**Princípios da Orientação a Objetos**

COESAO:

* “União harmônica entre uma coisa e outra”
  + Dentro de uma classe os atributos e métodos devem estar em harmonia, tratando da mesma coisa
* Classes não coesas tendem a crescer indefinidamente, o que as torna difíceis de manter!
* Uma classe coesa faz bem uma única coisa
* Classes coesas não devem ter várias responsabilidades

ENCAPSULAMENTO:

* “Incluir ou proteger alguma coisa em uma capsula
  + Blindar/proteger uma classe contra influencias externas que podem prejudicar, contra manipulações externas que podem prejudicar a consistências das informações
* Classes não encapsuladas permitem violação de regras de negócio, além de aumentarem o acoplamento
* Getters e setters não são formas eficientes de aplicar encapsulamento
* É interessante fornecer acesso apenas ao que é necessário em nossas classes
* O encapsulamento torna o uso das nossas classes mais fáceis e intuitivo

ACOPLAMENTO:

* “Ação de acoplar, agrupamento aos pares”
  + Quando temos dois componentes que estão interligados entre si, causando uma dependência entre ele, por exemplo, quando temos uma classe que faz a utilização de outra classe. Não necessariamente o acoplamento é algo ruim o problema é quando o acoplamento é muito forte!
* Um exemplo ruim é quando você manipula informações de uma classe fora dela
* Classes acopladas causam fragilidade no código da aplicação, o que dificulta sua manutenção
* Acoplamento é a dependência entre classes
* Acoplamento nem sempre é ruim, e que é impossível criar um sistema sem nenhum acoplamento
* Devemos controlar o nível de acoplamento na nossa aplicação (falaremos mais sobre isso)

**SOLID:**

Single Responsibility Principle:

* Classes/métodos/funções/módulos devem ter uma única responsabilidade bem definida;
* Segundo o Princípio de Responsabilidade Única (SRP), uma classe deve ter um e apenas um motivo para ser alterada;

Open Closed Principle:

* “Entidades de software (classes, módulos, funções, etc.) devem estar abertos para extensão, porém fechados para modificações”
* Cada classe deve conhecer e ser responsável por suas próprias regras de negócio;
* Devemos poder criar novas funcionalidades e estender o sistema sem precisar modificar muitas classes já existentes
* Uma classe que tende a crescer "para sempre" é uma forte candidata a sofrer alguma espécie de refatoração.

Liskov Substitution Principle:

* Embora a herança favoreça o reaproveitamento de código, ela pode trazer efeitos colaterais quando não utilizada da maneira correta;
* O Princípio de Substituição de Liskov (LSP) diz que devemos poder substituir classes base por suas classes derivadas em qualquer lugar, sem problema.

Interface Segregation Principle:

* Uma classe não deveria ser forçada a depender de métodos que não utilizara

Dependency Inversion Principle:

* Abstrações não devem depender de implementações. Implementações devem depender de abstrações
* Caso uma determinada implementação mude, não seremos afetados, pois dependemos apenas de sua interface.