ABSTRACCIÓN DE LOS SERVICIOS DE ACCESO A LA INFORMACIÓN

1. Planteamiento.

En cualquier ámbito, el diseño consiste en la elaboración de unos elementos, en dotarlos de un comportamiento o una función y en organizar ese comportamiento para que, a partir de la colaboración entre los elementos, se obtenga la funcionalidad deseada.

En lo que convencionalmente se denomina desarrollo de software, el diseño se refiere a la elaboración de código, a la asignación de su comportamiento y a la organización de la colaboración e interacción entre sus partes para obtener el resultado esperado.

A partir de aquí, las diferencias entre las variaciones del diseño se refieren al ámbito en el que se trabaja (presentación, servicios técnicos de apoyo, dominio del negocio) y a la perspectiva con la que se afronta (la granularidad del desarrollo: diseño detallado y diseño arquitectónico).

Por descontado, puede interpretarse que su faceta *creativa* dota al diseño de cierta libertad y subjetividad. Pero todo eso se circunscribe a la obtención del resultado deseado: el comportamiento, en cualquier aspecto, del producto final. Además, la única *materia prima* que se utiliza en los procesos de transformación del diseño es el código; razón por la que los grados de libertad de las creaciones en los diseños están estrictamente restringidos al comportamiento de una organización, final y concreta, de ese código (a la implementación o realización de esos diseños).

En resumen: las tareas del diseño de software consisten en organizar el código según unas especificaciones y objetivos que se refieren al resultado obtenido.

Volviendo a la granularidad en el diseño, una vez asumido que su labor se ejerce, exclusivamente, sobre el código, el diseño detallado se dirige a organizar las líneas de código para que su comportamiento satisfaga la realización de alguna función u operación que se haya definido como parte del comportamiento global. Evidentemente, estas construcciones están fuertemente condicionadas por el propio código y las capacidades del lenguaje utilizado. Esta organización lleva a la agrupación del código en *unidades funcionales*; y cada definición de la función que realizan (que también forma parte de la *creación* en el diseño) deriva en la colaboración con otras funciones y operaciones (realizadas en otras unidades) que, como resultado, producen el comportamiento global o uno con una semántica más amplia que el de la unidad funcional (llamémosle módulo).

A su vez, para obtener el comportamiento deseado en el sistema, se puede definir el comportamiento o la funcionalidad en el ámbito de los módulos (subsistemas, paquetes, componentes, etc.), cómo se organizan y cómo es la colaboración entre ellos. Aún más: la organización de esos módulos se puede presentar desde distintas perspectivas, para ilustrar la descripción del funcionamiento y agruparlos por capas, bloques u otras distribuciones. Esto es lo que hace el diseño arquitectónico. Aquí, el campo de trabajo es el comportamiento de los módulos y la organización de la colaboración entre ellos, por lo que el diseño no está tan supeditado al código o al lenguaje de codificación; pero sí está estrechamente vinculado al comportamiento del módulo y a su construcción.

Por último, hay una dependencia bidireccional y absoluta entre el diseño detallado y el diseño arquitectónico. Si el comportamiento global del sistema que se desarrolla depende de cómo se defina la colaboración entre los módulos (diseño arquitectónico), significa que hay interacción entre ellos y, por tanto, entre los elementos que los constituyen (unidades funcionales). Esto quiere decir que no se pueden entender ni construir las funcionalidades de las unidades que componen un módulo (diseño detallado) sin definir la interacción que existe entre ellos (diseño arquitectónico) y sin tener muy en cuenta cómo se produce y cuáles son los *artefactos* que se intercambian (información o servicios).

Aunque el libro y el programa de la asignatura recorren con amplitud un buen número de aspectos del diseño, detallado y arquitectónico, el objetivo de mantener a ultranza la naturaleza práctica de la asignatura impone la necesidad de simplificar y acotar, dramáticamente, los escenarios de trabajo en las evaluaciones.

En la evaluación de esta asignatura, el espacio de trabajo al que se refieren los enunciados se corresponde con el **diseño detallado** de la capa **del dominio de la lógica de la aplicación** (véase un ejemplo, simplificado y sólo para la Venta, de una descomposición arquitectónica, por capas, en la página 422 o 423 del libro). Además, el desarrollo que se pide en dichos enunciados se circunscribe a un caso de uso, concreto, definido y simplificado (en ocasiones '*recortado*'), ubicado en algún módulo, paquete o subsistema de esa capa.

Por consiguiente, la labor fundamental para las respuestas de los ejercicios consiste en construir la especificación del funcionamiento (del código) de forma que se comporte según se describe en el enunciado, para ese caso de uso concreto, y restringido al ámbito del módulo, bloque o paquete que le corresponda, dentro de la arquitectura general de la capa del dominio de la lógica del negocio.

La comprensión del funcionamiento en el nivel de las unidades funcionales que constituyen un módulo depende sustancialmente del funcionamiento de otros módulos que interaccionan con él; por lo que todos los esfuerzos para la simplificación, aislamiento y acotación del problema dan al traste con su solución si no se tiene en cuenta el papel significativo que pueden cumplir otros módulos. En parte, éste es el objetivo de este documento.

2. El *origen* de los datos.

Supongamos que en un caso de uso se necesita manejar la información de '*algo*'. Por ejemplo, en PdV, al gestionar una venta se determina que Venta está formada por distintos ítems de los elementos vendidos (Productos o Artículos). En esas Líneas de venta, la información que aporta el actor es la de la selección del Producto y la cantidad que compra (datos variables). El resto es la información del Producto (invariante en el caso de uso), la estrictamente necesaria para la operación que se realiza: el precio (para calcular el total) y la descripción (para que aparezca en el recibo). En el ejemplo, el libro lo denomina EspecificacionDelProducto.

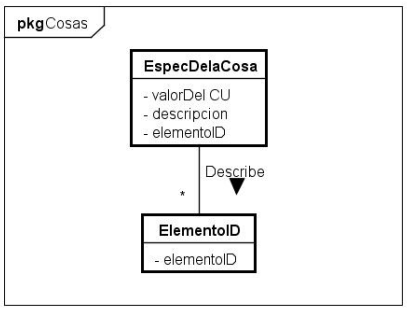
Si la aplicación hace más cosas, será necesaria otra información adicional o distinta sobre el Producto … y de los Proveedores, de los Medios de pago, etc. Toda esa información se sitúa en algún lugar (que denominaremos genéricamente Almacén) que resulte accesible para la aplicación, pero **no *en* ella**.

De igual forma que resulta inadmisible que el actor conozca el código que se maneja internamente, en la aplicación, para seleccionar e identificar un Producto, o que lo copie de un libro, y se interpone una IU para evitarlo e independizar esas acciones de la lógica de la aplicación, también es inadmisible, bien sea globalmente en la aplicación o de manera parcial en cada caso de uso, que se maneje toda la información que reside en el Almacén sobre algo concreto. Las razones ya se han argumentado en documentos anteriores, pero van desde la ineficacia e ineficiencia del funcionamiento, a la ilegibilidad, la rigidez y el alto acoplamiento del diseño, con un comportamiento cuyo mantenimiento o modificación resulta casi imposible. Al igual que se hace con el actor y la IU, en el caso del tráfico de datos, y su gestión, se interpone un *artificio* que lo independiza de la lógica de la aplicación y de su funcionamiento. Veamos en qué consiste ese artificio y cómo se construye.

Tanto si el Almacén (*lugar lógico* donde reside la mayor parte de la información que se maneja en la aplicación) consiste en ficheros, una base de datos, un SGBD o un framework con un servicio de información web, los objetivos son:

- Que el Almacén no se vea obligado a conocer y manejar las estructuras de datos (las clases) de la aplicación o cómo funciona ésta.
- Que la aplicación no se vea obligada a conocer cómo se organiza la información en el Almacén ni cuál es su funcionamiento para prestar sus servicios.

Continuando con el supuesto generalista que inicia esta sección, vamos a llamar *Cosa* a ese '*algo*' del que, por lo que sea, la aplicación o el caso de uso necesita manejar alguna característica. A esa información o datos de la Cosa que se va a utilizar en el caso de uso la llamaremos EspecDelaCosa. Estos datos son fundamentalmente invariantes, los estrictamente necesarios para realizar las operaciones que se definan, y están en el Almacén hasta que no se requieran en el caso de uso. Con invariantes quiero decir que, suponiendo que el Almacén está ya formado y tiene algún dato de Cosa, el contenido de EspecDelaCosa o no se va a transformar completamente o no se va a modificar en absoluto.

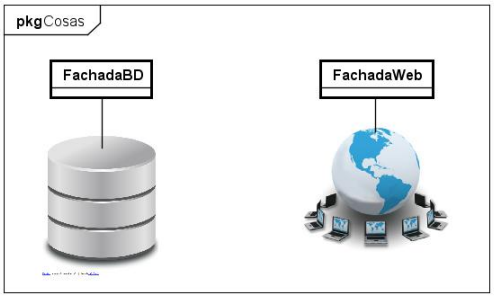


Evidentemente, si Almacén fuera un objeto contenido en la aplicación (con visibilidad global), contenedor, a su vez, de toda la información de Cosa y de la de cualquier otro objeto que se utilice en la aplicación, incluso fuera del caso de uso, la forma de obtener esa información sería:

```
(EspecDelaCosa)almacen.get(ElementoID id)
```

Pero esto significa que la aplicación, el caso de uso y el Almacén comparten todos los objetos entre sí (incluso los que no aparecen ahí). Justo lo que se quiere evitar.

La forma más socorrida para desacoplar es utilizar la *indirección*. Supongamos que Almacén es una base de datos o un SGBD, un sistema necesariamente externo, y vamos a denominar, por ejemplo, FachadaBD a la *puerta* por la intercambia todos sus servicios e información (en OO se denomina *interfaz estándar de comunicación*) con la aplicación. FachadaBD conoce la estructura de la información que se almacena en la BD y el protocolo para acceder a ella.



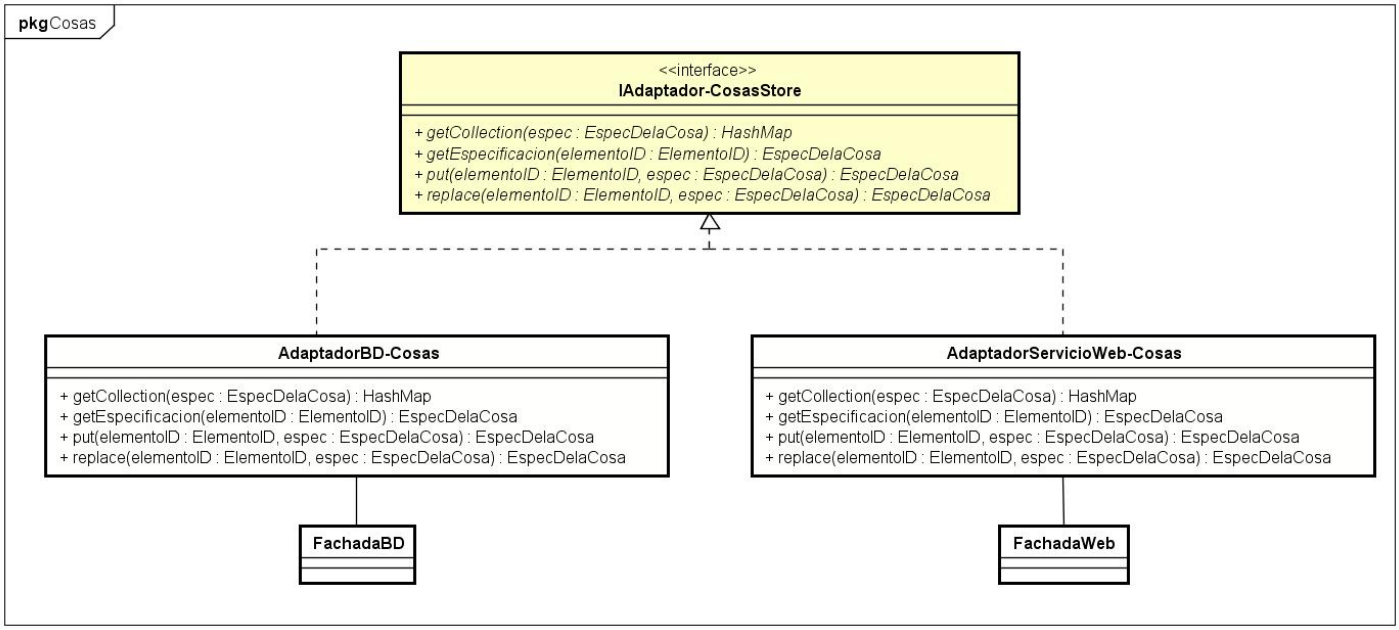
Por la parte de la aplicación se necesita la correspondiente *interfaz estándar de comunicación*, un adaptador, que se encargue de *traducir* la información que se requiere manejar en el caso de uso (es decir, en el paquete o módulo en el que se desarrolla), las clases en las que se estructuran esos datos (EspecDelaCosa), y las acciones de obtención (*get()* o *find()*) o modificación (*set()*, *put()*…), a las estructuras de datos y las acciones que admite FachadaBD para manejar la BD.

Vamos a llamar AdaptadorBD-Cosas a ese adaptador. Nótese que AdaptadorBD-Cosas sólo sirve para manejar la información de EspecDelaCosa con la base de datos. Sin embargo, mediante el par {AdaptadorBD-Cosas—FachadaBD} se ha conseguido desacoplar (aunque

no totalmente) nuestra aplicación respecto a la base de datos. AdaptadorBD-Cosas es de nuestra aplicación (su diseño e implementación es nuestra responsabilidad como desarrolladores de la aplicación) y está especializada para FachadaBD. Por el contrario, FachadaBD viene determinada por la BD, especializada en ella, y está fuera de nuestro ámbito de desarrollo.

¿Y si en lugar de una base de datos se utiliza un servicio de información web? Está claro que habría que hacer otro adaptador especializado para la correspondiente FachadaWeb.: por ejemplo, AdaptadorServicioWeb-Cosas.

Para obtener flexibilidad en el mantenimiento de la aplicación, esta variabilidad en los adaptadores (VP, *Variedades Protegidas*) invoca el polimorfismo; es decir, apela a la utilización de una interfaz cuyas realizaciones, especializadas, son los distintos adaptadores. A esta generalización la llamaremos IAdaptador-CosasStore.



Otro asunto es la visibilidad de IAdaptador-CosasStore, que da acceso a la base de datos. Desde luego, este adaptador está vinculado a EspecDelaCosa y, por ahora, sólo es visible dentro del módulo en el que se desarrolla el caso de uso. En PdV, el paquete Ventas no sólo utiliza IAdaptadorProductos para acceder a la base de datos externa; también IAdaptadorInventario, del paquete de gestión del Inventario, para acceder a esa misma base de datos. Evidentemente, esto aumenta el acoplamiento entre paquetes y distribuir entre ellos el trabajo de asociar el acceso a la base de datos también les resta cohesión. Para evitarlo, vamos a volver a aplicar la estrategia de la *indirección*, pero, esta vez, de otra forma.

Para proveer el servicio de información (es decir, la distribución de interfaces de adaptadores) a través de todos los módulos de la aplicación que lo necesiten, vamos a utilizar una *Factoría* de servicios de acceso a los datos:

FactoriaDeServicios. La manera de evitar el acoplamiento entre sí de los módulos que requieren acceso a los datos externos es centralizar ese servicio en un único objeto que gestione las interfaces y que sea visible, *global*, para todos ellos: un *Singleton*.

Podría ser algo similar a esto:

```
public final class FactoriaDeServicios
{
   private static final FactoriaDeServicios instancia = null;

   private IAdaptador-CosasStore adaptdorCosasAlmac;

   // Un adaptador para cada objeto (contenedor, colección o catálogo) que se necesite
   // obtener del almacén.
   // private IAdaptador-OtrasCosasStore adaptdorOtrasCosasAlmac; (Proveedores, etc.)

               // Vigila la singularidad del Singleton en multithread.
   public static synchronized FactoriaDeServicios getInstancia()
   {
      if (instancia == null) // 1er nivel de check.
      {
          instancia = new FactoriaDeServicios();
      }
      return instancia;
   }

   public IAdaptador-CosasStore getAdaptdorCosasAlmac()
   {
      if (adaptdorCosasAlmac == null)
      {
          // Devuelve la implementación requerida de la interfaz
          String nombreClase = System.getProperty("cosasalmac.class.name");
          adaptdorCosasAlmac = (IAdaptador-CosasStore)Class.forName(nombreClase).newInstance();
      }
      return adaptdorCosasAlmac;
   }

}
```

Hay que aclarar que, aunque se ha iniciado la argumentación en relación con la necesidad de manejar algún dato de Cosa, encapsulado en la estructura EspecDelaCosa, dicho objeto sólo puede acceder y utilizar sus componentes; no a sí mismo. De lo que se está discutiendo aquí es de dónde y cómo se obtiene el contenido de EspecDelaCosa. Por ello, se necesita un contenedor que incluya a EspecDelaCosa como componente y que asuma la responsabilidad de obtener esa información: ContainerEspecDelaCosa con el método `get(ElementoID id) : EspecDelaCosa`.

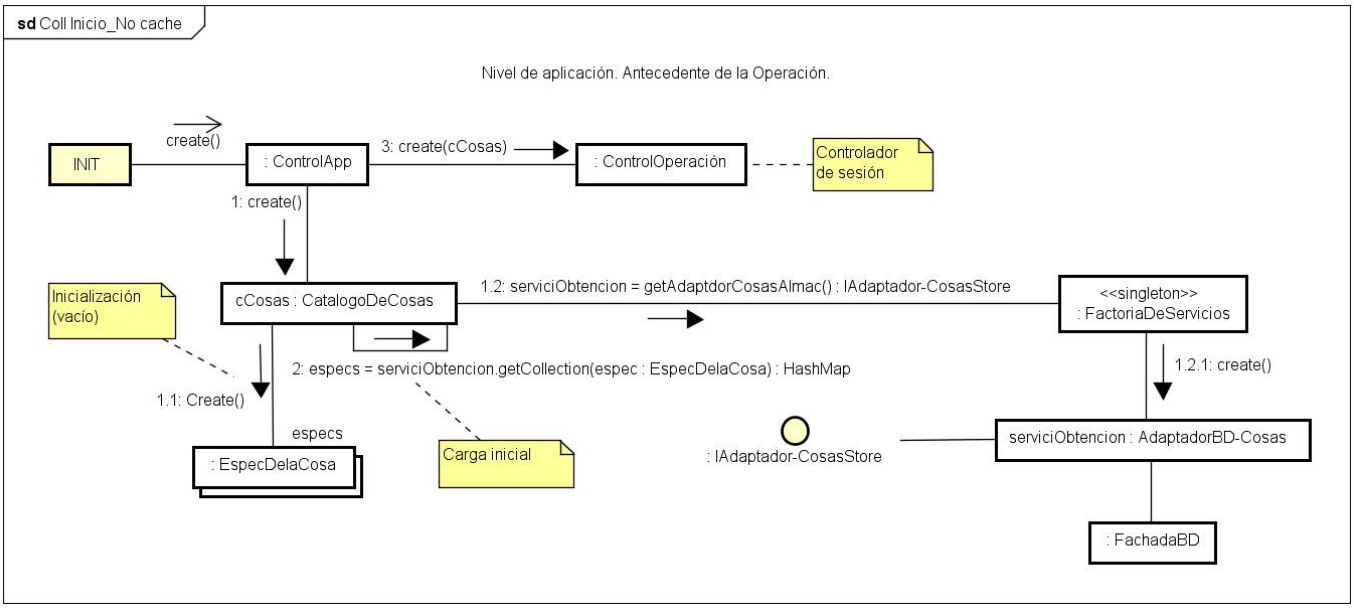
Ahora bien, si en el caso de uso se utilizan varias instancias de EspecDelaCosa, lo que incluye el contenedor es, precisamente, esa colección y los mecanismos para hacer búsquedas, modificaciones, etc. en ella.

Esta situación es absolutamente general y frecuente, especialmente en los exámenes y las PECs, y es lo que el libro denomina **Catálogo**. Se puede denominar como se quiera, Contenedor, Menú, etc. pero lo más importante es entender:

- Que el contenedor alberga una colección y, por tanto, tiene la responsabilidad de implementar los mecanismos para manejarla (búsqueda, modificación, etc.).
- Que lo que caracteriza al contenedor es el tipo de los elementos mapeados en la colección; es decir, la clase o estructura (EspecDelaCosa) que se agrupa en la colección. Para ser más ilustrativos: en PdV el CatalogoDeProductos debería llamarse CatalogoDeEspecificacionesDelProducto. En un estilo de diseño orientado a los datos, como la información fundamental que se maneja en EspecDelProcucto es el precio, se podría denominar CatalogoDePreciosDeProductos. En nuestro ejemplo debería llamarse CatalogoDeEspecDelaCosa.

Es importante tener esto muy en cuenta porque, al igual que los métodos de manejo local, en el catálogo, de los elementos de la colección se aplican a EspecDelaCosa, las realizaciones de esos métodos que se implementan en el adaptador necesitan mapear, exactamente, esa clase.

También significa que, si la información que necesitamos manejar en el caso de uso es sustancialmente diferente (por ejemplo, en lugar de valorDelCU en EspecDelaCosa, utilizamos proveedorDeCosa; lo que puede dar lugar a OtraEspecDelaCosa), **se debe utilizar otro contenedor, otro catálogo**. Queda al criterio del diseñador si la importancia de esa diferencia justifica otra clase como componente de otra colección, otro contenedor diferenciado, el uso de un adaptador distinto o la implementación de los métodos de servicio en el mismo o, incluso, del uso de una interfaz diferenciada.



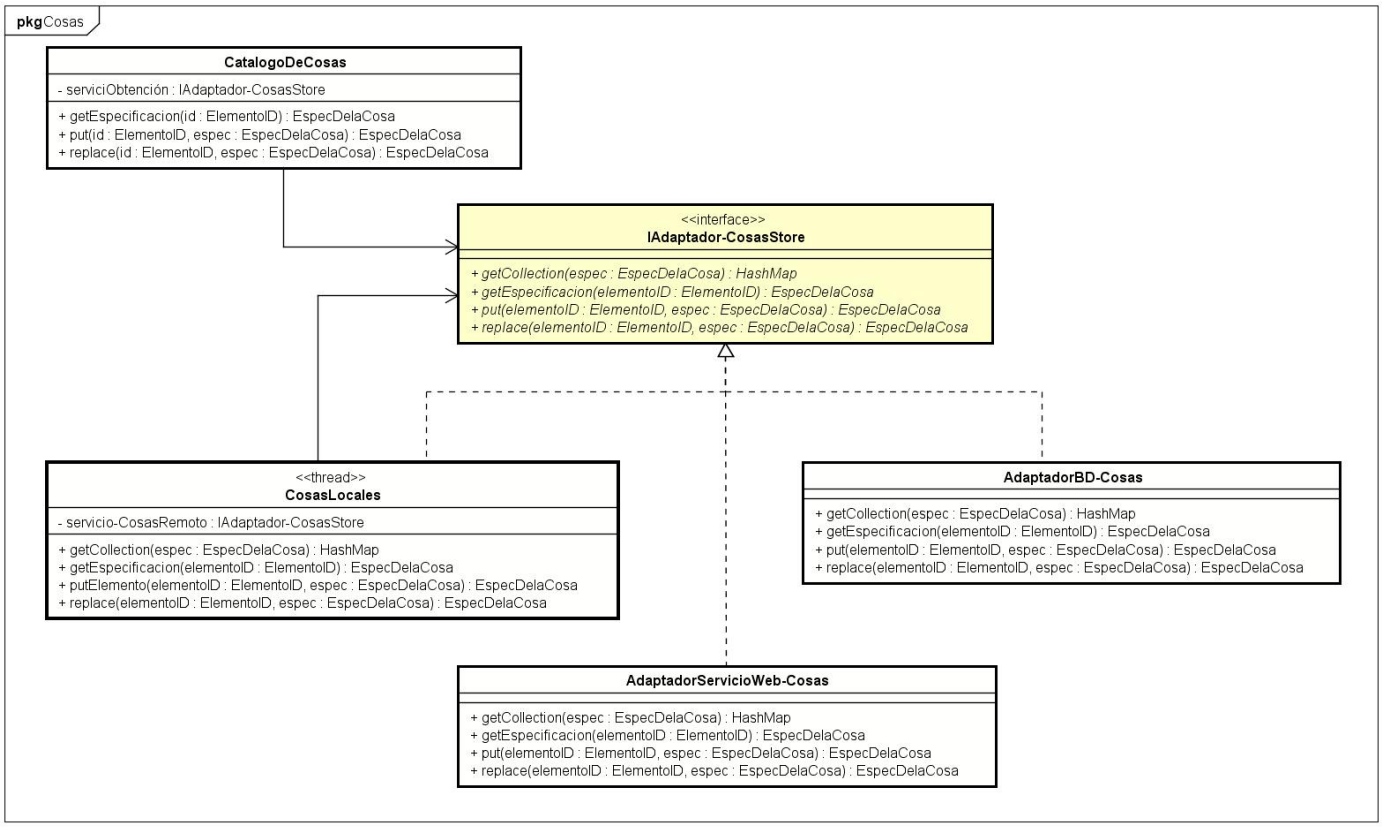
El mecanismo de inicialización y *carga impaciente* del contenedor sería de la siguiente manera:

Una vez diseñados los adaptadores y la factoría de servicios, la inicialización del contenedor cCosas : CatalogoDeCosas se correspondería con el siguiente código para él:

```
public class CatalogoDeCosas
{
   private Map especs = new HashMap(); // La colección vacía.
   private IAdaptador-CosasStore serviciObtencion;
   private EspecDelaCosa laCosa;

   public CatalogoDeCosas()
   {
      serviciObtencion = (IAdaptador-CosasStore)FactoriaDeServicios.getInstancia().getAdaptdorCosasAlmac();
      // Carga impaciente. Se podría haber limitado el tamaño del HashMap en la
      // inicialización de 'especs'.
      especs = (HashMap)serviciObtencion.getCollection(laCosa);
   }

   public EspecDelaCosa getEspecificacion(ElementoID id)
   {
      laCosa = (EspecDelaCosa)especs.get(id); // En 'memoria', no persistente
      if (laCosa == null)
      {
          laCosa = (EspecDelaCosa)seviciObtencion.getEspecificacion(id); // En el adaptador al servicio
                                                                         // externo (realización específica
                                                                         // para la BD o el servicio de
                                                                         // información Web).
          if (laCosa == null)
          {
             return null; // No la encuentra.
          }
          especs.put(id, laCosa); // Carga en memoria (no persistente).
      }
      return (EspecDelaCosa)laCosa;
   }

   // Igualmente, para modificar:
   public EspecDelaCosa putElemento(ElementoID id, EspecDelaCosa espec)
   {
      laCosa = (EspecDelaCosa)especs.put(id, espec); // En 'memoria'
      laCosa = (EspecDelaCosa)seviciObtencion.putElemento(id, espec); // Persistente.
      return (EspecDelaCosa)laCosa; // El valor anterior asociado a id, null si estaba vacío.
   }

}
```

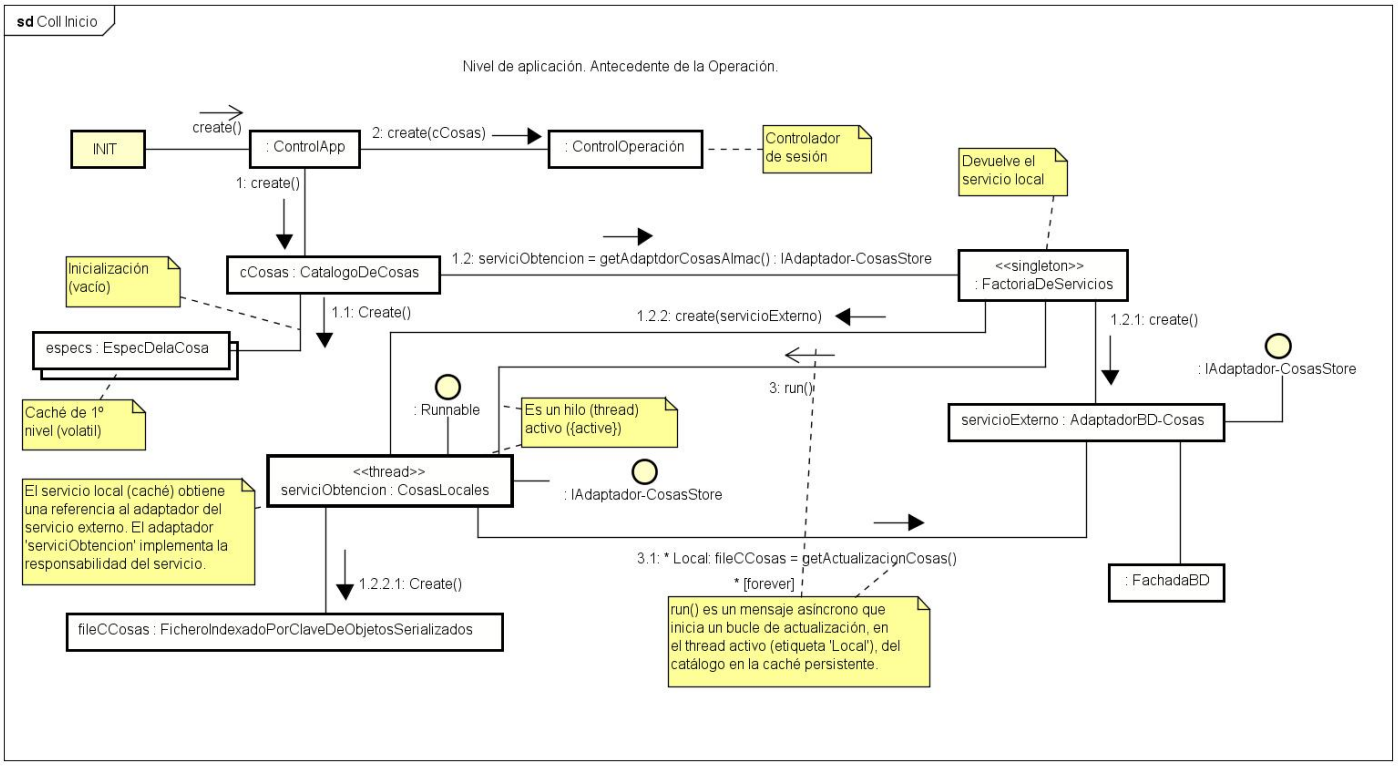
Tras comprender esto, mediante el diseño de un sistema de caché se puede mejorar notablemente el manejo de los elementos de la colección, la gestión de su persistencia. Con él, además, se puede cambiar la *carga* de la colección en el contenedor y hacerla *perezosa*.

El diseño consiste en agregar un adaptador a la interfaz IAdapdador-CosasStore: CosasLocales, que se va a encargar de manejar un almacén local, un fichero con parte de los elementos de la colección: FicheroIndexadoPorClaveDeObjetosSerializados.



En realidad, aunque derive de IAdaptador-CosasStore, CosasLocales no es un verdadero adaptador ni se comporta como una variedad protegida, sino que tiene el rol de *Proxy* de redirección: es a él a quien está direccionado el servicio en el contenedor de la colección, maneja localmente el almacenamiento en el fichero y, si se requiere, envía ese manejo al almacenamiento externo a través del verdadero adaptador que corresponda. Además, se convierte en el hilo (thread) activo del servicio que, aconsejablemente, mantiene un bucle de actualización para los almacenes.

Con la caché, el funcionamiento de la inicialización podría ser así:



3. Conclusiones.

Tras asumir que la labor del diseño consiste en definir y organizar todo el código que comprende los límites de la interacción de la aplicación con el exterior a ella, esta asignatura se centra en el diseño detallado.

Dada la envergadura del trabajo, aún en este ámbito reducido, se establecen fuertes restricciones y simplificaciones en los escenarios de la evaluación:

- El diseño detallado **se restringe a la capa del dominio de la lógica de la aplicación**. Desde el punto de vista arquitectónico, esta capa está formada por los paquetes o módulos por cuyo comportamiento, organización y colaboración se define la funcionalidad y el comportamiento global de la aplicación. A partir de cierta complejidad, que puede ser baja, es muy posible que el funcionamiento conjunto de esos paquetes requiera una coordinación, un control del funcionamiento global y su flujo (*workflow*). Es lo que se denomina nivel de aplicación, que forma parte de la lógica del negocio, *envuelve* a los módulos de esta capa, proyecta su influencia hacia la capa de presentación y, aunque no suele tener muchas responsabilidades, está más acoplado con ellos.
- La labor **se limita al diseño de un solo caso de uso**. Con frecuencia, la simplificación llega a que el caso de uso no sea primario o ni siquiera esté completo, sino que el escenario se plantea en un tramo de éxito intermedio o parcial hasta su final. A pesar de esta restricción, es

frecuente que en el transcurso de su funcionamiento intervenga el código de varios módulos.

- Ya que nos centramos en el diseño detallado, no se va a dedicar mayor esfuerzo en agrupar y organizar el código en paquetes, subsistemas o módulos (arquitectura). Sin embargo, acentuado por la simplificación en el caso de uso, la elaboración de su lógica, su funcionalidad y su código puede estar fuertemente condicionada por **el funcionamiento de otros módulos o los artefactos antecedentes** de otras operaciones (en especial en el nivel de la aplicación); por lo que **sí es necesario tenerlos en cuenta**.

> Con estas consideraciones, más las de la naturaleza y características de la OO, más los principios GRASP, la tarea para las respuestas de la parte central de las evaluaciones consiste, como mínimo, en construir los objetos (clases) cuyos métodos sean capaces de realizar la funcionalidad pedida.

Después de las recomendaciones, propuestas en años anteriores, para realizar estas tareas, más orientadas al diseño por descomposición funcional descendente, las de este año están más próximas al diseño dirigido por los datos:

1. Tras la lectura rápida del planteamiento general del sistema software, en el enunciado, léase con detenimiento el de la pregunta 2, que es donde se definen los límites y el contenido del caso de uso, el objetivo de todo el ejercicio. Si esto no está claro, es difícil que tenga éxito en los resultados. Con una idea bien formada sobre los límites del sistema, sus interacciones con el exterior y la situación del caso de uso del ejercicio, podrá contestar la pregunta 1.

2. Es fundamental que tenga presente la descripción del caso de uso. En la pregunta 2, debe descomponer el caso de uso descrito en una secuencia de *pasos* representados desde la perspectiva lógica de la interacción {estímulo (del actor) <–> reacción (del sistema)}. Tenga en cuenta que el actor tiene un objetivo y que la lógica de los estímulos que produzca está determinada por las opciones que le plantee el sistema para conseguirlo. En cuanto a la descripción de la reacción del sistema en cada paso, está enfocada a esa misma lógica de la interacción, por lo que representa la respuesta o la nueva situación de cara al actor y su siguiente estímulo, sin especificar detalles técnicos de lo que pase dentro del sistema. Por tanto, sobran los términos como *lee la tarjeta*, *registra*, *consulta la base de datos*, *crea* o *modifica algo*, etc. Si es caso, esas operaciones se verán reflejadas en el modelo de dominio. E, igualmente, por parte del actor: sobran los términos *introduce la tarjeta*, *pulsa la opción*, etc.

3. La elaboración del modelo de dominio (pregunta 3) aún forma parte del análisis y es una de las preguntas donde se detectan más dificultades. Se trata de *traducir* las reacciones del sistema, descritas en cada paso de la

pregunta anterior, a objetos que tengan capacidad para realizarlas ante los estímulos del actor. Se trata de recomponer la lógica del funcionamiento del sistema (descrita por esos pasos) mediante unos objetos que la realicen. ¿Si no se ponen métodos, cómo se sabe si un objeto puede *hacer algo*? La respuesta es: Por los datos que manejan, por sus atributos, otros objetos componentes y las relaciones (no las de uso ni las de dependencia) que se establecen entre ellos y sí se representan (con la cardinalidad de la relación) en el diagrama.

La recomendación para trabajar orientado por los datos que se manejan podría ser aplicada ya aquí, y está estrechamente relacionada con el objetivo principal de este documento y todo lo expuesto en el punto 2.

A la hora de hacer el modelo de dominio, la propuesta consiste en comenzar buscando un objeto que parezca ser el destinatario de la operación fundamental o el objetivo del trabajo del caso de uso. Por ejemplo, en PdV, la Venta, en EnDP, el PlanDeEntrenamiento, en BankBox, la Transferencia o la Transacción, etc.

A partir de ese candidato (llamémosle OObjetivo), y sin prejuzgar aún cuáles son sus atributos, el análisis continuaría evaluando qué acciones u operaciones requiere hacer el sistema (la lógica descrita en los pasos de la pregunta 2) para cumplir con el objetivo funcional del caso de uso, y qué objetos son candidatos para hacer cada operación, componentes de OObjetivo o no.

Aunque aquí, en el análisis y la elaboración del modelo de dominio, se realice de una manera incipiente, la tarea fundamental de todo el ejercicio consiste en fabricar estructuras de datos (objetos y clases) que realicen operaciones y que contengan los datos necesarios para hacerlas. Por ello, la estrategia más importante para hacer lo que se indica en el párrafo anterior es pensar, simultáneamente, en 2 cosas:

- ¿Qué operación va a hacer un objeto?
- ¿Qué datos necesita el objeto para hacerla?

El resultado típico es encontrar algún/os componente/s de OObjetivo, que realizan operaciones intermedias y que contienen los datos necesarios para hacerlas. A su vez, si algún componente no dispone de alguno de esos datos, habrá que construir otro objeto cuya responsabilidad sea la de aportárselo; y así sucesivamente.

Falta añadir un objeto que coordine la secuencia de esas operaciones y *cierre*, así, la representación de la lógica del caso de uso: un Controlador. Su función para finalizar la lógica del CU no sólo viene dada porque es el único que interacciona directamente con el actor (y se ve determinado por la secuencia del diálogo {estímulo—respuesta}), sino porque también debe tener la iniciativa y dirigir, con sus respuestas, los estímulos del actor; además de coordinar la actuación de los otros objetos. Un verdadero *director de orquesta*.

Un actor humano rara vez aporta un dato que se utilice, directamente, en las operaciones del caso de uso (un ejemplo de esa *rareza* podría ser la cantidad del artículo comprado o el número de cuenta en la transacción de BankBox). Y mucho menos, al interponerse la capa de presentación, proveerá un objeto que se utilice en el código del caso de uso. Aunque en la elaboración del modelo de dominio no hay que preocuparse por la visibilidad de los objetos (se resolverá en el diseño) sí hay que tener en cuenta que, por lo general, la aportación del actor consiste en valores, indicadores o datos que se utilizan, indirectamente, en el software del caso de uso, para obtener el dato o el objeto que sí se requiere para la operación correspondiente. Y esto afecta notablemente en la elaboración del modelo del dominio y en el modelo de diseño.

Como se ha insistido en el punto 2 de este documento, casi toda la información y los datos que se utilizan en una aplicación están en un Almacén **externo a ella** (es un sistema o un actor de apoyo a la aplicación). Toda la argumentación y los desarrollos de diseño expuestos en ese 2º punto tienen como objetivo mostrar que toda esa información es accesible y está disponible para su utilización en el caso de uso, sin la necesidad de hacer la barbaridad de *importar* el Almacén.

Ninguna de las clases que se han desarrollado allí (punto 2 de este documento), para justificarlo, deben aparecer en el modelo de dominio (puesto son artificios software) y ni siquiera en el diseño, en las respuestas de las evaluaciones, es necesario que se reproduzcan esos mecanismos ni dichas clases. Se pueden obviar, otra simplificación más.

Sin embargo, sí es importantísimo tener en cuenta este hecho: la accesibilidad y disponibilidad de esa información se consigue gracias a una identificación (código o clave para su búsqueda y para su modificación) y a una estructura que dote a dicha información de una semántica de uso.

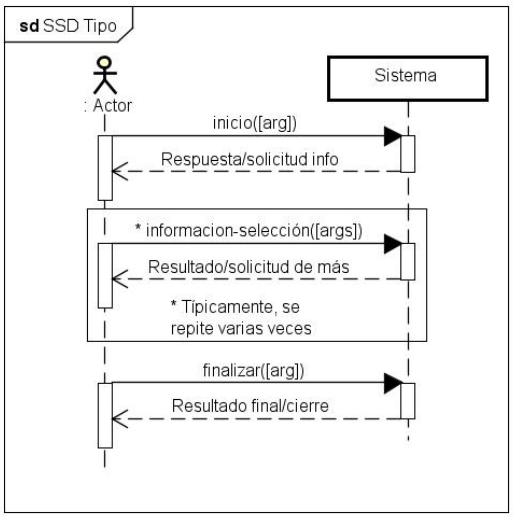
Por consiguiente, **sí es responsabilidad ineludible** del analista/diseñador **la construcción** (con su estructura y su funcionalidad) **de aquellos objetos destinados a contener y manejar**, según la lógica del funcionamiento del caso de uso, **determinados datos e información que**, aunque se ubique en alguna zona de almacenamiento externa, **se utilizan necesariamente en él**.

Si en el caso de uso procesarVenta, de PdV, se determina que objeto destinatario de la operación es la Venta, que sus constituyentes fundamentales son la relación de Artículos vendidos (seleccionados por el Cliente) y el importe Total de la venta, cuyo valor se obtiene a partir del precio de cada Producto, es obligado construir un objeto que contenga ese precio (EspecDelProducto; que, además, contendrá la clave de identificación y una descripción, para el recibo) y un contenedor, para la colección de esos objetos y con capacidad para gestionar la selección de los Productos concretos realizada por el Cliente: el CatálogoDeProductos. Se pueden usar otros nombres, pero no hay otra

forma más simple, precisa y legible de representar esa lógica funcional para este caso de uso.

4. **Diseño**. El éxito para aclarar la lógica del funcionamiento del caso de uso determina notablemente la realización del diseño. Son los objetos encontrados en la elaboración del modelo de dominio, y su capacidad para describir esa lógica, los primeros candidatos para construir su realización en el código.

   Además de *producir* código, el diseño consiste en dotar su ejecución con una secuencia temporal. Como una aproximación inicial a esa secuencia, se recomienda realizar un diagrama de secuencia del sistema (DSS): una representación temporal del diálogo, por paso de mensajes, entre el actor y el software del caso de uso, representado como un único objeto:



   La utilidad de este diagrama es identificar las operaciones más importantes del caso de uso, agrupar los mensajes, si es preciso, y ordenarlos en la línea temporal.

   Dada la escasez de tiempo en el examen, quizás sea ésta una de las recomendaciones de las que primero se pueda prescindir; pero sigue siendo aconsejable hacerlo, aunque sea con un boceto.

   Este diagrama no es el que se pide en las respuestas de la evaluación, sino el diagrama de secuencia en el que se '*desgrana*' este diálogo entre los distintos objetos que intervienen en el caso de uso (aquí son ya instancias de las clases software).

4.1. En el diagrama DS que se pide, el diálogo con el actor se establece únicamente con la clase con el rol de Controlador. Cada mensaje o estímulo que le envía el actor, lo transforma en acciones internas, en mensajes a otras clases o en la creación de nuevas instancias. Y así, sucesivamente, en cada clase, hasta llegar a los límites exteriores del sistema: los adaptadores.

   Recuerde, el envío de un mensaje a una clase significa (excepto, claro está, en el caso del actor, que tiene la IU de por medio):

   - Que la clase receptora tiene ese método.
   - Que el tipo de los argumentos que llegan en el mensaje es conocido por la clase receptora (es un atributo o un componente suyo, '*visibilidad de argumento*'). Igualmente ocurre, si se da el caso, con el tipo del valor que devuelve el método.
   - Recíprocamente, la clase emisora tiene a la destinataria como componente. Y lo mismo ocurre con los tipos de los argumentos o con el del valor que devuelve.

Es decir, cada mensaje establece una asociación (y un acoplamiento) entre las clases.

Es posible que, en esta descomposición de los mensajes, se aprecie que falta algún atributo en un objeto o que se debería asignar a algún otro. Si las modificaciones son mayores, si repercuten en la lógica del funcionamiento, será necesario cambiar el diagrama del modelo de dominio. Pero, si está medianamente bien hecho, las nuevas clases que construyamos en este diagrama DS serán artificios software, de '*fabricación pura*', cuya utilidad es la de implementar la lógica y el comportamiento definido en el modelo de dominio y no requieren cambiarlo.

En este punto, la incorporación del tiempo, del instante en el que ocurren las cosas, que caracteriza al diseño, desvía la dirección de la elaboración basada en los datos hacia la basada en las funciones; y sólo se percibe en cómo se manejan éstas y sus argumentos.

Otra cuestión muy importante en la elaboración de esta pregunta 4 y las 2 siguientes, es cuándo, por qué y cómo se instancian las clases. Es cierto que hay directrices GRASP para hacerlo, igual que hay otros 2 o 3 de esos principios que ya deberían estar aplicándose desde la elaboración del modelo de dominio. Pero quiero llamar otra vez la atención sobre algo que puede pasar desapercibido, quizás porque el libro sólo insiste en ello en los capítulos finales: la interacción con los otros módulos de la arquitectura y, especialmente, con el *nivel de aplicación*.

Un caso de uso, *recortado* o no, suele tener unos antecedentes cuyo control se realiza desde la lógica de la aplicación. También es frecuente que el caso de uso utilice algún artefacto (objeto) producido en otro módulo como resultado de uno de esos antecedentes. Mientras los paquetes o módulos de la descomposición arquitectónica se comunican a través de su interfase estándar (lo que les mantiene relativamente independientes) el control de la aplicación se dedica a coordinar los casos de uso en todos los módulos, sus antecedentes y sus resultados, y está más acoplado con todos ellos. La misma Factoría *Singleton* que, a través de una interfaz de adaptadores (IAdaptadorProductos), proporciona al paquete Productos independencia respecto a los servicios técnicos de acceso a los datos, se la concede al paquete Inventario, a Contabilidad o a CalculadorDeImpuestos (mediante sus específicas interfaces de adaptadores). Sin embargo, antes de que el actor opte por la realización de una de las operaciones de un grupo, es frecuente que el control de la aplicación ya tenga que tener dispuesto un artefacto que comparten todas ellas. Por ejemplo, el CatálogoDeProductos en las Ventas de PdV.

El modelo de dominio es una fotografía *estática* de la lógica del funcionamiento y no tiene sentido considerar qué objeto existía antes del caso de uso y cuál se crea en él. Aparte de las clases instanciadas en la inicialización de la aplicación, hay que tener muy presente qué clases se utilizan en el caso de uso, pero ya están instanciadas por ser

el producto de una operación anterior, y cuáles crea el controlador, o las sucesivas clases, en el transcurso del funcionamiento de dicho caso de uso.

Un ejemplo absolutamente frecuente en las evaluaciones es la abreviación del caso de uso, para simplificar aún más las respuestas, y establecer que, tanto la inicialización como el acceso (del actor e incluso a la operación en sí), ya se han realizado.

En la inicialización de la aplicación, es común que se instancien las clases con visibilidad global (singleton a nivel de aplicación, por ejemplo) y los adaptadores de uso más generalizado entre los módulos. ¡Ojo! Porque, llevados por los nombres de algunas clases, podemos caer en la tentación de sacarlas de su contexto y pensar que son de uso global en la aplicación; cuando se contrapone con el '*minimalismo*' que estamos utilizando, respecto a su contenido, para evitar el acoplamiento y aumentar la cohesión. En PdV, el CatalogoDeProductos (que hemos quedado en que debería llamarse CatalogoDePreciosDeProductos) del paquete Ventas, no puede ser el mismo que el que se utiliza en Alquileres, ni sirve en Inventario ni en Contabilidad.

En cada nivel del desarrollo de la lógica de la aplicación, el controlador activo creará las instancias que necesite (aquellas comunes que se vayan a utilizar en el siguiente grupo de operaciones) y de las que se tenga la suficiente información para hacerlo (*Experto en Información*).

Otro ejemplo claro es el del acceso del actor humano. Justo antes del acceso, el control de la aplicación debe *cargar* el contenedor de la colección (catálogo) de los usuarios, con sus credenciales de identificación y su rol; no necesita nada más para gestionar el acceso. Una vez que accede, el mismo control pondrá a su disposición las operaciones que le estén permitidas según su rol. Y así, en cada paso, con las clases que se requieran. Nótese que, aunque lo que se suele denominar *perfil de usuario* puede contener mucha información de él, no se carga en la aplicación todo el perfil cada vez que se necesita algo concreto, ni tampoco una vez '*por si se necesita más adelante*'. En cada momento, lo necesario.

4.2. En esta pregunta, la escritura de los contratos de 2 operaciones principales del caso de uso se utiliza para aclarar qué pasa antes y qué pasa después de producirse cada una. Ya se ha indicado la necesidad e importancia de utilizar un lenguaje próximo al formal para dicha escritura. Pero, sin duda también por la escasez de tiempo, existe la tentación inevitable de seleccionar, como operación *principal*, una llamada a una función que tiene escaso, o apenas ningún, alcance en el caso de uso. Esto sólo va en detrimento de la calificación obtenida y tiene aún mayor repercusión negativa puesto que, en los diagramas de colaboración de las 2 preguntas siguientes, se describe el funcionamiento de esa misma selección. Si no hay nada, o hay muy poco, no se puede calificar.

5. Se considera que la labor de diseño más intensa, teniendo en cuenta el enfoque de la asignatura y lo que se espera en las evaluaciones, se produce en los diagramas de interacción (secuencia y colaboración). Por tanto, todo lo que se ha explicado para el DS de la pregunta 4.1 es válido para las preguntas 5 y 6; con la salvedad de que es importante situar la creación de los objetos en el punto preciso de la línea de tiempos, para el DS, mientras que en los de colaboración lo importante para reflejar el tiempo es numerar el orden de los mensajes.

Como conclusiones para este documento:

1. La asignatura se centra en el diseño detallado. Sin embargo, para realizarlo es imprescindible tener en cuenta el diseño arquitectónico, puesto que es indivisible de aquel.
2. El trabajo de las respuestas, en las evaluaciones, se corresponde con el diseño detallado de un caso de uso, definido en la pregunta 2 de los ejercicios y circunscrito, en la arquitectura global del escenario descrito, al dominio de la lógica del negocio.
3. La labor de las preguntas específicas de diseño (4 y siguientes) consiste en crear estructuras de datos (clases software) capaces de colaborar entre sí (mediante sus métodos) para realizar la lógica y el funcionamiento descrito para el caso de uso.
4. Casi la totalidad de la información que manejan esas clases (datos, atributos, etc.) reside en un sistema de información externo al ámbito del trabajo de diseño. Su acceso, obtención y modificación está garantizado mediante mecanismos software que se explican en la asignatura y que, en principio, no hay que implementar en la evaluación. El trabajo en los ejercicios se reduce, por consiguiente, a determinar qué información es necesaria, cuándo se utiliza, a qué estructura de datos (clase) se asigna para que pueda ser manejada y cómo lo hace (sus métodos).