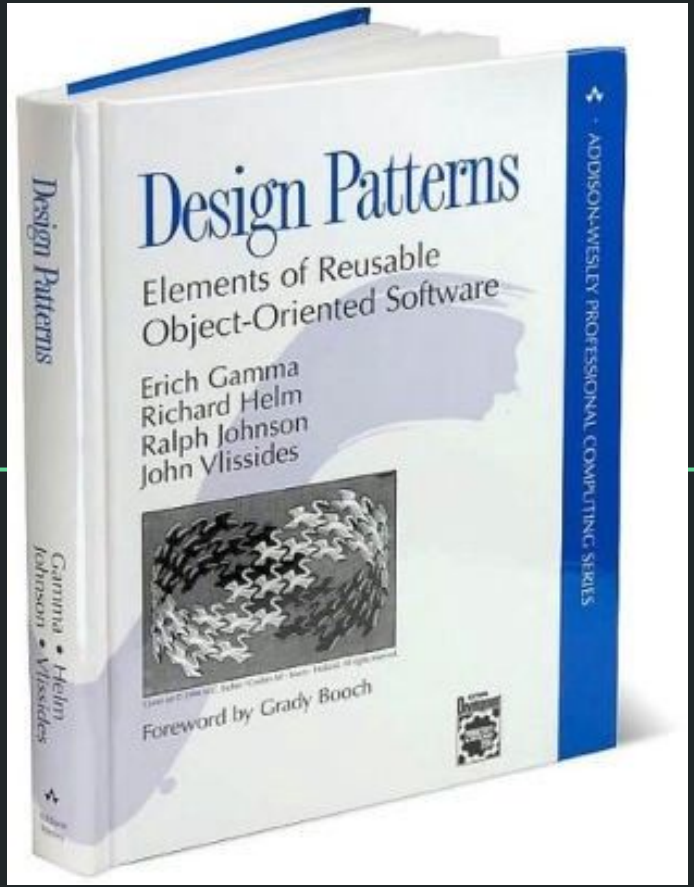
Nesta aula: Introdução a Padrões de Projeto, Padrão Factory Method e Visibilidade de Pacotes

Programação Orientada a Objetos Avançada - 2025/1

Prof. Dr. Valter Vieira de Camargo
Departamento de Computação - UFSCar

Design Patterns

Padrões de Projeto...
De “**projeto**”... não de “**código**” nem de “**análise**”



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
- PLOP Conference ➜ https://www.hillside.net/plop/2023/

O que são *design* patterns ?

- Design patterns são soluções para problemas recorrentes no desenvolvimento de software
- Popularizaram-se com o livro Design Patterns publicado em 1995 - Gangue dos 4 !
- Cada padrão é uma “forma” de se estruturar as classes/métodos/atributos de forma a resolver um problema recorrente
- Não são apenas soluções, mas soluções que visam a facilitar a evolução do sistema, pois baseiam-se totalmente em:
  - Separar código-cliente do código fornecedor (que é usado)
  - poilmorfismo/herança de forma a facilitar a inserção de novas classes sem afetar quem usa
- Uma grande vantagem também é a inserção de um vocabulário para que a equipe possa conversar mais facilmente sobre as soluções

“Design” é usualmente traduzido para “projeto” em português, mas geralmente não exprime muito bem o significado.

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 ?

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

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

Factory Method

FACTORY METHOD

Hoje você sabe quais são os tipos que o seu código cliente deve trabalhar, mas você também sabe que novos tipos irão aparecer no futuro ….

When to use?

Two situations:

1. Use Factory Method when you don't know in advance the exact types and dependencies of the objects your code should work with.
2. Use Factory Method when you want to make the client code independent (unconsciousness) of the concrete objects it uses

FACTORY METHOD

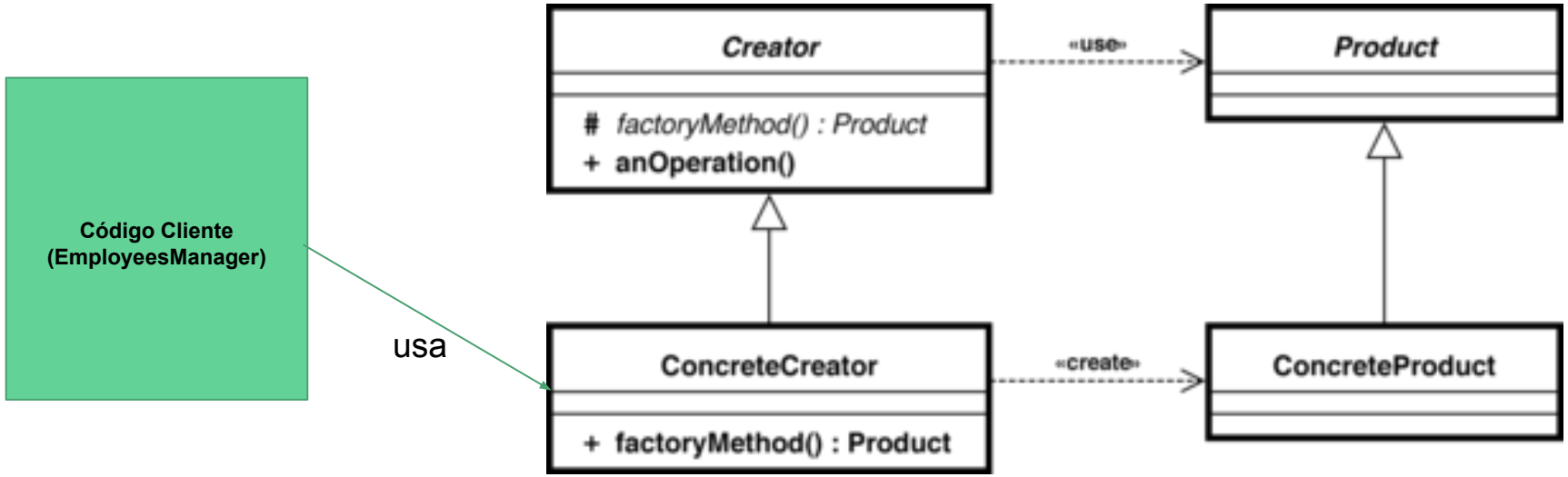
Example

Consider the case of a class for managing employees… one of the responsibilities of this class is **to calculate the salaries of each employee type** (secretaries, sales person, managers, technical leaders, etc). It would be good if this class (EmployeeManager (client code)) did not know the specific types the employees it manipulates…

```
class EmployeesManager {

    public calculateSalaries() {

        ...
        Employee e1 = new Secretary();
        ...
        Employee e2 = new SalesPerson();
        ...
```

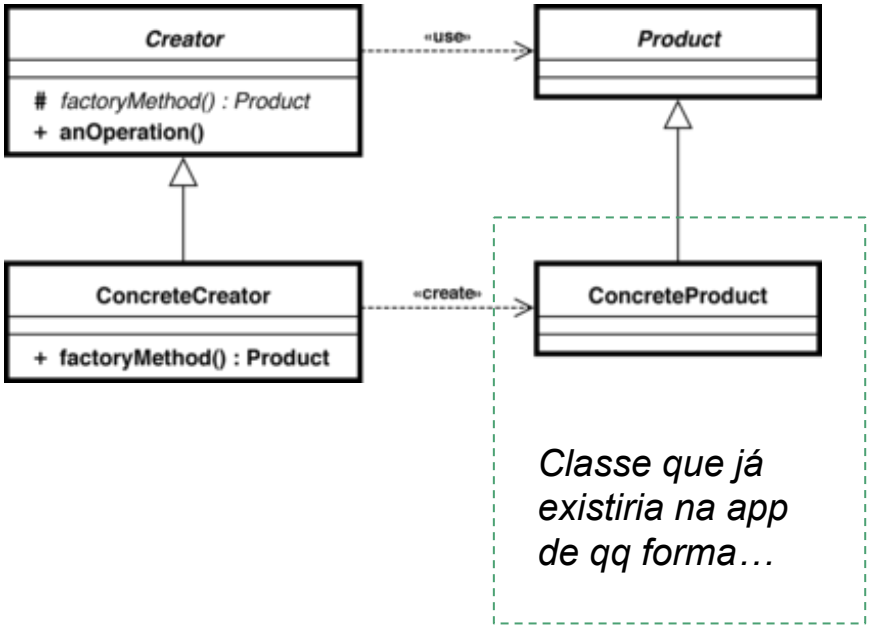
This is not good ……
My **client code** will be very dependent on the employee types..

FACTORY METHOD



O código cliente teria que criar diretamente (se não usar o padrão) o ConcreteProduct. Se ele criasse diretamente (dando new), ele ficaria dependente desse tipo existente …. se for possível evitar essa dependência, melhor !

FACTORY METHOD IN PYTHON

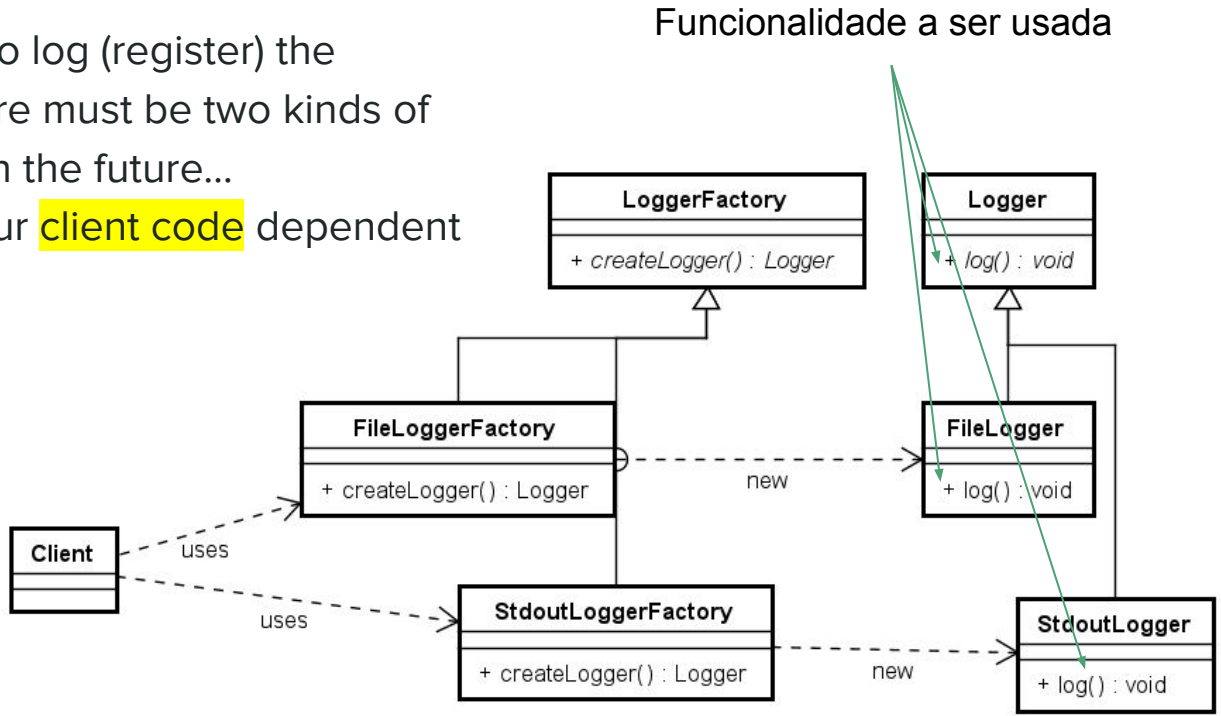



```python
class Creator:
    def factoryMethod(self) -> Product:
        raise NotImplementedError

class Product:
    def doStuff(self) -> None:
        raise NotImplementedError

class ConcreteCreator(Creator):
    def factoryMethod(self) -> Product:
        return ConcreteProduct()

class ConcreteProduct(Product):
    def doStuff(self) -> None:
        pass
```

FACTORY METHOD

Context:

- Your system must have way to log (register) the execution of the system. There must be two kinds of logs, but others can appear in the future...
- You wouldn´t like to make your client code dependent on the types of loggers...



FACTORY METHOD IN PYTHON

```
import logging
import sys

class Logger:
    def log(self) -> None:
        raise NotImplementedError

class FileLogger(Logger):

    _filePath: None

    def __init__(self, filePath: str) -> None:
        self._filePath = filePath

    def log(self) -> None:
        logging.basicConfig()
        logging.info('This message will get logged on to a file')

class StdoutLogger(Logger):

    def log(self) -> None:
        handler = logging.getLogger()
        handler.setLevel(logging.INFO)
        ch = logging.StreamHandler(sys.stdout)
        ...
        handler.info('This message will get logged on to a stdout')
```

```
class LoggerFactory:

    def createLogger(self) -> Logger:
        raise NotImplementedError

class FileLoggerFactory(LoggerFactory):
    def createLogger(self) -> Logger:
        return FileLogger('file.log')

class StdoutLoggerFactory(LoggerFactory):
    def createLogger(self) -> Logger:
        return StdoutLogger()
```

new()
Veja que isto é uma delegação
...

```
if __name__ == "__main__":
    fileLogger = FileLoggerFactory().createLogger()
    fileLogger.log()

    stdoutLogger =
StdoutLoggerFactory().createLogger()
    stdoutLogger.log()
```

Client code !

FACTORY METHOD IN JAVA

```
abstract class Logger:

    public void abstract log(Logger);

class FileLogger extends Logger {

    private String filePath

    public FileLogger(String filePath){
       this.filePath = filePath;
    }

    public void log (Logger) {
        logging.basicConfig()
        logging.info('This message will get logged on to a file')
    }
class StdoutLogger extends Logger {

    public void log(Logger) {
       ….
       Sustem.out.println(“This message will be logged to a stdout”)

}
```

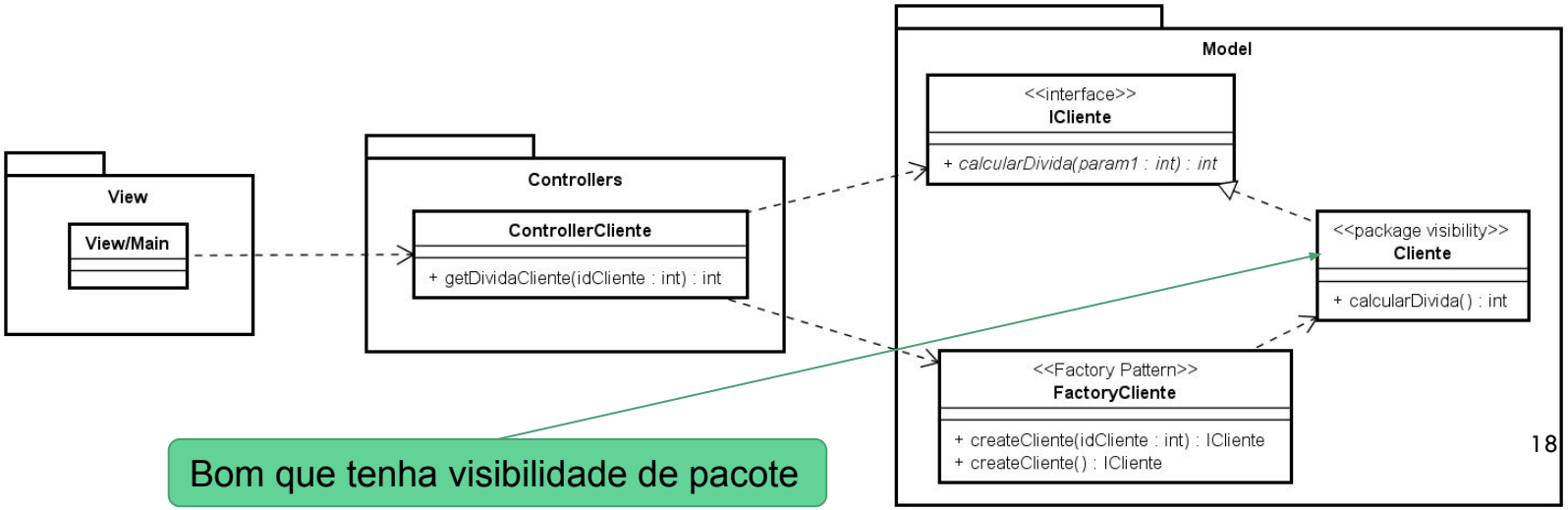
Visibilidade de pacote (se possível)

```
public abstract class LoggerFactory:

       public abstract Logger createLogger() ;

public class FileLoggerFactory extends LoggerFactory {
       public Logger createLogger() {
              return new FileLogger();
}


public class StdoutLoggerFactory extends LoggerFactory {
       public Logger createLogger() {
              Return new StdoutLogger();

}

public class main {
       …
       Logger logger = FileLoggerFactory.createLogger();
       logger.log();
       …
       logger = StdoutLoggerFactory.createLogger();
       logger.log();
```

Client code !

Visibilidade de Pacote + Factory

- Ótimo para gerenciar/controlar o acesso a classes entre pacotes
- Permite que você decida se classes de um pacote podem ser vistas por outro
- Se você atribuiu visibilidade de pacote a uma classe, ela não poderá ser nem ao mesmo importada dentro de outro pacote (pode mudar de linguagem para linguagem)



Visibilidade de pacote em Java

- Em Java, uma classe que tem visibilidade de pacote não é vista por classes fora daquele pacote, isto é, ela só é vista por classes dentro do mesmo pacote
- A forma de declarar é simplesmente não atribuindo nenhum modificador de visibilidade
- Simplesmente:

```
class Cliente {


}
```

Visibilidade de Pacote + Factory

```
package View;

import Controller.*;

public class View {

     public static void main (String args[]){

     float divida = Controller.getDividaCliente(1);

     System.out.println("A divida do cliente é " + divida);

     }

}
```

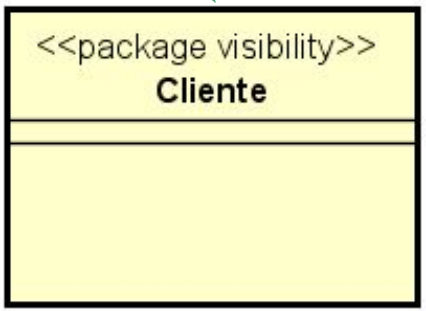
```
package Controller;

import Model.*;

public class Controller  {

public static float getDividaCliente(int idCliente){

    ICliente c = FactoryCliente.createCliente(idCliente);

    return c.calcularDivida(idCliente);
    }
}
```

em Java, se não colocar “public”, nenhum pacote externo conseguirá visualizar ..

```
package Model;

public interface ICliente {

    public float calcularDivida();

}
```

```
package Model;

class Cliente implements ICliente {

    public float calcularDivida() {

    return x*¾*(z + u);
    }
}
```

Represente a visibilidade de pacote com um estereótipo, pois não há uma convenção de como deixar isso evidente

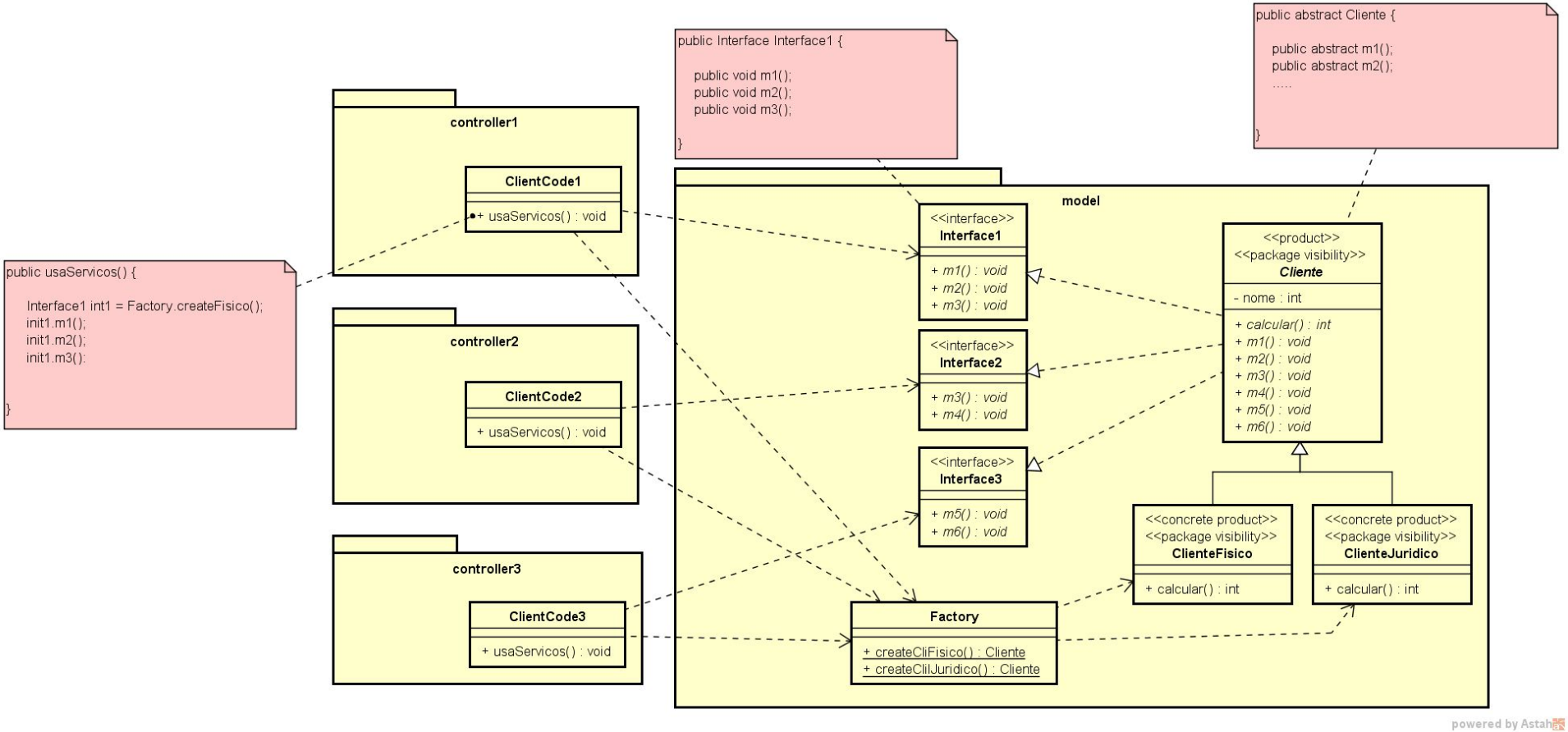



```
package Model;

public class FactoryCliente {

   public static ICliente criaObjetosCliente(int id){

       return //recupera cliente do BD;
   }
}
```

Atividade Prática

Implemente a seguinte situação:

- Um sistema deve manipular clientes físicos e jurídicos
- A família de clientes possui 6 serviços que são oferecidos (m1, m2, m3, m4, m5 e m6)
- existem três sistemas externos que usam serviços oferecidos pela família de clientes (três controladores)
  - O sistema externo 1 usa os serviços m1, m2 e m3
  - O sistema externo 2 usa os serviços m3 e m4
  - O sistema externo 3 usa os serviços m5 e m6
- Requisito não-funcional
  - Os clientes externos não devem conhecer os tipos concretos existentes de clientes
- Nos controladores, use os serviços disponibilizados.
- Os “clientes” não devem ser vistos fora do pacote

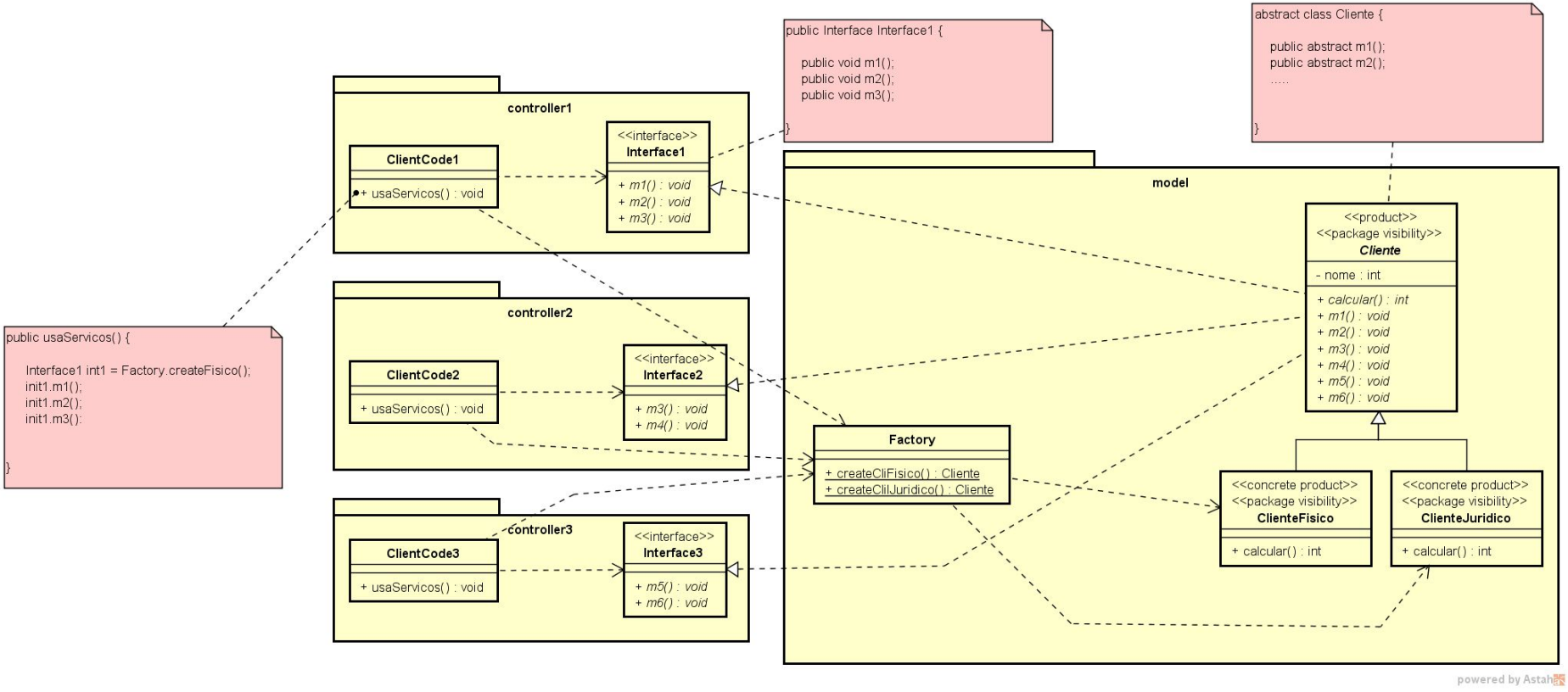

public Interface Interface1 {
public void m1();
public void m2();
public void m3();
}
public abstract Cliente {
public abstract m1();
public abstract m2();
.....
}
public usaServicos() {
Interface1 int1 = Factory.createFisico();
init1.m1();
init1.m2();
init1.m3();
}
controller1
ClientCode1
+ usaServicos() : void
controller2
ClientCode2
+ usaServicos() : void
controller3
ClientCode3
+ usaServicos() : void
model
<<interface>>
Interface1
+ m1() : void
+ m2() : void
+ m3() : void
<<interface>>
Interface2
+ m3() : void
+ m4() : void
<<interface>>
Interface3
+ m5() : void
+ m6() : void
<<product>>
<<package visibility>>
Cliente
- nome : int
+ calcular() : int
+ m1() : void
+ m2() : void
+ m3() : void
+ m4() : void
+ m5() : void
+ m6() : void
<<concrete product>>
<<package visibility>>
ClienteFisico
+ calcular() : int
<<concrete product>>
<<package visibility>>
ClienteJuridico
+ calcular() : int
Factory
+ createCliFisico() : Cliente
+ createCliJuridico() : Cliente
powered by Astah

Melhor resolução

Notem que na solução anterior, os controllers acabam tendo acesso a interfaces que eles não deveriam ver, já que teriam que importar o pacote model inteiro.

Uma solução é colocar as interfaces dentro do pacote dos controllers.

Este formato é o esquema usado na Clean Architecture de Inversão de Dependências



Entretanto, se allguém importar um controller dentro do outro ainda poderá acessar os outros controllers ... as interfaces não podem ter visibilidade de pacote pq devem ser implementadas pelo Cliente.