Un Marco de Trabajo para la Integración de Arquitecturas de Software con Metodologías Ágiles de Desarrollo

Luis Vivas, Mauro Cambarieri, Nicolás García Martínez,
Marcelo Petroff, Horacio Muñoz Abbate.

Laboratorio de Informática Aplicada - Universidad Nacional de Río Negro
{lvivas, mcambarieri, ngarciam, mpetroff ,hmunoz}@unrn.edu.ar

Abstract. La construcción de software dentro de un marco metodológico ágil ofrece la posibilidad de contar con procesos livianos y simples, aplicando técnicas de programación que permiten expresar el concepto de agilidad, garantizando código de calidad desde el inicio del proceso de desarrollo. Algunas técnicas que han sido probadas en metodologías tradicionales de desarrollo de software son usualmente utilizadas en metodologías ágiles, como por ejemplo las pruebas de unidad. En este trabajo proponemos un marco de trabajo basado en una arquitectura en capas, permitiendo guiar el desarrollo de software por medio de una técnica de programación centrada en pruebas de unidad, la cual fue formalizada en una metodología ágil de desarrollo - eXtreme Programming. La contribución es un marco de trabajo que permite la integración de una arquitectura de software con la técnica de programación guiada por pruebas de unidad, y la identificación de tecnologías a utilizar para cada una de las capas de la arquitectura. El marco de trabajo propuesto se valida mediante un caso de estudio**.**

Keywords: Metodologías ágiles; eXtreme Programming (XP); Test Unitarios; Arquitectura de Software; Desarrollo Guiado por Pruebas

1 Introducción

Realizar un desarrollo de software con éxito y de calidad depende de varios factores como, por ejemplo, las personas seleccionadas, las herramientas y tecnologías a utilizar, la arquitectura de software y la metodología que guiará el proceso. Su correcta elección es un factor crítico de éxito.

La arquitectura de software brinda una visión abstracta de alto nivel, permitiendo plantear la reutilización y la evolución del código. Por otro lado, las metodologías ágiles permiten generar productos de calidad, basándose en la adaptabilidad del proceso de desarrollo de software para aumentar sus posibilidades de éxito por la flexibilidad y eficacia sobre las metodologías tradicionales.

Este trabajo explora como adaptar la arquitectura de software con la práctica del desarrollo guiado por pruebas (Test Driven Development - TDD) dentro de metodologías ágiles de desarrollo de software. En particular, considera la arquitectura de software en capas y la metodología de desarrollo ágil eXtreme Programming [1].

La contribución del mismo es mostrar la factibilidad del enfoque, presentando un marco de trabajo que incluye la selección de tecnologías que permiten su implementación. El marco propuesto se valida mediante un caso de estudio.

El trabajo está estructurado de la siguiente manera: La Sección 2 presenta los conceptos relacionados, incluyendo eXtreme Programming, arquitectura de software, desarrollo dirigido por pruebas (Test Driven Development - TDD) y pruebas unitarias. La Sección 3 explica el marco de trabajo propuesto y las tecnologías seleccionadas. A continuación, la Sección 4 valida el marco de trabajo propuesto a través de un caso de estudio y muestra como aplica TDD en el desarrollo de una de las capas de la arquitectura. La Sección 5 presenta otros aportes científicos en esta línea de investigación y discute la contribución de este trabajo. Por último, la Sección 6 brinda conclusiones y explica los trabajos futuros.

2 Conceptos Utilizados

Las siguientes secciones presentan los conceptos básicos utilizados en este trabajo, incluyendo: eXtreme Programming (Sección 2.1), arquitectura de software (Sección 2.2), test driven development (Sección 2.3) y pruebas unitarias (Sección 2.4).

2.1 eXtreme Programming

eXtreme Programming (XP) es una metodología ágil, descrita por Kent Beck [2], que centra sus prioridades en las personas y no en los procesos, alentando a los desarrolladores a responder a requerimientos cambiantes de los usuarios, aún en fases tardías del ciclo de vida del desarrollo. Se basa principalmente en la comunicación e interacción permanente con el usuario y en la programación de a pares (técnica de programación por parejas donde uno de los programadores escribe el código y el otro lo prueba, y luego se intercambian los roles). De esta forma, desde el principio, el código se prueba en base a requerimientos funcionales.

El proceso consiste de tres etapas: 1) *Interacción con el Cliente* - el cliente está disponible durante todo el proyecto para interactuar con el equipo de trabajo. De esta manera, se elimina la fase inicial de recolección de requerimientos, y éstos se van incorporando ordenadamente a lo largo del desarrollo; 2) *Planificación del Proyecto* – se basa en un diálogo continuo entre las partes involucradas en el proyecto, siendo el equipo el que estima el esfuerzo requerido para la implementación de cada funcionalidad; 3) *Diseño y Desarrollo de Pruebas* – el desarrollo guiado por pruebas es un enfoque ágil, donde para cada funcionalidad que se desea implementar, primero se escriben las pruebas y luego el código necesario para que la prueba sea exitosa. Una vez que el código cumple el test exitosamente, se amplía y continúa. De este modo, se realiza una integración continua, evitando un proceso más complejo al finalizar el proyecto [1]. El desarrollo guiado por pruebas formalizado en XP se conoce como Test Driven Development (TDD) [3].

2.2 Arquitectura de Software

La arquitectura de software conforma la columna vertebral de cualquier sistema y constituye uno de sus principales atributos de calidad [4]. El documento de IEEE Std 1471-2000 [5] define: "La Arquitectura de Software es la organización fundamental de un sistema encarnada en sus componentes, las relaciones entre ellos y el ambiente y los principios que orientan su diseño y evolución".

En particular, una arquitectura comúnmente usada es la definida en capas. En el caso de una aplicación empresarial puede dividirse en tres capas lógicas bien definidas [6]: 1) la capa de presentación, 2) la capa de negocio y 3) la capa de persistencia. El principio para la separación en capas es que cada una esconde su lógica al resto y solo brinda puntos de acceso a dicha lógica.

En la capa de presentación los objetos trabajan directamente con las interfaces de de negocios, implementando el patrón arquitectónico Model-View-Controller [6]. En este, el modelo (Model) es modificable por las funciones de negocio, siendo estas solicitadas por el usuario, mediante el uso de un conjunto de vistas (View) que solicitan dichas funciones de negocio a través de un controlador (Controller), que es quien recibe las peticiones de las vistas y las procesa.

La capa de negocio está formada por servicios implementados por objetos de negocio. Estos delegan gran parte de su lógica en los modelos del dominio que se intercambian entre todas las capas.

Finalmente, la capa de persistencia facilita el acceso a los datos y su almacenamiento en una base de datos.

2.3 Test Driven Development (TDD)

TDD es una técnica de programación que consiste en guiar el diseño de una aplicación, por medio de pruebas unitarias. Esta, cambia el orden tradicionalmente establecido, de manera que en primero se definen las pruebas y a partir de estas se va desarrollando la funcionalidad, repitiendo el ciclo, de acuerdo a lo que se espera que haga el software, por medio de integraciones y refactorizaciones – **(**una refactorización consiste en realizar modificaciones en el código con el objetivo de mejorar su estructura interna, sin alterar su comportamiento externo [7]) - continúas del desarrollo en los casos en que las pruebas no cumplan con el requerimiento.

Con esta técnica las pruebas constituyen la documentación del software que se está desarrollando.

2.4 Pruebas Unitarias

En los últimos años, los test unitarios han ido tomando cada vez más fuerza en el proceso de desarrollo de software y se integraron de una forma altamente productiva [8]. El desarrollo y ejecución de pruebas es una actividad fundamental en los proyectos de desarrollo de software, los cuales permiten mantener código de calidad durante el ciclo de vida del proyecto. Fundamentalmente, las pruebas unitarias

representan una alternativa para encontrar y corregir la mayoría de los errores de codificación [8].

Las pruebas unitarias como primer paso en el proceso de desarrollo son la base de la filosofía TDD, ya que resulta la mejor manera de producir código rápidamente y de calidad, permitiendo conducir el diseño, mediante la codificación de las citadas pruebas, antes de codificar interfaces o implementaciones [9].

3 Arquitectura de Software y Entornos de Trabajo (Framework)

Aplicando los conceptos explicados anteriormente, en esta sección presentamos un marco de trabajo que integra distintas tecnologías y frameworks disponibles en el mercado para la implementación de una arquitectura en capas, dirigida por pruebas unitarias en XP.

Para la implementación de la capa de presentación se propone el framework JSF [10], basado en el patrón MVC, el cual permite desarrollar rápidamente aplicaciones dinámicas creando páginas (vistas) y manejadores de vista (ManagedBean) de manera sencilla, simplificando el diseño de interfaces de usuarios. Una ventaja de esta elección es su capacidad de extensión para definir nuevos componentes e incorporar librerías existentes, como PrimeFaces[11] entre otras.

Para la implementación de la capa de negocio se propone la utilización de Spring Framework [12]. Un entorno de trabajo de código abierto, utilizado para la simplificación en el desarrollo de Aplicaciones Java Empresariales (JEE). Spring provee de un contenedor de objetos quien se encarga de administrar el ciclo de vida de los mismos, implementa los objetos de dominio como POJOs (*Plain Old Java Object* - sigla creada por Martin Fowler, Rebecca Parsons y Josh MacKenzie [13]) y representa objetos que son parametrizables a través de sus propiedades o constructores por medio de archivos de configuración u anotaciones. El contenedor maneja dos conceptos muy importantes para administrar las instancias de los objetos (POJOs): la Inversión de Control (IoC: Inversion of Control) y la Inyección de Dependencias (DI: Dependency Injection). El principio de Inversión de Control consiste en que el control de la construcción de los objetos no recae directamente en el desarrollador, sino que es otra clase o conjunto de clases las que se encargan de construir los objetos que se necesitan.[14].

Finalmente, para la capa de persistencia diseñada con el patrón DAO (Data Access Object) [15] se propone la utilización del Framework de código abierto Hibernate [16] como ORM (Mapeo Objeto Relacional). A través de Hibernate se realiza el mapeo del modelo de objetos (POJOS) a la base de datos relacional, mediante archivos declarativos o anotaciones en los objetos POJOS que permiten establecer estas relaciones. Hibernate está diseñado para ser flexible en cuanto al esquema de tablas utilizado, para poder adaptarse a su uso sobre una base de datos ya existente.

El marco de trabajo propuesto se ilustra en la Figura 1.

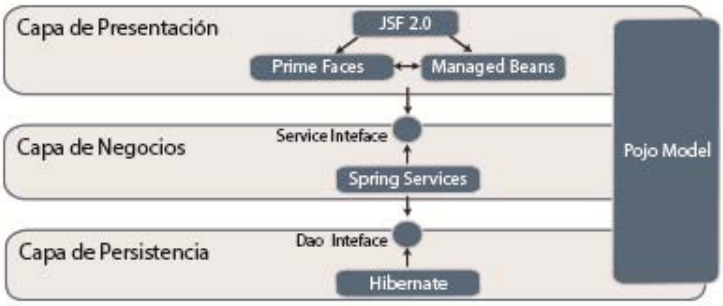

Figura 1. Arquitectura en Capas

4 Caso de Estudio

Una de las herramientas fundamentales para la realización de las pruebas de unidad más difundida en proyectos de software es conocida como JUnit [17]. Siendo una herramienta de prueba de código y el estándar para la técnica de TDD [9].

A continuación se presenta el caso de estudio mediante dos ejemplos, uno que se centra en depositar una suma de dinero en una cuenta bancaria y, el otro, que ejecuta una transferencia de un monto de dinero entre cuentas bancarias, guiados por TDD.

Para el primer ejemplo, la prueba de unidad permite probar la funcionalidad para el requerimiento "depositar dinero en la Cuenta". Una alternativa es la siguiente:

```
public void testDepositarEnCuenta() {
        // 1: Creamos un contexto de prueba
        Cuenta cuenta = new Cuenta();
        // 2: Ejecutamos el código bajo prueba.
        cuenta.depositar(100);
        // 3: Comprobamos el resultado.
        assertEquals(100, cuenta.getMonto(), 0.001);}
```

Prueba 1: Depositar Dinero en la Cuenta

Luego de la prueba de unidad se avanza en el diseño de la aplicación, por lo que se crea la clase "`Cuenta`" con su constructor y se implementa el método "`depositar`", el cual recibe un valor dado, para ello se modifica la estructura de la clase (el diseño) agregando el atributo "`monto`" que representa el valor.

El siguiente caso de prueba de unidad, permite comprobar la funcionalidad para el requerimiento "transferencia de dinero entre cuentas".

```
public void testTransferenciaEntreCuentas() {
   Cuenta cuenta1 = new Cuenta(100d);
   Cuenta cuenta2 = new Cuenta(100d);
   TransferenciaService transfService = new TransferenciaService();
   transfService.transferir(cuenta1, cuenta2, monto);}
```

Prueba 2: Transferencia de Dinero entre Cuentas

En este caso de prueba, se identifica un nuevo objeto de negocio, "TransferenciaService" que transfiere monto entre cuentas. Este implementa el método "transferir". La operación incluye a un nuevo objeto de negocio "CuentaService" que permite la actualización de las cuentas.

```
public void transferir(Cuenta cuenta1, Cuenta cuenta2, Double monto) {
  cuenta1.extraer(monto);
  cuenta2.depositar(monto);
  cuentaService.guardar(cuenta1);
  cuentaService.guardar(cuenta2);}
```

Implementación 1: Método Transferir de TransferenciaService

Siguiendo con la definición de la arquitectura propuesta, los objetos diseñados "CuentaService" y "TransferenciaService" se implementan en la capa de Negocio. De acuerdo al método "transferir" surge la necesidad de persistir los objetos "Cuenta", para ello se define el siguiente caso de prueba:

```
public void testGuardarCuenta() {
          CuentaService cuentaService = new CuentaService();
          Cuenta cuenta = new Cuenta();
          cuenta.setMonto(100);
          cuentaService.guardar(cuenta);}
```

Prueba 3: Guardar los Cambios de Cuenta

De la implementación del método `cuentaService.guardar(cuenta)`, citado, surge la necesidad de diseñar un nuevo objeto, llamado "CuentaDao", que se encarga de almacenar la "cuenta" en una fuente de datos. Este nuevo objeto se implementa en la capa de persistencia de la arquitectura.

```
public void guardar(Cuenta cuenta) {
        cuentaDao.guardar(cuenta);}
```

Implementación 2: Método Guardar de CuentaService

De los casos de prueba anteriores se realiza el diseño, de acuerdo al proceso de la técnica TDD, donde los requerimientos se traducen en pruebas. Es necesario, en ciertas oportunidades, que los objetos diseñados se comuniquen con otros para cumplir con su objetivo (función), para lo cual es importante simular la interacción entre ellos, sin llegar a construir el objeto real, de esta manera se logra que las mismas se realicen de forma independiente en cada una de las capas.

Las secciones a continuación explican, en detalle, la herramienta de pruebas para simular los objetos en la capa de negocio y las diferentes estrategias para aplicar pruebas unitarias en cada capa de la arquitectura.

4.1 Capa de Presentación

Esta capa contiene los manejadores ("ManagedBean"), que interactúan con otros objetos para colaborar con las acciones llevadas a cabo en la vista. La comunicación que existe con la capa subyacente es a través de la implementación de los objetos

Mocks (falsos) de la capa de Negocio utilizando la herramienta JMock. La Figura 2 muestra la implementación de las pruebas de unidad con JMock en la capa de presentación (`ManagedBean`).

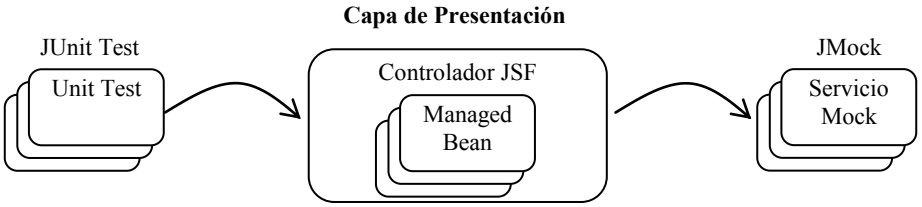

Figura 2. Implementación de las Pruebas en la Capa de Presentación

4.2 Capa de Negocio

A fin de realizar los test en la capa de negocios, se plantea la técnica denominada Mock Test. La utilización de esta técnica permite que las pruebas sean unitarias en lugar de que sean pruebas de integración (utiliza todos los componentes reales de los que depende). El desarrollo de pruebas unitarias en esta capa propone aislarla de los objetos de acceso a datos (DAO) y simular la implementación de los mismos con objetos Mocks (falsos).

Existen múltiples herramientas que permiten la creación de Objetos Mocks, como por ejemplo: MockObjects [18], jMock:[19], Mockito [20], EasyMock [21]. El diseño guiado en esta capa se realiza con la herramienta jMock, la guía de pasos para utilizar esta herramienta incluye: 1) declarar un contexto para la prueba, 2) crear los mocks dentro del contexto, 3) crear las expectativas (el comportamiento que se espera de los mocks), 4) ejecutar el código bajo prueba, y 5) comprobar si se han cumplido todas las expectativas.

A continuación, se describe el caso de prueba utilizando la herramienta JMock, siguiendo los pasos arriba descriptos:

```
//0. Crear un contexto.
Mockery context = new Mockery();

//1. Crear los mocks dentro del contexto a partir de las interfaces.
CuentaService cuentaService = context.mock(CuentaService.class);
TransferenciaService transfService = context.mock(TransferService.class);
public void testTransferirMontoEntreCuentas() {
 final Cuenta cuenta1 = new Cuenta(100);
 final Cuenta cuenta2 = new Cuenta(100);

//2. Definir las llamadas que esperan los mocks y los valores devueltos.
 context.checking(new Expectations() {
     {       oneOf(cuentaService).guardar(cuenta1);
             oneOf(cuentaService).guardar(cuenta2);
     }
  });

//3. Crear el objeto de la clase e invocar el método bajo prueba.
     transfService.transferir(cuenta1,cuenta2,50);
```

```
//4. Verificar que el comportamiento indicado en context se haya cumplido
 context.assertIsSatisfied();}
```

Prueba 4: Descripción del Caso de Prueba en JMock

Seguidamente se asegura la persistencia de la transferencia que se realiza utilizando objetos Mocks en el objeto `CuentaService`.

```
public void testGuardarCuenta() {
  final Cuenta cuenta = new Cuenta();
  final CuentaDao dao = context.mock(CuentaDao.class);
  context.checking(new Expectations() {{oneOf(dao).guardar(cuenta);} });
  cuentaService.setDao(dao);
  mockery.assertIsSatisfied();}
```

Prueba 5: Guardar en Cuenta utilizando JMock

La Figura 3 muestra como se implementan las pruebas de unidad con JMock en la Capa de Negocios.

Figura 3. Implementación de Pruebas de Unidad en la Capa de Negocios

4.3 Capa de Persistencia

La implementación de la capa lógica de persistencia, DAO (Data Access Object), se guía por pruebas, interactuando con un elemento externo (Base de datos). La utilización de DbUnit, como una extensión de JUnit, permite interactuar con un conjunto de datos de prueba y, también, dejar la base de datos en un estado conocido antes y después de cada ciclo de prueba, esto con el fin de prevenir que datos corruptos queden en la base de datos ocasionando problemas a los ciclos siguientes. Básicamente DbUnit usa archivos XML para cargar datos en la base de datos (dataset). [9]. La Figura 4 ilustra cómo se implementan las pruebas de unidad con DBUnit en la capa de persistencia (Dao).

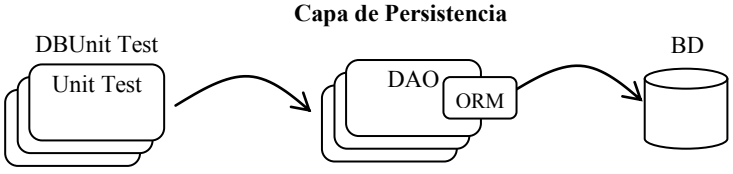

Figura 4. Implementación de Pruebas de Unidad en la Capa de Persistencia

5 Trabajos Relacionados

Este trabajo está basado en tareas de investigación que analizaron los aportes científicos que se encuentran en esta misma línea. En particular, se investigaron trabajos relacionados con la integración de las metodologías ágiles con arquitecturas de software que pudieran brindar un marco de calidad con un enfoque riguroso que resultara en mejoras de los procesos de diseño y de desarrollo de aplicaciones. Algunos de los trabajos más importantes se discuten a continuación.

Urquiza Yllescas, et al [22] plantea actividades de desarrollo que integran tácticas y estrategias de diseño de la arquitectura de software en metodologías agiles, desde una perspectiva general. Breivold y sus colegas [23] realizan una meta-investigación sobre publicaciones científicas que relacionan el desarrollo ágil y la arquitectura de software, concluyendo en la falta de evidencia científica para muchas afirmaciones sobre agilidad y arquitectura, resaltando la necesidad de estudios empíricos para demostrar ventajas y desventajas de aplicar un método ágil. En [24], se analizan dos casos de estudios adoptando patrones de diseños arquitectónicos en el desarrollo ágil de software para aplicaciones móviles. Resaltando la utilidad de una arquitectura basada en patrones y soluciones probadas. Finalmente, [25] presenta un arquitectura dirigida por modelos (MDA) y el proceso de XP, analizando los argumentos a favor y en contra, proponiendo una nueva arquitectura que supere las limitaciones identificadas.

Comparando el estado del arte con los resultados de este trabajo, nuestra tarea consistió en desarrollar, con el enfoque de TDD, una nueva perspectiva de aplicación ágil relacionándola con una determinada arquitectura e identificando un conjunto de tecnologías para su implementación, aportando, a nuestro criterio, evidencia práctica para su aplicación. De este modo, aportamos contribución empírica para demostrar las ventajas de combinar metodologías agiles con arquitecturas de software, como lo requerían las conclusiones de trabajos relacionados [22].

6 Conclusiones y Trabajos Futuros

Este trabajo presentó un marco de trabajo para el desarrollo de aplicaciones basado en una arquitectura en capas, aplicando la técnica de TDD para el diseño y desarrollo de cada una de las capas. La ventaja radica en producir capas de software altamente cohesivas, donde un nuevo requerimiento no impacta en el comportamiento entre cada una de ellas. La técnica aplicada sobre la arquitectura implica un cambio de mentalidad en el desarrollo de software, lo que permite implementar el código necesario para resolver cada caso de prueba concreto. Este enfoque permite reducir los típicos bloques de código que usualmente se agregan “por las dudas”, característica que habitualmente ocurre con metodologías de desarrollo tradicional.

Trabajos a futuro incluyen extender el marco de trabajo aplicando otras técnicas de desarrollo ágil como BDD (Behaviour Driven Development) y ATDD (Acceptance Test Driven Development) que permitirán diferentes posibilidades de integración dentro de nuestro ciclo de pruebas.

Referencias

[1] Mendes Calo, K, Estevez, E. and Fillottrani, P. "*Un Framework para Evaluación de Metodologías Agiles*" http://sedici.unlp.edu.ar/handle/10915/21086.

[2] Beck, K. "*Extreme Programming Explained. Embrace Change",* (1999).

[3] Wells, D., "*Extreme Programming Unit Tests*", disponible en: http://www.extreme programming.org/rules/unittests.html (accedido 01/06/2013).

[4] Clements, P., et al, "*Software Architecture in Practice*", Pearson Education, (2003).

[5] IEEE Standards Association, "*1471-2000 - IEEE Recommended Practice for Architectural Description for Software-Intensive Systems*", available at: http://standards.ieee.org/findstds/standard/1471-2000.html.

[6] Fowler, M. "*Patterns of Enterprise Application Architecture*", Addison-Wesley, (2002).

[7] Fowler, M., "*Refactoring: Improving the Design of Existing Code*", Addison-Wesley Longman, Inc., (1999).

[8] Johnson, R. and Hoeller, J., "*Expert One-on-One J2EE Development without EJB*", Wiley Publishing, (2004).

[9] Sam-Bodden, B., "*Beginning Pojos – From Novice to Professional*", APress, (2006).

[10] Oracle, "*Java Server Faces*", disponible en: http://java.sun.com/j2ee/javaserverfaces/ (accedido 02/07/2013).

[11] Prime Faces, "*PrimeFaces Ultimate JSF Component Suite*", disponible en: http://primefaces.org (accedido 13/03/2013).

[12] GoPIvotal, Inc., "*Spring Framework*", http://www.springframework.org/ (accedido 21/05/2013).

[13] Fowler, M., "*Plain Old Java Object (POJO)*", disponible en: http://www.martin fowler.com/bliki/POJO.html (accedido 07/06/2013).

[14] Johnson, R., et al, "*Professional Java Development with the Spring Framework*", Wiley Publishing Inc, (2005).

[15] Oracle, "*Data Access Object (DAO)*", disponible en: http://java.sun.com/blueprints/ corej2eepatterns/Patterns/DataAccessObject.html (accedido 21/03/2013).

[16] JBoss Community, "*Hibernate*", http://www.hibernate.org/ (accedido 22/03/2013).

[17] GitHub, "*JUnit*", www.junit.org/ (accedido 06/05/2013).

[18] Mock Blog, "*Mock Objects*", http://www.mockobjects.com (accedido 02/06/2013).

[19] GitHub, "*JMock*", http://www.jmock.org (accedido 16/06/2013).

[20] "*Mockito*", https://code.google.com/p/mockito/ (accedido 02/05/2013).

[21] Freese, T. and Tremblay, H. "*Easy Mock*", http://www.easymock.org (accedido 28/05/2013).

[22] Urquiza Yllescas, J.F., et al, "*Las Metodologías Agiles y las Arquitecturas de Software*". Coloquio Nacional de Investigación en Ingeniería de Software y Vinculación Academia-Industria 2010, 29-Setiembre al 1-Octubre 2010, León, Guanajuato, Mexico.

[23] Breivold,H.P., Sundmark, D., Wallin, P. and Larsson, S., "*What Does Research Say about Agile and Architecture?"* , en Proceedings of the 2010 Fifth International Conference on Software Engineering Advances (ICSEA), USA, (2010).

[24] Ihme, T. and Abrahamsson, P., "*The Use of Architectural Patterns in the Agile Software Development of Mobile Applications*", en Proceedings of International Conference on Agility. Helsinki, Finland, (2005).

[25] Guha, P., et al, "*Incorporating Agile with MDA Case Study: Online Polling System*", International Journal of Software Engineering & Applications (IJSEA), Vol.2, No.4, pp. 83-96, (Oct. 2011).