TRABAJO CUDECA

ANEXOS:

> **Actores y Requisitos**

> **Extracción de casos de uso**

> **Sesión de desarrollo de modelado de clases**

ACTORES:

→ **CRM:** conexión con la base de datos para la web de Cudeca, la cual, almacena información de los usuarios.

→ **Usuario:** persona que compra entradas para los eventos organizados por Cudeca.

→ **Cudeca:** persona que se encarga de organizar y administrar los eventos, encontrar una ubicación, asignar precios a las entradas, etc.

→ **Pasarela de pago:** se encarga de traspasar los pagos de los usuarios.

REQUISITOS FUNCIONALES Y NO FUNCIONALES:

REQUISITOS FUNCIONALES

- RF1. El sistema debe ofrecer la venta de un número ilimitado de entradas para distintos tipos de eventos. **[pg.3 §2-3; pg.14 §1]**

- RF2. En el sistema se deben poder configurar distintos tipos de eventos (loterías, conciertos, cenas, etc). **[pg.3 §2-3]**

- RF3. El sistema debe poder recopilar los datos de los compradores de eventos de forma voluntaria, para poder generar un certificado de donaciones. **[pg.3 §4]**

- RF4. La aplicación debe poder utilizar los datos de usuarios registrados anteriormente para que estos no se tengan que volver a rellenar en cada compra. []

- RF5. El sistema ha de informar a los usuarios, que hayan previamente aceptado el envío de correos de Cudeca, de eventos próximos y otras notificaciones. **[pg.4 §1]**

- RF6. La aplicación debe permitir el uso de múltiples métodos de pagos para la compra de entradas. **[pg.2 §4]**

- RF7. El sistema debe soportar la cancelación de eventos por parte de los organizadores del mismo y que gestione la devolución de los pagos de todas las entradas. **[pg.7 §1]**

- RF8. Cada evento puede incluir una meta de recaudación, actualizando el total recaudado tanto automáticamente por la venta de entradas como de forma manual por los organizadores del evento. **[pg.9 §2]**

- RF9. El sistema debe permitir la distinción de distintos tipos de entradas para un mismo evento (correspondientes a distintas localidades en un concierto por ejemplo), con distintos precios. **[pg.15 §2]**

- RF10. El usuario debe poder pagar un extra a la hora de comprar la entrada a modo de donación. **[pg.9 §4]**

- RF11. Las entradas deben poder identificarse con un código QR que pueda permitir la gestión de entrada al evento el día de su celebración. **[pg.9 §7]**

- RF12. El sistema debe enviar recordatorios a los compradores una vez se acerque la fecha de celebración del evento, preferiblemente por SMS. **[pg.16 §5; pg.17 §1]**

- RF13. La aplicación contará con una interfaz simplificada para facilitar las gestiones de compra de entradas para personas con poca experiencia digital. **[pg.11 §4]**

- RF14. La aplicación se incorporará en la ya existente página web de Cudeca. **[pg.18 §6]**

- RF15. El sistema debe conectarse con el CRM ya existente de Cudeca. **[pg.11 §6]**

- RF16. En la aplicación se deben poder consultar eventos pasados. **[pg.14 §1]**

- RF17. Los nombres y descripciones de los eventos deben traducirse al inglés automáticamente. **[pg.13 §4]**

- RF18. Las aplicación debe permitir múltiples métodos de pago para la compra de entradas. **[pg.2 §4]**

REQUISITOS NO FUNCIONALES

- RNF1. El sistema debe cumplir la ley de protección de datos. **[pg.3 §2]**

EXTRACCIÓN CASOS DE USO:

Posteriormente, mostraremos los casos de uso que hemos usado en nuestra aplicación:

· **CU01** (Usuario)←Alta usuario: el usuario se da de alta en la web de Cudeca como socio. Sus datos se almacenan en su base de datos.

· **CU02** (Usuario)←Donar extra: además de comprar las entradas, los usuarios tienen la posibilidad de donar un dinero extra.

· **CU03** (Usuario)← Comprar entrada: los usuarios pueden comprar todas las entradas que quieren para los eventos de la fundación.

· **CU04** (Cudeca)← Enviar entrada: el administrador de Cudeca se encarga de enviar la entrada al correo electrónico del usuario.

· **CU05** (Cudeca)← Recordatorio evento: Cudeca enviará un recordatorio días antes de la celebración del evento a los asistentes.

· **CU06** (Cudeca)← Crear evento: los miembros de la fundación se encargarán de crear el evento, así como, de elegir el tipo de evento, su ubicación, el precio de las entradas, etc.

· **CU07** (Cudeca)← Enviar Spam: si el usuario está dado de alta, este podrá ser informado de próximos eventos similares.

· **CU08** (Usuario)← Rellenar datos: el usuario debe rellenar sus datos para poder comprar entradas.

· **CU09** (Usuario)← Rellenar datos adicionales: el comprador podrá rellenar algunos datos adicionales, con la posibilidad de registrarse en la página web de Cudeca.

· **CU10** (Pasarela de pago)← Realizar pago: para comprar las entradas el usuario deberá realizar el pago mediante la pasarela de pago.

· **CU11** (CRM)← Enviar datos: los datos del usuario se almacenan en la base de datos de Cudeca.

· **CU12** (CRM) ← Envía certificados: la base de datos de Cudeca se encarga de enviar los certificados de donación a los usuarios donantes.

En esta sección explicaremos nuestras discusiones y decisiones sobre los aspectos de nuestro diagrama.

En primer lugar, el proceso de comprar una entrada (CU03) conlleva diversas acciones:

El donar dinero (CU02) de manera opcional (extends) y otras acciones ligadas al proceso de compra como son el rellenar (CU08) y enviar datos (CU11), realizar el pago de la entrada o enviar la entrada a Cudeca. Además se podrán rellenar datos adicionales (CU09) (extends) para poder registrarse en la web de Cudeca. Esto conlleva darse de alta como usuario (CU01) en la web de Cudeca, para poder almacenar los datos en la base de datos de Cudeca y posteriormente verse reflejados en la compra de entradas.

Por otro lado, en referencia a Cudeca decidimos añadir los siguientes casos de uso:

- La opción de recordar el evento a los diferentes usuarios que compraron entradas. (CU05)

- La posibilidad de crear un evento. (CU06)

- La posibilidad de enviar Spam (CU07) fue motivo de discusión, debido a que el Spam conlleva a que el usuario este dado de alta. No sabíamos si ponerlo como una dependencia o simplemente como una etiqueta de precondición. Como resultado, hemos decidido la segunda opción, ya que lo considerábamos más legible y correcto.

Para continuar, decidimos crear un actor que se encargase del pago externo de las entradas (include) (CU10)

Decidimos crear un actor que representase la conexión con la base de datos de Cudeca, la cual se encargará de recibir los datos enviados (CU11) por nuestra aplicación y de emitir los certificados de donación (CU12).

Establecimos que el usuario podría aportar ingresos extra desde la propia aplicación de venta de entradas, recibiendo posteriormente su certificado de donación. Sin embargo, también podrá hacerlo sin necesidad de comprar entradas a eventos.

Los usuarios pueden comprar todas las entradas que quieran sin ningún límite, al contrario, que en Entradium, que te limita a diez entradas máximo.

Decidimos que el administrador de Cudeca se encargaría de enviar las entradas a los usuarios para poder acceder al evento, de gestionarlos y administrarlos y de enviar Spam a los usuarios con eventos de gustos similares.

Las entradas serán nominativas para cada comprador.

EXTRACCIÓN MODELADOS DE CLASES:

Una parte sustancial de la arquitectura y planificación del proyecto es el modelado de las clases y paquetes del sistema, en lo que se conoce como diagrama de clases. En este Diagrama quedan recogidas todas las clases (dentro del paradigma de la programación orientada a objetos) así como sus atributos, su funcionalidad y las relaciones entre ellas.

De esta forma conseguimos una documentación viva y muy visual que permite una evolución progresiva del proyecto gracias a su escalabilidad así como una forma sencilla de adaptar a nuevos integrantes del proyecto en éste, pudiendo entender de una vista rápida la estructura y funcionamiento del diseño.

Paquetes

En primer lugar, para facilitar la comprensión y el trabajo, decidimos dividir el proyecto en una serie de paquetes de clases, lógicamente independientes aunque luego puedan estar relacionadas. Así facilitamos el trabajo grupal y la separación de tareas, además de añadir una capa más de significado al diagrama.

Los principales paquetes detectados son:

- Usuarios: en este paquete está recogida toda la información acerca de los usuarios y clientes que visiten la página, incluidos aquellos que estén registrados o sean socios.

- Eventos: Recoge todos los tipos de eventos así como las diferentes entradas que podemos encontrar para cada uno de ellos.

- Tickets: Una especie de paquete intermedio entre usuarios y eventos, recoge la información de los tickets de entradas que los usuarios han comprado.

Clases y relaciones

Una vez identificados los paquetes en los que dividiremos la aplicación, definimos su composición interna, es decir, las clases que los componen.

Paquete Usuario:

Dentro de este paquete definimos los tipos de usuario que existen así como los atributos y funciones que debe tener cada uno.

En primer lugar, siempre es buena práctica definir una interfaz y una clase abstracta que nos defina aquello que tienen en común todos los usuarios. Por ello creamos la interfaz Usuario, que usarán todas las clases que quieran implementar alguna relación con las clases que hereden de esta.

Para definir aquellos atributos que son comunes, creamos la clase *Usuario_Base*, que implementa la interfaz Usuario y contiene los atributos mínimos que debe tener un usuario.

Estos son *nombre*, de tipo string; *email*, de tipo string; *dni*, de tipo string y *spam*, de tipo bool. Dado que un usuario puede ser completamente definido por estos atributos (aquellos que no decidan rellenar los datos adicionales [requisito funcional]), no se trata de una clase abstracta sino de una clase normal.

Por otra parte, necesitamos una forma de salvar la información de aquellos usuarios que sean socios o que estén registrados para poder recibir los certificados de donaciones ([requisito funcional]). Para ello creamos la clase *Usuario_Registrado*, que extiende a *Usuario_Base* y que además de los datos ya mencionados contendrá *direccion*, de tipo string; *telefono*, de tipo integer y *socio*, de tipo bool. Con estos atributos obtenemos los datos necesarios para generar los certificados de donaciones y podemos comprobar si un usuario es además socio, es decir que paga la cuota mensual de socio. Pese a existir la posibilidad de crear una clase adicional *Usuario_Socio* para expresar esta funcionalidad, como equipo hemos preferido mantener la simplicidad de una sola clase y definirlo a través del atributo *socio* dado que los usuarios socio no tienen atributos adicionales que los registrados; simplemente son socios. En el caso de que existiesen diferentes planes de socio se consideraría la opción de crear dicha clase adicional puesto que habría que recoger dicho plan como un atributo extra de los socios.

Paquete Eventos:

Este es el paquete *core* de la aplicación, en el sentido de que es el más grande, técnico, importante y repleto de decisiones de desarrollo importantes. En éste se refleja la estructura de los eventos y tipos de eventos que existen, los tipos de

entrada que están relacionadas con los eventos y en general cuestiones de manejo de eventos.

Empezamos definiendo primero la estructura de los eventos en sí mísmos. Seguimos una estructura similar a la empleada con los usuarios, creando una interfaz *IEvento* en la que se recoge la funcionalidad más básica y la clase *Evento_abstracto* con los atributos genéricos, que son *nombre*, de tipo string; *recaudacion*, de tipo integer que representa la cantidad recaudada y *objetivo_recaudacion*, de tipo integer que representa el objetivo a conseguir. También incluirá una lista de patrocinadores ([requisito funcional]) y de Tags para categorizar los eventos. Estos serán usados de cara a enviar publicidad específica a usuarios que han asistido a ciertos eventos ([requisito funcional]). Estos tipos pueden ser tanto clases con atributos de tipo string o pueden ser enums para simplificar su uso. Cabe destacar que, a diferencia de en el caso de la clase *Usuario_Base*, hablamos de una clase por definición abstracta, puesto que no existe ningún evento que sea suficiente por sí mismo con estos atributos.

Tras haber definido la raíz, creamos 3 clases diferentes, una para cada tipo de evento:

- Evento_Concierto: Clase creada para definir los eventos de tipo concierto, en estos existen los atributos específicos de *artista*, de tipo string; *ubicación*, de tipo string y *fecha*, de tipo string o Date.

- Evento_Carrera: Con esta clase se recoge la información de aquellos eventos de tipo carrera, cuyos atributos son *ubicación*, de tipo string y *recorrido* de tipo integer.

- Evento_Rifa: Los eventos rifa son eventos un tanto diferentes, puesto que a diferencia de los demás no son presenciales. Estos eventos son un sorteo por lo que sus atributos son *premios*, de tipo lista de premios que pueden ser económicos o de otro tipo (se puede definir con posterioridad puesto que no tiene ninguna otra relación con el resto del sistema) y *fecha*, de tipo string o Date.

Esta estructura de clases, como se verá más adelante, se va a repetir en reiteradas ocasiones. Hemos tomado la decisión de seguir este tipo de jerarquía debido a varios motivos. En primer lugar nos permite gran escalabilidad y

adaptabilidad de cara a añadir nuevos formatos de evento, creando simplemente una nueva clase Evento_TipoDeEvento. Gracias a la interfaz *IEvento* no es necesario cambiar el código del resto de secciones del programa.

Además de esta ventaja de compatibilidad, es una arquitectura muy simple e intuitiva, fácil de entender para nuevos desarrolladores del equipo y lo suficientemente compleja como para cumplir todos los requisitos del cliente e incluso posibles futuros requisitos (nuevos eventos de otro tipo, que requieran otros atributos diferentes o nuevas funcionalidades).

Una vez definidos los eventos vamos a definir los tipos de entrada. El concepto de entrada surge del debate alrededor de la necesidad del cliente de disponer de distintos precios para distintas zonas de un concierto ([requisito funcional]). Inicialmente parecía lógico indicar que el precio fuera un atributo de *Evento_Abstracto*, puesto que resulta natural hablar del precio de un evento. Sin embargo, el hecho de que existan diferentes precios para un mismo evento nos lleva a necesitar definir una arquitectura que no solo soporte la idea inicial sino que además pueda llevar a cabo esta nueva necesidad. Para ello creamos la clase entrada. Esencialmente cada *IEvento* va a tener una serie de tipos de entrada diferentes (por ejemplo entrada primera fila, segunda fila; en una rifa puede haber distintos boletos con distinto precio y posibilidades de ganar un premio; dentro de una cena puede haber diferentes entradas para diferentes menús). Estos tipos de entradas van a estar organizados en la jerarquía de clases de entradas que describimos a continuación.

La estructura es exactamente igual que la comentada para los eventos: una interfaz *IEntrada* con la funcionalidad más básica y una clase abstracta *Entrada_Abstracta*. Esta clase va a tener los atributos *precio*, de tipo integer; el atributo *sub_aforo*, de tipo integer, para poder definir cuántas entradas de ese tipo se pueden vender y *evento*, de tipo *IEvento*, que indica el evento al que da acceso la entrada. Es importante notar que en el desarrollo se deberá tener la precaución de que la suma de los subaforos de los tipos de entrada sea igual al aforo del evento al que hacen referencia. Otra posible solución sería eliminar el atributo *aforo* de la clase *Evento_Abstracto*, sin embargo hemos tomado la decisión de mantenerlo y tener las precauciones pertinentes por simplicidad.

En el árbol jerárquico aparecen entonces las clases *Entrada_Concierto*, *Entrada_Carrera* y *Entrada_Rifa*. En cuanto a las entradas de concierto, el único atributo adicional es *fila*, de tipo integer que indica la fila del concierto correspondiente a ese precio para el evento correspondiente. Las otras dos

clases definidas están vacías. Si bien antes hemos comentado el caso del usuario socio, cuya clase preferimos no implementar dado que no poseía atributos específicos; para esta ocasión con problemática similar hemos tomado la decisión contraria y hemos mantenido las clases. Esto es así más que nada por un valor semántico y arquitectónico de la aplicación, de cara a mantener la congruencia con la jerarquía de los eventos y de los tickets que veremos con posterioridad. Además de ello, consideramos que es mucho más probable que eventualmente se requieran atributos específicos para entradas de tipo rifa o carrera, como los comentados anteriormente para permitir boletos con más posibilidades de ganar o posiciones adelantadas en la carrera.

Paquete Tickets:

Finalmente encontramos el paquete referente a los tickets. Como hemos comentado con anterioridad, este es el nexo que une los paquetes de usuarios y eventos, define lo que comúnmente llamamos entrada a un evento (el nombre entrada ya ha sido usado para definir los tipos de entrada de un evento, de ahí la elección de este otro nombre). Estos tickets recogen la información que es enviada al usuario para confirmar la compra de la entrada y permitir el acceso al evento.

Siguiendo el estilo estructural previo, tenemos una interfaz *ITicket* y una clase genérica *Ticket* con los atributos comunes de todos los tipos de ticket, a entender *id*, tipo integer, es un identificador (potencialmente convertible a qr); *comprador*, de tipo *IUsuario*; *evento*, de tipo *IEvento*; *nombre_beneficiario*, de tipo string y *dni_beneficiario*, de tipo string. Estos dos últimos atributos fueron una fuente de argumentación dentro del equipo.

Originalmente, se planeó la clase Ticket sin contar con estos atributos, suponiendo que el atributo de tipo *IUsuario* contendría la información del asistente del evento. Sin embargo releyendo la transcripción, vimos que el cliente propone la idea de que una persona pueda comprar varias entradas y poner los datos de otras personas en ellas (ejemplo típico de cena de empresa o carrera de empresa en la que el jefe compra las entradas pero pone los datos de los empleados que asisten [requisito funcional]). Por ello decidimos usar un esquema en el que se diferencia comprador y beneficiario, en el que el comprador es quien recibe la información de la donación pero puede existir un beneficiario ajeno, identificado con nombre y dni, que es el asistente al evento.

Se llegó a plantear la idea de que el beneficiario fuese también de tipo *IUsuario*, puesto que muchas veces comprador y beneficiario son la misma persona. Sin embargo consideramos innecesario que el beneficiario tenga que entregar el dato de correo electrónico (que es necesario para formar un usuario), por ello y para evitar duplicidad de los datos en el caso de que comprador y beneficiario son la misma persona, se indicará dejando *nombre_beneficiario* y *dni_beneficiario* como valores nulos o vacíos, de forma que se supondrá que los datos del beneficiario son los del comprador y se obtendrán los datos del atributo *comprador*.

Completando la jerarquía genérica, tenemos las clases *Ticket_Concierto*, *Ticket_Carrera* y *Ticket_Rifa. Ticket_Concierto* tiene el atributo *asiento*, de tipo integer; que indicará, en caso de existir varios asientos para ese tipo de entrada, cuál de ellos le corresponde.

Ticket_Carrera contiene el atributo *dorsal*, de tipo integer; que indica el dorsal que corresponde para la carrera. Finalmente *Ticket_Rifa* que contiene el identificador numérico *n_boleto* ([requisito funcional]) para el sorteo. Pese a poder usar el atributo *id* de la clase padre, preferimos incluir este atributo en la clase *Ticket_Rifa* para poder repetir números. Los identificadores de la clase padre deberán ser únicos, para poder ser guardados correctamente en la base de datos. Con este atributo adicional podemos permitir que haya dos tickets de rifa con el mismo número de dos eventos de rifa distintos. Será necesario entonces desde el punto de vista del programador asegurar que no haya duplicidad de asientos o números de boleto dentro de un mismo evento.

Otras arquitecturas planteadas

Durante el desarrollo del modelo, se propuso otra posible arquitectura que creemos que merece la pena comentar. El planteamiento venía a sugerir que dentro de nuestra clase *Evento_Abstracto* hubiese una lista con todas las entradas del evento individuales, de forma que las clases de *IEntrada* no representarían tipos de entrada sino entradas en sí mismas. Además de esta forma la clase Ticket no tendría por qué incluir la compleja jerarquía y se limitaría a indicar la entrada a la que hace referencia y el usuario que la compra.

Esta idea se descartó por el simple motivo de que existen eventos que pueden tener entradas “infinitas”, o por lo menos no determinadas. Ello implicaría una indeterminación en el tamaño de la lista de entradas de la clase representante del evento que se debería resolver con un tamaño arbitrario. Además sería

necesario alojar espacio para todas y cada una de las entradas del evento, hayan sido o no compradas, cada vez que se quiera crear el objeto de tipo *Evento_Abstracto*. Por ello optamos por esta otra implementación que, aunque es más compleja y requiere de más cuidado en la implementación por parte del programador, es más óptima y legible.

FOTOS DE LOS DIAGRAMAS ADJUNTOS: