UNIVERSIDAD DE LAS CIENCIAS INFORMÁTICAS

FACULTAD 2

Trabajo de Diploma para optar por el Título de Ingeniero en Ciencias Informáticas

Refactorización de la Arquitectura ABOS para el proyecto ABCD

Autor

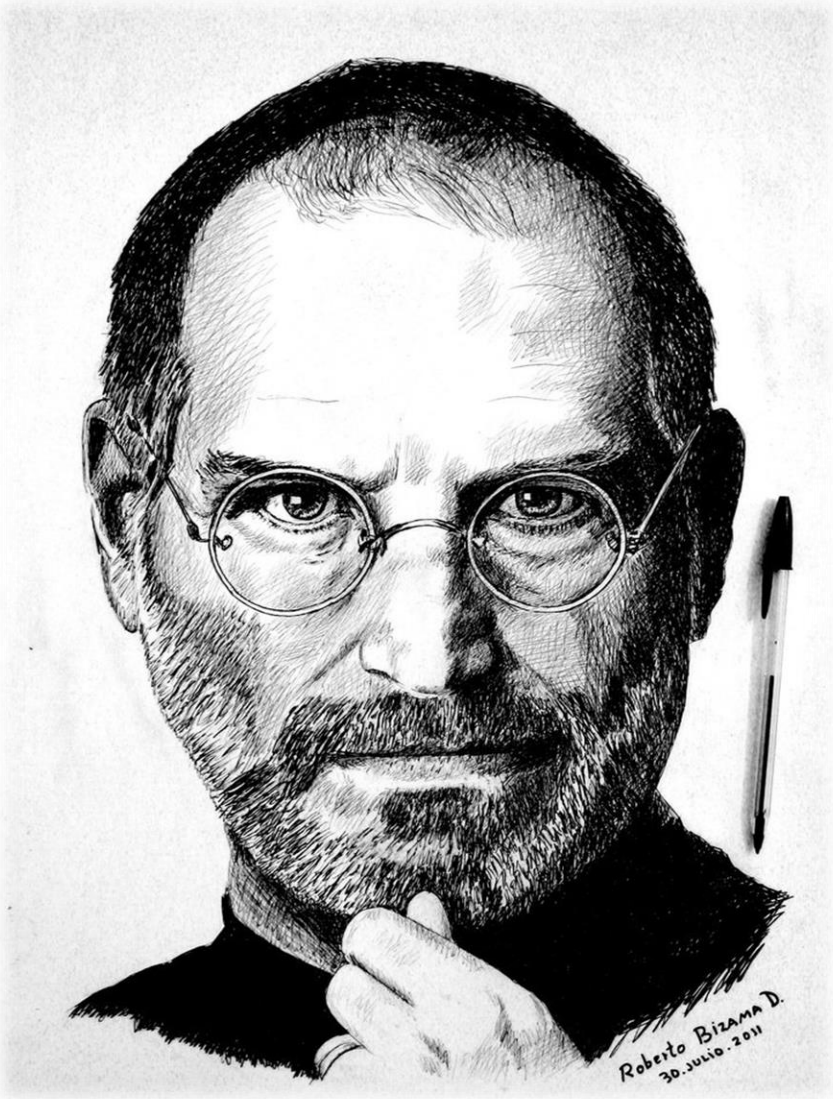
Rafael Alejandro Linares Torres

Tutor

Ing. Oigres Alvarez Pérez

La Habana, junio de 2016

"Año 58 de la Revolución"



"El diseño no es solo lo que se ve o lo que se siente.

Diseño es cómo funciona. "

Steve Jobs

DECLARACIÓN DE AUTORÍA

Declaro que soy autor de este trabajo y autorizo a la Facultad 2 de la Universidad de las Ciencias Informáticas que haga el uso que estime pertinente con este trabajo.

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

_____	_____
Autor	Tutor
Rafael Alejandro Linares Torres	Ing. Oigres Alvarez Pérez

SÍNTESIS DEL TUTOR

Graduado de Ingeniero en Ciencias Informáticas en la Facultad 2 de la Universidad de las Ciencias Informáticas (UCI), en el año 2008. Profesor de la misma que labora en el Departamento de Práctica Profesional del Centro de Informatización de la Gestión Documental impartiendo las asignaturas de Proyecto de Investigación y Desarrollo IV, VI y VII. Posee 7 años de experiencia en el desarrollo de aplicaciones empresariales implementadas en java. Actualmente, es arquitecto del proyecto Automatización de Bibliotecas y Centros de Documentación 3.0.

Correo oaperez@uci.cu

Dedicatoria

A mi eterna fuente de inspiración y de lucha, mi querida madre…

A mi hermano por ser faro y guía en mi futuro …

A todo el que siempre me apoyo y me brindó su ayuda…

A todos los que creyeron en mí; y a los que no también, por ustedes fue que más luche por cumplir este sueño…

Resumen

El Centro de Informatización de la Gestión Documental lleva más de un año desarrollando un sistema informático denominado Automatización de Bibliotecas y Centros de Documentación(ABCD). El cual tiene como fin automatizar la gestión de la documentación de una biblioteca o centro documental. Dicho sistema presentaba problemas con su arquitectura de software denominada Arquitectura Base Orientada a Servicios(ABOS), una arquitectura que tiene como base del diseño a *Open Service Gateway Iniciative* en su versión 4.3 y la Plataforma de Aplicaciones Remota en su versión 3.0.

El presente trabajo de diploma expone una investigación sobre las arquitecturas de software con el objetivo de **r**efactorizar la arquitectura ABOS, para facilitar el mantenimiento ABCD en su versión 3.0. Para la refactorización se utilizaron los métodos de evaluación de arquitecturas de software de Kazman específicamente el método (*Software Architecture Analysis Method*, SAAM). Como resultado de la evaluación se detectaron una serie de deficiencias que mostraba la arquitectura del software, entre las que se encontraba la falta de un estándar de diseño para las interfaces de usuario. Las deficiencias encontradas fueron solucionadas y validadas. Para la validación se utilizó como estrategias las pruebas unitarias y como técnica la prueba de caja blanca.

Palabras claves:

Arquitectura de Software, Evaluación, Refactorización.

Índice de contenidos

Índice de Tablas

Introducción

Las bibliotecas son lugares enriquecidos de sabiduría, a los cuales las personas asisten por el interés personal de adquirir conocimientos. La enorme cantidad de información que se almacena en una biblioteca puede llegar a deteriorarse y perderse por el trascurso de los años si no se lleva una correcta gestión de la misma. Actualmente existen una serie de sistemas informáticos encargados de automatizar este proceso, entre los que se encuentran el Microsoft SharePoint y gestión de documentos de Open Text. La mayoría de las bibliotecas cubanas, realizan la gestión de la información de forma manual, por lo que generan gran cantidad de documentación y datos importantes en formato duro, con la aparición de las nuevas tecnologías de la información y las comunicaciones y en particular el Internet, surgió la posibilidad de automatizar estos acervos.

En la Universidad de las Ciencias Informáticas, se está desarrollando un sistema para la gestión bibliotecaria, el mismo se nombra Automatización de Bibliotecas y Centros de Documentación (ABCD) en el Centro de Informatización de la Gestión Documental (CIGED). En la mayoría de los proyectos en desarrollo la correcta elaboración de la arquitectura de software es de vital importancia para el desarrollo sostenible del mismo. Pero se debe tener en cuenta que cualquier arquitectura de software no satisface las necesidades. En muchos casos solo se logra de manera parcial; lo que indica la necesidad de evaluar la arquitectura propuesta antes de emprender el desarrollo del proyecto en cuestión para garantizar un desarrollo sostenible del mismo.

Evaluar una arquitectura de software (ASW); es probar el potencial de la arquitectura diseñada, para alcanzar los atributos de calidad requeridos (1). Mediante la arquitectura de software es posible determinar la estructura del proyecto de desarrollo, en elementos como el control de la configuración, calendarios, el control de recursos, las metas de desempeño, la estructura del equipo de desarrollo y otras actividades que se realizan con apoyo de la arquitectura del sistema.

Por su parte en el campo de evaluación de las arquitecturas de software se reconocen dos grandes vertientes, una liderada por Kazman, creador de métodos de evaluación como el Método de Análisis de Arquitecturas de Software (*Software Architecture Analysis Method*, SAAM), el Método de Análisis de Acuerdos de Arquitectura (*Architecture Trade-off Analysis Method*, ATAM) y el Revisión Activa para el Diseño Intermedio (*Active Reviews for Intermediate Designs,* ARID), entre otros.

La otra vertiente, que encabeza Bosch, creador de técnicas de evaluación como: la basada en escenarios, basada en simulación, basada en modelos matemáticos y basada en experiencias. En este sentido las técnicas propuestas por Bosch, logran cierta evaluación en las fases tempranas del ciclo de desarrollo. Sin embargo, en su mayoría no hacen énfasis en todos los atributos de calidad relevantes para la arquitectura, y basan sus resultados en actividades que no pueden ser altamente medibles. Pues dependen en cierta medida de un factor subjetivo, como puede ser: la representatividad de los

escenarios, la exactitud del perfil seleccionado para cada atributo, la precisión con que el contexto del sistema simula las condiciones del mundo real, o la intuición y la experiencia.

Los métodos propuestos por Kazman, evalúan en el momento en que ya está definido el diseño arquitectónico; propuestas muchas veces no son aceptadas por los especialistas, pues se consumen tiempo y recursos en una arquitectura, que posiblemente al ser evaluada no logre algunos de los atributos de calidad requeridos por los clientes.

A esto se agrega que, en Cuba, en la Universidad de la Ciencias Informáticas, en el sistema ABCD 3.0 del centro CIGED de la Facultad 2 se trabaja con la arquitectura ABOS creada por José Rolando Lafourie Olivares en 2014. En el desarrollo de los módulos de ABCD 3.0 se han detectado un grupo de insuficiencias que quedaron formuladas a manera de **situación problemática** de la siguiente forma:

- ✓ El código desarrollado es muy extenso y carece de estructuras que lo hagan semánticamente entendible.
- ✓ Presenta problemas a la hora de comprender por los especialistas el código implementado, cambiar su estructura y diseño. También presenta problemas a la hora de añadir y eliminar un componente.
- ✓ Presenta duplicidad en la codificación existente en los módulos.
- ✓ No presenta una estandarización de los mensajes de retroalimentación.
- ✓ No presenta una política de tratamiento de excepciones y validaciones.
- ✓ Se hace engorroso el desarrollo, configuración y el mantenimiento presente y futuro del sistema.

Sobre la base de lo antes planteado se identificó como **problema a resolver**: ¿Cómo lograr el desarrollo, configuración y el mantenimiento presente y futuro del software ABCD 3.0?

Se define como **objeto de estudio:** Las arquitecturas de software elaboradas con RAP y OSGI.

Para resolver el problema se propone como **objetivo general:** Refactorizar la arquitectura ABOS para facilitar el mantenimiento del software ABCD 3.0. Siendo el **campo de acción**: Arquitectura ABOS.

Del objetivo general se derivan los siguientes **objetivos específicos**:

- Conceptualizar los referentes teóricos sobre las arquitecturas de software y sus métodos de evaluación.
- Evaluar la arquitectura ABOS y solucionar los problemas detectados.
- Realizar pruebas a la arquitectura ABOS luego hacerle las modificaciones.

Con el propósito de dar cumplimiento a los objetivos específicos planteados se trazaron las siguientes **tareas de investigación**:

1. Estudio y selección de los métodos de evaluación de arquitecturas de software de Kazman para detectar y solucionar los problemas de la arquitectura ABOS.
2. Estudio de RAP y OSGI para comprender la estructura de los componentes y el funcionamiento de la arquitectura.
3. Análisis de la estructura de los bundles de la arquitectura, para detectar deficiencias en la misma.
4. Refactorización de la arquitectura para eliminar errores y deficiencias detectadas.
5. Estudio de las técnicas y herramientas de pruebas para la corrección de errores a los componentes implementados.

Para apoyar el desarrollo de la investigación de emplean los siguientes métodos científicos.

Métodos Teóricos:

Analítico-Sintético: Permitió el estudio y selección del método de evaluación de arquitectura de software, en este caso SAAM. Además, posibilitó realizar un análisis detallado sobre el objeto de investigación y su conceptualización.

Histórico-Lógico*.* Permitió estudiar lo relacionado con las definiciones arquitecturas de software existentes, para valorar las regularidades del objeto de estudio y el campo de acción a través de los elementos que la componen.

Modelación. Contribuyó al estudio de RAP y OSGI como elementos fundamentales de la arquitectura ABOS. Así como la relación de sus componentes, estructura y funcionalidad lo que debeló una información actual y novedosa para un mejor funcionamiento del software ABCD 3.0.

Del nivel **empírico** se utilizaron los siguientes:

Revisión documental: La revisión de documentos emitidos por el Instituto de Ingeniería de Software (*Software Engineering Institute*, SEI), donde se exponen casos de estudio, que utilizan métodos de evaluación de arquitecturas, brindó resultados y dificultades de cada uno, de forma valorativa y comparativa.

Medición: Se utilizó para comprobar a través de las pruebas unitarias con JUnit el funcionamiento de la arquitectura.

El documento está estructurado por la introducción, tres capítulos, conclusiones, recomendaciones y bibliografía. A continuación, se hace una breve descripción de los capítulos.

El Capítulo 1 aborda la fundamentación teórica de la investigación. Se hace un análisis a los conceptos de arquitectura de software. Se profundiza en las diferentes vertientes existentes para la evaluación de las arquitecturas de software y se realiza una caracterización de la arquitectura ABOS donde se definen herramientas y tecnologías que la componen.

El Capítulo 2 aborda la evaluación de la arquitectura ABOS utilizando SAAM, método de evaluación de Kazman. También se proponen soluciones a los problemas detectados tras la evaluación.

El Capítulo 3 aborda la validación de las soluciones propuestas en el capítulo 2.

CAPÍTULO 1: FUNDAMENTOS TEÓRICOS SOBRE LAS ARQUITECTURAS DE SOFTWARE Y SUS MÉTODOS DE EVALUACIÓN

Este capítulo aborda las consideraciones teóricas y métodos de evaluación sobre la arquitectura de software. Se precisan las diferentes definiciones, así como los componentes y estructuras de la arquitectura ABOS.

1.1 Antecedentes de arquitectura de software

No es, sino hasta 1989 con la publicación del libro, Sistemas de Mayor Escala Requieren Abstracciones de Mayor Nivel (*"Larger Scale Systems Require Higher-Level Abstractions"*), de Mary Shaw, que se dan los primeros pasos por definir "arquitectura de software", en el cual Shaw define la arquitectura de software, de la siguiente manera:

… "Arquitectura de software es el estudio de la estructura a gran escala y el rendimiento de los sistemas de software. Aspectos importantes de la arquitectura de un sistema; incluyen la división de funciones entre los módulos del sistema, los medios de comunicación entre módulos, y la representación de la información compartida" (2).

Luego surge una marcada tendencia hacia la búsqueda de un modelo de estructuración de software, y se diseñan un conjunto de modelos de dominios, basados en diseños genéricos, como es el caso del Dominio Especifico de Arquitecturas de Software, (*Domain-Specific Software Architectures,* DSSA) elaborado por Mettala y Graham, en el año 1992 (3).

Sin embargo hasta 1994, con el lanzamiento del libro Introducción a la Arquitectura de Software, ("*An Introduction to Software Architecture*"), de Mary Shaw y David Garlan, se introduce, la necesidad de la creación de la arquitectura de software como disciplina científica, cuyo objeto de estudio no es más que la determinación de un conjunto de paradigmas que establezcan una organización del sistema a alto nivel, la interrelación entre los distintos componentes que lo conforman y los principios que orientan su diseño y evolución (4).

En esta ponencia además se introduce por primera vez el término Estilos Arquitectónicos, (*Architectural Styles*), y donde se definen una gran gama de ellos, entre los que se encuentran: Tubería y filtros, Repositorio, Arquitectura en capas, Arquitectura basada en eventos, etc.

En "el año de oro de la arquitectura de software", como podría ser considerado, 1994, tiene lugar otro acontecimiento arquitectónico relevante, pues se lanza el concepto de Lenguajes de Descripción Arquitectónica (ADSL), en el libro Características de Lenguajes de Alto Nivel de Arquitectura de Software, ("*Characteristics of Higher Level Languages for Software Architecture*"), de Mary Shaw y David Garlan (5).

En este libro se señalan las alternativas que hasta el momento se poseen para la definición de la arquitectura de software de un sistema. En primer lugar, a través de la modularización de las

herramientas de programación existentes y de los módulos de conexión entre ellas y, en segundo lugar, describir sus diseños usando diagramas informales y frases idiomáticas. A partir de estas reflexiones, Mary Shaw y David Garlan definen un conjunto de regularidades y propiedades específicas, que constituyeron las bases de los ADSL (4).

Otra definición de Arquitectura de Software es:
"Toda la arquitectura es diseño, pero no todo el diseño es arquitectura. La arquitectura representa las decisiones de diseño significativas que le dan forma a un sistema, donde lo significativo puede ser medido por el costo del cambio" (6).

Esta última definición se infiere que lo que distingue a la arquitectura de otro tipo de diseño es el significado en la dificultad o costo del cambio. La arquitectura del software es la que mantiene unidas las nociones de: diseño, estructura, estilo, racionalidad, proceso y costo. Actualmente, en la literatura, es posible encontrar numerosas definiciones del término arquitectura de software, cada una con planteamientos diversos. Lo que hace evidente que su conceptualización sigue todavía en discusión; puesto que no es posible referirse a un diccionario en busca de un significado, y tampoco existe un estándar que pueda ser tomado como marco de referencia único.

Sin embargo, al hacer un análisis detallado de cada uno de los conceptos disponibles, resulta interesante la existencia de ideas comunes entre los mismos, sin observarse planteamientos contradictorios, sino más bien complementarios. De aquí que la mayoría de los autores coinciden en que una arquitectura de software define la estructura del sistema. Esta estructura se constituye de componentes -módulos o piezas de código que nacen de la noción de abstracción, cumpliendo funciones específicas, e interactuando entre sí con un comportamiento definido (7); (8); (6).

Los componentes se organizan de acuerdo a ciertos criterios, que representan decisiones de diseño. En este sentido, existen autores que plantean que la arquitectura de software incluye justificaciones referentes a la organización y el tipo de componentes, garantizando que la configuración resultante satisface los requisitos del sistema (9).

En pocas palabras se puede definir a la arquitectura de software como la estructura del sistema en función de la definición de los componentes y sus interacciones (7).

Pudiera decirse entonces, que la arquitectura de software es el estudio de la estructura de un sistema, dividiendo las funciones entre los módulos del sistema en función de definir los medios de comunicación entre ellos, así como sus componentes e interacciones, garantizando que el sistema resultante satisface los requisitos.

1.2 Métodos de Evaluación de Arquitecturas de Software

Hasta hace poco no existían métodos de utilidad general para evaluar arquitecturas de software. Si alguno existía, sus enfoques eran incompletos, y no repetibles, lo que no brindaba mucha confianza (1).

En virtud de esto, múltiples métodos de evaluación han sido propuestos. A continuación, se explican algunos de los más importantes.

1.2.1 Software Architecture Analysis Method (SAAM)

El Método de Análisis de Arquitecturas de Software (*Software Architecture Analysis Method*, SAAM) es el primero que fue ampliamente promulgado y documentado. Fue originalmente creado para el análisis de la modificabilidad de una arquitectura, pero en la práctica ha demostrado ser muy útil para evaluar de forma rápida distintos atributos de calidad, tales como: modificabilidad o mantenibilidad, portabilidad, escalabilidad e integrabilidad (1).

Detallados a continuación:

- **Mantenibilidad**: La capacidad del producto software para ser modificado (10).
- **Portabilidad:** La capacidad del producto software para ser transferido de un entorno a otro (10).
- **Integrabilidad:** Es la medida de la habilidad de que un grupo de partes del sistema trabajen con otro sistema (11).
- **Escalabilidad:** Es el grado con el que se pueden ampliar el diseño arquitectónico, de datos o procedimental (12).

El método de evaluación SAAM se enfoca en la enumeración de un conjunto de escenarios que representan los cambios probables a los que estará sometido el sistema en el futuro. Como entrada principal, es necesaria alguna forma de descripción de la arquitectura a ser evaluada. Las salidas de la evaluación del método SAAM son: (1)

- Una proyección sobre la arquitectura de los escenarios que representan los cambios posibles ante los que puede estar expuesto el sistema.
- Entendimiento de la funcionalidad del sistema, e incluso una comparación de múltiples arquitecturas con respecto al nivel de funcionalidad que cada una soporta sin modificación.

Con la aplicación de este método, si el objetivo de la evaluación es una sola arquitectura, se obtienen los lugares en los que la misma puede fallar, en términos de los requisitos de modificabilidad. Para el caso en el que se cuenta con varias arquitecturas candidatas, el método produce una escala relativa

que permite observar qué opción satisface mejor los requisitos de calidad con la menor cantidad de modificaciones.

La Tabla 1 presenta los pasos que contempla el método de evaluación SAAM, con una breve descripción:

Tabla 1: Pasos del Método de Evaluación SAAM (1)

Pasos	Descripción
Desarrollo de escenarios	Un escenario es una breve descripción de usos anticipados o deseados del sistema. De igual forma, estos pueden incluir cambios a los que puede estar expuesto el sistema en el futuro.
Descripción de la arquitectura.	La arquitectura (o las candidatas) debe ser descrita haciendo uso de alguna notación arquitectónica que sea común a todas las partes involucradas en el análisis. Deben incluirse los componentes de datos y conexiones relevantes, así como la descripción del comportamiento general del sistema. El desarrollo de escenarios y la descripción de la arquitectura son usualmente llevados a cabo de forma intercalada, o a través de varias iteraciones.
Clasificación y asignación de prioridad de los escenarios.	La clasificación de los escenarios puede hacerse en dos clases: *directos* e *indirectos*. Un escenario *directo* es el que puede satisfacerse sin la necesidad de modificaciones en la arquitectura. Un escenario *indirecto* es aquel que requiere modificaciones en la arquitectura para poder satisfacerse. Los escenarios *indirectos* son de especial interés para SAAM, pues son los que permiten medir el grado en que una arquitectura puede ajustarse a los cambios de evolución que son importantes para los involucrados en el desarrollo.
Evaluación individual de los escenarios indirectos.	Para cada escenario *indirecto*, se listan los cambios necesarios sobre la arquitectura, y se calcula su costo. Una modificación sobre la arquitectura significa que debe introducirse un nuevo componente o conector, o que alguno de los existentes requiere cambios en su especificación.

Evaluación de la interacción entre escenarios.	Cuando dos o más escenarios *indirectos* proponen cambios sobre un mismo componente, se dice que interactúan sobre ese componente. Es necesario evaluar este hecho, puesto que la interacción de componentes semánticamente no relacionados revela que los componentes de la arquitectura efectúan funciones semánticamente distintas. De forma similar, puede verificarse si la arquitectura se encuentra documentada a un nivel correcto de descomposición estructural.
Creación de la evaluación global.	Debe asignársele un peso a cada escenario, en términos de su importancia relativa al éxito del sistema. La asignación de peso suele hacerse con base en las metas del negocio que cada escenario soporta. En el caso de la evaluación de múltiples arquitecturas, la asignación de pesos puede ser utilizada para la determinación de una escala general.

SAAM tiene como característica principal la realización de un análisis que delimita la forma en que variarán los atributos de calidad, como resultado de algunas modificaciones futuras de la arquitectura. Este elemento es fundamental pues, le da una visión arquitectónica al equipo de desarrollo, que conoce hasta qué punto puede variar la arquitectura sin que afecte el nivel requerido de los atributos de calidad. Sin embargo, el comportamiento de un atributo de calidad puede afectar el desempeño de otros, por lo que no solamente se debe tener en cuenta la estructura de los componentes, sino también las relaciones que se establecen entre los mismos. Es ahí donde este método, presenta su principal desventaja, ya que no valora la interrelación entre los distintos atributos.

1.2.2 Architecture Trade-off Analysis Method (ATAM)

El Método de Análisis de Acuerdos de Arquitectura (*Architecture Trade-off Analysis Method*, ATAM) está inspirado en tres áreas distintas: los *estilos arquitectónicos,* el *análisis de atributos de calidad* y el *método de evaluación SAAM*, explicado anteriormente. El nombre del método ATAM surge del hecho de que revela la forma en que una arquitectura específica satisface ciertos atributos de calidad, y provee una visión de cómo los atributos de calidad interactúan con otros. (1)

El método se concentra en la identificación de los estilos arquitectónicos o enfoques arquitectónicos utilizados. Propone el término enfoque arquitectónico dado que no todos los arquitectos están familiarizados con el lenguaje de estilos arquitectónicos, aun haciendo uso indirecto de estos.

De cualquier forma, estos elementos representan los medios empleados por la arquitectura para alcanzar los atributos de calidad, así como también permiten describir la forma en la que el sistema puede crecer, responder a cambios, e integrarse con otros sistemas, entre otros (1).

El método de evaluación ATAM comprende nueve pasos, agrupados en 4 fases. La Tabla 2 presenta las fases y sus pasos enumerados, junto a su descripción.

Tabla 2: Pasos del Método de Evaluación ATAM (1)

Fase 1: Presentación.	
1. Presentación del ATAM	El líder de evaluación describe el método a los participantes, trata de establecer las expectativas y responde las preguntas propuestas.
2. Presentación de las metas del negocio	Se realiza la descripción de las metas del negocio que motivan el esfuerzo, y aclara que se persiguen objetivos de tipo arquitectónico.
3. Presentación de la arquitectura	El arquitecto describe la arquitectura, enfocándose en cómo ésta cumple con los objetivos del negocio.
Fase 2: Investigación y análisis.	
4. Identificación de los enfoques arquitectónicos.	Estos elementos son detectados, pero no analizados.
5. Generación del árbol de utilidad.	Se solicitan los atributos de calidad que engloban la "*utilidad*" del sistema (*desempeño, disponibilidad, seguridad, modificabilidad, usabilidad, etc*.), especificados en forma de escenarios. Se anotan los estímulos y respuestas, así como se establece la prioridad entre ellos.
6. Análisis de los enfoques arquitectónicos.	Con base en los resultados del establecimiento de prioridades del paso anterior, se analizan los elementos del paso 4. En este paso se identifican riesgos arquitectónicos, puntos de sensibilidad y puntos de balance.
Fase 3: Pruebas	

7. Lluvia de ideas y establecimiento de prioridad de escenarios.	Con la colaboración de todos los involucrados, se complementa el conjunto de escenarios.
8. Análisis de los enfoques arquitectónicos	Este paso repite las actividades del paso 6, haciendo uso de los resultados del paso 7. Los escenarios son considerados como casos de prueba para confirmar el análisis realizado hasta el momento.
Fase 4: Reporte	
9. Presentación de los resultados	Basado en la información recolectada a lo largo de la evaluación del ATAM, se presentan los hallazgos a los participantes.

El método ATAM, centra su actividad de evaluación en la interacción entre los diferentes atributos de calidad arquitectónica. Este, al igual que el SAAM, basa sus evaluaciones sobre los escenarios desarrollados por los involucrados y un equipo de evaluación.

Los beneficios de la utilización de este método, es la organizada interacción que se establece entre los actores, arquitectos y equipo de evaluación, así como toda la documentación arquitectónica que genera el proceso de evaluación (13).

1.2.3 Active Reviews for Intermediate Designs (ARID)

El método Revisión Activa para el Diseño Intermedio (*Active Reviews for Intermediate Designs*, ARID) es conveniente para realizar la evaluación de diseños parciales en las etapas tempranas del desarrollo. En ocasiones, es necesario saber si un diseño propuesto es conveniente, desde el punto de vista de otras partes de la arquitectura. Según los autores, ARID es un híbrido entre Revisión Activa del Diseño (*Active Design Review,* ADR) *y* ATAM, descrito anteriormente (1) .

ADR es utilizado para la evaluación de diseños detallados de unidades de software como los componentes o módulos. Las preguntas giran en torno a la calidad y completitud de la documentación y la suficiencia, el ajuste y la conveniencia de los servicios que provee el diseño propuesto.

Se propone que tanto ADR como ATAM proveen características útiles para el problema de la evaluación de diseños preliminares, dado que ninguno por sí solo es conveniente. En el caso de ADR, los involucrados reciben documentación detallada y completan cuestionarios, cada uno por separado. En el caso de ATAM**,** está orientado a la evaluación de toda una arquitectura (1).

Tabla 3: Pasos del Método de Evaluación ARID (1)

Fase 1: Actividades Previas	
1. Identificación de los encargados de la revisión.	Los encargados de la revisión son los ingenieros de software que se espera que usen el diseño, y todos los involucrados en el diseño. En este punto, converge el concepto de encargado de revisión de ADR e involucrado del ATAM.
2. Preparar el informe de diseño	El diseñador prepara un informe que explica el diseño. Se incluyen ejemplos del uso del mismo para la resolución de problemas reales. Esto permite al facilitador anticipar el tipo de preguntas posibles, así como identificar áreas en las que la presentación puede ser mejorada.
3. Preparar los escenarios base	El diseñador y el facilitador preparan un conjunto de escenarios base. De forma similar a los escenarios del ATAM y el SAAM, se diseñan para ilustrar el concepto de escenario, que pueden o no ser utilizados para efectos de la evaluación.
4. Preparar los materiales	Se reproducen los materiales preparados para ser presentados en la segunda fase. Se establece la reunión, y los involucrados son invitados.
Fase 2: Revisión	
5. Presentación del **ARID**	Se explica los pasos del ARID a los participantes.
6. Presentación del diseño	El líder del equipo de diseño realiza una presentación, con ejemplos incluidos. Se propone evitar preguntas que conciernen a la implementación o argumentación, así como alternativas de diseño. El objetivo es verificar que el diseño es conveniente.
7. Lluvia de ideas y establecimiento de prioridad de escenarios	Se establece una sesión para la lluvia de ideas sobre los escenarios y el establecimiento de prioridad de escenarios. Los involucrados proponen escenarios a ser usados en el diseño para resolver problemas que esperan encontrar. Luego, los escenarios son sometidos

	a votación, y se utilizan los que resultan ganadores para hacer pruebas sobre el diseño.
8. Aplicación de los escenarios	Comenzando con el escenario que contó con más votos, el facilitador solicita el pseudo-código que utiliza el diseño para proveer el servicio, y el diseñador debe ayudar en esta tarea. Este paso continúa hasta que ocurra alguno de los siguientes eventos: • Se agota el tiempo destinado a la revisión. • El grupo se siente satisfecho con la conclusión alcanzada. Puede suceder que el diseño presentado sea conveniente, con la exitosa aplicación de los escenarios o, por el contrario, no conveniente, cuando el grupo encuentra problemas o deficiencias.
9. Resumen	Al final, el facilitador recuenta la lista de puntos tratados, pide opiniones de los participantes sobre la eficiencia del ejercicio de revisión, y agradece por su participación.

En resumen, el ARID, incorpora fuertes cualidades del método ADR y ATAM. Este método evalúa la idoneidad de los diseños arquitectónicos en las primeras fases de definición. Permite reunir las partes interesadas y los diseñadores en las primeras fases de desarrollo del software.

Los involucrados generan los escenarios para analizar la capacidad de la arquitectura, mientras que el uso de escenarios genera una lluvia de ideas por parte de los diseñadores, sobre nuevos diseños y evaluaciones. Luego de terminado el proceso de evaluación, se pasa a realizar las adecuaciones pertinentes en el diseño de la arquitectura.

Este método tiene como principal desventaja que sólo analiza los componentes de la arquitectura, sin tener en cuenta las conexiones que se establecen entre ellos. Unido a esto, está el inconveniente de que no contempla ningún atributo de calidad, sino que sólo evalúa la conveniencia del diseño arquitectónico.

1.3 Consideraciones sobre los Métodos de Evaluación de Arquitectura.

Los tres métodos descritos anteriormente tienen importantes propiedades, tales como:

- Son empleados para un alto nivel de análisis de toda arquitectura y no se centran en obtener más información.
- Realizan solamente el análisis cualitativo de los atributos de calidad. Indican lo que se debe mejorar en la arquitectura para la satisfacción de los atributos de calidad esperados.
- Son métodos bastante generales, por lo que es grande el esfuerzo que se realiza para su aplicación en una arquitectura en particular.
- Para sus evaluaciones, se basan en la técnica basada en escenarios.
- De manera independiente, no logran abarcar una evaluación integral de los atributos de calidad.

El método ATAM evalúa con más profundidad, en relación con otros métodos, cuestiones referentes a la arquitectura, como son: los atributos de calidad.

Hasta el momento se han presentado varios métodos de evaluación de arquitectura algunos más completos que otros, pero independientemente de esto todos tienen algo en común y es que utilizan la técnica de escenarios como vía de constatar en qué medida la arquitectura responde a los atributos de calidad requeridos por el sistema.

Tabla 4: Comparación entre los métodos de evaluación de Arquitectura ATAM, SAAM y ARID

	ATAM	SAAM	ARID
Atributos de calidad contemplados.	- Modificabilidad. - Seguridad. - Confiabilidad. - Desempeño.	- Modificabilidad - Funcionalidad	- Conveniencia del diseño evaluado.
Objetos analizados.	- Estilos arquitectónicos. - Documentación. - Flujos de Datos. - Vistas Arquitectónicas.	- Documentación - Vistas arquitectónicas	- Especificación de los componentes.
Etapas del proyecto en las que se aplica.	- Luego de que el diseño de la arquitectura ha sido establecido.	- Luego que la arquitectura cuenta con funcionalidades ubicadas por módulos.	- A lo largo del diseño de la arquitectura.
Enfoques utilizados.	- Árbol de utilidad y tormentas de ideas para articular los requisitos	- Tormenta de ideas para los escenarios y articular los requisitos de	- Revisiones del diseño, tormentas de ideas para

	de calidad. - Análisis arquitectónico que detecta puntos sensibles, puntos de balance y riesgos.	calidad. - Análisis de los escenarios para verificar funcionalidad o estimar el costo de los cambios.	obtener escenarios.

1.4 Otros Métodos de Evaluación de Arquitectura

Además de los métodos SAAM, ATAM y ARID antes vistos, existen otros, pero estos sólo se limitan a la evaluación de un atributo de calidad específico, por lo que su uso se ve limitado.

1.4.1 Architecture Level Modifiability Analysis (ALMA)

El método Análisis del Nivel de Modificabilidad de la Arquitectura (*Architecture Level Modifiability Analysis*, ALMA) es el resultado de los trabajos de investigación de *Brengtsson* y *Lassing.* El atributo de calidad que analiza este método es la facilidad de modificación. Esto se refiere a la capacidad de un sistema para ser ajustado debido a cambios en los requisitos, o en el entorno, así como la adición de nuevas funcionalidades (14).

ALMA es un método de evaluación orientado a metas, que se apoya en el uso de escenarios de cambio, los cuales escriben los eventos posibles que provocarían cambios al sistema, y como se llevarían a cabo estos. El mismo consta de cinco pasos como se muestra en figura 1 (14).

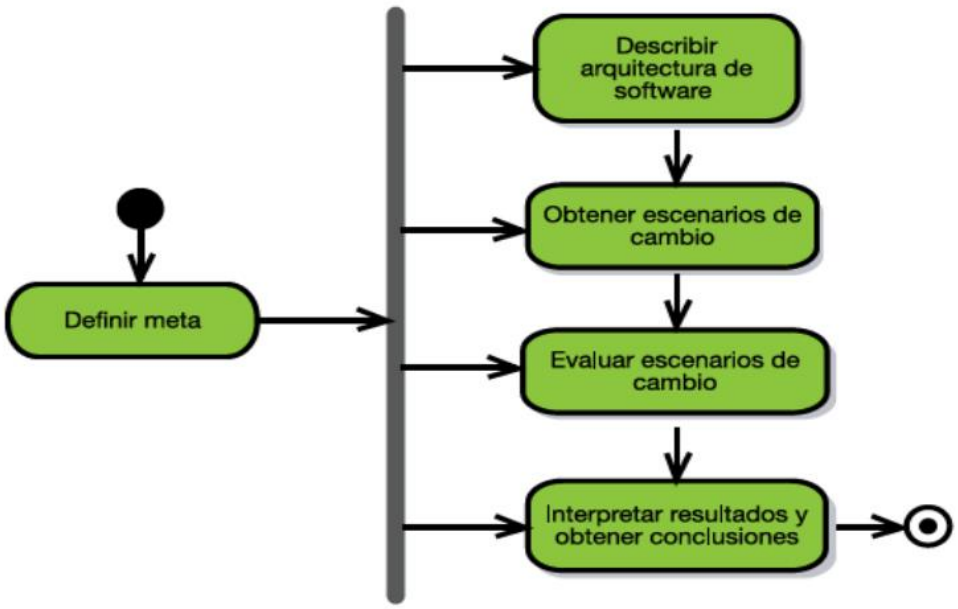


Fig. 1: Método ALMA (14)**.**

1.4.2 Performance Assessment of Software Architecture (PASA)

El atributo de calidad que analiza el método Valoración del Desempeño de la Arquitectura de Software (*Performance Assessment of Software Architecture*, PASA) es el desempeño. Se interesa en saber qué tanto tiempo le toma al sistema de software responder cuando uno o varios eventos ocurren, así como determinar el número de eventos procesados en un intervalo de tiempo dado. PASA es el resultado del trabajo de Williams y Smith, el mismo utiliza diversas técnicas de evaluación, tales como la aplicación de estilos arquitectónicos, anti-patrones, guías de diseño y modelos (14).

Este método también se basa en escenarios y puede aplicarse de forma temprana o tardía. Los escenarios generados en este método sirven como punto de partida para la construcción de modelos de desempeño (14).

Las personas involucradas durante el proceso de evaluación son: el arquitecto, equipo de desarrollo y en algunos momentos los gerentes del proyecto. La evaluación empleando este procedimiento puede durar una semana si se trabaja de forma intensiva.

Es considerado un método de evaluación maduro ya que ha sido probado en varios dominios de aplicación como sistemas web, aplicaciones financieras y sistemas en tiempo real (14).

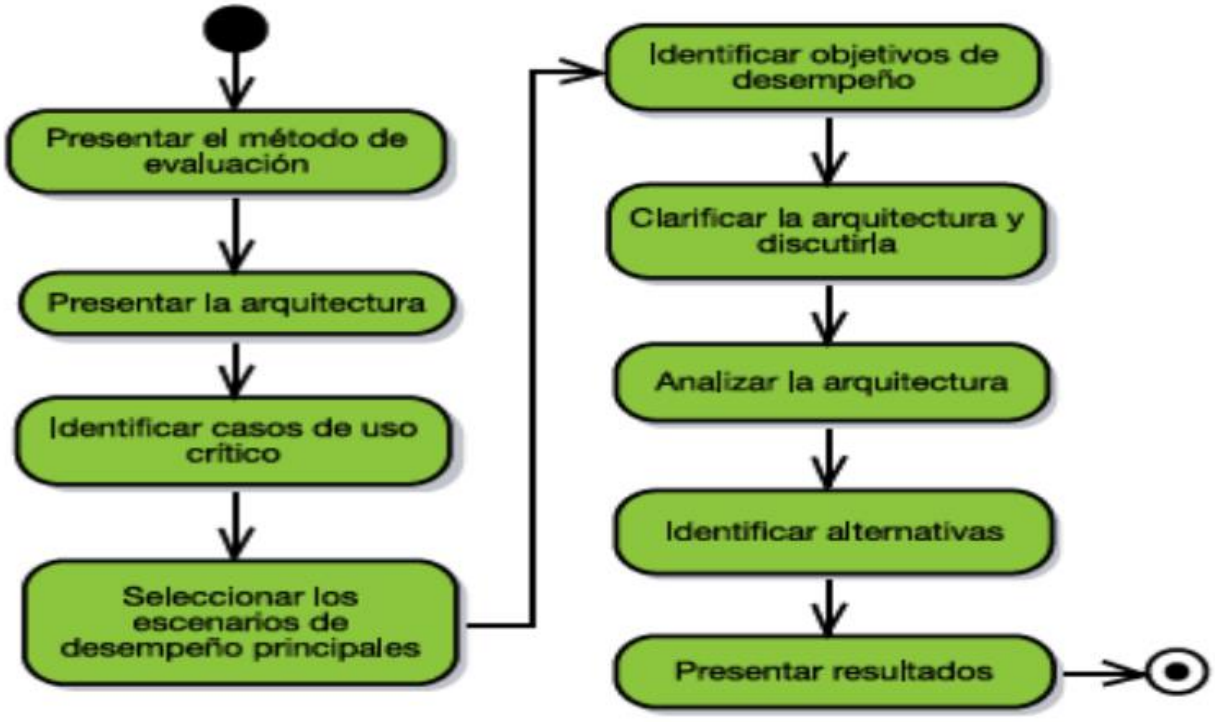


Fig. 2: Método PASA (14)**.**

1.4.3 Scenario Based Architecture Level Usability Analysis (SALUTA)

El método Análisis del Nivel de Usabilidad de la Arquitectura Basado en Escenario (*Scenario Based Architecture Level Usability Analysis*, SALUTA). Es el primer método desarrollado para evaluar arquitecturas desde la perspectiva de la facilidad de uso del sistema, siendo el resultado de los estudios de Folmer y Gurp (14).

Este método hace uso de marcos de referencia que expresan las relaciones que existen entre facilidad de uso y arquitectura de software. Dichos marcos de referencias incluyen un conjunto integrado de soluciones de diseño como patrones de diseños o propiedades que tienen un efecto positivo sobre la facilidad de uso en un sistema de software (14).

SALUTA analiza cuatro atributos que están directamente relacionados con la facilidad de uso de un sistema de software: facilidad de aprendizaje, eficiencia de uso, confiabilidad y satisfacción. El mismo se basa al igual que los dos métodos analizados anteriormente en escenarios, que en este caso, son escenarios de uso que agrupan uno o más perfiles de uso valga la redundancia, donde cada uno representa la facilidad de uso requerida por el sistema (14).

Se recomienda utilizarlo una vez que se ha especificado la arquitectura, pero antes de implementar (14). El mismo consta de cuatro pasos como se muestra a continuación:

Fig. 3: Método SALUTA (14)**.**

Las personas involucradas durante el proceso de evaluación son: el arquitecto de software, ingenieros de requisitos o ingenieros responsables por la facilidad de uso (14).

1.4.4 Survivable Network Analysis (SNA)

El método Análisis de la Supervivencia en la Red (*Survivable Network Analysis*, SNA) fue desarrollado por el Equipo Responsable de Emergencias Informáticas (*Computer Emergency Response Team,* CERT) que forma parte del Instituto de Ingeniería de Software (*Software Engineering Institute* SEI). Este método ayuda a identificar la capacidad de supervivencia de un sistema, analizando su arquitectura.

La supervivencia es la capacidad que tiene un sistema para completar su misión a tiempo, ante la presencia de ataques, fallas o accidentes. Para evaluar esta supervivencia SNA utiliza tres propiedades claves: Resistencia, Reconocimiento y Recuperación (14).

Este procedimiento puede ser realizado después de la especificación de la arquitectura, durante la implementación de esta, o posteriormente.

SNA se basa en la técnica de escenarios de uso, e identifica dos tipos de escenarios (14).

1. ***Escenarios normales***, que se componen de una serie de pasos donde los usuarios invocan servicios y obtienen acceso a activos, tales como bases de datos.
2. ***Escenarios de intrusión***, en los que se representan diferentes tipos de ataques al sistema.

Las personas involucradas durante la realización del método son: el arquitecto de software, el diseñador principal, los propietarios del sistema y usuarios del mismo.

Como resultado de esta evaluación se obtienen:

- Un documento que recoge todas las modificaciones y recomendaciones a la arquitectura, acompañadas del mapa de supervivencia. (14).

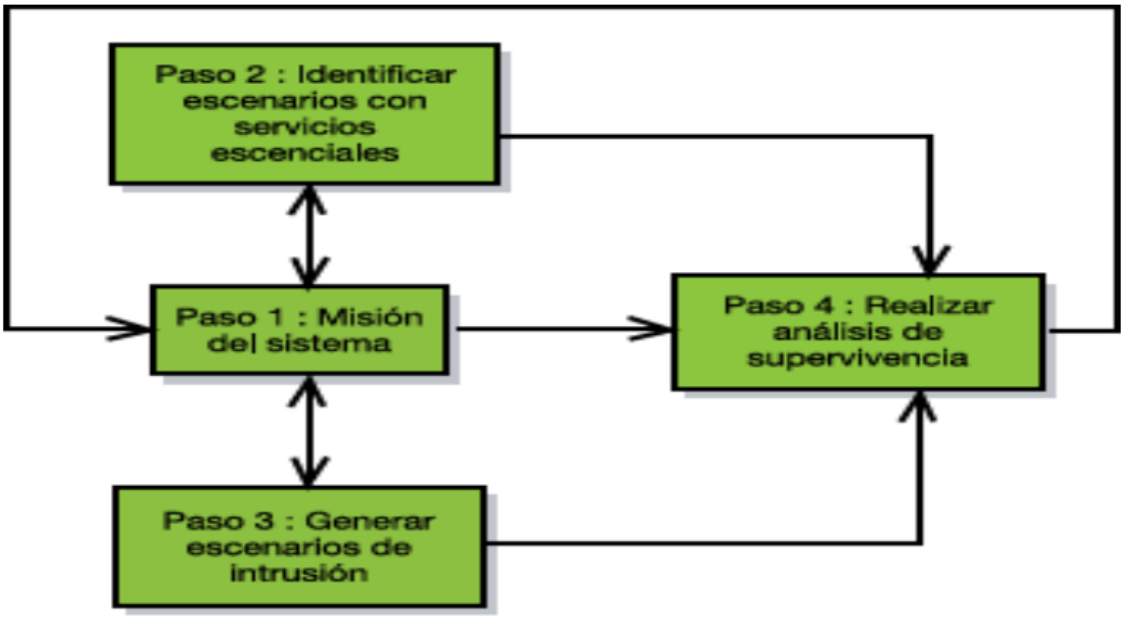


Fig. 4: Método SNA (14)**.**

La tabla #5 presenta a manera de resumen, un cuadro comparativo entre estos métodos analizados anteriormente.

Tabla 5: Comparación entre los métodos ALMA, PASA, SALUTA y SNA

	ALMA	PASA	SALUTA	SNA
Meta	Predecir el costo de mantenimiento, evaluar riesgos, comparación entre arquitecturas.	Analizar la arquitectura con respecto a los objetivos de desempeño de un sistema.	Predecir la factibilidad de uso en un sistema analizando la arquitectura	Identificar la capacidad de supervivencia en un sistema ante la presencia de ataques, fallas o accidentes.
Atributo de Calidad	Facilidad de Modificación	Desempeño	Facilidad de uso	Supervivencia

Técnica de Evaluación	Escenarios de Cambio	Escenarios	Escenarios de uso	Escenarios de uso normales, escenarios de instrucción.
Entradas	Especificación de la arquitectura, requisitos no funcionales.	Especificación de la arquitectura	Especificación de la arquitectura, requisitos no funcionales relacionados con la facilidad de uso.	Especificaciones de la arquitectura.
Salidas	Dependiendo de la meta de evaluación se generan los resultados.	Hallazgos encontrados, pasos específicos a seguir y recomendaciones.	Grado de facilidad de uso que soporta la arquitectura evaluada.	Modificaciones recomendadas a la arquitectura y mapa de supervivencia.
Personas Involucradas	Arquitecto y equipo de desarrollo	Arquitecto, equipo de desarrollo y administradores del proyecto.	Arquitectos, ingenieros de requisitos o ingenieros responsables por la facilidad de uso.	Arquitecto, diseñador principal, propietarios del sistema, usuarios.
Duración	No Especificado	7 días	No Especificado	No Especificado
Validación del Método	Sistema de control embebido, sistemas médicos, telecomunicaciones, sistemas administrativos.	Sistemas basados en Web, aplicaciones financieras y sistemas en tiempo real.	Algunos casos de estudio que incluyen principalmente sistemas Web.	Sistemas comerciales y de gobierno.

1.5 Caracterización de la Arquitectura ABOS

La arquitectura ABOS (Arquitectura Base Orientada a Servicios) es una arquitectura en capas, formada por componentes. Utiliza la especificación de OSGI 4.3 (15). Utiliza también Equinox 3.8 una implementación de OSGI 4.3 que puede ser utilizado como framework base donde desplegar aplicaciones modulares compuestas por bundles. Equinox y OSGI son tecnologías libres basadas fundamentalmente en Eclipse.

En la Figura 5 se ejemplifica la arquitectura ABOS, donde cada capa está muy bien delimitada de las demás. Una capa superior interactúa con una capa inferior mediante interfaces que definen las funcionalidades que la misma debe brindar. Las capas de una aplicación pueden residir tanto en el nodo físico como en nodos diferentes, pero en el caso de ABOS todas las capas que se exponen a continuación están en el nodo físico porque están obligadas a correr sobre la misma máquina virtual. En cada capa se especifica las tecnologías que conforman las mismas para su funcionamiento.

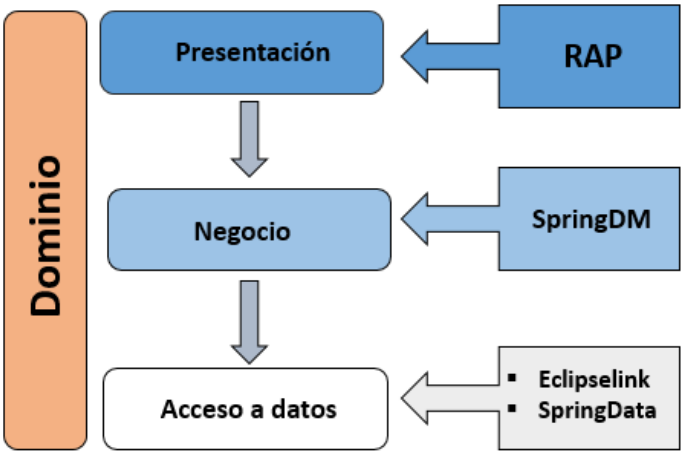


Fig. 5: Diseño de la arquitectura en ABOS

- **Capa de acceso a datos:** Es donde residen los datos de la aplicación y es la encargada de acceder a los mismos. Utiliza el gestor de base dato PostgresSQL para base datos relacionales y JISIS para base datos documentales. En esta capa son utilizados los componentes Eclipselink y SpringData.
- **Capa de negocio:** Es la capa intermedia donde residen los programas que se ejecutan, se reciben las peticiones del usuario, aquí es donde se establecen todas las reglas que deben cumplirse, recibe los datos de la capa de presentación y se comunica con la capa de acceso a datos para solicitar al gestor de base datos almacenar o recuperar los datos. Encapsula en su implementación las dependencias de las tecnologías de acceso a datos para que la capa de

presentación no tenga dependencias de las mismas. Esta capa esta implementada en código java y Spring DM.

- **Capa de presentación:** Esta capa esta implementada en RAP (Plataforma de aplicaciones remotas) que implementa como API para el diseño de interfaces usuario a SWT (Standard Widget Toolkit). Presenta la aplicación al usuario. Comunica y captura la información del usuario en un mínimo proceso. Esta consume los bundles de negocio.
- **Capa de Dominio:** Es la capa donde residen las entidades del dominio de la aplicación.

1.5.1 OSGI

Open Service Gateway Iniciative (OSGI) proporciona un marco de trabajo java de uso general, seguro y administrado que soporta el despliegue dinámico de aplicaciones conocidas como "Bundles" o módulos (15). Actualmente se utiliza en ABOS la versión 4.3 (16).

Algunas de las características que componen este marco de trabajo (17):

- Es un sistema de módulos para la plataforma java.
- Incluye reglas de visibilidad, gestión de dependencias y versionado de los bundles.
- Es dinámico.
- La instalación, arranque, parada, actualización y desinstalación de bundles se realiza dinámicamente en tiempo de ejecución sin tener que detener por completo la plataforma.
- Se trata de una arquitectura orientada a servicios.
- Los servicios se pueden registrar y consumir dentro de la máquina virtual.

Se pueden destacar algunos de los principales beneficios que proporciona esta tecnología (17):

- Simplifica los proyectos en los que participan muchos desarrolladores en diferentes equipos.
- Se puede utilizar en sistemas más pequeños.
- Gestiona los despliegues locales o remotos.
- Se trata de una herramienta fácilmente ampliable.
- No es una tecnología cerrada.
- Gran aceptación. Es una tecnología usada por muchas empresas fabricantes.

Capas de OSGI.

La funcionalidad del framework se divide en las siguientes capas (15):

- **Security Layer**: Capa de seguridad.
- **Module Layer**: Define el modelo de modularización, Esta capa define las reglas para el intercambio de paquetes java entre los bundles.

- **Life Cycle Layer**: Gestiona el ciclo de vida de un bundle dentro del framework, sin tener que detener la Máquina Virtual (VM).
- **Service Layer**: La capa de servicios proporciona un modelo programación dinámico para los desarrolladores de bundles, simplificando el desarrollo y despliegue de módulos a través del desacople de la especificación del servicio (java interface), de su implementación.
- **Execution Environment**: Entorno de ejecución OSGI.

Se pueden ver las diferentes capas de un sistema OSGI en la figura 6:

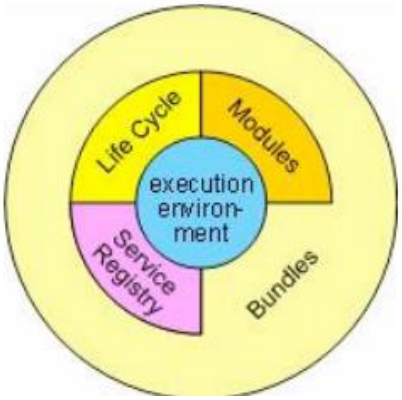


Fig. 6: Capas de Sistema OSGI (15)**.**

1.5.2 Bundle

Un bundle es una aplicación empaquetada en un fichero jar, que se despliega en una plataforma OSGI. Cada bundle contiene un fichero de metadatos organizado por pares de nombre clave: valor donde se describen las relaciones del bundle con el mundo exterior a él (17). (Por ejemplo, paquetes que importa o paquetes que exporta). Este fichero viene dado con el nombre MANIFEST.MF y se encuentra ubicado en el directorio META-INF.

A continuación, se muestra el fichero MANIFEST.MF del bundle cu.uci.abos.api:

Tabla 6: Ejemplo de archivo MANIFEST.MF

Manifest-Version: 1.0
Bundle-Manifest-Version: 2
Bundle-Name: CIGED Interface Api
Bundle-Description: API for manage modules contributors

Bundle-SymbolicName: cu.uci.abos.api
Bundle-Version: 2.0.0
Bundle-Vendor: UCI
Export-Package: cu.uci.abos.api. exception; *version*="2.0.0", cu.uci.abos.api.log; *version*="2.0.0", cu.uci.abos.api.ui; *version*="2.0.0", cu.uci.abos.api. util; *version*="2.0.0"
Import-Package: org. osgi. framework; *version*="[1.7.0,1.7.0]", org. osgi. service. component; *version*="[1.2.0,1.2.0]", org.osgi.util.tracker;*version*="[1.5.1,1.5.1]"

En la tabla de metadatos anterior, se puede diferenciar claves bastante intuitivas, donde se describen características del bundle, como, por ejemplo, el nombre del bundle, la versión, paquetes que importa o exporta, etc. Un bundle puede exportar sus paquetes, para que puedan ser usados por otro bundle. Además de eso, el bundle podrá establecer que versión del paquete desea exponer (17).

1.5.3 Eclipselink

Es una implementación libre de un ORM (Mapeador Objeto-Relacional), que permite la transformación de las tablas de una base de datos, en una serie de entidades que simplifiquen las tareas básicas de acceso a los datos para el programador. Gestiona todas las conversiones de entidades de dominio en tablas de la base datos (18) .ABCD utiliza la versión **org.eclipse.persistence.jpa_2.6.0** (19)**.**

Eclipselink permite:

- Persistencia de objetos Java a prácticamente cualquier base de datos relacional.
- Realizar las conversiones en la memoria entre los objetos de Java.
- Mapear cualquier modelo de objetos para cualquier esquema relacional.

1.5.4 SpringData

Es un componente del framework Spring que hace fácil el uso de tecnologías de acceso a datos sobre bases de datos relacionales, no relacionales y servicios de datos basados en la nube. Se trata de un proyecto global que contiene muchos sub proyectos que son específicos de una base de datos, es decir

te permite no tener que implementar el código de acceso a dato sobre cada entidad (20). ABOS utiliza **spring-data-jpa-1.2.1** a continuación se muestra un ejemplo de la implementación del código de acceso a dato del módulo **circulation.**

```
package cu.uci.abcd.dao.circulation;


import java.util.List;

import org.springframework.data.jpa.repository.JpaSpecificationExecutor;
import org.springframework.data.repository.PagingAndSortingRepository;

import cu.uci.abcd.domain.circulation.Reservation;
import cu.uci.abcd.domain.common.Nomenclator;

public interface ReservationDAO extends PagingAndSortingRepository<Reservation, Long>, JpaSpecificationExecutor<Reservation> {



}
```

Fig. 7: Implementación del código de acceso a dato con SpringData.

1.5.5 SpringDM

Permite escribir aplicaciones de Spring en un entorno de ejecución de OSGi, puede tomar ventaja de los servicios que ofrece el marco OSGi ya que puede consumir y exportar el servicio. En la capa de acceso a datos se utiliza SpringDM para exportar los modelos como servicio los cuales se consumen en la capa de negocio. La capa de negocio exporta la fachada de negocio como servicio, dicho servicio es consumido por la capa de presentación (20).Actualmente se en ABOS se utiliza la version 3.1.0. A continuación, se muestran imágenes que evidencias como se configura el consumo y exportación un servicio utilizando SpringDM en el bundle **cu.uci.abcd.acquisition.ui.**

```
<reference id="reportGeneratorExcel" interface="cu.uci.abcd.management.report.SpreadsheetGenerator">
    <listener bind-method="bindSpreadsheetGenerator" ref="allControllerManagement" />
</reference>
```

Fig. 8: Consumo de un servicio utilizando SpringDM.

```
<service ref="consultarRegistroDeAdquisiciones" interface="cu.uci.abos.ui.api.IContributorFactory">
    <service-properties>
            <beans:entry key="type" value="abcd"/>
            <beans:entry key="class" value="cu.uci.abcd.acquisition.ui.ConsultarRegistroDeAdquisiciones"/>
            <beans:entry key="viewController" value-ref="allControllerManagement"/>
    </service-properties>
</service>
```

Fig. 9: Exportación de un servicio utilizando SpringDM.

1.5.6 Virgo

Es un servidor de código libre basado en OSGI, mantenido actualmente por la Fundación Eclipse. Es compatible con el despliegue de paquetes o bundles OSGi, aplicaciones web en código java y aplicaciones Spring. Virgo permite crear un plan donde se configuran los bundles que se van a desplegar en el momento. El servidor tiene en su interior una carpeta llamada **pickup** en donde se configuran o cambian las dependencias de los bundles que se agregaron al plan (21). Actualmente ABCD utiliza la versión 3.6.3.

En la Fig. 10 se muestra un fragmento del archivo **org.eclipse.abcd.plan** donde se configura el plan del Virgo.

```
<plan name="org.eclipse.abcd" version="1.0.0" scoped="false" atomic="false"
                xmlns="http://www.eclipse.org/virgo/schema/plan"
                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                xsi:schemaLocation="
                    http://www.eclipse.org/virgo/schema/plan
                    http://www.eclipse.org/virgo/schema/plan/eclipse-virgo-plan.xsd">
    <!--Configuration artifact-->
    <artifact type="configuration" name="db.configuration" version="0"/>
    <artifact type="configuration" name="jisis.configuration" version="0"/>
    <artifact type="configuration" name="security.configuration" version="0"/>
    <artifact type="configuration" name="email.config" version="0"/>

    <!--ABCD core bundles-->
    <artifact type="bundle" name="cu.uci.abcd.dataprovider.jisis" version="2.0.0"/>
        <artifact type="bundle" name="cu.uci.abos.reports" version="2.0.0"/>
    <artifact type="bundle" name="cu.uci.abos.report.pdf" version="1.0.0"/>
    <artifact type="bundle" name="cu.uci.abos.report.xls" version="1.0.0"/>
    <artifact type="bundle" name="cu.uci.abos.core" version="2.0.0"/>

</plan>
```

Fig. 10: Configuración del plan.

1.5.7 SWT

SWT (Standard Widget Toolkit) se describe como un conjunto de widgets y bibliotecas gráficas para la creación de interfaces de usuario. Se integran con un sistema de ventanas nativos, pero con una API (Application program interface) independiente del sistema operativo (22).

Como se puede ver en la figura 11, SWT está constituido por tres componentes básicos: una biblioteca nativa que se comunica con el sistema operativo; una clase de visualización que actúa como una interfaz a través del cual SWT se comunica con la plataforma de interfaz gráfica de usuario; y una clase de Shell, que actúa como la ventana de nivel superior de la aplicación, que puede contener widgets o composites (otros componentes para el diseño).

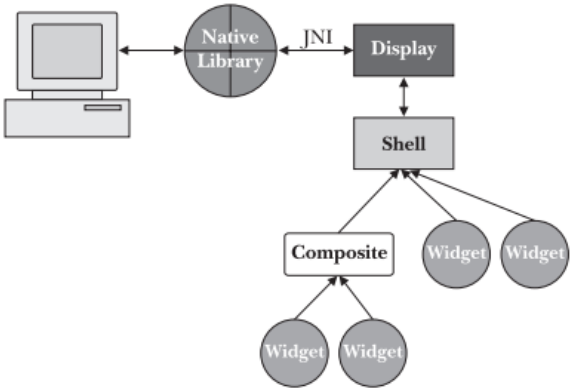


Fig. 11: Relación de SWT con el sistema operativo (23)**.**

Un **widget** es un elemento de una interfaz gráfica de usuario (GUI) que muestra información o proporciona una forma específica para un usuario interactuar con el sistema operativo o una aplicación (23).A continuación se muestra la jerarquía de clases que tienen como padre la clase widget.

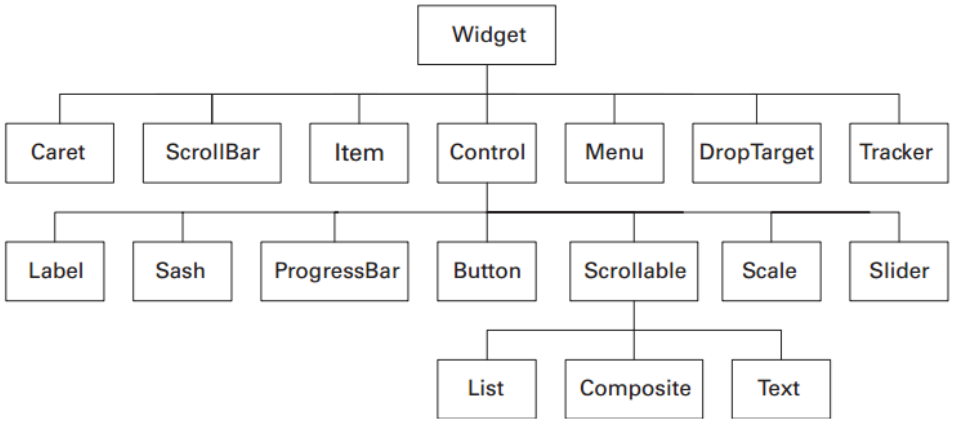


Fig. 12: Jerarquía de la clase Widget (23)**.**

Shell es una ventana que está gestionada por el gestor de ventanas del sistema operativo. Los shell pueden ser de nivel superior o shell secundarios (23).

Composite no es más que un widget que permite organizar en su interior widgets, permitiendo la modificación de su tamaño y posición (23).

Layout suele utilizarse para nombrar al esquema de distribución de los elementos dentro un diseño (23).

1.5.8 RAP

La plataforma de aplicaciones remota (RAP) es un potente marco de trabajo java utilizado para el desarrollo de aplicaciones empresariales modulares. Se integra bien con tecnologías probadas como OSGI y *Java Enterprise Edition* (JEE). RAP se ejecuta en la gran mayoría de los navegadores web, sin la necesidad de complementos necesarios. Está construido en un protocolo abierto, por lo que otros clientes también se pueden conectar. Con RAP no se crean interfaces de usuario con HTML sino con **Java SWT** (Standard Widget Toolkit) que es una interfaz de programación de aplicaciones (API) por sus siglas en ingles (24).

Navegadores Web que soportan el cliente web de RAP:

- Internet Explorer 9+
- Google Chrome 29+
- Firefox 23+
- Safari 6 +
- Opera 15 +
- iOs 6 +
- Android 4

Conclusiones parciales

En este capítulo se realizó una investigación sobre la evolución de los conceptos de arquitectura de software. Se profundizó en el estudio de las diferentes vertientes existentes para la evaluación arquitecturas. Se decidió elegir los Métodos de Kazman como vertiente para evaluar la arquitectura ABOS, estableciendo una comparación entre los mismos. Se realizó una caracterización de la

arquitectura ABOS haciendo énfasis en la función de cada una de sus capas y tecnologías que las componen.

CAPÍTULO 2: EVALUACIÓN DE LA ARQUITECTURA ABOS DEL PROYECTO ABCD

Este capítulo desarrolla la evaluación de la arquitectura ABOS mediante el método SAAM. Dicho método hace gran énfasis en evaluar la modificabilidad de la arquitectura en todo su alcance. Luego de la evaluación se detectan y solucionan los problemas de la arquitectura.

Introducción:

Evaluar una arquitectura, consiste en evaluar el potencial de la arquitectura diseñada, para lograr altos niveles de rendimiento del software, o para cumplir restricciones de hardware, software o de interfaz gráfica, exigidas por los clientes o un mercado específico. Además, mediante la arquitectura de software es posible determinar la estructura del proyecto de desarrollo del sistema, sobre elementos como el control de configuración, calendarios, control de recursos, metas de desempeño, estructura del equipo de desarrollo y otras actividades que se realizan con la arquitectura del sistema como apoyo principal. En este sentido, la garantía de una arquitectura correcta, cumple un papel fundamental en el éxito general del proceso de desarrollo y la calidad del mismo (1).

¿Por qué SAAM?

El método SAAM es fácil de aprender y de llevar a cabo con cantidades pequeñas de información y de preparación. Es un buen lugar para empezar si usted nunca ha hecho una evaluación de arquitectura antes, y sobre todo si lo que preocupa es la modificabilidad de su arquitectura (1).

Además de la síntesis de beneficios técnicos del método, SAAM logra que un amplio grupo de partes interesadas se unan por primera vez para discutir la arquitectura. Ya que la arquitectura es un vehículo de comunicación para ellos, un lenguaje compartido que permite a discutir sus preocupaciones en un mutuo lenguaje comprensible (1).

En resumen, se puede afirmar que, al evaluar una arquitectura, SAAM indica los puntos de fortalezas y debilidades, junto con los puntos de la arquitectura que no cumple con los requisitos de modificabilidad.

Fortalezas del SAAM: (25)**.**

- Aumento de la comprensión por los involucrados de la arquitectura que está siendo analizada.
- Luego de una sesión SAAM, la documentación de la arquitectura es refinada.
- Mejora la comunicación entre las partes interesadas.

Con respecto al análisis del atributo de modificabilidad sus fortalezas viene dadas por: (25)

- Se realiza un análisis de los escenarios desde la perspectiva arquitectónica de futuros cambios en el sistema.

- Se identifican zonas con un alto nivel de complejidad.
- Se estiman los costos y esfuerzos para realizar cambios necesarios.
- Basándose en los análisis de efectos secundarios ante los distintos cambios, es posible la creación de planes de trabajo para el desarrollo futuro del sistema.

Desarrollo de escenarios

Un escenario debe ilustrar las actividades que el sistema debe soportar, por lo que se debe anticipar los movimientos de los usuarios. El desarrollo de escenarios es crucial para capturar todos los usos principales del sistema. Los escenarios representan las tareas relevantes de diferentes roles como son: usuario final, cliente, administrador del sistema y el programador.

El grupo de trabajo que se conformó para la selección de los escenarios está conformado por los Arquitectos del proyecto ABCD y un programador del mismo.

La Tabla 5 representa una selección de los escenarios propuestos por el grupo de trabajo:

Tabla 7: Escenarios seleccionados

Número del escenario	Descripción del escenario
1	Cambio en la resolución de pantalla.
2	Detectar el bundle que ocasionó fallo en el sistema.
3	Cambiarle el idioma al sistema.
4	Mostrar información tabulada.
5	Gestión del versionado de los bundles.
6	Adaptar la arquitectura a otro proyecto.
7	Revisión del código de la capa de presentación.
8	Generar un reporte con imágenes .
9	Mostrar mensajes de información al usuario.
10	Ir a la página anterior

11	Autenticación del sistema contra LDAP
12	Validación de datos relacionados

Clasificación de los escenarios

En este punto se clasifican los escenarios en directos e indirectos. Los escenarios directos son los que están relacionados con el cumplimiento de uno o más atributos de calidad. Los escenarios indirectos es la secuencia de eventos para lograr que la arquitectura sufra los menores cambios posibles durante el desarrollo del sistema. La priorización de los escenarios se basa en un proceso de votación. Como SAAM aborda fundamentalmente la evaluación de la modificabilidad del sistema, el resultado de la votación lo constituyen un conjunto de escenarios indirectos con mayor probabilidad de efectuarse (1).

El proceso de votación para la prioridad de los escenarios se estableció con un peso entre 1 y10 puntos. Este paso tiene vital importancia ya que tiene como resultado los escenarios de más relevancia para el negocio.

La tabla 8 representa los escenarios seleccionados clasificados en directos e indirectos:

Tabla 8: Clasificación de escenarios

Número del escenario	Descripción del escenario	Clasificación Directo / Indirecto
1	Cambio en la resolución de pantalla.	indirecto
2	Detectar el bundle que ocasionó fallo en el sistema.	indirecto
3	Cambiarle el idioma al sistema.	directo
4	Mostrar información tabulada.	indirecto
5	Gestión del versionado de los bundles.	directo
6	Adaptar la arquitectura a otro proyecto.	indirecto
7	Revisión del código de la capa de presentación.	indirecto
8	Generar un reporte con imágenes .	indirecto

9	Mostrar mensajes de información al usuario.	indirecto
10	Ir a la página anterior	indirecto
11	Autenticación del sistema contra LDAP	indirecto
12	Validación de datos relacionados	indirecto

Evaluación de los escenarios indirectos

En el caso de un escenario directo el arquitecto demuestra cómo el escenario sería ejecutado por la arquitectura. En el caso de un escenario indirecto el arquitecto describe la forma en que la arquitectura debe ser modificada para lograr el escenario. En cada uno de estos escenarios indirectos se debe haber identificado las modificaciones arquitectónicas necesarias para facilitar el escenario, junto con los involucrados, los nuevos componentes del sistema, el costo estimado y el esfuerzo para aplicar la modificación (25).

Tabla 9: Evaluación global de los escenarios

Número del escenario	Descripción del escenario	Clasificación Directo / Indirecto	Peso del escenario	Cambios requeridos	Tiempo y personal necesitado
1	Cambio en la resolución de pantalla.	indirecto	10	Independizar los estilos de la arquitectura	2 personas/1-Mes
2	Detectar el bundle que ocasionó fallo en el sistema.	indirecto	8	Reestructuración de los bundles de la arquitectura	1-Persona/1-Semana
4	Mostrar información tabulada.	indirecto	8	Eliminar clases wrappers de las clases del dominio y usar reflection.	1-Persona/1-Semana

6	Adaptar la arquitectura a otro proyecto.	indirecto	10	Eliminar la dependencia del dominio de ABCD de la arquitectura	1- Persona/15 días
7	Revisión del código de la capa de presentación.	indirecto	7	Eliminar las especificaciones de la capa de presentación.	1- Persona/2- días
8	Generar un reporte con imágenes .	indirecto	9	Añadir componente de reportes.	1- Persona/15 días
9	Mostrar mensajes de información al usuario.	indirecto	5	Estandarizar los mensajes del proyecto y sustituir los diálogos que son de información por mensajes que se oculten automáticamente.	1-Persona/1- Semana
10	Ir a la página anterior	indirecto	6	No esta soportado	No esta soportado
11	Autenticación del sistema contra LDAP	indirecto	9	Implementar autenticación contra dominio	1-Persona/1- Semana
12	Validación de datos relacionados	indirecto	9	Agrupar los controles visuales por grupo de validación	1-Persona/1- Semana

Soluciones propuestas

En este epígrafe se exponen propuestas de solución implementadas para los escenarios más importantes.

Gestión del versionado de los bundles

En el desarrollo del proyecto no existía una correcta política para la versión de los bundles, lo que dificultaba el control de cambio, puesto que las modificaciones no llegaban a todos los módulos con el nivel requerido y con la misma fiabilidad.

Existían dos servidores, el ABCD-Work donde se almacenaba el compilado de la arquitectura en desarrollo de ABCD, el ABCD-Production donde se hacia el compilado final del proyecto y la carpeta **src** donde se almacenaba el código fuente de los módulos del sistema. El proceso de compilación funcionaba de la siguiente manera. Se realizaba un compilado con el código fuente de los módulos y la arquitectura, el resultado se subía a ABCD-Production. El compilado resultante era inestable porque en el proceso no permitía que cada desarrollador verificara el funcionamiento de su módulo integrado a todos los demás. En la Fig. 13 se muestra como ocurría el proceso compilación del proyecto.

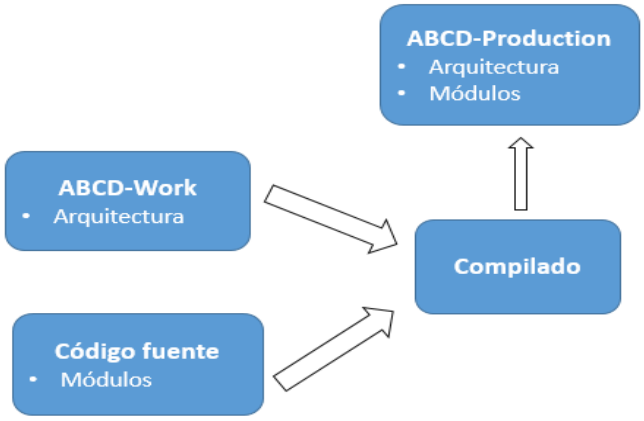
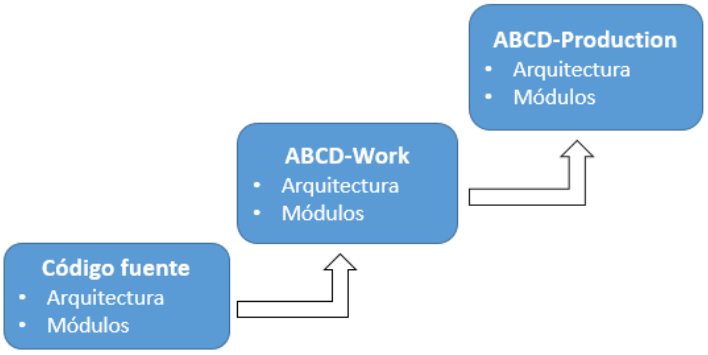


Fig. 13: Como ocurría la compilación del proyecto antes de la solución.

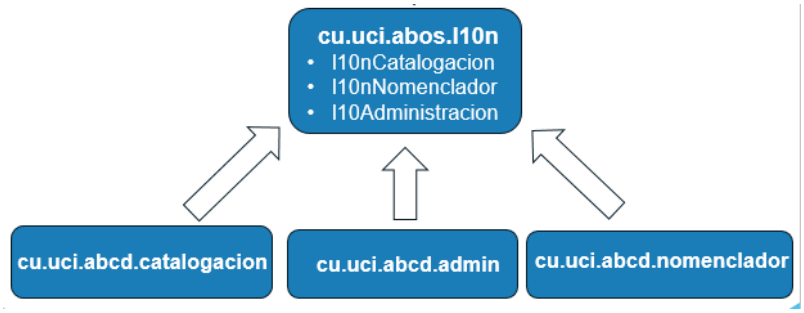
Como parte de la solución se decidió utilizar el servidor ABCD-Work como un servidor de prueba subiendo el código fuente de los módulos y la arquitectura, con el objetivo de realizar un compilado de prueba y verificar que los bundles se encontraban en su versión correcta. Luego se hace el compilado final en el servidor ABCD-Production. En la Fig.14 se muestra como quedo estructurada la solución.



. **Fig. 14: Como ocurre la compilación actual del proyecto.**

Internacionalización del sistema

La arquitectura mediante el bundle **cu.uci.core.l10n** era la encargada de manejar el proceso de internacionalización de todos los módulos de ABCD. Este era un componente inestable durante el desarrollo, cada vez que era necesario cambiar la internacionalización de un módulo, había modificar el componente y actualizarlo para que funcionara para todos. Acumulaba grandes cantidades de información en un solo bundle de la arquitectura lo que dificultaba la internacionalización individual de cada módulo. En la Fig. 15 se muestra como estaba estructurada la internacionalización.



. **Fig. 15: Estructura de la internacionalización de ABCD antes de la solución.**

La solución aplicada fue la siguiente:

Se dejó en la arquitectura solamente la internacionalización común que brindaban las funcionalidades de la arquitectura del sistema y se le agregó a cada módulo su propia internacionalización. Esta modificación elimina la sobrecarga de información del bundle de internacionalización haciendo el

bundle más simple y mantenible. Garantiza una mejor administración de la internacionalización del sistema. Los cambios realizados se muestran en la Fig. 16.

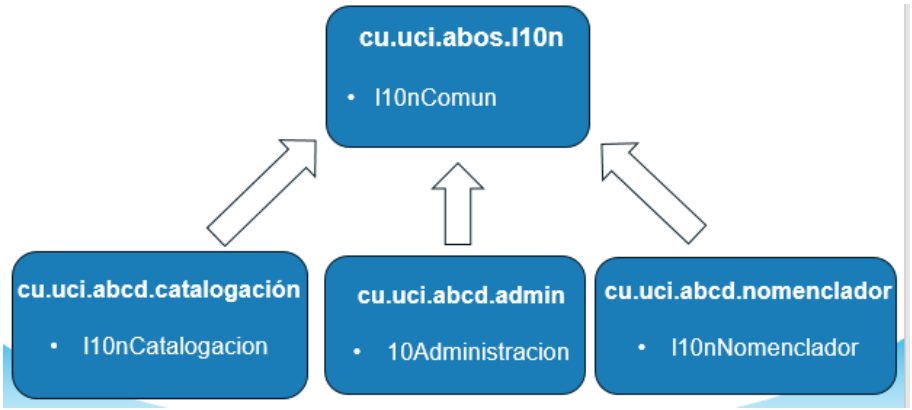


Fig. 16: Estructura actual de la internacionalización de ABCD.

Independizar los estilos de la arquitectura

Las interfaces de usuario de ABCD están implementadas en RAP. Durante el desarrollo de la mismas la repetición de código era notable ya que no existía un estándar de diseño. Los estilos se estaban escribiendo en las interfaces de usuario de cada módulo, asociándole a cada componente su tamaño, posición, etc. Eso dificultaba la uniformidad en la aplicación, porque cada desarrollador de cada módulo, podía poner el estilo que deseara a sus interfaces Otro problema notable era el no poder adaptar la aplicación a diferentes resoluciones de pantalla ya que los estilos al no estar estandarizados se mezclaban con la arquitectura.

Solución propuesta

Para darle solución a este problema se definieron los siguientes estándares de diseño:

- Para una resolución menor de 600 pixeles los componentes se ubican en una sola columna.
- Para una resolución entre 600 y 800 pixeles los componentes se ubican en dos columnas.
- Para una resolución mayor de 800 pixeles los componentes se ubican en cuatro columnas.

Para aplicar la solución se utilizaron dos tipos de layouts los cuales se explican a continuación.

GridLayout es un controlador de distribución que establece un componente de contenedores en una cuadrícula rectangular. El recipiente se divide en rectángulos de igual tamaño, y cada uno de los componentes se coloca en cada rectángulo (23).

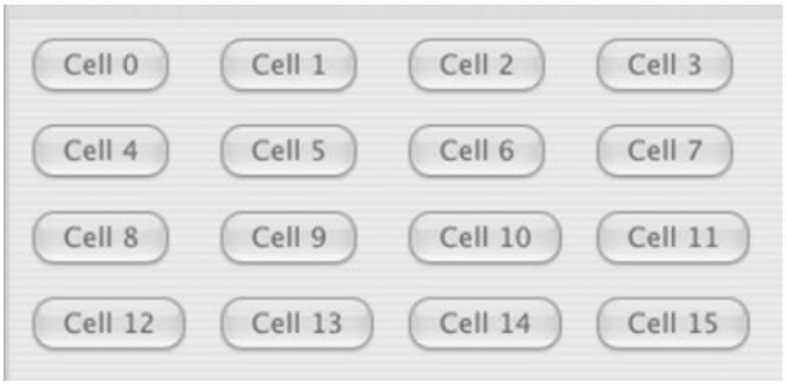


Fig. 17: Ejemplo de diseño con GridLayout (23)**.**

FormLayout es un controlador de distribución de propósito general. Es potente y flexible. Coloca los componentes en posición absoluta o relativa, permitiendo que los componentes especificados se coloquen en posiciones especificas dentro del diseño (23).

Fig. 18: Ejemplo de diseño con FormLayout (23)**.**

En la Fig. 19 se muestra el diagrama de clases de la solución propuesta.

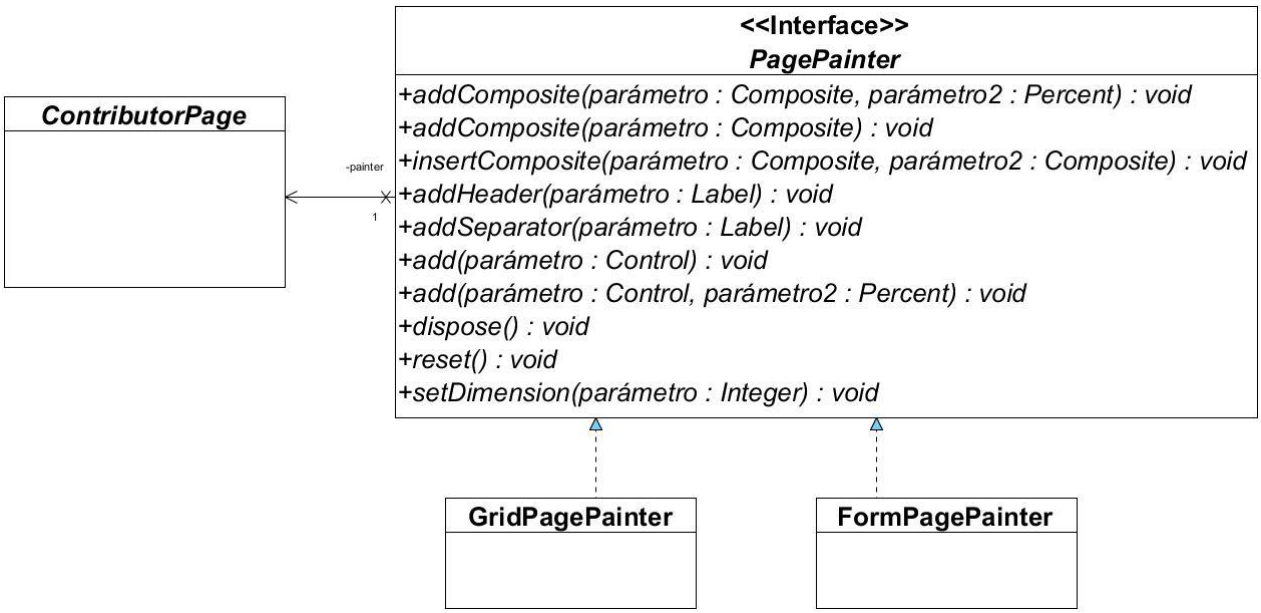


Fig. 19: Diagrama de clases de los estilos de ABCD.

Función de FormPagePainter

FormPagePainter en particular tiene la función de mediante sus métodos crear interfaces de usuario basadas fundamentalmente en las facilidades que brinda FormLayout.

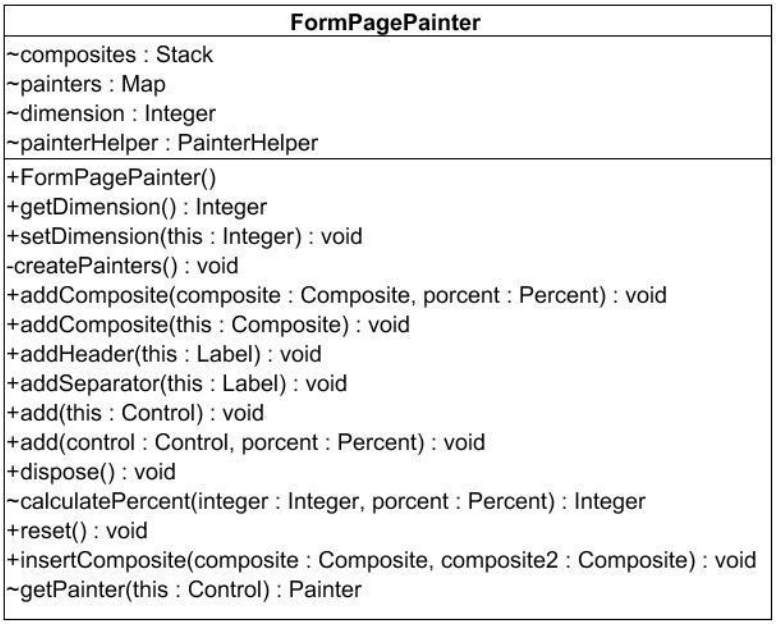


Fig. 20: Diagrama de clase de FormPagePainter.

Función de GridPagePainter

GridPagePainter tiene la función de mediante sus métodos crear interfaces de usuario basadas fundamentalmente en las facilidades que brinda GridLayout.

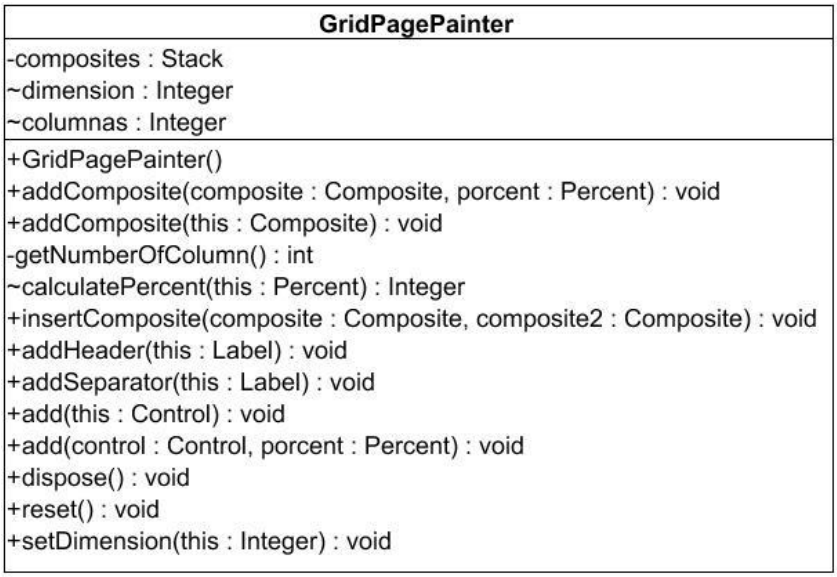


Fig. 21: Diagrama de clase de GridPagePainter.

Función de ContributorPage

Es la clase principal para el diseño. Las clases que extiendan de **ContributorPage** tienen la posibilidad de trabajar tanto con **FormPagePainter o GridPagePainter.** Esta clase implementa la interfaz PagePainter que es la que contiene los métodos esenciales para el diseño. En la tabla 10 se mencionan los métodos más esenciales y su descripción.

Tabla 10: Descripción de los métodos más importantes de ContributorPage

Nombre de Método	Descripción
dispose()	Elimina toda la página.
reset()	Implementa un salto de línea dentro del diseño.
addSeparator(Label separator)	Añade una línea horizontal o separador dentro del diseño.

add(Control control)	Añade un control(botón,label,…) dentro del diseño.
add(Control control,Porcent porcent)	Añade un control(botón,label,…) dentro del diseño y permite modificarle el tamaño mediante un porciento.
addHeader(Label header)	Añade una cabecera dentro del diseño.
addComposite(Composite Composite,Porcent Porcent)	Permite añadir un Composite y definir su tamaño en porciento dentro del diseño.

Implementar autenticación contra dominio

El protocolo ligero de acceso a directorio (LDAP) por sus siglas en ingles es un protocolo de servicio ligero que se ejecuta para acceder a los servicios de un directorio. LDAP se ejecuta a través de TCP / IP u otros servicios de transferencia orientados a la conexión. ABCD podía acceder a los servicios de LDAP, pero solo con verificando usuario y contraseña, y se necesitaba además de estos parámetros verificar cual era el dominio de conexión y la biblioteca a la que se estaba conectando el sistema.

Propuesta de solución.

En la Fig. 22 se muestra la clase GenericLoginModule que da solución al problema.

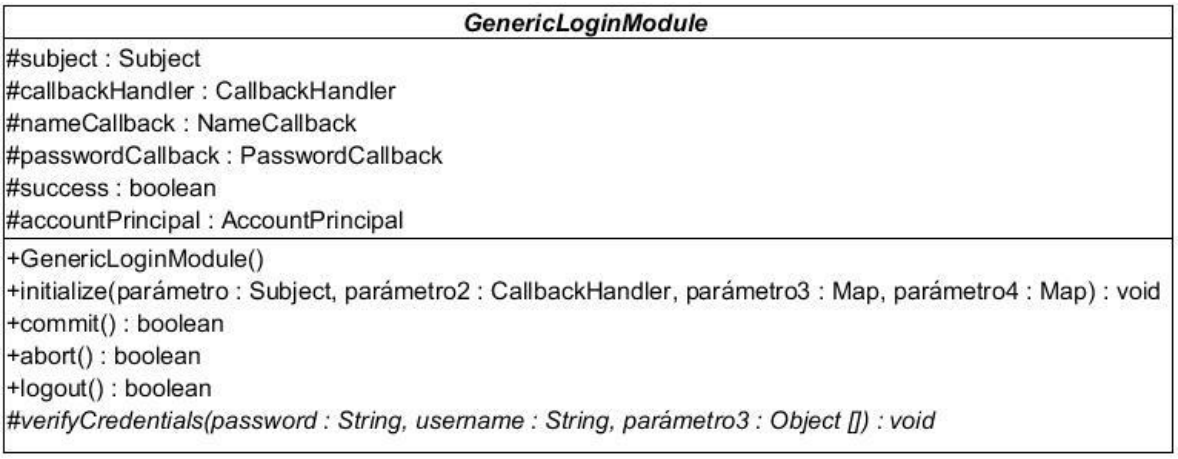


Fig. 22: Clase encargada de la autenticación por LDAP.

El método **verifyCredentials** recibe como parámetros el usuario, la contraseña y un tercer parámetro que es que un arreglo de tipo objeto que permite almacenar cualquier otro parámetro requerido por el servicio LDAP y a si brindar mayor libertad a la hora de la autenticación.

Validación de datos relacionados

La solución a la validación de los datos relacionados es importante para ABCD ya que se estaban detectando problemas con la validación de los componentes de fecha. Ya que en muchos casos era necesario seleccionar la fecha de inicio y de fin.

Solución Propuesta

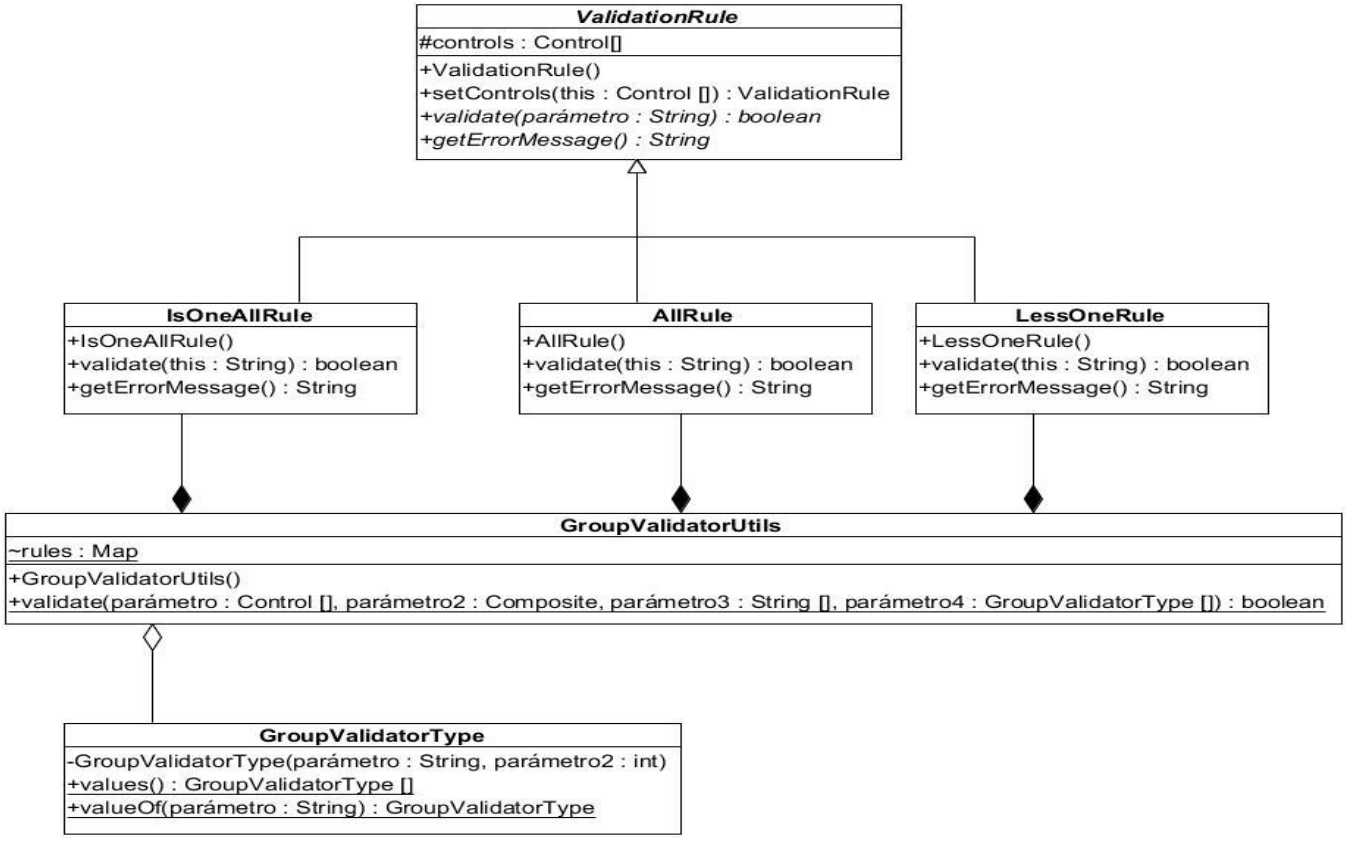


Fig. 23: Diagrama de clases de las validaciones de ABCD.

En la figura anterior se muestra el diagrama de clases de la solución implementada. Se puede observar tres clases: **IsOneAllRule, AllRule, LessOneRule**. Estas clases responden a tipos de validaciones específicas que se describen a continuación.

- La clase **IsOneAllRule** valida que si selecciona un campo se deben seleccionar todos los demás.
- La clase **AllRule** valida que se deben seleccionar todos los campos disponibles.
- La clase **LessOneRule** valida que se debe seleccionar al menos un campo.

La clase **GroupValidatorUtils** es la clase principal para las validaciones grupales. Contiene el método **validate** con los siguientes parámetros **(arrayControl: Control [], composite: Composite, arrayString: String [], type: GroupValidatorType []).** Primero recibe un arreglo de los posibles controles a validar. Segundo recibe el Composite donde se localizan esos controles. Tercero un arreglo String donde se almacena el grupo al que pertenece cada control y por último en el arreglo de tipo **GroupValidatorType** donde se almacenan los tres tipos de validaciones antes descritos.

Reestructuración de los bundles de la arquitectura

El exceso de bundles que contenía la arquitectura de ABCD dificultaba el trabajo de los desarrolladores a la hora de detectar errores en el sistema. Por lo que se hacía necesario realizar una reestructuración sus bundles, agrupando los que tenían dependencias comunes y eliminando los que ya no eran usados por el sistema.

Solución propuesta

Todos los bundles que eran necesarios para la arquitectura se encapsularon en el bundle **cu. uci. abos. core.** De los bundles encapsulados el **widget** fue en más modificado ya que fueron siete bundles que compartían dependencias y pasaron a formar parte del **core**. La reestructuración permite disminuir considerablemente la cantidad de bundles. Brinda mayor organización de la información de la arquitectura, permitiendo a los desarrolladores detectar con mayor facilidad posibles errores del sistema. A continuación, la Fig. 24 muestra con más detalles el proceso de encapsulación.

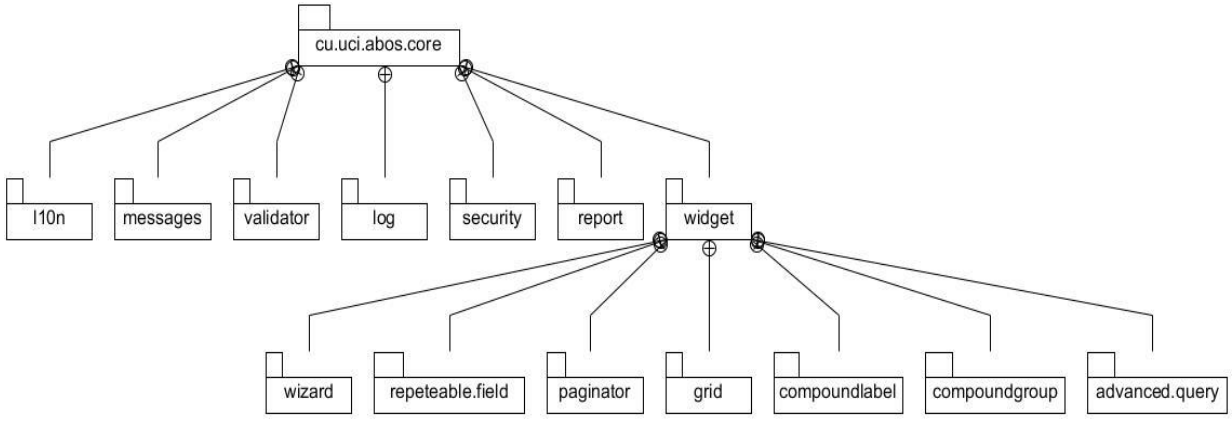


Fig. 24: Diagrama de paquetes del bundle core.

Estandarización los mensajes de información

La arquitectura ABOS no tenía soporte para mostrar ventanas emergentes informativas de carácter de negocio.

Solución Propuesta

El diagrama de clases que se muestra a continuación muestra la relación entre la clase **InformationMessage** encargada de definir la estructura de los mensajes de información y la clase **RetroalimentationUtils** que se encarga de mostrar y clasificar los diferentes tipos de mensajes de información.

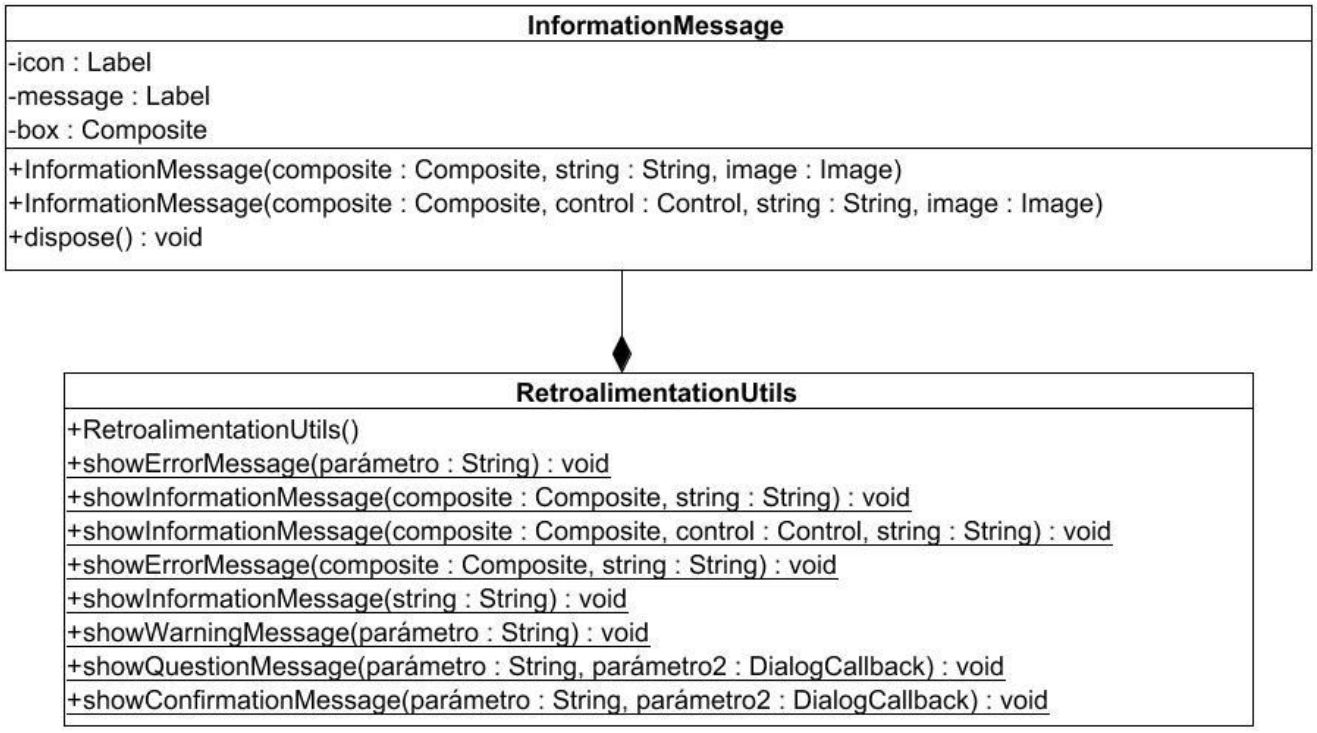


Fig. 25: Diagrama de clases de los mensajes de información.

A continuación, se muestra la descripción de los métodos definidos en la clase **RetroalimentationUtils.**

Tabla 11: Descripción de los métodos de RetroalimentationUtils

Nombre del método	Descripción
showErrorMessage(message : String)	Muestra un mensaje de error recibiendo el mensaje por parámetro.

showInformationMessage(composite: Composite, string: String)	Muestra un mensaje de información recibiendo el mensaje y el composite donde debe se debe mostrar.
showInformationMessage(composite : Composite, control : Control, string : String)	Muestra un mensaje de información recibiendo el mensaje, el composite y el control específico donde debe se debe mostrar .
showErrorMessage(composite : Composite, string : String)	Muestra un mensaje de error recibiendo el mensaje y el composite donde debe se debe mostrar.
showInformationMessage(message: String)	Muestra un mensaje de información recibiendo el mensaje por parámetro.
showWarningMessage(message: String)	Muestra un mensaje de advertencia recibiendo el mensaje por parámetro.
showQuestionMessage(message: String, action : DialogCallback)	Muestra un mensaje de información recibiendo el mensaje y la implementación de la respuesta de confirmación .
showConfirmationMessage(message: String, action : DialogCallback)	Muestra un mensaje de confirmación recibiendo el mensaje y la implementación de la respuesta de confirmación .

Añadir componente de reportes

ABCD no tenia soporte para generar reportes en formarto pdf y xls.

Solución Propuesta

Primero se definieron los parámetros necesarios para generar un reporte los cuales son:

Tabla 12: Descripción de los parámetros definidos para generar el reporte

Parámetros	Descripción

CiclegradhParam	Construye un gráfico de pastel.
ParagraphParam	Construye un párrafo.
LinialgradhParam	Construye un gráfico Lineal
PlotBarParam	Construye un gráfico de Barra
TabularParam	Construye una tabla.
EntityParam	Pasa entidades por parámetro.
ImageParam	Permite pasar imágenes por parámetro.

Después de definidos los parámetros necesarios para generar un reporte se implementó la interfaz **ReportGenerator** como clase padre de **XLSReportGenerator** para reportes en tipo exel y **PDFReportGenerator** para reportes tipo pdf. **ReportGenerator** contiene un método **generateReport ().** Este método recibe el estilo que se le va aplicar al pdf o xls, el título que va a tener el reporte, la dirección donde se va a guardar el documento y el tipo de parámetro en ese orden. La Fig. 26 muestra el diagrama de clase de **ReportGenerator.**

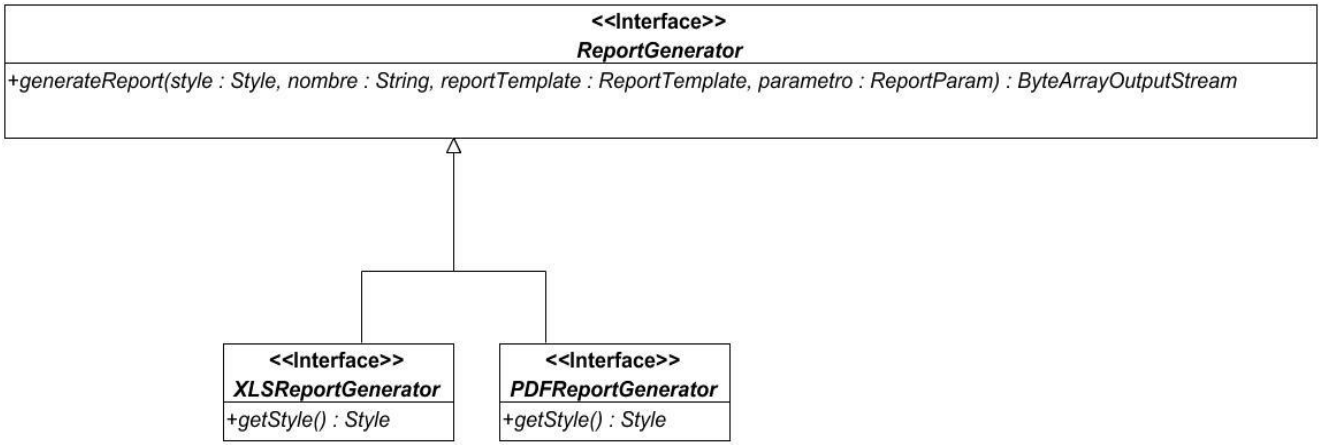


Fig. 26: Diagrama de clase de ReportGenerator.

Eliminar clases wrappers de las clases del dominio

El proceso de mostrar información tabulada en ABCD se realizaba utilizando **clases wrappers** que como su nombre lo indica envuelven los datos de las entidades y le da apariencia de objeto. Pero esto generaba una desventaja ya que había que crear una clase wrapper por cada tabla que mostrara información en la interfaz de usuario. Este problema generaba exceso de clases innecesarias en el sistema.

Solución propuesta

Java reflection: La reflexión es la capacidad de un programa para examinar y modificar la estructura y el comportamiento de un objeto en tiempo de ejecución (26).

En Java, es posible inspeccionar los atributos, clases, métodos, anotaciones, interfaces, etc. en tiempo de ejecución. No es necesario saber cómo se llaman clases o métodos, ni los parámetros que son necesarios, todos pueden recuperarse en tiempo de ejecución. También es posible crear instancias de clases nuevas, para ejecutar sus métodos, todo ello utilizando la reflexión. La reflexión proporciona un medio para invocar métodos de una clase. Típicamente esto sería necesario si no es posible crear una instancia de dicha clase. Los métodos se invocan con **java. lang. reflect. Method.**

La Fig. 27 es muestra el diagrama de la clase **Colum.** Esta clase es la que utiliza reflexión para acceder a los campos de las tablas de la base dato. Esta clase contiene el método **getCellData ()** que permite el acceso a las celdas de las tablas de la base dato.

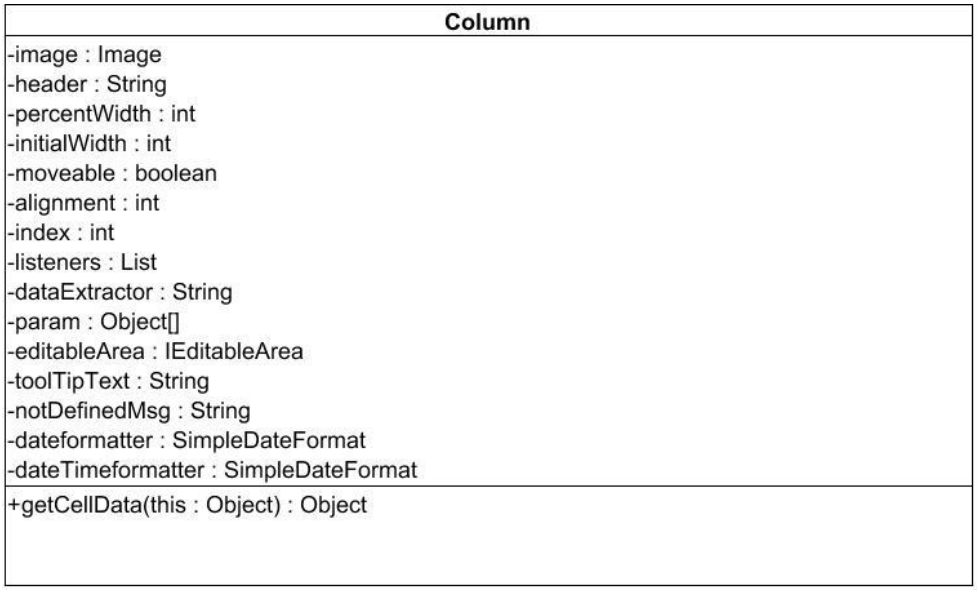


Fig. 27: Diagrama de clase de Column.

A continuación, se muestra un fragmento de la implementación del método **getCellData ().**

```
public Object getCellData(Object entity) {
    Object result = null;
    try {
        String[] methods = dataExtractor.split("\\.");

        Method method = ((BaseGridViewEntity<?>) entity).getRow().getClass().getMethod(methods[0]);
        method.setAccessible(true);
        result = method.invoke(((BaseGridViewEntity<?>) entity).getRow());

        for (int i = 1; i < methods.length; i++) {
            method = result.getClass().getMethod(methods[i]);
            method.setAccessible(true);
            result = method.invoke(result);
        }

    } catch (Exception e) {
        e.printStackTrace();
    }
    if (result != null) {
        if (result.getClass().isAssignableFrom(Date.class)) {
            return dateformatter.format(result);
        }
        if (result.getClass().isAssignableFrom(Timestamp.class)) {
            return dateTimeformatter.format(result);
        }
        return result;
```

Fig. 28: Implementación del método getCellData ().

Conclusiones parciales

Este capítulo realizó la evaluación de la arquitectura ABOS utilizando el método de evaluación de arquitectura de software SAAM. La evaluación detectó errores en el código implementado como son: falta de un estándar de diseño de interfaces de usuario, problemas en la retroalimentación y repetición excesiva de código. Se le dio solución a gran parte de los problemas detectados. Se logró una aplicación responsive, una retroalimentación unificada y un código semánticamente entendible. Se eliminó código repetido encapsulándolo en clases que son básicas en la arquitectura como ContributorPage o en clases utilitarias como RetroalimentationUtils. Se utilizó Java Reflection para disminuir el número de clases innecesarias de la arquitectura y lograr mostrar información tabulada de entidades del dominio. Se logró establecer un código limpio y mantenible.

CAPÍTULO 3: VALIDACIÓN DE LAS SOLUCIONES PROPUESTAS

3.1 Introducción

Todo el proceso de implementación se fue realizando a la par la gestión de la calidad del producto mediante el uso de herramientas y técnicas de pruebas de software. Como parte de las técnicas se abordan las pruebas de caja blanca que fueron realizadas directamente en la implementación de la arquitectura, y la utilización del componente Web Developers Tools para la realización de pruebas a la interfaz de usuario de ABCD. Se realizará un análisis de los resultados obtenidos durante el proceso de prueba, para conocer si se cumplió con el objetivo propuesto.

3.2 Pruebas del Software

Las pruebas del software son elementos críticos para la garantía de la calidad del software y representa una revisión final de las especificaciones, del diseño y de la codificación (12).A continuación se hace referencia a los objetivos y principios a tener en cuenta para realizar las pruebas de software.

Objetivos de las pruebas de software (12)

- La prueba es el proceso de ejecuci6n 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.
- Una prueba tiene éxito si descubre un error no detectado hasta entonces.

Principios de las pruebas de software (12)

- A todas las pruebas se les debería poder hacer un seguimiento hasta los requisitos del cliente.
- Las pruebas deberían planificarse mucho antes de que empiecen.
- Las pruebas deberían empezar por lo pequeño y progresar hacia lo grande.
- El principio de Pareto es aplicable a la prueba del software.
- Para ser más eficaces, las pruebas deberían ser realizadas por un equipo independiente.
- No son posibles las pruebas exhaustivas.

3.2.1 Elementos a tener en cuenta para realizar pruebas de software

Para lograr el éxito de las pruebas hay que tener en cuenta una serie de elementos, los cuales son descritos a continuación.

Estrategia de prueba del software: Integra los métodos de diseño de caso de pruebas del software en una serie bien planeada de pasos que terminará con la eficaz construcción del software. La estrategia proporciona un mapa que describe los pasos que se dan como parte de la prueba, indica cuando se planean y cuando se dan estos pasos, además de cuanto esfuerzo, tiempo y recursos se

consumen. Por tanto, cualquier estrategia de prueba debe incorporar la planeación de pruebas, el diseño de casos de pruebas, la ejecución de pruebas y la recolección y evaluación de los datos resultantes (12).

Una estrategia de prueba del software debe ser lo suficientemente flexible como para promover un enfoque personalizado. Al mismo tiempo, debe ser lo adecuadamente rígido como para promover una planeación razonable y un seguimiento administrativo del avance del proyecto (12).

La estrategia de prueba consta de los siguientes niveles:

- **Prueba de unidad**: Centra el proceso de verificación en la menor unidad del diseño del software.
- **Prueba de integración**: Es una técnica sistemática para construir la estructura del programa, mientras que al mismo tiempo se llevan a cabo pruebas para detectar errores asociados con la interacción.
- **Prueba de validación**: Se consigue cuando el software funciona de acuerdo con las expectativas del cliente.
- **Prueba del sistema:** Se realizan cuando el software es integrado con otros elementos del sistema.

3.2.2 Métodos de Prueba: Caja Negra y Caja Blanca

Prueba de Caja Blanca

Esta prueba es realizada sobre el código implementado. Se comprueban los caminos lógicos del software. Se puede examinar el estado del programa en varios puntos para determinar si el estado real coincide con el estado esperado. Permite encontrar:

- Funciones incorrectas o ausentes.
- Errores de rendimiento.
- Errores de inicialización.

Las técnicas para realizar las pruebas de caja blanca son:

- Prueba del camino básico.
- Prueba de condición.
- Prueba de flujo de datos.
- Prueba de bucles.

Este trabajo se rige por la estrategia de **Prueba de Unidad,** lo que implica que para el desarrollo de pruebas de caja blanca se utilizará el framework JUnit, con el objetivo realizar pruebas unitarias.

Pruebas de Caja Negra

Esta prueba se lleva a cabo sobre la interfaz del software. Tiene como objetivo demostrar que las funciones del software son operativas, que las entradas se aceptan de forma adecuada y producen el resultado esperado. Permite que la integridad de la información externa se mantenga.

Para desarrollar estas pruebas existen varias técnicas:

- **Técnica de partición de equivalencia**: esta técnica divide el campo de entrada en clases de datos que tienden a ejercitar determinadas funciones del software.
- **Técnica de análisis de valores límites**: esta técnica prueba la habilidad del programa para manejar datos que se encuentran en los límites aceptables.
- **Técnica de Grafos de causa-efecto**: es una técnica que permite al encargado de la prueba validar complejos conjuntos de acciones y condiciones.

Caso de Prueba

Los casos de prueba deben diseñarse para descubrir errores debidos a cálculos erróneos, comparaciones incorrectas o flujo de control inadecuado. Se diseñan para garantizar que: se satisfagan todos los requerimientos de funcionamiento, se logran todas las características de comportamiento, todo el contenido es preciso y se presenta la manera adecuada, se logran todos los requerimientos de rendimiento, la documentación es correcta y se satisfacen la facilidad de uso y otros requerimientos (12).

3.3 Pruebas de Caja Blanca implementadas

Como se mencionó en el punto anterior estas pruebas se realizan con el framework JUnit, herramienta utilizada para realizar pruebas unitarias.

¿Por qué JUnit?

JUnit es una framework Java que se utilizada para automatizar los procesos de prueba mediante la creación de pruebas. Realiza una prueba en el código que indique el usuario. Siempre que se vaya a desarrollar algún tipo de software, habrá que tener en cuenta las pruebas a realizar, con esto se demuestra que el programa o librería funciona correctamente (27).

Normalmente las pruebas se realizan por parte del programador, esto incluye que el orden elegido podría no ser correcto y con ello alargar demasiado el trabajo. Aunque inicialmente el proceso sea más

rápido, con el avance de la aplicación podría complicarse el trabajo, ya que cada vez que se fuera a probar algo, habría que volver a escribir el código para realizar la prueba ya que no se tiene la certeza de cuáles serán los módulos afectados con varios cambios, ni podremos adivinar exactamente la línea donde se ha generado el error (27).

Existen varias razones para utilizar JUnit a la hora de hacer pruebas de código:

- La herramienta no tiene coste alguno, se puede descargar directamente desde la Web oficial.
- Es una herramienta muy utilizada, por lo que no será complicado buscar documentación.
- Existen varios plugins para poder utilizar con diferentes Entornos de Desarrollo Integrado (**IDE**).
- Existen también muchas herramientas de pruebas de cobertura que utilizaran como base JUnit.
- Los resultados son chequeados por la propia aplicación y dará los resultados inmediatamente.
- Utilizando los tests programados en JUnit, la estabilidad de la aplicación mejorará sustancialmente.
- Los tests realizados se podrán presentar junto con el código, para validar el trabajo realizado.

3.3.1 Pruebas unitarias realizadas con JUnit

JUnit tiene una manera específica para visualizar los reportes de las pruebas realizadas:

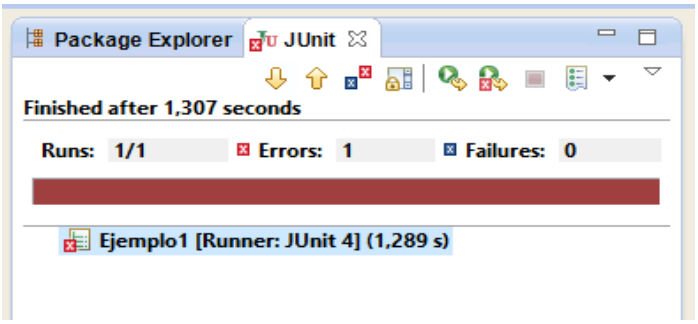


Fig. 29: Integración de JUnit con Eclipse, (prueba no satisfactoria).

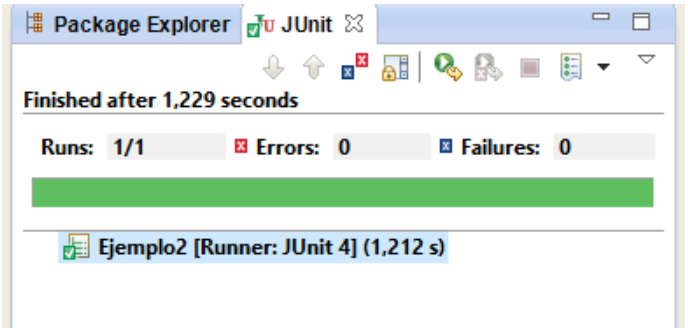


Fig. 30: Integración de JUnit con Eclipse, (prueba satisfactoria).

A continuación, se muestran imágenes que representan la realización de las pruebas de caja blanca. Se le aplicó esta prueba a la clase **GroupValidatorUtil**, específicamente al método **validate ().**

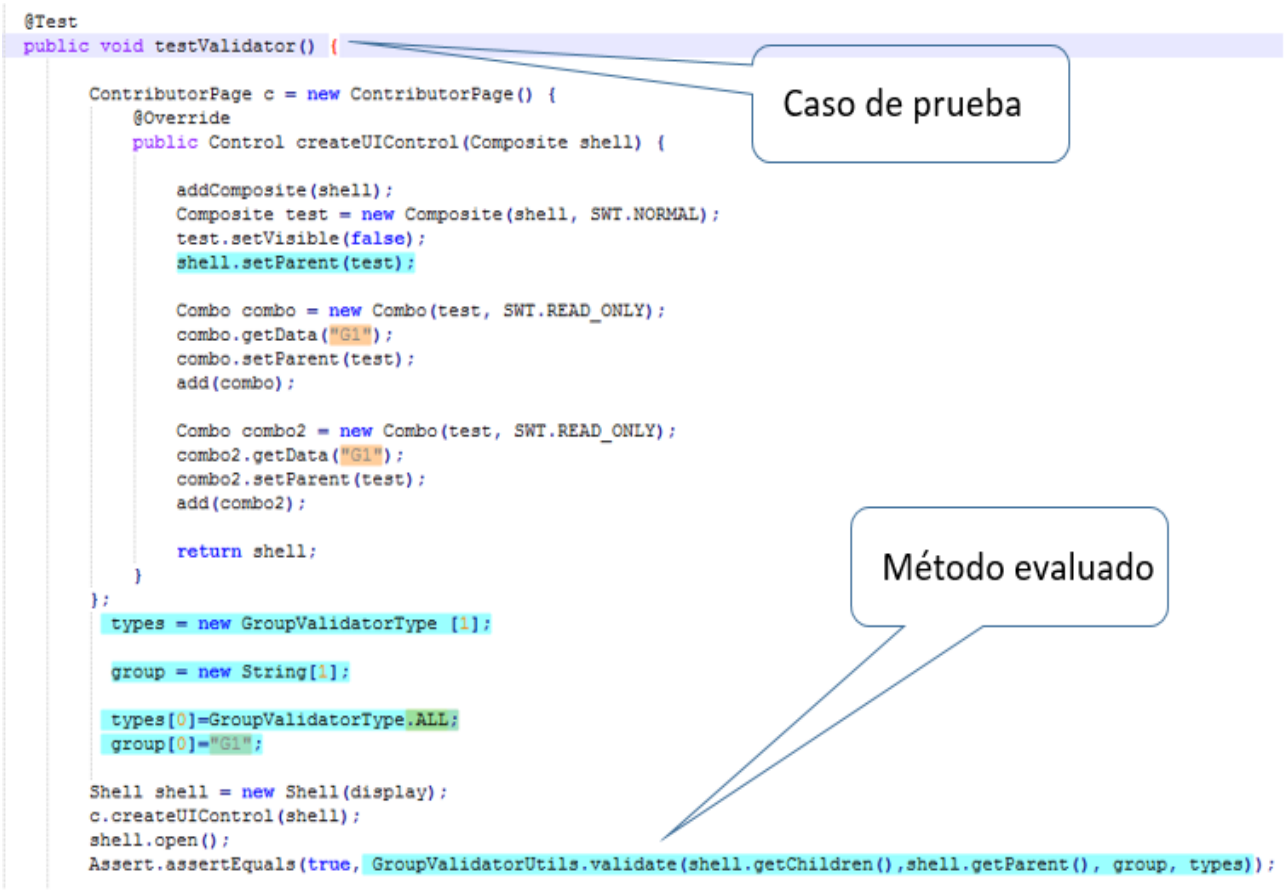


Fig. 31: Caso de prueba elaborado para GroupValidatorUtils.

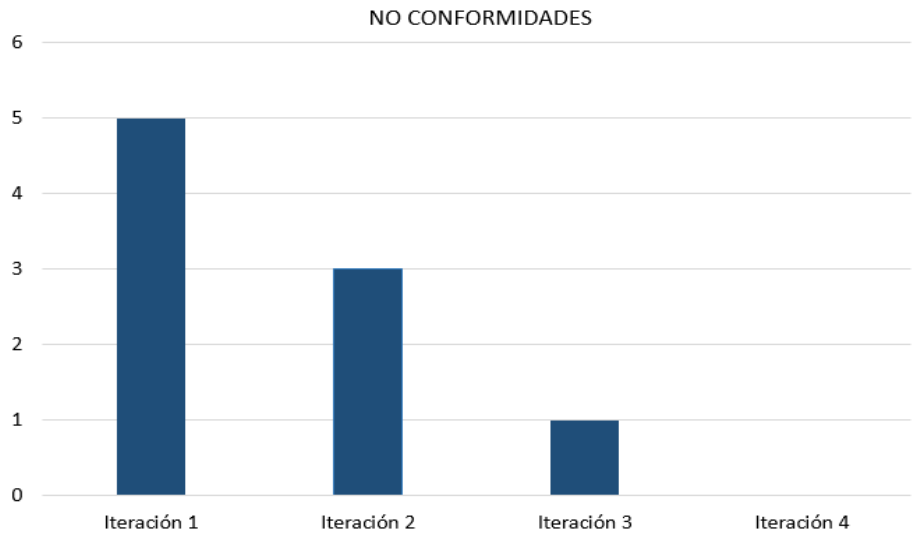


Fig. 32: Iteraciones realizadas para el caso de prueba testValidator.

Después de realizadas 4 iteraciones como se muestra en la gráfica de barras anterior se obtuvo el resultado esperado. Como se muestra a continuación.

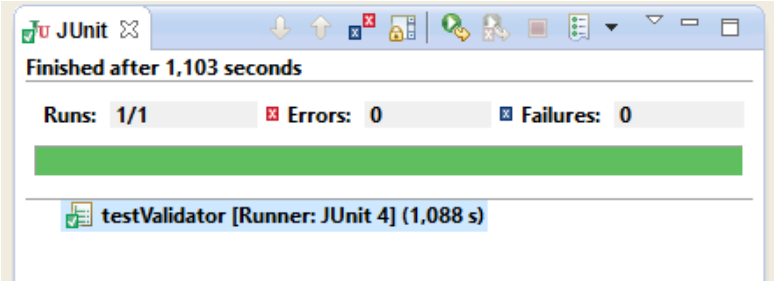


Fig. 33 Evaluación del caso de prueba testValidator con JUnit.

En la Fig. 34 se muestra el caso de prueba elaborado para las funcionalidades más importantes de la clase **ContributorPage** descritas en la **Tabla 10.**

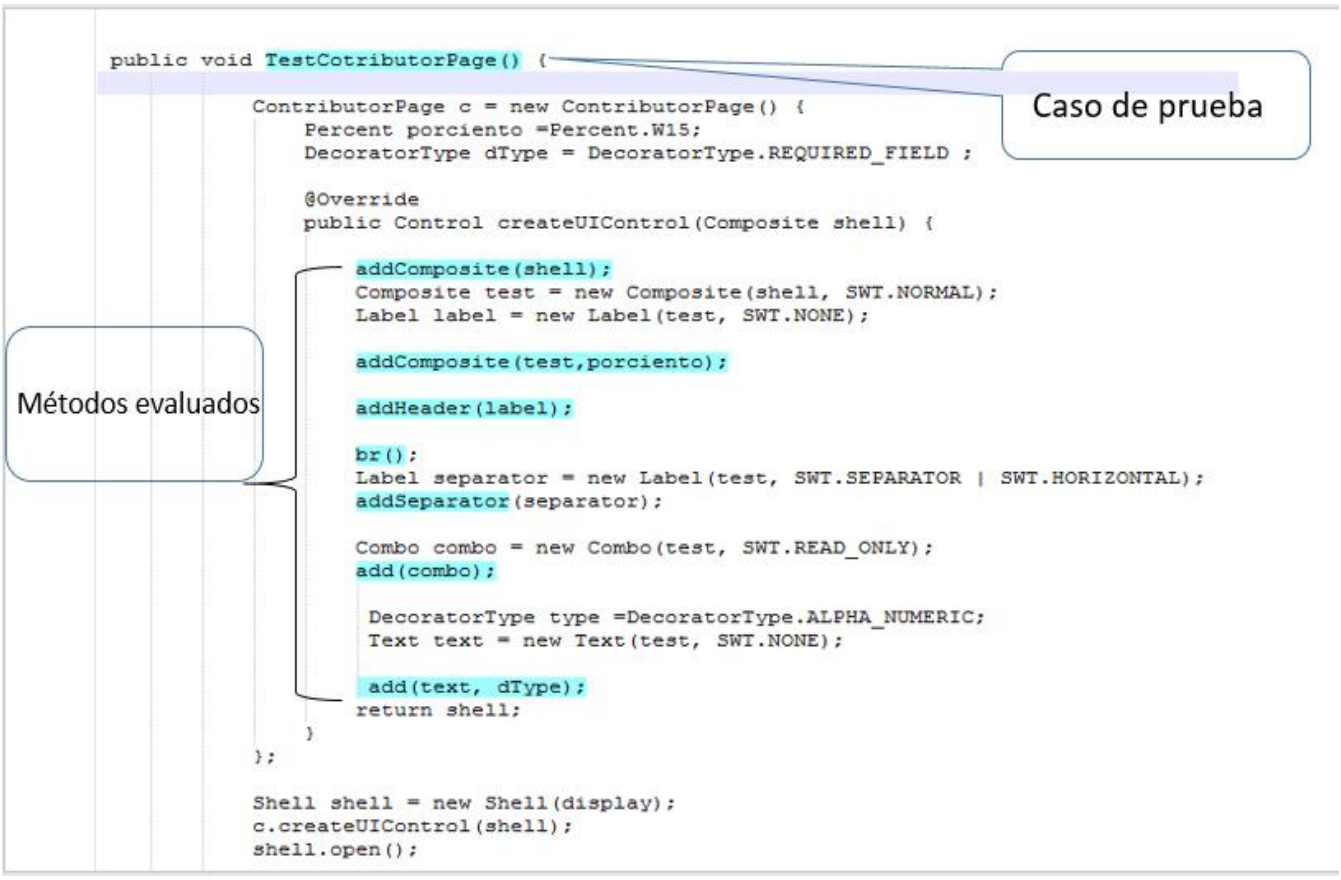


Fig. 34: Caso de prueba elaborado para ContributorPage.

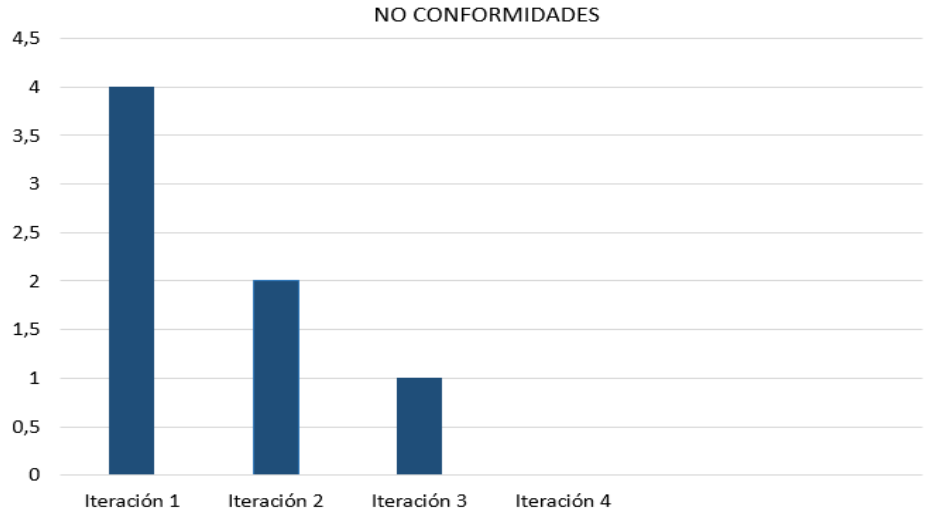


Fig. 35: Iteraciones realizadas para el caso de prueba testContributorPage.

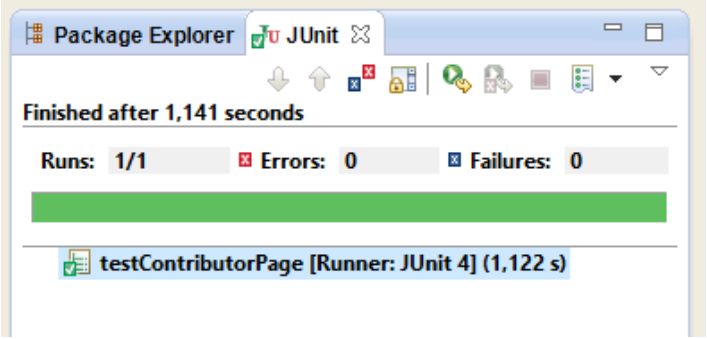


Fig. 36 Evaluación del caso de prueba testContributorPage con JUnit.

A continuación, se muestran imágenes que representan la realización de las pruebas de caja blanca. Se le aplicó esta prueba a la clase **Column**, específicamente al método **getCellData ().**

```
package cu.uci.abos.core.widget.grid;

import cu.uci.abcd.domain.common.Person;
import junit.framework.TestCase;

public class TestCellData extends TestCase {
    Person person;

    protected void setUp() throws Exception {
        super.setUp();
    }

    protected void tearDown() throws Exception {
        super.tearDown();
    }

    public void testGetCellData() {
        person = new Person();
        person.setFirstName("Rafael");
        Column column = new Column(15, 20);
        column.setDataExtractor("getFirstName");
        assertEquals("Rafael",column.getCellData(new BaseGridViewEntity<Person>(person)));
    }

}
```

Caso de prueba

Método evaluado

Fig. 37: Caso de prueba elaborado para la clase Column.

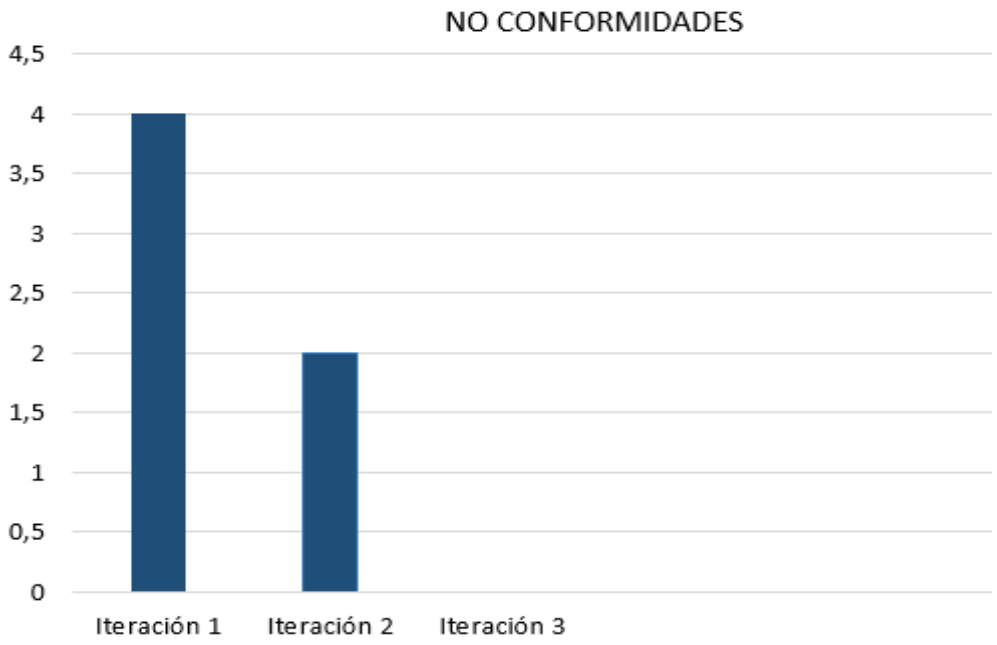


Fig. 38: Iteraciones realizadas para el caso de prueba testCellData.

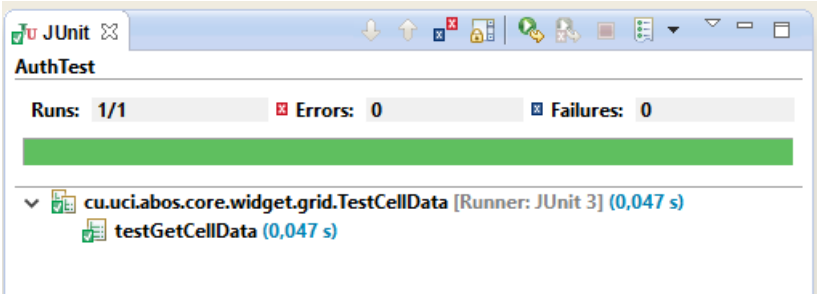


Fig. 39 Evaluación del caso de prueba testCellData con JUnit.

3.4 Herramienta de prueba para diseños responsive: Web Developers Tools

Los diseños web responsive son una serie de prácticas aplicadas al diseño web que permiten visualizar la página o sitio web correctamente desde cualquier dispositivo, dando prioridad a la experiencia del usuario. En un entorno cada vez más multidispositivo (28).

Las Web Developers Tools son una serie de herramientas que se instalan a modo de extensión en el navegador web, por el momento solo en Chrome y Firefox. Entre sus bondades y ventajas, no solo están el poder probar si una página web está adapta a dispositivos móviles, sino que incluyen múltiple opciones para ayudar en el desarrollo web. (29).Estas son algunas de las funcionalidades que brinda:

Disable: Sirve para deshabilitar diversos componentes o partes del navegador. Por ejemplo, para deshabilitar la caché, JavaScript, el bloqueador de popups y los colores de la página.

Cookies: Permite administrar las cookies, como por ejemplo no permitirlas o encontrar información completa de las cookies que nos envían en cada página.

CSS: Para realizar diversas acciones sobre las hojas de estilo. Permite no aplicarlas a la página, ver algunos estilos y otro no, incluso tiene una opción para editar las hojas de estilo en el propio Firefox y así ver cómo alterando el código CSS de la página se cambia el aspecto de la web.

Forms: Permite realizar diversas acciones sobre formularios, como resaltar y obtener información de todos sus campos.

Images: Dispone de diversas opciones para alterar la presencia de las imágenes. Puedes permitirlas o deshabilitarlas, incluso deshabilitar sólo las imágenes que son externas al sitio web. Mostrar información de las imágenes, como sus rutas, tamaño en bytes, dimensiones en píxeles.

Information: Permite ver información sobre el tamaño del documento, la profundidad de las tablas, el JavaScript que se está ejecutando, información de los enlaces.

Outline: Sirve para destacar elementos de la página, como los enlaces, frames, tablas, celdas, elementos obsoletos que se puedan estar utilizando en el código.

Resize: Es una interesante utilidad para redimensionar la ventana del navegador, para observar el aspecto que tiene la web si se redimensiona la ventana a otros tamaños o definiciones de pantalla.

Tools: Contiene unas herramientas muy útiles, como verificador de links, validadores del código, acceso a la consola de Java o JavaScript.

View Source: Como su nombre indica, sirve para ver el código fuente.

A continuación, se muestran imágenes de la herramienta especifica que se utilizó instalada en el Mozilla Firefox versión 40.0.2.

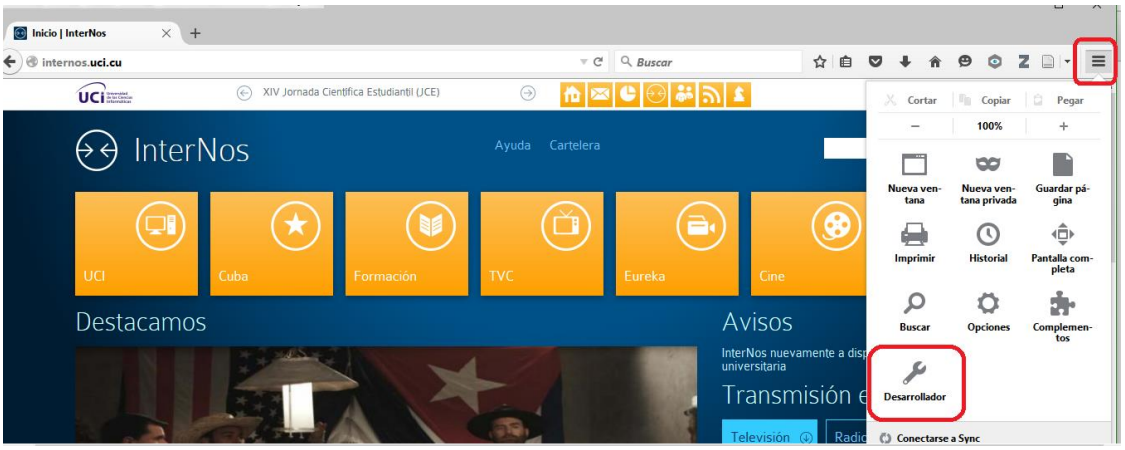


Fig. 40 Paso 1 para acceder a la herramienta.

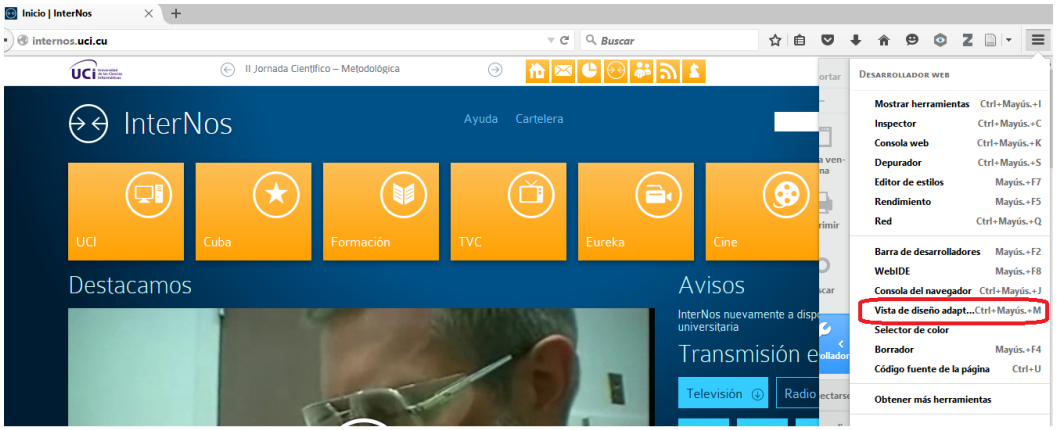


Fig. 41 Paso 2 para acceder a la herramienta.

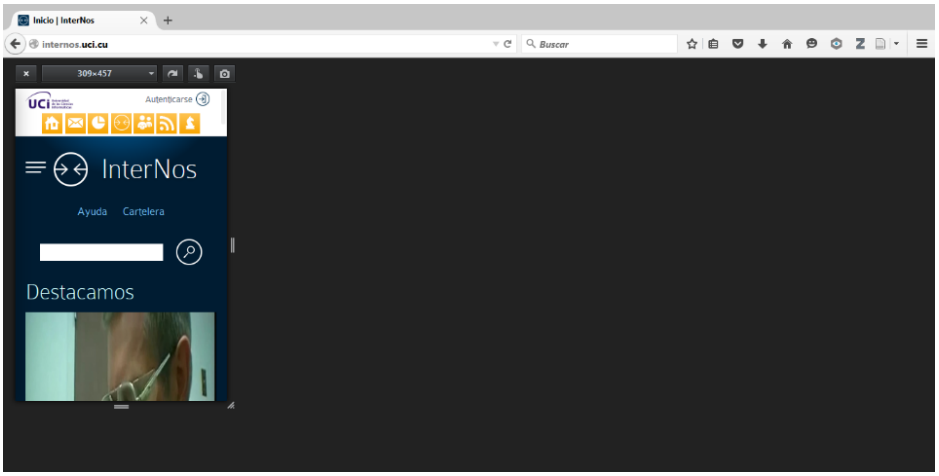


Fig. 42 Prueba con internos.uci.cu.

3.4.1 Pruebas realizadas a la interfaz de ABCD

Se eligió el módulo Nomenclador para realizar las pruebas. A continuación, se muestra dicho módulo puesto a prueba bajo diferentes resoluciones.

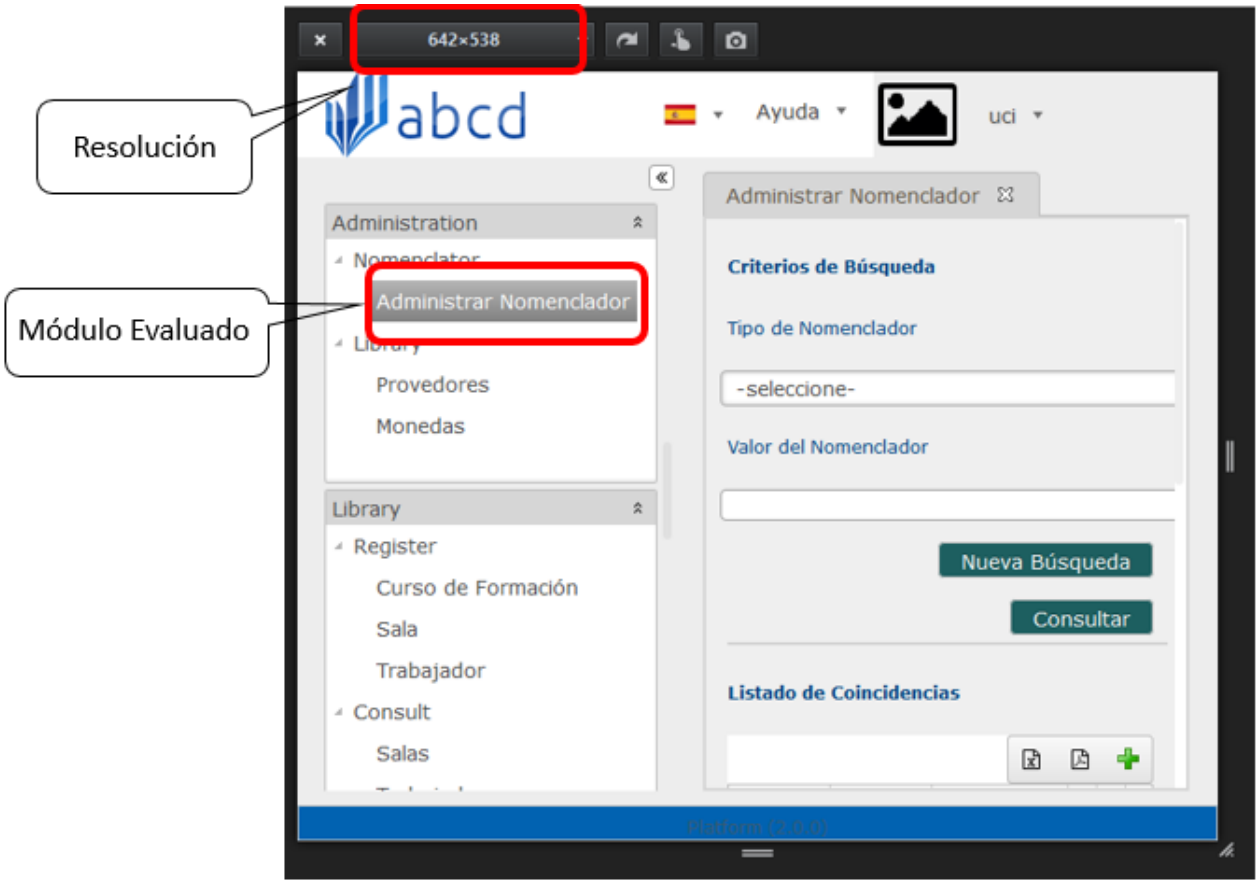


Fig. 43 Módulo Nomenclador puesto a prueba en la resolución 642x538.

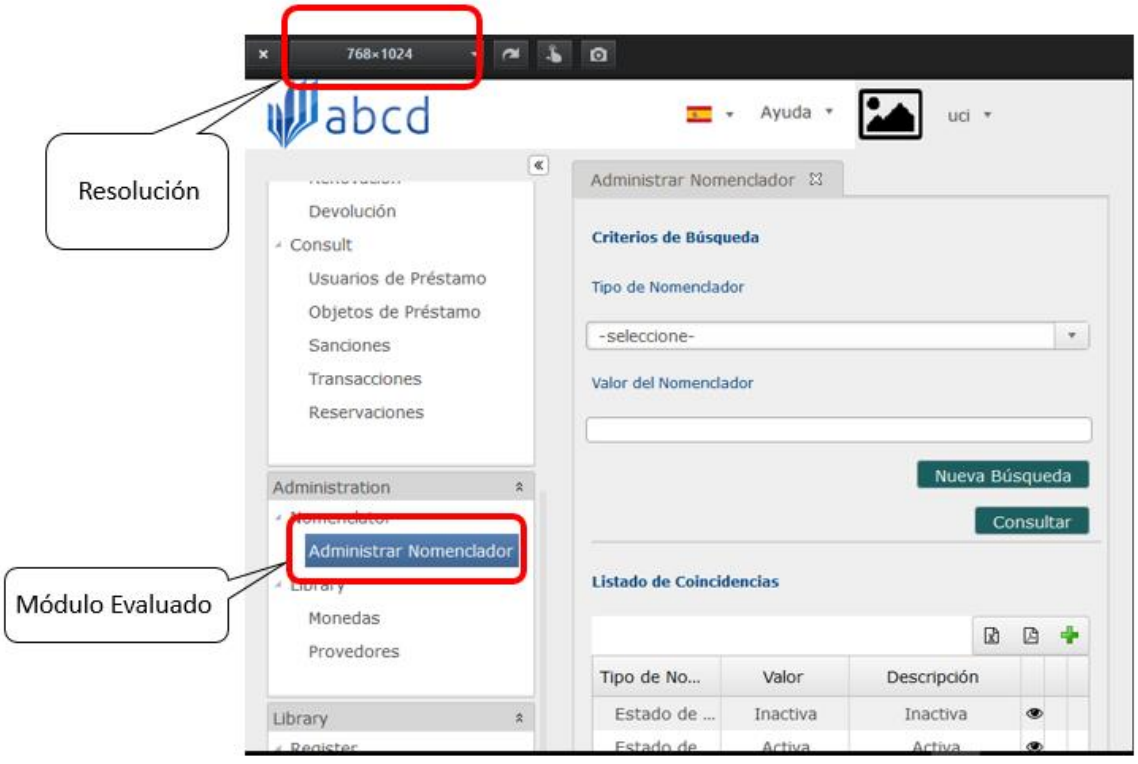


Fig. 44 Módulo Nomenclador puesto a prueba en la resolución 768x1024.

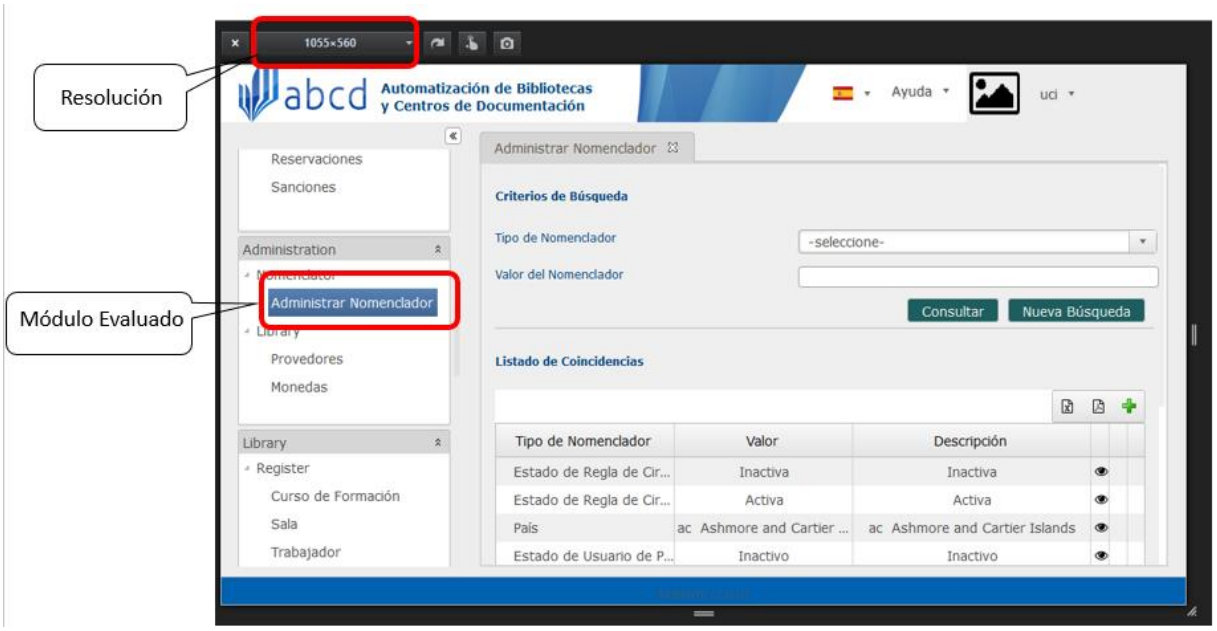


Fig. 45 Módulo Nomenclador puesto a prueba en la resolución 1055x560.

Conclusiones parciales

En este capítulo se expuso la validación de las soluciones propuestas en el capítulo anterior, realizando pruebas unitarias a los métodos implementados con la utilización del framework JUnit. Se evidenció el correcto funcionamiento de las funcionalidades implementadas, así como su interacción con la plataforma evidenciada en las pruebas realizadas a la interfaz de usuario de ABCD con las Web Developers Tools.

Conclusiones

- Se especificó que la evaluación de arquitectura de software es una vía que contribuye para garantizar la calidad de los proyectos.
- Se demostró que los métodos de evaluación de arquitecturas de software de Kazman era la vía que respondía a los intereses del grupo de desarrollo para realizar la evaluación de la arquitectura ABOS, a pesar de que las Técnicas evaluación de arquitecturas de software de Bosch son muy usadas en la actualidad.
- Se realizó una caracterización de la arquitectura ABOS haciendo énfasis en la funcionalidad de cada una de sus capas, estructura de los bundles que la componen y en las principales características de las herramientas y frameworks utilizados.
- Se demostró que el método de evaluación de arquitectura SAAM era el indicado para evaluar ABOS.

- Se solucionaron gran parte de los problemas detectados tras la evaluación de la arquitectura. Lográndose una aplicación responsive, se estableció un estándar de diseño para las interfaces de usuario, se mejoró la retroalimentación del sistema, se establecieron validaciones grupales y se modificó la estructura de los bundles del sistema.
- Se diseñaron y ejecutaron pruebas de a cada una de las soluciones implementadas utilizando JUnit para la realización de pruebas unitarias.

Recomendaciones

- Poner en práctica cuanto antes la evaluación de arquitectura de software en los proyectos de la UCI para obtener mejores resultados productivos.
- Investigar acerca de la posibilidad de comprobar los resultados obtenidos con otros métodos de evaluación de arquitectura.
- Preparar al personal que estará vinculado al proceso de evaluación, en el funcionamiento de la metodología.

Referencias Bibliográficas

1.Rick Kazman, Paul Clements,Mark Klein. 2005. *Evaluating Software Architectures.Methods and case studies.* s.l. : Adison-Wesley Professional, 2005. ISSN: 978-0201704822.

2.Shaw, Mary. 1989. *Larger scale systems require higher-level abstractions.* s.l. : ACM Sigsoft Software engineering notes, 1989.

3.LTC Erik Mettala, Marc H. Graham. 1992. *The Domain-Specific Software Architecture Program.* 1992.

4.David Garlan, Mary Shaw. 1993. *An introduction to software architecture.* 1993.

5. *The golden age of software architecture.* **Michael Mattsson, Håkan Grahn, Frans Mårtensson.** Carnegie Mellon Univ., Pittsburgh, PA, USA : IEEE Computer Society, 2006, Vol. 23. 0740-7459.

6.Grady Booch, James Rumbaugh,Ivar Jacobson. 1999. *The unified modeling language user guide.* Boston : s.n., 1999.

7.Len Bass, Gregory Abowd,Paul Clements,Rick Kazman. 1997. *Recommended Best Industrial Practice for Software Architecture Evaluation.* 1997.

8.Christine Hoffmeister, Robert Nord. 2000. *Applied Software Architecture.* 2000.

9.Barry Boehm, Ahmed Abd-Allah. 1995. *Reasoning about the composition of heterogeneous architectures.* Technical Report UCS-CSE-95-503, University of Southern California : s.n., 1995.

10. *Norma ISO/IEC 9126.* **Sastre, Instructor Benjamín del.** 2001.

12.Pressman, Roger S. *Ingeniería de Software.* s.l. : Mc Graw Hill.

13.Mugurel T. Ionita, Dieter K. Hammer,Henk Obbink. 2002. *Scenario-based software architecture evaluation methods: An overview.* 2002.

14. Gómez, Salvador Omar. 2007. *Evaluando la arquitectura de software.* México : Brainworx S.A, 2007.

15. Holly Cummins, Timothy Ward. 2013. *Enterprise OSGi in action: with examples using Apache Aries.* Shelter Island, NY : Manning, 2013.

17. Miguel, Roberto Montero. 2011. *OSGI.* 2011.

19. EclipseLink 2.6 Release. *EclipseLink 2.6 Release.* [En línea] [Citado el: 10 de 6 de 2016.] https://eclipse.org/eclipselink/releases/2.6.php.

20. Arnaud Cogoluègnes, Thierry Templier,Andy Piper. 2010. *Spring dynamic modules in action.* Greenwich, Conn : Manning, 2010.

22. Ritchie, Simon. 2002. *SWT – The Standard Widget Toolkit.* 2002.

23. Matthew Scarpino, Stephen Holder,Stanford Ng,Laurent Mihalkovic. 2005. *SWT/JFace in action.* Greenwich : Manning, 2005.

25.Regalado, Ing. Yamila Vigil. 2009. *Metodología de evaluación de arquitecturas de software.* Habana : s.n., 2009.

26.Ira Forman, Nate Forman. 2005. *Java Reflection in Action.* s.l. : manning, 2005.

27. Petar Tahchiev, Felipe Leme,Vincent Massol,Gary Gregory. *JUnit in Action Second Edition.* s.l. : manning, 2011.

28. Peterson, Clarissa. *Learning Responsive Web Desing.* Canadá : s.n., 2014. ISBN:978-1-4493-6294-2.

36. Bart Broekman, Edwin Notenboom. 2008. *Testing embedded software.* London : Reprinted, 2008.

38.Frank Buschmann, Kevlin Henney. 2008. *Pattern-Oriented Software Architecture.* 2008.

39.Gregory D. Abowd, Robert Allen,David Garlan. 1995. *Formalizing style to understand descriptions of software architecture.* 1995.

44. *A Basis for Analyzing Software Architecture Analysis Methods.* Kazman, Rick. 2005, Vol. 13. 329-355.

Bibliografía

1. **Rick Kazman, Paul Clements,Mark Klein.** *Evaluating Software Architectures.Methods and case studies.* s.l. : Adison-Wesley Professional, 2005. ISSN: 978-0201704822.

2. **Shaw, Mary.** *Larger scale systems require higher-level abstractions.* s.l. : ACM Sigsoft Software engineering notes, 1989.

3. **LTC Erik Mettala, Marc H. Graham.** *The Domain-Specific Software Architecture Program.* 1992.

4. **David Garlan, Mary Shaw.** *An introduction to software architecture.* 1993.

5. *The golden age of software architecture.* **Michael Mattsson, Håkan Grahn, Frans Mårtensson.** Carnegie Mellon Univ., Pittsburgh, PA, USA : IEEE Computer Society, 2006, Vol. 23. 0740-7459.

6. **Grady Booch, James Rumbaugh,Ivar Jacobson.** *The unified modeling language user guide.* Boston : s.n., 1996.

7. **Len Bass, Gregory Abowd,Paul Clements,Rick Kazman.** *Recommended Best Industrial Practice for Software Architecture Evaluation.* 1997.

8. **Christine Hoffmeister, Robert Nord.** *Applied Software Architecture.* 2000.

9. **Barry Boehm, Ahmed Abd-Allah.** *Reasoning about the composition of heterogeneous architectures.* Technical Report UCS-CSE-95-503, University of Southern California : s.n., 1995.

10. *Norma ISO/IEC 9126.* **Sastre, Instructor Benjamín del.** 2001.

11. **Len Bass, Paul Clements, Rick Kazman.** *Software Arquitecture in Practice Second Edition.* s.l. : SEI (Series in Software Ingineering), 2003.

12. **Pressman, Roger S.** *Ingeniería de Software.* s.l. : Mc Graw Hill.

13. **Mugurel T. Ionita, Dieter K. Hammer,Henk Obbink.** *Scenario-based software architecture evaluation methods: An overview.* 2002.

14. **Gómez, Salvador Omar.** *Evaluando la arquitectura de software.* México : Brainworx S.A, 2007.

15. **Holly Cummins, Timothy Ward.** *Enterprise OSGi in action: with examples using Apache Aries.* Shelter Island, NY : Manning, 2013.

16. **Release 4 Version 4.3 – OSGi™ Alliance.** ***Release 4 Version 4.3 – OSGi™ Alliance.*** **[En línea] [Citado el: 21 de 5 de 2016.] https://www.osgi.org/release-4-version-4-3/.**

17. Miguel, Roberto Montero. ***OSGI.*** **2011.**

18. EclipseLink. ***EclipseLink.*** **[En línea] [Citado el: 21 de enero de 2016.] http://www.eclipse.org/eclipselink/.**

19. EclipseLink 2.6 Release. ***EclipseLink 2.6 Release.*** **[En línea] [Citado el: 10 de 6 de 2016.] https://eclipse.org/eclipselink/releases/2.6.php.**

20. Arnaud Cogoluègnes, Thierry Templier,Andy Piper. ***Spring dynamic modules in action.*** **Greenwich, Conn : Manning, 2010.**

21. Virgo - Home. ***Virgo - Home.*** **[En línea] [Citado el: 5 de 21 de 2016.] http://www.eclipse.org/virgo/.**

22. Ritchie, Simon. ***SWT – The Standard Widget Toolkit.*** **2002.**

23. Matthew Scarpino, Stephen Holder,Stanford Ng,Laurent Mihalkovic. ***SWT/JFace in action.*** **Greenwich : Manning, 2005.**

24. Remote Application Platform (RAP). ***Remote Application Platform (RAP).*** **[En línea] [Citado el: 20 de diciembre de 2015.] http://www.eclipse.org/rap/.**

25. Regalado, Ing. Yamila Vigil. ***Metodología de evaluación de arquitecturas de software.*** **Habana : s.n., 2009.**

26. Ira Forman, Nate Forman. ***Java Reflection in Action.*** **s.l. : manning, 2005.**

27. Petar Tahchiev, Felipe Leme,Vincent Massol,Gary Gregory. *JUnit in Action Second Edition.* s.l. : manning, 2011.

28. Peterson, Clarissa. *Learning Responsive Web Desing.* Canadá : s.n., 2014. ISBN:978-1-4493-6294-2.

29. Estamos construyendo una Web mejor — Mozilla. *Estamos construyendo una Web mejor — Mozilla.* [En línea] [Citado el: 5 de 24 de 2016.] https://www.mozilla.org/es-ES/.

30. Benefits of Using OSGi – OSGi™ Alliance. *Benefits of Using OSGi – OSGi™ Alliance.* [En línea] [Citado el: 20 de enero de 2016.] https://www.osgi.org/developer/benefits-of-using-osgi/.

31. Persistencia en BD con Spring Data JPA (I): Primeros pasos | danielme.com. *Persistencia en BD con Spring Data JPA (I): Primeros pasos | danielme.com.* [En línea] [Citado el: 15 de enero de 2016.] http://danielme.com/2014/02/08/persistencia-en-bd-con-spring-data-jpa/.

32. Remote Application Platform (RAP). *Remote Application Platform (RAP).* [En línea] [Citado el: 10 de enero de 2016.] http://www.eclipse.org/rap/.

33. Arquitectura de Software. *Arquitectura de Software.* [En línea] [Citado el: 20 de diciembre de 2015.] http://sg.com.mx/revista/27/arquitectura-software.

34. Hibernate ORM - Hibernate ORM. *Hibernate ORM - Hibernate ORM.* [En línea] [Citado el: 19 de enero de 2016.] http://hibernate.org/orm/.

35. Arquitectura en Capas - EcuRed. *Arquitectura en Capas - EcuRed.* [En línea] [Citado el: 15 de 12 de 2015.] http://www.ecured.cu/Arquitectura_en_Capas.

36. Bart Broekman, Edwin Notenboom. *Testing embedded software.* London : Reprinted, 2008.

37. *Manning - SWT JFace in Action.* s.l. : Manning, 2000.

38. Frank Buschmann, Kevlin Henney. *Pattern-Oriented Software Architecture.* 2008.

39. Gregory D. Abowd, Robert Allen,David Garlan. *Formalizing style to understand descriptions of software architecture.* 1995.

40. Spring 3.1.0 M2 Released. *Spring 3.1.0 M2 Released.* [En línea] [Citado el: 21 de 5 de 2016.] https://spring.io/blog/2011/06/08/spring-3-1-0-m2-released.

42. Marcotte, Ethan. Responsive Web Design. *Responsive Web Design.* [En línea] 2010. [Citado el: 21 de 5 de 2016.] http://alistapart.com/article/responsive-web-design.

43. R. N. Taylor, N. Medvidovic, E. M. Dashofy. *Software Architecture: Foundations, Theory, and Practice.* s.l. : Wiley Publishing, 2009 . ISBN:0470167742 9780470167748.

44. ***A Basis for Analyzing Software Architecture Analysis Methods.*** **Kazman, Rick. 2005, Vol. 13. 329-355.**

45. ***Arquitectura orientada a servicios para software.*** **Erick Salinas, Narciso Cerpa, Pablo Rojas,. Chile : s.n., 2011, Vol. 19 . 0718-3291.**

46. ***Perspectivas y Experiencias en el Desarrollo de un Curso de Arquitectura de Software.*** **López G., Nicolás, Villamil G., Pilar y Casallas G., Rubby. 1, Colombia : s.n., 2008, Vol. 5. ISSN: 1657-7663.**

47. ***Systematic selection of software architecture styles.*** **Galster, M., Eberlein, A. y Moussavi, M. 5, 2010, Vol. 4. ISSN 1751-8806.**

Glosario de Términos

Refactorización: El término refactorización se usa a menudo para describir la modificación del código fuente sin cambiar su comportamiento, lo que se conoce informalmente por limpiar el código.

Calidad de Software: Grado en el cual el software posee una combinación deseada de atributos.

ASW (Arquitectura de Software): Es el estudio de la estructura de un sistema, dividiendo las funciones entre los módulos del sistema en función de definirlos medios de comunicación entre ellos, así como sus componentes e interacciones, garantizando que el sistema resultante satisface los requerimientos.

Evaluación de Arquitectura: Es un estudio de factibilidad que pretende detectar posibles riesgos, así como también buscar recomendaciones para contenerlos. Estimar el comportamiento de los atributos ante cambios arquitectónicos, lograr hacer predicciones de diseño.

Atributos de Calidad: Requerimientos adicionales del sistema, que hacen referencia a características que éste debe satisfacer. Las propiedades de un servicio que presta el sistema a sus usuarios.

Escenario: Breve descripción de la interacción de alguno de los involucrados en el desarrollo del sistema con este. Provee un vehículo que permite concretar y entender atributos de calidad. Consta de tres partes: *el estímulo, el contexto y la respuesta.*

ADSL: Es un lenguaje descriptivo de modelado que se focaliza en la estructura de alto nivel de la aplicación. Suministran construcciones para especificar abstracciones arquitectónicas y mecanismos para descomponer un sistema en componentes y conectores, especificando de qué manera estos elementos se combinan para formar configuraciones y definiendo familias de arquitecturas o estilos.