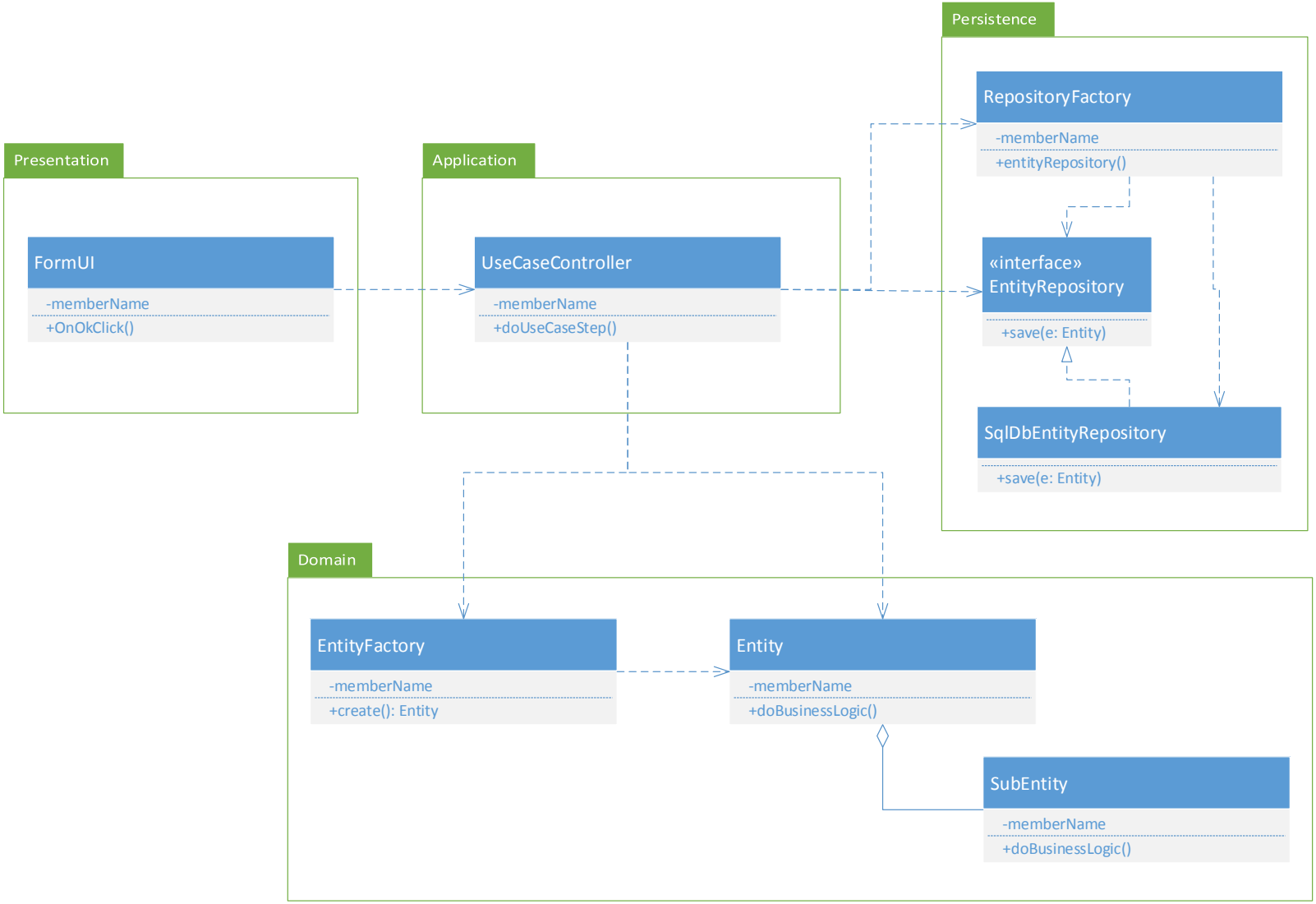
EAPLI

Princípios de Design OO: Organização de responsabilidades

Paulo Gandra de Sousa
pag@isep.ipp.pt

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?	Persistence Ignorance Entity, Value Object, Aggregate Domain Service Domain Event Observer
How to handle an object's lifecycle?	Factories Repositories
How to prepare the code for modification?	Protected Variation Open/Close Principle Dependency Inversion Principle Liskov Substitution Principle Template Method Strategy Decorator

Sterotypical architecture



Como organizar as responsabilidades do sistema?

Layers pattern

POSA

- **Problem:**
  - you are designing a system whose dominant characteristic is a mix of low- and high-level issues, where high-level operations rely on the lower-level ones.
- **Solution:**
  - Structure your system into an appropriate number of layers and place them on top of each other [each providing a specific level of abstraction].

Layers pattern

- The **layers** architectural pattern helps to structure applications that can be decomposed into groups of subtasks in which each group of subtasks is at a particular level of abstraction.

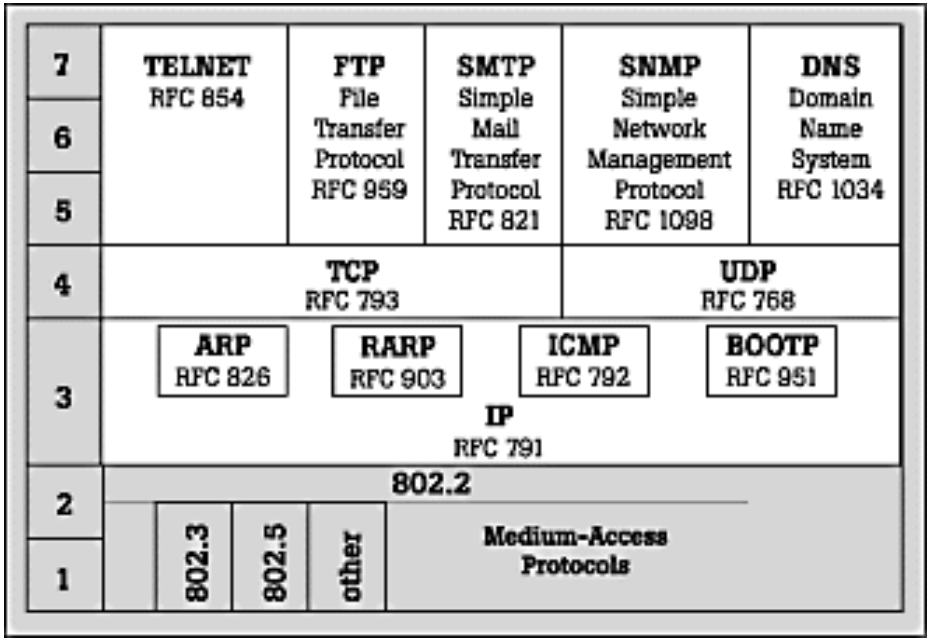


Image: http://networking.ringofsaturn.com/Protocols/sevenlayer.php

High Cohesion

GRASP

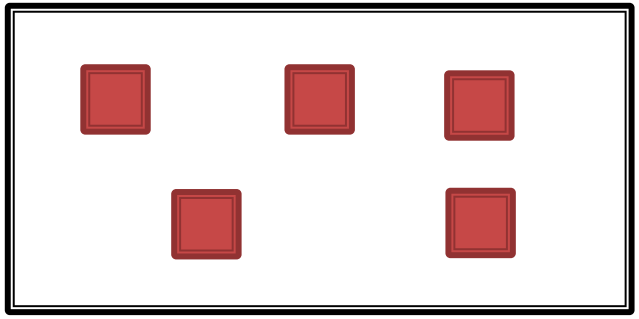
- **Problema**:
  - Como manter os objectos com funcionalidades coerentes e de fácil compreensão?
- **Solução**:
  - Atribuir a responsabilidade de modo a que a coesão e o relacionamento entre as funcionalidades seja elevado. Delegar responsabilidades.
- Coesão (Cohesion) é uma medida da coerência das responsabilidades atribuídas a um elemento.
  - Uma classe com elevada coesão tem um número relativamente pequeno de operações, intimamente relacionadas, e delegam ou colaboram com outras, para realizar tarefas mais complexas.
  - Uma classe com fraca coesão é:
    - Difícil de compreender
    - Difícil de reutilizar
    - Difícil de manter
    - Dependente de outras classes

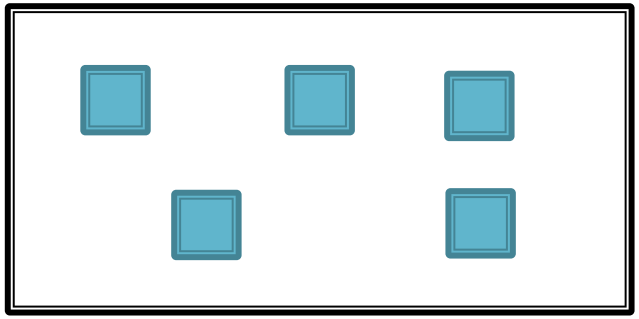
Low Coupling

GRASP

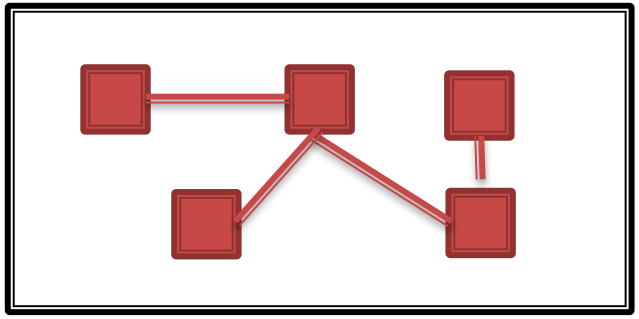
- **Problema**:
  - Como conseguir fraca dependência, baixo impacto à mudança e maximizar a reutilização ?
- **Solução**:
  - Atribuir responsabilidades de modo a manter um acoplamento fraco. Usar este padrão para avaliar alternativas.
- Coupling (Acoplamento) é uma medida do modo como um elemento está ligado, tem conhecimento ou depende de outros.
  - Uma classe com elevado acoplamento depende de outras classes:
    - Obriga a alterações por mudanças nas classes relacionadas
    - Difícil de entender isoladamente
    - Mais difícil de reutilizar

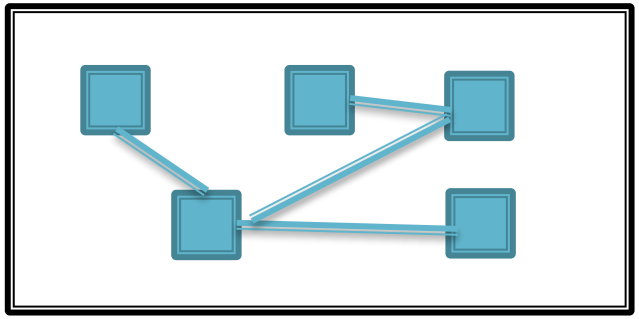
Cohesion inside layers



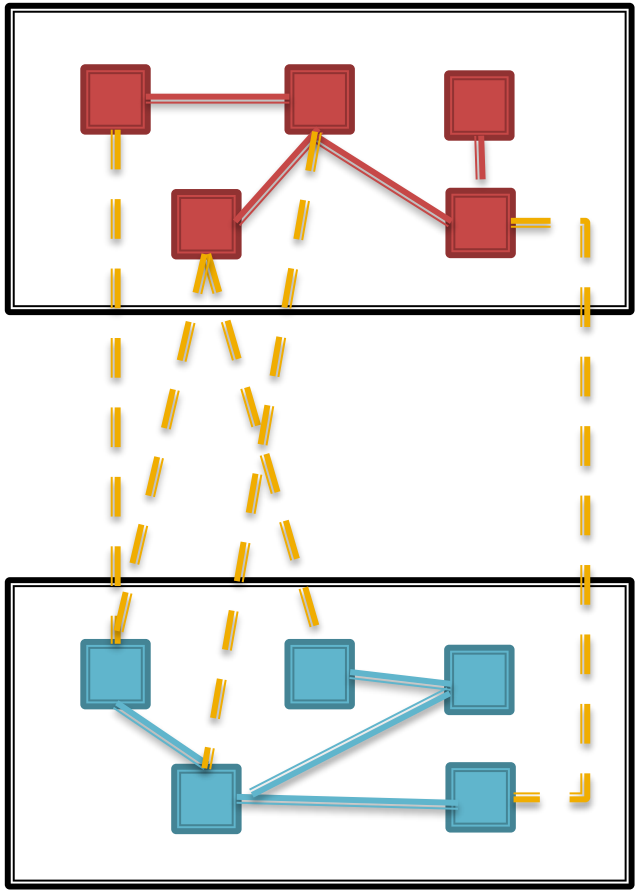


Coupling inside layers





Coupling between layers



Indirection

GRASP

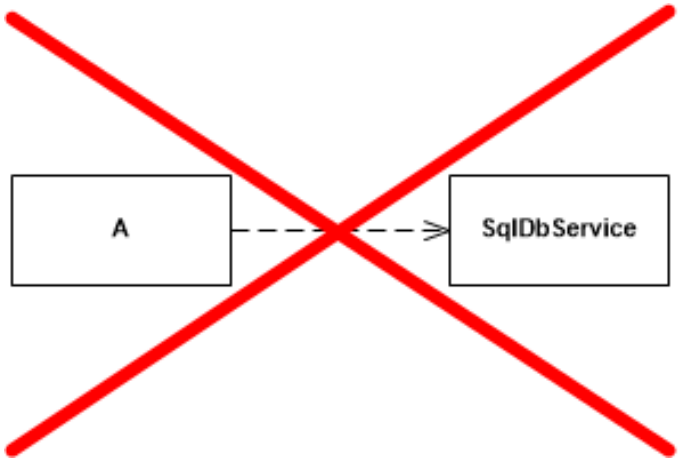
- **Problema**:
  - Como atribuir responsabilidade evitando a ligação directa entre dois objectos?
  - Como "desacoplar" dois objectos de modo a aumentar a possibilidade de reutilização?

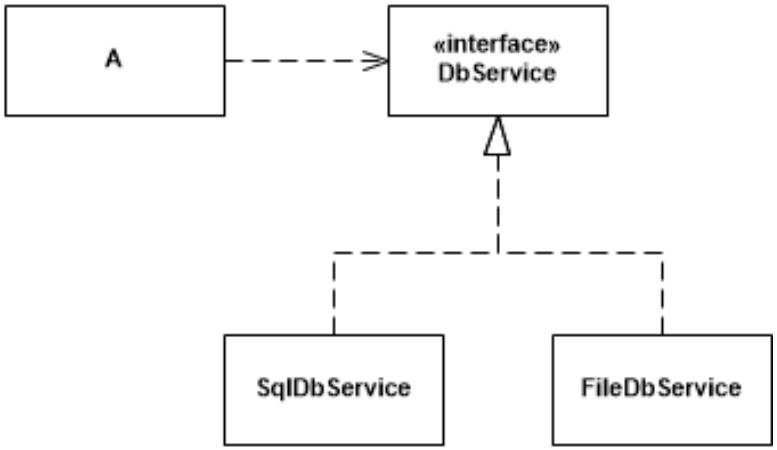
- **Solução**:
  - atribuir a responsabilidade a um objecto intermédio, que fará a mediação entre os componentes ou serviços, de modo a não estarem directamente ligados.

Dependency Inversion Principle

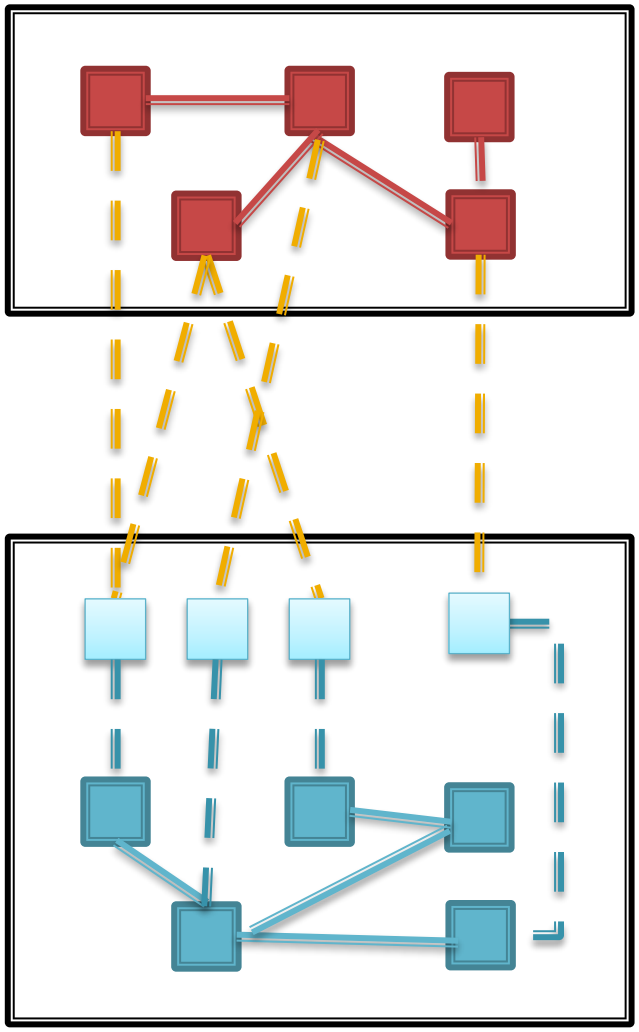
SOLID

Clients should depend on abstractions, not concretions. I.e., program to an interface not a realization.





DIP between layers



Façade

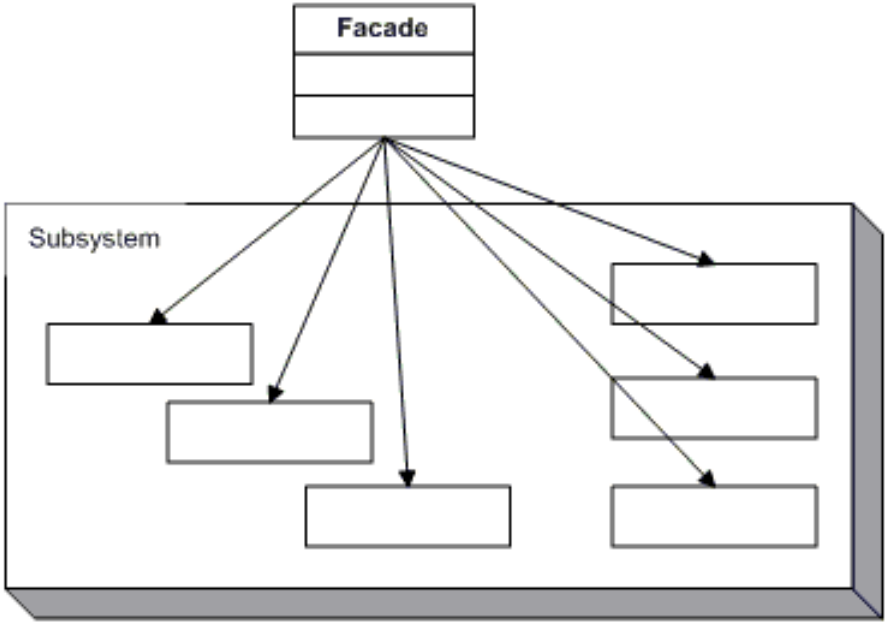
GoF

- Problema:
  - O trabalho é de facto desempenhado por mais objectos, mas este nível de complexidade deve ficar escondido do cliente.
- Solução:
  - Criação dum objecto façade (fachada) que recebe as mensagens, mas passa os comandos aos trabalhadores para os completarem

Façade

GoF

- Fornece uma interface unificada dum conjunto de interfaces num subsistema. Façade define uma interface de alto nível que torna o subsistema mais simples de usar.

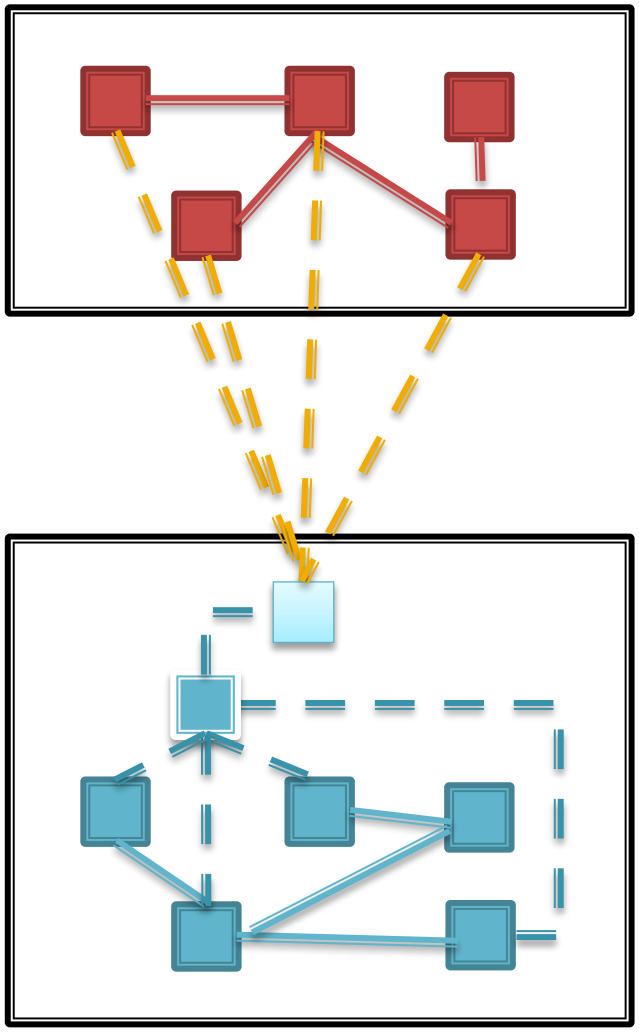


fonte: Design Patterns: Elements of Reusable Object-Oriented Software

Exemplo

```
public class OrderFacade
{
   public void MakeOrder(Order o, ...)
   {
     OrderManagement om = new OrderManagement();
     om.StoreOrder(o, ...);

     Accounting acc = new Accounting();
     acc.Register(o, ...);

     Billing b = new Billing();
     b.GenerateBillFromOrder(o);

     WarehouseHandling h = new WarehouseHandling();
     h.PrepareForShipping(o);

     InventoryManagement im = new InventoryManagement();
     im.TakeStock(o);

     DeliveryAgent da = new DeliveryFactory.CreateDeliveryAgent(o);
     da.ScheduleDelivery(o);
   }
}
```

DIP + Facade between layers



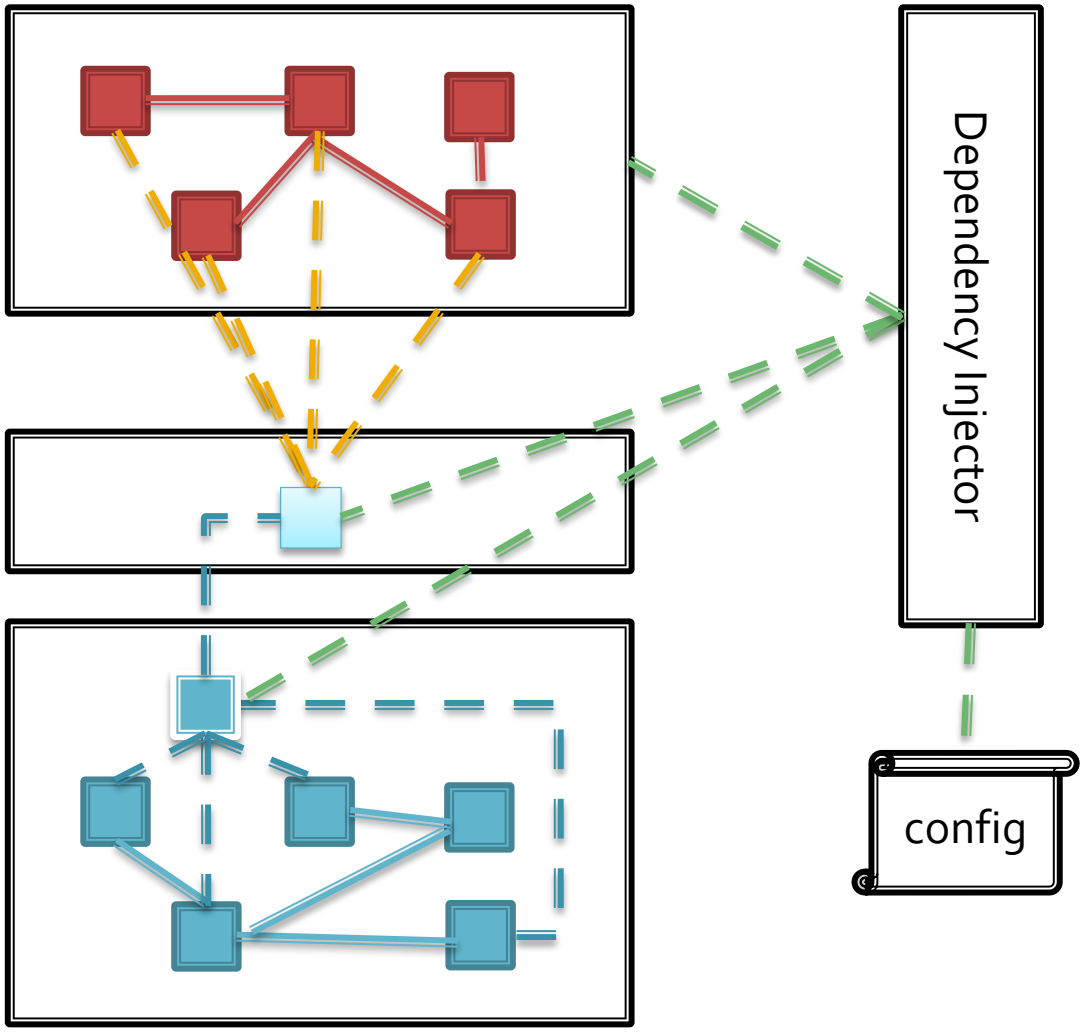
Dependency Injection

- Modules declare their dependencies but do not create them explicitily

Dependency Injection Containers

- Frameworks que facilitam a substituição de módulos:
  - Low coupling
  - Dependency Inversion
  - Factory
  - Registry
  - Strategy
  - Decorator
- E.g., Spring Framework http://Spring.io

Dependency Injection



Quem deve coordenar a interação de caso de uso?

Controller

GRASP

- **Problema:**
  - Quem deve ser responsável por responder a um evento de entrada no sistema, gerado na User Interface?
- **Solução:**
  - Criar uma API para a camada de domínio que permite responder aos casos de uso da aplicação através da coordenação de outros objetos.
    - Facade Controller: o sistema, um dispositivo ou um sub-sistema
    - Use Case Controller: um caso de uso

Model-View-Controller

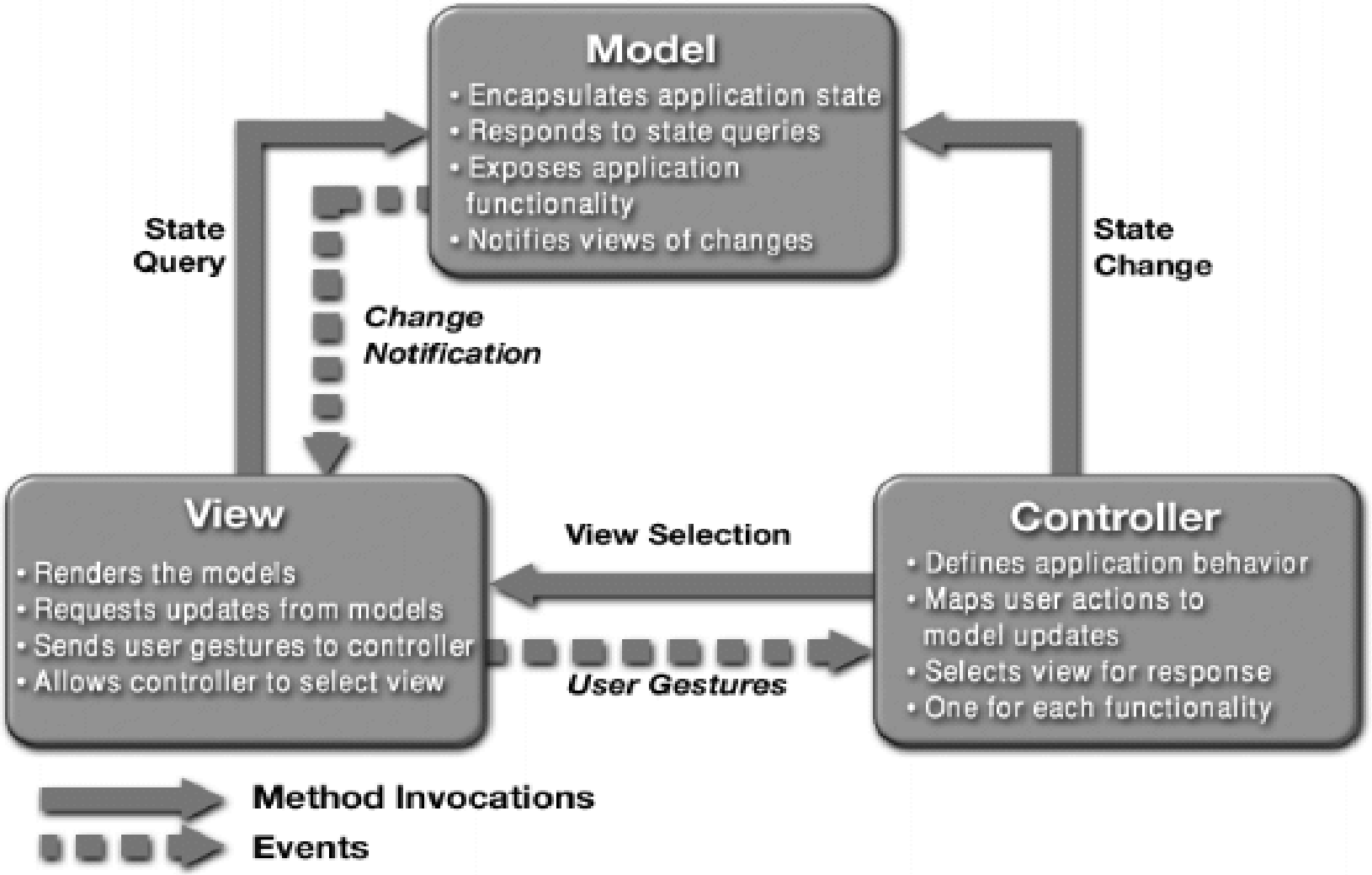
GoF

- Problem:
  - How to handle change requests to your UI while keeping the processing elements stable?
- Solution:
  - Divide an interactive application into the three areas: processing, output, and input.

Model-View-Controller

GoF

- The Model-View-Controller architectural pattern (MVC) divides an interactive application into three components.
  - The model contains the core functionality and data.
  - Views display information to the user.
  - Controllers handle user input.
- Views and controllers together comprise the user interface. A change-propagation mechanism ensures consistency between the user interface and the model.



GRASP Controller vs MVC Controller

- Different roles
- GRASP
  - Application facade/API
  - Controller is part of the Application layer
- MVC
  - Handling of user input
  - Controller is part of the Presentation layer

Sumário

Sumário

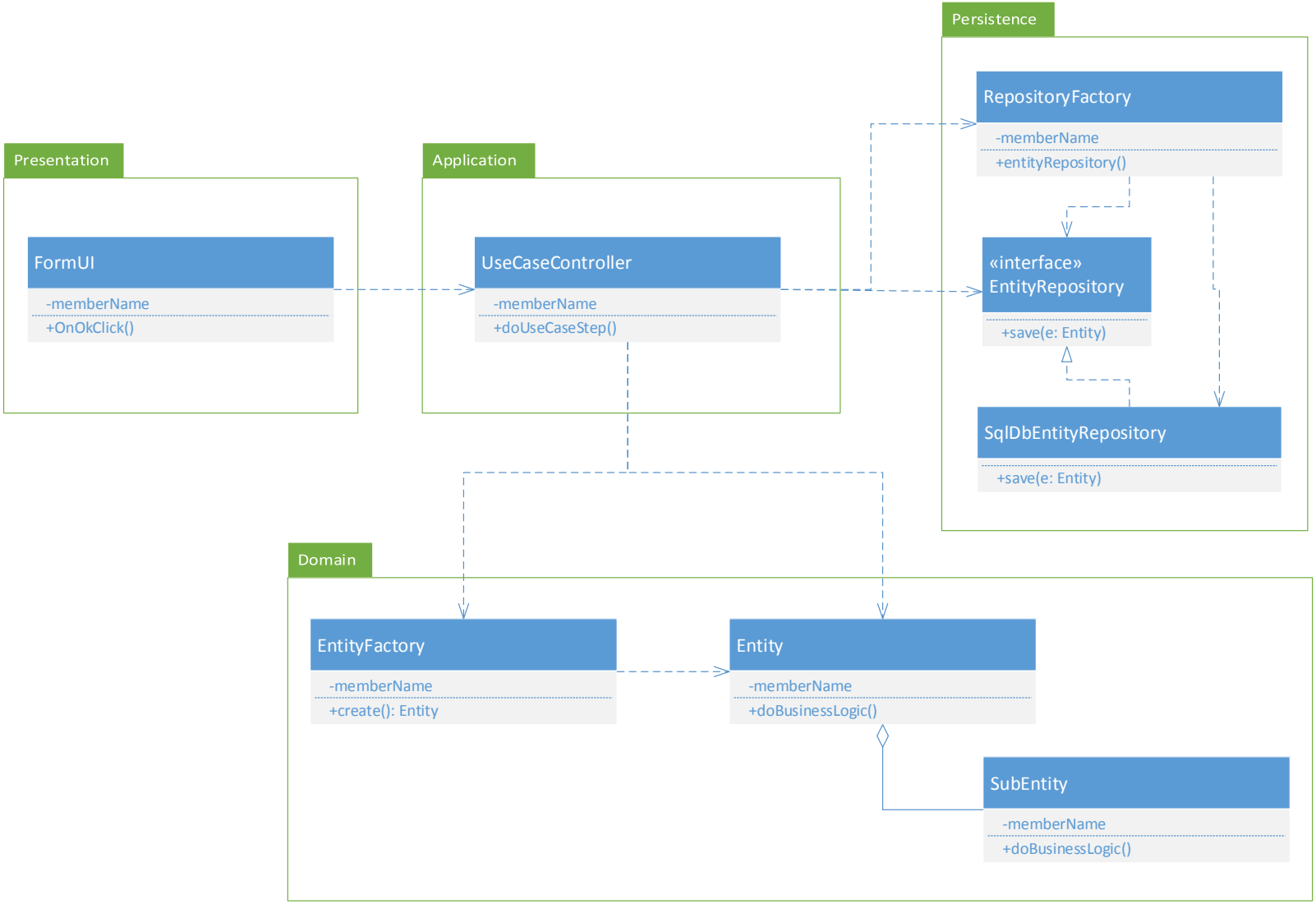
- Atribuição de responsabilidades deve obedecer a princípios consolidados

- GRASP
- SOLID
- DDD
- GoF

- Para promover :
  - Modularidade
  - Reutilização
  - Manutenção

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?	**Layers,** **High Cohesion, Low Coupling** **Facade** **Controller**
How to model the domain?	Persistence Ignorance Entity, Value Object, Aggregate Domain Service Domain Event Observer
How to handle an object's lifecycle?	Factories Repositories
How to prepare the code for modification?	Protected Variation Open/Close Principle Dependency Inversion Principle Liskov Substitution Principle Template Method Strategy Decorator

Sterotypical architecture



Bibliografia

- Domain Driven Design. Eric Evans
- Design Patterns-Elements of Reusable Object-oriented Software, Gamma et al. (Gang of Four)
- Design Principles and Design Patterns. Robert Martin. http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf
- Applying UML and Patterns; Craig Larman; (2nd ed.); 2002.
- Inversion of Control Containers and the Dependency Injection Pattern. Martin Fowler. http://martinfowler.com/articles/injection.html