DIVISIÓN DE ESTUDIOS DE POSGRADO E INVESTIGACIÓN

OPCION I.- TESIS

TRABAJO PROFESIONAL

"DESARROLLO DE UN GENERADOR DE CASOS DE PRUEBAS FUNCIONALES E INTEGRALES, A PARTIR DE CASOS DE USO DESCRITOS EN RDL, PARA LA EMPRESA SOFTTEK ®"

QUE PARA OBTENER EL GRADO DE:

MAESTRA EN
SISTEMAS COMPUTACIONALES

PRESENTA:
I.S.C. RAQUEL LIZUE MARTÍNEZ RAMOS

DIRECTOR DE TESIS:
M.C.E. BEATRIZ ALEJANDRA OLIVARES ZEPAHUA

ORIZABA, VER. MÉXICO NOVIEMBRE 2020

Orizaba, Veracruz, 13/11/2020
Dependencia: División de Estudios de Posgrado e Investigación
Asunto: Autorización de Impresión
OPCION: I

C. RAQUEL LIZUE MARTÍNEZ RAMOS
Candidato a Grado de Maestro en:
SISTEMAS COMPUTACIONALES
P R E S E N T E.-

De acuerdo con el Reglamento de Titulación vigente de los Centros de Enseñanza Técnica Superior, dependiente de la Dirección General de Institutos Tecnológicos de la Secretaría de Educación Pública y habiendo cumplido con todas las indicaciones que la Comisión Revisora le hizo respecto a su Trabajo Profesional titulado:

" DESARROLLO DE UN GENERADOR DE CASOS DE PRUEBAS FUNCIONALES E INTEGRALES, A PARTIR DE CASOS DE USO DESCRITOS EN RDL, PARA LA EMPRESA SOFTTEK®"

Comunico a Usted que este Departamento concede su autorización para que proceda a la impresión del mismo.

A T E N T A M E N T E
Excelencia en Educación Tecnológica®
CIENCIA – TÉCNICA - CULTURA®

EDUCACIÓN | TECNOLÓGICO NACIONAL DE MÉXICO
INSTITUTO TECNOLÓGICO DE ORIZABA
DIV. DE EST. DE POSGRADO E INVEST.

DR. MARIO L. ARRIOJA RODRÍGUEZ
JEFE DE LA DIVISIÓN DE ESTUDIOS
DE POSGRADO E INVESTIGACIÓN

FECHA: **21/07/2020**

ASUNTO: Revisión del Trabajo Escrito

C. MARIO LEONCIO ARRIOJA RODRIGUEZ
JEFE DE LA DIVISION DE ESTUDIOS
DE POSGRADO E INVESTIGACION
P R E S E N T E.

Los que suscriben miembros del jurado, han realizado la revisión de la Tesis del (la) C.:

RAQUEL LIZUÉ MARTÍNEZ RAMOS
No. DE CONTROL: **M13011178**

La cual lleva el título de:

"Desarrollo de un Generador de Casos de Pruebas Funcionales e Integrales, a partir de Casos de Uso descritos en RDL, para la empresa SOFTTEK®"

Y concluyen que se acepta.

A T E N T A M E N T E

PRESIDENTE: M.C.E. BEATRIZ A. OLIVARES ZEPAHUA

SECRETARIO: M.C. CELIA ROMERO TORRES

VOCAL: M.S.C. LUIS ÁNGEL REYES HERNÁNDEZ

VOCAL SUP.: M.R.T. IGNACIO LÓPEZ MARTÍNEZ

EGRESADO (A) DE LA MAESTRIA EN **SISTEMAS COMPUTACIONALES**

OPCION: **1 Tesis**

Agradecimientos

A mi lindo pez rojo, ya que es el héroe más grande que conozco, y mi regalo de toda la vida.

A mi mami Eréndira, por darme a mi hermana, por ser mi apoyo, mi mejor amiga, mi ejemplo a seguir; y porque sin tus mimos no hubiera podido concluir esta etapa.

A mi papá Sergio, porque tu fuerza de voluntad es lo que más admiro de ti, y nadie más que tú será mi padre.

A mis tíos, porque hicieron de mi estadía en otro estado algo inolvidable.

A Noé, simplemente por quererme tal cual soy.

A mis amigos, porque sin su compañía todo hubiera sido más triste.

A mi asesora la maestra Bety, quien cariñosamente me orientó y acompañó en esta travesía.

A todos mis maestros, porque la labor de enseñar no termina en el aula, y su participación en mi vida es la base de este proyecto.

A la empresa SOFTTEK ® por brindarme la oportunidad de colaborar con ellos, sobre todo al responsable de mi estancia profesional, Edgar Felipe Fuentes Perea, por su dedicación, tiempo y disponibilidad cuando trabajó conmigo.

A TECNM, porque me permitió desarrollar mis habilidades académicas.

A CONACYT, por seguir apoyando la ciencia en México y por hacerme parte de ese proceso.

Índice General

Índice de Figuras

Índice de Tablas

Resumen

La etapa de Pruebas se ha convertido en una de las más cruciales en el ciclo de vida del Software, ya que ahora es más imperioso generar y distribuir productos de alta calidad en poco tiempo, por lo que se hace uso de Escenarios de Prueba, ya que son procesos que validan la funcionalidad del software a través de la detección de errores, dichos escenarios representan pruebas funcionales o integrales y cuando se llena un escenario con cierta información, se dice que se creó un Caso de Prueba.

Por lo que los esfuerzos actuales se centran en generar estos escenarios de forma automática para reducir el tiempo de prueba y los costos de esta labor, sin embargo, esta labor es complicada, costosa y tardada, por lo que se buscó desarrollar un Generador de Escenarios de Prueba funcionales e integrales a partir de archivos JSON equivalentes a Casos de Uso descritos en el lenguaje de definición de requerimientos de la empresa Softtek®.

Dicho generador crea grafos dirigidos a partir de la interpretación de los antes mencionados archivos JSON, ya que los escenarios de prueba son el producto de recorrer todos los nodos presentes entre un inicio y un fin determinado. Los atributos de los nodos de los grafos incluyen información suficiente con respecto a los detalles de la prueba.

Se realizó el proyecto aplicando el enfoque de la metodología XP y el algoritmo de Búsqueda en Profundidad con las restricciones tecnológicas determinadas por la empresa: el generador es un API (*Application Program Interface*) en lenguaje Java listo para incorporarse a las herramientas de Ingeniería de Software de la empresa, usando la plataforma de desarrollo Eclipse y la base de datos orientada a grafos Neo4J.

Abstract

The Test stage has become one of the most crucial in the software life cycle, because now it is more imperative to generate and distribute high quality products in a short time, so we make use of Test Scenarios, as they are processes that validate the functionality of the software through error detection, these scenarios can represent functional or comprehensive testing and when a scenario is filled with certain information, it is said that a Test Case was created.

So the current efforts are focused on generating these scenarios automatically to reduce the testing time and costs of this work, however, this work has become complicated, expensive and time-consuming, so we seek to develop a functional and comprehensive Test Scenario Generator from JSON files equivalent to Use Cases described in the requirements definition language of the company Softtek®.

This generator must create directed networks based on the interpretation of the aforementioned JSON files, since the test scenarios must be the product of going through all the nodes present between a given start and end.

The attributes of the network nodes must include sufficient information regarding the test details.

Using the XP methodology approach and the Deep Search algorithm it is possible to carry out the project with the following technological designations made by the company: Eclipse development platform and the Neo4J network oriented database.

Introducción

Anteriormente, probar era una actividad de poco interés entre los programadores, e incluso era relegada ya que se consideraba una pérdida de tiempo, sin embargo, actualmente esto ha cambiado, ahora dicha etapa es una práctica crítica que requiere supervisión, dando paso a los escenarios de prueba, que no son más que especificaciones del proceso de ejecutar un producto de software con la intención de encontrar errores; cuando uno de estos escenarios es poblado con información específica, se dice que se tiene un caso de prueba, por lo que un escenario da paso a más de uno de estos casos.

Ahora es más común que un elemento de software genere cientos de escenarios de pruebas, sin embargo, no es necesario probar cada uno de estos para validar el funcionamiento deseado de tal elemento, por lo que gracias a una serie de algoritmos es posible obtener el mínimo conjunto de pruebas necesarias que cumplen con dicha labor de validación.

Debido a la importancia de probar el software, los esfuerzos actuales se centran en generar pruebas de forma automática para reducir el tiempo de validación de un producto programado y generar software más apegado al funcionamiento deseado por el usuario; sin embargo, esta tarea se dificulta desde el momento en que se decide de dónde partir para generar escenarios de pruebas, ya que lo ideal es que se tome una especificación de requisitos para contrastar el comportamiento programado contra el deseado, pero, dependiendo del lenguaje de definición de requerimientos la creación de la prueba se complica y/o se vuelve costosa, además de que es posible que desde este punto se introduzcan errores. Por lo tanto, al utilizar el lenguaje de requerimientos de la empresa Softtek®, es posible eliminar todas las desventajas que se presentan al seleccionar un punto de origen para generar automáticamente escenarios de prueba.

El presente documento está organizado en cuatro capítulos para explicar los antecedentes teóricos y prácticos relacionados al tema (Capítulo uno), el estado de la práctica, que incluye un estudio de todos los trabajos realizados con anterioridad para comprender el trabajo que existe actualmente en el mercado (Capítulo dos), la aplicación de la metodología para la realización del proyecto (Capítulo tres) y los resultados obtenidos tras sometes a dos casos de estudio los productos obtenido (Capítulo cuatro)

Capítulo 1 Antecedentes

En el presente capítulo se exponen los criterios más relevantes para la tesis, su planteamiento, objetivo general, objetivos específicos y justificación.

1.1 Marco Teórico

A continuación, se describen los conceptos más sobresalientes para la realización de este proyecto.

1.1.1 Pruebas de software

G. J. Myers [1] definió que una prueba de software es el proceso de ejecutar un programa con la intención de encontrar errores; además, este proceso ayuda a verificar el comportamiento del producto.

Durante muchos años, las actividades de prueba de software fueron consideradas como un castigo para los programadores, incluso se pensaba que eran una pérdida de tiempo, ya que se suponía que el software era correcto desde el principio. Sin embargo, los sistemas cambiaron y ahora la disciplina de prueba es considerada de extrema importancia. Hoy en día estos procedimientos son reconocidos como una parte integral del proceso de desarrollo de software a tal grado que incluso las grandes empresas de software delegan la realización de pruebas a otras empresas [2].

Las pruebas se clasifican de la siguiente manera [1]:

- Pruebas unitarias: sirven para verificar el comportamiento de las clases, incluidos sus métodos básicos y delegados.
- Pruebas funcionales: sirven para verificar sistemáticamente los resultados obtenidos después de ejecutar una prueba con un conjunto de parámetros válidos e inválidos.
- Pruebas del sistema: son básicamente pruebas de casos de uso sistemático.

1.1.1.1 Pruebas funcionales

Las pruebas funcionales consisten en una secuencia de pruebas que definen valores de entrada para una operación con el fin de observar si se llega al resultado esperado. Estas pruebas se

ejecutan sin ningún conocimiento del código de programación implementado, ya que solo es necesario evaluar el comportamiento del software. La cantidad de pruebas que se realizarán para asegurar que una operación sea correcta es virtualmente infinita, por lo que se usan mayormente dos técnicas particulares para reducir este número sin perder la cobertura, a continuación, se describen:

- **Partición equivalente**

  Uno de los principios de las pruebas funcionales es la identificación de situaciones equivalentes. Por ejemplo, si una operación acepta un conjunto de datos (normalidad) y rechaza otro conjunto (excepción), entonces se dice que existen dos conjuntos de equivalencia de datos de entrada para esa operación: valores aceptados y valores rechazados. Por lo general, es imposible probar cada valor incluido en esos conjuntos, por lo que la técnica de partición de equivalencia indica que al menos un elemento en cada conjunto de equivalencia se probó.

- **Análisis de valor límite**

  El análisis del valor límite consiste en no elegir ningún valor aleatorio de un conjunto de equivalencia, sino elegir dos o más valores límite determinados. Por ejemplo, si una operación requiere un parámetro entero que es válido sólo si está incluido en el intervalo [10...20], entonces hay tres conjuntos de equivalencia:

  - Inválido para cualquier x < 10.
  - Válido para x >= 10 y x = 20.
  - Inválido para cualquier x < 20.

  Así, el dominio de parámetros de valores límites es:

  - Nueve, para el primer conjunto inválido.
  - 10 y 20, para el conjunto válido.
  - 21, para el segundo conjunto inválido

Por lo tanto, si hay un error lógico en la operación para algunas de estas entradas, es más probable que sí se detecte el error en comparación a si se elige un valor aleatorio dentro de cada intervalo que define un conjunto de equivalencia.

Las pruebas funcionales se clasifican en tres niveles [2]:

- Prueba de unidad funcional para operaciones básicas y delegadas.
- Pruebas de funcionamiento del sistema funcional basadas en contratos.
- Pruebas de sistema y aceptación basadas en casos de uso del sistema.

1.1.1.2 Planeación de pruebas

Un plan es un documento que proporciona un marco o enfoque para lograr un conjunto de objetivos. Los planes de prueba tienden a ser más orientados técnicamente. Sin embargo, la planificación de pruebas debe comenzar temprano en el ciclo de vida del software, aunque para muchas organizaciones cuyos procesos de prueba son inmaduros, esta práctica aún no se ha implementado. Los modelos como el modelo V, o el modelo V ampliado, ayudan a respaldar las actividades de planificación de pruebas que comienzan en la fase de requisitos y continúan en fases sucesivas del desarrollo de software [3].

Es más factible primero planear el diseño de pruebas unitarias antes que las pruebas funcionales, ya que antes de verificar el comportamiento del software hay que validar el código. Sin embargo, si se usó algún tipo de generación automática de código, es posible prescindir de estas pruebas, ya que se sostiene la idea que estos generadores no cometen errores [2].

1.1.1.3 Criterios de adecuación de pruebas

Los evaluadores de software necesitan un marco para decidir qué elementos y datos de prueba son necesarios para considerar que los esfuerzos de prueba son lo suficientemente adecuados para finalizar el proceso con la confianza de que el software funciona correctamente. Dicho marco existe en forma de criterios de adecuación de prueba.

Formalmente, un criterio de adecuación de datos de prueba es una regla que sirve para determinar si se realizaron suficientes pruebas. Estos criterios representan estándares mínimos para probar un programa por lo que el análisis de cobertura se refiere a estos criterios.

Cuando un objetivo de prueba relacionado con la cobertura se expresa con un porcentaje, se le denomina grado de cobertura. El grado de cobertura planificado se especifica en el plan de prueba y luego se mide cuando las pruebas son ejecutadas por una herramienta de cobertura, además, generalmente se especifica en 100% si el evaluador desea satisfacer completamente la suficiencia de la prueba o los criterios de cobertura.

En algunas circunstancias, el grado de cobertura planificado es inferior al 100%, debido a lo siguiente:

- La naturaleza de la unidad.
- La falta de recursos.
- Otros problemas relacionados con el proyecto, como la programación [3].

Para motivos del presente trabajo, cabe destacar que el objetivo de prueba abarca el 100% de los casos de prueba sobre el sistema.

1.1.2 Casos de prueba

Un caso de prueba es un elemento que contiene la siguiente información:

1. Conjunto de entradas de prueba. Estos son elementos de datos recibidos de una fuente externa como hardware, software o humano.
2. Condiciones de ejecución. Estas son condiciones requeridas para ejecutar la prueba, por ejemplo, un estado de la base de datos o una configuración de un dispositivo de hardware.
3. Salidas esperadas. Estos son los resultados especificados para ser producidos por el código bajo prueba.

Además, son documentos muy complejos y detallados que usualmente incluyen las siguientes características:

- Objetivos generales de la prueba, por qué se está probando, qué se espera lograr con la prueba y cuáles son los riegos de realizar la prueba.
- Qué se va a probar (alcance de las pruebas), ¿qué elementos, características, procedimientos, funciones, objetos, grupos y subsistemas se probarán?

- ¿Quién va a probar? ¿Quién es el personal responsable de las pruebas?
- Cómo realizar la prueba, qué estrategias, métodos, hardware, herramientas de software, y técnicas se van a aplicar.
- ¿Cuándo probar?, cuáles son los horarios de las pruebas.
- ¿Cuándo dejar de probar?, ya que no es económicamente factible o práctico probar hasta que todos los defectos se corrijan, debido a las limitaciones de tiempo.

Los casos de prueba se organizan de varias maneras según la política de la organización; a continuación, se enlistan lo componentes de un plan de pruebas genérico [3].

1. Identificador de plan de prueba.
2. Introducción.
3. Objeto por probar.
4. Características por probar.
5. Enfoque.
6. Criterios de aprobación / rechazo.
7. Criterios de suspensión y reanudación.
8. Resultados de prueba.
9. Tareas de prueba.
10. Entorno de prueba.
11. Responsabilidades.
12. Necesidades de personal y formación.
13. Programación.
14. Riesgos y contingencias.
15. Costos de prueba.
16. Aprobaciones.

Existen dos formas de diseño de casos de prueba con respecto al orden de ejecución de la prueba.

- **Casos de prueba en cascada**

  Es posible construir los casos de prueba de forma anidada. Por ejemplo, la primera prueba ejerce una característica particular del software y luego deja el sistema en un

estado tal que el segundo caso de prueba se ejecute. O sea, cada uno se genera con base en la prueba anterior. La ventaja es que cada caso de prueba es típicamente más pequeño y simple. La desventaja es que, si una prueba falla, las pruebas posteriores son inválidas.

- **Casos de prueba independientes**

  Cada caso de prueba es totalmente independiente. Las pruebas no requieren que otras pruebas se hayan ejecutado con éxito. La ventaja es que es posible ejecutar cualquier cantidad de pruebas sin considerar un orden. La desventaja es que cada prueba tiende a ser más grande y más compleja, por lo tanto, es más difícil de diseñar, crear y mantener [4].

1.1.2.1 Algoritmos para la creación de casos de prueba

Aún es común encontrar que los casos de prueba son obra manual de quienes conforman el equipo de desarrollo de software, lo que provoca que se consuma más tiempo y dinero del destinado a probar, por lo que una forma más automatizada para crear casos de prueba, es usar algoritmos específicos que permitan la creación de estas usando menos recursos.

Los aportes fundamentales de las propuestas para la creación de casos de prueba, están dirigidos a la utilización de algoritmos metaheurísticos, estos provienen de la disciplina de "Ingeniería de Software" y se utilizan de forma empírica, y por tanto el problema combinatorio de técnicas que produzcan pruebas aceptables continúa sin reducirse significativamente. Varias propuestas son prototipos para validar la solución teórica, pero no se han incorporado a las soluciones comerciales de generación de los casos de prueba, de forma tal que desarrolladores y equipos de probadores los utilicen, reduciendo así el esfuerzo vinculado con esta actividad de diseño que es altamente costosa [5].

1.1.2.2 Priorización de Casos de Prueba

TCP (*Test Case Prioritization*, Priorización de Casos de Prueba), son técnicas de priorización de casos de prueba basadas en el grado de cobertura, de información histórica o en modelos, costos y tiempo o riesgos.

Las técnicas de selección de casos de prueba minimizan lo que se comprende como el paquete de pruebas (el total de todos los casos de prueba), por lo que en [6] se demostró una disminución en la tasa de fallas, entonces se dice que la reducción del paquete de pruebas garantiza una cobertura del 100% de los requisitos sobre el paquete de pruebas reducido, pero no se garantiza la misma capacidad de detección de fallas en el paquete real.

Debido a esta desventaja, las TCP actualmente tratan de no descartar los casos de prueba, sino de que las primeras pruebas que se ejecuten sean las más significativas, por lo que existen diferentes enfoques para el establecimiento de la prioridad, a continuación, se enlistan los criterios de prioridad más usados según [6]:

- **Priorización basada en la cobertura:** tienen como objetivo la maximización de la cobertura en los elementos del programa, por lo que se necesita de un conocimiento detallado del código fuente.

- **Priorización basada en información histórica:** utilizan métodos históricos basados en los registros del caso de prueba, prácticamente utilizan la experiencia de haber probado un elementó para decir su prioridad.

- **Priorización consciente de los costos:** es un intento de convertir el APFD (*Average Percentage of Faults Detected*, Porcentaje medio de fallos detectados) en una escala de beneficios; aunque esta técnica genera ahorros, tiene sus propios gastos generales de implementación.

- **Priorización de Requisitos y Riesgos:** diseña un número mínimo de casos de prueba a partir de los requisitos, tomando en cuenta que el costo de corregir un error encontrado en la prueba sea el mínimo.

- **Priorización basada en modelos:** se utilizan modelos de especificación como los diagramas de estado o de actividad para priorizar las pruebas de las que más dependa el funcionamiento del sistema.

- **Priorización enfocada en GUI/Aplicación Web:** al probar aplicaciones web las secuencias de eventos que son realizadas por los usuarios son grabadas, por lo que es posible utilizar estas combinaciones para crear un número de casos de prueba priorizados con respecto a los eventos más usados a través de la GUI (*Graphic User Interface*, Interfaz Gráfica de Usuario) de la aplicación web.

Los métodos de priorización conscientes de la cobertura son dominantes en el campo de las pruebas, y los segundos mejores son los basados en requisitos.

1.1.3 Grafo Dirigido

Un grafo dirigido (o dígrafo) G consiste de un conjunto de vértices y un conjunto de arcos E. A los vértices se les llama también nodos o puntos y a los arcos aristas dirigidas o líneas dirigidas. Un arco es un par ordenado de vértices (v, w) donde v es la cola y w la cabeza del arco. Un arco (v, w) se expresa también como *v w* y se dibuja como en la Figura 1.1.

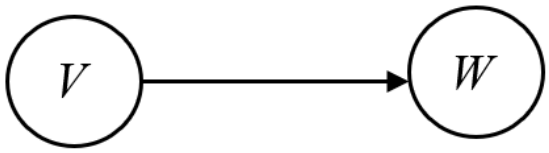


Figura 0.1 Representación de un grafo dirigido

Se dice que un arco *v w* va de *v* a *w* y que *w* es adyacente a *v*. Los vértices de un dígrafo se usan para representar objetos, y los arcos-relaciones entre los objetos.

Un camino en un dígrafo es una secuencia de vértices *v1*, *v2*, …, *vn*, tal que *v1 v2*, *v2 v3*.., *vn-1 vn* son arcos. El camino es de *v1* a *vn* y pasa a través de los vértices *v2, v3,…, vn-1* y termina en el *vn*. La longitud de un camino es el número de arcos del camino, en este caso, $n-1$ [7].

Un ciclo simple es un camino de longitud uno como mínimo, que empieza y termina en el mismo vértice. Un grafo etiquetado es un dígrafo en el que cada arco y/o vértice tiene una etiqueta asociada. Una etiqueta tiene distintos valores posibles como un nombre, un costo o un valor de algún tipo de dato dado [8].

1.1.3.1 Grafos de cobertura y flujo de control

El análisis de cobertura generalmente está asociado con el uso de modelos de control y flujo de datos para representar los elementos y datos estructurales del programa. Los elementos lógicos considerados para la cobertura se basan en el flujo de control de una unidad de código, por ejemplo, a) declaraciones del programa, b) decisiones / ramas (que influyen en el flujo de control del programa), c) condiciones (expresiones que se evalúan solo como verdaderas / falsas), d) combinaciones de decisiones y condiciones y e) caminos (secuencias de nodos en graficas de flujo), entonces, podemos decir que una forma de representar un flujo de proceso de prueba es un grafo.

1.1.3.2 Base de datos orientada a grafos.

Una Base de Datos en Grafo es una base de datos que tiene como propósito almacenar estructuras de datos que tienen topología de grafo, es decir, que la información que almacena se representa por medio de nodos y aristas entre ellos. Por definición, este tipo de base de datos agruparía a cualquier solución de almacenamiento en la que los elementos que están conectados se enlazan sin hacer uso de una referencia por medio de índices (que sería el método habitual de "simular" una relación en una Base de Datos Relacional), de esta forma, los vecinos de una entidad son accesibles directamente por ella por medio de una referencia directa, sin pasar por estructuras intermedias que hagan el proceso de referenciado.

En esta definición no se toma en cuenta el tipo de grafo (en su sentido más amplio) que nuestros datos seguirán, ni en el tipo de aristas (dirigidas o no), ni en la multiplicidad de las mismas entre dos nodos (unirelacional o multirelacional), ni en la n-aridad que reflejen las aristas (grafo o hipergrafo) [9].

Por tanto, una BDG debe cumplir los siguientes criterios:

- El almacenamiento está optimizado para representar datos como un grafo, proporcionando los nodos y las aristas que lo forman.
- El almacenamiento está optimizado para realizar trasversales a través del grafo, sin uso de índices que "representen" las aristas. Por ello, estas bases están optimizadas para

realizar consultas en los datos en las que intervengan relaciones de proximidad (conexiones) entre datos, y no para la realización de consultas globales.

- El modelo de datos es flexible... lo que en algunas ocasiones nos permite no declarar tipos de nodos o de aristas (el tipo de nodo o arista sería equivalente a las distintas tablas que se definen en un modelo relacional).
- Integra funciones para aplicar los algoritmos clásicos de la Teoría de Grafos (caminos más cortos, A*, medidas de centralidad, etc.).

Si un grafo es capaz de representar los pasos de un algoritmo a través de nodos y sus relaciones, es indispensable contar con un medio para su almacenamiento.

1.1.4 Lenguaje de definición de requerimientos

Un RDL (*Requirements Definition Language*, Lenguaje de Definición de Requerimientos), es aquel que permite representar de forma estandarizada las características deseables de un software específico.

El lenguaje natural controlado (CNL) es conocido por todos los participantes en el ciclo de vida de los requisitos, como deriva del Lenguaje Natural (NL), es común que se convierta en el mecanismo idóneo para expresar requerimientos, sin embargo, posee una serie de características que le dotan de ambigüedad y variabilidad estructural, propiciando la introducción de errores al desarrollo de casos de prueba [10].

1.1.5 Casos de uso

Un caso de uso (UC) es una unidad coherente del funcionamiento de un sistema de software que describe una secuencia de acciones que realiza dicho sistema y que lleva a un resultado de valor a un actor específico.

Un UC está compuesto por dos partes, un diagrama (gráfico) y una parte textual. El diagrama muestra las relaciones entre actores y funciones del sistema mientras que la parte textual muestra la descripción escrita en lenguaje natural de los pasos y demás características del caso de uso.

La Tabla 1.1 muestra las partes y las indicaciones básicas para iniciar la redacción de la descripción de un UC [11].

Tabla 0.1 Descripción general para un Caso de Uso

Caso de Uso	CU.1 XX
Actor	Pseudónimos de los actores que interaccionan con la funcionalidad del UC.
Descripción	Descripción del UC.
Flujo básico	**1. Título del paso** Descripción del paso. **2. Título del paso** Descripción del paso.
Flujos alternos	**1. Título del FA** Descripción del FA. **2. Título del FA** Descripción del FA.
Precondiciones	**1. Título del Precondición** Descripción del PRC.
Postcondiciones	**1. Título del Postcondiciones** Descripción de la PTC.
Requerimientos trazados	**1. Título del requerimiento** Descripción del requerimiento o por qué se enlaza a él desde este caso de uso.
Puntos de inclusión	**1. Título del punto de inclusión** Descripción del punto de inclusión.
Puntos de extensión	**1. Título del punto de extensión** Descripción del punto de extensión.
Notas	**1. Título de la Nota** Descripción de la nota.

1.1.5 Generación de casos de prueba a partir de casos de uso

La prueba de software es una actividad que requiere mucho tiempo y recursos, por lo tanto, las pruebas automatizadas se esfuerzan por cubrir estas deficiencias y dar un resultado más preciso que las pruebas manuales, ya que estas son vulnerables a los errores humanos. Actualmente, se está automatizando la generación de casos de prueba a partir de un código fuente o un modelo visual de software como lo es UML (*Unified Modeling Language*, Lenguaje de Modelado Unificado).

Otras investigaciones apuntan a la utilización de técnicas de recuperación de información para la extracción automática de conceptos de código fuente con el fin de reducir los casos de prueba.

Las pruebas basadas en modelos donde los casos de prueba se generan a partir del modelo del software muestran una mayor eficiencia de tiempo y esfuerzo. Además, la generación de casos de prueba en la fase temprana del ciclo de vida del desarrollo del software proporciona una gestión de control en la fase de construcción y prueba [12].

Algunos enfoques han preferido la descripción de requisitos en métodos formales para eliminar la ambigüedad del NL, lo que aumenta la complejidad de la generación de pruebas debido a la dificultad que representa el dominio de tales métodos, además, se requiere de mucha experiencia en la interpretación de los requisitos y solo un número limitado de personas los entiende, por lo que es mucho más factible para muchos la generación de casos de prueba a partir de los requisitos de software expresados en lenguaje natural utilizando una técnica de procesamiento de lenguaje natural [13].

1.2 Situación tecnológica, económica y operativa de la empresa

Softtek® es un proveedor global de servicios orientados a procesos de TI con 30 oficinas en Norteamérica, Latinoamérica, Europa y Asia. Con 15 Centros de Desarrollo Global en EE. UU., México, China, Brasil, Argentina, Costa Rica, España, Hungría e India.

Softtek® mejora el tiempo de entrega de soluciones de negocio, reduce costo de las aplicaciones existentes, entrega aplicaciones mejor diseñadas y probadas, y produce resultados predecibles para grandes empresas en más de 20 países. A través de modelos de servicios de entrega *on-site, on-shore* y su marca registrada Global Nearshore™, Softtek® ayuda a los CIO (*Chief Information Officer*, Director de la Oficina de Información) a incrementar el alineamiento con el negocio [14].

1.3 Planteamiento del problema

Los casos de prueba son un medio útil para corroborar la funcionalidad del software, sin embargo, generarlos es una tarea complicada porque se debe considerar el hecho de que no basta con crear solo uno por componente o por caso de uso, por lo que los esfuerzos se dirigen a generar casos de prueba de forma automática.

Los enfoques actuales apuntan a la construcción de pruebas a partir de la definición de requerimientos para cubrir las funcionalidades especificadas por el cliente, sin embargo, el

problema es transformar las especificaciones del lenguaje natural a formas menos ambiguas como las definiciones formales, sin embargo, esto es costoso y tardado puesto que se necesita de la intervención de un experto, por otra parte, está la opción de los lenguajes naturales controlados, aunque también presentan deficiencias tales como la selección y procesamiento de palabras clave que figuren en un diccionario limitado.

Aún sin resolver totalmente el problema de donde partir para generar estos casos, surge el concepto de cobertura, que se refiere al número de pruebas necesarias para determinar si una pieza de software funciona adecuadamente, ya que el número de casos de prueba que se generan para un componente tiende a infinito, esto significa una inversión de tiempo enorme solo para probar, lo cual no es idóneo si se tiene que liberar el proyecto en un periodo determinado de tiempo, por lo que es necesario optimizar este número de pruebas con el objetivo de cumplir con aquellos que formen la ruta crítica.

La ruta crítica se refiere a los casos de prueba que sirven para detectar fallas o incidentes en el sistema, por lo que la cobertura debe incluir estos casos, sin embargo, esto no siempre es así, el problema es cumplir con el objetivo de probar menos cubriendo más, y hasta el momento las técnicas dedicadas a la generación de pruebas no cumplen satisfactoriamente esta consigna.

1.4 Objetivo general y objetivos específicos

A continuación, se exponen el objetivo general y los objetivos específicos pertinentes para el desarrollo del proyecto de tesis.

1.4.1 Objetivo general

Desarrollo del generador de casos de prueba funcionales e integrales para el RDL de la empresa Softtek®.

1.4.2 Objetivos específicos

- Analizar las características del RDL de Softtek®.
- Analizar los estándares de definición de casos de uso y de casos de prueba de Softtek®.
- Analizar técnicas de generación de casos de prueba a partir de casos de uso para elegir la que mejor se adapte a las características de la empresa.

- Analizar las técnicas para la selección de casos de prueba basadas en cobertura y/o tiempo disponible para pruebas.
- Plantear la arquitectura del generador.
- Implementar el generador de casos de prueba funcionales.
- Implementar el generador de casos de prueba integrales.
- Probar, mediante un caso de estudio, ambos generadores.

1.5 Justificación

En vista de las presentes problemáticas, en el expuesto trabajo se muestra un enfoque capaz de generar casos de prueba de forma automática partiendo de un lenguaje de definición de requisitos funcionales proporcionado por la empresa, esto significa una ventaja con respecto a otras formas de partida para crear casos de prueba, ya que este lenguaje no es una definición formal pero tampoco se trata de un CNL, más bien es una representación de requisitos capaz de expresar pragmáticamente los pasos necesarios para generar una función específica.

Además, para garantizar el reconocimiento de la ruta crítica dentro de la cobertura de las pruebas, es necesario crear un grafo que marque los casos necesarios para decir que un software es funcional, y la forma en que este grafo se definirá es mediante la combinación de algoritmos y técnicas que limiten el número mínimo de pruebas necesarias capaces de probar la mayor parte de una función.

Los resultados de este trabajo beneficiarán la creación automática de casos de prueba, ya que reduciría el tiempo de producción de estos, se maximizarían los criterios de cobertura incluyendo el cumplimiento de rutas críticas y se disminuiría el costo de prueba.

Capítulo 2 Estado de la práctica

La generación de casos de prueba de forma automática es un tema que tomo importancia debido a la necesidad creciente de asegurar que el producto de software final cumpliera con las definiciones de requisitos iniciales, por lo que a continuación, se describe una serie de trabajos relacionados a este tema y una conclusión con base a estos.

2.1 Trabajos relacionados

Muchas investigaciones se centran en cómo alcanzar los niveles de seguridad requeridos para los sistemas críticos. Un enfoque para enfrentar este escenario es el Modelo Basado en Pruebas (MBT), el cual obtiene casos y datos de prueba a partir de un Modelo de Especificación, sin embargo, este está sujeto a fallas debido a que se realiza en notaciones de Lenguaje Natural (NL). En [15] se propuso $NAT2TEST_{SCR}$ como una estrategia para la generación automática de casos de prueba a partir de requisitos escritos en lenguaje natural, usando como regulador de ambigüedad textual un lenguaje natural controlado (CNL), llamado SysReq-CNL. El resultado derivó en una serie de traductores capaces de validar sintácticamente los requisitos de entrada y los significados semánticos para la generación de casos de prueba no redundantes hasta 30 minutos más rápido que los desarrolladores.

Tiwari et al., [16] determinaron que el desarrollo de software implica la identificación de escenarios a partir de la especificación de requisitos funcionales, y que los medios más populares para extraer estos son los casos de uso, los cuales se definen en Lenguaje Natural (NL), por lo que se propuso la creación de un enfoque de transformación sistemática que esté preparado para extraer escenarios de casos de uso a partir de la especificación de requisitos textuales, con la capacidad de utilizar un analizador textual NL, para identificar las Partes Del Habla (POS) etiquetas, Dependencias de Tipo (TD) y Roles Semánticos (SRL), utilizando una herramienta prototipo llamada Text2UseCase. El uso del enfoque propuesto permitió la generación de casos de uso más completos, consistentes, correctos y no redundantes en comparación con los casos de uso desarrollados manualmente.

Como se mencionó antes, la definición de casos de prueba es propensa a errores, y las investigaciones que se dedican a la generación automática de estos requiere de una intervención manual significativa, por lo que en [17] se propuso la Generación de Pruebas del Modelado de Casos de Uso (UMGT), un enfoque capaz de extraer información de los casos de uso a partir del Procesamiento de Lenguaje Natural (NLP). UMGT necesita restringir los casos de uso, por lo que se utilizó un Modelado de Casos de Uso Restringido (RUCM), que básicamente es una plantilla con reglas y restricciones que ayudan a reducir la imprecisión de estos. El enfoque demostró funcionar favorablemente en un caso de estudio de dominio automotriz con especificaciones de casos de uso para un sistema de sensores automotrices.

Para Yazdani et al., [18] no basta con solo generar los casos de prueba, también es necesario que estos sean un conjunto mínimo capaz de probar la suficiencia del software en escenarios reales del negocio, por lo que se definió una técnica para generar casos de prueba a partir de especificaciones de requisitos utilizando modelos de procesos de negocios, con el fin de obtener rutas de ejecución derivadas del proceso de negocio real. Se genera un BPMN (*Business Process Model and Notation*, Modelo y Notación de Procesos de Negocio) para representar el modelo de proceso de negocio, posteriormente, este se transforma en un grafo de estado del cual se extraen los casos de prueba utilizando una herramienta llamada Spec Explorer. El método propuesto ahorra tiempo y reduce el costo de la generación de cada caso de prueba, además aumenta la confiabilidad del software, ya que estos son más compatibles con los requisitos de negocio, sin embargo, el enfoque solo funciona adecuadamente bajo un proceso estructurado, por lo que se está trabajando para extender su alcance a casos contrarios.

Los casos de prueba también crecen conforme lo hace el sistema e incluso se actualizan simultáneamente con la funcionalidad unitaria a la que pertenecen, sin embargo, en una encuesta realizada en [19] a 212 desarrolladores de código abierto e industrial, se demostró que el 89.15% de estos mantiene la información de actualización de los casos de prueba en los comentarios dentro del código en lugar de hacerlo en la documentación. Para combatir esta problemática se desarrolló un enfoque llamado UnitTestScribe, el cual genera documentación automáticamente en NL a partir de la detección de métodos focales, afirmaciones y dependencias de datos en el código que corresponde a la función unitaria del caso de prueba. Además, se realizó a) un estudio

empírico para entender la manera en que los desarrolladores comentan y actualizan casos de prueba unitaria, y b) una encuesta para conocer la perspectiva y práctica con respecto a la documentación de pruebas unitarias. El enfoque se evaluó con la misma muestra de desarrolladores para evaluar su exhaustividad, concisión y expresividad, mostrando que las descripciones generadas por este no omitieron información importante (en un 87% y 96%), que no contienen información redundante (en un 87% y 93%) y que los resultados fueron legibles y comprensibles (en un 84% y 88%).

Las pruebas basadas en modelos (MBT) son una alternativa para la creación de casos de prueba, los cuales derivan de modelos de comportamiento descritos en lenguajes formales, por lo que la importancia de dichos lenguajes radica en su facultad de facilitar la identificación de escenarios que serán descompuestos para obtener los casos de prueba necesarios, sin embargo, dicho proceso es costoso y tardado, por lo que Sarmiento et al. [20] plantearon el uso del lenguaje natural para la definición de dichos escenarios. El resultado es C&L (*Cenários e Léxicos*, Escenarios y Léxicos), una herramienta que transforma automáticamente las descripciones de los requisitos en diagramas de actividad, para que más tarde este sea procesado en un grafo dirigido a partir del cual se extraen los casos de prueba, para esto se utilizó el lenguaje LEL (*Language Extended Lexicon*, Léxico Extendido de Lenguaje) para describir el comportamiento del software y así poder ver la aplicación como una serie de escenarios supervisados. Los resultados demostraron que el lenguaje de escenarios describe al software como un conjunto de módulos descritos en escenarios, por lo que al aplicar las pruebas generadas por C&L en escenarios de la aplicación, se está verificando la funcionalidad del software con respecto a la descripción de comportamiento de este.

Los sistemas de software para vehículos de hoy en día incluyen sensores y funciones avanzadas que dan como resultado un aumento en el número de combinaciones posibles para situaciones que necesitan ser probadas, por lo que es necesario describir requisitos de alto nivel industrial del sistema en lenguaje natural. En [21] se propuso un proceso para transformar requisitos de alto nivel en casos de prueba de manera sistemática. El proceso de traducción se logró gracias al enfoque SAGA, que se basa en el concepto de aseveraciones vigiladas apoyado por una cadena de herramientas que consisten en un editor de casos de prueba interactivo, así como el

lenguaje de descripción T-EARS (*Easy Approach to Requirement Specification,* Enfoque fácil para la especificación de requisitos). Se demostró la utilidad de traducir requisitos en NL para sistemas vehiculares a casos de prueba pasivos expresados en forma de aseveraciones vigiladas por SAGA, para apoyar a la industria a adoptar este enfoque y el uso de pruebas pasivas.

Por otra parte, generar casos de prueba que tomen en cuenta la variabilidad del entorno donde se desarrolle el sistema resulta aún más complicado, esto sucede frecuentemente con los denominados DSPL (*Dynamic Software Product Line*, Línea de Productos de Software Dinámico), un caso de este tipo son los sistemas móviles, ya que no es lo mismo descargar un video con 3G (tercera generación de transmisión de voz y datos a través de telefonía móvil) en lugar de tener 4G (cuarta generación), por lo tanto, lo ideal es que se desarrolle un caso de prueba para cada tipo de entorno aunque la entrada o la salida arrojen el mismo resultado. Araujo et al. [22] plantearon un enfoque para generar automáticamente casos de prueba para una familia de sistemas DSPL tomando en cuenta la información del contexto definido en CNL para expresar de forma textual las descripciones de cada entrada. Las pruebas realizadas demostraron que el proceso de generación de casos de prueba a partir del enfoque se realizó un 40% más rápido que cuando no se usa este.

En [23] se definió que además de generar pruebas unitarias, también es importante poder comprenderlas, ya que esta práctica se complica si la prueba se hizo hace algún tiempo, si la hizo otra persona o incluso si se generó de forma automática, por lo que se realizó un enfoque para mejorar la legibilidad de estas pruebas. Dicha técnica aumento la legibilidad en más de la mitad de los casos unitarios generados automáticamente y generó pruebas optimizadas alternativas un 2% más legible al promedio, además, los desarrolladores demostraron entender un 14% más rápido los cambios de precisión realizados en los casos de prueba generados por esta técnica. Todo esto se logró gracias a la implementación de EvoSuite, una herramienta que genera automáticamente pruebas unitarias para el software creado en Java.

Cada vez es más frecuente la representación formal para la generación de casos de prueba, sin embargo, estos procedimientos son costosos y requieren de especialistas con un fuerte conocimiento en el modelado formal, por lo que en [24] se expuso la generación de escenarios de prueba tomando como entrada una especificación de requisitos definida en lenguaje natural

restringido (RNL), para lograrlo, se hizo uso de las herramientas disponibles de redes de Petri, para representar los escenarios RNL como topologías distribuidas, paralelas o concurrentes. Los resultados mostraron que fue posible generar escenarios de prueba con características de concurrencia más completos y consistentes.

Como se mencionó anteriormente, la utilización de casos de uso para la generación de casos de prueba ayuda a relacionar directamente estos con los requerimientos originales de la aplicación, sin embargo, la tendencia actual de generar componentes de software propicia la creación de pruebas por cada componente elaborado, por lo que en Lipka et al. [25] se desarrolló un método para la generación semiautomática de escenarios de prueba para componentes de software a partir de casos de uso escritos en NL. Gracias a la herramienta SimCo, se derivan los escenarios del caso de uso para la generación de secuencias de invocaciones. Como resultado, se obtuvieron pruebas basadas en especificaciones originales comprensibles para los no programadores, garantizando la funcionalidad de un componente frente al cliente.

En [26] se analizó que la comprensión del manual de ejecución de casos de prueba es también un problema importante en el dominio de quienes escriben estos casos, debido a que el proceso de escritura de los requisitos y de los casos de prueba se hace de forma manual, lo que origina ambigüedad en la cobertura de dichos casos, por lo que se propuso un enfoque que permite definir diferentes criterios de análisis de pruebas que se hayan implementado anteriormente. Para la definición de requisitos, se usa NL y RTCM (*Restricted Use Case Modeling*, Modelado de Casos de Uso Restringido), que fue desarrollado a partir de TCS (*Test Case Specifications*, Especificación de Casos de Prueba) para generar casos de prueba automáticamente aplicando criterios de cobertura. Al final se obtuvo una herramienta llamada Toucan4Test, que implementa criterios de cobertura de estructura en las especificaciones RTCM, logrando generar TCS y casos de prueba de forma sistemática y automática a partir de las especificaciones de RUCM.

Los requisitos del software a menudo cambian durante su proceso de desarrollo, por lo que los casos de prueba relacionados con el sistema se deben reconstruir y ejecutar nuevamente, por lo que la generación y ejecución automática de tales casos ayuda a ahorrar tiempo y esfuerzo, reduciendo errores y fallas en la actividad de prueba. Entre los procesos de ejecución de pruebas automatizados, se encuentra el proceso basado en palabras, que es un marco dirigido por claves

KDTF (*Keyword-Driven Testing Framework*, Marco de prueba dirigido por palabras clave), que divide el caso de prueba en cuatro partes: pasos de prueba, objetos de cada caso, acciones dentro de los objetos y datos. En Hue et al. [27] se propuso un método para generar automáticamente los casos de prueba del sistema a partir de casos de uso que son capturados por modelos USL (*Use Case Specification Language*, Lenguaje de Especificación de Casos de Uso). Los casos de prueba generados están especificados con precisión gracias al modelo TCSL (*Test Case Specification Language*, Lenguaje de Especificación de Casos de Prueba) y contienen la información necesaria para un KDTF capaz de ejecutar casos de prueba directamente. Las principales contribuciones son: a) el método USLTG (*Test Generation from a USL model*, Generación de Pruebas a partir del modelo USL) para la generación automática de casos de prueba del sistema a partir de casos de uso, b) tres algoritmos para transformar los casos de uso en un modelo TCSL para capturar los casos de prueba del sistema generados y c) una herramienta de soporte para utilizar el método USLTG.

Tradicionalmente, la generación automática de casos de prueba se lleva a cabo desde lenguajes formales y como se describió anteriormente, esto exige esfuerzo adicional en el proceso de prueba, por lo que en [28] se propuso la descripción de casos de uso basados en estados por medio de CNL, como una mejor opción para optimizar la generación de casos de prueba. El enfoque demostró ser capaz de especificar tanto el flujo de control como los aspectos de los datos. El flujo de control se captura a través de la estructura tabular de cada caso de uso, además, está integrado en la especificación hecha en CNL para manipular entradas, salidas y variables de estado. Tras traducir los casos de uso, se crean automáticamente los casos de prueba necesarios. Se desarrolló una herramienta totalmente integrada a FDR (*Failures Divergence Refinement*, Fallos de Refinamiento de Divergencia), que es otra herramienta para el refinamiento de procesos descritos en CSP (*Communication Sequential Processes*, Procesos Secuenciales de Comunicación) para la edición de casos de uso y la generación de casos de prueba.

Ansari et al. [29] también desarrollaron un sistema apto para la generación de casos de prueba a partir de requisitos funcionales procesados en NLP. El sistema propuesto construye dichos casos basándose en palabras clave de los requisitos funcionales. El objetivo alcanzado fue

reducir el esfuerzo y el tiempo consumido por el probador de software gracias a la generación de casos de prueba automáticamente. El resultado fue el análisis automático del documento de requisitos en función de palabras clave que sirvieran para extraer los casos de prueba, además se garantizó la cobertura máxima de todos los requisitos especificados en SRS (*Software Requirement Specification,* Especificación de Requerimientos de Software) para que la implementación de las pruebas alcanzara la confiabilidad del sistema con el cliente.

2.2 Análisis comparativo

A continuación, en la Tabla 2.1 se muestra un análisis comparativo obtenido de la información de los artículos anteriormente descritos.

Tabla 2.1 Comparación de Artículos

Artículo	Problema	Contribución	Tecnologías	Resultado	Estado
Carvalho et al. [15]	Los casos de prueba para alcanzar la seguridad en sistemas críticos están propensos a fallas debido a que se definen en NL.	$NAT2TEST_{SCR}$ como una estrategia para la generación automática de casos de prueba a partir de requisitos escritos en NL.	SysReq-CNL, como regulador de ambigüedad textual.	Traductores capaces de validar sintácticament e-te los requisitos de entrada y los significados semánticos para la generación de casos de prueba no redundantes.	Finalizado.
Tiwari et al. [16]	Desarrollar software implica identificar escenarios a partir de casos de uso definidos en NL.	Un enfoque de transformación sistemática que extraiga escenarios de casos de uso a partir de la especificación de requisitos textuales.	Analizador textual Text2UseCa-se.	Generación de casos de uso más completos, consistentes, correctos y no redundantes.	Finalizado.

Artículo	Problema	Contribución	Tecnologías	Resultado	Estado
Wang et al. [17]	La generación automática de casos de prueba requiere de una intervención manual significativa.	UMGT, un enfoque que extrae información de los casos de uso, a partir del Procesamiento -to de NL para generar casos de prueba.	RUCM, plantillas con reglas y restricciones que ayudan a reducir la imprecisión de los casos de uso.	Funcionamiento-to óptimo en un sistema de sensores automotrices.	Finalizado.
Yazdani et al. [18]	Los casos de prueba deben ser un conjunto mínimo capaz de probar la suficiencia del software en escenarios de procesos de negocio reales.	Definición de una técnica para generar casos de prueba a partir de especificaciones de requisitos utilizando modelos de procesos de negocios.	BPMN para la representación del modelo de negocios. Spec Explorer para la extracción de casos de prueba a partir de un grafo.	Ahorro de tiempo y reducción de costos en la generación de casos de prueba, y un aumento en la confiabilidad del software.	Proceso de Mejora.
Li et al. [19]	Los desarrolladores de software hacen comentarios de las actualizaciones de los casos de prueba dentro del código en lugar de hacerlo en la documentación.	UnitTestScribe, un enfoque que genera documentación en NL de la detección de métodos focales, afirmaciones y dependencias de datos en el código.	No se menciona.	Descripciones de casos de prueba con información importante y no redundante, con resultados legibles y comprensibles.	Finalizado.
Sarmiento et al. [20]	Una forma de generar casos de prueba es haciendo uso de los escenarios de	C&L es una herramienta que genera casos de prueba a partir de la definición	LEL, para describir él comportamiento del software.	Definición de pruebas de software específicas de un módulo más apegadas a la	Proceso de Mejora.

Artículo	Problema	Contribución	Tecnologías	Resultado	Estado
	comporta-miento, sin embargo, estos se definen en métodos formales por lo que son costosos y tardados.	de escenarios en NL.		descripción de su comportamien-to.	
Daniel et al. [21]	Los sistemas de software para vehículos usan descripciones de requisitos de alto nivel, por lo que estos necesitan ser traducidos para generar casos de prueba pasivos.	Proceso de transformación de requisitos de alto nivel en casos de prueba pasivos de manera sistemática.	Enfoque SAGA y lenguaje T-EARS.	Se demostró la utilidad de traducir requisitos en NL para sistemas vehiculares a casos de prueba pasivos expresados en forma de aseveraciones vigiladas.	Procesos de Mejora.
Araujo et al. [22]	Generar casos de prueba para DSLP es complicado debido a la variabilidad del entorno de la función del sistema.	Un enfoque para la generación automática de casos de prueba para sistemas DSPL, a partir de las descripciones textuales definidas en CNL de cada entrada.	No se menciona.	Creación de casos de prueba 40% más rápido.	Finalizado.
Daka et al. [23]	La comprensión de pruebas	Una técnica para aumentar la legibilidad	EvoSuite, una herramienta para la	Pruebas unitarias un 2% más legibles	Finalizado.

Artículo	Problema	Contribución	Tecnologías	Resultado	Estado
	unitarias es difícil, sobre todo si estos casos fueron escritos hace tiempo, por otra persona o se generaron automática-mente.	de casos prueba unitarios.	generación automática de pruebas para software creado en Java.	que el promedio y un aumento de la comprensión de estos casos en un 14%.	
Sarmien-to et al. [24]	La generación de casos de prueba a partir de representacio nes formales es costosa y requiere de la intervención de especialistas.	Un proceso para la generación de escenarios de prueba tomando como entrada una especificación de requisitos definida en RNL.	Las herramientas del enfoque de redes de Petri.	Escenarios de prueba con características de concurrencia más completos y consistentes.	Finalizado.
Lipka et al. [25]	Es necesario crear casos de prueba para cada componente de software en una aplicación.	Un método para la generación semiautomáti-ca de escenarios de prueba para componentes de software.	SimCo, herramienta diseñada para la simulación de pruebas de componentes de software.	Pruebas apegadas a los requisitos originales y comprensibles para los no programado-res.	Finalizado.
Zhang et al. [26]	El proceso de escritura manual de los requisitos y los casos de prueba introduce ambigüedad en el manual de ejecución de estos casos.	Toucan4Test, una herramienta que implementa criterios de cobertura de estructura en las especificacio-nes RTCM.	RTCM, para especificar requisitos de software.	Generación de TCS y casos de prueba de forma sistemática y automática a partir de las especificacio-nes de RUCM.	Finalizado.

Artículo	Problema	Contribución	Tecnologías	Resultado	Estado
Hue et al. [27]	Ante el cambio continuo de los requisitos funcionales, es necesario crear casos de prueba de forma automática.	USLTG, un método para la generación automática de casos de prueba a partir de casos de uso.	KDTF, marco de pruebas dirigido basado en palabras clave. USL, lenguaje de especificación de casos de uso. TCSL, generador de pruebas a partir de USL.	Tres algoritmos para transformar los casos de uso en un modelo TCSL y una herramienta de soporte para utilizar el método USLTG.	Finalizado.
Nogueira et al. [28]	La generación de casos de prueba a partir de lenguajes formales es difícil y costosa.	Una herramienta integrada a FDR capaz de traducir casos de uso a CNL para generar a partir de estos casos de prueba.	FDR, una herramienta para verificar el refinamiento de procesos CSP.	Se consiguió especificar el flujo de control y los aspectos de los datos en los casos de prueba.	Finalizado.
Ansari et al. [29]	La generación de casos de prueba le toma demasiado tiempo y esfuerzo al probador de software, aumentando el costo de la producción de estos.	Un sistema apto para la generación de casos de prueba a partir de requisitos funcionales procesados en NLP.	No se menciona.	Análisis automático del documento de requisitos y reducción de tiempo, esfuerzo y costos en la generación de casos de prueba.	Finalizado.

Después de analizar la información de los artículos expuestos, se concluye que la generación automática de casos de prueba es de mucha ayuda en el desarrollo de software, ya que reduce tiempo, costos y esfuerzos. Esta práctica en la mayoría de los casos se lleva a cabo

transformando las especificaciones de requisitos definidos en NL a CNL o a cualquier otro tipo de lenguaje específico siempre y cuando se cumpla con el objetivo de eliminar la ambigüedad y el error del NL, esto también se aplica si se ocupan Casos de Uso. Las representaciones gráficas no son tan utilizadas, ya que de estas se tienen que derivar grafos dirigidos y finalmente están las definiciones en métodos formales, que si bien aseguran suprimir los errores en el origen que generará el caso de prueba, es muy difícil manejarlos debido a que se necesita un amplio dominio para su manejo, por lo que es necesario la intervención de especialistas, lo cual es costoso y tardado.

Por lo que la mejor forma de generar casos de prueba automáticamente es mediante casos de uso o descripciones de requisitos definidas en CNL o cualquier otro lenguaje específico, además, para cubrir aspectos de cobertura lo más idóneo es la utilización de grafos, ya que, aunque estos sean difíciles de implementar, demostraron que son más eficaces por los algoritmos matemáticos que usan para determinar los grados de cobertura. Por lo antes expuesto, el proyecto de tesis propone la combinación de estas técnicas para la generación de casos de prueba, a través de un enfoque novedoso y más eficiente, capaz de cubrir aspectos de cobertura y cumplimiento de requisitos funcionales.

2.3 Propuesta de solución

A continuación, se describirá el IDE (*Integrated Development Environment*, Ambiente de desarrollo integrado), marco de trabajo y lenguaje de programación que fueron seleccionados para la realización de este trabajo de tesis por la empresa en cuestión.

2.3.1 Eclipse

Eclipse es una plataforma de desarrollo, de código abierto, basada en Java. Por sí misma, es simplemente un marco de trabajo y un conjunto de servicios para la construcción del entorno de desarrollo de los componentes de entrada. Afortunadamente, Eclipse tiene un conjunto de complementos, incluidas las Herramientas de Desarrollo de Java (JDT).

También incluye el Entorno de Desarrollo de Complementos (PDE), que permite construir herramientas que se integran sin dificultades con el entorno de desarrollo [30].

2.3.2 Xtext

Xtext es un marco de trabajo para el desarrollo de lenguajes de programación y lenguajes específicos de dominio. Con Xtext es posible definir un lenguaje usando una serie de expresiones gramaticales. Como resultado, se obtiene una infraestructura completa que incluye un analizador, vinculador, tipógrafo y compilador, así como un soporte de edición para el entorno de desarrollo Eclipse.

Para especificar un lenguaje, se tiene que escribir una gramática en el lenguaje de las expresiones de Xtext, por lo que ofrece una serie de características que facilitan la programación como [31]:

- Resaltado de sintaxis.
- Autocompletado.
- Validación rápida de la sintaxis.
- Integración avanzada con Java.
- Integración con otras herramientas de Eclipse

2.3.3 Xtend

Xtend es un lenguaje de programación estático que se traduce en un código fuente de Java comprensible. De forma sintáctica y semántica, Xtend tiene sus raíces en el lenguaje de programación Java, pero mejora en los siguientes aspectos:

- Métodos de extensión: mejora los tipos cerrados con una nueva funcionalidad.
- Expresiones Lambda: sintaxis concisa para literales de funciones anónimas.
- Sobrecarga de operadores: para tener bibliotecas más extensas.
- Expresiones de conmutador potentes: conmutación basada en tipos con conversiones implícitas.
- Envío múltiple: invocación de métodos polimórficos.
- Expresiones de plantilla: manejo inteligente de espacios en blanco.
- Sin declaraciones: todo es una expresión.
- Propiedades: abreviaturas para acceder y definir captadores y definidores.

- Inferencia de tipos: ya no es necesario que escriba las firmas de tipos de datos.
- Soporte completo para los genéricos de Java, incluidas todas las reglas de conformidad y conversión.

A diferencia de otros lenguajes JVM, Xtend no tiene problemas de interoperabilidad con Java, todo lo que escribe interactúa con éste exactamente como se espera. Al mismo tiempo, Xtend es mucho más conciso, legible y expresivo. La pequeña biblioteca de Xtend es solo una capa delgada que proporciona utilidades y extensiones útiles sobre el JDK (*Java Development Kit*, Kit de desarrollo de Java).

Además, también es posible llamar a los métodos Xtend desde Java de una manera completamente transparente [32].

2.3.4 Neo4J

Neo4j es una base de datos orientada a grafos escrita en Java, es decir, la información se almacena de forma relacionada formando un grafo dirigido entre los nodos y las relaciones entre ellos. Se integra perfectamente con múltiples lenguajes como Java, PHP, Ruby, .Net, Python, Node, Scala, etc. La base de datos está embebida en un servidor Jetty (servidor HTTP y contenedor de Servlets 100% basado y escrito en Java), además, permite modelar todo tipo de escenarios definidos por entidades y relaciones entre sí, como una red social, motores de recomendaciones en tiempo real, etc.

Los beneficios de utilizar una base de datos de estas características son:

- Flexibilidad: los datos no tienen que mantener una estructura rígida, es posible que los nodos sean de tipos diferentes y se añadan con facilidad atributos adicionales.
- Búsqueda: Es posible hacer búsquedas más rápidas basadas en las relaciones establecidas entre los nodos, por ejemplo: *"¿cuándo se han hecho amigas dos personas?"*.
- Indexación: las bases de datos de grafo se indexan de forma natural por sus relaciones, proporcionando un acceso más rápido en comparación a cómo resuelve una base de datos relacional.

Neo4j permite acceder a sus datos de diversas formas y con distintos lenguajes de consulta, por ejemplo, desde una consola de texto, un entorno web gráfico o mediante una API. Respecto a sus lenguajes de consulta, se destaca Cypher, que es un lenguaje declarativo que permite consultar y manipular grafos, y Gremlin, que es un lenguaje específico de dominio para la gestión de grafos.

La herramienta identifica un conjunto de datos sobre los que aplicar una operación, se realiza lo que se denomina "navegación por el grafo" (la acción de navegar por el grafo a partir de uno o varios puntos de inicio), lo que permite obtener resultados deseados tales como obtener los elementos sobre los cuales realizar alguna operación en específico [33].

2.3.5 Algoritmos para el recorrido de grafos

Existen diversos algoritmos para recorrer grafos, a continuación, se describen algunos relacionados tanto con la cobertura como con la identificación de las rutas óptimas.

2.3.5.1 Algoritmo de Dijkstra

El algoritmo de Dijkstra, también llamado Algoritmo de Caminos Mínimos, es un algoritmo para la determinación del camino más corto, dado un vértice origen, hacia el resto de los vértices en un grafo que tiene pesos en cada arista. Su nombre alude a Edsger Dijkstra, científico de la computación de los Países Bajos que lo describió por primera vez en 1959.

La idea subyacente en este algoritmo consiste en ir explorando todos los caminos que parten del vértice origen y que llevan a todos los demás vértices; cuando se obtiene el camino más corto desde el vértice origen hasta el resto de los vértices que componen el grafo, el algoritmo se detiene. Se trata de una especialización de la búsqueda de costo uniforme y como tal, no funciona en grafos con aristas de costo negativo (al elegir siempre el nodo con distancia menor, es posible que se excluyan de la búsqueda nodos que en próximas iteraciones bajarían el costo general del camino al pasar por una arista con costo negativo).

Gracias a él, es posible resolver grafos con muchos nodos, lo que sería muy complicado resolver sin dicho algoritmo, encontrando así la ruta más cortas entre un origen y todos los destinos en una red [3].

2.3.5.2 Algoritmo de Complejidad Ciclomática de McCabe

Desarrollado en 1976 por Thomas J. McCabe, refleja directamente el número de caminos independientes que un elemento de software puede tomar durante su ejecución.

La Complejidad Ciclomática de McCabe V (G) se genera a partir de un grafo de flujo de control (G) mediante la fórmula:

$$V(G) = E - N + 2$$

El valor E es el número de bordes en el grafo de flujo de control y N es el número de nodos. Esta fórmula es aplicable a los grafos de flujo donde no hay componentes desconectados. Uno de sus usos es proporcionar una aproximación del número de casos de prueba necesarios para cumplir con la cobertura de un módulo de código especificado.

Si la capacidad de prueba de una pieza de software se define en términos de la cantidad de casos de prueba necesarios para probarlo adecuadamente, entonces la complejidad ciclomática de McCabe proporciona una aproximación de la capacidad de prueba de un módulo.

El probador usa el valor de V (G) junto con los datos de proyectos anteriores para aproximar el tiempo de prueba y los recursos necesarios para probar un módulo de software. Además, el valor de la complejidad ciclomática y el grafo de flujo de control le dan al probador otra herramienta para desarrollar casos de prueba de caja blanca utilizando el concepto de ruta.

Una ruta es una secuencia de nodos de flujo de control que generalmente comienza desde el nodo de entrada de un grafo hasta el nodo de salida. Una ruta atraviesa un segmento dado del grafo de flujo de control una o más veces. Por lo general se designa una ruta por la secuencia de nodos que abarca.

La complejidad ciclomática es la medida del número de las rutas "independientes" en el grafo. Una ruta independiente es un tipo especial de ruta en el grafo de flujo. La obtención de un conjunto de rutas independientes utilizando un grafo de flujo ayuda a un probador a identificar las funciones de flujo de control en el código y establecer objetivos de cobertura.

Un probador identifica un conjunto de rutas independientes para la unidad de software comenzando con una ruta simple en el grafo de flujo y agregando iterativamente nuevas rutas

al conjunto agregando nuevos bordes en cada iteración hasta que no haya más bordes nuevos que agregar.

Las rutas independientes se definen como cualquier ruta nueva a través del grafo que introduce un borde nuevo que no se ha atravesado antes de que se defina la ruta. Un conjunto de rutas independientes para un grafo a veces se denomina conjunto de bases. Para la mayoría de los módulos de software es posible derivar una serie de conjuntos de bases.

El número de rutas independientes en un conjunto de bases es igual a la complejidad ciclomática del grafo. Si se preparan casos de prueba de caja blanca para que las entradas causen la ejecución de todas estas rutas, se dice que se alcanzó la cobertura completa de la declaración y la decisión para el módulo.

Los evaluadores deben ser conscientes de que, si bien la identificación de las rutas independientes y el cálculo de la complejidad ciclomática en un módulo de código estructurado proporciona un apoyo útil para alcanzar los objetivos de cobertura de decisiones, en algunos casos, la cantidad de rutas independientes en el conjunto de bases lleva a una sobre aproximación de la cantidad de casos de prueba necesarios para la cobertura de decisión [3].

En la tabla 2.2 se muestra una comparativa de los algoritmos mencionados anteriormente.

Tabla 2.2 Algoritmos para Grafos

Algoritmo	Ventajas	Desventajas
Algoritmo de Dijkstra	• Permite la definición de una ruta específica. • Resuelve problemas con un número extenso de nodos. • Útil para redes sin nodos libres.	• No calcula otros caminos más que el idóneo.
Complejidad Ciclomática de McCabe	• Sirve para el cálculo de caminos que determinen cobertura. • Aplicable a los gráficos de flujo donde no hay	• En algunos casos puede darse una sobre aproximación de casos necesarios.

Algoritmo	Ventajas	Desventajas
	componentes desconectados. • Permite el cálculo de rutas independientes. • Sirve como herramienta en la generación de pruebas de Caja Blanca.	

2.3.6 Metodologías de desarrollo de software

A continuación, se describe la metodología a usar en la solución de este proyecto de tesis.

2.3.6.1 Metodología XP

XP (*Extreme Programming*, Programación Extrema) es una de las llamadas metodologías ágiles de desarrollo de software, por lo que asegura un mayor control sobre el proyecto, y una implementación más efectiva y eficiente. El ciclo de vida de XP se enfatiza en el carácter interactivo e incremental del desarrollo, a continuación, en la Figura 2.1 se ilustran las fases de su ciclo de desarrollo y las prácticas y herramientas utilizadas en cada una.

Figura 2.1 Fases del ciclo de desarrollo de XP

En la fase de **Planeación** se comienza escuchando (que es la actividad para recabar requerimientos que permiten a los miembros técnicos del equipo XP entender el contexto del negocio para el software, comprender las características de las funcionalidades y adquirir la sensibilidad de la salidas esperadas del producto de software) para posteriormente crear historias (también llamadas historias del usuario) donde se describen las salidas necesarias, características y funcionalidades del software que se van a elaborar.

Cada historia (similar a los casos de usos) se le asigna una prioridad con base en el valor general de su función para el negocio. Después, los miembros del equipo XP evalúan cada historia y le asignan un costo medido en semanas de desarrollo. Si se estima que la historia requiere más de tres semanas de desarrollo, se pide al cliente que la descomponga en historias más pequeñas y de nuevo se asigna un valor y costo. Es importante observar que en cualquier momento es posible escribir nuevas historias.

Posteriormente, se inicia la fase de **Diseño,** donde se sigue rigurosamente el principio MS (mantenlo sencillo). Un diseño sencillo siempre se prefiere sobre una representación más compleja. Además, el diseño guía la implementación de una historia conforme se escribe: Nada más y nada menos. Se desalienta el diseño de funcionalidad adicional.

Después de hecho el trabajo de diseño preliminar, el equipo no inicia la codificación, sino que desarrolla una serie de pruebas unitarias a cada una de las historias que se van a incluir en la entrega del software. Una vez creada la prueba unitaria, el desarrollador está capacitado para centrarse en lo que debe implementarse para pasar la prueba. Una vez que el código está terminado, se le aplica de inmediato una prueba unitaria, con lo que se obtiene retroalimentación instantánea para los desarrolladores, este proceso se delimita en la etapa de **Codificación.**

La creación de pruebas unitarias antes de comenzada la codificación es un elemento clave en la fase de **Prueba**, ya que dichas pruebas deben implementarse con el uso de una estructura que permita automatizarlas (de modo que se ejecuten en repetidas ocasiones fácilmente). Esto estimula una estrategia de pruebas de regresión siempre que se modifique el código (lo que ocurre con frecuencia, dada la filosofía del rediseño en XP) [34].

Capítulo 3 Aplicación de la Metodología

Es importante puntualizar que ya se encontraba realizado un prototipo del generador de casos de prueba funcionales e integrales, por lo que a continuación, en la Figura 3.1 se muestra su funcionamiento, ya que a partir de este prototipo se comenzó la construcción del proyecto de tesis.

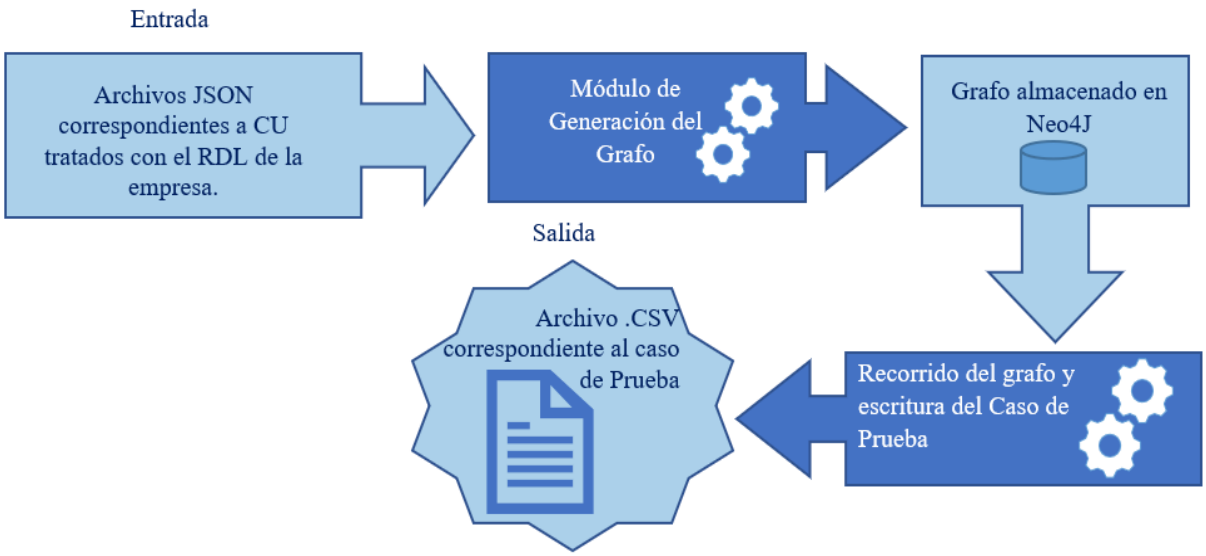


Figura 3.1 Funcionamiento del prototipo del proyecto

Como se observa en la figura anterior, lo que recibe como **entrada** esta solución, son archivos de tipo JSON resultantes de aplicar el RDL de la empresa en Casos de Uso; si se desea generar un caso de prueba funcional, se agrega el archivo JSON correspondiente al Caso de Uso que se desea probar, mientras que para los casos de prueba integrales (que en este caso es igual a varios Casos de Uso relacionados) es necesario agregar tantos archivos JSON como Casos de Uso.

Luego de las validaciones correspondientes sobre la entrada, el **Módulo de Generación del Grafo,** se encarga de recorrer dicho inicio con el objetivo de armar el dígrafo equivalente a todos los pasos posibles del Caso de Prueba, dicha representación se almacena en la anteriormente descrita base de datos orientada a grafos **Neo4j** y, por último, una vez completado

el proceso de almacenamiento, se recorrer el grafo para generar el archivo .CSV correspondiente al Caso de Prueba.

Los puntos favorables de esta primera versión son:

- La lectura de los archivos JSON definidos en el proceso de entrada.
- La creación de un grafo dirigido a partir de las características de los archivos JSON.
- El almacenado del grafo en Neo4J.

Las características que es necesario mejorar:

- El recorrido del grafo: ya que esta primera versión solo toma en cuenta los nodos correspondientes al escenario principal del Caso de Uso, aunque ya existan los nodos de los demás escenarios.
- Las características y el tipo del archivo de salida: ya que la empresa desea un obtener archivos de tipo .XLS con un formato en particular.

3.1 Análisis y Planificación

Antes de comenzar con el desarrollo de un software, es necesario entender a la perfección la problemática y el contexto que lo rodea, por lo que hay que destacar que la empresa define como Escenario de Prueba a la secuencia de pasos obtenida luego de recorrer el grafo desde un punto de inicio hasta un fin, mientras que el Caso de Prueba es el llenado de un Escenario de Prueba especifico con cierta combinación de datos, además de esta aclaración, en breve se adentrará en la descripción de los resultados de la fase correspondiente al Análisis y Planificación de la metodología XP.

3.1.1 Diseño de la solución

Luego del estudio de las necesidades de la empresa, y del funcionamiento de la primera versión del proyecto, se consiguió representar en la Figura 3.2 el proceso que sigue el Generador de Pruebas Integrales y Funcionales.

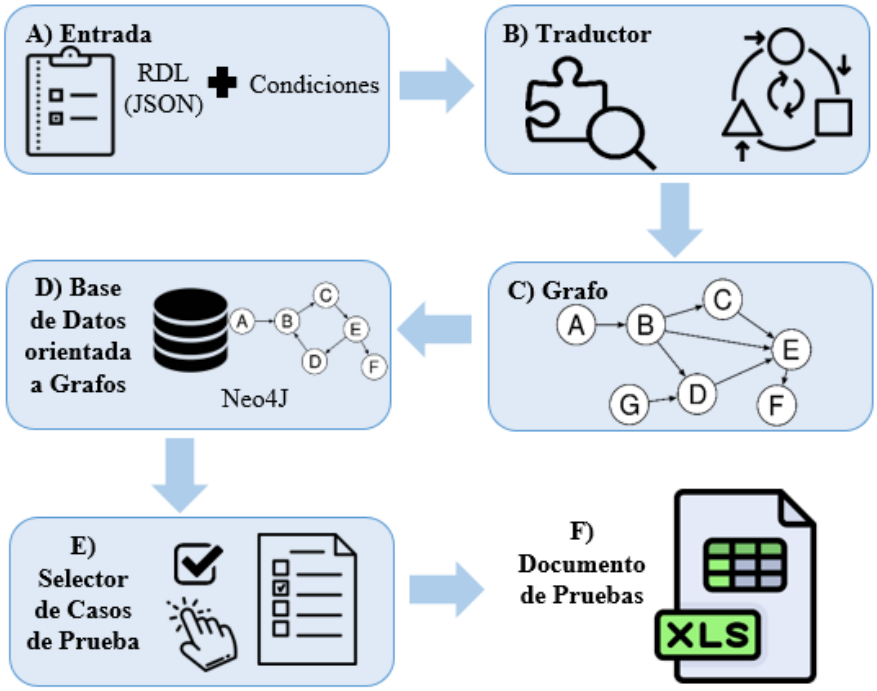


Figura 3.2 Proceso para la generación de pruebas Integrales y Funcionales

A continuación, se describirá cada una de las fases ilustradas anteriormente.

A) Entrada: este bloque representa todo lo que el usuario debe ingresar en la aplicación para que se generen las pruebas deseadas, o sea, el archivo JSON y las condiciones bajo las cuales se generarán los casos de prueba.

- El archivo JSON se obtiene como resultado de aplicar el RDL de la empresa Softtek® sobre un Caso de Uso, este archivo contiene detalles sobre los pasos que conforman el UC y que serán parte los Casos de Prueba, como el tiempo que tarda un usuario en efectuar una acción descrita en el Caso de Uso, quien la ejecuta y de quien se espera una respuesta, el tipo de dato y rangos que acepta un campo de captura.
- Las condiciones que el usuario debe especificar para generar Casos de Prueba son: el valor correspondiente al grado de cobertura, el tiempo con el que se cuenta para probar y/o que escenarios son los que se desean probar.

B) Traductor: en esta fase se procesan la entrada y las condiciones del usuario para crear un grafo dirigido. El algoritmo de esta sección, hace un reconocimiento de la información descrita en el archivo(s) JSON, para generar los nodos correspondientes a los escenarios (principal, alterno y de excepción) del Caso de Uso, y también se crean nodos con información relevante para la prueba, por ejemplo, si se requiere que se capture un campo de tipo numérico, en el grafo existirán los nodos que formen los caminos correspondientes a verificar el funcionamiento del campo de captura en situaciones donde se capturen letras, o números debajo y por encima del rango esperado.

C) Grafo: una vez creado el grafo para pruebas, éste se guarda en la **D) Base de Datos Orientada a Grafos** Neo4J, desde donde es posible realizar consultas para obtener el camino más corto o más largo según el escenario principal, alternos y/o de excepción, pero, esto no basta cuando se trata de identificar todas las secuencias de nodos posibles, por lo que se hace uso de un conjunto de algoritmos en el recorrido del grafo para cumplir con dicha meta.

E) Selector de Casos de Prueba: este componente se alimenta de lo que se definió en la **A) Entrada**, y se encarga de elegir los caminos que representan los casos de prueba en función de los requisitos especificados por el usuario, es decir, se hacen las operaciones necesarias para obtener una serie de combinaciones de pasos que satisfagan cuestiones de cobertura, tiempo disponible para realizar la prueba y/o tipo de escenario del Caso de Uso que se desea probar.

F) Caso de Prueba Resultante: una vez concluido la tarea del paso anterior, se exporta el resultado en un archivo de tipo .XLS, con las condiciones especificadas por el usuario.

3.1.2 Arquitectura

Tomando en cuenta el proceso descrito anteriormente para la obtención de Pruebas Funcionales e Integrales, en la Figura 3.3 se ilustra la arquitectura propuesta para el presente trabajo y se describe cada una de sus partes.

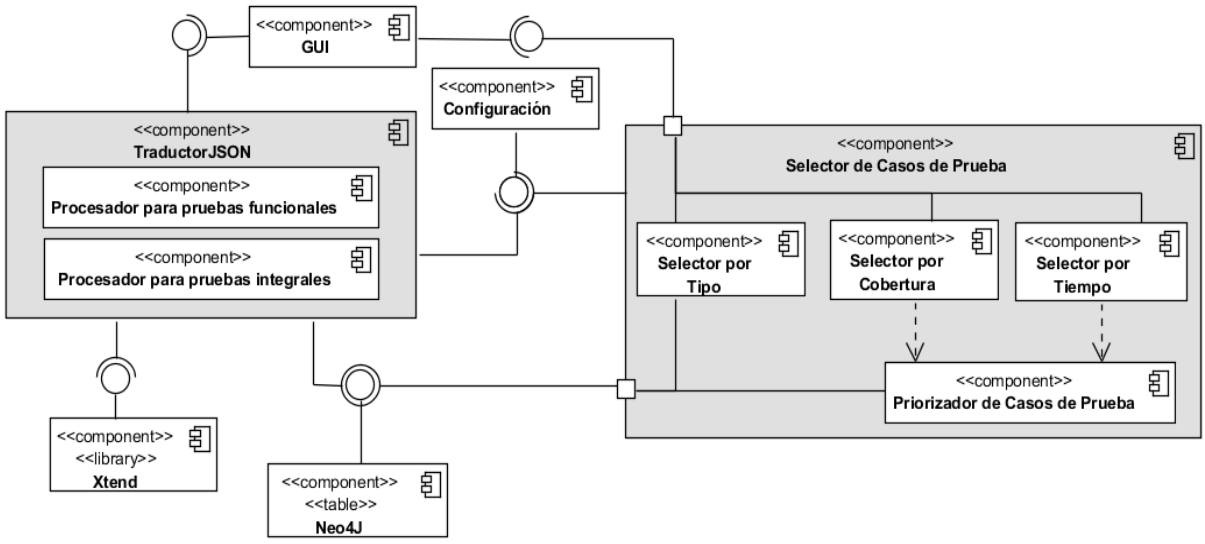


Figura 3.3 Arquitectura para el generador de Casos de Prueba

El componente **GUI** (*Graphic User Interface*, Interfaz Gráfica de Usuario) es aquel con el que interactúa directamente el usuario para introducir el/los archivo(s) JSON y las condiciones (como porcentaje de cobertura deseado, el tiempo disponible para probar, o los cursos principales, alternos, de excepción o de validación) que son necesarias para la creación de las pruebas.

Enseguida, el **Traductor JSON** se encarga de interpretar los datos de entrada para construir el grafo dirigido, esta tarea se logra gracias a la intervención de **Xtend** como lenguaje de programación de alto nivel y de **Neo4J**, como la base de datos orientada a grafos; el **Traductor JSON** también es responsable de armar las funcionales e integrales, por lo que internamente se compone de los siguientes dos elementos:

- **Procesador para pruebas funcionales:** este elemento de la arquitectura es el encargado de identificar las partes que constituyen el JSON, para luego transformar tal información en nodos que conformen al dígrafo correspondiente a la prueba funcional.
- **Procesador para pruebas integrales:** este elemento recibe tantos archivos JSON como Casos de Uso conformen la prueba integral, ya que ésta es equivalente a probar varios Casos de Uso relacionados.

Los componentes del **Selector de Casos de Prueba** ejecutan una serie de algoritmos al momento de recorrer el grafo almacenado para obtener Casos de Prueba con respecto a las condiciones especificadas por el usuario (como tiempo, porcentaje de cobertura y caminos a probar); a continuación, se describen dichos componentes:

- **Priorizador de Casos de prueba:** este realiza una consulta específica sobre el grafo para obtener un conjunto de caminos ordenados de acuerdo a la importancia de estos según el Caso de Uso original.

- El **Selector por Cobertura** y el **Selector por Tiempo** determinan cuántos elementos del conjunto obtenido del componente del punto anterior, corresponden al porcentaje de cobertura o al tiempo necesario para probar.

- **Selector por Tipo:** el comportamiento de este elemento es similar al de los antes mencionados, sin embargo, no hace uso del conjunto de caminos obtenidos con el Priorizador de Casos de Prueba, ya que actúa directamente sobre el grafo para obtener los caminos que corresponden al curso principal, alternos o de excepción del Caso de Uso original, aunado a lo anterior, también considera las validaciones que se incluirán en la prueba.

El componente Selector de Casos de Prueba, presenta al usuario a través del GUI los archivos con extensión .XLS que corresponden a los Casos de Prueba finales.

Finalmente, debido a que el proceso de generación de grafos y de casos de prueba utilizan datos como los nombres de los cursos del Caso de Uso, o los tiempos para cada tipo de paso en diferentes momentos, dichos datos se encapsularon en el componente de **Configuración;** además, se anticipa que el software genere pruebas en distintos idiomas, por lo que el soporte a una internacionalización también se encuentra en este componente de la arquitectura.

3.1.3 Plan de Iteraciones

Tomando en cuenta las virtudes del prototipo del Generador de Pruebas Funcionales e Integrales y los objetivos del presente trabajo, se desarrolló el siguiente programa de actividades para encaminar las acciones que logren dichos alcances:

1. Ajuste a nodos del grafo.
    A) Agregar un atributo que represente el tiempo de ejecución de un nodo en específico.
    B) En caso que el nodo sea de decisión, adicionar un atributo que represente si este tiene validaciones o no.
    C) Crear tantos nodos como casos de validaciones existan para un nodo de decisión.
2. Ajustes al JSON.
    A) Incluir un atributo que indique si los objetos del JSON incluyen validaciones.
        - De ser así, se crearán objetos para tales validaciones y se guardarán en un arreglo.
    B) Incluir un atributo que indique la complejidad de un Caso de uso y de cada uno de sus pasos.
    C) Incluir un atributo en los nodos de los cursos alterno y de excepción que indique que es posible terminar el Escenario de Prueba en ese punto.
3. Soporte a cursos alternos y de excepción.
4. Conseguir que la generación del Caso de Prueba soporte “ciclos”.
5. Diseñar el componente de Selector por Tipo.
6. Construir el Priorizador de Escenarios del Caso de Prueba.
7. Desarrollar el selector por tiempo.
8. Hacer el Selector por cobertura.

La especificación y el desarrollo de cada uno de los pasos presentados anteriormente, enseguida serán descritos en el presente documento.

3.2 Diseño

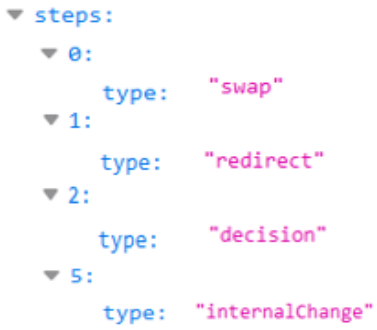
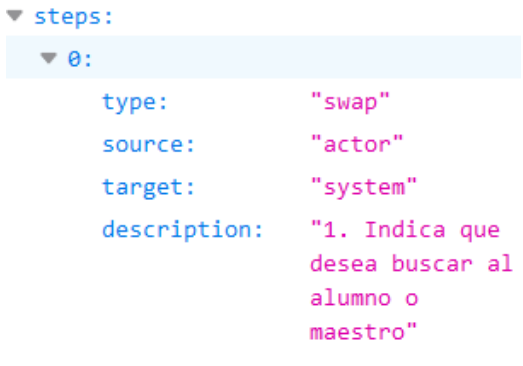
En la metodología XP, la fase de Diseño se caracteriza por la simplicidad, la recodificación, las soluciones *spike* y las metáforas, pero para este proyecto solo se considerarán los primeros dos conceptos, ya que se desea que con la mayor simpleza posible, se recodifiquen diferentes elementos del prototipo existente del proyecto, por lo que a continuación, se expone su funcionamiento (desde la estructura que tienen los archivos JSON que sirven como entrada en el componente GUI, hasta los tipos de nodos que conforman el grafo y la forma en que se enlazan) para posteriormente describir los cambios que se realizaron.

3.2.1 Estructura del archivo JSON para pruebas Funcionales

En la Tabla 3.1 se presenta la estructura de los archivos JSON, donde hay que señalar que los elementos visibles en la columna izquierda corresponden a datos de un registro usado para probar el prototipo del proyecto de tesis.

Tabla 3.1 Estructura de los archivos JSON para pruebas Funcionales

Fragmento del archivo JSON	Descripción
id: "CU0003" name: "Registrar Prestamo" goal: "guardar los datos..." trigger: "actor"	Esta sección del archivo JSON se llena con los datos generales del Caso de Uso (identificador, nombre, objetivo y quien lo ejecuta).
stages: […]	Un Caso de Uso tiene tres tipos de escenarios (principal, alterno y de excepción), y cada uno de estos se convertirá en un objeto que se almacenará en el arreglo llamado *stages* [] dentro del JSON.

Fragmento del archivo JSON	Descripción
	El primer elemento dentro del arreglo *stages* [] corresponde a un objeto de tipo *main*, lo que significa que pertenece al curso principal, mientras que el siguiente elemento (con respecto a la columna izquierda) es de tipo *alternate*, porque pertenece a un curso alternativo y finalmente el tipo *exception* corresponde al flujo de excepción. En un Caso de Uso solo tendremos un curso principal, pero no es así para el resto de los escenarios, por lo que en el JSON estos se nombran con alias relacionados al curso al que pertenecen más un número consecutivo ("alternative1").
	Los pasos que conforman cada curso dentro del Caso de Uso se convierten en objetos que se guardan en el arreglo *steps []*.
	Los objetos alojados en *steps []* clasifican como *swap, redirect, decision* e *internalChange*.
	Los objetos de tipo *swap* representan los pasos del Caso de Uso donde se hace un intercambio de información entre el usuario y el sistema o viceversa.

Fragmento del archivo JSON	Descripción
type: "redirect" source: "system" target: "system" description: "2.Ver CU consultar usuario Biblioteca" targetUseCaseName: "Consultar Usuario Biblioteca" targetStageName: "main" targetStageStepPosition: 0	Dentro de un Caso de Uso, existen instrucciones que indican cuando hay que consultar otro de estos, por lo que dichos pasos se clasifican como *redirect;* los objetos de este tipo tienen atributos particulares que corresponden al nombre (*targetUseCaseName)*, curso *(targetStageName)* e id del objeto (*targetStageStepPosition*) que pertenece al primer paso del UC referenciado.
▼ 13: type: "internalChange" source: "system" target: "system" description: "14.Marca cada libro incluico como Prestado"	Los objetos de tipo *internalChange* representan pasos del UC donde se manda a ejecutar una acción desde el sistema, y que se realiza por este mismo; por ejemplo, según la imagen de la columna izquierda, el atributo *description* de este paso nos indica que el software debe realizar la consulta necesaria para cambiar el estado de un libro a "Prestado".
type: "decision" source: "actor" target: "system" description: "3.Elige un alumno" ▼ options: ▼ 0: value: "true" targetUseCaseName: "this" targetStageName: "this" targetStageStepPosition: "3" probability: 0.2 ▼ 1: value: "false" targetUseCaseName: "this" targetStageName: "alternative1" targetStageStepPosition: "0" probability: 0.8	Dentro de un curso existen instrucciones que conducen a otro flujo dentro del mismo Caso de Uso, dichos pasos se tipifican como *decision*, y una de sus características es un arreglo llamado *options*, el cual se llena un objeto para el caso donde se continua con el mismo escenario (value: "true"), y otro para hacer referencia a otro curso (value: "false"), estos objetos incluyen atributos como: • *targetUseCaseName*: se llena con el valor "this" si se mantiene el flujo de acciones en el mismo Caso de Uso, o en caso contrario se escribe el nombre del Caso de Uso al que se hace referencia. • *targetStageName:* también se satisface con "this" si es que no se especifica el nombre del curso al que se apunta. • *targetStageStepPosition:* se llena con el valor del id del objeto que corresponde al primer paso del curso al que se direcciona la opción.

Fragmento del archivo JSON	Descripción
	• *Probability:* es la probabilidad de que un usuario escoja seguir el camino de un objeto de opción sobre otro.

3.2.2 Estructura del grafo para pruebas Funcionales

Tomando en cuenta la estructura de los archivos JSON, se infiere lo siguiente: un Caso de Uso está compuesto por diferentes flujos o tipos de escenarios (*main, alternate* y *exception*), y estos a su vez están formados por secuencias de pasos que dependiendo del tipo de tarea que realicen se clasifican en: *swap, redirect, internalChange* y *decisión*.

Un grafo está compuesto de nodos y relaciones; un objeto de un tipo de paso en específico del archivo JSON se convierte en un nodo, de tal manera que las relaciones entre estos se hacen con respecto al orden de ejecución de las instrucciones de un Caso de Uso, aunado a lo anterior, existen otros nodos que sirven para recorrer el grafo, por lo que en la Tabla 3.2 se describen las características de todos los ejemplares de nodos.

Tabla 3.2 Clasificación de los nodos en un grafo para pruebas Funcionales

Clasificación de los nodos	Descripción	Figura del nodo en el grafo	Atributos	Valor que almacena el atributo
info	Guardan la información general de un UC en específico y no se relacionan a otros nodos, por lo que se encuentran "libres" en el grafo.		*Goal*	El objetivo del UC.
			Id	Id del UC.
			Trigger	El actor que ejecuta el UC.
head	Los nodos *head* representan metafóricamente "la cabecera" de un curso del UC, por lo que contienen información relacionada a dicho escenario. Siempre están ligados al primer		*idNodeEnd*	Id del último nodo del curso.
			nameCaseUse	Nombre del UC al que pertenece.
			Type	Tipo de escenario al que está asociado el nodo (*main, alternate o exception*)

	nodo del curso al que hacen referencia.			
option	Se construyen a partir de los objetos almacenados en el arreglo *options []* por lo que siempre van precedidos de un nodo de *decision,* y por cada una de estos siempre existen dos nodos *option;* el nodo *true* se liga al siguiente nodo de la *decisión* mientras que el nodo *false* se liga al primer nodo del curso al que hace referencia. Si la información guardada en los atributos *targetStageName* y *targetStageStepPosition* es igual a "*this*", significa que se hace referencia al mismo UC y escenario del nodo de opción.	 	*idDecision*	Id del nodo de *decisión* que precede a la opción.
			nameCaseUse	Nombre del Caso de Uso al que pertenece el nodo.
			nameStage	Nombre del curso al que corresponde la opción.
			Probability	Probabilidad que tiene un nodo de opción de ser visitado.
			targetStage-Name	Nombre del escenario al que hace referencia la opción.
			targetStage-StepPosition	Id del nodo al que hace referencia la opción.
			targetUse-CaseName	Nombre del UC al que hace referencia la opción.
			Value	"*true*", para seguir en el mismo escenario de un UC y "*false*", si se dirige a otro UC y/o escenario.
reference	Este nodo sirve para indicar que se debe consultar otro UC solo si no se agrega el archivo JSON correspondiente, en caso contrario, este nodo se sustituye por el primer nodo del curso *main* del UC al que se hizo la referencia.		*type*	Siempre se usa el valor "*reference*".
			name	Nombre del Caso de Uso al que se hace referencia.

main, alternate, exception	Son los nodos correspondientes a los cursos *main* (son de color crema), *alternate* (color amarillo), *exception* (se pintan de color rojo). Además de los atributos de la última columna, existen otros dos llamados *Source* (quien ejecuta la acción guardada en el nodo) y *Target* (a quien va dirigida la acción del nodo), estos se llenan con los valores "*Actor*" o "*System*".		*Id*	Número progresivo que se le asigna a cada paso de un curso en un UC.
			nameCaseUse	Nombre del UC al que pertenece el nodo.
			nameStage	Nombre del escenario al que pertenece el nodo.
			type	Se llena con "*swap*", "*redirect*", "*reference*" o "*internalChange*" según el tipo de paso que represente el nodo.

Además de los atributos expuestos en la Tabla anterior, existen otros que están presentes en todos los nodos, en breve se describen dichos atributos:

- **<id>:** este atributo guarda un valor autogenerado por la base de datos, representa la posición del nodo dentro del grafo.
- **name:** los valores guardados por este atributo se mostrarán cómo se ejemplifica en el interior del nodo de la Figura 3.4.

Figura 3.4 Atributo name de un nodo

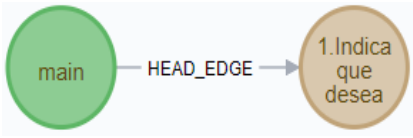
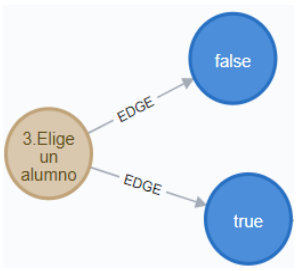
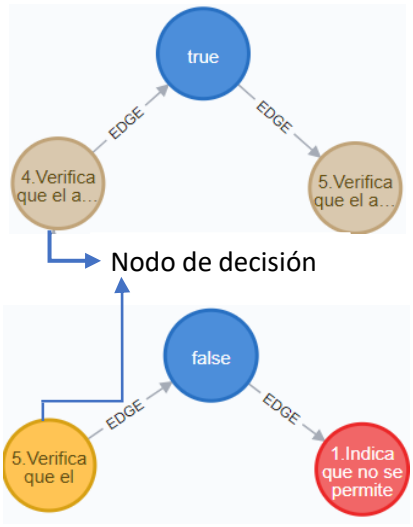
A continuación, se describen los tipos de relaciones existentes en el grafo:

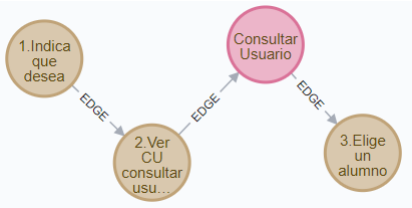
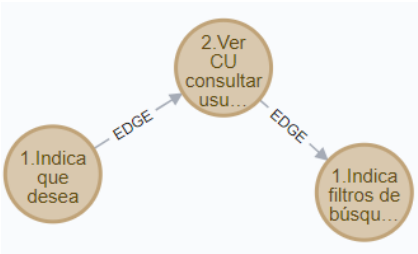
- **HEAD_EDGE**: este tipo de relación solo se usa cuando se unen un nodo *head*, con el primer nodo del curso al que hace referencia este.

- **EDGE**: es la relación común entre dos nodos mientras cualquiera de estos no sea de tipo *head.*

En la Tabla 3.3 se describen las estructuras que es posible formar con los distintos nodos y relaciones antes descritas.

Tabla 3.3 Descripción de la estructura del grafo para pruebas Funcionales

Clasificación de Nodo	Tipo de Relación	Visualización en la base de datos	Descripción
head	*HEAD_ EDGE*		Los nodos *head* siempre se ligan al primer nodo del escenario al que pertenecen con la relación *HEAD_EDGE*.
main, alternate, exception	*EDGE*		Un nodo *main, alternate,* o *exception* que guarda en su atributo *type* el valor *decision,* siempre está ligado (con una relación tipo *EDGE*) a dos nodos *option.*
option	*EDGE*		El nodo de opción *true* se relaciona al siguiente nodo del escenario al que pertenece dicha opción, mientras que el nodo *false*, se dirige al nodo del curso al que hace referencia.

Clasificación de Nodo	Tipo de Relación	Visualización en la base de datos	Descripción
reference	*EDGE*	Estructura 1  Estructura 2  	Los nodos *reference*, cuando no se agrega el archivo JSON del UC al que hace referencia, se ligan al siguiente nodo del curso al que pertenecen, como se muestra en la imagen de la Estructura 1 de la columna anterior, y en el caso donde sí se añade el JSON correspondiente al UC del *reference*, este se reemplaza por el primer nodo del escenario *main* del UC al que se refiere dicho nodo, como en la Estructura 2.
main, alternate, exception	*EDGE*	 	Los nodos con *type* igual a *swap*, e *internalChange* se relacionan con el nodo de *id* consiguiente del mismo escenario del Caso de Uso.
main, alternate, exception	*EDGE*	 	Los nodos que guardan el valor *redirect* en su atributo *type*, se relacionan al nodo del CU al que apuntan.

A continuación, en la figura 3.5 se presenta un ejemplo de un grafo generado con el prototipo del Generador de Casos de Prueba Funcionales e Integrales.

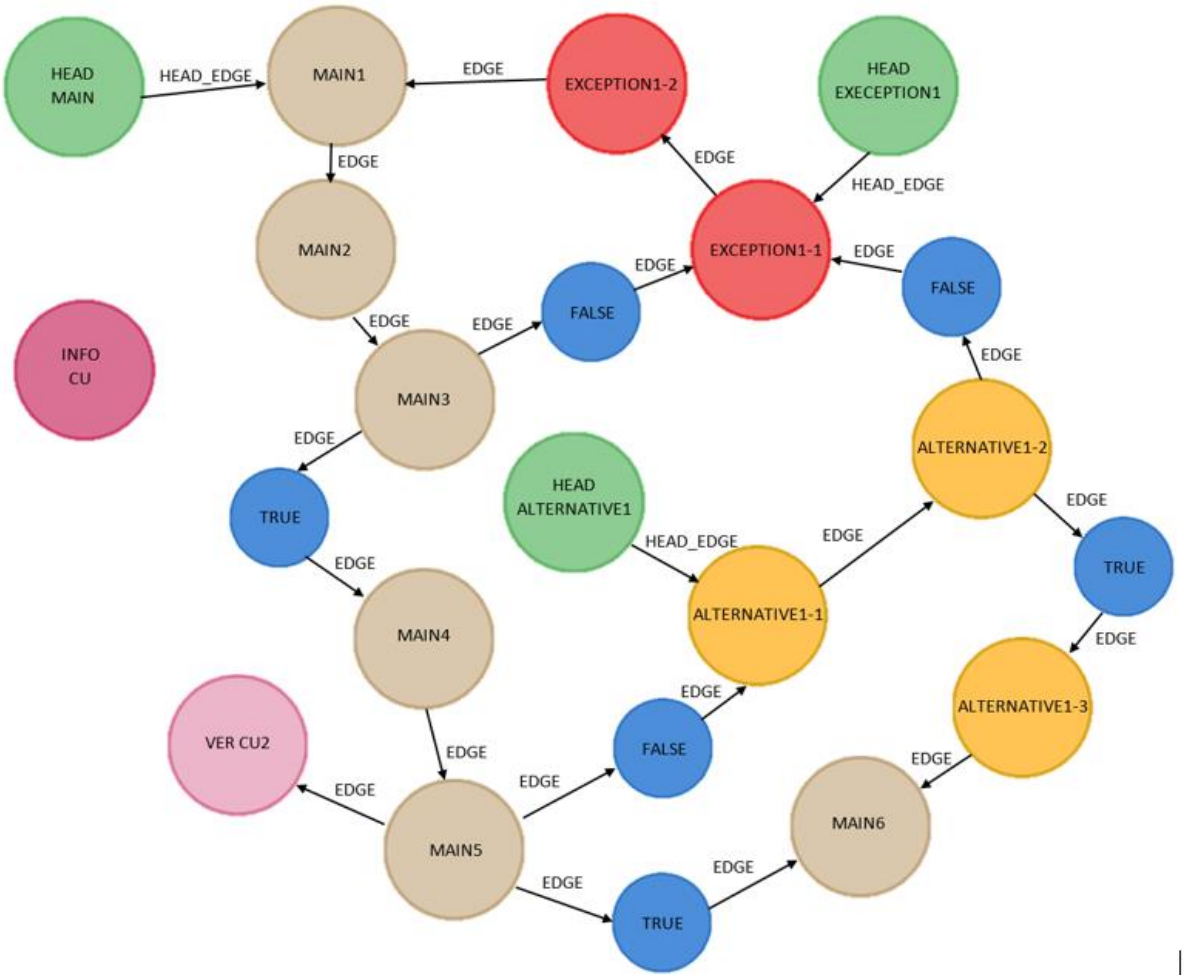


Figura 3.5 Ejemplo de grafo obtenido con el uso del prototipo del proyecto de tesis.

3.2.3 Estructura del archivo JSON para pruebas Integrales

Las pruebas Integrales se obtienen a partir del Diagrama de Actividades del Sistema, por lo que se hizo un análisis para transformar los elementos que se usan en dichos diagramas, en un archivo JSON, a continuación, en la Tabla 3.4 se presenta la descripción de la estructura de dichos archivos.

Tabla 3.4 Estructura de los archivos JSON para pruebas Integrales

Fragmento del archivo JSON	Descripción
`name: "Sistema Bibliotecario"` `goal: "Administrar Libros"`	Al principio del archivo JSON debe indicarse el *name* del sistema y el *goal* del mismo.

Fragmento del archivo JSON	Descripción
steps: […]	Las actividades que conforman el Diagrama de actividades se guardaran como objetos del arreglo *steps[]*.
steps: 0: type: "action" 1: type: "decision" 2: type: "point" 3: type: "object" 4: type: "fork" 5: type: "expansion" 6: type: "interuption"	Los objetos que se guardan en el arreglo *steps[]* se clasifican con la propiedad *type* como *action*, *decisión*, *point*, *object*, *fork*, *expansión*, e *interruption.*
0: id: 1 partition: "Catalogador" description: "Login Catalogador" type: "action" targetId: 2	Los objetos de tipo *action* tienen los siguientes atributos: • *Id:* que se llena con el paso al que corresponde la actividad en el diagrama de Actividades. • *Partition:* se refiere a la partición a la que pertenece la actividad. • *Description:* se llena con la descripción de la actividad. • *targetId:* corresponde al id de la siguiente actividad. La descripción anterior también se usa para los objetos de tipo *interruption.*

Fragmento del archivo JSON	Descripción
4: id: 5 partition: "Alumno/Maestro" description: "Inicio de decision 1" type: "decision" idEnd: 6 descriptionType: "start" ▼ options:	Los objetos de tipo *decisión* además de los atributos *Id, partition* y *description*, contienen los atributos: • *idEnd:* se llena con el *id* de la actividad con que termina la decisión. • *descriptionType:* toma el valor de *start* cuando la actividad corresponde al inicio de la decisión, y se usa el valor *end* para el caso contrario. • *Options[]:* arreglo que se llena con objetos correspondientes a que actividades se ejecutan a partir del rumbo que tome decisión.
▼ options: ▼ 0: targetIdOption: 6 condition: "no desea material" probability: 50 ▼ 1: targetIdOption: 7 condition: "mostrar material" probability: 50	Los objetos del arreglo *options[]* cuentan con los siguientes atributos: • *targetIdOption:* este atributo corresponde al *Id* de la actividad a la que se dirige la decisión a partir del atributo *condition.* • *Condition:* se llena con la descripción de la condición que es necesaria cumplir para continuar a la siguiente actividad. • *Probability:* se refiere a la probabilidad que tiene el usuario de cumplir con la condición de l opción presente.
id: 6 partition: "Alumno/Maestro" description: "Fin del Sistema" type: "point" descriptionType: "end"	Los objetos de tipo *point* hacen alusión a los elementos del diagrama de actividades que sirven para marcar el inicio o el fin de una secuencia de actividades. El atributo *descriptionType* se llena con la alguna de las siguientes palabras: • *end:* cuando es el fin de un curso de acciones. • *start:* en el caso de que se trate del inicio de un flujo de actividades, por lo que se agrega el atributo *targetID*, el cual se satisface con el *id* de la actividad siguiente del punto.

Fragmento del archivo JSON	Descripción
id: 9 partition: "Bibliotecario" description: "Objeto Prestamo" type: "object" targetId: 10 class: "Prestamo" stage: "new"	Los objetos de tipo *object* sirven para representar los casos donde se indica en el diagrama de actividades el estado particular de un objeto en el sistema, por lo que se usan los siguientes atributos: • *class:* clase a la que pertenece el objeto que se está representando. • *stage:* estado al que se cambia el objeto.
id: 10 partition: "" description: "Inicio de flujo paralelo 1" type: "fork" descriptionType: "start" idEnd: 14 ▼ targetId: 0: 11 1: 13	Los objetos de tipo *fork* forman nodos auxiliares que representen como tal la estructura fork de los diagramas de actividades, entonces, el atributo *partition* puede quedarse vacío, ya que un fork abarca más de un carril en los diagramas, además, los objetos deben incluir los siguientes atributos: • *idEnd:* se llena con el *id* de la actividad con que termina el fork. • *descriptionType:* que como ya se describió se complementa con las palabras *end/start.* • *targetId:* como un fork se une a varias actividades, este atributo se representa como un arreglo de Id, para que dentro de este se metan los Id de las actividades que se encuentran asociadas al fork.
id: 10 partition: "Bibliotecario" description: "Inicio de expansion 1" type: "expansion" descriptionType: "start" targetId: 11 numRepetitions: 2	Las estructuras de expansión se interpretan como ciclos dentro de los diagramas de actividades, por lo que, si el atributo *descriptionType* tiene la palabra *start*, se incluye el atributo *numRepetitions* en el objeto, para indicar el numero de veces que se repite una secuencia de actividades dentro del diagrama.

3.2.4 Estructura del grafo para pruebas Integrales

Gracias al archivo JSON que se obtiene a partir de la interpretación del Diagrama de Actividades del Sistema se crea el grafo que da origen a las pruebas integrales, por lo que en la Tabla 3.5 se describirán los elementos que conforman dicho dígrafo.

Tabla 3.5 Descripción de los componentes del grafo para pruebas Integrales

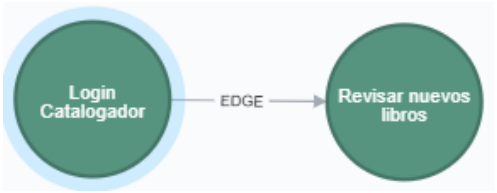
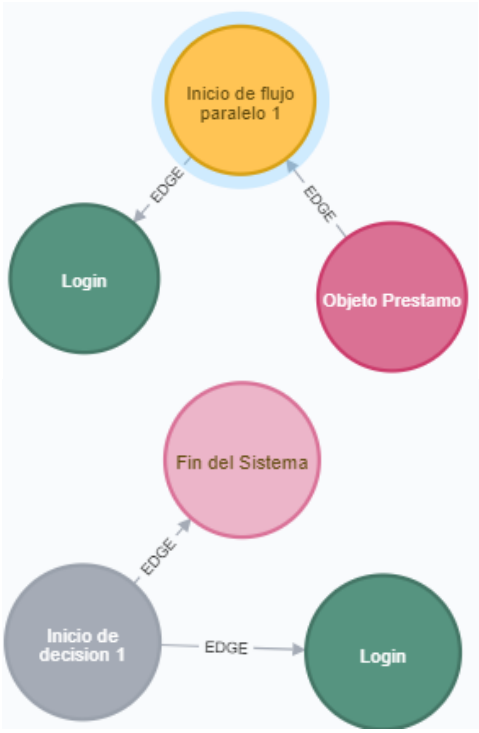
Clasificación de los nodos	**Descripción**	**Figura del nodo en el grafo**	**Atributos**	**Valor que almacena el atributo**
info	Nodo que contiene la información relacionada al sistema. Se crea gracias al atributo *name* y *goal* del archivo JSON. Este nodo no se liga a ningún otro dentro por lo que no se recorre cuando se hace la creación de pruebas integrales.		*goal*	Objetivo del sistema
			name	Nombre del sistema.
			lastID	Se llena con el Id del último nodo del grafo.
action/interruption	Corresponde a los nodos con el atributo *type* igual a *action.* Se ligan al nodo que tenga el Id referenciado en el atributo *targetId*		*id*	Id de la actividad.
			partition	Partición a la que pertenece la actividad.
			description	Descripción de la actividad.
			type	El tipo de este nodo es action.
			targetId	Id que corresponde a la siguiente actividad.
expansion	Estos nodos empiezan donde es necesario indicar que el conjunto de nodos subsecuentes se repetirá un cierto número de veces.		*descriptionType*	Se llena con el atributo *start* o *end*. Si se usa el valor *start* es necesario incluir el atributo *targetId*.
			numRepetitions	Se llena con el número que se repetirá la siguiente secuencia de nodos.

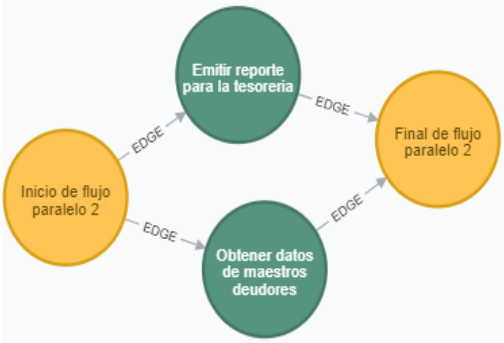
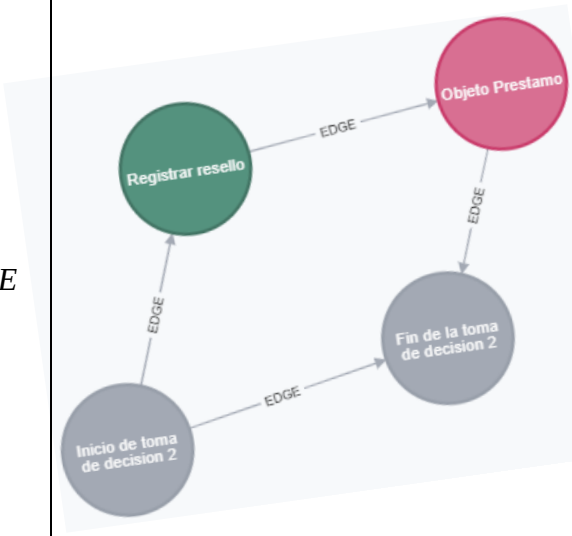
point	Este nodo sirve para indicar cuando inicia o termina una secuencia de actividades.		*descriptionType*	Se llena con el atributo *start* o *end*. Si se usa el valor *start* es necesario incluir el atributo *targetId*.
object	Este nodo representa los objetos que incluya el diagrama de actividades		*class*	Clase a la que pertenece el objeto en el sistema.
			stage	Estado del objeto.
fork	Nodos que sirven para representar las estructuras fork dentro de un diagrama de actividades		*targetId[]*	Arreglo de ID a los que va ligado el fork, siempre y cuando su atributo *descriptionType* igual a start
decision	Estos nodos sirven para representar el inicio de una decisión, van ligados a las opciones correspondientes a dicha decisión.		*Options[]*	Arreglo que contiene objetos de tipo option, dichos objetos contienen información que sirve para ligar el nodo de decision a sus respectivas opciones, por lo que la información que contengan los objetos del arreglo *options[]* estará guardada en la ligadura entre la decisión y los pasos de las opciones.

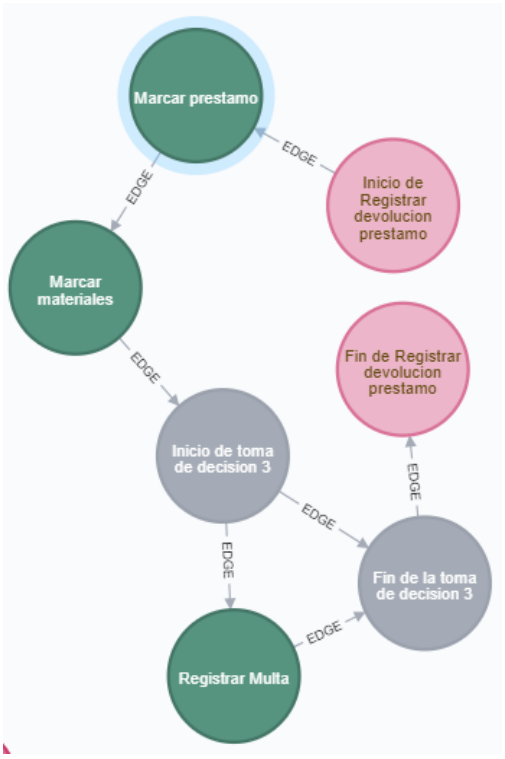
Las ligaduras para las relaciones entre los nodos son de tipo EDGE. Para el caso de los nodos de tipo *decisión*, los EDGE que los unan a sus correspondientes nodos contienen información como *condition* y *probability* (atributos descritos en la sección anterior).

A continuación, en la Tabla 3.6 se describirán las estructuras que se forman en el grafo a partir de la combinación entre diferentes tipos de ligaduras y nodos.

Tabla 3.6 Descripción de la estructura del grafo para pruebas Integrales

Clasificación de Nodo	Tipo de Relación	Visualización en la base de datos	Descripción
action	*EDGE*	  	Los nodos de tipo *action* siempre se ligan al nodo correspondiente a su atributo *targetId* al igual que los nodos de tipo *expansión, object, interruption, point, fork y decision*

Clasificación de Nodo	Tipo de Relación	Visualización en la base de datos	Descripción
Fork, action	*EDGE*		Las estructuras fork se caracterizan por tener un nodo fork de inicio y uno final, los nodos (o secuencias de nodos) involucrados en dicho fork se ligan al nodo de inicio y final de la estructura. Los nodos que quedan en medio de la estructura fork pueden ser de cualquier tipo.
decision	*EDGE*		Las estructuras de decisión también tienen un nodo de inicio y fin, si en el diagrama de actividades solo un camino conduce por una secuencia de nodos diferente, y el otro camino conlleva a la terminación de la decisión, en el grafo se vera una estructura como imagen de la columna de la izquierda.

Clasificación de Nodo	Tipo de Relación	Visualización en la base de datos	Descripción
point	*EDGE*		Con los nodos point es posible identificar el inicio y din de una secuencia de actividades. Como los nodos de tipo *point* también se usan para marcar el fin de un sistema, es posible que encontremos dichos nodos al finalizar el recorrido del grafo.

3.3 Desarrollo

En esta sección se describirán las acciones tomadas para cumplir con las tareas planteadas en la sección de **Análisis y Planificación** de la **metodología XP**.

1. Ajuste y creación de nodos.

Para cumplir con este objetivo, primero se agregó un atributo en los nodos para almacenar el tiempo que tarda un usuario o el sistema (según sea el caso), en ejecutar la acción del nodo, tal como se muestra en la Figura 3.6.

Figura 3.6 Atributo time en un nodo del grafo

En un UC existen pasos donde se indica que es necesario hacer la captura de uno o más datos, por lo que, para representar dichos pasos en el grafo dirigido es necesario usar un nodo *decision*, para que una opción de este, represente el caso de hacer una captura válida, y otra conduzca a los caminos donde no se hace un ingreso correcto de información, sin embargo, no todos los nodos *decision* representan estos pasos, por lo que se les agregó el atributo *isValidation* para distinguirlos, de tal forma que si el valor de dicho atributo es *true* significa que el nodo *decision* está ligado a caminos de validación (que son los que se usan en el grafo para verificar la captura de información), y si es *false*, se denota que la decisión no tiene relación a campos de captura.

En la Figura 3.7, se muestran los atributos de un nodo de *decisión* que cuenta con ramas o caminos de validación.

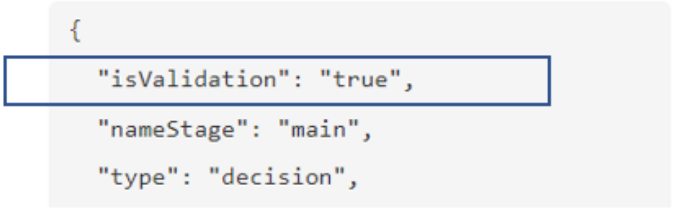


Figura 3.7 Atributo isValidation en un nodo de decision

Como se mencionó anteriormente, una de las opciones del nodo *decision* conduce a los casos donde no se captura correctamente un dato, y a continuación, se describen los criterios que se usan para realizar dicha acción en este proyecto:

- ***Required*:** si un campo es requerido obligatoriamente, entonces este se deja vacío.
- ***dataType:*** según el tipo de dato que acepte un campo de captura, se ingresa un valor que no corresponda a este.
- ***rangeMin:*** es común especificar un rango mínimo en los campos de captura, por lo que se introducen valores menores al aceptado.
- ***rangeMax:*** así como para el caso anterior, se llena el campo de captura con un valor mayor al aceptado.
- ***crossVal:*** existen casos donde un campo depende de otro, a esto se le llama *cross validation* (validaciones cruzadas), por lo que se ingresan valores para no cumplir con dicha validación.

Ingresar un dato erróneo sirve para verificar el funcionamiento de un campo de captura, ya que el software debe estar preparado para esos escenarios no ideales, por lo que los criterios antes expuestos son formas de verificar dichos campos, por lo que fue necesario crear nodos que representaran los pasos correspondientes a dichas validaciones.

En un UC existen uno o más campos de captura, por lo que, en cada uno de estos se aplican los criterios de validación que sean posibles, este proceso genera un conjunto de nodos de validación que no está disperso en el grafo, por lo que se agruparon dichos nodos en caminos, rutas o ramas de validación, donde una de estas conjunta todos los pasos de un mismo criterio, por lo tanto, un nodo *decision* puede tener hasta cinco caminos de validación (uno por cada criterio de validación), al final, cada rama lleva a la excepción donde se indica que no se ingresó el valor esperado, tal y como se muestra en la Figura 3.8.

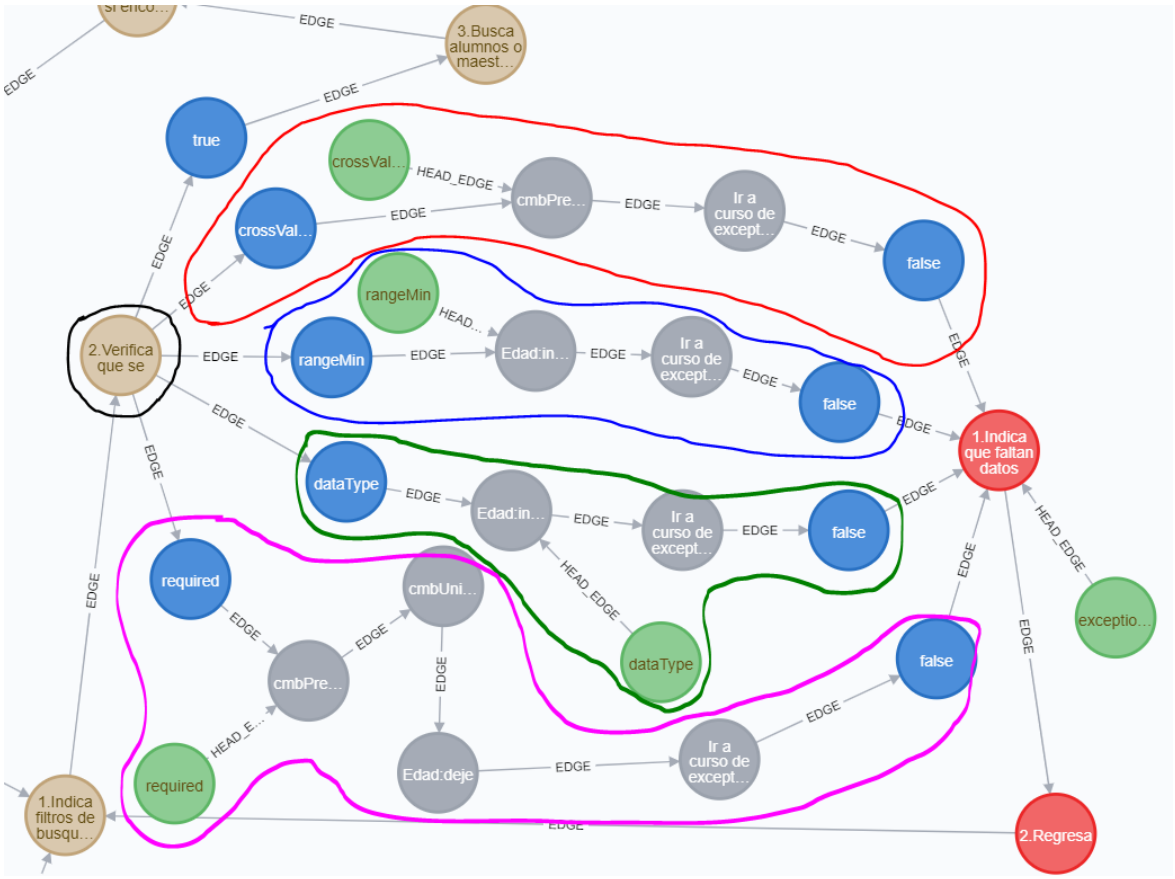


Figura 3.8 Representación de caminos de validación

Los conjuntos de nodos agrupados con líneas de colores representan caminos de validación, estos corresponden a verificaciones del funcionamiento de campos de captura del nodo remarcado en color negro, dichas validaciones parten de un nodo *option* en el que es posible apreciar en su atributo *name* el tipo de validación al que corresponde cada rama (*required*, *dataType*, *rangeMin*, *rangeMax* y *crossVal*), además, se observa como todas las ramas de validación culminan en el mismo flujo de excepción, también hay destacar que cada rama de validación es considerada como un curso, por lo que cada una cuenta con un nodo de cabecera ligado al inicio de cada validación y se recorre en el Generador de Casos de Prueba de la misma forma que se recorre cualquier otro flujo principal, alterno o de excepción.

2. Ajustes al JSON.

En cada uno de los pasos del archivo JSON se agregó un atributo llamado *complexityLevel*, este acepta valores como *medium*, *simple* o *complex*, y corresponde a la complejidad que toma realizar un paso de un UC.

Este atributo se implementó a nivel de UC y de paso, para que en caso de que no se determine la complejidad de una instrucción, la complejidad de esta se asigne con base a la del UC.

Dependiendo del valor del *complexityLevel* y del tipo de paso, la aplicación se encarga de determinar el tiempo promedio que tara en ejecutarse la acción de un nodo. En la Figura 3.9 se remarca el atributo de complejidad en el JSON.

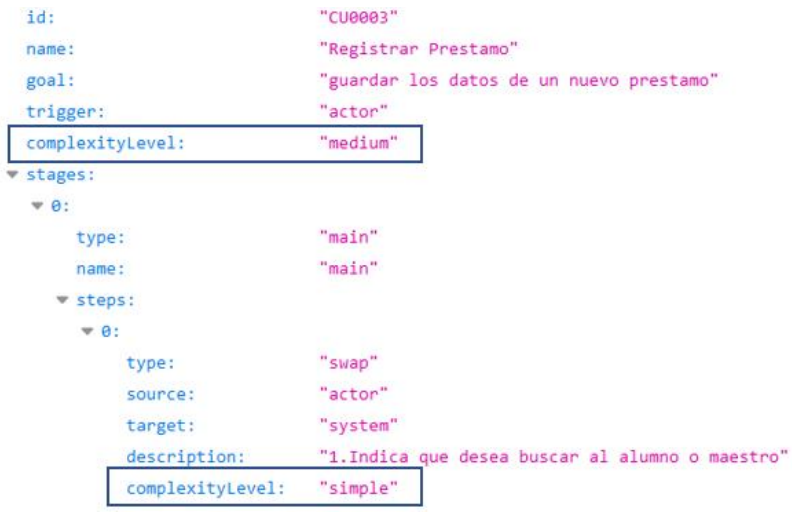


Figura 3.9 Atributo complexityLevel

También se agregó a los nodos con atributo *type* igual a *decision*, el atributo *isValidation*, el cual se llena con el valor de *true* o *false*, según se tengan o no validaciones, tal y como se muestra a continuación en la Figura 3.10.

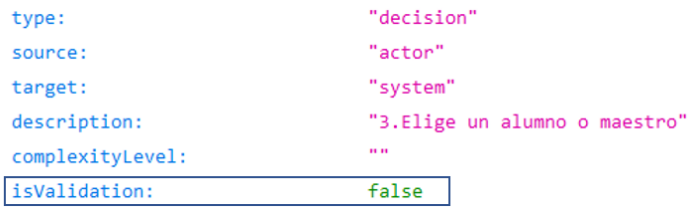


Figura 3.10 Atributo isValidation

En la Tabla 3.7 se explican las agregaciones que se hicieron para los casos donde una decisión si cuenta con validaciones.

Tabla 3.7 Agregaciones en el JSON para crear nodos de validación

Agregación en el JSON	Descripción
▼ validations:	*Validations* es un arreglo que se agregó en la entrada *steps* de los pasos tipo *decision* para almacenar las características de los campos de captura.
▼ 0: Id: "cmbPresentacion" dataType: "STRING" min: "" max: "" required: "true" pattern: "" ▼ crossVal:	Los objetos dentro *validations* tienen los siguientes atributos: • *Id:* acepta cadenas libres de restricciones. • *dataType:* acepta constantes como DATE, TIME, DATETIME, STRING, SHORT, INT, LONG, FLOAT, DOUBLE y es obligatorio que se llene. • *Min:* acepta valores de tipo cadena y va en función al valor de *dataType*, representa el valor mínimo que acepta el campo de captura. • *Max:* mismo caso que *min* pero para el rango máximo. • *Required:* acepta valores de *true* o *false*, según sea el caso de ser o no requerido el campo de captura. • *Pattern:* en caso de que el campo espere un valor en función a un patrón específico, en este atributo se guarda dicho patrón. • *crossVal:* es un objeto que se llena en caso de que el campo de captura cuente con alguna validación cruzada. Los atributos *min*, *max* y *pattern* aceptan valores de tipo *class.property*, en caso de que el valor con el que se espera llenar el campo de captura, sea un atributo en alguna clase del proyecto al que pertenece dicho campo.

Agregación en el JSON	Descripción
▾ crossVal: otherId: "cmbUnidad" validationType: "COMPARISON" otherFieldValue: "class.BULTO" comparisonOperator: "==" comparisonValue: "class.GRANO" comparisonId: ""	*crossVal* acepta los siguientes atributos: • *otherId:* se llena con un Id existente en el arreglo *validations*. Este Id es del que depende el campo de captura para operar correctamente. • *validationType:* se llena con REQUIRED o COMPARISON. El primer valor se usa cuando se requiere el llenado de otro campo obligatoriamente, y el segundo se usa cuando se compara el valor del actual campo de captura con otro. • *otherFieldValue:* este atributo solo se llena si *validationType* corresponde a COMPARISON, y también depende del campo especificado en *otherID.* • *comparisonOperator:* este atributo también se llena sólo si antes se especificó que el tipo de validación es COMPARISON, y se usan valores correspondientes a operadores lógicos como “==”, “!=”, “>”, “<”, “>=”, “<=” • *comparisonValue:* en el caso de que la validación sea COMPARISON, este atributo guarda el valor del campo de captura actual para que se compare con el valor del atributo *otherFieldValue.* • *comparisonId:* se llena con el Id del campo con el que se desea comparar la captura en caso de que el atributo *comparisonValue* este vacío, y *validationType* sea igual a COMPARISON

Para terminar un UC es necesario completar el último paso del flujo principal, sin embargo, hay ocasiones donde veces es posible realizar la misma acción desde cualquier otro flujo, por lo que se agregó un atributo en todos los pasos de los flujos alternos y de excepción, que indique si es posible terminar el Caso de Uso en ese punto, tal y como se muestra en la Figura 3.11.

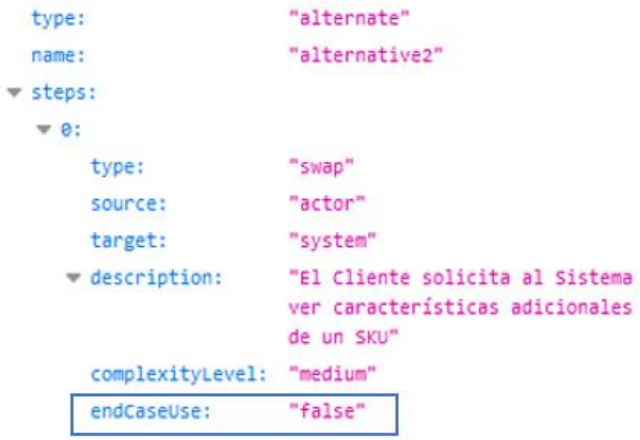


Figura 3.11 Atributo endCaseUse en un paso de un flujo alterno

3. Soporte a cursos alternos y de excepción.

En la versión prototipo del generador de Casos de Prueba, solo se obtenían pruebas correspondientes al curso principal del Caso de Uso, como en la Figura 3.12.

TEST CASE BASIC ('Registrar Prestamo')

ID: 'CU0003'
NAME: 'Registrar Prestamo'
GOAL: 'guardar los datos de un nuevoprestamo'
TRIGGER:'actor'

STAGES

NUM STEP	NAME STAGE	TYPE STEP	SOURCE	TARGET	DESCRIPTION	TIME
1	main	swap	actor	system	1.Indica que	5
2	main	redirect	system	system	2.Ver CU con	1
Redirect to Case Use : Consultar Usuario Biblioteca in the position: 1						
1	main	swap	actor	system	1.(nombre pr	5
2	main	decision	system	system	2.Verifica que	1
3	main	internalChan	system	system	3.Busca alum	1
4	main	decision	system	system	4.Verifica si e	1
5	main	swap	system	actor	5.Muestra si	5
Finish redirect for the Case Use 'Consultar Usuario Biblioteca'						
3	main	decision	actor	system	3.Elige un alu	2
4	main	decision	system	system	4.Verifica que	1
5	main	decision	system	system	5.Verifica que	3
6	main	internalChan	system	system	6.Muestra lo	10
7	main	swap	actor	system	7.Busca y elig	30
8	main	redirect	system	system	8.Ver CU Con	30

Figura 3.12 Caso de Prueba del prototipo del proyecto.

Pero gracias a la implementación del algoritmo DFS (*Depth First Search,* Búsqueda de Profundidad) se consiguió efectuar el recorrido de todos los cursos, tal y como se aprecia en la

Figura 3.13, donde es posible ver que en la columna correspondiente a “Categoría del Caso de Prueba”, aparecen las categorías *Happy Path, Happy Path-Exception* y *Happy Path-Alternate,* lo que significa que los Escenarios de Prueba se componen de pasos del flujo principal, del principal y de excepción, y del principal y el alterno respectivamente.

Código o Nombre del Caso de Uso	ID Escenario de Prueba	ID Caso de Prueba	Categoria del Caso de Prueba	Pasos	Descripción de los Pasos de Prueba
CU02.1: Colocar Pedido	EP1	CP1	Happy Path	1	El Sistema Valida que el Cliente no tiene Pendientes solicitudes de Cambio de Datos Generales
CU02.1: Colocar Pedido	EP1	CP1	Happy Path	2	El Sistema solicita al ERP los datos basicos del Cliente firmado en el sistema
CU02.1: Colocar Pedido	EP1	CP1	Happy Path	3	El ERP devuelve los datos basicos del Cliente por cada Cliente Hijo asociado
CU02.1: Colocar Pedido	EP1	CP1	Happy Path	4	El Sistema muestra los Clientes Hijos asociados al Cliente, mostrando los siguientes datos: Direccion de Consignacion (Nombre, Calle, Numero, Ciudad, C.P., Estado, Pais)
CU02.1: Colocar Pedido	EP1	CP1	Happy Path	5	El Cliente selecciona una Direccion de Consignacion como el Consignatario
CU02.1: Colocar Pedido	EP1	CP1	Happy Path	6	El Sistema muestra todas las Direcciones de Entrega asociadas al Consignatario seleccionado, con la siguiente informacion: Direccion de Entrega (Nombre, Calle, Numero, Ciudad, C.P., Estado, Pais)
CU02.1: Colocar Pedido	EP1	CP1	Happy Path	7	El Cliente selecciona una Direccion de Entrega.
CU02.1: Colocar Pedido	EP1	CP1	Happy Path	8	El Sistema muestra todas las Direcciones de Notificacion asociadas al Consignatario seleccionado con los siguientes datos:Direccion de Notificacion (Nombre, Calle, Numero, Ciudad, C.P., Estado, Pais)

Figura 3.13 Fragmento de un documento de Escenarios de Prueba, donde es posible ver la aparición de todos los flujos de un Caso de Uso.

DFS se utiliza para recorrer todos los nodos de un árbol o grafo dirigido de manera ordenada, su funcionamiento consiste en pasar de un nodo a otro en un camino concreto, y cuando ya no hay más puntos que visitar se repite el proceso desde uno de los nodos emparejados no visitados

del camino inicial; tomando en cuenta la Figura 3.14, el primer *path* (camino) que se forma con el algoritmo es 1-2-3-4, donde el nodo 1 es el inicio y 4 el nodo final.

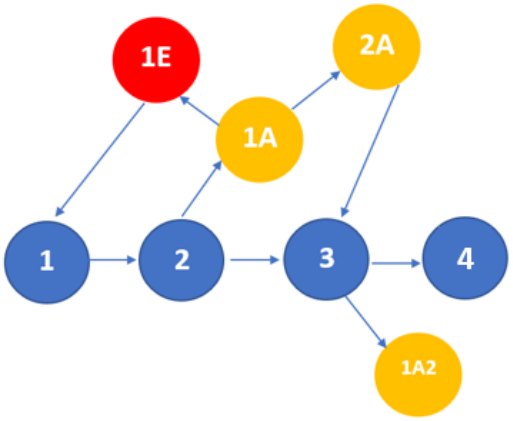


Figura 3.14 Ejemplo para demostrar el funcionamiento del algoritmo DFS

Ya que aún no se ha terminado de recorrer todo el grafo, se regresa al primer par no visitado más cercano al punto final, o sea, desde el nodo 4, en retroceso, el nodo que tiene una pareja que no fue visitada en el camino de la Figura anterior es el 3, ya que este no solo está ligado al nodo final, sino también tiene una relación con 1A2 tal y como se ve en la Figura 3.15.

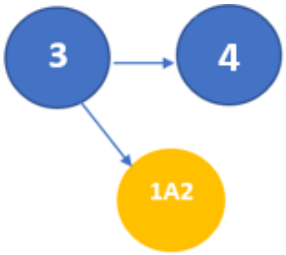


Figura 3.15 Parejas del nodo 3

Finalmente, cuando ya se recorrieron todos los nodos del grafo, pasando por todas las parejas no visitadas, el *path* final es el siguiente*;* 1-2-3-5-1A2-1A-2A-1E.

El resultado de la aplicación del algoritmo DFS es el conjunto de todos los nodos del grafo, pero para este proyecto es necesario obtener un conjunto de todos los caminos posibles de un nodo a otro, por lo que se hicieron las siguientes modificaciones en DFS para cumplir con el objetivo planeado:

- Al llegar a una pareja de nodos, se genera un nuevo camino combinando la secuencia del nodo de inicio hasta el primer nodo del par con el flujo de pasos que siguen del nodo no visitado hasta el nodo final; aplicando lo anterior a la Figura 3.14, se obtienen los siguientes caminos:
    - 1-2-3-4 (camino concreto).
    - 1-2-1A-2A-3-4 (la secuencia del camino concreto que se usó para construir este escenario es 1-2; 1A-2A-3-4 es el flujo resultante de pasar por el nodo no visitado 1A de la pareja 2-1A).

  Uno de los objetivos de la empresa es encontrar un conjunto de nodos finales, de tal forma que los Escenarios de Prueba se generen de un punto de inicio hasta un final especifico, entonces si se tiene como nodo de inicio al paso 1, y como finales del UC al nodo 4 y 1A2, deben encontrarse todos los caminos posibles de 1 a 4 y de 1 a 1A2.

  Por lo que los casos en donde no es posible llegar al nodo final a través de una pareja no visitada son descartados, sin embargo, más adelante, cuando se haga la búsqueda de caminos hacia otro final, se vuelve a retomar dicho camino.

- Cada camino nuevo se recorre con el algoritmo modificado con el objetivo de encontrar otros pares no visitados, para que a partir de estos se obtengan más Escenarios de Prueba, de tal forma que, con respecto al ejemplo anterior, se obtiene el siguiente camino*:*
    - 1-2-1A-1E-1-2-3-4 (el flujo de nodos 1-2-1A pertenece a la secuencia original, mientras que 1E-1-2-3-4 se obtiene del nodo no visitado de la pareja 1A-1E).

El conjunto de caminos obtenidos con el algoritmo DFS modificado en el grafo de la Figura 3.14 es el siguiente:

1. 1-2-3-5
2. 1-2-1A-2A-3-5
3. 1-2-1A-1E-1-2-3-5

4. Conseguir que la generación del Caso de Prueba soporte “ciclos”

Dentro de un grafo es posible encontrar flujos de nodos que se repiten más de una vez, por lo que se dice que forman una estructura cíclica, tal y como se muestra en la Figura 3.16.

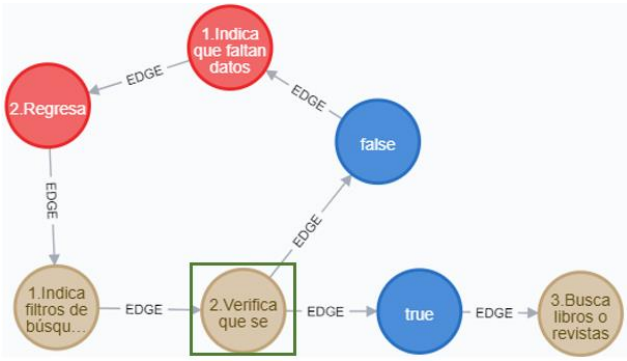


Figura 3.16 Estructura cíclica en un grafo

En la Figura anterior, el nodo dos del escenario *main* (enmarcado en un cuadro), es un nodo de decisión, entonces, si se cumple lo necesario para ir por la opción positiva (nodo *true*), seguimos el recorrido del grafo por el nodo tres del mismo escenario, sin embargo, si no se cumple con lo requerido en la decisión, es obligatorio ir por la opción negativa (nodo *false*), y esto significa que se ingresa a un escenario tipo *exception* que al finalizar, regresa al nodo inicial del escenario *main* (un nodo antes del de decisión), que eventualmente llega otra vez al nodo de decisión y esto significa que es posible volver a recorrer el flujo *exception;* dichos ciclos sirven para formar nuevos Escenarios de Prueba, puesto que a través de estas estructuras, es posible tocar nuevos nodos o generar nuevas combinaciones de pasos. Por lo que se trabajó en una nueva adaptación del algoritmo DFS para controlar el número de veces que se pasa por un ciclo y así, ampliar el conjunto de caminos posibles en un UC.

Primero se discernió que con ayuda del algoritmo DFS era posible guardar los nodos donde se iniciaba un ciclo, y la secuencia de este hasta el nodo final, para que después, en los caminos existentes, se buscaran dichos puntos, y se agregan los caminos asociados al ciclo, siempre y cuando no existiera antes dicha secuencia, pero, se descubrió que ésta estrategia (aunque muy provechosa para armar nuevos Escenarios de Prueba), decrementaba enormemente el rendimiento del programa, y esto se demuestra con la siguiente formula:

$$nCr = \frac{n!}{(n-r)!\,r!}$$

En donde *n* es igual al número de ciclos, y *r* la cantidad de bucles que es posible agregar en un camino (*path*); para un grafo que tiene cuatro ciclos, es posible obtener un total de 16 caminos, mientras que para ocho se obtienen 255, lo que quiere decir que la cifra de combinaciones posibles es un número exponencial con respecto a la cantidad de ciclos, por lo que el sistema tarda más en ejecutar una solución donde se combinan dichos ciclos para obtener nuevos caminos.

Con base en lo anterior, se decidió no incluir más de un ciclo para la solución del Generador de Pruebas y, como resultado, se obtuvo un conjunto de Escenarios de Prueba para un UC que incluye todas aquellas rutas por donde se pasa una sola vez por ciclo.

5. Diseñar el componente de Selector por Tipo.

En este proyecto el "*tipo*" se refiere al escenario que se desea probar, esto quiere decir que, si solo se requiere validar el flujo *main*, el Caso de Prueba resultante solo incluirá los pasos correspondientes a dicho flujo, tal y como se muestra en la Figura 3.17.

ID:	'CU0002'					
NAME:	'Consultar Usuario Biblioteca'					
GOAL:	'Buscar el o los usuarios de biblioteca (alumnos o maestros) que cumplan con un cierto filtro indicado por el actor'					
TRIGGER:'Actor o CU'						
NUM STEP	NAME STAGE	TYPE STEP	SOURCE	TARGET	DESCRIPTION	TIME
1	main	swap	actor	system	1.Indica filtro:	5
2	main	decision	system	system	2.Verifica que	1
	cmbUnidad:ingrese class.BULTO					
	cmbPresentacion:ingrese valor igual a class.GRANO					
	Edad:ingrese 24					
3	main	internalChang	system	system	3.Busca alumı	1
4	main	decision	system	system	4.Verifica si e	1
5	main	swap	system	actor	5.Muestra si e	5

Figura 3.17 Caso de Prueba para el escenario main

Luego de analizar la situación donde solo se requiere probar el flujo tipo *exception*, se llegó a la conclusión de que no es posible presentar una prueba donde solo existan los pasos de tal flujo, ya que estos dependen de lo que ocurra en el escenario principal, por lo que no tiene sentido mostrar al usuario los pasos de un flujo de excepción si no se sabe que acción hay que realizar para ejecutarlo, y pasa lo mismo para los casos donde se desea probar los escenarios tipo

validation y *alternative*, por lo que se consideró que para los casos donde el usuario indiqué que solo desea probar flujos de excepción, de validación o alternativos, el Caso de Prueba resultante incluirá dichos flujos y aquellos que los desencadenen.

6. Construir el Priorizador de Casos de Prueba.

Este módulo se encarga de ordenar los casos de prueba identificados con el algoritmo DFS con respecto a una prioridad, de manera que siempre en el archivo con extensión .XLS primero se vean pruebas con mayor probabilidad de suceder en el contexto del software que se esté probando.

Por ejemplo, en un UC se especifica que es necesario que un usuario se identifique en el sistema con su contraseña, por lo que existen dos posibilidades, que se ingrese correctamente dicha contraseña o que no, de tal forma que la situación que se presente con mayor regularidad es a la que se le dará mayor probabilidad.

Los pasos que cumplen con este tipo de condiciones son las decisiones, y dicho valor de probabilidad se encuentra alojado en el atributo *probability* de los objetos de *options[]*, por lo que al momento de pasar por un par no visitado con DFS, se pregunta por la probabilidad de la rama no visitada, para que en base a esta, se ordenen los caminos encontrados con el algoritmo.

7. Desarrollar el selector por tiempo.

En secciones anteriores se describió como se agregaron atributos relacionados al tiempo promedio en segundos que tarda en ejecutarse una acción del UC, ya que en este componente se explota dicha información.

Para explicar mejor el funcionamiento de este módulo, se hace el siguiente supuesto: un Caso de Uso genera un total de 300 pruebas, pero solo se cuenta con una hora para probarlo, por lo que el componente se encarga de seleccionar del conjunto priorizado solo aquellos casos de prueba que no rebasen el tiempo límite.

Por lo tanto, se contabilizan los segundos de cada paso de un caso de prueba para obtener el tiempo total de esta y después, dicho tiempo se resta al tiempo permitido de prueba, este proceso

se repite con el tiempo restante hasta que ya no se cuenta con suficiente tiempo para hacer otra prueba.

El resultado es el mismo, el generador entrega al usuario el documento .XLS correspondiente con los casos que es posible ejecutar dado un tiempo límite de prueba.

8. Hacer el Selector por cobertura.

Es posible que un Caso de Uso (UC) genere cientos de Casos de Prueba, pero esto no quiere decir que para validar el funcionamiento de dicho UC sea necesario probar todos esos casos, ya que sólo con ejecutar una cantidad de estos es posible decir que ya se probó lo necesario para validar el funcionamiento del software.

El porcentaje de cobertura está relacionado a lo anterior, y esto se comprueba con el siguiente ejemplo: se tiene un Caso de Uso con un total de 10 pasos por todos los escenarios, del que se generan los siguientes casos de prueba (donde un número es un paso en específico):

- 1-2-3-4-5
- 1-3-6-7-9
- 1-8-9-3-10
- 6-7-8-9-10

Para validar el funcionamiento del UC es necesario probar los 10 pasos, sin embargo, esto no significa que es imperioso ejecutar las cuatro pruebas anteriores, ya que solo con probar las tres primeras se cumple con la premisa de tocar todos los pasos, lo que quiere decir que el 100% de cobertura con respecto al total de pasos del UC se alcanza desde la prueba tres, entonces, si el usuario decide solo cubrir el 80% de cobertura de un UC, el módulo solo devolverá los casos de prueba que cubran con dicha cantidad de pasos.

Capítulo 4 Resultados

El Generador de Escenarios de Prueba Funcionales es una API (*Application Programming Interface,* Interfaz de Programación de Aplicaciones) que procesa uno o más archivos JSON para crear un grafo dirigido, posteriormente, recorre dicha estructura para obtener pruebas funcionales bajo criterios como el tiempo disponible para probar, el porcentaje de cobertura y/o el tipo de escenario que se desea validar, con ayuda de los módulos Selector por Tipo, Selector por Tiempo, y Selector por Cobertura, sin embargo, durante el periodo de Estancias Profesionales se convino con la empresa deshabilitar dichos componentes, puesto que los encargados del área de pruebas de la organización (a quienes les son útiles los resultados del proyecto), determinaron que dichos selectores se encuentran implementados en un nivel de granularidad muy alto, además de que es necesaria la intervención de variables no definidas en el alcance del presente proyecto.

La API del Generador de Escenarios de Prueba Integrales, procesa el archivo JSON correspondiente al diagrama de Actividades del sistema que se desea probar para obtener un dígrafo, para que a partir del recorrido de este se genere la prueba funcional.

En este capítulo se demuestra la funcionalidad de la versión del Generador de Escenarios de Prueba Funcionales que actualmente se usa en la empresa Softtek®, y la operatividad del Generador de Escenarios de Prueba Integrales en un caso de estudio, por lo que se construyó una aplicación “DEMO” para implementar ambas API, lo que significa que en las siguientes secciones se describirá la forma de usar los generadores a través de este software sin entrar en tantos detalles del funcionamiento del mismo, puesto que lo que se quiere presentar son las características de las API, no las de la aplicación DEMO.

4.1 Caso de Estudio

Para ambos generadores es necesario configurar el Usuario y Contraseña de la Base de Datos Orientada a Grafos Neo4j, por lo que a continuación, en la Figura 4.1, se muestra cómo se lleva a cabo dicha acción.

Figura 4.1 Conexión a la Base de Datos Neo4j a través de un usuario y contraseña

4.1.1 Caso de Estudio Generador de Escenarios de Pruebas Funcionales

La empresa desea obtener los escenarios de prueba funcionales de los siguientes Casos de Uso:

- CU02.1 - Colocar Pedido
- CU02.2 - Colocar Productos en Carrito
- CU02.3 - Cubicar Automático
- CU02.4 - Cubicar Manual
- CU02.5 - Generar Proforma
- CU02.6 - Validar Situación Financiera Proforma
- CU02.7 - Validar Lead Time Proforma
- CU02.8 - Generar Orden de Venta

A continuación, se presenta la información más relevante para este proyecto del primer UC de la lista anterior.

Generales del Caso de Uso			
Nombre Caso Uso	CU02.1 - Colocar Pedido		
Creación		**Fecha**	19/11/2008
Última modificación		**Fecha**	
Objetivo			
Permitir al Cliente el colocar un nuevo Pedido.			

Nivel del Caso de Uso	Prioridad	Complejidad
Usuario	Crítico	Alta
Actores involucrados		
Principal	**Secundarios**	
• A01 – Cliente		
Evento Disparador (Trigger)		
Solicitud de un nuevo Pedido		
Precondiciones		
• El Cliente fue identificado.		
Post-condiciones		
• El Pedido del cliente fue colocado. • Se generaron una o más Proformas asociadas al Pedido. • Se generaron una o más Órdenes de Venta asociadas a las Proformas.		

Escenario Principal

Paso	Acción
1.	El Sistema Valida que el Cliente no tiene Pendientes solicitudes de Cambio de Datos Generales.(***EX04***)
2.	El Sistema solicita al **ERP** los datos básicos del Cliente firmado en el sistema.
3.	El **ERP** devuelve los datos básicos del Cliente por cada Cliente Hijo asociado.
4.	El Sistema muestra los Clientes Hijos asociados al Cliente, mostrando los siguientes datos: • Dirección de Consignación (Nombre, Calle, Número, Ciudad, C.P., Estado, País)
5.	El Cliente selecciona una Dirección de Consignación como el Consignatario.
6.	El Sistema muestra todas las Direcciones de Entrega asociadas al Consignatario seleccionado, con la siguiente información: • Dirección de Entrega (Nombre, Calle, Número, Ciudad, C.P., Estado, País)
7.	El Cliente selecciona una Dirección de Entrega.

8.	El Sistema muestra todas las Direcciones de Notificación asociadas al Consignatario seleccionado con los siguientes datos: • Dirección de Notificación (Nombre, Calle, Número, Ciudad, C.P., Estado, País)
9.	El Cliente selecciona la Dirección de Notificación.
10	El Cliente proporciona el Número de Pedido Cliente.
11	El Sistema revisa que el Número de Pedido Cliente no se haya ingresado anteriormente.
12	El Cliente proporciona Fecha requerida y solicita los SKU's disponibles para comprar.
13	El Cliente selecciona los SKU's para su Pedido. (***CU02.2***)
14	El Sistema: • Identifica tantos Bloques como Lugares de Embarque estén implicados en los SKU seleccionados • Determina y muestra los SKUs en el Bloque correspondiente. • Determina y muestra, para cada Bloque identificado, las Rutas permitidas para la Dirección de Entrega. (La ruta se establece con base en el País de Dirección de Entrega y el País del Lugar de Embarque del Bloque) (***EX03***)
15	El Cliente selecciona la Ruta para cada Bloque.
16	El Sistema determina por cada SKU del Bloque: • Capacidad de Unidad de Transporte (***EA03***)
17	Por cada Bloque el Cliente indica la cantidad solicitada de SKU.
18	El Sistema calcula y muestra: • Subtotal por SKU • Total de Cantidades Solicitadas de todo el Bloque
19	El Sistema valida que cada Bloque puede ser ubicado automáticamente y lo ubica. (***CU02.3***) (***EA01***)
20	El Sistema muestra el Porcentaje de Uso de la Unidad de Transporte, por cada SKU, así como la cantidad de Unidades de Transporte por el Bloque.
21	El Cliente acepta la cubicación automática de cada Bloque. (***EA02***)
22	El Sistema valida que el Bloque usa Unidades de Transporte llenos. (***EX02***)
23	El Cliente indica que ha terminado de colocar su Pedido. (***EX01***)
24	El Sistema: • Genera el Número WEB.

	• Guarda el Pedido solicitado. • Guarda cada uno de los Bloques identificados. • Guarda datos de los SKU's Solicitados de cada Bloque.
25	El Sistema genera una Proforma por cada Bloque del Pedido. (***CU02.5***)
26	El Sistema realiza la Solicitud de Cubicación Manual de los Bloques marcados para cubicación manual con Proforma no cancelada. (***CU02.4***)
27	El Sistema realiza la Validación Financiera por cada Proforma aceptada y se encuentre con cubicación automática. (***CU02.6***)
28	El Sistema Valida el Lead Time de cada Proforma con Validación Financiera exitosa. (***CU02.7***)
29	El Sistema genera las Órdenes de Venta correspondientes por cada Proforma con Lead Time aprobado. (***CU02.8***)
30	El sistema le avisa al Cliente que su pedido ha sido colocado.
31	Fin del Caso de Uso.

Escenarios Alternos

EA01 – El Bloque requiere una cubicación manual	
Paso	**Acción**
1.	El Sistema valida que el Bloque debe ser cubicado manualmente.
2.	El Sistema marca el Bloque para solicitar cubicación manual.
3.	Fin del Escenario Alterno, continuar con el siguiente Bloque.

EA02 – El Cliente solicita una cubicación manual	
Paso	**Acción**
1.	El Cliente le indica al Sistema que necesita una cubicación manual.
2.	El Sistema marca el Bloque para solicitar cubicación manual.
3.	Fin del Escenario Alterno, continuar con el siguiente Bloque.

EA03 – El Bloque requiere una cubicación manual por falta de datos para cubicar automático	
Paso	**Acción**
1.	El Sistema determina que los datos son insuficientes para realizar cubicación automática.
2.	El Sistema marca el Bloque para solicitar cubicación manual.
3.	El Sistema envía notificación de datos insuficientes para cubicación automática.
4.	Fin del Escenario Alterno, continuar con el siguiente Bloque.

Excepciones

EX01 – El Cliente no desea colocar el Pedido	
Paso	**Acción**
1.	El Cliente le indica al Sistema que no desea colocar el Pedido.
2.	El Sistema avisa al Cliente que no se registrará el Pedido.
3.	Fin del Caso de Uso.

EX02 – No se están utilizando contendores completos en el Bloque	
Paso	**Acción**
1.	El Sistema determina que el Bloque no completa contenedores llenos
2.	El Sistema solicita al Cliente que el Bloque debe usar Contenedores llenos o que solicite una cubicación manual.
3.	Fin de la Excepción, continuar con el siguiente Bloque.

EX03 – No existen Rutas disponibles para el Bloque	
Paso	**Acción**
1.	El Sistema no encuentra Ruta disponible para el Bloque.
2.	Fin del Caso de Uso.

EX04 – El cliente tiene pendientes solicitudes de Cambio de Datos Generales

Paso	Acción
1.	El Sistema presenta aviso de solicitudes de cambio de datos generales pendientes.
2.	Fin del Caso de Uso.

En el escenario principal del UC anterior es posible notar que se hace referencia al resto de los Casos de Uso enlistados al principio de esta sección, debido a esta relación, el usuario incluyó en el generador todos los UC relacionados para obtener Escenarios de Prueba Funcionales certeros, tal y como se muestra en la Figura 4.2.

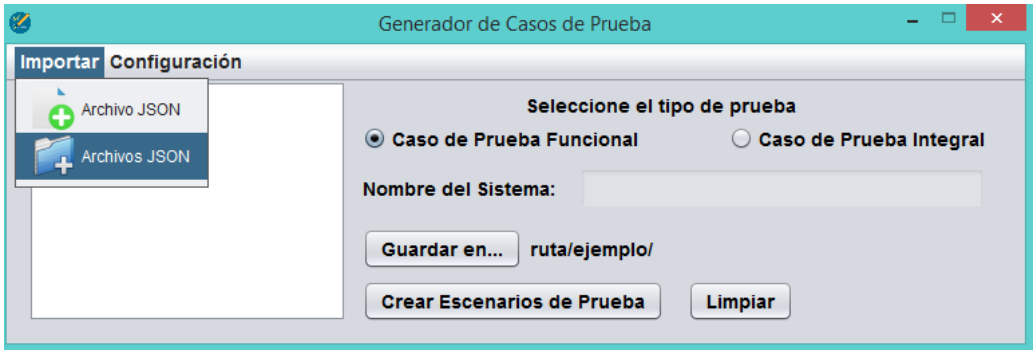


Figura 4.2 Selección de múltiples archivos JSON relacionados

En la Figura 4.3 se muestra como se agregaron los Casos de Uso relacionados y la ruta donde se guardó el documento de Pruebas correspondiente al CU02.1 Colocar Pedido.

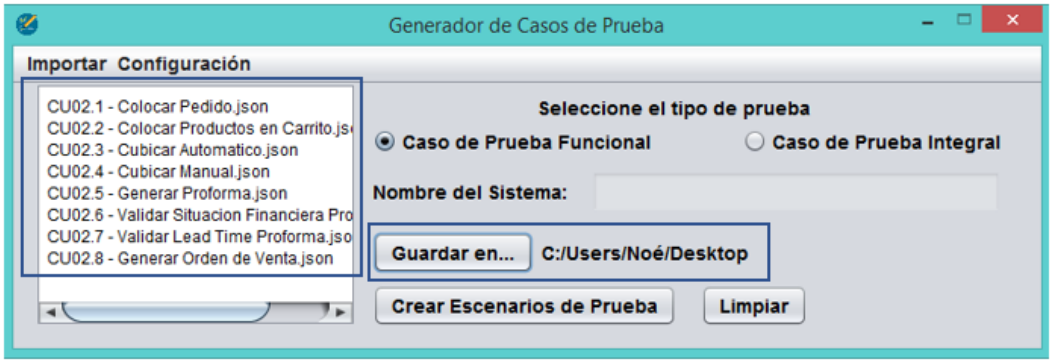


Figura 4.3 Agregación de Casos de Uso para la generación de Escenarios de Prueba Funcionales

Luego de generar la prueba correspondiente, se verificó que ésta guardó los resultados en la ruta definida como en la Figura 4.4.

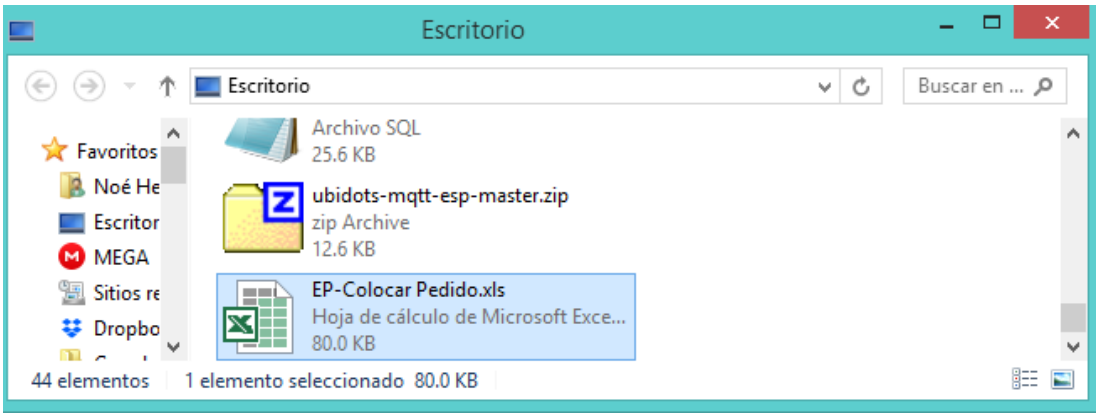


Figura 4.4 Comprobación de la existencia del documento de Escenarios de Prueba correspondiente al CU02.1 Colocar Pedido

A continuación, en la Figura 4.5, se muestra un fragmento del documento EP-Colocar Pedido.xls; hay que destacar, que fue necesario editar este para que en la Figura se mostraran las inclusiones a otros UC.

Aplicativo	Código o Nombre del Caso de Uso	ID Escenario de Prueba	Nombre Escenario de Prueba	ID Caso de Prueba	Categoria del Caso de Prueba	Condición y Datos de Entrada	Pasos	Descripción de los Pasos de Prueba	Resultados esperados
	CU02.1: Colocar Pedido	EP1		CP1	Happy Path		1	El Sistema Valida que el Cliente no tiene Pendientes solicitudes de Cambio de Datos Generales	
	CU02.1: Colocar Pedido	EP1		CP1	Happy Path		2	El Sistema solicita al ERP los datos basicos del Cliente firmado en el sistema	
	CU02.1: Colocar Pedido	EP1		CP1	Happy Path		3	El ERP devuelve los datos basicos del Cliente por cada Cliente Hijo asociado	
	CU02.1: Colocar Pedido	EP1		CP1	Happy Path		4	El Sistema muestra los Clientes Hijos asociados al Cliente, mostrando los siguientes datos: Direccion de Consignacion (Nombre, Calle, Numero,	
	CU02.1: Colocar Pedido	EP1		CP1	Happy Path		5	El Cliente selecciona una Direccion de Consignacion como el Consignatario	
	CU02.2: Colocar productos en	EP1		CP1	Happy Path		17	El Sistema muestra al Cliente su Carrito Vacio.	
	CU02.2: Colocar productos en	EP1		CP1	Happy Path		18	El Cliente agrega o quita SKUÂ´s a su Carrito.	
	CU02.3: Cubicar Automatico	EP1		CP1	Happy Path		27	El Sistema calcula numero total de Unidades de Transporte.	
	CU02.3: Cubicar Automatico	EP1		CP1	Happy Path		28	El Sistema devuelve el porcentaje de uso de la Unidad de Transporte por cada	
	CU02.3: Cubicar Automatico	EP1		CP1	Happy Path		29	El Sistema devuelve el numero total de Unidades de Transporte para el bloque	
	CU02.5: Generar ProForma	EP1		CP1	Happy Path		39	El sistema muestra una pantalla con los datos de la pro forma	
	CU02.4: Cubicar Manual	EP1		CP1	Happy Path		46	El BPM confirma que recibio solicitud de cubicacion.	
	CU02.4: Cubicar Manual	EP1		CP1	Happy Path		47	El Sistema marca la Proforma como Cubicacion Manual Solicitada	
	CU02.6: Validar Situacion Financiera Proforma	EP1		CP1	Happy Path		49	El Sistema envia una solicitud de Validacion Financiera al ERP- con los siguientes datos.	

Figura 4.5 Fragmento del documento de Escenarios de Pruebas Funcionales para el CU02.1 Colocar Pedido

Los pasos de los Casos de Uso a los que hace inclusión el CU02.1 Colocar Pedido, están incluidos en los Escenarios de Prueba Funcionales de éste, por lo que de esta forma se dice que todos los UC del listado anterior son probados, sin embargo, la empresa desea verificar las pruebas para el Caso de Uso CU02.2 Colocar Productos en Carrito, por lo que el usuario agregó al generador el archivo JSON correspondiente y seleccionó la ruta donde se guardó el documento de pruebas, tal y como se muestra en la Figura 4.6 y 4.7

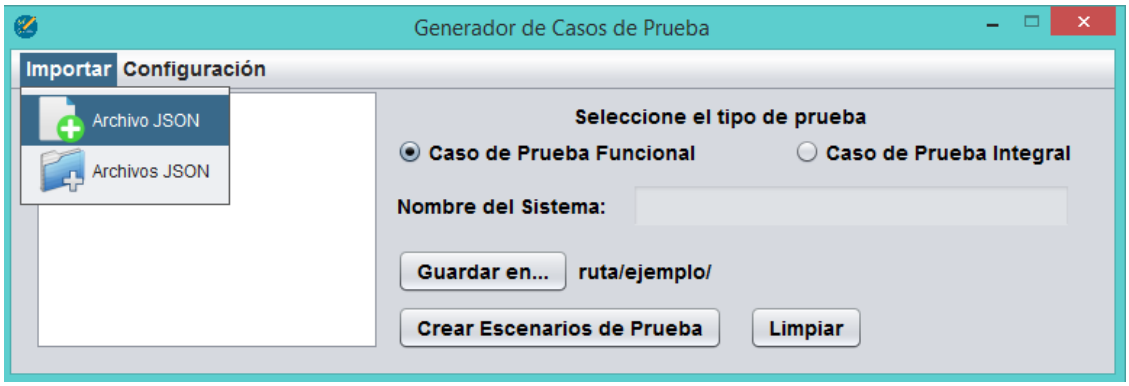


Figura 4.6 Selección de un archivo JSON.

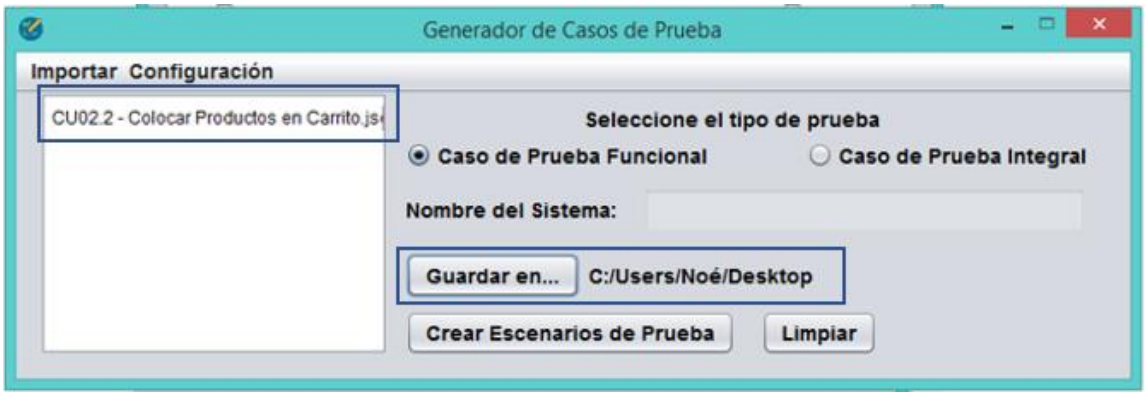


Figura 4.7 Agregación del JSON correspondiente a CU02.2 Colocar Productos en Carrito y selección de la ruta donde se guardará el documento de pruebas funcionales de este mismo UC.

Como el CU02.2 no hace referencia a otros UC, en la Figura 4.8 se muestra un fragmento del documento de Escenarios de Pruebas Funcionales de dicho Caso de Uso.

Aplicativo	Código o Nombre del Caso de Uso	ID Escenario de Prueba	Nombre Escenario de Prueba	ID Caso de Prueba	Categoria del Caso de Prueba	Condición y Datos de Entrada	Pasos	Descripción de los Pasos de Prueba	Resultados esperados
	CU02.2: Colocar productos en	EP1		CP1	Happy Path		1	El Sistema obtiene el Line Up del Cliente.	
	CU02.2: Colocar productos en carrito	EP1		CP1	Happy Path		2	El Sistema solicita al ERP el Precio vigente, Descripción corta, Foto, Ancho, Alto, Profundidad, Peso sin empaque, Peso bruto, Código cliente y Unidad de medida de los	
	CU02.2: Colocar productos en carrito	EP1		CP1	Happy Path		3	El Sistema muestra al Cliente la Lista de SKU´s obtenidos en el punto 2, excluyendo los marcados como "sin precio", muestra la siguiente información de los SKU´s. •Clave del SKU •País de la Planta •Lugar de Embarque	
	CU02.2: Colocar productos en	EP1		CP1	Happy Path		4	El Sistema muestra al Cliente su Carrito Vacío.	
	CU02.2: Colocar productos en	EP1		CP1	Happy Path		5	El Cliente agrega o quita SKU´s a su Carrito.	
	CU02.2: Colocar productos en	EP2		CP2	Happy Path-Exception		1	El Sistema obtiene el Line Up del Cliente.	
	CU02.2: Colocar productos en	EP2		CP2	Happy Path-Exception		2	El Sistema obtiene un Line Up vacío para el Cliente.	
	CU02.2: Colocar productos en	EP2		CP2	Happy Path-Exception		3	El Sistema notifica al Cliente que no tiene un Line Up definido.	
	CU02.2: Colocar productos en	EP3		CP3	Happy Path-Alternate		1	El Sistema obtiene el Line Up del Cliente.	
	CU02.2: Colocar productos en carrito	EP3		CP3	Happy Path-Alternate		2	El Sistema solicita al ERP el Precio vigente, Descripción corta, Foto, Ancho, Alto, Profundidad, Peso sin empaque, Peso bruto, Código cliente y Unidad de medida de los	

Figura 4.8 Escenarios de Pruebas Funcionales para el CU02.2 Colocar Productos en Carrito

4.1.2 Caso de Estudio Generador de Escenarios de Pruebas Integrales

Para probar el Generador de Escenarios de Prueba Integrales, el usuario seleccionó el JSON correspondiente al diagrama de actividades del sistema, escribió el nombre de este y designó la ruta donde se guardaría la prueba igual que en la Figura 4.9

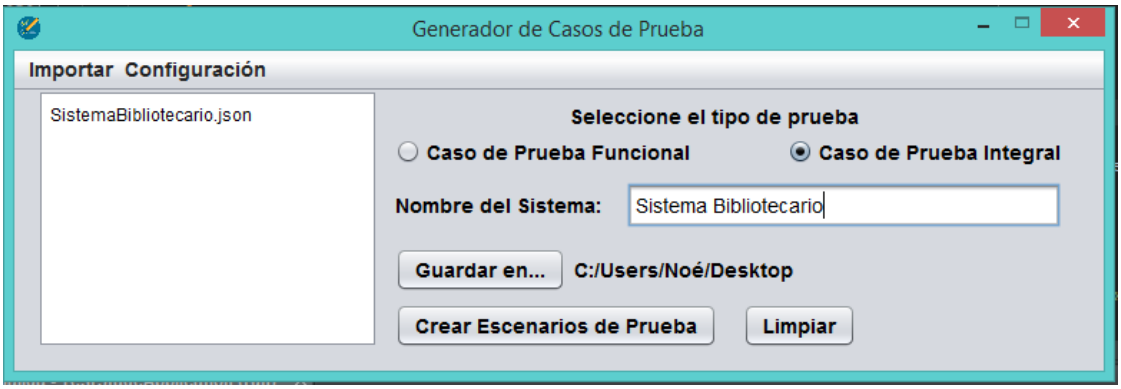


Figura 4.9 Selección del archivo JSON correspondiente al sistema, nombre de este y definición de la ruta donde se guardará el documento de prueba

Finalmente, un fragmento del archivo correspondiente al Escenario de Prueba Integral se presenta en la Figura 4.10.

NAME:	Sistema Bibliotecario	
GOAL:	Realizar prestamos y resellos de libros	
ID	PARTITION	DESCRIPTION
1	Catalogador	Login Catalogador
2	Catalogador	Revisar nuevos libros
3	Alumno/Maestro	Login ALumno/Maestro
4	Alumno/Maestro	Buscar material en catálogo
5	Alumno/Maestro	Inicio de toma de decisión 1
	no desea material (Probabilidad: 50.0%)	
6	Alumno/Maestro	Fin del Sistema
	mostrar material (Probabilidad: 50.0%)	
7	Bibliotecario	Login
8	Bibliotecario	Registrar Prestamo
9	Bibliotecario	Objeto Préstamo. Clase: class org.itcg.m
10	Inicio de flujo paralélo 1 [11-13]	
11	Administrador de la Biblio	Login
12	Administrador de la Biblio	Generar reporte de estadísticas de uso
13	Bibliotecario	Inicio de toma de decisión 2
	solicita resello (Probabilidad: 80.0%)	
15	Bibliotecario	Registrar resello
16	Bibliotecario	Objeto Préstamo. Clase: class org.itcg.m

Figura 4.10 Fragmento del documento de Escenarios de Prueba Integrales

Capítulo 5 Conclusiones y Recomendaciones

En el presente proyecto de tesis se realizó la interpretación de archivos JSON a grafos dirigido, para obtener escenarios de prueba funcionales e integrales a partir del recorrido de éstos.

El Generador de Escenarios de Prueba Funcionales, toma como punto de partida Casos de Uso descritos con el RDL de la empresa, de tal forma el documento de pruebas que esta API genera describe la prueba de un software, con respecto a su comportamiento esperado, además, la utilización del RDL evitó la introducción de errores en el producto final; aunado a lo anterior, el uso del algoritmo para recorrer el grafo, permitió el reconocimiento de todos los caminos mínimos posibles de un nodo a otro en un tiempo bastante aceptable, destacando que dichos caminos son equivalentes a los escenarios de prueba.

Para el caso del Generador de Escenarios de Prueba Integrales fue necesaria la exégesis de un diagrama de actividades para obtener el archivo JSON de entrada, por lo que el documento final corresponde al comportamiento de todo un sistema.

El trabajo que se realizó en conjunto con la empresa enriqueció significativamente el proyecto, y también sirvió para obtener nuevos objetivos para mejorar éste, ya que, aunque los resultados de las actuales API fueron satisfactorios, innovadores y útiles en el área de pruebas de la empresa, la misma desea continuar refinando ambos generadores en futuras versiones.

5.1 Trabajos a futuro

Generador de Escenarios de Pruebas Funcionales

- Además de la probabilidad, es necesario priorizar los escenarios de prueba funcionales con respecto a un criterio de criticidad.
- Un Caso de Prueba es la implementación de un Escenario de Prueba en específico con cierta información, por lo que se requiere crear tantos de estos casos para cada escenario con respecto a las reglas de negocio del sistema que se esté probando.
    - ❖ Los datos usados para llenar un Escenario de Prueba, y crear un Caso de Prueba, se obtiene de un modelo Entidad-Relación, por lo que se requiere crear un

componente o algoritmo, que permita la creación de dicha información a partir de la interpretación de estos modelos.

- Con respecto al formato de Pruebas Funcionales, deben hacerse las adecuaciones necesarias en el software para llenar las columnas correspondientes al Aplicativo, Nombre del Escenario de Prueba, Condición y Datos de Entrada y Resultados Esperados.

Generador de Escenarios de Pruebas Integrales

- Este generador debe funcionar como una extensión del anterior, ya que se desea que, a partir de la información de las pruebas funcionales, se generen las pruebas integrales.
- El conjunto de escenarios de pruebas integrales también debe priorizarse, por lo que debe discutirse con los encargados, cuáles serán dichos criterios de ordenamiento.
- Actualmente, para el funcionamiento de esta API se toma como punto de partida un diagrama de actividades, pero se aspira a que éste sea reemplazado por un diagrama BPMN (*Business Process Model and Notation*, Modelo y Notación de Procesos de Negocio).
    - ❖ Ahora la empresa hace uso del software Bizagi Process Modeler para generar diagramas BPMN que se exportan como archivos con formato XML, por lo que se desea que la creación del grafo se lleve a cabo a partir de dichos archivos.
        - ✶ Se sugiere acordar con los encargados del área de pruebas de la empresa, lo elementos que a usar en la construcción de los diagramas que sirvan al generador de pruebas integrales, puesto que Bizagi cuenta con una amplia gama de elementos visuales para crear BPMN.
- El documento de escenarios de prueba integrales también debe cumplir con un formato especificado por la empresa.

Los puntos enlistados anteriormente, son un resumen general de los objetivos que desea alcanzar la organización, por lo que, en el trabajo futuro, deben discutirse con más detalle cada uno de estos para obtener mejores resultados, además, por lo que se sugiere mantener activa la colaboración con los encargados del área de pruebas.

Es necesario hacer énfasis, que el Generador de Escenarios de Pruebas Funcionales tiene mayor prioridad con respecto al otro debido a cuestiones de tiempo y presupuesto, por lo que el esfuerzo debe concentrase en la mejora de este software.

Productos Académicos

Raquel Lizue Martínez Ramos, Beatriz Alejandra Olivares Zepahua, Celia Romero Torres, Luis Ángel Reyes Hernández, Ignacio López Martínez, *Propuesta de Arquitectura para un Generador de Casos de Prueba a partir de un RLD,* 3rd CoCSCE (*Conference on Computer Science and Computer Engineering*) en el marco del CI2T 2020

Vídeo explicativo para la empresa sobre la instalación de las API correspondientes al Generador de Escenarios de Prueba Funcionales e Integrales.

Vídeo demostrativo para la empresa sobre el funcionamiento de las API correspondientes al Generador de Escenarios de Prueba Funcionales e Integrales

Es necesario aclarar que, los archivos multimedia antes mencionados no se presentan por motivos de confidencialidad con la empresa.

Referencias

[1] G. J. Myers, The Art of Software Testing, Second Edition, 2.ª ed. New Jersey: Wiley, 2004.

[2] R. S. Wazlawick, «Testing», en Object-Oriented Analysis and Design for Information Systems, Massachusetts: Elsevier, 2013, pp. 271-287.

[3] I. Burnstein, Practical Software Testing. New York: Springer-Verlag, 2003.

[4] Lee Copeland, A Practitioner's Guide to Software Test Design. Boston: Artech House, 2004.

[5] M. D. Delgado Dapena et al., «Modelo para la generación automática de pruebas tempranas basadas en búsquedas», Comput. Sist., vol. 21, n.º 3, pp. 503-513, sep. 2017.

[6] R. Mukherjee y K. S. Patnaik, «A survey on different approaches for software test case prioritization», J. King Saud Univ. - Comput. Inf. Sci., oct. 2018.

[7] «El teorema de Turan: el comienzo de la teoría de grafos extrema», Gaussianos, 26-sep-2013. .

[8] «Grafo - EcuRed». [En línea]. Disponible en: https://www.ecured.cu/Grafo. [Accedido: 07-sep-2019].

[9] «Bases de Datos en Grafo - Fernando Sancho Caparrini». [En línea]. Disponible en: http://www.cs.us.es/~fsancho/?e=79. [Accedido: 07-sep-2019].

[10] H. M. Sneed, «Testing Against Natural Language Requirements», en Proceedings of the Seventh International Conference on Quality Software, Washington, DC, USA, 2007, pp. 380–387.

[11] «Understanding Requirements», en Software Engineering: A Practitioner's Approach, 7th ed., New York: McGrawHill, 2010, pp. 133-138.

[12] Meiliana, I. Septian, R. S. Alianto, Daniel, y F. L. Gaol, «Automated Test Case Generation from UML Activity Diagram and Sequence Diagram using Depth First Search Algorithm», Procedia Comput. Sci., vol. 116, pp. 629-637, ene. 2017.

[13] R. P. Verma y M. R. Beg, «Generation of Test Cases from Software Requirements Using Natural Language Processing», en 2013 6th International Conference on Emerging Trends in Engineering and Technology, 2013, pp. 140-147.

[14] «¿quiénes somos?», Softtek. [En línea]. Disponible en: https://www.softtek.com/es/acerca-de-softtek/iquienes-somos. [Accedido: 23-mar-2019].

[15] G. Carvalho et al., «NAT2TESTSCR: Test case generation from natural language requirements based on SCR specifications», Sci. Comput. Program., vol. 95, pp. 275-297, dic. 2014.

[16] S. Tiwari, D. Ameta, y A. Banerjee, «An Approach to Identify Use Case Scenarios from Textual Requirements Specification», en Proceedings of the 12th Innovations on Software Engineering Conference (Formerly Known As India Software Engineering Conference), New York, NY, USA, 2019, pp. 5:1–5:11.

[17] C. Wang, F. Pastore, A. Goknil, L. Briand, y Z. Iqbal, «Automatic Generation of System Test Cases from Use Case Specifications», en Proceedings of the 2015 International Symposium on Software Testing and Analysis, New York, NY, USA, 2015, pp. 385–396.

[18] A. Yazdani Seqerloo, M. J. Amiri, S. Parsa, y M. Koupaee, «Automatic test cases generation from business process models», Requir. Eng., vol. 24, n.º 1, pp. 119-132, mar. 2019.

[19] B. Li, C. Vendome, M. Linares-Vásquez, D. Poshyvanyk, y N. A. Kraft, «Automatically Documenting Unit Test Cases», en 2016 IEEE International Conference on Software Testing, Verification and Validation (ICST), 2016, pp. 341-352.

[20] E. Sarmiento, J. C. S. d P. Leite, y E. Almentero, «C amp;L: Generating model based test cases from natural language requirements descriptions», en 2014 IEEE 1st International Workshop on Requirements Engineering and Testing (RET), 2014, pp. 32-38.

[21] F. Daniel, E. Eduard, A. Wasif, S. Daniel, G. Thomas, y K. Avenir, «From Natural Language Requirements to Passive Test Cases Using Guarded Assertions», en 2018 IEEE International Conference on Software Quality, Reliability and Security (QRS), 2018, pp. 470-481.

[22] I. L. Araújo, I. S. Santos, J. B. F. Filho, R. M. C. Andrade, y P. S. Neto, «Generating Test Cases and Procedures from Use Cases in Dynamic Software Product Lines», en Proceedings of the Symposium on Applied Computing, New York, NY, USA, 2017, pp. 1296–1301.

[23] E. Daka, J. Campos, G. Fraser, J. Dorn, y W. Weimer, «Modeling Readability to Improve Unit Tests», en Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, New York, NY, USA, 2015, pp. 107–118.

[24] E. Sarmiento, J. C. S. P. Leite, E. Almentero, y G. Sotomayor Alzamora, «Test Scenario Generation from Natural Language Requirements Descriptions based on Petri-Nets», Electron. Notes Theor. Comput. Sci., vol. 329, pp. 123-148, dic. 2016.

[25] R. Lipka, T. Potuák, P. Brada, P. Hnetynka, y J. Vinárek, «A Method for Semi-automated Generation of Test Scenarios Based on Use Cases», en 2015 41st Euromicro Conference on Software Engineering and Advanced Applications, 2015, pp. 241-244.

[26] M. Zhang, T. Yue, S. Ali, H. Zhang, y J. Wu, «A Systematic Approach to Automatically Derive Test Cases from Use Cases Specified in Restricted Natural Languages», en System Analysis and Modeling: Models and Reusability, 2014, pp. 142-157.

[27] C. T. M. Hue, D. D. Hanh, y N. N. Binh, «A Transformation-Based Method for Test Case Automatic Generation from Use Cases», en 2018 10th International Conference on Knowledge and Systems Engineering (KSE), 2018, pp. 252-257.

[28] S. Nogueira, H. L. S. Araujo, R. B. S. Araujo, J. Iyoda, y A. Sampaio, «Automatic Generation of Test Cases and Test Purposes from Natural Language», en Formal Methods: Foundations and Applications, 2016, pp. 145-161.

[29] A. Ansari, M. B. Shagufta, A. S. Fatima, y S. Tehreem, «Constructing Test cases using Natural Language Processing», en 2017 Third International Conference on Advances in Electrical, Electronics, Information, Communication and Bio-Informatics (AEEICB), 2017, pp. 95-99.

[30] «Iniciándose en la plataforma Eclipse», 26-nov-2012. [En línea]. Disponible en: http://www.ibm.com/developerworks/ssa/library/os-ecov/index.html. [Accedido: 06-abr-2019].

[31] «Xtext - ¡La ingeniería lingüística es fácil!» [En línea]. Disponible en: http://www.eclipse.org/Xtext/#intro-quotes. [Accedido: 06-abr-2019].

[32] «Xtend - Documentación». [En línea]. Disponible en: http://www.eclipse.org/xtend/documentation/index.html. [Accedido: 06-abr-2019].

[33] «Neo4j Graph Platform – The Leader in Graph Databases», Neo4j Graph Database Platform. [En línea]. Disponible en: https://neo4j.com/. [Accedido: 13-may-2019].

[34] «Extreme Programming: A Gentle Introduction.» [En línea]. Disponible en: http://www.extremeprogramming.org/. [Accedido: 06-abr-2019].

[35] P. Coad, E. Lefebvre, y J. D. Luca, Java Modeling in Color With Uml: Enterprise Components and Process, Edición: Har/Cdr. Upper Saddle River, NJ: Prentice Hall, 1999.

[36] L. Madeyski, «Impact of pair programming on thoroughness and fault detection effectiveness of unit test suites», Softw. Process Improv. Pract., vol. 13, n.º 3, pp. 281-295, may 2008.