Universidad de las Ciencias Informáticas

Facultad 6

Extensión de la herramienta Visual Paradigm for UML para la evaluación y corrección de Diagramas de Casos de Uso

TRABAJO PARA OPTAR POR EL TÍTULO DE INGENIERO EN CIENCIAS INFORMÁTICAS

Autores: Dayana Mendoza Peña

Lionel Rodolfo Baquero Hernández

Tutores: MSc. Omar Mar Cornelio

Ing. Osviel Rodriguez Valdés

La Habana, junio de 2016

“Año 58 de la Revolución”

Declaración de autoría

Declaramos ser los autores de la presente tesis y reconocemos a la Universidad de las Ciencias Informáticas los derechos patrimoniales de la misma, con carácter exclusivo.

Para que así conste firmamos la presente a los __ días del mes de _____ del año 2016.

_____	_____
Autor	Autor
Dayana Mendoza Peña	Lionel Rodolfo Baquero Hernández
_____	_____
Tutor	Tutor
MSc. Omar Mar Cornelio	Ing. Osviel Rodriguez Valdés

Datos de contacto

Autor: Dayana Mendoza Peña

Facultad 6. Universidad de las Ciencias Informáticas. La Habana, Cuba.

Email: dmendoza@estudiantes.uci.cu

Autor: Lionel Rodolfo Baquero Hernández

Facultad 6. Universidad de las Ciencias Informáticas. La Habana, Cuba.

Email: lrbaquero@estudiantes.uci.cu

Tutor: MSc. Omar Mar Cornelio

Departamento de Programación y Sistemas Digitales. Facultad 6. Universidad de las Ciencias Informáticas. La Habana, Cuba.

Email: omarmar@uci.cu

Tutor: Ing. Osviel Rodriguez Valdés

Departamento de Programación y Sistemas Digitales. Facultad 6. Universidad de las Ciencias Informáticas. La Habana, Cuba.

Email: osviel@uci.cu

Dedicatoria

de Dayana

A mi mamá, por su amor infinito, su esfuerzo y su apoyo incondicional.

A mi papá, por dejarme cometer mis propios errores y estar siempre para levantarme.

A toda mi familia, mi hermana, mi sobrina, mis abuelas, mis tías, mis primos, que siempre se preocupan por mis estudios y siempre los tengo cerca pesar de las distancias.

A mi novio, por su amor, su paciencia y sus abrazos.

de Lionel

A mi mamá por su incondicionalidad, su dedicación y su amor.

A la memoria de mi abuela.

A mis sobrinitos por ser mi gran inspiración.

A mis hermanas por quererme y defenderme tanto.

A mi padrastro por sus grandes esfuerzos.

A mi familia por el apoyo en todo momento, por entender y aceptar mis diferencias.

A mis amigos por estar siempre en los momentos difíciles.

Resumen

En la Universidad de las Ciencias Informáticas se ha estandarizado el uso del Visual Paradigm for UML para el modelado de los procesos de desarrollo de software que en ella se llevan a cabo. Entre los diagramas que la herramienta permite modelar se encuentran los Diagramas de Casos de Uso. En el modelado de estos diagramas se pueden cometer errores de concepto que generan pérdidas de tiempo, lo que trae consigo retrasos en los cronogramas de proyecto. Además, los errores pueden traducirse, en su caso más crítico, en no conformidades en las pruebas al sistema. La herramienta no permite hacer una evaluación y corrección de los mismos, para servir como referente al usuario que está diseñando el software. La presente investigación se propone como objetivo desarrollar una extensión de la herramienta Visual Paradigm for UML para la evaluación y corrección de Diagramas de Casos de Uso. Para su desarrollo su utilizó la metodología OpenUP, el Lenguaje de Modelado Unificado en su versión 2.5, Visual Paradigm for UML 8.0 como herramienta de modelado, el lenguaje de programación Java 8.0 y el Entorno de Desarrollo Integrado NetBeans 8.0. Además, se utilizó la biblioteca OpenAPI para poder extender las funcionalidades del Visual Paradigm for UML. A la extensión se le aplicaron pruebas a nivel de unidad, sistema, integración y aceptación, para las que se obtuvieron resultados satisfactorios.

Palabras clave: Corrección, Diagrama de Casos de Uso, evaluación, Visual Paradigm for UML.

Abstract

At the University of Informatics Sciences it has standardized the use of Visual Paradigm for UML for modeling the processes of software development. Among the types of diagrams that this tool allows modeling are the Use Case Diagram. During the modeling of these types of diagrams can be committed conception mistakes, that can cause loss of time, which may bring delays in project schedules. Furthermore, these errors can result in the most critical case in nonconformity in the system testing phase. This tool does not allow an evaluation and correction of diagrams, to serve as a reference the user who is designing the software in question. This research aims to develop an extension of Visual Paradigm for UML tool for evaluation and correction of Use Case Diagrams. To carry out the proposed solution was defined use the OpenUP methodology, Unified Modeling Language in its version 2.5, Visual Paradigm as for UML 8.0 modeling tool, the Java 8.0 programming language and NetBeans Integrated Development Environment 8.0. In addition, the OpenAPI library is used to extend the functionality of Visual Paradigm for UML. To the extension is applied to test unit level, system, integration and acceptance, for which were obtained satisfactory results.

Keywords: *Correction, Use Case Diagram, evaluation, Visual Paradigm for UML.*

Índice general

Índice de figuras

Índice de tablas

Introducción

La Ingeniería de Software (ISW) es una disciplina fundamental en el proceso de desarrollo de software. Sirve como base para la construcción de un producto informático, al permitir analizar y diseñar las aplicaciones para que puedan ser implementadas. Permite trazar las directrices para la implementación y pruebas del sistema, así como identificar las condiciones ideales para que un software pueda ser desplegado.

En la ISW se emplea el Lenguaje Unificado de Modelado (UML, Unified Modeling Language) para el diseño de un software. Se encuentra en la versión 2.5 y permite modelar diagramas de Actividad, Clases, Comunicación, Componentes, Estructura Compuesta, Despliegue, Interacción, Objetos, Paquetes, Perfil, Máquina de Estado, Tiempo y Casos de Uso (OMG, 2015).

Dentro de las actividades que comprende el proceso de Análisis de la ISW se encuentra la Ingeniería de Requisitos, que es el uso sistemático de procedimientos, tecnologías y herramientas para obtener la documentación y especificación del comportamiento externo de un sistema. Una vez recopiladas estas necesidades del usuario o requisitos, se crean un conjunto de escenarios. Los escenarios son llamados casos de uso y facilitan la descripción de cómo el sistema se usará (Pressman, 2010).

Un Diagrama de Casos de Uso (DCU) muestra las relaciones existentes entre actores y casos de uso dentro de una aplicación. El actor es una abstracción de las entidades externas, subsistemas o clases que interactúan directamente con el sistema. Un actor participa en un caso de uso o conjunto coherente de casos de uso para llevar a cabo un propósito global. Las diferentes interacciones de los actores con un sistema se cuantifican en casos de uso. Un caso de uso es una especificación de las secuencias de acciones (Rumbaugh et al., 2007).

En el modelado de DCU se pueden cometer errores conceptuales, que pueden ser:

- Ausencia de algún elemento del diagrama.
- Elemento sin relacionar con otro en el diagrama.
- Elemento incorrectamente relacionado con otro en el diagrama.
- Existencia de relaciones invertidas en el diagrama.
- Elementos que comparten relaciones múltiples.

Para el modelado de los diagramas se utilizan las herramientas CASE (*Computer Aided Software Engineering*) que permiten el diseño de múltiples artefactos. Las herramientas CASE existentes no son capaces de evaluar y corregir DCU para evitar que se cometan errores conceptuales; a pesar de que han tomado gran relevancia en la planeación y ejecución de proyectos que involucren sistemas de información, ya que inducen a sus usuarios a la correcta utilización de metodologías que le ayudan

a diseñar con facilidad los productos de software (Quintero et al., 2012).

En la Universidad de las Ciencias Informáticas (UCI) se ha estandarizado el uso del Visual Paradigm for UML como herramienta CASE para el modelado de los procesos de desarrollo de software que en ella se llevan a cabo, por la gran cantidad de ventajas que posee, quienes están en concordancia con los intereses y políticas establecidas en la institución. Entre sus principales características se encuentran que es multiplataforma, facilita la colaboración en equipo y brinda apoyo al ciclo de vida completo del desarrollo de software (Rosales et al., 2013). Entre los diagramas que la herramienta Visual Paradigm for UML permite modelar se encuentran los DCU.

En el modelado de un DCU con la herramienta Visual Paradigm for UML también se pueden cometer los errores frecuentes identificados, que pueden desencadenar:

- Pérdidas de tiempo y traer consigo retrasos en los cronogramas de proyecto.
- No conformidades en las pruebas al sistema.
- Conflictos internos en el equipo de desarrollo.
- Litigios en la aceptación del producto por parte del cliente, y de esta forma generar pérdida de credibilidad e incremento del presupuesto para la empresa desarrolladora.

La herramienta Visual Paradigm for UML no permite hacer una evaluación y corrección de los DCU, para servir al usuario que diseña el software a mejorar el modelado de su diagrama, por lo que se plantea como **problema de la investigación:** ¿Cómo evaluar y corregir los Diagramas de Caso de Uso que se modelan con la herramienta Visual Paradigm for UML?

La investigación tiene como **objeto de estudio:** la evaluación y corrección de Diagramas de Casos de Uso, enmarcado en el **campo de acción:** la evaluación y corrección de Diagramas de Casos de Uso con la herramienta Visual Paradigm for UML.

Para ello se plantea como **objetivo general:** desarrollar una extensión de la herramienta Visual Paradigm for UML para la evaluación y corrección de Diagramas de Casos de Uso.

Para guiar el proceso investigativo se plantean las siguientes **preguntas de investigación**:

- ¿Cuáles son los fundamentos teóricos metodológicos relacionados con la evaluación y corrección de Diagramas de Casos de Uso?
- ¿Qué técnicas, tecnologías y herramientas de desarrollo se pueden utilizar en la implementación de la propuesta de solución?
- ¿Qué funcionalidades debe tener la propuesta de solución?
- ¿Qué tipos de pruebas se pueden utilizar en la validación de la propuesta de solución y cómo se deben aplicar?

Para dar cumplimiento al objetivo trazado se plantean las siguientes **tareas de la investigación**:

1. Definición de los elementos conceptuales referentes a la evaluación y corrección de Diagramas de Casos de Uso.
2. Análisis y selección de la metodología, herramientas y tecnologías para el desarrollo de la extensión.
3. Identificación y descripción de los requisitos funcionales y no funcionales.
4. Descripción de la arquitectura base para el desarrollo de la extensión.
5. Definición de los componentes de diseño que se ajusten a la arquitectura seleccionada.
6. Implementación de los componentes de diseño para el desarrollo de la extensión.
7. Aplicación de las pruebas funcionales a la extensión para dar cumplimiento a los requisitos definidos.

Para dar cumplimiento al objetivo general y realizar las tareas de investigación, se han combinado **métodos teóricos y empíricos** de la investigación científica.

Del nivel teórico

- Analítico-Sintético: El método fue empleado para analizar el estado del arte de los elementos asociados a los Diagramas de Casos de Uso, así como su evaluación y corrección, y de esta forma obtener conocimiento para luego sintetizarlo.
- Histórico-lógico: El método consiste en realizar una revisión exhaustiva del desarrollo evolutivo del objeto de investigación a lo largo del tiempo con el objetivo de definir las limitaciones actuales de su conocimiento. El método permitió reconocer los avances teórico-prácticos y problemas que actualmente existen en el área de investigación tratada.

Del nivel empírico

- Entrevista: Se aplicó a profesores del departamento de Ingeniería de Software y especialistas asociados al desarrollo de software con el objetivo de identificar los principales problemas y errores conceptuales que se pueden cometer en el modelado de Diagramas de Casos de Uso (ver Anexo 1).

Como resultados se espera la obtención de una extensión de la herramienta Visual Paradigm for UML para la evaluación y corrección de Diagramas de Casos de Uso, que contribuya a la eliminación de los errores conceptuales frecuentes en el modelado de estos tipos de diagramas.

Capítulo 1. Fundamentos teóricos sobre la evaluación y corrección de los Diagramas de Casos de Uso

En el presente capítulo se describen los conceptos básicos que se relacionan con el objeto de estudio de la investigación. Además, se detallan los errores que se pueden cometer en el modelado de Diagramas de Casos de Uso. Se fundamenta la selección de la metodología, tecnologías y herramientas a utilizar en el desarrollo de la solución propuesta. También se hace un análisis de las soluciones existentes para valorar posibles respuestas al problema planteado.

1.1. Lenguaje de Modelado Unificado

El Lenguaje Unificado de Modelado (UML, *Unified Modeling Language*) es el resultado de un esfuerzo por estandarizar el modelado de software Orientado a Objetos. UML fue adoptado en 1997 por OMG (*Object Management Group*) como una de sus especificaciones, y desde entonces se ha convertido en un estándar para visualizar, especificar y documentar los modelos que se crean durante la aplicación de un proceso de software. UML ha ejercido un gran impacto en la comunidad del software, tanto a nivel de desarrollo como de investigación (García et al., 2004).

UML es empleado para la documentación de proyectos, actualmente se encuentra en la versión 2.5. No dispone de una metodología de desarrollo, por lo que se apoya en metodologías de terceros para el desarrollo de proyectos. UML 2.5 modela diagramas de Actividad, Clases, Comunicación, Componentes, Estructura Compuesta, Despliegue, Interacción, Objetos, Paquetes, Perfil, Máquina de Estado, Tiempo y Casos de Uso (OMG, 2015).

1.1.1. Diagramas de Casos de Uso

Un Diagrama de Casos de Uso (DCU) muestra las relaciones existentes entre actores y casos de uso dentro de un sistema. El actor es una abstracción de las entidades externas a un sistema, subsistemas o clases que interactúan directamente con la aplicación. Un actor participa en un caso de uso o conjunto coherente de casos de uso para llevar a cabo un propósito global. Es una idealización con un propósito y significado concretos, que puede no corresponderse con objetos físicos. Un objeto físico puede combinar propósitos dispares y por tanto ser modelado por varios actores y viceversa, diferentes objetos físicos podrían incluir el mismo propósito, y por tanto ser modelados como el mismo actor (Rumbaugh et al., 2007).

Las diferentes interacciones de los actores con un sistema se cuantifican en casos de uso. Un caso de uso es una especificación de las secuencias de acciones, incluye secuencias variantes y secuencias de error, que pueden ser efectuadas por un sistema, subsistema o clase por interacción

con actores externos. Describe una secuencia completa iniciada por un actor. Entre los elementos que componen un DCU se encuentran las relaciones, que son una conexión semántica materializada (Rumbaugh et al., 2007). Entre las clases de relaciones presentes en un DCU se cuentan la Asociación o Comunicación, Inclusión, Extensión y Generalización o Especialización.

1.1.1.1 Errores en el modelado de Diagramas de Casos de Uso

A partir de las definiciones estudiadas sobre los elementos asociados a los DCU modelados con UML y de entrevistas realizadas a expertos (especialistas y profesores) del área de la Ingeniería de Software (ver Anexo 1), se pudieron identificar los posibles errores que pueden ser cometidos. Luego del estudio riguroso de los resultados de la investigación realizada, se concluyó que se pueden asumir como errores en el modelado de DCU:

Ausencia de algún elemento del diagrama

En un DCU deberán estar presentes al menos una instancia de cada uno de sus elementos (actor, caso de uso y relación), puesto que con la ausencia de uno de estos el diagrama pierde sentido y pertinencia. En la Figura 1 se muestra un DCU en el que solo se ha modelado un actor, con ausencia de caso de uso.

Figura 1. Diagrama de Casos de Uso modelado con Visual Paradigm for UML 8.0 con ausencia de caso de uso.

Elemento sin relacionar con otro en el diagrama

No deben existir elementos del diagrama que no se relacionen con ningún otro, en este caso se ha cometido un error en el modelado del DCU. Este error se debe evitar porque representaría un actor que no interactúa con el sistema o un caso de uso que no se ejecutará nunca (ver Figura 2).

Figura 2. Diagrama de Casos de Uso modelado con Visual Paradigm for UML 8.0 con elementos sin relacionar.

Elemento incorrectamente relacionado con otro en el diagrama

Que un actor o caso de uso esté incorrectamente relacionado con otro elemento, es la utilización incorrecta de las posibles relaciones entre estos elementos (ver Figura 3). Los actores solo se relacionarán con otros actores a través de la relación de tipo Generalización o Especialización. Un actor se relacionará con un caso de uso mediante una Asociación o Comunicación. Entre casos de uso se utilizarán únicamente las relaciones de Inclusión y Extensión.

Figura 3. Diagrama de Casos de Uso modelado con Visual Paradigm for UML 8.0 con elementos incorrectamente relacionados.

Existencia de relaciones invertidas en el diagrama

Si las relaciones han sido correctamente utilizadas, pero la dirección de estas no es la definida para relacionar los elementos, se puede afirmar que dicha relación está invertida. Este error puede provocar la no ejecución de casos de uso y en un caso crítico la no ejecución del sistema (ver Figura 4).

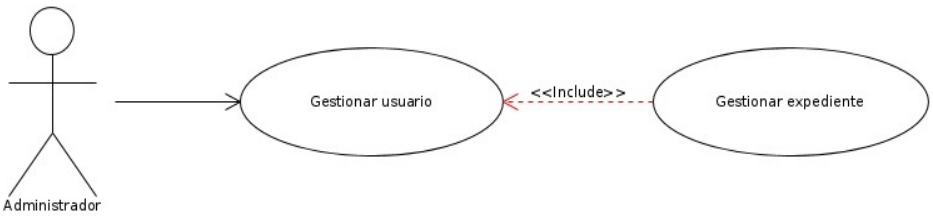


Figura 4. Diagrama de Casos de Uso modelado con Visual Paradigm for UML 8.0 con relación invertida.

Elementos que comparten relaciones múltiples

Entre dos elementos del diagrama, solo podrá existir una relación, para evitar de esta forma las relaciones múltiples (ver Figura 5). Las relaciones múltiples pueden generar ciclos infinitos de ejecución.

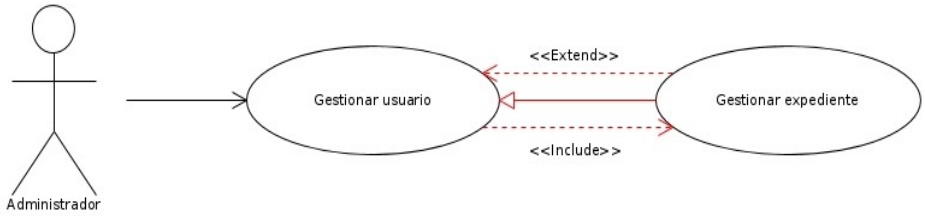


Figura 5. Diagrama de Casos de Uso modelado con Visual Paradigm for UML 8.0 con relaciones múltiples.

1.2. Evaluación y corrección de Diagramas de Casos de Uso

La evaluación de diagramas es un proceso diseñado específicamente para validar el modelado de estos. Debe estar encaminada a determinar si los elementos del diagrama están correctamente representados y relacionados entre sí. Este proceso de evaluación debe estar estructurado de forma tal que sea capaz de detectar los errores cometidos en el modelado del diagrama. Además, de ser posible, se deben brindar al usuario sugerencias para mejorar el diseño del diagrama.

La corrección de un Diagrama de Casos de Uso debe estar centrada en dar opciones para el mejoramiento del diseño. Este proceso se basa en los errores detectados durante el proceso de evaluación. Para cada error detectado se debe ofrecer al usuario la información suficiente para que este pueda aprender en la práctica cotidiana. También se deben brindar todas las alternativas posibles para la solución del problema detectado.

1.3. Análisis de soluciones existentes

El proceso de búsqueda de soluciones existentes se encaminó en función de encontrar las herramientas CASE más utilizadas en el mundo que modelan DCU. Se encontró que existen herramientas que modelan DCU y se estudiaron para conocer si eran capaces de evaluar y corregir estos diagramas. Los resultados de este estudio se pueden observar en la Tabla 1. Las características de estas herramientas fueron obtenidas de los estudios realizados por (López and Santa, 2012) y (Quintero, 2012).

Tabla 1. Análisis de las herramientas CASE más utilizadas.

Aplicación	Modelado de DCU	Evaluación de DCU	Corrección de DCU	Multiplataforma
ArgoUML	Si	No	No	Si
Rational Rose	Si	No	No	Si
WithClass	Si	No	No	No
Together	Si	No	No	Si
Poseidon for UML	Si	No	No	Si
StarUML	Si	No	No	Si
Enterprise Architect	Si	No	No	No
Visual Paradigm for UML	Si	No	No	Si

Luego de este análisis se puede evidenciar que las herramientas CASE más usadas no dan solución al problema de la presente investigación, ya que no son capaces de evaluar y corregir los DCU. Se concluye que se deberá implementar una extensión de la herramienta Visual Paradigm for UML, que ha sido estandarizada en la Universidad de las Ciencias Informáticas como herramienta CASE.

1.4. Herramienta a extender

La implementación de extensiones para aplicaciones constituye un mecanismo para la incorporación de funcionalidades que la aplicación no provee a los usuarios. Por lo general se asocian las extensiones con *plugin*, que son fragmentos de software que interactúan con el núcleo de la aplicación para proporcionar algunas funcionalidades que en la mayoría de los casos son muy específicos.

Visual Paradigm es una de las herramientas CASE más utilizadas para el modelado de software (Quintero et al., 2012). Esta herramienta cuenta con los medios necesarios para extender sus funcionalidades, pues da soporte a las extensiones de aplicación. Provee de forma libre una interfaz de programación, que permite a los desarrolladores implementar y reutilizar clases e interfaces y desarrollar funciones agregadas que son útiles para el desarrollo de software (Trujillo et al., 2013). Para poder desarrollar la extensión es necesario conocer cómo interactuar con la biblioteca OpenAPI proporcionada por la herramienta CASE y cómo lograr la integración de la extensión creada.

1.5. Metodología de desarrollo de software

A partir de un estudio realizado sobre las metodologías ágiles más usadas, se decide que el

desarrollo de la extensión de la herramienta Visual Paradigm for UML para la evaluación y corrección de Diagramas de Casos de Uso se apoya en un enfoque ágil; se centra en los individuos y tiene un enfoque común. Se define como metodología de desarrollo OpenUP, que es una forma de desarrollo ágil y ligera. Esta metodología permite colaborar para sincronizar intereses y compartir conocimiento. Permite equilibrar las prioridades para maximizar el beneficio obtenido por los interesados en el proyecto, centrarse en la arquitectura de forma temprana para minimizar el riesgo y organizar el desarrollo. Además de un desarrollo evolutivo para obtener retroalimentación y mejoramiento continuo (Saavedra et al., 2013).

Todo proyecto basado en OpenUP consta de cuatro fases: inicio, elaboración, construcción y transición. Cada una de estas fases se divide a su vez en iteraciones, que tienen como ventaja permitir a los integrantes del equipo de desarrollo aportar con micro-incrementos, que pueden ser el resultado del trabajo de unas pocas horas o unos pocos días. Además, este ciclo de vida provee a los clientes de una visión del proyecto, transparencia y los medios para que controlen la financiación, el riesgo, el ámbito y el valor de retorno esperado (Balduino, 2007).

La selección de la metodología de desarrollo OpenUP, estuvo fundamentada en que es muy apropiada para proyectos pequeños y de bajos recursos, como es el caso del presente equipo de desarrollo. La selección debe ayudar a disminuir las probabilidades de fracaso e incrementar las probabilidades de éxito, mediante la detección de errores tempranos a partir de su ciclo iterativo, lo que supone una ventaja importante en el desarrollo de la propuesta de solución.

1.6. Herramientas y tecnologías utilizadas

Se seleccionaron las herramientas y las tecnologías que más se ajustaban a las necesidades reales del desarrollo de la extensión. Se tuvieron en cuenta además las características del equipo de desarrollo, la complejidad de la aplicación y la metodología seleccionada, para la definición de estas herramientas y tecnologías.

1.6.1. Lenguaje de modelado

Como lenguaje de modelado se seleccionó el Lenguaje Unificado de Modelado (UML- *Unified Modeling Language*) en su versión 2.5, que es el lenguaje de modelado de sistemas de software más conocido y utilizado. Este lenguaje en la actualidad está respaldado por el Grupo Manejador de Objetos (OMG - *Object Management Group*). Es una consolidación de muchas de las notaciones y conceptos más usados en el desarrollo de software Orientado a Objetos. Ofrece un estándar para describir un "plano" del sistema (modelo), que incluye aspectos conceptuales tales como procesos de negocios, funciones del sistema, aspectos concretos como expresiones de lenguajes de

programación, componentes de software reutilizables y esquemas de bases de datos. Además, permite especificar y visualizar un sistema.

Se puede utilizar para definir un sistema de software, detallar los artefactos, especificar su documentación y construcción. Se puede aplicar en una gran variedad de formas para dar soporte a una metodología de desarrollo de software y cuenta con varios tipos de diagramas para llegar a representar los diferentes puntos de vistas de un sistema (Trujillo et al., 2013).

1.6.2. Herramienta CASE

Para el modelado con UML se utilizará Visual Paradigm for UML 8.0 por ser una herramienta UML profesional que soporta el ciclo de vida completo del desarrollo de software: análisis y diseño orientados a objetos, construcción, pruebas y despliegue. Permite modelar todos los tipos de diagramas de clases, generar código desde diagramas y generar documentación.

La herramienta agiliza la construcción de aplicaciones con calidad y a un menor coste. Posibilita la generación de bases de datos, transformación de diagramas de Entidad-Relación en tablas de base de datos, así como obtener ingeniería inversa de bases de datos (Rondón et al., 2011).

1.6.3. Lenguaje de programación

Entre los lenguajes de Programación Orientada a Objetos (POO) que existen en la actualidad se encuentran C, C++, C#, PHP y Java. Este ultimo, desarrollado por Sun Microsystems, ha evolucionado rápidamente en el ámbito de las aplicaciones a gran escala. Es un lenguaje de programación de alto nivel, uno de sus pilares es la posibilidad de ejecutar un mismo programa en diversos sistemas operativos (independiente de la plataforma). Su diseño se caracteriza por ser robusto, seguro, portable, independiente a la arquitectura, dinámico e interpretado (Deitel, 2004).

En la implementación de la extensión se utiliza Java en su versión 8.0 por ser el lenguaje que permite tener acceso a las funcionalidades administradas por la Interfaz de Programación de Aplicaciones (OpenAPI).

1.6.4. Entorno de Desarrollo Integrado

En la implementación de la propuesta de solución se define utilizar el IDE (Integrated Development Enviorment) NetBeans 8.0 que es una herramienta de código abierto con una gran base de usuarios y una comunidad en constante crecimiento. Existe además un número importante de módulos para extenderlo. Es un producto libre y gratuito sin restricciones de uso. Está compuesta también por una base modular y extensible usada como una estructura de integración para crear grandes aplicaciones de escritorio.

Entre sus características están las administraciones de ventanas, almacenamiento, interfaces y configuraciones de usuario. Soporta el desarrollo de aplicaciones Java (J2SE, web, EJB y aplicaciones móviles), empresariales con Java EE 5, JavaFX 2.0, WebLogic 12c y CSS3. Empresas independientes asociadas, especializadas en desarrollo de software, proporcionan extensiones adicionales que se integran fácilmente en la plataforma y que pueden también utilizarse para desarrollar sus propias herramientas y soluciones (Funes, 2008).

1.6.5. Interfaz de Programación de Aplicaciones

Para el desarrollo de la extensión es necesaria la utilización de una Interfaz de Programación de Aplicaciones (API – *Application Programming Interface*), que es una interfaz de comunicación entre componentes de software. Proporciona un conjunto de funciones de uso general, que son llamadas a bibliotecas que ofrecen acceso a ciertos servicios desde los procesos y representa un método para conseguir abstracción en la programación. En este caso se utilizará OpenAPI, que es un mecanismo de extensión proporcionada por Visual Paradigm for UML para extender las funcionalidades del software.

El API está basado en el lenguaje Java y permite tener acceso completo a los datos del modelo en el archivo de proyecto. Mediante el uso de este mecanismo se pueden implementar algunas funciones personalizadas llamadas *plugins* o extensiones para lograr determinados fines sobre la herramienta (Trujillo et al., 2013).

1.7. Conclusiones parciales

El análisis de los elementos que componen los Diagramas de Casos de Uso y la entrevista realizada a expertos del área de la Ingeniería de Software permitió identificar los errores que se pueden cometer en el modelado de estos diagramas. A partir del estudio de soluciones existentes para la evaluación y corrección de Diagramas de Casos de Uso se identificó que no existen herramientas CASE que tengan implementadas estas funcionalidades, por lo que se decide la construcción de una extensión para la herramienta Visual Paradigm for UML.

El estudio de las herramientas y tecnologías utilizadas para el desarrollo de aplicaciones de este tipo, posibilitó recopilar los basamentos teóricos para su selección. El análisis de las metodologías ágiles más utilizadas permitió la identificación de OpenUP como la más idónea para el desarrollo de la extensión.

Capítulo 2. Diseño de la propuesta de solución

En este capítulo se realiza la descripción de las principales definiciones asociadas al dominio del problema. También se especifican los requisitos funcionales y no funcionales a tener en cuenta en la implementación de la solución. Se agrupan los requisitos funcionales en casos de uso y se describe textualmente cada uno de ellos. Finalmente se definen los patrones a seguir en el diseño de la extensión.

2.1. Modelo de dominio

Para lograr una mayor comprensión del contexto en que se ubica la extensión y la posterior representación de la solución, se recurre al empleo del modelo de dominio. Es un subconjunto del modelo de negocio y se realiza cuando no están claros los procesos o no se identifican claramente los actores y trabajadores del negocio. Además, el modelo de dominio captura los tipos más importantes de objetos que existen, se identifican conceptos, se definen y se relacionan en un diagrama de clases UML (Milián, 2011).

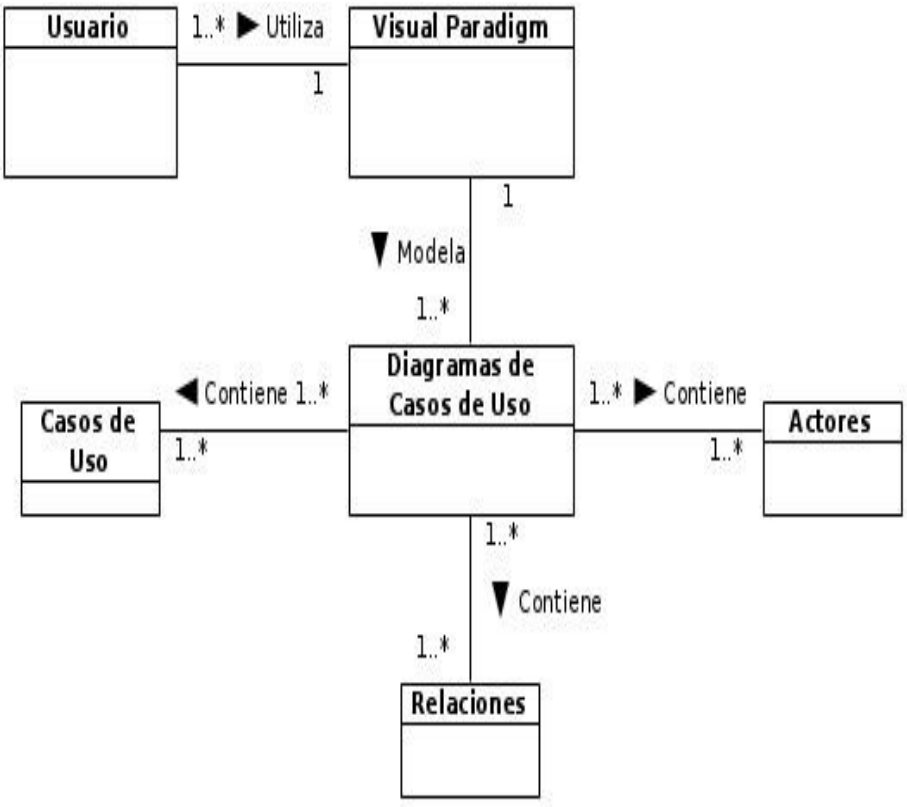


Figura 6. Diagrama de Clases del Dominio.

Tabla 2. Definición de los principales conceptos del modelo de dominio.

Elementos	Definiciones
Usuario	Persona que utiliza la herramienta Visual Paradigm para modelar y documentar el proceso de software.
Visual Paradigm	Herramienta CASE de modelado para el proceso de desarrollo de software. Entre sus características incluye el modelado de DCU e inclusión de funcionalidades a través de extensiones.
Diagrama de Casos de Uso	Modela las relaciones existentes entre los actores de un sistema y las funcionalidades del mismo.
Caso de Uso	Representa una o varias funcionalidades del sistema a construir.
Actor	Representa una abstracción de las entidades externas a un sistema.
Relación	Representa una conexión semántica materializada.

2.2. Propuesta de solución

Se propone desarrollar una extensión de la herramienta Visual Paradigm for UML para la evaluación y corrección de DCU, con la estructura que se muestra en la Figura 7. Esta extensión deberá, a partir de los DCU modelados con la herramienta, obtener los datos, identificar los errores en el modelado, mostrar al usuario los errores detectados y brindar la posibilidad de corregirlos. La solución debe ser capaz de ofrecer recomendaciones para mejorar el modelado del diagrama.

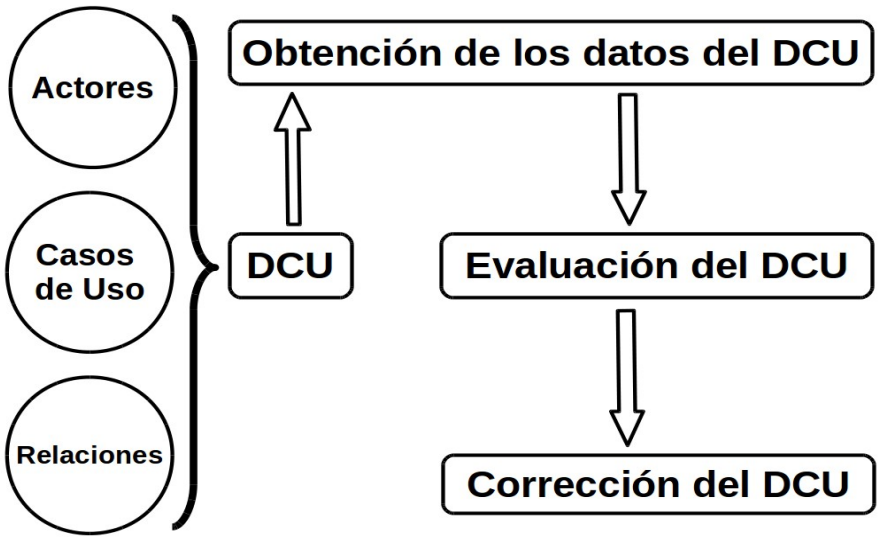


Figura 7. Propuesta de solución.

2.3. Especificación de requisitos

Los requisitos para un sistema son la descripción de los servicios proporcionados y sus restricciones operativas. Estos requisitos reflejan las necesidades de los clientes de un sistema que ayude a

resolver algún problema. El proceso de descubrir, analizar, documentar y verificar las funcionalidades del software se denomina Ingeniería de Requisitos (Sommerville, 2005). Los requisitos que se obtienen en este proceso pueden ser funcionales o no funcionales.

2.3.1. Requisitos funcionales

Los requisitos funcionales describen lo que el sistema debe hacer. Estos requisitos dependen del tipo de software que se desarrolle, de los posibles usuarios y del enfoque general tomado por la organización al redactarlos. Son declaraciones de los servicios que debe proporcionar el sistema (Sommerville, 2005). A continuación se listan los requisitos funcionales del sistema.

RF1: Listar Diagramas de Casos de Uso.

RF2: Identificar diagrama sin actor.

RF3: Identificar diagrama sin caso de uso.

RF4: Identificar actor sin relacionar.

RF5: Identificar caso de uso sin relacionar.

RF6: Identificar relación de Generalización entre casos de uso.

RF7: Identificar relación de Comunicación entre casos de uso.

RF8: Identificar relación de Comunicación entre actores.

RF9: Identificar relación de Extensión invertida entre casos de uso.

RF10: Identificar relación de Inclusión invertida entre casos de uso.

RF11: Identificar relación de Comunicación invertida entre actor y caso de uso.

RF12: Identificar relación de Generalización invertida entre actores.

RF13: Identificar relaciones múltiples entre actores.

RF14: Identificar relaciones múltiples entre casos de uso.

RF15: Identificar relaciones múltiples entre actores y casos de uso.

RF16: Identificar caso de uso incorrectamente nombrado.

RF17: Mostrar resultados de la identificación de errores en el diagrama.

RF18: Mostrar sugerencias para mejorar el diseño del diagrama.

RF19: Corregir diagrama sin actor.

RF20: Corregir diagrama sin caso de uso.

RF21: Corregir actor sin relacionar.

RF22: Corregir caso de uso sin relacionar.

RF23: Corregir relación de Generalización entre casos de uso.

RF24: Corregir relación de Comunicación entre casos de uso.

RF25: Corregir relación de Comunicación entre actores.

RF26: Corregir relación de Extensión invertida entre casos de uso.

RF27: Corregir relación de Inclusión invertida entre casos de uso.

RF28: Corregir relación de Comunicación invertida.

RF29: Corregir relación de Generalización invertida entre actores.

RF30: Corregir relaciones múltiples entre actores.

RF31: Corregir relaciones múltiples entre casos de uso.

RF32: Corregir relaciones múltiples entre actores y casos de uso.

RF33: Corregir caso de uso incorrectamente nombrado.

2.3.2. Requisitos no funcionales

Los requisitos no funcionales son propiedades o cualidades que el producto debe tener. Estas se refieren a las características que hacen al producto atractivo, usable, rápido o confiable. Los requisitos no funcionales a menudo se aplican a todo el sistema; normalmente apenas se aplican a características o servicios individuales del mismo (Sommerville, 2005). Entre los requisitos no funcionales necesarios para la realización de la extensión se encuentran:

Requisitos de Software

- Debe estar instalada en la computadora la versión 7.0 de Java o superior.
- Debe estar instalada en la computadora la versión 8.0 de la herramienta Visual Paradigm for UML.

Requisitos de Hardware

- Es necesario un mínimo de 512 MB (*MegaByte*) de RAM (*Random Access Memory*).
- Se necesita como mínimo 1GB (*Gigabyte*) de espacio libre en disco.
- Se requiere un procesador a 2.0 GHz como mínimo.

Requisito de Usabilidad

- Los errores detectados en el modelado del diagrama deben ser marcados para ganar en comprensión visual de lo que ha sido señalado.

2.4. Modelo de casos de uso del sistema

Concluido el proceso de levantamiento de requisitos, se continúa con la realización del modelo de casos de uso del sistema. Un modelo de casos de uso describe las funcionalidades propuestas de el producto a desarrollar, representa la interacción entre un usuario y el sistema. Para lograr un mejor entendimiento sobre los elementos que componen este modelo, se muestra el Diagrama de Casos de Uso del Sistema en la Figura 8. En la Tabla 3 se puede observar la descripción textual del caso de uso Evaluar DCU. La descripción del caso de uso Corregir DCU se puede ser consultada en el Anexo 3.

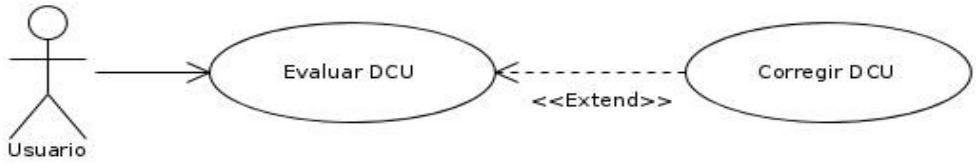


Figura 8. Diagrama de Casos de Uso del Sistema.

Tabla 3. Descripción del caso de uso Evaluar DCU.

Caso de Uso	Evaluar DCU.
Actor	Usuario.
Resumen	El caso de uso inicia cuando el Usuario selecciona la opción Evaluar DCU en la herramienta Visual Paradigm for UML. El sistema muestra el listado de los casos de uso modelados con la herramienta, para permitir seleccionar el que se desea evaluar. Una vez seleccionado el DCU a evaluar se muestra la lista de errores cometidos en el modelado del diagrama y sugerencias para mejorar el diseño de este.
Precondiciones	Debe existir al menos un Diagrama de Casos de Uso modelado en el Visual Paradigm for UML.
Referencias	RF1, RF2, RF3, RF4, RF5, RF6, RF7, RF8, RF9, RF10, RF11, RF12, RF13, RF14, RF15, RF16, RF17 y RF18.
Prioridad	Crítico.
Flujos Básicos de Eventos	
Acción del Actor	**Respuesta del Sistema**
1. El Usuario selecciona la opción Evaluar DCU a través del menú Herramientas/Evaluar DCU.	
	2. El sistema muestra la interfaz de la extensión.

Sección 1: “Listar Diagramas de Casos de Uso”	
Acción del Actor	**Respuesta del Sistema**
	1.1 El sistema muestra el listado de Diagramas de Casos de Uso modelados con la herramienta.
Flujos Alternos al paso 1 “Ausencia de Diagrama de Casos de Uso modelado”	
Acción del Actor	**Respuesta del Sistema**
	1.2 El sistema muestra un mensaje que informa que no existen Diagramas de Casos de Uso modelados con la herramienta.
Sección 2: “Evaluar Diagramas de Casos de Uso”	
Acción del Actor	**Respuesta del Sistema**
2.1 El Usuario selecciona el Diagrama de Casos de Uso a evaluar de la lista mostrada por el sistema.	
	2.2 El sistema muestra el listado con los errores cometidos en el modelado del diagrama y las sugerencias para mejorar el diseño del mismo.
Flujos Alternos al paso 2 “No se han cometido errores en el modelado del Diagrama de Casos de Uso ”	
Acción del Actor	**Respuesta del Sistema**
	2.3 El sistema muestra un mensaje que informa que no se han cometido errores en el modelado del diagrama.
Prototipo de Interfaz	

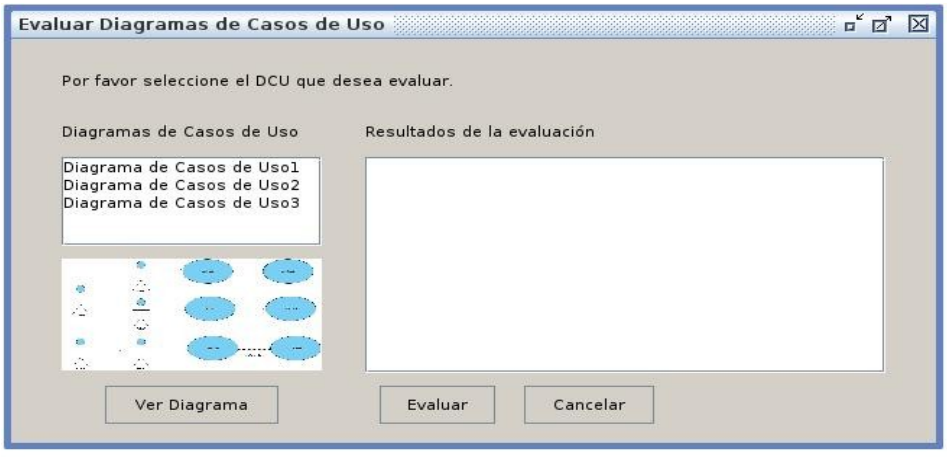


Figura 9. Prototipo de interfaz de usuario “Evaluar DCU”.

Poscondiciones	El Diagrama de Casos de Uso ha sido evaluado y se tiene el listado de los errores cometidos en su modelado.

2.5. Modelo de diseño

El modelo de diseño es un modelo de objetos que describe la realización de casos de uso, y sirve

como una abstracción para entrada al código fuente de la extensión. El modelo de diseño se utiliza como parte esencial para las actividades en ejecución y prueba, que se basa en el análisis y los requisitos de la arquitectura del sistema. Representa los componentes de la aplicación y determina su colocación adecuada dentro de la arquitectura en general propuesta para el desarrollo de la extensión (Trujillo et al., 2013). En la Figura 10 se muestra el Diagrama de Clases del Diseño.

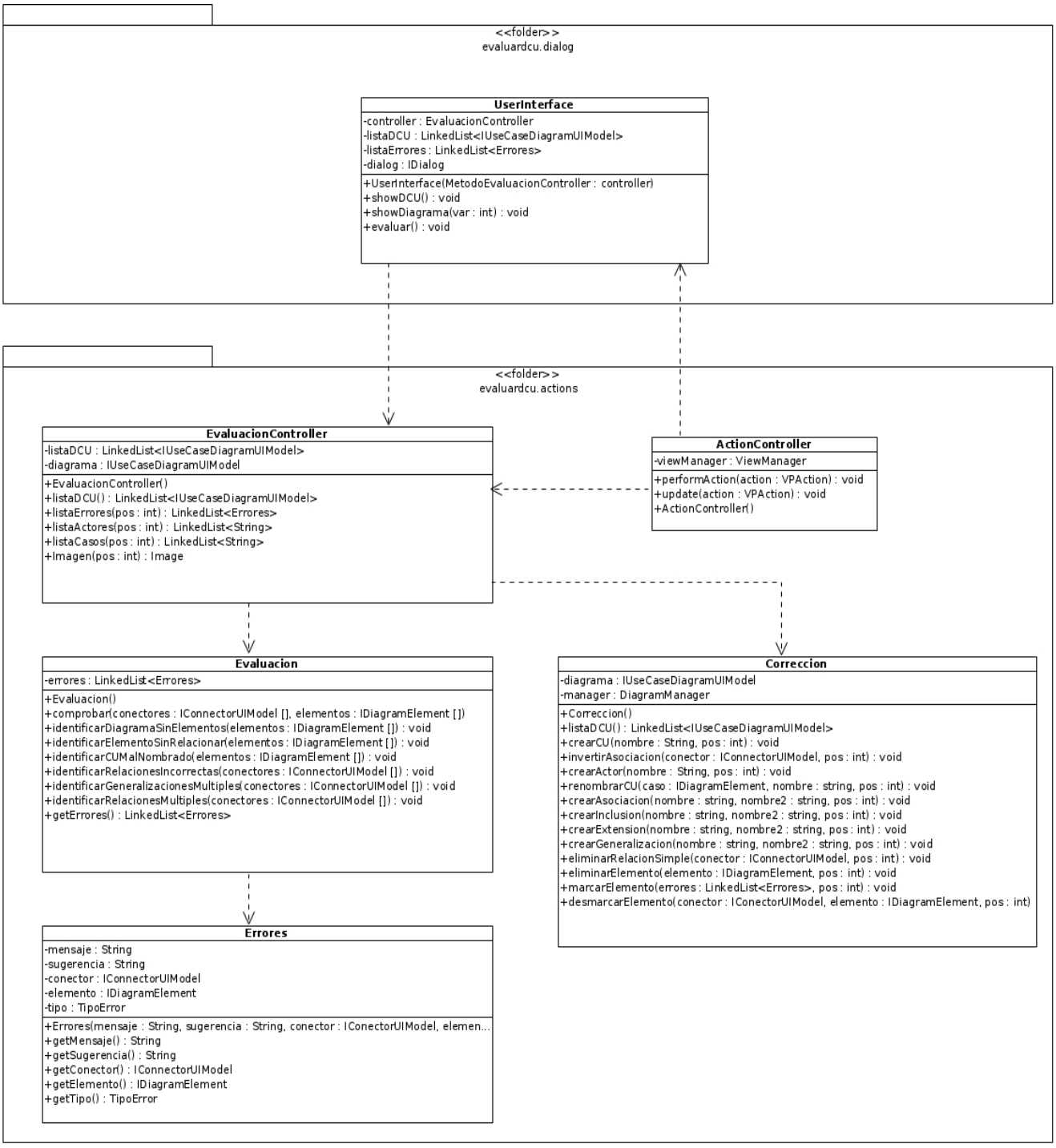


Figura 10. Diagrama de Clases del Diseño.

Tabla 4. Descripción de las clases del Diagrama de Clases del Diseño.

Clases	Descripción
ActionController	Es la clase que permite crear la interfaz visual de la extensión y junto con ella el objeto de la controladora que será utilizado durante todo el proceso que se realice desde la interfaz.
Action	Es la clase que brinda el OpenAPI por defecto para permitir la actualización de las acciones realizadas a través de la herramienta.
UserInterface	Interfaz visual que permite la interacción del usuario con la extensión, para realizar la evaluación y corrección de los DCU.
EvaluacionController	Es la clase que obtiene de la clase Evaluacion los errores cometidos en el modelado de los DCU para corregirlos.
Evaluacion	Es la clase donde se realizan todas las validaciones necesarias para la evaluación de los DCU.
Correccion	Es la clase donde se realizan todas las operaciones necesarias para la corrección de los DCU.
Errores	Esta clase contiene las propiedades de cada error.

2.6. Patrones utilizados

Para realizar un buen diseño durante el desarrollo de la extensión se propone el uso de patrones para permitir capturar, reutilizar y transmitir la experiencia obtenida en la presente investigación a otros desarrolladores en situaciones similares. El empleo de los patrones permite describir las formas de solucionar los problemas que se presentan de forma recurrente en el desarrollo de software, por lo que constituye un aspecto importante a destacar durante la implementación de la extensión.

Como características deseables para la aplicación de un patrón se encuentran:

- Debe solucionar un problema: los patrones capturan soluciones, no solo principios o estrategias abstractas.
- Debe ser un concepto probado: ser soluciones demostradas, no teorías o especulaciones.
- La solución no es obvia: muchas técnicas de solución de problemas tratan de hallar soluciones por medio de principios básicos. Los mejores patrones generan una solución a un problema de forma indirecta.
- Describe participantes y relaciones entre ellos: los patrones no sólo describen módulos sino estructuras del sistema y mecanismos más complejos (Marrero and Rosales, 2008).

Una vez analizadas las principales particularidades de los patrones de software resulta necesario analizar con mayor profundidad los patrones utilizados para el diseño de la propuesta de solución. Estos se encuentran agrupados en dos categorías: Patrón arquitectónico por capas y Patrones de

diseño, como se describe a continuación.

2.6.1. Patrón arquitectónico por capas

Los patrones arquitectónicos ofrecen soluciones a problemas de arquitectura de software, al proporcionar una descripción del tipo de relación y restricciones entre los elementos que conforman el diseño. Un patrón arquitectónico expresa un esquema de organización estructural esencial para un sistema de software, que consta de subsistemas, sus responsabilidades e interrelaciones. Son vistos con un nivel de abstracción mayor que los patrones de diseño, porque muchas arquitecturas diferentes pueden implementar el mismo patrón y por lo tanto compartir las mismas particulares.

La selección de un patrón de arquitectura está en correspondencia con las particularidades que posea el sistema y ajustándose a sus necesidades. Estas pueden variar en dependencia de la conformación del mecanismo de almacenamiento, presentación, controlador o negocio, entre otros componentes que pueda manejar el sistema.

Según las características de la extensión, se propone el uso del estilo arquitectónico por capas definido por (Buschmann, 2001). Esta arquitectura tiene como objetivo principal el de separar los diferentes aspectos del desarrollo en que está conformado el sistema y permite la construcción de sistemas débilmente acoplados. Al dividir un sistema en capas, cada una puede tratarse de forma independiente, sin tener que conocer los detalles de las demás.

La organización de las capas en la extensión quedaría agrupada en dos, dividiéndose en una capa Presentación y una Controladora. La capa de Presentación constituida por la clase UserInterface, permite al desarrollador la selección, evaluación y corrección de los Diagramas de Casos de Uso. La capa Controladora se encuentra representada principalmente por la clase ActionController a partir de la que se accede a las clases EvaluacionController, Evaluacion y Correccion que contienen la mayoría de las operaciones implementadas para el funcionamiento de la extensión. De esta manera se plantea que la capa de Presentación interactúa con la capa Controladora y desde la filosofía de arquitectura en capas, esto significa que la capa de Controladora presenta un conjunto de funcionalidades para brindar servicios a la capa inmediatamente superior de Presentación.

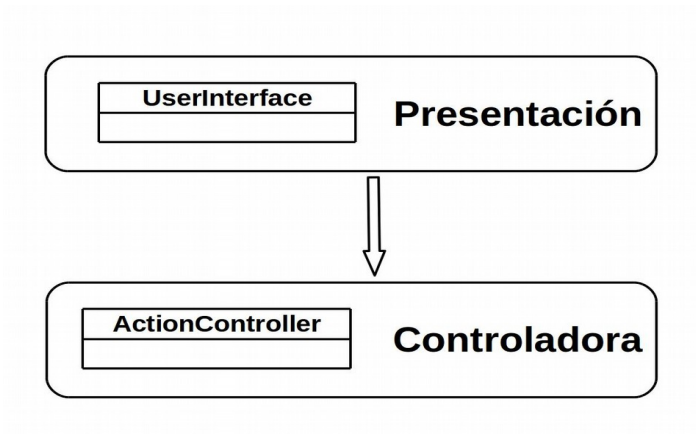


Figura 11. Arquitectura en dos capas de la extensión.

2.6.2. Patrones de diseño

El objetivo de los patrones es crear un cúmulo de bibliografía que ayude a los desarrolladores de software a resolver problemas recurrentes que surgen a lo largo del desarrollo. Los patrones ayudan a crear un lenguaje compartido para comunicar perspectiva y experiencia acerca de dichos patrones y sus soluciones. La codificación formal de estas soluciones y sus relaciones permite acumular con éxito el cuerpo de conocimientos que define nuestra comprensión de las buenas arquitecturas que satisfacen las necesidades de sus usuarios (Pressman, 2010).

Existen varios patrones de diseño utilizados en el proceso de desarrollo del software que describen los principios fundamentales del diseño de objetos. Estos son agrupados fundamentalmente por dos grandes grupos conocidos como Patrones de Software para la Asignación General de Responsabilidad (GRASP - *General Responsibility Assignment Software Patterns*) y “La Banda de los cuatro” (GOF - *Gang of Four*).

2.6.2.1. Patrones GRASP

Los patrones GRASP constituyen un apoyo para entender el diseño de objetos y aplican el razonamiento para el diseño de una forma sistemática, racional y explicable. En consecuencia, los patrones GRASP son una codificación de principios básicos ampliamente utilizados (Larman, 2003). Los patrones GRASP empleados en el desarrollo de la extensión son descritos a continuación.

Patrón Experto

La responsabilidad de asignar una labor a la clase que tiene o puede tener los datos necesarios para cumplir determinada responsabilidad, es la solución que pretende dar este patrón ante el problema de

cómo realizar la asignación de la forma más eficiente posible. Si estas asignaciones de responsabilidades se hacen adecuadamente, los sistemas tienden a ser más fáciles de entender, mantener y ampliar, lo que nos ofrece la garantía de poder reutilizar los componentes en futuras aplicaciones. Se utiliza con frecuencia en la asignación de responsabilidades; es un principio de guía básico que se utiliza continuamente en el diseño de objetos (Larman, 2003). En el Diagrama de Clases del Diseño de la extensión el patrón Experto se ve reflejado en la clase Evaluacion, que contiene todos los métodos y atributos necesarios para evaluar los diagramas.

Patrón Controlador

Es el encargado de asignar la responsabilidad del manejo de uno de los eventos del sistema, a una clase facultada de atender determinada funcionalidad, dándole solución así al problema fundamental de este patrón, al saber quién debería ocuparse de dicho evento. En todos los casos, si se recurre a un diseño orientado a objetos, hay que elegir los controladores que manejen esos eventos de entrada (Larman, 2003). En el Diagrama de Clases del Diseño de la extensión se ve reflejado el patrón Controlador en la clase EvaluacionController. Esta clase garantiza el flujo de datos entre las demás clases que componen la extensión.

Patrón Creador

Este patrón guía la asignación de responsabilidades relacionadas con la creación de objetos, una tarea muy común. La intención básica del patrón Creador es encontrar un creador que necesite conectarse al objeto creado en alguna situación. Con esto se favorece el bajo acoplamiento (Larman, 2003). El uso de este patrón se evidencia en la clase ActionController, que se encargan de crear instancias de las clases que proveen la información necesaria para su propio manejo, como por ejemplo la clase EvaluacionController y UserInterface.

Patrón Alta cohesión

Se basa en que los elementos de un componente colaboran para producir algún comportamiento bien definido, como una clase que tiene una responsabilidad moderada en un área funcional y colabora con otras clases para llevar a cabo las tareas (Larman, 2003). Se manifiesta el uso de este patrón en la propuesta de solución, como se puede apreciar en las clases EvaluacionController y Evaluacion muestran responsabilidades relacionadas coherentemente, que se complementan entre sí.

Patrón Bajo acoplamiento

Este patrón propone la asignación de responsabilidades de manera tal que la dependencia entre una

clase y otra sea la menor posible, de tal forma que se potencie la reutilización y se mitiguen los efectos que puedan producir en una, la realización de cambios en la otra. Para esto soporta el diseño de clases que son más independientes, lo que reduce el impacto del cambio (Larman, 2003). Se pone de manifiesto en la solución propuesta pues entre las clases de las capas Controladora y Presentación, existe una baja interdependencia, lo que se traduce en la posibilidad de efectuar cambios en estas sin que ocurran grandes afectaciones al resto del sistema.

2.6.2.2. Patrones GOF

Los patrones GOF se revelan como una forma indispensable de enfrentarse a la programación (Gamma, 1995). Entre los patrones utilizados para el desarrollo de la extensión se encuentran el *Iterator* y el *Singleton* agrupados como patrones de comportamiento y de creación respectivamente según la clasificación de GOF. Estos se encuentran representados a través de una descripción detallada acerca del propósito que recoge cada uno y la aplicación que tienen durante el desarrollo de la extensión como se muestra a continuación.

Patrón *Singleton* o Solitario

Este patrón es de tipo creacional a nivel de objetos. Su principal objetivo es garantizar que una clase solo tenga una única instancia, para proporcionar un punto de acceso global a la misma. Permite realizar refinamientos en las operaciones y en la representación, mediante la especialización por herencia de “Solitario”. Es fácilmente modificable para permitir más de una instancia y para controlar el número de las mismas (Gamma, 1995).

En la extensión es utilizado este patrón directamente en la implementación de la clase EvalaucionController para mantener la constancia entre los objetos obtenidos del DCU modelado por el usuario. Esta única instancia de la clase pone de manifiesto el uso del patrón *Singleton* y su ventaja de evitar la duplicidad a través del punto de acceso global que es capaz de definir. El siguiente código de la Figura 12 muestra la aplicación en la extensión del patrón descrito.

```
public static EvaluacionController metodoEvaluacion() {
    if (metodoEvaluacion == null) {
        metodoEvaluacion = new EvaluacionController();
    }
    return metodoEvaluacion;
}
```

Figura 12. Ejemplo de utilización del patrón Singleton en la extensión.

Patrón *Iterator* o Iterador

El patrón *Iterador* es de tipo comportamiento a nivel de objetos. A través de su utilización es posible acceder de forma secuencial a cada uno de los elementos de un objeto agregado sin exponer su representación interna. Además, permite realizar recorridos sobre objetos compuestos independientemente de la implementación de estos. Su utilización tributa al incremento de la flexibilidad porque es posible utilizar nuevas formas de recorrer una estructura con solo modificar el iterador en uso, cambiarlo por otro o definir uno nuevo. También facilita el paralelismo y la concurrencia, pues es posible que dos o más iteradores recorran una misma estructura simultánea o solapadamente (Gamma, 1995).

En la extensión propuesta, es aplicado este patrón directamente en la implementación de la clase controladora cuando se necesita proporcionar una interfaz uniforme para recorrer las diferentes estructuras proporcionadas por Visual Paradigm for UML. En la captura de todos los Diagramas de Casos de Uso creados, el Iterator permite obtener los elementos de un diagrama y almacenarlos en un listado. En el siguiente código se muestra la aplicación en la extensión del patrón descrito (ver Figura 13).

```
public LinkedList<IModelElement> listaElementos(int pos) {

    LinkedList<IModelElement> list = new LinkedList<IModelElement>();

    Iterator elementIterator = listaDCU().get(pos).diagramElementIterator();

    while (elementIterator.hasNext()) {

        IDiagramElement element = (IDiagramElement) elementIterator.next();

        if (!element.getModelElement().getName().equals("")) {
            list.add(element.getModelElement());
        }
    }
```

Figura 13. Ejemplo de utilización del patrón Iterator en la extensión.

2.7. Conclusiones parciales

La definición los requisitos funcionales permitió el diseño de la propuesta de solución en tres procesos fundamentales: obtención de los datos, evaluación y corrección del DCU. La utilización de los patrones de diseño permitió que la extensión sea escalable y se pueda realizar fácilmente un proceso de documentación de la implementación. La utilización del patrón arquitectónico por capas, hizo posible obtener como resultado un diseño con bajos niveles de acoplamiento entre capas y con una alta tolerancia al cambio.

Capítulo 3. Implementación y pruebas de la propuesta de solución

En este capítulo se explica la construcción del sistema propuesto, se describen sus componentes y los procesos de la implementación de la extensión. Además, se verifica la calidad de la aplicación y se validan los requisitos desarrollados a través de las pruebas.

3.1 Modelo de implementación

La implementación es aquella fase que describe el funcionamiento del sistema en un medio ejecutable. Para la implementación es preciso hacer que las decisiones tácticas de bajo nivel adapten el diseño al medio concreto de implementación. Si el diseño está bien hecho entonces las decisiones de implementación serán locales y ninguna afectará a grandes segmentos del software (Rumbaugh et al., 2007).

En el modelo de implementación se muestra la interacción de los componentes del sistema. Permite ver como han sido implementados los componentes del diseño y su organización. Evidencia también la jerarquía de los subsistemas de la implementación.

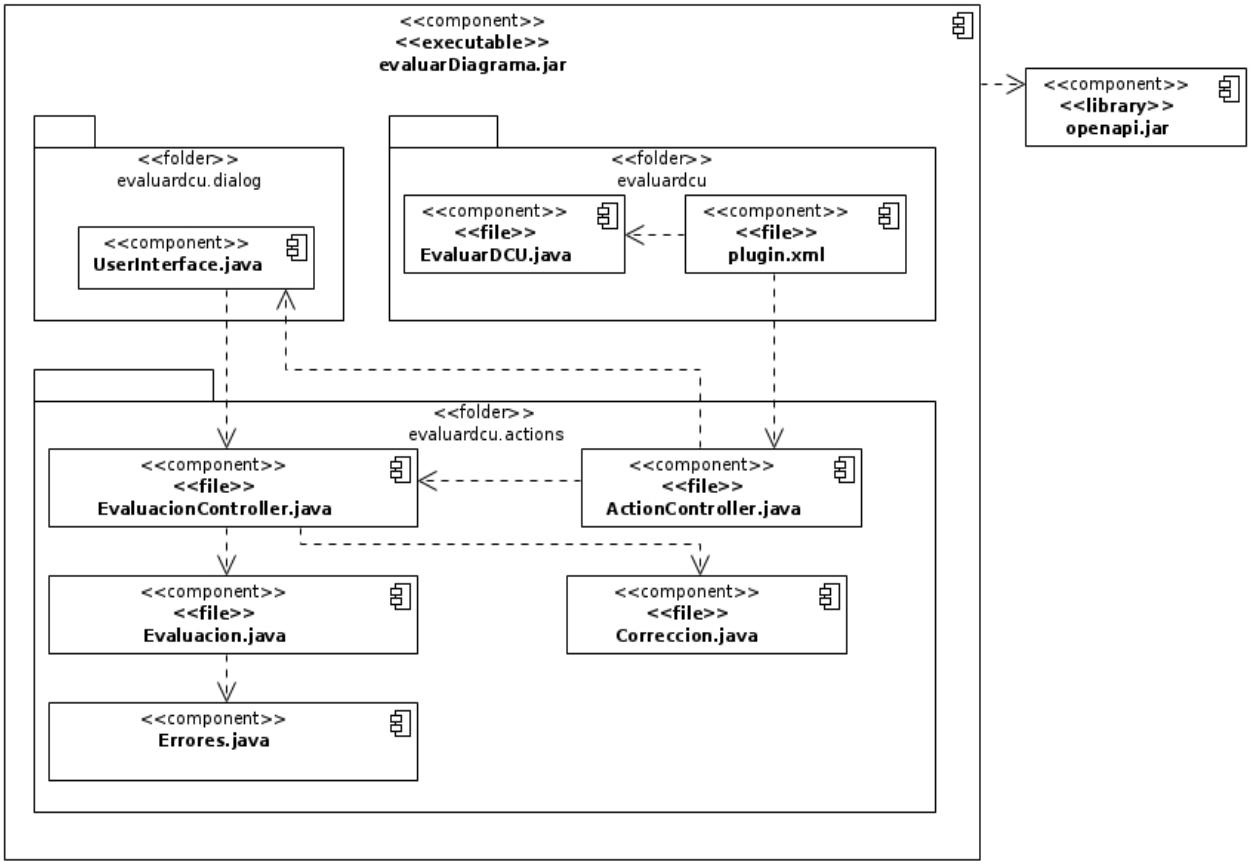


Figura 14. Diagrama de Componentes de la extensión.

Tabla 5. Descripción de los componentes del diagrama.

Clases	Descripción
Paquete evaluardcu	Paquete que contiene las clases relacionadas con la configuración de la extensión: plugin.xml y EvaluarDCU.java.
Paquete evaluardcu.actions	Paquete que agrupa las clases referentes a las acciones que son accesibles a través del menú herramientas. Ellas son: Action.java, ActionControler.java, Evaluacion.java, EvaluacionController.java y Correccion.java.
Paquete evaluardcu.dialog	Paquete que contiene la interfaz de la aplicación a través de la clase UserInterface.java.
EvaluarDCU.java	Carga y descarga el *plugin* mediante la clase VPPluginInfo que provee el OpenAPI.jar, para lo que utiliza la información definida en el archivo plugin.xml.
plugin.xml	Su función consiste en la configuración del *plugin*, define el nombre de la extensión, descripción, proveedor, las acciones a ejecutar y el menú en el que aparecerá dentro de la herramienta.

3.2 Implementación de la extensión

Como resultado del proceso de desarrollo del software se obtuvo una extensión de la herramienta Visual Paradigm for UML. Con esta extensión se pueden evaluar y corregir los DCU, de esta forma se reducen los errores de concepto frecuentemente reflejados en su modelado. La extensión se desarrolló tres procesos: obtención de los datos del DCU, evaluación del DCU y corrección del DCU.

3.2.1 Obtención de los datos

La obtención de los datos comienza cuando el usuario accede a la opción “Evaluar DCU” a través del menú: Herramientas/Evaluar DCU. Para esto, se construye una instancia de la clase controladora ActionController, que se encarga de crear la interfaz visual UserInterface, pasándole por parámetro una instancia de la clase EvaluacionController, encargada de evaluar los DCU modelados.

Todo esto es posible con la utilización de la clase Action, al implementar los métodos *performAction* y *update* de la clase VPContextActionController, ofrecida por el OpenAPI para actualizar las acciones que se realizan en la herramienta. La creación de la interfaz incluye mostrar el listado de los DCU modelados con la herramienta (ver Figura 15).

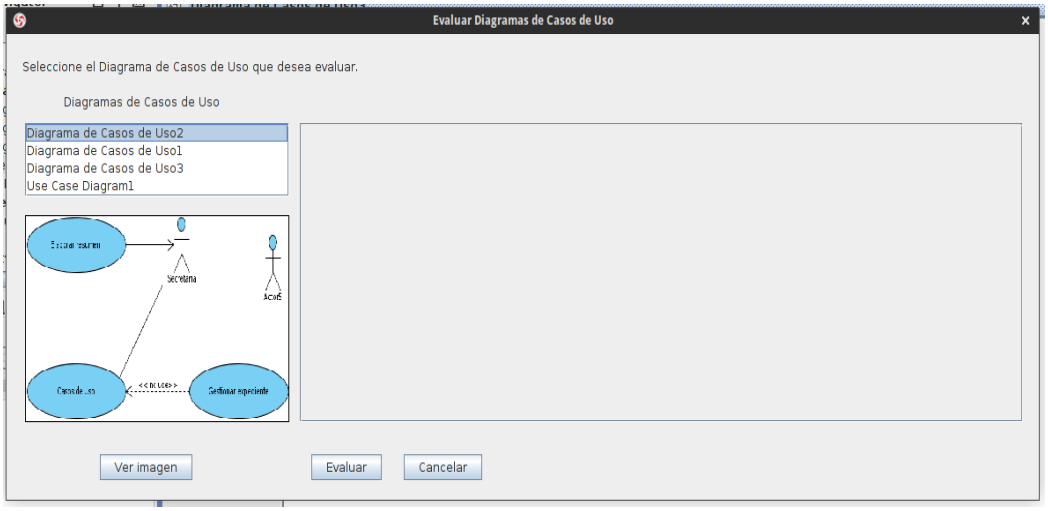


Figura 15. Interfaz de usuario de la extensión.

Para el proceso de obtención de los datos de un DCU modelado, es necesario el uso de un conjunto de componentes ofrecidos por el OpenAPI, como se describe en la Tabla 6.

Tabla 6. Descripción de los componentes utilizados para la obtención de los datos.

Componente	Descripción
DiagramManager	Interfaz que permite el control de los diagramas modelados.
IUseCaseDiagramUIModel	Interfaz que define los DCU modelados.
IConnectorUIModel	Interfaz que define los conectores del DCU.
IDiagramElement	Interfaz que define los elementos de un diagrama.
IDiagramProperty	Interfaz que define las propiedades de un diagrama.

3.2.2 Evaluación del Diagrama de Casos de Uso

La evaluación de un DCU inicia cuando el usuario, una vez mostrada la interfaz, selecciona un diagrama y la opción “Evaluar”. La finalidad del proceso de evaluación es obtener el listado de los errores en el modelado del DCU, para que estos puedan ser mostrados al usuario. Una vez obtenido el listado de errores, se muestran en la interfaz UserInterface. Por cada error mostrado en pantalla se da una sugerencia para que, si el usuario lo desea, pueda corregirlo manualmente (ver Figura 16).

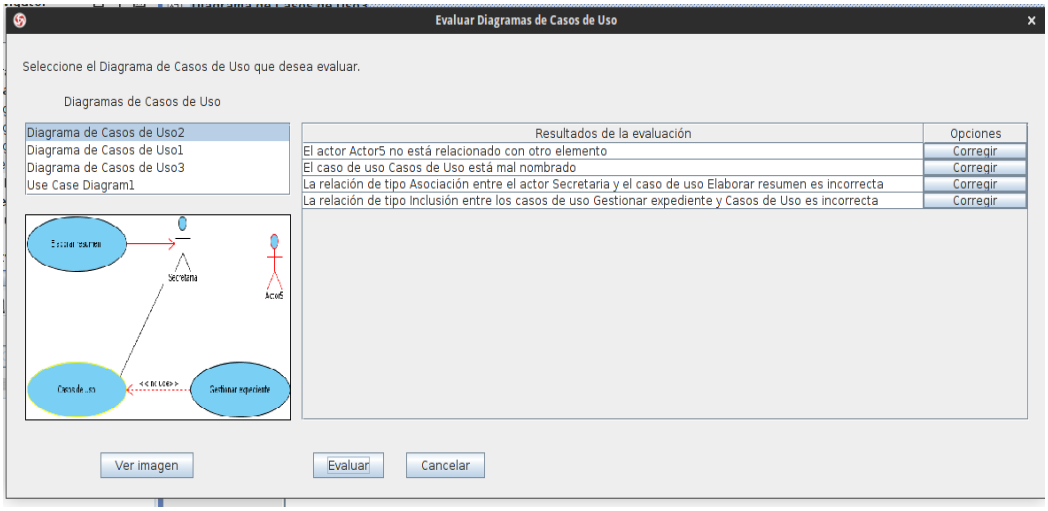


Figura 16. Interfaz de usuario de la extensión con los resultados de la evaluación de un DCU.

En la evaluación es utilizada la clase EvaluacionController, que obtiene la lista de los errores cometidos en el modelado del DCU, a través del método *listaErrores().* Este método crea una instancia de la clase Evaluacion, en la que se hacen todas las validaciones necesarias para la evaluación del diagrama. En la Tabla 7 se muestran los principales métodos de la clase Evaluacion.

Tabla 7. Descripción de los métodos para la evaluación del DCU.

Métodos	Parámetros	Descripción
comprobar	IConnectorUIModel conectores[], IDiagramElement diagramElements[]	Se encarga de hacer llamadas a los otros métodos de la clase y devuelve el listado de los errores cometidos en el modelado.
identificarDiagramaSinElementos	*diagramElements*	Valida si existe al menos un actor y un caso de uso en el diagrama, en caso de que no exista retorna un error.
identificarElementoSinRelacionar	*diagramElements*	Evalúa si todos los elementos han sido relacionados con al menos un elemento del diagrama, en caso contrario retorna un error.
identificarRelacionesMultiples	*conectores*	Verifica que no existan relaciones múltiples entre dos elementos del diagrama, en

		caso afirmativo retorna un error.
identificarRelacionesIncorrectas	*conectores*	Es responsable de evaluar que las relaciones entre los elementos del diagrama han sido correctamente modeladas, y la dirección de las mismas. En caso de que exista alguna relación incorrectamente modelada se retorna un error.

3.2.3 Corrección del Diagrama de Casos de Uso

Las correcciones del DCU deben hacerse de manera individual para cada error listado en la interfaz. El proceso comienza cuando el usuario selecciona la opción “Corregir” de un error mostrado en la interfaz. Al seleccionarlo se muestra otra ventana que contiene las posibles opciones y las sugerencias ofrecidas(ver Figura 17).

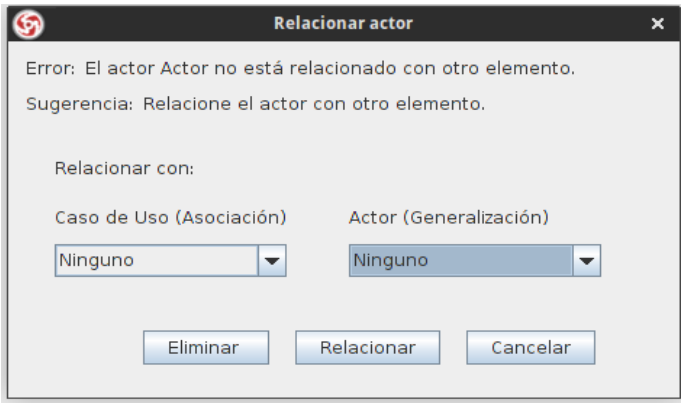


Figura 17. Interfaz de usuario de la extensión para la corrección de un error detectado.

En la corrección es utilizada la clase Correccion, donde se implementan los métodos necesarios para solucionar cada uno de los errores detectados. En esta clase se implementa un método complementario a cada uno de los métodos de la clase Evaluacion. Además, contiene algunos métodos claves para la modificación de los diagramas, responsables de crear, modificar y eliminar elementos de este.

3.3 Estándares de codificación

Los estándares de codificación son pautas de programación que no están enfocadas a la lógica del programa, sino a la estructura y apariencia física del código. Las normas de codificación definidas por

la línea de arquitectura están enfocadas a lograr una mejor comprensión y mantenimiento del código que responda a la complejidad dada por la cantidad de componentes y el alto nivel de integración que puede existir en el desarrollo del software (Quevedo et al., 2010).

Las realizaciones de revisiones frecuentes al código, la aplicación de forma continua de técnicas y estándares de codificación definidos, junto al empleo de buenas prácticas de programación, garantizan una alta calidad en el desarrollo de la extensión, además de proporcionar un código legible y reutilizable. Entre los estándares utilizados durante la implementación se encuentran:

Tamaño y organización de las líneas de código

La longitud de línea es aproximadamente de 80 caracteres. En caso de que una expresión ocupe más de este rango, podrá romper o dividirse en una nueva línea y deberán estar todas alineadas con respecto a la anterior, para mantener la legibilidad del código.

Declaraciones

Toda variable local tendrá que ser inicializada en el momento de su declaración, salvo que su valor inicial dependa de una llamada a otro método en determinado momento de ejecución de método. Las declaraciones deben situarse al principio de cada bloque principal en el que se utilicen y nunca en el momento de su uso.

Llaves de apertura y cierre

Se utiliza siempre llaves de apertura y cierre, incluso en situaciones en las que técnicamente son opcionales. Esto aumenta la legibilidad y disminuye la probabilidad de errores lógicos.

Nomenclaturas utilizadas

PascalCase establece que los nombres de los identificadores, las variables, métodos y clases están compuestos por una o más palabras juntas. Con ella cada palabra inicia con letra mayúscula y el resto en minúscula. Todas las clases están nombradas con la utilización del estándar PascalCase, nombrándolas de acuerdo al propósito y la función que corresponda. Ejemplo: las clases Action, ActionController, Evaluacion, EvaluacionController y UserInterface.

CamelCase es similar a PascalCase con diferencia en la letra inicial del identificador no comienza con mayúscula. Esta notación se utilizó para el nombre de funciones, atributos y variables.

- Nomenclatura de las funciones: El identificativo de todas las funciones o métodos se escribe con la primera palabra en minúscula de acuerdo a la función que realizan. Ejemplo: *listaDCU()*

y *listaElementos()*.

- Nomenclatura de los atributos: El identificativo de los atributos se escribe de acuerdo a su objetivo con la primera letra en minúscula. Ejemplos: listaElementos y listaRelaciones.

Notación húngara

Notación basada en definir prefijos para cada tipo de datos según el ámbito de las variables, con la finalidad de lograr mayor comprensión del nombre de la variable, método o función.

- Nomenclatura de las variables. El identificativo de los atributos se escribe según su objetivo y de acuerdo al estándar CamelCase, con la primera letra en minúscula. Ejemplo: listaDCU, que define una lista de diagramas y listaElementos, que define una lista de elementos de un diagrama.
- Nomenclatura de las constantes. Para el identificativo de las constantes se utilizan todas las letras en mayúscula. Ejemplo: DCU (Diagrama de Casos de Uso).

Ejemplo de código

El siguiente fragmento de código corresponde al escenario número uno del caso uso Evaluar DCU y su implementación se encuentra dentro de la etapa de obtención de los datos. En el código se evidencia el uso de los estándares de codificación.

```
public LinkedList<IDiagramUIModel> listaDCU() {
    IProject project = ApplicationManager.instance().getProjectManager().getProject();
    Iterator diagramIter = project.diagramIterator();
    LinkedList<IDiagramUIModel> listDiagram = new LinkedList<IDiagramUIModel>();
    String DCU = DiagramManager.DIAGRAM_TYPE_USE_CASE_DIAGRAM;
    while (diagramIter.hasNext()) {
        IDiagramUIModel diagrama = (IDiagramUIModel) diagramIter.next();
        if (diagrama.getType().equals(DCU)) {
            listDiagram.add(diagrama);
        }
    }
    return listDiagram;
}
```

Figura 18. Ejemplo de código de la extensión implementada.

3.4 Pruebas de software

Las pruebas son un proceso de ejecución de un programa con la intención de descubrir un error. Un buen caso de prueba es aquel que tiene una alta probabilidad de mostrar un error no descubierto

hasta entonces (Pressman, 2010).

Los autores (Pressman, 2010), (Somerville, 2005) y (Larman, 2003) coinciden en la definición de pruebas a nivel de unidad, sistema, integración y aceptación. La evaluación del sistema se realizó durante todo el ciclo de desarrollo de la extensión. Una vez implementada la extensión se realizaron pruebas a los cuatro niveles. Las pruebas a nivel de unidad se realizaron mediante pruebas de caja blanca y para las pruebas a nivel de sistema se utilizaron pruebas de caja negra.

3.4.1 Pruebas de caja blanca

La Prueba de caja blanca, es una filosofía de diseño de casos de prueba que usa la estructura de control descrita como parte del diseño a nivel de componentes para derivar casos de prueba. La prueba de caja blanca del software se basa en el examen cercano de los detalles de procedimiento. Las rutas lógicas a través del software y las colaboraciones entre componentes se ponen a prueba al revisar conjuntos específicos de condiciones y/o bucles. Puede seleccionarse y revisarse un número limitado de rutas lógicas y puede probarse la validez de las estructuras de datos importantes (Pressman, 2010).

Para la aplicación de las pruebas de caja blanca a la extensión se utilizó la biblioteca JUnit. Al aplicar pruebas funcionales a nivel de unidad con JUnit 4.5, se obtiene una clase de tipo Test para cada clase o componente de la implementación que se desea probar. Para la clase Evaluacion encargada de ejecutar todos los métodos necesarios para la evaluación de los DCU, se obtiene la clase de prueba EvaluacionIT. En esta clase se comprueba la respuesta cuando se agrega un nuevo error a la lista de los errores existentes. Los resultados obtenidos se pueden observar en la Figura 19.

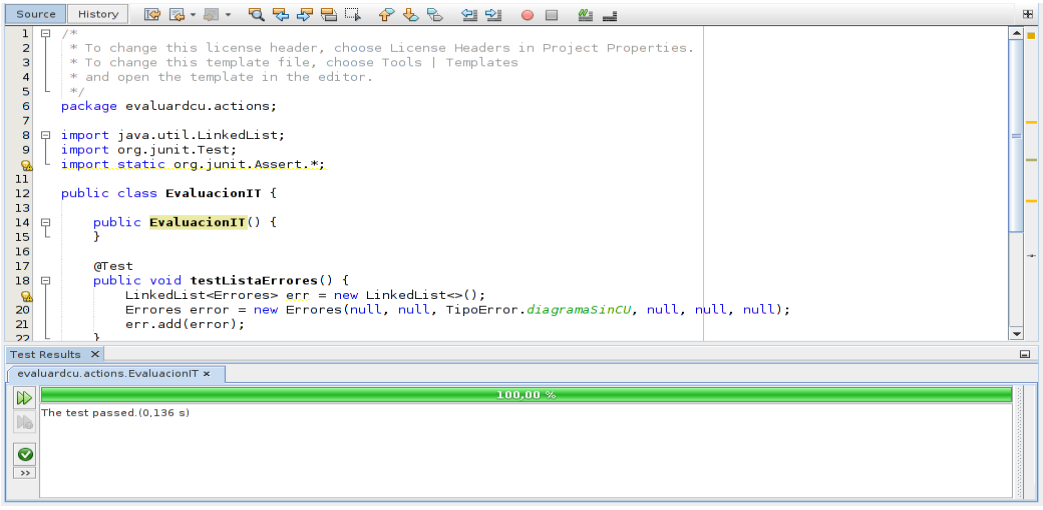


Figura 19. Pruebas con JUnit al método ListaErrores() de la clase Evaluacion.

Para aplicar el mismo procedimiento a la clase EvaluacionController, se obtiene la clase

EvaluacionControllerIT. En esta se comprueba la respuesta del sistema al añadir un nuevo elemento a la lista de los DCU existentes. Los resultados obtenidos fueron satisfactorios (ver Figura 20).

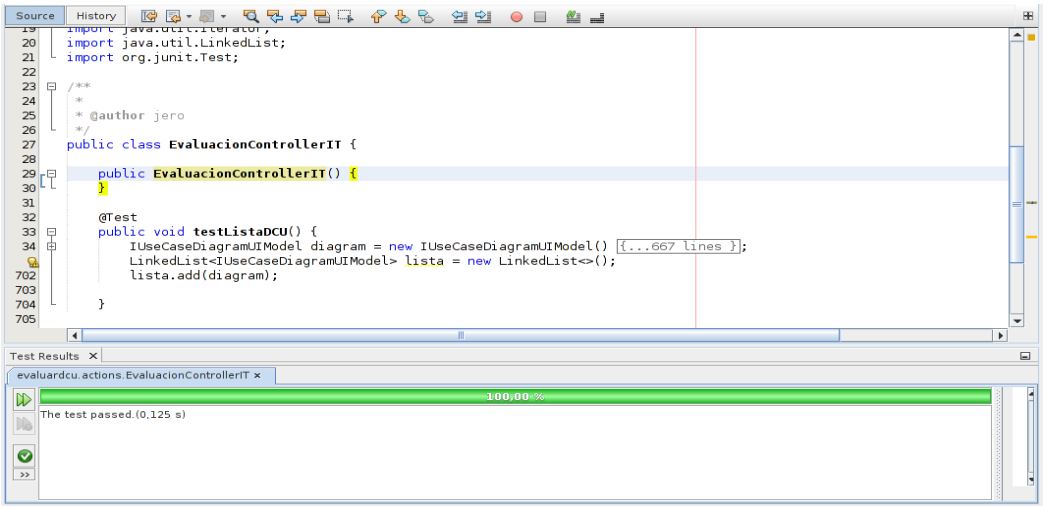


Figura 20. Pruebas con JUnit al método ListaDCU() de la clase EvaluacionController.

3.4.2 Pruebas de caja negra

La prueba de caja negra se refiere a las pruebas que se llevan a cabo en la interfaz del software. Una prueba de caja negra examina algunos aspectos fundamentales de un sistema con poca preocupación por la estructura lógica interna del software. Estas pruebas, también llamadas pruebas de comportamiento, se enfocan en los requisitos funcionales del software; es decir, las técnicas de prueba de caja negra permiten derivar conjuntos de condiciones de entrada que revisarán por completo todos los requisitos funcionales para un programa. Son un enfoque complementario que es probable que descubra una clase de errores diferente que los métodos de caja blanca. (Pressman, 2010).

Para realizar las pruebas a nivel de sistema a la extensión se utilizará la técnica de prueba de caja negra, con el método de partición de equivalencia. La partición de equivalencia es un método de prueba que divide el dominio de entrada de un programa en clases de datos de los que pueden derivarse casos de prueba. Un caso de prueba ideal descubre de primera mano una clase de errores que de otro modo podrían requerir la ejecución de muchos casos de prueba antes de observar el error general (Pressman, 2010).

3.4.2.1 Diseño de caso de prueba

Un caso de prueba se diseña según las funcionalidades descritas en los casos de uso. El propósito que se persigue con este artefacto es lograr una comprensión común de las condiciones específicas

que la solución debe cumplir. Se comienza con la descripción de los casos de uso del sistema, como apoyo para las revisiones. Cada planilla de caso de prueba recoge la especificación de un caso de uso, dividido en secciones y escenarios, que detalla las funcionalidades descritas en él y cada variable que recoge el caso de uso.

Diseño del caso de prueba del caso de uso Evaluar DCU

Este caso de uso inicia cuando el Usuario selecciona la opción Evaluar DCU en la herramienta Visual Paradigm for UML. El sistema muestra el listado de los casos de uso modelados con la herramienta, para que se pueda seleccionar el que se desea evaluar. Una vez seleccionado el DCU a evaluar se muestra la lista de errores cometidos en el modelado del diagrama y sugerencias para mejorar el diseño de este. Para esto debe existir al menos un Diagrama de Casos de Uso modelado en el Visual Paradigm for UML.

Tabla 8. Diseño de caso de prueba Evaluar DCU.

Sección	Escenario	Descripción	Flujo central
SC 1: Listar Diagramas de Casos de Uso.	EC 1.1: Existen DCU modelados con la herramienta.	El sistema muestra el listado de Diagramas de Casos de Uso modelados con la herramienta.	1. El usuario selecciona la opción Evaluar DCU. 2. El sistema muestra la interfaz de la extensión. 3. El sistema muestra el listado de los DCU modelados con la herramienta.
	EC 1.2: No existen DCU modelados con la herramienta.	El sistema muestra un mensaje que informa que no existen Diagramas de Casos de Uso modelados con la herramienta.	1. El usuario selecciona la opción Evaluar DCU. 2. El sistema muestra la interfaz de la extensión. 3. El sistema muestra un mensaje que informa que no existen DCU modelados con la herramienta.
SC 2: Evaluar Diagramas de Casos de Uso.	EC 2.1: Seleccionar DCU a evaluar.	El Usuario selecciona el Diagrama de Casos de Uso a evaluar de la lista mostrada en la interfaz del sistema.	1. El usuario selecciona el DCU que desea evaluar. 2. El usuario selecciona la opción Evaluar.
	EC 2.2: Mostrar	El sistema muestra el listado	1. El sistema muestra la lista de

	resultados de la evaluación.	con los errores cometidos en el modelado del diagrama y las sugerencias para mejorar el diseño del mismo.	errores cometidos en el modelado del DCU. 2. El sistema muestra recomendaciones para mejorar el diseño del DCU. 3. El sistema brinda la opción de corregir cada error cometido en el modelado del DCU.
	EC 2.3: No se han cometido errores en el modelado del Diagrama de Casos de Uso.	El sistema muestra un mensaje que informa que no se han cometido errores en el modelado del diagrama.	1. Se muestra una interfaz que informa que no se han cometido errores en el modelado del DCU.

Tabla 9. Matriz de datos de la sección 1: Listar Diagramas de Casos de Uso.

Escenario	Variables	Respuesta del sistema	Resultado de la prueba
EC 1.1: Existen DCU modelados con la herramienta.	V/ (Diagrama de Casos de Uso1, Diagrama de Casos de Uso2, Diagrama de Casos de Uso 3)	El sistema muestra el listado con los 3 Diagramas de Casos de Uso modelados con la herramienta.	Satisfactorio
EC 1.2: No existen DCU modelados con la herramienta.	V/ (“ ”)	El sistema muestra un mensaje que informa que no existen Diagramas de Casos de Uso modelados con la herramienta.	Satisfactorio

Tabla 10. Matriz de datos de la sección 2: Evaluar Diagramas de Casos de Uso.

Escenario	Variables	Respuesta del sistema	Resultado de la prueba
EC 2.1: Seleccionar DCU a evaluar.	V/ (Diagrama de Casos de Uso1)	Se selecciona el Diagrama de Casos de Uso1 a evaluar de la lista mostrada en la interfaz del sistema.	Satisfactorio
EC 2.2: Mostrar resultados de la evaluación.	V/ (Diagrama de Casos de Uso1)	El sistema muestra el listado con los errores cometidos en el modelado del diagrama y las	Satisfactorio

		sugerencias para mejorar el diseño del mismo.	
EC 2.3: No se han cometido errores en el modelado del Diagrama de Casos de Uso.	V/ (Diagrama de Casos de Uso1)	El sistema muestra un mensaje que informa que no se han cometido errores en el modelado del diagrama.	Satisfactorio

3.4.2.2 Resultado de las pruebas de caja negra

Las pruebas se desarrollaron en cuatro iteraciones, a los treinta y tres requisitos funcionales. En la primera iteración se obtuvieron cinco no conformidades, que fueron resueltas satisfactoriamente. En la segunda iteración se detectaron tres nuevas no conformidades que fueron solucionadas. En la tercera iteración donde se encontró una no conformidad que fue resuelta. Para la cuarta iteración no se detectaron nuevas no conformidades, por lo que se decidió no continuar con las iteraciones. En el gráfico de la Figura 21 se muestran los resultados de la aplicación de las pruebas funcionales al sistema.

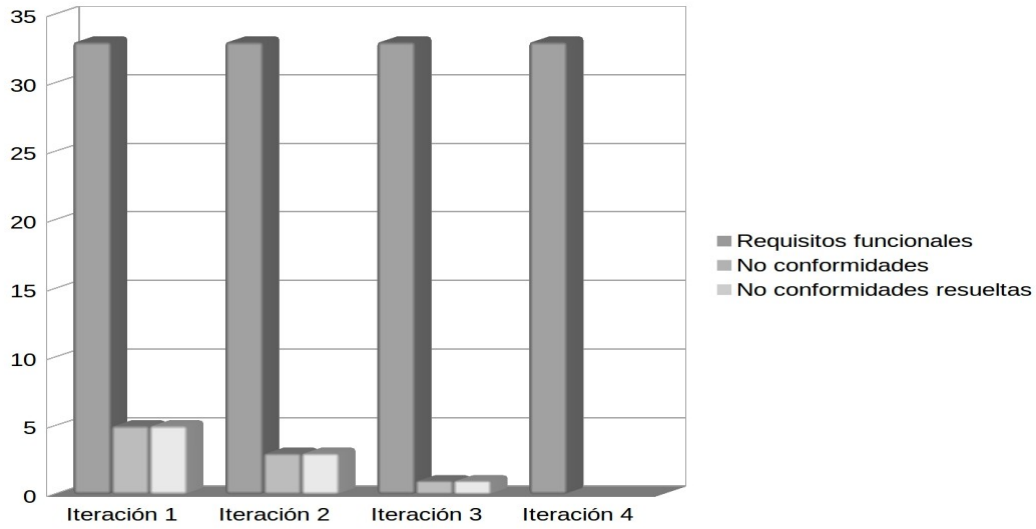


Figura 21. Resultados de las iteraciones de las pruebas funcionales.

Las no conformidades detectadas estuvieron relacionadas con el funcionamiento de las interfaces del usuario. Generalmente asociadas a mensajes de información no personalizados o respuestas no deseadas para los valores de entrada definidos. La realización de las pruebas permitió identificar los errores que no habían sido detectados hasta el momento.

3.4.3 Integración de la extensión con la herramienta Visual Paradigm for UML

Puesto que el software orientado a objetos no tiene una estructura de control jerárquico, las estrategias de integración tradicionales, descendente y ascendente, tienen poco significado. Además, integrar operaciones una a la vez en una clase con frecuencia es imposible debido a las interacciones directas e indirectas de los componentes que constituyen la clase. Existen dos estrategias para la prueba de integración de los sistemas OO.

La primera, prueba basada en hebra, integra el conjunto de clases requeridas para responder a una entrada o evento del sistema, donde cada hebra se integra y prueba de manera individual. El segundo enfoque de integración, prueba basada en uso, comienza la construcción del sistema al probar las clases independientes que usan otras clases. Después de probar las clases independientes, se examina la siguiente capa de clases que usan las clases independientes, llamadas dependientes. Esta secuencia de pruebas para las capas de clases dependientes continúa hasta que se construye todo el sistema (Pressman, 2010).

Para integrar la extensión desarrollada con la herramienta Visual Paradigm for UML 8.0 se utilizará el enfoque basado en uso, pues durante toda la implementación del sistema se probaron las clases dependientes e independientes a través de la integración de la extensión con la herramienta. Para lograr esta integración se deben garantizar primeramente tener instalada la versión 7.0 del OpenJDK de Java o superior y privilegios de administración sobre la computadora en la que va a ser montada. Para proceder a la integración se debe acceder a la carpeta de instalación de la herramienta y crear una carpeta llamada “plugins” (ver Figura 22).

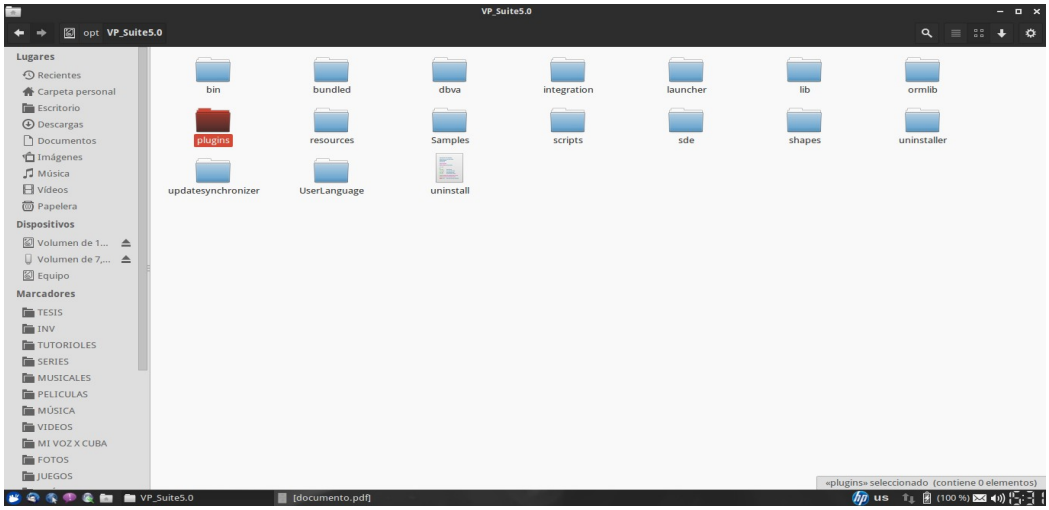


Figura 22. Creación de la carpeta plugins dentro de la carpeta de instalación del Visual Paradigm for UML.

Una vez creada esta se debe crear dentro ella otra con el nombre de la extensión, en este caso “evaluarDCU” (ver Figura 23).

Figura 23. Creación de la carpeta evaluarDCU dentro de la carpeta plugins.

Luego dentro de esta se copia el archivo plugin.xml y la carpeta del proyecto compilada con el NetBeans, esta es creada automáticamente dentro de la carpeta “build” del proyecto. De las subcarpetas dentro de “build” del proyecto se copian la carpeta “classes” y el fichero plugin.xml dentro de la carpeta “evaluarDCU” que se ha creado en el directorio de instalación del Visual Paradigm for UML (ver Figura 24).

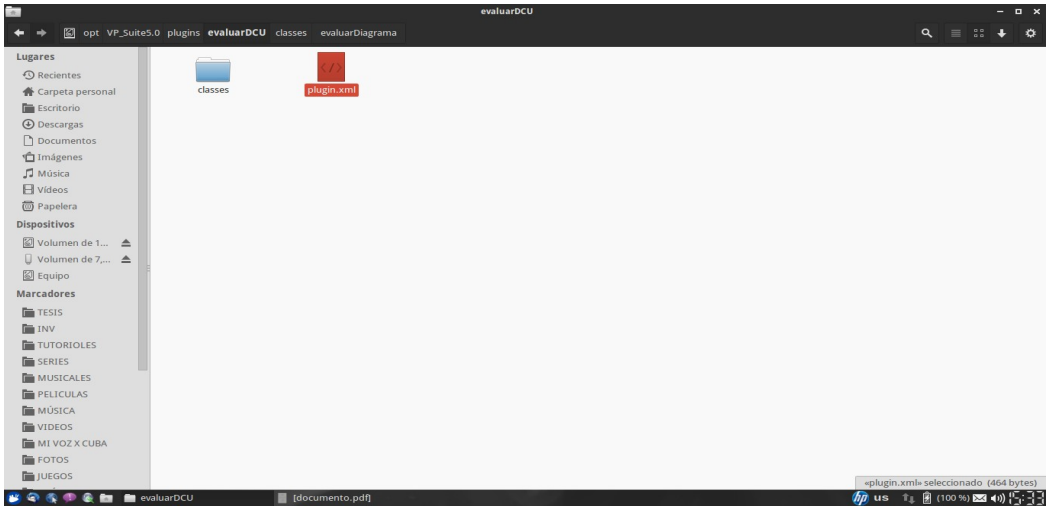


Figura 24. Copia de la carpeta classes compilada con el NetBeans y el fichero plugin.xml.

Una vez concluido este proceso se inicia la herramienta Visual Paradigm for UML 8.0 y la extensión

debe aparecer en el menú Tools/Evaluar DCU o en el menú Herramientas/Evaluar DCU para la versión en español.

3.4.4 Aceptación de la extensión

En las pruebas de aceptación se prueba una versión del sistema que podría ser entregada al cliente, quien valida que el sistema satisface sus necesidades. Si la entrega es lo suficientemente buena, el cliente puede entonces aceptarla para su uso (Somerville, 2005).

La extensión desarrollada fue sometida a un proceso de aceptación por parte del Departamento de Ingeniería y Gestión de Software de la Facultad 6 de la Universidad de las Ciencias Informáticas (ver Anexo 2). Durante este proceso el colectivo de Ingeniería de Software, ha reconocido que:

- El *plugin* obtenido tiene un alto valor didáctico como un recurso educativo en la asignatura de Ingeniería de Software I y II, permitiéndoles a los estudiantes autoevaluar la realización de sus Diagramas de Casos de Uso tanto en su trabajo de curso como en su estudio independiente o su desempeño en el rol de analistas en la Práctica Profesional.
- Con este producto el colectivo de Ingeniería de Software se nutre de una herramienta que incide en la calidad del autoaprendizaje de los estudiantes.
- Su uso no se limita exclusivamente a la formación pre-graduada sino también al propio desarrollo de software en los entornos productivos, especialmente para los analistas del sistema que trabajen sobre una Metodología guiada por casos de uso.
- La extensión ayuda a mitigar pérdidas de tiempo y retrasos en los cronogramas de proyecto. Además, contribuye a evitar no conformidades en las pruebas al sistema y conflictos internos en el equipo de desarrollo. Es una herramienta que puede contribuir a contrarrestar litigios en la aceptación del producto por parte del cliente, pérdida de credibilidad e incremento del presupuesto para la empresa desarrolladora.

3.5 Conclusiones parciales

La construcción de la extensión para el Visual Paradigm for UML permitió la evaluación y corrección de los Diagramas de Casos de Uso desde esta herramienta. Con las pruebas realizadas a la solución se comprobó que las funcionalidades se ejecutan correctamente y que la integración con la herramienta Visual Paradigm for UML fue satisfactoria. La aceptación del producto por parte del Departamento de Ingeniería y Gestión de Software validó el impacto e importancia de la extensión.

Conclusiones

Al finalizar la investigación se puede concluir que:

- A partir del estudio de la literatura científica sobre el modelado de DCU se pudo identificar que no existen soluciones que permitan evaluarlos y corregirlos, por lo que se decide la implementación de una extensión para la herramienta Visual Paradigm for UML que incorpore estas funcionalidades.
- Para implementar la propuesta de solución fueron seleccionadas un conjunto de herramientas y tecnologías que permitieron el desarrollo exitoso de la extensión, guiado por la metodología OpenUP.
- La implementación de la extensión para el Visual Paradigm for UML permitió la evaluación y corrección de los Diagramas de Casos de Uso desde esta herramienta, dándole cumplimiento al objetivo general de la investigación.
- Para garantizar el correcto funcionamiento de la solución desarrollada se realizaron pruebas al total de funcionalidades implementadas y se comprobó que la integración con la herramienta Visual Paradigm for UML fue satisfactoria.
- La aceptación del producto por parte del Departamento de Ingeniería y Gestión de Software validó el impacto e importancia de la extensión.

Recomendaciones

- Desplegar la extensión desarrollada para que esté a disposición a para toda la comunidad universitaria.
- Implementar nuevas funcionalidades a la extensión desarrollada que permitan la evaluación y corrección de otros tipos de diagramas.

Referencias Bibliográficas

- Balduino, R. Introduction to OpenUP. 2007. [En línea] [Consultado el: 27 de octubre de 2015]. Disponible en: [http://www.eclipse.org/epf/general/OpenUP.pdf].
- Burnstein, I. Practical software testing: a process-oriented approach. Berlin, Springer, 2006. 710p.
- Buschmann, F. Pattern-oriented software architecture: a system of patterns. Part II. New York, John Wiley & Sons, 2001. 467p.
- Deitel, H. M. and Deitel, P. J. Cómo programar en Java. Madrid, Pearson Educación, 2004.
- Dustin, E. et al. Automated software testing: introduction, management, and performance. Boston, Addison-Wesley Professional, 1999. 608p.
- Funes, L. Conociendo a NetBeans Platform: Introducción. 2008. [En línea] Le Funes, 2015 [Consultado el: 15 de abril de 2016]. Disponible en: [http://wiki.netbeans.org/ConociendoNetbeansPlatformIntroduccion].
- Gamma, E. Design patterns: elements of reusable object-oriented software. India, Pearson Education, 1995. 416p.
- García, J. J. et al. UML: el lenguaje estándar para el modelado de software. Novática: Revista de la Asociación de Técnicos de Informática, 2004, no 168, p. 4-5.
- Garlan, D. and Shaw, M. An introduction to software architecture. Advances in software engineering and knowledge engineering, 1993, vol. 1, no 3.4.
- Hernández, B. and Rodriguez, O. Sistema para la gestión del proceso de impresiones del Vicedecanato de Administración de la facultad 6 de la Universidad de las Ciencias Informáticas. Tesis para optar por el Título de Ingeniero en Ciencias Informáticas, Universidad de las Ciencias Informáticas, La Habana, 2015.
- Hernández, R. and Coello, S. El proceso de investigación científica. La Habana, Universitaria, 2011. 110p.
- Larman, C. UML y Patrones. Madrid, Pearson Educación, 2003. 624p.
- López, D. and Santa, J. A. Estudio comparativo de las herramientas CASE: StarUML, Poseidon for UML y Enterprise Architect, para el modelamiento de diagramas UML. Proyecto de Investigación como requisito para optar al Título de Ingeniero en Sistemas y Computación. Universidad Tecnológica de Pereira, Pereira, 2012.
- Marrero, Y. and Rosales, A. R. Propuesta de diseño arquitectónico de la Plataforma bioGRATO. Tesis para optar por el Título de Ingeniero en Ciencias Informáticas, Universidad de las Ciencias Informáticas, La Habana, 2008.
- Milián, C. Sistema Integrado de Gestión Estadística: Componente para la captación de

encuestas económicas. Tesis para optar por el Título de Ingeniero en Ciencias Informáticas, Universidad de las Ciencias Informáticas, La Habana, 2011.

- Myers, G. J. et al. The art of software testing. New York, John Wiley & Sons, 2011. 256p.
- OMG. Unified Modeling Lenguage, Versión 2.5. Massachusetts, OMG, 2015. 794p.
- Pressman, R. S. Ingeniería de Software, Un enfoque práctico. Séptima Edición. México, McGraw-Hill Companies, 2010. 777p.
- Quevedo, V. D. et al. CEDRUX. Solución para sistemas de control logístico. 15 Convención Científica de Ingeniería y Arquitectura. Instituto Superior Politécnico José Antonio Hecheverría. 2010. 9p.
- Quintero, J. B. et al. Un estudio comparativo de herramientas para el modelado con UML. Revista Universidad EAFIT, 2012, vol. 41, no 137, p. 60-76.
- Rondón, Y. et al. Diseño de la base de datos para sistemas de digitalización y gestión de medias. Revista de Informática Educativa y Medios Audiovisuales, 2011, vol. 8, no 15, p. 17-25.
- Rosales, Y. et al. Extensión de la herramienta Visual Paradigm para la generación de clases de acceso a datos con Doctrine 2.0. Serie Científica, 2013, vol. 6, no 10.
- Rumbaugh, J. et al. El Lenguaje Unificado de Modelado. Manual de Referencia. Denmark, Addison-Wesley, 2007. 688p.
- Saavedra, D. et al. Aplicación web para la realización de estudios farmacocinéticos, versión 2.0. Revista Cubana de Informática Médica, 2013, vol. 5, no 2, p. 118-131.
- Sommerville, I. et al. Ingeniería del software. Madrid, Pearson Educación, 2005. 687p.
- Trujillo, A. et al. Extensión de la herramienta Visual Paradigm para la generación de las clases de acceso a datos con Doctrine 2.0. Tesis para optar por el Título de Ingeniero en Ciencias Informáticas, Universidad de las Ciencias Informáticas, La Habana, 2013.
- Vidal, C. L., et al. Extensión del Diagrama de Secuencias UML (Lenguaje de Modelado Unificado) para el Modelado Orientado a Aspectos. Información tecnológica, 2012, vol. 23, no 6, p. 51-62.

Anexos

Anexo 1: Entrevista realizada para identificar errores frecuentes en el modelado de DCU.

I. Entrevista para identificar errores en el modelado de Diagramas de Casos de Uso

Instrucciones: Usted ha sido seleccionado para esta entrevista por su amplia experiencia en el proceso de desarrollo de software. Es necesario que responda a las siguientes preguntas basándose en su experiencia práctica y teórica en el modelado de Diagramas de Casos de Uso (DCU).

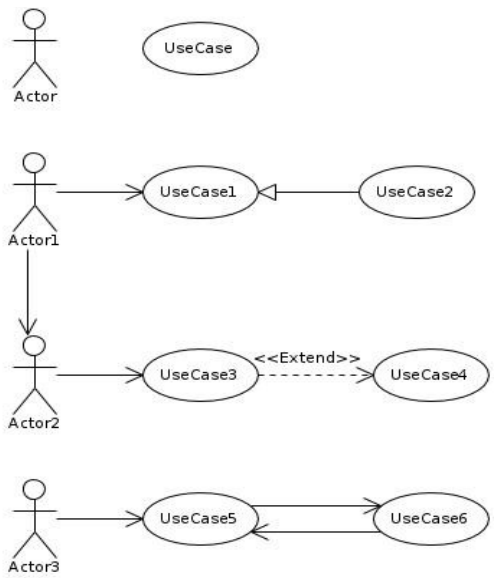
Pregunta 1: Mencione los elementos que componen un DCU.

Pregunta 2: ¿ Considera que esté correctamente modelado un DCU en el que no estén presentes todos sus elementos ? __ Si __ No

Pregunta 3: ¿ Cree que pueden existir elementos sin relacionar con otros en un DCU ? __Si __No

Pregunta 4: ¿ Cuáles son los tipos de relaciones existentes que usted conoce entre los elementos de un DCU?

Pregunta 5: ¿ Identifica usted algún error en el siguiente diagrama ? De ser positiva su respuesta marque los errores identificados en la imagen.



Pregunta 6: Mencione según su experiencia los errores que son cometidos frecuentemente en el modelado de DCU.

Anexo 2: Carta de aceptación por el Departamento de Ingeniería y Gestión de Software.

Departamento de Ingeniería y Gestión de Software
Facultad 6

La Habana, 24 de Junio de 2016
"Año 58 de la Revolución"

ACTA DE ACEPTACIÓN

De una parte, el Colectivo de Ingeniería de Software, del Departamento de Ingeniería y Gestión de Software de la Facultad 6, de la Universidad de las Ciencias Informáticas, representado en este acto por: Ing. Enier Alarcón Barbán, y de **otra parte** los estudiantes: Dayana Mendoza Peña y Lionel Rodolfo Baquero Hernández.

Primero: Se cumple con los 33 Requisitos Funcionales y los 3 Requisitos No Funcionales definidos en la investigación.

CONSIDERANDO: Que el plugin obtenido tiene un alto valor didáctico como un recurso educativo en la asignatura de Ingeniería de Software I y II. Permitiéndoles a los estudiantes autoevaluar la realización de sus Diagramas de Casos de Uso tanto en su trabajo de curso como en su estudio independiente o su desempeño en el rol de analistas en la Práctica Profesional, además les brinda retroalimentación de posibles soluciones a los 5 tipos de errores detectados.

CONSIDERANDO: Que con este producto el colectivo de Ingeniería de Software se nutre de una herramienta que incide en la calidad del autoaprendizaje de los estudiantes.

CONSIDERANDO: Que su uso no se limita exclusivamente a la formación pre-graduada sino también al propio desarrollo de software en los entornos productivos, especialmente para los analistas del sistema que trabajen sobre una Metodología guiada por Casos de Uso.

POR TANTO: Las Partes acuerdan formalizar mediante la presente acta, la aceptación del producto:
Extensión de la herramienta Visual Paradigm for UML para la evaluación y corrección de Diagramas de Casos de Uso.

Y para que conste, se extiende la presente Acta en dos (2) ejemplares, rubricados por **Las Partes**.

Entregan

Recibe

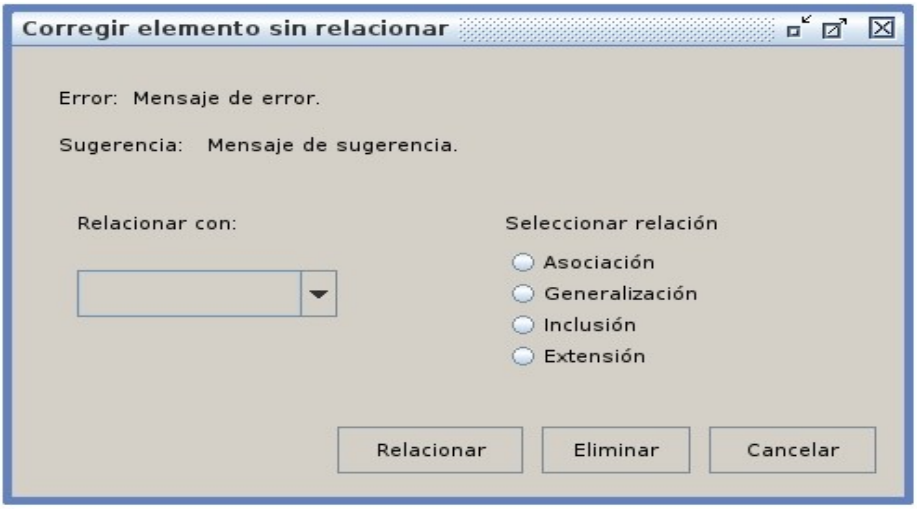
Universidad de las Ciencias Informáticas
Decanato Facultad 6

Anexo 3: Descripción textual del caso de uso Corregir DCU.

Caso de Uso	Corregir DCU.
Actor	Usuario.
Resumen	El caso de uso inicia cuando el Usuario selecciona la opción Corregir en la lista de errores obtenida. El Usuario selecciona el error que desea corregir. El sistema muestra una nueva interfaz que se corresponde con el tipo de error obtenido, permitiendo seleccionar las opciones para corregirlo.
Precondiciones	Debe existir al menos un error en el Diagrama de Casos de Uso.
Referencias	RF19, RF20, RF21, RF22, RF23, RF24, RF25, RF26, RF27, RF28, RF29, RF30, RF31, RF32, RF33.
Prioridad	Secundario.

Flujos Básicos de Eventos	
Acción del Actor	**Respuesta del Sistema**
1. El Usuario selecciona la opción Corregir que aparece a la derecha de cada error obtenido.	
	2. El sistema muestra la interfaz correspondiente al error.
Sección 1: "Corregir Diagrama sin caso de uso"	
Acción del Actor	**Respuesta del Sistema**
	1.1 El sistema muestra en la interfaz un formulario para crear un nuevo caso de uso.
1.2 El Usuario selecciona un actor y la opción "Adicionar".	
	1.3 El sistema crea un nuevo caso de uso y lo relaciona con el actor seleccionado mediante Asociación.
Sección 2: " Corregir Diagrama sin Actor"	
Acción del Actor	**Respuesta del Sistema**
	2.1 El sistema muestra en la interfaz un formulario para crear un nuevo actor.
2.2 El Usuario selecciona un caso de uso y la opción "Adicionar".	
	2.3 El sistema crea un nuevo actor y lo relaciona con el caso de uso seleccionado mediante Asociación.
Sección 3: "Corregir elemento sin relacionar"	
Acción del Actor	**Respuesta del Sistema**
	3.1 El sistema muestra en la interfaz un formulario para seleccionar los elementos a relacionar.
3.2 El Usuario selecciona un actor y/o un caso de uso y la opción "Relacionar".	

	3.3 El sistema crea las relaciones de Asociación y/o Inclusión/Extensión correspondientes.
Flujos Alternos al paso 2 “Corregir elemento sin relacionar”	
Acción del Actor	**Respuesta del Sistema**
3.2 El Usuario selecciona la opción “Eliminar”.	
	3.3 El sistema elimina el elemento sin relacionar.
Sección 4: “Corregir relación invertida”	
Acción del Actor	**Respuesta del Sistema**
	4.1 El sistema muestra en la interfaz un formulario con las opciones.
4.2 El Usuario selecciona la opción “Invertir”.	
	4.3 El sistema invierte la dirección de la relación.
Flujos Alternos al paso 2 “Corregir relación invertida ”	
Acción del Actor	**Respuesta del Sistema**
4.2 El Usuario selecciona la opción “Cambiar inicialización”.	
	4.3 El sistema elimina la inicialización del caso de uso secundario e inicializa el caso de uso primario.
Flujos Alternos al paso 2 “Corregir relación invertida ”	
Acción del Actor	**Respuesta del Sistema**
4.2 El Usuario selecciona la opción “Eliminar”.	
	4.3 El sistema elimina la relación.
Sección 5: “Corregir relación incorrecta”	
Acción del Actor	**Respuesta del Sistema**
	5.1 El sistema muestra en la interfaz un formulario para seleccionar la nueva relación.
5.2 El Usuario selecciona la opción “Sustituir”.	
	5.3 El sistema elimina la relación y crea en su lugar la relación correspondiente.
Flujos Alternos al paso 2 “Corregir relación incorrecta ”	
Acción del Actor	**Respuesta del Sistema**
5.2 El Usuario selecciona la opción “Eliminar”.	
	5.3 El sistema elimina la relación.
Sección 6: “Corregir relaciones múltiples”	
Acción del Actor	**Respuesta del Sistema**
	6.1 El sistema muestra en la interfaz un formulario con la opción “Eliminar”.
6.2 El Usuario selecciona la opción “Eliminar”.	
	6.3 El sistema elimina todas las relaciones múltiples.

Prototipo de Interfaz	
 Prototipo de interfaz de Usuario “Evaluar DCU”.	
Poscondiciones	El error seleccionado ha sido corregido.