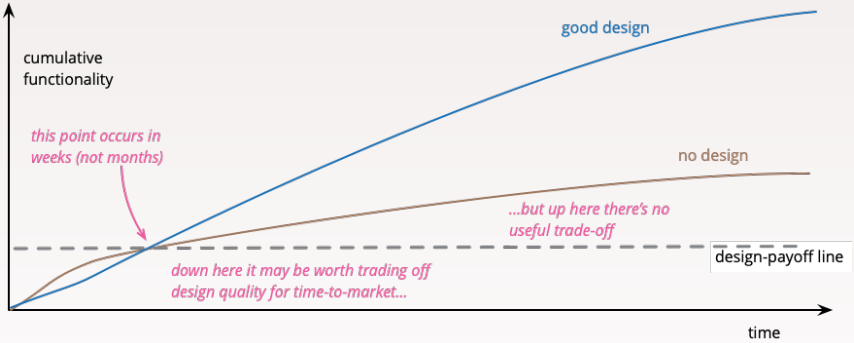
04834580 Software Engineering (Honor Track) 2025-26

Code Smells & Anti-Patterns

Sergey Mechtaev
mechtaev@pku.edu.cn
School of Computer Science, Peking University

Design Stamina Hypothesis [1]



You can save short-term time by neglecting design, but this accumulates **technical debt** *which will slow your productivity later. — Martin Fowler [1]*

Definition (Technical Debt)

The implied cost of additional work in the future resulting from choosing an expedient solution over a more robust one.

> *The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt. — Ward Cunningham [2]*

Definition (Code Smell [3])

Suboptimal design decisions applied by developers that can negatively affect the overall maintainability of a software system.

Code smells are one of the symptoms of the technical debt.

Code Smells

Common code smells discussed by Martin Fowler [4]:

- Duplicated Code
- Long Methods
- Large Classes
- Long Parameter Lists
- Divergent Change
- Shotgun Surgery
- Feature Envy
- Data Clumps
- Primitive Obsession
- Switch Statements
- Parallel Inheritance Hierarchies
- Lazy Class
- Speculative Generality
- Temporary Field
- Message Chains
- Middle Man
- Insider Trading
- Alternative Classes with Different Interfaces
- Incomplete Library Class
- Data Class
- Refused Bequest
- Comments

Code Smells

Code smells proposed by Wake [5]:

- Type Embedded in Name
- Uncommunicative Names
- Inconsistent Names
- Dead Code
- Null Check
- Complicated Boolean Expression
- Special Case
- Magic Numbers

Code Smells

Code smells proposed by Kerievsky [6]:

- Conditional Complexity
- Indecent Exposure
- Solution Sprawl
- Combinatorial Explosion
- Oddball Solution

Anti-Patterns

Development anti-patterns by Brown et al. [7]:

- The Blob
- Continuous Obsolescence
- Lava Flow
- Ambiguous Viewpoint
- Functional Decomposition
- Poltergeists
- Boat Anchor
- Golden Hammer
- Dead End
- Spaghetti Code
- Input Kludge
- Walking through a Minefield
- Cut-and-Paste Programming
- Mushroom Management

Anti-Patterns

Architectural anti-patterns by Brown et al. [7]:

- Autogenerated Stovepipe
- Stovepipe Enterprise
- Jumble
- Stovepipe System
- Cover Your Assets
- Vendor Lock-In
- Wolf Ticket
- Architecture by Implication
- Warm Bodies
- Design by Committee
- Swiss Army Knife
- Reinvent the Wheel
- The Grand Old Duke of York

Definition

The same code structure repeats in more than one place.

```
int sumA = 0;
for (int i = 0; i < 3; i++)
   sumA += arrayA[i];
int avgA = sumA / 3;

int sumB = 0;
for (int i = 0; i < 4; i++)
   sumB += arrayB[i];
int avgB = sumB / 4;
```

Eliminating duplicate code via refactoring:

```
int calcAvg(int[] a, int n) {
    int sum = 0;
    for (int i = 0; i < n; i++)
        sum += a[i];
    return sum / n;
}
int avgA = calcAvg(arrayA, 3);
int avgB = calcAvg(arrayB, 4);
```

Long Methods/Large Classes

Definition

Methods or classes, whose excessive length make them hard to understand.

Such methods/classes often violate **separation of concerns**. Need to be decomposed into smaller classes via refactoring.

Long Parameter Lists

北京大学

Definition

Long parameter lists that are hard to understand, because they become inconsistent and difficult to use, and that are frequently changing as you need more data.

```
drawLine(int xBegin, int yBegin, int xEnd, int yEnd,
         int red, int green, int blue, int alpha);
```

Long Parameter Lists

北京大学

Addressing long parameter list via refactoring:

```
Point begin = new Point(xBegin, yBegin);
Point end = new Point(xEnd, yEnd);
Color color = new Color(red, green, blue, alpha);
drawLine(begin, end, color);
```

Definition

Divergent change occurs when one class is commonly changed in different ways for different reasons (violation of **single responsibility principle**).

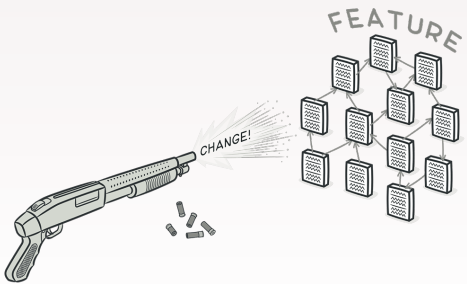
For example, the following holds inside the same class:

- "I will have to change these three methods every time I get a new database";
- "I have to change these four methods every time there is a new financial instrument".

You likely have a situation in which two objects are better than one.

Definition

When every time you make a kind of change, you have to make a lot of little changes to a lot of different classes.



Definition

A method that seems more interested in a class other than the one it actually is in.

```python
class ShoppingItem:
    name: str
    price: float
    tax: float

class Order:
    ...
    def get_bill_total(self, items: list[ShoppingItem]) -> float:
        return sum([item.price * item.tax for item in items])

    def get_receipt_string(self, items: list[ShoppingItem]) -> list[str]:
        return [f"{item.name}: {item.price * item.tax}$" for item in items]

    def create_receipt(self, items: list[ShoppingItem]) -> float:
        bill = self.get_bill_total(items)
        receipt = self.get_receipt_string(items).join('\n')
        return f"{receipt}\nBill {bill}"
```

Feature Envy (Solution)

北京大学

```python
class ShoppingItem:
    name: str
    price: float
    tax: float

    @property
    def taxed_price(self) -> float:
        return self.price * self.tax

    def get_receipt_string(self) -> str:
        return f"{self.name}: {self.price * self.tax}$"

class Order:
    ...
    def get_bill_total(items: list[ShoppingItem]) -> float:
        return sum([item.taxed_price for item in items])

    def get_receipt_string(items: list[ShoppingItem]) -> list[str]:
        return [item.get_receipt_string() for item in items]

    def create_receipt(items: list[ShoppingItem]) -> float:
        bill = self.get_bill_total(items)
        receipt = self.get_receipt_string(items).join('\n')
        return f"{receipt}\nBill: {bill}$"
```

Definition

Data items that tend to be used in groups together.

```
def colorize(red: int, green: int, blue: int):
    ...
```

Addressed by introducing a data structure:

```
class RGB:
    red: int
    green: int
    blue: int


def colorize(rgb: RGB):
    ...
```

Definition

Whenever a variable that is just a simple string, or an int simulates being a more abstract concept, which could be an object.

```
birthday_date: str = "1998-03-04"
name_day_date: str = "2021-03-20"
```

Addressed by introducing appropriate data structures:

```
class Date:
    year: int
    month: int
    day: int

    def __str__(self):
      return f"{self.year}-{self.month}-{self.day}"

birthday: Date = Date(1998, 03, 04)
name_day: Date = Date(2021, 03, 20)
```

Definition

Use of explicit switch statements instead of dynamic dispatch in object-oriented languages.

```
def calculate_area(shape):
    if shape['type'] == 'circle':
        return math.pi * (shape['radius'] ** 2)
    elif shape['type'] == 'rectangle':
        return shape['width'] * shape['height']
    else:
        raise ValueError("Unknown shape type")

circle = {'type': 'circle', 'radius': 5}
rectangle = {'type': 'rectangle', 'width': 4, 'height': 6}
```

Addressed by applying dynamic dispatch:

```
class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius
    def area(self):
        return math.pi * (self.radius ** 2)

class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height
    def area(self):
        return self.width * self.height

class Triangle(Shape):
    def __init__(self, base, height):
        self.base = base
        self.height = height
    def area(self):
        return 0.5 * self.base * self.height
```

Definition

Every time you make a subclass of one class, you also have to make a subclass of another (a special case of shotgun surgery).

```
class User(ABC):
    ...
    functions: Functions

class Functions(ABC):
    ...

class BasicUser(User):
    ...

class BasicFunctions(Functions):
    ...

class PremiumUser(User):
    ...

class PremiumFunctions(Functions):
    ...
```

Definition

Each class you create costs money to maintain and understand. A class that isn't doing enough to pay for itself should be eliminated.

```
class Strength:
    value: int

class Person:
    health: int
    intelligence: int
    strength: Strength
```

Definition

Features added in preparation for the future, guessing they will be useful, but that time never came.

```
class Animal:
    health: int

class Human(Animal):
    name: str
    attack: int
    defense: int

class Swordsman(Human):
    ...

class Archer(Human):
    ...

class Pikeman(Human):
    ...
```

Definition

Temporary field is a variable created where it is not needed.

```python
@dataclass
class MyDateTime:
    def __init__(self, year, month, day):
        self.year = year
        self.month = month
        self.day = day
        self.full_date = f"{year}, {month}, {day}"

    def foo(self):
        ...

    def goo(self):
        ...

    def hoo(self):
        ...

    def __str__(self):
        return self.full_date
```

Definition

Long sequences of methods calls indicate hidden dependencies by being intermediaries.

```
class Minion:
    _location: Location

    def action(self):
        ...
        if self._location.field.is_frontline():
            ...

class Location:
    field: Field

class Field:
    def is_frontline(self)
        ...
```

Definition

The class that only performs delegation work to other classes.

```
class Minion:
    _location: Location

    def action(self):
        ...
        if self.is_frontline():
            ...

    def is_frontline(self)
        return self._location.is_frontline()

class Location:
    _field: Field

    def is_frontline(self)
        return self._field.is_frontline()

class Field:
    def is_frontline(self)
        ...
```

Definition

"Classes spend too much time delving in each other's private parts."

```python
@dataclass
class Commit:
    name: str

    def push(self, repo: Repo):
        repo.push(self.name)

    def commit(self, url: str):
        ...

@dataclass
class Repo:
    url: str

    def push(self, name: str):
        ...

    def commit(self, commit: Commit):
        commit.commit(self.url)
```

Definition

If two classes have the same functionality but different implementations.

```
class Snowman(Humanoid):
    def hug_snowman():
        ...

class Zombie(Humanoid):
    def hug_zombie():
        ...
```

Incomplete Library Class

Definition

A library API arbitrarily omits some useful capabilities.

Null Check

Definition

Use of NULL causes a multitude of undefined or null checks everywhere: in guard checks, in condition blocks, and verifications clauses.

Definition

Use of hard-to-understand boolean expressions.

```
def cook(ready: bool, bag: list):
  if (ready):
    if (['raspberry', 'apple', 'tomato'] in bag and
        ['carrot', 'spinach', 'garlic'] not in bag):
          ...
```

Complicated Boolean Expression

Addressed via refactoring:

```
# "ready" extracted out of the function scope
def cook(bag: list):
    def hasFruit(container: list) -> bool:
        return ['raspberry', 'apple', 'tomato'] in container
    def hasVeggie(container: list) -> bool:
        return ['carrot', 'spinach', 'garlic'] in container

    if not hasFruit(bag):
        return
    if hasVeggie(bag):
        return
    ...
```

Magic Numbers

Definition

Using numbers that do not convey clear meaning.

```
def calculateDamage(...) -> int:
    total_damage = ...
    return math.max(100, damage)
```

Better:

```
def calculateDamage(...) -> int:
    total_damage = ...

    MAX_DAMAGE_CAP: int = 100
    return math.max(MAX_DAMAGE_CAP, total_damage)
```

References I

[1] Martin Fowler.
Design stamina hypothesis.
https://martinfowler.com/bliki/DesignStaminaHypothesis.html, 2007.

[2] Ward Cunningham.
The wycash portfolio management system.
https://c2.com/doc/oopsla92.html, 1992.

[3] Dario Di Nucci, Fabio Palomba, Damian A Tamburri, Alexander Serebrenik, and Andrea De Lucia.
Detecting code smells using machine learning techniques: Are we there yet?
In *2018 ieee 25th international conference on software analysis, evolution and reengineering (saner)*, pages 612–621. IEEE, 2018.

[4] Martin Fowler.
Refactoring: improving the design of existing code.
Addison-Wesley Professional, 2018.

[5] William C Wake.
Refactoring workbook.
Addison-Wesley Professional, 2004.

[6] Joshua Kerievsky.
Refactoring to patterns.
Pearson Deutschland GmbH, 2005.

[7] William H Brown, Raphael C Malveau, Hays W" Skip" McCormick, and Thomas J Mowbray.
AntiPatterns: refactoring software, architectures, and projects in crisis.
John Wiley & Sons, Inc., 1998.