



ESCUELA UNIVERSITARIA DE INGENIERÍA TÉCNICA INDUSTRIAL DE BILBAO



GRADO EN INGENIERÍA INFORMÁTICA DE GESTIÓN Y SISTEMAS DE INFORMACIÓN

TRABAJO FIN DE GRADO

2014 / 2015

COLECCIONAPP

MEMORIA

DATOS DE LA ALUMNA O DEL ALUMNO

NOMBRE: IVÁN

APELLIDOS: REVUELTA ANTIZAR

FDO.:

FECHA:

DATOS DEL DIRECTOR O DE LA DIRECTORA

NOMBRE: MIKEL

APELLIDOS: VILLAMAÑE GIRONÉS

DEPARTAMENTO: LENGUAJES Y SISTEMAS INFORMÁTICOS

FDO.:

FECHA:

Índice de contenidos

1.	Introducción	13
2	Planteamiento inicial	17
2.1	Objetivos.....	17
2.2	Arquitectura	17
2.3	Alcance	18
2.4	Planificación temporal	29
2.5	Herramientas	31
2.6	Gestión de riesgos.....	32
2.7	Evaluación económica	41
3	Antecedentes	45
4	Captura de requisitos.....	47
4.1	Modelo de casos de uso	47
4.2	Casos de uso extendidos (Anexo).....	49
4.3	Modelo de dominio.....	50
5	Análisis y Diseño.....	53
5.1	Transformación del Modelo de dominio a BBDD.....	53
5.2	Diagrama de clases	54
5.2.1	Aplicación del administrador	55
5.2.2	Aplicación del usuario.....	57
5.3	Diagramas de secuencia (Anexo)	63
6	Desarrollo.....	65
6.1.	Base de datos	65
6.2	Administrador.....	67
6.2.1.	Estructura y componentes.....	67
6.2.2.	Conexión con la base de datos.....	70
6.2.3	Tratamiento de tablas.....	72
6.2.4	Explorador de ficheros y validación.....	75
6.2.5	Tratamiento de imágenes	77
6.2.6	Tratamiento de datos	
6.3	Usuario	81
6.3.1.	Introducción	81
6.3.2.	Frameworks.....	82
6.3.3.	Main.storyboard	84
6.3.4.	Estructura y creación de las vistas en ColeccionApp.....	87
6.3.5.	Base de datos SQLite.....	91
6.3.6.	Php, Json y Parseo de datos	92
6.3.6.	UITableView	95
6.3.7.	Animaciones y paso de parámetros entre controladores	98
6.3.8.	UITabBarController	100
6.3.9.	IUCollectionView	101
6.3.10.	AlertView	104
6.3.11.	CMAcceleration, CMMotionManager	105
6.3.12.	SupportedInterfaceOrientations.....	109

6.3.13. CoreLocation y Mapkit.....	111
7 Validación y pruebas	113
7.1. Administrador.....	113
7.1.1. Pruebas de interfaz	113
7.1.2. Pruebas funcionales.....	113
7.2. Usuario	119
7.2.1. Pruebas de interfaz	119
7.2.2. Pruebas funcionales.....	119
7.2.3. Pruebas de rendimiento	127
8 Conclusiones.....	129
9 Trabajo futuro	133
10 Bibliografía.....	135
10.1 Libros.....	135
10.2 Sitios Web	135
ANEXO I.....	137
Casos de uso	137
Administrador.....	139
Usuario	168
ANEXO II Diagramas de secuencia.....	189
Administrador.....	191
Identificarse	191
cargarColecciones – Controlador	193
cargarFamilias – Controlador	194
cargarTodosLosCromos() – Controlador	195
Agregar colección	196
Modificar colección.....	197
Eliminar colección.....	199
Cromos (Gestionar una colección).....	200
Agregar cromos – Cromos (Gestionar una colección).....	201
Modificar cromos – Cromos (Gestionar una colección)	203
Eliminar cromos – Cromos (Gestionar una colección)	206
Traer cromos – Cromos (Gestionar una colección).....	207
Gestionar familias – Cromos (Gestionar una colección).....	210
LlevarFamilia – GestionarFamilias.....	212
traerFamilia – GestionarFamilias	214
Agregar Familia – GestionarFamilias.....	215
Modificar Familia – GestionarFamilias.....	217
Eliminar Familia – GestionarFamilias.....	218
Usuario	219
Registrarse.....	219
Identificarse	221
Acceder al catálogo completo	224
Listar tus colecciones.....	229
Acceder a una colección	230

Afrontar un reto	233
Batalla.....	237
Obtener cromo de un usuario (tras batalla).....	245
Intercambio	251
Proceso de intercambio de cromos.....	259
Buscar tesoro.....	265

Índice de figuras

FIGURA 1. ARQUITECTURA.....	17
FIGURA 2. CICLO DE VIDA INCREMENTAL.....	18
FIGURA 3. ESTRUCTURA DE DESCOMPOSICIÓN DEL TRABAJO	20
FIGURA 4. DIAGRAMA GANTT SEMANAL	29
FIGURA 5. RECUPERACIÓN DE LA INVERSIÓN MEDIANTE BANNERS	44
FIGURA 6. CÁLCULO DE BENEFICIOS MEDIANTE ITEM-SELLING	44
FIGURA 7. EL EQUIPAZO VIRTUAL	45
FIGURA 8. WAR OF THE FALLEN	45
FIGURA 9. CLASH OF CLANS	46
FIGURA 10. CSR RACING	46
FIGURA 11. CANDY CRASH	46
FIGURA 12. CASOS DE USO DEL ADMINISTRADOR.....	47
FIGURA 13. CASOS DE USO DEL USUARIO	48
FIGURA 14. JERARQUÍA DE ACTORES	48
FIGURA 15. MODELO DE DOMINIO	50
FIGURA 16. DISEÑO DE BASE DE DATOS.....	54
FIGURA 17. DIAGRAMA DE CLASES DEL ADMINISTRADOR.....	55
FIGURA 18. DIAGRAMA DE CLASES DEL USUARIO.....	58
FIGURA 19. JAVA.....	67
FIGURA 20. ECLIPSE.....	67
FIGURA 21. MODELO HIBRIDO. HTTP://WWW.GLIMFY.COM/	67
FIGURA 22. COMPONENTES JAVA SWING	68
FIGURA 23. SUBCLASES AWTEVENT,.....	69
FIGURA 24. VISTA IDENTIFICARSE	70
FIGURA 25. PATRÓN SINGLETON	71
FIGURA 26. JDBC (JAVA DATABASE CONNECTIVITY)	71
FIGURA 27. RELACIÓN MODELO -> VISTA	73
FIGURA 28. TABLA COLECCIONES.....	73
FIGURA 29. VALIDACIÓN DE LOS CAMPOS.....	76
FIGURA 30. XCODE & OBJECTIVE-C.....	81
FIGURA 31. TIPOS DE DISPOSITIVOS	82
FIGURA 32. SIMULADOR IOS.....	82
FIGURA 33. LAS CAPAS DE LOS FRAMEWORKS IOS.....	83
FIGURA 34. IUNAVIGATIONCONTROLLER Y UIVIEWCONTROLLER.....	84
FIGURA 35. TRANSICIONES ENTRE VISTAS	85
FIGURA 36. MAIN.STORYBOARD	86
FIGURA 37. ESTRUCTURA DE LA INTERFAZ DE XCODE	87
FIGURA 38. NUEVA CLASE COCOA TOUCH.....	88
FIGURA 39. SUBCLASE.....	88
FIGURA 40. ASOCIAR VISTA CON CLASE CONTROLADORA.....	89
FIGURA 41. EVENTOS ENVIADOS	89
FIGURA 42. ASSISTANT EDITOR.....	90
FIGURA 43. TABLE VIEW CONTROLLER.....	95
FIGURA 44. VIEW CONTROLLER.....	96
FIGURA 45. TABLE VEW Y TABLE VIEW CELL.....	96
FIGURA 46. ESQUELETO DE LA VISTA	96
FIGURA 47. IDENTIFICADOR DE LA CELDA.....	97
FIGURA 48. TAB BAR CONTROLLER	100
FIGURA 49. UITABBARCONTROLER	100

FIGURA 50. COLLECTION VIEW Y COLLECTION VIEW CELL	101
FIGURA 51. COLLECTION REUSABLE VIEW	102
FIGURA 52. COLISIONES CON MUROS.....	109
FIGURA 53. PROPIEDADES DEL PROYECTO.....	109
FIGURA 54. COLECCIONAPP-INFO.PLIST.....	110
FIGURA 55. ERROR AL MOSTRAR LAS COLECCIONES.....	114
FIGURA 56. IDENTIFICADOR DE LA CELDA.....	120
FIGURA 57. COLECCIONAPP-INFO.PLIST	127
FIGURA 58. INSTRUMENTS	127
FIGURA 59. GRAFICO DE COMPARATIVA	131

Índice de tablas

TABLA 1. ELABORACIÓN DEL DOCUMENTO DE OBJETIVOS DEL PROYECTO.....	21
TABLA 2. ELABORACIÓN DE LA MEMORIA DEL TRABAJO FIN DE GRADO	21
TABLA 3. DIAGRAMA DE CASOS DE USO.....	22
TABLA 4. MODELO DE DOMINIO	22
TABLA 5. CASOS DE USO EXTENDIDOS.....	23
TABLA 6. TRANSFORMACIÓN DEL MODELO DE DOMINIO A BASE DE DATOS.....	23
TABLA 7. GESTIONAR LAS COLECCIONES.....	24
TABLA 8. IDENTIFICACIÓN Y REGISTRO DE USUARIO	24
TABLA 9. ACCEDER AL CATÁLOGO COMPLETO	25
TABLA 10. LISTAR COLECCIONES DEL USUARIO	25
TABLA 11. ACCEDER A UNA COLECCIÓN	26
TABLA 12. ACCEDER A LOS CROMOS.....	26
TABLA 13. AFRONTAR UN RETO CONTRA LA MÁQUINA	27
TABLA 14. INTERCAMBIAR CROMOS.....	27
TABLA 15. PROVOCAR BATALLA.....	28
TABLA 16. BUSCAR TESORO.....	28
TABLA 17. DURACIÓN TOTAL DE LAS TAREAS	30
TABLA 18. PERIODO DE REALIZACIÓN DE LAS TAREAS	31
TABLA 19. PROBABILIDAD	33
TABLA 20. IMPACTO	33
TABLA 21. RIESGOS	34
TABLA 22. CAMBIO EN EL FIRMWARE DE IOS	34
TABLA 23. CARENCIA DE UNA API NECESARIA PARA EL DESARROLLO DE ALGUNA FUNCIONALIDAD.....	35
TABLA 24. PLANIFICACIÓN TEMPORAL INCORRECTA.....	35
TABLA 25. CAMBIO DE LAS ESPECIFICACIONES DEL PROYECTO	35
TABLA 26. INDISPOSICIÓN TEMPORAL DEL DESARROLLADOR.....	36
TABLA 27. RIESGOS DE HARDWARE Y SOFTWARE	37
TABLA 28. DESCRIPCIÓN DE ALGUNA LIBRERÍA UTILIZADA.....	38
TABLA 29. MAYOR NÚMERO DE USUARIOS DE LO PLANIFICA	38
TABLA 30. RIESGOS DE SEGURIDAD EN LA BASE DE DATOS.....	39
TABLA 31. VALORACIONES.....	40
TABLA 32. VALORACIÓN DE LOS RIESGOS DE DESARROLLO	40
TABLA 33. VALORACIÓN DE LOS RIESGOS DE PRODUCTO GENERADO	41
TABLA 34. COSTE DE LAS LICENCIAS DE SOFTWARE	42
TABLA 35. COSTE DEL EQUIPO.....	42
TABLA 36. COSTE DEL MÓVIL.....	42
TABLA 37. COSTE DE DESARROLLO	42
TABLA 38. GATOS TOTALES DEL PROYECTO.	43
TABLA 39. COMPONENTES INTERFAZ GRÁFICA.....	69
TABLA 40. AGREGAR COLECCIÓN	115
TABLA 41. MODIFICAR COLECCIÓN.....	115
TABLA 42. ELIMINAR COLECCIÓN	116
TABLA 43. AGREGAR CROMO	116
TABLA 44. MODIFICAR CROMO.....	117
TABLA 45. ELIMINAR CROMO.....	118
TABLA 46. TRAER CROMO	118
TABLA 47. GESTIONAR FAMILIAS.....	118
TABLA 48. REGISTRASE	119
TABLA 49. IDENTIFICARSE	119

TABLA 50. ACCEDER AL CATALOGO COMPLETO	120
TABLA 51. LISTAR TUS COLECCIONES	121
TABLA 52. ACCEDER A UNA COLECCIÓN	121
TABLA 53. RETO DE MEMORIA	122
TABLA 54. RETO DE HABILIDAD.....	123
TABLA 55. BATALLA.....	124
TABLA 56. INTERCAMBIO	125
TABLA 57. TESORO	126
TABLA 58. COMPARACIÓN DE LOS TIEMPOS	130

Índice de imágenes

IMAGEN 1. VISTA INICIAL.....	139
IMAGEN 2. DATOS DE ACCESO NO VÁLIDOS.....	140
IMAGEN 3. LISTADO DE TODAS LAS COLECCIONES.....	141
IMAGEN 4. VISTA CREAR COLECCIÓN.....	143
IMAGEN 5. DATOS DE LA COLECCIÓN NO VÁLIDOS.....	143
IMAGEN 6. NOMBRE DE COLECCIÓN YA EXISTENTE.....	144
IMAGEN 7. CONFIRMACIÓN PARA CREAR COLECCIÓN.....	144
IMAGEN 8. GESTIONAR UNA COLECCIÓN.....	145
IMAGEN 9. CAMBIAR EL NÚMERO DEL CROMO.....	145
IMAGEN 10. APROVECHAR CROMO EXISTENTE.....	146
IMAGEN 11. MODIFICAR COLECCIÓN.....	148
IMAGEN 12. CONFIRMAR MODIFICACIÓN.....	148
IMAGEN 13. CONFIRMAR ELIMINACIÓN.....	149
IMAGEN 14. AGREGAR CROMO.....	150
IMAGEN 15. DATOS DEL CROMO VÁLIDOS.....	151
IMAGEN 16. CROMO AÑADIDO A LA COLECCIÓN LIGA BBVA 2013 CORRECTAMENTE.....	151
IMAGEN 17. DATOS DEL CROMO NO VÁLIDOS.....	152
IMAGEN 18. MODIFICAR CROMO.....	153
IMAGEN 19. MODIFICAR CROMO, DATOS VÁLIDOS.....	154
IMAGEN 20. CROMO MODIFICADO CORRECTAMENTE.....	154
IMAGEN 21. MODIFICAR CROMO, DATOS NO VÁLIDOS.....	155
IMAGEN 22. ELIMINAR CROMO.....	156
IMAGEN 23. GESTIONAR FAMILIAS.....	158
IMAGEN 24. FAMILIA EN USO POR ALGÚN CROMO DE ESTA COLECCIÓN.....	159
IMAGEN 25. AGREGAR FAMILIA EXISTENTE.....	159
IMAGEN 26. AGREGAR UNA FAMILIA QUE NO EXISTE.....	159
IMAGEN 27. AGREGAR FAMILIA, MENSAJE DE CONFIRMACIÓN.....	160
IMAGEN 28. FAMILIA AGREGADA CORRECTAMENTE.....	160
IMAGEN 29. INTENTANDO AGREGAR UNA FAMILIA QUE YA EXISTE.....	161
IMAGEN 30. MODIFICAR FAMILIA. ADVERTENCIA.....	161
IMAGEN 31. MODIFICAR FAMILIA, CUADRO DE TEXTO.....	162
IMAGEN 32. MODIFICAR FAMILIA. CONFIRMACIÓN.....	162
IMAGEN 33. FAMILIA MODIFICADA CORRECTAMENTE.....	163
IMAGEN 34. MODIFICAR FAMILIA. ERROR, LA FAMILIA YA EXISTE.....	163
IMAGEN 35. ELIMINAR FAMILIA. CONFIRMACIÓN.....	164
IMAGEN 36. ELIMINAR FAMILIA. ERROR, FAMILIA EN USO POR ALGUNA COLECCIÓN.....	164
IMAGEN 37. CROMO YA AGREGADO.....	165
IMAGEN 38. NÚMERO DE CROMO OCUPADO.....	166
IMAGEN 39. CONFIRMAR CROMO.....	166
IMAGEN 40. CROMO AGREGADO CORRECTAMENTE.....	167
IMAGEN 41. VISTA INICIO.....	168
IMAGEN 42. DATOS USUARIO.....	168
IMAGEN 43. CAMPOS VACÍOS.....	169
IMAGEN 44. DATOS NO VÁLIDOS.....	170
IMAGEN 45. CAMPOS VACÍOS.....	170
IMAGEN 46. MIS COLECCIONES.....	171
IMAGEN 47. COLECCIONES DISPONIBLES.....	172
IMAGEN 48. EMPEZAR UNA COLECCIÓN.....	172
IMAGEN 49. ABRIR SOBRE.....	173

IMAGEN 50. RETO DE MEMORIA.....	175
IMAGEN 51. INICIAR RETO ALEATORIO	175
IMAGEN 52. RETO DE MEMORIA SUPERADO.....	175
IMAGEN 53. RETO DE MEMORIA NO SUPERADO.....	175
IMAGEN 54. RETO DE HABILIDAD SUPERADO	176
IMAGEN 55. RETO DE HABILIDAD NO SUPERADO	176
IMAGEN 56. ABANDONAR	180
IMAGEN 57. PROPUESTAS DE BATALLA.....	180
IMAGEN 58. PROPUESTAS RECIBIDAS.....	180
IMAGEN 59. BATALLA GANADA.....	180
IMAGEN 60. BATALLA PERDIDA.....	181
IMAGEN 61. BATALLA SIN DETERMINAR AÚN.....	181
IMAGEN 62. BATALLA SELECTIVA GANADA	181
IMAGEN 63. BATALLA ALEATORIA GANADA.....	181
IMAGEN 64. ANULAR BATALLA.....	182
IMAGEN 65. ELEGIR TIPO DE BATALLA.....	182
IMAGEN 66. LISTA DE USUARIOS PARA BATALLA	182
IMAGEN 67. EMPEZAR BATALLA.....	182
IMAGEN 68. BATALLAS GANADAS Y PERDIDAS SIN CERRAR.....	183
IMAGEN 69. BATALLAS GANADAS, PERDIDAS Y EN CURSO.....	183
IMAGEN 70. LISTA DE PROPUESTAS DE INTERCAMBIO.....	185
IMAGEN 71. SELECCIONAR CROMO	185
IMAGEN 72. PROPUESTAS DE INTERCAMBIO RECIBIDAS.....	185
IMAGEN 73. PROPONER CROMO	185
IMAGEN 74. PROPUESTA REALIZADA.....	186
IMAGEN 75. MENU MI COLECCIÓN.....	186
IMAGEN 76. PESTAÑA INTERCAMBIO	186
IMAGEN 77. ANULAR PROPUESTA DE INTERCAMBIO.....	186
IMAGEN 78. TESORO.....	187
IMAGEN 79. CROMO ENCONTRADO.....	187
IMAGEN 80. ERROR AL OBTENER LA UBICACIÓN.....	188

1. Introducción

La tecnología móvil está a la orden del día. Apenas han pasado unos pocos años desde el salto al 'mundo Smartphone' y, sin embargo, ya se ha convertido en una realidad tan arraigada en la sociedad que no se es capaz de concebir un solo día sin ella. Hoy por hoy se haría muy difícil desprenderse de las ventajas que las aplicaciones móviles proporcionan. Algo que no es de extrañar si se tiene en cuenta que el número de aplicaciones móviles no ha parado de crecer a pasos agigantados. Tanto es así, que las dos gigantes de la industria tecnológica en cuanto a 'Smartphones' se refiere, "Apple y Google; superaron recientemente la escalofriante cifra de 900.000 de aplicaciones en *App Store* y *Google Play* respectivamente" (2013 Apple Inc., 2013 Forbes.com LLC™).

Por todo ello, cada vez se hace más difícil encontrar algo nuevo, diferente, algo que no esté ya inventado, algo que funcione y que despierte interés, que sea entretenido. *Que sea entretenido*, este es el punto de partida y la piedra angular donde se sustenta la idea de la aplicación móvil que se pretende desarrollar.

Se pretende desarrollar una aplicación móvil que quiere potenciar el ocio, entretener. Durante el día existen infinidad de momentos muertos, momentos en los que la gente no sabe muy bien que hacer. Algunos de estos pueden ser cuando se está yendo en el autobús o en el metro, cuando se está esperando alguna cola o por ejemplo durante los anuncios publicitarios mientras se ve la televisión. El propósito es transformar esos momentos aburridos en momentos entretenidos que casi inciten a la persona a no querer bajarse en la próxima parada con tal de poder disfrutar de unos pocos minutos más de entretenimiento.

El proyecto se enmarca en un ámbito de ocio, social y educativo. De ocio, por tratarse de un juego y de ámbito social porque una parte importante del mismo transcurre a través de una actividad en la que lo social adquiere gran protagonismo. Precisamente es esta parte social una de las bazas clave con las que se pretende dar a conocer la aplicación en nuevos círculos, consiguiendo renombre y fomentando la participación al mayor número de personas posible. Además de su función lúdica, pretende aportar también una función educativa gracias a la información contenida en la misma. También se pretende que sirva como refuerzo de algunas actividades cerebrales y motoras, tanto para menores, como para jóvenes y adultos e incluso para personas en situaciones especiales que precisen potenciar algunas habilidades que serán incorporadas dentro de la aplicación a modo de juegos.

La idea de desarrollar una aplicación móvil en una plataforma que es, a día de hoy en términos de desarrollo, totalmente desconocida, supone todo un reto. No se trata simplemente de poner en práctica lo aprendido a lo largo de la carrera, el 'Trabajo Fin de Grado' ha de ir más allá. Partiendo de esta premisa, y puestos a empezar con algo nuevo, qué mejor manera de hacerlo que con un juego para *iOS*.

Antes de comenzar, existían serias dudas acerca de si se debería hacer el proyecto utilizando *Android* o *iOS*. Finalmente se ha decidido escoger esta segunda opción por pequeños detalles. La idea de aprender a usar el lenguaje *Objective-C* resulta muy atractiva. Además, se dispone de antemano de un *iPhone 4* y un *MacBook Pro*, con lo cual se cumplen todos los requisitos necesarios para poder empezar a desarrollar para *iOS*.

ColeccionApp, nombre con el que se ha bautizado a la aplicación, es por tanto, un juego con un objetivo claramente definido. Se trata de un gestor de colecciones de cromos/cartas. Esta aplicación permitirá gestionar colecciones de diferentes tipos pero con una estructura similar. El usuario final podrá escoger qué colecciones desea realizar entre una serie de colecciones previamente establecidas (Liga de fútbol, Formula 1, Los Simpson, etc.). El objetivo es completar la/s colección/es en la/s que participa.

Cada usuario dispondrá de un listado de las colecciones en las que participa. Por ejemplo, en el caso de la colección de Liga de fútbol, la estructura estará formada por el cromo del jugador, la información asociada al mismo y el equipo en el que juega, que es a su vez la familia donde se agruparán los jugadores. De la misma forma, esta estructura se puede hacer extensible a otro tipo de colecciones:

- Formula 1: piloto, información y escudería a la que pertenecen.
- Oliver y Benji: jugador, información y equipos al que pertenecen.
- NBA: jugador, información y equipo al que pertenecen.
- ...

Puede ocurrir que una colección no tenga una estructura tan definida como para que pueda ser dividida en varias familias. Es este caso la colección constará de una única familia:

- Big Bang Theory: personaje e información del personaje.
- Friends: personaje e información del personaje.
- ...

Dicho así, da la impresión de que no se trata de un juego. Sin embargo, el juego entra en escena en el momento en el que los usuarios intentan conseguir los cromos. Se han ideado varias formas para ello: *Reto*, *Batalla*, *Intercambio* y *Tesoro*.

La primera modalidad, *Reto*, es mediante pruebas que el usuario ha de afrontar y superar. En caso de superarlo el jugador conseguirá un cromo al azar.

Tal y como se ha mencionado anteriormente, el juego tiene un fuerte componente socializador. En la segunda modalidad, *Batalla*, los usuarios pueden conseguir cromos mediante duelos entre sí. Cada usuario tendrá la posibilidad de retar a otro usuario a través de diferentes tipos de prueba. Las batallas entre usuarios podrán ser de dos tipos: aleatorias o selectivas. En una batalla aleatoria, el jugador derrotado perderá un cromo al azar de su colección. La batalla selectiva en cambio, permitirá al jugador vencedor escoger un cromo de la colección de su rival. En cualquiera de las diferentes batallas el cromo del usuario perdedor pasará a formar parte de la colección del usuario que lo ha derrotado.

El *Intercambio* se ha pensado teniendo en cuenta que es más que probable que muchos usuarios tengan cromos repetidos. Se implementará la posibilidad de que aquellos jugadores que lo deseen puedan intercambiar cromos entre sí sin necesidad de afrontar ninguna batalla. Esta modalidad permite conservar la esencia de las colecciones de cromos.

Por último la modalidad *Tesoro*, utiliza la geo localización y es la más innovadora. Se dará al jugador la posibilidad de hacerse con un cromo simplemente yendo hacia él. Partiendo desde la ubicación en la que el jugador esté, se le enviará a unas coordenadas relativamente cercanas donde podrá recoger el cromo en cuestión.

Para poder gestionar las distintas colecciones existirá la figura de un administrador. El administrador será el único capaz de añadir, modificar y eliminar tanto las colecciones como los cromos que formen parte de las mismas. Por ejemplo, en el caso de la colección de la Liga; si con la liga ya empezada, un jugador cambia del Betis al Athletic en el mercado de invierno, el administrador podría hacer que el cromo correspondiente a ese jugador dejase de pertenecer a la familia Betis y pasase a formar parte de la familia Athletic, además, podría cambiar la imagen del cromo para adecuarla a su nuevo equipo. De igual modo, si un jugador viene como fichaje desde otra liga o, al contrario, abandona la Liga, el administrador podría añadir o eliminar el cromo correspondiente si lo considerase oportuno.

2 Planteamiento inicial

2.1 Objetivos

El principal objetivo es desarrollar un juego para *iOS*. Se desea comprobar la capacidad para llevar a cabo un proyecto completo de principio a fin. Sin embargo, no es ni mucho menos el único objetivo que se pretende alcanzar. Desenvolverse en el lenguaje de programación *Objective-C*, establecer conexiones remotas desde una base de datos externa a una aplicación móvil o explotar las distintas utilidades, como la geo localización, que proporciona esta tecnología, son sin duda otros de los objetivos que se persiguen.

Desde el punto de vista del proyecto, el objetivo es claro. Se busca fusionar el entusiasmo del coleccionismo con el placer que provocan los juegos en una misma aplicación. En definitiva, lograr la satisfacción de los usuarios a través del entretenimiento.

2.2 Arquitectura

Se empleará una arquitectura Cliente/Servidor, (Ver Figura 1). La aplicación del administrador será desarrollada en Java y accederá a la base de datos mediante la API *JDBC* para insertar los datos necesarios, a excepción de las imágenes que se transferirán a una carpeta del servidor. Esta conexión se establecerá mediante el protocolo *SFTP*. Por su parte, la aplicación móvil del usuario, ejecutará sentencias contra la base de datos mediante llamadas a ficheros *PHP* alojados en el servidor. Además, la aplicación del usuario tendrá su propia base de datos local *SQLite* incluida en el móvil. Tanto la base de datos del servidor como la de la aplicación comparten el mismo diseño. Las únicas diferencias son el tipo de base de datos empleada y que los datos incluidos en la aplicación del usuario únicamente hacen referencia al propio usuario.

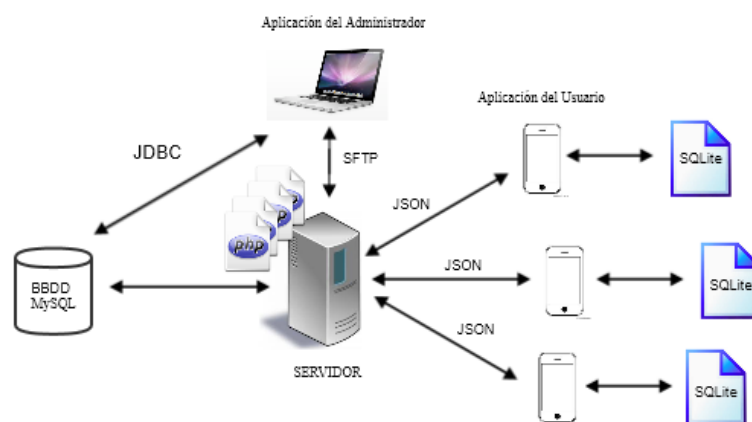


Figura 1. Arquitectura

2.3 Alcance

El proyecto será desarrollado según un ciclo de vida incremental. Consiste en la iteración de varios ciclos de vida en cascada. Con cada iteración, se obtienen un prototipo con más funcionalidades que el anterior. Se van generando versiones cada vez más completas. De esta forma, cada una de las versiones tiene más funcionalidades que la anterior tal y como se muestra en la Figura 2. Los ciclos se suceden hasta satisfacer los requisitos.

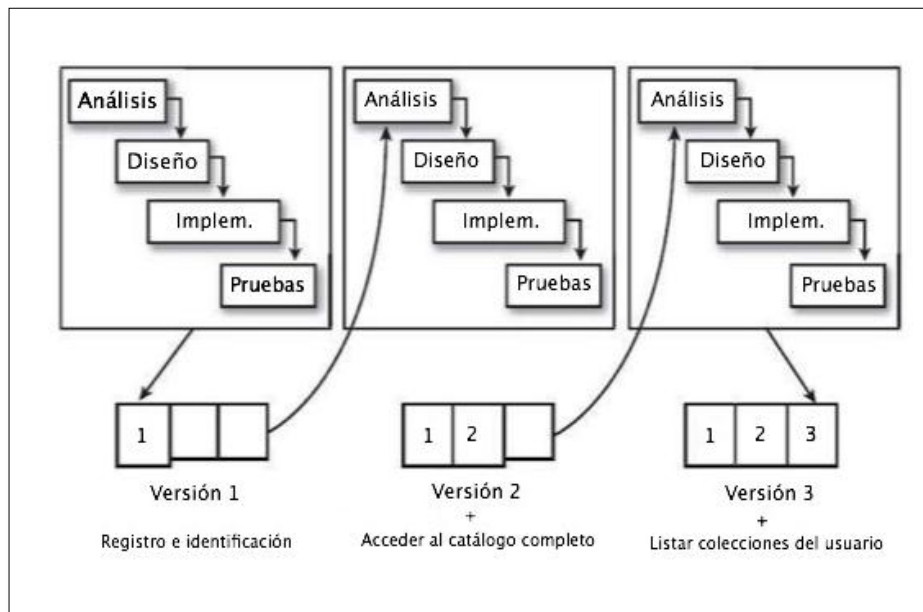


Figura 2. Ciclo de vida incremental – Apuntes Ingeniería del Software

El ciclo de vida incremental se basa en la generación de prototipos. La razón por la que se ha escogido este ciclo de vida frente a otros es que, a pesar de que la gestión pueda resultar más complicada, proporciona otras ventajas que se adecuan a la manera en la que se pretende trabajar. Por un lado, permite controlar mejor la calidad del software. Si hay un error grave, éste, solo afectará a la última iteración. Además, al tratarse de prototipos se puede probar cada paquete individualmente e ir ampliando las funcionalidades de la aplicación progresivamente.

Cada una de las tareas que forman parte del proyecto, se representan en la Estructura de Descomposición del Trabajo (EDT), representado en Figura 3. Todo aquello que no se encuentre reflejado en el EDT no existe en el proyecto. El EDT se compone de tres partes fundamentales: Documentación, Captura de requisitos y Módulos o Prototipos.

La Documentación se subdivide en dos apartados. El primero es la Elaboración del Documento de Objetivos del Proyecto (DOP), donde quedan recogidas las intenciones del proyecto, es decir, lo que se quiere conseguir al desarrollarlo. El segundo apartado es la Memoria del Proyecto. Es la documentación que recoge toda la información asociada al mismo.

La Captura de Requisitos, presente en el segundo módulo del EDT, está formada por el Diagrama de Casos de Uso con su correspondiente Jerarquía de Actores, el Modelo de Dominio, los Casos de Uso Extendidos y la Transformación del Modelo de Dominio a Base de Datos.

Por último se encuentran los Módulos, pequeños paquetes de software que se irán implementado de forma progresiva generando así versiones cada vez más completas de la aplicación. Los paquetes se han dividido de la siguiente manera: *Gestionar las colecciones*, *Registro e Identificación de usuario*, *Acceder al catálogo completo*, *Listar colecciones del usuario*, *Acceder a una colección*, *Afrontar un reto contra la máquina*, *Intercambiar cromos*, *Provocar batalla* y *Buscar tesoro*. Cada uno de estos paquetes tiene su propia fase de Análisis, Diseño, Implementación y Pruebas.

Estructura de Descomposición del Trabajo

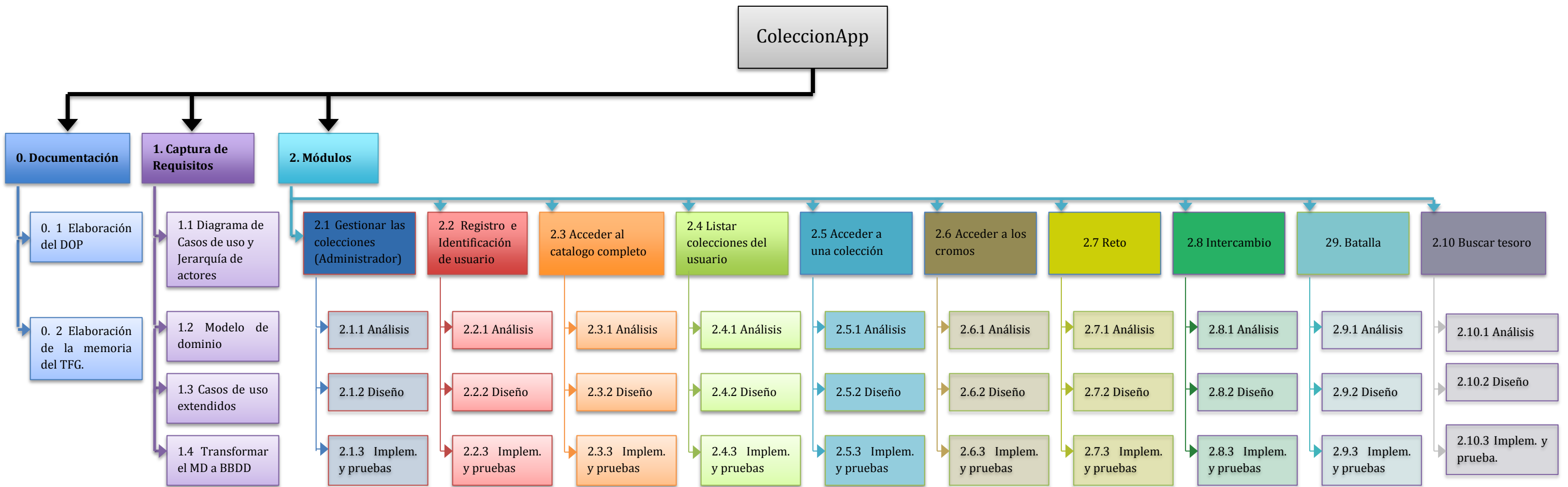


Figura 3. Estructura de Descomposición del Trabajo

Descripción del paquete de trabajo

Tarea: Elaboración del Documento de Objetivos del Proyecto (DOP)

Duración estimada: 40 horas

Descripción: Documento donde se encuentran recogidas las intenciones del proyecto.

Entradas: -----

Salidas/Entregables: Documento de Objetivos del Proyecto (DOP)

Recursos necesarios: Un ordenador y Microsoft Word

Precedencias: -----

Tabla 1. Elaboración del Documento de Objetivos del Proyecto

Descripción del paquete de trabajo

Tarea: Elaboración de la memoria del Trabajo Fin de Grado (TFG).

Duración estimada: 50 horas

Descripción: Memoria del proyecto. Recoge toda la información asociada al Trabajo Fin de Grado.

Entradas: Documento de objetivos del proyecto

Salidas/Entregables: Memoria del Trabajo Fin de Grado.

Recursos necesarios: Un ordenador, Microsoft Word, Microsoft Excel, Visual Paradigm, Eclipse y XCode.

Precedencias: Análisis, Diseño e Implementación y Pruebas de cada uno de los módulos de software.

Tabla 2. Elaboración de la memoria del Trabajo Fin de Grado

Descripción del paquete de trabajo

Tarea: Diagrama de Casos de Uso y Jerarquía de Actores

Duración estimada: 6 horas

Descripción: Diagrama que representa las funcionalidades de la aplicación así como los actores que hacen uso de ellas y su jerarquía.

Entradas: -----

Salidas/Entregables: Diagrama de Casos de Uso

Recursos necesarios: Visual Paradigm

Precedencias: -----

Tabla 3. Diagrama de Casos de Uso

Descripción del paquete de trabajo

Tarea: Modelo de Dominio

Duración estimada: 6 horas

Descripción: Diagrama que representa aquellas entidades cuyos datos se necesitan almacenar para desarrollar la aplicación y las relaciones entre ellas.

Entradas: -----

Salidas/Entregables: Modelo de Dominio

Recursos necesarios: Visual Paradigm

Precedencias: Diagrama de Casos de Uso

Tabla 4. Modelo de dominio

Descripción del paquete de trabajo

Tarea: Casos de Uso Extendidos

Duración estimada: 25 horas

Descripción: Este paquete de trabajo abarca en primer lugar la generación de las interfaces gráficas. Una vez se tienen, se describen a fondo las funcionalidades de los Casos de Uso. Se detalla punto por punto cada una de las posibilidades que se pueden dar en cada funcionalidad.

Entradas: Diagrama de Casos de Uso

Salidas/Entregables: Casos de Uso Extendidos

Recursos necesarios: Microsoft Word, Xcode y Eclipse para las interfaces.

Precedencias: Diagrama de Casos de Uso

Tabla 5. Casos de Uso Extendidos

Descripción del paquete de trabajo

Tarea: Transformación del Modelo de Domino a Base de Datos

Duración estimada: 10 horas

Descripción: Esta tarea consiste en convertir el modelo de dominio previamente generado en una base de datos relacional.

Entradas: Modelo de Dominio

Salidas/Entregables: Base de datos

Recursos necesarios: Visual Paradigm

Precedencias: Modelo de dominio

Tabla 6. Transformación del Modelo de Dominio a Base de Datos

Descripción del paquete de trabajo

Tarea: Gestionar las colecciones

Duración estimada: 62 horas

Descripción: Esta es una tarea para el administrador. Todo el desarrollo de la aplicación del administrador se desarrollará en este prototipo. Incluye *Análisis, Diseño, Implementación y pruebas* así como la documentación del paquete de software.

Entradas: Casos de uso extendidos del administrador

Salidas/Entregables: Paquete de software

Recursos necesarios: SQLite y Eclipse.

Precedencias: Captura de Requisitos

Tabla 7. Gestionar las colecciones

Descripción del paquete de trabajo

Tarea: Registro e Identificación del usuario

Duración estimada: 32 horas

Descripción: Funcionalidad correspondiente al registro y la identificación del usuario. Incluye *Análisis, Diseño, Implementación y pruebas* así como la documentación del paquete de software.

Entradas: Caso de uso extendido *Identificarse y Registrarse*

Salidas/Entregables: Paquete de software

Recursos necesarios: SQLite y Xcode.

Precedencias: Captura de Requisitos

Tabla 8. Identificación y registro de usuario

Descripción del paquete de trabajo

Tarea: Acceder al catálogo completo

Duración estimada: 18 horas

Descripción: Funcionalidad correspondiente al listado de todas las colecciones existentes. Incluye *Análisis, Diseño, Implementación y pruebas* así como la documentación del paquete de software.

Entradas: Caso de uso extendido *Acceder al catálogo completo*

Salidas/Entregables: Paquete de software

Recursos necesarios: SQLite y Xcode.

Precedencias: Captura de Requisitos

Tabla 9. Acceder al catálogo completo

Descripción del paquete de trabajo

Tarea: Listar colecciones del usuario

Duración estimada: 15 horas

Descripción: Funcionalidad correspondiente al listado de todas las colecciones en la que el usuario participa. Incluye *Análisis, Diseño, Implementación y pruebas* así como la documentación del paquete de software.

Entradas: Caso de uso extendido *Listar tus colecciones*

Salidas/Entregables: Paquete de software

Recursos necesarios: SQLite y Xcode.

Precedencias: Captura de Requisitos

Tabla 10. Listar colecciones del usuario

Descripción del paquete de trabajo

Tarea: Acceder a una colección

Duración estimada: 18 horas

Descripción: Funcionalidad correspondiente al acceso a una colección por parte de usuario. Incluye *Análisis, Diseño, Implementación y pruebas* así como la documentación del paquete de software.

Entradas: Caso de uso extendido *Listar tus colecciones*

Salidas/Entregables: Paquete de software

Recursos necesarios: SQLite y Xcode.

Precedencias: Captura de Requisitos

Tabla 11. Acceder a una colección

Descripción del paquete de trabajo

Tarea: Acceder a los cromos

Duración estimada: 15 horas

Descripción: Funcionalidad correspondiente al acceso de todos los cromos del usuario. Incluye *Análisis, Diseño, Implementación y pruebas* así como la documentación del paquete de software.

Entradas: Caso de uso extendido *Acceder a los cromos*

Salidas/Entregables: Paquete de software

Recursos necesarios: SQLite y Xcode.

Precedencias: Captura de Requisitos

Tabla 12. Acceder a los cromos

Descripción del paquete de trabajo

Tarea: Afrontar un reto contra la máquina

Duración estimada: 52 horas

Descripción: Funcionalidad correspondiente al reto que el usuario afronta contra la máquina para conseguir un cromó. Incluye *Análisis, Diseño, Implementación y pruebas* así como la documentación del paquete de software.

Entradas: Caso de uso extendido Afrontar un reto contra la máquina

Salidas/Entregables: Paquete de software

Recursos necesarios: SQLite y Xcode.

Precedencias: Captura de Requisitos

Tabla 13. Afrontar un reto contra la máquina

Descripción del paquete de trabajo

Tarea: Intercambiar cromos

Duración estimada: 18 horas

Descripción: Funcionalidad correspondiente al intercambio de cromos entre usuarios. Incluye *Análisis, Diseño, Implementación y pruebas* así como la documentación del paquete de software.

Entradas: Caso de uso extendido *Intercambiar cromos*

Salidas/Entregables: Paquete de software

Recursos necesarios: SQLite y Xcode.

Precedencias: Captura de Requisitos

Tabla 14. Intercambiar cromos

Descripción del paquete de trabajo

Tarea: Provocar batalla

Duración estimada: 100 horas

Descripción: Funcionalidad correspondiente a las batallas entre usuarios. Incluye *Análisis, Diseño, Implementación y pruebas* así como la documentación del paquete de software.

Entradas: Caso de uso extendido *Provocar batalla*

Salidas/Entregables: Paquete de software

Recursos necesarios: SQLite y Xcode.

Precedencias: Captura de Requisitos

Tabla 15. Provocar batalla

Descripción del paquete de trabajo

Tarea: Buscar tesoro

Duración estimada: 100 horas

Descripción: Funcionalidad correspondiente a la búsqueda de un tesoro. Incluye *Análisis, Diseño, Implementación y pruebas* así como la documentación del paquete de software.

Entradas: Caso de uso extendido *Buscar tesoro*

Salidas/Entregables: Paquete de software

Recursos necesarios: SQLite y XCode

Precedencias: Captura de Requisitos

Tabla 16. Buscar tesoro

Al tratarse de un diagrama semanal no se aprecian al detalle la duración de las tareas. Como solución se ha generado una tabla para agrupar la duración de las mismas, (Ver Tabla 17).

TAREA	DURACIÓN (Horas)
Documentación	
<i>Elaboración del DOP</i>	40
<i>Memoria del Trabajo Fin de Grado</i>	50
Captura de requisitos	
<i>Diagrama de Casos de uso y Jerarquía de actores</i>	6
<i>Modelo de Dominio</i>	6
<i>Casos de uso extendidos</i>	25
<i>Transformación MD en BBDD</i>	10
Módulos	
<i>Gestionar las colecciones</i>	62
<i>Registro e Identificación del Usuario</i>	32
<i>Acceder al catálogo completo</i>	18
<i>Acceder a una colección</i>	15
<i>Listar colecciones del usuario</i>	18
<i>Acceder a los cromos</i>	15
<i>Afrontar un reto contra la máquina</i>	62
<i>Intercambiar cromos</i>	18
<i>Provocar batalla</i>	100
<i>Buscar tesoro</i>	100
Duración total estimada	577

Tabla 17. Duración total de las tareas

En base a estas estimaciones se ha generado una nueva tabla con el intervalo entre fechas, para indicar cuando se estima que estarán hechas, (Ver Tabla 18). De esta manera, queda una estimación más precisa de cuantos días costará llevar una tarea a cabo. En cualquier caso no se debe olvidar que se trata de una estimación orientativa.

Documentación	
<i>Elaboración del DOP</i>	19 de Noviembre de 2013 – 17 de Diciembre de 2013
<i>Memoria del Trabajo Fin de Grado</i>	19 de Noviembre de 2013 – 12 de Junio de 2014
Captura de requisitos	
<i>Diagrama de Casos de uso y Jerarquía de actores</i>	18 de Diciembre de 2013 – 19 de Diciembre de 2013
<i>Modelo de Dominio</i>	20 de Diciembre de 2013 – 23 de Diciembre de 2013
<i>Casos de uso extendidos</i>	24 de Diciembre de 2013 – 01 de Enero de 2014
<i>Transformación MD en BBDD</i>	02 de Enero de 2014 – 06 de Enero 2014
Módulos	
<i>Gestionar las colecciones</i>	07 de Enero de 2014 – 29 de Enero de 2014
<i>Registro e Identificación del Usuario</i>	30 de Enero de 2014 – 07 de Febrero de 2014
<i>Acceder al catálogo completo</i>	10 de Febrero de 2014 – 14 Febrero de 2014
<i>Acceder a una colección</i>	17 de Febrero de 2014 – 21 de Febrero de 2014
<i>Listar colecciones del usuario</i>	24 de Febrero de 2014 – 27 de Febrero de 2014
<i>Acceder a los cromos</i>	28 de Febrero de 2014 – 05 de Marzo de 2014
<i>Afrontar un reto contra la máquina</i>	06 de Marzo de 2014 – 25 de Marzo de 2014
<i>Provocar batalla</i>	26 de Marzo de 2014 – 30 de abril de 2014
<i>Intercambiar cromos</i>	01 de Mayo de 2014 – 07 de Mayo de 2014
<i>Buscar tesoro</i>	08 de Mayo de 2014 – 08 de Junio de 2014

Tabla 18. Periodo de realización de las tareas

2.5 Herramientas

En esta sección, se listan aquellas herramientas de software que se pretenden utilizar para el desarrollo de la aplicación.

Para poder llevar a cabo los paquetes de trabajo se necesita hacer uso de herramientas de todo tipo. Algunas de ellas se utilizarán simplemente para tareas de documentación, muchas otras con fines de implementación y el resto para tareas de análisis y diseño.

Herramientas para tareas de documentación, análisis y diseño:

- *Visual Paradigm for UML standard edition*, software para la Captura de Requisitos, Diseño de software y Bases de Datos. Permite generar Modelo de Dominio, Diagramas de Casos de Uso, Diagramas de Clases, Diagramas de Secuencia, así como diseñar Bases de Datos relacionales.
- Office para Mac Hogar y Estudiantes 2011, paquete de software ofimático que incluye Microsoft Word 2011, Microsoft Excel 2011 y Microsoft PowerPoint 2011.

Herramientas para la implementación:

- *XCode*, entorno de desarrollo integrado de Apple Inc. que se suministra gratuitamente junto con Mac OS X. Es donde se llevará a cabo el desarrollo de la aplicación móvil. Permite compilar código *Objective-C* mediante una amplia gama de modelos de programación, como por ejemplo *Cocoa Touch*.
- *Cocoa Touch*, framework o biblioteca que permite el desarrollo de aplicaciones nativas para *iOS*.
- *Eclipse*, entorno de desarrollo integrado de código abierto multiplataforma que permite la compilación en tiempo real. Es donde se llevará a cabo el desarrollo de la aplicación del administrador.
- *MAMP*, paquete de distribución que cuenta con una colección de aplicaciones entre las cuales podemos encontrar: *Apache*, *MySQL*, *PHP*, *phpMyAdmin*, *MiniPerl*, *Webalizar*, *FileZilla FTP Service*, *SQLite*, *Mercury Mail*, *PEAR*, *OpenSSL*, entre otras. Lo que logra *MAMP* es instalar todo en conjunto y configurarlo para evitar trabajo adicional al desarrollador.
- *SQLite*, biblioteca de software que implementa un motor de base datos *SQL* transaccional contenido en sí mismo, sin configuración, y sin servidor.

2.6 Gestión de riesgos

Para evitar pérdidas potenciales durante el transcurso del proyecto, es importante generar un plan de gestión de riesgos para poder mitigar al máximo los efectos que estos puedan tener sobre el mismo, así como prevenir las causas que puedan conducir a ellos. Debido a la subjetividad que puede suponer el cálculo de la probabilidad de determinados riesgos, se ha diseñado una tabla orientativa, (Ver Tabla 19).

Probabilidad	Porcentaje
<i>Improbable</i>	0 - 30 %
<i>Poco probable</i>	30 - 65 %
<i>Muy probable</i>	65 - 100 %

Tabla 19. Probabilidad

Además de la probabilidad de que ocurra un determinado riesgo, otro factor a tener en cuenta es el impacto que causaría en caso de producirse. Habrá riesgos que no supongan demasiado impacto. Otros en cambio, si se produjeran, podrían retrasar el proyecto de forma considerable. Al igual que sucede con las probabilidades, se ha generado otra tabla con la intención de ofrecer una idea más precisa del retraso que supondría, en términos de días, los diferentes impactos en caso de producirse. Puede observarse en la Tabla 20.

Impacto	Retraso
<i>Muy leve</i>	Menos de un día (horas)
<i>Leve</i>	1 día
<i>Medio</i>	2 - 3 días
<i>Grande</i>	3 - 7 días
<i>Muy grande</i>	7 días o más

Tabla 20. Impacto

En base a estos criterios, probabilidad e impacto, se ha de generar una prevención y un plan de contingencia que se adecue al riesgo en cuestión.

A lo largo de todo el proyecto, *ColeccionApp* se verá sometido a todo tipo de contingencias. Los riesgos podrían afectar a lo que se pretende hacer, y por tanto, ocurrirían durante el desarrollo del mismo. *Riesgos de desarrollo*, podrían ser el *cambio del firmware de iOS*, la *modificación de las especificaciones del proyecto* o la *indisposición del desarrollador* entre otros. Este último, a su vez, tiene subcategorías según las causa (enfermedad, trabajo, etc.). Otro tipo de riesgo que puede suceder es el que afecte directamente al hardware y software durante el proceso de implementación; infección de un virus, rotura del portátil, etc. Es un riesgo fácil de categorizar por lo que se ha incluido dentro de los *Riesgos de desarrollo* en el apartado *Riesgos de hardware y software*.

Sin embargo, no todos los riesgos son propios del proceso de desarrollo, algunos de ellos pueden afectar al proyecto una vez que éste ya se ha desarrollado. Son los *Riesgos del*

producto generado. En este apartado se incluyen los *Riesgos de seguridad* y los *Riesgos de que haya más usuarios de lo planificado*.

D. Riesgos de desarrollo	P. Riesgos del producto generado
<i>D1. Cambio en el firmware de iOS</i>	<i>P1. Desaparición de alguna librería utilizada</i>
<i>D2. Carencia de una API necesaria</i>	<i>P2. Mayor número de usuarios de lo planificado</i>
<i>D3. Planificación temporal incorrecta</i>	<i>P2.1. Tamaño de la BD demasiado pequeña</i>
<i>D4. Cambio de las especificaciones del proyecto</i>	<i>P2.2. Sobrecargar el servidor de peticiones</i>
<i>D5. Indisposición temporal del desarrollador</i>	<i>P3. Riesgos de seguridad en la base de datos</i>
<i>D5.1. Enfermedad</i>	<i>P3.1. Inyección SQL</i>
<i>D5.2. Empleo / Prácticas</i>	<i>P3.2. Pérdida de autenticación y gestión de sesiones</i>
<i>D5.3. Irse a vivir a otro país</i>	<i>P3.3. Defectuosa configuración de seguridad</i>
<i>D6. Riesgos de hardware y software</i>	<i>P3.4. Almacenamiento criptográfico inseguro</i>
<i>D6.1. Infección de un virus</i>	
<i>D6.2. Rotura del equipo</i>	
<i>D6.3. Perdida / Robo del teléfono móvil</i>	

Tabla 21. Riesgos

Una vez identificados los posibles riesgos, es hora de analizarlos individualmente. Como se ha mencionado con anterioridad, se ha de minimizar la posibilidad de que los riesgos ocurran mediante una prevención. Aun así, el riesgo puede producirse, en tal caso se ha de saber que hay que hacer a través de un plan de contingencia. Tanto en la identificación, como en la prevención y contingencia se ha tratado de adecuar lo más posible estos factores al proyecto *ColeccionApp*, especificando al detalle cada una de las medidas que se llevarán a cabo.

En primer lugar se analizan los *Riesgos de desarrollo*, riesgos que pueden ocurrir durante el desarrollo del mismo.

D1. Cambio en el firmware de iOS	
Prevención	Estar al corriente de las funcionalidades que desaparecerán con la actualización de las siguientes versiones del firmware y realizar la programación acorde a ello tratando de evitarlas
Plan de contingencia	Cambiar o eliminar las funcionalidades incompatibles
Probabilidad	Improbable
Impacto	Muy grande

Tabla 22. Cambio en el firmware de iOS

D2. Carencia de una API necesaria para el desarrollo de alguna funcionalidad	
Prevención	Documentarse. Comprobar si existen APIs disponibles para lo que se pretende desarrollar con antelación.
Plan de contingencia	Replanteamiento de las funcionalidades afectadas. Búsqueda de alternativas a implementar.
Probabilidad	Poco probable
Impacto	Grande

Tabla 23. Carencia de una API necesaria para el desarrollo de alguna funcionalidad

D3. Planificación temporal incorrecta	
Prevención	Realizar una planificación temporal lo más precisa posible
Plan de contingencia	Replantear la planificación temporal para recuperar el máximo tiempo perdido posible.
Probabilidad	Muy probable
Impacto	Medio

Tabla 24. Planificación temporal incorrecta

D4. Cambio de las especificaciones del proyecto	
Prevención	Generar un proyecto lo más modular posible de modo que se puedan aprovechar al máximo las funcionalidades ya implementadas
Plan de contingencia	Replanteamiento de las especificaciones modificadas.
Probabilidad	Poco probable
Impacto	Grande

Tabla 25. Cambio de las especificaciones del proyecto

D5. Indisposición temporal del desarrollador	
D5.1. Enfermedad	
Prevención	Extremar las precauciones para evitar exponerse a enfermedades
Plan de contingencia	Tomar las medidas necesarias para recuperarse lo antes posible, y reorganizar la planificación temporal para recuperar el tiempo perdido
Probabilidad	Muy probable
Impacto	Leve
D5.2. Empleo / Prácticas	
Prevención	Escoger un empleo / prácticas a media jornada de modo que permita compaginar ambas actividades
Plan de contingencia	Reorganizar el tiempo dedicado a actividades extraescolares para que la planificación temporal del proyecto no se vea resentida
Probabilidad	Probable
Impacto	Medio
D5.3. Irse a vivir a otro país	
Prevención	Utilizar un equipo propio para poder seguir trabajando en el extranjero
Plan de contingencia	En lugar de asistir físicamente realizar las consultas con el tutor del proyecto a través del correo electrónico
Probabilidad	Poco probable
Impacto	Grande

Tabla 26. Indisposición temporal del desarrollador

D6. Riesgos de hardware y software	
D6.1. Rotura del equipo	
Prevención	Mantener el equipo protegido de agentes externos dañinos (comida, bebidas,...). Hacer uso responsable del mismo. Se realizarán copias de seguridad semanales en un disco duro externo y en la plataforma Dropbox.
Plan de contingencia	Restaurar las copias de seguridad. Utilización de otro equipo del aula de tecnologías móviles para seguir trabajando mientras se repara el equipo.
Probabilidad	Poco probable
Impacto	Leve
D6.2. Infección de un virus	
Prevención	Instalación, mantenimiento y actualización del antivirus gratuito AVG para Mac. Se realizarán copias de seguridad en un disco duro externo y en la plataforma Dropbox.
Plan de contingencia	Revisar el equipo, eliminar la amenaza y cargar los datos de la última copia de seguridad
Probabilidad	Poco probable
Impacto	Muy leve
D6.3. Perdida / robo del teléfono móvil *	
Prevención	Extremar las precauciones para evitar robos o pérdidas. Activar servicio 'Buscar mi iPhone' proporcionado por Apple.
Plan de contingencia	Denunciar el Robo/Perdida. Tratar de localizar el móvil a través del servicio mencionado. En caso de no localizarlo, pedir prestado un móvil Apple para poder continuar con el proyecto.
Probabilidad	Poco probable
Impacto	Muy grande

Tabla 27. Riesgos de hardware y software

* La pérdida/robo del móvil no se verificará en función de la valoración que se le ha dado. Más bien es un riesgo que será comprobado a diario. Los riesgos de producto generado únicamente podrán ser verificados una vez el producto sea generado, es decir, cuando la aplicación esté completada.

Analizados ya los riesgos asociados al desarrollo del proyecto, se procede al análisis de los riesgos que pueden amenazar a la aplicación cuando ya está terminada, *Riesgos del producto generado*.

P1. Desaparición de alguna librería utilizada	
Prevención	Documentarse de las librerías que podrían desaparecer en el futuro. Descargarse la librería si es posible.
Plan de contingencia	Si se pudo descargar la librería, utilizarla en modo local. Si no, no utilizar la funcionalidad que requiera esa librería.
Probabilidad	Improbable
Impacto	Muy grande

Tabla 28. Descripción de alguna librería utilizada

P2. Mayor número de usuarios de lo planificado	
P2.1. Tamaño de la base de datos demasiado pequeña	
Prevención	Realizar una base de datos de tamaño suficiente para soportar una gran cantidad de usuarios.
Plan de contingencia	Ampliar el tamaño de la base de datos y cargar la copia de seguridad
Probabilidad	Probable
Impacto	Grande
P2.2. Sobrecarga en el servidor de peticiones de usuarios	
Prevención	Tratar de implementar una funcionalidad capaz de soportar una gran cantidad de usuarios realizando peticiones al mismo tiempo.
Plan de contingencia	Limitar el número de usuarios que pueden estar conectados al mismo tiempo.
Probabilidad	Probable
Impacto	Medio

Tabla 29. Mayor número de usuarios de lo planificado

P3. Riesgos de seguridad en la base de datos	
P3.1. Inyección SQL	
Prevención	Implementar una funcionalidad que escape los caracteres para evitar el acceso no autorizado.
Plan de contingencia	Revisar la funcionalidad implementada. Comprobar que los datos no hayan sido corrompidos, y si es el caso restaurarlos con la última copia de seguridad.
Probabilidad	Poco probable
Impacto	Medio
P3.2. Pérdida de autenticación y gestión de sesiones del administrador	
Prevención	Implementar un código que introduzca un tiempo de inactividad en el sistema. Al cabo de un tiempo determinado la sesión del administrador se cerrará.
Plan de contingencia	Revisar la funcionalidad implementada. Comprobar que los datos no hayan sido corrompidos, y si es el caso restaurarlos con la última copia de seguridad.
Probabilidad	Poco probable
Impacto	Medio
P3.3. Defectuosa configuración de seguridad	
Prevención	En Mamp en lugar de usar el usuario <i>root</i> crear un nuevo usuario que solo tenga permisos para acceder a la base de datos <i>ColeccionApp</i>
Plan de contingencia	Comprobar que los datos no hayan sido corrompidos, y si es el caso restaurarlos con la última copia de seguridad.
Probabilidad	Poco probable
Impacto	Medio
P3.4. Almacenamiento criptográfico inseguro	
Prevención	Cifrar los datos antes de introducirlos a la base de datos
Plan de contingencia	Comprobar que los datos no hayan sido corrompidos, y si es el caso restaurarlos con la última copia de seguridad.
Probabilidad	Poco probable
Impacto	Medio

Tabla 30. Riesgos de seguridad en la base de datos

Tras el análisis de los distintos riesgos posibles llega el momento de dar una valoración en función de la probabilidad y el impacto de los mismos. La valoración puede ser leve, normal e importante. Es importante hacer un seguimiento periódico para comprobar si se está cumpliendo el plan de riesgos. Según el grado de la valoración se le dará un tipo de seguimiento al riesgo u otro, (Ver Tabla 31).

Valoración	Seguimiento de los riesgos
<i>Leve</i>	Cada mes
<i>Normal</i>	Cada 15 días
<i>Importante</i>	Cada semana

Tabla 31. Valoraciones

Se han valorado los riesgos tal y como se muestran en la Tabla 32 y Tabla 33.

Riesgos de desarrollo	Probabilidad	Impacto	Valoración
<i>D1. Cambio en el firmware de iOS</i>	<i>Improbable</i>	<i>Muy grande</i>	<i>Normal</i>
<i>D2. Carencia de una API necesaria</i>	<i>Poco probable</i>	<i>Grande</i>	<i>Normal</i>
<i>D3. Planificación temporal incorrecta</i>	<i>Muy probable</i>	<i>Medio</i>	<i>Importante</i>
<i>D4. Cambio de las especificaciones del proyecto</i>	<i>Poco probable</i>	<i>Grande</i>	<i>Normal</i>
D5. Indisposición temporal del desarrollador			
<i>D5.1. Enfermedad</i>	<i>Muy probable</i>	<i>Leve</i>	<i>Normal</i>
<i>D5.2. Empleo / Prácticas</i>	<i>Probable</i>	<i>Medio</i>	<i>Leve</i>
<i>D5.3. Irse a vivir a otro país</i>	<i>Poco probable</i>	<i>Grande</i>	<i>Normal</i>
D6. Riesgos de hardware y software			
<i>D6.1. Infección de un virus</i>	<i>Poco probable</i>	<i>Leve</i>	<i>Normal</i>
<i>D6.2. Rotura del equipo</i>	<i>Poco probable</i>	<i>Muy leve</i>	<i>Normal</i>
<i>D6.3. Perdida / Robo del teléfono móvil</i>	<i>Poco probable</i>	<i>Muy grande</i>	<i>Importante</i>

Tabla 32. Valoración de los riesgos de desarrollo

Riesgos de producto generado	Probabilidad	Impacto	Valoración
<i>P1. Desaparición de alguna librería utilizada</i>	<i>Improbable</i>	<i>Muy grande</i>	<i>Normal</i>
P2. Mayor número de usuarios de lo planificado			
<i>P2.1. Tamaño de la base de datos demasiado pequeña</i>	<i>Probable</i>	<i>Grande</i>	<i>Importante</i>
<i>P2.2. Sobrecargar el servidor de peticiones</i>	<i>Probable</i>	<i>Medio</i>	<i>Normal</i>
P3. Riesgos de seguridad de la base de datos			
<i>P3.1 Inyección SQL</i>	<i>Poco probable</i>	<i>Medio</i>	<i>Leve</i>
<i>P3.2. Pérdida de autenticación y gestión de sesiones</i>	<i>Poco probable</i>	<i>Medio</i>	<i>Leve</i>
<i>P3.3. Defectuosa configuración de seguridad</i>	<i>Poco probable</i>	<i>Medio</i>	<i>Leve</i>
<i>P3.4. Almacenamiento criptográfico inseguro</i>	<i>Poco probable</i>	<i>Medio</i>	<i>Leve</i>

Tabla 33. Valoración de los riesgos de producto generado

Dado que la mayor parte de los riesgos requieren copias de seguridad, ya sean de los diagramas, desarrollo, documentación o base de datos, se realizarán copias de seguridad cada 15 días de todo ello. Se utilizará la plataforma *Dropbox* y un disco duro externo para hacerlas.

2.7 Evaluación económica

Tras realizar la planificación temporal y obtener como resultado el diagrama Gantt, se presenta la evaluación económica del proyecto. Para su elaboración, se han tenido en cuenta los siguientes factores:

- Cada hora de documentación cuesta 25 euros.
- Cada hora de análisis, diseño e implementación cuesta 40 euros.
- Gasto de las licencias del software.
- Gasto del hardware empleado.
- Gastos indirectos.
- Los beneficios que generará el proyecto una vez terminado.

El coste total de las licencias de software que se emplearán es de 337 euros. Teniendo en cuenta que la media de vida útil del software asciende a unos 3 años, la amortización supone un coste de 74,9 euros. Se pueden ver los cálculos en la Tabla 34.

Descripción	Coste	Amortización	Coste en 8 meses
<i>Visual Paradigm for UML standard edition</i>	218 €		48,45 €
<i>Office para Mac Hogar y Estudiantes 2011</i>	119 €	3 años	26,45 €
<i>Total software</i>	337 €		74,9 €

Tabla 34. Coste de las licencias de software

Para el desarrollo de *ColeccionApp* se empleará un equipo portátil valorado en 1.300 euros. También se utilizará un Smartphone de Apple, modelo iPhone 4 valorado en 250 euros. La vida media útil del portátil es de 5 años, con lo cual la amortización correspondiente en este caso asciende a 174,34 euros, (Ver Tabla 35).

Descripción	Coste	Amortización	Coste en 8 meses
<i>MacBook Pro 2,5 GHz Intel Core i5</i>	1.300 €	5 años	173,34 €

Tabla 35. Coste del equipo

Descripción	Coste	Amortización	Coste en 3 meses
<i>iPhone 4 *</i>	250 €	4 años	15,63 €

Tabla 36. Coste del móvil

* Aunque generalmente la vida útil de un móvil no supera los dos años de vida, en este caso se utilizará un dispositivo móvil que ya tiene casi 4 años.

Según la planificación temporal se estima que se invertirán un total de 567 horas aproximadas, 90 horas se emplearán en labores de documentación, mientras que 477 horas serán destinadas a tareas de análisis, diseño e implementación y pruebas. Teniendo en cuenta lo que cuesta cada hora respectiva de trabajo, el coste de desarrollo asciende a 21.330 euros. Ver coste de desarrollo en la Tabla 37.

Descripción	Horas	Coste / Hora	Coste
<i>Horas de elaboración de documentación</i>	90 Horas	25 €/Hora	2.250 €
<i>Horas de análisis, diseño e imp. y pruebas</i>	477 Horas	40 €/Hora	19.080 €
<i>Total desarrollo</i>			21.330 €

Tabla 37. Coste de desarrollo

A los gastos de las licencias de software, hardware y desarrollador, hay que añadir los llamados gastos indirectos. Los gastos indirectos son los gastos de mantenimiento que se producen durante el proceso de elaboración del proyecto; luz, agua, electricidad, etc. Se ha considerado que suponen un 5% del coste total de proyecto. Con todo ello, los gastos totales del proyecto suponen 22.673,57 euros. (Ver Tabla 38).

Descripción	Coste
<i>Gastos de licencias de software</i>	74,9 €
<i>Gastos de hardware</i>	188,97 €
<i>Gastos de desarrollo</i>	21.330 €
<i>Gastos indirectos</i>	1079,7 €
<i>Total proyecto</i>	22.673,57 €

Tabla 38. Gatos totales del proyecto.

Como se puede comprobar realizar una aplicación móvil no es barato. Es importante pararse a pensar de qué manera se obtendrán los ingresos con los que recuperar la inversión. Existen diferentes estrategias que permiten rentabilizar las aplicaciones. *ColeccionApp* se centra en dos fórmulas fundamentales para conseguirlo: *Publicidad e Ítem selling*.

En primer lugar se pretende incorporar publicidad a la aplicación a través de banners. El objetivo es generar ingresos mediante impresión y clics de banners publicitarios.

Para llevar a cabo esta empresa, se hará uso de *iAd*, la plataforma de publicidad de Apple. La cantidad que puede llegar a pagar *iAd* oscila entre 0,217 euros y 2,17 euros el *eCPM*, (*Coste Efectivo por cada Mil Impresiones*). Este indicador proporciona información de cuanto se está ganando con un banner por cada 1.000 veces que se muestra. Si el *eCPM* es de 1,50 euros significa que por cada 1.000 veces que se muestre la publicidad, se obtiene 1,50 euros de beneficio. (STARTCAPPS)

La otra fórmula a implementar para obtener beneficios es el *Ítem selling*. El concepto *Ítem selling* consiste en la compra de objetos y mejoras dentro de un juego mediante dinero virtual. Éste dinero virtual podrá ser conseguido mediante logros dentro del juego o comprado con dinero real. En el caso de *ColeccionApp*, podría intercambiarse por cromos. Otra alternativa sería implementar un sistema de vidas limitadas para poder superar los retos que el usuario afronta contra la máquina. De este modo, en caso de agotarse las vidas, podría comprarse un nuevo paquete de vidas y seguir jugando.

Llegados a este punto, es necesario calcular cuánto costaría recuperar los 22.673,57 euros invertidos en el proyecto. En lo que se refiere a la publicidad a través del banner, se va a estimar que la cantidad obtenida por cada mil impresiones será de 2 euros. Se pretende

recuperar la inversión en 3 años. Teniendo en cuenta que se espera que un usuario medio abra la aplicación al menos 7 veces al día, se ha calculado que sería necesario conseguir en torno a 1.500 usuarios. (Ver Figura 5).

¿Cuántos usuarios hacen falta para recuperar la inversión de 22.673,57 € ?
3 años = 1.095 días
Un usuario 7 Impresiones al día
Por cada 1000 impresiones 2 € de beneficio

$7 \text{ impresiones} / 1 \text{ día} \times 1.095 \text{ días} = 7.665 \text{ impresiones un usuario en 3 años}$

$2 \text{ €} / 1.000 \text{ impresiones} \times 7.665 \text{ impresiones} = 15,33 \text{ € de beneficio en 3 años por usuario}$

$1 \text{ usuario} / 15,33 \text{ €} \times 22.673,57 \text{ €} = 1.479,03 \approx 1.500 \text{ usuarios para recuperar la inver.}$

Figura 5. Recuperación de la inversión mediante banners

Con esta cantidad de usuarios se tendrían cubiertos los costes. Por fortuna, la publicidad mediante los banners no será la única fuente de ingresos. También se cuenta con la venta de artículos o *Ítem Selling*. La idea es vender paquetes de intentos para que cuando el usuario se quede sin intentos pueda continuar jugando. El precio de cada paquete de intentos será de 1,50 euros. Si 1 de cada 300 usuarios compra al día un paquete de intentos, suponiendo que la aplicación está siendo utilizada por 1.500 usuarios, en 3 años se obtendrían unos beneficios aproximados de 8.200 euros. Ver cálculos en la Figura 6.

¿Cuántos beneficios se obtendrían en 3 años?
3 años = 1.095 días
1.500 usuarios
1 de cada 300 usuarios al día compra al menos un paquete de intentos
Cada paquete de intentos cuesta 1,50 €

$1 \text{ paq} / 300 \text{ usuarios} \times 1.500 \text{ usuarios} = 5 \text{ paquetes de intentos al día}$

$5 \text{ paq} / 1 \text{ día} \times 1.095 \text{ días} = 5.475 \text{ paquetes de intentos en 3 años}$

$1,50 \text{ €} / 1 \text{ paq} \times 5.475 \text{ paq} = 8.212,5 \text{ € de beneficio en 3 años}$

Figura 6. Cálculo de beneficios mediante Item-selling

3 Antecedentes

Cuando surge la idea de una aplicación, surge también el temor, o cuando menos la inquietud, de saber si alguien ya la ha llevado a cabo. Es el momento de rastrear el mercado y cruzar los dedos para no encontrar materializada esa idea que hasta el momento se creía propia. Por suerte, a día de hoy, no se ha encontrado ninguna aplicación móvil que se acerque a lo que *ColeccionApp* pretende ofrecer. Sin embargo, no sería justo negar que tenga fuertes influencias de otras aplicaciones existentes en el mercado.

ColeccionApp se inspira en múltiples juegos basados en cartas coleccionables. Un ejemplo de ello es la aplicación *War of the Fallen*, juego de rol desarrollado por *Zynga Inc.* que permite batallas basadas en las propias características de las cartas, y cuyo objetivo es conseguir el mayor número de cartas posibles. Además, también permite batallas tanto contra la máquina como entre usuarios. (© 2013 Zynga)

Otra aplicación que se asemeja es *Equipazo Virtual 2013-14*. Es una aplicación de la empresa *Panini* y está vinculada a la colección de cromos real. En realidad la aplicación es un complemento de la colección original, ya que únicamente consta de veinte cromos, uno de cada equipo de la liga, que se consiguen mediante los sobres que se venden en las tiendas y establecimientos habituales. El enfoque es distinto, pero la esencia es la misma: coleccionar. (©2013 Google)



Figura 7. El Equipazo Virtual

<http://www.1mobile.com/equipazo-virtual-2013-14-890526.html>



Figura 8. War Of The Fallen.

<http://www.manzanamagica.com/zynga-estrena-war-of-the-fallen-para-ios/>

Cabe señalar que no se ha podido encontrar un gestor de colecciones como tal. Todas las aplicaciones localizadas giran en torno a la gestión de una sola colección.

En cuanto a la manera en la que se obtendrán los ingresos mencionada en la evaluación económica, se puede afirmar que se trata de un modelo de negocio muy presente en el mercado. Predomina entre las aplicaciones y juegos líderes en ingresos en los rankings de los mercados. Un ejemplo de ello es el juego '*CSR Racing*' que ha alcanzado los 12 millones de dólares al mes de ingresos mediante venta de coches y mejoras dentro del mismo

(APPLESFERA 2012) o *'Clash of clans'*, otro ejemplo de juego adictivo con 500.000 euros de ventas diarias (2013 VIDEOGAMING247 LTD).



Figura 10. CSR Racing



Figura 9. Clash Of Clans

<http://www.deconstructoroffun.com/2012/07/csr-racing-dissected.html>

<http://xombitgames.com/2013/11/clash-of-clans-como-conseguir-gemas-gratis>

El hecho de limitar el número de vidas no persigue únicamente que el jugador compre más, tiene un objetivo oculto, crear adicción. Un claro ejemplo de ello es el juego *'Candy Crush Saga'*. (Ver Figura 11 <https://itunes.apple.com/us/app/candy-crush-saga/id553834731?mt=8>.)

“Cuando un jugador pasa un nivel de *'Candy Crush Saga'* lo olvida, pero cuando pierde, tiene la urgente necesidad de solventarlo porque se le queda «clavado» en la cabeza. A esta teoría se le conoce como el *'Efecto Zeigarnik'*. Este efecto fue descubierto a principios de siglo por la psicóloga Bluma Zeigarnik tras observar el comportamiento de los camareros. Estos podían recordar una gran cantidad de pedidos pendientes, sin embargo, una vez resueltos (servidos) los pedidos, los olvidaban. La psicóloga experimentó en el laboratorio con un grupo de voluntarios a los que proponía unos 20 retos mentales, de los cuales la mitad serían interrumpidos. Al finalizar el experimento, los voluntarios recordaban estas tareas con más claridad que las que habían resuelto satisfactoriamente.”



Figura 11. Candy Crash

(DIARIO ABC, S.L).

Cuando a un jugador se le acaban las vidas, se quedan con el nivel a medias, pendiente de terminar. Esa es la idea que se quiere implantar en *ColeccionApp*. Partiendo de esta base, una idea que puede dar su fruto es que los usuarios puedan interactuar los unos con los otros enviándose vidas entre sí, y a su vez, que puedan invitar a jugar a sus contactos fomentando la participación y generando así más usuarios. A más usuarios, más beneficios. Una iniciativa diferente pero que aportará beneficios de forma indirecta a través de actividad social.

4 Captura de requisitos

4.1 Modelo de casos de uso

La aplicación tiene dos actores: el usuario y el administrador. Cada uno de ellos tiene un rol muy definido. El administrador es el encargado de la gestión mediante una aplicación de escritorio, mientras que el usuario es el que hace uso del juego a través de móvil. Debido a ello, las funcionalidades que le corresponden a cada uno son bien distintas. Para representarlas en un diagrama se utilizan los modelos de casos de uso. (Ver Figura 12 y Figura 13).

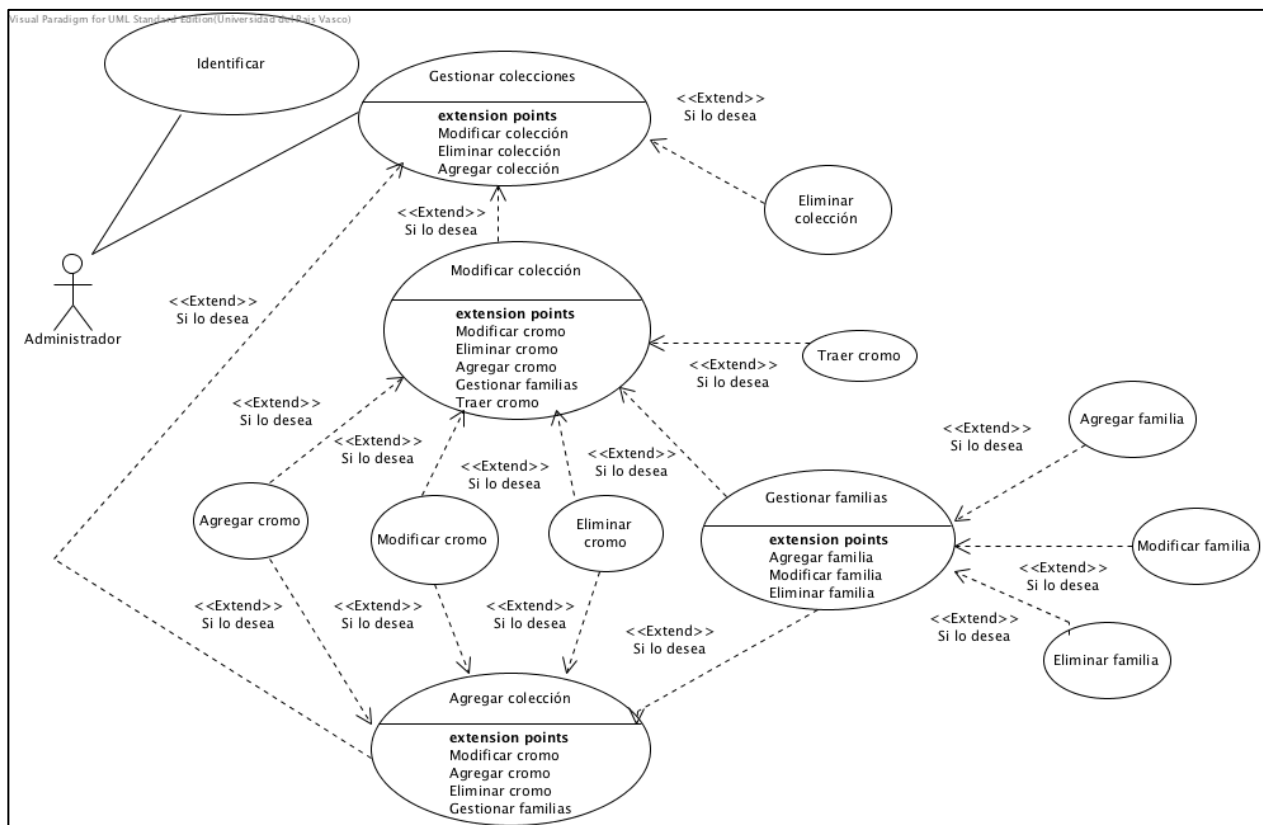


Figura 12. Casos de uso del Administrador

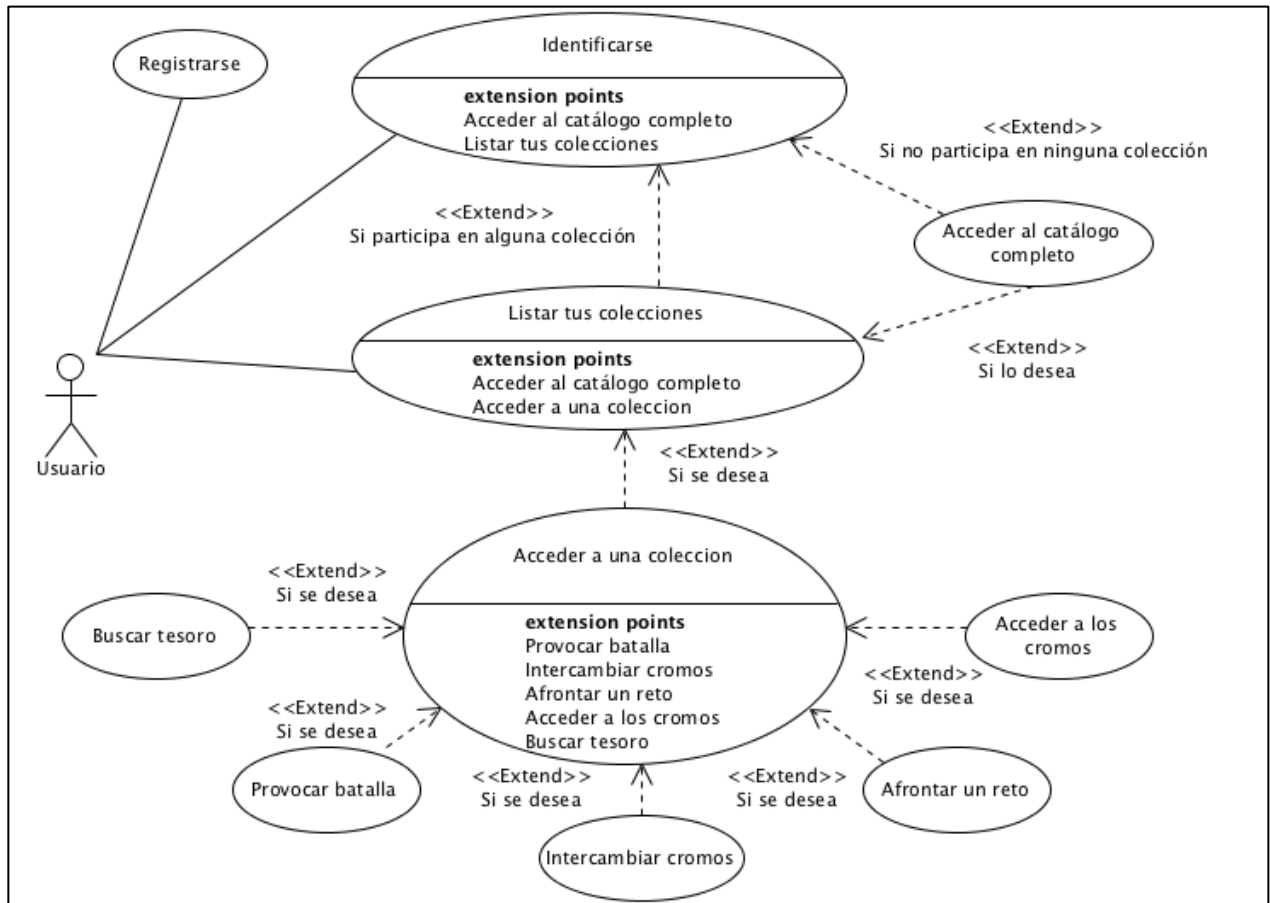


Figura 13. Casos de uso del Usuario

Dado que las dos partes están tan diferenciadas, ni el usuario ni el administrador tienen funcionalidades comunes. Por tanto, la *jerarquía de actores* es bastante simple. Se representan los actores de forma separada, sin ningún nexo que los una, se puede ver en la Figura 14.

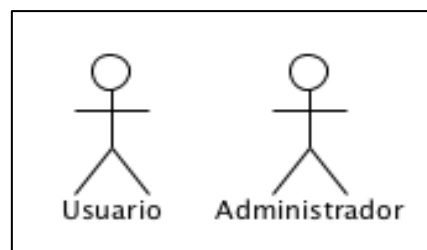


Figura 14. Jerarquía de actores

4.2 Casos de uso extendidos (Anexo)

Los casos de uso extendidos presentan cada una de las funcionalidades descritas en el modelo de casos de uso de manera mucho más precisa, ayudándose de interfaces gráficas y descripciones detalladas. Describen el proceso completo de las funcionalidades teniendo en cuenta todas las opciones posibles que se pueden dar. Debido al tamaño que abarcan en cuanto a espacio de documentación se refiere, este apartado es presentado aparte en forma de anexo.

4.3 Modelo de dominio

El modelo de dominio refleja el tratamiento de los datos de la aplicación.

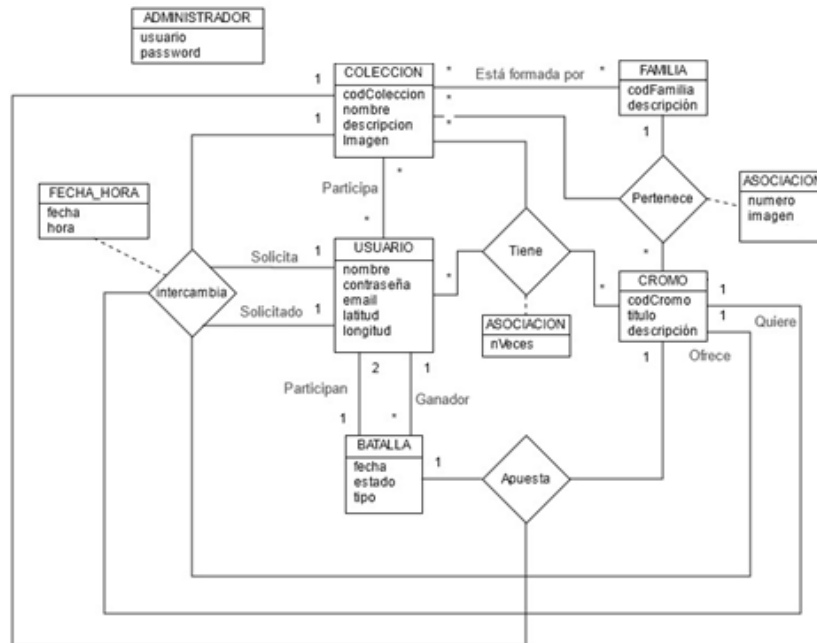


Figura15. Modelo de dominio

Está formado por las siguientes entidades:

- ADMINISTRADOR: administrador de la aplicación. No tiene una relación directa con ninguna otra entidad.
- USUARIO: usuarios que utilizan la aplicación. Tiene un nombre único de usuario, una contraseña, un email, unas coordenadas de referencia de latitud y longitud.
- COLECCIÓN: representa una colección. Cada colección tiene un código, un nombre, una descripción y una imagen.
- FAMILIA: se trata de una categoría que permite realizar agrupaciones dentro de una colección. Sus atributos son un código identificativo y una descripción.
- CROMO: cromo perteneciente a una o varias colecciones. Un cromo tiene un código, un título y una descripción.
- BATALLA: representa una lucha entre usuarios. Toda batalla ha de tener una fecha, un estado que puede ser pendiente o finalizada, y un tipo que indica el tipo de batalla.

Las relaciones que completan el modelo de dominio se explican a continuación.

- Participa: relación **N a M** entre las entidades USUARIO y COLECCIÓN.
 - Un usuario puede participar en varias colecciones y en una colección pueden participar varios usuarios.
- Está formada por: relación **N a M** entre las entidades FAMILIA y COLECCIÓN.
 - Una familia puede pertenecer a varias colecciones y una colección puede estar compuesta por varias familias.
- Participan: relación **2 a 1** entre las entidades USUARIO y BATALLA.
 - En una batalla participan dos usuarios.
- Ganador: relación **1 a 1** entre las entidades USUARIO y BATALLA.
 - En una batalla hay un único usuario ganador.
- Pertenece: relación múltiple compuesta por las entidades CROMO, FAMILIA, y COLECCIÓN. Representa un cromó de una colección que tiene una familia asociada.
 - Un cromó de una familia puede formar parte de más de una colección.
 - Un cromó que pertenece a una colección solo puede tener asociada una familia.
 - Una familia que forma parte de una colección puede estar vinculada a más de un cromó.

Además, esta combinación tiene un número de cromó, una imagen y un atributo que indica si se trata de un cromó especial.

- Tiene: relación múltiple compuesta por las entidades CROMO, COLECCIÓN y USUARIO. Representa los cromos que tiene un usuario en posesión.
 - Un cromó de una colección puede pertenecer a más de un usuario.
 - Un cromó que pertenece a un usuario puede formar parte de más de una colección.
 - Un usuario que participa en una colección puede poseer más de un cromó.

Además, esta combinación tiene un atributo nVeces para indicar el número de veces que el usuario tiene un cromó repetido.

- Intercambia: relación múltiple compuesta por las entidades CROMO, COLECCIÓN y USUARIO. Representa un intercambio de cromos que hace un usuario con otro.
 - Un Intercambio entre dos usuarios de dos cromos solo puede producirse si los cromos pertenecen a la misma colección.
 - Un usuario puede solicitar el intercambio ofreciendo un cromo de una colección por un cromo que quiere de otro usuario de esa misma colección.
 - A un usuario se le puede solicitar el intercambio pidiéndole un cromo por un cromo ofrecido por otro usuario. Los dos cromos pertenecen a la misma colección.
 - Solo puede ser ofrecido un cromo de una colección por otro cromo solicitado de la misma colección entre dos usuarios.
 - Solo puede ser solicitado un cromo de una colección por otro cromo ofrecido de la misma colección entre dos usuarios.

Además, se guardara la fecha y la hora en la que se produjo el intercambio.

- Apuesta: relación múltiple compuesta por las entidades CROMO, COLECCIÓN y BATALLA. Representa el cromo que obtiene el usuario ganador de la batalla.
 - En una batalla se obtiene un cromo de una colección.
 - Un cromo que se obtiene en una batalla pertenece a una colección.

5 Análisis y Diseño

5.1 Transformación del Modelo de dominio a BBDD

ColeccionApp tiene una parte importante de base de datos. De hecho, tal y como figura en la arquitectura se utilizan dos bases de datos que comparten el mismo modelo de dominio, (Ver *Arquitectura*). Para poder empezar a trabajar sobre una base de datos primero es necesario realizar una transformación basándose en el modelo de dominio de la aplicación. Lo que se hace es transformar las entidades y las relaciones en tablas dentro de la base de datos.

Las entidades que forman parte del modelo se transforman en tablas directamente con sus atributos como campos de la misma.

- ADMINISTRADOR
- USUARIO
- COLECCIÓN
- FAMILIA
- CROMO
- BATALLA

También es necesario transformar sus relaciones.

- Relaciones N a M
 - Participa --> Colecciones_Usuario
 - Está formada por --> Familia_Coleccion
- Relaciones múltiples
 - Pertenece --> Cromo_Fam_Coleccion
 - Tiene --> Cromos_Usuario
 - Intercambia --> Intercambio
 - Apuesta --> Apuesta
 - Tesoro --> Tesoro
- Para el caso de la relación *Ganador* y *Participan*, se ha establecido una relación directa. entre las tablas USUARIO y BATALLA.
 - Ganador, relación directa entre USUARIO_2 y BATALLA
 - Participan, relación directa entre USUARIO, USUARIO_1 y BATALLA

Una vez se tienen las tablas, se normaliza la base de datos. Se asignan las claves privadas y foráneas y se establecen las relaciones entre sí. El diseño final se muestra a continuación, en la Figura 16.

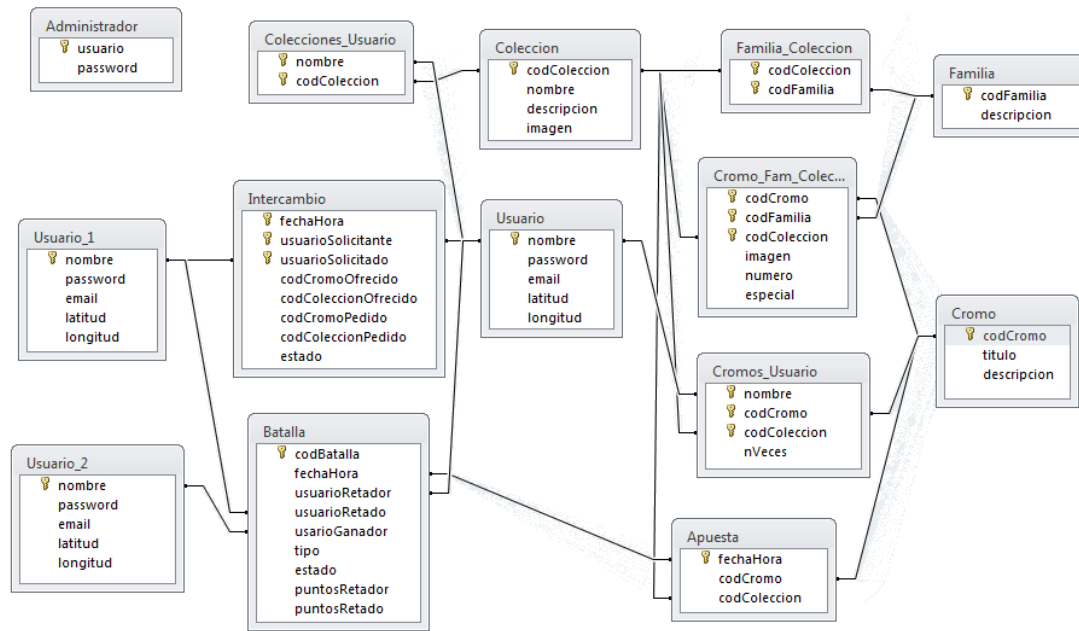


Figura 16. Diseño de base de datos.

La base de datos alojada en el servidor está definida en *MySQL* y es la que da soporte a todos los usuarios. En ella quedan incluidas todas las tablas del modelo. Por otro lado la base de datos alojada en el propio dispositivo es definida en *SQLite* y únicamente da soporte al usuario identificado en el dispositivo. Las tablas que no forman parte del modelo local son *Colecciones_Usuario*, *Cromos_Usuario* y *Administrador*.

5.2 Diagrama de clases

En este apartado se muestra el diagrama de clases utilizado para el desarrollo de las aplicaciones del administrador y del usuario. Para simplificar se han incluido únicamente los atributos y métodos más relevantes de cada clase.

5.2.1 Aplicación del administrador

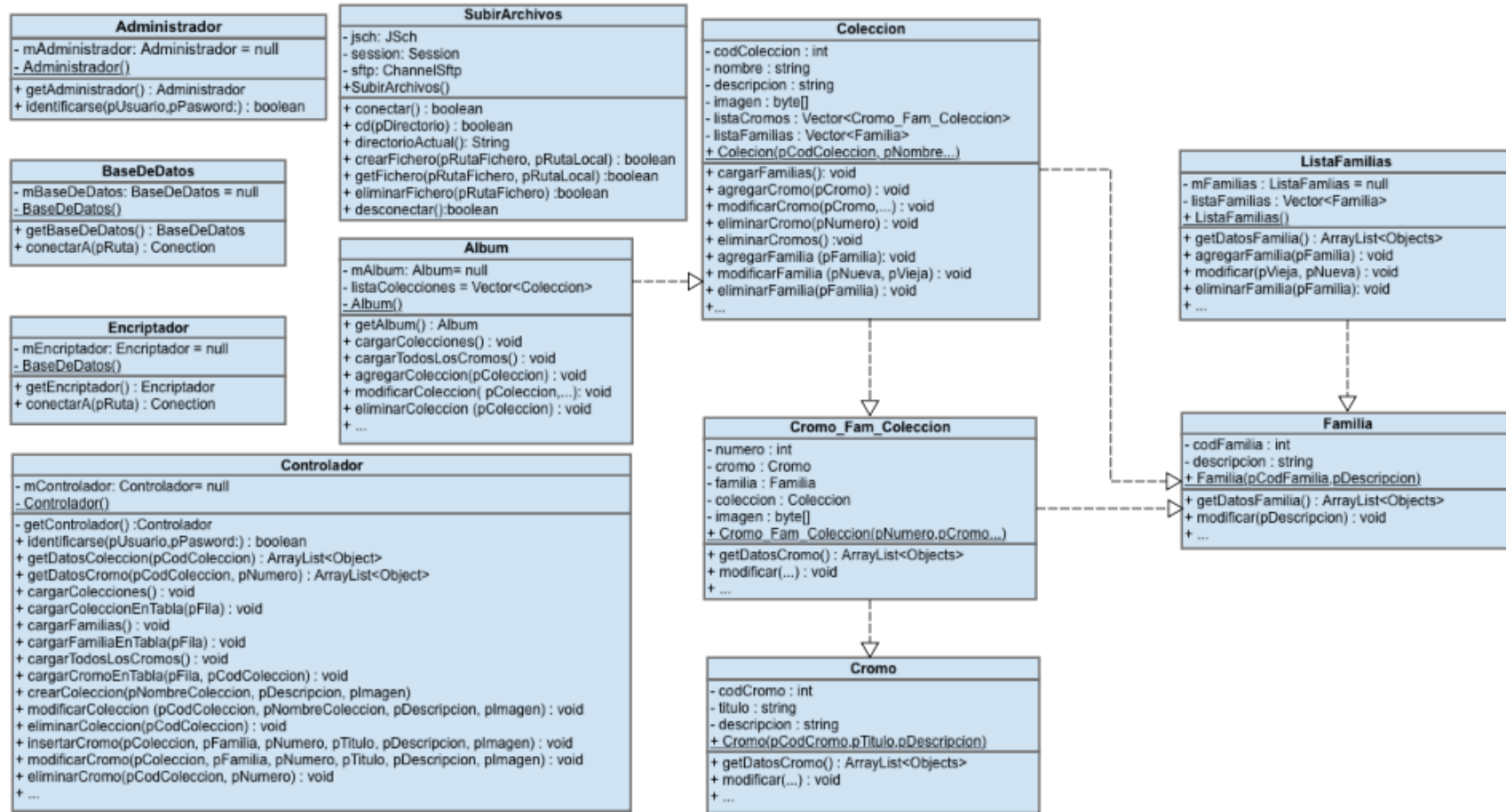


Figura 17. Diagrama de clases del administrador

A continuación se ofrece una descripción breve de cada una de las clases:

- Administrador: clase que no comparte una relación directa con ninguna otra del modelo. Esta implementada mediante el patrón *Singleton* y se emplea para realizar la identificación del administrador y empezar así con la gestión de las colecciones.
- BaseDeDatos: clase implementada mediante el patrón *Singleton* que se utiliza para establecer la conexión con el servidor *MySQL*.
- Encriptador: clase implementada mediante el patrón *Singleton* que se utiliza para aplicar un cifrado de datos *MD5* a la contraseña del administrador.
- SubirArchivo: clase que permite gestionar conexiones *SFTP*. Permite conectarse a un servidor *SFTP*, cambiar de directorio, crear ficheros, eliminar ficheros y descargar ficheros.
- Album: clase implementada mediante el patrón *Singleton*. Representa el conjunto de todas las colecciones que existen en la aplicación. Su principal atributo es *listaColecciones*, y se encarga de guardar una lista de instancias de la clase *Coleccion*. Entre sus funciones principales se encuentran: cargar todas las colecciones; cargar todos los cromos; agregar, modificar y eliminar colecciones.
- Coleccion: clase que representa a una colección. Sus atributos se componen por los mismos campos que tiene esta tabla en la base de datos y además una lista de cromos, compuesta por instancias de la clase *Cromo_Fam_Coleccion*, y una lista de familias compuesta por instancias de la clase *Familia*. Entre sus funciones principales se encuentran la agregación, modificación y eliminación de cromos y familias.

Una particularidad de esta clase es que, a diferencia de la base de datos donde en el campo imagen se guarda la ruta que la imagen tiene en el servidor, en la clase *Coleccion* el atributo imagen es de tipo *byte []* y guarda la imagen en bytes.

- Cromo_Fam_Coleccion: representa un cromo completo. Es decir, un cromo que ya forma parte de una familia y de una colección, y que además, tiene asignado un número y una imagen para esa colección. Es por ello que esos son precisamente los atributos que maneja esa clase. Su función principal es la de devolver los datos que la describen. Al igual que sucede con la clase *Coleccion*, en la clase *Cromo_Fam_Coleccion*, la imagen se guarda en bytes.
- Cromo: representa un cromo reusable. Se compone de los datos que no cambian de un cromo, independientemente de la colección a la que pertenezca. El código, el título y la descripción de un cromo siempre serán los mismos. Su función principal es la de devolver los datos que la describen.
- ListaFamilias: clase implementada mediante el patrón *Singleton*. Representa el conjunto de todas las familias que existen en la aplicación. Es una forma sencilla de

consultar todas las familias existentes. Entre sus funciones principales destacan la agregación, modificación y eliminación de las familias que la componen.

- Familia: clase que representa una familia. Sus atributos son el código de la familia y su descripción. Como funciones ofrece la posibilidad de modificar y devolver los datos de la familia.
- Controlador: clase implementada mediante el patrón *Singleton*, para que sea accesible desde cualquier vista. Dentro del modelo-vista-controlador representa al controlador. Se encarga de hacer de intermediario entre las clases y las vistas. Cualquier dato que requiera del modelo para mostrarse en alguna de las vistas pasa primero por esta clase. Es por ello que tiene un número elevado de métodos.

5.2.2 Aplicación del usuario

Mientras que en la aplicación del administrador se utiliza una única clase *Controlador* que se encarga de enlazar cada una de las vistas con las clases que forman el modelo, en la aplicación del usuario y debido a la estructura del lenguaje *Objective-C* es necesario implementar un controlador por cada una de las vistas generadas. *ColeccionApp* se nutre principalmente de las tablas de las bases de datos, es por ello que la mayor parte del diagrama de clases del usuario está formado por entidades tipo *ViewController*. Únicamente se ha necesitado crear tres clases de datos adicionales para la implementación. En base al criterio *MVC* y para diferenciar las clases controladoras de las clases de datos se han representado con diferentes colores. Las clases controladoras son las clases representadas con fondo azul, mientras que las clases de datos se muestran con fondo verde.

Además, se ha generado una clase adicional de fondo rojo para representar de forma simbólica las vistas que forman parte de la aplicación. Al haberse utilizado una clase *Main.storyboard* para generar las vistas, no hay ficheros de imagen como tal que puedan ser representados en el diagrama. En este momento puede resultar algo confuso, pero el funcionamiento de esta clase se explicará más adelante en el apartado 6.3.3. *Main.storyboard*.

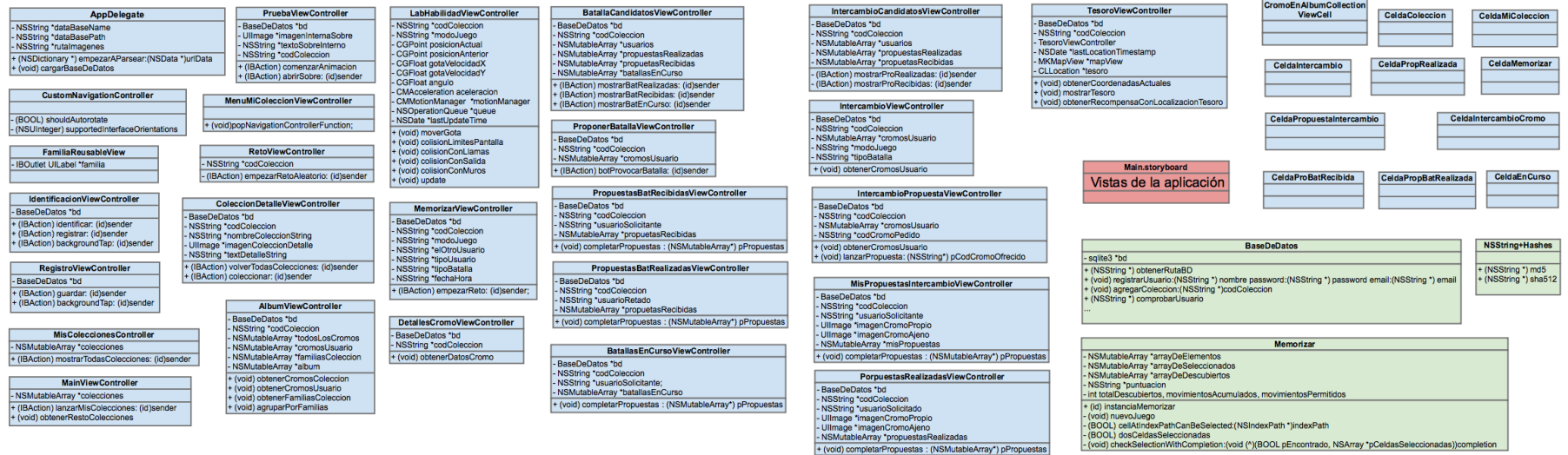


Figura 18. Diagrama de clases del usuario

A continuación se ofrece una descripción breve de cada una de las clases:

- `AppDelegate`: sólo debería haber una por aplicación. Es donde se especifica el código a ejecutar dependiendo del evento que lance la aplicación en general, como por ejemplo, cuando la aplicación comienza, cuando termina, cuando se pone en reposo, etc.

Los archivos `ViewController` son donde se define el código que “controla” cada una de las vistas de la aplicación. Hay uno por cada vista y es donde se especifica lo que hace cada elemento de la propia vista.

- `IdentificaciónViewController`: su función principal es gestionar el proceso de identificación en la base de datos del servidor. Controla la vista donde el usuario introduce sus datos de acceso para poder identificarse o pulsar el botón ‘Registrarse’ si aún no lo está.
- `RegistroViewController`: su función principal es gestionar el proceso de registro en la base de datos del servidor. Controla la vista donde el usuario introduce los datos para poder realizarlo.
- `MainViewController`: su función principal es cargar las colecciones en las que no participa el usuario. Controla la vista donde se muestra el listado de las mismas.
- `ColeccionDetalleViewController`: clase que muestra en una vista la imagen, el nombre y la descripción de la colección seleccionada en la vista controlada por la clase `MainViewController`. Su función principal es iniciar el proceso en el que un usuario empieza una colección.
- `MisColeccionesViewController`: similar a la clase `MainViewController` su función es cargar las colecciones en las que participa el usuario. Controla la vista donde se muestra un listado de las mismas. También gestiona el proceso que permite a un usuario eliminar una colección.
- `PruebaViewController`: es la clase que controla la vista donde se muestra una pequeña animación en la que sale un sobre de una caja. Además de la animación también se encarga de insertar un cromó aleatorio en la colección del usuario.
- `MenuMiColeccionViewController`: es una clase de tipo `UITabBarController`. No tiene una vista asociada como tal. Su función es gestionar la navegación por las diferentes pestañas que se muestran en la barra inferior cuando un usuario accede a una colección. (*Album, Reto, Batalla, Intercambio y Tesoro*)
- `AlbumViewController`: es la clase controladora de la vista donde se muestra el álbum virtual del usuario agrupado por familias. Por ejemplo, en la colección de la liga, los cromos del Athletic estarán agrupados en la misma sección. Su función principal es cargar todos los cromos y mostrar aquellos que el usuario posee. Si

todavía no tiene el cromó, muestra un hueco con el número del cromó correspondiente.

- **FamiliaReusableView:** no es una clase controladora. Es una vista que se puede reusar en las clases de colección, *UICollectionReusableView*. Se inserta dentro de la vista del álbum y se utiliza tantas veces como familias tenga la colección. De este modo se le da un encabezado a cada sección del álbum. Por ejemplo, para el mismo caso de la colección de la liga, los cromos que pertenezcan a la familia Athletic, tendrán la palabra 'Athletic' como encabezado.
- **DetallesCromoViewController:** es una clase más que nada informativa y no tiene excesiva funcionalidad. Controla la vista que muestra los detalles del cromó cuando el usuario lo selecciona desde su álbum virtual. Tales como, el número, el nombre, la imagen, descripción, familia y también el número de veces que el usuario tiene el cromó y cuantas veces lo tiene bloqueado.
- **RetoViewController:** clase muy sencilla cuya única función es llevar al usuario a uno de los dos retos de forma aleatoria cuando éste quiere iniciar un reto.
- **LabHabilidadViewController:** es la clase controladora del juego de habilidad. Sus funciones principales son controlar los movimientos de la gota y las llamas, gestionar las colisiones de la gota con muros, límites de la pantalla, llamas y la salida.
- **MemorizarViewController:** es la clase que controla el juego de memoria. Entre sus funciones se encuentra, controlar el estado de cada celda cada vez que el usuario las toca. Se encarga de mostrar, mantener, ocultar o aplicar un fondo verde según corresponda. También determina el resultado del juego según el propio progreso del jugador en el mismo. Puede tener dos funciones distintas. En base al contenido del atributo *modoJuego* actúa de diferente manera. Si se trata de una batalla, se tendrán en cuenta las puntuaciones y se comprobará si se ha ganado la batalla, si se ha perdido o si está aún sin determinar. Si no se trata de una batalla, al criterio utilizado para discernir el éxito del fracaso será el límite de movimientos permitidos. Controlar el funcionamiento de esta vista para cada caso es tarea de esta clase.
- **BatallaCandidatosViewController:** clase controladora de la vista que muestra un listado de usuarios a los que poder proponer una batalla. Además, se encarga de comprobar si el usuario ha recibido, realizado alguna propuesta de batalla o si hay alguna batalla ya en curso. Si es así, muestra u oculta para cada caso el botón correspondiente.
- **ProponerBatallaViewController:** Su función principal es la de gestionar el proceso de proponer una batalla a un usuario. Controla la vista en la que se muestran los cromos del usuario candidato, y un selector donde se puede escoger el tipo de batalla a proponer, pudiendo ser 'aleatoria' o 'selectiva'.

- **BatallasEnCursoViewController**: clase que controla la vista en la que se muestran las batallas en curso. Una batalla en curso tiene diferentes estados dependiendo si el usuario ha ganado la batalla, la ha perdido o aún está sin determinar. Además, incluso habiendo ganado o perdido, según el orden en el que se haya producido el desenlace, el estado de la celda también cambia. La función de esta clase es gestionar el estado que le corresponde a cada batalla en curso en cada caso.
- **PropuestasBatRecibidasViewController**: clase que controla la vista en la que se muestran las propuestas de batalla recibidas. Ofrece la posibilidad de aceptar o rechazar la batalla. Se encarga de gestionar el proceso que corresponda en cada caso.
- **PropuestasBarRealizadasViewController**: clase que controla la vista en la que se muestran las propuestas de batalla que se han realizado. Si la batalla aún no ha sido aceptada por el otro usuario permite anularla, en caso de haber sido ya aceptada ofrece la opción de empezar. Su función principal es gestionar esos procesos en cada caso.
- **IntercambioCandidatosViewController**: clase controladora de la vista que muestra un listado de usuarios a los que poder proponer un intercambio. Además, se encarga de comprobar si el usuario ha recibido, realizado alguna propuesta de intercambio. Si es así, muestra u oculta un botón para cada caso.
- **IntercambioViewController**: controla la vista en la que se muestran el álbum virtual del oponente junto con el número de veces que ese usuario tiene cada cromo. Al igual que la clase *MemorizarViewController* puede tener dos funciones distintas. En base al contenido del atributo *modoJuego* actúa de diferente manera. Por ejemplo, si se trata de una batalla, ésta vista se utiliza para que el usuario obtenga un cromo de su oponente. Si no, es utilizada para elegir un cromo de su oponente y proponer un intercambio. Gestionar este proceso, es la tarea principal de esta clase.
- **IntercambioPropuestaViewController**: controla la clase que muestra una vista parecida al álbum virtual propio. La salvedad es que muestra también un número que indica el número de veces que el usuario tiene ese cromo junto con el cromo. Esta vista sirve al usuario para proponer un cromo como intercambio. Esta clase se encarga de gestionar este proceso.
- **MisPropuestasIntercambioViewController**: clase que controla la vista en la que se muestran las propuestas de intercambio recibidas. Ofrece la posibilidad de aceptar o rechazar el intercambio. Se encarga de gestionar el proceso que corresponda en cada caso.
- **PropuestasRealizadasViewController**: clase que controla la vista en la que se muestran las propuestas de intercambio que se han realizado. Si el intercambio aún no ha sido aceptado por el otro usuario permite anularlo. Su función principal es gestionar este proceso.

- **TesoroViewController:** esta clase controla la vista donde se muestra el mapa. Es la clase encargada de mostrar la ubicación del usuario en cada momento, así como de situar el tesoro a una distancia relativamente cercana a la posición del usuario en el mapa. Se encarga también de actualizar la posición del usuario e ir comprobando la distancia entre ambas coordenadas para determinar si ha encontrado el tesoro.
- **Celdas:** son clases que controlan celdas. Las celdas pueden formar parte de una tabla o de una colección. Aunque tienen una configuración propia, son instanciadas desde la clase controladora de la vista donde están incluidas, y es responsabilidad de ésta clase la de controlar el funcionamiento de las mismas. Por ejemplo, la celda intercambio tiene asociada un botón para aceptar y otro para rechazar. Sin embargo, le corresponde a la clase *MisPropuestasIntercambioViewController* implementar el evento resultante de la acción de pulsar un botón. En cualquier caso, el funcionamiento de las celdas se explicará en el apartado 6.3.6. `UITableView`.

Las clases que se muestran a continuación son las que completan el modelo de datos:

- **BaseDeDatos:** es la clase encargada de establecer las conexiones con la base de datos de la aplicación. Cada vez que una clase controladora quiere insertar algo en la base de datos local lo hace instanciando primero esta clase. De este modo queda separados los accesos a la base de datos local de los de la base de datos del servidor.
- **NSString+Hashes:** su única función es la de realizar un encriptado de los datos que se le pasen como parámetro. Los cifrados que permite son md5 y sha512.
- **Memorizar:** clase que determina la estructura del juego de memoria. Entre sus funciones más importantes se encuentran la de iniciar el juego, mezclar los elementos para que queden de distinta forma cada vez que se inicia un nuevo juego, comprobar si un elemento ha sido descubierto, comprobar si ha habido una coincidencia entre elementos o comprobar si se ha resuelto el juego.

5.3 Diagramas de secuencia (Anexo)

Los diagramas de secuencia permiten modelar la interacción entre objetos en un sistema según UML (*Lenguaje Unificado de Modelado*). Al igual que los casos de uso extendidos, los diagramas de secuencias se presentan en un anexo aparte.

6 Desarrollo

La implementación de *ColeccionApp* se empezó a desarrollar por la parte de la base de datos, después se realizó la aplicación del administrador para finalmente concluir con el desarrollo de la aplicación móvil. Se eligió implementarla en este orden ya que es un proceso que permite ir comprobando progresivamente el correcto funcionamiento de la aplicación a medida que ésta va creciendo. Se pretende ofrecer una visión cronológica del desarrollo.

6.1. Base de datos

Para desarrollar el trabajo sobre la base de datos se utilizó una base de datos *MySQL*. La primera versión del diseño de la base de datos era algo diferente a la versión final. A lo largo del desarrollo del proyecto se ha necesitado realizar algunas modificaciones debido a problemas que han ido surgiendo. Se explicará cual era el diseño inicial y, cuáles han sido los cambios más significativos y por qué ha sido necesario aplicarlos.

En el diseño original en la tabla *Cromo*, además de la descripción, figuraban también los campos imagen y número. Teniendo en cuenta que la imagen del cromó va ligada a la familia, con este diseño no era posible utilizar dos imágenes diferentes del mismo cromó. Es decir, si se quería crear un mismo cromó en dos colecciones donde cada una de ellas perteneciese a diferentes familias, el sistema obligaba a generar el mismo cromó dos veces. Por ejemplo, se crea una nueva colección Liga de fútbol y se quieren aprovechar los cromos de la colección del año pasado. Se tiene que añadir el cromó del hasta ahora jugador del Athletic Laporte que ha fichado por el Barça, no tiene ningún sentido que en la imagen siga teniendo la camiseta del Athletic. Hará falta otra imagen con la indumentaria de su nuevo equipo.

El planteamiento inicial de aprovechar el cromó creado se venía abajo. Por otro lado era interesante que el número que el cromó tuviera en la colección variara de un año a otro, por tanto tampoco tenía sentido fijar un número a un cromó de forma definitiva. La solución fue llevar esos dos atributos (imagen y número) de la clase *Cromo* a la tabla *Cromo_Fam_Coleccion* que es donde realmente se crea el cromó que indica a que familia y a que colección pertenece. De esta manera cada cromó puede tener una imagen y un número diferente para cada colección sin necesidad de crear otro cromó.

Otro cambio muy significativo ha sido la manera de almacenar las imágenes en la base de datos. En el primer diseño el campo imagen de las tablas *Coleccion* y *Cromo_Fam_Coleccion* era de tipo *blob*, es decir, la imagen se almacenaba en la base de datos directamente. Teniendo en cuenta este diseño se implementó la aplicación del administrador. Tras varias peleas, se logró que desde la aplicación del administrador se guardara una imagen en la base de datos, una pérdida de tiempo como se verá a continuación. En cualquier caso, aunque en el desarrollo final no se incluya, en su momento formó parte del desarrollo de la aplicación y será explicado. Sin embargo, para no mezclar la parte de base de datos y la parte del desarrollo del administrador, la explicación detallada del proceso completo que

se realizaba desde que se seleccionaba una imagen hasta que ésta era almacenada en la base de datos del servidor se dará más adelante en el apartado 6.2.5 Tratamiento de imágenes.

Una vez se tenía toda esta parte desarrollada, se empezó con la aplicación del usuario. Tal y como se explica en el apartado 2.2 Arquitectura, la aplicación móvil tiene su propia base de datos *SQLite* que comparte el mismo modelo de dominio que la base de datos de la aplicación. Fue entonces cuando surgió la necesidad de utilizar un servidor accesible desde internet. Se empezó a utilizar un servidor *MySQL* proporcionado por la UPV-EHU. En él simplemente se importó la base de datos que ya se tenía implementada en el servidor local. Con todo ello se empezó con el desarrollo de la aplicación del usuario.

Aunque se explica en la arquitectura y se detallará más adelante en el apartado de desarrollo del usuario, es importante mencionar que para obtener los datos del servidor se utiliza *JSON*. Se vio que había muchos problemas para extraer las imágenes de la base de datos a través de *JSON*. Imágenes que habían sido previamente insertadas mediante la aplicación del administrador. Tras darle muchas vueltas, tristemente se decidió que la mejor opción era cambiar la manera en la que se guardaban las imágenes en el servidor. El campo imagen de las tablas *Coleccion* y *Cromo_Fam_Coleccion* pasó de ser de tipo *blob* a ser de tipo *varchar*, de modo que a partir de esta modificación la imagen se guardaría en una carpeta del servidor, mediante *SFTP*, y en la base de datos se guardaría la ruta de la imagen. Este proceso se explicará también más adelante.

Tristemente porque fue un inconveniente importante que retraso de forma considerable la fase de desarrollo. Había que modificar la aplicación del administrador. Fue necesario cambiarla para que en lugar de guardar las imágenes en la base de datos lo hiciera en una carpeta del servidor proporcionado, y que en la base de datos guardaría la ruta correspondiente a dicha imagen. Finalmente, se consiguió mediante conexiones *SFTP*.

6.2 Administrador

6.2.1. Estructura y componentes

Es una aplicación de escritorio desarrollada en Java, en este caso se ha utilizado el entorno de desarrollo Eclipse, aunque podría haberse utilizado otro compatible.



Figura 20. Eclipse



Figura 19. Java

Java es uno de los lenguajes de programación más usados, particularmente para aplicaciones cliente-servidor. La aplicación del administrador tiene su propia arquitectura interna basada en el patrón *Model-View-Controller*. Se trata de una arquitectura de software que separa los datos y la lógica de negocio de la interfaz de usuario. Permite el desarrollo por separado de ambos aspectos, y fomenta la reutilización de código, características que otorgan flexibilidad al desarrollador, facilitan las tareas de desarrollo y hacen más sencillo su posterior mantenimiento. Su estructura se compone de tres componentes principales: el *modelo*, la *vista* y el *controlador*.

En el caso de *ColeccionApp* se podría decir que se trata de un modelo híbrido, ya que los datos están incluidos en la en la base de datos pero también en las clases de objetos que se han creado en java para facilitar el manejo de los mismos.

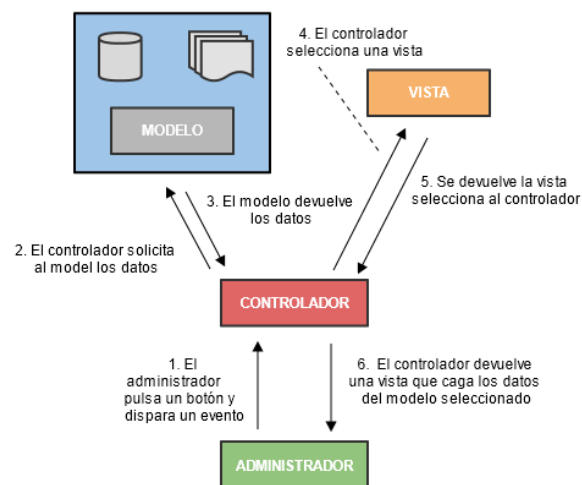


Figura 21. Modelo híbrido. <http://www.glify.com/>

El modelo envía aquella parte de la información que en cada momento se le solicita para que sea mostrada. Las peticiones de acceso o manipulación de información llegan al modelo a través del controlador, que es el encargado de incluir los datos que el modelo le devuelve para que sean mostrados en la vista.

Para tener una mejor organización de las vistas, el modelo y el controlador, se han generado dos paquetes, *packModelo* donde se encuentran las clases donde se implementa la lógica de negocio y *packVistas* donde se encuentran las vistas y el controlador.

El entorno de desarrollo Eclipse no tiene por defecto vistas, por lo que es necesario instalar el plugin *WindowsBuilder* para poder crear interfaces gráficas mediante *Swing* simplemente arrastrando los componentes sobre la vista.

Estructura de los componentes Java Swing.

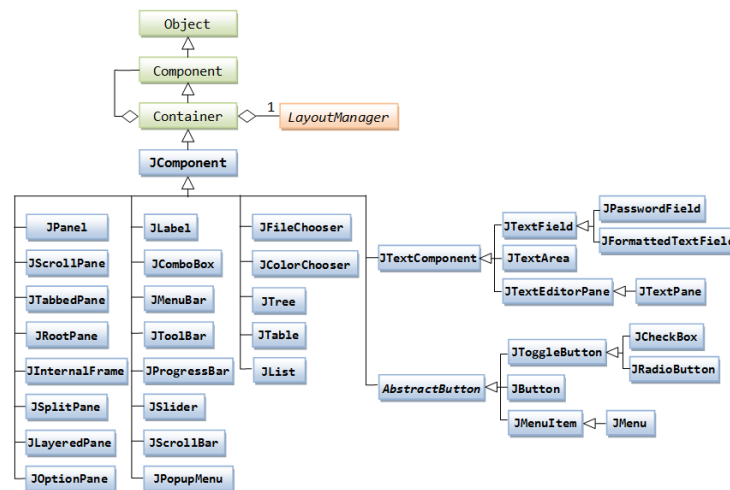


Figura 22. Componentes Java Swing

http://www.ntu.edu.sg/home/ehchua/programming/java/I4a_GUI_2.html#zz-1.

Las dos tipos de vistas principales que proporciona Java son *JFrame* y *JDialog*. Las diferencias más significativas entre ambas son que un *JDialog* admite otra ventana (*JFrame* o *JDialog*) como padre en la constructora, mientras que *JFrame* no admite padres. Además, Un *JDialog* puede ser modal, cosa que un *JFrame* no. Por estas razones se ha decidido que la ventana principal sea un sea *JFrame* y las secundarias sean *JDialog*.

Tanto las aplicaciones *AWT* como las aplicaciones *Swing* utilizan las clases de tratamiento de eventos *AWT* incluidas en la librería *java.awt.event*. *Swing*, además, añade algunos nuevos eventos incluidos en la librería *javax.swing.event*, aunque la mayoría no se usan habitualmente.

En la Tabla 39 se muestran los componentes de la interfaz gráfica que generan eventos a través de acciones que el usuario lleva a cabo.

Acción del usuario	Evento disparado	Interfaz Event Listener
Click a Button, JButton	ActionEvent	ActionListener
Open, iconify, close Frame, JFrame	WindowEvent	WindowListener
Click a Component, JComponent	MouseEvent	MouseListener
Change texts in a TextField, JTextField	TextEvent	TextListener
Type a key	KeyEvent	KeyListener
Click/Select an item in a Choice, JCheckbox,	ItemEvent,	ItemListener,
JRadioButton, JComboBox	ActionEvent	ActionListener

Tabla 39. Componentes interfaz gráfica.

Las subclases de *AWTEvent* son las siguientes:

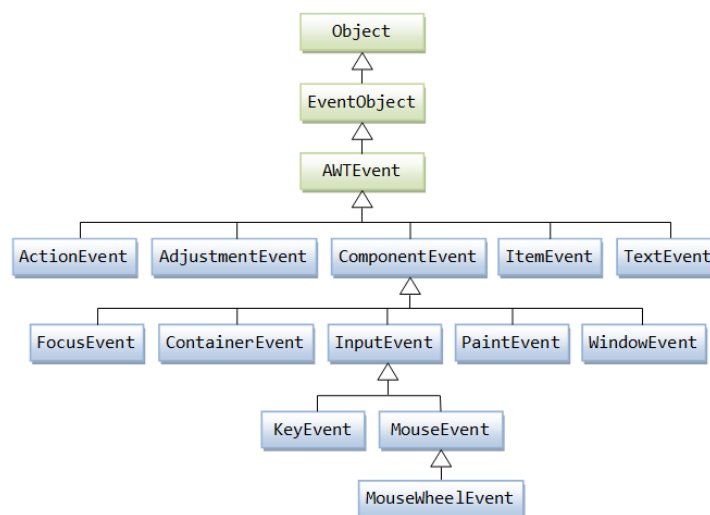


Figura 23. Subclases AWTEvent,

http://www.ntu.edu.sg/home/ehchua/programming/java/J4a_GUI_2.html#zz-1.

Todos los botones de la aplicación del administrador utilizan el evento *ActionEvent* -> *ActionPerformed*. Cuando un botón es presionado se dispara y se llama a la clase *Controlador.java*. Se utiliza una sola clase controladora, así que se ha implementado con el patrón Singleton. La clase controladora tiene acceso a las clases del modelo, y se encarga de, a través de diferentes métodos, llevar hasta la vista los datos que esta le requiera.

Por ejemplo, para el caso de inicio de sesión del administrador cuando el administrador pulsa el botón 'Entrar' se lanza el evento que llama al controlador para que gestione el inicio de sesión.

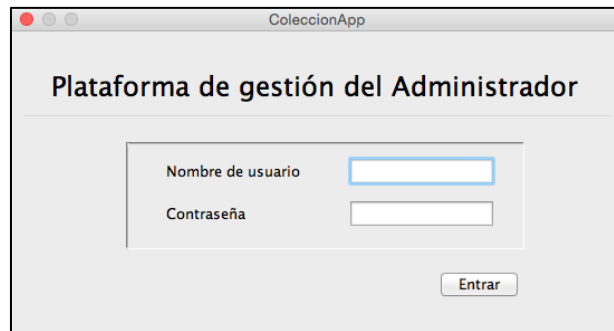


Figura 24. Vista identificarse

Se crea una instancia de tipo Controlador y se le asigna la instancia única Controlador. Una vez se tiene instanciada la clase se puede acceder a sus métodos:

```
btnEntrar.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent arg0) {
        Controlador controlador = Controlador.getControlador();
        ...
        permitirAcceso = controlador.identificarse(textUsuario.getText(),
            passwordField.getText());
        ...
    }
})
```

En este caso ejecuta un método propio *identificarse* que lo que hace es llamar al método *identificarse* de la clase *Administrador.java* para obtener el resultado. El modelo, en este caso, la clase *Administrador.java* recibe la petición y ejecuta el método *identificarse*. El método determina si la identificación es correcta. Junto a cada método que vaya contra la base de datos se ha de incluir la declaración *throws SQLException* para poder tratar las excepciones SQL que se puedan producir.

```
public boolean identificarse(String pUsuario, String pPassword) throws SQLException{
    ...
}
```

6.2.2. Conexión con la base de datos

Todos los casos de uso del administrador requieren accesos a la base de datos. Se ha decidido implementar una clase llamada *BaseDeDatos.java* que se va a encargar de establecer todas las conexiones necesarias. Es una clase que tendrá una sola instancia accesible desde cualquier otra clase, así que se utiliza el patrón 'Singleton' para su desarrollo.

La constructora de esta clase y el método que la devuelve cuando es llamada desde otra de las clases se muestra en la Figura 25. Este proceso es idéntico para cada una de las clases del proyecto que adopten este patrón.

```

/**
 * Clase encargada de pasar la conexión a la base de datos.
 * @author Iván Revuelta Antizar
 */
public final class BaseDeDatos {

    // Atributos
    private static BaseDeDatos mBaseDeDatos = null;

    // Constructora
    private BaseDeDatos(){

    }

    // Métodos
    public static BaseDeDatos getBaseDaDatos(){
        if (mBaseDeDatos == null){
            mBaseDeDatos = new BaseDeDatos();
        }
        return mBaseDeDatos;
    }
}

```

Figura 25. Patrón Singleton

Una vez se tiene el esqueleto de la clase, se implementa el método que permite conectar las diferentes clases del modelo de datos con la Base de datos del servidor. El modelo de datos se conecta directamente con la base de datos a través de *JDBC* (*Java Database Connectivity*). *JDBC* es una API que permite la ejecución de operaciones sobre bases de datos desde el lenguaje de programación Java mediante lenguaje *SQL* del modelo de base de datos que se utilice. En este caso se está utilizando *MySQL*.

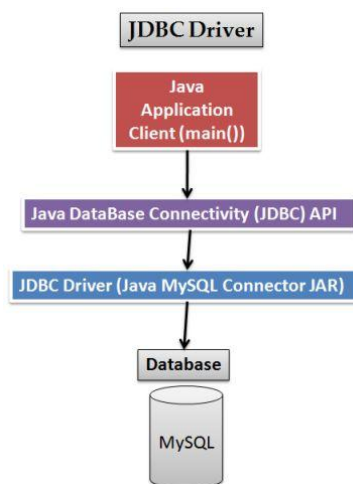


Figura 26. JDBC (Java Database Connectivity)

<http://theopentutorials.com/tutorials/java/jdbc/jdbc-oracle-connection-tutorial>

Se ha implementado un método *conectarA()* que se encarga de establecer la conexión utilizando los datos del servidor (*url*, *usuario* y *contraseña*). Establece la conexión y la devuelve.

```
public Connection conectarA(){
    ...
    DriverManager.getConnection("jdbc:mysql://galan.ehu.es/Xirevuelta005_coleccionapp",
    "Xirevuelta005", "*****");
    ...
    return conn;
}
```

Recuperando el ejemplo de la identificación, donde se necesita establecer una conexión con la base de datos. Se crea una instancia que sea de tipo *Connection* y se llama al método *conectar* de la clase *BaseDeDatos* instanciada previamente que le devuelve la conexión. Además se crea una instancia *Statement* que se encarga de ejecutar la sentencia SQL requerida y otra *ResultSet*, que guarda el resultado de la misma. Mediante el método *next()* de esa clase se va recorriendo el resultado de la consulta para comprobar los datos.

```
public boolean identificarse(String pUsuario, String pPassword) throws SQLException{
    ...
    Connection conexion = bd.conectarA();
    Statement sentencia = conexion.createStatement();
    ResultSet resultado = sentencia.executeQuery("select * from administrador");
    resultado.next();
    ...
    if (resultado.getString("usuario").compareTo(pUsuario) == 0){
        if (resultado.getString("password").compareTo(pPassword) == 0){
            permitirAcceso = true;
            ...
        }
    }
    ...
    return permitirAcceso;
}
```

6.2.3 Tratamiento de tablas

Uno de los objetos más empleados en la creación de la aplicación del administrador son las tablas. En ellas se listan las colecciones, los cromos y las familias que forman parte de la aplicación. De todos los componentes que forman parte de swing probablemente los *JTable* sean los componentes con APIs más extensas, la clase *JTable* tiene más de cien métodos.

Afortunadamente, esa variedad les permite ser también de los componentes *Swing* más personalizables y potentes. Para construir una simple tabla se podría usar simplemente la siguiente constructora que ofrece *JTable*:

```
JTable(Object[][] rowData, Object[] columnNames)
```

Permite construir una tabla a partir de dos parámetros; el primero de ellos; *rowData*, es un array bidimensional de objetos que representa el contenido de la tabla. El segundo; *columnNames*, representa los nombres de cada columna, contenidos también en un array que por lo general es un array de *strings*. Sin embargo, al empezar el desarrollo se comprobó que se presentaban las siguientes desventajas:

- La primera, es que para la construcción de la tabla se tienen que tener de antemano los datos que se desea que contenga la tabla ya sea en un array o en un vector, lo que le resta flexibilidad al llenado de la tabla.
- La segunda desventaja es que estos constructores hacen automáticamente que todas las celdas sean editables.
- Y la tercera, es que todos los datos contenidos en la tabla, son tratados como *strings*, incluso aunque los datos hayan sido declarados de otro tipo antes de pasárselos al constructor.

Por fortuna, estas tres desventajas pueden ser solventadas si se utiliza un *modelo de tabla*. Los modelos de tabla implementan la interfaz *TableModel*; a través de ellos es posible personalizar más y mejor el comportamiento de los componentes *JTable*, permitiendo utilizar al máximo sus potencialidades. Todas las tablas cuentan con un *modelo de tabla*. El modelo predeterminado es la clase *DefaultTableModel*. La Figura 27 intenta mostrar cómo cada componente *JTable* obtiene siempre sus datos desde un *modelo de tabla*.



Figura 27. Relación Modelo -> Vista

http://www.javahispano.org/contenidos/archivo/63/jtable_1.pdf

La mejor forma de apreciarlo es mediante un ejemplo. En la Figura 28 se muestra como queda la tabla de la vista colecciones.

Nombre	Descripción	Imagen
Dragon Ball Z	Serie de animacion.	
Liga BBVA 2013-2014	Liga de fútbol de la temporada 2013 - 2014.	
Campeones	Serie de dibujos	
Simpsons	Serie basada en el modelo de vida de una familia americana	
NBA	Temporada 14/15	

Figura 28. Tabla colecciones

Se ha implementado un modelo propio de tabla que extiende el modelo predeterminado *DefaultTableModel* y que solo utiliza dos de sus métodos; *isCellEditable* que evita que las celdas sean editables y *getColumnClass* que devuelve el tipo datos de una columna. Son precisamente los métodos que solucionan el problema mencionado previamente. Este modelo es utilizado por cada una de las tablas de la aplicación.

```
public boolean isCellEditable(int row, int column) {
    ...
}

public Class<?> getColumnClass(int columnIndex){
    ...
}
```

Para cada tabla se ha creado una vista aparte. Para el caso que ocupa, la clase creada es *TablaColecciones* que extiende la clase *JTable*. En la constructora de esta clase se instancia el modelo creado previamente y se especifica simplemente cuales son las columnas que va a adoptar esta tabla y la altura de cada fila:

```
public TablaColecciones() {
    ...
    String [] columnNames = new String[]{"Nombre", "Descripción", "Imagen"};

    Object [][] data = new Object[][]{};
    MyTableModel model = new MyTableModel(data, columnNames);
    this.setRowHeight(30);
    this.setModel(model);
    ...
}
```

Lo único que falta para completar el proceso, es instanciar ésta tabla desde la vista donde se pretende mostrar, en éste caso desde la clase *Colecciones*.

```
TablaColecciones table = new TablaColecciones();
```

Esta clase se ha implementado de modo que cuando se inicie la vista, se cargue la tabla con las colecciones existentes. Para ello, la vista llama al controlador que carga en un array de objetos las colecciones y las trae hasta la vista. Después recorre el array agregando cada uno de los datos de cada colección fila a fila en la tabla:

```
public void cargarColecciones() throws SQLException{

    DefaultTableModel modelo =(DefaultTableModel) table.getModel();
    Controlador con = Controlador.getControlador();
    ...
    for (int i=0;i<cuantasColecciones;i++){
        datosColeccion = con.cargarColeccionEnTabla(i);
        ...
        Object nuevo[]= {datosColeccion.get(1),datosColeccion.get(2), iconoImagen};
        pModelo.addRow(nuevo);
    }
    ...
}
```

6.2.4 Explorador de ficheros y validación

Cada uno de los objetos tiene una serie de atributos que son fáciles de tratar, nombre de una colección, descripción, número de un cromó, etc. Este tipo de campos son rellenados en un cuadro de texto y luego basta con recogerlos. Sin embargo, tanto las colecciones como los cromos tienen asociada una imagen, con lo que hace falta hacer algo para poder explorar los ficheros del equipo y poder recoger la imagen deseada. (Ver Figura 29).

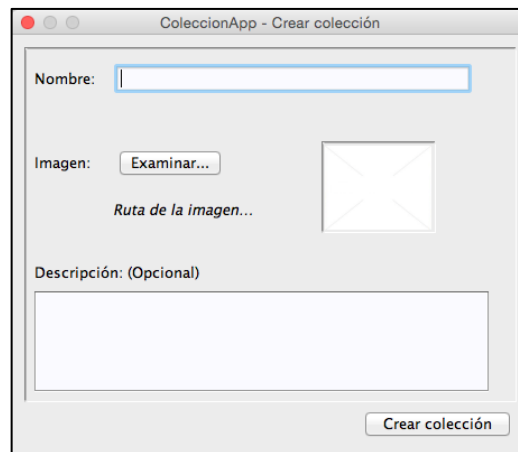


Figura 29. Crear Colección

Gracias al componente Swing *JFileChooser* es posible llevarlo a cabo. Al pulsar el botón se dispara el evento *actionPerformed*. En él, se declara un nuevo objeto de tipo *JFileChooser*. Como únicamente interesa que recoja archivos de tipo imagen se aplica el siguiente filtro:

```
private FileNameExtensionFilter filter = new FileNameExtensionFilter("Archivo de imagen", "jpg", "jpeg", "png");
```

Se le indica que únicamente interesan los ficheros cuya extensión sea *jpg*, *jpeg* o *png*. Una vez aplicado el filtro se abre la ventana de diálogo que permite escoger un fichero. Cuando el usuario selecciona la imagen se guarda el nombre del fichero y su ruta. Además, se ha querido ofrecer una vista previa de la imagen en una etiqueta, así que el siguiente paso es transformar la imagen para poder mostrarla.

La clase *ImageIcon* permite recuperar la imagen a partir de su ruta. Se crea una nueva instancia de este tipo y se le pasa la ruta como parámetro para que obtenga la imagen. Como la imagen ha de tener una serie de medidas acordes al tamaño de la etiqueta creada, es necesario modificarla. Para ello, hace falta utilizar la clase *Image*. Se crea una instancia *Image* y se le asigna la imagen desde la clase *ImageIcon* mediante el método *getImage()*. Una vez hecha esta asignación se crea una nueva imagen partiendo de la original pero modificando su tamaño:

```
Image nuevaImagen = imagen.getScaledInstance(95, 75, java.awt.Image.SCALE_SMOOTH);
```

Una vez se tiene la imagen del tamaño deseado, se crea una nueva instancia del tipo *ImageIcon* a la que se le pasa como parámetro la imagen modificada. Por último, se inserta el icono en la etiqueta y se le da el mismo tamaño que la imagen. A continuación se presenta la parte más relevante del código descrito.

```
private JButton getBtnExaminar() {
    ...
    btnExaminar.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent arg0) {
            ...
            dlg = new JFileChooser();
            dlg.setFileFilter(filter);
            String nombreImagen = dlg.getSelectedFile().getName();
            String fil = dlg.getSelectedFile().getPath();
            lblFoto.setIcon(new ImageIcon(fil));
            ImageIcon icon = new ImageIcon(fil);
            Image imagen = icon.getImage();
            nuevaImagen = imagen.getScaledInstance(95, 75, java.awt.Image.SCALE_SMOOTH);
            ImageIcon newIcon = new ImageIcon(nuevaImagen);
            lblFoto.setIcon(newIcon);
            ...
            cargarImagenValidar()
        }
    });
    ...
    return btnExaminar;
}
```

Al final del código se observa un método *cargarImagenValidar()*. Este método simplemente añade un tic verde a una etiqueta que esta oculta y que se utiliza en señal de aprobación, indicando que la imagen se ha cargado correctamente. Se ha implementado así debido a que el atributo imagen es un campo obligatorio. Esto es algo que se hace para cada uno de los campos obligatorios al crear o modificar una colección o un cromó. (Ver Figura 29).



Figura 29. Validación de los campos

Al pulsar el botón *Crear colección* se lanza un método que comprueba si cada uno de los campos obligatorios está validado y, si lo está, permite la creación de la colección.

6.2.5 Tratamiento de imágenes

En el punto anterior se ha indicado como se extrae una imagen de un fichero y como se muestra en una etiqueta. A continuación se explica cómo se guardaba la imagen en la base de datos directamente. Como se menciona en el apartado de base de datos esta forma de almacenar la imagen ya no forma parte del proyecto. Sin embargo, dado que en su día lo hizo y se implementó, se ha creído conveniente incluirlo en la documentación.

En este caso el formato del campo imagen en la base de datos era de tipo *blob*, es decir, que la imagen se guardaba en bytes. La clase *File* permite obtener el fichero a partir de su ruta. La clase *FileInputStream* obtiene los bytes de lectura de un fichero. Se instancia una variable de ese tipo y se le asigna el contenido de la imagen. Se crea una instancia *imagenEnBytes* de tipo *byte[]* cuya longitud sea igual a la del fichero. Finalmente se carga la variable con el contenido.

```
File imagen = new File(lblRutaImg.getText());
FileInputStream entrada = null;
entrada = new FileInputStream(imagen);
byte[] imagenEnBytes = new byte[(int) imagen.length()];
entrada.read(imagenEnBytes);
```

Una vez se tiene la variable de tipo *byte[]* se pasa como un parámetro más para que el controlador llame al método correspondiente del modelo que inserta los datos en la base de datos. Por ejemplo, cuando se crea una colección, el controlador llamaba al método *crearColeccion* de la clase *Album*.

En primer lugar se crea la conexión con la base de datos y después la sentencia de inserción SQL. Se incluyen las variables recibidas como parámetro según su tipo. En el caso del nombre y la descripción de la colección se utiliza el método *setString*. Para el caso de la imagen basta con utilizar *setBytes* en su lugar.

```
public void crearColeccion(String pNombreColeccion, String pDescripcion, byte[] pImagen)
throws SQLException{
    ...
    // Insertando datos

    PreparedStatement prep = conexion.prepareStatement("insert into coleccion
(nombre, descripcion, imagen) values(?,?,?);");

    prep.setString(1, pNombreColeccion);
    prep.setString(2, pDescripcion);
    prep.setBytes(3, pImagen);
    prep.execute();
    ...
}
```

Este proceso funcionaba correctamente, cabe reseñar que tuvo que ser modificado por la dificultad de extraer la imagen de la base de datos. A continuación se detalla la manera definitiva de almacenar las imágenes.

Se utiliza el método *crearColeccion*. Este método recibe como parámetro los datos de la colección a insertar en la base de datos y la ruta de la imagen. Al igual que el método antiguo, se genera una conexión con la base de datos solo que, en este caso, además se genera una instancia de tipo *SubirArchivos*. En el apartado de arquitectura se menciona que para realizar la transferencia de imágenes se utiliza una conexión *SFTP*. Para ello se ha creado una clase *SubirArchivos* que implementa los métodos necesarios para realizar esa transferencia, (Ver Diagrama de clases).

Lo primero que se hace es introducir la imagen en el servidor partiendo de la ruta recibida por parámetro. Se llama al método *conectar* de la clase instanciada *SubirArchivos* para realizar la conexión.

JSch es una implementación de *SSH2* que permite establecer una conexión a un servidor *SFTP*. Se genera una instancia de ese tipo y se le pasa como parámetro los datos de acceso y el puerto correspondiente, en este caso el puerto 22. Este tipo de protocolo requiere un intercambio de claves así que se le asigna una propiedad que le indica que acepte la conexión sin pedir confirmación. Por último es necesario abrir el canal por el que se establecerá la conexión y establecerla, ver el código a continuación:

```
public boolean conectar(){
    ...
    // Conectar al servidor FTP por el puerto 22
    jsch = new JSch();
    session = jsch.getSession("irevuelta005", "galan.ehu.es", 22);
    session.setPassword("KAuUpa4mxm");

    // El protocolo SFTP requiere un intercambio de claves al asignarle esta propiedad se
    // le dice que acepte la clave sin pedir confirmación
    Properties prop = new Properties();
    prop.put("StrictHostKeyChecking", "no");
    session.setConfig(prop);
    session.connect();

    // Se abre el canal de sftp y conectamos
    sftp = (ChannelSftp) session.openChannel("sftp");
    sftp.connect();

    ...
    return connect;
}
```

Este método determina si se ha establecido o no la conexión. Es necesario importar las siguientes librerías:

```
import com.jcraft.jsch.ChannelSftp;
import com.jcraft.jsch.JSch;
import com.jcraft.jsch.JSchException;
import com.jcraft.jsch.Session;
import com.jcraft.jsch.SftpException;
```

Una vez establecida la conexión hay que ubicarse en el directorio donde se quiere que se guarde la imagen y crearla. Gracias al método *cd(directorio)* de la clase *SubirArchivos* es

posible. Este método simplemente llama al método del canal que lo sitúa en el directorio que se le pasa por parámetro:

```
sftp.cd(directorio);
```

Para crear la imagen se utiliza el método *crearFichero(rutaFicheroSftp,rutafichero)* de la clase. Se le pasa la ruta donde se encuentra la imagen y la ruta donde se quiere crear el fichero. Para ello utiliza el método *put()*. *CrearFichero* devuelve un boolean que indica si se ha creado el fichero correctamente.

```
sftp.put(rutaFichero, rutaFicheroSftp);
```

Para finalizar se desconecta del servidor mediante el método *desconectar*.

```
public boolean desconectar(){
    sftp.exit();
    sftp.disconnect();
    session.disconnect();
    return true;
}
```

El resto de campos se introducen directamente en la base de datos tal y como se hacía antes con la particularidad de que en lugar de la imagen, ahora se guarda su ruta:

```
prep2.setString(3,"http://galan.ehu.es/irevuelta005"+sftp.directorioActual()+"/"+pNombre
Imagen);
```

6.2.6 Tratamiento de datos

A lo largo del desarrollo de la aplicación del administrador han ocurrido varios problemas. Errores de concepto, de implementación o diseño. A continuación se describen al detalle los problemas más significativos y, por ende, que más han retrasado el desarrollo de la aplicación, y la forma en la que se les han dado solución.

Antes del comienzo del desarrollo, la aplicación del administrador estaba basada única y exclusivamente en un modelo de dominio transformado a base de datos. La idea era traer todos los datos de golpe desde la base de datos a la interfaz del administrador.

Al empezar a programar empezaron a surgir problemas. Tratar de arrastrar desde una consulta a la base de datos de una sola vez toda la lista de colecciones existentes resultaba tedioso y complicado. Cada una de las tablas tenía que ser insertada en memoria a través de matrices, Array de Arrays, etc. Tras varios intentos fallidos surgió la idea de simplificar y modularizar haciendo uso de las clases que java proporciona.

La solución pasaba por crear una estructura combinada entre un modelo de base de datos y un modelo orientado a objetos. Así que eso fue lo que se hizo.

Con este nuevo planteamiento, la aplicación se ejecuta de la siguiente manera: cuando arranca la aplicación toda la información de la base de datos se carga en las clases de java. Para mostrar los datos en pantalla, y una vez que las clases ya tienen los datos cargados,

mediante el modelo vista controlador la interfaz gráfica se nutre de la información contenida en las clases en lugar de 'atacar' directamente a la base de datos. Cuando el administrador introduce alguna colección, familia o cromos nuevo, el procedimiento es parecido. Primero se insertan/modifican los nuevos registros en la base de datos y a continuación se vuelven a cargar las clases de java para que los cambios se puedan ver reflejados en pantalla al momento.

Cuando se inicia la vista que contiene la tabla colecciones se cargan todas las colecciones con sus cromos y sus familias en las clases de java mediante los métodos *cargarColecciones()*, *cargarFamilias()* y *cargarTodosLosCromos()*. Después mediante el controlador se llevan los datos cargados de las clases a las vistas para que sean mostrados donde corresponda.

Sin embargo, cuando se soluciona un problema a menudo surge otro. En este caso cuando el administrador introducía una nueva colección en la base de datos, volvían a cargarse todas las colecciones, y no únicamente la nueva, por lo que en la lista de colecciones mostrada en la interfaz, las colecciones aparecían repetidas. Por fortuna para solucionarlo bastó con controlar la existencia de la colección en la lista de colecciones cuando los datos se cargan de la base de datos a las clases de java. Este mismo problema y su solución, se puede hacer extensible al resto de clases (familias, cromos, etc.).

Sin embargo, el error más grave que se cometió durante la fase de desarrollo fue generar las clases de java partiendo del modelo de base de datos en lugar de hacer la transformación orientada a objetos. Por ejemplo, la clase colección en lugar de contener una lista de objetos tipo Cromo, tenía simplemente como atributos los mismos campos que la tabla colección tenía en la base de datos. Con lo cual, todas las ventajas que proporciona la programación orientada a objetos quedaba desperdiciada. Cuando se cayó en el grave error, la aplicación estaba bastante avanzada, lógicamente hubo que cambiar la estructura de las clases y, con ello, muchos de los métodos que ya no se adecuaban a la nueva estructura y no tenían nada que ver con el modelo orientado a objetos. El impacto fue importante y retrasó la planificación temporal establecida.

6.3 Usuario

6.3.1. Introducción

Tal y como se menciona en la introducción, se trata de una aplicación móvil desarrollada en *iOS*, mediante el lenguaje de programación *Objective-C*.



Figura 30. iOS

<http://www.hkbutterfly.org/post/apple-ios-9-release-date-specs-and-features/>

Objective-C es un lenguaje de programación orientado a objetos que tiene su origen en el lenguaje C. De hecho, cada clase comparte la misma estructura de archivos que C. Por un lado se tiene un archivo de cabecera que posee el sufijo `.h`. En él, únicamente se realizan las declaraciones de los métodos. Éstos pueden ser de clase si llevan un símbolo '+' delante, o de instancia si el símbolo es '-'. Los métodos de clase no pueden ser accesibles desde las instancias de la clase. El otro tipo de archivo es el de implementación cuyo sufijo es `.m`, es aquí donde se escribe el código de implementación de cada método.

A diferencia del lenguaje Java que puede ser implementado en múltiples entornos de programación como Eclipse, Netbeans, etc., *Objective-C* tiene su propio entorno de desarrollo llamado *Xcode*.



Figura 30. Xcode & Objective-C

http://www.acmegalicia.com/curso_ios.html

XCode se suministra de forma gratuita por Apple junto con Mac OS X. Proporciona un diseño de interfaz de usuario unificado, codificación, pruebas y depuración, todo ello en una sola ventana. Permite programar, compilar, depurar y optimizar las distintas aplicaciones desarrolladas en él. Incluye múltiples herramientas que facilitan la labor del desarrollador. A continuación se mencionan las más relevantes:

- *iPhone Simulator*, permite probar las aplicaciones sin necesidad de utilizar un dispositivo *iOS* real. Permite realizar simulaciones de diferentes tipos de dispositivos y de diferentes versiones del sistema operativo de *iOS* como muestra la Figura 34. Permite modificar la orientación del dispositivo, imitar varios gestos táctiles, donde intervienen uno, dos o varios dedos.

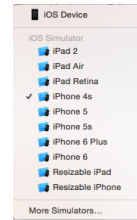


Figura 31. Tipos de dispositivos

El *Simulador de iOS* también tiene limitaciones.. No se pueden instalar aplicaciones de la App Store. No posee acelerómetro por lo que no se puede comprobar si el juego de habilidad funciona desde aquí. Tampoco se puede utilizar la cámara ni la localización GPS. Aunque sí que se puede simular una localización introduciéndola manualmente para comprobar el correcto funcionamiento de la parte del tesoro.

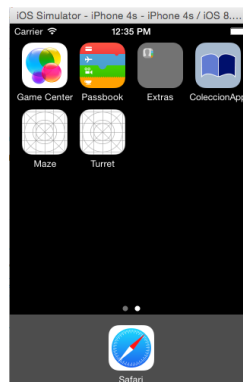


Figura 32. Simulador iOS

- *Interface Builder*, herramienta que permite diseñar visualmente las interfaces gráficas de las aplicaciones. En la versión actual, está totalmente integrado como un lienzo de diseño del IDE de Xcode mediante la clase *MainStoryboard*. Esta clase se explica más adelante.
- *Instruments*, conjunto de herramientas que permiten analizar y optimizar el código midiendo distintas características de rendimiento como el uso de la CPU, la memoria consumida, la creación y destrucción de objetos, y el uso de disco; mostrando gráficas que se actualizan en tiempo real.

6.3.2. Frameworks

Apple ofrece un gran número de frameworks para que la implementación de propiedades en diferentes plataformas sea uniforme. Los frameworks se agrupan en capas, los de las

capas superiores se construyen a partir de los ubicados en capas inferiores. (Ver Figura 33)

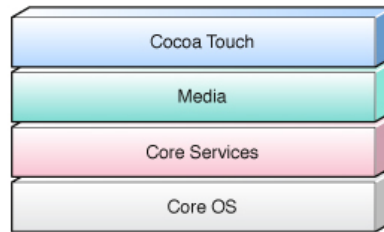


Figura 33. Las capas de los frameworks iOS

<http://jcampohucn.blogspot.com.es/2013/05/arquitectura.html>

La capa superior es *Cocoa Touch*, se utiliza para controlar aspectos comunes de las aplicaciones iOS, como eventos de proceso, eventos táctiles, gestuales, multihilo, compatibilidad con el mapa y acelerómetro. Todos los proyectos *XCode* que se crean de una plantilla estándar para aplicaciones incluyen *CoreGraphics*, *Foundation* y *UIKit*.

Las clases que tienen el prefijo *UI* forman parte de *UIKit*. A lo largo del apartado de desarrollo del usuario se irán viendo cuales son las utilizadas en el desarrollo de *ColeccionApp*.

A continuación se listan algunos ejemplos más de frameworks incluidos en *Cocoa Touch*.

- *Core Animation*, permite crear animaciones compuestas de capas gráficos independientes, partiendo de un sencillo modelo de programación.
- *Core Audio*, permite reproducir, procesar y grabar audio añadiendo potentes y sencillas características a la aplicación
- *Core Data*, proporciona un flexible y potente modelo de datos para construir aplicaciones basadas en el modelo vista controlador.



<https://developer.apple.com/technologies/ios/cocoa-touch.html>

6.3.3. Main.storyboard

Antes de empezar con la explicación del desarrollo conviene sentar las bases de lo que es el storyboard de la aplicación. Principalmente para ir introduciendo algunos conceptos que serán mencionados más adelante.

Antes de la llegada de *iOS5*, las interfaces se guardaban en archivos *nib*. Los storyboard son una de las novedades que se presentan a partir de esta versión.

Los storyboards permiten ver todas las pantallas y las conexiones que hay entre ellas. Un storyboard está compuesto por escenas y secuencias. Cada escena contiene una interfaz de usuario de una única vista, y es el principal componente visual. Las secuencias aparecen como flechas dibujadas entre las escenas. Existen tres tipos de secuencias: *Modal*, *Push* y *Custom* (Modales, de empuje y personalizadas). Las modales se utilizan para presentar contenido modal permitiendo asignar un estilo a la transición. Las secuencias *Push* son las que más se han utilizado en la aplicación. Se utilizan junto con un controlador de navegación para deslizar una nueva escena en la pantalla. Una escena *Custom* permite especificar el estilo que se utilizará en la aplicación.

El controlador de navegación es una clase que proporciona Apple para gestionar la presentación entre vistas desde la aplicación. Esta clase es *UINavigationController* y controla una jerarquía de clases *UIViewController* tal y como se muestra en la Figura 34.

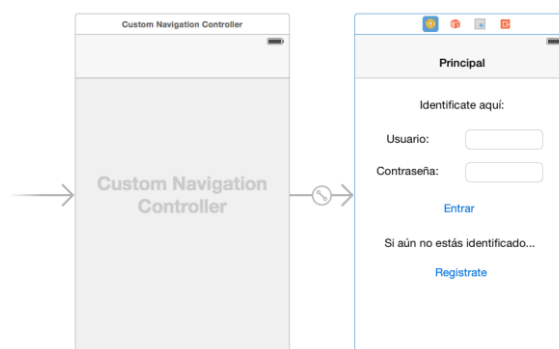


Figura 34. UINavigationController y UIViewController

Para definir una transición de forma gráfica basta con arrastrar el puntero del ratón con el *ctrl* pulsado de una escena a la otra. Se mostrarán las opciones de transición disponibles. (Ver Figura 35).



Figura 35. Transiciones entre vistas

En el caso de *ColeccionApp*, debido a la necesidad de pasar parámetros entre vistas la mayor parte de las transiciones se han realizado a través de código. Esa es la razón por la que en el *Main.Storyboard* no se muestren demasiadas secuencias. (Ver Figura 36).

6.3.4. Estructura y creación de las vistas en ColeccionApp

El primer paso para la creación de la aplicación móvil a través de XCode es crear un proyecto nuevo. XCode proporciona una serie de plantillas a escoger. En este caso para empezar desde cero se escogió una *Single View Application*. Este tipo de plantilla genera una única escena que representa la interfaz de usuario de esta vista. Además de la escena creada creará dos conjuntos de ficheros: *AppDelegate* (.h y .m) y *identificacionViewController* que es el nombre que se le ha dado para esta vista inicial.

La pantalla que muestra XCode se compone de varias partes. A la izquierda de la ventana se muestra el navegador donde se encuentra el listado de todos los archivos del proyecto. En la parte inferior se encuentra el área de debug, y en la parte derecha se encuentra el área de utilidades.

A medida que se van generando escenas en el *MainStoryboard*, es necesario también ir creando las clases *ViewController* que irán asociadas a cada una de ellas. La forma de diseñar una escena es muy sencilla. El diseño se realiza a través de la librería de objetos proporcionada en la parte inferior derecha de la ventana.

Para agregar una vista simplemente basta con arrastrar un objeto tipo *ViewController* al storyboard. De la misma forma, se van añadiendo las etiquetas, botones, cuadros de texto, etc. que sean requeridos para cada una de las escenas. La estructura de XCode se puede ver en la Figura 37.

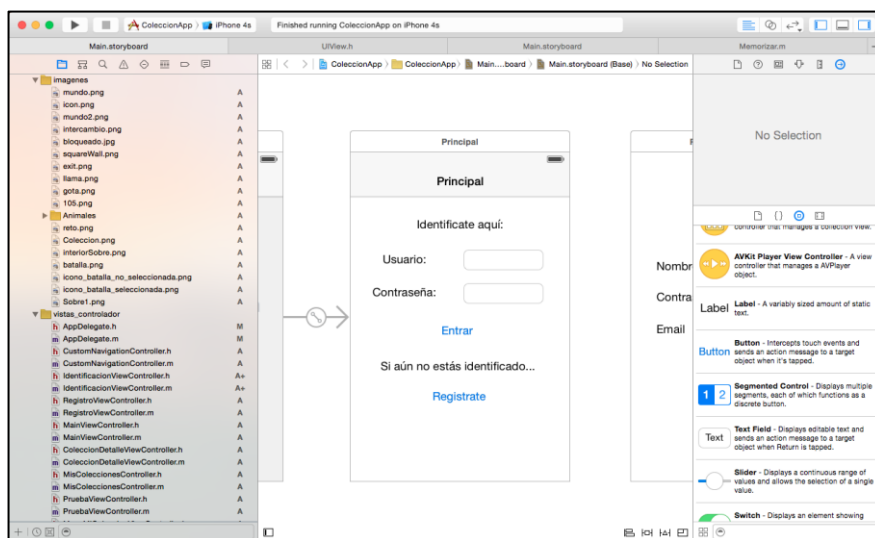


Figura 37. Estructura de la interfaz de Xcode

Una vez se tiene una vista diseñada, es necesario crear los archivos de clase que harán la función de controlador de esa vista. Para ello se selecciona *File > New File*, en la ventana emergente escoger *Cocoa Touch Class*.

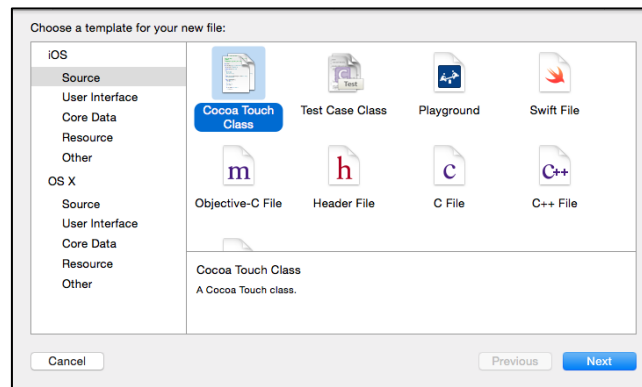


Figura 38. Nueva clase Cocoa Touch

En la siguiente ventana se le da un nombre y se elige el tipo de subclase que se corresponda con la vista a la que se quiere asociar. Por ejemplo, al haber creado la escena de identificación con un objeto de tipo *ViewController*, es necesario que los archivos controladores sean de la de misma subclase. Por tanto, se elegirá la subclase *UIViewController*. Para el control de las celdas en cambio, habría que elegir una subclase de tipo *UITableViewCell*. Cada clase controladora ha de coincidir con el tipo de vista que vaya a controlar. Automáticamente, se crearán un fichero cabecera (.h) y uno de implementación (.m) con el nombre que se le haya dado.

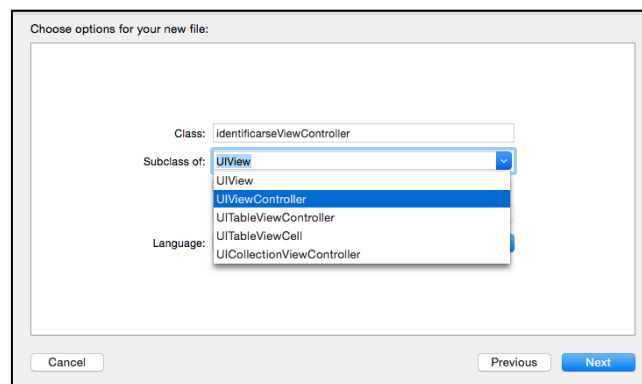


Figura 39. Subclase

El siguiente paso es enlazar la vista con su clase controladora. Hay que estar situado en la vista del storyboard y seleccionar la vista que se pretende asociar. En la parte superior derecha se muestran una serie de propiedades asociadas a la ventana seleccionada. A través de estas propiedades, entre otras cosas, se pueden modificar fondos, fuentes, tamaños... de los objetos de la vista y de la propia vista contenedora. Para realizar esta asociación, es necesario estar situado en la tercera pestaña *identity inspector* y en el campo *Class* introducir el nombre de la clase que se pretende asociar tal y como se muestra en la Figura 40. Es necesario introducir también un nombre en el campo Storyboard ID, sin rellenar este campo el paso de parámetros entre vistas no funciona correctamente. Se recomienda utilizar el mismo nombre para ambos campos.

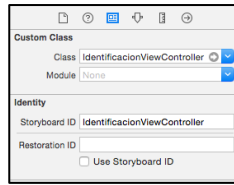


Figura 40. Asociar vista con clase controladora

De la misma forma que es necesario asociar la vista a su controladora, también lo es asociar los objetos incluidos en ella. Por fortuna, XCode proporciona la herramienta *Assistant editor* que agiliza notablemente este proceso. Permite dividir la ventana en dos mitades. En una mitad permanece la vista, en la otra, la cabecera de la clase controladora de esa vista. El proceso consiste en seleccionar los objetos de la vista, y, con el ctrl pulsado arrastrar el ratón hasta el código de la clase. Al hacerlo se muestra una pequeña ventana en la que se elige el tipo de relación que se va a crear para ese objeto, pudiendo ser esta de tipo *Outlet*, *Outlet Collection* o *Action*. Se muestra un ejemplo en la Figura 42.

Los objetos *Outlet* permiten leer o modificar una propiedad desde el código. Por ejemplo, un elemento *Outlet* en *ColeccionApp*, sería el cuadro de texto donde el usuario introduce sus datos para identificarse. *Outlet Collection* permite agrupar objetos del mismo tipo en un array para que posteriormente puedan compartir el mismo comportamiento. En *ColeccionApp* se ha utilizado para definir el comportamiento de los muros en el juego de habilidad. Por último, *action* genera un evento que se dispara a raíz de una interacción del usuario. Todos los botones generados en la aplicación son objetos de este tipo, y están asociados al evento *Touch Up Inside*. Éste evento se dispara cuando el usuario pulsa un botón. El resto de eventos disponibles se pueden ver en la Figura 41.

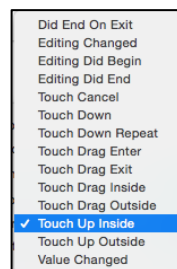


Figura 41. Eventos enviados

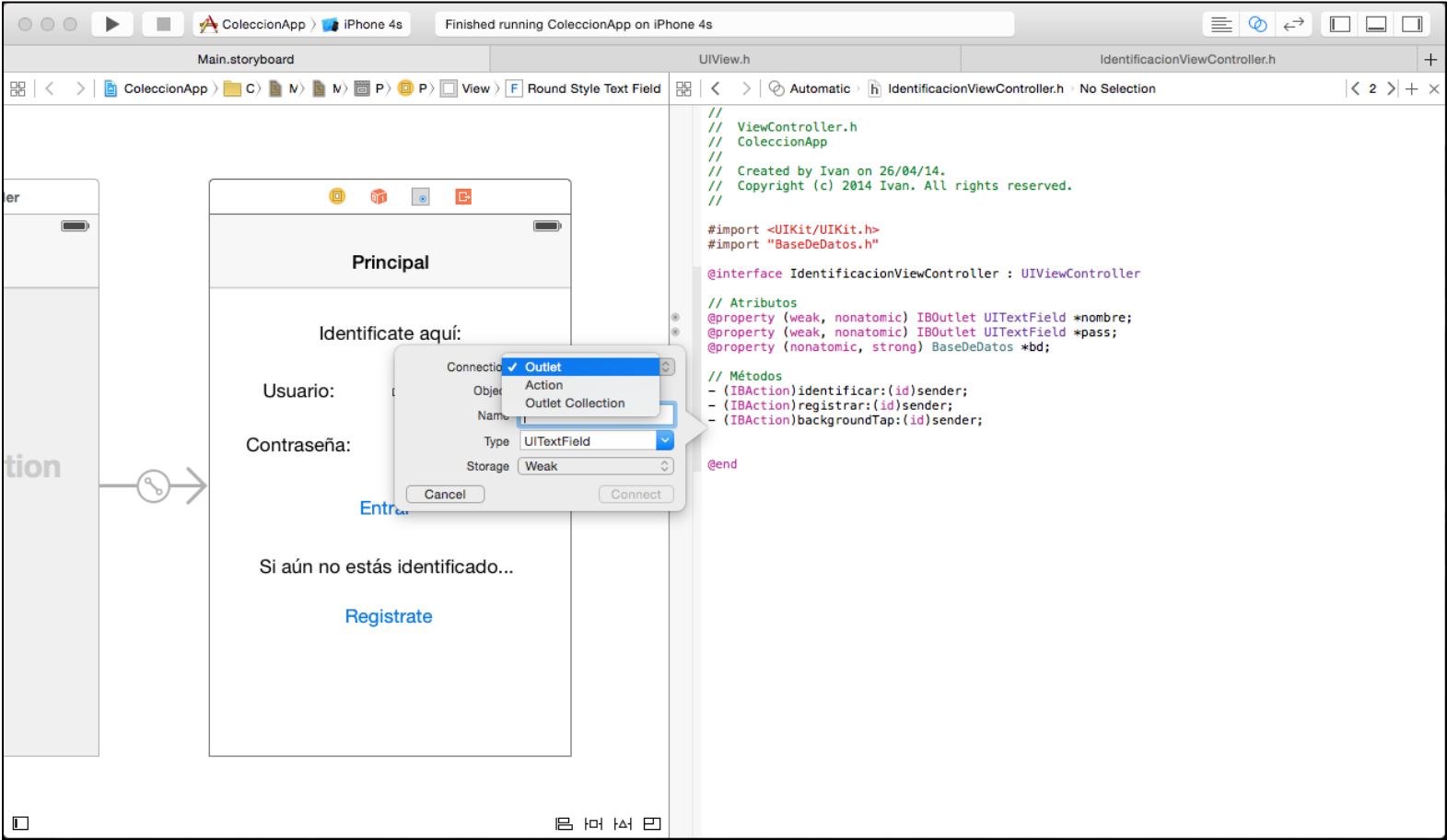


Figura 42. Assistant editor

6.3.5. Base de datos SQLite

Como ya es sabido, la aplicación del usuario tiene incorporada una base de datos de tipo *SQLite*. La base de datos *SQLite* se ha desarrollado a través del framework de Firefox, *SQLite Manager*. Esta herramienta permite exportar la base de datos para que pueda ser importada como un archivo más al proyecto. Sin embargo, no basta con esto para que la base de datos se guarde en la aplicación.

Aprovechando la clase *AppDelegate* se ha creado un método *cargarBaseDeDatos* que se ejecuta cada vez que comienza la aplicación. El fragmento de código que se muestra a continuación accede a la carpeta *library* dentro de la aplicación y comprueba si la base de datos ya existe. Si aún no existe, algo que ocurriría la primera vez que se instala, la crea a partir del archivo *ColeccionApp.sqlite* importado previamente.

```
- (void) cargarBaseDeDatos{
...
NSArray *paths = NSSearchPathForDirectoriesInDomains(NSLibraryDirectory, NSUserDomainMask, YES);
NSString *documentsDirectory = [paths objectAtIndex:0];
NSString *writableDBPath= [documentsDirectory stringByAppendingPathComponent: @"ColeccionApp.sqlite"];

    exito = [filemanager fileExistsAtPath:writableDBPath];
    // Si existe la Base de datos, es que ya está cargada. No hagas nada.
    if (exito){
        NSLog(@"Se ha cargado correctamente la base de datos");
        return;
    }

    NSString *defaultDBPath = [[[NSBundle mainBundle] resourcePath] stringByAppendingPathComponent:
    @"ColeccionApp.sqlite"];

    // Si aún no existe entonces cárgala.
    exito = [filemanager copyItemAtPath:defaultDBPath toPath:writableDBPath error:&error];

...
}
```

La aplicación está pensada de modo que solo pueda haber un usuario en cada dispositivo. Es por eso que no se ha implementado ningún caso de uso que permita desloguearse. La idea es que la primera vez que se abre la aplicación, el usuario tenga la posibilidad de identificarse o bien de registrarse. Sin embargo, una vez que el usuario ya está identificado, se trata de que no tenga que volver a identificarse.

Cuando un usuario se identifica se guardan sus datos en la base de datos local del dispositivo: datos de usuario, sus colecciones, etc. Al terminar de cargar la aplicación, la primera vista que se muestra es la vista controlada por la clase *IdentificacionViewContoller*.

Toda clase controladora tiene un método *viewDidLoad* que se ejecuta cuando se carga la vista. El método *viewDidLoad* de la clase *IdentificacionViewContoller* llama al método *comprobarUsuario* de la clase *BaseDeDatos*. Este método devuelve el usuario de la base de datos local.

En primer lugar, es necesario importar la librería que permite trabajar contra la base de datos SQLite:

```
#import <sqlite3.h>
```

Como cualquier acceso a una base de datos lo primero es abrir la base de datos.

```
sqlite3_open([appDelegate.dataBasePath UTF8String], &bd) == SQLITE_OK
```

Si la conexión es correcta, se guarda la sentencia *SQL* en una variable tipo *NSString* y después la ejecuta. Se recoge el resultado del campo nombre. Todas las clases que tiene el prefijo *NS* son subclases que heredan de la clase raíz *NSObject* lo que les permite comportarse como objetos de *Objective-C*.

```
NSString *sqlSelect = [NSString stringWithFormat:@"SELECT * FROM usuario"];
if (sqlite3_step(sentencia)==SQLITE_ROW){
    ...
    NSString *nombre = [NSString stringWithUTF8String:(char *) sqlite3_column_text(sentencia, 0)];
}

return nombre;
```

Por último se finaliza la sentencia y se cierra la base de datos de la siguiente manera:

```
sqlite3_finalize(sentencia);
sqlite3_close(bd);
```

La clase controladora de la vista *Identificación* mantiene esa vista si no ha devuelto ningún usuario, permitiendo la identificación o el registro. Sin embargo, si devuelve un usuario, llama a método *pushViewController* que transfiere al usuario a la vista que le corresponda dependiendo si ya participa o no en alguna colección.

6.3.6. Php, JSon y Parseo de datos

Obtener información del servidor no es tan ‘sencillo’ como hacerlo de la base de datos local. Para traer los datos del servidor se requiere una serie de pasos que mezclan diferentes tipos de tecnologías. Las utilizadas en *ColeccionApp* son *PHP* y *JSON*.

Para mostrar un ejemplo más completo que el proceso de identificación, y por mantener un orden en el sentido en el que el usuario va viendo las escenas de la aplicación, se va a explicar punto por punto el proceso completo que se lleva a cabo para traer los datos de las colecciones en las que no participa un usuario.

Todas las acciones que se hagan contra la base de datos del servidor (*SELECT*, *INSERT*, *UPDATE*, *DELETE*) se han de hacer mediante ficheros php alojados en el servidor proporcionado por la UPV-EHU. Al igual que en el caso anterior lo primero es establecer la conexión con la base de datos.

Para no tener que repetir código por cada una de las acciones, se han creado un fichero de parámetros y otro de funciones que permiten establecer la conexión desde cualquier otro fichero php mediante una línea de código.

parametros.php

```
<?php
    $host = "localhost";
    $baseDeDatos = "Xirevuelta005_coleccionapp";
    $usuario = "Xirevuelta005";
    $password = "*****";
?>
```

funciones.php

```
<?php
    function conectar() {
        require_once("parametros.php");
        $conexion = mysql_connect($host, $usuario, $password) or die('No se puede conectar a la base de
        datos.');
```

```
mysql_select_db($baseDeDatos, $conexion); return $conexion; }
?>
```

Se ha creado un fichero *obtenerRestoColecciones.php*. Tras establecer la conexión se establece la consulta que permite obtener las colecciones en las que aún no participa el usuario. Estas colecciones serán mostradas en el menú 'Todas las colecciones'. A medida que se va recorriendo en resultado de la consulta fila a fila se van guardando los datos en un array. Una vez se tiene el array se codifica para que los datos sean devueltos en formato *JSON*. Por último se cierra la base de datos.

```
<?php
    require_once("funciones.php");
    $lnk = conectar();
    ...
    $usuario= $_GET["usuario"];
    $colecciones = array();
    $rs = mysql_query("SELECT codColeccion FROM colecciones_usuario WHERE nombre ='$usuario");

    if (mysql_numrows($rs) != 0){
        while($row = mysql_fetch_object( $rs )){
            $colecciones[] = $row;
            header('Content-type: application/json');
```

```
        }
        echo '{"colecciones":'.json_encode($colecciones).'}';
    }
    else {...}
    mysql_close();
?>
```

JSON es un formato para el intercambios de datos, básicamente describe los datos con una sintaxis dedicada que se usa para identificar y gestionar los datos. Una de las mayores ventajas que tiene el uso de *JSON*, es que puede ser leído por cualquier lenguaje de programación. Por lo tanto, puede ser usado para el intercambio de información entre distintas tecnologías. Los datos devueltos en este caso tendrán este aspecto:

```

{"coleccion": [
  {
    "codColeccion": "2",
    "nombre": "Liga BBVA 2013-2014",
    "descripcion": "Liga de fútbol de la temporada 2013 - 2014.",
    "imagen": "http://galan.ehu.es/irevuelta005/DAS/imagenes/coleccion/Lfp14.jpg"
  },
  {
    "codColeccion": "13",
    "nombre": "NBA",
    "descripcion": "Temporada14/15",
    "imagen": "http://galan.ehu.es/irevuelta005/DAS/imagenes/coleccion/NBA_13.jpeg"
  },
  {
    "codColeccion": "14",
    "nombre": "Animales",
    "descripcion": "Colección dedicada a los animales",
    "imagen": "http://galan.ehu.es/irevuelta005/DAS/imagenes/coleccion/animalicos.jpg"
  }
]}

```

Se podría decir que los datos son devueltos como si estuvieran en un array de objetos donde cada objeto incluye una serie de registros. Por suerte, *Objective-C* proporciona la subclase *NSDictionary*. Esta clase se amolda perfectamente a esta estructura de datos. Permite crear una instancia que contiene un conjunto de claves, y por cada clave, un valor asociado de forma inmutable. Tanto las claves como valores deben ser objetos; las claves deben ser diferentes unas de otras, y no deben ser nil; los valores tampoco pueden ser nil. En el caso de la aplicación del usuario en múltiples clases, se ha utilizado en su lugar la subclase heredada *NSMutableDictionary* que, a diferencia de *NSDictionary*, permite modificar los datos contenidos.

Se ha implementado un método *empezarAParsear*, que recoge los datos ejecutados por un archivo.php, los parsea y los devuelve a una instancia de tipo *NSDictionary*. La clase *AppDelegate* es accesible desde cualquier clase, al tratarse de un método que será necesario ejecutar repetidas veces desde distintos controladores, se ha decidido implementarlo en ésta clase. Pero antes de parsear los datos, es necesario recogerlos de la siguiente manera:

```

NSMutableDictionary *mtURL = [[NSMutableDictionary alloc]
initWithFormat:@"http://galan.ehu.es/irevuelta005/DAS/obtenerRestoColecciones.php?usuario=%@", [bd comprobarUsuario]];

[mtURL setString:[mtURL stringByAddingPercentEscapesUsingEncoding:NSUTF8StringEncoding]];
NSData *urlData = [NSData dataWithContentsOfURL:[NSURL URLWithString:mtURL]];

...

```

En *Objective-C* la llamada a los métodos se hace incluyéndolos entre corchetes y con un espacio entre la clase y su método. En el código mostrado [*bd comprobarUsuario*], donde *bd* es una instancia de la clase *BaseDeDatos* y *comprobarUsuario* uno de sus métodos.

La llamada al método *empezarAParsear* incluido en la clase *AppDelegate* se realiza de una manera algo particular. Generalmente la llamada a la clase *AppDelegate* es así:

```

AppDelegate *appDelegate = (AppDelegate *) [[UIApplication sharedApplication] delegate];

```

El problema de hacer una llamada de esta forma es que no se genera el autocompletado de código referente al App Delegate. Es decir, que si se escribe [appDelegate ...] no mostrará la lista de métodos ni propiedades incluidos en esta clase.

Para evitar esto, existe una manera de obtener un acceso global a esa clase sin necesidad de hacer una llamada tan larga. La solución es hacer uso de una simple macro de compilador. En el archivo cabecera *AppDelegate.h* se escribe la siguiente línea.

```
#define ApplicationDelegate ((AppDelegate *)[UIApplication sharedApplication].delegate)
```

A partir de este momento, escribir *ApplicationDelegate* será lo mismo que escribir *((AppDelegate *)[UIApplication sharedApplication].delegate)*. Además, se consigue que XCode muestre el autocompletado. La llamada, por tanto, al método *empezarAParsear* será tal y como se muestra a continuación.

```
dicColecciones = [ApplicationDelegate empezarAParsear:urlData];
```

Dentro del método se incluye un método *NSJSONSerialization* que es el encargado de hacer la transformación de los datos. Los datos son insertados en un *NSDictionary*.

```
jsonDic = [NSJSONSerialization JSONObjectWithData:urlData
          options:kNilOptions
          error:&error];
```

6.3.6. UITableView

En el punto anterior se han recogido los datos de modo que ya se pueda trabajar con ellos. Siguiendo con este mismo ejemplo, el siguiente paso es mostrar en una tabla las colecciones en las que el usuario no participa.

Las clases que intervienen a la hora de representar una tabla son:

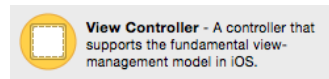
- UITableView : La tabla en sí
- UITableViewCell : Cada una de las celdas de una tabla

Se podría crear un controlador que sea subclase de *UITableViewController*, en ese caso toda la vista sería una tabla. En este caso interesa poder mostrar un botón fuera de la lista. Por lo que no conviene esta solución.



Figura 43. Table View Controller

En su lugar se ha escogido un objeto de tipo *UIViewController* y después se le ha añadido un objeto tipo *tableView* primero y posteriormente un *TableViewCell*.



View Controller - A controller that supports the fundamental view-management model in iOS.

Figura 44. View Controller



Table View - Displays data in a list of plain, sectioned, or grouped rows.



Table View Cell - Defines the attributes and behavior of cells (rows) in a table view.

Figura 45. Table View y Table View Cell

De esta forma se obtiene una jerarquía de vistas, que pueden ser controladas por la misma clase. En este caso la clase controladora se ha llamado *MainViewController*.

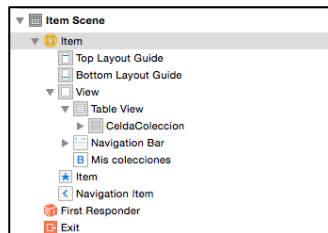


Figura 46. Esqueleto de la vista

En el apartado 6.3.4. Estructura y creación de las vistas en ColeccionApp, se ha explicado la forma de relacionar los objetos de una vista con la clase controladora. Xcode proporciona muchos métodos propios de un controlador de tabla. Estos métodos en realidad son de los protocolos *UITableViewDelegate* y *UITableViewDataSource*. El protocolo delegate determina cómo se tiene que comportar la tabla, por ejemplo el tamaño de las celdas, el método que se ejecuta cuando se selecciona una celda, etc. Por otro lado, el datasource tiene información del contenido que tiene que mostrar la tabla, por ejemplo el número de celdas, el número de secciones, el contenido de las celdas, etc.

UITableViewDataSource

Tal y como se intuye en sus nombres, los métodos *numberOfSectionsInTableView* y *numberOfRowsInSection* determinan el número de filas y de secciones respectivamente que tendrá la tabla.

```
- (NSInteger) numberOfSectionsInTableView:(UITableView *)tableView
```

```
- (NSInteger) tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section
```

El método *cellForRowAtIndexPath* configura el contenido de la celda. Antes de definir este método se tiene que definir el tipo de celda que se va a utilizar. Como ya se menciona en el diagrama de clases del usuario, se han creado y diseñado diferentes clases de celda. En este caso la celda utilizada está formada por una imagen y una etiqueta. La celda se diseña en la propia vista donde se encuentra la tabla. Después se genera una subclase de tipo *UITableViewCell* y se le da un nombre por el que la celda será identificada desde el campo *identificador*, en la ventana de propiedades del storyboard, tal y como se muestra en la Figura 47. A esta celda se la ha llamado *CeldaColeccion*.

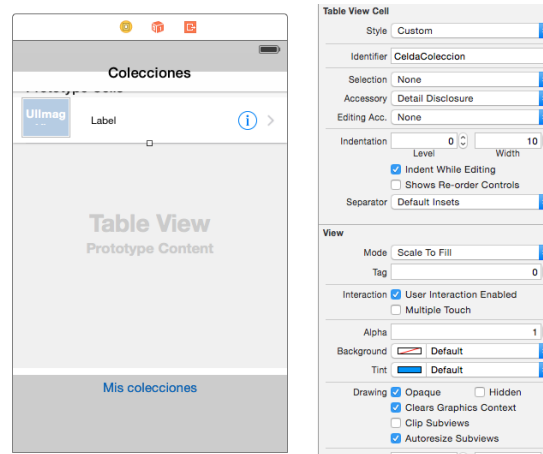


Figura 47. Identificador de la celda

Una vez definida la celda basta con instanciarla y asociar los datos contenidos en el *NSDictionary* a la vista.

```
- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)indexPath {

    // Instanciar la celda
    CeldaColeccion *celda = [tableView dequeueReusableCellWithIdentifier:@"CeldaColeccion"];
    NSDictionary *coleccion = [colecciones objectAtIndex:indexPath.row];

    // Configura la celda
    NSString *imageUrlString = [coleccion objectForKey:@"imagen"];
    NSURL *imageUrl = [[NSURL alloc] initWithString:imageUrlString];
    NSData *datos = [NSData dataWithContentsOfURL:imageUrl];

    celda.nombreColeccion.text = [coleccion objectForKey:@"nombre"];
    celda.imagenColeccion.image = [[UIImage alloc] initWithData:datos];

    return celda;
}
```

La mayoría de las celdas tienen incluidos botones. Por ejemplo, el botón que permite aceptar un intercambio. Para hacer la llamada a ese método se utiliza un selector. Un selector identifica el método que se va a ejecutar sobre el objeto. Cuando el usuario pulse en 'aceptar' se ejecuta la siguiente línea de código.

```
[celda.aceptar addTarget:self action:@selector(intercambiar:) forControlEvents:UIControlEventTouchUpInside];
```

El selector no permite el paso de parámetros, suponiendo que el usuario tenga cinco propuestas pendientes de aceptar, cuando acepte una de ellas, hace falta hacer algo para que el método intercambiar sepa que propuesta está siendo aceptada. Xcode proporciona un parámetro *Tag* que permite otorgar un valor numérico diferente a objetos del mismo tipo. Se le asigna la fila de la propuesta seleccionada por el usuario que viene marcada por la propiedad *indexPath.row*.

```
celda.aceptar.tag = indexPath.row;
```

De este modo, el método intercambiar puede recoger la propuesta que se corresponda con la posición del array correspondiente.

```
NSDictionary *propuesta = [misPropuestas objectAtIndex:sender.tag];
```

UITableViewDelegate:

En este caso únicamente se ha utilizado el método *didSelectRowAtIndexPath* que determina lo que se ejecuta cuando el usuario selecciona una celda.

```
-(void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath *)indexPath
```

6.3.7. Animaciones y paso de parámetros entre controladores

Cuando el usuario seleccione una celda, se tiene que mostrar una nueva ventana con los datos de la colección y un botón que le permita empezar dicha colección. La controladora de esa nueva vista necesita recibir los parámetros de la colección que ha seleccionado el usuario. Gracias a la variable *indexPath.row* se sabe cuál es la fila que ha presionado el usuario, ese parámetro coincide con la posición de la colección en el array de colecciones por lo que se puede insertar el objeto correspondiente a esa posición en un *NSDictionary*.

```
NSDictionary *coleccion = [colecciones objectAtIndex:indexPath.row];
```

Después se instancia el controlador de la vista a la que se quiere transferir al usuario, y se le pasan los parámetros requeridos de la siguiente manera:

```
...
ColeccionDetalleViewController *detalle = [self.storyboard
instantiateViewControllerWithIdentifier:@"ColeccionDetalleViewController"];
[detalle setCodColeccion:[coleccion objectForKey:@"codColeccion"];
[detalle setNombreColeccionString:[coleccion objectForKey:@"nombre"];
...

```

La controladora destino ya se encargará después de asociar esos datos a la vista cuando se cargue la imagen mediante el método *viewDidLoad*. Por último mediante el método *pushViewController* se realiza la transferencia entre vistas.

```
[self.navigationController pushViewController:detalle animated:YES];
```

Cuando el usuario empieza una colección, se le da la bienvenida obsequiándole con un cromó. Para hacer la aplicación algo más vistosa, se ha realizado una pequeña animación en la que sale un sobre de una caja de sobres. Gracias al framework *Core Animation* proporcionado por Apple es posible hacerlo de una manera muy sencilla.

Se guarda en un array una serie de imágenes secuenciales previamente diseñadas, en este caso las imágenes han sido diseñadas personalmente con el programa *Adobe Illustrator*.

```
animacionCajaCromos = @[
    [UIImage imageNamed:@"caja de sobres-0.png"],
    // [UIImage imageNamed:@"caja de sobres-1.png"],
    [UIImage imageNamed:@"caja de sobres-2.png"],
    // [UIImage imageNamed:@"caja de sobres-3.png"],
    [UIImage imageNamed:@"caja de sobres-4.png"],
    ...]

```

Se define una instancia *UIImage* que realizará la animación. Se asigna el array mediante la propiedad de la imagen *animationImages*

```
self.cajaDeCromos.animationImages = animacionCajaCromos;
```

Para comenzar la animación se utiliza el método *animateWithDuration*, que haciendo uso de bloques, permite definir los siguientes tres parámetros para completar la animación:

- Duración de la animación (en segundos)
- Definición de las propiedades que tendrá el objeto al finalizar la animación.
- Código que se desea ejecutar cuando dicha animación finalice

En este caso se trata de una animación compuesta de dos pasos secuenciales. La primera secuencia consiste en un sobre saliendo de la caja. En la segunda secuencia, la caja desaparece y el tamaño del sobre va aumentando progresivamente.

A primera vista, se podría pensar que para hacer esto basta con poner dos llamadas consecutivas al método de clase de *UIView animateWithDuration*. Sin embargo, no es así, ya que al tratarse de un método asíncrono, la ejecución de la segunda llamada, se realizaría inmediatamente a continuación de la primera, no esperando a que finalice dicha animación. Dado que este no es el comportamiento que se espera, debe existir alguna forma de lograrlo. Y efectivamente, así es. Se hace utilizando el parámetro *completion*, que básicamente es un nuevo bloque, que se ejecutará una vez que finalice la animación incluida en el primer bloque.

Para que se entienda mejor, por ejemplo, en la animación de la caja, primero se le asigna un parámetro alpha con valor 0 al sobre. El parámetro alpha mide la transparencia de un objeto utilizando valores comprendidos entre 0 y 1. El valor 0 indica transparencia total, el 1 opacidad. Se le asigna una duración al método y dentro del bloque se establece el valor alpha a 1. Cuando se ejecute la animación, la imagen del sobre pasará del estado inicial donde el sobre esta totalmente transparente al estado final, incluido en el bloque, donde se le dice que la imagen ha de estar opaca, y lo hará de forma progresiva en el tiempo asignado.

```
// Primera secuencia
[UIView animateWithDuration:1 animations:^(
    [self.cajaDeCromos setAnimationRepeatCount:1]; // 0 = loops forever
    [self.cajaDeCromos startAnimating];
    [self.cajaDeCromos setAlpha:0.95];
} completion:^(BOOL finished) {
    [self.cajaDeCromos setAlpha:1];
    [self comenzarAnimacion];
}];

// Segunda secuencia
- (IBAction)comenzarAnimacion {
    [self.imagenSoloSobre setAlpha:0];
    ...
    [UIView animateWithDuration:1.5 animations:^(
        ...
        [self.imagenSoloSobre setAlpha:1];
        ...
    )];
}
}
```

6.3.8. UITabBarController

Cuando el usuario abre el sobre, se le muestra una nueva vista con una barra de herramientas en la parte inferior que está compuesta por pestañas. Se trata de una vista *UITabBarController*.

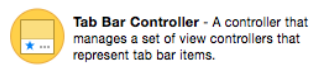


Figura 48. Tab Bar Controller

Este tipo de vistas se encargan de gestionar un conjunto de vistas de tipo *ViewController*, que son representadas en cada una de las pestañas de la barra de herramientas tal y como se muestra en la Figura 49:

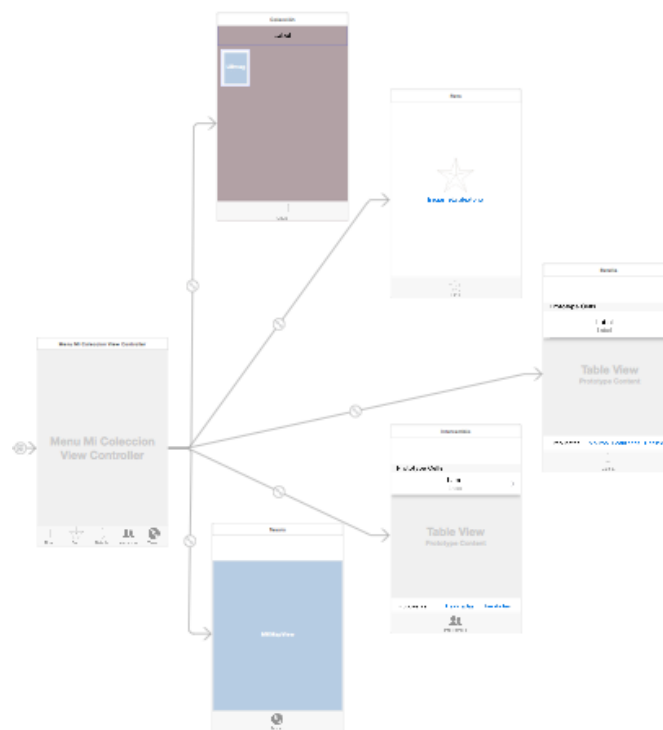


Figura 49. UITabController

Debido a que se trata de una clase administradora de otras, apenas tiene código implementado. Cada una de las vistas incluidas en las pestañas tiene su propia clase controladora que se encarga de su propia gestión.

Hasta ahora se ha visto como el paso de los parámetros se hace a la clase controladora de la vista a la que es transferido el usuario. En esta ocasión sucede un caso algo particular. La transición desde la ventana donde el usuario abre el sobre se tiene que hacer a la subclase *UITabBarController*. Si se hiciera a cualquiera de las otras subclases *ViewController*, no se permitiría la navegación por pestañas. Como se acaba de puntualizar esta clase

controladora no tiene ningún poder de gestión sobre las otras vistas, aparte de la propia navegación, con lo cual no necesita recibir parámetros.

Por otra lado, cada una de las vistas sí que necesitan recibir parámetros para saber en qué colección se encuentran y poder así, mostrar los cromos de esa colección, hacer los intercambios, batallas, etc.

Es decir, hay que hacer una transición a la vista *MenuMiColeccionViewCrontrroller*, subclase de *UITabBarController*, pero se necesita que los parámetros sean transferidos a las otras vistas subclase *UIViewController* (*AlbumViewController*, *RetoViewController*, etc.). Lo primero que se pensó para solucionar este problema fue realizar el paso de parámetros a la clase *MenuMiColeccion*, y que el método *viewDidLoad* de ésta clase volviera a hacer ese paso de parámetros a cada una de sus clases hijas. Lamentablemente esto no funciona. Tenía que existir alguna manera de realizar el paso de parámetros desde una vista a las clases gestionadas por una clase *UITabView*. Efectivamente, existe una solución que lo permite. Para solucionarlo, se utiliza la propiedad *childViewControllers[i]* de la clase *MenuMiColeccionViewController*. Esta propiedad permite instanciar sus clases hijas y realizar tanto la transición como el paso de parámetros en unas pocas líneas de código.

```
// Transición a otra vista
MenuMiColeccionViewController *menuColeccion=[segue destinationViewController];

// Instanciar vistas gestionadas por el TabViewController
AlbumViewController *album = menuColeccion.childViewControllers[0];
RetoViewController *reto = menuColeccion.childViewControllers[1];
BatallaCandidatosViewController *batalla = menuColeccion.childViewControllers[2];
IntercambioCandidatosViewController *intercambioCandidatos = menuColeccion.childViewControllers[3];
TesoroViewController *tesoro = menuColeccion.childViewControllers[4];

// Paso de una propiedad origen a la propiedad
album.codColeccion = codColeccion; destino
reto.codColeccion = codColeccion;
...
```

6.3.9. IUCollectionView

La primera ventana que se muestra en la vista *MenuMiColeccion*, es el álbum virtual donde se encuentran los cromos que tiene el usuario agrupados por familia. La clase controladora de esta vista se llama *AlbumViewController* y como el resto, es subclase *ViewController*.

En este caso se le ha incorporado una *CollectionView* y una *CollectionViewCell* desde la librería de objetos. (Ver Figura 50).

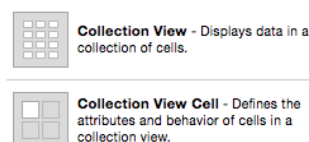


Figura 50. Collection View y Collection View Cell

Los métodos de las clases de una *CollectionView* comparten el mismo comportamiento que los de una *TableView*. De modo que proporcionan muchos métodos propios de un controlador de colección *UICollectionViewDelegate* y *UICollectionViewDataSource*.

En esta vista, cada celda de la colección es un cromó y cada sección una familia. Al igual que con las tablas, como método delegado se utiliza *didSelectItemAtIndexPath* que se encarga de mostrar en otra vista los detalles del cromó seleccionado. Del mismo modo, para definir el número de secciones, el número de celdas y para dar forma a las celdas se utilizan los mismos métodos del *DataSourceViewController* mencionados en el apartado *TableView*. Además de todo esto, interesa también incluir un encabezado por cada sección con el nombre de la familia que corresponda. Para eso, se utiliza el siguiente método adicional que permite incluirlo:

```
- (UICollectionView *)collectionView:(UICollectionView *)collectionView
viewForSupplementaryElementOfKind:(NSString *)kind atIndexPath:(NSIndexPath *)indexPath
```

En este encabezado se pretende mostrar únicamente el nombre de la familia, pero se podría incluir cualquier otra cosa, como botones, imágenes, etc. Para hacerlo se añade justo encima de la celda un objeto de tipo *UICollectionViewReusableView*, al igual que el resto, se arrastra desde la librería de objetos hasta la vista del álbum. (Ver Figura 51). En este caso, solo hace falta añadir una etiqueta en su interior, que es donde aparecerá el nombre de la familia.

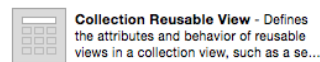


Figura 51. Collection Reusable View

Al igual que ocurre con cada vista, celda, etc., hace falta crear los ficheros de subclase. A esta subclase se le ha llamado *FamiliaReusableView*. Se asocia la etiqueta al archivo de cabecera para crear un *Outlet* que permita ir cambiando de nombre a la etiqueta. El archivo cabecera queda de la siguiente manera:

```
@interface FamiliaReusableView : UICollectionViewReusableView
@property (weak, nonatomic) IBOutlet UILabel *familia;
@end
```

Es muy importante acordarse de incluir el nombre *FamiliaReusableView* en el campo *Identifier* para que funcione. Una vez generada la subclase ya puede ser instanciada y se le puede asignar el valor de la familia que le corresponda.

```

- (UICollectionView *)collectionView:(UICollectionView *)collectionView
viewForSupplementaryElementOfKind:(NSString *)kind atIndexPath:(NSIndexPath *)indexPath{

...
// Instanciar la vista reusable
FamiliaReusableView *headerView = [collectionView
dequeueReusableCellWithIdentifier:@"FamiliaReusableView" forIndexPath:indexPath];

// Obtener el nombre de la familia que corresponda
NSDictionary* datosCromo = [album[indexPath.section] objectAtIndex:indexPath.row];
NSString *title = [[NSString alloc]initWithFormat:@"%@",[datosCromo objectForKey:@"familia"]];

// Asignar el nombre a la familia
headerView.familia.text = title;

...
}

return headerView;

```

La otra vista en la que se utiliza una *CollectionView*, es en el juego de memoria. El juego de memoria consiste en un conjunto de pares de imágenes ocultas que el usuario tiene que ir descubriendo de dos en dos. Cuando el usuario toca una celda, ésta se muestra. Si la siguiente celda que el usuario descubre se corresponde con la anterior, ese subconjunto, queda descubierto y marcado de con un fondo verde. Si no, las celdas vuelven a ocultarse. El usuario tiene que tratar de recordar las imágenes ocultas tras las celdas que va descubriendo, y así poder completarlas con el menor número de movimientos posible. El juego termina cuando el usuario descubre las 16 parejas.

Para implementar el comportamiento de las celdas, se utilizan los mismos métodos delegados y fuente que en el caso del álbum. Pero además, se usan dos métodos más cuya finalidad es encargarse de ajustar el tamaño del conjunto de celdas al tamaño de la pantalla en cada caso, establecer el espacio entre filas y columnas, así como el espacio entre celdas de la misma fila y de la misma columna.

```

- sizeForItemAtIndexPath
- minimumInteritemSpacingForSectionAtIndex
- minimumLineSpacingForSectionAtIndex

```

El juego de memoria utiliza una clase adicional *NSObject*. En ella, se utilizan varios arrays donde se guardan las imágenes incluidas en las celdas. Por ejemplo, hay un *arrayDeElementos* donde se guardan las 32 imágenes del juego, un *arrayDeSeleccionados* para guardar las imágenes de las celdas que han sido seleccionadas y un *arrayDeDescubiertos* que indica cuales son las celdas descubiertas. Entre los métodos de ésta clase destacan los siguientes:

- *nuevoJuego*, se encarga de poblar el array de elementos con las imágenes, y de inicializar los otros dos arrays (seleccionados y descubiertos).
- *mezclar*, hace uso una función *arc4random()*, para que cada vez que se inicie un nuevo juego se muestren las imágenes en el array de elementos aleatoriamente.
- *cellAtIndexPathCanBeSelected*, comprueba si una celda puede ser seleccionada. Para

ello, comprueba el array de elementos descubiertos.

- `dosCeldasSeleccionadas`, comprueba si hay dos celdas seleccionadas mediante el array de elementos seleccionados.
- `checkSelectionWithCompletion`, comprueba si las dos celdas descubiertas coinciden. Disminuye los movimientos permitidos y actualiza la puntuación.

Por otro lado, la clase controladora del juego se encarga de la gestión de la visualización y ocultamiento de las celdas en la vista. Va comprobando los diferentes estados de cada uno de los arrays de la clase *Memorizar* para saber en qué situación se encuentra el juego en el momento en el que el usuario pulsa una celda. El juego no se comporta igual si ya hay una celda seleccionada o si no hay ninguna. Para mostrar u ocultar una imagen se utiliza el parámetro `alpha` otorgándole un 1 para mostrar y un 0 para ocultar.

```
cell.ImagenOcultada.alpha = 0.0;
cell.imagenTapa.alpha = 1.0;
```

Cuando hay una coincidencia, es decir, cuando en el array de descubiertos los elementos son iguales, las celdas no se ocultan y se les cambia el color del fondo a verde.

```
cell.backgroundColor = [UIColor greenColor];
```

Cuando el número de parejas descubiertas sea igual a 16 se lanza una alerta para notificar que el juego ha terminado.

6.3.10. UIAlertView

Una de las clases más utilizadas en *ColeccionApp* es la *UIAlertView*, clase que define las alertas que se muestran en pantalla. Mediante esta clase se puede configurar el título de la alerta, el mensaje mostrado, o los botones que tendrá una alerta.

```
UIAlertView *alerta = [[UIAlertView alloc] initWithTitle:@"Reto superado!!" message:@"Felicidades, obtén tu recompensa"
delegate:self cancelButtonTitle:nil otherButtonTitles:@"Sobre", nil];
```

El método *show* se encarga de mostrar la alerta.

```
[alerta show];
```

Cuando se quiere implementar alguna acción tras presionar un botón de una alerta, ha de hacerse dentro del método *clickedButtonAtIndex*. Para provocar la llamada a este método hay que introducir en el parámetro *delegate* el valor *self*. En caso contrario, al presionar el botón la alerta simplemente desaparecerá sin provocar ninguna acción.

El problema surge cuando, en una misma clase controladora, hay definidas varias alertas. Además, es algo habitual que una alerta tenga más de un botón. Es necesario algún tipo de mecanismo que permita realizar acciones diferentes según que alerta, o que botón, haya sido seleccionada.

En *ColeccionApp* esto es algo que sucede en varias vistas. Por ejemplo, en la clase controladora del juego de memoria hay definidas una cuantas alertas cuando esta clase

está actuando en modo batalla. Para los casos en los que el usuario gana la batalla, la pierde o ésta queda sin determinar, se muestran alertas diferentes, cada una de ellas con su propia acción. En este punto se vuelve a hacer uso del parámetro *tag* descrito previamente. Lo que se hace, por tanto, tras definir una alerta, es utilizar el método *setTag* y asignarle un valor numérico diferente a cada una de ellas.

```
[alerta setTag:1];
```

Por otro lado para distinguir entre botones dentro de una misma alerta se puede utilizar el parámetro *buttonIndex*. Después, con una sencilla programación, se puede controlar qué alerta ha sido pulsada en cada caso.

```
- (void)alertView:(UIAlertView *)alertView clickedButtonAtIndex:(NSInteger) buttonIndex{
    if (alertView.tag == 1) {
        ...
    }
    else if (alertView.tag == 2 && buttonIndex == 0){
        ...
    }
}
```

6.3.11. CMAcceleration, CMMotionManager

En esta primera versión de la aplicación, como ya se sabe, han sido desarrollados dos juegos, el juego de memoria y el de habilidad. En este apartado se va a explicar el proceso de desarrollo de éste segundo juego.

Se pretendía hacer un juego que aprovechara las funcionalidades que un dispositivo móvil proporciona. Al principio se pensó en hacer una especie de laberinto delimitado por muros donde el usuario tendría que ir simplemente recorriéndolo en un tiempo determinado hasta encontrar la salida. El problema de este planteamiento es que una vez que el usuario se supiera el laberinto iba a ser capaz de resolverlo con facilidad. Otra posibilidad pasaba por hacer una serie de pantallas distintas y mostrar laberintos diferentes en cada caso, pero suponía una fuerte carga de trabajo adicional. También se barajó la idea de hacer un laberinto dinámico que distribuyera aleatoriamente los muros a lo largo de la pantalla. Tras valorarlo detenidamente, se desechó esta idea ya que suponía controlar situaciones como que la salida quede inaccesible.

Tras darle varias vueltas, se decidió finalmente optar por implementar un juego de la misma estructura, es decir, con muros, un usuario y una salida, pero que se basara en la habilidad del propio usuario. Gracias las clases *CMAcceleration* y *CMMotionManager* se han podido generar un juego en el que el usuario controla una gota mediante el movimiento del dispositivo. El usuario tiene que ir esquivando las llamas que se mueven a lo largo de la pantalla hasta llegar a la salida.

En primer lugar, como con cada vista, se diseña la interfaz. En este caso, se cree conveniente que la vista se muestre en horizontal. Se van generando los *Outlets* en la clase controladora *LabHabilidadViewController*, con cada una de las imágenes; gota, llamas, muros y salida como se ha explicado en apartados anteriores.

Una vez asociadas las imágenes, ya se puede empezar a trabajar con ellas. Se empieza animando las llamas. La clase *CGPoint* representa un punto bidimensional en las coordenadas del sistema y permite situar una imagen en distintos puntos de la pantalla, se utiliza para fijar el punto hasta donde se moverá la llama. Por ejemplo, en el caso de la primera llama, desde el storyboard, se sitúa su punto de origen pegado al borde inferior de la pantalla. El punto objetivo (*target1*) se ha definido a 124 puntos de distancia, donde se encuentra uno de los muros. Al estar en horizontal y querer que se mueva únicamente de arriba abajo, solo se ha tocado el eje de coordenadas Y.

```
// Animar las llamas
CGPoint origin1 = self.llama1.center;
CGPoint target1 = CGPointMake(self.llama1.center.x, self.llama1.center.y-124);
```

El siguiente paso es generar el movimiento de la llama a través de la clase *CABasicAnimation*. Esta clase permite definir el punto origen, el punto destino, la duración del movimiento, las repeticiones y el sentido. Se ha establecido que una llama haga el recorrido en dos segundos de forma ininterrumpida y en ambas direcciones.

```
// Se le incide que la animación únicamente afecta al eje Y
CABasicAnimation *bounce1 = [CABasicAnimation animationWithKeyPath:@"position.y"];

// Se establecen el resto de propiedades de la animación
bounce1.duration = 2;
bounce1.fromValue = [NSNumber numberWithInt:origin1.y];
bounce1.toValue = [NSNumber numberWithInt:target1.y];
bounce1.repeatCount = HUGE_VALF;
bounce1.autoreverses = YES;
[self.llama1.layer addAnimation:bounce1 forKey:@"position"];
```

Para animar el resto de las llamas se utiliza el mismo mecanismo. Para el movimiento de la gota, en cambio, el proceso es algo diferente ya que dependerá de la orientación del dispositivo en cada momento. Por esa razón aparte de la posición, el tiempo también, pasa a ser una variable a tener en cuenta. Se ha definido una instancia para controlar la hora de la última actualización de la gota y otras dos para identificar su posición actual y anterior. También hacen falta definir las variables X e Y que representen la velocidad de la gota en cada momento.

```
@property (strong, nonatomic) NSDate *lastUpdateTime;
@property (assign, nonatomic) CGPoint posicionActual;
@property (assign, nonatomic) CGPoint posicionAnterior;
@property (assign, nonatomic) CGFloat gotaVelocidadX;
@property (assign, nonatomic) CGFloat gotaVelocidadY;
```

Cuando se carga el juego, la variable de tiempo se inicializa con la hora actual, la posición actual de la gota se establece en las coordenadas (0, 144) que corresponde a la parte izquierda de la pantalla, a la mitad en el eje Y para que empiece a media altura.

```
self.lastUpdateTime = [[NSDate alloc] init];
self.posicionActual = CGPointMake(0, 144);
```

Después se inicializa el gestor que permite la comunicación con el acelerómetro y el objeto que permite poner en cola y lanzar las peticiones enviadas por el acelerómetro y establecer la frecuencia de actualización del mismo.

```
#define kUpdateInterval (1.0f / 60.0f)

self.motionManager = [[CMMotionManager alloc] init];
self.queue = [[NSOperationQueue alloc] init];
self.motionManager.accelerometerUpdateInterval = kUpdateInterval;
```

Se inician las actualizaciones del acelerómetro. Se realiza mediante un bloque, lo cual permite guardar la aceleración actual en la propiedad *aceleración* y llamar al método *update*.

```
[self.motionManager startAccelerometerUpdatesToQueue:self.queue withHandler:
^(CMAccelerometerData *accelerometerData, NSError *error) {
[id self setAceleracion:accelerometerData.acceleration];
[self performSelectorOnMainThread:@selector(update) withObject:nil waitUntilDone:NO];
}];
```

El método *update*, se encarga de calcular la nueva posición de la gota, y a su vez, de llamar al método *moverGota* que actualiza esa posición. A continuación se explican cada uno de los métodos mediante los comentarios incluidos.

```
-(void)update {

// Se obtiene el tiempo transcurrido desde la última actualización
NSTimeInterval secondsSinceLastDraw = -(self.lastUpdateTime timeIntervalSinceNow);

// Se actualiza la velocidad del eje X e Y de la gota
self.gotaVelocidadY = self.gotaVelocidadY - (self.aceleracion.x * secondsSinceLastDraw);
self.gotaVelocidadX = self.gotaVelocidadX - (self.aceleracion.y * secondsSinceLastDraw);

// En base al tiempo transcurrido, y la velocidad de la gota se actualiza el parámetro posición actual
CGFloat xDelta = secondsSinceLastDraw * self.gotaVelocidadX * 500;
CGFloat yDelta = secondsSinceLastDraw * self.gotaVelocidadY * 500;

self.posicionActual = CGPointMake(self.posicionActual.x + xDelta, self.posicionActual.y + yDelta);
[self moverGota];

// Se actualiza la hora de la última actualización
self.lastUpdateTime = [NSDate date];

}
```

El método *moverGota*, además de actualizar la posición de la gota, también controla las diferentes tipos de colisiones que se pueden producir.

```
-(void)moverGota {

// Gestión de colisiones

[self colisionLimitesPantalla];
[self colisionConLamas];
[self colisionConMuros];
[self colisionConSalida];

...
}
```

Para dar una sensación más dinámica al usuario, también se añaden una líneas de código que se encargan de gestionar la rotación de la gota cuando se mueve el dispositivo. Para

hacerlo se calcula el ángulo de la gota en función de la velocidad en los ejes X e Y que ésta lleve.

A continuación, se genera una nueva animación donde se le indica como destino el nuevo ángulo calculado, tal y como se muestra en el código presentado.

```
- (void)moverGota {
...
    // Mueve la gota a su nueva posición
    CGRect frame = self.gota.frame;
    frame.origin.x = self.posicionActual.x;
    frame.origin.y = self.posicionActual.y;

    self.gota.frame = frame;

    // Cálculo del nuevo ángulo para la rotación de la gota
    CGFloat newAngle ángulo = (self.gotaVelocidadX + self.gotaVelocidadY) * M_PI * 4;
    self.angulo += newAngle * kUpdateInterval;

    // Se le indica que va a ser una animación de rotación.
    CABasicAnimation *rotate;
    rotate = [CABasicAnimation animationWithKeyPath:@"transform.rotation"];

    // Se establecen el resto de propiedades de la animación
    rotate.fromValue = [NSNumber numberWithInt:0];
    rotate.toValue = [NSNumber numberWithInt:self.angulo];
    rotate.duration = kUpdateInterval;
    rotate.repeatCount = 1;
    ...
    [self.gota.layer addAnimation:rotate forKey:@"10"];

    // Guarda la posición anterior
    self.posicionAnterior = self.posicionActual;
}

```

La gestión de las colisiones se realiza de diferente manera en función del tipo de colisión que se produzca. Las colisiones entre la gota y los límites de la pantalla simplemente se hacen controlando que la posición de la gota no exceda de las coordenadas que delimitan la pantalla. Además, se pretende que se genere un rebote cuando la gota colisiona con alguna de las paredes. Para ello, basta con dividir la velocidad de la gota e invertir su signo, de este modo, su movimiento cambia de sentido y su velocidad baja a la mitad, provocando esa sensación de rebote. Las propiedades de la vista *self.view.bounds.size.width* y *self.view.bounds.size.height* se utilizan para obtener esos límites. Por ejemplo estas líneas de código controlan la colisión con la pared de la derecha.

```
if (self.posicionActual.x > self.view.bounds.size.width-self.gota.image.size.width/2.0)
{
    _posicionActual.x = self.view.bounds.size.width - self.gota.image.size.width / 2.0;
    self.gotaVelocidadX = -(self.gotaVelocidadX / 2.0);
}

```

Para el caso de las colisiones con las llamas y con la salida, se utiliza el método *CGRectIntersectsRect* que permite comprobar si se produce una intersección entre dos objetos.

```
if (CGRectIntersectsRect(self.gota.frame, llamaLayer1.frame){
...
}

```

Las colisiones con los muros también utilizan este mismo método. Los muros son bloques cuadrados. La idea es que cuando se produzca una colisión, la gota salga rebotada hacia una u otra dirección dependiendo de la parte del muro en la que haya chocado. Hay que calcular el ángulo que determina esa dirección comprobando los centros de ambos objetos. Tal y como aparece representado en la Figura 52, si el ángulo del componente X es mayor que el ángulo del componente Y, se ha de invertir la velocidad del eje Y para que la gota salga rebotada hacia abajo o hacia arriba. En caso contrario, se invertirá en la velocidad del eje X para que la gota rebote en horizontal.

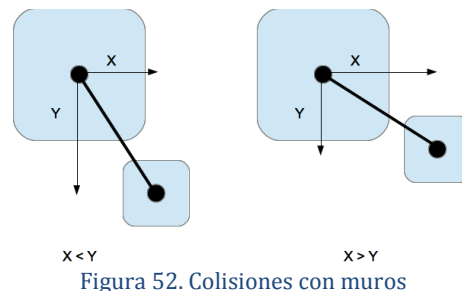


Figura 52. Colisiones con muros

```

if (abs(angleX) > abs(angleY)) {
    _posicionActual.x = self.posicionAnterior.x;
    self.gotaVelocidadX = -(self.gotaVelocidadX / 2.0);
}
else {
    _posicionActual.y = self.posicionAnterior.y;
    self.gotaVelocidadY = -(self.gotaVelocidadY / 2.0);
}

```

6.3.12. SupportedInterfaceOrientations

El juego de habilidad es el único de toda la aplicación cuya orientación se muestra en horizontal, concretamente en modo *LandscapeRight*. Los tipos de orientación que permite un dispositivo se definen en la pestaña general de las propiedades del proyecto (Ver Figura 53).

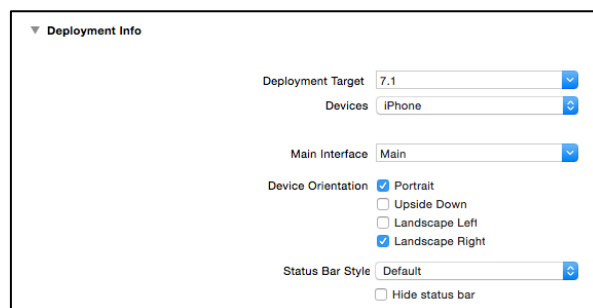


Figura 53. Propiedades del proyecto

En *iOS 6* e *iOS 7*, únicamente se permite utilizar la orientación que este definida en el archivo de la aplicación Info.plist. (Ver Figura 54)



Figura 54. ColeccionApp-Info.plist

Una vista controladora puede sobrescribir el método *supportedInterfaceOrientations* para limitar la orientación de la vista que controla. Generalmente, el sistema llama a este método solo en la controladora de la vista inicial y las vistas 'hijas' heredan este comportamiento de su clase padre sin poder modificarlo.

Para el caso de *ColeccionApp*, se requiere que todas vistas permitan únicamente el modo *Portrait*, salvo el juego de habilidad que hace falta que sólo permita el modo *LandscapeRight*.

Para intentar de modificar el comportamiento de esa vista se trató de usar los métodos *shouldAutorotate* y *supportedInterfaceOrientations* permitiendo la rotación y estableciendo en el juego de habilidad como única orientación soportada *LandscapeRight*.

```

- (BOOL)shouldAutorotate;
{
  return YES;
}

- (NSUInteger)supportedInterfaceOrientations;
{
  return UIInterfaceOrientationMaskLandscapeRight;
}

```

Sin embargo, no bastaba con esto. Al probar la aplicación en el móvil no funcionaba y sólo se veía en vertical. Para solucionarlo ha sido necesario forzar la orientación de la vista del dispositivo de la siguiente manera.

```

NSNumber *value = [NSNumber numberWithInt:UIInterfaceOrientationLandscapeRight];
[[UIDevice currentDevice] setValue:value forKey:@"orientation"];

```

6.3.13. CoreLocation y MapKit

Una de las funcionalidades más empleada en las aplicaciones, es la *geo localización*. No se quería desaprovechar la ocasión de utilizar esta tecnología, así que se decidió crear una clase que la emplease como alternativa para conseguir un cromó. La clase controladora creada para esta vista se ha llamado *TesoroViewController* y se utiliza de una forma muy básica.

Las librerías necesarias para realizar esta implementación son *MapKit* y *CoreLocation*

```
#import <MapKit/MapKit.h>
#import <CoreLocation/CoreLocation.h>
```

Cuando se carga esta vista se obtiene la localización del usuario mediante el método *startUpdatingLocation*. La clase *CLLocationManager* permite elegir la precisión con la que se obtiene la localización. Cuanto más precisa sea, más memoria consume. Se ha establecido la precisión más alta posible a través de la propiedad *kCLLocationAccuracyBest*. Además de la precisión, otro de los métodos utilizados es *setDistanceFilter*. Permite establecer la distancia mínima que ha de ser movido el dispositivo antes de realizar la siguiente actualización. Este parámetro se mide en metros. Se ha establecido una distancia de 10 metros para la ocasión.

```
-(void)obtenerCoordenadasActuales {
    ...
    [locationManager startUpdatingLocation];
    [self setLocationAccuracyBestDistanceFilterNone];
}

-(void)setLocationAccuracyBestDistanceFilterNone {
    [locationManager setDesiredAccuracy:kCLLocationAccuracyBest];
    [locationManager setDistanceFilter:10];
}
```

Lo siguiente es configurar una región e incluirla en el mapa. La región permite ampliar el mapa asignando el tamaño que se desee. La medida ha de hacerse en m². Se ha establecido una región de 2 km² y se ha situado al usuario en el centro de ella.

```
CLLocationCoordinate2D coordenadasUsuario={locationManager.location.coordinate.latitude,
locationManager.location.coordinate.longitude};

MKCoordinateRegion region = MKCoordinateRegionMakeWithDistance(coordenadasUsuario, 2000, 2000);
[self.mapView setRegion:region animated:YES];
```

Por último, se sitúa un tesoro en el mapa. Se ha generado un método *mostrarTesoro*. Lo que hace este método es calcular unas coordenadas relativamente cercanas a la posición del usuario a través de una pequeña función. Después, mediante la clase *MKPointAnnotation* se genera un puntero. Las propiedades de esta clase permiten asignar a este puntero unas coordenadas, un título y un subtítulo entre otras cosas.

Una vez configurado el puntero, se asigna en el mapa representando la posición del tesoro.

```

- (void) mostrarTesoro{

    CGFloat latDelta = rand()*0.035/RAND_MAX -0.02;
    CGFloat longDelta = rand()*0.03/RAND_MAX -0.15;

    CGFloat nuevaLat = locationManager.location.coordinate.latitude + latDelta;
    CGFloat nuevaLon = locationManager.location.coordinate.longitude + longDelta;

    CLLocationCoordinate2D coordenadaTesoro = {nuevaLat, nuevaLon};

    // Se añade el tesoro

    MKPointAnnotation *point = [[MKPointAnnotation alloc] init];
    point.coordinate = coordenadaTesoro;
    point.title = @"Tesoro";
    point.subtitle = @"Ven hasta aquí y obtén tu cromó!";

    [self.mapView addAnnotation:point];
    _tesoro = [[CLLocation alloc] initWithLatitude:nuevaLat longitude:nuevaLon];
}

```

El encargado de realizar las actualizaciones es el método delegado *didUpdateToLocation*. Se ha generado un método llamado *obtenerRecompensaConLocalizacionTesoro* que será el que compruebe si el usuario ha encontrado el tesoro. En él, se llama a la función *distanceFromLocation* de la clase *CLLocationDistance*. Esta función permite obtener la distancia, medida en metros, entre dos puntos. Se ha establecido una distancia prudencial de 150 metros por si la ubicación del cromó resulta inaccesible. Si el usuario se encuentra a menos de esta distancia del tesoro, automáticamente verá una alerta que le permite obtener un cromó de esa colección. Si se encuentra el tesoro, se detendrán las actualizaciones de localización mediante el método *stopUpdatingLocation*.

```

- (void) obtenerRecompensaConLocalizacionTesoro{

    CLLocationDistance distance = [locationManager.location distanceFromLocation:_tesoro];
    ...
    if (distance < 150 && distance > -150) {
        [locationManager stopUpdatingLocation];
        UIAlertView *alert = [[UIAlertView alloc] initWithTitle:@"Felicidades"
        message:@"Has encontrado el tesoro! Obtén tu recompensa"
        delegate:self
        cancelButtonTitle:@"Sobre"
        otherButtonTitles:nil];
        ...
    }
}

```


7 Validación y pruebas

Toda aplicación ha de ir acompañada de un proceso que permita validar su correcto funcionamiento. En este apartado se recogen las pruebas a las que han sido sometidas las aplicaciones de administrador y usuario. Lo ideal es que el resultado sea el esperado, sin embargo en ocasiones se detectan casos especiales que han sido pasados por alto y han de ser corregidos, otras veces puede que sea más conveniente cambiar la especificación del método.

Para mantener el criterio seguido hasta ahora, primero se indicarán las pruebas realizadas a la aplicación del administrador y posteriormente al usuario.

7.1. Administrador

7.1.1. Pruebas de interfaz

Gracias a *Windows buider* proporcionado por eclipse, el diseño de las interfaces es una tarea más sencilla. Permite ir modificando el aspecto de la aplicación fácilmente por lo que, en general, no ha generado demasiados problemas.

En cuanto al comportamiento de las vistas emergentes, se detectó un problema. En la vista *Listado de todas las colecciones* al pulsar 'Agregar colección' se debía mostrar encima otra pequeña ventana emergente donde introducir los datos de la nueva colección. Hasta aquí todo correcto. El problema es que al pinchar en la vista subyacente, el foco cambiaba y volvía a la vista anterior. Esto suponía que, la pequeña ventana donde se crea la colección se mantenía oculta tras la vista principal, pero, además permitía al administrador volver a pulsar el botón que le permite crear una colección.

Se trata de que cuando una de estas ventanas este activa, no se pueda pulsar en la ventana anterior hasta que ésta se haya cerrado. La solución pasaba por convertir la pequeña ventana emergente en una ventana modal. *Windows builder* permite hacerlo desde las propiedades de su interfaz de manera sencilla: `modalityType :APPLICATION_MODAL`

7.1.2. Pruebas funcionales

Comprueban el correcto funcionamiento de cada uno de los casos de usos implementados en la aplicación. Algunos de estos errores ya han sido mencionados en el apartado de desarrollo de la aplicación.

Identificarse

Funcionalidad	Resultado
<i>Identificar al usuario en el servidor</i>	Correcto
<i>Introducir datos erróneos o dejar campos vacíos en la identificación</i>	Correcto
<i>Cargar todas las colecciones en las clases de la aplicación</i>	Correcto
<i>Mostrar las colecciones en la tabla de la interfaz</i>	Error

Al tratar de pasar los datos de las clases a las vistas, en la columna imagen en lugar de cargarse la imagen se mostraba el nombre del objeto y su código en bytes, ver Figura 55. Error al mostrar las colecciones

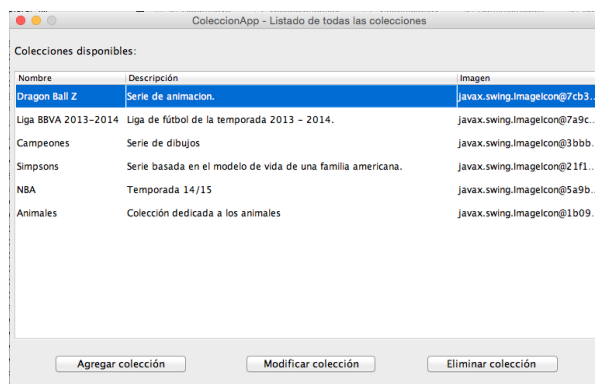


Figura 55. Error al mostrar las colecciones

El problema es que en el modelo de datos que define la tabla, no se había implementado el método `getColumnClass` que se encarga de detectar el tipo de objeto en cada columna. El modo en el que se genera un modelo de tabla se explica en profundidad en el apartado 6.2.3 Tratamiento de tablas.

```
public Class<?> getColumnClass(int columnIndex) {
    Class<?> clazz = Object.class;
    Object aux = getValueAt(0, columnIndex);
    if (aux != null) {
        clazz = aux.getClass();
    }
    return clazz;
}
```

Agregar colección

Funcionalidad	Resultado
<i>Crear una colección</i>	Error
<i>Validación de los campos</i>	Correcto
<i>Explorador de ficheros</i>	Correcto
<i>Cargar Imagen</i>	Correcto

Tabla 40. Agregar colección

No se insertaba bien la ruta de la imagen en la base de datos. Al acceder al explorador de ficheros, tanto la imagen, como su ruta sí se cargaban correctamente en la vista. El proceso que crea una colección, lo primero que hace es subir la imagen al servidor SFTP y luego inserta los datos de la colección en la base de datos. Tal y como se explica en el apartado 6.1. Base de datos, en el campo imagen de la tabla colección va incluida la ruta de la imagen. El problema es que como ruta de la imagen insertada se estaba utilizando solamente el método *directorioActual* de la clase sftp sin especificar el nombre de la imagen. Este método devuelve el directorio donde se acaba de guardar la imagen. En el campo imagen de la base de datos, además del directorio hay que decirle cual es el nombre del fichero.

Para solucionarlo, solo había que recoger además de la ruta y de la imagen, el nombre del fichero:

```
nombreImagen = dlg.getSelectedFile().getName();
```

De esta manera, desde la clase *Album* bastaba con añadir ese nombre al final de la ruta del directorio de la siguiente manera:

```
prep.setString(3, "http://galan.ehu.es/irevuelta005"+sftp.directorioActual()+"/"+pNombreI  
magen);
```

Modificar colección

Funcionalidad	Resultado
<i>Modificar el nombre de una colección si se pone uno que ya existe</i>	Error
<i>Modificar la imagen de una colección</i>	Correcto
<i>Modificar la descripción de una colección</i>	Correcto
<i>Modificar la colección en el servidor</i>	Correcto

Tabla 41. Modificar colección

No estaba controlado el caso que comprueba si el nombre de la colección ya existe. Para solucionarlo se hace el control mediante una condición que lo compruebe antes de crear la colección.

Eliminar colección

Funcionalidad	Resultado
<i>Eliminar los cromos de la colección</i>	Correcto
<i>Eliminar la colección</i>	Correcto

Tabla 42. Eliminar colección

En este caso los resultados fueron los esperados.

Agregar Cromo

Funcionalidad	Resultado
<i>Insertar un número mayor que el número de cromos de la colección</i>	Error
<i>Tratar de insertar un número que ya existe</i>	Correcto
<i>Trata de insertar una familia que no existe</i>	Error
<i>Crear un cromo con una descripción larga</i>	Error
<i>Validación de los campos, campos vacíos o formatos no válidos</i>	Correcto
<i>Explorador de ficheros</i>	Correcto
<i>Cargar Imagen</i>	Correcto
<i>Mostrar el nuevo cromo en la tabla de cromos</i>	Correcto

Tabla 43. Agregar cromo

Este caso es una circunstancia especial ya que el error no se genera en la aplicación del administrador. En cambio, sí que se genera un error en la aplicación del usuario. Cuando un usuario consigue un cromo tras un reto, o cuando inicia una colección, se le ofrece un cromo aleatorio. Existe una función que genera un número aleatorio que va desde 1 hasta el número que esa colección tenga. El número resultante ha de corresponderse con el número del cromo que se le ofrece al usuario. Por ejemplo, si la colección 'Los Simpson' tiene 20 cromos, la función generará un número entre 1 y 20 al azar. Si la función genera el número 5, al usuario se le dará el cromo que se corresponda a ese número.

Partiendo de esta base, si los cromos no están numerados del 1 al 20, puede darse el caso de que ese número de cromo no exista, generando un error en la aplicación del usuario al no poder ofrecerle ningún cromo.

En este caso, parece de sentido común que toda colección vaya numerada en un orden ascendente desde el número 1 hasta el número de cromos que tenga esa colección. Como solución provisional, se ha establecido como precondition del caso de uso *Agregar colección* que el número del cromo tenga que estar comprendido entre el número 1 y el total de cromos de la colección.

Cuando se crea un cromo hay un campo desplegable que permite escoger una familia de un listado a la que el cromo será asociado. Si se creaba el cromo escogiendo una familia que ya se encontraba en el desplegable, no había ningún problema y el cromo se generaba bien. El problema surgía cuando se pretendía escoger una familia que aún no existía en esta colección. Junto al desplegable hay un botón 'Gestionar familias'. Este botón fue implementado aquí, precisamente para que el administrador no tuviera que salir de esta vista y perder los datos cargados solo para tener que generar una nueva familia. La cuestión es que, el administrador accedía bien al menú que le permite generar una familia. Lo que pasaba es que tras generar la familia y volver a la ventana de la creación de la colección, en el desplegable no aparecía la familia creada. En realidad la familia se había creado bien, pero hacía falta volver a llamar al método *cargarFamilias* que las cargase de nuevo en el desplegable. Eso fue lo que se hizo para solucionarlo.

Por último, para solucionar el problema de la descripción del cromo, bastaba con aumentar la longitud asignada al campo *descripcion* de la tabla cromo en la base de datos del servidor. El tipo de datos utilizado es tipo *varchar*.

Modificar cromo

Funcionalidad	Resultado
<i>Tratar de modificar el número por uno que ya existe</i>	Correcto
<i>Modificar la imagen de un cromo</i>	Correcto
<i>Modificar la familia de una colección</i>	Correcto
<i>Modificar la descripcion de un cromo</i>	Correcto
<i>Modificar el nombre de un cromo</i>	Correcto
<i>Mostrar el cromo modificado en la tabla de cromos</i>	Correcto

Tabla 44. Modificar cromo

Al tratarse de una vista muy parecida a la que permite agregar una colección, los problemas que podían surgir ya se sabía cómo solucionarlos.

Eliminar Cromo

Funcionalidad	Resultado
<i>Eliminar el cromo</i>	Correcto
<i>Dejar de ver el cromo en la tabla</i>	Correcto

Tabla 45. Eliminar cromo

El resultado de estas pruebas fueron los esperados.

Traer cromo

Funcionalidad	Resultado
<i>Reutilizar un cromo existente</i>	Error
<i>Mostrar el cromo traído en la tabla de cromos</i>	Correcto

Tabla 46. Traer cromo

Cuando se reutilizaba un cromo de otra colección se pasaban todos los atributos del cromo, incluido su número. En ocasiones sucedía que el número de un cromo que se traía ya estaba siendo utilizado por otro cromo. Por ejemplo, si se está modificando la colección de la liga de fútbol 2014, y se tiene creados dos cromos, uno tiene asignado el número 1 y el otro tiene asignado el número 2. Ahora se quiere aprovechar un cromo de la colección de la liga de fútbol 2013, pero se da el caso que ese cromo tiene asignado el número 1. Para solucionarlo se implementó una vista intermedia que permite asignarle otro número.

Gestionar familias

Funcionalidad	Resultado
<i>Crear una familia</i>	Correcto
<i>Añadir una familia a una colección</i>	Correcto
<i>Listar familias de la colección seleccionada en el comboBox</i>	Correcto
<i>Quitar una familia de una colección</i>	Correcto
<i>Modificar una familia</i>	Correcto
<i>Eliminar una familia</i>	Error

Tabla 47. Gestionar familias

El problema en este caso es que no se estaba contemplando la posibilidad de que se podía estar eliminando una familia que estaba siendo utilizada. Para solucionarlo, se comprueba antes de eliminar una familia si hay alguna colección que la esté utilizando. Mientras la familia forme parte de alguna colección no podrá ser eliminada. Únicamente se podrá quitar la asociación de esa familia con una colección.

7.2. Usuario

7.2.1. Pruebas de interfaz

Se han realizado pruebas de navegación tanto por el simulador iOS, como cargando la aplicación en un iPhone 4. En ambos casos, la navegación por las diferentes vistas es correcta. Al comparar las dos ejecuciones sí que se aprecia una mayor fluidez cuando la aplicación se ejecuta desde el simulador.

7.2.2. Pruebas funcionales

Registrarse

Funcionalidad	Resultado
<i>Registro del usuario</i>	Correcto

Tabla 48. Registrarse

En esta vista se ha tenido que implementar un método que oculte el teclado al tocar la pantalla para poder pulsar sobre el botón que permite hacer el registro que se encuentra justo bajo él. Con añadir la siguiente línea de código es suficiente:

```
[self.view endEditing:YES];
```

Identificarse

Funcionalidad	Resultado
<i>Identificación del usuario</i>	Correcto
<i>Validación de los datos y comprobación de campos vacíos</i>	Correcto
<i>Cargar las colecciones del usuario en la base de datos local</i>	Correcto

Tabla 49. Identificarse

Tanto el proceso de identificación como el de validación de los campos han sido satisfactorios.

Acceder al catálogo completo

Funcionalidad	Resultado
<i>Cargar las colecciones en el TableView</i>	Error
<i>Transición a vista de las colecciones en las que participa el usuario</i>	Error
<i>Mostrar detalles de una colección</i>	Correcto
<i>Empezar una colección</i>	Correcto
<i>Abrir el sobre</i>	Correcto
<i>Agregar un cromó aleatorio</i>	Error*

Tabla 50. Acceder al catálogo completo

No se cargaban correctamente los datos en las celdas de las tablas. Este problema surge por algo que ya se ha mencionado en el apartado del desarrollo. Cuando se instancia una celda reusable se utiliza su identificador.

```
CeldaColeccion *celda = [tableView dequeueReusableCellWithIdentifier:@"CeldaColeccion"];
```

Lo que estaba sucediendo es que se había olvidado rellenar ese campo en las propiedades del storyboard mostradas en la Figura 56, con lo cual el código era incapaz de identificar la celda y no mostraba los datos de cada una de las colecciones.

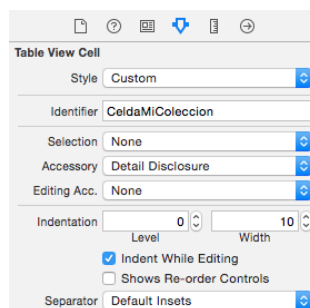


Figura 56. Identificador de la celda

Por otro lado, cuando el usuario pulsaba sobre 'Mis colecciones' la transición a la otra vista se realizaba correctamente. Pero si la tabla actual, no tenía colecciones, el botón dejaba de funcionar. No se estaba tratando el caso especial que determina lo que sucede cuando la tabla del catálogo no tiene colecciones. Se ha modificado para que cuando no haya

colecciones se genere una transición que lleve al usuario a la vista de sus colecciones de manera automática al cargar la vista, esto es, en el método *viewDidLoad*.

El problema que surgía al agregar un cromó aleatorio, es el mencionado en el apartado anterior, cuando el usuario introducía números de cromó no comprendidos entre los límites de la colección. (Ver 7.1.2. Pruebas funcionales)

Finalmente se comprobó que si se iniciaba la aplicación sin conexión a internet, la aplicación se cerraba. Esto se debía a que, cuando no había conexión, el método *empezarAPasarsar* recibía un parámetro nil y generaba un error. En este caso, basta con comprobar si el método ha recibido datos o no. De esta manera, cuando no hay conexión simplemente no se cargan las colecciones del servidor, en lugar de cerrarse la aplicación.

Listar tus colecciones

Funcionalidad	Resultado
<i>Eliminar una colección</i>	Correcto
<i>Mostrar las colecciones del usuario</i>	Correcto

Tabla 51. Listar tus colecciones

En este punto se comprueba si las colecciones del usuario se muestran correctamente y si se el borrado de cada una se realiza correctamente. En ninguno de los dos casos se ha detectado problema alguno.

Acceder a una colección

Funcionalidad	Resultado
<i>Cargar los cromos del usuario</i>	Correcto
<i>Mostrar los cromos agrupados por sección</i>	Error
<i>Acceder a los detalles de un cromó</i>	Correcto

Tabla 52. Acceder a una colección

Al principio se hizo la prueba de visualizar el álbum con unos pocos cromos y tan solo dos familias. Los cromos se visualizaban bien. Más adelante, para tener un álbum algo más poblado, se decidió elegir una colección y empezar a insertar unos 10 cromos adicionales. En este caso se escogió la colección 'Los Simpson'. Una vez insertados los cromos se comprobó que cuando se movía el scroll hacia abajo, se mostraban cromos repetidos en celdas en las que no les correspondía estar. Por ejemplo, si se tenía el cromó de Bart que

ocupa el número 2, al bajar el scroll y volver a subirlo se mostraba el mismo cromó también en la posición 3.

Este problema fue un fallo de programación del método *cellForItemAtIndexPath* encargado de cargar los cromos en las celdas. En el código original lo que se hacía era comprobar si se tenía el cromó con un método propio *comprobarSiTieneCromo* de modo que mostrase solo aquellos que el usuario tuviera. Si no los tenía el método no hacía nada. Al parecer, al bajar el scroll, el método *cellForItemAtIndexPath* vuelve a ser llamado, y por alguna razón que se desconoce, estaba volviendo a cargar esos cromos de nuevo en celdas que no se correspondían con su posición. En cualquier caso, para implementar un código más completo, se probó a añadir una condición *else* que ocultase todos aquellos cromos que el usuario no tenía. El problema se solucionó. A continuación se muestra el resultado del código final.

```
if([self comprobarSiTieneCromo:[datosCromo objectForKey:@"numero"]]){
    NSString *imageUrlString = [datosCromo objectForKey:@"imagen"];
    NSURL *imageUrl = [[NSURL alloc] initWithString:imageUrlString];
    NSData *datos = [NSData dataWithContentsOfURL:imageUrl];
    cromos.imgCromo.hidden = NO;
    cromos.imgCromo.image = [[UIImage alloc] initWithData:datos];
    cromos.numCromo.hidden = YES;
}
else{
    cromos.imgCromo.hidden = YES;
    cromos.numCromo.hidden = NO;
    cromos.numCromo.text = [datosCromo objectForKey:@"numero"];
}
}
```

Reto de memoria

Funcionalidad	Resultado
<i>Cargar reto de memoria</i>	Correcto
<i>Abandonar reto de memoria</i>	Correcto
<i>Reiniciar reto de memoria</i>	Correcto
<i>Superar reto de memoria</i>	Error
<i>No superar reto de memoria</i>	Correcto
<i>Jugabilidad del reto</i>	Error

Tabla 53. Reto de memoria

Cuando se superaba un reto, no se hacía el paso de parámetros que permite que se muestre la imagen de la colección, el nombre, etc. Tampoco se le pasaba el código de la

colección, con lo cual la controladora no sabía en qué colección basarse para generar el número aleatorio. Realizando ese paso de parámetros entre vistas se soluciona.

El fallo en la jugabilidad consistía en lo siguiente. Si se destapaba una celda, y rápidamente se volvía a pulsar sobre otra, se disminuía la cantidad de movimientos permitidos dos veces. Esto se debe a que la duración de la animación asignada al destape era demasiado larga. Permitía pulsar una segunda celda sin que la acción derivada de la selección de la primera hubiera terminado. Para el sistema, es como si hubiera presionado dos celdas en lugar de una. Para solucionar el problema se redujo esta duración.

Reto de habilidad

Funcionalidad	Resultado
<i>Cargar reto de habilidad</i>	Error
<i>Colisión con los límites de la pantalla</i>	Error
<i>Colisión con los muros</i>	Correcto
<i>Colisión con las llamas</i>	Correcto
<i>Superar reto de habilidad – Colisión con la salida</i>	Correcto
<i>Movimiento de la gota</i>	Correcto
<i>Jugabilidad del reto</i>	Correcto

Tabla 54. Reto de habilidad

El juego de habilidad está pensado para que únicamente se pueda jugar con el móvil girado a la derecha, *LandscapeRight* (ni en vertical, ni horizontal a la izquierda). Al cargar el juego la vista aparecía en vertical y era necesario girar el dispositivo para que se mostrara en horizontal. Además, al ser un juego que se basa en el giro del dispositivo, la orientación cambiaba de nuevo a vertical al realizar algún movimiento brusco. La solución para fijar la orientación se puede consultar en el apartado de desarrollo del usuario 6.3.12. `SupportedInterfaceOrientations`.

El problema con los límites de la pantalla ocurrió al insertar una barra de navegación que permite al usuario volver atrás. Al realizar el diseño de los componentes no se tuvo en cuenta. Tampoco cuando se programaron los movimientos de las llamas y el de la gota. Esta configuración provocaba que la gota y las llamas desaparecieran detrás de la barra de navegación. Hubo que reubicar los componentes y modificar el movimiento de las llamas y las colisiones de la gota con la pared superior de la pantalla.

Batalla

Funcionalidad	Resultado
<i>Listar usuarios para batalla</i>	Correcto
<i>Proponer batalla</i>	Correcto
<i>Consultar propuestas de batalla recibidas</i>	Correcto
<i>Consultar propuestas de batalla realizadas</i>	Correcto
<i>Consultar batallas en curso</i>	Correcto
<i>Aceptar / Rechazar propuestas de batalla</i>	Correcto
<i>Empezar batalla</i>	Correcto
<i>Obtener cromos tras victoria</i>	Error
<i>Rendirse</i>	Correcto

Tabla 55. Batalla

En este caso la mayor parte de las funcionalidades ya se habían utilizado. Al fin y al cabo el proceso de listar usuarios es idéntico al que lista colecciones. La vista que le permite al usuario proponer una batalla es similar a la composición del álbum. En este caso, no ha habido demasiados problemas y las pruebas han devuelto el resultado esperado. Para comprobar el correcto funcionamiento se ha utilizado un iPhone 4 para simular un usuario y el simulador iOS para simular su oponente. Se ha probado a realizar solicitudes desde ambos sitios. También se han finalizado batallas perdidas, ganadas y sin determinar y, en distinto orden para probar cada uno de los siguientes casos:

- Terminar una batalla primero y ganar a posteriori
- Terminar una batalla primero y perder a posteriori
- Terminar una batalla el último y ganar
- Terminar una batalla el último y perder.
- Rendirse

En todas estas situaciones el funcionamiento ha sido el esperado.

A pesar de todo, sí que se ha detectado un fallo cuando un usuario gana una batalla y trata de obtener un cromos en ciertas circunstancias. No se ha considerado la posibilidad de que el otro usuario no tenga ningún cromos, o tenga pocos y justo los tenga bloqueados. Si esto sucediera el usuario accedería a la colección del otro usuario y no podría obtener su recompensa. Pero es que, además, por como está implementado el código, la base de datos registraría que ya lo ha obtenido. Para solucionar este problema se ha comprobado si el usuario oponente tiene cromos no bloqueados antes de ofrecerle la posibilidad de quitarle un cromos. Si se da el caso, se le muestra una alerta para que acceda más tarde al listado de batallas en curso y pueda obtenerlo. Naturalmente desde esta lista de batallas en curso, se

realiza la misma comprobación antes de permitir que el usuario obtenga el cromo que le corresponde.

Intercambio

Funcionalidad	Resultado
<i>Listar usuarios intercambio</i>	Correcto
<i>Proponer intercambio</i>	Correcto
<i>Consultar propuestas de intercambio recibidas</i>	Correcto
<i>Consultar propuestas de intercambio realizadas</i>	Correcto
<i>Empezar batalla</i>	Correcto
<i>Aceptar / Rechazar intercambio</i>	Error

Tabla 56. Intercambio

En el proceso de intercambio, más que un fallo de programación, podría considerarse un fallo conceptual. Había varias situaciones que no se estaban teniendo en cuenta.

Podía darse el caso de tener varias propuestas de intercambio con un solo cromo. Por ejemplo, en la colección de 'Los Simpson', si un usuario tenía una sola vez el cromo de Homer, el sistema le permitía ofrecerlo en tantos intercambios como el usuario quisiera. De modo que si dos usuarios aceptaban el intercambio, el sistema provocaría un error. Se pensó en introducir un campo booleano *bloqueado* en la base de datos que permitiera controlarlo. El problema de esta 'solución' es que si el usuario tenía el cromo 4 veces, al ofrecerlo en una propuesta se bloqueaba. No se consideró como un funcionamiento correcto que teniendo el cromo 4 veces solo pudiera ser ofrecido una vez. Se decidió cambiar el atributo *bloqueado* de booleano a numérico. Con este cambio, cada vez que el cromo es ofrecido como intercambio, se incrementa en uno el valor de éste campo. De este modo, el cromo únicamente se bloquea si los campos *nVeces* y *bloqueado* tienen el mismo valor. Cuando el otro usuario acepta o rechaza el intercambio se disminuye el valor de este campo. Por otro lado, si otro usuario le quiere proponer a éste un intercambio, se le muestra el estado de los cromos, impidiéndole 'pedir' los que estén bloqueados.

Sin embargo, seguía habiendo un caso sin tratar. El bloqueo solo se añade cuando un usuario ofrece un intercambio. Es decir, que si un usuario tiene el cromo de Homer una sola vez, puede recibir todas las solicitudes de intercambio que el resto de usuarios quiera y el cromo no se bloqueará. Partiendo de esa premisa, podía darse la situación de que un usuario tuviera propuestas por un cromo y aun así el usuario, posteriormente, generase una propuesta en la que lo ofreciera. Si le aceptan esa propuesta, se queda sin el cromo y entonces, ¿qué ocurre con las solicitudes que ya tenía de antes?. Esas propuestas que tenía deberían desaparecer. Para solucionarlo se programó que si el usuario se quedaba sin un cromo todas las solicitudes que hubiera por ese cromo, se trataran como solicitudes rechazadas, desbloqueando así los cromos de los usuarios que solicitaron el intercambio.

Al igual que en el caso de la batalla se pudo comprobar el correcto funcionamiento del proceso de intercambio utilizando un usuario con el simulador iOS y otro con el iPhone 4.

Tesoro

Funcionalidad	Resultado
<i>Mostrar región en el mapa</i>	Correcto
<i>Obtener ubicación actual y situar al usuario en la región</i>	Error
<i>Mostrar tesoro</i>	Correcto
<i>Comprobar si se obtiene un tesoro al aproximarse a menos de 150 m</i>	Correcto

Tabla 57. Tesoro

El simulador iOS permite simular una localización que se le indique. Es posible situar al usuario en la misma posición del tesoro para ver si muestra la alerta que permite obtener el cromó. Así se hizo y los resultados fueron buenos. Sin embargo, con la actualización del XCode, la localización a través del simulador dejó de funcionar. Tras buscar información sobre el tema se encontró que el funcionamiento de CoreLocation había cambiado con la actualización de XCode a *iOS 8*. Ver <http://nevan.net/2014/09/core-location-manager-changes-in-ios-8/>

Para solucionarlo primero había que incluir una clave al fichero *ColeccionApp-info.plist* tal y como se muestra en la Figura 57. *ColeccionApp-Info.plist* Después en el código, justo antes de ejecutar *startUpdatingLocation*, hay que llamar al método *requestWhenInUseAuthorization*. Con esto se soluciona desde el simulador, el problema es que provoca que la localización en el móvil deje de funcionar. La razón es que el iPhone 4 corre sobre *iOS 7*. Hubo que controlar mediante código una forma de identificar el sistema operativo sobre el que estaba corriendo la aplicación en cada momento. Se hizo mirando si el sistema era capaz de responder al método *requestWhenInUseAuthorization* de la siguiente manera.

```
// Comprueba si es iOS 8. Evita en error en caso de ser iOS 7.(no olvidar incluirlo en pinfo.list)
if ([CLLocationManager respondsToSelector:@selector(requestWhenInUseAuthorization)]) {
    [CLLocationManager requestWhenInUseAuthorization];
}
[CLLocationManager startUpdatingLocation];
...
}
```

Con esta solución se logró que la localización funcionara en ambos sitios sin tener que degradar de nuevo la versión de XCode. Es importante no olvidar incluir la clave en el fichero *ColeccionApp-info.plist*. (Ver Figura 57. *ColeccionApp-Info.plist*)



```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
<key>CFBundleDevelopmentRegion</key>
<string>en</string>
<key>CFBundleDisplayName</key>
<string>${PRODUCT_NAME}</string>
<key>CFBundleExecutable</key>
<string>${EXECUTABLE_NAME}</string>
<key>CFBundleIdentifier</key>
<string>Ivan.${PRODUCT_NAME:rfc1034identifier}</string>
<key>CFBundleInfoDictionaryVersion</key>
<string>6.0</string>
<key>CFBundleName</key>
<string>${PRODUCT_NAME}</string>
<key>CFBundlePackageType</key>
<string>APPL</string>
<key>CFBundleShortVersionString</key>
<string>1.0</string>
<key>CFBundleSignature</key>
<string>????</string>
<key>CFBundleVersion</key>
<string>1.0</string>
<key>LSRequiresIPhoneOS</key>
<true/>
<key>NSLocationWhenInUseUsageDescription</key>
<string>"when"</string>
<key>UIInAppStoragePathFile</key>
<string>Main/</string>
<key>UIRequiredDeviceCapabilities</key>
<array>
<string>armv7</string>
</array>
<key>UISupportedInterfaceOrientations</key>
<array>
<string>UIInterfaceOrientationPortrait</string>
<string>UIInterfaceOrientationLandscapeRight</string>
</array>
</dict>
</plist>

```

Figura 57. ColeccionApp-Info.plist

7.2.3. Pruebas de rendimiento

Para probar aspectos de rendimiento como la memoria o el uso de la CPU, se ha utilizado la herramienta Instruments, proporcionada por el SDK de iPhone, (Ver Figura 58. Instruments).

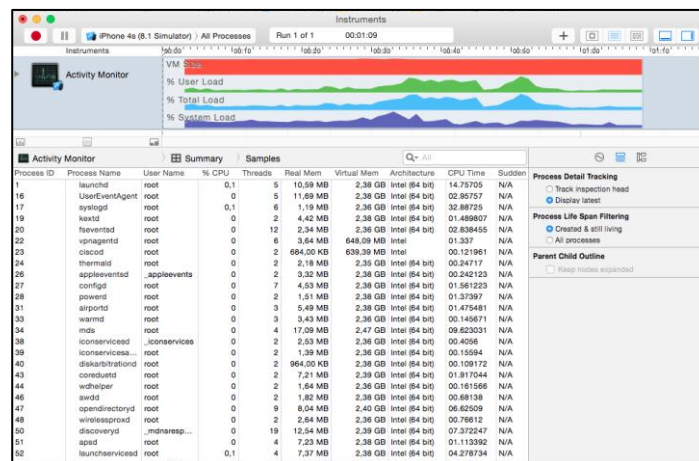


Figura 58. Instruments

Se comprobó que cuando se abría un sobre, se producía un pico en el uso de memoria que en ocasiones provocaba que se cerrara la aplicación. El problema es que se utilizaron demasiadas imágenes para generar la animación que hace que salga el sobre de la caja. Tras reducir el número de imágenes, el problema se solucionó.

8 Conclusiones

Tras finalizar el proyecto, llega el momento de echar la vista atrás. Es hora de hacer un análisis del largo camino recorrido hasta finalmente alcanzar la meta. Pero hasta alcanzar esa meta, el camino ha sufrido cambios de dirección, desvíos inesperados y en algunas ocasiones se ha tenido que dar marcha atrás. Dar un paso atrás para dar dos adelante. Son muchas las conclusiones que se pueden sacar tras la finalización de la aplicación. Sin duda la más importante es la demostración de que se puede lograr desarrollar un proyecto completo en un lenguaje de programación desconocido, y todo ello, partiendo desde cero.

Si se consulta el apartado de Objetivos se puede ver que el objetivo principal era desarrollar un juego para *iOS*. También se menciona que se desea comprobar la capacidad de llevar a cabo un proyecto de principio a fin. En este sentido, se puede afirmar, que se han cumplido. Quizás habría que matizar eso de principio a fin. Se prefiere considerar la aplicación como una primera versión que aún tiene mucho margen de mejora.

Además de los objetivos fijados, se han ido consiguiendo otros que en su momento pasaron completamente desapercibidos. Por ejemplo, gracias a este proyecto se ha mejorado de forma notable la búsqueda de información en internet. Cuando se hace una búsqueda aparece infinidad de 'soluciones' posibles para un mismo problema. Es tal cantidad de información, que se puede correr el riesgo de perderse en toda ella y pasarse horas buscando sin encontrar algo que se adecue a lo que uno necesita pudiendo llegar a la desesperación. Al tratarse de un proyecto en el que se ha utilizado un lenguaje de programación desconocido, la búsqueda de información era algo obligado. Saber buscar, saber elegir, cosas de las que antes de empezar se adolecía bastante, y que ahora se puede afirmar que es un ámbito en lo que se ha mejorado muchísimo.

En cuanto a la parte más entusiasta del proyecto, más orientada al entretenimiento de los usuarios, poco se puede decir. La idea está correctamente plasmada, pero no ha habido ocasión de hacer un estudio del posible éxito que podría tener la aplicación si saliera al mercado. Únicamente se ha podido sondear a personas de círculos cercanos, donde, todo hay que decirlo, *ColeccionApp* ha tenido gran acogida y, salvo por pequeñas observaciones más ligadas a temas de diseño, ha sorprendido por su originalidad y suscitado muy buenas críticas.

Tras analizar la gestión de riesgos, se comprueba que uno de los riesgos era realizar una incorrecta planificación temporal. Tal y como se temía, era un riesgo muy probable que finalmente sucedió. La prevención consistía en tratar de hacer una estimación lo más precisa posible. El problema en este caso fue la falta de experiencia en proyectos tan grandes. El plan de contingencia era replantear la planificación temporal para recuperar el máximo tiempo perdido posible. El proyecto se fue retrasando progresivamente. Una de las razones precisamente, fue otro de los riesgos planteados, indisponibilidad del desarrollador por prácticas o empleo. La prevención para este riesgo era escoger unas prácticas a media jornada para poder tener más tiempo para dedicarle al proyecto. La realidad fue que se empezaron prácticas a 35 horas semanales. El plan de contingencia

consistía en reorganizarse, tratando reducir el tiempo dedicado a actividades de ocio o extraescolares. Se cree que es un planteamiento correcto pero, difícil de cumplir por la exigencia y por la falta de presión. Por fortuna no se ha producido ningún riesgo que no se hubiera planificado salvo en el caso de una actualización de XCode mencionada en el apartado 6.3.13. CoreLocation y Mapkit y que se pudo resolver mediante código.

Aparte de esto, hay que valorar también otras cosas como lo difícil que es elaborar una planificación temporal que se acerque mínimamente a la realidad, sobretodo, cuando no se tiene experiencia en proyectos de esta magnitud. A continuación se presenta una tabla con la duración que se estimó antes de empezar el desarrollo del proyecto y la duración real aproximada de cada uno de los módulos del *EDT*, nada que ver.

TAREA	DURACIÓN ESTIMADA (Horas)	DURACIÓN REAL APROX. (Horas)
Documentación		
<i>Elaboración del DOP</i>	40	55
<i>Memoria del Trabajo Fin de Grado</i>	50	170
Captura de requisitos		
<i>Diagrama de Casos de uso y Jerarquía de actores</i>	6	6
<i>Modelo de Dominio</i>	6	10
<i>Casos de uso extendidos</i>	25	60
<i>Transformación MD en BBDD</i>	10	10
Módulos		
<i>Gestionar las colecciones</i>	62	160
<i>Registro e Identificación del Usuario</i>	32	40
<i>Acceder al catálogo completo</i>	18	35
<i>Acceder a una colección</i>	15	25
<i>Listar colecciones del usuario</i>	18	20
<i>Reto contra la máquina</i>	62	75
<i>Intercambiar cromos</i>	18	60
<i>Batalla</i>	100	80
<i>Buscar tesoro</i>	100	30
Duración total estimada	577	836

Tabla 58. Comparación de los tiempos

En relación a esto, hay que decir que, en algunas fases del proyecto se ha invertido probablemente demasiado tiempo en investigar temas más relacionados con pequeños detalles de diseño y animaciones que se salían del objetivo del proyecto y que hacían perder de vista la meta final. El hecho de no tener una fecha límite ha generado una falta de presión. Esto, sumado a querer mejorar cosas que ya estaban hechas, hace que se haya estancado el proyecto por momentos. Aprender a gestionar mejor los tiempos, marcarse uno mismo una fecha límite y comprender que es más importante sentar las bases y no detenerse demasiado tiempo en los pequeños detalles, es otra de las cosas que se ha aprendido y aportan un gran valor de cara al futuro.

Tal y como se puede observar hay más de 200 horas de diferencia. Si se analiza la tabla se ve como, por ejemplo, la duración de la elaboración de la memoria del proyecto triplica la duración estimada. No se contaba con que muchas de las cosas se han tenido que ir cambiando, volver a redactar, etc. Especialmente la parte de desarrollo ha sido la que más tiempo ha llevado. Tampoco se esperaba que la aplicación del administrador, *Gestionar las colecciones*, fuera a ser tan laboriosa. De haberlo sabido, en la estructura de descomposición del trabajo se habría dividido en módulos más pequeños, al igual que se hizo con la aplicación del usuario. A continuación se ofrece un gráfico con la comparativa entre la duración estimada y la duración real.

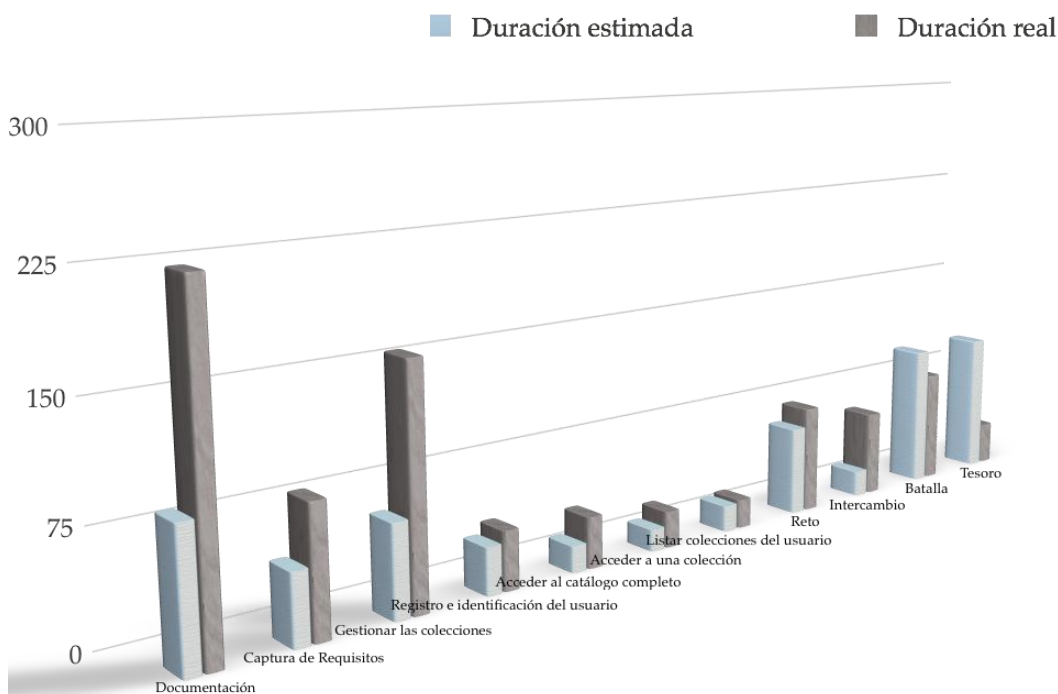


Figura 59. Gráfico de comparativa

Prácticamente todas las estimaciones se quedan cortas. *Batalla* es una excepción. Se estimaron 100 horas cuando en realidad ha costado unas 80. Fue de las últimas funcionalidades en ser implementadas, con lo que pudieron aprovecharse los

conocimientos adquiridos en fases anteriores. Al fin y al cabo es una funcionalidad que utiliza el juego de memoria que ya estaba implementado. Fue la gestión de datos lo que hizo que se necesitaran tantas horas, es decir, controlar bien las distintas posibilidades que podían darse en una batalla. La única tarea sobrestimada es la de la búsqueda del tesoro. A parte de ser lo último en desarrollarse, gracias a las librerías *CoreLocation* y *Mapkit* resultó ser más sencillo de lo esperado. La única pega, que retrasó algo el desarrollo de esta parte fue la actualización de XCode a *iOS 8* mencionada en el apartado Tesoro.

Casi con toda certeza, si hoy por hoy hubiera que repetir una aplicación de estilo similar, se estimaría con mayor precisión. Además, partiendo de los conocimientos adquiridos, se implementaría probablemente en la mitad del tiempo y con toda seguridad sería una aplicación más eficiente y funcional.

9 Trabajo futuro

En cualquier caso, no se descarta seguir trabajando en *ColeccionApp*, como se ha mencionado previamente se considera una aplicación en mejora continua. Hay infinidad de funcionalidades que se tenían en mente, pero para no retrasar aún más la duración de ésta primera versión finalmente han sido pospuestas para ampliaciones futuras. A continuación se mencionan las más importantes.

Desde el punto de vista del usuario, por ejemplo, cuando éste abre un sobre, directamente le aparece el álbum con los cromos de esa colección. Sin embargo, el usuario no sabe que cromo ha obtenido hasta que no va mirando el álbum detenidamente. La solución más optimista sería una animación rápida que pegará el cromo en el hueco que le corresponda. Si no fuera posible por la dificultad que conlleva, habría que generar al menos una vista intermedia que permitiera mostrar el cromo obtenido antes de que aparezca en el álbum.

Habría que mejorar también el funcionamiento de la aplicación cuando no hay conexión a internet. La base de datos interna de la aplicación no está bien aprovechada. En este punto hay mucho que mejorar ya que lo único que muestra la colección cuando no hay acceso a internet son las colecciones en las que participa el usuario. Obviamente, para gestionar intercambios, proponer batallas o buscar un tesoro sí que hace falta que haya conexión. Sin embargo, se podría hacer otras funcionalidades como superar los retos o empezar batallas recibidas, y que cuando se active la conexión a internet se sincronicen ambas bases de datos.

Para mejorar la gestión de batallas y de intercambio sería conveniente utilizar el servicio *Apple Push Notification Service* que ofrece Apple para gestionar las notificaciones. Aplicar este servicio a la aplicación supondría un importante salto de calidad. En lugar de que el usuario tenga que ir a las diferentes pestañas para ver si tiene solicitudes, podría recibir una notificación cada vez esto se produzca. Se le podría enviar una notificación cuando finalice la batalla que estaba en curso, o cuando un usuario responde a una solicitud de intercambio. Todo ello sin que el usuario tenga que estar pendiente del móvil.

Por otro lado, será necesario ampliar la gama de juegos para que el usuario no termine aburriéndose. Podrían añadirse variantes a los juegos ya existentes. Se podrían diseñar diferentes pantallas en el juego de habilidad, incluir un tiempo límite para superar los juegos, etc. También incluir un nuevo tipo de batalla que permita elegir el cromo que se quiere apostar.

Tal y como se menciona en el apartado Evaluación económica sería interesante incluir banners de publicidad a través *iAds*, así como aplicar el concepto *Ítem selling*. Por supuesto, para ello, habría que publicar la aplicación en la App Store y ofrecer mantenimiento y actualizaciones periódicas que vayan ampliando y mejorando la experiencia del usuario.

Probablemente la mejora que aportaría un mayor valor a la aplicación sería ampliar la gama de los juegos disponibles. Sin embargo, si hubiera que elegir una mejora a implementar en primer lugar, desarrollar más juegos llevaría demasiado tiempo. Probablemente sería más factible implementar el servicio de notificaciones proporcionado por Apple. Sería una aplicación mucho más práctica y atractiva que mejoraría sustancialmente la experiencia del usuario. Por otro lado, desde el punto de vista del desarrollador es una funcionalidad muy utilizada y probablemente no requiera de demasiadas horas de implementación. Tampoco sería complicado hacer que se muestre el cromó que ha ganado el usuario y que no se ha realizado simplemente porque no se cayó en la cuenta hasta que un usuario lo mencionó, cuando la aplicación ya estaba finalizada.

Quizás la mejora que se tardaría más en implementar sería la que está más relacionada con el marketing y la evaluación económica. Antes de llegar a ese punto, habría que valorar primero el potencial de la aplicación que, a día de hoy, es desconocido. Se trata de que la aplicación se haga popular. Si se empieza desde el principio con publicidad y venta de artículos la aplicación podría suscitar rechazo. Si en el futuro llegase a publicarse y, se tuviera la suerte de obtener un número elevado de descargas, pasaría a tener una prioridad mayor llegando a incluso a plantearse la posibilidad de extender el juego a otras plataformas como Android o Firefox..

10 Bibliografía

10.1 Libros

- Abhishek Mishra, Gene Backlin. iPhone y iPad Manual práctico de desarrollo, Anaya
- John Ray, iOS 6 Application Development, Sams Teach Yourself

10.2 Sitios Web

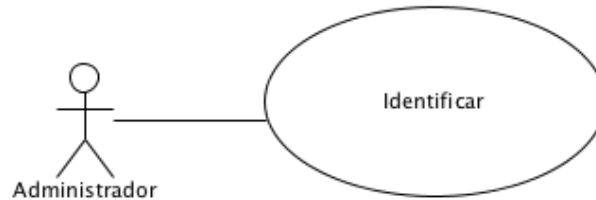
- Apple Inc., 2013, ¿Cientos de miles de apps? No, cientos de miles de apps alucinantes. 15/11/2013, <https://www.apple.com/es/iphone-5c/app-store/>, Pág. 7.
- 2013 Forbes.com LLC™. How Much Do Average Apps Make? 8/10/2013. <http://www.forbes.com/sites/tristanlouis/2013/08/10/how-much-do-average-apps-make/> Pág. 7.
- © 2013 Google, Google Play - Aplicaciones: Equipazo Virtual 2013-, 17/09/2013 <https://play.google.com/store/apps/details?id=air.com.generamobile.equipazovirtual&hl=es>, Pág. 41.
- © 2013 Zynga, Inc. War of the Fallen, <http://zynga.com/game/war-of-the-fallen>, Pág. 41.
- Applesfera, 2012, CSR Racing, el juego gratuito más rentable de la App Store genera 12 millones de dólares al mes. 18/08/2012, <http://www.applesfera.com/aplicaciones-ios-1/csr-racing-el-juego-gratuito-mas-rentable-de-la-app-store-genera-12-millones-de-dolares-al-mes>, Pág. 41.
- 2013 videogaming247 Ltd, Supercell cashing in on free-to-play tablet titles, 10/10/2012, <http://www.vg247.com/2012/10/10/supercell-cashing-in-on-free-to-play-tablet-titles/>, Pág. 41.
- DIARIO ABC, S.L, El «efecto Zeigarnik» o por qué el Candy Crush crea adicción, 08/08/2013, <http://www.abc.es/tecnologia/moviles-aplicaciones/20130808/abci-candy-crush-saga-adictivo-201308071929.html>, Pág. 41.
- STARTCAPP, Como ganan dinero las aplicaciones y juegos para móviles, 22/06/2013 <http://www.startcapps.com/blog/como-ganan-dinero-las-aplicaciones-y-juegos-para-moviles/> Pág. 41.

ANEXO I

Casos de uso

Administrador

Nombre: Identificar



Descripción: El administrador se identifica en el sistema y accede a su panel de inicio.

Actores: Administrador

Precondiciones: No estar ya identificado

Requisitos no funcionales: Ninguno

Flujo de eventos:

1. El administrador introduce su usuario y contraseña y pulsa 'Entrar', Imagen 1
2. [\[Si el usuario y la contraseña son correctos\]](#)
 - 2a. Accede al caso de uso *Gestionar colecciones*.[\[Si no\]](#)
 - 2b. Aparece un mensaje de error. El administrador pulsa 'Aceptar' y vuelve a la ventana de inicio, Imagen 2

Pos condiciones: Ninguna

Interfaz gráfica:



Imagen 1. Vista inicial

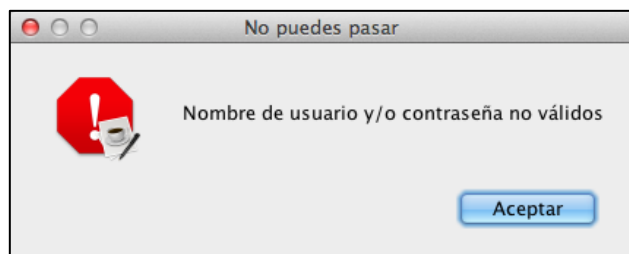
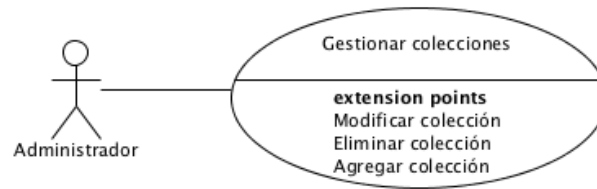


Imagen 2. Datos de acceso no válidos.

Nombre: Gestionar colecciones

Descripción: Permite al administrador gestionar cada una de las colecciones. Puede modificar, eliminar y añadir colecciones.

Actores: Administrador

Precondiciones: Estar identificado

Requisitos no funcionales: Ninguno

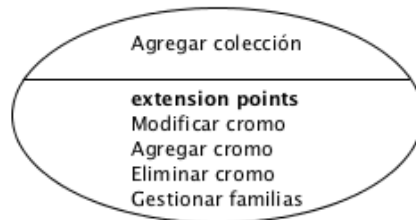
Flujo de eventos:

1. El administrador accede a la pantalla de inicio del Gestor de Colecciones. En esta vista se cargan todas las colecciones existentes, Imagen 3.
[Si pulsa 'Agregar colección']
 - 2a. Accede al subcaso de uso *Agregar colección*.
[Si selecciona una colección y pulsa 'Modificar colección']
 - 2b. Accede al subcaso de uso *Modificar colección*.
[Si selecciona una colección y pulsa 'Eliminar colección']
 - 2c. Accede al subcaso de uso *Eliminar colección*.

Pos condiciones: Ninguna

Interfaz gráfica:

Imagen 3. Listado de todas las colecciones.

Nombre: Agregar colección (*subcaso de uso*)

Descripción: Subcaso de uso que permite al administrador crear una nueva colección. Puede agregar, modificar, eliminar cromos y gestionar familias.

Actores: Administrador

Precondiciones: Estar identificado

Requisitos no funcionales: Ninguno

Flujo de eventos:

1. En primer lugar se muestra una pequeña ventana donde rellenar los datos de la colección; nombre, imagen y una breve descripción, Imagen 4
[\[Si selecciona 'Crear colección'\]](#)
[\[Si los campos requeridos no están correctamente rellenos\]](#)
 - 2aa. Aparece un mensaje de error que indica que hay datos no válidos, Imagen 5
[\[Si están correctamente rellenos\]](#)
 - 3aba. Aparece un mensaje de error que indica que el nombre ya existe, Imagen 6
[\[Si el nombre de la colección ya existe\]](#)
 - 3abb. Aparece un mensaje de confirmación, ¡Error! No se encuentra el origen e la referencia.
[\[Si selecciona 'No'\]](#)
 - 4abba. No se crea la colección. El administrador vuelve a la ventana anterior donde puede modificar los datos de la colección.
[\[Si selecciona 'Sí'\]](#)
 - 4abbb. Se crea la colección vacía en la base de datos y aparece la vista 'Gestionar una colección'. Esta ventana permite agregar cromos existentes de otras colecciones, crear nuevos, modificar y eliminarlos. Además permite gestionar todas las familias, Imagen 11
[\[Si pulsa 'Agregar cromos'\]](#)
 - 5abbba. Accede al subcaso de uso *Agregar cromos*.
[\[Si selecciona un cromos y pulsa 'Modificar cromos'\]](#)
 - 5abbbb. Accede al subcaso de uso *Modificar cromos*.
[\[Si selecciona un cromos y pulsa 'Eliminar cromos'\]](#)
 - 5abbbc. Accede al subcaso de uso *Eliminar cromos*.
[\[Si pulsa 'Gestionar familias'\]](#)
 - 5abbbd. Accede al subcaso de uso *Gestionar familias*.
[\[Si selecciona un cromos de una colección de abajo y pulsa en la flecha que apunta hacia arriba\]](#)
[\[Si el numero de ese cromos está ocupado\]](#)

6abbbea. Aparece una ventana para cambiarlo, Imagen 12

[Si el numero de ese cromó no está ocupado]

6abbbea. Se crea el cromó seleccionado en la colección en la que el administrador se encuentra, Imagen 13.

[Si pulsa 'Cerrar']

5abbbf. Accede al caso de uso *Gestionar colecciones*.

Pos condiciones: Se agrega una colección

Interfaz gráfica:

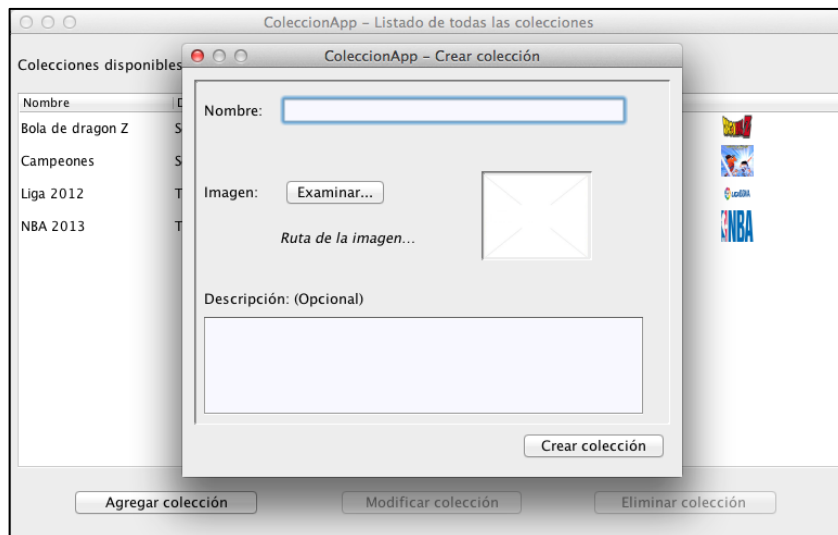


Imagen 4. Vista crear colección

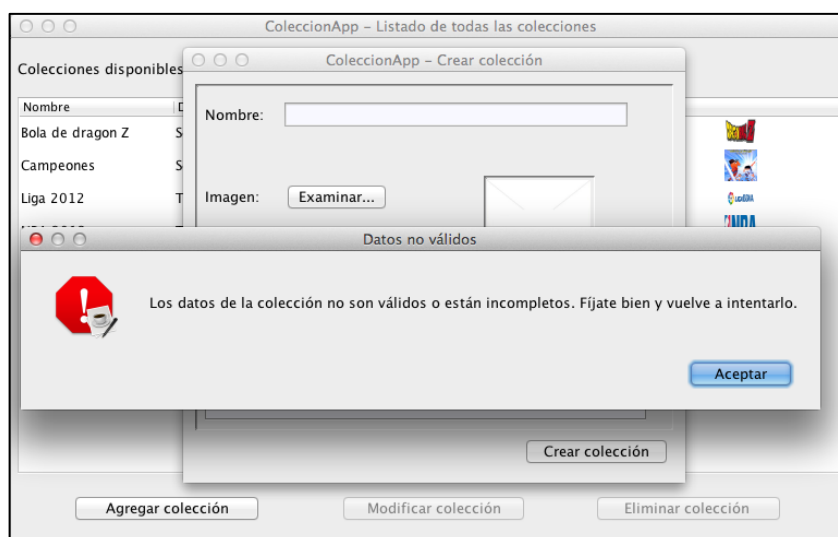


Imagen 5. Datos de la colección no válidos

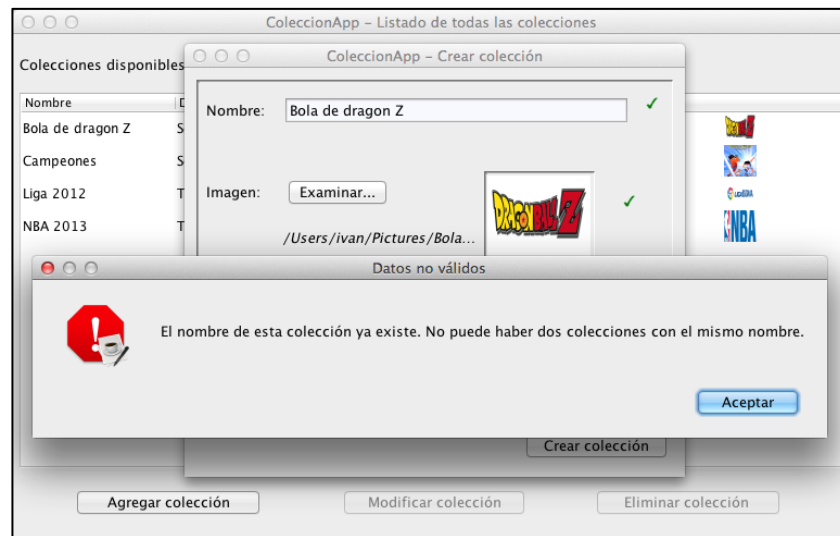


Imagen 6. Nombre de colección ya existente

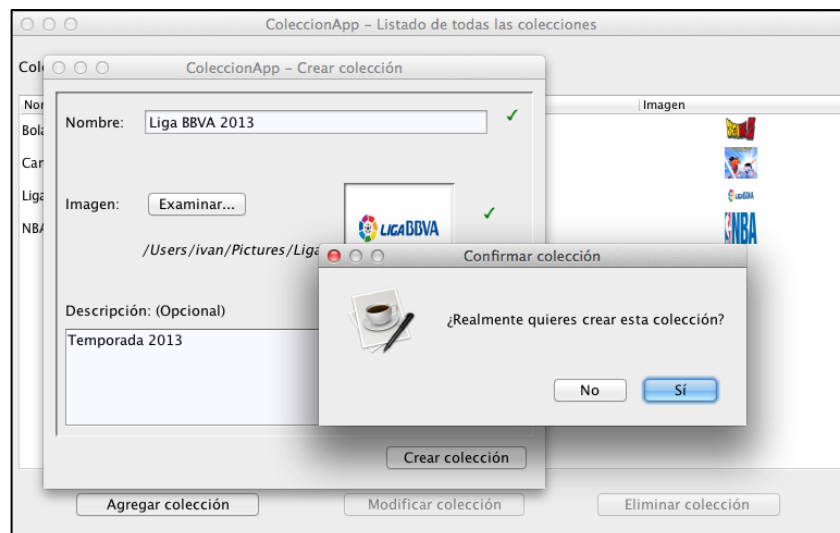


Imagen 7. Confirmación para crear colección

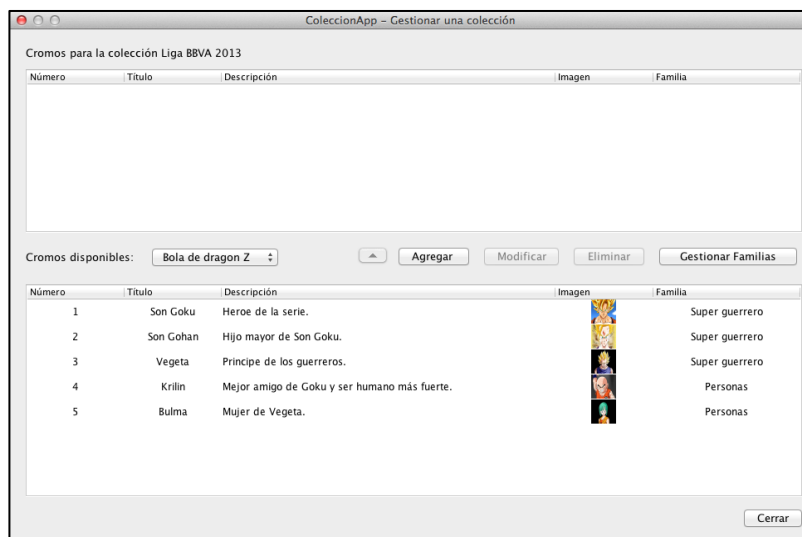


Imagen 8. Gestionar una colección

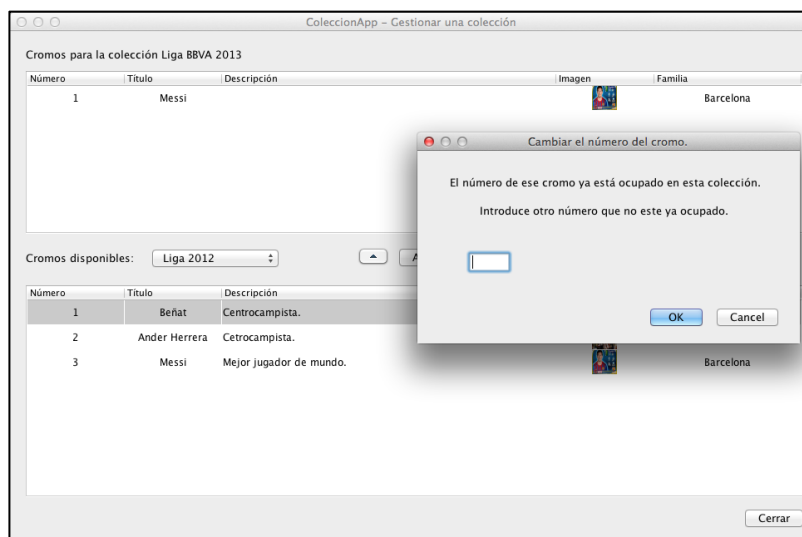


Imagen 9. Cambiar el número del cromo

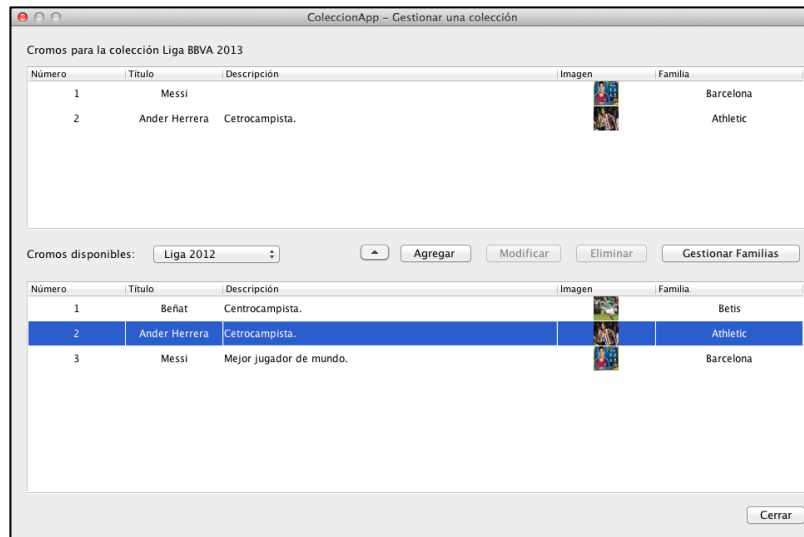
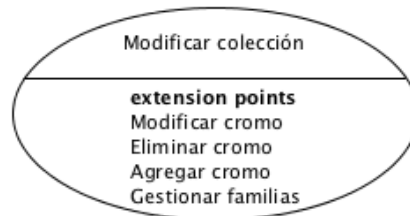


Imagen 10. Aprovechar cromo existente

Nombre: Modificar colección (*subcaso de uso*)

Descripción: Subcaso de uso que permite al administrador modificar colección existente. Puede agregar, modificar, eliminar cromos y gestionar familias.

Actores: Administrador

Precondiciones: Estar identificado y que exista alguna colección

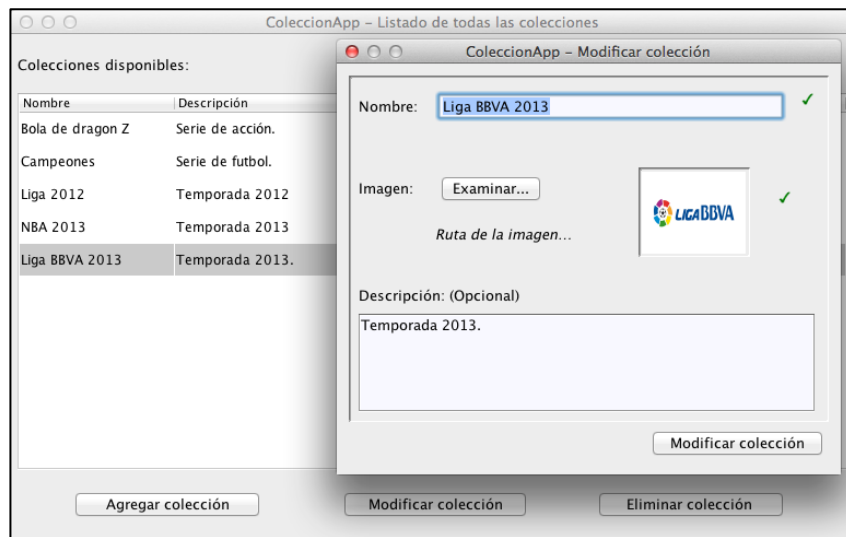
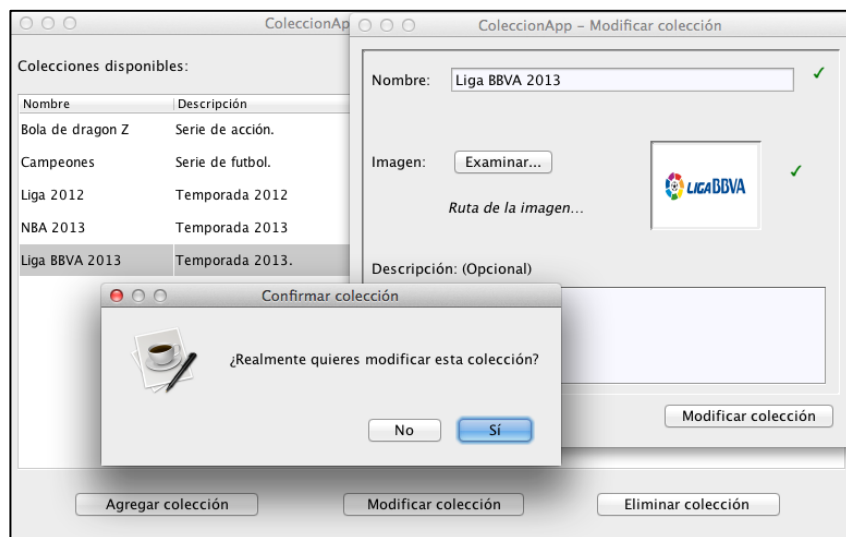
Requisitos no funcionales: Ninguno

Flujo de eventos:

1. Se muestra una pequeña ventana completa con los datos de la colección seleccionada. Es posible modificar los datos de la colección; nombre, imagen y una breve descripción. Ver Imagen 11.
2. Cambia los datos si quiere y pulsa 'Modificar colección'
 - [Si los campos requeridos no están correctamente rellenos]
 - 2a. Aparece un mensaje de error que indica que hay datos no válidos, Imagen 5 [Si están correctamente rellenos]
 - [Si cambia el nombre de la colección por uno que ya existe]
 - 3ba. Aparece un mensaje de error que indica que el nombre ya existe, Imagen 6
 - [Si el nombre de la colección no existe]
 - 3bb. Aparece un mensaje de confirmación, Imagen 12.
 - [Si selecciona 'No']
 - 4bba. No se modifica la colección. El administrador vuelve a la ventana anterior donde puede modificar los datos de la colección.
 - [Si selecciona 'Sí']
 - 4bbb. Se accede a la vista 'Gestionar una colección'. Es la misma ventana que aparece cuando se crea una colección con la única diferencia de que se cargan los cromos de la colección que se está modificando.

** A partir de este punto el funcionamiento es el mismo que en el caso de uso Agregar colección.*

Pos condiciones: Se modifica una colección

Interfaz gráfica:**Imagen 11. Modificar colección****Imagen 12. Confirmar modificación**

Nombre: Eliminar colección (*subcaso de uso*)



Descripción: Subcaso de uso que permite al administrador eliminar una colección existente.

Actores: Administrador

Precondiciones: Estar identificado y que exista alguna colección

Requisitos no funcionales: Ninguno

Flujo de eventos:

1. Aparece un mensaje de confirmación, ver Imagen 13.
[Si pulsa 'Sí']
- 2a. La colección se elimina.
[Si pulsa 'No']
- 2b. La colección no se elimina.
3. El administrador vuelve a la pantalla de gestión de colecciones.

Pos condiciones: Se elimina una colección

Interfaz gráfica:

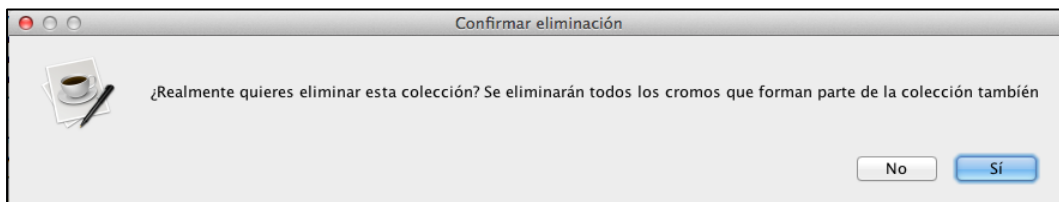


Imagen 13. Confirmar eliminación.

Nombre: Agregar cromo (subcaso de uso)**Descripción:** Subcaso de uso que permite crear un nuevo cromo y añadirlo a la colección.**Actores:** Administrador**Precondiciones:** Estar identificado,**Requisitos no funcionales:** Ninguno**Flujo de eventos:**

1. Accede al formulario para rellenar los campos del nuevo cromo, Imagen 14.

[Si pulsa 'Agregar']

[Si los campos requeridos están correctamente rellenos]

2aaa. Aparece un mensaje de confirmación, Imagen 15.

[Si pulsa 'Sí']

3aaaa. El cromo se crea y se añade a la colección. El administrador vuelve a la pantalla de gestión de cromos, donde el aparece el nuevo cromo creado, Imagen 16.

[Si pulsa 'No']

3aaab. El cromo no se crea. El administrador vuelve al formulario donde rellenar los campos del cromo.

[Si los campos requeridos no están correctamente rellenos]

2aab. Aparece un mensaje de error. El administrador pulsa 'Aceptar' y vuelve a la pantalla de gestión de cromos, Imagen 17.

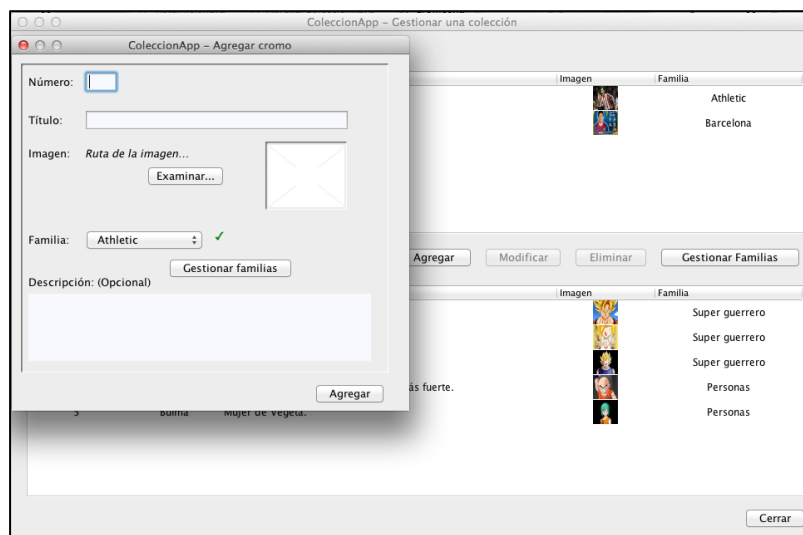
Pos condiciones: Se agrega un cromo, el número del cromo ha de estar comprendido entre el número 1 y el total de cromos de la colección.**Interfaz gráfica:**

Imagen 14. Agregar cromo.

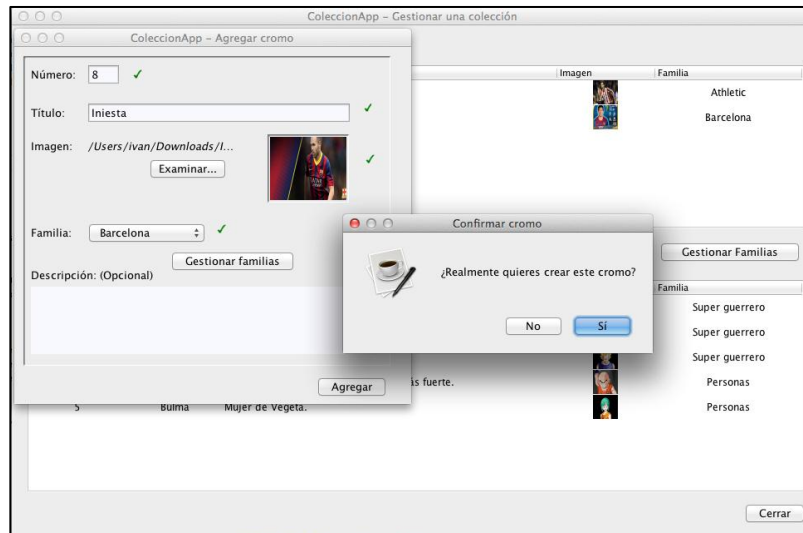


Imagen 15. Datos del cromo válidos.

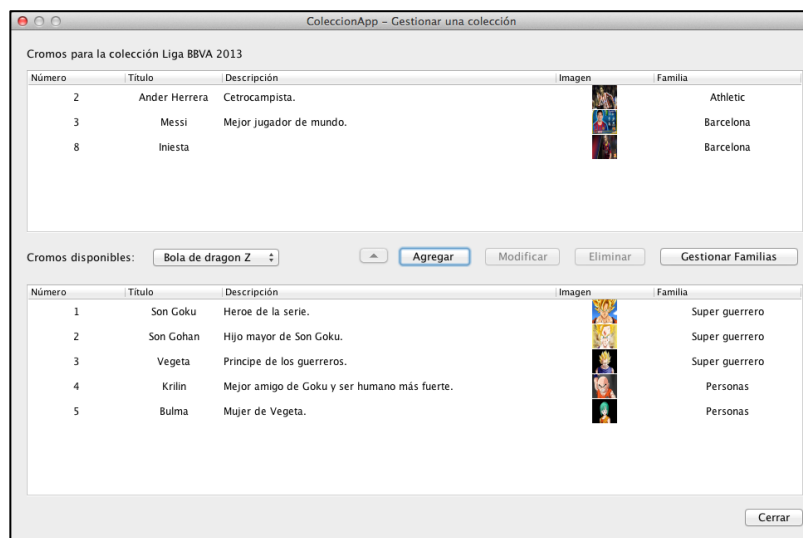


Imagen 16. Cromo añadido a la colección Liga BBVA 2013 correctamente.

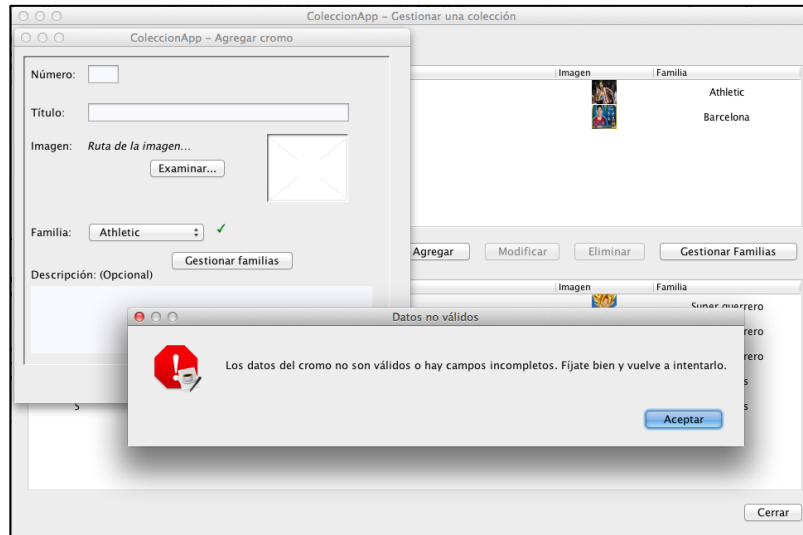


Imagen 17. Datos del cromo no válidos.

Nombre: Modificar cromos (subcaso de uso)**Descripción:** Subcaso de uso que permite al administrador modificar un cromos existente.**Actores:** Administrador**Precondiciones:** Estar identificado y que exista algún cromos**Requisitos no funcionales:** Ninguno**Flujo de eventos:**

1. Accede al formulario para cambiar los campos del cromos seleccionado, Imagen 18 .
 [Si pulsa 'Modificar']
 [Si los campos requeridos están correctamente rellenos]
 2aaa. Aparece un mensaje de confirmación, Imagen 19.
 [Si pulsa 'Si']
 3aaaa. El cromos se guarda con los cambios realizados. El administrador vuelve a la pantalla de gestión de cromos, Imagen 20.
 [Si pulsa 'No']
 3aaab. El cromos no se modifica. El administrador vuelve a la pantalla donde puede modificar el cromos.
 [Si los campos requeridos no están correctamente rellenos]
 2aab. Aparece un mensaje de error. El administrador pulsa 'Aceptar' y vuelve a la pantalla de gestión de cromos, Imagen 21.

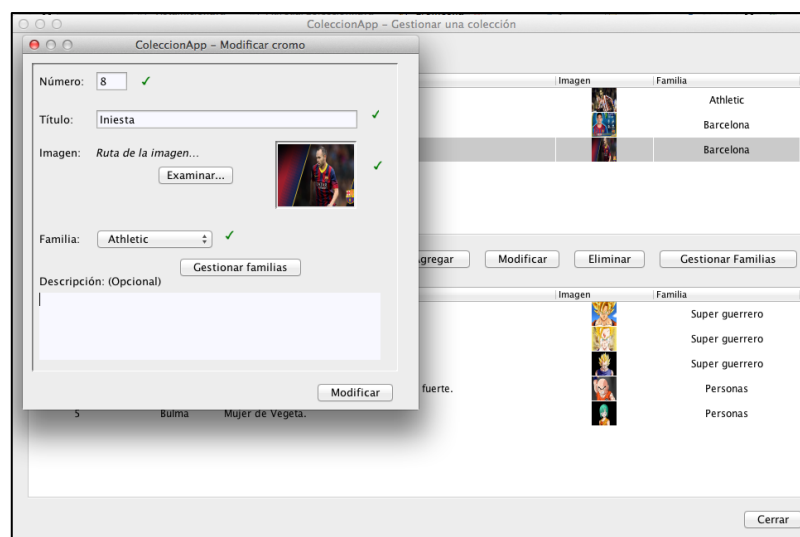
Pos condiciones: Se modifica un cromos, el número del cromos ha de estar comprendido entre el número 1 y el total de cromos de la colección.**Interfaz gráfica:**

Imagen 18. Modificar cromos.

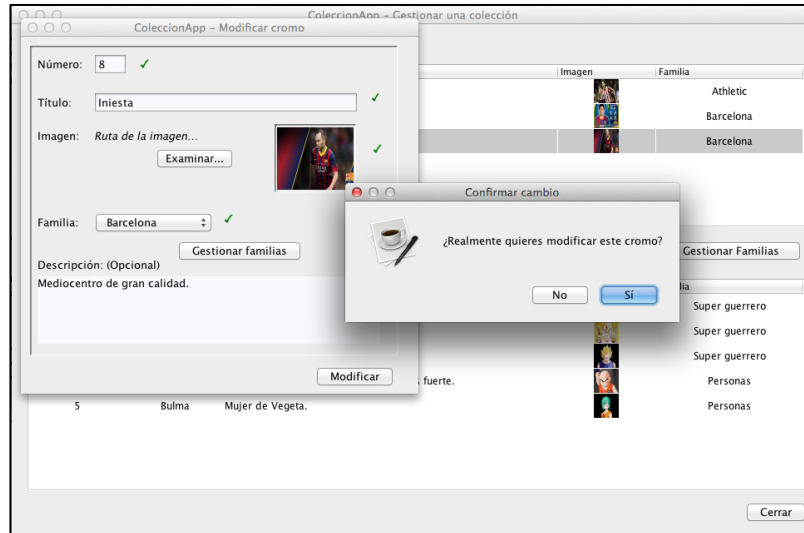


Imagen 19. Modificar cromo, datos válidos.

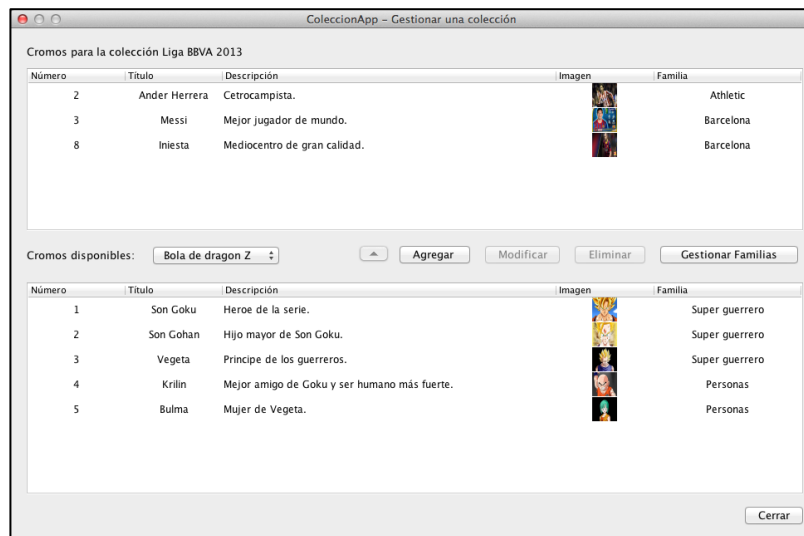


Imagen 20. Cromo modificado correctamente.

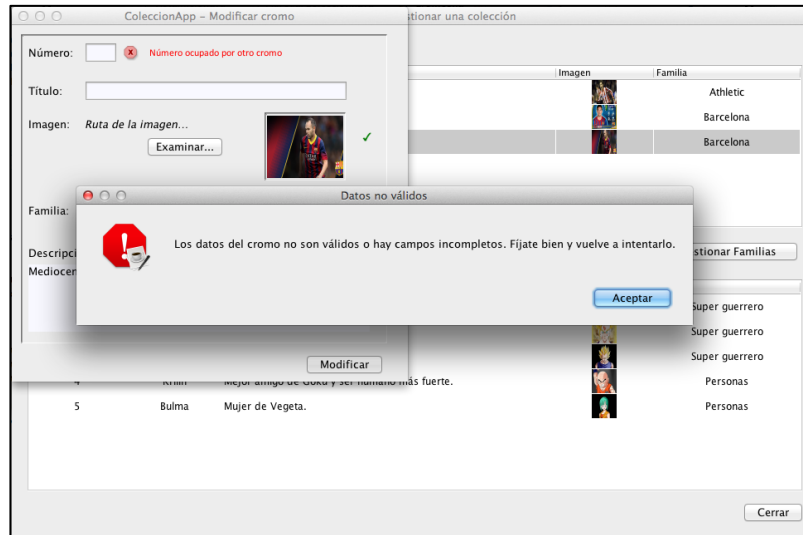


Imagen 21. Modificar cromos, datos no válidos.

Nombre: Eliminar cromo (*subcaso de uso*)



Descripción: Subcaso de uso que permite al administrador eliminar un cromo de una colección

Actores: Administrador

Precondiciones: Estar identificado y que exista algún cromo

Requisitos no funcionales: Ninguno

Flujo de eventos:

1. Aparece un mensaje de confirmación, Imagen 22.
 - [Si pulsa 'Sí']
 - 2a. El cromo se elimina. El administrador vuelve a la gestión de cromos ya sin el cromo eliminado.
 - [Si pulsa 'No']
 - 2b. El cromo no se elimina. El administrador vuelve a la gestión de cromos.

Pos condiciones: Se elimina un cromo

Interfaz gráfica:

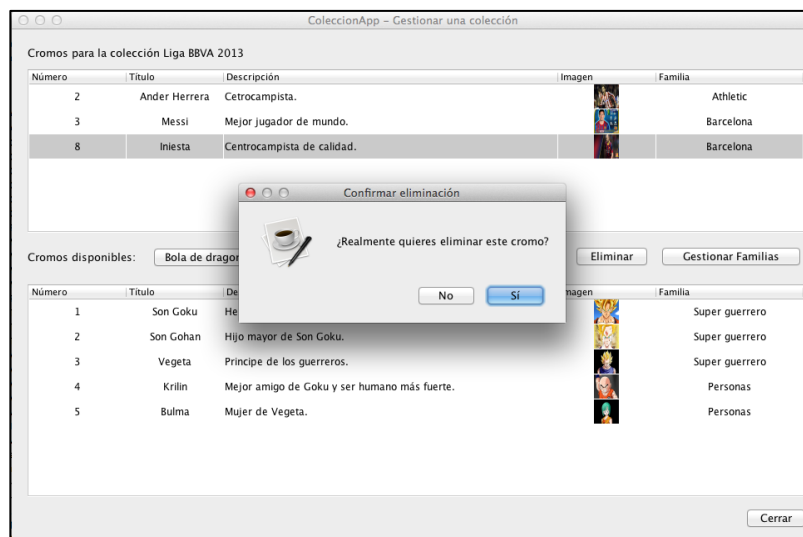
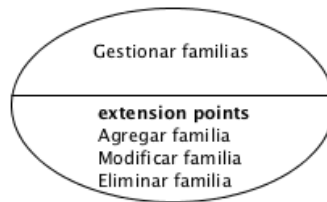


Imagen 22. Eliminar cromo.

Nombre: Gestionar familias (*subcaso de uso*)



Descripción: Subcaso de uso que permite al administrador gestionar familias. Puede agregar, modificar y eliminarlas.

Actores: Administrador

Precondiciones: Estar identificado.

Requisitos no funcionales: Ninguno

Flujo de eventos:

1. Accede a la pantalla donde puede gestionar las familias. En el recuadro de la izquierda se muestran las familias para la colección que se está gestionando. En el recuadro de la derecha se muestran todas las familias disponibles. Es posible filtrarlas por colección mediante el desplegable, Imagen 23.

[Si selecciona una familia del cuadro de la izquierda y pulsa sobre '>']

[Si la familia no está siendo usada por ningún cromó]

3aa. Se elimina de la colección y desaparece del recuadro de la izquierda.

[Si la familia está siendo usada por algún cromó]

3ab. Aparece un mensaje de error indicando que no se puede eliminar, Imagen 24.

[Si selecciona una familia del cuadro de la derecha y pulsa sobre '<']

2b. Se añade la familia a la colección y aparece en el cuadro de la izquierda, Imagen 25.

[Si selecciona una familia y pulsa 'Agregar familia']

2c. Aparece un pequeño recuadro para añadir la nueva familia, Imagen 26.

3c. Introduce un nombre y pulsa 'Agregar'.

[Si el nombre de la familia no existe]

4ca. Aparece un cuadro de confirmación, Imagen 27.

[Si selecciona 'Sí']

5caa. Se añade la familia a ambos cuadros, Imagen 28.

[Si selecciona 'No']

5cab. Vuelve al recuadro donde puede introducir de nuevo el nombre de la familia que desea agregar.

[Si el nombre de la familia ya existe]

4cb. Aparece un mensaje de error indicándolo, Imagen 29.

[Si selecciona una familia y pulsa 'Modificar familia']

2d. Aparece una ventana de advertencia, Imagen 30.

[Si selecciona 'Sí']

3da. Aparece un pequeño recuadro para añadir la nueva familia, Imagen 31.

4da. Introduce un nombre y pulsar 'Modificar'

[Si el nombre de la familia no existe]

5daa. Aparece un cuadro de confirmación, Imagen 32.

[Si selecciona 'Sí']

6daaa. Se modifica la familia en todos los cromos que la tengan asociada, Imagen 33.

[Si selecciona 'No']

6daab. No se modifica la familia. Vuelve al recuadro donde puede introducir de nuevo el nombre de la familia que desea modificar.

[Si el nombre de la familia ya existe]

5dab. Aparece un mensaje de error indicándolo, Imagen 34.

[Si selecciona una familia y pulsa 'Eliminar familia']

[Si la familia no esta incluida en ninguna colección]

3ea. Aparece un cuadro de confirmación, Imagen 35.

[Si selecciona 'Sí']

4eaa. Se elimina del listado de todas las familias y desaparece del recuadro de la derecha.

[Si selecciona 'No']

4eab. Vuelve al 'Gestor de familias'

[Si la familia esta incluida en alguna colección]

3eb. Aparece un mensaje de error indicando que no se puede eliminar, Imagen 36.

[Si pulsa 'Cerrar']

2f. Se cierra el 'Gestor de familias' y se accede al gestor de una colección.

Pos condiciones: Ninguna

Interfaz gráfica:

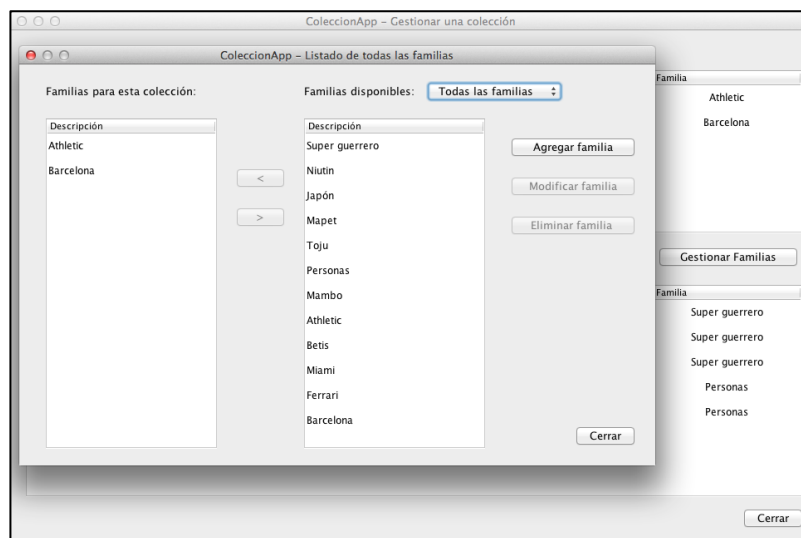


Imagen 23. Gestionar familias.

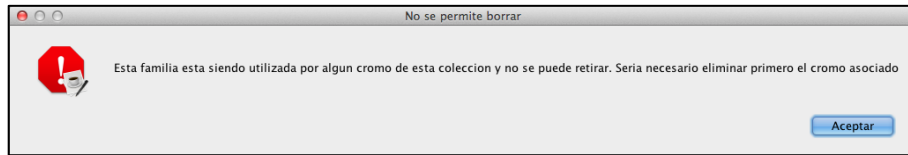


Imagen 24. Familia en uso por algún cromo de esta colección.

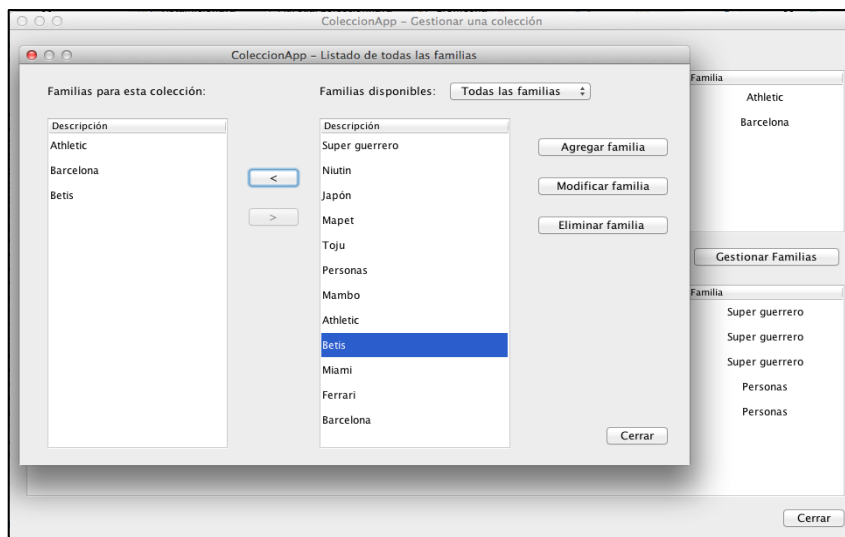


Imagen 25. Agregar familia existente.

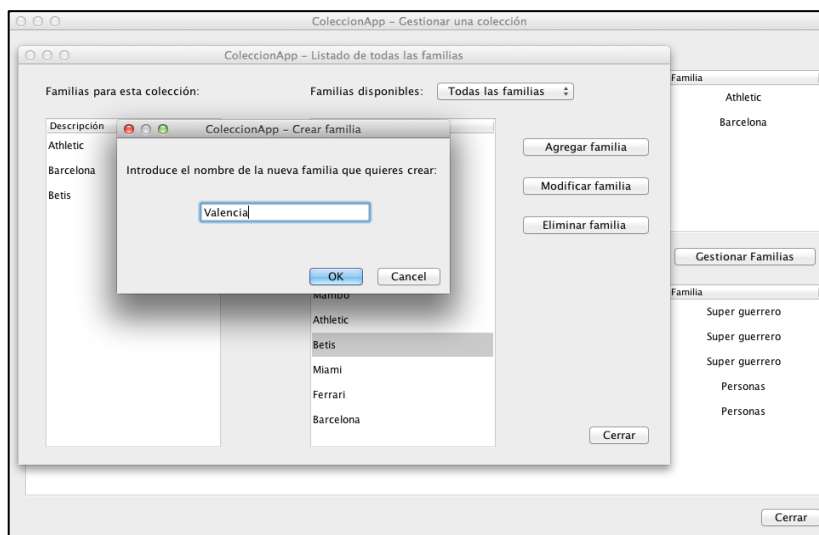


Imagen 26. Agregar una familia que no existe.

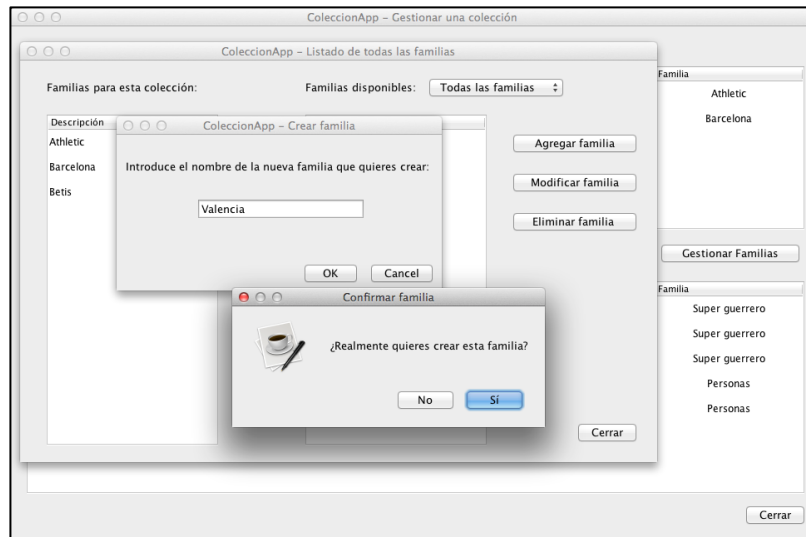


Imagen 27. Agregar familia, mensaje de confirmación.

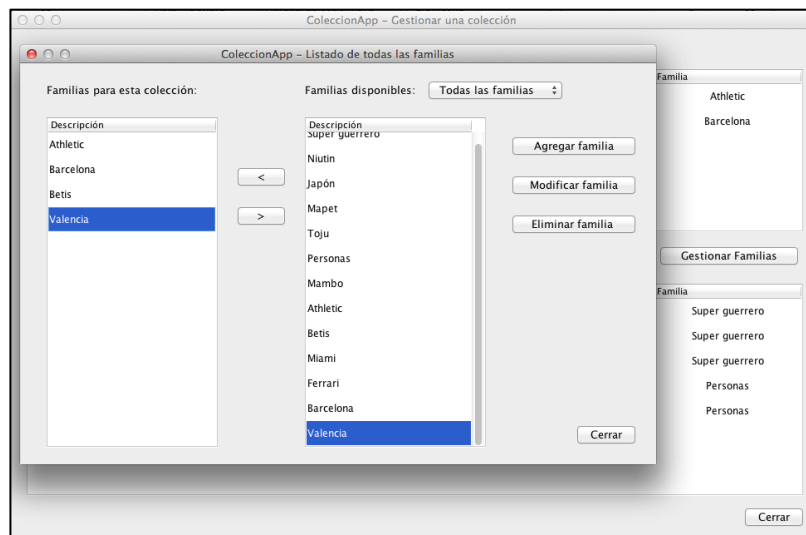


Imagen 28. Familia agregada correctamente.

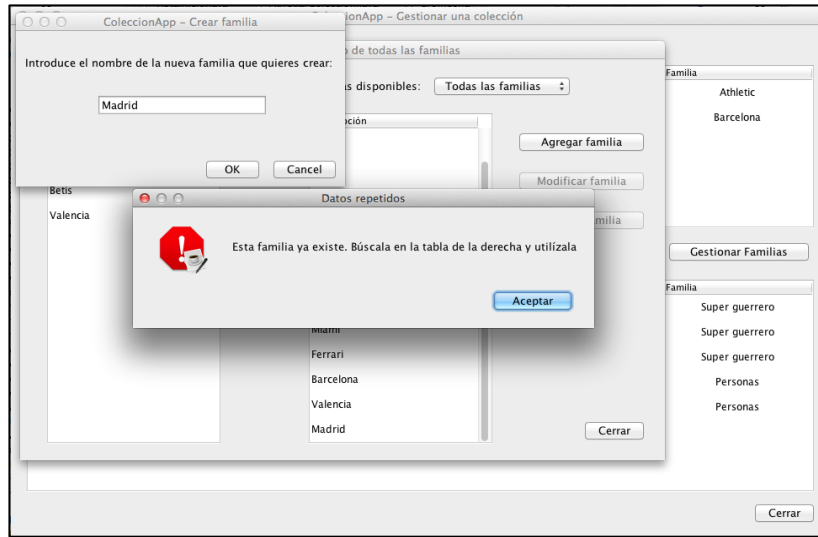


Imagen 29. Intentando agregar una familia que ya existe.

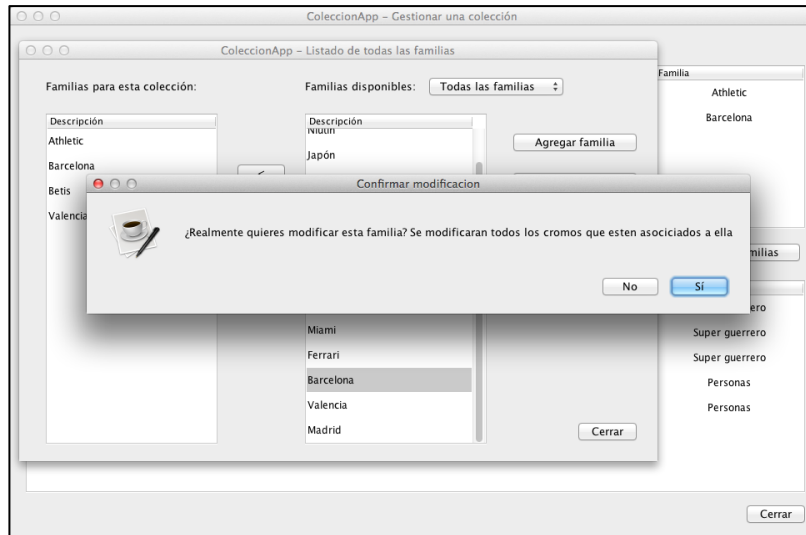


Imagen 30. Modificar familia. Advertencia.

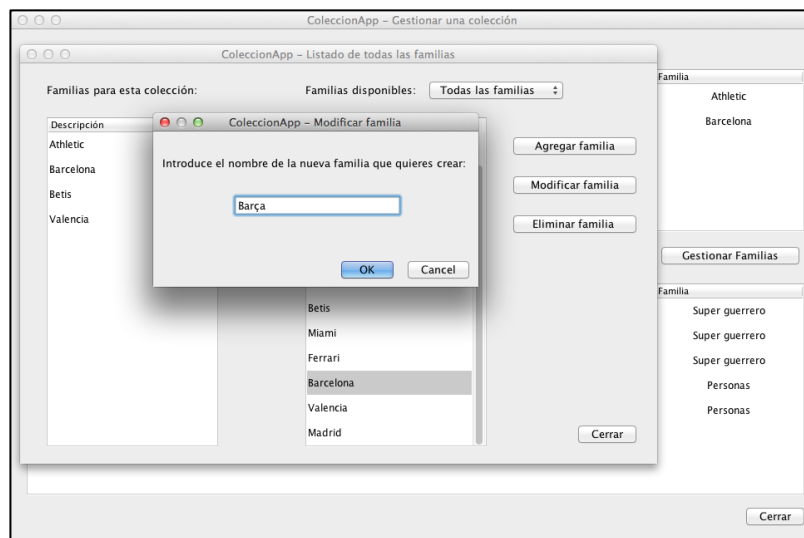


Imagen 31. Modificar familia, cuadro de texto.

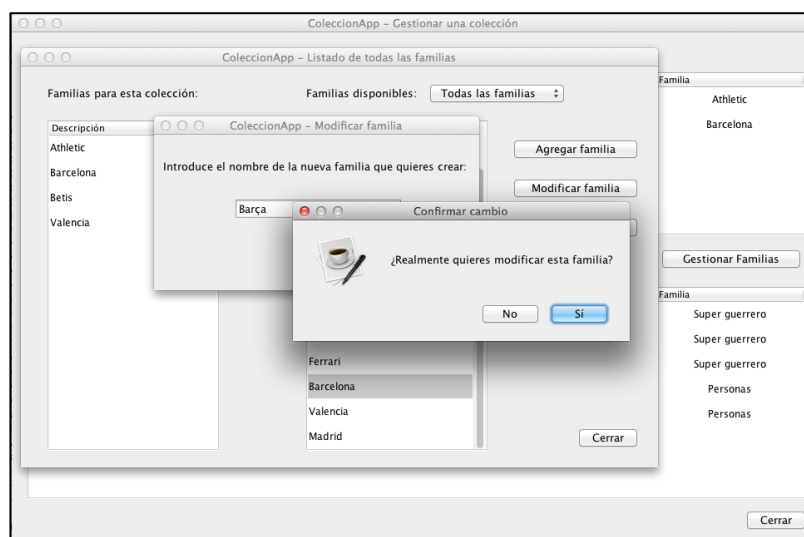


Imagen 32. Modificar familia. Confirmación.

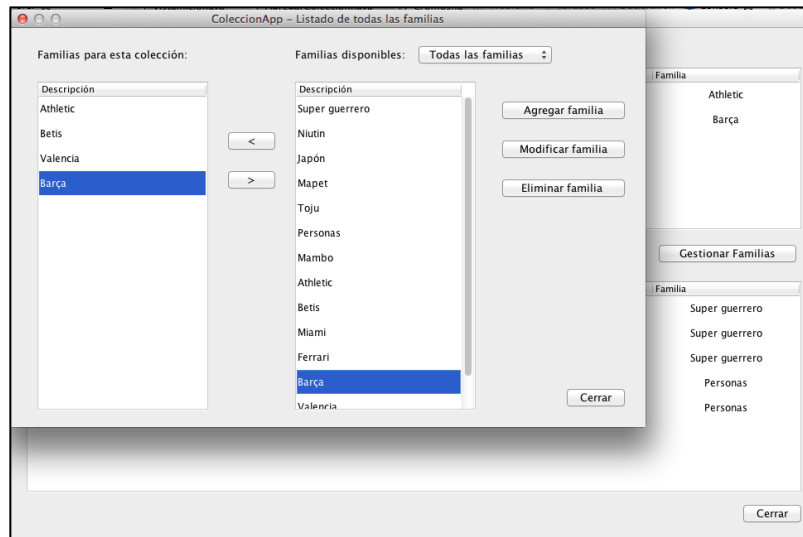


Imagen 33. Familia modificada correctamente.

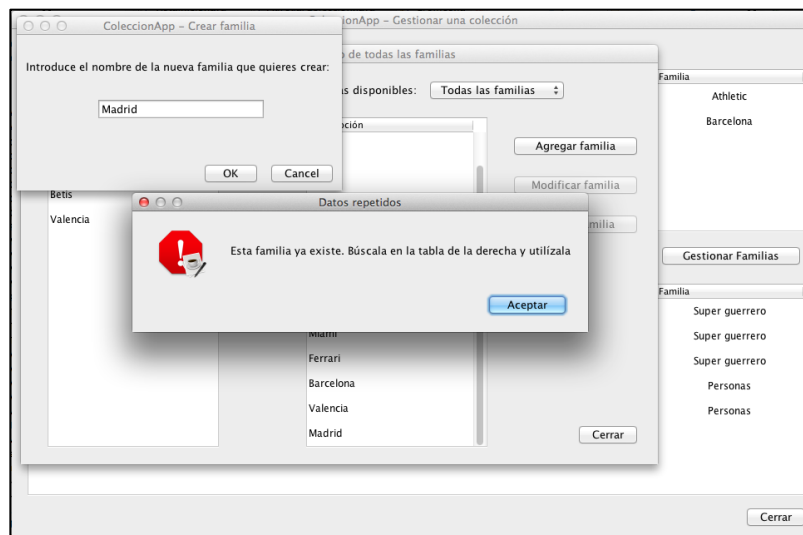


Imagen 34. Modificar familia. Error, la familia ya existe.

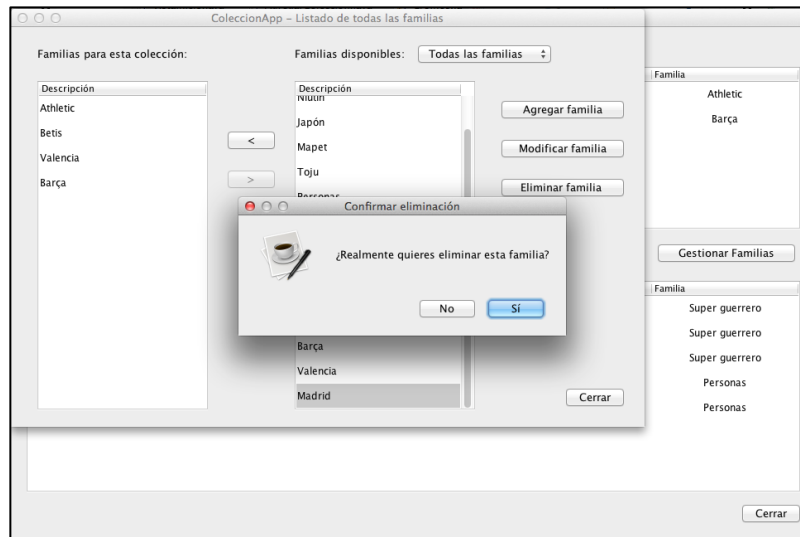


Imagen 35. Eliminar familia. Confirmación.

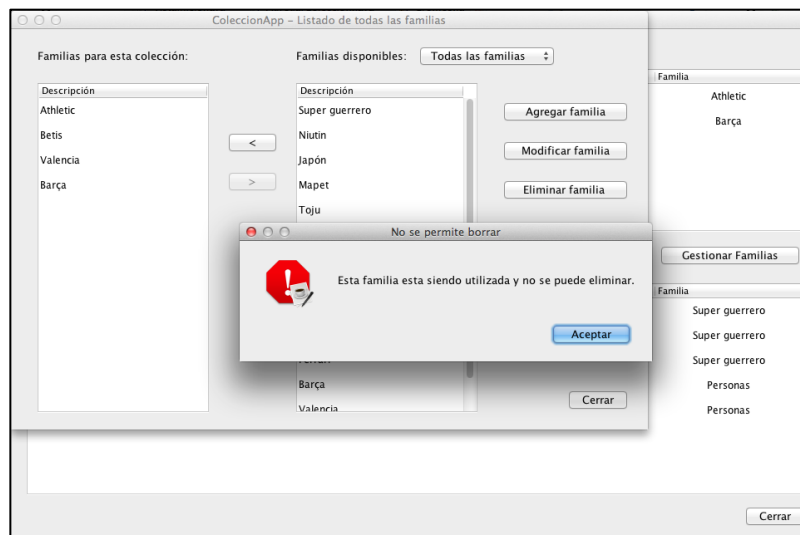
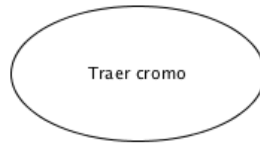


Imagen 36. Eliminar familia. Error, familia en uso por alguna colección.

Nombre: Traer cromos (*subcaso de uso*)


Descripción: Subcaso de uso que permite al administrador agregar un cromos existente en otra colección a la colección que se está gestionando.

Actores: Administrador

Precondiciones: Estar identificado, que exista alguna colección que tenga algún cromos.

Requisitos no funcionales: Ninguno

Flujo de eventos:

1. El administrador selecciona un cromos de la tabla de abajo y pulsa .
 - [Si el cromos ya forma parte de la colección]
 - 2a. Aparece una ventana de advertencia, Imagen 37.
 - [Si el número del cromos está ocupado en la colección]
 - 2b. Aparece una ventana que pide que se introduzca un número no ocupado. Cuando el administrador introduzca un número no ocupado aparecerá un tic verde, Imagen 38.
 - 3b. Introduce un número válido y pulsa 'ok'.
 - 4b. Aparece una ventana de confirmación, Imagen 39.
 - [Si no]
 - 2c. El cromos se agrega correctamente a la colección. Pasa de la tabla de abajo a la tabla de arriba, Imagen 40.

Pos condiciones: Se añade un cromos de una colección existente

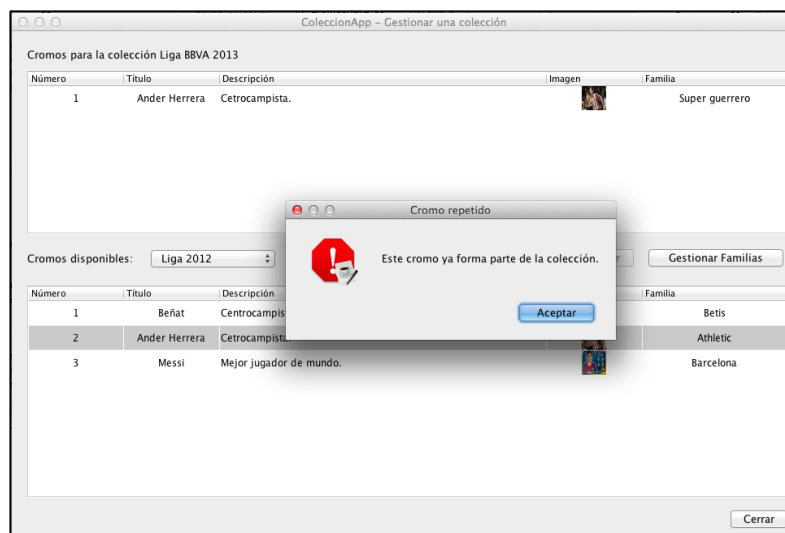
Interfaz gráfica:

Imagen 37. Cromos ya agregado.

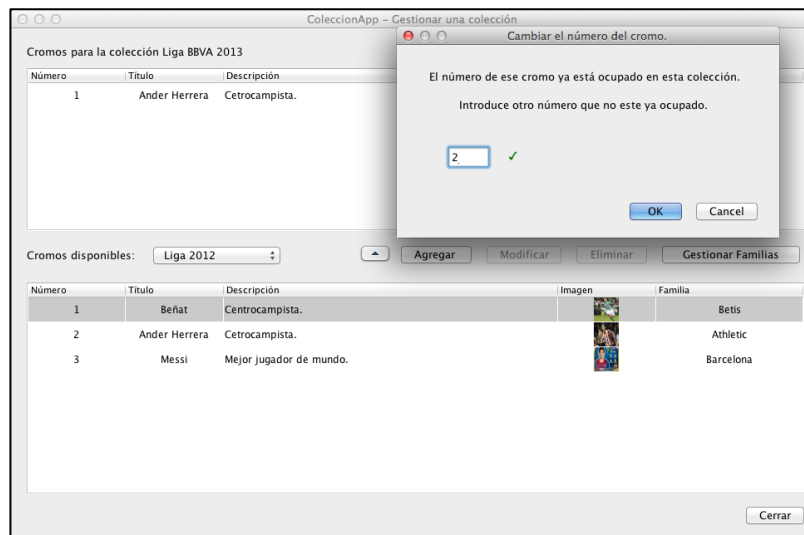


Imagen 38. Número de cromo ocupado.

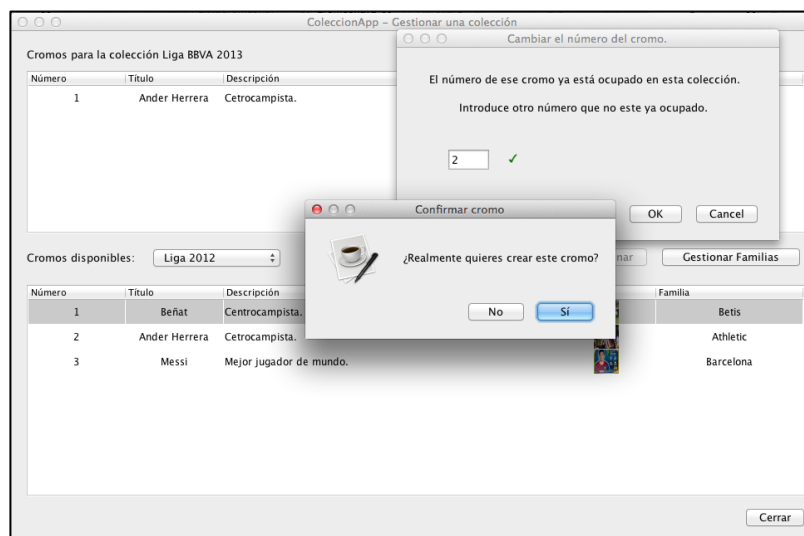


Imagen 39. Confirmar cromo.

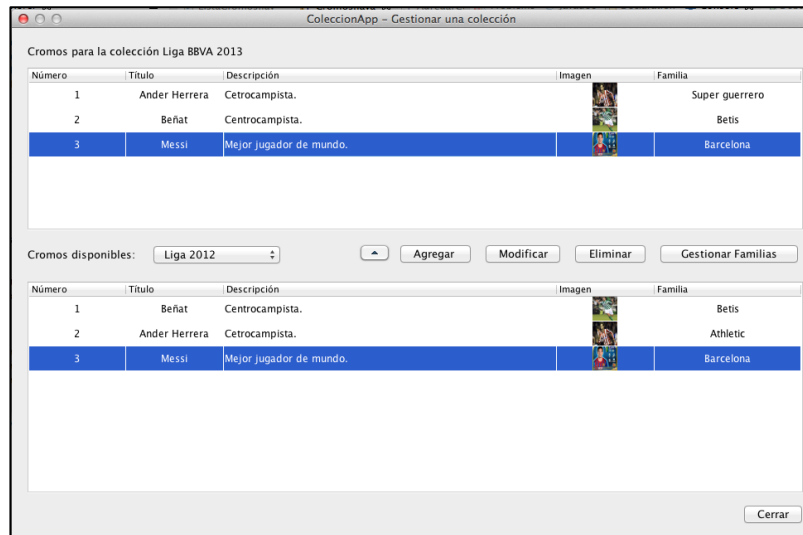
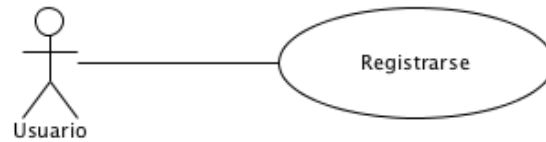


Imagen 40. Cromo agregado correctamente.

Usuario

Nombre: Registrarse



Descripción: Permite al usuario registrarse en la aplicación.

Actores: Usuario

Precondiciones: No estar ya registrado

Requisitos no funcionales: Ninguno

Flujo de eventos:

1. Pulsa en el botón 'Registrarse', ¡Error! No se encuentra el origen de la referencia..
2. Rellena los campos del formulario de registro y pulsa en 'Aceptar', ¡Error! No se encuentra el origen de la referencia.

[\[Si los campos están correctamente rellenos\]](#)

3a. Accede al subcaso de uso *Acceder al catálogo completo*.

[\[Si no\]](#)

3b. Aparece un mensaje que indica que faltan datos por rellenar. Pulsa 'Ok' y vuelve al formulario de registro, ¡Error! No se encuentra el origen de la referencia.

Pos condiciones: Ninguna

Interfaz gráfica:

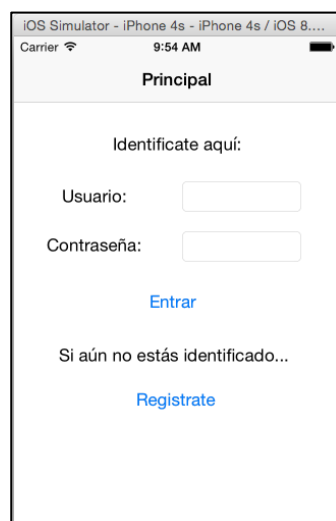


Imagen 41. Vista Inicio

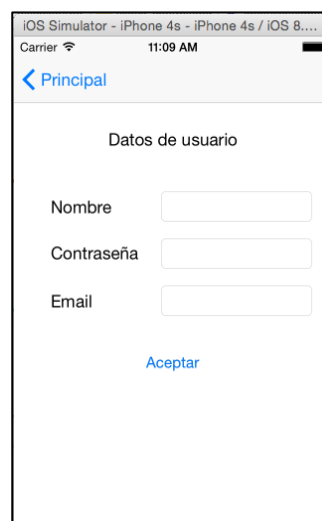


Imagen 42. Datos usuario

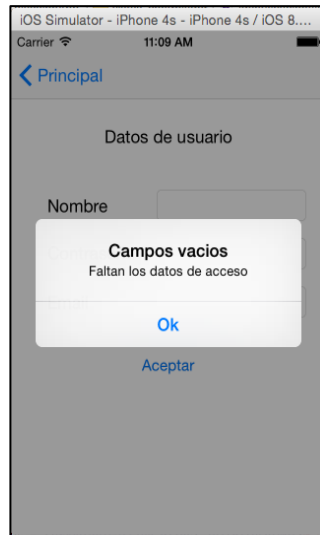
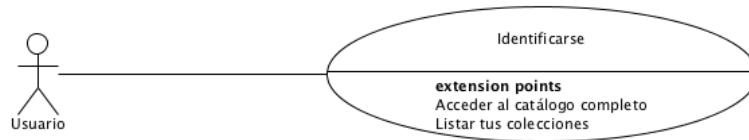


Imagen 43. Campos vacíos

Nombre: Identificarse

Descripción: El usuario se identifica y accede al juego.

Actores: Usuario

Precondiciones: Estar registrado y no estar ya identificado

Requisitos no funcionales: Ninguno

Flujo de eventos:

1. El usuario introduce su usuario y contraseña y pulsa 'Entrar', ¡Error! No se encuentra el rígen de la referencia.

[Si el usuario y la contraseña son correctos]

[Si ya participa en alguna colección]

2aa. Accede al caso de uso *Listar tus colecciones*.

[Si aún no participa en ninguna colección]

2ab. Accede al subcaso de uso *Acceder al catálogo completo*.

[Si no]

[Si los campos están vacíos]

2ba. Se muestra una alerta indicándolo, ¡Error! No se encuentra el origen de la eferencia.

[Si los datos no son válidos]

2bb. Se muestra una alerta que lo indica, ¡Error! No se encuentra el origen de la eferencia.

Pos condiciones: Ninguna

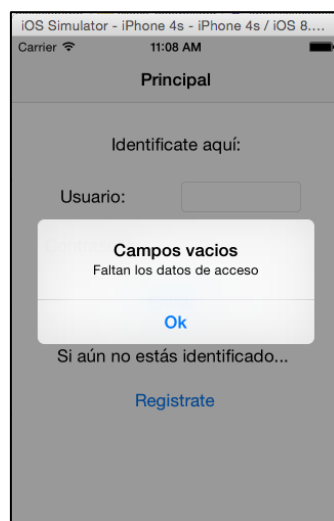
Interfaz gráfica:

Imagen 45. Campos vacíos

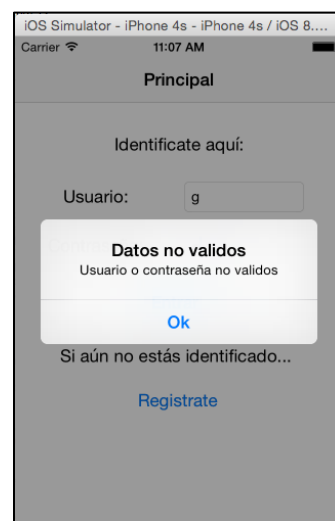
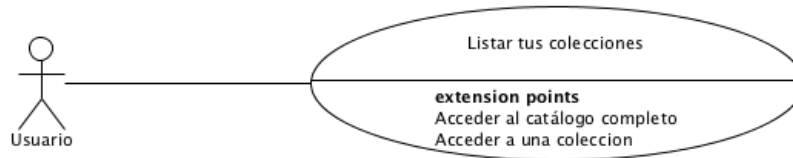


Imagen 44. Datos no válidos

Nombre: Listar tus colecciones

Descripción: El usuario accede al listado de colecciones en las que participa.

Actores: Usuario

Precondiciones: Estar identificado

Requisitos no funcionales: Ninguno

Flujo de eventos:

1. El usuario accede a la pantalla de sus colecciones
[Si lo desea]
- 2a. El usuario arrastra el dedo de derecha a izquierda sobre la colección y pulsa 'Eliminar' para eliminar una colección, ¡Error! No se encuentra el origen de la referencia.
[Si al eliminar se queda sin participar en ninguna colección]
- 3aa. Accede al subcaso de uso *Acceder al catálogo completo*.
[Si aún participa en alguna colección]
- 3ab. La colección se elimina y el usuario se mantiene en la misma ventana.
[Si pulsa sobre una colección]
- 2b. Accede al subcaso de uso *Acceder a una colección*.
[Si pulsa 'Mostrar otras colecciones']
- 2c. Accede al subcaso de uso *Acceder al catálogo completo*.

Pos condiciones: Ninguna

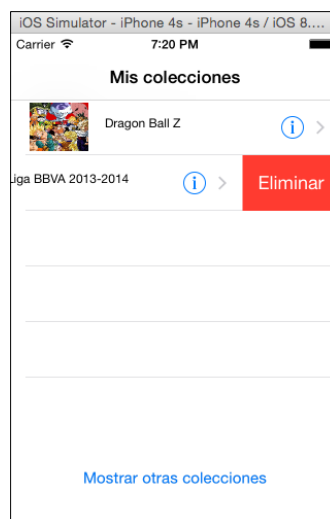
Interfaz gráfica:

Imagen 46. Mis colecciones

Nombre: Acceder al catálogo completo (subcaso de uso)

Acceder al catálogo completo

Descripción: Subcaso de uso que permite al usuario acceder al catálogo completo de colecciones existentes

Actores: Usuario

Precondiciones: Estar identificado

Requisitos no funcionales: Ninguno

Flujo de eventos:

1. Accede a un listado de colecciones disponibles en la que el usuario no participa,
2. Selecciona la colección en la que quiere participar y se muestra una nueva ventana con los detalles de la colección,
[Si pulsa 'Empezar esta colección']
- 3a. Se muestra una animación de un sobre saliendo de una caja. El usuario toca el sobre para abrirlo,
- 4a. Accede al subcaso de uso *Acceder a una colección*.
[Si pulsa 'Atrás']
- 3b. Accede a la pantalla de sus colecciones.

Pos condiciones: Ninguna

Interfaz gráfica:

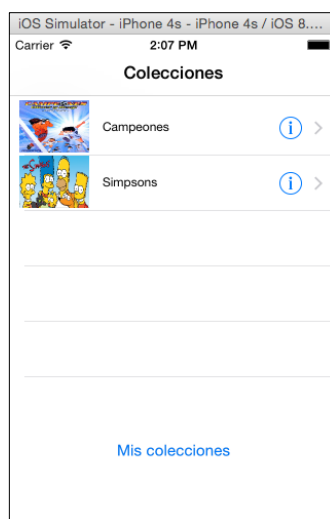


Imagen 47. Colecciones disponibles



Imagen 48. Empezar una colección



Imagen 49. Abrir sobre

Nombre: Afrontar un reto (*subcaso de uso*)

Afrontar un reto

Descripción: Subcaso de uso que permite al usuario enfrentarse contra la máquina mediante un cuestionario de preguntas con opciones.

Actores: Usuario

Precondiciones: Estar identificado.

Requisitos no funcionales: Ninguno

Flujo de eventos:

1. Pulsa en la pestaña 'Reto' se muestra una vista que ofrece la posibilidad de iniciar un reto aleatorio
[Si pulsa 'Iniciar reto aleatorio']
- 2a. Aparecerá uno de los dos juegos.
[Si es el juego de memoria]
- 3aa. Aparecerá un juego en el que tendrá que ir descubriendo las parejas antes de que se agoten los 15 movimientos permitidos.
- 4aa. Puede pulsar el botón 'Reiniciar' para empezar desde 0.
[Si supera el juego]
- 5aaa. Se muestra un mensaje indicándolo.
- 6aaa. Pulsa 'Sobre' para obtener un cromó al azar
[Si no lo supera]
- 5aab. Se muestra un mensaje indicándolo.
- 6aab. Pulsa 'ok' y si lo desea puede seguir intentándolo.
[Si es el juego de habilidad] *
- 3ab. Aparecerá un juego de habilidad en que el usuario controla una gota de agua y tiene que llegar hasta el punto amarillo sin que le toquen las llamas que se mueven de arriba abajo. Tiene que ir haciéndolo moviendo el dispositivo.
[Si toca el punto de salida]
- 4aba. Se muestra un mensaje indicándolo.
- 5aba. Pulsa 'Sobre' para obtener un cromó al azar.
[Si toca una de las llamas]
- 4abb. Se muestra un mensaje indicando que no ha superado el reto.
- 5abb. Pulsa 'ok' y si lo desea puede seguir intentándolo.
- 3a. Tras abrir el sobre el usuario vuelve a la ventana *de la colección* donde le aparecerán sus cromos, incluido el que acaba de conseguir.

Pos condiciones: Ninguna

Interfaz gráfica:



Imagen 51. Iniciar reto aleatorio



Imagen 50. Reto de memoria



Imagen 52. Reto de memoria superado

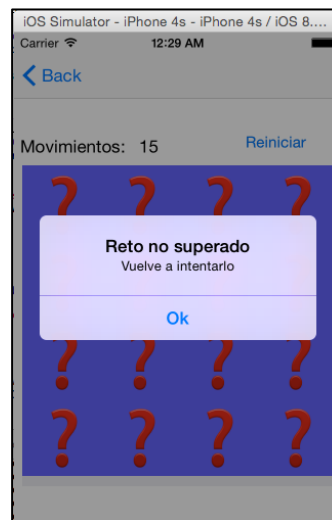


Imagen 53. Reto de memoria no superado



Imagen 54. Reto de habilidad superado



Imagen 55. Reto de habilidad no superado

Nombre: Provocar batalla (*subcaso de uso*)

Provocar batalla

Descripción: Subcaso de uso que permite al usuario provocar una batalla contra otro usuario.

Actores: Usuario

Precondiciones: Estar identificado y tener cromos.

Requisitos no funcionales: Ninguno

Flujo de eventos:

1. Cuando se carga la vista primero se comprueba si se tiene alguna solicitud de batalla
[Si se tiene]
 - 2a. Aparece una alerta que permite acceder al listado de las propuestas o dejarlo para más adelante.
 - 3a. Se muestra el botón 'Recibidas' que permite ver también las propuestas de batalla recibidas .
[Si pulsa 'Ver Propuestas']
 - 4aa. Se muestra el listado de propuestas donde por cada propuesta se indica quien es el usuario que la propone, la fecha, el tipo de batalla propuesta y la posibilidad de aceptar o rechazarla.
[Si pulsa 'Aceptar']
 - 5aaa. Se inicia el juego de memoria sin límite de movimientos.
 - 6aaa. El usuario va haciendo parejas hasta completar las 8 parejas.
[Si pulsa 'Abandonar']
 - 7aaaa. Le sale una pantalla que le indica que si abandona perderá automáticamente
[Si pulsa 'Rendirme']
 - 8aaaaa. Perderá automáticamente e irá a la ventana de la colección
[Si pulsa 'continuar la batalla']
 - 8aaaab. Continuará con el juego en el punto en el que estaba.
[Si completa el juego]
 - [Si ha ganado la batalla]
 - 7aaaba. Se muestra una alerta que se lo indica.
 - 8aaaba. Le aparece una ventana con los cromos de su oponente y el número de veces que tiene cada cromo repetido.
[Si la batalla es selectiva]
 - 9aaabaa. El usuario selecciona un cromo de su oponente.
[Si la batalla es aleatoria]
 - 9aaabab. El usuario pulsa un botón para recibir un cromo de su oponente al azar.
 - 10aaaba. El cromo es añadido a la colección del usuario.
 - 11aaaba. Se muestra la vista de la colección.
[Si ha perdido la batalla]
 - 7aaabb. Se muestra una alerta que se lo indica.

8aaabb. Se muestra la vista de la colección.

[Si aún no se sabe quien es el ganador]

7aaabc. Se muestra una alerta que se lo indica.

8aaabb. Se muestra la vista de la colección.

[Si pulsa 'Rechazar']

5aab. Se rechaza la propuesta y desaparece de la lista

[Si pulsa 'Dejarlo para luego']

4ab. Le aparecerá una lista con los usuarios que participan es esa colección.

5ab. Si tiene batallas en curso, propuestas realizadas y recibidas se le mostrará un botón donde puede acceder a cada una de ellas.

[Si pulsa sobre un usuario de la lista]

6ab. Le aparece una lista con los cromos que ese usuario tiene en su colección y el número de veces que los tiene. Le aparecerá también un selector para elegir el tipo de batalla y un botón 'Batalla' para hacer la propuesta.

7ab. El usuario selecciona el tipo de propuesta y pulsa sobre 'Batalla'

8ab. Se muestra la vista de la colección.

[Si tiene propuestas realizadas]

6ac. Pulsa sobre el botón 'Realizadas'.

7ac. Le aparece un listado de las propuestas de batalla realizadas

[Si aun esta sin respuesta por parte del oponente]

8aca. Se muestra un botón con la posibilidad de anular la batalla.

9aca. El usuario pulsa el botón y anula la batalla

10aca. La batalla desaparece del listado.

[Si el oponente ha aceptado]

8acb. Se muestra un botón con la posibilidad de empezar la batalla.

9acb. El usuario pulsa 'Empezar' y empieza la batalla.

[Si tiene batallas en curso]

6ad. Pulsa sobre el botón 'En curso'

7ad. Le aparece un listado de las batallas en curso. Son aquellas que están pendientes de ser cerradas por al menos uno de los dos usuarios.

[Si aún no hay ganador]

8ada. Se muestra una celda de fondo azul indicando que la batalla está aún en curso. Es decir, ha sido empezada pero, al menos uno de los dos usuarios no ha completado el juego.

[Si se ha ganado]

8adb. Se muestra una celda de fondo verde indicando que la batalla se ha ganado. También se muestran las puntuaciones.

[Si se ha obtenido ya el cromo]

9adba. Ocurre cuando el usuario termina el juego y ya se sabe que ha ganado. En este caso, el oponente ya había terminado de jugar y su puntuación ya estaba en la base de datos. Se muestra porque el oponente aún no ha visto que ha perdido la batalla.

10adba. Se muestra un mensaje que indica que ya obtuvo el cromo.

[Si aún no se ha obtenido el cromo]

9adbb. Ocurre cuando el usuario termina el juego y aún su oponente no ha

jugado. El oponente termina de jugar más tarde y pierde.

10adbb. Se muestra un mensaje que indica que obtenga el cromo

11adbb. El usuario pulsa sobre 'Cromo ' y obtiene un cromo en base al tipo de batalla que sea. La batalla se da por finalizada.

11adbb. Se accede a la vista colección.

[Si se ha perdido]

8adb. Se muestra una celda de fondo rojo indicando que la batalla se ha perdido. También se muestran las puntuaciones.

[Si aún no ha perdido ya el cromo]

9adba. Ocurre cuando el usuario termina el juego y ya se sabe que ha perdido el cromo. Aparece aún porque el oponente aún no ha visto que ha ganado. Desaparece cuando el oponente le quite el cromo al usuario.

10adbb. Se muestra un mensaje que indica que ya obtuvo el cromo.

[Si ya ha perdido el cromo]

9adbb. Ocurre cuando el usuario termina el juego y aún su oponente no ha jugado. El oponente termina de jugar más tarde y gana.

10adbb. Se muestra un mensaje que indica ha perdido un cromo.

11adbb. El usuario pulsa sobre 'Cerrar ' para indicar que ya lo ha visto.

12adbb. La batalla desaparece de la lista y se da por finalizada.

[Si tiene batallas Recibidas]

6ae. Los mismos pasos que se dan en el punto 4.aa

Pos condiciones: Ninguna

Interfaz gráfica:



Imagen 57. Propuestas de batalla

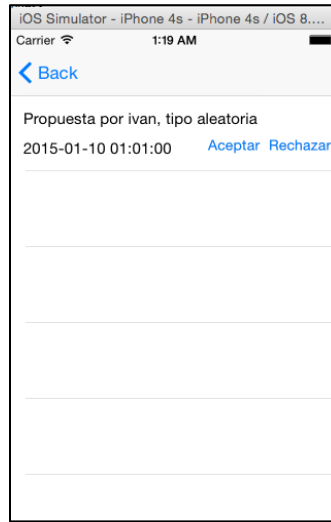


Imagen 58. Propuestas recibidas



Imagen 56. Abandonar



Imagen 59. Batalla ganada



Imagen 62. Batalla selectiva ganada

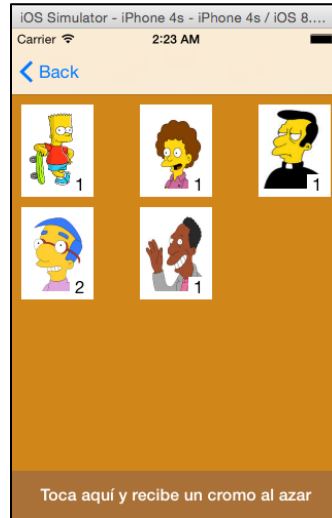


Imagen 63. Batalla aleatoria ganada



Imagen 60. Batalla perdida



Imagen 61. Batalla sin determinar aún

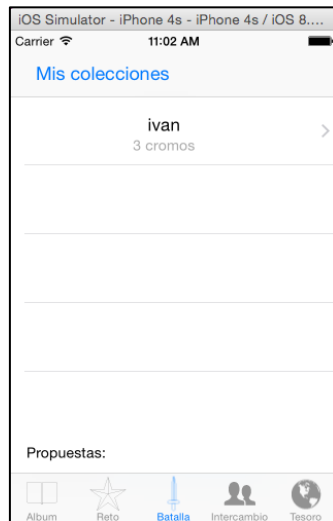


Imagen 66. Lista de usuarios para batalla

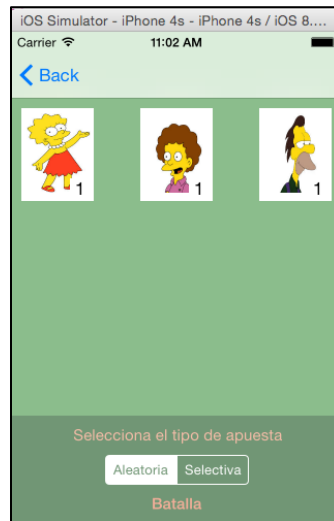


Imagen 65. Elegir tipo de batalla



Imagen 64. Anular batalla



Imagen 67. Empezar batalla



Imagen 69. Batallas ganadas, perdidas y en curso

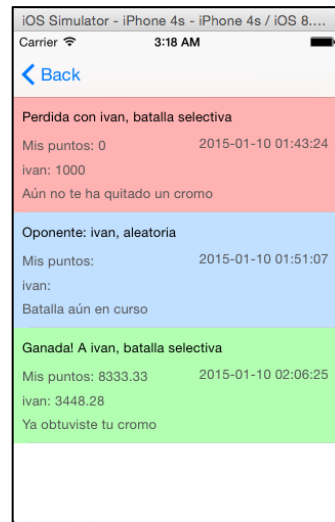



Imagen 68. Batallas ganadas y perdidas sin cerrar

Nombre: Intercambiar cromos (*subcaso de uso*)



Agregar cromo

Descripción: Subcaso de uso que permite al usuario intercambiar un cromo con otro.

Actores: Usuario

Precondiciones: Estar identificado y tener cromos

Requisitos no funcionales: Ninguno

Flujo de eventos:

1. Cuando se carga la vista primero se comprueba si se tiene alguna propuesta de intercambio.

[Si se tiene]

- 2a. Aparece una alerta que permite acceder al listado de las propuestas o dejarlo para más adelante.

- 3a. Se muestra el botón 'Recibidas' que permite ver también las propuestas de intercambio recibidas .

[Si pulsa 'Ver ofertas']

- 4aa. Se muestra el listado de propuestas donde por cada propuesta se indica quien es el usuario que la propone, la fecha, el cromo pretendido (izquierda), el cromo ofertado (derecha) y la posibilidad de aceptar o rechazar el intercambio.

[Si pulsa 'Aceptar']

- 5aaa. Se produce el intercambio de los cromos.

- 6aaa. Desaparece la propuesta y se muestra la ventana de la colección.

[Si pulsa 'Rechazar']

- 5aab. Se anula la propuesta y desaparece del listado.

- 6aab. El cromo del oponente se desbloquea.

- 7aab. El usuario permanece en la misma ventana de propuestas recibidas.

[Si pulsa 'Dejarlo para luego']

- 4ab. Le aparecerá una lista con los usuarios que participan es esa colección.

- 5ab. Si tiene propuestas realizadas y recibidas se le mostrará un botón donde puede acceder a cada una de ellas.

[Si pulsa sobre un usuario de la lista]

- 6ab. Le aparece una lista con los cromos que ese usuario tiene en su colección, el número de veces que los tiene y su estado de bloqueo. Le aparecerá también un mensaje que le indica que seleccione el cromo que le interesa.

- 7ab. El usuario selecciona un cromo que le interesa y no este bloqueado. Le aparece una nueva vista con los cromos que tiene el usuario, incluyendo el número de veces y su estado de bloqueo.

- 8ab. El usuario selecciona un cromo que no este bloqueado. Aparece un mensaje que le informa de que la propuesta de intercambio ha sido realizada.

- 9ab. Pulsa sobre 'Menú mi colección' y accede a su colección.

[Si tiene propuestas realizadas]

- 6ac. Pulsa sobre el botón 'Realizadas'.

7ac. Le aparece un listado de las propuestas de intercambio realizadas. Significa que el oponente aún ni ha aceptado ni rechazado la propuesta.

8ac. El usuario pulsa el 'Anular'. Desaparece la propuesta y se desbloquea el cromo que el usuario ofreció como intercambio.

[Si tiene propuestas recibidas]

6ad. Los mismos pasos que a partir del punto 4aa.

Pos condiciones: Ninguna

Interfaz gráfica:



Imagen 72. Propuestas de intercambio recibidas

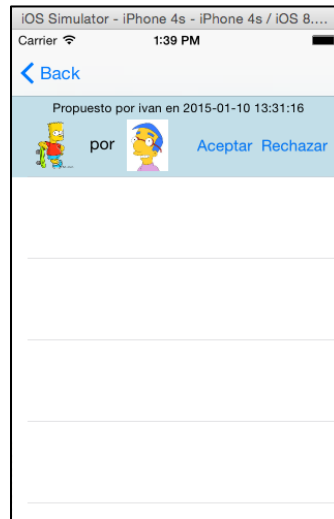


Imagen 70. Lista de propuestas de intercambio



Imagen 71. Seleccionar cromo

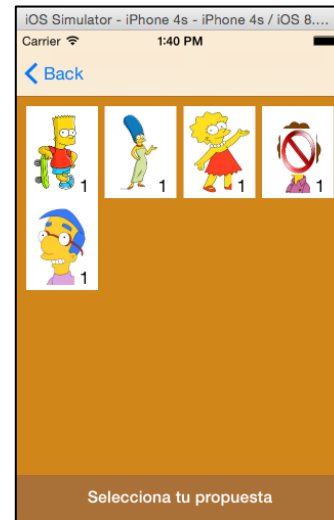


Imagen 73. Proponer cromo



Imagen 74. Propuesta realizada

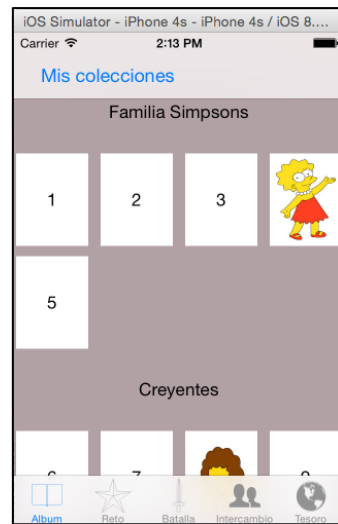


Imagen 75. Menu mi colección

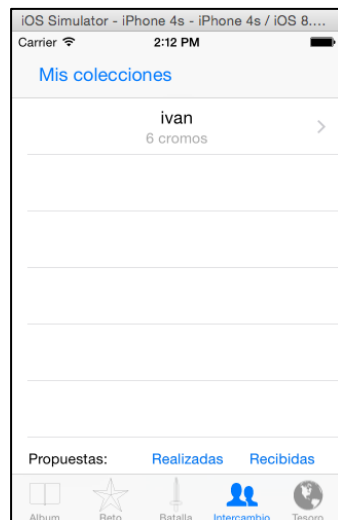


Imagen 76. Pestaña intercambio



Imagen 77. Anular propuesta de intercambio

Nombre: Buscar tesoro (*subcaso de uso*)

Buscar tesoro

Descripción: Subcaso de uso que permite al usuario buscar un cromó especial mediante geo localización.

Actores: Administrador

Precondiciones: Estar identificado y haber recibido una notificación de búsqueda.

Requisitos no funcionales: Ninguno

Flujo de eventos:

1. Aparece el mapa con un selector azul que representa la posición actual y otro rojo que representa una ubicación del cromó relativamente cerca (2 km aproximadamente).
2. El usuario se acerca a la posición del cromó
[\[Si esta a menos de 150 metros\]](#)
 - 2a. Aparece una alerta que le indica que ha encontrado el cromó.
 - 3a. Pulsa en sobre para abrirlo y accede al menú de su colección.
[\[Si no hay conexión a internet\]](#)
 - 2b. Aparece una alerta que le indica que no se puede obtener la localización

Pos condiciones: Ninguna

Interfaz gráfica:

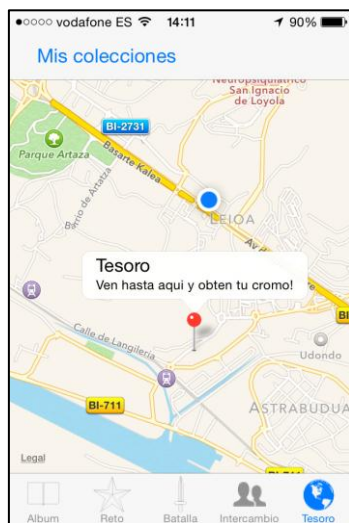


Imagen 78. Tesoro



Imagen 79. Cromo encontrado



Imagen 80. Error al obtener la ubicación

ANEXO II

Diagramas de secuencia

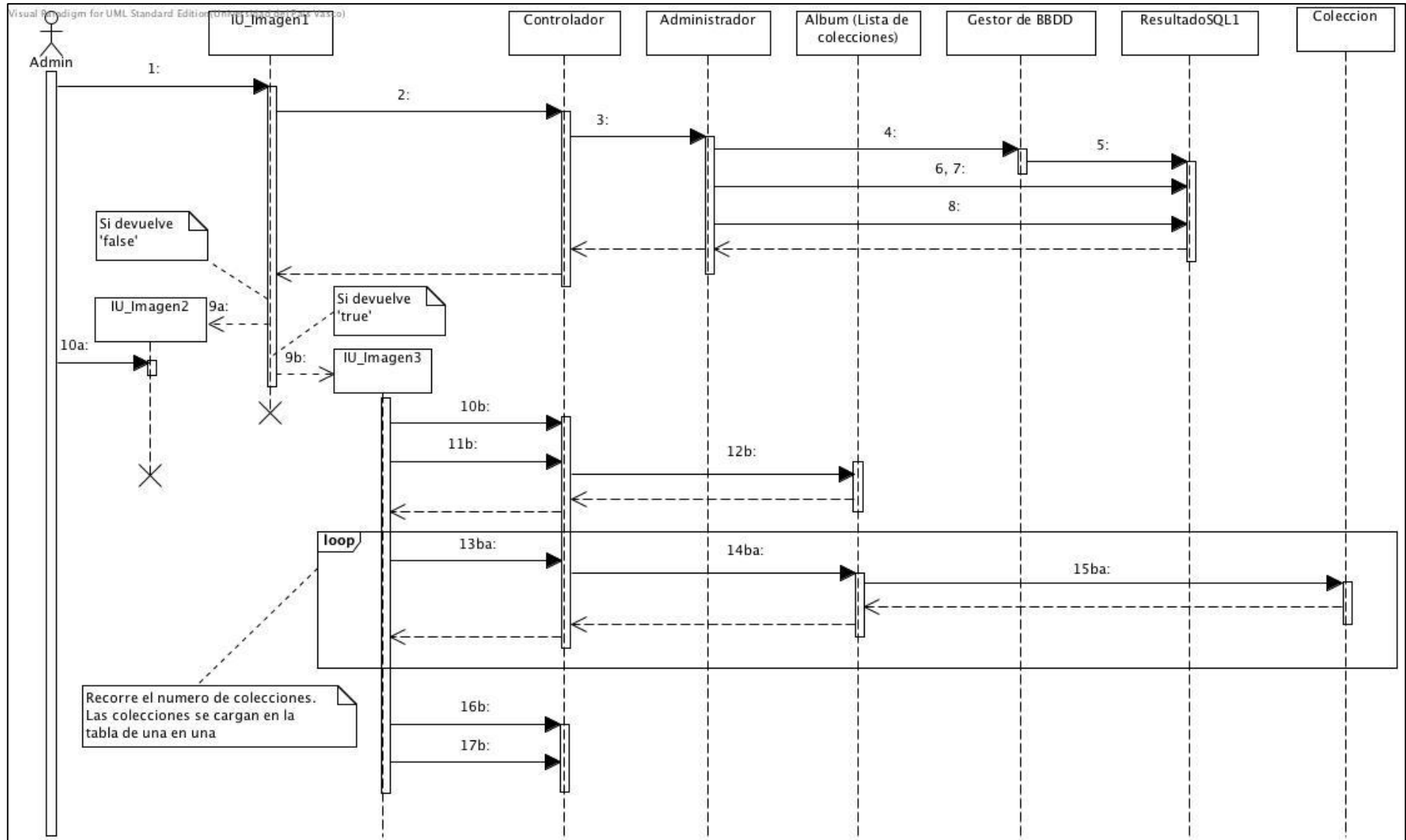
Administrador

Identificarse

1. El administrador introduce 'Usuario' y 'Contraseña' y pulsa 'Entrar'
2. identificarse(usuario, pContraseña) : bool
3. identificarse(pUsuario, pContraseña) : bool
4. execSQL1(sql1) :

```
SELECT usuario, password
FROM administrador
```

5. new()
6. getUsuario(): String
7. getContraseña() : String
8. close
 - [Si es 'false']
 - 9a. new()
 - 10a. Pulsa Aceptar
 - [Si es 'true']
 - 9b. new
 - 10b. cargarColecciones() : void *// [de BBDD a Clases]*
 - 11b. contarColecciones(): int
 - 12b. cuantasColecciones(): int
 - [Para cada una de las filas] *// [de Clases a Tabla]*
 - 13ba. cargarColeccionEnTabla(pFila) : ArrayList <Object>
 - 14ba. getDatosColeccion(pPos) : ArrayList <Object>
 - 15ba. getDatosColeccion() : ArrayList <Object>
 - 16b. cargarFamilias() : void *// [de BBDD a Clases]*
 - 17b. cargarTodosLosCromos() : void *// cargarTodosLosCromos() – Controlador [de BBDD a Clases]*



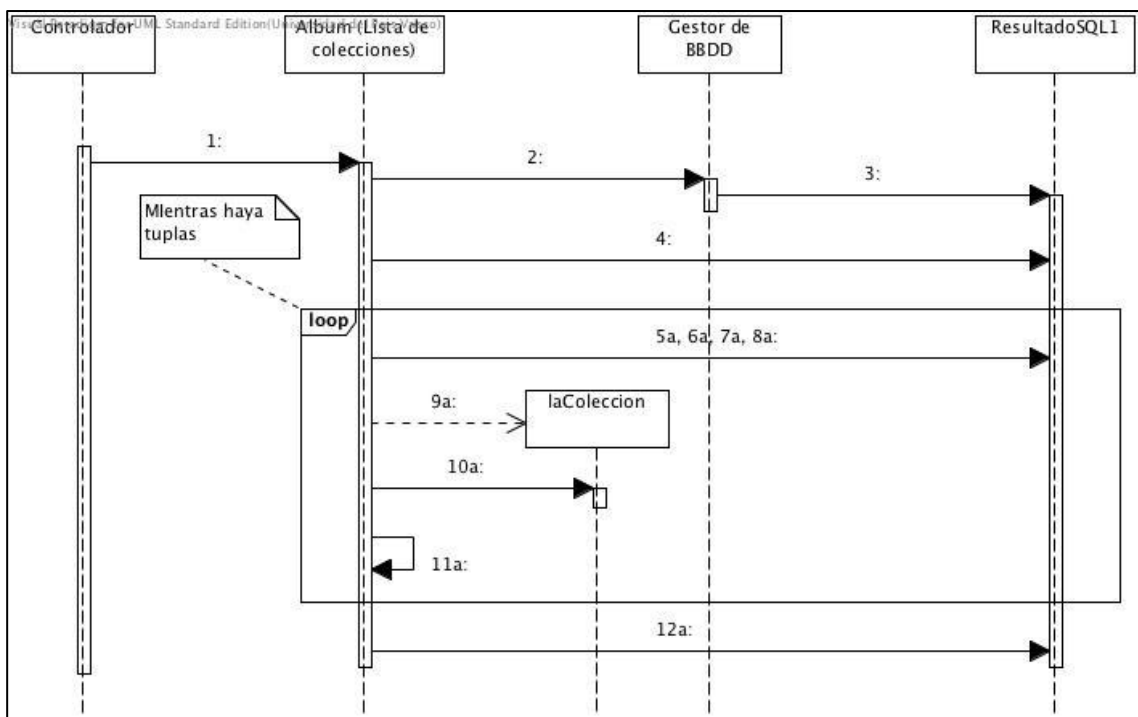
cargarColecciones – Controlador

// Se cargan las colecciones de la base de datos a la clase Album (Lista de colecciones).

1. cargarColecciones(): void
2. execSQL1(sql1)

```
SEELCT codColeccion, nombre, descripcion, imagen
FROM coleccion
```

3. new()
4. next()
 - [Mientras haya tuplas]
 - 5a. getCodColeccion(): int
 - 6a. getNombre(): String
 - 7a. getDescripcion(): String
 - 8a. getImagen(): byte[]
 - 9a. new(pCodColeccion, pNombre, pDescripcion, pImagen) : Coleccion
 - 10a. cargarFamilias() : void
 - 11a. agregarColeccion(pLaColeccion) : void // Únicamente se agrega si no existe ya, se controla desde dentro del método.
 - 12a. close



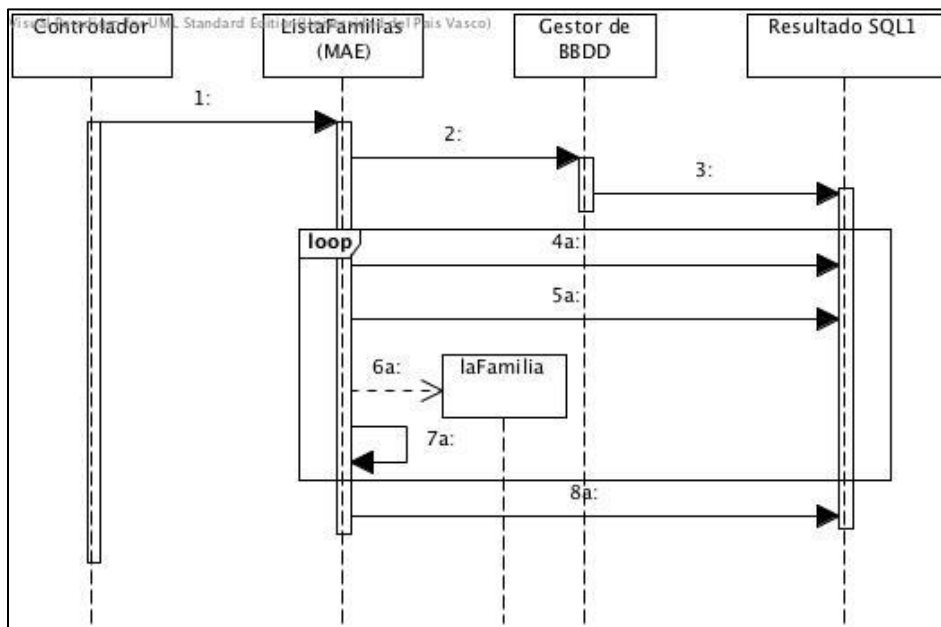
cargarFamilias – Controlador

// Se cargan las familias de la base de datos a la clase ListaFamilias.

1. cargarFamilias() : void
2. execSQL1(sql1)

```
SELECT codFamilia, descripcion
FROM familia
```

3. new()
4. next()
 - [Mientras haya tuplas]
 - 5a. getDatosFamilia()
 - 6a. new(pCodFamilia, pDescripcion) : Familia
 - 7a. agregarFamilia() : void // Únicamente se agrega si no existe ya, se controla desde dentro del método.
 - 8a. close



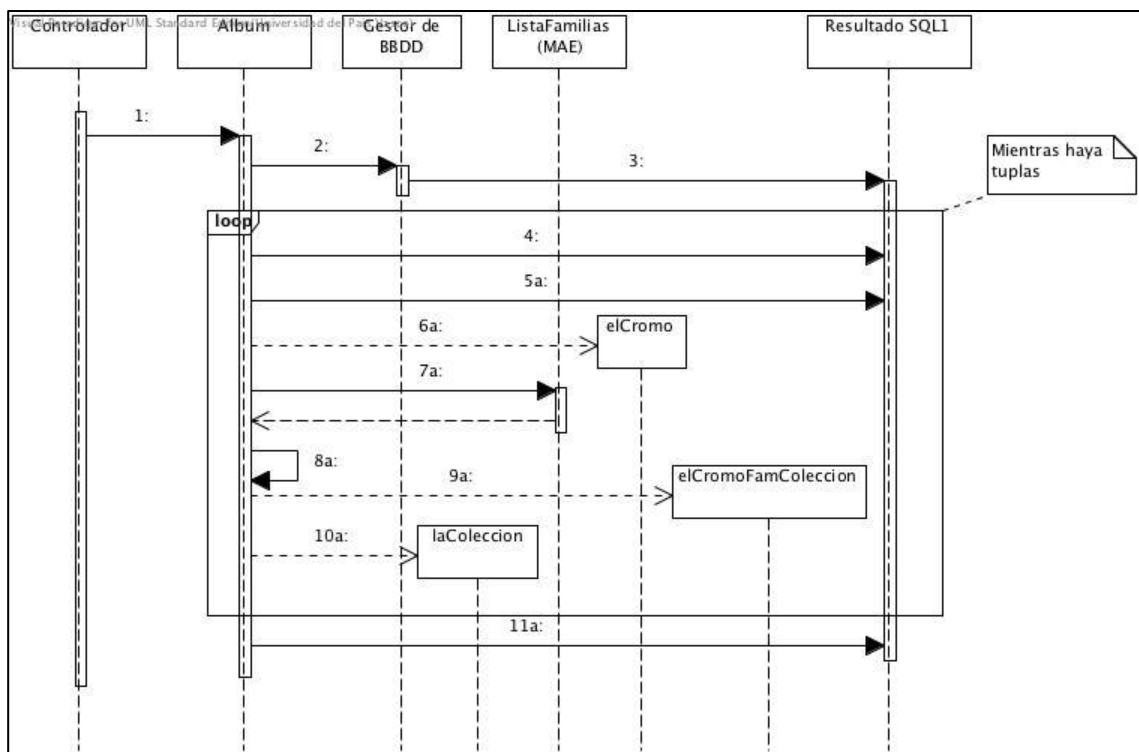
cargarTodosLosCromos() – Controlador

// Se cargan los cromos desde la base de datos a la lista de cromos de la clase colección correspondiente.

1. cargarTodosLosCromos() :void
2. execSQL1(sql1)

```
SELECT  cfc.numero, cfc.codCromo, cfc.codFamilia, cfc.codColeccion, cfc.imagen,
        c.titulo, c.descripcion
FROM    cromos_fam_coleccion cfc, cromos c
WHERE   cfc.codCromo = c.codCromo
```

3. new()
4. next()
 - [Mientras haya tuplas]
 - 5a. getDatosCromo()
 - 6a. new(pCodCromo, pTitulo, pDescripcion) : Cromo
 - 7a. buscarFamilia(pCodFamilia) : Familia
 - 8a. buscarColeccion(pCodColeccion) : Coleccion
 - 9a. new (pNumero, pCromo, pColeccion, pFamilia, pImagen) : Cromo_Fam_Coleccion
 - 10a. agregarCromo(pElCromoFamColeccion) : void
 - 11a. close



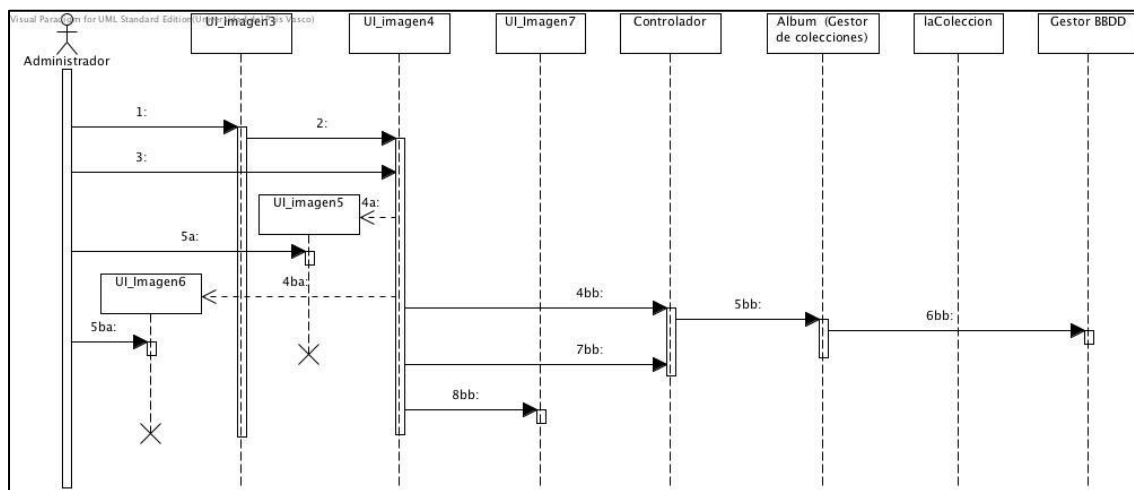
Agregar colección

// Se crea la nueva colección en la base de datos y se carga en la clase Album

1. El administrador pulsa 'Agregar colección'
2. new()
3. Introduce los datos necesarios y pulsa 'Crear colección'
- [Si los capos no están correctamente rellenos]
- 4a. new()
- 5a. Pulsa 'Aceptar'
- [Si los campos están correctamente rellenos]
- [Si el nombre de la colección ya existe]
- 4ba. new()
- 5ba. Pulsa 'Aceptar'
- [Si el nombre de la colección no existe]
- // En este bloque se crea la colección en la base de datos.
- 4bb. crearColeccion() : void
- 5bb. crearColeccion() : void
- 6bb. execSQL1(sql1)

```
INSERT INTO coleccion (nombre, descripcion, imagen)
VALUES nombreColeccion, descripcion, imagen
```

- 7bb. cargarColecciones();
- 8bb. new(pCodColeccion)



Modificar colección

// Se modifica la colección en la base de datos y se carga en la clase Album

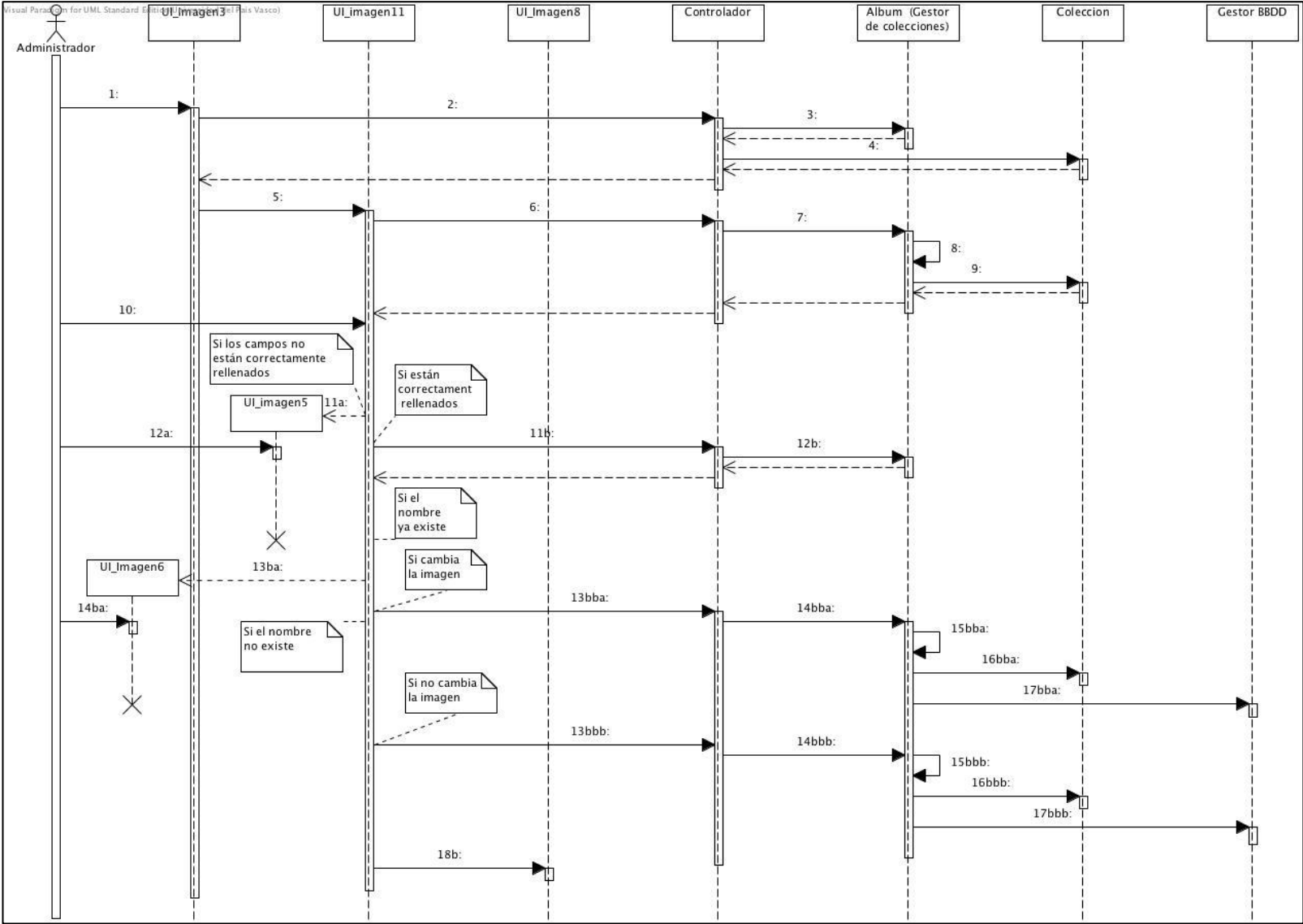
1. El administrador selecciona una colección y pulsa 'Modificar colección'
2. getCodColeccion(pNombreColeccion) : int
3. buscarColeccion(pNombreColeccion) : int
4. getCod(): int
5. new(pCodColeccion)
6. getDatosColeccion(pCodColeccion) : ArrayList<Object> //cargarDatosColeccion
7. getDatosColeccionCod(pCodColeccion) : ArrayList<Object>
8. buscarColeccion(pCodColeccion) : Coleccion
9. getDatosColeccion() : ArrayList<Object>
10. Cambia los datos que desee y pulsa 'Modificar colección'
 - [Si los campos no están correctamente rellenos]
 - 11a. new()
 - 12a. pulsa 'Aceptar'
 - [Si los campos están correctamente rellenos]
 - 11b. existeColeccion(pNombre) : boolean
 - 12b. existe(pNombre) : boolean
 - [Si el nombre ya existe]
 - 13ba. new()
 - 14ba. pulsa 'Aceptar'
 - [Si el nombre escogido no existe]
 - [Si ha cambiado la imagen de la colección]
 - 13bba. modificarColeccion(pCodColeccion, pNombre, pDescripcion, pImagen) : void
 - 14bba. modificarColeccion(pCodColeccion, pNombre, pDescripcion, pImagen) : void
 - 15bba. buscarColeccion(pCodColeccion) : Colección
 - 16bba. modificarColeccion(pNombre, pDescripcion, pImagen) : void // modifica en la clase
 - 17bba. exectSQL1(sql1) // Se modifica en la BBDD

```
UPDATE coleccion
SET nombre = pNombre, descripcion = pDescripcion, imagen = pImagen
WHERE codColeccion = pCodColeccion
```

- [Si no ha cambiado la imagen de la colección]
- 13bbb. modificarColeccionSinImagen(pCodColeccion, pNombre, pDescripcion): void
- 14bbb. modificarColeccion(pCodColeccion, pNombre, pDescripcion,) : void
- 15bbb. buscarColeccion(pCodColeccion) : Coleccion
- 16bbb. modificarColeccion(pNombre, pDescripcion) : void // Se modifica en la clase
- 17bbb. exectSQL2(sql2) // Se modifica en la BBDD

```
UPDATE coleccion
SET nombre = pNombre, descripcion = pDescripcion
WHERE codColeccion = pCodColeccion
```

- 18bb. new()



Eliminar colección

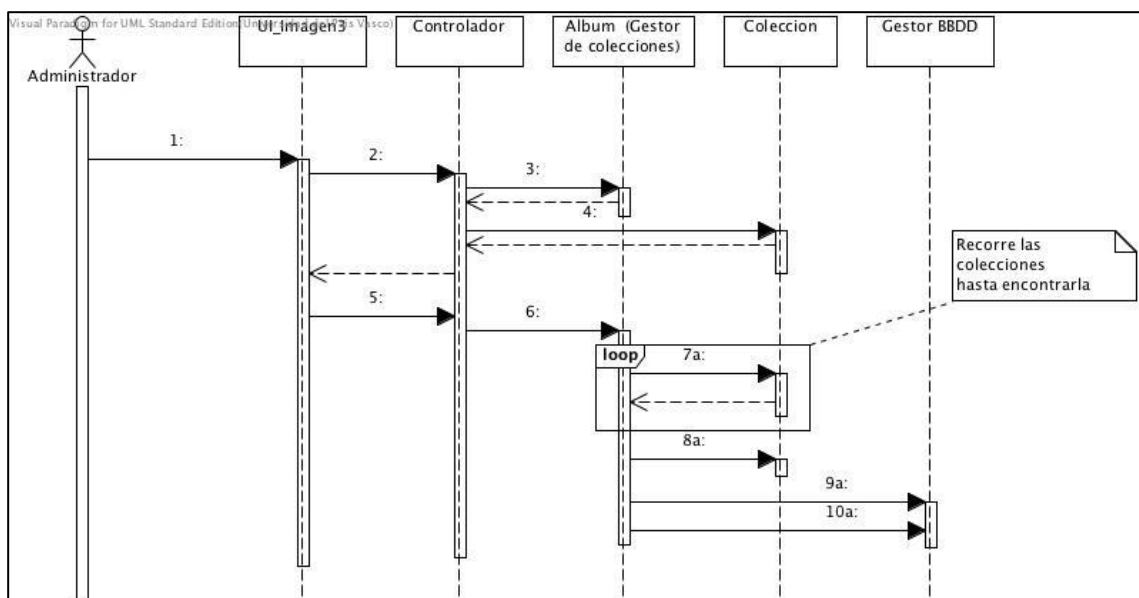
// Se elimina colección de la base de datos y de la clase Album

1. El administrador selecciona una colección y pulsa 'Eliminar colección'
2. getCodColeccion(pNombreColeccion)
3. buscarColeccion(pNombreColeccion) : int
4. getCod(): int
5. eliminarColeccion(pCodColeccion) : void
6. eliminarColeccion(pCodColeccion) : void // Se elimina en la clase
[Hasta que encuentre la colección]
- 7a. getCod() : int
- 8a. eliminarCromos() : void
- 9a. execSQL1(sql1) // Se eliminan primero los cromos de esa colección en la BBDD.
(integridad referencial)

```
DELETE FROM cromo_fam_coleccion
WHERE codColeccion = pCodColeccion
```

- 10a. execSQL2(sql2) // ahora se elimina la colección de la BBDD

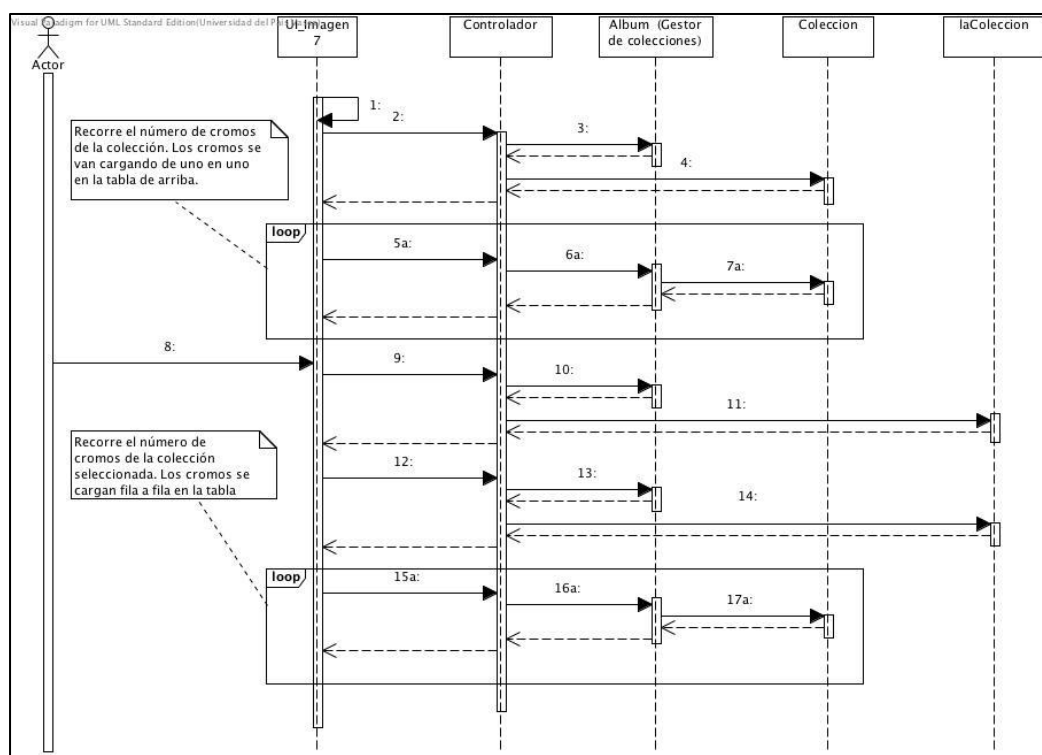
```
DELETE FROM coleccion
WHERE codColeccion = pCodColeccion
```



Cromos (Gestionar una colección)

// En la tabla de arriba se cargan los cromos de la colección que se esta gestionando. En la tabla de abajo los cromos de la colección que se seleccione en el comboBox.

1. cargarColeccionesEnComboBox() : void
 2. contarCromos (pCodColeccion) : int
 3. buscarColeccion(pCodColeccion) : Coleccion
 4. contarCromos() : int
[para cada fila de la tabla] // El número de cromos marca en número de filas de la tabla.
 - 5a. cargarCromoEnTabla(pFila, pCodColeccion) : ArrayList <Object>
 - 6a. getDatosCromoPosicion(pPos, pCodColeccion) : ArrayList <Object>
 - 7a. getDatosCromoPosicion(pPos) : ArrayList <Object>
- // Para cargar la tabla de abajo.
8. El administrador selecciona una colección del comboBox.
 9. getCodColeccion(pNombreColeccion) : int
 10. buscarColeccion(pNombreColeccion): Coleccion
 11. getCodColeccion() : int
 12. contarCromos (pCodColeccion) : int
 13. buscarColeccion(pCodColeccion): Colección
 14. contarCromos() : int
[para cada fila de la tabla] // El número de cromos marca en número de filas de la tabla.
 - 15a. cargarCromoEnTabla(pFila, pCodColeccion) : ArrayList <Object>
 - 16a. getDatosCromoPosicion(pPos, pCodColeccion) : ArrayList <Object>
 - 17a. getDatosCromoPosicion(pPos) : ArrayList <Object>



Agregar cromo – Cromos (Gestionar una colección)

// Permite agregar un cromo nuevo a la colección que se está gestionando (tabla de arriba).

1. El administrador pulsa 'Agregar'.
2. new(pCodColeccion)
3. El administrador rellena los campos y pulsa 'Agregar'
4. camposRequeridosValidos() : boolean
[Si devuelve 'false']
- 5a. new()
- 6a. pulsa 'Aceptar'
[Si devuelve 'true']
- 5b. insertarCromo(pCodColeccion, pNombreFamilia, pNumero, pTitulo, pDescripcion, pImagen) : void
- 6b. insertarCromo(pCodColeccion, pNombreFamilia, pNumero, pTitulo, pDescripcion, pImagen) : void
- 7b. buscarColeccion(pCodColeccion) : Colección
- 8b. getCodFamilia(pNombreFamilia) : int
- 9b. execSQL1(sql1)

```
INSERT INTO cromo(titulo, descripcion)
VALUES titulo, descripcion
```

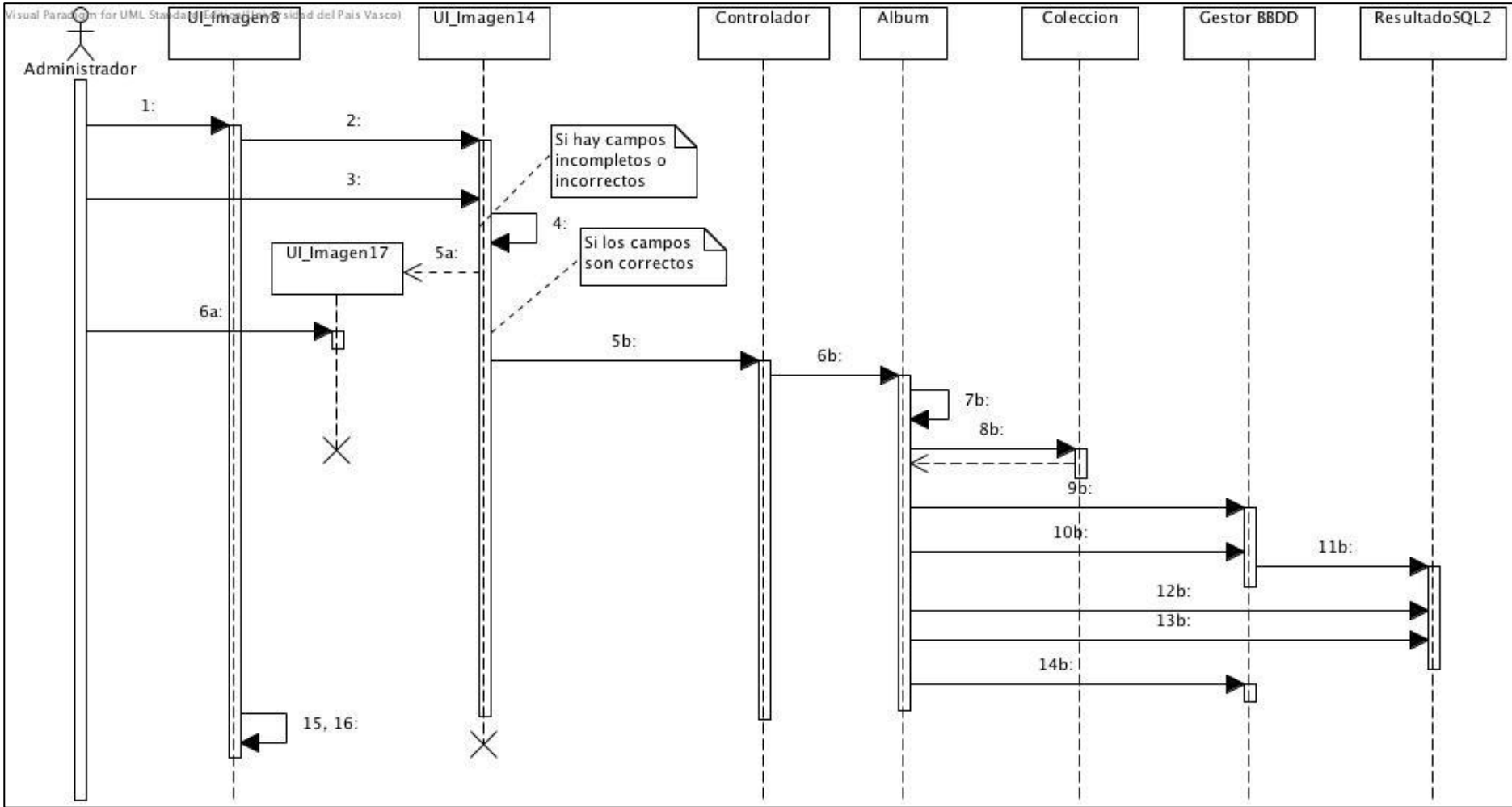
- 10b. execSQL2(sql2) // Se obtiene el codCromo del cromo que se acaba de insertar.

```
SELECT codCromo
FROM cromo
ORDER BY codCromo DESC
```

- 11b. new()
- 12b. getCodCromo() : int
- 13b. close
- 14b. execSQL3(sql3)

```
INSERT INTO cromo_fam_coleccion (numero, codCromo, codFamilia, codColeccion, imagen)
VALUES numero, codCromo, codFamilia, codColeccion, imagen
```

15. cargarTodosLosCromos() : void // Cargar todos los cromos
16. Se cargan las tablas de la vista cromos con los cambios // Cromos (Gestionar una colección)



Modificar cromos – Cromos (Gestionar una colección)

// Permite modificar un cromos de la colección que se está gestionando (tabla de arriba).

1. El administrador seleccionar un cromos y pulsa 'Modificar'.
2. new(pCodColeccion, pNumero, pFamilia)
3. getDatosCromo(pCodColeccion, pNumero) : ArrayList<Object>
4. getDatosCromo(pCodColeccion, pNumero) : ArrayList<Object>
5. buscarColeccion(pCodColeccion) : Coleccion
6. buscarCromo(pNumero) : Cromo_Fam_Coleccion
7. getDatosCromo() : ArrayList<Object>
8. El administrador cambia los campos que desee y pulsa 'Modificar'
9. camposRequeridosValidos() : boolean
 - [Si devuelve 'false']
 - 10a. new()
 - 11a. pulsa 'Aceptar'
 - [Si devuelve 'true']
 - 10b. buscarCodCromo(pCodColeccion, pNumero) : int
 - 11b. buscarColeccion(pCodColeccion) : Coleccion
 - 12b. buscarCromo(pNumero) : Cromo
 - 13b. getCodCromo() : int
 - [Si ha modificado la imagen]
 - 14ba. modificarCromo(pCodColeccion, pCodCromo, pNombreFamilia, pNumero, pTitulo, pDescripcion) : void
 - 15ba. modificarCromo(pCodColeccion, pCodCromo, pNombreFamilia, pNumero, pTitulo, pDescripcion, pImagen) : void
 - 16ba. buscarColeccion(pCodColeccion) : Colección
 - 17ba. getCodFamilia(pNombreFamilia) : int
 - 18ba. modificarCromo(pCodCromo, pCodFamilia, pTitulo, pDescripcion, pNumero, pImagen) : void
 - 19ba. getCodCromo() : int
 - [Si coincide]
 - 20baa. modificar(pTitulo, pDescripcion) : void
 - 21baa. modificar(pTitulo, pDescripcion) : void
 - 22baa. modificar(pNumero, pCodFamilia, pImagen) : void
 - 23ba. execSQL1(sql1)

```
UPDATE cromos
SET titulo = titulo, descripcion = pDescripcion
WHERE codCromo = pCodCromo
```

24ba. execSQL2(sql2)

```
UPDATE cromos_fam_coleccion
SET numero = pNumero, codFamilia = pCodFamilia, imagen = pImagen
WHERE codCromo = pCodCromo AND codColeccion = pCodColeccion
```

[Si no ha modificado la imagen]

14bb. modificarCromoSinImagen(pCodColeccion, pCodCromo, pNombreFamilia, pNumero, pTitulo, pDescripcion, pImagen) : void

15bb. modificarCromoSinImagen(pCodColeccion, pCodCromo, pNombreFamilia, pNumero, pTitulo, pDescripcion) : void

16bb. buscarColeccion(pCodColeccion) : Colección

17bb. getCodFamilia(pNombreFamilia) : int

18bb. modificarCromoSinImagen(pCodCromo, pCodFamilia, pTitulo, pDescripcion, pNumero) : void

19bb. getCodCromo() : int

[Si coincide]

20bba. modificar(pTitulo, pDescripcion) : void

21bba. modificar(pTitulo, pDescripcion) : void

22bba. modificar(pNumero, pCodFamilia, pImagen) : void

23bb. execSQL1(sql1)

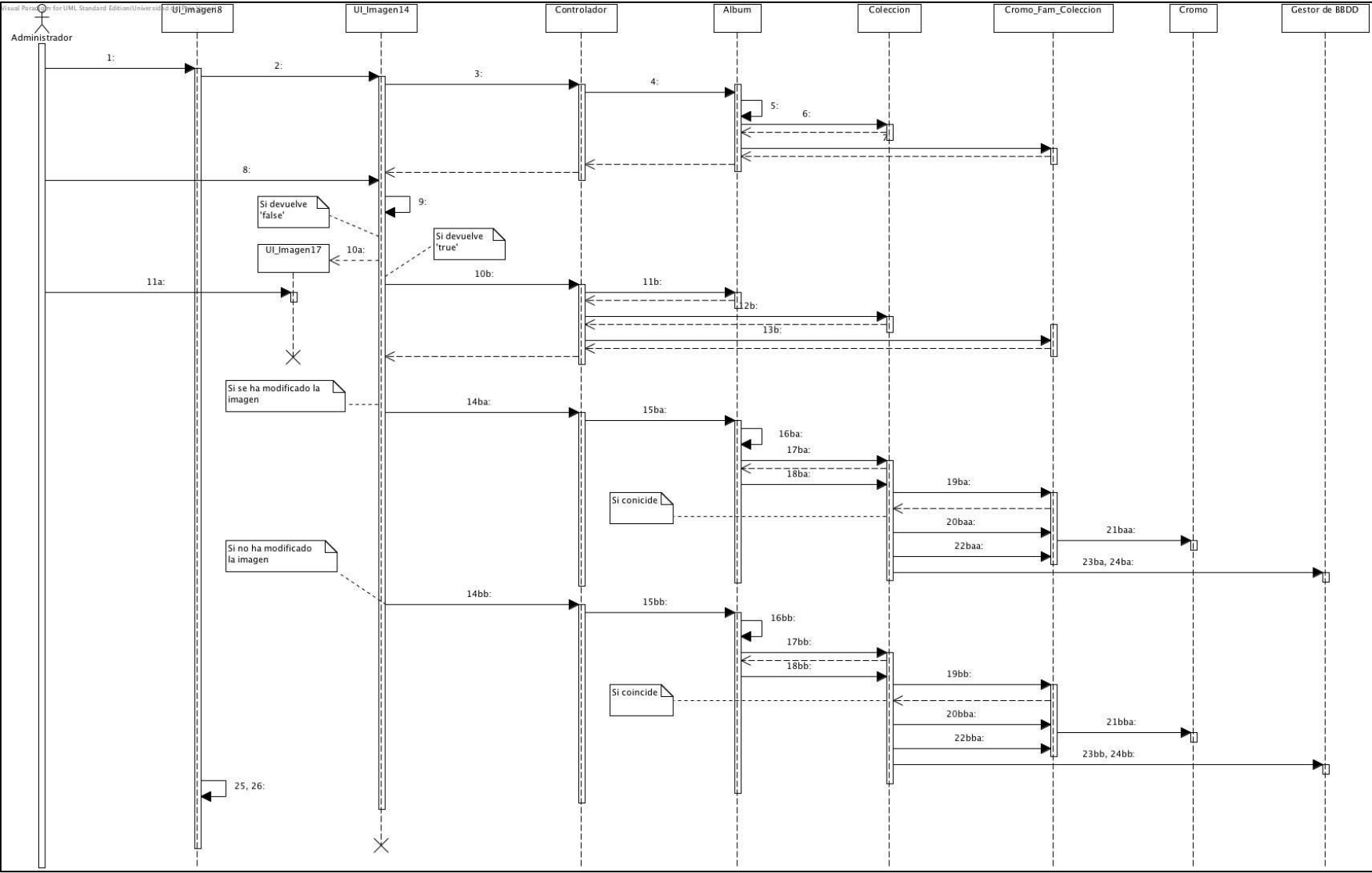
```
UPDATE cromos
SET titulo = pTitulo, descripcion = pDescripcion
WHERE codCromo = pCodCromo
```

24bb. execSQL2(sql2)

```
UPDATE cromos_fam_coleccion
SET numero = pNumero, codFamilia = pCodFamilia, imagen = pImagen
WHERE codCromo = pCodCromo AND codColeccion = pCodColeccion
```

25. cargarTodosLosCromos() : void // Cargar todos los cromos

26. Se cargan las tablas de la vista Cromos con todos los cambios realizados // Cromos (Gestionar una coleccion)



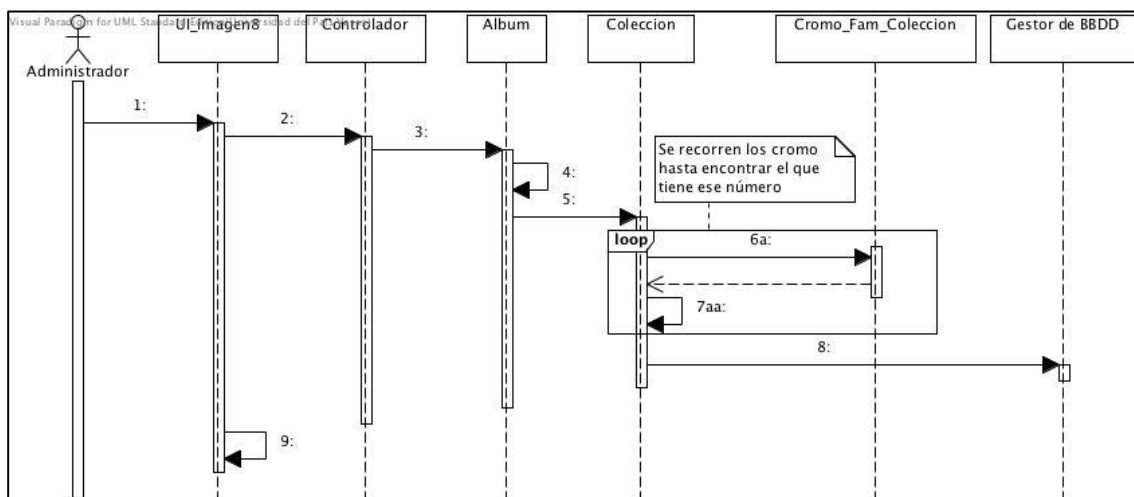
Eliminar cromo – Cromos (Gestionar una colección)

// Permite eliminar un cromo de la colección que se está gestionando (tabla de arriba).

1. El administrador seleccionar un cromo y pulsa 'Eliminar.'
2. eliminarCromo(pNumero, pCodColeccion) : void
3. eliminarCromo(pNumero, pCodColeccion) :void
4. buscarColeccion(pCodColeccion) : Colección
5. eliminarCromo(pNumero) : void
[Se recorren los cromos de la colección]
- 6a. getNumero() : int
[Si coincide]
- 7aa. Se elimina el cromo
8. execSQL(sql1)

```
DELETE FROM cromo_fam_coleccion
WHERE codColeccion = pCodColeccion
AND numero = pNumero
```

9. Se cargan las tablas de la vista Cromos con todos los cambios realizados // Cromos (Gestionar una colección)



Traer cromos – Cromos (Gestionar una colección)

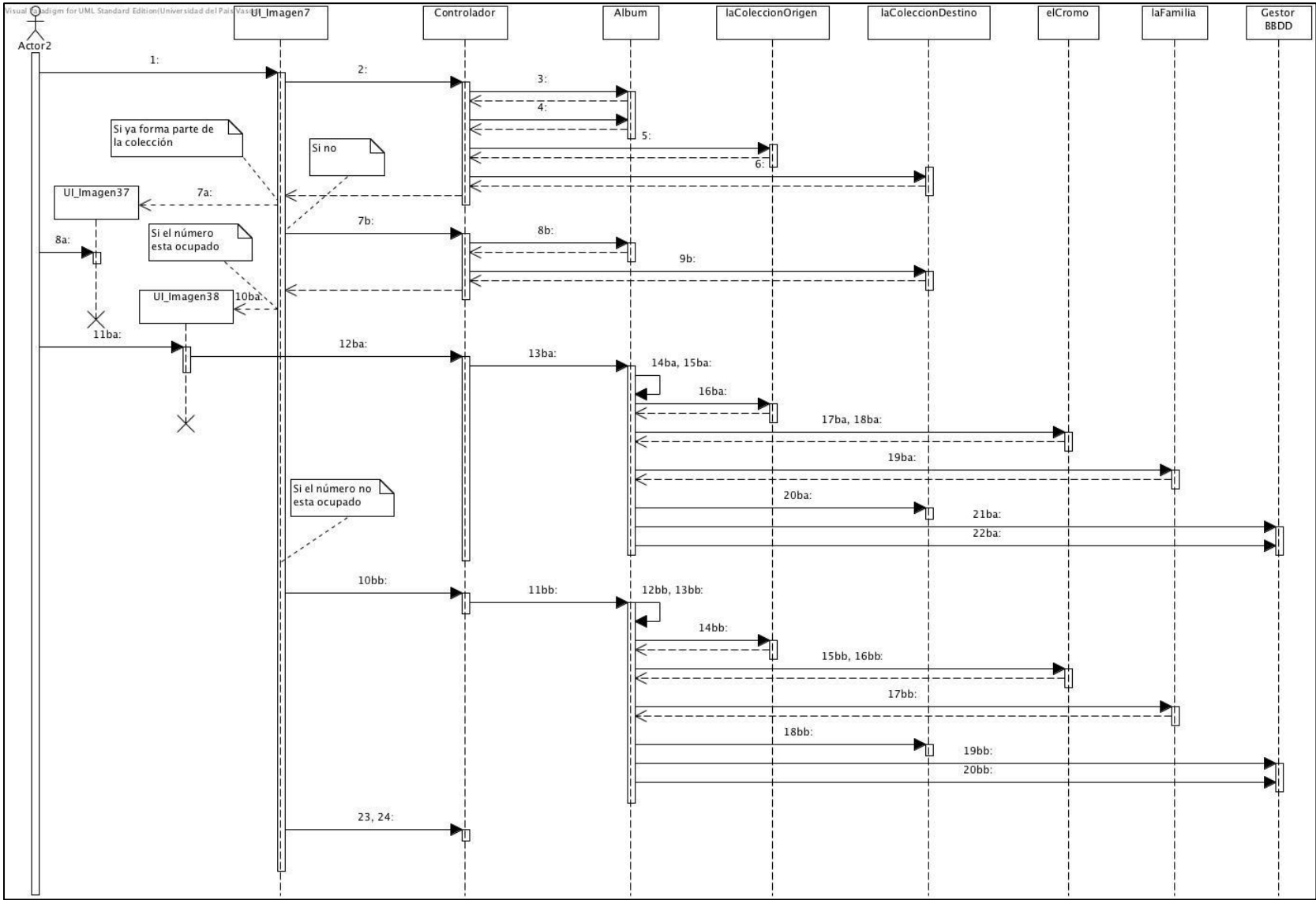
// Permite agregar un cromos de una colección existente (tabla de abajo) a la colección que se está gestionando (tabla de arriba).

1. El administrador selecciona un cromos de la tabla de abajo y pulsa .
2. existeCromo(pNumero, pNombreColeccionOrigen, pCodColeccionDestino) : boolean
3. buscarColeccion(pNombreColeccionOrigen) : Coleccion
4. buscarColeccion(pCodColeccionDestino) : Coleccion
5. buscarCromo(pNumero) : Cromo
6. existeCromo(pElCromo) : boolean
[Si ya formaba parte de la colección]
- 7a. new()
- 8a. pulsa 'Aceptar'
- [Si no]
- 7b. comprobarNumeroCromo(pNumero, pCodColeccionDestino) : boolean
- 8b. buscarColeccion(pCodColeccionDestino) : Coleccion
- 9b. existeNumero(pNumero) : boolean
[Si el número está ocupado en la colección destino]
- 10ba. new(pNumeroOrigen, pNombreColeccionOrigen, pCodColeccionDestino)
- 11ba. Introduce un número que no este ocupado y pulsa 'Aceptar'
- 12ba. traerCromo(pNumeroOrigen, pNumeroDestino, pNombreColeccionOrigen, pCodColeccionDestino) : void
- 13ba. traerCromo(pNumeroOrigen, pNumeroDestino, pNombreColeccionOrigen, pCodColeccionDestino) : void
- 14ba. buscarColeccion(pNombreColeccionOrigen) : Coleccion
- 15ba. buscarColeccion(pCodColeccionDestino) : Colección
- 16ba. buscarCromo(pNumeroOrigen) : Cromo
- 17ba. getFamilia() : Familia
- 18ba. getImagen() : byte[]
- 19ba. getCodFamilia() : int
- 20ba. agregarFamilia(pLaFamilia) : void
- 21ba. execSQL1(sql1)
- 22ba. close
- [Si el número está libre]
- 10bb. traerCromo(pNumero, pNombreColeccionOrigen, pCodColeccionDestino) : void
- 11bb. traerCromo(pNumero, pNombreColeccionOrigen, pCodColeccionDestino) : void
- 12bb. buscarColeccion(pNombreColeccionOrigen) : Coleccion
- 13bb. buscarColeccion(pCodColeccionDestino) : Colección
- 14bb. buscarCromo(pNumero) : Cromo
- 15bb. getFamilia() : Familia
- 16bb. getImagen() : byte[]
- 17bb. getCodFamilia() : int
- 18bb. agregarFamilia(pLaFamilia) : void
- 19bb. execSQL2(sql2)

20bb. close

23. cargarTodosLosCromos() : void // Diagrama cargarTodosLosCromos

24. cargarCromosEnTabla(pCodColeccion) : void // Diagrama Vista Cromos



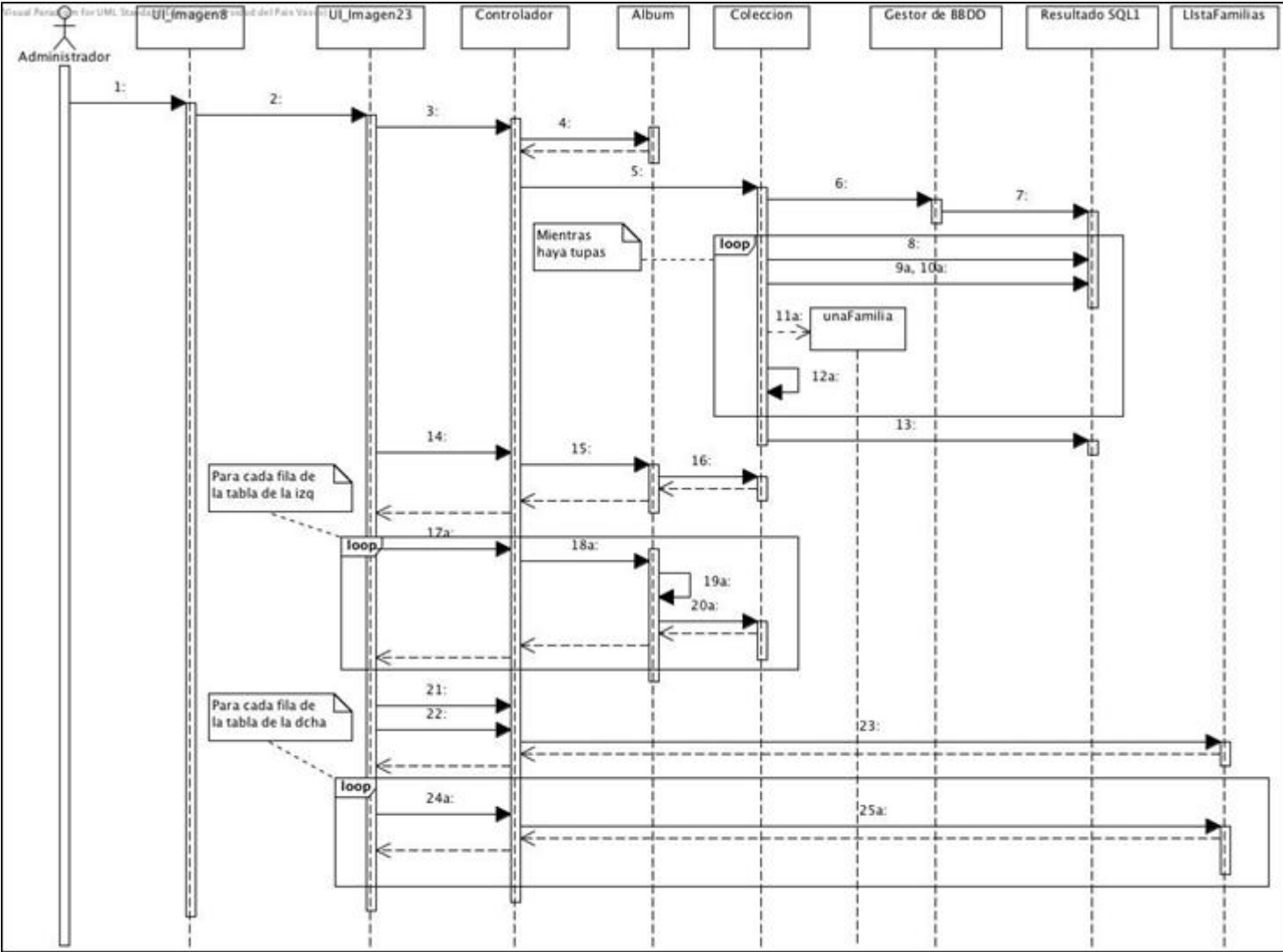
Gestionar familias – Cromos (Gestionar una colección)

// Se cargan las familias de la colección que se está gestionando en la tabla de la izquierda y en la tabla de la derecha se cargan todas las familias.

1. El administrador pulsa 'Gestionar familias'
2. new(pCodColeccion)
3. cargarFamilias(pCodColeccion) : void
4. buscarColeccion() : int
5. cargar Familias() : void
6. execSQL1(sql1)

```
SELECT familia.codFamilia, familia.descripcion
FROM familia, familia_coleccion
WHERE familia.codFamilia = familia_coleccion.codFamilia
and familia_coleccion.codColeccion = pCodColeccion
```

7. new()
8. next()
 - [Mientras haya tuplas]
 - 9a. getCodFamilia(): int
 - 10a. getDescripcion() : String
 - 11a. new Familia(pCodFamilia, pDescripcion) : void
 - 12a. agregarFamilia(pFamilia) : void
13. close
14. contarFamilias(pCodColeccion) : int
15. contarFamilias(pCodColeccion) : int
16. contarFamilias() : int
 - [Para cada una de las filas de la tabla] // Tantas filas como familias en la colección (Tabla izq.)
 - 17a. cargarFamiliaEnTabla(pFila, pCodColeccion) : void
 - 18a. getDatosFamilia(pPos, pCodColeccion) : ArrayList<Object>
 - 19a. buscarColeccion(pCodColeccion) : Coleccion
 - 20a. getDatosFamilia(pPos) : ArrayList<Object>
17. contarColecciones() : int // Se carga el nombre de cada colección en el comboBox
 - [Para cada una de las filas de la tabla] // Tantas filas como colecciones
 - 17a. cargarColeccionEnTabla(pFila) : String
 - 18a. getDatosColeccion(pFila) : String
 - 19a. getDatosColeccion(pPos) : String
20. cargarFamilias() : void // Diagrama cargar Familias
22. contarFamilias() : int
23. contarFamilias() : int
 - [Para cada una de las filas de la tabla] // Tantas filas como familias en total (Tabla dch.)
 - 24a. cargarFamiliaEnTabla(pFila) : void
 - 25a. getDatosFamilia(pFila) : String



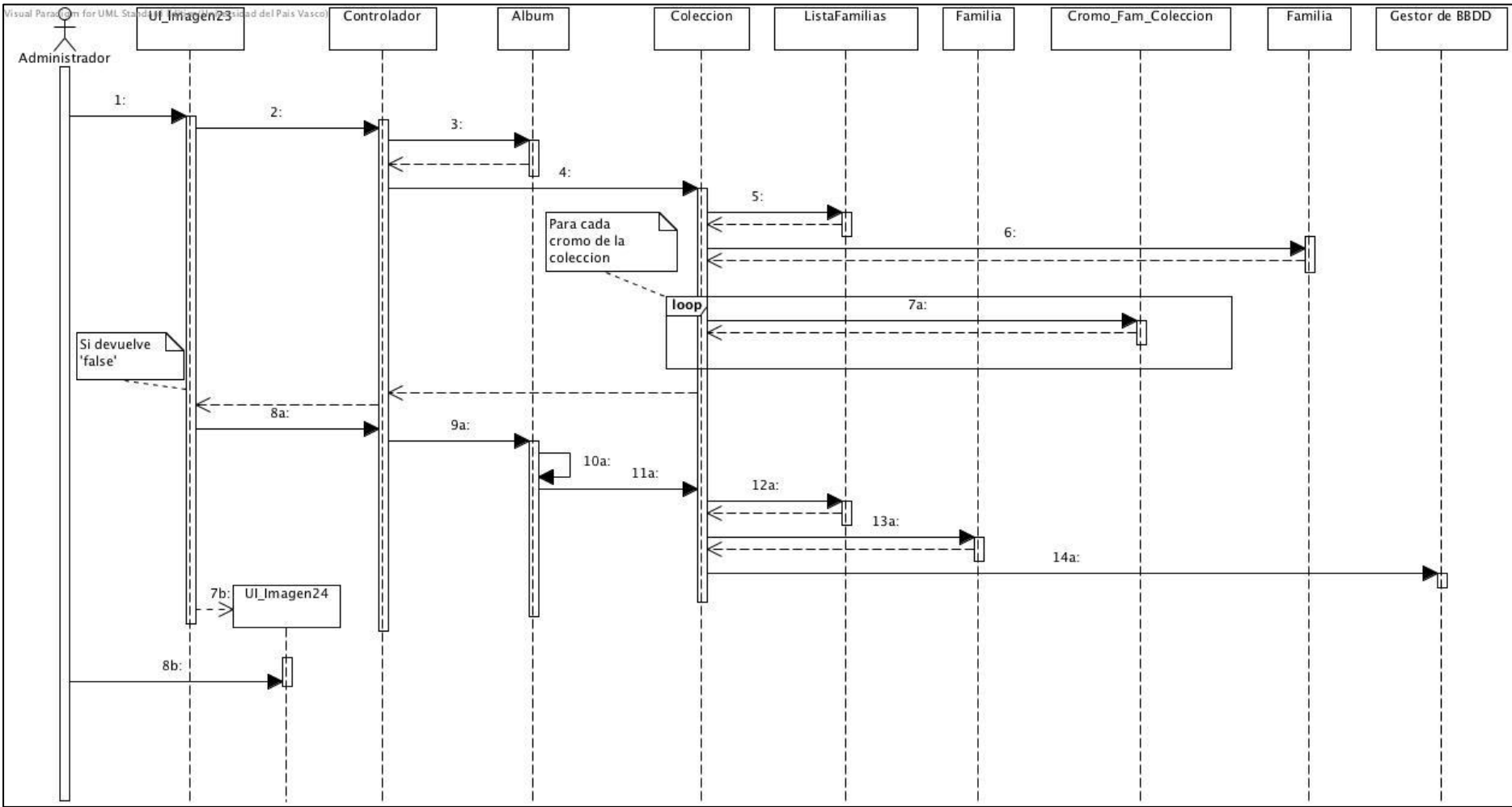
LlevarFamilia – GestionarFamilias

// Elimina una familia de la colección que se está gestionando, siempre que no este siendo usada por otro cromó de la misma.

1. El administrador selecciona una familia de la tabla de la izquierda y pulsa '>'.
 2. existeFamiliaEnCromo(pDescripcion, pCodColeccion) : boolean
 3. buscarColeccion(pCodColeccion) : Coleccion
 4. existeFamiliaEnCromo(pDescripcion) : boolean
 5. buscarFamilia(pDescripcion) : String
 6. getCodigo() : int
 - [Para cada cromó]
 - 7a. getCodFamilia() : int
 - [Si devuelve 'false']
 - 8a. eliminarFamiliaDeColeccion(pDescripcion, pCodcoleccion) : void
 - 9a. eliminarFamiliaDeColeccion(pDescripcion, pCodcoleccion) : void
 - 10a. buscarColeccion(pCodColeccion) : Colección
 - 11a. eliminarFamilia(pDescripcion) : void
 - 12a. buscarFamilia(pDescripcion) : Familia
 - 13a. getCodigo() : int
 - 14a. execSQL1(sql)

```
DELETE FROM familia_coleccion
WHERE codColeccion = pCodColeccion and codFamilia = pCodFamilia
```

- [Si devuelve 'true']
7b. new()
8b. Pulsa 'Aceptar'

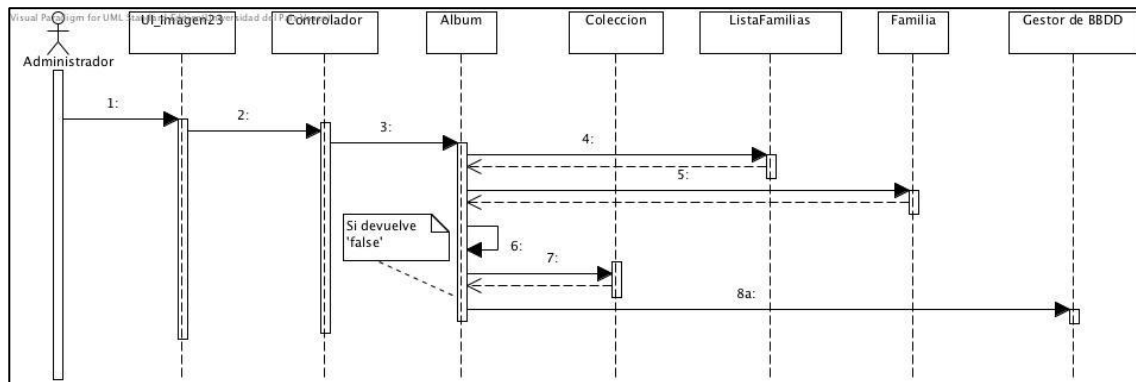


traerFamilia – GestionarFamilias

// Agrega una familia existente a la colección que se está gestionando

1. El administrador selecciona una familia de la tabla de la derecha y pulsa '<'.
 2. agregarFamiliaAColeccion(pDescripcion, pCodColeccion) : void
 3. agregarFamiliaAColeccion(pDescripcion, pCodColeccion) : void
 4. buscarFamilia(pDescripcion) : Familia
 5. getCodigo() : int
 6. buscarColeccion(pCodColeccion) : Colección
 7. existeFamilia(pUnaFamilia) : boolean
 - [Si no existe ya]
 - 8a. execSQL1(sql1)

```
INSERT INTO familia_coleccion (codColeccion, codFamilia)
VALUES pCodColeccion, pCodFamilia
```



Agregar Familia – GestionarFamilias

// Agrega una nueva familia a la colección que se está gestionando y al listado de todas las colecciones.

1. El administrador pulsa 'Agregar familia'.
2. new(pCodColeccion)
3. El administrador introduce el nombre de la familia y pulsa 'ok'
4. existeFamilia(pDescripcion) : boolean
5. existeFamilia(pDescripcion) : boolean
[Si ya existe]
 - 6a. new()
 - 7a. pulsa 'Aceptar'
[Si no existe]
 - 6b. insertarFamilia(pDescripcion, pCodColeccion) : void
 - 7b. insertarFamilia(pDescripcion, pCodColeccion) : void
 - 8b. execSQL1(sql1)

```
INSERT INTO familia (descripcion)
VALUES pDescripcion
```

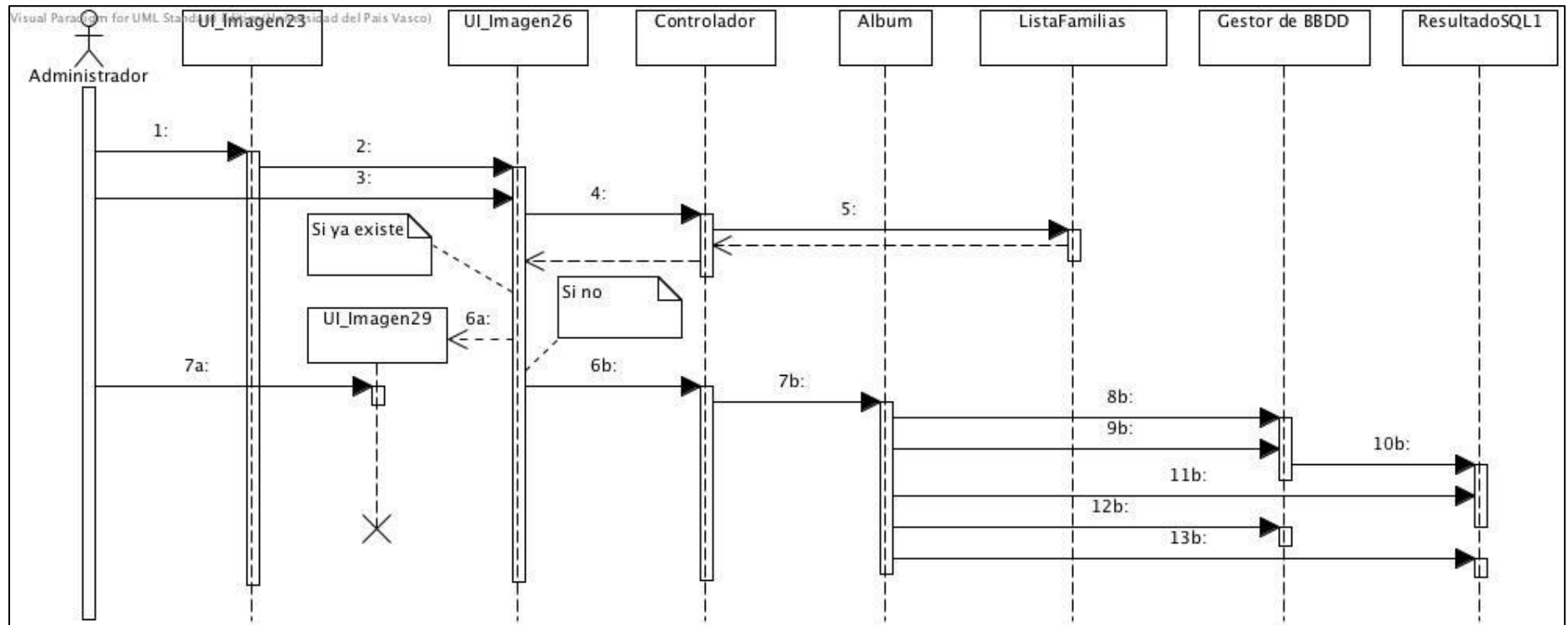
- 9b. execSQL2(sql2)

```
SELECT codFamilia
FROM familia
ORDEN codFamilia desc
```

- 10b. new()
- 11b. getCodFamilia() : int
- 12b. execSQL3(sql3)

```
INSERT INTO familia_coleccion (codColeccion, codFamilia)
VALUES pCodColeccion, pCodFamilia
```

- 13b. close



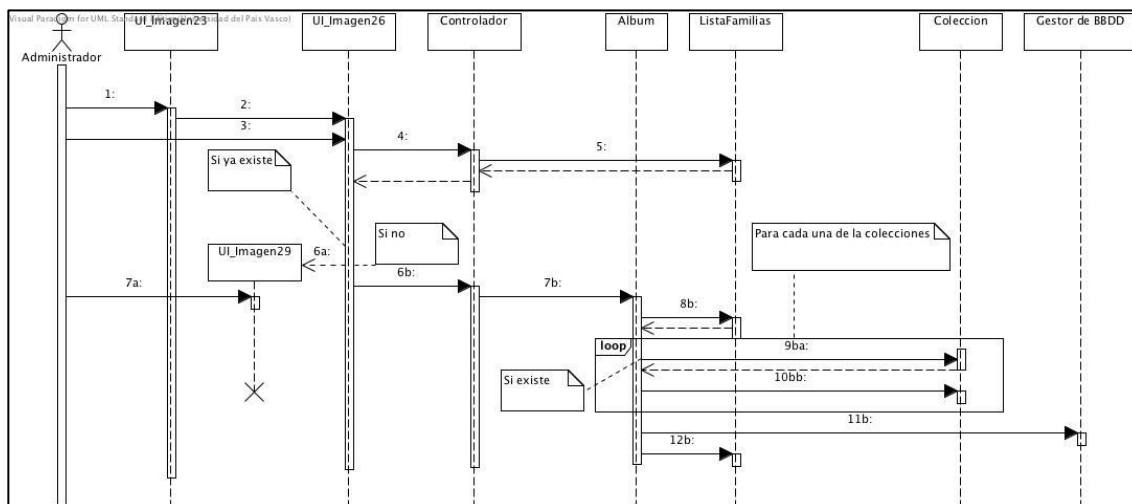
Modificar Familia – GestionarFamilias

// Modifica el nombre de la familia en la base de datos.

1. El administrador selecciona una familia y pulsa 'Modificar familia'.
2. new(pCodColeccion)
3. El administrador introduce el nombre nuevo de la familia y pulsa 'ok'
4. existeFamilia(pDescripcion) : boolean
5. existeFamilia(pDescripcion) : boolean
[Si ya existe]
- 6a. new()
- 7a. pulsa 'Aceptar'
- [Si no]
- 6b. modificarFamilia(pFamiliaVieja, pFamiliaNueva) : void
- 7b. modificarFamilia(pFamiliaVieja, pFamiliaNueva) : void
- 8b. buscarFamilia(pFamiliaVieja) : Familia
[Recorrer el listado de colecciones]
- 9ba. existeFamilia(pFamilia) : boolean
[Si existe]
- 10baa. modificarFamilia(pFamiliaVieja, pFamiliaNueva) : void
- 11b. execSQL1(sql1)

```
UPDATE familia
SET descripcion = pFamiliaNueva
WHERE descripcion = pFamiliaVieja
```

- 12b. modificarFamilia(pFamiliaVieja, pFamiliaNueva) : void



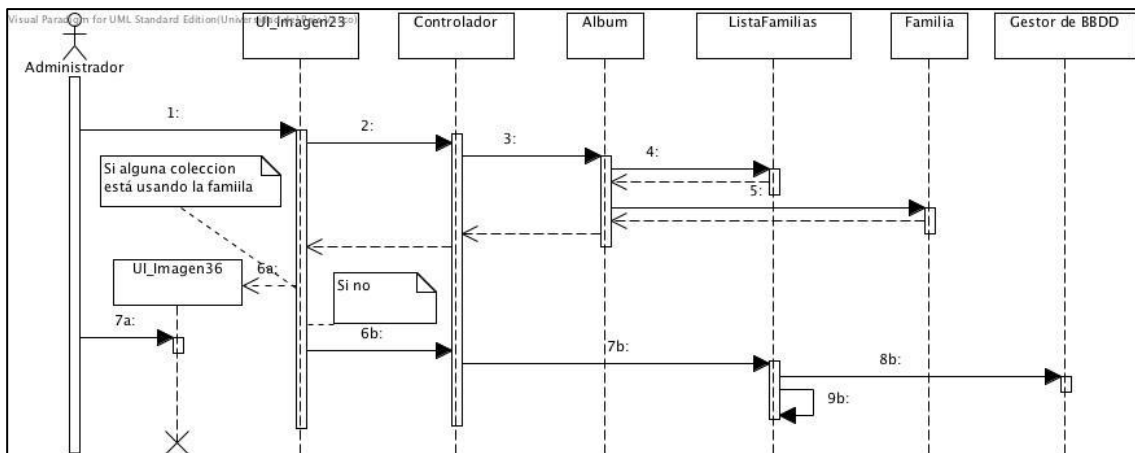
Eliminar Familia – GestionarFamilias

// Modifica el nombre de la familia en la base de datos.

1. El administrador selecciona una familia y pulsa 'Eliminar familia'.
2. existeFamiliaEnColeccion(pDescripcion) : boolean
3. existeFamiliaEnColeccion(pDescripcion) : boolean
4. buscarFamilia(pDescripcion) : Familia
5. existeFamilia(pUnaFamilia) : boolean
[Si ya existe]
- 6a. new()
- 7a. pulsa 'Aceptar'
- [Si no]
- 6b. eliminarFamilia(pDescripcion) : void
- 7b. eliminarFamilia(pDescripcion) : void
- 8b. execSQL1(sql1)

DELETE FROM familia
WHERE descripcion = pDescripcion

9b. Se elimina la familia de la clase lista de familias también



Usuario

Registrarse

1. Pulsa en el botón 'Registrarse'
2. registrar: (id) sender
3. setData(...) -- De controlador a controlador
4. Introduce 'Usuario', 'Contraseña' y 'Email' y pulsa 'Aceptar'
5. guardar: (id) sender
 - [Si los campos no están correctamente rellenos]
 - 6a. mostrarAlerta
 - 7a. Pulsa 'ok'
 - [Si los campos están correctamente rellenos]
 - 6b. registro.php?nombre=%@&pass=%@&email=%@
 - 7b. execSQL1(sql1) :

// Inserción de los datos en el servidor

```
INSERT INTO usuario (nombre, password, email)
VALUES ('$nombre', '$pass', '$email')
```

// Devuelve el password con una encriptación tipo md5

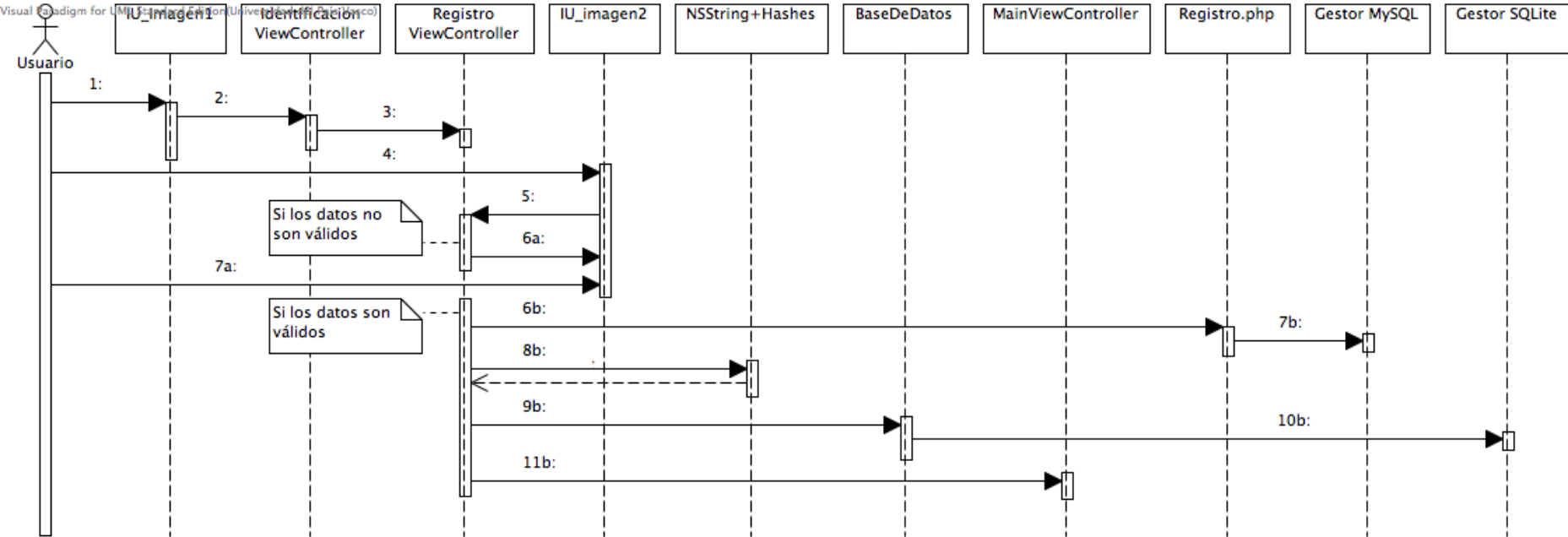
8b. [pass.text md5] : NSString

// Inserción de datos en la base de datos de la aplicación.

9b. [bd registrarUsuario:nombre.text password:[pass.text md5] email:email.text];

10b. execSQL1(sql1)

11b. setData(...)



Identificarse

1. Introduce los datos de acceso y pulsa 'Entrar'
2. identificar:(id)sender
 - [Si hay campos vacíos]
 - 3a. mostrar alerta
 - 4a. pulsa 'ok'
 - [Si no]
 - 3b. [pass.text md5] : NSString
 - 4b. comprobarUsuario.php?nombre=%@&password=%@
 - 5b. execSQL2(sql2) :

```
SELECT nombre, password, email, latitud, longitud
FROM usuario
WHERE nombre = '$nombre' and password = '$pass'
```

- 6b. new()
- 7b. getDatos() : NSData
 - [Si los datos no son correctos]
 - 8ba. mostrar alerta
 - 9ba. Pulsa 'ok'
 - [Si los datos son correctos]
 - // Se insertan los datos en la base de datos local
 - 8bb. [empezarAParsear:urlData]: NSDictionary
 - 9bb. [registrarUsuario:nombre.text password:[pass.text md5] email: [dicDatos objectForKey:@"email"]];
 - 10bb. execSQL1(sql1):

```
INSERT INTO usuario (nombre, password, email)
VALUES ('$nombre', '$pass', '$email')
```

- 11bb. obtenerColeccionesUsuario.php?usuario=%@
- 12bb. execSQL3(sql3)
- // Se obtienen las colecciones en las que participa el usuario.

```
SELECT codColeccion
FROM colecciones_usuario
WHERE nombre = '$nombre'
```

- 13bb. new()
- 14bb. getDatos() : NSData
 - [Si tiene colecciones]
 - [Para cada colección]
 - 15bba. (void) agregarColeccion:(NSString *) codColeccion
 - 16bba. obtenerUnaColeccion.php?codColeccion=%@

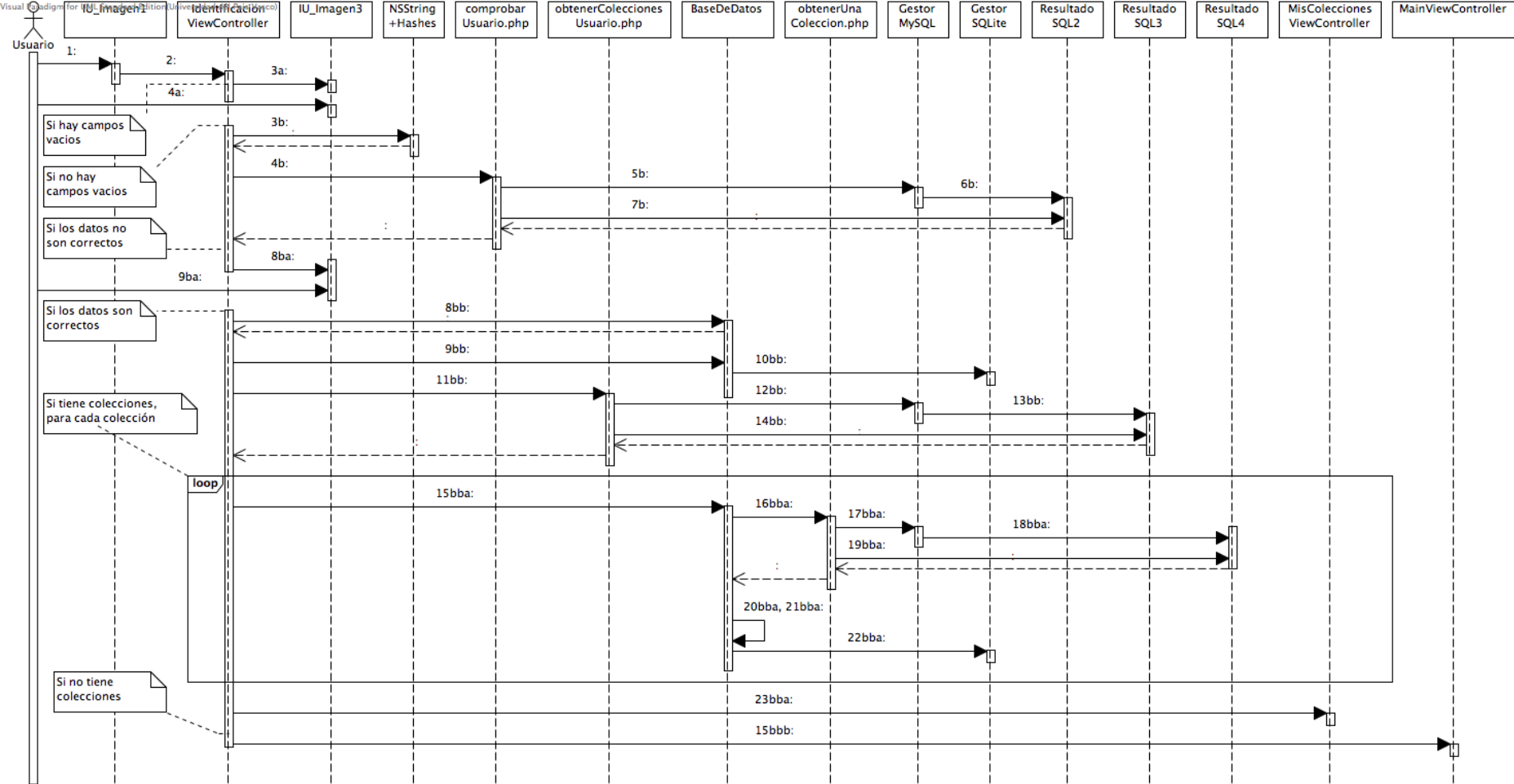
```
17bba. exeSQL4(sql4)
// Se obtienen los datos de una colección
```

```
SELECT codColeccion, nombre, descripcion, imagen
FROM coleccion WHERE codColeccion = '$codColeccion'
```

```
18bba. new()
19bba. getDatos() : NSData
20bba. [empezarAParsear:urlData]: NSDictionary
// Se guarda la imagen en un directorio de la aplicación
21bba. [writeToFile:pngFilePath]
22bba. exeSQL5(sql5):
// Insertar datos de la colección en la base de datos de la aplicación
```

```
INSERT INTO coleccion (codColeccion, nombre, descripcion, imagen)
VALUES ('codColeccion', 'nombre', 'descripcion', 'rutaImagen')
```

```
// Accede a la lista de sus colecciones
23bba. setDatos(...)
[ Si no tiene colecciones ]
// Accede al catálogo completo
15bbb. setDatos(...)
```



Acceder al catálogo completo

// Primero se cargan las colecciones en las que no participa el usuario

1. exeSQL6(sql6):

```
SELECT codColeccion, nombre, descripcion, imagen
FROM coleccion
WHERE codColeccion NOT IN (SELECT codColeccion
FROM colecciones_usuario
WHERE nombre = '$usuario')
```

2. new()
3. getDatos() : NSData
4. [empezarAParsear:urlData]: NSDictionary
[Por cada una de las colecciones]
- 5a. Se cargan en la tableView los datos de cada colección.
6. El usuario selecciona una colección.
7. [detalle setCodColeccion:[coleccion objectForKey:@"codColeccion"]];
8. [detalle setNombreColeccionString:[coleccion objectForKey:@"nombre"]];
9. [detalle setTextDetalleString:[coleccion objectForKey:@"descripcion"]];
10. [detalle setImagenColeccionDetalle:[UIImage alloc] initWithData:datosImagen];
11. [nombreColeccion setText:[NSString stringWithFormat:@"%@",nombreColeccionString]];
12. [detalleColeccion setText:[NSString stringWithFormat:@"%@",textDetalleString]];
13. [imagenColeccion setImage:imagenColeccionDetalle];
14. Pulsa sobre 'Empezar colección'
15. coleccionar:(id)sender
16. insertarColeccionUsuario.php?nombre=%@&codColeccion=%@
17. execSQL7(sql7):

// Insertar colección en la tabla colecciones_usuario

```
INSERT INTO colecciones_usuario (nombre,codColeccion)
VALUES ('$nombre','$codColeccion')
```

// Inserción de datos en la base de datos de la aplicación.

18. [agregarColeccion:codColeccion];
19. execSQL7(sql7)
20. [agregarFamilias:codColeccion];
21. obtenerFamiliasColeccion.php?codColeccion=%@
22. execSQL8(sql8):

```
SELECT A.codFamilia, A.descripcion
FROM familia A, familia_coleccion B
WHERE A.codFamilia = B.codFamilia AND B.codColeccion = '$codColeccion'
```

23. new()
24. getDatos() : NSData

[Por cada una de las familias de esa colección]

25a. execSQL9(sql9) :

```
INSERT INTO familia (codFamilia,descripcion) VALUES ('codFamilia','descripcion');

INSERT INTO familia_coleccion (codFamilia,codColeccion)
VALUES ('codFamilia','codColeccion');
```

26. [setDatosColeccion] // Paso de parámetros entre controladores.

27. [serDatosVista]; // Se pasan los parámetros del controlador a la vista

28. El usuario toca el sobre

29. abrirSobre:(id)sender

30. insertarCromo.php?nombre=%@&codColeccion=%@&numero=%d

31. execSQL10(sql10):

// Inserción de un cromos en la base de datos Hay comprobar si se tiene ya el cromos para ver si es una inserción o una actualización. Se ha implementado todo en un php.

// Se comprueba si se tiene el cromos

```
SELECT A.codCromo FROM cromosUsuario A, cromos_fam_coleccion B
WHERE A.codCromo = B.codCromo
AND A.nombre = '$nombre'
AND B.codColeccion = '$codColeccion'
AND B.numero = $numero
```

// Si no se tiene, se inserta

```
INSERT INTO cromosUsuario ( nombre, codColeccion, codCromo, nVeces)
SELECT '$nombre', '$codColeccion', codCromo, 1
FROM cromos_fam_coleccion
WHERE codColeccion = '$codColeccion' AND numero = '$numero'
```

// Si se tiene, hay que identificar el codCromo e incrementar en 1 el campo nVeces.

1. Del resultado de la primera select, \$rs, se obtiene el campo codCromo de la fila 0

```
$codCromo = mysql_result($rs,0,'codCromo');
```

2. Se obtiene el numero de veces que el usuario tiene ese cromos:

```
SELECT nVeces FROM cromosUsuario WHERE nombre = '$nombre'
AND codColeccion = '$codColeccion' AND codCromo = '$codCromo'
```

3. Se obtiene el numero de veces de la select y actualiza la tabla:

```
$nVeces = mysql_result($select,0,'nVeces');

UPDATE cromosUsuario SET nVeces = '$nVeces' + 1
WHERE nombre = '$nombre'
AND codColeccion = '$codColeccion'
AND codCromo = '$codCromo';
```

Ahora hay que o bien insertar o bien aumentar el cromos en la base de datos de la aplicación.

32. obtenerUnCromo.php?codColeccion=%@&numero=%d

33. execSQL11(sql11):

```
SELECT A.numero, B.codCromo, B.titulo, B.descripcion, C.codFamilia, C.descripcion
desFamilia, A.imagen FROM cromos_fam_coleccion A, cromos B, familia C
WHERE A.codCromo = B.codCromo AND A.codFamilia = C.codFamilia
AND A.codColeccion = '$codColeccion' AND A.numero = '$numero'
AND codCromo = '$codCromo';
```

34. new()

35. getDatos() : NSData

36. [bd existeCromo: codCromo codColeccion:codColeccion] : boolean

[Si no existe cromos]

37a. [agregarCromo:datosCromo]

38a. execSQL12(sql12)

```
INSERT INTO cromos (codCromo, titulo, descripcion)
VALUES ('codCromo', 'titulo', 'descripcion');

INSERT INTO cromos_fam_coleccion (codCromo, codFamilia, codColeccion, imagen,
numero, numeroVeces)
VALUES ('codCromo', 'codFamilia', 'codColeccion', 'rutaImagen', 'numero', 'nVeces')
```

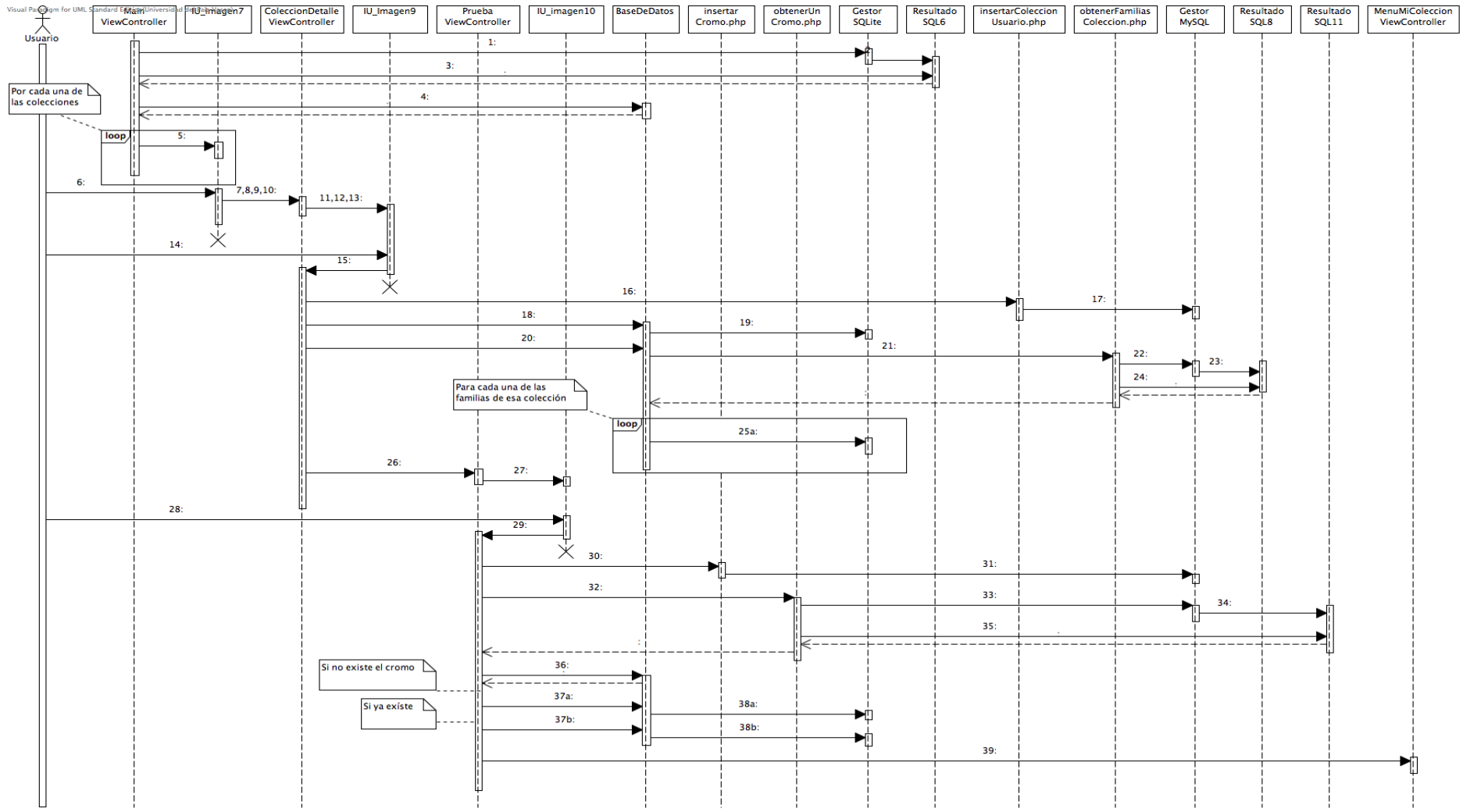
[Si existe cromos]

37b. [actualizarCromo:datosCromo]

38b. execSQL13(sql13)

```
UPDATE cromo_fam_coleccion  
SET numeroVeces = (SELECT numeroVeces FROM cromo_fam_coleccion  
WHERE codColeccion = 'codColeccion' AND codCromo = 'codCromo') +1  
WHERE codColeccion = 'codColeccion' AND codCromo = 'codCromo'
```

39. setDatos(codColeccion, imagenColeccion, nombreColeccion) *Caso de uso acceder a una colección*



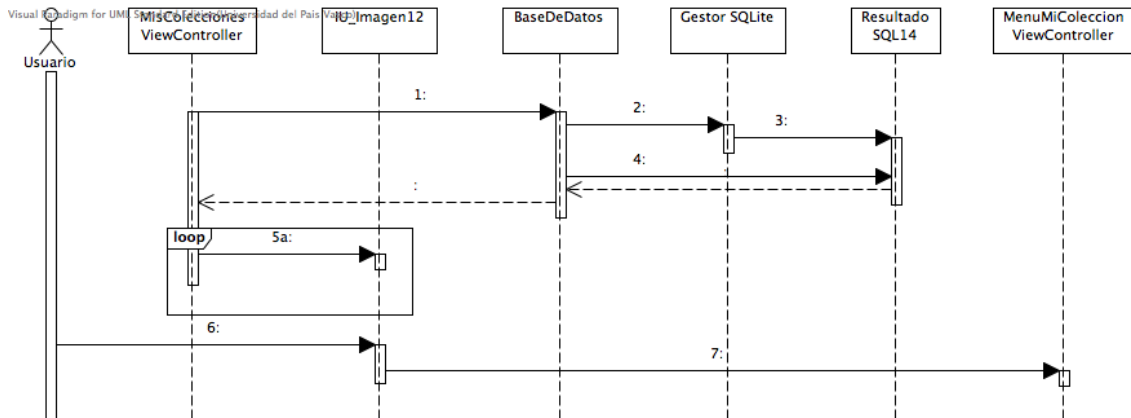
Listar tus colecciones

// Primero se cargan las colecciones que el usuario posee de la base de datos de la aplicación.

1. [obtenerMisColecciones]: NSMutableArray
2. exeSQL14(sql14):

```
SELECT * FROM coleccion
```

3. new()
4. getDatos() : NSDictionary
[Por cada una de las colecciones]
 - 5a. Se cargan en la tableView los datos de cada colección.
6. El usuario selecciona una colección.
7. setData(codColeccion, imagenColeccion, nombreColeccion) *Caso de uso acceder a una colección*



Acceder a una colección

Se parte desde la pestaña Album

1. obtenerCromos.php?codColeccion=%@
2. exeSQL15(sql15):

```
SELECT A.numero, B.titulo, B.descripcion, C.descripcion familia, A.imagen
FROM cromo_fam_coleccion A, cromo B, familia C
WHERE A.codCromo = B.codCromo
AND A.codFamilia = C.codFamilia
AND A.codColeccion = '$codColeccion'
ORDER BY A.numero
```

3. new()
4. getDatos() : NSData
5. obtenerCromosUsuario.php?codColeccion=%@&usuario=%@
6. exeSQL16(sql16):

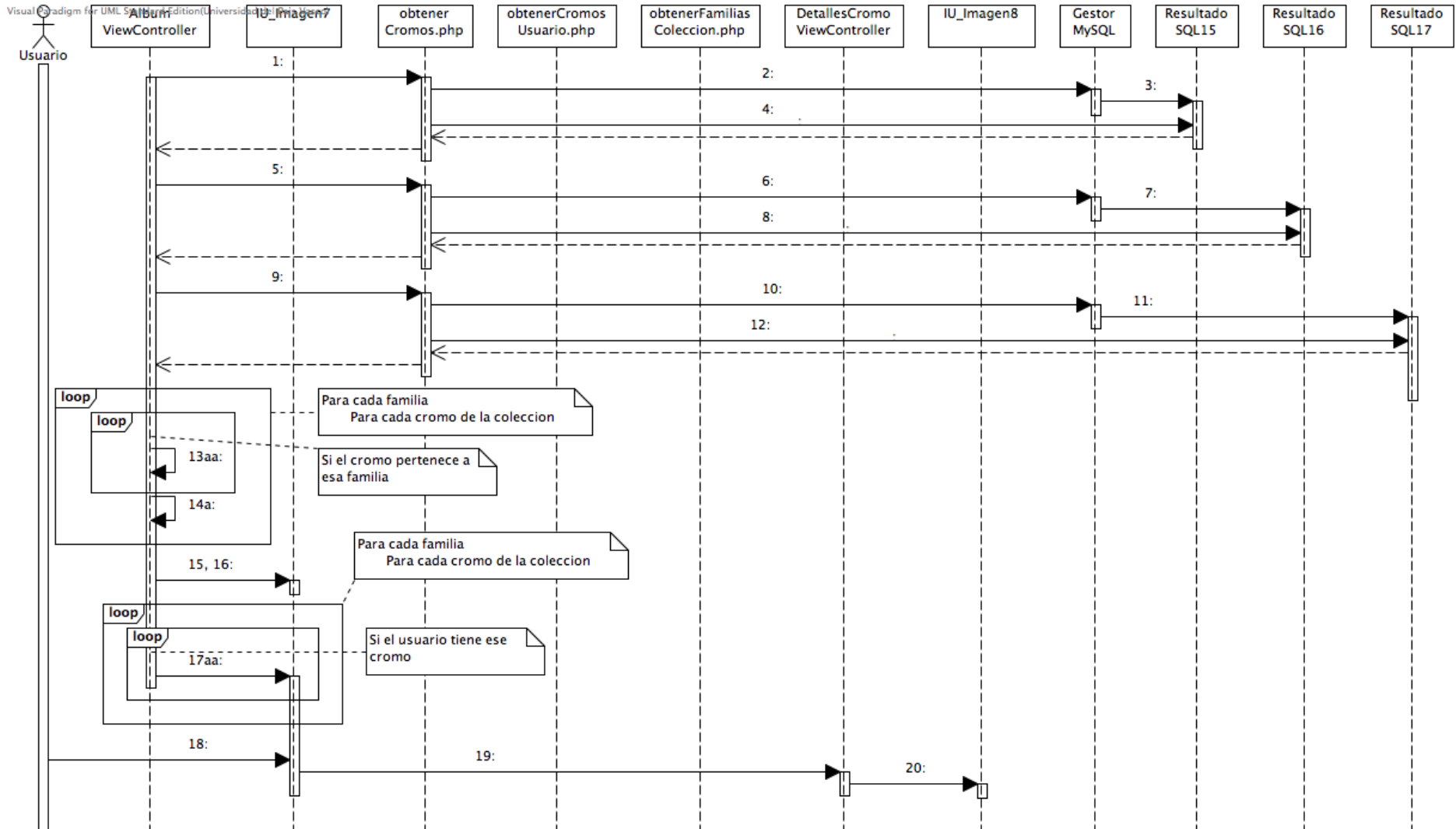
```
SELECT A.numero, B.codCromo, B.titulo, B.descripcion, C.codFamilia,
C.descripcion desFamilia, A.imagen, D.nVeces, D.bloqueado
FROM cromo_fam_coleccion A, cromo B, familia C, cromosUsuario D
WHERE A.codCromo = B.codCromo
AND A.codFamilia = C.codFamilia
AND A.codColeccion = D.codColeccion
AND A.codCromo = D.codCromo
AND D.nombre = '$usuario'
AND A.codColeccion = '$codColeccion'
```

7. new()
8. getDatos() : NSData
9. obtenerFamiliasColeccion.php?codColeccion=%@
10. exeSQL17(sql17):

```
SELECT A.codFamilia, A.descripcion FROM familia A, familia_coleccion B
WHERE A.codFamilia = B.codFamilia AND B.codColeccion = '$codColeccion'
```

11. new()
12. getDatos() : NSData
 - // Se agrupan los cromos por familia para poder mostrarlo en secciones.
 - [Para cada familia de esa colección]
 - [Para cada cromo de esa colección]
 - [Si el cromo pertenece a esa familia]
 - 13aa. [Array cromosFamilia addObject:datosCromo];
 - 14a. [Array album addObject:cromosFamilia];
 - // Se establece el numero de secciones, tantas como familias
 - 15. numberOfSectionsInCollectionView -> familiasColeccion.count
 - // Se establece el numero de ítems por seccion, tantos como cromo haya en cada familia

16. numberOfItemsInSection -> [album objectAtIndex:index:section] count];
[Para cada cromo de la colección]
[Para cada cromo del usuario]
[Si el usuario tiene ese cromo]
- 17aa. mostrarCromo
18. El usuario toca un cromo de su colección.
19. [setDatosDetalleCromo]
20. [setDatosVista]

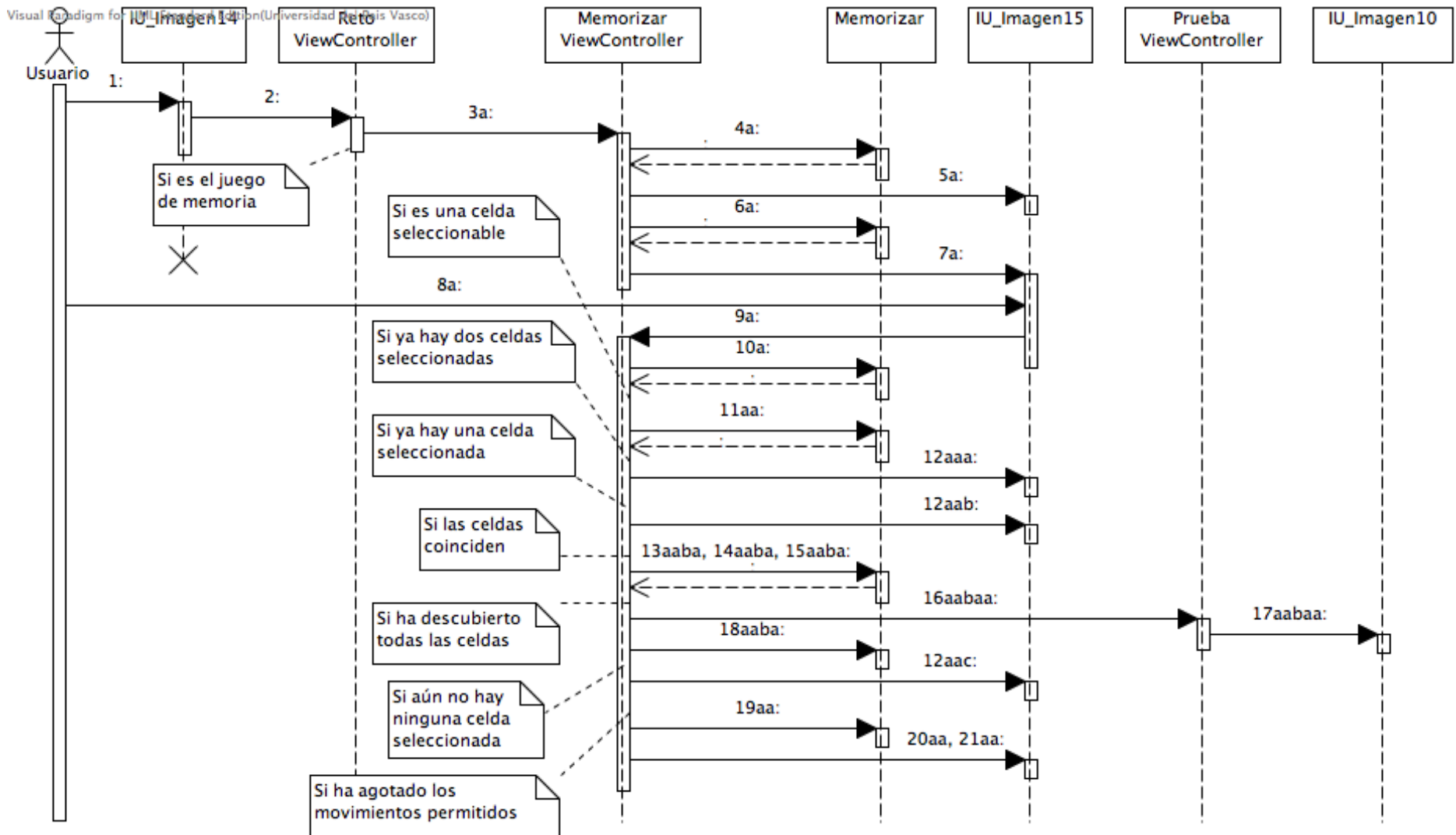


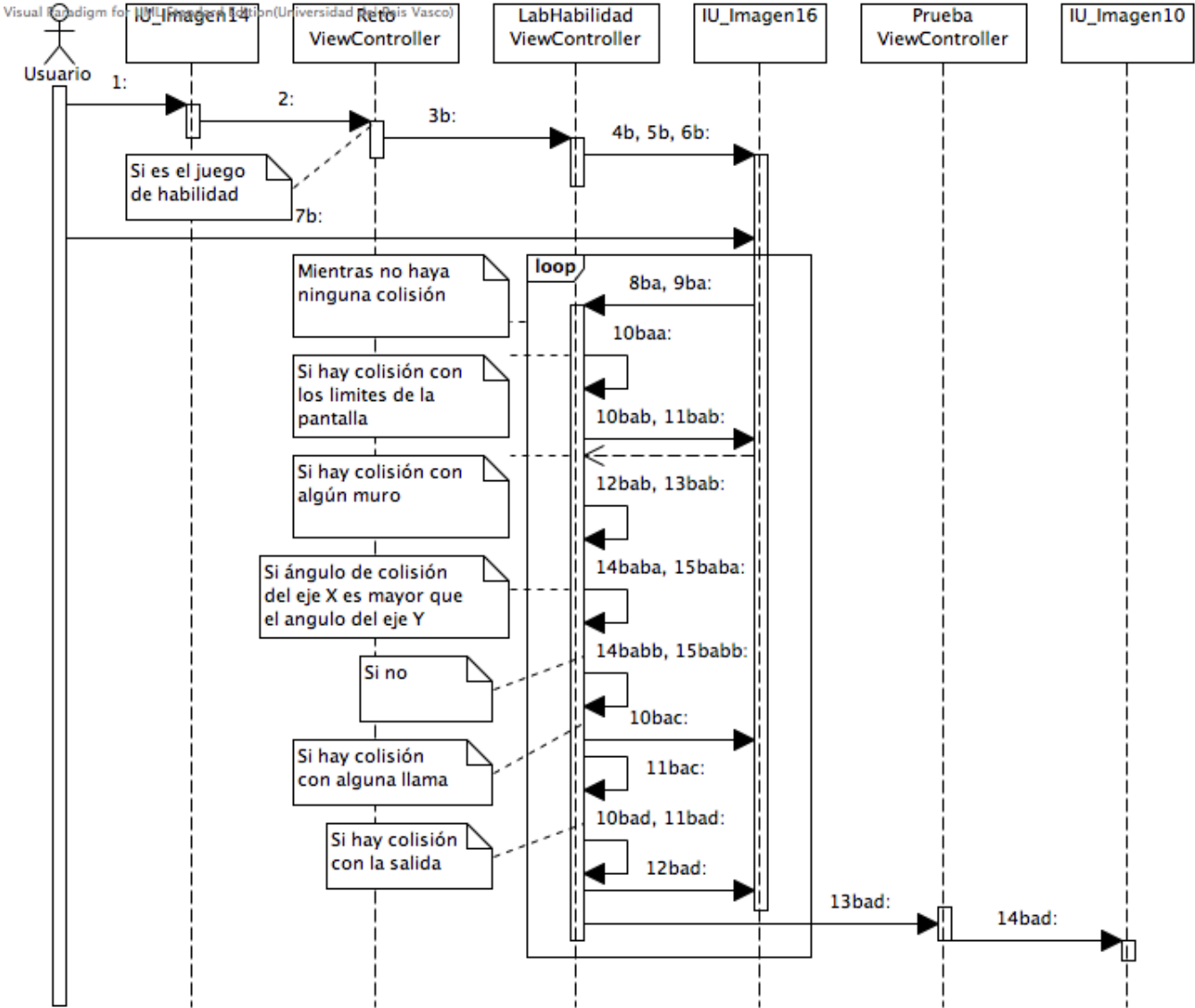
Afrontar un reto

Se parte desde la pestaña Reto

1. El usuario pulsa sobre 'Empezar reto aleatorio'
2. (IBAction)laberinto:(id)sender
 - [si numero == 0]
 - 3a. setDatos
 - 4a. juegoMemorizar = [instancia Memorizar]
 - 5a. numberOfSectionsInCollectionView : 1
 - 6a. [_juegoMemorizar.arrayDeElementos count]: int
 - 7a. numberOfItemsInSection : [_juegoMemorizar.arrayDeElementos count];
 - 8a. El usuario toca una de las celdas
 - 9a. [collectionView didSelectItemAtIndexPath]
 - 10a. cellAtIndexPathCanBeSelected: Boolean // Se comprueba si la celda es seleccionable
 - [Si es una celda seleccionable]
 - 11aa. [_juegoMemorizar arrayDeSeleccionados count] : int
 - [Si [_juegoMemorizar.arrayDeSeleccionados count] == 2]
 - 12aaa. Oculta las celdas que había mostradas
 - [Si [_juegoMemorizar.arrayDeSeleccionados count] == 1]
 - 12aab. Mostrar celda seleccionada
 - [Si coinciden las dos celdas]
 - 13aabaa. _juegoMemorizar.totalDescubiertos +2
 - 14aabaa. Vaciar arrayDeSeleccionados
 - 15aabaa. [_juegoMemorizar totalDescubiertos] : int
 - [Si _juegoMemorizar.totalDescubiertos == 16]
 - 16. aabaaa. [setDatosColecion] // Paso de parámetros entre controladores.
 - 17. aabaaa. [serDatosVista]; // Se pasan los parám. del controlador a la vista
 - 18aab. _movimientosPermitidos -= 1
 - [Si [_juegoMemorizar.arrayDeSeleccionados count] == 0]
 - 12aac. Mostrar celda seleccionada
 - [Si movimientosPermitidos == 0]
 - 19aa. [_juegoMemorizar nuevoJuego];
 - 20aa. [_collectionView reloadData];
 - 21aa. actualizarEtiqueta movimientos permitidos;
- [Si numero ==1]
- 3b. setDatos
- 4b. [supportedInterfaceOrientations: UIInterfaceOrientationMaskLandscapeRight];
- 5b. animar las llamas
- 6b. iniciar acelerómetro y establecer frecuencia de actualización
- 7b. El usuario mueve el móvil
- [Mientras no haya alguna colisión]
- 8ba. Update // Se establece la nueva posición de la gota
- 9ba [MoverGota] // Se establece la nueva posición de la gota y y se situa la gota
- [Si hay colisión con los limites de la pantalla]
- 10baa. gotaVelocidad = -(self.gotaVelocidad / 2.0); // Rebota la gota
- [Si hay colisión con el muro]
- 10bab. obtenerCentroDelMuro
- 11bab. obtenerCentroDeGota
- 12bab. anguloColisionX = CentroGotaX - CentroMuroX
- 13bab. anguloColisionY = CentroGotaY - CentroMuroY

```
[Si anguloColisionX< anguloColisionY]
14baba. _posicionActual.x = self.posicionAnterior.x;
15baba. gotaVelocidadX = -(self.gotaVelocidadX / 2.0);
[si no]
14babb. _posicionActual.y = self.posicionAnterior.y;
15babb. gotaVelocidadY = -(self.gotaVelocidadY / 2.0);
[Si hay colisión con la llama]
10bac. Mostrar alerta
11bac. posicionActual = CGPointMake(0, 144);
[Si hay colisión con la salida]
10bad. self.posicionActual = CGPointMake(0, 144);
11bad. [motionManager stopAccelerometerUpdates];
12bad. MostrarAlerta
13bad. [setDatosColecion] // Paso de parámetros entre controladores.
14bad. [serDatosVista]; // Se pasan los parámetros del controlador a la vista
```





Batalla

1. obtenerUsuariosColeccion.php?codColeccion=%@&usuario=%@
2. exeSQL18(sql18):

```
SELECT DISTINCT A.nombre
FROM usuario A, colecciones_usuario B, cromosUsuario C
WHERE A.nombre = B.nombre AND A.nombre = C.nombre
AND B.codColeccion = C.codColeccion AND A.nombre <> '$usuario'
AND B.codColeccion = '$codColeccion'
```

3. new()
4. getDatos() : NSData
5. empezarAParsear: NSDictionary

– Propuestas de batalla recibidas

6. obtenerPropuestasBatRecibidas.php?codColeccion=%@&usuario=%@
7. execSQL19(sql19):

```
SELECT DISTINCT A.fechaHora, A.usuarioRetador, A.codColeccion, A.tipo
FROM batalla A
WHERE A.usuarioRetado = '$usuario'
AND A.codColeccion = '$codColeccion' AND A.estado = 'pendiente'
```

8. new()
9. getDatos():NSData
 - [Si ha recibido propuestas]
 - 10a. empezarAParsear:NSDictionary
 - 11a. mostrar btnPropuestasBatRecibidas
 - 12a. mostrar alerta
 - 13a. El usuario selecciona algo
 - [Si el usuario selecciona 'Ver Propuestas']
 - 14aa. clickedButtonAtIndex(buttonIndex) // Se pasa el índice del botón seleccionado
 - 15aa.setDatos(propuestasRecibidas)
 - [Para cada propuesta recibida]
 - 16aaa. Se cargan los datos en la celda correspondiente del tableview
 - 17aa. El usuario pulsa un botón dentro de una celda de propuesta
 - [Si el usuario selecciona 'Rechazar']
 - 18aaa.rechazarBat: (UIButton*) sender
 - 19aaa.rechazarBatalla.php?fecha=%@&hora=%@&usuarioRetador=%@&usuarioRetado=%@&codColeccion=%@
 - 20aaa. execSQL20(sql20):

```
UPDATE batalla SET estado = 'rechazada'
WHERE fechaHora = '$fechaHora'
AND usuarioRetador = '$usuarioRetador'
AND usuarioRetado = '$usuarioRetado'
```

21aaa. [propuestasRecibidas removeObjectAtIndex:sender.tag];
 22aaa. [tableView reloadData];
 [Si el usuario selecciona 'Empezar']
 18aab.empezarBatalla: (UIButton*) sender
 19aab.empezarBatalla.php?fecha=%@&hora=%@&usuarioRetador=%@
 &usuarioRetado=%@&codColeccion=%@
 20aab. execSQL21(sql21):

```
UPDATE batalla SET estado = 'en curso'
WHERE fechaHora = '$fechaHora'
AND usuarioRetador = '$usuarioRetador'
AND usuarioRetado = '$usuarioRetado'
```

21aab. [propuestasRecibidas removeObjectAtIndex:sender.tag];
 22aab. setDatos(...)

– Propuestas de batalla realizadas

23. obtenerPropuestasBatRealizadas.php?codColeccion=%@&usuario=%@
 24. execSQL22(sql22):

```
SELECT A.fechaHora, A.usuarioRetado, A.tipo, A.estado
FROM batalla A
WHERE A.usuarioRetador = '$usuario'
AND A.codColeccion = '$codColeccion'
AND (A.estado = 'pendiente' OR A.estado = 'en curso')
AND A.puntosRetador is null
```

25. new()
 26. getDatos(): NSData
 [Si ha realizado alguna propuesta]
 27a. empezarAParsear: NSDictionary
 28a. mostrar btnPropuestasBatRealizadas
 29a. El usuario pulsa el botón 'Realizadas'
 30a. clickedButtonAtIndex(buttonIndex)
 31a. setDatos(propuestasRealizadas)
 [Para cada propuesta recibida]
 32aa. Se cargan los datos en la celda correspondiente del tableview
 33a. El usuario pulsa un botón dentro de una celda de propuesta
 [Si el usuario selecciona 'Anular']

34aa. anularBatalla: (UIButton*) sender
 35aa. anularBatalla.php?fecha=%@&hora=%@&usuarioRetador=%@
 &usuarioRetado=%@&codColeccion=%@
 36aa. execSQL23(sql23):

```
UPDATE batalla SET estado = 'anulada' WHERE fechaHora = '$fechaHora'
AND usuarioRetador = '$usuarioRetador' AND usuarioRetado = '$usuarioRetado'
```

37aa. [propuestasRealizadas removeObjectAtIndex:sender.tag];
 38aa. [tableView reloadData];
 [Si el usuario selecciona 'Empezar']
 34ab. empezarBatalla: (UIButton*) sender
 35ab. empezarBatalla.php?fecha=%@&hora=%@&usuarioRetador=%@
 &usuarioRetado=%@&codColeccion=%@
 36aab. execSQL21(sql21):
 37aab. [propuestasRealizadas removeObjectAtIndex:sender.tag];
 38aab. setDatos(...)

– Batallas en curso

39. obtenerBatallasEnCurso.php?codColeccion=%@&usuario=%@
 40. execSQL24(sql24):

```
SELECT A.fechaHora, A.usuarioRetado, A.usuarioRetador, A.tipo, A.estado,
A.usuarioGanador, A.puntosRetador, A.puntosRetado
FROM batalla A
WHERE (A.usuarioRetador = '$usuario' OR A.usuarioRetado = '$usuario')
AND A.codColeccion = '$codColeccion'
AND (A.estado = 'en curso' OR A.estado = 'resuelta')
```

41. new()
 42. getDatos(): NSData
 [Si hay batallas en curso]
 43a. empezarAParsear: NSDictionary
 44a. mostrar btnPropuestasBatEnCurso
 45a. El usuario pulsa el botón 'En curso'
 46a. clickedButtonAtIndex(buttonIndex)
 47a. setDatos(propuestasEnCuso)
 [Para cada propuesta en curso]
 48aa. Se cargan los datos en la celda correspondiente del tableview
 49a. El usuario pulsa un botón dentro de una celda de propuesta
 50a. empezarBatalla: (UIButton*) sender
 51a. finalizarBatalla.php?fecha=%@&hora=%@&usuario=%@&tipoUsuario=%@
 &codColeccion=%@
 52a. execSQL25(sql25):

// Se le indica si el usuario que ha pulsado el botón ha sido el retador o el retado para que actualice correctamente el registro que le corresponde.

```

if ($tipoUsuario == 'usuarioRetador'){
UPDATE batalla SET estado = 'finalizada'
WHERE fechaHora = '$fechaHora' AND usuarioRetador = '$usuario'
AND codColeccion = '$codColeccion';
}
else{
UPDATE batalla SET estado = 'finalizada'
WHERE fechaHora = '$fechaHora' AND usuarioRetado = '$usuario'
AND codColeccion = '$codColeccion';
}

```

[Si el botón pulsado es 'Cerrar'] // Ha perdido y aún no lo sabía
53aa. [propuestasEnCurso removeObjectAtIndex:sender.tag];
[Si el botón pulsado es 'obtén un cromó'] // Ha ganado y aún no lo sabía
53ab. [propuestasEnCurso removeObjectAtIndex:sender.tag];
54ab. comprobarBloqueos.php?codColeccion=%@&usuario=%@
55ab. execSQL26(sql26):

```

SELECT A. NUM_BLOQUEADOS_USUARIO
FROM
(SELECT count(*) NUM_BLOQUEADOS_USUARIO FROM cromosUsuario
WHERE nombre = '$usuario' AND bloqueado <> nVeces
AND codColeccion = '$codColeccion') A,
(SELECT count(*) TOT_CROMOS_USUARIO FROM cromosUsuario
WHERE nombre = '$usuario' AND codColeccion = '$codColeccion') B
WHERE A.NUM_BLOQUEADOS_USUARIO = B. TOT_CROMOS_USUARIO

```

56ab. new()
57ab. getDatos(): NSData
[Si no están todos bloqueados]
58aba. setDatos(...) // Entre los param se le dice que es modo batalla

* El siguiente paso es obtener un cromó de un usuario. Al ser un proceso de varios pasos se ha preferido desarrollar en un diagrama aparte -> Obtener Cromó Usuario

– Provocar batalla

59. El usuario selecciona un usuario de la lista.
60. didSelectRowAtIndexPath:(NSIndexPath *)indexPath
61. setDatos(...)
62. obtenerCromosUsuario.php?codColeccion=%@&usuario=%@
63. execSQL27(sql27):


```
SELECT A.numero, B.codCromo, B.titulo, B.descripcion, C.codFamilia,
C.descripcion desFamilia, A.imagen, D.nVeces, D.bloqueado
FROM cromosUsuario D
WHERE A.codCromo = B.codCromo AND A.codFamilia = C.codFamilia
AND A.codColeccion = D.codColeccion AND A.codCromo = D.codCromo
AND D.nombre = '$usuario' AND A.codColeccion = '$codColeccion'
```

64. new()

65. getDatos(): NSData

66. empezarAParsear(): NSDictionary

[Para cada cromos del usuario]

67a. cargar datos cromos en vista.

68. El usuario selecciona el tipo de batalla y pulsa 'Batalla'

69. botProvocarBatalla:(id)sender

70. propuestaBatalla.php?usuarioRetador=%@&usuarioRetado=%@&tipo=%@&codColeccion=%@

71. execSQL28(sql28):

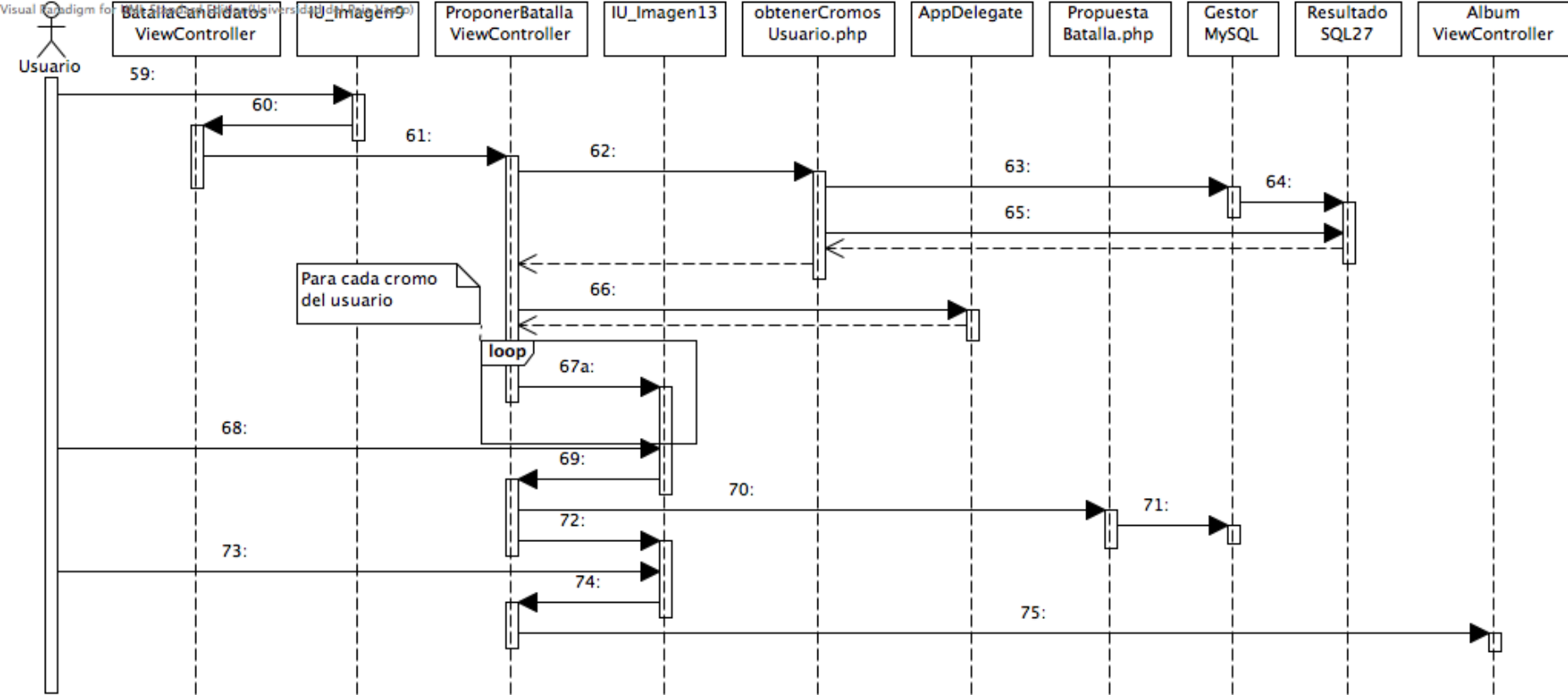
```
INSERT INTO batalla ( fechaHora, usuarioRetador, usuarioRetado, tipo, estado,
codColeccion )
VALUES ( '$fechaHora', '$usuarioRetador', '$usuarioRetado', '$tipo', 'pendiente',
'$codColeccion' );
```

72. mostrar alerta

73. El usuario pulsa el botón 'Menú mi colección'

74. clickedButtonAtIndex:(NSInteger) buttonIndex

75. setDatos(...)



Obtener cromos de un usuario (tras batalla)

1. setDatos(...)
 - [Si es una batalla selectiva]
 - 2a. El usuario selecciona un cromo
 - [Si es una batalla aleatoria]
 - 2b. El usuario pulsa el botón ' Pincha aquí para obtener un cromo aleatorio'
 - 3. didSelectItemAtIndexPath:(NSIndexPath *)indexPath
 - 4. insertarCromo.php?nombre=%@&codColeccion=%@&numero=%@
 - 5. execSQL29(sql29):
- // Se comprueba si se tiene el cromo

```
SELECT A.codCromo FROM cromosUsuario A, cromo_fam_coleccion B
WHERE A.codCromo = B.codCromo
AND A.nombre = '$nombre'
AND B.codColeccion = '$codColeccion'
AND B.numero = $numero
```

// Si no se tiene, se inserta

```
INSERT INTO cromosUsuario ( nombre, codColeccion, codCromo, nVeces)
SELECT '$nombre', '$codColeccion', codCromo, 1
FROM cromo_fam_coleccion
WHERE codColeccion = '$codColeccion' AND numero = '$numero'
```

// Si se tiene, hay que identificar el codCromo e incrementar en 1 el campo nVeces.

1. Del resultado de la primera select, \$rs, se obtiene el campo codCromo de la fila 0

```
$codCromo = mysql_result($rs,0,'codCromo');
```

2. Se obtiene el numero de veces que el usuario tiene ese cromo:

```
SELECT nVeces FROM cromosUsuario WHERE nombre = '$nombre'
AND codColeccion = '$codColeccion' AND codCromo = '$codCromo'
```

3. Se obtiene el numero de veces de la select y actualiza la tabla:

```

$nVeces = mysql_result($select,0,'nVeces');

UPDATE cromosUsuario SET nVeces = '$nVeces' + 1
WHERE nombre = '$nombre'
AND codColeccion = '$codColeccion'
AND codCromo = '$codCromo';

```

6. obtenerUnCromo.php?codColeccion=%@&numero=%d
7. execSQL30(sql30):

```

SELECT A.numero, B.codCromo, B.titulo, B.descripcion, C.codFamilia, C.descripcion
desFamilia, A.imagen FROM cromos_fam_coleccion A, cromos B, familias C
WHERE A.codCromo = B.codCromo AND A.codFamilia = C.codFamilia
AND A.codColeccion = '$codColeccion' AND A.numero = '$numero'
AND codCromo = '$codCromo';

```

8. new()
9. getDatos() : NSData
10. empezarAParsear():NSDictionary
11. [bd existeCromo: codCromo codColeccion:codColeccion] : boolean
- [Si no existe cromos]
- 12a. [agregarCromo:datosCromo]
- 13a. execSQL31(sql31)

```

INSERT INTO cromos (codCromo, titulo, descripcion)
VALUES ('codCromo', 'titulo', 'descripcion');

INSERT INTO cromos_fam_coleccion (codCromo, codFamilia, codColeccion, imagen,
numero, numeroVeces)
VALUES ('codCromo', 'codFamilia', 'codColeccion', 'rutaImagen', 'numero', 'nVeces')

```

- [Si existe cromos]
- 12b. [actualizarCromo:datosCromo]
- 13b. execSQL32(sql32)

```

UPDATE cromos_fam_coleccion
SET numeroVeces = (SELECT numeroVeces FROM cromos_fam_coleccion
WHERE codColeccion = 'codColeccion' AND codCromo = 'codCromo') + 1
WHERE codColeccion = 'codColeccion' AND codCromo = 'codCromo'

```

14. eliminarCromo.php?nombre=%@&codColeccion=%@&numero=%@
15. execSQL33(sql33):

- Primero se comprueba que el cromó existe

```
$rs = mysql_query("SELECT A.codCromo, A.nVeces
FROM cromosUsuario A, cromos_fam_coleccion B
WHERE A.codCromo = B.codCromo
AND A.nombre = '$nombre'
AND B.codColeccion = '$codColeccion'
AND B.numero = '$numero'") or die(mysql_error());
```

```
if (mysql_numrows($rs) == 0){
    echo "Ese cromó no existe";
}
```

- Si existe se comprueba si se tiene una vez o más veces.

```
else{
```

```
$nVeces = mysql_result($rs,0,'nVeces');
```

- Si se tiene una vez, entonces se elimina directamente

```
if($nVeces == 1){
    $codCromo = mysql_result($rs,0,'codCromo');
    $delete = "DELETE FROM cromosUsuario
WHERE codColeccion = '$codColeccion'
AND codCromo = '$codCromo'
AND nombre = '$nombre'";
```

- Si se tiene más de una vez, entonces se disminuye el campo nVeces

```
else{
    $codCromo = mysql_result($rs,0,'codCromo');
    $update = "UPDATE cromosUsuario SET nVeces = '$nVeces' - 1
WHERE nombre = '$nombre'
AND codColeccion = '$codColeccion'
AND codCromo = '$codCromo'";
```

```
...
}}
```

16 .obtenerCromoUsuario.php?usuario=%@&codColeccion=%@&numero=%@

17 .execSQL34(sql34):

```

SELECT A.numero, B.codCromo, B.titulo, B.descripcion, C.codFamilia, C.descripcion
desFamilia, A.imagen, D.nVeces, D.bloqueado
FROM cromosUsuario A, cromos B, familias C, cromosUsuario D
WHERE A.codCromo = B.codCromo
AND A.codFamilia = C.codFamilia
AND A.codColeccion = D.codColeccion
AND A.codCromo = D.codCromo
AND A.numero = '$numero'
AND A.codColeccion = '$codColeccion'
AND D.nombre = '$usuario'

```

18. new()

19. getDatos(): NSData

[Si cromos no encontrados] -- significa que era su único cromos.

20a. empezarAParsear(): NSDictionary

21a. obtenerPropuestasUnCromo.php?usuarioSolicitado=%@&codColeccion=%@
&codCromoPedido=%@

22a. execSQL35(sql35):

```

SELECT A.usuarioSolicitante, A.codCromoOfrecido, B.bloqueado
FROM intercambios A, cromosUsuario B
WHERE A.usuarioSolicitante = B.nombre
AND A.codCromoOfrecido = B.codCromo
AND A.usuarioSolicitado = '$usuarioSolicitado'
AND A.codCromoPedido = '$codCromoPedido'
AND A.codColeccion = '$codColeccion'
AND A.estado = 'pendiente'

```

23a. new()

24a. getDatos(): NSData

25a. empezarAParsear(): NSDictionary.

[Por cada solicitante que hizo una oferta por ese cromos]

26aa. desbloquearCromoUsuario.php?usuario=%@&codColeccion=%@
&codCromo=%@

27aa. execSQL36(sql36):


```
$select = mysql_query("SELECT bloqueado
FROM cromosUsuario
WHERE nombre = '$nombre'
AND codColeccion = '$codColeccion'
AND codCromo = '$codCromo'") or die(mysql_error());

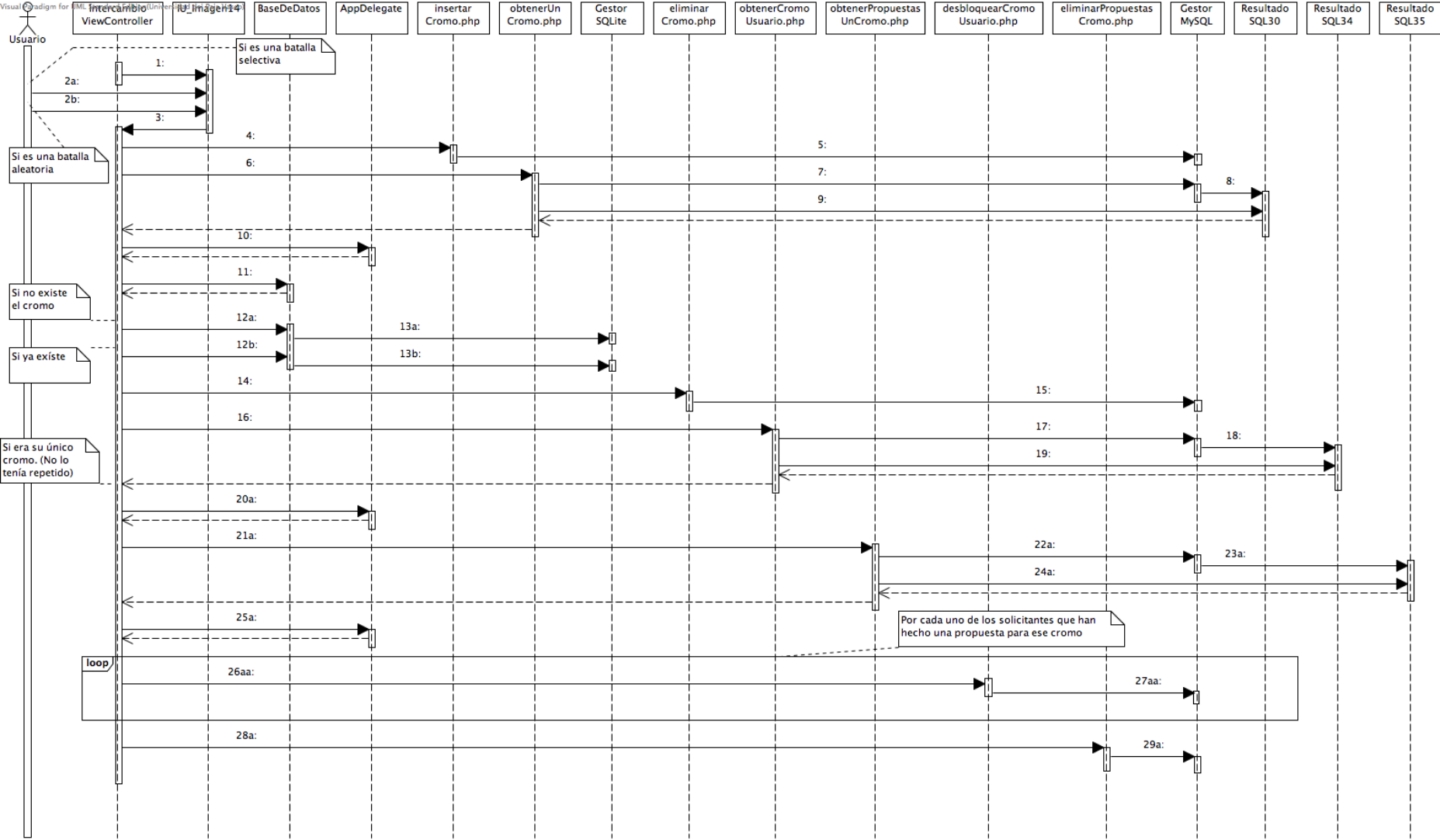
$nBloqueos = mysql_result($select,0,'bloqueado');

$update = "UPDATE cromosUsuario SET bloqueado = '$nBloqueos' - 1
WHERE nombre = '$nombre'
AND codCromo = '$codCromo'
AND codColeccion = '$codColeccion'";
```

28a. eliminarPropuestasCromo.php?codColeccion=%@&usuario=%@&codCromo=%@
29a. execSQL37(sql37):

```
UPDATE intercambio SET estado = 'anulada'
WHERE estado = 'pendiente'
AND usuarioSolicitante = '$usuario'
AND codCromoOfrecido = '$codCromo';

UPDATE intercambio SET estado = 'rechazada'
WHERE estado = 'pendiente'
AND usuarioSolicitado = '$usuario'
AND codCromoPedido = '$codCromo';
```



Intercambio

1. obtenerUsuariosColeccion.php?codColeccion=%@&usuario=%@
2. exeSQL38(sql38):

```
SELECT DISTINCT A.nombre FROM usuario A, colecciones_usuario B, cromosUsuario C
WHERE A.nombre = B.nombre AND A.nombre = C.nombre AND B.codColeccion = C.codColeccion
AND A.nombre <> '$usuario' AND B.codColeccion = '$codColeccion'
```

3. new()
4. getDatos() : NSData
5. empezarAParsear: NSDictionary

– Propuestas de intercambio recibidas

6. obtenerPropuestasIntercambio.php?codColeccion=%@&usuario=%@
7. execSQL39(sql39):

```
SELECT DISTINCT A.fechaHora, A.usuarioSolicitante, A.codCromoOfrecido,
A.codCromoPedido, A.codColeccion, B.imagen imagenOfrecido, C.imagen
imagenPedido, B.numero numeroOfrecido, C.numero numeroPedido
FROM intercambio A, cromos_usuario B, cromos_usuario C
WHERE A.codCromoOfrecido = B.codCromo
AND A.codCromoPedido = C.codCromo
AND A.usuarioSolicitado = '$usuario'
AND A.codColeccion = '$codColeccion'
AND A.estado = 'pendiente'
```

8. new()
9. getDatos():NSData
 - [Si ha recibido propuestas]
 - 10a. empezarAParsear:NSDictionary
 - 11a. mostrar botPropuestasRecibidas
 - 12a. mostrar alerta
 - 13a. El usuario selecciona algo
 - [Si el usuario selecciona 'Ver Propuestas']
 - 14aa. clickedButtonAtIndex(buttonIndex) // Se pasa el índice del botón seleccionado
 - 15aa.setDatos(propuestasRecibidas)
 - [Para cada propuesta recibida]
 - 16aaa. Se cargan los datos en la celda correspondiente del tableview
 - 17aa. El usuario pulsa un botón dentro de una celda de propuesta
 - [Si el usuario selecciona 'Rechazar']
 - 18aaa.rechazarIntercambio: (UIButton*) sender
 - 19aaa. rechazarIntercambio.php?fecha=%@&hora=%@&usuarioSolicitante=%@&usuarioSolicitado=%@&codColeccion=%@&codCromoOfrecido=%@
 - 20aaa. execSQL40(sql40):

```
UPDATE intercambio SET estado = 'rechazada'
WHERE fechaHora = '$fechaHora' AND usuarioSolicitante = '$usuarioSolicitante'
AND usuarioSolicitado = '$usuarioSolicitado' AND codColeccion = '$codColeccion';
```

// Proceso para quitar el bloqueo del cromó propuesto

```
$select = mysql_query("SELECT bloqueado FROM cromosUsuario
WHERE nombre = '$usuarioSolicitante'
AND codColeccion = '$codColeccion'
AND codCromo = '$codCromoOfrecido'")
or die(mysql_error());

$nbloqueos = mysql_result($select,0,'bloqueado');

$update2 = " UPDATE cromosUsuario SET bloqueado = '$nbloqueos' - 1
WHERE nombre = '$usuarioSolicitante'
AND codCromo = '$codCromoOfrecido'
```

```
21aaa. [propuestasRecibidas removeObjectAtIndex:sender.tag];
22aaa. [tableView reloadData];
[Si el usuario selecciona 'Aceptar']
18aab. intercambiar: (UIButton*) sender
19aab. responderPropuesta.php?usuarioSolicitante=%@&usuarioSolicitado=%@
&fechaHora=%@&codColeccion=%@&respuesta=aceptada
&codCromoOfrecido=%@
20aab. execSQL41(sql41):
```

```
UPDATE intercambio SET estado = '$estado'
WHERE fechaHora = '$fechaHora' AND usuarioSolicitante = '$usuarioSolicitante'
AND usuarioSolicitado = '$usuarioSolicitado' AND codColeccion = '$codColeccion';
```

// Proceso para quitar el bloqueo del cromó propuesto

```
$select = mysql_query("SELECT bloqueado FROM cromosUsuario
WHERE nombre = '$usuarioSolicitante'
AND codColeccion = '$codColeccion'
AND codCromo = '$codCromoOfrecido'")
or die(mysql_error());

$nbloqueos = mysql_result($select,0,'bloqueado');

$update2 = " UPDATE cromosUsuario SET bloqueado = '$nbloqueos' - 1
WHERE nombre = '$usuarioSolicitante'
AND codCromo = '$codCromoOfrecido'
AND codColeccion = '$codColeccion';
```

- 21aab. [propuestasRecibidas removeObjectAtIndex:sender.tag];
- 22aab. Proceso de intercambio de cromos, representado aparte en otro diagrama.
- 23aab. setData(...)

– Propuestas de intercambio realizadas

- 24. obtenerPropuestasRealizadas.php?codColeccion=%@&usuario=%@
- 25. execSQL42(sql42):

```
SELECT DISTINCT A.fechaHora, A.usuarioSolicitado, A.codCromoOfrecido,
A.codCromoPedido, A.codColeccion, B.imagen imagenOfrecido, C.imagen imagenPedido,
B.numero numeroOfrecido, C.numero numeroPedido
FROM intercambio A, cromos_fam_coleccion B,cromos_fam_coleccion C
WHERE A.codCromoOfrecido = B.codCromo AND A.codCromoPedido = C.codCromo
AND A.usuarioSolicitante = '$usuario' AND A.codColeccion = '$codColeccion'
AND A.estado = 'pendiente'
```

- 26. new()
- 27. getData(): NSData
- [Si ha realizado alguna propuesta de intercambio]
- 28a. empezarAParsear: NSDictionary
- 29a. mostrar btnPropuestasRealizadas
- 30a. El usuario pulsa el botón 'Realizadas'
- 31a. mostrarProRealizadas:(id)sender
- 32a. setData(propuestasRealizadas)
- [Para cada propuesta realizada]
- 33aa. Se cargan los datos en la celda correspondiente del tableView
- 34a. El usuario pulsa un botón dentro de una celda de propuesta
- [Si el usuario selecciona 'Anular']
- 35aa. anularIntercambio: (UIButton*) sender
- 36aa. anularIntercambio.php?fecha=%@&hora=%@&usuarioSolicitante=%@&usuarioSolicitado=%@&codColeccion=%@&codCromoOfrecido=%@
- 37aa. execSQL43(sql43):

```
UPDATE intercambio SET estado = 'anulada'
WHERE fechaHora = '$fechaHora' AND usuarioSolicitante = '$usuarioSolicitante'
AND usuarioSolicitado = '$usuarioSolicitado'
```

// Proceso para quitar el bloqueo del cromo propuesto

```

$select = mysql_query("SELECT bloqueado FROM cromosUsuario
WHERE nombre = '$usuarioSolicitante'
AND codColeccion = '$codColeccion'
AND codCromo = '$codCromoOfrecido'")
or die(mysql_error());

$nbloqueos = mysql_result($select,0,'bloqueado');

$update2 = " UPDATE cromosUsuario SET bloqueado = '$nbloqueos' - 1
WHERE nombre = '$usuarioSolicitante'
AND codCromo = '$codCromoOfrecido'
AND codColeccion = '$codColeccion'";

```

38aa. [propuestasRealizadas removeObjectAtIndex:sender.tag];
39aa. [tableView reloadData];

– Proponer intercambio

40. El usuario selecciona un usuario de la lista.
41. didSelectRowAtIndexPath:(NSIndexPath *)indexPath
42. setDatos(...)
43. obtenerCromosUsuario.php?codColeccion=%@&usuario=%@
44. execSQL44(sql44):

```

SELECT A.numero, B.codCromo, B.titulo, B.descripcion, C.codFamilia,
C.descripcion desFamilia, A.imagen, D.nVeces, D.bloqueado
FROM cromo_fam_coleccion A, cromo B, familia C, cromosUsuario D
WHERE A.codCromo = B.codCromo AND A.codFamilia = C.codFamilia
AND A.codColeccion = D.codColeccion AND A.codCromo = D.codCromo
AND D.nombre = '$usuario' AND A.codColeccion = '$codColeccion'

```

45. new()
46. getDatos(): NSData
47. empezarAParsear():NSDictionary
[Para cada cromo del usuario al que se le pide un cromo]
- 48a. cargar datos cromo en vista.
49. Al no ser batalla... El usuario selecciona el cromo que le interesa y no este bloqueado
50. didSelectItemAtIndexPath:(NSIndexPath *)indexPath
51. setDatos(...)
52. obtenerCromosUsuario.php?codColeccion=%@&usuario=%@
53. execSQL44(sql44):
54. new()
55. getDatos
56. empezarAParsear():NSDictionary
[Para cada cromo del usuario que propone el intercambio]
- 57a. cargar datos cromo en vista.
58. El usuario selecciona un cromo a modo de propuesta que no este bloqueado.

59. propuestaIntercambio.php?usuarioSolicitante=%@&usuarioSolicitado=%@&codCromoOfrecido=%@&codCromoPedido=%@&codColeccion=%@

```
INSERT INTO intercambio ( fechaHora, usuarioSolicitante, usuarioSolicitado, codCromoOfrecido,
codCromoPedido,
codColeccion, estado )
VALUES
( '$fechaHora', '$usuarioSolicitante', '$usuarioSolicitado', '$codCromoOfrecido',
'$codCromoPedido', '$codColeccion', 'pendiente' )
```

// Proceso para añadir un bloqueo al cromos propuesto

```
$select = mysql_query("SELECT bloqueado FROM cromosUsuario
WHERE nombre = '$usuarioSolicitante'
AND codColeccion = '$codColeccion'
AND codCromo = '$codCromoOfrecido'")
or die(mysql_error());

$nbloqueos = mysql_result($select,0,'bloqueado');

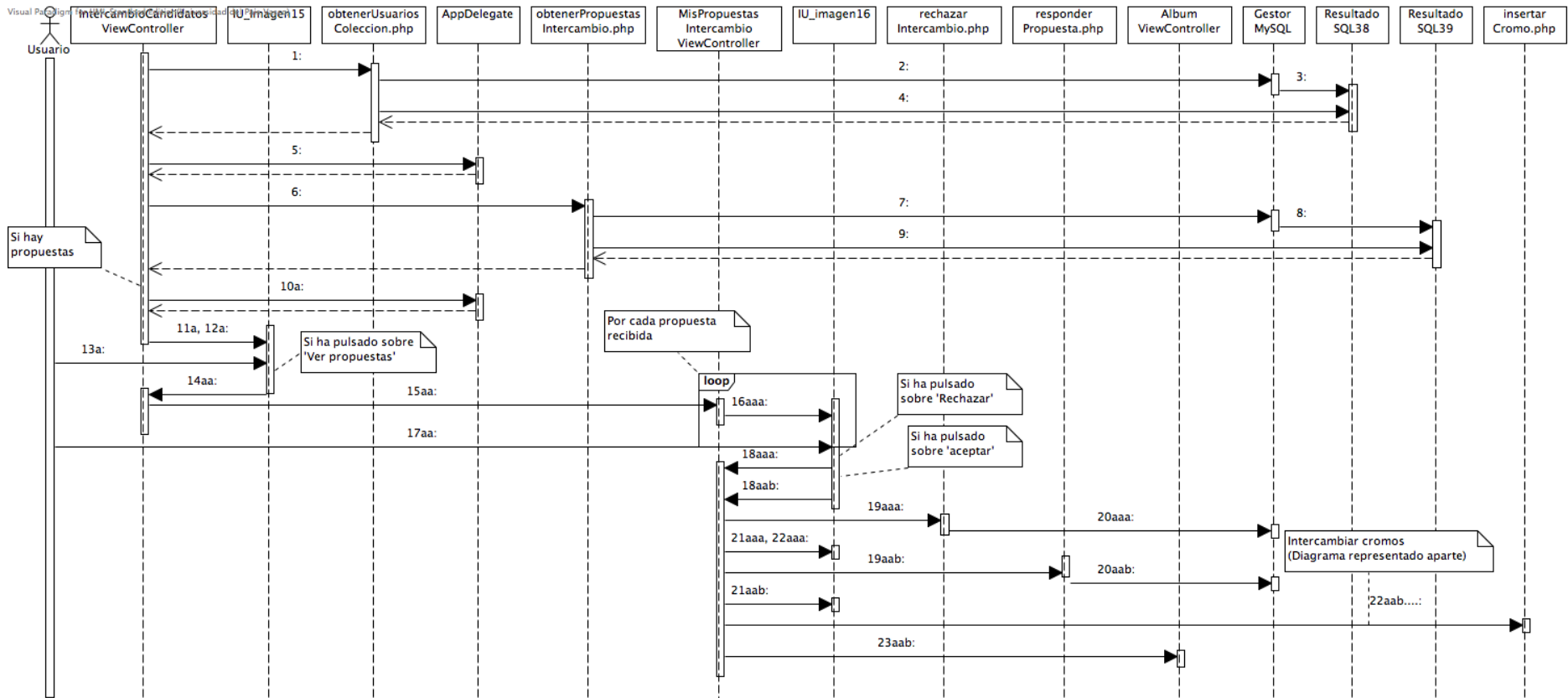
$update2 = " UPDATE cromosUsuario SET bloqueado = '$nbloqueos' +1
WHERE nombre = '$usuarioSolicitante'
AND codCromo = '$codCromoOfrecido'
AND codColeccion = '$codColeccion'";
```

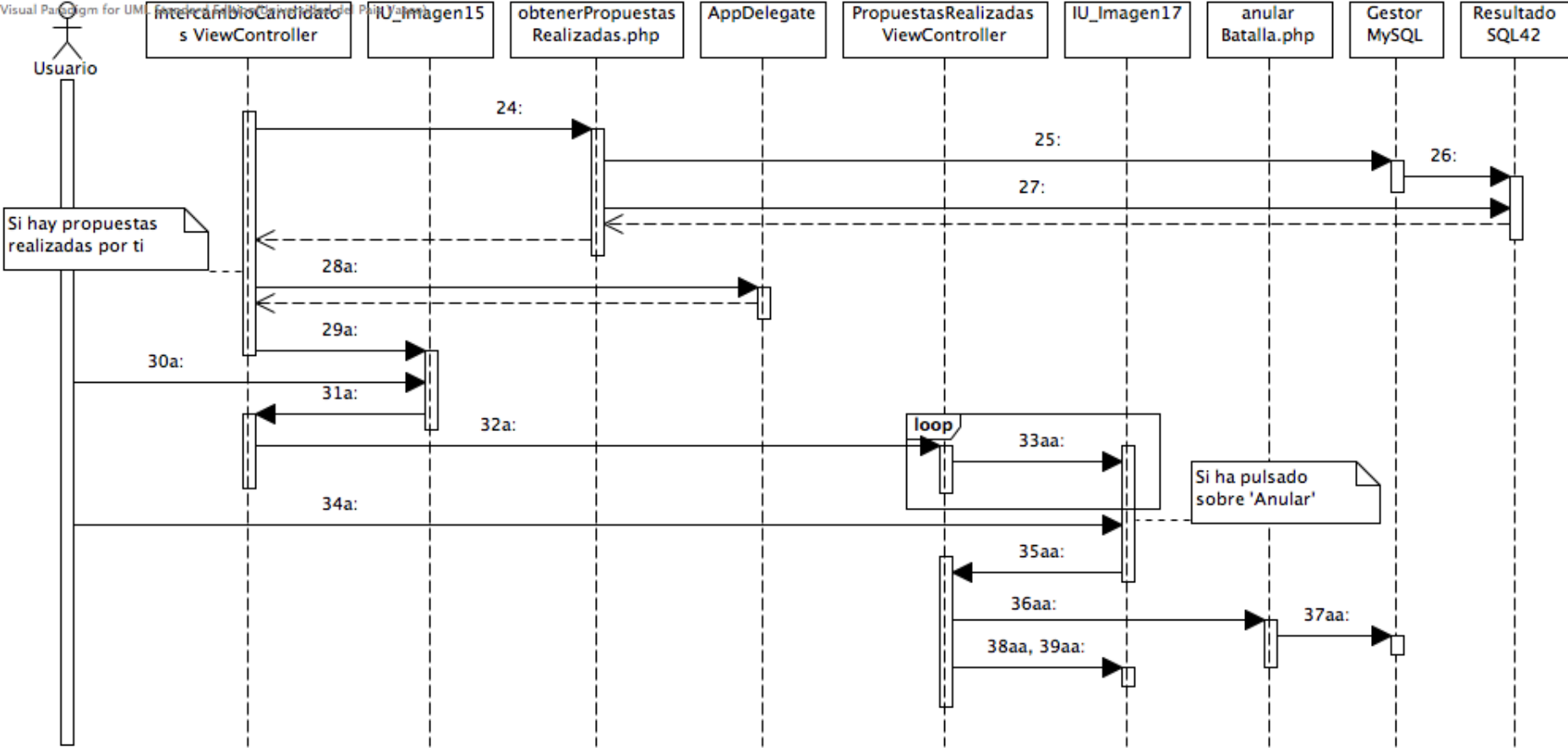
60. mostrar alerta

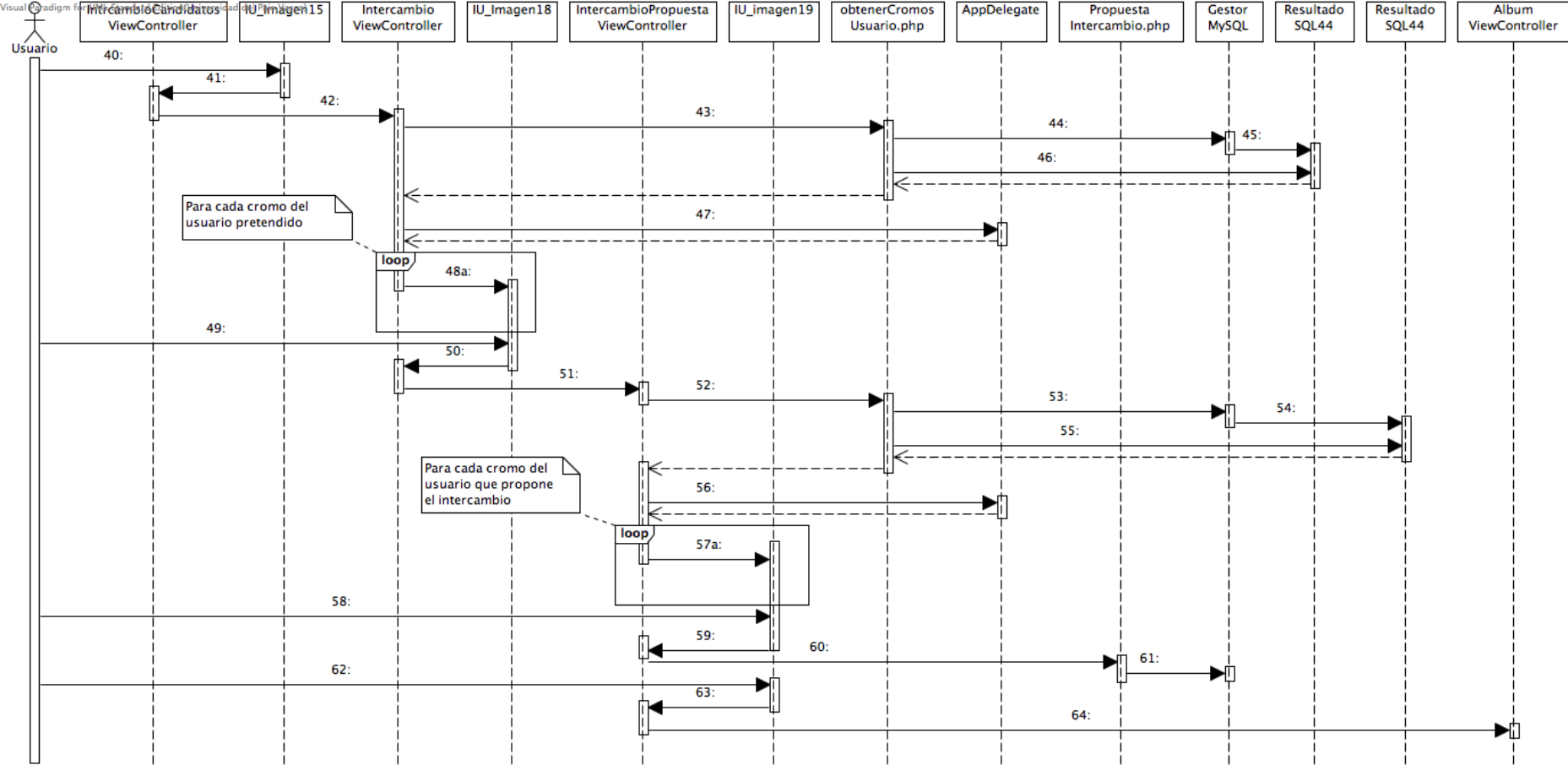
61. El usuario pulsa el botón 'Menú mi colección'

62. clickedButtonAtIndex:(NSInteger) buttonIndex

63. setDatos(...)







Proceso de intercambio de cromos

Partimos desde el punto 22aab del diagrama Intercambio.

[Para cada usuario que forma parte del intercambio]

1. insertarCromo.php?nombre=%@&codColeccion=%@&numero=%@
2. execSQL45(sql45):
// Se comprueba si se tiene el cromo

```
SELECT A.codCromo FROM cromosUsuario A, cromo_fam_coleccion B
WHERE A.codCromo = B.codCromo
AND A.nombre = '$nombre'
AND B.codColeccion = '$codColeccion'
AND B.numero = $numero
```

// Si no se tiene, se inserta

```
INSERT INTO cromosUsuario ( nombre, codColeccion, codCromo, nVeces)
SELECT '$nombre', '$codColeccion', codCromo, 1
FROM cromo_fam_coleccion
WHERE codColeccion = '$codColeccion' AND numero = '$numero'
```

// Si se tiene, hay que identificar el codCromo e incrementar en 1 el campo nVeces.

4. Del resultado de la primera select, \$rs, se obtiene el campo codCromo de la fila 0

```
$codCromo = mysql_result($rs,0,'codCromo');
```

5. Se obtiene el numero de veces que el usuario tiene ese cromo:

```
SELECT nVeces FROM cromosUsuario WHERE nombre = '$nombre'
AND codColeccion = '$codColeccion' AND codCromo = '$codCromo'
```

6. Se obtiene el numero de veces de la select y actualiza la tabla:

```
$nVeces = mysql_result($select,0,'nVeces');

UPDATE cromosUsuario SET nVeces = '$nVeces' + 1
WHERE nombre = '$nombre'
AND codColeccion = '$codColeccion'
AND codCromo = '$codCromo';
```

3. obtenerUnCromo.php?codColeccion=%@&numero=%d

4. execSQL46(sql46):

```
SELECT A.numero, B.codCromo, B.titulo, B.descripcion, C.codFamilia, C.descripcion
desFamilia, A.imagen FROM cromos_usuario A, cromos B, familias C
WHERE A.codCromo = B.codCromo AND A.codFamilia = C.codFamilia
AND A.codColeccion = '$codColeccion' AND A.numero = '$numero'
AND B.codCromo = '$codCromo';
```

5. new()

6. getDatos() : NSData

7. empezarAParsear():NSDictionary

8. [bd existeCromo: codCromo codColeccion:codColeccion] : boolean

[Si no existe cromos]

9a. [agregarCromo:datosCromo]

10a. execSQL47(sql47)

```
INSERT INTO cromos (codCromo, titulo, descripcion)
VALUES ('codCromo', 'titulo', 'descripcion');

INSERT INTO cromos_usuario (codCromo, codFamilia, codColeccion, imagen,
numero, numeroVeces)
VALUES ('codCromo', 'codFamilia', 'codColeccion', 'rutaImagen', 'numero', 'nVeces')
```

[Si existe cromos]

9b. [actualizarCromo:datosCromo]

10b. execSQL48(sql48)

```
UPDATE cromos_usuario
SET numeroVeces = (SELECT numeroVeces FROM cromos_usuario
WHERE codColeccion = 'codColeccion' AND codCromo = 'codCromo') + 1
WHERE codColeccion = 'codColeccion' AND codCromo = 'codCromo'
```

11. eliminarCromo.php?nombre=%@&codColeccion=%@&numero=%@

12. execSQL(sql):

■ Primero se comprueba que el cromos existe

```
$rs = mysql_query("SELECT A.codCromo, A.nVeces
FROM cromos_usuario A, cromos_usuario B
WHERE A.codCromo = B.codCromo
AND A.nombre = '$nombre'
AND B.codColeccion = '$codColeccion'
AND B.numero = '$numero'") or die(mysql_error());
```

```
if (mysql_numrows($rs) == 0){
    echo "Ese cromó no existe";
}

■ Si existe se comprueba si se tiene una vez o mas veces.

else{

$nVeces = mysql_result($rs,0,'nVeces');

    ■ Si se tiene una vez, entonces se elimina directamente

if($nVeces == 1){
    $codCromo = mysql_result($rs,0,'codCromo');
    $delete = "DELETE FROM cromosUsuario
    WHERE codColeccion = '$codColeccion'
    AND codCromo = '$codCromo'
    AND nombre = '$nombre'";

    ■ Si se tiene mas de una vez, entonces se disminuye el campo nVeces

else{
    $codCromo = mysql_result($rs,0,'codCromo');
    $update = "UPDATE cromosUsuario SET nVeces = '$nVeces' -1
    WHERE nombre = '$nombre'
    AND codColeccion = '$codColeccion'
    AND codCromo = '$codCromo'";
    ...
}}
```

13. obtenerCromoUsuario.php?usuario=%@&codColeccion=%@&numero=%@

14 .execSQL49(sql49):

```

SELECT A.numero, B.codCromo, B.titulo, B.descripcion, C.codFamilia, C.descripcion
desFamilia, A.imagen, D.nVeces, D.bloqueado
FROM cromosUsuario D, cromosFamilia C, cromosColeccion B, cromosFamilia A
WHERE A.codCromo = B.codCromo
AND A.codFamilia = C.codFamilia
AND A.codColeccion = D.codColeccion
AND A.codCromo = D.codCromo
AND A.numero = '$numero'
AND A.codColeccion = '$codColeccion'
AND D.nombre = '$usuario'

```

15. new()

16. getDatos(): NSData

[Si cromos no encontrados] -- significa que era su único cromos.

17a. empezarAParsear(): NSDictionary

18a. obtenerPropuestasUnCromo.php?usuarioSolicitado=%@&codColeccion=%@
&codCromoPedido=%@

19a. execSQL50(sql50):

```

SELECT A.usuarioSolicitante, A.codCromoOfrecido, B.bloqueado
FROM intercambio A, cromosUsuario B
WHERE A.usuarioSolicitante = B.nombre
AND A.codCromoOfrecido = B.codCromo
AND A.usuarioSolicitado = '$usuarioSolicitado'
AND A.codCromoPedido = '$codCromoPedido'
AND A.codColeccion = '$codColeccion'
AND A.estado = 'pendiente'

```

20a. new()

21a. getDatos(): NSData

22a. empezarAParsear(): NSDictionary.

[Por cada solicitante que hizo una oferta por ese cromos]

23aa. desbloquearCromoUsuario.php?usuario=%@&codColeccion=%@
&codCromo=%@

24aa. execSQL51(sql51):

```
$select = mysql_query("SELECT bloqueado
FROM cromosUsuario
WHERE nombre = '$nombre'
AND codColeccion = '$codColeccion'
AND codCromo = '$codCromo'") or die(mysql_error());

$nBloqueos = mysql_result($select,0,'bloqueado');

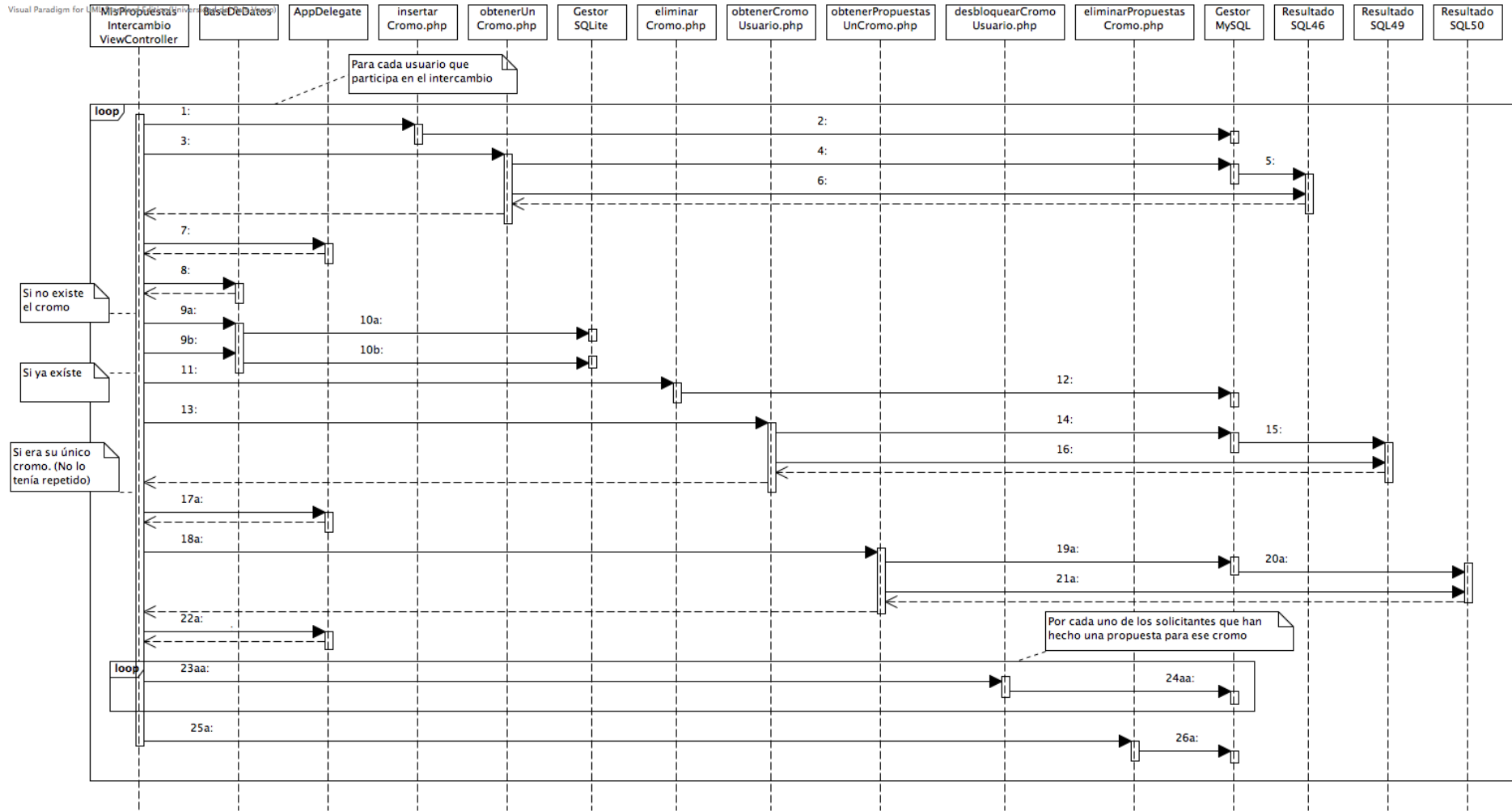
$update = "UPDATE cromosUsuario SET bloqueado = '$nBloqueos' - 1
WHERE nombre = '$nombre'
AND codCromo = '$codCromo'
AND codColeccion = '$codColeccion'";
```

25a. eliminarPropuestasCromo.php?codColeccion=%@&usuario=%@&codCromo=%@

26a. execSQL52(sql52):

```
UPDATE intercambio SET estado = 'anulada'
WHERE estado = 'pendiente'
AND usuarioSolicitante = '$usuario'
AND codCromoOfrecido = '$codCromo';

UPDATE intercambio SET estado = 'rechazada'
WHERE estado = 'pendiente'
AND usuarioSolicitado = '$usuario'
AND codCromoPedido = '$codCromo';
```



Buscar tesoro

1. MKCoordinateRegion region = [self.mapView setRegion:
MKCoordinateRegionMakeWithDistance(coordenadasUsuario,2000,2000);animated:YES];
2. [locationManager startUpdatingLocation];
3. mostrarTesoro
[Mientras el usuario este a mas de 150 metros de distancia]
- 4a didUpdateToLocation:(CLLocation *)newLocation fromLocation:(CLLocation *)
5. mostrar alerta
6. El usuario pulsa en 'Sobre'
7. clickedButtonAtIndex:(NSInteger) buttonIndex
8. setDatos(...)

