Maintainable code

Concepts and principles

pb@retrorabbit.co.za | Jan 2023

Background

Maintenance in industry code

Industry code

- Long-lived
- Multiple authors
- Parts are poorly written
- Frequently changed
- Not aligned with a single set of standards
- Maintenance is often most of the job!

Ease of changing code

- Easy/difficult to reason about
  - What does it do?
  - How does it work?
  - What is its intended purpose?
- Complexity
- Coupling

Overview

What we'll cover

- Naming and comments
- Pure functions
- Immutability
- Handling invalid state
- Declarative style
- Modularisation and indirection
- SOLID principles for OOO

Outcomes

- Be informed
- Apply these concepts and principles
  - In new code
  - When refactoring for maintainability
  - In reviews
- Further learning

Naming and comments

Tips for naming

- Consistency >>>
- Don’t abbreviate
- No Hungarian notation please
- Use names to clarify ambiguity
- Avoid “Base”
- Avoid “Utils” or “Helper”

Possible naming standards

- Use appropriate parts of speech
  - Verbs for functions
  - Nouns for variables and classes; gerunds for action-focused classes
- Use “is” for boolean values
- Use “result” for accumulators

Comments - yay or nay?

Possible guideline

- Avoid comments when code is self-documenting
- Use comments when they help explain **why**, rather than **what**

Pure functions

A pure function has no side-effects

Functions should either return a value, or have a side effect, but not both

Benefits of pure functions

- Easy to reason about
- Easy to combine (compose)
- Good for unit testing
- Good for parallelisation

Immutability

An immutable object does not change its value after creation

Benefits of immutability

- Easy to reason about
- Thread safe
- Testable

Benefits of pure functions

- Easy to reason about
- Easy to combine (compose)
- Good for unit testing
- Good for parallelisation

Managing invalid state

Examples of invalid state

- Negative numbers where they don’t make sense
- Malformed URLs
- Closed sets represented by numbers, when numbers go out of range
- Value/error pair with both or neither
- Unexpected empty, zero, or null value
- Unit mismatch - seconds vs milliseconds
- Mixing values from different conceptual spaces - eg user IDs as post IDs

Make invalid state unrepresentable

Leveraging type systems

- Choose a strict type
  - E.g. use Uint > Int for age
- Create special types
- Use enums for closed sets of values
- Use non-interchangeable types for similar but non-interchangeable values

In addition
(Or if you don’t have a good type system)

- Use linters
- Write comprehensive tests
- Use CI
- Manual testing :(

Declarative coding style

What > how

Benefits of declarative coding

- Easy to reason about
- Less data mutability
- Less code, less bugs

Modularisation and indirection

Large codebases and complexity

As a codebase grows, its complexity increases.

A monolithic (single-module) codebase develops internal coupling.

The standard solution is to sub-divide the codebase into decoupled modules.

Each module has a high-level public interface that other modules can reference, and lower-level implementation details.

Aim for loose coupling among modules. (How?)

Indirection:
Referring to things indirectly

“We can solve any problem by introducing an extra level of indirection”

(except for the problem of too many levels of indirection)

David Wheeler

SOLID principles

SOLID principles

As per Robert “Uncle Bob” Martin

- Single-responsibility principle
- Open–closed principle
- Liskov substitution principle
- Interface segregation principle
- Dependency inversion principle

Single responsibility principle

There should never be more than one reason for a class to change

Benefits of Single responsibility principle

- Contain (limit) changes
- Avoid conflicts
- Easy to reason about

Open-closed principle

Classes should be open for extension, but closed for modification

(Or: we should be able to add new behaviours to code without having to change existing behaviours)

Benefits of open-closed principle

- Avoids the risk of changing existing code

Downsides?

- Business requirements are often to change existing behaviour, not add new behaviour
- Old code increases complexity

How to implement this

If code wasn’t originally written with open/closed-ness in mind, we may need to start off by refactoring.

Standard approach:

Make the “class” an interface.

Add new behaviour to a new concrete implementation.

Some languages also support extensions on existing types.

Liskov substitution principle

Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.

Or:
Subclasses should be substitutable for their base classes. Wherever you can use an instance of a base class, you must also be able to use an instance of a subclass instead, without breaking anything.

Also see: Design by contract

Interface segregation principle

Clients should not be forced to depend upon interfaces that they do not use

Or:
Prefer multiple small interfaces over single large interfaces

Benefits of interface segregation

- Classes don’t need to implement irrelevant parts of interfaces
- When depending on interfaces, you have fine-grained control over what functionality you want to use

Dependency inversion principle

Depend upon abstractions, not concretions

Or:
High-level modules should not import anything from low-level modules. Both should depend on abstractions (e.g., interfaces).

(The open-closed principle states the *goal* of OO architecture. Dependency inversion states the principle *mechanism.)*

Recap