UNIVERSIDAD REY JUAN CARLOS

LICENCIATURA EN INGENIERÍA SUPERIOR EN INFORMÁTICA


**PROYECTO FIN DE CARRERA**

# ..TÍTULO PROYECTO..

AUTOR: LÓPEZ CEREZO, ALEJANDRO M.

TUTOR: FERNÁNDEZ GIL, ALBERTO

# Agradecimientos

Tras ser capaz de desarrollar mis estudios universitarios de un modo satisfactorio, llegué al último paso: el Proyecto Fin de Carrera.

La presente memoria hace referencia a, por diferentes motivos, el tercer proyecto que realizo, quedando su desarrollo y elaboración compaginada con mi actividad laboral; circunstancia que ha implicado numerosas dificultades ligadas fundamentalmente a la falta de tiempo. Por ello, su finalización no hubiese sido posible sin el apoyo y ayuda de una serie de personas que me gustaría tener en cuenta en este punto final.

A mi tutor, Alberto Fernández, por haberme dado la oportunidad de desarrollar el proyecto bajo su supervisión cuando, después de una mala experiencia previa, me quedé sin ninguno asignado.

A mis padres, mi hermano y mi novia, Ana, por su comprensión y apoyo constante durante el proceso y, sobre todo, en esos fines de semana de dedicación interminables en los que no estaba para nada ni nadie después de toda la semana de trabajo.

A todos, gracias.

# Resumen

El presente proyecto ha supuesto el desarrollo de una aplicación Android completa para la gestión de parques de bicicletas públicos, estableciendo una infraestructura tecnológica base para desarrollos e investigaciones futuras.

Aplicaciones como la presentada cobran gran importancia en la actualidad dados dos factores básicos: por un lado, la utilización, cada vez mayor, de la bicicleta pública como alternativa a los medios de transporte tradicionales; por otro lado, la presencia de los dispositivos móviles como herramientas de gestión de tareas diarias, como podría ser la reserva de un bicicleta. Así, se hace necesario contar con aplicaciones y sistemas eficientes, eficaces y fácilmente utilizables para la gestión de dichos parques públicos.

La herramienta presenta un mapa actualizado de las estaciones disponibles sobre el que poder operar, así como pantallas adicionales para facilitar la interacción del usuario con la aplicación a la hora de gestionar su perfil o conocer el estado del parque de bicicletas.

Desde un punto de vista técnico, la aplicación sigue una arquitectura cliente-servidor de tres capas, estructurándose la comunicación entre el ambos mediante el estándar REST de los servicios web.

El desarrollo ha seguido un modelo de proceso incremental bajo un paradigma orientado a objetos, tratando de tener en todo momento en mente las buenas prácticas establecidas por la Ingeniería del Software, las *Reglas de Oro* para el diseño de interfaces de usuario, recomendaciones de diseño e implementación Android, etc. Dado el modelo señalado, el proceso se ha dividido en una serie de incrementos jerarquizados por importancia y desarrollados de manera individual para, finalmente, quedar integrados en el producto final.

La aplicación ha sido probada mediante pruebas planificadas de unidad, de integración y de validación para tratar de lograr una elevada cobertura de errores y calidad final.

Finalmente, señalar que como se ha indicado al principio de este resumen, la herramienta presentada supone una infraestructura base para investigaciones futuras que conduzcan a modelos de gestión más eficientes y con mayor capacidad de atracción de usuarios que los utilizados actualmente.

# Índice general

# Índice de figuras

# Índice de tablas

# Capítulo 1

# Introducción

## 1.1. Motivación

Los parques públicos de bicicletas están cada vez más extendidos en las ciudades como alternativa a los medios de transporte tradicionales. Asimismo, los dispositivos móviles representan, para gran parte de la población, la herramienta básica para el manejo de numerosas tareas diarias.

Ambos factores suponen la motivación básica para el desarrollo de una aplicación de gestión de dichos parques, de manera que los usuarios registrados puedan conocer y operar con las estaciones disponibles y se logre una experiencia de usuario positiva al mismo tiempo que se realice una gestión eficiente del parque.

El presente proyecto surge con la idea de suponer una infraestructura tecnológica base para el desarrollo e investigaciones futuras sobre el área que mejoren la gestión del conjunto de estaciones.

## 1.2. Metodología

El modelo de proceso seguido ha sido el *modelo incremental*, que combina elementos de los flujos de proceso lineal y paralelo. Como se puede observar en la figura 1.1, el modelo aplica secuencias lineales en forma escalonada a medida que avanza el calendario de actividades. Cada secuencia lineal produce *incrementos* de software susceptibles de entregarse de manera parecida a los incrementos producidos en un flujo de proceso evolutivo.

Cuando se utiliza un modelo incremental, es frecuente que el primer incremento sea el *producto fundamental*. Es decir, se abordan los requerimientos básicos, pero no se proporcionan muchas características suplementarias (algunas conocidas y otras no). Como resultado del uso y/o evaluación de este primer producto, se desarrolla un plan para el incremento que sigue. El plan incluye la modificación

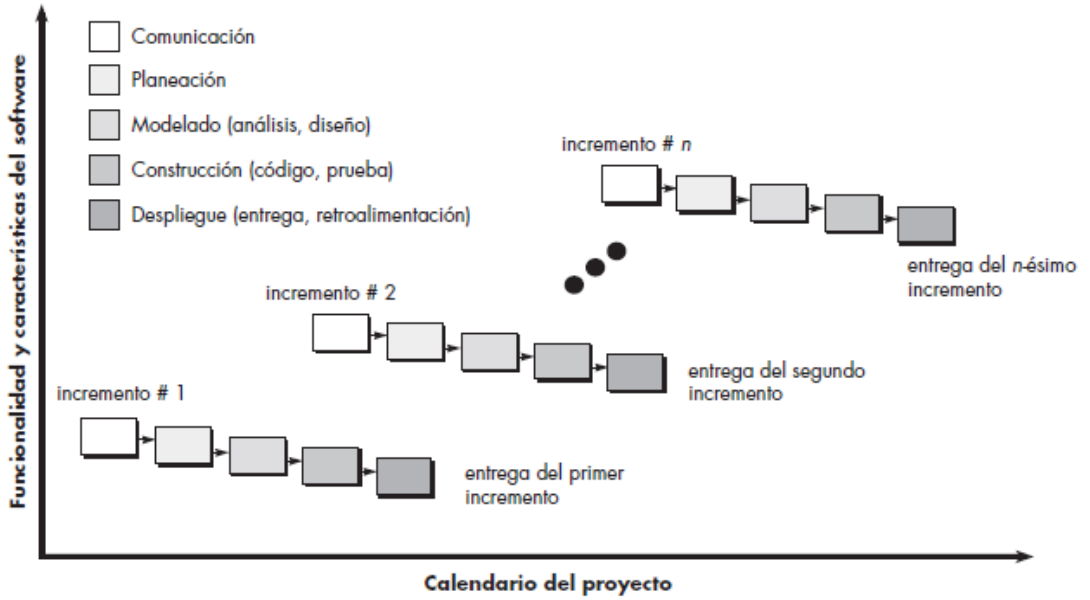


Figura 1.1: El modelo incremental *Fuente: [Pre10]*

del producto fundamental para cumplir mejor las nuevas necesidades, así como la entrega de características adicionales y más funcionalidad. Este proceso se repite después de entregar cada incremento, hasta terminar el producto final.

El modelo de proceso incremental se centra en que en cada incremento se entrega un producto que ya opera. Los primeros incrementos son versiones desnudas del producto final, pero proporcionan capacidad que sirve al usuario y también le dan una plataforma de evaluación [Pre10].

Considerando las características anteriores, los incrementos generales establecidos han sido los siguientes:

1. Mapa con estaciones sobre las que poder operar con datos locales.
2. Lista con el detalle de las estaciones mostradas en el mapa.
3. Gráficos con el estado del parque de estaciones.
4. Desarrollo e integración con la aplicación del Servidor y base de datos.
5. Sistema gestor de usuarios.
6. Sistema de gestión de reservas ligadas al usuario.

# Capítulo 2

# Objetivos

El objetivo primario del presente proyecto ha sido el desarrollo de una aplicación móvil para la gestión de un parque público de bicicletas, de manera desglosada:

- Desarrollar una aplicación Android comunicada con un servidor y base de datos mediante servicios web que implemente un sistema de gestión de bicicletas que permita coger, dejar o reservar bicicletas y anclajes a usuarios registrados.
- Introducir un primer nivel de adaptación dinámica de precios dependiendo de la disponibilidad individual de cada estación de bicicletas.
- Trasladar, en la medida de los posible, la lógica de las operaciones al servidor, haciendo más eficiente la aplicación instalada.
- Seguir las recomendaciones generales de diseño e implementación procedentes de fuentes especializadas para asegurar una adecuada experiencia de usuario.
- Evitar condiciones de carrera ocasionadas como consecuencia de accesos simultáneos a un mismo recurso.
- Cubrir el mayor volumen de dispositivos posible a nivel tecnológico (versiones Android), lingüístico (diferentes idiomas), etc.

# Capítulo 3

# Descripción informática

## 3.1. Especificación de Requisitos Software

### 3.1.1. Introducción

#### Propósito

El propósito de esta sección es el de presentar los requisitos de la aplicación acorde al estándar *IEEE Std. 830-1998: Especificación de Requisitos Software* (ERS en adelante), mostrando y esquematizando la funcionalidad básica del software a desarrollar.

El documento va principalmente dirigido a futuros usuarios y desarrolladores de la aplicación, de modo que cuenten con una aproximación teórica a la misma y a sus posibilidades.

#### Ámbito del Sistema

El futuro sistema, de nombre *Bikesmanager*, consistirá en una aplicación Android encargada de la gestión de parques públicos de bicicletas a través de usuarios previamente registrados.

Se buscará desarrollar una aplicación intuitiva y fácil de usar que cubra las necesidades de manera eficiente y eficaz, proporcionando una adecuada experiencia al usuario final.

#### Definiciones, Acrónimos y Abreviaturas

- ERS: Especificación de Requisitos Software.
- Bikesmanager: Nombre de la aplicación a desarrollar.

- Android: Sistema operativo diseñado principalmente para dispositivos móviles con pantalla táctil.
- Usuario: Persona que hará uso de la aplicación.
- RF: Requerimiento Funcional.
- RNF: Requerimineto No Funcional.

#### Referencias

- IEEE Std. 830-1998: Especificaciones de los Requisitos del Software [IE830].

#### Visión General del Documento

Una vez realizada la introducción general previa, se aportará a continuación una primera descripción general del sistema a desarrollar para, finalmente, pasar a detallar los requisitos específicos básicos del mismo.

En el apartado dedicado a la descripción general se aporta una visión global de la aplicación, así como de las funciones básicas de la misma. Funciones que se detallan en la sección siguiente, junto con otros aspectos como los atributos del sistema sobre los que se implantará el desarrollo.

### 3.1.2. Descripción General

#### Perspectiva del Producto

La aplicación *Bikesmanager* será un producto diseñado para trabajar sobre dispositivos móviles con sistema operativo Android, donde los datos quedarán almacenados en una base de datos a la que se accederá mediante el servidor de la aplicación. La conexión de la herramienta con el servidor se realizará mediante servicios web.

#### Funciones del Producto

En la figura 3.1 se aporta el diagrama de casos de uso que muestra, a grandes rasgos, las funciones del futuro sistema.

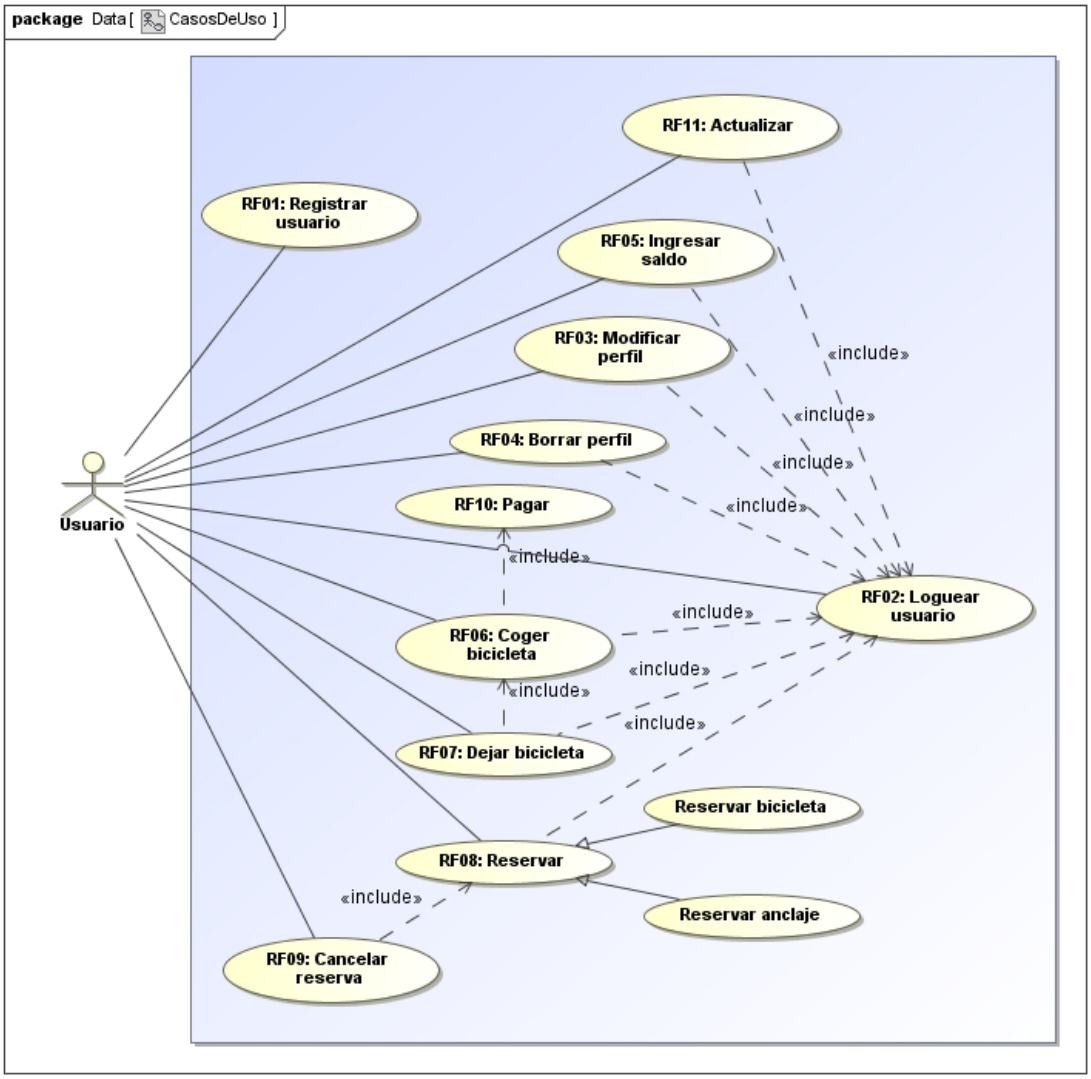


Figura 3.1: Diagrama de casos de uso

### Características de los Usuarios

- Tipo de usuario: Usuario

  - Nivel educacional: irrelevante.
  - Experiencia técnica: experiencia en el manejo de *smartphones*.
  - Actividad: manejo de la aplicación.

### Restricciones

- Limitaciones hardware:
  - Los servidores han de ser capaces de atender consultas concurrentes.
  - Los dispositivos móviles deberán estar gestionados por el sistema operativo Android.
- Arquitectura del sistema: cliente/servidor de tres capas, con servidor de aplicación GlassFish y SQL Server de base de datos.
- Lenguaje(s) en uso: JAVA, SQL.
- Protocolos de comunicación:
  - Aplicación – Servidor: HTTP (RESTful Web Services - JSON).
  - Servidor Aplicación – Base de Datos: MySQL Connector/JDBC.
- Consideraciones acerca de la seguridad:
  - El acceso a la aplicación se realizará mediante el par usuario-contraseña.
  - Las claves de usuario deberán almacenarse de manera segura mediante encriptación SHA-1.
  - Desde la aplicación ningún usuario tendrá acceso a la administración interna de la misma, sino que dicha tarea se realizará directamente sobre el servidor o base de datos.

### Suposiciones y Dependencias

- Se asume que los requisitos aquí descritos son estables.
- Los equipos en los que se vaya a ejecutar el sistema deben cumplir los requisitos antes indicados para garantizar el adecuado funcionamiento de la aplicación.

### Requisitos Futuros

No se consideran requisitos futuros.

### 3.1.3. Requisitos Específicos

#### Interfaces Externas

La interfaz con el usuario consistirá en un conjunto de pantallas con los controles habituales: botones, campos de texto, listas, etc. sobre la que se deberá asegurar una adecuada experiencia de usuario mediante diseños que sigan las normas Android [1]. Esta interfaz deberá ser construida para el sistema especificado y será visualizada mediante dispositivos móviles. Del mismo modo, destacar que el diseño de la interfaz gráfica de usuario deberá fundamentarse en los llamados *fragments* de Android, de modo que el mismo pueda ser reutilizable y fácilmente transportable a otras dimensiones u orientaciones de pantalla.

En relación a las interfaces hardware-software, se deberá contar con un dispositivo móvil con conexión a internet y que implemente, como mínimo, la versión 4.0 del sistema Android (llamada *Ice Cream Sandwich*[2]).

Finalmente, acerca de las interfaces de comunicación, la aplicación se comunicará con su servidor mediante el protocolo HTTP haciendo uso de RESTful Web Services y cadenas JSON. La conexión entre el servidor de aplicación y la base de datos se realizará mediante las tecnologías MySQL Connector/JDBC.

#### Funciones

En esta sección se desarrolla la descripción de los diferentes requerimientos funcionales y no funcionales con que deberá contar el sistema.

Se comienza por los *Requerimientos Funcionales*, para los que se aporta su definición acompañada de una descripción textual y del diagrama de actividad correspondiente, de modo que cada requerimiento quede adecuadamente descrito.

- **RF01: Registrar usuario**. El sistema deberá permitir el registro de nuevos usuarios. Para ello, se solicitarán el nombre usuario, contraseña, dirección de correo electrónico y nombre y apellidos reales. El nombre de usuario y la dirección de correo han de ser únicos, de modo que no se permita la duplicidad de los mismos en la aplicación. Asimismo, la contraseña ha de ser encriptada a la hora de ser almacenada en la base de datos.

  La descripción textual:

[1] Los detalles al respecto se pueden consultar en `https://developer.android.com/design/index.html`, a partir de [AnDev].

[2] Android denomina alfabéticamente y con nombres de dulces a cada una de las versiones de su sistema operativo. En el momento de la redacción de este documento, la versión más actualizada es la 7, *Nougat*.

| **RF01: Registrar usuario** | |
|---|---|
| **Usuario** | **Sistema** |
| 1. Seleccionar registro | |
| | 2. Mostrar interfaz |
| 3. Introducir datos | |
| | 4. Comprobar datos obligatorios |
| | 5. Validar campos únicos |
| | 6. Crear registro |

Tabla 3.1: Descripción textual de RF01: Registrar usuario

Y el diagrama de actividad:

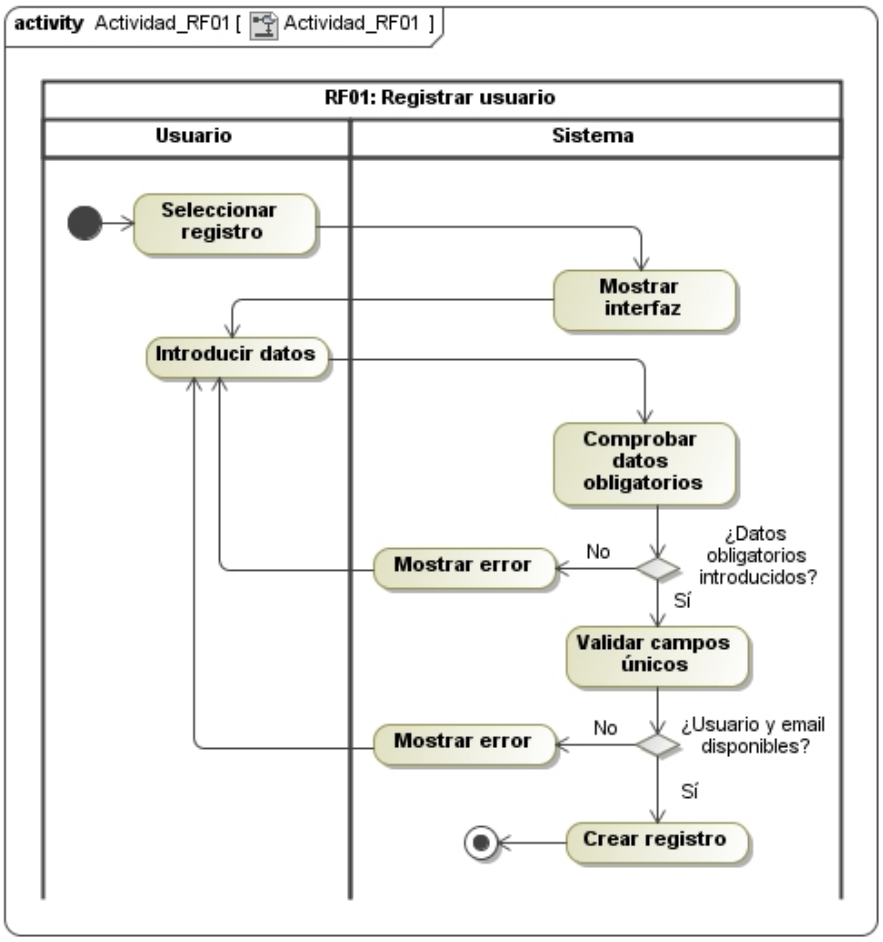


Figura 3.2: Diagrama de actividad de RF01: Registrar usuario

- **RF02: Loguear usuario**. El sistema deberá permitir la autenticación de usuarios para su acceso. Este proceso se realizará mediante el par usuario-contraseña. Señalar que, como se ha especificado en el requisito previo, la contraseña se almacena encriptada, de modo que la introducida por el usuario que trata de acceder ha de ser tratada por el mismo algortimo para poder realizar la comparación.

  La descripción textual:

| **RF02: Loguear usuario** | |
|---|---|
| **Usuario** | **Sistema** |
| 1. Seleccionar login | |
| | 2. Mostrar interfaz |
| 3. Introducir datos | |
| | 4. Comprobar campos obligatorios |
| | 5. Autenticar |
| | 6. Completar login |

Tabla 3.2: Descripción textual de RF02: Loguear usuario

Y el diagrama de actividad:

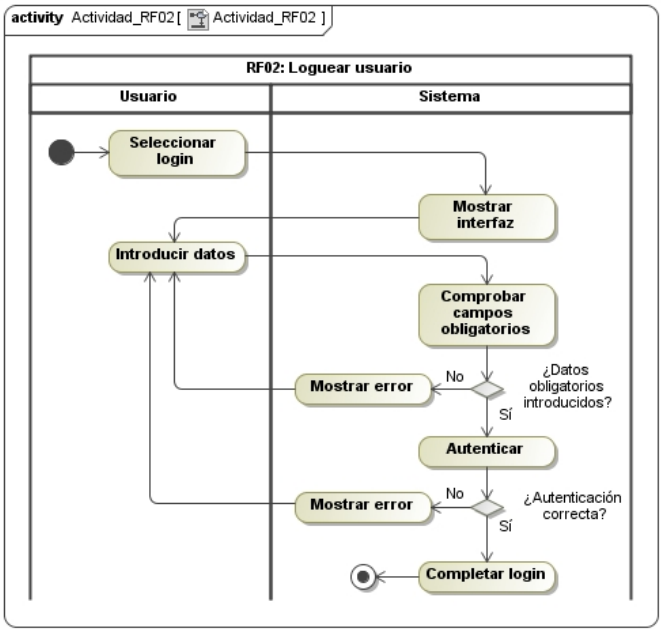


Figura 3.3: Diagrama de actividad de RF02: Loguear usuario

- **RF03: Modificar perfil**. El sistema deberá permitir la modificación del perfil de usuario solicitados en el proceso de registro, considerando que el nombre de usuario y dirección de correo han de ser únicos.

  La descripción textual:

| **RF03: Modificar perfil** | |
|---|---|
| **Usuario** | **Sistema** |
| [Pto. inclusión: RF02: Loguear usuario] | |
| 1. Seleccionar modificar perfil | |
| | 2. Mostrar interfaz |
| 3. Introducir datos | |
| | 4. Validar campos únicos |
| | 5. Solicitar confirmación |
| 6. Confirmar operación | |
| | 7. Actualizar registro |
| | 8. Confirmar operación terminada |

Tabla 3.3: Descripción textual de RF03: Modificar perfil

Y el diagrama de actividad:

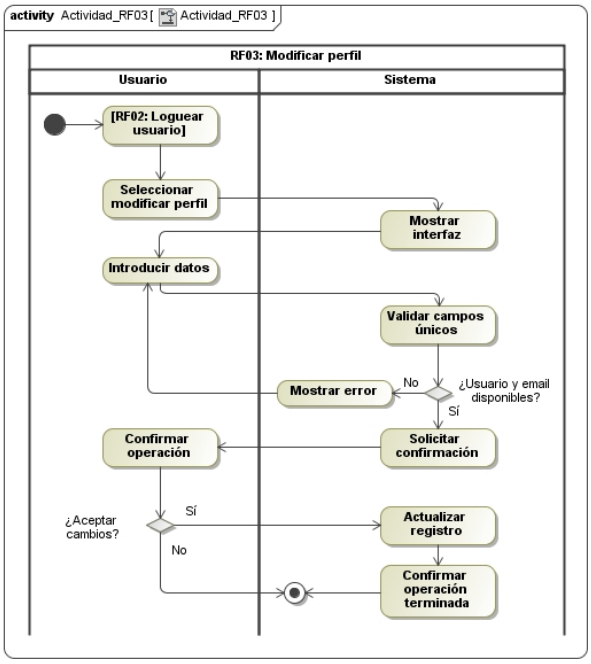


Figura 3.4: Diagrama de actividad de RF03: Modificar perfil

- **RF04: Borrar perfil**. El sistema deberá permitir el borrado completo de perfiles. Este proceso requerirá de una confirmación explícita por parte del usuario y, en el momento en que se realice, será irreversible, borrando el registro específico de la base de datos. El dinero ingresado será devuelto y las posibles reservas, canceladas.

  La descripción textual:

| **RF04: Borrar perfil** | |
|---|---|
| **Usuario** | **Sistema** |
| [Pto. inclusión: RF02: Loguear usuario] | |
| 1. Seleccionar borrar perfil | |
| | 2. Solicitar confirmación |
| 3. Confirmar operación | |
| | 4. Devolver saldo y eliminar posibles reservas |
| | 5. Borrar registro |

Tabla 3.4: Descripción textual de RF04: Borrar perfil

Y el diagrama de actividad:

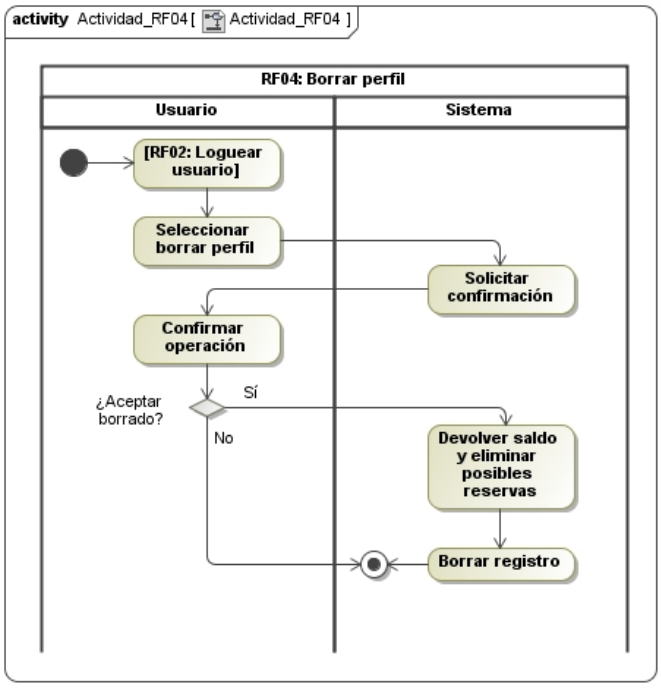


Figura 3.5: Diagrama de actividad de RF04: Borrar perfil

- **RF05: Ingresar saldo**. El sistema deberá permitir el ingreso de dinero, de modo que el usuario cuente con saldo suficiente para que pueda operar con el parque de bicicletas.

  La descripción textual:

| **RF05: Ingresar saldo** | |
|---|---|
| **Usuario** | **Sistema** |
| [Pto. inclusión: RF02: Loguear usuario] | |
| 1. Seleccionar ingreso | |
| | 2. Mostrar interfaz |
| 3. Introducir datos | |
| | 4. Actualizar registro |
| | 5. Confirmar operación terminada |

Tabla 3.5: Descripción textual de RF05: Ingresar saldo

Y el diagrama de actividad:

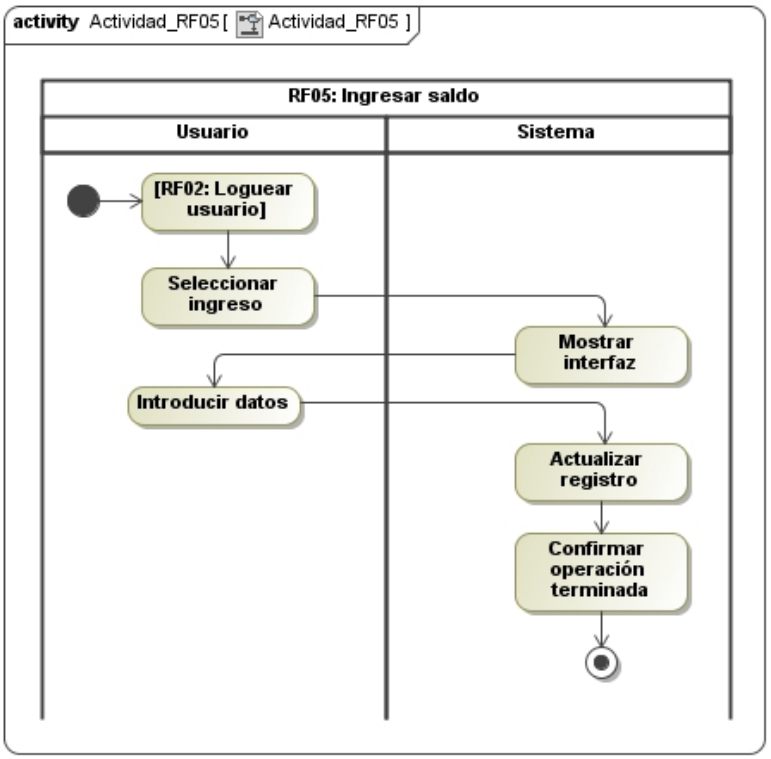


Figura 3.6: Diagrama de actividad de RF05: Ingresar saldo

- **RF06: Coger bicicleta**. El sistema deberá permitir coger bicicletas a los usuarios registrados y logueados en el sistema. Se han tener en cuenta las siguientes restricciones:

  - El usuario sólo puede tener una bicicleta cogida a la vez.
  - El usuario ha de disponer de saldo suficiente para completar la operación.
  - En caso de haber reservado previamente, el usuario sólo podrá coger la bicicleta de la estación en la que haya realizado dicha reserva.
  - Las bicicletas reservadas por otros usuarios no están disponibles.

  Señalar, a su vez, que la estación deberá mostrar una tarifa adaptada a la disponibilidad de bicicletas. Es decir, partiendo de una tarifa base, ésta se verá acrecentada a medida que las bicis disponibles se vean reducidas.

  El objetivo es introducir un primer nivel de adaptación a las condiciones del entorno, de modo que los usuarios tengan alicientes para coger bicicletas de estaciones con una mayor disponibilidad.

  La descripción textual:

| **RF06: Coger bicicleta** | |
|---|---|
| **Usuario** | **Sistema** |
| [Pto. inclusión: RF02: Loguear usuario] | |
| 1. Seleccionar estación | |
| | 2. [RF11: Actualizar] |
| 3. Seleccionar coger bici | |
| | 4. Comprobar restricciones |
| | 5. [RF10: Pagar] |
| | 6. Actualizar estado global |
| | 7. Confirmar operación terminada |

Tabla 3.6: Descripción textual de RF06: Coger bicicleta

Y el diagrama de actividad:

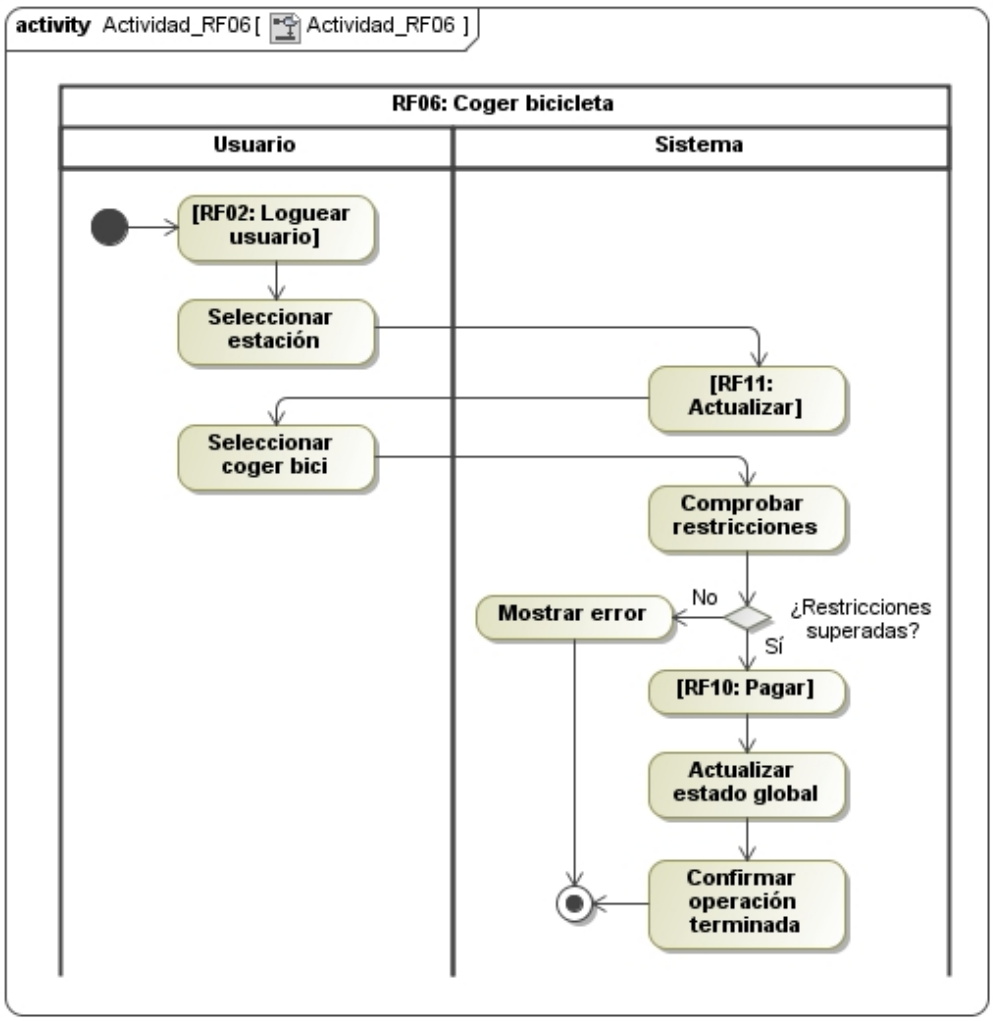


Figura 3.7: Diagrama de actividad de RF06: Coger bicicleta

- **RF07: Dejar bicicleta**. El sistema deberá permitir dejar, en estaciones con disponibilidad suficiente, bicicletas previamente cogidas por usuarios. Se han tener en cuenta las siguientes restricciones:
  - En caso de haber reservado un anclaje previamente, el usuario sólo podrá dejar la bicicleta en la estación en la que haya realizado dicha reserva.
  - Los anclajes reservados por otros usuarios no están disponibles

  La descripción textual:

| **RF07: Dejar bicicleta** | |
|---|---|
| **Usuario** | **Sistema** |
| [Pto. inclusión: RF02: Loguear usuario] | |
| 1. Seleccionar estación | |
| | 2. [RF11: Actualizar] |
| 3. Seleccionar dejar bici | |
| | 4. Comprobar restricciones |
| | 5. Actualizar estado global |
| | 6. Confirmar operación terminada |

Tabla 3.7: Descripción textual de RF07: Dejar bicicleta

Y el diagrama de actividad:

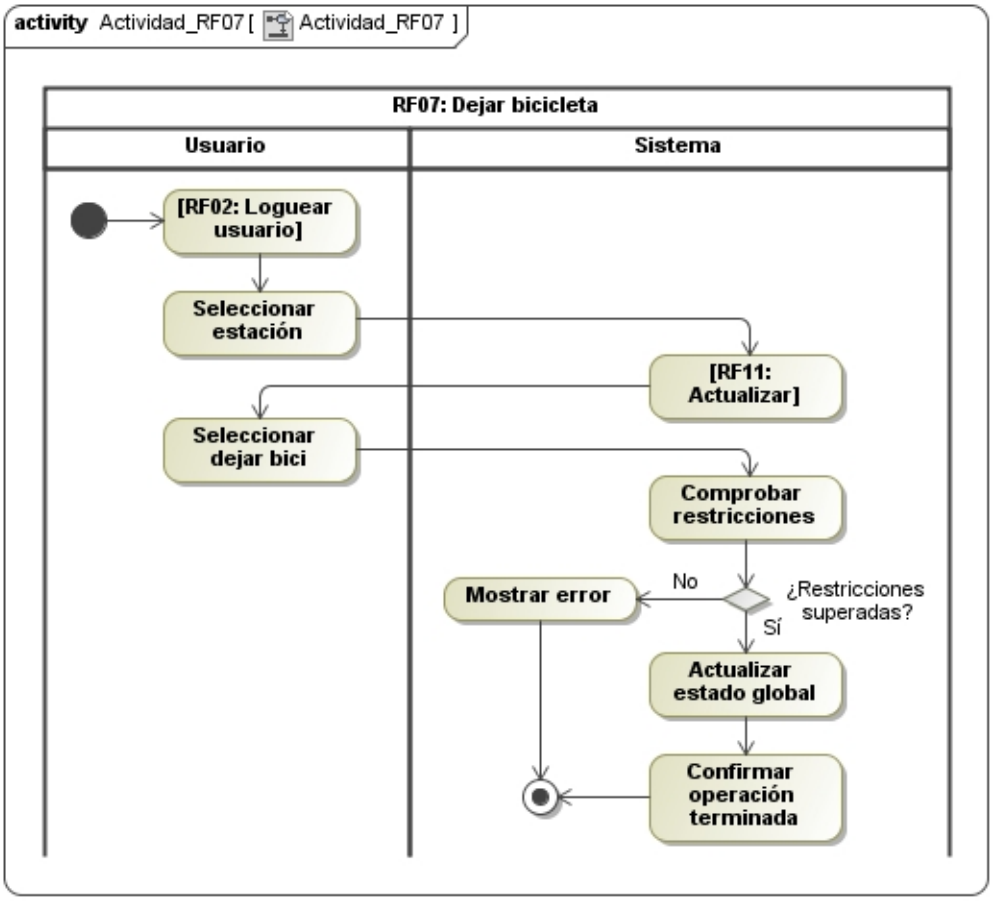


Figura 3.8: Diagrama de actividad de RF07: Dejar bicicleta

- **RF08: Reservar**. El sistema deberá permitir la reserva de recursos, que podrán ser de dos tipos: bicicletas o anclajes. Se han tener en cuenta las siguientes restricciones:
  - El usuario sólo puede tener una reserva de cada tipo a la vez. Es decir, se permiten tener, como mucho, dos reservas simultáneas por usuario: una bicicleta y un anclaje.
  - En caso de tener una bicicleta cogida, el usuario no podrá realizar la reserva de otra.

  La descripción textual:

| **RF08: Reservar** | |
|---|---|
| **Usuario** | **Sistema** |
| [Pto. inclusión: RF02: Loguear usuario] | |
| 1. Seleccionar estación | |
| | 2. [RF11: Actualizar] |
| 3. Seleccionar reservar | |
| | 4. Mostrar opciones de reserva |
| 5. Seleccionar tipo de reserva | |
| | 6. Comprobar restricciones |
| | 7. Actualizar estado global |
| | 8. Confirmar operación terminada |

Tabla 3.8: Descripción textual de RF08: Reservar

Y el diagrama de actividad:

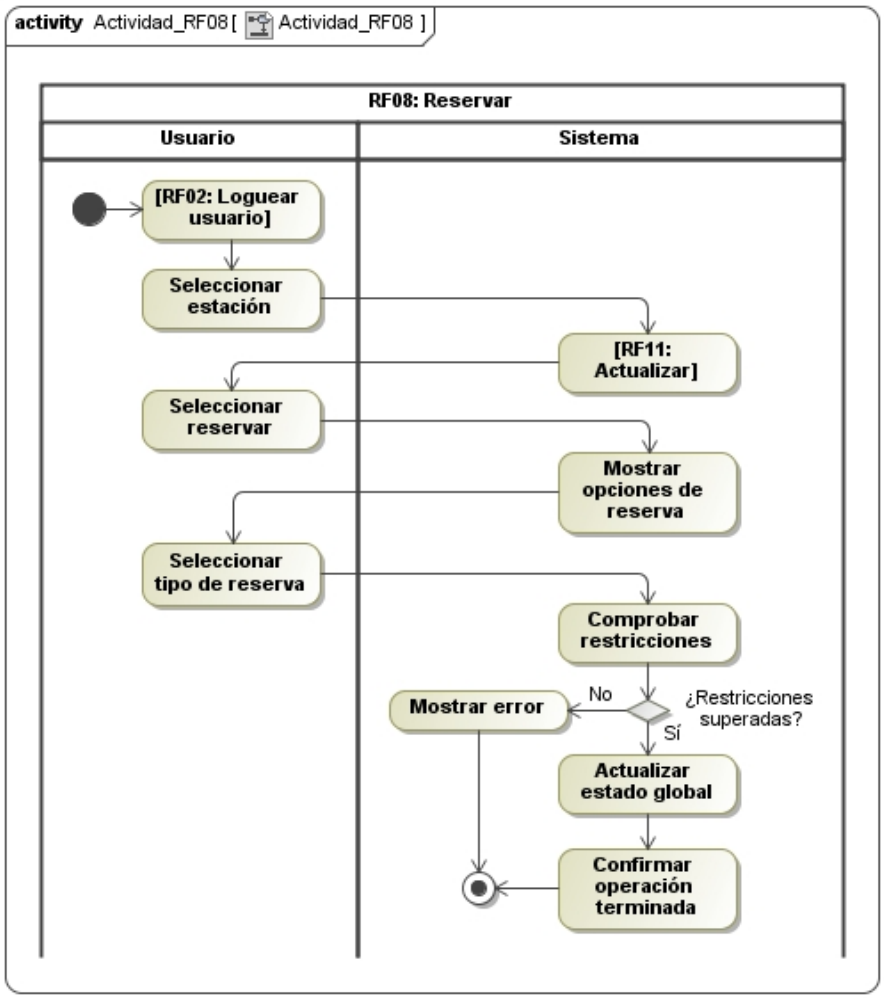


Figura 3.9: Diagrama de actividad de RF08: Reservar

- **RF09: Cancelar reserva**. El sistema deberá permitir la cancelación de reservas, tanto de bicicletas como de anclajes. Esta cancelación se podrá realizar por las siguientes vías:

  - Cancelación explícita por parte del usuario. Este tipo de cancelación implica una confirmación por parte del usuario.
  - Cancelación implícita al cabo de 30 minutos del momento de la reserva.

  Si un usuario ejerce su derecho sobre una reserva, la misma queda automáticamente cancelada.

  Se aporta la descripción textual del requisito referido a la cancelación explíci-

ta, puesto que es la única en la que interviene el usuario de manera directa:

| **RF09: Cancelar reserva (explícita)** | |
|---|---|
| **Usuario** | **Sistema** |
| [Pto. inclusión: RF08: Reservar] | |
| 1. Seleccionar cancelar reserva | |
| | 2. Solicitar confirmación |
| 3. Confirmar operación | |
| | 4. Actualizar estado global |
| | 5. Confirmar operación terminada |

Tabla 3.9: Descripción textual de RF09: Cancelar reserva (explícita)

Y el diagrama de actividad:

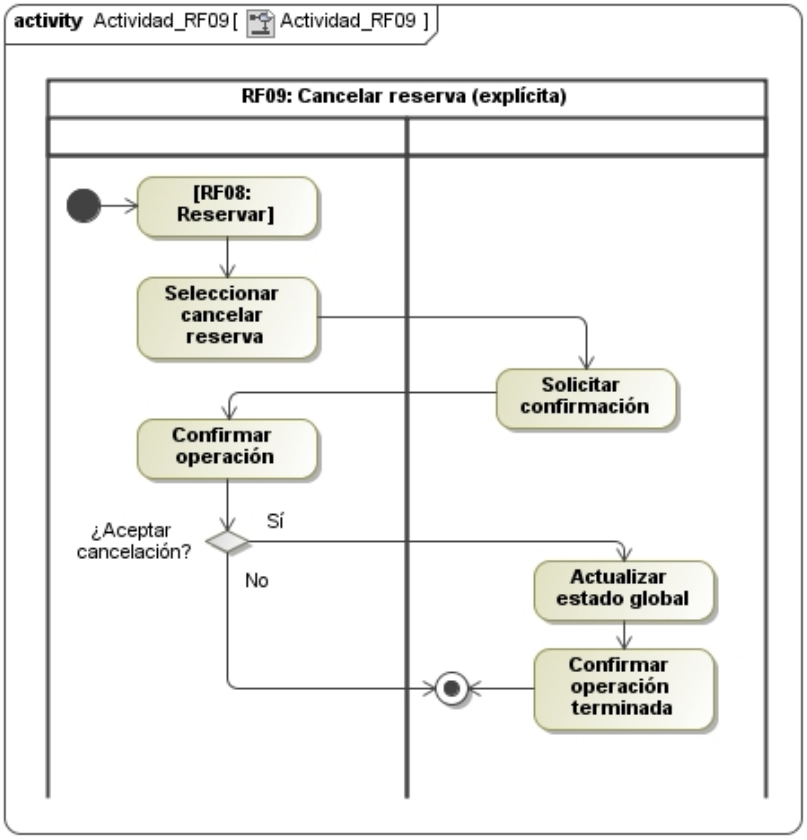


Figura 3.10: Diagrama de actividad de RF09: Cancelar reserva (explícita)

- **RF10: Pagar**. El sistema deberá permitir realizar el pago de las operaciones desarrolladas por los usuarios.

  La descripción textual:

| **RF10:Pagar** | |
|---|---|
| **Usuario** | **Sistema** |
| [Pto. inclusión: RF06: Coger bicicleta] | |
| | 1. Solicitar confirmación |
| 2. Confirmar operación | |
| | 3. Realizar pago |
| | 4. Actualizar usuario |
| | 5. Confirmar operación terminada |

Tabla 3.10: Descripción textual de RF10: Pagar

Y el diagrama de actividad:

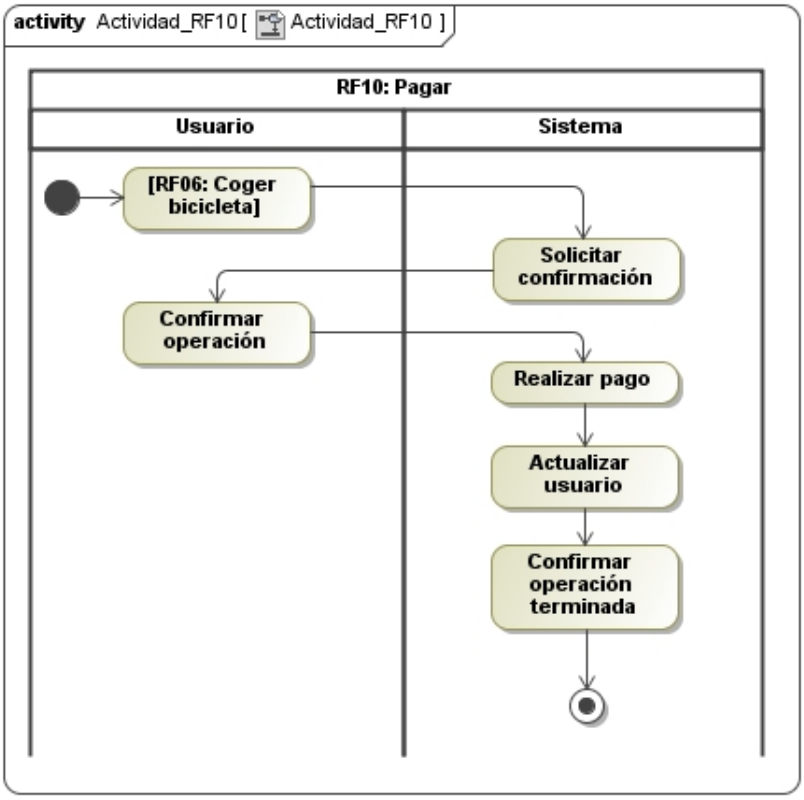


Figura 3.11: Diagrama de actividad de RF10: Pagar

- **RF11: Actualizar**. El sistema deberá permitir actualizar el estado global tomando la información más reciente almacenada en el servidor. En este sentido, con cada solicitud de actualización, en el proceso se deberá comprobar si existen reservas caducadas, cancelándolas automáticamente de modo que se ofrezca al usuario una visión consistente de la disponibilidad del parque de bicicletas.

  La descripción textual:

| **RF11:Actualizar** | |
|---|---|
| **Usuario** | **Sistema** |
| [Pto. inclusión: RF02: Loguear usuario] | |
| 1. Seleccionar actualizar | |
| | 2. Actualizar estado |
| | 3. Mostrar estado actualizado |

Tabla 3.11: Descripción textual de RF11: Actualizar

Y el diagrama de actividad:

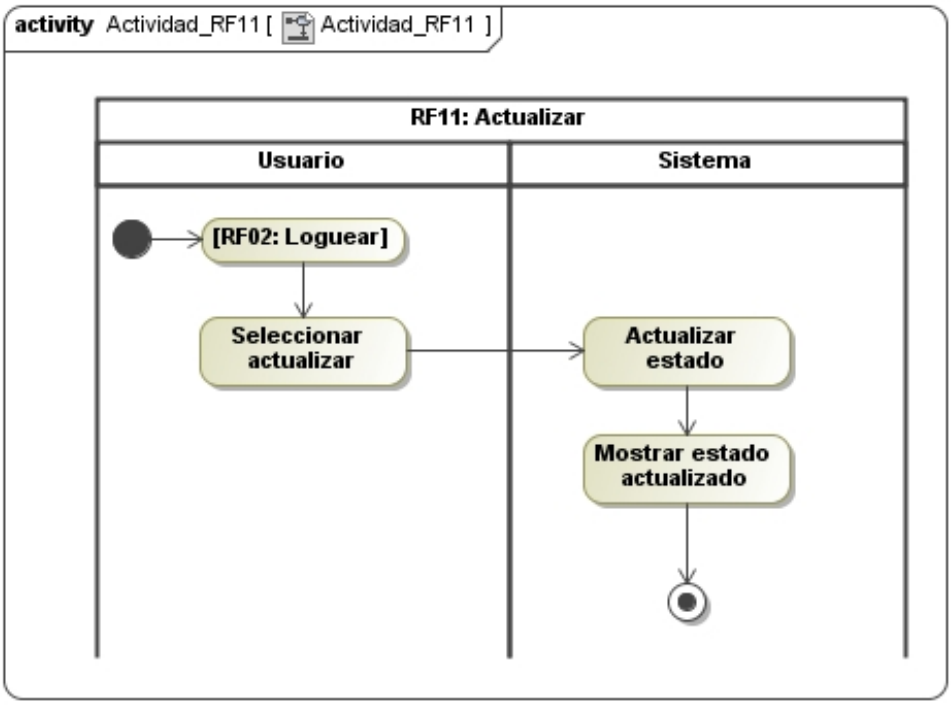


Figura 3.12: Diagrama de actividad de RF11: Actualizar

Finalmente, se pasan a detallar los *Requerimientos No Funcionales* más relevantes:

- **RNF01: Usabilidad**. El sistema ha de presentar una interfaz sencilla e intuitiva, respetando las normas de diseño Android y proporcionando una adecuada retroalimentación al usuario de las operaciones terminadas y de posibles errores en su manejo.

- **RNF02: Eficiencia**. El sistema ha de trasladar toda operación posible a la capa de servidor, de modo que el cliente suponga la menor carga posible para el dispositivo móvil y se reduzcan los tiempos de espera.

- **RNF03: Seguridad**. El sistema ha de asegurar la seguridad en el tratamiento de la información del usuario.

  El detalle de este RNF se recoge en el apartado posterior referido a los Atributos del Sistema.

- **RNF04: Fiabilidad**. El sistema ha operar según lo esperado y minimizando en la medida de los posible la probabilidad de aparición de fallos.

  El detalle de este RNF se recoge en el apartado posterior referido a los Atributos del Sistema.

- **RNF05: Mantenibilidad**. El sistema se ha de desarrollar de modo que pueda ser conservado y restituido (en caso necesario) con facilidad.

  El detalle de este RNF se recoge en el apartado posterior referido a los Atributos del Sistema.

- **RNF06: Portabilidad**. El sistema ha de desarrollarse de modo que pueda ser ejecutado en diferentes plataformas.

  El detalle de este RNF se recoge en el apartado posterior referido a los Atributos del Sistema.

- **RNF07: Disponibilidad**. El sistema ha de permanecer en funcionamiento continuo para prestar una servicio adecuado a los usuarios.

  El detalle de este RNF se recoge en el apartado posterior referido a los Atributos del Sistema.

Este conjunto de RNF suponen una adaptación tomada de los factores de calidad más extendidos como son los Factores de Calidad de McCall [McC77], de Garvin [Gar87] o de la norma ISO 25010 (antigua 9126) [ISO25].

### Requisitos de Rendimiento

Desde el punto de vista de la aplicación móvil, no se espera una carga excesiva en el dispositivo por gestionar únicamente los datos del usuario registrado.

El servidor, por su parte, ha de ser capaz de dar respuesta a una serie de peticiones concurrentes que pueden escalar a la cantidad de dispositivos que tengan instalada la aplicación.

Finalmente, en relación a la cantidad de registros almacenados en la base de datos, se espera que queden registrados de manera individual tanto usuarios como puestos de bicicletas, además de posibles tablas adicionales.

### Restricciones de Diseño

Para el diseño de la aplicación se deberán utilizar componentes compatibles con la versión 4.0 de Android, en caso de no estar disponibles, se deberá recurrir a librerías de soporte para dar servicio a dicha versión, cumpliendo con ella, las restricciones hardware quedan, a su vez, cubiertas.

En relación al servidor de aplicación, éste ha de ser capaz de gestionar accesos concurrentes a recursos compartidos, controlando las posibles condiciones de carrera que puedan aparecer y dando un respuesta oportuna al usuario.

### Atributos del Sistema

- Seguridad.
  - El acceso a la aplicación se realizará mediante el par usuario-contraseña.
  - Las claves de usuario deberán almacenarse de manera segura mediante encriptación SHA-1.
  - Desde la aplicación ningún usuario tendrá acceso a la administración interna de la misma, sino que dicha tarea se realizará directamente sobre el servidor o base de datos donde sólo el desarrollador o mantendor de la aplicación podrá acceder.
- Fiabilidad.
  - El sistema ha de quedar adecuadamente testeado previa distribución para obtener una adecuada cobertura de fallos.
  - La interfaz de usuario ha de ser sencilla e intuitiva y, en caso de ser necesario, deberá informar con el resultado de las operaciones para una adecuada experiencia en su uso.
- Mantenibilidad.

    - La aplicación ha de quedar desarrollada siguiendo las buenas prácticas del desarrollo software (código adecuadamente organizado y comentado, utilización de patrones de diseño estandarizados, utilización de *idioms*, etc.).
    - La herramienta deberá dejar *logs* internos de las áreas que se consideren más relevantes o susceptibles a fallos de modo que puedan ser consultados por los desarrolladores para realizar tareas de depuración.

- Portabilidad.
    - La aplicación podrá ser instalada en cualquier dispositivo con sistema operativo Android (versión mínima 4.0).
    - Los servidores de aplicación y base de datos se podrán desplegar sobre cualquiera de los sistemas operativos más extendidos (Windows, Linux, iOS, etc.).
    - Queda como tarea futura la adaptación de la aplicación móvil a entornos web e iOS.

- Disponibilidad.
    - Los servidores han de estar operativos 24x7 para atender las necesidades de uso.

**Otros Requisitos**

No se consideran otros requisitos de los especificados en secciones previas.

## 3.2. Análisis

### 3.2.1. Introducción

El análisis de los requerimientos descritos ha de dar como resultado la especificación de las características operativas del software, indicando la interfaz de éste y otros elementos del sistema, y estableciendo las restricciones que limitan al software.

Este proceso de análisis otorga la información que se traduce en diseños de arquitectura, interfaz y componentes, así como los medios para evaluar la calidad del software una vez construido.

Por lo tanto, el modelo de análisis supone un puente entre la descripción del sistema y su diseño posterior. Esta relación queda esquematizada en la figura 3.13.

Este modelo se desarrollará en dos pasos:

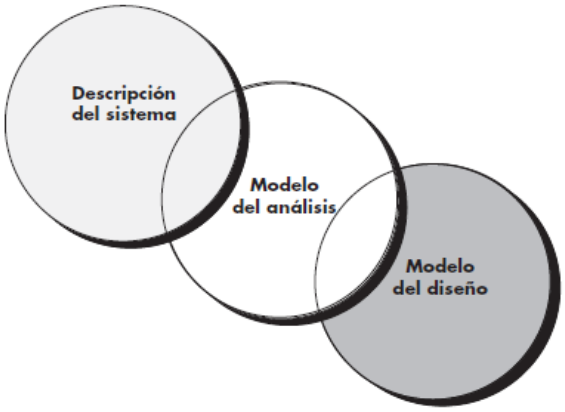


Figura 3.13: El modelo de análisis como puente entre los requerimientos y el diseño del software *Fuente: [Pre10]*

1. **Análisis conceptual del sistema**. Se presentará la vista estática general del sistema mediante un diagrama de clases de análisis. El objetivo es el de obtener una perspectiva global y con un alto nivel de abstracción del software y sus entidades básicas.

2. **Análisis y desarrollo de requisitos**. Los requisitos descritos en la sección anterior se comenzarán a desarrollar mediante diagramas de colaboración (o comunicación), de modo que se tenga una primera visión de la futura implementación de cada uno y se muestre una primera aproximación a la interacción entre entidades.

Cabe señalar que, en este punto, la atención se centra en el *qué*, no en el *cómo*, con lo no se entrará en detalles técnicos, de diseño o de implementación.

### 3.2.2. Análisis conceptual del sistema

Para el análisis y representación del modelo conceptual de datos se recurrirá diagrama de clases de análisis, que de lugar a la vista estática general del sistema e identifique sus entidades básicas usando la terminología del dominio aplicable. Si bien se cuenta con un cierto grado de subjetividad, las clases potenciales a incluir en este punto deberán cubrir todos (o casi todos) los puntos siguientes [Coa91]:

1. *Información retenida*. Debe recordarse la información sobre la clase para que el sistema pueda funcionar.

2. *Servicios necesarios*. La clase potencial debe tener un conjunto de operaciones identificables que cambien en cierta manera el valor de sus atributos.

3. *Atributos múltiples*. Centrando la atención en la información "principal";

4. *Atributos comunes*. Para la clase potencial se define un conjunto de atributos y se aplican éstos a todas las instancias de la clase.

5. *Operaciones comunes*. Se define un conjunto de operaciones para la clase potencial y éstas se aplican a todas las instancias de la clase.

6. *Requerimientos esenciales*. Las entidades externas que aparezcan en el espacio del problema y que produzcan o consuman información esencial para la operación de cualquier solución para el sistema casi siempre se definirán como clases en el modelo de requerimientos.

Con todo, quedan identificadas las siguientes entidades básicas:

- **Gestor Conexión Remota**. Desde el momento en el que la aplicación contará con una arquitectura cliente/servidor y la comunicación entre ambas entidades se realizará de manera remota, se requiere de una entidad que gestione dicha comunicación y sepa dar respuesta a las operaciones CRUD[3].
- **Entidad Remota**. La entidad que quedará gestionada por la clase anterior, supone la generalización de alguna de las siguientes entidades:
  - **Usuario**. Se deberán recoger los atributos básicos para poder realizar el login, como el nombre de usuario y contraseña, datos personales, como el nombre y apellidos, y datos operativos, como el saldo o datos acerca de si se tiene una reserva o bicicleta. Del mismo modo, el usuario deberá ser capaz de interactuar con las estaciones de bicicletas, cogiendo y dejando las mismas o gestionando reservas.
  - **Estación**. La estación de bicicletas, deberá contar con datos identificativos, como la dirección donde se encuentre ubicada, y con datos operativos referidos al total de bicicletas o anclajes disponibles, reservados, etc., así como la tarifa base de esa estación. La estación deberá poder gestionar las operaciones comenzadas por los usuarios y mostrar su estado convenientemente actualizado.
  - **Reserva**. Se considera relevante contar con una clase para almacenar las reservas de usuarios, de modo que su gestión global sea más sencilla. Esta entidad deberá contar con los datos de la reserva (tipo (anclaje o bicicleta), nombre de usuario, fecha y estación) y con la constante que defina el tiempo máximo por reserva (30 minutos, de acuerdo a los requisitos).

[3]Siglas en inglés de las operaciones básicas para la gestión de datos: Create, Read, Update y Delete

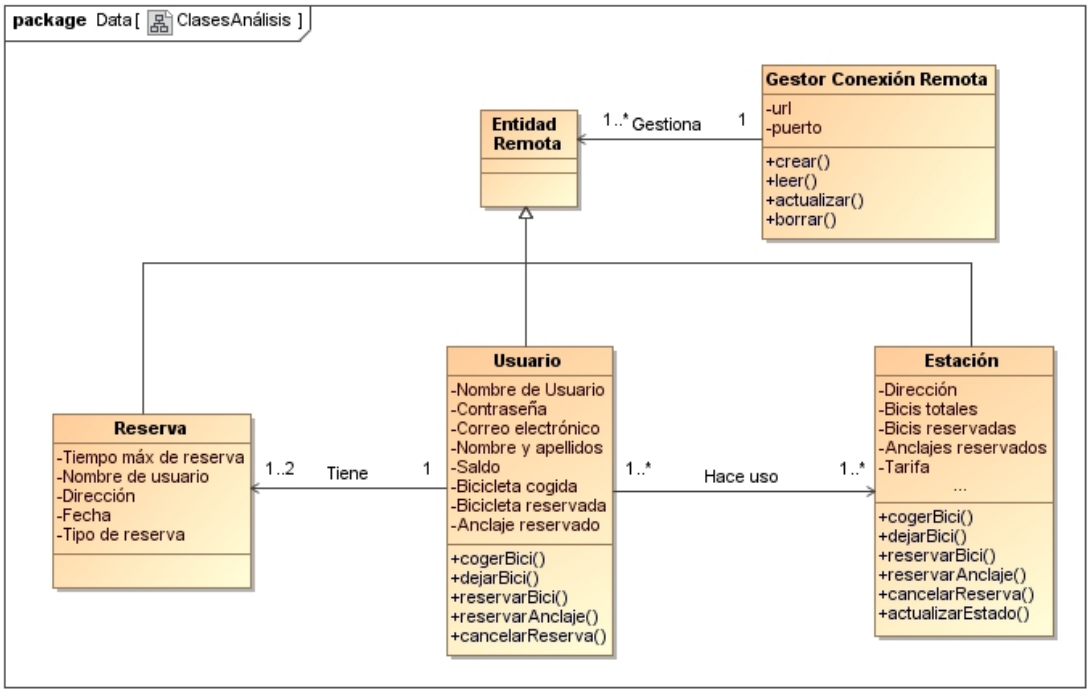


Figura 3.14: Diagrama de clases de análisis

De este modo, el *Gestor de Conexión Remota* será el encargado de enviar y recibir datos para las entidades, sin necesidad de diferenciar entre ninguna de ellas por atender directamente la general *Entidad Remota*.

Este conjunto de entidades básicas, junto con sus potenciales atributos y operaciones básicas, queda modelizado en la figura 3.14.

### 3.2.3. Análisis y desarrollo de requisitos

Con la vista estática del futuro sistema presentada, a continuación se pasa a describir las realizaciones más relevantes de cada caso de uso presentados en la figura 3.1. Esto es, por cada requisito funcional se modelizará la realización de su camino principal y caminos alternativos de relevancia obtenidos a partir de los diagramas de actividad anteriores.

El objetivo es el de componer un análisis adecuado de las posibles operaciones e interacciones que cada requisito puede desencadenar y las entidades afectadas ante los caminos de ejecución más probables.

- **RF01: Registrar usuario**. Tres caminos básicos:

1. Camino principal, la operación de registro finaliza satisfactoriamente. Ver figura 3.15.
2. Camino alternativo 1, el usuario no introduce todos los datos obligatorios. Ver figura 3.16.
3. Camino alternativo 2, se aportan todos los datos requeridos, pero el nombre de usuario o dirección de correo electrónico ya están en uso. Ver figura 3.17.

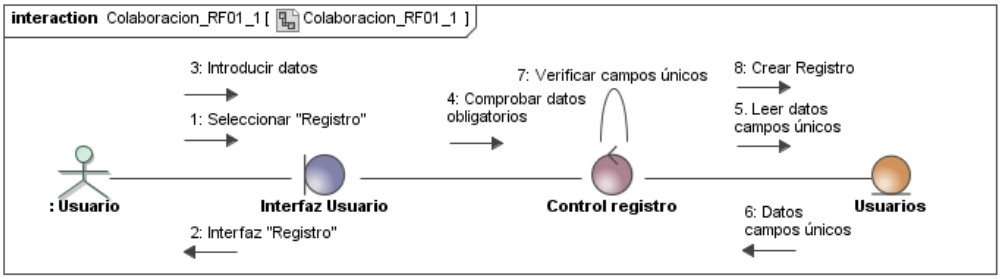


Figura 3.15: Diagrama de colaboración del camino principal del RF01

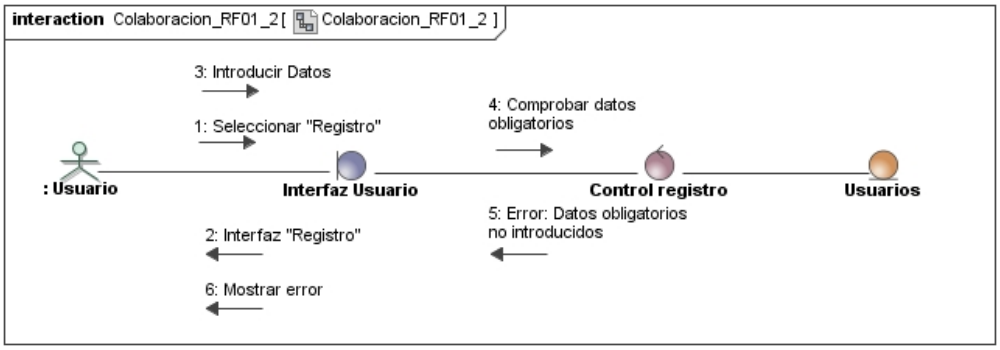


Figura 3.16: Diagrama de colaboración del primer camino alternativo del RF01

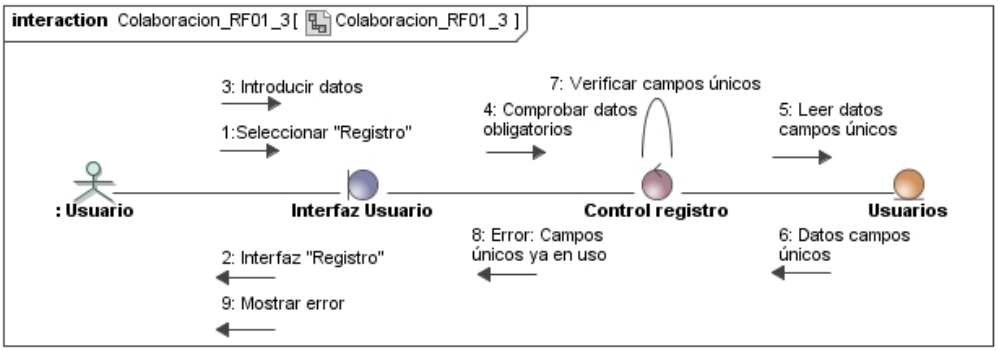


Figura 3.17: Diagrama de colaboración del segundo camino alternativo del RF01

- **RF02: Loguear usuario**. Tres caminos básicos:

  1. Camino principal, la operación de inicio de sesión finaliza satisfactoriamente. Ver figura 3.18.
  2. Camino alternativo 1, el usuario no introduce todos los datos obligatorios. Ver figura 3.19.
  3. Camino alternativo 2, autenticación incorrecta, se aportan todos los datos requeridos, pero el nombre de usuario y/o contraseña no son correctos. Ver figura 3.20.

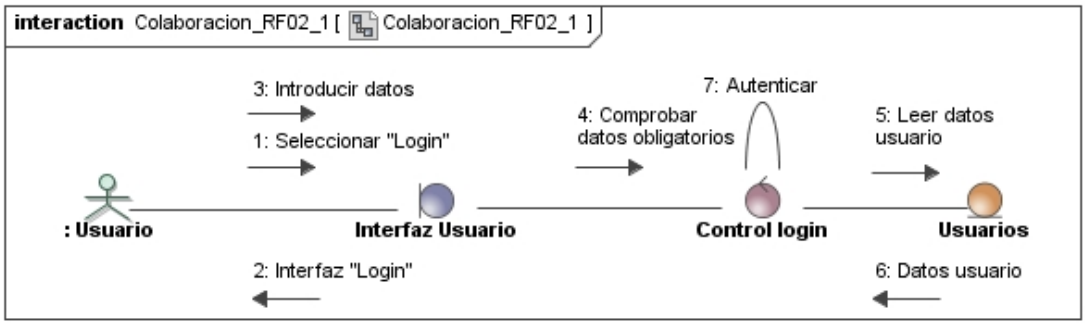


Figura 3.18: Diagrama de colaboración del camino principal del RF02

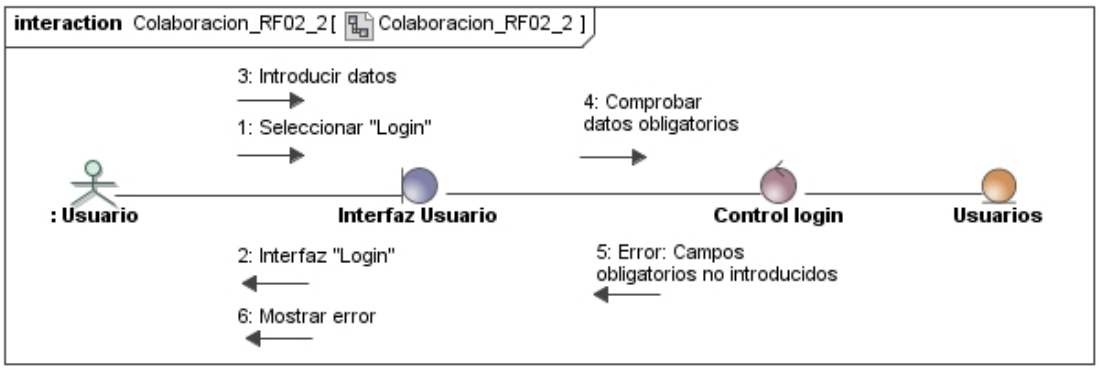


Figura 3.19: Diagrama de colaboración del primer camino alternativo del RF02

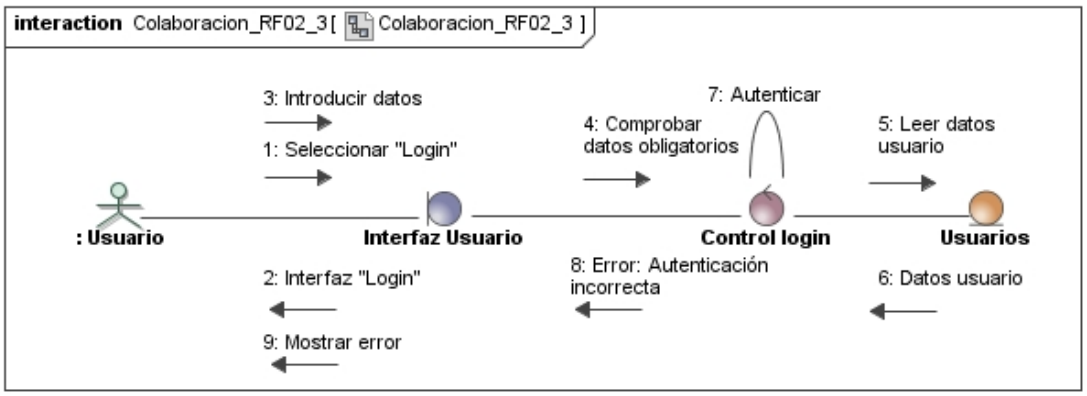


Figura 3.20: Diagrama de colaboración del segundo camino alternativo del RF02

- **RF03: Modificar perfil**. Se consideran tres caminos básicos:

  1. Camino principal, la operación de modificación de perfil finaliza satisfactoriamente. Ver figura 3.21.
  2. Camino alternativo 1, el nuevo nombre de usuario y/o dirección de correo electrónico ya están en uso. Ver figura 3.22.
  3. Camino alternativo 2, el usuario cancela la operación ante la confirmación final. Ver figura 3.23.

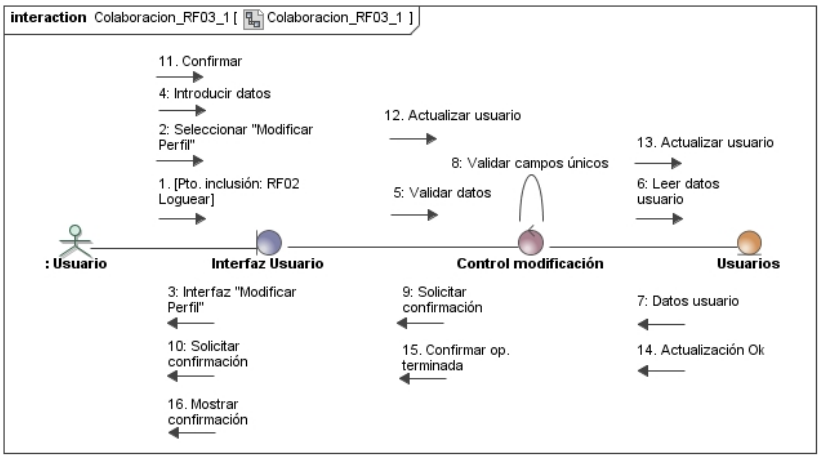


Figura 3.21: Diagrama de colaboración del camino principal del RF03

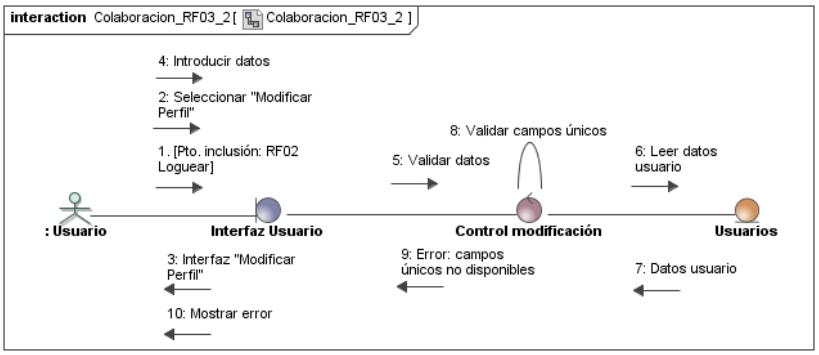


Figura 3.22: Diagrama de colaboración del primer camino alternativo del RF03

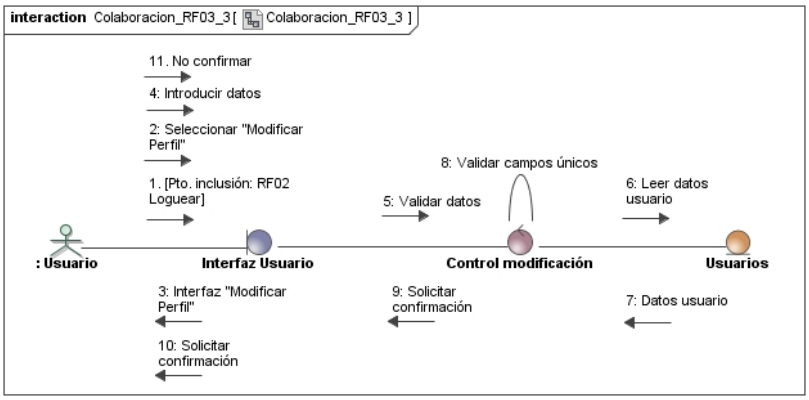


Figura 3.23: Diagrama de colaboración del segundo camino alternativo del RF03

- **RF04: Borrar perfil**. Se consideran dos caminos básicos:

  1. Camino principal, la operación de borrado de perfil finaliza satisfactoriamente. Ver figura 3.24.
  2. Camino alternativo 1, el usuario cancela la operación ante la confirmación final. Ver figura 3.25.

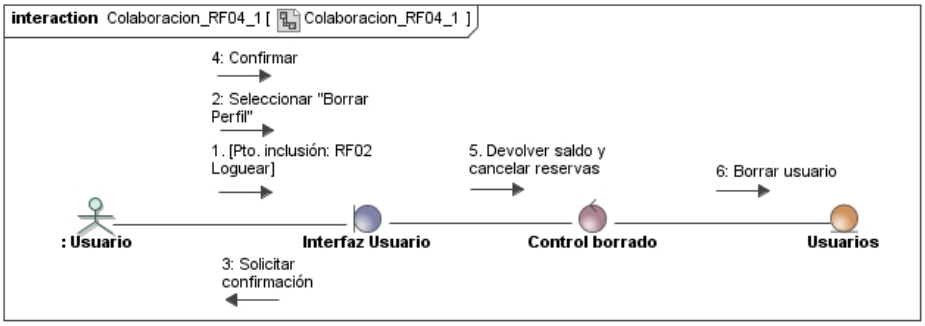


Figura 3.24: Diagrama de colaboración del camino principal del RF04

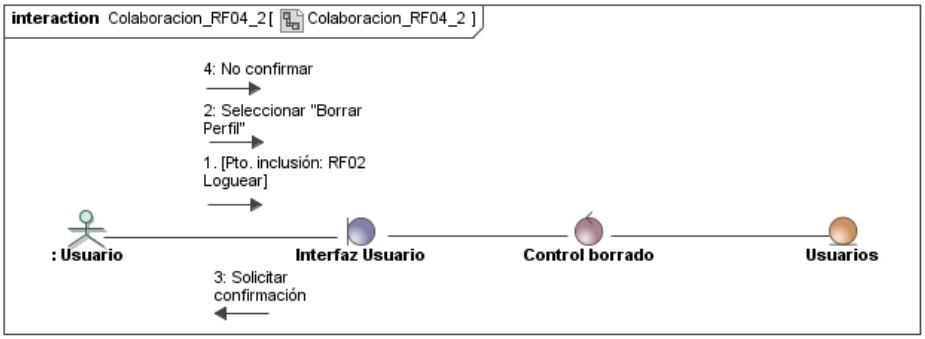


Figura 3.25: Diagrama de colaboración del primer camino alternativo del RF04

- **RF05: Ingresar saldo**. Se considera un único camino básico: el ingreso sin incidencias de saldo. Ver figura 3.26.

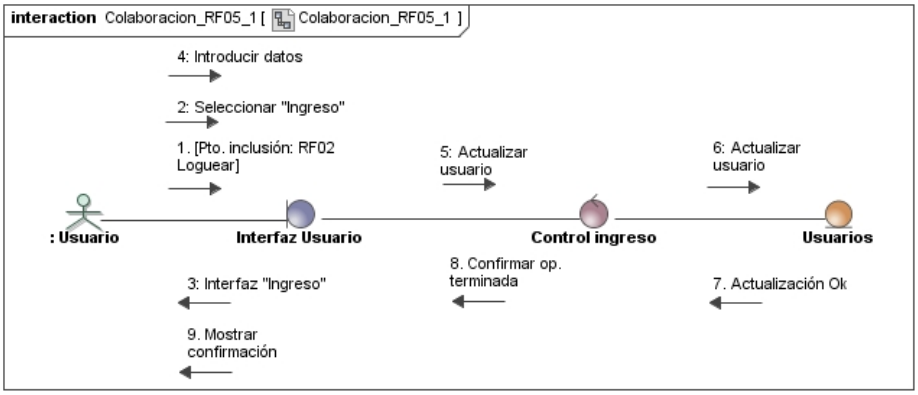


Figura 3.26: Diagrama de colaboración del camino principal del RF05

- **RF06: Coger bicicleta**. Se consideran dos caminos básicos:

  1. Camino principal, la operación finaliza satisfactoriamente. Ver figura 3.27.
  2. Camino alternativo 1, el usuario no supera alguna de las restricciones impuestas sobre la operación (mencionadas en la especificación de requisitos anterior). Ver figura 3.28.

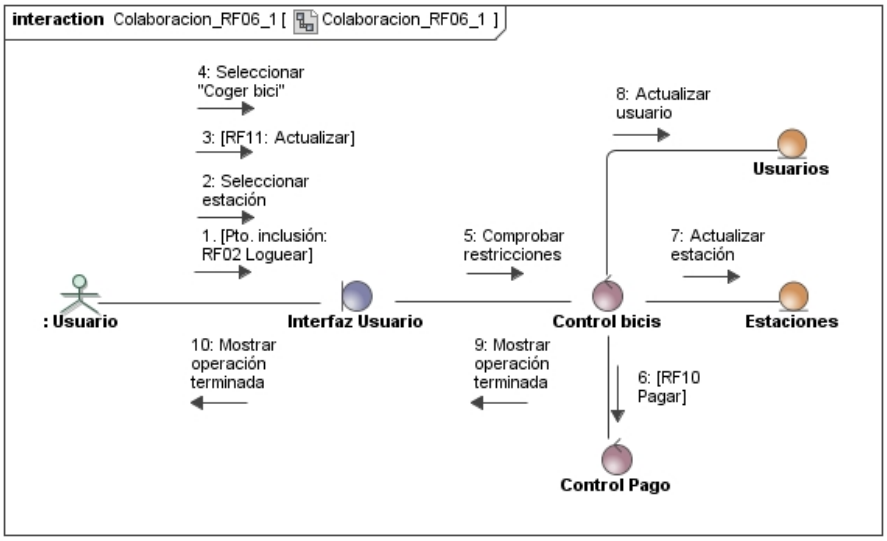


Figura 3.27: Diagrama de colaboración del camino principal del RF06

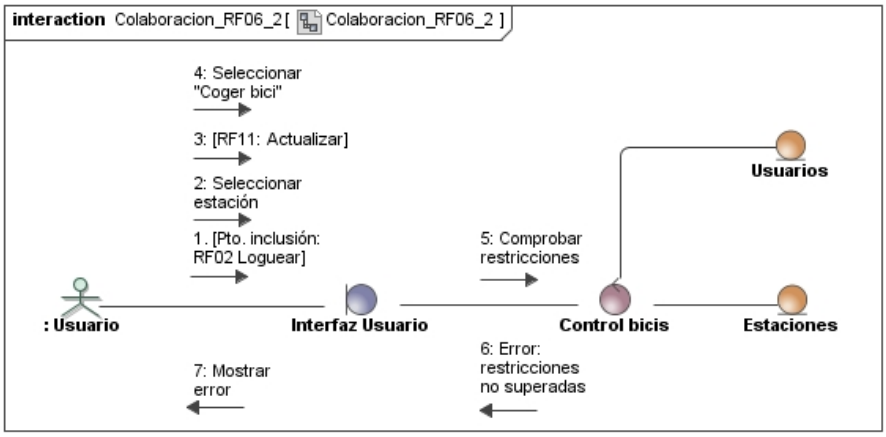


Figura 3.28: Diagrama de colaboración del primer camino alternativo del RF06

- **RF07: Dejar bicicleta**. Se consideran dos caminos básicos:

  1. Camino principal, la operación finaliza satisfactoriamente. Ver figura 3.29.
  2. Camino alternativo 1, el usuario no supera alguna de las restricciones impuestas sobre la operación (mencionadas en la especificación de requisitos anterior). Ver figura 3.30.

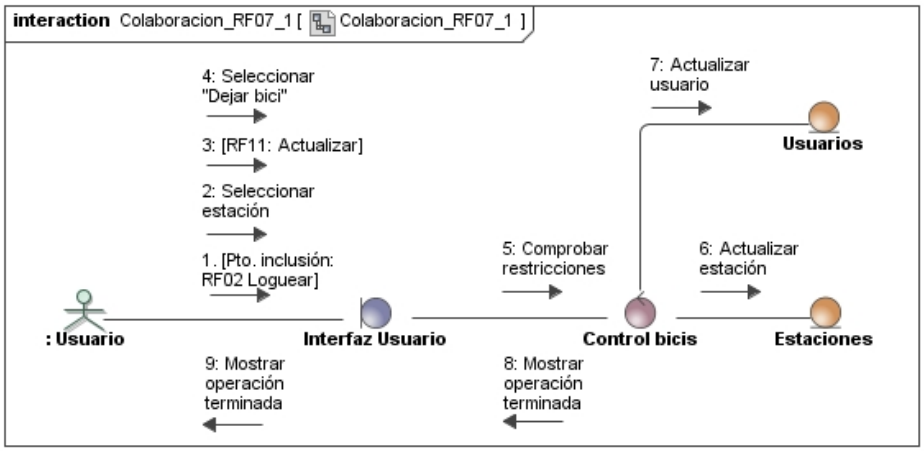


Figura 3.29: Diagrama de colaboración del camino principal del RF07

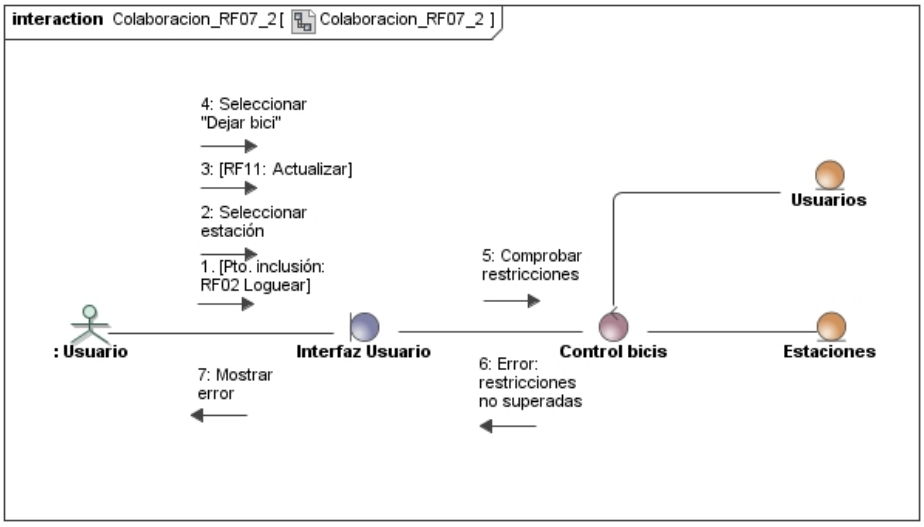


Figura 3.30: Diagrama de colaboración del primer camino alternativo del RF07

- **RF08: Reservar**. Se consideran dos caminos básicos:

  1. Camino principal, la operación finaliza satisfactoriamente. Ver figura 3.31.
  2. Camino alternativo 1, el usuario no supera alguna de las restricciones impuestas sobre la operación (mencionadas en la especificación de requisitos anterior). Ver figura 3.32.

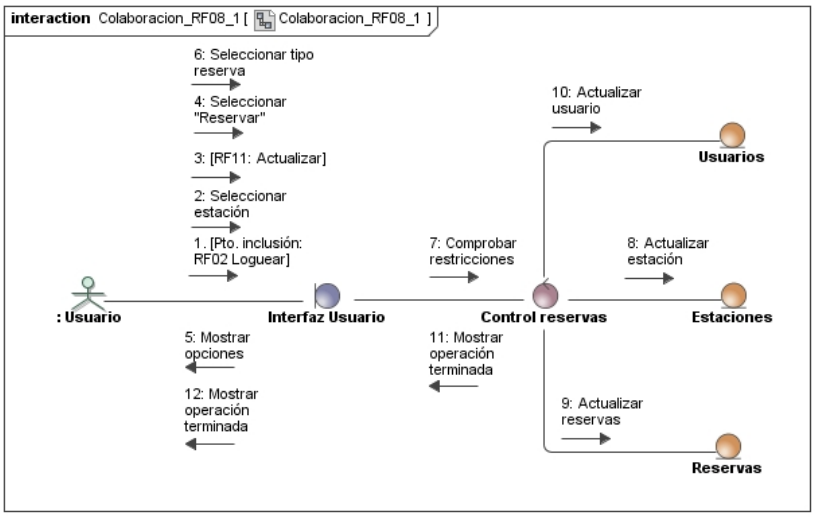


Figura 3.31: Diagrama de colaboración del camino principal del RF08

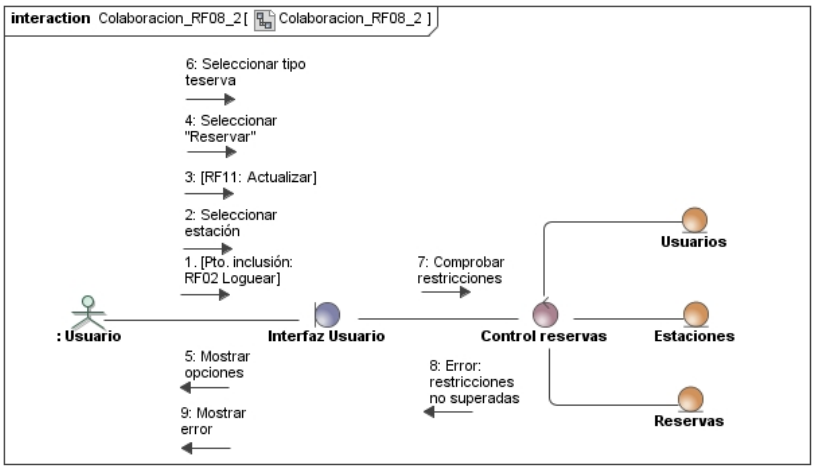


Figura 3.32: Diagrama de colaboración del primer camino alternativo del RF08

- **RF09: Cancelar reserva**. Se consideran dos caminos básicos:

  1. Camino principal, la operación finaliza satisfactoriamente. Ver figura 3.33.
  2. Camino alternativo 1, el usuario cancelar la operación ante la confirmación final. Ver figura 3.34.

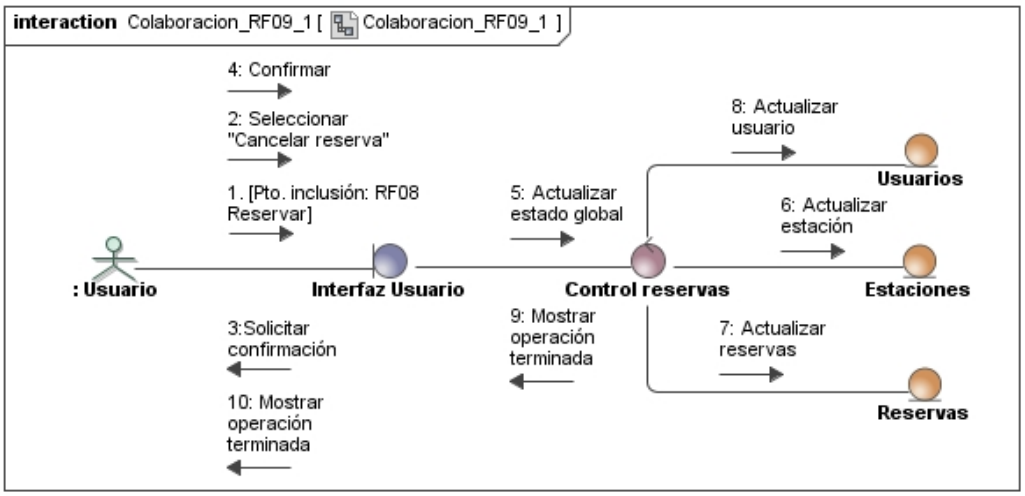


Figura 3.33: Diagrama de colaboración del camino principal del RF09

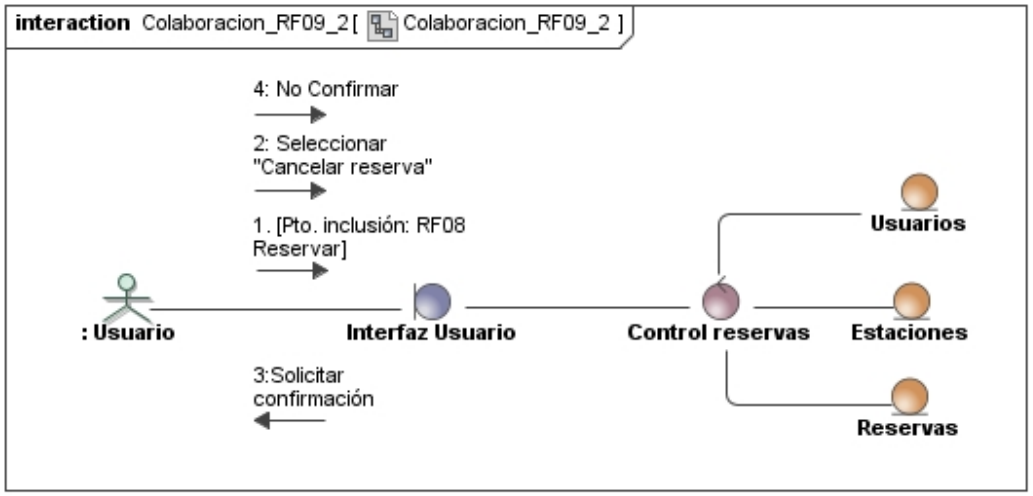


Figura 3.34: Diagrama de colaboración del primer camino alternativo del RF09

- **RF10: Pagar**. Se consideran dos caminos básicos:

  1. Camino principal, la operación finaliza satisfactoriamente. Ver figura 3.35.
  2. Camino alternativo 1, el usuario cancelar la operación ante la confirmación final. Ver figura 3.36.

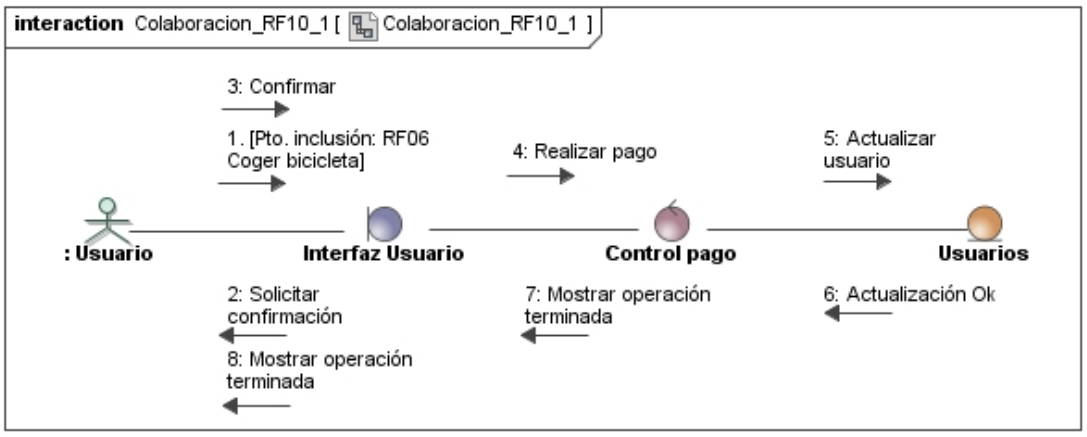


Figura 3.35: Diagrama de colaboración del camino principal del RF10

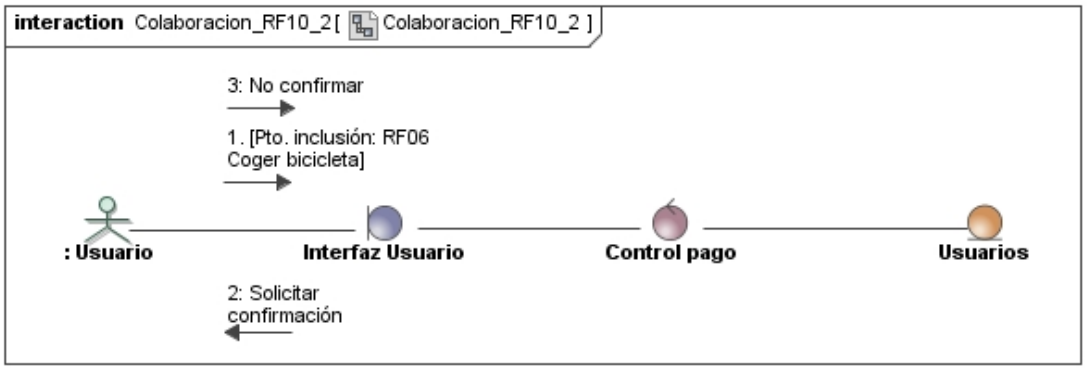


Figura 3.36: Diagrama de colaboración del primer camino alternativo del RF10

- **RF11: Actualizar**. Se consideran dos caminos básicos:

1. Camino principal, en el proceso de actualización se identifican reservas caducadas, que se cancelan automáticamente. Ver figura 3.37.

2. Camino alternativo 1, no se identifican reservas caducadas, el proceso consiste en leer datos actualizados. Ver figura 3.38.

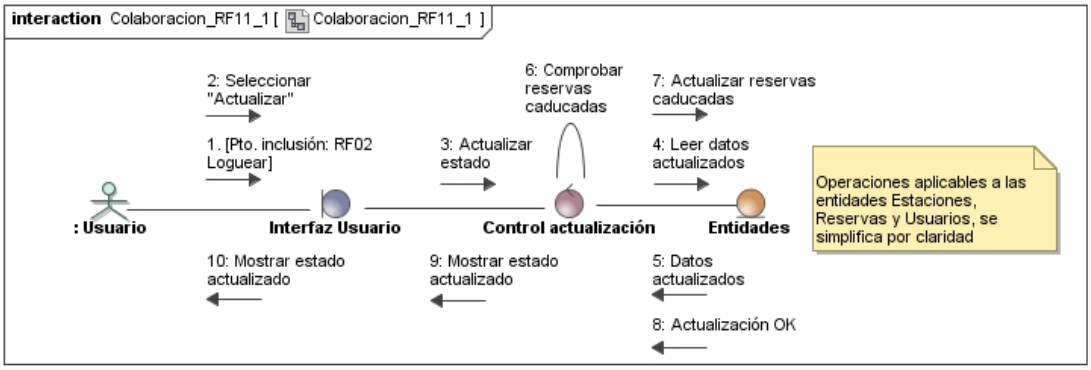


Figura 3.37: Diagrama de colaboración del camino principal del RF11

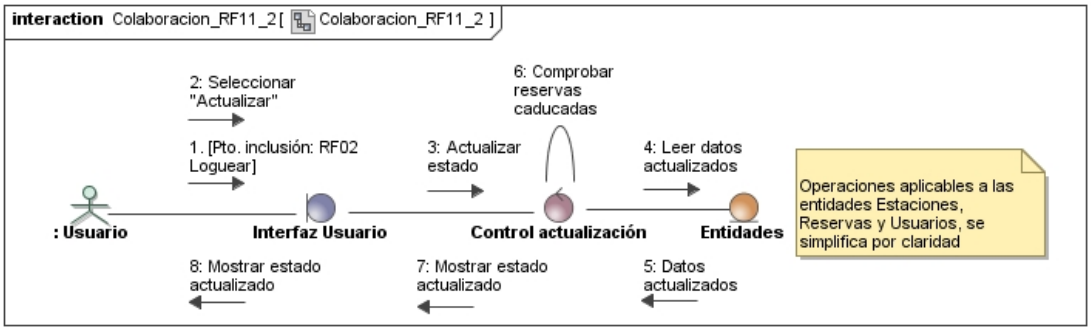


Figura 3.38: Diagrama de colaboración del primer camino alternativo del RF11

## 3.3. Diseño

### 3.3.1. Introducción

Sintetizando a Pressman [Pre10], el diseño del software comienza una vez que se han analizado y modelado los requerimientos (etapas centradas en describir los datos que se necesitan, la función y el comportamiento), es la última acción de la ingeniería de software dentro de la actividad de modelado y prepara la etapa de construcción (generación y prueba de código) proporcionando detalles sobre arquitectura del software, estructuras de datos, interfaces y componentes.

Así, el modelo de diseño que a continuación se desarrolla, se dividirá dos partes principales:

1. **Diseño de la arquitectura**. Se presentará el diseño de la arquitectura general que adoptará el sistema y de las interrelaciones entre sus componentes. La modelización de dicha estructura se llevará a cabo mediante el diagra-

ma de clases de diseño, dando lugar a visión más detallada del sistema a implementar.

2. **Diseño de las interfaces**. En esta segunda parte se recogerán las decisiones de diseño más relevantes con referencia a las interfaces de usuario y comunicación de la aplicación.

Por su parte, el diseño en el nivel de componentes y despliegue se tratará en la sección 3.4 siguiente, acerca del *Modelo de Implementación*.

### 3.3.2. Diseño de la arquitectura

La *arquitectura del software* supone "la estructura o estructuras del sistema, lo que comprende a los componentes del software, sus propiedades externas visibles y las relaciones entre ellos" [Bas03].

Si bien cabe señalar que la arquitectura no es el software operativo. Es una representación que permite [Pre10]: 1) analizar la efectividad del diseño para cumplir los requerimientos establecidos, 2) considerar alternativas arquitectónicas en una etapa en la que hacer cambios al diseño todavía es relativamente fácil y 3) reducir los riesgos asociados con la construcción del software.

La definición previa pone el énfasis en el papel de los "componentes del software", que puede ser algo tan simple como un módulo de programa o una clase orientada a objeto, si bien también puede ampliarse a conceptos más extensos y complejos.

#### Estilo arquitectónico

Considerando los conceptos anteriores y con base en las especificación de requisitos previa, el estilo arquitectónico sobre el que se sustentará el sistema es el *Cliente/Servidor* en su variación de tres capas, tal y como se muestra en la figura 3.39:

1. **Cliente**. Supone la capa de presentación, la Interfaz de Usuario. Su función es traducir tareas y/o resultados para resultar comprensibles por el usuario.

2. **Servidor**. La capa lógica encargada de coordinar la aplicación y procesar datos entre el resto de capas.

3. **Base de datos**. Implica la capa de datos, donde la información es almacenada y recuperada por parte de la capa lógica para su procesamiento.

En la sección 3.4 se entra en mayor detalle sobre estas capas, especificando las tecnologías para la implementación de cada una y la comunicación entre ellas.

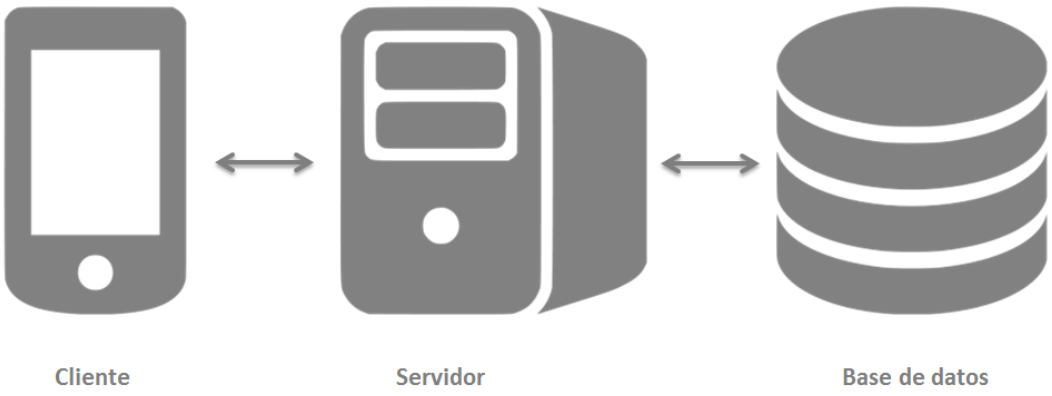


Figura 3.39: Esquema de la arquitectura Cliente/Servidor de tres capas

### Modelización de la arquitectura

En el modelo de análisis desarrollado en la sección 3.2 quedaron definidas un conjunto de clases de análisis (representadas en la figura 3.14). Con un nivel de abstracción relativamente alto, cada una describe algún elemento del dominio del problema y se centra en aspectos de éste visibles para el usuario.

En este punto, se reduce el nivel de abstracción y las clases de análisis se afinan hacía las clases de diseño, aportando los detalles que permitirán que las clases se implementen y generen una infraestructura para el software a desarrollar.

A la hora de realizar este refinamiento, se han de tener en cuenta una serie de características que aseguren que las clases de diseño quedan bien formadas [Arl02]:

1. *Completa y suficiente.* Una clase de diseño debe encapsular todos los atributos y métodos que sea razonable esperar y que existan para la clase.

2. *Primitivismo.* Los métodos asociados con una clase de diseño deben centrarse en el cumplimiento de un servicio para la clase. Una vez implementado el servicio con un método, la clase no debe proveer otro modo de hacer lo mismo.

3. *Mucha cohesión.* Una clase de diseño cohesiva tiene un conjunto pequeño y centrado de responsabilidades; para implementarlas emplea atributos y métodos de objetivo único.

4. *Poco acoplamiento.* Dentro del modelo de diseño, es necesario que las clases de diseño colaboren una con otra. Sin embargo, la colaboración debe mantenerse en un mínimo aceptable. Si un modelo de diseño está muy acoplado (todas las clases de diseño colaboran con todas las demás), el sistema es difícil

de implementar, probar y mantener con el paso del tiempo. En general, las clases de diseño dentro de un subsistema deben tener sólo un conocimiento limitado de otras clases.

Se ha de tener en cuenta que el sistema cuenta tanto con una capa de aplicación como con otra de servidor, con lo que se ha de modelizar la arquitectura de ambas capas.

Así, en la figura 3.40 se aporta el diagrama de clases de diseño para la capa del cliente del futuro sistema, si bien cabe señalar que el objetivo del diagrama de clases de diseño es el de modelizar la arquitectura básica del sistema a implementar; con lo que, para mantener la vista de dicha arquitectura útil y clara, no se incluyen clases, atributos o métodos que tengan un carácter accesorio o no aporten de manera directa a la consecución de los requisitos descritos en secciones anteriores.

De este modo, no se incluyen métodos ligados a inicializaciones de interfaces de usuario o atributos relativos a dichas interfaces, métodos básicos como los *get()* y *set()* o clases accesorias sin implicación directa en los casos de uso, entre otros.

Sobre el diagrama presentado, destaca el empleo de dos patrones de diseño:

1. **Singleton**. Patrón creacional utilizado con el objetivo de asegurarse de que una clase se instancia una única vez.

   Dada la naturaleza de la entidad *Usuario* según la cual una ejecución únicamente puede contar con un usuario activo a la vez (con posibles modificaciones sobre su estado), se considera que este patrón ayuda a forzar dicho objetivo, eliminando posibles errores o fallos futuros.

2. **Observer**. Patrón de comportamiento utilizado para implementar la actualización de un objeto (o varios) ante el cambio en el estado de otro.

   Dado que los datos se almacenan por unas entidades y se muestran por otras, es necesario que cada vez que se produzca una operación con el servidor y éste notifique con el resultado, las clases del cliente que provocaron dicha llamada se actualicen convenientemente mediante la modificación de su estado, mostrando un mensaje de error si aplica, etc. Con ello, dicho patrón se aplica sobre las clases que provocan llamadas directas al servidor y las que gestionan las mismas. La base del patrón es una interfaz a ser implementada por las clases a ser actualizadas y sobre la que se notificarán los cambios o respuestas del servidor por parte de las clases gestoras de las comunicaciones.

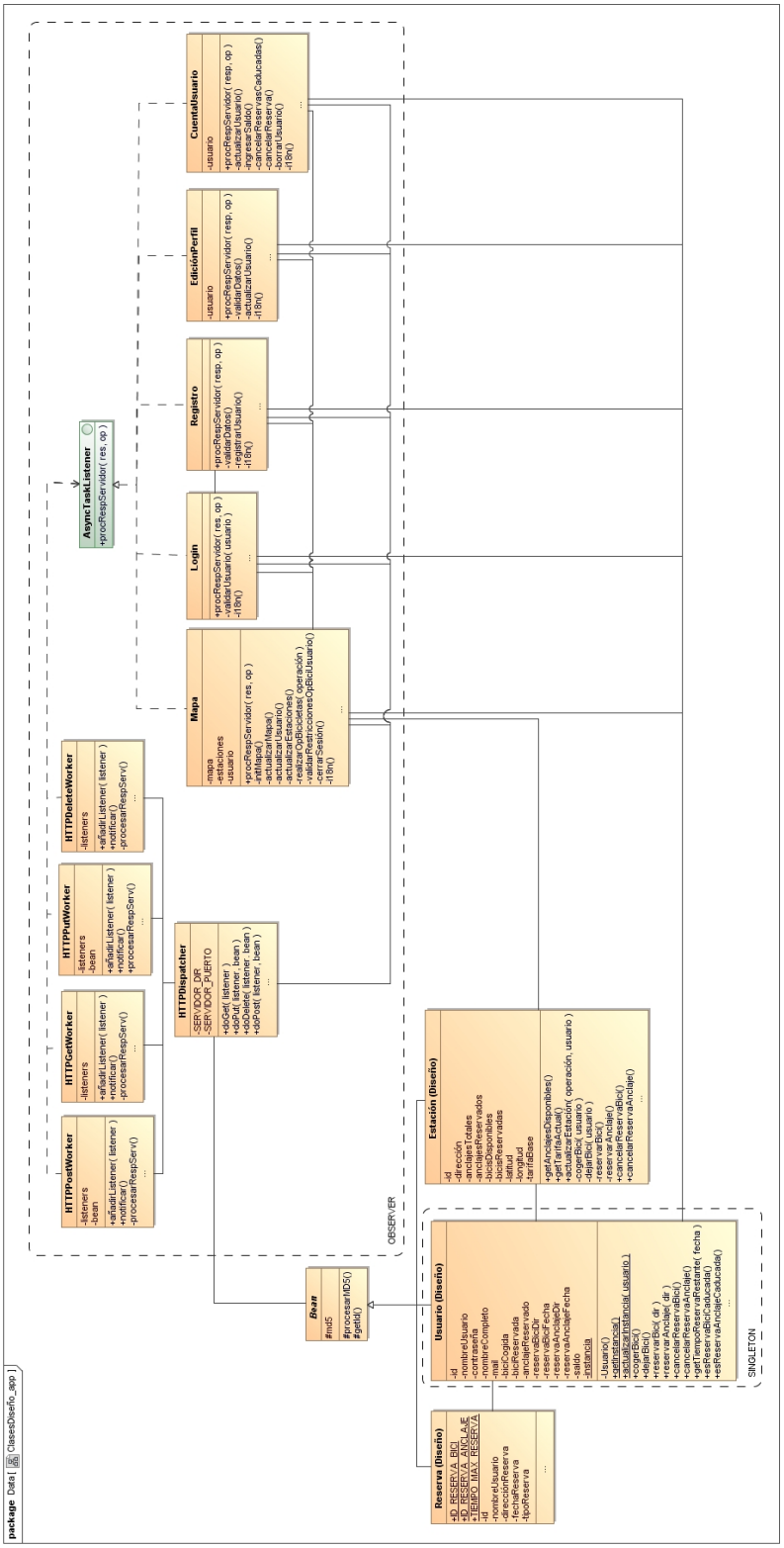


Figura 3.40: Diagrama de clases de diseño de la capa del cliente

Por su parte, en la figura 3.41 queda esquematizada la arquitectura del servidor.

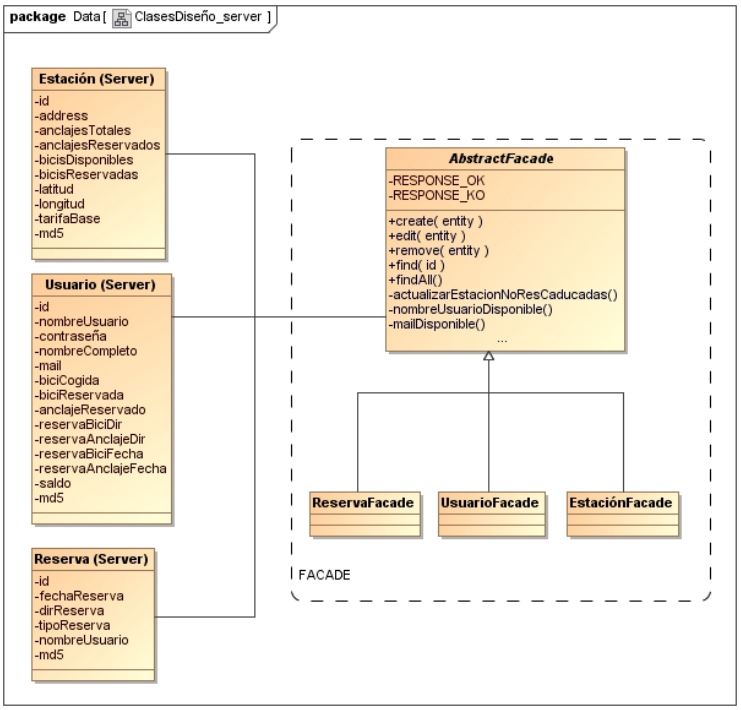


Figura 3.41: Diagrama de clases de diseño de la capa del servidor

En este caso, se identifica el patrón *Facade*, un patrón estructural utilizado para simplificar las interfaces o comunicaciones entre subsistemas.

Dado que el servidor supone la capa intermedia entre el cliente y la base de datos, este patrón ayuda a facilitar la comunicación entre ellos. El patrón se implementará partiendo de un clase abstracta de la que heredan las *fachadas* específicas de cada entidad, siendo estas especificaciones las que recibirán las llamadas remotas. Todas ellas contarán con los métodos CRUD básicos implementados en la fachada padre, que es la que finalmente realiza la conexión con la base de datos. Esta clase padre será la que implementará, a su vez, los métodos de control para todas las entidades, de modo que las fachadas hijas sólo se dedican a recibir llamadas y e invocar el método de la clase padre oportuno.

Por claridad en el esquema, en las fachadas hijas no se han incluido los métodos de comunicación básicos por tratarse de llamadas al método CRUD específico de la clase padre. Asimismo, en esta fachada padre se apuntan dichos métodos CRUD sin entrar en posibles adecuaciones a cada entidad, además de métodos operativos necesarios para la funcionalidad del sistema.

Se observa, a su vez, que la fachada padre ha de incluir dos mensajes estándar

de respuesta para indicar si la operación ha ido bien o no, de modo que el cliente (que también conoce estos mensajes) reaccione de manera oportuna.

### 3.3.3. Diseño de las interfaces

Una vez establecida la arquitectura básica sobre la que se fundamentará el sistema, el segundo apartado a definir en el modelo de diseño son las interfaces de comunicación y de usuario. Estos elementos permiten que el software se comunique externamente y permita la comunicación y colaboración internas entre los componentes que constituyen la arquitectura del software.

#### Interfaces de comunicación

Dentro de la arquitectura Cliente/Servidor se hace necesario definir la comunicación entre ambas capas. Si bien los detalles técnicos para su implementación se comentan en la siguiente sección 3.4, en este punto se aporta una capa de diseño que suponga una introducción conceptual.

Como quedó definido en la sección 3.1 dedicada a la ERS, la comunicación Cliente/Servidor se ha de regir de acuerdo a la tecnología RESTful Web Services, que se comentará más adelante. Esta tecnología parte de comunicaciones HTTP y alcanzan los diferentes métodos remotos mediante URLs diferenciadas. Así, se hace necesario definir una URL estándar que sirva de base para todas las concretas de cada método a ejecutar en el servidor. Se considera la siguiente plantilla:

$$http://[IP\_SERV]:[PORT]/BikesManager/rest/entities.[ENTITY]$$

donde $IP\_SERV$ indica la IP dirección del servidor, $PORT$ indica el puerto donde el servidor se encuentra escuchando y $ENTITY$ indica la entidad (Usuario, Estación o Reserva) que realiza la llamada remota.

A partir de esta URL general se formarán las específicas de la forma siguiente:

$$[BASE\_URL]/[METHOD]$$

donde $BASE\_URL$ es la URL básica indicada más arriba y $METHOD$ indica el método o parámetros que se añaden en la dirección para la llamada del método que sea necesario.

Una vez un método se ejecuta en el servidor, éste devolverá la cadena resultante al cliente. En caso de que se tenga que indicar si la operación se ha realizado correctamente o no, se enviarán cadenas del estilo $[ENTITY]\_SERVER\_OK$ o $[ENTITY]\_SERVER\_KO$, respectivamente.

Por su parte, las cadenas que se intercambian entre ambas capas seguirán el formato de texto JSON[4], la tecnología ha utilizar para la traducción de cadenas se comenta en la siguiente sección 3.4.

La selección de este formato en lugar del tradicional XML queda principalmente motivada por el hecho de que el primero es más sencillo a la hora compartir *datos tradicionales* como texto o números, que son los utilizados por el sistema a desarrollar, y no datos estructurados como podrían ser documentos, fotos, vídeos... donde XML es más eficiente.

Finalmente, comentar la utilización de los patrones *Observer* y *Facade* para mejorar y estandarizar la comunicación entre componentes ante las problemáticas específicas apuntadas anteriormente.

### Interfaces de usuario

La interfaz de usuario (IU en adelante) se considera como uno de los elementos más importantes de un sistema o producto basado en computadora. Una interfaz inadecuada perjudica la capacidad del usuario de aprovechar todo el potencial de una aplicación, pudiendo dar lugar al fracaso de un adecuado diseño e implementación.

De acuerdo a Pressman [Pre10], el diseño de la IU (o diseño de la *usabilidad*) crea un medio eficaz de comunicación entre los seres humanos y la computadora. Siguiendo un conjunto de principios de diseño de la interfaz, se identifican los objetos y acciones de ésta y luego crea una plantilla de pantalla que constituye la base del prototipo de la interfaz de usuario.

Este apartado está dedicado, por tanto, a desarrollar brevemente la decisiones y principios de diseño de IU de mayor relevancia adoptados para alcanzar una IU consistente y adecuada.

En primer término, el diseño general de cualquier IU se ha de basar en las conocidas como *Reglas de Oro* establecidas por Mandel [Man97]:

1. *Dejar el control al usuario.*
2. *Reducir la carga de memoria del usuario.*
3. *Hacer que la interfaz sea consistente.*

Cada una de las tres Reglas anteriores supone una agrupación de reglas más específicas y su estudio teórico queda fuera del alcance de este documento, que se limita a su nombrado para su aplicación práctica[5].

[4]Una introducción a este lenguaje se recoge en [JSON].

[5]El detalle de cada regla se puede consultar en el manual de Theo Mandel al respecto en [Man97].

Con la base anterior establecida, se ha de tener en cuenta que la IU se presentará sobre dispositivos móviles con sistema operativo Android, que presenta numerosa documentación oficial acerca del adecuado diseño de IUs[6]. Se hace necesario, por tanto, hacer un seguimiento de estos principios específicos para lograr una IU adecuada a la naturaleza del dispositivo sobre el que se despliega el sistema.

Una vez considerados los principios generales y los específicos para los elementos específicos de Android, la decisión de mayor relevancia en el diseño de IU en Android es la utilización de los llamados *Fragments* (o Fragmentos). Un Fragmento representa un comportamiento específico de una porción de la IU en una *Actividad*[7]; es decir, agrupa elementos de una interfaz (tanto el diseño gráfico como sus controles lógicos asociados) de modo que estos sean reutilizables entre diversos tamaños u orientaciones de pantalla.

En la figura 3.42 se aporta un esquema gráfico de la utilidad de estos elementos de diseño para una mejor comprensión.

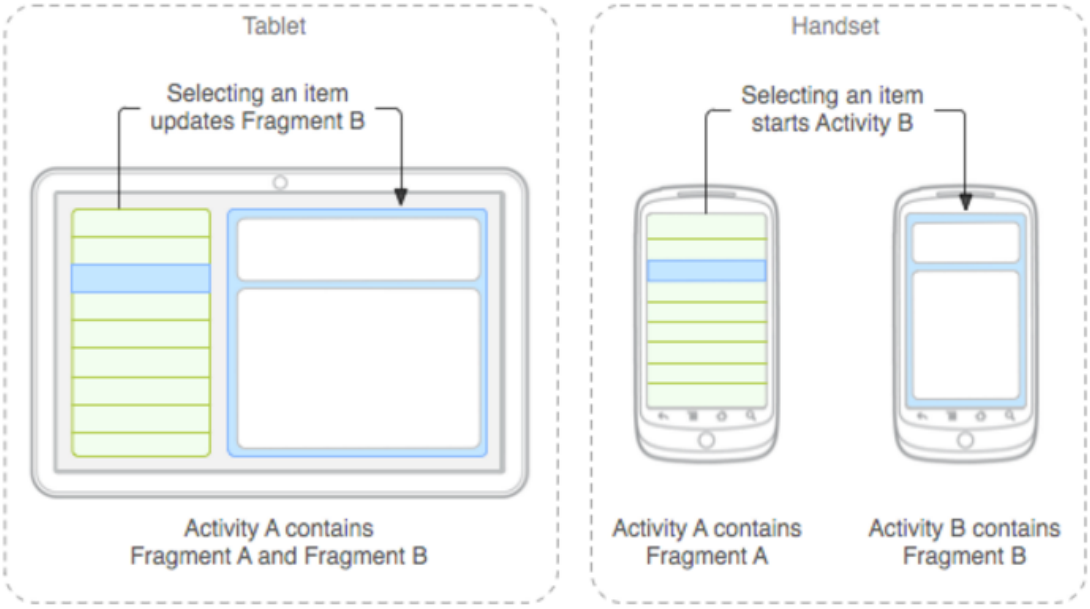


Figura 3.42: Fragmentos en el diseño de IUs *Fuente: [AnDev]*

De este modo, si bien el sistema se desarrollará para dispositivos móviles, la

[6]La documentación puede ser consultada en `https://developer.android.com/design/index.html`, a partir de [AnDev].

[7]Una Actividad es uno de los elementos básicos de cualquier aplicación Android. Sin entrar en detalles y por dar una primera aproximación, cada una de las pantallas o vistas de una aplicación suponen una Actividad. Más detalles en `https://developer.android.com/reference/android/app/Activity.html`, a partir de [AnDev].

utilización de Fragmentos posibilitará su adaptación a otros tamaños de pantalla de manera sencilla y simple.

Con todo, a modo de resumen, las decisiones o principios de diseño de IU adoptados son tres:

1. Las Reglas de Oro establecidas por Mandel.
2. Las normas de diseño Android para cada uno de sus componentes y elementos gráficos.
3. El empleo de Fragmentos para la agrupación y reutilización de elementos de IU entre diveros tamaños u operaciones de pantalla.

Estas decisiones regirán la construcción de la futura interfaz gráfica, proceso iterativo que consta de cuatro actividades estructurales [Man97]: 1) análisis y modelado, 2) diseño, 3) construcción y 4) validación.

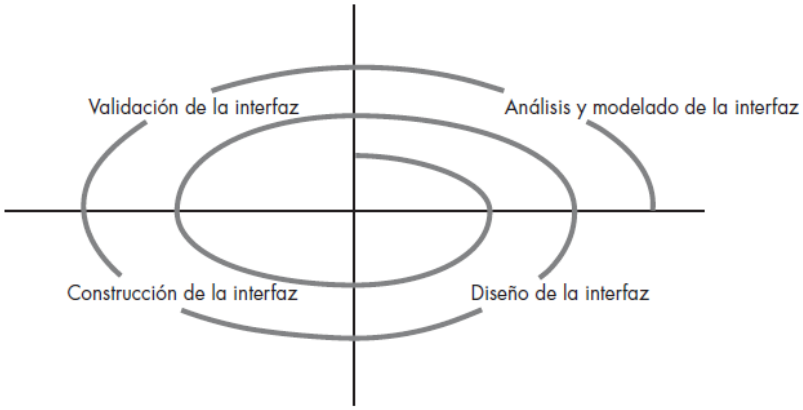


Figura 3.43: Proceso de diseño y construcción de una IU *Fuente: [Pre10]*

El *análisis de la interfaz* se centra en el perfil de los usuarios que interactuarán con el sistema; es decir, se trabaja para entender la percepción del sistema para cada clase de usuarios.

La meta *del diseño de la interfaz* es definir un conjunto de objetos y acciones de ésta (y sus representaciones en la pantalla) que permitan al usuario efectuar todas las tareas definidas cumpliendo con los requisitos de usabilidad y diseño básicos..

La *construcción de la interfaz* comienza por lo general con la creación de un prototipo que permite evaluar los escenarios de uso.

La *validación de la interfaz* se centra en: 1) la capacidad de la interfaz para implementar correctamente todas las tareas y requerimientos del usuario; 2) el grado en el que la interfaz es fácil de usar y de aprender y 3) la aceptación que tiene por parte del usuario.

## 3.4. Implementación

### 3.4.1. Introducción

El modelo de implementación indica la distribución física de los artefactos software que componen el sistema, cuya arquitectura ha quedado modelada en el apartado anterior.

Con el objetivo de desarrollar adecuadamente este modelo, se consideran dos apartados:

1. **Herramientas utilizadas**. Se enumeran y comentan brevemente las tecnologías de mayor relevencia utilizadas en la implementación del software. Sin ahondar en detalles, se pretende aportar una aproximación a las herramientas seleccionadas con el objetivo de establecer una adecuada relación entre el software modelado anteriormente y el hardware que lo desarrollará y soportará.

2. **Distribución física**. Una vez comprendidas las tecnologías a utilizar, se muestra la distribución física de sistema, de modo que se tenga una visión global del mismo con los nodos involucrados y la comunicación entre ellos.

### 3.4.2. Herramientas utilizadas

Dada la arquitectura Cliente/Servidor que soporta el sistema, las herramientas y tecnologías se pueden agrupar en cada una de sus capas (cliente, servidor y base de datos) y en la comunicación entre ellas.

Este apartado no pretende profundizar en detalles técnicos, sino enumerar las tecnologías de mayor relevancia presentes en el sistema para comprender adecuadamente la relación entre la parte lógica y física de la aplicación.

#### Cliente

El cliente del sistema queda pensado, en su primera versión, para ser desplegado en dispositivos móviles con sistema operativo Android. El entorno de desarrollo para la programación de esta capa de datos es Android Studio.

- **Android**. Android es un sistema operativo basado en un núcleo Linux diseñado principalmente para dispositivos móviles con pantalla táctil. Se lanzó inicialmente en 2008 y su versión estable más actual es la 7, *Nougat*[8].

[8]Tal y como se indicó en la sección de requisitos, Android denomina alfabéticamente y con nombres de postres o dulces a cada una de las versiones de su sistema operativo.

Las razones para la elección de este sistema, frente a otros como iOS o Windows, son principalmente la cobertura de mercado (en España, superior al 90 % en 2016[9]) y que el desarrollo de aplicaciones sobre esta plataforma es gratuito y queda basado en el lenguaje de programación Java, ampliamente extendido y con un fuerte soporte.

- **Android Studio**. Android Studio es el entorno de desarrollo integrado (IDE) oficial para el desarrollo de aplicaciones Android y se basa en IntelliJ IDEA[10]. Además del editor de códigos y las herramientas para desarrolladores de IntelliJ, Android Studio ofrece más funciones para la mejora de la productividad durante la compilación de apps para Android[11].

Como se ha comentado, Android actualmente tiene desplegada la versión 7. Cada una de estas nuevas versiones incorpora nuevas funcionalidades y desestima o sustituye otras de versiones previas. Esto provoca que, a la hora de desarrollar una aplicación, se tenga que tener en cuenta el grado de compatibilidad de la misma con las versiones previas para lograr un alcance y cobertura suficientes. Para ello, Android prevé la utilización de numerosas *librerías de compatibilidad*, que posibilitan la utilización de nuevos componentes o funcionalidades en versiones anteriores que, originalmente, no contaban con ellos. Del mismo modo, a la hora de crear una aplicación Android, se ha de dejar especificado la API mínima (y máxima) sobre la que dicha aplicación puede ser ejecutada (de modo que GooglePlay ofrezca aplicaciones adaptadas a la versión de cada dispositivo).

Para la aplicación desarrollada en este documento se ha seleccionado una compatibilidad hasta la versión 4.1 *Jelly Bean* (API 16), que alcanza una cobertura total del 97,5 % respecto al total de dispositivos repartidos por el mercado a nivel global[12], tal y como se muestra en la figura 3.44.

### Servidor

La capa intermedia del sistema queda soportado por el servidor de software libre GlassFish. El entorno de desarrollo empleado en este caso es NetBeans.

- **GlassFish**. GlassFish es un servidor basado en Apache Tomcat para el desarrollo y ejecución de aplicaciones regidas por la especificación *Java Platform,*

[9]Dato a junio de 2016: `http://cincodias.com/cincodias/2016/06/15/tecnologia/1465987235_750846.html`

[10]IDE para el desarrollo de sistemas informáticos desarrollado por JetBrains, más detalles en `https://www.jetbrains.com/idea/`.

[11]El detalle sobre estas funciones adicionales se puede consultar en `https://developer.android.com/studio/intro/index.html`, a partir de [AnDev].

[12]Dato a diciembre de 2016, Android actualiza esta información periódicamente en `https://developer.android.com/about/dashboards/index.html`, a partir de [AnDev].

| Version | Codename | API | Distribution |
|---|---|---|---|
| 2.2 | Froyo | 8 | 0.1% |
| 2.3.3 - 2.3.7 | Gingerbread | 10 | 1.2% |
| 4.0.3 - 4.0.4 | Ice Cream Sandwich | 15 | 1.2% |
| 4.1.x | Jelly Bean | 16 | 4.5% |
| 4.2.x | | 17 | 6.4% |
| 4.3 | | 18 | 1.9% |
| 4.4 | KitKat | 19 | 24.0% |
| 5.0 | Lollipop | 21 | 10.8% |
| 5.1 | | 22 | 23.2% |
| 6.0 | Marshmallow | 23 | 26.3% |
| 7.0 | Nougat | 24 | 0.4% |

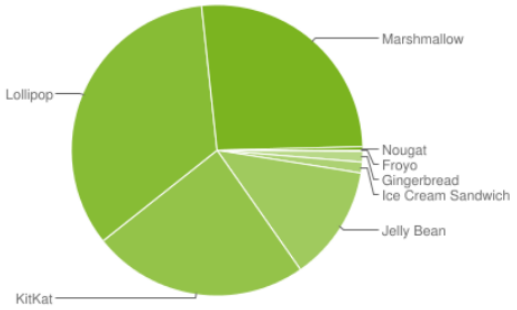


Figura 3.44: Datos oficiales Android de distribución de versiones a diciembre de 2016

*Enterprise Edition* (Java EE platform)[13] y tecnologías web basadas en Java.

El motivo para la elección de este servidor es su carácter gratuito y de código libre, además del hecho de que, a diferencia de Tomcat, su instalación básica incluye un conjunto amplio de librerías, suficiente para cubrir las necesidades del sistema, sin que con ello se vea afectado su rendimiento de manera perceptible respecto a otras opciones, como el citado Apache Tomcat.

- **NetBeans**. NetBeans es el IDE oficial para Java 8, supone un entorno de desarrollo de libre distribución y queda pensado principalmente para el lenguaje de programación Java.

  La utilización de este entorno viene motivada principalmente por el soporte altamente desarrollado que tiene para la implementación de aplicaciones distribuidas, controlando desde el IDE las conexiones con servidor y base de datos y construyendo automáticamente el esqueleto de la aplicación a desplegar sobre dicho servidor (dependiendo de la naturaleza de la misma, como una basada en Web Services, por ejemplo).

[13]Plataforma de programación que incluye varias especificaciones para el desarrollo de aplicaciones distribuidas y empresariales, más detalles en [OraEE].

### Base de datos

Capa de datos de la aplicación, implementada sobre un sistema MySQL y apoyado por el entorno MySQL Workbench.

- **MySQL Server**. MySQL es un sistema de gestión de bases de datos relacional y de código abierto. Tradicionalmente se ha enfocado a aplicaciones web donde la principal preocupación es la optimización de consultas sencillas, aspecto compartido por la aplicación en desarrollo.
- **MySQL Workbench**. MySQL Workbench es una herramienta visual de diseño de bases de datos que integra un conjunto de funcionalidades para facilitar la gestión de bases de datos MySQL.

### Comunicación

Dado su carácter transversal, se ha decidido crear un grupo separado de tecnologías relativas a la comunicación entre capas de datos, si bien cada una puede quedar implementada en una u otra capa de la arquitectura.

- **Jackson**. Librería implementada en la capa cliente para la traducción de cadenas JSON a objetos Java y viceversa. El empleo de esta librería frente a otras es debido fundamentalmente a que esta librería suele dar mejores índices de rendimiento a medida que los ficheros a traducir van creciendo, aspecto potencialmente preferible[14]..
- **RESTful**. Servicios web basados en la arquitectura REST[15] utilizados para el intercambio de datos entre el cliente y el servidor.

  Se ha decidido utilizar este tipo de servicios web frente a otros como SOAP por varias razones:

  - *Simpleza*. REST hace uso del protocolo estándar HTTP, lo que desemboca en una facilidad de desarrollo y entendimiento superior a SOAP. Ésto facilita, adicionalmente, su mantenimiento posterior. En general, hay pocas cosas que REST no haga de un modo más sencillo que SOAP.

[14] En Internet se pueden identificar numerosos estudios de rendimiento al respecto, todos con las mismas conclusiones mencionadas, se aporta uno de ellos: `https://dzone.com/articles/compare-json-api`

[15] Arquitectura que, haciendo uso del protocolo HTTP, proporciona una API que utiliza cada uno de sus métodos (GET, POST, PUT, DELETE, etc.) para poder realizar diferentes operaciones entre la aplicación que ofrece el servicio web y el cliente. Una introducción teórica se puede encontrar en `https://es.wikipedia.org/wiki/Transferencia_de_Estado_Representacional`.

- *Amplitud de formatos*. REST permite multitud de formatos, mientras que SOAP sólo acepta XML. Aspecto de relevancia considerando, además, que JSON es más eficiente y sencillo que XML para el sistema en desarrollo, como ya se comentó en el apartado 3.3.3.
- *Rendimiento y escalabilidad*. REST presenta mejores métricas de rendimiento y escalabilidad respecto a SOAP.

Las características anteriores han hecho que REST se imponga a SOAP como servicio web dominante. De hecho, compañías como Yahoo o Google hacen uso de REST para todos sus servicios.

- **MySQL Connector / JDBC**. Conector estándar para la comunicación entre el servidor y la base de datos utilizando la API *Java Database Connectivity* (JDBC)[16].

### 3.4.3. Distribución física

Una vez definidas las principales tecnologías utilizadas para la implementación del sistema, queda modelar la distribución física de los artefactos software presentes en el mismo con el objetivo de tener una visión general del soporte físico del sistema.

En la figura 3.45 se esquematiza la distribución mencionada. Se identifican tres nodos, distribución acorde a la arquitectura Cliente/Servidor utilizada:

- **Nodo Cliente**. Representa la capa de presentación del sistema, implementado en el dispositivo Android. Este nodo recoge la arquitectura de la aplicación modelada anteriormente en la figura 3.40 y es el utilizado por el usuario para interactuar.
- **Nodo Servidor**. La capa intermedia de la arquitectura, desplegada sobre GlassFish. Recoge la arquitectura del servidor modelada en la figura 3.41 y supone la capa que comunica la capa de presentación con la de datos. La comunicación con el cliente se realiza mediante el protocolo HTTP con los servicios web RESTful y el lenguaje JSON.
- **Nodo Base de Datos**. La capa de datos soportada por el sistema gestor MySQL. La comunicación con el servidor se realiza mediante los conectores estándar MySQL Connector y la API JDBC.

[16] API que permite la ejecución de operaciones sobre bases de datos desde el lenguaje de programación Java utilizando el dialecto SQL, más detalles en [OraDB].

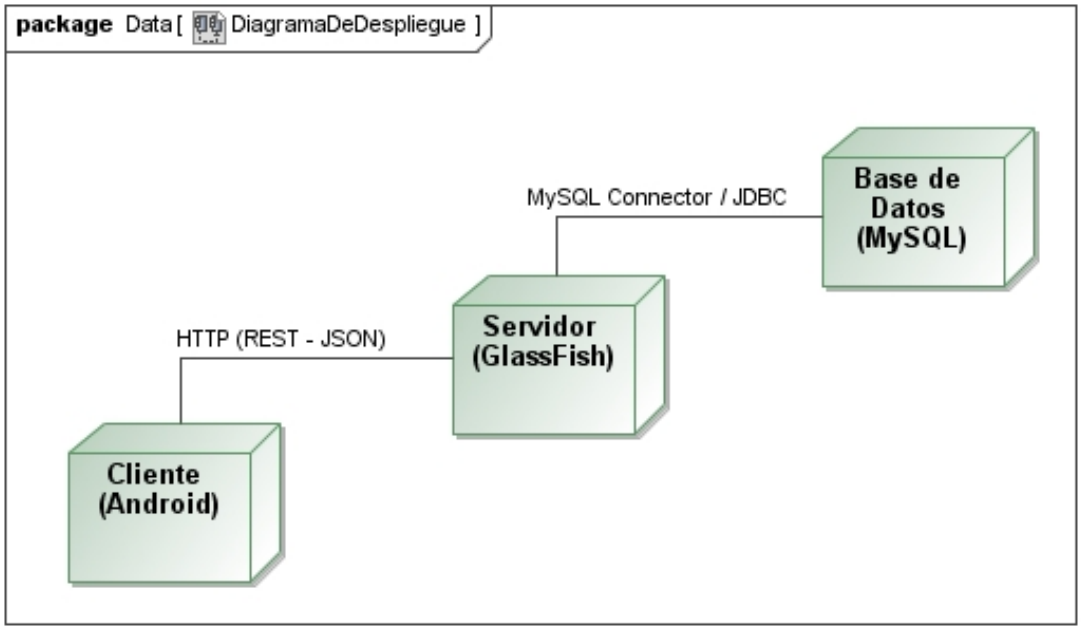


Figura 3.45: Diagrama de despliegue del sistema

# Capítulo 4

# Pruebas

## 4.1. Introducción

Mediante la actividad conocida como *aseguramiento de la calidad del software*[1] se busca conseguir unos mínimos de calidad sobre las diversas etapas de ingeniería del software por las que pasa el producto: requerimientos, diseño e implementación (o código). Sin embargo, en este proceso pueden pasar errores inadvertidos que, sin unas pruebas adecuadamente planteadas y planificadas permanecerían en el sistema construido hasta su publicación.

Las pruebas representan, por tanto, la última oportunidad para valorar la calidad y, desde un punto de vista más práctico, descubrir errores. Sin embargo, citando a Pressman [Pre10], "no se puede probar la calidad. Si no está ahí antes de comenzar las pruebas, no estará cuando termine de probar". Es decir, la calidad se ha de incorporar al software a lo largo de todo el proceso ingeniería, de modo que quede confirmada durante la etapa de pruebas.

Por tanto, el software se prueba para descubrir errores que se cometieron de manera inadvertida conforme se diseñó y construyó.

Las pruebas de software forman parte de un tema más amplio conocido como verificación y validación. La *verificación* se refiere al conjunto de tareas que garantizan que el software implementa correctamente una función específica. La *validación* es un conjunto diferente de tareas que aseguran que el software que se construye sigue los requerimientos establecidos en las etapas iniciales. La estrategia de pruebas se deberá enfocar a cubrir ambos aspectos: el sistema ha de funcionar adecuadamente de acuerdo a las funciones marcadas en la ERS (sección 3.1).

[1] Para conocer más en detalle esta actividad se puede recurrir a [Pre10].

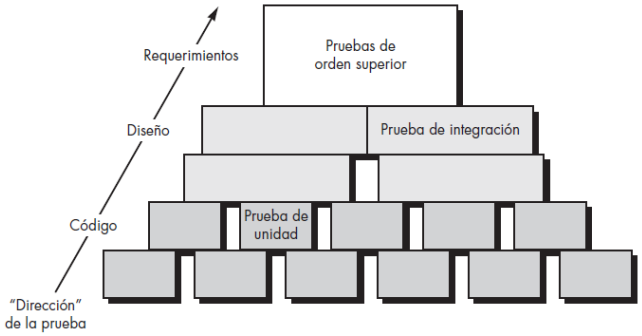


Figura 4.1: Pasos de las pruebas software *Fuente: [Pre10]*

## 4.2. Estrategia de pruebas

Una estrategia de pruebas software proporciona una guía que incorpora la planificación de la prueba, el diseño de casos de prueba, la ejecución de la prueba y la recolección y evaluación de los resultados.

Dentro del contexto de la Ingeniería del Software, las pruebas suponen una serie de cuatro pasos que se implementan de manera secuencial, tal y como se muestra en la figura 4.1.

Inicialmente, las pruebas se enfocan en cada componente de manera individual, son las *pruebas de unidad* y garantizan que todo módulo de software funciona adecuadamente como unidad, de ahí el nombre. A continuación, los componentes deben ensamblarse e integrarse para formar el paquete de software completo. La *prueba de integración* aborda los conflictos asociados con los problemas de verificación y construcción de programas. Después de integrar (construir) el software, se realiza una serie de *pruebas de orden superior*, donde se identifican las *pruebas de validación*, que proporcionan la garantía final de que el software cumple con todos los requerimientos funcionales y de compartimento abordados en la fase de requisitos, y las *pruebas del sistema*, para verificar que todos los elementos se mezclan de manera adecuada y que se logra el funcionamiento/rendimiento global del sistema deseado.

En este punto, cabe mencionar los atributos con que cuenta una "buena" prueba [Kan99]:

- *Una buena prueba tiene una alta probabilidad de encontrar un error.* Para ello, es conveniente que el examinador conozca el software para saber dónde podría fallar y probar dichos caminos.

- *Una buena prueba no es redundante.* Cada prueba ha debe tener un propósito diferente a otra.

- *Una buena prueba debe ser "la mejor de la camada"* [Kan99]. Para un conjunto de pruebas similares, se debe recurrir a la que tenga la mayor probabilidad de descubrir errores.

- *Una buena prueba no debe ser demasiado simple o demasiado compleja.* En general, cada prueba debe ejecutarse por separado.

Considerando el proceso descrito y las características mencionadas, en los apartados siguientes se aportarán los elementos más destacables empleados para probar la aplicación construida; si bien en este documento sólo se indicará el detalle concreto de las *pruebas de validación*, por ser aquellas que se centran en la funcionalidad específica del sistema y cuyo éxito depende del de las pruebas previas.

### 4.2.1. Pruebas de unidad

La *prueba de unidad* enfoca los esfuerzos de verificación en la unidad más pequeña del diseño de software: el componente o módulo de software, enfocándose en la lógica de procesamiento interno y de las estructuras de datos dentro de sus límites.

Las validaciones se han realizado siguiendo principalmente los métodos de *caja blanca*, filosofía de diseño de casos de prueba que [Pre10]: 1) garanticen que todas las rutas independientes dentro de un módulo se revisaron al menos una vez, 2) revisen todas las decisiones lógicas en sus lados verdadero y falso, 3) ejecuten todos los bucles en sus fronteras y dentro de sus fronteras operativas y 4) revisen estructuras de datos internas para garantizar su validez.

Sin entras en detalles concretos, a continuación se mencionan las herramientas fundamentales utilizadas para realizar estas pruebas sobre los diferentes elementos del sistema.

#### Cliente - Android

La utilidad de mayor relevancia empleada para probar la aplicación Android de manera individual ha sido *Logcat*. Consiste en una herramienta de línea de comandos que vuelca un registro de mensajes del sistema, incluidos los seguimientos de pila, los casos de error del sistema y los mensajes escritos por el programador desde la app con la clase *Log*. Muestra los mensajes en tiempo real y mantiene un histórico para su consulta en caso de necesidad.

Con esta herramienta se puede hacer un seguimiento exhaustivo del estado de cada componente, estructura de datos o variable utilizada. Las posibilidades ofrecidas por la clase *Log* hacen que dicho seguimiento se pueda realizar de manera

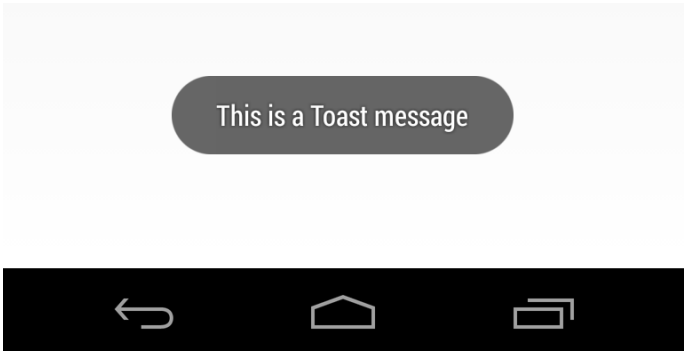


Figura 4.2: Ejemplo de un toast en Android

ordenada y sencilla[2].

Adicionalmente a Logcat, y desde un punto de vista de abstracción mayor, también se ha recurrido a la clase *Toast*. Un *toast* proporciona un *feedback* sencillo acerca de una operación mediante un mensaje de texto que se muestra durante unos instantes en la pantalla. Mientras que Logcat recurre a la línea de comandos del sistema, los *toast* se muestran en el propio dispositivo, siendo muy convenientes para mostrar al usuario el resultado de alguna operación que haya realizado[3], en la figura 4.2 se aporta un ejemplo de uso de esta utilidad.

### Servidor - GlassFish

El servidor queda programado en Java y como herramientas básicas de depuración se utilizaron las habituales escrituras en línea de comandos mediante la clase *System* y sus atributos *out* y *err*[4], de modo que se pudiese hacer un seguimiento paso a paso del estado del elementos contenidos en esta capa de datos.

Adicionalmente, para la prueba de los diferentes módulos implementados en el servidor de manera aislada a la aplicación móvil, se ha recurrido a un complemento del navegador web llamado *RESTClient*, un depurador para servicios web RESTful que permite realizar consultas tal y como las haría la aplicación. En la figura 4.3 se aporta una captura de pantalla del depurador.

Este tipo de prueba se podría considerar una prueba de mayor nivel al incluir también una conexión con la base de datos (la invocación de un servicio web

[2]Para más detalles acerca de Logcat se puede consultar `https://developer.android.com/studio/debug/am-logcat.html`, y acerca de la clase Log `https://developer.android.com/reference/android/util/Log.html`; ambos a partir de [AnDev].

[3]Como en la nota anterior, para más detalles acerca de Toast se puede consultar `https://developer.android.com/guide/topics/ui/notifiers/toasts.html`, a partir de [AnDev].

[4]Más detalles en `https://docs.oracle.com/javase/7/docs/api/java/lang/System.html`, a partir de [DocOr]

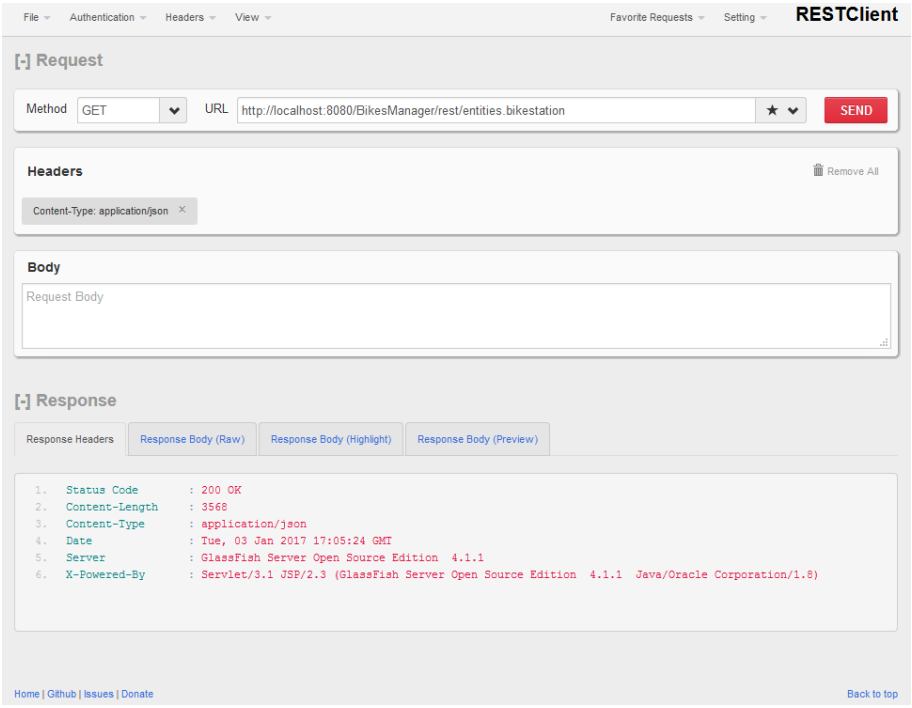


Figura 4.3: RESTClient, depurador de servicios web RESTful

finaliza mediante la obtención de un resultado al que se llega mediante la lectura o modificación de la base); sin embargo, considerando que el primer objetivo con el que se recurrió a esta herramienta fue el de ejercitar, bajo un entorno controlado, los diferentes métodos del servidor (los servicios web individuales), se considera adecuado incluirla en esta sección.

### Base de datos - MySQL

De manera individual, sobre la base de datos se han realizado consultas SQL directas mediante de la herramienta *MySQLWorkbench*, mencionada en la sección 3.4.2.

Cabe señalar que este tipo de pruebas sobre la capa de base de datos pueden no tener tanto interés como otras, puesto que dicha capa carece de lógica implementada para ser probada. El objetivo principal es el de probar la precisión y la integridad de los datos para asegurar que se almacenen, actualicen y recuperen de manera adecuada.

### 4.2.2. Pruebas de integración

El objetivo de las *pruebas de integración* es tomar los componentes probados de manera individual y construir la arquitectura señalada en la etapa de diseño. Dos han sido los enfoques fundamentales que se han seguido durante el desarrollo de este tipo de pruebas:

1. **Integración ascendente**. Enfoque principal seguido, se comienza por la verificación de módulo atómicos y se van integrando y construyendo hacia niveles superiores.

2. **Pruebas de humo**. Enfoque secundario al anterior utilizado para agregados o arreglos no previstos, supone una integración constante según la cual el software se reconstruye (con el agregado de nuevos componentes) y se prueba día a día.

Los métodos para realizar las pruebas sobre los módulos y la integración entre ellos se han basado principalmente en las técnicas de pruebas de *caja negra*, enfocadas en los requerimientos funcionales del software. Es decir, las pruebas de caja negra intentan encontrar errores en las categorías siguientes [Pre10]: 1) funciones incorrectas o faltantes, 2) errores de interfaz, 3) errores en las estructuras de datos o en el acceso a bases de datos externas, 4) errores de comportamiento o rendimiento y 5) errores de inicialización y terminación.

Las herramientas utilizadas para completar este tipo de pruebas son las mismas que las empleadas en las pruebas de unidad, la diferencia está en que esté nivel tiene un enfoque más amplio, considerando los componentes y sus interrelaciones.

### 4.2.3. Pruebas de validación

**Especificación de las pruebas planteadas**

**Resultados**

### 4.2.4. Pruebas de sistema

La *prueba del sistema* es una serie de diferentes pruebas cuyo propósito principal es ejercitar por completo, saliendo de la Ingeniería del Software y entrando en la de Sistemas, el sistema construido. Algunos de los ejemplos más representativos son [Pre10]:

- **Pruebas de recuperación**, dedicadas a la capacidad de relanzamiento (autónomo o con intervención humana) del software ante fallos que provoquen momentos de inactividad.

- **Pruebas de seguridad**, para asegurar el sistema ante ataques o intentos de entrada impropios.
- **Pruebas de esfuerzo**, destinadas a evaluar el comportamiento del sistema ante una demanda anormal de recursos.
- **Pruebas de rendimiento**, similares a las anteriores, estas pruebas están centradas en obtener unas métricas de funcionamiento mínimas.
- **Pruebas de despliegue o configuración**, para aquellos sistemas destinados a implementarse en varias plataformas, estas pruebas evalúan su desempeño en cada una de ellas.

Dado el tipo de pruebas encuadradas en este epígrafe, y considerando la naturaleza académica del proyecto aquí presentado, las *pruebas de sistema* quedan fuera del alcance de la estrategia de pruebas para la aplicación construida. Algunas de las anteriores, sin embargo, sí se pueden asimilar a ciertas *pruebas de validación* antes mencionadas, como la evaluación del correcto funcionamiento en diferentes plataformas como *prueba de despliegue*, o el impedimento de acceso a un usuario no registrado como *prueba de seguridad*.

# Capítulo 5

# Conclusiones

NO ME GUSTA MUCHO ESTO, NO SÉ SI LAS CONCLUSIONES SE ENFOCAN TANTO A LO PERSONAL

El presente proyecto ha supuesto la elaboración de una aplicación Android completa comunicada de manera remota con su servidor y base de datos mediante el estándar REST de los servicios web.

Dadas las diferentes tecnologías y capas de datos utilizadas, el conocimiento técnico adquirido ha sido amplio, permitiéndome conocer, comparar y seleccionar diferentes soluciones de bases de datos, servidores, transmisión y serialización de datos... dando lugar a una visión final del desarrollo software para dispositivos Android bastante sólida y completa.

Desde el punto de vista de las capacidades personales, cabría destacar la mejora y asentamiento de las capacidades propias de organización, análisis y síntesis, que me han permitido llevar a buen puerto la herramienta desarrollada.

Más allá de los apartados personales, se espera que la herramienta pueda ser utilizada como base de estudio para la mejora de los servicios de gestión de parques públicos de bicicletas... SEGUIR AÑADIENDO COSAS...

## 5.1. Líneas futuras

La aplicación supone una base tecnológica para el desarrollo y mejora de los servicios de gestión de parques públicos de bicicletas.

De manera concreta, se consideran las siguientes líneas futuras de desarrollo y estudio:

- Desarrollo de la adaptación de precios a la hora de coger, dejar o reservar bicicletas, dependiendo de la disponibilidad de cada estación.

- Desarrollo de un sistema de alertas automático que avise al usuario de la disponibilidad de bicicletas o anclajes en una estación elegida por él.
- Desarrollo de versiones sobre diferentes plataformas (web, tablet, etc.), con el objetivo de llegar a un público más amplio.

# Apéndice A

# Manual de usuario

Aquí podemos poner un manual de usuario.

# Bibliografía


[AnDev] Portal de Android Developers en *https://developer.android.com*.

[Arl02] Arlow, J. y I. Neustadt, *UML and the Unified Process*, Addison-Wesley, 2002.

[Bas03] Bass, L., P. Clements y R. Kazman, *Software Architecture in Practice*, 2a. ed., Addison-Wesley, 2003.

[Bei90] Beizer, B., *Software Testing Techniques*, 2a. ed., Van Nostrand-Reinhold, 1990.

[Coa91] Coad, P. y E. Yourdon, *Object-Oriented Analysis*, 2a. ed., Prentice Hall, 1991.

[Coo00] Cooper, J. W., *Java Design Patterns: A Tutorial*, Addison-Wesley Professional, 2000.

[DocOr] Portal de documentación técnica Oracle sobre Java en *http://docs.oracle.com/javase*.

[EckPa] Eckel, B. *Thinking in Patterns*.

[Gar87] Garvin D., *Competing on the Eight Dimensions of Quality*, Harvard Business Review, noviembre 1987.

[GF13] *GlassFish Server Open Source Edition - Quick Start Guide*, 4.0, 2013.

[IE830] Institute of Electrical and Electronics Engineers (IEEE), *IEEE Std. 830-1998: Especificaciones de los Requisitos del Software*.

[ISO25] International Organization for Standardization (ISO), *ISO/IEC 25010:2011 - Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models*.

[JSON] "Introducción a JSON" en *http://www.json.org/json-es.html*.

[Kan99] Kaner, C., J. Falk y H. Q. Nguyen, *Testing Computer Software*, 2a. Rev. ed., John Wiley & Sons, 1999.

[Man97] Mandel, T., *The Elements of User Interface Design*, John Wiley & Sons, 1997.

[McC77] McCall, J., P. Richards y G. Walters, *Factors in Software Quality, Volume I. Concepts and Definitions of Software Quality*, 1977.

[Mye11] Myers, G., C. Sandler, y T. Badgett, *The Art of Software Testing*, 3a. ed., John Wiley & Sons, 2011.

[NetRF] "Getting Started with RESTful Web Services" en *https://netbeans.org/kb/docs/websvc/rest.html*.

[Oet15] Oetiker T. et al, *The Not So Short Introduction to LaTeX 2ε*, 5.05, 2015.

[OraDB] "Java SE Technologies - Database" en *http://www.oracle.com/technetwork/java/javase/jdbc/index.html*.

[OraEE] "Java EE at a Glance" en *http://www.oracle.com/technetwork/java/javaee/overview/index.html*.

[Pre10] Pressman, R. S., *Ingeniería del software: un enfoque práctico*, 7a. ed., McGrawHill, 2010.

[Sha96] Shaw, M. y D. Garlan, *Software Architecture*, Prentice Hall, 1996.

[WREST] "Representational state transfer" en *https://en.wikipedia.org/wiki/Representational_state_transfer*.

[WSOAP] "SOAP" en *https://en.wikipedia.org/wiki/SOAP*.

[WWSer] "Web Service" en *https://en.wikipedia.org/wiki/Web_service*.