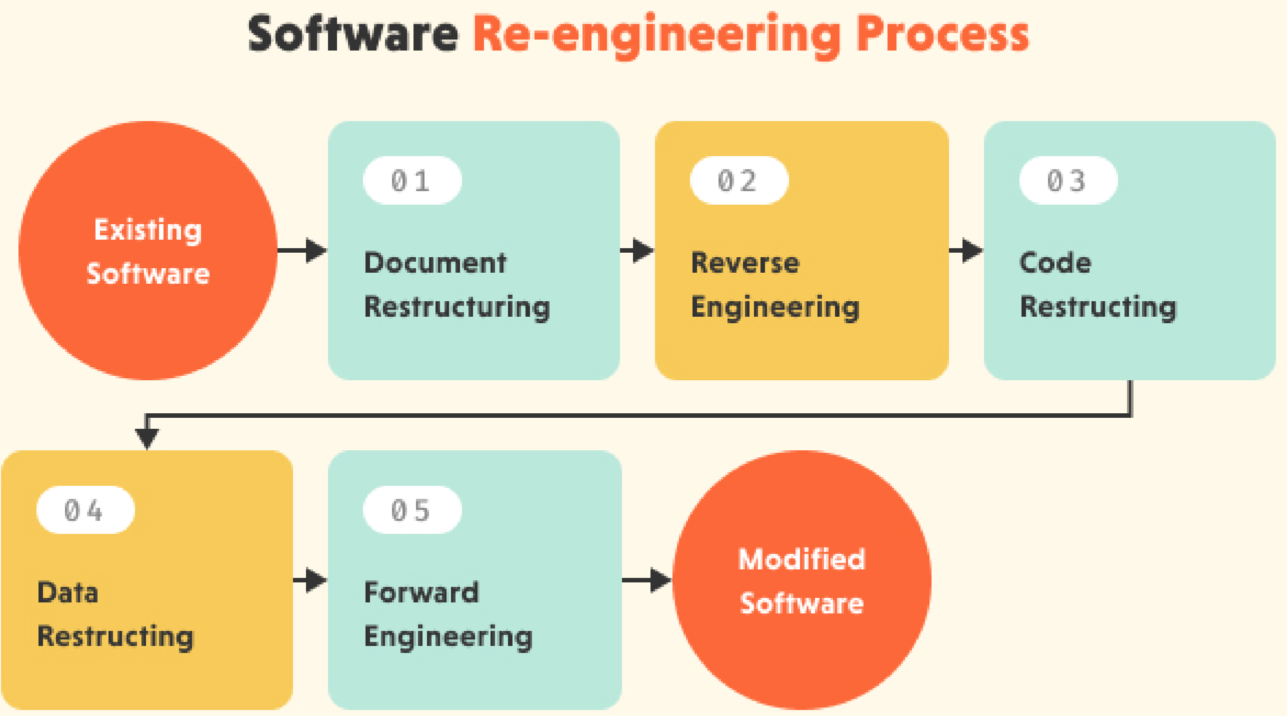


https://modlogix.com/blog/legacy-software-re-engineering-risks-and-mitigations/

System transformation:

- The abstract representations obtained during reverse engineering are further transformed into other representations at the same abstraction level.
- The aim is to improve the software structure, quality, and stability.

SOFTWARE REENGINEERING PROCESS

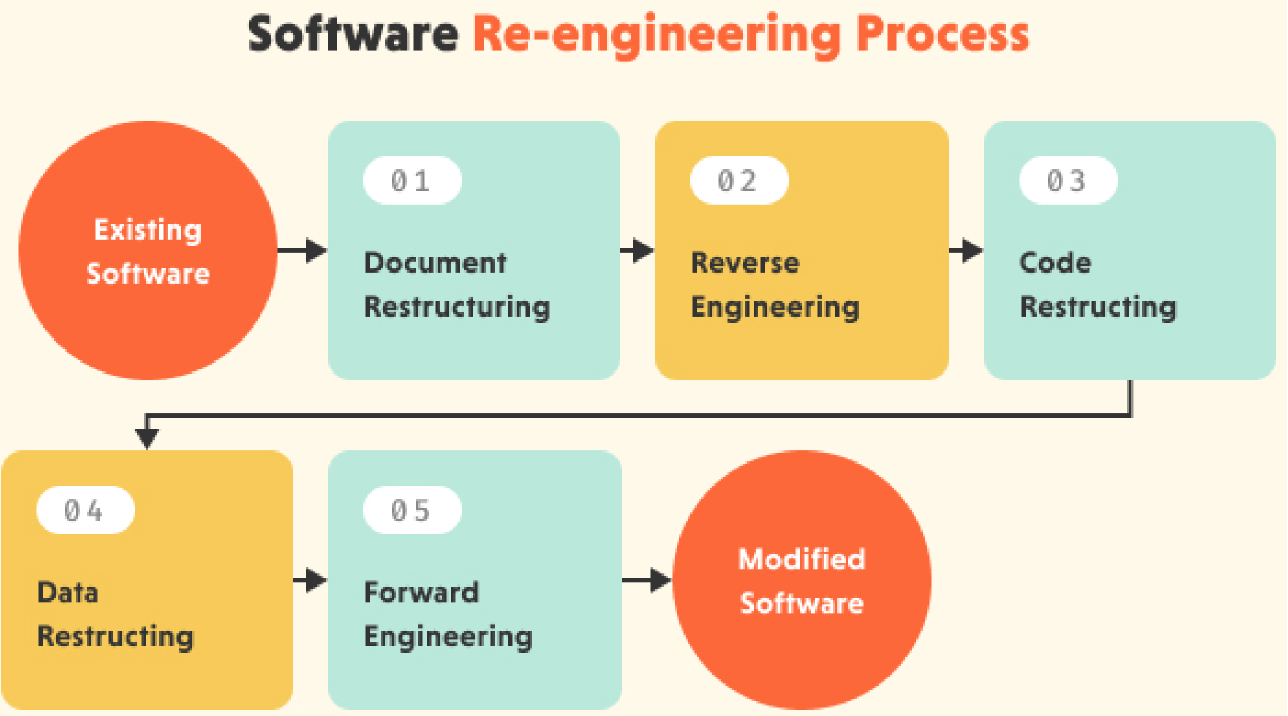


https://modlogix.com/blog/legacy-software-re-engineering-risks-and-mitigations/

System transformation may involve:

- **Refactoring**: restructuring the code at the level of methods and classes
- **Rearchitecting**: refactoring at the level of modules and components
- **Rewriting**: rearchitecting at the highest possible level

SOFTWARE REENGINEERING PROCESS

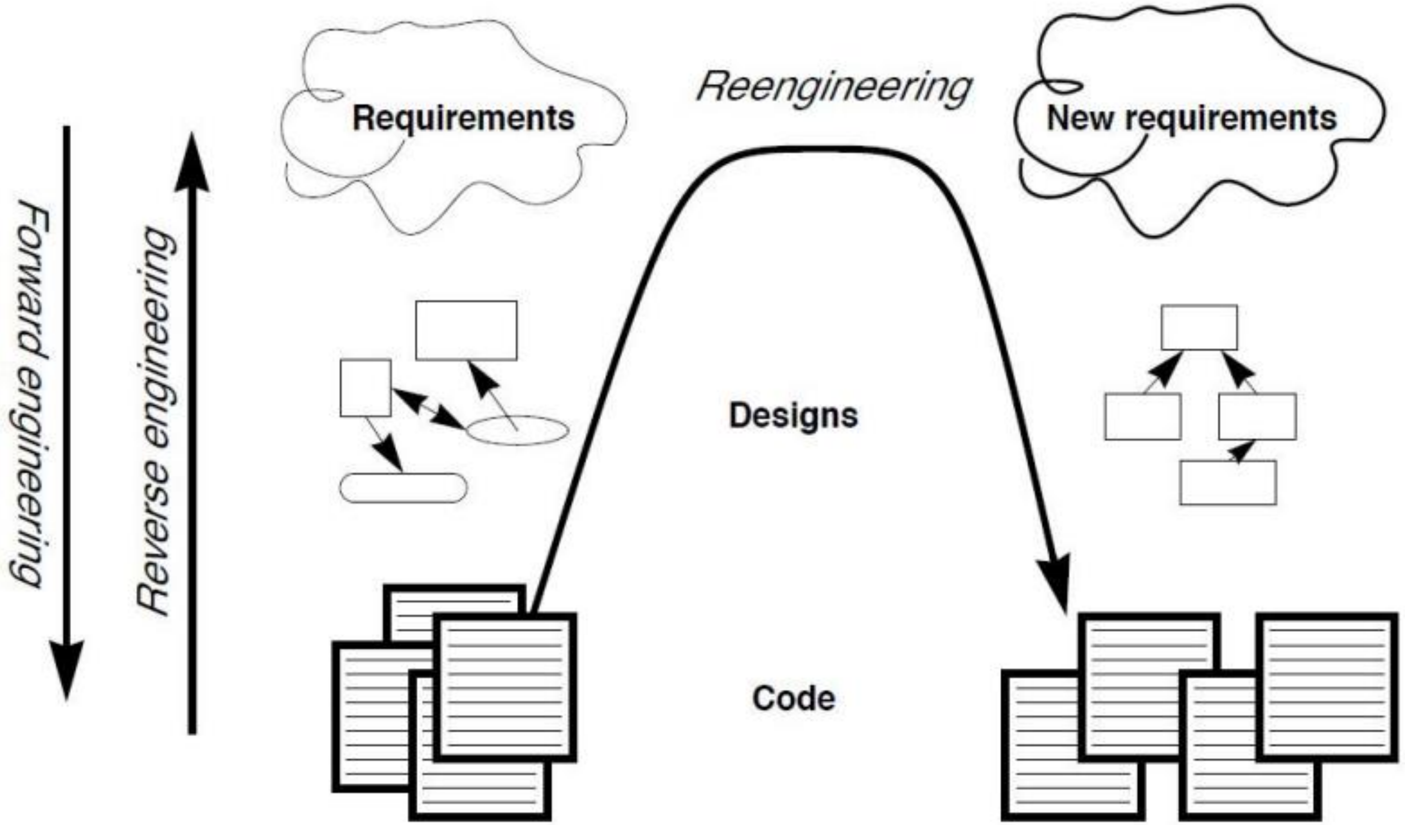


https://modlogix.com/blog/legacy-software-re-engineering-risks-and-mitigations/

Forward engineering:

- After system transformation, the transformed system representations can be used to generate physical implementations of the initial system, e.g., upgraded code and executables
- Forward engineering starts with system specification and includes the design and implementation of a new system – like an ordinary software development process.

SOFTWARE REENGINEERING PROCESS



Forward engineering:

- Requirements -> design -> code

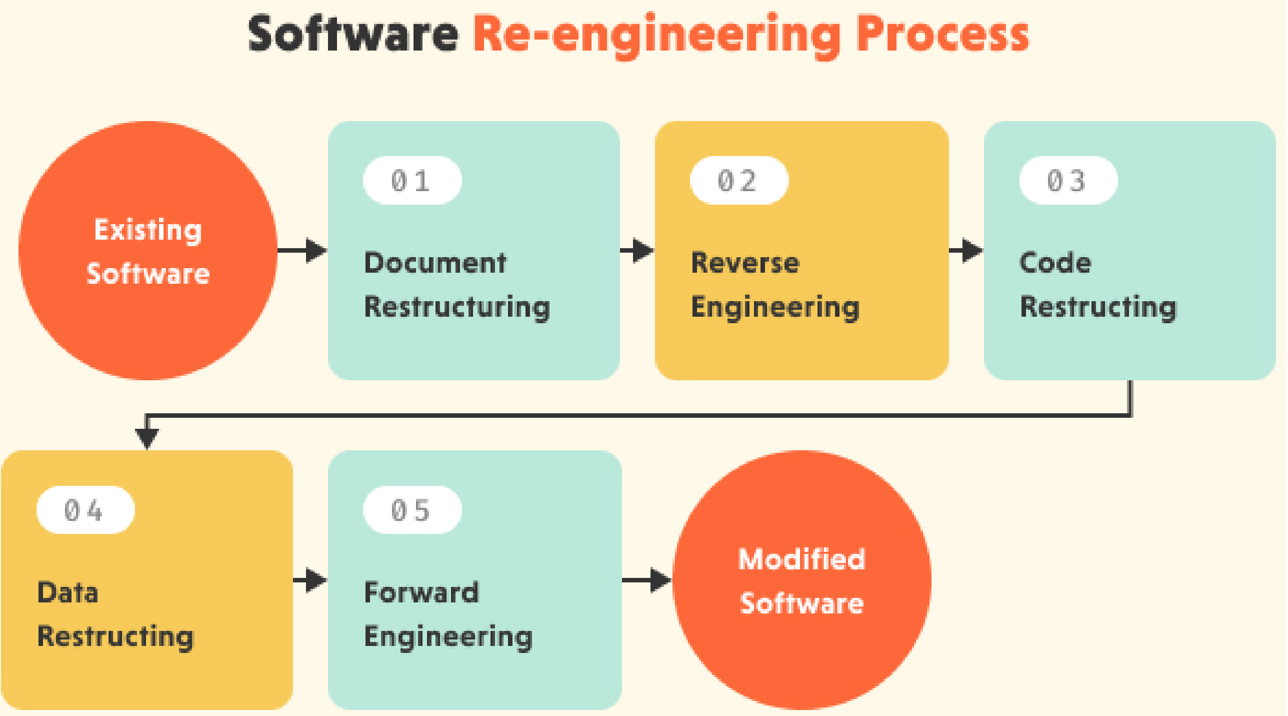
Reverse engineering:

- Code -> design -> requirements

Reengineering

- Old code -> new code (via design transformation)

SOFTWARE REENGINEERING PROCESS

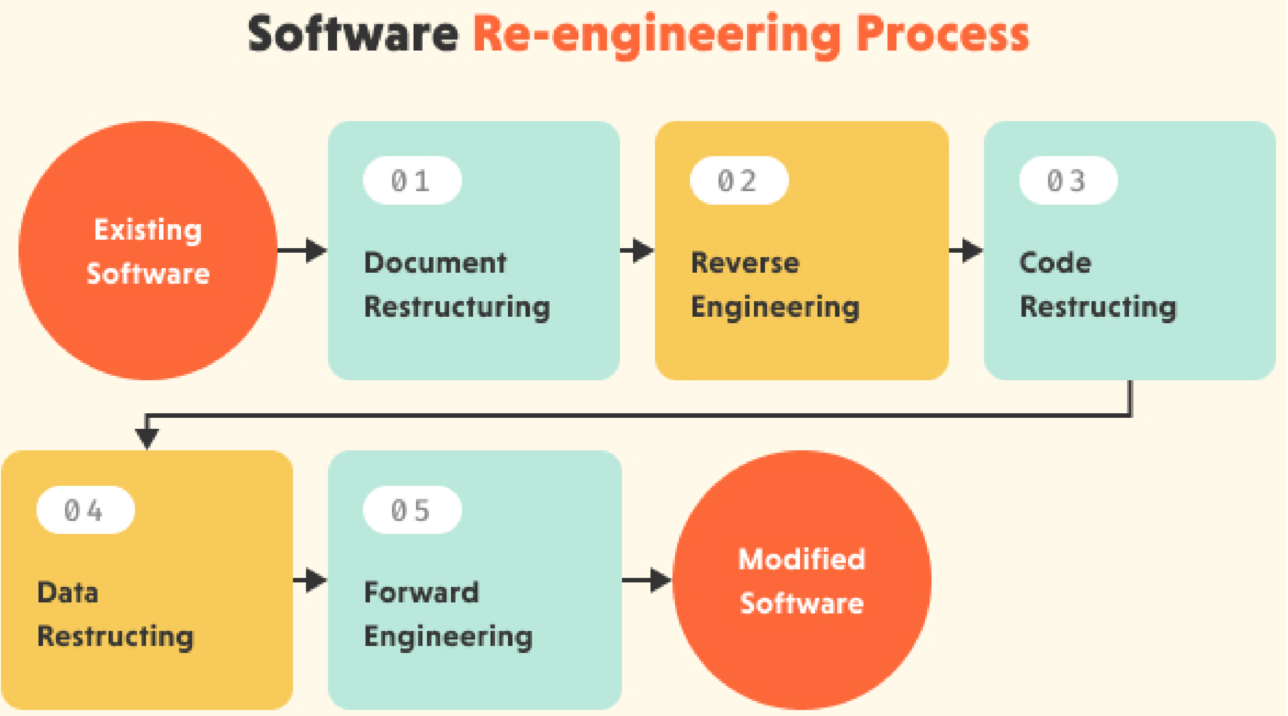


https://modlogix.com/blog/legacy-software-re-engineering-risks-and-mitigations/

Document restructuring:

- Update the documentation for the parts of the system that is currently undergoing change
- If the legacy system is business critical, may need to fully redocument the system

SOFTWARE REENGINEERING PROCESS

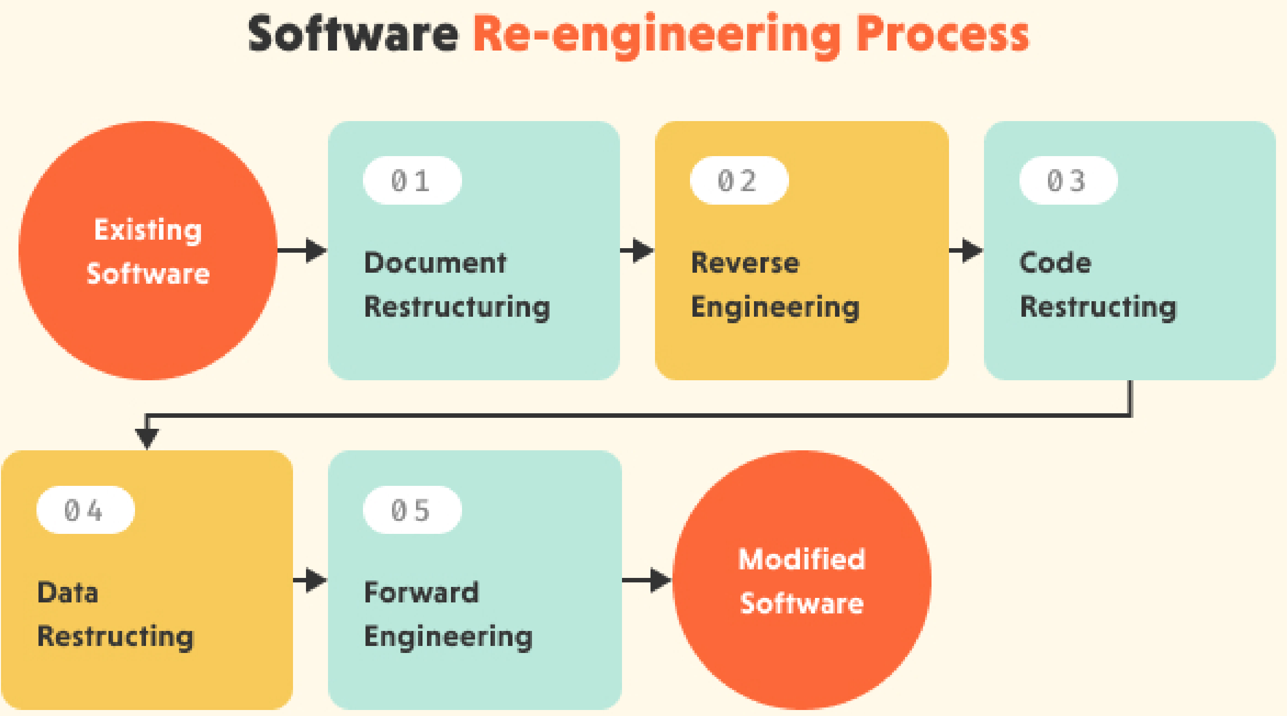


https://modlogix.com/blog/legacy-software-re-engineering-risks-and-mitigations/

Data restructuring

- Updating the data to reflect program changes, e.g., correcting mistakes, removing duplicates, cleaning up data
- This may involve redefining database schemas and converting existing databases to the new structure
- Can be a very expensive and prolonged process

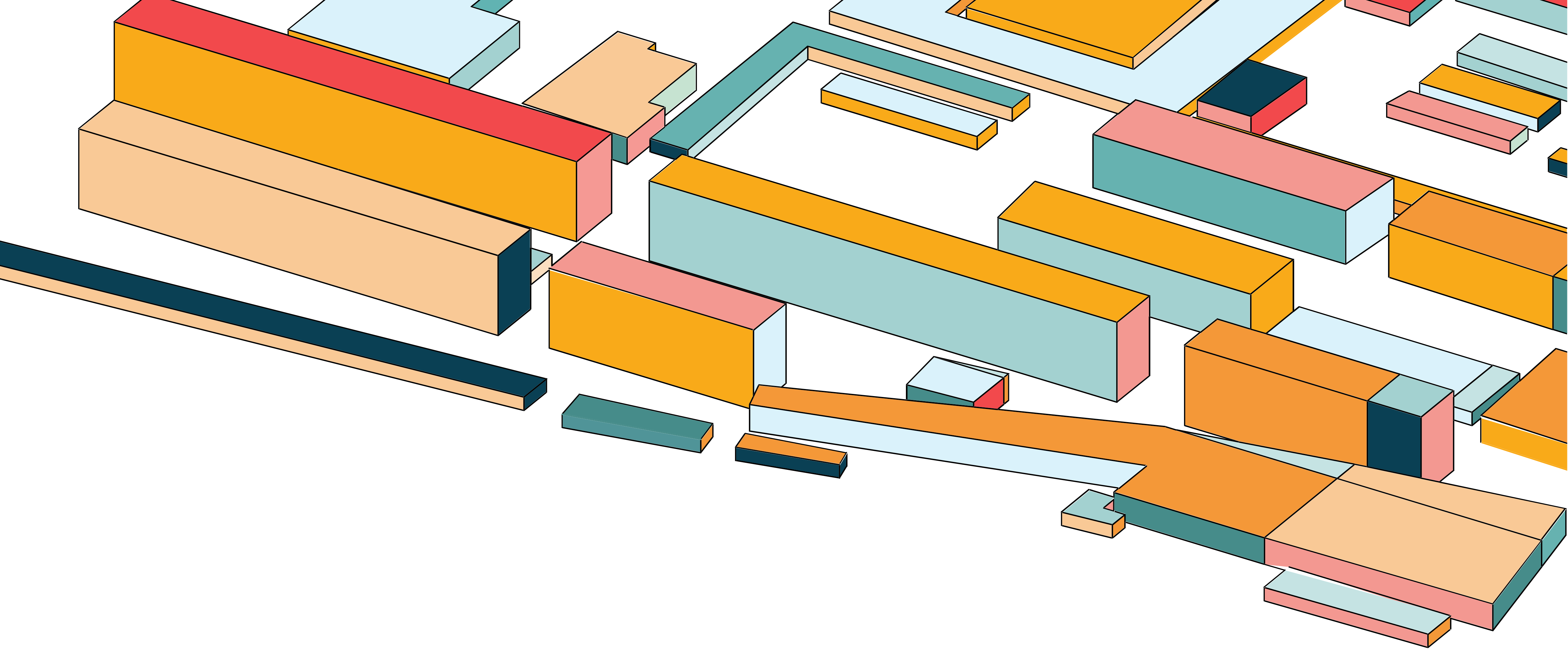
SOFTWARE REENGINEERING PROCESS



https://modlogix.com/blog/legacy-software-re-engineering-risks-and-mitigations/

Code restructuring/refactoring

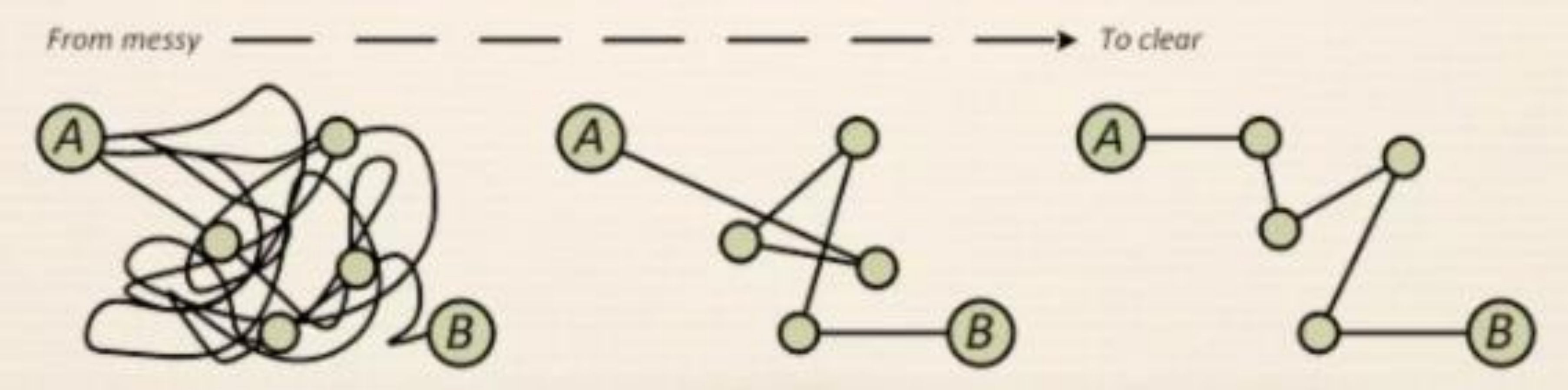
- The control structure of the program is analyzed and modified to make it easier to read and understand.
- Related parts of the program are grouped together and redundancy is removed
- This can be partially automated, but some manual intervention is usually required.



REFACTORING (代码重构)

WHAT IS REFACTORING

- A refactoring is a software transformation that
  - Preserves the external behavior of the software
  - Improves the internal structure of the software



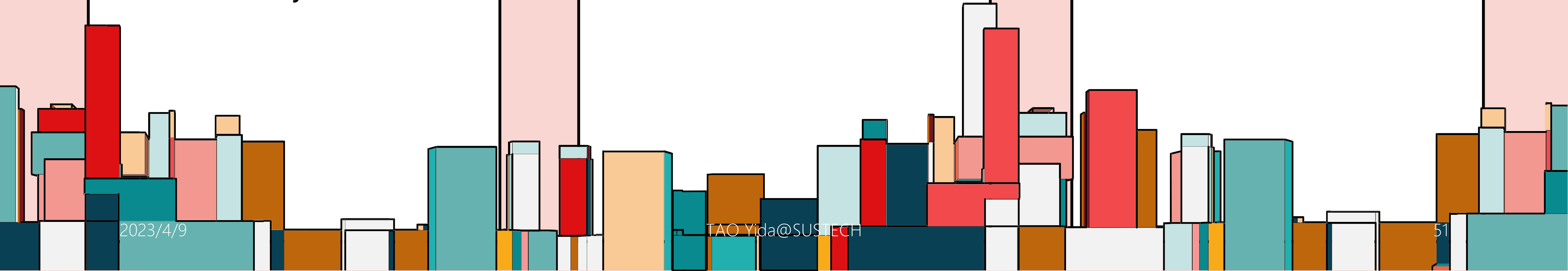
WHY REFACTORING?

Refactoring is a disciplined way to clean up code so that it is easier to read and cheaper to maintain

- To improve the design of software
- To counter code decay or software aging
- To increase software readability

- To find bugs and write more robust code
- To increase productivity (program faster) on a long term basis

- To reduce cost of software maintenance
- To prepare for future customization

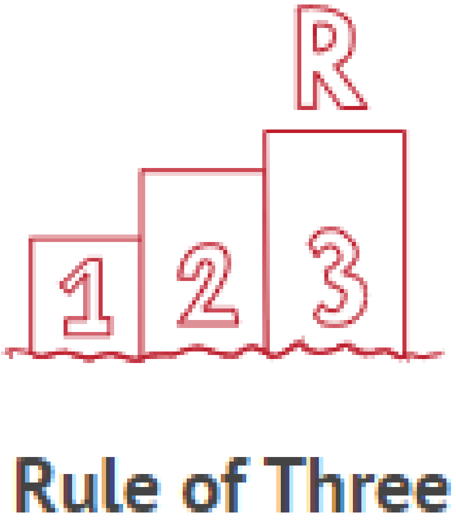


WHEN TO REFACTOR

- When you’re doing something for the first time, just get it done.

- When you’re doing something similar for the second time, cringe at having to repeat but do the same thing anyway.

- When you’re doing something for the third time, start refactoring.



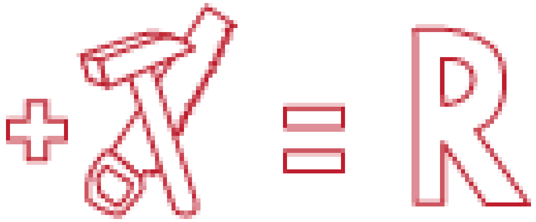
https://refactoring.guru/refactoring/when

WHEN TO REFACTOR

- Refactoring makes it easier to add new features, especially if the feature is difficult to integrate with the existing code

- If you have to deal with someone else's dirty code, try to refactor it first. Clean code is much easier to grasp.

- You will improve it not only for yourself but also for those who use it after you.



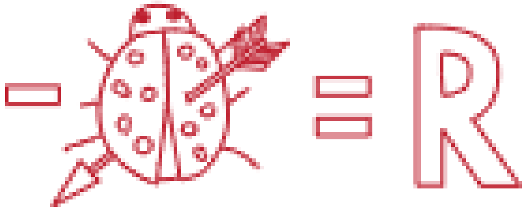
When adding a feature

https://refactoring.guru/refactoring/when

WHEN TO REFACTOR

- If a bug is very hard to trace, refactor first to make the code more understandable,

- Clean your code and the errors will practically discover themselves.



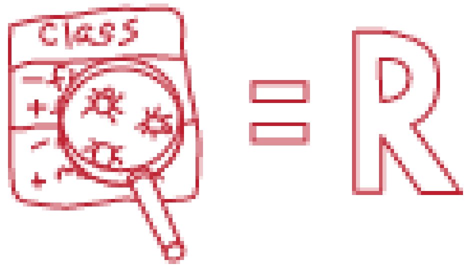
When fixing a bug

https://refactoring.guru/refactoring/when

WHEN TO REFACTOR

- Code review may be the last chance to tidy up the code before it becomes available to the public

- Best to perform code reviews in a pair with an author.

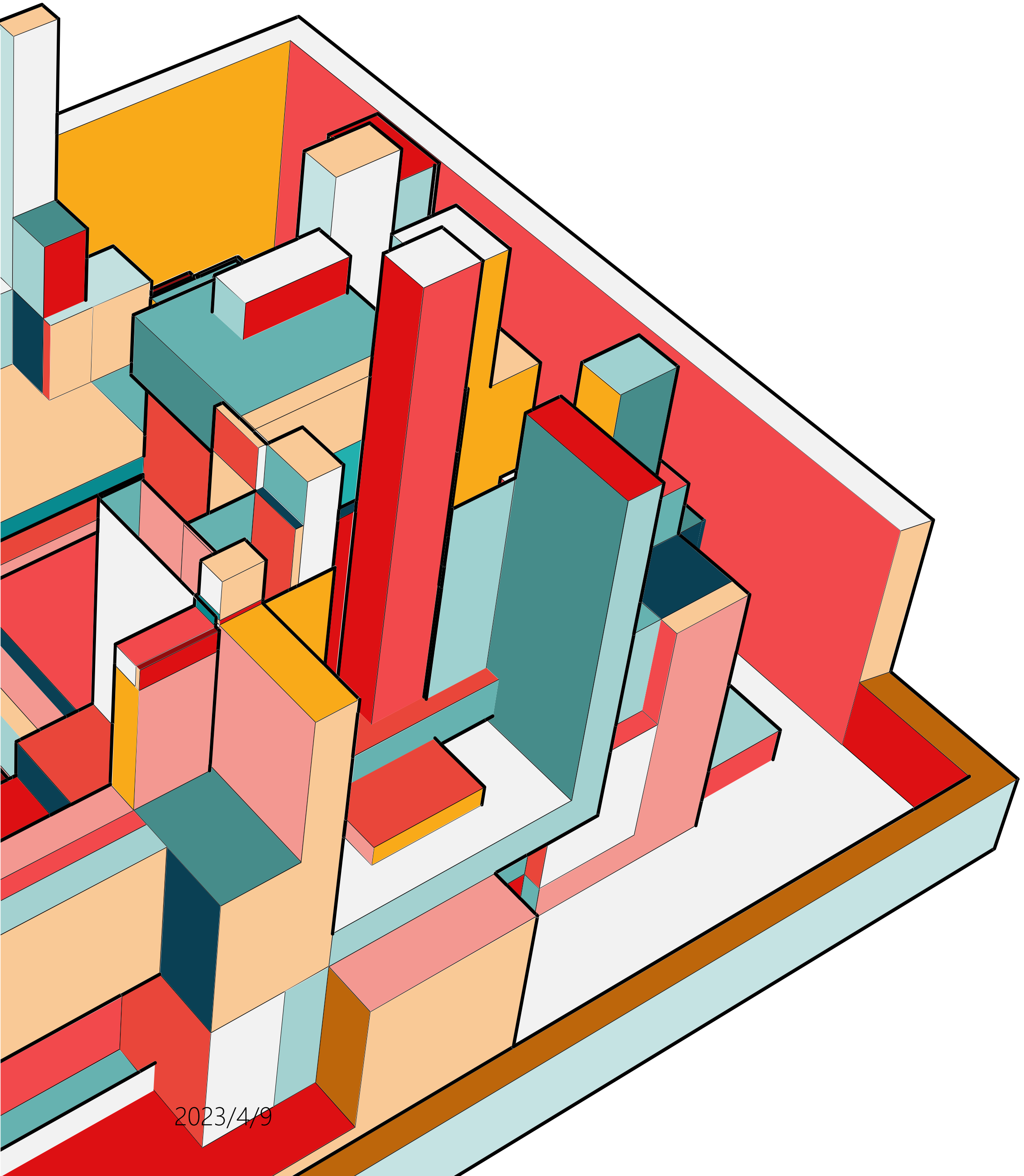


During a code review

https://refactoring.guru/refactoring/when

WHAT TO REFACTOR - CODE SMELLS

- A warning sign of your code
- A surface indication that usually corresponds to a deeper problem in the code or system
- Code that doesn’t smell good / doesn’t feel right



TYPES OF CODE SMELLS

- Bloaters
- OO Abusers
- Change Preventers
- Dispensables
- Couplers

BLOATERS

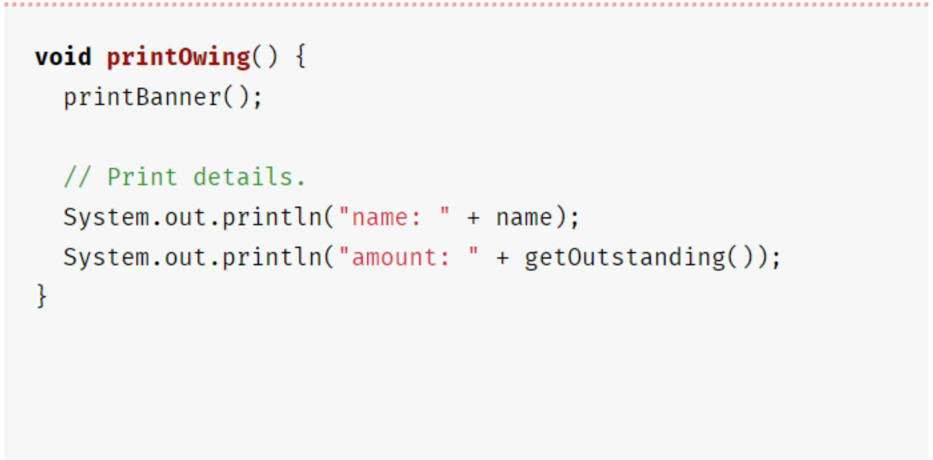
- Bloaters: 代码臃肿、膨胀剂、吸血者
- Bloaters are code, methods and classes that have increased to such giant proportions that they’re hard to work with

- Usually these smells don't crop up right away, rather they accumulate over time as the program evolves, especially when nobody makes an effort to eradicate them

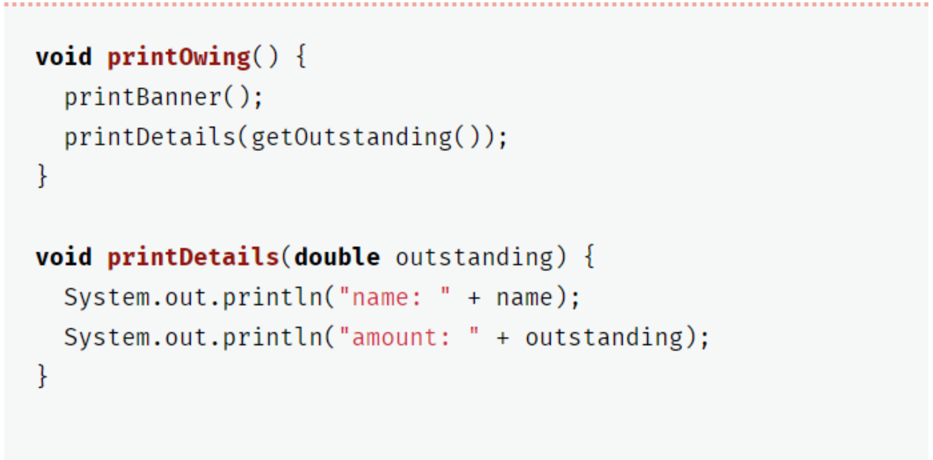
https://refactoring.guru/refactoring/smells/bloaters

BLOATERS - LONG METHOD

- A method contains too many lines of code. The more lines found in a method, the harder it's to figure out what the method does.
- Refactorings: Extract Method, etc.



```java
void printOwing() {
  printBanner();

  // Print details.
  System.out.println("name:  " + name);
  System.out.println("amount:  " + getOutstanding());
}
```



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

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

https://refactoring.guru/extract-method

BLOATERS - LONG PARAMETER LIST

- A method has more than 3 or 4 parameters, which are hard to understand and error-prone
- Refactorings: Replace parameter with method call, etc.

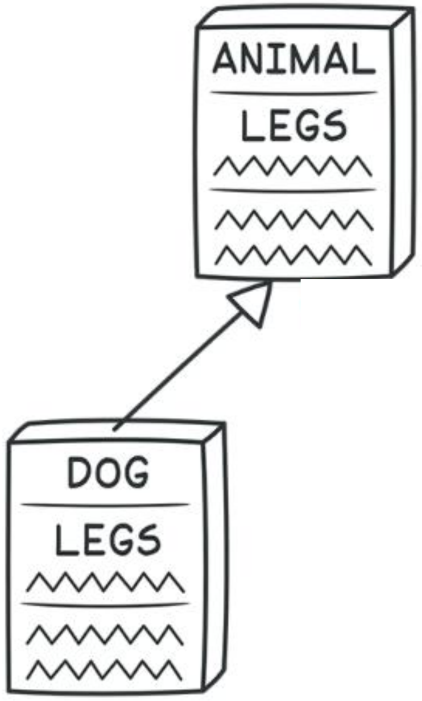
```
int basePrice = quantity * itemPrice;
double seasonDiscount = this.getSeasonalDiscount();
double fees = this.getFees();
double finalPrice = discountedPrice(basePrice, seasonDiscount, fees);
```

```
int basePrice = quantity * itemPrice;
double finalPrice = discountedPrice(basePrice);
```

https://refactoring.guru/replace-parameter-with-method-call

OO ABUSERS

Object-orientation abusers are a type of code smells that refers to incorrect or incomplete implementation of Object Oriented Concepts

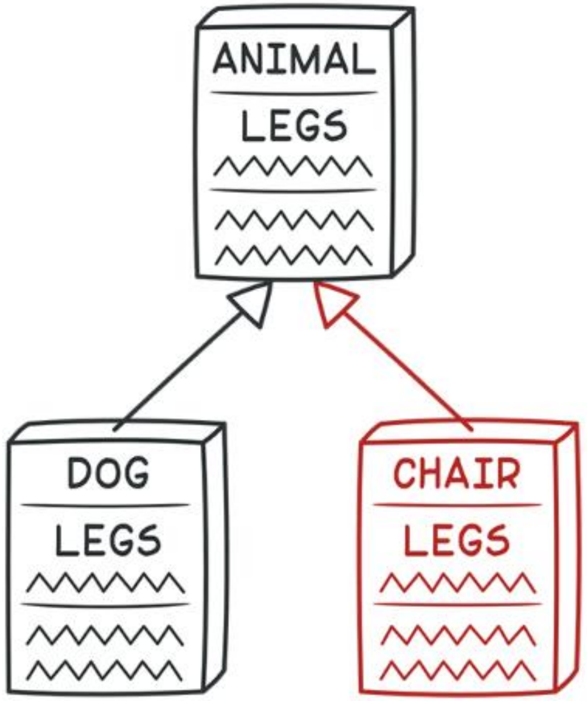


OO ABUSERS - REFUSED BEQUEST

- A subclass inherits from a parent class, but the subclass does not need all behaviors provided by the parent class.

- Simply put, the subclass refuses some inherited behaviors (bequest 遗产) of the parent class.

- This code smell may indicate that the inheritance does not make sense, and the subclass is not an example of its parent.

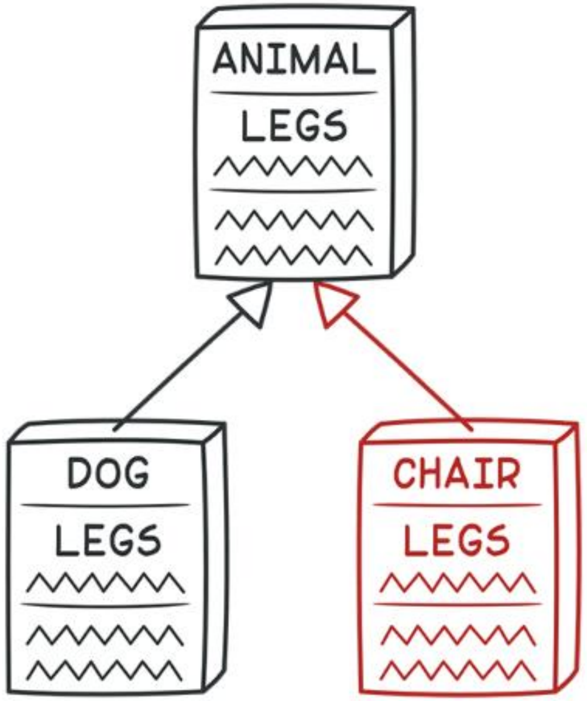


OO ABUSERS - REFUSED BEQUEST

Violation of the Liskov Substitution Principle

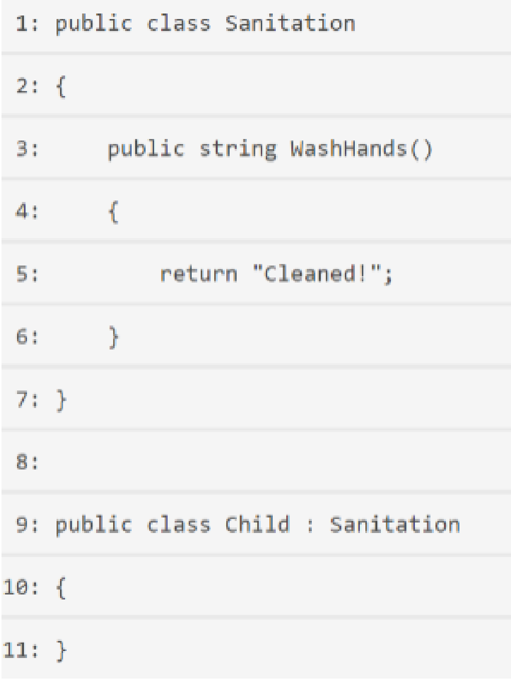
- Objects of subclasses should behave in the same way as the objects of superclass.

- Objects of a superclass should be replaceable with objects of its subclasses without breaking the application.

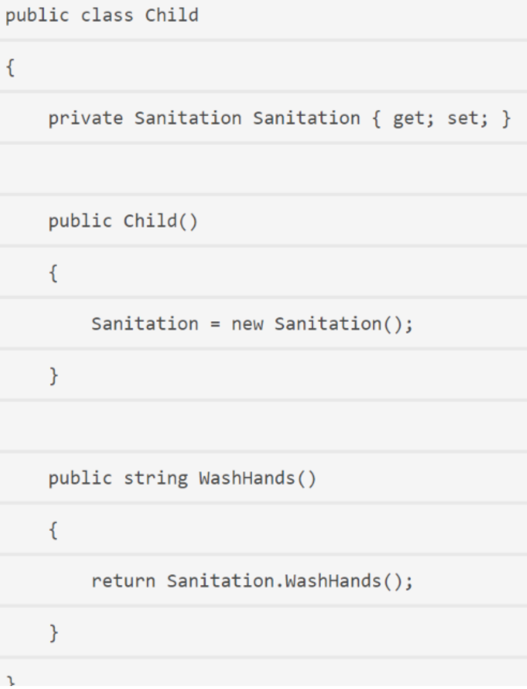


OO ABUSERS - REFUSED BEQUEST

Refactoring: Replace inheritance with delegation



```
public class Sanitation
{
    public string WashHands()
    {
        return "Cleaned!";
    }
}

public class Child : Sanitation
{
}
```

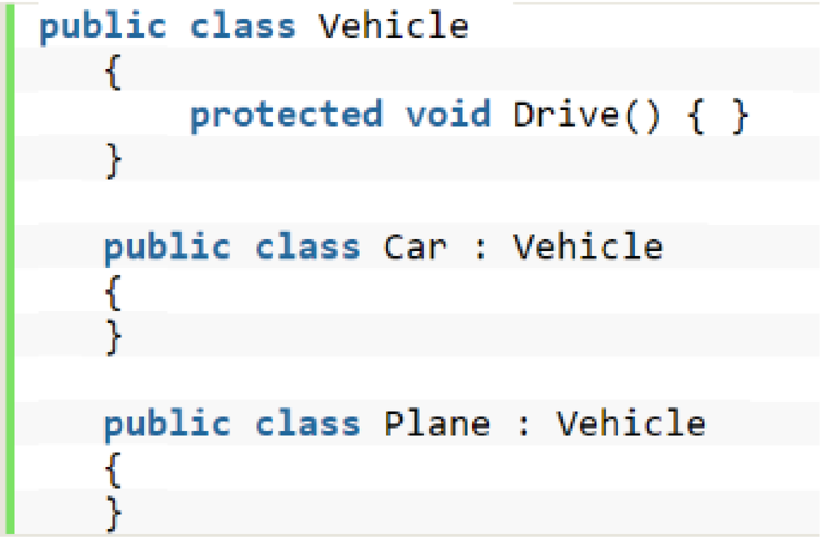


```
public class Child
{
    private Sanitation Sanitation { get; set; }

    public Child()
    {
        Sanitation = new Sanitation();
    }

    public string WashHands()
    {
        return Sanitation.WashHands();
    }
}
```

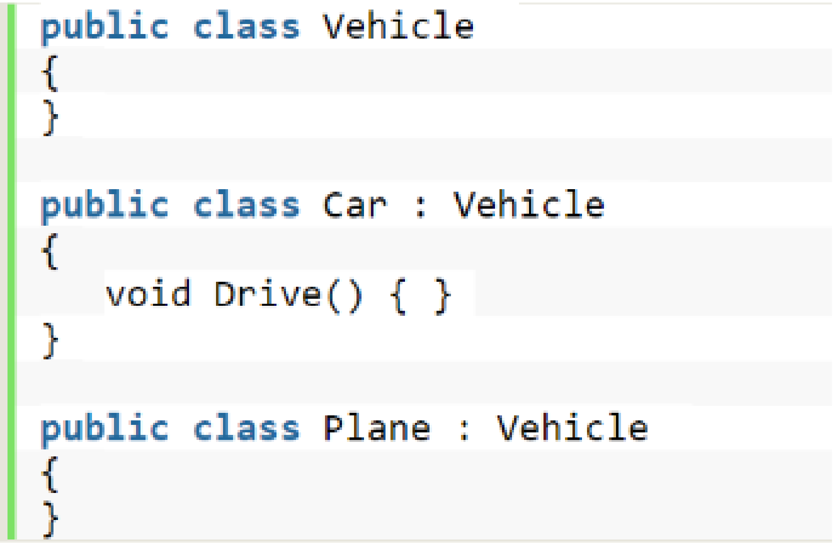
https://prezi.com/p/vwdhx4yp_l3z/code-smell-refused-bequest/

OO ABUSERS - REFUSED BEQUEST

Refactoring: pushdown methods/fields: move methods or properties from the superclass to subclasses where they fit



```
public class Vehicle
   {
       protected void Drive() { }
   }

   public class Car : Vehicle
   {
   }

   public class Plane : Vehicle
   {
   }
```



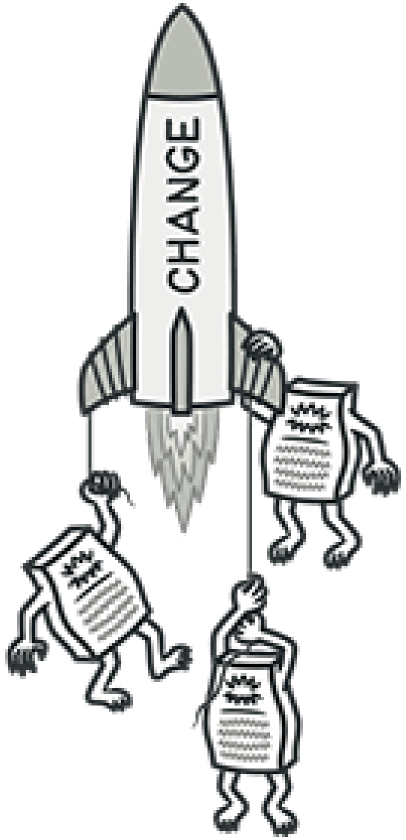
```
public class Vehicle
{
}

public class Car : Vehicle
{
   void Drive() { }
}

public class Plane : Vehicle
{
}
```

https://prezi.com/p/vwdhx4yp_l3z/code-smell-refused-bequest/

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.

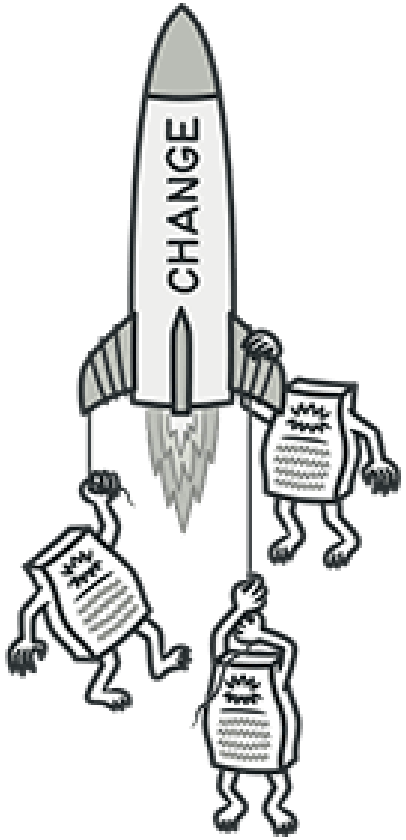


https://refactoring.guru/refactoring/smells

CHANGE PREVENTERS

Violation of the **Single Responsibility Principle**

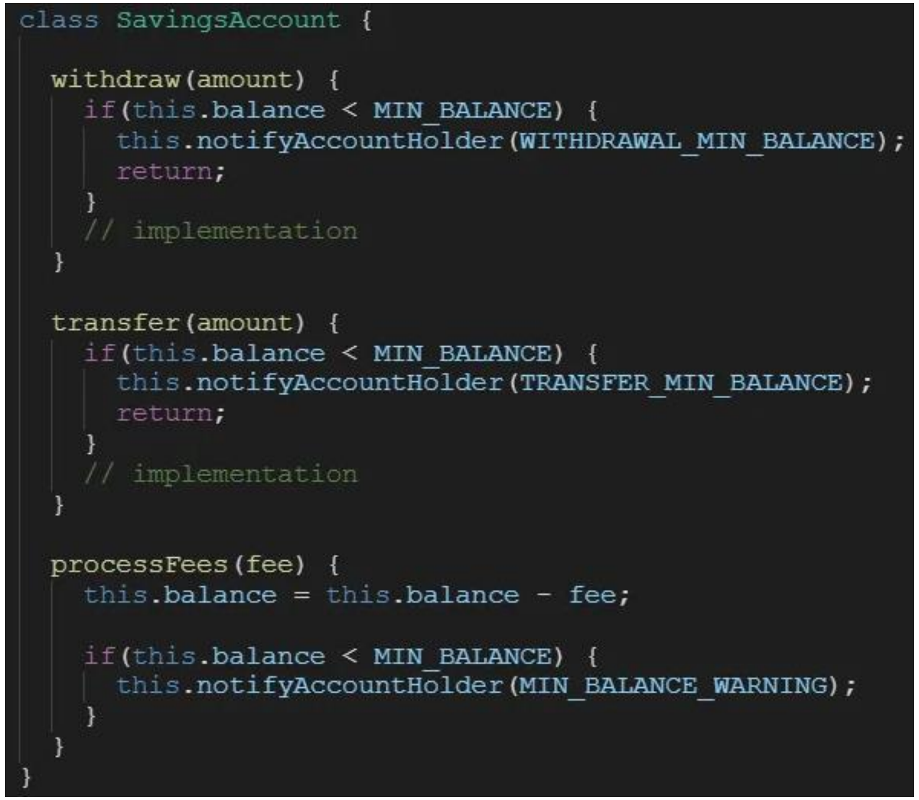
- Every class, module, or function in a program should have one responsibility/purpose in a program.
- Every class, module, or function should have only one reason to change



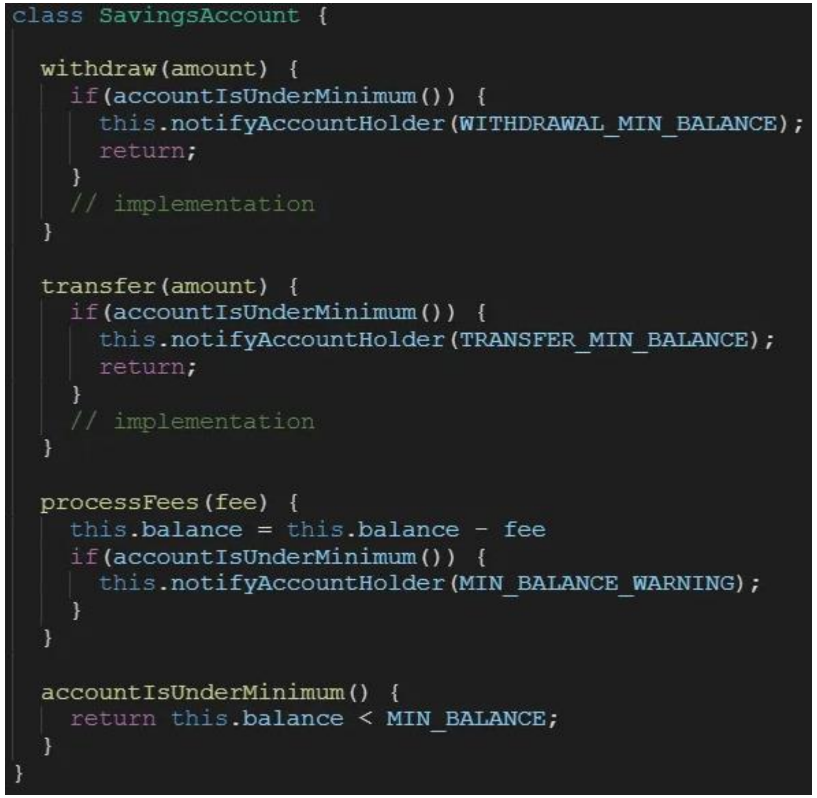
https://refactoring.guru/refactoring/smells

CHANGE PREVENTERS - SHOTGUN SURGERY

Making any modifications requires that you make many small changes to many different places.



```
class SavingsAccount {

  withdraw(amount) {
    if(this.balance < MIN_BALANCE) {
      this.notifyAccountHolder(WITHDRAWAL_MIN_BALANCE);
      return;
    }
    // implementation
  }

  transfer(amount) {
    if(this.balance < MIN_BALANCE) {
      this.notifyAccountHolder(TRANSFER_MIN_BALANCE);
      return;
    }
    // implementation
  }

  processFees(fee) {
    this.balance = this.balance - fee;

    if(this.balance < MIN_BALANCE) {
      this.notifyAccountHolder(MIN_BALANCE_WARNING);
    }
  }
}
```

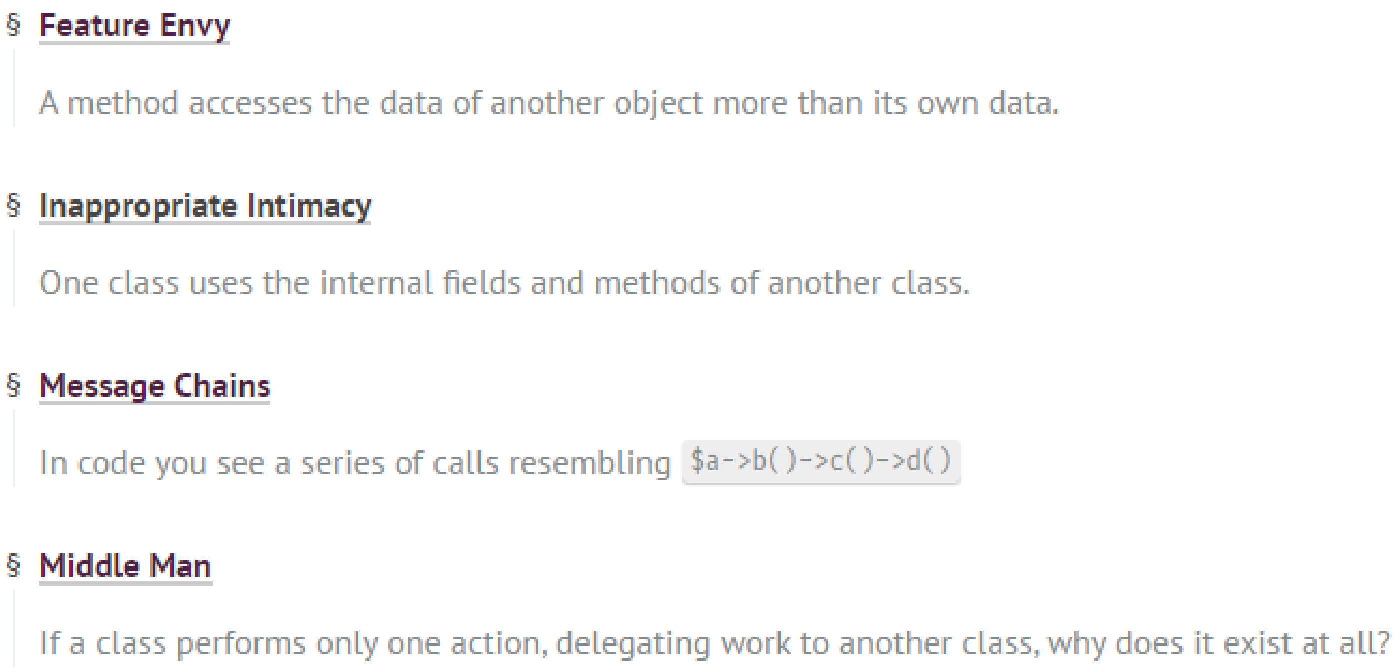


```
class SavingsAccount {

  withdraw(amount) {
    if(accountIsUnderMinimum()) {
      this.notifyAccountHolder(WITHDRAWAL_MIN_BALANCE);
      return;
    }
    // implementation
  }

  transfer(amount) {
    if(accountIsUnderMinimum()) {
      this.notifyAccountHolder(TRANSFER_MIN_BALANCE);
      return;
    }
    // implementation
  }

  processFees(fee) {
    this.balance = this.balance - fee
    if(accountIsUnderMinimum()) {
      this.notifyAccountHolder(MIN_BALANCE_WARNING);
    }
  }

  accountIsUnderMinimum() {
    return this.balance < MIN_BALANCE;
  }
}
```

Possible refactoring: extract method

COUPLERS

Couplers are simply code smells that represent high coupling between classes or entire modules.



§ **Feature Envy**

A method accesses the data of another object more than its own data.

§ **Inappropriate Intimacy**

One class uses the internal fields and methods of another class.

§ **Message Chains**

In code you see a series of calls resembling `$a->b()->c()->d()`

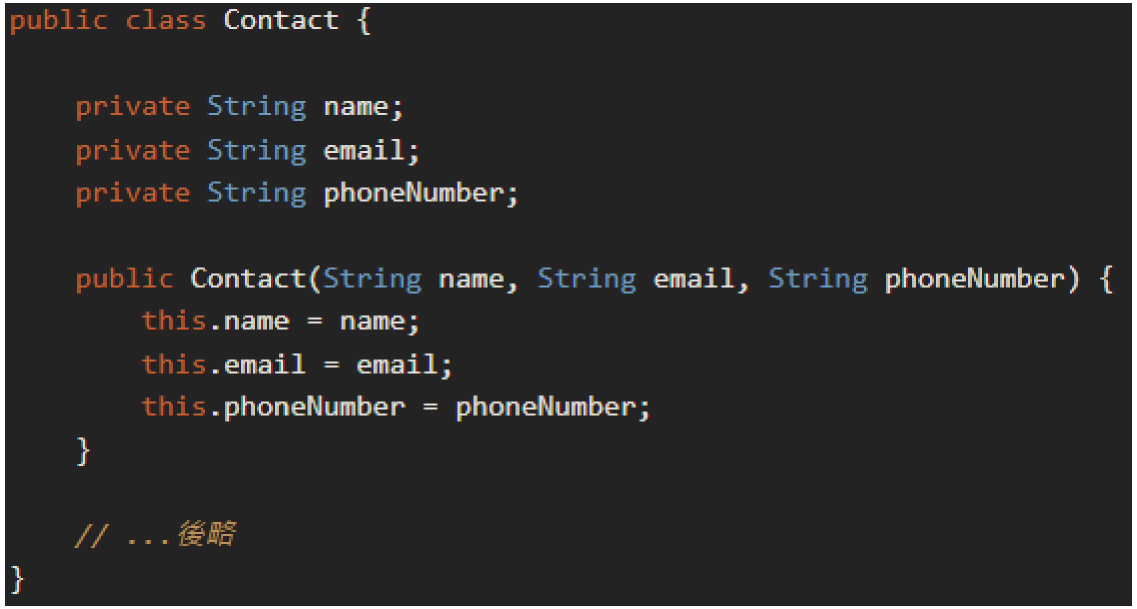
§ **Middle Man**

If a class performs only one action, delegating work to another class, why does it exist at all?

https://refactoring.guru/refactoring/smells/couplers

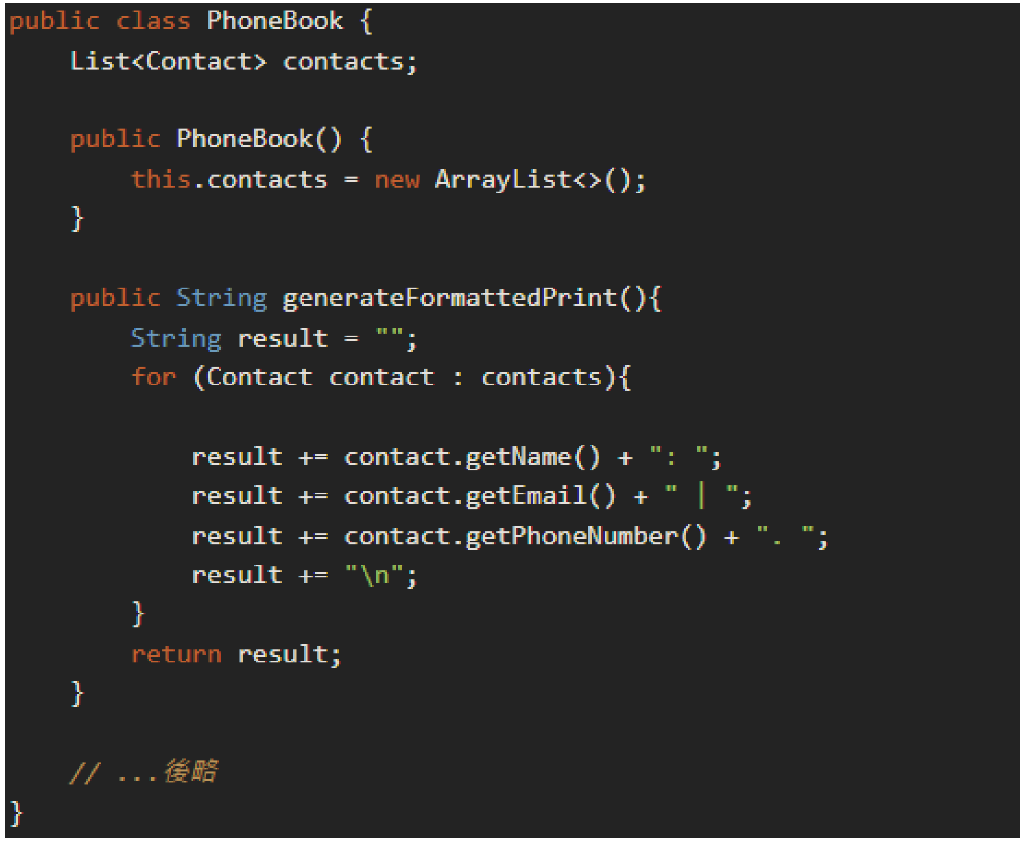
COUPLERS - FEATURE ENVY

A method accesses the data of another object more than its own data.



```java
public class Contact {

    private String name;
    private String email;
    private String phoneNumber;

    public Contact(String name, String email, String phoneNumber) {
        this.name = name;
        this.email = email;
        this.phoneNumber = phoneNumber;
    }

    // ...後略
}
```

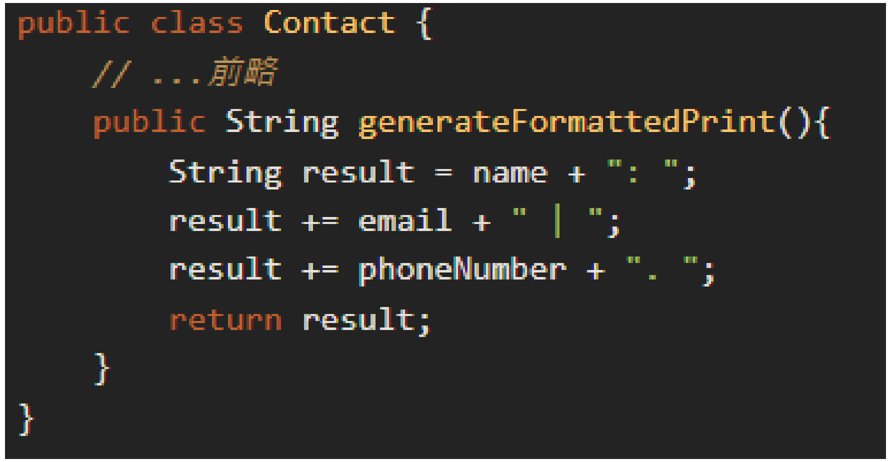
https://ithelp.ithome.com.tw/articles/10195504



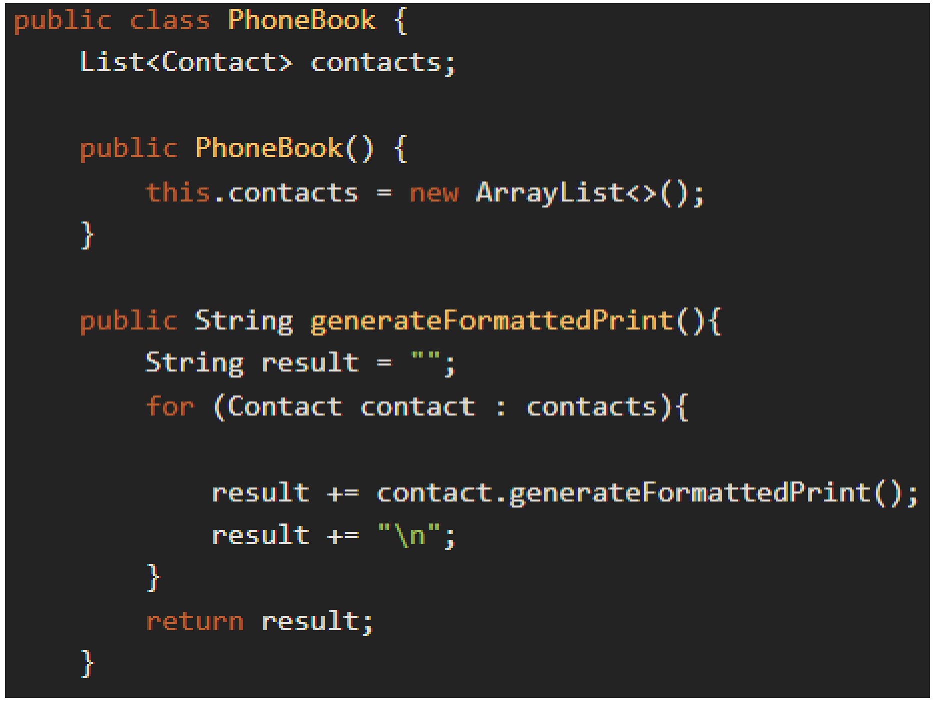
```java
public class PhoneBook {
    List<Contact> contacts;

    public PhoneBook() {
        this.contacts = new ArrayList<>();
    }

    public String generateFormattedPrint(){
        String result = "";
        for (Contact contact : contacts){

            result += contact.getName() + ": ";
            result += contact.getEmail() + " | ";
            result += contact.getPhoneNumber() + ". ";
            result += "\n";
        }
        return result;
    }

    // ...後略
}
```

COUPLERS - FEATURE ENVY

A method accesses the data of another object more than its own data.



```
public class Contact {
    // ...前略
    public String generateFormattedPrint(){
        String result = name + ": ";
        result += email + " | ";
        result += phoneNumber + ". ";
        return result;
    }
}
```



```
public class PhoneBook {
    List<Contact> contacts;

    public PhoneBook() {
        this.contacts = new ArrayList<>();
    }

    public String generateFormattedPrint(){
        String result = "";
        for (Contact contact : contacts){

            result += contact.generateFormattedPrint();
            result += "\n";
        }
        return result;
    }
```

https://ithelp.ithome.com.tw/articles/10195504

DISPENSIBLES

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

- Examples
  - Duplicate code
  - Dead code (unused and obsolete code)
  - Lazy class (classes that don’t do enough to earn your attention)

FURTHER NOTES ON CODE SMELLS

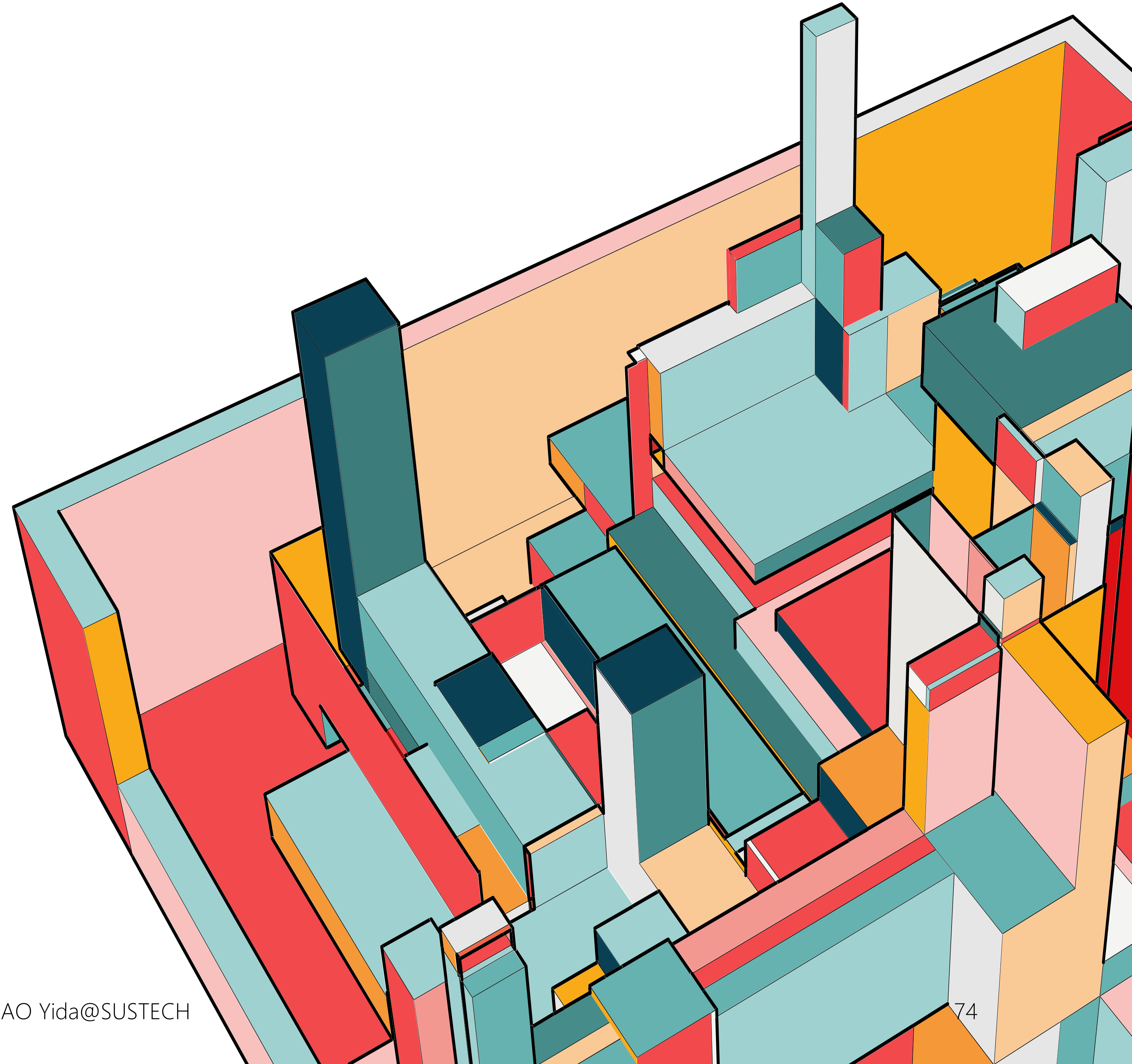
- Code smells are usually **not bugs;** they are not technically incorrect and do not prevent the program from functioning
  - Yet, they indicate weaknesses in design that may slow down development or increase the risk of bugs or failures in the future

- Code smells don't always indicate a problem. You must look deeper to see if there is an underlying problem there
  - For example, some long methods or large classes are just fine.

READINGS

- Chapter 9. Software Evolution. Software Engineering by Ian Sommerville, 10th edition

- Chapter 15. Deprecation. Software Engineering at Google by Titus Winters et al.



NEXT

- Software Documentation