



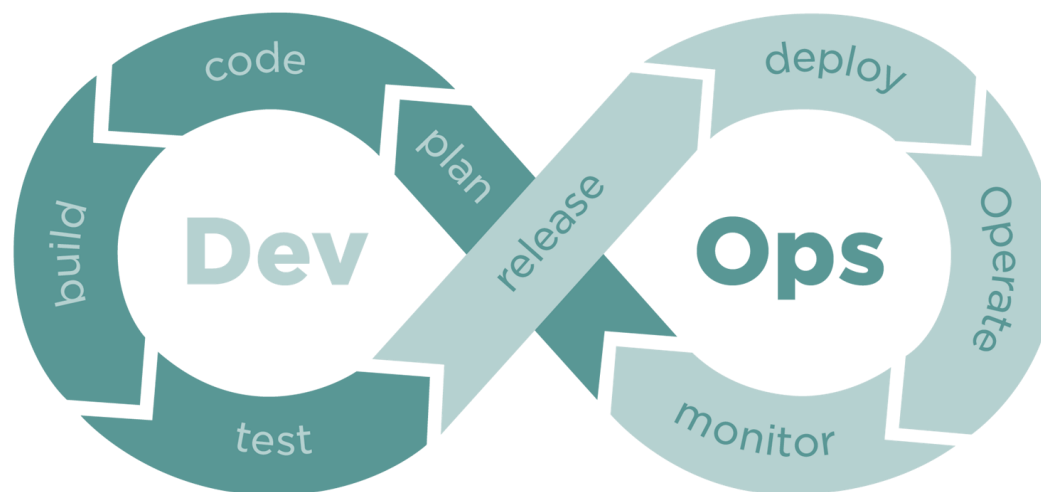
南方科技大学  
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

# CS304 SOFTWARE ENGINEERING

Yida Tao

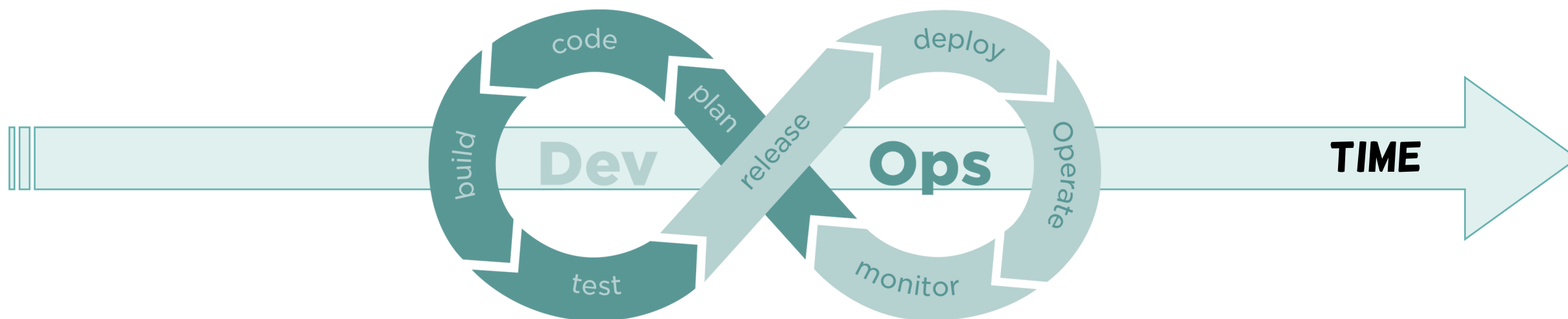
taoyd@sustech.edu.cn

# WHERE ARE WE NOW?



What happens after deploying our software to production?  
How do we maintain the current release?  
How do we start the next release?

# WHERE ARE WE NOW?



Software begins a new phase of life: one where it needs to be monitored, updated, corrected, adapted, and improved — often for years.



# LECTURE 13

- Software maintenance (软件维护)
  - Maintainability
  - Technical debt
  - Refactoring
  - Re-architecting
- Software evolution (软件演化)
  - Legacy systems
  - Modernization
  - Deprecation



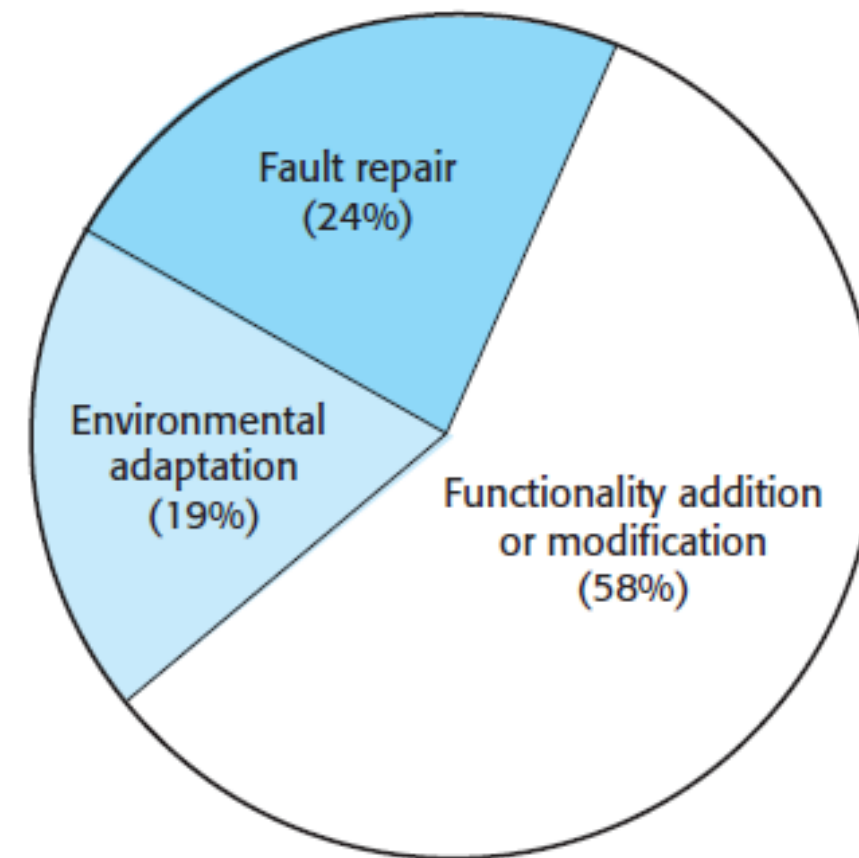
# SOFTWARE MAINTENANCE

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

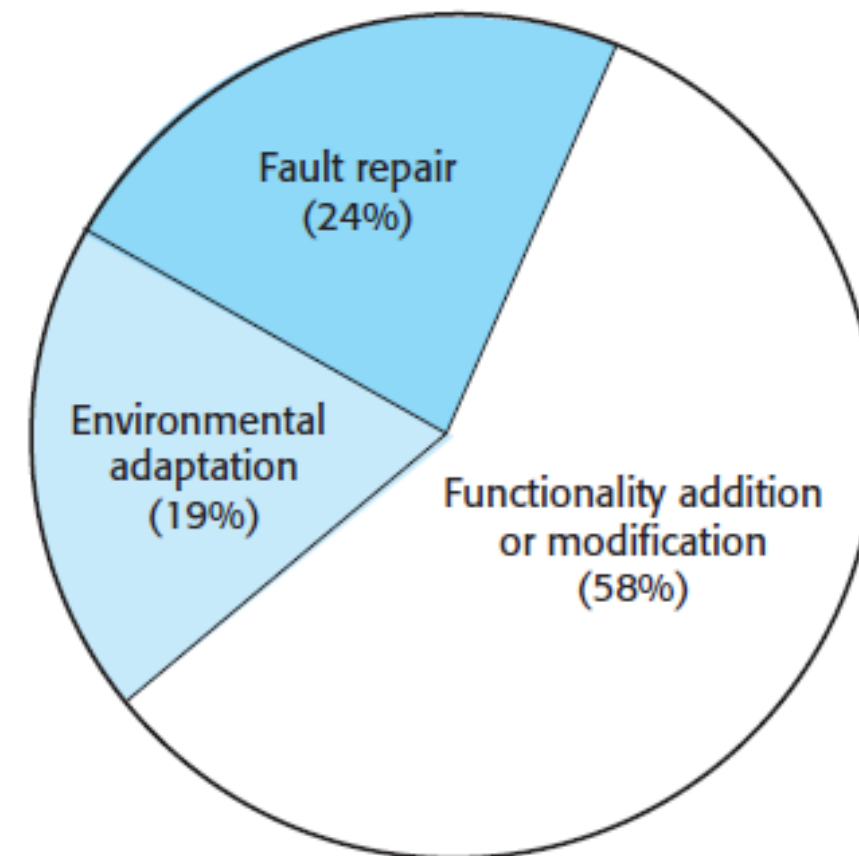
- *Coding errors* are usually relatively cheap to correct;
- *Design errors* are more expensive because they may involve rewriting several program components.
- *Requirements errors* are the most expensive to repair because extensive system redesign may be necessary.



# MAINTENANCE EFFORT DISTRIBUTION

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

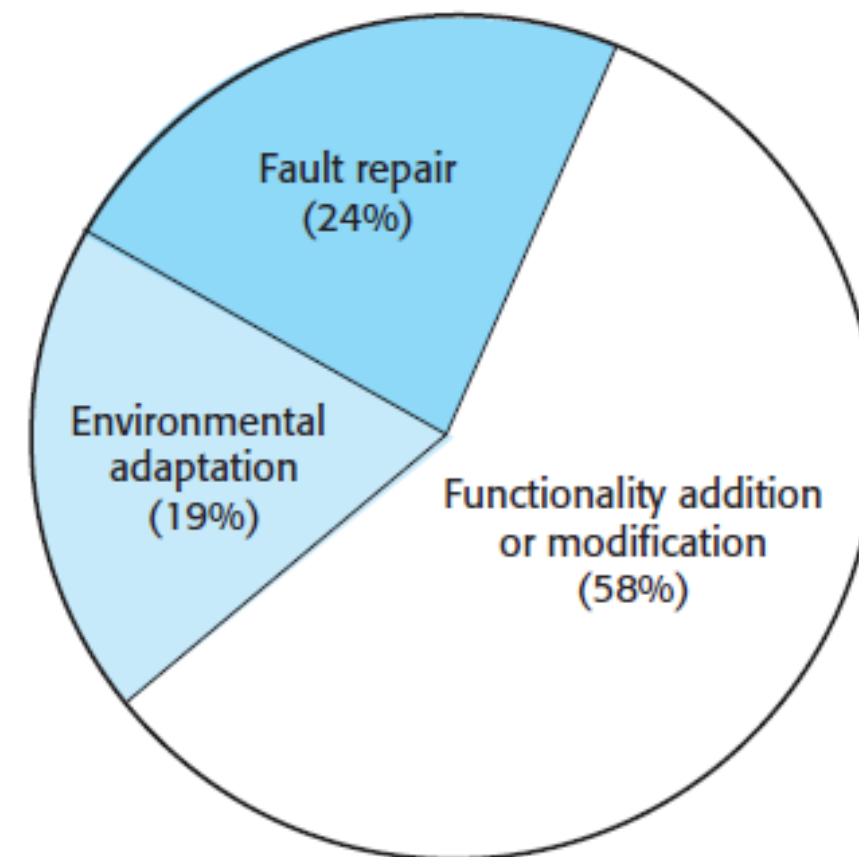
- This type of maintenance is required when some aspect of a system's environment, such as the hardware, the platform operating system, or other support software, changes.
- Application systems may have to be modified to cope with these environmental changes.



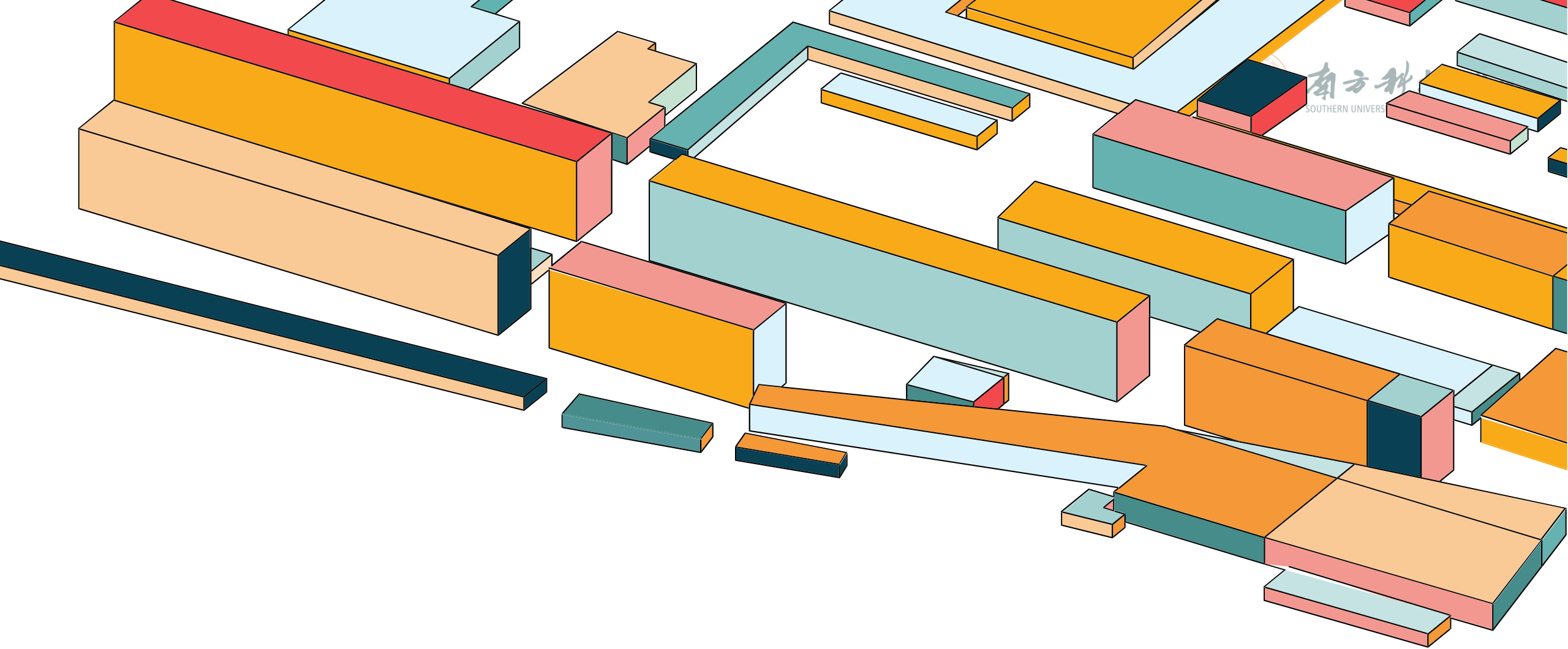
# MAINTENANCE EFFORT DISTRIBUTION

**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.







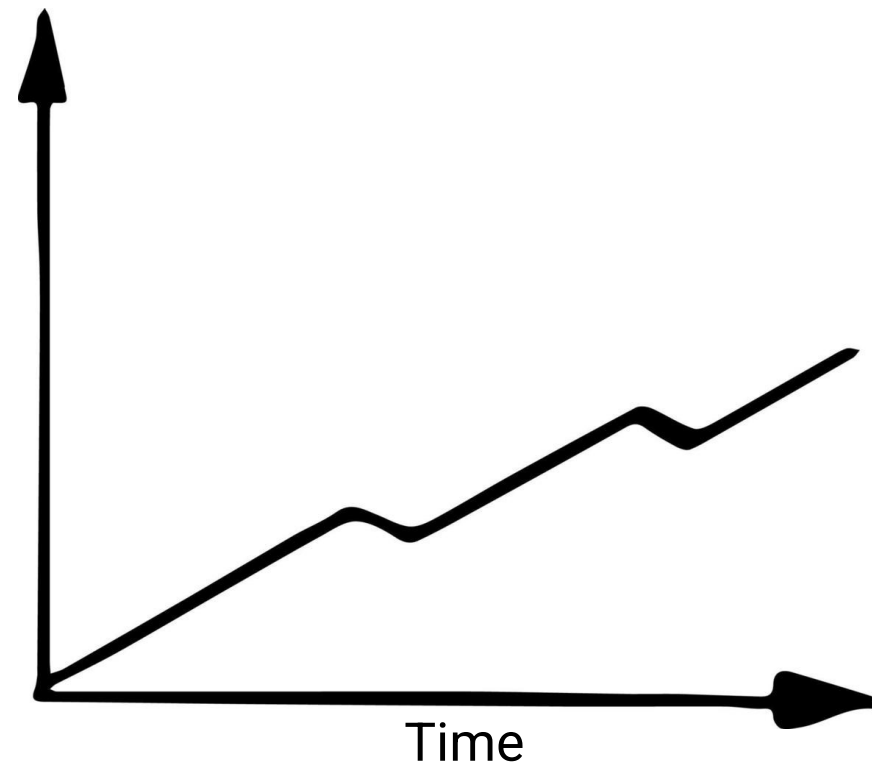
# HOW TO MEASURE MAINTAINABILITY?

# 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.

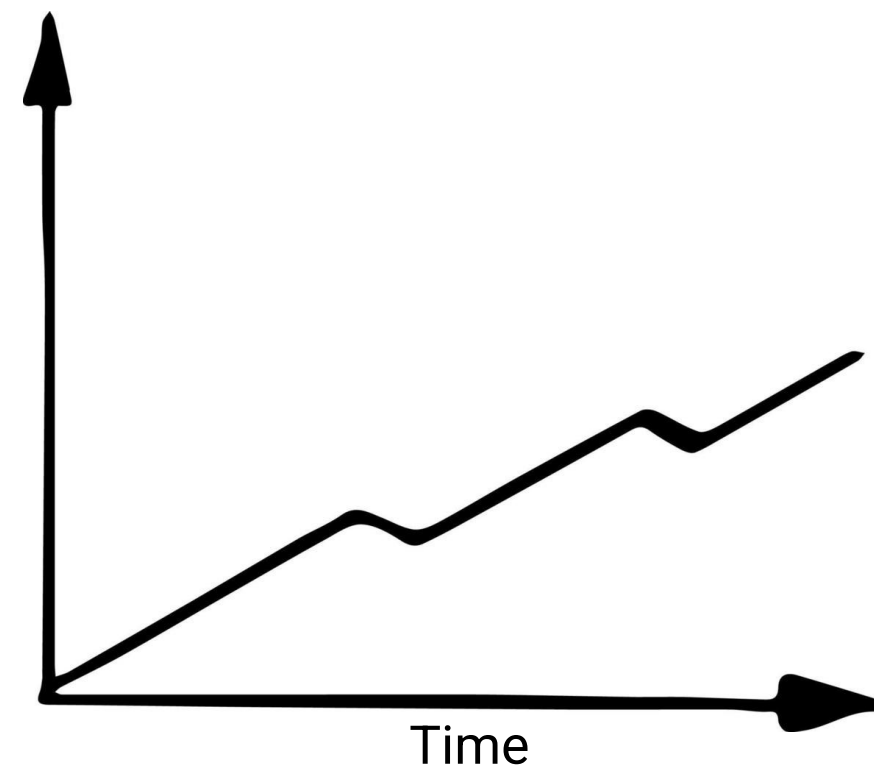




# EXAMPLE METRICS FOR MAINTAINABILITY

## 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**

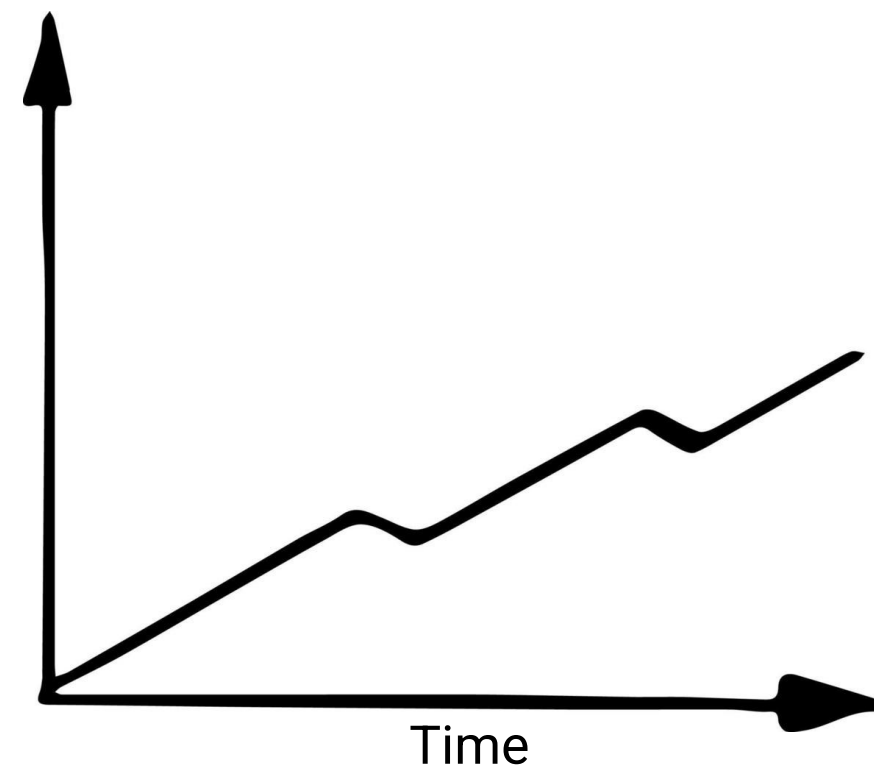




# EXAMPLE METRICS FOR MAINTAINABILITY

## 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.**

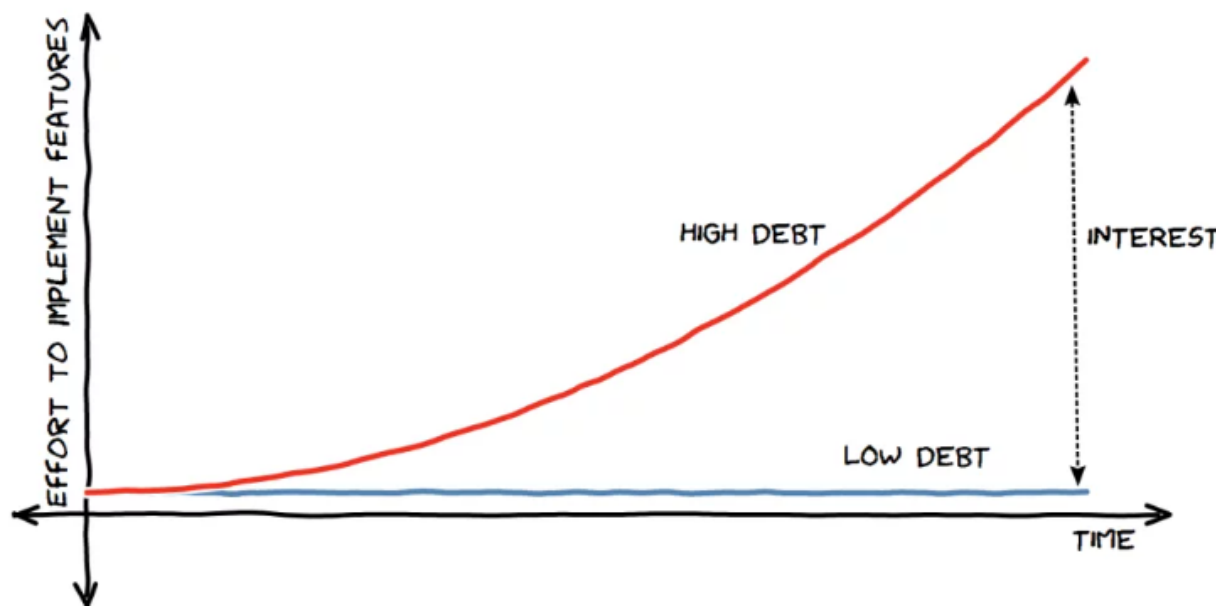


# TECHNICAL DEBT (技术债)

Technical debt refers to the **long-term cost** of making **short-term or suboptimal decisions** in software design, architecture, or code.

These decisions might make things faster in the short term, but they “borrow time” from the future, requiring extra work later to fix, refactor, or rework.

## INTEREST ON TECHNICAL DEBT



It's getting harder and harder to update the software



# TECHNICAL DEBT (技术债)

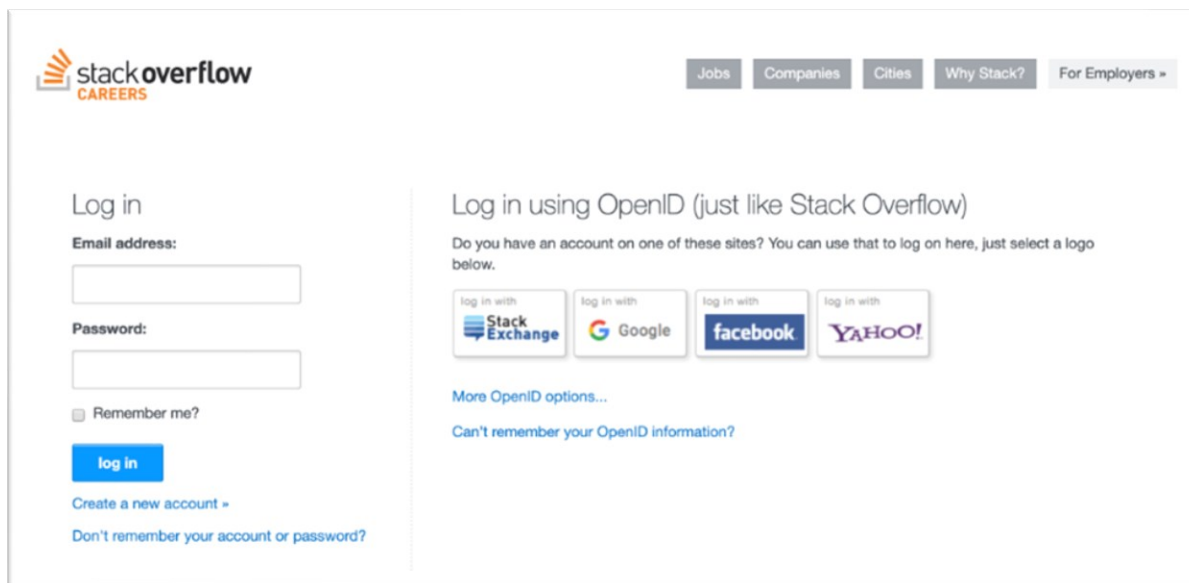
## Quick and dirty fixes

- Code is **rushed** to meet deadlines.
- Corners are cut: minimal testing, no documentation, poor design.
- Example: Hardcoding logic or skipping error handling.

## Well-intentioned past decisions that don't scale well anymore

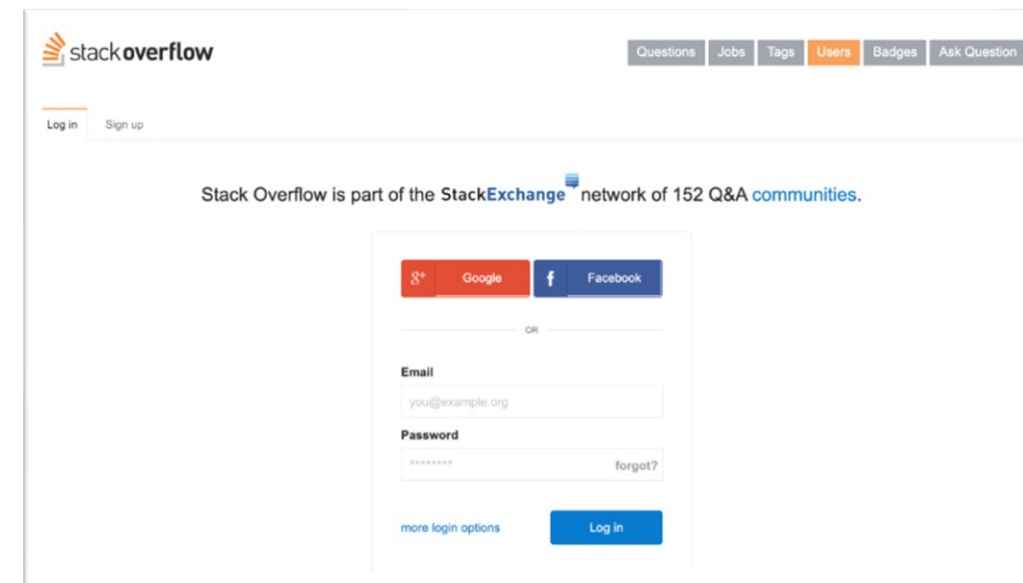
- The decision was right at the time, given context.
- But as the system or business evolves, it becomes a **bottleneck** or **liability**.

# CASE STUDY: TECHNICAL DEBT AT STACK OVERFLOW



The old login page features a simple layout. On the left, there is a 'Log in' section with fields for 'Email address' and 'Password', a 'Remember me?' checkbox, and a 'log in' button. Below these are links for 'Create a new account' and 'Don't remember your account or password?'. On the right, there is a 'Log in using OpenID (just like Stack Overflow)' section. It includes a text prompt: 'Do you have an account on one of these sites? You can use that to log on here, just select a logo below.' Below this are four buttons: 'log in with Stack Exchange', 'log in with Google', 'log in with facebook', and 'log in with YAHOO!'. Further down are links for 'More OpenID options...' and 'Can't remember your OpenID information?'. The top navigation bar includes links for 'Jobs', 'Companies', 'Cities', 'Why Stack?', and 'For Employers'.

Old login page



The new login page has a more complex layout. At the top, there is a navigation bar with links for 'Questions', 'Jobs', 'Tags', 'Users', 'Badges', and 'Ask Question'. Below this, there are 'Log in' and 'Sign up' links. A text line states: 'Stack Overflow is part of the StackExchange network of 152 Q&A communities.' Below this is a section with buttons for 'Google' and 'Facebook'. A 'OR' separator is followed by an 'Email' field (with 'you@example.org' as a placeholder), a 'Password' field (with a 'forgot?' link), and a 'Log in' button. There is also a 'more login options' link. The overall design is more cluttered than the old version.

New login page

*This update should be easy, but . . . . .*

*It is more complex than expected, and takes much longer than expected.*

Source: <https://techdebtguide.com/case-study-stack-overflow>

## CASE STUDY: TECHNICAL DEBT AT STACK OVERFLOW

- **Split codebases:** The code for handling user login and registration was split across two different codebases
- **Functional limitations:** Email addresses were stored in two separate databases; users couldn't reset the password with their new email addresses
- **Outdated Dependencies:** *CareersAuth* had not been updated for several years, necessitating time-consuming updates to various dependencies to make it operational on a local development environment

Source: <https://techdebtguide.com/case-study-stack-overflow>





# CASE STUDY: TECHNICAL DEBT AT STACK OVERFLOW

## Historical reasons

- A technical decision made in 2008: use OpenID for Stack Overflow authentication. This was a technical decision that made sense at the time.
- In 2009, Stack Overflow launched Careers, which forks from SO but uses CareersAuth to address problems with OpenID.
- In 2011, Stack Overflow implements a new authentication mechanism.
- In 2016, OpenID is dying.

Source: <https://techdebtguide.com/case-study-stack-overflow>

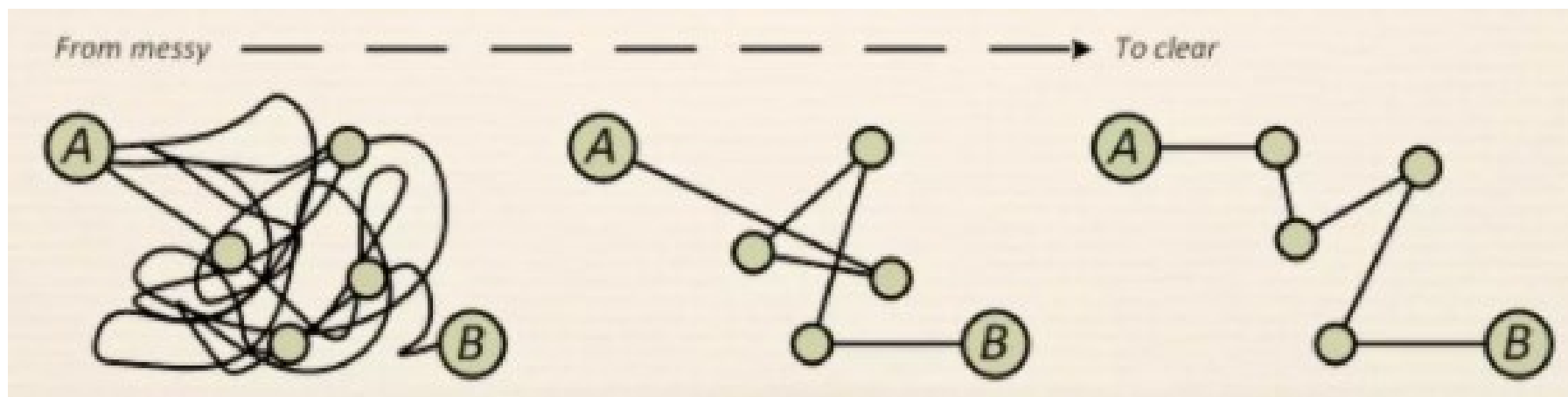


# LECTURE 13

- Software maintenance (软件维护)
  - Maintainability
  - Technical debt
  - Refactoring
  - Re-architecting
- Software evolution (软件演化)
  - Legacy systems
  - Modernization
  - Deprecation

# 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 simplifying code by removing unnecessary complexity or duplication.

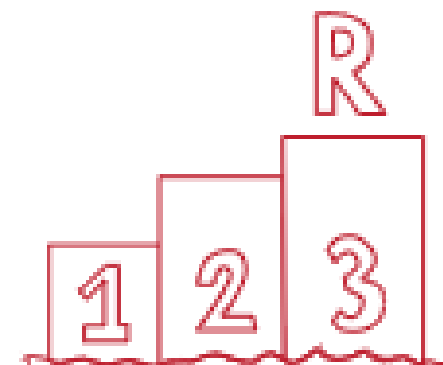
- To find bugs and write more robust code
- To increase readability and productivity (program faster) on a long term basis

- To reduce cost of software maintenance by modularizing it into smaller, more focused units.
- 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.



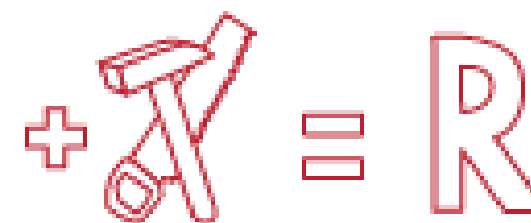
**Rule of Three**

<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.

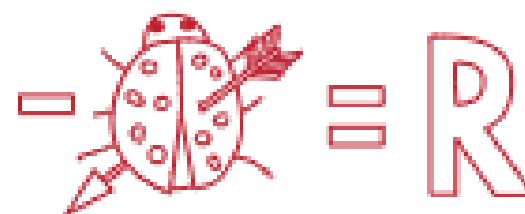


**When adding a feature**

<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.

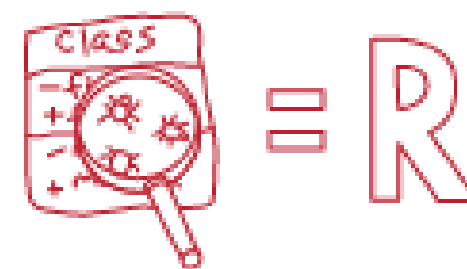


When fixing a bug

<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

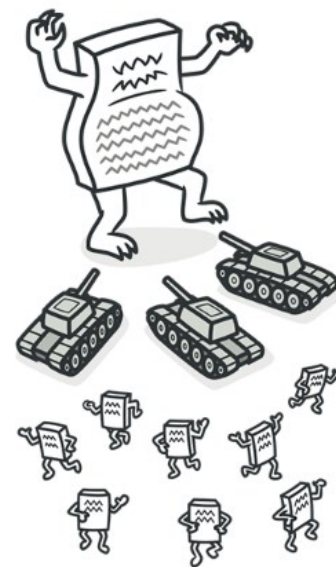


# TYPES OF CODE SMELLS

- 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.

```
void printTaskAssignments() {  
    printUser();  
  
    System.out.println("task name:" + task);  
    System.out.println("task desc" + description);  
}
```

```
void printTaskAssignment() {  
    printUser();  
    printTask();  
}  
  
void printTask() {  
    System.out.println("task name:" + task);  
    System.out.println("task description" + description);  
}
```



# BLOATERS - LONG PARAMETER LIST

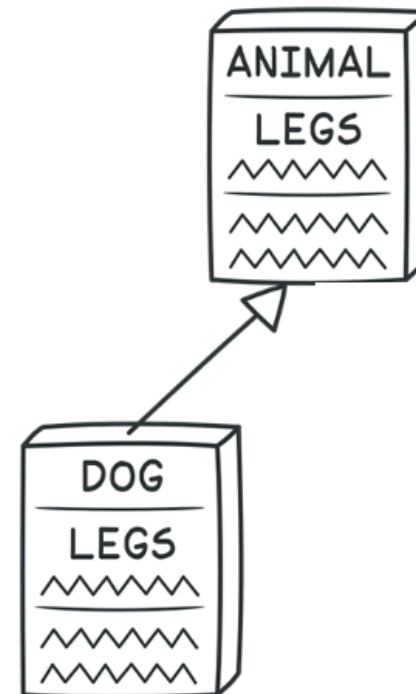
- A method has more than 3 or 4 parameters, which are hard to understand and error-prone
- Refactorings: Replace parameter with method call, etc.

```
double basePrice = itemPrice * quantity;  
double fees = this.getFee();  
double discount = this.getDiscount();  
double finalPrice = getPayment(basePrice, fees, discount);
```

```
double basePrice = itemPrice * quantity;  
double finalPrice = getDiscountPrice(basePrice);
```

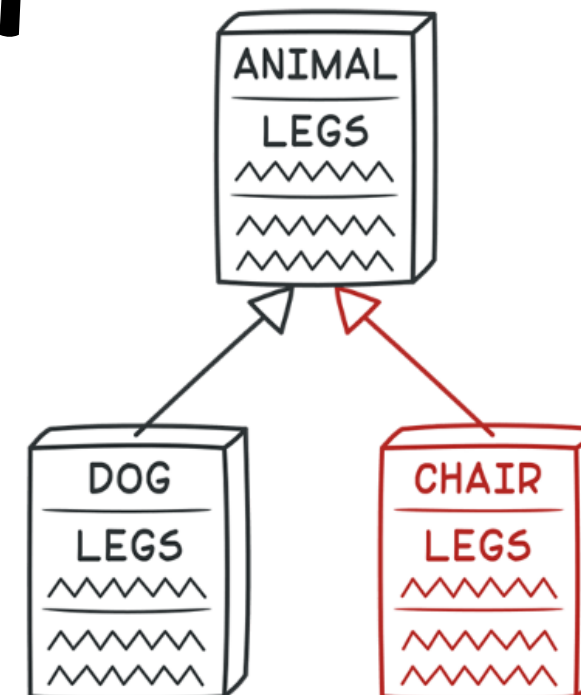
# OO ABUSERS

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



# OO ABUSERS - REFUSED BEQUEST

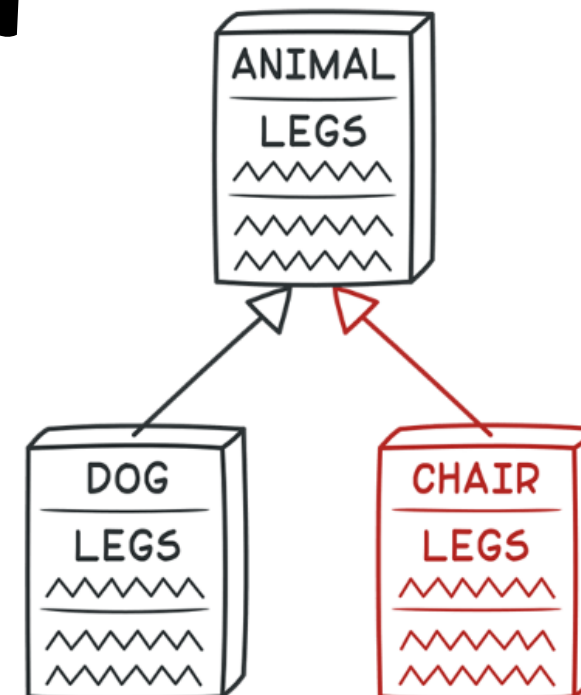
- A subclass inherits from a parent class, but the subclass does not need all behaviors provided by the parent class.
- Simply put, the subclass refuses some inherited behaviors (bequest 遗产) of the parent class.
- This code smell may indicate that **the inheritance does not make sense**, and the subclass 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 "hands cleaned";  
    }  
}  
  
class Child extends Sanitation{  
  
}
```

```
public class Child {  
    private Sanitation sanitation;  
  
    public Child{  
        sanitation = new Sanitation();  
    }  
  
    public String washHands(){  
        return sanitation.washHands();  
    }  
  
}
```

# 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
{
}

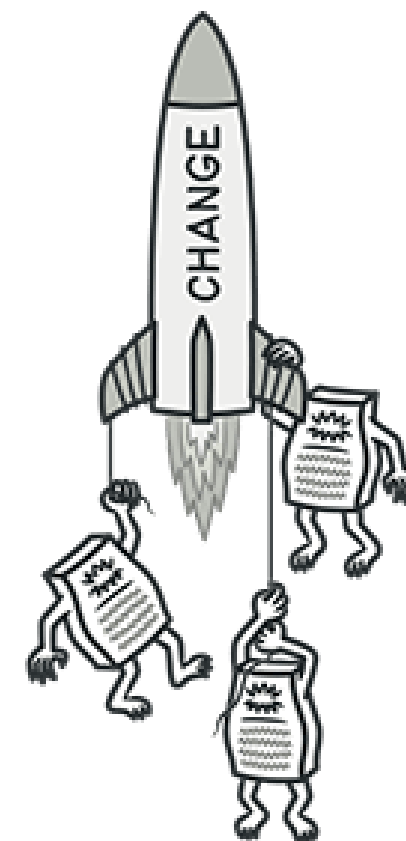
public class Car : Vehicle
{
    void Drive() { }
}

public class Plane : Vehicle
{
}
```

[https://prezi.com/p/vwdhx4yp\\_l3z/code-smell-refused-bequest/](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.

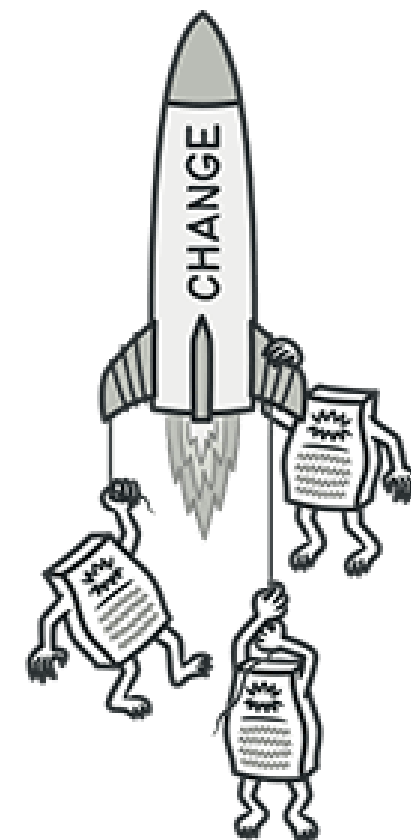


<https://refactoring.guru/refactoring/smells>

# 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.

```
public class Account {  
    public void withdraw(double amount){  
        if(this.balance < MIN_BALANCE){  
            notifyUser();  
        }  
        // code for withdraw money  
    }  
  
    public void deposit(double amount){  
        // code for deposit  
  
        if(this.balance < MIN_BALANCE){  
            notifyUser();  
        }  
    }  
  
    public void transfer(Account to, double amount){  
        if(this.balance < MIN_BALANCE){  
            notifyUser();  
        }  
        // code for transfer money  
    }  
}
```

```
public class Account {  
  
    public void checkBalance(){  
        if(this.balance < MIN_BALANCE){  
            notifyUser();  
        }  
    }  
  
    public void withdraw(double amount){  
        checkBalance();  
        // code for withdraw money  
    }  
  
    public void deposit(double amount){  
        // code for deposit  
        checkBalance();  
    }  
  
    public void transfer(Account to, double amount){  
        checkBalance();  
        // code for transfer money  
    }  
}
```

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.

```
public class Contact {  
    String name;  
    String phone;  
    String email;  
  
    public Contact(String name, String phone, String email) {  
        this.name = name;  
        this.phone = phone;  
        this.email = email;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    // other getters.....  
}
```

```
public class Phonebook {  
    List<Contact> contactList;  
  
    public Phonebook() {  
        this.contactList = new ArrayList<Contact>();  
    }  
  
    public String toString(){  
        String result="";  
        for(Contact contact:contactList){  
            result += contact.getName() + ": ";  
            result += " " + contact.getPhone() + "\n";  
            result += contact.getEmail() + "\n";  
        }  
        return result;  
    }  
}
```



# DISPENSIBLES

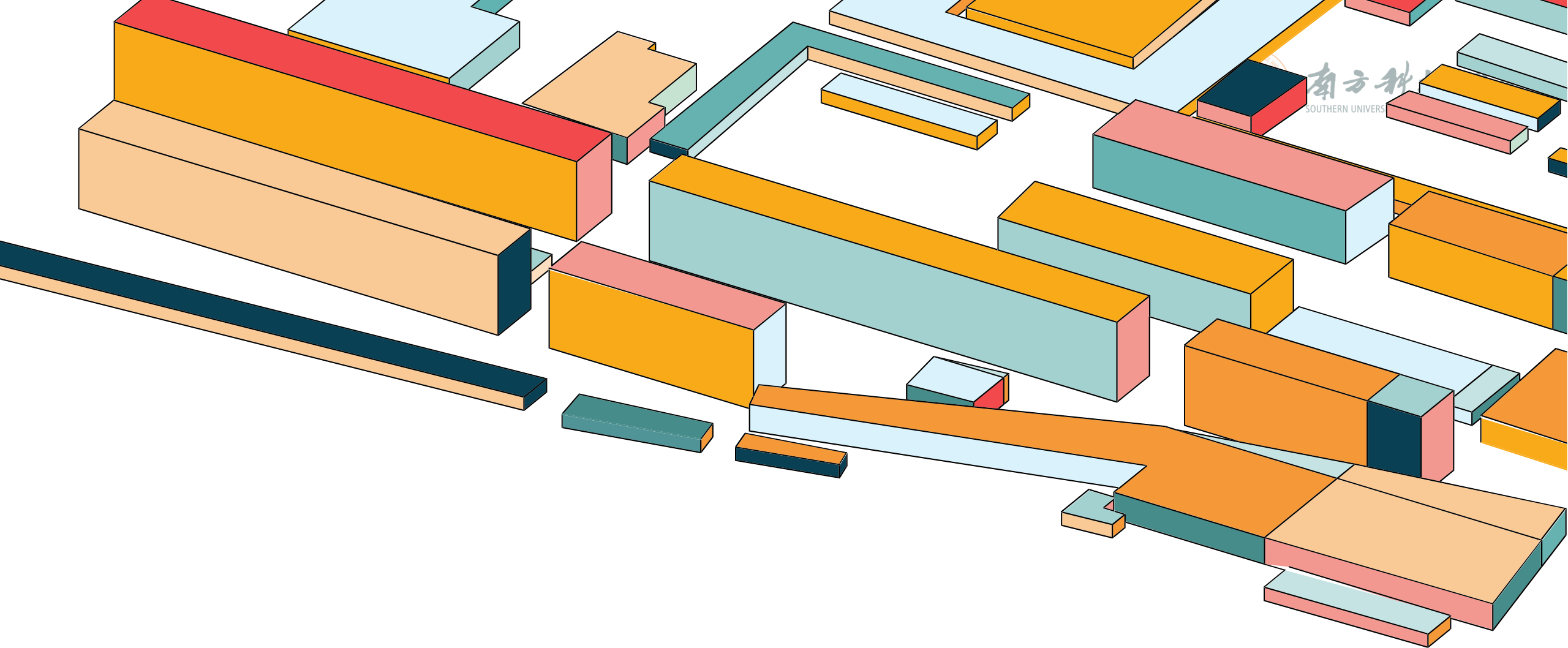
- A dispensable is something pointless and unneeded whose absence would make the code cleaner, more efficient and easier to understand.
- Examples
  - Duplicate code
  - Dead code (unused and obsolete code)
  - Lazy class (classes that don't do enough to earn your attention)





# 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.



# RE-ARCHITECTING



# WHEN TO RE-ARCHITECT MONOLITHICS?

- If monolithic architecture has the following problems:
  - Slow delivery
  - Buggy releases
  - Poor scalability
- ... and the same problems still persist if you've already tried the following:
  - Improved deficient software development process
  - Replaced manual testing by automated testing
  - Optimized other aspects without changing the architecture



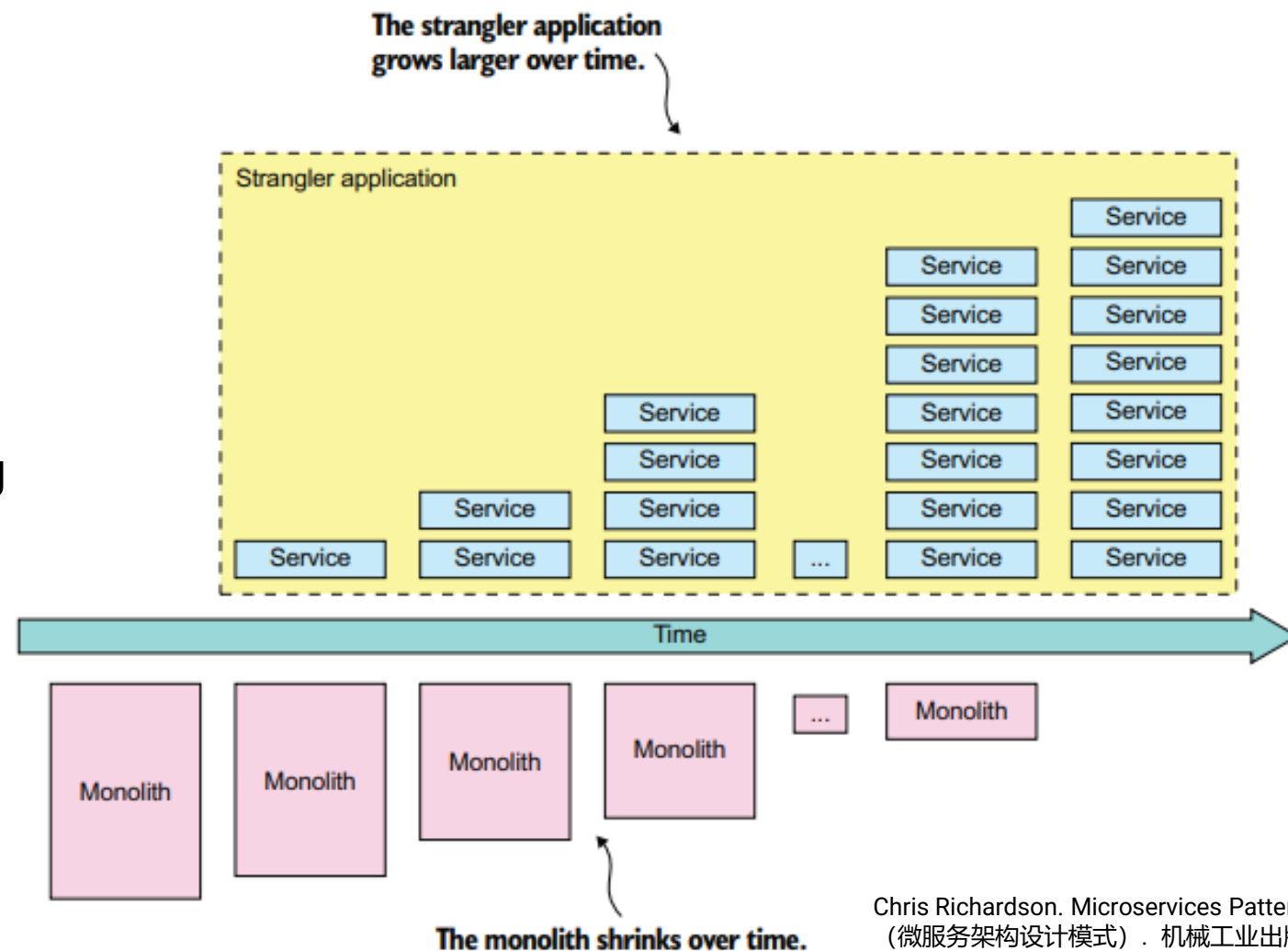
# RE-ARCHITECT TO MICROSERVICES

**Strangler pattern:** instead of developing a new microservice application from scratch, a better approach should be **incrementally and gradually** refactoring the monolithic application.



# STRATEGIES

1. Implement new features as services
2. Separate the presentation tier and backend
3. Break up the monolithic by extracting functionality into services

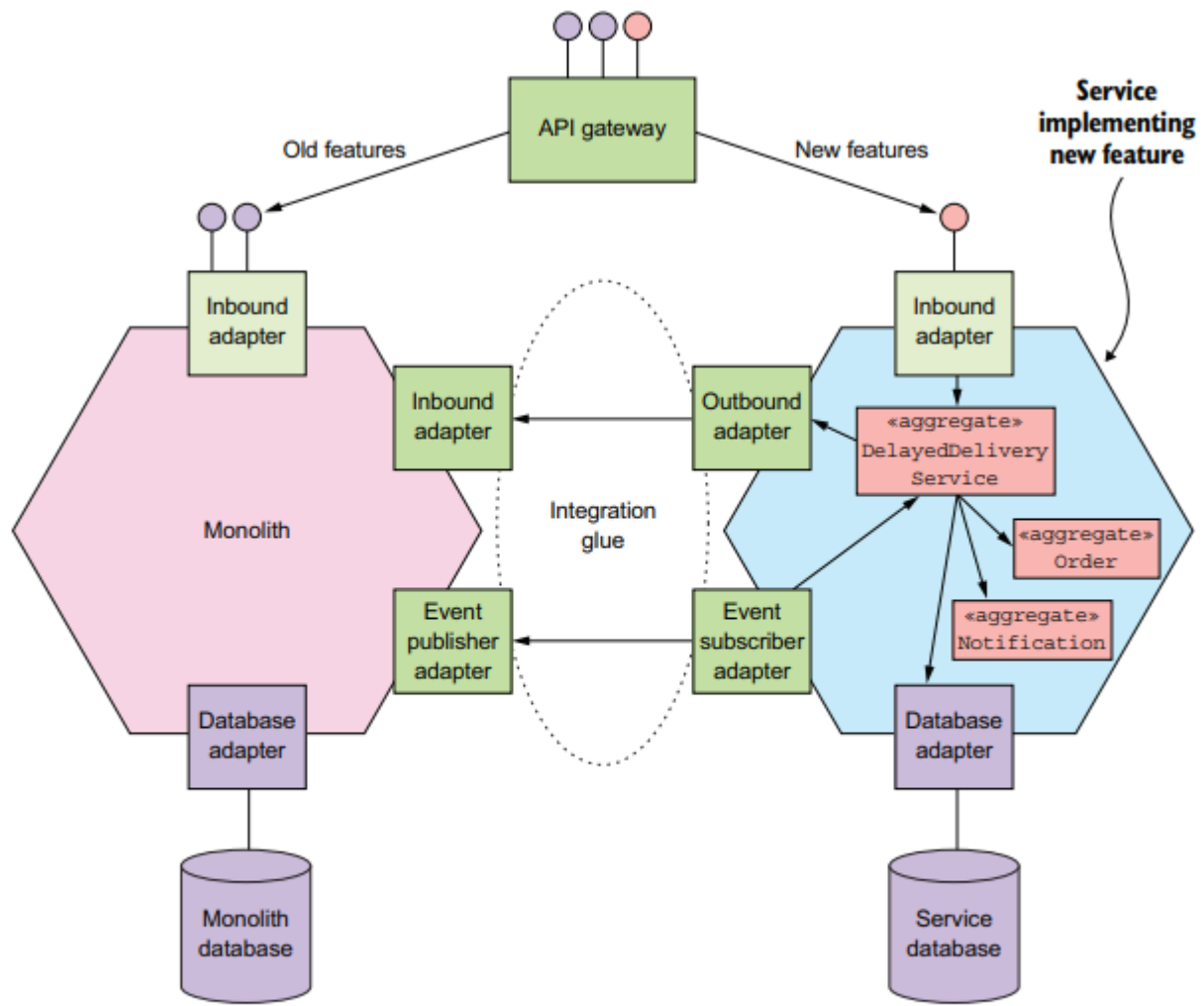


Chris Richardson. Microservices Patterns  
(微服务架构设计模式). 机械工业出版社

# REFACTORING STRATEGY I

## Implement new features as services

Chris Richardson. Microservices Patterns (微服务架构设计模式) . 机械工业出版社



**Figure 13.2** A new feature is implemented as a service that's part of the strangler application. The integration glue integrates the service with the monolith and consists of adapters that implement synchronous and asynchronous APIs. An API gateway routes requests that invoke new functionality to the service.

# REFACTORING STRATEGY 2

Separate presentation tier from the backend

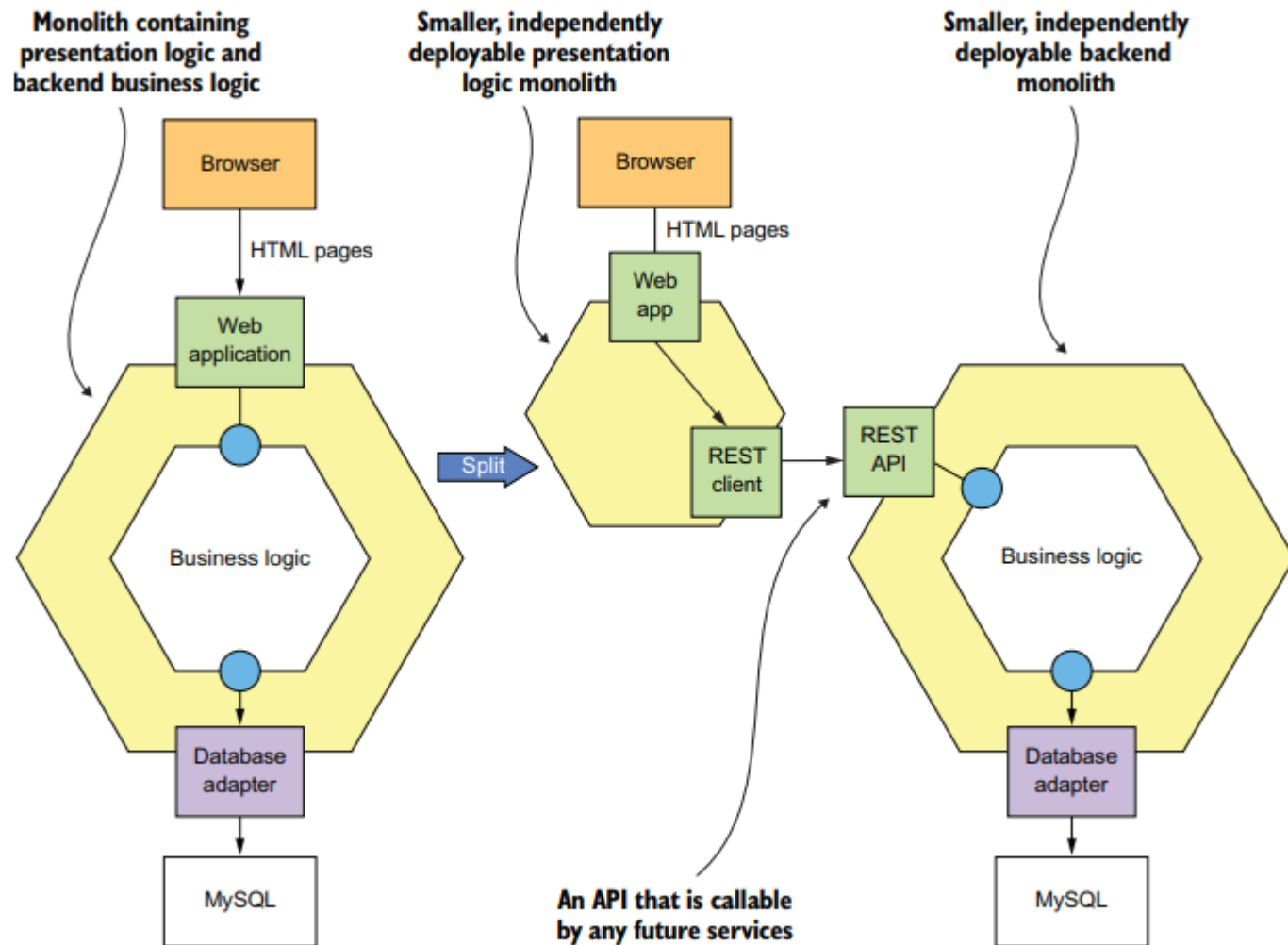


Figure 13.3 Splitting the frontend from the backend enables each to be deployed independently. It also exposes an API for services to invoke.

# REFACTORING STRATEGY 3

Extract business capabilities into services

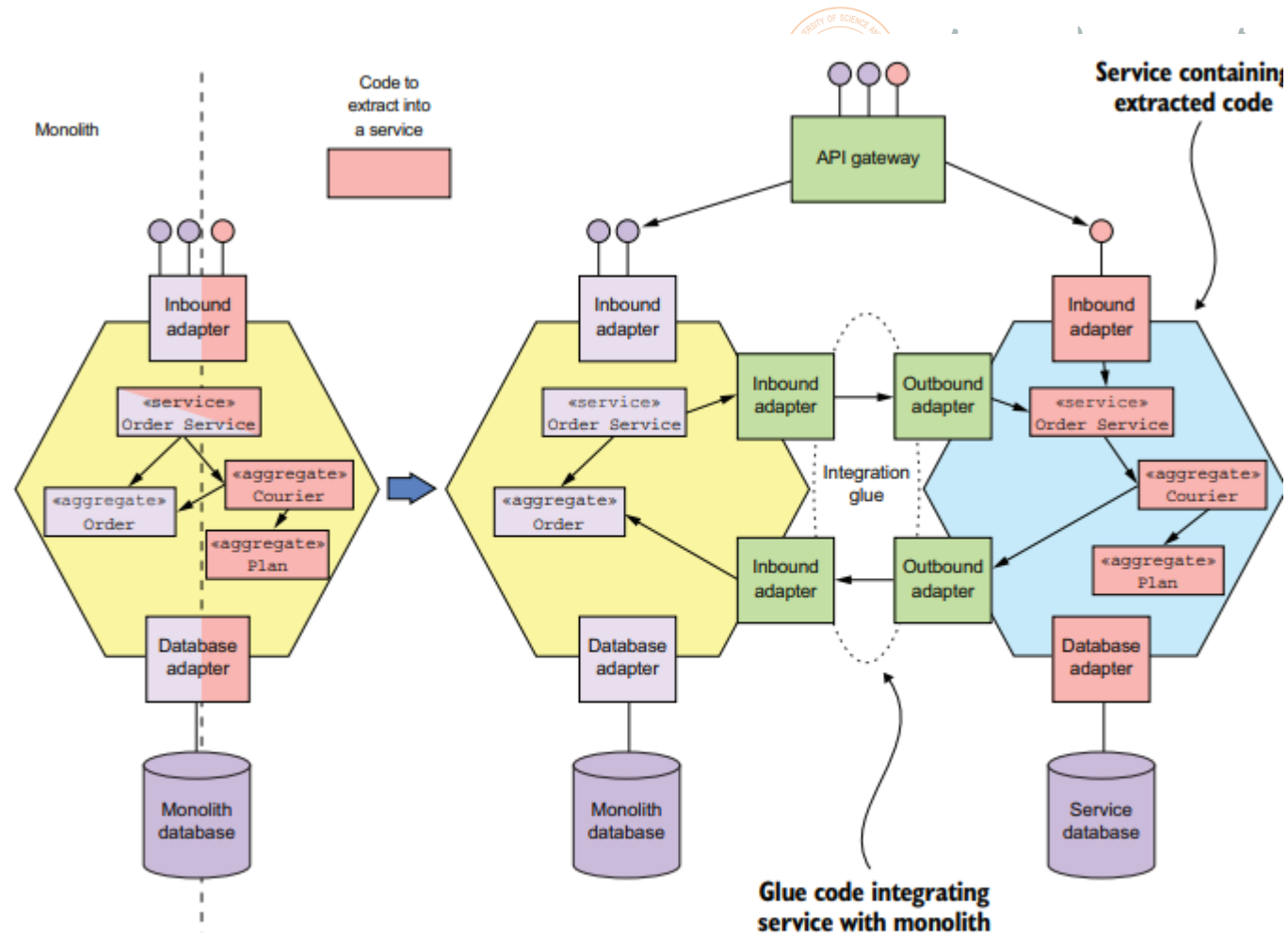


Figure 13.4 Break apart the monolith by extracting services. You identify a slice of functionality, which consists of business logic and adapters, to extract into a service. You move that code into the service. The newly extracted service and the monolith collaborate via the APIs provided by the integration glue.





# LECTURE 13

- Software maintenance (软件维护)
  - Maintainability
  - Technical debt
  - Refactoring
  - Re-architecting
- Software evolution (软件演化)
  - Legacy systems
  - Modernization
  - Deprecation

# THE EVOLUTION OF MICROSOFT WORD



1983-1987



1987-1991



1991-1993



1993-1995



1995-1999



1999-2003



2003-2007



2007-2010



2010-2013



2013-2019



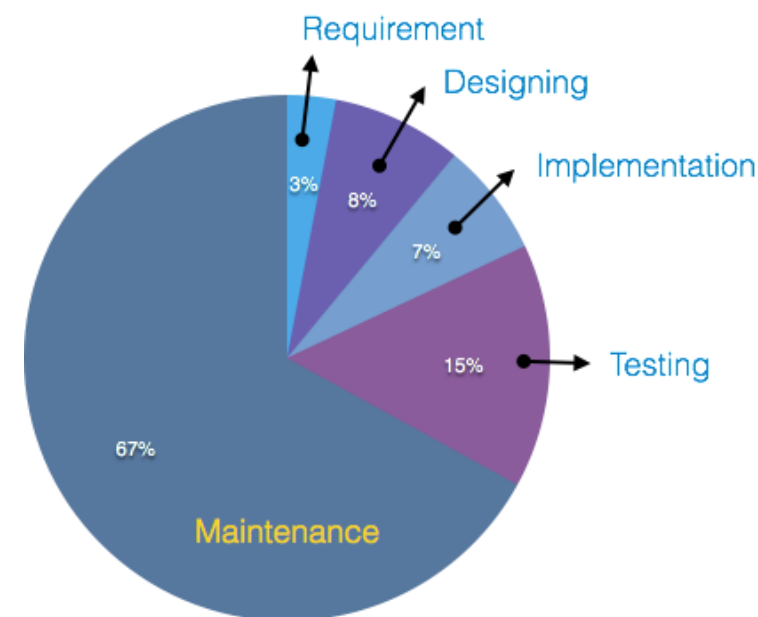
2019-present

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



# SOFTWARE EVOLUTION & MAINTENANCE COSTS

- 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/>

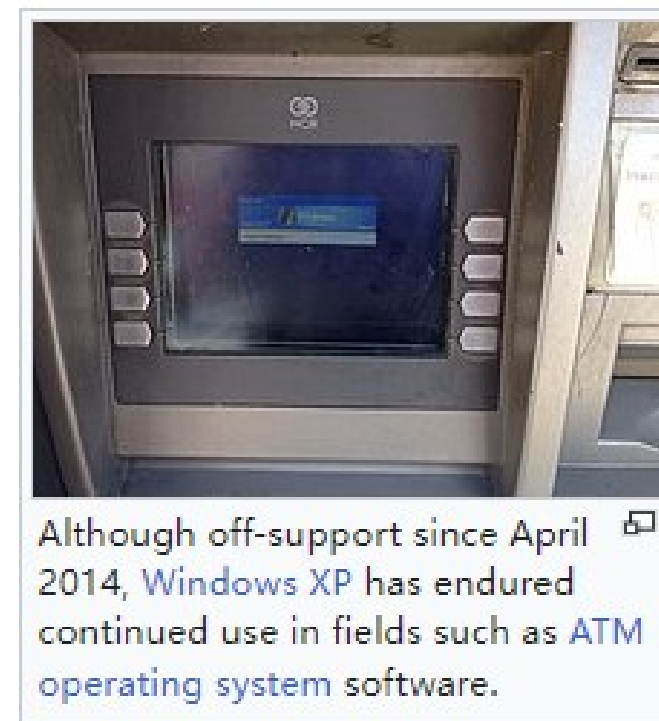


# WHY COSTLY?

- 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



[https://en.wikipedia.org/wiki/Legacy\\_system](https://en.wikipedia.org/wiki/Legacy_system)



# LEGACY SYSTEM

Denver	813	Southwest			
Denver	995	Southwest	7:50	18B	Canceled
El Paso	2411	Southwest	2:30	12B	Canceled
Honolulu	3845	Southwest	7:45	17B	Now 7:56 PM
Houston Hobby	2403	Southwest	1:10	18B	On Time
Houston Hobby	2457	Southwest	5:20	17B	Canceled
Kahului Maui	8829	Southwest	2:10	17B	Now 3:59 PM
Las Vegas	2403	Southwest	1:10	18B	On Time
Las Vegas	4558	Southwest	4:20	16	Canceled
Las Vegas	753	Southwest	6:40	16	Canceled
Little Rock	1278	Southwest	5:25	17A	Canceled
Miami	1482	Southwest	11:40	12B	Canceled
Nashville	1527	Southwest	11:20	12B	Canceled
Nashville	1447	Southwest	4:10	18A	Now 6:13 PM
Nashville	1278	Southwest	5:25	17A	Canceled
New Orleans	1972	Southwest	9:15	15	Now 11:04 AM
New Orleans					Now 2:33 PM



## Southwest Meltdown Shows Airlines Need Tighter Software Integration

The airline industry is long overdue for a tech overhaul that takes full advantage of the cloud and data integration, analysts say



# LEGACY SYSTEM

Southwest relies on crew-assignment software called SkySolver, an old application developed decades ago

"Southwest Airlines has imploded. Their antiquated software system has completely fried. Planes are parked. Crews are stranded in the airports with the passengers, volunteering to take the passengers in the parked planes but the software won't accept it."

# 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
- .....

OBSERVER

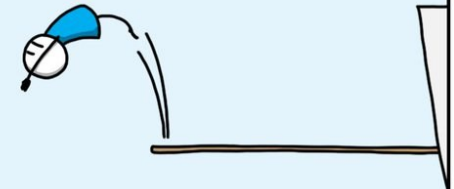
UNIVERSITY OF SCIENCE AND

MONKEYUSER.COM

WATCHING A SENIOR DEV



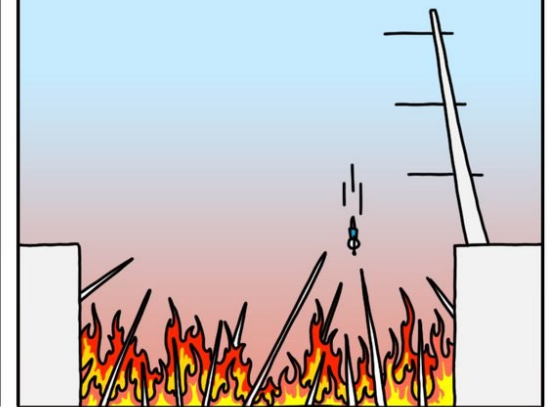
GRACEFULLY AND CONFIDENTLY



DIVE INTO



LEGACY CODE







# 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/decays** the system structure, making it increasingly difficult to understand or to extend
- Lack of tests; Integration with new systems is complex



# 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



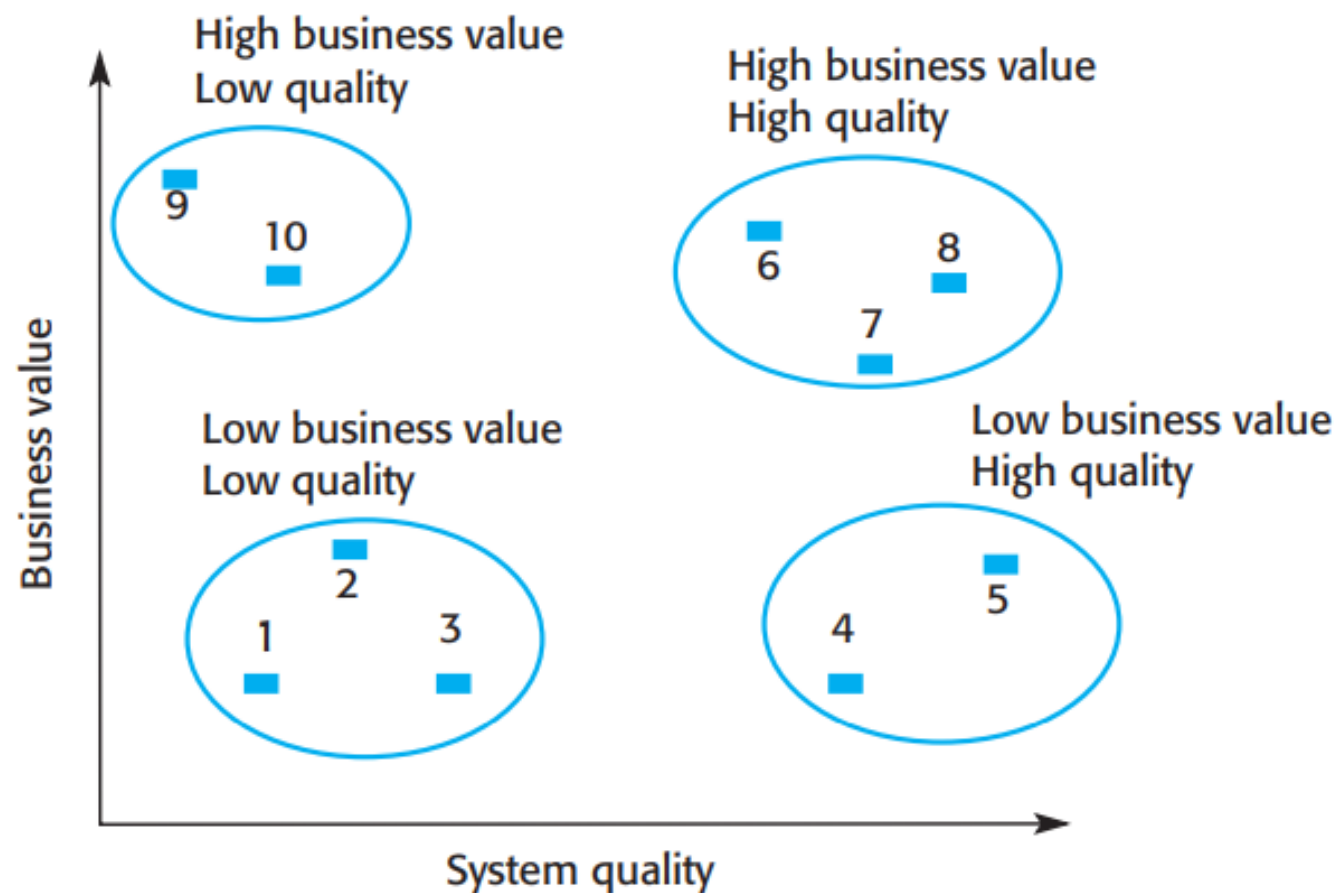
# DECISIONS FOR LEGACY SYSTEM

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:

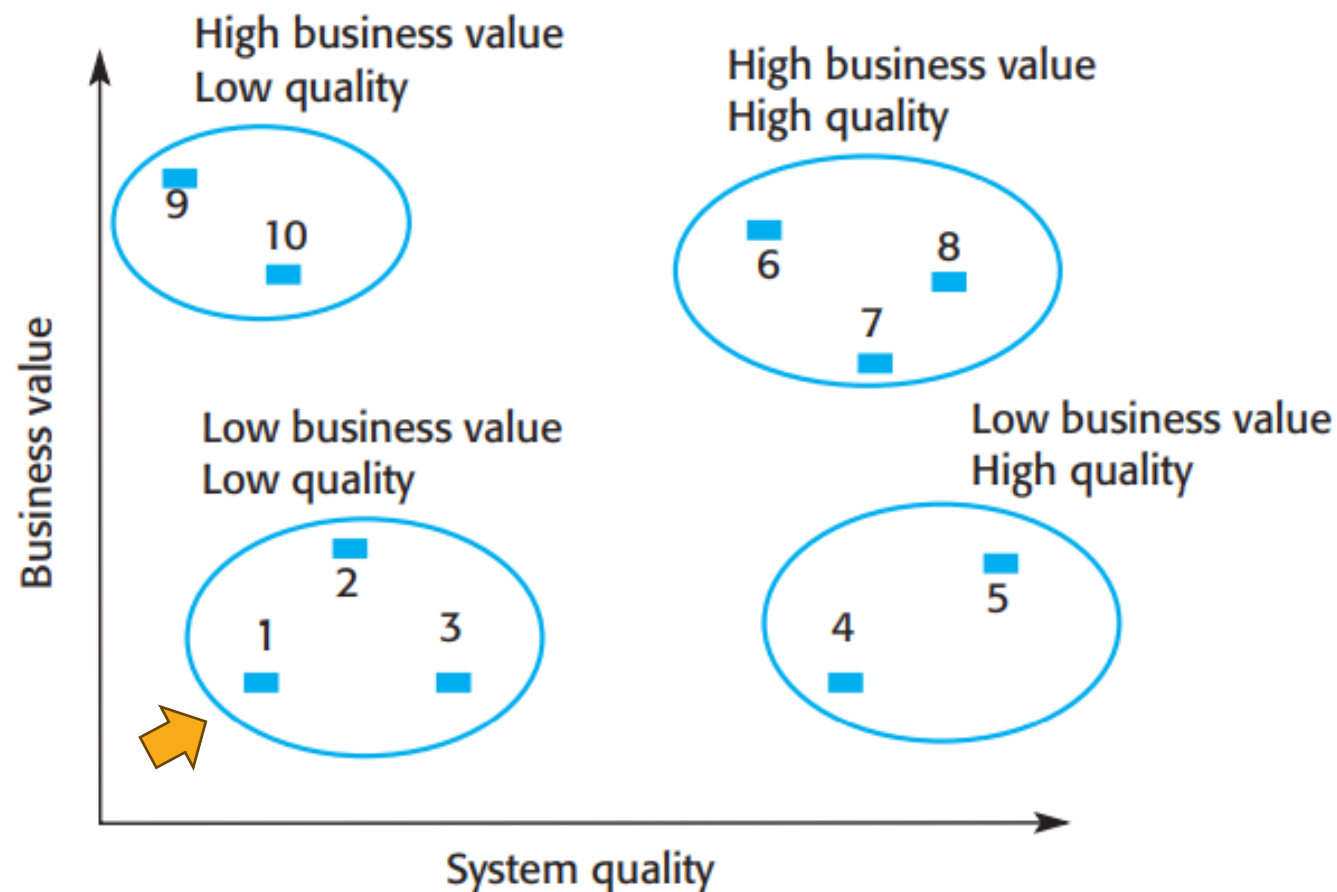
- System quality
- Business value



# WHICH DECISION TO MAKE?

## Low quality, low business value:

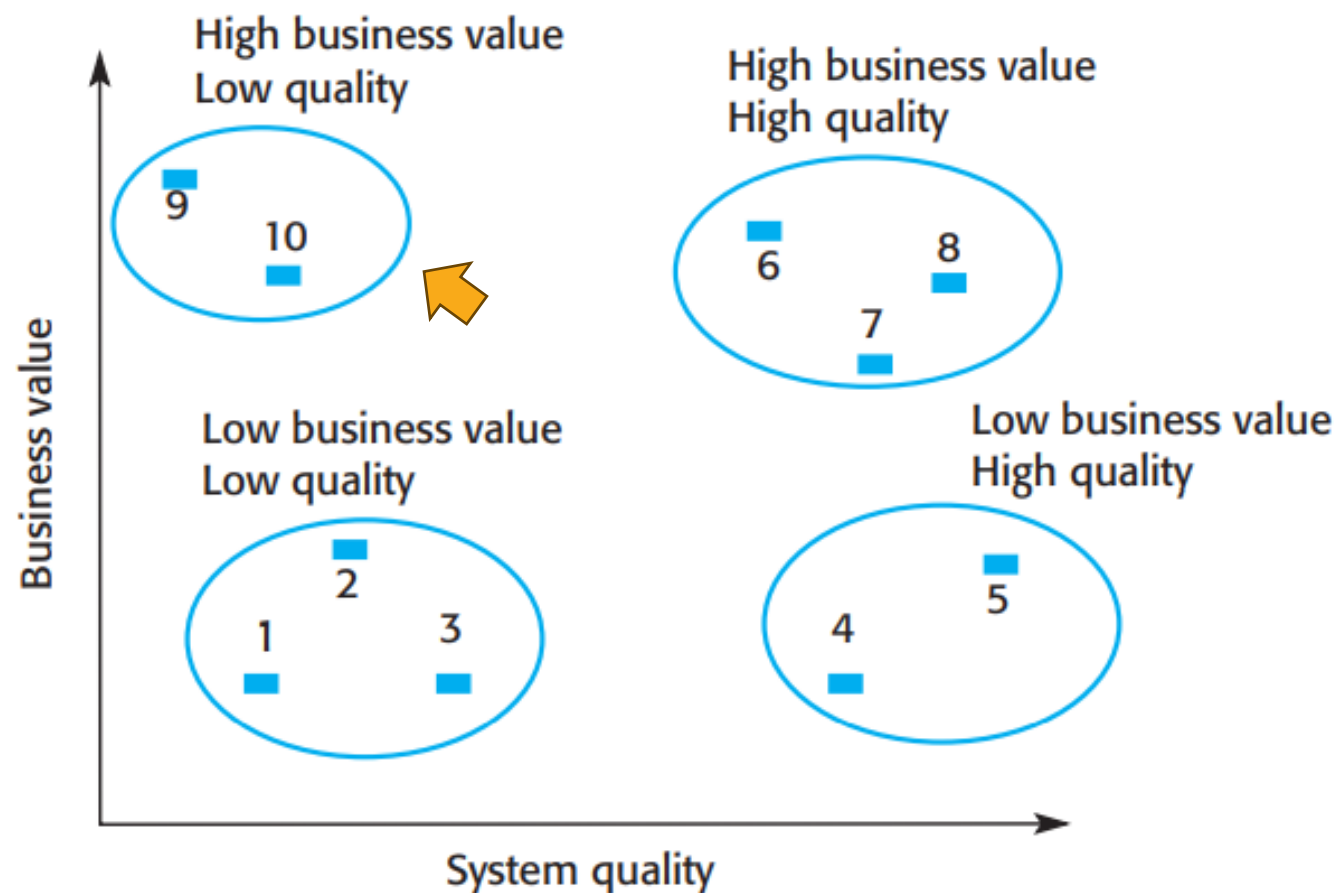
- Keeping these systems in operation will be expensive
- The rate of the return to the business will be fairly small.
- These systems should be **abandoned**



# WHICH DECISION TO MAKE?

## Low quality, high business value:

- These systems are making an important business contribution, so they cannot be abandoned.
- But 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.

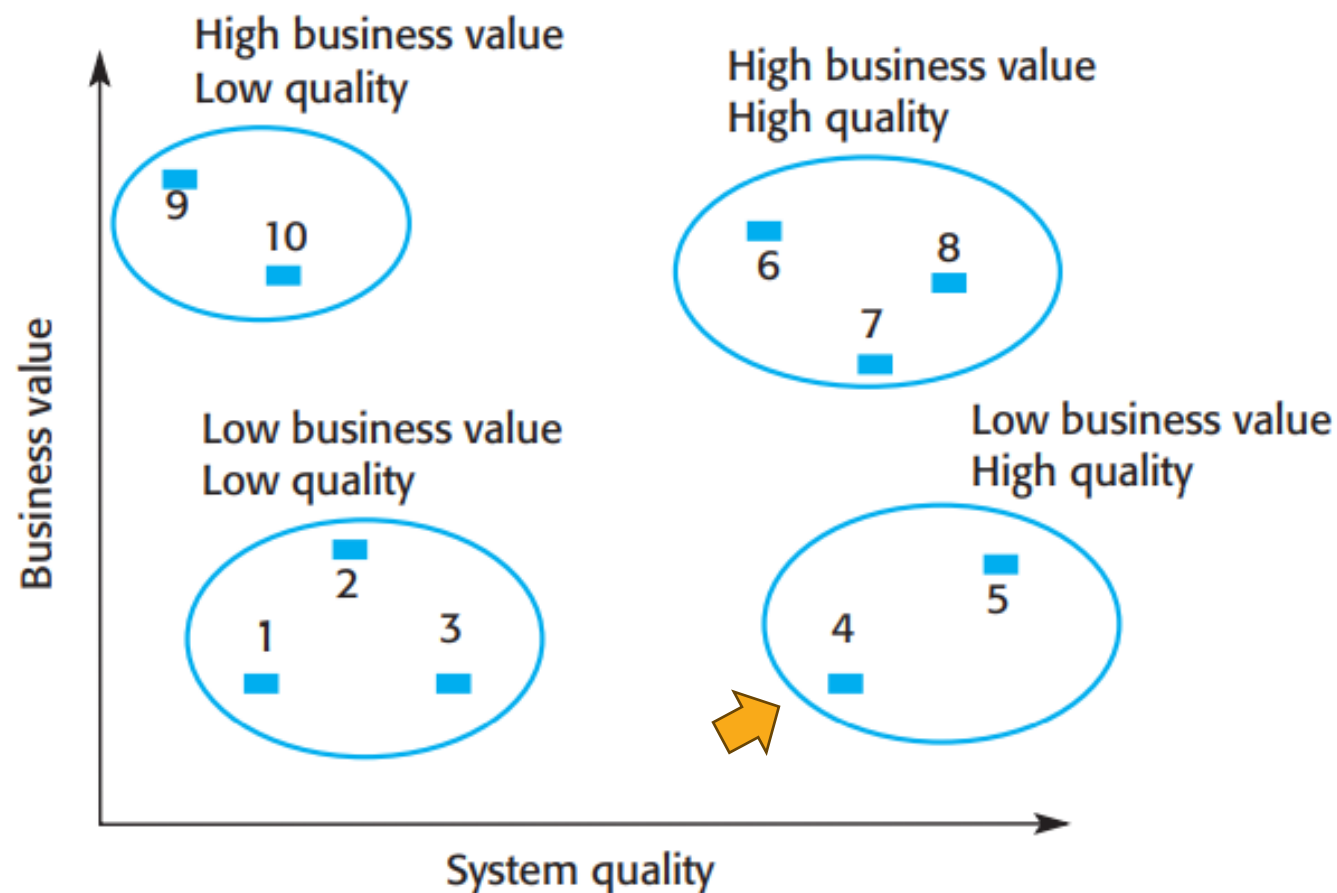




# WHICH DECISION TO MAKE?

## High quality, low business value:

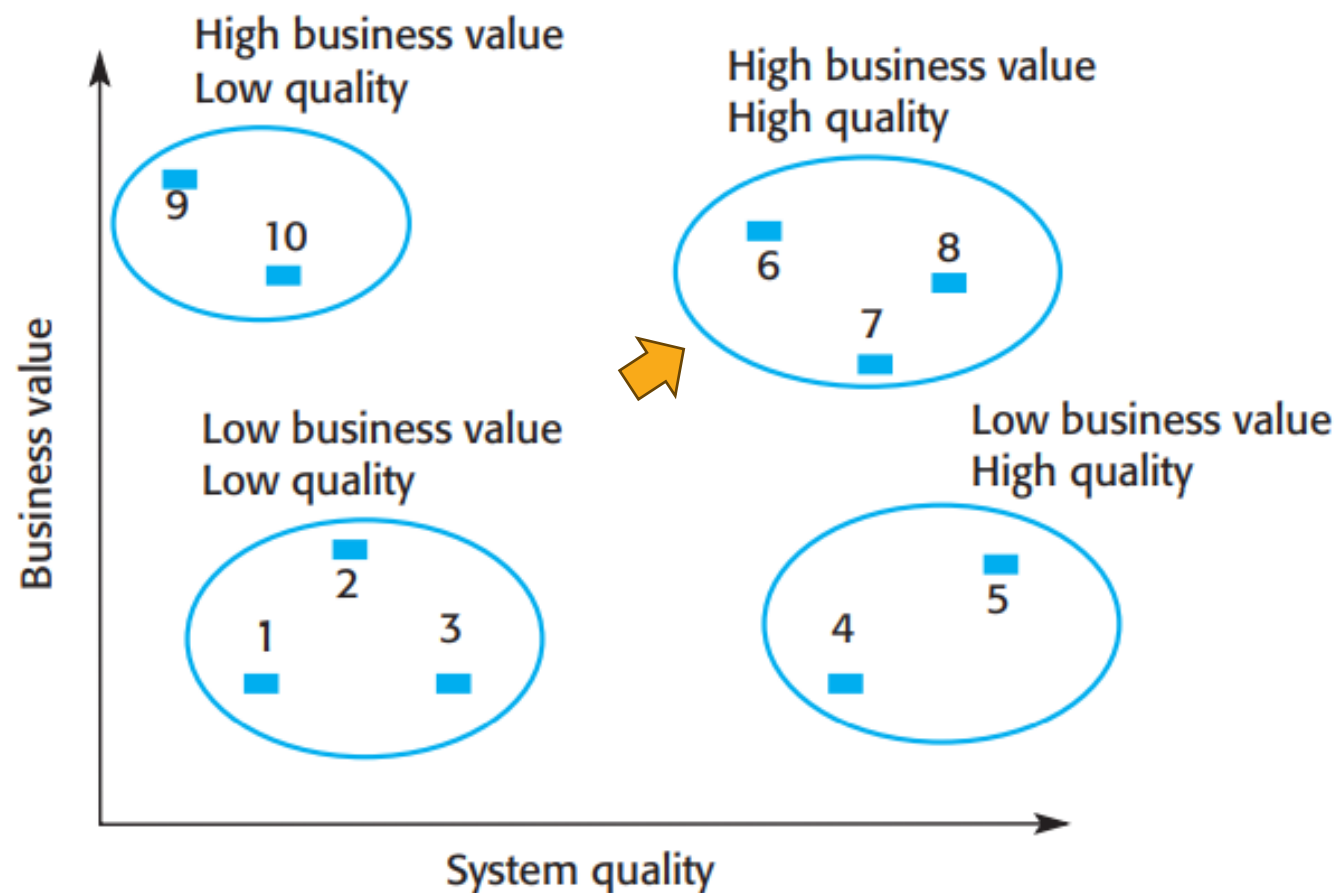
- These systems don't contribute much to the business but may not be very expensive to maintain.
- It's not worth replacing these systems, so **normal system maintenance** may be continued if expensive changes are not required and the system hardware remains in use.
- If expensive changes become necessary,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



# WHICH DECISION TO MAKE?

## High quality, high business value:

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

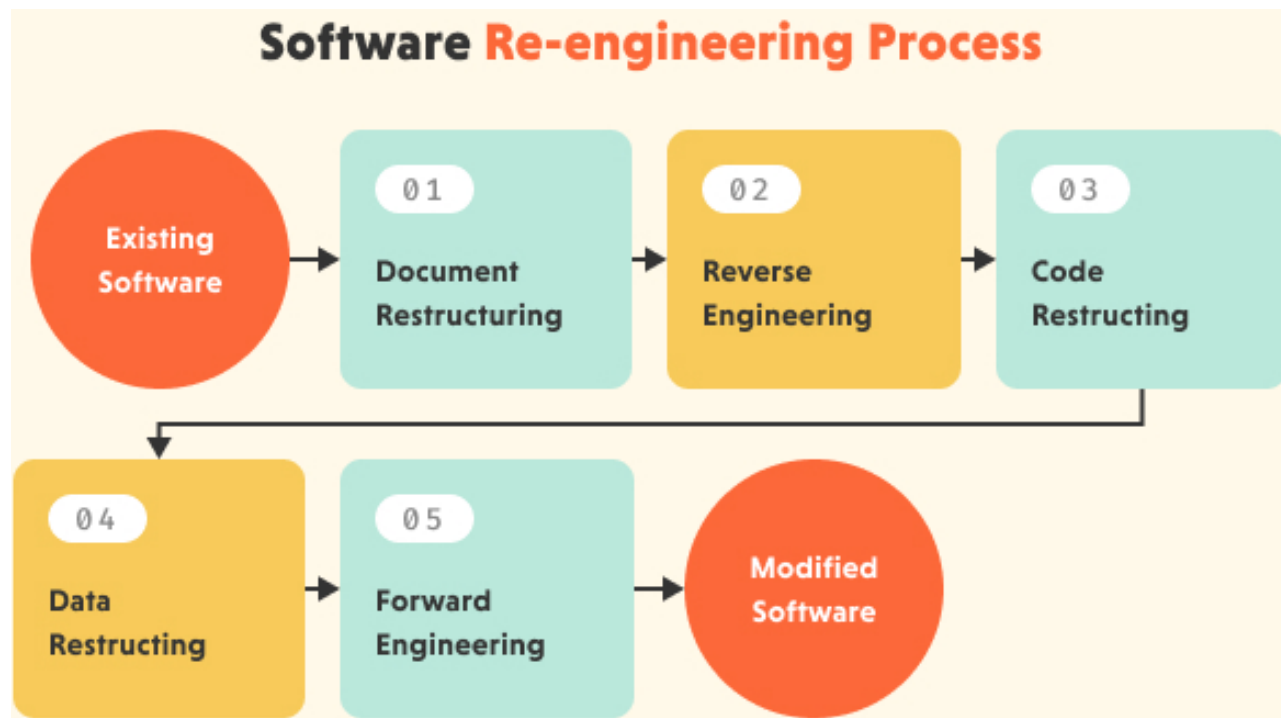




# SOFTWARE REENGINEERING

- To make legacy software systems easier to maintain, you can **reengineer** (再工程) 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



## Input:

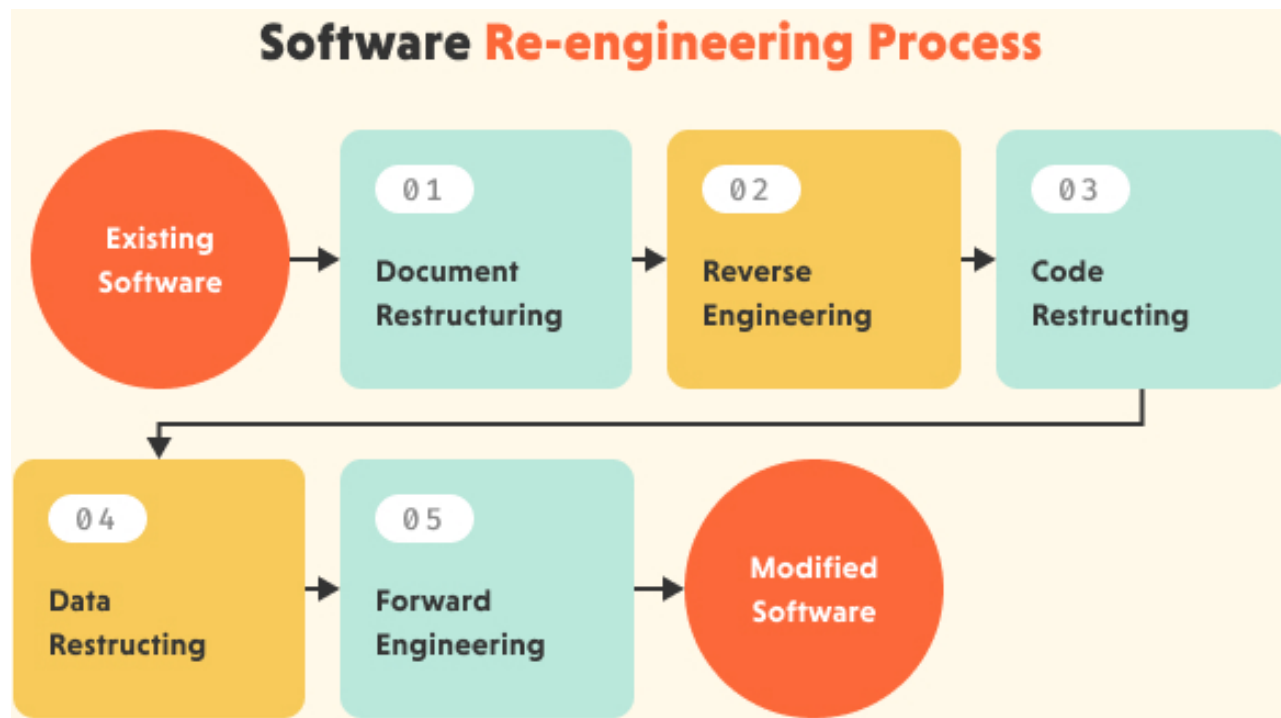
- The original legacy system

## Output:

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

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

# SOFTWARE REENGINEERING PROCESS

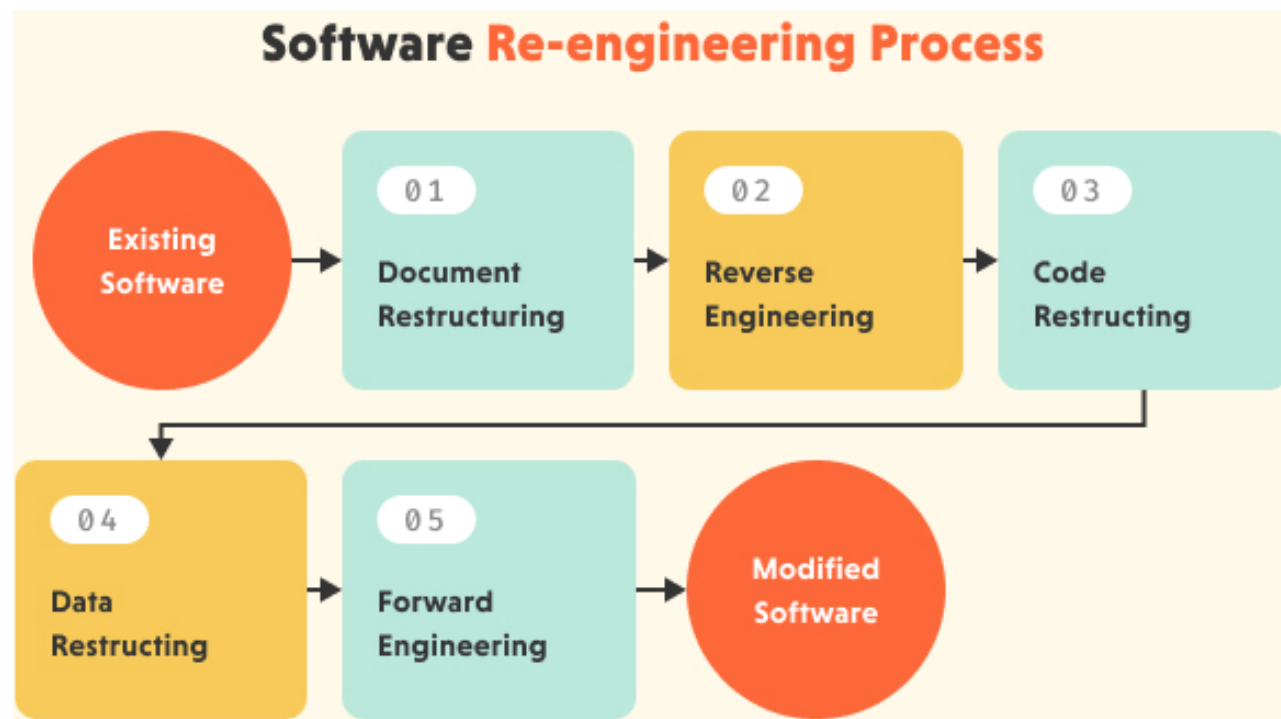


## 3 stages

- Reverse engineering
- System transformation
- Forward engineering

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

# SOFTWARE REENGINEERING PROCESS

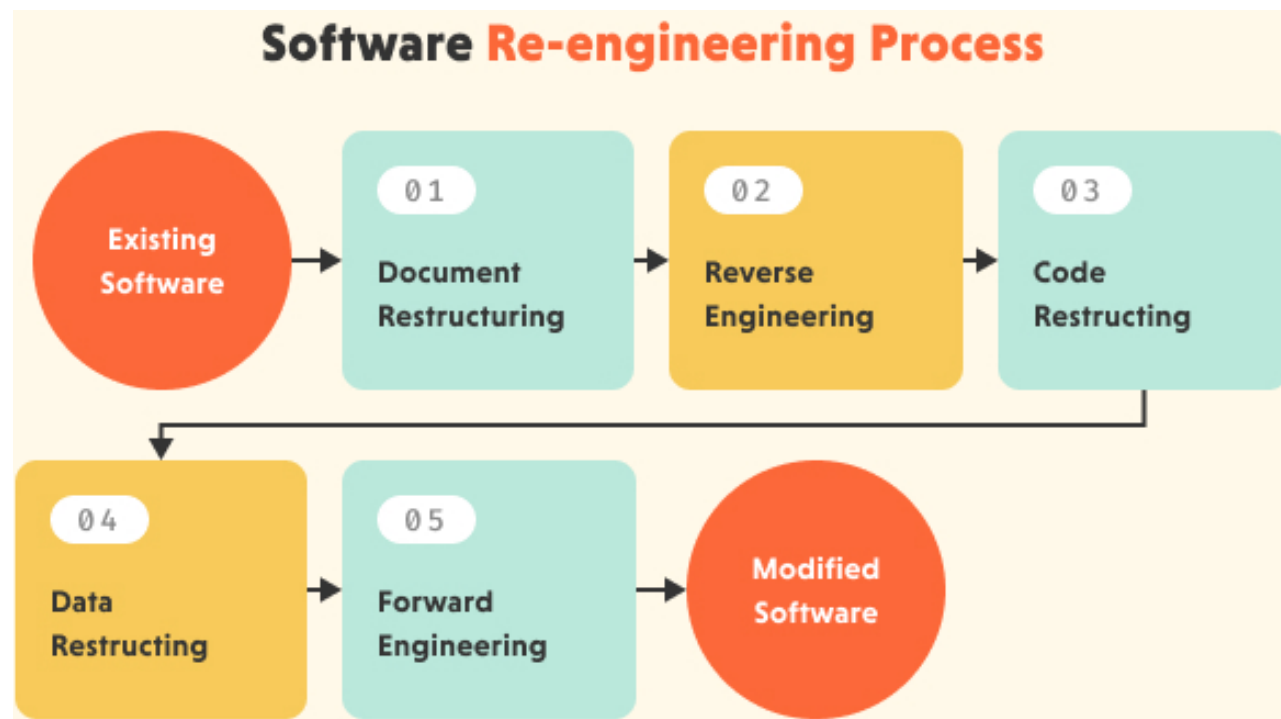


<https://modlogix.com/blog/legacy-software-re-engineering-risks-and-mitigations/>

## Reverse engineering (逆向工程):

- 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

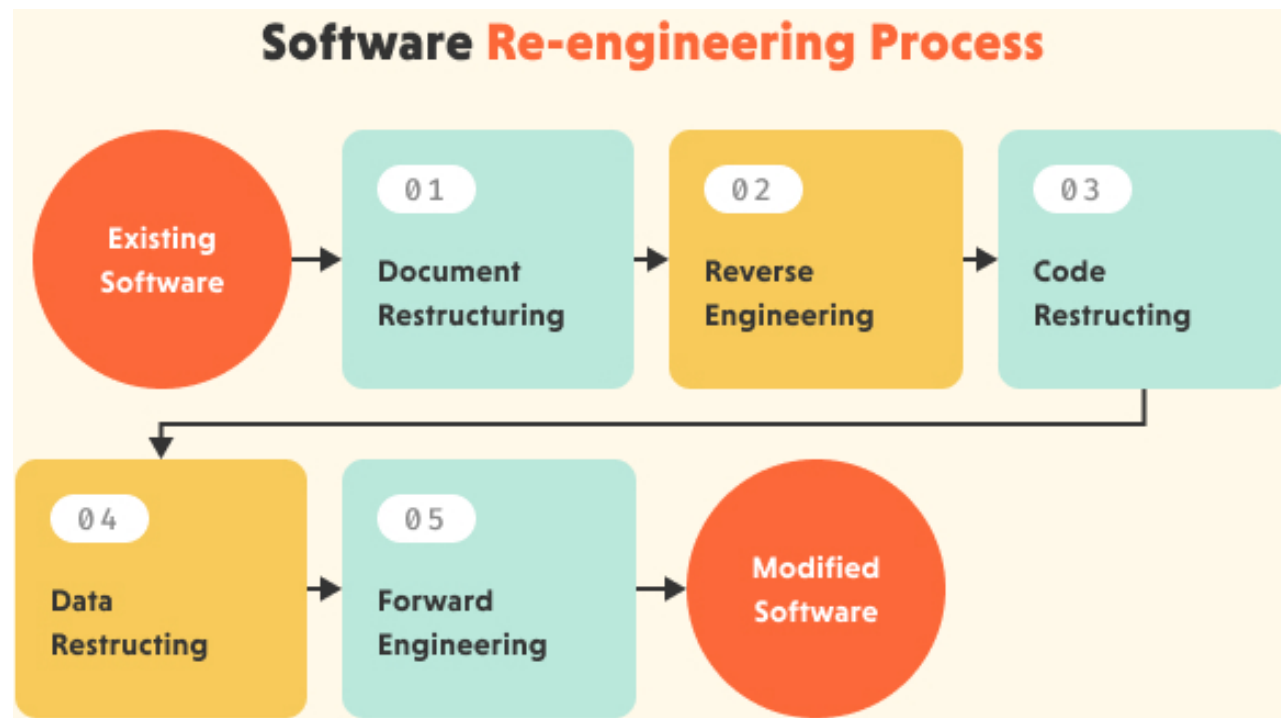


<https://modlogix.com/blog/legacy-software-re-engineering-risks-and-mitigations/>

## 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



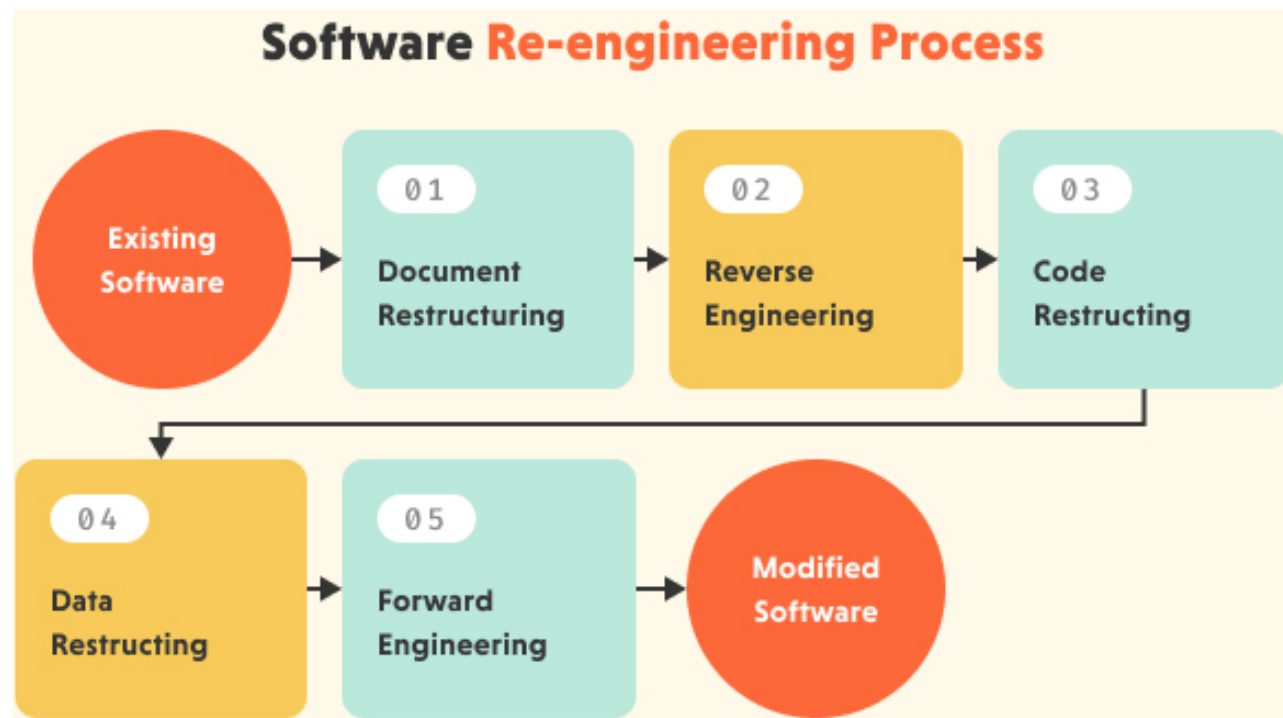
<https://modlogix.com/blog/legacy-software-re-engineering-risks-and-mitigations/>

## 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

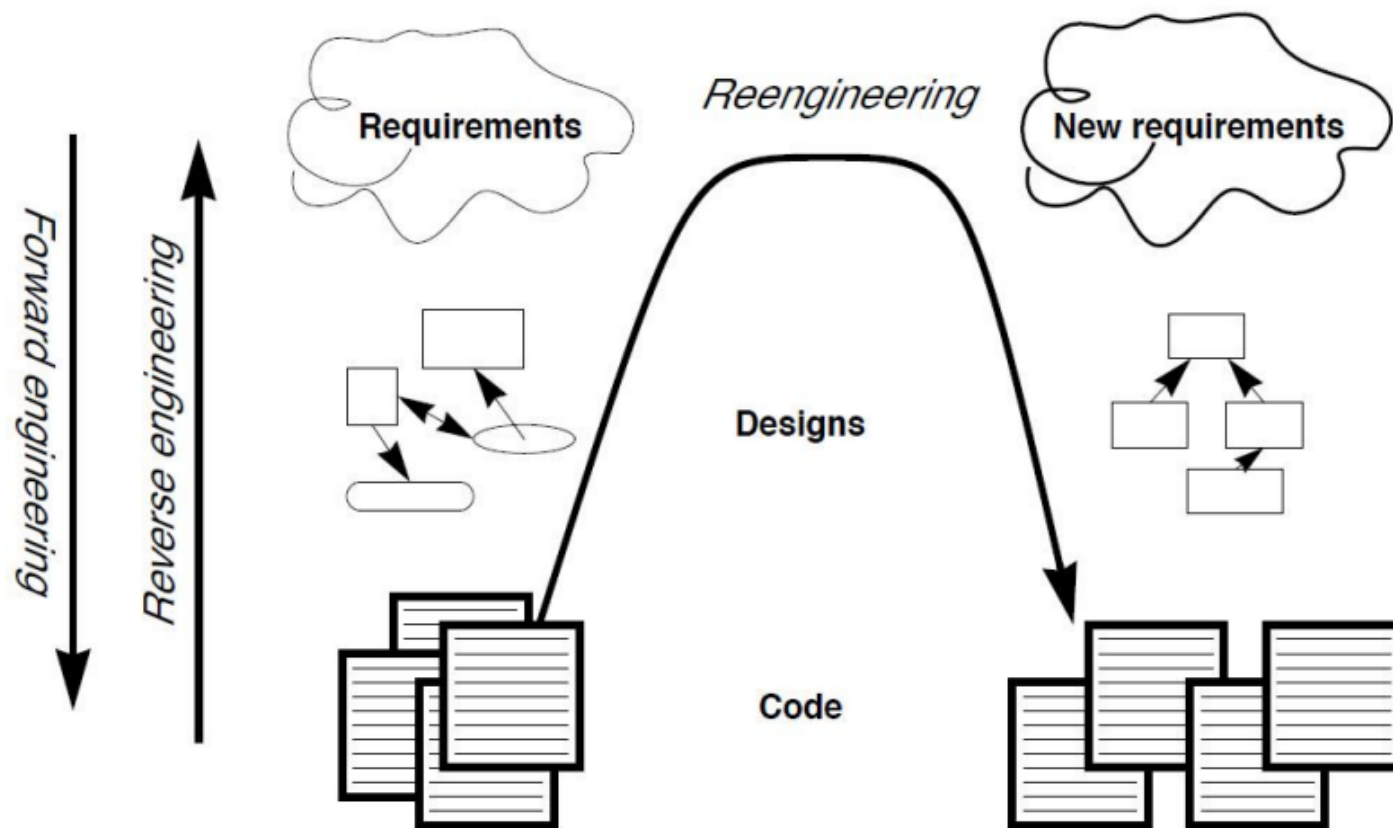


<https://modlogix.com/blog/legacy-software-re-engineering-risks-and-mitigations/>

## 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

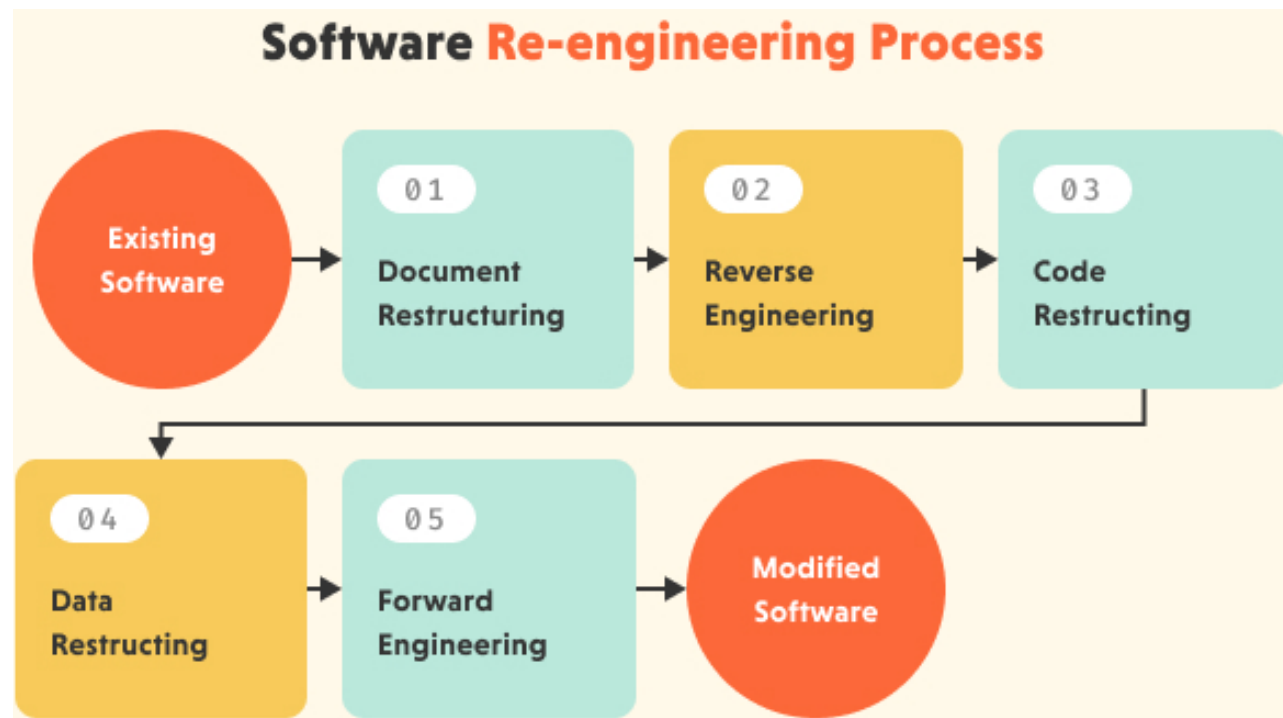
## Reverse engineering:

- Code -> design -> requirements

## Reengineering

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

# SOFTWARE REENGINEERING PROCESS

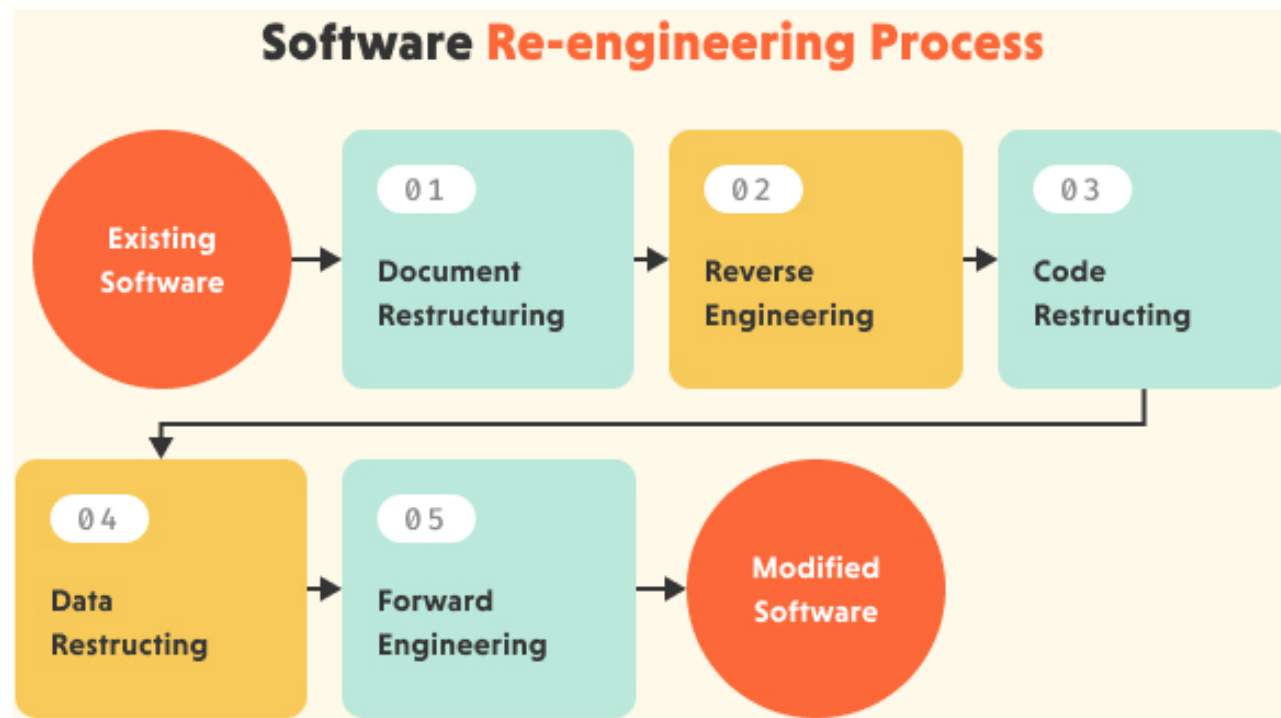


<https://modlogix.com/blog/legacy-software-re-engineering-risks-and-mitigations/>

## 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



# DEPRECATION

- Deprecation (弃用): the formal 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**



The **L<sup>A</sup>T<sub>E</sub>X** Project

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



# WHAT SHOULD BE DEPRECATED?

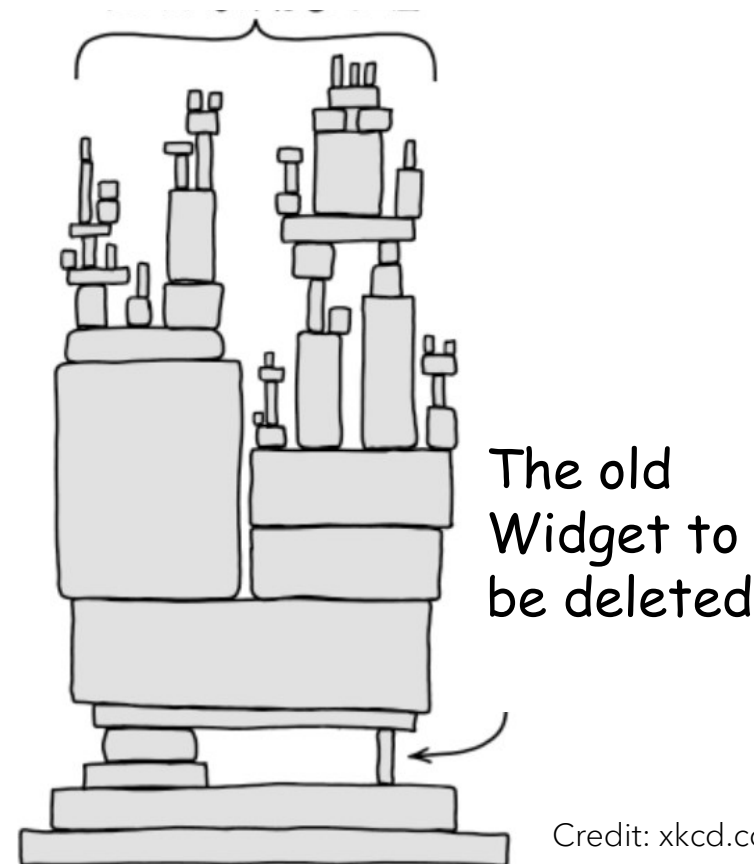
- Deprecation is best suited for systems/modules/code/APIs/features that are demonstrably **obsolete** and **a replacement exists** that provides comparable functionality.
- The new replacement might use resources **more efficiently**, have **better** security properties, be built in **a more sustainable fashion**, or just fix bugs.

# CASE STUDY: DEPRECATATION

- A new Widget has been developed.
- The decision is made that everyone should use the new one and stop using the old one.

Possible deprecation strategies?

All the nice projects that are built

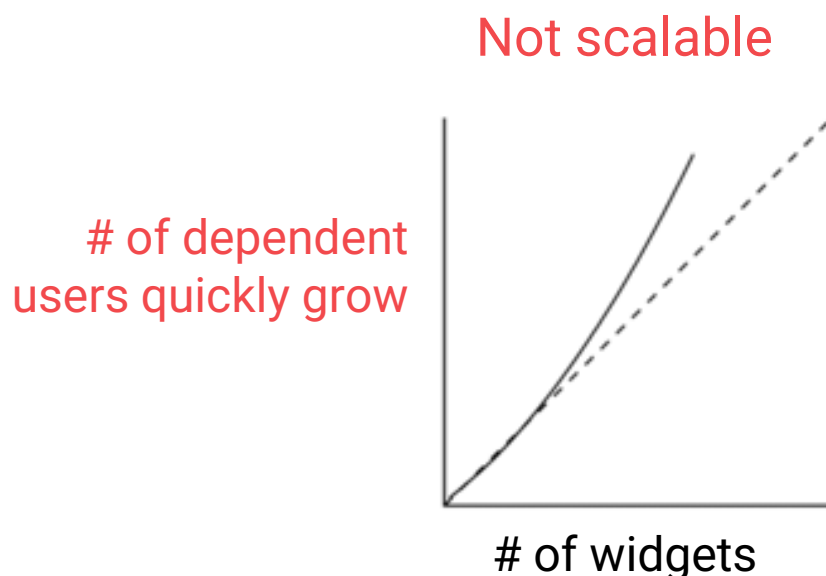






# CASE STUDY: DEPRECIATION

- Strategy 1: the project lead says “We’ll delete the old Widget on August 15th; everyone make sure you’ve converted to the new Widget.”



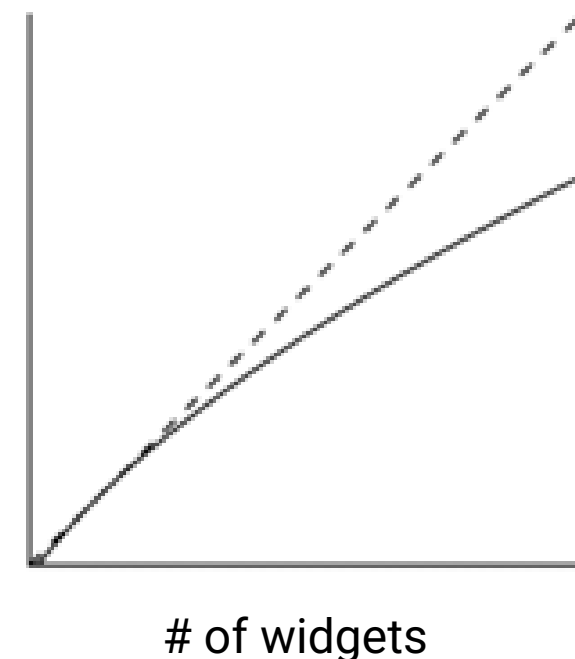
This strategy might work in a small software setting, but quickly fails as **both the depth and breadth of the dependency graph increases**.

Teams depend on an ever-increasing number of Widgets, and a single build break can affect a growing percentage of the company.

# CASE STUDY: DEPRECATION

- Strategy 2: A dedicated group of experts in the infrastructure team
  - Either do the update in place, in backward-compatible fashion.
  - Or do the work to migrate their users to new versions
- Having a dedicated group of experts execute the change **scales better** than asking for more maintenance effort from every user

A small group of experts: scales better





# 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

```
Date date = new Date( year: 2024, Calendar.JANUARY, date: 30);
```

'Date(int, int, int)' is deprecated

```
@Deprecated
@Contract(pure = true)
public Date(
    int year,
    @MagicConstant(intValues = {Calendar.JANUARY
    int date
})
```



# HOW TO DEPRECATE (ELEGANTLY)?

## End of support for Windows 10, Windows 8.1, and Windows 7

This page has the information, tools, and tips you need to easily transition from Windows 10, Windows 8.1, and Windows 7, to Windows 11.

Windows 10 Windows 8.1 Windows 7

### Support for Windows 10 will end in October 2025

After October 14, 2025, Microsoft will no longer provide free software updates from Windows Update, technical assistance, or security fixes for Windows 10. Your PC will still work, but we recommend moving to Windows 11. Windows 11 offers a modern and efficient experience designed to meet current demands for heightened security.

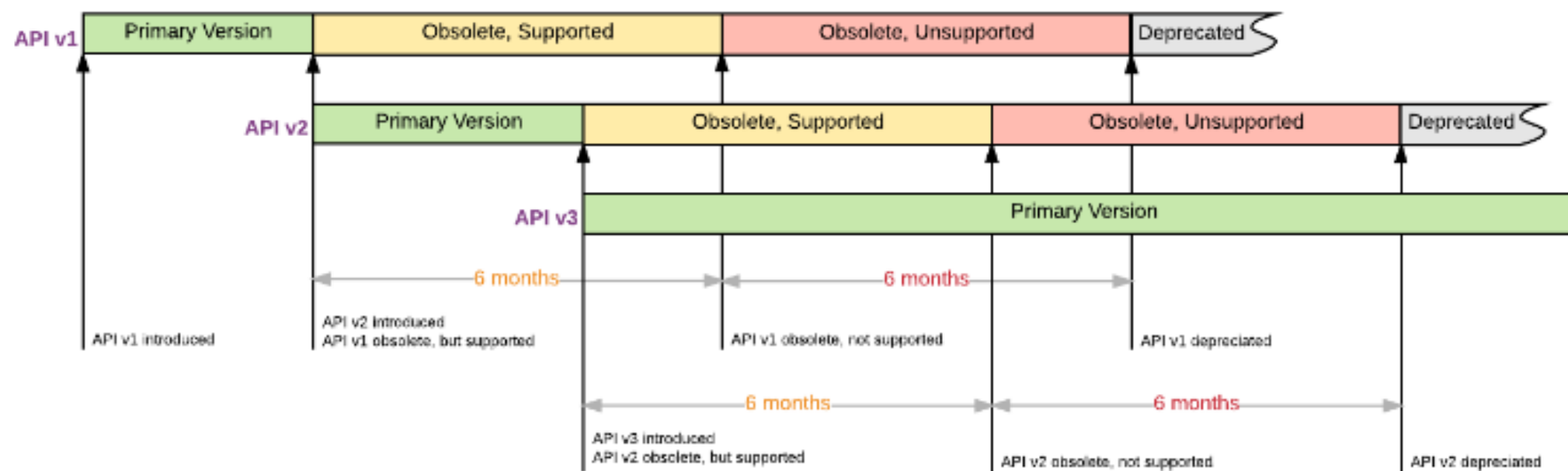


## Sunset period (sunsetting)

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

# HOW TO DEPRECATE (ELEGANTLY)?

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.



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



# HOW TO DEPRECATE (ELEGANTLY)?

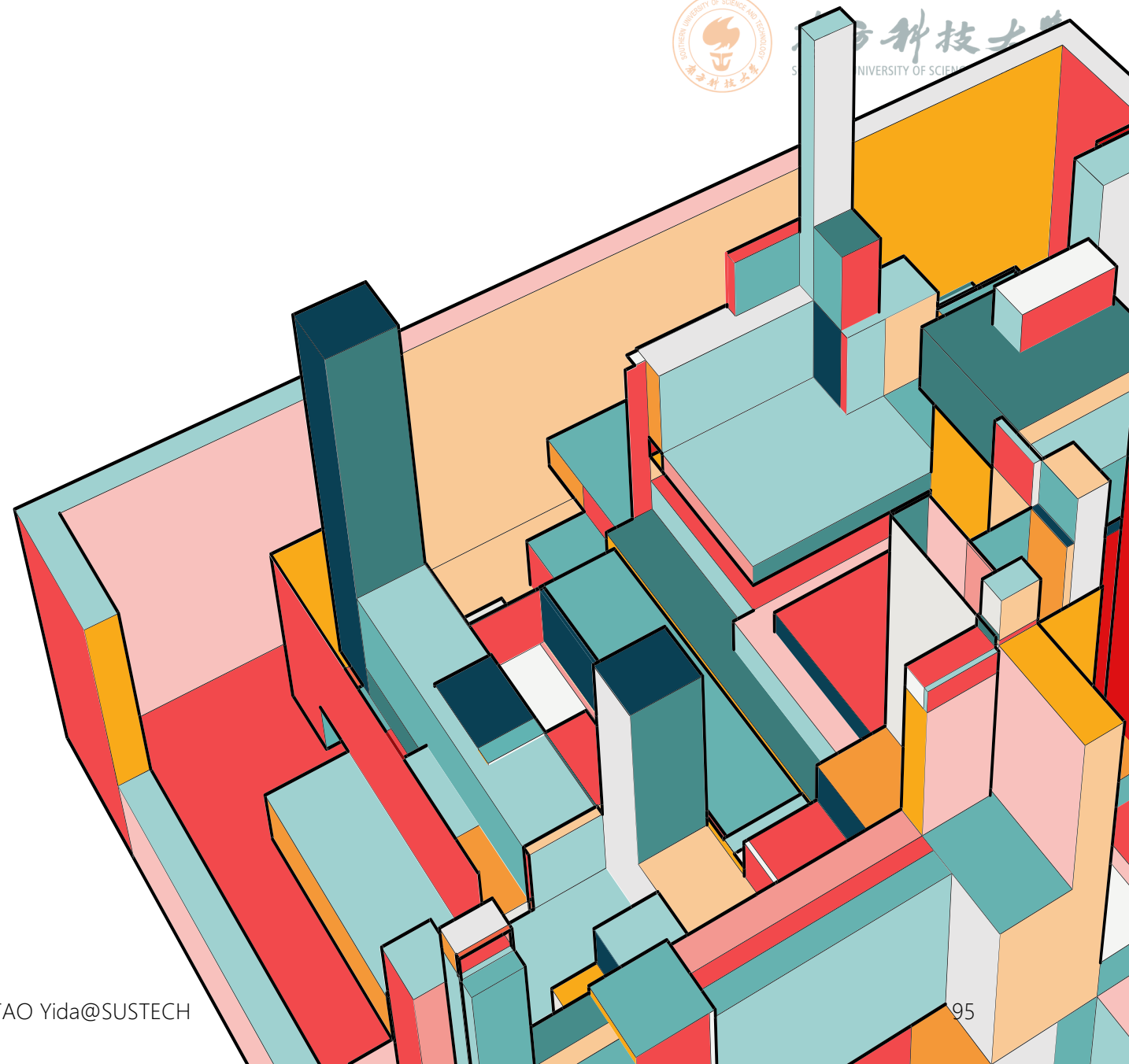
## Migration & Testing

- 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



# READINGS

- Chapter 9. Software Evolution. Software Engineering by Ian Sommerville, 10<sup>th</sup> edition
- Chapter 15. Deprecation. Software Engineering at Google by Titus Winters et al.
- Chris Richardson. Microservices Patterns







# NEXT

- Course review