Design and Implementation Pitfalls

Object-oriented Programming

Pitfalls

- Pitfalls[1] 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 Smells”

[1] Webster, B., *Pitfalls of Object Oriented Development*, M&T Books, 1995

List of Interesting Pitfalls

- Uncommunicative Names
- Inconsistent Names
- Types Embedded in Names
- Long Methods
- Duplicate Code
- Long Message Chains
- Class Explosion
- Large Message Chains
- Large Classes
- Conditional Complexity
- Oddball Solution
- Redundant or meaningless comments
- Dead Code
- Speculative Generality
- Temporary Field
- Refused Bequest
- Inappropriate Intimacy
- Feature Envy

Uncommunicative Names

- Consider everything in a design and implementation that is referenced with a name or label (i.e., identifier)
  - Class
  - Instance of a class (object or variable which holds and object)
  - Object Properties
    - Data members or getter/setter properties
    - Operations and their parameters
  - Relations
  - Packages or Namespaces
- Identifiers should communicate exactly what the component represents: its role, responsibilities, intent, and scope

Uncommunicative Names

- Problem Description:
  - the design or implementation uses identifiers that don't accurately communicate what the component represents
  - This problem exists in multiple places throughout the design or code
- Consequences:
  - Poor understandability and readability → lower maintainability and reuse
- 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

Uncommunicative Names

- Avoidance:
  - Thoughtful design and lexicon
  - Ensuring that an implementation adheres to a design and the chosen lexicon
  - Willingness to refactor (i.e., rename) when an identifier is no longer accurate
  - Follow naming conventions, e.g.
    - Nouns or noun phrases for classes that represent an instance of the class
    - Verb or verb phrases for operations
- Recognition:
  - Design / Code walkthroughs and inspects
  - Consider each identifier in the context of the component's meaning
- Extrication:
  - Refactor names (identifiers)

Inconsistent Names

- Problem Description:
  - The design does not follow any logical or standard set of terminology or the implementation does not follow the design.
    - For example, if a component has an Open() method to enter a state of “opened”, then Close() method would be a consistent choice for a method that causes the component to leave the “opened” state.
    - If there are five different components that can all have “open” methods, those methods should about be called Open() or use the word open in a verb phrase
- Consequences:
  - Poor understandability and readability → lower maintainability and reuse
- Causes:
  - Old habits (e.g., always using given variable names for things like iterators or return values)
  - Incomplete or lazy thinking about a component

Inconsistent Names

- Avoidance:
  - Thoughtful design and lexicon
  - Ensuring that an implementation adheres to a design
  - Willingness to refactor
  - Follow naming conventions and standard terminology
- Recognition:
  - Design / Code walkthroughs and inspects
  - Consider each name or label in the context of the component's intended role and scope
- Extrication:
  - Refactor names

Embedded Types in Names

- Problem Description:
  - The identifiers in the implementation include type information, e.g. firstNameString instead of just firstName for a first name property
- Consequences:
  - Poor understandability and readability → lower maintainability and reuse
  - Can break encapsulation, because it exposes the implementation type
- Causes:
  - Misguided efforts to provide more information in the code
  - Poor development environments that don't provide type-lookup tools

Embedded Types in Names

- Avoidance:
  - Don’t include type names in identifiers
- Recognition:
  - Design / Code walkthroughs and inspects
- Extrication:
  - Refactor names

Long Methods

- Problem Description:
  - A long method that lacks of cohesion (a single defining purpose)
- Consequences:
  - Lower maintainability
  - Lower understandability
  - Harder to test
  - Accidental complexity
- 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

Long Methods

- Avoidance:
  - Localization of design decisions
  - Good modularization that maximizes cohesion, with increasing coupling
  - Refactoring as code evolves
- Recognition:
  - Design / Code walkthroughs and inspects
- Extrication:
  - Extract method
  - Extract class

Duplicate Code

- Problem Description:
  - The same algorithm is implemented in multiple places in the code, for similar purpose and the same basic context
- Consequences:
  - Scattering of decision designs across multiple components
  - Lower maintainability
    - Harder to debug
    - Harder fix an error in the scattered logic
  - Accidental complexity
- Causes:
  - Insufficient thought put into the design
  - Lead of understanding of existing code
  - Evolution without refactoring
  - Multiple programmers working on the same system

Duplicate Code

- Avoidance:
  - Localization of design decisions
  - Refactoring as needed when new features are added, or changes are made to the system
- Recognition:
  - Design / Code walkthroughs and inspects
- Extrication:
  - Extract superclass or extract class
  - Extract method

Class Explosion

- Problem Description:
  - There are lots of subclasses of a base class that do nearly the same thing
- Consequences:
  - Lower maintainability
  - Accidental complexity
  - Lower reusability
  - Lower flexibility
- Causes:
  - Poor OO design
  - Using inheritance for reuse when aggregation would have been better

Class Explosion

- Avoidance:
  - Prefer aggregation over inheritance, also delegation over inheritance
- Recognition:
  - Design / Code walkthroughs and inspects
- Extrication:
  - Refactor towards the decorator or strategy patterns

Long Message Chains

- Problem Description:
  - An object has to delegate a method call to a contained object, which in turn has to delegate to method call to an object that it contains, and so on. This can occur when the decorator pattern or a composition relationship is recursive and misused
- Consequences:
  - Poor performance
  - Accidental complexity in the runtime flow of control
- Causes:
  - Inappropriate use of the decorator pattern or abuse of any recursive composition relationship

Long Message Chains

- Avoidance:
  - Think carefully about how decorators or recursive compositions will be used and whether long message chains will occur frequently
- Recognition:
  - Design / Code walkthroughs and inspects
  - Runtime performance test and benchmarks
- Extrication:
  - Refactor towards strategies, template methods, or other design patterns that will not result in long message chains
  - Constrain the depth of composition for decorators or recursive composition relationships

Large Classes

- Problem Description:
  - A class is trying to do too much and is “bloated”
  - Although a class may start out small and cohesion, over time features and functionality get added that lessens its cohesion
- Consequences:
  - The class becomes hard to understand, maintain, and reuse
- Causes:
  - Poor localization of design decision or modularization
  - Evolution

Large Classes

- Avoidance:
  - Thoughtful OO design where particular attention is given to where design decisions are localized and encapsulated
- Recognition:
  - A class has become big, with disjoint collections of attributes or methods
- Extrication:
  - Refactor with Extract Class, Extract Subclass, or Extract Interface methods
  - Consider turning the original class into a façade that manages coordinates operations among the smaller, extracting objects
  - Other patterns, such as Strategy, Observer, and Template Method might also help

Conditional Complexity

- Problem Description:
  - A method with large conditional logic blocks, especially if number of conditionals or size of the blocks tend to grow larger or change over time
- Consequences:
  - The method becomes hard to understand, test, maintain, and reuse
- 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

Conditional Complexity

- Avoidance:
  - Thoughtful OO design with respect to object behavior
- Recognition:
  - A method has lots of conditions (if-then-else, switches, etc.)
- Extrication:
  - Refactor using a decorator, strategy, template method, or state patterns
  - Refactor – Extract method

Oddball Solution

- Problem Description:
  - There are multiple solutions to similar problems in the code, i.e., nearly duplicate code
- Consequences:
  - The system becomes hard to test, maintain, and reuse
- 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

Oddball Solution

- Avoidance:
  - Thoughtful OO design with good “Object-class Congruency”
- Recognition:
  - Multiple instances of very similar code
- Extrication:
  - Refactor using a class extraction, method extraction, or even an adapter
  - Refactor towards template methods, strategy, state, or decorator

Redundant or Meaningless Comments

- Problem Description:
  - Comments in the code add no value; they simply restate the obvious
- Consequences:
  - The code is hard to read
  - There is a possibility that the comment is not updated when the code is change, leading to confusion
- Causes:
  - Lack of understanding or skills related to writing useful comments

Redundant or Meaningless Comments

- Avoidance:
  - Practice writing good comments that describe design decisions
  - Critically review comments
- Recognition:
  - Code includes comments that don’t add value
- Extrication:
  - Remove worthless comments
  - Add meaningful comments, particularly those that generate documentation or tool tips

Dead Code

- Problem Description:
  - A variable, statement, parameter, field, method, or class is no longer used
- Consequences:
  - Lower understandability
  - Lower maintainability
  - Possible security risks
- Causes:
  - Refactor of the code to improve the quality of the implementation
  - Improves to the design to improve its quality, which in turn cause changes to the code
  - Change to requirements, which in turn cause changes to the design, and then the code

Dead Code

- Avoidance:
  - Use a good IDE that tell you when something is “dead”
  - Code walkthroughs
- Recognition:
  - Same
- Extrication:
  - Remove dead code
  - Make comments about design decision
  - Use a version control system, like Git
  - Keep design updated

Speculative Generality

- Problem Description:
  - Over designing or programming based on speculations about what might be need
- Consequences:
  - Lower understandability
  - Lower maintainability
- 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

Speculative Generality

- Avoidance:
  - Do a meaningful analysis of the problem domain and review with stake holders
  - Prioritize features
  - Allow for change in the right places, but don’t over generalize
- Recognition:
  - Watch out for deep generalization/specialization hierarchies and classes that don’t serve a purpose
- Extrication:
  - Remove unnecessary abstracts
  - Collapse class hierarchies where possible

Temporary Field

- Problem Description:
  - Temporary fields get their values only in certain situations and are otherwise need useless.
- Consequences:
  - Lower understandability
  - Lower maintainability
  - Lower extensibility
- 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

Temporary Field

- Avoidance:
  - Also define a variable in the closest scope
    - Don’t define a variable at the top of method if it is only needed inside a loop or conditional block
    - Don’t define an object attribute if it is only needed in a single method
    - Don’t define a static class attribute if it is only needed within the context of an object
  - Prioritize features
  - Allow for change in the right places, but don’t over generalize
- Recognition:
  - Variables that often are unused or null
  - Unnecessary coupling
- Extrication:
  - Rethink abstractions and encapsulations
  - Encapsulate data structures with the operations that effect them
  - Think from an object-oriented perspective

Refused Bequest

- Problem Description:
  - A derived class is really not a specialization (subset) of the base class
  - A derived class only need some of the methods or properties of the base class
- Consequences:
  - Lower maintainability
  - Lower reusability
  - Lower extensibility
- Causes:
  - Trying to reuse something via inheritance, without thinking about whether that fits from a conceptual modeling perspective

Refused Bequest

- Avoidance:
  - Do a meaningful OO analysis of the problem or the application domain, follow by a meaningful OO design of the desired solution
  - Don’t rely only on inheritance for reuse – consider aggregation and generics
- Recognition:
  - A derive class has inherited methods or properties that don’t make sense in the context of that class
- Extrication:
  - Consider use the adapter pattern inside of inheritance
  - Consider refactor the class hierarchy so there is better object-class congruency

Inappropriate Intimacy

- Problem Description:
  - One class uses the internal fields and methods of another class
- Consequences:
  - Lower maintainability
  - Lower reusability
  - Lower extensibility
- Causes:
  - Poor localization of design decisions
  - Poor encapsulation

Inappropriate Intimacy

- Avoidance:
  - Localize behaviors close to the data they operate on
  - Use the tighten possible encapsulation for properties and methods
    - Things that should be private, declare them private
    - Same for protected and internal
- Recognition:
  - Unnecessary coupling
- Extrication:
  - Refactor using Move Method, Move Field, Extract Class, or Hide Delegate refactoring techniques
  - Redo design with better localization of design decisions and encapsulation

Feature Envy

- Problem Description:
  - A method accesses the data of another object more than its own data.
- Consequences:
  - Lower maintainability
  - Lower reusability
  - Lower extensibility
- Causes:
  - Poor localization of design decisions
  - Poor encapsulation
  - Evolution where properties move from one class to another, but closely related behaviors didn’t follow

Feature Envy

- Avoidance:
  - Localize behaviors close to the data they operate on
  - Use the tighten possible encapsulation for properties and methods
    - Things that should be private, declare them private
    - Same for protected and internal
  - Note: in some cases, like when encapsulating dynamically selected algorithm (i.e., the Strategy Pattern), you want the context information in a separate class from the algorithm
- Recognition:
  - Unnecessary coupling
- Extrication:
  - Refactor using Move Method, Move Field refactoring techniques
  - Redo design with better localization of design decisions and encapsulation