Code smells

Code smells

“A person who never made a mistake never tried anything new.”

Albert Einstein

Code smells

- A *code smell* is a surface indication, a symptom - that something is not quite right in the system.
- It is not inherently bad on its own, but it’s a clue suggesting to investigate potential problems.
- Study the recurring types of smells is an effective way to identify the issues and solve them as early as possible, hence in the refactoring phase.

What can we do to go in the right direction?
Follow the **core principles!**

- **KISS** (keep it simple, simian!)
- **DRY** (don’t repeat yourself)
- **YAGNI** (you aren’t gonna need it)
- **SRP** (Single Responsibility Principle)
- **OCP** (Open Closed Principle)
- **LSP** (Liskov Substitution Principle)
- **Tell don’t ask** (Law of **Demeter**)

Law of Demeter

- Your method can call other methods in its class directly
- Your method can call methods on its own fields directly (but not on the fields' fields)
- When your method takes parameters, your method can call methods on those parameters directly.
- When your method creates local objects, that method can call methods on the local objects.
- But
    - One should not call methods on a global object (but it can be passed as a parameter
    - One should not have a chain of messages a.getB().getC().doSomething() in some class other than a's class.

What do we want to achieve following those principles?

- Maximize **Cohesion**
  - Cohesion is a metric telling **how strongly related and coherent** are the **responsibilities within the classes** of an application
- Minimize **Coupling**
  - Coupling is a metric for measuring **the degree of interdependence between the classes** of an application

Code smells

- **Duplicated Code**
  - When identical or very similar code exists in more than one location or duplicated knowledge. **DRY** violation. **Cohesion** violation.
- **Long Method**
  - Methods should do only one thing. Should do it well. Should do it only. One level of abstraction. No more than 10-15 lines.
  - **SRP** violation. **Cohesion** violation.
- **Large Class**
  - Classes should have only one responsibility. No more than 50 lines per class. **SRP** violation. **Cohesion** violation.

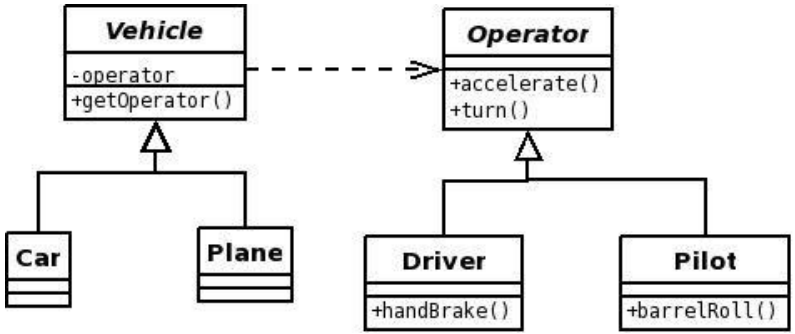
Code smells

- **Long Parameter List**
  - **0** (*niladic*) => **Ideal**, **1** (*monadic*) => **Ok**, **2** (*dyadic*) => **Acceptable**, **3** (*triadic*) => Debatable (but we **avoid it**), **3+**(*polyadic*) => **Only with special justification**. **Coupling** violation. Connascence of position.
- **Divergent Change**
  - When one class is commonly changed in different ways for different reasons (God class). **OCP**, **SRP** violation. **Cohesion** violation.
- **Shotgun Surgery**
  - Opposite of Divergent change. One change, forces lots of little changes in different classes. **DRY** violation. **Coupling** violation.

Code smells

- **Feature Envy**
  - A class that uses methods or properties of another class excessively.
    Tell, don’t ask violation. **Cohesion** and **Coupling** violation.
- **Data Clumps**
  - Same data items together in lots of places. Special case of duplicated code. **DRY** violation. **Cohesion** violation.
- **Primitive Obsession**
  - Don't use primitive types as substitutes for classes. If the data type is sufficiently complex, use a class to represent it. **Coupling** violation. **Connascence** of *meaning*.

- **Switch Statements**
  - Can lead to same switch statement scattered in different places. Can lead to **Dry** violation or **Cohesion** violation.
- **Parallel Inheritance Hierarchies**
  - Special case of shotgun surgery. Creating a subclass of one class, forces subclass of another.



Code smells

- **Data Class**
  - Classes that have fields, properties, and nothing else. Anaemic classes that contain no behaviour. Special case of lazy class. **Cohesion** violation.
- **Speculative Generality**
  - There is an unused class, method, field or parameter. **YAGNI** violation. **Cohesion** violation.
- **Temporary Field**
  - Class contains an instance variable set only in certain circumstances. **Cohesion** violation.

Code smells

- **Message Chains**
  - Too many dots: Dog.Body.Tail.Wag()
    Should be: Dog.ExpressHappiness()
    Law of **Demeter** violation. **Coupling** violation.
- **Middle Man**
  - If a class is delegating all its work, cut out the middleman. Beware classes that are wrappers over other classes or existing functionality. Special case of lazy class. **Cohesion** violation.
- **Inappropriate Intimacy**
  - A class that has dependencies on implementation details of another class. Special case of **feature envy**. **Cohesion** violation.

Code smells

- **Alternative Classes with Different Interfaces**
  - If two classes are similar on the inside, but different on the outside, perhaps they can be modified to share a common interface.
- **Incomplete Library Class**
  - Adding missing functionality by not changing library can lead to functionality implemented in odd places.
- **Lazy Class**
  - A class that does too little. May be acting only as middle man or a data class or can be caused by speculative generality.

Code smells

- **Refused Bequest**
  - When a subclass uses only some of the methods and properties inherited from its parents. Usually done for functionality reuse instead then modelling correct abstraction. **LSP** violation.
- **Comments**
  - Make effort to create code that expresses intent instead of comments. **KISS** violation.
- **Dead Code**
  - Code that has no references to it, commented code. Remember that "*Deleted code has no bugs and improves readability*"

Smelly Tic Tac Toe

We've created a very smelly implementation of TicTacToe. There are quite a few code smells in the implementation:

• Primitive obsession • Feature envy • Data class • Message chain • Long method • Comments	• Long parameter list • Shotgun surgery • Duplicated code • Large class • Divergent change • Data clump • Lazy class • Dead code

Start by identifying the smells and then slowly refactor the code. Remember to keep the tests passing at all times during the refactor. It's ok to revert back to a previous working state at any moment.