Code Smells & how to solve them

A practical reference showing common code smells and how to fix them with **SOLID** principles, and common design patterns.

> IE University - BCSAI: Software Development & DevOps. 2025

Introduction

The SOLID Principles

The **SOLID principles** are five foundational guidelines in object-oriented software design. They are meant to make code:

- **Maintainable** – easier to understand, extend, and refactor.
- **Flexible** – resistant to breaking when requirements change.
- **Reusable** – components can be used across projects or modules.
- **Testable** – smaller, decoupled units of code are easier to unit test.
- **Scalable** – supports the growth of systems without turning into “spaghetti code.”

In short, they help prevent the **fragile code problem**: when making a small change in one place causes unexpected bugs in many others. Following SOLID encourages **clean architecture**, **separation of concerns**, and **dependency management**, all of which are crucial in professional software engineering.

The principles themselves existed **individually** before the acronym. They were popularized by **Robert C. Martin (Uncle Bob)** in his writings on software design in the 1990s and 2000s. The **acronym “SOLID”** was introduced later by **Michael Feathers** (a software developer and author) around 2000–2002, as a mnemonic to group these five core principles.

Principles

1. **Single Responsibility Principle (SRP)**
   A class should only have one reason to change. Each class or module should focus on a single piece of functionality. For example, a class that both validates and saves data has two reasons to change (validation rules vs. persistence), violating SRP.
2. **Open/Closed Principle (OCP)**
   Software entities should be open for extension but closed for modification. This avoids breaking existing behavior when new features are introduced. Achieving OCP often means relying on abstractions and polymorphism so new behaviors can be added without editing old code.

3. **Liskov Substitution Principle (LSP)**
   Subtypes should be substitutable for their base types. If a derived class breaks the expectations of its parent (for example, throwing `NotImplementedError` in a required method), it violates LSP. Following LSP ensures reliable polymorphism and prevents fragile hierarchies.
4. **Interface Segregation Principle (ISP)**
   Clients should not be forced to depend on methods they do not use. Instead of large, fat interfaces with unrelated responsibilities, prefer smaller, more cohesive abstractions. This reduces coupling and makes systems easier to adapt and maintain.
5. **Dependency Inversion Principle (DIP)**
   High-level modules should not depend on low-level modules; both should depend on abstractions. This principle ensures flexibility and testability.

Design Patterns

Design patterns are *proven, reusable solutions* to common problems in software design. They're not code snippets you copy-paste, but rather **templates or blueprints** you can adapt to your situation. They're important because they:

- **Provide shared vocabulary** – developers can say "Singleton" or "Observer" and immediately understand the structure.
- **Capture best practices** – distilled wisdom from decades of software engineering.
- **Improve communication** – teams can reason about architecture more effectively.
- **Reduce errors** – patterns are tested solutions, less likely to introduce design flaws.
- **Speed up development** – instead of reinventing the wheel, you reuse tried-and-true approaches.

In short, design patterns help you **design better systems faster** with less risk.

The term comes from **Christopher Alexander**, an architect (not a programmer!) who published *A Pattern Language* (1977), describing design patterns in architecture (buildings, towns, spaces). Then, the concept was **adapted to software engineering** in the late 1980s/early 1990s. Finally, the book **"Design Patterns: Elements of Reusable Object-Oriented Software" (1994)** by the "Gang of Four" (Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides) made these concepts widely spreaded along software development comunity.

This book introduced **23 classic design patterns**, grouped into three categories:

1. **Creational** (object creation, e.g., Singleton, Factory Method, Builder)
2. **Structural** (composition of classes/objects, e.g., Adapter, Composite, Decorator)
3. **Behavioral** (communication and responsibility, e.g., Observer, Strategy, Command)

Code Smells

A **code smell** is a surface indicator that usually signals a deeper problem in the codebase. It's not a bug (the code may still work), but it suggests that **the design or implementation is flawed** and may cause trouble later. **Code smells are warning signs** that your software may rot if you don't refactor.

They're important because they:

- **Indicate maintainability issues** – messy or overcomplicated code is harder to extend or debug.
- **Highlight technical debt** – code smells are often symptoms of rushed or poorly designed solutions.
- **Guide refactoring** – spotting a smell tells you where to clean up.
- **Prevent fragility** – smelly code tends to break easily when new requirements are added.
- **Improve readability & collaboration** – cleaner code is easier for teammates (and your future self) to understand.

The idea of "smelly code" was **popularized by Kent Beck** while he was helping **Martin Fowler** write *Refactoring: Improving the Design of Existing Code* (1999). Fowler formally catalogued them in that book as a **taxonomy of poor design choices** that developers should look out for.

Here are some classic examples (from Fowler's catalog, plus modern usage):

- **Bloaters** (too big)
  - **Long Method** – a method does too much, becomes unreadable.
  - **Large Class** – a class has too many responsibilities.
  - **Primitive Obsession** – using basic types (ints, strings) instead of small objects.
- **Object-Orientation Abusers**
  - **Switch Statements** – using switches instead of polymorphism.
  - **Refused Bequest** – subclasses don't want inherited behavior.
- **Change Preventers**
  - **Divergent Change** – one class needs to be modified for many different reasons.
  - **Shotgun Surgery** – one change forces edits in many different classes.
- **Dispensables** (unnecessary code)
  - **Duplicate Code** – same logic in multiple places.
  - **Lazy Class** – a class that doesn't do enough to justify its existence.
  - **Speculative Generality** – code written "just in case" but never used.
- **Couplers** (too tangled)
  - **Feature Envy** – a method uses more of another class's data than its own.
  - **Inappropriate Intimacy** – classes know too much about each other's internals.
  - **Message Chains** – long chains like `a.getB().getC().getD()`.

Related Bibliography

- Robert C. Martin, *Agile Software Development: Principles, Patterns, and Practices* (2002)
- Robert C. Martin, *Clean Architecture* (2017)
- Martin Fowler, *Refactoring: Improving the Design of Existing Code* (2nd ed., 2018)
- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software* (1994)
- Michael Feathers, *Working Effectively with Legacy Code* (2004)
- Kent Beck, *Implementation Patterns* (2007)
- Sandro Mancuso, *The Software Craftsman: Professionalism, Pragmatism, Pride* (2014)

Code Examples

1) Long Method / Complex Conditionals → Strategy + Factory Method

Smell: One function handles multiple shipping rules with branching logic. As business expands, more conditions are added, making the function longer, harder to read, and riskier to modify. It also forces unrelated tests into the same place, increasing coupling and reducing clarity.

Before:

```python
def shipping_cost(speed: str, weight: float) -> float:
    if speed == "ground":
        return 5 + 0.5 * weight
    elif speed == "air":
        return 10 + 1.2 * weight
    elif speed == "drone":
        return 15 + 0.8 * weight
    else:
        raise ValueError("unknown speed")
```

- Violates: **SRP**, **OCP**.
- Why: One function mixes multiple shipping rules (many reasons to change), and adding a new rule means editing this function instead of extending behavior.

After:

```python
from abc import ABC, abstractmethod

class ShippingCost(ABC):
```

```
    @abstractmethod
    def cost(self, weight: float) -> float: ...

class Ground(ShippingCost):
    def cost(self, weight: float) -> float: return 5 + 0.5 * weight
class Air(ShippingCost):
    def cost(self, weight: float) -> float: return 10 + 1.2 * weight
class Drone(ShippingCost):
    def cost(self, weight: float) -> float: return 15 + 0.8 * weight

class ShippingFactory:
    @staticmethod
    def for_speed(speed: str) -> ShippingCost:
        return {"ground": Ground(), "air": Air(), "drone": Drone()}[speed]

cost = ShippingFactory.for_speed("air").cost(3.0)
```

---

2) Giant Dispatcher → Command

Smell: Controllers often grow with `if/elif` statements to route actions. Every new command requires editing this block, mixing orchestration with business logic. This centralizes too much responsibility, reducing extensibility and readability.

Before:

```
def handle(action: str, **payload):
    if action == "create_user":
        return {"status": "user_created"}
    elif action == "export_csv":
        return {"status": "csv_exported"}
    else:
        raise KeyError(action)
```

- Violates: **SRP**, **OCP**.
- Why: A single handler owns orchestration for many actions (too many responsibilities). New actions require modifying the central `if/elif` block.

After:

```
from abc import ABC, abstractmethod

class Command(ABC):
    @abstractmethod
    def execute(self, **kwargs): ...
```

```
class CreateUser(Command):
    def execute(self, **kwargs): return {"status": "user_created"}
class ExportCSV(Command):
    def execute(self, **kwargs): return {"status": "csv_exported"}

REGISTRY: dict[str, Command] = {
    "create_user": CreateUser(),
    "export_csv": ExportCSV(),
}

def handle(action: str, **payload):
    return REGISTRY[action].execute(**payload)
```

---

3) God Object → Facade

Smell: A single class performs unrelated tasks (validation, persistence, notifications). This concentration of responsibility makes it hard to modify without breaking other features. It becomes a bottleneck that accumulates technical debt and brittle tests.

Before:

```
class OrderService:
    def place(self, order):
        if order.total <= 0: raise ValueError("bad")
        print("saved", order.id)
        print("emailed", order.customer)
```

- Violates: **SRP**, **ISP** (by implication).
- Why: `OrderService` validates, persists, and notifies—distinct responsibilities. Clients call a “fat” surface to get unrelated work done.

After:

```
class Validator:
    def valid(self, order): return order.total > 0
class Repository:
    def save(self, order): print("saved", order.id)
class Notifier:
    def email(self, order): print("emailed", order.customer)

class OrderService:  # Facade
    def __init__(self, v: Validator, r: Repository, n: Notifier):
        self._v, self._r, self._n = v, r, n
    def place(self, order):
        if not self._v.valid(order): raise ValueError("invalid")
```

```
        self._r.save(order)
        self._n.email(order)
```

---

4) Tight Coupling to Third-Party API → Adapter

Smell: Business code imports vendor SDK types, leaking external details across modules. Replacing the vendor requires sweeping edits and breaks tests. This tight coupling violates DIP and increases risk when providers change interfaces.

Before:

```python
from vendor import StripeSDK

def charge(cents: int, token: str) -> str:
    sdk = StripeSDK()
    res = sdk.pay(cents, token)
    return res["id"]
```

- Violates: **DIP**, often **ISP**.
- Why: High-level code depends directly on vendor concretes and their types; swapping the provider forces widespread edits and leaks third-party concerns into clients.

After:

```python
from typing import Protocol

class Payments(Protocol):
    def charge(self, cents: int, token: str) -> str: ...

class StripeSDK:
    def pay(self, amount_cents: int, source: str) -> dict:
        return {"id": "ch_123"}

class StripeAdapter(Payments):
    def __init__(self, sdk: StripeSDK): self.sdk = sdk
    def charge(self, cents: int, token: str) -> str:
        return self.sdk.pay(cents, token)["id"]

payments: Payments = StripeAdapter(StripeSDK())
charge_id = payments.charge(5000, "tok_visa")
```

---

5) Uncontrolled Access / Expensive Calls → Proxy

Smell: Many parts of the code hit slow or sensitive resources directly, duplicating access checks and caching or forgetting them entirely. Tests become slow and flaky; cross-cutting concerns leak into business logic, increasing coupling and maintenance cost.

Before:

```python
class WeatherAPI:
    def fetch(self, city: str) -> dict:
        print("remote...")
        return {"city": city, "temp": 21}

service = WeatherAPI()
service.fetch("Madrid")
service.fetch("Madrid")
```

- Violates: **SRP**, **OCP** (practically).
- Why: Callers must worry about caching/rate-limiting/permissions (cross-cutting concerns mixed with business logic). Adding access policies requires touching many call sites.

After:

```python
from functools import lru_cache

class WeatherAPI:
    def fetch(self, city: str) -> dict:
        print("remote...")
        return {"city": city, "temp": 21}

class CachedWeather(WeatherAPI):  # Proxy
    @lru_cache(maxsize=128)
    def fetch(self, city: str) -> dict:  # type: ignore[override]
        return super().fetch(city)

service = CachedWeather()
service.fetch("Madrid")  # remote
service.fetch("Madrid")  # cached
```

---

6) Global State / Hidden Dependencies → Singleton (careful) + DI

Smell: Modules mutate globals for configuration or clients, creating implicit coupling and test interference. Order-dependent imports and hard-to-reproduce bugs follow. The design hides dependencies and blocks substitution in tests.

Before:

```
# settings.py
API_KEY = "dev"

# service.py
from settings import API_KEY
print(API_KEY)
```

- Violates: **SRP**, **DIP**.
- Why: Implicit globals hide configuration dependencies, creating order-dependent behavior and making substitution/testing difficult; high-level code depends on concrete module state.

After:

```
from dataclasses import dataclass

@dataclass(frozen=True)
class Settings:
    db_url: str
    api_key: str

class AppConfig:
    _instance: "AppConfig | None" = None
    def __new__(cls, settings: Settings):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
            cls._instance.settings = settings
        return cls._instance

cfg = AppConfig(Settings(db_url="sqlite://", api_key="xyz"))
print(cfg.settings.api_key)
# Prefer passing cfg/settings into classes (DI) rather than global access.
```

---

7) Telescoping Constructors / Long Parameter Lists → Builder

Smell: Creating rich objects requires many ordered parameters and optional combinations. Call sites are noisy and error-prone, and defaults are unclear. Tests repeatedly construct similar objects with small variations.

Before:

```
class Report:
    def __init__(self, title, body, footer=None, fmt="pdf"):
        self.title, self.body, self.footer, self.fmt = title, body,
footer, fmt

r = Report("Sales Q3", "...", None, "html")
```

- Violates: **SRP** (construction concerns bleed into usage) and nudges **OCP** issues.
- Why: Call sites carry construction complexity and optional combos; many reasons to change the constructor surface, encouraging scattered modifications.

After:

```
from dataclasses import dataclass

@dataclass
class Report:
    title: str
    body: str
    footer: str | None = None
    fmt: str = "pdf"

class ReportBuilder:
    def __init__(self, title: str, body: str):
        self._r = Report(title, body)
    def footer(self, text: str): self._r.footer = text; return self
    def as_format(self, fmt: str): self._r.fmt = fmt; return self
    def build(self) -> Report: return self._r

report = (ReportBuilder("Sales Q3", "...")
          .footer("confidential")
          .as_format("html")
          .build())
```

---

8) Duplicated Code Across Similar Algorithms → Strategy

Smell: Multiple functions replicate the same pipeline with minor differences, causing copy-paste drift. Changes require editing many places, increasing defects. The variation should be isolated while shared steps remain reusable.

Before:

```
def process_square(xs):
    xs = sorted(set(xs))
```

```
    return [x*x for x in xs]

def process_negate(xs):
    xs = sorted(set(xs))
    return [-x for x in xs]
```

- Violates: **SRP**, **OCP**.
- Why: Each variant repeats the same pipeline with tiny differences (multiple responsibilities per function set); adding a variant means duplicating/editing existing code.

After:

```
from collections.abc import Callable
Process = Callable[[list[int]], list[int]]

def pipeline(data: list[int], transform: Process) -> list[int]:
    return transform(sorted(set(data)))

square: Process = lambda xs: [x*x for x in xs]
negate: Process = lambda xs: [-x for x in xs]

pipeline([3,1,2,2], square)
pipeline([3,1,2,2], negate)
```

---

9) Message Chains (Law of Demeter Violation) → Facade / Tell-Don't-Ask

Smell: Callers traverse deep object graphs like `order.customer.address.city`, exposing internals. Small changes force widespread edits (shotgun surgery). Encapsulation erodes, tests become brittle, and cognition cost rises for readers.

Before:

```
city = order.customer.address.city
```

- Violates: **SRP**, **ISP** (leaky interfaces).
- Why: Callers traverse deep object graphs, coupling to internals; small structural changes force edits everywhere, showing poor encapsulation and overly broad interfaces.

After:

```
class Address:
    def __init__(self, city: str): self.city = city
class Customer:
    def __init__(self, address: Address): self._address = address
    def city(self) -> str: return self._address.city
class Order:
    def __init__(self, customer: Customer): self._customer = customer
    def customer_city(self) -> str: return self._customer.city()

city = Order(Customer(Address("Sevilla"))).customer_city()
```

---

10) Scattered Notifications / Tight Publisher–Listener Coupling → Subscriber (Observer)

Smell: Subjects directly call many listeners or branch on who should react. Adding behavior edits the subject, increasing coupling and violating OCP. Event ordering and error handling are inconsistent across call sites.

Before:

```
def place_order(order):
    print("invoice→", order["id"])  # direct
    print("email→", order["id"])    # direct
```

- Violates: **OCP**, **DIP**.
- Why: The subject calls listeners directly; adding/removing reactions requires modifying the subject, and it depends on concretes instead of abstractions/events.

After:

```
from collections import defaultdict
from typing import Callable
Callback = Callable[[dict], None]

class EventBus:
    def __init__(self): self._subs: dict[str, list[Callback]] =
defaultdict(list)
    def subscribe(self, topic: str, cb: Callback):
self._subs[topic].append(cb)
    def publish(self, topic: str, event: dict):
        for cb in list(self._subs.get(topic, [])): cb(event)

bus = EventBus()
bus.subscribe("order.placed", lambda e: print("invoice→", e["id"]))
```

```
bus.subscribe("order.placed", lambda e: print("email→", e["id"]))

bus.publish("order.placed", {"id": 42})
```

---

11) Inappropriate Intimacy with Files/CLI → Facade + Command

Smell: Business logic touches the filesystem or shells out, mixing policy with mechanism. Side effects complicate testing and error handling. The code violates DIP by relying on concretes instead of abstractions.

Before:

```
from pathlib import Path

def save_report(path: str, payload: str):
    Path(path).write_text(payload, encoding="utf-8")
```

- Violates: **DIP**, **SRP**.
- Why: Business logic directly touches the filesystem/shell, mixing policy with mechanism; high-level use cases depend on low-level IO details.

After:

```
from pathlib import Path
from abc import ABC, abstractmethod

class FileSystem:
    def write(self, path: Path, content: str):
        path.write_text(content, encoding="utf-8")

class Command(ABC):
    @abstractmethod
    def execute(self, *args, **kwargs): ...

class SaveReport(Command):
    def __init__(self, fs: FileSystem): self.fs = fs
    def execute(self, path: str, payload: str):
        self.fs.write(Path(path), payload)

SaveReport(FileSystem()).execute("/tmp/r.txt", "hello")
```

---

12) Divergent Change (one module changes for many reasons) → Facade + Adapters

Smell: A single user module changes for UI, validation, and persistence simultaneously. Responsibilities clash, deployment risk increases, and unrelated edits cause conflicts. Splitting concerns reduces blast radius and aligns with SRP.

Before:

```
class UserModule:
    def register(self, username, password):
        if len(password) < 8: raise ValueError("weak")
        print("INSERT INTO users ...")
        print("render welcome page")
```

- Violates: **SRP**, **OCP**.
- Why: UI rendering, validation, and persistence live together; each concern changes for different reasons, and new requirements mean editing the same class repeatedly.

After:

```
class PasswordPolicy:
    def ensure_ok(self, pwd: str):
        if len(pwd) < 8: raise ValueError("weak")

class UserRepo:
    def add(self, username: str, password: str):
        print("insert", username); return True

class UserFacade:
    def __init__(self, repo: UserRepo, policy: PasswordPolicy):
        self.repo, self.policy = repo, policy
    def register(self, username: str, password: str):
        self.policy.ensure_ok(password)
        return self.repo.add(username, password)
```

Final reflections

Looking back at SOLID principles, design patterns, and code smells, we see they all help us write better code. SOLID gives strong foundations, design patterns provide proven solutions, and code smells warn us about problems. Together, they guide developers to build cleaner, easier, and more reliable software over time by using familiar definitions for every developer.

Pattern Designs Summary

- **Strategy**: Many algorithm variants → replace conditionals.
- **Command**: Encapsulate actions → queue/undo/logging.
- **Factory Method**: Delegate creation based on context.
- **Builder**: Construct complex objects step-by-step.
- **Adapter**: Fit third-party/legacy to your port.
- **Proxy**: Add caching, security, lazy loading transparently.
- **Facade**: Simplify a complex subsystem.
- **Subscriber (Observer)**: Decouple publishers from subscribers.
- **Singleton**: Single shared configuration (use sparingly).

Other Principles

These principles are very common, although less formal than SOLID. Very useful as guidelines when writing code.

- **KISS — Keep It Simple, Stupid**
  - **Idea:** Prefer the simplest solution that works. Complexity is a cost.
  - **Why:** Simple code is easier to read, test, change, and debug.
  - **Example:** Use a straightforward loop before reaching for metaprogramming or a heavy framework.
  - **Watch-out:** “Simple” ≠ “naive”. Keep correctness and clarity; avoid oversimplifying away needed structure.
- **DRY — Don’t Repeat Yourself**
  - **Idea:** Every piece of knowledge should have a single, unambiguous home.
  - **Why:** Duplicated logic drifts out of sync, causing bugs and extra maintenance.
  - **Example:** Extract shared validation into one function/module instead of copying it across endpoints.
  - **Watch-out:** Don’t over-abstract. If two bits of code only look similar today, premature unification can make things worse.
- **YAGNI — You Aren’t Gonna Need It**
  - **Idea:** Don’t build features, hooks, or abstractions until they’re actually needed.
  - **Why:** Speculative code adds complexity, slows delivery, and is often wrong about the future.
  - **Example:** Skip adding a plugin system on day one; implement it when a second real use-case appears.
  - **Watch-out:** YAGNI doesn’t mean ignoring **known** requirements like security, observability, or performance constraints you already have.

KISS keeps code clear, **DRY** keeps knowledge centralized, and **YAGNI** keeps scope honest. Together they reduce bugs, cost, and cognitive load.

Development Tips

- Depend on **abstractions** (Protocols/ABCs), not concretions (**DIP**).
- Keep classes tiny and cohesive (**SRP**, **ISP**).
- Prefer composition over inheritance; inject collaborators.
- Add behavior by **adding** classes/strategies, not editing existing ones (**OCP**).

Summary Table

Code Smell	**Violated Principle(s)**	**Suggested Pattern(s)**
Long method / complex conditionals	SRP, OCP	Strategy, Factory Method
Giant if/elif dispatcher	SRP, OCP	Command
God object / large class	SRP, ISP	Facade
Tight coupling to vendor API	DIP, ISP	Adapter
Uncontrolled access / expensive calls	SRP, OCP	Proxy
Global state / hidden dependencies	SRP, DIP	Singleton, Dependency Injection
Long parameter lists / telescoping constructors	SRP	Builder
Duplicated code across algorithms	SRP, OCP	Strategy
Message chains (Law of Demeter)	SRP, ISP	Facade, Proxy
Scattered notifications	OCP, DIP	Subscriber (Observer)
Business logic tied to filesystem/CLI	DIP, SRP	Facade, Command
Divergent change (multiple reasons to modify one module)	SRP, OCP	Facade, Adapter