SOLID/GRASP

DECORATOR (Recuperação)

SOLID

- **Single Responsibility Principle (SRP)**: O objeto pode manter sua responsabilidade única, pois as funcionalidades adicionais são encapsuladas nos seus decoradores
- **Open/Closed Principle (OCP)**: Permite que novas funcionalidades sejam adicionadas a um objeto (classe) sem alterar sua implementação, somente a de seus decoradores, sendo assim aberto para extensão e fechado para modificação
- **Liskov Substitution Principle (LSP)**: Os decoradores devem poder substituir os objetos que estão decorando sem mudar o comportamento do programa, pois os decoradores são subtipo da classe original

GRASP

- **Alta Coesão:** Separa as responsabilidades em decoradores, onde cada um adiciona uma funcionalidade específica ao objeto original, resultando em classes mais coesas
- **Acoplamento Fraco:** Permite adicionar novas funcionalidades a um objeto sem modificar a classe original do objeto ou outras classes que interagem com ele, reduzindo a dependência entre classes

ADAPTER (Recuperação)

SOLID

- **Single Responsibility Principle (SRP):** O objeto pode manter sua responsabilidade única, pois a lógica necessária para adaptar uma interface para outra será encapsulada no adapter
- **Open/Closed Principle (OCP):** Permite que funcionalidades de adaptação sejam adicionadas ao sistema sem modificar o código existente das classes a serem adaptadas

.

GRASP

- **Alta Coesão:** Separa a lógica de conversão no Adapter, assim cada interface pode se concentrar em suas responsabilidades específicas, mantendo a coesão
- **Acoplamento Fraco:** Permite que duas interfaces incompatíveis trabalhem juntas sem modificar suas implementações e sem depender uma da outra

SINGLETON (Loteria)

SOLID

- **Não favorece nenhum princípio SOLID**

.

GRASP

- **Não favorece nenhum padrão GRASP**

OBSERVER (Aposta-Loteria)

SOLID

- **Single Responsibility Principle (SRP)**: O objeto pode manter sua responsabilidade única, pois a lógica de notificação estará apenas no objeto observado e os observadores apenas lidam com as notificações
- **Dependency Inversion Principle (DIP)**: O objeto observado e os observadores dependem de interfaces ou classes abstratas, e não de implementações específicas

GRASP

- **Acoplamento Fraco:** O objeto observado não precisa saber detalhes sobre os observadores, apenas notificá-los através de uma interface comum

STRATEGY (Calculo)

SOLID

- **Single Responsibility Principle (SRP)**: Separa estratégias em classes, sendo cada estratégia responsável por implementar um algoritmo
- **Open/Closed Principle (OCP)**: Permite que novas estratégias sejam adicionadas sem modificar o contexto que usa essas estratégias
- **Liskov Substitution Principle (LSP)**: Permite que diferentes estratégias tenham implementações substituídas e diferentes umas pelas outras, desde que implementem a mesma interface ou herdem da mesma classe base
- **Dependency Inversion Principle (DIP)**: Depende de abstrações (interfaces ou classes abstratas) em vez de implementações concretas

GRASP

- **Alta Coesão:** Agrupa algoritmos em classes de estratégia separadas, tendo cada estratégia responsabilidade única
- **Acoplamento Fraco:** O contexto que utiliza as estratégias não conhece os detalhes internos das estratégia

Exemplos da refatoração dos diagramas e código devido aos princípios SOLID e GRASP:
Calculo:
● Open/Closed Principle (OCP) e Alta Coesão -> Conteve a adição de novos métodos que seriam comuns a todas as subclasses, servindo como uma interface de manipulação abstrata de cada instância, que por vez, cada subclasse pode ou não conter métodos adicionais, desde que não mude implementação anteriores. Sendo assim, usado em “calcularMultiplicador():float”.

Loteria/Avião/Roleta:
● Liskov Substitution Principle (LSP) -> Toda subclasses devem implementar os métodos de seu “pai”, sendo assim,foi necessário a adição de métodos como “sorteia():int”, “iniciarJogo():void”, entre outros, que deveriam ser comuns a todos os games que herdam da classe em questão.

● Single Responsibility Principle (SRP) -> Em “calcularSaldo(multiplicador:float):void” foi usado tal princípio para subdividir a tarefa aos especialista da informação, desta forma, por exemplo, a classe “Jogos” possui como método adicional “calcularMultiplicador():int”, que retorna parcialmente a informação necessária para calcular o saldo da aposta.

Linha de Produção de Software (LPS)