

S.O.L.I.D.

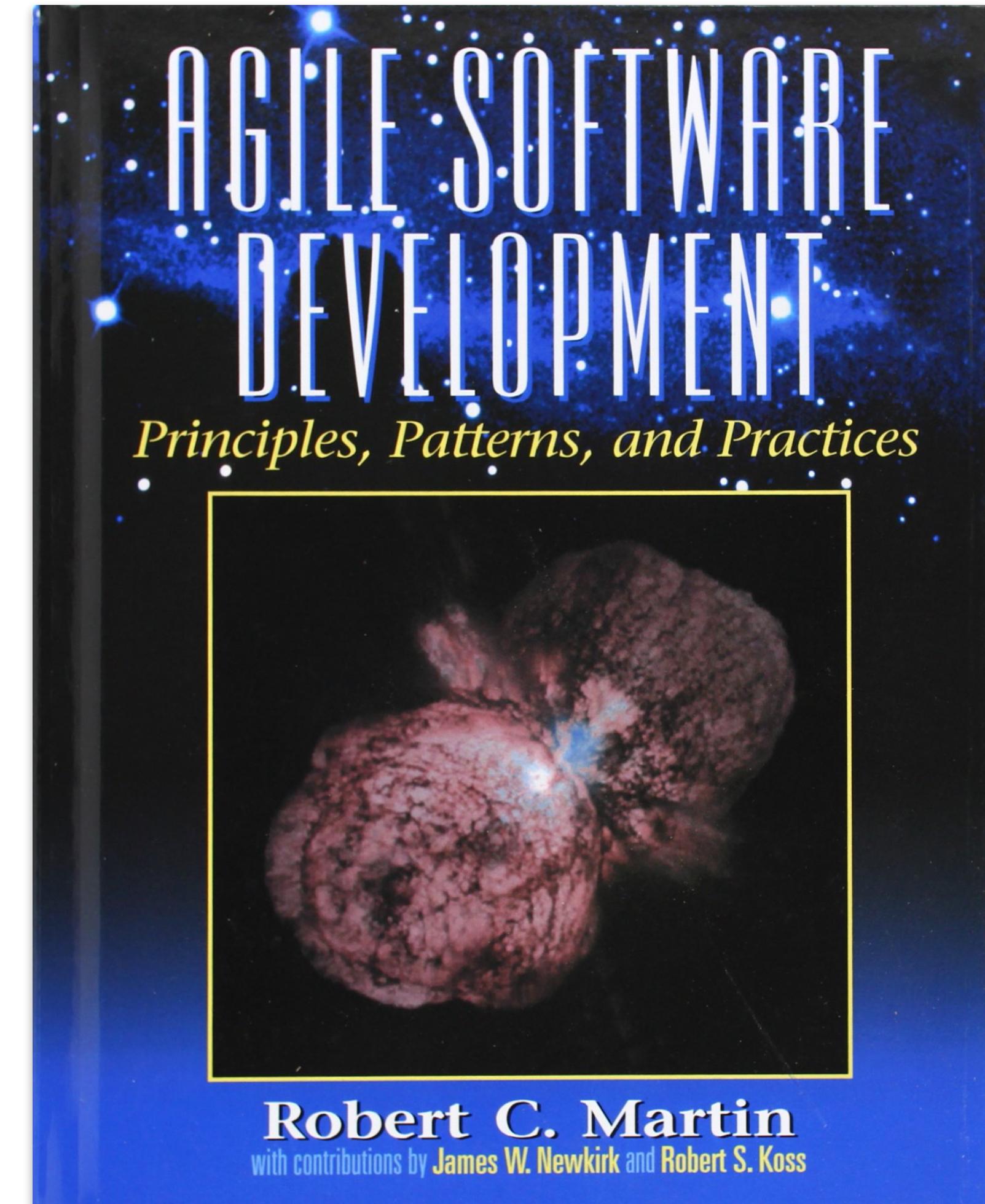
MES - Manutenção e Evolução de Software

Prof. Renato Sampaio

Robert C. Martin - princípios de OO

- S** - Single-responsibility principle
- O** - Open-closed principle
- L** - Liskov substitution principle
- I** - Interface segregation principle
- D** - Dependency Inversion Principle

Michael Feathers - nomeou o SOLID



Objetivos de O.O.

- Ser fácil de se manter, adaptar e ajustar à alterações de escopo;
- Ser testável e de fácil entendimento;
- Ser extensível para alterações com o mínimo de esforço;
- Fornecer o máximo de reaproveitamento;
- Ser utilizável pelo máximo tempo possível.

- **Evita problemas como:**
 - dificuldade na testabilidade (criação de testes unitários);
 - código sem estrutura padronizada (macarrão);
 - dificuldade de isolar funcionalidades (acoplamento alto);
 - duplicação de código (não ter que mudar algo em vários lugares distintos na hora da manutenção);
 - fragilidade de código (uma pequena mudança que gera grandes efeitos colaterais).

Single Responsibility Principle

Open-Closed Principle

Liskov Substitution Principle

Interface Segregation Principle

Development Inversion Principle

Single Responsibility Principle

Open-Closed Principle

Liskov Substitution Principle

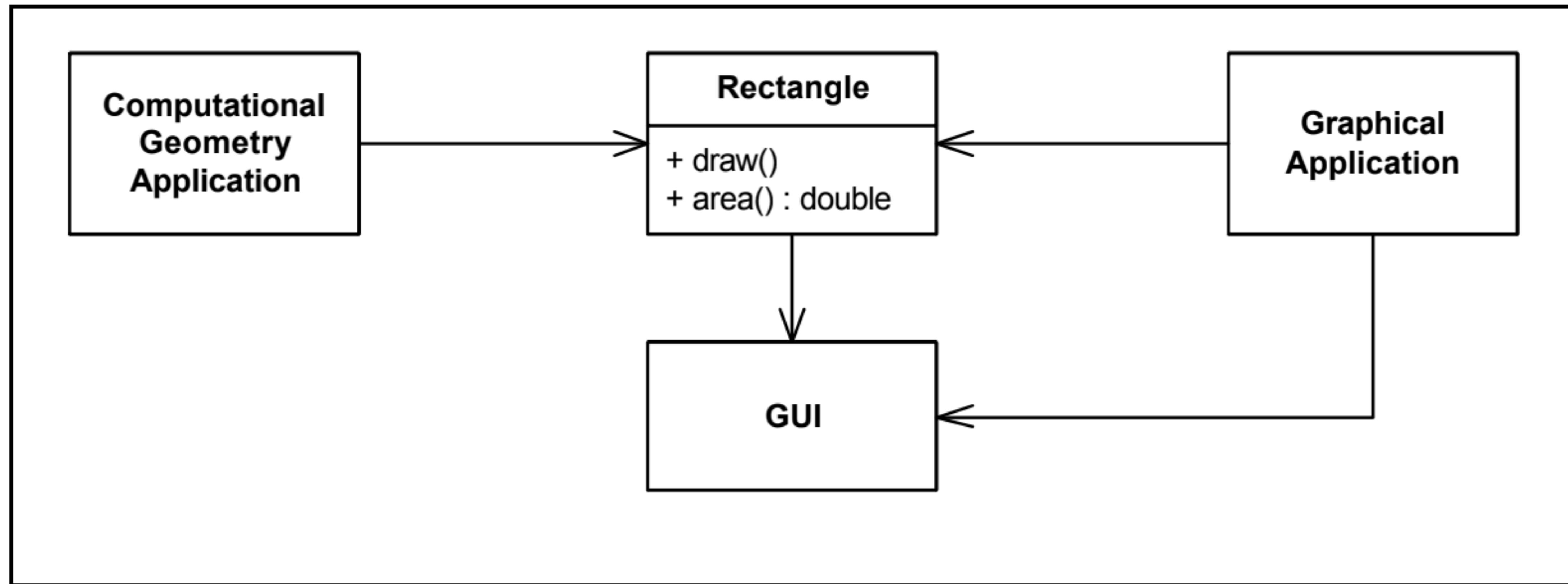
Interface Segregation Principle

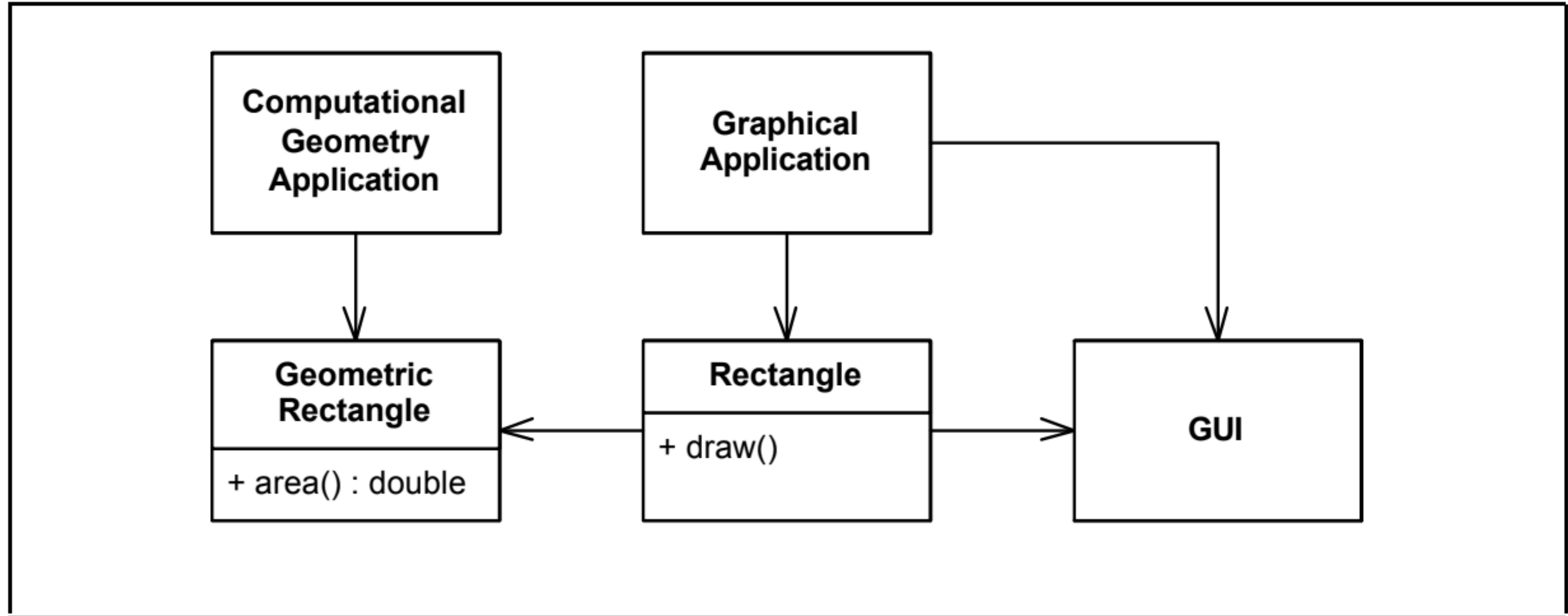
Dependency Inversion Principle

"A class should have one, and only one, reason to change."

Martin Fowler

- A classe deve ter uma única responsabilidade.
- Pode ser aplicado a classe, métodos, arquivos (um por classe, por exemplo).
- Responsabilidade não é função. Significa pontos de mudança.





Single Responsibility Principle

Open-Closed Principle

Liskov Substitution Principle

Interface Segregation Principle

Dependency Inversion Principle

OCP - Open Closed Principle

"Modules (classes, functions, etc.) should be open for extension, but closed for modification."

(Bertrand Meyer)

- Uma classe após pronta não deve ser mexida. Deve ser extensível.
- O polimorfismo é a base da extensão. Você cria novo código mas não mexe no código anterior.
- Ex: classe abstrata com herança e sobrescrita de métodos. Se a classe muda, seus testes terão que ser modificados. Se é estendida, cria-se um novo teste.

Shape.h

```
enum ShapeType {circle, square};  
struct Shape  
{enum ShapeType itsType;};
```

Circle.h

```
struct Circle  
{  
    enum ShapeType itsType;  
    double itsRadius;  
    Point itsCenter;  
};  
void DrawCircle(struct Circle*)
```

Square.h

```
struct Square  
{  
    enum ShapeType itsType;  
    double itsSide;  
    Point itsTopLeft;  
};  
void DrawSquare(struct Square*)
```

DrawAllShapes.c

```
#include <Shape.h>  
#include <Circle.h>  
#include <Square.h>  
  
typedef struct Shape* ShapePtr;  
  
void  
DrawAllShapes(ShapePtr list[], int n)  
{  
    int i;  
    for( i=0; i< n, i++ )  
    {  
        ShapePtr s = list[i];  
        switch ( s->itsType )  
        {  
            case square:  
                DrawSquare((struct Square*)s);  
                break;  
            case circle:  
                DrawCircle((struct Circle*)s);  
                break;  
        } } }
```

OCP - Open Closed Principle

- A tentativa de inserir uma nova classe, por exemplo um círculo ou uma elipse tem um grande efeito colateral.
- Para adicionar uma nova forma geométrica temos que inserí-la em Shape.h. Tudo terá que ser compilado.
- Todos as ocorrências do *switch statement* terão que ser reescritas, etc.

Shape.h

```
Class Shape
{
public:
    virtual void Draw() const=0;
};
```

Square.h

```
Class Square: public Shape
{
public:
    virtual void Draw() const;
};
```

Circle.h

```
Class Circle: public Shape
{
public:
    virtual void Draw() const;
};
```

DrawAllShapes.cpp

```
#include <Shape.h>

void
DrawAllShapes(Shape* list[], int n)
{
    for(int i=0; i< n; i++)
        list[i]->draw();
}
```

- Com a herança de classes e o polimorfismo, atendemos o OCP.
- Não temos mais um código tão rígido.
- Nada precisa ser recompilado caso seja adicionada uma nova classe com uma nova forma geométrica.

Single Responsibility Principle

Open-Closed Principle

Liskov Substitution Principle

Interface Segregation Principle

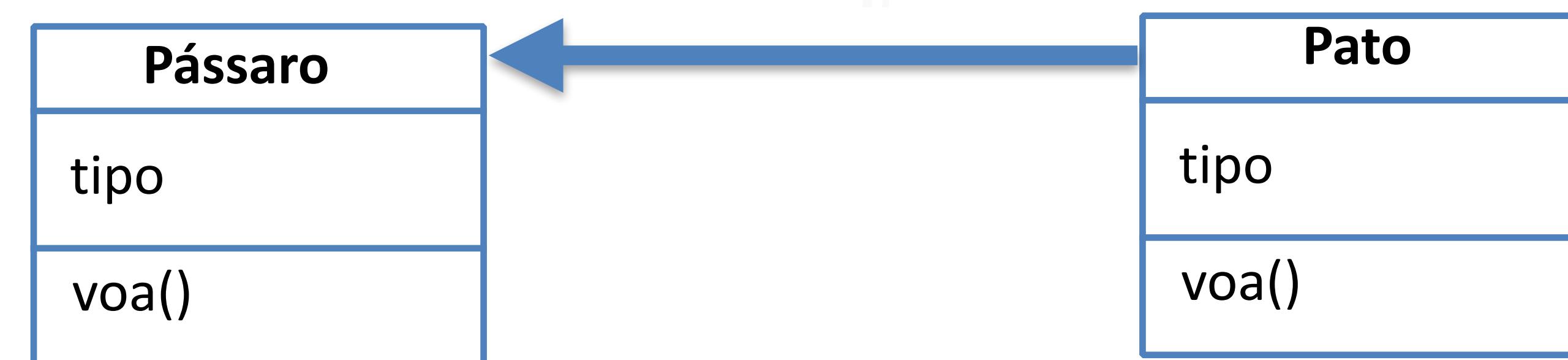
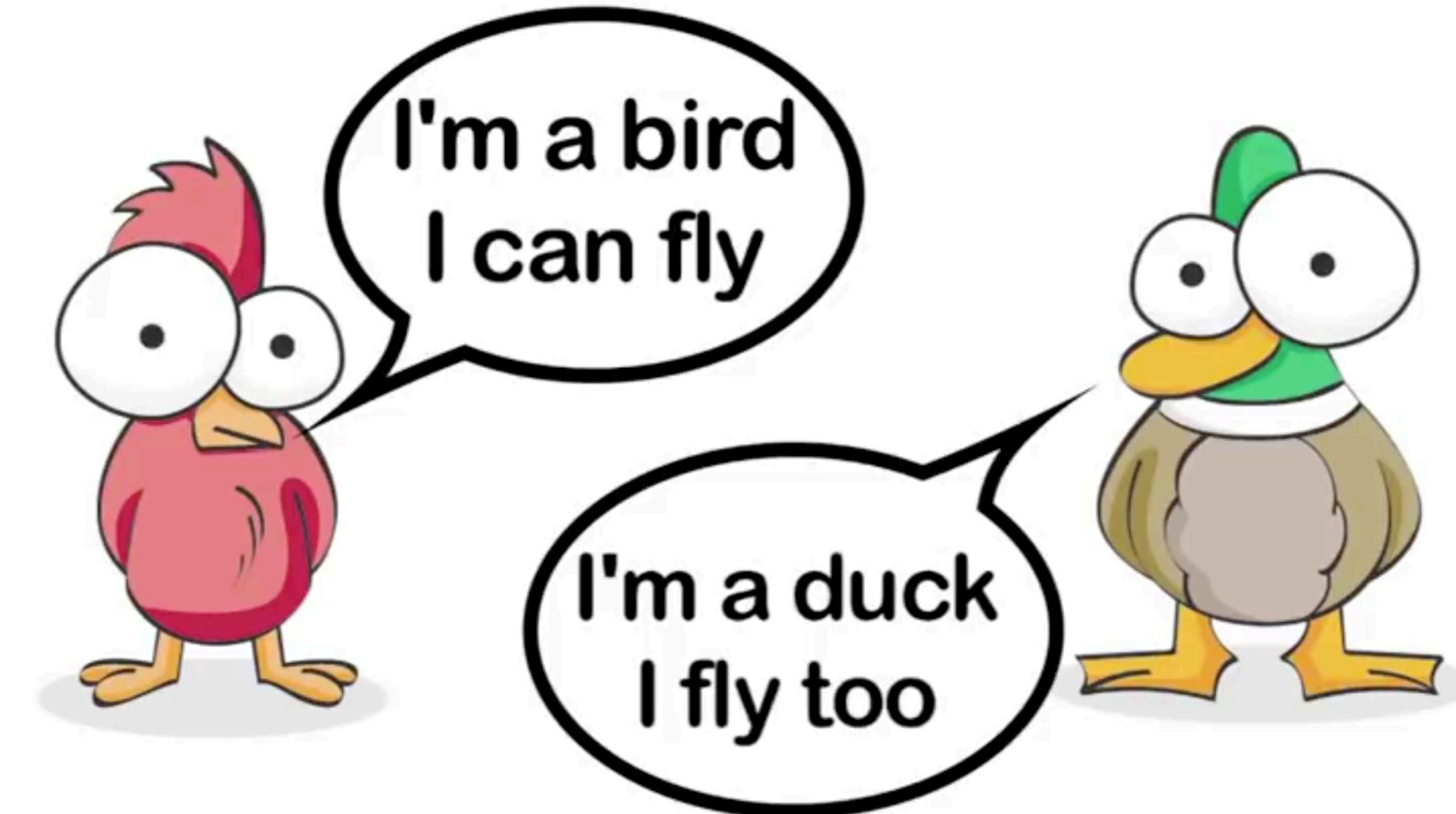
Dependency Inversion Principle

LSP - Liskov Substitution Principle

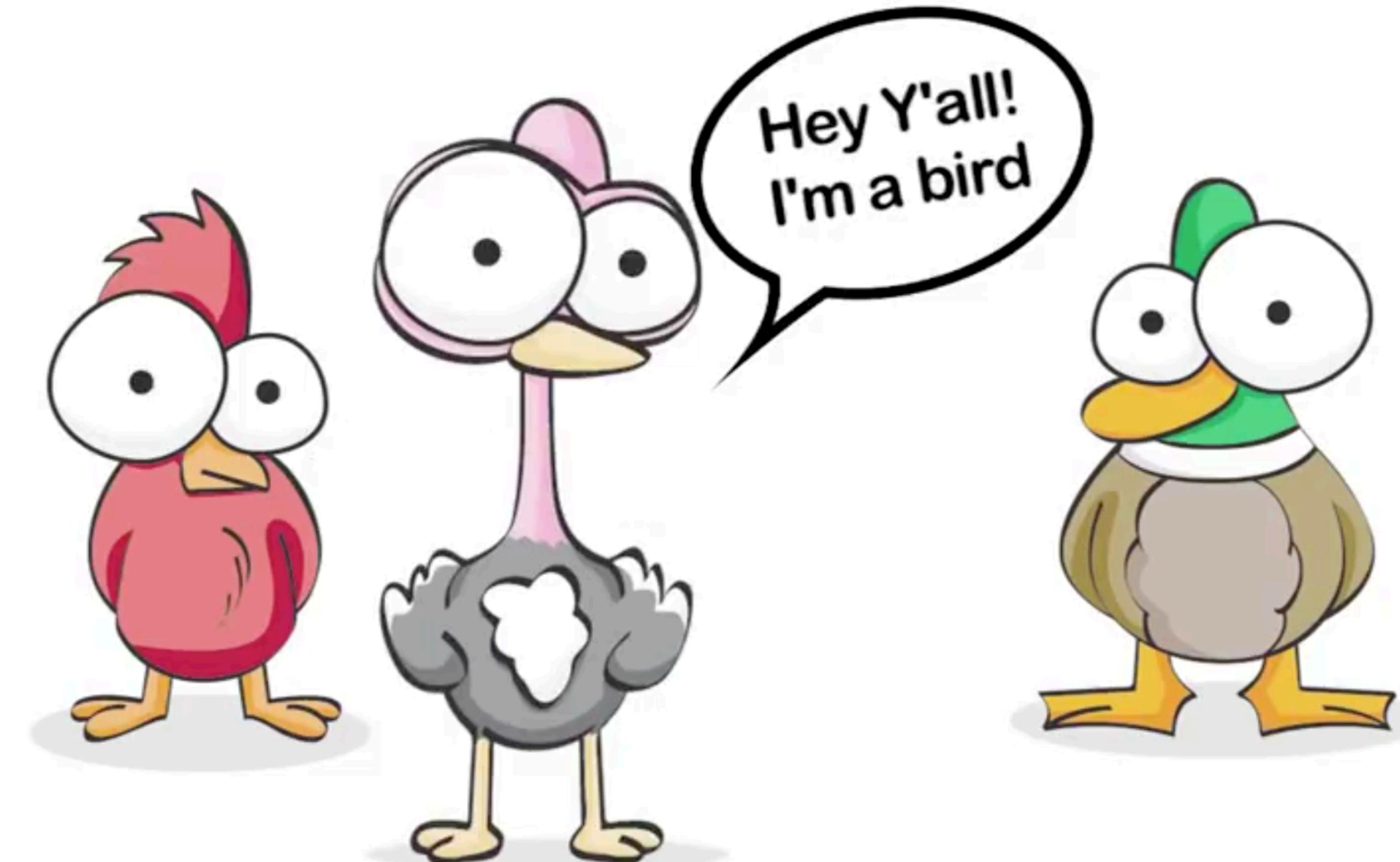
“Let $q(x)$ be a property provable about objects x of type T . Then $q(y)$ should be provable for objects y of type S , where S is a subtype of T .” Barbara Liskov - 1988 (MIT)

- Uma classe base deve poder ser substituída por sua classe derivada.
- Classes derivadas devem ser utilizáveis através da interface da classe base, sem a necessidade do usuário saber a diferença.

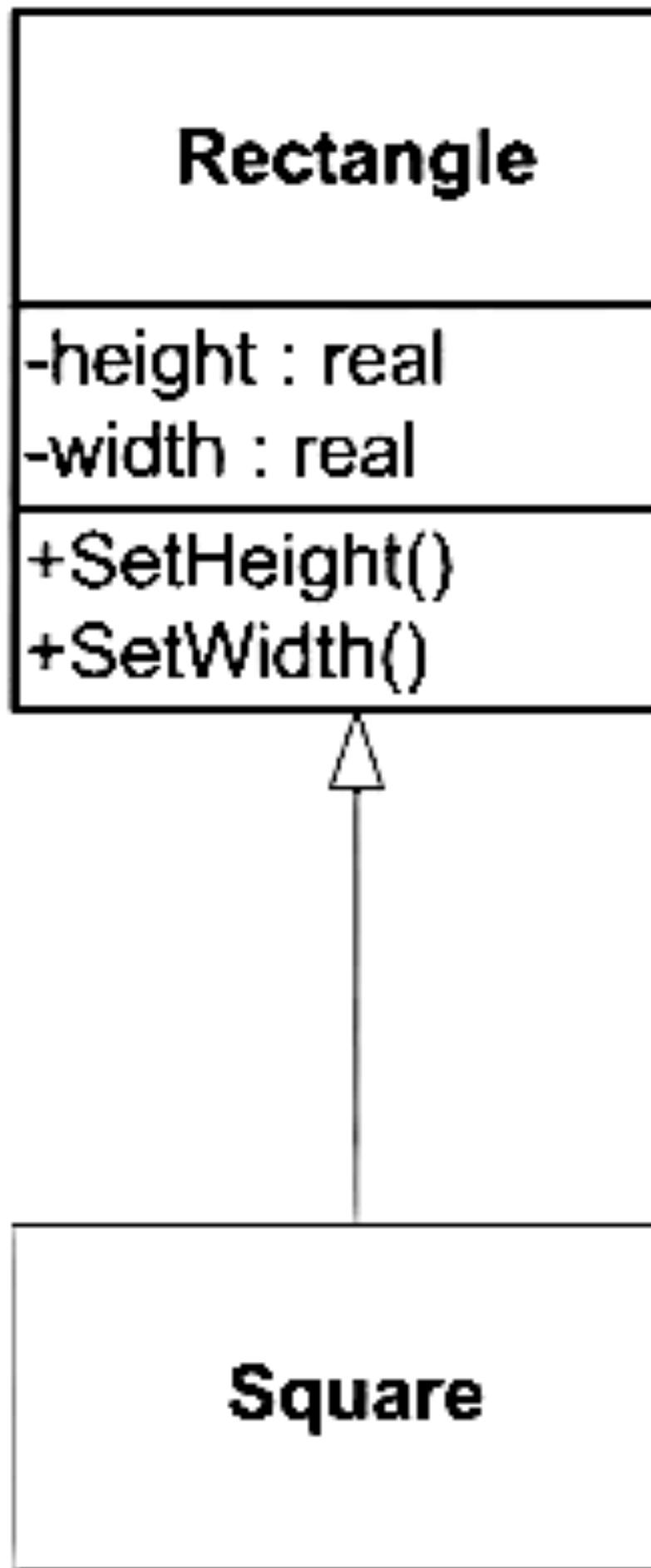
LSP - Liskov Substitution Principle



LSP - Liskov Substitution Principle



LSP - Liskov Substitution Principle



```
void Square::SetWidth(double w)
{
    width = w;
    height = w;
}

void Square::SetHeight(double h)
{
    width = h;
    height = h;
}
```

- Exemplo de solução: perguntar o tipo do objeto antes de usá-lo. (IF) Instanceof(), is(), as(), etc.
- Já quebrou o OCP. Criou uma dependência.
- Relação É UM (is a) nem sempre funciona.
- Qual seria a relação entre retângulo e quadrado? Podem ambos derivar de uma classe base. Porém, elas não tem relação direta. É uma violação do LSP.
- Se você viola o LSP eventualmente você irá quebrar a regra e usar um IF em algum lugar!

- Respeitar o LSP implica em respeitar o OCP

LSP => OCP

Single Responsibility Principle

Open-Closed Principle

Liskov Substitution Principle

Interface Segregation Principle

Dependency Inversion Principle

“States that no client should be forced to depend on methods it does not use.”

- Clientes (classes) não devem ser forçados a depender de métodos que não usam.
- Muitas interfaces simples é melhor do que uma única interface genérica.

```
interface StreamIO {  
    void reset();  
    void read( ... );  
    void write( ... );  
}
```

Como usar esta interface para um sensor? (Read-only)

Ou para uma impressora? (Write-only)

```
interface ReadableStream {  
void reset();  
void read( ... );  
}
```

```
interface WritableStream {  
void write( ... );  
}
```

Single Responsibility Principle

Open-Closed Principle

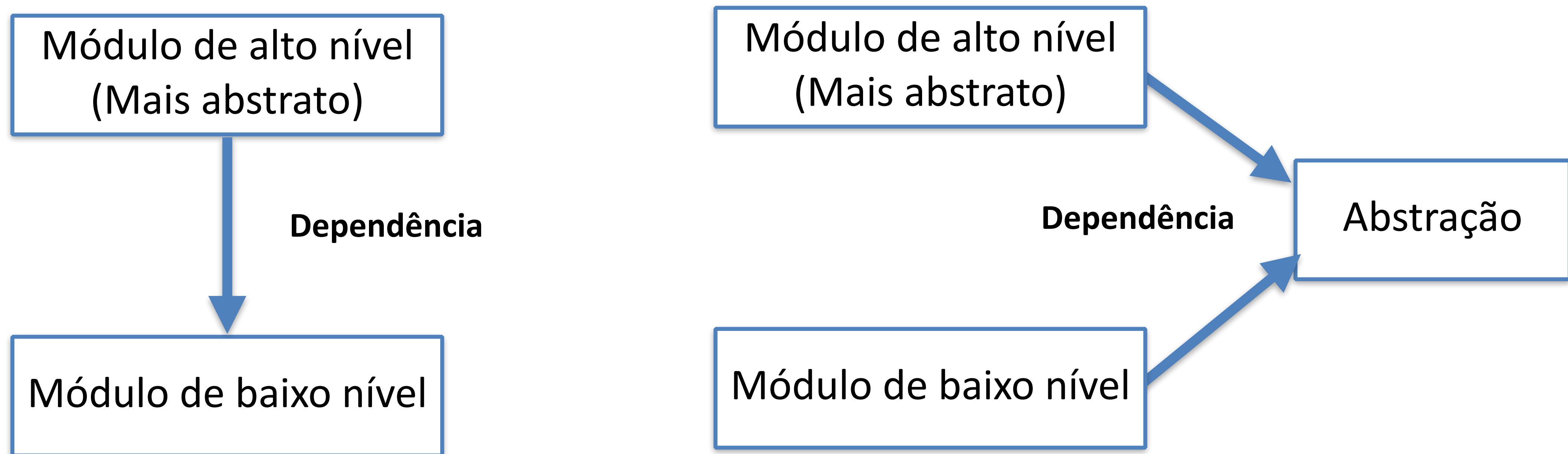
Liskov Substitution Principle

Interface Segregation Principle

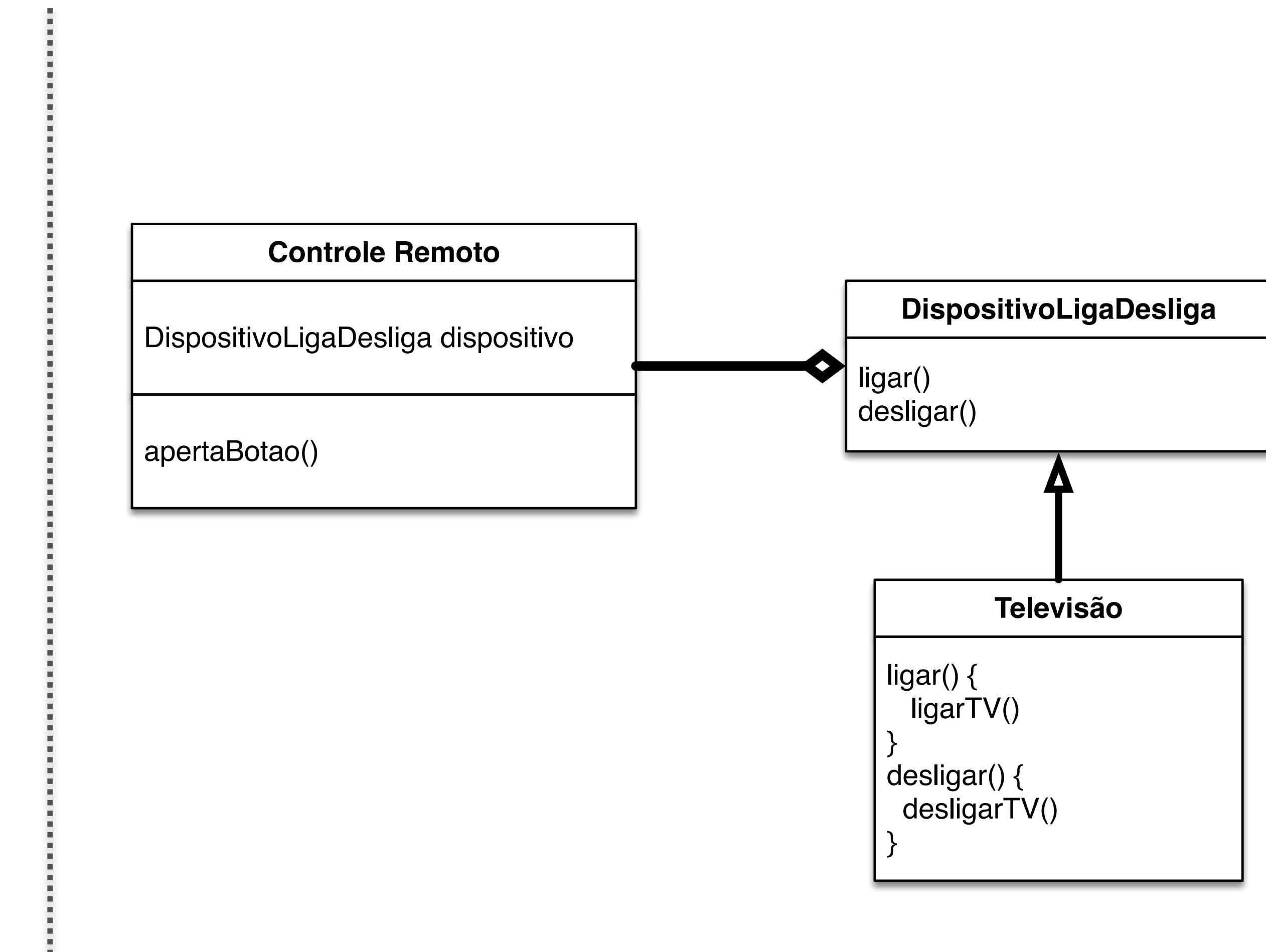
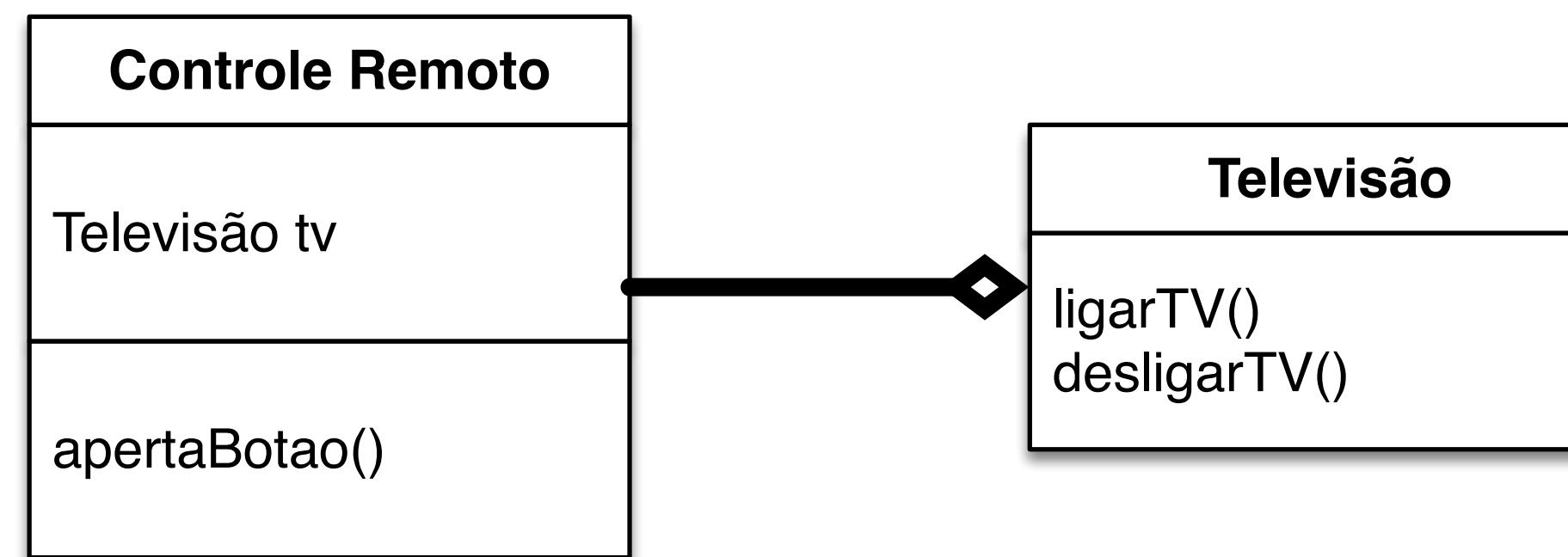
Dependency Inversion Principle

“High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions”

- 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.
- Dependa de uma abstração e não de uma implementação.



- Ex: a classe de cadastro de cliente e a classe de cadastro de banco de dados. A classe deveria depender da abstração da classe de banco de dados e não da implementação.
- Se eu implemento uma classe diretamente no código de outra eu estou criando um **acoplamento**.
- Qualquer mudança na classe, vai impactar as classes que a utilizam.



Outros Princípios Comuns

- Encapsular o que varia muito das áreas de código mais permanentes.
- Dar preferência a **Composição** ao invés de **Herança**
- Reduzir acoplamentos (*Loose Coupling*)
- Abstração através de **interfaces**.

Referências

- **Principles of OOD - Robert C. Martin**
<http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod>
- **Alan Barber** - <https://alanbarber.com/2015/08/06/solid-principles-five-principles-of-objectoriented-programming-and-design/>
- **Eduardo Pires** - <http://eduardopires.net.br/2015/01/solid-teoria-e-pratica/>
- Samuel Oloruntoba, **S.O.L.I.D: The First 5 Principles of Object Oriented Design**: <https://scotch.io/bar-talk/s-o-l-i-d-the-first-five-principles-of-object-oriented-design>
- [http://www.cvc.uab.es/shared/teach/a21291/temes/object oriented design/materials adicionais/principles and patterns.pdf](http://www.cvc.uab.es/shared/teach/a21291/temes/object_oriented_design/materials_adicionais/principles_and_patterns.pdf)