REFLEXÃO EM JAVA

INVERSÃO DE DEPENDÊNCIA
FACTORY METHODS
FACTORY CLASSES
IoC, CONTAINERS e BEANS
SPRING

PARTE III

CLASS<T>

- A partir de JAVA5 a classe java.lang.Class passou a ser uma classe genérica definida como Class<T>;

- O tipo parâmetro T representa o tipo que uma instância desta classe representa.

  String.class é do tipo Class<String>

- A classe Class<T> contém métodos que nos permitem saber em tempo de execução todas as informações de definição da classe T, e até utilizá-los de forma anónima (não explícita) para criar instâncias usando diferentes construtores, etc.

- De forma simples, quando temos uma instância de uma dada classe, o método getClass() devolve uma Class<?> que é a classe do objecto:

```
Ponto2D p = new Ponto2D(0, -1);
Class<?> cp = p.getClass();
 String cname = p.getClass().getSimpleName();
```

CLASS<T> - API

Métodos principais de Class<T>	
`Class<?> forName(String nm)`	Devolve a `Class<>` cujo nome é dado
`Class[] getClasses()`	Devolve um *array* com todas as classes e interfaces públicas que são membros da classe receptora
`Class<?> getComponentType()`	Devolve a classe dos componentes do *array*
`Construtor[] getConstructors()`	Devolve *array* de construtores
`Class[] getDeclaredClasses()`	Devolve um *array* com todas as classes e interfaces que são membros da classe receptora
`Field/Method/Constructor getDeclaredTipo()`	Devolvem campos, métodos e construtores declarados
`Class<?> getDeclaringClass()`	Devolve a classe de que é membro
`Class[] getInterfaces()`	Determina as interfaces implementadas
`Type getGenericSuperClass()`	Devolve o tipo da superclasse directa
`Method getMethod(String m, Class... tipoParametros)`	Devolve o método público especificado; 2º parâmetro `null` para método sem parâmetros
`Method[] getMethods()`	*Array* de métodos públicos
`String getName()`	Devolve o nome desta classe/interface
`String getSimpleName()`	Devolve o nome simples desta classe/interface
`boolean isInterface()`	Determina se é uma interface
`isEnum(), isArray(), isPrimitive(), isLocalClass()`	Determinações sobre o tipo da classe
`boolean isInstance(Object o)`	Verifica se parâmetro é compatível para atribuição
`T newInstance()`	Cria uma instância desta classe
`String toString()`	Converte para `String`

CLASS<T> - REFLEXÃO

Estes são os métodos principais, dos quais já usámos alguns ao longo dos capítulos anteriores, como por exemplo:

```
out.println(obj.getClass().getSimpleName());
Class[] inters = hi9.getClass().getInterfaces();
for(Class c : inters) out.println(c);
```

A maior parte dos resultados destes métodos devolve entidades que estão definidas no *package* `java.lang.reflect`, tais como `Field`, `Method`, `Type`, etc. A cada um destes objectos são ainda aplicáveis métodos que permitem obter sobre os mesmos informação mais detalhada, por vezes muito útil. A classe `Reflection` oferece também serviços de validação de operações baseadas em reflexão.

As antigas colecções de JAVA sempre foram muito parcas em informação para reflexão, pois sobre os tipos dos elementos nelas contidos nada era registado. Muitos destes métodos são novos e permitem obter informação de **tipo genérico**. Por exemplo, o tipo `Field` possui agora um método que permite obter o tipo genérico da variável, método de nome `getGenericType()`.

CLASS<T> - REFLEXÃO

- O método `newInstance()` de `Class<T>` devolve agora um `T` em vez de um `Object`, pelo que é possível criar instâncias de uma classe qualquer de forma segura usando o mecanismo de reflexão.

```
String s = String.class.newInstance();
Ponto2D p = Ponto2D.class.newInstance();
```

- Se um método tem como parâmetro de entrada `Class<T>`

```
public static <T> T mkInstance(Class<T> clazz) { }
```

A sua invocação é, em geral, feita usando uma classe literal, cf.

```
out.println(mkInstance(String.class));
out.println(mkInstance(Ponto2D.class));
```

CLASS<T> - REFLEXÃO

- O MÉTODO GENÉRICO TÍPICO `mkInst(), mkInstance()` OU `mkObject()` QUE USA O CONSTRUTOR VAZIO DA CLASSE, TEM UMA CODIFICAÇÃO MUITO SIMPLES BASEADA NO MÉTODO `newInstance()`:

```
public static <T> T mkInstance(Class<T> clazz) {
    try {
        return clazz.newInstance();
    }
    catch(Exception e) { out.println(e.getMessage()); return null; }
}
```

NOTA: Obrigatório usar try/catch !!

CLASS<T> - REFLEXÃO

- Assim já poderemos criar instâncias dos tipos parâmetro.

```
public class ParGen<P, S> {
   // Construtor genérico
   public ParGen() {
      p = mkInst(P.class); s = mkInst(S.class);
}
```

- Mas não funciona porque os **tipos parâmetro** de classes genéricas **não possuem classe literal**. Temos que usar a sua **Classe<T>**.

```
public class ParGen<P, S> {
   //...
   public ParGen(Class<P> tipoP, Class<S> tipoS)
      { p = mkInst(tipoP); s = mkInst(tipoS); }
```

CLASS<T> - REFLEXÃO

Retomemos então o problema da criação de objectos genéricos. Consideremos uma classe genérica simples e um seu construtor:

```
public class ParGen<P, S> {
   ...
   public ParGen() { p = new P(); s = new S(); } // ERRO
```

A criação de um objecto genérico é apenas possível através de reflexão. Vimos antes alguns métodos de `Class<T>` que indiciam como tal poderá ser feito. O esquema é criar um designado ***factory method***, um método estático, `private` caso não seja invocável do exterior da classe, e que cumpre a missão de criar um objecto instância do tipo de um dado parâmetro `T`.

```
private static <T> T mkInst(Class<T> clazz) {
   try {
     return clazz.getConstructor(new Class[0]).
                  newInstance(new Object[0]);
   }
   catch(Exception e) { return null; }
}
```

arrays vazios !!

REFLEXÃO – FACTORY METHODS

- Para evitar que o utilizador tenha que escrever código muito Redundante como:

```
ParGen<String, Integer> par =
   new ParGen<String, Integer>(String.class, Integer.class);
```

- A solução é criar um FACTORY METHOD que encapsule todo o processo. Tipos são automaticamente inferidos !

```
public static <X, Y> // mét. genérico; X e Y são tipos param.
   ParGen<X, Y> mkDefault(Class<X> tipoX, Class<Y> tipoY) {
     return new ParGen<X, Y>(mkInst(tipoX),mkInst(tipoY));
```

- A criação de uma instância passará a ser:

```
ParGen<String, Integer> par =
   ParGen.mkDefault(String.class, Integer.class);
```

FACTORY METHODS

- Por tudo isto a maior parte dos exemplos sobre Factory Methods que podem ser encontrados na Net estão obsoletos !!

- A utilização de factory methods torna-se evidente por exemplo Quando se usam os novos tipos enumerados de JAVA5.

```
Enum.valueOf(Direccao.class, dir);
EnumSet<Cafe> temp = EnumSet.noneOf(Cafe.class);
```

Passamos a ter a forma **NomeClasse.factoryMethod()** em vez da usual forma baseada em **new construtorClasse()**.

FACTORY METHODS

- Por tudo isto a maior parte dos exemplos sobre Factory Methods que podem ser encontrados na Net estão obsoletos !!

- A utilização de factory methods torna-se evidente por exemplo Quando se usam os novos tipos enumerados de JAVA5.

```
Enum.valueOf(Direccao.class, dir);
EnumSet<Cafe> temp = EnumSet.noneOf(Cafe.class);
```

Passamos a ter a forma **NomeClasse.factoryMethod()** em vez da usual forma baseada em **new construtorClasse()**.

OUTROS MÉTODOS CRUCIAIS

- `Class<?> forName(String className)`

```
public static Motor mkMotor() {
    try {
        Class<?> cm = Class.forName("Motor");
        return (Motor) cm.newInstance();
    }
    catch(Exception e) { out.println(e.getMessage()); return null; }
}
```

Nota: Depois de se obter **Class<?>** note-se que o uso de **mkInstance()** está sujeito a "casting" por razões que têm a ver com o mecanismo de **"wildcard capture"**.

OUTROS MÉTODOS CRUCIAIS

- `import java.lang.reflect.Constructor;`
- `Constructor<?> getConstructor(Class[] parmsClasses);`

```
public static Motor mkMotor(int pot, int cil, String comb) {
    try {
        Class<?> cm = Class.forName("Motor");
        Constructor<?> cons = cm.getConstructor(int.class, int.class, String.class);
        return (Motor) cons.newInstance(pot, cil, comb);
    }
    catch(Exception e) { out.println(e.getMessage()); return null; }
}
```

Nota: Construtor vazio é invocado usando por parâmetro `new Class[0]`, ou seja, uma array de classes vazio.

OUTROS MÉTODOS CRUCIAIS

- **ESTES MÉTODOS ESTÃO NA BASE DO CÓDIGO APRESENTADO ANTERIORMENTE**

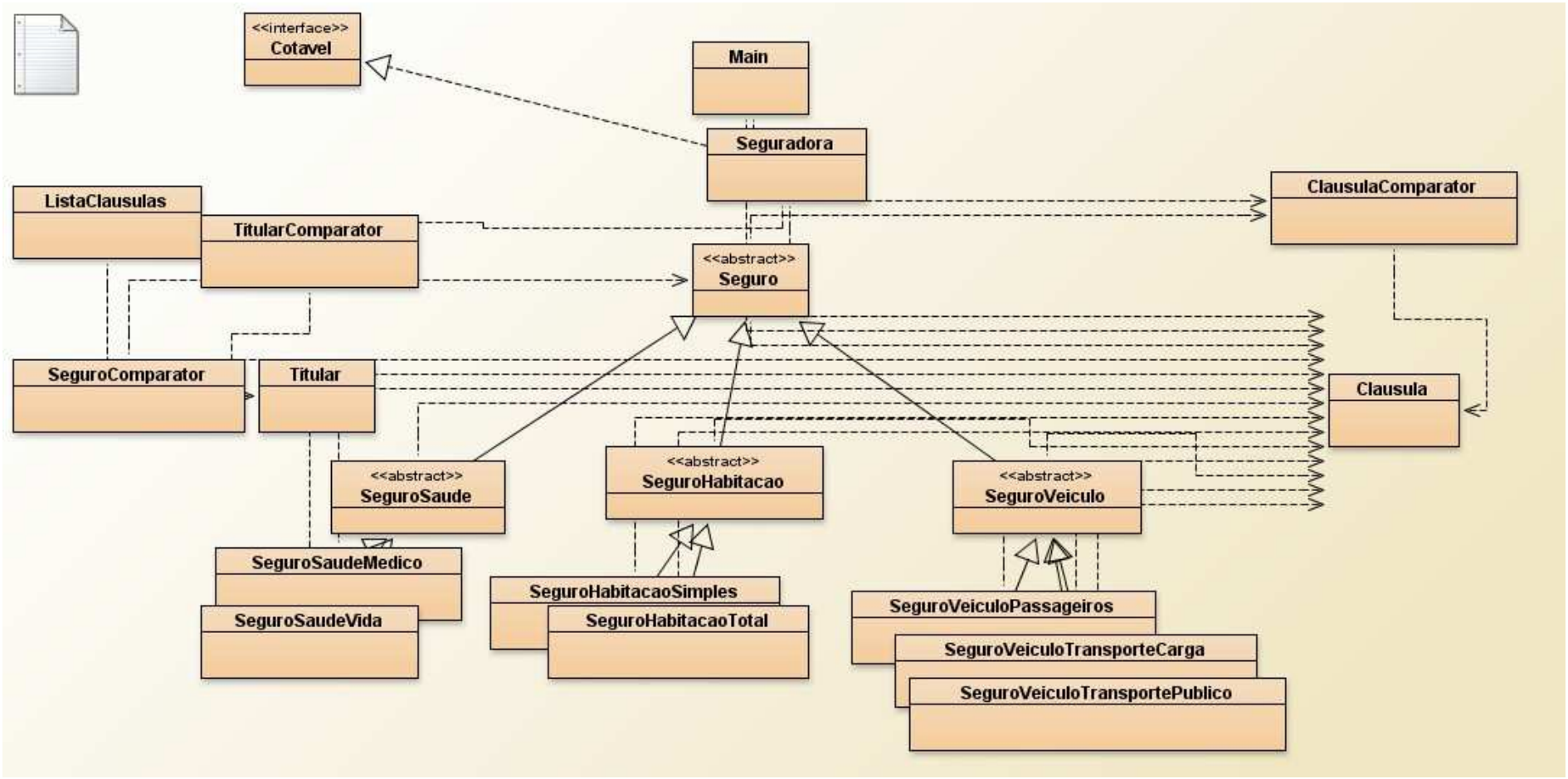
```
private static <T> T mkInst(Class<T> clazz) {
   try {
     return clazz.getConstructor(new Class[0]).
                  newInstance(new Object[0]);
   }
   catch(Exception e) { return null; }
}
```

arrays vazios !!

Método que cria uma instancia T de uma qualquer Classe<T> usando o construtor vazio.

OUTROS MÉTODOS CRUCIAIS

- **Estes métodos e a reflexão permitem coisas muito genéricas e extensíveis.**



OUTROS MÉTODOS CRUCIAIS

- Criação de um seguro de dado tipo usando o mecanismo de reflexão.
  Este código poderia ser ainda melhorado de forma a ser mais genérico.

```
/**
 * Cria um novo seguro de um dado tipo
 */
public Seguro createSeguro(String tipo, String newCodTitular, double newPrecoAnual, double newTaxaOcorrencias,
                           String newDesc, Set<Clausula> newClausulas)
        throws NoSuchMethodException,
               InstantiationException,
               IllegalAccessException,
               IllegalArgumentException,
               InvocationTargetException {
    Constructor<? extends Seguro> cons = tiposSeguros.get(tipo).getConstructor(String.class, double.class, double.class,
                                                                               String.class, Set.class);
    return cons.newInstance(newCodTitular, newPrecoAnual, newTaxaOcorrencias, newDesc, newClausulas);
}
```

ENGENHARIA DE SOFTWARE

ALGUNS PRINCÍPIOS

O QUE É UMA MÁ CONCEPÇÃO DE SOFTWARE ?

DIFÍCIL POIS PODE SER MUITA COISA. MAS SABEMOS ALGUMAS PROPRIEDADES QUE SÃO ABSOLUTAMENTE INDISPENSÁVEIS A UMA BOA CONCEPÇÃO:

1) NÃO SATISFAZER OS REQUISITOS;
2) SER RÍGIDO, IE., UMA PEQUENA ALTERAÇÃO NUMA PARTE IMPLICA GRANDES MUDANÇAS EM MUITAS OUTRAS PARTES;
3) SER FRÁGIL, IE., QUANDO SE FAZ UMA MUDANÇA HÁ PARTES DO SOFTWARE QUE "PARTEM";
4) É IMÓVEL (POUCO REUTILIZÁVEL). É DIFÍCIL DE USAR NOUTRA APLICAÇÃO PORQUE ESTÁ DEMASIADO DEPENDENTE DA APLICAÇÃO ACTUAL.

O QUE É UMA BOA CONCEPÇÃO DE SOFTWARE ?

1) SATISFAZER OS REQUISITOS E CAPTURAR (COMPREENDER) QUAL É A DESIGNADA "HIGH LEVEL POLICY", OU SEJA, QUAIS SÃO AS VERDADES IMUTÁVEIS E A REGRAS FUNDAMENTAIS, QUAISQUER QUE SEJAM OS MECANISMOS ENVOLVIDOS ;

2) SER MODULAR E INDEPENDENTE DO CONTEXTO, OU SEJA, DIVIDOR E SEGMENTAR AS RESPONSABILIDADES E APENAS DEPENDER DE ABSTRACÇÕES E NÃO DE IMPLEMENTAÇÕES REAIS;

3) EM OO O PONTO ANTERIOR SIGNIFICA QUE DEVEMOS PROGRAMAR COM INTERFACES E NÃO DIRECTAMENTE COM CLASSES. AS CLASSES SÃO APENAS IMPLEMENTAÇÕES DE INTERFACES. SE MUDAREM DE IMPLEMENTAÇÃO QUEM DEPENDE DAS INTERFACES NÃO MUDA;

4) O PONTO ANTERIOR CONVIDA TAMBÉM AO PRINCÍPIO DA SEPARAÇÃO DE CAMADAS. NUNCA MISTURAR CÓDIGO DE I/O COM CÓDIGO DA CAMADA COMPUTACIONAL OU BUSINESS (CF. MODELO MVC);

5) SER EXTENSÍVEL, O QUE EM OO IMPLICA HERANÇA E POLIMORFISMO.

EM QUE CONSISTE O PRINCÍPIO DA INVERSÃO DE CONTROLO OU INVERSÃO DE DEPENDÊNCIA ?

Trata-se de um conjunto de pequenas regras de implementação de software modular que visa, de forma simples, eliminar um conjunto de vícios de programação resultantes da designada Programação Estruturada dos anos 80.

Programação Estruturada => refinamento progressivo + dividir para conquistar + módulos de alto nível são refinados em módulos de baixo nível que invocam (cf. subprogramas). Módulos de alto nível são completamente dependentes dos módulos de baixo nível. Apenas os módulos-folha podem ser reutilizados.

Exemplo
Programa que lê caracteres de um teclado e os escreve numa Impressora.

```
void copia() {
   int c;
   while( (c = LeTeclado()) != -1)
        EscImpr(c);
}
```

- Os dois módulos de baixo nível são perfeitamente reutilizáveis.
- O módulo Copia apenas é reutilizável com LeTeclado e EscImpr.
- O módulo Copia é quem tem o **controlo** (ou seja, a lógica).

Qual é a designada "high level policy" ou "high level logic" ?

Ler caracteres de um dispositivo e escrevê-los noutro.

Quem implementa tal lógica ? Módulo Copia.

Quem não é reutilizável ? Módulo Copia.

Que lógica é então reutilizável ? Nenhuma.

```
void copia() {
   int c;
   while( (c = LeTeclado()) != -1)
        EscImpr(c);
}
```

- Os dois módulos de baixo nível são perfeitamente reutilizáveis.
- O módulo Copia apenas é reutilizável com LeTeclado e EscImpr.
- O módulo Copia é quem tem o **controlo** (ou seja, a lógica).

Qual é a designada "high level policy" ou "high level logic" ?

Ler caracteres de um dispositivo e escrevê-los noutro.

Quem implementa tal lógica ? Módulo Copia.

Quem não é reutilizável ? Módulo Copia.

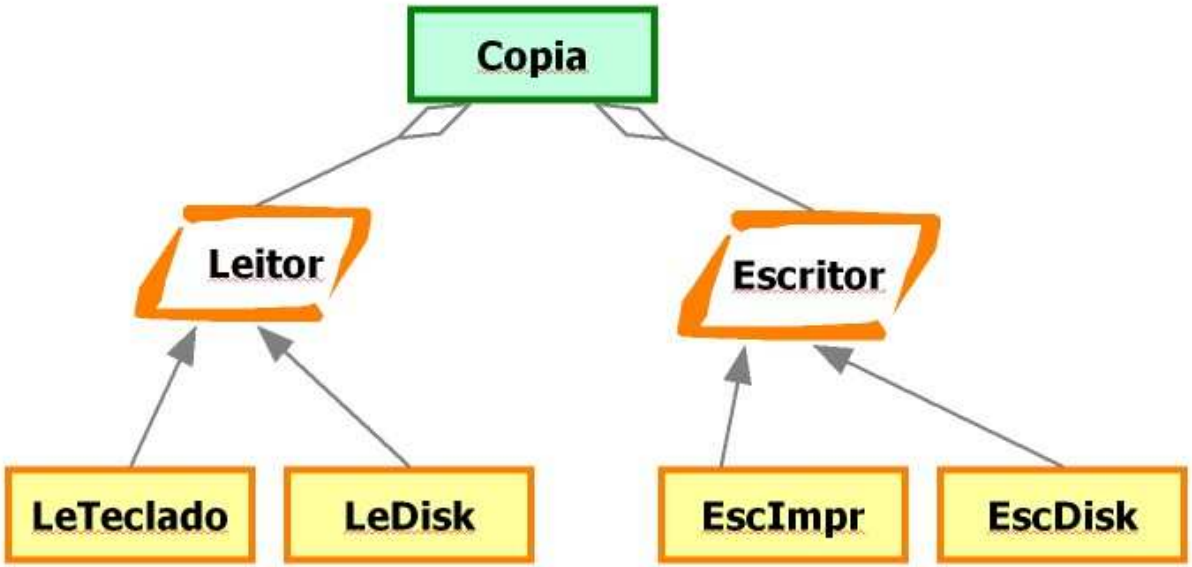
Que lógica é então reutilizável ? Nenhuma.

Princípio da Inversão de Dependência

• Módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender de abstracções (cf. interfaces e classes abstractas em OO);

• Abstracções não devem depender de detalhes/implementações. Implementações devem depender de abstracções.

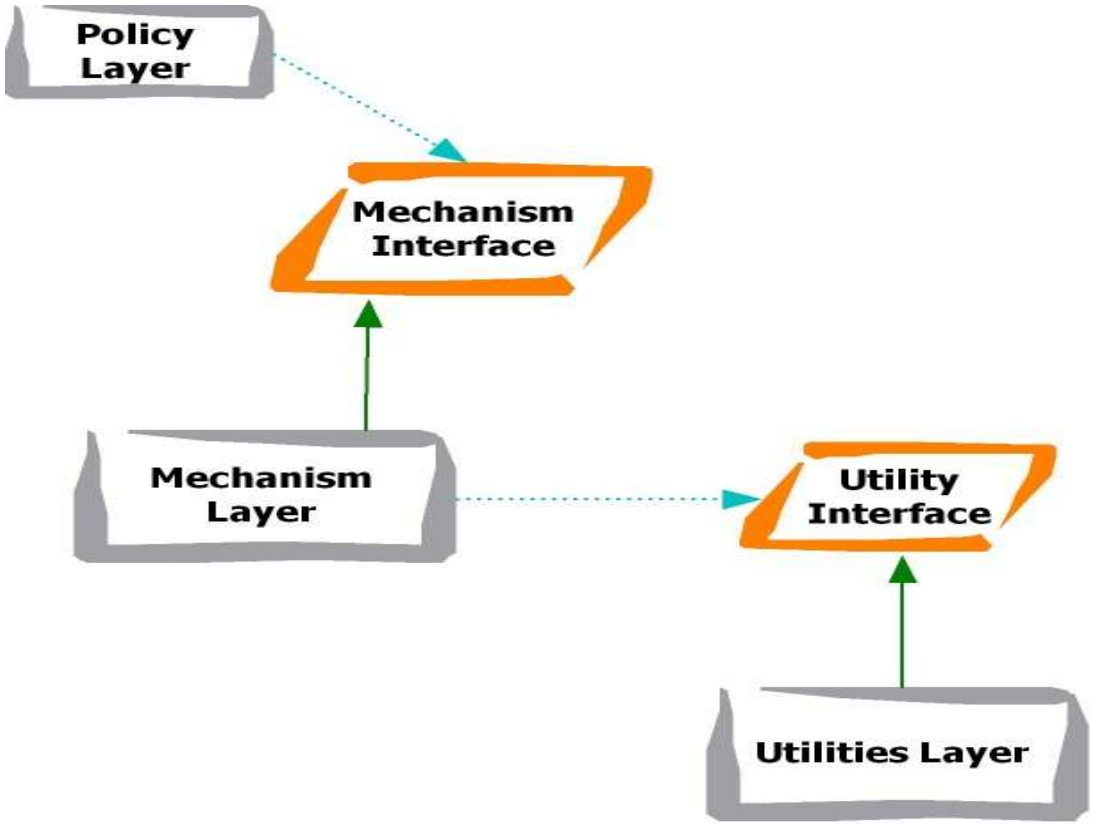
• Em OO a estrutura de dependências passa a estar **invertida** em relação àquela que seria normal numa abordagem "top-down";




```
void copia(Leitor l, Escritor e) {
   int c;
   while( (c = l.read()) != -1)
       e.write(c);
}
```

Modificações nos módulos de baixo nível não têm impacto nos módulos que possuem a lógica das aplicações, e são estes que queremos reutilizar.

▣ Quando estendemos estes princípios às diferentes camadas de SW, passamos a compreender muito melhor o que certos "frameworks" são capazes de nos fornecer.



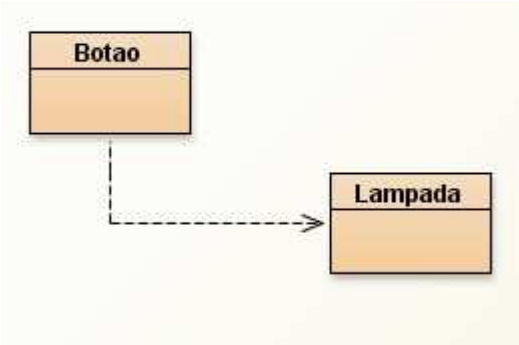
Exemplo básico:

Programar uma classe **Botao** e uma classe **Lampada** de modo a que uma instância de **Botao** seja o controlador de uma instância de **Lampada** e a apague e acenda.

```
public class Botao {

    Lampada lamp = null;

    public Botao() { new Lampada(); }     // botão depende de lampada
    public Botao(Lampada l) { lamp = l; }

    public void click() {
        if(lamp.getEstado().equals("ON")) lamp.turnOFF();
        else lamp.turnON();
    }

}
```

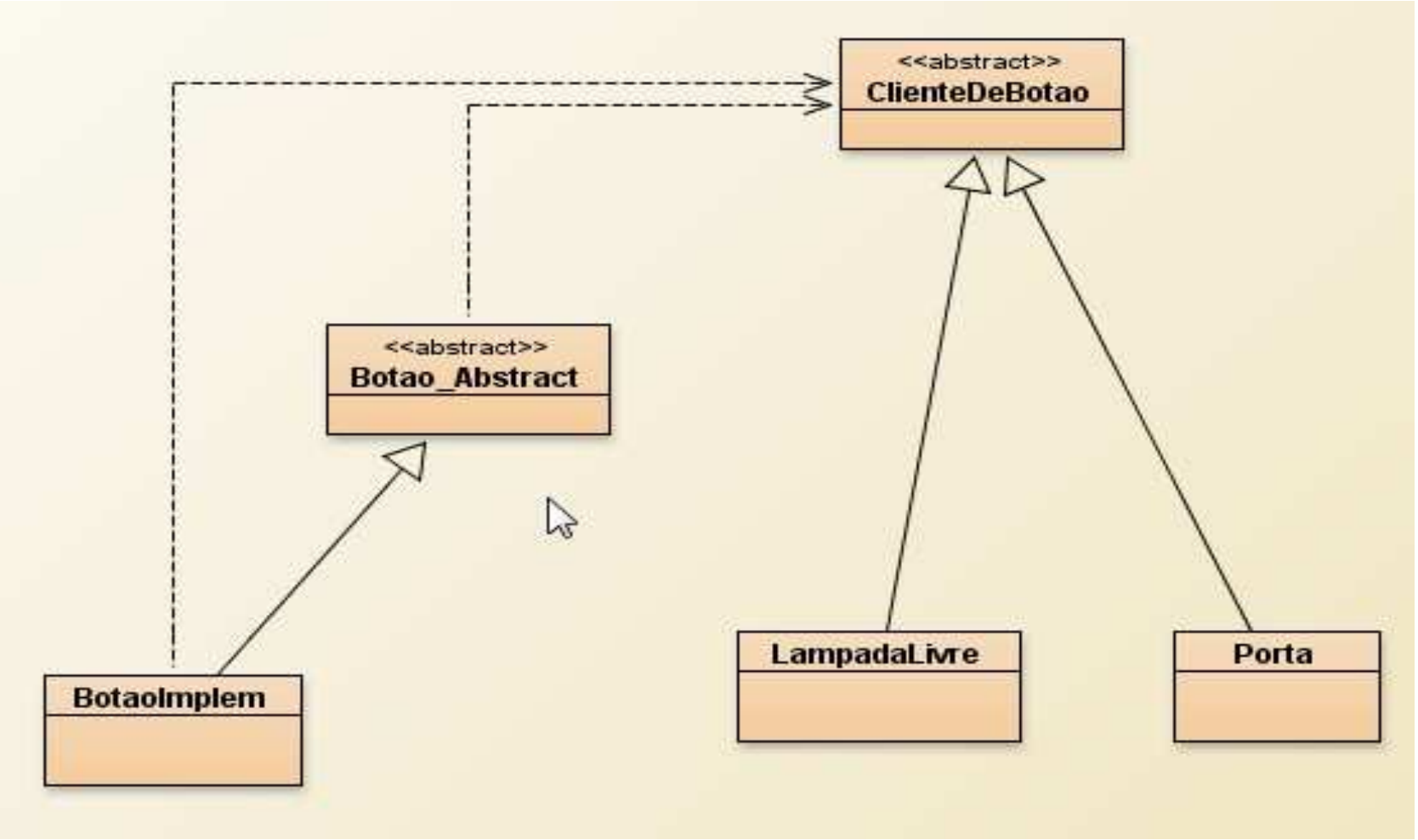
Nota: Esta é uma programação normal mas apenas aceitável ao nível da pequena programação.

As classes **Botao** e **Lampada** são "strongly coupled", ie., a classe Botao depende da implementação de Lampada pois usa-a directamente ao utilizar a construção **new Lampada()** !!




```
public class Lampada {
    String est = "OFF";

    public Lampada() { est = "OFF"; }

    void turnON() { est = "ON"; }
    void turnOFF() { est = "OFF"; }
    String getEstado() { return est; }
}
```

Nota: Vamos seguir as regras do princípio da inversão de dependência


<<abstract>>
ClienteDeBotao
<<abstract>>
Botao_Abstract
BotaoImplem
LampadaLivre
Porta

```
public class LampadaLivre extends ClienteDeBotao {

    String est = "OFF";

    public LampadaLivre() { est = "OFF"; }

    public void turnON() { est = "ON"; }
    public void turnOFF() { est = "OFF"; }
    public String getEstado() { return est; }
}
```

```
public abstract class Botao_Abstract {

    private ClienteDeBotao cliente = null;

    public Botao_Abstract(ClienteDeBotao clb) { cliente = clb; }

    public abstract void changeEstado();
    public abstract boolean getEstado();

    public void click() {
        if(this.getEstado()) cliente.turnOFF(); else cliente.turnON();
        this.changeEstado();
    }

}
```

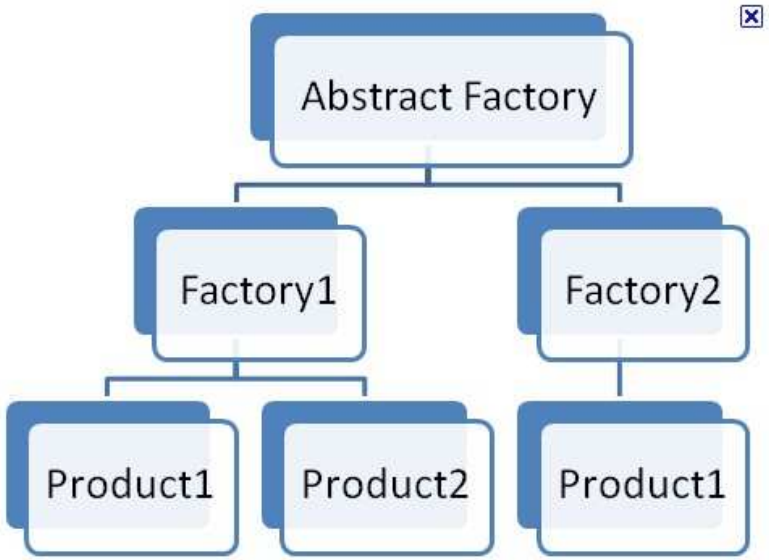
```
public class BotaoImplem extends Botao_Abstract {

    private boolean estado = false;

    public BotaoImplem(ClienteDeBotao clb) { super(clb); }

    public boolean getEstado() { return estado; }

    public void changeEstado() { estado = estado ? false : true; }

}
```

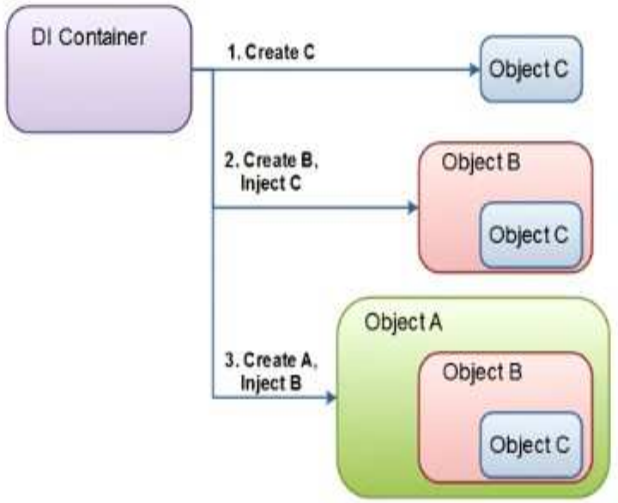
```
public abstract class ClienteDeBotao {

    public abstract void turnOFF();
    public abstract void turnON() ;

}
```

Factories, Beans e Containers

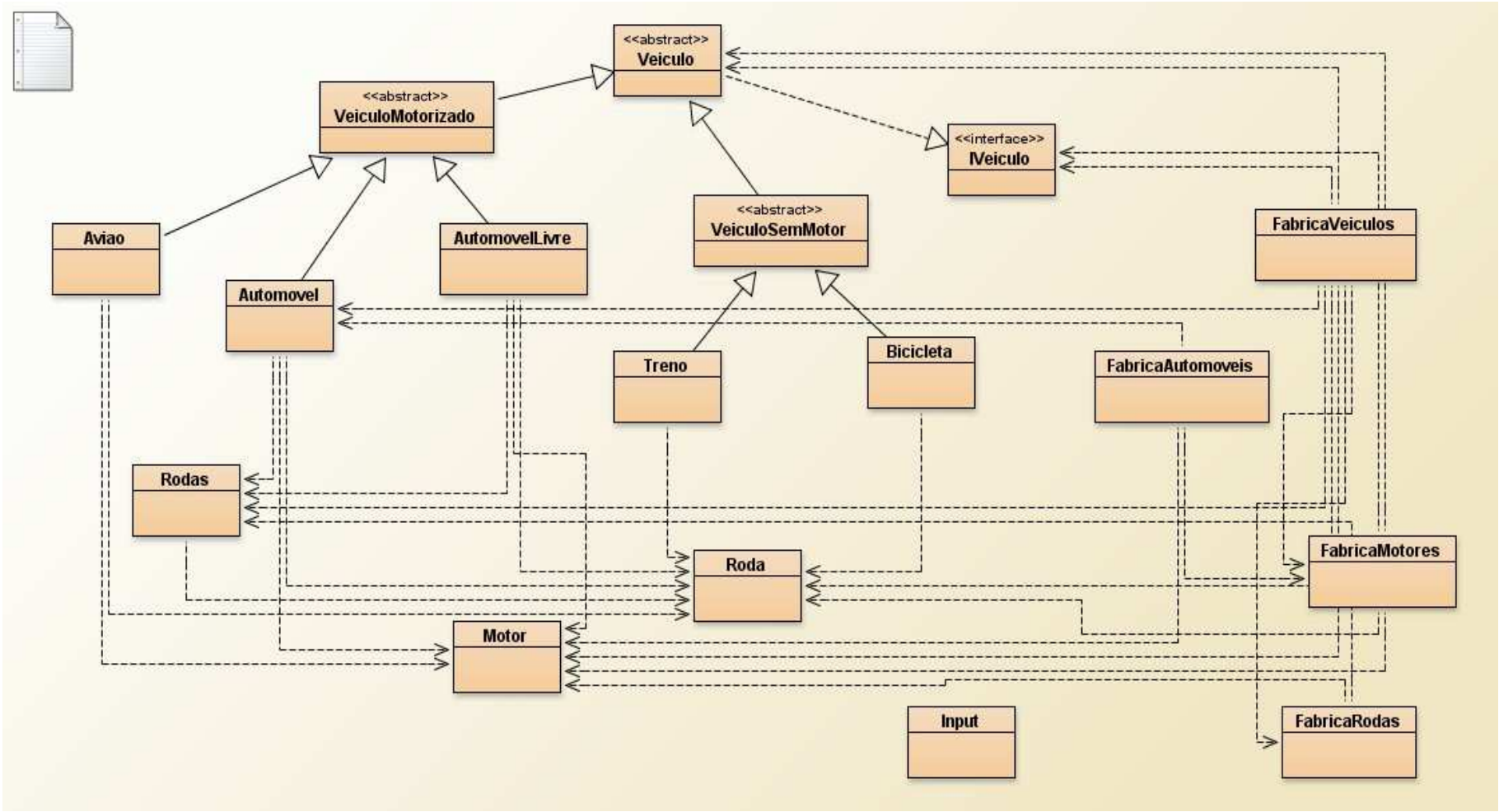
Construída correctamente a arquitectura, precisamos agora de criar mecanismos expeditos de configuração e de injecção de instâncias.

1) **Factories** programadas por nós (usando reflexão, etc.);
2) **Beans** e **Containers** cf. Spring





Factories, Beans e Containers



Algumas classes e a interface IVeiculo

```
public class AutomovelLivre extends VeiculoMotorizado {

    private String marca;
    private Motor motor;
    private Rodas rodas;

    public AutomovelLivre() {
      super();
      marca = "??"; motor = null; rodas = null;
    }

    public AutomovelLivre(String mk, Motor mt, Rodas lrodas) {
        super();
        marca = mk; motor = mt; rodas = lrodas;
    }

    public void setRodas(Rodas rds) { rodas = rds; }
    public void setMotor(Motor m) { motor = m; }

    public void start() { motor.start(); }
    public void stop() { motor.stop(); }
    public void move(int vel) { super.setVeloc(vel); }
```

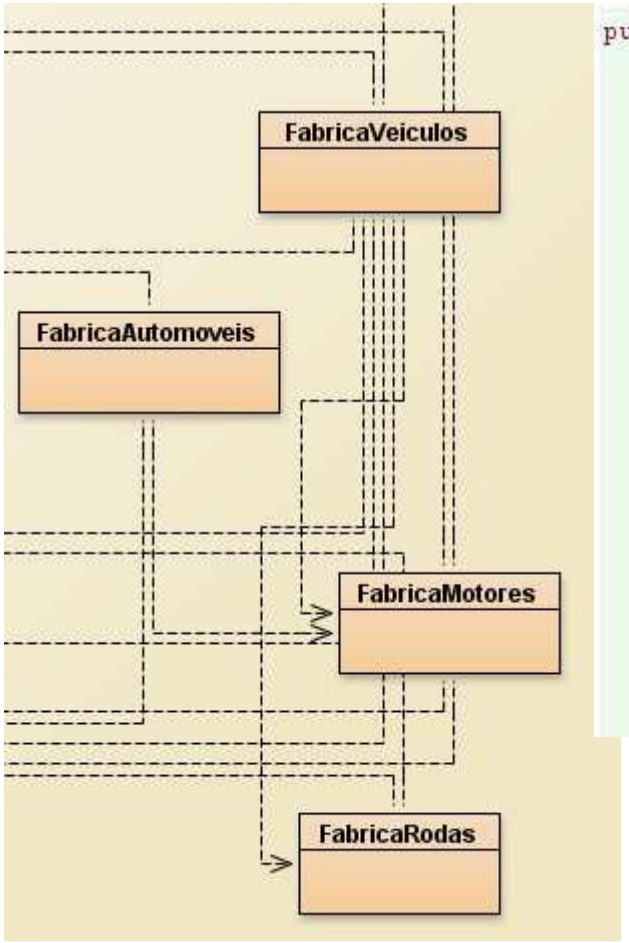
Factories, Beans e Containers

```
public interface IVeiculo {

    void start();
    void stop();
    void move(int veloc);
}
```

```
public abstract class Veiculo implements IVeiculo {

    private int velocidade;

    public Veiculo() { velocidade = 0; }
    public Veiculo(int vel) { velocidade = vel; }

    public int getVeloc() { return velocidade; }
    public void setVeloc(int vel) { velocidade = vel; }

    public abstract void start();
    public abstract void stop();
    public abstract void move(int veloc);
    public abstract String toString();
}
```

Factories, Beans e Containers

As fábricas !




```
public class FabricaRodas {

    public static Rodas mkRodas() {
        try {
            Class<?> cm = Class.forName("Rodas");
            return (Rodas) cm.newInstance();
        }
        catch(Exception e) { out.println(e.getMessage()); return null; }
    }

    public static Rodas mkRodas(int numr) {
        try {
            Class<?> cm = Class.forName("Rodas");
            Constructor<?> cons = cm.getConstructor(int.class);
            return (Rodas) cons.newInstance(numr);
        }
        catch(Exception e) { out.println(e.getMessage()); return null; }
    }
```

Factories, Beans e Containers

```
public class FabricaMotores {

    public static <T> T mkInstance(Class<T> clazz) {
        try {
            return clazz.newInstance();
        }
        catch(Exception e) { out.println(e.getMessage()); return null; }
    }

    public static Motor mkMotor() {
        try {
            Class<?> cm = Class.forName("Motor");
            return (Motor) cm.newInstance();
        }
        catch(Exception e) { out.println(e.getMessage()); return null; }
    }

    public static Motor mkMotor(int pot, int cil, String comb) {
        try {
            Class<?> cm = Class.forName("Motor");
            Constructor<?> cons = cm.getConstructor(int.class, int.class, String.class);
            return (Motor) cons.newInstance(pot, cil, comb);
        }
        catch(Exception e) { out.println(e.getMessage()); return null; }
    }
```

Factories, Beans e Containers

```
public static Veiculo mkVeiculo(String nmv, Motor m, Rodas rds) {
    try {
        Class<?> cv = Class.forName(nmv);
        Constructor<?> cons = cv.getConstructor(String.class, Motor.class, Rodas.class);
        return (Veiculo) cons.newInstance(nmv, m, rds);
    }
    catch(Exception e) { System.out.println(e.getMessage()); return null; }
}
```

```
public static Veiculo mkVeiculo(String nmv, Motor m, int rds) {
    try {
        Class<?> cv = Class.forName(nmv);
        Constructor<?> cons = cv.getConstructor(String.class, Motor.class, int.class);
        return (Veiculo) cons.newInstance(nmv, m, rds);
    }
    catch(Exception e) { System.out.println(e.getMessage()); return null; }
}
```

```
public static IVeiculo mkIVeiculo(String nmv) {
    try {
        Class<?> cv = Class.forName(nmv);
        return (IVeiculo) cv.newInstance();
    }
    catch(Exception e) { return null; }
}
```

Factories, Beans e Containers

```
public static void main() {
    out.println(mkVeiculo("Automovel").toString());
    out.println(mkVeiculo("Treno").toString());
    out.println(mkVeiculo("Bicicleta").toString());

    out.println(mkVeiculo("Bicicleta", 4).toString());
    out.println(mkVeiculo("Automovel", 6).toString());


    out.println(mkVeiculo("Automovel", FabricaMotores.mkMotor(110, 1600, "TDI"), 4).toString());

    out.println(mkVeiculo("Aviao", "Airbus A740", 4).toString());

    out.println(mkVeiculo("AutomovelLivre", FabricaMotores.mkMotor(120, 1900, "sem chumbo 98"),
                                            FabricaRodas.mkRodas(4)).toString());

    //-------------------------------------------------------------------------------------
```

Factories, Beans e Containers

▣ **No código anterior usamos principalmente injecção de instância via construtores, mas também poderá ser feita por métodos set() depois de usado o construtor vazio e, em ambos, usando interfaces.**

> Implementation techniques are influenced by the computer language used.
>
> In Java there are six basic techniques to implement Inversion of Control. These are:
>
> 1. using a factory pattern
> 2. using a service locator pattern
> 3. using a constructor injection
> 4. using a setter injection
> 5. using an interface injection
> 6. using a contextualized lookup
>
> Constructor, setter, and interface injection are all aspects of Dependency injection.

Factories, Beans e Containers

- **Mas como poderemos tornar todo este processo de criação de sw mais automatizado, seguro e configurável ?**

Usando frameworks que implementam e suportam os princípios da inversão de controlo (IoC) e injecção de dependência (DI) através da implementação dos conceitos de bean (objecto) e container (injector, etc.).

- NOTA: Para quem, como nós, sabe como e o que custa fazer tudo isto "à mão", tal como vimos atrás, estes frameworks são "o paraíso" ou, em alguns casos "a primavera".

- **Alguns "frameworks": Castle, Avalon, Spring, ...**

SPRING: Beans e Containers

Spring Framework

spring source

Chapter 3. The IoC container

3.1. Introduction

This chapter covers the Spring Framework's implementation of the Inversion of Control (IoC) [1] principle.

The `org.springframework.beans` and `org.springframework.context` packages provide the basis for the Spring Framework's IoC container. The BeanFactory interface provides an advanced configuration mechanism capable of managing objects of any nature. The ApplicationContext interface builds on top of the `BeanFactory` (it is a sub-interface) and adds other functionality such as easier integration with Spring's AOP features, message resource handling (for use in internationalization), event propagation, and application-layer specific contexts such as the `WebApplicationContext` for use in web applications.

In short, the `BeanFactory` provides the configuration framework and basic functionality, while the `ApplicationContext` adds more enterprise-centric functionality to it. The `ApplicationContext` is a complete superset of the `BeanFactory`, and any description of `BeanFactory` capabilities and behavior is to be considered to apply to the `ApplicationContext` as well.

This chapter is divided into two parts, with the first part covering the basic principles that apply to both the `BeanFactory` and `ApplicationContext`, and with the second part covering those features that apply only to the `ApplicationContext` interface.

`BeanFactory` or `ApplicationContext`?

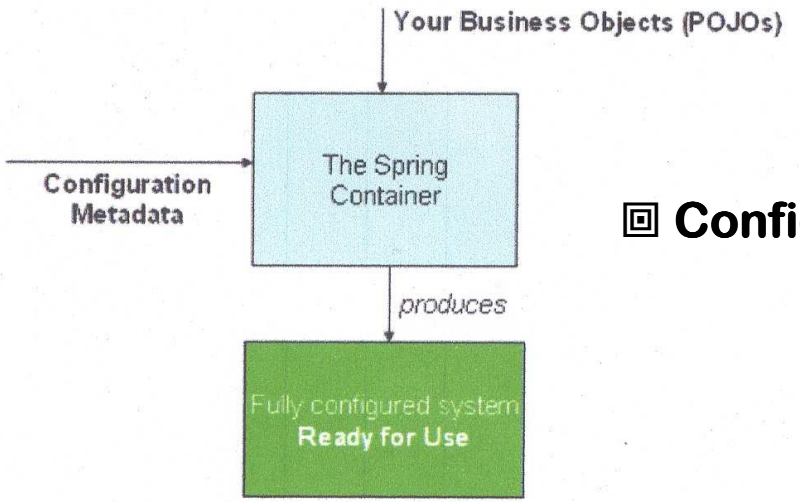
Users are sometimes unsure whether a `BeanFactory` or an `ApplicationContext` is best suited for use in a particular situation. A `BeanFactory` pretty much just instantiates and configures beans. An `ApplicationContext` also does that, *and* it provides the supporting infrastructure to enable *lots* of enterprise-specific features such as transactions and AOP.

In short, favor the use of an `ApplicationContext`.

(For the specific details behind this recommendation, see this section.)

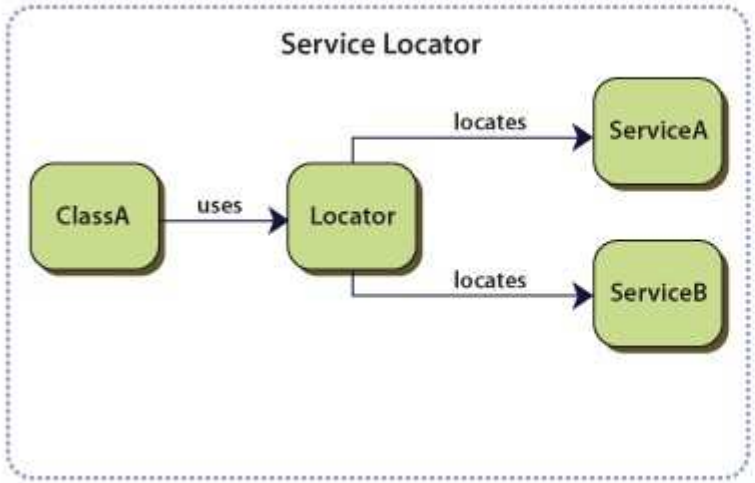
Interface fundamental : `ApplicationContext`

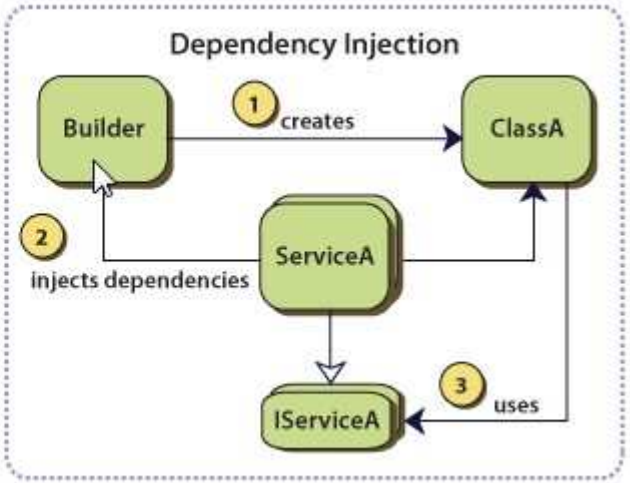
Factories, Beans e Containers



The Spring IoC container

- **Configuration Metadata = XML ou Annotations**





Configuração usando metadados em XML

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

  <bean id="..." class="...">
    <!-- collaborators and configuration for this bean go here -->
  </bean>

  <bean id="..." class="...">
    <!-- collaborators and configuration for this bean go here -->
  </bean>

  <!-- more bean definitions go here -->

</beans>
```

```
ApplicationContext context = new ClassPathXmlApplicationContext(
        new String[] {"services.xml", "daos.xml"});

// an ApplicationContext is also a BeanFactory (via inheritance)
BeanFactory factory = context;
```

Factories, Beans e Containers

- Agora teremos que saber escrever em XML todas as formas de instanciação de beans tal como anteriormente fizemos escrevendo código explícito.

Table 3.1. The bean definition

Feature	Explained in...
class	Section 3.2.3.2, "Instantiating beans"
name	Section 3.2.3.1, "Naming beans"
scope	Section 3.4, "Bean scopes"
constructor arguments	Section 3.3.1, "Injecting dependencies"
properties	Section 3.3.1, "Injecting dependencies"
autowiring mode	Section 3.3.5, "Autowiring collaborators"
dependency checking mode	Section 3.3.6, "Checking for dependencies"
lazy-initialization mode	Section 3.3.4, "Lazily-instantiated beans"
initialization method	Section 3.5.1.1, "Initialization callbacks"
destruction method	Section 3.5.1.2, "Destruction callbacks"

Factories, Beans e Containers

```
<bean id="exampleBean"
      class="examples.ExampleBean2"
      factory-method="createInstance"/>
```

Usando factory method.

```
<!-- the factory bean, which contains a method called createInstance() -->
<bean id="serviceLocator" class="com.foo.DefaultServiceLocator">
  <!-- inject any dependencies required by this locator bean -->
</bean>

<!-- the bean to be created via the factory bean -->
<bean id="exampleBean"
      factory-bean="serviceLocator"
      factory-method="createInstance"/>
```

Usando factory bean e método de instância

Factories, Beans e Containers

3.3.1.1.1. Constructor Argument Type Matching

The above scenario *can* use type matching with simple types by explicitly specifying the type of the cons

```
<bean id="exampleBean" class="examples.ExampleBean">
  <constructor-arg type="int" value="7500000"/>
  <constructor-arg type="java.lang.String" value="42"/>
</bean>
```

3.3.1.1.2. Constructor Argument Index

Constructor arguments can have their index specified explicitly by use of the `index` attribute. For exam

```
<bean id="exampleBean" class="examples.ExampleBean">
  <constructor-arg index="0" value="7500000"/>
  <constructor-arg index="1" value="42"/>
</bean>
```

Invocando os construtores usando a respectiva assinatura e dando os valores dos parâmetros (recordar método getConstructor()).

Factories, Beans e Containers

```
<bean id="exampleBean" class="examples.ExampleBean">

  <!-- setter injection using the nested <ref/> element -->
  <property name="beanOne"><ref bean="anotherExampleBean"/></property>

  <!-- setter injection using the neater 'ref' attribute -->
  <property name="beanTwo" ref="yetAnotherBean"/>
  <property name="integerProperty" value="1"/>
</bean>

<bean id="anotherExampleBean" class="examples.AnotherBean"/>
<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>
```

```
public class ExampleBean {

    private AnotherBean beanOne;
    private YetAnotherBean beanTwo;
    private int i;

    public void setBeanOne(AnotherBean beanOne) {
        this.beanOne = beanOne;
    }

    public void setBeanTwo(YetAnotherBean beanTwo) {
        this.beanTwo = beanTwo;
    }

    public void setIntegerProperty(int i) {
        this.i = i;
    }
```

Setter IOC

Factories, Beans e Containers

```
<bean id="exampleBean" class="examples.ExampleBean">

  <!-- constructor injection using the nested <ref/> element -->
  <constructor-arg>
    <ref bean="anotherExampleBean"/>
  </constructor-arg>

  <!-- constructor injection using the neater 'ref' attribute -->
  <constructor-arg ref="yetAnotherBean"/>

  <constructor-arg type="int" value="1"/>
</bean>

<bean id="anotherExampleBean" class="examples.AnotherBean"/>
<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>
```

```
public class ExampleBean {

    private AnotherBean beanOne;
    private YetAnotherBean beanTwo;
    private int i;

    public ExampleBean(
        AnotherBean anotherBean, YetAnotherBean yetAnotherBean, int i) {
        this.beanOne = anotherBean;
        this.beanTwo = yetAnotherBean;
        this.i = i;
    }
}
```

Constructor IOC

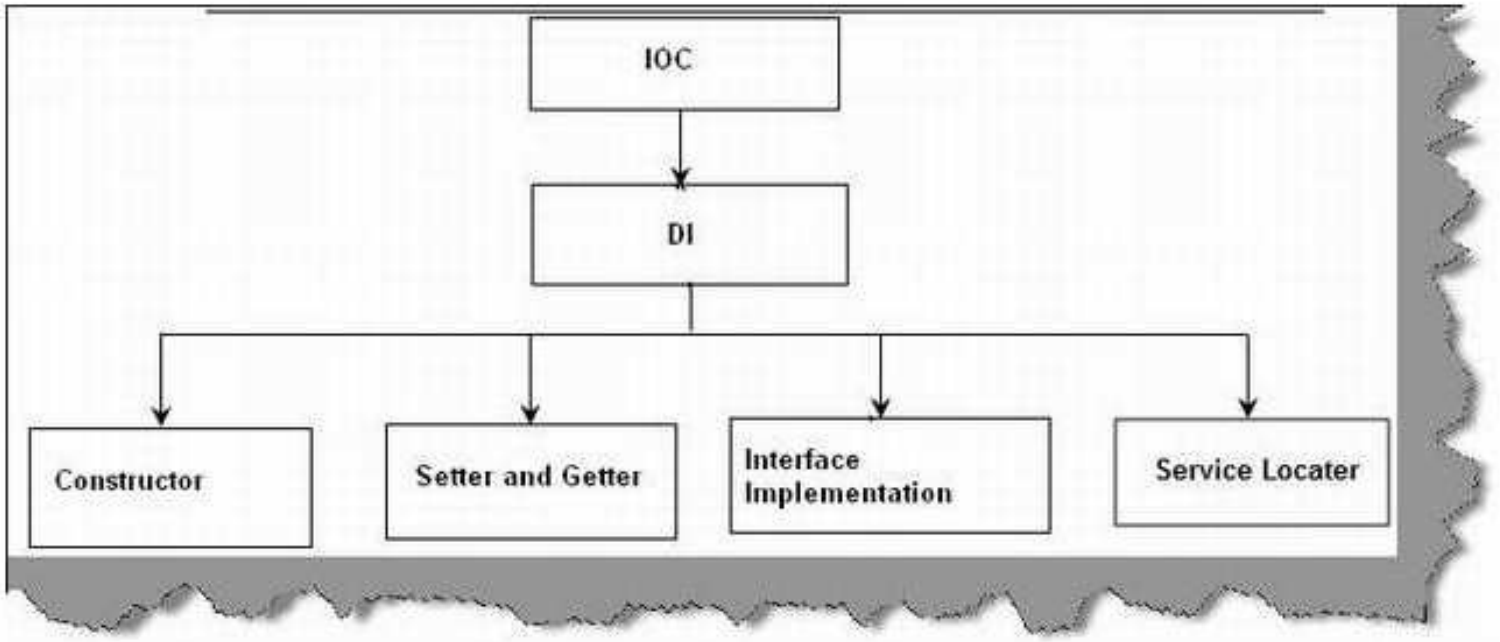


Figure: - IOC and DI

TIPOS FUNDAMENTAIS DE DI

Factories, Beans e Containers

Agora teríamos as bases adequadas para um profundo estudo e um uso mais adequado doSpring framework ou de outros semelhantes baseados em princípios como IoC e DI.