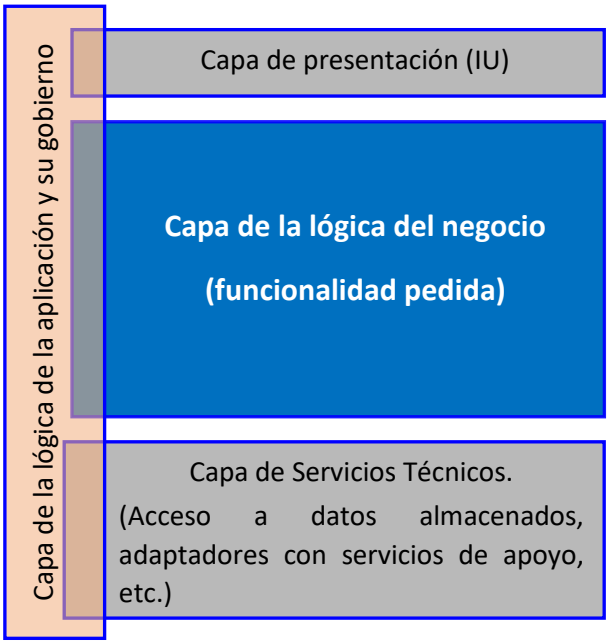
Por diversas razones, para aproximarse a las indicaciones en la realización de la PEC 2019, se proponen las siguientes consideraciones:

1. Suponer que la aplicación cuya funcionalidad vamos a implementar se puede plantear, de manera muy genérica, según el siguiente estilo arquitectónico, el fundamento del conocido Modelo—Vista—Controlador (MVC):



   a) La capa de servicios técnicos proporciona el soporte, de nivel medio-bajo, necesario para el funcionamiento de la capa que implementa la lógica del negocio. Por ejemplo, independiza las estructuras de datos utilizadas 'ad hoc' en la lógica del negocio del lugar en el que estén almacenados y la forma en la que se gestione el almacenamiento. Esto se implementa, en esta capa, mediante una interfaz abstracta (polimórfica) y los adaptadores concretos que se encargan de intercambiar datos con cada fachada del sistema de almacén que se utilice. Así, el código del dominio del negocio queda relevado de tener que implementar las consultas a una base de datos, sea relacional u orientada a objetos, a un fichero de objetos serializables o a un servicio de información web: ese código se implementa en los adaptadores. La manera de transmitir estos servicios al dominio del negocio es a través de una factoría abstracta de servicios, un singleton (o su equivalente: un TAD encapsulado) con visibilidad global, que se sitúa en la capa de la lógica de la aplicación y proporciona, exactamente, la realización del adaptador requerido. Este tipo de diseño permite, además, construir aquí un sistema de persistencia de la información que se maneja en el dominio del negocio, independientemente de esa capa.

      En esta capa también se proveen otros servicios cuya ubicación, como la del anterior acceso a los datos almacenados, permite el desacoplo con el código del dominio del negocio: son servicios de acceso a sistemas de información o de apoyo externos. Por ejemplo, un servicio de autorización de transacciones comerciales (pagos a crédito). En el caso de la PEC, es razonable prever que el sistema se quiera ampliar con una red de sensores

que automaticen la toma de medidas en los embalses. Significaría un incremento en la funcionalidad de la aplicación que habría que implementar en el dominio del negocio pero, gracias a que el acceso a las medidas de los sensores (y su origen) se construye en el adaptador correspondiente de esta capa de servicios técnicos, el código de la capa de la lógica del negocio puede tratar esas medidas de igual manera a como lo hace con una entrada manual: la estructura de datos de la medida, TAD, clase, componente o como se quiera llamar, es la misma.

b) La capa de presentación es '*Thin Layer*', es decir, apenas tiene responsabilidades sobre la funcionalidad que se quiere implementar. Básicamente, sólo formatea la información de entrada – salida. Esto permite independizar la implementación de la lógica del negocio respecto a la del tipo de interfaz que se quiera utilizar para su uso: texto alfanumérico (como en el enunciado original), gráfico con ventanas, web, etc. Al igual que hace la capa de servicios técnicos en su ámbito, en esta capa se utilizan las estructuras de datos, específicas y privadas, que resulten más útiles para la labor de presentación que realiza. De la misma manera, también, el flujo de información con la capa del dominio del negocio está desacoplado mediante adaptadores (otra vez interfaces polimórficas), que se encargan de mapear el contenido de las estructuras que se manejan en un lado, en las estructuras del otro. Así, si se cambia el sistema de presentación (IU), sólo se debería trabajar en la implementación del adaptador que traslada la lógica de la interacción que requiere la funcionalidad de la aplicación (en el dominio de negocio) a su presentación ante los actores.

c) La capa de la lógica de la aplicación y su gobierno controla el flujo de trabajo de la aplicación, el acceso a las funciones que provee y la secuencia en que se produce. Por hacer de *puente* entre las otras capas, es la más acoplada a ellas. Pero, si se cuida que cada capa implemente, estrictamente, lo que es de su responsabilidad, el acoplamiento es mínimo. La manera en la que se construye el gobierno en la lógica del flujo de trabajo de la aplicación suele ser 'por delegación' y según un esquema arbóreo de jerarquía: en el nivel 'raíz', el controlador de la aplicación proporciona los recursos necesarios (frecuentemente con visibilidad global) para la función seleccionada (que suele ser un caso de uso primario) y delega el control en ella (en su controlador correspondiente), pasando al siguiente nivel del funcionamiento (y control) que se desenvuelve, ya, en el ámbito del dominio del negocio.

d) La capa de la lógica del negocio. Es en la que se propone realizar todo el trabajo de la PEC, una vez eliminadas las interferencias de las otras 2 capas y la rigidez que supone tener que implementar, '*ad hoc*', la interfaz de usuario y el acceso a los datos en algún almacén. En esta capa se reciben los estímulos (también llamados 'eventos al sistema') que provienen de los actores; como si la capa de la IU fuera transparente y el actor invocara, directamente, una función y le pasara sus argumentos.

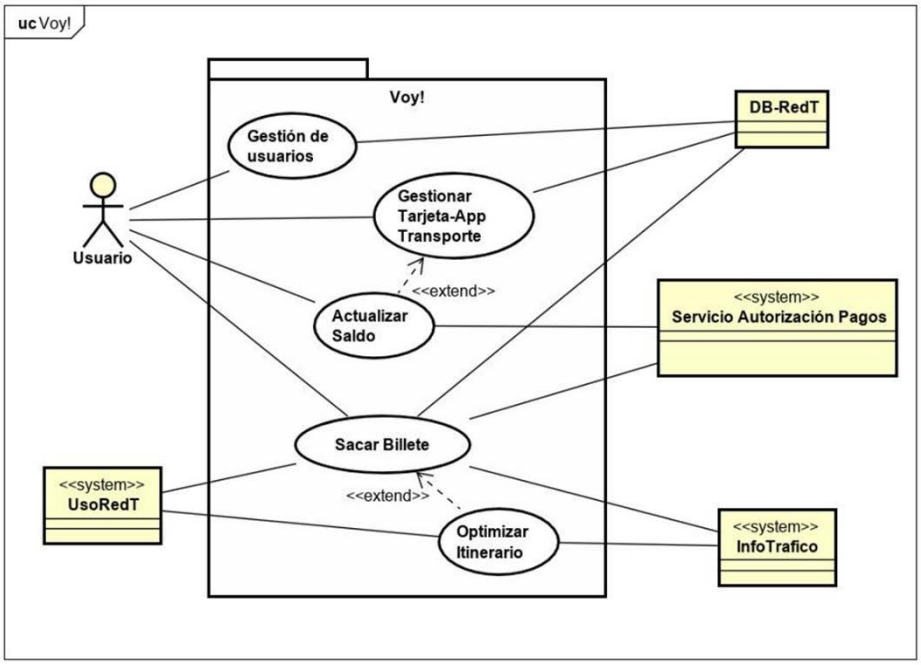
   Es en esta capa donde se elaborará el análisis, el diseño y la codificación de la PEC. Se propone hacer un análisis con casos de uso (generalizando, yo lo reduciría a 3 CU principales: "Gestión de recursos hídricos",

"Adquisición de medidas" y "Consultas"), tras lo que se prosigue con el diseño y la codificación inicial de cada uno. Tras avanzar en el diseño detallado de cada CU, ya se puede aventurar una agrupación de las clases, en módulos, que dará como resultado la arquitectura. Es decir, el código que va a residir en esta capa, también se puede agrupar y organizar en módulos que deberían ser funcionalmente independientes (o el máximo posible), dando lugar al diseño arquitectónico. Esta arquitectura (que es la que se va a elaborar) no tiene nada que ver con la división en capas que se ha hecho en la aplicación.

2. Metodología de desarrollo próxima a Rational Unified Process (RUP), de tipo iterativo (una adaptación del ciclo de vida en espiral), situando el desarrollo en la fase denominada 'elaboración' y limitado a un ciclo o iteración. En esta fase se seleccionan los casos de uso que permiten definir la arquitectura base del sistema y se van a desarrollar en ella, se realiza la especificación de los casos de uso seleccionados y el primer análisis del dominio del problema, se diseña la solución preliminar y se codifica para probar y medir resultados.

   Los artefactos que se utilizan en las distintas etapas y sus actividades son:

   a. Diagrama de Casos de Uso. Es de la fase anterior a la elaboración, la del 'inicio', pero la utilizamos aquí porque permite situar la funcionalidad principal de la aplicación, en el dominio del negocio, con relación a los actores principales, que mantienen la interacción con ella, y a los actores o servicios de apoyo, también externos, que aportan soporte de información u otro tipo de servicios complementarios a esa funcionalidad.



En ninguna de estas actividades, hasta que no se entre en el diseño, existe el código. Es decir, hasta que no se comience el diseño, no hay clases software, ni abstracciones, ni herencia, ni interfaces, ni polimorfismo, ni nada parecido; y no deben representarse en los diagramas. Se utilizan entidades, objetos conceptuales que, aunque puedan ser candidatos para convertirse en elementos del software (clases), ahora sólo son los

contenedores del manejo de la información en el uso que se hace de la aplicación.

Un caso de uso es un relato que describe la utilización de la aplicación para conseguir un fin o para realizar una operación. La entidad que usa la aplicación, que interacciona con ella porque tiene la voluntad de conseguir ese objetivo, es el actor principal, generalmente, humano. El caso de uso toma el nombre de la operación u objetivo que quiere alcanzar el actor principal (por ejemplo, 'Procesar una Venta'). Para que un caso de uso sea principal o primario, debe responder a un *Elementary Business Process* (EBP). Es decir: debe responder a un objetivo fundamental del uso de la aplicación. Por ejemplo, la autorización y acceso de un usuario a la aplicación no es un CU primario, porque ningún usuario tiene como objetivo final, del uso de la aplicación, entrar sólo en ella.

Es muy frecuente encontrarse con operaciones que se realizan sobre algún elemento de información que pertenece a una colección. Esas operaciones se denominan CRUD (de las siglas de Create, Recovery, Update, Delete) y se agrupan, como subfunciones, en el caso de uso 'Gestión de…' (por ejemplo, 'GestionDeRecursosHidricos').

Con estas reglas, es relativamente sencillo determinar la funcionalidad principal del sistema que se va a construir.

b. Una vez seleccionados los casos de uso primarios que se van a elaborar, se inicia **el análisis** y se prosigue todo el desarrollo con cada uno, en paralelo a los demás o uno después de otro, es indiferente. Un caso de uso primario tiene principio y fin en el uso de la aplicación. El primer artefacto del análisis es la escritura del caso de uso. La escritura de un caso de consiste en el relato, ordenado, de la secuencia de interacciones, que se producen entre el actor principal y el sistema (estímulo del actor – respuesta del sistema), descrito en el flujo normal de los acontecimientos al usarlo, con éxito, en ese CU. Se dispone de un ejemplo en la documentación del grupo de tutoría. Sólo se describe la acción que realiza el actor (en términos *cotidianos*, del ámbito del negocio) y la reacción que se obtiene desde el sistema (también descrita en los términos que se manejan en el ámbito del negocio). En este sentido, es importante el *estilo esencial*; que significa excluir de la descripción «TODOS LOS DETALLES ESPECÍFICOS DE LA TECNOLOGÍA UTILIZADA EN LA ACCIÓN O DE SU FUNCIONAMIENTO INTERNO». Por ejemplo, se escribe "Se identifica el artículo" en lugar de "Se lee el código de barras del artículo" o "Se incorpora la medida" en lugar de "Se escribe la variación de volumen, en hm^3, con 5 dígitos enteros y 3 decimales". También se recomienda el formato '*a 2 columnas*' (actor ←→ sistema) para hacer más visual la interacción estímulo – respuesta.

A continuación de ese flujo principal de éxito, se escriben, en el mismo formato, los flujos alternativos: situaciones variantes que significan una bifurcación en la secuencia normal. Por ejemplo, "7bis. El Cliente paga con tarjeta de crédito" o "5bis. La medida excede el rango permitido". En estas variantes, se indica el punto de bifurcación y se escribe la secuencia para tratar situación, hasta el final o hasta su reincorporación al flujo principal.

La utilidad de este artefacto es obtener la comprensión de qué pasos hay que dar en el caso de uso (desde el punto de vista del uso que se hace de la aplicación, del negocio) y, sobre todo, qué elementos de información se están manejando en el CU.

c. El tercer elemento es el diagrama del Modelo de Dominio. En UML no existe un diagrama con esta denominación. Es decir, este lenguaje no se superpone a este tipo de modelado conceptual del comportamiento deseado para la aplicación que, por otro lado, es la representación más importante que se busca en el análisis. En un caso general, hay muchas formas de llegar a esa comprensión del comportamiento deseado para sistema, y en el libro hay un número suficiente de diagramas que representan estos modelos, cada uno con su perspectiva. En el caso de modelado mediante objetos, su característica fundamental (que es la que se emplea para construir estos modelos de dominio) es que *encierran* una información que es la única que pueden manejar. Esto también es característico de cualquier dato abstracto (TAD) y, como sólo se manejan elementos conceptuales en este nivel de representación del análisis (al margen de la existencia o cualquier referencia al software: herencia, polimorfismo, etc.), valdrían igualmente.

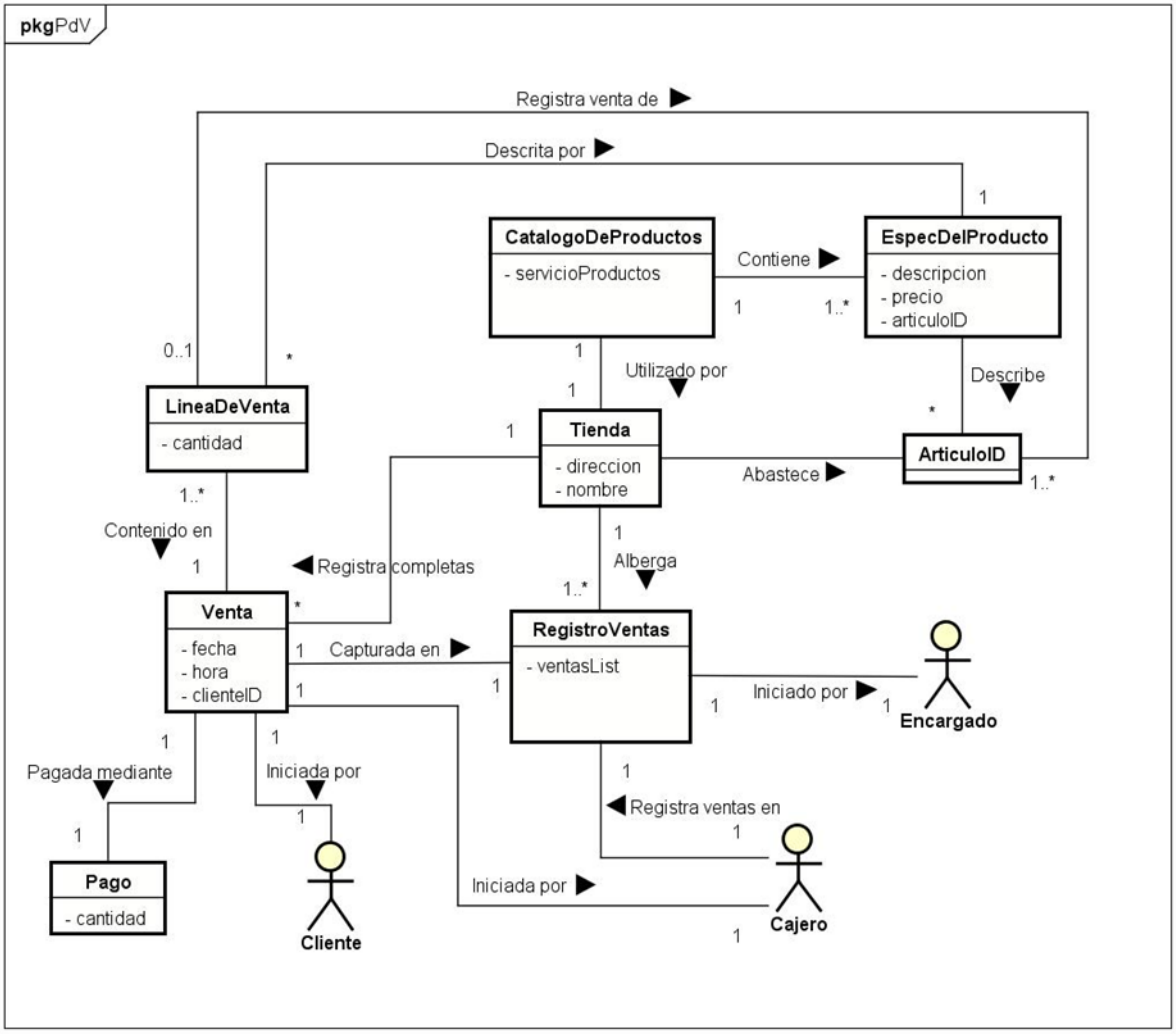
   En UML, el modelo de dominio se elabora con un diagrama de clases (pero ni son clases software, ni es diseño). Las 'clases' representan *objetos conceptuales*, en los que no aparece la sección de los métodos o funciones que pudieran realizar. Así, el elemento fundamental para construir el diagrama del modelo de dominio son los objetos conceptuales: la información (o agrupación de datos) que está representando **lo que se podría hacer** con ella. Quizás sea éste el paso más delicado en la construcción de los modelos de dominio: identificar unos objetos conceptuales cuyo contenido da idea de lo que hacen, de cuál es su responsabilidad (porque sólo pueden hacerlo con ese contenido).

   ¿Cómo se traducen las acciones que hemos identificado en la escritura del CU a los objetos que podrían realizarlas? Mediante el método de Abbott se pueden buscar algunos candidatos (ver libro): puesto que se buscan *entidades de información* (estructuras de datos, objetos *conceptuales*) que sean el objetivo del manejo que se deriva de las operaciones identificadas en la escritura del caso de uso, la recomendación es seguir esos objetivos. Por ejemplo, en una venta (una transacción –intercambio— comercial) el objetivo de las operaciones es la Venta y los elementos que la constituyen (cada producto, su precio, etc.).

   Aunque la estrategia de '*seguir el* objeto *objetivo del CU*' es la línea fundamental para construir el modelo de dominio, evidentemente, es necesario tener en cuenta un buen número de aspectos adicionales:

   - UML recomienda que, si un atributo o componente es de un tipo derivado (no primitivo, no es un número, un *string*, etc.), se represente fuera del objeto y relacionado con él. En el caso del modelo de dominio (los objetos no son software), esto se debería interpretar en relación con que la justificación para representar un

componente fuera de su contenedor, más que si su tipo es primitivo o no, es si va a ser objeto de alguna acción o manipulación.



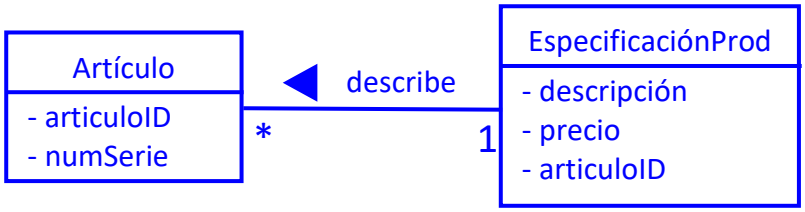
Por ejemplo, en el caso de Venta, en PdV, cada ítem vendido es una cantidad determinada de algún Artículo:

Pero, en la Venta, lo que se maneja de cada ítem vendido es:

Artículo
- descripción - precio - articuloID - numSerie

- El tipo de artículo que es, o el producto: 'articuloID', un código que identifica el producto.
- Su precio, el mismo para todas las instancias de ese producto. Aunque pueda ser una cantidad, el tipo de este atributo se puede generalizar e, incluso más que *currency*, se puede abstraer en un tipo *Dinero*.
- Su descripción, usada para que el producto sea *reconocible*. Un *string*.
- 'numSerie' es para ilustrar que puede haber varios artículos del mismo tipo de producto (con los mismos valores en las demás características).

Por tanto, para manejar cada ítem vendido, en Venta, es mejor organizarlo así:



- Otra situación muy común es que el objeto que contiene la información que se necesita manejar es una opción entre varias posibilidades. En nuestro ejemplo, la reflexión sobre el objetivo, la Venta, ha llevado a la conclusión de que es un intercambio de artículos por un valor acordado: su precio, una cantidad de Dinero. El seguimiento de la información que debería contener Venta, para alcanzar el objetivo de realizar la transacción, aparte de la fecha y hora en que se produce, lleva a que es una agregación de diversas cantidades de artículos (*LineaDeVenta*).

  En realidad, parte del éxito en el caso de uso consiste en poder calcular el valor del precio total en la transacción: la suma de los precios de cada artículo que, en diversas cantidades, se está vendiendo. Es decir, la característica fundamental que se necesita conocer de cada elemento, en esa agregación de ítems vendidos, es su precio.

  En esta misma línea deductiva de '*seguir la información*', en el punto anterior se ha concluido que el precio de un artículo no es una característica de cada elemento, sino del tipo de producto que lo representa. Es decir, para obtener el precio del ítem vendido hay que saber de qué tipo de producto se trata o, lo que es lo mismo, es necesario realizar una selección entre todos los tipos de producto que hay a la venta. Esta conclusión lleva a la necesidad de agrupar las características que necesitamos manejar de cada artículo (el precio en 'EspecificaciónProd'), para todos los artículos que se puedan vender, en una colección. Para manejar de forma eficiente los elementos de esa colección (búsqueda, selección, agregar o quitar elementos, etc.), se incluye como componente principal de un contenedor. En un restaurante, los precios de los platos se organizarían en una 'Carta de Platos y sus precios'. En general, a ese contenedor se le denomina '**Catálogo**' y, tanto él como la estructura de la colección que incluye y la de cada uno de sus elementos (EspecificaciónProd), son objetos esenciales para seleccionar y obtener los datos que se utilizan en el caso de uso.

  ¿De dónde vienen esos datos del CatalogoDeProductos y el contenido de EspecDelProducto? Excepto las decisiones del actor principal o sus aportaciones de datos variables, toda, absolutamente toda la información está almacenada o proviene de un sistema de apoyo externo. Precisamente, por partir de la consideración de esa organización en capas del punto 1, es totalmente irrelevante su procedencia; mucho menos en el análisis y sin considerar la

existencia del código. En el caso del contenido del catálogo, proviene de algún sistema de información, base de datos, fichero o similar, que maneja estos datos según su conveniencia; seguramente con una denominación y un formato completamente distinto al que se va a usar en la implementación del caso de uso. Entre su procedencia y el uso que se le da en la capa de la lógica del negocio, esa información se transfiere a través de la capa de servicios técnicos y mediante implementaciones software de procedimientos de los servicios de acceso a datos (algunos bastante sofisticados, ver documento 'Ejemplos de Diseño y Arquitectura'). Por consiguiente, al construir el modelo de dominio, la única responsabilidad del diseñador es determinar:

a. Qué datos necesita para cumplir el objetivo del caso de uso.
b. Cómo organizar esos datos en estructuras (objetos, que sólo acceden a sus atributos) que permitan su uso según las pautas y la secuencia de la escritura del caso de uso.

Ahora bien, si la relevancia de la procedencia de los datos del catálogo ha sido sustituida por la necesidad de crear las estructuras (objetos) que permitan su uso, surge la importante pregunta de quién maneja el catálogo. La respuesta tiene mucho que ver con la secuencia en la que se realizan las operaciones, con la dinámica del funcionamiento y, por lo tanto, con el diseño. Pero, en el diagrama del modelo de dominio, aunque pertenezca al análisis y deba estar exento de cualquier referencia a su implementación y al código, también es necesario reflejar, parcialmente, esa dinámica.

Este aspecto se construye, otra vez, mediante el principio de que «*cualquier objeto sólo puede manejar su contenido*». Por ello, en este diagrama, casi todas las relaciones indican que un objeto es un componente de otro (y su cardinalidad); indicando así que el contenedor maneja su contenido.

¿De quién es el catálogo? Todo depende de dónde y para qué se use. Esta es la cuestión fundamental para la elaboración del diagrama del modelo de dominio, del diseño y su implementación final; ya que la única manera de conseguir independencia funcional y acoplamiento bajo es seleccionar, con un cuidado exquisito, la mínima cantidad de información que sea útil para la operación que se está realizando en cada momento. Nótese que en el catálogo de precios de los productos en venta la magnitud fundamental que se maneja en los elementos de la colección es el precio. El catálogo para el inventario será otro, porque lo que manejan los elementos de su colección es el número de unidades en stock de cada producto, en el de devoluciones los elementos la colección serán las ventas realizadas, etc.; aunque toda la información esté en la misma base de datos o almacén.

Esto quiere decir que se requiere una lógica de funcionamiento que actúe en un nivel superior al del caso de uso; que, al seleccionar una operación determinada (el caso de uso), realice la consulta precisa al almacén y construya la colección, exacta, que va a

necesitar. Esta gestión se realiza en la capa de la lógica de la aplicación (el equivalente al *main* de la PEC) y, en el modelo de dominio del ejemplo anterior, está representado por la 'Tienda'.

Es decir, al centrar todo el trabajo en la capa de la lógica del negocio de un caso de uso concreto, sí es cierto que sólo hay que organizar los datos en estructuras útiles (con el mínimo contenido) para que se puedan realizar las operaciones del caso de uso; pero no hay que ignorar que hay otra lógica de funcionamiento, supra-CU, perfectamente sincronizada con él y que comparte el manejo de alguna de esas estructuras (acoplamiento: el Catálogo y el Registro), que, por formar parte de su interacción, hay que representarlo en el modelo de dominio del caso de uso y, posteriormente, diseñar en detalle e implementar en código esa funcionalidad. Al fin y al cabo, el resultado pasa por codificar absolutamente todo, hasta el límite de las fachadas con los sistemas externos (mediante los adaptadores, que hay que codificar también). La buena noticia es que, al trabajar de esta manera, ya se está organizando la arquitectura de toda la aplicación.

- Otro de los objetivos de este caso de uso es '*dejar constancia de la operación y de sus datos relevantes*'. Está claro que la relevancia de los datos de una Venta depende de qué se quiera hacer con ellos y para qué se vayan a usar. Pero, en este caso, ese mismo criterio es el que ha llevado a decidir qué debe contener el objeto Venta. Un concepto parecido sería el de un *logger*, que registra toda la actividad (*log*) en un sistema. En general (sobre todo en transacciones), el objeto cuya responsabilidad es registrar la operación se denomina **Registro**. La manera en la que cumple con esa responsabilidad se representa en el diagrama con la relación '*captura*'; que puede parecer que describe una acción, aunque ninguna de las relaciones que se presentan en un diagrama de dominio son operaciones o métodos de los objetos. El significado de esa relación es, simplemente, que la Venta es un componente de RegistroVentas.

- El último grupo de elementos de este diagrama es el que se refiere a la organización de la interacción y de la secuencia de las operaciones.

  Puesto que el caso de uso es una descripción de una utilización concreta de la aplicación, en el modelo de dominio debe aparecer **el actor/es principales** que *conducen* esa interacción (generan eventos, o estímulos, a la aplicación; **nunca son objetos**); pero no los subsistemas software, de apoyo, ya que su interacción se resuelve mediante adaptadores, artefactos software que se implementarán en el diseño pero que no están implicados en la representación conceptual del comportamiento deseado, en el uso de la aplicación, del modelo de dominio.

  Si este diagrama contiene actores (únicos elementos que generan acciones), estructuras de datos con contenidos estáticos y relaciones de composición ¿cómo se representa la organización de

las acciones? Para hacerlo, al menos parcialmente, se necesita representar un rol del '*director de orquesta*', un objeto que, aunque su representación siga siendo estática, su responsabilidad sea la de organizar la secuencia de las operaciones que constituyen la funcionalidad del caso de uso. Este rol se suele representar mediante el término '***Controlador***' o '*Manejador*' del caso de uso.

Como se ha indicado al hablar de la estructura de la lógica funcional en la capa de la lógica de la aplicación, el controlador es el elemento fundamental en una jerarquía arbórea que se desarrolla en profundidad mediante llamada—retorno (delegación). En cualquier nivel, su controlador supervisa las acciones que se deriven de cualquier evento y, si se requiere descender al siguiente nivel, se elaboran los datos que se vayan a necesitar en él y se delega en el controlador correspondiente, que atiende a sus responsabilidades y, al finalizar, devuelve el control al invocante.

Evidentemente, esta dinámica se resuelve en el diseño. En el modelo de dominio el rol de controlador (su papel dinámico) se representa como en los demás objetos: mediante sus componentes y lo que potencialmente puede hacer con ellos. En el ejemplo de Procesar una Venta, en PdV, la responsabilidad de organizar las operaciones de una Venta debería residir en el objeto que tuviera a Venta como componente: el RegistroVentas. En este caso, a ese objeto se le asignan 2 responsabilidades: la de dejar constancia de los datos significativos de la operación (en su rol como registro) y la de organizar las acciones para realizarla (en su rol como controlador).

Por su parte, la Tienda maneja el catálogo del caso de uso (posiblemente también utilizado en otros) tiene al RegistroVentas (el controlador del caso de uso) como componente: es el controlador de nivel superior, de la lógica de la aplicación, que supervisa la funcionalidad del sistema para acceder a su servicio de ventas.

Aunque el diagrama del modelo de dominio es una representación puramente estática (el contenido de los objetos), ya se ha visto que también describe la *intención* y la potencialidad de acción de sus objetos; algo fundamental para construir el diseño detallado (la implementación de la dinámica de su funcionamiento mediante el código). El rol del controlador se sitúa en esta naturaleza dinámica, de supervisión de la evolución temporal en el funcionamiento, y, sin embargo, es necesario representarlo en el modelo de dominio; aunque no revele abiertamente el papel que juega. Sin él no está completo el mapa de los objetos conceptuales (no software, aún) que intervienen en el uso de la aplicación para este CU. Que el control supervisor (la Tienda) construya o maneje el Catálogo de Productos, cree el Registro de Ventas (el controlador del caso de uso), le pase el Catálogo como argumento y le ceda el control, todo ello tras haber solicitado a la aplicación el servicio de asistencia en las ventas, es algo que pertenece a la descripción del diseño; pero,

también, es una consecuencia de la comprensión del funcionamiento del sistema, fruto de la reflexión sobre su uso y sobre las necesidades que ello requiere.

Por último, en esa dimensión del diseño, hay que tener en cuenta que el rol de recepcionista de los eventos, de interlocutor en la interacción entre el actor principal y el sistema software, sólo lo puede ejercer un controlador a la vez. Si se cambia de controlador, el original transfiere toda la gestión de la interacción con el actor principal al controlador en el que delega; y sólo él acepta eventos y envía sus reacciones al actor.

d. Dentro de esta 1ª iteración, en la fase de elaboración del ciclo de vida que se está siguiendo (similar a RUP), en una transición entre el análisis y el **diseño detallado**, es muy recomendable realizar un Diagrama de Secuencia del Sistema (DSS) para cada caso de uso. Los diagramas de secuencia, junto a los de colaboración, pertenecen a una categoría denominada *representación de la interacción*. La característica principal de ambos diagramas es que representan el funcionamiento detallado del sistema software mediante el paso de mensajes entre instancias (llamadas a funciones específicas –métodos—, con sus argumentos) de las correspondientes clases software, y bajo una estricta referencia temporal. Por tanto, desde el punto de vista de la descripción del funcionamiento del código (la especificación del diseño detallado), tienen el máximo nivel de detalle y resultan bastante más descriptivos que los Diagramas de Clases del Diseño (DCD). Tanto es así que, prácticamente, son el código de cada caso de uso.

   Los diagramas de secuencia (DS) son representaciones gráficas de *desarrollo vertical*, a lo largo de la línea temporal de cada objeto. Aunque éste no es el ámbito para entrar en los detalles de la sintaxis de estos diagramas, sus elementos principales son:

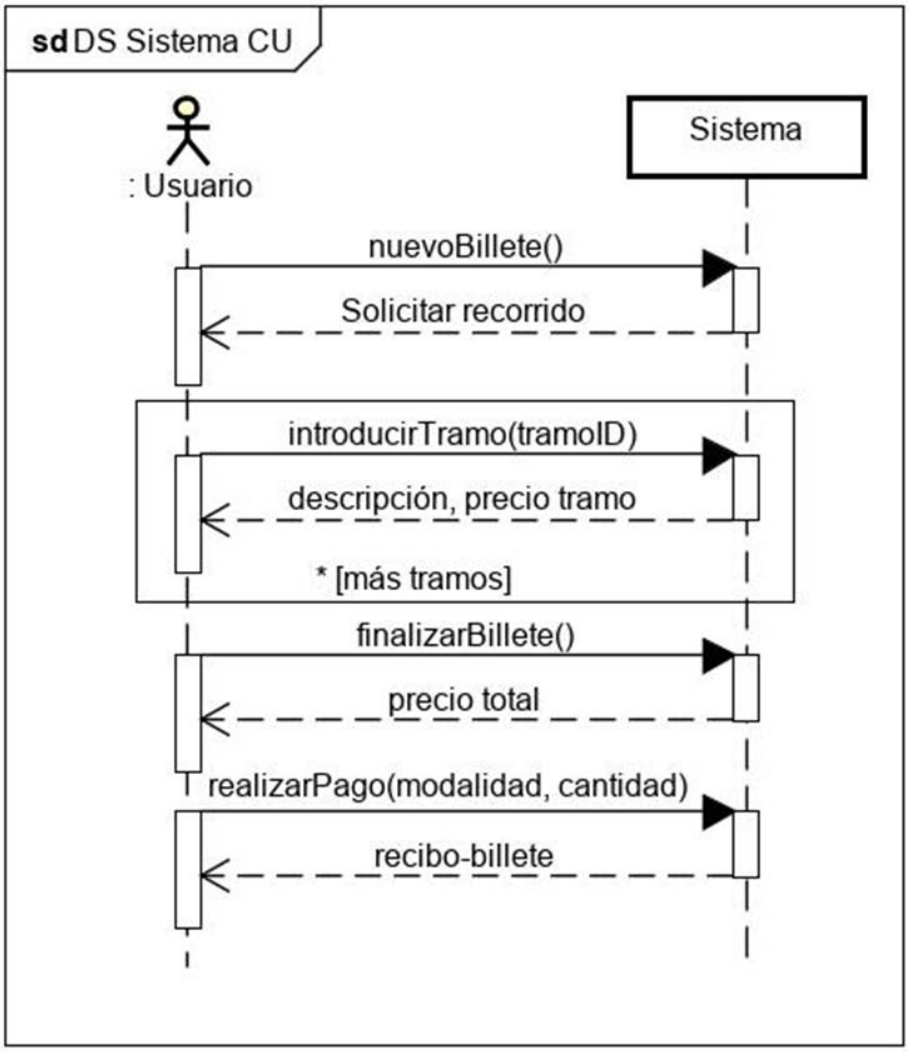
   - Los objetos que intervienen en el funcionamiento: '`instancia : Clase`', inscrito en una caja o rectángulo. Si no se indica el nombre de la instancia, se suele poner la clase subrayada; indicando que está instanciada: '`: Clase`'.
   - La línea temporal de cada objeto: una línea vertical discontinua anclada en la base de la caja del objeto. Representa la existencia de la instancia del objeto, durante la cual puede aceptar mensajes (excepto el de creación, `<<create>>`) y realizar acciones.
   - Los mensajes. Se representan mediante una línea continua horizontal, que finaliza en punta de flecha, cerrada, y un texto con el siguiente formato:

     ```
     método (arg1, arg2, …)
     ```

     El origen del mensaje (base de la flecha horizontal) se sitúa en un punto determinado de la línea temporal del objeto que invoca el método del objeto destinatario, indicando el instante preciso en el que se produce el evento y estableciendo con exactitud cómo se produce la secuencia de acciones del funcionamiento. A partir del punto en el que produce el mensaje, en ambas líneas temporales, se

representa un rectángulo o bloque que indica la duración del procesamiento del mensaje. En la línea temporal del origen del mensaje, esa duración debe ser mayor que la suma de la del destinatario y la de los subsiguientes mensajes, a otros objetos, que este procesamiento suscite.

En el caso del **diagrama de secuencia del sistema** (DSS) únicamente se representa la interacción entre el actor principal y el sistema, **representado como un único objeto**. Esto tiene la ventaja de que permite descomponer el caso de uso en una secuencia de operaciones y determinar cuáles son las principales.



Como se ve en el ejemplo para el caso de uso Sacar Billete, de una aplicación para la gestión de transportes urbanos (Voy!, ver estos documentos: planteamiento y presentación), se representan los eventos o estímulos del actor como invocaciones directas a métodos de alguna clase en la aplicación. Obviamente, cuando se detalle el diagrama, esa clase será el controlador del caso de uso; único que acepta los mensajes del actor. Esto es una simplificación: un actor humano no procesa el código directamente, actúa sobre la IU (capa de presentación) que se encarga de transformar el formato la acción en una llamada al método que

corresponda, con sus respectivos argumentos. Lo mismo ocurre en el otro sentido (mensajes de respuesta, con línea discontinua y flecha abierta): el actor no es capaz de procesar variables ni funciones y lo que recibe, tras su paso por la IU, es información significativa para él. La IU actúa como una capa *transparente*, interpuesta en medio de la interacción, y en el diagrama sólo se representa lo que reciben los destinatarios de los mensajes, a un lado y a otro.

Para expresar bucles de iteración, se encierran los mensajes de una de ellas en un rectángulo y se marcan mediante una etiqueta con un '*'.

Este diagrama no puede ser de diseño porque no permite construir el código de las clases que actúan en el funcionamiento del CU (excepto, quizás, parte del controlador); pero ya utiliza invocaciones a métodos de ese código, con argumentos que son objetos software, o sus componentes, y permite establecer una secuencia temporal clara del procesamiento de esos mensajes (que, ahora, se llaman operaciones).

e. El siguiente paso es entrar en el diseño detallado mediante el **diagrama de secuencia detallado** (**DS**). Es como el anterior (DSS) pero el objeto 'Sistema' se descompone en todas las instancias de los objetos software que lo constituyen y con las que se implementa el funcionamiento del caso de uso.

   Profundizando en la sintaxis de los mensajes, el texto tiene el siguiente formato:

   ```
   [guard] returnVar = método(arg1, arg2, …)
   ```

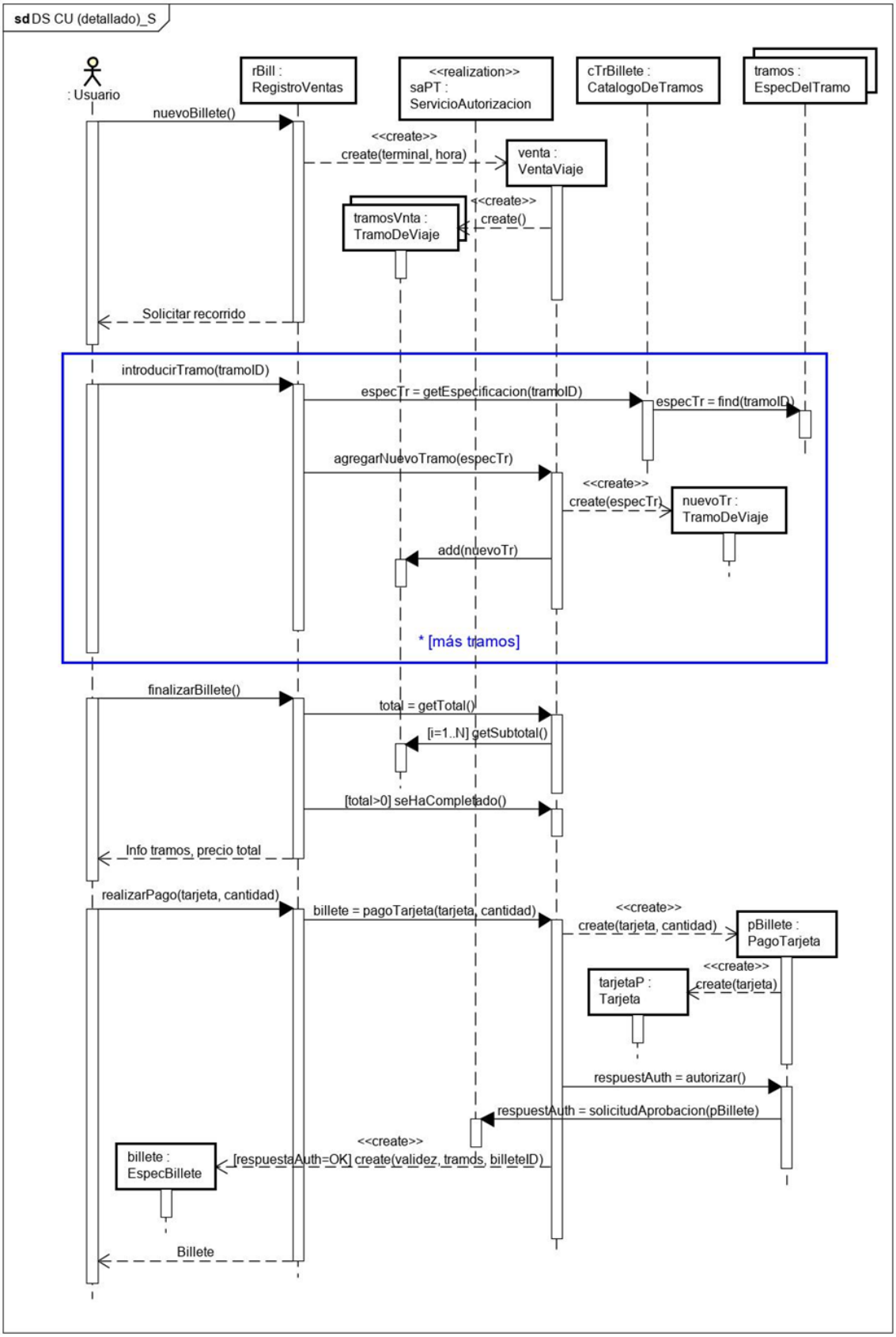
   - `[guard]` es una condición (opcional) cuyo cumplimiento determina el envío del mensaje.
   - `returnVar` es el nombre del objeto que devuelve el mensaje (opcional).
   - No se suele indicar el tipo de ningún elemento del mensaje.

   El mensaje `<<create()>>` es uno especial, se representa con una línea discontinua horizontal y la punta de flecha abierta, por el que se invoca el constructor del objeto destinatario. Significa que esa instancia (y su línea temporal) inicia su existencia justo en ese instante de la línea temporal del objeto creador y para el resto de los objetos que ya existen.

   El origen del mensaje (base de la flecha horizontal) se sitúa en un punto determinado de la línea temporal del objeto que invoca el método del objeto destinatario. Eso quiere decir:

   a. El objeto destinatario debe ser un componente del objeto invocante. De forma análoga debe verificarse el control de tipos en la visibilidad de los argumentos del mensaje y en el objeto o variable del retorno.
   b. La acción invocada en el mensaje debe ser un método existente en objeto destinatario.

c. La acción se desarrolla en ese instante, ni antes que otra acción situada más arriba en esa línea temporal, ni antes de que exista la instancia destinataria del mensaje, ni después que otra acción posterior de la línea temporal.



Como se ha indicado, la elaboración de este diagrama se realiza según el orden temporal de los mensajes, desde el primero que inicia el CU y que debe recibir (como todos los emitidos por el actor) el controlador. En cuanto a qué objetos se ponen a la derecha, en sustitución del 'Sistema', obviamente, hay que mantener la coherencia con el modelo de dominio y utilizar los candidatos que aparecen allí; convertidos en objetos software.

Por ser anterior, y estar fuera del caso de uso, en el diagrama no aparece cómo el controlador de nivel superior crea u obtiene el Catálogo de Tramos (cTrBillete), crea el Registro de Ventas (rBill, controlador del CU) y le transfiere cTrBillete como argumento. Por ese motivo, rBill, cTrBillete y los elementos de la colección que contiene (tramos:EspecDelTramo) ya existen al comenzar el CU.

Al comenzar a ejercitarse en esta forma de desarrollo, es muy común darse cuenta de algún defecto en los elementos presentados en el diagrama del modelo de dominio. Si sólo afecta a las clases que provienen de ese diagrama, es necesario corregirlo allí porque es importante mantener la consistencia en todo el desarrollo. Sin embargo, también es muy frecuente que la implementación del funcionamiento del software requiera la intermediación de clases fabricadas expresamente para ello (fabricación pura) y que, al no corresponderse con objetos conceptuales, del uso del negocio, no pueden aparecer en el modelo de dominio. Es el caso de los adaptadores, artefactos puramente software que se utilizan para obtener servicios o datos desde ámbitos externos a la aplicación, que hay que incluir en el diagrama, y cuya visibilidad normalmente es global en el caso de uso (por eso, también existen desde su inicio).

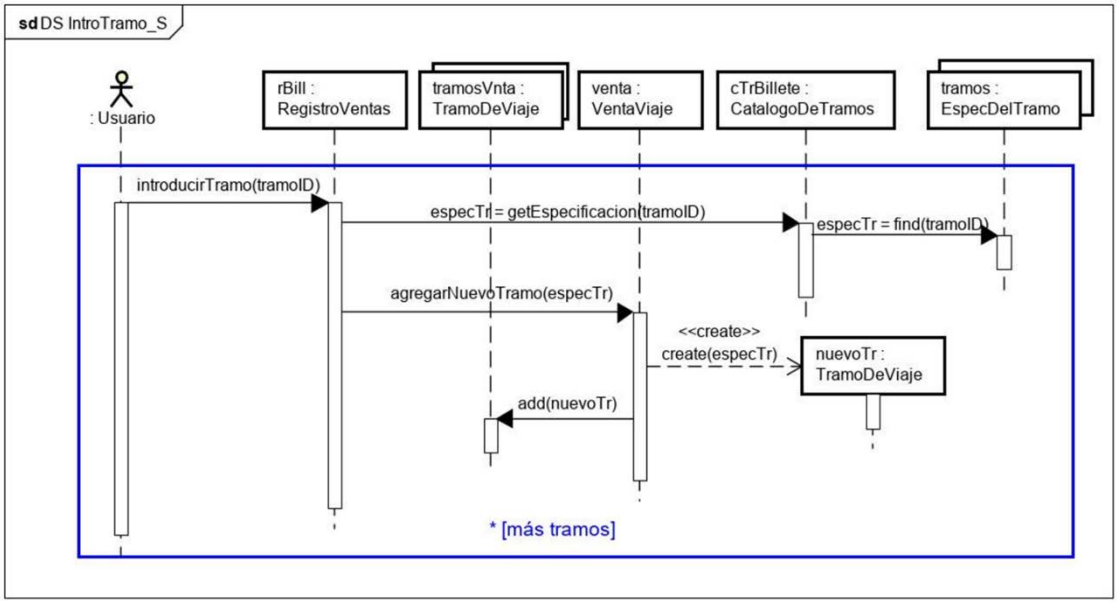
El resto consiste en desgranar, por orden y paso a paso, cada una de las acciones que se han determinado para que el caso de uso cumpla su objetivo; pero cada acción debe realizarse **en** el objeto que tiene los datos que son necesarios para hacerla. Se puede encontrar una guía para hacerlo en este esquema, con unos principios generales para asignar responsabilidades (funciones y métodos) a cada clase (los principios GRASP). En este documento se han estado manejando los más importantes (Experto, Controlador, Acoplamiento Bajo) desde el análisis.

Por otro lado, también es importante tener en cuenta cómo se representan determinados mecanismos, propios del funcionamiento del código:

- Se admite la invocación de métodos predefinidos en determinados objetos: *setters* y *getters* (por supuesto, aunque es preferible evitarlos), *add* de un elemento en una List, *put* o *find*, para una colección HashMap, etc.
- Si se van a agregar elementos a una colección (derivada de List o de Collection –Map o HashMap—), primero se crea la colección vacía (un multiobjeto, que se representa con un doble recuadro descentrado). Después, para agregar el elemento, se hace así:
    1. Se crea un elemento (instancia) vacío y se inicializa con su contenido (mediante su constructor, opción preferible), o bien
    2. Tras la creación, se le asigna su contenido (*setters*).
    3. Esa instancia se añade a la colección (*add()* o *put()*).

- Es preferible expresar la respuesta a un mensaje como la variable o la instancia de retorno del método, y no con otro mensaje de respuesta (flecha y texto del mensaje). Es decir, así: `especTr = getEspecificacion(tramoID)`. Sin embargo, en la creación de instancias no hay que poner el nombre de la instancia como valor de retorno: tras la invocación de `create(tarjeta, cantidad)`, el elemento creado, e inicializado, `pBillete:PagoTarjeta`, ya es una instancia componente de `venta:VentaViaje` (el creador).

Es muy frecuente que se obtenga un mapa de funcionamiento muy extenso con el diagrama anterior (que sólo es para un caso de uso). En esos casos, es posible fragmentarlo en las secuencias correspondientes a cada evento de la interacción del actor (cada *operación*). Si se hace así, hay que tener especial cuidado en situar correctamente las clases que ya existen en cada segmento y excluir las que no intervienen:



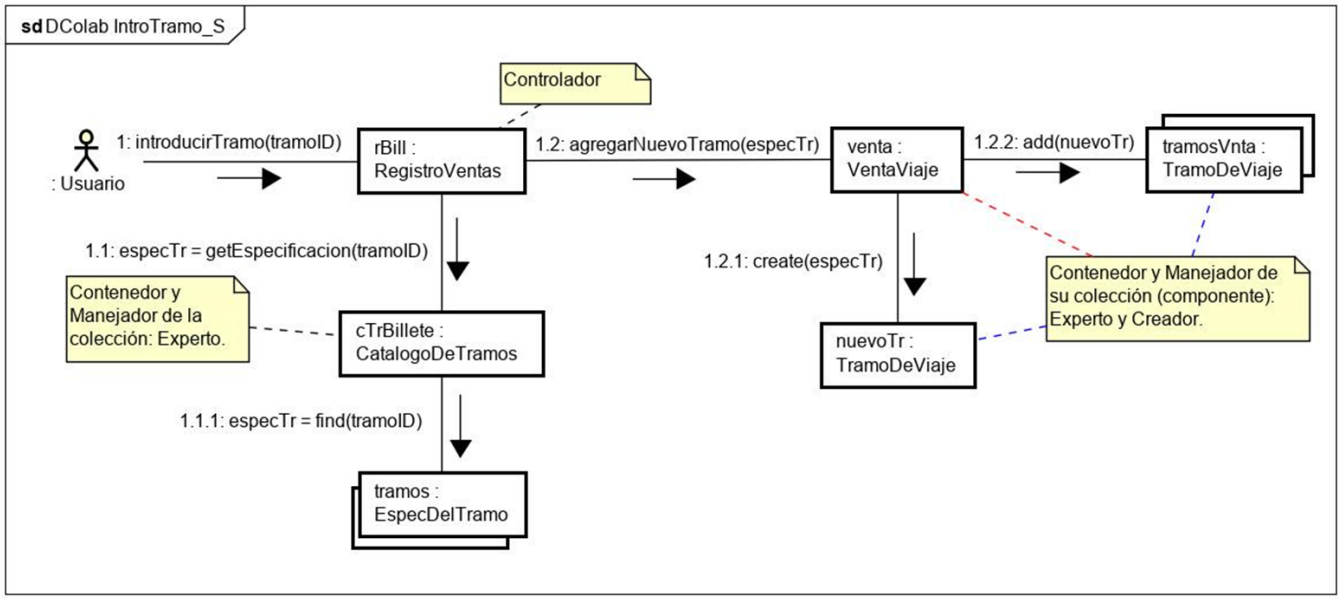
Al finalizar todos los diagramas de iteración (en este caso, los de secuencia detallados), la especificación del diseño es suficiente para determinar, unívocamente, el comportamiento del código. La codificación sería *casi* innecesaria. Obviamente, hay que hacerlo para comprobar que cada caso de uso funciona según lo deseado; pero, si está bien hecho el diseño, el esfuerzo debe ser mínimo. Idealmente hay que hacer pruebas del funcionamiento de cada clase, por lo que se recomienda una estrategia de *codificar con las pruebas*, con lo que se obtiene una realimentación que facilita las modificaciones. En esta misma línea, otra recomendación es codificar comenzando por las clases menos acopladas hasta las más acoladas. En el ejemplo de Procesar Venta, de PdV, se comenzaría por *Pago*, una abstracción polimórfica que se especializa en cada modalidad de pago, y se terminaría con el controlador de nivel superior al caso de uso: Tienda, la clase más acoplada.

En esta situación (aunque también puede ser útil el DCD de cada CU, de factura posterior a los diagramas de interacción –DS o diagramas de colaboración—), se puede pensar en cómo integrar las clases de cada caso de uso para que colaboren entre sí, entre ellas y el control del flujo de trabajo de la aplicación, con la capa de servicios técnicos y con la capa de presentación. En definitiva, a falta de la experiencia necesaria para haber esbozado una organización de las distintas partes de la aplicación, una arquitectura, la integración de los elementos obtenidos tras la especificación detallada de los casos de uso es la que puede permitir tomar las decisiones adecuadas a la hora de definir cómo se agrupan esos elementos en componentes, o módulos, y cómo colaboran entre sí (integración '*botton-up*', a la vez que se define la arquitectura).

El primer paso para definir la arquitectura es realizar una partición en la capa del dominio del negocio (en la que se está trabajando durante todo este desarrollo), agrupando en módulos las clases que se han obtenido en cada caso de uso. Los criterios para realizar esta agrupación son los mismos que se han utilizado al desarrollar el diseño detallado: los principios GRASP; que no son más que una extensión, detallada y aplicada a la orientación a los objetos, de los principios que se exponen en el libro de la asignatura.

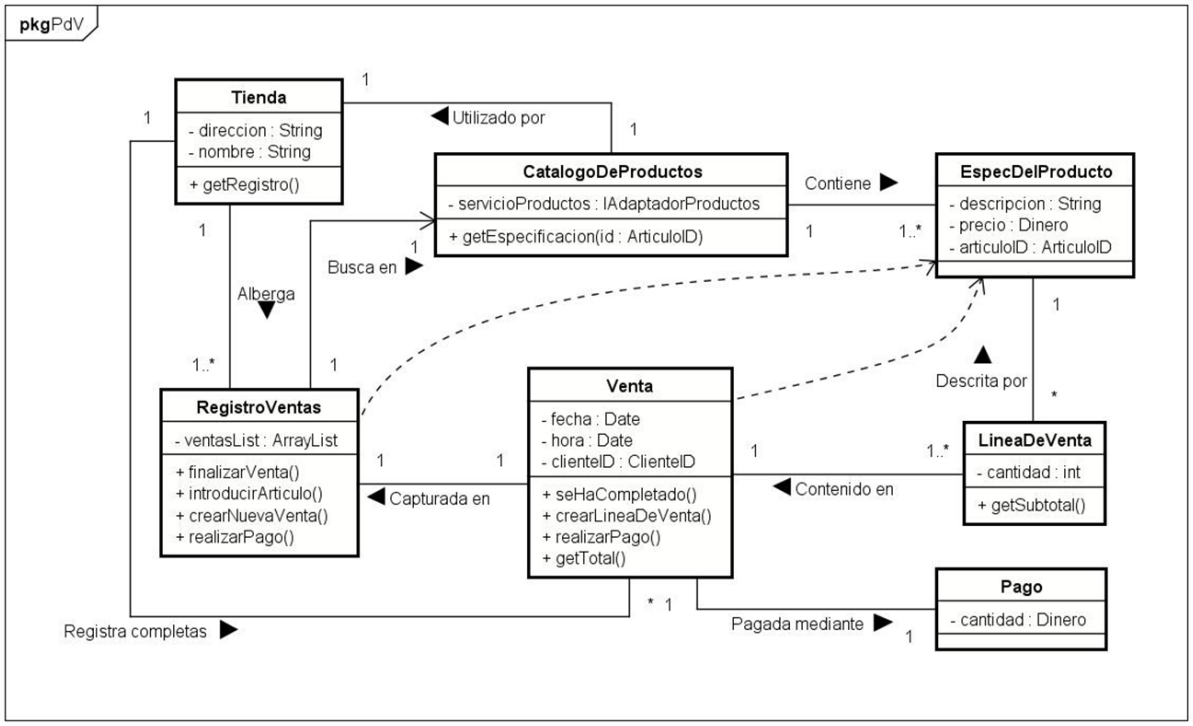
f. Como se ha indicado anteriormente, otro artefacto de diseño detallado, totalmente equivalente al DS anterior, es el **Diagrama de Colaboración**. Igual que el DS, es una representación detallada del paso de mensajes entre las instancias de las clases que participan en el funcionamiento del caso de uso, y sigue sus mismas convenciones formales. Sin embargo, este diagrama no es *vertical*, como el DS, sino *horizontal*; debido a que no se representa la línea temporal de cada instancia. En su lugar, cada mensaje está afectado por un índice (y subíndices, en el caso de que un mensaje se descomponga en otros) que indica, con exactitud, la secuencia temporal en la que se producen esos mensajes y sus consecuencias.

   En este ejemplo, se presenta el diagrama de colaboración equivalente al fragmento de la operación iterativa `introducirTramo(tramoID)`, del DS:



Como se ve, una de las ventajas de ser una representación *horizontal* es que resulta más sencillo descomponer el caso de uso en los diagramas de colaboración de cada operación de la interacción con el actor principal, al contrario que con el DS. Sin embargo, es necesario poner una atención especial a cómo se numeran los mensajes.

g. El último artefacto del diseño detallado, previo a la codificación, es el Diagrama de Clases de Diseño para cada caso de uso. Este diagrama apenas aporta más información para la codificación que el DS, pero recoge los hallazgos de las asociaciones entre clases (especialmente las relaciones de uso, que describen el acoplamiento entre ellas) que sí resultan muy útiles para decidir cómo agruparlas en módulos y construir el diseño arquitectónico.



El ejemplo anterior corresponde al diagrama DCD, simplificado, del caso de uso Procesar Venta en el sistema PdV. Como se puede apreciar, es idéntico al modelo de dominio, pero se han incluido los métodos de cada clase (excluidos los setters y getters), sus argumentos, sus tipos y los de los atributos y, sobre todo, las asociaciones de uso de EspecDelProducto encontradas en las clases principales RegistroVentas y Venta.

Otra diferencia con el modelo de dominio es que, en este diagrama, los objetos que se representan son clases software (código), no *conceptuales*, del uso cotidiano de la aplicación (no debe aparecer el actor ningún otro elemento no software). Está simplificado porque, por el contrario, sí deberían aparecer los artefactos software que intervienen en el funcionamiento. Por ejemplo, el desarrollo de la especificación de Pago y su conexión con el adaptador de Autorización de Transacciones del Pago (especialización de la correspondiente interfaz), si fuera el caso de que el pago fuera a crédito y lo requiriera, o la conexión del servicio de mantenimiento del catálogo con el adaptador para el acceso al almacén de

datos y la interfaz correspondiente. La complementación del diagrama con estas clases puramente software, y las correspondientes asociaciones, son una ayuda adicional para organizar los módulos de la arquitectura.

3. Por último, los diagramas que se utilizan en UML para la representación de la arquitectura son: diagrama de composición de la estructura, diagrama de componentes o, incluso, diagrama de despliegue.

   Como continuación de esta metodología de desarrollo, cercana al proceso unificado (RUP), a la primera implementación y las pruebas unitarias de cada caso de uso, de la 1ª iteración, en esta misma fase de elaboración, le sigue una 2ª iteración en la que, antes de integrar, se refina el desarrollo de cada caso de uso, incorporando atributos cualitativos globales que tienen que ver con la flexibilidad y robustez de la aplicación (persistencia de datos, tratamiento de errores, resistencia a fallos, etc.). Es aquí, en esta 2ª iteración de la fase de elaboración, cuando se trabaja más intensamente en la integración para la que, previamente, es necesario realizar una partición, en módulos o componentes, en cada uno de los ámbitos, o capas, en los que se ha delimitado la elaboración. Nótese que esas capas (en realidad son 3: la del control de la lógica y gobierno de la aplicación está embebida en la de la lógica del negocio y notablemente acoplada con las otras 2) no son una arquitectura de la aplicación, sino un estilo arquitectónico que se ha adoptado y propuesto, aquí, por considerar que la manera de desarrollar a la que lleva tiene una repercusión favorable en el arquitecto inexperto.

   De las notaciones UML indicadas para representar la arquitectura, la más cercana a las que se presentan en el libro, y a las técnicas de diseño que allí se exponen, serían los diagramas de estructura: muy similares a los DCD, pero aplicados a los módulos, paquetes o subsistemas en los que se particiona el software.

   En esta situación, en la 2ª iteración de la fase de elaboración, en la que se han implementado (y probado) todas las clases de los casos de uso, los adaptadores con la IU y con los servicios de acceso externos (datos, etc.), las del gobierno de la lógica de la aplicación y las de la propia funcionalidad en esas 2 capas (presentación y servicios técnicos), la labor del diseño arquitectónico consiste en agrupar esas clases en módulos y *registrar* las asociaciones, dependencias y acoplamientos que existen entre ellos. Esta operación de partición se realiza de manera independiente en cada capa y, tras ello, se *mapean* las dependencias entre las capas y sus respectivos componentes. Como es natural, la partición directriz es la que se realiza en la capa de la lógica del negocio.

   Con poca experiencia, la técnica para realizar la partición, en el ámbito de cada capa, podría ser perfectamente paralela a como se han construido las clases en el diseño detallado y con razonamientos idénticos: agrupando los atributos y elementos primitivos en objetos, según el rol y la función que se pretende asignarles, para que colaboren entre sí de forma que se alcance el objetivo, en este caso, del funcionamiento esperado para la aplicación. Esta técnica se puede asimilar a la descrita en el libro como "*descomposición funcional ascendente*" para las abstracciones: a partir de las estructuras primitivas de los datos, realizar una composición para elevar el nivel de abstracción y, así, sucesivamente, hasta llegar a una abstracción funcional o una representación válida que describa su comportamiento.

   Evidentemente, los criterios que guían cómo agrupar las clases según la función y el comportamiento deseado para el módulo o paquete, en cada paso de esta técnica

ascendente, son los mismos que entonces: los principios GRASP; especialmente los de Experto en información, Acoplamiento bajo y Coherencia alta

A continuación, se presenta un ejemplo utilizando un diagrama UML de estructura: se trata de una supuesta partición con los módulos que se obtendrían a partir del desarrollo de un caso de uso ficticio y del acoplamiento (parcial, por tanto) que se produce.