Paulo Gandra Sousa
pag@isep.ipp.pt

Arquitetura e organização

"A BIG BALL OF MUD is haphazardly structured, sprawling, sloppy, duct-tape and bailing wire, spaghetti code jungle."

Brian Foote and Joseph Yoder, 1997, "Big Ball of Mud", PLoP

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**.

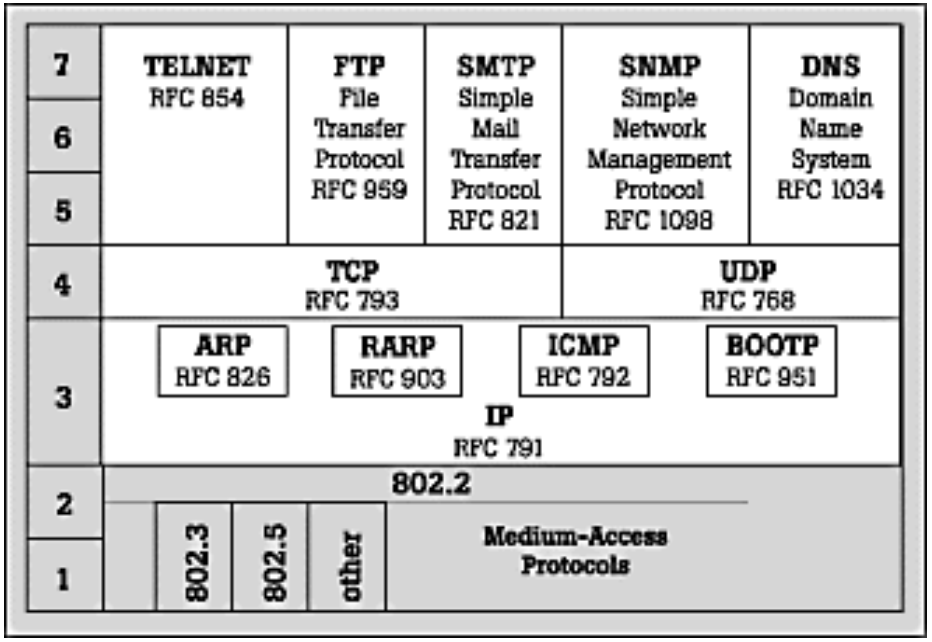
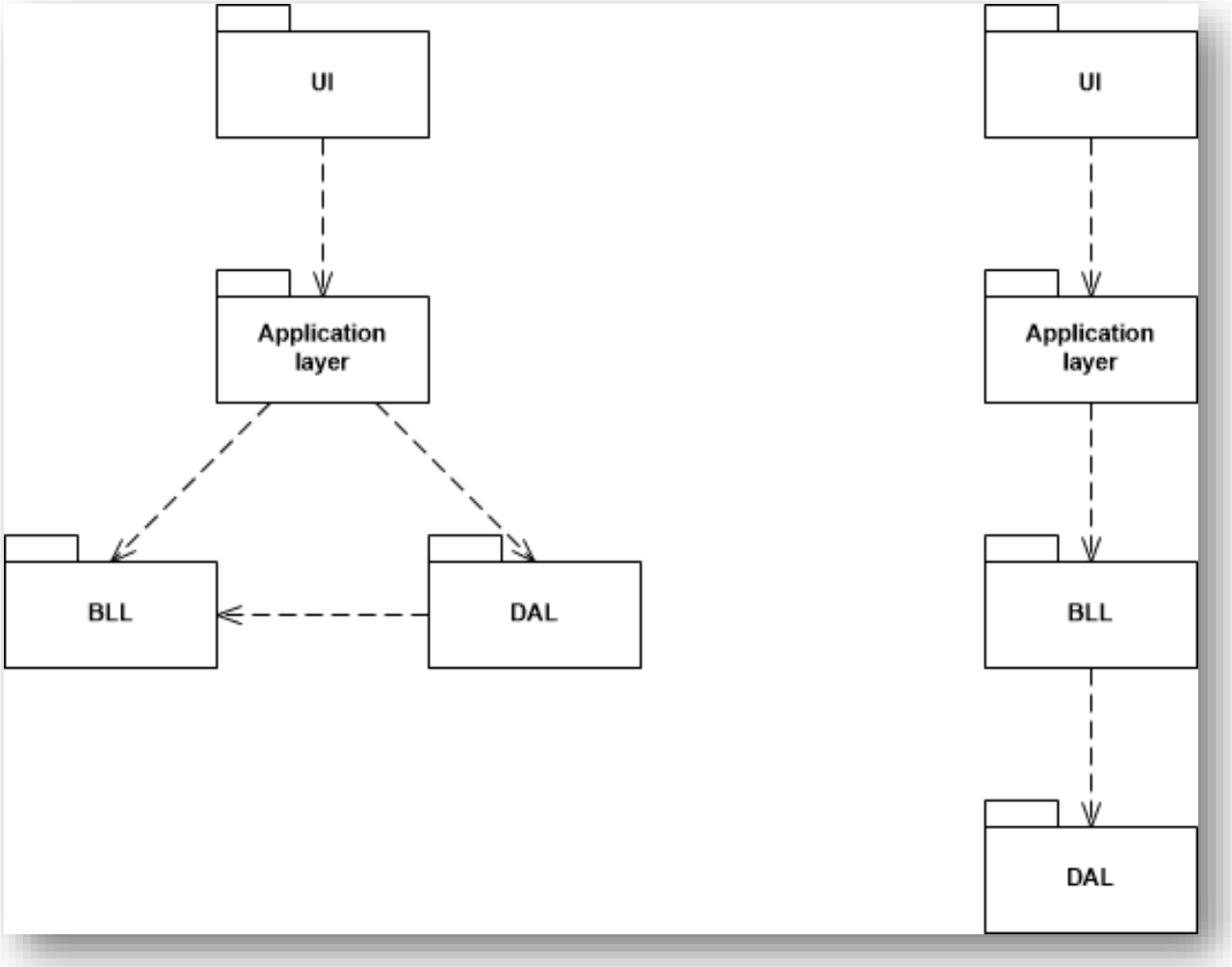


Image: http://networking.ringofsaturn.com/Protocols/sevenlayer.php

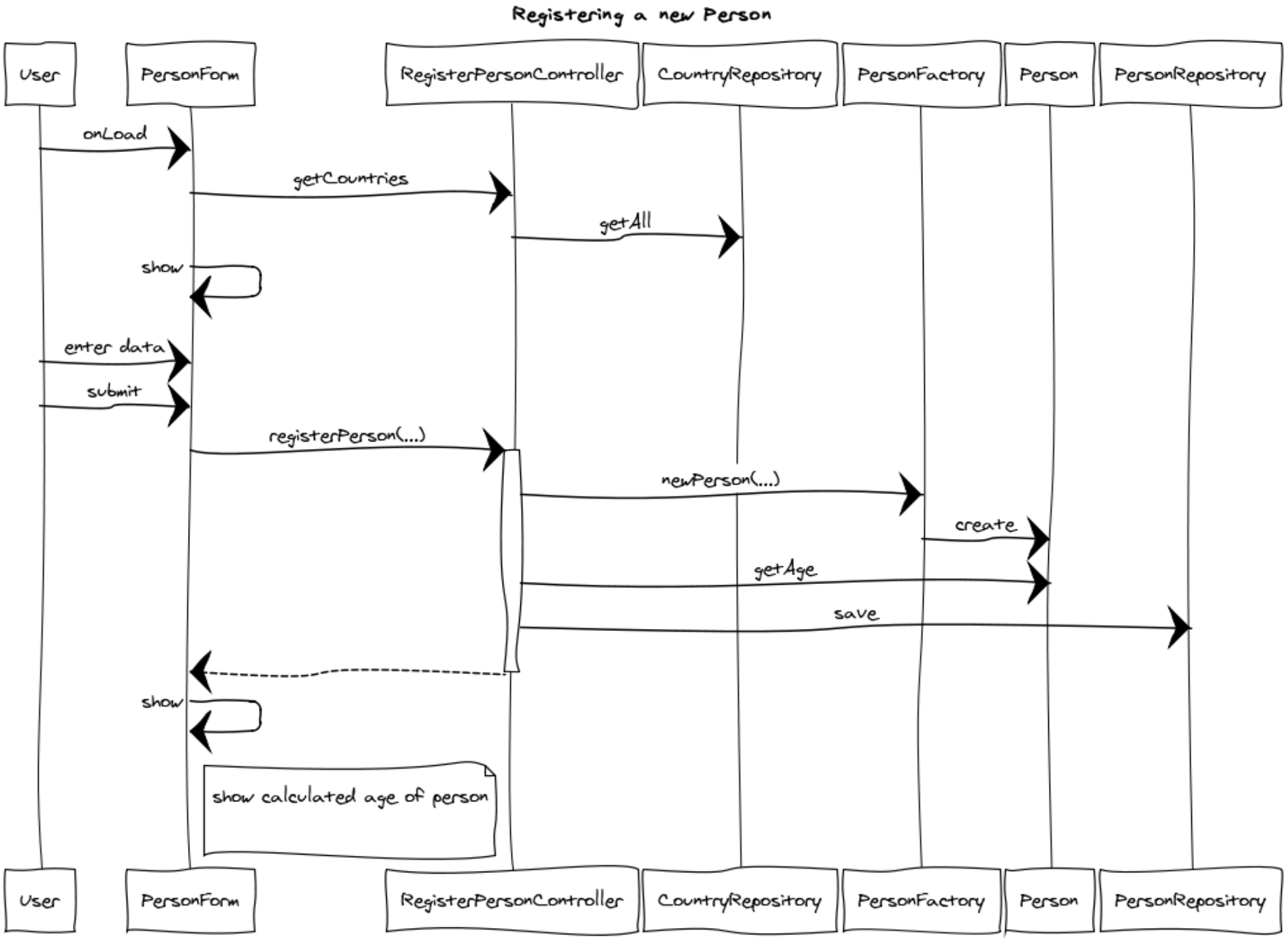
Estruturas típicas

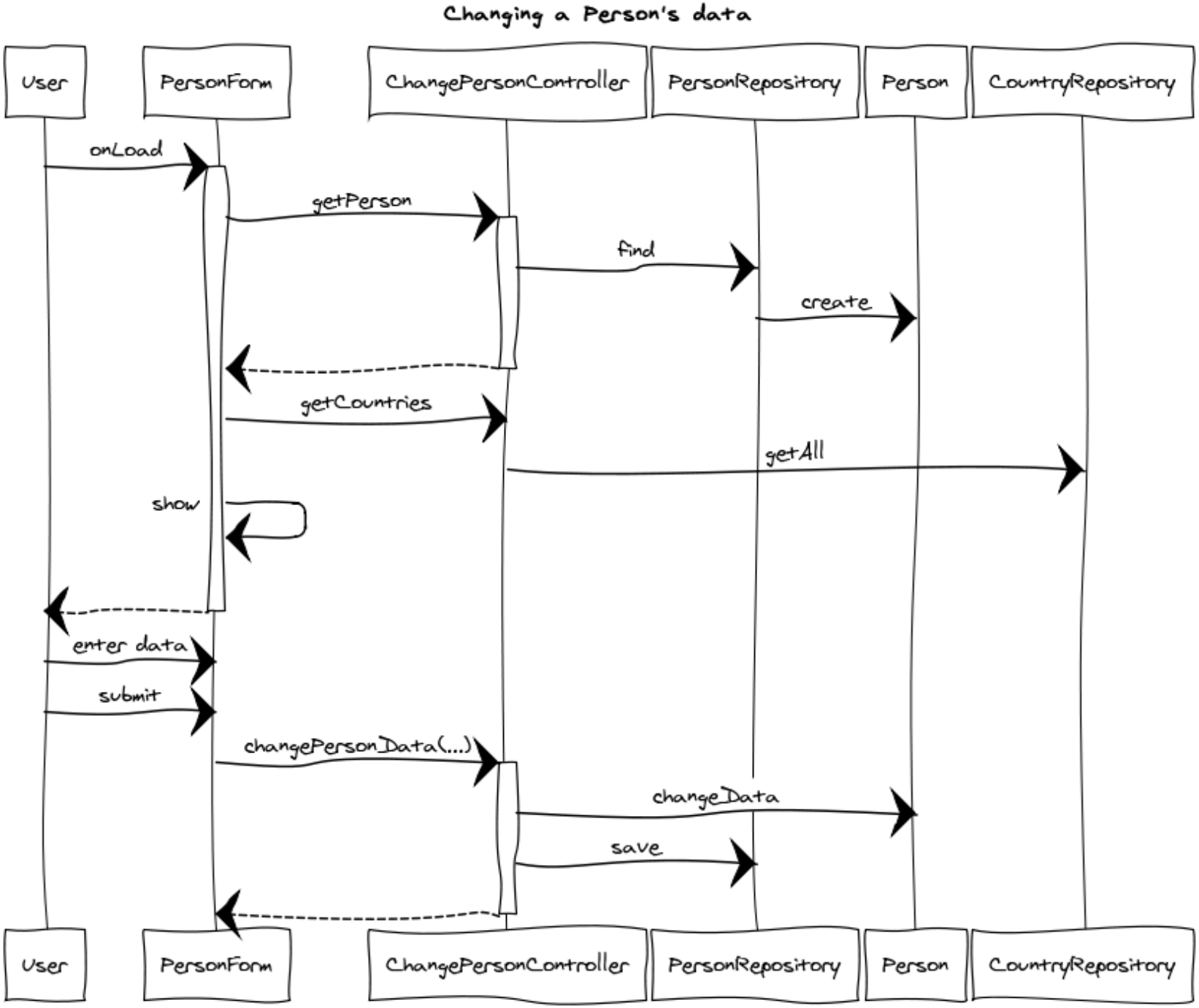


http://www.codingthearchitecture.com/2015/03/08/package_by_component_and_architecturally_aligned_testing.html

An example

- Contact information application for person's data, e.g., name, phone number, country of birth, etc.
- Use case 1: register a new contact
  - After selecting the country of birth from the list of known countries, the user enters the remainder person's data and the system saves the new person showing the age of the person
- Use case 2: change a person's data
  - After entering the person's id, the system shows the person's data to the user which may change the desired information. The system updates the person's data.



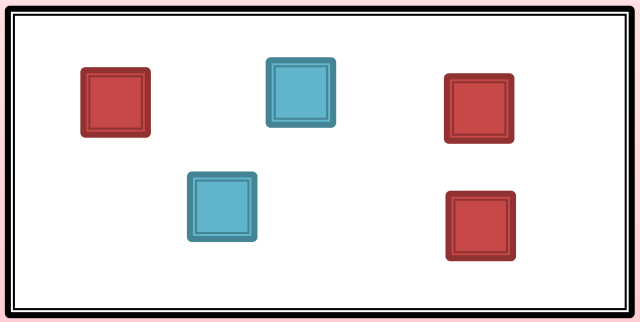


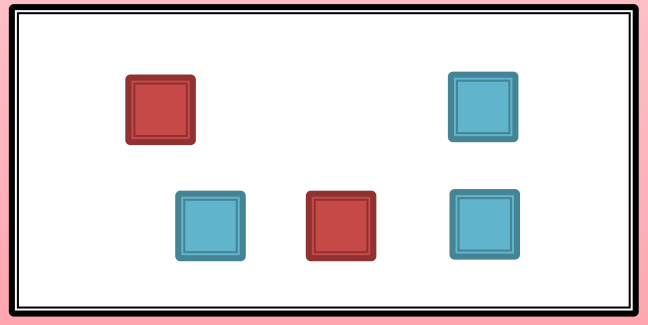
Organização de responsabilidades

- Módulos (packages) com responsabilidades bem definidas
  - High cohesion
  - Low coupling
  - Information Expert

GRASP

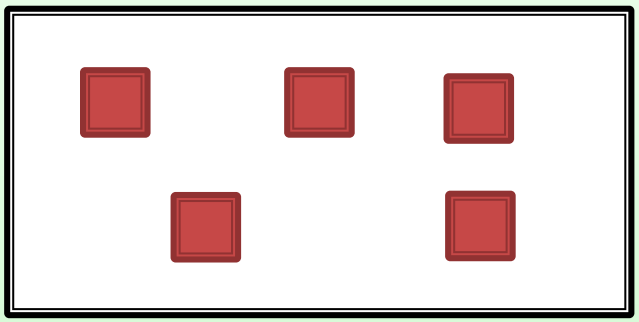
Cohesion

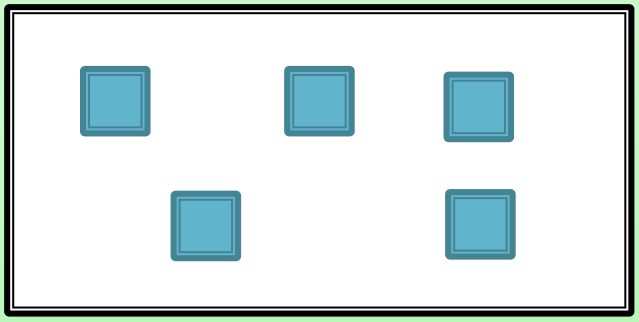




Big ball of mud

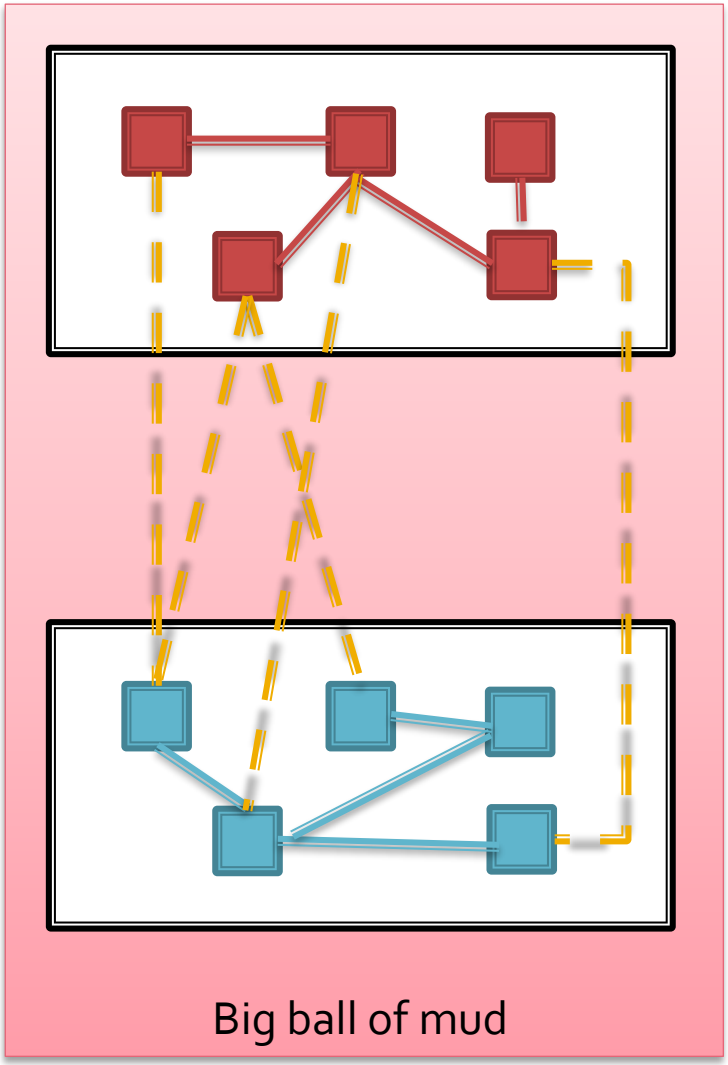
vs.





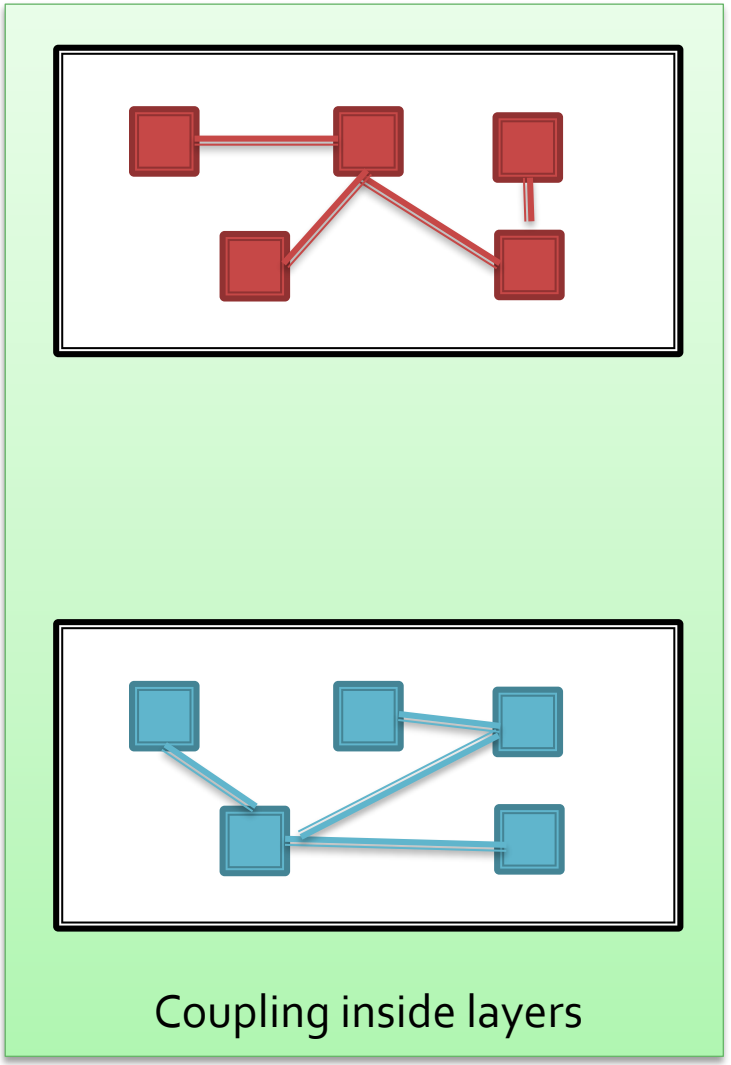
Cohesion inside layers

Coupling



Big ball of mud

vs.



Coupling inside layers

Indirection

GRASP

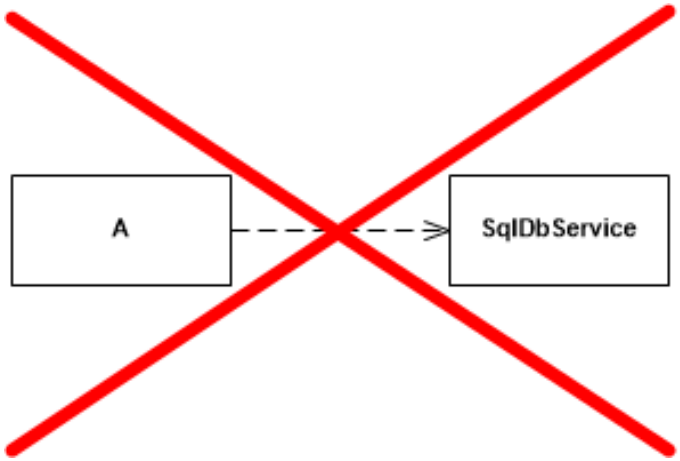
- **Problema**:
  - Como atribuir responsabilidade evitando a ligação directa entre dois objectos?
  - Como "desacoplar" dois objectos de modo a aumentar a possibilidade de reutilização?

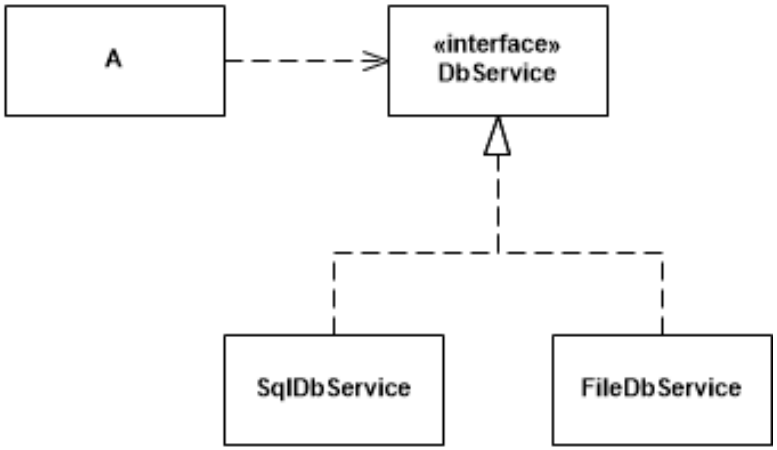
- **Solução**:
  - atribuir a responsabilidade a um objecto intermédio, que fará a mediação entre os componentes ou serviços, de modo a não estarem directamente ligados.

Dependency Inversion Principle

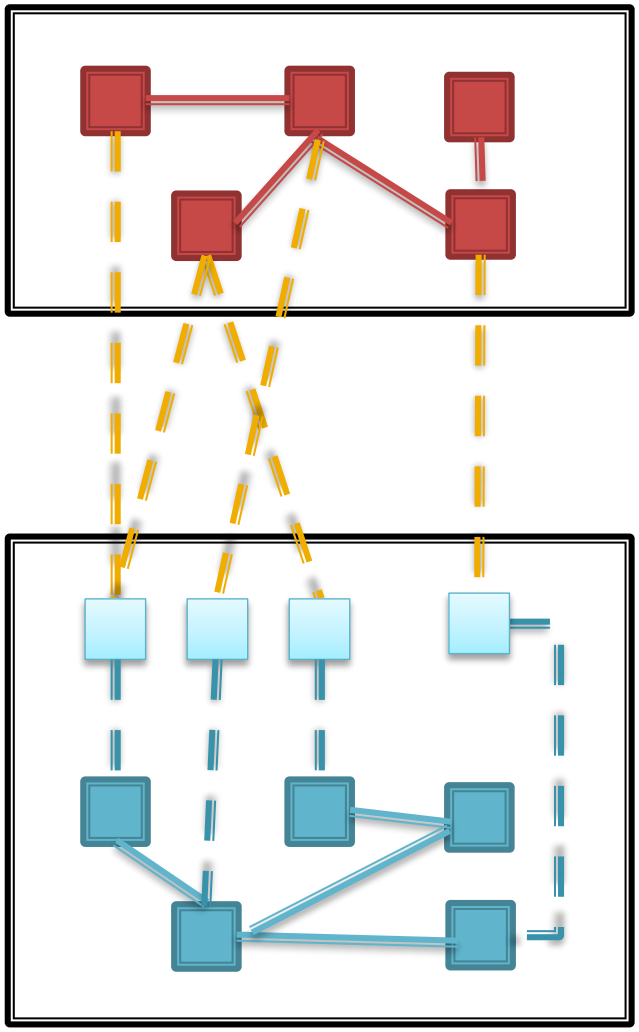
SOLID

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





DIP between layers



Façade

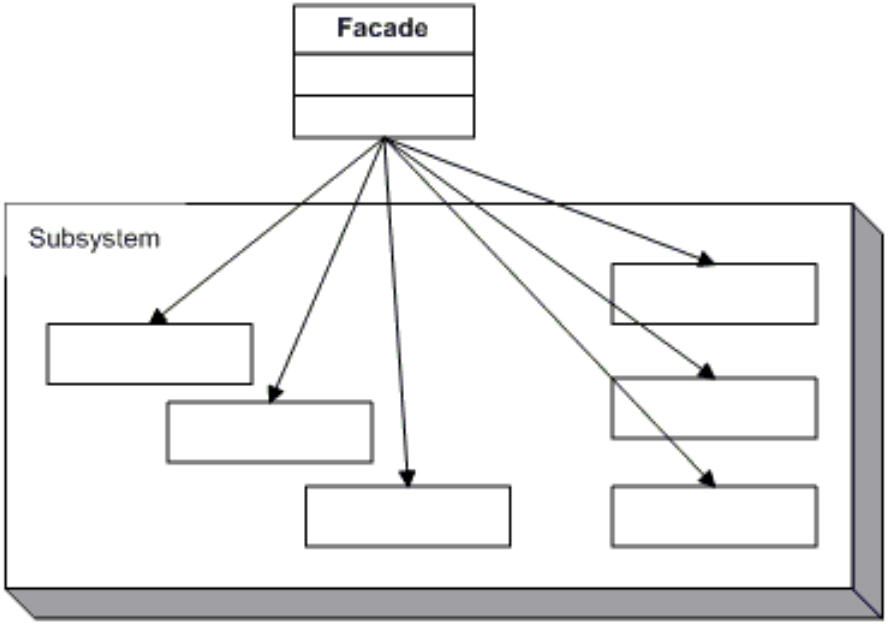
GoF

- 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.

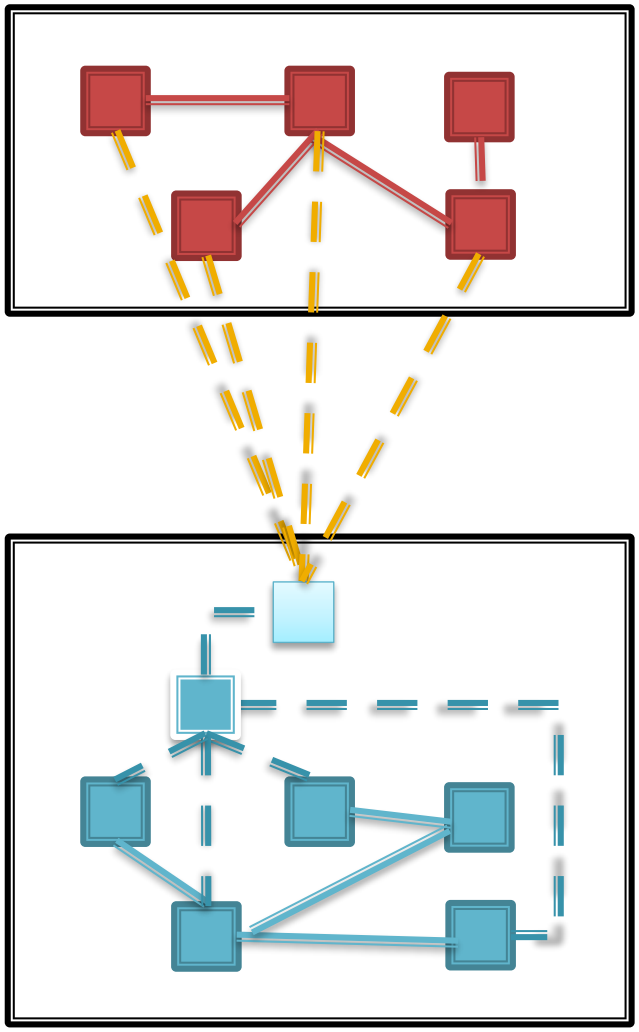


fonte: Design Patterns: Elements of Reusable Object-Oriented Software

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



These classes are **not public**

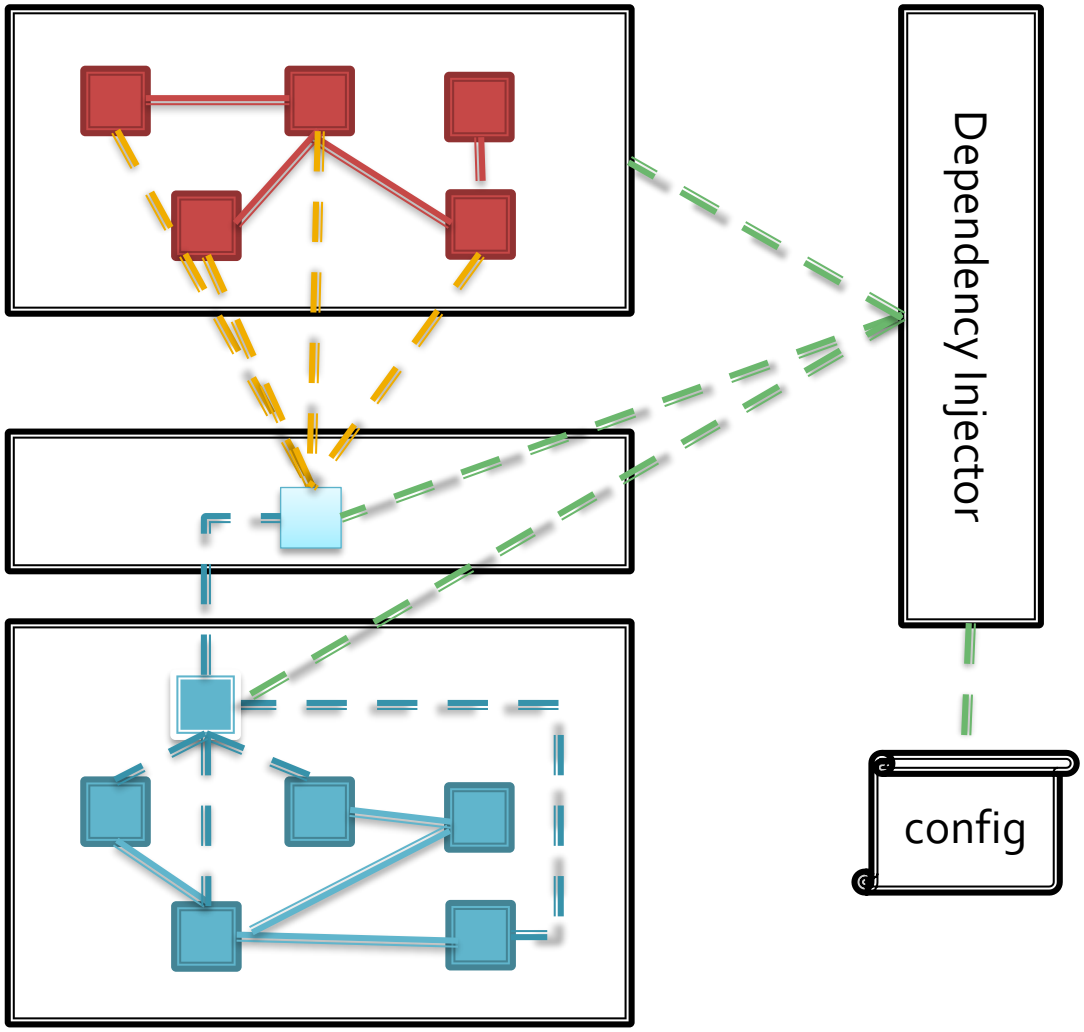
Dependency Injection

- Modules declare their dependencies but do not create them explicitily

```
class Component {
        NeededService svc;

        Component() {
                svc = new ServiceImplementation();
        }
}
```

```
class Component {
        NeededService svc;

        Component(NeededService impl) {
                svc = impl;
        }
}
```

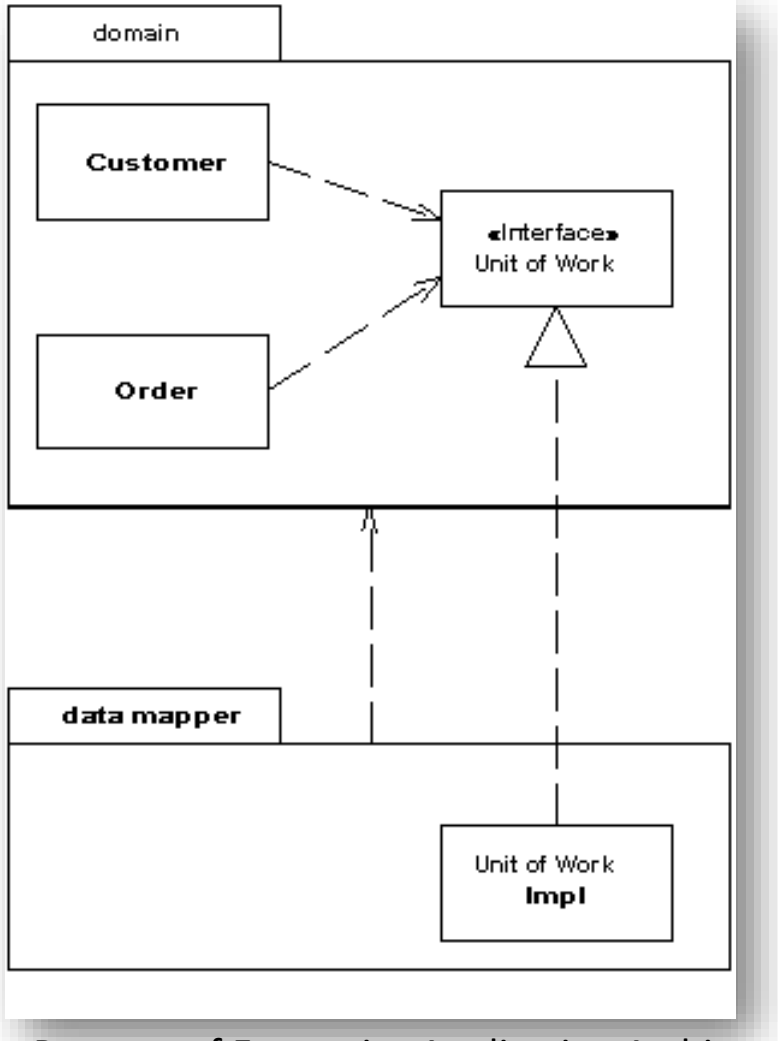
Dependency Injection



Separated Interface

Defines an interface in a separate package from its implementation.

The interface is owned by the client and not by the provider



source: Patterns of Enterprise Application Architecture

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

```
class Component {
    NeededService svc;

    Component(NeededService impl) {
        svc = impl;
    }

    ...
}
```

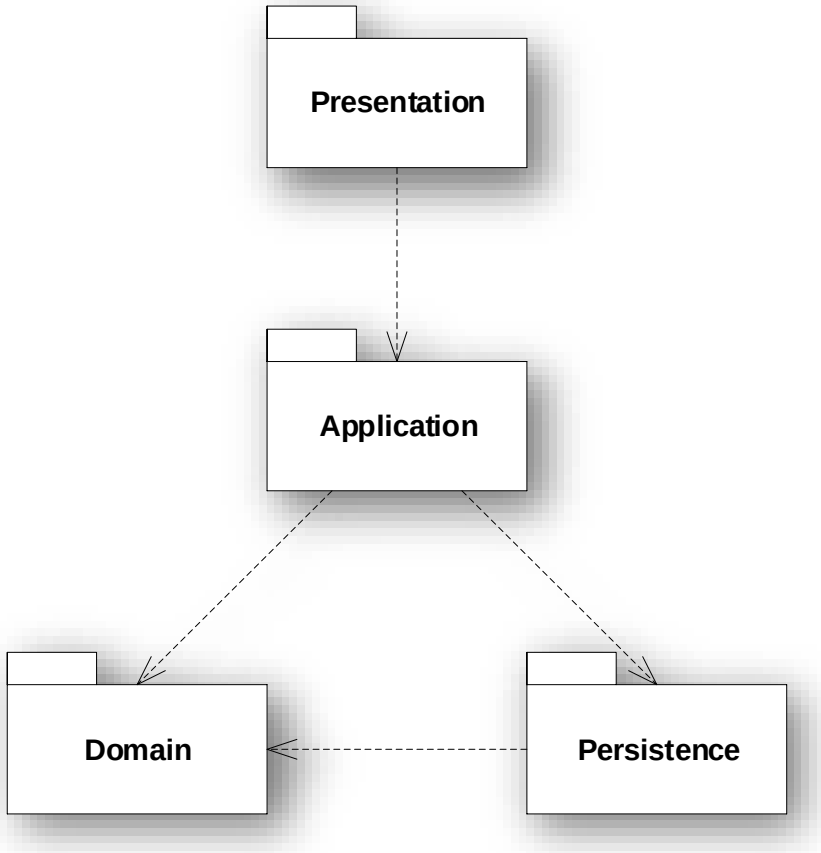
```
class ServiceImplementation
    implements NeededService {

    ...
}
```

```
class MainApp{

    static void run(…) {
        ApplicationContext ctx = new ClassPathXmlApplicationContext("conf.xml");
        Component c = (Component) ctx.getBean("comp");

        c.doStuff();
    }
}
```

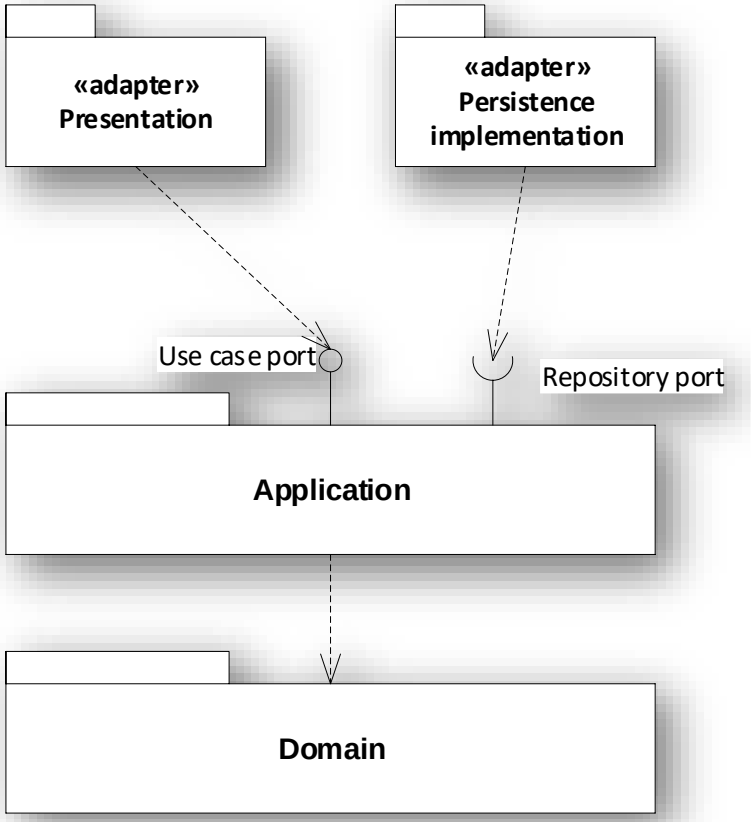
```
<beans …>
    <bean id="comp" class="Component">
        <constructor-arg value-ref="impl" />
    </bean>
    <bean id="impl" class ="ServiceImplementation/>
</beans>
```

Escaping the layer trap

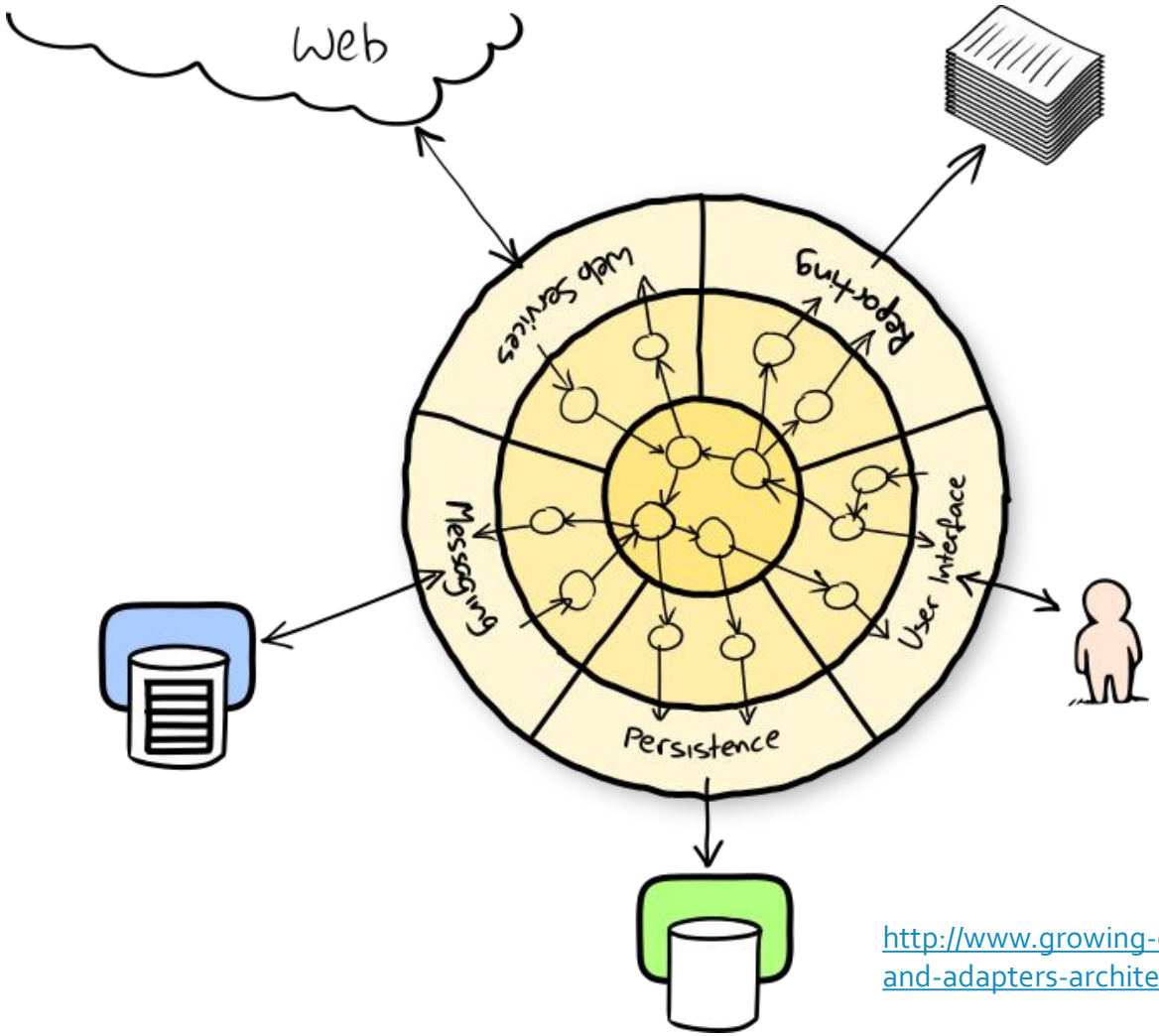
LAYERS



PORTS & ADAPTERS

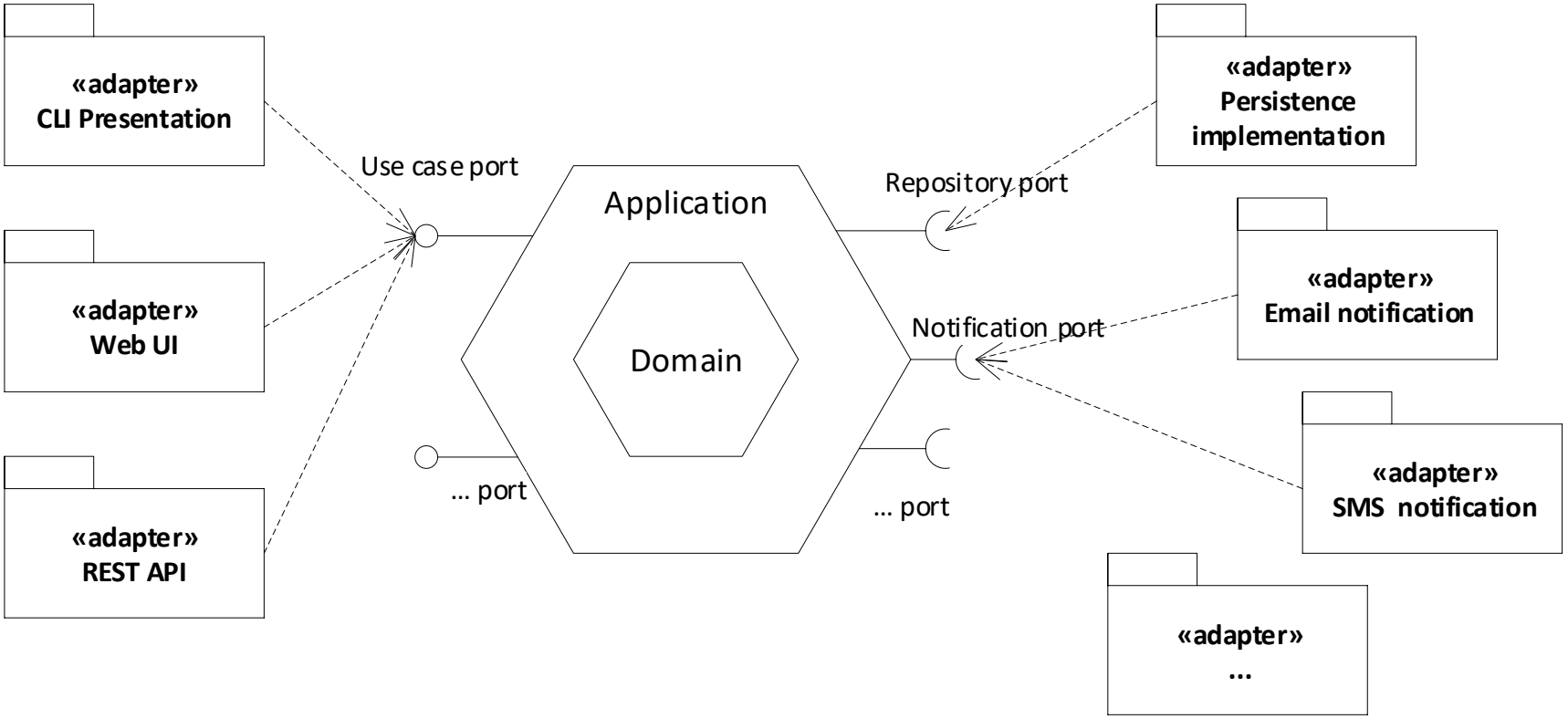


Ports & Connectors / Hexagonal Architecture

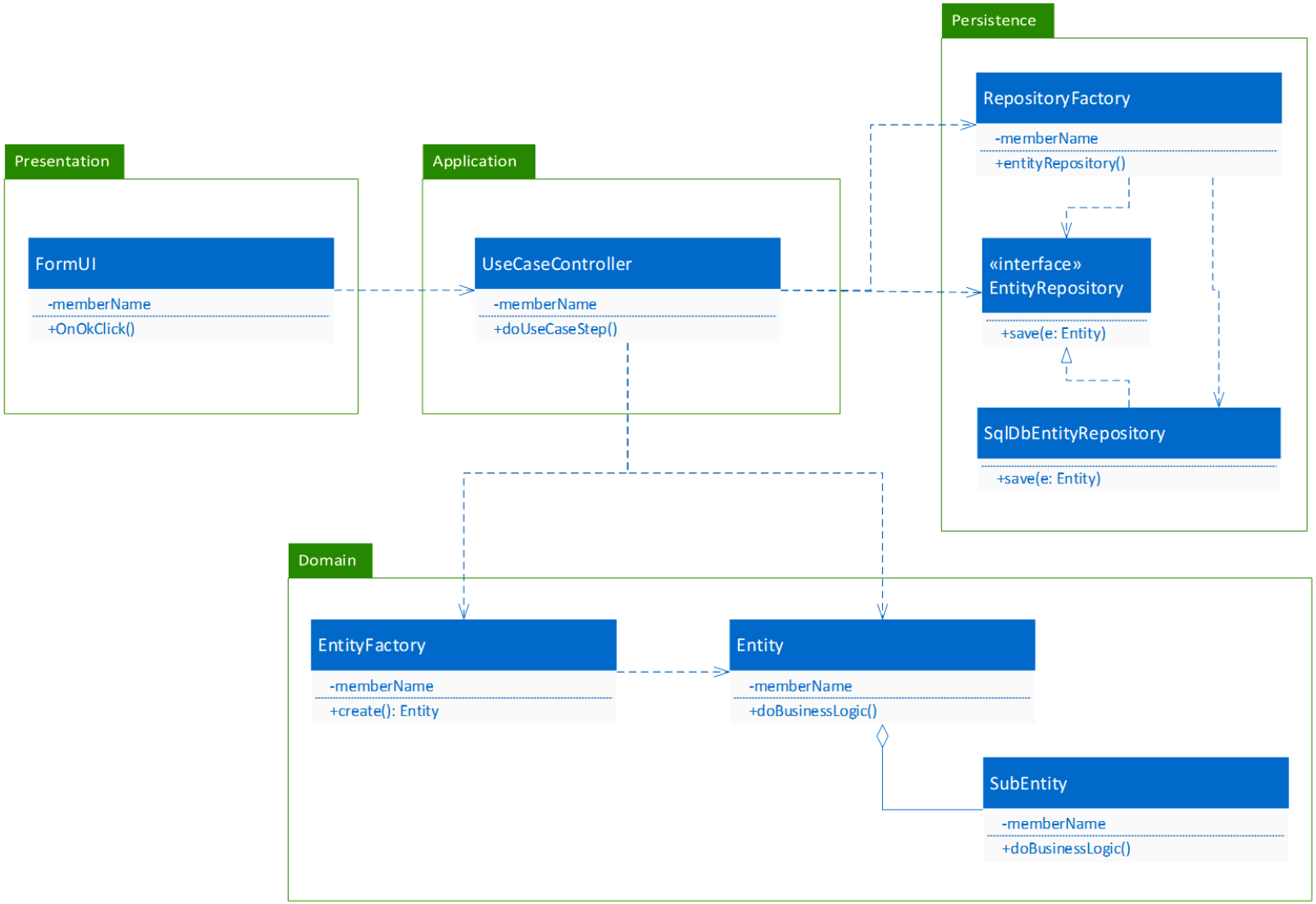


- Core domain
- Application
- Port
- Connector

http://www.growing-object-oriented-software.com/figures/ports-and-adapters-architecture.svg


«adapter»
CLI Presentation
«adapter»
Web UI
«adapter»
REST API
Use case port
... port
Application
Domain
Repository port
Notification port
... port
«adapter»
Persistence
implementation
«adapter»
Email notification
«adapter»
SMS notification
«adapter»
...

Layers, Repositories, Factories



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 Módulos/packages Information Expert High cohesion/low coupling
How to model the domain?	**Persistence Ignorance** **Entity** **Value Object** **Domain Service** **Aggregate** **Domain Event** **Observer**
How to handle an object's lifecycle?	Factory Repository
How to handle extension and variation?	Protected Variation Dependency Inversion Principle

Bibliografia

- Brian Foote and Joseph Yoder, 1997, "Big Ball of Mud", PLoP
- 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.

Bibliografia

- Simon Brown. Package By Component, http://www.codingthearchitecture.com/2015/03/08/package_by_component_and_architecturally_aligned_testing.html
- Package by feature, not layer. http://www.javapractices.com/topic/TopicAction.do?Id=205
- Simon Brown. Layers, Features and Components. http://www.codingthearchitecture.com/2016/04/25/layers_hexagons_features_and_components.html
- Thomas Pierrain (2016). Hexagonal != layers. http://tpierrain.blogspot.com/2016/04/hexagonal-layers.html
- Herberto Graça (2017) DDD, Hexagonal, Onion, Clean, CQRS, … How I put it all together. https://medium.com/the-software-architecture-chronicles/ddd-hexagonal-onion-clean-cqrs-how-i-put-it-all-together-f2590c0aa7f6