**Universidad de las Ciencias Informáticas**

**Facultad 3**


# API para el desarrollo de aplicaciones clientes de escritorio con el marco de trabajo XEGFORT

**Trabajo de Diploma para optar por el título de Ingeniero en Ciencias Informáticas**

**Autor:**

Rafael Mayor Alberto

**Tutores:**

Ing. Reinier Silverio Figueroa

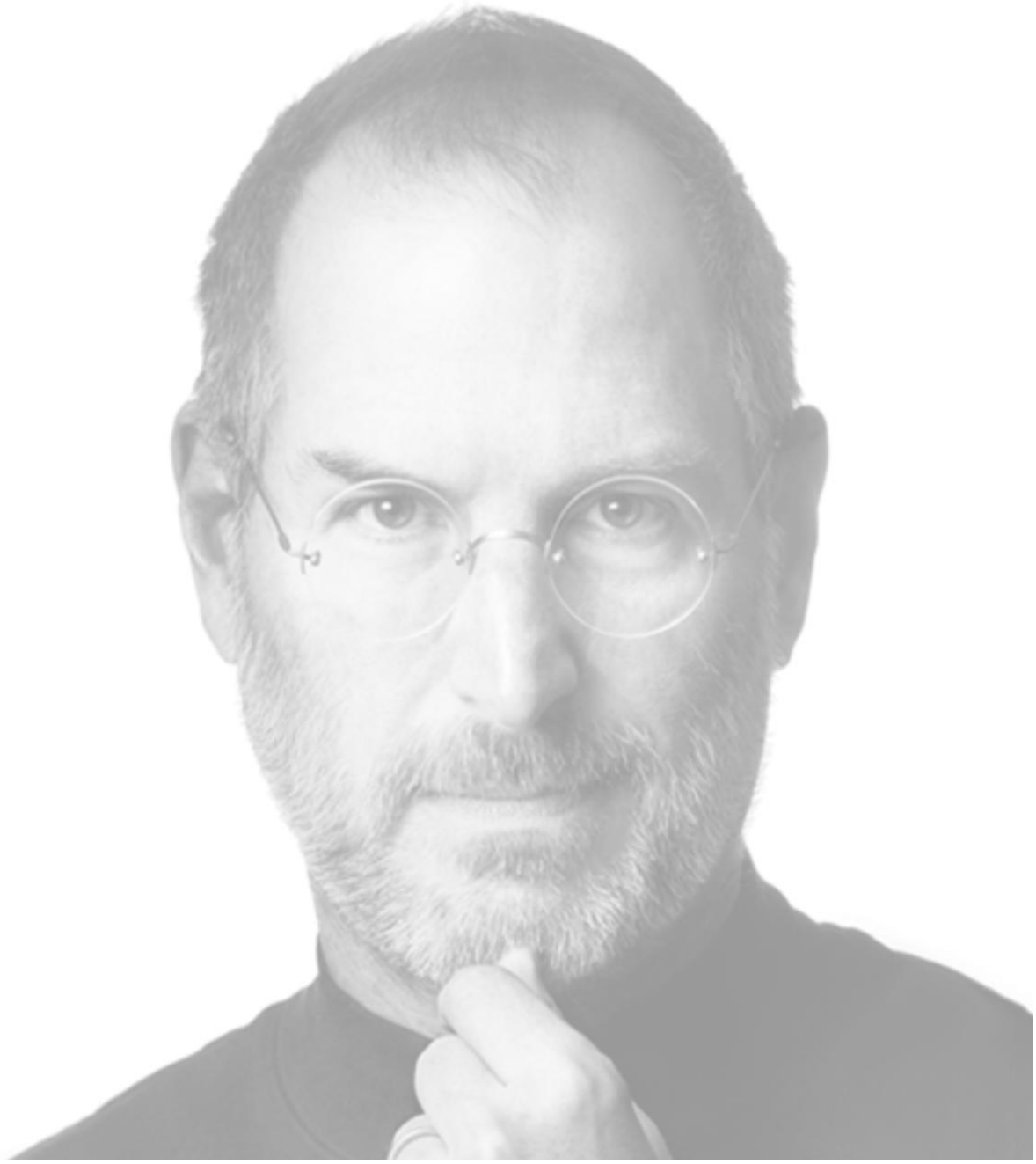
MSc. Yordanis García Leiva

**Cotutor:**

Ing. Juan David Gómez Amador

**Junio, 2017**

**"Año 59 de la Revolución"**



“LA GENTE QUE ESTÁ LO SUFICIENTEMENTE LOCA COMO PARA PENSAR QUE PUEDEN CAMBIAR EL MUNDO, SON LAS QUE LOGRAN HACERLO”

STEVE JOBS
1955-2013

# DECLARACIÓN DE AUTORÍA

Declaramos que somos los únicos autores de este trabajo y autorizamos a la Facultad 3 de la Universidad de las Ciencias Informáticas a hacer uso del mismo en su beneficio.

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

_____

Rafael Mayor Alberto

_____ _____

Ing. Reinier Silverio Figueroa MSc. Yordanis García Leiva

_____

Ing. Juan David Gómez Amador

## AGRADECIMIENTOS

*No es tarea fácil poner en papel a todos los que de una manera u otra estuvieron a mi lado durante este tiempo y mucho menos expresarles cuanto agradezco que así lo hicieran.*

*Me siento inmensamente dichoso y agradezco de la forma más grandiosa posible a mi mamá Marlyn por estar siempre a mi lado, cuidarme, educarme y brindarme todo su amor.*

*A mis papás Roque y Rafael sin su sacrificio nada de esto hubiese sido posible.*

*A mi hermano Julio por confiar siempre en mí y brindarme todo el apoyo posible.*

*A mis abuelas Nery, Cusa y Cora por su amor incondicional.*

*A mis abuelos Rafael Adolfo (Felo), Roque y Felipe porque sé que desde cualquier lugar donde estén desean lo mejor para mí (E.P.D).*

*A Jaime César Mulet por brindarme sus consejos para ser mejor persona cada día, gracias abuelo.*

*A Vivi por escuchar mis problemas y ayudarme a encontrar la solución. Te quiero mucho. A sus padres, Liliana y Luis, por permitirme ser parte de su familia, gracias.*

*A mi tío Sergio, por ser mi ejemplo cada día y querer para mí lo mejor.*

*A Gladys y Aniuta por toda su ayuda.*

*A César, Enzo, Dany y Gaby por lo momentos que hemos pasado juntos, los quiero.*

*A mi tío Alberto por sus buenos consejos.*

*A mis primas Amanda, María, Elena, Leyanis, a mi tía Mariela y tío Ariel por estar pendientes de mí en todo momento.*

*A mis amigos de siempre Javielito, Nano, Yosbel, Carlos y Luis (E.P.D), por los lindos momentos que hemos compartido.*

*A mis tutores que más que tutores son amigos Yordanis, Reinier y Juan David por su apoyo para que esta investigación saliera adelante. Gracias por todo.*

*A mi oponente Reinier Fernández, por sus buenos consejos.*

*A todos mis compañeros de la universidad en especial a mi familia de amigos: Javier, Celso, Adolfo, Marcelo, David, Félix, José Carlos, Alejandro (El flaco) y Rubén.*

*En general a todos mis amigos, discúlpenme que no los mencioné uno a uno, si lo hiciera sería infinita la lista. A todos siempre les tendré en mi corazón.*

**DEDICATORIA**

*A mi familia, en especial a mi mamá Marlyn, mis abuelas Nery, Cusa y Cora, mi hermano Julio y mis papás Roque y Rafael.*

**RESUMEN**

XEGFORT es un marco de trabajo dirigido a la programación en Java, desarrollado en la Universidad de las Ciencias Informáticas. Este utiliza la biblioteca de componentes Swing para el diseño de interfaces gráficas del cliente y una API[1] denominada Acciones. La API Acciones brinda al programador una forma flexible y rápida de desarrollar la lógica de la vista. Es válido señalar que a pesar de las facilidades que brinda el trabajo con Swing y la API Acciones, existen un conjunto de limitantes que complejizan el diseño de determinados componentes de interfaz gráfica. La presente investigación tiene como objetivo desarrollar una API que permita reducir el tiempo en el desarrollo de aplicaciones clientes de escritorio con XEGFORT agilizando el proceso de implementación. El desarrollo de la solución se centra en el empleo de la biblioteca de componentes JavaFX, guiando el proceso a través del uso de la metodología de desarrollo de software Variación del Proceso Unificado Ágil para la Universidad de Ciencias Informáticas y la utilización de tecnologías de código abierto. El resultado de la investigación fue validado utilizando una estrategia de pruebas en los niveles de unidad y aceptación. Además, se establecen un conjunto de criterios que permiten evaluar a través de un antes y un después la relación causa-efecto de la variable independiente sobre la dependiente.

**PALABRAS CLAVES:** API, biblioteca de componentes, Java, marco de trabajo, XEGFORT

[1] Interfaz de programación de aplicaciones

**ABSTRACT**

*XEGFORT is a framework for JAVA programming, developed at the University of Informatics Sciences. It uses the Swing components library to design the client's graphic interfaces and an API named Actions. The Actions API gives the programmer a flexible and fast way to develop the view logic. It is important to say that despite the easiness provided by working with Swing and the Actions API, there is a set of limitations that complicates the design of several graphic interface components. The present research has as a purpose to develop an API that allows to reduce time-consuming when developing desktop client applications with XEGFORT, speeding up the implementation process. The development of the application is focused in the use of JAVAFX components library, leading the process by the usage of the software development methodology Agile Unified Process personalized by the UCI and the usage of abstract code technology. The result of the research was validated using a strategy of software testing at the unit and acceptance levels. Moreover, a set of criteria is established allowing the evaluation using a before and after analysis on the cause-effect relationship that the independent variable has on the dependent one.*

***KEYWORDS:*** *API, components library, Java, framework, XEGFORT*

## ÍNDICE DE CONTENIDOS

## ÍNDICE DE FIGURAS

## ÍNDICE DE TABLAS

## INTRODUCCIÓN

En el ciclo de desarrollo de software, el diseño de interfaces y la realización de la lógica de la vista son pasos fundamentales para la obtención de un producto o solución a la medida con la usabilidad requerida por los clientes. En la actualidad este proceso se realiza con el uso de herramientas, posibilitando un desarrollo ágil y sencillo.

XEGFORT es un marco de trabajo dirigido a la programación en Java, desarrollado en el Centro de Gobierno Electrónico (CEGEL) adjunto a la Facultad 3 de la Universidad de las Ciencias Informáticas (UCI). Este utiliza como herramientas para el diseño de interfaces gráficas del cliente la biblioteca de componentes Swing y una API llamada Acciones que facilita el desarrollo del cliente utilizando el estilo de Asistente. Swing hace posible extender los componentes visuales, con el propósito de orientarlos al trabajo con entidades de negocio. Además, permite implementar configuraciones personalizadas que impactan positivamente en la usabilidad final del software desarrollado a través de XEGFORT. La API Acciones, diseñada en función de las capacidades de Swing, brinda al programador una forma flexible y rápida de desarrollar la lógica de la vista.

Es válido señalar que a pesar de las facilidades que brinda Swing, este tiene un conjunto de limitantes que complejizan en ocasiones el diseño de determinados componentes de interfaz gráfica. Entre estas limitantes se encuentra que: al extender un componente en específico, es necesario relacionar varias interfaces de la biblioteca. Además, para personalizar el comportamiento y la vista de todos los componentes es necesario crear un *Look and Feel*[2]*,* esto implica tener que configurar por separado cada uno de los elementos gráficos de la interfaz, tales como: colores, sombras, formas, letras e íconos y su comportamiento ante las acciones de los usuarios. El uso de Swing complejiza también la implementación de componentes avanzados que tributan a la usabilidad del cliente.

Por otra parte, la API Acciones desempeña el rol de presentador y está condicionada por la forma de trabajo de Swing, en caso de existir un cambio de biblioteca de componentes, se hace necesario modificar su diseño.

Cada uno de los aspectos mencionados anteriormente afectan el tiempo de desarrollo de aplicaciones cliente de escritorio con XEGFORT utilizando la API Acciones.

[2] Aspecto y comportamiento de una interfaz gráfica

La situación problemática antes descrita ha generado el siguiente **problema de investigación:**
¿Cómo reducir el tiempo en el desarrollo de aplicaciones clientes de escritorio con XEGFORT agilizando el proceso de implementación?

**Objeto de estudio:** API para el desarrollo de aplicaciones clientes de escritorio.

**Campo de acción:** API para el desarrollo de aplicaciones clientes de escritorio con XEGFORT.

**Objetivo general:** desarrollar una API que permita reducir el tiempo en el desarrollo de aplicaciones clientes de escritorio con XEGFORT agilizando el proceso de implementación.

**Objetivos específicos:**

- ✓ Elaborar el marco teórico referencial de la investigación relacionada con las API para el desarrollo de aplicaciones clientes de escritorio.
- ✓ Implementar una API que permita el desarrollo de aplicaciones clientes en el marco de trabajo XEGFORT.
- ✓ Validar la solución propuesta aplicando métodos y métricas de validación de software.
- ✓ Validar el cumplimiento de la relación causa-efecto de la variable independiente sobre la variable dependiente de la investigación.

Teniendo como **idea a defender:** con el desarrollo de una API se reduce el tiempo en el desarrollo de aplicaciones clientes de escritorio con XEGFORT agilizando el proceso de implementación.

**Posible resultado:**

- ✓ Obtención de una API que permita reducir el tiempo en el desarrollo de aplicaciones clientes con XEGFORT agilizando el proceso de implementación.

**Tareas de la investigación:**

- ✓ Desarrollo del marco teórico referencial de la investigación en función de los términos relacionados con las API para el desarrollo de aplicaciones clientes de escritorio.
- ✓ Caracterización de cada una de las herramientas a utilizar en la investigación.
- ✓ Identificación de los requisitos funcionales y no funcionales a tener en cuenta en el desarrollo de la solución.

- ✓ Validación de los requisitos funcionales y no funcionales a tener en cuenta en el desarrollo de la solución.
- ✓ Definición de la arquitectura a utilizar en el desarrollo del trabajo.
- ✓ Implementación de la API para el desarrollo de aplicaciones clientes de escritorio con XEGFORT.
- ✓ Realización de pruebas de validación a la solución obtenida, aplicando métodos y métricas.
- ✓ Validación de la relación causa-efecto de la variable independiente sobre la dependiente.
- ✓ Realización del informe final de la investigación.

**Métodos Teóricos:**

Para el cumplimiento de estas tareas se utilizarán los siguientes métodos teóricos:

- ✓ **Analítico-sintético:** este método permitió luego del análisis de las características del marco de trabajo XEGFORT y las herramientas para el diseño de interfaces gráficas del cliente para el desarrollo en Java, identificar JavaFX como la biblioteca de componente a utilizar en el desarrollo de la API.
- ✓ **Análisis histórico-lógico:** se utilizó en el estudio de herramientas para el diseño de interfaces gráficas para el desarrollo en Java.
- ✓ **Modelación:** se utilizó en el proceso de modelación del diseño de clases.

**Métodos Empíricos:**

Para el cumplimiento de estas tareas se utilizó el siguiente método empírico:

- ✓ **Entrevista:** permitió obtener información concerniente a las experiencias de la utilización del marco de trabajo XEGFORT, así como las principales deficiencias en el desarrollo de las interfaces gráficas y desarrollo de la lógica de la vista.
- ✓ **Medición:** permitió la aplicación de métricas y estándares para obtener información cuantitativa acerca de las propiedades o cualidades de los elementos de estudio.

**Estructuración del contenido:**

El contenido de este trabajo consta de tres capítulos, definidos de la siguiente forma:

**Capítulo 1: Fundamentación teórica.**

Se brinda una descripción general de los conocimientos tratados, así como, el estudio y análisis de algunas herramientas que existen para el diseño de GUI[3]. Además, se analiza la metodología de desarrollo de software, las herramientas, lenguajes de programación, patrones de diseño y estrategia de pruebas empleadas en la implementación y validación de la API propuesta en la presente investigación.

**Capítulo 2: Descripción de la solución informática.**

En este capítulo se describe la solución propuesta aplicando las disciplinas: Requisitos, Análisis y diseño e Implementación, definidas por la metodología seleccionada para guiar el proceso de desarrollo. Entre los contenidos analizados se encuentran el proceso de captura y validación de los requisitos, obteniéndose los requerimientos funcionales (RF) y no funcionales (RnF) con los que debe cumplir la API propuesta en la presente investigación. Además, se muestra la arquitectura de la API y del marco de trabajo XEGFORT, así como los patrones de diseño aplicados. Por otra parte, se exponen los estándares de codificación utilizados durante la implementación.

**Capítulo 3: Validación de la solución informática.**

El capítulo contiene los elementos que forman parte de la validación de la solución. En él se exponen los resultados de la aplicación de métricas para evaluar el diseño y los niveles de pruebas definidos para la validación del software. La evaluación del diseño se realiza utilizando las métricas relaciones entre clases y tamaño operacional de las clases. Por otra parte, el desarrollo de las pruebas de software se centra en los niveles de unidad y aceptación. También, se muestran los resultados obtenidos en la comprobación de la relación causa-efecto entre las variables definidas en el diseño metodológico de la presente investigación.

[3] Interfaz gráfica de usuario (Carrillo, y otros, 2005)

# CAPÍTULO 1: FUNDAMENTACIÓN TEÓRICA

## 1.1. Introducción

En el presente capítulo se expone un análisis sobre los principales conceptos relacionados con el desarrollo de aplicaciones de escritorio desarrolladas en java y las bibliotecas de componentes más utilizadas para el desarrollo con el lenguaje de programación antes mencionado. Presentando un acercamiento a la situación actual de estas, haciendo énfasis en las características, ventajas y desventajas que las distinguen. Además, se realiza una breve descripción de la metodología, patrones de diseño, pruebas y herramientas de desarrollo a utilizar durante la implementación y validación de la API propuesta en la presente investigación.

## 1.2. Conceptos fundamentales asociados al dominio del problema

Para una mejor comprensión del tema de investigación, es necesario dominar los siguientes conceptos:

**Interfaz gráfica de usuario (GUI)**: constituye una pieza fundamental de cualquier producto de software. Su surgimiento se remonta al año 1973, cuando fueron realizados los primeros esbozos de lo que a la postre serían las primeras GUI (Carrillo, y otros, 2005).

En las interfaces gráfica de usuario generalmente las acciones se realizan mediante manipulación directa, para facilitar la interacción del usuario con la computadora. Surge como evolución de las interfaces de línea de comandos que se usaban para operar los primeros sistemas y es pieza fundamental en un entorno gráfico. Como ejemplos de interfaz gráfica de usuario, cabe citar los entornos de escritorio Windows, el X-Window de GNU/Linux o el de Mac OS X (Borja , y otros, 2017).

En el contexto del proceso de interacción persona-computadora, la interfaz gráfica de usuario es el artefacto tecnológico de un sistema interactivo que posibilita, a través del uso y la representación del lenguaje visual, una interacción amigable con un sistema informático (Carrillo, y otros, 2005).

**Interfaz de programación de aplicaciones o API (del inglés *Application Programming Interface*)**: es el conjunto de funciones y procedimientos (o métodos, en la programación

orientada a objetos) que ofrece cierta biblioteca para ser utilizado por otro software como una capa de abstracción. Son usados generalmente en las bibliotecas (Alejandra, 2017).

Una interfaz de programación representa una interfaz de comunicación entre componentes de software. Se trata del conjunto de llamadas a ciertas bibliotecas que ofrecen acceso a ciertos servicios desde los procesos y representa un método para conseguir abstracción en la programación, generalmente (aunque no necesariamente) entre los niveles o capas inferiores y los superiores del software. Uno de los principales propósitos de una API consiste en proporcionar un conjunto de funciones de uso general, por ejemplo, para dibujar ventanas o iconos en la pantalla. De esta forma, los programadores se benefician de las ventajas de la API haciendo uso de su funcionalidad, evitándose el trabajo de programar todo desde el principio. Las API asimismo son abstractas: el software que proporciona una cierta API generalmente es llamado la implementación de esa API (Alejandra, 2017).

Ventajas

- ✓ Una buena API hace más fácil el trabajo de desarrollo de un programa, ya que debe proveer todos los bloques para construirlo. El programador lo único que hace es poner todos los bloques juntos.
- ✓ API está diseñado especialmente para los programadores, ya que garantiza que todos los programas que utilizan API, tendrán interfaces similares. Asimismo, esto le facilita al usuario aprender la lógica de nuevos programas.
- ✓ Cuando se realiza una requisición, el servidor llamará a la API, brindando la ventaja de disponer de una mayor cantidad de servicios (Mora, 2002).

**Marco de trabajo (Framework):** es un conjunto de componentes físicos y lógicos estructurados de tal forma que permiten ser reutilizados en el diseño y desarrollo de nuevos sistemas de información. Es una estructura de soporte definida en la cual otro proyecto de software puede ser organizado y desarrollado, a partir de una estructura software compuesta de componentes personalizables e intercambiables para el desarrollo de una aplicación. Contienen patrones y buenas prácticas que apoyan el desarrollo de un producto y un proceso con calidad (Guerrero, y otros, 2014).

Por otra parte, el autor Ralph Johnson (1988) define que un marco de trabajo es una aplicación reusable y semicompleta que puede ser utilizada para producir aplicaciones

personalizadas o específicas y proporciona una plantilla extensible para el desarrollo de aplicaciones.

## 1.3. Tendencias actuales

El lenguaje de programación Java fue originalmente desarrollado por James Gosling de Sun Microsystems (la cual fue adquirida por la compañía Oracle) y publicado en 1995 como un componente fundamental de la plataforma Java de Sun Microsystems. La apariencia externa de Java ha ido desarrollándose guiado por tres bibliotecas de componentes *Abstract Windows Toolkit, Swing* y *JavaFx* (Expósito, 2006).

*AWT* (*Abstract Window Toolkit*): es un kit de herramientas de gráficos, interfaz de usuario, y sistema de ventanas independiente de la plataforma original de Java, para el desarrollo de interfaces era muy básico, quizás más que suficiente para algunos, pero para otros era demasiado elemental. Las aplicaciones Java eran muy planas, fácilmente reconocibles por las particularidades de su apariencia (Expósito, 2006).

*Swing*: es un conjunto de clases desarrolladas por primera vez para Java 1.2 (el llamado Java 2), para mejorar el anterior paquete (*AWT*). Los componentes de GUI de *Swing* permiten especificar una apariencia visual uniforme para una aplicación a través de todas las plataformas, o para usar la apariencia visual personalizada de cada plataforma (Northover, y otros, 2015).

Por otra parte, *JavaFX* es una plataforma que permite a los desarrolladores crear e implementar fácilmente aplicaciones de internet enriquecidas (*RIA*) que se comportan de la misma forma en distintas plataformas. JavaFX amplía la potencia de Java, permitiendo a los desarrolladores utilizar cualquier biblioteca de Java en aplicaciones *JavaFX*. Los desarrolladores pueden ampliar sus capacidades en Java y utilizar la tecnología de presentación que *JavaFX* proporciona para crear experiencias visuales que resulten atractivas (Weaver, 2007).

En la Tabla 1 se muestra una comparación entre las tres bibliotecas de componentes de Java, teniendo en cuenta las principales características que debe cumplir la biblioteca elegida para el desarrollo de la API propuesta en la presente investigación.

**Tabla 1.** Comparación de las bibliotecas de componentes de Java

| Características | AWT | SWING | JAVAFX |
|---|---|---|---|
| Apariencia visual | Bajo | Medio | Alto |
| Independencia de plataforma | No | Si | Si |
| Soporte para animaciones | No | No | Si |
| Soporte para interfaces *touch*[4] | No | No | Si |
| Componentes avanzados | Si | Si | Si |
| Cantidad alta de interfaces para implementar una personalización | Si | Si | No |
| Personalización mediante Hojas de Estilo en Cascada | No | Si | Si |

**Nota:** Fuente elaboración propia.

Después de analizar cada uno de estos elementos, se decide utilizar la biblioteca de componentes *JavaFX*, teniendo en cuenta que esta tecnología está escrita como un API de Java y por tanto puede integrarse a cualquier aplicación escrita en este lenguaje. Además, la apariencia de las aplicaciones de JavaFX se puede personalizar utilizando hojas de estilo en cascada (CSS), una tecnología que separa la apariencia y estilo de la lógica de la aplicación, para que los desarrolladores puedan centrarse en el código y los diseñadores gráficos pueden personalizar fácilmente el aspecto y el estilo de la aplicación a través de CSS. También se elige esta biblioteca de componentes puesto que presenta soporte para sofisticadas animaciones que pueden ser aplicadas a cualquier componente, lo que permite realizar una aplicación que agrade a la vista del usuario final. Por último, el soporte para

[4] Soporte para dispositivos táctiles.

dispositivos móviles hace a esta biblioteca superior a las demás, ya que en la actualidad la gran mayoría de los dispositivos electrónicos presentan esta funcionalidad.

## 1.4. Metodología de Desarrollo de Software

El desarrollo de software no es una tarea sencilla, por mucho tiempo esta labor se ha llevado adelante sin una metodología definida. Algunos autores definen una metodología como una colección de procedimientos, técnicas, herramientas y documentos auxiliares que ayudan a los desarrolladores de software en sus esfuerzos por implementar nuevos sistemas de información (Gómez, 2014).

Desarrollar un buen software depende de un gran número de actividades y etapas, donde el impacto de elegir la metodología para un equipo en un determinado proyecto es trascendental para el éxito del producto. Según la filosofía de desarrollo se pueden clasificar las metodologías en dos grupos. Las metodologías tradicionales, que se basan en una fuerte planificación durante todo el desarrollo, y las metodologías ágiles, en las que el desarrollo de software es incremental, cooperativo, sencillo y adaptado (Burback, 1998).

Al no existir una metodología de desarrollo software universal, y como toda metodología debe ser adaptada a las características de cada proyecto. La UCI toma la iniciativa de hacer una variación de la metodología Proceso Unificado Ágil (AUP por sus siglas en inglés de *Agile Unified Process*), de forma tal que se adapte al ciclo de vida definido para la actividad productiva de la entidad (Rodríguez, 2015). Teniendo en cuenta que la presente investigación responde a uno de los proyectos de la red de centros de la UCI, se decide utilizar esta metodología para guiar el proceso de desarrollo de la API propuesta.

### 1.4.1. Metodología de desarrollo Variación de AUP para la UCI

La metodología de desarrollo de software Variación de AUP para la UCI tiene como propósito, que todos los proyectos de la UCI converjan hacia el uso de una sola metodología (Rodríguez, 2015).

**Fases de Variación de AUP para la UCI**

La metodología Variación de AUP para la UCI está forma por tres fases, (Inicio, Ejecución y Cierre) para el ciclo de vida de los proyectos de la universidad, las cuales contienen las características de las cuatro fases (Inicio, Elaboración, Construcción y Transición)

propuestas en AUP. Las características de las fases de la metodología de la universidad son (Rodríguez, 2015):

- ✓ **Inicio**: durante el inicio del proyecto se llevan a cabo las actividades relacionadas con la planeación del proyecto. En esta fase se realiza un estudio inicial de la organización cliente que permite obtener información fundamental acerca del alcance del proyecto, realizar estimaciones de tiempo, esfuerzo y costo y decidir si se ejecuta o no el proyecto.
- ✓ **Ejecución**: en esta fase se ejecutan las actividades requeridas para desarrollar el software, incluyendo el ajuste de los planes del proyecto considerando los requisitos y la arquitectura. Durante el desarrollo se modela el negocio, se obtienen los requisitos, se elaboran la arquitectura y el diseño, se implementa y se libera el producto. Durante esta fase el software es transferido al ambiente de los usuarios finales o entregado al cliente junto con la documentación. Además, en esta transición se capacita a los usuarios finales sobre la utilización de la aplicación.
- ✓ **Cierre**: en esta fase se analizan tanto los resultados del proyecto como su ejecución y se realizan las actividades formales de cierre del proyecto.

**Disciplinas de Variación de AUP para la UCI**

AUP propone 7 disciplinas (Modelo, Implementación, Prueba, Despliegue, Gestión de configuración, Gestión de proyecto y Entorno). En el caso de la variación para la UCI, propone un total de 7 disciplinas también, pero a un nivel más atómico que el definido en AUP, siendo estas: Modelado de Negocio, Requisitos, Análisis y diseño, Implementación, Pruebas internas, Pruebas de liberación, Pruebas de Aceptación y las restantes tres disciplinas de AUP se cubren con las áreas de proceso que define el Modelo Integrado de Capacidad y Madurez para Desarrollo en su Versión 1.3 (CMMI-DEV v1.3 por su siglas en ingles de *Capability Maturity Model Integration for Development, version 1.3*) para el nivel 2, las cuales serían Gestión de la configuración, Planeación de proyecto y Monitoreo y control de proyecto (Rodríguez, 2015).

**Escenarios para la disciplina Requisitos**

A partir de que el Modelado de negocio propone tres variantes a utilizar en los proyectos: Casos de Uso del Negocio (CUN), Descripción del proceso de negocio (DPN) o Modelo

Conceptual (MC) y existen tres formas de encapsular los requisitos: Casos de Uso del Sistema (CUS), Historias de Usuario (HU) y Descripción de requisitos por proceso (DRP), surgen cuatro escenarios para modelar el sistema en los proyectos (Rodríguez, 2015), manteniendo en dos de ellos el MC, quedando de la siguiente forma:

- ✓ **Escenario No.1:** proyectos que modelen el negocio con CUN solo pueden modelar el sistema con CUS.

**CUN + MC = CUS**

- ✓ **Escenario No.2:** proyectos que modelen el negocio con MC solo pueden modelar el sistema con CUS.

**MC = CUS**

- ✓ **Escenario No.3:** proyectos que modelen el negocio con DPN solo pueden modelar el sistema con DRP.

**DPM + MC =DRP**

- ✓ **Escenario No.4:** proyectos que no modelen negocio solo pueden modelar el sistema con HU.

**HU**

**Características por escenario**

**Escenario No.1**: aplica a los proyectos que hayan evaluado el negocio a informatizar y como resultado obtengan que puedan modelar una serie de interacciones entre los trabajadores del negocio/actores del sistema (usuario), similar a una llamada y respuesta respectivamente, donde la atención se centra en cómo el usuario va a utilizar el sistema. Es necesario que se tenga claro por el proyecto que los Casos de uso del negocio (CUN) muestran como los procesos son llevados a cabo por personas y los activos de la organización (Rodríguez, 2015).

**Escenario No.2**: aplica a los proyectos que hayan evaluado el negocio a informatizar y como resultado obtengan que no es necesario incluir las responsabilidades de las personas que ejecutan las actividades, de esta forma modelarían exclusivamente los conceptos fundamentales del negocio. Se recomienda este escenario para proyectos donde el objetivo primario es la gestión y presentación de información (Rodríguez, 2015).

**Escenario No.3**: aplica a los proyectos que hayan evaluado el negocio a informatizar y como resultado obtengan un negocio con procesos muy complejos, independientes de las personas que los manejan y ejecutan, proporcionando objetividad, solidez, y su continuidad. Se debe tener presente que este escenario es muy conveniente si se desea representar una gran cantidad de niveles de detalles y la relaciones entre los procesos identificados (Rodríguez, 2015).

**Escenario No.4**: aplica a los proyectos que no hayan modelado un negocio. El cliente estará siempre acompañando al equipo de desarrollo para convenir los detalles de los requisitos y así poder implementarlos, probarlos y validarlos. Se recomienda en proyectos no muy extensos, ya que una historia de usuario (HU) no debe poseer demasiada información (Rodríguez, 2015). La HU constituye el artefacto a generar en este escenario.

Las disciplinas definidas en la variación de AUP para la UCI se desarrollan en la Fase de Ejecución, de ahí que en la misma se realicen Iteraciones y se obtengan resultados incrementales, ver Figura 1.

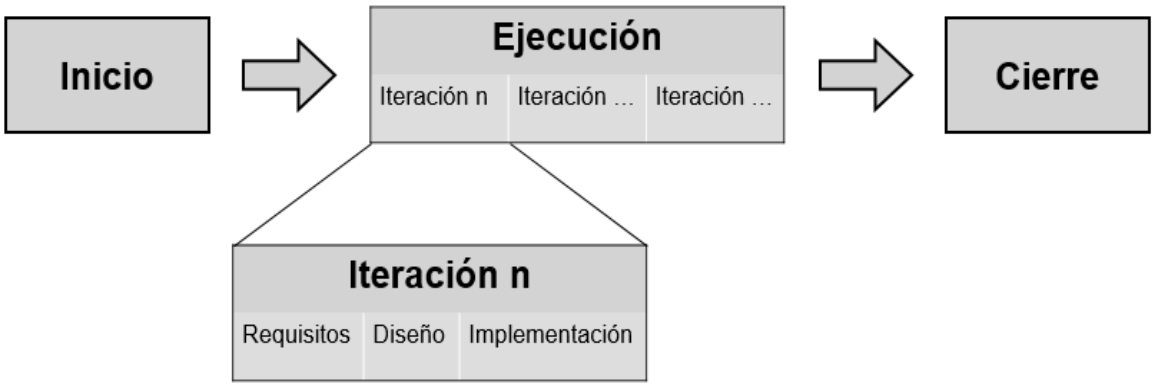



**Figura 1**. Fases e Iteraciones de la variación de AUP para la UCI

**Fuente:** (Rodríguez, 2015).

Para el desarrollo de la solución propuesta, en la disciplina de Requisitos el autor de la presente investigación decide utilizar el escenario número 4. Esta selección se realiza, teniendo en cuenta, que el proyecto está bien definido, el cliente en todo momento estará junto al equipo de desarrollo para convenir los detalles de los requisitos. Además, a pesar de la complejidad mostrada por el proyecto, este no es extenso, posibilitando el uso de las Historias de Usuario.

## 1.5. Herramientas de Ingeniería del Software Asistidas por Computadoras

Las herramientas de Ingeniería de Software Asistida por Computadora (CASE, por sus siglas en inglés de *Computer Aided Software Engineering*) se pueden definir como un conjunto de programas y ayudas que dan asistencia a los analistas, desarrolladores e ingenieros de software, durante el ciclo de vida de un software, proporcionándole un aumento en su productividad y logrando un mayor ahorro de tiempo (Ambler, 2017). Para el modelado de los artefactos generados durante el desarrollo de la presente investigación se decide utilizar la herramienta Visual Paradigm.

Visual Paradigm: es una herramienta que soporta el ciclo de vida completo de desarrollo de un software: análisis y diseño orientado a objetos, construcción, pruebas y despliegue. Permite representar gráficamente varios diagramas y facilita la generación de código. Es una herramienta multiplataforma, fácil de instalar y utilizar (Visual Paradigm, 2017).

Características:

- ✓ Soporte de UML.
- ✓ Diagramas de Procesos de Negocio.
- ✓ Ingeniería inversa.
- ✓ Modelo a código, diagrama a código.
- ✓ Diagramas de flujo de datos.
- ✓ Soporte ORM.
- ✓ Generación de bases de datos.
- ✓ Transformación de diagramas de Entidad-Relación en tablas de base de datos.
- ✓ Distribución automática de diagramas.

A partir de los elementos antes expuestos el autor de la presente investigación decide utilizar Visual Paradigm en su versión 8.0, teniendo en cuenta que esta herramienta propicia un conjunto de funcionalidades para el desarrollo de programas informáticos, desde la planificación, pasando por el análisis y el diseño, hasta la generación del código fuente de

los programas y la documentación. Además, se tuvo en cuenta que la UCI posee una licencia para su uso.

## 1.6. Lenguajes de programación

Los lenguajes de programación son lenguajes artificiales que pueden utilizarse para definir una secuencia de instrucciones para su procesamiento por una computadora. Son herramientas que permiten crear programas y software. Los lenguajes de programación están formados por un conjunto de símbolos, reglas sintácticas y semánticas que definen su estructura, además el significado de sus elementos y expresiones. Son utilizados para controlar el comportamiento lógico de una máquina y para expresar algoritmos con precisión. Estos lenguajes permiten ser leídos y escritos por personas y a su vez resultan independientes del modelo de computadora a utilizar. Existen varios lenguajes de programación tales como: Python, Java y C++ (Joyanes, 2013).

Java: es un lenguaje de programación de propósito general, concurrente, orientado a objetos que fue diseñado específicamente para tener tan pocas dependencias de implementación como fuera posible. Su intención es permitir que los desarrolladores de aplicaciones escriban el programa una vez y lo ejecuten en cualquier dispositivo (conocido en inglés como WORA, o "*write once*, *run anywhere*"), lo que quiere decir que el código que es ejecutado en una plataforma no tiene que ser recompilado para correr en otra (Joyanes, 2013).

Java es la base para prácticamente todos los tipos de aplicaciones de red, además del estándar global para desarrollar y distribuir aplicaciones móviles y embebidas, juegos, contenido basado en web y software de empresa. Con más de 9 millones de desarrolladores en todo el mundo, Java le permite desarrollar, implementar y utilizar de forma eficaz interesantes aplicaciones y servicios (Expósito, 2006).

Para realizar la solución informática propuesta, se utiliza Java en su versión 1.8.60 como lenguaje de programación puesto que este es el lenguaje en que está desarrollado el marco de trabajo XEGFORT. Además, es uno de los lenguajes de programación más utilizados en la actualidad, debido a su arquitectura neutral, simplicidad en la curva de aprendizaje, seguridad y portabilidad.

## 1.7. Herramienta de gestión y configuración de proyectos Java

Maven es una herramienta *open source* para administrar proyectos de software creada por Jason van Zyl, de Sonatype. Administrar, se refiere a gestionar el ciclo de vida desde la creación de un proyecto en un lenguaje dado, hasta la generación de un binario que pueda distribuirse con el proyecto (Camacho, 2013).

Esta herramienta es similar en funcionalidad a Apache Ant (la herramienta de compilación más usada en el mundo Java), pero tiene un modelo de configuración de construcción más simple, basado en un formato de lenguaje de marcado extensible (XML, por sus siglas en inglés de *eXtensible Markup Language*). Maven utiliza un modelo de objetos de proyectos (POM, por sus siglas en inglés de *Project Object Model*) para describir el proyecto de software a construir, sus dependencias de otros módulos y componentes externos, y el orden de construcción de los elementos (Apache, 2015).

Una característica clave de Maven es que está listo para ser usado en red. El motor incluido en su núcleo puede dinámicamente descargar *plugins* de un repositorio, el mismo repositorio que provee acceso a muchas versiones de diferentes proyectos *open source* en Java, de Apache y otras organizaciones y desarrolladores (Apache, 2015).

En el caso de la presente investigación se decide utilizar esta herramienta, teniendo en cuenta que es la definida por el equipo de desarrollo del marco de trabajo XEGFORT para administrar su proceso de construcción.

## 1.8. Entorno de Desarrollo Integrado

Un Entorno de Desarrollo Integrado (IDE, por sus siglas en inglés de *Integrated Development Environment*), es un medio de desarrollo que ha sido empaquetado como un programa de aplicación, generalmente está compuesto por un conjunto de herramientas de programación en las que se encuentra el compilador, editor de código, depurador y constructor de interfaces gráficas. Este proporciona un marco de trabajo amigable para diversos lenguajes de programación (Martínez, 2008).

Netbeans: es un entorno de desarrollo gratuito y de código abierto. Permite el uso de un amplio rango de tecnologías de desarrollo tanto para escritorio, como aplicaciones web, o para dispositivos móviles. Da soporte a las siguientes tecnologías: Java, PHP, Groovy,

C/C++ y HTML5. Además puede instalarse en varios sistemas operativos: Windows, Linux o Mac OS (Netbeans.org, 2016).

Se selecciona para el desarrollo de la solución informática propuesta, la herramienta Netbeans en su versión 8.1, teniendo en cuenta que es libre, se integra perfectamente con el lenguaje de programación Java que es el definido para el desarrollo de la solución propuesta. Además, el equipo posee dominio de esta herramienta.

## 1.9. Patrones de diseño

Los desarrolladores con experiencia acumulan un conjunto de principios generales como de soluciones basadas en aplicar ciertos estilos que les guían en la creación de software. Estos principios y estilos, si se codifican con un formato estructurado que describa el problema y la solución, son definidos como “Patrones” (Rojas, 2014).

Los patrones de diseño (*design patterns*) son la base para la búsqueda de soluciones a problemas comunes en el desarrollo de software y otros ámbitos referentes al diseño de interacción o interfaces (Rojas, 2014).

Un patrón de diseño es una solución a un problema de diseño. Para que una solución sea considerada un patrón debe poseer ciertas características, una de ellas es que debe haber comprobado su efectividad resolviendo problemas similares en ocasiones anteriores, otra es que debe ser reusable, lo que significa que es aplicable a diferentes problemas de diseño en distintas circunstancias (Rojas, 2014).

**Objetivos de estos patrones**

Los patrones de diseño pretenden (Visconti, y otros, 2013):

- ✓ Proporcionar catálogos de elementos reusables en el diseño de sistemas software.
- ✓ Evitar la reiteración en la búsqueda de soluciones a problemas ya conocidos y solucionados anteriormente.
- ✓ Formalizar un vocabulario común entre diseñadores.
- ✓ Estandarizar el modo en que se realiza el diseño.
- ✓ Facilitar el aprendizaje de las nuevas generaciones de diseñadores condensando conocimiento ya existente.

Así mismo, no pretenden (Visconti, y otros, 2013):

- ✓ Imponer ciertas alternativas de diseño frente a otras.
- ✓ Eliminar la creatividad inherente al proceso de diseño.
- ✓ No es obligatorio utilizar los patrones siempre, sólo en el caso de tener el mismo problema o similar que soluciona el patrón, siempre teniendo en cuenta que en un caso particular puede no ser aplicable. Abusar o forzar el uso de los patrones puede ser un error.

Se pueden agrupar en dos grandes grupos; los Patrones Generales de Software para Asignación de Responsabilidades (GRASP por sus siglas en ingles de *General Responsibility Assignment Software Patterns*) y los Patrones Banda de los Cuatro (GOF por sus siglas en ingles de *Gang of Four*), encargados de la inicialización, agrupación y comunicación de los objetos.

A continuación, se describen los patrones utilizados en el desarrollo de la API propuesta en la presente investigación:

**Principales patrones GRASP** (Larman, 1999)

- ✓ **Experto en Información:** permite asignar responsabilidades específicas a cumplir, a las clases, de acuerdo con la información que manejan.
- ✓ **Creador:** permite asignar a la clase B la responsabilidad de crear una instancia de la clase A si se cumple alguno de los puntos siguientes:
    - B contiene a A
    - B registra a A
    - B agrega a A
    - B utiliza estrechamente a A
    - B tiene los datos de inicialización de A
- ✓ **Controlador:** permite asignar la responsabilidad de gestionar un mensaje de un evento del sistema a una clase que represente una de estas opciones:
    - Representa el sistema global, dispositivo o un subsistema (controlador de fachada).

- Representa un escenario de caso de uso en el que tiene lugar el evento del sistema (controlador de caso de uso o sesión).

✓ **Bajo Acoplamiento:** el acoplamiento mide la fuerza con que una clase está conectada a otra, de esta forma una clase con bajo acoplamiento debe tener un número mínimo de dependencia con otras clases.

✓ **Alta Cohesión:** se aplica para realizar un diseño que evite contener clases con un alto grado de abstracción, que asuman responsabilidades que podían haber delegado a otros objetos o que tengan responsabilidades complejas.

**Principales patrones GOF**

**Patrones creacionales** (Rojas, 2014)

✓ **Fábrica abstracta (*Abstract Factory*):** permite trabajar con objetos de distintas familias de manera que las familias no se mezclen entre sí y haciendo transparente el tipo de familia concreta que se esté usando.

✓ **Método de fabricación (*Factory Method*):** centraliza en una clase constructora la creación de objetos de un subtipo de un tipo determinado, ocultando al usuario la casuística para elegir el subtipo que crear.

✓ **Instancia única (*Singleton*):** garantiza la existencia de una única instancia para una clase y la creación de un mecanismo de acceso global a dicha instancia.

**Patrones de Comportamiento** (Patrones de diseño, 2006)

✓ **Estrategia (*Strategy*):** permite encapsular algoritmos relacionados en clases y hacerlos intercambiables, teniendo en cuenta que la selección del algoritmo se haga según el objeto que se trate.

La utilización de patrones de diseño es un aspecto que denota la existencia y necesidad de la reutilización. Debido que identifican aspectos claves de la estructura de un diseño que puede ser aplicado en diferentes situaciones. Además, utilizar patrones es una buena práctica teniendo en cuenta que se está desarrollando para incorporar la solución obtenida de la presente investigación al marco de trabajo XEGFORT, el cual debe proveerse de componentes reutilizables.

## 1.10. Pruebas de software

Uno de los elementos principales para certificar la calidad de una aplicación informática lo constituye el resultado de las pruebas que se practican. Un control y una gestión de calidad implementados de forma adecuada, especialmente durante el proceso de desarrollo del software, aumentan la calidad del sistema final, reducen los costos de avance y acortan el tiempo necesario para el desarrollo. Para determinar el nivel de calidad se deben efectuar medidas o pruebas que permitan comprobar el grado de cumplimiento respecto a las especificaciones principales del sistema (Pressman, 2010).

**Objetivo de las pruebas:**

- ✓ Validar y probar los requisitos que debe cumplir el software.
- ✓ Verificar que los requisitos fueron implementados correctamente.
- ✓ Encontrar y documentar los defectos que puedan afectar la calidad del software.
- ✓ Verificar que el software trabaje como fue diseñado.

Teniendo en cuenta la metodología seleccionada para guiar el desarrollo de la API propuesta por la presente investigación, a continuación, se describen las disciplinas de pruebas de software que esta plantea:

- ✓ **Pruebas internas**: son realizadas por sus propios desarrolladores, verificando el resultado de la implementación, probando cada construcción según sea necesario, así como las versiones finales a ser liberadas. Los artefactos necesarios para la realización de estas pruebas son los casos de pruebas (Rodríguez, 2015).
- ✓ **Pruebas de liberación**: son diseñadas y ejecutadas por una entidad certificadora de la calidad externa, a todos los entregables de los proyectos antes de ser entregados al cliente para su aceptación (Rodríguez, 2015).
- ✓ **Pruebas de aceptación**: son las únicas pruebas que son realizadas por los clientes. Consiste en comprobar si el producto está listo para ser implantado para el uso operativo en el entorno del usuario (Rodríguez, 2015).Para la realización de esta tarea se define utilizar el proceso llamado prueba alfa, el cual se lleva a cabo en el sitio del desarrollador por un grupo representativo de usuarios finales; el software se utiliza en

un escenario natural con el desarrollador registrando los errores y problemas de uso detectados por los clientes (Pressman, 2010).

## 1.11. Conclusiones parciales

El desarrollo del marco teórico referencial relacionado con los términos interfaz gráfica de usuario, marco de trabajo e interfaz de programación de aplicaciones, facilita una mejor comprensión del objeto de estudio de la presente investigación. Además, el estudio de las bibliotecas de componentes AWT, Swing y JavaFX, constituye la base para la definición de la API propuesta. Por otra parte, el análisis de las características de la metodología Variación de AUP para la UCI, las distintas herramientas a utilizar, así como los niveles y métodos de pruebas de software, fundamenta la correspondencia de su selección para la implementación y validación de la solución propuesta.

# CAPÍTULO 2: DESCRIPCIÓN DE LA SOLUCIÓN INFORMÁTICA

## 2.1. Introducción

En el presente capítulo se realiza una descripción de la solución a desarrollar, este contiene la información relacionada con la disciplina Requisitos, Análisis y diseño e Implementación. Además, se detallan las tareas llevadas a cabo para la construcción de la solución que da lugar a la presente investigación, estas fueron guiadas por la metodología de desarrollo de software seleccionada. Por otra parte, se reflejan las necesidades del cliente, se define el comportamiento de la solución, las restricciones del diseño concebidas para la construcción de la solución, así como los estándares de codificación a seguir durante la implementación de sus funcionalidades.

## 2.2. Descripción de la solución propuesta

La solución propuesta se centra en la obtención de una API que permite crear aplicaciones clientes de escritorio utilizando la biblioteca de componentes JavaFX y la herramienta de software para la gestión y construcción de proyectos Java, denominada Maven. El producto de software obtenido brinda la posibilidad de iniciar una aplicación JavaFX de manera sencilla. La API permite usar un menú lateral o de ítem[5], ayudando a utilizar la mayor parte del espacio de la aplicación en función del negocio. Su uso hace posible, iniciar un asistente con el propósito de asistir a determinadas entidades de negocio, en caso de estas necesitar más espacio que el presentado por la aplicación para mostrar información, así como tener la posibilidad de pasar a una pantalla siguiente o regresar a la anterior. La API también ayuda a generar diálogos, los cuales constituyen elementos necesarios a la hora de realizar un producto de software.

## 2.3. Requisitos

El esfuerzo principal en la disciplina Requisitos es desarrollar un modelo del sistema que se va a construir y comprende la administración de los requisitos funcionales y no funcionales del producto (Rodríguez, 2015). Teniendo en cuenta que en la presente investigación no se realiza modelado del negocio, el autor de esta tesis decide aplicar la disciplina de requisitos a través del escenario número cuatro, definido en la metodología de

[5] Cada una de las partes individuales que conforman un conjunto.

desarrollo de software seleccionada. En este epígrafe se presentan las Historias de usuario obtenidas.

### 2.3.1. Técnicas para la identificación de requisitos

Para identificar las necesidades del cliente se utilizan técnicas que permiten determinar de manera explícita, documentada, los requisitos implícitos que tiene el cliente. Esta actividad es continua durante el ciclo de desarrollo y combina, en diferentes puntos, diversas técnicas de identificación para obtener la visión más completa de las necesidades del usuario final. Para la captura de los requisitos se utilizaron las siguientes técnicas:

- ✓ **Entrevista:** es de gran utilidad para obtener información cualitativa, requiere seleccionar bien a los entrevistados para obtener la mayor cantidad de información en el menor tiempo posible. Es muy aceptada y permite acercarse al problema de una manera natural (Soulary, y otros, 2010). Esta técnica fue aplicada a los desarrolladores de CEGEL a cargo de la construcción y mantenimiento de XEGFORT. En el Anexo 1, se encuentra la guía de preguntas utilizada en el desarrollo de la entrevista.
- ✓ **Tormentas de ideas**: para la aplicación de esta técnica se realizó un encuentro entre los miembros del equipo de desarrollo y el cliente, donde ambas partes brindaban sus ideas en cuanto a la propuesta de solución. Esta técnica se puede utilizar para identificar un primer conjunto de requisitos en aquellos casos donde no están muy claras las necesidades que hay que cubrir (Soulary, y otros, 2010). En el Anexo 2 , se muestran los acuerdos tomados en el encuentro con el cliente, los cuales están recogidos en la minuta de reunión generada en el intercambio.

### 2.3.2. Especificación de requisitos

Los requisitos constituyen un punto clave en el desarrollo de aplicaciones informáticas. Un gran número de proyectos de software fracasan debido a una mala definición, especificación o administración de requisitos. Factores tales como requisitos incompletos o mal manejo de los cambios de los requisitos, llevan a proyectos completos al fracaso total. Los requisitos se enfocan en las especificaciones de lo que se desea desarrollar y tienen dos clasificaciones: requisitos funcionales y no funcionales. Los requisitos funcionales son declaraciones de los servicios que debe proporcionar el sistema, de la manera en que éste debe reaccionar a entradas particularidades y de cómo se debe comportar en distintas

situaciones y los requisitos no funcionales son restricciones de los servicios o funciones ofrecidos por el sistema (Sommerville, 2005). A continuación, se presentan los requisitos de la API propuesta.

### Requisitos funcionales

**Tabla 2.** Requisitos funcionales

| No. | Nombre | Descripción |
|---|---|---|
| RF1 | Iniciar aplicación cliente. | El programador debe poder iniciar una aplicación cliente a través de un punto de entrada en la API. |
| RF2 | Visualizar un FXML[6] en el marco de la aplicación. | Se debe permitir al desarrollador iniciar una acción dentro del marco de la aplicación sin necesidad de iniciar un asistente. |
| RF3 | Confeccionar un asistente. | Se debe permitir al desarrollador confeccionar un asistente. |
| RF4 | Navegar dentro del asistente. | Se debe permitir al desarrollador navegar dentro del asistente, es decir, poder interactuar con los botones que presente y asignar una funcionalidad a cada uno de estos. |
| RF5 | Confeccionar menú. | Se debe proveer de un menú que se adapte a las necesidades de la aplicación y además que permita utilizar mejor el espacio de la pantalla asignado a mostrar la información. |
| RF6 | Mostrar mensajes. | La API debe proveer de funcionalidades para construir y mostrar mensajes de manera intuitiva. |

**Nota:** Fuente elaboración propia.

[6] Lenguaje de marcado declarativo utilizado para la construcción de interfaces de usuario de aplicaciones JavaFX.

**Requisitos no funcionales**

- ✓ **Requisitos de usabilidad**

  RNF 1: la API debe proveer que los menús generados por esta, no se encuentren siempre visible para aprovechar mejor el espacio de trabajo para mostrar información.

  RNF 2: la API debe ser fácil de aprender a utilizar.

  RNF 3: cada funcionalidad debe brindar información referida a su comportamiento.

  RNF 4: contribuir al trabajo de los programadores que utilicen el marco de trabajo XEGFORT, facilitando el diseño de las aplicaciones creadas con el uso de la API.

  RNF 5: los diálogos que proporcione la API deben ser de fácil acceso para el desarrollador que la utilice.

- ✓ **Requisitos de rendimiento**

  RNF 6: la API no debe demorar más de 6 segundos en realizar sus funcionalidades.

- ✓ **Requisitos de interfaz y apariencia**

  RNF 7: debe ser fácil de configurar el aspecto y estilo de la aplicación.

  RNF 8: los menús presentados por la API deben tener una interfaz amigable.

  RNF 9: el asistente y los diálogos que muestre la API deben ajustarse al estilo de la aplicación.

- ✓ **Requisitos de software**

  RNF 10: se debe tener instalada en la computadora la Máquina Virtual de Java en su versión 8 o superior.

  RNF 11: una aplicación realizada utilizando la API debe ser compatible con los sistemas operativos Windows, Linux y OS X.

- ✓ **Requisitos de hardware**

  RNF 12: para ejecutar una aplicación que utilice la API, la computadora debe contar con las siguientes características:

  - 512 MB de memoria RAM, equivalente o superior.

- Procesador a 3.2 GHz, equivalente o superior.

### 2.3.3. Validación de los requisitos

Con el objetivo de asegurar que el software está de acuerdo con su especificación, se comprueba que el sistema cumple con los requisitos funcionales y no funcionales que se han especificado utilizando las siguientes técnicas de validación de requisitos:

- ✓ **Revisiones formales de los requisitos:** se realizaron revisiones formales de cada requisito, por parte del cliente y el equipo de desarrollo, validando que la interpretación de cada una de las descripciones no sea ambigua, ni presenten omisiones o errores. En el Anexo 3, se evidencian los acuerdos tomados en la reunión de revisiones formales de los requisitos, los cuales están recogidos en la minuta generada en este encuentro.
- ✓ **Construcción de prototipos:** se realizó un modelo ejecutable utilizando la API para que los clientes puedan experimentar con él y determinar si cumple sus necesidades.

Para medir la calidad de la especificación de los requisitos de software se aplicó la métrica Calidad de la Especificación (CE). El empleo de esta métrica permite obtener un alto nivel de entendimiento y precisión de los requisitos, para llegar a ello, se debe primeramente calcular el total de requisitos de software como se muestra a continuación:

**Nr**: total de requisitos de software.

**Nf**: cantidad de requisitos funcionales.

**Nnf**: cantidad de requisitos no funcionales.

$$Nr = Nf + Nnf$$

Sustituyendo los valores en la ecuación, se obtiene:

$$Nr = 6 + 12$$

$$Nr = 18$$

Finalmente, para calcular la Especificidad de los Requisitos (ER) o ausencia de ambigüedad en los mismos se realiza la siguiente operación:

**Nui**: número de requisitos para los cuales todos los revisores tuvieron interpretaciones idénticas.

**Q1 = Nui / Nr**

Para sustituir los valores de las variables en la ecuación se tuvo en cuenta que de los requisitos especificados para el desarrollo del API dos de ellos causaron contradicción (RF1 y RF6) en sus interpretaciones. Por tanto, la variable Q1 obtiene el siguiente valor:

**Q1 = 16 / 18**

**Q1 = 0.89**

Es importante aclarar que mientras más cerca de 1 está el valor de Q1, menor es la ambigüedad. Teniendo en cuenta el resultado anterior, igual a 0.89, se concluye que el 89% de los requisitos es entendible. Los dos requisitos identificados como ambiguos fueron modificados y validados para garantizar su correcta comprensión, llegando al resultado ideal de Q1=1.

### 2.3.4. Historias de usuario

Las historias de usuario (HU) son una forma rápida de administrar los requisitos de los usuarios sin tener que elaborar gran cantidad de documentos formales y sin requerir de mucho tiempo para administrarlos. Las historias de usuario permiten responder rápidamente a los requisitos cambiantes, por lo que una historia de usuario puede tener varios cambios a lo largo de un desarrollo sin afectarse el tiempo (Beck, 2002).

A continuación, se describen dos historias de usuario de prioridad alta en el desarrollo de la API, para consultar el resto ver Anexo 4 .

**Tabla 3.** Historia de usuario Iniciar aplicación cliente

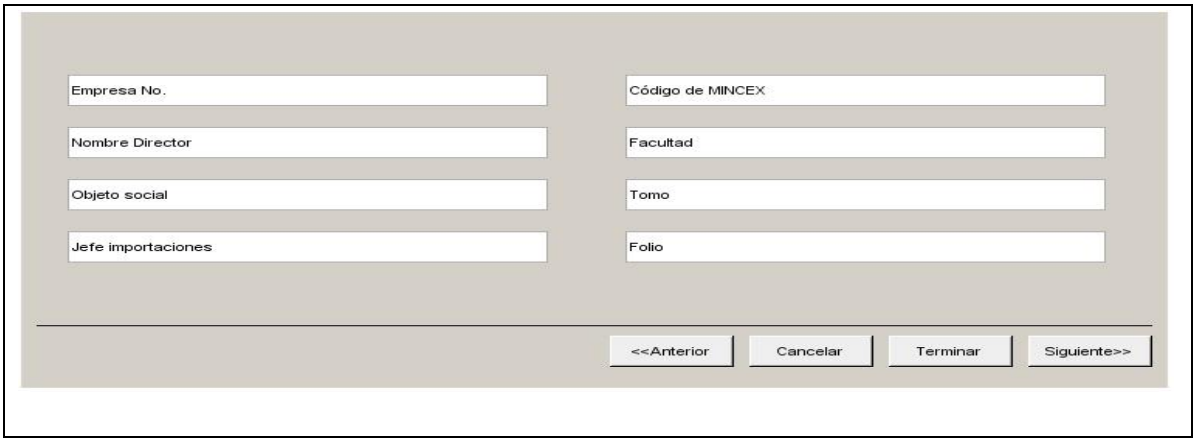
| **Historia de Usuario** | |
|---|---|
| **Número:** 1 | **Nombre de la Historia de Usuario:** Iniciar aplicación cliente. |
| **Programador:** Rafael Mayor Alberto | **Iteración asignada:** 1 |
| **Prioridad:** alta | **Tiempo estimado:** 15 días |

| | |
|---|---|
| **Riesgo en desarrollo:** alto | **Tiempo real:** 15 días |
| **Descripción:** se debe ofrecer funcionalidades para permitir al desarrollador iniciar una aplicación cliente. | |
| **Observaciones:** debe permitir crear un cliente con su CSS correspondiente. | |
| **Prototipo de interfaz:** no aplica | |

**Nota:** Fuente elaboración propia.

**Tabla 4.** Historia de usuario Confeccionar un asistente

| **Historia de Usuario** | |
|---|---|
| **Número:** 3 | **Nombre de la Historia de Usuario:** Confeccionar un asistente. |
| **Programador:** Rafael Mayor Alberto | **Iteración asignada:** 3 |
| **Prioridad:** alta | **Tiempo estimado:** 15 días |
| **Riesgo en desarrollo:** alto | **Tiempo real:** 15 días |
| **Descripción:** se debe permitir al desarrollador confeccionar un asistente. | |
| **Observaciones:** ubicar los botones en la parte inferior del espacio asignado para mostrar el asistente. | |
| **Prototipo de interfaz:** | |




**Nota:** Fuente elaboración propia.

## 2.4. Análisis y diseño

En esta disciplina los requisitos pueden ser refinados y estructurados para conseguir una comprensión más precisa de estos, y una descripción que sea fácil de mantener y ayude a la estructuración del sistema (incluyendo la arquitectura). Además, en esta disciplina se modela el sistema y su forma para que soporte todos los requisitos, incluyendo los requisitos no funcionales. Los modelos desarrollados son más formales y específicos que el de análisis (Rodríguez, 2015).

### 2.4.1. Diseño arquitectónico

La arquitectura de Software consiste en un conjunto de patrones y abstracciones coherentes que proporcionan un marco definido y claro para interactuar con el código fuente del software (Fernández, 2000). Para el desarrollo de la presente investigación se utilizó el patrón arquitectónico Modelo–Vista–Presentador (MVP por sus siglas en inglés de *Model View Presenter*), este es considerado una derivación del Modelo–Vista–Controlador (MVC por sus siglas en inglés de *Model View Controller*). El MVP, separa el modelo del dominio, la presentación y las acciones basadas en la interacción con el usuario en tres capas separadas.

El concepto de este patrón se enfoca en sus tres capas. Por un lado, está la vista, que se encarga de mostrar la información al usuario y de interactuar con él para hacer ciertas operaciones. Por otro, el modelo, que ignorando cómo la información es mostrada al usuario, realiza toda la lógica de las aplicaciones usando las entidades del dominio. Y por

último, el presentador, que es el que "presenta" a ambos actores sin que haya ningún tipo de dependencia entre ellos.

En la Figura 2 se evidencia la aplicación de este patrón en la solución propuesta. El modelo está compuesto por los objetos que conocen y manejan los datos dentro de la aplicación que utilice la API, ejemplo las clases que conforman el modelo del negocio de dicha aplicación. La vista, se encarga de manejar los aspectos visuales, mantiene una referencia a su presentador al cual le delega la responsabilidad del manejo de los eventos, esta es propiciada por la aplicación que utilice la API. El presentador, contiene la lógica para responder a los eventos y manipula el estado de la vista, utiliza el modelo para saber cómo responder a los eventos, además es responsable de establecer y administrar el estado de una vista, esto se evidencia en los componentes *GestorMarcoTrabajo, GeneradorOpciónDiálogo y GeneradorMenú.*

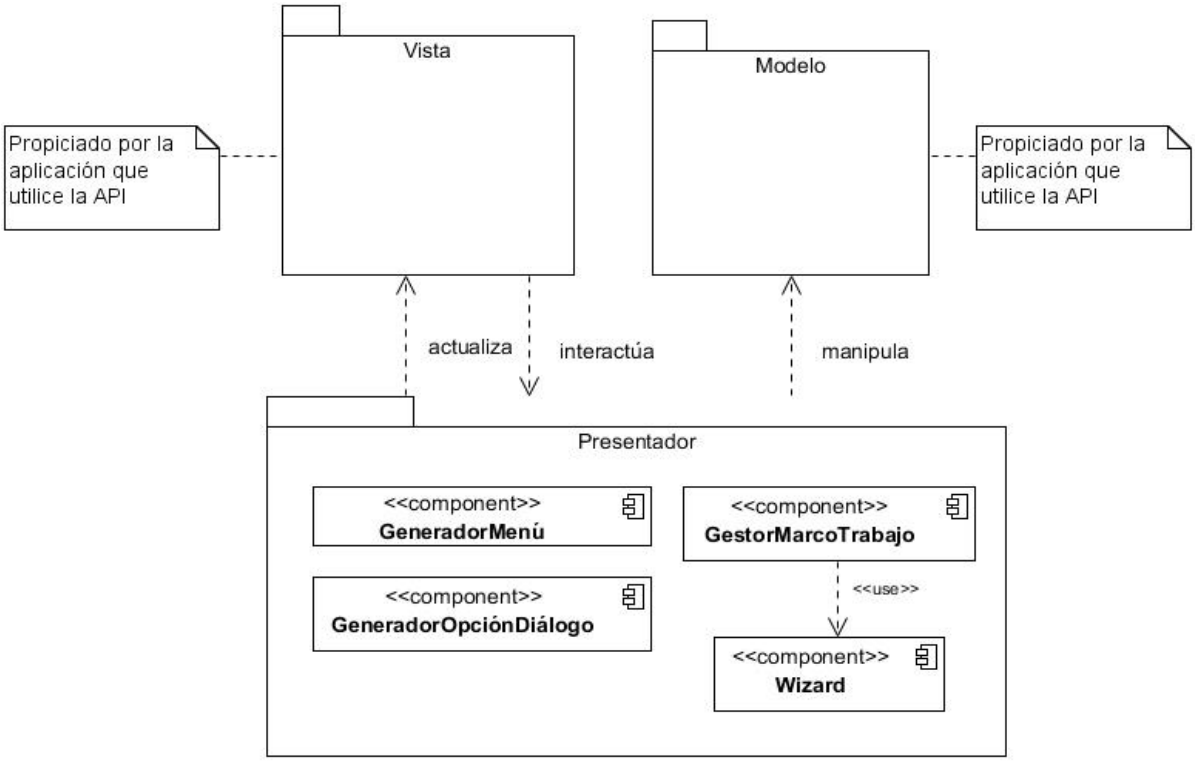



**Figura 2.** Aplicación del patrón Modelo Vista Presentación

**Fuente:** elaboración propia

El marco de trabajo XEGFORT presenta una arquitectura Cliente-Servidor distribuida en N-Capas, la API se ubicará en la capa Presentación del Cliente específicamente sustituyendo los componentes Acciones y Formularios, ver Figura 3.

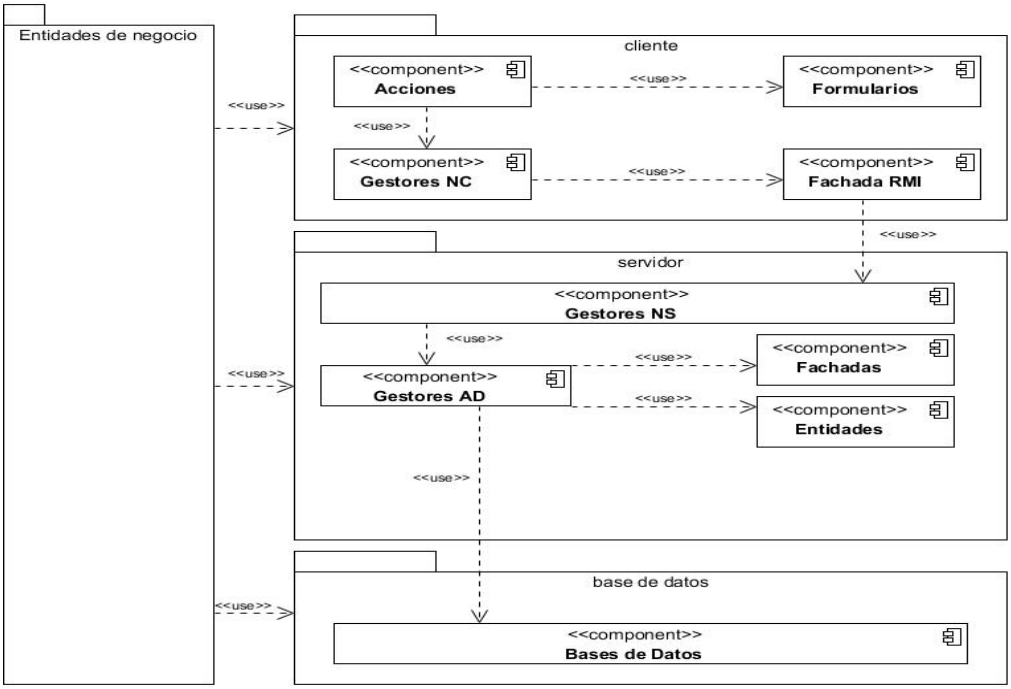



**Figura 3.** Representación de la arquitectura del marco de trabajo XEGFORT.

**Fuente:** elaboración propia.

### 2.4.2. Diagrama de clases

En ingeniería de software, un diagrama de clases en Lenguaje Unificado de Modelado (UML por sus siglas en inglés de *Unified Modeling Language*), es un tipo de diagrama de distribución estática que describe la estructura de un sistema mostrando las clases, sus atributos, operaciones (o métodos), y las relaciones entre los objetos (Gramage, 2016). En el Anexo 5 se observa el diagrama de clases que responde a la API propuesta, agrupado por paquetes.

### 2.4.3. Patrones de diseño

Para el diseño de la API se utilizaron un conjunto de patrones de diseño, los cuales constituyen la base para solucionar problemas comunes en el desarrollo de software. A continuación, se describen los patrones empleados.

**Patrones de diseño GRASP**

- ✓ **Experto:** el uso de este patrón se evidencia en la clase *Wizard*. Esta es la encargada de conformar el asistente, es decir, convertir la vista de cada entidad y sus botones en un solo componente. Ver Figura 4.

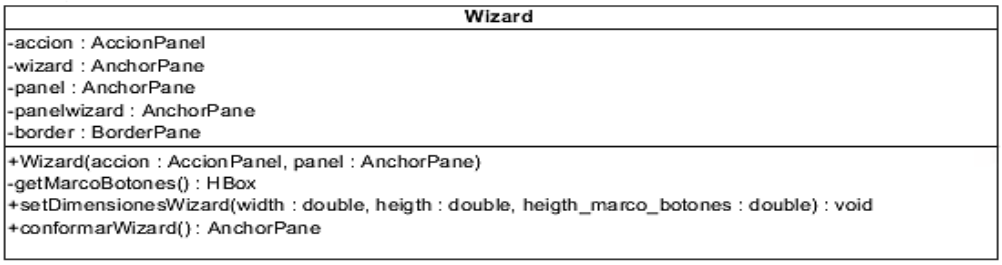



**Figura 4.** Clase Wizard de la API

**Fuente:** elaboración propia.

- ✓ **Creador:** el uso de este patrón se evidencia en la clase *GestorMarcoTrabajo, GeneradorOpciónDiálogo, JFXMenúLateral y GeneradorMenú.* Estas clases tienen la responsabilidad de instanciar objetos para cumplir sus funcionalidades. En la Figura 5, se evidencia el uso de este patrón en la clase *GestorMarcoTrabajo* específicamente en el método *iniciarAccionWizard* cuando instancia un objeto de tipo Wizard para mostrarlo en la aplicación.

```
public void iniciarAccionWizard(String nombreFXML) throws IOException, IllegalArgumentException {
    historialPaneles.clear();
    FXMLLoader fxmlLoader = Loader.getInstancia().CargarFXML(nombreFXML);
    AnchorPane root1 = (AnchorPane) fxmlLoader.load();
    accionActual = fxmlLoader.getController();
    if (iniciaFlujo(accionActual)) {
        this.asistente = new Wizard(accionActual, root1);
        asistente.setDimensionesWizard(panelPrincipal.getPrefWidth(), panelPrincipal.getPrefHeight(), 35);
        limpiarPanel(this.panelPrincipal);
        historialPaneles.put(nombreFXML, accionActual);
        panelPrincipal.getChildren().add(asistente.conformarWizard());
    } else {
        throw new IllegalArgumentException("La clase : " + accionActual.getClass().getName() +
            "  debe contener la anotación iniciarFlujo");
    }

}
```

**Figura 5.** Método iniciarAccionWizard

**Fuente:** elaboración propia.

- ✓ **Controlador:** este patrón se evidencia en la clase *GestorMarcoTrabajo* que se encarga de iniciar una acción y del montaje de una aplicación cliente, ver Figura 6.

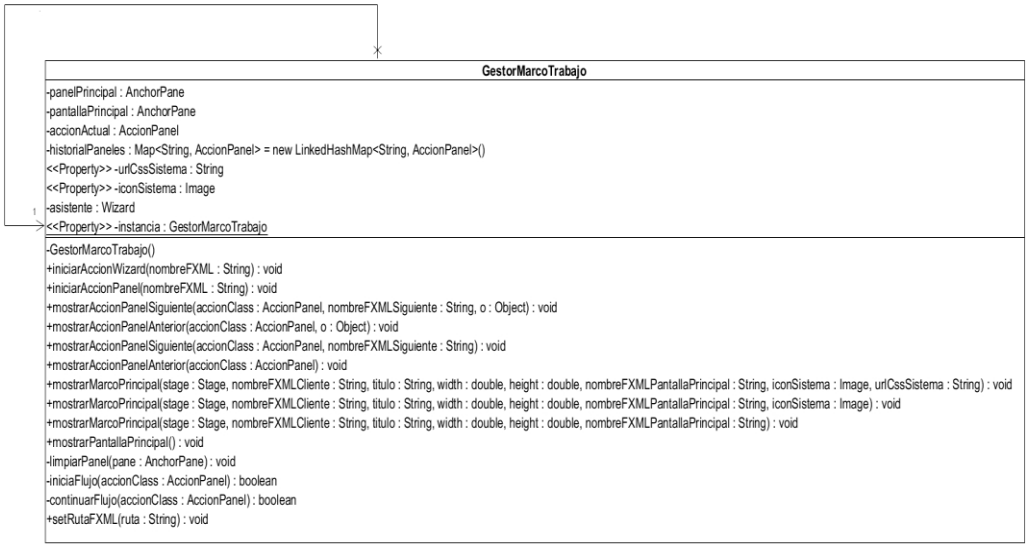



**Figura 6.** Clase GestorMarcoTrabajo de la API

**Fuente:** elaboración propia.

- ✓ **Alta cohesión:** el patrón se evidencia en cada de una de las clases de la API, de tal forma que se elimina la sobrecarga de responsabilidades.
- ✓ **Bajo acoplamiento:** este patrón se tuvo presente debido a la importancia que se le atribuye a realizar un diseño de clases independientes que puedan soportar los cambios de una manera fácil y que a su vez permitan la reutilización. El patrón se evidencia en cada una de las clases diseñadas para la API, teniendo en cuenta la independencia que existe entre estas.

**Patrones de diseño GOF**

- ✓ **Instancia única (*Singleton*):** su uso se aprecia en las clases *Loader* y *GestorMarcoTrabajo*, asegurando que cada clase tenga una sola instancia y propiciando un punto de acceso global a ella. En la Figura 7, se muestra el uso de este patrón en la clase *Loader* de la API.

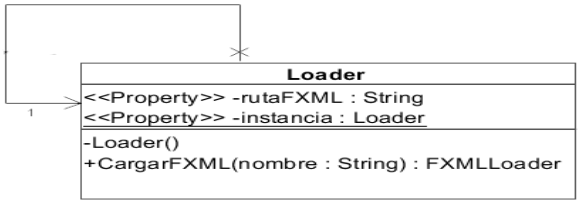



**Figura 7.** Clase Loader de la API

**Fuente:** elaboración propia.

- ✓ **Mediador (*Mediator*):** se evidencia en la clase *GestorMarcoTrabajo*, ver Figura 6. Es la encargada de coordinar la comunicación entre los objetos con el propósito de que todos trabajen conjuntamente, para realizar cada funcionalidad contenida en ella.
- ✓ **Método de fabricación (*Factory Method*):** se puede apreciar en las clases *GeneradorMenú* y *GeneradorOpciónDiálogo*, en la Figura 8, se muestra un fragmento de código donde se muestra el uso de este patrón.

```
public static IDiálogoAlerta crearDialogoError(String titulo, String encabezadoTexto, String contenidoTexto) {
    return new DiálogoAlertas(titulo, encabezadoTexto, contenidoTexto, TipoAlerta.ALERTA_ERROR);
}
```

**Figura 8.** Método crearDialogoError de la clase GeneradorOpciónDiálogo

**Fuente:** elaboración propia.

- ✓ **Estrategia (*Strategy*):** se aprecia en las clases *DiálogoConfirmación, DiálogoReporteExcepción, DiálogoLogin, DiálogoEntradaTexto* y *DiálogoSelección*. En la Figura 9, se muestra el uso de este patrón.

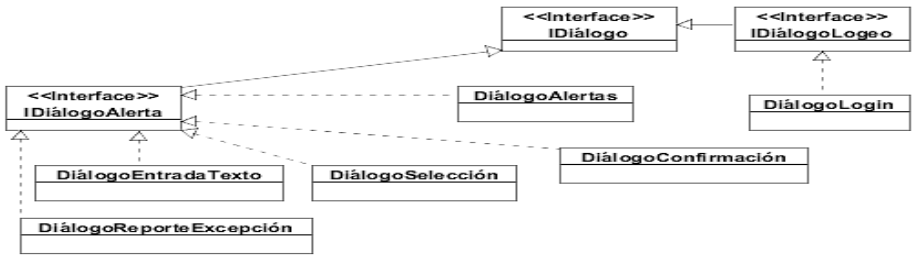



**Figura 9.** Ejemplo del uso del patrón estrategia

**Fuente:** elaboración propia.

## 2.5. Implementación

En esta fase a partir de los resultados del Análisis y el Diseño se construye la solución que da lugar a la presente investigación.

### 2.5.1. Estándares de codificación

El estándar de codificación utilizado en el desarrollo de la API fue definido por el equipo de desarrollo. A continuación, se muestra la aplicación de este.

- ✓ Los nombres de las clases serán con mayúscula, en caso de ser un nombre compuesto las siguientes palabras se escribirán de igual forma, en la Figura 10 se evidencia el uso de este estándar en la clase *GestorMarcoTrabajo*.

```
/**
 * @author xegfort-team
 * @version javafx 1.0
 */
public class GestorMarcoTrabajo {
```

**Figura 10**. Clase GestorMarcoTrabajo

**Fuente:** elaboración propia.

- ✓ Los nombres de los métodos serán con minúscula, en caso de ser un nombre compuesto las siguientes palabras se escribirán con mayúscula, ver Figura 11.

```
public void iniciarAccionWizard(String nombreFXML) throws IOException, IllegalArgumentException
```

**Figura 11.** Empleo del estándar de codificación para los métodos

**Fuente:** elaboración propia.

- ✓ Los identificadores para las variables y los parámetros serán con letras en minúsculas y en caso de ser un nombre compuesto las siguientes palabras se escribirán con mayúscula, en la Figura 12 se evidencia el uso de este estándar de codificación.

```
private Wizard asistente;
private static GestorMarcoTrabajo instancia;
private AnchorPane panelPrincipal;
private AnchorPane pantallaPrincipal;
private AcciónPanel accionActual;
private final Map<String, AcciónPanel> historialPaneles = new LinkedHashMap<String, AcciónPanel>();
private String urlCssSistema;
private Image iconSistema;
private double widthMarcoPrincipal;
private double heightMarcoPrincipal;
```

**Figura 12.** Atributos de la clase *GestorMarcoTrabajo*

**Fuente:** elaboración propia.

- ✓ La clase gestora de la API comienza con el prefijo Gestor y luego el nombre de la clase (*GestorMarcoTrabajo.java*), ver Figura 10.
- ✓ Los nombres de variables o funciones deben ser lo suficientemente descriptivos, sin exceder de 30 caracteres, ver Figura 10, Figura 11 y Figura 12.

✓ Todas las funciones deben tener comentarios, que describan su propósito (*Javadoc*[7]). En la Figura 13, se evidencia el uso de este estándar en el método *iniciarAccionPanel*.

```
/**
 * Inicia un Wizard.
 *
 * @param nombreFXML de tipo {@code String} ,será el nombre del FXML que va
 * a iniciar dicho wizard.El controlador correspondiente a este FXML debe
 * contener la anotación IniciaFlujo e implementar la Interfáz AcciónPanel.
 *
 * @exception IOException cuando el fxml no es encontrado o contiene algún
 * error a la hora de su configuración.
 * @exception IllegalArgumentException cuando el controlador correspondiente
 * al FXML no contiene la anotación IniciaFlujo.
 */
public void iniciarAccionWizard(String nombreFXML) throws IOException, IllegalArgumentException {
    historialPaneles.clear();
    FXMLLoader fxmlLoader = Loader.getInstancia().CargarFXML(nombreFXML);
    AnchorPane root1 = (AnchorPane) fxmlLoader.load();
    accionActual = fxmlLoader.getController();
    if (iniciaFlujo(accionActual)) {
        this.asistente = new Wizard(accionActual, root1);
        asistente.setDimensionesWizard(panelPrincipal.getPrefWidth(), panelPrincipal.getPrefHeight(), 35);
        limpiarPanel(this.panelPrincipal);
        historialPaneles.put(nombreFXML, accionActual);
        panelPrincipal.getChildren().add(asistente.conformarWizard());
    } else {
        throw new IllegalArgumentException("La clase : " + accionActual.getClass().getName() + "
         debe contener la anotación iniciarFlujo");
    }

}
```

**Figura 13.** Ejemplo del uso de javadoc en las funcionalidades de la API

**Fuente:** elaboración propia.

## 2.6. Conclusiones parciales

El empleo de la metodología Variación de AUP para la UCI en función de describir el proceso de desarrollo de la API permitió generar los artefactos necesarios para el desarrollo de la investigación, así como, ajustarse al uso de la metodología definida para el proceso de desarrollo de software en la universidad. Por otra parte, el uso del patrón arquitectónico Modelo–Vista–Presentador y el empleo de patrones de diseño, garantizó obtener una solución con poca dependencia entre clases, flexible al mantenimiento y a la introducción de cambios. Por lo tanto, la API obtenida, facilita el diseño de interfaces de usuario a través de la biblioteca de componentes JavaFX, contribuyendo a reducir el tiempo en el desarrollo de aplicaciones clientes de escritorio con XEGFORT.

[7] Es una utilidad de Oracle para la generación de documentación de interfaz gráficas de aplicaciones en formato HTML a partir de código fuente Java.

# CAPÍTULO 3: VALIDACIÓN DE LA SOLUCIÓN INFORMÁTICA

## 3.1. Introducción

En el presente capítulo se valida el diseño de la solución propuesta a través de la aplicación de métricas. Además, se comprueba la calidad de la implementación desarrollando pruebas de software en los niveles de unidad y aceptación. Por otra parte, se realiza la validación de las variables de la investigación.

## 3.2. Validación del diseño

Con el objetivo de comprobar la calidad del diseño se emplearon las métricas de diseño relaciones entre clases (RC) y tamaño operacional de clase (TOC).

### 3.2.1. Relaciones entre clases (RC)

La métrica RC está dada por el número de relaciones de uso de una clase con otra. El primer paso es evaluar un conjunto de atributos de calidad entre los que se encuentran el acoplamiento, complejidad de mantenimiento y reutilización de cada clase (Pressman, 2002).

A continuación, se exponen los pasos que se llevaron a cabo para aplicar la métrica a todas las clases de la API propuesta en la presente investigación:

1. Determinar la Cantidad de Relaciones de Uso (CRU) que poseen las clases a medir.
2. Calcular el promedio de las CRU.
3. Teniendo en cuenta los valores antes obtenidos se determina la incidencia de los atributos de calidad en cada una de las clases, según los criterios expuestos en la Tabla 5.

**Tabla 5.** Rango de valores para medir la afectación de los atributos de calidad (RC).

| Atributos de calidad | Clasificación | Criterio |
|---|---|---|
| | Ninguna | CRU = 0 |

| | | |
|---|---|---|
| Acoplamiento | Baja | CRU = 1 |
| | Media | CRU = 2 |
| | Alta | CRU > 2 |
| Complejidad de mantenimiento | Baja | CRU <= Promedio |
| | Media | Promedio < CRU < = 2* promedio |
| | Alta | CRU > 2* promedio |
| Reutilización | Baja | CRU > 2* promedio |
| | Media | Promedio < CRU < = 2* promedio |
| | Alta | CRU <= Promedio |
| Cantidad de pruebas | Baja | CRU <= Promedio |
| | Media | Promedio < CRU < = 2* promedio |
| | Alta | CRU > 2* promedio |

**Nota:** Fuente (Lorenz, y otros, 1994).

En la Figura 14 se muestra el resultado de aplicar la métrica RC:

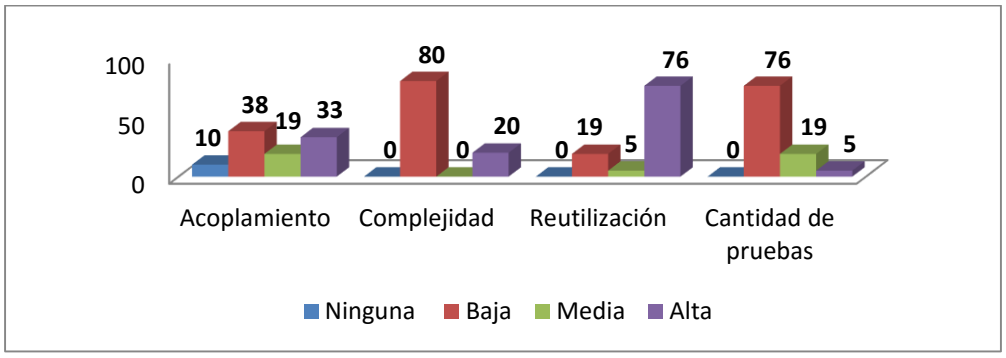



**Figura 14.** Representación en (%) de los resultados de la aplicación de la métrica RC
**Fuente:** elaboración propia.

**Acoplamiento**: según los resultados que se muestran, el 10% de las clases no posee relaciones de uso y el 38 % posee un acoplamiento bajo, el 19% un acoplamiento medio y el 33 % un acoplamiento alto por lo que la mayoría de las clases poseen valores de acoplamiento nulo, bajos y medio, validando una realización correcta del diseño.

**Complejidad de mantenimiento**: según los resultados que se muestran en la figura anterior, el 80% de las clases presentan una complejidad de mantenimiento baja, demostrando que son de fácil soporte.

**Reutilización**: según los resultados que se muestran en la figura, el 76% de las clases tiene un grado alto de reutilización.

**Cantidad de pruebas**: luego de aplicar la métrica se obtuvo que el 76% de las clases poseen un grado bajo de esfuerzo a la hora de realizar cambios, rectificaciones y pruebas de software.

Teniendo en cuenta lo anterior se concluye que el diseño de la API propuesta en la presente investigación alcanzó resultados positivos durante la evaluación de la métrica. Se obtuvo un 48% de bajo acoplamiento, 80% de baja complejidad de mantenimiento, el 76% presenta un bajo grado de esfuerzo destinado a pruebas y una reutilización alta. Estos datos muestran que las clases poseen bajo acoplamiento, complejidad de mantenimiento, esfuerzo para realizar pruebas y en efecto presentan un alto grado de reutilización.

### 3.2.2. Tamaño Operacional de Clases (TOC)

La métrica TOC fue aplicada a cada una de las clases del diseño con el objetivo de medir la calidad de las mismas con respecto a su grado de responsabilidad, complejidad de implementación y reutilización (Pressman, 2002).

A continuación, se explican los pasos que se llevaron a cabo para aplicar la métrica:

1. Cálculo del umbral. El umbral se toma del tamaño general de una clase que se determina sumando todas las operaciones que posee.
2. Calcular el promedio de los umbrales.
3. Teniendo en cuenta los valores antes obtenidos se determina la incidencia de los atributos de calidad en cada una de las clases, según los criterios expuestos en la Tabla 6.

**Tabla 6.** Rango de valores para medir la afectación de los atributos de calidad (TOC).

| Atributos de calidad | Clasificación | Criterio |
|---|---|---|
| Responsabilidad | Baja | Umbral<= Promedio |
| | Media | Promedio < Umbral < = 2* promedio |
| | Alta | Umbral > 2* promedio |
| Complejidad de implementación | Baja | Umbral <= Promedio |
| | Media | Promedio < Umbral < = 2* promedio |
| | Alta | Umbral > 2* promedio |
| Reutilización | Baja | Umbral > 2* promedio |
| | Media | Promedio < Umbral < = 2* promedio |
| | Alta | Umbral <= Promedio |

**Nota:** Fuente (Lorenz, y otros, 1994)

En la Figura 15 se muestra el resultado de aplicar la métrica TOC:

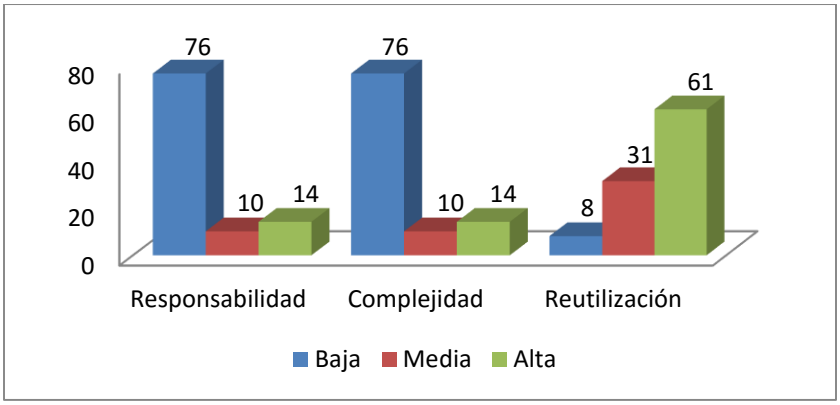



**Figura 15.** Representación en (%) de los resultados de la aplicación de la métrica TOC

**Fuente:** elaboración propia.

**Responsabilidad**: después de aplicar la métrica, se obtuvieron resultados satisfactorios que reflejan una responsabilidad baja con un valor del 76%.

**Complejidad de implementación**: luego de haber realizado la medición de la métrica, se obtuvieron resultados positivos ya que la complejidad de las clases es baja en un 76%.

**Reutilización**: se obtuvieron valores que según muestra la gráfica de la figura anterior la reutilización se comporta en un nivel alto con un 61%.

Teniendo en cuenta lo anterior se concluye que el diseño de la API propuesta en la presente investigación alcanzó resultados positivos durante la evaluación de la métrica. Se obtuvo un 76% de baja responsabilidad de las clases y complejidad de implementación respectivamente, mientras que la reutilización es alta para un 61%. Estos datos muestran que las clases tienen poca responsabilidad, lo cual amplía su reutilización y facilita la implementación.

## 3.3. Pruebas de software

Las pruebas de software comprenden el conjunto de actividades que se realizan para identificar posibles fallos de funcionamiento, configuración o usabilidad de un programa o aplicación, por medio de pruebas sobre el comportamiento del mismo.

Los niveles de pruebas de software desarrollados como parte del proceso de validación de la API propuesta en la presente investigación son: unidad y aceptación. A continuación, se describen las pruebas realizadas en cada uno de estos niveles y los resultados obtenidos en las mismas.

### 3.3.1. Pruebas internas

En esta disciplina se verifica el resultado de la implementación probando cada construcción, incluyendo tanto las construcciones internas como intermedias, así como las versiones finales a ser liberadas, desarrollando artefactos de prueba como diseños de casos de prueba (Rodríguez, 2015).

#### 3.3.1.1. Pruebas de unidad

La prueba de unidad se concentra en el esfuerzo de verificación de la unidad más pequeña del diseño: el componente o módulo de software. Tomando como guía la descripción del diseño a nivel de componente, se prueban importantes caminos de control para describir errores dentro de los límites del módulo. El alcance restringido que se ha determinado para las pruebas de unidad limita la relativa complejidad de las pruebas y los errores que éstas descubren (Pressman, 2005).

##### 3.3.1.1.1. Pruebas de caja blanca

El método de caja blanca, en ocasiones llamada pruebas de cristal, es un método que usa la estructura de control descrita como parte del diseño al nivel de componentes para derivar los casos de prueba. Al emplear los métodos de esta prueba se derivan casos de pruebas que:

1. Garantizan que todas las rutas independientes dentro del módulo se han ejercitado por lo menos una vez.
2. Ejercitan los lados verdaderos y falsos de todas las decisiones lógicas.
3. Ejecutan todos los bucles en sus límites y dentro de sus límites operacionales.
4. Ejercitan estructuras de datos internos para asegurar su validez (Pressman, 2010).

Para aplicar esta prueba se empleó la técnica de la ruta básica, utilizando como ejemplo el método iniciarAccionPanel, perteneciente a la clase *GestorMarcoTrabajo*. La selección del método se realizó teniendo en cuenta que el mismo responde a una de las principales funcionalidades de la API. La complejidad ciclomática obtenida con la aplicación de la técnica indica la cantidad de caminos independientes a recorrer para probar el código analizado, es decir, se obtiene el número de casos de pruebas a desarrollar para la

validación del método de la Figura 16. En esta se muestra la numeración de los nodos definidos en cada porción del código.

```
public void iniciarAccionPanel(String nombreFXML) throws IOException, IllegalArgumentException {
    historialPaneles.clear(); (1)
    FXMLLoader fxmlLoader = Loader.getInstancia().CargarFXML(nombreFXML);(2)
    AnchorPane root1 = (AnchorPane) fxmlLoader.load(); (3)
    if (fxmlLoader.getController().getClass().isAnnotationPresent(IniciarFlujo.class)) { (4)
        limpiarPanel(this.panelPrincipal);
        AnchorPane.setTopAnchor(root1, 0.0);
        AnchorPane.setRightAnchor(root1, 0.0);
        AnchorPane.setLeftAnchor(root1, 0.0);
        AnchorPane.setBottomAnchor(root1, 0.0);
        panelPrincipal.getChildren().add(root1);
    } else {                                                                    (5)
        throw new IllegalArgumentException("La clase : " + accionActual.
                getClass().getName() + "  debe contener la anotación iniciarFlujo"); (6)
    } (7)
}
```

**Figura 16.** Método *iniciarAccionPanel* de la clase *GestorMarcoTrabajo*

**Fuente:** elaboración propia.

A continuación, se detallan los pasos que se utilizaron al aplicar la técnica ruta básica:

1. **Confeccionar el grafo de flujo**: usando el código de la Figura 16 se realizó la representación del grafo de flujo, el cual describe un flujo de control lógico y está compuesto por los siguientes elementos:

   - ✓ **Nodos de gráfica de flujo:** son círculos que representan una o más instrucciones procedimentales.
   - ✓ **Aristas o enlaces:** son flechas que representan el flujo de control y son análogas a las flechas del diagrama de flujo.
   - ✓ **Regiones:** son las áreas delimitadas por aristas y nodos.

   En la Figura 17 se presenta el grafo de flujo obtenido:

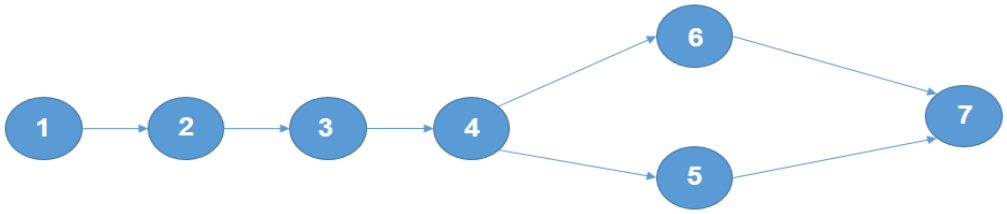



**Figura 17.** Grafo de la ruta básica del método *iniciarAccionPanel*

**Fuente:** elaboración propia.

2. **Calcular la complejidad ciclomática**: proporciona una medición cuantitativa de la complejidad lógica de un programa. El valor calculado define el número de caminos independientes del conjunto básico de un programa, y proporciona un límite superior para el número de pruebas que deben aplicarse para asegurar que todas las instrucciones se hayan ejecutado al menos una vez. La complejidad ciclomática se calcula mediante las tres formas siguientes:

   - ✓ El número de regiones del gráfico de flujo (Pressman, 2010).

     El gráfico de flujo tiene dos regiones.

   - ✓ **V (G) = E – N + 2**, donde E es el número de aristas del grafo de flujo y N es el número de nodos del mismo (Pressman, 2010).

     **V (G) = E – N + 2 = 8 aristas – 8nodos + 2 = 2**

   - ✓ **V (G) = P + 1,** donde P es el número de nodos predicados[8] contenidos en el gráfico de flujo (Pressman, 2010).

     **V (G) = 1 nodo predicado + 1 = 2**

1. **Determinar un conjunto básico de rutas linealmente independientes**: el valor de V(G) indica el número de rutas linealmente independientes de la estructura de control del programa, por lo que se definen los 2 caminos obtenidos:

**Ruta básica 1:** 1 – 2 – 3 – 4 – 5 – 7

**Ruta básica 2:** 1 – 2 – 3 – 4 – 6 – 7

2. **Obtención de casos de pruebas**: cada ruta independiente es un caso de prueba a realizar, de forma que los datos introducidos provoquen que se visiten las sentencias vinculadas a cada nodo del camino. En este caso se obtuvieron 2 caminos básicos, que dan lugar a la confección de igual número de casos de pruebas, para aplicar las pruebas a este método. A continuación, se muestran los casos de pruebas:

[8] Cada nodo que contiene una condición y está caracterizado porque dos o más aristas emergen de él.

**Tabla 7**. Caso de prueba de caja blanca para el camino básico #1

| **Caso de prueba: Camino básico #1** | |
|---|---|
| **Descripción:** este método permite iniciar una acción sin necesidad de asistente. | |
| **Entrada** | Nombre del FXML que da comienzo al asistente. |
| **Resultados esperados** | Se muestra el FXML en el espacio de la aplicación para mostrar información, con la forma de un asistente. |
| **Condiciones** | Debe existir el FXML en la ruta especificada para estos y además el controlador de dicho FXML debe tener la anotación *iniciarFlujo*. |

**Nota:** Fuente elaboración propia.

**Tabla 8**. Caso de prueba de caja blanca para el camino básico #2.

| **Caso de prueba: Camino básico #2** | |
|---|---|
| **Descripción:** este método permite iniciar una acción sin necesidad de asistente. | |
| **Entrada** | Nombre del FXML que da comienzo al asistente. |
| **Resultados esperados** | Una excepción *IllegalArgumentException*, debido a que el controlador del FXML no tenga la anotación *iniciarFlujo*. |
| **Condiciones** | Debe existir el FXML en la ruta especificada para estos y el controlador del FXML no debe tener la anotación *iniciarFlujo*. |

**Nota:** Fuente elaboración propia.

**Descripción de la ejecución de los casos de prueba**

Para ejecutar cada caso de prueba se realizaron pruebas manuales utilizando un modelo ejecutable confeccionado a través de la API. En el caso de prueba # 1 se introdujo el nombre de un FXML existente, donde su controlador contenía la anotación *iniciarFlujo,* mostrándose este FXML en el espacio de la aplicación para mostrar información, esto evidencia que el caso de prueba se ejecutó satisfactoriamente. En el caso de prueba # 2 se introdujo el nombre de un FXML donde su controlador no contenía la anotación *iniciarFlujo,* obteniéndose la excepción *IllegalArgumentException* siendo esta la respuesta esperada en este escenario.

Una vez ejecutados todos los casos de pruebas obtenidos a través de la aplicación de la técnica ruta básica, se concluye que los mismos fueron probados satisfactoriamente, demostrando que todas las rutas de este código se ejecutaron al menos una vez.

#### 3.3.1.1.2. Pruebas de caja negra

Las pruebas de caja negra, también denominadas, pruebas de comportamiento, se concentran en los requisitos funcionales del software. Es decir, permiten al ingeniero de software derivar conjuntos de condiciones de entrada que ejercitarán por completo todos los requisitos funcionales de un programa. La prueba de caja negra no es una opción frente a las técnicas de caja blanca. Es, en cambio, un enfoque complementario que tiene probabilidades de describir una clase diferente de errores de los que se descubrirán con los métodos de caja blanca (Pressman, 2005).

Las pruebas de caja negra tratan de encontrar errores en las siguientes categorías:

1. Funciones incorrectas o faltantes.
2. Errores de interfaz.
3. Errores en estructuras de datos o en acceso a bases de datos externas.
4. Errores de comportamiento o desempeño.
5. Errores de inicialización y término (Pressman, 2005).

En la validación de la solución propuesta se realizaron pruebas funcionales a la API utilizando la herramienta automatizada JUnit. Esta es un marco de trabajo, creado por Erich Gamma y Kent Beck, utilizado para automatizar las pruebas unitarias en el lenguaje Java. Las pruebas en JUnit se realizan por medio de casos de prueba escritos en clases Java.

Una de las grandes utilidades de JUnit es que los casos de prueba quedan definidos para ser ejecutados en cualquier momento, por lo cual es sencillo, luego de realizar modificaciones al programa, comprobar que los cambios no introdujeron nuevos errores (Ávila, y otros, 2014). Para la presente investigación se realizaron un total de 24 casos de pruebas a los componentes Menú, Diálogos y la clase *Loader*, teniendo en cuenta su relevancia en el diseño de la solución.

Las pruebas de caja negra realizadas a la API utilizando la herramienta automatizada mencionada anteriormente, se ejecutaron en un total de cuatro iteraciones. Las no conformidades (NC) identificadas en cada iteración fueron de tipo validación, funcionamiento e interfaz. En la Figura 18 se muestra la distribución de cada una de estas NC por iteraciones, las cuales fueron resueltas en el tiempo establecido, permitiendo obtener una API que cumple con todos los casos de prueba propuestos por el equipo de desarrollo.

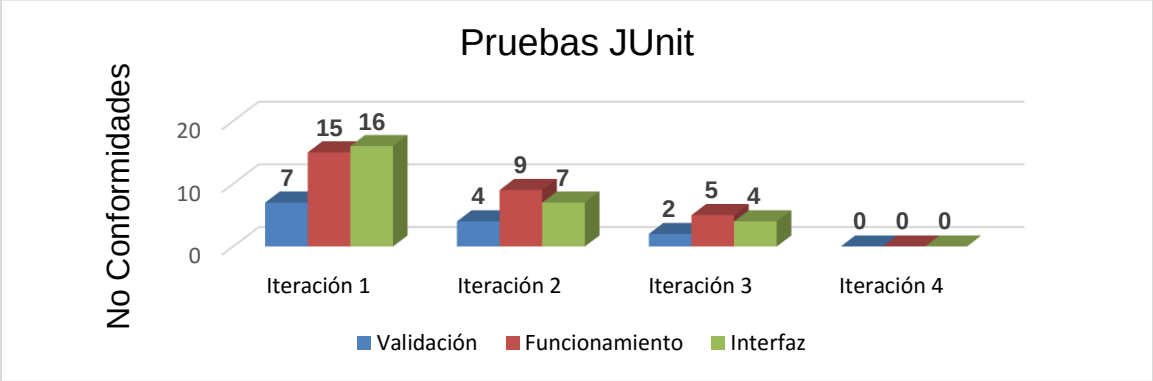



**Figura 18.** Relación de no conformidades por cada iteración

**Fuente:** elaboración propia.

En el paquete *test* del código de la API se encuentran todos los casos de prueba utilizados en la construcción de la misma. A continuación, en la Figura 19 se presenta un caso de prueba implementado para la Historia de usuario Confeccionar menú.

```java
@Test
public void showMenuLateralListaVacia() {
    AnchorPane panel_padre = new AnchorPane();
    panel_padre.getChildren().add(new Label());
    AnchorPane panel_contenido = new AnchorPane();
    Image iconBannerMenu = Mockito.mock(Image.class);
    ObservableList<Nodo> lista = FXCollections.observableArrayList();
    IMenúLateral menuLateral = GeneradorMenú.generarMenuLateral(panel_padre, 1,
     panel_contenido, iconBannerMenu, "Prueba Lista Vacia", lista);
    Assert.assertTrue(menuLateral.getListaNodos().isEmpty());
}

@Test
public void showMenuLateralListaNull() {
    AnchorPane panel_padre = new AnchorPane();
    panel_padre.getChildren().add(new Label());
    AnchorPane panel_contenido = new AnchorPane();
    Image iconBannerMenu = Mockito.mock(Image.class);
    ObservableList<Nodo> lista = null;
    IMenúLateral menuLateral = GeneradorMenú.generarMenuLateral(panel_padre, 1,
        panel_contenido, iconBannerMenu, "Prueba Lista Vacia", lista);
    Assert.assertTrue(menuLateral.getListaNodos().isEmpty());

}
@Test
public void showMenuLateralListaValida() {
    AnchorPane panel_padre = new AnchorPane();
    panel_padre.getChildren().add(new Label());
    AnchorPane panel_contenido = new AnchorPane();
    Image iconBannerMenu = Mockito.mock(Image.class);
    ObservableList<Nodo> lista = FXCollections.observableArrayList();
    lista.add(new NodoFinal("Empresas Cubanas", new EventoClick() {
        @Override
        public void onClick() {
            System.out.println("Empresa");
        }
    }));

    lista.add(new NodoFinal("Importadores y exportadores", new EventoClick() {
        @Override
        public void onClick() {
            System.out.println("Importadores");
        }
    }));
    ObservableList list = FXCollections.observableArrayList();
    ObservableList list2 = FXCollections.observableArrayList();
    ObservableList list3 = FXCollections.observableArrayList();
    lista.add(new NodoContenido("Prueba", list));
    list.add(new NodoContenido("Prueba 1", list2));
    list2.add(new NodoContenido("Prueba 2", list3));
    list2.add(new NodoFinal("Prueba Final", new EventoClick() {
        @Override
        public void onClick() {
            System.out.println("Final");
        }
    }));
    IMenúLateral menuLateral = GeneradorMenú.generarMenuLateral(panel_padre, 1,
     panel_contenido, iconBannerMenu, "Prueba Lista Vacia", lista);
    Assert.assertTrue(lista.get(0) == menuLateral.getListaNodos().get(0));

}
```

**Figura 19.** Caso de Prueba para la HU_Confeccionar menú

**Fuente:** elaboración propia.

A continuación, en la Figura 20 se muestra el resultado de la ejecución satisfactoria de todos los casos de prueba utilizados en el desarrollo de la API.

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running apiTest.dialogosTest.FactoriaOpcionDialogoTest
Tests run: 16, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 3.597 sec - in apiTest.dialogosTest.FactoriaOpcionDialogoTest
Running apiTest.LoaderTest
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.002 sec - in apiTest.LoaderTest
Running menuTest.MenuTest
Tests run: 6, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 1.483 sec - in menuTest.MenuTest

Results :

Tests run: 24, Failures: 0, Errors: 0, Skipped: 0
```

**Figura 20.** Resultados de JUnit

**Fuente:** elaboración propia.

Estas pruebas fueron supervisadas por los especialistas del grupo de calidad de CEGEL, generándose así el Acta de Liberación Interna de Productos de Software (ver Anexo 6).

### 3.3.2. Pruebas de aceptación

Es la prueba final antes del despliegue del sistema. Su objetivo es verificar que el software está listo y que puede ser usado por usuarios finales para ejecutar aquellas funciones y tareas para las cuales el software fue construido (Rodríguez, 2015).

Para la aplicación de estas pruebas, se confeccionó un caso de prueba de aceptación por cada HU y un caso de estudio con el propósito que el cliente valide si la API se encuentra lista en relación a las necesidades del mismo. A continuación, se muestra el caso de prueba de aceptación de la HU Iniciar aplicación cliente, teniendo en cuenta que esta HU responde a una de las principales funcionalidades de la API.

**Tabla 9.** Caso de prueba de Aceptación de la HU “Iniciar aplicación cliente”

| **Caso de prueba de aceptación** | |
|---|---|
| **Código de caso de prueba:** 01 | **Nombre historia de usuario:** Iniciar aplicación cliente |

<table>
<tr><td colspan="3">Nombre de la persona que realiza el caso de prueba: Rafael Mayor Alberto</td></tr>
<tr><td colspan="3">Descripción de la prueba: revisar a través del caso de estudio el correcto funcionamiento del RF Iniciar aplicación cliente.</td></tr>
<tr><td colspan="3">Condiciones de ejecución: se debe especificar la dirección donde se encuentran ubicados los FXML y se procede a iniciar la aplicación cliente introduciendo los parámetros correspondientes.</td></tr>
<tr><td colspan="2">Entrada/Pasos de ejecución</td><td rowspan="2">Resultados esperados:</td></tr>
<tr><td>Acción:</td><td>Entrada:</td></tr>
<tr><td>Se decide realizar una aplicación cliente utilizando la API.</td><td>Dirección de los FXML y parámetros de iniciación.</td><td>La aplicación debe iniciar correctamente.</td></tr>
<tr><td colspan="3">Evaluación de prueba: Satisfactoria</td></tr>
</table>

**Nota:** Fuente elaboración propia.

Se realizó un encuentro con los ingenieros Reinier Silverio Figueroa y Juan David Gómez Amador, profesionales a cargo de la construcción y mantenimiento del marco de trabajo XEGFORT, con el objetivo de revisar la API, considerando los casos de pruebas definidos. Obteniendo resultados satisfactorios, avalados por los profesionales antes mencionados. Al finalizar el encuentro se generó el Acta de Aceptación del Producto (ver Anexo 7 ).

## 3.4. Validación de los resultados de la investigación

Teniendo en cuenta que en la investigación realizada se define como idea a defender que: "Con el desarrollo de una API se reduce el tiempo en el desarrollo de aplicaciones clientes de escritorio con XEGFORT agilizando el proceso de implementación".

Para analizar la relación causa efecto entre la variable independiente "el desarrollo de un API" y la variable dependiente "reducir el tiempo en el desarrollo de aplicaciones clientes de escritorio con XEGFORT agilizando el proceso de implementación", el autor de la presente investigación, define un conjunto de criterios de medida que permiten validar cómo a través de la API obtenida se logra la relación entre ambas variables. La obtención de estos criterios se realiza a partir de las principales deficiencias identificadas en la situación problemática que dan lugar al desarrollo de la API propuesta.

**Criterios de medida definidos:**

- ✓ Cantidad de interfaces para implementar una personalización: este criterio influye en el tiempo empleado a la hora de personalizar un componente.
- ✓ Facilidad en el manejo de eventos: contribuye a la agilidad en la utilización de eventos en ambas API.
- ✓ Facilidad en el uso de los componentes de la interfaz gráfica: este criterio mide la facilidad al usar cada funcionalidad de los componentes de interfaz gráfica.
- ✓ Facilidad de personalización o modernización visual de componentes existentes: mide cuanto tiempo se debe invertir para personalizar un componente de la interfaz gráfica, existente en ambas API.

A continuación, en la Tabla 10, se evalúan cada uno de los criterios de medida definidos. La evaluación se realiza a través de un antes (API Acciones) y un después (API propuesta en la presente investigación), con el propósito de verificar cómo mediante la solución propuesta se reduce el tiempo en el desarrollo de aplicaciones clientes de escritorio con XEGFORT agilizando el proceso de implementación. Los datos tenidos en cuenta en la comparación fueron tomados a partir del desarrollo de un caso de estudio con el uso de ambas API por parte de los dos principales arquitectos del marco de trabajo XEGFORT.

**Tabla 10.** Validación de las variables de investigación

| Criterios de medida | Antes | Después |
|---|---|---|
| Cantidad de interfaces para implementar una personalización. | Para personalizar un componente es necesario utilizar múltiples interfaces, teniendo en cuenta que esta se basa en el comportamiento de SWING. Ejemplo: para personalizar el *JComboBox* con el objetivo que acepte objetos como parámetro, se hace necesario implementar un grupo de cinco clases, empleándose en este proceso un tiempo promedio 1 hora. | Para personalizar un componente basta con extender la clase del componente que se desea personalizar. Ejemplo: para personalizar un *ComboBox* con el mismo objetivo planteado en la anterior proposición, solo se debe extender una clase, empleándose en este proceso un tiempo promedio de 20 minutos. |
| Facilidad en el manejo de eventos. | Los eventos están orientados solamente a los componentes de la interfaz gráfica y a sus modelos | Los eventos son extendidos a nivel de propiedades, es decir, cualquier objeto que cuente con una propiedad |

| | | |
|---|---|---|
| | correspondientes. Esto implica tener que configurar de forma manual la actualización de la vista, proceso que tiene una duración promedio de 4 minutos por componentes. | puede registrar un evento. Esto implica que las actualizaciones de la vista se realicen declarando su vínculo una sola vez. Reduciéndose este proceso a un tiempo de 3 minuto por componente. |
| Facilidad en el uso de los componentes de la interfaz gráfica. | Para usar un componente de la interfaz gráfica se genera mucho código *boilerplate*[9]. Ejemplo: para cargar los datos en un *JTable* es necesario crear un modelo o iterar sobre una lista de objetos extrayendo los atributos que se van a mostrar en la misma. Proceso que tarda 10 minutos aproximadamente. | Para realizar la misma acción con el componente *Table* de JavaFX, se configura una sola vez, lo que se va a mostrar en cada celda, y se provee la lista de objetos. Este proceso tarda 5 minutos aproximadamente. |

[9] Código repetitivo pero necesario para el correcto funcionamiento de la aplicación.

| | | |
|---|---|---|
| Facilidad de personalización o modernización visual de componentes existentes. | Visualmente en la API acciones para personalizar o modernizar hay que implementar el *Look and Feel* siendo este un trabajo difícil y extenso. Proceso que demora de 8 a 12 horas de implementación dependiendo del conocimiento que presente el desarrollador. | Teniendo en cuenta el mismo propósito anterior con el uso de JavaFX y la API propuesta, se puede configurar utilizando directamente la tecnología CSS, siendo esta una tecnología más conocida que la anterior lo que implica invertir un tiempo de 1 a 2 horas en este proceso. |

**Nota:** Fuente elaboración propia.

La comparación realizada en la tabla anterior, a través de los criterios de medida antes definidos, demuestra cómo utilizando la API obtenida en la presente investigación, se reduce el tiempo en el desarrollo de aplicaciones cliente con XEGFORT agilizando el proceso de implementación.

## 3.5. Conclusiones parciales

La aplicación de métricas de validación del diseño, pruebas de unidad y de aceptación a la solución propuesta, certifica la obtención de una API funcional, avalada en cada caso por las actas de liberación y aceptación emitidas. Por otra parte, la validación del resultado de la investigación, demuestra el cumplimiento de la relación causa efecto de la variable independiente “el desarrollo de una API” sobre la variable dependiente “reducir el tiempo en el desarrollo de aplicaciones clientes de escritorio con XEGFORT agilizando el proceso de implementación”.

# CONCLUSIONES GENERALES

El estudio de las bibliotecas de componentes existentes para el diseño de interfaz en aplicaciones desarrolladas en Java, demuestra que Swing presenta un conjunto de limitantes con respecto a JavaFX, en relación a la complejidad en el diseño de determinados componentes de interfaz gráfica, lo cual fundamenta la necesidad de implementar la API propuesta.

El empleo de patrones de diseño, el uso del patrón arquitectónico Modelo-Vista-Presentador, los estándares de codificación utilizados y la implementación de los requisitos identificados permiten obtener una solución informática que disminuye el tiempo de desarrollo de aplicaciones clientes de escritorio con XEGFORT.

El desarrollo de pruebas de unidad, aceptación y el resultado arrojado por las métricas de validación de diseño de software, corroboran la validez y utilidad de la API propuesta en la presente investigación.

Con la validación de las variables de la investigación, se demuestra que con el desarrollo de la API propuesta se reduce el tiempo en el desarrollo de aplicaciones clientes de escritorio con XEGFORT, lo que implica rapidez en el proceso de implementación.

## RECOMENDACIONES

- ✓ Incorporar a la API obtenida en la presente investigación un mecanismo para modelar el flujo independiente a las acciones.
- ✓ Incorporar todos los componentes de versiones anteriores del marco de trabajo XEGFORT a la API, utilizando la biblioteca de componentes JavaFX.

## BIBLIOGRAFÍA REFERENCIADA

Alejandra. 2017. elwebmaster. *elwebmaster.* [En línea] 30 de enero de 2017. http://www.elwebmaster.com/referencia/api-interface-de-programacion-de-aplicaciones.

Ambler, Scott W. 2017. Agile Modeling. *Agile Modeling.* [En línea] 4 de enero de 2017. http://www.agilemodeling.com/essays/simpleTools.htm#SelectingCASE.

Apache. 2015. Apache Maven Project. *Apache Maven Project.* [En línea] 3 de 5 de 2015. [Citado el: 16 de 6 de 2017.] http://maven.apache.org/.

Ávila, Adriana, y otros. 2014. *Pruebas Unitarias en Java JUnit y TestNG.* s.l. : UdelaR, 2014.

Beck, Kent . 2002. *Extreme programming explained: embrace change.* s.l. : Addison-wesley professional, 2002.

Borja , Federico, y otros. 2017. Breve historia de las GUI. *Breve historia de las GUI.* [En línea] 28 de enero de 2017. http://sabia.tic.udc.es/gc/Contenidos%20adicionales/trabajos/Interfaces/enlightment/guis_1.html.

Burback, Ron. 1998. *Software Engineering Methodology.* 1998.

Camacho, Erick . 2013. Tutorial Introducción a Maven 3. [En línea] 6 de 12 de 2013. [Citado el: 16 de 06 de 2017.] http://www.javahispano.org.

Carrillo, Michel Gallego y Montalvo, Soto. 2005. *Interfaces Gráficas en Java.* s.l. : Editorial Universitaria Ramón Areces, 2005.

Expósito, C. M. 2006. *Interfaz Gráfica de Usuario. Aproximación semiótica y cognitiva.* 2006.

Fernández, David R. 2000. *Arquitectura de Software.* s.l. : Universidad Tecmilenio, 2000.

Gómez, O. T., López, P. P. R., & Bacalla, J. S. 2014. *Criterios de selección de metodologías de desarrollo de software. .* s.l. : Industrial Data, 2014.

Gramage, María Carmen Penades. 2016. *Diagrama de clases.* 2016.

Guerrero, C.A, y otros. 2014. *Estudio comparativo de Marcos de Trabajo para el Desarrollo Software Orientado a Aspectos.* s.l. : Información tecnológica, 2014. Vol. 25.

Joyanes, L. 2013. *Fundamentos de programación. Algoritmos, estructuras de datos y objetos.* s.l. : McGrawHill, 2013.

Larman, Craig. 1999. *UML y Patrones. Pearson.* 1999.

Lorenz, Mark y Kidd, Jeff. 1994. *Object-oriented software metrics: a practical guide.* Nueva Jersey : Prentice-Hall, 1994.

Martínez, L. H. 2008. *Intérprete y Entorno de Desarrollo para el Aprendizaje de Lenguajes de Programación Estructurada.* 2008.

Mora, S. L. 2002. *Programación de aplicaciones web: historia, principios básicos y clientes web.* s.l. : Club Universitario, 2002.

Netbeans.org. 2016. Bienvenido a NetBeans y www.netbeans.org. *NetBeans.* [En línea] 2016. http://netbeans.org/index_es.html.

Northover, Stave y Wilson, MIke. 2015. SWT: The Standard Widget Toolkit, Volume 1. s.l. : Addison-Wesley, 2015, pág. 592.

*Patrones de diseño.* Johansen, Ernst. 2006. 2, s.l. : Revista ABB, 2006, págs. 62-65.

Pressman, Roger S. 2010. *Ingeniería de Software.Un enfoque práctico.Séptima Edición.* s.l. : MC Graw Hill, 2010.

Pressman, Roger. 2005. *Ingeniería de Software.Un enfoque práctico.Sexta edición.* s.l. : MC Graw Hill, 2005.

Pressman, Roger S. 2002. *Ingeniería del Software. Un enfoque práctico. Quinta Edición.* s.l. : Mc Graw Hill, 2002.

Rodríguez, Tamara. 2015. *Metodología de desarrollo para la Actividad productiva de la UCI.* La Habana : s.n., 2015.

Rojas, MC Juan Carlos Olivares. 2014. *Patrones de Diseño.* 2014.

Sommerville, Ian. 2005. *Ingeniería de Software Séptima edición.* s.l. : Addison Wesley, 2005.

Soulary, Beyris y Liliam, Celia. 2010. *Proceso de medición y análisis para el polo de hardware y automática.* 2010. Vol. 3.

Visconti, Marcello y Astudillo, Hernán. 2013. *Fundamentos de Ingeniería de Software.* s.l. : Universidad Técnica Federico Santa María, 2013.

Visual Paradigm. 2017. Visual Paradigm. *Visual Paradigm.* [En línea] 23 de enero de 2017. [Citado el: 25 de octubre de 2016.] https://www.visual-paradigm.com/.

Weaver, Jame. 2007. JavaFX Script: Dynamic Java Scripting for Rich Internet/Client-side Applications. *JavaFX Script: Dynamic Java Scripting for Rich Internet/Client-side Applications.* s.l. : Apress, 2007.

## ANEXOS

**Anexo 1** Guía de preguntas utilizadas en el desarrollo de la entrevista con los profesionales a cargo del marco de trabajo XEGFORT.

1. ¿Ustedes conocen de soluciones informáticas que cumplan las características deseadas para la nueva API?
2. Teniendo en cuenta que se desea realizar una API que permita iniciar una aplicación cliente de forma sencilla. ¿Qué elementos consideran ustedes necesarios que tienen que cumplir estas funcionalidades en la nueva API?
3. ¿El menú que presente la API debe ser de forma estática o debe desplazarse para ocupar menos espacio en pantalla?
4. ¿Cuántos tipos de menú desean que presente la nueva API?
5. ¿Qué otros componentes debe proveer la nueva API?
6. Para que sistemas operativos debe ser compatible las aplicaciones construidas utilizando la API.
7. Mencione las especificaciones de hardware mínimas sobre las que la API deba funcionar.
8. ¿Debe ser fácil configurar el aspecto y estilo de la aplicación que utilice la API?
9. ¿Es recomendable que la nueva API incluya los *javadoc* en cada funcionalidad?

**Anexo 2** Acuerdos tomados en la Tormenta de ideas.

| No | Acuerdo | Responsable | Fecha |
|---|---|---|---|
| 1 | La API debe tener funcionalidades que permitan iniciar una aplicación cliente de forma sencilla | Rafael Mayor Alberto | 26-10 |
| 2 | Debe ser fácil de aprender a utilizar la nueva API | Rafael Mayor Alberto | 26-10 |

**Anexo 3** Acuerdos tomados en las revisiones formales de los requisitos.

| No | Acuerdo | Responsable | Fecha |
|---|---|---|---|
| 1 | Mejorar la descripción del requisito funcional Visualizar un FXML en el marco de la aplicación. | Rafael Mayor Alberto | 28-10 |
| 2 | Mejorar la descripción del requisito funcional Navegar dentro del asistente. | Rafael Mayor Alberto | 28-10 |

| | | | |
|---|---|---|---|
| 3 | Falta requisito no funcional relacionado con tener instalada en la computadora la Máquina Virtual de Java en su versión 8 o superior para el funcionamiento de una aplicación escrita en Java. | Rafael Mayor Alberto | 28-10 |
| 4 | Falta requisito no funcional relacionado a los tiempos que debe demorar la API en realizar sus funcionalidades | Rafael Mayor Alberto | 28-10 |

**Anexo 4** Historias de usuario

**Tabla 11.** Historia de usuario Visualizar un FXML en el marco de la aplicación.

| **Historia de Usuario** | |
|---|---|
| **Número:** 2 | **Nombre de la Historia de Usuario:** Visualizar un FXML en el marco de la aplicación. |
| **Programador:** Rafael Mayor Alberto | **Iteración asignada:** 2 |
| **Prioridad:** media | **Tiempo estimado:** 5 días |
| **Riesgo en desarrollo:** bajo | **Tiempo real:** 5 días |
| **Descripción:** se debe permitir al desarrollador iniciar una acción dentro del marco de la aplicación sin necesidad de iniciar un asistente. | |
| **Observaciones:** el desarrollador debe haber realizado el FXML correspondiente. | |
| **Prototipo de interfaz:** no aplica | |

**Nota:** Fuente elaboración propia.

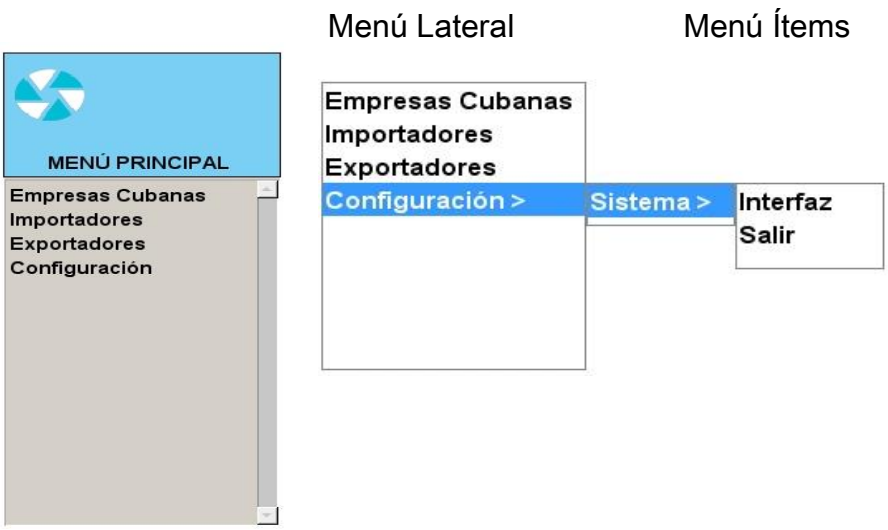
**Tabla 12.** Historia de usuario Navegar dentro del asistente.

| **Historia de Usuario** | |
|---|---|
| **Número:** 4 | **Nombre de la Historia de Usuario:** Navegar dentro del asistente. |
| **Programador:** Rafael Mayor Alberto | **Iteración asignada:** 4 |
| **Prioridad:** alta | **Tiempo estimado:** 10 días |
| **Riesgo en desarrollo:** alto | **Tiempo real:** 10 días |
| **Descripción:** se debe permitir al desarrollador navegar dentro del asistente, es decir poder interactuar con los botones que presente y asignar una funcionalidad a cada uno de estos. | |
| **Observaciones:** debe tener por defecto los botones: anterior, terminar, cancelar, terminar y siguiente. Además debe permitir agregar otros botones. | |
| **Prototipo de interfaz:** no aplica | |

**Nota:** Fuente elaboración propia.

**Tabla 13.** Historia de Usuario Confeccionar menú.

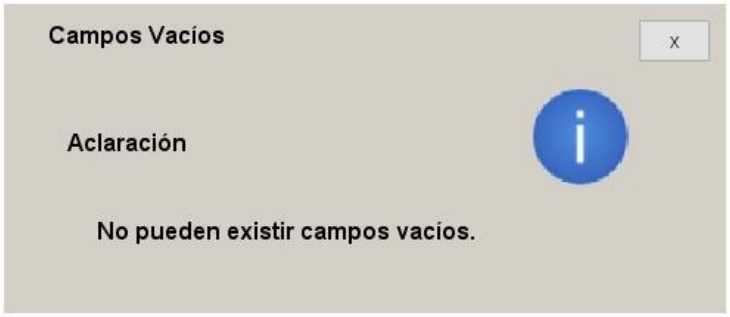
| **Historia de Usuario** | |
|---|---|
| **Número:** 5 | **Nombre de la Historia de Usuario:** Confeccionar menú. |
| **Programador:** Rafael Mayor Alberto | **Iteración asignada:** 5 |
| **Prioridad:** alta | **Tiempo estimado:** 10 días |

| | |
|---|---|
| **Riesgo en desarrollo:** alto | **Tiempo real:** 10 días |
| **Descripción:** se debe proveer de un menú que se adapte a las necesidades de la aplicación y además que permita utilizar mejor el espacio de la pantalla asignado a mostrar la información. | |
| **Observaciones:** debe realizar dos menús uno superior y uno lateral. | |
| **Prototipo de interfaz:**   | |

**Nota:** Fuente elaboración propia.

**Tabla 14.** Historia de usuario Mostrar mensajes

| **Historia de Usuario** | |
|---|---|
| **Número:** 6 | **Nombre de la Historia de Usuario:** Mostrar mensajes. |
| **Programador:** Rafael Mayor Alberto | **Iteración asignada:** 6 |
| **Prioridad:** alta | **Tiempo estimado:** 5 días |
| **Riesgo en desarrollo:** bajo | **Tiempo real:** 5 días |

| |
|---|
| **Descripción:** la API debe proveer de funcionalidades para mostrar mensajes que permitan interactuar de fácil modo con el usuario. |
| **Observaciones:** debe contener mensajes de error, confirmación, selección, entre otros. |
| **Prototipo de interfaz:**   |

**Nota:** Fuente elaboración propia.

**Anexo 5** Diagrama de clases de cada paquete

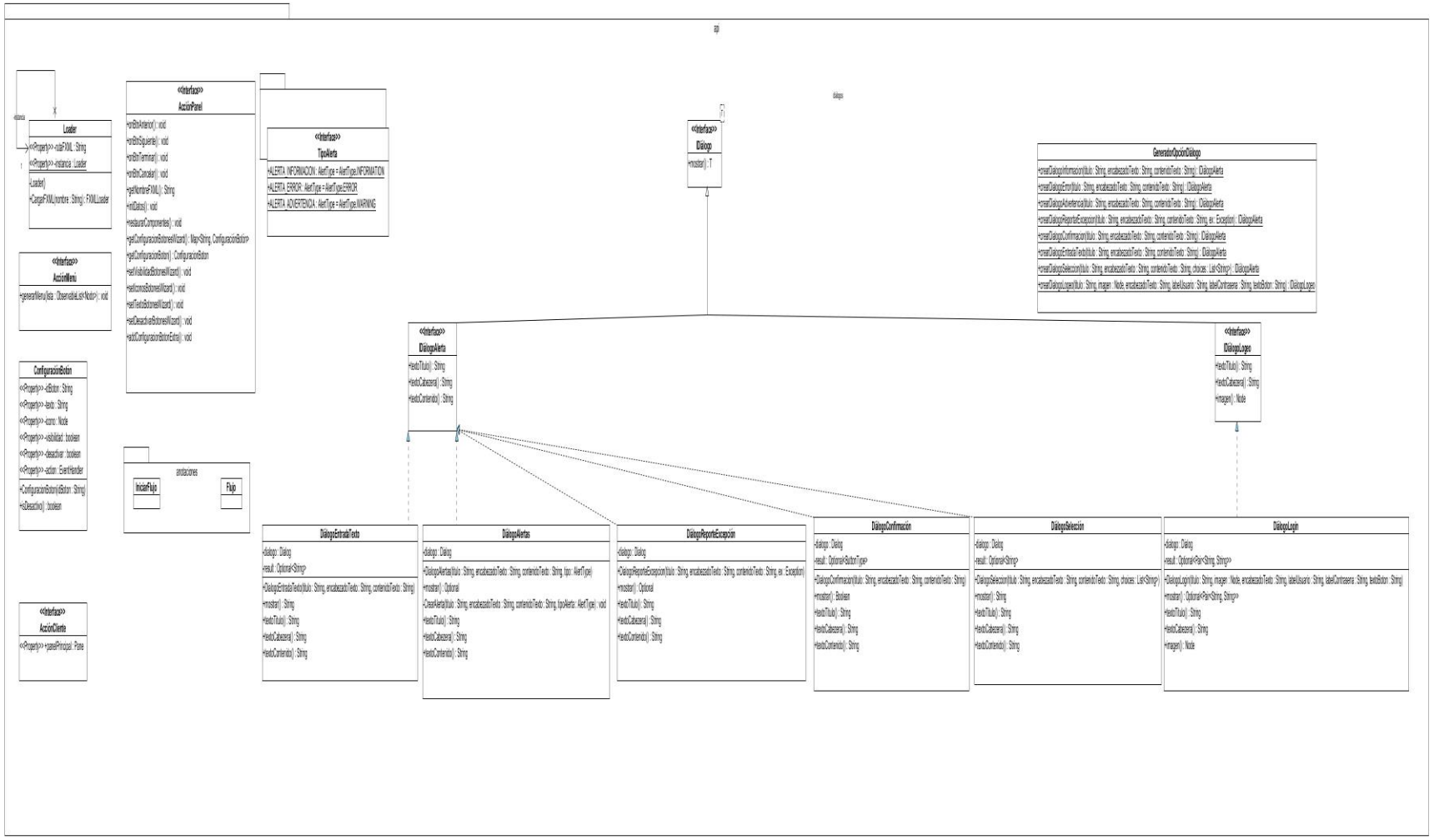



**Figura 21.** Diagrama de clases paquete api

Fuente: elaboración propia.

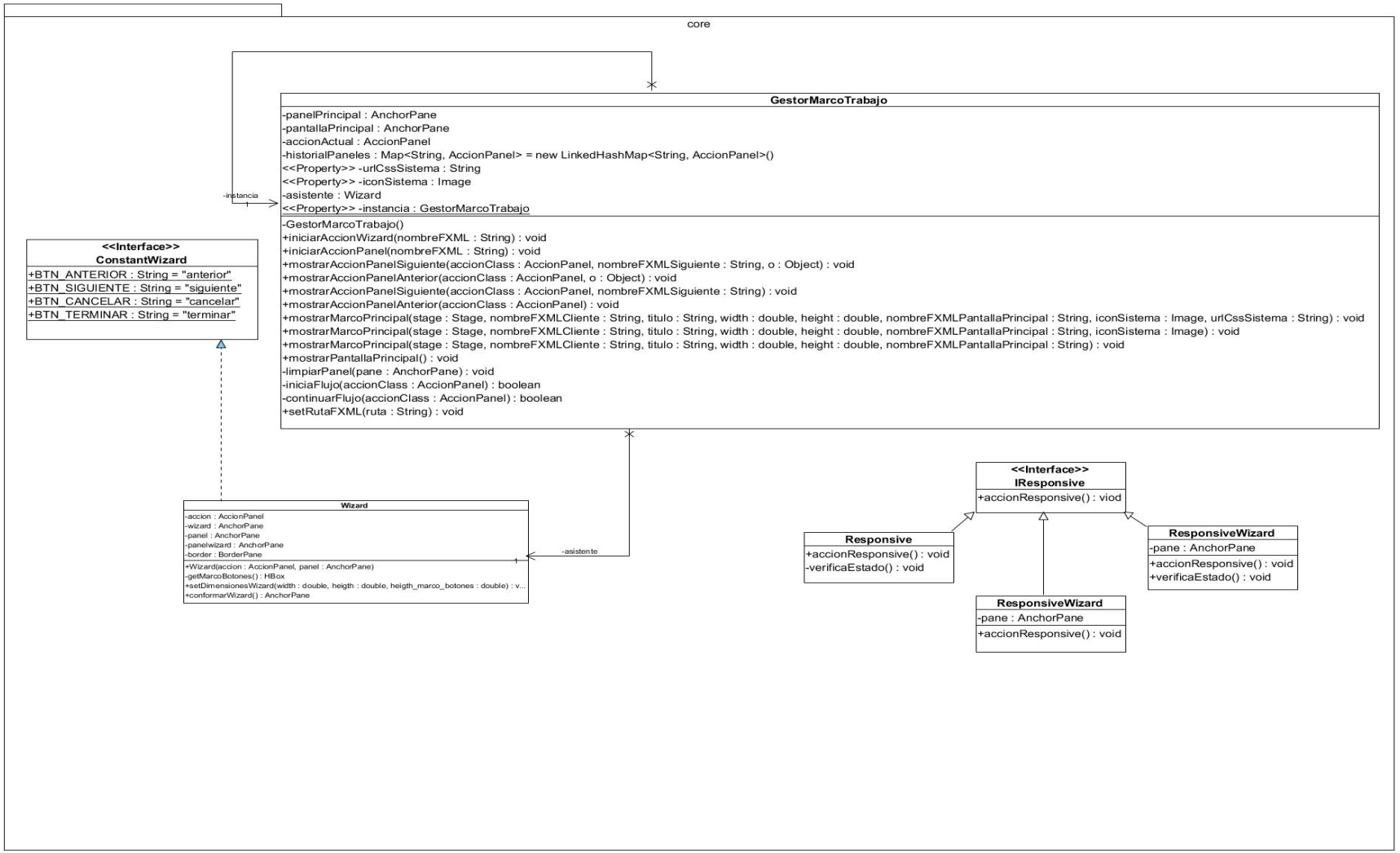



**Figura 22.** Diagrama de clases paquete core

Fuente: elaboración propia.

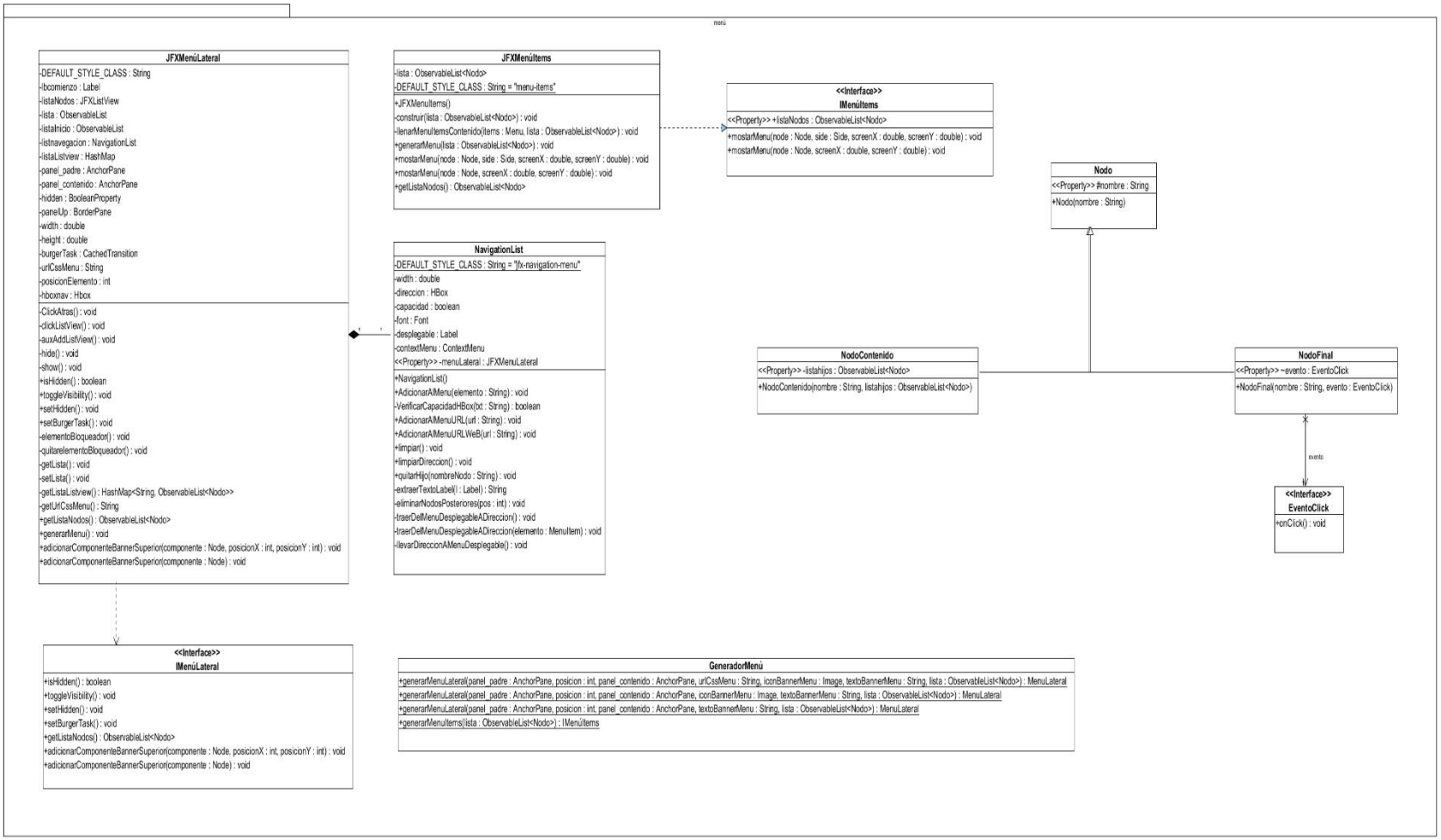



**Figura 23.** Diagrama de clases paquete menú

Fuente: elaboración propia.

**Anexo 6** Acta de Liberación Interna de Productos de Software.

FACULTAD # 3
CENTRO DE GOBIERNO ELECTRÓNICO

## Acta de Liberación Interna de Productos Software

**Fecha de emisión del acta:** *11*/05/2017

**Emitida a favor de:** Tesis *"API para el desarrollo de aplicaciones clientes en el marco de trabajo XEGFORT".*

***Datos del producto***

| Artefacto | Versión | Estado final | Cantidad Iteraciones | Tipos de pruebas realizadas | Fecha de liberación |
|---|---|---|---|---|---|
| *App: API para el desarrollo de aplicaciones clientes en el marco de trabajo XEGFORT* | *1.0* | *0* | *3* | *Evaluación dinámica Pruebas de Funcionalidad* | *11/05/2017* |

MSc. Yordanis García Leiva
**Asesor de Calidad CEGEL**

Rafael Mayor Alberto
**Autor**

1

**Figura 24.** Acta de Liberación Interna de Productos de Software

**Anexo 7** Acta de Aceptación del Producto.

UCI Universidad de las Ciencias Informáticas

## Acta de aceptación

En cumplimiento del desarrollo de la Tesis: **API para el desarrollo de aplicaciones clientes en el marco de trabajo XEGFORT.** Se hace entrega del producto:

- API para el desarrollo de aplicaciones clientes en el marco de trabajo XEGFORT.

La Parte Cliente, luego de haber revisado el software, considera que la solución cumple con cada uno de los requisitos que originaron su desarrollo. Conforme a lo anterior se expresa la aceptación de la misma.

| **Entrega** | **Recibe** |
|---|---|
| **Nombre y apellidos:** Rafael Mayor Alberto | **Nombre y apellidos:** Ing. Reinier Silverio Figueroa |
| **Cargo:** Estudiante | **Cargo:** Arquitecto principal de XEGFORT |

**Fecha:** 10/05/2017

**Figura 25.** Acta de Aceptación del Producto