SOLID

Software Development is not a Jenga game

Writing SOLID Code

- SOLID mnemonic acronym was formed to track Uncle Bob's first five principles of OOP and OOD. They are:
- Single Responsibility Principle (SRP)
- Open-closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)

SINGLE RESPONSIBILITY PRINCIPLE

Just Because You Can, Doesn't Mean You Should

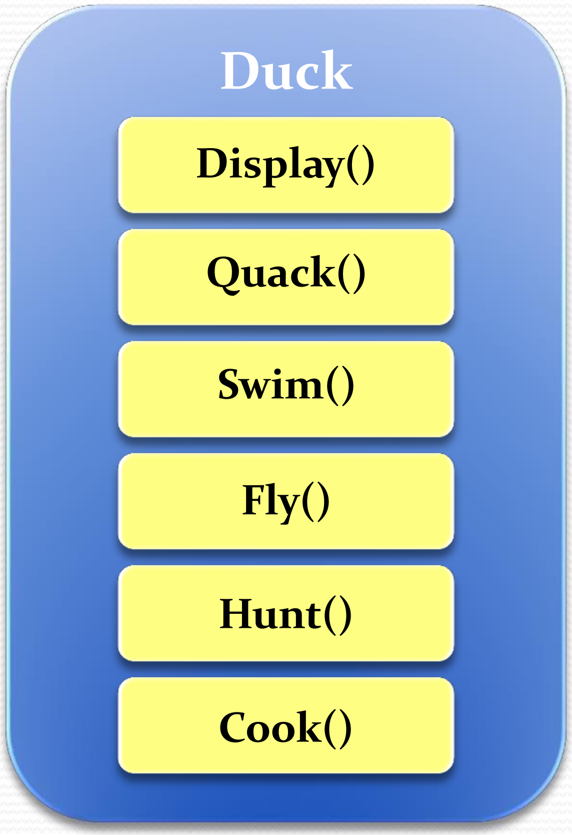
Single Responsibility Principle (SRP)

- **Responsibility**

  Set of ***related functionality or behavior*** that the class performs.

- **Each *object* in your application should have *one and only responsibility.***
- **And, each *method* (service) in that object should focus on carrying out that single responsibility.**
- That responsibility should be entirely encapsulated by the context (class, function, variable)
- All its services should be narrowly aligned with that responsibility
- Uncle Bob further offers this rubric: Responsibility is a *reason to change*

Single Responsibility Principle



How many responsibilities does this class have?

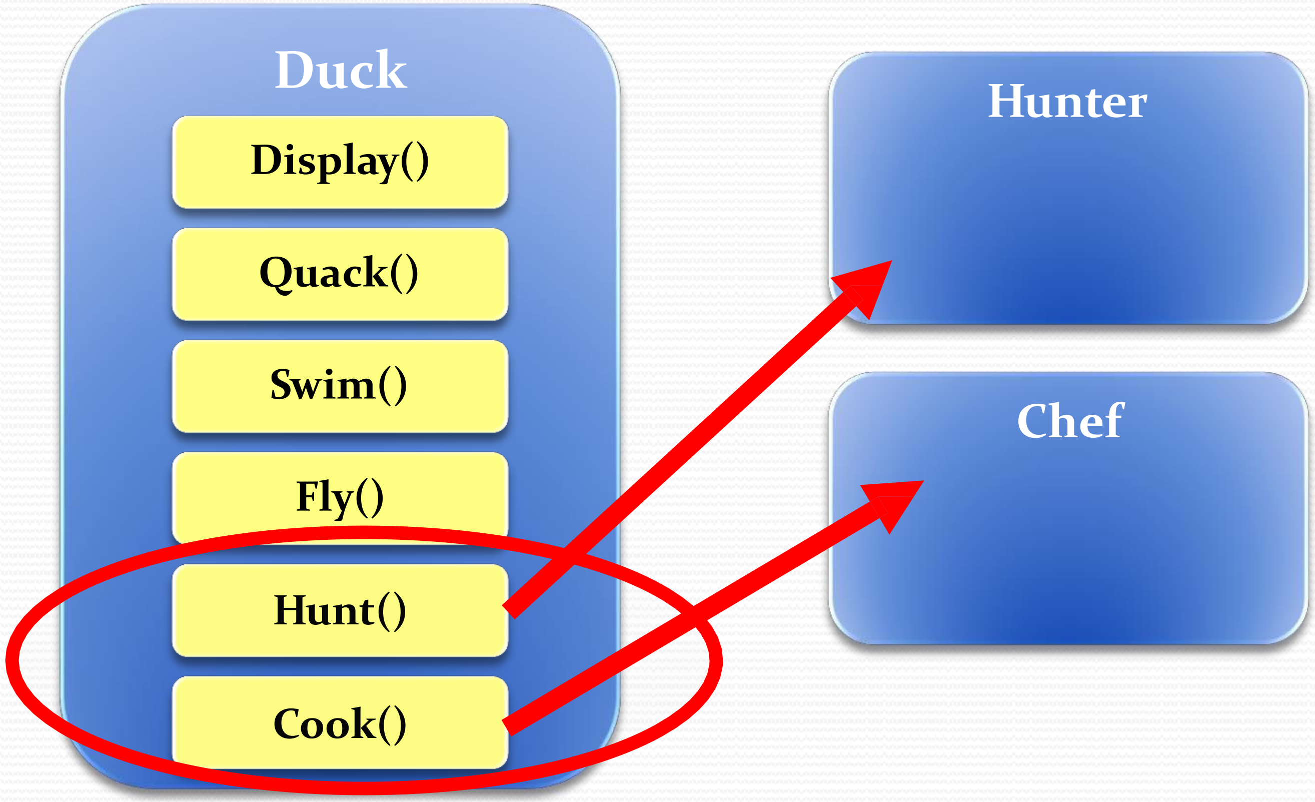
SRP Analysis

- **Simple test to identify responsibility**
- **The <Object> <Method> Itself.**

	Follows SRP
The Duck Display[s] Itself.	✓
The Duck Quack[s] Itself.	✓
The Duck Swim[s] Itself.	✓
The Duck Fly[s] Itself.	✓
The Duck Hunt[s] Itself.	✗
The Duck Cook[s] Itself.	✗

Refactoring Duck

- Factor out *unrelated behavior* so that Duck ends up with a *single responsibility.*



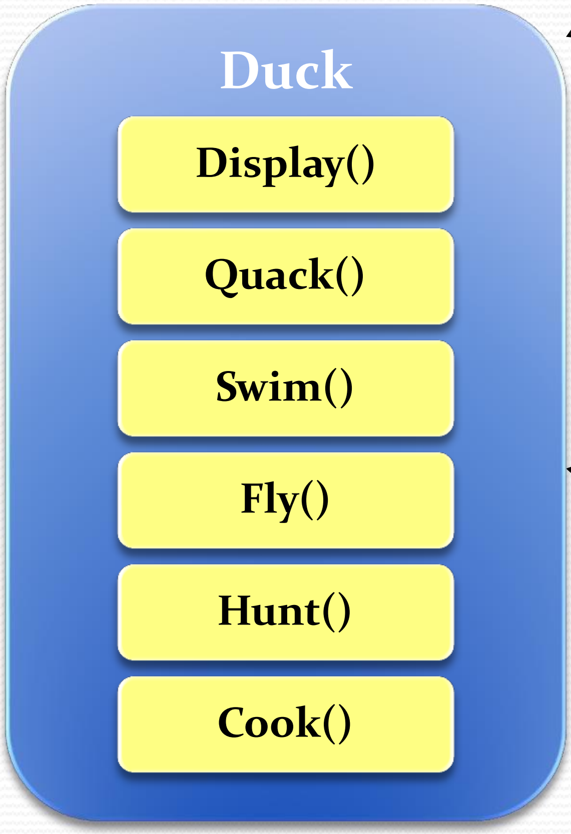
Measuring SRP

- FYI: *Cohesion*
  - Term used to measure ***how strongly*** the services i your object support a single responsibility.

- Architect Translation:
  - **High Cohesion**: Class designed around set of ***related*** functionality
  - **Low Cohesion**: Class designed around set of ***unrelated*** functionality

Why Bother?

- *WHY* is it good for a class to have a single responsibility?



Each responsibility is area for potential *change*

Multiple responsibilities mean *multiple* reasons to change

- changes to one responsibility *may impair* the class's ability to meet the others
- may *force* rebuilding, retesting and redeployment of behavior not changed

Multiple responsibilities can *impair reuse* opportunities

Multiple responsibilities can make class *hard to understand*

Multiple responsibilities can make class *delicate* – hard to maintain

OPEN CLOSED PRINCIPLE

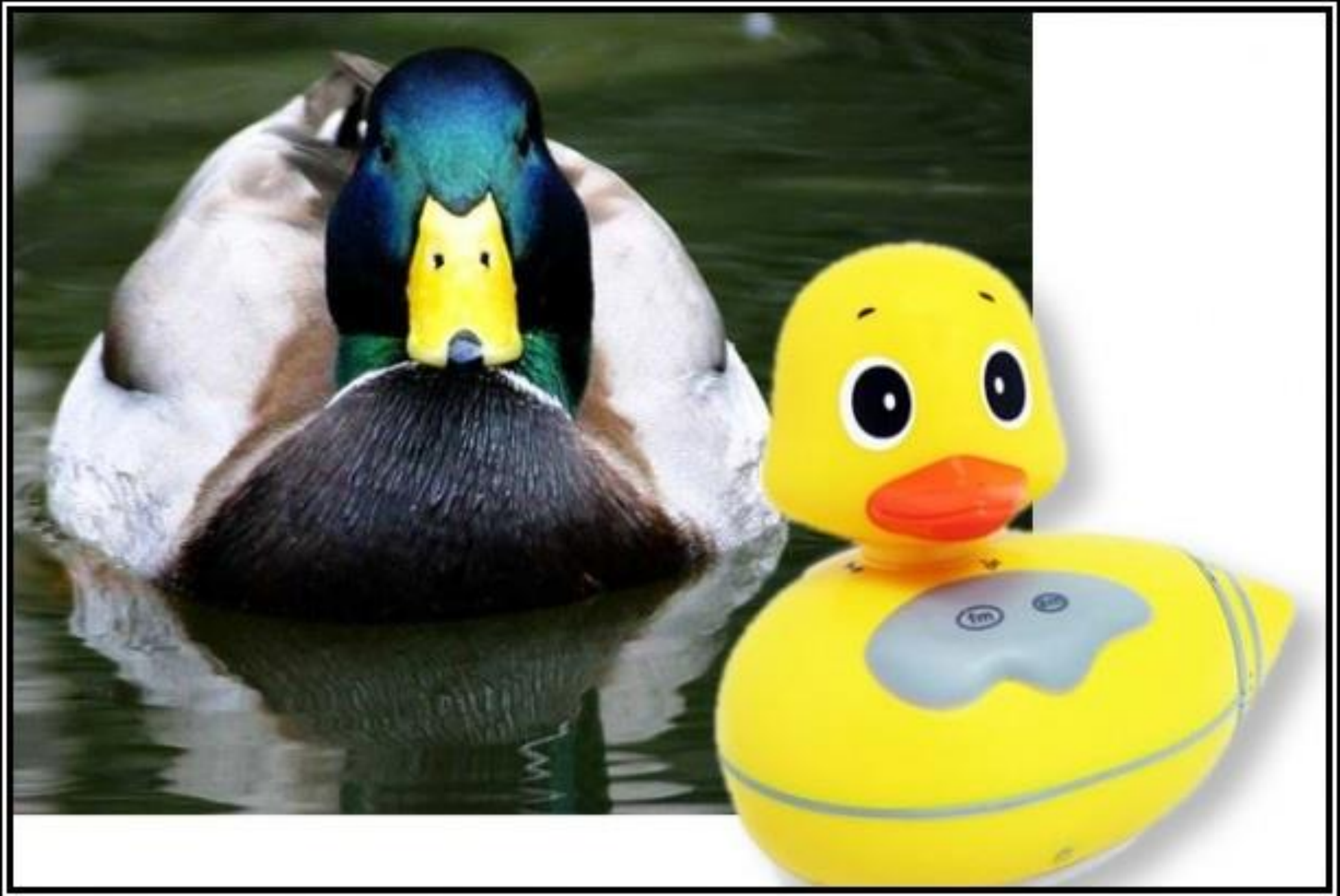
Open Chest Surgery Is Not Needed When Putting On A Coat

Open-closed Principle (OCP)

- The OCP states that "*software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification*“
- Such an entity can allow its behavior to be modified without altering its source.
- Code obeying this principle doesn't change when it is extended, and therefore needs no code reviews and unit tests outsideof it’s extended functionality.

Open-closed Principle (OCP)

- Two models for this exist
- Bertrand Meyer's model (early 90's) which subclasses the implementation, but the interface may change – now called "implementation inheritance".
- Uncle Bob's Model ('96) reuses interfaces through inheritance (abstract classes or interfaces) but not implementation code.
- **The existing interface is closed to modifications.**



LISKOV SUBSTITUTION PRINCIPLE

If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction

Liskov Substitution Principle (LSP)

- Introduced by Barbara Liskov in 1988
- States that, Derived classes must be substitutable for their base classes
- In otherwords…
  - Any code in which a specific class' properties or methods are being used should work just as well with any class of that subtype
- Makes code more robust and less coupled

Liskov Substitution Principle (LSP)

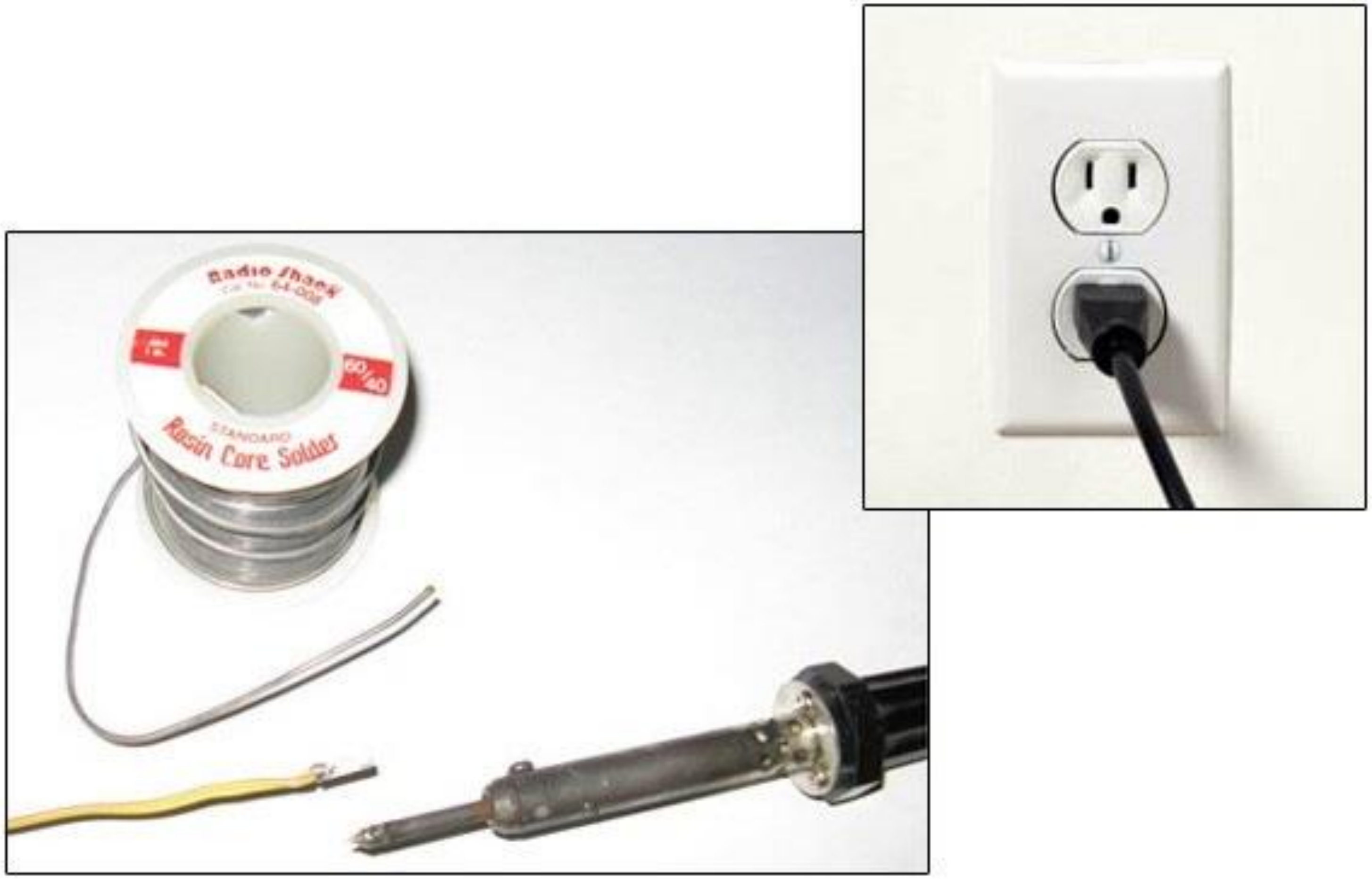
- More abstractly speaking…
  - The LSP is a particular definition of a subtyping relation called (strong) behavioral subtyping
  - It defines a notion of substitutability for mutable objects
  - If S is a subtype of T, then objects of type T in a program may be replaced with objects of type S without altering the correctness of the code.

INTERFACE SEGREGATION PRINCIPLE

You Want Me To Plug This In, Where?

Interface Segregation Principle (ISP)

- States that, No client should be forced to depend on interfaces it does notuse
- In otherwords...
  - Do not put anything into an interface that each implementing class cannotuse
- Create a larger number of smaller interfaces
- Such shrunken interfaces are also called *role interfaces*
- The ISP is intended to keep a system decoupled and thus easier to refactor, change, and redeploy



DEPENDENCY INVERSION PRINCIPLE

Would You Solder A Lamp Directly To The Electrical Wiring In A Wall?

Dependency Inversion Principle (DIP)

- The DIP refers to a specific form of decoupling software modules.
- Two key aspects to this:
  - High-level modules should not depend on low-level modules. Both should depend on abstractions.
  - Abstractions should not depend on details. Details should depend on abstractions.
- Thus, both high- and low-level objects must depend on the same set of abstractions.

Lab

02 Lab - SOLID