

EAPLI

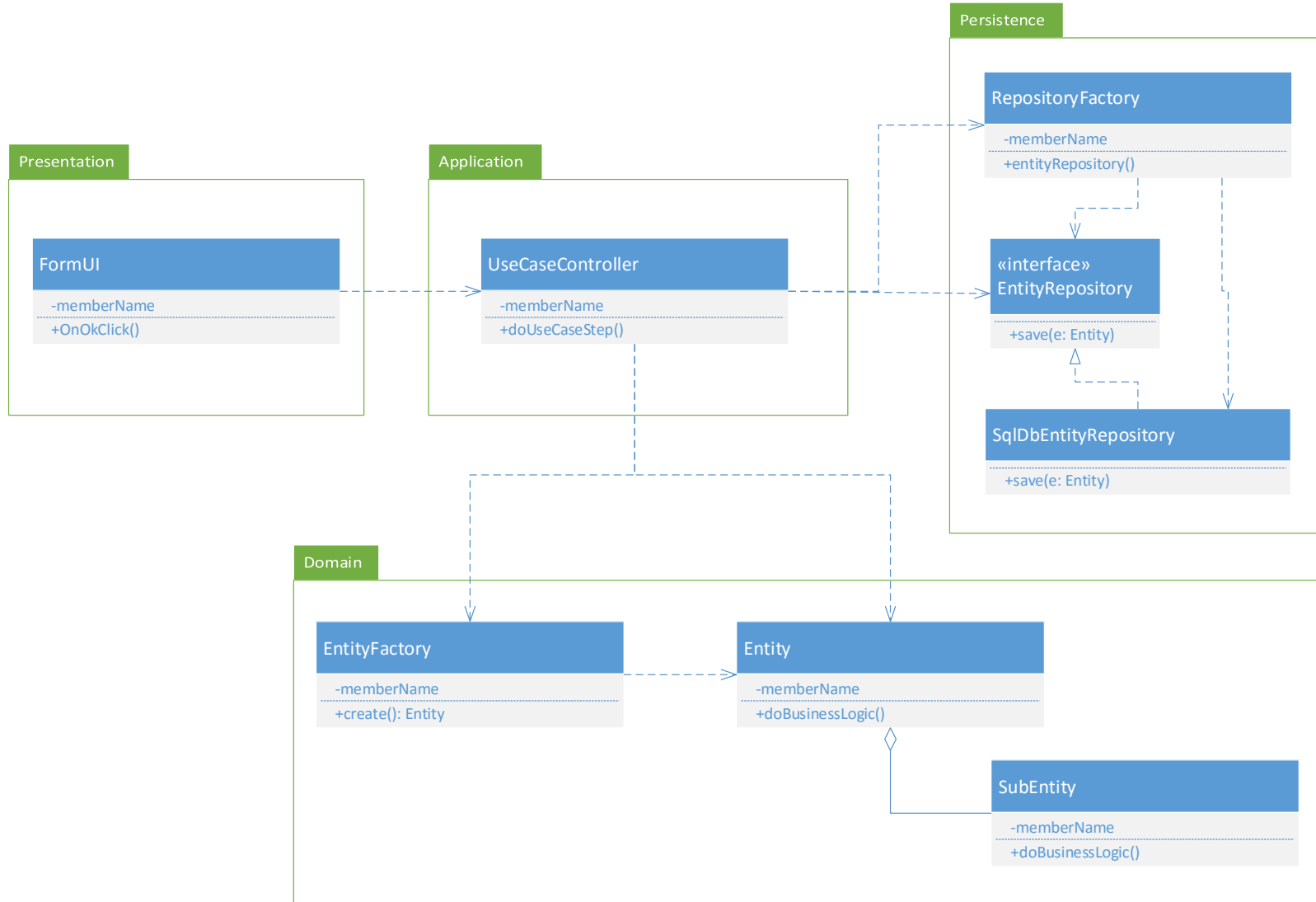
Princípios de Design OO: Padrões e Responsabilidades

Paulo Gandra de Sousa
pag@isep.ipp.pt

Questões comuns

- A que classe atribuir uma dada responsabilidade?
- Como organizar as responsabilidades do sistema?
- Quem deve ter responsabilidade de coordenar a interação de caso de uso?
- Quem deve ter a responsabilidade de representar e implementar a lógica de negócio?
- Como gerir o ciclo de vida de um objeto?
 - Criar um objeto?
 - persistir objetos?
- Como proteger o código para modificação?

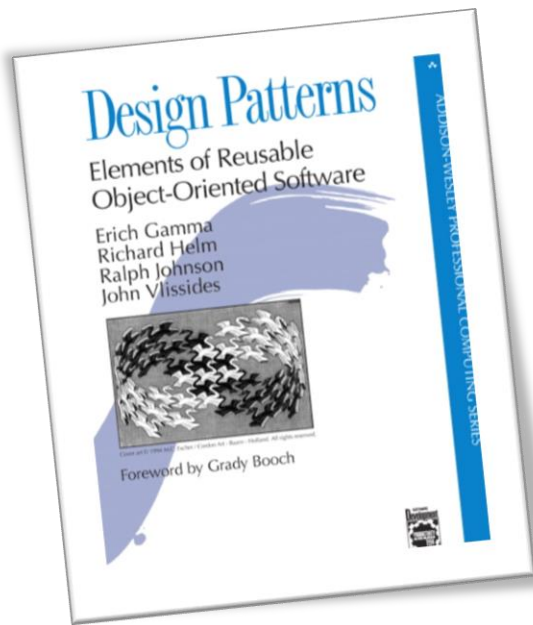
Sterotypical architecture



Padrão de Design de Software

Each pattern describes a **problem that occurs over and over** again in our environment and then describes the **core of the solution** to that problem in such a way that you can **use this solution a million times over without ever doing it the same way twice.**



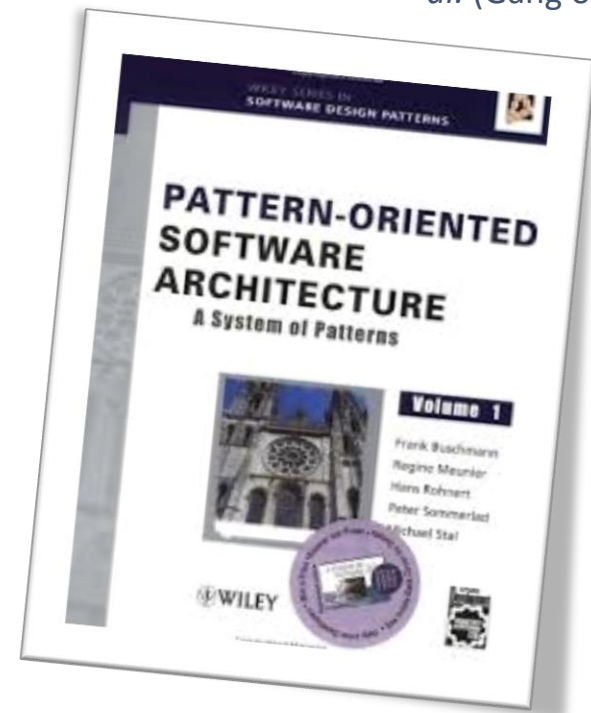


“A Software Design Pattern **names, abstracts, and identifies the key aspects** of a common **design structure** that make it useful for creating a **reusable** object-oriented design.”

Design Patterns-Elements of Reusable Object-oriented Software, Gamma *et al.* (Gang of Four) 1995

“A particular **recurring design problem** that arises in specific design **contexts**, and presents a **well-proven generic scheme** for its solution. The solution scheme is specified by describing its **constituent components**, their **responsibilities** and **relationships**, and the ways in which they **collaborate**”

Pattern Oriented Software Architecture, volume 1
Bushman *et al.* 1996



O que não é um padrão

- Um padrão **não** é uma receita milagrosa que resolve todos os problemas da aplicação ou empresa



imagem: British Medical Journal

O que é um padrão?

- Um padrão (*pattern*) é um conjunto de boas práticas (*best-practices*)
- Representa uma solução tipificada para um problema usual num dado contexto
- “Não reinventar a roda”
- Facilita a comunicação ao criar um vocabulário comum
- Os padrões descobrem-se, não se inventam

Vantagens de padrões

- Facilita a comunicação
- Descreve solução tipificada e “provada”
- Facilita a reutilização em larga-escala
- Capturam conhecimento dos peritos e *trade-offs* efectuados
- São maioritariamente agnósticos
 - Tecnologia
 - Linguagem de programação
 - Filosofia (*open source*, ...)

Desvantagens de padrões

- Não permite reutilização directa de código
- A equipa pode sofrer de *pattern overload*
- São validados através da experiência e da discussão em vez de testes automáticos
- A integração de padrões no processo de desenvolvimento de software é uma tarefa intensa do ponto de vista humano

Princípios OO

Os Padrões GRASP

- Information Expert
- Low Coupling
- High Cohesion
- Creator
- Controller
- Polymorphism
- Indirection
- Pure Fabrication
- Protected Variations

SOLID

- Single Responsibility Principle
- Open/Close
- Liskov Substitution Principle
- Interface Segregation
- Dependency Inversion

Questões comuns

- A que classe atribuir uma dada responsabilidade?
- Como organizar as responsabilidades do sistema?
- Quem deve ter responsabilidade de coordenar a interação de caso de uso?
- Quem deve ter a responsabilidade de representar e implementar a lógica de negócio?
- Como gerir o ciclo de vida de um objeto?
 - Criar um objeto?
 - persistir objetos?
- Como proteger o código para modificação?

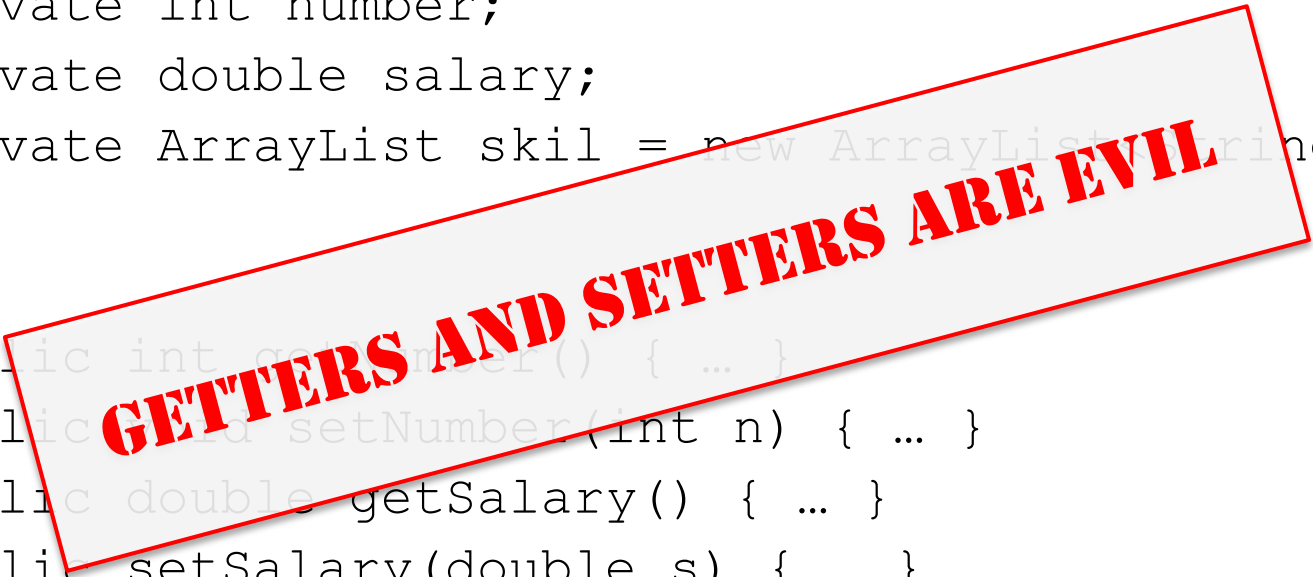
**A que classe atribuir uma dada
responsabilidade?**

Information Expert

- Problema:
 - qual é o princípio geral para a atribuição de responsabilidades aos objetos?
- Solução:
 - Atribuir a responsabilidade ao information expert
 - a classe que contém a informação necessária para desempenhar essa responsabilidade

What's wrong with this code?

```
class Employee {  
    private int number;  
    private double salary;  
    private ArrayList skill = new ArrayList<String>;  
  
    ...  
    public int getNumber() { ... }  
    public void setNumber(int n) { ... }  
    public double getSalary() { ... }  
    public void setSalary(double s) { ... }  
    public ArrayList getSkills() { ... }  
    public void setSkills(ArrayList s) { ... }  
}
```



Information Hiding

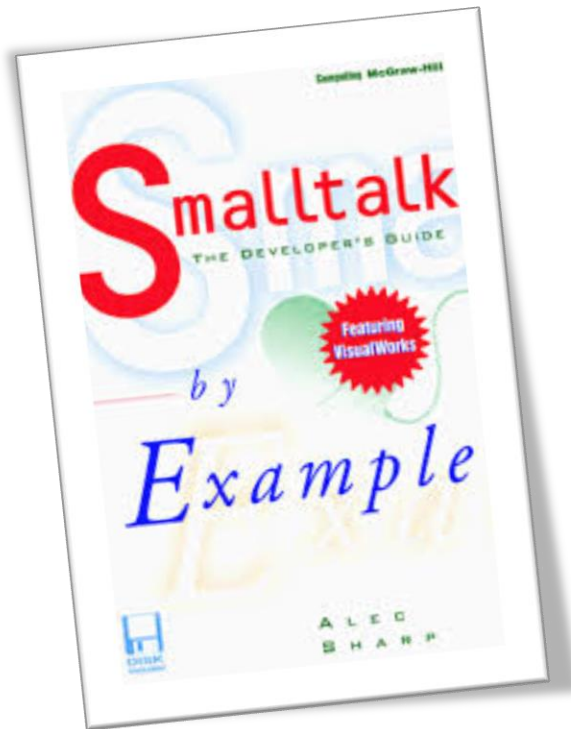
Segregation of the design decisions that are most likely to change, thus protecting other parts from extensive modification if the design decision is changed.

Anemic Domain Model

Procedural thinking where Objects are just Plain data structures devoid of any behaviour, and behaviour is concentrated in "service" and "controller" classes.



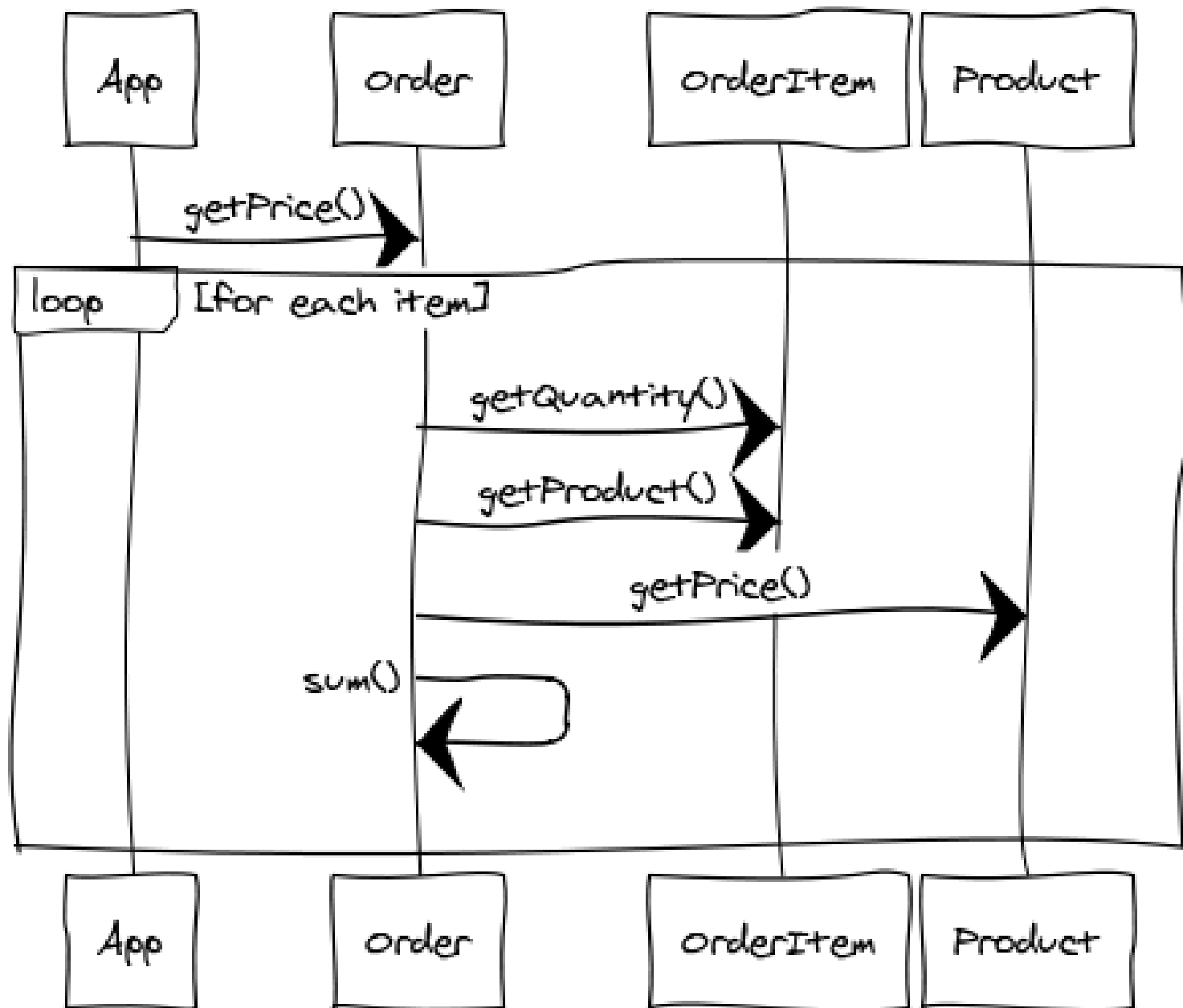
Tell, don't ask



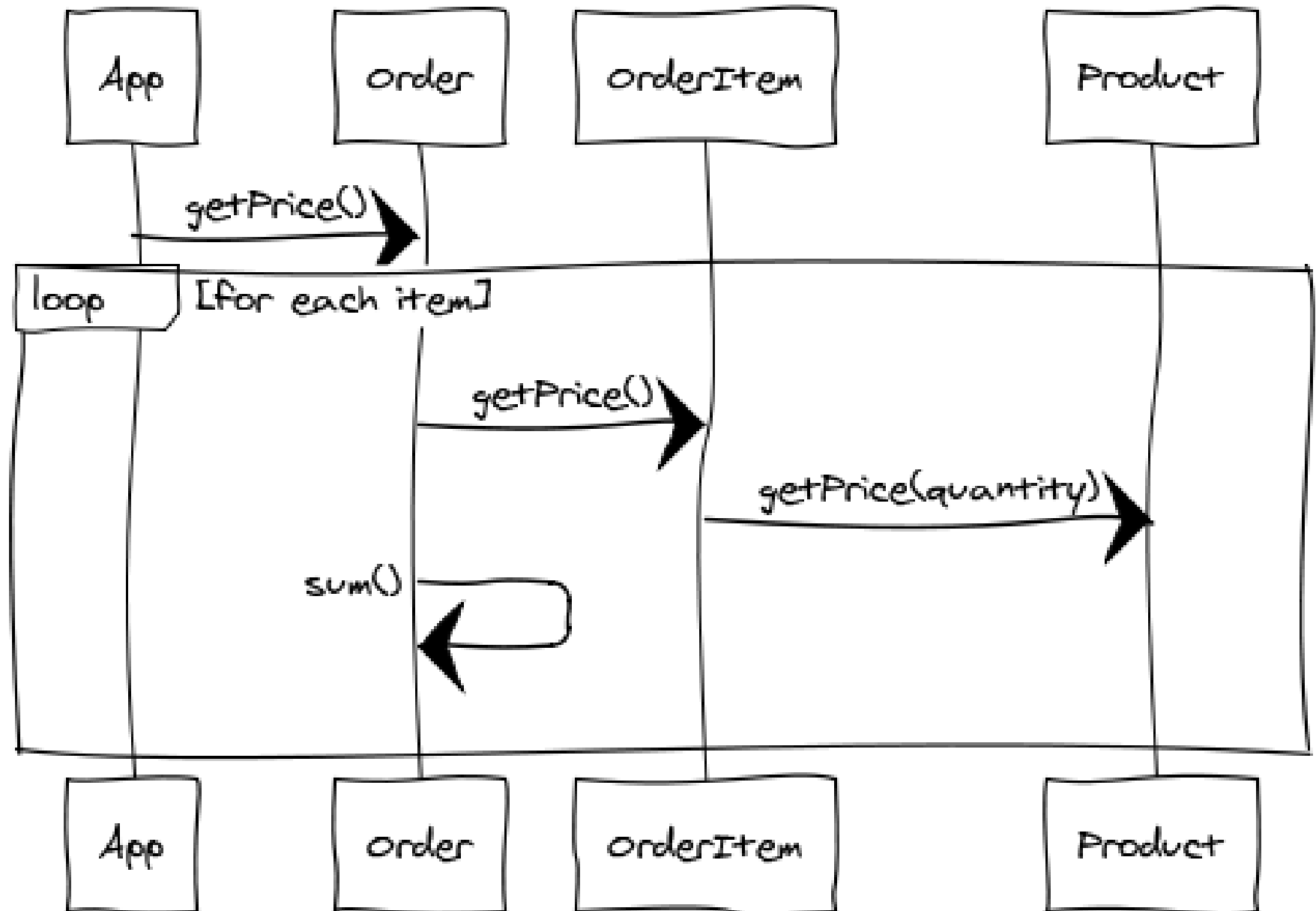
Procedural code gets information then makes decisions.
Object-oriented code tells objects to do things.

— Alec Sharp

calculating the total for an order (procedural)



calculating the total for an order (Object orientation)



Replace “set” syntax with strong names

- `setFirstName(string fn)`
- `setLastName(string ln)`

vs.

- `changeName(
 string first,
 string last)`

- `setStatus(STATUS st)`
- `getStatus()`

vs.

- `activate()`
- `deactivate()`
- `isActive()`

Intention Revealing Interface

Name classes and operations to describe their effect and purpose, without reference to the means by which they do what they promise.

Intention Revealing Interface

```
interface ISapService {  
    double getHourlyRate(int sapID);  
}
```

VS.

```
interface IPayrollService {  
    double getHourlyRate(Employee e);  
}
```

Single Responsibility Principle

A class should have only one reason to change.



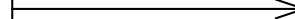
Single Responsibility Principle

Person
first: string last:string street: string zip: string email:string
changeName(first, last) changeAddress(street, zip)



Person
first: string last:string email:string
changeName(first, last) changeAddress(address)

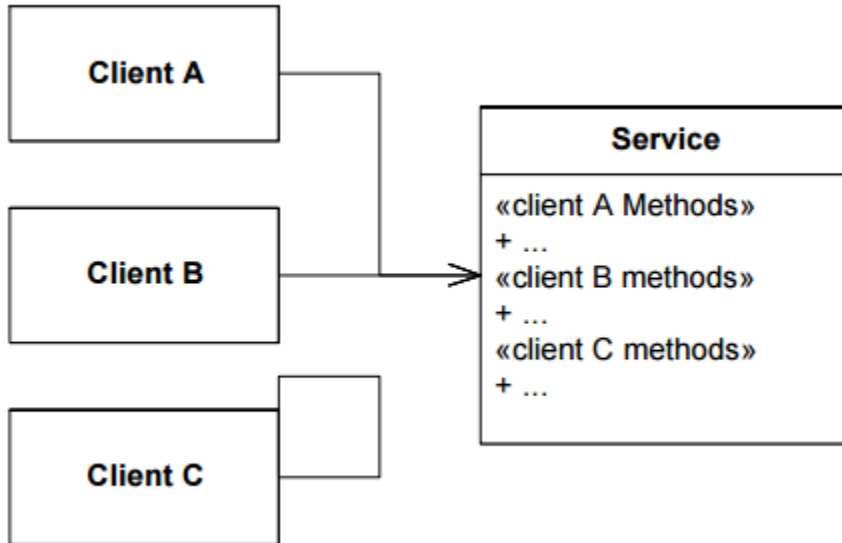
Address
street: string zip: string
changeAddress(street, zip)



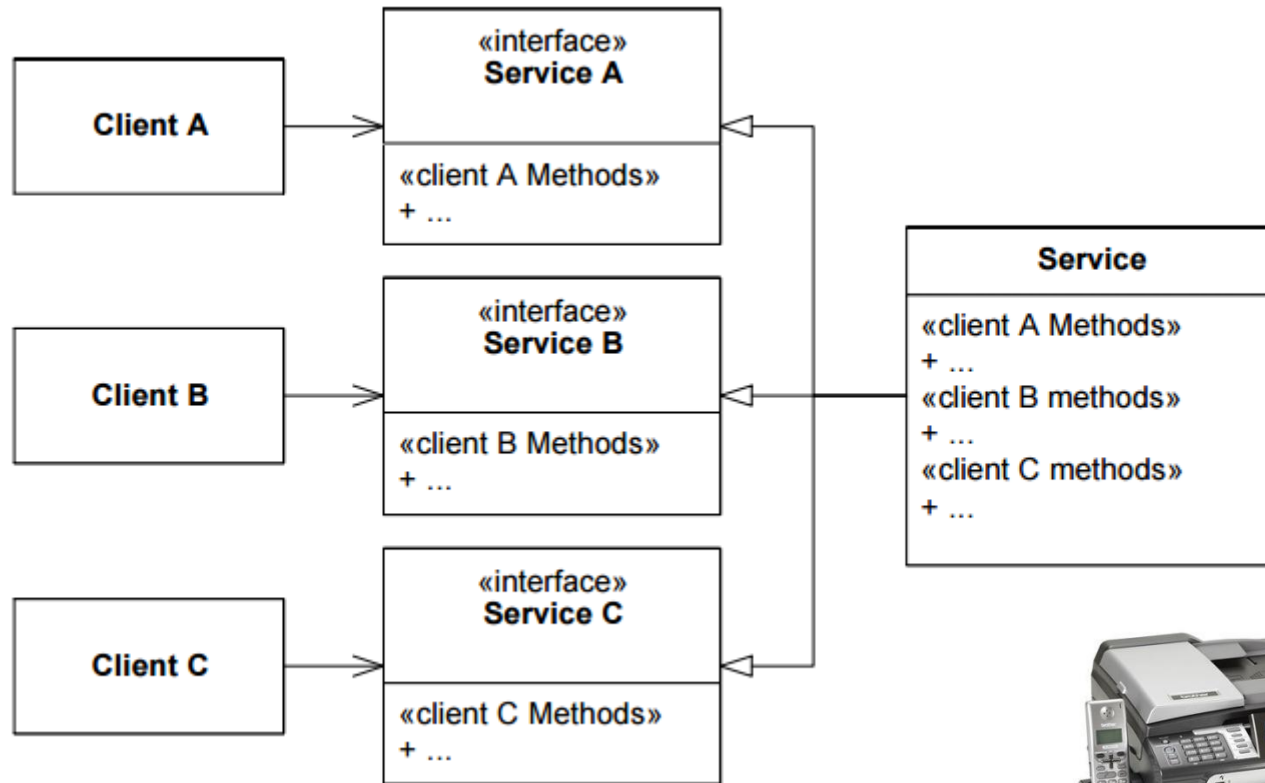
Interface Segregation Principle

Many client specific interfaces are better than one general purpose interface.

Interface Segregation Principle



Interface Segregation Principle



Topic	Principles and patterns
Which class should a responsibility be assigned to?	Information Expert Tell, don't ask Single Responsibility Principle Interface Segregation Principle Intention Revealing Interfaces
How to organize the system's responsibilities?	
How to model the domain?	
How to handle an object's lifecycle?	
How to prepare the code for modification?	

Bibliografia

- Why getters and setters are Evil. Allan Holub.
<http://www.javaworld.com/article/2073723/core-java/why-getter-and-setter-methods-are-evil.html>
- Tell, don't ask. The Pragmatic Programmers.
<https://pragprog.com/articles/tell-dont-ask>
- Design Principles and Design Patterns. Robert Martin.
http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf
- Domain Driven Design. Eric Evans. 2004
- Applying UML and Patterns; Craig Larman; (2nd ed.); 2002.