
SOLID

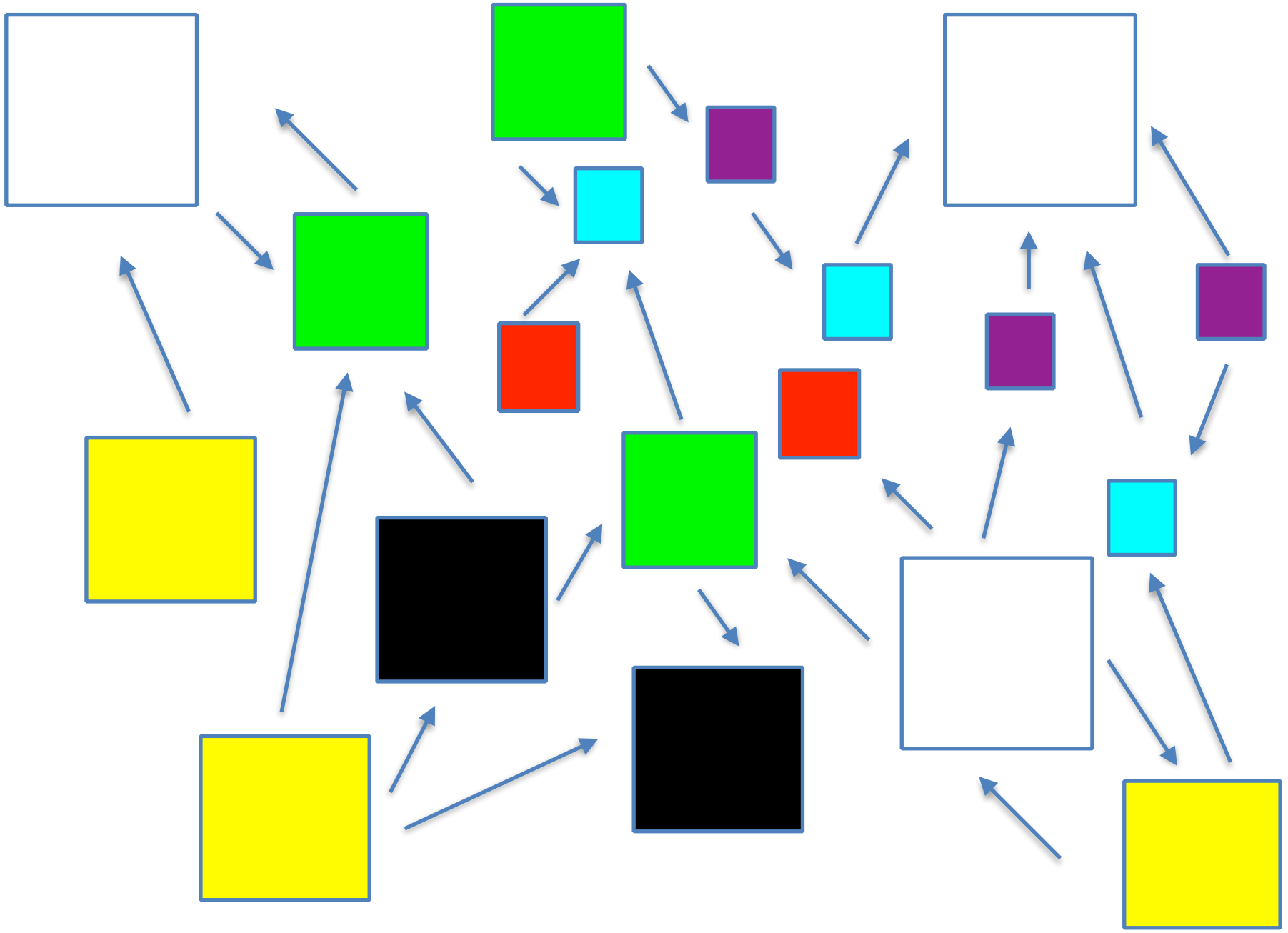
No final dos anos 80, Bob Martin reuniu cerca de 9 princípios em um artigo, que na verdade foi uma resposta a outro artigo sobre orientação a objetos, que acabou servindo de base para os SOLID principles

SOLID é a reunião de 5 princípios de desenvolvimento de software orientado a objetos que são a base para criar um design capaz de tolerar mudanças ao longo do tempo, ser fácil de entender e reusar

Rigidity: É difícil de mudar porque cada alteração afeta outras partes da aplicação, forçando outras alterações, causa longos builds, tempo de execução de testes e sensibilidade à mudança

Fragility: Quando você faz uma mudança, quebra outras partes da aplicação, ou seja, existe um alto acoplamento, causa falhas inesperadas e difíceis de detectar

Immobility: É difícil de reusar porque o código-fonte está emaranhado, dentro de outros lugares, deixando o sistema difícil de modularizar



Princípios do SOLID

Single Responsibility
Open/Closed
Liskov Substitution
Interface Segregation
Dependency Inversion

Single Responsibility

Devemos separar as coisas que mudam por motivos diferentes e nesse contexto a palavra responsabilidade significa motivo para mudar

"The SRP is one of the simplest of the principle, and one of the hardest to get right. If a class has more then one responsibility, then the responsibilities become coupled and changes to one responsibility may affect others. Finding and separating those responsibilities from one another is much of what software design is really about"

Robert C. Martin

Open/Closed

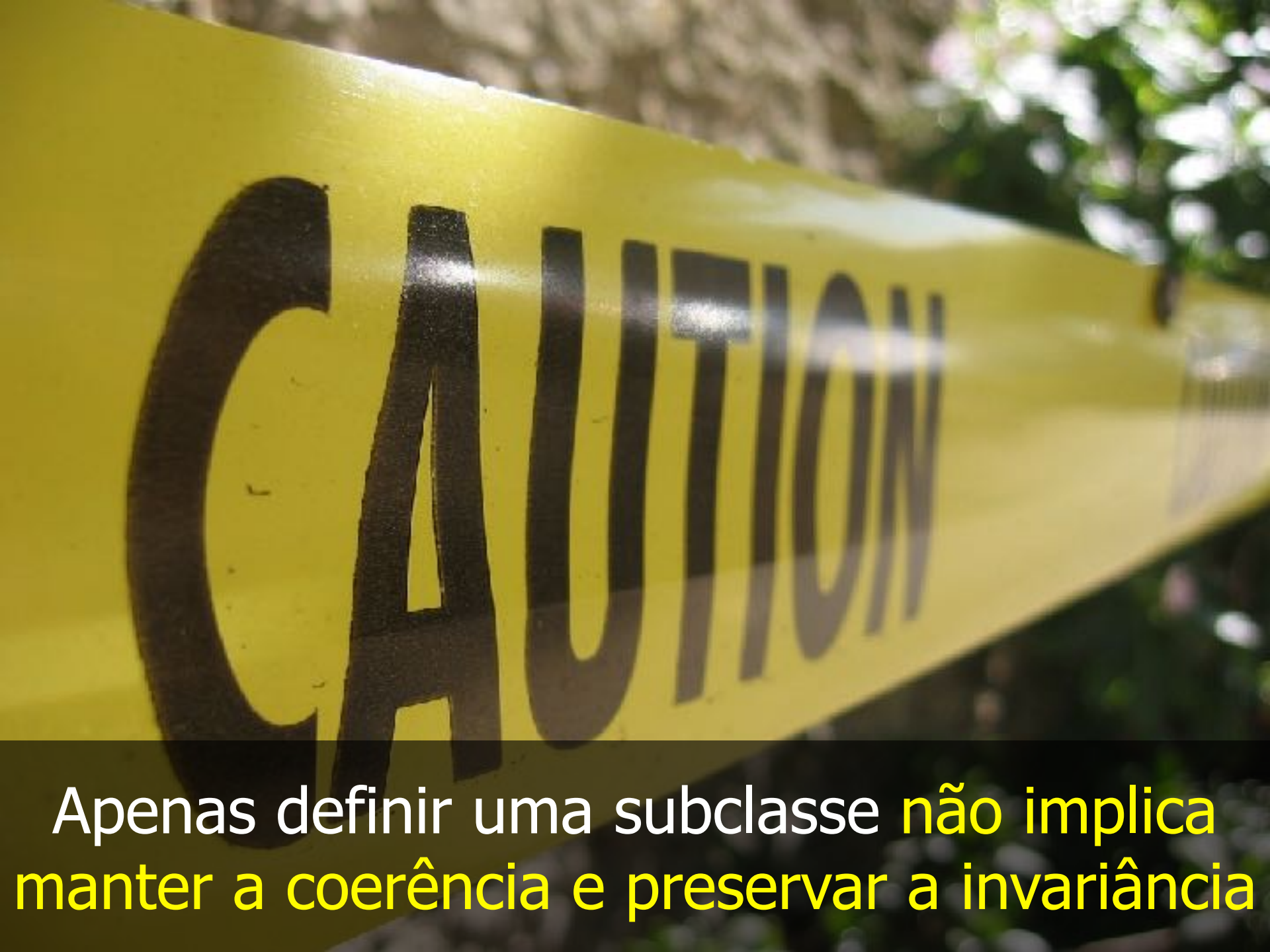
Os componentes da arquitetura devem estar abertos para extensão e fechados para modificação

A maior parte dos design patterns como o Abstract Factory, Builder, Adapter, Proxy, Chain of Responsibility, Iterator, State, Strategy e Template Method respeita o Open/Closed Principle por ser polimórfico, ou seja, o objetivo é criar pontos de extensão

Fechado para mudança não significa que o código-fonte não pode mudar se for necessário

Liskov Substitution

Se S é subclasse de T então objetos do tipo T podem ser substituídos por objetos do tipo S sem quebrar o funcionamento do programa



Apenas definir uma subclasse não implica manter a coerência e preservar a invariância

A proposta da Barbara Liskov foi justamente garantir que as subclasses possam ser intercambiadas sem causar problemas durante a execução do programa

Preconditions cannot be strengthened in subtype

As operações definidas na subclasse devem aceitar no mínimo as mesmas entradas da superclasse, ou mais

Exemplo: Se a interface permite um parâmetro que é um número inteiro, caso uma instância limite a número negativos ela estará fortalecendo as pré-condições, causando possíveis incompatibilidades

Postconditions cannot be weakened in the subtype

As operações definidas na subclasse devem produzir no mínimo o mesmo tipo de retorno da superclasse, ou algum que seja mais específico

Exemplo: Se a superclasse limita o retorno aos números positivos a subclasse não deve enfraquecer a regra e permitir qualquer número, positivo e negativo, mas ela pode retornar os positivos até 10

Invariants must be preserved in the subtype

Tudo que é assumido como verdade na superclasse deve seguir sendo verdade na subclasse

Exemplo: Se na superclasse um determinado campo deve ser um número positivo, ele não pode ser zero na subclasse

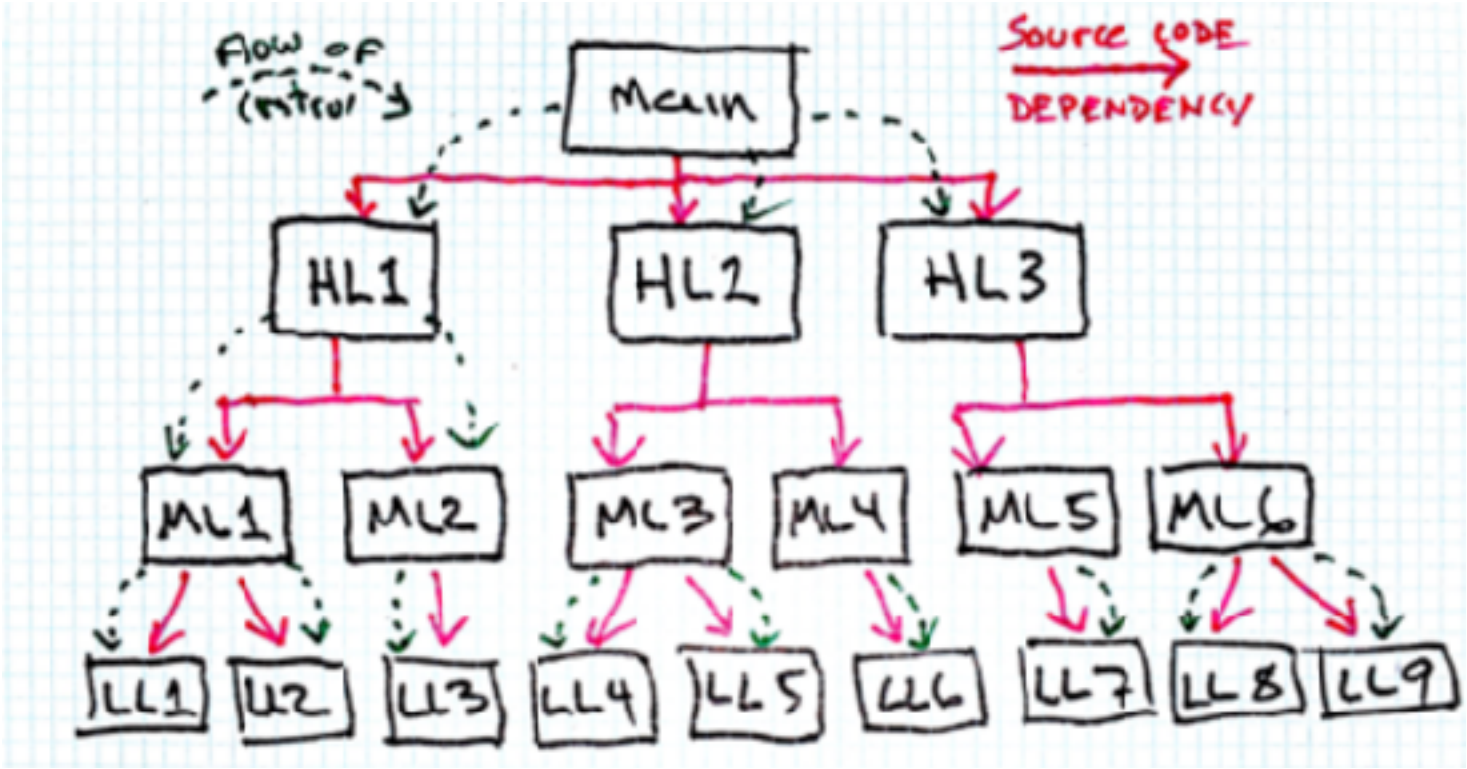
Interface Segregation

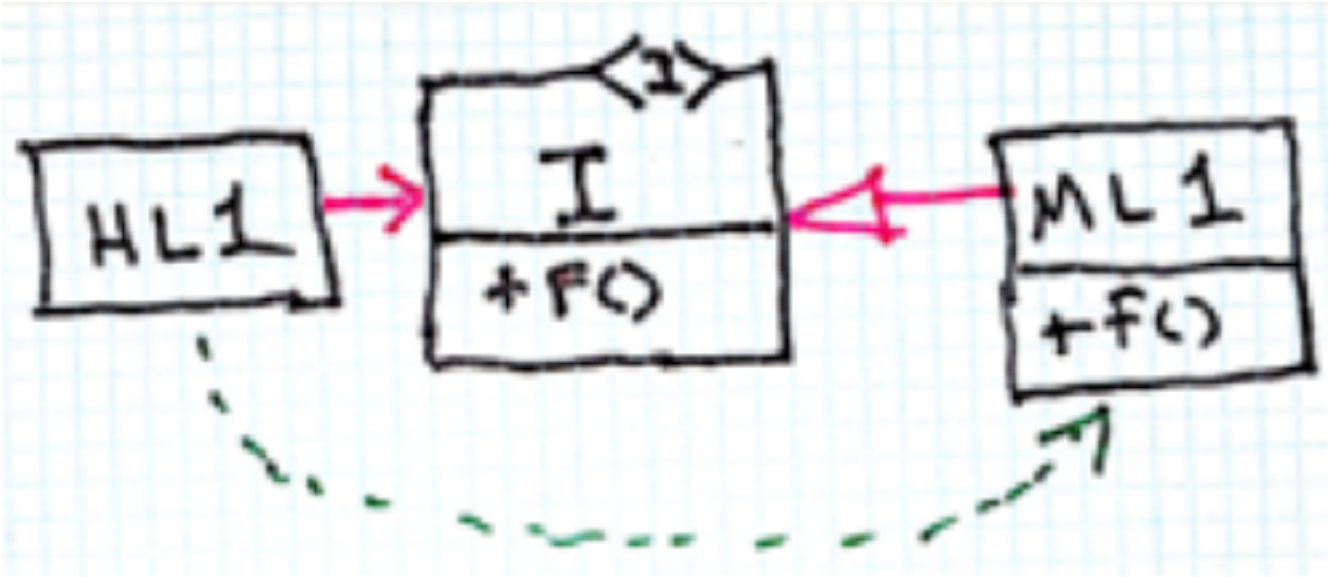
Uma subclasse não deveria implementar métodos que ela não usa, decomponha interfaces muito abrangentes em outras interfaces específicas

Este princípio lida com as desvantagens de utilizar interfaces muito grandes, que podem acabar perdendo a coesão e criando dependência desnecessária com outras classes

Dependency Inversion

É a base do desacoplamento permitindo intercambiar dependências de acordo com a necessidade, facilitando os testes e a evolução com o passar do tempo


Flow of Control
Main
Source code dependency
HL1
HL2
HL3
ML1
ML2
ML3
ML4
ML5
ML6
LL1
LL2
LL3
LL4
LL5
LL6
LL7
LL8
LL9


HL1
<I>
I
+f()
ML1
+f()

High-level modules should not depend on low-level modules, both should depend on abstractions


Porque inversão?

Frankly, it is because more traditional software development methods, such as Structured Analysis and Design, tend to create software structures in which high level modules depend upon low level modules, and in which abstractions depend upon details. Thus, the dependency structure of a well designed object oriented program is “inverted” with respect to the dependency structure that normally results from traditional procedural methods.

Robert C. Martin