Desarrollo de aplicaciones

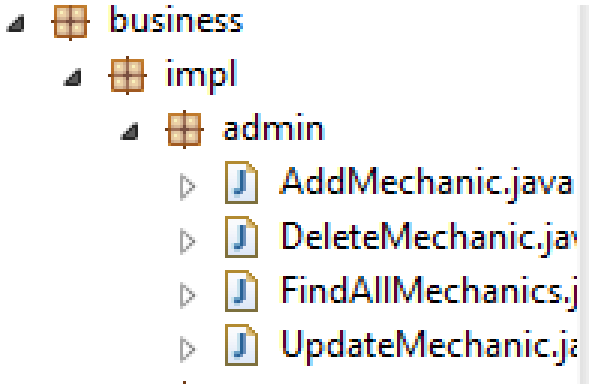
Repositorios de Información

Introducción

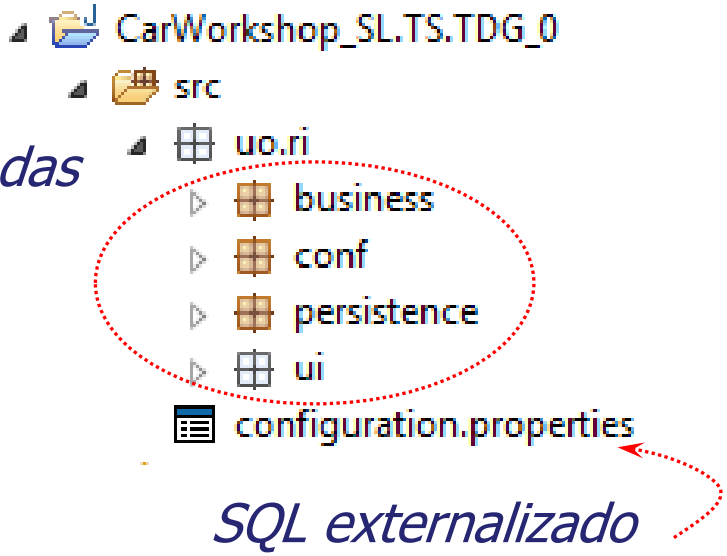
Vamos a adaptar el diseño de SL.TS.TDG_0

Arquitectura:

- Capas separadas
- Factorías entre capas
- SQL externalizado
- Transaction Script para la lógica
- TDG para la persistencia

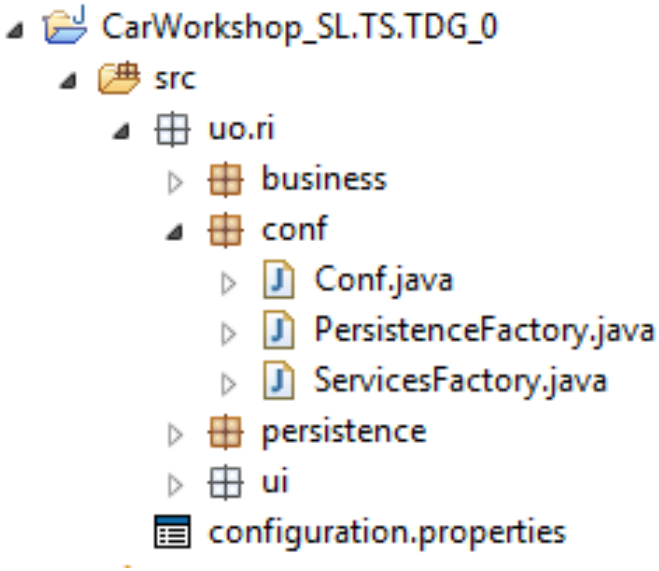


Capas separadas



SQL externalizado

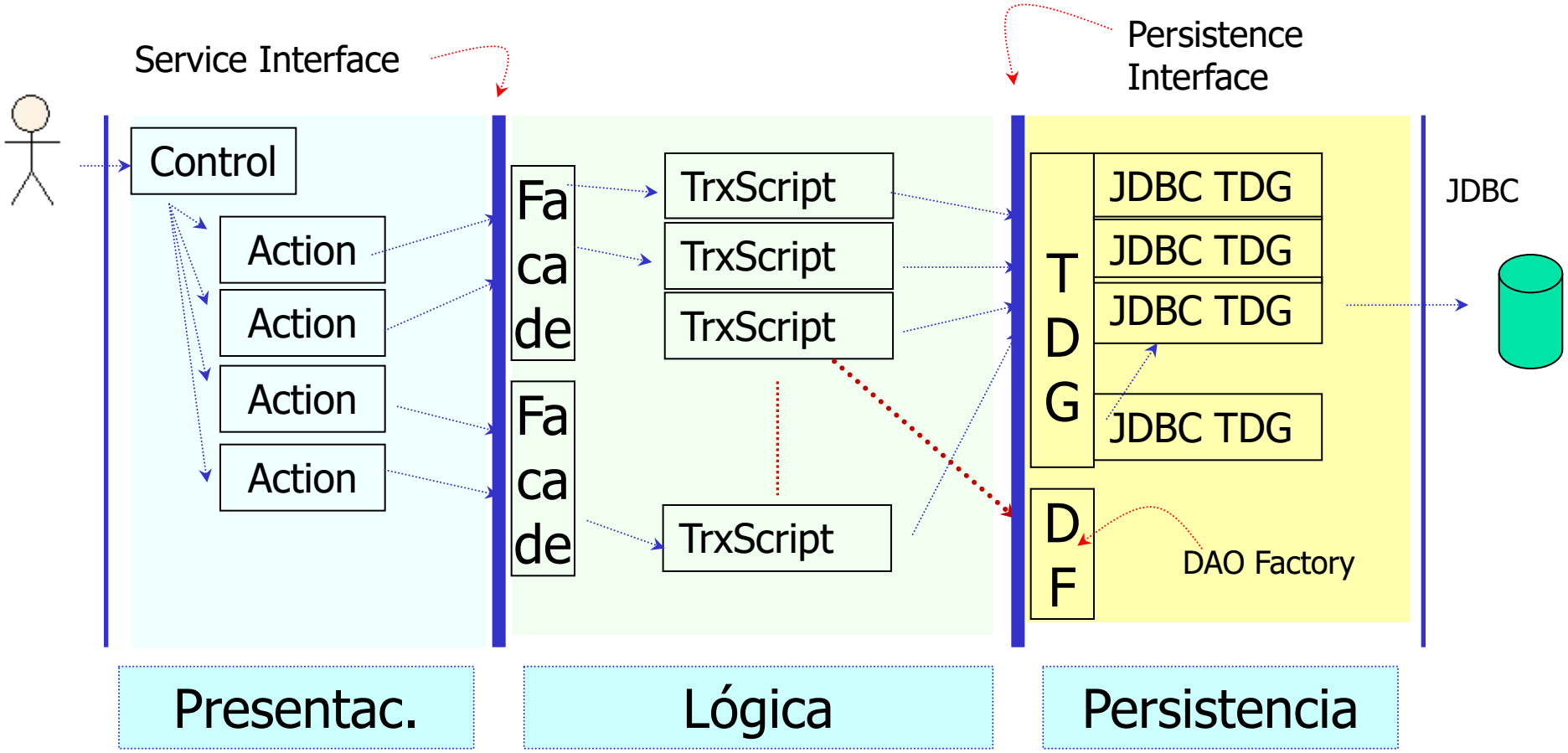
Transaction Script para la lógica



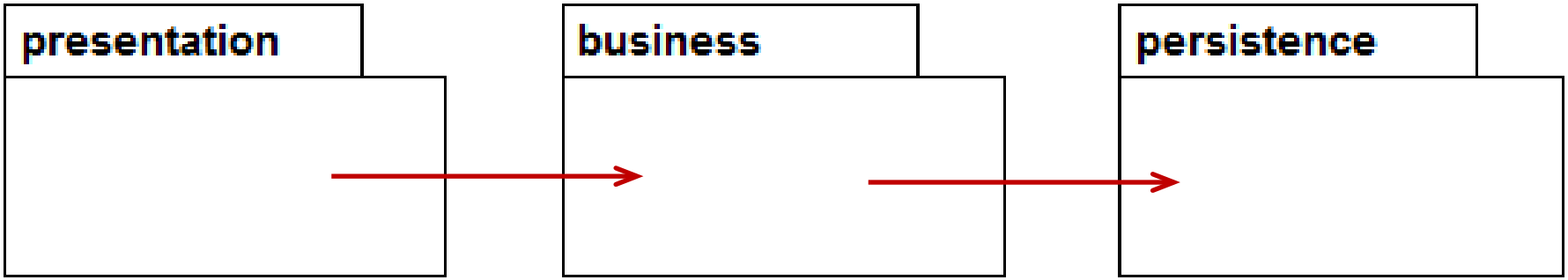
SL.TS.TDG_0

Factorías entre capas

Solución en capas



Dependencias



Las dependencias deberían ir hacia los paquetes más importantes

¿persistencia?

La persistencia es un detalle…

Ahora tendremos…

Arquitectura Hexagonal

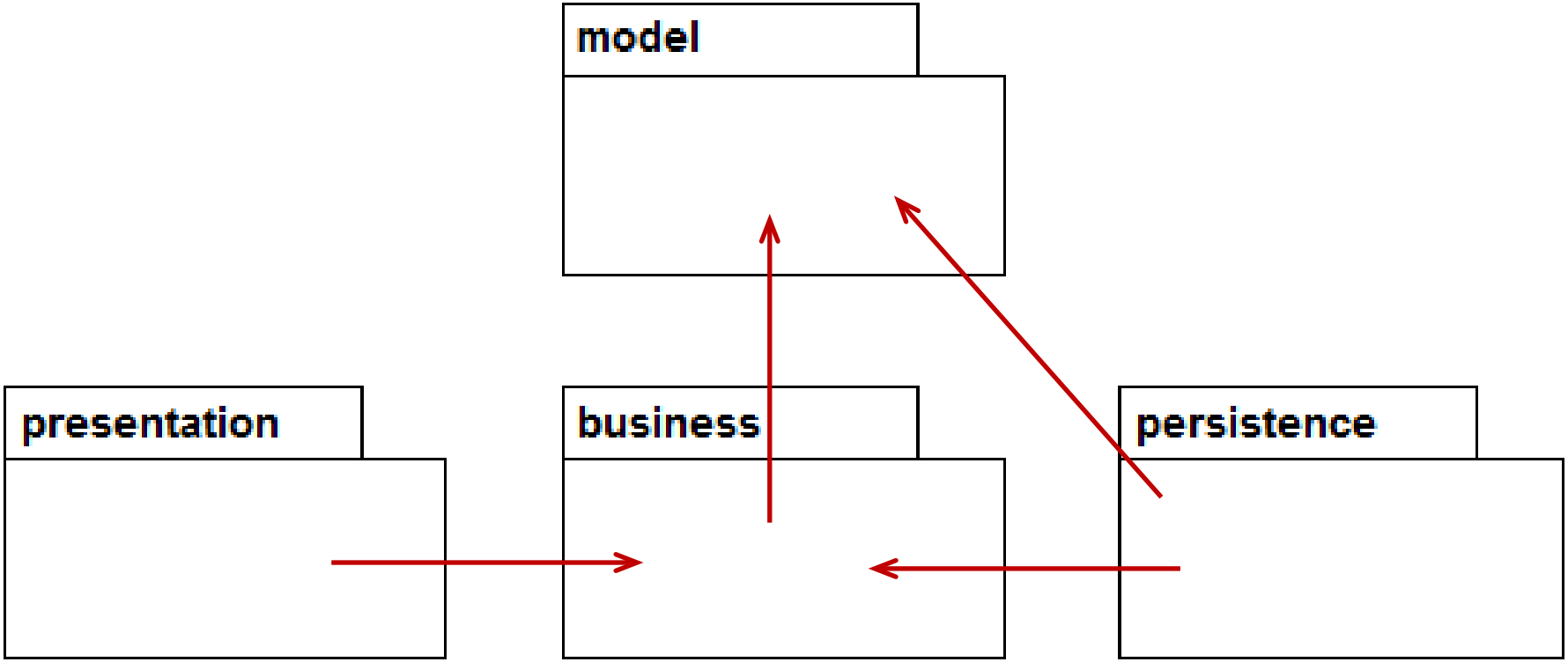
- Lógica en las clases de dominio
- No toda, pero sí la mayor parte y la fundamental
- Servicios en paquete application
  - Los transaction script aligeran, se transforman en comandos (patrón Command)

Ahora tendremos…

El mapeador hace persistencia

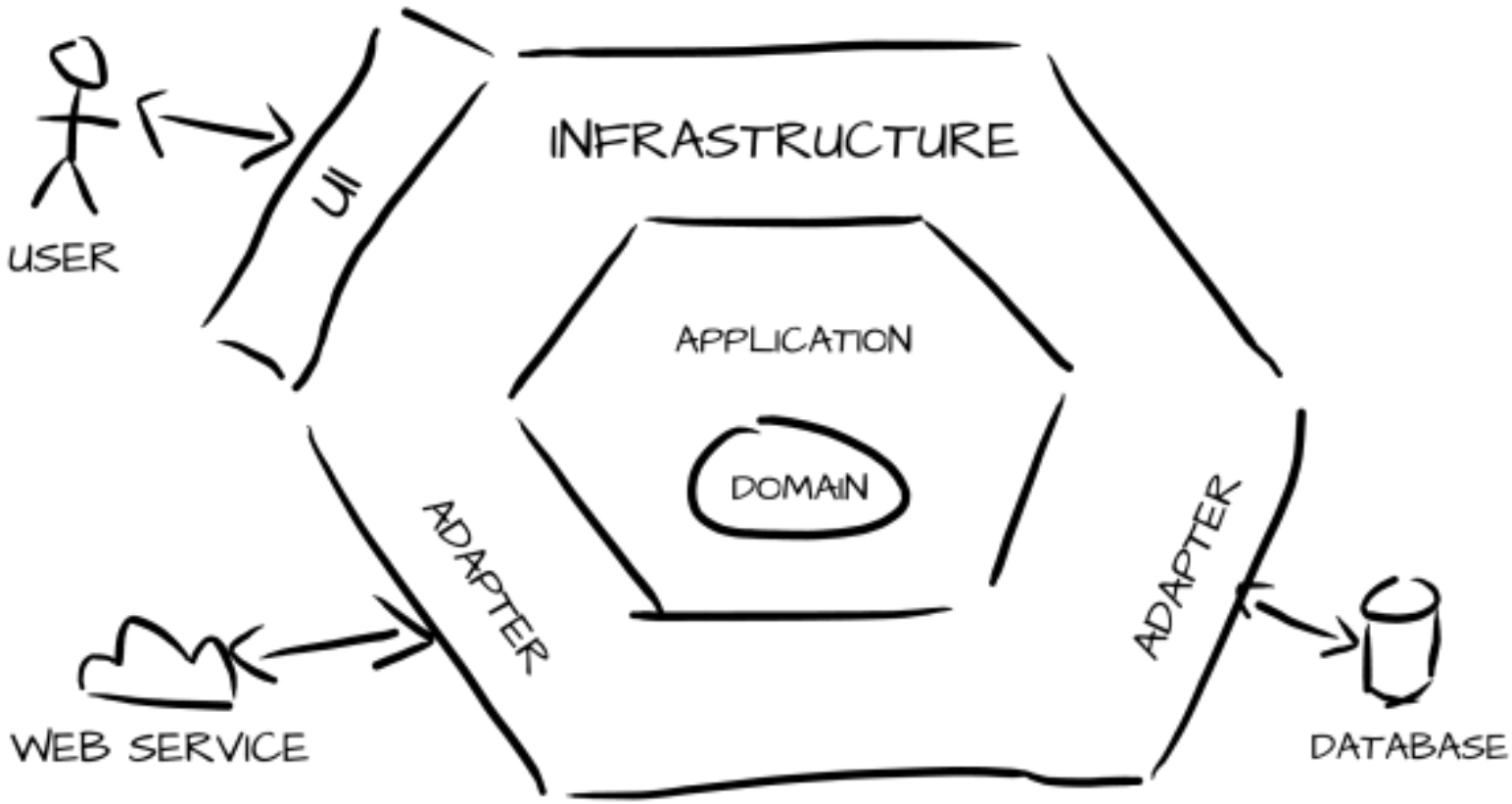
- La capa de persistencia casi desaparece
- Se reduce a métodos de consulta
- Consultas externalizadas en orm.xml

Dependencias



El modelo de dominio es el más importante

Hexagonal architecture



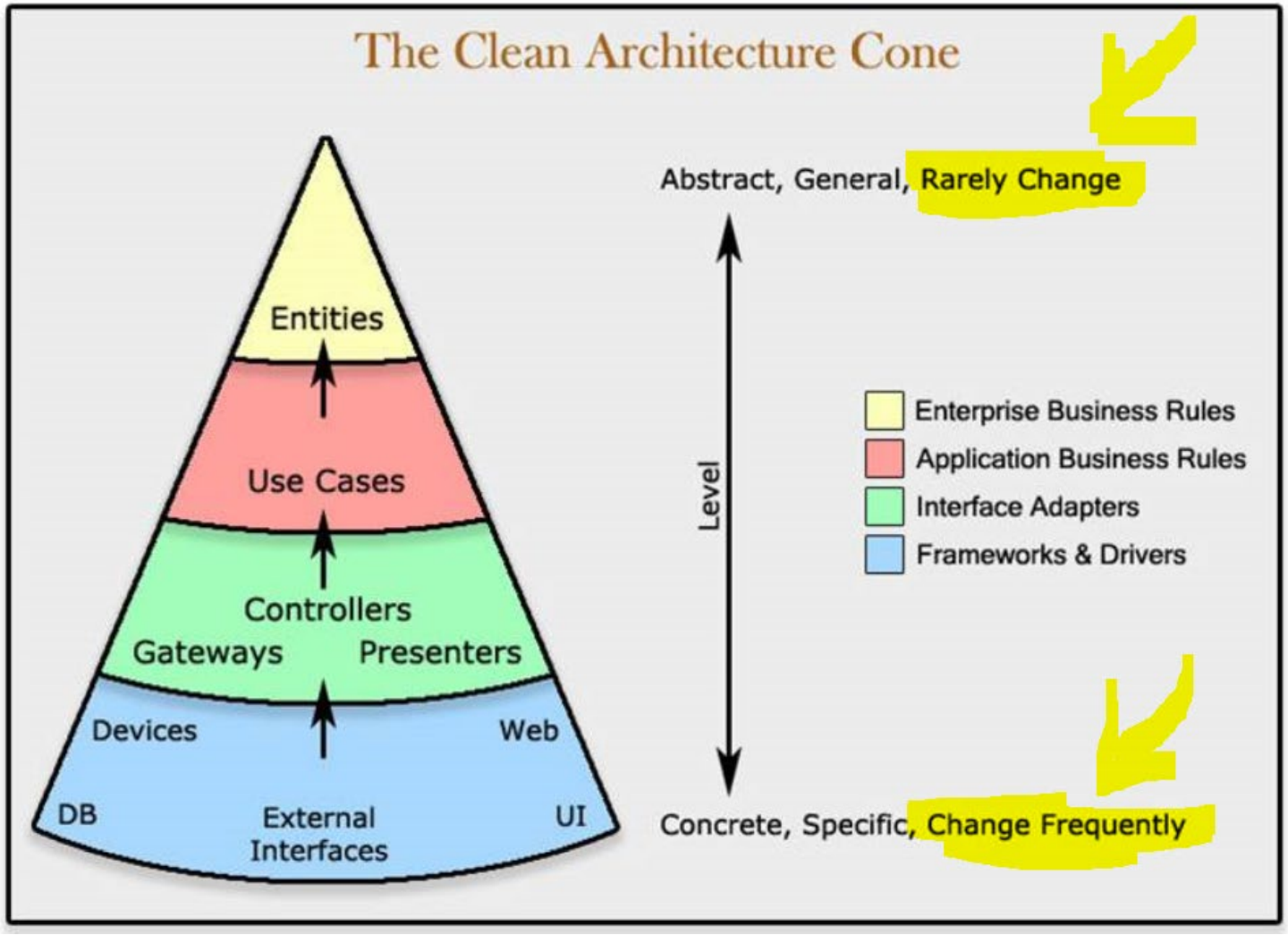
Documented in 2005 by Alistair Cockburn, Hexagonal Architecture is a software architecture that has many advantages and has seen renewed interest since 2015.

Produce sistemas que son

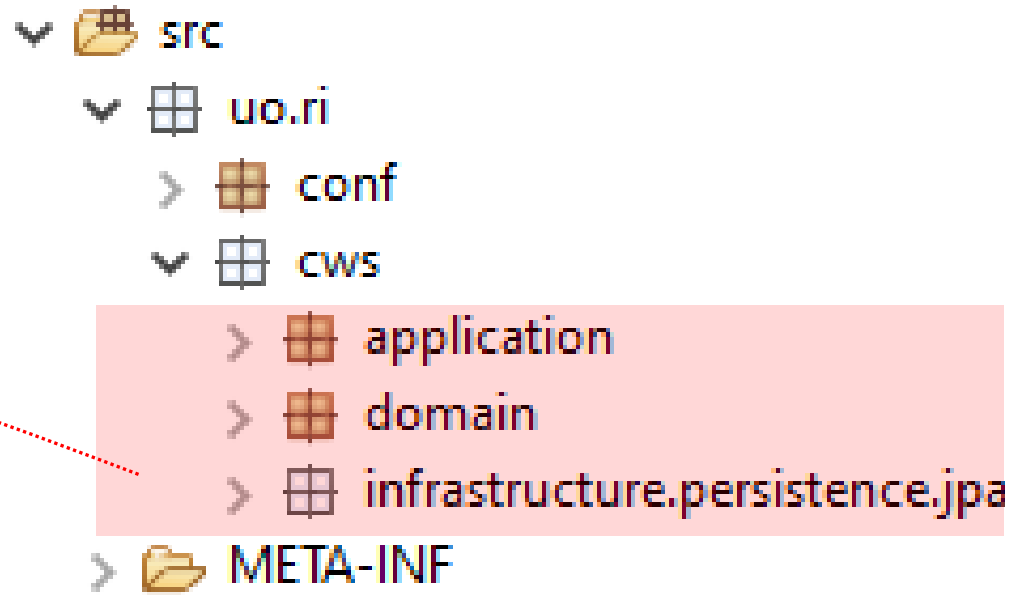
- Independientes de frameworks
- Testeables
- Independientes de interfaz de usuario
- Independientes de base de datos
- Independientes de agentes externos

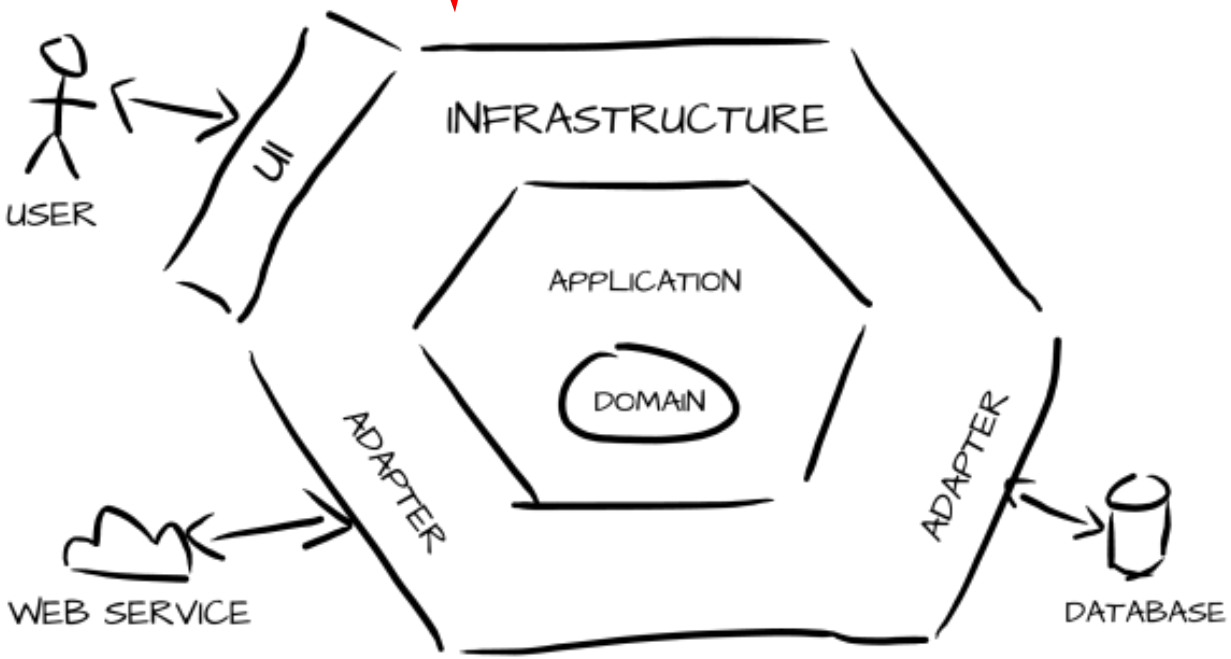
https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html

Cono de dependencias



Source and credit: https://8thlight.com/blog/uncle-bob/2012/08/13/the-clean-architecture.html
.https://www.codingblocks.net/podcast/clean-architecture-make-your-architecture-scream/


src
uo.ri
conf
cws
application
domain
infrastructure.persistence.jpa
META-INF


USER
UI
INFRASTRUCTURE
APPLICATION
DOMAIN
ADAPTER
ADAPTER
WEB SERVICE
DATABASE

Cambios en el diseño

Paquete del dominio

- Paquete supremo
- Sin dependencias de ninguna otra capa

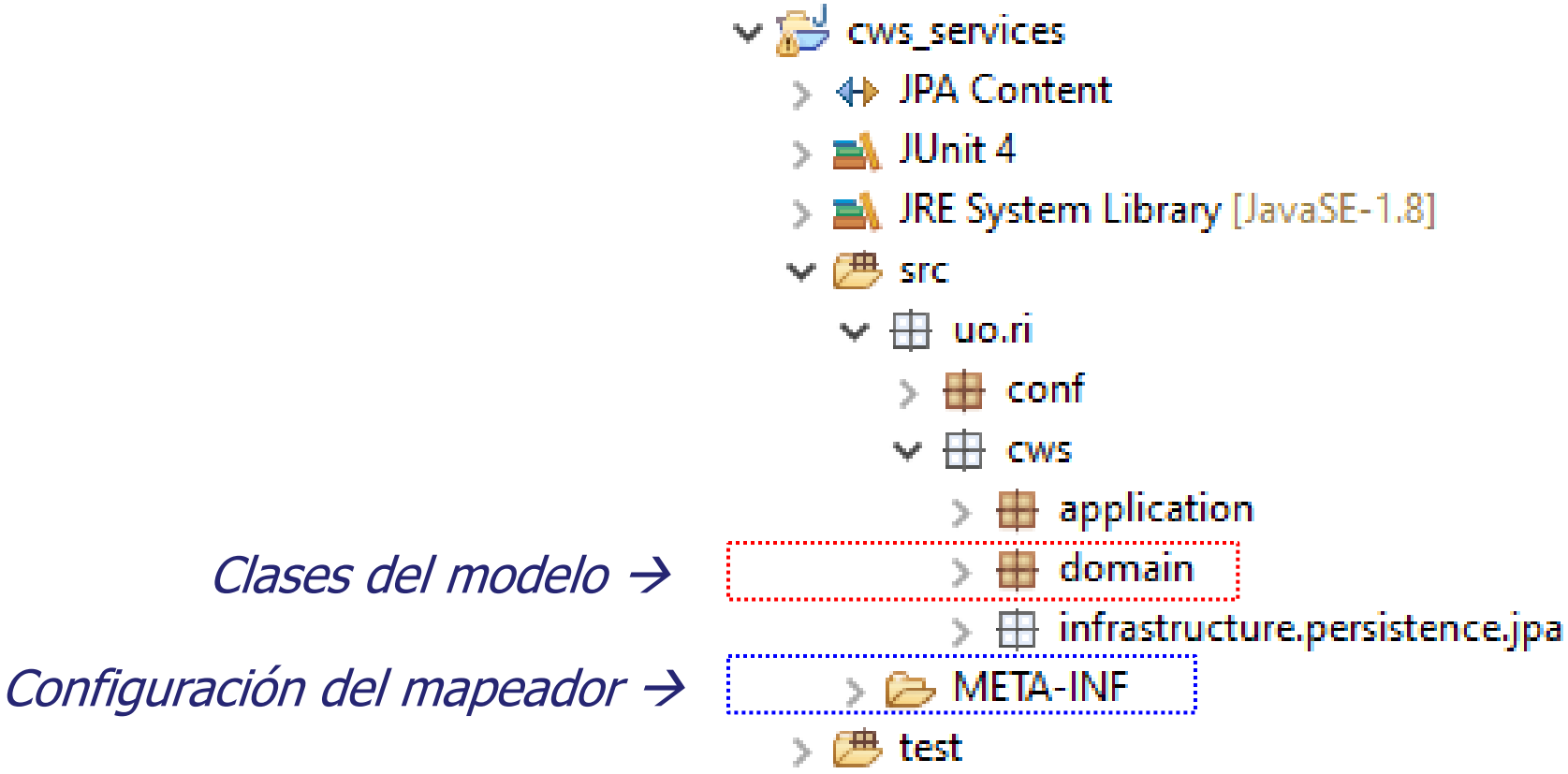
Paquete de aplicación

- Capa de servicios, solo depende de dominio
- Una interfaz por cada caso de uso
- Transaction Script refactorizados →comand
- Lanza BusinessException

Infraestructura, repositorios

- Interfaz de tipo colección
- y consultas

Paquete del dominio



Application

- application
  - repository
  - service
    - client
    - invoice
    - mechanic
      - crud
      - MechanicCrudService.java
      - MechanicDto.java

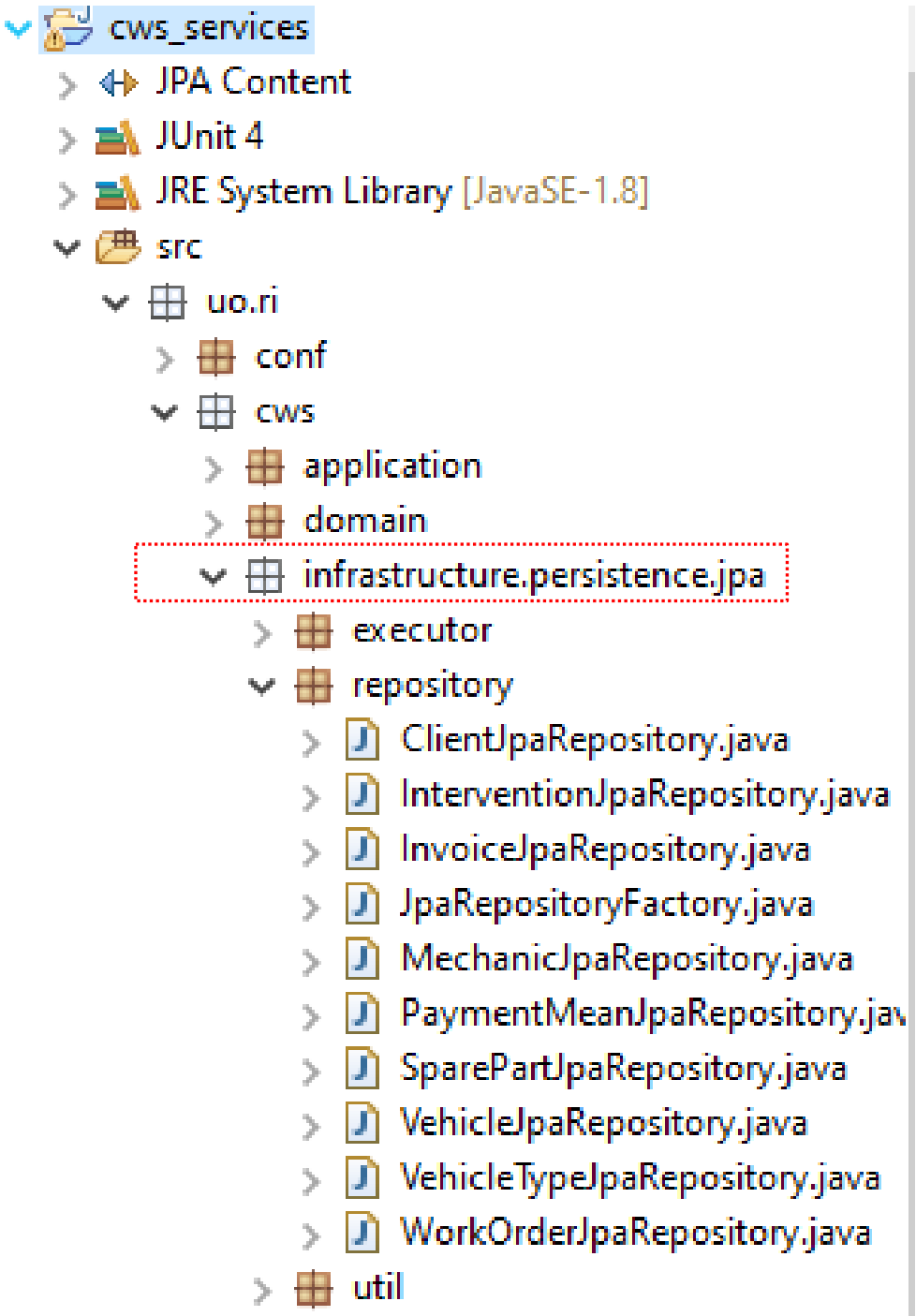
Interfaz y dto

- mechanic
  - crud ← *Caso de uso*
    - command
      - AddMechanic.java
      - DeleteMechanic.java ← *Commands*
      - FindAllMechanics.java
      - FindMechanicById.java
      - UpdateMechanic.java
    - MechanicCrudServiceImpl.java ← *Fachada*
  - MechanicCrudService.java
  - MechanicDto.java

Servicios →

- src
  - uo.ri
    - conf
    - cws
      - application
        - repository
        - service
          - client
          - invoice
          - mechanic
          - sparepart
          - vehicle
          - vehicletype
          - workorder
        - BusinessException.java
        - BusinessFactory.java
        - util
        - ServiceFactory.java
      - domain
      - infrastructure.persistence.jpa

Paquete infraestructura



Implementación de repositorios para JPA

Interfaces de servicio

Una por cada caso de uso

- Comunicación por DTO
- Lanzan BusinessException
- Uso de Optional

```
public interface MechanicCrudService {

    MechanicDto addMechanic(MechanicDto mecanico) throws BusinessException;

    void deleteMechanic(String idMecanico) throws BusinessException;
    void updateMechanic(MechanicDto mechanic) throws BusinessException;

    Optional<MechanicDto> findMechanicById(String id) throws BusinessException;
    List<MechanicDto> findAllMechanics() throws BusinessException;

}
```

https://es.wikipedia.org/wiki/Objeto_de_transferencia_de_datos

Patrón DTO Data Transfer Object

- Objetos simples → contenedor de datos
- Usados entre presentación y servicio
- Sólo datos, sin lógica

```
public class ClientDto {

    public String id;
    public Long version;

    public String dni;
    public String name;
    public String surname;
    public String addressStreet;
    public String addressCity;
    public String addressZipcode;
    public String phone;
    public String email;

}
```

```
public class MechanicDto {

    public String id;
    public Long version;

    public String dni;
    public String name;
    public String surname;

}
```

TS se convierten en comandos

```
public AddMechanic(String nombre, String apellidos) {
    this.nombre = nombre;
    this.apellidos = apellidos;
}

public void execute() {
    try {
        c = Jdbc.getConnection();

        MecanicosGateway db = PersistenceFactory.getMecanicoGateway();

        db.setConnection(c);
        db.save(nombre, apellidos);

    } catch (SQLException e) {
        throw new RuntimeException(e);
    }
    finally {
        Jdbc.close(c);
    }
}
```

En SL.TS.TDG_0

TS se convierten en comandos

```java
public AddMechanic(Mecanico mecanico) {
    this.mecanico = mecanico;
}

public Object execute() {
    EntityManagerFactory emf = Persistence.createEntityManagerFactory("caveatemptor");
    EntityManager em = emf.createEntityManager();
    EntityTransaction trx = em.getTransaction();

    em.persist( mecanico );

    trx.commit();
    em.close();

    return null;
}
```

Con modelo de dominio

```java
public AddMechanic(MechanicDto mechanic) {
    this.dto = mechanic;
}

@Override
public MechanicDto execute() throws BusinessException {
    checkValidData( dto );
    checkNotRepeatedDni( dto.dni );

    Mechanic m = new Mechanic(dto.dni, dto.name, dto.surname);
    repository.add( m );

    dto.id = m.getId();
    return dto;
}
```

```
public class CreateInvoiceFor {
```

← *En SL.TS.TDG_0*

Ejemplo crear factura

Con modelo de dominio

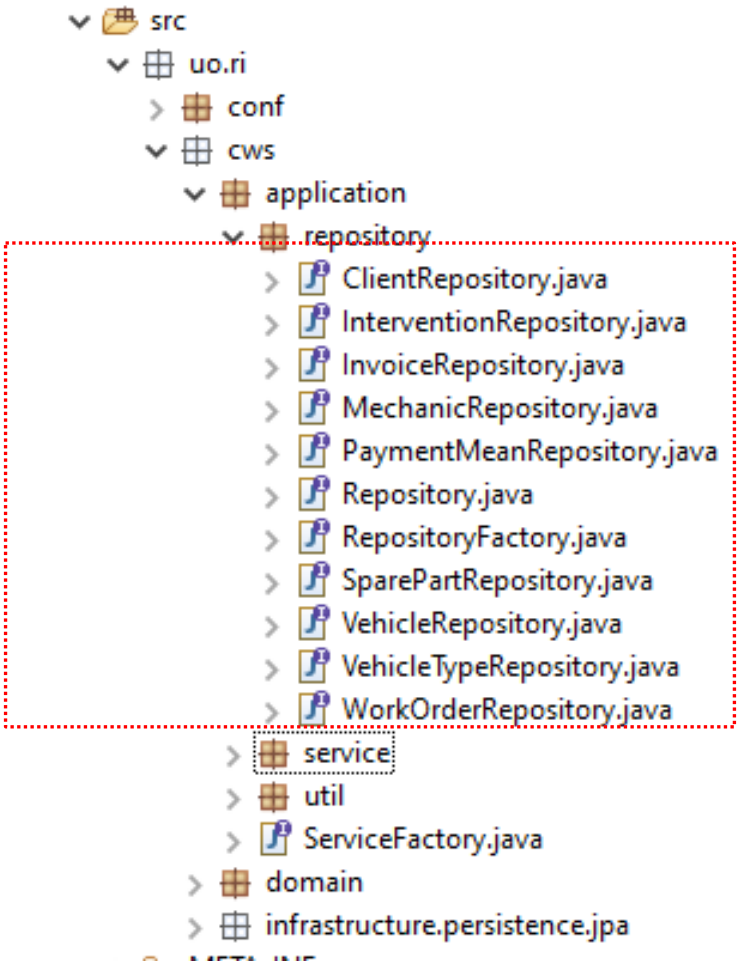
```
public class CreateInvoiceFor implements Command<InvoiceDto> {

    private List<Long> idsAveria;
    private AveriaRepository avrRepo = Factory.repository.forAveria();
    private FacturaRepository fctrRepo = Factory.repository.forFactura();

    public CreateInvoiceFor(List<Long> idsAveria) {
        this.idsAveria = idsAveria;
    }

    @Override
    public InvoiceDto execute() throws BusinessException {

        List<Averia> averias = avrRepo.findByIds( idsAveria );
        Long invoiceNumber = fctrRepo.getNextInvoiceNumber();

        Factura factura = new Factura( invoiceNumber, averias );
        fctrRepo.add( factura );

        return DtoAssembler.toDto( factura );
    }
```

https://martinfowler.com/eaaCatalog/repository.html

Repositorios

Almacén de objetos
Interfaz tipo colección: add, remove

¡¡¡No hay update!!!

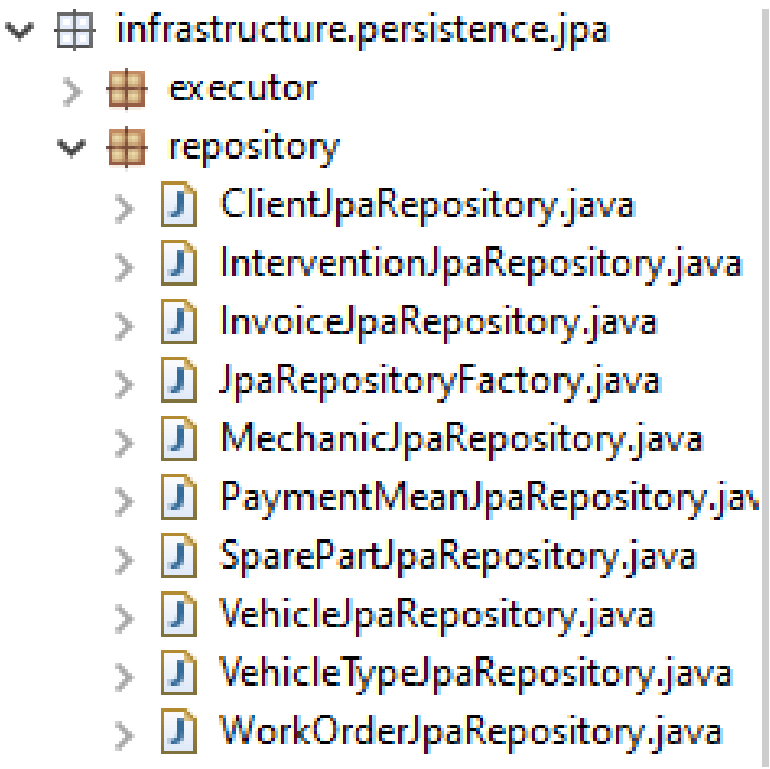



```java
public interface Repository<T> {
    void add(T t);
    void remove(T t);
    Optional<T> findById(String id);
}
```

+ consultas

```java
public interface MechanicRepository extends Repository<Mechanic> {

    Optional<Mechanic> findByDni(String dni);
    List<Mechanic> findAll();
```

Infraestructura: impl de repositorios



Las implementaciones de repositorios resuelven métodos de consulta usando el mapeador

```
public class WorkOrderJpaRepository
        extends BaseJpaRepository<WorkOrder>
        implements WorkOrderRepository {

    @Override
    public List<WorkOrder> findByIds(List<Long> idsAveria) {
        return Jpa.getManager()
                .createNamedQuery("WorkOrder.findByIds", WorkOrder.class)
                .setParameter( 1, idsAveria )
                .getResultList();
        }

}
```

Un paso más allá…

Centralizar el control de transacciones

- Y si se necesita, el de acceso, el de auditoría, etc.

- Eliminar código repetitivo de los TS (Transaction Script)

Eliminar código repetitivo

```
public class UpdateMechanic {

    private Mecanico mecanico;

    public UpdateMechanic(Mecanico mecanico) {..

    public Object execute() throws BusinessException {
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("car
        EntityManager em = emf.createEntityManager();
        EntityTransaction trx = em.getTransaction();

        Mecanico m = em.merge( mecanico );

        trx.commit();
        em.close();

        return m;
    }

}
```

Eliminar código repetitivo

```
public class AddMechanic {

    private Mecanico mecanico;

    public AddMechanic(Mecanico mecanico) {..

    public Object execute() {
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("car
        EntityManager em = emf.createEntityManager();
        EntityTransaction trx = em.getTransaction();

        em.persist( mecanico );

        trx.commit();
        em.close();

        return null;
    }

}
```

Compara con la anterior… ¿qué cambia?

Eliminar código repetitivo

```
public class AddMechanic {

    private Mecanico mecanico;

    public AddMechanic(Mecanico mecanico) {

    public Object execute() {
        EntityManagerFactory emf = Persistence.createEntityManagerFa
        EntityManager em = emf.createEntityManager();
        EntityTransaction trx = em.getTransaction();

        em.persist( mecanico );

        trx.commit();
        em.close();

        return null;
    }

}
```

Ese código se repite una y otra vez…
Vamos a factorizarlo

Centralizar transacciones

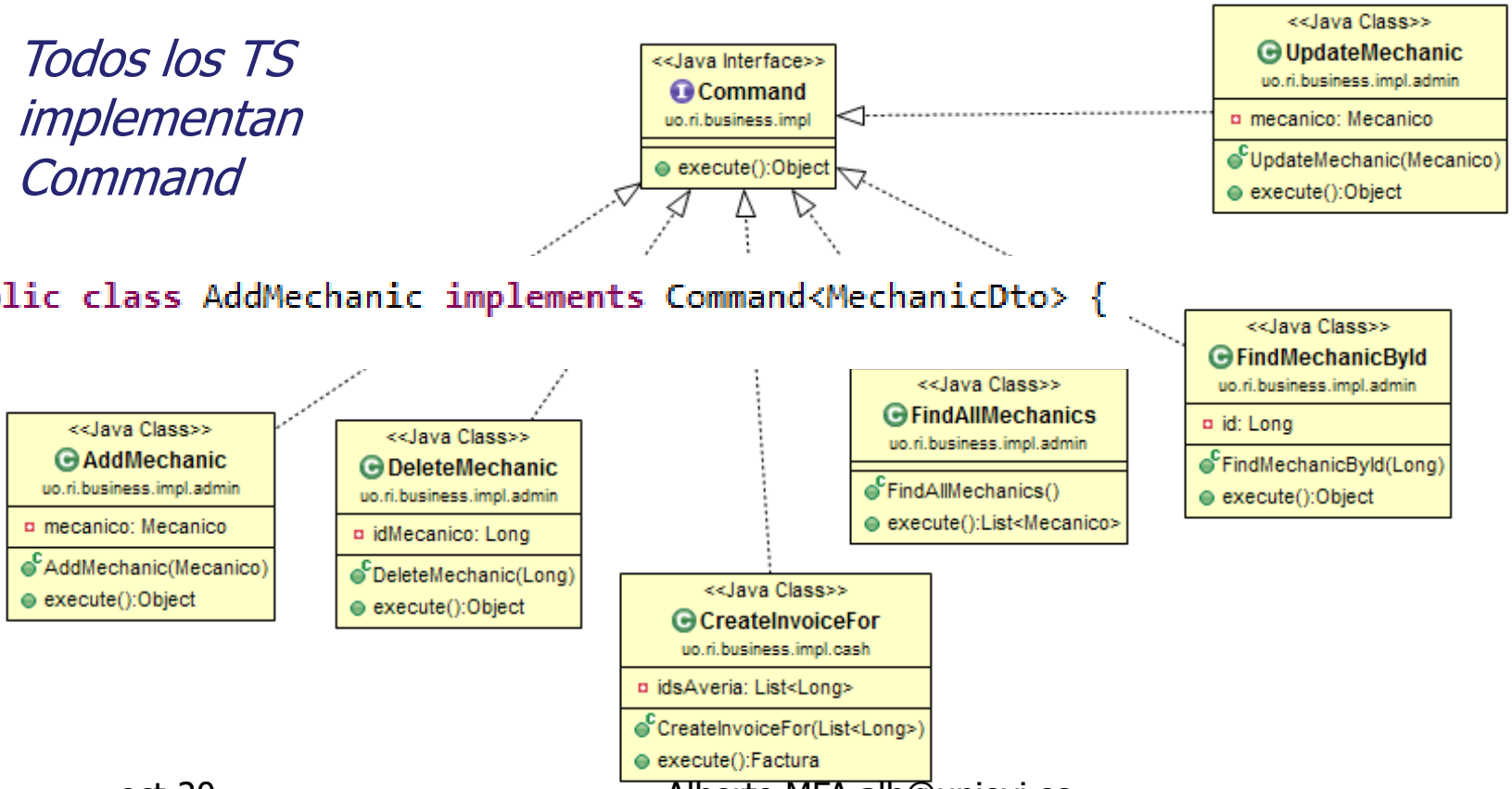
Pasos:

- Uniformizar los Transaction Script
  - Interfaz Command
- Extraer el control de trx a una única clase
  - Command Executor
- Modificar las implementaciones de las fachadas
  - MechanicCrudServiceImpl, etc.
- Clase de Utilidad para gestionar el EntityManager

Uniformizar los TS

```
public interface Command<T> {

    T execute() throws BusinessException;
}
```

Todos los TS implementan Command




```
public class AddMechanic implements Command<MechanicDto> {
```

Extraer control a una única clase

```java
public class JpaCommandExecutor implements CommandExecutor {

    @Override
    public <T> T execute(Command<T> cmd) throws BusinessException {
        EntityManager mapper = Jpa.createEntityManager();
        try {
            EntityTransaction trx = mapper.getTransaction();
            trx.begin();

            try {
                T res = cmd.execute();
                trx.commit();

                return res;

            } catch (BusinessException | RuntimeException ex) {
                if ( trx.isActive() ) {
                    trx.rollback();
                }
                throw ex;
            }
        } finally {
            if ( mapper.isOpen() ) {
                mapper.close();
            }
        }
    }
}
```

Gestión de excepciones y transacción centralizado

```java
public interface CommandExecutor {

    <T> T execute(Command<T> cmd) throws BusinessException;

}
```

Modificar implement. de fachada

Gestión de excepciones y transacción centralizado

```
public class MechanicCrudServiceImpl implements MechanicCrudService {

    private CommandExecutor executor = Factory.executor.forExecutor();

    @Override
    public void addMechanic(MechanicDto mecanico) throws BusinessException {
        executor.execute( new AddMechanic( mecanico ) );
    }

    @Override
    public void updateMechanic(MechanicDto mecanico) throws BusinessException {
        executor.execute( new UpdateMechanic( mecanico ) );
    }

    @Override
    public void deleteMechanic(Long idMecanico) throws BusinessException {
        executor.execute( new DeleteMechanic(idMecanico) );
    }
```

Clase de utilidad para EM

```
public class Jpa {
    public static EntityManager getManager() {..
    public static EntityManager createEntityManager() {..
```

createEntityManager()

- Crea uno nuevo
- Solo invocado por el Command Executor

getManager()

- Invocado desde los repositorios
- Devuelve el contexto de persistencia actual

Clase de utilidad para EM

Usada desde

- Command executor
- Repositorios

```java
public class BaseJpaRepository<T> {

    public void add(T t) {
        Jpa.getManager().persist( t );
    }

    public void remove(T t) {
        Jpa.getManager().remove( t );
    }

    public Optional<T> findById(String id) {
        T found = Jpa.getManager().find(type, id);
        return found != null
                ? Optional.of(found)
                : Optional.empty();
    }
```

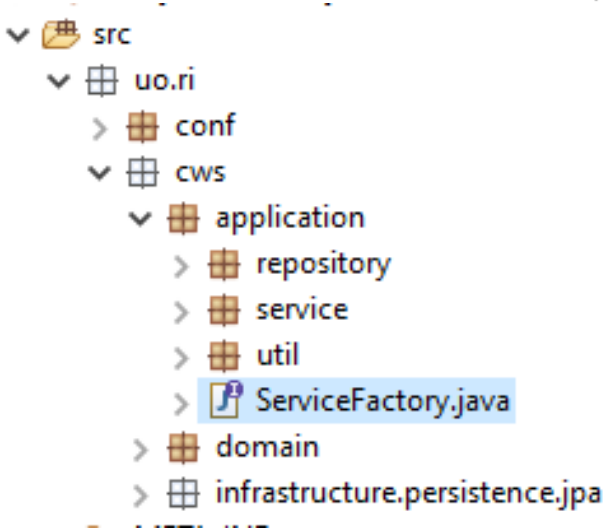
```java
public class JpaCommandExecutor implements CommandExecutor {

    @Override
    public <T> T execute(Command<T> cmd) throws BusinessException {
        EntityManager mapper = Jpa.createEntityManager();
        try {
            EntityTransaction trx = mapper.getTransaction();
            trx.begin();

            try {
                T res = cmd.execute();
                trx.commit();

                return res;
```

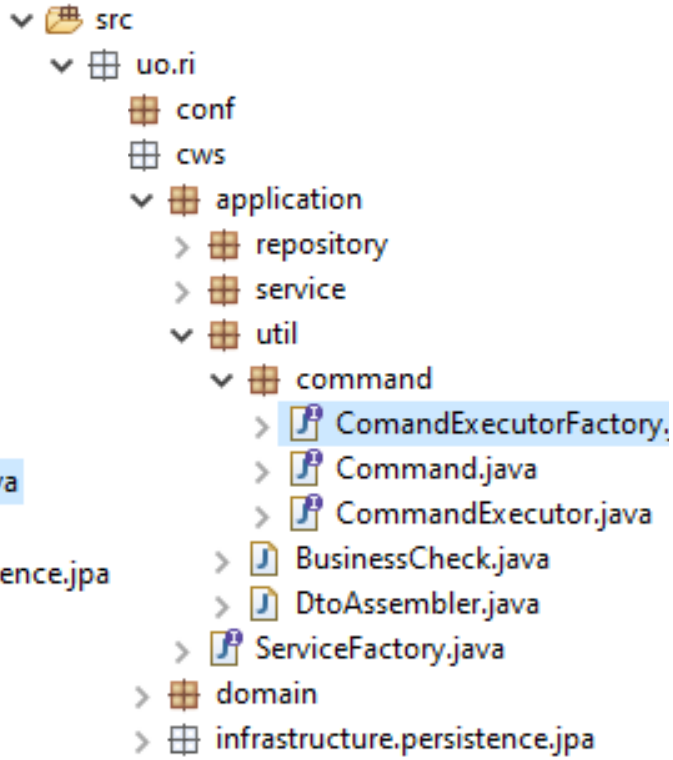
```java
public class AddMechanic implements Command<MechanicDto> {

    private MechanicDto dto;
    private MechanicRepository repository = Factory.repository.forMechanic();

    public AddMechanic(MechanicDto mechanic) {
        this.dto = mechanic;
    }

    @Override
    public MechanicDto execute() throws BusinessException {
        checkValidData( dto );
        checkNotRepeatedDni( dto.dni );

        Mechanic m = new Mechanic(dto.dni, dto.name, dto.surname);
        repository.add( m );

        dto.id = m.getId();
        return dto;
    }
```

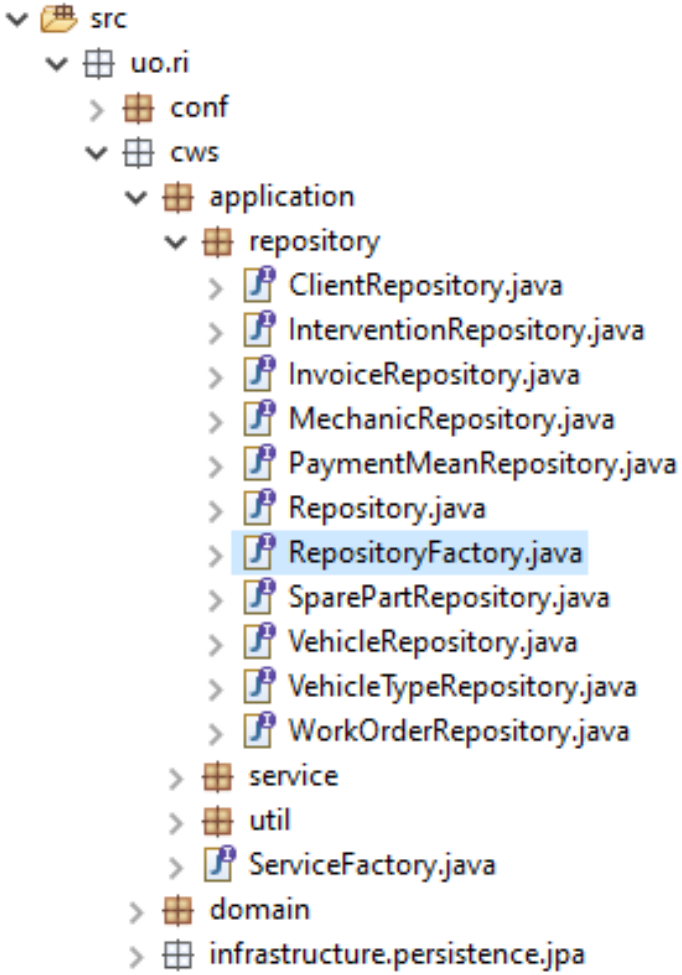
Código sin duplicidades ni dependencias

```java
public class CreateInvoiceFor implements Command<InvoiceDto>{

    private List<String> workOrderIds;
    private WorkOrderRepository wrkrsRepo = Factory.repository.forWorkOrder();
    private InvoiceRepository invsRepo = Factory.repository.forInvoice();

    public CreateInvoiceFor(List<String> workOrderIds) {
        this.workOrderIds = workOrderIds;
    }

    @Override
    public InvoiceDto execute() throws BusinessException {
        List<WorkOrder> avs = wrkrsRepo.findByIds( workOrderIds );
        BusinessCheck.isFalse( avs.isEmpty(), "There are no such work orders");
        BusinessCheck.isTrue( allFinished(avs), "Not all orders are finished");

        Long numero = invsRepo.getNextInvoiceNumber();

        Invoice f = new Invoice(numero, avs);
        invsRepo.add( f );

        return DtoAssembler.toDto( f );
    }
```

Factorías







Patrón Abstract Factory

Interfaces factoría y repositorio

Un repositorio por cada entidad

```
public interface RepositoryFactory {

    MechanicRepository forMechanic();
    WorkOrderRepository forWorkOrder();
    PaymentMeanRepository forPaymentMean();
    InvoiceRepository forInvoice();
    ClientRepository forClient();
    SparePartRepository forSparePart();
    InterventionRepository forIntervention();
    VehicleRepository forVehicle();
    VehicleTypeRepository forVehicleType();
}
```

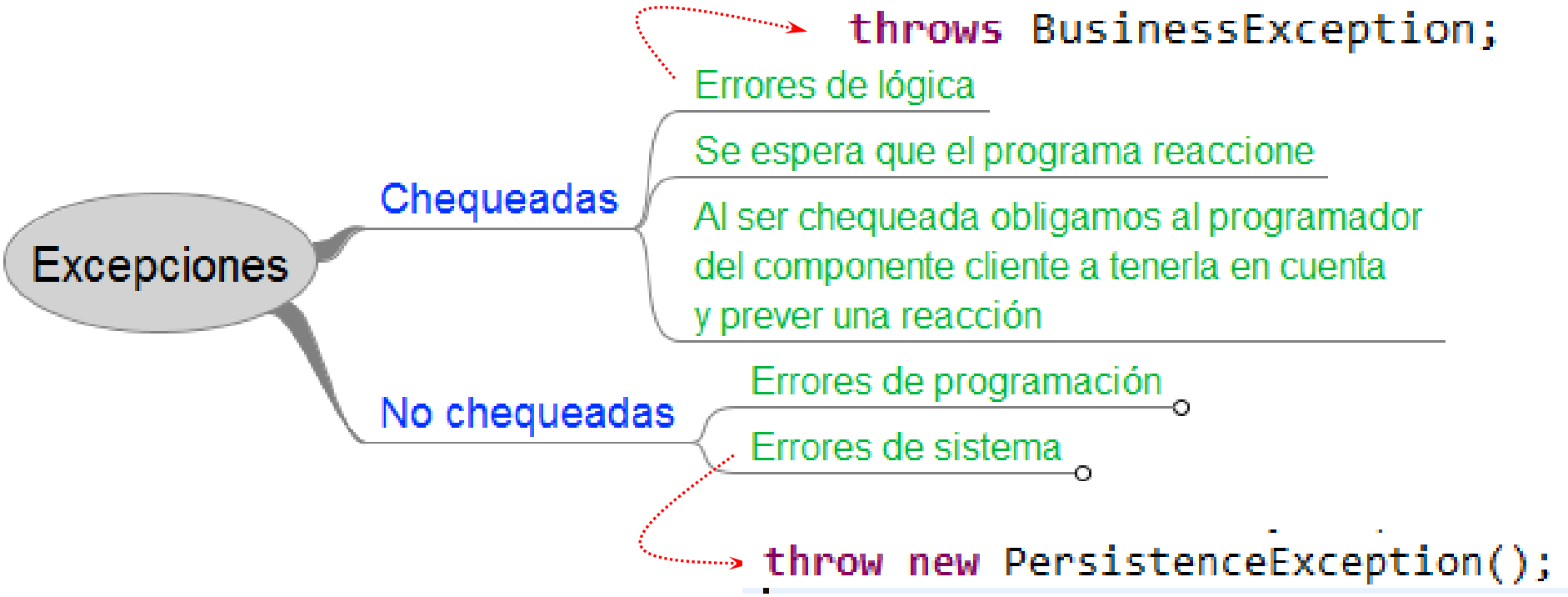
```
public interface ServiceFactory {

    // Manager use cases
    MechanicCrudService forMechanicCrudService();
    VehicleTypeCrudService forVehicleTypeCrudServi
    SparePartCrudService forSparePartCrudService();

    // Cash use cases
    CreateInvoiceService forCreateInvoiceService();
    SettleInvoiceService forSettleInvoiceService();

    // Foreman use cases
    VehicleCrudService forVehicleCrudService();
    ClientCrudService forClienteCrudService();
    ClientHistoryService forClientHistoryService();
    WorkOrderCrudService forWorkOrderCrudService();

    // Mechanic use cases
    CloseWorkOrderService forClosingBreakdown();
    ViewAssignedWorkOrdersService forViewAssignedWorkOrdersService();

}
```

Una interfaz de servicio por cada caso de uso

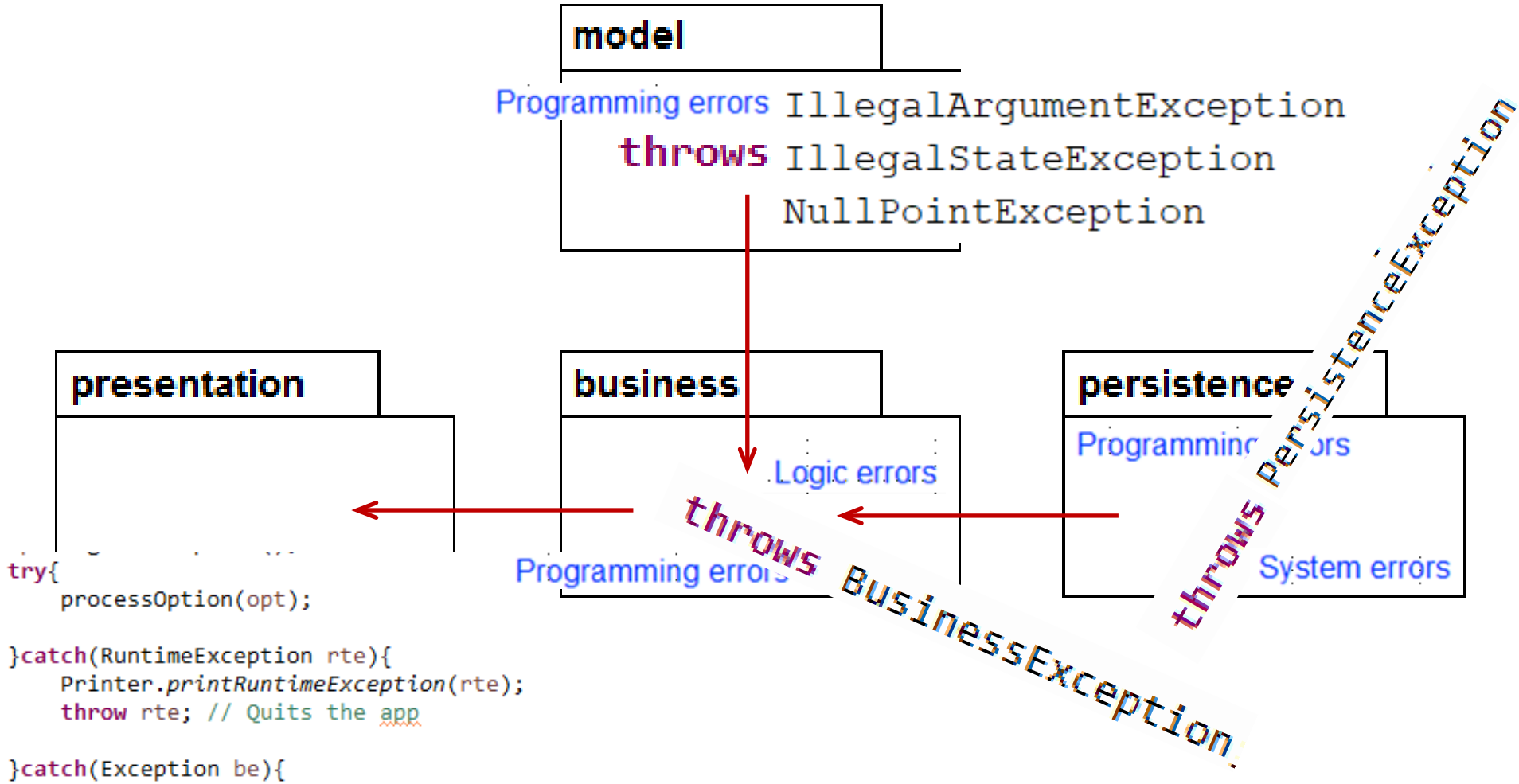
Gestión de errores

- De lógica
  - Algún problema con las reglas de negocio
    - facturar avería que no existe
    - borrar vehiculo que tiene averías
    - insertar usuario con dni repetido
    - ...
  - Suelen ser culpa del usuario
  - Reacción del programa
    - Aborta operación actual
    - Logea detalles
    - Avisa al usuario
    - El programa continua...
  - El usuario debe corregir
- De sistema
  - Algún compomente del sistema no funciona bien
    - mala configuración de parámetros
    - caída de red
    - bdd no operativa
    - disco lleno, estropeado...
  - No se espera que el programa sea capaz de recuperarse de estos problemas
  - Reacción del programa
    - Aborta operación actual
    - Logea detalle del problema
    - Informa al usuario, y
    - Se para la ejecución
  - El administrador debe solucionar
- De programación
  - Señalan bugs
    - NullPointerException
    - InvalidArguementException
    - IllegalStateException
    - ...
  - Culpa del programador
  - No se gestionan, se arreglan, no hay catch especifico
  - El programador debe solucionar

Gestión de errores

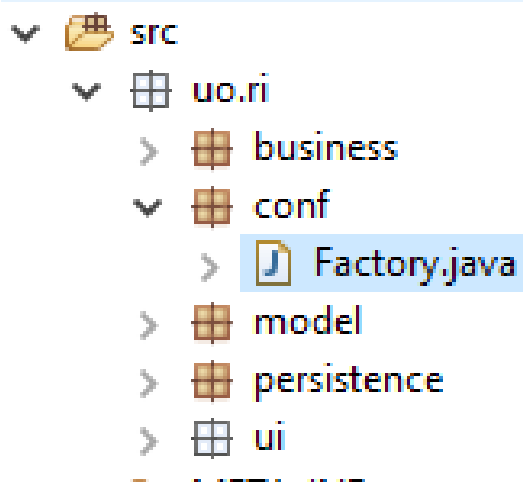


Propagación de excepciones




```
try{
    processOption(opt);

}catch(RuntimeException rte){
    Printer.printRuntimeException(rte);
    throw rte; // Quits the app

}catch(Exception be){
    Printer.printBusinessException(be);
}
```

```java
public class AdminMain {

    public static void main(String[] args) {
        new AdminMain()
            .configure()
            .run()
            .close();
    }

    private AdminMain configure() {
        Factory.service = new BusinessFactory();
        Factory.repository = new JpaRepositoryFactory();
        Factory.executor = new JpaExecutorFactory();

        return this;
    }
```



La misión de main() es configurar las dependencias y arrancar el programa

Main & Configuración

```java
public class Factory {

    public static RepositoryFactory repository;
    public static ServiceFactory service;
    public static CommandExecutorFactory executor;

}
```