INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
FLUMINENSE

Princípios de Design Orientado a Objetos

Prof. Mark Douglas de Azevedo Jacyntho

Resumo

- Qualidade em Design de Software;
- Princípios de Design Orientado a Objetos;
- Conclusão.

O que é um Design de Qualidade?

- Um sistema bem projetado é:
  - Fácil de entender;
  - Fácil de mudar / evoluir;
  - Fácil de reutilizar;
  - Prazeroso de se trabalhar.

Sintomas de um Design Pobre

1. Rigidez
   - O sistema é difícil de mudar porque toda mudança gera outras mudanças em outras partes do sistema.
2. Fragilidade
   - Mudanças fazem o sistema parar de funcionar em lugares que não têm relacionamento conceitual com a parte que mudou.
3. Imobilidade
   - É difícil separar o sistema em componentes que possam ser reusados em outros sistemas.
4. Viscosidade
   - Fazer corretamente é mais difícil do que fazer erroneamente.
5. Complexidade Desnecessária
   - O design contém infra-estrutura que não traz benefícios diretos.
6. Repetição Desnecessária
   - O design contém estruturas repetidas que poderiam ser unificadas sob uma única abstração.
7. Opacidade
   - É difícil ler e entender. O design não expressa bem sua intenção.

Gerenciamento de Dependências

- Cada um dos sintomas anteriores é direta ou indiretamente causado por dependências inapropriadas entre módulos do software.
  - Origem do termo “espaguete de código”.
  - A degradação da arquitetura de dependências implica em software não manutenível.
- Dependências precisam ser gerenciadas.
  - Gerenciamento consiste em criar firewalls além dos quais a dependência não se propaga.
  - Os princípios que serão discutidos objetivam criar estes firewalls.

Princípios de Design de Classe

- A seguir serão apresentados os seguintes princípios:

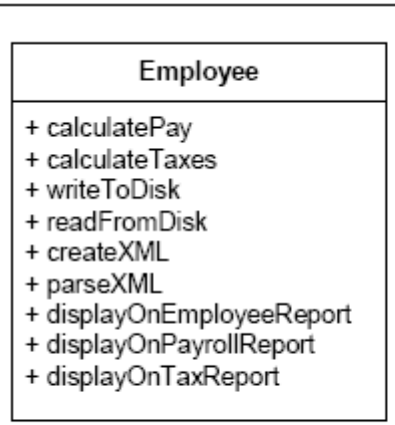
1. SRP – Single Responsibility Principle
2. OCP – Open-Closed Principle
3. LSP – Liskov Substitution Principle
4. DIP – Dependency Inversion Principle
5. ISP – Interface Segregation Principle

SRP – Single Responsibility Principle

- SRP – Single Responsibility Principle
  - “Uma classe deve ter uma única razão para mudar”.
- Classes devem ter uma única responsabilidade.
  - Neste contexto, responsabilidade significa razão para mudar.
  - Mudanças nos requisitos implicam mudanças na responsabilidades das classes.
  - Classes com muitas responsabilidades, têm mais de uma razão p/ mudar.
  - Usuários de uma responsabilidade podem ser afetados pode uma mudança numa responsabilidade completamente diferente (**fragilidade**).

Exemplo de Violação do SRP

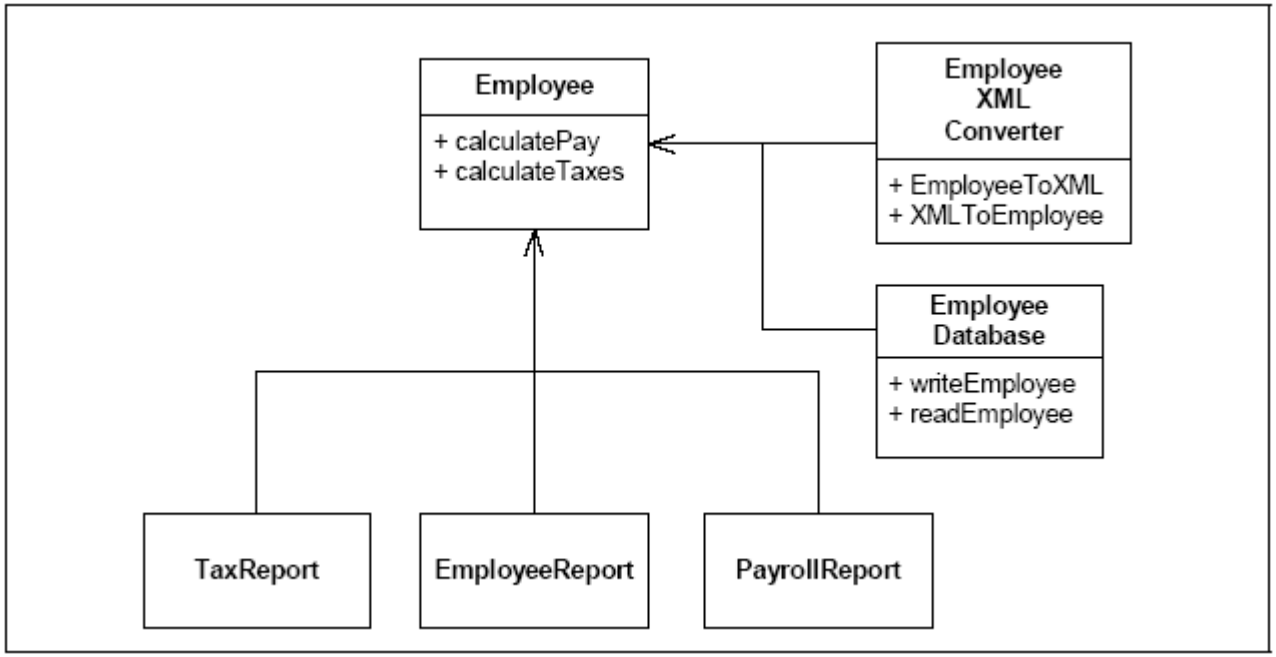
- A Classe Employee sabe muito. Sabe calcular pagamentos e taxas, sabe se persistir no disco, sabe se converter em XML e gerar relatórios.




```
public class Employee {
  public double calculatePay();
  public double calculateTaxes();
  public void writeToDisk();
  public void readFromDisk();
  public String createXML();
  public void parseXML(String xml);
  public void displayOnEmployeeReport(
                        PrintStream stream);
  public void displayOnPayrollReport(
                        PrintStream stream);
  public void displayOnTaxReport(
                        PrintStream stream);
}
```

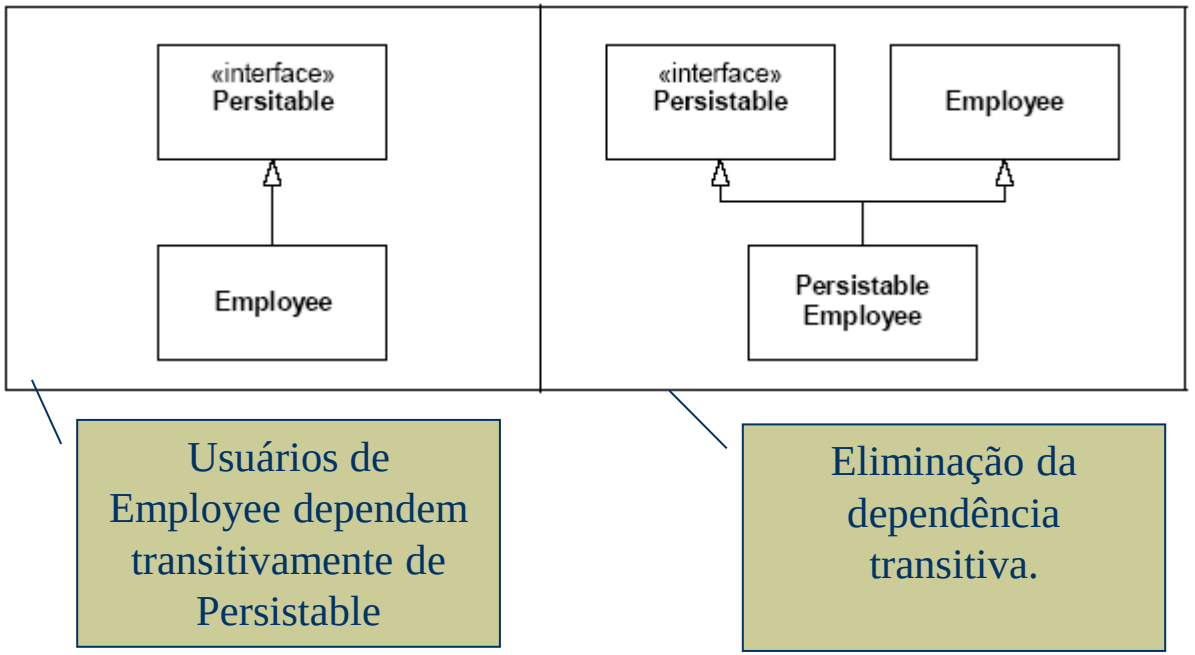
Exemplo em Conformidade com SRP

- Uma classe para cada responsabilidade. Cada classe tem uma única razão para mudar.



Violação Clássica do SRP

- Procure por classes que tenham dependências em mais de uma área.
  - Implementam interfaces com propósitos distintos.
    - P. ex: Mistura de regras de negócio com persistência.



OCP – Open-Closed Principle

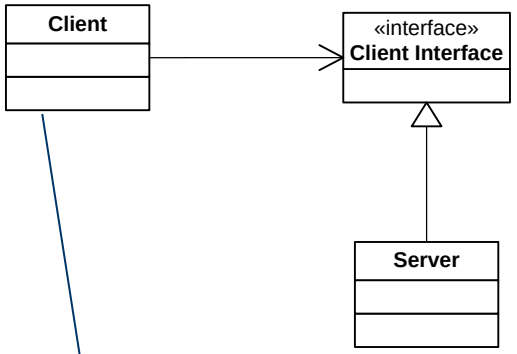
- OCP – Open-Closed Principle
  - Entidades de software (classes, módulos, funções, etc) devem ser abertas para extensão e fechadas para modificação.
- Este princípio enunciado por Bertrand Meyer é o mais importante de todos.
  - Nossas entidades de software devem poder se estendidas sem sofrer modificações.
  - Queremos mudar o que a entidade faz sem mudar seu código.
    - Evitar a rigidez (mudanças em cascata).

OCP – Open-Closed Principle

- “Aberta para Extensão”
  - O comportamento da entidade pode ser estendido para satisfazer mudanças nos requisitos.
- “Fechada para Modificação”
  - Extensão de comportamento não resulta em mudanças no código fonte/binário.
- Abstração é a chave do sucesso.
  - Encapsulamento polimórfico.
    - Encapsule as variações sob um conceito abstrato e conheça somente o conceito abstrato.

OCP - Exemplo

Cliente não é open-closed. Para o cliente usar um outro servidor ele tem que ser modificado.



Cliente é open-closed. O cliente pode ser estendido criando novas subclasses de ClientInterface.

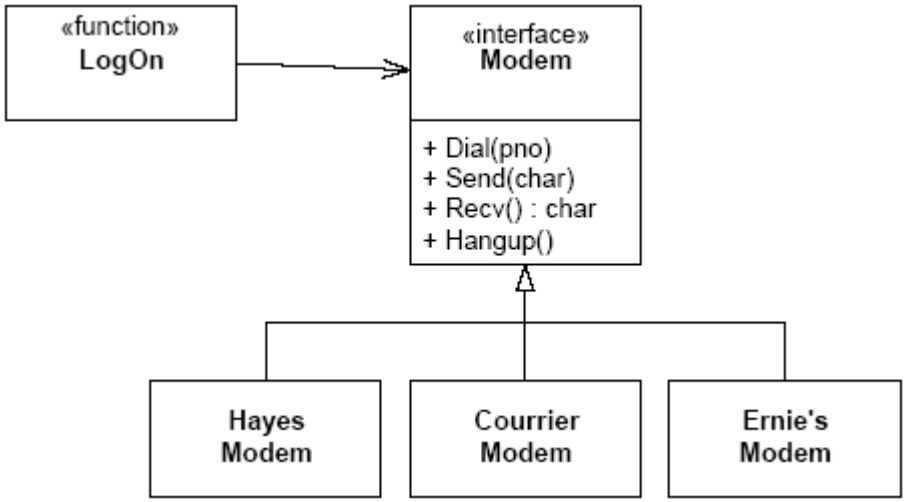
Exemplo de Violação do OCP

- A função `logOn` precisa ser modificada para acomodar novos tipos de modem. Código frágil, rígido e imóvel.

```
public void logOn(Modem m)
{
    switch(m.geType())
    {
        case HayesModem.TYPE:
            this.dialHayes();
        break;

        case CourrierModem.TYPE:
            this.dialCourrier();
        break;

        case ErniesModem.TYPE:
            this.dialErnies();
        break;

        default:
            this.dialOther();
    }

}
```

Exemplo em Conformidade com OCP

- Uso de encapsulamento polimórfico. `LogOn` conhece um conceito abstrato fixo.




```
public void logOn(Modem m)
{
    m.dial();
}
```

LSP – Liskov Substitution Principle

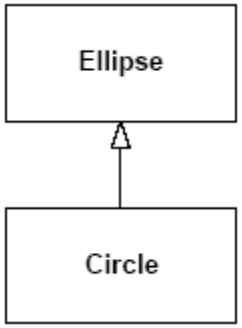
- LSP – Liskov Substitution Principle
  - Subtipos (subclasses) têm que ser substitutos para seus tipos base (classes base) em qualquer contexto.
- Este princípio foi enunciado por Barbara Liskov e também deriva do conceito de “Design By Contract” de Bertrand Meyer.
  - O usuário (cliente) de uma classe base tem que continuar funcionado apropriadamente se um derivado desta classe base for passado para ele.

LSP – Liskov Substitution Principle

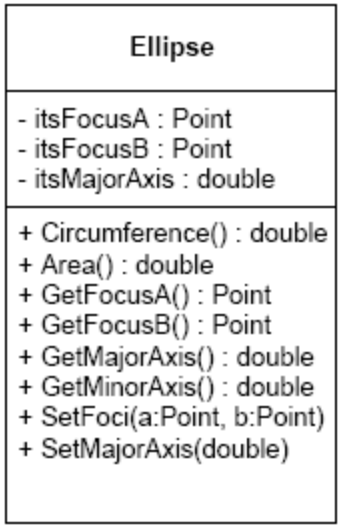
- Usuários de classes base não devem fazer nada especial para usar classes derivadas.
  - Não devem ser usados instanceof ou downcasts.
  - Não devem ter nenhum conhecimento das classes derivadas.
- Se uma função `f` recebe um argumento da classe base, tem que ser legal passar uma instância da classe derivada e o comportamento `f` deve permanecer inalterado.
  - Se uma instância `i` da classe derivada for passada e causar mal funcionamento de `f`, então `f` é frágil na presença de `i`.
    - Se usarmos instanceof para resolver o problema, violaremos o OCP.

Exemplo de Violação do LSP

- O dilema Círculo/Elipse. Círculo é uma elipse com excentricidade nula (focos coincidentes).
  - Com base no relacionamento IS-A modelamos o problema usando herança.



Exemplo de Violação do LSP (cont.)



- Apesar de conceitualmente correto, temos problemas:
  - Elipse tem 3 elementos de dados. Circulo só precisa de 2 (centro e raio).
  - Ignorando o overhead de espaço, para garantir que os dois focos coincidam, fazemos override do método `setFoci`:

```
public void setFoci(Point a, Point b)
{
    this.itsFocusA = a;
    this.itsFocusB = a;
}
```

Exemplo de Violação do LSP (cont.)

- Apesar do modelo está consistente intrinsecamente, Círculo e Elipse se comunicam com outras entidades e provêem suas interfaces públicas que implicam num contrato.
  - P. ex: Usuários de Elipse estão certos que o código seguinte terá sucesso. Porém, se for passado um Círculo o código falhará.

```
public void test(Ellipse e)
{
    Point a = new Point(-1, 0);
    Point b = new Point(1,0);
    e.setFoci(a, b);
    e.setMajorAxis(3);
    Assertiv.isTrue(e.getFocusA().equals(a));
    Assertiv.isTrue(e.getFocusB().equals(b));
    Assertiv.isTrue(e.getMajorAxis() == 3);
}
```

LSP - Design By Contract

- Círculo violou o contrato do método `setFoci` de Elipse que tem como pós-condição:
  - Valores de entrada copiados no campos e o eixo maior inalterado.
- Para ser um substituto a subclasse tem que respeitar o contrato da superclasse.
- Para explicitar o contrato de um método, devemos:
  - Declarar o que tem que ser verdade antes de chamar o método (pré-condição ou assertiva de entrada).
    - Se a pré-condição falhar o resultado do método é indefinido e o método não deve ser chamado.
  - Declarar o que o método garante ser verdade quando terminar (pós-condição ou assertiva de saída)
    - Se a pós-condição falhar o método não deve retornar.

LSP - Design By Contract

- Em termos de contratos, uma classe derivada é um substituto para sua classe base se:

1. Suas pré-condições não são mais fortes do que as do método da classe base.

2. Sua pós-condições não são mais fracas do que as do método da classe base.

- Em outras palavras:
  - Método derivados devem esperar não mais e prover não menos.

LSP x OCP

- Violações de LSP são violações latentes de OCP.
  - P. ex: Para evitar o overhead de mudar o design, poderíamos violar o OCP e resolver o problema do Círculo com `getClass()`.
    - Quando for criada uma nova subclasse de Ellipse está função precisa ser checada para ver se ainda está OK.

```
public void test(Ellipse e) throws NotAnEllipseException
{
    if (e.getClass().equals(Ellipse.class))
    {
        Point a = new Point(-1,0);
        Point b = new Point(1,0);
        e.setFoci(a,b);
        e.setMajorAxis(3);
        Assertiv.isTrue(e.getFocusA().equals(a));
        Assertiv.isTrue(e.getFocusB().equals(b));
        Assertiv.isTrue(e.getMajorAxis() == 3);
    }
    else
        throw new NotAnEllipseException(e);
}
```

LSP – Notas:

- Validade não é intrínseca.
  - Um modelo isolado não pode ser completamente validado.
  - A validade somente pode ser expressa em termos dos clientes.
    - Test-Driven Development.
- Relacionamento IS-A tem a ver com comportamento.
  - Apesar de um Círculo ser uma Elipse, comportamentalmente não o é.
    - Portanto, não deve ser usada herança.
- Método derivados degenerados, usualmente levam a violação do LSP.
  - P. ex: Redefinir um método para não fazer nada.

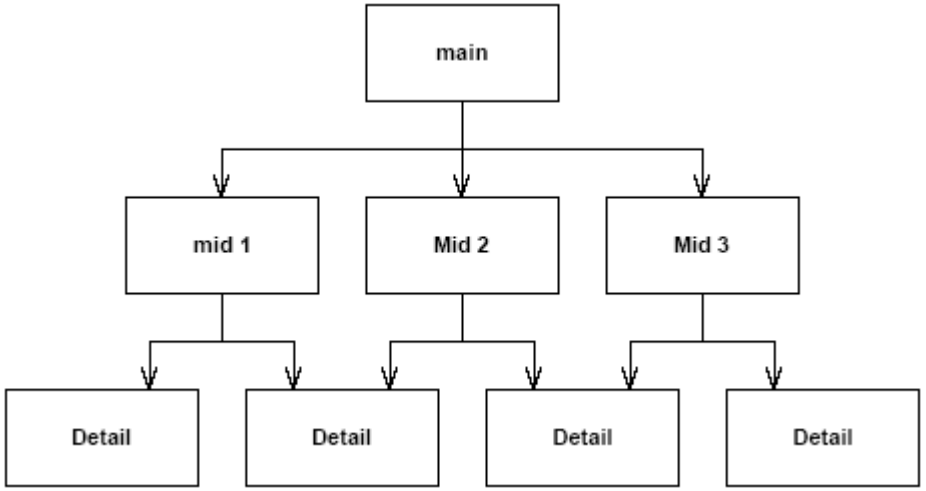
DIP – Dependency Inversion Principle

- DIP - Dependency Inversion Principle
  - Entidades de alto nível não devem depender de entidades de baixo nível. Ambas devem depender de abstrações.
  - Abstrações não devem depender de detalhes. Detalhes devem depender de abstrações.
- O termo inversão é porque numa arquitetura procedural tradicional módulos de alto nível dependem de módulos de baixo nível, ou seja, política depende do detalhe.
  - Num bom design OO, esta dependência é invertida.

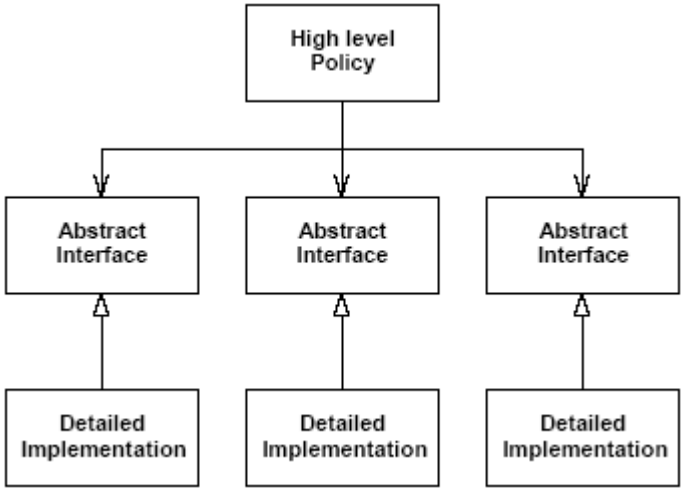
DIP – Dependency Inversion Principle

- São os módulos de alto nível que contêm a política da aplicação (regras e modelos de negócio).
  - Quando este módulos dependem de módulos de baixo nível, temos:
    - Rigidez. Mudança de baixo nível podem ter efeitos diretos nos módulos de alto nível.
    - Imobilidade. Dificulta o reuso de módulos de alto nível.
  - Deve haver uma inversão:
    - Módulos de negócio que devem influenciar os módulos de implementação.
    - Módulos de negócio têm precedência e têm que ser independentes de módulos de implementação.
      - Princípio de frameworks.

DIP – Dependência Procedural x Dependência OO

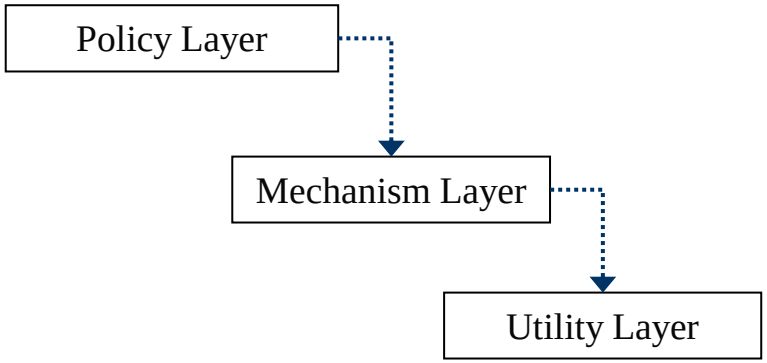


Estrutura de Dependência de uma Arquitetura Procedural

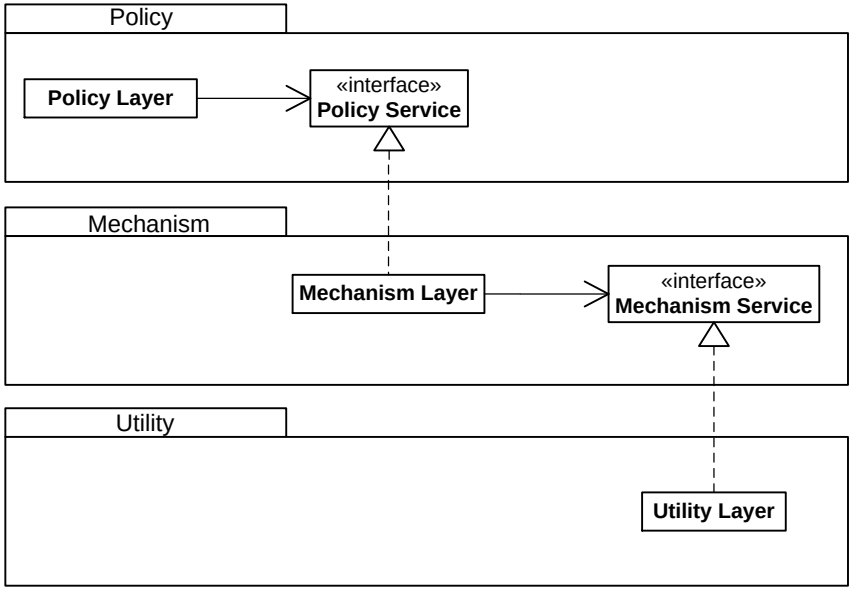


Estrutura de Dependência de uma Arquitetura OO

DIP – Exemplo de Arquitetura em Camadas



Esquema Naive – Violação do DIP

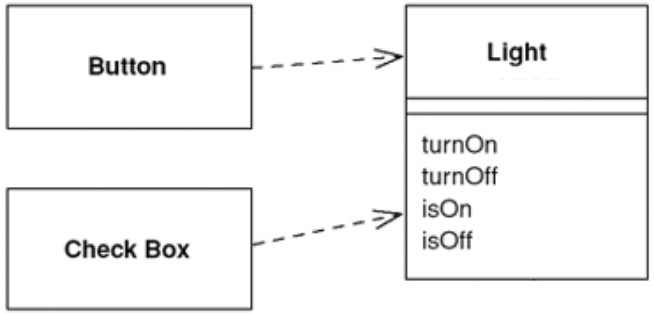


Esquema Invertido – Conformidade com DIP

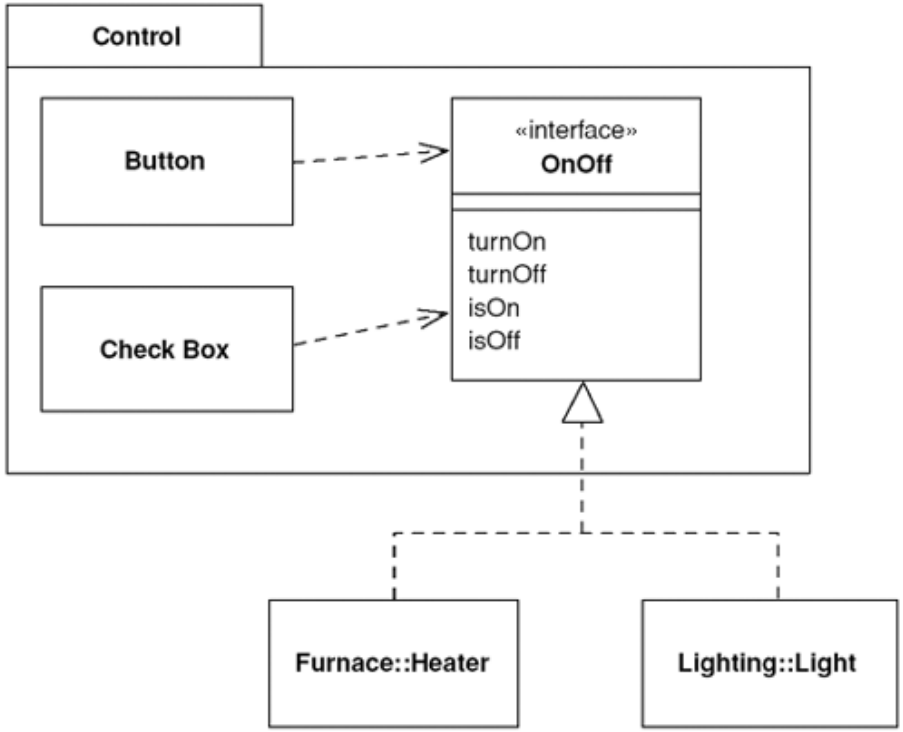
DIP – Dependa de Abstrações

- Heurística. Não dependa de classes concretas voláteis.
  - Coisas concretas mudam com mais freqüência.
  - Conceitos abstratos mudam menos.
  - Abstrações são pontos de extensão.
    - OCP é o objetivo e DIP é o mecanismo primário.
  - É uma boa prática o cliente definir a interface abstrata
    - Mudanças na interface das subclasses concretas não afetam a interface abstrata que as representam;
    - Só vai haver mudança na interface quando o cliente precisar.
  - Depender de classes concretas não voláteis é perfeitamente aceitável.

DIP - Exemplo



Esquema Naive:
Botão pode ser afetado por mudanças na classe Luz.
Botão só pode controlar a classe Luz.



Esquema Invertido:
Botão não é afetado por mudança em subclasses concretas.
Botão por controlar qualquer subclasse.

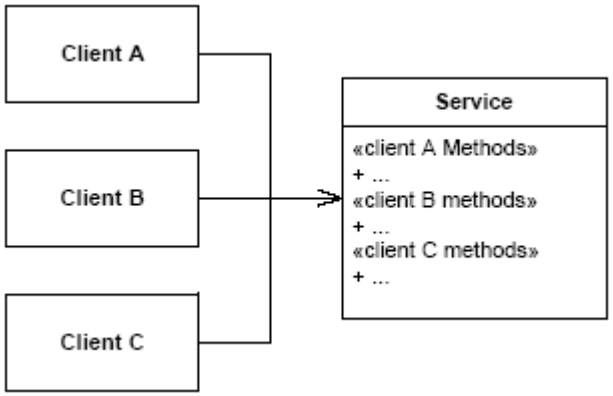
ISP – Interface Segregation Principle

- ISP – Interface Segregation Principle
  - Clientes não devem ser forçados a depender de métodos que eles não usam.

- Várias interfaces específicas para cada cliente é melhor do que uma interface de propósito geral não coesiva.
  - Evita que o cliente seja afetado por mudanças em métodos usados por outros clientes.
    - Evita rigidez e fragilidade.

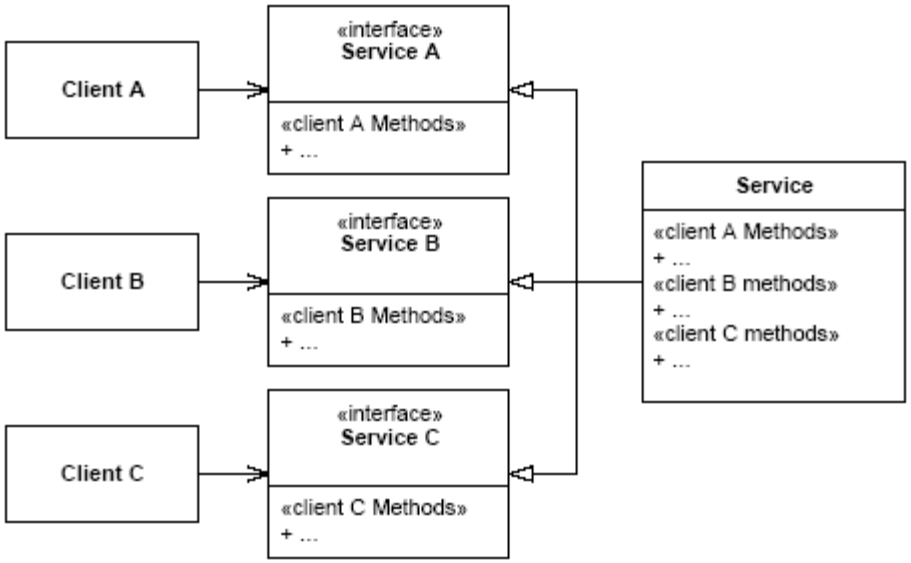
ISP – Interface Segregation Principle

- Uma classe “gorda” com interfaces integradas.
  - Quando uma mudança é feita é um método que o cliente A chama, os clientes B e C podem ser afetados (pode ser necessário recompilá-los e/ou reinstalá-los.)



ISP – Interface Segregation Principle

- Melhor colocar as operações necessárias por cada cliente num interface coesiva específica para aquele cliente.
  - As interfaces são implementadas pela classe.
  - Se a interface do cliente A precisa mudar, os outros clientes não serão afetados de maneira alguma.

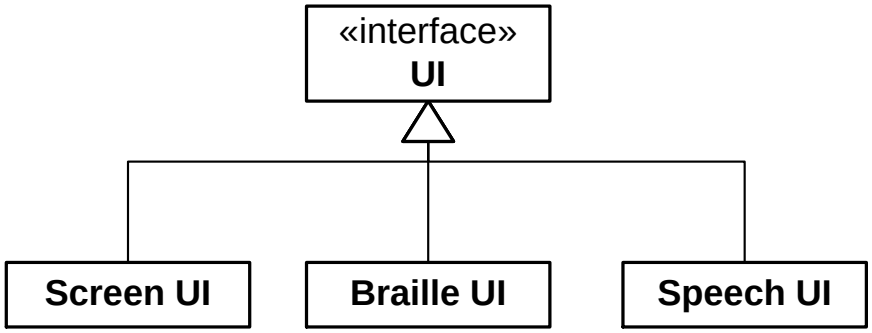


ISP – Interface Segregation Principle

- Qual é o real significado do termo “cliente”?
  - Não significa tem que haver uma interface para exatamente todos os possíveis clientes da classe.
  - Clientes devem ser categorizados por seu tipo e para cada tipo de cliente define-se uma interface.

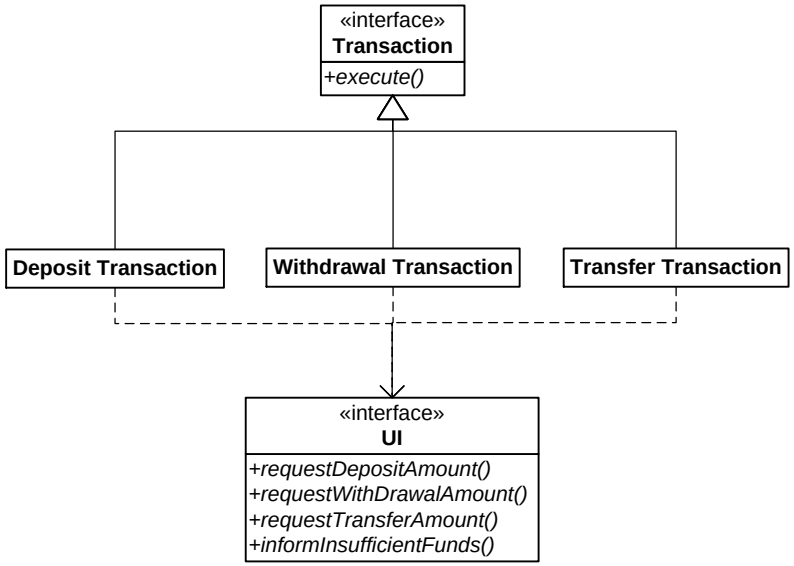
ISP – Exemplo Caixa Eletrônico

- Um caixa eletrônico sofisticado precisa ter uma interface de usuário bem flexível. A saída precisa apresentada: na tela, numa tabuleta em Braille, num sintetizador de voz, etc.
  - Claramente, podemos encapsular estas variações sob um interface abstrata UI com todas as mensagens que precisamos enviar.



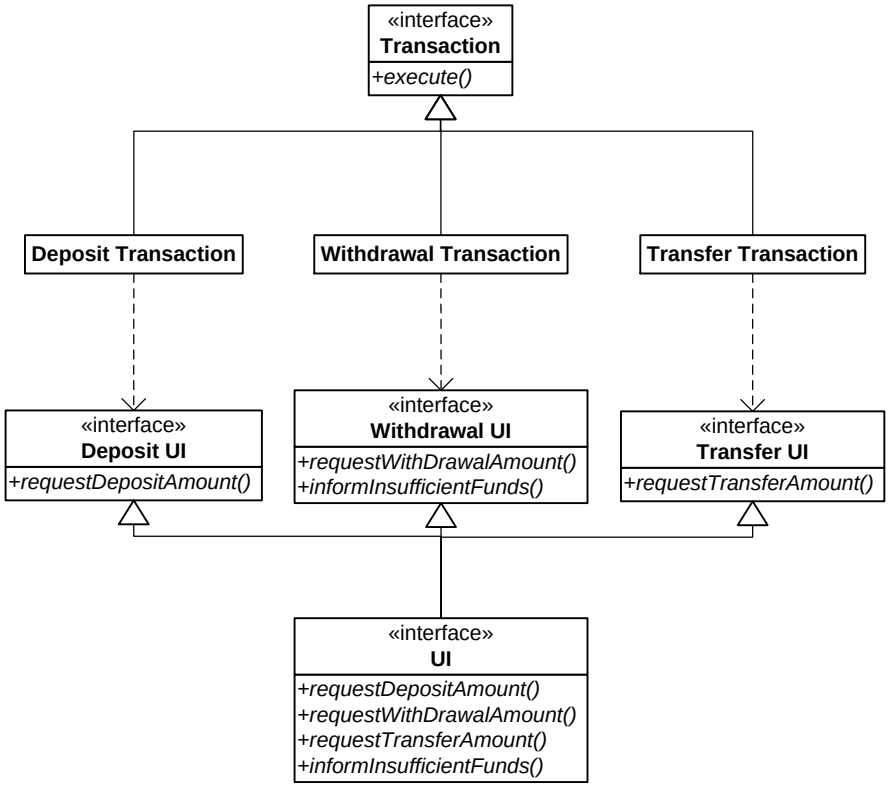
Exemplo de Violação do ISP

- UI - interface não coesiva.
  - Uma mudança em uma subclasse pode gerar mudança em UI e, por conseguinte, afetar outra subclasse que não tem nada a ver com a mudança.



Exemplo em Conformidade com ISP

- Interfaces segregadas, específica para cada tipo de transação.



Conclusão

- Foram apresentados 5 princípios que tendem tornar o design flexível, resiliente e reutilizável.

- Quando aplicar estes princípios?
  - Quando for detectado um problema estrutural que torna o sistema não manutenível.
  - É impossível ter todo o sistema em conformidade com todos os princípios todo tempo. Uma complexidade desnecessária.

- Devemos:
  1. Detectar o problema (p. ex: com casos de testes antecipados);
  2. Diagnosticar o problema aplicando os princípios;
  3. Resolver o problema aplicando o design pattern apropriado.

Referências

- MARTIN, Robert C. Agile software development: principles, patterns, and practices. Prentice Hall, 2002. ISBN: 0135974445