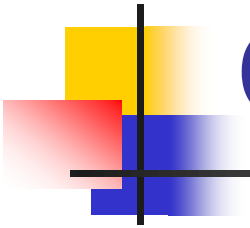
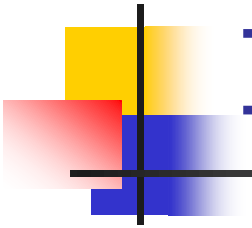
Diseñando la aplicación



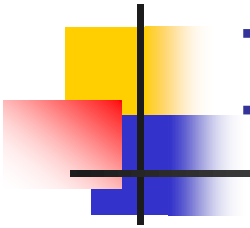
Contenido

- Introducción
- Mejorando nuestro diseño
  - Evoluciones de la aplicación



Introducción

- Hemos desarrollado una aplicación (muy sencilla)
  - ¿Es escalable?
  - ¿Es adaptable?
  - ¿Es fácilmente mantenible?
  - ¿El código es reutilizable?
  - ¿Es modulable?
  - ¿Acoplamiento?
  - ¿Cohesión?
- Si ahora quisiese que fuese una aplicación Web ¿sería fácil hacerlo? (¿implicaría muchos cambios?)



Introducción

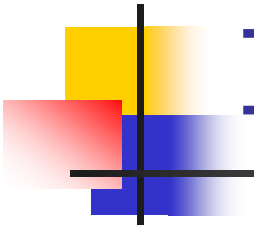
- Según vaya aumentando el tamaño y la complejidad de nuestra aplicación más necesario es hacer las cosas BIEN.
- Tenemos que buscar
  - Código legible
  - Código reutilizable
  - Escalable
  - Mantenibilidad
  - ……

Introducción

- El diseño OO es difícil. Y si pretendemos poder reutilizarlo (en parte) aún lo es más.
- No se trata de inventar la rueda →YA ESTÁ INVENTADA.
- Seguro que alguien se ha encontrado anteriormente ante un problema similar al nuestro y ya hay una solución adecuada.

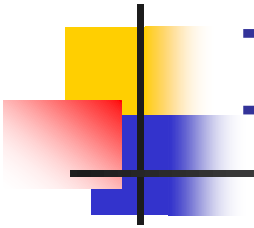
Introducción

- *Cada Patrón describe un problema que ocurre una y otra vez en nuestro entorno, así como la solución a ese problema, de tal forma que se pueda aplicar esta solución un millón de veces, sin hacer lo mismo dos veces (Christopher Alexander. Arquitecto….de edificios)*



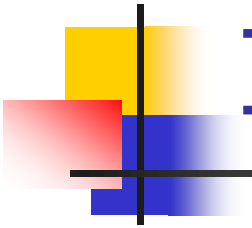
Introducción

- *Los patrones de diseño son descripciones de clases y objetos relacionados que están particularizados para resolver un problema de diseño general en un determinado contexto. (Gamma)*



Introducción

- Los patrones no son recetas que se aplican sin más. Es necesario adaptarlos a nuestro problema concreto (y elegir adecuadamente entre el catálogo de patrones)



Introducción

- Todas las aplicaciones empresariales comparten algunos rasgos:
  - Datos persistentes, muchos, con modelos complejos y en evolución
  - Acceso concurrente a los datos persistentes
  - Interfaces de usuario complejas
  - La necesidad de integrarse con otras aplicaciones
  - Lógica de negocio compleja, habitualmente con casos específicos para clientes específicos

Introducción

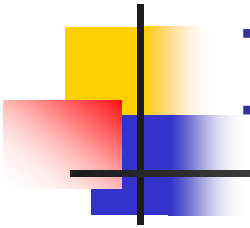
- Debido a estas similitudes han surgido una serie de patrones de arquitectura que pretenden facilitar, homogeneizar, el desarrollo de estas aplicaciones.
- Aún así es necesario personalizar profundamente estos patrones para aplicarlos en nuestro contexto.

Introducción

- Quizá la estructura más común es la separación en capas
  - Las capas de niveles superiores tienen dependencias de las capas inferiores
  - Las capas inferiores no dependen de las capas superiores

Introducción

- La arquitectura en 3 capas es el estándar de facto en el desarrollo de aplicaciones empresariales
  - Capa de presentación
    - Interfaz con el usuario
  - Capa de negocio
    - Implementa la lógica del negocio, las reglas que deben cumplirse
  - Capa de datos
    - Se comunica con otros sistemas para enviar y/o recibir información
    - Normalmente un SGBD para conseguir la persistencia



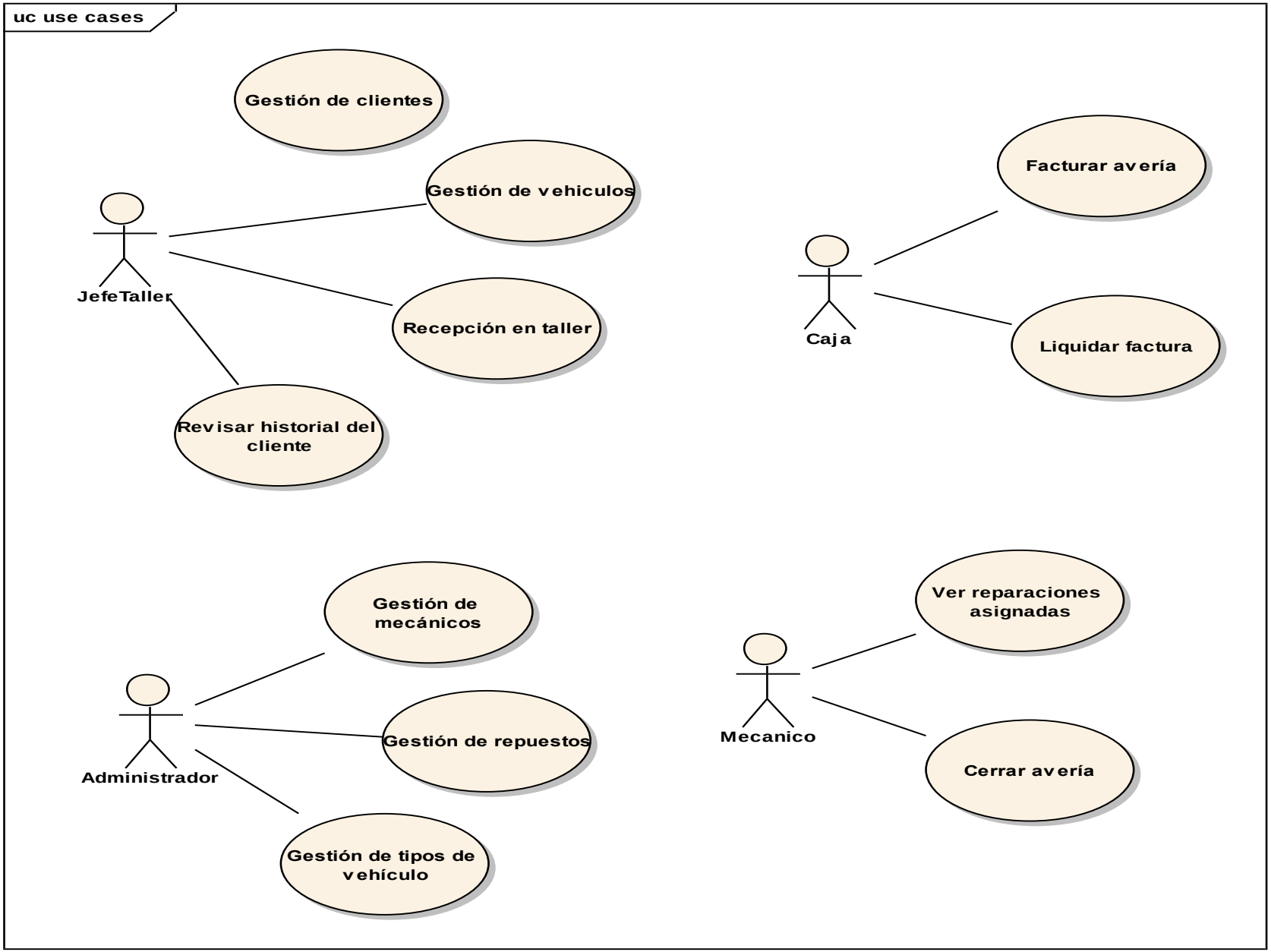
Introducción

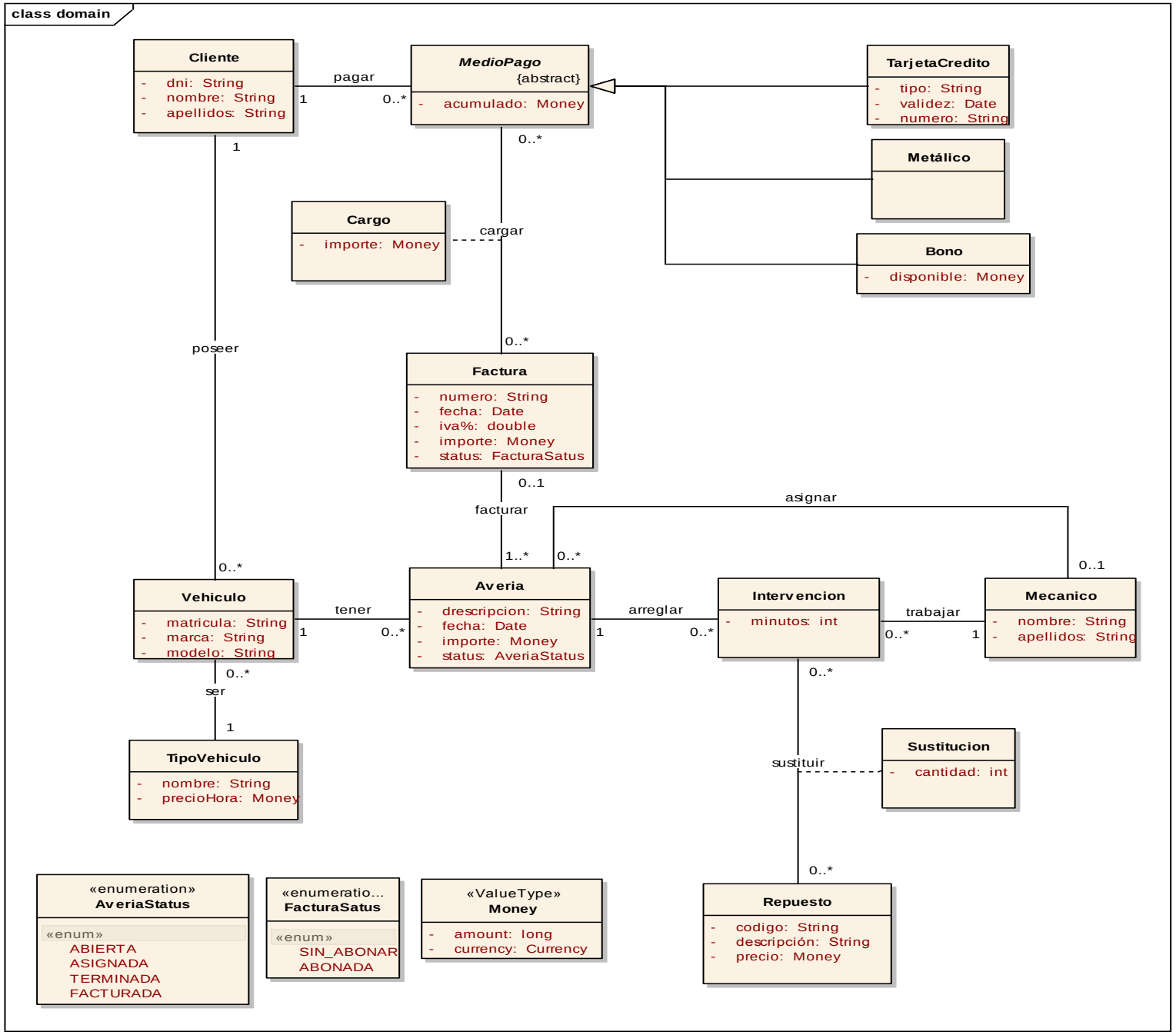
- Tenéis dos asignaturas dedicadas a estas materias donde os ensañarán en profundidad cómo hacer correctamente las cosas
  - Diseño del Software
  - Arquitectura del Software

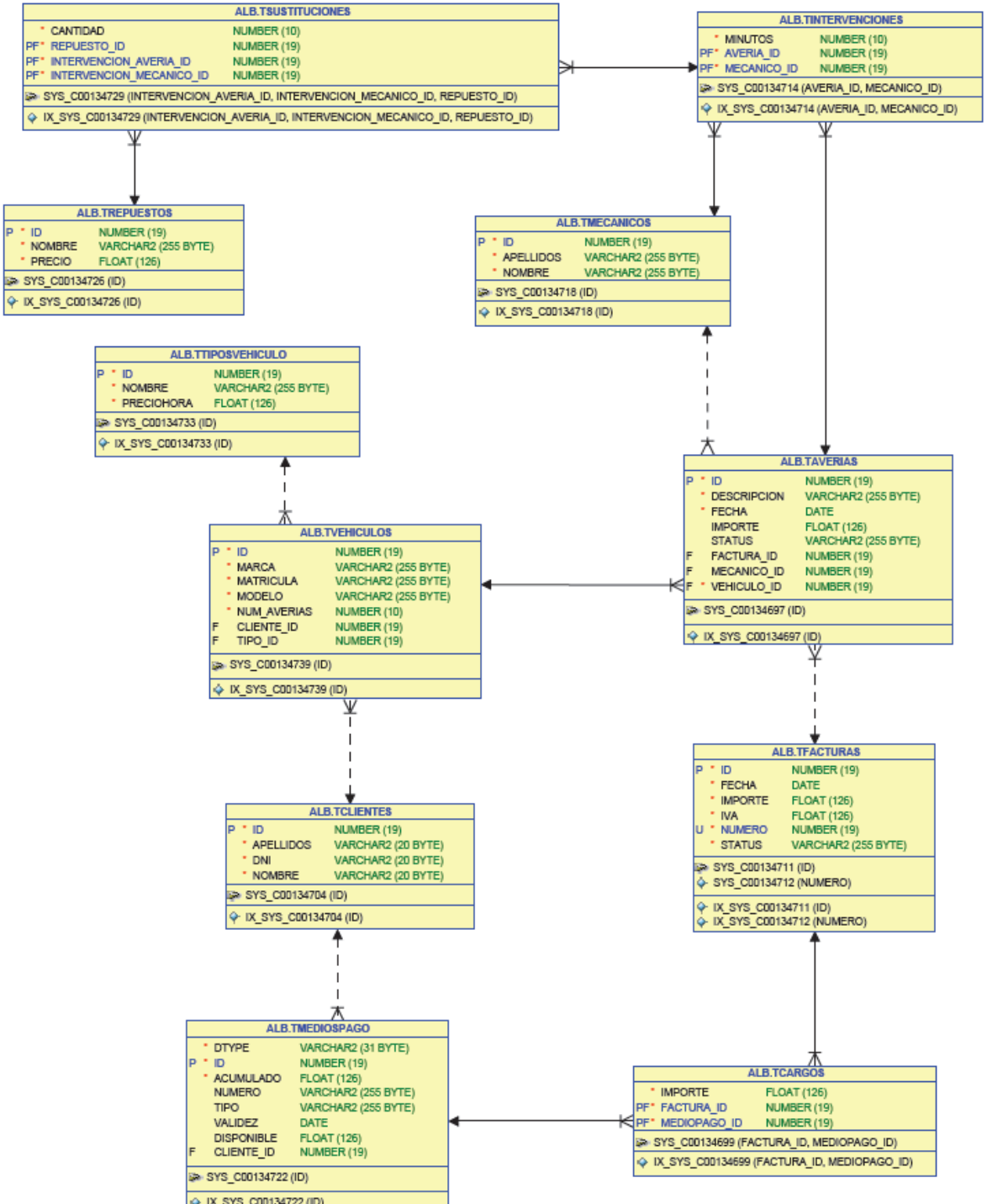
Mejorando nuestro diseño

Mejoramos el diseño

- Vamos a ir refinando la aplicación de referencia (CarWorkShop) paulatinamente de tal manera que al finalizar tengamos una aplicación bien diseñada.
- Partimos de la Versión 0→V0, la cual ha sido desarrollada sin tener en cuenta ninguna buena práctica.

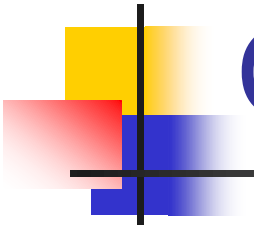

uc use cases
Gestión de clientes
Gestión de vehiculos
JefeTaller
Recepción en taller
Revisar historial del cliente
Facturar avería
Caja
Liquidar factura
Gestión de mecánicos
Administrador
Gestión de repuestos
Gestión de tipos de vehículo
Ver reparaciones asignadas
Mecanico
Cerrar avería


class domain
Cliente
- dni: String
- nombre: String
- apellidos: String
pagar
1
0..*
MedioPago
{abstract}
- acumulado: Money
TarjetaCredito
- tipo: String
- validez: Date
- numero: String
Metálico
Bono
- disponible: Money
Cargo
- importe: Money
cargar
poseer
Factura
- numero: String
- fecha: Date
- iva%: double
- importe: Money
- status: FacturaSatus
facturar
asignar
Vehiculo
- matricula: String
- marca: String
- modelo: String
tener
Averia
- drescripcion: String
- fecha: Date
- importe: Money
- status: AveriaStatus
arreglar
Intervencion
- minutos: int
trabajar
Mecanico
- nombre: String
- apellidos: String
ser
TipoVehiculo
- nombre: String
- precioHora: Money
sustituir
Sustitucion
- cantidad: int
Repuesto
- codigo: String
- descripción: String
- precio: Money
«enumeration»
AveriaStatus
«enum»
ABIERTA
ASIGNADA
TERMINADA
FACTURADA
«enumeratio...
FacturaSatus
«enum»
SIN_ABONAR
ABONADA
«ValueType»
Money
- amount: long
- currency: Currency


ALB.TSUSTITUCIONES
* CANTIDAD NUMBER (10)
PF* REPUESTO_ID NUMBER (19)
PF* INTERVENCION_AVERIA_ID NUMBER (19)
PF* INTERVENCION_MECANICO_ID NUMBER (19)
SYS_C00134729 (INTERVENCION_AVERIA_ID, INTERVENCION_MECANICO_ID, REPUESTO_ID)
IX_SYS_C00134729 (INTERVENCION_AVERIA_ID, INTERVENCION_MECANICO_ID, REPUESTO_ID)
ALB.TINTERVENCIONES
* MINUTOS NUMBER (10)
PF* AVERIA_ID NUMBER (19)
PF* MECANICO_ID NUMBER (19)
SYS_C00134714 (AVERIA_ID, MECANICO_ID)
IX_SYS_C00134714 (AVERIA_ID, MECANICO_ID)
ALB.TREPUESTOS
P * ID NUMBER (19)
* NOMBRE VARCHAR2 (255 BYTE)
* PRECIO FLOAT (126)
SYS_C00134726 (ID)
IX_SYS_C00134726 (ID)
ALB.TMECANICOS
P * ID NUMBER (19)
* APELLIDOS VARCHAR2 (255 BYTE)
* NOMBRE VARCHAR2 (255 BYTE)
SYS_C00134718 (ID)
IX_SYS_C00134718 (ID)
ALB.TTIPOSVEHICULO
P * ID NUMBER (19)
* NOMBRE VARCHAR2 (255 BYTE)
* PRECIOHORA FLOAT (126)
SYS_C00134733 (ID)
IX_SYS_C00134733 (ID)
ALB.TAVERIAS
P * ID NUMBER (19)
* DESCRIPCION VARCHAR2 (255 BYTE)
* FECHA DATE
IMPORTE FLOAT (126)
STATUS VARCHAR2 (255 BYTE)
F FACTURA_ID NUMBER (19)
F MECANICO_ID NUMBER (19)
F * VEHICULO_ID NUMBER (19)
SYS_C00134697 (ID)
IX_SYS_C00134697 (ID)
ALB.TVEHICULOS
P * ID NUMBER (19)
* MARCA VARCHAR2 (255 BYTE)
* MATRICULA VARCHAR2 (255 BYTE)
* MODELO VARCHAR2 (255 BYTE)
* NUM_AVERIAS NUMBER (10)
F CLIENTE_ID NUMBER (19)
F TIPO_ID NUMBER (19)
SYS_C00134739 (ID)
IX_SYS_C00134739 (ID)
ALB.TFACTURAS
P * ID NUMBER (19)
* FECHA DATE
* IMPORTE FLOAT (126)
* IVA FLOAT (126)
U * NUMERO NUMBER (19)
* STATUS VARCHAR2 (255 BYTE)
SYS_C00134711 (ID)
SYS_C00134712 (NUMERO)
IX_SYS_C00134711 (ID)
IX_SYS_C00134712 (NUMERO)
ALB.TCLIENTES
P * ID NUMBER (19)
* APELLIDOS VARCHAR2 (20 BYTE)
* DNI VARCHAR2 (20 BYTE)
* NOMBRE VARCHAR2 (20 BYTE)
SYS_C00134704 (ID)
IX_SYS_C00134704 (ID)
ALB.TMEDIOSPAGO
* DTYPE VARCHAR2 (31 BYTE)
P * ID NUMBER (19)
* ACUMULADO FLOAT (126)
NUMERO VARCHAR2 (255 BYTE)
TIPO VARCHAR2 (255 BYTE)
VALIDEZ DATE
DISPONIBLE FLOAT (126)
F CLIENTE_ID NUMBER (19)
SYS_C00134722 (ID)
ALB.TCARGOS
* IMPORTE FLOAT (126)
PF* FACTURA_ID NUMBER (19)
PF* MEDIOPAGO_ID NUMBER (19)
SYS_C00134699 (FACTURA_ID, MEDIOPAGO_ID)
IX_SYS_C00134699 (FACTURA_ID, MEDIOPAGO_ID)

Mejoramos el diseño

- Nos vamos a centrar en unos casos de uso específicos de tal manera que podamos ver su evolución según vamos mejorando el diseño
  - CRUD de mecánicos
  - Procesos de Caja

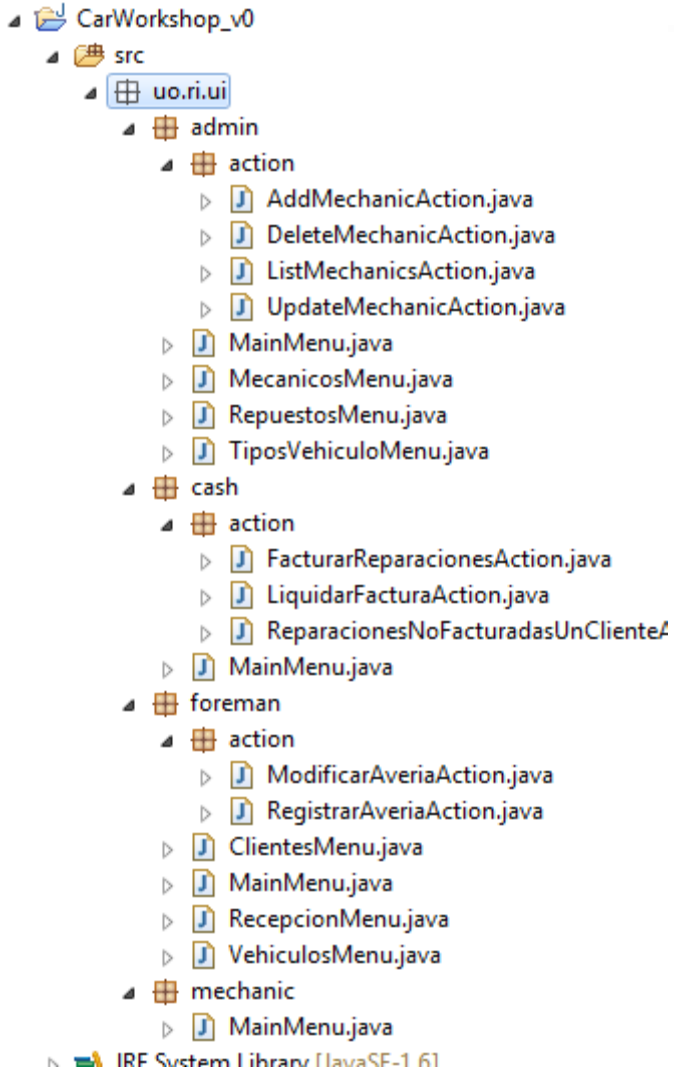


CarWorkShop V0

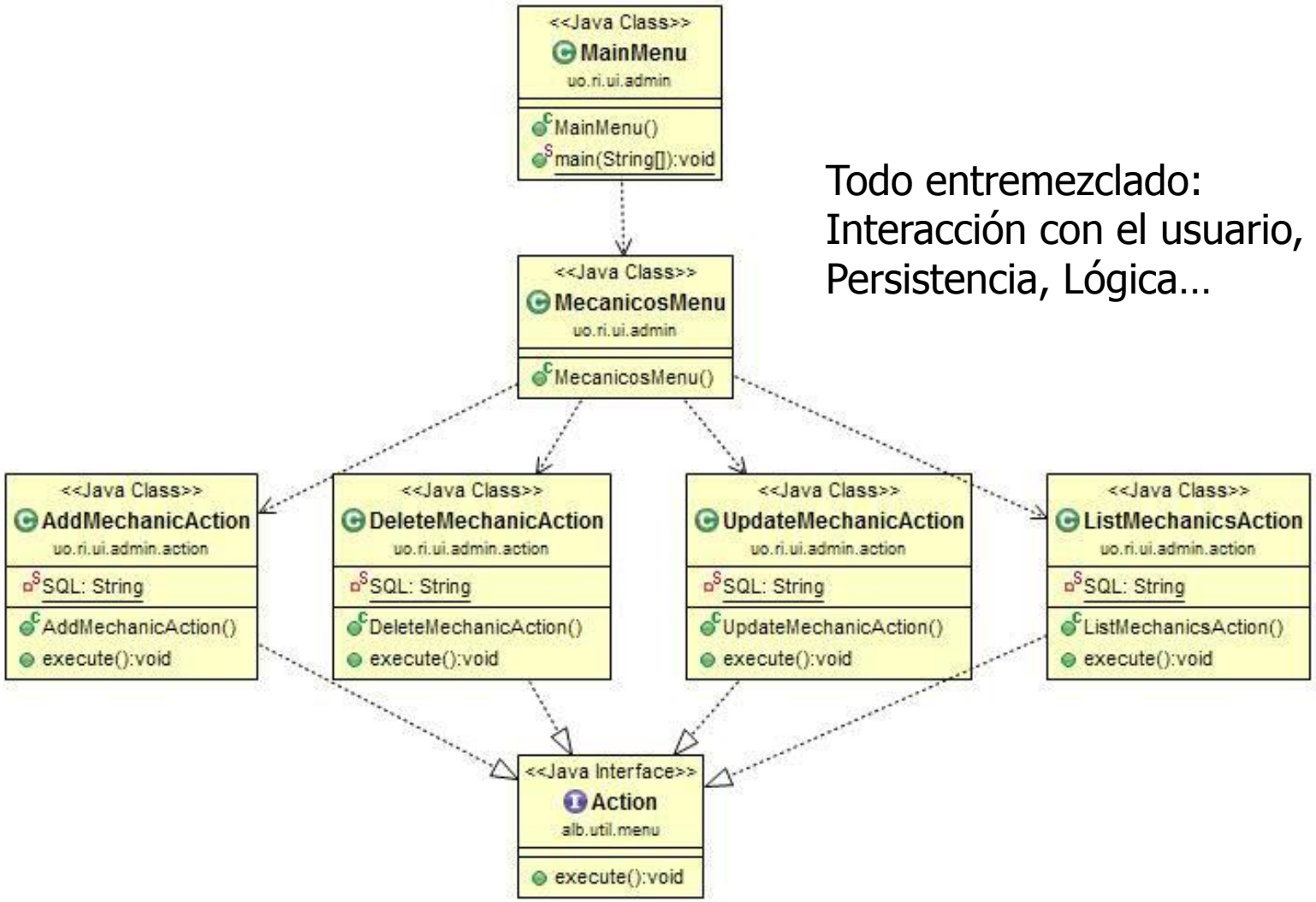
- Implementada en una sola capa
- La presentación se realiza mediante consola con menús de texto
- Cierta infraestructura de menús
- En cada clase se hace
  - Interacción con el usuario
  - Lógica
  - Persistencia
- Es un ejemplo de como **NO** se debe hacer

CarWorkShop V0

- Estructura de paquetes
  - Un paquete por cada actor
  - En cada paquete se encuentran los menús adecuados y un subpaquete *action* que incluye las clases que realizan las operaciones



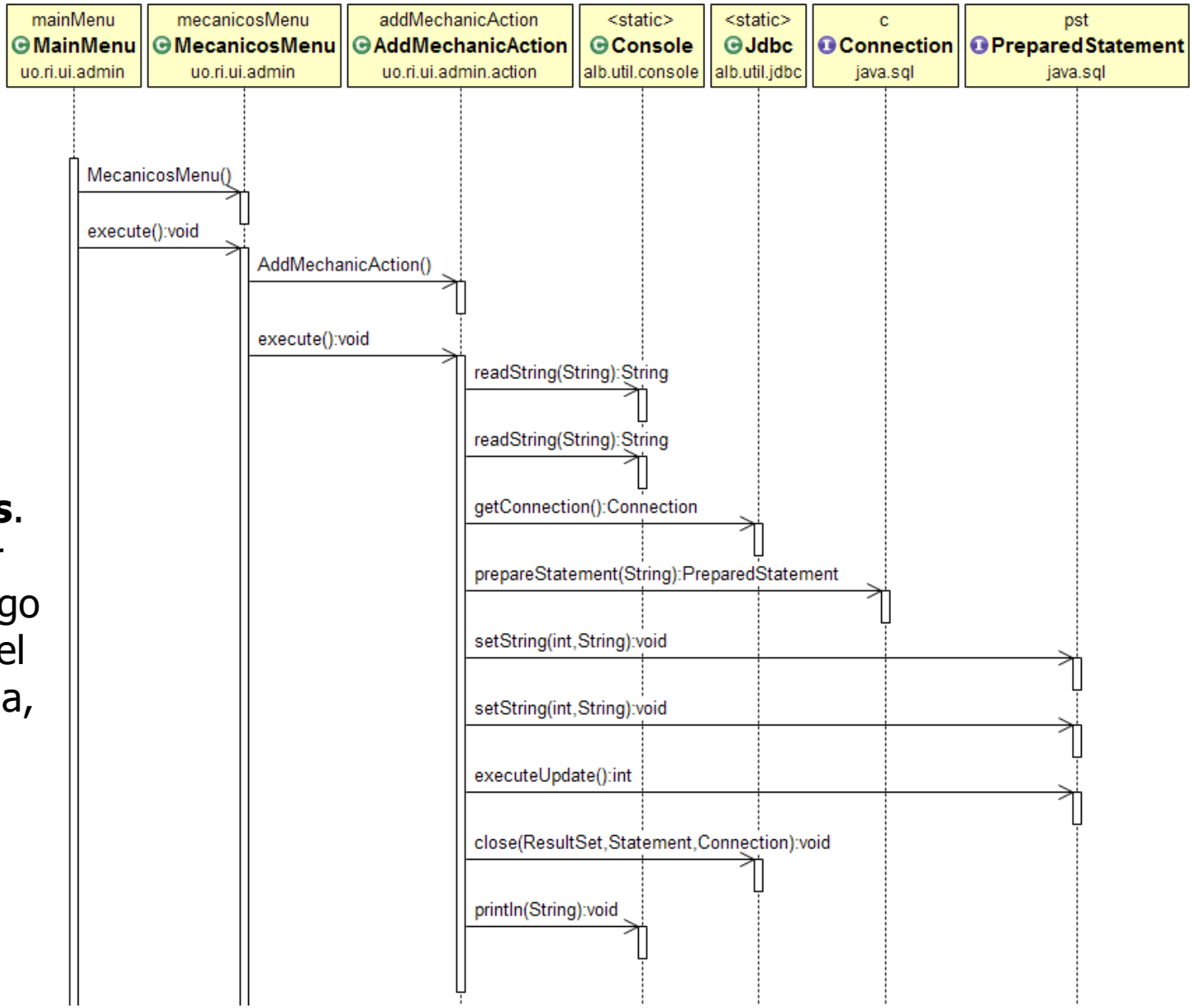
V0 Mecánicos



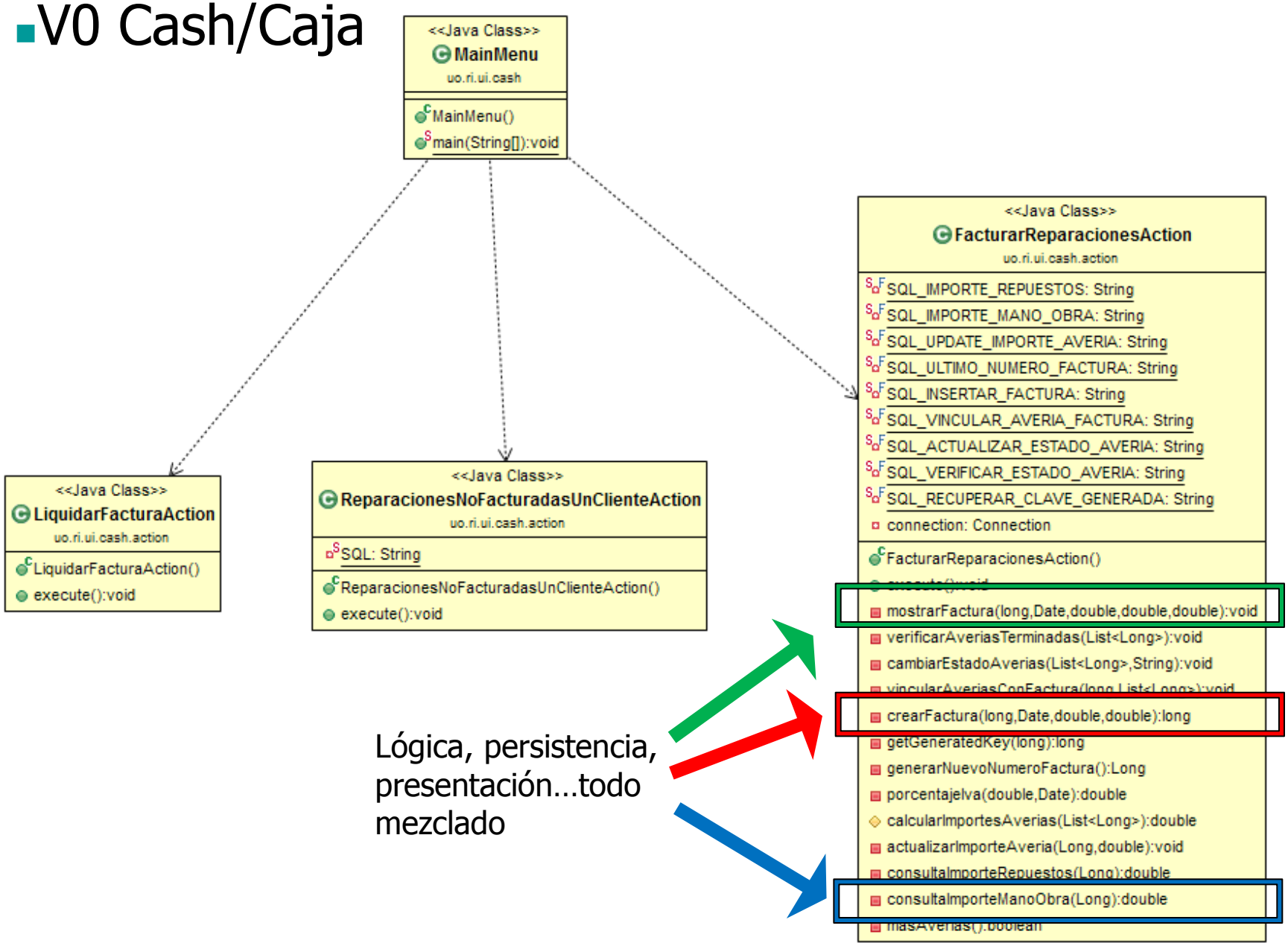
Todo entremezclado:
Interacción con el usuario,
Persistencia, Lógica…

V0 Mecánicos

Diagrama de secuencia para **añadir mecánicos**. Como se puede ver está mezclado código de interacción con el usuario, persistencia, etc.



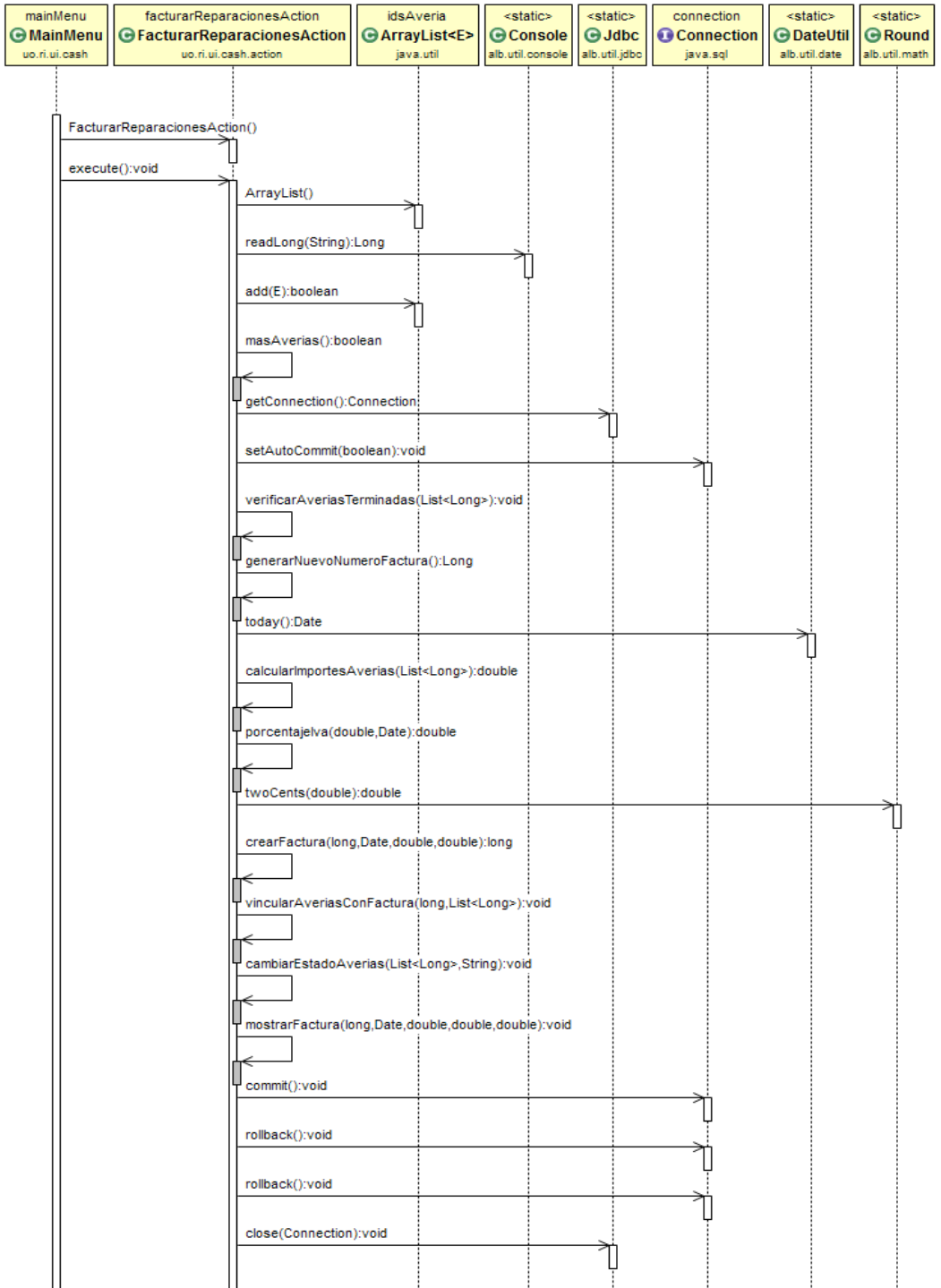
V0 Cash/Caja


<<Java Class>>
MainMenu
uo.ri.ui.cash
MainMenu()
main(String[]):void
<<Java Class>>
LiquidarFacturaAction
uo.ri.ui.cash.action
LiquidarFacturaAction()
execute():void
<<Java Class>>
ReparacionesNoFacturadasUnClienteAction
uo.ri.ui.cash.action
SQL: String
ReparacionesNoFacturadasUnClienteAction()
execute():void
<<Java Class>>
FacturarReparacionesAction
uo.ri.ui.cash.action
SQL_IMPORTE_REPUESTOS: String
SQL_IMPORTE_MANO_OBRA: String
SQL_UPDATE_IMPORTE_AVERIA: String
SQL_ULTIMO_NUMERO_FACTURA: String
SQL_INSERTAR_FACTURA: String
SQL_VINCULAR_AVERIA_FACTURA: String
SQL_ACTUALIZAR_ESTADO_AVERIA: String
SQL_VERIFICAR_ESTADO_AVERIA: String
SQL_RECUPERAR_CLAVE_GENERADA: String
connection: Connection
FacturarReparacionesAction()
mostrarFactura(long,Date,double,double,double):void
verificarAveriasTerminadas(List<Long>):void
cambiarEstadoAverias(List<Long>,String):void
crearFactura(long,Date,double,double):long
getGeneratedKey(long):long
generarNuevoNumeroFactura():Long
porcentajeIva(double,Date):double
calcularImportesAverias(List<Long>):double
actualizarImporteAveria(Long,double):void
consultaImporteRepuestos(Long):double
consultaImporteManoObra(Long):double
Lógica, persistencia, presentación…todo mezclado

V0 Cash/Caja

Diagrama de secuencia para **crear una factura**.

Como se puede ver está mezclado código de interacción con el usuario, persistencia, lógica, validaciones, etc.



V0 Cash/Caja

Método execute() de FacturarReparacionesAction

```
do {
Long id = Console.readLong("ID de averia");
idsAveria.add(id);
} while ( masAverias() );
try {
connection = Jdbc.getConnection();
connection.setAutoCommit(false);
verificarAveriasTerminadas(idsAveria);
long numeroFactura = generarNuevoNumeroFactura();
Date fechaFactura = DateUtil.today();
double totalFactura = calcularImportesAverias(idsAveria);
double iva = porcentajeIva(totalFactura, fechaFactura);
double importe = totalFactura * (1 + iva/100);
importe = Round.twoCents(importe);
long idFactura = crearFactura(numeroFactura, fechaFactura, iva, importe);
vincularAveriasConFactura(idFactura, idsAveria);
cambiarEstadoAverias(idsAveria, "FACTURADA");
mostrarFactura(numeroFactura, fechaFactura, totalFactura, iva, importe);
connection.commit();
}
catch (SQLException e) {
try { connection.rollback(); } catch (SQLException ex) {};
throw new RuntimeException(e);
}
catch (BusinessException e) {
try { connection.rollback(); } catch (SQLException ex) {};
throw e;
}
finally {
Jdbc.close(connection);
}
```

CarWorkShop V0

- Como se puede ver todo el código está enmarañado y diseminado.
  - Si quiero cambiar el sistema de persistencia tengo que tocar (casi) todas las clases
  - Si quiero cambiar la interacción con el usuario tengo que tocar (casi) todas las clases
  - Si quiero reutilizar código (por ejemplo facturación) tengo que modificarlo profundamente para adaptarlo a la presentación y persistencia que se desee
- Difícil de mantener, modificar, ampliar, entender….

CarWorkShop V1

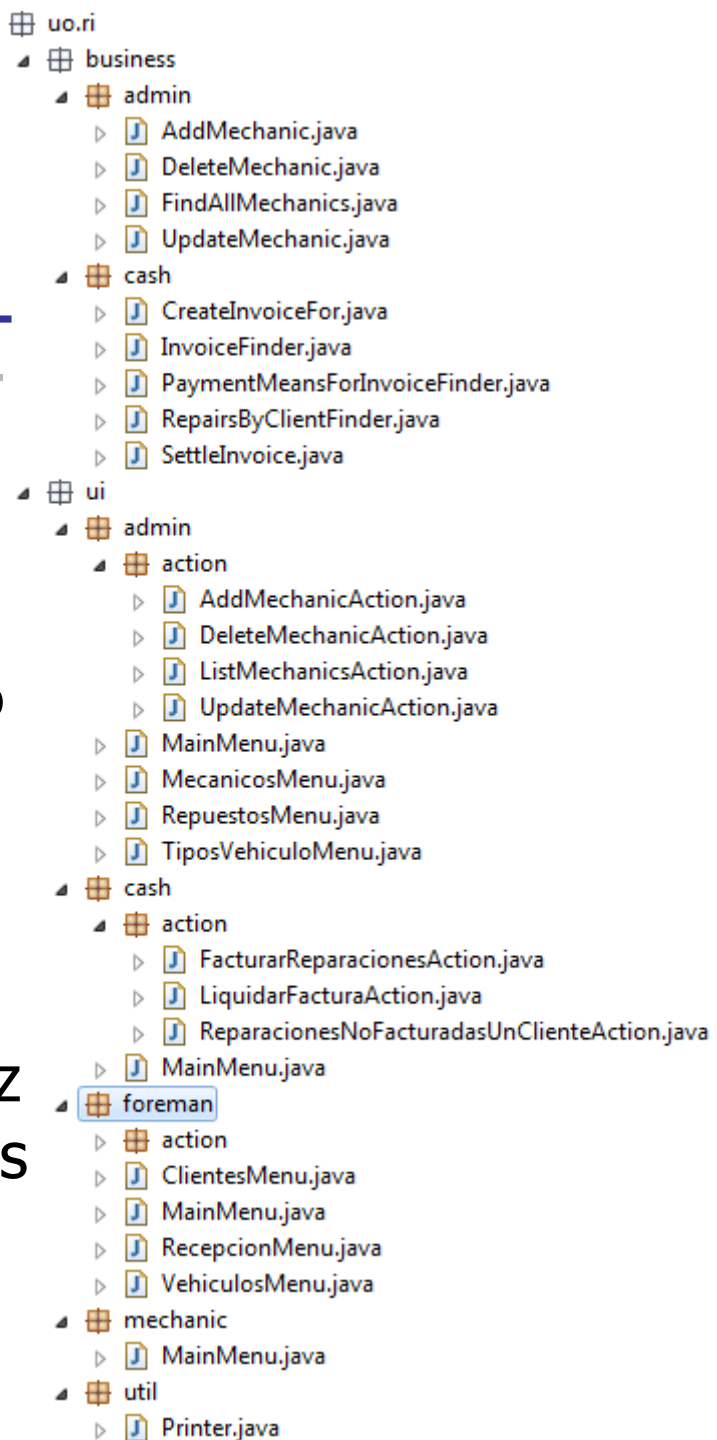
- En un primer refinamiento vamos a separar la capa de presentación del resto
  - Capa de presentación (pantalla+teclado+listados+validaciones)
  - Capa de lógica+persistencia
- Para pasar argumentos se utilizan Maps (interface Map<k,v>) o listas de Maps
- Se redistribuyen los paquetes (aparecen nuevos)

CarWorkShop V1

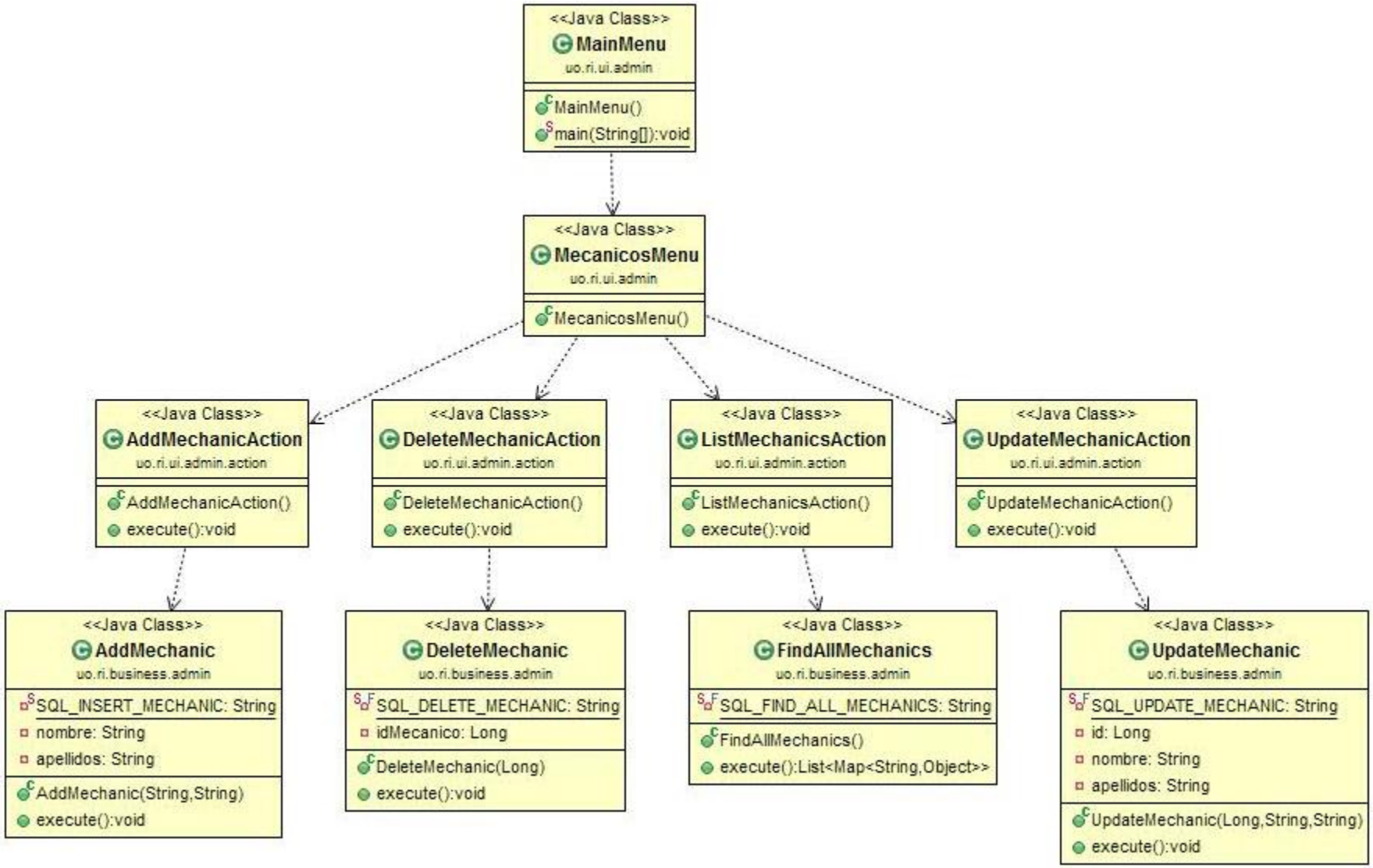
- Algunas operaciones han tenido que ser modificadas al separar interacción del proceso
- Aparece clase Printer para mostrar por consola con formato. Puesto que aparece en varios sitios se factoriza a un lugar común.
- Todas las clases de lógica (+persistencia) tienen apariencia similar
  - Reciben sus datos en el constructor
  - Tienen método execute() que realiza la tarea
  - Vamos acercándonos al patrón Comando (Gof) (también se conoce por Action/Transaction)

CarWorkShop V1

- División de paquetes
  - UI. Interacción con el usuario
  - Business. Lógica de negocio + persistencia
- En los ui.XX.action se interactúa con el usuario. Puede haber algunas comprobaciones de validez de datos, etc.. Invoca a las clases de *business* necesarias



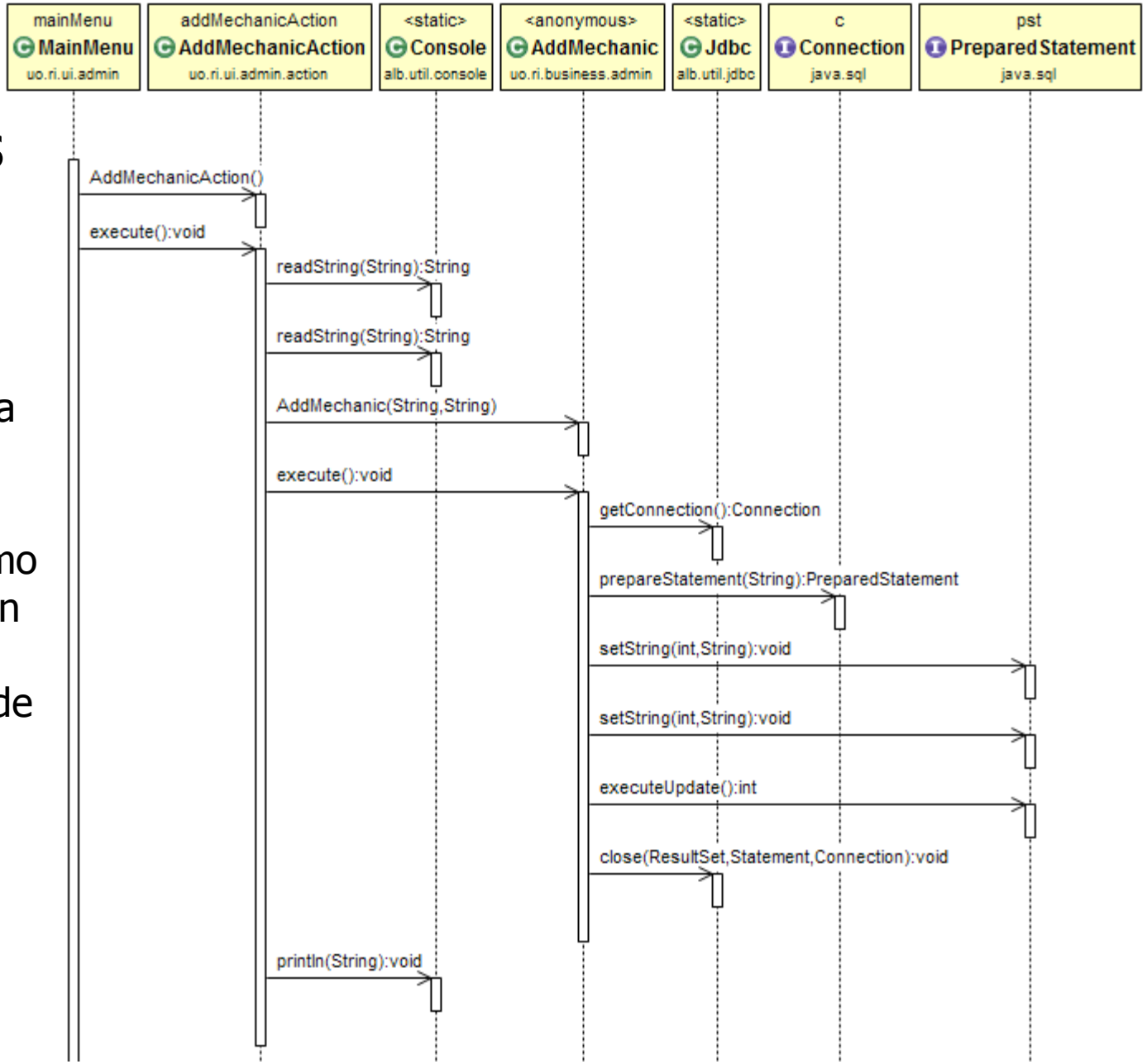
V1 Mecánicos



V1 Mecánicos

Diagrama de secuencia para **añadir un mecánico**.

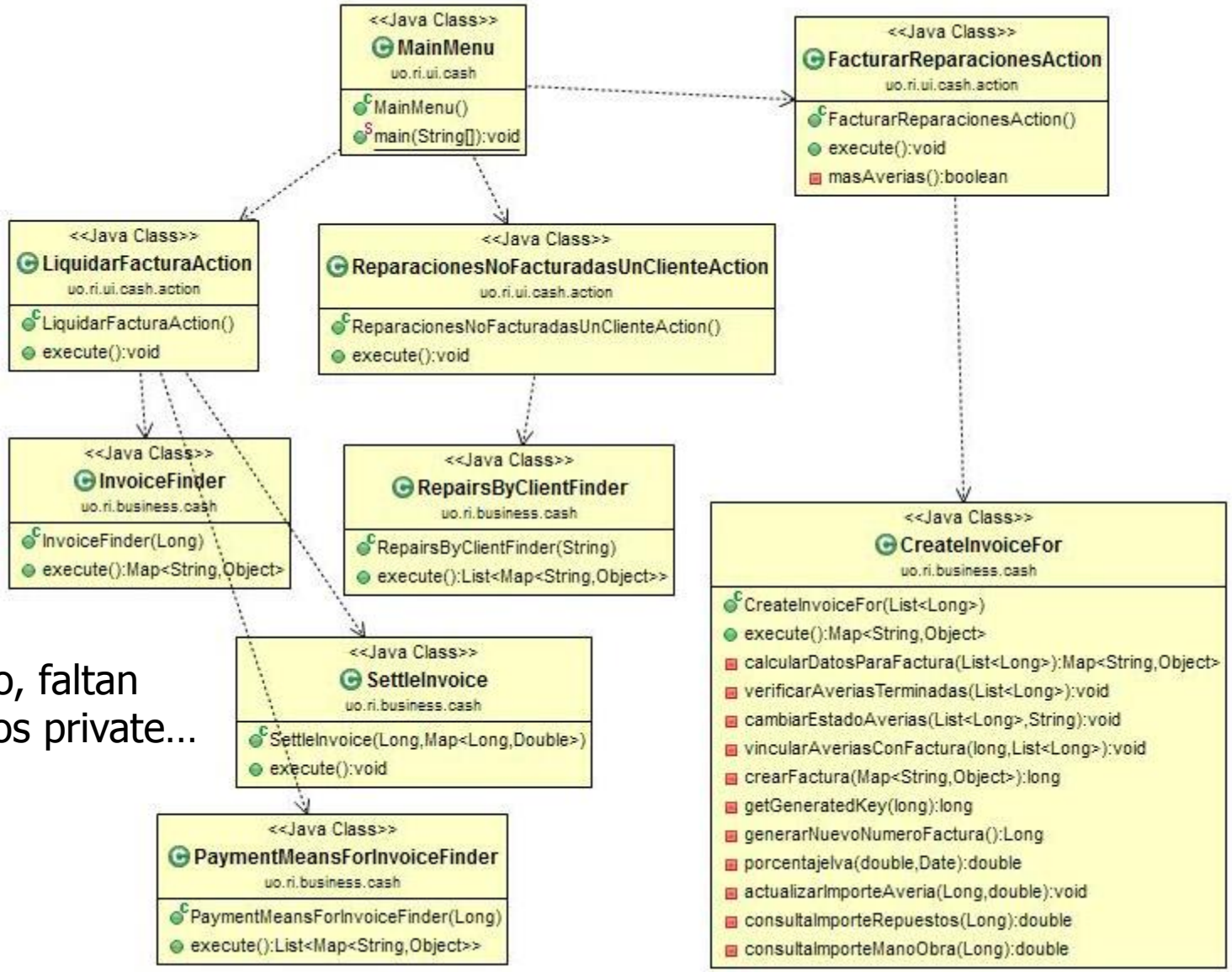
Podemos observar cómo se separa la interacción con el usuario (AddMechanicAction) de la lógica+persistencia (AddMechanic)



CarWorkShop V1

- Haced en grupos:
  - El diagrama de secuencia para crear factura
  - El diagrama de clases de Cash/Caja
- Objetivo: separar en capas
- Tiempo: 10-15 minutos
- Se entrega con identificación del grupo

V1 Cash/Caja

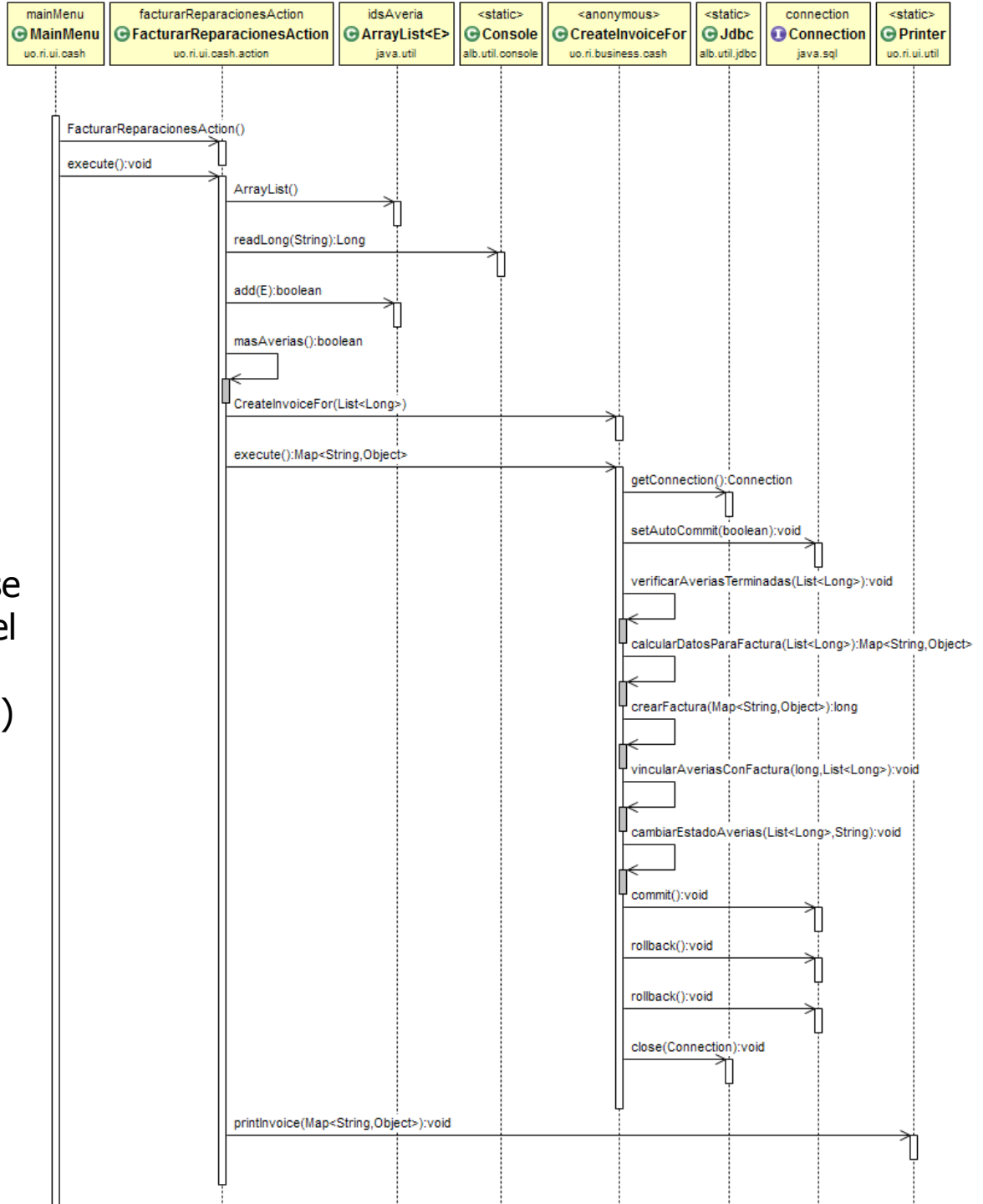


No está completo, faltan atributos, métodos private…

V1 Cash/Caja

Diagrama de secuencia para **crear una factura**.

Podemos observar cómo se separa la interacción con el usuario (FacturarReparacionAction) de la lógica+persistencia (CreateInvoiceFor)



CarWorkShop V2

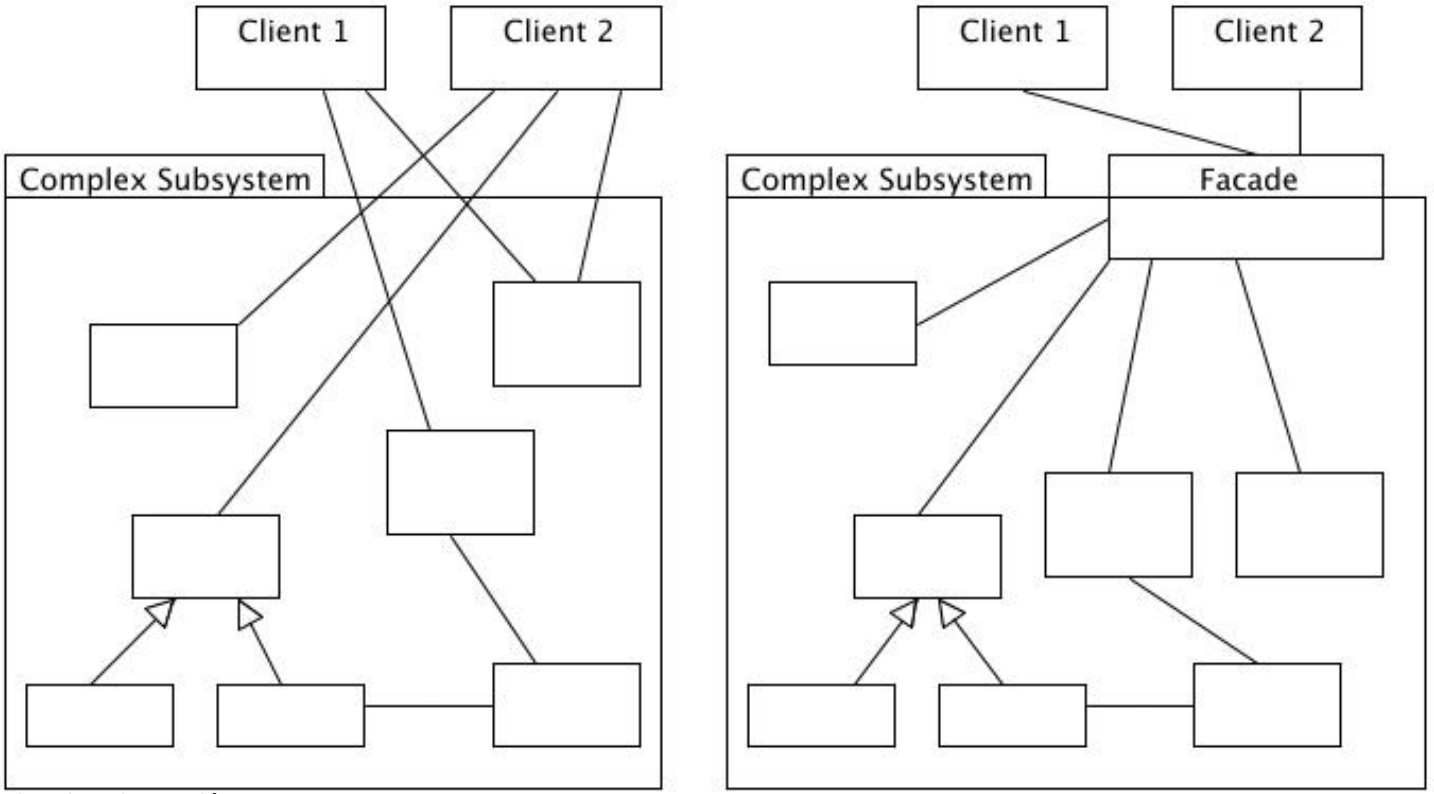
- Vamos a seguir refinando el diseño.
- Podemos ver que desde la capa de presentación se invoca a diversas clases de la capa de negocio (y por lo tanto hay dependencias)
- Un cambio en la capa de negocio (en su implementación) puede implicar cambios en la capa de presentación

CarWorkShop V2

- Vamos a modificar el diseño para desacoplar (en la medida de lo posible) las capas
- P.Ej: subsistema cash→vamos a proporcionar un único punto de entrada al subsistema consiguiendo de esta forma simplificar dependencias, disminuir el acoplamiento, etc.

CarWorkShop V2

- Vamos a implementar el patrón Facade (Gof) o Service Layer (Fowler) (muy similares)

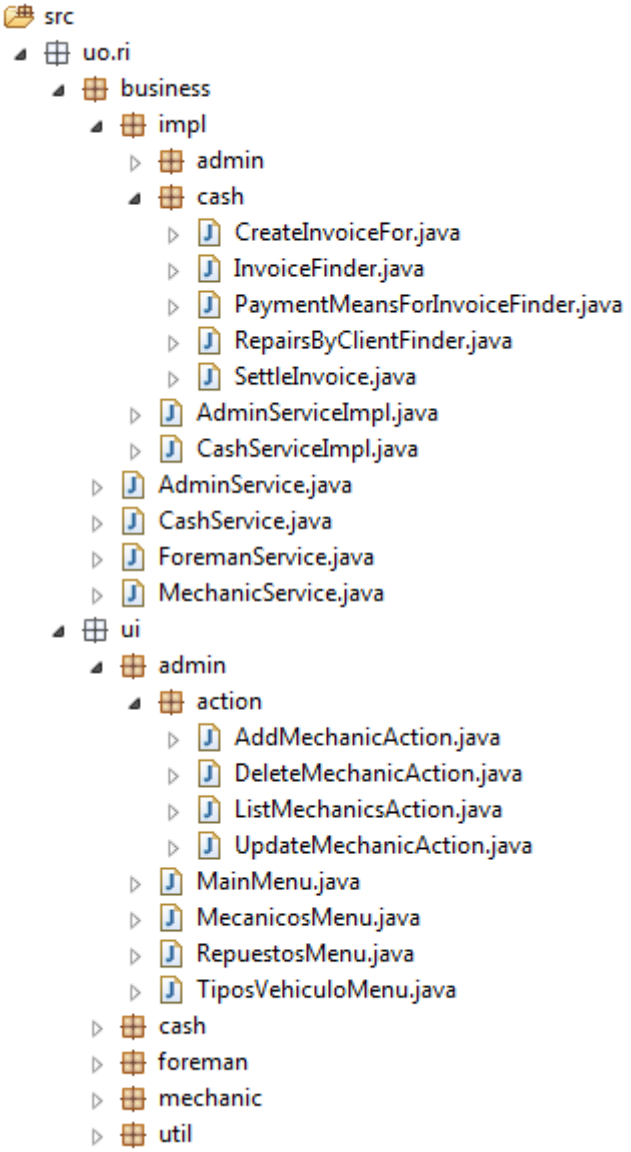


CarWorkShop V2

- El acoplamiento entre cliente y subsistema puede reducirse aún más si se utiliza una clase abstracta (o mediante interface) para la Fachada, y con clases concretas (implementaciones), de tal manera que el cliente no tiene que saber con qué implementación concreta está trabajando
- Esto nos ayuda en aplicaciones distribuidas, desarrollo por varios equipos, a la hora de compilar…

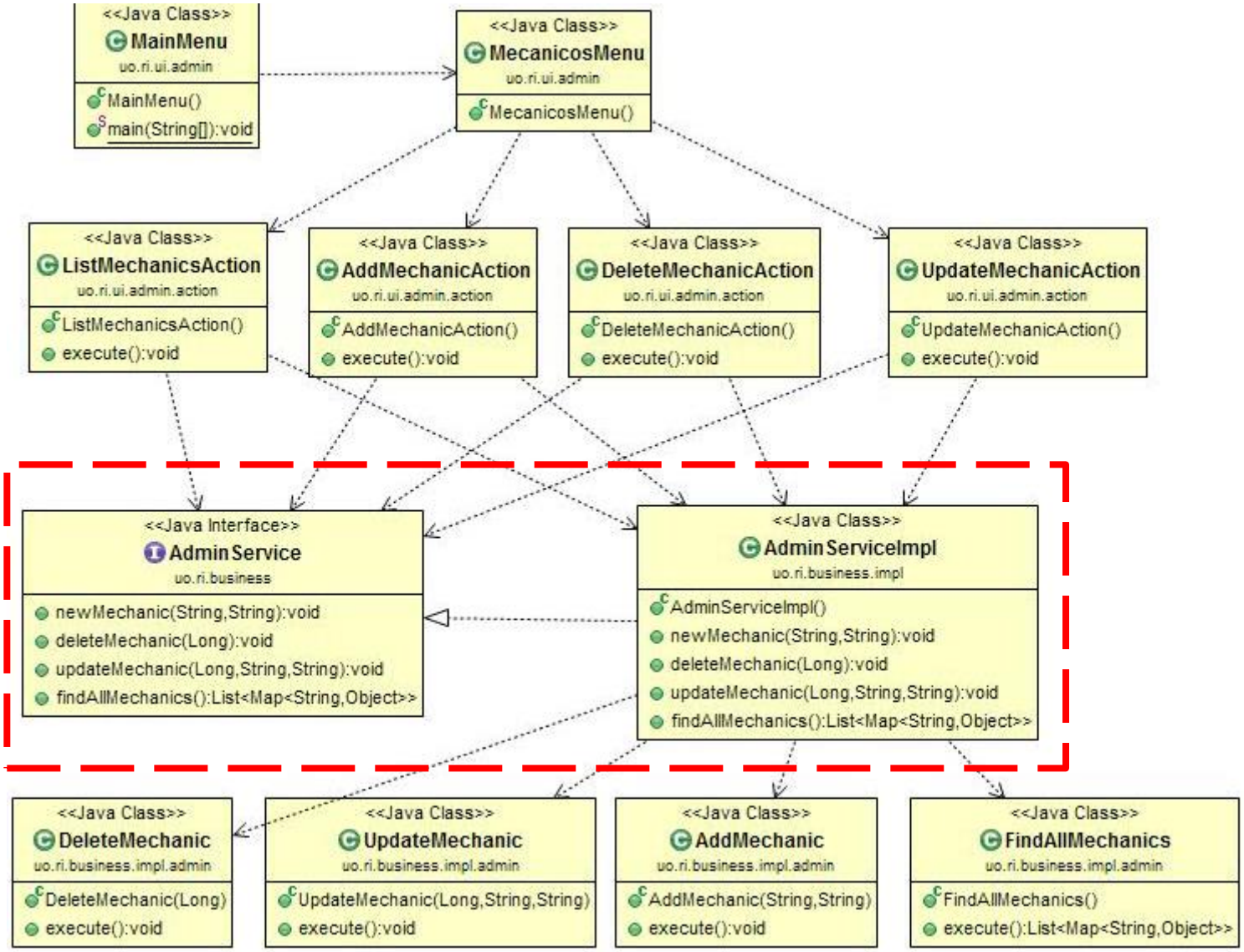
CarWorkShop V2

- Se crea una interfaz por cada actor (XXXService) y una clase que la implementa (XXXServiceImpl)
- La interfaz define los servicios que ofrece la fachada y conforma la ServiceLayer (Fowler)

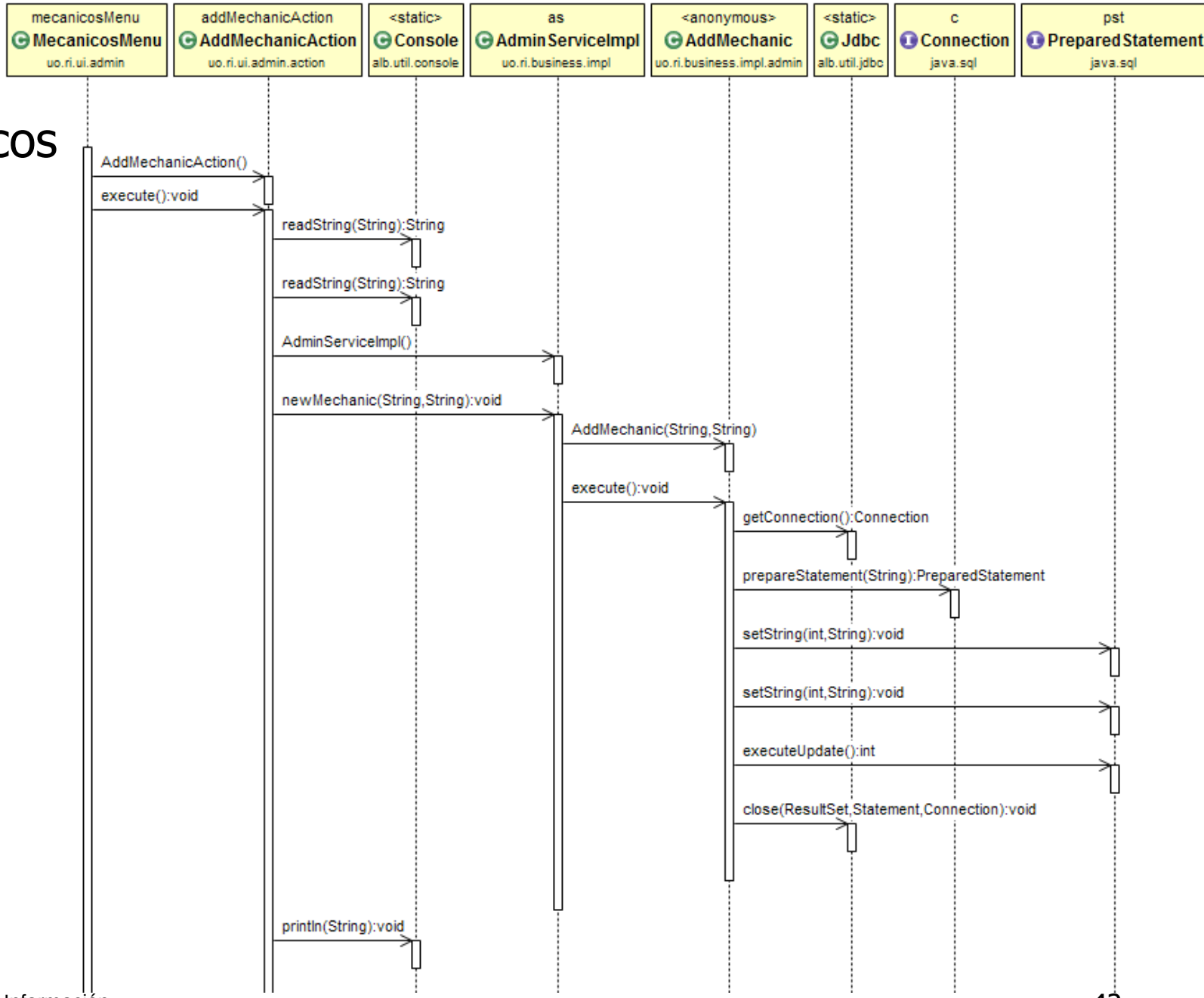


V2 Mecánicos

Fachada (interfaz y su implementación). No se accede desde presentación a la parte interna de las otras capas



V2 Mecánicos

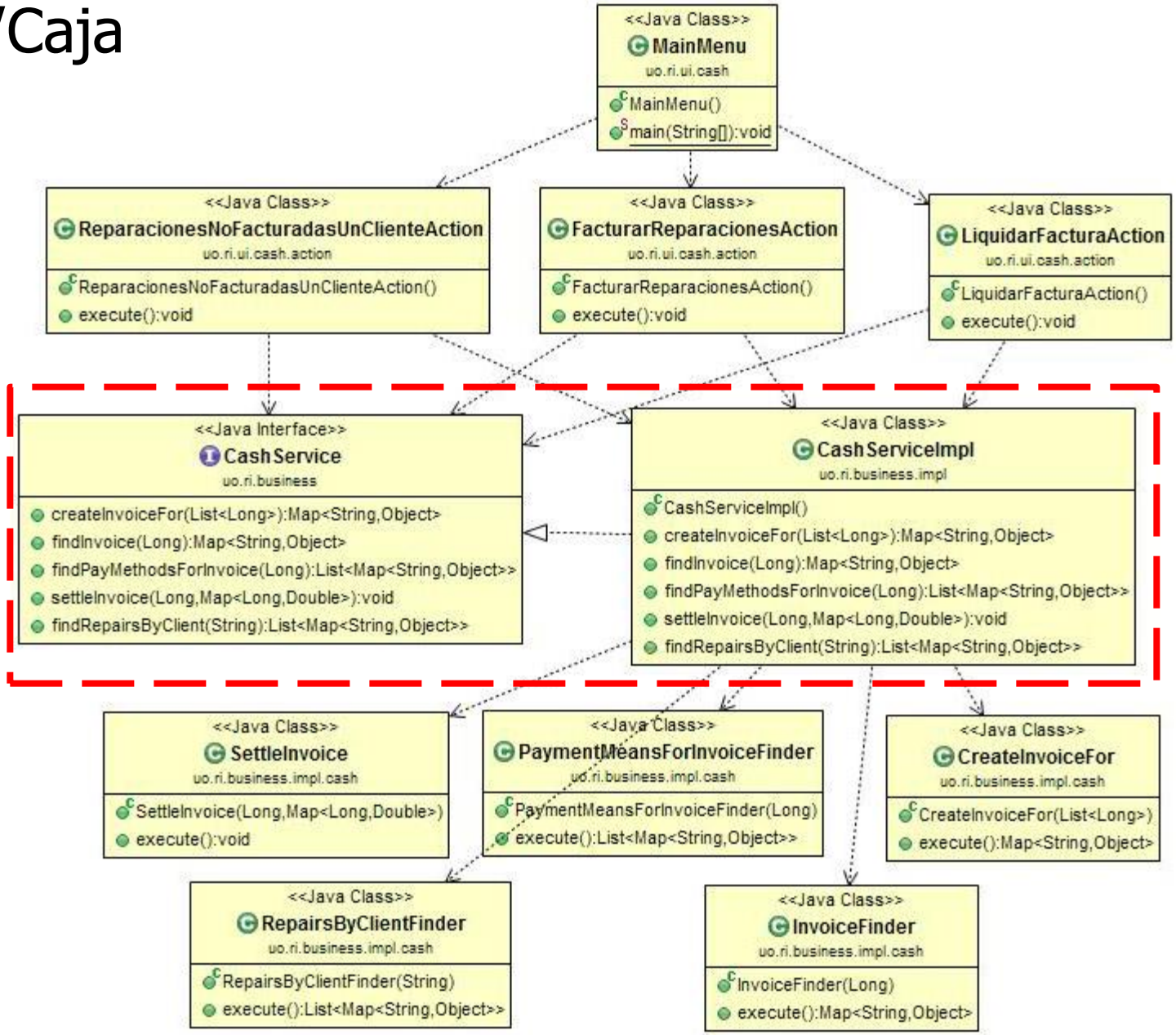


V2 Cash/Caja

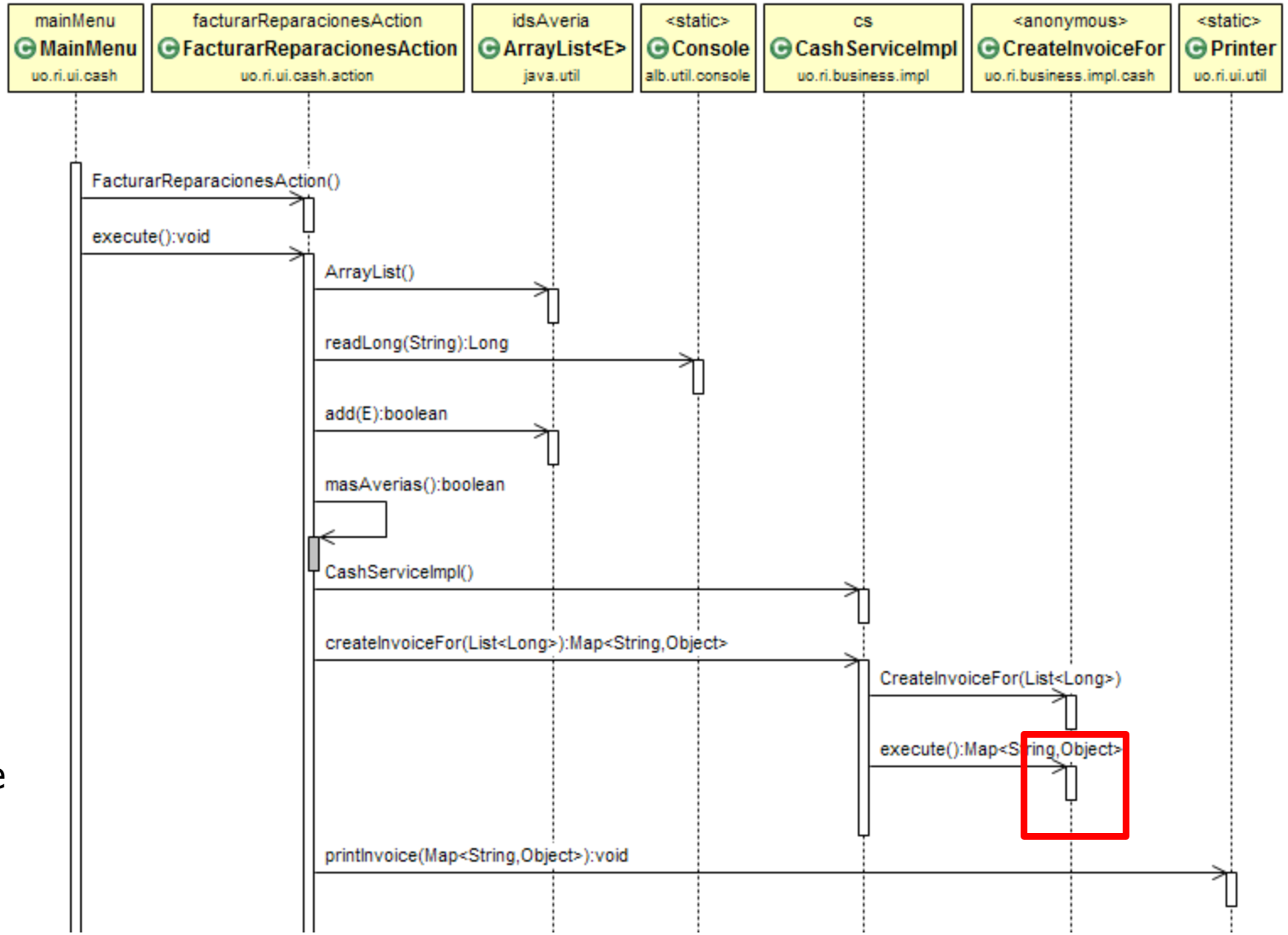
A la hora de compilar sólo necesito conocer la interfaz y (por ahora) la implementación.

Si la aplicación está distribuida sólo necesito conocer la interfaz

Sólo se muestran métodos públicos



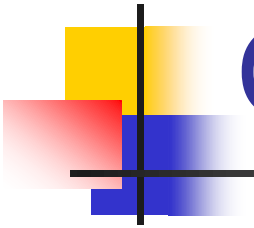
V2 Cash/Caja



Para simplificar no se muestra el desarrollo del método execute de CreateInvoiceFor

CarWorkShop V3

- Aún queda una pequeña dependencia entre la capa de presentación y la de lógica: la capa de presentación necesita conocer cuál es la clase que implementa la Fachada.
- Si se modifica esta clase (se cambia por otra) será necesario recompilar la clase de presentación que la instancia.
- Para solucionarlo vamos a usar una Factoría (Factory Method del Gof)



CarWorkShop V3

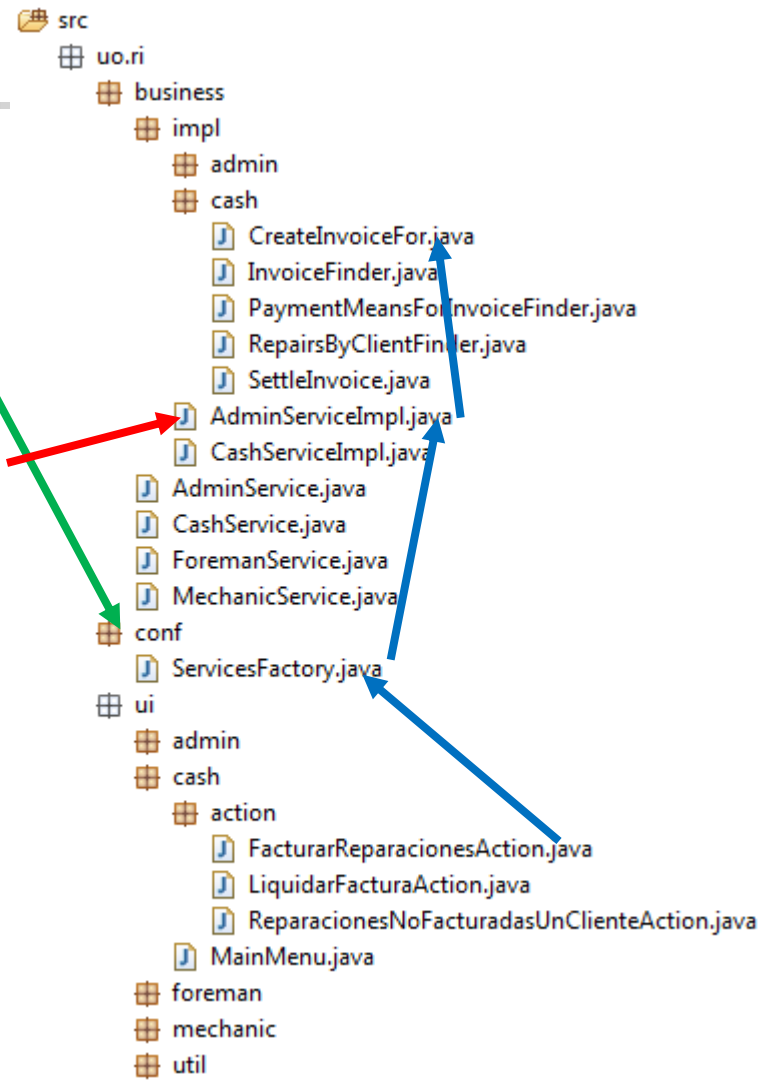
- Cuando necesitemos una instancia de un Servicio se la pediremos a la Factoría (un método estático) que la creará y la devolverá
- De esta forma no necesitamos conocer la implementación del Servicio y por tanto no tenemos dependencia (conocemos su interfaz y la Factoría)
- Un cambio en la implementación del Servicio no afecta a la capa de presentación
- La Factoría puede adaptarse sin cambiar su interfaz (getXXXService)

CarWorkShop V3

Aparece la factoría (ServicesFactory, paquete conf).

Las clases de presentación solicitan a la Factoría un Service (getXXXService) y ésta les devuelve una implementación del Servicio (XXXXServiceImpl).

Las clases de la capa de presentación conocen la Factoría y las interfaces de los servicios (XXXService)→se elimina la dependencia de la implementación del Servicio de la capa inferior.

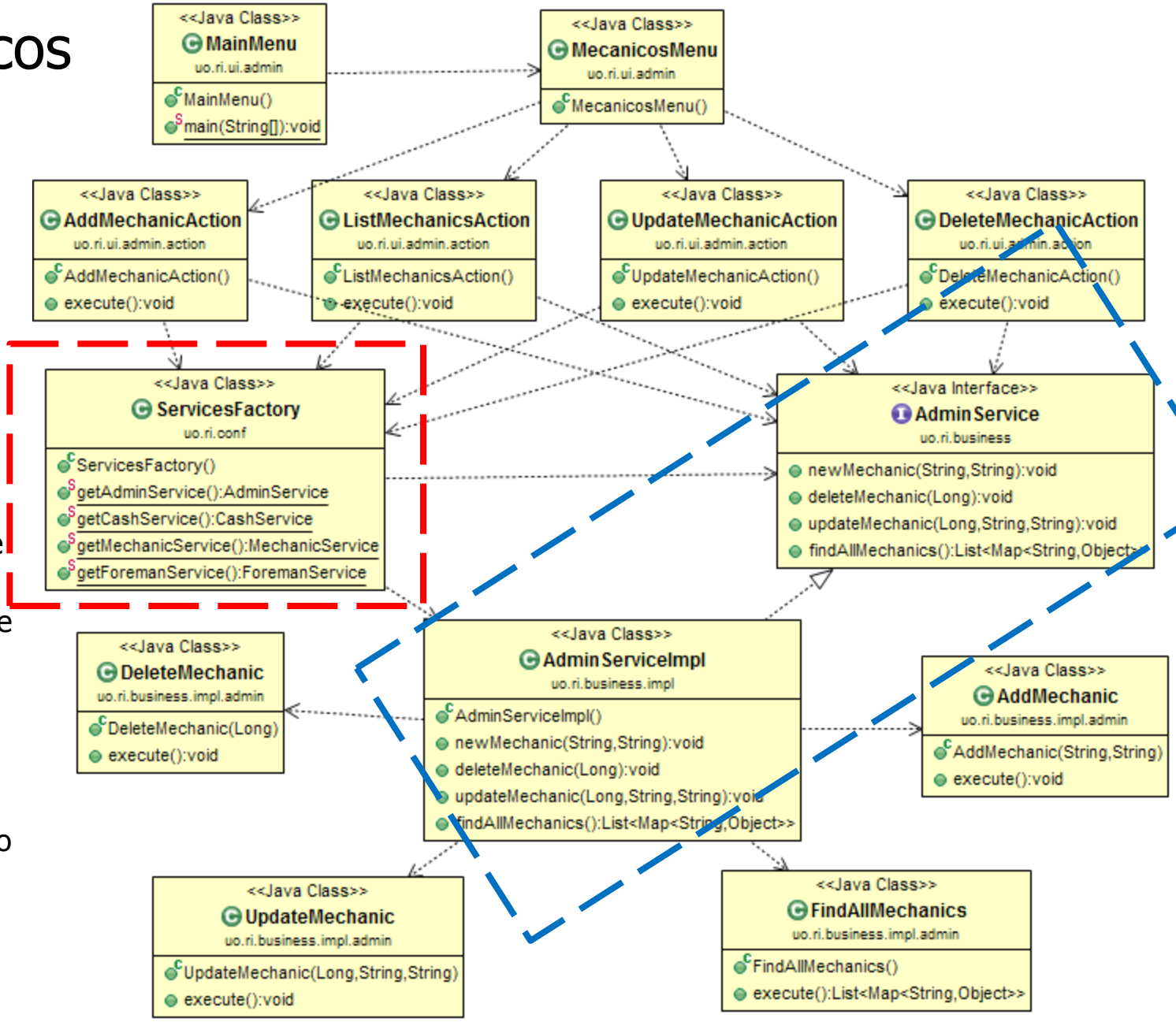


V3 Mecánicos

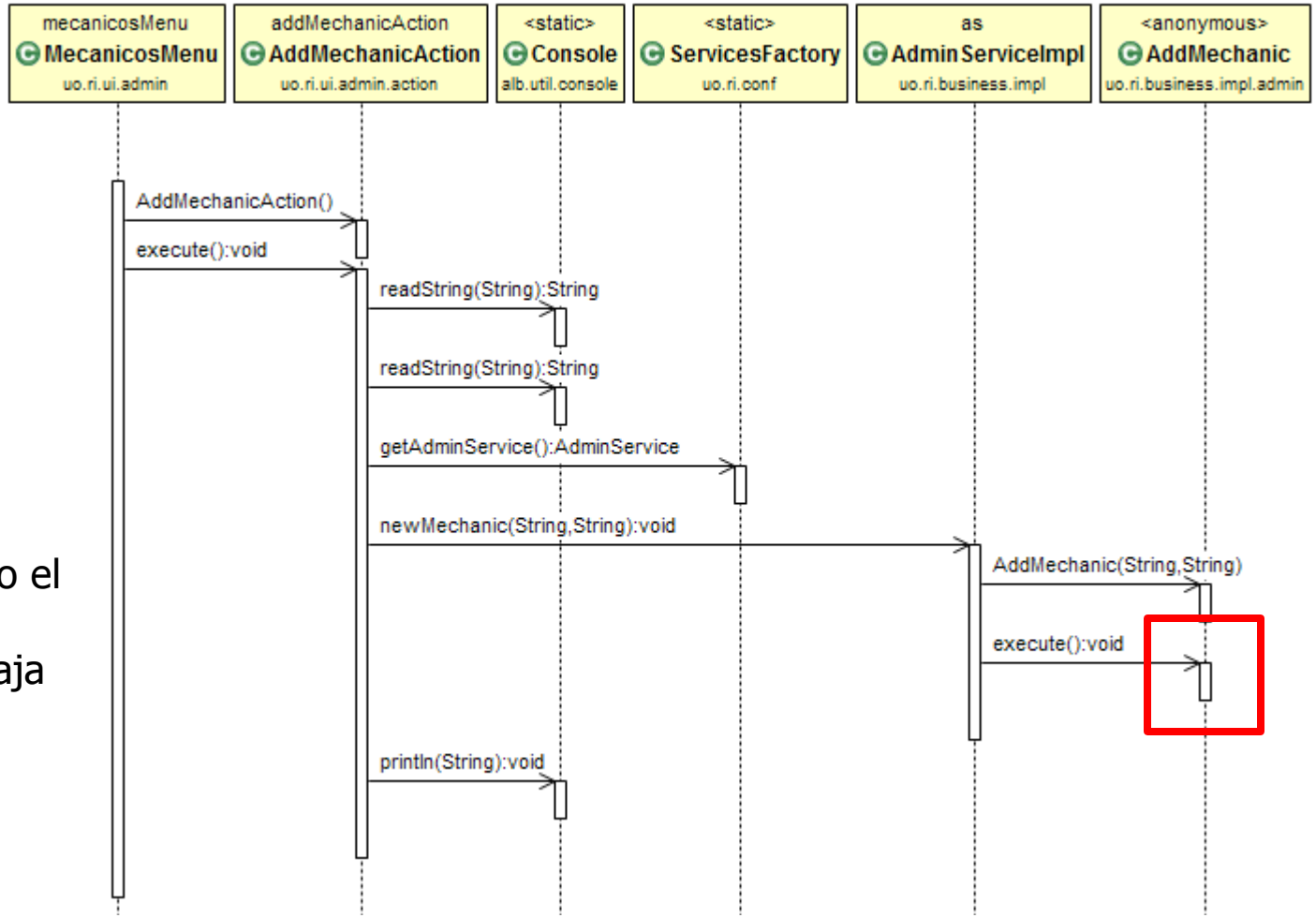
La factoría (en rojo) devuelve la fachada (en azul, su interfaz y la clase que la implementa).

Los métodos getXXService de la Factoría son estáticos

Un cambio en la implementación del servicio implica una modificación en el método getXXXService correspondiente



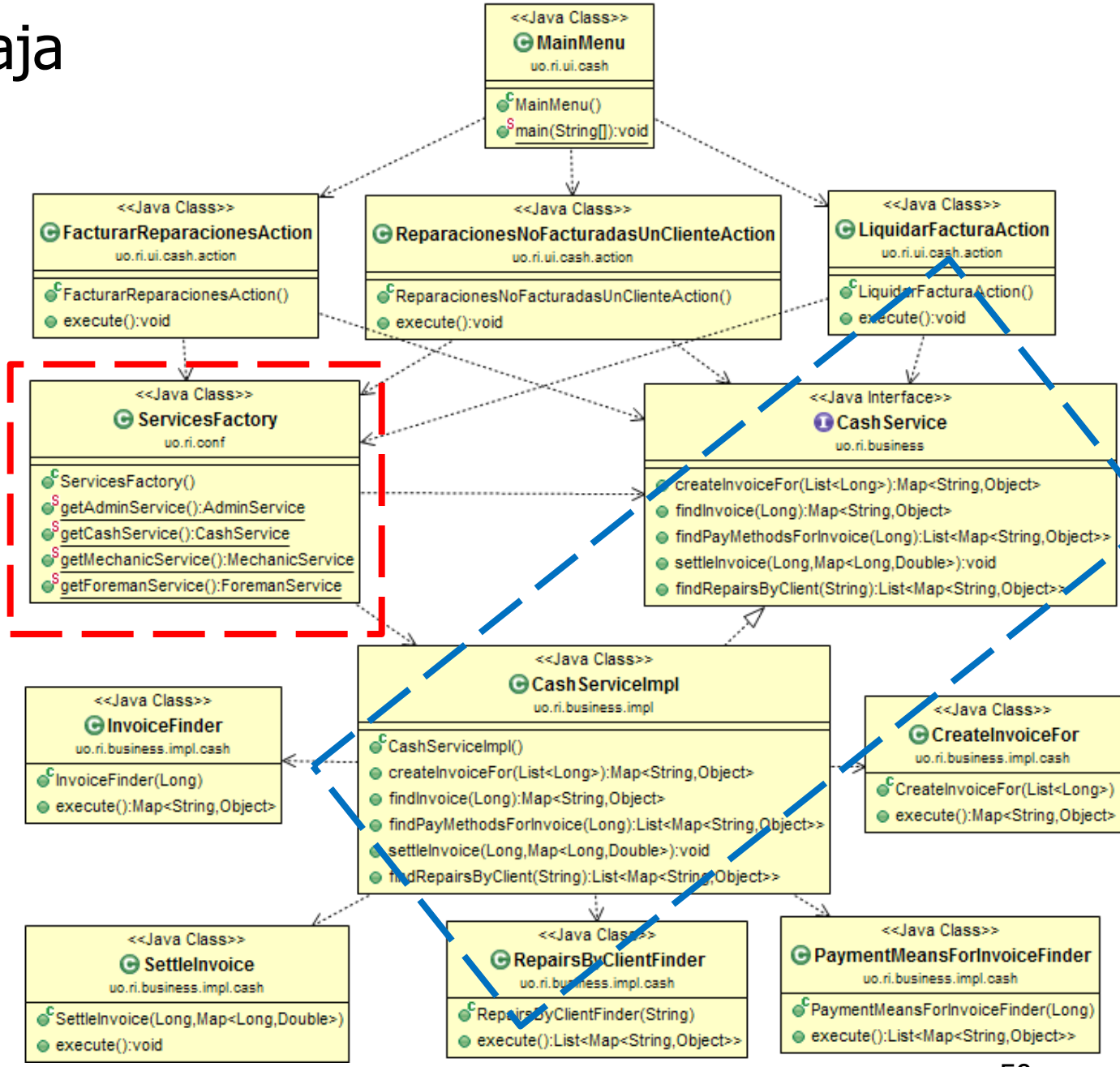
V3 Mecánicos



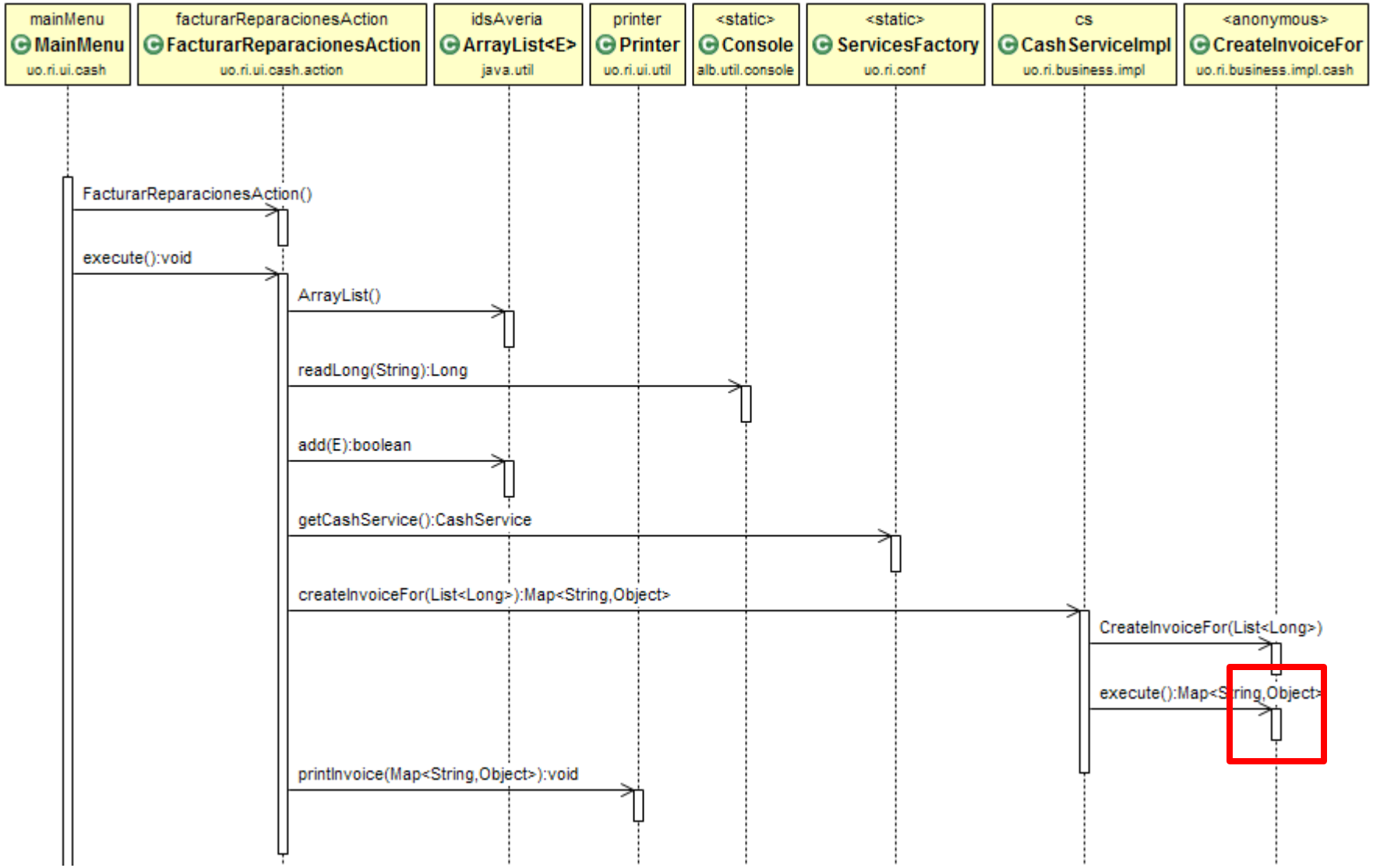
Se le pide el servicio a la Factoría. Una vez recibido el servicio (instancia de su implementación) se trabaja con él.

Para simplificar no se muestra el desarrollo del método execute de AddMechanic

V3 Cash/Caja


<<Java Class>>
MainMenu
uo.ri.ui.cash
MainMenu()
main(String[]):void
<<Java Class>>
FacturarReparacionesAction
uo.ri.ui.cash.action
FacturarReparacionesAction()
execute():void
<<Java Class>>
ReparacionesNoFacturadasUnClienteAction
uo.ri.ui.cash.action
ReparacionesNoFacturadasUnClienteAction()
execute():void
<<Java Class>>
LiquidarFacturaAction
uo.ri.ui.cash.action
LiquidarFacturaAction()
execute():void
<<Java Class>>
ServicesFactory
uo.ri.conf
ServicesFactory()
getAdminService():AdminService
getCashService():CashService
getMechanicService():MechanicService
getForemanService():ForemanService
<<Java Interface>>
CashService
uo.ri.business
createInvoiceFor(List<Long>):Map<String,Object>
findInvoice(Long):Map<String,Object>
findPayMethodsForInvoice(Long):List<Map<String,Object>>
settleInvoice(Long,Map<Long,Double>):void
findRepairsByClient(String):List<Map<String,Object>>
<<Java Class>>
CashServiceImpl
uo.ri.business.impl
CashServiceImpl()
createInvoiceFor(List<Long>):Map<String,Object>
findInvoice(Long):Map<String,Object>
findPayMethodsForInvoice(Long):List<Map<String,Object>>
settleInvoice(Long,Map<Long,Double>):void
findRepairsByClient(String):List<Map<String,Object>>
<<Java Class>>
InvoiceFinder
uo.ri.business.impl.cash
InvoiceFinder(Long)
execute():Map<String,Object>
<<Java Class>>
CreateInvoiceFor
uo.ri.business.impl.cash
CreateInvoiceFor(List<Long>)
execute():Map<String,Object>
<<Java Class>>
SettleInvoice
uo.ri.business.impl.cash
SettleInvoice(Long,Map<Long,Double>)
execute():void
<<Java Class>>
RepairsByClientFinder
uo.ri.business.impl.cash
RepairsByClientFinder(String)
execute():List<Map<String,Object>>
<<Java Class>>
PaymentMeansForInvoiceFinder
uo.ri.business.impl.cash
PaymentMeansForInvoiceFinder(Long)
execute():List<Map<String,Object>>

V3 Cash/Caja


mainMenu
MainMenu
uo.ri.ui.cash
facturarReparacionesAction
FacturarReparacionesAction
uo.ri.ui.cash.action
idsAveria
ArrayList<E>
java.util
printer
Printer
uo.ri.ui.util
<static>
Console
alb.util.console
<static>
ServicesFactory
uo.ri.conf
cs
CashServiceImpl
uo.ri.business.impl
<anonymous>
CreateInvoiceFor
uo.ri.business.impl.cash
FacturarReparacionesAction()
execute():void
ArrayList()
readLong(String):Long
add(E):boolean
getCashService():CashService
createInvoiceFor(List<Long>):Map<String,Object>
CreateInvoiceFor(List<Long>)
execute():Map<String,Object>
printInvoice(Map<String,Object>):void

CarWorkShop V4

- Vamos a aplicar una “buena práctica”: Extraer el código SQL de entre el código Java.
  - Normalmente se saca a un fichero de propiedades, o similar
  - Permite modificar, ajustar, adaptar, optimizar el SQL sin tener que recompilar.
  - Lo puede hacer un DBA modificando el fichero.
  - En el fichero también puede estar la configuración de la conexión: usuario, contraseña, driver, url…

CarWorkShop V4

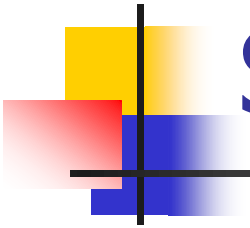
Fichero configuration.properties

```
SQL_FIND_ALL_MECHANICS = select id, nombre,
apellidos from TMecanicos

SQL_INSERT_MECHANIC = insert into
TMecanicos(nombre, apellidos) values (?, ?)

SQL_UPDATE_MECHANIC = update TMecanicos set
nombre = ?, apellidos = ? where id = ?

……
```



Separamos más en capas

- Hasta ahora hemos
  - Separado la capa de presentación de la de lógica+persistencia
  - Dejado un código más independiente, cohesionado, con menos dependencias… en definitiva mejor.
- Ahora tenemos que separar las capas de lógica y persistencia

Separamos más en capas

- Existen múltiples posibilidades en la «literatura», unas más complejas que otras y que se adaptan a distintos escenarios o modelos (y herramientas de las que disponemos).
- Vamos a centrarnos en
  - Patrón DAO (Java BluePrints)
  - Table Data Gateway (de Fowler)
- Son muy similares (más o menos lo mismo con distintos nombres)

Separamos más en capas

- Muchas veces tenemos una clase por cada tabla (de base de datos). La clase actúa como Gateway (puerta/ pasarela/ entrada)/ DAO (Data Access Object) a las tablas.

Patrón DAO

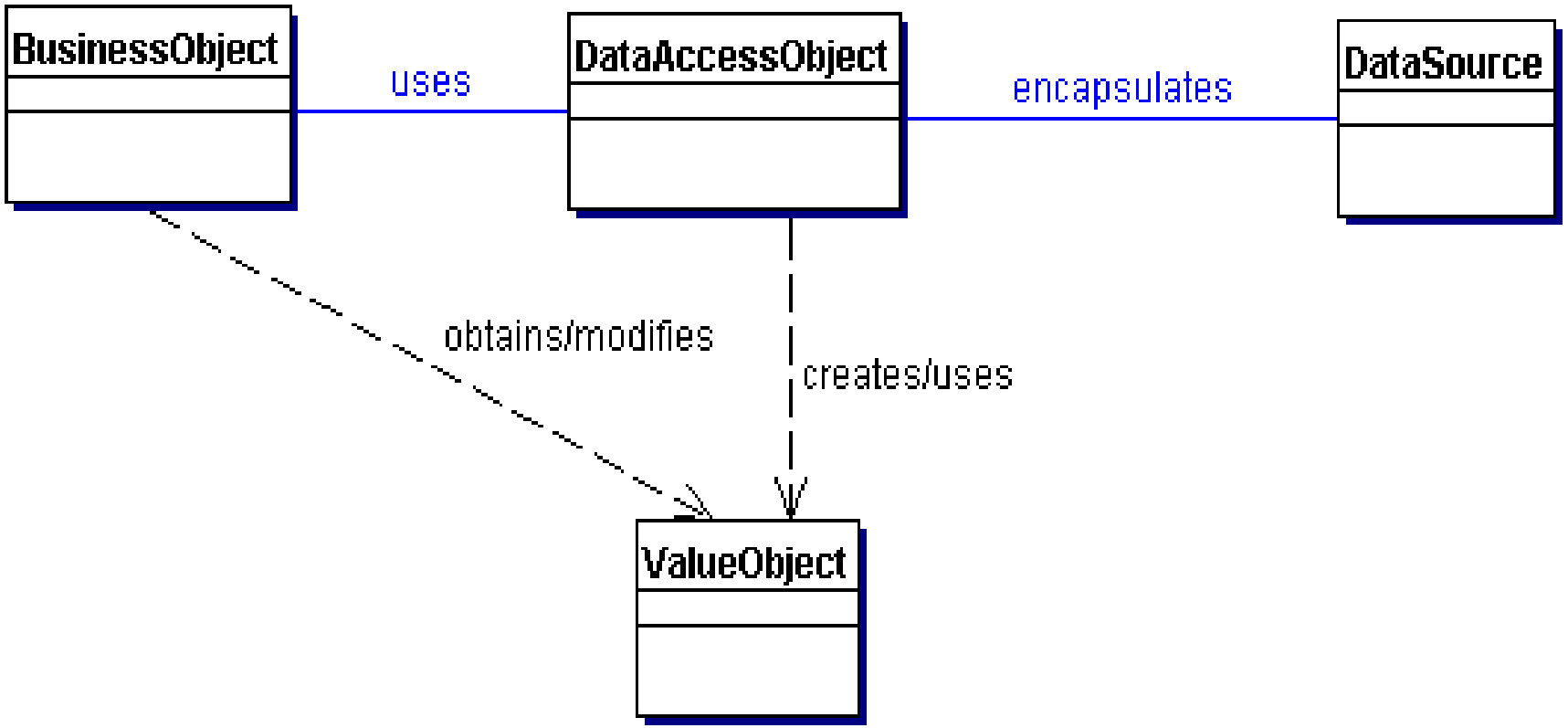
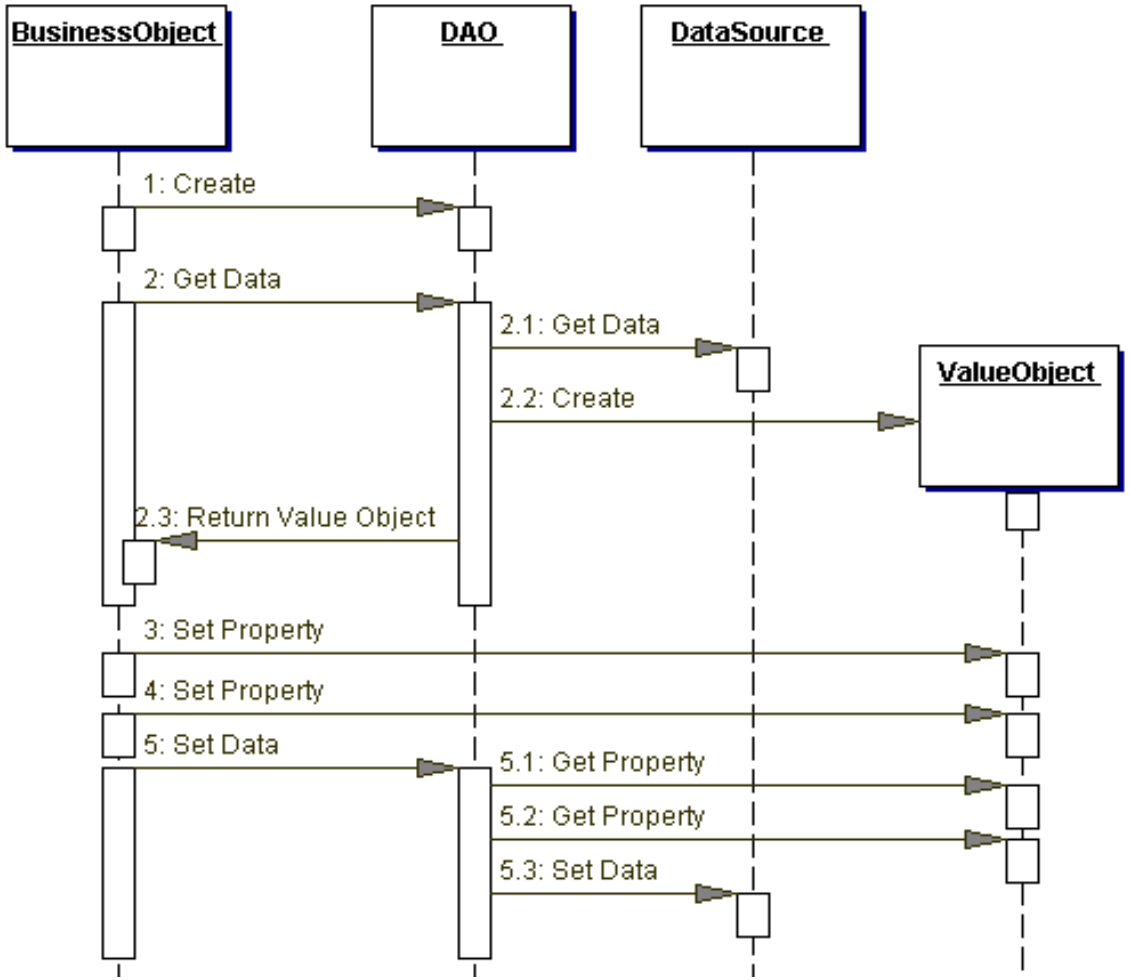
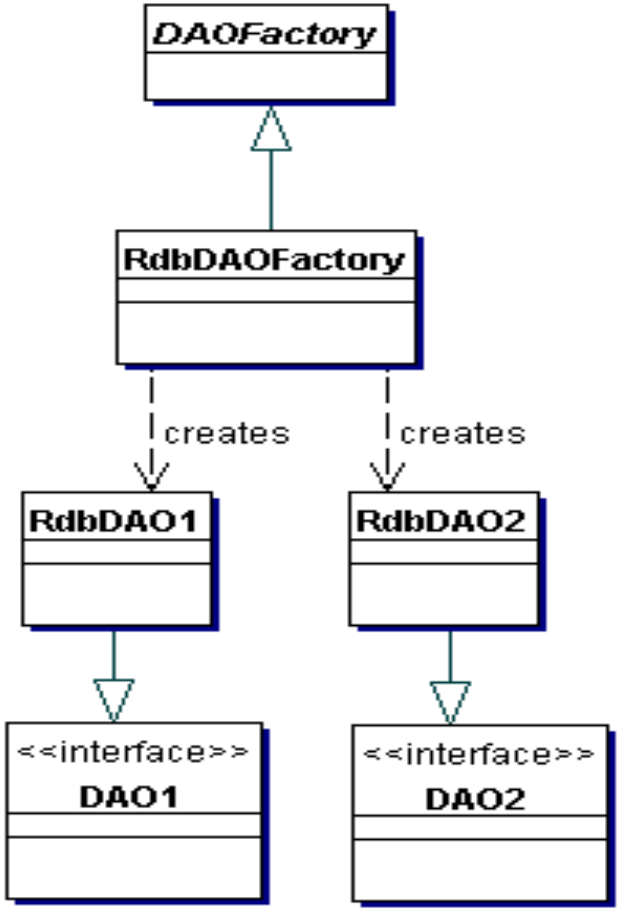

BusinessObject
uses
DataAccessObject
encapsulates
DataSource
obtains/modifies
creates/uses
ValueObject

Diagrama de secuencia DAO



Factory Method - DAO



Abstract Factory - DAO

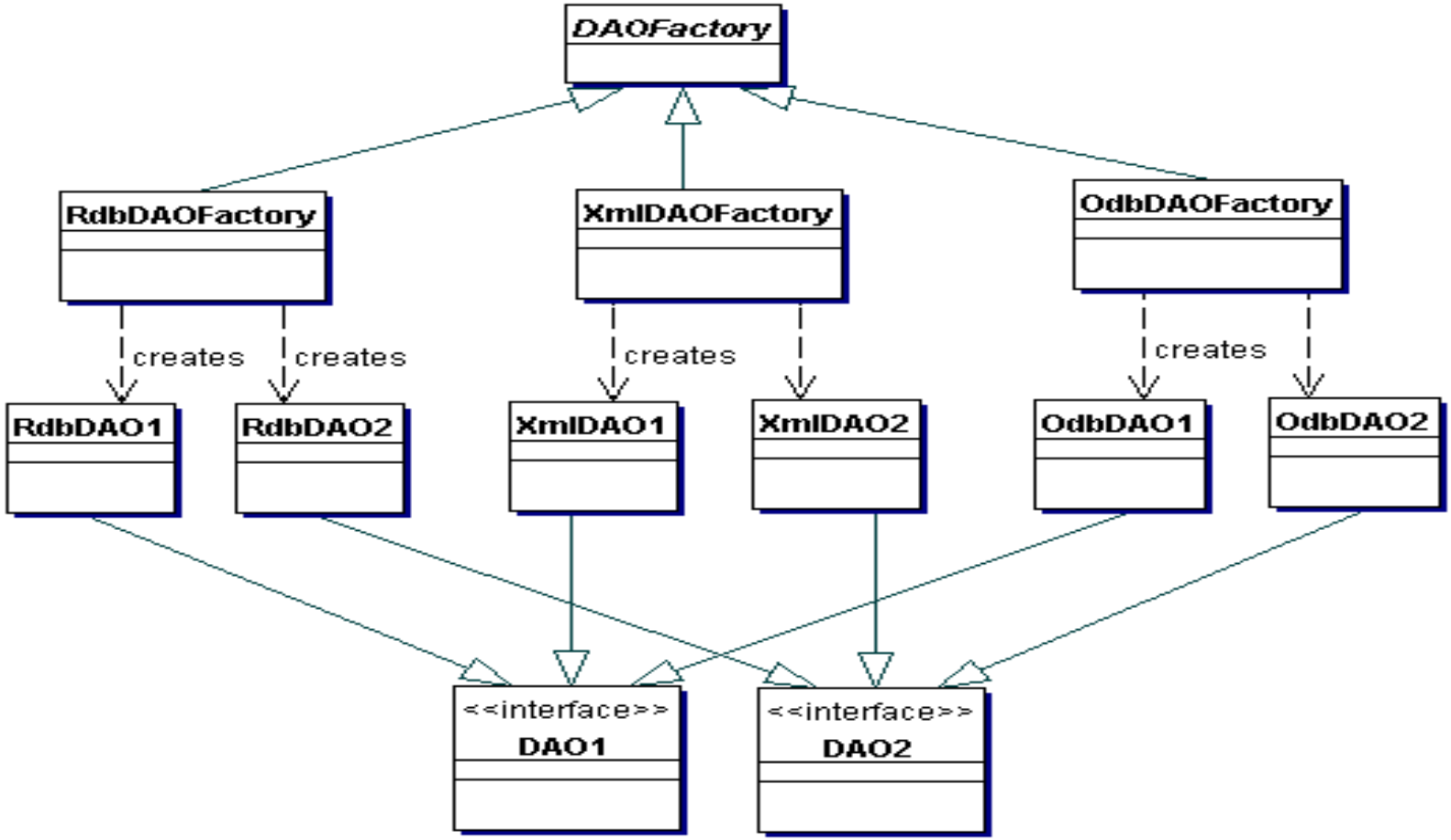
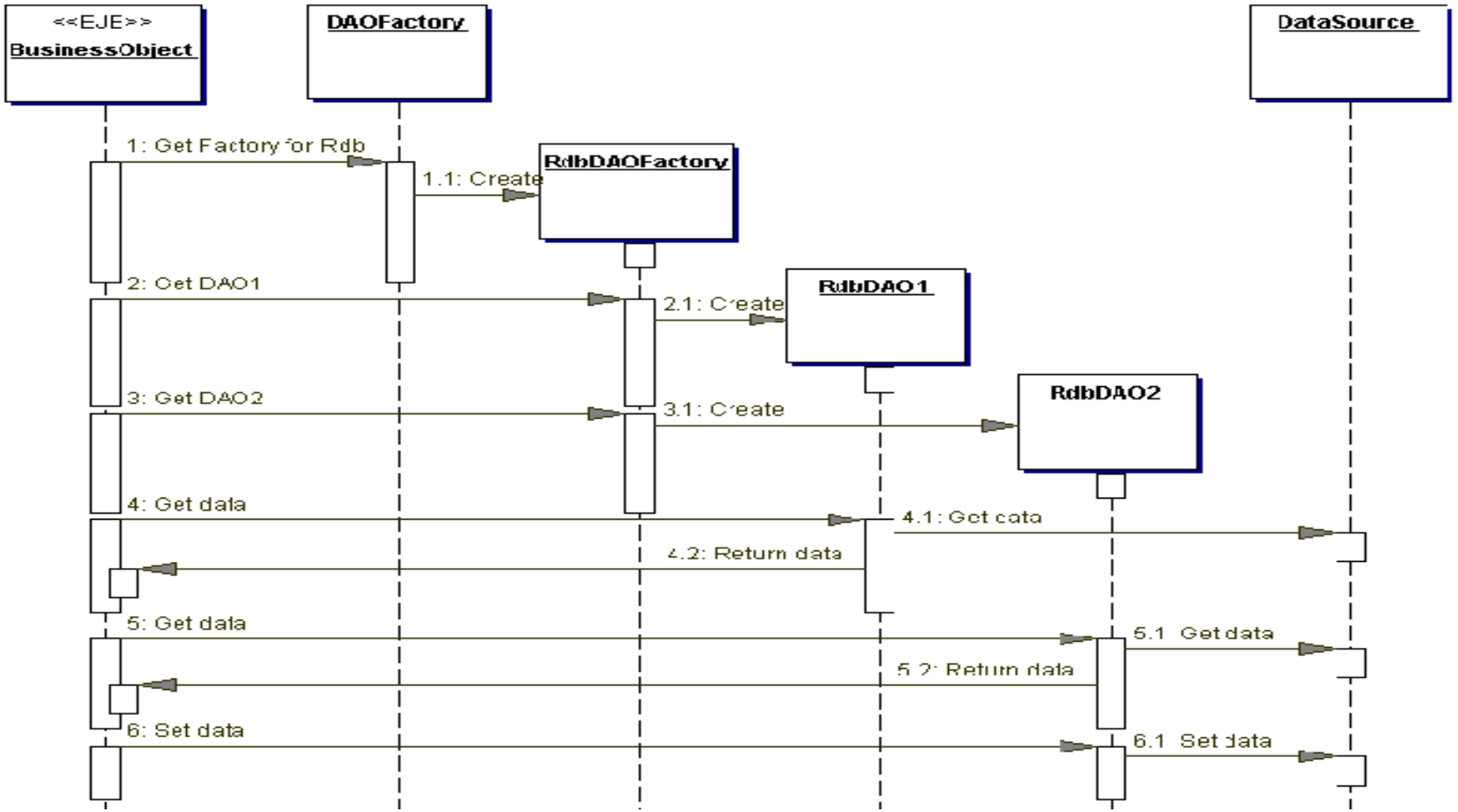


Diagrama de secuencia DAO



Siguiente iteración

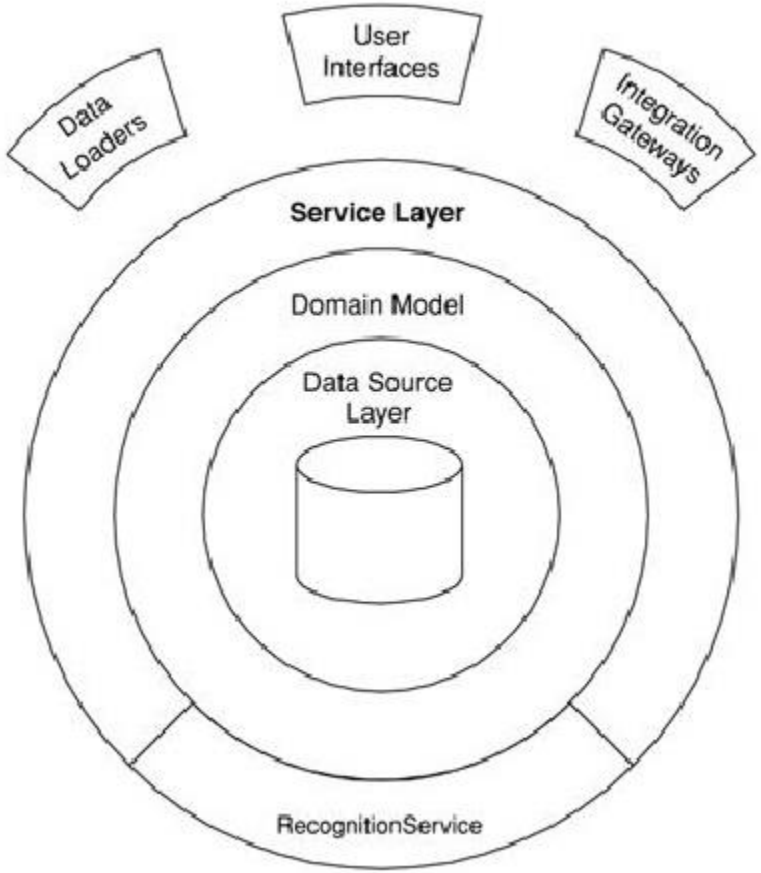
- Vamos a implementar la aplicación haciendo uso de (Fowler)
  - Service Layer
  - Transaction Script
  - Table Data Gateway

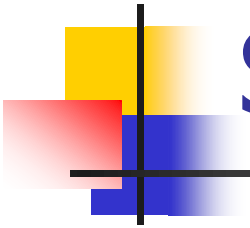
Service Layer

Un *Service Layer* define los límites (fronteras….fachada) de una aplicación con una capa de servicios que establece el conjunto de operaciones disponibles y coordina la respuesta de la aplicación en cada operación

Además de desacoplar puede servir para hacer cosas como

- Control de transacciones
- Seguridad



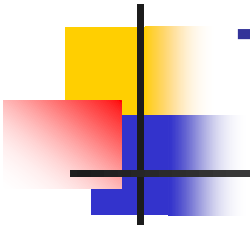


Service Layer

- Las aplicaciones empresariales suelen tener diferentes interfaces para la misma funcionalidad
  - Un cliente pesado (rich client)
  - Un cliente ligero (thin client)
  - Una pasarela para integración con otras aplicaciones
- En este caso se puede dividir la lógica de negocio en dos partes
  - Lógica de dominio: puramente acerca del dominio del problema (ej: estrategia de gestión de cobros)
  - Lógica de la aplicación

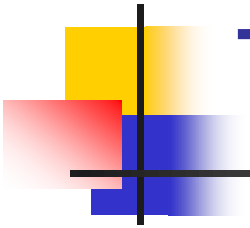
Service Layer

- Una *Service Layer* puede factorizar cierta lógica en su capa haciendo que la capa de la lógica del dominio sea más reusable
- ¿Cuánta lógica se puede poner?
  - En un extremo puede ser una simple Facade
  - En el otro puede ser *Transaction Script* (gestionando toda la lógica) con un *Active Record*
  - Normalmente se encuentra en un punto intermedio



Transaction Script

- Es un patrón (el más simple) que sirve para organizar la lógica de negocio o dominio de la siguiente manera:
  - Utiliza procedimientos de tal forma que cada procedimiento se encarga de gestionar o procesar una petición única desde la capa de presentación

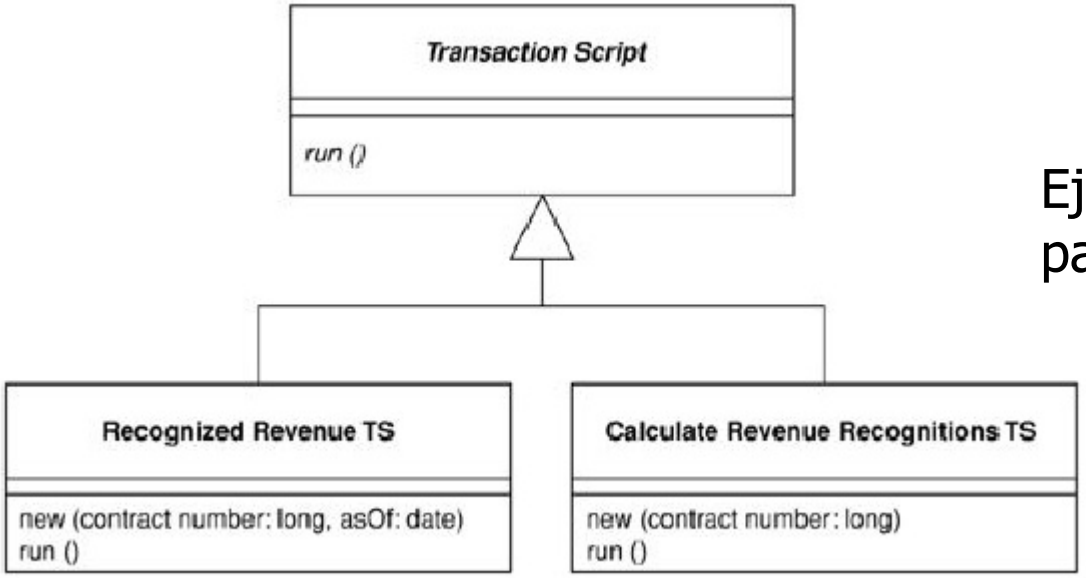


Transaction Script

- Típicamente un *Transaction Script*:
  - Recibe datos de entrada de la capa de presentación
  - Procesa los datos mediante validaciones y cálculos
  - Actualiza la base de datos
  - Invoca operaciones de otros sistemas
  - Responde a la capa de presentación con los datos a mostrar
- Se implementa un procedimiento único para cada caso de uso, operación del sistema o transacción

Transaction Script

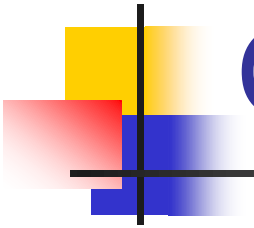
- Los *scripts* deben estar en clases separadas de las capas de presentación y de persistencia (una clase puede tener uno o varios scripts)



Ejemplo de TS con el patrón Command (GoF)

Transaction Script

- Se utiliza en problemas sencillos que no necesitan un modelo de objetos complicado
- Es simple, con un modelo procedural fácil de entender
- Funciona muy bien con modelos simples que usan Table Data Gateway (o Row Data Gateway)
- Los límites de la transacción están muy claros (se comienza la transacción al comienzo del procedimiento y se termina al finalizar)
- Como desventaja cabe citar la duplicación de código entre transacciones. Como el objetivo es gestionar una transacción el código común (abrir y cerrar conexión, control de excepciones, etc.) tiende a duplicarse. Si el dominio del negocio es complejo habrá que utilizar otro modelo (*Domain Model)*

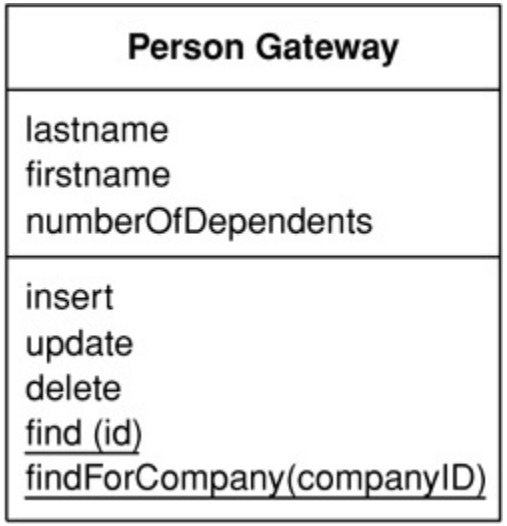


Gateway

- Es una clase que se encarga de gestionar por completo el acceso a los datos (persistencia) de una entidad. Hace de pasarela.
- Muy similar al DAO (iguales)
- Puede ser
  - Row Data Gateway
  - Table Data Gateway

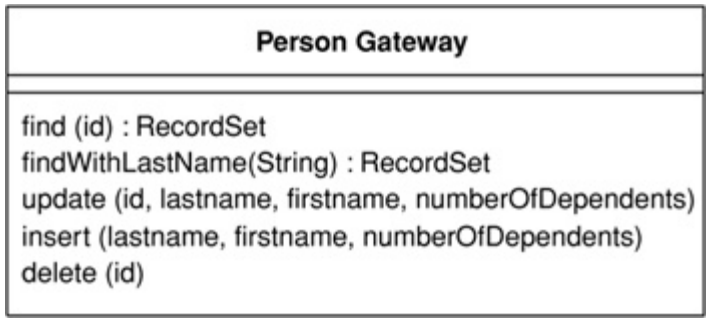
Gateway

- **Row Data Gateway (RDG)** → Devuelve una instancia por cada fila que nos devuelve una consulta



Gateway

- **Table Data Gateway (TDG)** → devuelve un conjunto de resultados (record set) con todas las filas seleccionadas (p.ej un ResultSet de jdbc). (Similar al DAO)

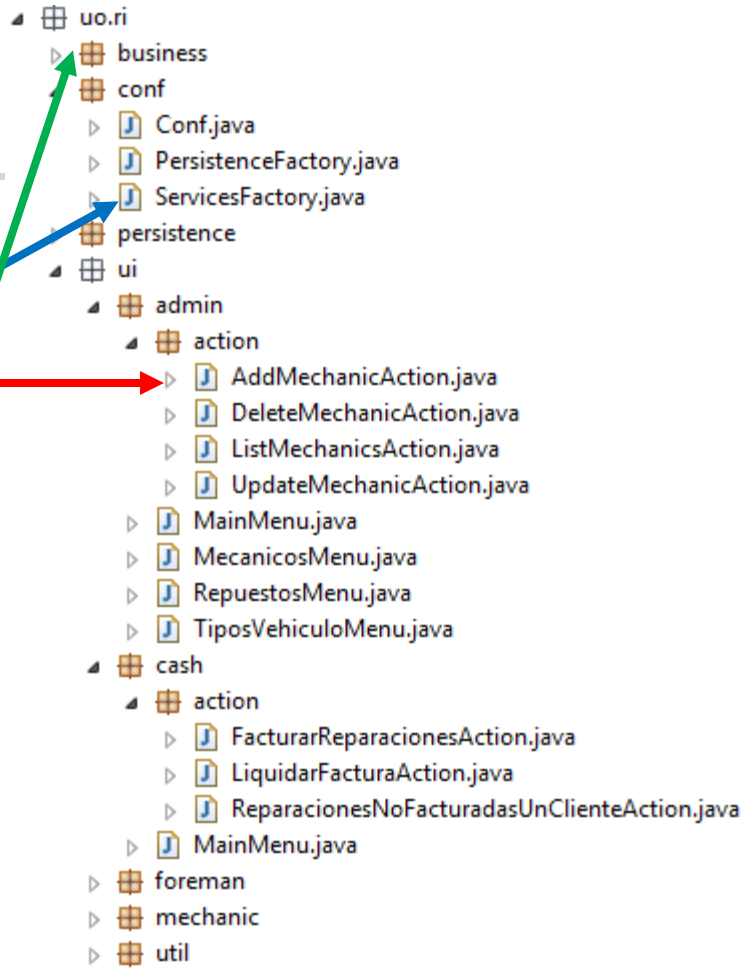


SL.TS.TDG_0

- Vamos a refinar la aplicación siguiendo lo comentado y tendrá las siguientes partes:
  - Capa de presentación→ interacción con el usuario
  - Service Layer (SL)→punto de entrada desde la presentación a la capa de negocio
  - Transaction Script (TS)→ capa de negocio. Se crea una clase por cada caso de uso
  - Table Data Gateway (TDG)→ se encarga de la persistencia. Hay una clase por cada entidad/tabla

SL.TS.TDG_0

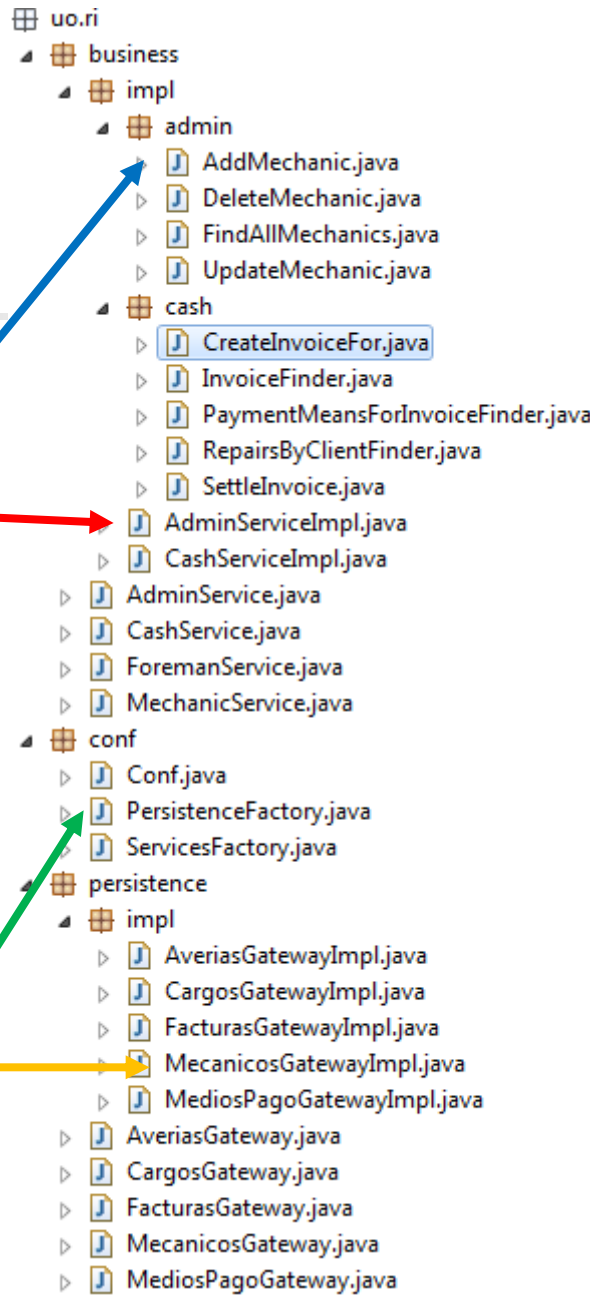
La capa de presentación (paquetes ui.XX) hace uso de una factoría de servicios (paquete conf) para acceder a la implementación del servicio (Service Layer, paquete business) que redirigirá la petición a la clase apropiada (Transaction Script, paquete Business)



SL.TS.TDG_0

Una vez que se ha recibido una petición en la Service Layer (XXXServiceImpl) se redirige la petición a la clase adecuada del Transaction Script (business.impl.XXXX.YYYY).

En esa clase se realizan las tareas y se invoca a las clases encargadas de la persistencia(XXXXGatewayImpl) a través de una factoría (paquete Conf)



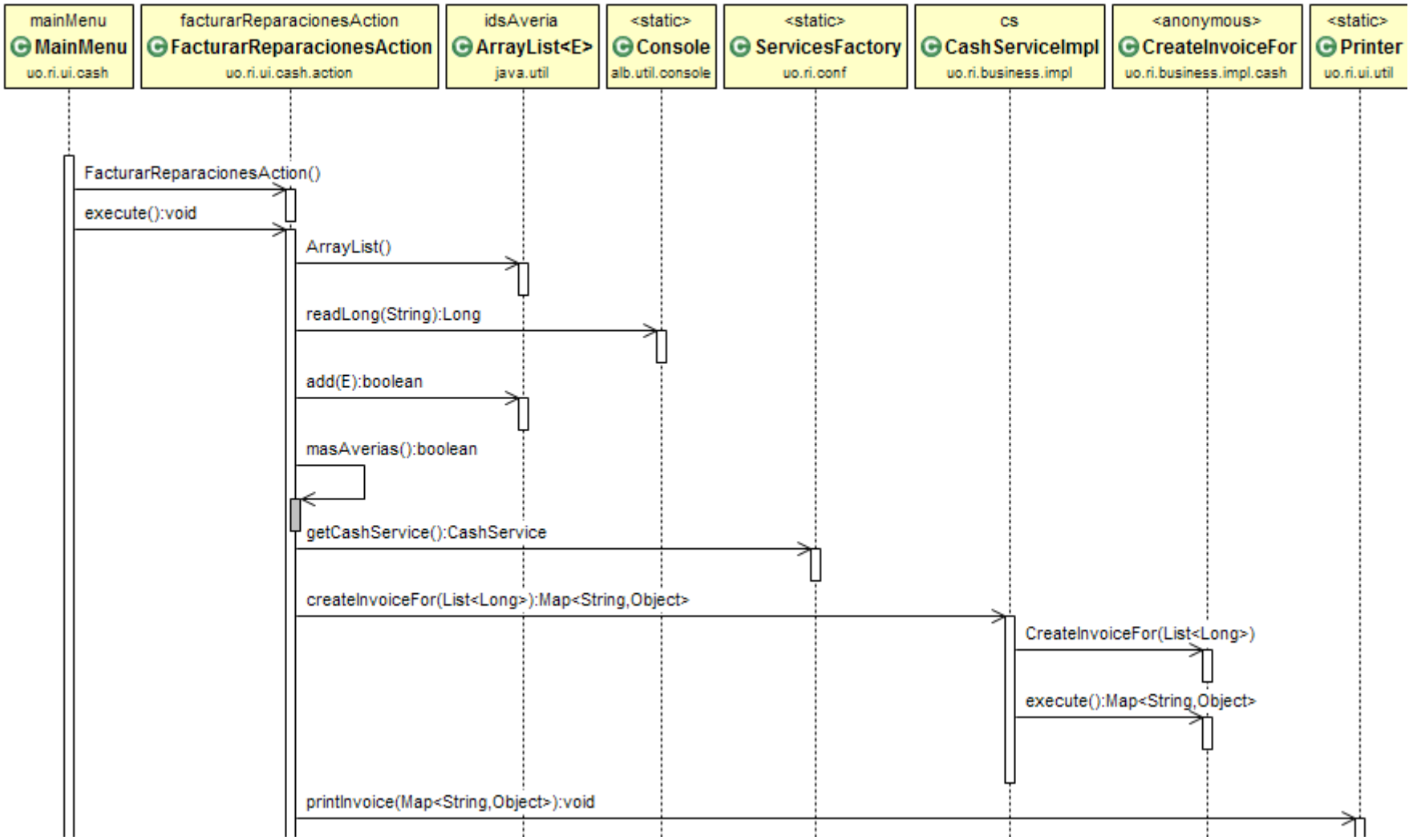
SL.TS.TDG_0

- Diagrama de Clases Mecánicos
- Diagrama de Secuencia Añadir Mecánico
- Observad las distintas capas
  - Presentación
  - Service Layer
  - Transaction Script
  - Table Data Gateway
- Las factorías permiten desacoplar unas capas de las inferiores (a las que utiliza)

SL.TS.TDG_0

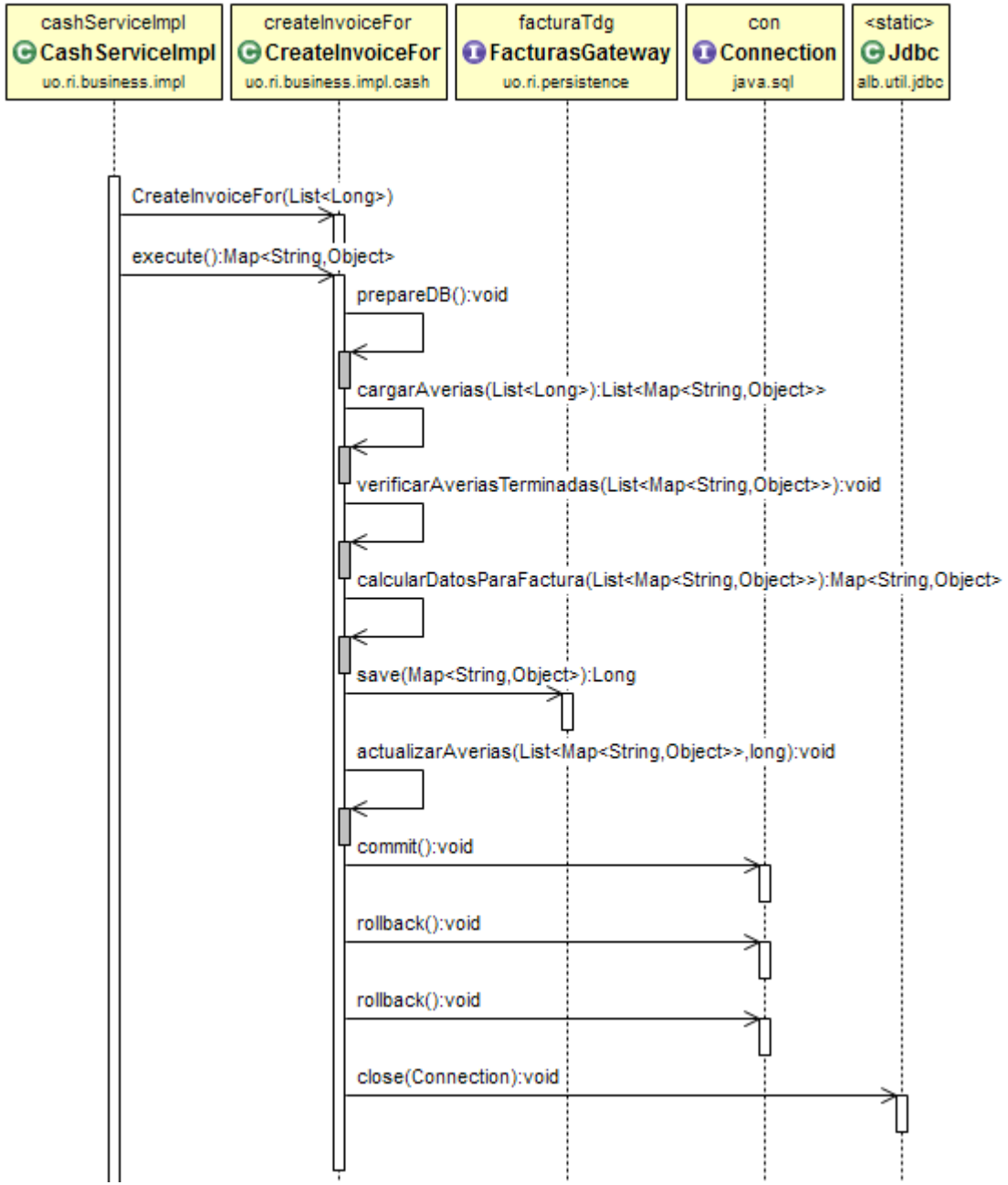
- Diagrama de Clases Facturación
- Diagrama de Secuencia Crear Factura (desglosado en 3 pasos)
  - Paso 1. Presentación + SL + TS
  - Paso 2. SL+TS+TDG
  - Paso 3. TS+TDG. Explicación acceso a persistencia

Crear Factura Paso 1. Presentación + SL + TS



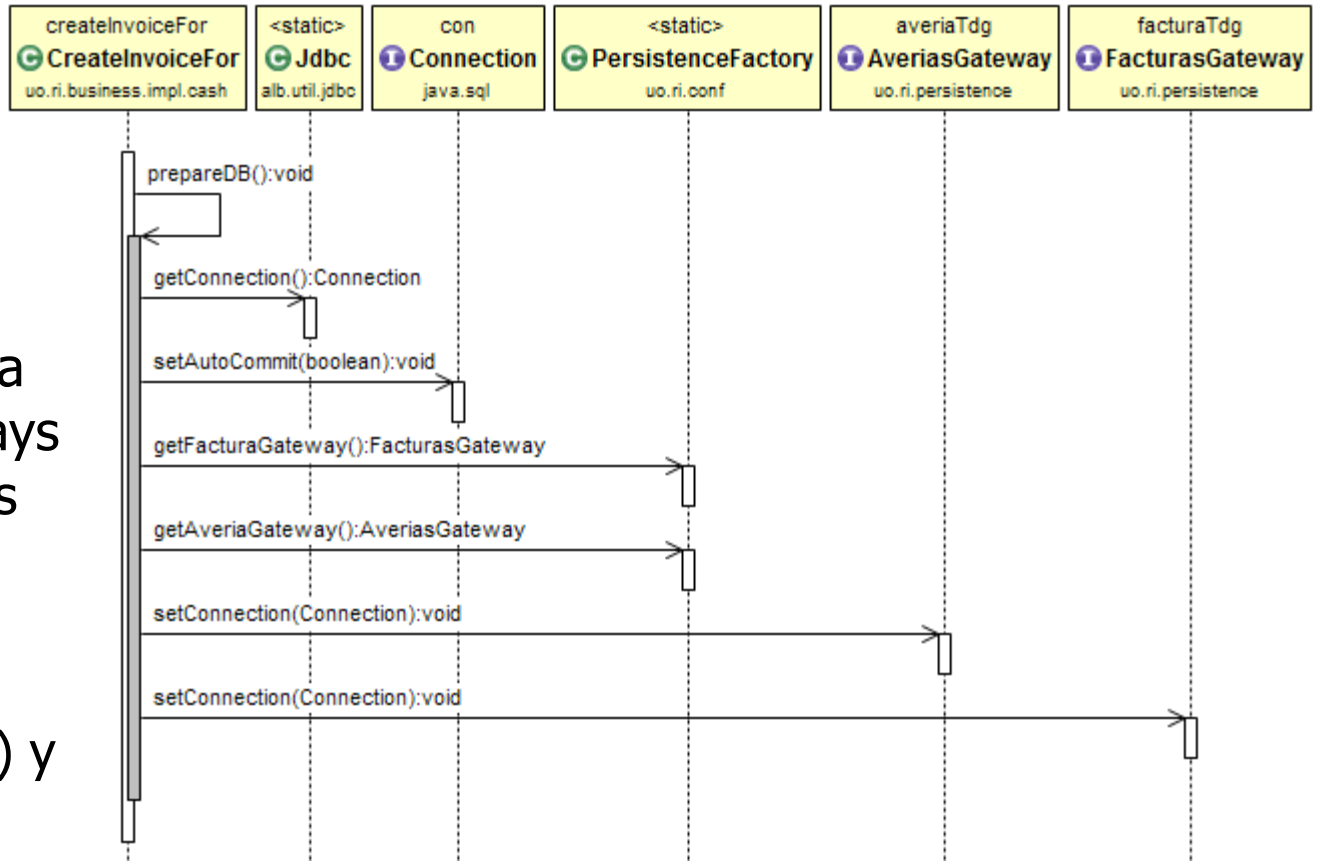
Crear Factura Paso 2. SL+TS+TDG

Los métodos (prepareDB, actualizarAverias, etc.) no están desglosados.
Utilizan Gateways para acceder a los datos.
En el execute es donde se controla la transacción.



Crear Factura Paso 3. TS+TDG

Desglose del método prepareDB (a modo de ejemplo).
Utiliza (pide a la Factoría de persistencia) Gateways para acceder a los datos de facturas y averías.
Fija el modo de autocommit (para controlar la transacción) y asigna la conexión para las Gateways



SL.TS.TDG_0

- Hemos separado completamente por capas
- Usamos Factorías para desacoplar las implementaciones de:
  - La capa Service Layer de la de presentación
  - La capa de persistencia de la de lógica de negocio