Good Enough Design

CODESAI

Code Smells & Design

- It's not enough to detect that your code has design problems.

- We also need to know what good design looks like

How does good design look like?

The only constant in software is change

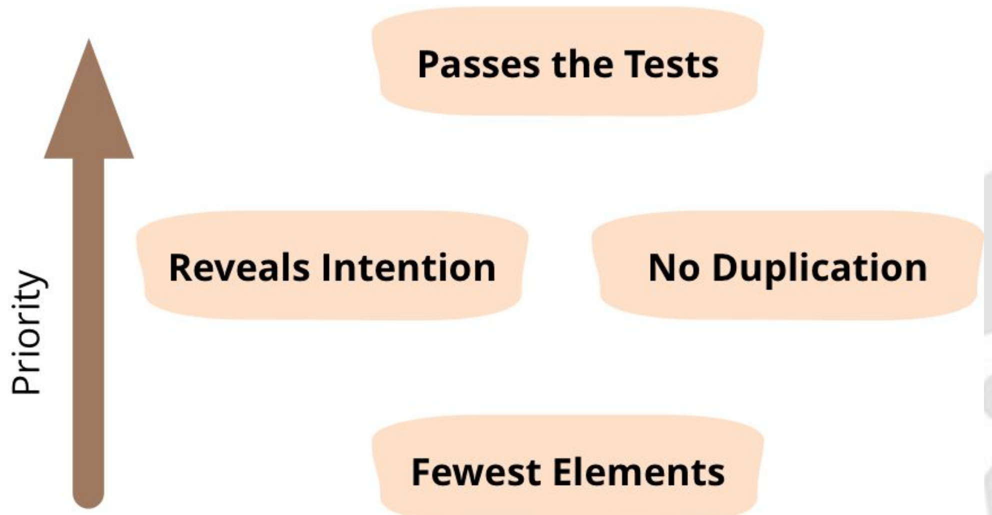
Any design imposes constraints on how we can change the code

A good design must support evolution

Tests are required to change a code base with confidence at a sustainable pace

A good design must be testable

Rules of Simple Design



Beck Design Rules, Martin Fowler

Design Principles

from Implementation Patterns

- Local Consequences
- Minimize Repetition
- Logic and Data Together
- Symmetry
- Rate of Change (Temporal Symmetry)

DRY

- Code duplication is not the issue. It’s about **knowledge** (example)
- There is a relation between duplicate code and **Rate of Change** principle
- *Duplication smells* are a violation of this principle, but there are many other smells related with duplication (*Null Check*, *Special Case*, *Duplicate Switch*…)

YAGNI

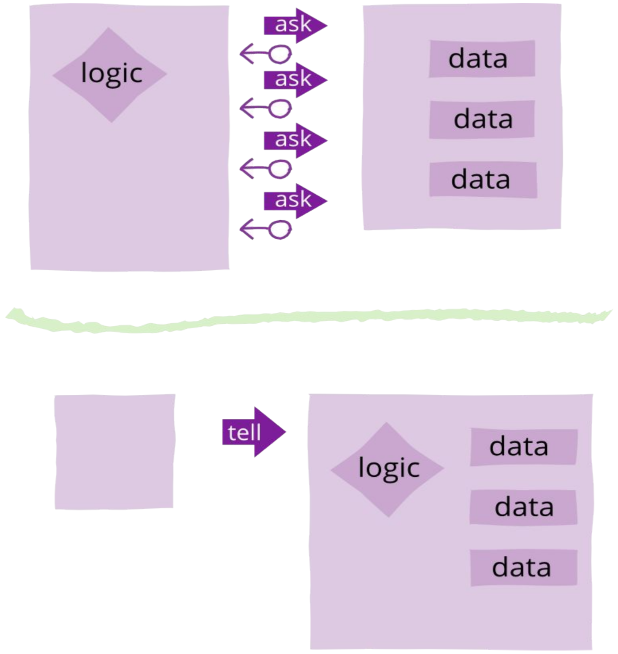
- You save time, because you **avoid writing code** that you turn out not to need.
- When you change code you face only the **current complexity**.
- *Speculative Generality smell* is a violation of this principle.

Law of Demeter

- **Principle of least knowledge** or don’t talk to strangers
- Decrease the coupling in your codebase.
- Related to **Local Consequences Principle**.
- *Message Chains smell* is a direct violation of this principle. *Feature Envy smell* is often present.

Tell, Don't Ask

- A pretty good heuristic for **requests that change state**.
- It might be useful to avoid some violations of the **Law of Demeter**.
- Some smells like *Message Chains* and *Feature Envy* can be avoided by applying this style.
- But sometimes ask.



SOLID

- **S**ingle Responsibility Principle (SRP)
- **O**pen Closed Principle (OCP)
- **L**iskov Substitution Principle (LSP)
- **I**nterface Segregation Principle (ISP)
- **D**ependency Inversion Principle (DIP)

SOLID

- **Guidelines, not rules.**
- Remember: **flexibility is not free**, it **increases complexity.**
- **Much better if you get to them by refactoring when the need is clear.**

Single Responsibility Principle

A module/function/object

should have

only one reason to change.

Single Responsibility Principle

- In this context a **responsibility** is **a reason for change**.
- Important to separate responsibilities because **each responsibility is an axis of change**.
- If a module has more than one responsibility the responsibilities become coupled.

Single Responsibility Principle

- Depending on how the application is changing, responsibilities should be separated or not.
- Wait for the need to appear. Avoid *Speculative Generality*.
- At a minimum identify responsibilities as private functions that do only one thing (SRP at method level).

Open/Closed Principle

Software entities

should be

open for extension

but

closed for modification.

Open/Closed Principle

- A module using such abstractions can be **closed for modification**, since it depends on an abstraction that is fixed. **Yet its behavior can be extended** by creating new derivatives of the abstraction.
- It separates generic functionality from its implementation.

Open/Closed Principle

- Resist premature abstraction. Avoid *Speculative Generality*.
- Wait until the changes happen!
- Get to OCP by **refactoring**.

Liskov Substitution Principle

Subtypes must be substitutable for their base types.

Liskov Substitution Principle

- Software is really about **behavior** (responsibilities).
- Any implementation must **respect the contract** of the abstraction.
- LSP allows a module, expressed in terms of an abstraction, to **comply with OCP**.

Interface Segregation Principle

Clients should not be forced to depend on methods they do not use.

Interface Segregation Principle

- A **non-cohesive interface** contains **different groups of methods each serving different clients**.
- Changes forced by one client on methods it uses will affect other clients that do not use those methods.
- To avoid this **separate clients should mean separate interfaces**.
- **Non-cohesive interfaces** increase the chances of **violating LSP**.

Dependency Inversion Principle

a. High level modules should not depend upon low level modules. Both should depend upon abstractions.

b. Abstractions should not depend upon details. Details should depend upon abstractions.

Dependency Inversion Principle

- It makes a **high level module independent of the details** it controls.
- Critically important for constructing code that is resilient to change.
- The code will **still be vulnerable to changes in interfaces**.
- Essential for hexagonal architecture.