# DIM00506
# Projeto Detalhado de Software

Professor: Uirá Kulesza

DIMAp / UFRN, 2013.2

# Aula 4:
# Princípio Liskov Substitution

# Uso de Herança

- Os principais mecanismos por trás do princípio open-close são abstração e polimorfismo
- Tais mecanismos são tipicamente implementados na forma de herança em linguagens OO fortemente tipadas
- Subclasses podem estender classes abstratas de forma não invasiva

Quais são as regras para uso de herança?

Quais são as características de boas hierarquias de herança?

Quais são as armadilhas para criar hierarquias de herança que não estão de acordo com o princípio Open-Closed?

# Princípio Liskov Substitution

- “Tipos base podem ser substituídos por seus subtipos”

[Liskov88] "Data Abstraction and Hierarchy," Barbara Liskov, SIGPLAN Notices, 23(5) (May 1988).

- Substituição do tipo por seus subtipos

# Exemplo

- Dado um método:
  - public void m(B b)
- Se passarmos para m, um objeto da classe D que deriva da classe B
- Caso m() não funcione corretamente, nós dizemos que D viola o princípio de substituição
  - Dizemos que D é frágil na presença de m()

# Outro Exemplo...

- Classe Shape
  - Método estático drawShape(Shape s)
    - Imprime os shapes chamando seus respectivos métodos draw()
- Classes Square e Circle
  - Método draw() para imprimir suas informações

# Código de Shape

```
public class Shape {
   ...

   public static void drawShape(Shape shape){
      if (shape instanceof Circle){

         ((Circle) shape).draw();

      }else if (shape instanceof Square){

         ((Square) shape).draw();
      }

    }

}
```

# Código de Circle

```
public class Circle extends Shape {

    int x, y;
    int radius;

    public Circle(int x, int y, int radius){
        this.x = x;
        this.y = y;
        this.radius = radius;
    }

    public void draw() {
        System.out.println("A Circle");
        System.out.println("   X = " + x);
        System.out.println("   Y = " + x);
        System.out.println("   Radius = " + x);
        System.out.println();
    }

}
```

# Existe violação do princípio de Liskov ?

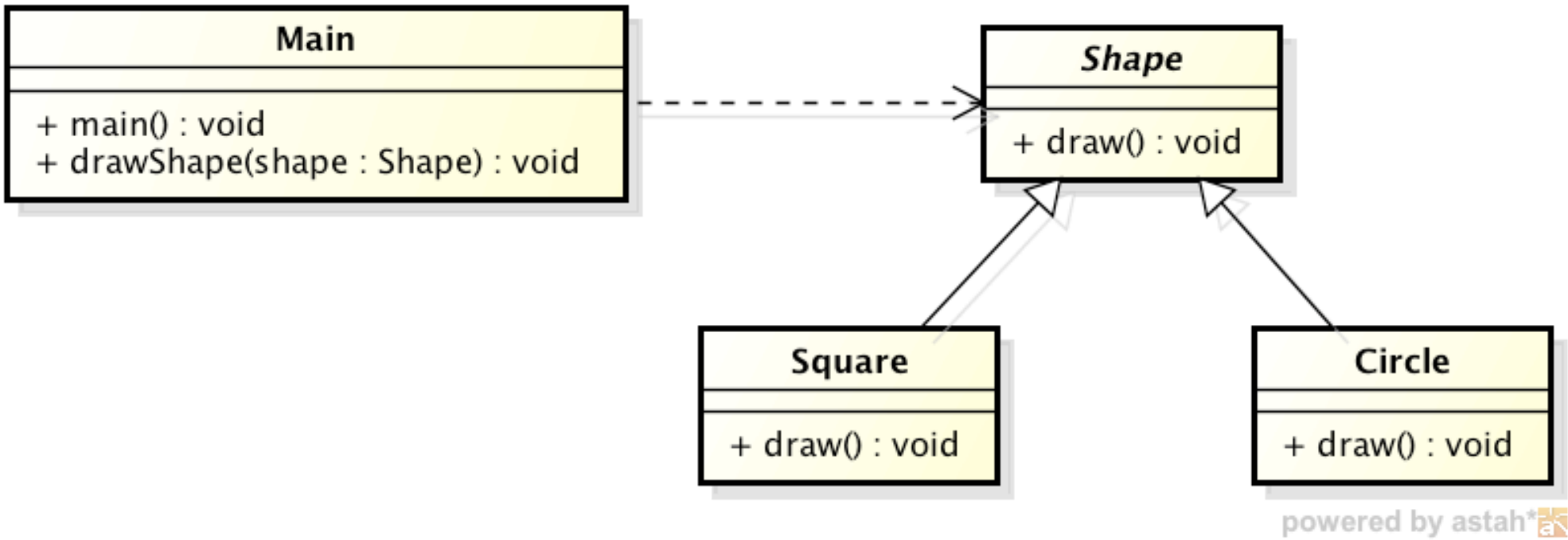
Square e Circle são substituíveis por Shape ?

Por que ?

Não são substituíveis, porque é necessário fazer cast de ambos no método drawShape(), para permitir a chamada a seus respectivos métodos draw()

Como resolver tal problema?

# Solução


Main
+ main() : void
+ drawShape(shape : Shape) : void
Shape
+ draw() : void
Square
+ draw() : void
Circle
+ draw() : void
powered by astah*

# Novo código de Shape

```
public abstract class Shape {

    public abstract void draw();

}
```

# Código de Circle (idêntico)

```
public class Circle extends Shape {

    int x, y;
    int radius;

    public Circle(int x, int y, int radius){
        this.x = x;
        this.y = y;
        this.radius = radius;
    }

    public void draw() {
        System.out.println("A Circle");
        System.out.println("   X = " + x);
        System.out.println("   Y = " + x);
        System.out.println("   Radius = " + x);
        System.out.println();
    }

}
```

# Método de impressão de shapes

```
public class Main {

  public static void main(String[] args) {

     Shape shape1 = new Circle(5, 5, 10);
     Shape shape2 = new Square(5, 5);

     Main.drawShape(shape1);
     Main.drawShape(shape2);

  }

  public static void drawShape(Shape shape){
     shape.draw();
  }

}
```

# Exemplo: Projeto de uma API

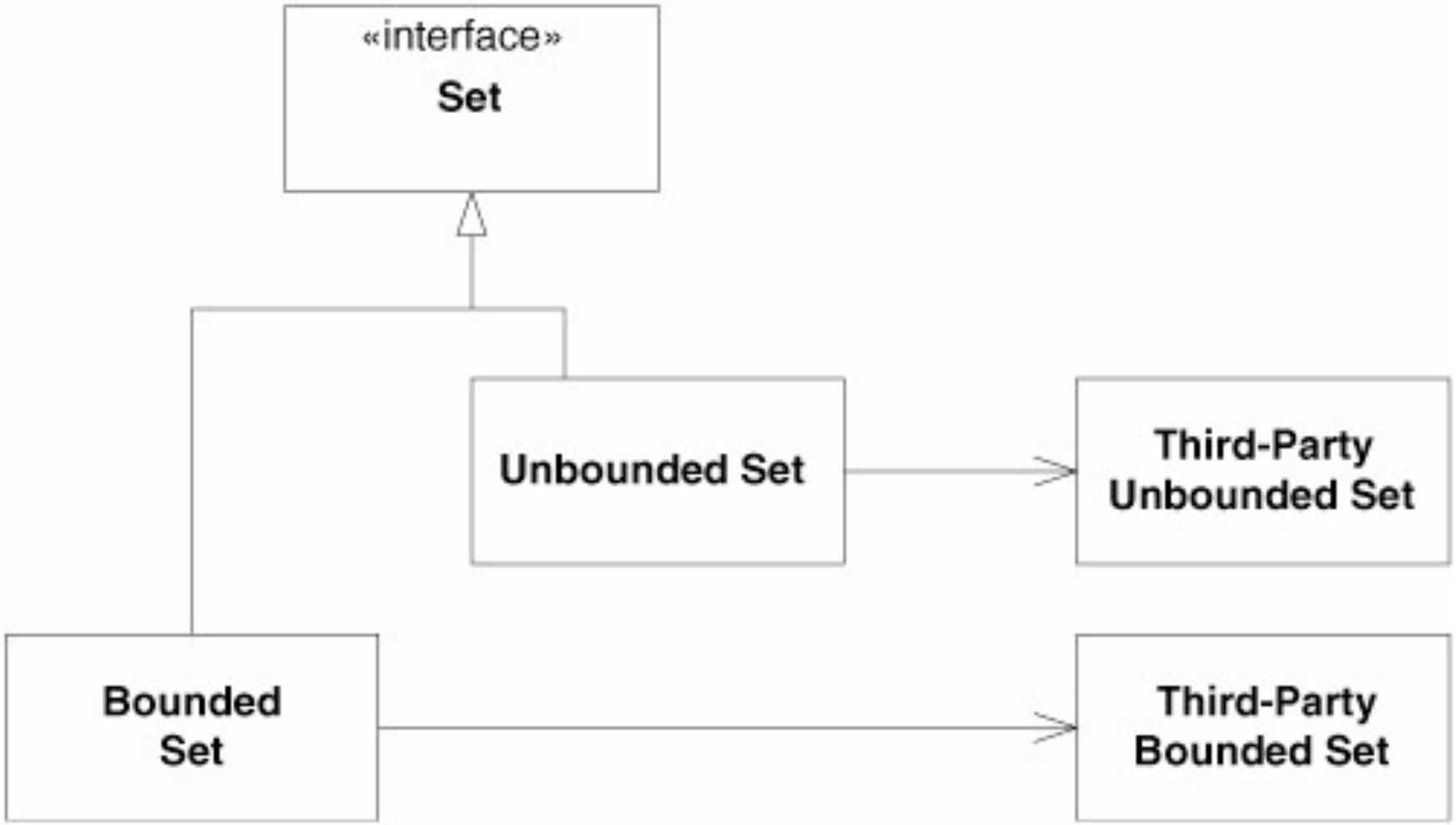
- Compra de classes Set e Bag de uma empresa terceirizada

- Implementação de Set de dois tipos:
  - BoundedSet: tamanho limitado baseado em alocação estática (array)
    - Maior gasto de memória, mais rápido
  - UnboundedSet: tamanho ilimitado baseado em alocação dinâmica (listas ligadas)
    - Economia de espaço, mais lento

Classes compradas possuem métodos com nomes e tipos de argumentos não adequados para a aplicação

Como tornar a aplicação independente das interfaces/ assinatura das classes específicas compradas?

# Projeto Independente



# Interface Set

```
public interface Set {

    public void add(Object o);
    public void delete(Object o);
    public boolean isMember(Object o);
    public List elements();

}
```

# Classe BoundedSet

```
public class BoundedSet implements Set {
    CompanyBoundedSet boundedSet = null;

    public BoundedSet(int total){
       this.boundedSet = new CompanyBoundedSet(total);
    }

    public void add(Object obj) {
       this.boundedSet.insert(obj);
    }

    public void delete(Object obj) {
       this.boundedSet.remove(obj, true);
    }

    public boolean isMember(Object obj) {
       return this.boundedSet.exists(obj);
    }
}
```

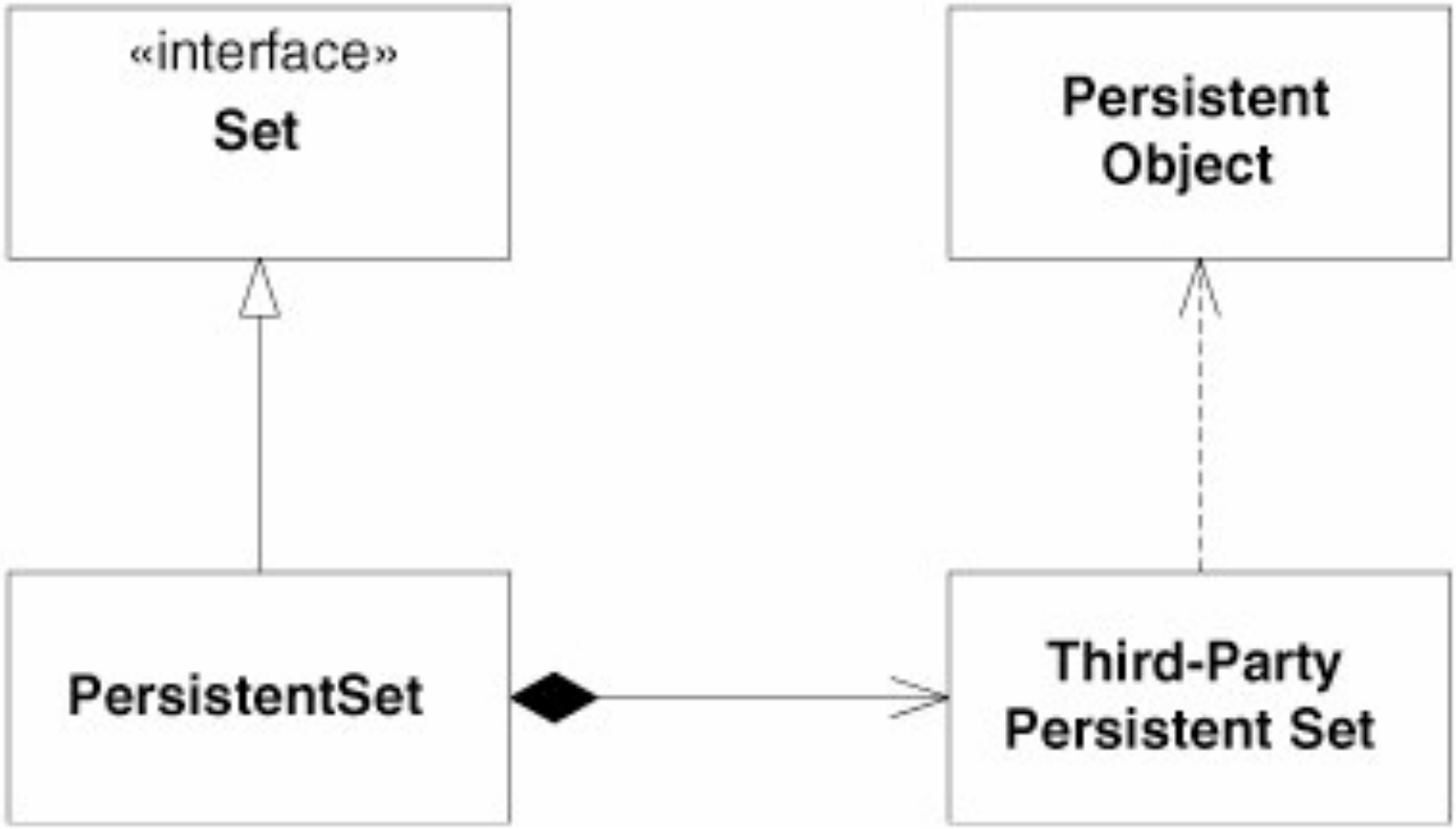
# Projeto Independente

- Código de aplicação cliente, pode permutar entre implementações de SetBounded e SetUnbounded livremente

```
public void printObjects(Set mySet){

   List elementos = mySet.elements();

   // Itera na lista de elementos
      // imprime cada elemento

}
```

Como evoluir tal projeto para tratar um PersistentSet que foi comprado de um outro fornecedor de API ?

# Projeto com PersistentSet



# Problema

- Classe PersistentSet comprada armazena apenas objetos que derivam de PersistentObject (classe ou interface)
- Clientes de classe Set têm que necessariamente passar objetos do tipo PersistentObject para serem armazenados
- Solução usando cast dentro do método add() em PersistentSet (ver próximo slide)

# Problema

```
public class PersistentSet implements Set {
    CompanyPersistentSet persistentSet = null;

    public PersistentSet(...){
       this.persistentSet = new CompanyPersistentSet(...);
    }

    public void add(Object obj) {
       PersistentObject pObj = (PersistentObject) obj;
       this.persistentSet.insert(pObj);
    }
    ...
}
```

- Qual o problema com esse código?

# Problema

```
public class PersistentSet implements Set {
    CompanyPersistentSet persistentSet = null;

    public PersistentSet(...){
       this.persistentSet = new CompanyPersistentSet(...);
    }

    public void add(Object obj) {
       PersistentObject pObj = (PersistentObject) pObj;
       this.persistentSet.insert(pObj);
    }
    ...
}
```

- Erro em tempo de execução caso objeto passado para método add() não seja da instância de PersistentObject

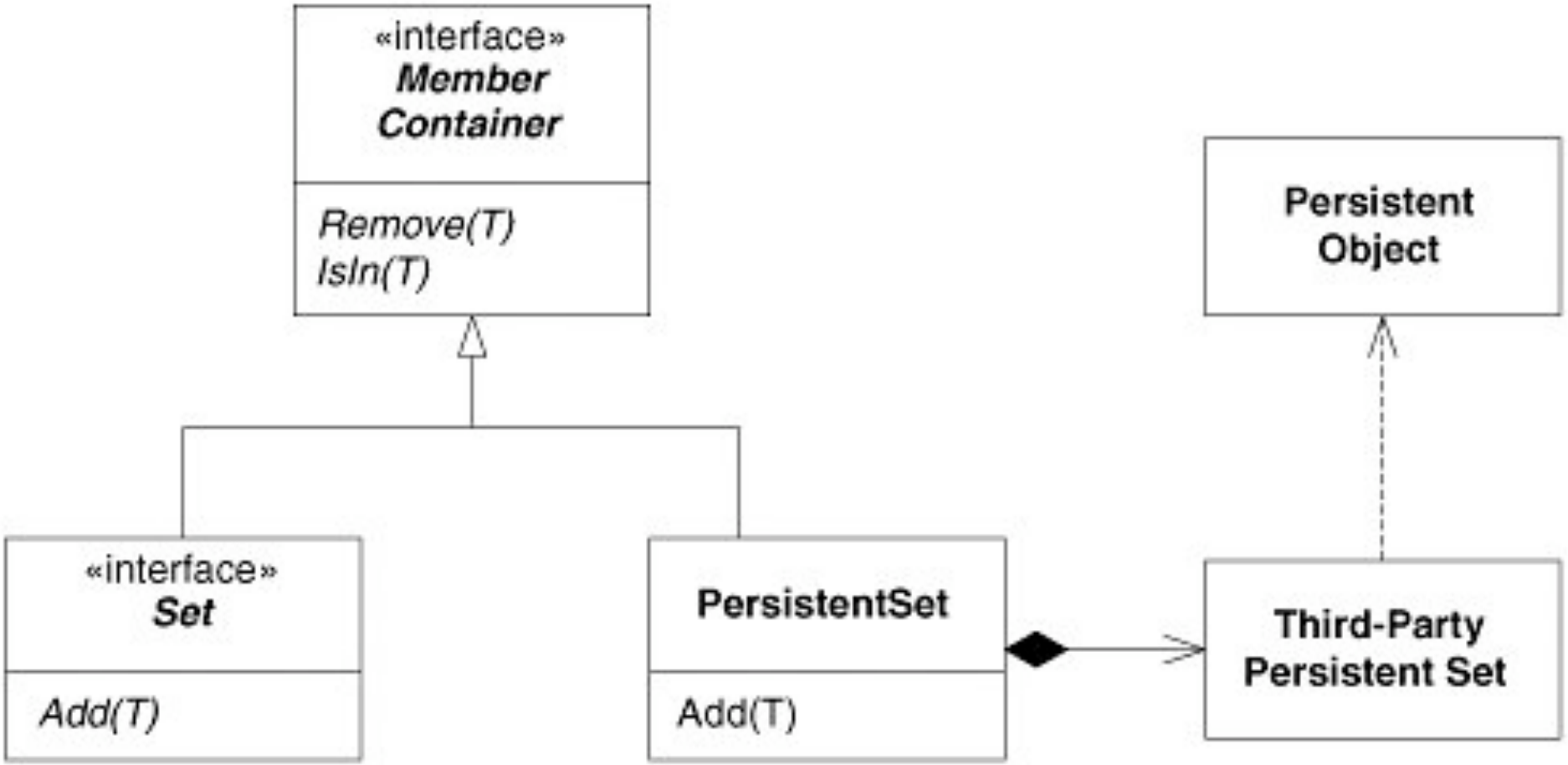
Como resolver tal
problema de projeto com classe
PersistentSet ?

# Solução

- PersistentSet não tem uma relação “é-um” (is-a) com Set
  - Porque não implementam ambos o mesmo método add()

- Criação de hierarquias separadas garante interface/assinatura específica de método add() diferente

# Solução


«interface»
Member
Container
Remove(T)
IsIn(T)
Persistent
Object
«interface»
Set
Add(T)
PersistentSet
Add(T)
Third-Party
Persistent Set

# Heurística / Convenção

- Uma boa heurística para verificar o princípio de substituição é a seguinte:
  - Uma classe A que implementa menos comportamentos do que sua superclasse B não é, em geral, substituível
  - Neste caso, A viola o princípio
- Exemplos:
  - Uma classe B herda de A sobrepõe alguns de seus métodos, mas não oferece nenhuma implementação para eles

# Considerações Finais

- O princípio de Liskov substitution ajuda a garantir o princípio open-closed
- Permite que um módulo definido em termos de um tipo seja estendido sem modificação, através da possibilidade de substituição do tipo por seus subtipos
- O tipo base deve definir um contrato claro e bem definido, que sempre que possível é forçado pelo código
- Embora, "é-um" possa servir de indicador para uso de herança, substituibilidade é o que realmente traz segurança para criar um subtipo

# Aplicação de Princípios e Padrões

- Terça – 20/Agosto:
  - Entrega do relatório de aplicação dos princípios
- Terça e Quinta – 20 e 22/Agosto
  - Apresentação dos projetos
  - Sorteio no dia 20/Agosto
  - Sorteio de padrões para estudar
- Terça – 27/Agosto
  - Aula de padrões
- Quinta – 29/Agosto
  - Dúvidas sobre padrões
- Terça e Quinta – 03 e 05/Setembro
  - Apresentações dos grupos dos padrões

# Grupos

- Grupo 1: Luis Rogerio / Matheus
  - Projeto Locadora
- Grupo 2: Igor / Lucas Bibiano
  - Projeto ??
- Grupo 3: Luan / Andreza
  - Projeto ??
- Grupo 4: Raul / Hivana
  - Projeto ??
- Grupo 5: Rafael Lins / Ezequely
  - Projeto ??
- Grupo 6: Wendell / Adelino Segundo
  - Projeto ??
- Grupo 7: Alisson / Paulo
  - Projeto de Controle de Casas
- Grupo 8: Maychell / Silas (faltou)
  - Projeto Gerenciador de Estacionamento
- Grupo 9: Lucas / Rafael
  - Projeto Gerenciador de Estacionamento

# Referências


- R. Martin, Agile Software Development: Principles, Patterns and Practices, Prentice Hall, 2002.