DISEÑOS FUNCIONALMENTE INDEPENDIENTES, COMPRENSIBLES Y ADAPTABLES

1. Justificación.

En todos los documentos informativos sobre la asignatura (guía, curso virtual, página web, etc.) los resultados del aprendizaje esperados en la asignatura se identifican con el "Diseño de Software". En el documento 'Objetivos de la Asignatura y su Evaluación' se aclara y justifica en qué consiste el **diseño de software**:

> El **diseño** es un conjunto de actividades enfocadas a la **elaboración de representaciones que especifican un código** cuyo comportamiento satisface unos requisitos (especialmente los funcionales) **y, además, es funcionalmente independiente, comprensible y adaptable**.
>
> **Ni** esta asignatura, **ni ninguna** de las indicaciones de su libro de referencia, se refieren a cómo elaborar el software cuyo comportamiento cumpla con unos requisitos funcionales dados (programación, lógica, algoritmia, estructuras de datos, etc.), **sino a cómo hacerlo para que**, sobre la garantía de que los cumple, **el resultado tenga 'independencia funcional'** (acoplamiento débil y cohesión alta), **'comprensibilidad' y 'adaptabilidad'** (flexibilidad, modificabilidad y facilidad de mantenimiento). El malentendido puede deberse a que los procedimientos para conseguirlo se aplican desde el inicio del desarrollo del software, a la vez que se construye el código para que funcione según los requisitos funcionales.

Las destrezas asociadas a las actividades del diseño se superponen, posteriormente, a las de elaborar el código que satisfaga unos requisitos funcionales. Es cierto que construir y especificar el código que únicamente cumple con la funcionalidad pedida también se podría relacionar con un '*diseño*'. Pero el objetivo fundamental de la Ingeniería del Software es organizar las actividades del desarrollo para añadir valores sustanciales a los productos resultantes, lo que convierte en inadmisibles a esos diseños.

Conviene resaltar que la progresión para llegar a las competencias que se manejan en esta asignatura se diversifica en varias etapas:

I. En '*Fundamentos de Programación*' el aprendizaje se dirige al manejo de los recursos del lenguaje, a la elaboración de algoritmos sencillos, la organización en estructuras de los datos que se manejan y, en definitiva, al control del funcionamiento mediante la combinación de esos algoritmos con las estructuras de datos. En esta línea se orientan, conjuntamente, asignaturas como '*Programación Orientada a Objetos*'

y, en general, las relacionadas con la programación, la lógica, la algoritmia o las estructuras de datos.

Es decir, es en este conjunto de asignaturas donde se adquieren las destrezas y conocimientos para elaborar el código que cumpla con unos requisitos funcionales sencillos.

II. En la asignatura de 'Introducción a la Ingeniería del Software' es donde:

- Se presentan una serie de propiedades del producto software que justifican el interés de la ingeniería.
- Se muestra cómo organizar las actividades en la elaboración de software para facilitarla y hacer posible la incorporación de una serie de cualidades que lo mejoran sustancialmente (modelos del ciclo de vida del software).
- Se indica dónde se sitúa el diseño dentro del ciclo de vida, **qué es un diseño** y **qué cualidades aporta al producto software** elaborarlo según las pautas indicadas en esa asignatura.

La conclusión es:

a. **Un diseño es una representación del producto software** (del código), **una especificación técnica que define todos los elementos que lo componen, sus características, el funcionamiento de cada uno y conjuntamente, su organización y cómo ésta posibilita una colaboración cuyo resultado es el comportamiento deseado para el producto.**

b. **El objetivo del diseño es que el código esté dotado de los atributos cualitativos: 'independencia funcional'** (acoplamiento débil y cohesión alta), **'comprensibilidad' y 'adaptabilidad'** (flexibilidad, modificabilidad y facilidad de mantenimiento).

Es decir, en esa asignatura se aprende (entre otras muchas cosas) qué es el diseño de software y qué se persigue con él, pero **no cómo se consigue ese objetivo**.

Por consiguiente, la asignatura de 'Diseño de Software' (incluida en las disciplinas de Ingeniería de Software), no tiene como objetivo aprender a manejar los recursos de un lenguaje de programación, ni a elaborar el código que cumpla con unos requisitos determinados (normalmente los funcionales), sino **aprender cómo, al elaborar ese código, hacer que también sea funcionalmente independiente, comprensible y adaptable**.

De esta forma, esos atributos son cualidades añadidas al código que ya cumple con unos requisitos funcionales y no tiene ningún sentido estudiar cómo se obtienen si no funciona o no lo hace como se ha especificado.

2. Rechazar el acoplamiento.

Quizás por los malentendidos sobre qué trata esta asignatura, se observan algunos defectos en la comprensión de los procedimientos que se proponen en ella y para qué sirven. Parece que el interés se dirige, principalmente, a cómo rellenar los distintos diagramas y a reflejar la aplicación de algún principio GRASP, descuidando:

a. Elaborar la sencilla funcionalidad del caso de uso. Si esto no se consigue, nada del resto tiene sentido.

b. Incorporar los objetivos de esos principios GRASP: independencia funcional, claridad y flexibilidad.

c. Representarlo de una forma razonablemente legible en UML.

Efectivamente, todas las pautas y procedimientos que se dan en la asignatura, los principios GRASP, incluso la aplicación de los patrones GoF, están enfocados a incorporar la 'Independencia Funcional', la 'Comprensibilidad' y la 'Adaptabilidad'.

Considerando que la flexibilidad de un diseño y la facilidad para entender su organización y funcionamiento están directamente relacionados con la independencia funcional, podría ser una simplificación muy eficaz interpretar que lo que se pretende en la asignatura es conseguir un acoplamiento bajo en la implementación de la funcionalidad.

En la orientación a objetos, determinados datos e información se encapsula, estructurada, junto con la capacidad de acción de esa estructura. Es decir, las estructuras de datos son, a la vez, unidades de operación.

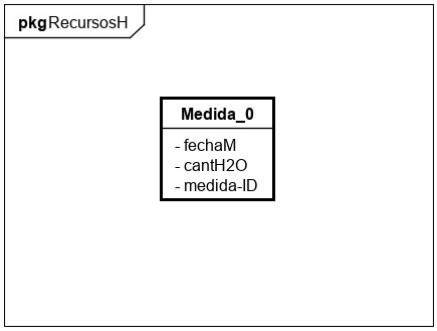
Se ha observado que, a la hora de implementar la funcionalidad pedida para los casos de uso, las estructuras de datos utilizadas por los estudiantes (objetos o clases) están tremendamente acopladas entre sí, interfiriendo notablemente en la aplicación de los principios GRASP (que pretenden lo contrario). Es decir, frecuentemente se tiende a implementar la funcionalidad del caso de uso mediante objetos muy acoplados, que producen diseños monolíticos, y la aplicación de los principios GRASP llevan a *romper* esos diseños, haciendo zozobrar, en numerosas ocasiones, el propio funcionamiento pedido para el caso de uso.

La conclusión es que **es fundamental aprender a construir el funcionamiento del caso de uso mediante objetos desacoplados**; desde el mismo inicio (en el modelo de dominio).

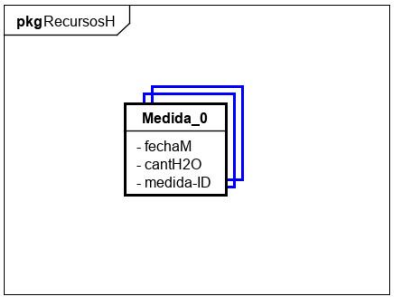
Supongamos que, en un caso de uso, se necesita manejar una característica específica (o un conjunto de ellas) de algún elemento utilizado en la aplicación. Si el manejo se realiza en un conjunto de esos elementos, la funcionalidad del caso de uso requerirá unas operaciones de búsqueda y selección, por lo que es necesario que esos datos estén organizados en unas estructuras (clases) que permitan esas operaciones. Si esas características no dependen de otras, o de otras entidades, lo más eficaz es organizar toda la colección de elementos en **un único catálogo**, aunque la colección se pueda subdividir en distintas categorías.

Para disminuir el acoplamiento en un diseño, es imprescindible evitarlo entre sus clases. Se va a ilustrar con un ejemplo:

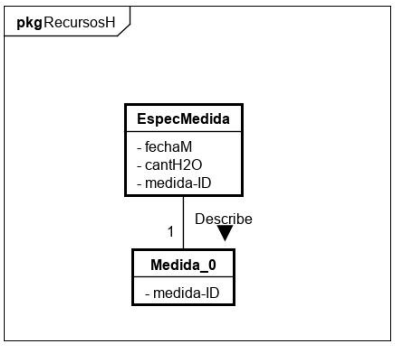
Supongamos una aplicación que maneja la información sobre los recursos hídricos: la cantidad de agua embalsada en una presa. Esta característica, que corresponde a un objeto que llamaremos 'Medida', está asociada al momento de la medición y se podría representar así:



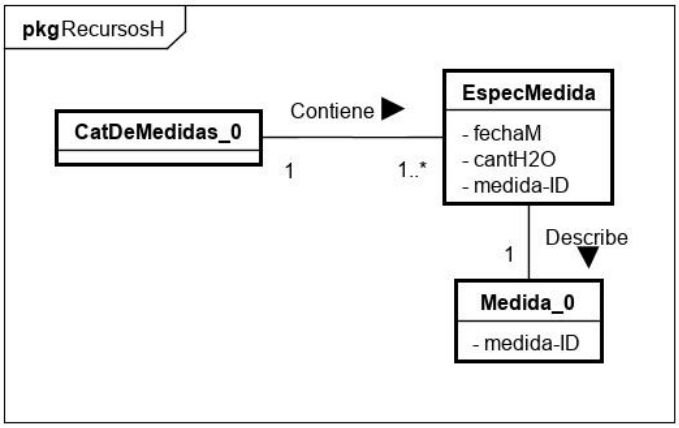
Evidentemente, se tiene una amplia colección de medidas; no sólo las obtenidas en las distintas fechas, sino las de cada presa:



Sin embargo, lo específico de cada elemento 'Medida' son los atributos que aparecen ahí. Por ello, más que representar a una medida, ese objeto delimita las características o especificaciones de cada medida y, también por conveniencia en el manejo que se va a hacer con él, conviene desdoblarlo así:



Ahora bien, lo que presenta en el gráfico anterior es un objeto: el representante de un elemento de una colección, la cual es otra clase distinta. Según sea la estructura elegida para esa colección (Array, List, Map, HashMap, etc.), tendrá unos métodos primitivos para manejar sus elementos. Pero, para manejar a ese objeto, a la colección, es necesario otro objeto contenedor: el catálogo.



Si la colección se organiza en un HasMap {(<Medida>, <EspecMedida>)}, ahora su contenedor, el catálogo, puede realizar la responsabilidad de buscar una medida (según la clave 'medida-ID') y seleccionar las características que se necesitan manejar ('EspecMedida').

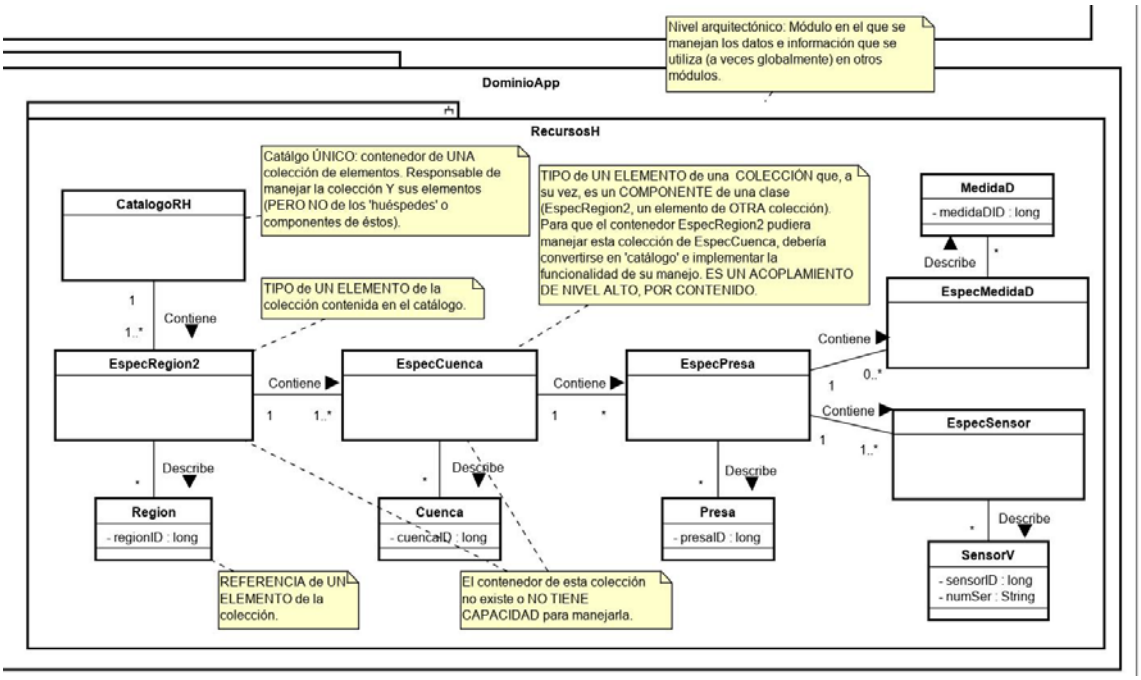
Si la colección se organizara en una List [<EspecMedida>], la implementación, en el catálogo, de la responsabilidad de buscar una medida (según la misma clave, 'medida-ID') y seleccionar el elemento con las características que se necesitan manejar significaría recorrer toda la lista y extraer la clave de búsqueda de cada elemento, para compararlo y seleccionar el buscado. Esta implementación '*irrumpiría*' en la privacidad de los elementos de la colección y produciría un acoplamiento indeseable.

Tal como está construida, la colección contiene la información de todas las medidas recogidas, la que se necesitaría utilizar en los casos de uso de la aplicación. Sin embargo, su semántica es insuficiente para la utilidad que podría buscarse en cualquier aplicación relacionada con el manejo de los recursos hídricos:

> La cantidad de agua embalsada ('Medida') es inherente a una presa concreta. Para facilitar su manejo, los recursos hídricos se pueden organizar en categorías asociadas a conceptos geográficos. Las presas, y las medidas de cada una, se agrupan en 'Cuencas' hidrográficas y éstas en 'Regiones' o vertientes hidrográficas. De esta forma, el interés en manejar la cantidad de agua embalsada se puede referir a una presa, a una cuenca o a toda la región.

Es obvio que para permitir este manejo es necesario incorporar más información, dotar a los datos de una estructura que posibilite su uso y, lo más importante, que esa estructura no esté acoplada porque implicaría que la implementación del funcionamiento tendría un acoplamiento inadmisible.

Uno de los errores más comunes es seguir la lógica de la organización semántica de la información: cada elemento de la colección de características de las regiones contiene una colección de las de sus cuencas y, a su vez, cada especificación de una cuenca tiene una colección de las especificaciones de sus presas y, por fin, cada especificación de una presa tiene una colección de sus medidas. Algo así:



En la estructura anterior se recogen todos los datos que se necesitan en la aplicación y la información está *enlazada* de forma que se obtiene todo el significado que se requiere. Por supuesto, con total independencia de cómo se almacenen en el exterior a la aplicación: en un fichero, en una base de datos, en un sistema de información deslocalizado, etc.

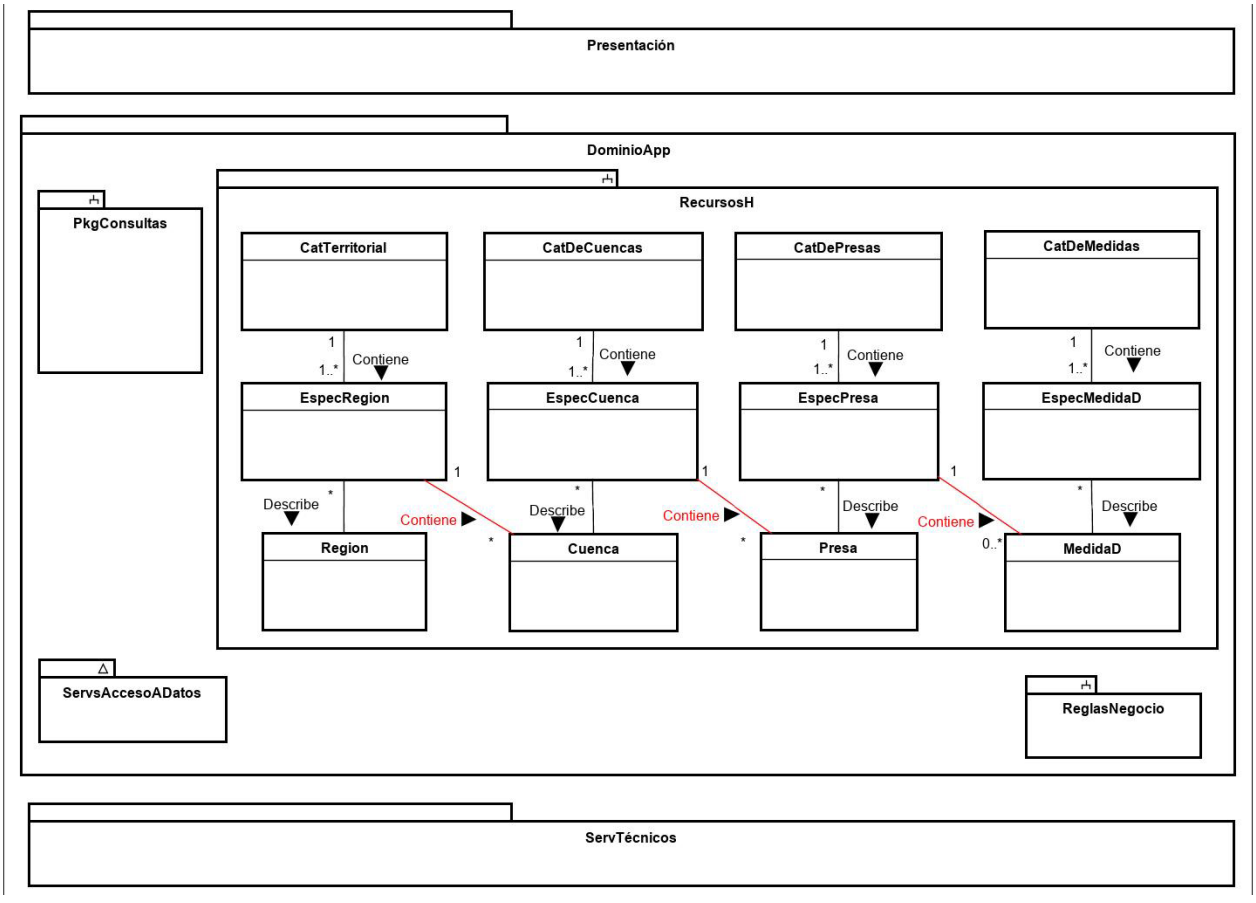
Por otro lado, un único contenedor (el catálogo) tiene la responsabilidad de manejar todas las colecciones que, aunque estén una dentro de otra, son de naturaleza completamente diferente (ignorando la de los sensores, hay 4 colecciones distintas). Esto obliga a que cada elemento debe asumir la responsabilidad de manejar la colección contenida en él y que cualquier maniobra para usar los datos se debe iniciar desde el catálogo general, transmitir la orden a la colección que interese y transgredir la privacidad del elemento destinatario al recibir el resultado.

Lo que lo hace más intolerable es que el acoplamiento entre las colecciones es por contenido: cada elemento contiene una colección con los datos que se necesitan manejar en la aplicación.

Para minimizar ese acoplamiento inadmisible, **hay que organizar los datos de otra manera**:

1. Independizar los catálogos, con uno por cada tipo de colección que contiene (cohesión alta).

2. Vinculando las colecciones, no por los datos que se manejan (por contenido), sino por una referencia o etiqueta a ellos.



Esta forma de organizar la información (que también es diseño) mantiene el significado que se necesita para el uso de los datos, pero está enfocado a independizar dicho uso; **a la independencia funcional** (el acoplamiento débil y la cohesión alta), a **la comprensibilidad y a la adaptabilidad que debe tener todo diseño**.

Es muy probable que esta información (la de los recursos hídricos) se utilice transversalmente, con amplitud, en distintas funciones (casos de uso) de la capa de negocio de la aplicación. Seguramente debido a ese uso generalizado, para recoger la responsabilidad de mantener los datos de esas estructuras y, a la vez, conseguir una cohesión alta, se decida situarlas en un módulo específico en el diseño arquitectónico.

En este ámbito de la capa del negocio, y con la misma perspectiva de la descripción del funcionamiento global de la aplicación y de la colaboración de los diferentes módulos que la constituyen (diseño arquitectónico), el controlador del flujo de trabajo en su funcionamiento general (el nivel de la aplicación) tomará las estructuras de los datos que se requieran en un caso de uso determinado y se las transmitirá a ese controlador específico. Por este motivo, **en** la descripción detallada del funcionamiento de **un caso de uso** (diseño detallado) **nunca se crean los catálogos ni sus colecciones** de datos, sino que existen, ya, en un nivel de funcionamiento anterior (generalmente se crean en la inicialización de la aplicación). En principio, un caso de uso no se localiza en un único módulo de la arquitectura, sino que su diseño detallado describe, por un lado, qué objetos software se utilizan en su funcionamiento (y cómo se utilizan) y, a la vez, con qué objetos software (y de qué manera) se implementa la colaboración entre un módulo de la arquitectura y otro. Es decir, el diseño detallado no solo es la especificación del funcionamiento del código, sino la especificación del diseño arquitectónico (de qué manera se implementa esa descripción de la colaboración entre módulos).

Otro ejemplo de colaboración entre módulos de la arquitectura, siempre en la capa de las funciones del negocio, es la que se refiere a los servicios de acceso a los datos externos. El módulo encargado de ello tiene una Factoría de servicios (singleton y global para la aplicación) que provee, a cada contenedor de las colecciones de datos (catálogo), una interfaz capaz de realizar todas las operaciones de mantenimiento de esos datos, pero a través de la Fachada del sistema externo en el que están almacenados los valores de esos datos. Este es el motivo de que, al diseñar en detalle un caso de uso, no haya que preocuparse de dónde se obtienen los datos, **sino de qué datos son necesarios y con qué estructura hay que organizarlos** para que su uso sea 'Funcionalmente Independiente', 'Comprensible' y 'Adaptable'. En definitiva: **que estén desacoplados**.

En conclusión, a la hora de diseñar el código que implementa un caso de uso:

1. Hay que entender cuál es el objetivo de la aplicación y su funcionalidad principal (casos de uso primarios), los límites de su capa de negocio y sus interacciones con el exterior (a través de la capa de presentación y de la de servicios técnicos).

2. Una vez ubicado, centrándonos en el caso de uso concreto, entender qué hace, qué operaciones realiza, con qué secuencia se hacen y qué datos o información se necesita para hacerlo. Con estas conclusiones, se crea un modelo de funcionamiento (modelo de dominio). En este modelo se representa la capacidad de los objetos para realizar esas operaciones, por lo que deben quedar claros los roles funcionales que cumplen: controlador o manejador del CU, el objetivo del CU, la organización de los datos para que puedan ser utilizados, etc. Y, sobre todo, **es necesario *idear* esas estructuras con acoplamiento bajo entre ellas**. Los principios GRASP que más se van a manejar en este momento (elaboración del modelo de dominio, un modelo *conceptual* del funcionamiento del caso de uso), son:

    - Principio de Controlador: ¿Qué objeto va a organizar el funcionamiento?

    - Principio de Experto: ¿Qué datos contiene cada objeto, por lo que sólo él va a recibir la responsabilidad de realizar las operaciones que utilizan esos datos?

3. El siguiente paso es construir una representación del funcionamiento detallado del código: implementar el diseño detallado. En UML, la representación que se corresponde con la especificación detallada del funcionamiento de todos los elementos del código, y las interacciones que se producen entre ellos, es **el diagrama de secuencia, completo y detallado** para el caso de uso, **o la colección completa de los diagramas de colaboración para todas las operaciones** que componen el caso de uso. Es en estos diagramas donde queda patente (y sin lugar a duda, porque son una especificación técnica de lo que se aporta como solución del problema planteado):

    a. Si el código cumple con los requisitos funcionales del caso de uso. Es decir, el diagrama debe contener la descripción técnica completa del funcionamiento de todos los elementos del código que intervienen en el caso de uso; mediante la secuencia, ordenada, del paso de mensajes (y sus argumentos) entre las instancias que componen ese código. Si esta representación no existe, es incorrecta o ilegible, o no funciona como se pide, no hay diseño o no se puede valorar sus características.

    b. Si se cumple la condición anterior, se puede valorar si el diseño está **débilmente acoplado** y la cohesión de sus elementos, si la organización de su funcionamiento **es comprensible** o si es flexible, **adaptable** y facilita su mantenimiento.