UNIVERSIDAD DE LAS CIENCIAS INFORMÁTICAS

Facultad 5

Centro de Entornos Interactivos 3D

Propuesta de arquitectura extensible para la creación de una aplicación orientada a la resolución de problemas de regresión no lineal

Trabajo de Diploma para optar por el título de
Ingeniero en Informática

Autor: Francisco Javier Amador Sala
Tutor: Ing. Yasmany Alfonso Monteagudo

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

A mi madre, por ser mi fuerza, mi inspiración, mi vida

AGRADECIMIENTOS

A mi padre, quien me apoyó en todo momento

A mi abuela, por su preocupación y amor constantes

A mis amigos, mi segunda familia, mi apoyo, mi mayor riqueza

A Máximo, Luis Enrique, Richel, Nayibi y Yasmany

A Cesar

DECLARACIÓN JURADA DE AUTORÍA Y AGRADECIMIENTOS

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

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

Francisco Javier Amador Sala	Yasmany Alfonso Monteagudo
Firma del Autor	Firma del Tutor

RESUMEN

En el artículo “Diseño de un marco de trabajo extensible para resolución de problemas de regresión no lineal”[1], se propone una arquitectura basada en *plugins* que logra expresar dicho proceso de resolución en términos de componentes simples. Sin embargo, hallar la solución de estos problemas no es suficiente, es necesario además permitir a los investigadores la realización de otras tareas que van más allá de los procedimientos de la regresión. Con el objetivo de extender la arquitectura de software existente para lograr una aplicación orientada a la resolución de problemas de regresión no lineal, y que permita resolver problemas de flexibilidad, extensibilidad y facilidad de mantenimiento de la solución actual, se utilizaron como métodos científicos de investigación los métodos teóricos: “Análisis Histórico-Lógico” y “Analítico-Sintético”. Además, se aplicó la metodología de desarrollo ágil AUP que abarca todo el proceso de desarrollo del software con iteraciones pequeñas. La solución ha sido diseñada e implementada usando herramientas como: *Visual Paradigm*, *Qt Framework*, *Qt Creator*, Armadillo y *QCustomPlot*. Con la solución propuesta, los investigadores podrán implementar herramientas que le permitan conocer y medir la calidad del ajuste, generar datos sintéticos, incorporar gráficos y realizar análisis estadísticos a los datos de entrada, entre otras funcionalidades como comparar la calidad de las soluciones obtenidas empleando diferentes modelos y algoritmos, que podrán ser agregadas a la aplicación en tiempo de ejecución.

Palabras claves: regresión, arquitectura, extensible

[1] Autor: Ing. Yasmany Alfonzo Monteagudo

ABSTRACT

In the article "Design of an extensible framework for non-linear regression problems solving", is proposed a plugin based architecture which express this resolution process in terms of simple components. However, solve the non-linear regression problems isn't enough, is also need it to allow to the researchers the realization of others task which are out of the regression procedures. Seeking the goal of extend the existing software architecture for getting an application oriented to the resolution of non-linear regression problems and that allows reduce flexibilities, extensibilities and handling issues of the actual solution, were used as research scientific methods the theory methods: "Historical-Logical Analysis" and "Analytic-Synthetic". Also, was applied the development methodology AUP which covers all the development process with smalls iterations. The solution has been designed and developed using tools as: Visual Paradigm, Qt Framework, Qt- Creator, Armadillo and QCustomPlot. With the proposed solution, the researches will be able to develop tools which will allow them to know and measure the quality of the fittings, generate synthetic data, incorporate graphs or do statistical analysis of the input data, among other functionalities like to compare the quality of the different solutions obtained using diverse models and algorithms, which may be added to the application in runtime.

Keywords: *regression, architecture, extensible*

CONTENIDO

INTRODUCCIÓN

La estimación de parámetros a partir de datos conocidos, constituye uno de los retos en casi todas las ramas de la ingeniería. Por ejemplo, en la Meteorología se necesitan predecir parámetros atmosféricos a partir de un conjunto de observaciones obtenidas de instrumentos de medición. Asimismo, en la Industria del Petróleo se requiere estimar magnitudes de un depósito tales como la temperatura y la porosidad de las rocas a partir lecturas de presión obtenidas durante un período de tiempo.

La estimación de parámetros se realiza mediante técnicas de regresión lineal o no lineal, donde utilizando un modelo se construye una función objetivo que se minimiza para encontrar los parámetros de mejor ajuste. Las funciones de regresión no lineal típicamente son funciones multimodales[2]; conllevando a que en muchas ocasiones sea necesario realizar diferentes combinaciones de un algoritmo global y uno local, para encontrar un óptimo global o una aproximación del mismo (1).

Precisamente en el artículo "*Diseño de un marco de trabajo extensible para resolución de problemas de regresión no lineal*" (2) del autor Ing. Yasmany Alfonso Monteagudo se propone una arquitectura basada en *plugins* que logra expresar dicho proceso de resolución en términos de "*una interacción entre componentes simples, donde cada componente tiene una responsabilidad bien definida dentro del mismo"*(2). Esta arquitectura permite que los investigadores puedan hallar la solución de un problema de regresión no lineal seleccionando un algoritmo de optimización global, uno local o una combinación de ambos. Sin embargo, si se desea resolver el mismo problema empleando algoritmos diferentes, el investigador se ve obligado a realizar todo el proceso varias veces lo que causa notables atrasos en el tiempo planificado. Además, hallar la solución de un problema no es suficiente. En la mayoría de los casos los investigadores necesitan conocer o medir la calidad del ajuste, generar datos sintéticos, incorporar gráficos o realizar análisis estadísticos a los datos de entrada, así como comparar la calidad de las soluciones que se obtienen empleando diferentes modelos y algoritmos. Con la solución existente sólo se pueden hacer estas tareas manualmente, a vista, o modificando totalmente el código fuente cada vez que se desee agregar una nueva funcionalidad.

Lo expresado anteriormente conlleva a plantearse la siguiente interrogante ¿Cómo resolver problemas de flexibilidad, extensibilidad y facilidad de mantenimiento en la creación de una solución orientada a la resolución de problemas de regresión no lineal? La cual constituye el **problema a resolver**.

[2] Funciones que tienen varios mínimos.

De aquí que el **objeto de estudio** sea la Arquitectura de software y el **campo de acción** la Arquitectura basada en *plugins*. Para dar solución al problema planteado se define como **objetivo general** extender la arquitectura existente para lograr una solución orientada a la resolución de problemas de regresión no lineal que permita resolver problemas de flexibilidad, extensibilidad y facilidad de mantenimiento, desglosándose en los siguientes **objetivos específicos**:

1. Elaborar el marco teórico de la investigación a partir del estudio del estado del arte de otras herramientas de regresión no lineal.
2. Realizar el análisis y diseño de la solución.
3. Implementar y validar la solución propuesta.

Para lograr el cumplimiento de los objetivos, se proponen las siguientes **tareas de investigación**:

1. Elaboración del marco conceptual para precisar las definiciones fundamentales de la investigación.
2. Construcción del estado del arte referente al tema.
3. Investigación, análisis y selección de las herramientas, lenguaje y tecnologías a emplear para desarrollar la propuesta de solución.
4. Definición de los requisitos funcionales y no funcionales.
5. Descripción de la arquitectura.
6. Elaboración del análisis y el diseño de la solución.
7. Implementación de la solución propuesta.
8. Validación de la solución propuesta.

Para realizar la investigación se hará uso de los **métodos científicos,** entre ellos los **métodos teóricos**:

1. **Análisis Histórico-Lógico** que posibilitará iniciar la investigación teniendo en cuenta las descripciones históricas de herramientas de regresión, además de poner en función los resultados del estudio para obtener una base que ayude a dar solución al problema planteado.
2. **Analítico-Sintético** que hará posible profundizar y desglosar toda la información encontrada sobre herramientas de regresión, y así determinar la solución más apropiada en dependencia del problema planteado.

Teniendo en cuenta el objetivo general se plantea como **idea a defender**: La implementación de una arquitectura que permita adaptar la aplicación actual a requisitos que deban ser implementados en el futuro mediante el uso de *plugins*, permitirá resolver problemas de flexibilidad, extensibilidad y facilidad de mantenimiento.

El presente documento consta de tres capítulos que a su vez se dividen en epígrafes y estos en sub-epígrafes de acuerdo al nivel de detalle que requiere el contenido abordado en cada uno de ellos. A continuación, se explica brevemente el contenido de cada capítulo:

Capítulo I. Fundamentación Teórica: en este capítulo se define la base teórica de la presente investigación, incluye el análisis de la información existente acerca del tema a tratar y las tendencias actuales que existen en el mundo. También incluye una descripción dando un sustento teórico a la selección de las herramientas, el lenguaje y el *framework* a utilizar para resolver el problema planteado.

Capítulo II. Características y Diseño del sistema: en este capítulo se realiza una breve descripción de la solución propuesta, sus requisitos funcionales y no funcionales. Se describe además la arquitectura propuesta a partir de estilos arquitectónicos y patrones de diseño. Asimismo, se representan los artefactos generados a través del proceso de desarrollo del sistema.

Capítulo III. Implementación y Validación: en este capítulo se muestra parte de la implementación de la solución, especificando el estándar de codificación. Contiene además la validación de la solución y el análisis de los resultados obtenidos en este proceso.

1 FUNDAMENTACIÓN TEÓRICA

En el presente capítulo se abordarán conceptos relacionados con la regresión. Se realizará un estudio del estado del arte de diferentes herramientas centrándose en el proceso de regresión. Como metodología de desarrollo de software se utilizará Proceso Unificado Ágil (AUP por sus siglas en inglés) propuesta por la Universidad de las Ciencias Informáticas. Y por último se hará una caracterización de las herramientas, lenguaje y tecnologías de los que se hará uso para la implementación de la solución.

1.1 MARCO CONCEPTUAL

A continuación, se muestran algunos conceptos tratados durante el transcurso de la investigación, a fin de introducir el tema y lograr una mejor comprensión del trabajo en cuestión.

1.1.1 REGRESIÓN

La regresión es un proceso empleado para estimar los parámetros de cierta función matemática (conocida como modelo) que mejor describa el comportamiento conjunto de las variables que se investigan. Aunque son varios los enfoques que se emplean al utilizar esta técnica, el objetivo siempre radica en estimar la función de regresión, que ayude a comprender cómo cambia el valor típico de una variable dependiente cuando varía cualquiera de las independientes (3).

1.1.2 REGRESIÓN NO LINEAL

La regresión no lineal es una forma de análisis de regresión en la que los datos de observación se modelan mediante una función que depende de manera no lineal de al menos uno de sus parámetros (4).

Los datos consisten en un conjunto de variables independientes que en muchos casos se conciben que están libres de errores x, y sus variables dependientes observadas y; donde cada y es modelada como una variable aleatoria con una media dada por una función no lineal $f(x, \beta)$. De esta forma, una vez minimizada la función objetivo es posible estimar el valor de la variable dependiente a partir de las independientes (1).

1.1.3 OPTIMIZACIÓN GLOBAL Y LOCAL

Los problemas de regresión no lineal pueden tener soluciones locales que no sean soluciones globales a diferencia de otros tipos de problemas de optimización, usualmente, los problemas de regresión lineal, en los que las soluciones locales son además globales.

Según solución local a *"un punto en el que la función objetivo es más pequeño que en todos los demás puntos cercanos factibles"*(1) y por solución global a *"el punto con valor de la función más baja entre todos los puntos posibles"*(1).

1.1.4 ALGORITMOS DE OPTIMIZACIÓN

Los algoritmos de optimización no lineales son iterativos. Estos comienzan con una aproximación inicial de los valores óptimos de las variables y generan una secuencia de soluciones mejoradas hasta hallar la solución. Lo que distingue a un algoritmo de otro es la estrategia empleada para pasar de una iteración a otra. Algunos algoritmos acumulan la información obtenida durante las iteraciones previas, mientras otros simplemente usan la información obtenible desde la iteración en que se encuentra(1).

Para cada problema de optimización dado, lo usual es comparar los resultados obtenidos por la mayor cantidad posible de los algoritmos aplicables al problema; y en general, se encuentra el "mejor" algoritmo de optimización para la situación dada. Sin embargo, realizar esta comparación no siempre resulta una tarea sencilla y se requiere de mucho cuidado cuando se establecen los valores de los parámetros de la función para las pruebas de cada algoritmo (1).

Una certera y fiable manera de comparar dos algoritmos, es ejecutar uno hasta que el valor de la función converja en un valor f_A, y después ejecutar el segundo hasta alcanzar el mismo valor f_A, con el objetivo de medir cuánto tiempo le toma a cada uno alcanzar el mismo valor de la función.

1.2 ANÁLISIS DE TRABAJOS RELACIONADOS

Durante el desarrollo de la investigación se analizaron productos relacionados con la resolución y análisis de problemas de regresión no lineal. Se seleccionaron tres productos que realizan, en cierta medida, funciones homólogas a las de la solución actual.

El estudio de los productos NLREG, GraphPad Prism y OriginPro tuvo como objetivo analizar las funcionalidades que estas brindan para la resolución y análisis de los problemas de NLR (Regresión No Lineal, por sus siglas en inglés) para identificar las capacidades de extensibilidad y así definir en qué medida estas soluciones dan respuesta a la problemática planteada. A continuación, se detallan las características de cada uno de los productos seleccionados.

1.2.1 NLREG – AJUSTE DE CURVAS Y REGRESIÓN NO LINEAL (*NONLINEAR REGRESSION AND CURVE FITTING*)

NLREG es un software de análisis estadístico que realiza análisis de regresión lineal y no lineal, así como ajuste de curvas. NLREG determina los valores de parámetros para una ecuación cuya forma es especificada por el usuario. Asimismo, NLREG puede manejar funciones lineales, polinómicas, exponenciales, logísticas, periódicas, y funciones no lineales en general. NLREG puede manejar prácticamente cualquier función cuya forma pueda ser especificada algebraicamente (5).

NLREG cuenta con un lenguaje de programación completo con una sintaxis similar a C para especificar la función que se va a ajustar a los datos. Esto permite calcular variables de trabajo intermedias, utilizar

condicionales, e incluso iterar en los bucles. Además, dado que el lenguaje NLREG incluye matrices, incluso se pueden utilizar métodos tabulares de consulta para definir la función (5).

1.2.2 GRAPHPAD PRISM

GraphPad Prism es un software estadístico disponible para ordenadores Windows y Mac. GraphPad fue diseñado originalmente para ser empleado por biólogos experimentales en las escuelas de medicina y en las compañías farmacéuticas (6).

Regresión no lineal

GraphPad Prism permite ajustar curvas de manera sencilla seleccionando una ecuación dentro de una amplia lista de ecuaciones de uso común o introduciendo una ecuación propia. Además, ***Prism*** permite ajustar modelos por separado para cada conjunto de datos.

Entre las opciones de ajuste avanzadas que ofrece ***Prism*** se encuentran:

- Informar los intervalos de confianza de los parámetros de ajuste óptimo como rangos asimétricos, que son mucho más precisos que los intervalos simétricos usuales.
- Comparar los ajustes de dos ecuaciones usando una prueba F o el Criterio de Información de Akaike (AIC).
- Identificar valores atípicos.
- Otras (6)

1.2.3 ORIGINPRO

Origin es un software de análisis de datos y de representación gráfica empleado por científicos e ingenieros de industrias comerciales, instituciones académicas y laboratorios gubernamentales (7).

Origin gráfica y analiza los resultados de la regresión y permite actualizar automáticamente los cambios en los datos o en los parámetros, lo que permite crear plantillas para las tareas repetitivas o para realizar operaciones por lotes desde la interfaz de usuario, sin necesidad de programación. Asimismo, es posible conectar ***Origin*** con otras aplicaciones tales como MATLAB ™, LabVIEW ™ o Microsoft © Excel y crear rutinas personalizadas dentro de ***Origin*** utilizando secuencias de comandos o lenguajes de programación como C y Python o la consola R(7).

1.2.4 RESULTADO DEL ESTUDIO

Después de analizar las características y funcionalidades de los productos antes vistos se pudo concluir que son soluciones de alta calidad, y ofrecen funcionalidades muy útiles para los investigadores, como pueden ser determinados análisis estadísticos de las soluciones obtenidas y la posibilidad de crear gráficos. No obstante, la gran mayoría de este tipo de herramientas profesionales se caracterizan por ser especies de cajas negras, limitando a los usuarios a trabajar con los algoritmos definidos por el fabricante.

Uno de los elementos que tienen en común estos productos con el resto de sus homólogos en el mercado es que realizan el proceso de regresión no lineal utilizando la formulación de mínimos cuadrados, que es, sin duda alguna, la formulación más utilizada para resolver este tipo de problemas (8). El algoritmo clásico para resolver problemas de mínimos cuadrados que se encuentra en la literatura es Levenberg-Marquardt[3], que, a su vez, es el que emplean la mayoría de los productos comerciales, incluyendo los citados al inicio de la sección. Sin embargo, mínimos cuadrados no es la única formulación existente para resolver problemas de regresión no lineal.

Por tanto, ninguna de las aplicaciones estudiadas soluciona la problemática planteada, pues en la práctica ocasionalmente se presentan problemas que necesitan de estrategias muy específicas.

1.3 METODOLOGÍA DE DESARROLLO DE SOFTWARE

Las metodologías para el desarrollo imponen un proceso disciplinado sobre el desarrollo de software con el fin de hacerlo más predecible y eficiente. Tienen como principal objetivo aumentar la calidad del software que se produce en todas y cada una de sus fases de desarrollo, haciendo énfasis en la calidad y menor tiempo de construcción del software o lo que es lo mismo "producir lo esperado en el tiempo esperado y con el coste esperado"(9). Cada una de las metodologías de desarrollo tienen características diferentes y son adaptables al proyecto en dependencia de sus particularidades (recursos, equipo de desarrollo, tamaño, entre otras), por lo que se pueden dividir en dos grandes grupos: las ágiles y las robustas o tradicionales.

Tabla 1 Comparación entre las metodologías ágiles y las tradicionales

Metodologías Ágiles	Metodologías Tradicionales
Especialmente preparados para cambios durante el proyecto.	Cierta resistencia a los cambios.
Proceso menos controlado, con pocos principios.	Proceso mucho más controlado, con numerosas políticas/normas.
No existe contrato tradicional o al menos es bastante flexible.	Existe un contrato prefijado.
El equipo de desarrollo es pequeño (menos de 10 integrantes) y trabajando en el mismo sitio, se definen pocos roles y el cliente es parte del equipo.	El equipo de desarrollo es grande y posiblemente distribuido, se definen muchos roles y el cliente interactúa con el equipo mediante reuniones.
Orientadas al resultado del producto, por lo que se genera sólo la documentación más importante con pocos artefactos.	Centradas en la planificación del proyecto definida desde la fase de inicio, por lo que se genera una documentación exhaustiva con muchos artefactos.

[3] Es un algoritmo de optimización local, por tanto, puede arribar a resultados que no satisfacen las restricciones que se imponen sobre algunos problemas.

Menos énfasis en la arquitectura del software.	La arquitectura del software es esencial y se expresa mediante modelos.
Propicio para proyectos pequeños, de corta duración, con requisitos variables.	Propicio para proyectos grandes, de larga duración y con requisitos estables.

A partir de la comparación realizada entre ambos grupos de metodologías y las características propias del proyecto donde se lleva a cabo el proceso de desarrollo, lo más adecuado y acorde es utilizar una metodología ágil, para así lograr un producto de calidad con los recursos y tiempos establecidos.

Precisamente, la Universidad de las Ciencias Informáticas propone la utilización de la metodología AUP (Proceso Unificado Ágil) adaptada al ciclo de vida definido para la actividad productiva de la universidad, por lo que se decide su utilización para guiar el proceso de desarrollo de la solución.

1.3.1 DESCRIPCIÓN DE LA METODOLOGÍA AUP

El Proceso Unificado Ágil fue creado por Scott Ambler, es una versión simplificada del Proceso Unificado de *Rational* (RUP). Este describe de una manera simple y fácil de entender la forma de desarrollar aplicaciones de software de negocio usando técnicas ágiles y conceptos que aún se mantienen válidos en RUP. Se caracteriza por estar dirigido por casos de uso, centrado en la arquitectura y por ser iterativo e incremental. Algunas técnicas usadas por AUP incluyen el desarrollo orientado a pruebas, modelado y gestión de cambios ágiles, y refactorización de base de datos para mejorar la productividad (7).

1.3.2 FASES DE AUP

Al igual que en RUP, en AUP se establecen cuatro fases que transcurren de manera consecutiva: Inicio, Elaboración, Construcción y Transición. De las 4 fases que propone AUP se decide para el ciclo de vida de los proyectos de la UCI mantener la fase de Inicio, pero modificando el objetivo de la misma, se unifican las restantes 3 fases de AUP en una sola, llamada Ejecución y se agrega la fase de Cierre (10).

Para una mayor comprensión se muestra la siguiente tabla:

Tabla 2 Fases de AUP

Fases AUP	Fases (Variación AUP-UCI)	Objetivos de las fases (Variación AUP-UCI)
Inicio	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 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.
Elaboración	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
Construcción		
Transición		

		obtienen los requisitos, se elaboran la arquitectura y el diseño, se implementa y se libera el producto.
	Cierre	En esta fase se analizan tanto los resultados del proyecto como su ejecución y se realizan las actividades formales de cierre del proyecto.

1.3.3 DISCIPLINAS DE AUP

AUP define 7 disciplinas (4 ingenieriles y 3 de gestión de proyectos): Modelo, Implementación, Prueba, Despliegue, Gestión de configuración, Gestión de proyectos y Entorno. Para el ciclo de vida de los proyectos de la UCI se definen 7 disciplinas también, pero a un nivel más atómico. La disciplina Modelo se desglosa en los flujos de trabajos: Modelado de negocio, Requisitos y Análisis y diseño; donde se consideran a cada uno de ellos como disciplinas. Se mantiene la disciplina Implementación. En el caso de Prueba se desagrega en 3 disciplinas: Pruebas Internas, de Liberación y Aceptación. Las restantes 3 disciplinas de AUP asociadas a la parte de gestión para la variación UCI se cubren con las áreas de procesos que define CMMI DEV v1.3 para el nivel 2 (10). Para una mayor comprensión se muestra la siguiente tabla:

Tabla 3 Disciplinas de AUP

Disciplinas AUP	Disciplinas (Variación AUP-UCI)	Objetivos de las disciplinas(Variación AUP-UCI)
Modelo	Modelado de negocio (Opcional)	Destinada a comprender los procesos de una organización y cómo funciona el negocio que se desea informatizar para tener garantías de que el software desarrollado va a cumplir su propósito.
	Requisitos	Enfocada a desarrollar el modelo del sistema que se va a construir. Esta disciplina comprende la administración y gestión de los requisitos funcionales y no funcionales del producto.
	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. Además, se modela el sistema y su forma para que soporte todos los requisitos, incluyendo los requisitos no funcionales.
Implementación	Implementación	En la implementación, a partir de los resultados del Análisis y Diseño, se construye el sistema.
Prueba	Pruebas interna	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.

	Pruebas de liberación	Pruebas 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.
	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.
Gestión de configuración	Se cubren con las áreas de procesos Planeación de Proyecto, Monitoreo y Control de Proyecto, y Administración de la Configuración que propone CMMI DEV v1.3. Las mismas son áreas de procesos de gestión y soporte respectivamente.	
Gestión de proyecto		
Entorno		

1.3.4 ESCENARIO PARA LA DISCIPLINA DE REQUISITOS

Existen tres formas de encapsular los requisitos: Casos de Uso del Sistema (CUS), Historias de usuario (HU) y Descripción de requisitos por proceso (DRP), agrupados en cuatro escenarios condicionados por el Modelado de negocio. Para este escenario en específico no se modela el negocio sólo se puede modelar el sistema a partir de las HU. Es aplicado a los proyectos que hayan evaluado el negocio a informatizar y como resultado obtengan un negocio muy bien definido. Además, el cliente siempre acompañará 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, pues una HU no debe poseer demasiada información.

1.4 TECNOLOGÍAS, LENGUAJES Y HERRAMIENTAS

Para la implementación de la nueva solución se decide hacer uso del lenguaje, las tecnologías y herramientas empleadas en el desarrollo de la solución existente, con el objetivo de reutilizar los componentes desarrollados y de mantener la compatibilidad entre versiones manteniendo como lenguaje de programación C++ y como *framework* de desarrollo Qt en la versión 5.4.

1.4.1 LENGUAJE DE PROGRAMACIÓN C++

C++ es un lenguaje versátil y potente a la hora de realizar sistemas complejos. Debido a la documentación que posee para su entendimiento y utilización es muy empleado para el desarrollo de software. Existen muchos algoritmos y bibliotecas implementadas en C++, por lo que se puede adaptar fácilmente a cualquier solución. Además, C++ tiene a su favor que es utilizado por un conjunto de herramientas libres de desarrollo, es orientado a objetos y en cuanto a ejecución es uno de los más rápidos (11).

1.4.2 FRAMEWORK QT 5.4

Qt es un *framework* multiplataforma y orientado a objetos. La función más conocida de Qt es la creación de interfaces de usuario, sin embargo, no se limita a esto, ya que también provee varias clases para facilitar ciertas tareas de programación, soporte para programación multihilo, comunicación con bases de datos, manejo de cadenas de caracteres y también para el desarrollo de programas sin interfaz gráfica; como herramientas de la consola y servidores. Además, Qt es completamente gratuito para aplicaciones de código abierto y una de sus licencias: la LGPL, permite que se utilice gratuitamente con fines comerciales (12).

1.4.3 ENTORNO DE DESARROLLO INTEGRADO (IDE) QT CREATOR 3.4

Qt Creator trae consigo una serie de componentes integrados: un editor de código C++, un administrador de proyectos, documentación de la API y de todas las funcionalidades que provee Qt. Además, trae una serie de ejemplos de distintos tipos de aplicaciones y la gran mayoría de ellas completamente documentadas (13).

Se empleará como IDE Qt Creator en su versión 3.4, debido a que está orientado al desarrollo de aplicaciones a través del *framework* Qt. Además, es multiplataforma y posee riquezas en sus funciones y bibliotecas permitiendo el desarrollo de interfaces con alto rendimiento empleando C++ (lenguaje de programación seleccionado para el desarrollo).

1.4.4 HERRAMIENTA CASE VISUAL PARADIGM FOR UML 8.0

Visual Paradigm for UML es una herramienta CASE que utiliza UML como lenguaje de modelado. Entre sus principales características se encuentran la disponibilidad en múltiples plataformas y que soporta el ciclo de vida completo del desarrollo de software. Brinda a los usuarios un entorno que posibilita la creación de diagramas para UML 2.0. Permite dibujar todos los tipos de diagramas de clases, código inverso, generar código desde diagramas y generar documentación. Permite generar toda la documentación posible a partir de lo que se hace, bajo determinados estándares previamente establecidos. Está concebido bajo el paradigma orientado a objetos e incorpora el soporte para trabajo en equipo, lo que permite que varios desarrolladores trabajen a la vez en el mismo diagrama y vean en tiempo real los cambios hechos por sus compañeros (14).

Se determina que, para llevar a cabo todo el proceso referente a la ingeniería de software del sistema en desarrollo, la herramienta CASE indicada es *Visual Paradigm for UML* en la versión 8.0. Se selecciona en primer lugar porque una herramienta CASE que se utiliza en gran variedad de proyectos de gran alcance y al ser multiplataforma puede ser utilizada en diferentes sistemas operativos. Además, porque contribuye a lograr mayor rapidez en la construcción de aplicaciones informáticas.

1.5 CONCLUSIONES PARCIALES

A partir del análisis e investigación de los aspectos abordados en este capítulo se puede concluir que:

- Los *software* de regresión estudiados no resuelven el problema que se plantea, por lo que se hace necesario diseñar e implementar una solución.
- Se empleará la metodología de desarrollo AUP propuesta por la UCI para guiar el proceso de desarrollo de software de forma simple, ágil y centrado en actividades de alto valor, que son esenciales para el perfeccionamiento del diseño e implementación de la solución.
- Se utilizarán las mismas herramientas, lenguaje y tecnologías que fueron empleados para el desarrollo de la solución existente, permitiendo reutilizar componentes y ahorrar tiempo de implementación.

2 CARACTERÍSTICAS Y DISEÑO DEL SISTEMA

En este capítulo se describirá la propuesta de solución mediante el análisis y diseño del sistema, tomando como referencia los conceptos y términos definidos en el primer capítulo, y regido por la metodología de desarrollo AUP. Se hará mayor énfasis en la arquitectura propuesta, describiendo los estilos arquitectónicos y patrones de diseño empleados.

2.1 DESCRIPCIÓN DEL PROCESO A AUTOMATIZAR

Con la aplicación actual los investigadores pueden obtener una solución a un problema de regresión no lineal a partir de un conjunto de datos, empleando algoritmos con una tarea muy específica dentro del proceso. Para solucionar un problema, los investigadores deben seguir el siguiente procedimiento:

1. **Importar los datos:** se importan los datos del problema de regresión no lineal a resolver desde una fuente de datos determinada.
2. **Refinar los datos *:** opcionalmente es posible, entre otras tareas; refinar los datos antes de iniciar la optimización, con el objetivo de eliminar ruidos, empleando algoritmos de pre-procesamiento implementados para ello.
3. **Seleccionar el modelo:** se selecciona el modelo no lineal con el cual se desea realizar el ajuste de los datos y que se empleará junto con las observaciones (datos) para construir la función objetivo.
4. **Seleccionar la función objetivo:** se selecciona el tipo de función objetivo que se empleará para resolver el problema en cuestión.
5. **Seleccionar los algoritmos de optimización:** para minimizar la función objetivo se selecciona un algoritmo de optimización global, uno local o una combinación de ambos; en dependencia de la solución deseada (hallar el mínimo global o local).
6. **Ejecutar el proceso de optimización.**

La deficiencia en este procedimiento es que el investigador se ve obligado a realizar todo el proceso desde el inicio si quisiera resolver el mismo problema empleando algoritmos diferentes. En la mayoría de los casos los investigadores necesitan conocer o medir la calidad del ajuste, realizar análisis estadísticos a los datos de entrada, así como comparar la calidad de las soluciones que se obtienen empleando diferentes modelos y algoritmos. Con la solución existente sólo se pueden hacer estas tareas de forma manual, a vista o modificando el código fuente.

Para resolver los problemas de flexibilidad, extensibilidad y facilidad de mantenimiento presentados se propone modificar la arquitectura de *plugins* existente para obtener una aplicación orientada a la resolución de problemas de regresión no lineal.

2.2 MODELO DEL SISTEMA

Un modelo del sistema describe lo que supuestamente hará un sistema, pero no cómo implementar dicho sistema. Idealmente, una representación de un sistema, debería mantener toda la información sobre la entidad que se está representando (15).

2.2.1 REQUISITOS FUNCIONALES

Los requisitos funcionales son declaraciones de los servicios que debe proporcionar el sistema, de la manera en que éste debe reaccionar a entradas particulares y de cómo se debe comportar en situaciones específicas. Estos requisitos dependen del tipo de software que se desarrolle, de los posibles usuarios del software y del enfoque general tomado por la organización al redactar requisitos (16). A continuación, se muestran los requisitos funcionales identificados en la problemática en cuestión:

- **RF1** Gestionar regiones
  - **RF1.1** Adicionar región
  - **RF1.2** Obtener regiones por nombre
- **RF2** Adicionar *widgets* a las regiones
- **RF3** Crear vista principal de la aplicación
- **RF4** Configurar el contenedor de dependencia
- **RF5** Gestionar servicios
  - **RF5.1** Registrar servicios
  - **RF5.2** Obtener la instancia de servicios
  - **RF5.3** Administrar el tiempo de vida de los servicios
- **RF6** Cargar *plugins*

2.2.2 ESPECIFICACIÓN DE REQUISITOS

En la Tabla 4 se describen los requisitos identificados:

Tabla 4 Especificación de requisitos

No	Nombre	Descripción	Prioridad	Complejidad
RF1	Gestionar regiones	El sistema debe permitir al usuario adicionar una colección de regiones en el administrador de regiones, y obtener una región dado su nombre.	Alta	Alta

RF2	Adicionar *widgets* a las regiones	El sistema debe permitir al usuario adicionar *widgets* a las regiones de la aplicación.	Alta	Alta
RF3	Crear vista principal de la aplicación	El sistema debe permitir al usuario crear la vista principal de la aplicación con los elementos visuales deseados.	Alta	Alta
RF4	Configurar el contenedor de dependencia	El sistema debe permitir al usuario configurar las dependencias de los componentes principales de la aplicación.	Alta	Alta
RF5	Gestionar servicios	El sistema debe permitir al usuario registrar servicios y obtener instancias de servicios, así cómo administrar tiempos de vida de los servicios.	Alta	Alta
RF6	Cargar *plugins*	El sistema debe permitir al usuario cargar los *plugins* de la aplicación.	Alta	Alta

2.2.3 REQUISITOS NO FUNCIONALES

Los requerimientos no funcionales son restricciones de los servicios o funciones ofrecidos por el sistema. Incluyen restricciones de tiempo, sobre el proceso de desarrollo y estándares. Los requisitos no funcionales a menudo se aplican al sistema en su totalidad. Normalmente apenas se aplican a características o servicios individuales del sistema (16).

Por su parte, la calidad del producto software se puede interpretar como el grado en que dicho producto satisface los requisitos de sus usuarios aportando de esta manera un valor. Con el objetivo de estandarizar la redacción de los requisitos no funcionales (RNF) de Atributos de calidad se utiliza una taxonomía partiendo de la ISO 9126. En la Tabla 5, se muestran los requisitos no funcionales del sistema.

Tabla 5 Requisitos no funcionales

Requisito no funcional	Descripción	Sub-atributos de calidad
Confiabilidad	Capacidad de un sistema o componente para desempeñar las funciones especificadas, cuando se usa bajo unas condiciones y período de tiempo determinados.	**Madurez**: Capacidad del sistema para satisfacer las necesidades de fiabilidad en condiciones normales.
Mantenibilidad	Capacidad del producto software para ser modificado efectiva y eficientemente, debido a necesidades evolutivas, correctivas o perfectivas.	**Analizabilidad:** Facilidad con la que se puede evaluar el impacto de un determinado cambio sobre el resto del software, diagnosticar las deficiencias o causas de fallos en el

		software, o identificar las partes a modificar. **Cambiabilidad:** Capacidad del producto que permite que sea modificado de forma efectiva y eficiente sin introducir defectos o degradar el desempeño.
Portabilidad	Capacidad del producto o componente de ser transferido de forma efectiva y eficiente de un entorno hardware, software, operacional o de utilización a otro.	**Adaptabilidad:** Capacidad del producto que le permite ser adaptado de forma efectiva y eficiente a diferentes entornos determinados de hardware, software, operacionales o de uso.
Compatibilidad	Capacidad de dos o más sistemas o componentes para intercambiar información y/o llevar a cabo sus funciones requeridas cuando comparten el mismo entorno hardware o software.	**Co-existencia:** Capacidad de un producto para coexistir con otro software independiente, en un entorno común, compartiendo recursos comunes sin detrimento.
Funcionabilidad	Capacidad de un producto de software para proporcionar funciones que satisfacen las necesidades declaradas e implícitas, cuando el producto se usa en las condiciones especificadas.	**Funcionabilidad:** Grado en el cual el conjunto de funcionalidades cubre todas las tareas y los objetivos del usuario especificados.
Eficiencia	Representa el desempeño relativo a la cantidad de recursos utilizados bajo determinadas condiciones.	**Utilización de recursos:** Las cantidades y tipos de recursos utilizados cuando el software lleva a cabo su función bajo condiciones determinadas.

El Anexo Descripción de requisitos no funcionales muestra el objetivo y el origen de los requisitos no funcionales y sus sub-atributos de calidad, así como el entorno donde se aplica. Además, detalla el flujo de eventos que responde a un estímulo aplicado a cada entorno.

2.2.4 DESCRIPCIÓN DE HISTORIAS DE USUARIO

Entre los artefactos que define la metodología seleccionada se encuentran las historias de usuario (HU) que son utilizadas para especificar las funcionalidades que brindará el sistema. Cada historia de usuario es una representación de un requisito de software escrito en una o dos frases utilizando el lenguaje común del usuario; además, estas brindan 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 (17).

Tabla 6 HU1 Gestionar regiones

Número: 1	**Nombre del requisito:** Gestionar regiones
Programador: Francisco Javier Amador Sala	**Iteración Asignada:** 1
Prioridad: Alta	**Tiempo Estimado:** 20 días
Riesgo en Desarrollo: ▪ Rotura de alguna estación de trabajo perteneciente al proyecto ▪ Problemas eléctricos	**Tiempo Real:** 4 semanas
Descripción: El sistema deberá permitir al usuario añadir una región en el administrador de regiones, representado por una tabla Hash. Cada región deberá contener un identificador que estará dado por su nombre, para que luego se visualice en la interfaz de usuario. Luego el sistema podrá obtener una región almacenada dentro del administrador de regiones a partir del identificador (nombre de la región).	

Tabla 7 HU2 Adicionar widgets a las regiones

Número: 2	**Nombre del requisito:** Adicionar *widgets* a las regiones
Programador: Francisco Javier Amador Sala	**Iteración Asignada:** 1
Prioridad: Alta	**Tiempo Estimado:** 10 días
Riesgo en Desarrollo: ▪ Rotura de alguna estación de trabajo perteneciente al proyecto ▪ Problemas eléctricos	**Tiempo Real:** 2 semanas
Descripción: El sistema deberá permitir al usuario adicionar *widgets* a la región de forma tal que se pueda decidir cómo se va a agregar un hijo a la región.	

Tabla 8 HU3 Crear vista principal de la aplicación

Número: 3	**Nombre del requisito:** Crear vista principal de la aplicación
Programador: Francisco Javier Amador Sala	**Iteración Asignada:** 1
Prioridad: Alta	**Tiempo Estimado:** 10 días
Riesgo en Desarrollo: ▪ Rotura de alguna estación de trabajo perteneciente al proyecto ▪ Problemas eléctricos	**Tiempo Real:** 2 semanas
Descripción: El sistema deberá permitir al usuario crear la vista principal de la aplicación la cual contendrá las regiones	

que fueron añadidas al administrador de regiones.

Tabla 9 HU4 Configurar el contenedor de dependencia

Número: 4	**Nombre del requisito:** Configurar el contenedor de dependencia
Programador: Francisco Javier Amador Sala	**Iteración Asignada:** 1
Prioridad: Alta	**Tiempo Estimado:** 15 días
Riesgo en Desarrollo: ▪ Rotura de alguna estación de trabajo perteneciente al proyecto ▪ Problemas eléctricos	**Tiempo Real:** 3 semanas
Descripción: El sistema deberá permitir al usuario configurar el localizador de servicios con las dependencias de los componentes principales de la aplicación de forma tal que sean accesibles desde los *plugins*.	

Tabla 10 HU5 Gestionar servicios

Número: 5	**Nombre del requisito:** Gestionar servicios
Programador: Francisco Javier Amador Sala	**Iteración Asignada:** 1
Prioridad: Alta	**Tiempo Estimado:** 25 días
Riesgo en Desarrollo: ▪ Rotura de alguna estación de trabajo perteneciente al proyecto ▪ Problemas eléctricos	**Tiempo Real:** 5 semanas
Descripción: El sistema deberá permitir al usuario registrar una instancia de servicios o un tipo de datos. En caso de ser una instancia de servicio deberá recibir una instancia de un objeto y opcionalmente el nombre. Si es un tipo de datos deberá registrar un tipo asociado a un subtipo y opcionalmente el nombre, además del administrador de tiempos de vida. El sistema deberá obtener una instancia de una clase de servicios, a partir, opcionalmente, del nombre de la instancia o del tipo. El sistema deberá administrar el tiempo de vida de las instancias de servicios en dependencia de si es un una instancia única o si es temporal.	

Tabla 11 HU6 Cargar plugins

Número: 6	**Nombre del requisito:** Cargar *plugins*
Programador: Francisco Javier Amador Sala	**Iteración Asignada:** 1
Prioridad: Alta	**Tiempo Estimado:** 10 días

Riesgo en Desarrollo: ▪ Rotura de alguna estación de trabajo perteneciente al proyecto ▪ Problemas eléctricos	**Tiempo Real:** 2 semanas
Descripción: El sistema deberá permitir al usuario cargar los *plugins* creados que pueden estar almacenados en una ruta especificada o pueden ser insertados a la interfaz visual manualmente.	

2.3 DESCRIPCIÓN DE LA ARQUITECTURA

El diseño de la arquitectura de un sistema es el proceso por el cual se define una solución para los requisitos técnicos y operaciones de mismo. Este proceso define los componentes que forman parte del sistema, las relaciones entre ellos, y cómo mediante su interacción llevan a cabo la funcionalidad especificada cumpliendo con los criterios de calidad (18).

La arquitectura del software es la representación a mayor nivel de la estructura de un sistema, que aporta elementos necesarios para la toma de decisiones y, al mismo tiempo, proporciona conceptos y un lenguaje común que posibilita la comunicación entre los equipos que participen en un proyecto determinado (18).

2.3.1 PRINCIPIOS DE DISEÑO SOLID

A la hora de diseñar un sistema, es importante tener presente una serie de principios de diseño fundamentales que ayudarán a crear una arquitectura que se ajuste a prácticas demostradas, que minimicen los costes de mantenimiento y maximicen la usabilidad y la extensibilidad. Estos principios clave seleccionados y muy reconocidos por la industria del software, son los Principios de Diseño SOLID.

El acrónimo SOLID deriva de las siguientes frases/principios en inglés:

- *Single Responsability Principle*
- *Open Closed Principle*
- *Liskov Subtitution Principle*
- *Interface Segregation Principle*
- *Dependency Inversion Principle* (19)

Principio de Única Responsabilidad (*Single Responsability Principle*):

Una clase debe tener una única responsabilidad o característica. Dicho de otra manera, una clase debe de tener una única razón por la que está justificado realizar cambios sobre su código fuente. Una consecuencia de este principio es que, de forma general, las clases deberían tener pocas dependencias con otras clases/tipos (19).

Principio Abierto Cerrado (*Open Closed Principle*):

Una clase debe estar abierta para la extensión y cerrada para la modificación. Es decir, el comportamiento de una clase debe poder ser extendido sin necesidad de realizar modificaciones sobre el código de la misma (18).

Principio de Sustitución de Liskov (*Liskov Subtitution Principle*):

Los subtipos deben poder ser sustituibles por sus tipos base (interfaz o clase base). Este hecho se deriva de que el comportamiento de un programa que trabaja con abstracciones (interfaces o clases base) no debe cambiar porque se sustituya una implementación concreta por otra. Los programas deben hacer referencia a las abstracciones, y no a las implementaciones. Este principio está muy relacionado con la Inyección de Dependencias y la sustitución de unas clases por otras siempre que cumplan la misma interfaz (18).

Principio de Segregación de Interfaces (*Interface Segregation Principle*):

Los implementadores de Interfaces de clases no deben estar obligados a implementar métodos que no se usan. Es decir, las interfaces de clases deben ser específicas dependiendo de quién las consume y por lo tanto, tienen que estar granularizadas/segregadas en diferentes interfaces; no debiendo crear nunca grandes interfaces. Las clases deben exponer interfaces separadas para diferentes clientes/consumidores que difieren en los requerimientos de interfaces (18).

Principio de Inversión de Dependencias (*Dependency Inversion Principle*):

Las abstracciones no deben depender de los detalles. Los detalles deben depender de las abstracciones. Las dependencias directas entre clases deben ser reemplazadas por abstracciones (interfaces) para permitir diseños *top-down* sin requerir primero el diseño de los niveles inferiores (18).

2.3.2 ESTILOS ARQUITECTÓNICOS

Para resolver el problema de la investigación se necesita desarrollar una plataforma extensible; que brinde facilidades para la integración de nuevas funcionalidades y tecnologías, y con la capacidad de adaptarse a los cambios de requerimientos.

A partir del estudio realizado se propone una arquitectura basada en la combinación de los estilos arquitectónicos: arquitectura basada en capas, arquitectura basada en componentes y arquitectura basada en *plugins*.

Arquitectura basada en Capas

Se define al estilo en capas como una organización jerárquica tal que cada capa proporciona servicios a la capa inmediatamente superior y se sirve de las prestaciones que le brinda la inmediatamente inferior. El estilo soporta un diseño basado en niveles de abstracción, lo cual a su vez permite a los implementadores la partición de un problema complejo en una secuencia de pasos incrementales. En segundo lugar, el estilo admite optimizaciones y refinamientos, y proporciona una amplia reutilización.

Al igual que los tipos de datos abstractos, se pueden utilizar diferentes implementaciones o versiones de una misma capa en la medida que soporten las mismas interfaces de cara a las capas adyacentes (18).

Finalmente, la solución propuesta quedó estructurada en dos capas (como se muestra en la Figura 1). La capa ***Core*** es la de más bajo nivel, y ofrece soporte a la resolución de problemas matemáticos de regresión no lineal. En esta se realiza el procesamiento de los datos y ejecución de los algoritmos y, además, contiene los servicios de infraestructura.

La capa ***App*** de más alto nivel proporciona soporte al sistema de extensibilidad basado en *plugins*, implementa la infraestructura para la creación de aplicaciones compuestas mediante el uso de regiones y ofrece servicios de alto nivel para el consumo de los *plugins*.

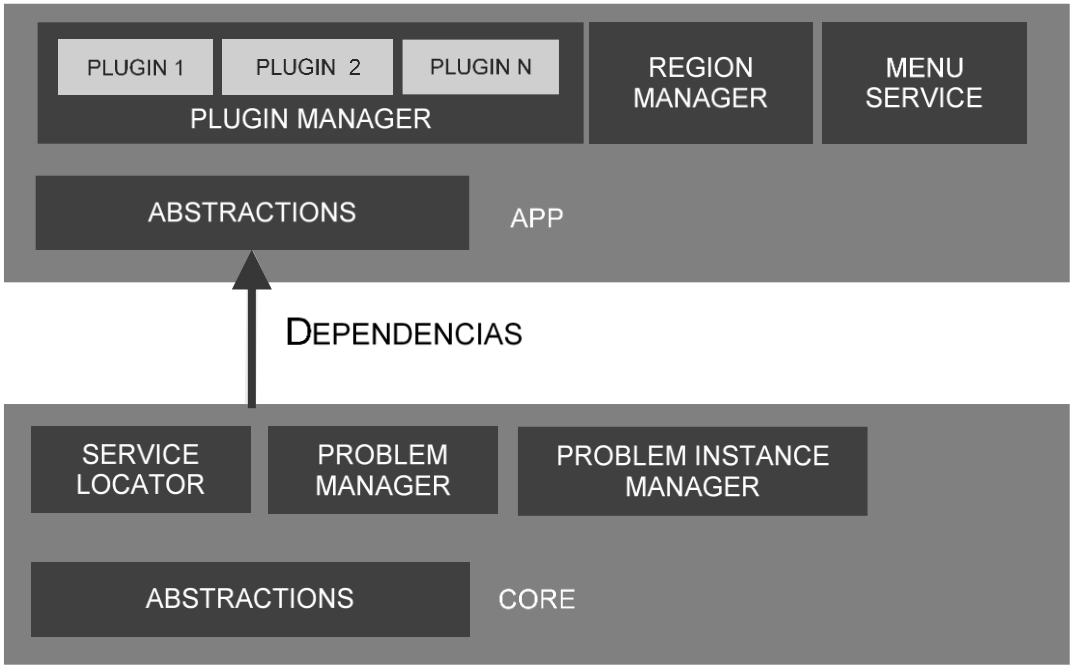

Figura 1 Distribución de las capas en la arquitectura propuesta

Finalmente se puede observar que en cada capa existe un conjunto de abstracciones que no son más que interfaces, para dar cumplimiento al principio de Inversión de Dependencias, donde cada implementación de un aspecto concreto depende de una abstracción y no de otra implementación. Quedando de esta forma lo que algunos autores reconocen por arquitectura invertida, donde las capas inferiores dependen de las abstracciones de las capas superiores, esto se logra mediante el uso del patrón Inversión de Control (IoC, por su siglas en inglés): "*Se delega a un componente o fuente externa, la función de seleccionar un tipo de implementación concreta de las dependencias de nuestras clases"*(18). Este patrón describe técnicas para soportar una arquitectura tipo *plugin* donde los objetos pueden buscar instancias de otros objetos que requieren y de los cuales dependen.

Entre las diferentes capas no se instancian de forma explícita las dependencias. Para conseguir esto, se puede hacer uso de una clase base o un interfaz que defina una abstracción común que pueda ser utilizada

para inyectar instancias de objetos en componentes que interactúen con dicha interfaz abstracta compartida.

Arquitectura basada en Componentes

Cada capa de la solución contiene una serie de componentes que implementan la funcionalidad de dicha capa. El diseño de componentes debe ser altamente cohesivo: no sobrecargar los componentes añadiendo funcionalidad mezclada o no relacionada. Por ejemplo, el componente ***ProblemManager*** de la capa ***Core***, solo se encarga del manejo de entidades que representan un problema de regresión no lineal. Cuando la funcionalidad es cohesiva, entonces se puede crear bibliotecas que contengan más de un componente y situar los componentes en las capas apropiadas de la aplicación. Este principio está por lo tanto muy relacionado con el principio de *"Single Responsability Principle".* Por otra parte, el desarrollo de software basado en componentes busca, entre otros objetivos, reducir el tiempo de trabajo, el esfuerzo que requiere implementar una aplicación y los costes del proyecto, incrementando de esta forma el nivel de productividad y minimizando los riesgos.

Existen varias definiciones de componentes realizadas por expertos. De acuerdo con el Instituto de Ingeniería de Software, "*un componente es una implementación opaca de una funcionalidad, que está sujeto a composición por terceros y que cumple con un modelo de componentes"* (20).

La principal motivación para el criterio de que un componente es una implementación opaca se debe a un precepto bien establecido en la informática: los clientes de componentes de software no deben depender de los detalles de implementación que pueden cambiar en el futuro. En informática esto ha conducido al soporte de programación para las abstracciones y la ocultación de información; la opacidad sirve para lo mismo en los componentes. En la solución propuesta cada componente desarrollado posee un interfaz que los demás componentes usan para acceder a las funcionalidades.

La motivación para la composición de terceros es simple: el uso de componentes no debe depender de herramientas o conocimientos que están en posesión del proveedor del componente. Este criterio implica que un sistema basado en componentes puede comprender componentes de fuentes múltiples e independientes; y además el sistema puede ser integrado por un tercero que no es el proveedor de los componentes. Este criterio debe ser cierto, incluso, si ninguno de los componentes utilizados en un sistema proviene de proveedores externos.

El último criterio, establece que los modelos de componentes determinan cómo los componentes interactúan entre sí, y por lo tanto expresan restricciones globales, o de diseño arquitectónico. La conformidad con los modelos de componentes transforma las implementaciones de software en las implementaciones de arquitectura, los sistemas basados en componentes se fundamentan en la uniformidad y en esquemas de coordinación estandarizados. En la solución que se propone, el modelo de componentes resulta una mezcla entre las interfaces definidas para cada componente, y las impuestas

por el *framework* Qt, como la herencia de la clase base **QObject** que permite el uso de ***signls*** y ***slots*** además del manejo de memoria automático, así como el uso de técnicas avanzadas como *reflection*, esto se puede observar en la Figura 2 donde se hace uso de la clase **QMetaObject** para obtener información de los componentes en tiempo de ejecución (línea 10).

```
1  #include "qservicelocator.h"
2  #include "singletonlifetimemanager.h"
3  #include "lifetimemanager.h"
4
5  QServiceLocator QServiceLocator::container;
6
7  void QServiceLocator::privateRegisterInstance(QObject *instance, const QSt
8  {
9      SingletonLifetimeManager* manager = new SingletonLifetimeManager();
10     QString type = instance->metaObject()->className();
11    // manager->setUp(type);
12     manager->SetValue(instance);
13
14     Registration *registration = new Registration();
15     registration->setLifeTimeManager(manager);
16     registration->setMappedToType(type);
17     registration->setName(name.isEmpty()?DEFAULT_NAME:name);
18     registration->setRegisteredType(registerType);
19
20     add(registration);
21 }
```

*Figura 2 Fragmento de código de la clase **QServiceLocator***

Arquitectura basada en *Plugins*

Se considera un *plugin* a cualquier pieza de software que extienda o cambie el comportamiento o interfaz de una aplicación. Las arquitecturas orientadas a *plugins* proveen una manera de extender las capacidades de una aplicación adicionando nuevas funcionalidades; por esta razón, la solución propuesta emplea una combinación de estilos arquitectónicos que incluye a la arquitectura basada en *plugins*, pues la misma resulta útil cuando se diseña una aplicación que debe acomodarse a requisitos que deban ser implementados en el futuro.

En la solución propuesta, los plugins se implementaron basados en el mecanismo que ofrece el *framework* Qt. Las aplicaciones pueden ser extendidas mediante *plugins* haciendo uso de la clase ***QPluginLoader***, la cual permite detectar y cargar los *plugins* en tiempo de ejecución.

2.4 PATRONES DE DISEÑO UTILIZADOS

En el diseño e implementación de la propuesta de solución, se tuvo presente el estudio realizado sobre los patrones de diseño y de asignación general de responsabilidades (ver secciones 1.3 y 1.4), teniendo en cuenta las características de la arquitectura propuesta se utilizaron los patrones: Singleton, Adapter, Service Locator, Composite y Command.

Singleton

Se asegura de que una clase sólo tenga una instancia y provee un punto global de acceso a esta (21). Este patrón se empleó en la clase QServiceLocator.

***Adapter* (Adapador)**

Convierte la interfaz de una clase en otra interfaz que el cliente espera. El patrón *Adapter* permite trabajar en conjunto, a clases que no podrían hacerlo, debido a que las interfaces son incompatibles (21). El patrón se evidencia en las clases que implementan la interfaz IRegion.

***Service Locator* (Localizador de Servicio)**

El patrón de localizador de servicio es un patrón de diseño utilizado en el desarrollo de software para encapsular los procesos que intervienen en la obtención de un servicio con una fuerte capa de abstracción. Este patrón utiliza un registro central, conocido como el "localizador de servicios", que en la solicitud devuelve la información necesaria para realizar una tarea determinada (22). Este patrón se evidencia en la clase QServiceLocator.

***Composite* (Composición)**

Compone objetos en estructuras de árbol para representar jerarquías. El patrón *Composite* permite a los clientes tratar objetos individuales y composiciones de objetos uniformemente (21). Este patrón se evidencia en las regiones, pues una región puede estar compuesta por otras regiones.

***Command* (Orden)**

El patrón *Command* permite solicitar una operación a un objeto sin conocer el contenido de esta operación, ni el receptor de la misma, encapsulando la petición como un objeto (21). Este patrón se refleja en la funcionalidad adicionar acción de la clase ***IMenuService***.

2.5 MODELO DEL DISEÑO

Un modelo de diseño se encuentra más cerca de la solución que se desea obtener y adquiere algunas de las entidades reflejadas en el modelo de dominio para convertirlas en clases. Su propósito es especificar una solución que trabaje y pueda convertirse fácilmente en código fuente. Representa la solución a las HU definidas y es utilizado como entrada en las tareas de implementación.

2.5.1 DIAGRAMA DE PAQUETES

Un paquete es un mecanismo utilizado para agrupar elementos de UML. Permite organizar los elementos modelados con UML, facilitando de ésta forma el manejo de los modelos de un sistema complejo. En la Figura 3 se muestra el diagrama de paquetes que componen la solución propuesta.

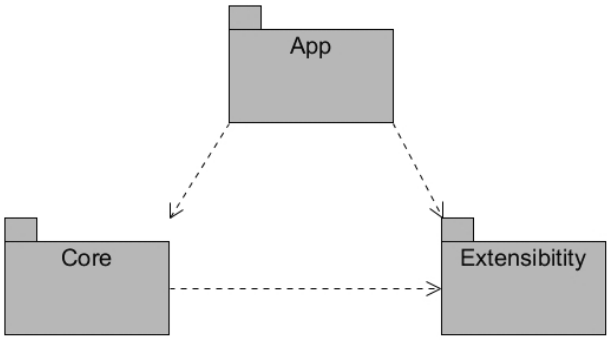

Figura 3 Diagrama de Paquetes

2.5.2 DIAGRAMA DE CLASES

Un diagrama de clases es una representación de las clases que involucran al sistema y las relaciones entre ellas que pueden ser de asociación, de herencia, de uso y de agregación. Una clase es una definición de un conjunto de entidades u objetos que comparten los mismos atributos, operaciones y relaciones (23). Los diagramas de clases que componen los paquetes de la solución propuesta se muestran en las Figuras 4, 5, 6 y 7.

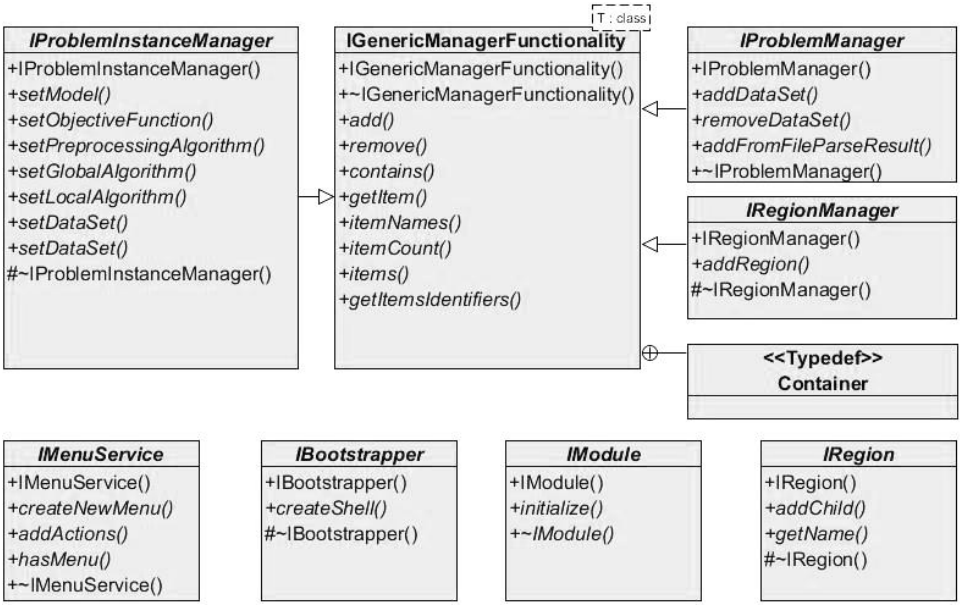

Figura 4 Diagrama de clases del paquete Extensibility

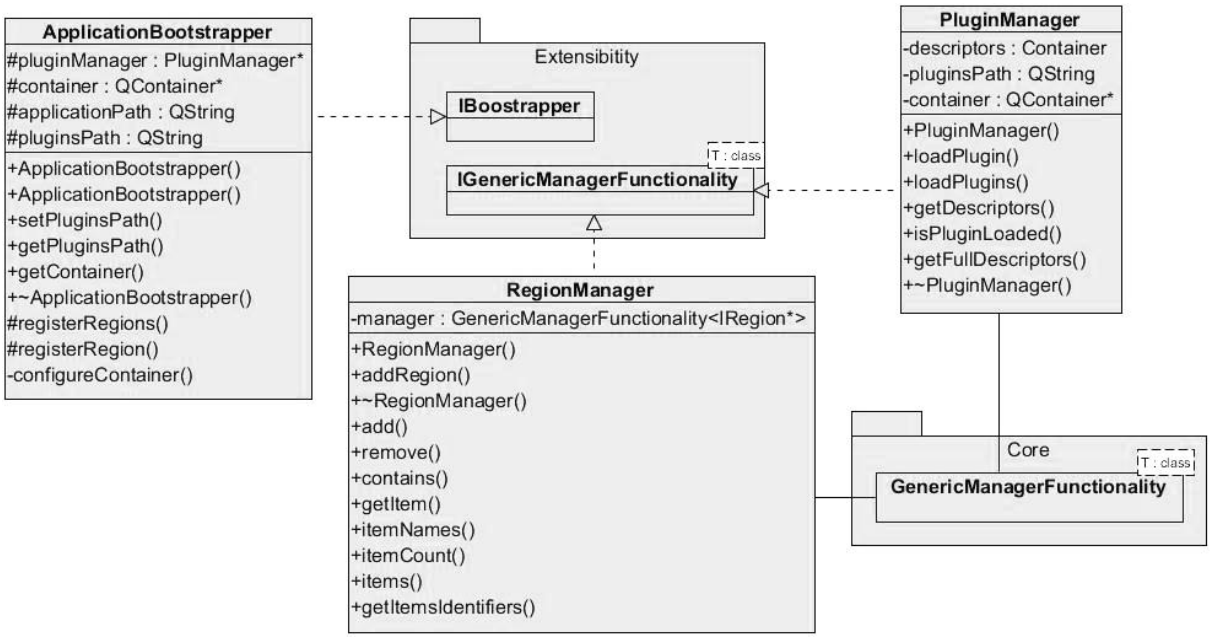

Figura 5 Diagrama de clases del paquete App

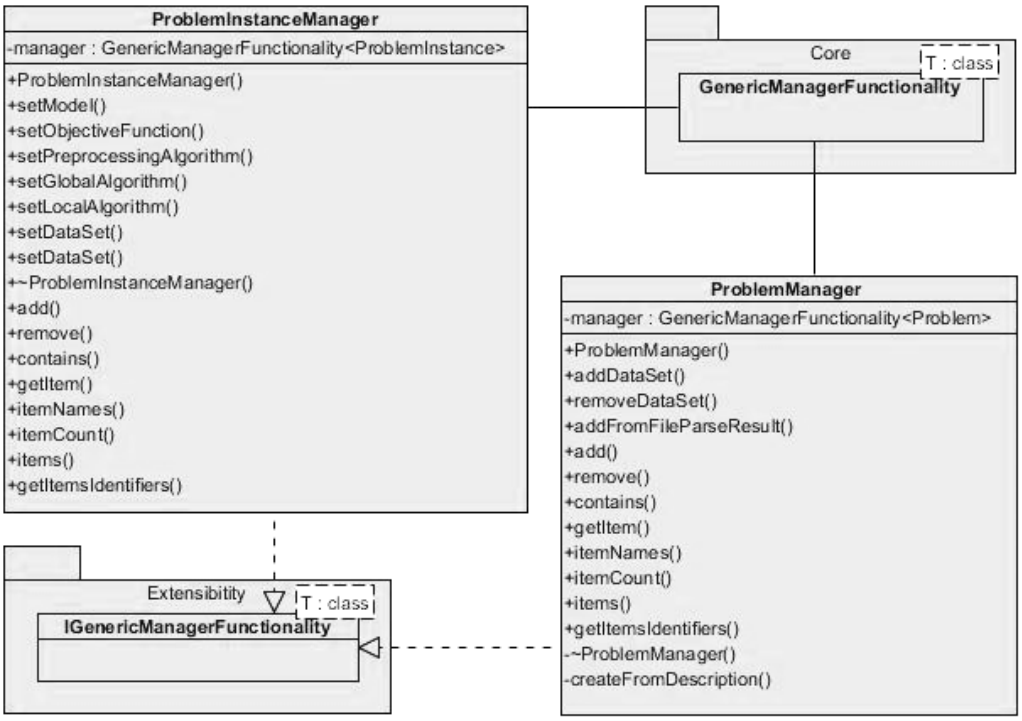

Figura 6 Diagrama de clases del paquete Core [1ra Parte]

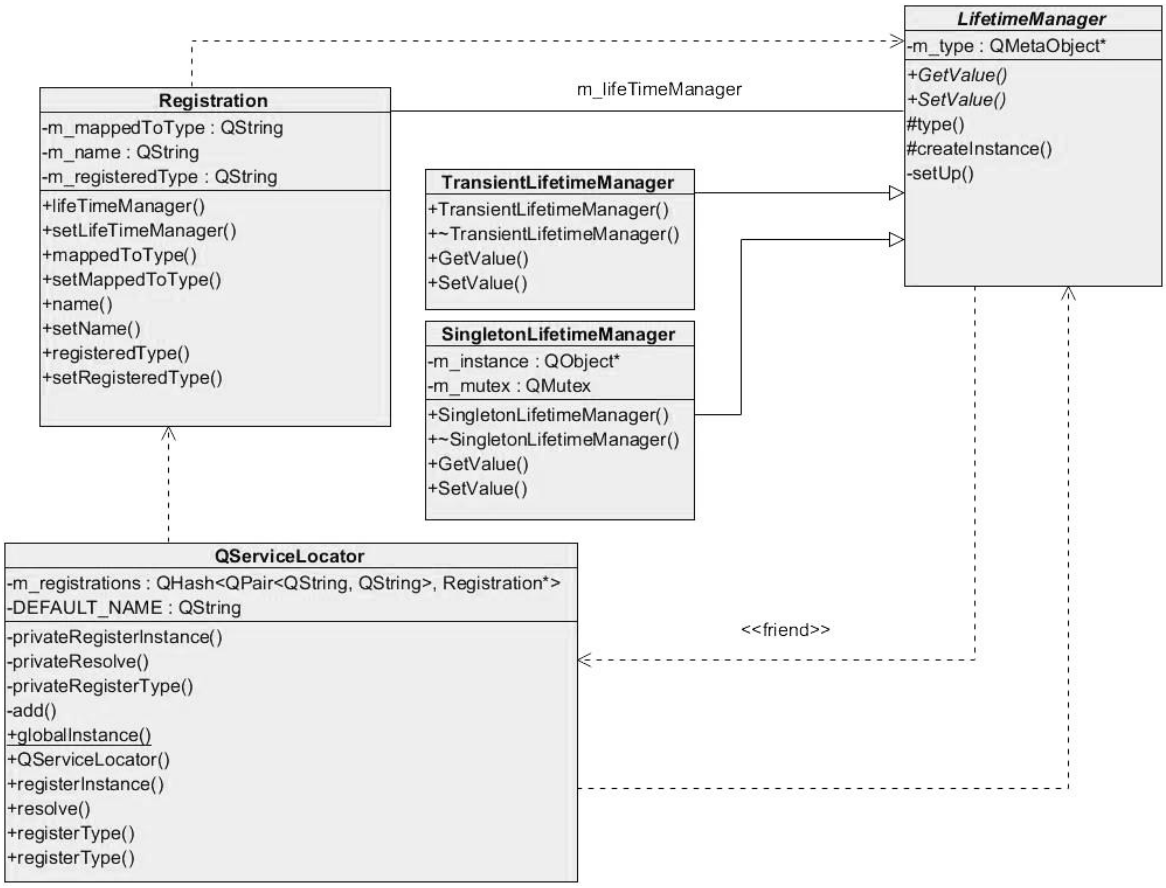

Figura 7 Diagrama de clases del paquete Core [2da Parte]

2.5.3 DESCRIPCIÓN DE LAS CLASES

A continuación, se muestran las descripciones de las clases y métodos principales de la solución propuesta:

Tabla 12 Descripción de la clase IBootstrapper

Descripción de la clase IBootstrapper	
Nombre: IBootstrapper	
Tipo de clase: Interfaz	
Responsabilidades	
Nombre: createShell()	
Descripción: Crea el Shell y configura el localizador de servicios con los componentes del núcleo.	

Tabla 13 Descripción de la clase IGenericManagerFunctionality

Descripción de la clase IGenericManagerFunctionality	
Nombre: IGenericManagerFunctionality	
Tipo de clase: Interfaz	
Atributo:	**Tipo:**

Container	public
Responsabilidades	
Nombre: add(pName: QString, pValue: T): bool	
Descripción: Adiciona un objeto de tipo T identificado por su nombre.	
Nombre: remove(pName: QString): bool	
Descripción: Elimina un objeto dado su nombre.	
Nombre: contains(pName: QString): bool	
Descripción: Verifica que existe un objeto dado su nombre.	
Nombre: getItem(pName: QString): T	
Descripción: Devuelve un objeto dado su nombre.	
Nombre: itemNames(): QList<QString>	
Descripción: Devuelve una lista con todos los nombres de los objetos.	
Nombre: itemCount(): int	
Descripción: Devuelve la cantidad de objetos existentes.	
Nombre: items(): Container	
Descripción: Devuelve objeto de tipo Container que contiene todos los objetos existentes.	
Nombre: getItemsIdentifiers(): QList<QString>	
Descripción: Devuelve una lista con los identificadores de los objetos.	

Tabla 14 Descripción de la clase IMenuService

Descripción de la clase IMenuService
Nombre: IMenuService
Tipo de clase: Interfaz
Responsabilidades
Nombre: createNewMenu(pCaption: QString): bool
Descripción: Crea un submenú en el menú principal de la aplicación dado el nombre del menú.
Nombre: addActions(pMenuName: QString, pActions: QList<QAction*>)
Descripción: Añade una lista de acciones a un menú dado.
Nombre: hasMenu(pMenuName: QString): bool
Descripción: Verifica que existe un menú dado su nombre.

Tabla 15 Descripción de la clase IModule

Descripción de la clase IModule
Nombre: IModule
Tipo de clase: Interfaz
Responsabilidades
Nombre: initialize(container: QServiceLocator*)
Descripción: Inicializa el módulo que implementa la interfaz.

Tabla 16 Descripción de la clase IProblemInstanceManager

Descripción de la clase IProblemInstanceManager
Nombre: IProblemInstanceManager
Tipo de clase: Interfaz
Responsabilidades
Nombre: setModel(pInstanceName: QString, pModel: IModel*): bool
Descripción: Establece el modelo de la instancia de un problema dado su nombre.
Nombre: setObjectiveFunction(pInstanceName: QString, pModel: IObjectiveFunction*): bool
Descripción: Establece la función objetivo de la instancia de un problema dado su nombre.
Nombre: setPreprocessingAlgorithm(pInstanceName: QString, pModel: IPreprocessingAlgorithm*): bool
Descripción: Establece el algoritmo de pre procesamiento de la instancia de un problema dado su nombre.
Nombre: setGlobalAlgorithm(pInstanceName: QString, pModel: IGlobalAlgorithm*): bool
Descripción: Establece el algoritmo de optimización global de la instancia de un problema dado su nombre.
Nombre: setLocalAlgorithm(pInstanceName: QString, pModel: ILocalAlgorithm*): bool
Descripción: Establece el algoritmo de optimización local de la instancia de un problema dado su nombre.
Nombre: setDataSet(pInstanceName: QString, pDataSet: DataSet): bool
Descripción: Establece la fuente de datos de la instancia de un problema dado su nombre.
Nombre: setDataSet(pInstanceName: QString, pProblem: Problem, pDataSet: QString): bool
Descripción: Establece la función objetivo de la instancia de un problema dado su nombre y el problema.

Tabla 17 Descripción de la clase IProblemManager

Descripción de la clase IProblemManager
Nombre: IProblemManager
Tipo de clase: Interfaz
Responsabilidades
Nombre: addDataSet(pProblemName: QString, pDataSet: DataSet): bool
Descripción: Adiciona una nueva fuente de datos.
Nombre: removeDataSet(pProblemName: QString, pDataSet: QString): bool
Descripción: Elimina una fuente de datos.
Nombre: addFromFileParseResult(pDescription: FileParseResult, pRejectAllIfDuplicates: bool): QList<QString>
Descripción: Añade una fuente de datos desde una archivo.

Tabla 18 Descripción de la clase IRegionMgr

Descripción de la clase IRegionMgr
Nombre: IRegionMgr
Tipo de clase: Interfaz
Responsabilidades

Nombre: addRegion(pRegion: IRegion*)
Descripción: Añade una nueva región.

Tabla 19 Descripción de la clase IRegion

Descripción de la clase IRegion	
Nombre: IRegion	
Tipo de clase: Interfaz	
Responsabilidades	
Nombre: addChild(pChild: QWidget*, pTitle: QString)	
Descripción: Añade un widget a una región.	
Nombre: getName(): QString	
Descripción: Obtiene el nombre de una región.	

Tabla 20 Descripción de la clase QServiceLocator

Descripción de la clase QServiceLocator	
Nombre: QServiceLocator	
Tipo de clase: Controladora	
Atributo:	**Tipo:**
m_registrations	private
DEFAULT_NAME	private
container	private
Responsabilidades	
Nombre: globalInstance(): QServiceLocator	
Descripción: Devuelve la instancia del localizador de servicios.	
Nombre: registerInstance(instance: QObject*, name: QString)	
Descripción: Registra una instancia de un objeto en el localizador de servicios.	
Nombre: resolve(name: QString): T*	
Descripción: Devuelve un objeto almacenado en el localizador de servicios dado su nombre.	
Nombre: registerType(name: QString, manager: LifetimeManager*)	
Descripción: Registra un nuevo tipo en el localizador de servicios.	
Nombre: registerType(manager: LifetimeManager*)	
Descripción: Registra un nuevo tipo en el localizador de servicios.	

2.5.4 DIAGRAMA DE DESPLIEGUE

Un diagrama de despliegue visualiza los elementos de hardware y software utilizados en la implementación del sistema y además las relaciones entre ellos. Es utilizado para modelar la topología de procesadores y dispositivos empleados en el software (23). La Figura 8 muestra el diagrama de despliegue de la solución propuesta.

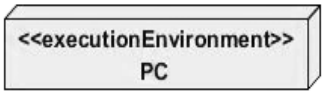

Figura 8 Diagrama de Despliegue

2.6 CONCLUSIONES PARCIALES

Luego de realizarse el análisis y diseño del sistema se obtuvieron las siguientes conclusiones:

- Una vez descrito el proceso a automatizar se identificaron los requisitos, tanto funcionales como no funcionales, para proceder a la elaboración de las HU.
- Se realizó la descripción de la arquitectura utilizada y se fundamentaron y ejemplificaron los patrones de diseño utilizados.
- Se presentaron los diagramas de clases y fueron brevemente descritas las clases principales.

3 IMPLEMENTACIÓN Y EVALUACIÓN

En este capítulo se elaborará el Modelo de implementación donde se obtendrá el Diagrama de componentes y se describirán algunos de los estándares de codificación utilizados. Asimismo, se analizarán algunas de las técnicas que se emplean comúnmente para la validación de arquitecturas y se seleccionará una para evaluar la solución propuesta. De igual forma se validará el diseño de la arquitectura a través de métricas de software.

3.1 MODELO DE IMPLEMENTACIÓN

El modelo de implementación es comprendido por un conjunto de componentes y subsistemas que constituyen la composición física de la implementación del sistema. Entre los componentes se pueden encontrar datos, archivos, ejecutables, código fuente y los directorios. Fundamentalmente, se describe la relación que existe desde los paquetes y clases del modelo de diseño a subsistemas y componentes físicos.

3.1.1 DIAGRAMAS DE COMPONENTES

En los diagramas de componentes se muestran los elementos de diseño de un sistema de software. Un diagrama de componentes permite visualizar con más facilidad la estructura general del sistema y el comportamiento del servicio que estos componentes proporcionan y utilizan a través de las interfaces (23). En la Figura 9 se muestra el diagrama de componentes del paquete Extensibility.

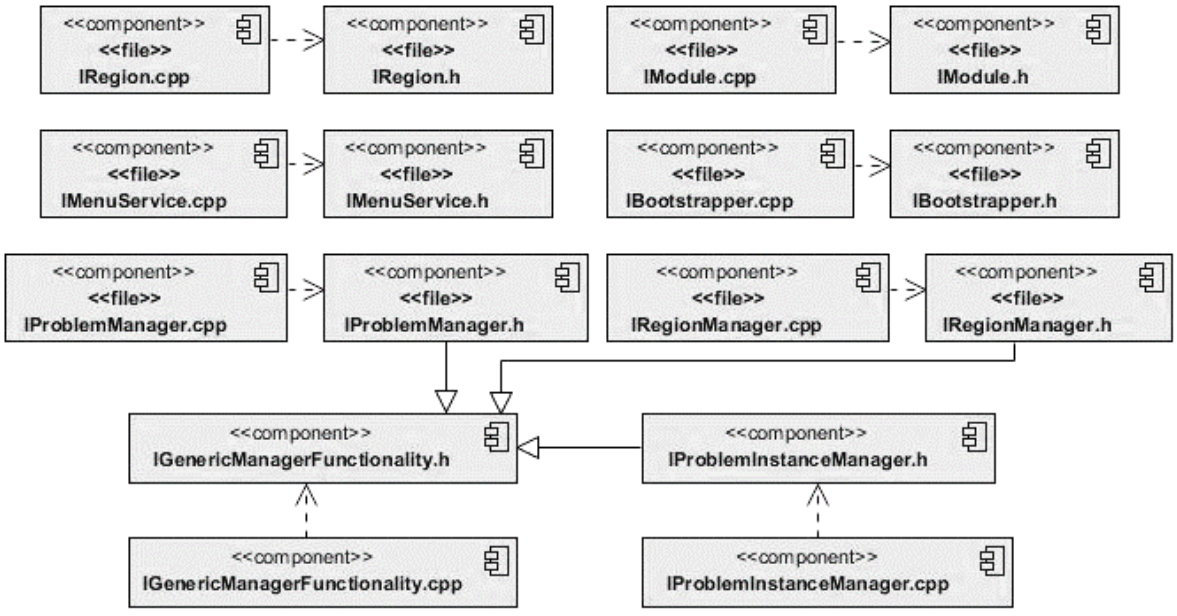

Figura 9 Diagrama de Componentes del paquete Extensibility

3.1.2 CÓDIGO FUENTE

Se define como código fuente al texto desarrollado en un lenguaje de programación y que debe ser compilado o interpretado para poder ejecutarse en un ordenador.

El código fuente de un programa informático lo constituye un conjunto de líneas de texto que describen su funcionamiento. Con el objetivo de alcanzar un mayor entendimiento del código fuente se definió como estándar de codificación el utilizado por el cetro VERTEX perteneciente a la UCI. Entre las reglas definidas se encuentran:

- Utilizar la variante *lowerCamelCase* para los identificadores del tipo variables y métodos. Las palabras comenzarán con minúsculas y si los identificadores están compuestos por varias palabras, las siguientes empezarán con mayúscula.
- Utilizar la variante *UpperCamelCase* para los identificadores del tipo clase, enumeradores e interfaces. Todas las palabras que componen a dichos identificadores empezarán con mayúscula (24).

Además, se definieron dos reglas necesarias con el fin de homogenizar el código fuente, y que se muestran a continuación:

- Añadir al inicio del nombre de las clases que poseen todos sus métodos virtuales puros la letra I mayúscula. (I de interface). Ejemplo: ***IProblemManager***.
- Añadir al inicio de los nombres de los parámetros de los métodos letra p minúscula y luego el nombre del parámetro con letra inicial mayúscula. Ejemplo: *pName* (24).

A continuación, se muestran fragmentos de código fuente pertenecientes a implementaciones importantes de la clase ***IProblemInstanceManager***:

```
#ifndef IPROBLEMINSTANCEMANAGER_H
#define IPROBLEMINSTANCEMANAGER_H
#include <QObject>
#include "problem-instance.h"
#include "problem.h"
#include "i-generic-manager-functionality.h"
namespace AppName
{
    namespace Extensibility {
        class EXTENSIBILITYSHARED_EXPORT IProblemInstanceManager: virtual public IGenericM
        {

        public:
            IProblemInstanceManager();
            virtual bool setModel(const QString& pInstanceName, IModel* pModel)=0;
            virtual bool setObjectiveFunction(const QString& pInstanceName, IObjectiveFunct
            virtual bool setPreprocessingAlgorithm(const QString& pInstanceName, IPreproce
            virtual bool setGlobalAlgorithm(const QString& pInstanceName, IGlobalAlgorithm
            virtual bool setLocalAlgorithm(const QString& pInstanceName, ILocalAlgorithm* p
            virtual bool setDataSet(const QString& pInstanceName, DataSet& pDataSet)=0;
            virtual bool setDataSet(const QString &pInstanceName, const Problem& pProblem,
                                    const QString& pDataSet)=0;

        protected:
            virtual ~IProblemInstanceManager();
        };
    }
}


#endif // IPROBLEMINSTANCEMANAGER_H
```

Figura 10 Fragmentos de código fuente de la clase IProblemInstanceManager

3.2 EVALUACIÓN DE LA ARQUITECTURA DE SOFTWARE

La arquitectura de software posee gran impacto sobre la calidad de un sistema, por lo que se hace necesario evaluarla para determinar su potencial y alcanzar los atributos de calidad requeridos. Es importante destacar que la evaluación no define si una arquitectura es buena o no, simplemente expresa donde se encuentran los riesgos y fortalezas de la misma.

3.2.1 TÉCNICAS DE EVALUACIÓN DE ARQUITECTURAS

Las técnicas de evaluación de arquitecturas tienen como principal objetivo, crear especificaciones y predicciones respecto a una propuesta arquitectónica, para saber si satisface las cualidades que el software debe cumplir. Estas técnicas pueden clasificarse en dos vertientes fundamentales: cuantitativas y cualitativas (25).

Por lo regular, las técnicas de evaluación cualitativas son utilizadas cuando la arquitectura está en construcción, mientras que las técnicas de evaluación cuantitativas, se usan cuando la arquitectura ya ha sido implantada.

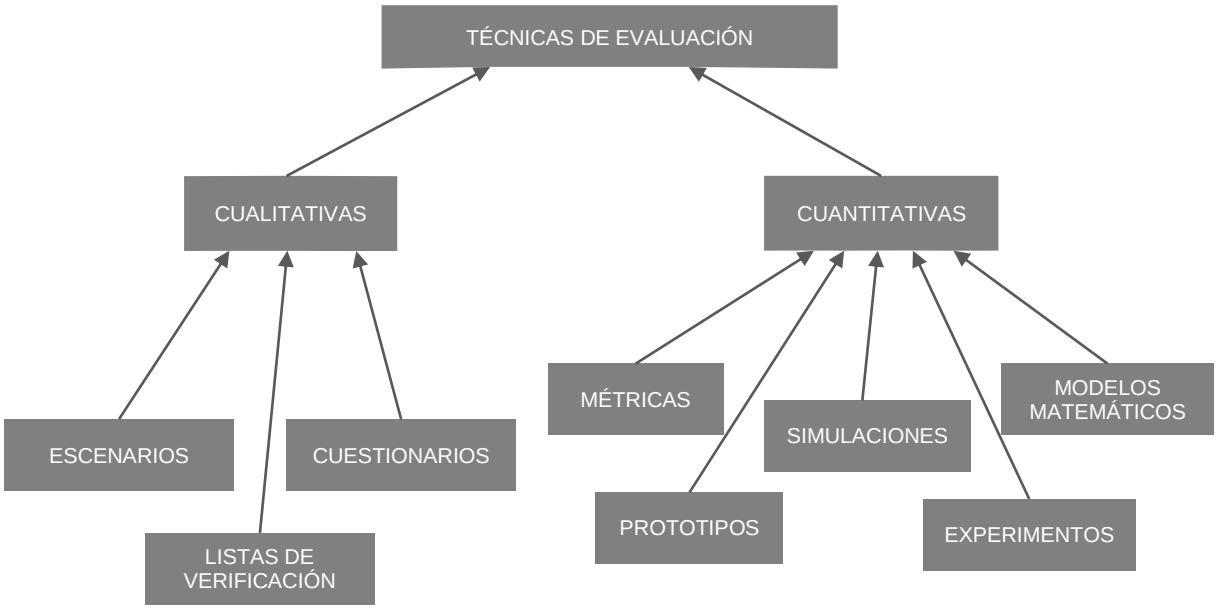

Figura 11 Técnicas de evaluación de arquitecturas

A continuación, se hace un análisis de las técnicas más propicias a ser empleadas para la evaluación de la arquitectura propuesta.

Evaluación basada en la experiencia

En muchas ocasiones los arquitectos e ingenieros de software otorgan valiosas ideas que resultan de utilidad para la evasión de decisiones erradas de diseño, estas ideas se basan en factores subjetivos (como la experiencia) y están respaldadas por una línea lógica de razonamiento, que se puede adquirir por el trabajo realizado en proyectos similares. Por tanto, el principal instrumento de evaluación con que cuenta esta técnica es precisamente la intuición y experiencia con que cuentan los arquitectos y demás miembros del equipo de desarrollo. Existen dos tipos de evaluación basada en experiencia: la evaluación informal, que es realizada por los arquitectos de software durante el proceso de diseño, y la realizada por equipos externos de evaluación de arquitecturas (25).

Evaluación basada en simulación

La evaluación basada en simulación utiliza una implementación de alto nivel de la arquitectura de software. El enfoque básico consiste en la implementación de componentes de la arquitectura, a cierto nivel de abstracción del contexto del sistema donde se supone va a ejecutarse. Su finalidad es evaluar el comportamiento de la arquitectura bajo diversas circunstancias, una vez disponibles estas implementaciones, pueden usarse los perfiles respectivos para evaluar los atributos de calidad.

La exactitud de los resultados de la evaluación depende, a su vez, de la exactitud del perfil utilizado para evaluar el atributo de calidad y de la precisión con la que el contexto del sistema simula las condiciones del mundo real. Los instrumentos asociados a esta técnica son los lenguajes de descripción

arquitectónica (ADL[4] por sus siglas en inglés) y los modelos de colas, siendo esto una de sus desventajas, pues el evaluador debe tener experiencia en el uso de los ADL o los modelos de cola para realizar una correcta evaluación (25).

Evaluación basada en prototipo

Esta técnica consiste en implementar una parte de la arquitectura de software y ejecutarla en el contexto del sistema. Para su uso se necesita mayor información sobre el desarrollo y disponibilidad de hardware, y los elementos que constituyen el contexto del sistema de software. Mediante esta técnica se obtiene un resultado de evaluación con mayor exactitud (25).

Entre los aspectos favorables del uso de esta técnica se destacan la confiabilidad, pues se puede ver de manera directa que tanto se afecta o no, un atributo de calidad en el sistema que se desarrolla, los usuarios finales pueden observar el resultado que se está obteniendo y evaluar junto al equipo de desarrollo si satisface o no sus expectativas. De acuerdo al nivel de fidelidad con que se desarrolle el prototipo del producto final, puede suponer desventajas el empleo de esta técnica, pues el tiempo de desarrollo del prototipo puede ser largo (lo que convierte a la técnica en costosa) y demorar en poder realizar la evaluación.

3.3 EVALUACIÓN DE LA ARQUITECTURA PROPUESTA

Después del análisis realizado sobre las diferentes técnicas de evaluación de arquitecturas, se selecciona, para evaluar la arquitectura propuesta, la técnica de evaluación basada en prototipo.

3.3.1 TÉCNICA DE EVALUACIÓN: PROTOTIPO

Para la validación de la arquitectura propuesta, se confeccionó el prototipo funcional de una aplicación. Además, se desarrollaron dos *plugins* que una vez cargados en la aplicación, añaden un nuevo menú a través de la interfaz ***IMenuService*** que permite acceder a sus funcionalidades. El primero de estos *plugins* añade dos funcionalidades al menú: la primera muestra un cuadro de dialogo en la interfaz principal de la aplicación y la segunda añade a la región principal de la aplicación un ***QLineEdit*** con un texto personalizable. La Figura 12 muestra la interfaz principal de la aplicación y las entradas en el menú de los dos *plugins* desarrollados.

[4] ADL: Architecture Description Languages (Lenguaje de Descripción de Arquitectura).

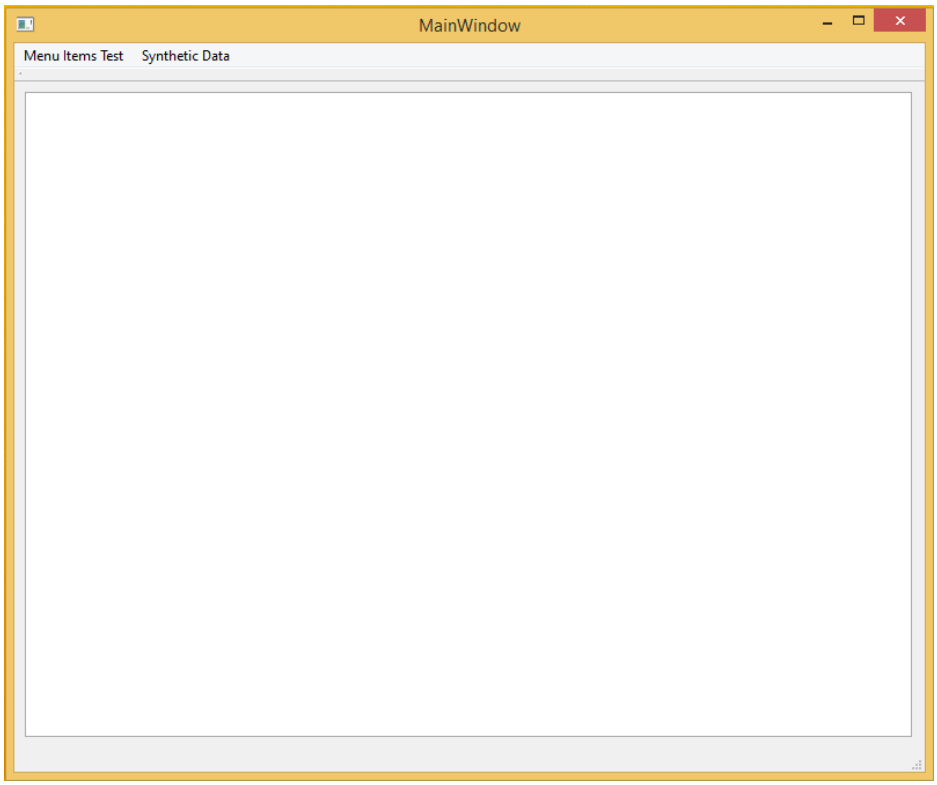

Figura 12 Interfaz principal del prototipo funcional

El segundo *plugin* permite generar un juego de datos sintéticos solicitando a través de un cuadro de dialogo los parámetros para la generación de los mismos empleando el modelo

$$m(a, b, x) = ax^2 - 20\cos(bx),$$

la Figura 13 muestra el cuadro de dialogo generado por este *plugin*.

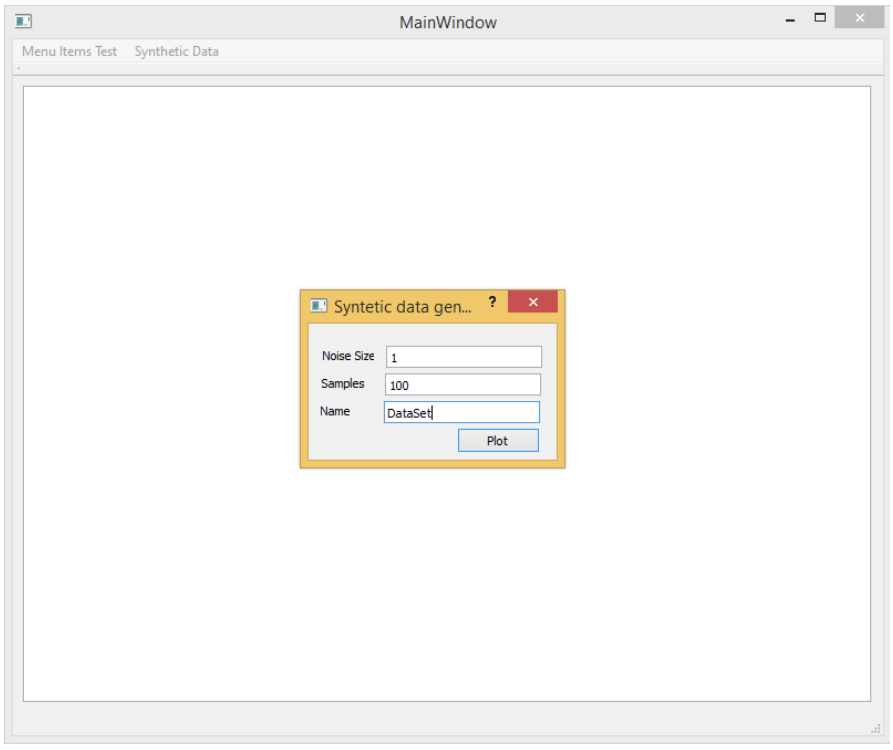

Figura 13 Interfaz principal del prototipo funcional

Una vez introducidos los parámetros, se generan los datos y el *plugin* añade un componente de graficación en la interfaz principal de la aplicación, donde se muestran los datos calculados. La Figura 14 visualiza la gráfica resultante.

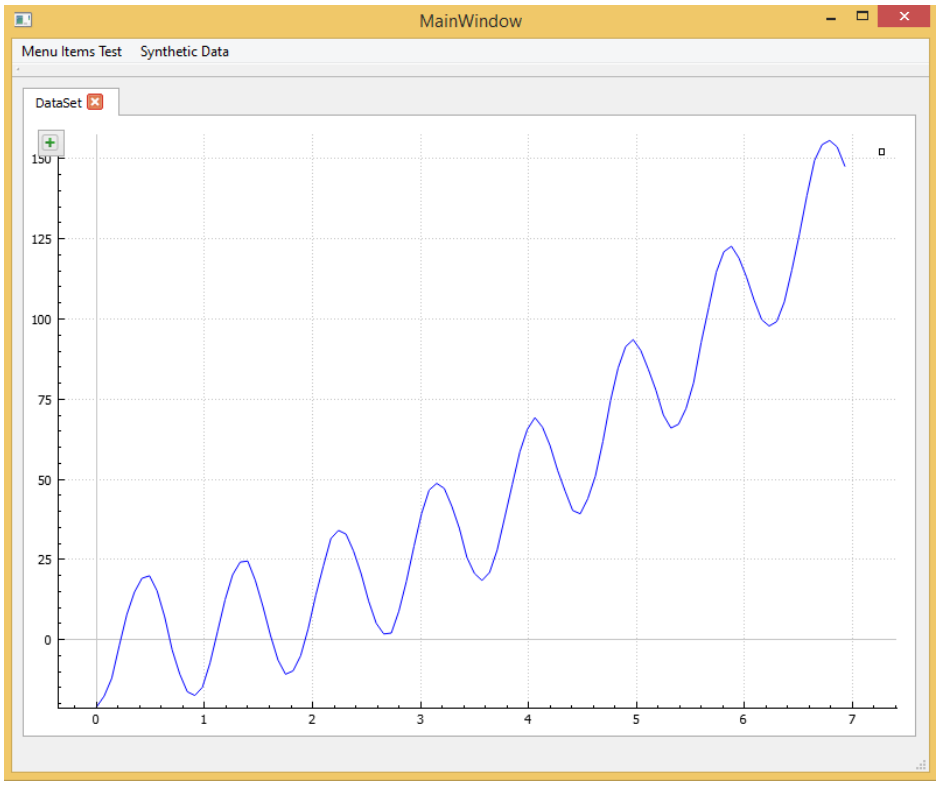

Figura 14 Interfaz principal del prototipo funcional

La creación de este prototipo permitió evaluar, de forma práctica, los atributos de calidad de interés para los usuarios, tal es el caso de la compatibilidad entre los componentes que fueron desarrollados de forma separada al integrarse a una misma aplicación. De esta forma la arquitectura propuesta satisface los principales atributos de calidad definidos para la investigación.

3.4 EVALUACIÓN DEL DISEÑO APLICANDO MÉTRICAS DE SOFTWARE

La evaluación del software a partir de métricas es una medida cuantitativa que proporciona una visión profunda de la eficacia del proceso del software. Se reúnen datos básicos y son analizados, comparados con promedios, y evaluados para determinar las mejoras en la calidad y productividad. Las métricas son también utilizadas para señalar áreas con problemas de manera que se puedan desarrollar los remedios y mejorar el proceso del software (15).

Las métricas empleadas están diseñadas para evaluar los siguientes atributos de calidad:

- **Responsabilidad:** consiste en la responsabilidad asignada a una clase en un marco de modelado de un dominio o concepto, de la problemática propuesta.

- **Complejidad de implementación**: consiste en el grado de dificultad que tiene implementar un diseño de clases determinado.
- **Reutilización:** consiste en el grado de reutilización presente en una clase o estructura de clase, dentro de un diseño de software.
- **Acoplamiento:** consiste en el grado de dependencia o interconexión de una clase o estructura de clase, con otras, está muy ligada a la característica de Reutilización.
- **Complejidad del mantenimiento:** consiste en el grado de esfuerzo necesario a realizar para desarrollar un arreglo, una mejora o una rectificación de algún error de un diseño de software. Puede influir indirectamente, pero fuertemente en los costes y la planificación del proyecto.
- **Cantidad de pruebas:** consiste en el número o el grado de esfuerzo para realizar las pruebas de calidad (unidad) del producto (componente, módulo, clase, conjunto de clases, entre otras) diseñado (15).

Las métricas seleccionadas para evaluar la calidad del diseño de la solución propuesta fueron Tamaño Operacional de Clases (TOC) y Relaciones entre Clases (RC).

Tamaño Operacional de Clases (TOC): Está dado por el número de métodos u operaciones (de instancia privada y heredada) que están encapsulados dentro o por una clase. Evalúa los siguientes atributos de calidad:

Tabla 21 Atributos de calidad de evalúa TOC

Atributo de Calidad	Modo en que lo afecta
Responsabilidad	Aumento del TOC provoca aumento de la responsabilidad asignada a la clase.
Complejidad de implementación	Aumento del TOC provoca aumento de la complejidad de implementación de la clase.
Reutilización	Aumento del TOC provoca disminución del grado de reutilización de la clase.

Para la evaluación de estos atributos de calidad se definieron criterios y categorías de evaluación. Para ver estos criterios consultar Anexo Criterios de evaluación de la métrica TOC.

Relaciones entre Clases (RC): Está dado por el número de relaciones de uso de una clase con otra. Evalúa los siguientes atributos de calidad:

Tabla 22 Atributos de calidad que evalúa RC

Atributo de Calidad	Modo en que lo afecta
Acoplamiento	Aumento del RC provoca aumento del Acoplamiento de la clase.
Complejidad de mantenimiento	Aumento del RC provoca aumento de la complejidad del mantenimiento de la clase.
Reutilización	Aumento del RC provoca disminución en el grado de reutilización de la clase.
Cantidad de pruebas	Aumento del RC provoca aumento de la Cantidad de pruebas de unidad necesarias para probar una clase.

Para la evaluación de dichos atributos de calidad, se definieron los criterios y categorías de evaluación. Para ver estos criterios consultar Anexo Criterios de evaluación de la métrica RC.

3.4.1 RESULTADOS OBTENIDOS EN LA APLICACIÓN DE LA MÉTRICA TOC

Ver Instrumento de evaluación de la métrica TOC en el Anexo Instrumento de evaluación de la métrica TOC.

La Figura 15 muestra la representación de los resultados obtenidos agrupados en los intervalos definidos. El gráfico refleja que la mayoría de las clases tienen de 1 a 20 procedimientos. Esto demuestra que el funcionamiento general de la solución está distribuida equitativamente entre las clases.

Los resultados obtenidos luego de aplicar las métricas TOC arrojaron que el diseño propuesto tiene una calidad aceptable, teniendo en cuenta que el 50 % de las clases poseen una cantidad de procedimientos menor o igual al promedio general de 7,333, esto conlleva a que las evaluaciones sean positivas en los atributos de calidad involucrados.

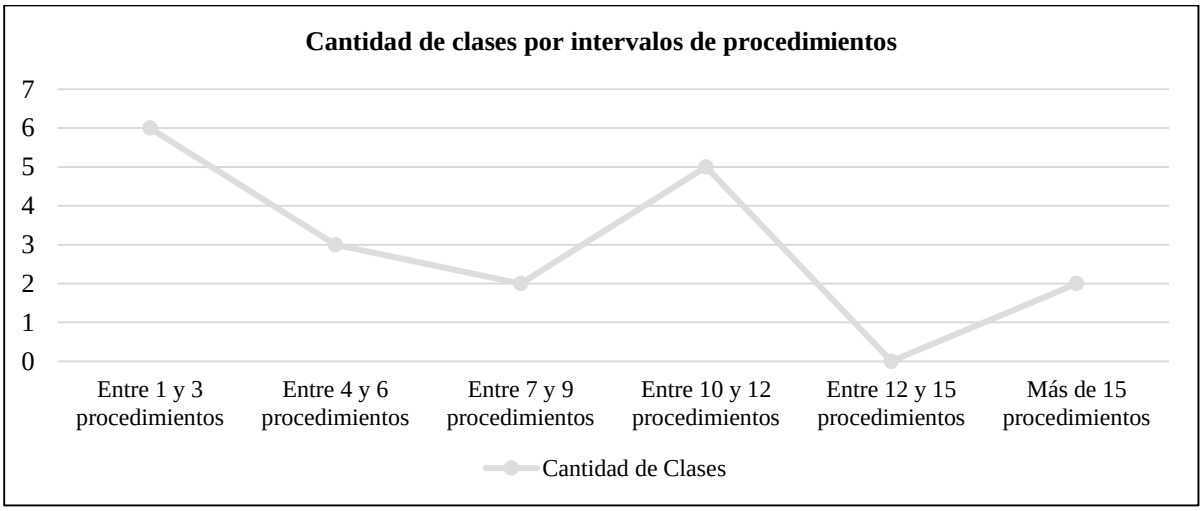

Figura 15 Representación de las clases según la cantidad de operaciones

La Figura 16 muestra la representación de la incidencia de los resultados de la evaluación de la métrica TOC en el atributo Responsabilidad. Quedó demostrado que el 50% de las clases tienen una baja responsabilidad ya que este atributo se distribuyó equitativamente entre todas las clases del sistema. Esta característica permite que en caso de fallos como la responsabilidad está distribuida de forma equilibrada ningún componente sea demasiado crítico como para dejar fuera de servicio el sistema.

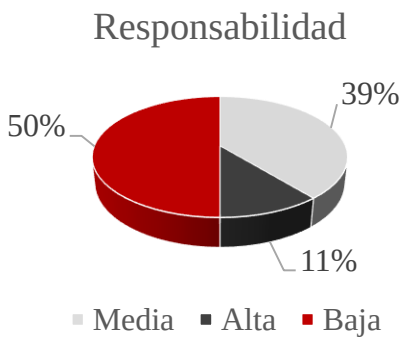

Figura 16 Resultados de la valuación de la métrica TOC para el atributo de calidad Responsabilidad

La Figura 17 muestra la representación de la incidencia de los resultados de la evaluación de la métrica TOC en el atributo Complejidad de implementación. El gráfico muestra que el 50% de las clases tienen una baja complejidad, este atributo está distribuido equitativamente. Esta característica permite mejorar el mantenimiento y soporte de las clases.

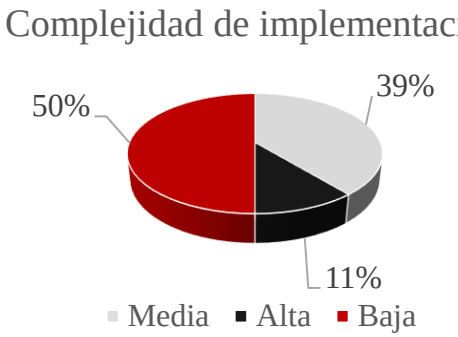

Figura 17 Resultados de la evaluación de la métrica TOC para el atributo de calidad Complejidad de implementación

La Figura 18 muestra la representación de la incidencia de los resultados de la evaluación de la métrica TOC en el atributo Reutilización. Queda demostrado que el diseño de la solución es eficiente ya que el 50 % de las clases tienen un alto grado de reutilización.

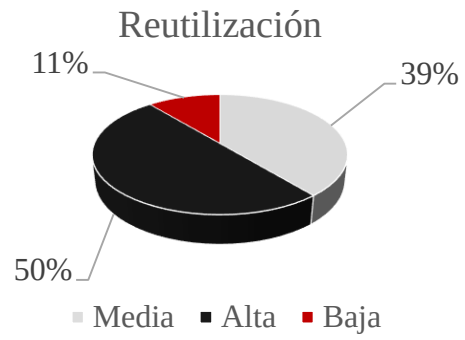

Figura 18 Resultados de la evaluación de la métrica TOC para el atributo de calidad Reutilización

Haciendo un análisis de los resultados obtenidos en la evaluación del instrumento de medición de la métrica TOC, se puede concluir que el diseño de la herramienta para la resolución de problemas de regresión tiene una buena calidad.

3.4.2 RESULTADOS OBTENIDOS EN LA APLICACIÓN DE LA MÉTRICA RC

Ver Instrumento de evaluación de la métrica RC en el Anexo Instrumento de evaluación de la métrica RC.

La Figura 19 muestra la representación de los resultados obtenidos en el instrumento agrupados en los intervalos definidos. La Figura 20 muestra la representación en % de los resultados obtenidos en el instrumento agrupados en los intervalos definidos.

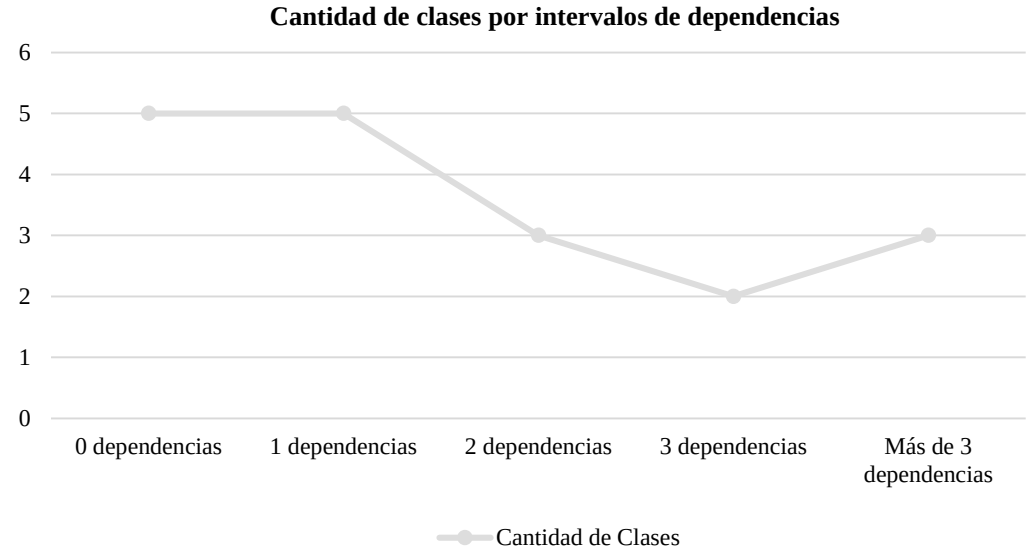

Figura 19 Representación de las clases según la cantidad de relaciones de uso

Los resultados obtenidos luego de aplicar las métricas RC arrojan que el diseño propuesto tiene una calidad aceptable, teniendo en cuenta que el 53 % de las clases poseen una cantidad de relaciones de uso menor o igual al promedio general de 2,1, esto conlleva a que las evaluaciones sean positivas en los atributos de calidad involucrados.

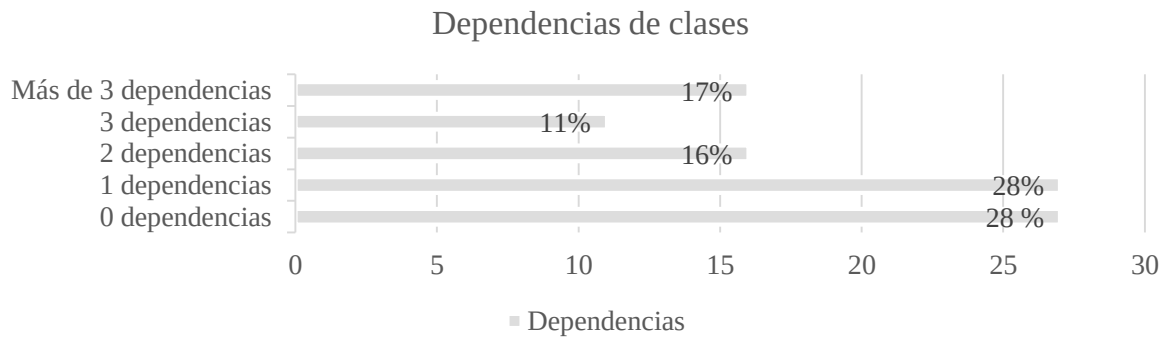

Figura 20 Representación en % de los resultados obtenidos en el instrumento agrupados en los intervalos definidos

La Figura 21 muestra la representación de la incidencia de los resultados de la evaluación de la métrica RC en el atributo Acoplamiento. Se evidencia un diseño eficiente al quedar reflejado que un 55 % de las clases cuentan con un bajo acoplamiento.

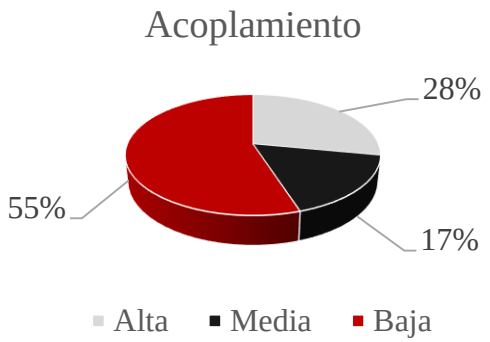

Figura 21 Resultados de la evaluación de la métrica RC para el atributo de calidad Acoplamiento

La Figura 22 muestra la representación de la incidencia de los resultados de la evaluación de la métrica RC en el atributo Complejidad de Mantenimiento. Queda demostrada la eficiencia de la arquitectura del sistema al evidenciarse que un 72 % de las clases posee una baja complejidad de mantenimiento.

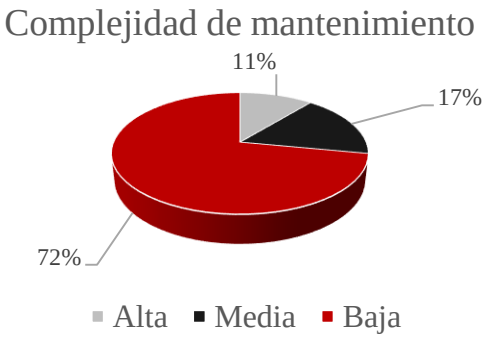

Figura 22 Resultados de la evaluación de la métrica RC para el atributo de calidad Complejidad de mantenimiento

La Figura 23 muestra la representación de la incidencia de los resultados de la evaluación de la métrica RC en el atributo Reutilización. Esto evidencia que el 72% de las clases poseen una alta reutilización lo que es un factor fundamental que debe ser tenido en cuenta en el desarrollo de software.

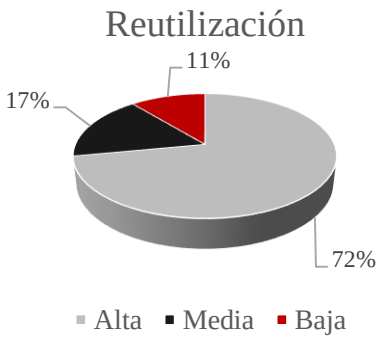

Figura 23 Resultados de la evaluación de la métrica RC para el atributo de calidad Reutilización

La Figura 24 muestra la representación de la incidencia de los resultados de la evaluación de la métrica RC en el atributo Cantidad de pruebas. Esto evidencia que un 72% de las clases poseen baja cantidad de pruebas, lo que representa valores favorables para el diseño realizado.

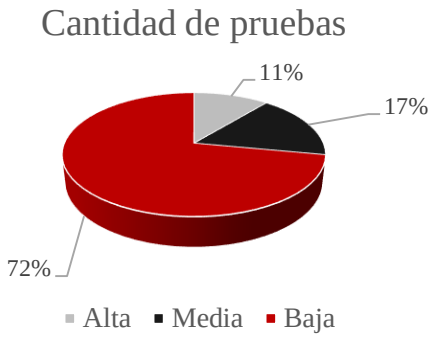

Figura 24 Resultados de la evaluación de la métrica RC para el atributo de calidad Cantidad de pruebas

3.4.3 MATRIZ DE INFERENCIA DE INDICADORES DE CALIDAD

La matriz inferencia de indicadores de calidad, también llamada matriz de cubrimiento, es una representación estructurada de los atributos de calidad y métricas utilizadas para evaluar la calidad del diseño propuesto. Dicha matriz permite conocer si los resultados obtenidos de las relaciones atributo/métrica son positivo o no, llevando estos resultados a una escalabilidad numérica.

Si los resultados son positivos se le asigna el valor de 1, si son negativos toma valor 0 y si no existe relación es considerada como nula y es representada con un guión simple (-). Luego se puede obtener un resultado general para cada atributo promediando todas sus relaciones no nulas. A continuación, se muestran los resultados obtenidos.

Tabla 23 Rango de valores para la evaluación de la relación atributo/métrica

Atributo/Métrica	TOC	RC	Promedio
Responsabilidad	1	(-)	1
Complejidad de Implementación	1	(-)	1
Reutilización	1	1	1

Acoplamiento	(-)	1	1
Complejidad de mantenimiento	(-)	1	1
Cantidad de pruebas	(-)	1	1

Tabla 24 Rango de valores para la evaluación de la relación atributo/métrica

Categoría	Rango de Valores
Malo	<= 0.4
Regular	>0.4 y <=0.7
Bueno	>0.7

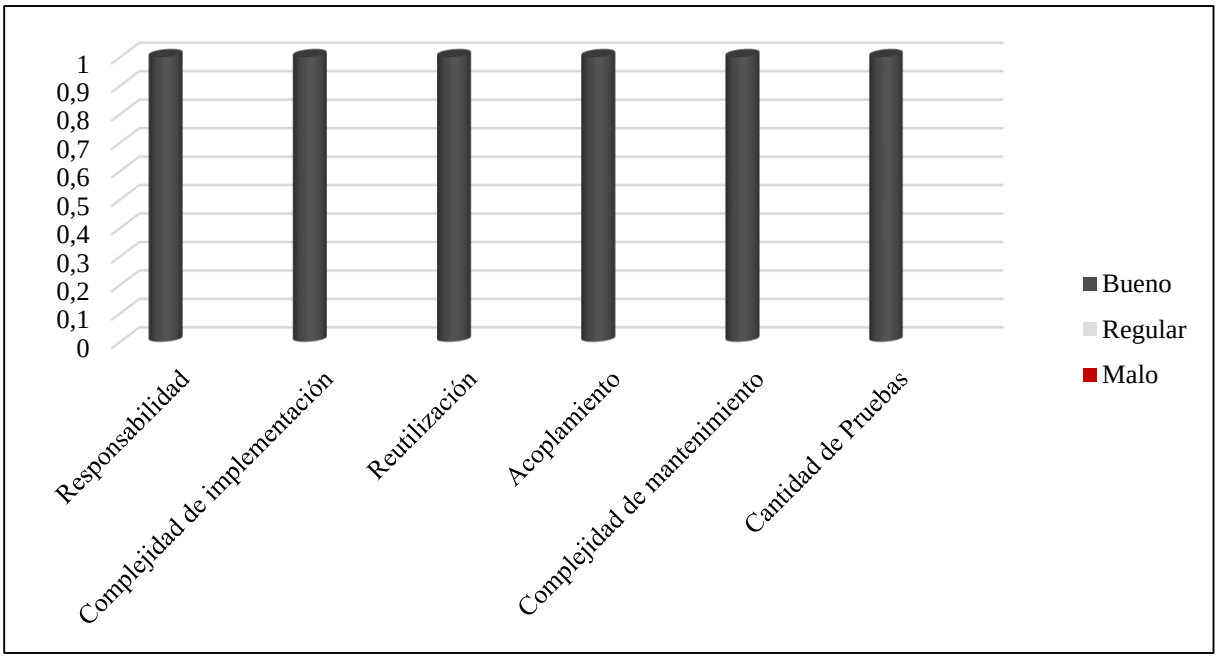

Figura 25 Resultados obtenidos de la evaluación de los atributos de calidad

La Figura 25 muestra la gráfica de los resultados obtenidos de los atributos de calidad evaluados en las métricas aplicadas anteriormente, donde todos los atributos de calidad mantienen un buen comportamiento.

CONCLUSIONES

- La elaboración del marco teórico a partir del estudio de diversas suites de software comerciales para la regresión no lineal, permitió comprobar de que los fabricantes imponen restricciones que en muchos escenarios de la vida practica provocan que su utilización no sea factible.
- El análisis y diseño de la solución posibilitó definir una combinación de estilos arquitectónicos que responden a los requisitos funcionales y a los atributos de calidad definidos, permitiendo resolver los problemas de flexibilidad, extensibilidad y facilidad de mantenimiento de la solución existente.
- La validación de la arquitectura a partir de la técnica “Prototipo” y de las métricas de software TOC y RC, permitió comprobar que el diseño arquitectónico cumple con los requisitos y atributos de calidad definidos.

RECOMENDACIONES

- Implementar uno o varios mecanismos para comprobar la autenticidad de los *plugins*.

BIBLIOGRAFÍA

1. NOCEDAL, Jorge y WRIGHT, Stephen J. *Numerical optimization*. 2. ed. New York, NY: Springer, 2006. Springer series in operation research and financial engineering. ISBN 978-0-387-30303-1.

2. ALFONSO MONTEAGUDO, Yasmany. Diseño de un marco de trabajo extensible para la resolución de problemas de regresión no lineal. 2016. P. 7.

3. NOCEDAL, Jorge y WRIGHT, Stephen J. *Numerical optimization* [en línea]. New York: Springer, 1999. ISBN 978-0-387-22742-9. Disponible desde: http://site.ebrary.com/id/10003036This work covers numerical methods for finite-dimensional optimisation problems involving fairly smooth functions. It concentrates on methods for unconstrained optimisation, with attention given at the end to problems with constraints.OCLC: 54849297

4. DASTAN, Aysegul, HORNE, Roland N., GERRITSEN, Margot G. y MUKERJI, Tapan. *A new look at nonlinear regression in well testing* [en línea]. Stanford University, 2010. Disponible desde: https://books.google.com/books?hl=en&lr=&id=zuVYEMu71LAC&oi=fnd&pg=PR5&dq=%22the+pressure+derivative.+We+developed+four+different+strategies+to+form+a%22+%22and+provide+faster+convergence+for+dual-porosity+reservoirs.+We%22+%22giving+much+narrower+confidence%22+%22methods+were+applied+to+20+real+well+test+data+sets+from+a+selection%22+&ots=J4iwW3Izhq&sig=dMRf4ZEmsGKHxqD7fiGqSeAOGqE

5. NLREG -- Nonlinear Regression Analysis Program. [en línea]. Disponible desde: http://www.nlreg.com/

6. Prism - graphpad.com. [en línea]. Disponible desde: http://www.graphpad.com/scientific-software/prism/#whychoose

7. Origin: Data Analysis y Graphing Software. [en línea]. Disponible desde: http://www.originlab.com/index.aspx?go=Products/Origin

8. *Handbook of Statistical Methods*. 2013

9. FIGUEROA, Roberth G., SOLÍS, Camilo J. y CABRERA, Armando A. Metodologías tradicionales vs. Metodologías ágiles. [en línea]. 2008. Disponible desde: http://infotics.org/ecommerce/ingenieria%20web/articulo-metodologia-de-sw-formato.doc

10. RODRÍGUEZ SÁNCHEZ, Tamara. *Metodología de desarrollo para la Actividad productiva de la UCI*. 2015.

11. JORDAN, David. Implementation benefits of C++ language mechanisms. *Communications of the ACM*. 1990. Vol. 33, no. 9, p. 61–64.

12. Qt - Product | Qt for Application Development. *Qt* [en línea]. Disponible desde: https://www.qt.io/qt-for-application-development/

13. Qt Creator/es - Qt Wiki. [en línea]. Disponible desde: http://wiki.qt.io/Qt_Creator/es

14. HERNÁNDEZ, Lionel R. Baquero. Extensión de la herramienta Visual Paradigm for UML para la evaluación y corrección de Diagramas de Casos de Uso. *Serie Científica-Universidad de las Ciencias*

Informáticas [en línea]. 2016. Vol. 9, no. 4. Disponible desde: http://publicaciones.uci.cu/index.php/SC/article/view/1758

15. PRESSMAN, Roger S, CAMPOS OLGUÍN, Víctor y ENRÍQUEZ BRITO, Javer. *Ingeniería del software: un enfoque práctico*. México: McGraw-Hill, 2010. ISBN 978-607-15-0314-5. OCLC: 642704490

16. SOMMERVILLE, I. *Ingeniería de Software*. 7ma. España: Educación, 2005. ISBN 84-7829-074-5.

17. BECK, Kent. *Implementation patterns*. 3. printing. Upper Saddle River, NJ: Addison-Wesley, 2008. The Addison-Wesley signature series. ISBN 978-0-321-41309-3. OCLC: 855869861

18. TORRE, César de la. *Guía de arquitectura n-capas orientada al dominio con .NET 4.0 (beta)*. Vigo: Krasis Consulting, 2010. ISBN 978-84-936696-3-8. OCLC: 776289602

19. ESPOSITO, Dino y SALTARELLO, Andrea. *Microsoft .NET architecting applications for the enterprise* [en línea]. Redmond, WA: Microsoft Press, 2014. ISBN 978-0-13-398641-9. Disponible desde: http://proquest.safaribooksonline.com/?fpi=9780133986419OCLC: 892345233

20. BACHMANN, Felix. *Technical Concepts of Component-Based Software Engineering*. 2da. 2000.

21. SHALLOWAY, Alan y TROTT, James R. *Design patterns explained: a new perspective on object-oriented design*. Pearson Education, 2004.

22. The Service Locator Pattern. [en línea]. Disponible desde: https://msdn.microsoft.com/en-us/library/ff648968.aspx

23. SCHMULLER, Joseph. *Aprendiendo UML en 24 horas*. Mexico: Pearson Educación, 2000. ISBN 978-968-444-463-8. OCLC: 48646875

24. LOMBERA RODRÍGUEZ, Hassan. *ESTÁNDARES DE CODIFICACIÓN PARA EL LENGUAJE C++ UTILIZADO EN EL CENTRO DE DISEÑO Y SIMULACIÓN DE ESTRUCTURAS MECÁNICAS*. 2011.

25. BOCH, Jan. *Design & Use of Software Architectures. Adopting and evolving a product-line approach*. Addison-Wesley, 2000. ISBN 978-0-201-67494-1.

26. ABELLÓ UGALDE, Isidro A. *Métodos Estructurados Secantes para Mínimos Cuadrados*. 2007.

27. CSENDES, Tibor. *Nonlinear Parameter Estimation by Global Optimization – Efficiency and Reliability*. 2005.

28. MITCHENER, Garrett. *Design P atterns by Example*. 1997.

ANEXOS

Anexo Descripción de requisitos no funcionales

Atributo de Calidad	Fiabilidad.
Sub-atributos/Sub-característica	Madurez.
Objetivo	Capacidad del producto para satisfacer las necesidades de fiabilidad en condiciones normales.
Origen	Desarrollador.
Artefacto	El sistema.
Entorno	El sistema desplegado y funcionando en periodos de tiempos determinados.
Estímulo	**Respuesta: Flujo de eventos (Escenarios)**
<<1>>. <<a>> <El sistema desplegado en los escenarios especificados por el cliente>	
El sistema funciona correctamente bajo condiciones y periodos de tiempos determinados por el cliente.	NA
Medida de respuesta	
Desplegar el sistema.	

Atributo de Calidad	Mantenibilidad.
Sub-atributos/Sub-característica	Analizabilidad.
Objetivo	Facilidad con la que se puede evaluar el impacto de un determinando cambio sobre el resto del software, diagnosticar las deficiencias o causas de fallos en el software, o identificar las partes a modificar.
Origen	Desarrollador.
Artefacto	El código fuente.
Entorno	El ambiente de desarrollo del sistema.
Estímulo	**Respuesta: Flujo de eventos (Escenarios)**
<<1>>. <<a>>< Impacto de un determinado cambio sobre el resto del sistema>	

Se evalúa el impacto de un cambio en el sistema.	1. El arquitecto de software identifica los paquetes y clases implicados en el cambio, así como las funciones que se reutilizarán o se crearán para introducir el cambio. 2. Se evalúa el impacto partiendo de los resultados que arrojó el análisis anterior con respecto a la arquitectura del sistema.
<<1>>. <<b>>< Diagnosticar las deficiencias o causas de fallos en el sistema>	**Respuesta: Flujo de eventos (Escenarios)**
Se diagnostica las deficiencias o causas de fallos en el sistema.	1. El arquitecto de software identifica las posibles deficiencias o causas de fallos que se pueden originar en el sistema. 2. Se diagnostican las deficiencias o causas de fallos partiendo de los resultados que arrojó el análisis anterior.
<<1>>. <<c>>< Identificar las partes a modificar>	**Respuesta: Flujo de eventos (Escenarios)**
Se identifica las partes a modificar en el sistema.	1. El arquitecto de software identifica los paquetes y clases implicados en el cambio, así como las funciones que se reutilizarán o se crearán para introducir el cambio.
Medida de respuesta	
Analizar el cambio en el sistema.	

Atributo de Calidad	Mantenibilidad.
Sub-atributos/Sub-característica	Cambiabilidad.
Objetivo	Capacidad del producto que permite que sea modificado de forma efectiva y eficiente sin introducir defectos o degradar el desempeño.
Origen	Desarrollador.
Artefacto	El código fuente.
Entorno	El ambiente de desarrollo del sistema.
Estímulo	**Respuesta: Flujo de eventos (Escenarios)**
<<1>>. <<a>>< Capacidad de modificación del sistema>	
La arquitectura del sistema está diseñada para brindar facilidades a la hora de introducir modificaciones en el sistema. Esto permite que se puedan introducir cambios o modificaciones, que no afecten el correcto desempeño del resto de la solución.	NA
Medida de respuesta	
Introducir una modificación al sistema.	

Atributo de Calidad	Portabilidad.
Sub-atributos/Sub-característica	Adaptabilidad.

Objetivo	Capacidad del producto que le permite ser adaptado de forma efectiva y eficiente a diferentes entornos determinados de hardware, software, operacionales o de uso.
Origen	Desarrollador.
Artefacto	El sistema.
Entorno	El sistema desplegado.
Estímulo	**Respuesta: Flujo de eventos (Escenarios)**
<<1>>. <<a>>< Capacidad del sistema de adaptarse de forma efectiva a diferentes entornos>	
El sistema está diseñado con tecnologías que permiten que este se adapte a cualquier sistema operativo.	NA
Medida de respuesta	
El ambiente de despliegue del sistema.	

Atributo de Calidad	Compatibilidad.
Sub-atributos/Sub-característica	Co-existencia.
Objetivo	Grado en que un producto puede realizar sus funciones necesarias de manera eficiente mientras comparte un entorno común y recursos con otros productos, sin impacto perjudicial en cualquier otro producto.
Origen	Desarrollador.
Artefacto	El sistema.
Entorno	El sistema desplegado.
Estímulo	**Respuesta: Flujo de eventos (Escenarios)**
<<1>>. <<a>>< Facilidad con la que el sistema comparte un entorno común y recursos con otros sistemas>	
El sistema cuenta con un alto grado de co-existencia pues es capaz de convivir con otras aplicaciones que comparten los recursos de la estación de trabajo donde se encuentra sin impactar perjudicialmente a otras aplicaciones.	NA
Medida de respuesta	
Desplegar el sistema.	

Atributo de Calidad	Funcionabilidad.
Sub-atributos/Sub-característica	Funcionabilidad.
Objetivo	Grado en el que el conjunto de funciones cubre todas las tareas y objetivos del usuario especificados.
Origen	Proveedor de requisitos.
Artefacto	El sistema.

Entorno	El sistema desplegado.
Estímulo	**Respuesta: Flujo de eventos (Escenarios)**
<<1>>. <<a>>< Grado en el que el sistema cubre todas las tareas y objetivos especificados>	
El desarrollo del sistema está guiado por las necesidades expresadas por parte de los proveedores de requisitos, dándole total cumplimiento a sus especificaciones declaradas e implícitas.	NA
Medida de respuesta	
Analizar las funcionalidades del sistema.	

Atributo de Calidad	Eficiencia en el rendimiento.
Sub-atributos/Sub-característica	Utilización de recursos.
Objetivo	Grado en el que las cantidades y tipos de recursos utilizados por un producto o sistema, al realizar sus funciones cumplen con los requisitos.
Origen	Desarrollador.
Artefacto	El sistema.
Entorno	El sistema desplegado.
Estímulo	**Respuesta: Flujo de eventos (Escenarios)**
<<1>>. <<a>>< Características de Hardware >	
PC: • Cantidad: 1 PC. • CPU: Core i3. • RAM: 4Gb. • HDD: 200 Gb.	1. El sistema funcionando correctamente.
Estímulo	**Respuesta: Flujo de eventos (Escenarios)**
<<1>>. <<a>>< Características de Software >	
• Tener instalado sistema operativo Windows o Linux	1. El sistema funcionando correctamente.
Medida de respuesta	
Navegar en el sistema.	

Anexo Criterios de evaluación de la métrica TOC

Atributo de Calidad	Categoría	Criterio
Responsabilidad	Baja	<=Promedio
	Media	Entre Promedio y 2*Promedio
	Alta	>Promedio
Complejidad de implementación	Baja	<=Promedio
	Media	Entre Promedio y 2*Promedio
	Alta	>2*Promedio
Reutilización	Baja	>2*Promedio
	Media	Entre Promedio y 2*Promedio
	Alta	<=Promedio

Anexo Criterios de evaluación de la métrica RC

Atributo de Calidad	Categoría	Criterio
Acoplamiento	Baja	Entre 0 y 1
	Media	2
	Alta	>2
Complejidad de mantenimiento	Baja	<=Promedio
	Media	Entre Promedio y 2*Promedio
	Alta	>2*Promedio
Reutilización	Baja	>2*Promedio
	Media	Entre Promedio y 2*Promedio
	Alta	<=Promedio
Cantidad de pruebas	Baja	<=Promedio
	Media	Entre Promedio y 2*Promedio
	Alta	>2*Promedio

Anexo Instrumento de evaluación de la métrica TOC

Clase	Cantidad de procedimientos	Responsabilidad	Complejidad de implementación	Reutilización
IBootstrapper	1	Baja	Baja	Alta
IGenericManagerFunctionality	9	Media	Media	Media
IMenuService	3	Baja	Baja	Alta
IModule	1	Baja	Baja	Alta
IProblemInstanceManager	16	Alta	Alta	Baja
IProblemManager	12	Media	Media	Media
IRegionMgr	10	Media	Media	Media
IRegion	2	Baja	Baja	Alta
QServiceLocator	11	Media	Media	Media
LifetimeManager	5	Baja	Baja	Alta
SingletonLifetimeManager	2	Baja	Baja	Alta
TransientLifetimeManager	2	Baja	Baja	Alta
AppBootst	6	Baja	Baja	Alta
GenericMgrFunct	9	Media	Media	Media
ProbInstMgr	16	Alta	Alta	Baja
ProbMgr	12	Media	Media	Media
RegionMgr	10	Media	Media	Media
PluginMgr	5	Baja	Baja	Alta
Promedio de métodos por clases	**7,333333**			

Anexo Instrumento de evaluación de la métrica RC

Clase	Cantidad de relaciones de uso	Acoplamiento	Complejidad de mantenimiento	Reutilización	Cantidad de pruebas
IBootstrapper	0	Baja	Baja	Alta	Baja
IGenericManagerFunctionality	0	Baja	Baja	Alta	Baja
IMenuService	0	Baja	Baja	Alta	Baja
IModule	1	Baja	Baja	Alta	Baja
IProblemInstanceManager	8	Alta	Alta	Baja	Alta
IProblemManager	3	Alta	Media	Media	Media
IRegionMgr	2	Media	Baja	Alta	Baja
IRegion	0	Baja	Baja	Alta	Baja
QServiceLocator	2	Media	Baja	Alta	Baja
LifetimeManager	0	Baja	Baja	Alta	Baja

SingletonLifetimeManager	1	Baja	Baja	Alta	Baja
TransientLifetimeManager	1	Baja	Baja	Alta	Baja
AppBootst	4	Alta	Media	Media	Media
GenericMgrFunct	1	Baja	Baja	Alta	Baja
ProbInstMgr	8	Alta	Alta	Baja	Alta
ProbMgr	3	Alta	Media	Media	Media
RegionMgr	2	Media	Baja	Alta	Baja
PluginMgr	1	Baja	Baja	Alta	Baja
Promedio	**2,0555555**				