Patrones y estándares de acceso a datos

Universidad de Oviedo

Repositorios de Información

Escuela de Ingeniería Informática

2022-2023

Car Workshop - El proyecto inicial CWS0

Atributos de calidad para el software

Patrones

Patrón arquitectónico: Separación en capas

Patrón Fachada (Façade)

Factoría Simple (o Class Factory)

Section 1

Car Workshop - El proyecto inicial CWS0

Estructura de paquetes de CWS0

Por actores: Cada paquete contiene menús de texto y subpaquete (action) que implementa las opciones de menú.

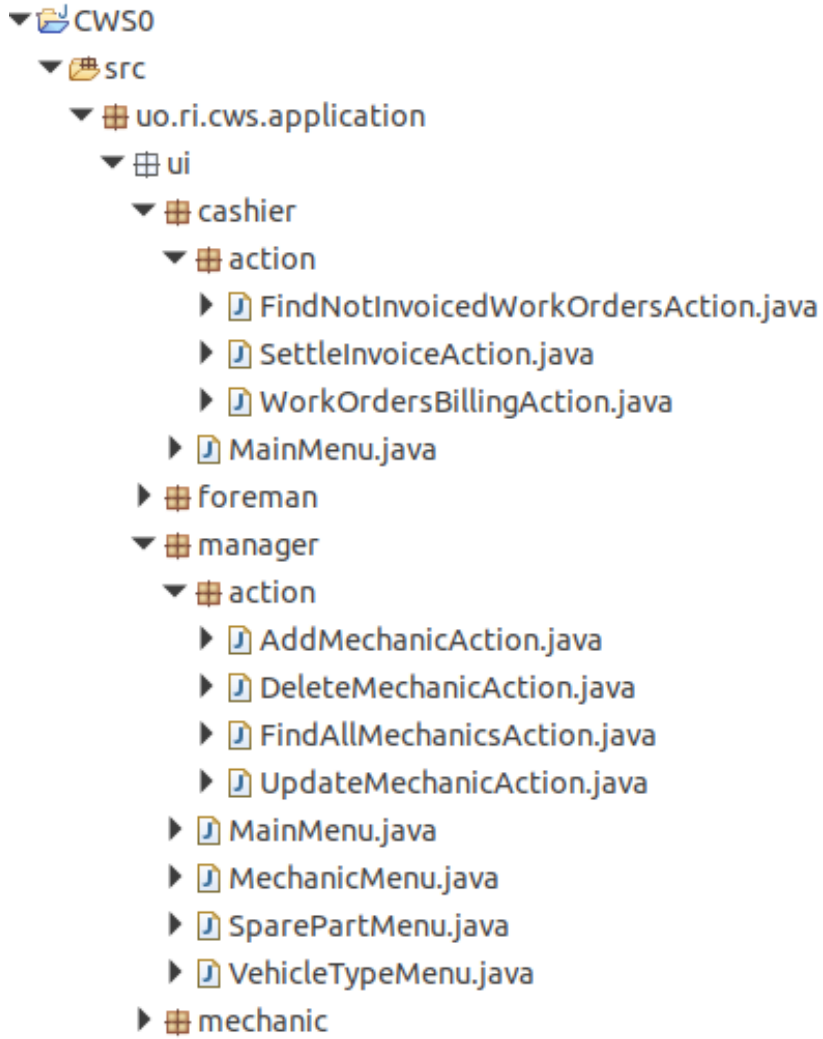
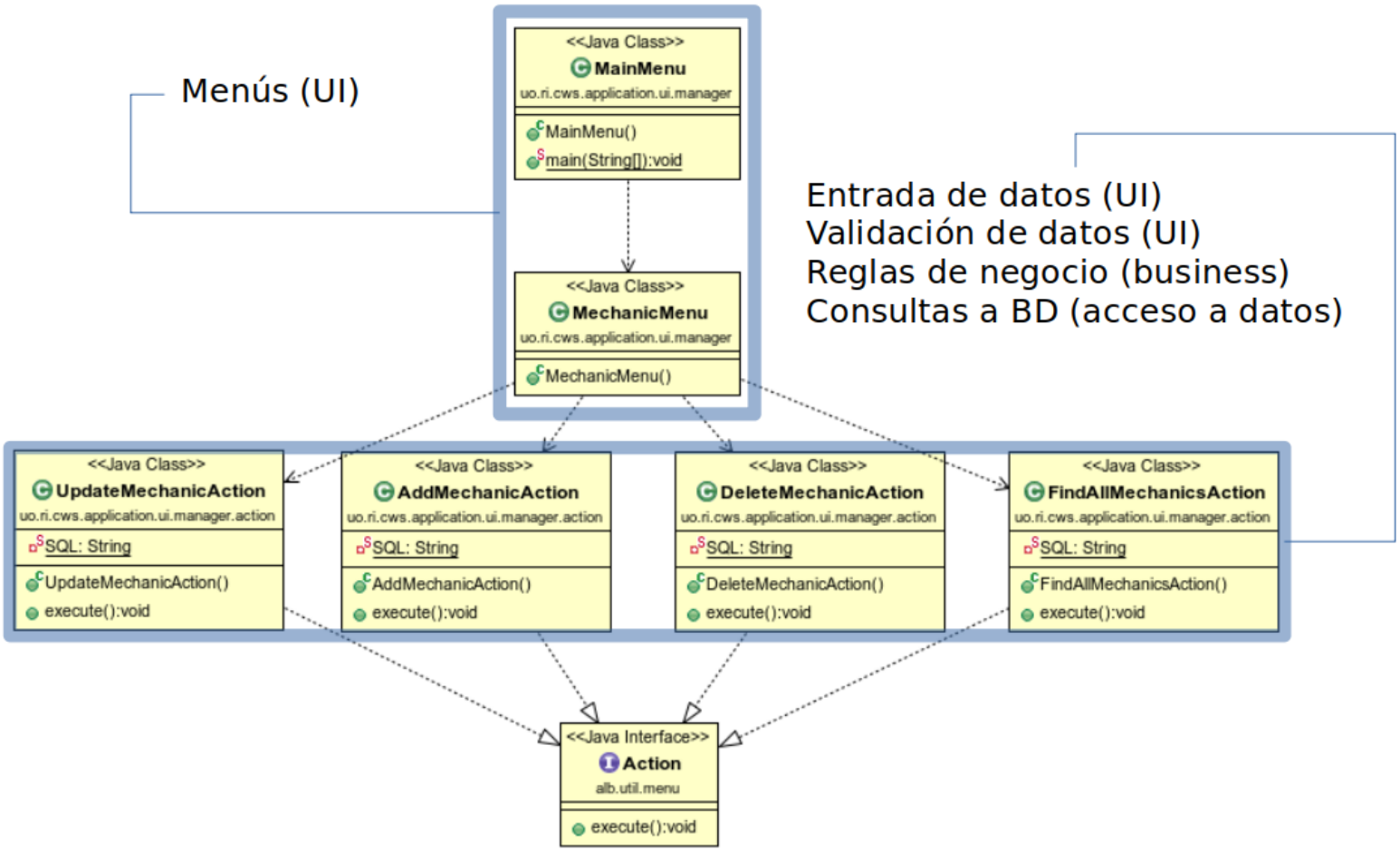


Diagrama de clases de Mechanics Management



Volver a 24

Diagrama de secuencia de Add mechanic

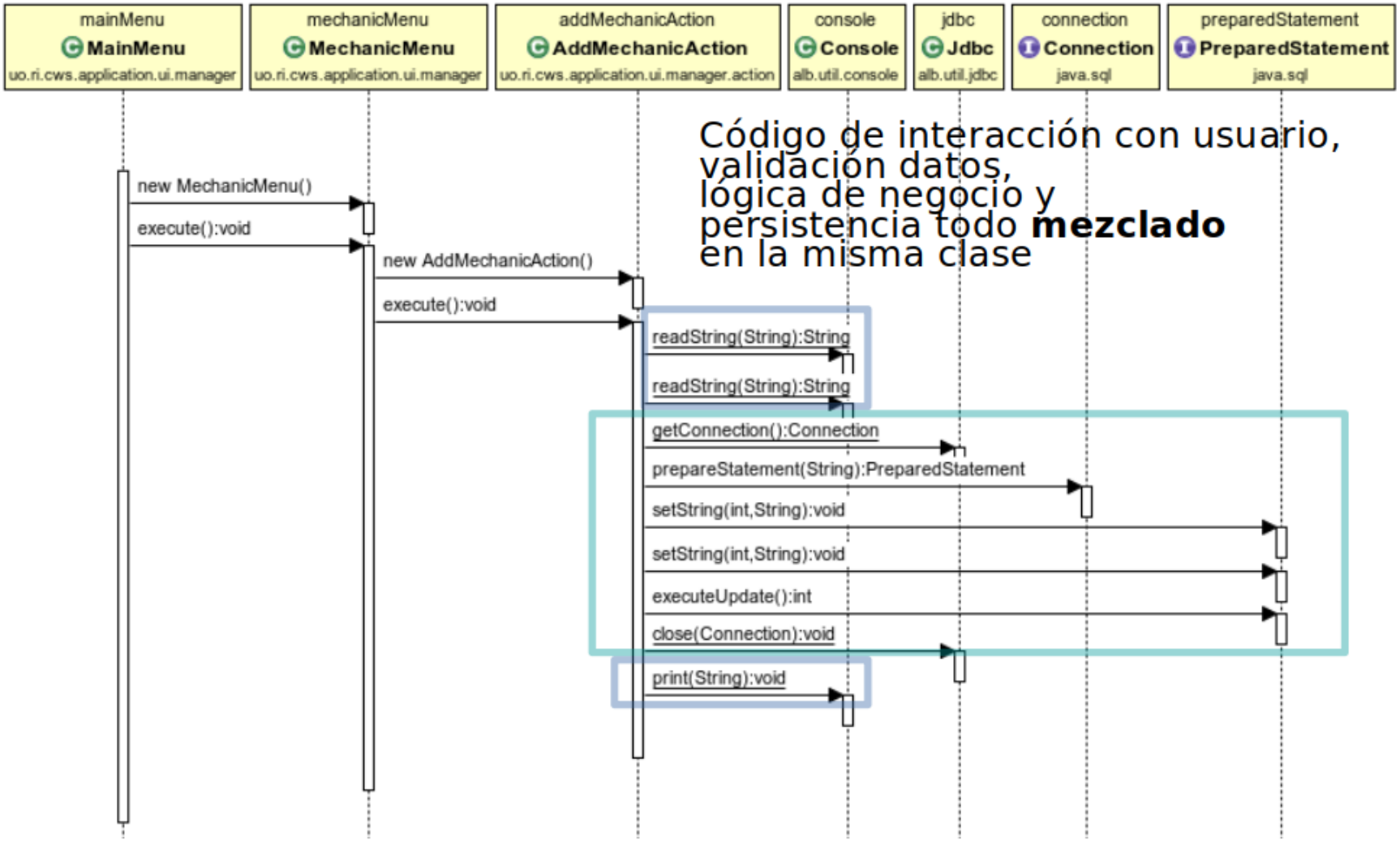
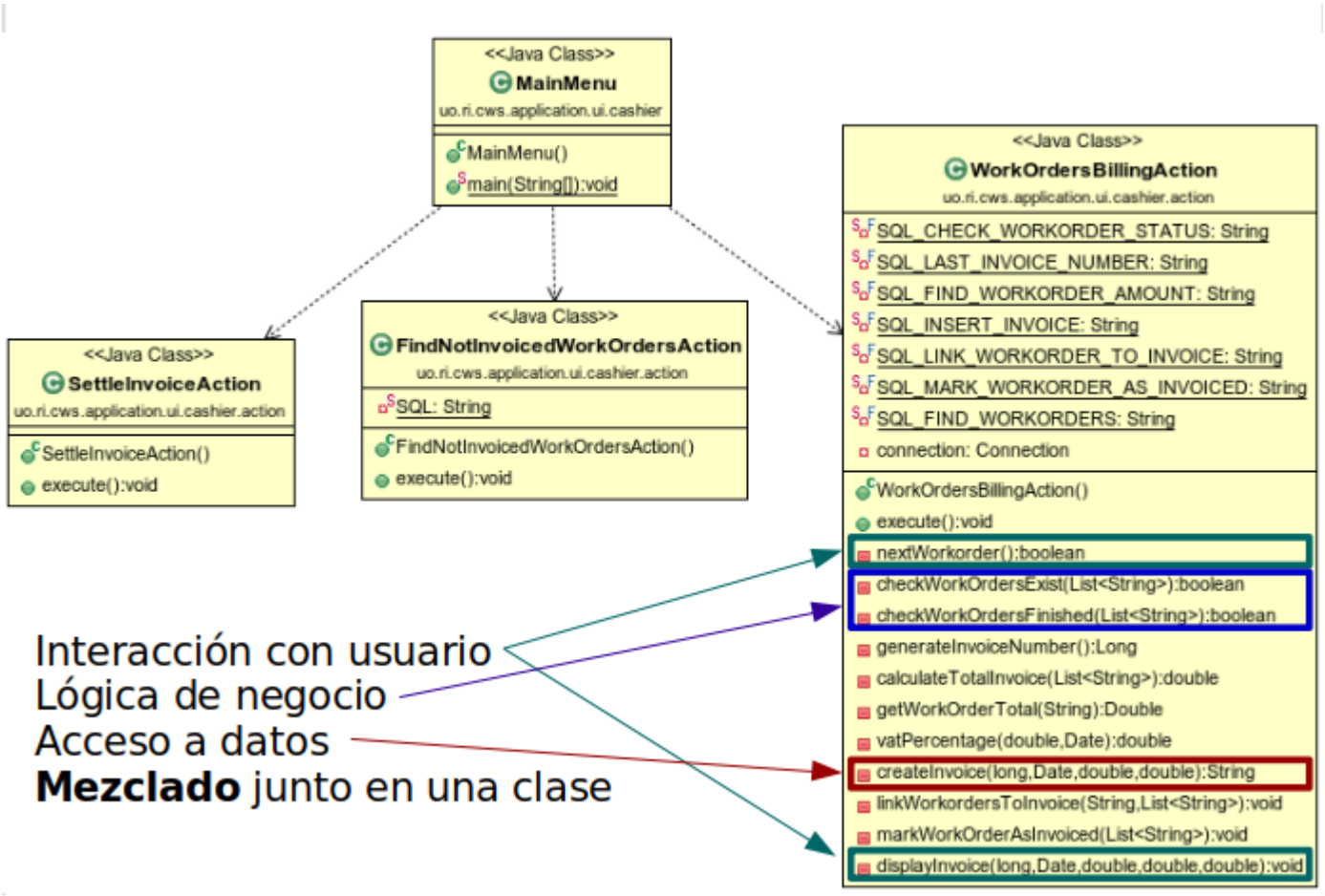


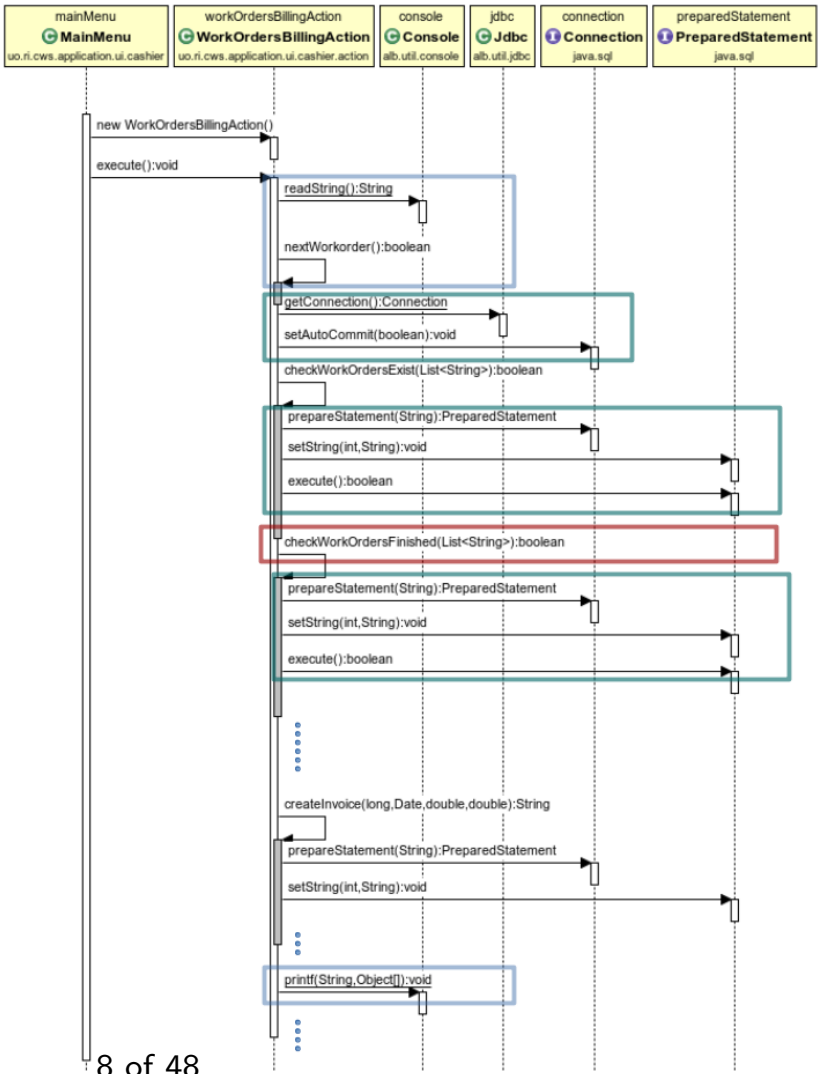
Diagrama de clases de WorkOrder Billing



Volver a 26

Create invoice sequence diagram

CWS0 implementation



- Interacción del usuario, validación de datos
- Código de lógica de negocios
- Código de acceso a datos
- Responsabilidades enmarañadas en la misma clase

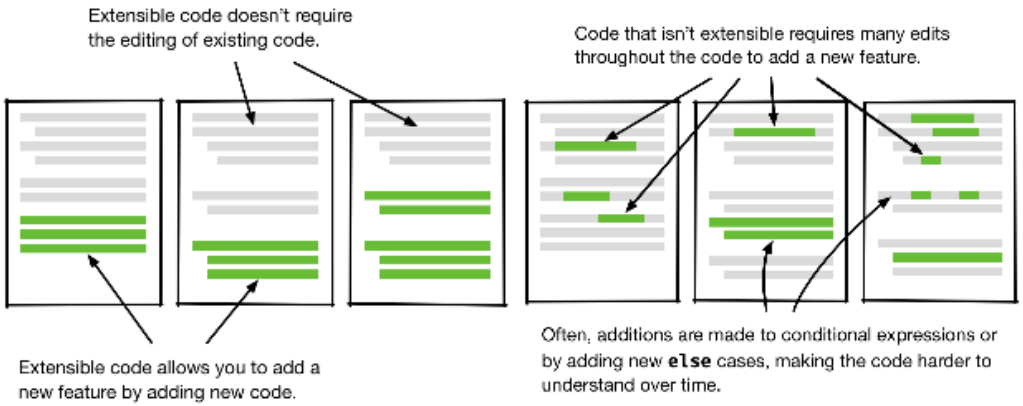
WorkOrdersBillingAction::execute()

```
// type work order ids to be invoiced in the invoice
do {
      String id = Console.readString("Type work order ids:  ");
      workOrderIds.add(id);
} while ( nextWorkorder() );

try {
      connection = Jdbc.getConnection();

      if (! checkWorkOrdersExist(workOrderIds) )
            throw new BusinessException ("Workorder does not exist");
      if (! checkWorkOrdersFinished(workOrderIds) )
            throw new BusinessException ("Workorder is not finished

      long numberInvoice = generateInvoiceNumber();
      Date dateInvoice = Dates.today();
      // vat not included
      double amount = calculateTotalInvoice(workOrderIds);
      double vat = addVat(amount, dateInvoice);
      // vat included
      double total = amount * (1 + vat/100);
      total = Round.twoCents(total);

      long idInvoice = createInvoice(numberInvoice, dateInvoice, vat, total);
      linkWorkordersToInvoice(idInvoice, workOrderIds);
      markWorkOrderAsInvoiced(workOrderIds);

      displayInvoice(numberInvoice, dateInvoice, amount, vat, total);

}
catch (Exception e) {
      throw new RuntimeException(e);
}
finally {
      Jdbc.close(connection);
}
```

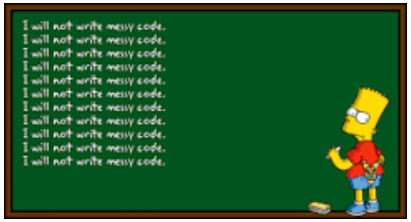
Section 2

Atributos de calidad para el software

- **Escalabilidad** A medida que el sistema crece (en volumen de datos o complejidad), qué alternativas razonables existen de lidiar con ese crecimiento. Ejemplo: Publicar un tweet lo inserta en colección global de tweets vs lo inserta en el timeline de sus seguidores
- **Mantenibilidad** son los costes relativos de reparación o actualización a lo largo de su vida útil Actualizaciones automáticas de los SSOO
- **Reusabilidad** es la capacidad de los elementos software para ser utilizados en la construcción de diferentes aplicaciones. Bibliotecas (Math), lenguajes de programación de alto nivel, widgets, ...
- **Extensibilidad** La facilidad de adaptar el software a los cambios de especificación sin modificar el software ya existente.



Evaluación de CWS0



Pobre Mantenibilidad

- Cambios (UI o persistencia) $\rightarrow$ cambiar *casi* todas las clases.
- El **código repetido** tiene un gran impacto en la mantenibilidad

Pobre Extensibilidad

- Agregar cifrado datos sensibles como datos de facturación o de clientes, provoca cambios en muchas clases (código enredado)

Pobre Reusabilidad

- Reutilizar código (acceso a datos) en alguna otra aplicación es imposible.

Pobre Escalabilidad

- Intentar optimizar el acceso a la base de datos (pool de conexiones) provoca muchos cambios.

Clases *Action implementan todo: interacción con el usuario + reglas negocio + acceso a datos

No **separación de asuntos o responsabilidades**

- El código correspondiente a un asunto no está encapsulado en un módulo sino **disperso** (*scattered*) entre muchos módulos.
- Es difícil localizar y comprender la implementación de un asunto porque está **enredado** con código que implementa otros.
- El **código se repite** en todos los lugares donde se necesita (acceso a una tabla específica).

La clave para crear código de calidad es adherirse al *principio de bajo acoplamiento, alta cohesión*.

- **Acoplamiento** evalúa cuán estrechamente se relaciona un módulo con otros. Cuántas **dependencias** hay entre módulos.
- **Cohesión** se refiere a cuán estrechamente están relacionadas las funciones en un módulo. **Responsabilidad única** (sin clases esquizofrénicas).

Section 3

Patrones

Patrones

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)

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.

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides

Muy recomendable:[1]

Aspectos comunes a muchas aplicaciones

- **Lógica de negocio compleja** Compuesta por componentes individuales que hacen algo concreto (nóminas, facturación, lista de clientes, . . .)
- **Involucran datos persistentes**
- **Sistemas intensivos en datos**
- **Acceso simultáneo a los datos**
- **Gran cantidad de pantallas de interfaz de usuario**
- **Necesidad de integración con otras aplicaciones** dispersas en el sistema de la empresa

Debido a eso, han surgido una serie de **patrones** que facilitan el desarrollo de estas aplicaciones.

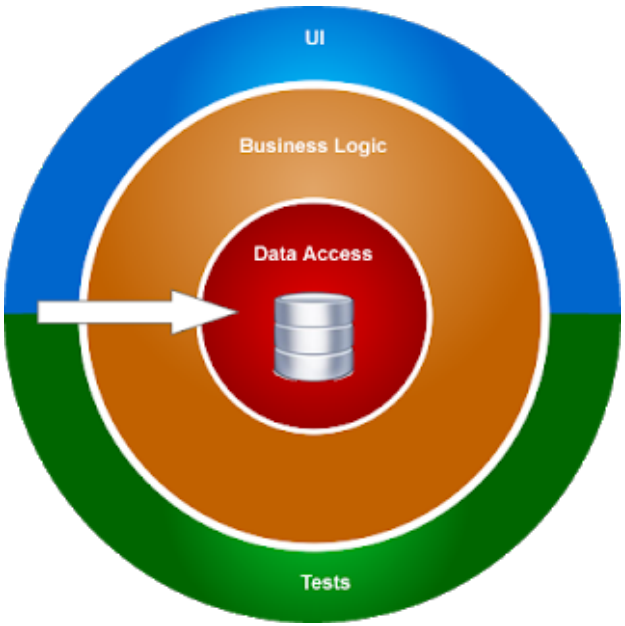
Section 4

Patrón arquitectónico: Separación en capas

Arquitectura de referencia

Secuencia ordenada de capas, donde cada capa ofrece servicios que pueden ser usados por componentes que residen en la capa superior.

- Cada capa expone a la capa superior una interfaz simple para usar un sistema potencialmente complejo.
- Cada capa llama a métodos de la API de la capa inferior



Responsabilidades organizadas en capas.

- La **capa de presentación** se ocupa de la interacción entre el cliente y la capa de negocio (interfaz amigable, solicitudes a la capa de negocio, . . .)
- La **capa de negocio** implementa las reglas del problema.
- La **capa de acceso a datos** interactúa con los datos persistentes.

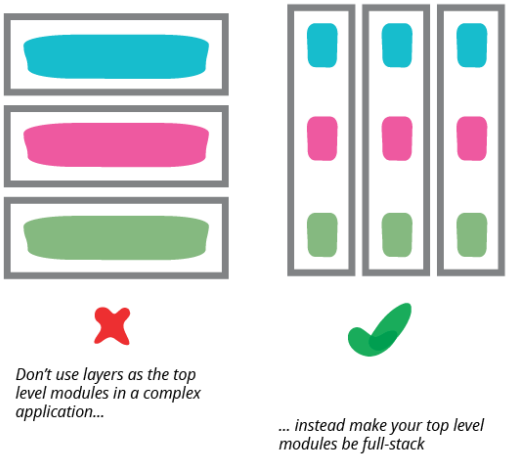
Ventajas

Recomendable: [2]

- **Reutilización**: Una vez construída, una capa (ej, capa de negocio) puede ser utilizada por muchos clientes distintos (interfaces gráficas, otras aplicaciones, tests)
- **Cohesión**: Cada capa se ocupa de una responsabilidad (no esquizofrenia), tiene un propósito.
- **Mantenibilidad**: Mientras se mantenga su interfaz, las capas ignoran los cambios en la implementación de otras capas.

Desventajas

- No siempre es posible una división estricta (transacciones)
- Las capas adicionales pueden degradar el rendimiento.
- La arquitectura en capas exige dependencias siempre hacia abajo.
  - Cambios en la implementación de la capa de negocio no deben suponer cambios en la capa de acceso a datos o interfaz de usuario



Aplicable en una granularidad relativamente pequeña.
Lectura recomendada:
`https://dzone.com/articles/reevaluating-the-layered-architecture`

Primera evolución de CWS0: arquitectura en capas

- Dividir la aplicación en dos capas (cambios profundos)
  - **Capa Presentación o UI**: entrada/salida de datos (validaciones)
  - **Capa Business**: Reglas de negocio + Persistencia

  Cada capa da soporte a una capa superior.

  Cada capa tiene soporte de una capa inferior. **Las llamadas a métodos son todas en una dirección**.

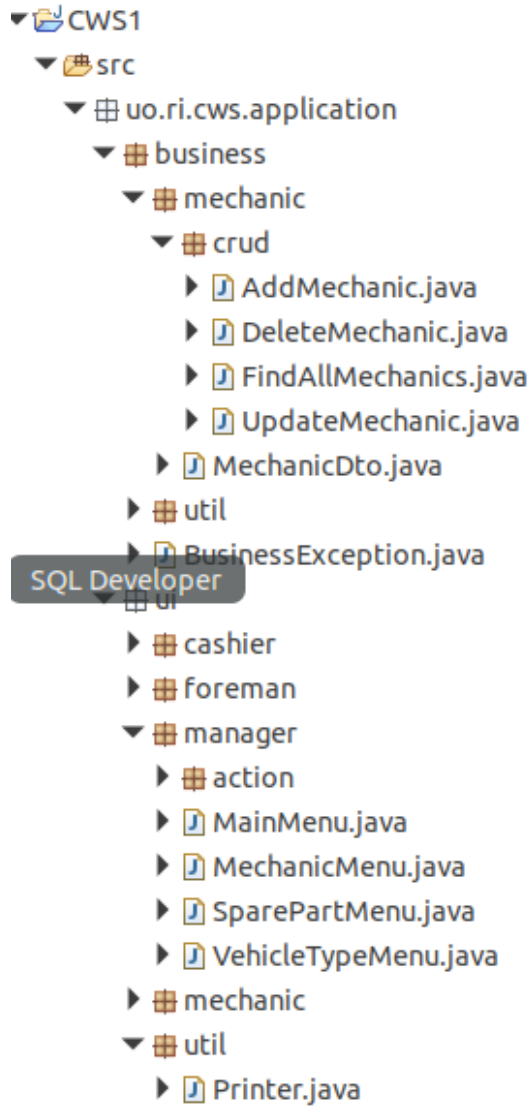
- Para pasar argumentos, se usa **Data Transfer Objects** o **DTO**

1

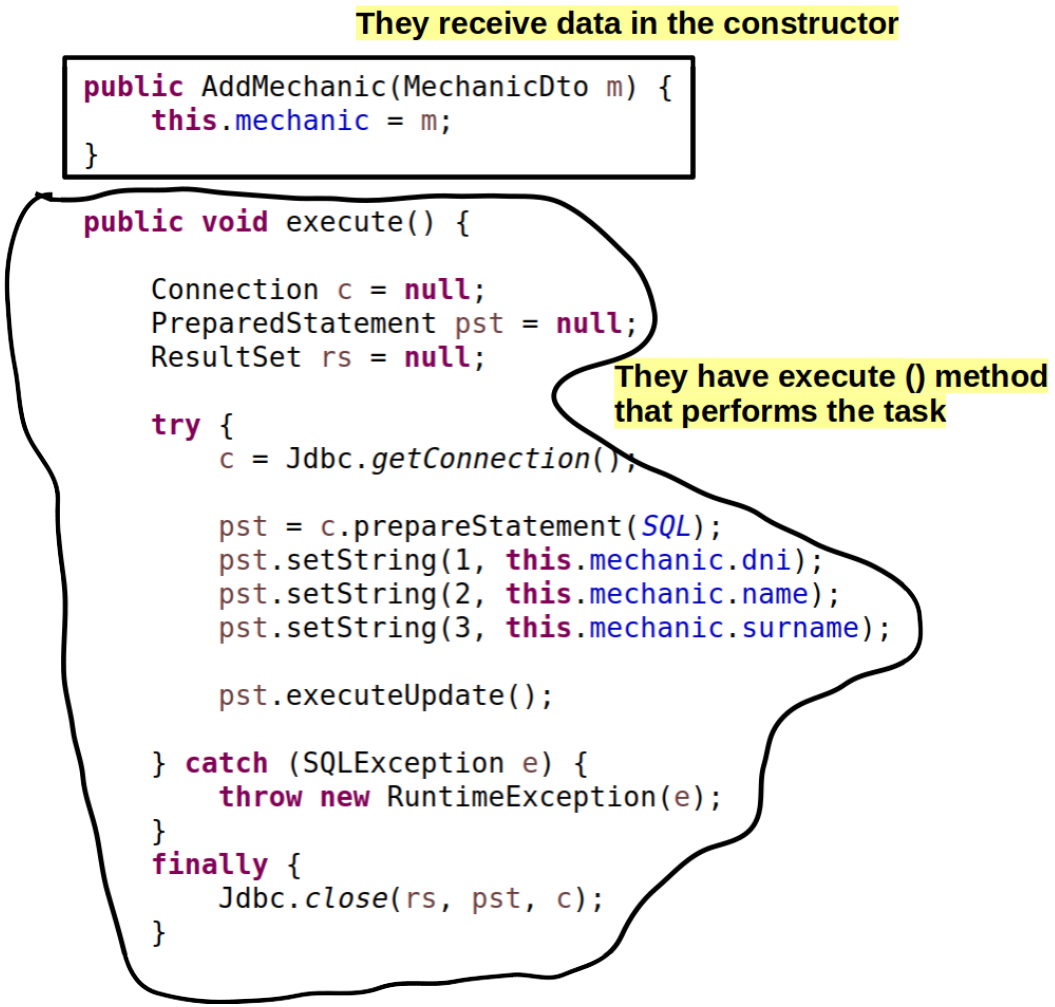
[1] En los siguientes diagramas UML, podría haber algunos errores tipográficos relacionados con los nombres de paquetes, clases o métodos.

Estructura de paquetes

- **UI**. Clases `ui.***.action` implementan **interacción con usuario**, algunas validaciones de datos, e invocan clases de negocio.
- **Business**. Lógica de negocio + persistencia. Paquete organizado en subpaquetes, en general, **por caso de uso**.



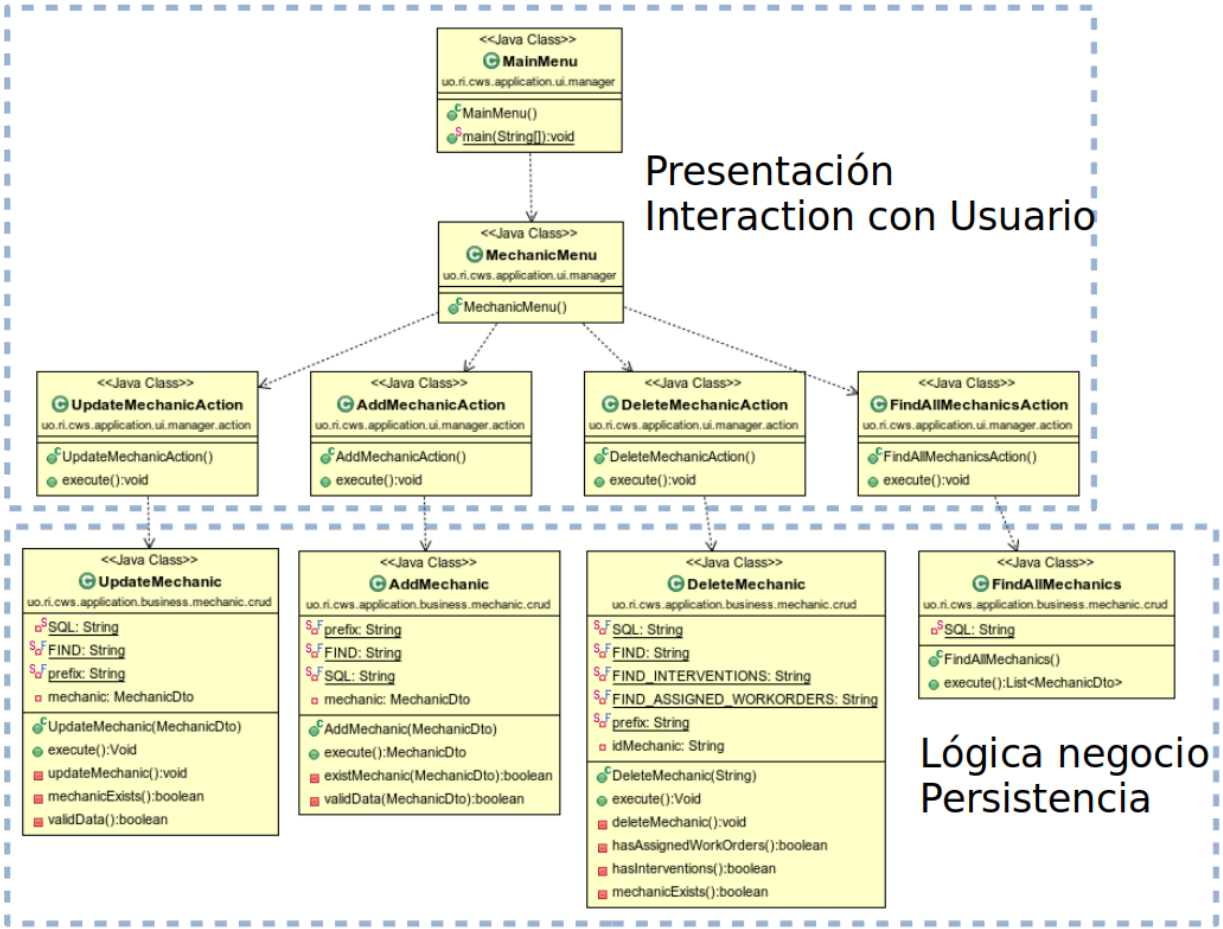
Clases en la capa business



- Las clases de la capa business implementan las reglas de negocio (`AddMechanic`, `ListMechanics`, ...)
- Todas tienen una apariencia parecida.

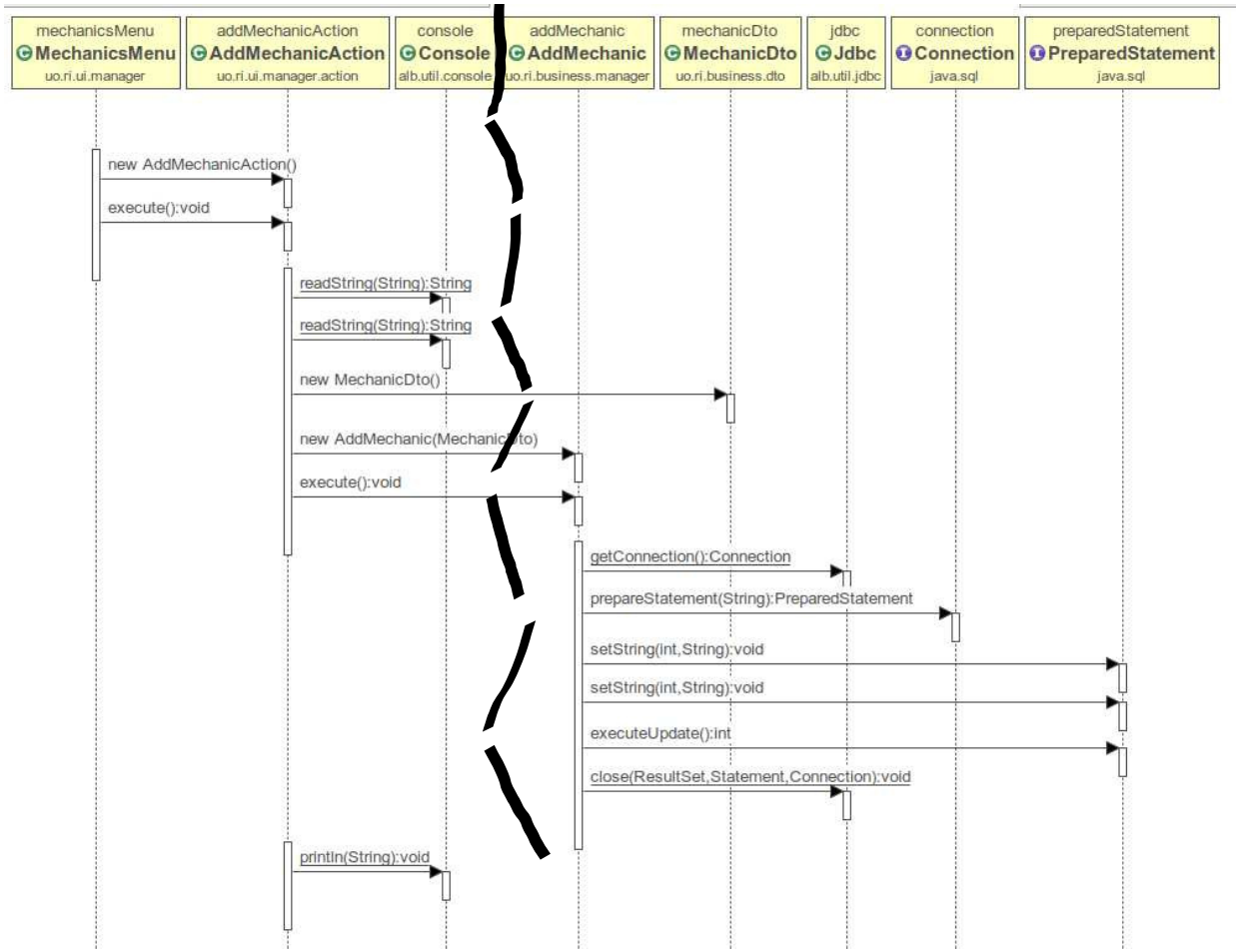
Diagrama clases Gestión de Mecánicos (CRUD)

Código de IU separado del código de negocio y acceso a datos



(compara con 5)

Diagrama de interacción Gestión de Mecánicos



Caso de uso Workorder billing

Crear una factura para una o más workorders

- Dibuja un diagrama de clases para el caso de uso con separación de asuntos: Presentación – Negocio (+persistencia))
  Ir a 7

- Dibuja un diagrama de secuencia para el caso de uso con separación de asuntos

  En tiempo de compilación, ¿ qué dependencias tendría el cliente? (imports)

Objetivo: Dividir el código en capas diferentes según el asunto del que se ocupe.

Section 5

Patrón Fachada (Façade)

Problemas que quedan

La capa de presentación **depende de la implementación de las clases** que forman la lógica de negocio (**Acoplamiento**).

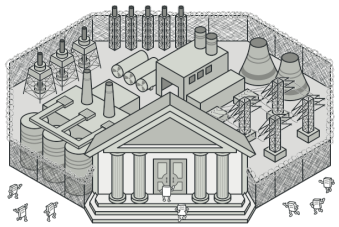
- El código de la capa UI **crea instancias** de las **clases que implementan** la capa de negocio y **llaman a los métodos**.
- Cambios en la implementación de una clase de la capa de negocio supondrá cambios en clases de la capa de presentación y viceversa.
- Ir a 23. Imaginar cómo sería la llamada del cliente.

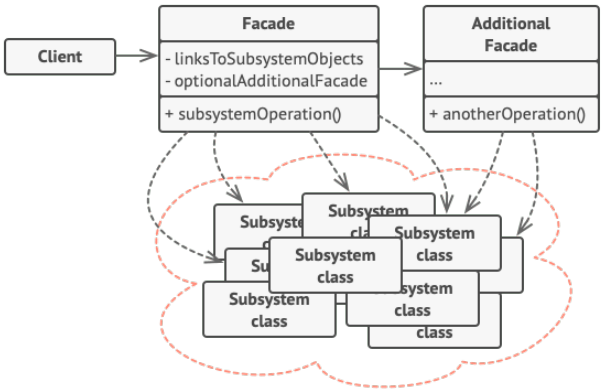
Algunas veces, **clases de la capa de UI hacen referencia a muchas clases en la capa de negocio**, con interfaces diferentes, lo que hace difícil implementar, cambiar, testear y reusar las clases. Por ejemplo, `PayOffInvoice`

El patrón Fachada

Es un **patrón de diseño estructural** análogo a una fachada en arquitectura.

Proporciona un **interfaz sencilla** (pero limitada) a un conjunto de objetos existentes (**sistema complejo**).





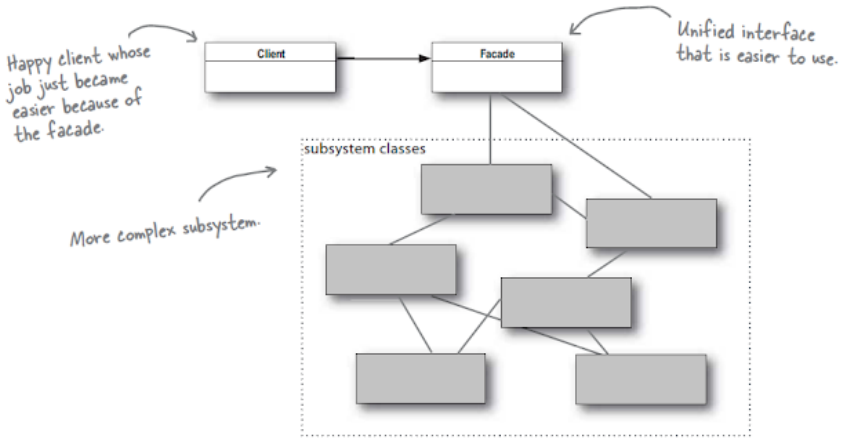
- Los clientes interactúan con el subsistema a través de la fachada pero ésto **no impide el acceso a las clases del subsistema**.
- Podría haber más de una interfaz para el mismo subsistema.

Ventajas

Reduce el acoplamiento entre subsistemas y entre clientes y subsistemas

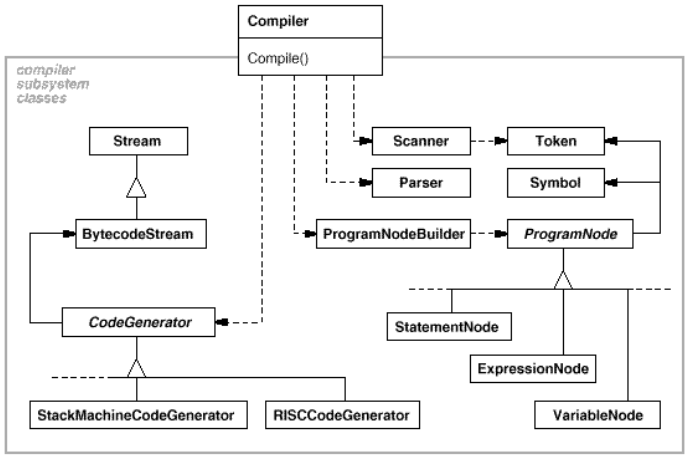
```
public class Client {
  public void method( ) {
      SubsystemClass sc =
        new SubsystemClass();
      sc.doSomething();
    }
}
```

```
public class Client
{
  public void method( ) {
      new Facade().
         callDoSomething();
    }
}
```



Una fachada también **oculta los detalles** detrás de la interfaz. Es posible **cambiar la implementación de un subsistema**, mientras se siga manteniendo la **interfaz**

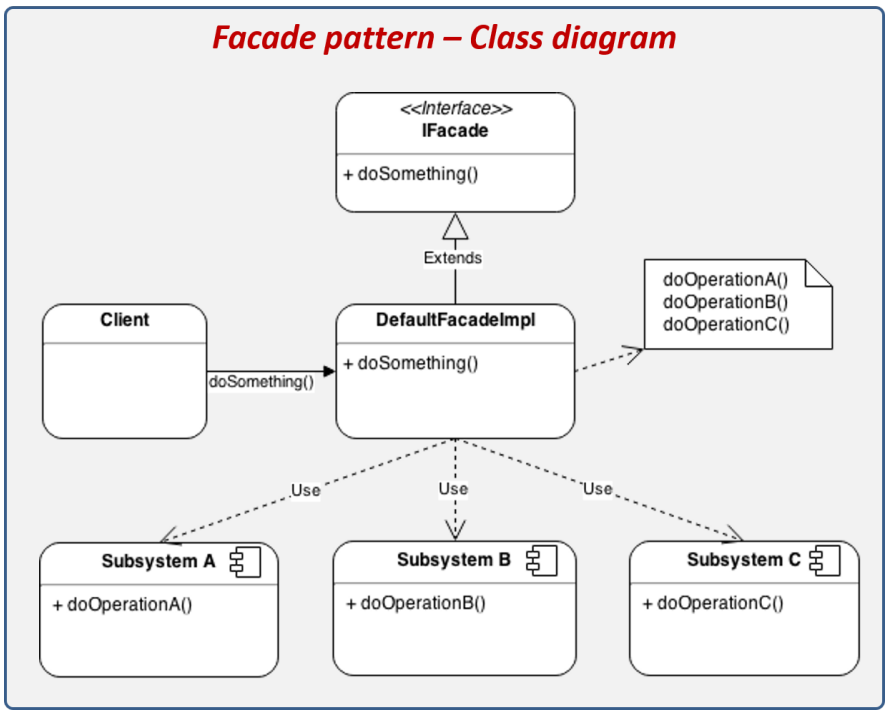
```
public class Client
{
  public void method( ) {
    Scanner s = new Scanner();
    Parser p = new Parser();

    s.scan();
    p.parse();
    }
}
```




```
public class Client
{
  public void method( ) {
      new Compiler().
          compile();
    }
}
```

Diagrama de clases

- **Client**: Sistema que interacciona con la fachada.
- **IFacade**: Interfaz que define los métodos de acceso a las clases de negocio que implementan una operación de alto nivel (pagar factura).



- **DefaultFacadeImpl**: Implementación de la interfaz (IFacade). Crea instancias de clases de negocio, invoca a sus métodos, ...
- **Subsystems**: Clases que implementan las reglas de negocio.

Patrón Fachada en el caso de uso Mechanics Management

- Crear una **interfaz** (`XXXService`) por caso de uso.
- **Implementar** interfaz (`XXXServiceImpl`)

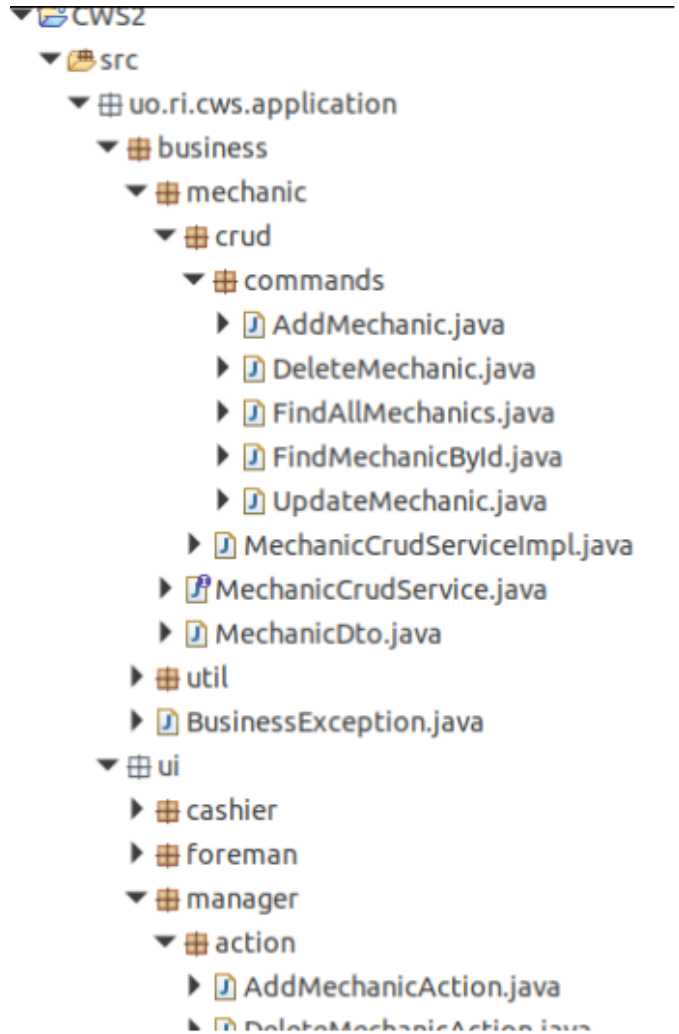


Diagrama de clases

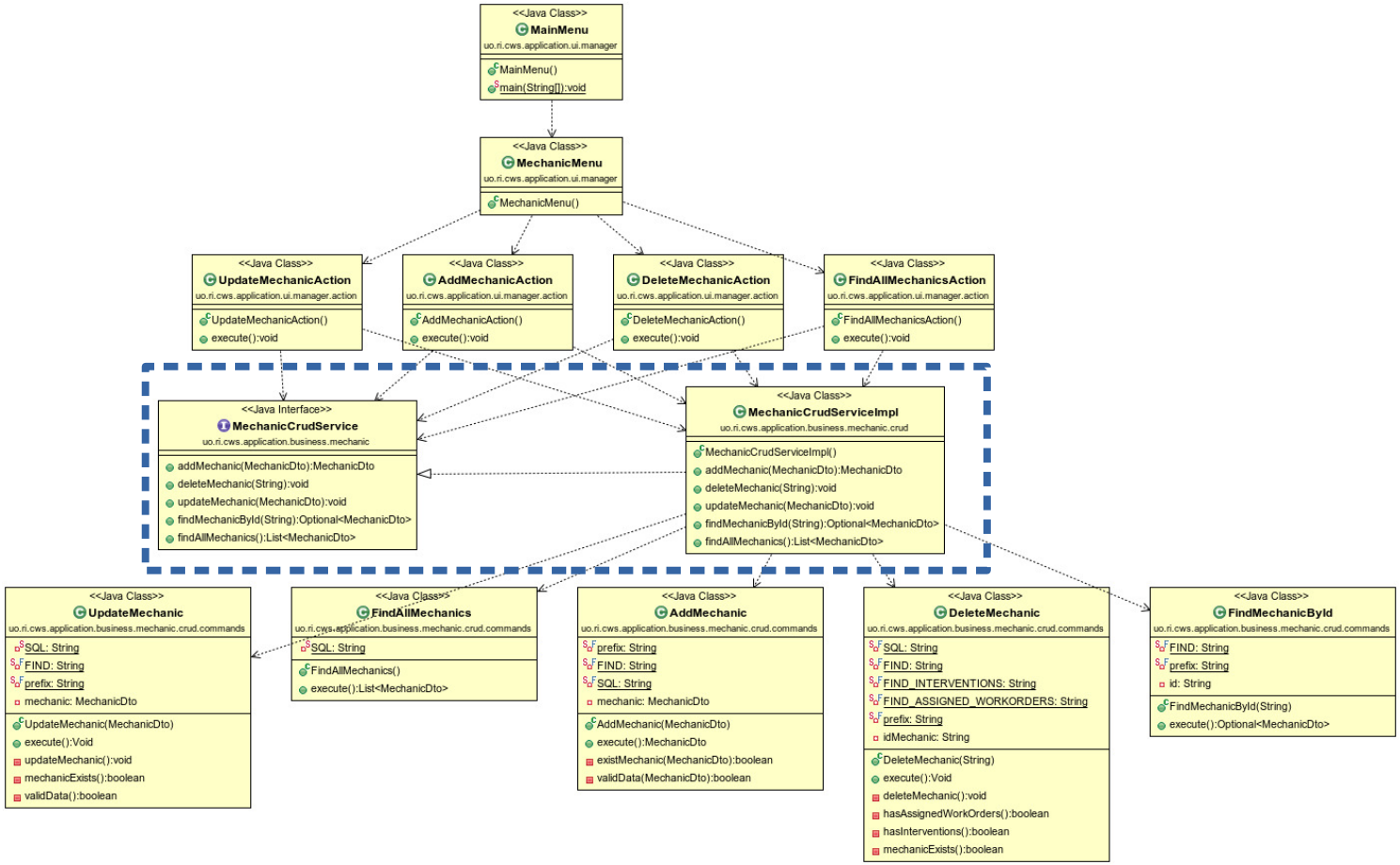
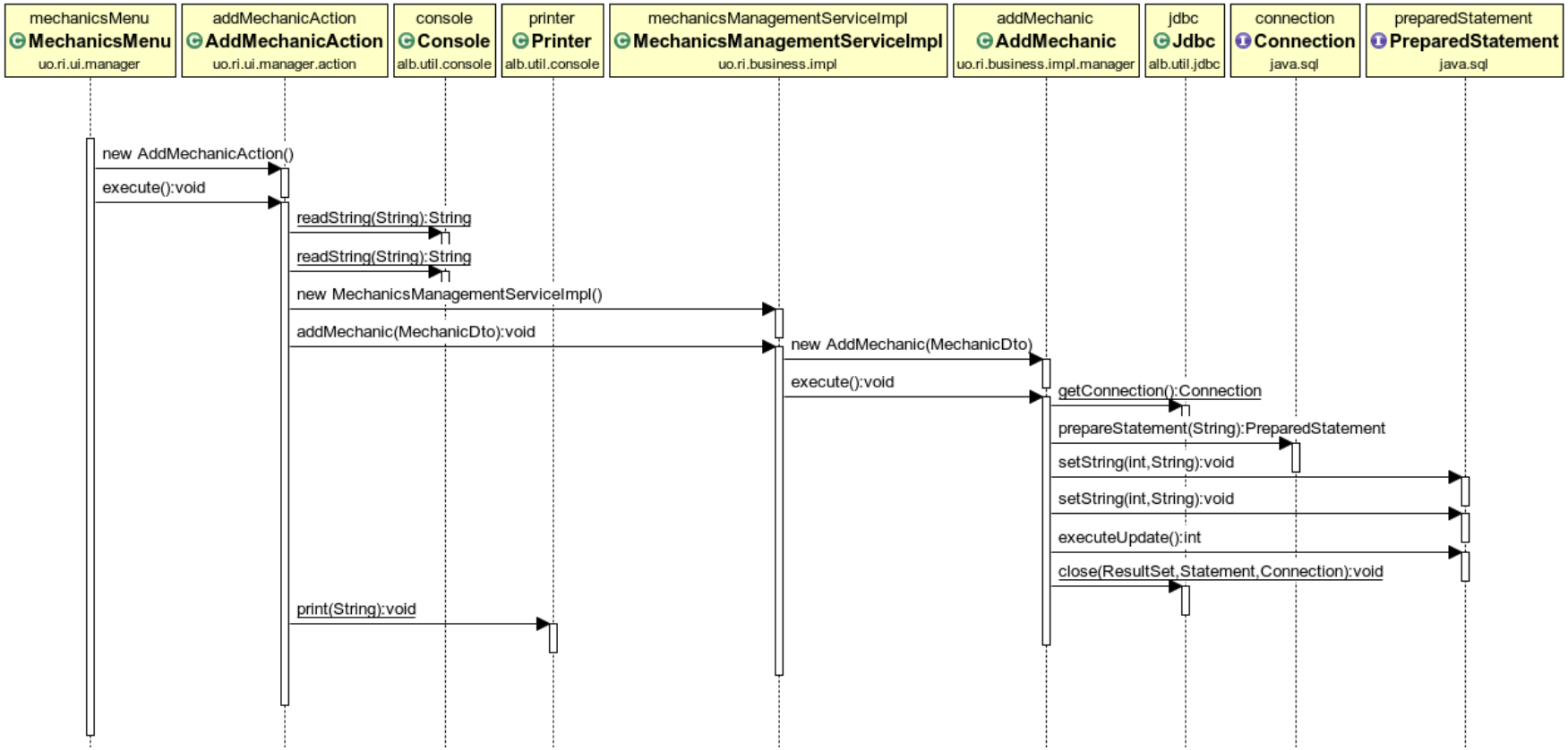


Diagrama de secuencia



Caso de uso Workorder Billing

- Dibuja un diagrama de clases para el caso de uso, con una fachada
- ¿ Qué clases incluye la capa UI y cuáles la capa de negocio?
- De las clases de la capa de negocio, identifica a la interfaz de la fachada, la implementación, el cliente y los subsistemas.
- Dibuja un diagrama de secuencia para el caso de uso.

Section 6

Factoría Simple (o Class Factory)

Problemas restantes

Todavía hay **acoplamiento entre la capa de interacción del usuario y la capa de negocio**. Estas líneas de código

```
MechanicService s = new MechanicServiceImpl( );
s.addMechanic();
```

No están menos acopladas que estas:

```
AddMechanic m = new AddMechanic();
m.addMechanic();
```

- Las clases de UI **dependen de la implementación** de clases de la capa de negocio porque las clases de `***Action` necesitan saber el **nombre de la clase que implementa** la fachada para crear una instancia.
- Cambiar esta clase obliga a cambiar las clases de presentación.

El patrón **Simple Factory** evita este tipo de dependencias[6]

Patrón Simple Factory

Una clase Simple Factory es una clase que **crea objetos sin mostrar al cliente la lógica de creación de instancias**.

Los métodos de creación a menudo se declaran estáticamente.

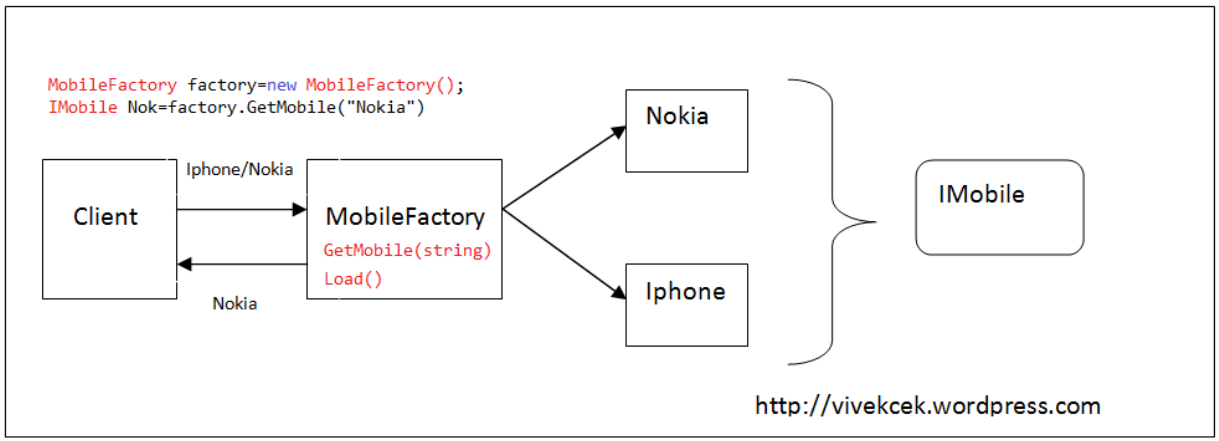
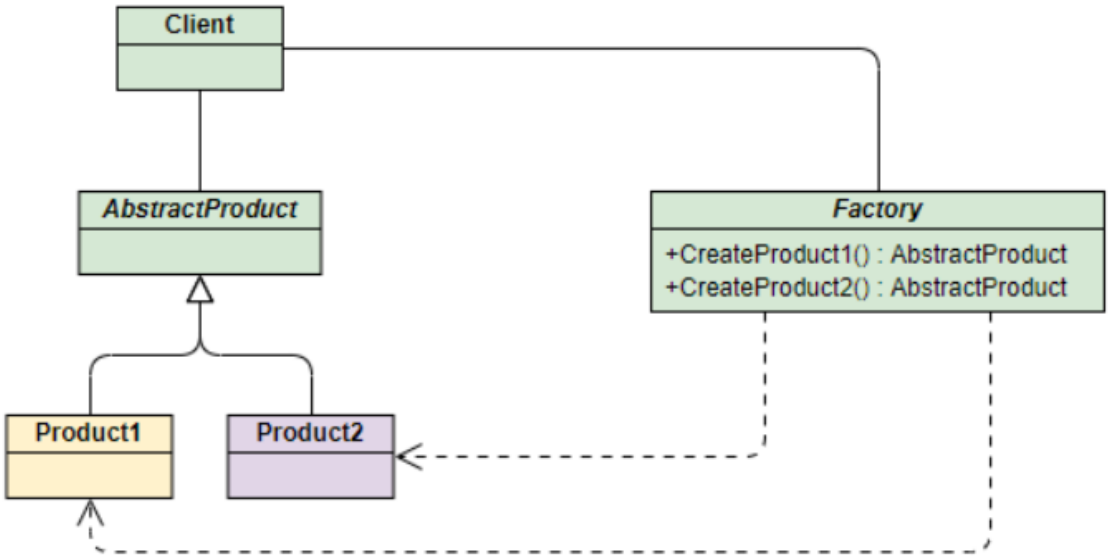


Diagrama de clases



Ventajas

- Cuando el cliente necesita una instancia de una clase, le pide a la factoría que cree una. El cliente sólo conoce la interfaz de la clase y la clase factoría.
- Cambiar la implementación de la clase no significa cambios en el cliente (capa de presentación).
- La factoría puede adaptar sin cambiar su interfaz (solo cambiando la implementación de `getXXXService()`).

Estructura de paquetes Mechanics Management con Class Factory

2. UI solicita una clase service (`forXXXService`) a `BusinessFactory`
3. `BusinessFactory` retorna una instancia de una clase que implemente la interfaz del servicio solicitado (`XXXServiceImpl`).
4. Las clases UI conocen `BusinessFactory` y las interfaces de los services (`XXXService`); no las clases que implementan el servicio.

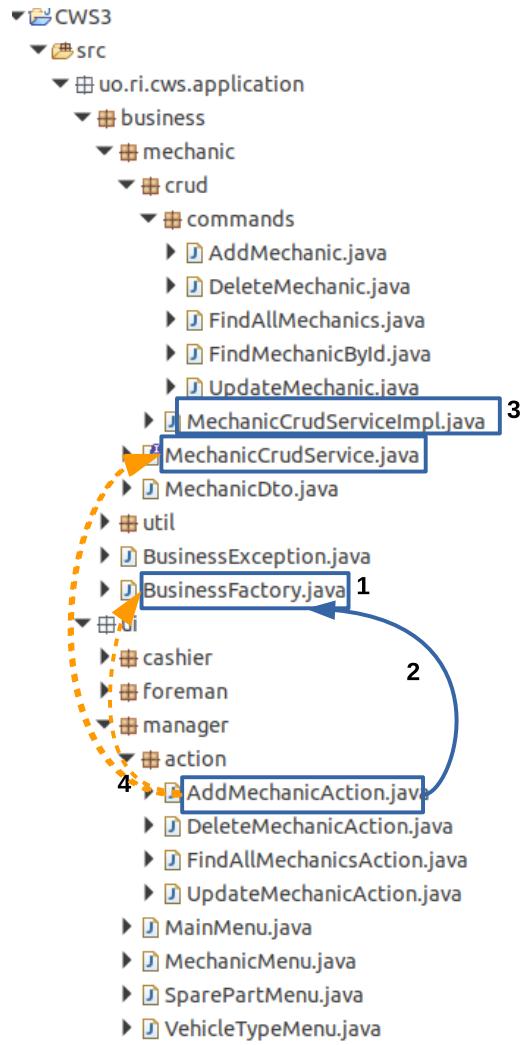
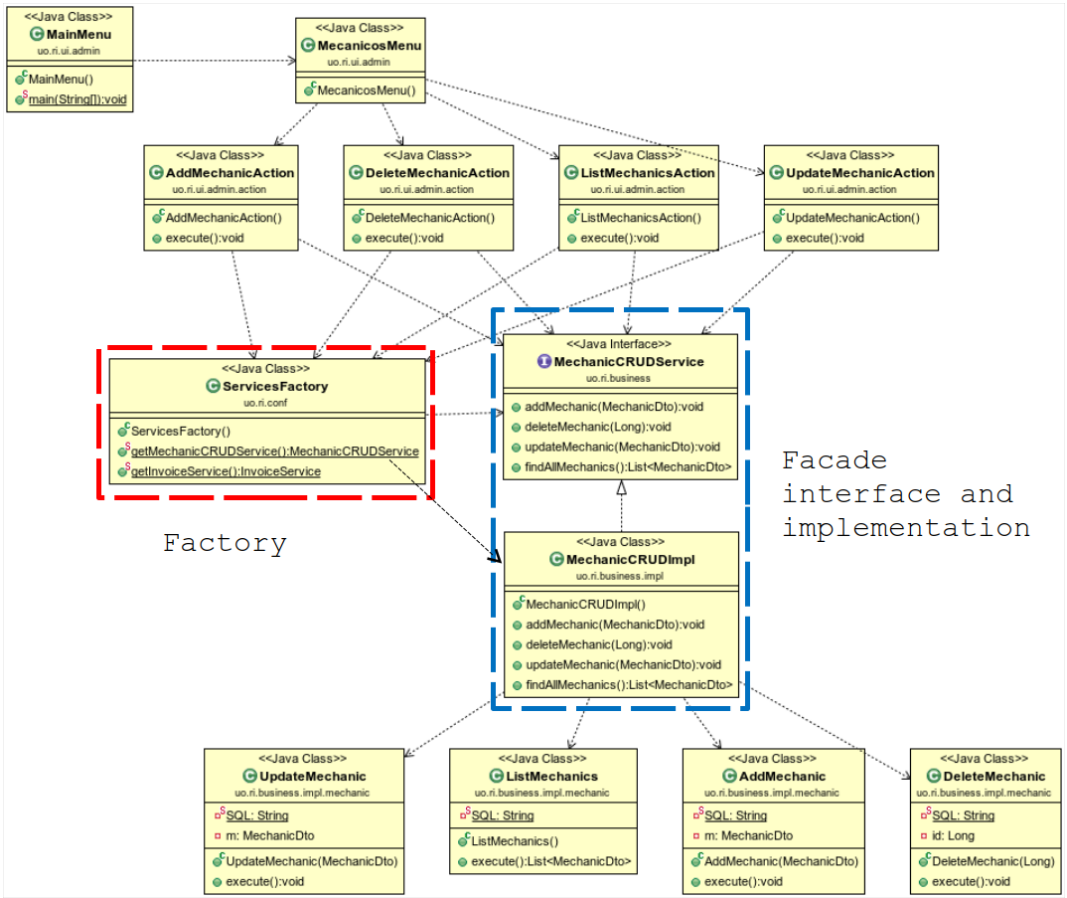
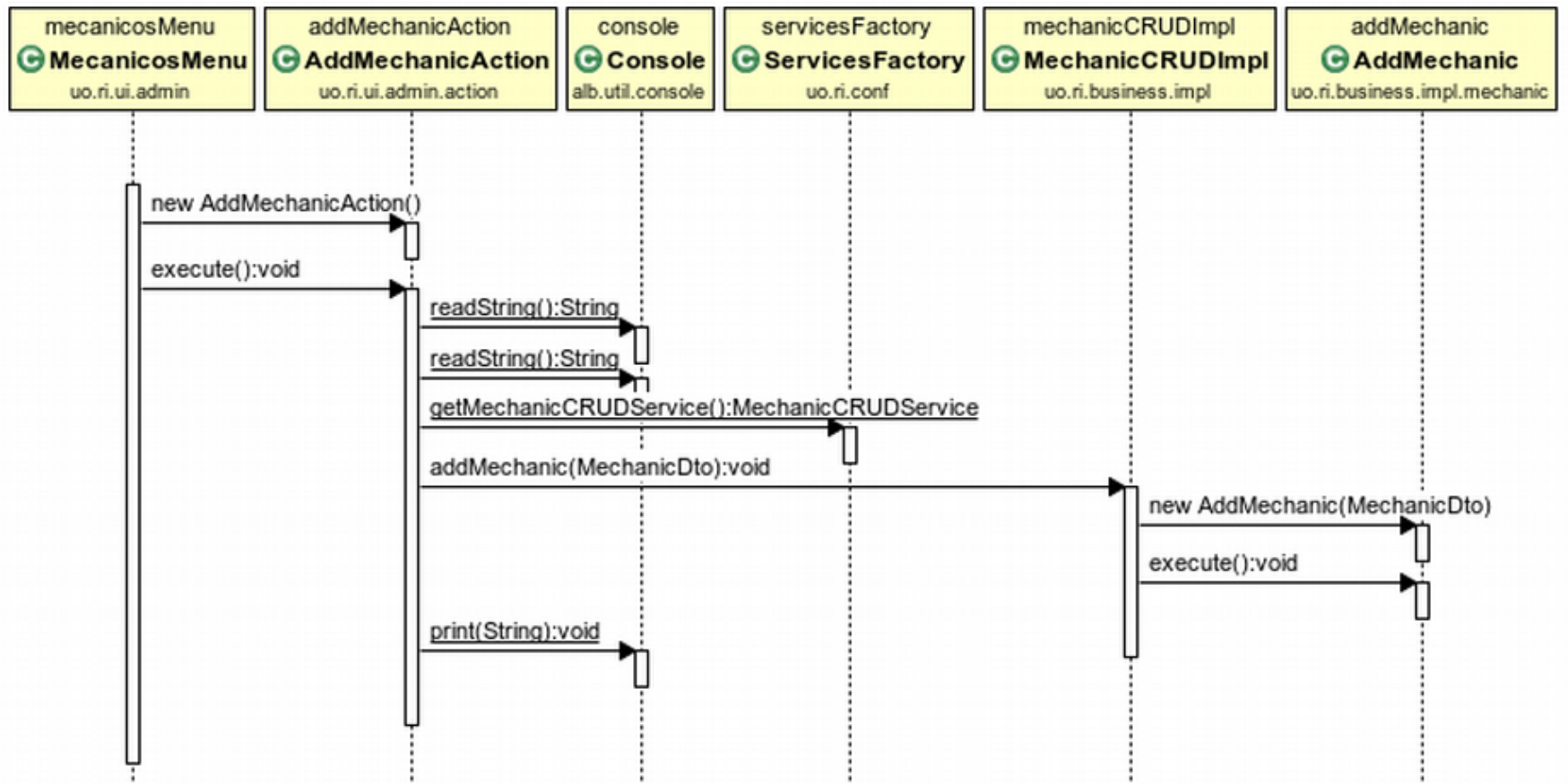


Diagrama de clases



- `BusinessFactory` retorna una fachada.
- Cambiar la implementación del servicio implica únicmante cambiar el método `forXXXService`.

Diagrama de secuencia



Caso de uso Workorder Billing con Class factory

- Dibuja un diagrama de clases para el subsistema invoicing incluyendo una fachada.

- Dibuja un diagrama de secuencia para Workorder Billing.

Further Reading (1)

[1] https://www.youtube.com/watch?v=8Zoz2njk5f8

[2] https://www.oreilly.com/library/view/software-architectu

[3] https://javadevguy.wordpress.com/2019/01/06/reevaluating-the-layered-architecture/

[4] https://martinfowler.com/bliki/PresentationDomainDataLayering.html

[5] https://refactoring.guru/design-patterns/façade

[6] https://javajee.com/factory-design-patterns-simple-facto

Further Reading (2)

[7] https://www.oracle.com/java/technologies/data-access-obj

[8] https://www.martinfowler.com/eaaCatalog/tableDataGateway

[9] https://www.oracle.com/technetwork/java/dataaccessobject-138824.html

[10] https://martinfowler.com/eaaCatalog/gateway.html

[11] https://www.sourcecodeexamples.net/2018/04/row-data-gateway.html

[12] https://martinfowler.com/eaaCatalog/gateway.html

Further Reading (3)

[13] http://what-when-how.com/Tutorial/topic-263rdl/POJOs-in-Action-Developing-Enterprise-Applications-with-Lightweight-Frameworks-352.html