SOLID PRINCIPLES

Nadia Comanici

DEMYSTIFYING THE ACRONYM

SINGLE RESPONSABILITY PRINCIPLE

SINGLE RESPONSIBILITY PRINCIPLE

Câte responsabilități ar trebui să aibă o clasă?

SINGLE RESPONSABILITY PRINCIPLE

Every object should have a single responsability and that responsability should be entirely encapsulated by the class

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

(Robert C. Martin – Uncle Bob)

COHESION & COUPLING

Cohesion

- The degree to which a part of a code base forms a logically single, atomic unit

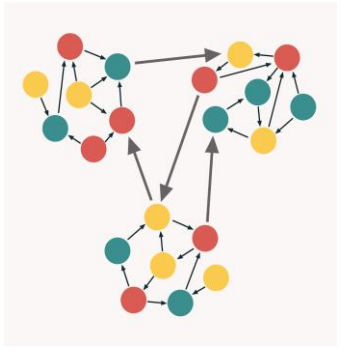
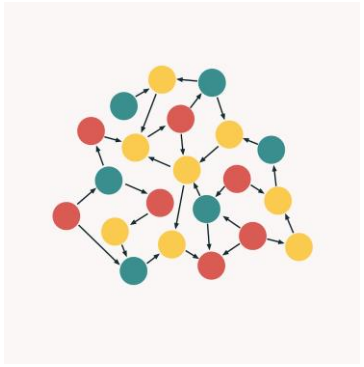
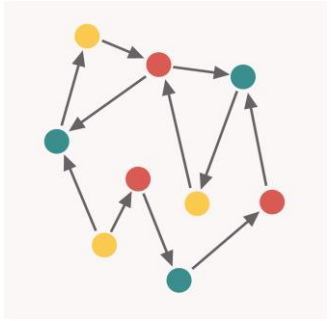
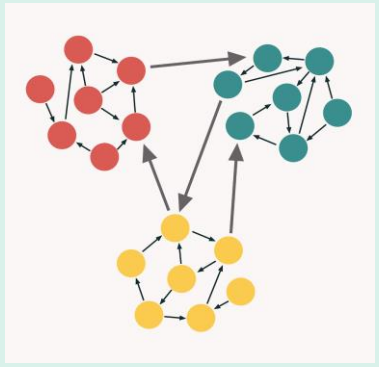
High cohesion = **keeping parts of a code base that are related to each other in a single place.**

Coupling

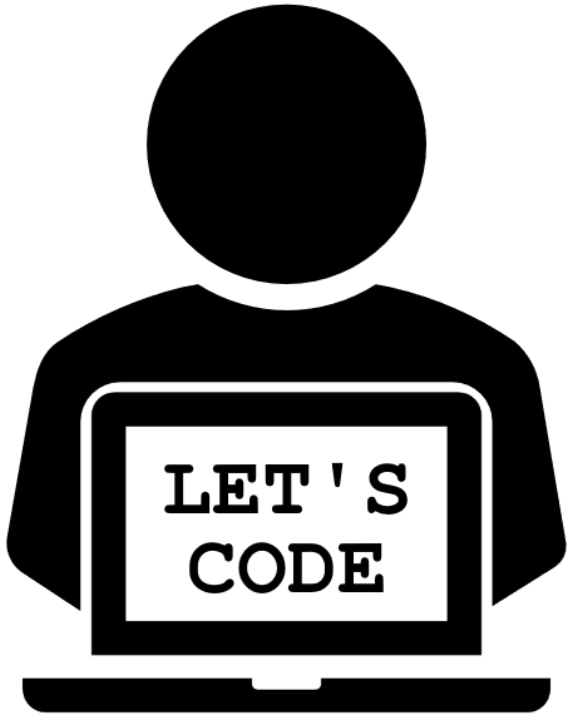
- The degree to which a single unit is independent from others

Low coupling = **separating unrelated parts of the code base as much as possible**

COHESION & COUPLING

	Low Cohesion	High Cohesion
High Coupling	 Pooly selected boundaries	 God Object
Low Coupling	 Destructive Decoupling	 Ideal

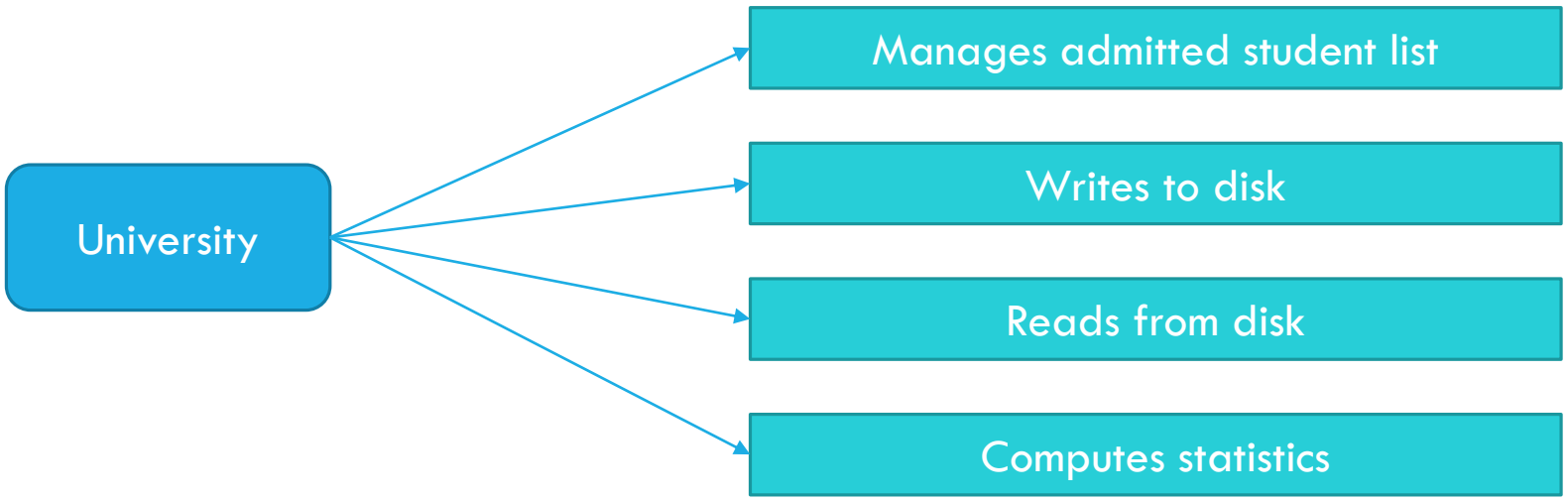
Source: https://enterprisecraftsmanship.com/2015/09/02/cohesion-coupling-difference/

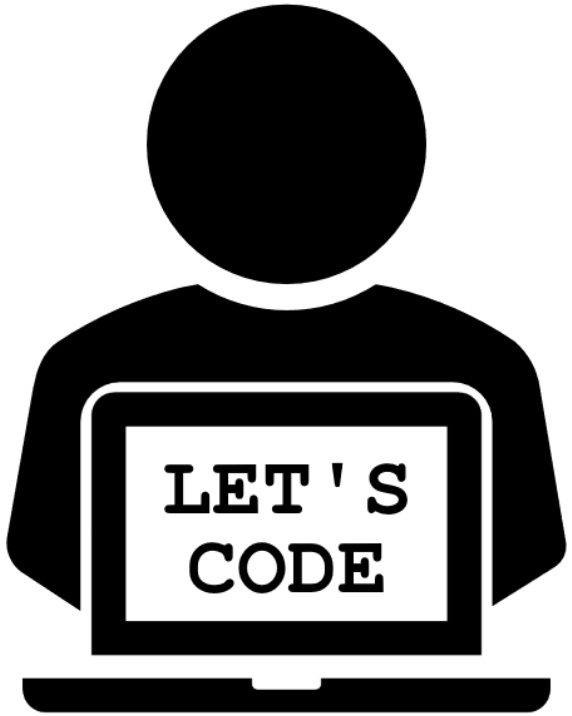


STUDENTS SAMPLE

Single Responsibility Principle

HOW MANY RESPONSABILITIES FOR UNIVERSITY?

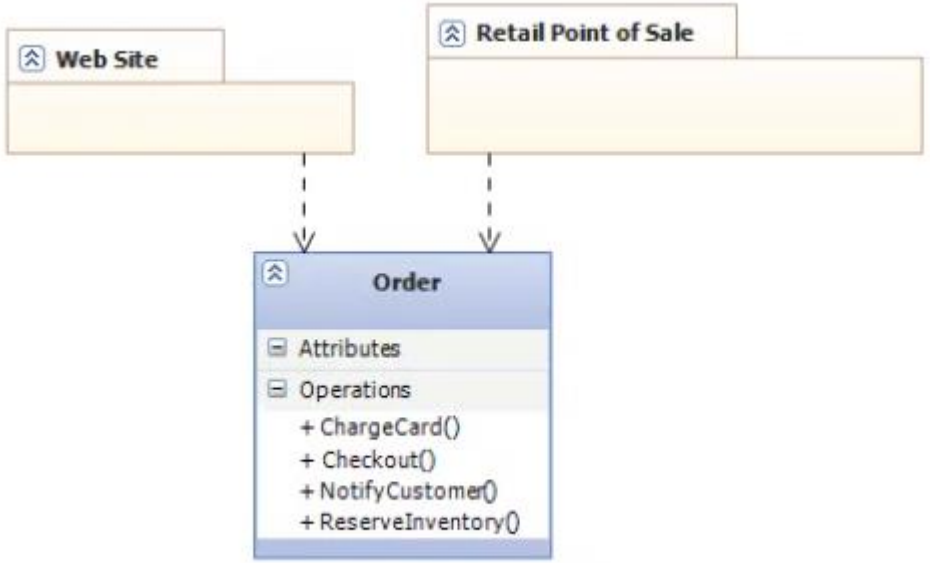


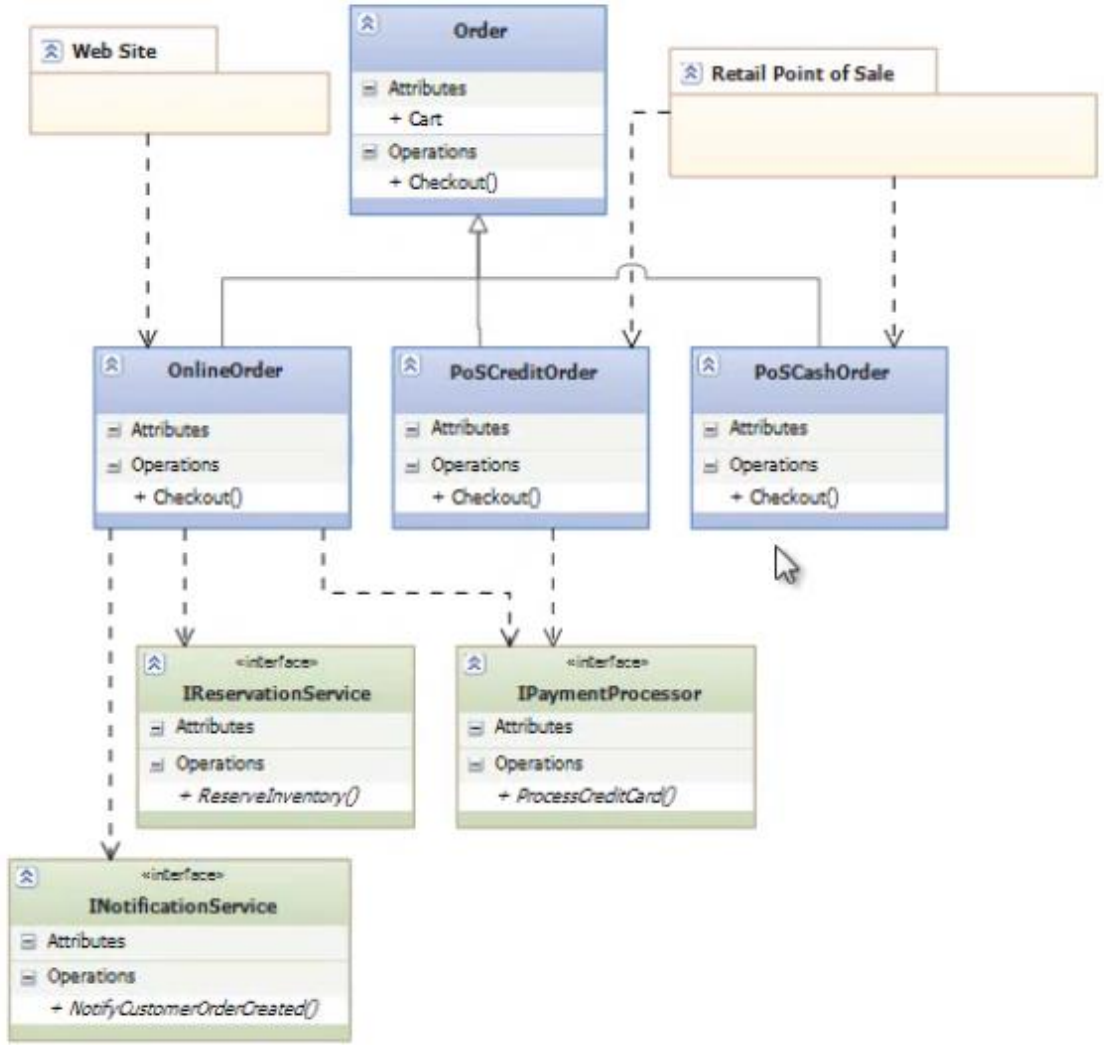


ORDERS SAMPLE

Single Responsibility Principle

ORDERS SAMPLE





DEPENDENCY INVERSION PRINCIPLE

DEPENDENCY INVERSION PRINCIPLE

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

(Agile Principles, Patterns and Practices in C#)

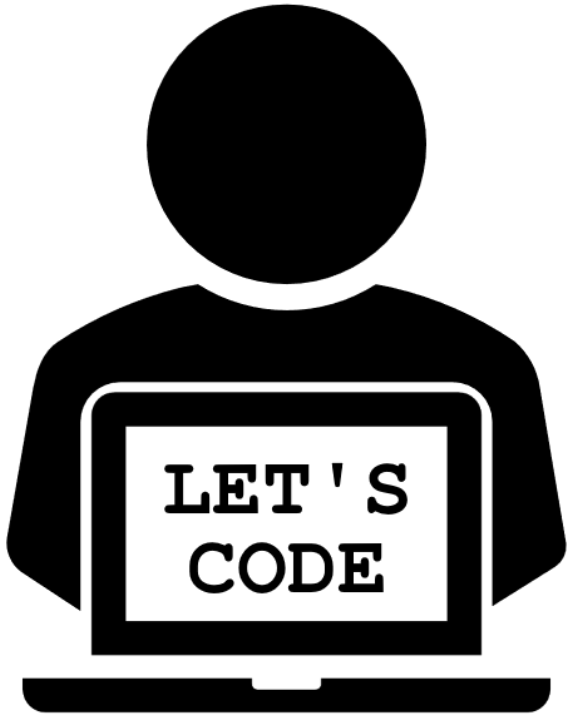
WHAT ARE DEPENDENCIES?

- Framework
- Third Party Libraries
- Database
- File System
- Email
- Web Services
- System Resources (DateTime.Now)
- Configuration
- The new keyword
- Static methods
- Thread.Sleep
- Random

CLASS DEPENDENCIES

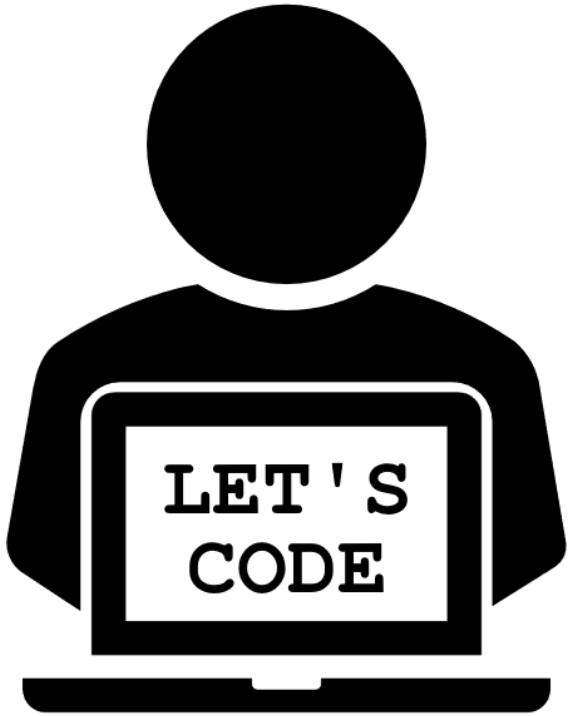
- Class constructors should require all the dependencies that the class needs

1. Explicit dependencies (clear dependencies in constructor)
2. Implicit dependencies (hidden dependencies in constructor)



DATETIME SAMPLE

Dependency Inversion Principle



ORDERS SAMPLE

Dependency Inversion Principle

IOC CONTAINERS

- Responsible object graph instantation
- Initializes at application startup via code or configuration
- Managed interfaces and the implementation to use are registered with the container
- Dependencies on interfaces are Resolved at application startup or runtime

Examples:

- **Autofac**
- Microsoft Unity
- StructureMap
- Ninject
- Windsor
- Funq/Munq

HOW DO WE INJECT DEPENDENCIES?

1. Constructor Injection
2. Property (Setter) Injection
3. Parameter Injection

CONSTRUCTOR INJECTION

- Stretegy Pattern
- Pros:
  - Classes are always in a valid state once constructed
  - Works well with or without container
- Cons:
  - Constructor my have a lot of dependencies
  - You may need a paramterless constructor (Serialization)
  - Some methods may not need all dependencies
  - Dependencies cannot change during the object lifetime

PROPERTY (SETTER) INJECTION

- Pros:
  - Dependencies can change during the lifetime of the object
  - Flexible
- Cons:
  - Objects may be in invalid state after construction
  - You don't know all the dependencies and the order in which they should be set

PARAMETER INJECTION

- Pros:
  - Flexible
  - No change for the rest of the class, only that method
- Cons:
  - You need to update all usages of that method
  - A method might need a lot of dependencies -> too many parameters

INTERFACE SEGREGATION PRINCIPLE

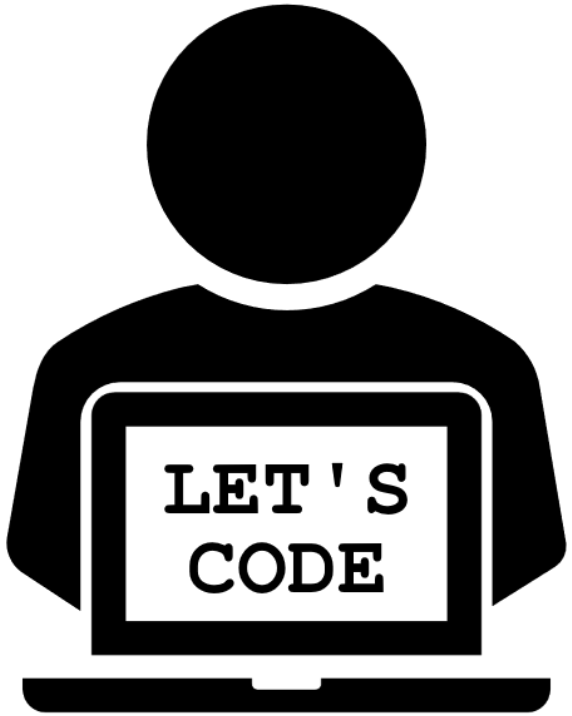
I JUST NEEDED A USB INPUT PLUG...

INTERFACE SEGREGATION PRINCIPLE

- No implementation should be forced to have methods that it does not use (or need)
- Use multiple, smaller interfaces

- Advantages:
  - ✓ Simpler classes, just with needed functionality
  - ✓ No need to implement a lot of unnecesairy methods – e.g. MembershipProvider

- More:
  - ✓ https://dotnettutorials.net/lesson/interface-segregation-principle/
  - ✓ https://exceptionnotfound.net/simply-solid-the-single-responsibility-principle/



VEHICLES SAMPLE

Interface Segregation Principle

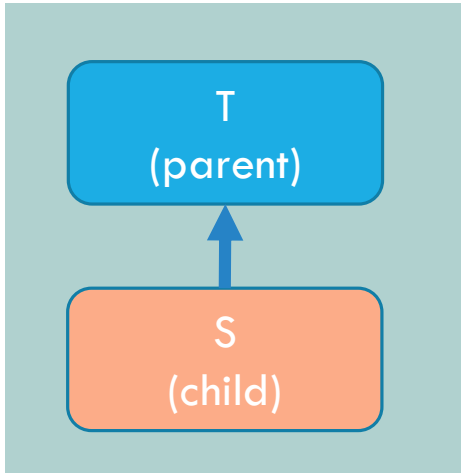
LISKOV'S SUBSTITUTION PRINCIPLE

WHO IS LISKOV?

- Barbara Liskov
  https://en.wikipedia.org/wiki/Barbara_Liskov
- One of the first women with Stanford PhD
- With Jeannette Wing, she developed a particular definition of subtyping, commonly known as the Liskov substitution principle
- Currently, she leads the Programming Methodology Group at MIT

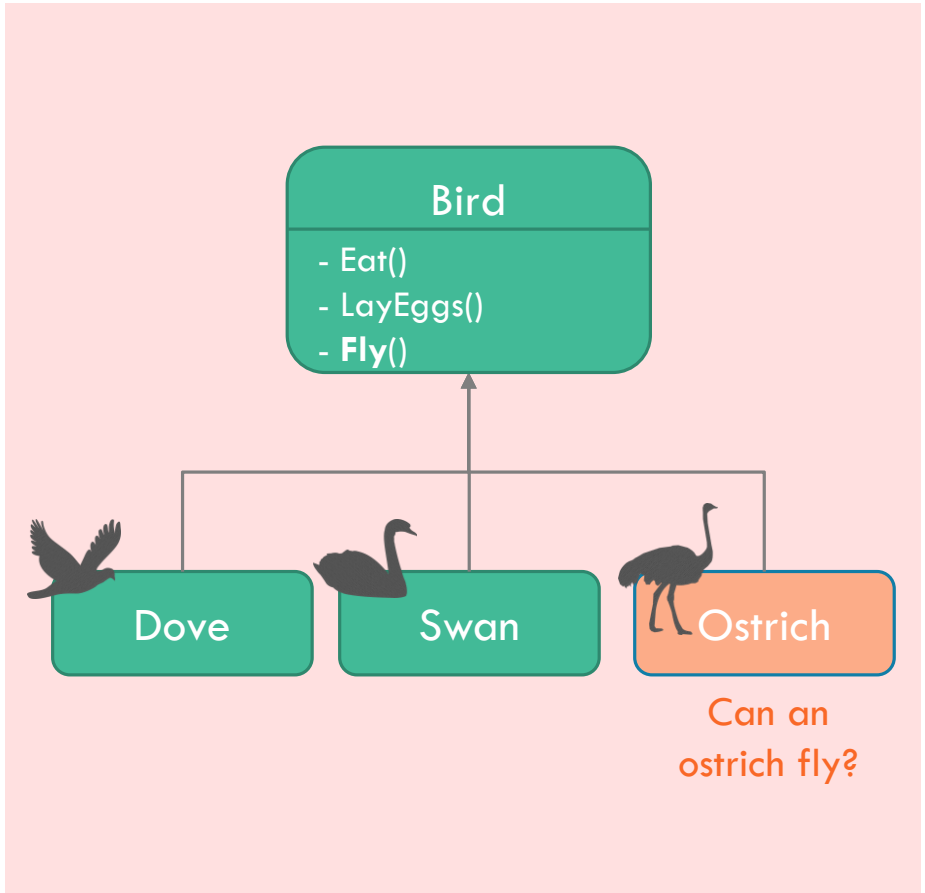
LISKOV'S SUBSTITUTION PRINCIPLE

Let $\phi(x)$ be a property provable about objects x of type T.
Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T.



- If ChildClass is a subtype of ParentClass, then any object of type ParentClass may be replaced with an object of type ChildClass without breaking the program
- **Any child class should be able to do anything the parent can do**
  - Always ask yourself "Is S a T"?
- Signs that this principle is broken:
  - Return NotImplementedException() in a child method
  - Child method has empty (not implemented) body

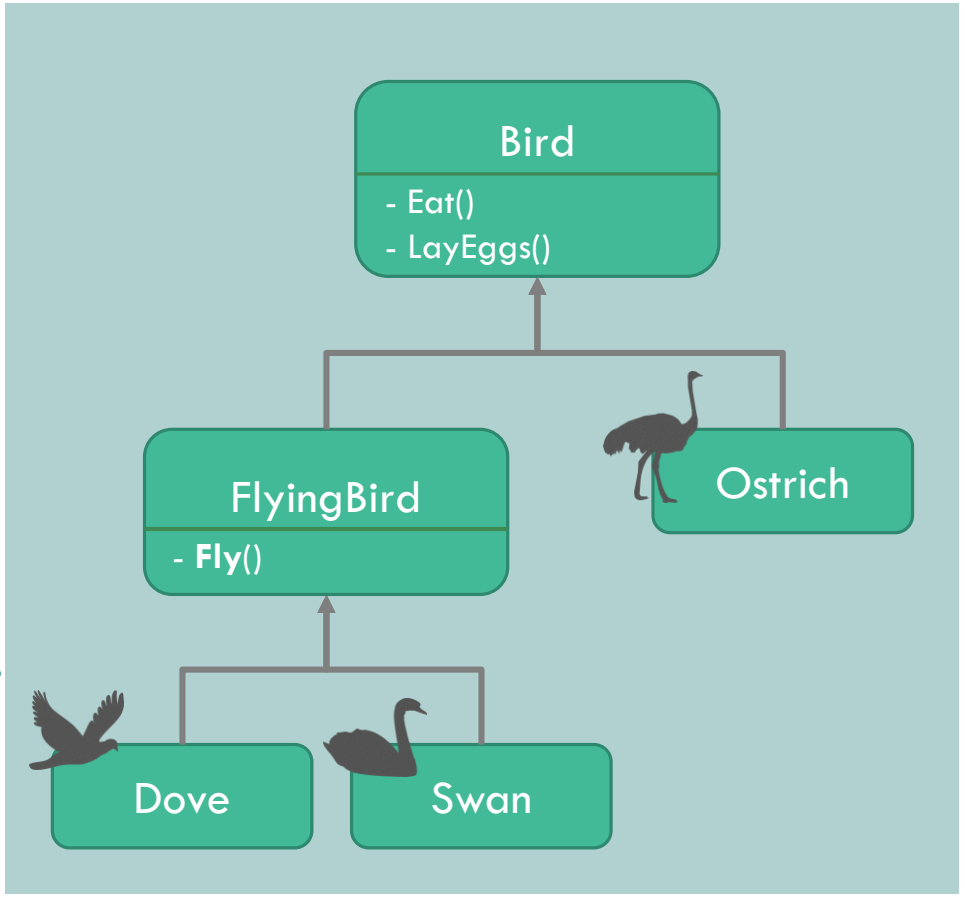
LISKOV'S SUBSTITUTION PRINCIPLE

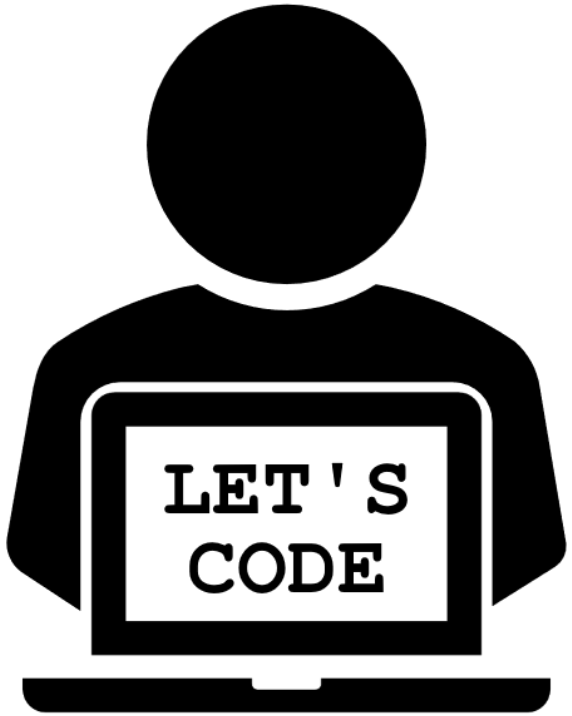


Barbara Liskov

Hypothesis:
- **T** can do **OpA()**
- **S** is a subtype of **T**

Conclusion:
- **S** should be able to do **OpA()**

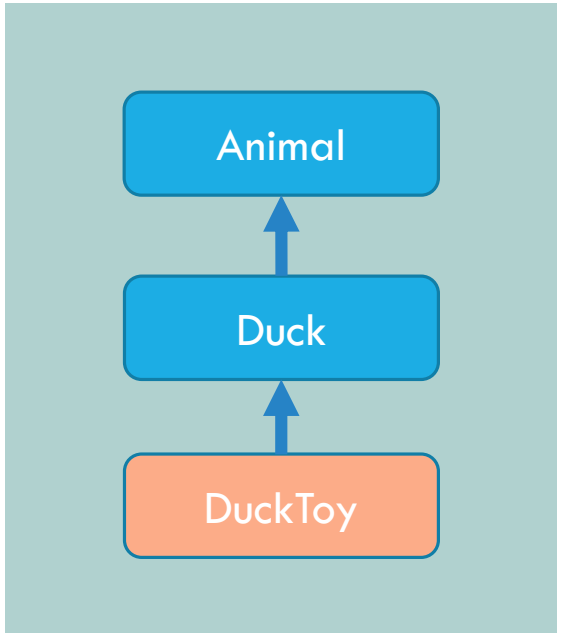


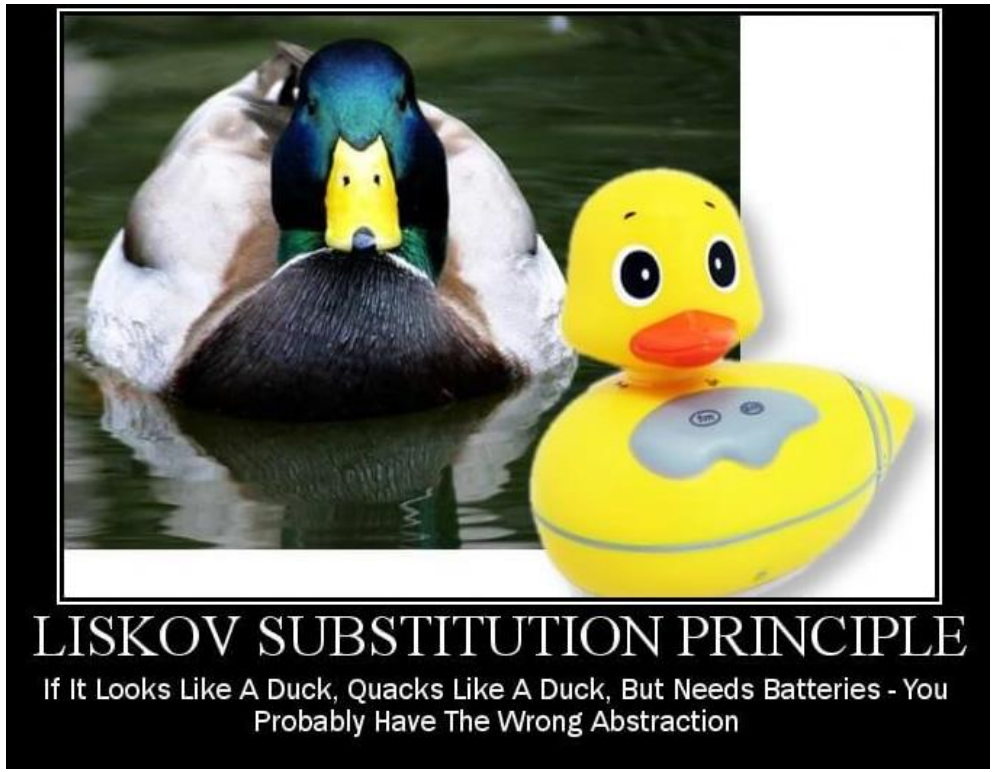


SQUARE SAMPLE

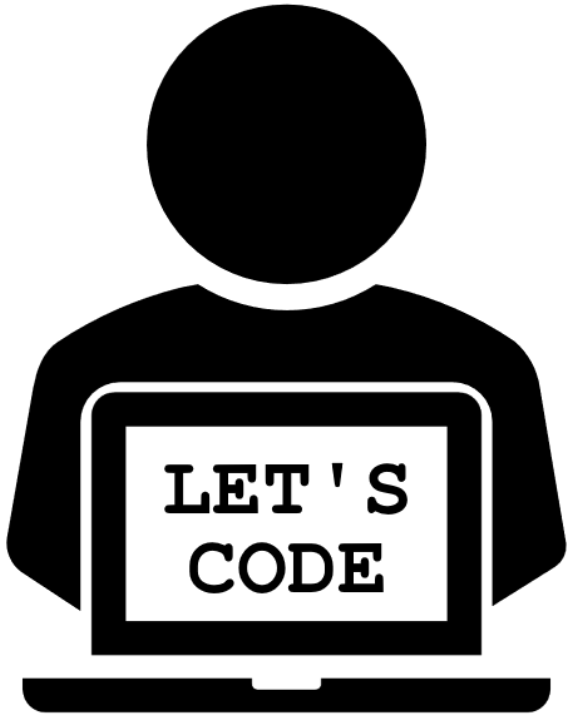
Liskov Substitution Principle

OTHER EXAMPLES – IS "S" A "T"?





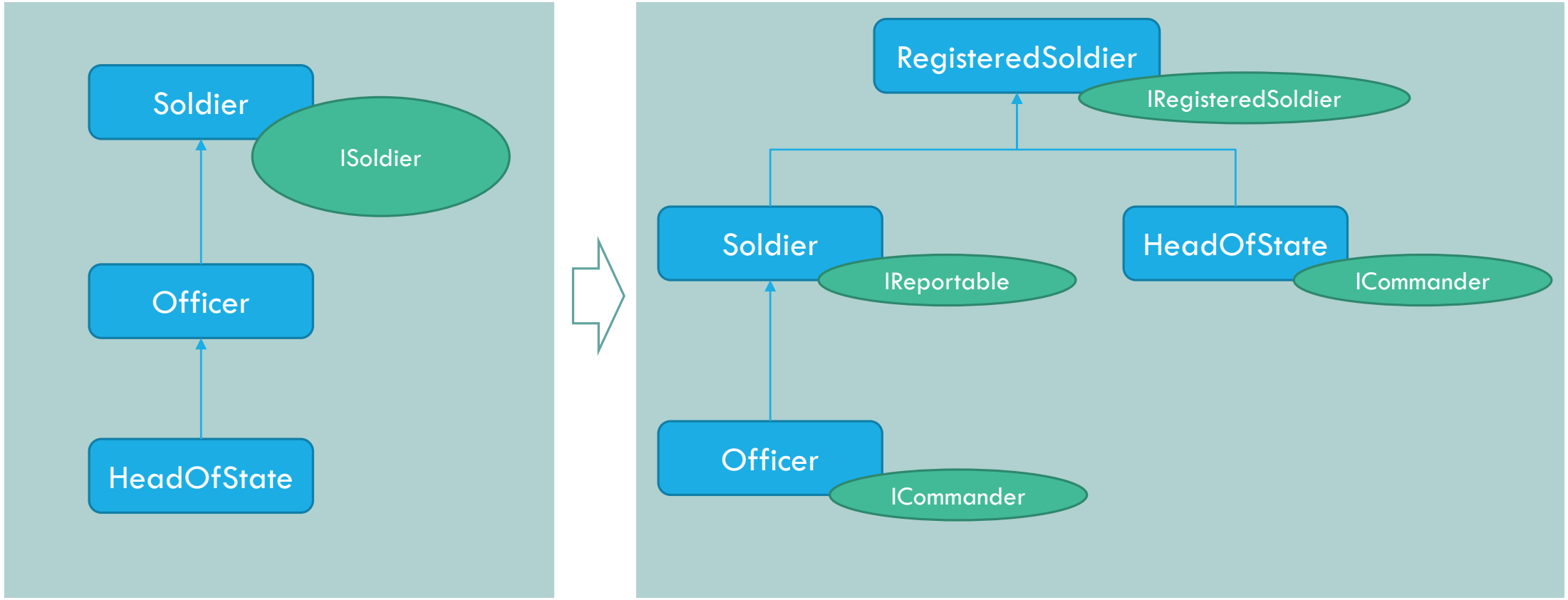
- Image Source: https://exceptionnotfound.net/simply-solid-the-liskov-substitution-principle/



ARMY SAMPLE

Liskov Substitution Principle + Interface Segregation Principle

CLASS HIERARCHY



OPEN / CLOSED PRINCIPLE

OPEN / CLOSED PRINCIPLE

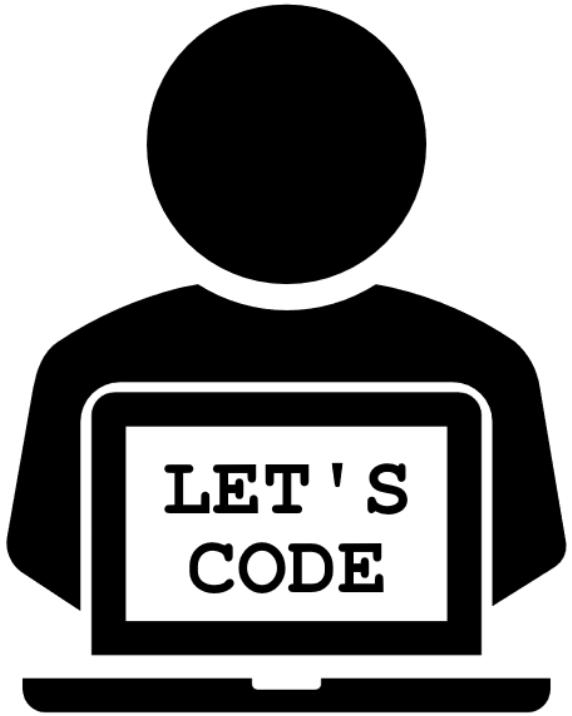
- Software entities should be:
  - **Opened for extension** = You can add new behviour
  - **Closed for modification** = You should not change the source/binary code

- Write your code so that you will be able to add new functionality without changing the existing code

- **Bug fixes are exceptions** - you need to change the code

OPEN/CLOSED PRINCIPLE

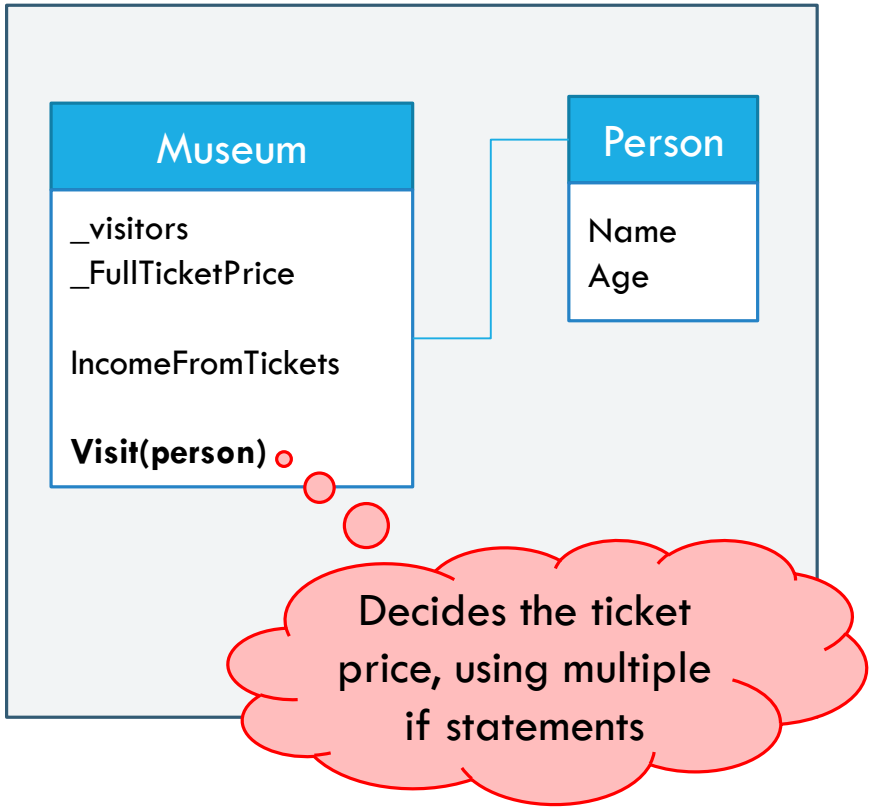
Source: https://www.exceptionnotfound.net/simply-solid-the-open-closed-principle/



MUSEUM SAMPLE

Open/Closed Principle

WHY IT ISN'T OPEN/CLOSED?



HOW TO APPLY OPEN/CLOSED PRINCIPLE?

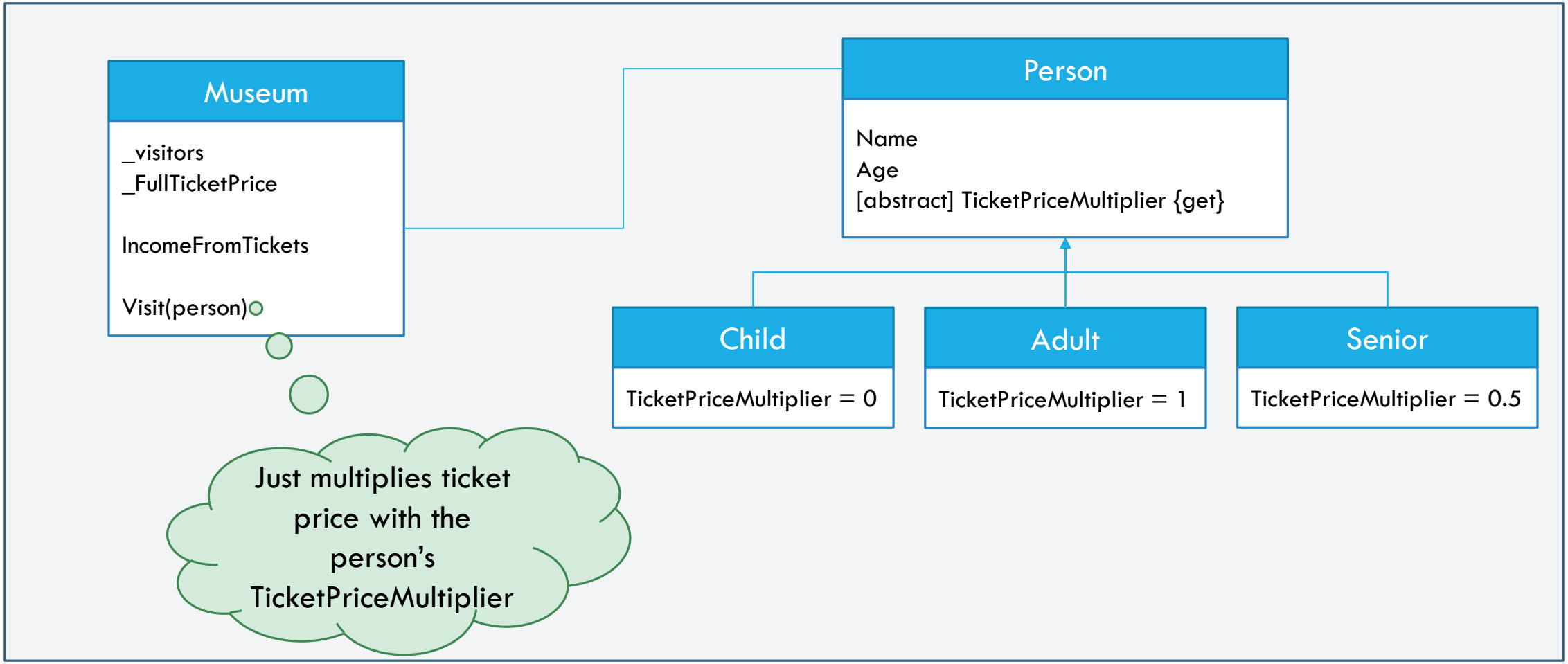
1. Use parameters
   - What is the price of a full ticket?

- **Use abstractions:**
  - Interfaces
  - Abstract clases

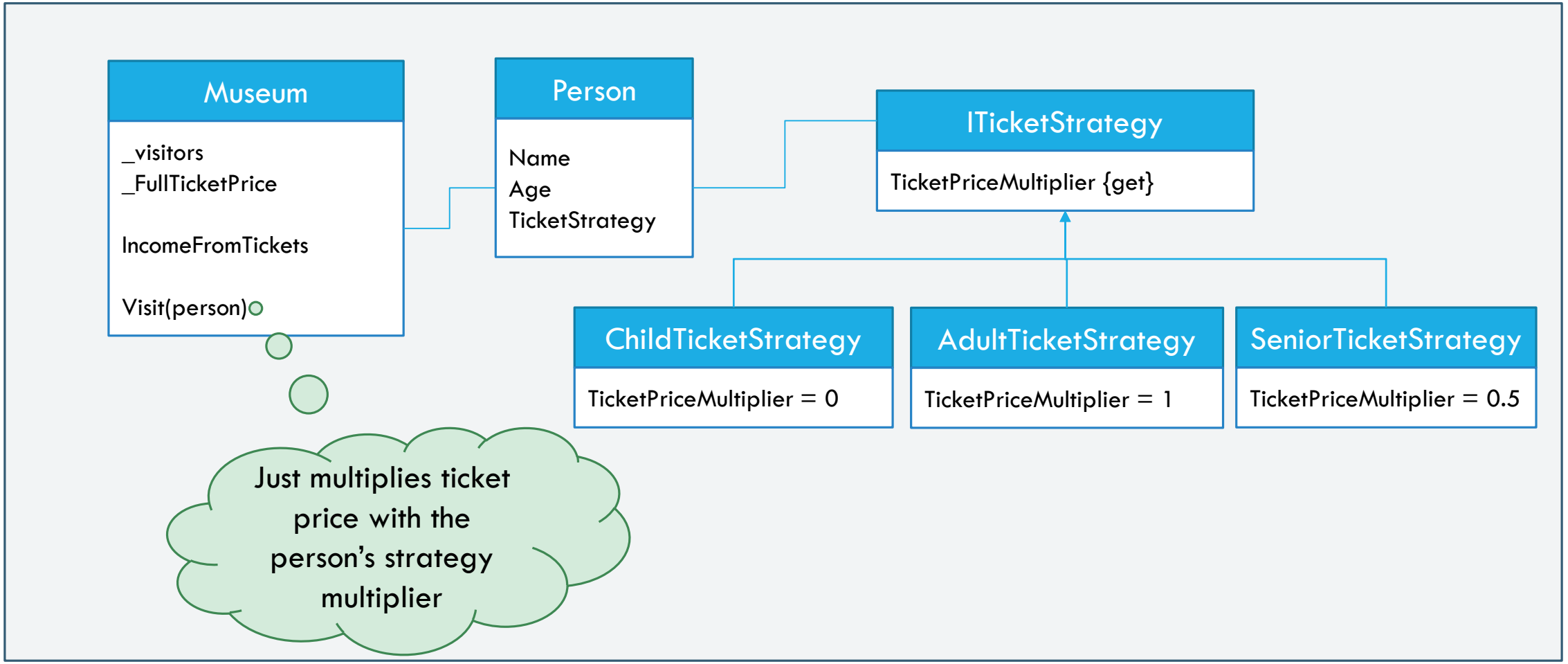
2. Inheritance / „Template Method" Pattern
   - Create a base class and add a method that will be implemented differently in each child class

3. Composition / „Strategy" Pattern
   - The class depends on an abstraction, that plugs in with the actual implementation

INHERITANCE – „TEMPLATE METHOD" PATTERN



COMPOSITION – „STRATEGY" PATTERN



TO USE OR NOT TO USE OPEN/CLOSED PRINCIPLE

- Advantages:
  - Less likely to introduce bugs
  - No need to recompile and deploy code that is already compiled and deployed
  - Simpler code
    - Usually uses if / switch cases, making it more difficult to follow and test

- Disadvantages:
  - Abstractions add complexity
  - Too many classes

RECAP

Code on GitHub: https://github.com/nadiacomanici/SolidPrinciples/tree/agilehub_2020_06_27