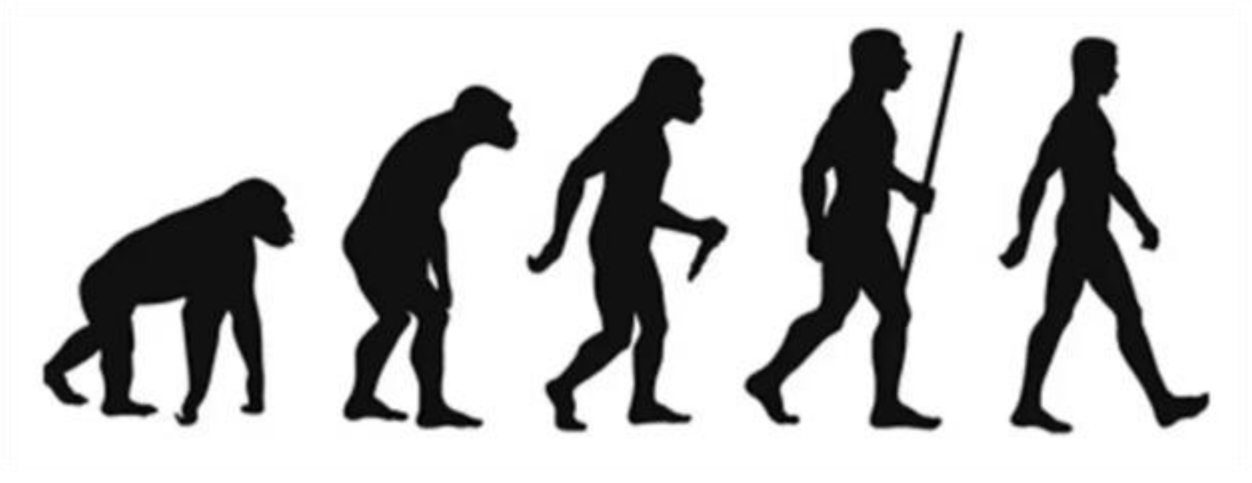
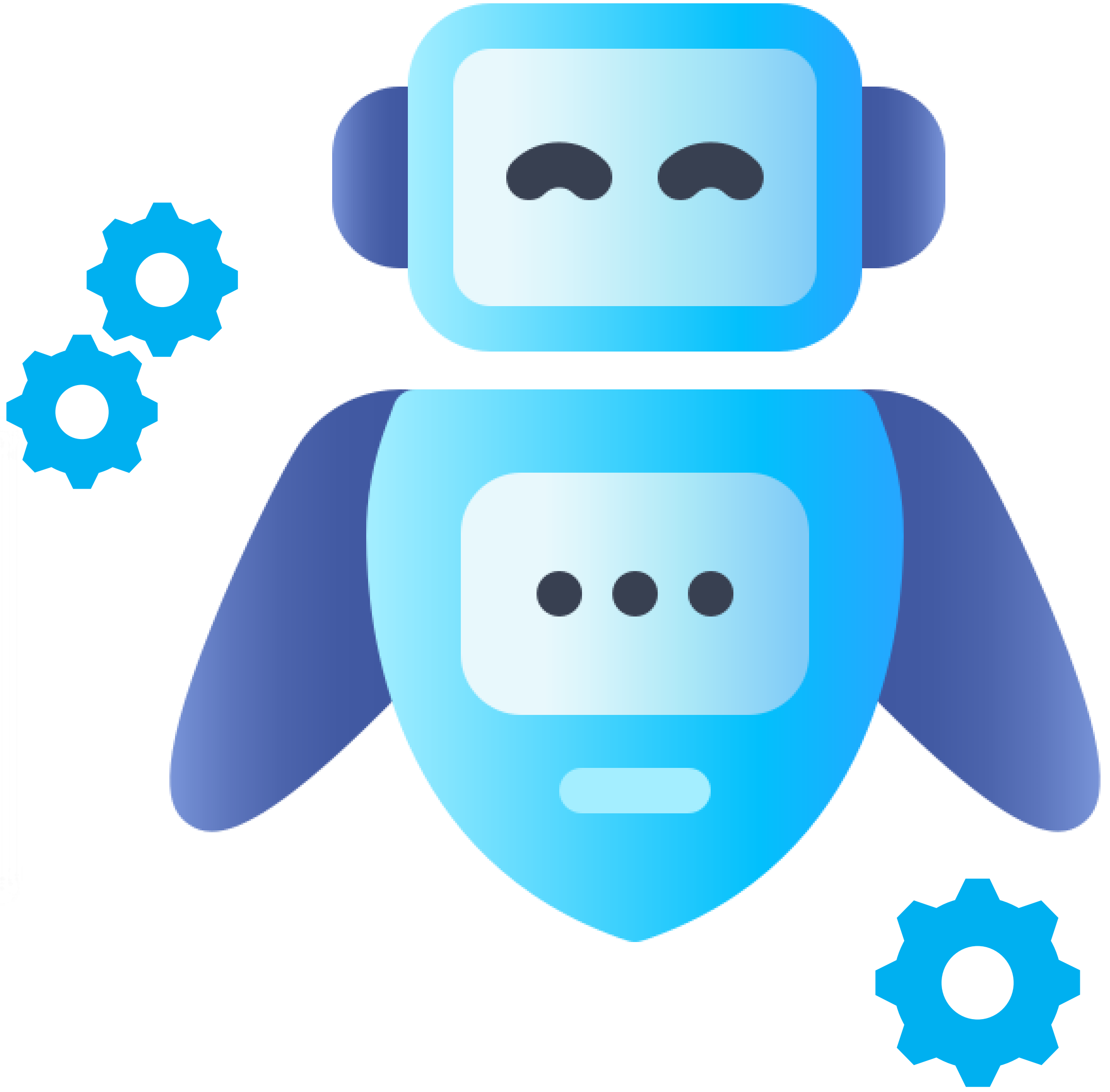
Your Professional Mission in Software Engineering :

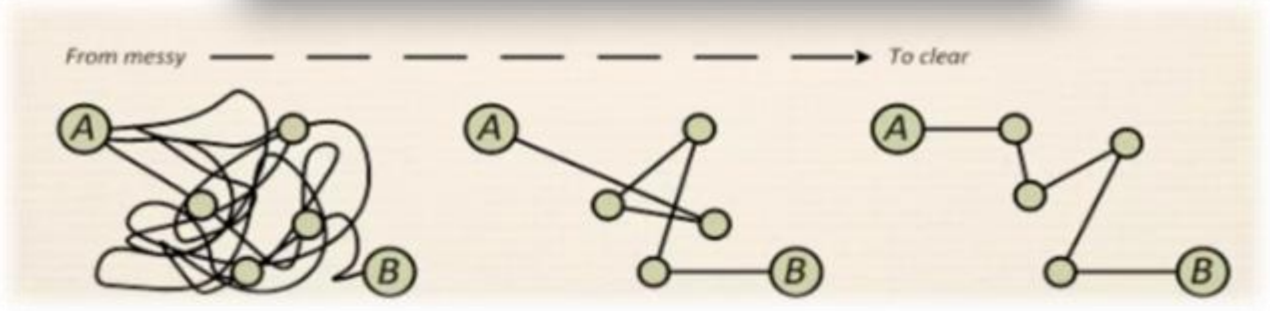
Code Refactoring





By Sheyma Naghshbandi

What is Refactoring ?



Improves the internal structure of the software but preserves the external behaviour.

Why You Should Refactor?

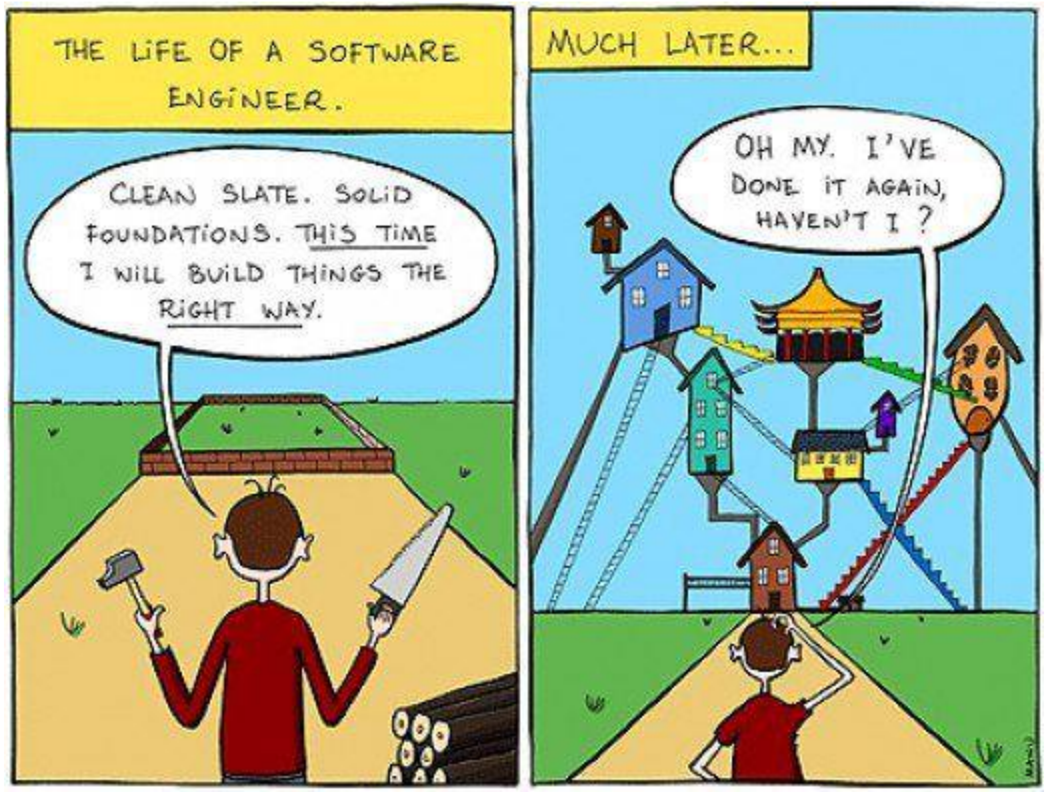
- Readability
- Maintainability
- Teamwork
- Reducing Technical Debt

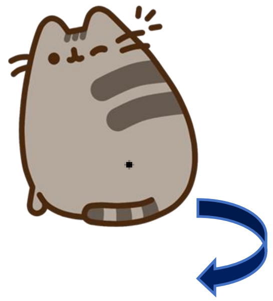
- Scalability for the Future
- Reduce Complexity
- Better Performance

Workplace Example: At companies like Google, engineers must deliver clean code because every sloppy line hikes up maintenance costs and frustrates clients!

When Should You Refactor?

- After Quick-and-Dirty Code
- Before Adding Features
- When Bugs Take Over
- In Team Projects
- When Complexity Goes Wild


THE LIFE OF A SOFTWARE ENGINEER.
CLEAN SLATE. SOLID FOUNDATIONS. THIS TIME I WILL BUILD THINGS THE RIGHT WAY.
MUCH LATER...
OH MY. I'VE DONE IT AGAIN, HAVEN'T I ?



Refactoring also fits naturally in the **Agile methods** philosophy
Is needed to address the principle "Maintain simplicity"
Wherever possible, actively work to eliminate complexity from the system
By refactoring the code.

Workplace Example: In a startup, if you don't refactor before launching a product, clients will face bugs, and your project could flop!

What is Dirty Code?

Why it Happens?

Hard to Read

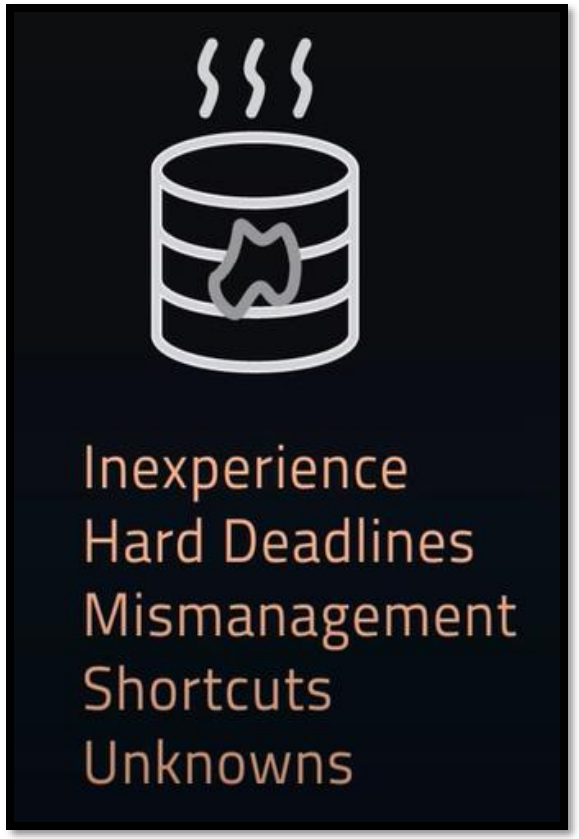
Overly Complex

Redundant

Tightly Coupled

Unmaintainable

Untested or Poorly Tested



Inexperience
Hard Deadlines
Mismanagement
Shortcuts
Unknowns

Example : At InnoTech, Sara's team struggled with a data analytics platform where dirty code caused slow data processing, buggy dashboard filters, and low morale, costing the company time and customer trust.

What Are Code Smells?

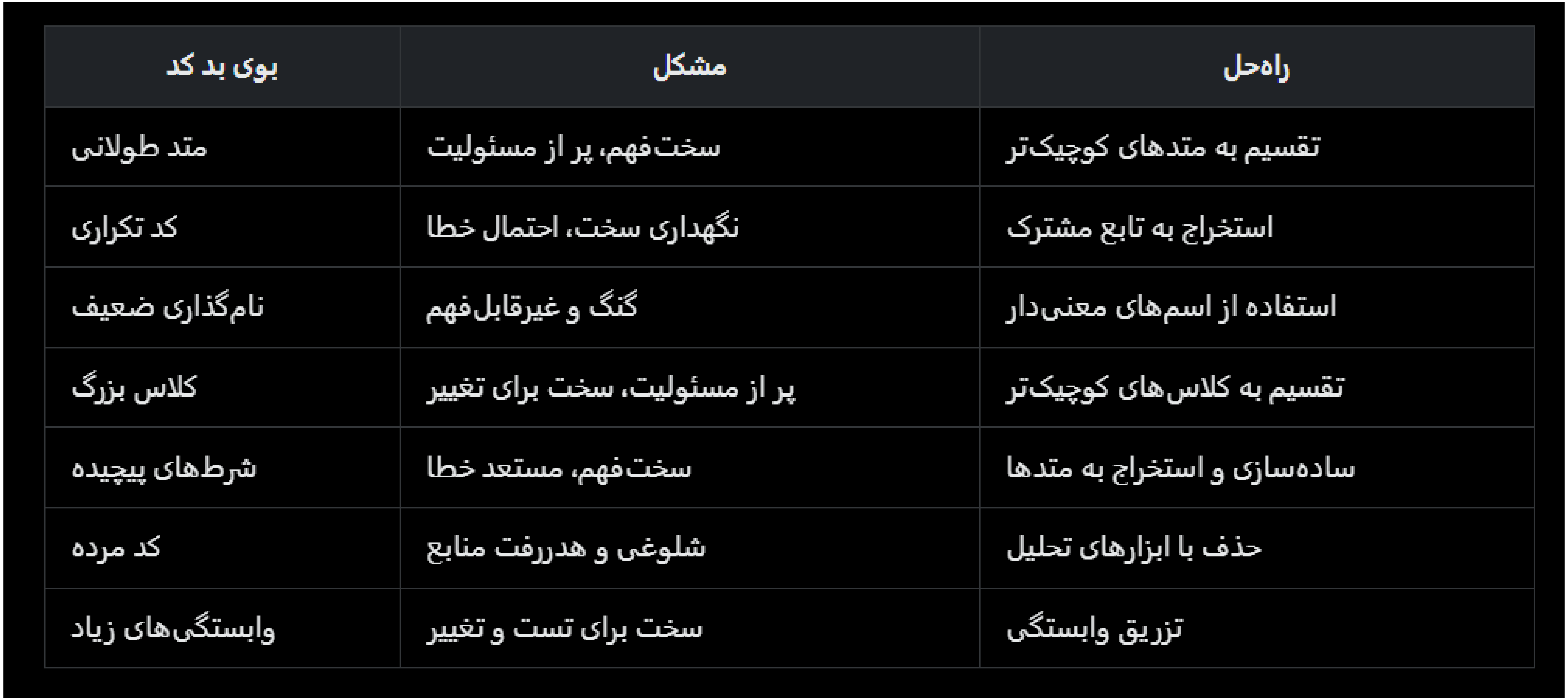
Code smells are specific patterns or symptoms in code that indicate potential design or quality issues. They're not bugs (the code may still work), but they suggest underlying problems that could lead to dirty code, making the codebase harder to maintain, extend, or test. Code smells are like warning signs—red flags that prompt developers to investigate and refactor before issues escalate.

What Should You Do to Avoid Code Smells? Refactor Your Code !

- **Duplicated Code**
- **Larg classes**
- **Long Parameter List**
- **Long Methods/ Functions**
- **Tight Coupling**
- **Dead Code**

Ew! Your Code Smells!

How to get rid of Code Smells



بوی بد کد	مشکل	راه‌حل
متد طولانی	سخت‌فهم، پر از مسئولیت	تقسیم به متدهای کوچیک‌تر
کد تکراری	نگهداری سخت، احتمال خطا	استخراج به تابع مشترک
نام‌گذاری ضعیف	گنگ و غیرقابل‌فهم	استفاده از اسم‌های معنی‌دار
کلاس بزرگ	پر از مسئولیت، سخت برای تغییر	تقسیم به کلاس‌های کوچیک‌تر
شرط‌های پیچیده	سخت‌فهم، مستعد خطا	ساده‌سازی و استخراج به متدها
کد مرده	شلوغی و هدررفت منابع	حذف با ابزارهای تحلیل
وابستگی‌های زیاد	سخت برای تست و تغییر	تزریق وابستگی

Refactoring Techniques

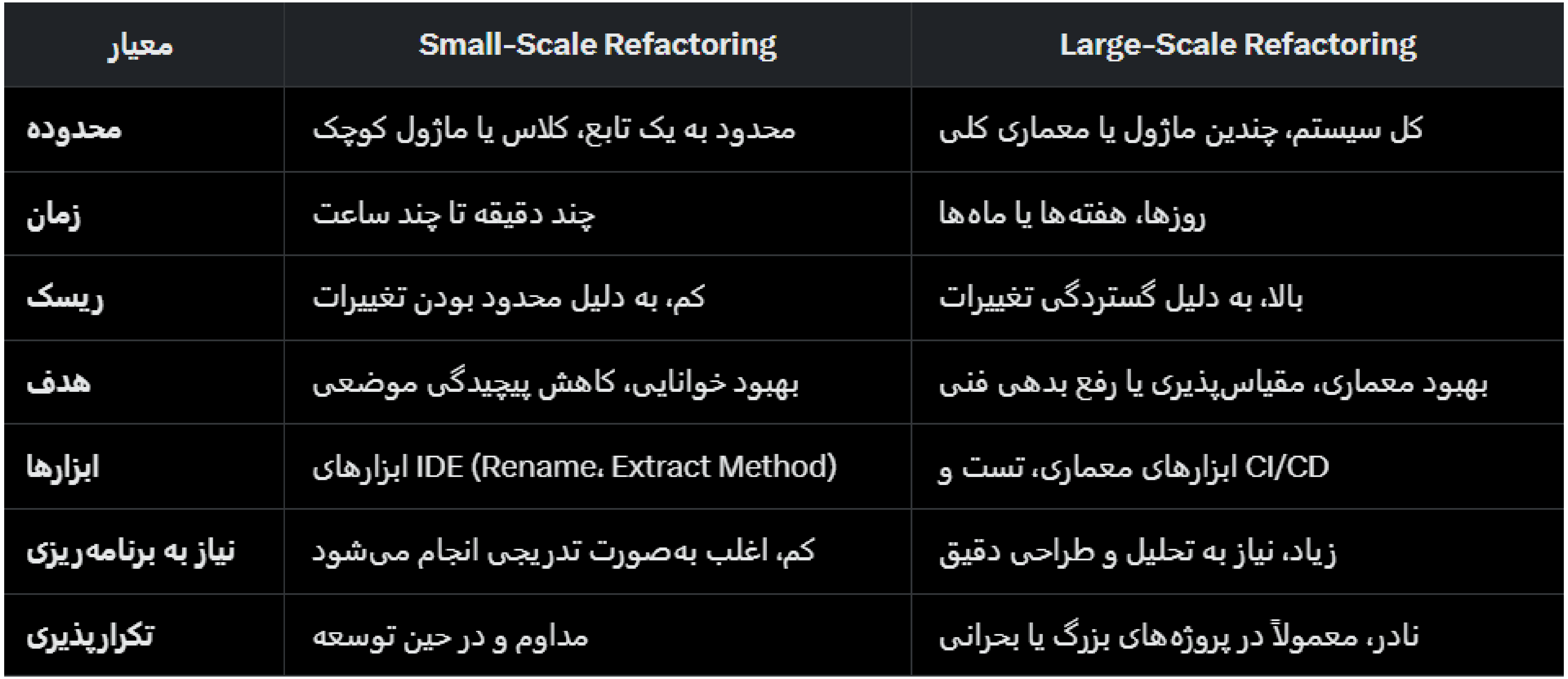
Small Refactoring

- Extract Method and Extract Class
- Rename Method/Variable
- Red-Green-Refactor
- Introduce Parameter Object
- Remove Dead code Inline Method
- User Face Refactoring

Big Refactoring

Large-Scale Refactoring involves major changes to a project's code structure or system architecture, spanning multiple modules or the entire software. It addresses deep structural issues, enhances scalability, or prepares for new requirements.

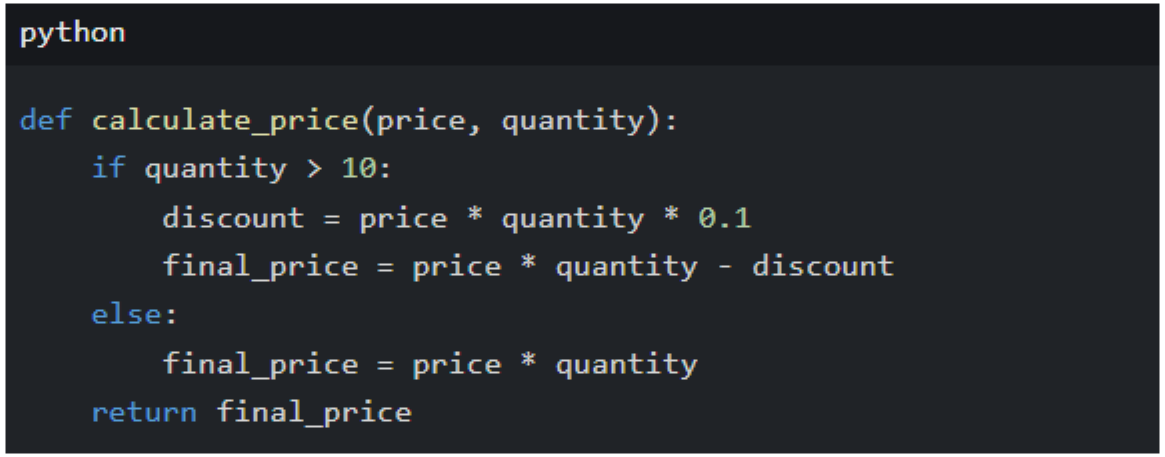
Small-Scale vs Large-Scale Refactoring



معیار	Small-Scale Refactoring	Large-Scale Refactoring
محدوده	محدود به یک تابع، کلاس یا ماژول کوچک	کل سیستم، چندین ماژول یا معماری کلی
زمان	چند دقیقه تا چند ساعت	روزها، هفته‌ها یا ماه‌ها
ریسک	کم، به دلیل محدود بودن تغییرات	بالا، به دلیل گستردگی تغییرات
هدف	بهبود خوانایی، کاهش پیچیدگی موضعی	بهبود معماری، مقیاس‌پذیری یا رفع بدهی فنی
ابزارها	ابزارهای IDE (Rename، Extract Method)	ابزارهای معماری، تست و CI/CD
نیاز به برنامه‌ریزی	کم، اغلب به‌صورت تدریجی انجام می‌شود	زیاد، نیاز به تحلیل و طراحی دقیق
تکرارپذیری	مداوم و در حین توسعه	نادر، معمولاً در پروژه‌های بزرگ یا بحرانی

Practical Example:

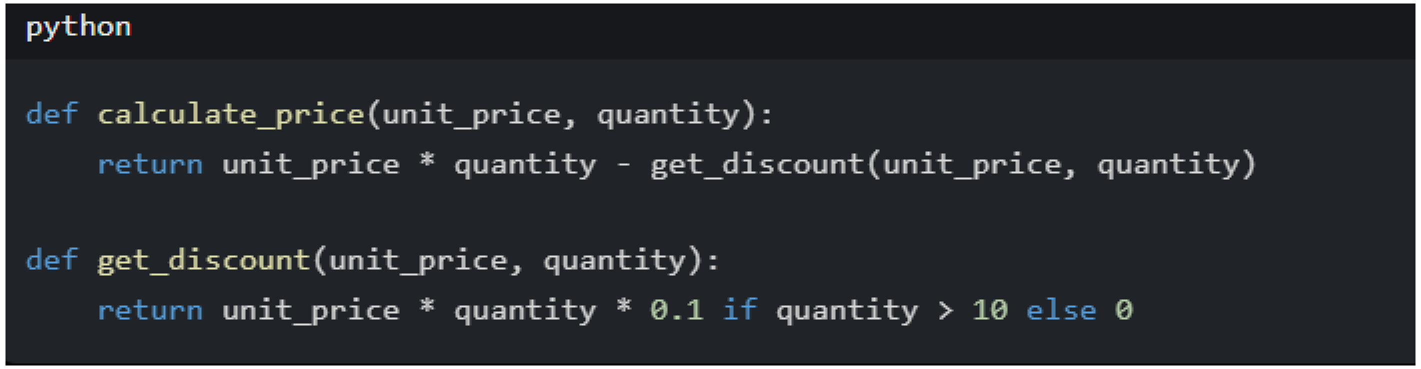
فرض کنید کدی به شکل زیر دارید که یک تابع
برای محاسبه تخفیف یک محصول است



```python
def calculate_price(price, quantity):
    if quantity > 10:
        discount = price * quantity * 0.1
        final_price = price * quantity - discount
    else:
        final_price = price * quantity
    return final_price
```

Rename : تغییر نام متغیرها برای خوانایی بهتر

Extract Method : جدا کردن منطق محاسبه تخفیف به یک متد جدید



```python
def calculate_price(unit_price, quantity):
    return unit_price * quantity - get_discount(unit_price, quantity)

def get_discount(unit_price, quantity):
    return unit_price * quantity * 0.1 if quantity > 10 else 0
```

Small Refactoring

کد خواناتر شده است

منطق تخفیف به‌صورت جداگانه قابل تست و استفاده مجدد است

تغییر کوچکی است که در چند دقیقه انجام شده و ریسک کمی دارد

Practical Example:

```
#include
int main() {
  printf("Employee: Ali - Active\n");
  printf("Employee: Reza - Active\n");
  return 0;
}
```

```
#include
void log_employee(char *name) {
  printf("Employee: %s - Active\n", name);
}
int main() {
  log_employee("Ali");
  log_employee("Reza");
  return 0;
}
```

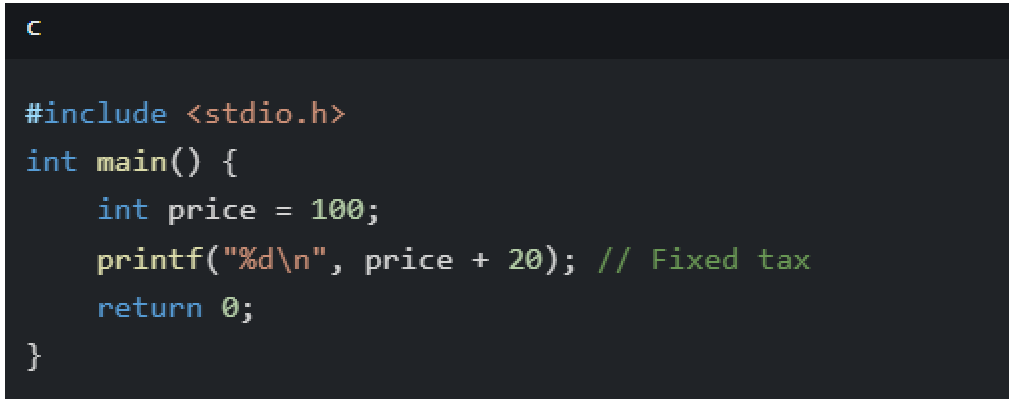
مشکل:
دو خط
printf
داریم که کار
یکسانی می‌کنن
فقط اسم‌ها فرق
داره

- کد تکراری داریم
- اگه بخوای فرمت پیام رو تغییر بدی (مثلاً یه کلمه بهش اضافه کنی)، باید دو جا عوضش کنی

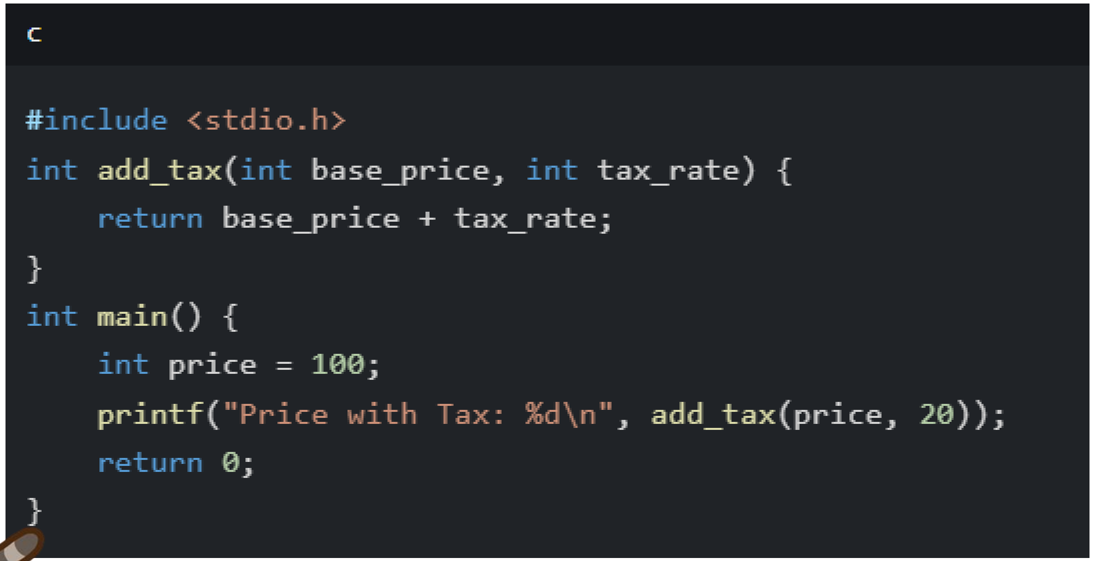
تکنیک بازسازی:
Remove Duplication
استفاده کردم، یعنی کد تکراری رو توی یه تابع جدا گذاشتم

اگه بخوای فرمت پیام رو تغییر بدی
فقط توی تابع عوضش می‌کنی و نیازی نیست چند جا
تغییر بدی
تو پروژه‌های بزرگ، این کار زمان زیادی صرفه‌جویی
می‌کنه

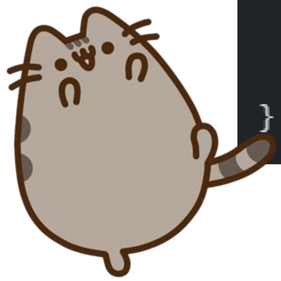
Practical Example: Introduce Parameter



```c
#include <stdio.h>
int main() {
    int price = 100;
    printf("%d\n", price + 20); // Fixed tax
    return 0;
}
```

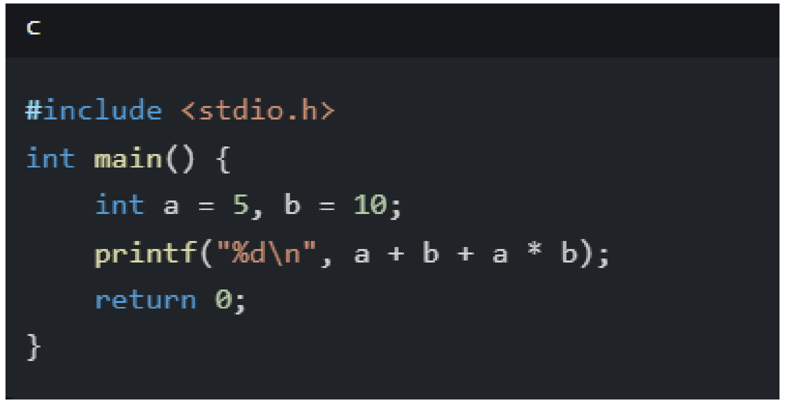


```c
#include <stdio.h>
int add_tax(int base_price, int tax_rate) {
    return base_price + tax_rate;
}
int main() {
    int price = 100;
    printf("Price with Tax: %d\n", add_tax(price, 20));
    return 0;
}
```

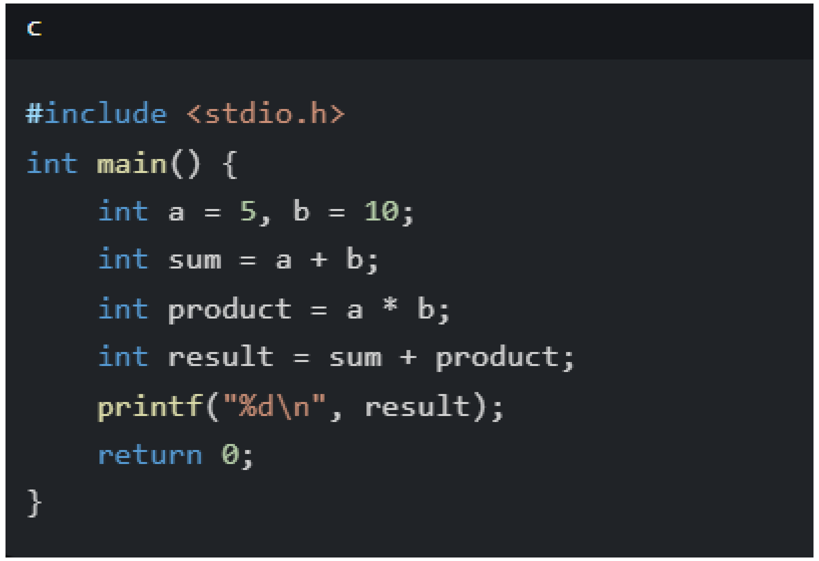


- **Issue in Before Code**: The tax rate (20) is hard-coded directly in the code. If you want to change the tax rate, you'd need to modify the code itself, which reduces flexibility.
- **Refactoring Technique**: I used **Introduce Parameter**, which means I added a parameter for the tax rate.
- **Changes in After Code**: I created a new function called add_tax that takes two parameters: base_price (base price) and tax_rate (tax rate). In main, instead of adding the fixed number 20, I called add_tax with the tax rate as a parameter.
- **Benefit**: Now you can change the tax rate without modifying the code. This is very useful in real projects because client requirements might change, and flexibility is important.

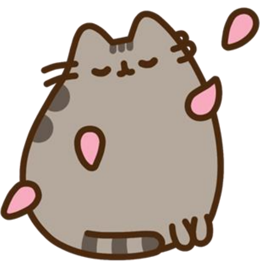
Practical Example: Extract Variable



```c
#include <stdio.h>
int main() {
    int a = 5, b = 10;
    printf("%d\n", a + b + a * b);
    return 0;
}
```



```c
#include <stdio.h>
int main() {
    int a = 5, b = 10;
    int sum = a + b;
    int product = a * b;
    int result = sum + product;
    printf("%d\n", result);
    return 0;
}
```



Issue in Before Code: A complex expression (a + b + a * b) is used directly in the printf statement. This makes the code **hard to read**, and if you need to **modify or debug** this expression, it's more difficult.

Refactoring Technique: I used Extract Variable, which means I broke down the expression into separate variables.

Changes in After Code: I stored a + b in a variable called sum, a * b in a variable called product, and the final result (sum + product) in a variable called result, which I then used in printf.

Benefit: The code is now more readable, and if you need to modify the calculations or find an error, it's easier. In large projects, this makes debugging faster.

Practical Example: Simplify Logic

```
#include
int main() {
 int years = 6, active = 1;
 if (years > 5) {
   if (active = =  1) {
     printf("Gets Bonus\n");
   }
 }
 return 0;
}
```

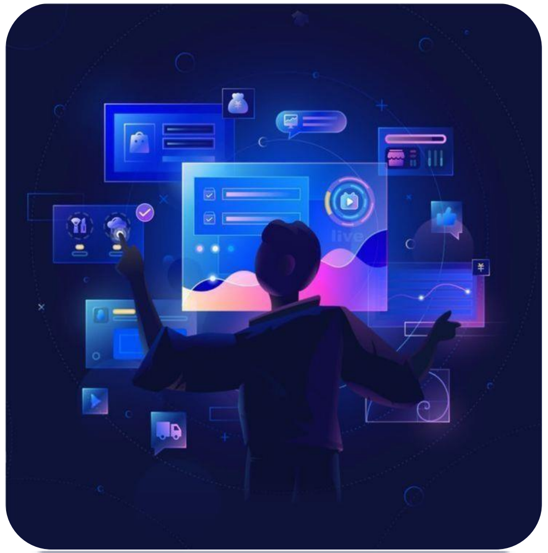
```
#include
int main() {
  int years = 6, active = 1;
  int gets_bonus = years > 5 && active == 1;
  if (gets_bonus)
    printf("Gets Bonus\n");
  return 0;
}
```

Refactoring Tools

Built-in IDE Tools :

- IntelliJ IDEA (for Java, Kotlin, etc.)
- Visual Studio (for C#, C++, etc.)
- Eclipse (for Java)
- PyCharm (for Python)

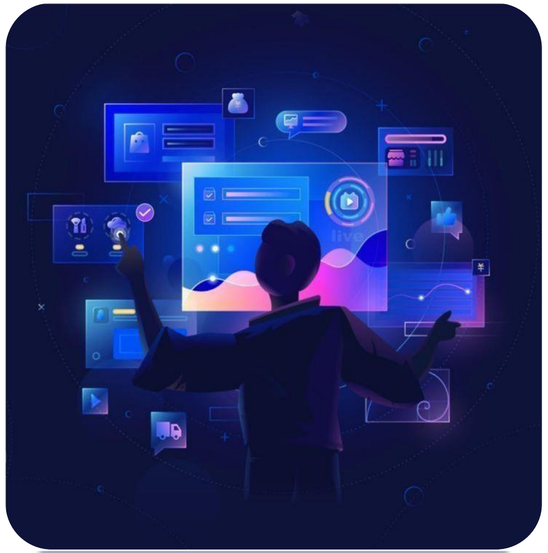
Plugins and Extensions



Refactoring Tools

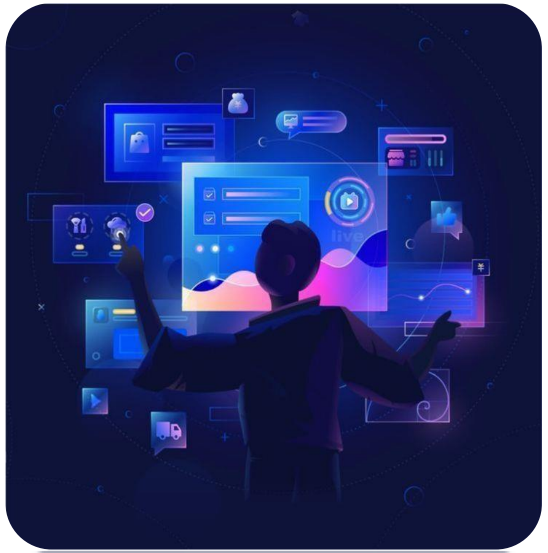
Static Analysis and Code Quality Tools:

- **SonarQube:**
  - **Capabilities:** Detecting duplicated code, high complexity, and suggesting structural changes.
- **Coverity:**
  - **Capabilities:** Suggesting removal of unnecessary code and performance improvements.
- **PMD (for Java):**
  - **Capabilities:** Detecting code smells and recommending optimizations.



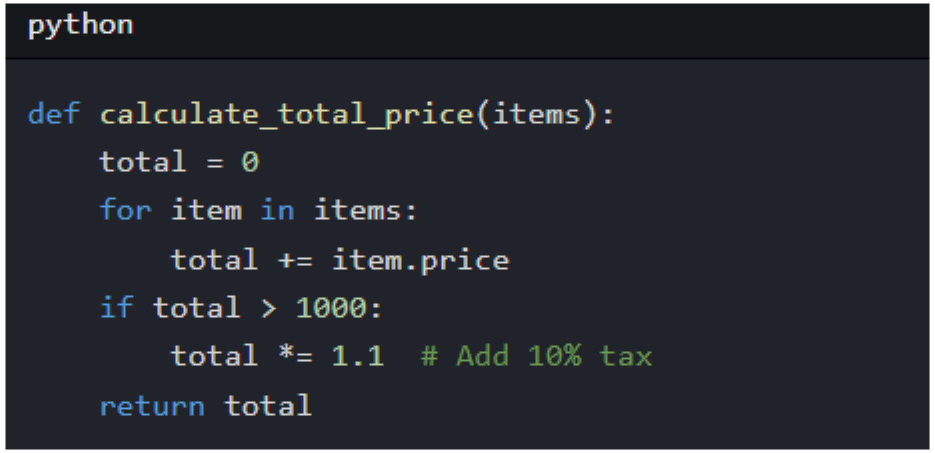
Common Capabilities of Refactoring Tools

- **Rename**: Renaming variables, methods, classes, or packages across a project.
- **Example:** Renaming variable `x` to `userCount` in IntelliJ IDEA.
- **Extract Method/Function**: Separating a code block into a new method or function.
- **Example:** Extracting discount calculation logic into a separate function in PyCharm.
- **Inline Variable/Method**: Replacing a variable or method with its direct value or functionality.
- **Example:** Replacing a temporary variable with a constant in Visual Studio.
- **Move**: Relocating a class, method, or file to another location (e.g., a new package or module).
- **Example:** Moving a class to a new package in Eclipse.
- **Change Signature**: Modifying a method's parameters, return type, or name.
- **Example:** Adding a new parameter to a method in IntelliJ IDEA.
- **Extract Class/Interface**: Creating a new class or interface from existing code.
- **Example:** Extracting validation logic into a new class in Visual Studio.
- **Simplify Conditionals**: Simplifying complex conditional statements.
- **Example:** Converting nested conditions to a simpler form in PyCharm.
- **Remove Dead Code**: Eliminating unused or unnecessary code.
- **Example:** Removing undefined variables in SonarLint.
- **Organize Imports**: Sorting and removing unnecessary imports.
- **Example:** Removing unused imports in Eclipse.



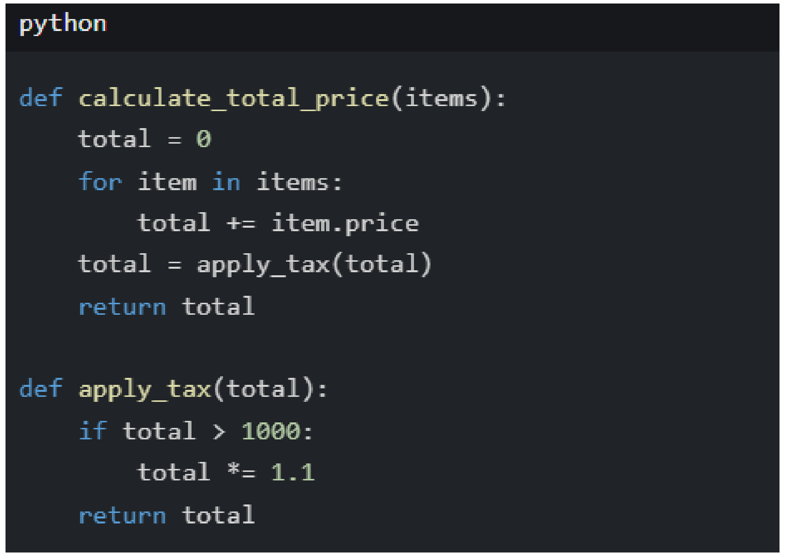
Examples with Refactoring Tools

A Python function contains complex tax calculation logic



python

```python
def calculate_total_price(items):
    total = 0
    for item in items:
        total += item.price
    if total > 1000:
        total *= 1.1  # Add 10% tax
    return total
```



python

```python
def calculate_total_price(items):
    total = 0
    for item in items:
        total += item.price
    total = apply_tax(total)
    return total

def apply_tax(total):
    if total > 1000:
        total *= 1.1
    return total
```

Using PyCharm for Extract Method

Action:

- Select the tax calculation block
- (`if total > 1000: total *= 1.1`).
- Choose **Extract Method** from the Refactor menu.
- Name the new method `apply_tax`.
- PyCharm creates a new function:

Result: Tax logic is separated, making the code more modular.

Real life Scenario

The Story Begins...

a spirited software engineer

- **Terrible Slowness:** When order volume spikes, the system practically goes into a coma.
- **Messy Code:** Long methods, weird variable names (like x and tmp), and logic that looks like it was written in a rush.
- **Adding New Features? A Nightmare!** The product team wants to add PayPal payment support, but touching the code feels like walking through a minefield.

Stage 1: Sleuthing and Mapping (Analysis and Planning)

The engineer needs to figure out why this system is such a pain. Like a detective, she digs into the code and talks to the team.

Diving into the Code

Talking to the Team

Setting Goals

Checking Tests

Tools:

- **PyCharm:** Like a treasure map showing the entire codebase.
- **SonarLint/SonarQube:** Like a tracker that pinpoints code issues.
- **Jira:** For logging problems and planning

Stage 2: Arming Up with Shields (Tests and Environment)

Strengthening Tests

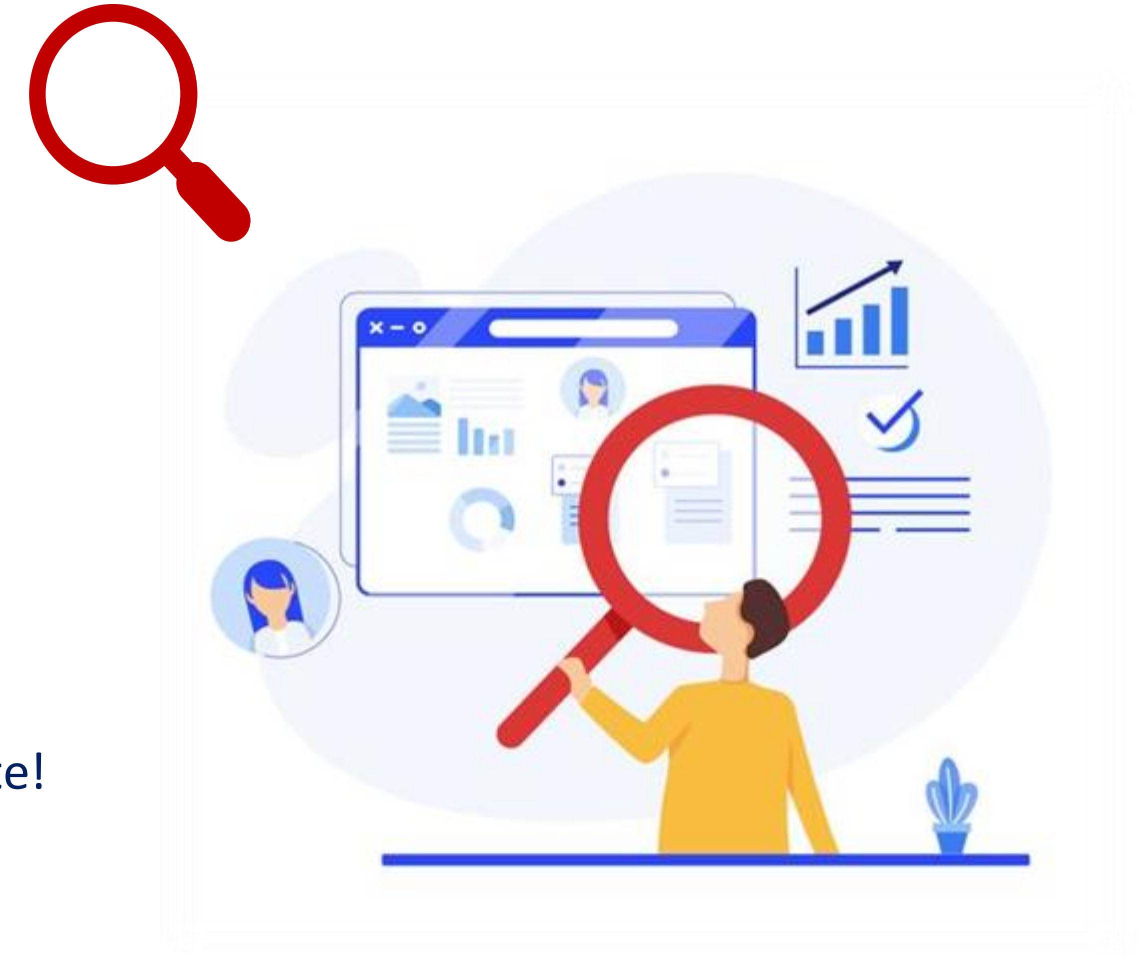
Running Tests

Backup Version

Setting Up CI/CD

Tools:

- **pytest:** Like a guard ensuring the code doesn’t break.
- **Git:** So you always have an escape route!
- **GitHub Actions:** For automating tests.
- **Docker:** To make the test environment match production.



Stage 3: Small-Scale Refactoring

Make the code clean

Code Review

Stage 4: Large-Scale Refactoring

- **Designing a New Blueprint**
- **Building the New Service**
- **Separating the Database**
- **Cool APIs**
- **Connecting to Others**
- **Testing and Launching**

Tools:

- **FastAPI:** For building fast, cool APIs.
- **SQLAlchemy:** For managing the database like a pro.
- **Docker/Kubernetes:** To make the service launch-ready like a spaceship.
- **Locust:** To test if the service can handle pressure without crumbling!

Stage 5: Celebrating and Documenting 🙌

- Code Review
- Cool Documentation
- Gathering Feedback
- Celebrating Success

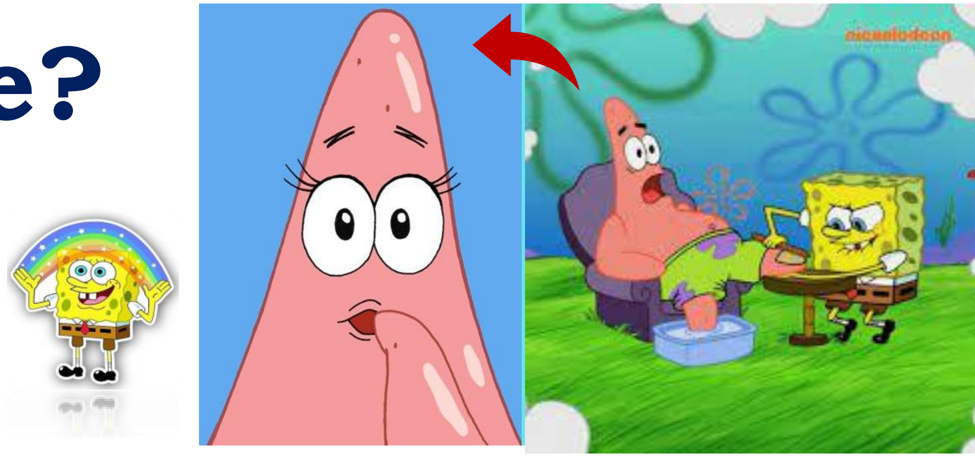
The new service goes live in production, and the system’s speed triples. The product team loves it because adding PayPal is now a breeze!

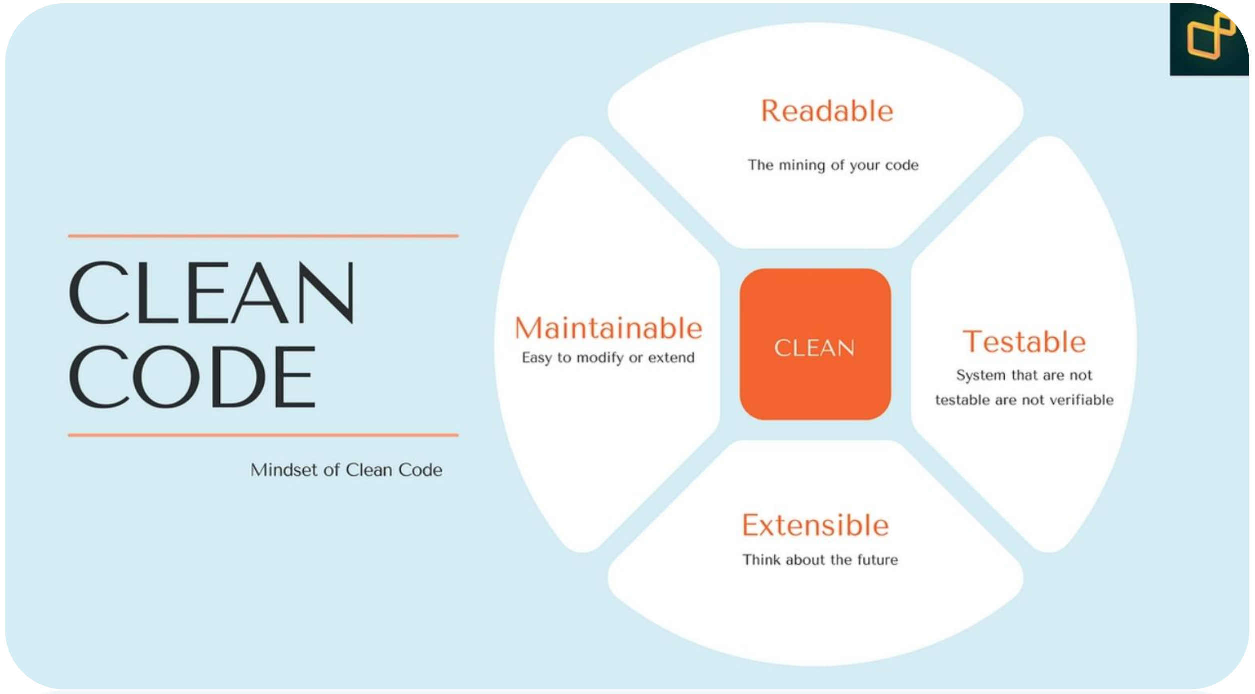
Best Refactoring Practices

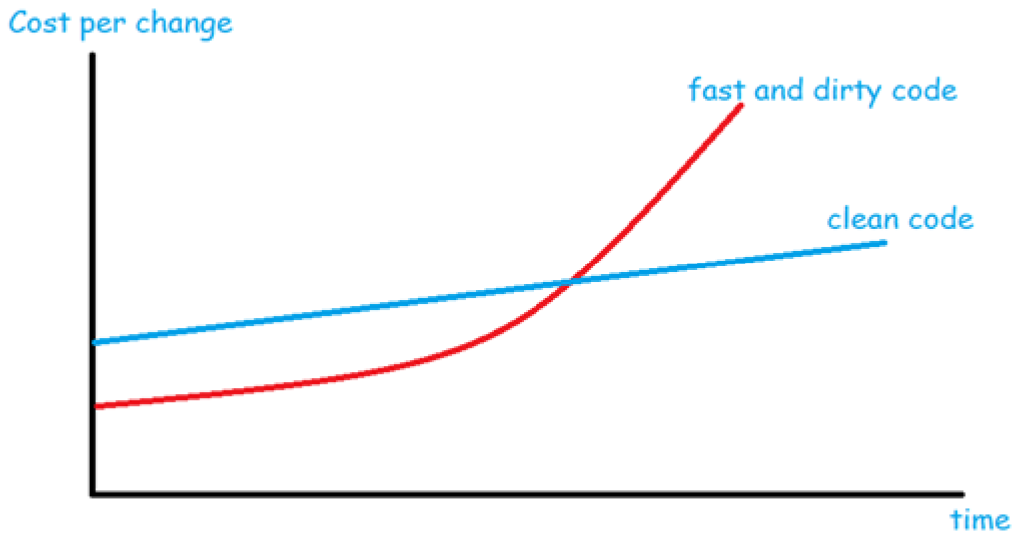
- **Always Start with Tests**: Without tests, refactoring is like diving without checking the water's depth.
- **Take Small Steps**: Break big changes into tiny, manageable pieces.
- **Use Tools**: IDEs like PyCharm and analyzers like SonarQube are your best friends.
- **Keep the Team in the Loop**: Code reviews and team alignment prevent tons of issues.
- **Monitor Performance**: Post-refactoring, use tools like Prometheus to confirm the system's better.
- **Don't Forget Docs**: Clean code without docs is like a book without a table of contents.

What is Clean Code?

The Art of Programming







If it works don't touch it ✗

Not just work, Should be EASY to work with

Clean Code Practices

- **Meaningful Naming**
- **Small and Focused Functions**
- **Avoid Duplication**
- **Useful Comments**
- **Error Handling**
- **Consistent Formatting**
- **Testability**

Keep
It
Simple
Stupid

Clean Code Architecture

Separation of Concerns

Dependency Inversion

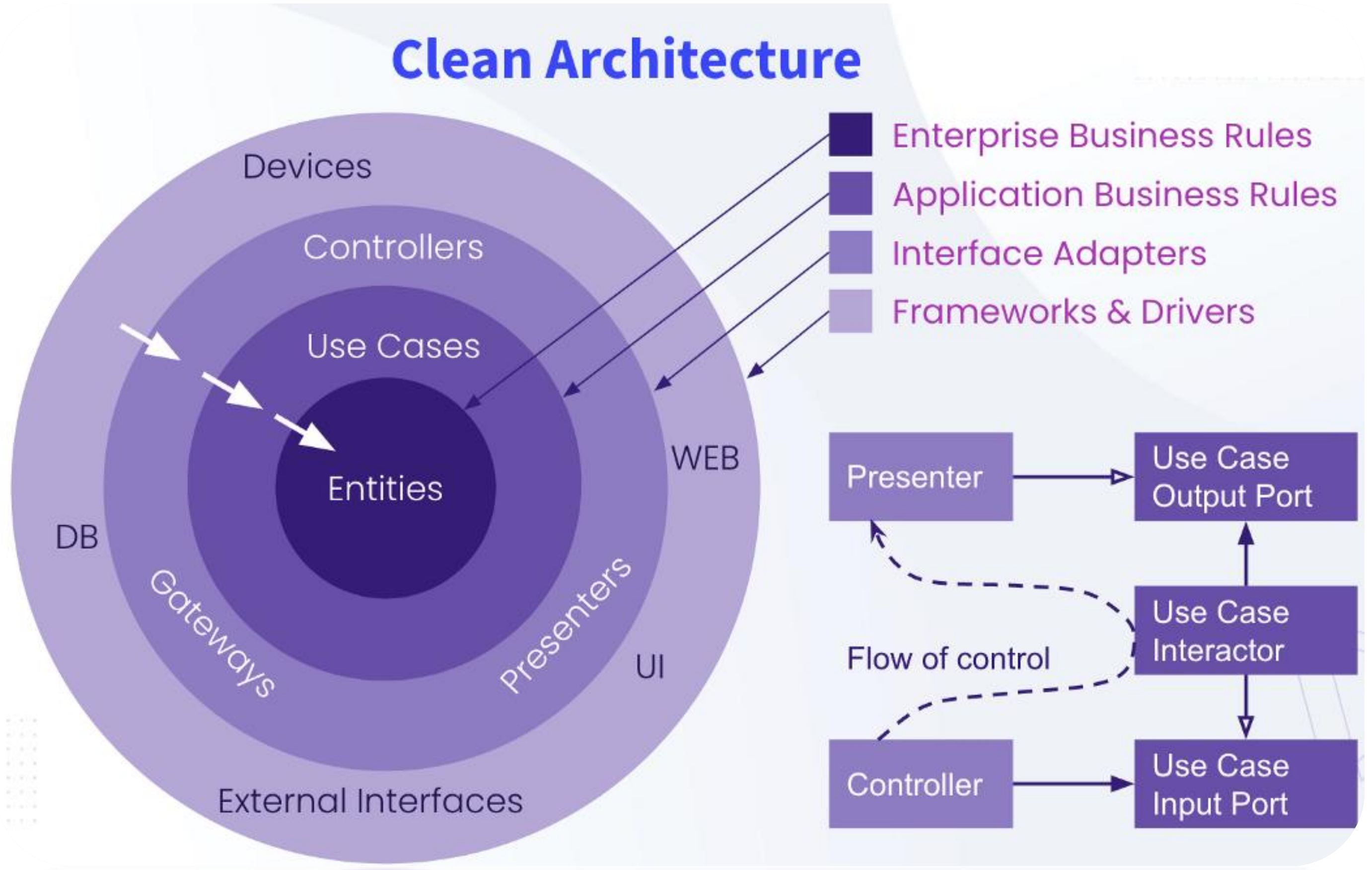
Layered Architecture

Design Patterns

Framework Independence

Dependency Management

Architecture Documentation



بهبود مستمر از طریق بازسازی

-اشکال‌زدایی و تحویل سریع‌تر
-مشتریان و مدیران راضی
-رشد شغلی
-صرفه‌جویی در هزینه‌ها