Solid		
S - Single Responsibility Principle	(Princípio da Responsabilidade Única)	Uma classe deve ter apenas uma única responsabilidade. Em outras palavras, uma classe deve ter apenas um motivo para mudar.
O - Open/Closed Principle	(Princípio Aberto/Fechado)	Uma classe deve estar aberta para extensão, mas fechada para modificação. Isso significa que você deve ser capaz de adicionar novas funcionalidades a uma classe sem precisar alterar seu código existente.
L - Liskov Substitution Principle	(Princípio da Substituição de Liskov)	Os objetos de uma classe derivada devem poder ser substituídos pelos objetos da classe base sem afetar o comportamento do programa. Em outras palavras, as classes filhas devem ser capazes de substituir as classes pai sem quebrar a funcionalidade do sistema.

Solid (cont)		
I - Interface Segregation Principle	(Princípio da Segregação de Interfaces)	Os clientes de uma classe não devem ser forçados a depender de interfaces que eles não usam. Em vez disso, as interfaces devem ser separadas em grupos menores e mais específicos.
D - Dependency Inversion Principle	(Princípio da Inversão de Dependência)	Os módulos de alto nível não devem depender de módulos de baixo nível. Em vez disso, ambos devem depender de abstrações. Isso é feito através do uso de interfaces ou classes abstratas, que definem os contratos que as classes concretas devem seguir. Isso permite que as classes de alto nível sejam mais flexíveis e reutilizáveis, porque elas não dependem diretamente das implementações das classes de baixo nível.

O SOLID é uma forma de design de software que foi criada por Robert C. Martin para ajudar os desenvolvedores a criar sistemas melhores. Ele é composto por cinco princípios que ajudam a tornar o código mais robusto, flexível e fácil de manter. Esses princípios são muito usados na indústria de software para criar sistemas de alta qualidade.

Design Patterns	
Singleton	Garante que apenas uma instância de uma classe seja criada e fornecida em toda a aplicação. Ele é comumente usado em situações onde há uma única instância de um recurso que precisa ser compartilhado por toda a aplicação.
Factory Method	Fornece uma interface para criar objetos em uma superclasse, mas permite que as subclasses alterem o tipo de objetos que serão criados. É útil quando você não sabe de antemão que tipo de objeto precisará criar.
Observer	Permite que um objeto observe outro objeto e seja notificado quando ocorrerem alterações nesse objeto. Ele é frequentemente usado em situações onde há uma relação entre objetos que precisam estar cientes das mudanças uns dos outros.
Decorator	Permite que você adicione comportamentos adicionais a um objeto dinamicamente. Ele é útil quando você deseja adicionar funcionalidade a um objeto existente sem modificar sua estrutura.
Facade	Padrão de design que fornece uma interface simplificada para um conjunto complexo de classes. Ele encapsula a complexidade do sistema subjacente e fornece uma interface única e unificada para o cliente.

Design Patterns (Padrões de Projeto) são soluções comuns para problemas recorrentes que surgem durante o desenvolvimento de software. Eles são um conjunto de boas práticas e soluções testadas e comprovadas que podem ser aplicadas em diferentes projetos de software para resolver problemas específicos.