Laboratorio de Ingeniería del Software

Expresar un modelo de dominio en software

Contenidos

- Aislar el dominio
  - Arquitectura por capas
- Un modelo expresado en el software
  - Asociaciones
  - Entidades
  - Objetos valor
  - Servicios
  - Paquetes
- Paradigmas de modelizado

Aislar el dominio

Aislar el dominio

- La parte de un software que resuelve específicamente problemas del dominio suele ser pequeña
  - Pero su importancia es mucho mayor de lo que el tamaño implicaría
- Lo esencial es asegurar que se mantienen los **invariantes del dominio** de problema que requieran **nuestras historias de usuario**
  - Ni más, ni menos que eso
- Es importante desacoplar los objetos del dominio del resto de funcionalidad y aspectos técnicos del sistema para poder focalizarnos

Aislar el dominio

- En un programa se puede mezclar GUI, acceso a datos y código de soporte con el código que expresa el dominio de problema
  - Por ejemplo, incluyendo reglas de dominio en los componentes de la GUI o en las consultas de la base de datos
  - Esto es lo más cómodo a corto plazo
- Si el código relacionado con el domino está disperso y mezclado con otro código:
  - Es más difícil localizarlo y razonar sobre el
  - Cualquier cambio superficial (en la GUI o en la B.D.) pude alterar, sin querer, la lógica del dominio
  - Un cambio en alguna regla del dominio implica buscar el código relacionado, que está mezclado y disperso
  - Las pruebas automáticas se complican

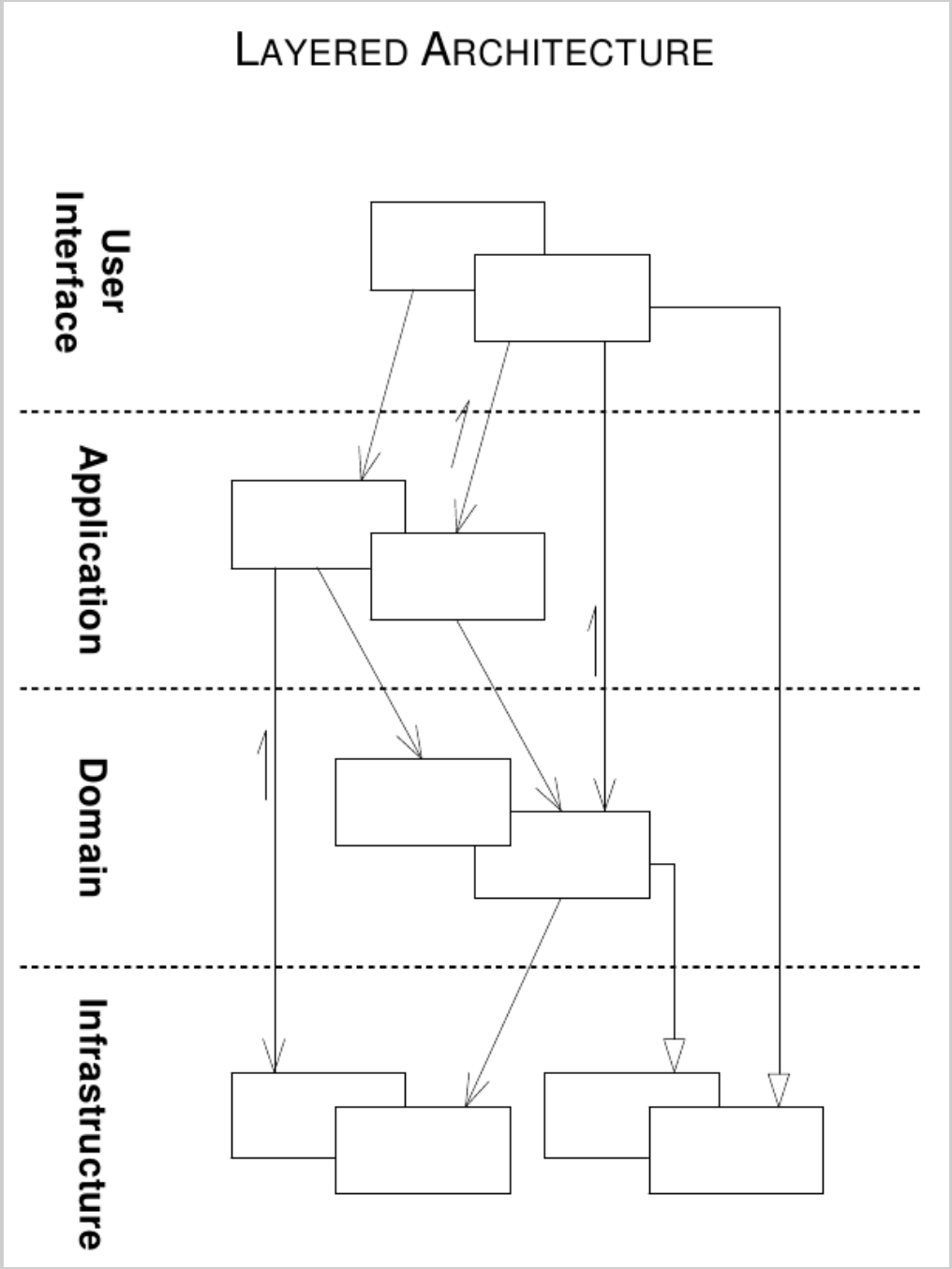
Arquitectura por capas

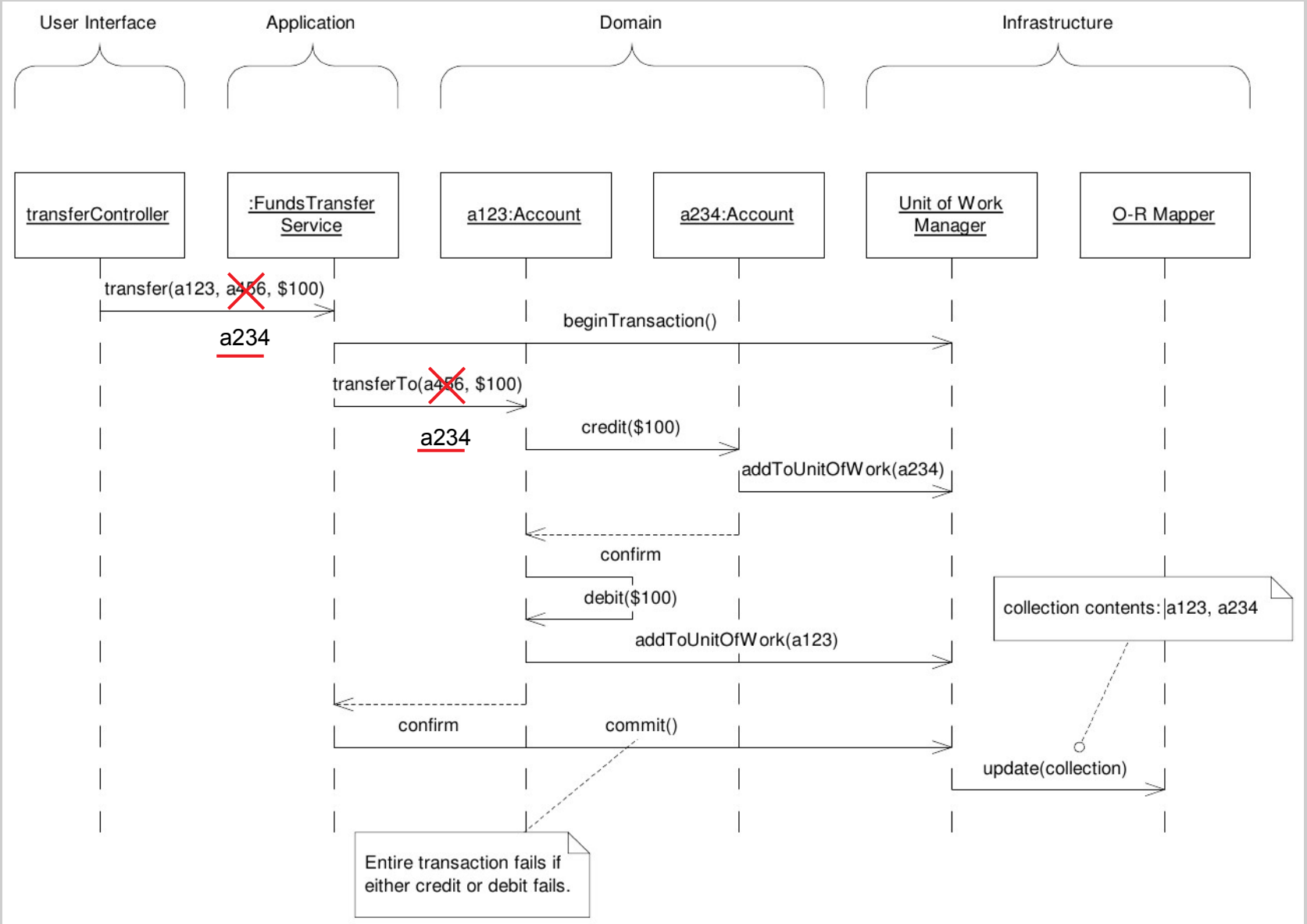
- La solución más común a este problema es una arquitectura por capas, donde cada capa está especializada en un aspecto del software
  - La versión más común tiene tres o cuatro capas
- Interfaz de usuario (capa de presentación)
  - Muestra información al usuario e interpreta sus órdenes
- Capa de aplicación
  - Define lo que hace el programa que estamos construyendo y encarga tareas concretas a los objetos del dominio
  - No contiene conocimiento del dominio, esencialmente es una capa de coordinación
  - No guarda estado que refleje la situación del dominio, pero puede guardar estado que refleje el progreso de una tarea del usuario o del programa

Arquitectura por capas

- Capa de dominio (de modelo, de lógica de negocio)
  - Contiene los conceptos y reglas del dominio y el estado del mismo
    - Ese estado se controla y usa aquí, pero los detalles técnicos sobre el almacenamiento del mismo se delegan a la infraestructura
  - **Esta capa es el núcleo del software**
- Capa de infraestructura
  - Tecnología genérica para dar soporte a las capas superiores
    - Envío de mensajes, persistencia, componentes de la GUI...
- En algunos proyectos la capa de interfaz de usuario y la de aplicación no están separadas
  - Otros tienen varias capas de infraestructura
  - **Lo crucial es que la capa de dominio esté separada**

LAYERED ARCHITECTURE





Arquitectura por capas

- Cada capa tiene que ser cohesiva y estar conectada con las otras capas, pero no demasiado acoplada con ellas
- Los objetos de una capa deben tener dependencias solo con objetos de alguna capa inferior
  - Llamando a sus interfaces públicas y guardando referencias a ellos (temporalmente)
- Si un objeto de una capa inferior necesita comunicarse con uno de una capa superior (además por supuesto de responder cuando ha sido invocado desde arriba) debe hacerlo por métodos indirectos
  - Por ejemplo usando *callbacks*, o el patrón sujeto-observador
- La conexión entre la capa de interfaz de usuario y la del dominio típicamente usará alguna variante del patrón modelo-vista-controlador

Arquitectura por capas

- La capa de infraestructura puede ofrecer servicios genéricos (p.ej. enviar e-mails) a la capa de aplicación
  - Los objetos de la capa de aplicación saben cuándo hay que enviar un e-mail, pero no tienen por qué saber cómo se hace
- No toda la infraestructura se puede separar limpiamente en forma de servicios
  - Por ejemplo, es típico que para implementar un *widget* de GUI tengamos que extender una clase "*widget* genérico" de nuestra infraestructura
  - Hay que tener cuidado para que nuestra elección de *frameworks* de infraestructura no complique el adecuado reparto de responsabilidades entre los objetos de nuestro programa

Un modelo expresado en el software

Conectar diagramas e implementación

- Para que funcione el diseño dirigido por el dominio, el modelo de dominio tiene que ser único, aunque lo expresemos de distintas formas y con distintos niveles de detalle
  - Por ejemplo, si nuestros diagramas UML dicen una cosa y nuestro código hace otra, no tenemos un buen modelo de dominio
- Todas las asociaciones entre clases son igual de fáciles de dibujar en un diagrama, pero luego unas son más complicadas de implementar que otras

Asociaciones

- Por cada asociación navegable en los diagramas tiene que haber un mecanismo de implementación que la soporte y tenga las mismas propiedades
  - Pueden implementarse con referencias entre objetos o incluso encapsulando una consulta de base de datos
- Por ejemplo, una asociación de 1 a * puede implementarse como
  - Una colección que sea propiedad del objeto del lado 1
  - O bien como un método de ese objeto que consulta a una BD, coge la respuesta y la convierte en * objetos
- El mundo real tiene muchas asociaciones * a * e, ingenuamente, nos puede parecer que casi todas deberían ser bidireccionales
  - Pero esas son las más complicadas de implementar

Asociaciones

- Hay tres formas de hacer que las asociaciones sean más fáciles de manejar
  - Imponer una dirección de navegación
  - Reducir la multiplicidad
  - Eliminar asociaciones que existen en el dominio pero que no son esenciales en nuestro problema
- Restringir las asociaciones es importante para crear y mantener el tipo de modelo único que buscamos en DDD
  - Lo bastante expresivo para entender el problema y lo bastante simple para que el diseño/implementación sea un proceso directo

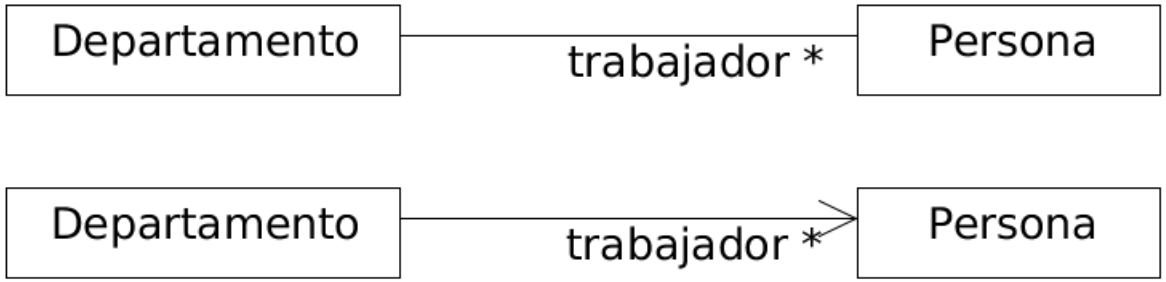
Restringir las asociaciones

- Coger una relación 1 a * o * a * y especificar una dirección de navegabilidad simplifica mucho el diseño
  - Por ejemplo, si la asociación es bidireccional, y añado un objeto a la colección del lado * ese objeto debe recibir información sobre cuál está en el lado 1
  - Si es unidireccional, puedo añadir objetos a la colección del lado * sin que estos tengan por qué saber cuál está en el lado 1
- Si se restringe la navegabilidad en una relación * a *, esta se puede implementar como una relación 1 a *, mucho más fácil de manejar
  - Seguirá siendo cierto que ambos lados tienen multiplicidad *, pero como los enlaces entre los objetos solo serán navegables en un único sentido, la implementación puede ser una simple colección propiedad del lado no navegable, como si fuera una 1 a *

Pregunta

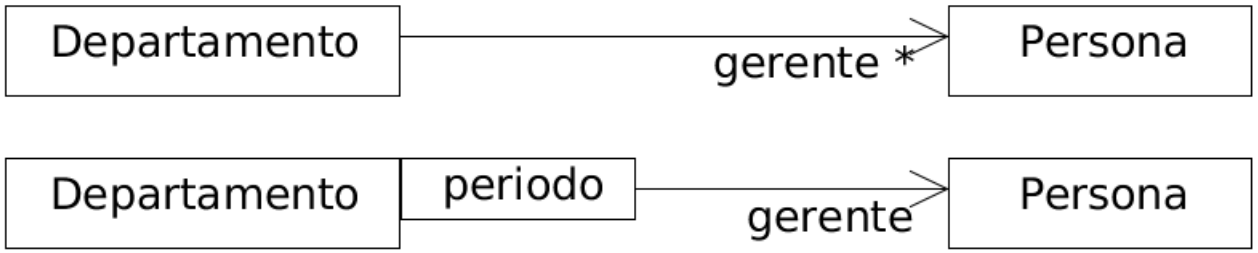
- ¿Recordáis cómo implementamos una relación * a * entre dos clases en Java, sin BD?

- Supongamos que un departamento de una empresa tiene muchos trabajadores
  - Y cada trabajador/a pertenece a un único departamento
  - Nos puede parecer una relación bidireccional (el departamento tiene trabajadores, los trabajadores pertenecen al departamento)
- Sin embargo, especificar una navegabilidad en la asociación separa mejor el concepto "Persona" del concepto "Trabajador en un departamento"
  - Si la relación es bidireccional, pertenecer o no a un departamento está muy ligado a ser persona
    - Cada instancia de Persona tiene acceso directo a su objeto Departamento correspondiente
  - Si es unidireccional, las personas no están directamente enlazadas con departamentos, luego el departamento (o departamentos) al que pertenece cada Persona ya no parece que sea tan importante
- Además, si nuestro problema consiste principalmente en modelizar una empresa, el concepto Departamento será más importante, las consultas generalmente vendrán desde ahí y la segunda versión lo refleja mejor



Restringir las asociaciones

- Una comprensión más detallada de una asociación puede llevar a modelizarla restringiéndola con un clasificador (*qualifier*)
  - P. ej. Un departamento habrá tenido muchos gerentes, pero solo uno en cada periodo de tiempo (en cada periodo por tanto la relación es 1 a 1)



Restringir las asociaciones

- Por supuesto, la restricción definitiva es eliminar la asociación del modelo, aunque exista en el dominio, si esta no es esencial
  - Esencial, significa esencial para la aplicación que estamos construyendo
    - Recordad: solo nos interesan **los invariantes de dominio que requieren nuestras historias de usuario**
- Restringir las asociaciones entre clases del dominio de acuerdo a nuestras necesidades, hace el modelo más simple de entender y de implementar
  - Y además las relaciones más complejas (las bidireccionales y/o * a *) que queden en el modelo estarán ahí porque son realmente importantes para expresar bien el dominio y las necesidades de nuestra aplicación y no por ingenuidad

Entidades

Entidades

- Muchas cosas se definen por su identidad, no por sus atributos
  - P.ej. la identidad de una persona permanece desde su nacimiento hasta su muerte, y más allá (p.ej. por temas legales)
    - Pero sus atributos cambian (altura, peso...) o pueden cambiar (nombre, dirección...) a lo largo de su vida e incluso desaparecen después
- La identidad conceptual debe ser mantenida entre distintas implementaciones de los objetos, sus formas almacenadas (p.ej en BD) y ser compartida con la entidad de la realidad
  - Cuando llega un cliente al banco y enseña su DNI debemos poder relacionarlo de forma inequívoca con un objeto en nuestro sistema, un registro en nuestra BD...

Entidades

- Aunque el modelizado de objetos tiende a focalizarse en los atributos de los mismos, el concepto fundamental de una entidad es una continuidad abstracta que lo enlaza en el tiempo, a través de su ciclo de vida y entre sus distintos formatos
- Debemos poder distinguir entre dos entidades distintas aunque tengan los mismos atributos, y determinar que dos objetos con atributos distintos pueden ser la misma entidad

Entidades

- Una entidad (u objeto referencia) es un objeto definido principalmente por su identidad
  - Debe tener una operación que devuelva un símbolo único que sea su identidad / identificador
- Tienen ciclos de vida que cambian radicalmente su forma y sus contenidos, pero debe mantenerse un hilo de continuidad
  - Debe poder distinguirse su identidad independientemente de su forma o su historia
- El modelo tendrá que dejar claro qué significa exactamente que dos entidades sean la misma
- Muchas entidades no tendrán reflejo en entidades tangibles del mundo real y representarán los conceptos abstractos del dominio que tienen identidad y continuidad

Entidades

- La identidad no es intrínseca a las cosas del mundo, es algo que les ponemos porque nos resulta útil
  - La misma cosa del mundo real puede representarse como entidad o como no entidad en distintos modelos de dominio
  - Por ejemplo, en un sistema de reserva de asientos en un estadio:
    - Un asiento puede modelizarse como una entidad si cada entrada da derecho a un asiento concreto
    - Un asiento no necesita ser una entidad si la entrada da derecho a usar cualquier asiento libre

Modelizar entidades

- La clase que modeliza una entidad debería limitarse a sus características intrínsecas
  - Aquellas que la definen y/o se usan normalmente para buscarla
    - Por ejemplo, aunque cada objeto que modeliza una entidad Persona de nuestro sistema tenga un identificador numérico propio, el nombre y el DNI deben pertenecer a esta clase, porque se usarán muchas veces para localizar la persona en el sistema
- Otras características de la entidad deberían ir en otros objetos asociados
  - Objetos valor
- Las entidades suelen tener responsabilidades conceptuales importante en el sistema y necesitarán también operaciones para implementarlas
  - Muchas veces coordinando operaciones de los objetos que poseen

Identificar entidades

- Algunas veces un conjunto de atributos se puede garantizar que son únicos dentro del sistema y nos sirven como identidad para algunas entidades
- Pero **la solución que sirve siempre** es asignar a cada instancia un símbolo (p.ej., número o cadena) único dentro de la clase
  - Este ID debe ser inmutable y acompañar al objeto en todo su ciclo de vida
  - Este ID seguirá siendo válido aunque con el tiempo modifiquemos los atributos de la entidad
- A veces necesitamos que el ID sea único incluso más allá de nuestro sistema
  - Esto requiere especial cuidado
- También es distinto el caso en que el ID tenga que ser o manipulado por personas
  - Por ejemplo, que sea fácil de leer o escribir solo es importante en este caso

Identificadores únicos [universales]

- "Identificador único" casi es redundante, puesto que un identificador que puede corresponder a varias entidades no es un buen identificador
  - Pero se suele decir así, supongo que para resaltar su principal cualidad
- Podemos distinguir los identificadores únicos por ser o no inteligentes
  - Uno inteligente tiene al menos un parte con significado en el dominio del problema
    - Por ejemplo un DNI como identificador en nuestra base de datos para nuestros clientes
  - De uno no inteligente no podemos deducir ninguna información
- También podemos distinguirlos por ser o no universales
  - Un identificador único universal (UUID) está [casi] garantizado que es totalmente único (incluso fuera de nuestro sistema de información) si se genera de acuerdo a ciertas reglas

Pregunta

- ¿Es mejor que nuestros identificadores sean inteligentes o no?
- ¿Es mejor que nuestros identificadores sean universales o solo únicos en nuestro sistema de información?
- Deberes: consultad, p.ej., la documentación de la clase `java.util.UUID`

Objetos valor

Objetos valor

- Muchos objetos no tienen una identidad conceptual
  - Son descripciones de las características o de la naturaleza de algo
- Asignar identidad a objetos que no son entidades puede empeorar las prestaciones del sistema
  - Además de que complica el modelo haciendo que todos los objetos parezcan igualmente importantes
- El diseño de software es una batalla constante contra la complejidad
  - El tratamiento especial que requiere una entidad hay que usarlo solo cuando es necesario

Objetos valor

- Un objeto que describe un aspecto del dominio que no tiene identidad conceptual se llama objeto valor
  - Representan elementos del dominio que nos interesan por lo que son, no por quiénes son
- Los tipos básicos (cadenas, números...) y no tan básicos (fechas, colores...) son ejemplos de objetos valor (en lenguajes orientados a objetos)
  - Pero pueden ser bastante más complejos y seguir siendo objetos valor
  - Incluso pueden ser grupos de objetos relacionados

Objetos valor

- Un objeto valor puede referenciar a algunas entidades
  - Por ejemplo una ruta automáticamente calculada entre dos ciudades puede representarse como un objeto valor que tiene referencias a varias entidades
    - Esas entidades serían las ciudades origen y destino y las carreteras y calles intermedias
- Tienen varios usos habituales
  - Ser parámetros en las llamadas entre objetos
  - Crearse para una operación y luego descartarse
  - Ser atributos de entidades y de otros objetos valor
    - La persona se modeliza como una entidad, su nombre se modeliza con un objeto valor

Objetos valor

- Si de un elemento del dominio solo te interesan sus atributos:
  - Modelízalo como un objeto valor que tenga los atributos y la funcionalidad necesarios
    - Los atributos de un objeto valor deberían estar estrechamente relacionados (cohesión alta)
  - Trátalo como si fuera inmutable
  - No le des identidad
    - No es una entidad y no la necesita

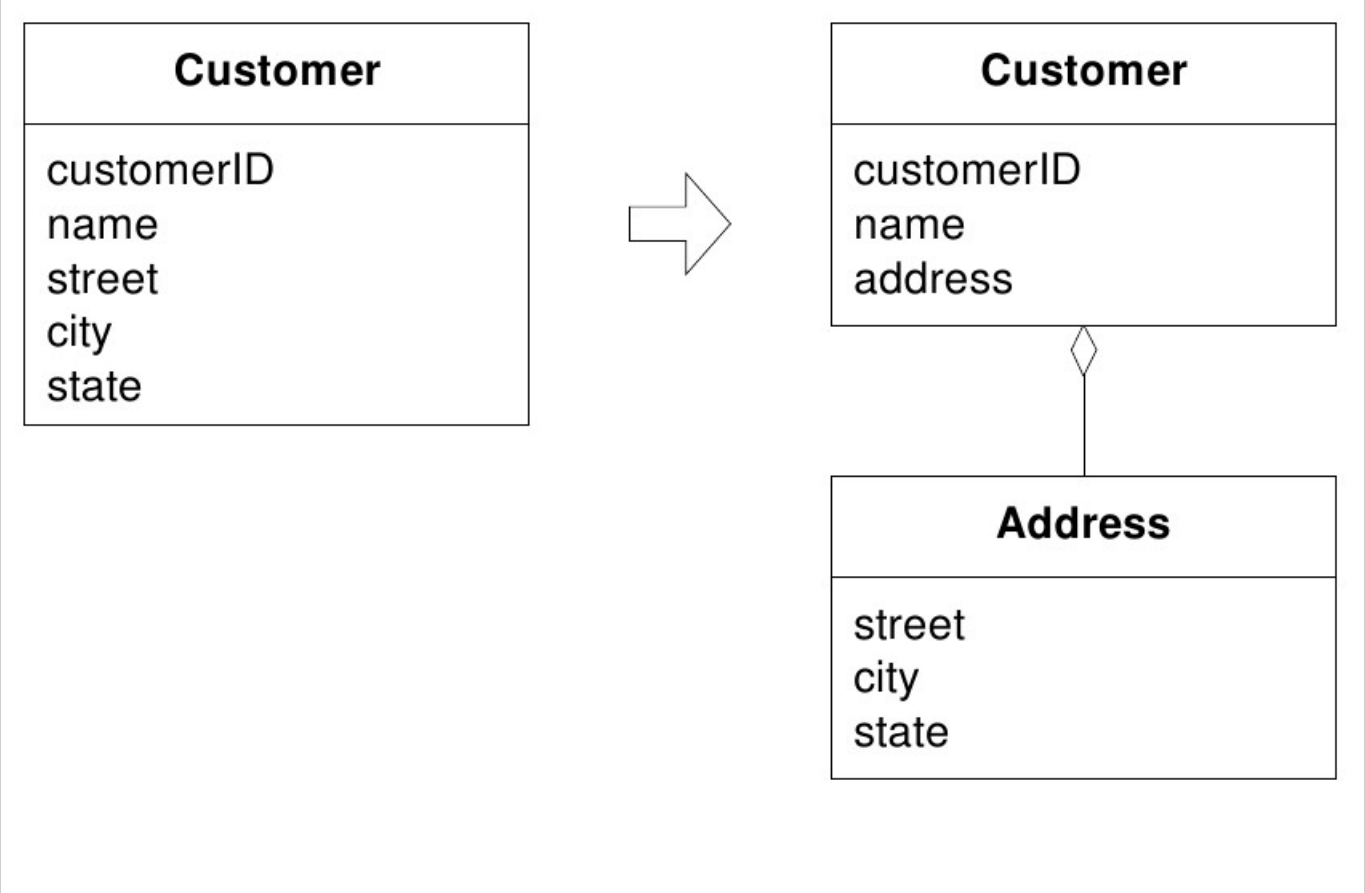


Figure 5. 2

Eric Evans
Domain-Driven Design...
Figure 4.2

Pregunta

- En la figura anterior, *Address* es un objeto valor. ¿Es esto siempre una buena elección para una dirección postal?

Modelizado de objetos valor

- Dos objetos valor de la misma clase y con los mismos atributos son totalmente equivalentes
  - Nos da igual qué instancia tenemos, o cuántas copias pueda haber en distintas partes del sistema
- Un objeto valor puede copiarse libremente
  - P.ej. no tenemos que preocuparnos de generar un ID nuevo para cada copia
- Si es inmutable (una vez construido no puede ser alterado) puede compartirse sin peligro entre distintos objetos, o pasarse como parámetro sin riesgo de que sea alterado indebidamente
  - Si hay muchos objetos valor, y suele ser así, el poder reutilizarlos es una gran ventaja en prestaciones (la creación de objetos suele ser relativamente cara) y uso de memoria

¿Objetos valor mutables o inmutables?

- La inmutabilidad simplifica la implementación y hace que compartir y pasar referencias sea seguro
  - Facilita compartir el objeto. **Uno mutable no debe ser compartido libremente**
- Sin embargo, hay ocasiones en que la mutabilidad de los objetos valor será necesaria para mejorar prestaciones:
  - Cuando el objeto valor cambia a menudo, especialmente si la creación/borrado de objetos es cara
  - Si el reemplazo complica más algunas tareas (por ejemplo la gestión de la B.D.) que la modificación
- **Diseña objetos valor inmutables salvo que tengas un problema con las prestaciones**
- Si trabajas en Java, hay que tener en cuenta unas cuantas cosas para tener objetos valor inmutables
  - Clase final, atributos private y final, constructor único, sin setters, copias defensivas en los getters si es necesario…
  - También puedes darle un vistazo al proyecto Lombok (anotación @Value)

Pregunta

- ¿Tendría sentido que tuviéramos entidades inmutables?

Asociaciones que involucran objetos valor

- En general, como hemos visto antes, cuantas menos asociaciones haya, y más simples sean, mejor
- **Las asociaciones bidireccionales**, que en las entidades pueden ser algo complicadas de mantener, **no tienen sentido para objetos valor**
  - Decir que si desde un objeto a:A podemos navegar hasta uno b:B entonces desde b:B podemos navegar hasta a:A (asociación bidireccional) solo tiene sentido si podemos *distinguir* a:A de a':A cuando a y a' tienen los mismos atributos, lo que requeriría que tuvieran *identidad*
  - Por tanto, elimina las asociaciones bidireccionales entre objetos valor
    - Y si queda alguna que te sigue pareciendo necesaria, posiblemente tienes alguna entidad de la que no eras consciente

Servicios

Servicios

- Puede haber conceptos en el dominio de problema que son intrínsecamente actividades o acciones
  - Si nuestro paradigma de modelizado son los objetos, trataremos de encajarlas en los objetos que tengamos
- Si forzamos mucho esto, la responsabilidad de algunos objetos dejará de ser clara
  - En ellos estaremos violando el principio de responsabilidad única
- Otras veces se crean objetos solo para contener operaciones
  - No tienen estado ni significado propio en el dominio
  - Suelen acabar llamándose `XManager`, `Utils`... o algo así
  - Esta alternativa no es ideal, aunque es mejor que la del punto anterior

Servicios

- Un servicio (de dominio) es una operación ofrecida a través de una interfaz que tiene presencia por sí misma en el modelo, sin encapsular estado
  - Representa un concepto dinámico del dominio, un comportamiento que tiene sentido para sus expertos
  - Entidades y objetos valor sí que encapsulan estado, los servicios no
  - No es el mismo tipo de servicio que al que nos referimos al hablar por ejemplo de servicio web, o arquitectura orientada a servicios, que son patrones técnicos

Servicios

- Los servicios se definen solo en función de lo que pueden hacer por los objetos que los invocan
  - Se les suele nombrar por una actividad y no por un nombre
    - P.ej. `CalculadorRutas` o `ServicioCalculoRutas` y no `Ruta`
- Igual que los objetos, los servicios tienen responsabilidades
  - Las responsabilidades, y las interfaces que las soportan, deben definirse en términos del lenguaje del dominio
  - Los nombres de las operaciones deben venir del lenguaje compartido o ser añadidos a este

Servicios

- Un buen servicio tiene tres características
  - La operación se relaciona con un concepto del dominio que no es parte natural de una entidad o de un objeto valor
  - La interfaz se define en términos de otros elementos del modelo del dominio
    - P.ej. sus parámetros son objetos valor o entidades
  - Las operaciones no guardan memoria o estado (*stateless*)
    - Cualquiera puede usar cualquier instancia del servicio independientemente de la historia individual de esa instancia, porque la historia no afecta a su ejecución
- La ejecución de un servicio puede usar información que esté accesible globalmente (p.ej. en un repositorio de datos compartido) e incluso puede cambiarla
  - Pero no guarda un estado propio que afecte a su comportamiento futuro

Servicios de dominio y de aplicación

- Cuando se habla de servicios en sistemas informáticos, normalmente son técnicos y su sitio natural es la capa de infraestructura
- Los servicios de dominio (y los de aplicación), que son los que nos interesan aquí, colaboran con ellos
  - Por ejemplo, un servicio web (infraestructura) puede ofrecer cuatro operaciones, que en la capa de aplicación se implementan como cuatro servicios distintos, dos de los cuales llaman a un mismo servicio de la capa de dominio
- Muchos servicios llaman a operaciones ofrecidas por las entidades y los objetos valor del dominio
  - Estos servicios pueden pertenecer al dominio o a la aplicación, y a veces no habrá una clara razón para preferir una capa a la otra

Preguntas

- En una aplicación de banca con las cuatro capas que hemos visto, en qué capa pondríamos...
- ... un servicio para exportar transacciones financieras a una hoja de cálculo en formato OpenDocument
- … un servicio para transformar hojas de cálculo OpenDocument a hojas de cálculo Excel

Los servicios en la capa del Dominio

- Los servicios de dominio también pueden ayudar a simplificar nuestro sistema
  - Uno que dé acceso a operaciones del dominio que están repartidas en distintas entidades/objetos valor puede ser una buena opción si tenemos muchos objetos
    - Sería un tipo de **Façade**
- Hay distintas formas de proporcionar acceso a un servicio (especialmente relevantes en sistemas distribuidos)
  - La decisión crucial es la de encapsular cierta parte del domino como servicio
  - Un simple objeto **Singleton** puede ser una implementación perfectamente válida para servicios de dominio en un sistema no distribuido, o un buen sitio para una **Façade** si el sistema es distribuido

Paquetes (*Modules*)

Los paquetes

- Como elemento de diseño están más que establecidos, pues ayudan a organizar el código y a ocultar detalles poco relevantes desde cierto nivel de abstracción
- Sin embargo, los paquetes también tienen un papel en la capa del dominio, y este papel es rara vez considerado

Paquetes (de dominio)

- Los paquetes deben tener alta cohesión conceptual y bajo acoplamiento entre ellos
- Si hay mucho acoplamiento entre objetos en distintos paquetes, quizás falte algún concepto que pueda servir para unificar esos objetos en un nuevo paquete
- Busca el acoplamiento bajo en el sentido de tener conceptos que se pueden entender (sobre los que se puede razonar) por separado
- Los paquetes deben reflejar el dominio
  - Deben corresponder a conceptos de alto nivel en este dominio y sus nombres ser parte del lenguaje compartido
  - Deben evolucionar en paralelo al resto de cambios que hagamos en el modelo del dominio

Preguntas

- ¿Qué estilo preferimos al importar unas clases del dominio desde otras?

```
import pac1.B1;
import pac1.B2;
import pac1.B3;
import pac2.C1;
...
```

```
import pac1.*;
import pac2.*;
```

Paradigmas de modelizado

¿Por qué predomina el diseño orientado a objetos?

- Buena relación entre potencia y simplicidad
  - Hay desarrolladores experimentados que aún no entienden todo su potencial, pero hasta los no expertos pueden seguir las ideas básicas en un diagrama de objetos
- Se adapta bien a muchos dominios de problema
- Desde su inicio hay lenguajes de programación orientados a objetos que permiten implementar los diseños de manera directa
- Madurez: lenguajes de programación, entornos de desarrollo, bibliotecas de software, libros, patrones, diseñadores experimentados...

Alternativas

- Hay algunos dominios de problema para los que el paradigma orientado a objetos no es la mejor opción
  - Por ejemplo los que son intensamente matemáticos, o los que están muy basados en conocimiento estructurado/formalizado
- En ellos podemos aplicar igualmente el diseño dirigido por el dominio, pero nuestros diseños podrían usar otras técnicas
  - Un sistema fuertemente basado en conocimiento podría usar lógica de predicados como diseño y un sistema basado en reglas para su implementación
- Si hay una parte relativamente pequeña de nuestro dominio que no se expresa bien con objetos, normalmente usaremos objetos y conviviremos con algunos que sean "un poco extraños"
  - Por ejemplo, sistemas que usan bases de datos relacionales, motores de reglas, motores de *workflows...*

Ejercicio

- Diseñad una interfaz y una clase base sencillas para entidades
  - Pensad en la interfaz mínima común a todas las entidades en vuestro sistema, y en qué funcionalidad podéis querer proporcionar en la clase base

Ejercicio

- Diseñad un objeto valor para representar medidas de distancias en metros en vuestro sistema
  - Cada instancia representará una distancia en metros, ni más ni menos
  - Funcionalidad: poder devolver la distancia en kilómetros, poder sumar medidas de distancia en metros, asegurarnos de que no tenemos medidas negativas y comprobar si dos distancias en metros son iguales

Ejercicio

- Tenemos una entidad de dominio para reservas de vacaciones

```
public class Reserva extends BaseEntity {
   private viajeroId, noche1, noche2
   public Reserva (int viajeroId, DateTime noche1, DateTime noche2) {
     super(); // nos dará un id único
     // atributos privados = params del constructor
   }
   // getters públicos, como mucho, para los atributos privados
}
```

- Hay varias cosas importantes que tenemos que comprobar
  - Que noche1 es una fecha anterior en el tiempo a noche2
  - Que se cumple la duración mínima. La duración mínima cambia con frecuencia, pero se puede chequear siempre en este servicio de dominio:

    ```
    boolean PolíticaDuraciónMínima.seCumple(DateTime noche1, DateTime noche2)
    ```

- Podríamos ampliar la entidad Reserva para que chequeara ella misma el cumplimiento de las políticas pero, por mantenerla lo más simple posible, preferimos delegar esta tarea en un objeto valor: diseñad un objeto valor que podamos asociar a la Reserva y donde tenga sentido chequear las políticas mencionadas

Ejercicio

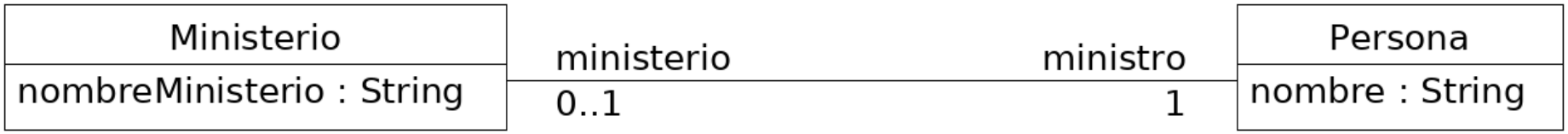
- Tenemos una entidad de dominio para personas apuntadas a un sitio de citas online, donde se calcula la compatibilidad con otras personas

```
public class PersonaSolitaria extends BaseEntity {
  private sexo, fechaNacimiento, foto, intereses, aficiones...
  public PersonaSolitaria (...) {
    super(); // nos dará un id único
    // atributos privados = params del constructor
  }
  public int tasaDeCompatibilidadCon(PersonaSolitaria other) {
    // compara los this.X con other.X calcula el índice y devuélvelo
  }
  // getters públicos, como mucho, para los atributos privados
}
```

- El cálculo que se hace en el método `tasaDeCompatibilidadCon` es complicado y no nos parece esencial en la definición de la entidad `PersonaSolitaria`. Refactorizad el código de esta entidad para extraer ese cálculo

Ejercicio

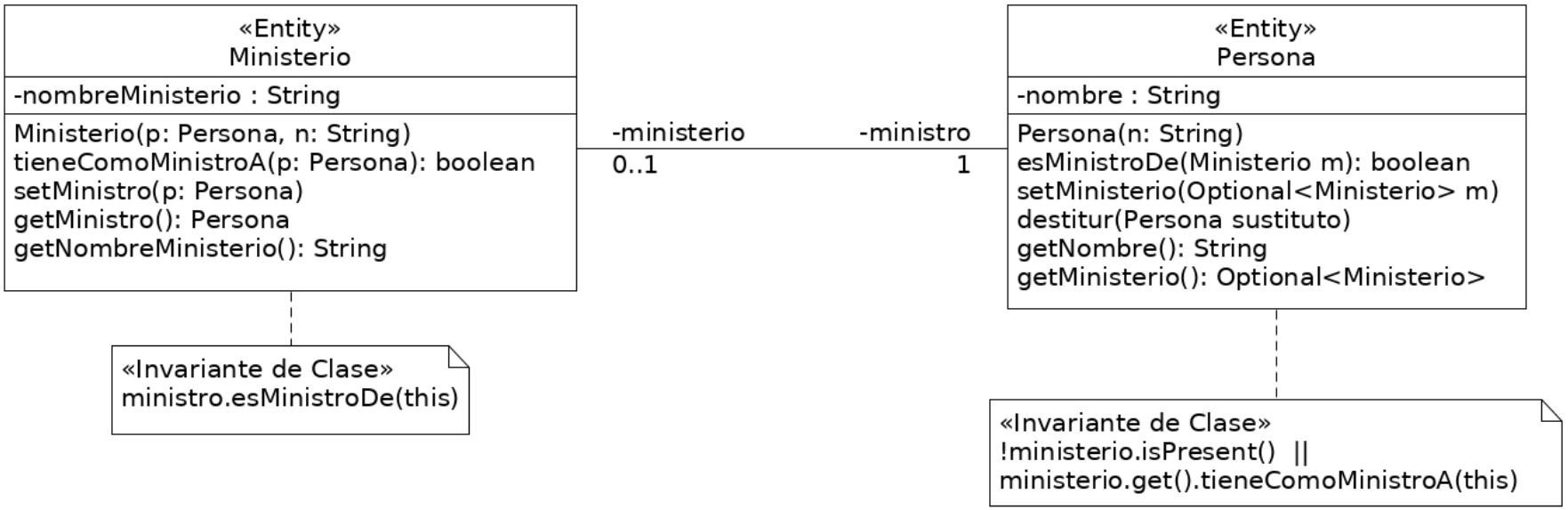
- Implementa el siguiente diagrama de clases UML en Java 8
- Luego haz lo mismo pero asumiendo que las multiplicidades son 1 en ambos lados y finalmente asumiendo que son * en ambos lados



Solución

- Tenéis un esbozo de solución en https://github.com/UNIZAR-30249-LabIS/ejercicios
  - Para discutirla y señalar temas interesantes que surgen, no necesariamente como la mejor, o mucho menos la única, alternativa
    - De hecho en algunos aspectos es más rebuscada de lo que es habitual

Comentarios de la solución 0..1 a 1

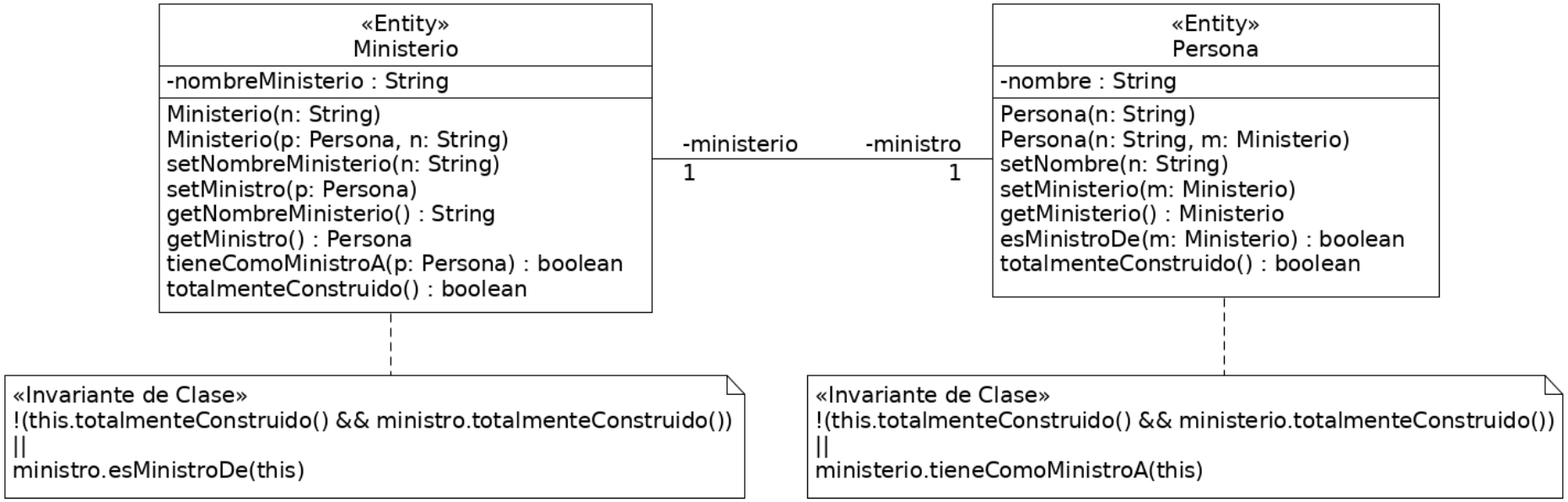


- El constructor de Ministerio requiere una Persona (lado 1) pero el de Persona no requiere un Ministerio (lado 0..1)
- El uso de Optional<> para el Ministerio de una Persona no es el esperado en Java 8 para esto, pero así lo podemos comentar
- Notad el uso de invariantes de clase (ya lo veremos mejor)
- ¿Qué pasa cuando un ministro es destituido? Dado que todo Ministerio debe tener siempre un ministro (lado 1), hemos decidido que la forma de hacerlo sea con un método destituir que exige un sustituto
  - Pero esto es un tema de Dominio y la solución dependerá de este

Comentarios de la solución 1 a 1

- Una forma sencilla de implementar una asociación bidireccional 1 a 1 es elegir una clase como dominante en la relación y otra como dominada. La dominante crea un objeto de la dominada en su propio constructor y le pasa this como parámetro al constructor de esa
  - Y luego no permitimos más cambios
- Eso puede servir con Objetos Valor, pero puede ser problemático con Entidades, que generalmente pueden cambiar durante su vida, así que en la solución hemos planteado una alternativa más complicada

Comentarios de la solución 1 a 1



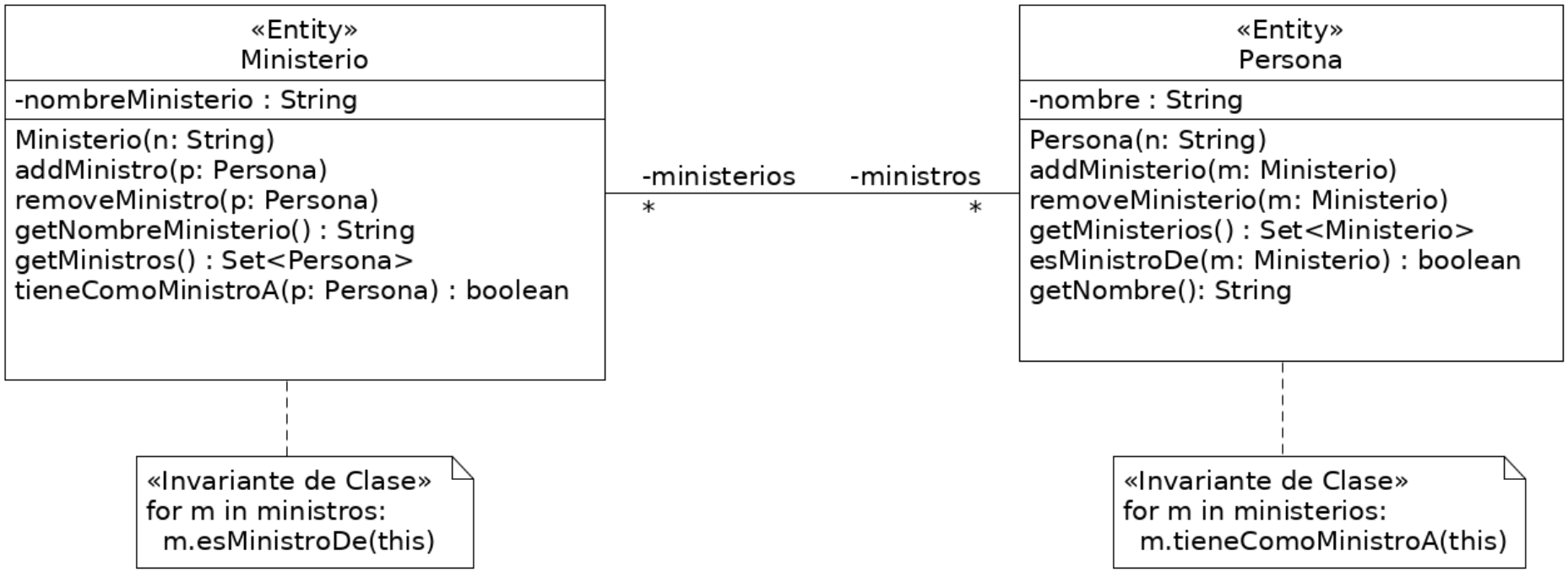
Comentarios de la solución 1 a 1

- Con entidades, el requisito 1 a 1 puede ser imposible de mantener todo el tiempo, especialmente durante la construcción de los objetos y mientras se hacen cambios. Una opción es relajarlo temporalmente:
  - El método totalmenteConstruido() nos dice si tenemos un objeto completo o no
  - Notad que el invariante de clase tiene que contemplar que el objeto es válido si no está totalmenteConstruido(), o nunca será válido
  - Aunque en el ejemplo no lo he hecho, una factoría para instanciar pares <Ministerio, Persona> sería una buena idea
- ¿Qué pasa cuando asignamos un nuevo ministro a un Ministerio? La multiplicidad 1 a 1 exige que toda Persona tenga un Ministerio, así que ¿qué hacemos con el antiguo ministro?
  - Primero, este problema es de dominio. Segundo, este tipo de cosas sugieren que las multiplicidades elegidas no son lo que realmente necesitamos.
  - En el código se nulifica el ministerio del antiguo ministro, dejando a ese ministro en estado incompleto, dado que no tenemos más información para trabajar
  - Algo similar para el caso de asignar un nuevo Ministerio a un ministro
- El código de los métodos setMinistro y setMinisterio se encarga de mantener la coherencia entre los dos objetos involucrados (el this, y el parámetro)
  - Recordad que en estos casos hay que tener cuidado de no entrar en un ciclo de llamadas infinitas entre el uno y el otro que acabe en un desbordamiento de pila

Comentarios de la solución 1 a 1

- Aunque todo se puede solucionar, mejor o peor, hay una pregunta importante: ¿de verdad necesitamos una asociación 1 a 1, bidireccional, entre dos entidades?
- Dos objetos con una asociación de ese tipo son a casi todos los efectos el mismo objeto: siempre van juntos y desde uno siempre podemos llegar al otro. ¿Por qué tenemos dos?
  - Pasa lo mismo si son objetos valor
- En general, preferiremos evitar las asociaciones bidireccionales 1 a 1, especialmente entre entidades
  - Es difícil decir "siempre", pero salvo la absoluta certeza de que en vuestro caso no hay una alternativa válida, evitadlas

Comentarios de la solución * a *



Comentarios de la solución * a *

- Lo hemos implementado con dos colecciones de tipo Set en ambos lados de la asociación
  - Esto por sí solo no es una relación * a * son dos relaciones 1 a *
    - Necesitamos hacer algo para asegurarnos de mantener la coherencia
- Cada vez que se añade o quita un elemento a esas colecciones a través de los métodos de las entidades (addMinistro(), removeMinistro(), addMinisterio(), removeMinisterio(), se modifica en el otro lado para mantener la coherencia
  - P.ej. Al hacer m.addMinistro(p), se acaba llamando a p.addMinisterio(m) si es necesario (ojo de nuevo con evitar desbordamientos de pila con esto)
- Hay una puerta trasera que hay que cerrar: p.ej. si el método getMinistros() devuelve el Set de ministros de un ministerio y el código que lo ha llamado modifica la colección, habremos roto la restricción impuesta por la cardinalidad de la asociación * a *
  - Lo que hacemos es devolver una copia defensiva del Set. Otra alternativa, Java 9+, sería trabajar con colecciones inmutables

Bibliografía

- Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*
  - Capítulos 4,5