Princípios SOLID

www.cotiinformatica.com.br

O que é SOLID?

Os princípios **SOLID** devem ser aplicados no desenvolvimento de software de forma que o software produzido tenha as seguintes características:

- **Seja fácil de manter, adaptar e se ajustar às constantes mudanças exigidas pelos clientes;**
- **Seja fácil de entender e testar;**
- **Seja construído de forma a estar preparado para ser facilmente alterado com o menor esforço possível;**
- **Seja possível de ser reaproveitado;**
- **Exista em produção o maior tempo possível;**
- **Que atenda realmente as necessidades dos clientes para o qual foi criado;**

O que é SOLID?

Os princípios **SOLID** para programação e design orientados a objeto são de autoria de *Robert C. Martin* (*mais conhecido como Uncle Bob*) e datam do início de 2000. A palavra **SOLID** é um acróstico onde cada letra significa a sigla de um princípio, são eles: **SRP, OCP, LSP, ISP e DIP**

SRP	The Single Responsibility Principle Principio da Responsabilidade Única	**Uma classe deve ter um, e somente um, motivo para mudar.** **A class should have one, and only one, reason to change.**
OCP	The Open Closed Principle Princípio Aberto-Fechado	Você deve ser capaz de estender um comportamento de uma classe, sem modificá-lo. You should be able to extend a classes behavior, without modifying it.
LSP	The Liskov Substitution Principle Princípio da Substituição de Liskov	As classes derivadas devem poder substituir suas classes bases. Derived classes must be substitutable for their base classes.
ISP	The Interface Segregation Principle Princípio da Segregação da Interface	Muitas interfaces específicas são melhores do que uma interface geral Make fine grained interfaces that are client specific.
DIP	The Dependency Inversion Principle Princípio da inversão da dependência	Dependa de uma abstração e não de uma implementação. Depend on abstractions, not on concretions.

O que é SOLID?

Letra	Sigla	Nome	Definição
S	**SRP**	Principio da Responsabilidade Única	*Uma classe deve ter um, e somente um, motivo para mudar.*
O	**OCP**	Princípio Aberto-Fechado	*Você deve ser capaz de estender um comportamento de uma classe, sem modificá-lo.*
L	**LSP**	Princípio da Substituição de Liskov	*As classes derivadas devem ser substituíveis por suas classes base.*
I	**ISP**	Princípio da Segregação da Interface	*Muitas interfaces específicas são melhores do que uma interface única.*
D	**DIP**	Princípio da inversão da dependência	*Dependa de uma abstração e não de uma implementação.*

S

SRP

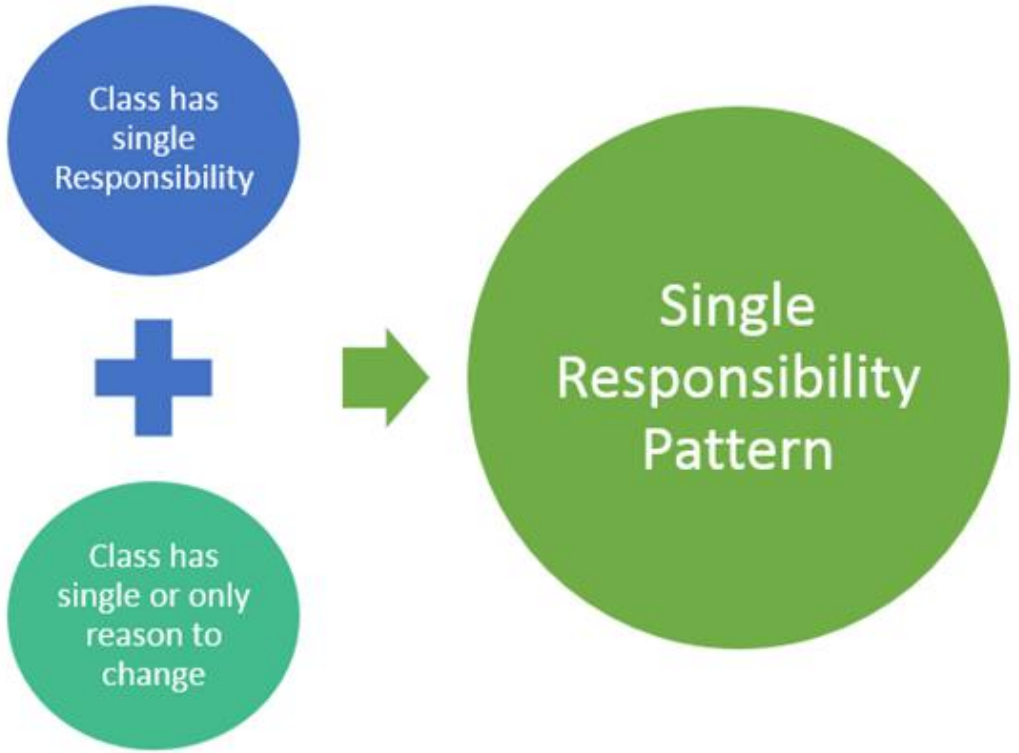
Single Responsability Principle

PRINCÍPIO DA RESPONSABILIDADE ÚNICA

www.cotiinformatica.com.br

Princípio da Responsabilidade Única

Cada
responsabilidade
deve ser uma **classe**
separada porque cada
responsabilidade é eixo
para a mudança



Princípio da Responsabilidade Única

“Uma classe só deveria ter um único motivo para mudar.”

A Audiência

Determinar a única responsabilidade de uma classe ou módulo, é muito mais complexo que, simplesmente, verificar em uma lista pré-determinada. Por exemplo, uma dica para encontrarmos as razões para mudanças é verificar o audiência da nossa classe. Os usuários da aplicação ou sistema que desenvolvemos é que requisitarão mudanças para ela. Aqueles que usam que pedirão mudanças. Eis alguns módulos e suas possíveis audiências.

Módulo de Persistência - Pode incluir os DBAs e arquitetos de software;
Módulo de Relatório - Pode incluir os contadores, secretários e administradores;
Módulo de Computação de Pagamento para um Sistema de Pagamento - Pode incluir os advogados, administradores e contadores;
Módulo de Busca de Livros para um Sistema de Administração de Biblioteca - Pode incluir os bibliotecários e/ou os próprios clientes.

Princípio da Responsabilidade Única

Uma responsabilidade é um conjunto de funções que serve a um ator em particular. (Robert C. Martin)

Atores e Papéis

Associar pessoas reais a todas os papéis existentes pode ser difícil. Em uma pequena empresa, uma única pessoa pode ter de satisfazer vários papéis, enquanto em uma empresa maior, pode ter várias pessoas alocadas para um mesmo papel. Assim, faz bastante sentido pensar sobre esses papéis. Mas os papéis, em si, são um pouco difíceis de definir. O que é um papel? Como os encontramos? É muito mais fácil pensar em pessoas realizando determinados papéis e associa-los à nossa audiência.

Assim, se nossa audiência define os motivos para as mudanças, os atores definem a audiência. Isso nos ajuda a reduzir os conceitos de pessoas como "John o arquiteto" em Arquitetura, ou "Maria a administradora" em Administração.

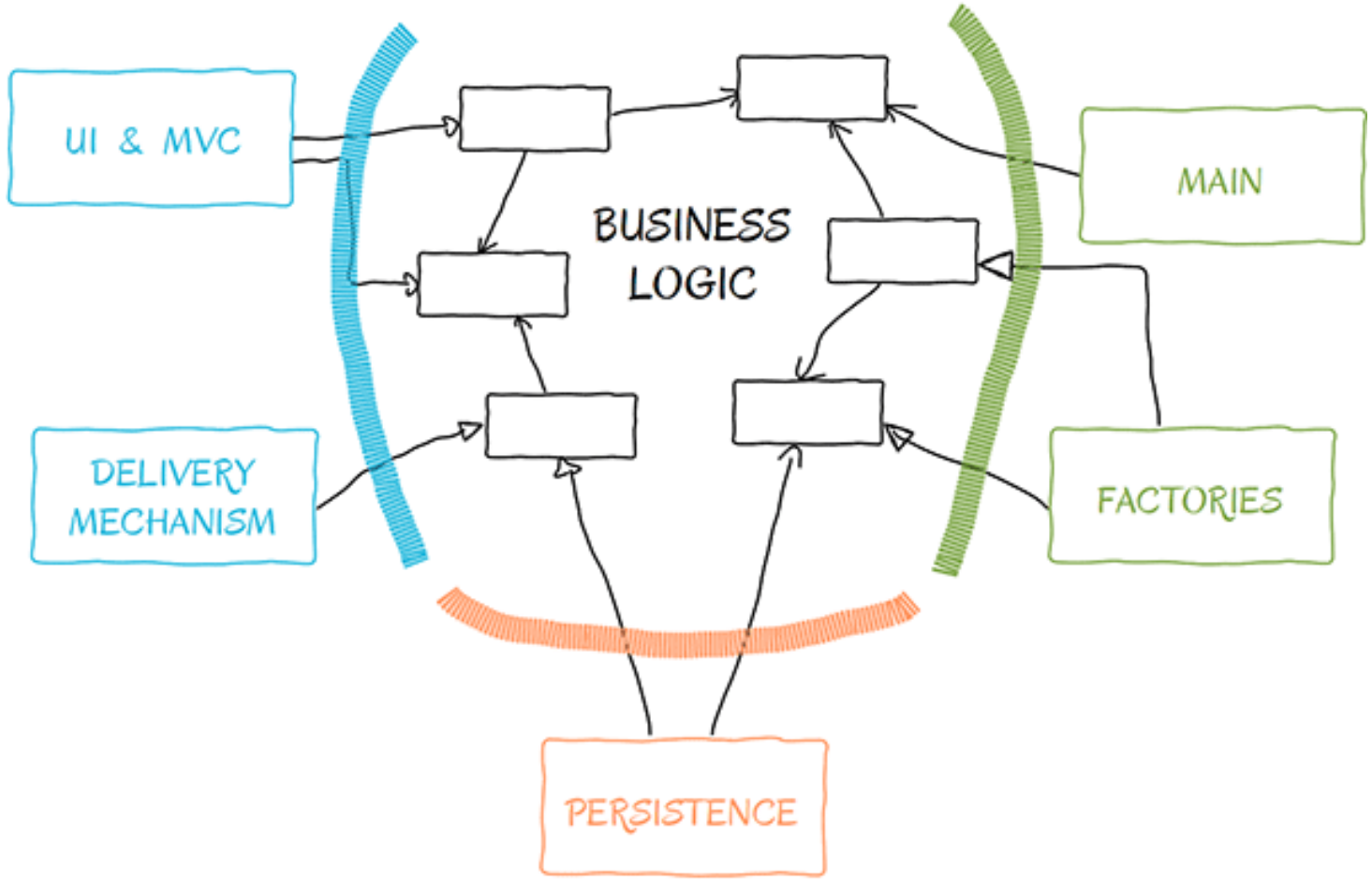
Princípio da Responsabilidade Única

O ator de uma responsabilidade é a única fonte de mudança para aquela responsabilidade.
(Robert C. Martin)

Fonte da Mudança

Seguindo esse raciocínio, os atores tornam-se a fonte da mudança para a família de funções que os servem. De acordo com que suas necessidades mudam, aquela família de funções também deve mudar para adaptar-se às suas necessidades.

Princípio da Responsabilidade Única

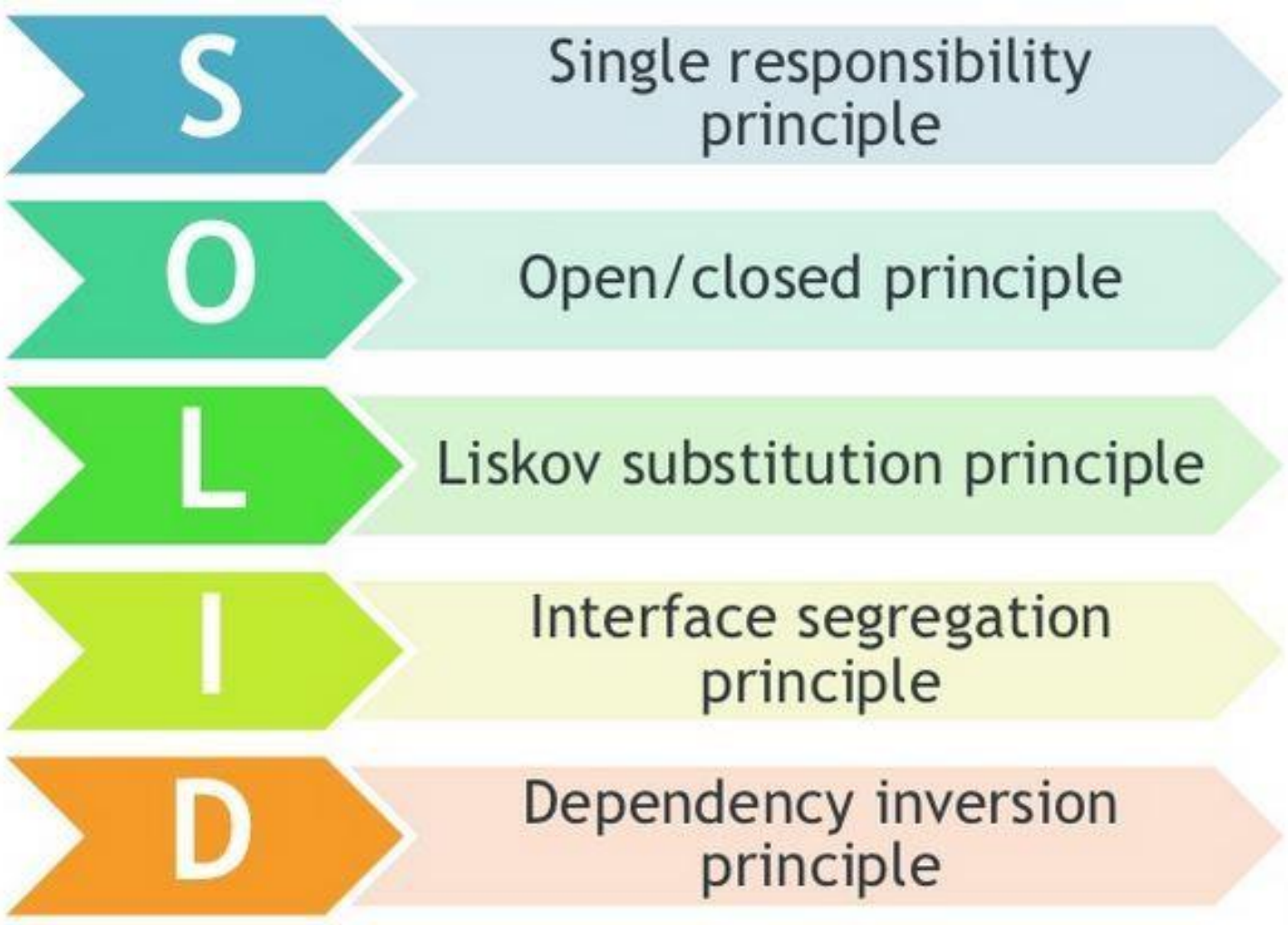


Se analisarmos a imagem, veremos como o Princípio da Responsabilidade Única é respeitado. A criação de objetos está separada na direta, em Fábricas (Factories) e no ponto de entrada principal (Main) de nossa aplicação, um ator uma responsabilidade. A persistência também é levada em conta, na parte de baixo. Um módulo separado para um responsabilidade diferente. Finalmente, na esqueda, nós temos a apresentação ou o mecanismo de entrega, se assim preferir, no padrão MVC ou qualquer outro tipo de interface de usuário. A SRP foi respeitada novamente. Só nos resta descobrir o que fazer na nossa lógica de negócio.

Princípio da Responsabilidade Única

O Princípio da Responsabilidade Única deve sempre ser considerado quando for codificar seus aplicativos. Projetos de classes e módulos são bastante afetados pela SRP, o que leva a um projeto com baixo acoplamento, com dependências menores e em menor quantidade.

Assim, toda vez que perceber que um módulo ou classe começaram a mudar por inúmeras razões diferentes, não hesite, tome os passos necessários para respeitar a SRP, entretanto, não se deixe levar por ela, uma vez que otimizações prematuras podem enganar você.



O

OCP

Open / Closed Principle

PRINCÍPIO DO ABERTO E FECHADO

www.cotiinformatica.com.br

Princípio do Aberto e Fechado

*“Entidades de software (classes, módulos, funções, etc.) devem ser **abertas** para extensão mas **fechadas** para modificação.”*

Quando eu precisar estender o comportamento de um código, eu crio código novo ao invés de alterar o código existente.

Princípio do Aberto e Fechado

Extensibilidade

É uma das chaves da orientação a objetos, quando um novo comportamento ou funcionalidade precisar ser adicionado é esperado que as existentes sejam estendidas e e não alteradas, assim o código original permanece intacto e confiável enquanto as novas são implementadas através de extensibilidade.

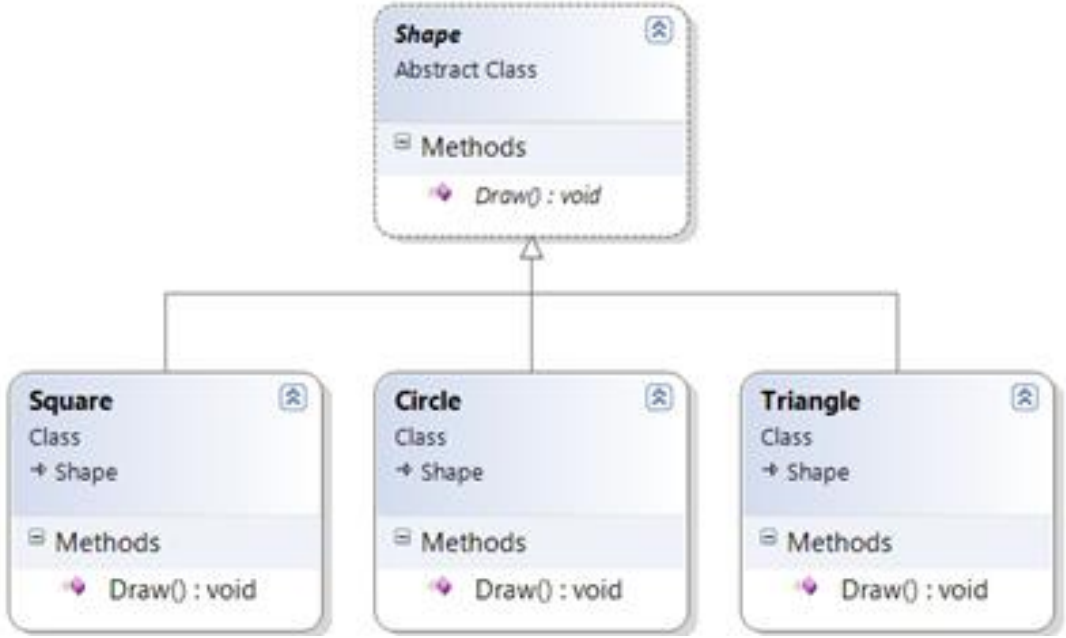
Criar código extensível é uma responsabilidade do desenvolvedor maduro, utilizar design duradouro para um software de boa qualidade e manutenibilidade.

Abstração

Quando aprendemos sobre orientação a objetos com certeza ouvimos sobre abstração, é ela que permite que este princípio funcione. Se um software possui abstrações bem definidas logo ele estará aberto para extensão.

Princípio do Aberto e Fechado

Mas por que na prática eu fecho para modificações uma classe?
A razão é simples: porque assim eu posso desenvolver meu software como se fosse em camadas.
Estando uma camada muito bem escrita e bem definida, eu tenho certeza de que todas as classes derivadas também funcionarão bem.
As classes derivadas na prática poderiam apenas usar os métodos fechados e acrescentar novos comportamentos ao sistema conforme novas necessidades fossem surgindo.



L

LSP

Liskov
Substitution
Principle

PRINCÍPIO DA SUBSTITUIÇÃO DE LISKOV

www.cotiinformatica.com.br

Princípio da Substituição de Liskov

“Classes derivadas devem poder ser substituídas por suas classes base”

"Se você pode invocar um método q() de uma classe T (base), deve poder também invocar o método q() de uma classe T'(derivada) que é derivada com herança de T (base)."

Em outras palavras: **"*Uma classe base deve poder ser substituída pela sua classe derivada."***

Princípio da Substituição de Liskov

O que isso quer dizer afinal?

Significa dizer que classes derivadas devem poder substituídas por suas classes base e que classes base podem ser substituídas por qualquer uma das suas subclasses. Uma subclasse deve sobrescrever os métodos da superclasse de forma que a funcionalidade do ponto de vista do cliente continue a mesma.

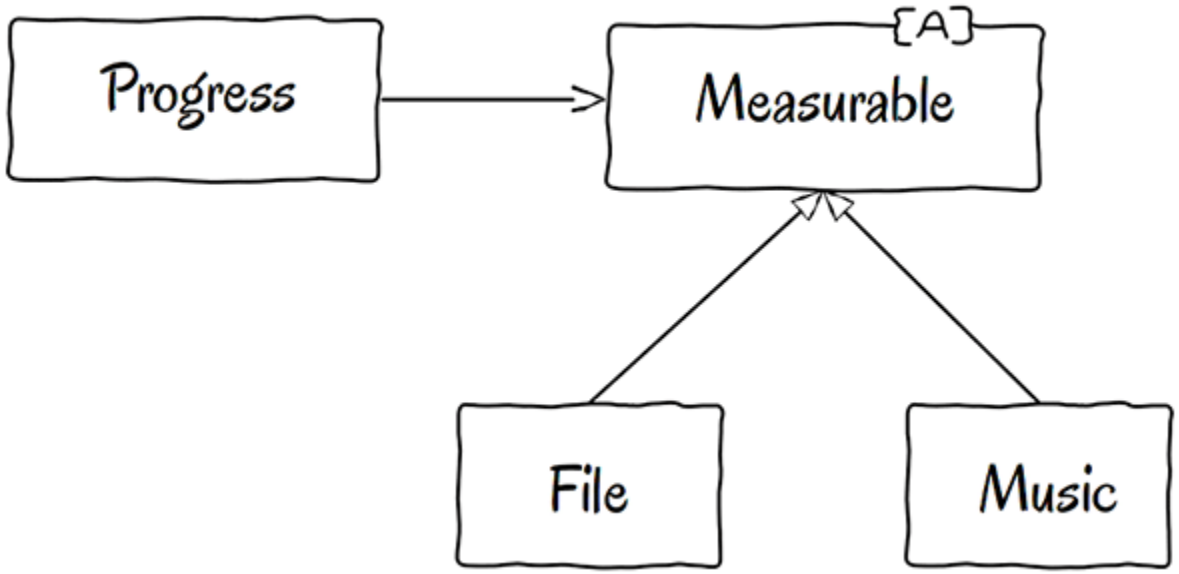
O princípio da substituição de Liskov nos mostra que devemos tomar cuidado ao fazer uso da herança, devemos verificar se o polimorfismo faz mesmo sentindo, ou seja, se qualquer subclasse pode ser utilizada no lugar da superclasse.

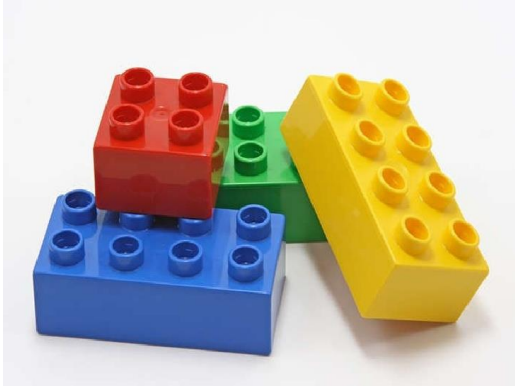
Caso não, significa dizer que a herança está sendo utilizada de forma inadequada.

Princípio da Substituição de Liskov

O WardsWiki formula e responde a seguinte pergunta: Por que o Princípio da Substituição de Liskov é importante?

Porque se não, as hierarquias de classe seriam uma bagunça. Pois podem ocorrer comportamentos estranhos quando uma instância da subclasse for passada como parâmetro para um método.
Porque se não, testes de unidade para a superclasse nunca teria sucesso para uma subclasse.
A solução: Temos disponíveis várias técnicas para resolver ou evitar o problema de violação do princípio de Liskov, onde podemos usar alguns padrões de projeto no nosso código e principalmente o Composition instead Inheritance (evite herança, prefira composição).





I

ISP

Interface Segregation Principle

PRINCÍPIO DA SEGREGAÇÃO DE INTERFACES

www.cotiinformatica.com.br

Princípio da Segregação de Interfaces

O Princípio da Segregação de Interface trata da coesão de interfaces e diz que **clientes não devem ser forçados a depender de métodos que não usam**.

O Princípio da Segregação de Interface nos alerta quanto à dependência em relação a "interfaces gordas", forçando que classes concretas implementem métodos desnecessários e causando um acoplamento grande entre todos os clientes.

Ao usarmos interfaces mais específicas, quebramos esse acoplamento entre as classes clientes, além de deixarmos as implementações mais limpas e coesas.

Princípio da Segregação de Interfaces

"Faça interfaces de fina granularidade, que sejam específicas para quem vai utilizá-las";

Muitas interfaces específicas são melhores
do que uma única interface geral.

Esse principio ajuda a evitar a criação de *fat interfaces*(interfaces gordas), termo utilizado para interfaces com mais funcionalidades do que o necessário. Classes que implementam uma interface assim não são coesas. As interfaces podem ser divididas em grupos de métodos, e cada grupo atende uma conjunto diferentes de classes, cada classe pode implementar apenas as funcionalidades que fazem sentido;

Uma dos indicadores para identificar a quebra deste principio é no seguinte cenário você ter uma interface com 4 funcionalidades porém ao implementar essa interface em uma classe faz sentido todas os métodos. Porém em outra classes uma funcionalidade ou outra não faz sentido ser implementada.

Princípio da Segregação de Interfaces

Uma boa dica para evitar a quebra desse principio é sempre que adicionar um método em uma interface, analise quem implementa essa interface e se aquele método faz sentido para todas as classes que implementam. Se tiver sentido adicione nessa interface sem nenhum problema, caso não faça sentido crie um outra interface(ou verifique se faz sentido em outra interface já existente) para adicionar o método que você precisa implementar.

Indicadores de quebra do principio:

- Métodos de classes, implementados com base em uma interface, retornando valores padrões ou jogando exceções
- Implementações de métodos que não fazem sentido para a classe
- Pouco sentido nas interfaces que existem no sistema
- Muita alterações no código para adicionar um novo método na interface.
- Ao chamar um método em uma classe não ter certeza se ele foi realmente implementado(isso acontece e bastante)

D

DIP

Dependency Inversion Principle

PRINCÍPIO DA INVERSÃO DE DEPENDÊNCIA

www.cotiinformatica.com.br


Rosin Core Solder

Princípio da Inversão de Dependência

Módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender de abstrações;

Abstrações não devem depender de detalhes. Detalhes devem depender de abstrações.

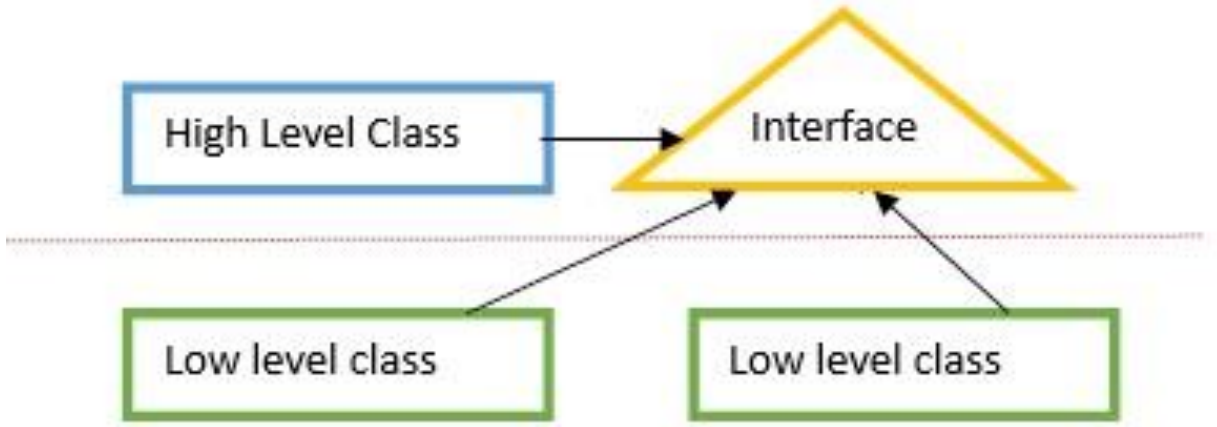
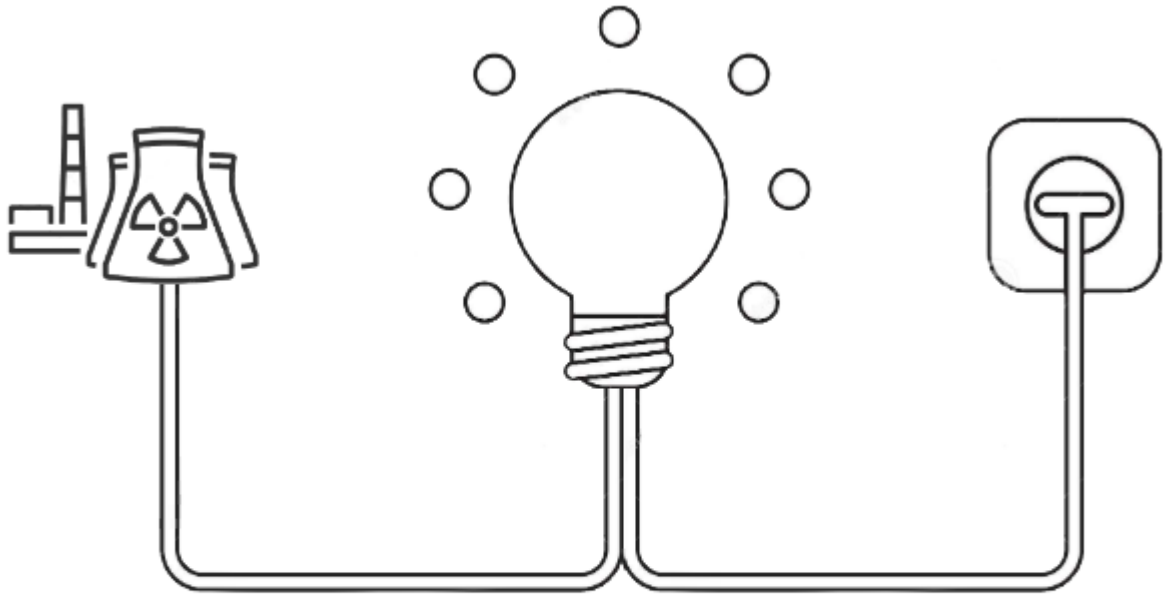


fig: Interface was defined by high level class

Princípio da Inversão de Dependência

Inverter a dependência faz com que um cliente não fique frágil a mudanças relacionadas a detalhes de implementação.
Isto é, alterar o detalhe não quebra o cliente. Além disso, o mesmo cliente pode ser reutilizado com outro detalhe de implementação.

O Princípio da Inversão de Dependência é um dos pilares para uma boa arquitetura de software, focada na resolução do problema e flexível quanto a detalhes de implementação, como bancos de dados, serviços web, leitura/escrita de arquivos, etc.
Esta separação de interface e implementação em componentes distintos é um padrão conhecido por **Separated Interface**, como catalogado no livro PoEAA, do Martin Fowler.



Dependency Inversion Principle

When knowing how things work becomes a burden

Princípio da Inversão de Dependência

Outro exemplo bem comum deste padrão está no uso do padrão **Repositório**. Neste caso, aplicamos o DIP para que nosso domínio dependa de uma abstração do Repositório, ficando totalmente isolado de detalhes sobre persistência

O Princípio da Inversão de Dependência é um princípio essencial para um bom design orientado a objetos, ao passo que o oposto leva a um design engessado e procedural.
Identificar abstrações e inverter as dependências garantem que o software seja mais flexível e robusto, estando melhor preparado para mudanças.

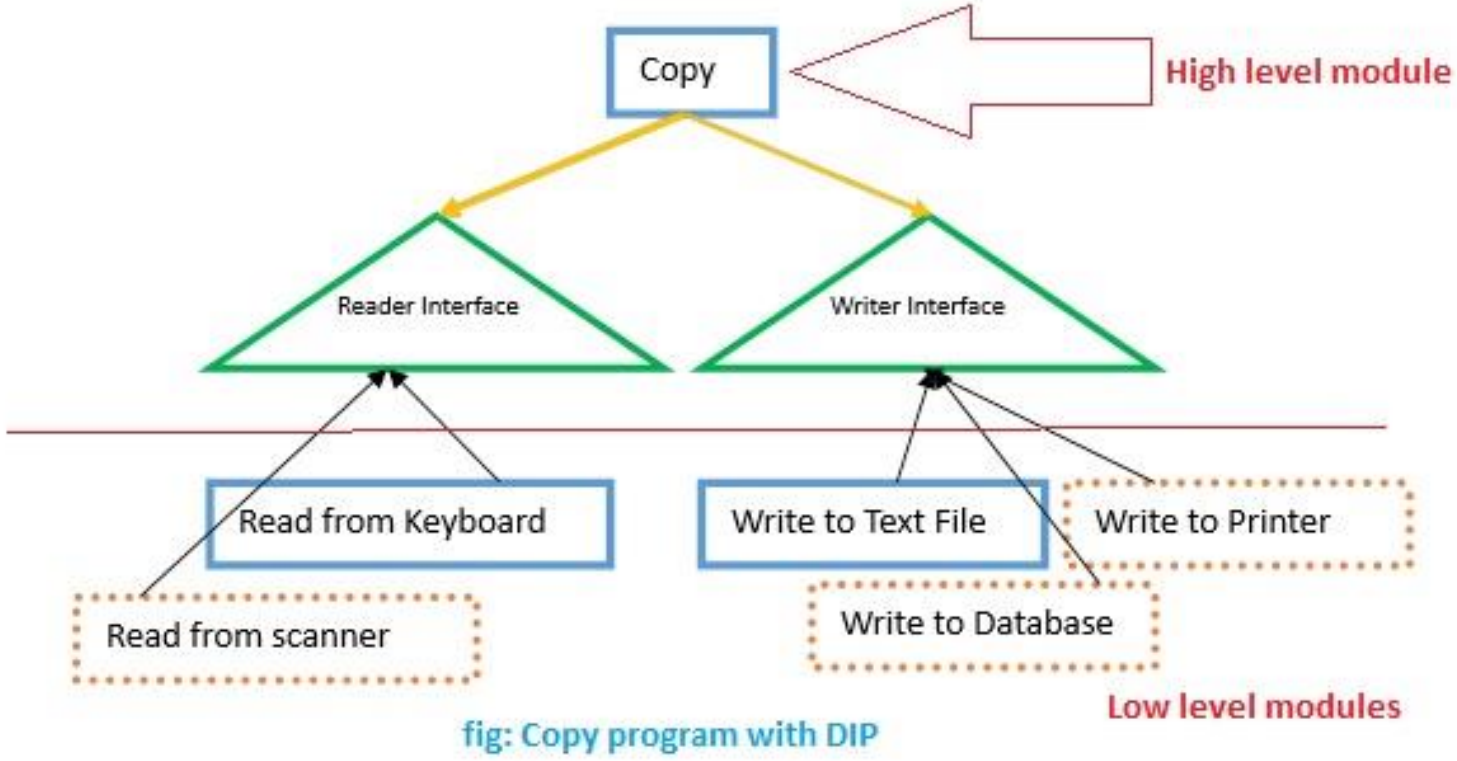


fig: Copy program with DIP

www.cotiinformatica.com.br