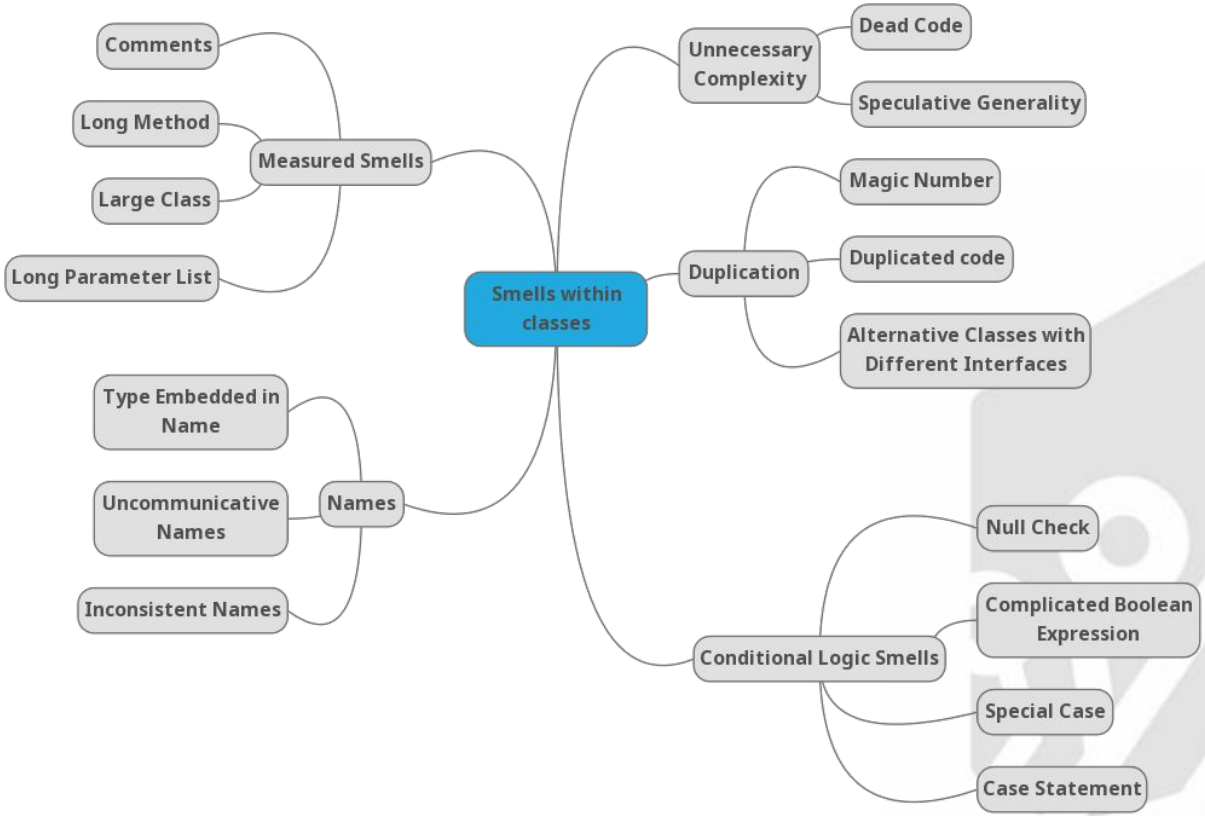
# Code Smells within classes

CODESAI

# Smells within classes



# Smells Within Classes

- **Measured Smells**
- Duplication Smells
- Conditional Logic smells
- Names Smells
- Unnecessary Complexity Smells

# Measured Smells

- They are common.
- The easiest to identify.
- Not necessarily the easiest to fix.

# Measured Smells

- Comments
- Long Method
- Large Class
- Long Parameter List

# Common Problems

- They hinder understanding.
- They may hinder reuse.
- They may hide other problems.

# Comments

- **Deodorant**, not real smell.
- **Earn the right to remove them**.
- **Removal** 💰:
  - Improves communication and may expose duplication.
- ⚠️ **Be careful, some comments are useful**
  - Explaining why.
  - Citing algorithms that are not obvious.

# Long Method (Bloater)

- Really **about number of responsibilities**, (a.k.a. Multi-Responsibility Method) => **Lack of cohesion**.
- Refactoring tips (**see Composed Method Pattern**):
  - Extract semantically meaningful methods (good naming is crucial). Get hints from the structure of the code.
- Heuristic to avoid this code smell: when you feel the need to comment something, write a method instead.

# Large Class (Bloater)

- Large number of fields, methods or lines.
- Really **about number of responsibilities**.
  - **Many responsibilities: lack of cohesion**.
- Refactoring tips:
  - **Long Methods,** address that **first**.
  - Then look for **cohesive subsets of methods and fields** and **extract new meaningful abstractions**, (good naming is crucial).

# Large Class & Long Method

- **Removal** 💰:
  - Improves communication, composability and reuse.
  - May expose duplication.
  - Often helps new abstractions to emerge.
  - Reduces size.

# Long Parameter List (Bloater)

- 3 or more (might be related to **Data Clump**).
- Prone to errors, difficult to understand.
- Refactoring tips:
  - Look for **Data Clumps,** (meaningful grouping), address that **first**.
  - Look for flag arguments (control parameters).
  - Only then consider the rest of related refactorings: *Replace Parameter with Method*, *Preserve Whole Object*, *Introduce Parameter Object*.

# Long Parameter List (Bloater)

- **Removal** 💰:
  - Improves communication, less error prone code and reduces size.
- ⚠️ Sometimes you avoid using *Preserve Whole Object* in order to avoid a dependency between two classes.
- ⚠️ Sometimes you avoid using *Introduce Parameter Object* because the parameters have no meaningful grouping.

# Smells Within Classes

- Measured Smells
- **Duplication Smells**
- Conditional Logic smells
- Names Smells
- Unnecessary Complexity Smells

# Duplication Smells

- **Duplication causes many problems**:
  - More code to maintain (*conceptual & physical burden*).
  - *Perceptual problems*:
    - Parts that vary are buried inside parts that stay the same.
    - Code variations often hide deeper similarities.
  - *Error prone*: tendency to fix or change things in some places but not in other ones.

# Duplication Smells

- It's a **root problem**.
  - Many other smells are special cases of duplication.

- Strive to make your code express each idea **"once and only once"**.

# Duplication Smells

- Magic Number
- Duplicated code
- Alternative Classes with Different Interfaces

# Magic Number

- A literal appears in the code, and has no meaning by itself (Magic Literal). Poor communication. Often a form of duplication.
- Refactoring tips:
  - *Replace Magic Literal*.
  - *Use enums to group* related constants.
- **Removal** 💰:
  - Improves communication and reduces duplication.
- ⚠️ Beware when using *Replace Magic Literal* in tests.

# Duplicated Code (Dispensable)

- **Easy form**: Fragments of code that look nearly identical.
- **Hard form**: Fragments of code have nearly identical effects (at any conceptual level).
- It **can be hidden by other smells**: many code smells are special forms of duplication.
- Refactoring tips: Read the catalog discussion to choose the appropriate refactoring for each kind of duplication.

# Duplicated Code (Dispensable)

- **Removal** 💰:
  - Reduces duplication and size.
  - Improves maintainability.
  - Can lead to better abstractions and more flexible code.
- ⚠️ Beware of overdoing **DRY**.
  - **DRY is about knowledge** not code. Excessive DRY may lead to coupling and/or code that is more complex that it should be.
- ⚠️ DRY vs explicit tests.

# Alternative classes with different interfaces (OO Abuser)

- Two classes seem to be doing the same thing but are using different method names.
- Usually, caused by poor communication in the team.
- Example

# Alternative classes with different interfaces (OO Abuser)

- Refactoring tip:

  **Harmonize the classes so you can remove one of them.**

  1. Make the methods similar.
  2. *Remove one class* if they are identical, or *Extract class* for common behavior.

- **Removal** 💰:
  - Reduces duplication and may improve communication.

# Smells Within Classes

- Measured Smells
- Duplication Smells
- **Conditional Logic smells**
- Names Smells
- Unnecessary Complexity Smells

# Conditional Logic smells

- **Hard to reason about**: multiple execution paths
- Tempting to add **special-case handling** rather than develop the general case.
- **Sometimes** used as **a weak substitute for object-oriented mechanisms**.

# Conditional Logic Smells

- Null Check
- Complicated Boolean Expression
- Special Case
- Case Statement

# Null Check

- Repeated conditionals checking for nulls, (see example).
- **Caused by using null as default**.
- **LSP** violation.
- Refactoring:
  - If there's a reasonable default, use it.
  - Otherwise, *Introduce Null Object*, to create a default object that you explicitly use, (**Null Object Pattern**).

# Null Check

- **Removal** 💰:
  - Reduces duplication, logic errors and exceptions.
- ⚠️ Ok if, for a given meaning of null, the null check is only in one place.
- ⚠️ Null Object's methods need to have **safe behavior**.
- ⚠️ When null can mean different things in different contexts, you need one different Null Object for each meaning.
- ⚠️ Similar problems with undefined, Optional, nullables,

# Complicated Boolean Expression

- Code has complex conditions.
- Refactoring:
  - Apply **DeMorgan's Law** if it helps.
  - Use **guard clauses** to eliminate certain conditions.
  - *Introduce Explaining Variable* to clarify, or, *Decompose Conditional* to pull each part into its own method.
- **Removal** 💰: Improves communication.

# Special Case

- **Complex if statements or checks for particular values before doing work**, especially comparisons to constants or enumerations (example).
- Refactoring:
  - *Replace Conditional with Polymorphism*.
- **Removal** 💰:
  - Improves communication. May expose duplication.
- ⚠️ Trade-off: **simplicity vs flexibility**.

# Case Statement (OO Abuser)

- Code uses **switch statement or several if statements comparing on a type field**, **constants with the same meaning** or **checking the type of an object to decide what to do** (see example).
- aka **Simulated Inheritance** or **Switch Statements**.
- Refactoring tips:
  - **If** the **switch** is **hidden** behind **nested conditions** and/or **complicated boolean expressions**, first make it emerge
  - Then replace it with polymorphism.

# Case Statement (OO Abuser)

- **Removal** 💰:
  - Improves communication and flexibility . Removes duplication. Less error prone. **OCP**.
- ⚠️ Fowler: *“bad only when duplicated”* (Repeated Switches).
- ⚠️ **Only in one place might be ok**. Common at boundaries where **data** is **converted into objects (factories, ...)**
- ⚠️ Trade-off: **simplicity vs flexibility**

# Smells Within Classes

## Gilded Rose example

# Smells Within Classes

- Measured Smells
- Duplication Smells
- Conditional Logic smells
- **Names Smells**
- Unnecessary Complexity Smells

# Names Smells

- Creating good mental models is a key challenge in developing software
- Tools that help:
  - Project Glossaries
  - Ubiquitous Language
  - XP-style Metaphors

# Names are important

- They provide a vocabulary for discussing the domain.
- They communicate intent.
- They support subtle expectations about how the system works.
- They support each other in **a system of names**.

# Names Smells

- Type Embedded in Name
- Uncommunicative Name
- Inconsistent Name

# Type Embedded in Name

CustomerImplementation

CustomerImpl

updateItemSellin(String name, int sellin, int quality);

updateItemQuality(String name, int sellin, int quality);

IDisplay

DisplayInterface

arru8NumberList

AddDTO

szName strName

TruncatedNewtonMinimizationStrategy

ConjugatedGradientMinimizationStrategy

- They might be unnecessary.
- They might have been added to improve communication.
- They may represent duplication or missing abstractions.

# Type Embedded in Name

- **Removal** 💰:
  - Improve communication and may expose duplication.
- ⚠️ Common affixes in different methods may signal a missing abstraction. If so, introduce it.
- ⚠️ Be careful with overloading methods.
- ⚠️ Follow **conventions**.

# Uncommunicative Name

```
void getData();    update1(Item item);    updateOld(Item item);
                                          updateNew(Item item);
bar()              update2(Item item);
Command cm;        let cra;    let foo;    int obelix;    f()
bogus()    boolean noBuenoPeroNoMalo;    let g;    let n;
int name;          Minimization cgm;    String s;    for (Car cars : car)
```

- They **don't communicate intent** well enough.
- **Removal** 💰:
  - Improve communication.

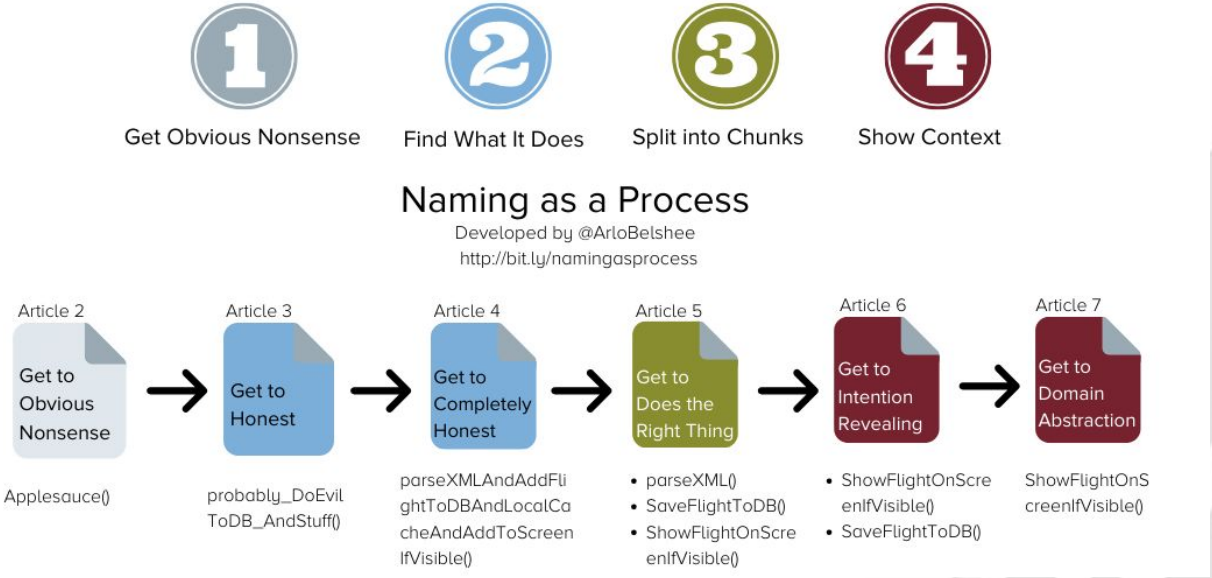
# Uncommunicative Name

- ⚠️ Don’t overdo it (e.g., i, j, k).
- ⚠️ Follow **conventions**.
- ⚠️ Beware of scope and context.
  - e.g.: (s) => s.toUpperCase(); ✅
  - e.g.: function isEven(n) { return n%2 == 0;  } ✅

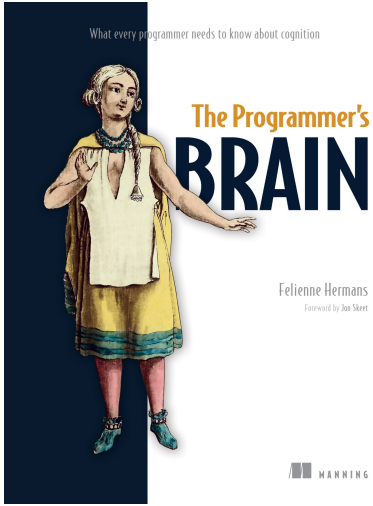
# Inconsistent Names

- One name is used in one place, and a different one is used for the same thing somewhere else.
- Refactoring:
  - Pick the **best name**, and **use it everywhere** it makes sense.
- **Removal** 💰:
  - Improve communication and may expose duplication.
- ⚠️ It’s ok to use the same name in different contexts.

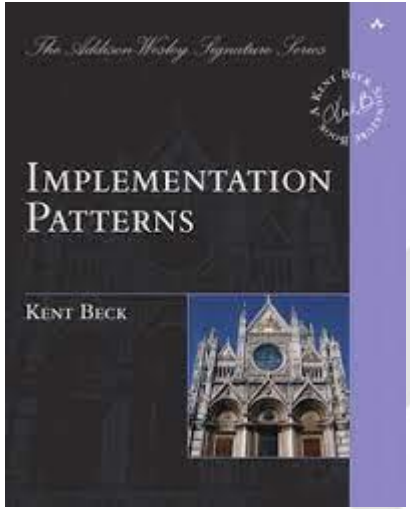
# Naming as a process



# Naming tips




Chapter 8 on getting better at naming, and 9 commenting linguistic antipatterns and cognitive load.




Great advice on naming classes and interfaces.

🎥 Felienne Hermans: How patterns in variable names can make code easier to read

# Smells Within Classes

- Measured Smells
- Duplication Smells
- Conditional Logic smells
- Names Smells
- **Unnecessary Complexity Smells**

# Unnecessary Complexity Smells

- YAGNI violation.
- Sometimes code gets complicated for historical reasons, but it no longer needs the complexity
- Sometimes it’s because the design has been overgeneralized just in case.

# Unnecessary Complexity Smells

- Dead Code
- Speculative Generality

# Dead Code (Dispensable)

- Some part of the code is **not used anywhere** (perhaps other than the tests).
- **Commented code is dead code!**
- Refactor tip: Lean on your tools to detect it.
- **Removal** 💰:
  - Reduces size, improves communication and simplicity.
- ⚠️ Beware of **published interfaces**.

# Speculative Generality (Dispensable)

- Code is more complicated that it has to be for the currently implemented behavior. Apply **YAGNI** to avoid it!
- Refactoring tip:
  - To remove design patterns, see Refactoring to Patterns.
- **Removal** 💰:
  - Reduces size, improves communication and simplicity.
- ⚠️ Beware if you are creating a framework.