UNIVERSIDAD DE MURCIA
FACULTAD DE INFORMÁTICA
Máster en Nuevas Tecnologías en Informática

Infraestructura de Simulación Social para el Testado y Validación Previa al Despliegue de Aplicaciones Android

Memoria del Trabajo Fin de Máster

Itinerario de computación ubicua

Autor:
Pablo Campillo Sánchez
pablocampillo@um.es

Tutores:
Juan A. Botía Blaya
Alberto García Sola

Murcia, Septiembre de 2011

ÍNDICE GENERAL

1. INTRODUCCIÓN 3

2. OBJETIVOS 5

3. ESTADO DEL ARTE 7

4. TAXONOMÍA DE LAS APLICACIONES PARA MÓVILES 10

INTERACCIÓN DEL TELÉFONO INTELIGENTE CON EL EXTERIOR 11

5. LA PLATAFORMA ANDROID 13

SERVICIOS BASADOS EN LA POSICIÓN 13
SENSORES EN ANDROID 14
Sensor NFC 15
EL MARCO DE TRABAJO DEL AUDIO Y EL VÍDEO 15
Grabar audio 16
La cámara para recibir imágenes 16
MECANISMOS DE COMUNICACIÓN ENTRE APLICACIONES 17
HERRAMIENTAS PARA LA PRUEBA DE APLICACIONES EN ANDROID 17

6. ARQUITECTURA DEL SISTEMA 19

MECANISMO DE COMUNICACIÓN: R-OSGI 21
EL SIMULADOR: UBIKSIM 22
EL TELÉFONO ANDROID O EL EMULADOR 24
Marco de trabajo de OSGi móvil 24
Capa Android de validación 26

7. EJEMPLO PRÁCTICO 29

MODELADO DEL MUSEO 29
TRABAJO RELATIVO A LOS CÓDIGOS QR 31
DESARROLLAR LA APLICACIÓN DE LECTURA DE CÓDIGOS QR (ZXING) 32
PUESTA EN MARCHA DE LA INFRAESTRUCTURA DE VALIDACIÓN 33

8. PROYECTO REAL EN EL MAM 35

9. CONCLUSIONES Y TRABAJO FUTURO 39

RESUMEN EXTENDIDO

El trabajo de investigación consiste en el diseño de una infraestructura que permita a un usuario validar aplicaciones para móviles y también el sistema en sí, es decir, el entorno y la interacción entre los elementos. La validación se refiere a cualquier tarea orientada a comprobar que lo que se crea se comporta como esperamos. En este trabajo nos centramos en la validación de las aplicaciones para móviles. Esta validación puede llegar a ser muy costosa si se tiene que realizar a través de una prueba de campo, es decir, en su entorno natural, donde este entorno es muy grande, por ejemplo, un aeropuerto. La solución que se plantea es utilizar la simulación híbrida donde hay elementos que se simulan, en nuestro caso el entorno, y otros que son reales, el móvil y la persona. El usuario se desenvuelve por el mundo virtual a través de su personaje virtual que controla con el teclado y el ratón. El personaje virtual dispone de un móvil virtual que es controlado por el usuario a través del móvil real donde está instalada la aplicación a validar. Por tanto, la infraestructura se compone de un simulador (UbikSim) y de un sistema en el móvil que permita comunicar la aplicación a validar con el simulador. Se utiliza una herramienta para crear los mundos virtuales, Ubik Editor.

El objetivo principal, y la mayor aportación del trabajo, consiste en no tener que modificar prácticamente el código fuente de la aplicación de su versión de validación a la de comercialización. Este requisito no se puede cumplir para los dos sistemas operativos para móviles más importantes, Android e iOS. Por ello, la infraestructura está dirigida a Android por ofrecer una plataforma de código abierto a diferencia de iOS que es totalmente cerrada en su versión oficial teniendo que programar en C#. Además, aunque el ámbito de las aplicaciones que nos interesa está relacionado con la computación ubicua y, concretamente, las aplicaciones sociales basadas en la posición, nuestra infraestructura es de propósito general y las aplicaciones Android se desarrollan siguiendo la misma metodología que la del resto.

En el trabajo hemos diseñado la infraestructura de forma general, sin entrar en mucho detalle, y hemos implementado parte de ésta para poder mostrar la metodología a seguir para crear una aplicación a través de un ejemplo. El ejemplo consiste en una aplicación para un museo que lee códigos QR asociados a cada cuadro. Estos códigos contienen direcciones a entradas de la wikipedia con información sobre la obra.

Palabras clave: Validación, Android, OSGi, Simulación híbrida, emulador, contexto.

1. INTRODUCCIÓN

Las aplicaciones móviles están provocando una revolución en los últimos años. Y esta revolución la han llevado a cabo principalmente dos compañías, en principio con enfoques parecidos, a través de sus sistemas operativos, iOS y Android. iOS (iPhone Operative System) es un sistema operativo cerrado que viene incorporado en los dispositivos fabricados por Apple. Por ello, las aplicaciones desarrolladas con este sistema se tienen que limitar y ajustar a las políticas de Apple. Android es un sistema operativo diseñado por Google y sigue una filosofía de desarrollo totalmente diferente: código abierto y desarrollo basado en Java. De acuerdo con los informes de Fall 2010[1], Android es el segundo sistema operativo para móviles más usado, por encima de BlackBerry e iOS, pero se prevee que para 2014 supere al número uno, el sistema operativo Symbian de Nokia [3]. De hecho, desde 2009, Android tiene la plataforma con el mayor número de aplicaciones para descargar [24]. A día de hoy, las aplicaciones que se desarrollan para cualquiera de las dos plataformas, se pueden probar, verificar y validar usando técnicas convencionales de aplicaciones de escritorio. Para la prueba de aplicaciones sí existen herramientas específicas y a medida para cada plataforma, que no distan mucho de un simulador de eventos para un único usuario.

Nosotros estamos interesados en aplicaciones sociales para móviles, que implican a un gran número de usuarios y una coordinación entre ellos a través de una interacción directa. Un proyecto de ingeniería del software se suele componer de varias fases: análisis, diseño, implementación, prueba, verificación y validación; y nos interesa específicamente esta última. La validación se refiere a cualquier tarea orientada a comprobar que lo que se crea se comporta como esperamos, es decir, sigue los requisitos impuestos. Normalmente, la validación de un sistema complejo (p.e. distribuido, con muchos elementos hardware heterogéneos) debe pasar por una prueba piloto. En una prueba piloto el sistema se despliega en un entorno real pero controlado. Debe ser controlado porque si algo va mal, se puede solucionar y no provocan los daños que podría causar de no serlo. Pero aun así, modificaciones en esta fase provoca gastos de tiempo debido a que surgen aspectos que hay que estudiar de nuevo, desarrollarlos y depurarlos en el laboratorio. Hay que destacar que la diferencia entre una prueba en el laboratorio y una piloto puede llegar a ser muy grande. Por ejemplo, es común que un sistema ubicuo se componga de docenas de sensores que se prueban parcialmente en el laboratorio (p.e. normalmente no todos los nodos de la red se prueban al mismo tiempo) y sólo en la prueba piloto el sistema se despliega totalmente. Un ejemplo parecido podría ocurrir con las aplicaciones sociales para móviles.

Precisamente nuestro principal objetivo es minimizar estas diferencias permitiendo desarrollar los sistemas completos en el laboratorio, y así obtener los beneficios de las

[1] http://www.tuaw.com/2010/11/10/iphone-and-android-shares-rise-while-blackberry-symbian-fall/

pruebas piloto como la validación de la aplicación en un entorno natural. Para ello, se hará uso del simulador, donde se reproducirá el entorno físico, y por otro lado, de móviles reales o emuladores. A esta combinación de usar elementos virtuales y reales se le conoce como simulación híbrida.

En el siguiente apartado 2 veremos con más detalle las pruebas de campo y las de laboratorio. Después, en el apartado 3, pasaremos a desarrollar varios objetivos del trabajo y nos centraremos en cumplir una parte de ellos. Luego, realizaremos un estudio del trabajo relacionado con las simulaciones híbridas, apartado 4. En el apartado 5, hacemos una reflexión sobre las aplicaciones para móviles, para después, en el apartado 6, repasar las herramientas y funcionalidades que nos ofrece la plataforma Android para cumplir nuestro objetivo. Teniendo estas características en mente, en el apartado 7, pasaremos a diseñar el sistema que nos permita validar aplicaciones Android para móviles a través del simulador. En el apartado 8, desarrollaremos un pequeño ejemplo para aplicar la plataforma diseñada a la validación de una aplicación para un museo que muestra información de los cuadros a partir de la lectura de códigos QR. El apartado 9, se dedica a exponer un proyecto que acaba de iniciarse con el Museo Arqueológico de Murcia y en el que se aplicará la infraestructura desarrollada en el trabajo. Por último, en el apartado 10, expondremos las conclusiones del trabajo realizado y del pendiente para el futuro.

2. OBJETIVOS

Como hemos visto, la simulación híbrida consiste en probar sistemas o aplicaciones simulando ciertos elementos y usando otros reales. En nuestro caso, lo que se simula es el entorno donde los usuarios utilizarían la aplicación, como por ejemplo un museo, y los teléfonos y la aplicación que corren en estos serían reales. Principalmente, la simulación híbrida se puede emplear para evaluar dos aspectos:

- Validación de la aplicación del móvil por parte de los usuarios. Para ello, uno o varios usuarios acceden al mundo virtual a través de su personaje virtual que lleva asociado también un móvil virtual. Los usuarios pueden desplazarse a través del mundo virtual a través del teclado y el ratón, y al mismo tiempo usar el teléfono móvil real para realizar acciones sensibles al contexto virtual. Por ejemplo, si nos hemos aproximado a un cuadro con una etiqueta QR que contiene los parámetros para obtener información del mismo (p.e. una url apuntando a la Wikipedia), desde el móvil real podemos activar la aplicación para leer códigos de este tipo. Esta aplicación lo que hace es activar la cámara para procesar las imágenes y detectar el código QR para decodificarlo y obtener la información que contenga. El usuario podría realizar esta acción e inmediatamente recibir en el móvil las mismas imágenes del museo virtual en primera persona que el usuario ve en el monitor del ordenador. El usuario tendría que dirigir la mirada de su personaje virtual al modelo 3D del código QR de la simulación (lo que equivaldría a enfocarlo con el móvil real) para que la imagen se transmita al móvil y así éste la procese. Un ejemplo más sencillo podría ser una aplicación que usase la brújula del móvil. Cuando ésta se activase en el móvil real se reflejarían los cambios de orientación que el usuario realiza en el simulador.

- Validación del comportamiento global del sistema. Este tipo de validación busca que el comportamiento global del sistema junto con su entorno y los usuarios sea satisfactorio. Siguiendo con el ejemplo del museo, este tipo de evaluación permitiría analizar cómo se comporta un sistema de visita guiada que tuviera en cuenta las preferencias del usuario y la densidad de visitantes en cada sala. Para esta prueba se necesitan varias ejecuciones variando los parámetros como el número de personas. Realizar estas evaluaciones, aun usando el tipo de simulación híbrida descrita en el objetivo anterior, puede consumir mucho tiempo y dinero. Este coste se debe a que se tiene que usar muchas personas durante mucho tiempo. Por ello, una solución para reducir el coste sería prescindir de estas. Esto implica que también habría que simular el comportamiento del usuario y su interacción con el emulador. Descartamos usar móviles reales, ya que a la hora de gestionarlos dinámicamente y de forma autónoma en varias simulaciones es más complicado. De esta forma, la simulación, al ser totalmente autónoma, podría ejecutarse a un ritmo mucho mayor comparado con el ritmo natural necesario cuando intervienen personas. Destacar que la simulación se sigue considerando híbrida ya que la aplicación es la misma que ejecuta en el móvil real.

Ambos objetivos son muy interesantes y están aún por explotar, sobre todo la validación de sistemas con un conjunto de teléfonos móviles o emuladores. Un objetivo ambicioso consistiría en tener las dos opciones en el grado que se quisiese. Es decir, básicamente consistiría en decidir qué personajes virtuales son controlados por humanos que validan la aplicación y cuáles no. En los juegos en red, por ejemplo el Counter Strike[2], esta configuración es bien conocida. El usuario que crea la partida decide el número de usuarios que se pueden conectar y el número de participantes manejados por el ordenador, conocidos como bots.

Llegar a alcanzar estos objetivos conlleva mucho trabajo, tanto en el simulador, para permitir la interacción del móvil virtual y el real, como en el móvil real para poder interactuar con el simulador. Evidentemente, para esta tesis de máster nos hemos limitado a realizar el estado del arte de las simulaciones híbridas con móviles y a diseñar e implementar parte de la infraestructura relativa a los dispositivos móviles con sistemas operativo Android. Limitándonos a esta parte del trabajo, nuestro objetivo principal es que:

> El programador que está acostumbrado a desarrollar aplicaciones Android no tenga que modificar su forma de trabajar, ni usar APIs con clases nuevas, cuando quiera validar la aplicación.

En definitiva, lo ideal es que el código fuente de la aplicación no se tenga que modificar cuando ésta se quiera validar.

[2] http://www.counter-strike.net

3. ESTADO DEL ARTE

El método más razonable de evaluación de sistemas o aplicaciones es el que se realiza en el entorno real con los usuarios, lo que se conoce como prueba de campo. Se han realizado y descrito varias pruebas de campo de aplicaciones para móviles [7], [20], [21]. Por ejemplo Häkkila y su equipo [7] realizaron pruebas con usuarios en el centro de la ciudad de Oulu. Durante la prueba se usaron varias aplicaciones, como calendarios de eventos y servicios de asistencia. Durante la prueba se realizaron entrevistas diarias a los usuarios que aportaron datos que valoraban el comportamiento de las aplicaciones.

Es evidente de que las pruebas de campo proporcionan datos muy realistas y valiosos porque se realizan en el mundo real con restricciones contextuales realistas. Sin embargo, también tienen algunos problemas:

- Podría llevar a entornos contextuales no controlados provocando resultados inútiles.
- Requieren mucho tiempo para coordinarlas y dirigirlas.
- Son costosas económicamente.

Por ello, la idea es usar las pruebas de laboratorio simulando partes del entorno real en un entorno más controlado, el laboratorio. Una revisión [11] mostró que el 71% de las evaluaciones de aplicaciones de teléfonos móviles por usuarios se realizaron a través de pruebas de laboratorio.

Varios estudios comparan ambos tipos de pruebas. Kjeldskov y Stage [12] [13] buscaron técnicas apropiadas que permitieran evaluaciones de aplicaciones para móviles in-situ (de campo) y en el laboratorio. Encontraron varias diferencias entre ambas pruebas, tales como la comodidad de uso a nivel social (la interacción con el mundo real es más fácil que en el laboratorio) pero los problemas más básicos de usabilidad se detectaban en ambas pruebas por igual. Kaikkonen y su equipo [10] revelaron que las pruebas de laboratorio son suficientes en la mayoría de los casos ya que, con frecuencia, las pruebas de campo no aportan ningún valor añadido.

En términos generales, los resultados mostraron que las pruebas de campo no se pueden sustituir completamente por las pruebas de laboratorio y estas pruebas deberían realizarse al menos al final del proceso de desarrollo para estudiar el comportamiento específico del usuario con diferentes configuraciones. Las pruebas de laboratorio, sin embargo, se pueden usar iterativamente en el proceso de desarrollo si se utilizan métodos y técnicas apropiadas de evaluación. Pero llegados a este punto surgen problemas. Duh y su equipo [4] [19] y también Kjeldskov [12] [13] han identificado que faltan técnicas y métodos apropiados de evaluación en el laboratorio. En particular, cuando se trata de investigar el comportamiento de los usuarios en etapas tempranas del proceso de desarrollo [4].

Por otra parte, las simulaciones a través de mundos virtuales pueden mejorar el proceso de desarrollo de sistemas [14] ya que estos pueden promover ideas para nuevas funcionalidades y permitir la realización de primeras validaciones de las aplicaciones por parte de los usuarios. En el contexto de la computación ubicua, hay una tendencia a usar simulaciones virtuales para investigar los entornos ubicuos en sí y su funcionamiento [22, 23]. El uso de simulaciones virtuales para el estudio de las interacciones ubicuas entre dispositivos es un campo nada explotado. Los proyectos que estudian este tema [9, 17, 23, 1], incluyen los dispositivos en el mundo virtual. La interacción física con los dispositivos móviles no está disponible. En su lugar, son virtuales y se tienen que realizar con el ratón y el teclado siendo una forma de interacción muy alejada de la real. Por ejemplo, Manninen [17] utilizó representaciones virtuales de la interacción de los dispositivos en su entorno. Su objetivo principal era desarrollar y probar fácilmente diferentes mundos virtuales y sus dispositivos de entrada. Barton [1] se interesó también por lo mismo y desarrolló UbiWise, que es una herramienta de simulación para aplicaciones de computación ubicua. Esta herramienta ayuda a investigar aplicaciones que usan cámara como medio de interacción, por ejemplo los teléfonos móviles. Como los dispositivos sólo se representaban en el mundo virtual 3D, el usuario tiene que interactuar a través de los métodos tradicionales de entrada, lo que supone, como ya hemos dicho, una degeneración del uso real del dispositivo.

Para solucionar el problema de la falta de realismo en la interacción con el dispositivo, hemos buscado otras alternativas a las simulaciones que involucren al menos parte del mundo real. Morla y Davies fueron pioneros en este tipo de trabajo [26]. Utilizaron las simulaciones para probar aplicaciones basadas en localización para controlar la salud de personas. Usando su entorno simulado, pueden evaluar virtualmente el funcionamiento de los sensores adjuntos a un sistema de control médico. Aunque su principal objetivo no era investigar las interacciones de los usuarios con el entorno ubicuo sino el funcionamiento de este entorno, el trabajo de Morla y Davies da el primer paso de lo que se conoce como simulación híbrida. La simulación híbrida significa integrar y combinar el mundo real y virtual. El sistema de control médico no se simulaba y realmente reaccionaba al contexto generado artificialmente.

A la simulación híbrida también se le denomina "realidad dual" debido a Lifton y sus colegas [16]. Éstos utilizaron Second Life[3] como herramienta de visualización virtual de los flujos que generaban los sensores del mundo real (p.e. sensores de temperatura). [15] no utilizan el mundo virtual como plataforma de visualización del funcionamiento de los dispositivos reales como Davis y Lifton. En su caso, lo utilizan como plataforma de evaluación para los usuarios usando teléfonos móviles reales como dispositivo de interacción y una simulación virtual del entorno ubicuo. Haesen también realiza algo parecido [6]. Emplean una simulación virtual de un museo para realizar una prueba de usuario donde los móviles son reales y se usan para interactuar con el museo virtual.

[3] http://secondlife.com

Haesen considera el concepto de simulación híbrida como una nueva técnica prometedora de validación de aplicaciones en etapas tempranas en el proceso de desarrollo centradas al usuario. Los autores de [15] defienden su propuesta frente a [6] debido a que:

- Usan una plataforma de simulación virtual para el entorno ubicuo bien conocida como es Second Life.
- Usan una herramienta (MoPeDT) para generar prototipos, fácil y rápidamente.

Por nuestra parte, nosotros defendemos nuestra propuesta frente a [15] ya que:

- El uso de UbikSim proporciona facilidades para ampliar el entorno y, además, su programación es en Java, al igual que en Android. Por tanto, no es necesario aprender el lenguaje script de Second Life.
- El proceso para desarrollar la aplicación del móvil es exactamente el mismo que se sigue para cualquier aplicación Android. Lo que implica que se puede crear cualquier programa que se desee con el aspecto que uno quiera. Sólo hay que cambiar el sdk a usar y la aplicación pasa de la versión de prueba a la de producción en segundos y sin errores. Por el contrario, MoPeDT ofrece la generación de prototipos rápidos pero cerrados.
- La comunicación se realiza directamente entre el simulador y el móvil, sin tener que utilizar un servidor que gestione el contexto. En MoPeDT, la gran parte de la comunicación entre el simulador y el móvil real se tienen que realizar a través de un servidor.
- La interacción del móvil virtual con el entorno no la hemos limitado. Sin embargo, MoPeDT define 3 tipos de interacción que básicamente consisten en formas de seleccionar objetos del entorno que tienen asociados servicios.

4. TAXONOMÍA DE LAS APLICACIONES PARA MÓVILES

Las aplicaciones Android podrían agruparse dependiendo del uso que hagan de los recursos en el teléfono, por ejemplo, según [18] hay cuatro categorías: foreground, background, intermitentes y widget. Las aplicaciones también se puede clasificar dependiendo de la arquitectura que usan en términos de dónde se encuentra la lógica de negocio de los servicios que se invocan. Así, se pueden diferenciar las que siguen una arquitectura centralizadas (p.e. todo se realiza en el teléfono móvil), o las que siguen una distribuida como las cliente-servidor o las P2P. Pero también se pueden clasificar en términos de la dimensión social que éstas tengan, en sociales y no sociales. A partir de esta información surge de forma natural la siguiente pregunta: ¿qué es una aplicación social? El siguiente párrafo obtenido de interconnected.org puede ayudarnos:

> Social software's purpose is dealing with groups, or interactions between people. This is as opposed to conventional software like Microsoft Word, which although it may have collaborative features ("track changes") isn't primarily social.

Entonces está claro que el software social se refiere a lo que puedes hacer con él y no cómo se desarrolla. Por ejemplo, una herramienta que ofrece un entorno de trabajo colaborativo es una aplicación social. También Twitter y Skype son aplicaciones sociales. ¿Nos interesa centrarnos en este tipo de aplicaciones? Realmente no. La combinación de software social y aplicaciones móviles lleva a una nueva categoría, software social para móviles. Este tipo de programas normalmente manejan una información de contexto, principalmente la localización y otra información relacionada con redes sociales, como elemento clave en su funcionamiento. Así que también se denominan programas sociales basados en la localización o servicios. Por tanto, se pueden crear nuevos servicios basados en la localización de cada miembro del grupo. Un ejemplo simple e ilustrativo de programa social para móviles es Instagram[4]. Este permite compartir fotos con sus colegas en una red social (p.e. a través de twitter o Facebook). Se basa en la localización porque se tiene en cuenta el lugar donde se hizo la foto. La comunicación entre usuarios es asíncrona en el sentido de que cuando el usuario publica su foto, el resto de usuarios no tienen porqué ver la foto inmediatamente. Otro aspecto importante es que, para que la interacción realmente se lleve a cabo, no es importante dónde se encuentren los usuarios en el momento de ver la foto. La posición sólo es importante en el momento que el usuario hace la foto. Se puede organizar en secciones o como se desee en función del tipo de trabajo de investigación. Da igual, incluso, dónde estaba el usuario cuando la publicó. Por ello, la posición no es importante para la interacción. Otro programa muy conocido es WhatsApp. Este permite comunicación síncrona y asíncrona donde la posición de los usuarios no es importante.

[4] http://instagr.am/about/

Si prestamos atención a las distintas características detectadas, se puede crear una clasificación de las diferentes aplicaciones en las que podríamos estar interesados para nuestro trabajo.

- Social: una aplicación social implica que esta tenga sentido cuando un grupo de usuarios la usan para un propósito común (p.e. trabajo, diversión).

- Que dependen de la localización: significa que gran parte de la funcionalidad de la aplicación depende de conocer la posición de los elementos de interés. Algunas aplicaciones podrían depender de la localización pero no necesitar servicio de localización (p.e. en Instagram los usuarios podrían indicar dónde hicieron la foto de forma manual). Otras aplicaciones, por ejemplo Foursquare, necesitan del servicio para que funcionen de forma correcta.

- Síncronas/asíncronas: una aplicación es síncrona cuando su funcionamiento depende de que el resto del grupo esté conectado al mismo tiempo (p.e. las batallas en foursquares). Por otro lado, las aplicaciones asíncronas funcionan correctamente independientemente de que el resto del grupo esté conectado o no al mismo tiempo (p.e. Instagram). Señalar que algunas aplicaciones podrían usar ambos modos dependiendo de la situación.

- Las aplicaciones que no usan ningún tipo de comunicación (bluetooth, NFC, lectura de códigos de barras)

- Número de usuarios: el número de usuarios es importante a la hora de probar la aplicación en el móvil. Principalmente, en términos de escalabilidad y usabilidad de la aplicación.

- Basadas en los sensores del móvil. Pertenecen a esta categoría las aplicaciones en las que es importante distinguir entre los tipos de sensores usados y cómo se gestionan.

INTERACCIÓN DEL TELÉFONO INTELIGENTE CON EL EXTERIOR

Las aplicaciones de móviles tienen normalmente un gran número de interacciones externas. Ballagas y su equipo [1] ya dieron un repaso a los diferentes métodos de entrada disponibles en los móviles actuales (p.e. la cámara o acelerómetro) que se pueden usar para llevar a cabo técnicas de interacción. A nosotros también nos interesa identificar cuáles son los tipos de interacción existentes para trabajar con el móvil y el simulador. En la Figura 1 se representan las interacciones y los elementos implicados. Se han coloreado los elementos con los que se interactúa dependiendo de su dificultad a la hora de simularlos. Analizamos cada elemento para ver si es interesante simularlo o no:

- Percibir el entorno físico a través de sensores, mecanismos de localización, la cámara y el micrófono. Estas interacciones se realizan en un sólo sentido y resulta muy interesante su simulación. La aplicación a validar debe percibir el entorno virtual como si fuese real, es decir, que la cámara tome imágenes del simulador,

que la posición obtenida sea relativa al mundo virtual, etc. Simular estos elementos no es difícil. No es necesario simular los satélites ni las torres de telefonía móvil. Es suficiente con un modelo que nos dé la posición con un error que dependa del proveedor utilizado.

- NFC podría ser interesante para determinadas aplicaciones pero de momento no es una tecnología muy extendida, muy pocos teléfonos móviles están equipados con tecnología NFC. Además, no se puede simular de manera trivial, como veremos en la sección 6.

- Bluetooth resulta difícil simularlo. Pero se podría simplificar dependiendo del caso. Por ejemplo, si nuestra aplicación lo usa simplemente como mecanismo para descubrir otros dispositivos. Este procedimiento de búsqueda consiste en el envío de un mensaje que llega a todo otro dispositivo bluetooth activo que se encuentre a su alcance (10 o 100 metros) y éste le responde con cierta información sobre el mismo.

- Simular la interacción con la interfaz gráfica nos interesa para el caso de que queramos validar sistemas, es decir, la interacción con el emulador debe ser autónoma ya que no hay un usuario para interactuar directamente con éste. La tarea, al igual que la anterior, tampoco es sencilla. En el lado del simulador se necesita un diseño a medida para cada actividad con las acciones disponibles (p.e. pulsar un botón, escribir algo, etc). Y luego, en el lado del teléfono o emulador, se tendría que poder ofrecer los servicios con llamadas al sistema que simulen la interacción (tocar un punto determinado de la pantalla, pulsar un botón). Veremos posteriormente qué ofrece Android relacionado con la prueba de interfaces gráficas.

- El uso del sistema telefónico e internet no tiene ningún sentido que se simule. Accedemos a los servicios reales de la telefonía e internet a través del teléfono o el emulador.

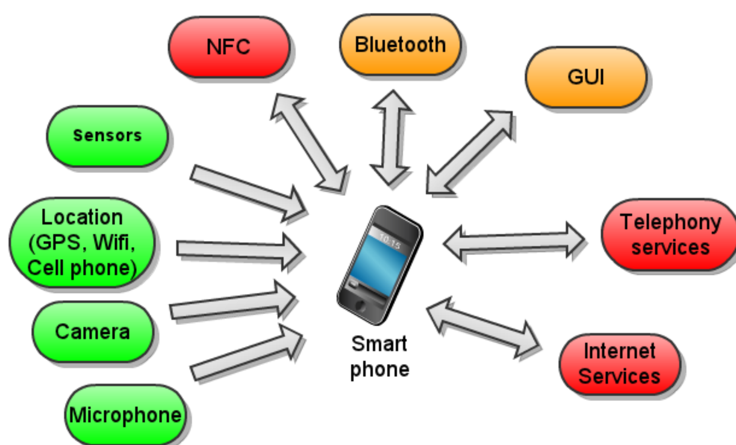


Figura 1 Interacciones del teléfono móvil con el exterior.

5. La plataforma Android

En esta sección se recogen distintas características específicas de Android relativas a las clases que ofrece su SDK para desarrollar, así como las herramientas de depuración. Esta sección se basa en el libro [18], para más detalles sobre la plataforma Android acudir a la referencia ya que nos centramos en los elementos que nos interesan: localización, sensores, audio y video, mecanismos de comunicación entre aplicaciones y herramientas para la prueba de aplicaciones.

Servicios basados en la posición

Los servicios de localización en los teléfonos inteligentes se basan en las redes (WiFi y antenas de teléfono) y/o el GPS. Los servicios de localización en Android pertenecen a la capa de servicios. Se obtienen a través del contexto indicando el nombre del servicio; no se pueden obtener directamente. La clase `android.app.Activity` proporciona un método `getSystemService()` que se usa para obtener los servicios. En este caso, los servicios de localización se obtienen pasándole `Context.LOCATION_SERVICE` como parámetro. Al hacerlo, nos devuelve una instancia de la clase `LocationManager`. Con una instancia de esta clase, y dependiendo del proveedor de localización usado, obtenemos la posición. Hay 3 tipos de proveedores:

- El proveedor GPS usa el sistema de posicionamiento global para obtener la información sobre la posición.
- El proveedor de redes usa las torres de telefonía o las redes WiFi para btener la información sobre la posición.
- El proveedor pasivo es como un interceptor de actualizaciones de posición cuando otras aplicaciones las solicitan. Si ninguna aplicación solicita información sobre la posición, no se obtendrá ninguna.

El primer método es el preferido por su precisión pero gasta mucha energía y no funciona en entornos cerrados. El segundo tiene bastante menos precisión pero funciona en entornos cerrados y gasta menos batería.

La clase `LocationManager` maneja perfectamente las actualizaciones debidas a los cambios de posición y automáticamente manda alertas cuando se sobrepasan unos límites. Internamente, el código registra a los interesados (independientemente del proveedor) y les informa cada segundo y cuando el desplazamiento es al menos de un metro. No se puede cambiar este comportamiento. Por eso, si se deja funcionando durante mucho tiempo, se puede agotar la batería rápidamente.

A través de Geocoding, las coordenadas latitud y longitud se pueden pasar a direcciones y viceversa.

Una utilidad interesante de Android permite al programador simular en el emulador diferentes localizaciones por las que el usuario pasaría para depurar las aplicaciones. De

esta forma, el usuario no se mueve pero el emulador lo hace virtualmente. Ésto nos permite depurar de forma natural servicios basados en localización. Pero estos servicios deberían ser consumidores de servicios basados en localización. Ésto quiere decir que la información sobre la posición que es importante para el servicio es aquella que corresponde al momento en que el usuario utiliza el servicio. En los últimos dos años, han surgido varias aplicaciones muy sencillas basadas en localización. Un ejemplo es *foursquares*. En esta aplicación, los usuarios utilizan la posición para que el sistema le sugiera lugares que ofrezcan descuentos o alguna ventaja. Estos lugares deben estar registrados en el sistema. La parte interesante es que el marco de trabajo ofrece una API que permite a los desarrolladores crear nuevas aplicaciones, por ejemplo, aplicaciones sociales para móviles que usen de forma efectiva la información sobre la posición (sitios de interés como tiendas).

SENSORES EN ANDROID

En Android, un sensor es un componente hardware que se integra en el dispositivo y que alimenta a las aplicaciones de datos relativos a magnitudes físicas del mundo. Los sensores funcionan en una sola dirección; son de solo lectura (con una excepción, el sensor NFC). Los sensores se gestionan de forma parecida a la localización. Éstos también se gestionan a través de una clase (`SensorManager`) que se encarga de todos los sensores del teléfono. Algunos de los tipos de sensores que pueden disponer los dispositivos Android son:

- Sensor de luz
- Sensor de proximidad
- Sensor de temperatura
- Sensor de presión
- Giroscopio
- Acelerómetro
- Sensor de campo magnético
- Sensor de orientación
- Sensor de gravedad (a partir de Android 2.3)
- Sensor de aceleración lineal (a partir de Android 2.3)
- Sensor de vector de rotación (a partir de Android 2.3)
- Sensor de Comunicación de campo cercano (NFC) (a partir de Android 2.3)

Sería muy extraño que un dispositivo Android tuviese todos estos sensores. La mayoría sólo dispone de algunos de ellos.

La clase `Sensor` es una intermediaria que conecta al resto del código con el sensor físico. Es fácil usar un sensor registrándose a éste a través de un *listener* que nos proporciona los valores. En el momento de registrarse, se indica el sensor y la frecuencia de actualización deseada. Obviamente, cada sensor gestiona sus propias magnitudes y cada una tiene su interpretación. Android permite gestionar todos los sensores de la misma forma.

Además de estas clases para trabajar con sensores, hay herramientas no oficiales que nos ayudan a trabajar con los sensores. Por ejemplo, el *Sensor Simulator*, es una aplicación de *OpenIntents*[5] que permite simular el movimiento del móvil y sus correspondientes sensores (p.e. acelerómetro, compás, orientación) con sólo un movimiento del ratón. Obviamente, este simulador se conecta al emulador a través de una aplicación instalada en el PC. Samsung también ofrece un simulador que te permite registrar sensores obtenidos por el móvil simulado. Éste también te ofrece conectarte al móvil real para obtener registros reales del mismo.

SENSOR NFC

Android 2.3.3 soporta la tecnología NFC permitiendo al dispositivo actuar como una etiqueta para un lector, o como un lector para detectar y leer etiquetas NFC. La tecnología NFC se parecen a la RFID, pero en el primer tipo la cobertura es menor, de unos pocos centímetros. La tecnología NFC ofrece tres modos de funcionamiento:

- Lectura y escritura de etiquetas sin llegar al contacto.
- Emulador de tarjeta. Permite al dispositivo Android actuar como una etiqueta de forma que podría reemplazar a una tarjeta de crédito.
- Comunicación usando un protocolo P2P definido por Google.

El segundo modo no está todavía disponible en el SDK porque es muy difícil de implementar, en parte debido a las múltiples formas de fabricar los componentes NFC. Pero es posible hacerlo a nivel de la capa de controladores usando el kit de desarrollo nativo para Android (NDK).

La forma en que Android maneja NFC no es como los otros tipos de sensores. En vez de usar la clase `SensorManager`, se usa la clase `NfcAdapter`. Si el adaptador está activado, y se detecta una etiqueta NFC, se produce un complejo proceso para determinar qué actividad, si la hay, debería recibir un *intent* informando sobre la etiqueta detectada. La lógica para crear dichos *intents* usa elementos que no son públicos en el SDK de Android. Esto significa que no es nada sencillo crear una actividad que envíe falsos *intents* sobre la lectura de una etiqueta NFC. Por tanto, si se quiere probar una aplicación que use esta tecnología, necesitamos usar un móvil real.

EL MARCO DE TRABAJO DEL AUDIO Y EL VÍDEO

[5] http://code.google.com/p/openintents/wiki/SensorSimulator

Android ofrece un marco de trabajo muy interesante para reproducir y grabar audio y vídeo desde varias fuentes. Para nuestro trabajo, nos interesa las acciones que estén relacionadas con la percepción del entorno, es decir, percibir sonidos e imágenes.

GRABAR AUDIO

En principio, no nos resulta de mucho interés recibir el audio en nuestras aplicaciones, pero de todas formas veremos cómo lo hace Android. Una forma de obtener audio es a través de la clase `android.media.MediaRecroder`. Se debe crear una instancia de la clase

```
MediaRecorder recorder = new MediaRecorder();
```

y establecer la fuente de audio

```
recorder.setAudioSource(MediaRecorder.AudioSource.MIC);
```

Hasta la versión 1.6 de Android sólo soporta el micrófono como fuente. En esta versión se añadieron fuentes relacionadas con las llamadas:

- `MediaRecorder.AudioSource.VOICE_CALL` para grabar la llamada entera.
- `MediaRecorder.AudioSource.VOICE_UPLINK` para grabar la voz del usuario.
- `MediaRecorder.AudioSource.VOICE_DOWNLINK` para grabar la voz del usuario al otro lado.

Con la versión 2.1 del SDK de Android se añadieron dos fuentes más:

- `MediaRecorder.AudioSource.CAMCODER` Usa el micrófono asociado a la cámara o si no el que tenga por defecto.
- `MediaRecorder.AudioSource.VOICE_RECOGNITION` Usa el micrófono asignado para el reconocimiento de voz y si no el que tenga por defecto. El audio recogido por el micrófono se envía a la aplicación en el formato más crudo posible.

Esta clase se usa para grabar vídeo, pero si lo que se quiere es procesar el audio percibido, la clase que hay que utilizar es `AudioRecord`. Con esta clase, el audio se graba en sus buffers y nuestra aplicación puede hacer lo que quiera con él. En esta clase también hay que indicar la fuente de audio y son las mismas que con la clase anterior.

LA CÁMARA PARA RECIBIR IMÁGENES

Poder acceder al flujo de vídeo de la cámara permite incorporar vídeo en directo en nuestras aplicaciones.

Algunas de las aplicaciones que más llaman la atención usan esta funcionalidad para implementar realidad aumentada (proceso de superponer dinámicamente información contextual sobre las imágenes obtenidas de la cámara).

Para acceder a los servicios de la cámara, hay que usar la clase `android.hardware.Camera`. Esta proporciona unos métodos estáticos para obtener y liberar la cámara, `open()` y `release()` respectivamente.

Para previsualizar las imágenes de la cámara se tiene que usar la clase `SurfaceHolder`. Pero para que se vea el flujo de imágenes en la aplicación, hay que usar un *Surface View* en la interfaz gráfica.

Para empezar y dejar de recibir imágenes, la clase `Camera` tiene los métodos `startPreview()` y `stopPreview()` respectivamente.

MECANISMOS DE COMUNICACIÓN ENTRE APLICACIONES

Los intents son los elementos usados por Android para invocar a componentes. Un intent es una acción con datos asociados. La lista de componentes incluye actividades (componentes con interfaz gráfica), servicios (código en segundo plano), recibidores de broadcast (código que responde a mensajes de broadcast), y proveedores de contenido (código que abstrae datos).

El siguiente código es un ejemplo de un intent que llama al navegador para que muestre una dirección de internet determinada:

```
public static void invokeWebBrowser(Activity activity)
{
    Intent intent = new Intent(Intent.ACTION_VIEW);
    intent.setData(Uri.parse("http://www.google.com"));
    activity.startActivity(intent);
}
```

Pero nosotros estamos interesados en la comunicación con servicios. Android soporta dos tipos de servicios: locales y remotos. Los servicios locales son accesibles sólo por la aplicación que los aloja, sin embargo, a los servicios remotos se puede acceder desde varias aplicaciones. Los servicios remotos publican su interfaz a los clientes usando un lenguaje de definición de interfaces de Android (AIDL). A nosotros nos interesa este segundo tipo de servicio para que la aplicación a validar se comunique con los servicios OSGi.

HERRAMIENTAS PARA LA PRUEBA DE APLICACIONES EN ANDROID

El marco de trabajo para la prueba de aplicaciones en Android[6] proporciona una arquitectura y unas herramientas potentes que ayudan a probar muchos aspectos de una aplicación. Revisamos esas herramientas en busca de que alguna nos sirva para cumplir nuestros objetivos.

El marco de trabajo para las pruebas tiene las siguientes características principales:

[6] http://developer.android.com/guide/topics/testing/testing_android.html

- Se basa en *JUnit* para realizar las pruebas. *Plain JUnit* no se puede utilizar para probar una clase que llame a la API de Android, pero con las extensiones de *JUnit* que ofrece Android sí.

- Las extensiones de *JUnit* de Android proporcionan componentes específicos para realizar pruebas de casos a las clases. Esas clases proporcionan métodos que facilitan la creación de objetos y métodos "falsos" que a su vez ayudan a controlar el ciclo de vida de los componentes.

- Las herramientas del SDK para construir y probar aplicaciones están disponibles en Eclipse a través de ADT, y también usando la línea de comandos. Esas herramientas obtienen información del proyecto de la aplicación que se quiere probar y la usan para generar de forma automática toda la estructura para las pruebas.

- El SDK también ofrece *monkeyrunner*, una API para probar dispositivos a través de programas en *Python*, y *UI/Application Exerciser Monkey*, una herramienta de línea de comandos para realizar pruebas de estrés en los dispositivos.

No estamos interesados en realizar pruebas de casos ni unitarias. Esta tarea, que consiste en encontrar errores de programación, debería ser previa

```
monkeyrunner -plugin <plugin_jar><program_filename><program_options>
```

al proceso de validación de la aplicación, aunque estos errores se detectar ían tarde o temprano usando el simulador. Por ello, descartarmos JUnit y las herramientas SDK que ofrece Android para eclipse. Quizás, resulta más interesante para la simulación híbrida, en el futuro, la segunda herramienta que se encarga de interactuar con la UI.

La herramienta *monkeyrunner* no tiene nada que ver con *UI/Application Exerciser Monkey*, más conocida como herramienta *monkey*. Esta última se ejecuta en el terminal del teléfono o el emulador con `adb shell` y genera flujos pseudo-aleatorios de eventos del sistema. Esta herramienta no nos interesa porque nosotros querríamos que se generasen eventos específicos que correspondiesen a los generados por el personaje virtual en el simulador. Por el contrario, la herramienta *monkeyrunner* sí nos podría servir, ya que controla los dispositivos y los emuladores desde el ordenador enviando comandos y eventos específicos desde una API. La sintaxis del comando *monkeyrunner* es:

Esta herramienta podría resolver el problema con el inconveniente de que se lanza desde un comando y utiliza *Jython*, una implementación de *Python* que usa el lenguaje de programación Java.

Por otro lado, el SDK de Android no ofrece soporte para lanzar eventos del sistema relativos a la UI desde el código fuente.

6. ARQUITECTURA DEL SISTEMA

El objetivo de la tesis es desarrollar una plataforma que nos permita validar aplicaciones de teléfonos móviles Android a través de la simulación híbrida. La arquitectura se ha diseñado con el objetivo de conseguir que sea modular y sencilla de utilizar. Debemos tener en cuenta que todos los elementos de la arquitectura no se utilizarán cuando el móvil se compile para su comercialización. Debido a estos requisitos, gran parte de nuestra plataforma se basa en una ampliación de OSGi (Open Services Gateway Initiative) llamada Remote-OSGi. El objetivo de OSGi es definir las especificaciones abiertas de software que permita diseñar plataformas compatibles que puedan proporcionar múltiples servicios. La mayoría de las implementaciones están hechas en Java y se componen de módulos (.jar) que ofrecen y consumen servicios. Además, R-OSGi permite que los módulos se puedan comunicar entre distintas máquinas virtuales. Lo que resulta muy útil e interesante para comunicar el móvil con el PC. Por tanto, la plataforma se compone principalmente de 3 elementos:

1. Marco de trabajo (framework) OSGi en el PC sobre el que se encuentra el módulo del simulador que virtualiza el entorno donde la aplicación se quiere validar y ofrece servicios para que el móvil se pueda comunicar con éste.

2. Plataforma que permita al móvil interactuar con el simulador de forma que la aplicación a validar perciba el mundo virtual como si fuese el real. Esta plataforma se compone de dos elementos, framework OSGi para poder comunicarse con el simulador y capa software que se compila junto con la aplicación a validar para poder sustituir a las clases de Android usadas para percibir el entorno, por ejemplo, `Camera`, `SensorManager`, etc.

3. Mecanismo de comunicación entre el simulador y el móvil real que básicamente se compone del módulo R-OSGi que veremos en el siguiente apartado.

El procedimiento para utilizar estos elementos consistiría en crear el entorno y arrancar OSGi en el PC con todos los módulos necesarios, incluido el simulador. En móvil habría también que tener una aplicación OSGi que tuviese instalados todos los módulos necesarios para comunicarse con el servidor. Al igual que con la versión de PC de OSGi en la versión móvil sólo hay que instalar los módulos una vez, las próximas veces con iniciar OSGi también se iniciarán los módulos. Y por último, compilar la aplicación a validar con el paquete adecuado (middleware, API ó SDK) para que al arrancarla en el teléfono que tiene instalado OSGi, la aplicación perciba el entorno virtual como si fuese el real.

En la Figura 2 se muestra la arquitectura global del sistema con los elementos descritos anteriormente y algunos más. La parte izquierda representa el ordenador con OSGi donde está corriendo el simulador. En éste hay 4 módulos principales. El módulo *UbikSim* que corresponde al simulador e implementa la interfaz ofrecida por el módulo *UbikSimServices* y usa el módulo *AndrodServices* para acceder a los servicios del móvil Android a través del

módulo *R-OSGi*. La parte derecha representa al teléfono móvil con sistema operativo Android o al emulador. Como puede comprobarse, ésta es la parte más compleja y se compone de dos entornos. Por un lado, el entorno de trabajo de OSGi para móviles que permite comunicar y gestionar el estado del móvil con el simulador. Y por otro lado, la aplicación a validar compilada con una capa intermedia para la simulación que se comunica con el marco de trabajo OSGi a través de *intents*.

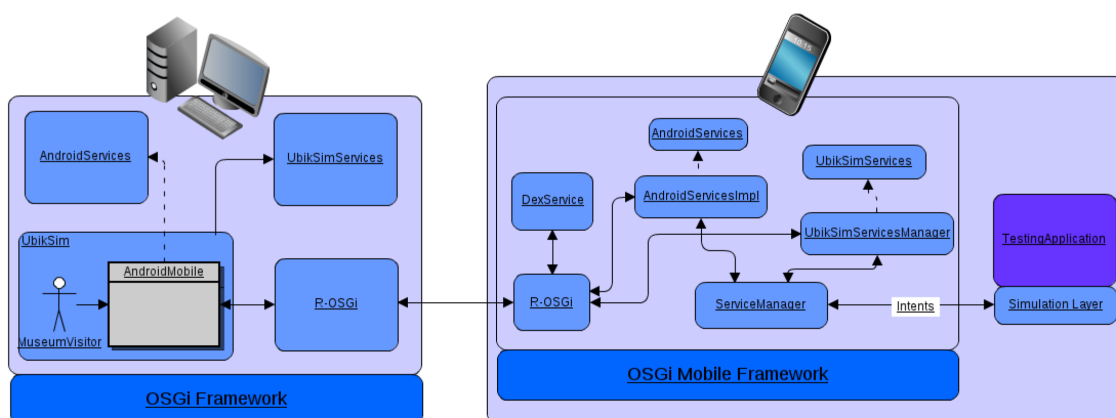


Figura 2 Arquitectura global del sistema.

Para tener una idea de cómo se comunicarían estos módulos, mostramos el diagrama de la Figura 3 que muestra un ejemplo del intercambio de mensajes realizados cuando la aplicación del móvil real quiere recibir cambios de temperatura del entorno. Toda esta secuencia de mensajes sirve para que el móvil reciba la temperatura del simulador como si fuese la real. Lo primero que hace la aplicación a validar es registrarse para recibir cambios de temperatura. La capa de simulación registra esta petición y la envía a través de un *intent* al módulo *ServiceManager*. Éste simplemente captura el *intent* y reenvía la petición al módulo *UbikSimServicesManager* que a su vez utiliza el módulo *R-OSGi* para llamar de forma remota a los servicios de *UbikSim*. Éste servicio informa al móvil virtual (`AndroidMobile`) de que su equivalente real desea recibir notificaciones sobre cambios en la temperatura. Entonces, cuando la persona virtual se mueva a otro sitio con temperatura diferente, el móvil virtual lo detectará y se lo comunicará al móvil real utilizando el servicio remoto *AndroidServices*. Estos servicios implementados en el módulo *AndroidServicesImpl* del móvil, los retransmiten al *ServiceManager* para que éste informe a través de los intents a la capa de simulación. Esta capa, al recibir la notificación de un cambio de temperatura, genera un evento y a través del *listener* registrado por la aplicación informa a esta con el método `onSensorChanged()` pasándole como parámetro el evento. Destacar, que las interfaces y la forma en la que la aplicación se registra y es notificada para obtener la temperatura es la misma que se haría con el SDK de Android oficial.

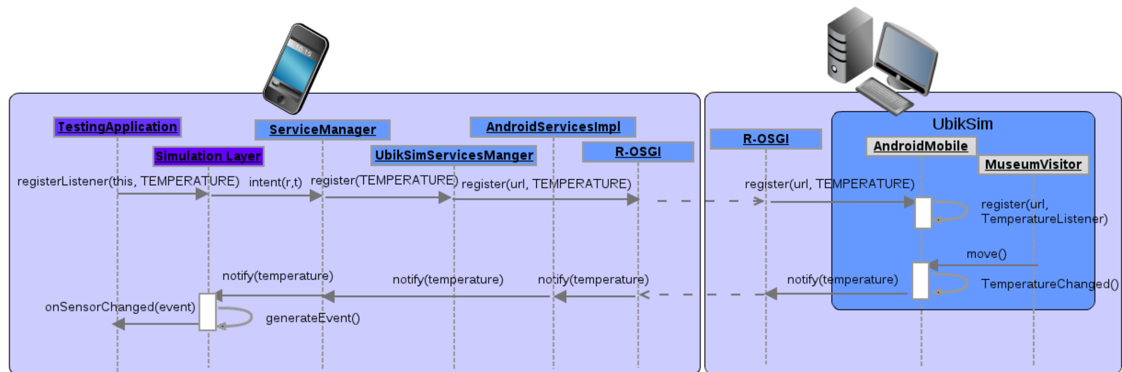


Figura 3 Diagrama de secuencia del sistema global.

Veamos más en detalle cada uno de los elementos que componen la plataforma. Empezaremos por el mecanismo de comunicación, luego veremos la parte de los módulos que corresponden al simulador y, por último el móvil.

MECANISMO DE COMUNICACIÓN: R-OSGI

Necesitamos una arquitectura flexible y modular que nos permita manejar frecuentes cambios de versión o funcionalidades de forma incremental y dinámica. Todo esto lo ofrece la plataforma OSGi y, de hecho, se usa en muchos sistemas ubicuos como hacen Gu y sus colegas [5]. Éstos propusieron una infraestructura basada en OSGi para manejar servicios sensibles al contexto de forma fiable y segura y que soportara de forma eficiente el descubrimiento, adquisición y razonamiento de contexto.

Las ventajas que ofrece OSGi son:

- un núcleo de ejecución ligero, válido para sistemas con pocos recursos,
- dinámicamente extensible para adaptarse a requisitos diferentes,
- funciona como "Sistema Operativo"para la plataforma Java,
- gestión del ciclo de vida de los componentes de una aplicación,
- control de la gestión y el despliegue de las versiones de paquetes y librerías, y de resolución de dependencias,
- programación orientada a servicios dentro de una VM,
- dinamismo: publicación y suscripción,
- ocultación, modularización y reutilización.

OSGi remoto (R-OSGi) permite que los módulos puedan usar servicios que están en otras máquinas virtuales Java. Ésto significa que los módulos pueden estar en el mismo

ordenador o en ordenadores diferentes. Además, la forma de usar cada servicio remoto desde el punto de vista del código fuente del módulo es el mismo, lo que es una ventaja.

R-OSGi no es una plataforma nueva que se distinga de OSGi. De hecho, las implementaciones sobre la especificación consiste en un módulo que se encarga de gestionar la publicación y uso de servicios remotos.

Por todas estas ventajas y aprovechando el hecho de que UbikSim tiene una versión para OSGi, hemos decidido usar R-OSGi para que los móviles y el simulador se comuniquen a base de servicios.

EL SIMULADOR: UBIKSIM

UbikSim es un simulador de entornos de computación ubicua que trata de ayudar a probar las aplicaciones y servicios cuyo comportamiento depende del entorno y los usuarios [2]. La metodología que se emplea con UbikSim consiste en crear el entorno, implementar los comportamientos, dejar que la simulación corra de forma autónoma durante el tiempo que se considere conveniente y, por último, analizar los resultados. UbikSim está programado completamente en Java y se basa en dos productos de código libre:

- MASON[7], una librería de simulación multi-agente basada en eventos discretos. Cada entidad que desee obtener un turno en la simulación para realizar las acciones que considere debe implementar la interfaz `Stepable`.
- SweetHome 3D[8], una aplicación para el diseño de interiores de forma sencilla. Esta herramienta se usa tanto para el modelado de los entornos como para el renderizado en la simulación.

Esta forma de funcionar con UbikSim es la que nos sirve para validar el entorno y el sistema donde se desenvuelven conjuntamente aplicación móvil y usuario. UbikSim dispone de comportamientos basados en autómatas que pueden utilizarse para que el personaje virtual se desenvuelva autónomamente por el entorno. Pero no se puede utilizar UbikSim como simulador para que los usuarios reales validen aplicaciones móviles. Ésto se debe principalmente a que es el usuario el que tiene que controlar su personaje virtual. Por tanto, los principales cambios que deberían realizarse en UbikSim son:

- Permitir que las personas virtuales puedan controlarse desde teclado y ratón.
- Añadir el objeto móvil Android al simulador.
- Ampliar las clases de comportamiento para que interaccione con el móvil.

[7] http://cs.gmu.edu/ eclab/projects/mason/

[8] http://www.sweethome3d.com

- Ofrecer los servicios para que el móvil o emulador se pueda comunicar con el simulador.

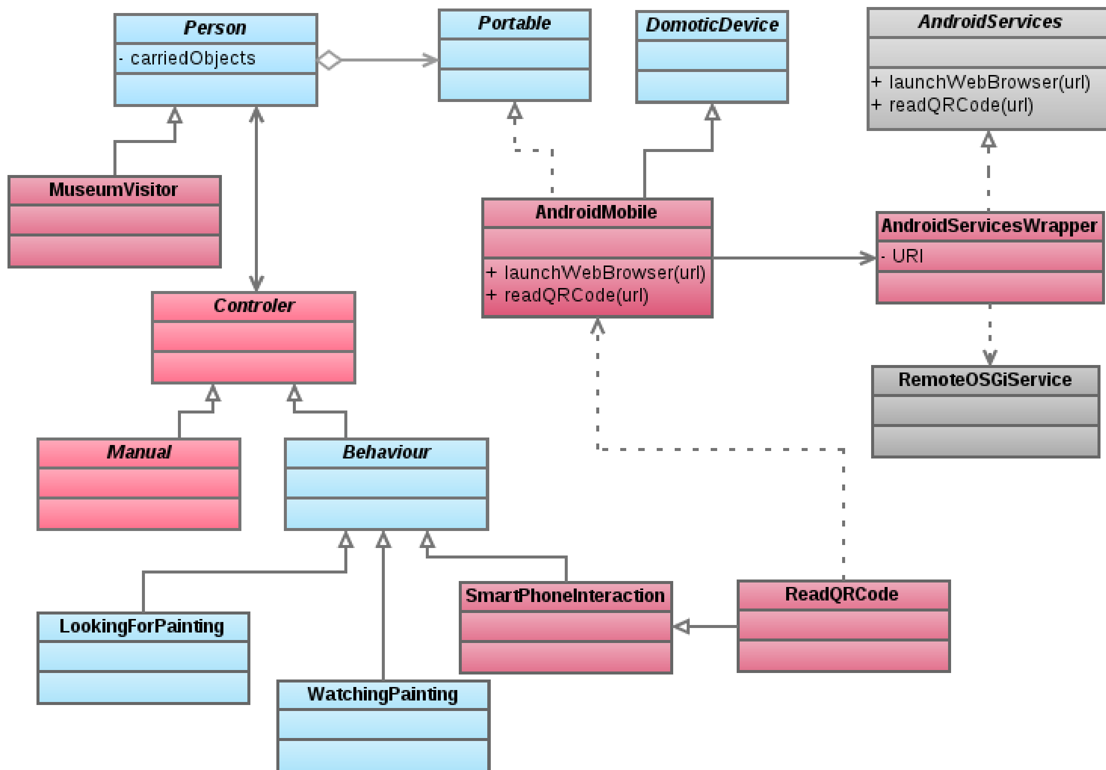


Figura 4 Diagrama de clases de UbikSim.

Recordemos que como el trabajo se centra en la plataforma relativa al móvil y concretamente en la parte que permite "engañar" al móvil, no vamos a entrar en detalles en este apartado. Como esbozo a una solución de los cambios necesarios para UbikSim, mostramos el diagrama de clases de la Figura 4 donde las clases de color rojo son las que habría que añadir. En UbikSim se pueden simular personas con la clase `Person` que implementan la interfaz `Stepable` para que puedan actuar en la simulación. Además, éstas pueden transportar objetos que implemente la interfaz `Portable`, por ello, la clase que representa el teléfono móvil Android (`AndroidMobile`) tiene que implementar esta interfaz. Cada vez que se quiera añadir un nuevo rol de persona, por ejemplo, un visitante de museo (clase `MuseumVisitor`), ésta extiende la clase `Person`. En UbikSim, las personas se controlaban de forma autónoma a través de autómatas que modelaban el comportamiento. Para ello usa un patrón compuesto de comportamientos, clase `Behaviour`. Cada clase representa un tipo de interacción o una acción concreta. Por ejemplo, una de ellas es la que representa la interacción de la persona con el teléfono móvil, clase `SmartPhoneInteraction`. Pero también queremos que una persona virtual se pueda controlar manualmente por un usuario usando la clase `UserController`. Para desacoplar este control y que un personaje se pueda controlar indistintamente de una u otra forma, se crea la clase abstracta `PersonController` que debe recoger todas las

acciones que puede realizar una persona. A su vez, una subacción puede consistir en escanear un código QR. Por último, la clase AndroidMobile usa la clase `AndroidServicesWrapper` para poder acceder a los servicios remotos del teléfono Android real. Esta clase tiene por ejemplo la URI donde se encuentra accesible el móvil real asociado.

Una vez visto R-OSGi y el simulador, pasamos al diseño de la plataforma relativa al teléfono móvil Android o el emulador. Esta es la parte que más nos interesa y la de más valor a nivel innovador.

EL TELÉFONO ANDROID O EL EMULADOR

En esta sección veremos el diseño de los componentes software necesarios para permitir que las aplicaciones Android que se quieran validar se comuniquen con el simulador de la forma más sencilla para el desarrollador. Uno de los objetivos principales en esta parte del sistema es que el código fuente tenga que sufrir las menos modificaciones posibles al pasar de la versión de validación a la versión comercial.

Como mostraba la Figura 2, los componentes software del móvil se dividen en dos partes. Una que se basa en el marco de trabajo OSGi para que el móvil pueda utilizar los servicios del simulador, y viceversa, ambos de forma remota, y otra compuesta por una capa intermedia que usa la aplicación para comunicarse con los módulos OSGi del teléfono.

Figura 5 Aplicación Android para soportar OSGi de ProSyst.

MARCO DE TRABAJO DE OSGI MÓVIL

Por ahora, las aplicaciones Android no se pueden construir como módulos OSGi de la misma forma que se crean el resto. Hay una compañía (*ProSyst*) que ofrece una aplicación Android para ejecutar módulos OSGi, ver Figura 5. Con *OSGi Mgmt* se gestionan lo módulos OSGi: instalarlos, desinstalarlos, iniciarlos, pararlos, etc. Al iniciar *OSGi Apps* nos muestra

un escritorio con los módulos que son aplicaciones Android, es decir, aquellos módulos que tienen interfaz gráfica y pueden usar los recursos del teléfono. Además de estas aplicaciones, *Prosyst* proporciona un marco de trabajo (*mBS Mobile SDK*) para desarrollar aplicaciones Android como módulos, pero no puede aprovechar el entorno que ofrece Android para Eclipse (a través del plugin ADT). Por ello, las aplicaciones Android se deben desarrollar a base de escribir código a mano, incluidas las interfaces gráficas.

Un módulo en OSGi es un paquete jar que contiene en su manifiesto información extra relativa a éste, como interfaces que publica y módulos que usa. Este módulo se gestiona a través del marco de trabajo que ofrece OSGi. Por el contrario, una aplicación Android se compone de varios elementos: xmls (para la interfaz gráfica, el manifiesto con permisos para usar recursos) y ficheros java que se compilan a través de la herramienta ADT generando un fichero apk que se instala en el sistema operativo Android aceptando que la aplicación use los recursos indicados en su manifiesto. De hecho, el marco de trabajo OSGi en Android es una aplicación más que solicita acceso a todos los recursos para que así, los módulos que se instalen en éste como aplicaciones OSGi Android puedan acceder a lo que necesiten.

En este punto surge un problema cuando se intenta usar el módulo de OSGi remoto en el teléfono móvil Android. Este problema radica en que Android se programa en Java pero no se compila igual que las aplicaciones normales y, por tanto, no se ejecuta sobre la máquina virtual de java. En su lugar, Android tiene su propia máquina virtual llamada Dalvik que ejecuta una versión de los *bytecodes* de java optimizados. Por ello, cuando se quiere usar un servicio remoto, lo que se recibe son los *bytecodes* normales de java que no se pueden ejecutar en Android. De ahí que exista el módulo *DexService*. Éste es el responsable de traducir los bytecodes de java a los que usa Android. A la inversa no hay problema porque los módulos OSGi en Android contienen, dentro del jar, tanto los *bytecodes* java como un fichero llamado classes.dex que es la traducción de los *bytecodes* y este último fichero simplemente no es usado por las aplicaciones no Android.

El módulo *ServiceManager* es el que se encarga de retransmitir y gestionar los servicios entre el simulador y el midleware sobre el que se desarrolla la aplicación real. Para que el diseño sea modular y flexible, este módulo usa otros dos más:

- `AndroidServicesImpl` implementa la interfaz del módulo *AndroidServices* que se encarga de ofrecer los servicios al simulador para retransmitírselos a la aplicación. Por ejemplo, una notificación indicando la posición del móvil a través de un proveedor GPS o torres de teléfono.

- `UbikSimServicesManager` usa la interfaz `UbikSimServies` para usar los servicios del simulador de forma remota. Por ejemplo, informar al simulador de que quiere recibir la temperatura donde se encuentre el móvil virtual con una frecuencia determinada.

Este diseño que separa la interfaz de la implementación nos permite cambiar el módulo correspondiente a la implementación incluso de forma dinámica sin tener que para el sistema. Por ejemplo, podríamos tener varias versiones de los módulos que difieran en la forma de publicar los servicios: a través de R-OSGi, ofreciendo un servicio web, etc.

En la Figura 6 se muestra el diagrama de clases básico agrupado por módulos. La clase `SericeManager` es la clase principal del módulo que lleva el mismo nombre. Como todo módulo, que quiera publicar y usar servicios, implementa la interfaz `BundleActivator` del Framework OSGi. Además, esta clase implementa la interfaz `ApplicationFactory` para poder usar la clase `ActivityWrapper` y registrarse como aplicación a través de la clase ApplicationRegistry. La clase `ActivityWrapper` es la que nos permite acceder a los servicios del sistema operativo Android a través del parámetro (con el contexto del sistema) que ofrece el método `createView()`. El servicio que más nos interesa está relacionado con los Intents que se utilizan como mecanismo de comunicación entre aplicaciones. En nuestro caso, el módulo `ServiceManager` y la aplicación Android a validar. Cuando se registra la aplicación OSGi a través de `ApplicationRegistry`, ésta se muestra en el escritorio de la aplicación OSGi Apps de ProSyst.

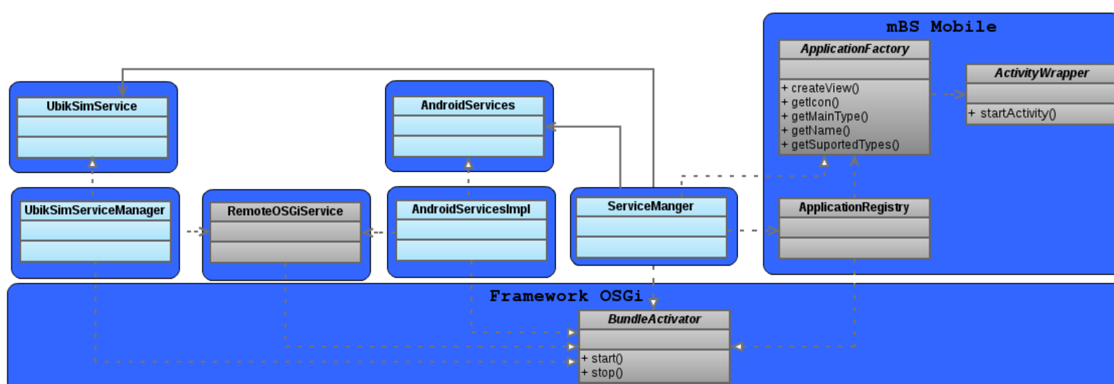


Figura 6 Diagrama de clases agrupados por módulos OSGi en el teléfono inteligente.

La clase `ServiceManager` usa los servicios `UbikSimServiceManager` y `AndroidService` y ambos, a su vez, usan el servicio para obtener servicios remotos.

CAPA ANDROID DE VALIDACIÓN

El objetivo de este apartado es diseñar un componente software que esté ligado a la aplicación para que ésta se pueda validar a través del simulador. Este componente no será necesario cuando la aplicación se compile para su versión comercial. El principal requisito para el diseño de este componente software es no tener que modificar el código fuente cuando ésta se compila para ser validada o para usarla en el entorno real y comercial. Este requisito es importante por dos motivos principales. Primero, tener que realizar cambios en el código fuente sería una foco de posibles errores. Y segundo, si no hay que realizar

cambios significa que este componente software no es intrusivo y no hay que lidiar con ninguna API nueva.

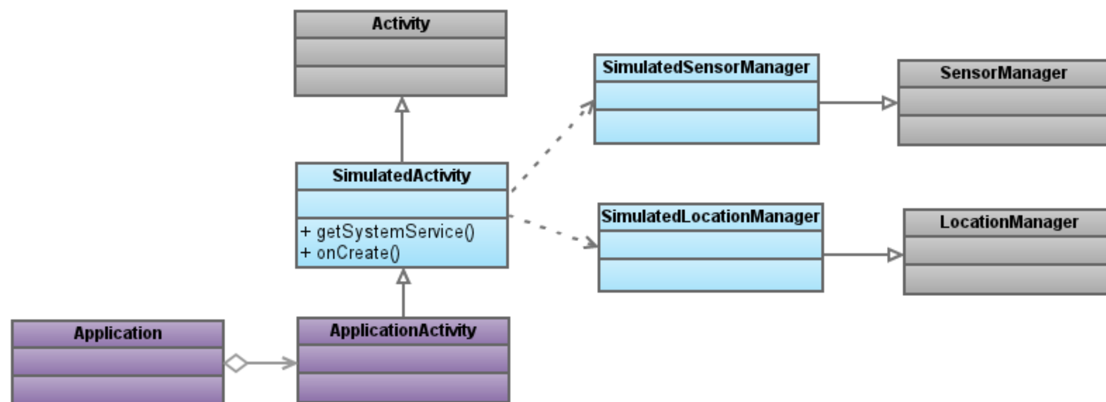


Figura 7 Ejemplo del diagrama de clases de VAPI, API de validación.

Inicialmente se pensó en desarrollar una API de validación (VAPI) con la estructura que se muestra en la Figura 7. Cada clase `Activity` representa el elemento básico para generar aplicaciones Android con interfaz gráfica. A su vez, las aplicaciones Android se componen de una o varias clases Activity. Cada una de estas clases se asocia a una interfaz grá_ca, lo que en Java correspondería a la clase JFrame que modela a una ventana. En nuestro modelo para validar la aplicación, cada Actividad tendría que heredar de la clase `SimulatedActivity` que a su vez hereda de la clase `Activity`. En la clase `SimulatedActivity` se redefiniría el método `getSystemService()` que es el medio a través del cual se accede a los servicios del sistema. Por ejemplo, con este método podríamos acceder a la clase `SensorManager` encargada de gestionar los sensores, o al de localización con la clase `LocationManager`. Sólo se reemplazarían aquellos servicios que están relacionados con obtener información del entorno, por ejemplo, los que aparecen en la Figura 7. Así que cuando se desee otro servicio se devolverá la clase original ofrecida por Android y cuando no, se devolverá una clase ofrecida por la VAPI que tendrá el mismo nombre e interfaz que la original, pero ésta redirige las llamadas al simulador en lugar de a los drivers del móvil real. El único cambio que hay que hacer es modificar el paquete a importar. Y para hacerlo más sencillo, hacemos que se llamen igual pero cambiando el primer nombre del paquete, sim. Por ejemplo, la clase android.hardware.SensorManger se llamaría sim.android.hardware.SensorManger en la VAPI. De modo que sólo habría que suprimir sim. para que la aplicación se compilase para producción. Lo primero que se pensó para implementar estas clases de la VAPI fue heredar de las originales. Así sería más sencilla su implementación y se reutilizaría código. Además, de esta forma no haría falta modificar el paquete porque por la herencia se puede usar la clase hijo como si fuese la padre. Pero no se puede heredar de estas ya que los constructores son de tipo `protected` y además los métodos son de tipo final lo que significa que no se pueden redefinir. Por tanto, usar VAPI no sirve como solución que satisfaga el requisito de no tener que modificar el código. Señalar que hay otros medios para percibir el entorno y que no se

acceden a través del método `getSystemService()`, como, por ejemplo, la cámara del teléfono móvil. El acceso a imágenes a través de la cámara se hace usando la clase `Camera`.

Una alternativa a lo anterior, y la más prometedora, consiste en modificar el SDK de Android, ver Figura 8, y lo denominamos VSDK. De esta forma, tendríamos dos SDKs, uno para generar la aplicación a validar y otro, el oficial, para generar la aplicación para su comercialización o para una prueba piloto. Por tanto, el único cambio que hay que hacer es muy sencillo y seguro, cambiar el SDK que compila la aplicación. Esta solución es la mejor para el usuario debido a la sencillez, no tiene que aprender ninguna metodología nueva, ni API y tampoco tiene que cambiar el código fuente de su aplicación. Sin embargo, esta solución es más difícil para nosotros ya que conlleva tener que manejar el código fuente del SDK de Android.

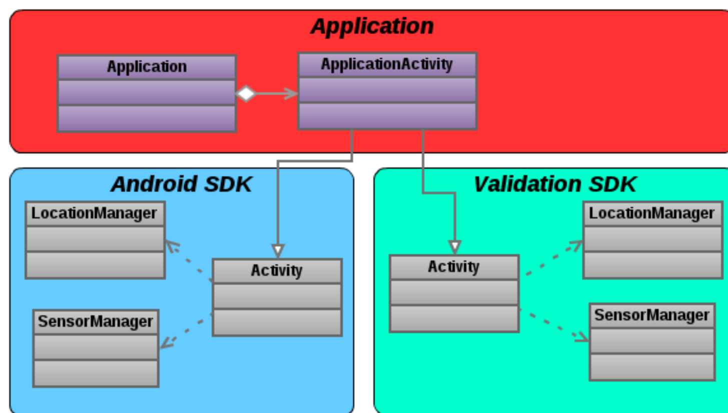


Figura 8 Dos marcos de trabajo, uno oficial de Android y otro para validación (VSDK).

7. EJEMPLO PRÁCTICO

El escenario de ejemplo que se va a utilizar consiste en desarrollar una primera aproximación de una aplicación basada en realidad aumentada para mostrar material multimedia en un museo. Este tipo de aplicaciones son muy comunes en el dominio de los museos. Esta aplicación se instalaría en el móvil del usuario antes de iniciar la visita. Entonces, cuando el usuario esté frente a una obra de arte (p.e. un cuadro), le dice a la aplicación qué cuadro es a través de la lectura de un código QR localizado junto al cuadro. A partir de la lectura del código, la aplicación obtiene la información necesaria relativa al cuadro. Este código contiene una dirección url a la entrada de la wikipedia que corresponda aunque este podría contener cualquier información que nos permita acceder al material del cuadro. Una versión más avanzada de la aplicación podría detectar el cuadro con la cámara y realizar realidad aumentada sobre este mostrando o resaltando partes del cuadro que tendrían asociada información. De esta forma se segmentaría la información ofrecida al visitante del museo. La aplicación también podría ser social integrándole la funcionalidad de comentar y compartir las experiencias de cada pieza visitada en las redes sociales típicas (p.e. Twitter y Facebook). Esos comentarios los captaría el museo de forma que podría ayudar a mejorar el diseño de los contenidos asociados a cada objeto en exposición o incluso la disposición de dichos objetos.

El ejemplo se reduce a una sala de un museo hipotético que contiene cuatro pinturas muy famosas: el Guernica, las meninas, el grito y la Monalisa. Cada cuadro tiene asociado un código QR que contiene la url a la entrada a la wiki. Sólo hay un visitante del museo con un teléfono Android y ambos son controlados por el usuario que va a validar la aplicación.

El objetivo de este apartado es mostrar los pasos que tiene que realizar un desarrollador que quiere utilizar la infraestructura expuesta en el diseño para aplicarla al ejemplo expuesto. Lo primero que hay que hacer es modelar todos los elementos de la sala del museo con la herramienta que proporciona UbikSim, SweetHome 3D. Posteriormente, generar las imágenes de los códigos QR que contengan la url a la entrada de la wikipedia del cuadro que corresponda. También, generar la aplicación Android para la lectura de códigos QR a validar utilizando la VAPI. No se ha usado el VSDK debido a problemas de compilación que todavía no los hemos resuelto. Por último, lanzar todo el entorno para validar la aplicación.

MODELADO DEL MUSEO

El modelado de la sala del museo se realiza con el editor de UbikSim, Ubik Editor, ver Figura 9. El editor se compone de cuatro paneles:

- arriba a la izquierda podemos ver la paleta donde se encuentran los objetos a insertar en el mundo. Para insertarlos sólo hay que arrastrarlos al panel de edición donde se muestra la vista de la planta del mundo.

- arriba a la derecha se encuentra el panel de edición. En este panel creamos el mundo (suelo, paredes) e insertamos sus elementos.
- abajo a la izquierda podemos ver una lista de los elementos ya insertados en los mundos con algunas de sus características.
- abajo a la derecha se encuentra la vista del mundo 3D renderizado.

Esta vista nos sirve para comprobar que estamos creando el mundo virtual realmente como queremos. Para modelar la sala de museo lo primero que hemos hecho ha sido crear el suelo y las paredes utilizando la paleta de herramientas en la parte superior. Después, se han añadido los 4 cuadros y los 4 códigos QR próximos a cada cuadro. Por último, hemos añadido al visitante del museo y le hemos asociado un móvil. Para asociar un móvil virtual a una persona con el editor simplemente hay que colocar el móvil encima de la persona, ver Figura 9.

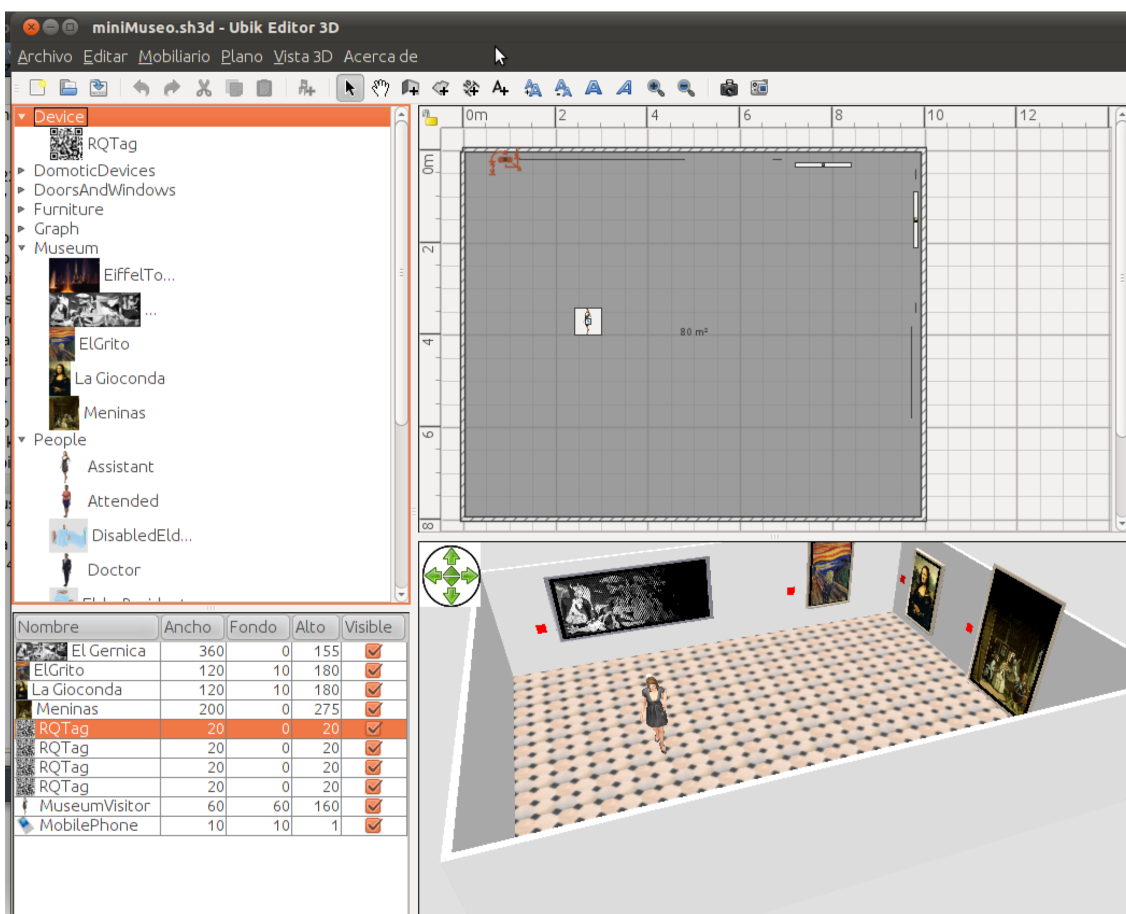


Figura 9 Modelado del museo con el editor Ubik Editor 3D.

TRABAJO RELATIVO A LOS CÓDIGOS QR

Cada cuadro tiene asociado un código QR que contiene una url a la información relativa a esta. Generar este código es muy simple ya que hay numerosas páginas web que ofrecen este servicio gratuitamente. Nosotros hemos usado la web *http://www.codigos-qr.com/generador-de-codigos-qr/* donde simplemente se introduce la url que se quiere codificar, el tamaño y pulsar el botón. En un segundo se nos muestra la imagen del código QR en formato PNG, ver Figura 10 y la guardamos.

El siguiente paso es incrustar el código QR en una imagen que tenga la misma resolución que la pantalla del móvil.

Una vez obtenida la imagen completa que se mostrará en la pantalla, hay que convertirla a formato yuv debido a que es en este formato en el que la cámara real entrega las imágenes a las aplicaciones. Existen muchas aplicaciones y comandos para realizar esta tarea. Nosotros hemos usado el comando `convert` del paquete ImageMagick10. El formato del comando usado es:

```
convert in.png -depth 8 -sampling-factor 4:2:0 out.yuv
```

Cuando ya disponemos de la imagen que realmente pasaremos a la aplicación, como si fueste tomada por la imagen, hay que publicarla en el servidor web para que sea accesible por el teléfono. Y por último, sólo queda indicar en cada etiqueta QR virtual la url donde se encuentra la imagen del correspondiente código QR en el servidor, ver Figura 11.

Figura 10 Código QR generado desde la web que contiene la url: *http://es.wikipedia.org/wiki/Guernica_(cuadro)*.

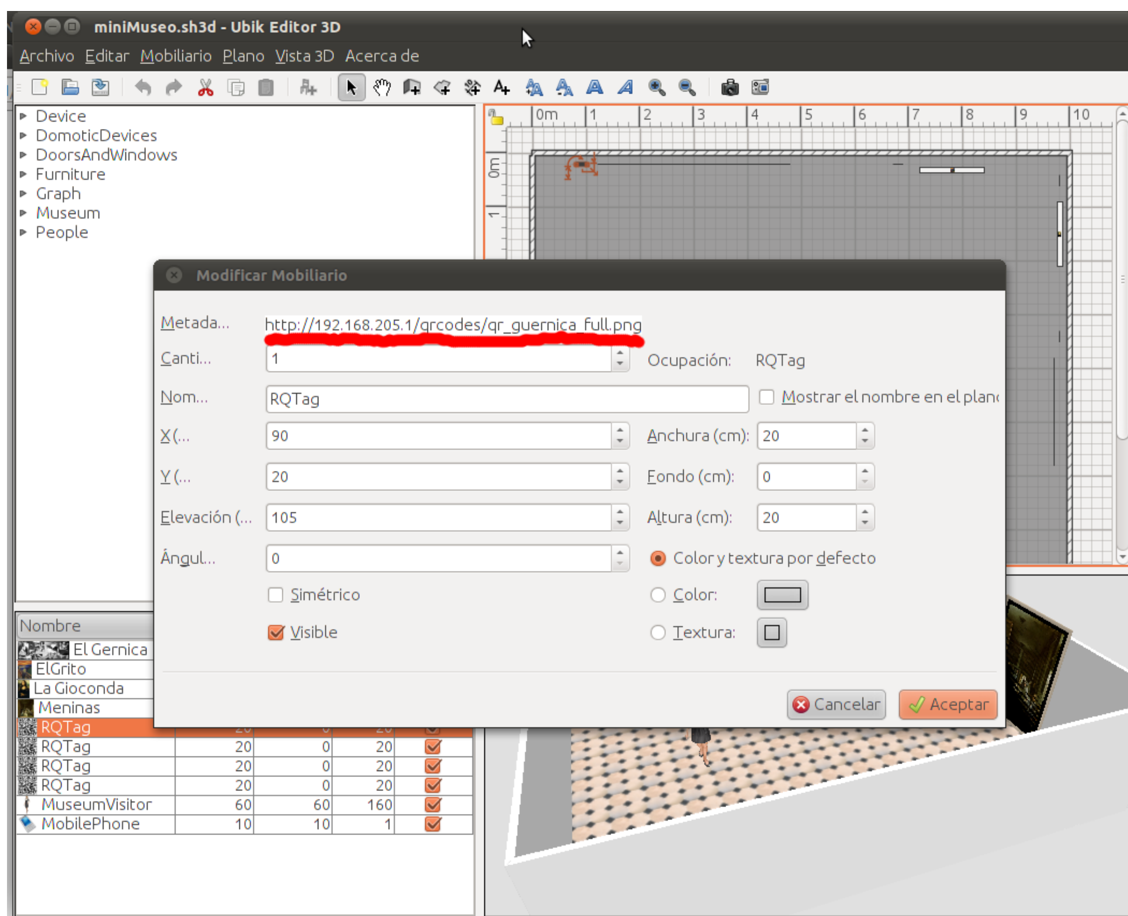


Figura 11 Editando etiqueta QR virtual para insertar la url de la imagen en editor Ubik Editor 3D.

DESARROLLAR LA APLICACIÓN DE LECTURA DE CÓDIGOS QR (ZXING)

El desarrollo de la aplicación, como ya hemos predicho, se realiza como cualquier otra aplicación Android. Para ello hemos utilizado el entorno de desarrollo Eclipse con el plugin ADT de Android.

Para ahorrarnos tiempo de programación y, de paso, para probar que la VAPI funciona correctamente, hemos utilizado la aplicación zxing[9] de código libre. Así que hemos tomado este código fuente y hemos generado un proyecto realizando los siguientes cambios para que se pueda validar:

1. Importar la librería para la validación.
2. Cambiar la clase Camera a importar en todas aquellas clases en las que se utilice. Esto significa buscar todos los ficheros donde se encuentre la sentencia

```
import android.hardware
```

[9] http://code.google.com/p/zxing/

y cambiarla por la sentencia

```
import sim.android.hardware.Camera
```

3. Indicar en el manifiesto del proyecto que la aplicación utiliza el servicio de simulación de la cámara ofrecido por OSGi.

Estos son todos los cambios que hay que realizar y como se puede comprobar éstos son mínimos y muy sencillos. Si utilizásemos el VSDK no se tendría que realizar el paso 2 que es el más engorroso. Simplemente habría que cambiar el SDK a utilizar desde las propiedades del proyecto en Eclipse.

Como se puede comprobar, la única clase que utiliza la aplicación zxing que interviene con el simulador es la clase `Camera`. Si se usase otras, como por ejemplo el SensorManager, habría que realizar el paso 2 con ésta para cambiar la clase a importar.

PUESTA EN MARCHA DE LA INFRAESTRUCTURA DE VALIDACIÓN

Una vez generados todos los artefactos, la puesta en marcha es muy sencilla. El primer paso consiste en lanzar el entorno OSGi del PC donde se encuentra el módulo de UbikSim y el resto. Para ello, se puede utilizar cualquier implementación de OSGi. Nosotros ofrecemos un paquete con la implementación de OSGi Apache Felix[10] y todos los módulos necesarios. Con éste, el desarrollador sólo tiene que configurar el fichero de propiedades de UbikSim para que utilice el escenario del museo modelado. Para lanzar el entorno simplemente hay que ejecutar en la consola:

```
<felix root>$java –jar bin/felix.jar
```

Con este comando se inician todos los módulos y se muestra la ventana de configuración del simulador UbikSim. No es necesario modificar nada, pulsar el botón de comenzar y se iniciará el entrono virtual.

Una vez iniciado todos los componentes necesarios en el PC, hay que pasar a los del móvil. Éste necesita básicamente dos componentes: aplicación OSGi de ProSyst con los módulos necesarios y la aplicación a validar compilada con la VAPI. Para conseguir la aplicación se puede usar el Android Market disponible en cualquier teléfono Android. Esta aplicación da acceso al extensísimo mercado de aplicaciones Android, tanto gratuitas como de pago. Para descargar la que nos interesa hay que iniciar el Market y buscar "mBS Mobile OSGI". La seleccionamos e instalamos aceptando las condiciones. Cuando ya disponemos de la plataforma OSGi en el móvil hay que instalar los módulos necesarios. Para ello, primero hay que copiarlos en la memoria del móvil. Una forma sencilla de copiarlos es poniendo el dispositivo en modo de almacenamiento. Una vez copiados los módulos necesarios iniciamos la aplicación *OSGi Mngn.* Ésta dispone de una opción para instalar módulos seleccionándolos desde un explorador de archivos en el teléfono. Los archivos hay que

[10] http://felix.apache.org

instalarlos uno a uno, pero sólo hay que hacerlo una vez. La próxima vez simplemente hay que lanzar el entorno OSGi.

Por último, nos queda iniciar la aplicación a validar como cualquier otra aplicación Android. En este momento se puede empezar a validar la aplicación interactuando con el mundo virtual a través del teclado, el ratón y el teléfono móvil.

Puede parece un poco pesado iniciar el entorno, pero una vez que se haya hecho, las siguientes veces será mucho más rápido. En el PC sólo habrá que modificar si se quiere el modelo del mundo y lanzar OSGi. En el móvil también sólo habrá que lanzar el entorno OSGi y por último iniciar la aplicación a validar.

8. PROYECTO REAL EN EL MAM

En este apartado describimos un proyecto que se inició a mediados de julio en colaboración con Luis Enrique de Miguel Santed, conservador responsable del Museo Arqueológico de Murcia (MAM[11]) y la Fundación Integra[12] que nos servirá para aplicar y mejorar la infraestructura para la validación de aplicaciones móviles fruto de este trabajo. El conservador del museo, Luis, es el comprador de la aplicación y el que marca los requisitos de ésta. La fundación integra está promovida por la Comunidad de Murcia y su misión fundamental es acelerar el proceso de implantación de la Sociedad de la Información en la Región de Murcia. La fundación nos proporcionará el material multimedia relativo a las obras del museo y de las que la aplicación se nutrirá.

El objetivo del proyecto es desarrollar una aplicación para teléfonos móviles que ofrezca servicios al visitante del museo arqueológico de forma contextual de tal forma que hagan la visita más interesante y formativa. La aplicación deberá ser integral e incorporar toda la funcionalidad de lectura de códigos QR, posicionamiento en interiores, visor de realidad aumentada y soporte para redes sociales.

Aunque sólo realizamos 2 reuniones, obtuvimos algunas conclusiones:

- El museo tendrá que ofrecer acceso a su red WiFi para poder descargarse la aplicación y los contenidos durante la visita, ya que no todos los usuarios disponen de tarifa de datos y, aunque la tuviesen, no estarían dispuestos a gastar gran parte de sus limitados megas en descargarse contenidos multimedia.

- La información que se muestre al usuario debe ser estructurada de alguna forma. No resulta atractivo, sino más bien desmotivador, mostrar toda la información relativa a cada pieza. Por ejemplo, la información se podría estructurar por campos: ubicación, materia, técnica, dimensiones, descripción, etc. y el visitante ver aquellos que le interesa.

- Establecer la primera fase del proyecto.

La primera fase del proyecto consistirá en desarrollar la funcionalidad relativa a la lectura de códigos QR asociados a las piezas arqueológicas de una sala del museo. La sala elegida ha sido la dedicada al románico. En esta sala hay piezas grandes aisladas, ver Figura 12, y piezas pequeñas agrupadas dentro de vitrinas, ver Figura 13. La sala dispone de unas 200 piezas pero para la primera fase se va a considerar una pieza por vitrina y sólo se

[11] http://www.murciaturistica.es/museos/museos.inicio?museo=museo-arqueol%F3gico-de-murcia-(mam)&id=1

[12] http://www.f-integra.org/

considerará una vitrina completa, la que aparece en la Figura 13. De esta forma se reduce el número de piezas a 22.

Figura 12 Ejemplo de gran pieza aislada del museo.

Figura 13 Ejemplo de vitrina con varias piezas pequeñas.

Por tanto, ahora estamos en la primera tarea del proceso que consiste en modelar la sala dedicada al románico con la herramienta Ubik Editor. Para facilitar la tarea, el conservador del museo nos proporcionó el plano de la planta donde se encuentra la sala, ver Figura 14. Esta imagen se puede insertar como fondo en el editor, de forma que es muy sencillo trazar los muros guardando las dimensiones.

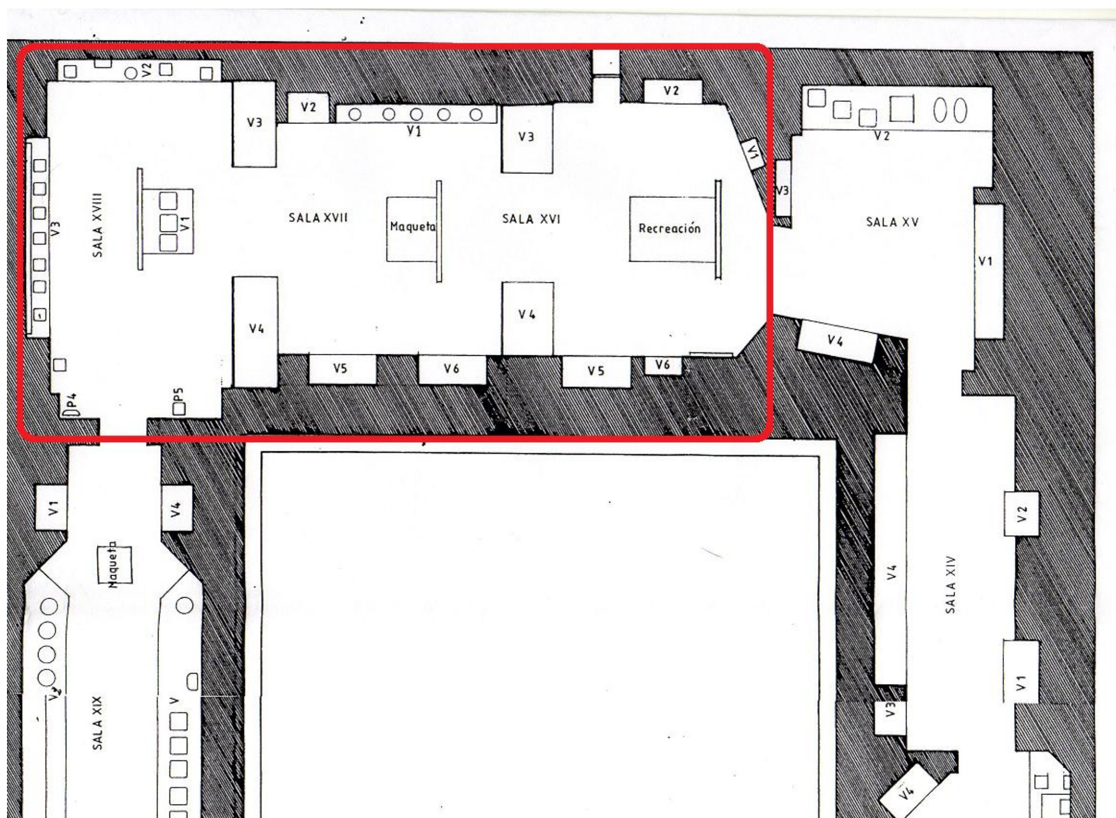
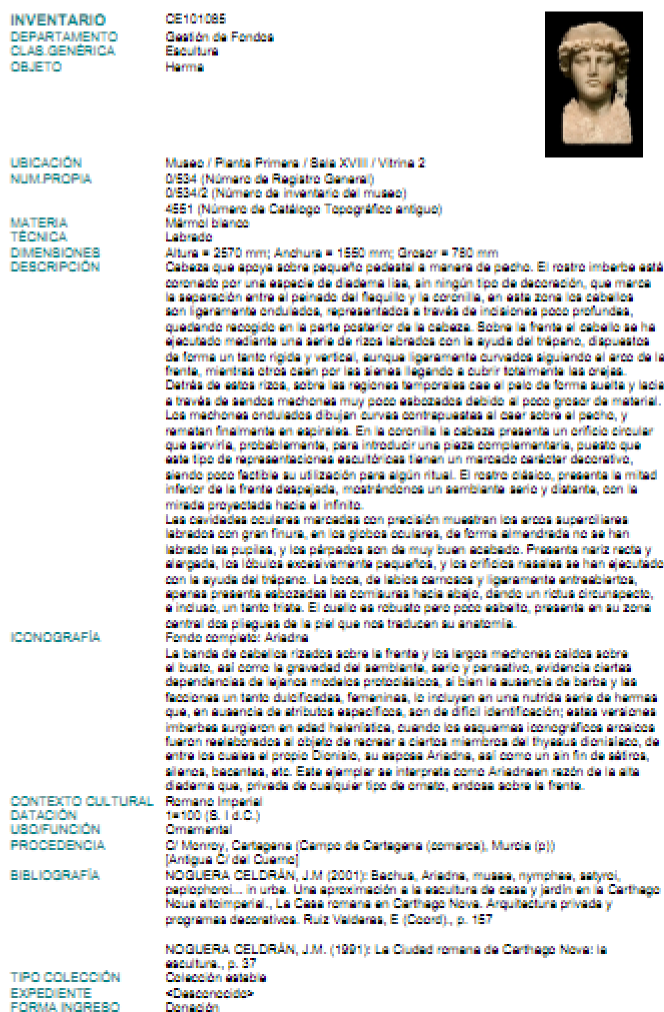


Figura 14 Plano de la planta donde se encuentra la sala dedicada al románico, enmarcada en rojo.

Por último, para cuando se termine de modelar la sala y completar la primera fase, el conservador nos facilitó las 22 fichas asociadas a las piezas, ver ejemplo de ficha en Figura 15. Estas fichas habrá que publicarlas en un servidor web y generar los códigos QR con las urls de éstas.



INVENTARIO	CE101085
DEPARTAMENTO	Gestión de Fondos
CLAS.GENÉRICA	Escultura
OBJETO	Herma
UBICACIÓN	Museo / Planta Primera / Sala XVIII / Vitrina 2
NUM.PROPIA	0/534 (Número de Registro General) 0/534/2 (Número de inventario del museo) 4551 (Número de Catálogo Topográfico antiguo)
MATERIA	Mármol blanco
TÉCNICA	Labrado
DIMENSIONES	Altura = 2570 mm; Anchura = 1550 mm; Grosor = 780 mm
DESCRIPCIÓN	Cabeza que apoya sobre pequeño pedestal a manera de pecho. El rostro imberbe está coronado por una especie de diadema lisa, sin ningún tipo de decoración, que marca la separación entre el peinado del flequillo y la coronilla, en esta zona los cabellos son ligeramente ondulados, representados a través de incisiones poco profundas, quedando recogido en la parte posterior de la cabeza. Sobre la frente el cabello se ha ejecutado mediante una serie de rizos labrados con la ayuda del trépano, dispuestos de forma un tanto rígida y vertical, aunque ligeramente curvados siguiendo el arco de la frente, mientras otros caen por las sienes llegando a cubrir totalmente las orejas. Detrás de estos rizos, sobre las regiones temporales cae el pelo de forma suelta y lacia, a través de sendos mechones muy poco esbozados debido al poco grosor de material. Los mechones ondulados dibujan curvas contrapuestas al caer sobre el pecho, y rematan finalmente en espirales. En la coronilla la cabeza presenta un orificio circular que serviría, probablemente, para introducir una pieza complementaria, puesto que este tipo de representaciones escultóricas tienen un marcado carácter decorativo, siendo poco factible su utilización para algún ritual. El rostro clásico, presenta la mitad inferior de la frente despejada, mostrándonos un semblante serio y distante, con la mirada proyectada hacia el infinito. Las cavidades oculares marcadas con precisión muestran los arcos superciliares labrados con gran finura, en los globos oculares, de forma almendrada no se han labrado las pupilas, y los párpados son de muy buen acabado. Presenta nariz recta y alargada, los lóbulos excesivamente pequeños, y los orificios nasales se han ejecutado con la ayuda del trépano. La boca, de labios carnosos y ligeramente entreabiertos, apenas presenta esbozadas las comisuras hacia abajo, dando un rictus circunspecto, e incluso, un tanto triste. El cuello es robusto pero poco esbelto, presenta en su zona central dos pliegues de la piel que nos traducen su anatomía.
ICONOGRAFÍA	Fondo completo: Ariadna La banda de cabellos rizados sobre la frente y los largos mechones caídos sobre el busto, así como la gravedad del semblante, serio y pensativo, evidencia ciertas dependencias de lejanos modelos protoclásicos, si bien la ausencia de barba y las facciones un tanto dulcificadas, femeninas, lo incluyen en una nutrida serie de hermas que, en ausencia de atributos específicos, son de difícil identificación; estas versiones imberbes surgieron en edad helenística, cuando los esquemas iconográficos arcaicos fueron reelaborados al objeto de recrear a ciertos miembros del thyasus dionisíaco, de entre los cuales el propio Dionisio, su esposa Ariadna, así como un sin fin de sátiros, silenos, bacantes, etc. Este ejemplar se interpreta como Ariadnaen razón de la alta diadema que, privada de cualquier tipo de ornato, endosa sobre la frente.
CONTEXTO CULTURAL	Romano Imperial
DATACIÓN	1=100 (S. I d.C.)
USO/FUNCIÓN	Ornamental
PROCEDENCIA	C/ Monroy, Cartagena (Campo de Cartagena (comarca), Murcia (p)) [Antigua C/ del Cuerno]
BIBLIOGRAFÍA	NOGUERA CELDRÁN, J.M (2001): Bachus, Ariadna, musae, nymphae, satyroi, peplophoroi... in urbe. Una aproximación a la escultura de casa y jardín en la Carthago Noua altoimperial., La Casa romana en Carthago Nova. Arquitectura privada y programas decorativos. Ruiz Valderas, E (Coord)., p. 157 NOGUERA CELDRÁN, J.M. (1991): La Ciudad romana de Carthago Nova: la escultura., p. 37
TIPO COLECCIÓN	Colección estable
EXPEDIENTE	<Desconocido>
FORMA INGRESO	Donación

Figura 15 Ejemplo de ficha asociada a una pieza románica de la sala.

9. CONCLUSIONES Y TRABAJO FUTURO

El propósito de la tesis de máster era el de conseguir realizar una plataforma que nos permita validar las aplicaciones de teléfonos móviles (en concreto Android) sin tener que utilizar hardware y equipo que en ocasiones puede ser costoso y/o difícil de desplegar. Para ello hemos hecho uso de la tecnología OSGi que nos permite interconectar la plataforma del PC donde está el simulador con la del móvil donde se encuentra la aplicación a validar. El único elemento que conecta la aplicación con OSGi es la VAPI o el VSDK, diseñados por nosotros, y que son fácilmente eliminados en el proceso de compilación. Nuestra arquitectura consigue alcanzar los objetivos de forma óptima ofreciendo una infraestructura para a partir de un proyecto de una aplicación Android general obtener la versión para su validación usando el VAPI o VSDK. Al no tener que modificar el código se minimiza al máximo la posibilidad de cometer errores al cambiar de la versión de validación a la de comercialización. El proceso para desarrollar la aplicación del móvil es exactamente el mismo que se sigue para cualquier aplicación Android, lo que implica que se puede crear cualquier programa que se desee con el aspecto que uno quiera. Con esta versión de la aplicación y lanzando los entornos OSGi (PC y móvil), la infraestructura nos permite obtener las ventajas de una prueba de campo pero con un coste menor de tiempo en la coordinación y gestión, y una reducción del coste económico. Además, podemos ofrecer un entorno más controlado para que las pruebas no produzcan resultados inútiles. La desventaja respecto a la prueba piloto es que el usuario se tiene que desplazar con el teclado y el ratón alejándose un poco de la realidad. Pero la interacción con el móvil, que suele ser la parte más importante cuando un usuario valida una aplicación, es totalmente real ya que la interacción con el móvil virtual se hace a través del móvil real.

Se han realizado diversas pruebas que confirman la viabilidad de la propuesta, e incluso, tal y como se describen el apartado 7 esta metodología se ha utilizado en un caso real para la validación de una aplicación para lectura de códigos QR en un museo con resultados positivos, por lo que consideramos que hemos alcanzado los objetivos inicialmente planteados. Aun así, para que la infraestructura se complete queda mucho trabajo por realizar. Enumeramos algunos de los aspectos que consideramos más interesantes:

1. Generar el SDK de validación (VSDK) para que realmente no haya que realizar cambios en el código fuente.

2. Poder simular la cámara es un avance muy importante ya que muchas aplicaciones la utilizan. Pero se podrían mejorar varios aspectos. Primero, ofrecer un servicio en el PC que dada cualquier imagen y dependiendo del móvil, la procesase para que el móvil recibiera la imagen en la resolución y formato adecuado (yuv). Así se eliminaría el engorro de estar generando las imágenes manualmente. Además, este servicio lo podría utilizar UbikSim para enviar capturas del mundo virtual.

3. Completar el resto de servicios para soportar sensores, sonido y localización.

4. Referente a la validación de sistemas donde se simularía al usuario y su interacción con el móvil habría que realizar bastante más trabajo. Lo más importante sería establecer un mecanismo que gestionase los emuladores. Por cada móvil virtual Android se tendría que lanzar su equivalente emulador donde corriese la aplicación. Además, también habría que implementar los comportamientos de los usuarios y su interacción con el móvil, y más importante y complicado, reflejar los eventos que se realizasen en el móvil virtual en el móvil real.

REFERENCIAS

1. John J. Barton and Vikram Vijayaraghavan. Ubiwise, a ubiquitous wireless infrastructure simulation environment. HP LABS, 2002.

2. Francisco Campuzano, Teresa Garcia-Valverde, Alberto Garcia-Sola, and Juan Botia. Flexible simulation of ubiquitous computing environments. In Paulo Novais, Davy Preuveneers, and Juan Corchado, editors, Ambient Intelligence - Software and Applications, volume 92 of Advances in Intelligent and Soft Computing, pages 189_196. Springer Berlin / Heidelberg, 2011. 10.1007/978-3-642-19937-024:

3. Gartner Corporation. Gartner says worldwide mobile phone sales grew 35 percent in third quarter 2010; smartphone sales increased 96 percent. http://www.gartner.com/it/page.jsp?id=1466313, Nov 2010.

4. Henry Been-Lirn Duh, Gerald C. B. Tan, and Vivian Hsueh-hua Chen. Usability evaluation for mobile device: a comparison of laboratory and _eld tests. In Proceedings of the 8th conference on Human-computer interaction with mobile devices and services, MobileHCI '06, pages 181_186, New York, NY, USA, 2006. ACM.

5. Tao Gu, Hung Keng Pung, and Da Qing Zhang. Toward an osgi-based infrastructure for context-aware applications. IEEE Pervasive Computing, 3:66_74, October 2004.

6. Mieke Haesen, Joan De Boeck, Karin Coninx, and Chris Raymaekers. An interactive coal mine museum visit: prototyping the user experience. In Proceedings of the 2nd conference on Human System Interactions, HSI'09, pages 543_550, Piscataway, NJ, USA, 2009. IEEE Press.

7. Jonna Häkkilä and Jani Mäntyjärvi. Developing design guidelines for context-aware mobile applications. In Proceedings of the 3rd international conference on Mobile technology, applications & systems, Mobility '06, New York, NY, USA, 2006. ACM.

8. Cuixiong Hu and Iulian Neamtiu. Automating gui testing for android applications. In Proceeding of the 6th international workshop on Automation of software test, AST '11, pages 77_83, New York, NY, USA, 2011. ACM.

9. Dongsik Jo, Ungyeon Yang, and Wookho Son. Design evaluation using virtual reality based prototypes: towards realistic visualization and operations. In Proceedings of the 9th international conference on Human computer interaction with mobile devices and services, MobileHCI '07, pages 246_258, New York, NY, USA, 2007. ACM.

10. A Kaikkonen and T Kallio. Usability testing of mobile applications : A comparison between laboratory and field testing. Journal of Usability Studies, 1(1):4_16, 2005.

11. Jesper Kjeldskov and Connor Graham. A review of mobile hci research methods. In In L. Chittaro (Ed.), Mobile HCI, pages 317_335. Springer-Verlag, 2003.

12. Jesper Kjeldskov, Connor Graham, Sonja Pedell, Frank Vetere, Steve Howard, Rine Balbo, and Jessica Davies. Evaluating the usability of a mobile guide: The influence of location, participants and resources. Behaviour and Information Technology.

13. Jesper Kjeldskov and Jan Stage. New techniques for usability evaluation of mobile systems. International Journal of Human-Computer Studies, 60:599_620, 2004.

14. Sangyoon Lee, Tian Chen, Jongseo Kim, Sungho Han, Zhi-geng Pan, and Gerard J. Kim. Affective property evaluation of virtual product designs. In Proceedings of the IEEE Virtual Reality 2004, pages 207_, Washington, DC, USA, 2004. IEEE Computer Society.

15. Karin Leichtenstern, Elisabeth André, and Matthias Rehm. Using the hybrid simulation for early user evaluations of pervasive interactions. In Proceedings of the 6th Nordic Conference on Human-Computer Interaction: Extending Boundaries, NordiCHI '10, pages 315_324, New York, NY, USA, 2010. ACM.

16. Joshua Lifton and Joseph A. Paradiso. Dual Reality: Merging the Real and Virtual. In Proceedings of the First International ICST Conference on Facets of Virtual Environments (FaVE), July 2009.

17. Tony Manninen. Multimedia game engine as distributed conceptualization and prototyping tool - contextual virtual reality prototyping. In Proceedings IMSA2000 Conference, Las Vegas, NV, IASTED/ACTA Press 2000; 99_104, pages 7_8. University Press, 2000.

18. Reto Meier. Professional Android 2 Application Development. Wrox Press Ltd., Birmingham, UK, UK, 1st edition, 2010.

19. Jakob Nielsen and Thomas K. Landauer. A mathematical model of the finding of usability problems. In Proceedings of the INTERACT '93 and CHI '93 conference on Human factors in computing systems, CHI '93, pages 206_213, New York, NY, USA, 1993. ACM.

20. Antti Oulasvirta. Finding meaningful uses for context-aware technologies: the humanistic research strategy. In Proceedings of the SIGCHI conference on Human factors in computing systems, CHI '04, pages 247_254, New York, NY, USA, 2004. ACM.

21. Antti Oulasvirta, Esko Kurvinen, and Tomi Kankainen. Understanding contexts by being there: case studies in bodystorming. Personal Ubiquitous Comput., 7:125_134, July 2003.

22. Vinny Reynolds, Vinny Cahill, and Aline Senart. Requirements for an ubiquitous computing simulation and emulation environment. In Proceedings of the _rst international conference on Integrated internet ad hoc and sensor networks, InterSense '06, New York, NY, USA, 2006. ACM.

23. Ichiro Satoh. Flying emulator: Rapid building and testing of networked applications for mobile computers. In Proceedings of the 5th International Conference on Mobile Agents, MA '01, pages 103_118, London, UK, 2002. Springer-Verlag.

24. M. Swift. Android operating system is expected to surge past rivals. Los Angeles Times, Sep 2010.