Karting Cars Kata

SOLID Kata based on **Racing Car Katas** by Emily Bache

https://github.com/emilybache/Racing-Car-Kata

What is SOLID?

- Popular set of design principles for OOP
- Defined in the early 2000s by Robert C. Martin (Uncle Bob).
- SOLID principles often relate to each other
- Mnemonic acronym for the following 5 design principles:
  - Single Responsibility Principle - SRP
  - Open/Closed Principle - OCP
  - Liskov Substitution Principle - LSP
  - Interface Segregation Principle - ISP
  - Dependency Inversion - DI

Why SOLID? (I)

- Your code will change, deal with it
- Your old project is going to come back, deal with it

Why SOLID? (II)

It is all about handling better your dependencies

Less Coupling

Degree to which a software entity (class, method or any other) is directly linked to another. This degree of coupling can also be seen as a degree of dependence.

More Cohesion

Measure in which two or more parts of a system work together to obtain better results than each part individually.

Why SOLID? (III)

Applying these principles leads to better quality code:

- Easier to maintain
- Easier to extend
- More robust

Single Responsibility Principle - Description

- A software entity should have one, and only one, reason to change
- A software entity has responsibility over a single part of the functionality

responsibility = reason to change

NOTE: Software entity: class, module, microservice...

Single Responsibility Principle - Goals

- High cohesion
- Reduce coupling
- Ease composition
- Avoid duplicity

Single Responsibility Principle - How to detect it?

- Before modifying your entity after a CR, ask yourself:
  - *What is the responsibility of this class/component/microservice?*
  - If your answer includes the word *and*, you're most likely breaking SRP
- Large setup needed in tests (multiple mocks, etc.)
- Too many merge conflicts or regressions in same file

NOTE: Top 10 modified files - *git log --pretty=format: --name-only | sort | uniq -c | sort -rg | head -10*

Single Responsibility Principle - How to fix it?

- Write Small entities with narrowed objectives
- For existing codebase:
  - Extract the methods belonging to one of the responsibilities and create a separate class for them
  - Continue doing this until you have only one responsibility per class

Open/Closed Principle - Description

- Software should be open to extension and closed to modification

- We should write our modules so that they can be extended without being modified

NOTE: Robert C. Martin considered this principle as the "the most important principle of object-oriented design"

Open/Closed Principle - Goals

- Easier to add new use cases
- Code is more readable

Open/Closed Principle - How to detect it?

- You directly work with a concrete implementation instead of an abstraction
- You have private methods that almost do the same thing
- You use ifs to control behavior (e.g. old way or new way)
- Abstraction used but concrete implementation checked to control flow

Open/Closed Principle - How to fix it?

- Do not depend on concrete implementations
- Promote the use of interfaces to enable you to adapt the functionality of your application without changing the existing code:
    - Strategy Pattern: extract interface + client using interface
    - Template Method Pattern: abstract class template + concrete impls

Liskov Substitution Principle - Description

- Objects of a superclass shall be replaceable with objects of its subclasses without breaking the application

- Objects of your subclasses should behave in the same way as those of your superclass

NOTE: This principle extends the OCP by focusing on the behavior of a superclass and its subtypes

Liskov Substitution Principle - Goals

- Code is easier to maintain
- Code is easier extend in the future
- Code is less error prone

Liskov Substitution Principle - How to detect it?

- You are handling differently subclasses of a parent class
- An overridden method does nothing or just throws an exception
- In your test cases, you can execute a specific part of your application with objects of all subclasses to make sure that none causes an error or significantly affects performance

NOTE: This is one of the most complicated principles to detect, as it may happen only at runtime

Liskov Substitution Principle - How to fix it?

- Don’t implement more stricter validation rules on input parameters than in the parent class

- Apply at the least the same rules to all output parameters as applied by the parent class

- It is usually caused by a bad abstraction: try using composition instead of inheritance

Interface Segregation Principle - Description

- Clients should not be forced to depend upon interfaces they don’t use

- Interfaces belong to clients using them, not to classes implementing them

- Better many client specific interfaces than one general purpose interface

Interface Segregation Principle - Goals

- Reduce the side effects and frequency of required changes by splitting the software into multiple, independent parts

- Avoids:
  - creating “fat” interfaces
  - forcing classes to implement methods they shouldn’t
  - polluting classes with lots of methods

- Easier to comply with OCP

- Easier to comply SRP

Interface Segregation Principle - How to detect it?

- Your class depends on an interface but uses only a subset of its methods
- Like with LSP:
  - You are handling differently subclasses of a parent class
  - An overridden method does nothing or just throws an exception

NOTE: It is better to have duplicated code than a bad abstraction

Interface Segregation Principle - How to fix it?

- Define interfaces according to clients using them, not to existing implementations

- Think about use cases before creating the interface, then create it with the required interactions

- Avoid Header Interfaces and promote Role Interfaces
    - **Header interface**: promoting all the public methods of a class to the interface
    - **Role interface**: defined by looking at a specific interaction between suppliers and consumers

NOTE: A supplier component will usually implement several Role Interfaces, one for each interaction

Dependency Inversion Principle - Description

- Depend upon abstractions, not on concretions

- Business rules (high level) should not change when implementation details (low level) change

- *Dep. Inversion != Dep. Injection*, the latter is one of the ways to achieve the former. Other ways:
  - factory pattern
  - service locator pattern: interface + initialcontext (+ cache) + servicelocator

Dependency Inversion Principle - Goals

- Ease implementation and dependencies substitution
- Decouple higher-level components from their dependency upon lower-level
- Ease testability (mocking, stubbing, etc.)
- Coupling between classes is more explicit

Dependency Inversion Principle - How to detect it?

- You need to reference a low-level module from a high-level module
- You find it hard to add or replace a low-level part of the application
- Hard to unit test a high-level component due to dependencies on concrete, low-level classes
- Search for the keyword *new* used to instantiate non basic classes

Dependency Inversion Principle - How to fix it?

- Inject dependencies:
    - Framework
    - Constructor
    - Setter

- Depend upon interfaces of these dependencies

- LSP as premise

References

- https://martinfowler.com
- https://pro.codely.tv
- http://coding-is-like-cooking.info
- https://stackify.com/solid-design-principles
- https://github.com/emilybache/Racing-Car-Katas
- https://medium.com/@ricartfe/principios-solid-89213a854528
- https://devonblog.com
- https://www.tutorialspoint.com/design_pattern/index.htm

Now it is Kata time!

Please follow these steps:

1. Clone the repo: *https://github.com/illuque/karting-car-katas.git*

2. Check README.md