Application development

Information repositories

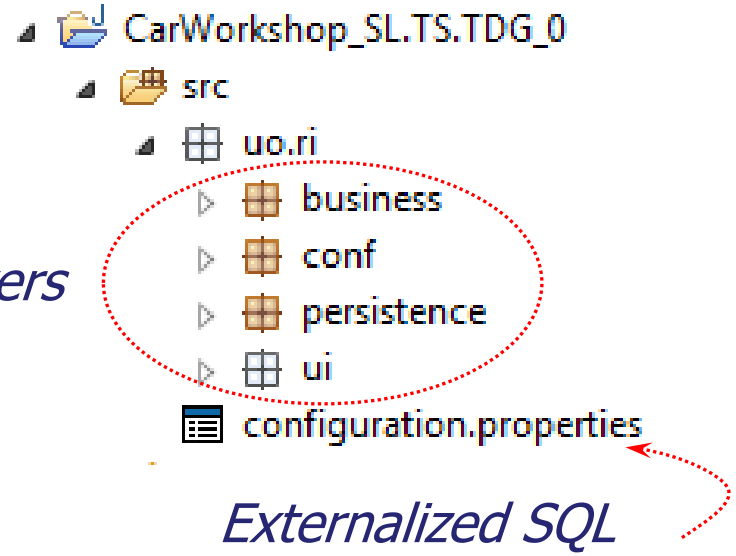
Introduction

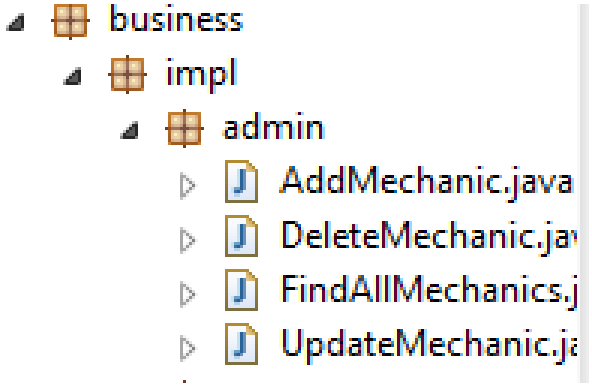
Lets adapt the design from SL.TS.TDG_0

Its architecture:

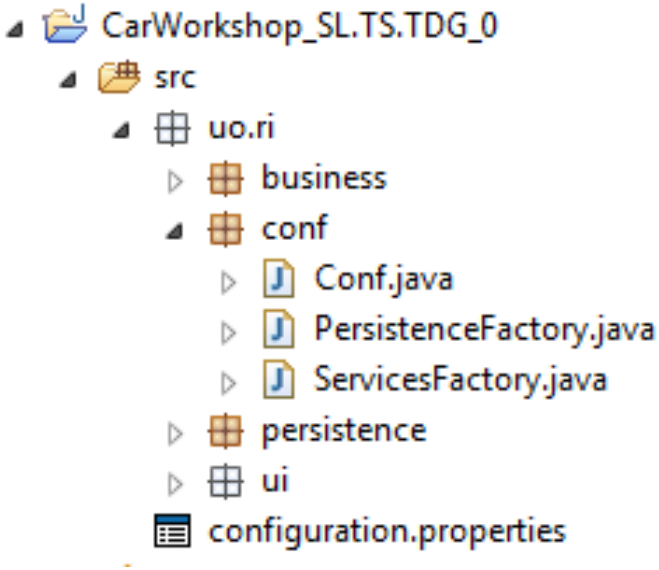
- 3 Layers
- Factories between layers
- Externalized SQL
- Logic with Transaction Script
- Persistence with TDG

In SL.TS.TDG_0



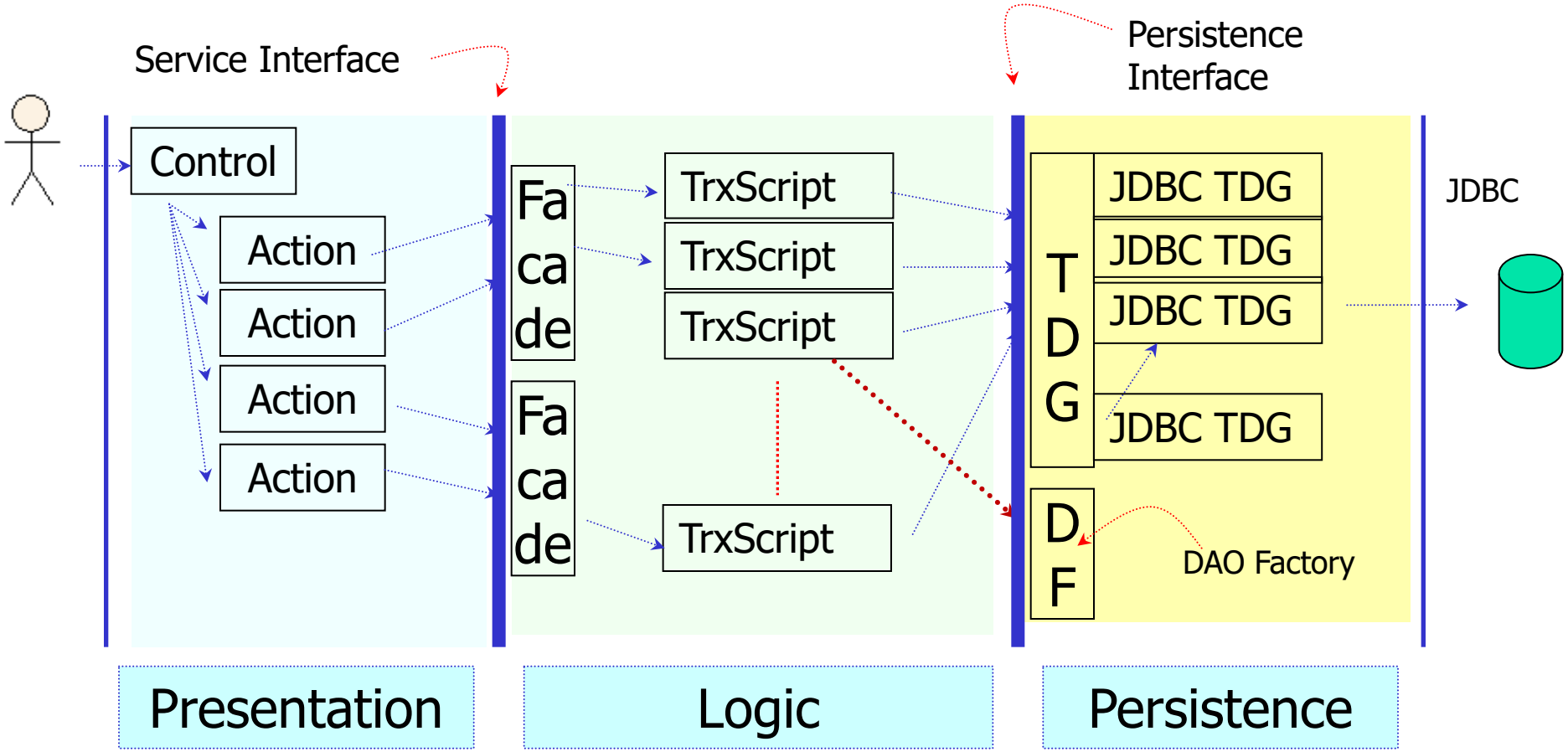


Logic with Transaction Script

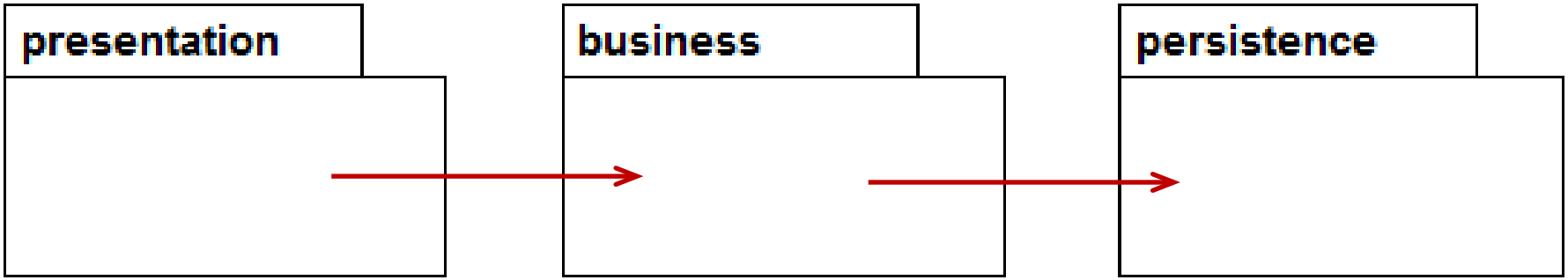


Factories between layers

Layered solution



Dependencies



Dependencies should point to the most important packages/components

persistence?

Persistence is a detail…

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

Now...

Hexagonal architecture

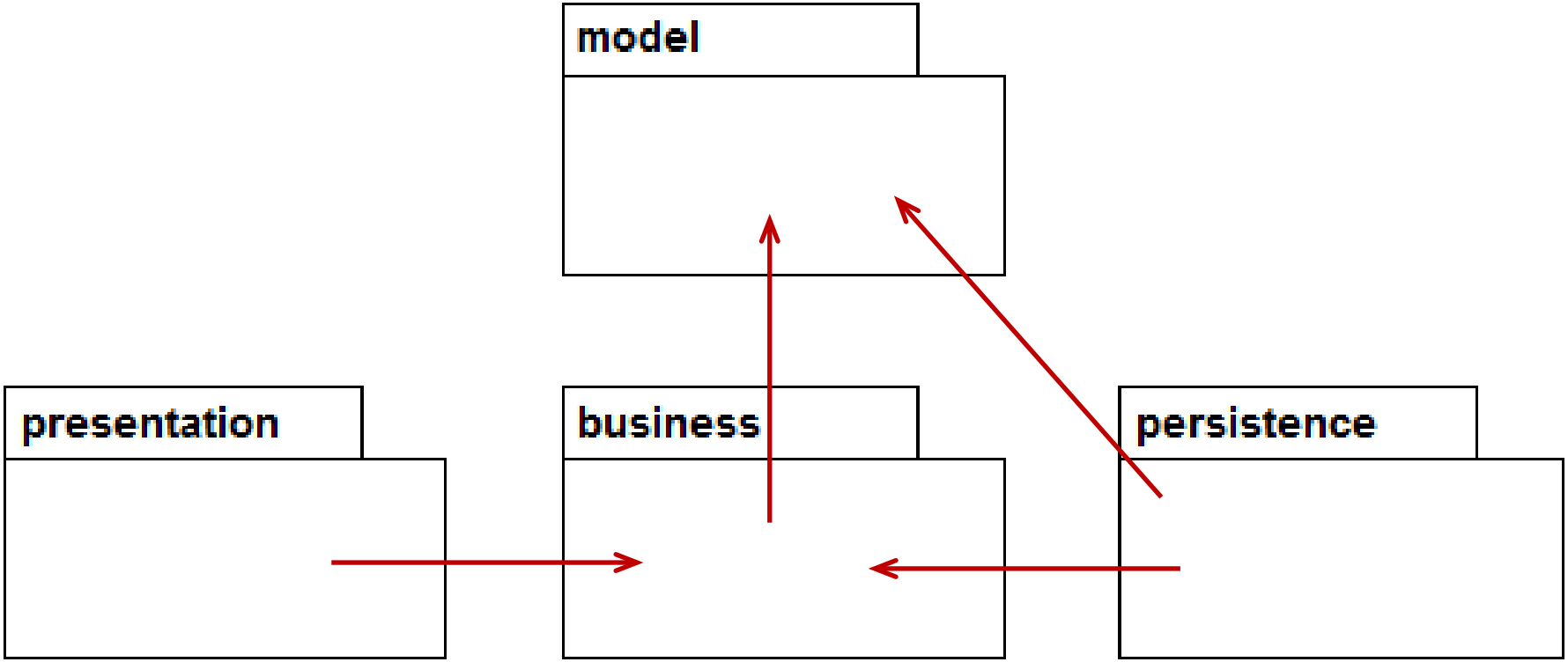
- Logic in the domain classes
- Not all, but the great part, and the most fundamental
- Application logic, the rest of the domain logic
  - Transaction Scripts reduced, now they are commands (Command pattern)

Now...

The mapper solves all the persistence

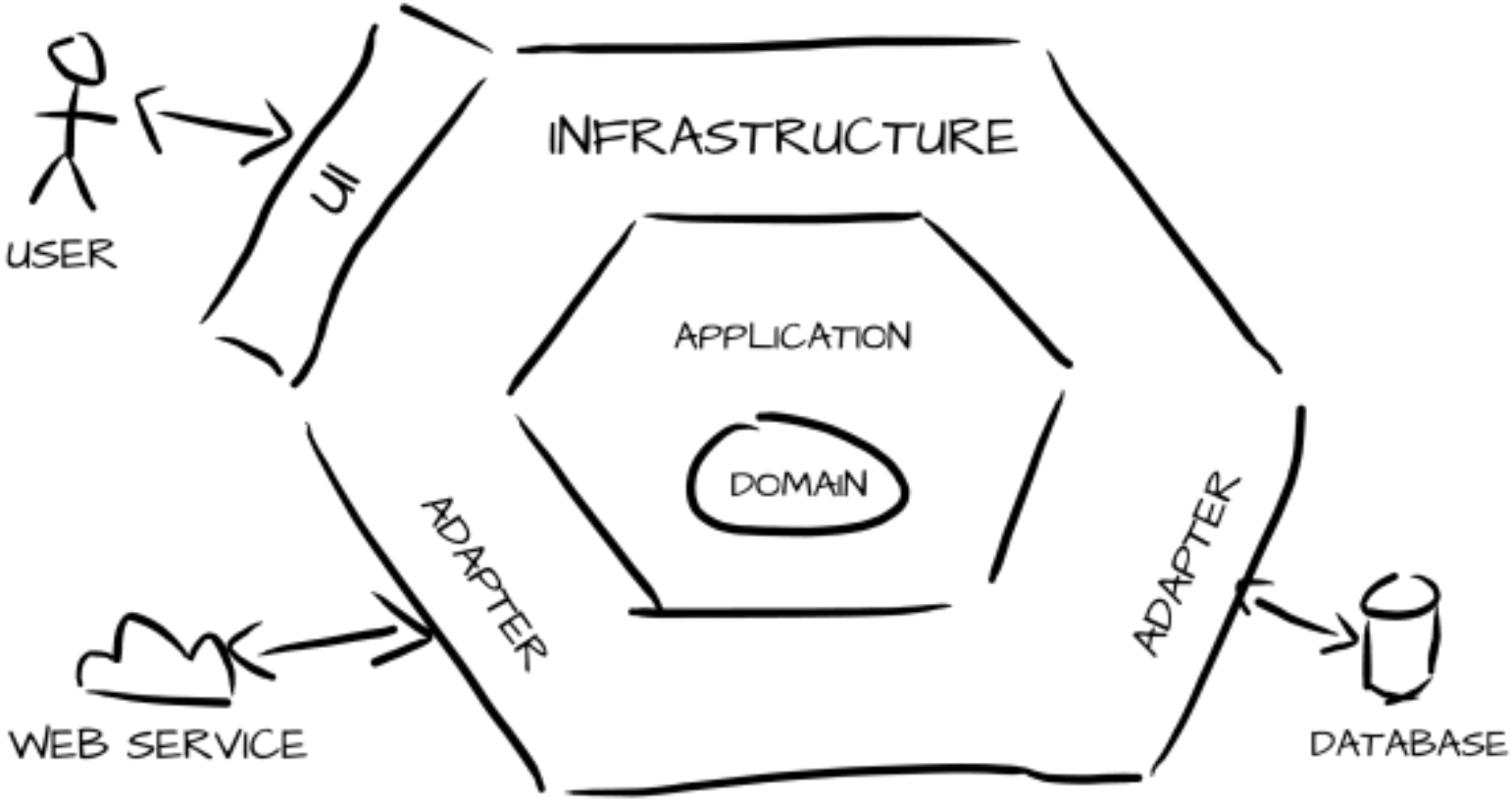
- Persistence layer very thin
- Mostly reduced to query methods
- Queries externalized to orm.xml file

Now dependencies



The domain model is most important part

Hexagonal architecture



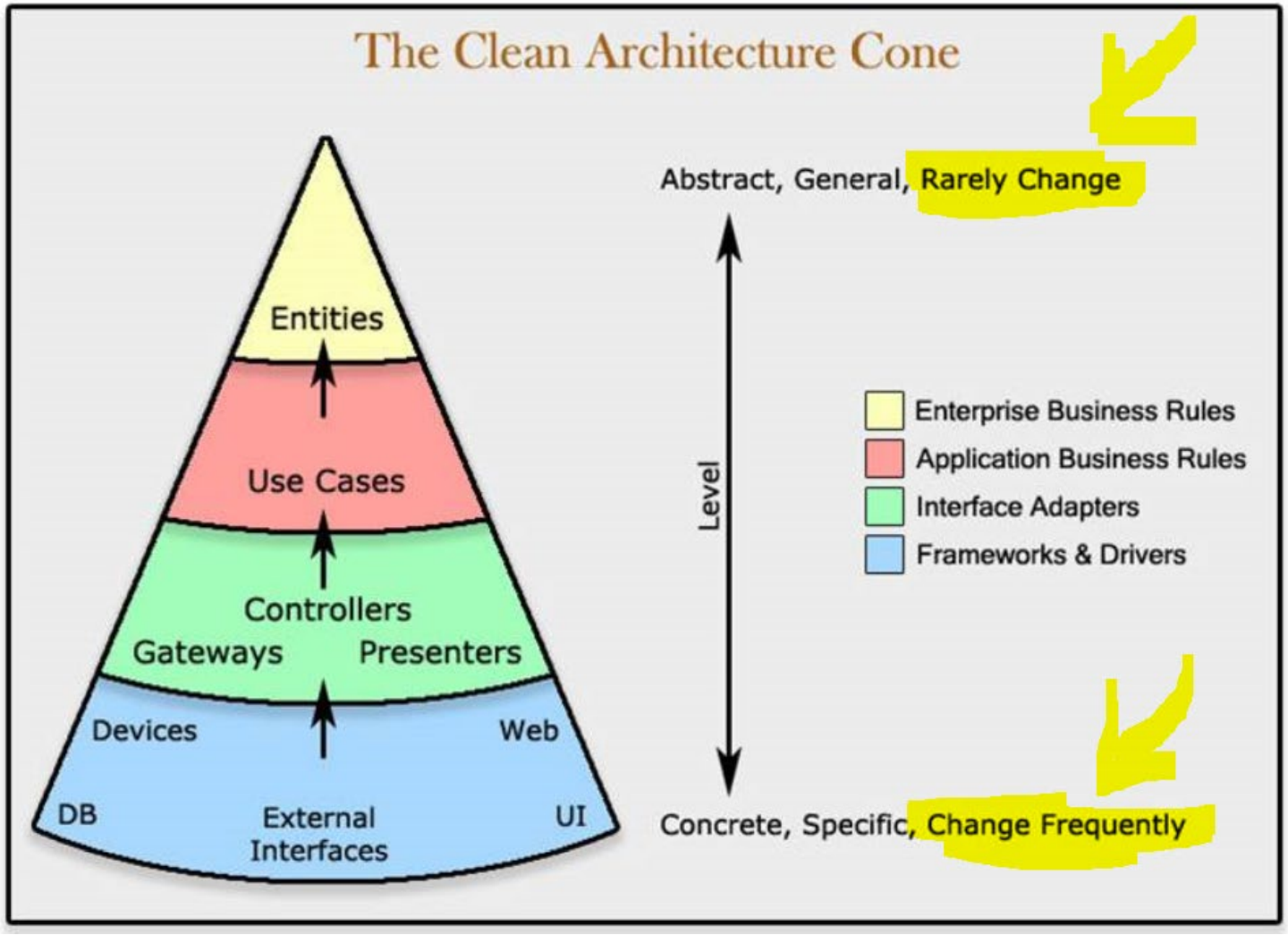
Documented in 2005 by Alistair Cockburn, Hexagonal Architecture is a software architecture that has many advantages and has seen renewed interest since 2015.

Produce systems that are

- Independent of frameworks
- Testable
- Independent of UI
- Independent of database
- Independent of any external agency

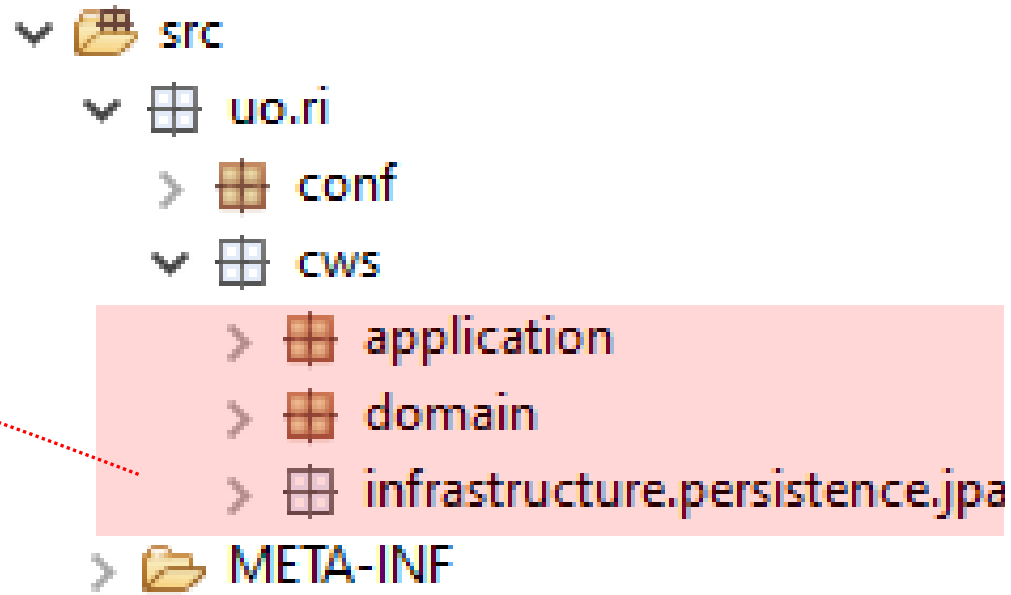
https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html

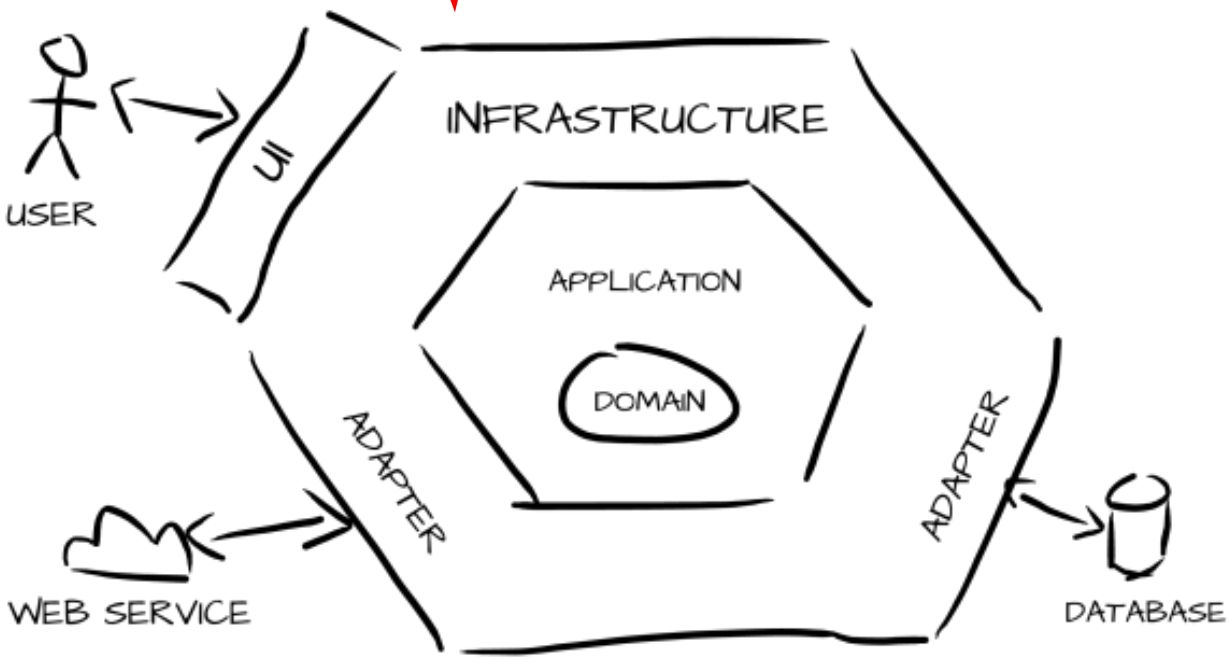
The dependencies cone



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

Changes on the design

Domain model package

- Supreme package
- Without any dependencies

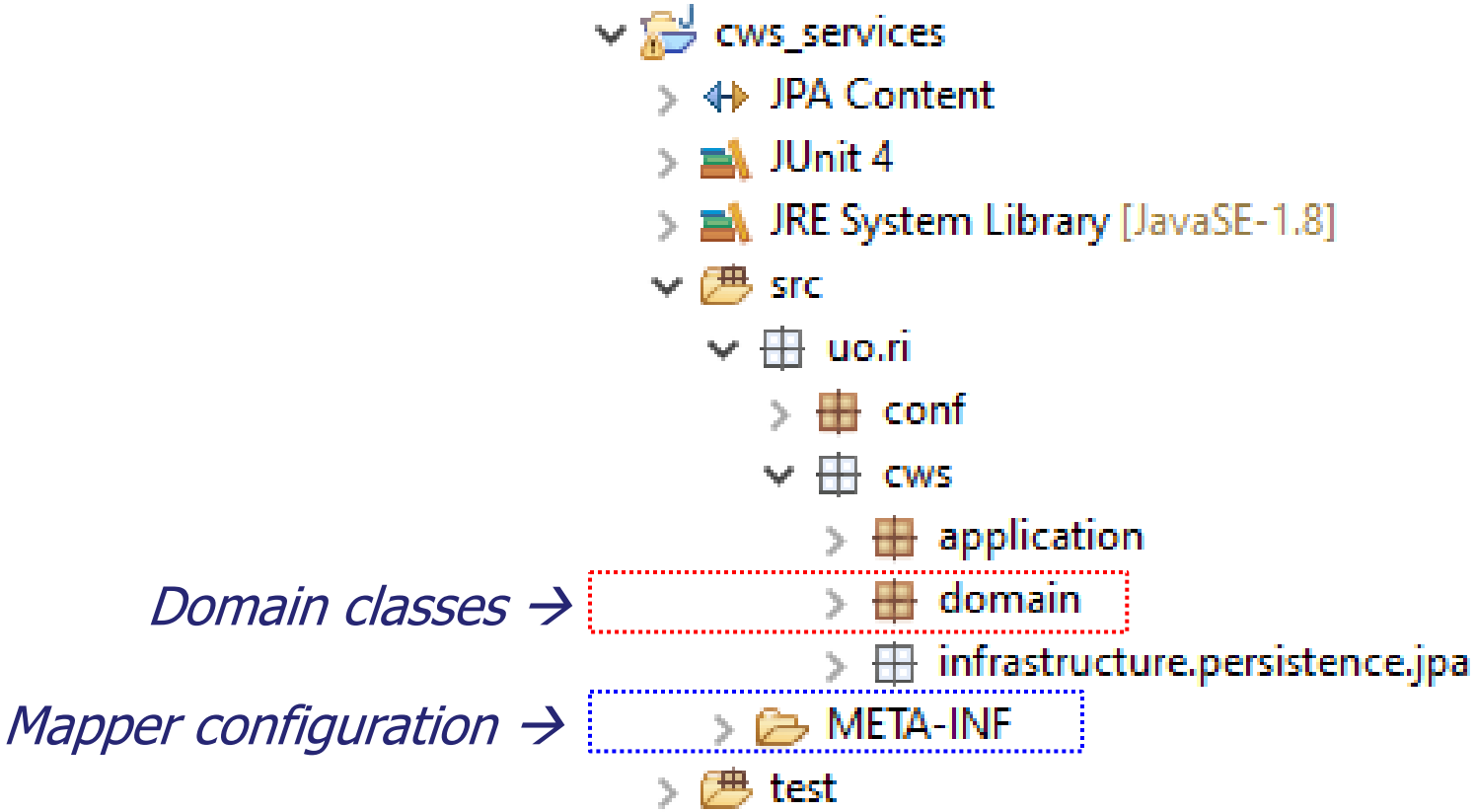
Application package

- Layer of services, only dependent on domain
- One service interface per use case
- Transaction Script refactored into command
- Throws BusinessException

Infraestructure, repositories

- Collection type interface
- Plus queries

Domain model package



Application

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

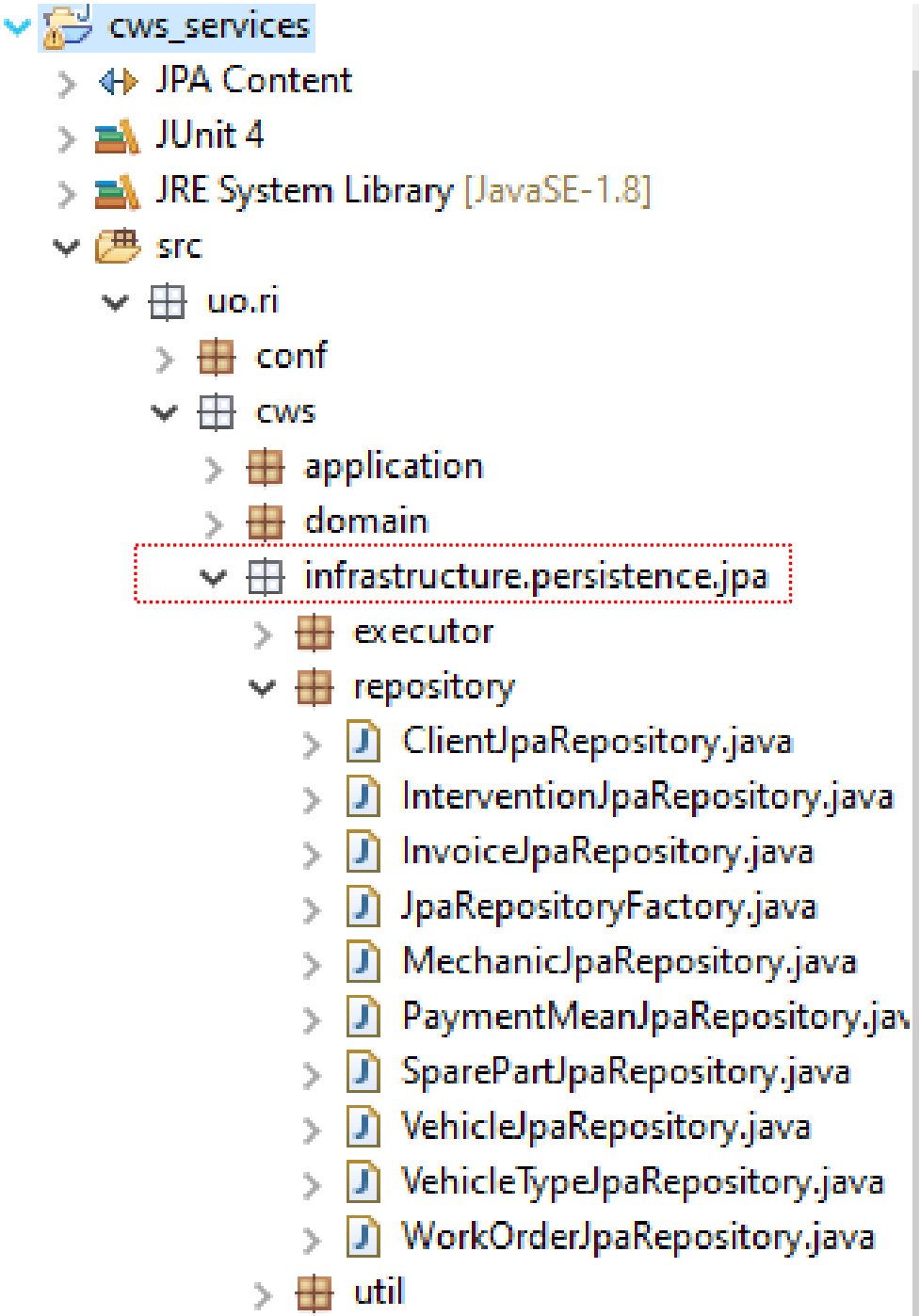
Interface and dto

- mechanic
  - crud ← *Use case*
    - command
      - AddMechanic.java
      - DeleteMechanic.java ← *Commands*
      - FindAllMechanics.java
      - FindMechanicById.java
      - UpdateMechanic.java
    - MechanicCrudServiceImpl.java ← *Façade*
  - MechanicCrudService.java
  - MechanicDto.java

- 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

Services →

Infrastructure package



Implementation of JPA repositories

Service Interfaces

One per use case

- Data interchange with DTO
- Throws BusinessException
- Use of 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

DTO pattern Data Transfer Object

Simple object

Just a data container, no logic

```
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;

}
```

As of JDK 14, we can replace our DTO class classes with records

```
public record ClientDto(
        String id,
        long version,
        String dni,
        String name,
        String surname,
        String phone,
        String email
    ) {}
```

TS refactored into Commands

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

In SL.TS.TDG_0

```
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);
    }
}
```

TS refactored into commands

```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;
}
```

With domain model

```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 {
```

← *SL.TS.TDG_0*

Example, create invoice

Now with domain model

```
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

Repositories

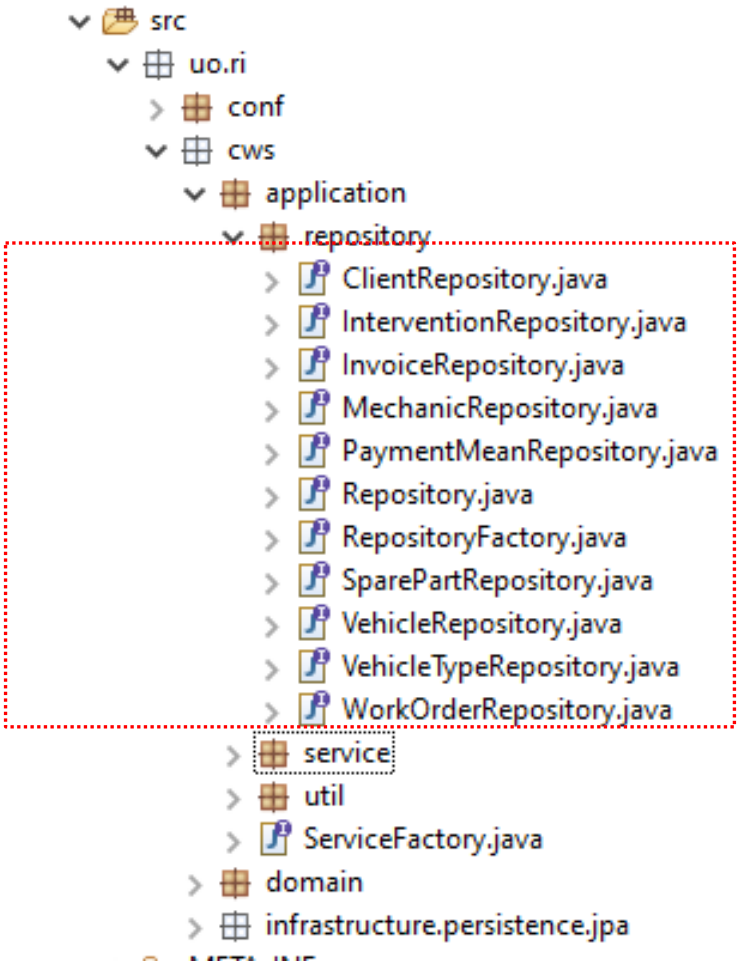
A store for objects
Collection like interface: add, remove

There is not update!!!

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

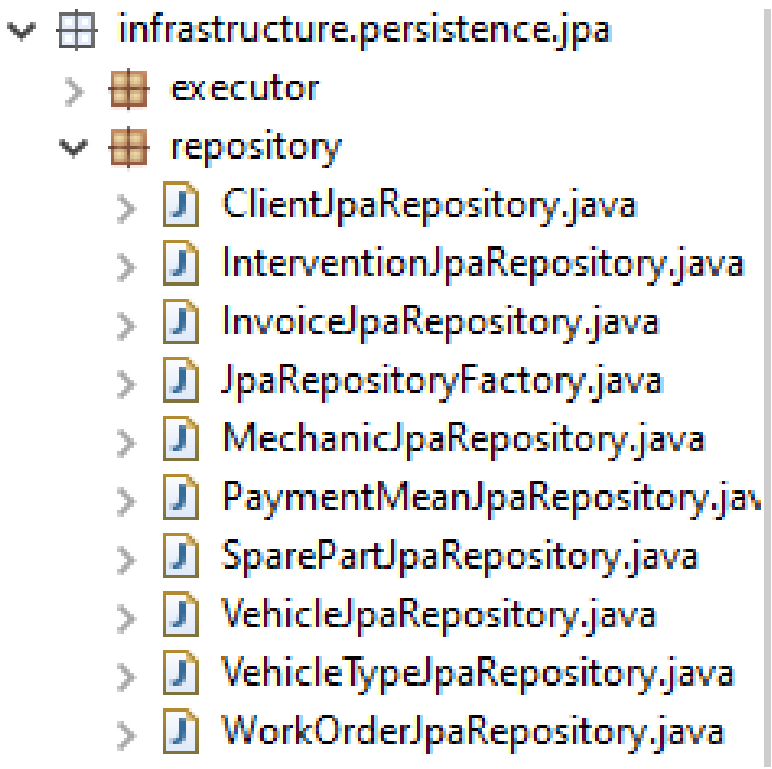
+ queries

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

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



Infrastructure: repositories impl

Repositories solve query and collection methods using the mapper




```
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();
        }

}
```

Going further…

Transaction control centralization

- And any other aspect, if needed: access control, auditoring, etc.

Remove repetitive code from the Transaction Scripts

Remove repetitive code

```
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;
    }

}
```

Remove repetitive code

```
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;
    }

}
```

Compare with the previous… what changes?

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;
    }

}
```

Repetitive code must be factored out

Transaction centralization

Steps:

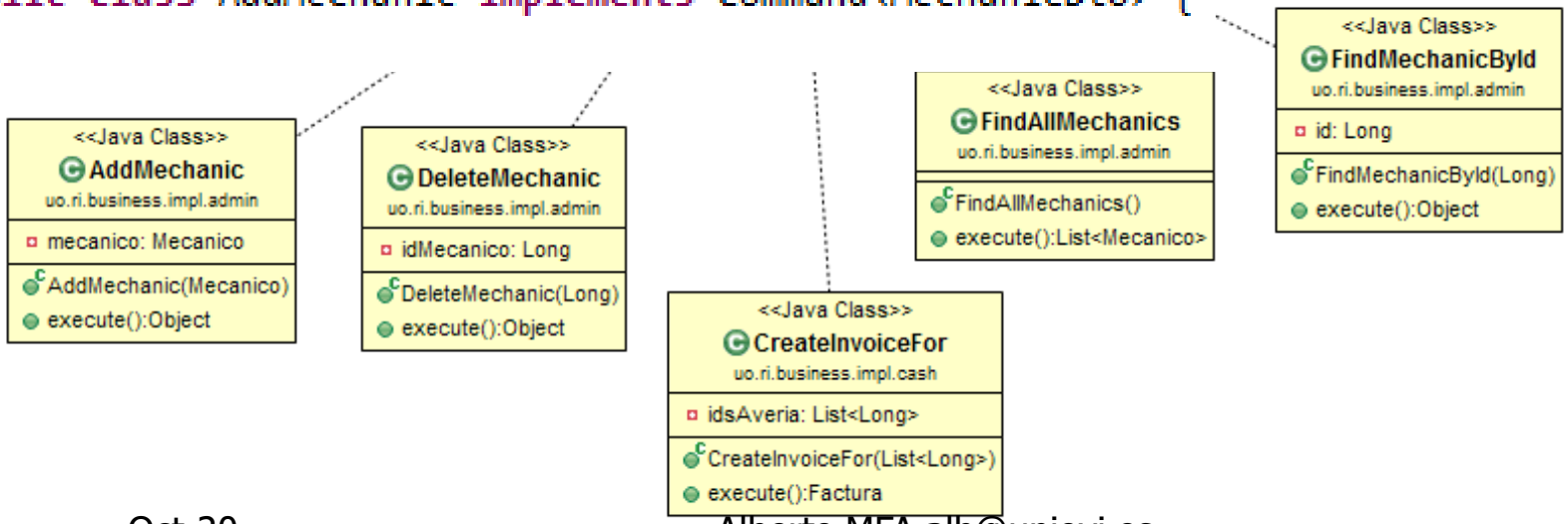
- Standardize the Transaction Script
  - Command interface
- Extract the trx control to a unique class
  - Command Executor
- Adapt the façades
  - MechanicCrudServiceImpl, etc.
- We need an utility class to access an EntityManager

Uniformizar los TS

```
public interface Command<T> {

    T execute() throws BusinessException;
}
```

All commands implement the Command interface


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



Extract control to a unique class

```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();
            }
        }
    }
}
```

Transactions and exception handling now centralized

```java
public interface CommandExecutor {

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

}
```

Modify façade implementation

Execute commands through the command executor

```
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) );
    }
```

Utility class for the EntityManager

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

createEntityManager()

- Creates a new EntityManager
- Only called from the Command Executor

getManager()

- Called from the repositories
- Returns the current persistence context

Utility class for the EntityManager

Used from

- Command executor
- Repositories

```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();
    }
```

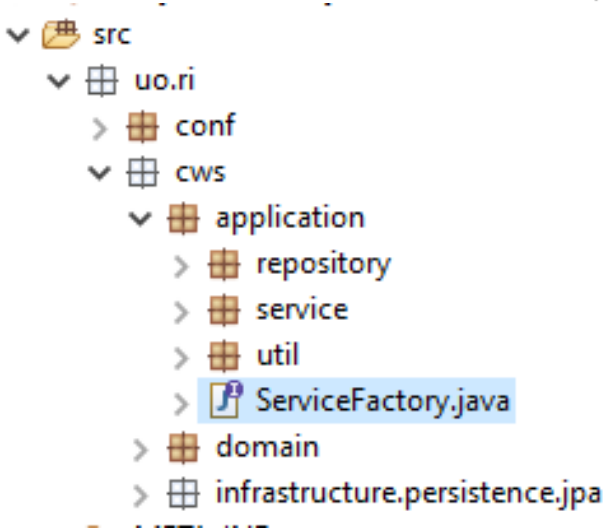
```java
public class JpaCommandExecutor implements CommandExecutor {

    @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;
```

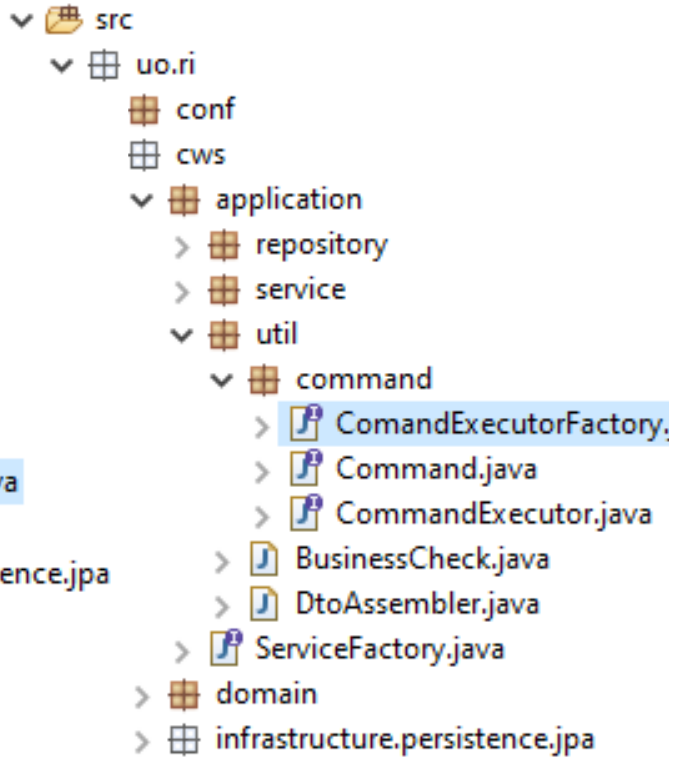
```java
public class AddMechanic implements Command<MechanicDto> {

    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;
    }
```

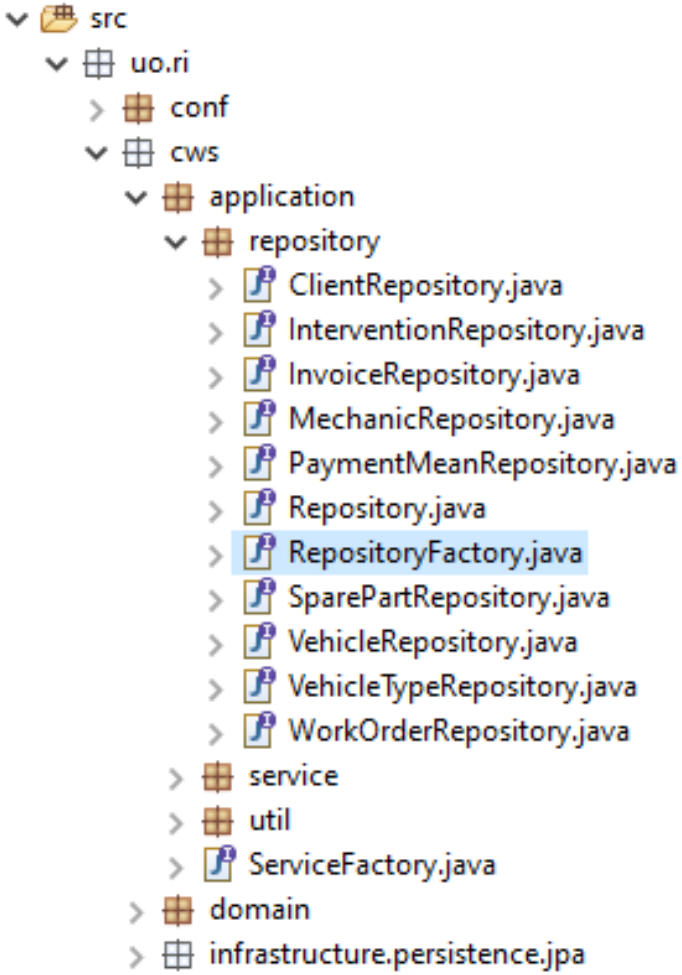
Neither duplicities, nor dependencies

```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 );
    }
```

Factories







Abstract Factory Pattern

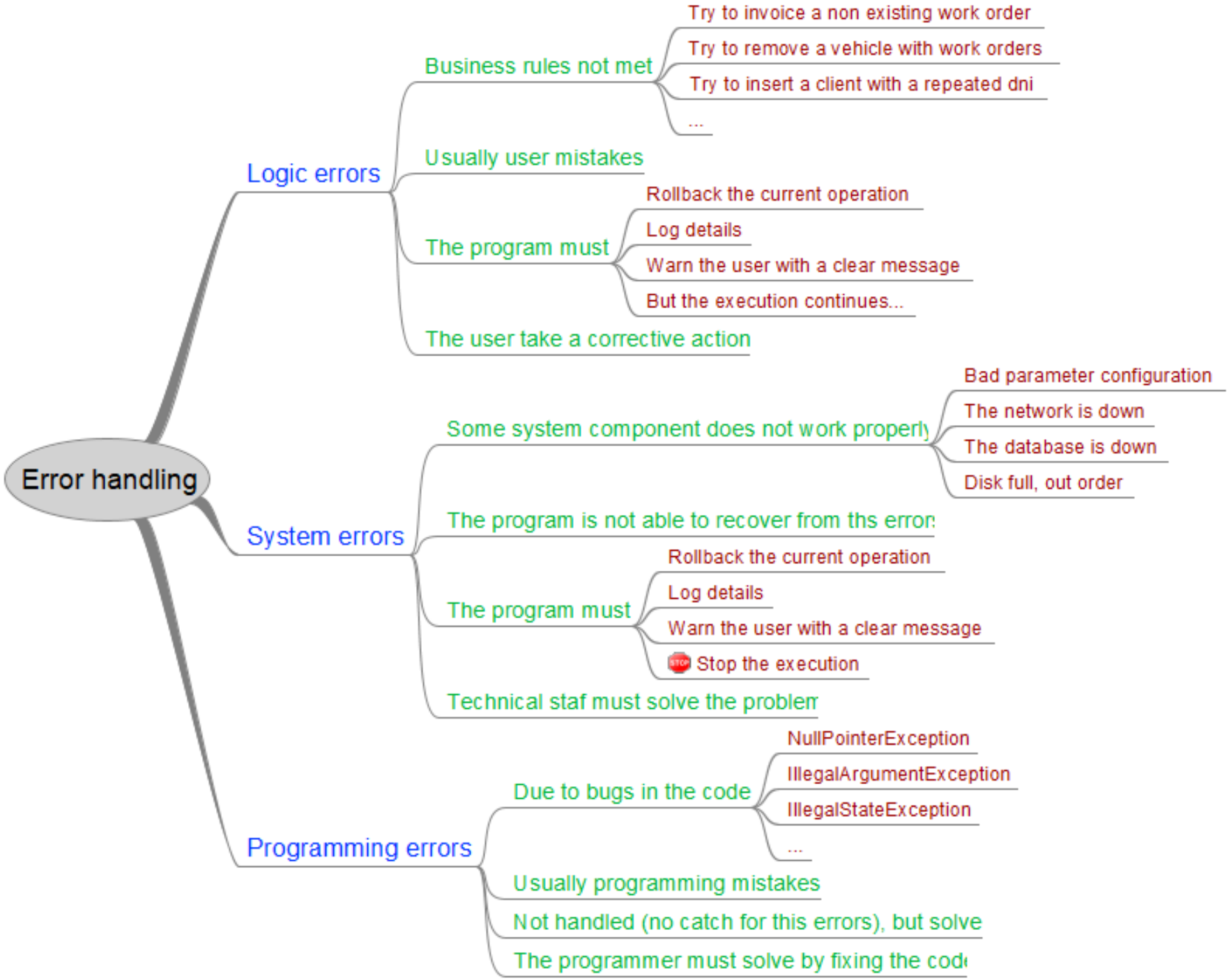
Factories and repository interfaces

A repository per entity (or aggregate)

```
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();

}
```

A service interface per use case


Error handling
Logic errors
Business rules not met
Try to invoice a non existing work order
Try to remove a vehicle with work orders
Try to insert a client with a repeated dni
...
Usually user mistakes
The program must
Rollback the current operation
Log details
Warn the user with a clear message
But the execution continues...
The user take a corrective action
System errors
Some system component does not work properly
Bad parameter configuration
The network is down
The database is down
Disk full, out order
The program is not able to recover from ths error
The program must
Rollback the current operation
Log details
Warn the user with a clear message
Stop the execution
Technical staf must solve the problem
Programming errors
Due to bugs in the code
NullPointerException
IllegalArgumentException
IllegalStateException
...
Usually programming mistakes
Not handled (no catch for this errors), but solve
The programmer must solve by fixing the code

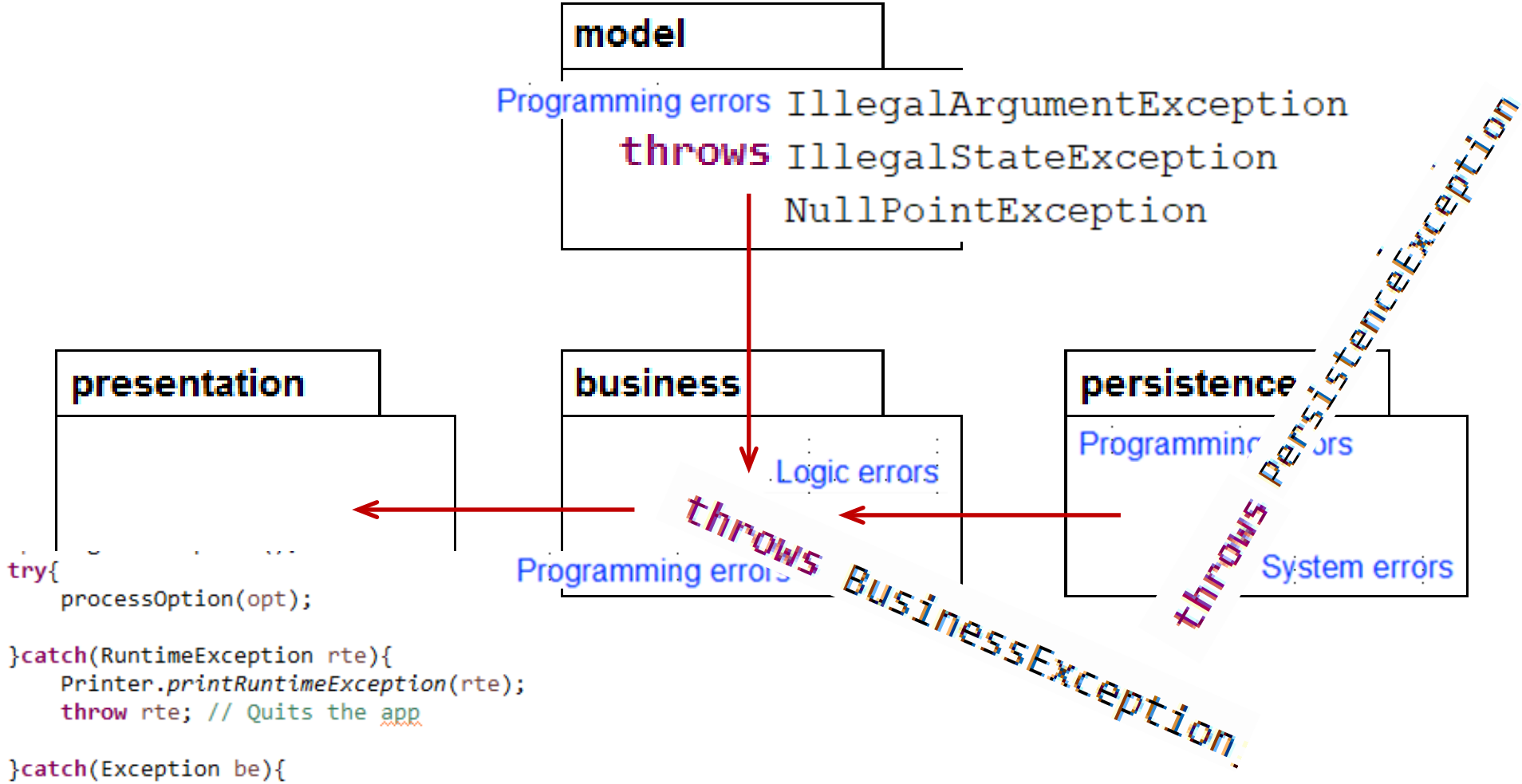
Error handling

```
throws BusinessException;
```

- Exceptions
  - Checked
    - Signal logic errors
    - We expect the client code reacts to the error
    - We force the programmer of the client code to think about the error
  - Unchecked
    - System errors
    - Programming errors

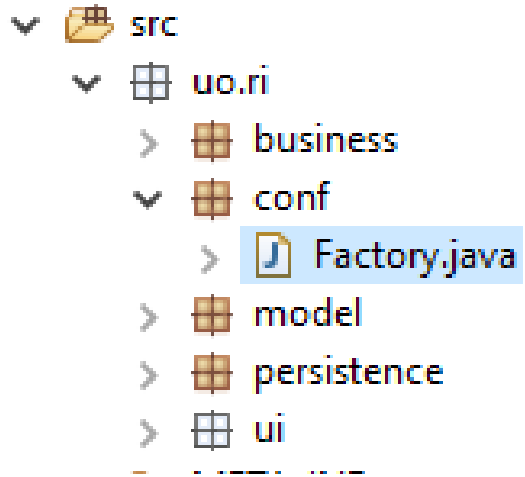
```
throw new PersistenceException();
```

Exception propagation map




```
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;
    }
```



The mission of main() is to configure the dependencies and start the application

Main & Configuration

```java
public class Factory {

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

}
```