Uma breve introdução ao S.O.L.I.D.

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**.

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;**



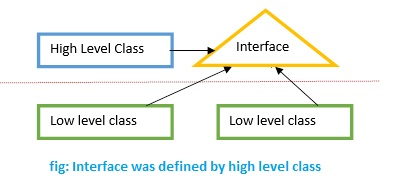
|  |  |  |
| --- | --- | --- |
| **SRP** | The Single Responsibility Principle Principio da Responsabilidade Única | Uma classe deve ter um, e somente um, motivo para mudar. A class should have one, and only one, reason to change. |
| **OCP** | The Open Closed Principle Princípio Aberto-Fechado | Você 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. |

*“Módulos de alto nível não devem   
depender de módulos de baixo nível.  
Ambos devem depender de abstrações.”*

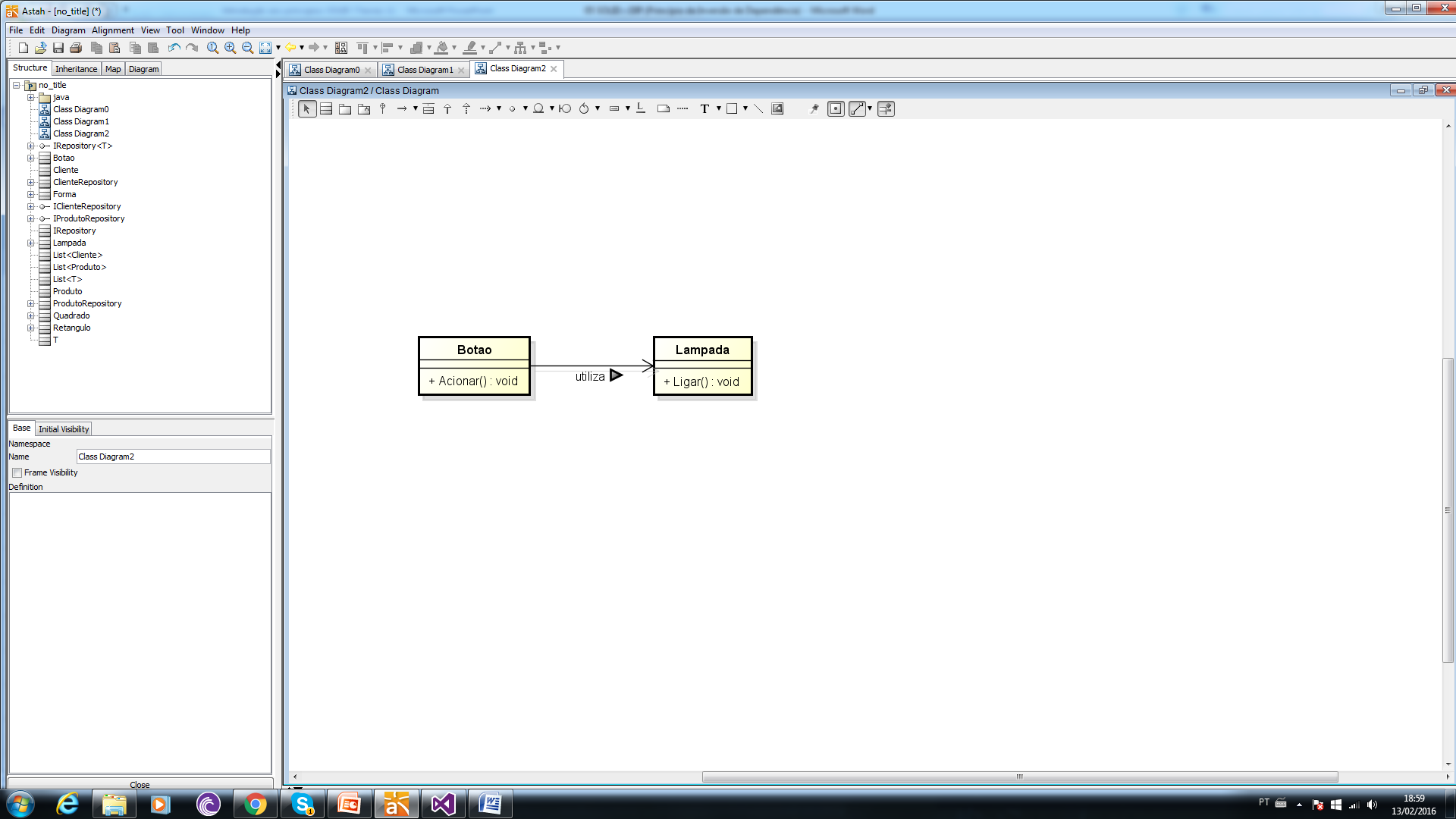
Inverter a dependência faz com que um cliente não fique frágil a mudanças relacionadas a detalhes de implementação. Isto é, alterar o detalhe não quebra o cliente. Além disso, o mesmo cliente pode ser reutilizado com outro detalhe de implementação.

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.

Este princípio reforça que a abstração está mais relacionada ao seu cliente do que ao servidor (a classe que realiza a abstração). No exemplo que veremos abaixo, IDispositivo (a abstração) está diretamente ligado ao cliente (Botao). Sua implementação (Lampada) é um mero detalhe. Sendo assim, Dispositivo ficaria no mesmo pacote (ou componente) do Botao e não junto com sua implementação Lampada. (Esta separação de interface e implementação em componentes distintos é um padrão conhecido por[**Separated Interface**](http://martinfowler.com/eaaCatalog/separatedInterface.html), como catalogado no livro PoEAA, do Martin Fowler.)



**Exemplo 01: Ferindo o padrão de Inversão de dependência.**



using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace DIP.ModoErrado

{

public class Lampada

{

public void Ligar()

{

Console.WriteLine("Lampada Acesa");

}

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace DIP.ModoErrado

{

public class Botao

{

private Lampada lampada;

public Botao(Lampada lampada)

{

this.lampada = lampada;

}

public void Acionar()

{

lampada.Ligar();

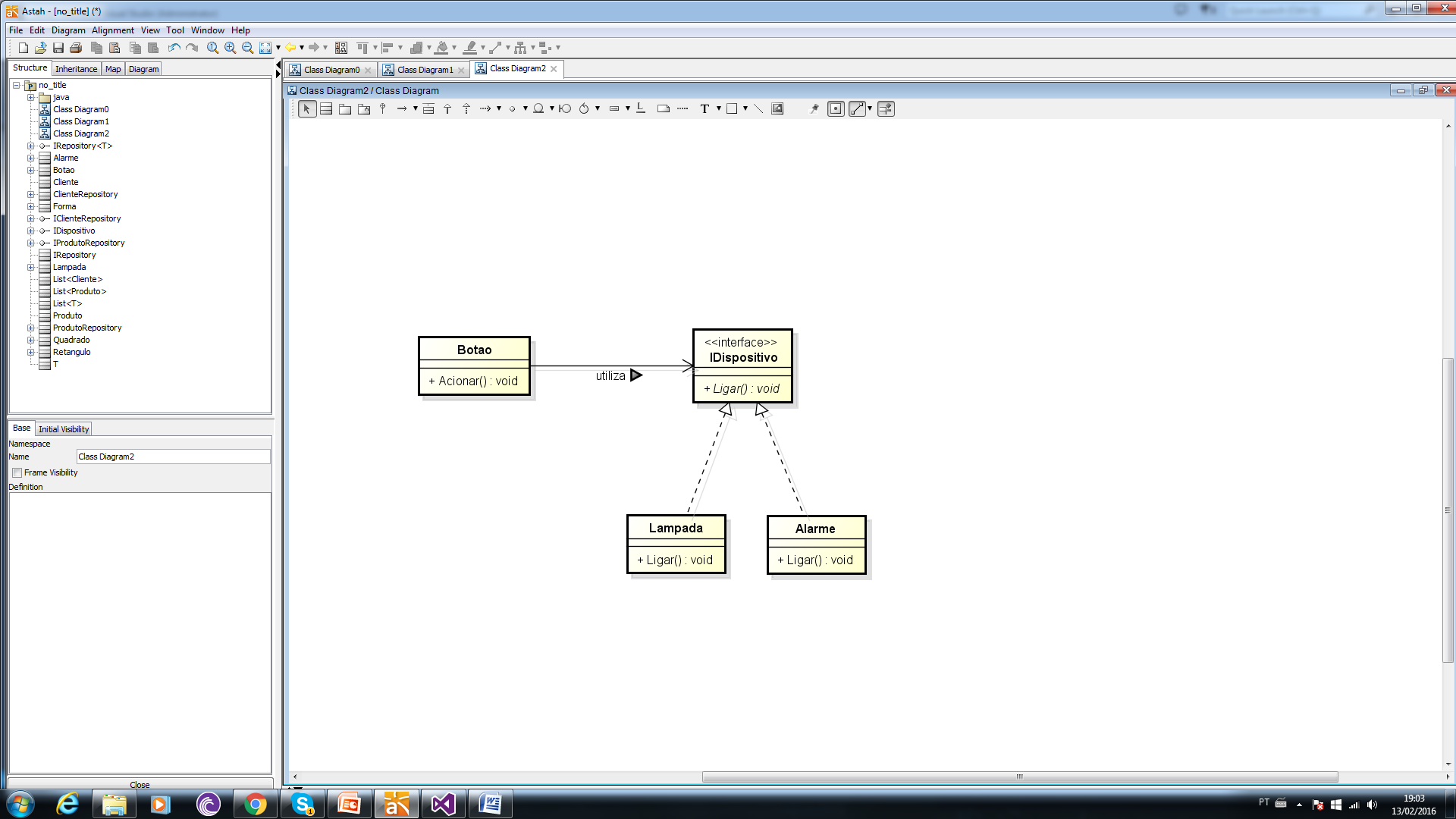
}

}

}

O design acima viola o DIP uma vez que Botao depende de uma classe concreta Lampada. Ou seja, Botao conhece detalhes de implementação ao invés de termos identificado uma abstração para o design. Que abstração seria essa? Botao deve ser capaz de tratar alguma ação e ligar ou desligar algum **dispositivo**, seja ele qual for: uma lâmpada, um motor, um alarme, etc.

**Exemplo 02: Aplicando Inversão de dependência.**



Codificando:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace DIP.ModoCorreto

{

public interface IDispositivo

{

void Ligar();

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace DIP.ModoCorreto

{

public class Lampada : IDispositivo

{

public void Ligar()

{

Console.WriteLine("Lampada Acesa");

}

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace DIP.ModoCorreto

{

public class Alarme : IDispositivo

{

public void Ligar()

{

Console.WriteLine("Alarme Ligado");

}

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace DIP.ModoCorreto

{

public class Botao

{

private IDispositivo dispositivo;

public Botao(IDispositivo dispositivo)

{

this.dispositivo = dispositivo;

}

public void Acionar()

{

dispositivo.Ligar();

}

}

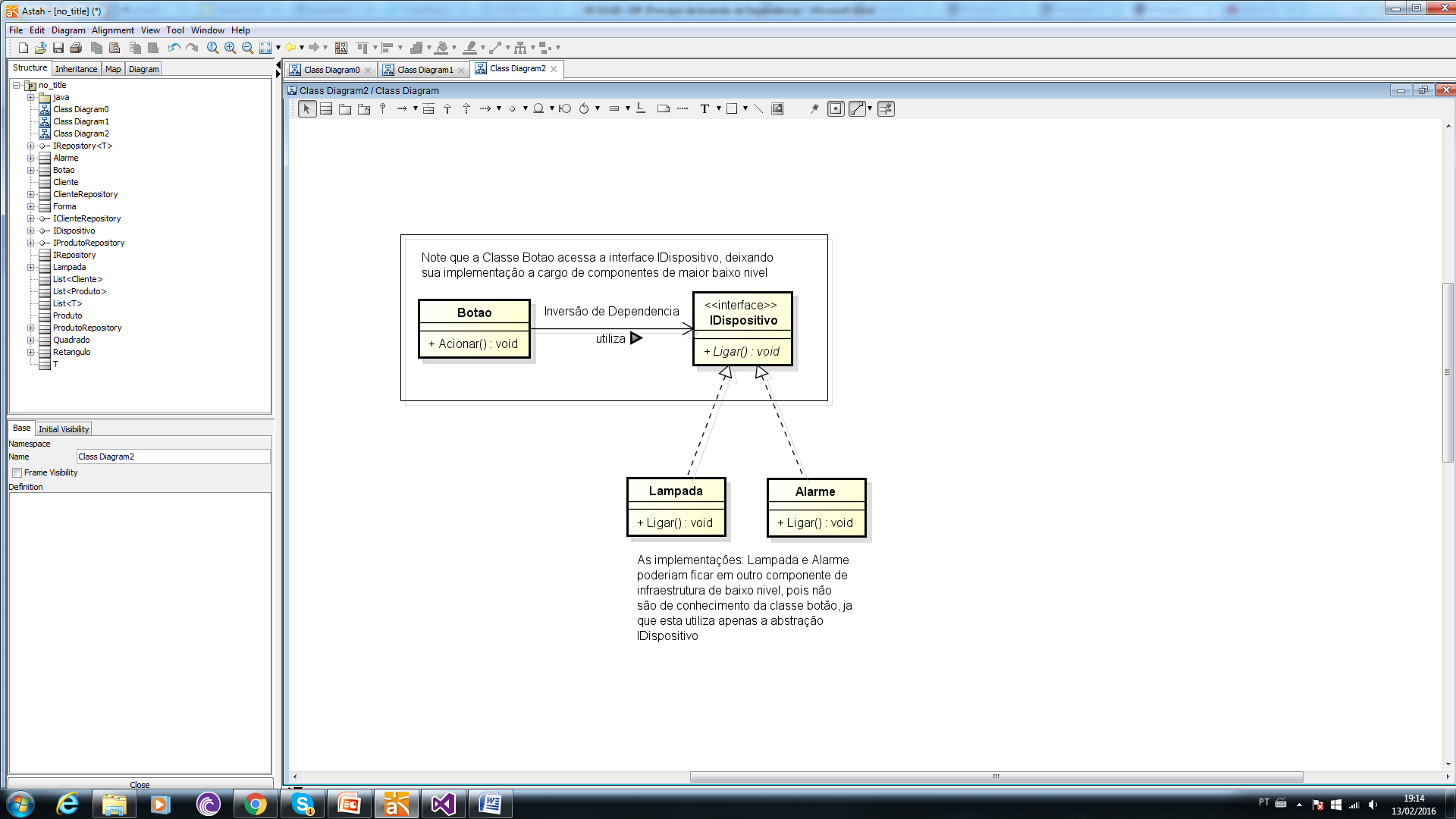
}

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.

Este princípio reforça que a abstração está mais relacionada ao seu cliente do que ao servidor (a classe que realiza a abstração).

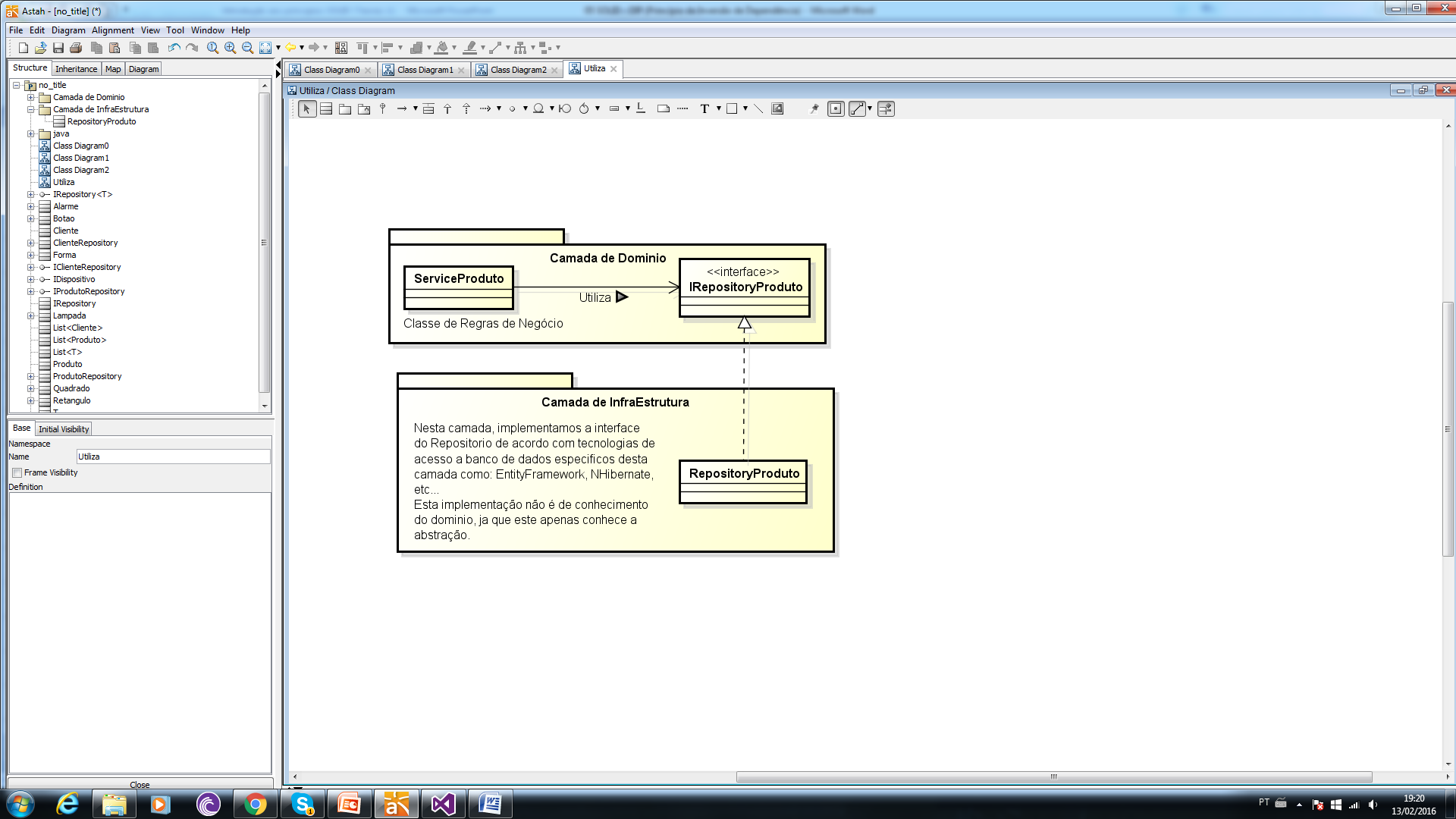
No exemplo acima, Dispositivo (a abstração) está diretamente ligado ao cliente (Botao).   
Sua implementação (Lampada) é um mero detalhe.

Sendo assim, IDispositivo ficaria no mesmo pacote (ou componente) do Botao e não junto com sua implementação Lampada ou Alarme.



Levando a interface para junto do cliente, estamos dizendo “o cliente funciona dessa maneira” e quem implementa a interface (em outro componente) é que deve atender a essa exigência. Ou seja, a interface só mudará por necessidade DO CLIENTE.

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



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.

**Fontes de Estudo:**

* <http://blog.thedigitalgroup.com/rakeshg/2015/05/06/solid-architecture-principle-using-c-with-simple-c-example/>
* <https://robsoncastilho.com.br/2013/05/01/principios-solid-principio-da-inversao-de-dependencia-dip/>