Padrões de Projeto

Unidade 1

Edison Silva e
Valter Camargo

ITI
MBA

INFORMAÇÃO
TECNOLOGIA
& INOVAÇÃO

INTRODUCTION

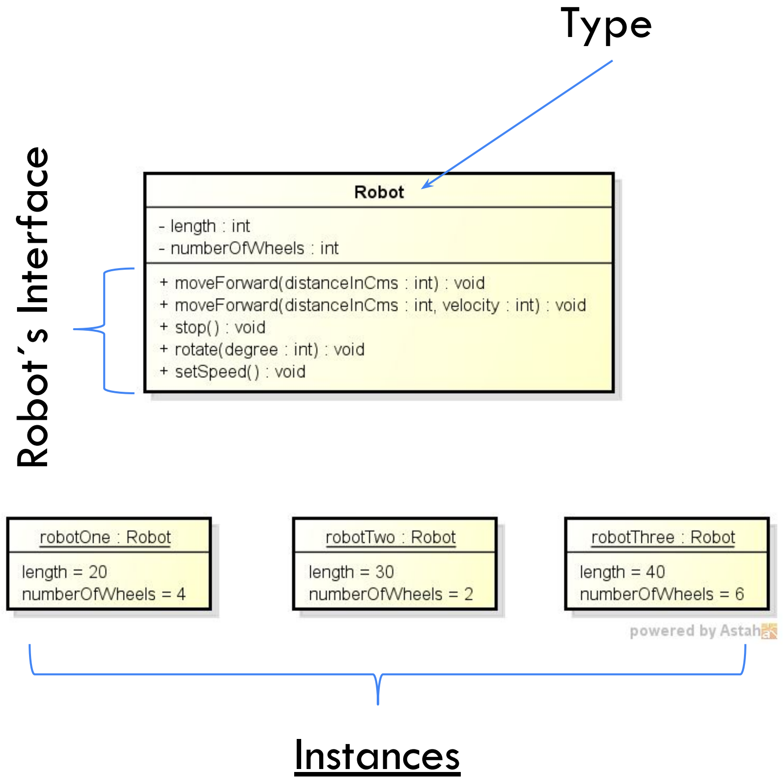
- What is a design pattern ?
- How to decompose a system into objects ?
  - Nouns ?
  - Collaborations among objects ?
  - Responsibilities ?
  - Real world ?
- Modelling a system as the real world may represent the present, but not the future ☹️
- The **abstractions** used during the project are the key for flexible designs

INTRODUCTION

- Design Patterns assist in finding out not so obvious abstractions
  - States,
  - Strategies,
  - Observers,
  - Composites,
  - Etc..
- Would you create a class with these names in your project ?

INTERFACES

- Interfaces
  - Interfaces are the set of signatures an object exposes
  - The interface of an object states all the requisitions can be sent to it
  - A Type is a name used to denote a specific interface
  - Objects can have several types (hierarchy)
  - Interfaces do not say anything about the object’s implementation (i.e., the o methods)



Type

Robot´s Interface

Robot

- length : int
- numberOfWheels : int

+ moveForward(distanceInCms : int) : void
+ moveForward(distanceInCms : int, velocity : int) : void
+ stop() : void
+ rotate(degree : int) : void
+ setSpeed() : void

robotOne : Robot

length = 20
numberOfWheels = 4

robotTwo : Robot

length = 30
numberOfWheels = 2

robotThree : Robot

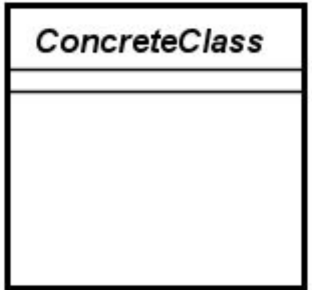
length = 40
numberOfWheels = 6

powered by Astah

Instances

STATIC OR DYNAMIC BINDING ?
POLYMORPHISM ?

- Static Binding / Dynamic
  - Static: The link between a **request** to an object **to one of its operations** is in design/compile time
  - Dynamic: The link between a **request** to an object **to one of its operations** is in run-time



```
Client c = new Client
...
c.myMethod()
```

Client não é abstrata/interface
Os serviços oferecidos são "linkados" ao tipo concreto antecipadamente… em tempo de compilação

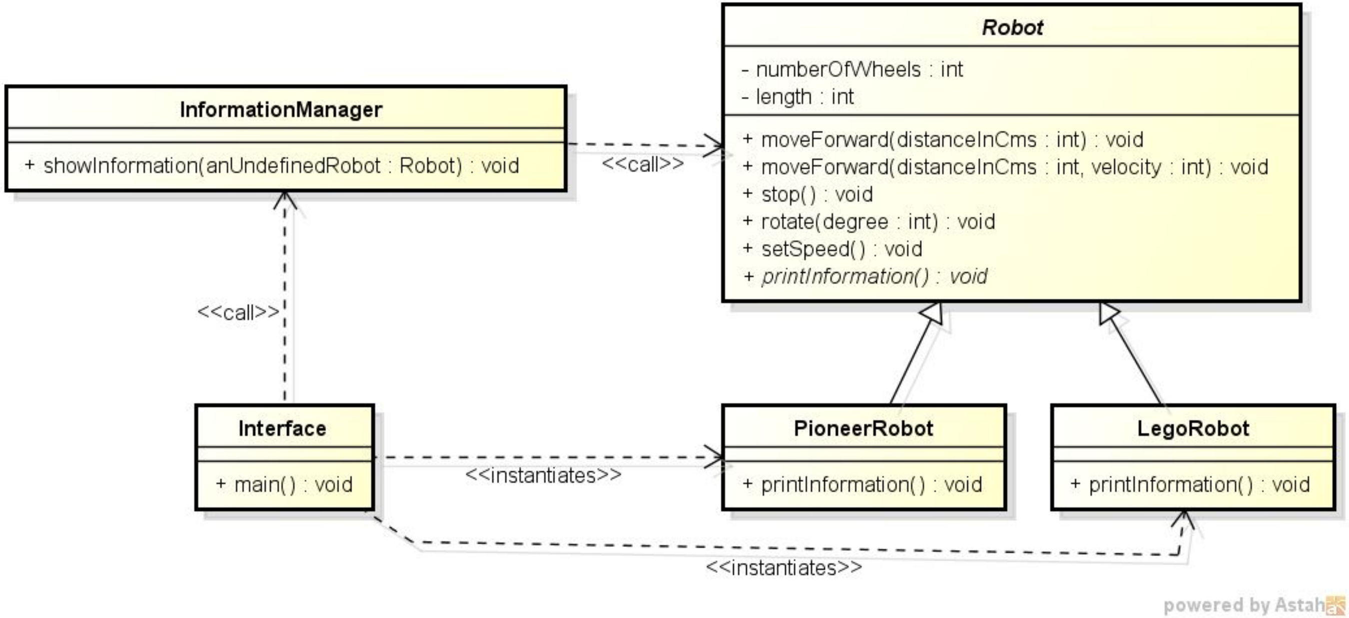
AbstractClass

```
Abstract Client c;
...
c.myMethod()
```

Client é abstrata
Os serviços oferecidos podem ser demandados mesmo sem saber quem
irá efetivamente executá-los.

STATIC OR DYNAMIC BINDING ? POLYMORPHISM ?

- Let´s understand this example :



InformationManager

+ showInformation(anUndefinedRobot : Robot) : void

Robot

- numberOfWheels : int
- length : int

+ moveForward(distanceInCms : int) : void
+ moveForward(distanceInCms : int, velocity : int) : void
+ stop() : void
+ rotate(degree : int) : void
+ setSpeed() : void
+ *printInformation() : void*

Interface

+ main() : void

PioneerRobot

+ printInformation() : void

LegoRobot

+ printInformation() : void

STATIC OR DYNAMIC BINDING ? POLYMORPHISM ?

- What is “Dynamic binding” ?
  - The link between a **request** to an object **to one of its operations** in runtime (thanks to polymorphism)

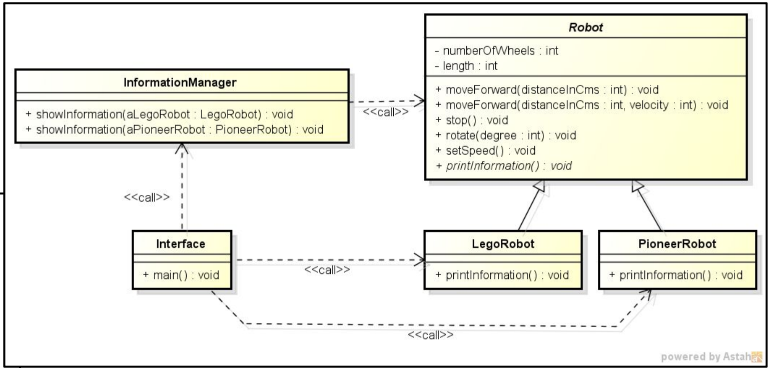
```
public static void main(String args[]){

    Robot robotOne;

    if (condition) {
        robotOne = new LegoRobot();
    } else {
        robotOne = new PioneerRobot();
    }

    InformationManager.showInformation(robotOne);
}
```

Important !

Aqui está a mágica

```
public class PioneerRobot extends Robot {

@Override
public void printInformation() {
      System.out.println(
              "This is a pionner robot"
       );
}
```

DYNAMIC BINDING...


Robot
- numberOfWheels : int
- length : int
+ moveForward(distanceInCms : int) : void
+ moveForward(distanceInCms : int, velocity : int) : void
+ stop() : void
+ rotate(degree : int) : void
+ setSpeed() : void
+ printInformation() : void
InformationManager
+ showInformation(aLegoRobot : LegoRobot) : void
+ showInformation(aPioneerRobot : PioneerRobot) : void
<<call>>
<<call>>
Interface
+ main() : void
<<call>>
LegoRobot
+ printInformation() : void
PioneerRobot
+ printInformation() : void
<<call>>
powered by Astah

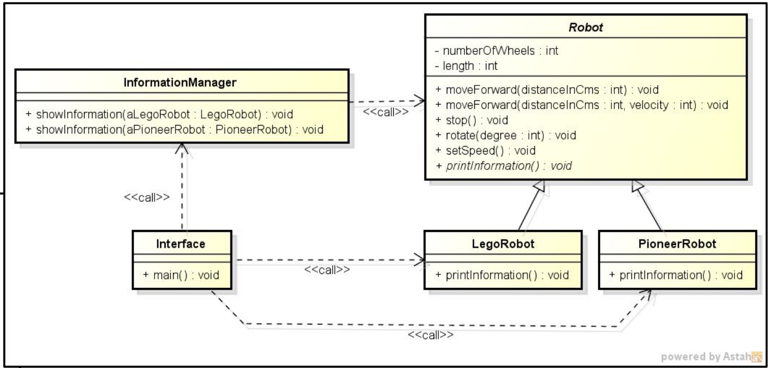
```
public class InformationManager {

    public static void showInformation(Robot robot){

        robot.printInformation();

    }

}
```

```
public class PioneerRobot extends Robot   {
   public void printInformation() {
          System.out.println("This is a pionner robot");
          // this could be much more complex
    }
}
```

```
public class LegoRobot extends Robot {
    public void printInformation() {
          System.out.println("This is a lego robot !");
          // this could be much more complex
    }
}
```

```
public static void main(String args[]){


Robot robotOne;

    if (condition) {
        robotOne = new LegoRobot();
    } else {
        robotOne = new PioneerRobot();
    }

    InformationManager.showInformation(robotOne);
```

STATIC BINDING...



Robot

- numberOfWheels : int
- length : int

+ moveForward(distanceInCms : int) : void
+ moveForward(distanceInCms : int, velocity : int) : void
+ stop() : void
+ rotate(degree : int) : void
+ setSpeed() : void
+ *printInformation() : void*

InformationManager

+ showInformation(aLegoRobot : LegoRobot) : void
+ showInformation(aPioneerRobot : PioneerRobot) : void

Interface

+ main() : void

LegoRobot

+ printInformation() : void

PioneerRobot

+ printInformation() : void

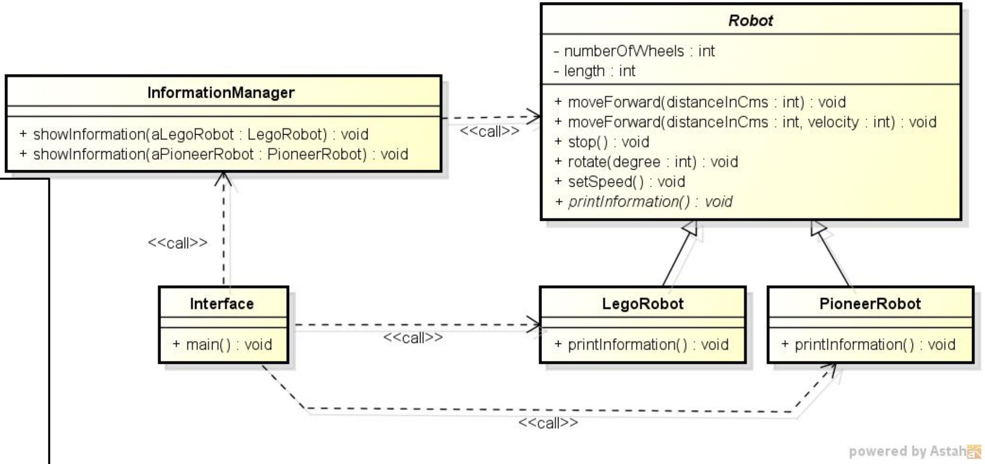
```
public class InformationManager {

    public static void showInformation(LegoRobot aLegoRobot){
        aLegoRobot.printInformation();
    }

    public static void showInformation(PioneerRobot aPioneerRobot){
        aPioneerRobot.printInformation();
     }
}
```

```
public class PioneerRobot extends Robot  {
   public void printInformation() {
         System.out.println("This is a pionner robot");
     }
}
```

```
public class LegoRobot extends Robot {
    public void printInformation() {
         System.out.println("This is a lego robot !");
     }
}
```

```
public static void main(String args[]){

PioneerRobot pioneerRobot = new PioneerRobot();
LegoRobot aLegoRobot = new LegoRobot();

String type = "null";

if (type == "pioneer"){
   ShowInformation.showInformation(pioneerRobot);
} else {
   ShowInformation.showInformation(aLegoRobot);
}
}
```

STATIC BINDING...



Robot

- numberOfWheels : int
- length : int

+ moveForward(distanceInCms : int) : void
+ moveForward(distanceInCms : int, velocity : int) : void
+ stop() : void
+ rotate(degree : int) : void
+ setSpeed() : void
+ *printInformation() : void*

InformationManager

+ showInformation(aLegoRobot : LegoRobot) : void
+ showInformation(aPioneerRobot : PioneerRobot) : void

Interface

+ main() : void

LegoRobot

+ printInformation() : void

PioneerRobot

+ printInformation() : void

```
class InformationManager:
    def showInformation(aLegoRobot: LegoRobot):
        aLegoRobot.printInformation()

    def showInformation(aPioneerRobot: PioneerRobot):
        aPioneerRobot.printInformation()
}
```

Obs. Esta forma de implementação não é possívle em Python, é apenas um exemplo

```
class Robot:
    def printInformation(__self__):
        raise NotImplementedError

class PioneerRobot(Robot):
    def printInformation(__self__):
        print("This is a pionner robot")

class LegoRobot(Robot):
    def printInformation(__self__):
        print("This is a lego robot")
```

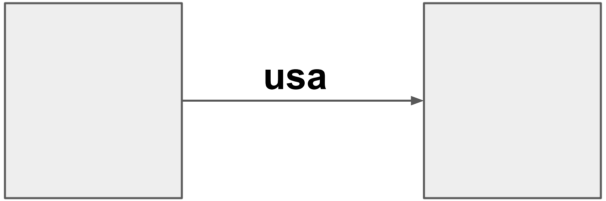
```
if __name__ == "__main__":
    if condition:
        robotOne = LegoRobot()
    else:
        robotOne = PioneerRobot()

    InformationManager.showInformation(robotOne)
```

YOU NEED TO PROGRAM FOR INTERFACES...

- **You need to program for interfaces, not for particular implementations**
  - Using inheritance you can define *family of objects* sharing identical interfaces
  - All subclasses are able to reply to the requests forwarded to the interface of the abstract class
  - The main benefit of manipulating objects only considering their interfaces is that clients keep **unaware** of the type of the objects they use

Client-Code

Do you remember what “client code” means ?

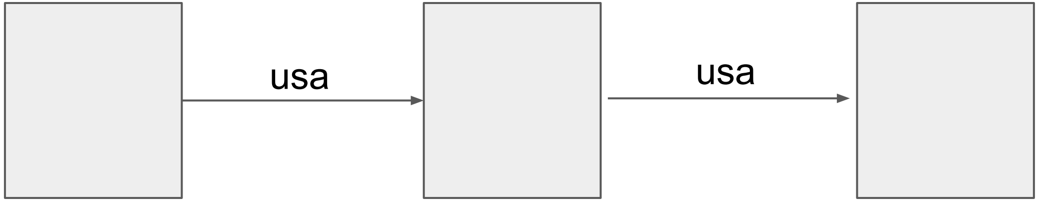


usa

Cliente

Fornecedor

*Código cliente é aquele que **usa** (cria objetos e chama métodos desses objetos). Muitas vezes é importante fazer com que esses códigos sejam independentes das classes que eles usam. Mas, como identificá-los ?*



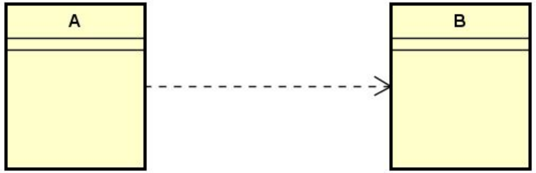
usa

usa

Cliente

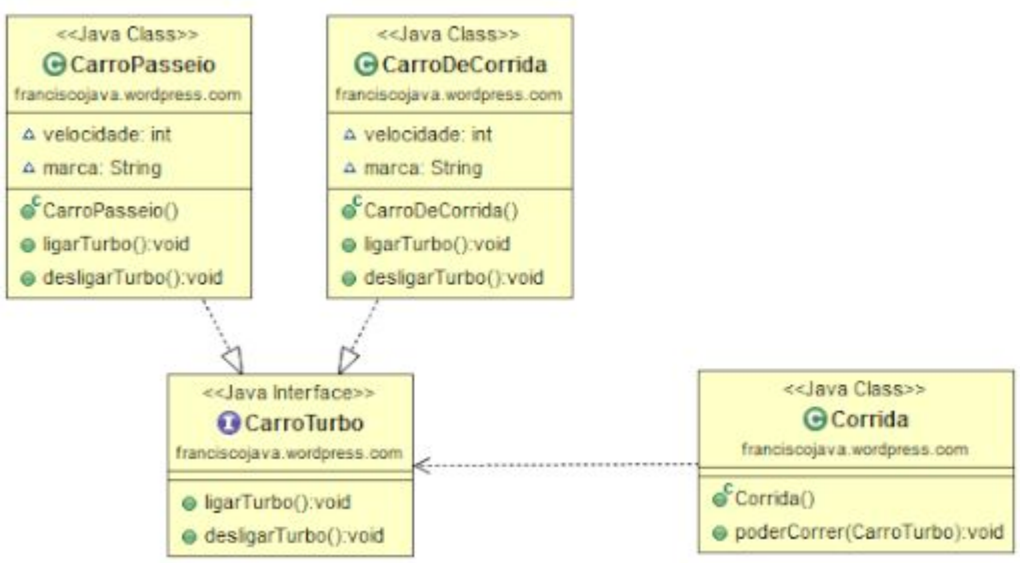
Cliente…. Mas Fornecedor também….

Pense em código cliente como aquele trecho de código que “consome” ou “requisita serviços” de alguma entidade….



Do you remember what “client code” means ?

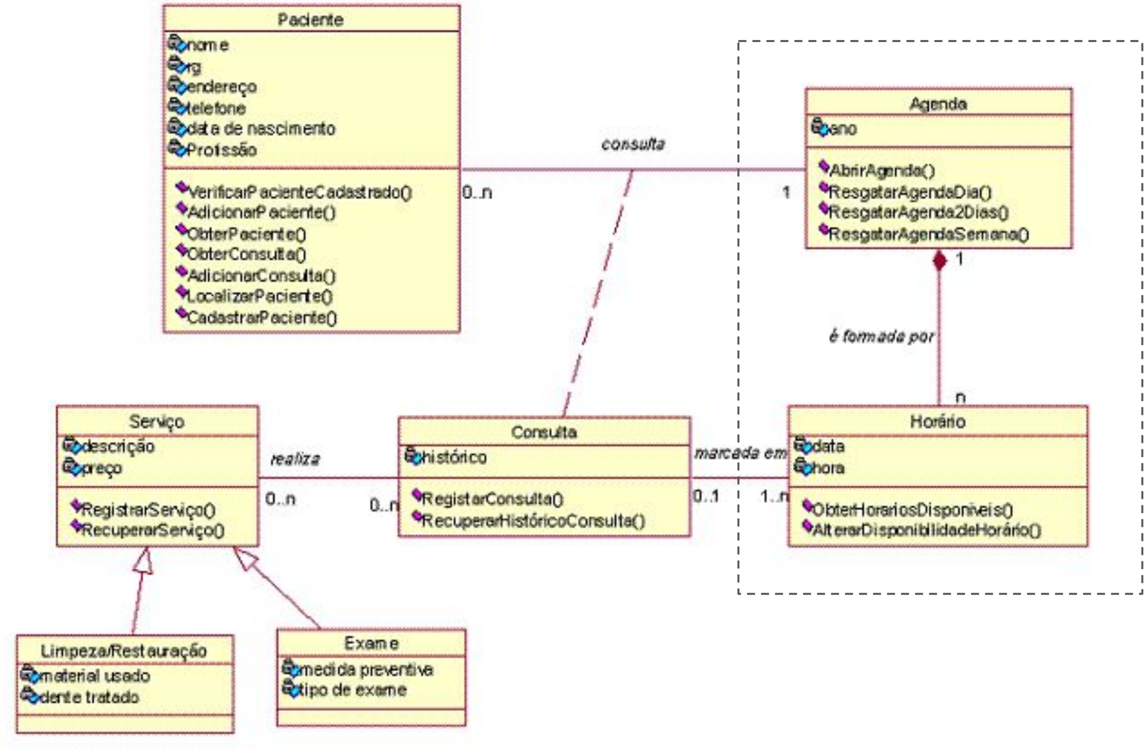
Diagrama obtido na Web !



Quais classes se caracterizam
como
Código-cliente ?

Do you remember what “client code” means ?

Diagrama obtido na Web !

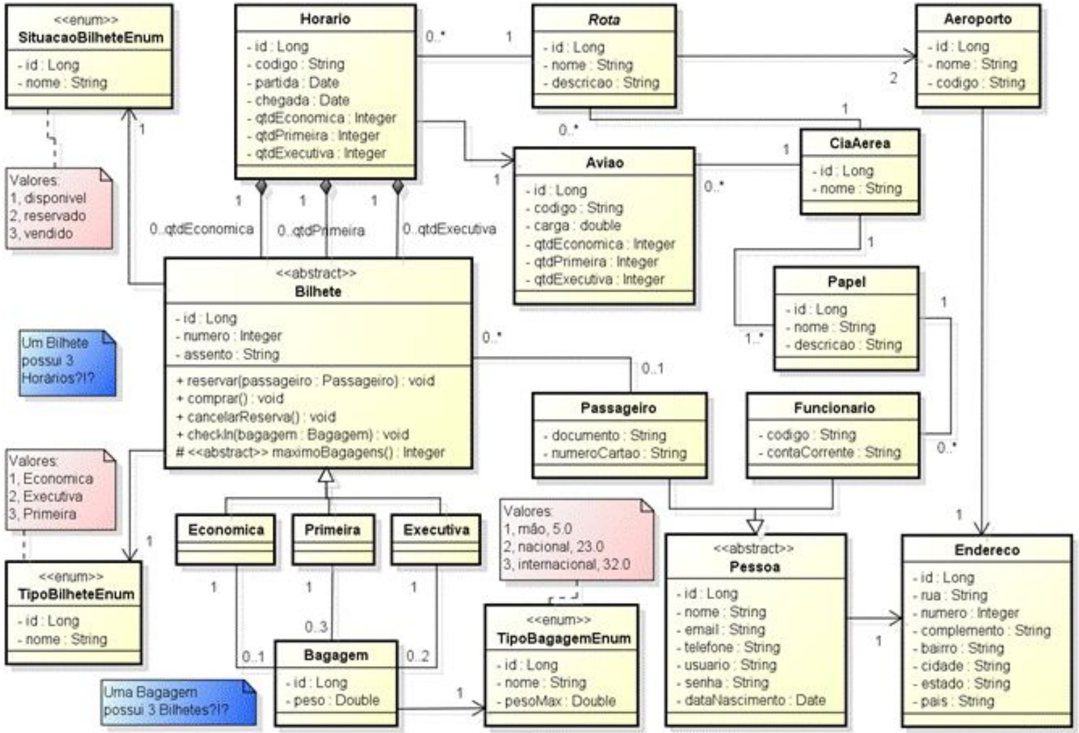


Quais classes se caracterizam como Código-cliente ?

Importante ! Agregações acabam sendo representações de uma única entidade….

Do you remember what “client code” means ?

Diagrama obtido na Web !



Quais classes se caracterizam como
Código-cliente ?

A análise de códigos que são **clientes** deve ser feita em nível de classes, sempre procurando identificar classes que atuam como clientes de outras.

A análise também faz sentido de for expandida para nível de pacotes… identificando pacotes que são clientes de outros

Do you remember what “client code” means ?

Diagrama obtido na Web !

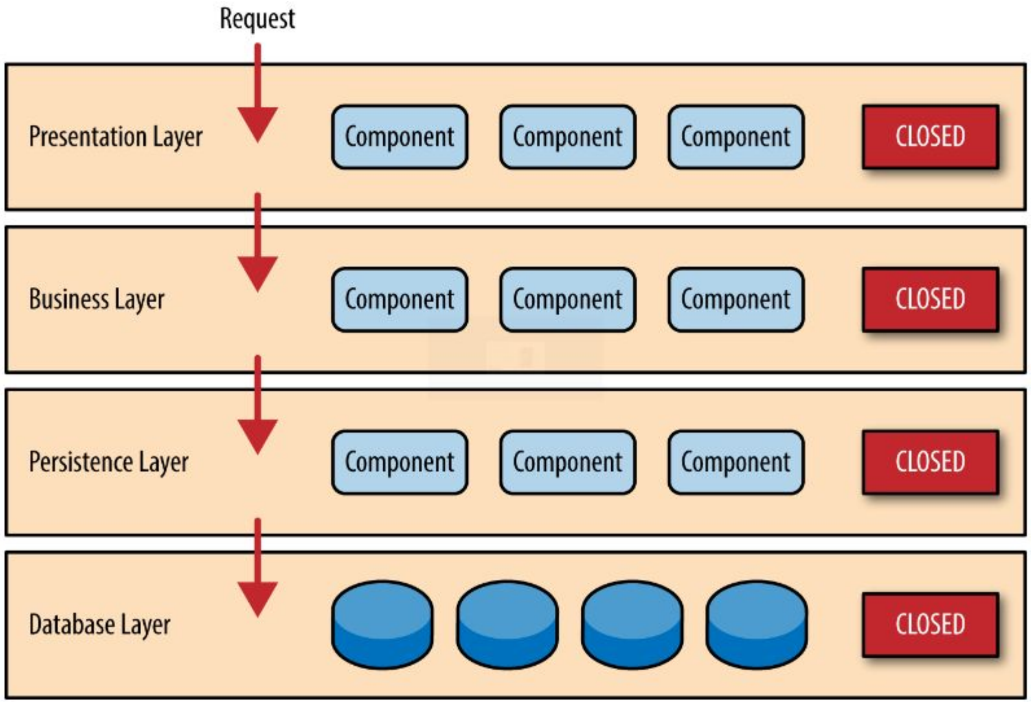
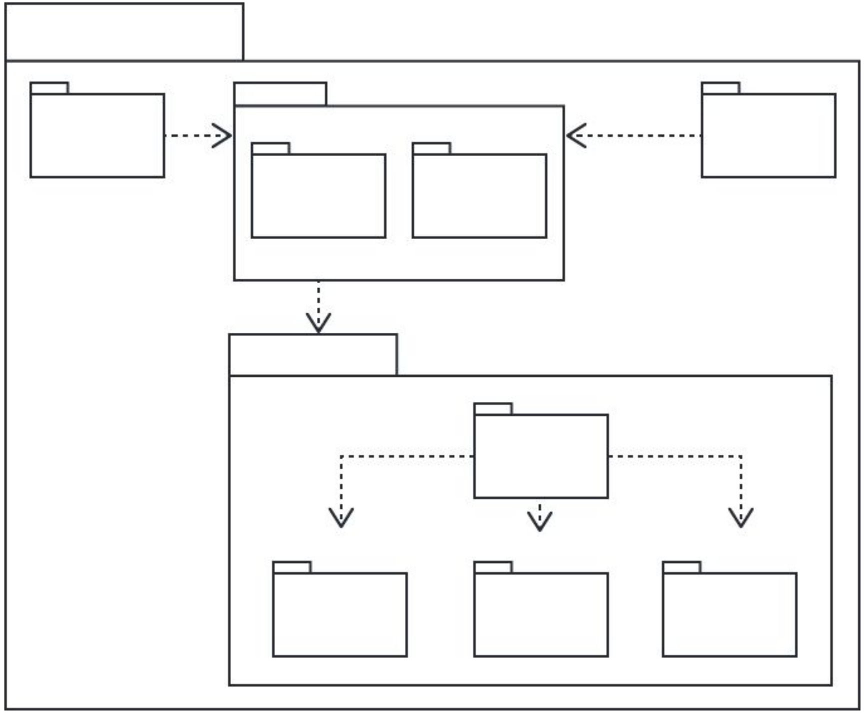


Figure 1-2. Closed layers and request access

Quais classes se caracterizam
como
Código-cliente ?

Do you remember what “client code” means ?

Diagrama obtido na Web !



Quais classes se caracterizam como Código-cliente ?

Estrutura muito boa ! Separação do sistema em três camadas, sendo cada uma um pacote.

GAMMA´S DESIGN PATTERNS

		Purpose		
		Creational	Structural	Behavioral
Escopo	Class	Factory Method	Adapter	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Façade Flyweight proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

TEMPLATE FOR DESCRIBING PATTERNS

- Pattern Name and Classification
- Intent
- Also Known As
- Applicability
- Structure

- Participants
- Collaborations
- Consequences
- Implementation
- Sample Code
- Known Uses
- Related Patterns

Singleton

(Creational pattern)

ITI MBA

INFORMAÇÃO
TECNOLOGIA
& INOVAÇÃO

SINGLETON

When to use ?

You should use this pattern when classes must have just one instance

SINGLETON

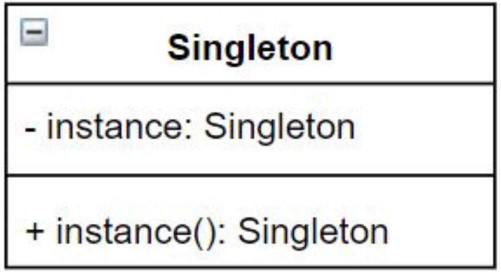
Examples

Classes responsible for establishing connections with DB. It is not necessary create a new connection object every time you need to connect to the DB… this will overload memory and make your code polluted. You can use the same object every time.

Classes that represent sensors in robots

Classes responsible for logging events of the system execution

SINGLETON



Singleton

- instance: Singleton

+ instance(): Singleton

Java

```java
public class Singleton {

    private static Singleton instance;
    private Singleton() {
      this.instance = new Singleton()
    }

    public Singleton getInstance(){
      if (instance == null)
      {
            instance= new Singleton();
      }
      return instance;
}}
```

Python

```python
class Singleton:

    __instance = None

    @classmethod
    def instance(cls): #cls == class
        if cls._instance is None:
            cls._instance = cls()
        return cls._instance
```

Observe que usar esse padrão não significa que você terá que criar uma classe chamada Singleton…. Ela deve ser vista como um papel que outras classes assumirão…

ROBOT SENSOR - SINGLETON IN JAVA

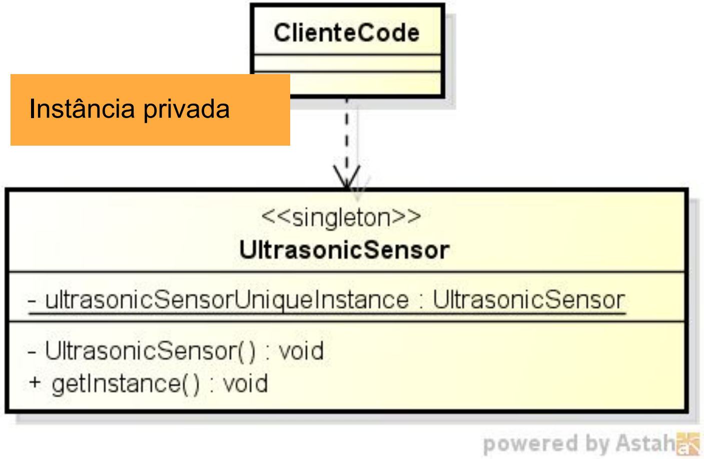
```
public class UltrasonicSensor {

    private static UltrasonicSensor ultrasonicSensorUniqueInstance;

    private UltrasonicSensor()  { ... }

    public static UltrasonicSensor getInstance(){

        if (ultrasonicSensorUniqueInstance ==null)
        {
                synchronized (UltrasonicSensor.class)
                {
                    if (ultrasonicSensorUniqueInstance ==null) {
                        ultrasonicSensorUniqueInstance =new UltrasonicSensor();
            }       }
        return ultrasonicSensorUniqueInstance;
    }
}
```

Construtor privado

Instância privada

Esse método é que Faz o papel do construtor

Aqui é onde efetivamente se cria um objeto dessa classe ....


ClienteCode
<<singleton>>
UltrasonicSensor
- ultrasonicSensorUniqueInstance : UltrasonicSensor
- UltrasonicSensor() : void
+ getInstance() : void
powered by Astah

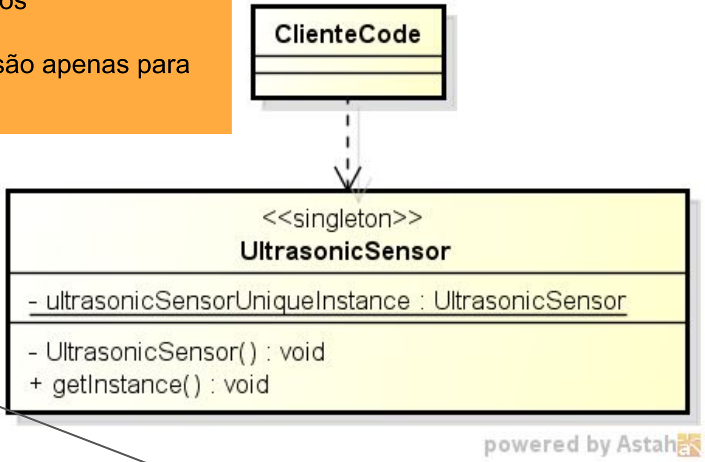
Como dito no slide anterior, essa classe assume o papel como um singleton.
Não é necessário criar uma classe chamada Singleton

INFORMAÇÃO TECNOLOGIA & INOVAÇÃO

ROBOT SENSOR - SINGLETON IN PYTHON

```
class UltrasonicSensor:

    _ultrasonicSensorinstance =None

    @classmethod
    def instance(cls): #cls == class
        if (cls.ultrasonicSensorinstance is None):
            cls.ultrasonicSensorinstance =cls()
        return cls.ultrasonicSensorinstance

if __name__ == "__main__":
    ultrasonicSensor1 = UltrasonicSensor.instance()
    ultrasonicSensor2 = UltrasonicSensor.instance()

    print(ultrasonicSensor1 == ultrasonicSensor2)#true
```

No Python não existe métodos/atributos verdadeiramente privados.
Os símbolos de underline colocados são apenas para convenção….



ClienteCode

<<singleton>>
UltrasonicSensor

- ultrasonicSensorUniqueInstance : UltrasonicSensor

- UltrasonicSensor() : void
+ getInstance() : void

Cria um objeto da classe

Provando que os dois objetos são o mesmo !

INFORMAÇÃO
TECNOLOGIA
& INOVAÇÃO

Exercício

Criar uma classe em Java, Python ou Pseudo-Código que seja responsável por criar conexões com Banco de Dados.

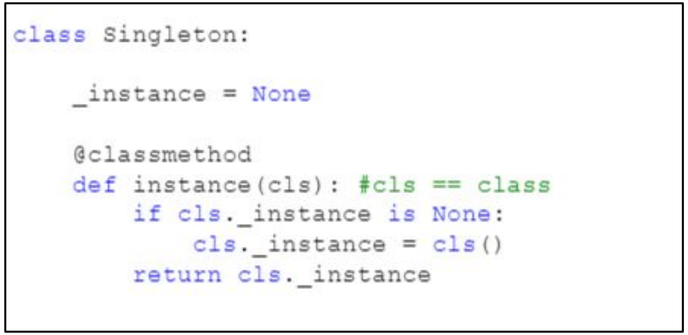
Primeira resolução em Python

```python
class DatabaseConnection:

    __dbInstance = None

    @classmethod
    def getInstance(cls):
        if cls.__dbInstance is None:
            cls.__dbInstance = cls()
            cls.__dbInstance.connect()
        return cls.__dbInstance

    def connect(self):
        print("Connected to database")

if __name__ == "__main__":

    print(DatabaseConnection.getInstance() == DatabaseConnection.getInstance())
```

DB CONNECTION - SINGLETON IN JAVA

```
public class DatabaseConnection {

    private static DatabaseConnection instance = null;

    private DatabaseConnection() {
                    // Establish a database connection
            ...
        }

        public synchronized static DatabaseConnection instance() {

            if( instance == null ) {
                instance = new DatabaseConnection( );
            }
            return instance;

        }

}
```

Pontos importantes que caracterizam o Singleto



```
class Singleton:

    _instance = None

    @classmethod
    def instance(cls): #cls == class
        if cls._instance is None:
            cls._instance = cls()
        return cls._instance
```

DB CONNECTION - SINGLETON IN JAVA

```
public class Client {

    public void printTranscript(String student) {

      DatabaseConnection db = DatabaseConnection.instance();
      ...
      String grade = db.getAssignmentGrade (student, course, assignment);
      ...
    }
}
```

Strategy

(Behavioral pattern)

ITI
MBA

INFORMAÇÃO
TECNOLOGIA
& INOVAÇÃO

STRATEGY

When to use?

Use the Strategy pattern when you want to use different variants of an algorithm (family of algorithms) and be able to switch between them during execution.

STRATEGY

Example:

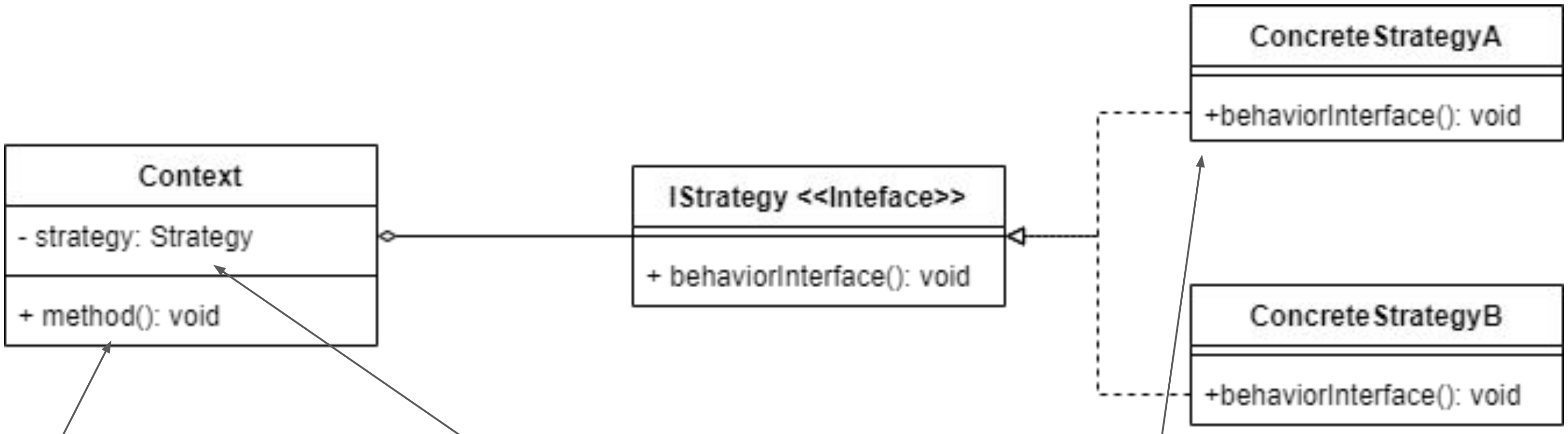
Consider the example of sorting. We implemented bubble sort, but the data started to grow and bubble sort started getting very slow.

In order to tackle this we implemented Quick sort. But now although the quick sort algorithm was doing better for large datasets, it was very slow for smaller datasets.

In order to handle this we implemented a strategy **where for small datasets, bubble sort will be used and for larger, quick sort**.

Another example is regarding google meetings. If the quality of the connection is low the image resolution is also low to avoid “travamentos”. If the connection quality is high, the image resolution can also be high. This can vary in runtime.

STRATEGY



Correção: O tipo aqui deve ser IStrategy ! … sorry

Uma regra de negócio qualquer ….

Aqui são duas estratégias, mas poderia have várias !

STRATEGY IN PYTHON

Uso de polimorfismo… lembram-se ?
Qualquer estratégia concreta pode ser passada aqui !

```
class Strategy:
    def behavior(self) -> None:
        raise NotImplementedError


class ConcreteStrategyA(Strategy):
    def behavior(self) -> None:
        Pass


class ConcreteStrategyB(Strategy):
    def behavior(self) -> None:
        Pass
```

```
class Context():

    def   init  (self, strategy: Strategy) -> None:
        self.__strategy = strategy

    def method(self) -> None:
        self.__strategy.behavior()
```

Vai chamar o behavior da estratégia que estiver instanciada no momento ….

Novamente…. A utilização desse padrão não significa que você criará uma classe chamada Strategy. Na verdade, as classes existentes do sistema assumirão esses “papéis”.

STRATEGY IN JAVA

```
public abstract class  IStrategy {

     public abstract void behaviorInterface();
}
```

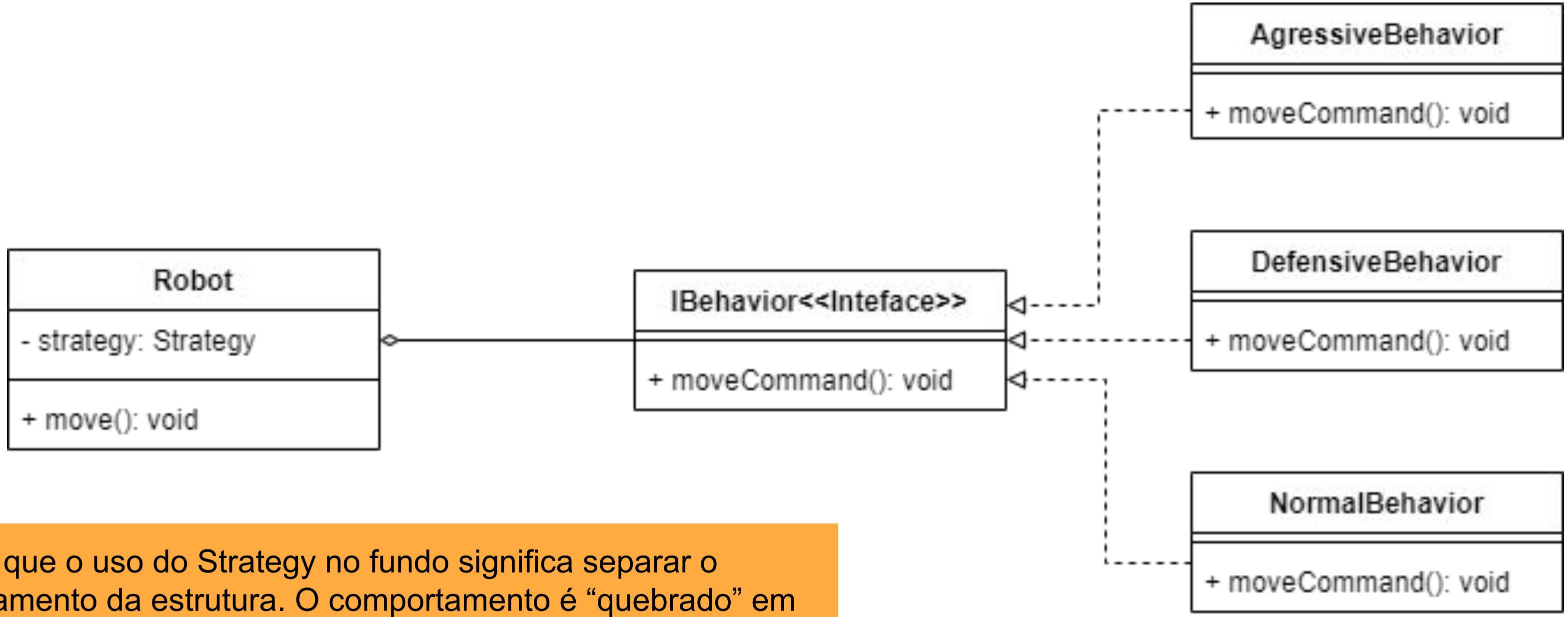
```
public class Context {
     private strategy  IStrategy;
     ...
}
```

```
public class ConcreteStrategyA extends IStrategy {

    public void behaviorInterface(){
    //code for behaviorInterface A
    }
}
```

```
public class ConcreteStrategyB extends IStrategy {

    public void behaviorInterface(){
    //code for behaviorInterface B
    }
}
```

```
public class ConcreteStrategyC extends IStrategy {

    public void behaviorInterface(){
    //code for behaviorInterface C
    }
}
```

ROBOT EXAMPLE



Perceba que o uso do Strategy no fundo significa separar o comportamento da estrutura. O comportamento é “quebrado” em várias classe separadas … e removidos da própria classe …
Se esse padrão não fosse usado, provavelmente, todos os comportamentos estariam como métodos dentro da classe Robot.

ROBOT EXAMPLE - IN JAVA

```
public interface IBehaviour {
      public int moveCommand();
}
```

```
public class DefensiveBehaviour implements Ibehaviour {
   public int moveCommand()
   {
     System.out.println("\tDefensive Behaviour: if find
      another robot run from it");
     return -1;
    }
}
```

```
public class AgressiveBehaviour implements Ibehaviour{

     public int moveCommand()
    {
             System.out.println("\tAgressive Behaviour: if
find another robot attack it");
             return 1;
     }
}
```

```
public class NormalBehaviour implements Ibehaviour {

    public int moveCommand()
    {
        System.out.println("\tNormal Behaviour: if
find
        another robot ignore it");
        return 0;
     }
}
```

ROBOT EXAMPLE - IN JAVA

```
public class Robot {

    IBehaviour behaviour;
    String name;
    public Robot(String name)
    {
         ...
    }

    public void move()
    {
        ...
        int command = behaviour.moveCommand();
        ....
     }
.....
```

Represent the behavior of the class, i.e., all methods that would be here, they are not anymore !

It is a way of separating the structure from the behavior. The class is less propense to receive modifications

ROBOT EXAMPLE - IN JAVA

```java
public class Main {
public static void main(String[] args) {
     Robot r1 = new Robot("Big Robot");
     Robot r2 = new Robot("C3PO");
     Robot r3 = new Robot("R2D2");

     r1.setBehaviour(new AgressiveBehaviour());
     r2.setBehaviour(new DefensiveBehaviour());
     r3.setBehaviour(new NormalBehaviour());
     r1.move();
     r2.move();
     r3.move();
     if (some condition)
          r1.setBehaviour(new DefensiveBehaviour());
     ....
     if (some condition)
         r2.setBehaviour(new AgressiveBehaviour());


     r1.move();
     r2.move();
     r3.move();
   }
}
```

Initial behavior (Default)

Behavioral changes in runtime

Behavioral changes in runtime

Interessante observar que se o padrão não estivesse aplicado e todos os comportamentos estivessem dentro de uma única classe, até mesmo essa lógica de troca de um comportamento para o outro também estaria todo dentro daquela classe única. Com o uso do padrão, cada comportamento é encapsulado em uma classe separada e até a lógica de troca (condição) é removida e também pode ser encapsulada em outro local, como mostra esse método Main.

ROBOT EXAMPLE - IN PYTHON

```python
class BehaviorStrategy:

    def moveCommand(self) -> None:
        raise NotImplementedError
```

```python
class DefensiveBehavior(BehaviorStrategy):
    def moveCommand(self) -> None:
        print("Defensive Behaviour: if find another robot run from it)"
```

```python
class AgressiveBehavior(BehaviorStrategy):
    def moveCommand(self) -> None:
        print("Agressive Behaviour: if find
```

```python
class NormalBehavior(BehaviorStrategy):
    def moveCommand(self) -> None:
        print("Normal Behaviour: if find another robot ignore it)"
```

ROBOT EXAMPLE - IN PYTHON

```
if __name__ == "__main__":
    robot1 = Robot("Big Robot")
    robot2 = Robot("C3PO")
    robot3 = Robot("R2D2")

    robot1.behavior(AgressiveBehavior())
    robot2.behavior(DefensiveBehavior())
    robot3.behavior(NormalBehavior())

    robot1.move()
    robot2.move()
    robot3.move()

    if condition:
        robot1.behavior(DefensiveBehavior())
    ...
    if condition:
        robot2.behavior(AgressiveBehavior())

    robot1.move()
    robot2.move()
    robot3.move()
```

Initial behavior (Default)

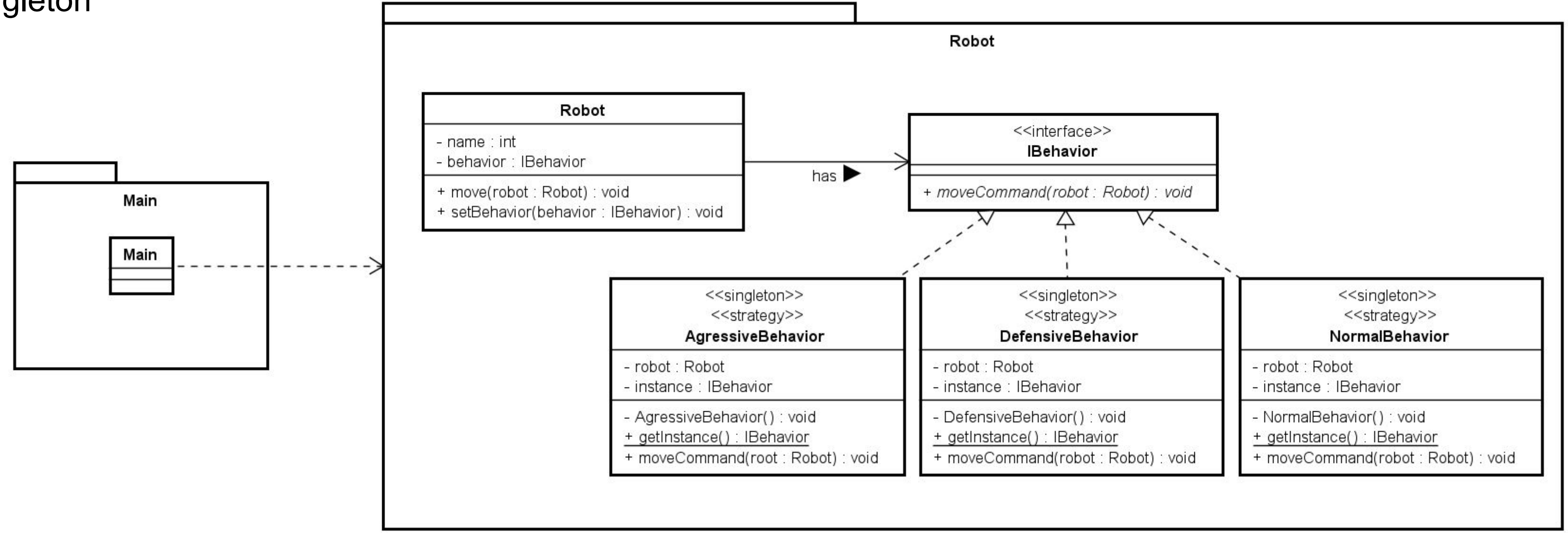
Behavioral changes in runtime

Behavioral changes in runtime

Strategy + Singleton

Notem que, na maioria dos casos, não faz sentido existir mais do que **uma única instância** de cada estratégia em tempo de execução.

Dessa forma é bastante normal que cada **estratégia** seja um singleton



Main

Main

Robot

Robot
- name : int
- behavior : IBehavior
+ move(robot : Robot) : void
+ setBehavior(behavior : IBehavior) : void

has ▶

<<interface>>
IBehavior
+ *moveCommand(robot : Robot) : void*

<<singleton>>
<<strategy>>
AgressiveBehavior
- robot : Robot
- instance : IBehavior
- AgressiveBehavior() : void
+ getInstance() : IBehavior
+ moveCommand(root : Robot) : void

<<singleton>>
<<strategy>>
DefensiveBehavior
- robot : Robot
- instance : IBehavior
- DefensiveBehavior() : void
+ getInstance() : IBehavior
+ moveCommand(robot : Robot) : void

<<singleton>>
<<strategy>>
NormalBehavior
- robot : Robot
- instance : IBehavior
- NormalBehavior() : void
+ getInstance() : IBehavior
+ moveCommand(robot : Robot) : void

Strategy + Singleton

```
public class Main {


        public static void main(String[] args) {

            Robot r1 = new Robot("Big Robot");
            Robot r2 = new Robot("C3PO");
            Robot r3 = new Robot("R2D2");

            r1.setBehavior(AgressiveBehavior.getInstance());
            r2.setBehavior(DefensiveBehavior.getInstance());
            r3.setBehavior(NormalBehavior.getInstance());

            r1.move(r1);
            r2.move(r2);
            r3.move(r3);

            r1.setBehavior(DefensiveBehavior.getInstance());
            r1.move(r1);


        }
```

```
public class Robot {

        private IBehavior behavior;
        private String name;

         public Robot(String name) {
                 this.name = name;
         }

         public void move(Robot robot) {
                 behavior.moveCommand(robot);
         }
         public void setBehavior(IBehavior behavior) {

                 this.behavior = behavior;
         }

}
```

Strategy + Singleton

```
public interface IBehavior {

        public void moveCommand(Robot robot);

}
```

```
public class AgressiveBehavior implements IBehavior {

          private static IBehavior instance = null;  //singleton
          private Robot robot;

               //singleton
               private AgressiveBehavior() {          }

               public void setRobot(Robot robot) {
                       this.robot = robot;
               }
               //singleton

               public static IBehavior getInstance() {
                 if (instance == null) {
                       instance = new AgressiveBehavior();
                 }
                return instance;
               }

         public void moveCommand(Robot robot) {
           System.out.println("The robot " + robot.toString() + " will attack you
        now ");
        }
}
```

HOMEWORK FOR 12/04

- Suponha um sistema de geração de rotas seguras. O App deve gerar uma rota que se classifica em “altamente segura”, “segura” e “aceitável”. Como a rota deve/pode ser alterada à medida que o pedestre caminha, o sistema deve ser capaz de trocar o nível de segurança da rota em tempo de execução. Assim, ora a rota se caracteriza como “altamente segura”, ora como “segura” e ora como “aceitável”.
- Elabore um projeto que reflita essa dinamicidade do sistema
- **Observações**
  - Como a memória do dispositivo é limitada, deve-se cuidar para que não haja sobrecarga de objetos desnecessários em memória
- **Forma de resolução**
  - Diagrama de classe UML
  - Trechos de código (pseudo-código) das partes importantes

FIM