Bad Smells

There are only two types of issues in the codebase:

1. Some issues, like bugs, are so bad that they should be removed immediately. They cause the applications to behave incorrectly and usually lead to a loss in time or money for end-users.

2. The other type of issue is less subtle. These issues are called code smells. A code smell is a metaphoric term for a pattern in the application code that indicates a likely problem. It could be a symptom of a bad design or a sign of an impending problem. A typical example of a code smell is a duplicated code, a long method, or a long class.

Bad Smells are tangible and observable indications that there is something wrong with an application's underlying code that could eventually lead to serious failures and kill an application's performance.
Particularly "smelly" code could be inefficient, non-performant, complex, and difficult to change and maintain. While code smells may not always indicate a particularly serious problem, following them often leads to discoveries of decreased code quality, drains on application resources or even critical security vulnerabilities embedded within the application's code.
There are many situations that can cause code smells, such as improper dependencies between modules, an incorrect assignment of methods to classes, or needless duplication of code segments. Code that is particularly smelly can eventually cause profound performance problems and make business-critical applications difficult to maintain.
Code smells are structural characteristics of software that may indicate a code or design problem that makes software hard to evolve and maintain, and may trigger refactoring of code.
A code smell is a problem in source code that is not a bug or strictly technically incorrect, but a surface indication that usually corresponds to a deeper problem in the system. Code will still compile and work as expected, but instead, they indicate violation of design principles and weaknesses in design that may slow down development or increase the risk of bugs or failures in the future. Bad code smells can be an indicator of factors that contribute to technical debt.
Technical debt, like financial debt, compounds. A small amount of technical debt is fine - perhaps even inevitable - but it accumulates quickly and can end up hampering development velocity. As engineers and developers write more code to accommodate the smelly code in the code base, the rot in the code spreads. Eventually, Smelly code can easily become rotten code. Code rot is the process by which code slowly degrades over time.
A nice advantage of code smells is that they are detectable by even the most junior members on the team. Because they don't require extensive knowledge of business logic, or even advanced coding chops, anyone can spot them and contribute to improving the quality, performance, and stability of the source code.
Typical examples of code smells include the following:

Application level Smells:

Bad Smell	Description	Example	Refactoring

Duplicate Code	When the same or very similar code appears in multiple places, making maintenance difficult.	class Employee {
		public double calculateBonus(double salary) {
		// **Bonus calculation**
		return salary * 0.10; }
		}
		
		class Manager {
		public double calculateBonus(double salary) {
		// **Same logic duplicated**
		return salary * 0.10; }
		}
			Extract a common method in a parent class or utility class.
			
			class Person {
			public double calculateBonus(double salary) {
			// **Centralized logic**
			return salary * 0.10; }
			}
			
			class Employee extends Person { }
			class Manager extends Person { }

Bad Smell	Description	Example	Refactoring

Shotgun Surgery	A single modification requires updating multiple classes, making the system fragile.	class Order {
		public void completeOrder() {
		System.out.println("Order completed.");
		new EmailService().sendEmail("Order confirmation");
		}
		}
		
		class Payment {
		public void processPayment() {
		System.out.println("Payment processed.");
		new EmailService().sendEmail("Payment confirmation");
		}
		}
		
		class EmailService {
		public void sendEmail(String message) {
		System.out.println("Sending email: " + message);
		}
		}
			Use Dependency Injection or an Observer Pattern to decouple email logic.
			
			interface NotificationService {
			void sendNotification(String message); }
			
			class EmailService implements NotificationService {
			public void sendNotification(String message) {
			System.out.println("Sending email: " + message); }
			}
			
			class Order {
			private NotificationService notificationService;
			
			public Order(NotificationService notificationService) {
			this.notificationService = notificationService; }
			
			public void completeOrder() {
			System.out.println("Order completed.");
			
			notificationService.sendNotification("Order confirmation”); }
			}
			
			class Payment {
			private NotificationService notificationService;
			
			public Payment(NotificationService notificationService) {
			this.notificationService = notificationService;
			}
			
			public void processPayment() {
			System.out.println("Payment processed.");
			
			notificationService.sendNotification("Payment confirmation");
			}
			}

Bad Smell	Description	Example	Refactoring

Contrived Complexity	Using complex design patterns where a simpler uncomplicated design could be used.	abstract class Vehicle {
		    abstract void drive();
		}
		
		class Car extends Vehicle {
		    void drive() {
		            System.out.println("Driving a car”); }
		}
		
		class Bike extends Vehicle {
		    void drive() {
		            System.out.println("Riding a bike”); }
		}
		
		// **Unnecessary Factory Pattern**
		class VehicleFactory {
		        public static Vehicle createVehicle(String type) {
		        if (type.equalsIgnoreCase("car")) {
		            return new Car();
		        } else if (type.equalsIgnoreCase("bike")) {
		            return new Bike();
		        }
		        return null;
		    }
		}
		
		public class Main {
		    public static void main(String[] args) {
		            Vehicle car = VehicleFactory.createVehicle("car");
		        car.drive();
		
		            Vehicle bike = VehicleFactory.createVehicle("bike");
		        bike.drive();
		    }
		}
			Remove the factory and instantiate objects normally.
			
			public class Main {
			    public static void main(String[] args) {
			            Vehicle car = new Car(); // **Direct instantiation**
			        car.drive();
			
			        Vehicle bike = new Bike();
			        bike.drive();
			    }
			}

Method Level Smells:

Bad Smell	Description	Example	Refactoring
Long Method	A method does too much, making it difficult to read, understand, and maintain.	public void generateReport(List<Employee> employees) { System.out.println("Generating Employee Report..."); for (Employee emp : employees) { System.out.println("Name: " + emp.getName()); System.out.println("Department: " + emp.getDepartment()); System.out.println("Salary: $" + emp.getSalary()); double tax = emp.getSalary() * 0.2; double bonus = emp.getSalary() * 0.1; double netSalary = emp.getSalary() - tax + bonus; System.out.println("Tax: $" + tax); System.out.println("Bonus: $" + bonus); System.out.println("Net Salary: $" + netSalary); System.out.println("--------------------------------"); } }	**Extract Method** public void generateReport(List<Employee> employees) { System.out.println("Generating Employee Report..."); for (Employee emp : employees) { printEmployeeDetails(emp); } } private void printEmployeeDetails(Employee emp) { System.out.println("Name: " + emp.getName()); System.out.println("Department: " + emp.getDepartment()); System.out.println("Salary: $" + emp.getSalary()); double tax = calculateTax(emp.getSalary()); double bonus = calculateBonus(emp.getSalary()); double netSalary = emp.getSalary() - tax + bonus; System.out.println("Tax: $" + tax); System.out.println("Bonus: $" + bonus); System.out.println("Net Salary: $" + netSalary); System.out.println("--------------------------------"); } private double calculateTax(double salary) { return salary * 0.2; } private double calculateBonus(double salary) { return salary * 0.1; }

Bad Smell	Description	Example	Refactoring
Speculative Generality	Code is written for future use but is never actually used in the project.	public class Calculator { public int add(int a, int b) { return a + b; } public int multiply(int a, int b) { return a * b; } **// This method is never used in the actual project** public double power(int base, int exponent) { return Math.pow(base, exponent); } }	**Remove Unused Method** public class Calculator { public int add(int a, int b) { return a + b; } public int multiply(int a, int b) { return a * b; } }
Message Chains	One method calls another, which calls another, leading to deep nesting.	public String getEmployeeCity(Employee emp) { return emp.getDepartment().getManager().getAddress().getCity(); }	**Hide Delegation** public class Employee { private Department department; public String getManagerCity() { return department.getManagerCity(); } } public class Department { private Manager manager; public String getManagerCity() { return manager.getCity(); } }
Too Many Parameters	A method has too many parameters, making it difficult to use and refactor.	public void createUser(String firstName, String lastName, String email, String phone, String address, int age, String gender) { // Creating user... }	**Use Object Instead** public void createUser(User user) { // Creating user... } class User { private String firstName; private String lastName; private String email; private String phone; private String address; private int age; private String gender; // Constructor and Getters/Setters... }

Bad Smell	Description	Example	Refactoring

Oddball Solutions	Different parts of the codebase solve the same problem in different ways.	class Employee {
		public String getFullName() {
		return firstName + " " + lastName;
		}
		}
		
		class Manager {
		public String getFullName() {
		return lastName.toUpperCase() + ", " + firstName;
		}
		}
			Consistent Implementation
			
			abstract class Person {
			protected String firstName;
			protected String lastName;
			
			public String getFullName() {
			return firstName + " " + lastName;
			}
			}
			
			class Employee extends Person { }
			class Manager extends Person { }

God Line	A single line of code is too long, making it unreadable	System.out.println("Employee: " + employee.getName() + ", Age: " + employee.getAge() + ", Salary: $" + employee.getSalary() + ", Department: " + employee.getDepartment().getName());	**Split Into Multiple Lines** String employeeInfo = "Employee: " + employee.getName() + ", Age: " + employee.getAge() + ", Salary: $" + employee.getSalary() + ", Department: " + employee.getDepartment().getName(); System.out.println(employeeInfo);

Excessive Returner	A method returns unnecessary data instead of only what's needed.	public Employee getEmployeeDetails(int id) {
		return database.findEmployeeById(id); // **Returns the entire object, even if only name is needed**
		}
			Return Only Required Data
			
			public String getEmployeeName(int id) {
			return database.findEmployeeById(id).getName();
			}

Identifier Size	Variable or method names are either too short (unclear) or too long (verbose).	int a = 5; // **What does 'a' represent?**
		String userFirstNameAndLastNameInUpperCaseButOnlyIfAdmin = "John Doe"; // **Too verbose**
			Meaningful and Concise Identifiers
			
			int employeeCount = 5; // **Clearer**
			String adminFullName = "John Doe"; // **More concise**

<u>Class Level Smells:</u>

Bad Smell	Description	Example	Refactoring

Freeloader	A class has almost no functionality and is just taking up space.	class Logger {
		public void log(String message) {
		System.out.println("Log: " + message);
		}
		}
			Inline into Existing Class
			
			class Application {
			public void processRequest() {
			System.out.println("Log: Processing request...");
			}
			}

Feature Envy	A method accesses another class's data instead of using its own.	class SalaryCalculator {
		public double calculate(Employee emp) {
		return emp.getBaseSalary() + emp.getBonus();
		}
		}
			Move Method to Employee
			
			class Employee {
			public double calculateSalary() {
			return baseSalary + bonus;
			}
			}

Divergent Code	A single class changes for multiple unrelated reasons.	class Report {
		public void generatePDFReport() { /* PDF logic */ }
		public void generateExcelReport() { /* Excel logic */ }
		public void logReportAccess() { /* Logging logic */ }
		}
			Separate Responsibilities
			
			class PDFReportGenerator { public void generate() { /* PDF logic */ } }
			class ExcelReportGenerator { public void generate() { /* Excel logic */ } }
			class ReportLogger { public void log() { /* Logging logic */ } }

Data Clump	Several variables are always together and should be encapsulated.	class Order {
		private String street;
		private String city;
		private String zipCode;
		}
			Encapsulate in Object
			
			class Address {
			private String street;
			private String city;
			private String zipCode;
			}
			class Order {
			private Address address;
			}

Inappropriate Intimacy	A class directly accesses private fields of another class.	class EmployeeService {
		public void increaseSalary(Employee emp, double amount) {
		emp.salary += amount; // Accessing private field
		}
		}
			Use Getters/Setters
			
			class Employee {
			private double salary;
			public void increaseSalary(double amount) { salary += amount; }
			}

Bad Smell	Description	Example	Refactoring

Middle Man	A class does little more than delegate methods to another class	class OrderService {
		private OrderRepository repository;
		public void saveOrder(Order order) { repository.save(order); }
		}
			Remove Unnecessary Layer
			
			class Order {
			public void save() { repository.save(this); }
			}

Downcasting	Using explicit typecasting where it's unnecessary.	void processAnimal(Animal a) {
		if (a instanceof Dog) {
		Dog d = (Dog) a;
		d.bark();
		}
		}
			Polymorphism
			
			abstract class Animal {
			abstract void makeSound();
			}
			class Dog extends Animal {
			void makeSound() { System.out.println("Bark!"); }
			}

Parallel Inheritance Hierarchy	Every time you add a subclass, you must add another subclass elsewhere.	class Car { }
		class ElectricCar extends Car { }
		class CarEngine { }
		class ElectricCarEngine extends CarEngine { }
			Composition over Inheritance
			
			class Car {
			private Engine engine;
			}
			interface Engine { }
			class ElectricEngine implements Engine { }

Refused Bequest	A subclass inherits methods that it does not use.	class Animal {
		public void eat() { }
		public void fly() { }
		}
		class Dog extends Animal { }
			Use Interfaces
			
			interface CanEat { void eat(); }
			interface CanFly { void fly(); }
			class Dog implements CanEat { }

Cyclomatic Complexity	Too many conditional statements(Branches and Loops) make the code complex.	if (x > 10) {
		if (y > 5) {
		if (z < 3) {
		// Deeply nested logic
		}
		}
		}
			Extract Methods, Reduce Nesting
			
			if (isConditionMet(x, y, z)) {
			process();
			}

Smells can also be divided into the following groups:

<u>1) The Bloaters</u>

- Long Method
- Large Class
- Primitive Obsession
- Long Parameter List
- Data Clumps

Bloater smells represents something that grows and accumulates over time as a program evolves and eventually grows so large that it cannot be effectively handled.
Long Methods and Large Classes grow over time to have too many variables, such that it becomes hard to understand.
Primitive Obsession is actually more of a symptom that causes bloats than a bloat itself. When a Primitive Obsession exists,

there are no small classes for small entities (e.g. phone numbers). Thus, the functionality is added to some other class, which increases the class and method size in the software.
The same holds for Data Clumps. With Data Clumps there exists a set of primitives that always appear together (e.g. 3 integers for RGB colors). Since these data items are not encapsulated in a class this increases the sizes of methods and classes.

The Object- Orientation Abusers

- Switch Statements
- Temporary Field
- Refused Bequest
- Alternative Classes with Different Interfaces

The common denominator for the smells in the Object-Orientation Abuser category is that they represent cases where the solution does not fully exploit the possibilities of object-oriented design.
For example, a Switch Statement might be considered acceptable or even good design in procedural programming, but is something that should be avoided in object-oriented programming. The situation where switch statements or type codes are needed should be handled by creating subclasses. Parallel Inheritance Hierarchies and Refused Bequest smells lack proper inheritance design, which is one of the key elements in object-oriented programming. The Alternative Classes with Different Interfaces smell lacks a common interface for closely related classes, so it can also be considered a certain type of inheritance misuse.
The Temporary Field smell means a case in which a variable is in the class scope, when it should be in method scope. This violates the information hiding principle.

The Change Preventers

- Divergent Change
- Shotgun Surgery
- Parallel Inheritance Hierarchies

Change Preventers are smells that hinder changing or further developing the software.
These smells violate the rule suggested by Fowler and Beck which says that classes and possible changes should have a one-to-one relationship. For example, changes to the database only affect one class, while changes to calculation formulas only affect the other class.
An unspoken rule in programming is making code as decoupled as possible while remaining cohesive. Often programmers will find excessive coupling between classes - this group of smells comes from programmers' need to create entities in order to delegate everything.
The Divergent Change smell means that a single class needs to be modified by many different types of changes.
On the other hand, Shotgun Surgery smell is the opposite, many classes need to be modified when making a single change to a system (change several classes when changing database from one vendor to another)
Parallel Inheritance Hierarchies, means a duplicated class hierarchy. It was originally placed in OO- abusers, but, also fits with The Dispensables, since there is redundant logic that should be replaced.

The Dispensables

- Lazy class
- Data class
- Duplicate Code
- Dead Code,
- Speculative Generality

The common thing for the Dispensable smells is that they all represent something unnecessary that should be removed from the source code.
Programmers, many a times, include too much pointless information in the program, which is damaging for the health of the code. Not having these dispensables would improve readability, proficiency and make the code easier to maintain.
This group contains two types of smells (dispensable classes and dispensable code), however, they violate the same principle. If a class is not doing enough it needs to be removed or its responsibility needs to be increased. This is the case with the Lazy Class and the Data Class smells. Code that is not used or is redundant needs to be removed. This is the case with Duplicate Code, Speculative Generality and Dead Code smells.

The Couplers

- Feature Envy

- Inappropriate Intimacy
- Message Chains
- Middle Man

This group has four coupling-related smells.
One design principle that has been around for decades is low coupling.
This group has 3 smells that represent high coupling: The Feature Envy smell means a case where one method is too interested in other classes. Inappropriate Intimacy smell means that two classes are coupled tightly to each other.
Message Chains is a smell where class A needs data from class D. To access this data, class A needs to retrieve object C from object B, where A and B have a direct reference. When class A gets object C it then asks C to get object D. When class A finally has a reference to class D, A asks D for the data it needs. The problem here is that A becomes unnecessarily coupled to classes B, C, and D, when it only needs some piece of data from class D. The following example illustrates the message chain smell: ***A.getB().getC().getD().getTheNeededData()***
Middle Man smell, on the other hand, represent a problem that might be created when trying to avoid high coupling with constant delegation. Middle Man is a class that is doing simple delegation instead of really contributing to the application.

After Discovering a Code Smell

After checking the correctness of functionality of the application. It is important to review the code for smells. In case, there is a whiff of a code smell.
The first thing to do is to remember that the code smell only suggests that there is a deeper issue.
The second step is to fully understand the smell - What exactly is smelly about it? From there, exploring for its causes begin.
Ultimately, the programmers need to decide one of the following actions:

- **No action**: the code smell is detected, but programmers decide to do nothing about it. Mostly this is done when smelliness of the code is justified.
- **Discard the code smell:** There are two possible reasons for discarding the smell:
    1. It is impossible to refactor the existing code. Maybe the code is in a really bad place, and more tests need to be added around it first you refactoring it, or the segment of code is not a priority in current time.
    2. If the Code Smell detection was completely accurate. For example, for a really large method, upon further inspection, it is realised that it cannot be split into smaller methods as the statements in the existing method are highly coupled and belong together.
- **Refactor the code and remove the code smell:** Depending upon the application, refactoring for all the discovered smells should be done, until the smell is completely removed.

Whether a smell can and should be fixed or whether its smelliness is justified is a judgment call, so it's worth getting the opinions of other programmers and developers.

Refactoring:

Refactoring is the process of changing a software system in such a way that it does not alter the external behaviour of the software yet improves its internal structure and enhances some non-functional qualities, such as, simplicity, flexibility, understandability, and/or performance.
Refactoring removes code smells, but is much more than that, it is about ongoing maintenance of source code. Refactoring does not have tangible value because it does not alter the functionality or features or the code. However, continuous refactoring of code prevents what is known as “design decay”, cleans up and improves the quality of the code, and over time, makes code much more readable, maintainable and less prone to bugs. There are many types of refactoring and they can be applied either individually or in bulk in order to clean the code.
In Refactoring process, the code is divided into smaller sections according to the identified smells, then, a decision is made to either remove or replace them with a better series of code. After refactoring, tests are run to ensure things still work correctly. Sometimes this process has to be repeated until the smell is gone.

Some commonly used refactoring techniques are:

- **Extract Method:** To reduce the length of a method body, or move loops to a separate method.
  Steps:
    1. Create a new method and name it in a way that makes its purpose self-evident.

2. Copy the relevant code fragment to your new method, Delete the fragment
   from its old location and put a call for the new method there instead.
3. Find all variables used in this code fragment. If they are declared inside the fragment and not used outside of it, simply leave them unchanged – they will become local variables for the new method. However, If the variables are declared prior to the code that is extracted, these variables will be passed as parameters to the new method in order to use the values previously contained in them. Sometimes it is easier to get rid of these variables by resorting to Replace Temp with Query.
4. Check if a local variable changes in the extracted code in some way, this may
   mean that this changed value will be needed later in the main method. Double-check! And if this is indeed the case, return the value of this variable to the main method to keep everything functioning.

- **Move Method:** If a method clearly should be moved to another place.
  Steps:
  1. Verify all features used by the old method in its class. It may be a good idea to move them as well. As a rule, if a feature is used only by the method under consideration, move the feature to it. If the feature is used by other methods too, move these methods as well. Sometimes it is much easier to move a large number of methods than to set up relationships between them in different classes.
  2. Make sure that the method is not declared in superclasses and subclasses. If this is the case, refrain from moving the method or else implement a kind of polymorphism in the recipient class in order to ensure varying functionality of a method split up among donor classes.
  3. Declare the new method in the recipient class. Name the method that is more appropriate for it in the new class.
  4. Decide how to refer to the recipient class, through an existing field or method that returns an appropriate object, or, a new method or field to store the object of the recipient class.
  5. Check if the old method can be deleted entirely. If so, place a reference to the new method in all places that use the old one.

- **Extract Class:** If part of the behavior of the large class can be spun off into a separate component.
  Steps:
  1. Decide on how to split up the responsibilities of the class.
  2. Create a new class to contain the relevant functionality.
  3. Create a relationship between the old class and the new one. Optimally, this
     relationship is unidirectional; this allows reusing the second class without any issues. Nonetheless, if there is a need for two-way relationship, this can always be set up.
  4. Use Move Field and Move Method for each field and method that has to move to the new class. For methods, start with private ones in order to reduce the risk of making a large number of errors. Try to relocate just a little bit at a time and test the results after each move, in order to avoid a pileup of error-fixing at the very end.
  5. After moving of the functions, check the resulting classes. An old class with changed responsibilities may be renamed for increased clarity. Check again to see whether two-way class relationships can be removed, if any are present.
  6. Check the accessibility to the new class from the outside. The class can be hidden from the client entirely by making it private, managing it via the fields from the old class. Alternatively, can be made public by allowing the client to change values directly.

- **Extract Subclass:** If part of the behaviour of the large class can be implemented in different ways or is used in rare cases.
  Steps:
  1. Create a new subclass from the class of interest.
  2. If you need additional data to create objects from a subclass, create a constructor and add the necessary parameters to it. Do not forget to call the constructor's parent implementation.
  3. Find all calls to the constructor of the parent class. When the functionality of a subclass is necessary, replace the parent constructor with the subclass constructor.
  4. Move the necessary methods and fields from the parent class to the subclass. Do this via Push Down Method and Push Down Field. It is simpler to start by moving the methods first. This way, the fields remain accessible throughout the whole process: from the parent class prior to the move, and from the subclass itself after the move is complete.
  5. After the subclass is ready, find all the old fields that controlled the choice of functionality. Delete these fields by using polymorphism to replace all the operators in which the fields had been used.