CES 22 aula 13

Princípios para OO Design

Objetivos

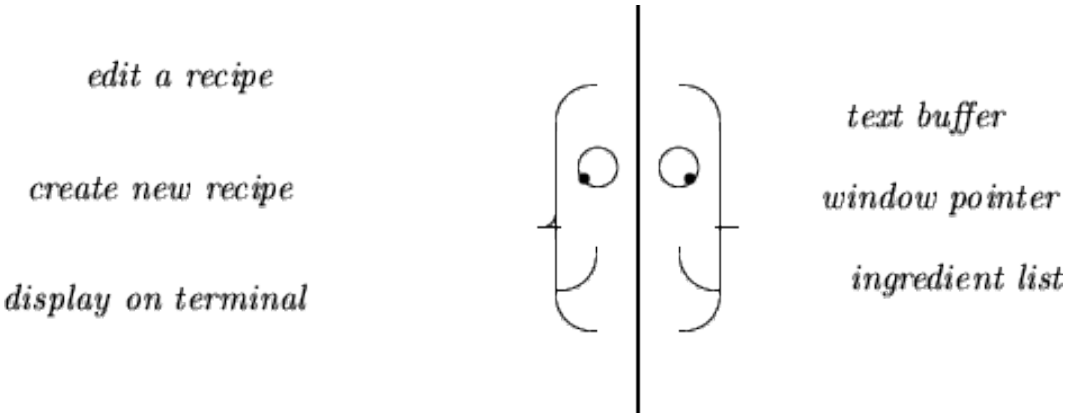
- Princípios de Parnas (1972)
- Príncipios para OO design
- Princípios SOLID

Introdução

- A adoção de princípios de Design é essencial para a construção de programas fáceis de serem modificados e expandidos.

Principios de David Parnas

- Duas visões de um sistema de software:



• Information Hiding busca esconder detalhes de implementação dos usuários.
• O usuário deve saber o que o módulo faz e não saber o como.

•Objetivo: Os usuários não são afetados quando houver modificações na implementação.

Os princípios

- O usuário de um módulo de software deve receber todas as informações necessárias para o uso efetivo dos serviços providos pelo módulo e nenhuma outra informação adicional.
- O implementador de um módulo de software deve receber todas as informações necessárias para executar as responsabilidades atribuídas ao módulo e nenhuma outra informação adicional.

Abstração e Encapsulamento

- Os princípios de Parnas influenciaram o desenvolvimento de linguagens com suporte a modularização e as linguagens Orientada a Objetos.
- **Abstração** através do uso de interfaces. Os usuários interagem apenas com as interfaces. Os detalhes ficam escondidos dentro dos módulos.
- Com **encapsulamento** os comportamentos internos de objetos não são acessados diretamente pelos clientes dos objetos.

Princípios para OO Design

- Na literatura são encontrados vários exemplos de boas práticas a serem adotadas pelos desenvolvedores.

DRY (Don´t repeat yourself)

- Não repita código. Use abstração para coisas comuns.
- Cuidado:
  - Não abusar. O foco é evitar a duplicação de funcionalidades e não do código.

Encapsular o que muda

- Uma única constante em software é a mudança.
- Assim encapsular todo o código que poderá ser modificado no futuro.

Programar para a Interface e não para a Implementação

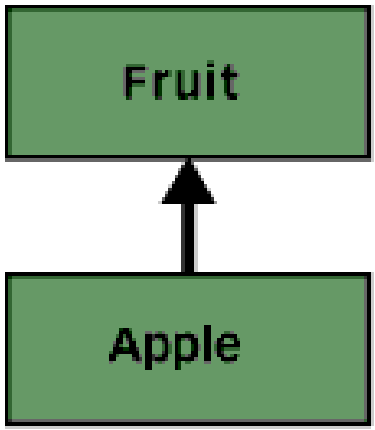
- Sempre programe considerando apenas a especificação da interface. O código funcionará para qualquer implementação da interface.

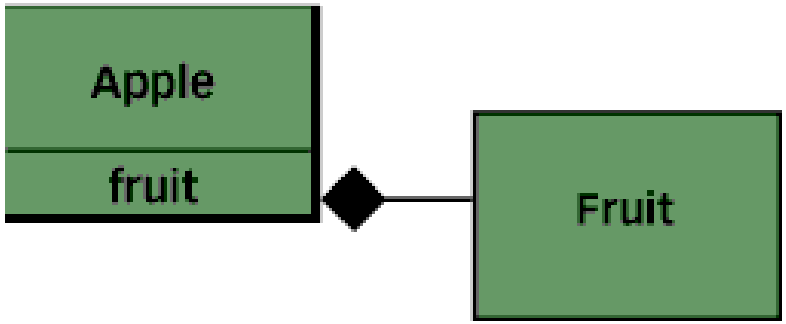
Princípio da Delegação

- Não implemente responsabilidades que devem ser delegadas para as super classes.
- Benefício: evita duplicação de código e facilita futuras modificações.
- Exemplo: métodos da classe objeto.

Preferir a composição ao invés da herança

- Sempre que possível prefira a composição ao invés da herança.
- Composição é mais flexível pois permite a troca de comportamento em tempo de execução através de modificação de propriedades e o uso de polimorfismo através do uso de interfaces.
- A herança é mais frágil pois pequenas mudanças na superclasse podem acarretar modificações em diferentes partes do programa.


Fruit
Apple


Apple
fruit
Fruit

```
class Fruit:
´´´ fruta ´´´
    def peel(self):
´´´ retorna o número de cascas ´´´
        print("Peeling is appealing")
        return 1

class Apple(Fruit):
    pass

apple=Apple()
pieces=apple.peel()
print(pieces)
```

```
class Peel:
''' casca '''
    def __init__(self, peelCount):
        self.peelCount=peelCount

    def getPeelCount(self):
        return self.peelCount

class Fruit:
''' fruta '''
    def peel(self):
        print("Peeling is appealing")
        return Peel(1)

class Apple(Fruit):
    pass

apple=Apple()
pieces=apple.peel()
print(pieces)  # imprime o objeto
pieces !!!
```

```
class Fruit:

    def peel(self):
        print("Peeling is appealing")
        return 1

class Apple:
    def __init__(self):
        self.fruit=Fruit()  # composicao

    def peel(self):
        return self.fruit.peel()

apple=Apple()
pieces=apple.peel()
print(pieces)
```

```
class Peel:
    def __init__(self, peelCount):
        self.peelCount=peelCount
    def getPeelCount(self):
        return self.peelCount



class Fruit:

    def peel(self):
        print("Peeling is appealing")
        return Peel(1)

class Apple:
    def __init__(self):
        self.fruit=Fruit()

    def peel(self):   # classe Apple é modificada
        self.peel=self.fruit.peel()
        return self.peel.getPeelCount()

apple=Apple()  # programa cliente não requer
modificacao
pieces=apple.peel()
print(pieces)
```

SOLID principles

- **S**ingle Responsability
- **O**pen Closed Design
- **L**iskov Substitution
- **I**nterface Segregation
- **D**ependency Injection (ou Inversion)

Single Responsability

- Classes devem assumir uma única responsabilidade ou “razão para modificação”.

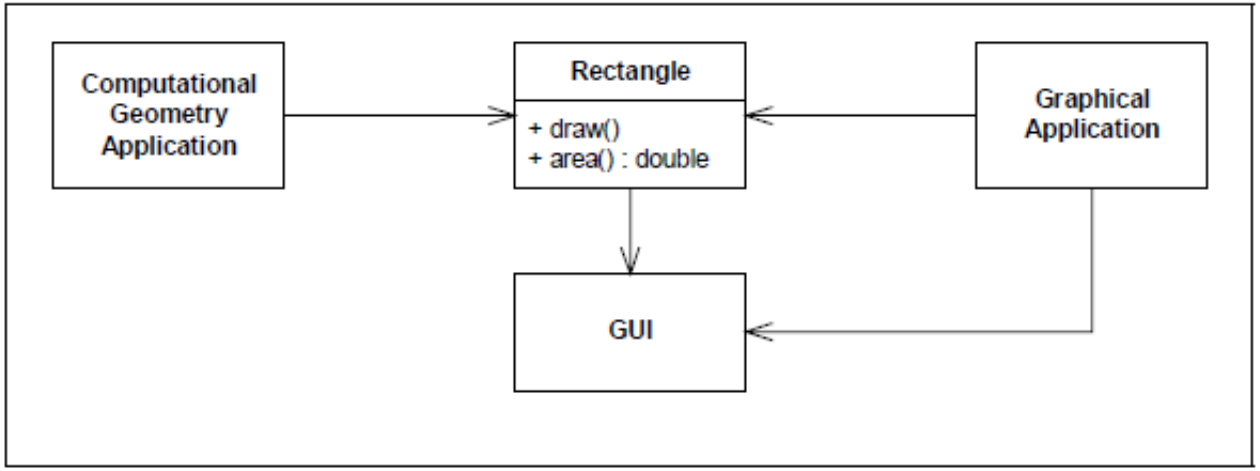


Figure 9-1
More than one responsibility

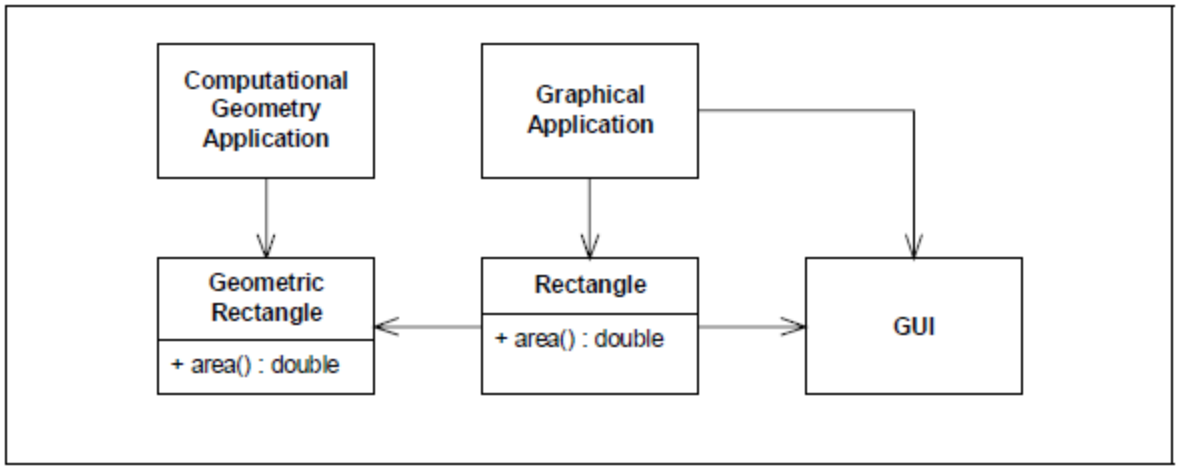


Figure 9-2
Separated Responsibilities

Listing 9-1

Modem.java -- SRP Violation

```
interface Modem
{
  public void dial(String pno);
  public void hangup();
  public void send(char c);
  public char recv();
}
```

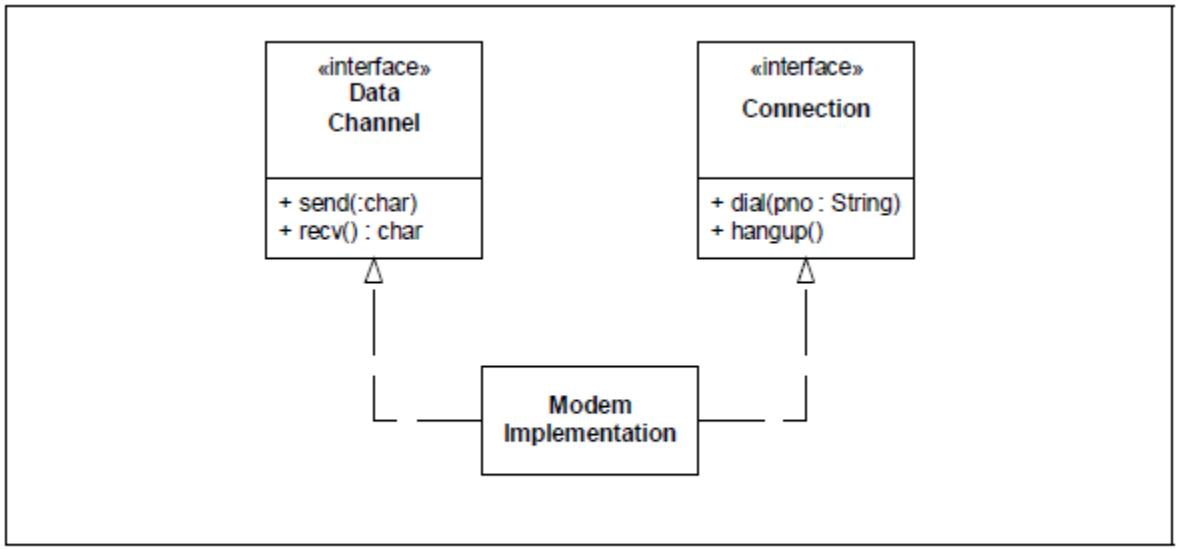


Figure 9-3
Separated Modem Interface

Open Closed Design

- Classes devem ser abertas para uso e fechadas para modificações.

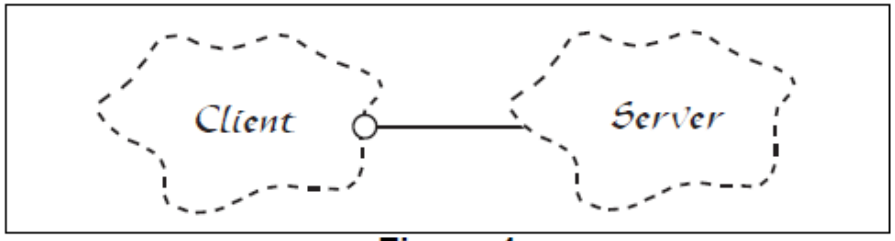


Figure 1
Closed Client

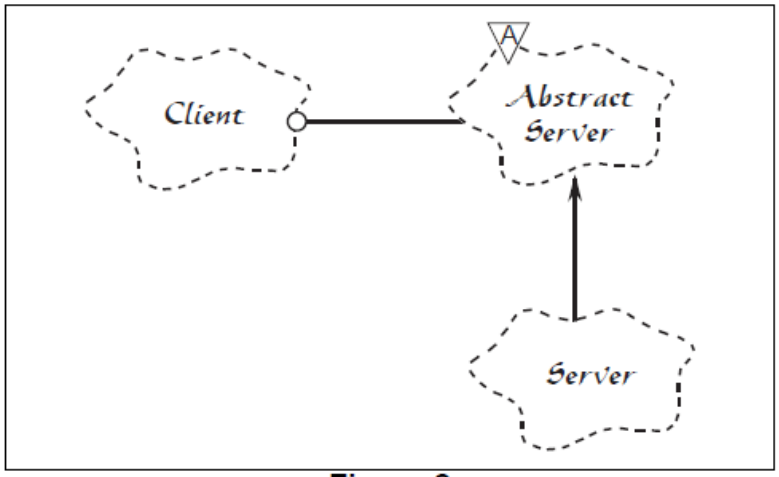
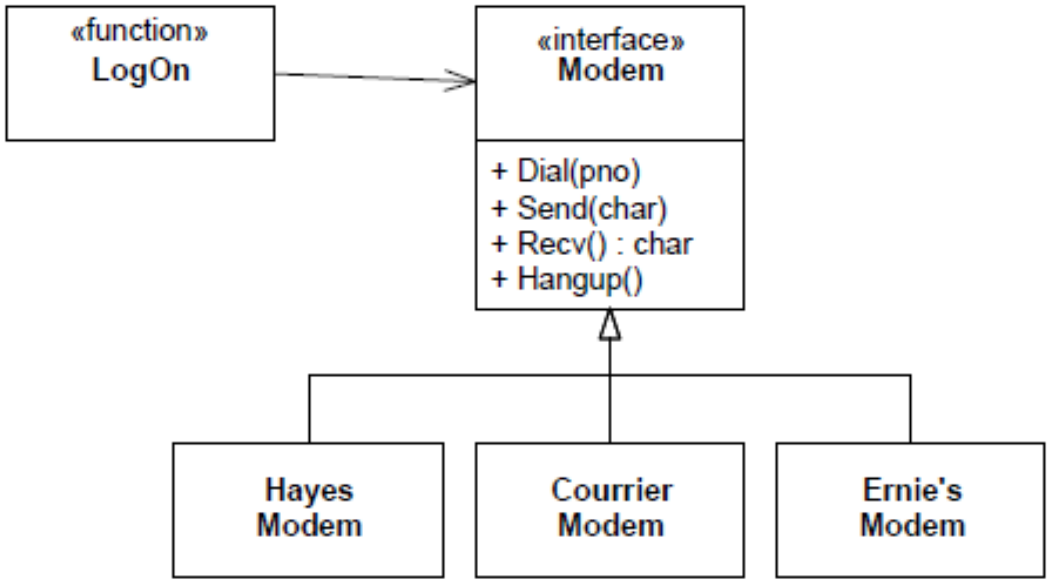


Figure 2
Open Client


«function»
LogOn
«interface»
Modem
+ Dial(pno)
+ Send(char)
+ Recv() : char
+ Hangup()
Hayes
Modem
Courrier
Modem
Ernie's
Modem

Liskov Substitution

- Subclasses devem aumentar e nunca reduzir as funcionalidades das superclasses.

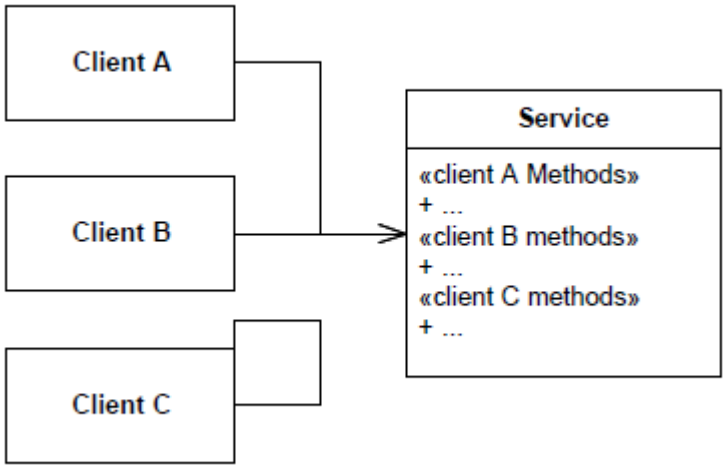
```
// Violation of Likov's Substitution Principle
class Rectangle
{
        protected int m_width;
        protected int m_height;

        public void setWidth(int width){
                m_width = width;
        }

        public void setHeight(int height){
                m_height = height;
        }


        public int getWidth(){
                return m_width;
        }

        public int getHeight(){
                return m_height;
        }

        public int getArea(){
                return m_width * m_height;
        }
}
```

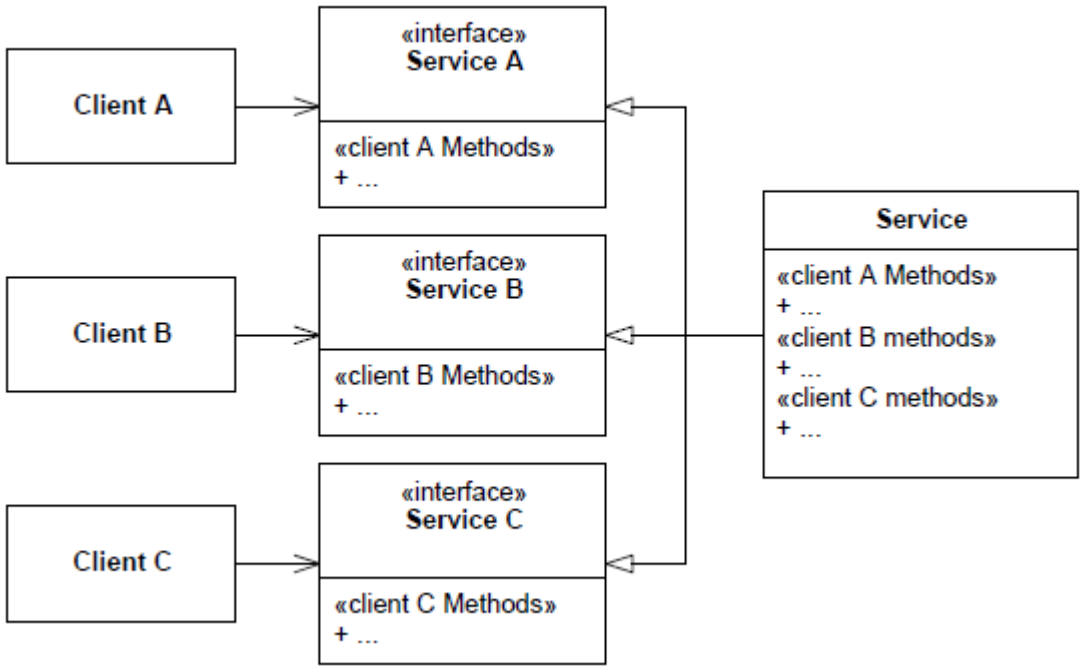
```
class Square extends Rectangle
{
        public void setWidth(int width){
                m_width = width;
                m_height = width;
        }

        public void setHeight(int height){
                m_width = height;
                m_height = height;
        }

}
```

```
class LspTest
{
        private static Rectangle getNewRectangle()
        {
                // it can be an object returned by some factory ...
                return new Square();
        }

        public static void main (String args[])
        {
                Rectangle r = LspTest.getNewRectangle();

                r.setWidth(5);
                r.setHeight(10);
                // user knows that r it's a rectangle.
                // It assumes that he's able to set the width and height as for the base class



               System.out.println(r.getArea());
               // now he's surprised to see that the area is 100 instead of 50.
        }
}
```

Interface Segregation

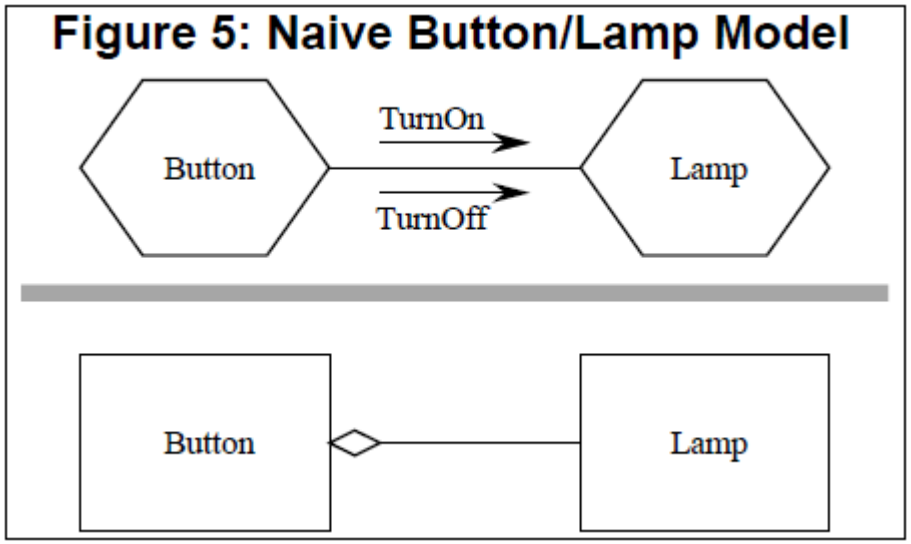
- Uma classe não deve implementar uma interface que ela não precisa.
- Ter interfaces para diferentes responsabilidades.
- Evita a criação de classes “poderosas”.


Client A
Client B
Client C
Service
«client A Methods»
+ ...
«client B methods»
+ ...
«client C methods»
+ ...


Client A
«interface»
Service A
«client A Methods»
+ ...
Client B
«interface»
Service B
«client B Methods»
+ ...
Client C
«interface»
Service C
«client C Methods»
+ ...
Service
«client A Methods»
+ ...
«client B methods»
+ ...
«client C methods»
+ ...

Dependency Injection (ou Inversion)

- Módulos de nível mais alto não devem depender de módulos de nível mais baixo. Ambos devem depender apenas de abstrações.
- Abstrações não devem depender de detalhes. Detalhes devem depender de abstrações.



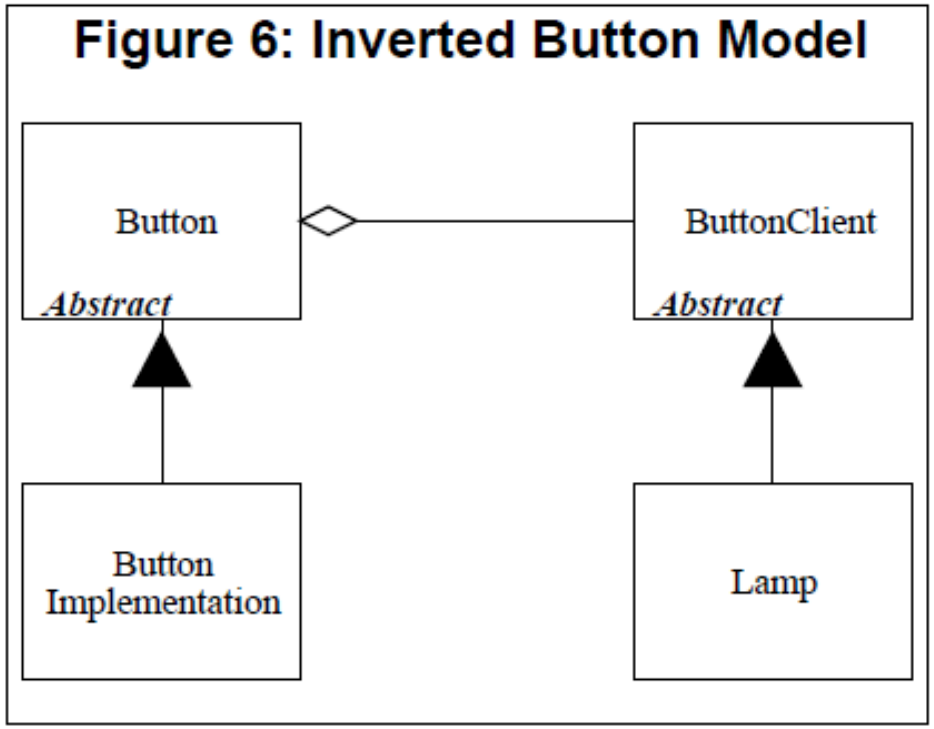


Figure 6: Inverted Button Model
Button
Abstract
ButtonClient
Abstract
Button
Implementation
Lamp

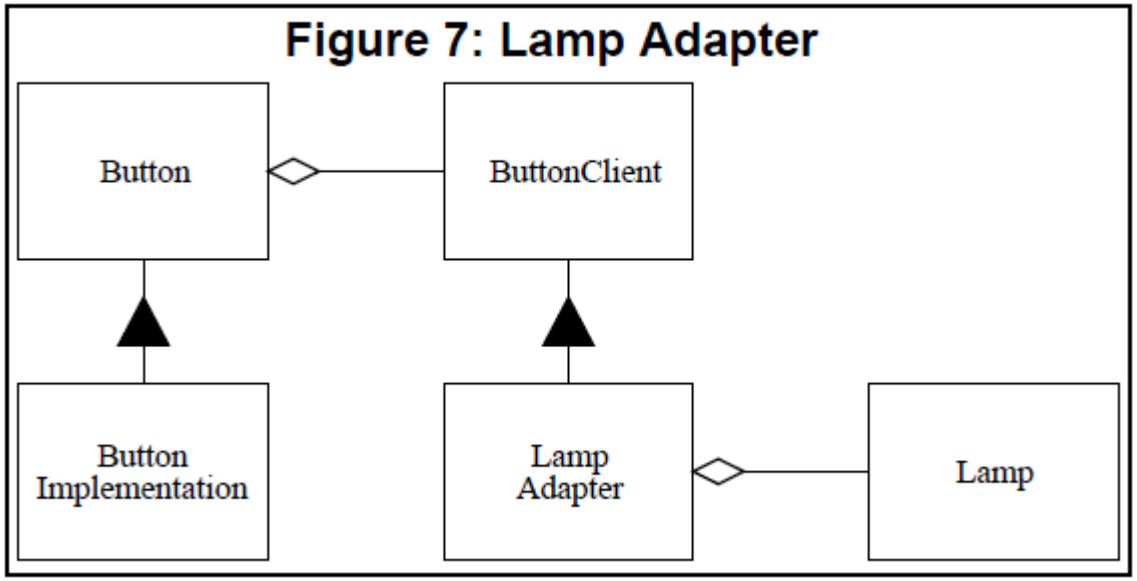
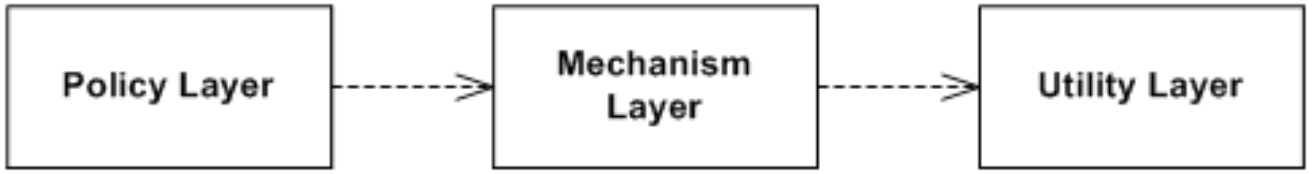
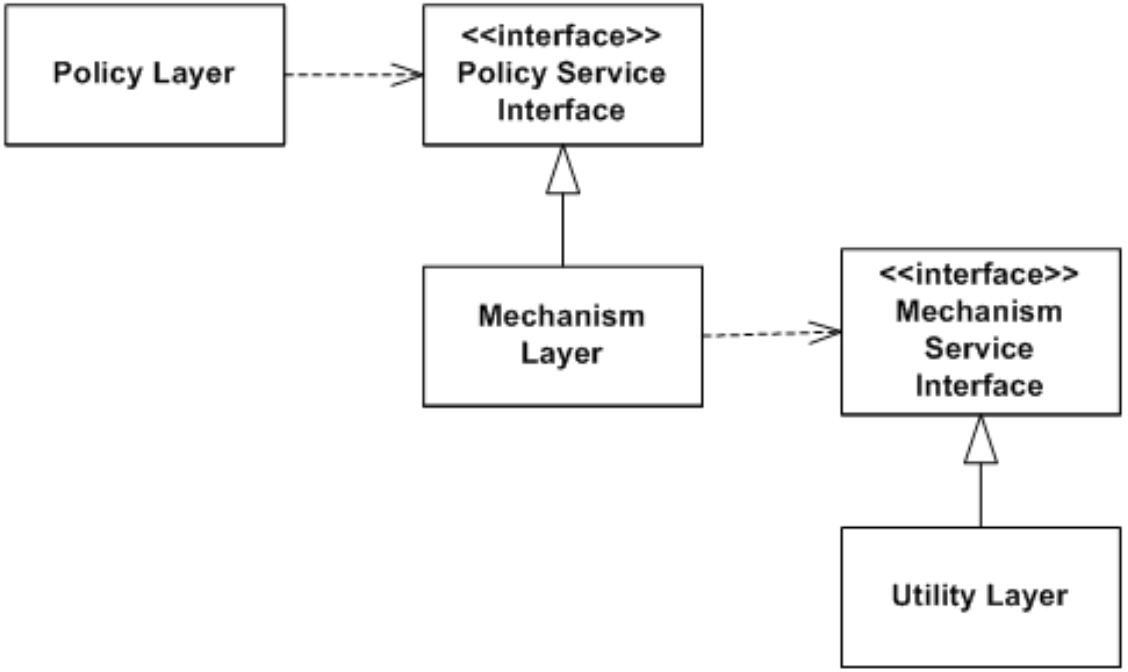
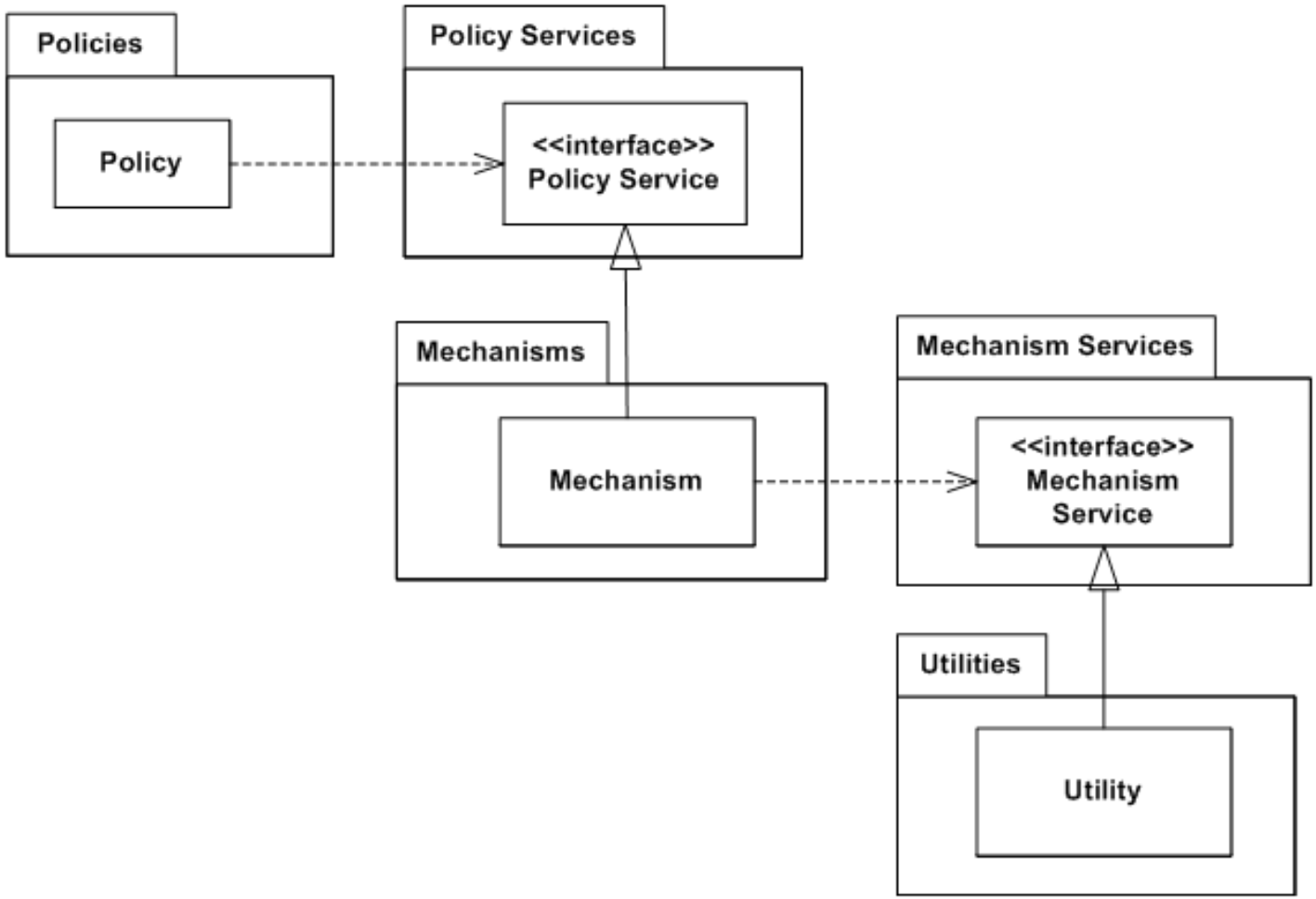
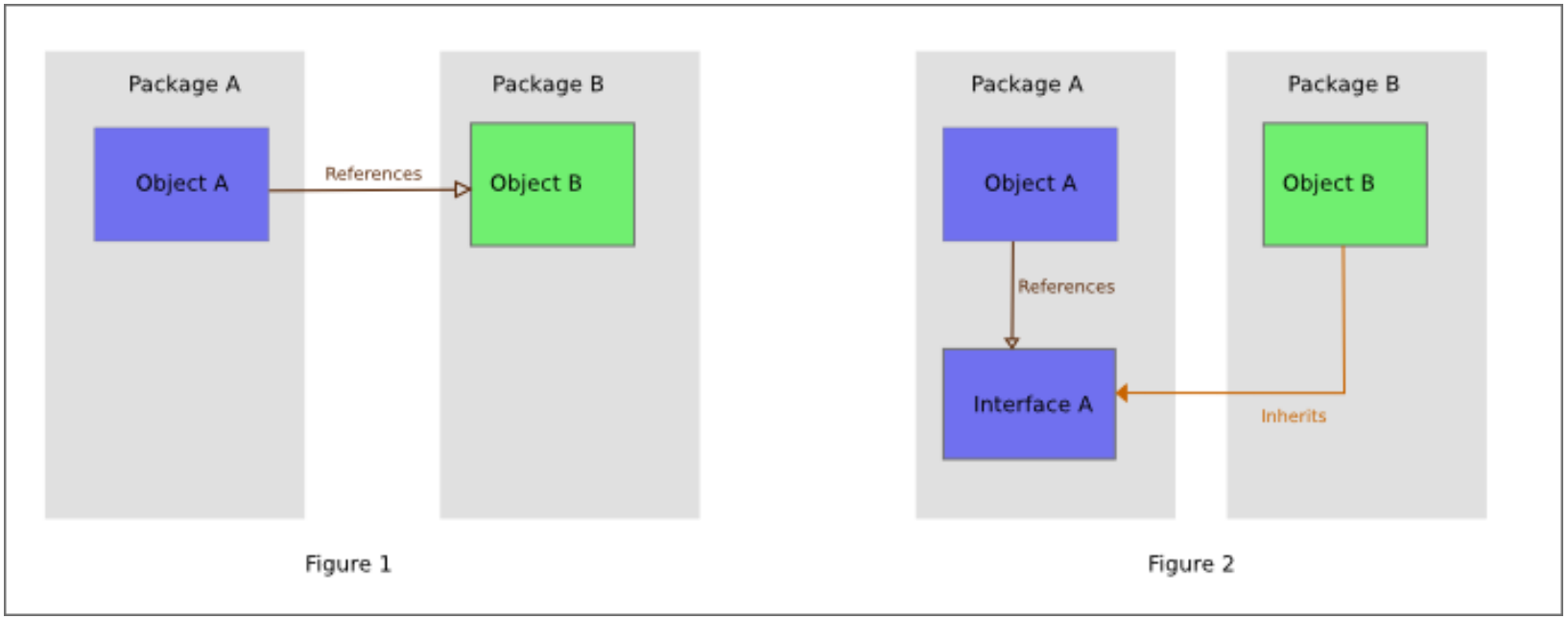


Figure 7: Lamp Adapter
Button
ButtonClient
Button Implementation
Lamp Adapter
Lamp


Policy Layer
Mechanism Layer
Utility Layer


Policy Layer
<<interface>>
Policy Service Interface
Mechanism Layer
<<interface>>
Mechanism Service Interface
Utility Layer


Policies
Policy
Policy Services
<<interface>>
Policy Service
Mechanisms
Mechanism
Mechanism Services
<<interface>>
Mechanism
Service
Utilities
Utility


Package A
Object A
References
Package B
Object B
Figure 1
Package A
Object A
References
Interface A
Package B
Object B
Inherits
Figure 2

Exercício

- Identifique no Projeto do Grupo situações onde os princípios SOLID poderiam ser (ou foram) aplicados.