Names: Aguirre Almache, Gabriel Sebastian
Ango Alquinga, Johny Javier
Arias Campos, Marco Antonio
Aulestia Hermosa, Ariel Mateo
Barrera Taco, Kevin Alejandro
Beltran Gavilanes, Jennifer Alejandra
Cadena Monar, Fernanda Maribel
Cadena Roman, Benjamin Abel

Team 1: Design and Implementation Pitfalls

- Pitfalls are common developer mistakes that lead to lower quality software and may even cause a project to fail.
- For each pitfall, it would be useful to know
    - Problem description: What it is?
    - Consequences: What harm or risks do it introduce into a project?
    - Causes: What leads to the pitfall?
    - Avoidance: How can a project avoid it?
    - Recognition: How to recognize that a project as succumb to this pitfall?
    - Extrication: How can a project get out of this pitfall?
    - When documented with this knowledge, Pitfalls are Anti-Patterns
    - Some authors call implementation-oriented pitfalls “Code Sme

List of Interesting Pitfalls

- **Uncommunicative Names**

❖ Causes

- Old habits (e.g., always using the same variable names for things like iterators)
- Incomplete or lazy thinking about a component
- A misguide attempt to achieve job-security

❖ Consequences:

- Poor understandability and readability -> lower maintainability and reuse

- **Inconsistent Names**

❖ Causes:

- Old habits (e.g., always using given variable names for things like iterators or return values)
- Incomplete or lazy thinking about a component

❖ Consequences:

- Poor understandability and readability à lower maintainability and reuse

- **Types Embedded in Names**

❖ Causes:

- Misguided efforts to provide more information in the code
- Poor development environments that don't provide type-lookup tools

❖ Consequences:

- Poor understandability and readability à lower maintainability and reuse
- Can break encapsulation, because it exposes the implementation type

- **Long Methods**

❖ Causes:

- Insufficient localization of decision decisions
- Lack of attention to the defining purpose of a method or to cohesiveness of its functionality
- Evolution without refactoring

❖ Consequences:

- Lower maintainability
- Lower understandability
- Harder to test
- Accidental complexity

- **Duplicate Code**

❖ Causes:

- Insufficient thought put into the design
- Lead of understanding of existing code
- Evolution without refactoring
- Multiple programmers working on the same system

❖ Consequences:

- Scattering of decision designs across multiple components
- Lower maintainability
    - Harder to debug
    - Harder fix an error in the scattered logic
- Accidental complexity

```
extern int arreglo1[];
extern int arreglo2[];

int suma1 = 0;
int suma2 = 0;
float promedio1 = 0.0;
float promedio2 = 0.0;

for (int i = 0; i < 4; i++)
{
   suma1 += arreglo1[i];
}
promedio1 = suma1 / 4;

for (int i = 0; i < 4; i++)
{
   suma2 += arreglo2[i];
}
promedio2 = suma2 / 4;
```

- **Long Message Chains**

❖ Causes:

- Inappropriate use of the decorator pattern or abuse of any recursive composition relationship

❖ Consequences:

- Poor performance
- Accidental complexity in the runtime flow of control

- **Class Explosion**

❖ Causes:

- Poor OO design
- Using inheritance for reuse when aggregation would have been better

❖ Consequences:

- Lower maintainability
- Accidental complexity
- Lower reusability
- Lower flexibility

- **Large Message Chains**

❖ Causes:

- Poor localization of design decision or modularization
- Evolution

❖ Consequences:

- The class become hard to understand, maintain, and reuse

- **Large Classes**

❖ Causes:

- Poor localization of design decision or modularization
- Evolution

❖ Consequences:

- The class become hard to understand, maintain, and reuse

- **Conditional Complexity**

❖ Causes:

- Poorly thought out behaviors or missing opportunity for generalization with respect to those behaviors
- The behavior of the object depends on modes of operation or “states”

- Evolution

❖ Consequences:

- The method become hard to understand, test, maintain, and reuse

- **Oddball Solution**

❖ Causes:

- Poorly generalization and localization of design decisions. There should only be one way of solving the same problem in your code.
- Missed opportunities to reuse existing components, perhaps through an adapter if the interface is not exactly what is needed

❖ Consequences:

- The system become hard to test, maintain, and reuse

- **Redundant or meaningless comments**

❖ Causes:

Lack of understanding or skills related writing useful comments

❖ Consequences:

- The code because hard to read
- There is a possibility that the comment is not update when the code is change, leading to confusion

- **Dead Code**

❖ Causes:

- Refactor of the code to improve the quality of the implementation.
- Improves the design to improve its quality, which in turn causes changes to the code.
- Change to requirements, which in turn cause changes to the design, and then the code.

❖ Consequences:

- Lower understandability
- Lower maintainability • Possible security risks

```
int foo (int X, int Y) {
    int Z = X/Y;
    return X*Y;
}
```

- **Speculative Generality**

❖ Causes:

- Developers get into habits and create certain kinds of classes or methods, just because they have always done so in the pass
- Developers consider where the system my change (which is good), but over react by implementing unnecessarily generalizations
- Developers over estimate the scope of the system – don't build a mansion if the customer only wants (and is paying for) a cottage

❖ Consequences:

- Lower understandability
- Lower maintainability

- **Temporary Field**

❖ Causes:

- Depending on the developers the backgrounds (first languages), they made be in the habit of defining all their variables at the top of their classes or methods, instead of in the context they are needed
- Not thinking about scope or coupling
- Not thinking from an object-oriented perspective

❖ Consequences:

- Lower understandability
- Lower maintainability
- Lower extensibility

- **Refused Bequest**

❖ Causes:

- Trying to reuse something via inheritance, without thinking about whether that fits from a conceptual modeling perspective

❖ Consequences:

- Lower maintainability
- Lower reusability
- Lower extensibility

- **Inappropriate Intimacy**

❖ Causes:

- Poor localization of design decisions
- Poor encapsulation

❖ Consequences:

- Lower maintainability
- Lower reusability
- Lower extensibility

- **Feature Envy**

❖ Causes:

- Poor localization of design decisions
- Poor encapsulation
- Evolution where properties move from one class to another, but closely related behaviors didn't follow

❖ Consequences:

- Lower maintainability
- Lower reusability
- Lower extensibility

Quiz

1. **Are common developer mistakes that lead to lower quality software and may even**
    a. Consistent names
    b. Short methods

c. Long Methods

2. What do some authors call implementation-oriented errors?

a. Pitfalls
b. Code Smells
c. Long Methods

3. What is the trap that leads to lower quality software?

a. Communicative names
b. Short methods
c. Dead code

4. Choose the correct option for one of the uncommunicative names

a. A class is trying to do too much and is “bloated”
b. A long method that lack of cohesion
c. A method with large conditional logic blocks.

5. What are the consequences of the class explosion?

a. Less reuse
b. Greater rendering
c. Greater flexibility