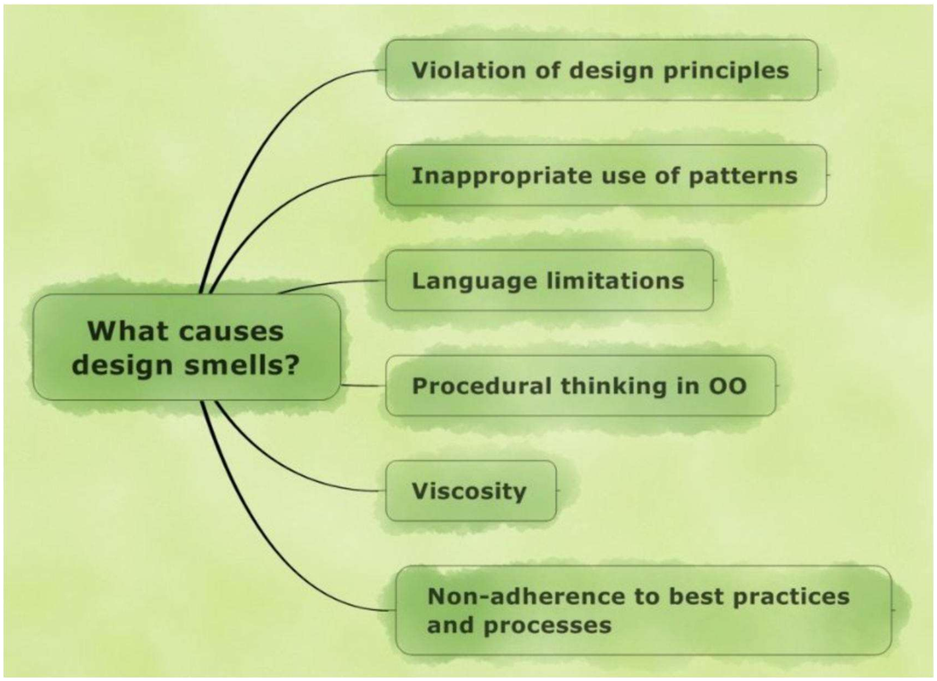
DESIGN PRINCIPLE-BASED REFACTORING

What Causes Design Smells?


What causes design smells?
Violation of design principles
Inappropriate use of patterns
Language limitations
Procedural thinking in OO
Viscosity
Non-adherence to best practices and processes

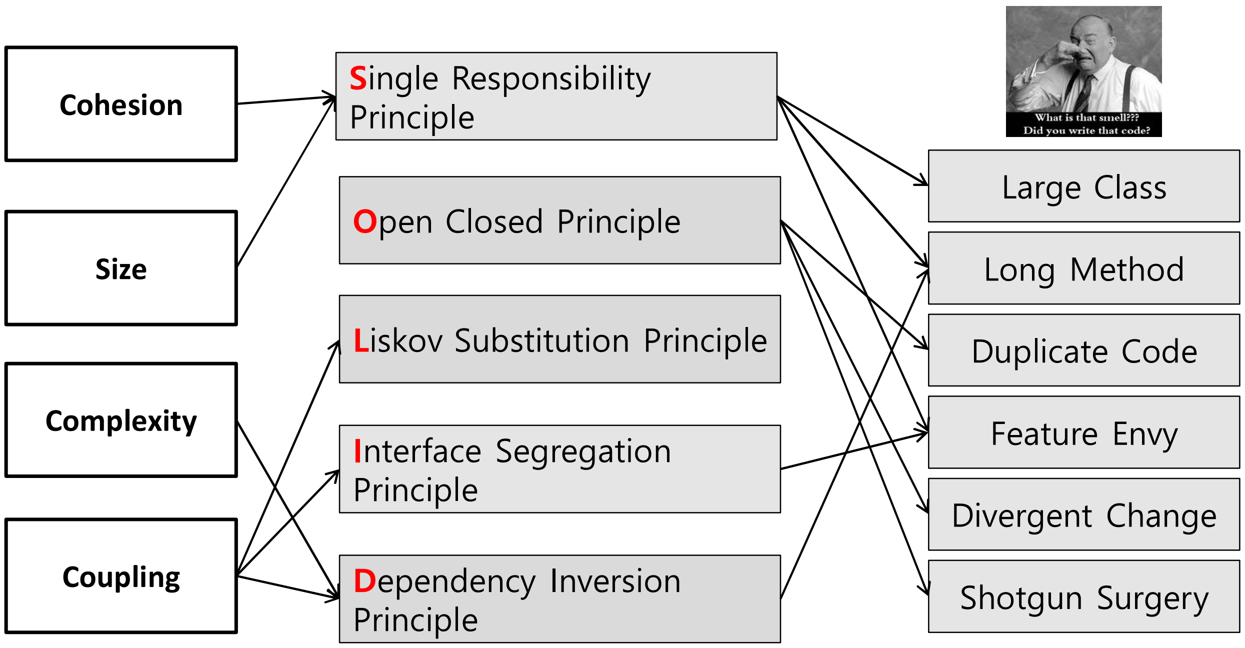
… and … many more!

Design Smells as Violations of Fundamental Principles

- What do smells indicate?
  - Violations of fundamental design principles
- Connection between smells and principles helps identify *cause of the smell* and *potential refactoring* as well.

Many Concepts

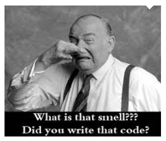
- Concerned with design understandability/maintainability


Cohesion
Size
Complexity
Coupling
Single Responsibility Principle
Open Closed Principle
Liskov Substitution Principle
Interface Segregation Principle
Dependency Inversion Principle
What is that smell??? Did you write that code?
Large Class
Long Method
Duplicate Code
Feature Envy
Divergent Change
Shotgun Surgery

Principle-based Refactoring

- Design should be evaluated and improved toward design principles.

Cohesion/Coupling/Size/Complexity
SOLID



Design Smells

OO Principles

- SRP - Single Responsibility Principle
- OCP - Open Closed Principle
- LSP - Liskov Substitution Principle
- ISP - Interface Segregation Principle
- DIP -Dependency Inversion Principle
- DRY - Don't Repeat Yourself
- SCP - Speaking Code Principle
- REP - Reuse/ Release Equivalency Principle
- CRP - Common Reuse Principle
- CCP - Common Closure Principle
- ADP - Acyclic Dependencies Principle
- SDP - Stable Dependencies Principle
- SAP - Stable Abstractions Principle
- TDA - Tell Don't Ask
- LOD – Law of Demeter
- SOC - Separation of Concerns
- Separate interface from implementation (Encapsulation)
- Program to an interface
- Favor composition over inheritance
- High cohesion and low coupling
- Protected Variation
- Indirection
- Single level of abstraction principle (SLAP)
- YAGNI – You ain't gonna need it
- KISS – Keep it simple and stupid
- …

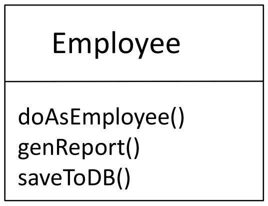
SRP: *A class should have one, and only one reason to change*

Operations in the class should be closely related to one subject area → *strong cohesion*

Shift the meaning a little bit, then operations in the class should be closely related to one responsibility.

In the context of the SRP, a responsibility is defined as *"a reason for change"* or *"axis of change"*

If you can think of more than one motive for changing a class, you are violating the SRP.



OCP: The Open-Closed Principle

Software entities (classes, modules, functions etc.) should be open for extension, but closed for modification

- Bertrand Meyer, 1988 -

The OCP is the single most important guide for the OO designers.

LSP: The Liskov Substitution Principle

Derived classes must be usable through the base class interface without the need for the user to know the difference

- Barbara Liskov, 1988 -

Subtypes must be substitutable for their base types (super types).

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

ISP: The Interface Segregation Principle

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

The ISP is a structural principle like DIP.

Deals with the disadvantages of "fat" interface

-- fat interface, polluted interface, non-cohesive interface

Violation of the ISP violates the SRP.

DIP: The Dependency Inversion Principle

High level modules should not depend upon low level modules. Both should depend on abstractions. Abstractions should not depend upon details. Details should depend upon abstraction.

OCP states the goal; DIP states the mechanism.

A base class in an inheritance hierarchy should not know any of its subclasses.

Modules with detailed implementations are not depended upon, but depend themselves upon abstractions.

Speaking Code Principle (SCP)

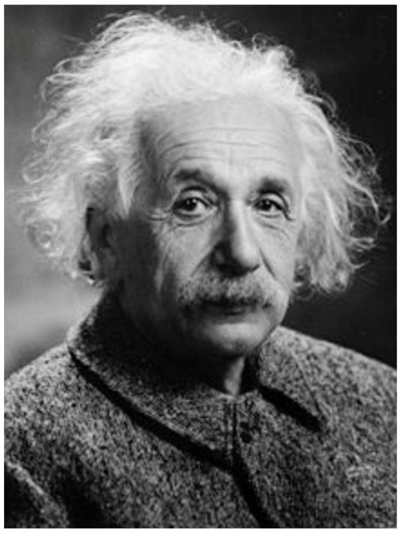
- The code should communicate its purpose.

"Any fool can write code that a computer can understand.
Good programmers write code that humans can understand"

-- Martin Fowler, 1999.

Keep it Simple Stupid (KISS)

Keep it short
and
simple



Keep it simple
and
straightforward

"If you can't explain it simply, you don't understand it well enough."
– Albert Einstein

Keep it Simple Stupid (KISS)

- A simple system is easier to maintain and debug than a complex system using lots of design patterns and undecipherable code.
- Beginner programmer writes complicated, verbose code while the more experienced developer writes clean, simple and short code.

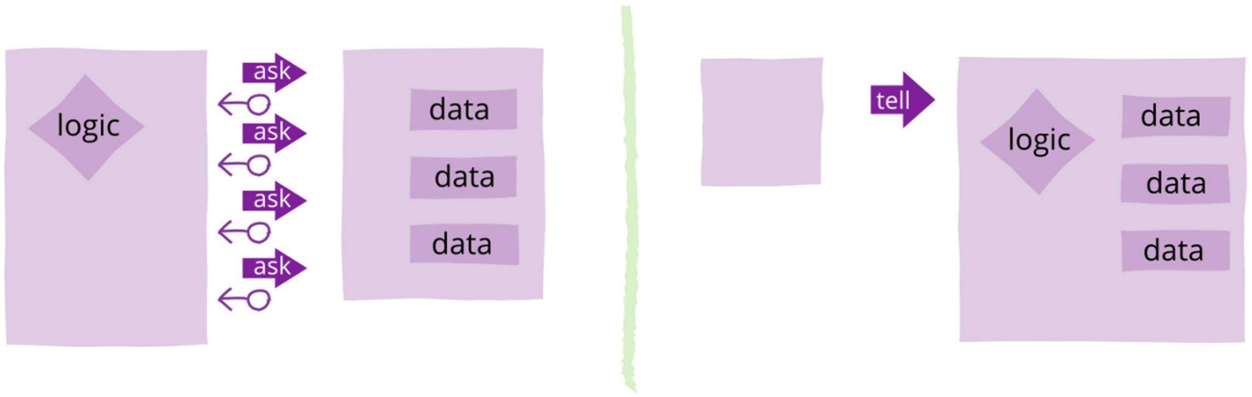
KISS it to make it better!

You Ain't Gonna Need It (YAGNI)

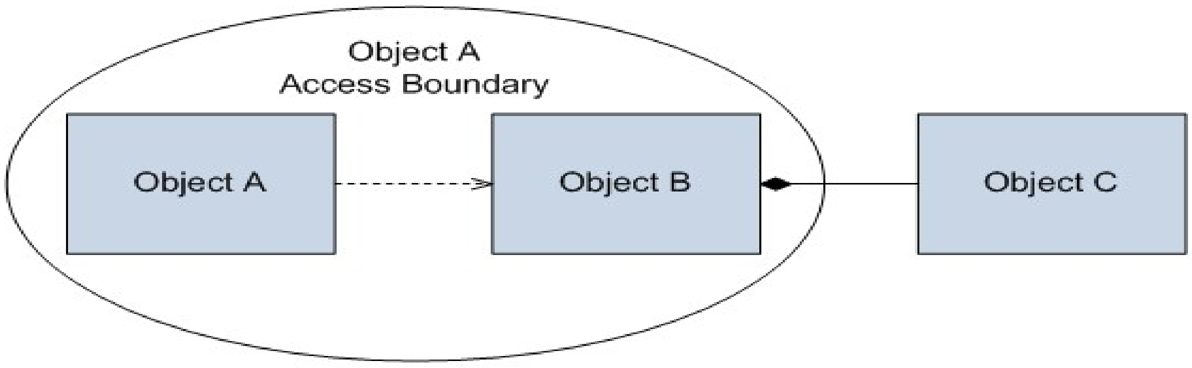
- Coding is one of the areas where laziness is a good thing!
  - The less the code, the less to maintain and introduce bugs.
- Strive for a minimalism.
- Let the specification guides you.
  - Do as little as possible to make it work, no more & no less.

Tell Don’t Ask

- Object-orientation is about bundling data with the functions that operate on that data.
- Rather than asking an object for data and acting on that data, we should instead tell an object what to do.



Don’t Talk to Strangers



Solution

- Assign the responsibility to a client’s direct object (i.e., *familiar*) to collaborate with an indirect object (i.e., *stranger*), so that the client does not know about the indirect object.

Law of Demeter

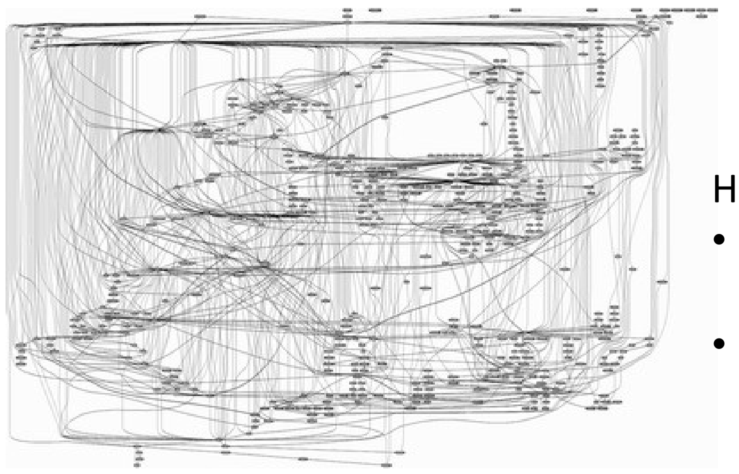
(Principle of Least Knowledge)

Constrains on what objects you should send a message to within a method.

- The *this* object (or *self*).
- The parameter of the method.
- An attribute of *self*.
- An element of a collection which is an attribute of *self*.
- An object created within the method.

Low Coupling

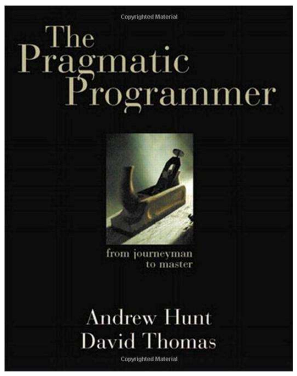
- Degree of interdependence between two modules.
- Highly coupled systems are harder to understand and maintain.



How to achieve low coupling
- Eliminate unnecessary relationships
- Reduce the number of necessary relationships

Don't Repeat Yourself (DRY)

- The Pragmatic Programmer, by Andy Hunt and Dave Thomas, 2010.
- The DRY principle states that these small pieces of knowledge may **only occur exactly** once in your entire system.
- Every piece of knowledge must **have a single, unambiguous, authoritative representation** within a system.
- When you find yourself writing code that is similar or equal to something you've written before, take a moment to think about what you're doing and don't repeat yourself.



Once and Only Once
Single Source of Truth
Single Point of Truth

DRY *vs.* WET Solution

Violations of DRY are typically referred to as WET solutions, which is commonly taken to stand for

- “Write Everything Twice”,
- “We Enjoy Typing” or

“Waste Everyone's Time”.

GRASP Patterns

General Responsibility Assignment Software Patterns

- (Information) Expert
- Creator
- High Cohesion
- Low Coupling
- Controller
- Polymorphism
- Pure Fabrication
- Indirection
- Protected Variations

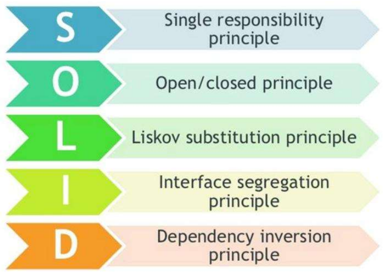
Package-related Principles

- Stable Abstractions Principle (SAP)
  – Packages that are maximally stable should be maximally abstract.
- Common Closure Principle (CCP)
  – This principle tries to pull classes that are tightly coupled to each other together into the same package.
- Acyclic Dependencies Principle (ADP)
  – There can be no cycles in the dependency structure
- Stable Dependencies Principle (SDP)
  – Any packages that are volatile should not be depended on by a package that is difficult to change

The Rules

- Apply Common Sense
- Don’t get too dogmatic / religious
- All other principles are just that
  - Guidelines
  - “best practices”
  - Consider carefully if you should violate them - but, know you can.

Good design is all about trade-offs

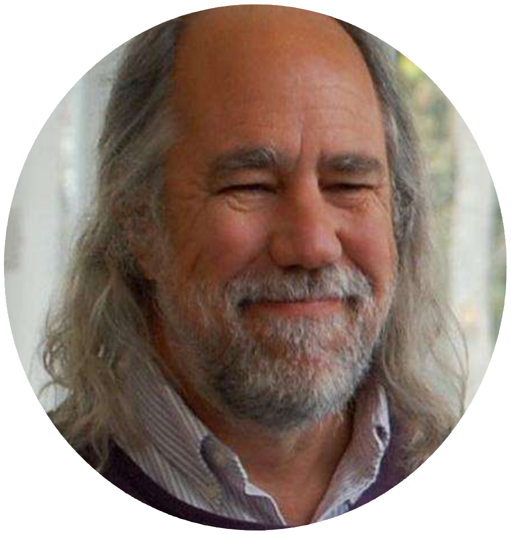

S Single responsibility principle
O Open/closed principle
L Liskov substitution principle
I Interface segregation principle
D Dependency inversion principle

is a symptom indicating the violation of

is screaming for

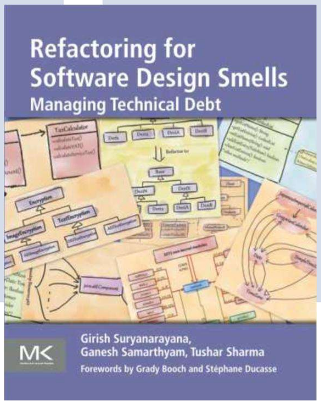
Booch’s Fundamental Principles

- Abstraction
- Encapsulation
- Modularization
- Hierarchy



Yet Another Classification of Design Smells

Abstraction	Modularization	Encapsulation	Hierarchy
Missing	Broken	Deficient	Missing
Imperative	Insufficient	Leaky	Unnecessary
Incomplete	Cyclically-dependent	Missing	Unfactored
Multifaceted	Hub-like	Unexploited	Wide
Unnecessary			Speculative
Underutilized			Deep
Duplicated			Rebellious
			Broken
			Multipath
			Cyclic


Refactoring for Software Design Smells
Managing Technical Debt
Girish Suryanarayana, Ganesh Samarthyam, Tushar Sharma
Forewords by Grady Booch and Stéphane Ducasse

Some Important Principles

Principle	Description
SCP	The code should communicate its purpose.
Cohesion, SRP	A class should have one, and only one, reason to change
Coupling	A class should minimize its dependency on others
DRY	Do not write the same or similar code more than once.
DIP	High-level concepts shall not depend on low-level concepts/implementations.

Design Principles and Bad Smells

- Bad smells describes the situation where design principles are not satisfied

Principle	Bad smells
SCP	Comment, Deeply Nested Condition, Error Code
Cohesion, SRP	Long Method Large Class, Divergent Change Feature Envy, Primitive Obsession, Refused Bequest
Coupling	Long Parameter List, Control Couple, Message Chains Fat Interface
DRY	Duplicate Code, Dependent Logic
DIP	Inappropriate Intimacy, Switch Statements

Design Principles and Design Patterns

- Refactoring towards a pattern can satisfy certain design principle(s) in removing bad smells.

Principle	Patterns
SCP	-
Cohesion, SRP	Decorator Pattern, Observer Pattern
Coupling	Façade Pattern
DRY	Factory Method Pattern, Abstract Factory Pattern, Template Method Pattern
DIP	Iterator Pattern, Composite Pattern, Strategy Pattern, Command Pattern, State Pattern