Universidad de las Ciencias Informáticas
Facultad CITEC

Trabajo de Diploma para optar por el título de

Ingeniero en Ciencias Informáticas.

Arquitecturas Limpias para el Sistema de Gestión Académica XAUCE AKADEMOS para el MINED.

Autor:

Claudia Palenzuela Pérez

Tutores:

Dr. C. Gerdys Jiménez Moya

Ing. Denis Sixto Francia Karell

La Habana, noviembre de 2022

Declaración de autoría

Declaro ser el único autor del trabajo de diploma *"Arquitecturas Limpias para el Sistema de Gestión Académica XAUCE AKADEMOS para el MINED"*, concedo a la Universidad de las Ciencias Informáticas y en especial al Centro de Informática Médica la autorización a hacer uso del mismo en su beneficio.

Para que conste firmo el presente documento a los ___ días del mes de _____ del año 20___.

Diplomante

_____

Firma del autor

Tutor

_____

Firma del tutor

Tutor

_____

Firma del tutor

Dedicatoria

A mi Dios por ser el más importante en mi vida, por ser ese padre celestial que me dio la fuerza para continuar.

A mis padres por la comprensión y el amor dedicado, a ellos mis respetos y mi corazón.

A mi abuelita linda, que, aunque no pudo ver este gran logro, sé que estuviera muy orgullosa.

A mi gran Amor, Enriquito, gracias a ti por tu apoyo incondicional, por ese brazo que me sostuvo y sobre todo por darme un poquito de tu inteligencia y sabiduría.

A mis tutores por el ánimo y el esfuerzo.

A mi prima bella, Estercita, esa lucecita que me da una palabra de aliento, por esa voz tan dulce, gracias por poner tu granito.

A mi tío Saul, mi tía Margui y prima Elsi, que a pesar de la distancia sé que siempre me desearon lo mejor, a ustedes gracias.

A todo el resto de la familia que me dieron su apoyo y sus buenos deseos, los llevo dentro.

Gracias a mis amigos y no tan amigos, a mi Yesi, Yiyi, Ernesto, Rixon, Yanetiti ustedes siempre desde primer año, gracias por las risas y el abrazo.

A todos los profesores que contribuyeron a que esta estudiante digo hoy bien grande “YA SOY INGENIERA”

Resumen

El centro de Tecnologías de la Formación es el encargado de la gestión académica de la Universidad de las Ciencias Informáticas (UCI). El sistema XAUCE AKADEMOS para el MINED administra los datos a informatizar. Dicho software tiene varias deficiencias lo que hace que sea poco mantenible. A partir de aquí se crea la necesidad de llevar a cabo una Arquitectura Limpia por capas para el módulo "*Usuario"* en la parte del BACKEND. La investigación se centra en diseñar una Arquitectura Limpia que proporcione mantenibilidad, facilidad de prueba, poco acoplamiento entre los datos, entre otras ventajas. Para la ejecución del trabajo se siguió una estrategia explicativa y se emplearon los métodos: análisis documental, inductivo-deductivo y modelado del resultado. Como ambiente de desarrollo se utilizaron Python, Django, Visual Studio Code y Git. La división de una arquitectura en capas facilita el diseño modular, en la que cada capa encapsula un aspecto concreto del sistema y permite además la construcción de sistemas débilmente acoplados, lo que significa que, si se minimiza las dependencias entre capas, resulta más fácil sustituir la implementación de una capa sin afectar al resto del sistema.

Palabras clave: Arquitectura Limpia, mantenibilidad, acoplamiento, Arquitectura por Capas.

Abstract

The Training Technology Center is in charge of the academic management of the University of Informatics Sciences (UCI). The XAUCE AKADEMOS system for MINED manages the data to be computerized. This software has several deficiencies which makes it not very maintainable. From this point on, the need to carry out a Clean Architecture by layers for the "User" module in the BACKEND part is created. The research is focused on designing a Clean Architecture that provides maintainability, ease of testing, low coupling between data, among other advantages. For the execution of the work an explanatory strategy was followed and the following methods were used: documentary analysis, inductive-deductive and modeling of the result. Python, Django, Visual Studio Code and Git were used as the development environment. The division of an architecture into layers facilitates modular design, in which each layer encapsulates a specific aspect of the system and also allows the construction of loosely coupled systems, which means that, if the dependencies between layers are minimized, it is easier to replace the implementation of a layer without affecting the rest of the system.

***Keywords**: Clean Architecture, maintainability, coupling, Layered Architecture.*

Contenido

Introducción

Desde la aparición de los primeros programas computacionales hasta la actualidad, el desarrollo de software ha pasado por un proceso evolutivo tan marcado y propio de cualquier área tecnológica[CITATION Joh18 \l 3082]. Para llevar a cabo dicho proceso se requiere estándares de codificación y buenas prácticas con el objetivo de obtener una buena anazabilidad, estabilidad y mantenibilidad.

Según Cabezas Gutiérrez [CITATION Cri16 \l 3082] en su investigación plantea:

- Conocer la situación actual de las organizaciones del estado y la gestión de los proyectos de desarrollo de software, identificando si se aplica o no buenas prácticas.
- Realizar una propuesta de integración entre las buenas prácticas y normas de control interno correspondientes a la Dirección de Proyectos Tecnológicos de la Contraloría General del Estado.

Cuando no se aplican buenas prácticas de programación, como no comentar el código, no llevar un control de versiones, la no existencia de una hoja de ruta documentada; al final el desarrollo se obtendrá un código fuente inmanejable, no se podrá regresar a una versión anterior del código, y al efectuarse cambios en el grupo de desarrollo, un nuevo programador no podrá hacerse cargo del desarrollo elaborado [CITATION Jor19 \l 3082].

Para la organización de un sistema es importante utilizar una arquitectura de software. En el libro “Ingeniería del software, un enfoque práctico” de Roger S. Pressman [CITATION Rog10 \l 3082] se define la arquitectura de software de un programa como la estructura o estructuras del sistema. Además, comprende a los componentes del software, sus propiedades externas y las relaciones entre ellos.

En los últimos años, la industria del software se ha enfrentado a nuevos desafíos en cuanto a complejidad, costos, tiempo de comercialización, estándares de calidad y evolución. Para enfrentar estos desafíos tanto en la academia como en la industria, se encuentran enfoques y herramientas como, las arquitecturas limpias [CITATION Fed21 \l 3082].

En el mundo todos quieren aplicar buenas prácticas en sus proyectos software, basado en arquitecturas limpias, para que las organizaciones puedan seguir un estándar y mejorar tiempos de

producción. La ventaja es que el proyecto será mantenible para todos los implicados y para futuros integrantes que lleguen nuevos [CITATION Jul20 \l 3082].

Una Arquitectura limpia se puede aplicar o implementar en muchas plataformas, en esta investigación, se trata para el Sistema de Gestión Académica. Se propone una estructura de manera que sean escalables, mantenibles, fácil de realizar pruebas, entre otros atributos de calidad.

A continuación, se mencionan algunos trabajos relacionados en los que se ha trabajado con el uso de Arquitecturas limpias. Por ejemplo, según Difabio (2021) [CITATION Fed21 \l 3082] en su trabajo de investigación, diseña e implementa una arquitectura de software que permite el desarrollo guiado por el dominio, quedando como resultado en el backend la arquitectura hexagonal. En el trabajo de diploma Concepción (2008) [CITATION Ore08 \l 3082] propuesta de una arquitectura en php, hace uso de una arquitectura por capas. Según Solanes (2018) [CITATION Sol18 \l 3082] en su trabajo de investigación, desarrolla una aplicación Android para un restaurante aplicando principios de Arquitectura Limpia y su implementación. Es el caso del trabajo de Rodríguez (2019) [CITATION Rod19 \l 3082], quien propone en su trabajo de fin de máster llamado “Un caso práctico de Mobile-D”, como su nombre lo dice, un caso práctico de mobile-D, como metodología ágil, en el desarrollo de aplicaciones móviles y utiliza Arquitectura limpia para la aplicación desarrollada; utiliza Django como tecnología. Es el caso de Knill (2019) [CITATION Kni19 \l 3082], propone en su trabajo de investigación sobre Refactoring con Arquitectura limpia, en donde demuestra cómo puede refactorizar una aplicación existente con Arquitectura Limpia sin tener que iniciar el proyecto desde cero. La mayoría de los proyectos mencionados anteriormente son para uso en aplicaciones móviles, en el caso del presente trabajo será para el sistema de gestión académico.

En el Centro de Tecnologías para la Formación (FORTES) está adscrito a la Facultad de Tecnologías Educativas, desarrollas productos y servicios para la implementación de soluciones de formación aplicando las Tecnologías de la Información y las Comunicaciones, a todo tipo de instituciones con diferentes modelos de información. En este se desarrolla el sistema Akademos, el cual es un sistema para la gestión académica. Se divide en *Frontend*, *Backend* y Diseño de Base de Datos. En esta investigación se aborda las Arquitecturas Limpias para el desarrollo *Backend* [CITATION Mig21 \l 3082]

Debido a que en la UCI la matricula es creciente y se trabaja con varios planes de estudio a la vez, la gestión académica resulta engorrosa. El software crece en complejidad, número de desarrolladores y cargas de trabajo, quedando un programa informático extenso.

El alto grado de complejidad alcanzado por el Sistema Akademos conlleva a que con el tiempo los componentes se acoplan, afectando la gestión. El proyecto tiene varias desventajas:

- Tiene poca facultad para diagnosticar sus deficiencias o causas de fallos, o de identificar las partes que deben ser modificadas.
- Baja capacidad para realizar cambios al sistema porque no permite implementar una modificación especifica previamente, incluyendo los cambios en el diseño, el código y la documentación.
- Corta estabilidad para minimizar los efectos inesperados de las modificaciones.
- Poca cabida a la facilidad de prueba porque no permite evaluar las partes modificadas.
- Poca suficiencia de conformidad ya que no satisface los estándares o convenciones relativas con la mantenibilidad.

Como consecuencia de la situación problemática antes expuesta se deriva el siguiente **problema científico de investigación:** ¿Cómo mejorar los estándares de mantenibilidad en el Akademos?

Objeto de estudio: Arquitectura Limpia en el desarrollo de sistemas backend.

Campo de acción: Arquitectura Limpia en el desarrollo de sistemas backend en el sistema Akademos.

Objetivo general: Aplicar una Arquitectura Limpia en el desarrollo del backend del sistema Akademos para mejorar los estándares de mantenibilidad.

Preguntas científicas:

- ¿Cuáles son los principales conceptos relacionados con la Arquitectura Limpia?
- ¿Cuáles son los métodos, metodologías, tecnologías, herramientas y técnicas más adecuadas para el desarrollo de la propuesta de solución?
- ¿Cuáles son los artefactos según la metodología seleccionada que permitirán un mejor diseño de la propuesta de solución?
- ¿Cuáles son las buenas prácticas que propone la implementación de la Arquitectura Limpia?

- ¿Cuáles son las pruebas de software que se llevarán a cabo para la validación de la propuesta de solución?

Hipótesis: Si se aplica una arquitectura limpia por capas para desarrollar el sistema de gestión académico, entonces se contribuye a que el software tenga una mejor mantenibilidad, cambiabilidad y facilidad de prueba.

La investigación se desarrollará a través de las siguientes **tareas de investigación**:

1. Análisis de los principales conceptos relacionados con las Arquitecturas Limpias.
2. Selección de los métodos, metodologías, tecnologías, herramientas y técnicas a utilizar en el sistema backend utilizando una Arquitectura Limpia que mejore la mantenibilidad, cambiabilidad y facilidad de prueba del sistema Akademos.
3. Diseño de la Arquitectura Limpia.
4. Implementación de la Arquitectura Limpia.
5. Validación de la Arquitectura limpia.

Métodos teóricos:

- Histórico – Lógico: Para revisar los resultados que se han obtenido implementando una Arquitectura Limpia para la gestión académica y se compara con la solución que brinda la hipótesis.
- Análisis – Síntesis: Para efectuar indagaciones bibliográficas acerca del tema de investigación que se identificó en revistas científicas.
- Hipotético - Deductivo: Para plantear la hipótesis de investigación.

Método Empírico:

- Modelación: Para representar gráficamente la propuesta de solución.

Técnicas de recolección de datos:

Para esta investigación se utilizaron las técnicas siguientes:

- La entrevista: Para obtener opiniones sobre la experiencia de los especialistas con el sistema Akademos.
- La encuesta: Para adquirir datos de interés que se utilizan en estimaciones poblacionales y toma de decisiones prácticas y estadísticas, para medir la variable dependiente e independiente.

Instrumentos:

El cuestionario es un instrumento básico de la observación en la encuesta y en la entrevista. Con él, se formulan una serie de preguntas que permiten medir una o más variables y posibilita observar los hechos a través de la valoración que hace de los mismos el encuestado o entrevistado extendiéndose la investigación a las valoraciones subjetivas de este.

Este trabajo estará estructurado por tres capítulos.

El primer capítulo contemplará la argumentación teórica del tema, donde se define el marco teórico y del modelo teórico de la investigación, estudio del estado del arte de los principales conceptos abordados durante la investigación. El capítulo dos contendrá la propuesta arquitectónica, con la descripción de los aspectos fundamentales para la solución, y el tercer capítulo la evaluación de la arquitectura propuesta.

Capítulo 1: Fundamentación teórica de la investigación

En este capítulo se definen los principales conceptos que serán empleados en el trabajo de investigación y se presenta el marco teórico fundamental relacionado con sistemas de gestión académica y arquitectura de software. Se esclarecen las definiciones de tipos de arquitecturas, patrones y estilos arquitectónicos. Además, se analiza la arquitectura por capas, sus características, su estructura y su aplicación en el sistema de gestión académica. Se realiza una caracterización de las herramientas y técnicas a emplear.

1.1 Conceptos generales

1.1.1 Sistema de Gestión Académica (GA)

El Sistema de Gestión Académica brinda mecanismos de control y gestión de la información en beneficio de los estudiantes, egresados, docentes y administrativos, logrando agilidad, oportunidad, seguridad y calidad en la información. En el marco de los planes de modernización y desarrollo institucional, el sistema de Gestión Académica se convierte en un medio eficaz para lograr los objetivos institucionales y de los programas, para facilitar la acreditación y en especial entregar productos y servicios de calidad a la comunidad [CITATION Ern14 \l 3082].

Los Sistemas de GA tienen como tarea fundamental almacenar y procesar toda la información referente al proceso docente de un centro de estudios. Por lo general engloban todo lo relacionado a la matrícula y prematricula de los estudiantes, al control de las evaluaciones y la asistencia de los mismos, así como otros datos académicos importantes, además abarca toda la gestión de los planes de estudio y de los horarios de clases y de la información de los profesores. Estos sistemas constituyen una poderosa herramienta de trabajo que le permiten a cualquier centro de estudios hacer eficientes sus procesos, optimizando así sus recursos [CITATION Yir05 \l 3082].

1.1.1.1 Soluciones existentes sobre Sistemas de GA

La Universidad de Talca, Chile se planteó el desarrollo del Sistema de Gestión de la Investigación (SGI) [CITATION Ivá07 \l 3082]. Con miras a apoyar la investigación que realizan sus académicos por la vía de un sitio Web que sirva de punto de encuentro entre la oferta investigativa de la universidad y la demanda de investigación de la sociedad y las empresas. Dicho sistema tuvo las necesidades siguientes:

- Mantener informados a interesados en los resultados de las investigaciones realizadas

- Ampliar la cantidad de servicios institucionales aportados

- Intercambiar conocimientos a nivel nacional e internacional que enriquezcan el bagaje cultural y profesional de los investigadores.

Por otro lado, está el sistema de gestión académica vía web para Institutos de investigación y posgrado implementado en la Facultad de ingeniería de la Universidad Central de Ecuador el cual permite realizar sus procesos de una manera estructurada y automatizada. El proyecto tiene necesidades tales como: el manejo de los datos en los programas de posgrado, el acceso al historial de programas en curso, calificaciones entre otros; todo esto con el propósito de apoyar la toma de decisiones en el instituto [CITATION Yag15 \l 3082].

También se encuentra el Campus Virtual de la Universidad de Barcelona (UB) donde tiene como objetivo dar apoyo a la docencia presencial y semipresencial de la UB. Lo que se pretende es impulsar la adaptación de las enseñanzas de las directrices del Espacio Europeo de Educación Superior (EEES), centrando la atención en el proceso de aprendizaje [CITATION Joa08 \l 3082].

Por otra parte, se tiene el Sistema de Gestión Académica Akademos de la Universidad de las Ciencias Informáticas[CITATION Mig21 \l 3082], del cual se trata nuestra investigación. XAUCE AKADEMOS es una herramienta multiplataforma que contribuye al perfeccionamiento de los procesos académicos de una institución. Su uso permite el desarrollo coherente de una estrategia organizacional en los diferentes niveles de la estructura educacional con la que cuenta el Ministerio de Educación de la República de Cuba.

Finalmente, los Sistemas de GA facilitan y mejoran los procesos formativos que imparten las instituciones de educación superior. Posibilitan que se pueda realizar procesos de manera estructurada y automatizada.

1.1.2 Mantenibilidad

La necesidad de utilizar un sistema de información por largo tiempo, conlleva a la decisión de someter o no el producto de software a mantenimiento. Este problema tiene varios puntos de vista, los que pueden ser analizados de manera individual. Por un lado, se encuentran los intereses de la empresa que desarrolló el producto de software, en otra perspectiva está la empresa que hace uso del software

y, por último, se encuentra el punto de vista de la empresa que realiza la mantención del software. Cada uno de estos actores es responsable de asegurar la continuidad del funcionamiento de un producto de software [CITATION Jen20 \l 3082].

1.1.2.1 Estándares de mantenibilidad

Los estándares de mantenibilidad representan la capacidad del producto software para ser modificado efectiva y eficientemente, debido a necesidades evolutivas, correctivas o perfectivas. Estos estándares se subdivide a su vez en las siguientes puntos[CITATION Fra01 \l 3082]:

- ✓ Modularidad. Capacidad de un sistema o programa de ordenador (compuesto de componentes discretos) que permite que un cambio en un componente tenga un impacto mínimo en los demás.

- ✓ Reusabilidad. Capacidad de un activo que permite que sea utilizado en más de un sistema software o en la construcción de otros activos.

- ✓ Capacidad de ser analizado. Facilidad con la que se puede evaluar el impacto de un determinado cambio sobre el resto del software, diagnosticar las deficiencias o causas de fallos en el software, o identificar las partes a modificar.

- ✓ Capacidad para ser modificado. Capacidad del producto que permite que sea modificado de forma efectiva y eficiente sin introducir defectos o degradar el desempeño.

- ✓ Capacidad para ser probado. Facilidad con la que se pueden establecer criterios de prueba para un sistema o componente y con la que se pueden llevar a cabo las pruebas para determinar si se cumplen dichos criterios.

La mantenibilidad se refiere a las medidas o pasos tomados durante la fase de diseño del producto. Esto, para incluir características que aumenten la facilidad de mantenimiento y garanticen el menor tiempo de inactividad y bajos costes de soporte del ciclo de vida [CITATION Ele18 \l 3082].

En definitiva, la “Mantenibilidad” es la facilidad con la que se modifica, mejora y/o adapta un producto software. Esta característica es identificada y definida por normas de calidad ampliamente aceptadas, que recomiendan establecer métricas para su evaluación [CITATION Ele18 \l 3082].

1.1.3 Arquitectura de Software

Un aspecto crítico a la hora de construir sistemas complejos es el diseño de la estructura del sistema, y por ello el estudio de la Arquitectura Software se ha convertido en una disciplina de especial relevancia en la ingeniería del software [CITATION Gus17 \l 3082].

La arquitectura de software es la representación de alto nivel de la estructura de un sistema, describe las partes que la integran, las interacciones entre ellas, los patrones que supervisan su composición y las restricciones que aplican esos patrones [CITATION Gus17 \l 3082]. Se trata de detallar cómo se implementa un proceso en un determinado granularidad, dados ciertos supuestos o requisitos. La calidad de una arquitectura puede entonces ser juzgado sobre la base de parámetros tales como su costo, la calidad de los resultados, su simplicidad o "elegancia", la cantidad de esfuerzo necesario para cambiarlo, y así sucesivamente [CITATION Ele18 \l 3082].

1.1.4 Arquitectura Limpia

La arquitectura limpia trata de capturar tanto la jerarquía conceptual de los componentes como el tipo jerarquía a través de un enfoque en capas. En particular, una arquitectura limpia es una estructura esférica, con capas internas (inferiores) completamente rodeadas por las externas (superiores), y siendo el primero ajeno a la existencia del segundo [CITATION Rob18 \l 3082]. Es una filosofía de diseño de software que separa los elementos de un diseño en niveles de anillo. La regla principal de la arquitectura limpia es que las dependencias de código solo pueden provenir de los niveles externos hacia adentro. El código en las capas internas no puede tener conocimiento de las funciones en las capas externas. Las variables, funciones y clases (cualquier entidad) que existen en las capas externas no pueden mencionarse en los niveles más internos. Se recomienda que los formatos de datos también permanezcan separados entre niveles [CITATION Rob18 \l 3082]. Se refiere a organizar el proyecto para que sea fácil de entender y cambiar a medida que el proyecto crece. Esto no sucede por casualidad. Se necesita planificación intencional [CITATION Rob18 \l 3082].

1.1.4.1 ¿Por qué Limpio?

La arquitectura tiene muchos nombres, pero el que está principalmente en uso hoy en día es “arquitectura limpia”. Este es el nombre utilizado por Martín (2018) en su libro donde él afirma claramente que esta estructura no es una novedad, sino que ha sido promovida por muchos diseñadores de software a través de los años. El adjetivo “limpio” describe uno de los aspectos

fundamentales tanto de la estructura del software y el enfoque de desarrollo de esta arquitectura. Es limpio, es decir, es fácil para entender lo que pasa [CITATION Leo20 \l 3082].

1.1.5 Tipos de Arquitecturas

Los tipos de arquitecturas son un conjunto de decisiones de diseño arquitectónico específicas que es aplicable a un problema de diseño recurrente, y parametrizado para considerar distintos contextos de desarrollo software en los que se presenta el mismo problema [CITATION Ele18 \l 3082].

Por lo tanto, un patrón arquitectónico o un tipo de arquitectura ofrece buenas soluciones de diseño reutilizables para un mismo problema en un contexto que ha sido verificado, validado y aceptado por la comunidad. La solución debe describir una relación entre los elementos y pueden estar caracterizados según el tipo de elementos arquitectónicos que utilizan [CITATION Ele18 \l 3082].

1.1.5.1 Arquitectura Orientada a Servicios (SOA)

La Arquitectura Orientada a Servicios modularizó el sistema de información en servicios. Con SOA, estos importantes programas se convierten en servicios comerciales. Con un único servicio comercial para una función determinada que se utiliza en todas partes de la organización. Cuando se necesita cambiar la política de negocios, se puede cambiar en un lugar y debido a que el mismo servicio se usa en todas partes, la consistencia se mantendrá en toda la organización. SOA permite a las empresas tomar decisiones comerciales respaldada por la tecnología en lugar de tomar decisiones comerciales determinadas o limitadas por la tecnología [CITATION Ele18 \l 3082].

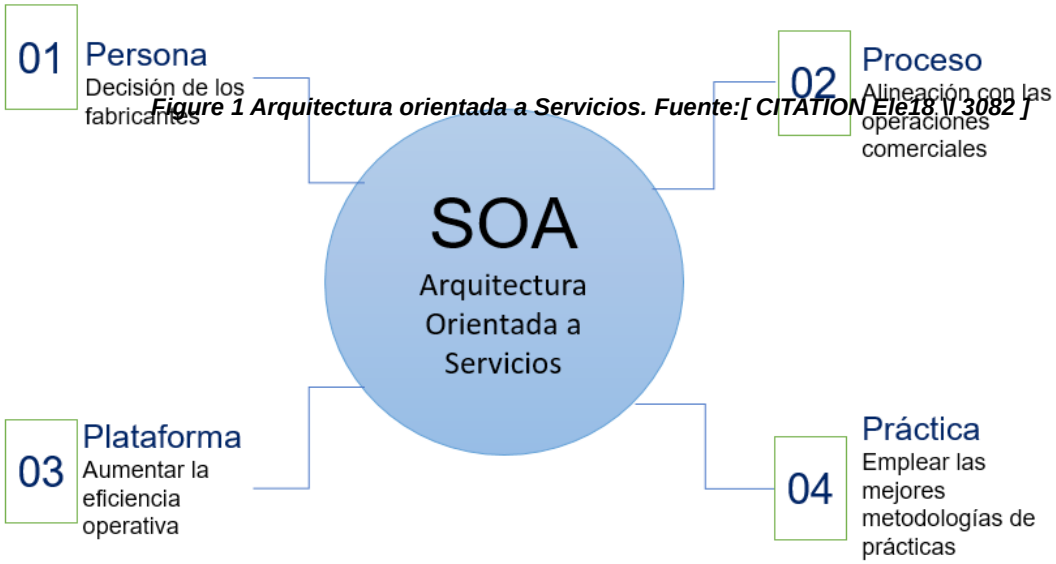


Figure 1 Arquitectura orientada a Servicios. Fuente:[CITATION Ele18 \l 3082]

1.1.5.2 Modelo-Vista-Controlador

El Modelo-Vista-Controlador MVC proporciona un nivel de separación entre la lógica empresarial y de presentación. El modelo se encarga de gestionar los datos de la aplicación, recibe información del controlador mientras la vista significa presentación del modelo en un formato particular, asimismo el controlador responde a la entrada de datos del usuario y realiza interacciones en los objetos del modelo de datos, luego el controlador recibe la entrada, opcionalmente la valida y luego pasa la entrada al modelo [CITATION Ele18 \l 3082].

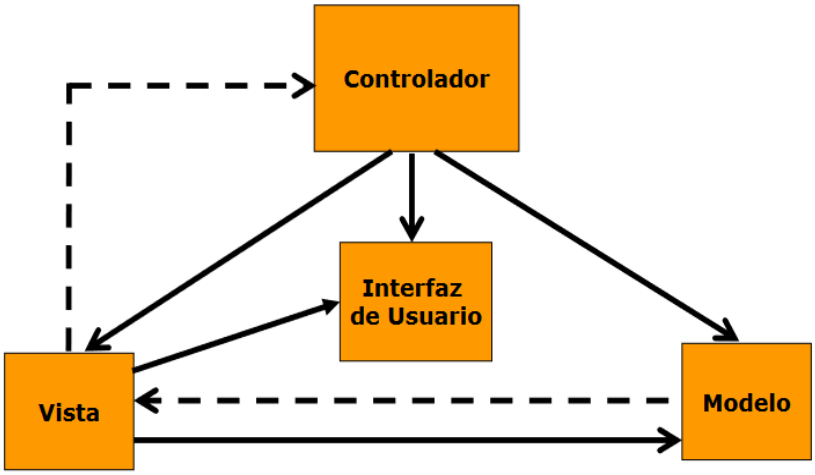


Figure 2 Modelo Vista Controlador. Fuente:[CITATION Ele18 \l 3082]

1.1.5.3 Arquitectura de Microservicios

La arquitectura de microservicios consta de una colección de pequeños servicios autónomos. Cada servicio es autónomo y debe implementar una única capacidad empresarial. Los microservicios son pequeños, independientes y poco acoplados. Un solo equipo pequeño de desarrolladores puede escribir y mantener un servicio. Cada servicio es una base de código separada, que puede ser administrada por un pequeño equipo de desarrollo. Los servicios se pueden implementar de forma independiente. Un equipo puede actualizar un servicio existente sin reconstruir y volver a implementar toda la aplicación [CITATION Ele18 \l 3082].

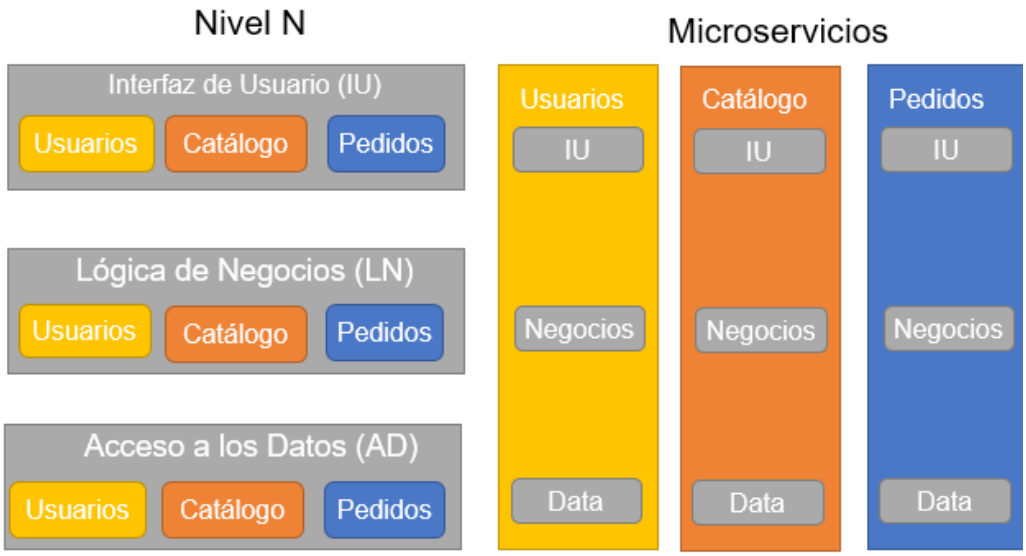


Figure 3 Microservicios. Fuente:[CITATION Rob18 \l 3082]

1.1.5.4 Arquitectura Basadas en Componentes

Arquitectura que enfatiza en la separación de preocupaciones con respecto a la amplia funcionalidad disponible en un sistema de software dado. Esta separación nos permite manejar problemas de arquitectura tanto a nivel de definición como restricciones (qué tipos de componentes y conectores pueden conectarse entre sí.) y en nivel de instancia como multiplicidad (un servidor puede estar conectado por 0 ... n clientes) y dinámico. Un componente puede tener su propia arquitectura interior. Tales componentes se llaman componentes compuestos. Con este concepto, podemos refinar la arquitectura gradualmente y hacer que el diseño sea más controlable. Además, el componente compuesto puede ser reutilizados y compuestos también: podemos reutilizar y componer artefactos de diseño a alto nivel [CITATION Ele18 \l 3082].

1.1.5.5 Arquitectura por Capas

Una arquitectura por capas ofrece como ventaja inmediata la capacidad de prueba. Los principios de diseño llamados "Clean Architecture" ayudan a implementar y utilizar con eficacia conceptos como la separación de preocupaciones, abstracción, implementación e inversión de control [CITATION Leo20 \l 3082].

Cada capa proporciona servicios a la capa superior y se sirve de las prestaciones que le brinda la inferior, al dividir un sistema en capas, cada capa puede tratarse de forma independiente, sin tener que conocer los detalles de las demás. La división de un sistema en capas facilita el diseño modular, en la que cada capa encapsula un aspecto concreto del sistema y permite además la construcción de sistemas débilmente acoplados, lo que significa que, si se minimiza las dependencias entre capas, resulta más fácil sustituir la implementación de una capa sin afectar al resto del sistema [CITATION Ore081 \l 3082].

Cada capa es un grupo de módulos que ofrece un conjunto de servicios, y expone su funcionalidad a través de una interfaz API. La relación entre capas es unidireccional siendo las capas de nivel más alto las que pueden usar los servicios de las capas adyacentes definidas más abajo, sin permitir comunicación circular. Este patrón está diseñado para satisfacer los requisitos de modularidad, portabilidad, mantenibilidad y reusabilidad del código en diversas aplicaciones [CITATION Ore081 \l 3082].

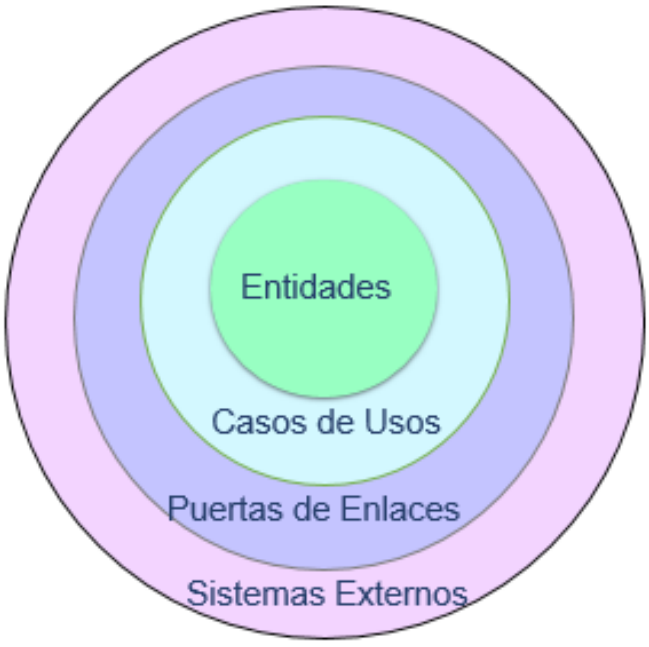


Figure 4 Arquitectura por Capas. Fuente:[CITATION Rob18 \l 3082]

Entidades: Esta capa de la arquitectura limpia contiene una representación de los modelos de dominio, es decir todo lo que su sistema necesita para interactuar. Esta capa contiene clases, con métodos que simplifican la interacción con ellos. Esta es la capa más interna, dichas entidades tienen conocimiento mutuo ya que viven en la misma capa, por lo que la arquitectura les permite interactuar directamente. La capa entidades proporciona una base sólida que las capas externas pueden usar para intercambiar datos [CITATION Leo20 \l 3082].

Casos de Usos: La parte más importante de un sistema limpio son los casos de uso, ya que implementan las reglas del negocio, que son la razón fundamental de la existencia del propio sistema. Los casos de uso deben ser lo más pequeños posible, es muy importante aislar las pequeñas acciones en casos de uso, ya que esto hace que todo el sistema sea más fácil de probar, comprender y mantener. Los casos de uso tienen acceso completo a la capa de entidades, para que puedan instanciarlas y sumarlas directamente [CITATION Leo20 \l 3082].

Puertas de Enlaces: Esta capa contiene componentes que definen interfaces para sistemas externos, que es un acceso común de modelo a servicios que no implementan las reglas del negocio. La capa de puertas de enlace está íntimamente conectada con los sistemas externos. Las puertas de enlace se utilizan para enmascarar la implementación de sistemas externos [CITATION Leo20 \l 3082].

Sistemas Externos: Esta capa de la arquitectura está poblada por componentes que implementan las interfaces definidas en la capa anterior. Los sistemas externos tienen acceso total a puertas de enlace, casos de uso y entidades. Un usuario al hacer clic en un botón, visitar una URL o ejecutar un comando son ejemplos típicos de interacciones con un sistema externo que ejecuta un caso de uso directamente [CITATION Leo20 \l 3082].

1.2 Estilos arquitectónicos

En el epígrafe anterior se definieron los patrones de diseño, que se utilizan para resolver problemas en un dominio y que son implementados y/o derivados en estilos arquitectónicos. Los conceptos de estilos y patrones arquitectónicos son similares y no siempre es posible es posible identificar un límite entre ellos. Los estilos están diseñados para capturar conocimiento de diseños específicos para lograr objetivos dentro de un particular contexto de aplicación, mientras que los patrones focalizan el diseño en la resolución de un problema concreto [CITATION Ele18 \l 3082].

Un estilo arquitectónico es el diseño de la aplicación a un alto nivel de abstracción, en concreto, un conjunto de reglas de diseño que identifica las clases de componentes y conectores que se pueden utilizar para componer el sistema o el subsistema. Una única arquitectura puede contener varios estilos arquitectónicos, y cada estilo puede utilizar varios patrones arquitectónicos. En la siguiente tabla se muestran algunos ejemplos de estilos arquitectónicos comunes y vigentes, conforme a ciertos criterios basados en el análisis realizado en el artículo Reynoso CKicillof N (2004) [CITATION Ele18 \l 3082].

Tabla 1: Estilos Arquitectónicos. Fuente: [CITATION Ele18 \l 3082]

Estilo Arquitectónico	Sub-estilos	Observaciones
Flujo de datos.	Tubería-filtros.	Implementan transformaciones de datos en pasos sucesivos. Reutilización y la modificabilidad.
Centrados en Datos.	Arquitecturas de Pizarra o Repositorio.	Sistemas de acceso y actualización de datos en estructuras de almacenamiento.
Estilos de Llamada y Retorno.	Model-View-Controller (MVC). Arquitecturas en Capas. Arquitecturas orientadas a Objetos. Arquitecturas Basadas en Componentes.	Estilos más generalizados en sistemas a gran escala. Enfatiza la modificabilidad y la escalabilidad.
Estilos de Código Móvil.	Arquitectura de Máquinas Virtuales.	Enfatiza la portabilidad.
Estilos Peer-to-Peer.	Arquitecturas Basadas en Eventos (Invocación implícita). Arquitecturas Orientadas a Servicios. Arquitecturas Basadas en Recursos.	Procesos independientes o entidades que se comunican a través de mensajes. Enfatiza la modificabilidad por medio de la separación de las diversas partes que intervienen en la computación.
Estilos heterogéneos.	Sistemas de control de procesos. Arquitecturas Basadas en Atributos.	Difícil categorizar estos estilos en familias de estilos existentes.

1.3 Sistemas Homólogos

Durante el desarrollo de la investigación se analizan sistemas y plataformas para la gestión y visualización del usuario. Durante el proceso de elección de estas, se tuvieron en cuenta que fueran sistemas académicos en Cuba como, por ejemplo: Entorno Virtual de Enseñanza y Aprendizaje de la

Universidad de la Habana y el Entorno Virtual de Enseñanza y Aprendizaje de la Universidad de Oriente.

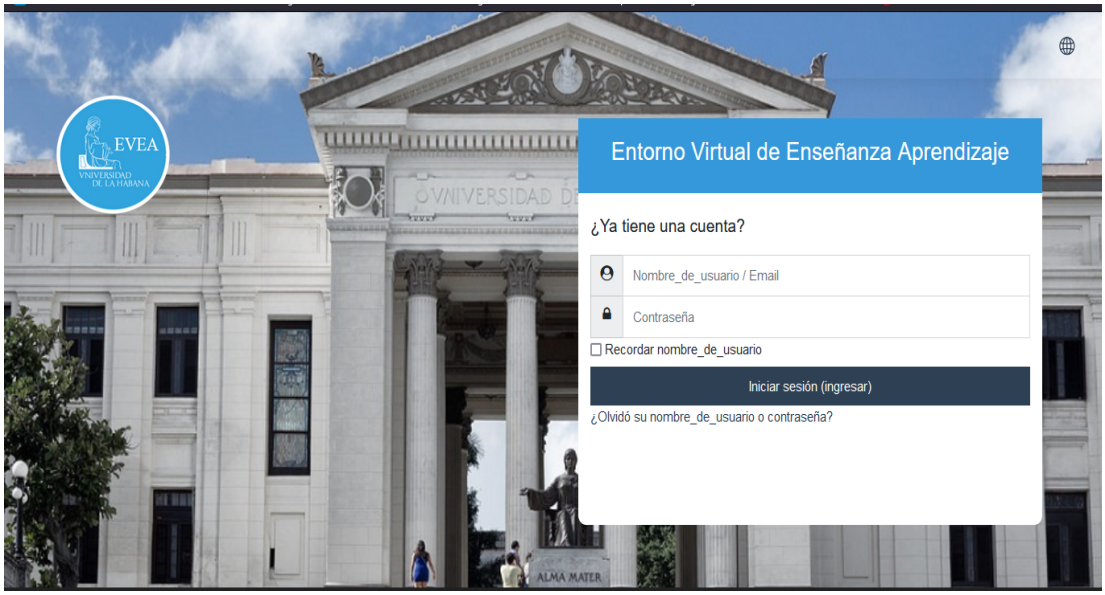


Figure 5 EVEA de la Universidad de La Habana. Fuente:[CITATION Jul21 \l 3082]

1.3.1 Entorno Virtual de Enseñanza y Aprendizaje (EVEA) de la Universidad de la Habana

El Entorno Virtual de Enseñanza y Aprendizaje de la Universidad de la Habana surgió hace algunos años. Al llegar la pandemia, y ante la necesidad del distanciamiento físico, la dirección de la UH orienta a sus profesores la implementación de las asignaturas en EVEA. Dicha plataforma incluye cursos de posgrado a distancia para la superación rápida de su claustro docente y para los estudiantes [CITATION Jul21 \l 3082].

1.3.2 El EVEA de la Universidad de Oriente

El Entorno Virtual de Enseñanza y Aprendizaje de la Universidad de Oriente también se desarrolló durante el período de la Covid-19. En dicho entorno se desarrolla un conjunto de actividades y acciones en la modalidad de trabajo a distancia, que inciden en los procesos esenciales: Docencia, Investigación y extensionismo [CITATION gus21 \l 3082].

Figure 6 Entorno Virtual de la Universidad de oriente. Fuente:[CITATION gus21 \l 3082]

Sistemas de Gestión	Definición	Arquitectura
UH	Evea para la gestión de asignaturas.	Monolítica.
UO	Evea para la gestión de asignaturas.	Monolítica.
UCI	Sistema de Gestión Académica para el MINED	Monolítica y Microservicios.

Los entornos virtuales de la Universidad de la Habana y de la Universidad de Oriente surgieron en la etapa del COVID-19 debido a la necesidad de informatizar la información y que se diera continuidad al proceso de estudio. Sin embargo, el sistema Akademos de la Universidad de las Ciencias Informáticas surge desde hace más de diez años y gestiona desde los planes de estudio de cada estudiante hasta cada dato e información importante de los docentes. Su arquitectura, aunque es muy parecida a los otros sistemas nacionales, tiene una gran diferencia porque está dividida en monolitos, tiene el despliegue en microservicios y se tiene una propuesta en Arquitectura Limpia. Por tanto, aunque tiene algunas desventajas y deficiencias, es un mejor software para la gestión académica.

1.4 Ambiente de desarrollo

En las siguientes secciones se describe la metodología de desarrollo de software, los lenguajes de programación usados, las tecnologías y herramientas usadas.

1.4.1 Metodología de desarrollo de software

Resulta necesario establecer un enfoque sistemático y disciplinario para desarrollar un software. El uso de una metodología permite el dominio del proceso descrito. Una metodología es un enfoque, una manera de interpretar la realidad o la disciplina en cuestión, que en caso particular correspondía a la ingeniería del software. Establece el orden en el que la mayoría de las actividades tienen que realizarse y los enlaces entre ellas. Indica cómo tienen que realizarse algunas tareas proporcionando las herramientas concretas e intelectuales [CITATION Dry17 \l 3082].

Se utilizó como metodología de desarrollo AUP-UCI, puesto que es la metodología del sistema Akademos. El Proceso Unificado Ágil de Scott Ambler o Agile Unified Process (AUP) en inglés es una versión simplificada del Proceso Unificado de Rational (RUP). Este describe de una manera simple y fácil de entender la forma de desarrollar aplicaciones de software de negocio usando técnicas ágiles y conceptos que aún se mantienen válidos en RUP. La Universidad de las Ciencias Informáticas (UCI) ha definido una variación de la metodología AUP en unión con el modelo CMMI-DEV v 1.3, para que pueda emplearse en los proyectos productivos de la Universidad de las Ciencias Informáticas (UCI) [CITATION Dry17 \l 3082].

1.4.1.1 Disciplinas de la metodología Variación AUP-UCI

Modelado de negocio: Destinado a comprender los procesos de negocio de una organización. Se define cómo funciona el negocio que se desea informatizar para tener garantías de que el software desarrollado va a cumplir su propósito. Para modelar el negocio se proponen las siguientes variantes: Casos de Uso del Negocio (CUN), Descripción de Proceso de Negocio (DPN) y Modelo Conceptual (MC) [CITATION Dry17 \l 3082].

Requisitos: El esfuerzo principal en la disciplina Requisitos es desarrollar un modelo del sistema que se va a construir. Esta disciplina comprende la administración y gestión de los requisitos funcionales y no funcionales del producto. Existen tres formas de encapsular los requisitos: Casos de Uso del Sistema (CUS), Historias de usuario (HU) y Descripción de requisitos por proceso (DRP), agrupados en cuatro escenarios condicionados por el Modelado de negocio [CITATION Dry17 \l 3082].

Análisis y diseño: Los requisitos pueden ser refinados y estructurados para conseguir una comprensión más precisa de estos, y una descripción que sea fácil de mantener y ayude a la estructuración del sistema (incluyendo su arquitectura). Además, en esta disciplina se modela el sistema y su forma (incluida su arquitectura) para que soporte todos los requisitos, incluyendo los requisitos no

funcionales. Los modelos desarrollados son más formales y específicos que el de análisis [CITATION Dry17 \l 3082].

A partir de que el Modelado de negocio propone tres variantes a utilizar en los proyectos (CUN, DPN o MC) y existen tres formas de encapsular los requisitos (CUS, HU, DRP), surgen cuatro escenarios para modelar el sistema en los proyectos, manteniendo en dos de ellos el MC, quedando de la siguiente forma [CITATION Dry17 \l 3082].

Escenario No 1: Aplica a los proyectos que hayan evaluado el negocio a informatizar y como resultado obtengan que puedan modelar una serie de interacciones entre los trabajadores del negocio/actores del sistema (usuario), similar a una llamada y respuesta respectivamente, donde la atención se centra en cómo el usuario va a utilizar el sistema [CITATION Tam14 \l 3082].

Escenario No 2: Aplica a los proyectos que hayan evaluado el negocio a informatizar y como resultado obtengan que no es necesario incluir las responsabilidades de las personas que ejecutan las actividades, de esta forma modelarían exclusivamente los conceptos fundamentales del negocio[CITATION Tam14 \l 3082].

Escenario No 3: Aplica a los proyectos que hayan evaluado el negocio a informatizar y como resultado obtengan un negocio con procesos muy complejos, independientes de las personas que los manejan y ejecutan, proporcionando objetividad, solidez, y su continuidad [CITATION Tam14 \l 3082].

Escenario No 4: Aplica a los proyectos que hayan evaluado el negocio a informatizar y como resultado obtengan un negocio muy bien definido. El cliente estará siempre acompañando al equipo de desarrollo para convenir los detalles de los requisitos y así poder implementarlos, probarlos y validarlos [CITATION Tam14 \l 3082].

1.4.2 Lenguajes

Los lenguajes definidos para el proyecto son:

- Python v3.10: es un lenguaje de programación potente y fácil de aprender. Tiene estructuras de datos de alto nivel eficientes y un simple pero efectivo sistema de programación orientado a objetos. La elegante sintaxis de Python y su tipado dinámico, junto a su naturaleza interpretada lo convierten en un lenguaje ideal para scripting y desarrollo rápido de aplicaciones en muchas áreas, para la mayoría de plataformas [CITATION Ive14 \l 3082].

- UML, el Lenguaje Unificado de Modelado (UML) fue creado para forjar un lenguaje de modelado visual común y semántica y sintácticamente rico para la arquitectura, el diseño y la implementación de sistemas de software complejos, tanto en estructura como en comportamiento [CITATION Spa00 \l 3082]. Se hizo un diagramado UML a través de una extensión de Visual Studio Code llamado Drawio.

1.4.3 Tecnologías

- Django v3.1.5: es un marco web Python de alto nivel que fomenta un desarrollo rápido y un diseño limpio y pragmático. Creado por desarrolladores experimentados, es gratis y de código abierto. Fue diseñado para ayudar a los desarrolladores a llevar las aplicaciones desde el concepto hasta su finalización lo más rápido posible. Django se toma la seguridad en serio y ayuda a los desarrolladores a evitar muchos errores de seguridad común. Algunos de los sitios más concurridos de la web aprovechan la capacidad de Django para escalar de forma rápida y flexible[CITATION jos15 \l 3082].

1.4.4 Herramientas

- Visual Studio Code v1.68: es un editor de código fuente ligero pero potente que se ejecuta en su escritorio y está disponible para Windows, macOS y Linux. Viene con soporte incorporado para JavaScript, TypeScript y Node.js y tiene un rico ecosistema de extensiones para otros lenguajes (como C++, C#, Java, Python, PHP, Go) y tiempos de ejecución (como .NET y Unity).

- GITLAB v15.3.1: es una herramienta que permite la creación limitada de proyectos, en los cuales se pueden alojar los trabajos colaborativos que los estudiantes deban desarrollar, donde los mismos son monitoreados y evaluados de manera individual y colectiva[CITATION Áng19 \l 3082].

- Git v2.36: es un sistema de control de versiones distribuido gratuito y de código abierto diseñado para manejar todo, desde proyectos pequeños hasta proyectos muy grandes, con rapidez y eficiencia. Git es fácil de aprender y ocupa poco espacio con un rendimiento ultrarrápido [CITATION jos15 \l 3082].

1.5 Conclusiones parciales

Como parte del desarrollo del presente capítulo se determinan las siguientes conclusiones:

- ✓ El estudio de un grupo de definiciones asociados a la arquitectura limpia permitió un mejor entendimiento para la investigación.
- ✓ Fueron establecidas las herramientas y tecnologías necesarias para la correcta implementación de la arquitectura limpia.

Capítulo 2: Propuesta de solución

En el presente capítulo se realiza la descripción de la propuesta de solución y se detallan los principales requerimientos no funcionales de la aplicación. Además, se explica el uso de cada capa de la arquitectura y su implementación. Posteriormente se diseñan los diagramas de paquete y diagrama de clases, y por último se muestran los patrones de diseño del sistema.

2.1 Descripción de la propuesta de solución

La propuesta de solución consiste en arquitectura limpia por capas que permitirá mejorar la mantenibilidad, la cambiabilidad y la facilidad de prueba para el sistema Akademos. El mismo cuenta con el desarrollo del backend y tiene dos roles (Administrador y Usuario). El Administrador es el encargado de gestionar los usuarios que interactúan con el sistema y el usuario tendrá acceso a sus datos de estudiante mediante una autenticación.

Como el sistema de gestión académica XAUCE AKADEMOS para el MINED ya está implementado, posteriormente se muestra la arquitectura en capas teniendo el diseño y el código funcional del sistema. Se migrará el código del backend para dicha arquitectura quedando un sistema mantenible y mejor organizado.

2.2 Metodología AUP-UCI

El Sistema de Gestión Académica XAUCE AKADEMOS está enfocado a la metodología AUP-UCI en el escenario 3.

2.2.3 Escenario No. 3 DPN + MC = DRP

Aplica a los proyectos que hayan evaluado el negocio a informatizar y como resultado obtengan un negocio con procesos muy complejos, independientes de las personas que los manejan y ejecutan, proporcionando objetividad, solidez, y su continuidad. Se debe tener presente que este escenario es muy conveniente si se desea representar una gran cantidad de niveles de detalles y la relaciones entre los procesos identificados.

La Descripción de Procesos de Negocio (DPN) es donde se modela el negocio. En ella se generaron 16 requisitos funcionales y 6 requisitos no funcionales del sistema del módulo Seguridad. Después se da paso al Modelo Conceptual (MC) donde se describe el entendimiento común alcanzado por los involucrados respecto a los objetivos/conceptos del dominio relacionado con los elementos de

seguridad y acceso a la información en el sistema Akademos MINED. Los dos artefactos anteriores dan como resultado 14 Descripciones de Requisitos por Procesos (DRP) donde se describen las acciones que se realizan en el requisito funcional. A continuación, los artefactos que modelan dicho escenario.

2.3 Requisitos del sistema

“Los requerimientos para un sistema son la descripción de los servicios proporcionados por el sistema y sus restricciones operativas. Estos requerimientos reflejan las necesidades de los clientes de un sistema que ayude a resolver algún problema como el control de un dispositivo, hacer un pedido o encontrar información’’ [CITATION Roi16 \l 3082].

2.3.1 Requisitos funcionales del sistema

“Los requisitos funcionales son declaraciones de los servicios que debe proporcionar el sistema, de la manera en que este debe reaccionar a entradas particulares y de cómo se debe comportar en situaciones particulares’’ [CITATION Roi16 \l 3082]. Para el desarrollo de Akademos en el módulo de seguridad, se levantaron los siguientes requisitos funcionales (RF):

Tabla 2. Requisitos Funcionales del sistema. Fuente: [CITATION Mai21 \l 3082]

No	Nombre	Descripción	Prioridad	Complejidad	Referencias cruzadas
RF1.	Listar permisos	El sistema debe permitir mostrar los permisos existentes en el sistema.	Alta	Baja	
RF2.	Incluir permiso	El sistema debe permitir la inclusión de un permiso nuevo.	Alta	Baja	
RF3.	Modificar permisos	El sistema debe permitir la modificación de un permiso existente.	Alta	Baja	
RF4.	Listar roles	El sistema debe permitir mostrar los roles existentes en el sistema.	Alta	Baja	
RF5.	Incluir rol	El sistema debe permitir que se registre un nuevo rol en el sistema.	Alta	Baja	
RF6.	Modificar rol	El sistema debe permitir la modificación de un rol existente.	Alta	Baja	
RF7.	Ver datos de un rol	El sistema debe permitir ver los datos de un rol existente.	Baja	Baja	

RF8.	Listar usuarios	El sistema debe permitir mostrar los usuarios existentes en el sistema.	Alta	Baja	
RF9.	Incluir usuario	El sistema debe permitir que se registre un nuevo usuario.	Alta	Baja	
RF10.	Modificar un usuario	El sistema debe permitir que se modifique un usuario existente.	Alta	Baja	
RF11.	Ver datos de un usuario	El sistema debe permitir ver los datos de un usuario existente.	Baja	Baja	
RF12.	Autenticar	El sistema debe permitir que un usuario acceda al sistema haciendo uso de sus credenciales de acceso.	Alta	Baja	
RF13.	Cerrar sesión	El sistema debe permitir un usuario autenticado cierre su sesión.	Alta	Baja	
RF14.	Modificar contraseña olvidada	El sistema debe permitir que un usuario pueda obtener una nueva contraseña sin acceder al sistema.	Alta	Baja	
RF15.	Listar usuarios con permisos en estructuras	El sistema debe permitir que se listen los usuarios a los cuales se le han asociado las estructuras.	Alta	Baja	
RF16.	Asociar permisos a usuarios en estructuras	El sistema debe permitir que se asocien a los usuarios estructuras, para que puedan tener acceso a las funcionalidades del negocio.	Alta	Baja	

2.3.2 Requisitos no funcionales del sistema

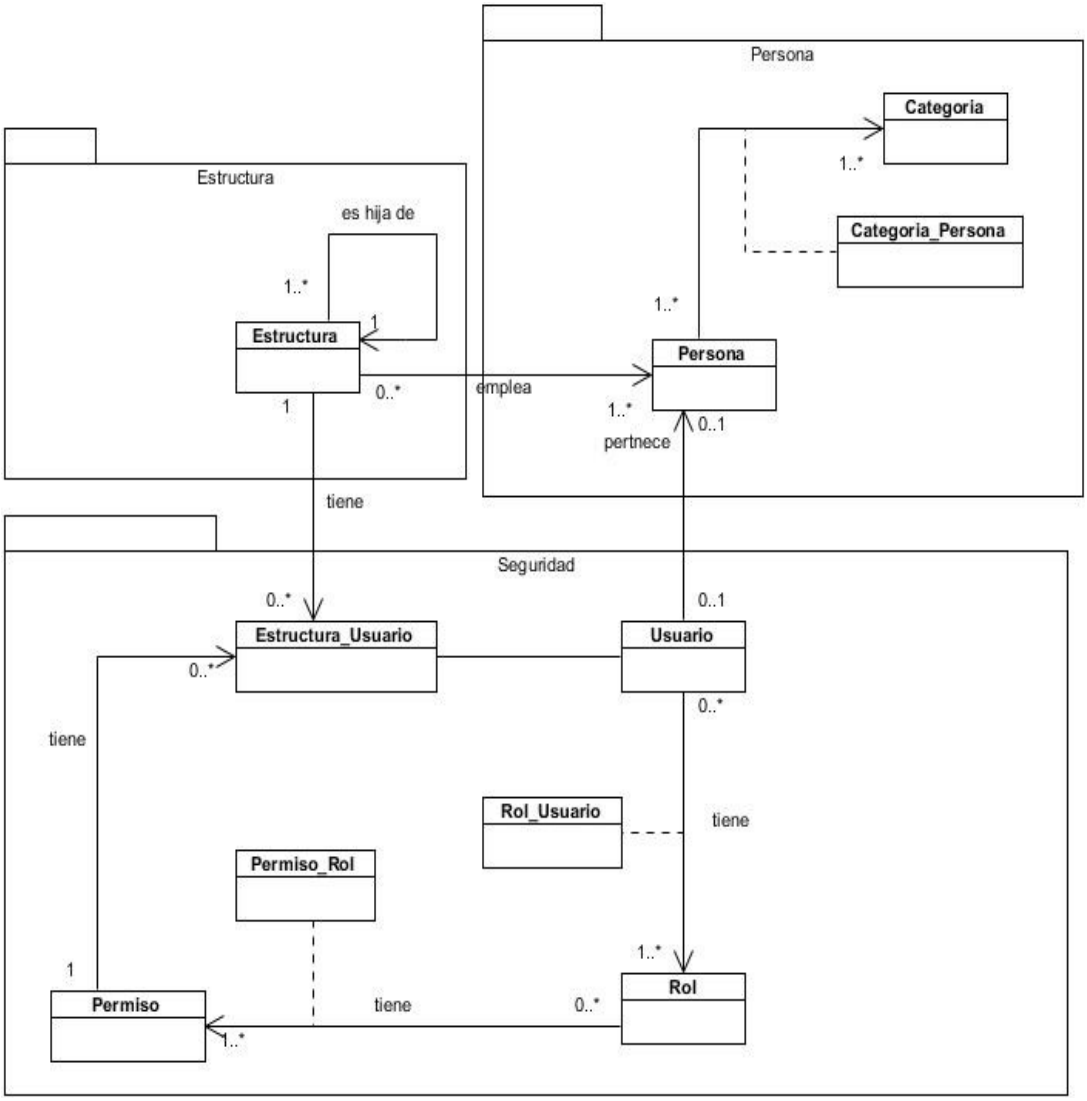
“Los requerimientos no funcionales son restricciones de los servicios o funciones ofrecidos por el sistema. Incluye restricciones de tiempo, sobre el proceso de desarrollo y estándares” [CITATION Roi16 \l 3082].Para el desarrollo de la aplicación se capturaron los siguientes requisitos no funcionales (RnF):

Tabla 3. Requisitos no Funcionales. Fuente: [CITATION Mig21 \l 3082]

No.	Tipo de RnF	Descripción
RnF. 1	Eficiencia de desempeño.	El sistema debe proporcionar un tiempo temporal de respuesta del sistema al realizar sus funciones

		bajo ciertas condiciones. El sistema debe asegurar que el límite máximo de peticiones simultáneas al sistema cumple con los requisitos. El sistema debe asegurar que el límite máximo de usuarios concurrentes del sistema cumple con los requisitos.
RnF. 2	Usabilidad.	El sistema debe cumplir con las pautas de diseño establecidas en la Estrategia Marcaria de la UCI. El sistema debe garantizar la adecuada visualización e interacción en múltiples dispositivos de acceso. El sistema debe permitir al usuario operar y controlar el sistema con facilidad. El sistema debe proteger a los usuarios de cometer errores durante el uso del sistema. El sistema debe permitir al usuario aprender a reconocer las acciones en el sistema. El sistema debe informar al usuario de errores del sistema.
RnF. 3	Seguridad.	El sistema debe verificar la la identidad de un sujeto o un recurso. El sistema debe prevenir el acceso y modificación no autorizada de los datos almacenados en el sistema. El sistema debe garantizar la continuidad del funcionamiento del sistema ante la entrada de datos inmóviles. El sistema debe proteger los datos y la información contra accesos autorizados.
RnF. 4	Funcionalidad.	El sistema debe proveer resultados correctos con el nivel de precisión requerido.
RnF. 5	Mantenibilidad.	El sistema debe evaluar el impacto de un determinado cambio sobre el resto del software a partir del diagnóstico de las deficiencias o causas de fallos en el sistema y la identificación de las partes a modificar. El sistema debe asegurar que un cambio en un componente tenga un impacto mínimo en los demás.
RnF. 6	Portabilidad.	El sistema debe garantizar que se pueda utilizar el sistema de forma exitosa.

2.4 Modelo Conceptual de Seguridad



2.5 Descripción de Requisitos por Procesos

Tabla 4. DRP de Autenticar. Fuente: [CITATION Mai211 \l 3082]

Precondiciones	El usuario debe estar registrado en el sistema. El usuario debe poseer los permisos para la funcionalidad.
Flujo de eventos	
Flujo básico Autenticar	

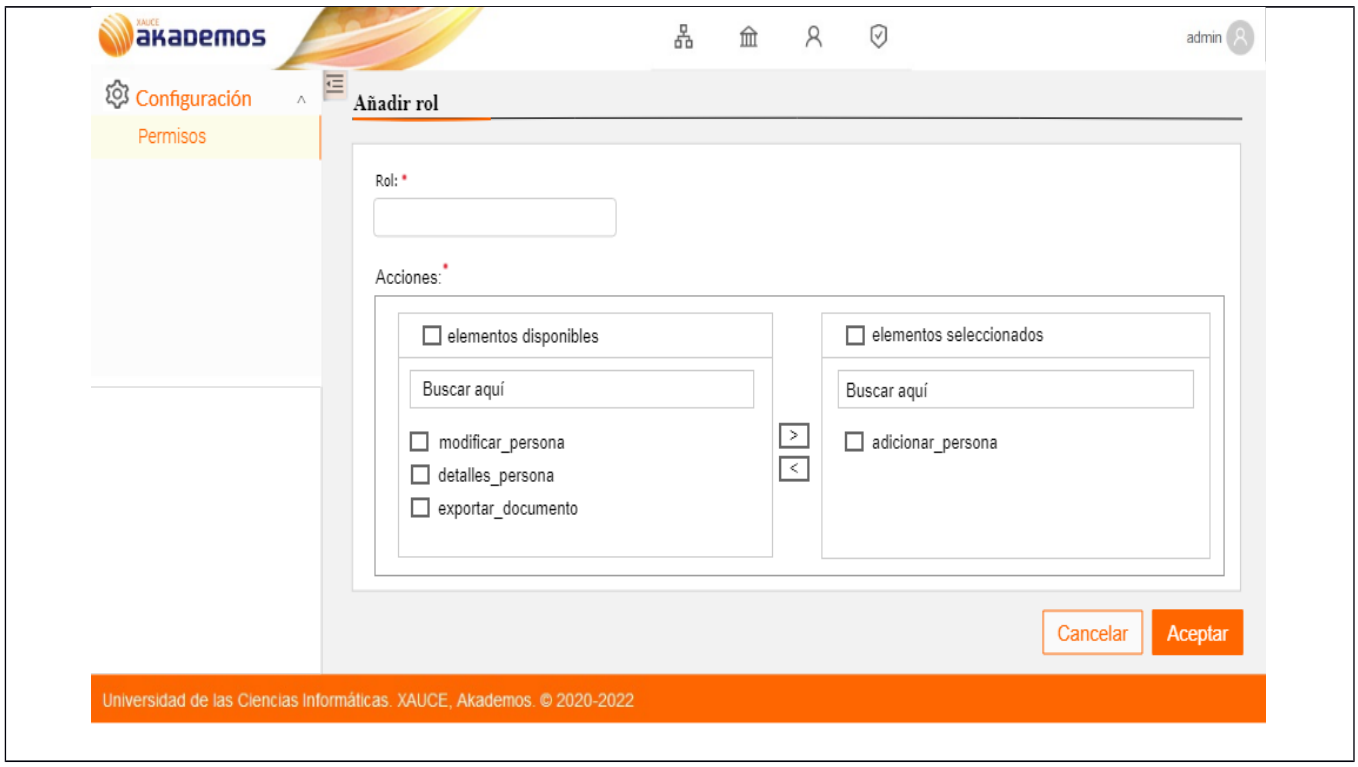
1.	Se introducen los datos. Se selecciona una de las siguientes opciones: • **Botón Iniciar sesión:** Ver Flujo alternativo 1.a. • **Opción** ¿Se te olvidó tu contraseña?**:** Ver FORTES_Levantamiento_AKADEMOS_MINED_Descripción_de_requisitos_por_proceso_Modificar_contraseña_olvidada.odt	
2.	Se valida que los campos obligatorios estén llenos. • En caso de no estar llenos se muestra el mensaje debajo del campo y en rojo: ¨Campo requerido¨. Al comenzar a escribir en el campo se elimina el mensaje. • En caso de estar llenos, Ver Flujo básico, Paso 3.	
3.	Se valida que el usuario tenga permisos a la funcionalidad y que tanto este como la contraseña estén correctos. • En caso de que los datos insertados estén correctos y exista el usuario, se pasa a la página de inicio del sistema. • En caso de no estar registrado el usuario o que estén los datos ingresados incorrectos, se muestra el mensaje: ¨ Usuario o contraseña incorrecta ¨.	
Pos-condiciones		
1.	Se pasa a la página de inicio del sistema.	
Flujos alternativos		
Flujo alternativo 1.a Botón Iniciar sesión.		
1	Se introducen los datos correspondientes (usuario y contraseña)	
2	Se selecciona el botón **Iniciar sesión**. Se pasa al paso 3 del flujo básico	
4	Se pasa al Paso 1 del Flujo básico.	
Pos-condiciones		
1	N/A	
Validaciones		
1	Componente de texto Usuario	Nombre de usuario de la persona. Debe tener el siguiente formato Usuario@uci.cu
2	Componente de texto Contraseña	Contraseña con la que puede acceder el usuario al sistema. Admite hasta 20 caracteres.
Conceptos	**Autenticar**	**Visibles en la interfaz:** - usuario - contraseña **Utilizados internamente**: - N/A
Requisitos especiales	N/A	
Asuntos pendientes	N/A	

Tabla 4. DRP del Incluir rol. Fuente: [CITATION Mai211 \l 3082]

Precondiciones	
Flujo de eventos	
Flujo básico Incluir rol	
4.	Se selecciona la funcionalidad Seguridad/Rol/Añadir rol.
5.	Se introducen los datos. Se selecciona una de las siguientes opciones: • **Botón Cancelar:** Ver Flujo alternativo 2.a. • **Botón Aceptar:** Ver Flujo básico, Paso 3.
6.	Se valida que los campos obligatorios estén llenos. • En caso de no estar llenos se muestra el mensaje debajo del campo y en rojo: ¨Campo requerido¨. Al comenzar a escribir en el campo se elimina el mensaje. • En caso de estar llenos, Ver Flujo básico, Paso 4.
7.	Se valida que los valores introducidos en los campos sean válidos. • En caso de no estarlo se muestra el mensaje debajo del campo y en rojo los mensajes correspondientes. Al comenzar a escribir en el campo valores válidos se elimina el mensaje. • En caso de estar correctos, Ver Flujo básico, Paso 5.

8.	Se valida que no exista un rol con el mismo nombre en el sistema. • En caso de existir, se muestra el mensaje: ¨ El elemento que trata de registrar ya existe. ¨, señalando en rojo el campo Rol. • En caso de no existir, se guardan los datos en **Rol** y **Permiso_Rol**, Se muestra el mensaje: ¨ La acción ha sido realizada satisfactoriamente. ¨ Se pasa a FORTES_Levantamiento_AKADEMOS_MINED_Descripcion_de_requisitos_por_proceso_Listar_roles.odt
Pos-condiciones	
2.	Queda registrado el rol.
Flujos alternativos	
Flujo alternativo 2.a Botón Cancelar.	
1	Se muestra el mensaje: ¿Está seguro que desea cancelar la operación?
2	Se selecciona lo opción: • **Botón Cancelar:** Se cierra el mensaje y se mantiene en la misma vista. • **Botón Aceptar:** Se cierra el mensaje y se pasa a FORTES_Levantamiento_AKADEMOS_MINED_Descripcion_de_requisitos_por_proceso_Listar_roles.odt
3	Se pasa al Paso 2 del Flujo básico.
Pos-condiciones	
1	N/A
Validaciones	
3	Campo de texto Rol
4	Componente de selección multiple Acciones

Conceptos	**Rol**
	Permiso
	Rol_Permiso
Requisitos especiales	N/A
Asuntos pendientes	N/A
Prototipo elemental de interfaz gráfica de usuario	



2.6 Modelo de diseño

El modelo de diseño es la organización o estructura de los componentes importantes que interactúan en el mismo. Durante el desarrollo de un software frecuentemente se encuentran distintos puntos de vistas entre el equipo de trabajo responsable de la construcción de un sistema, la arquitectura es el instrumento encargado de unificar estas diferencias. La misma tiene como propósito principal brindar elementos que ayuden a la toma de decisiones y proporcionar un lenguaje común [CITATION Roi16 \l 3082].

Para el desarrollo de la Arquitectura propuesta se seleccionó el patrón arquitectónico Modelo de “4+1” Vistas. El modelo 4+1 describe la arquitectura del software usando cinco vistas concurrentes. Tal como se muestra en la Ilustración 7, cada vista se refiere a un conjunto de intereses de diferentes stakeholders del sistema.

- La vista lógica describe el modelo de objetos del diseño cuando se usa un método de diseño orientado a objetos. Para diseñar se puede usar un enfoque alternativo para desarrollar algún otro tipo de vista lógica, tal como diagramas de entidad-relación.
- La vista de procesos describe los aspectos de concurrencia y sincronización del diseño.

- La vista física describe el mapeo del software en el hardware y refleja los aspectos de distribución.
- La vista de desarrollo describe la organización estática del software en su ambiente de desarrollo.

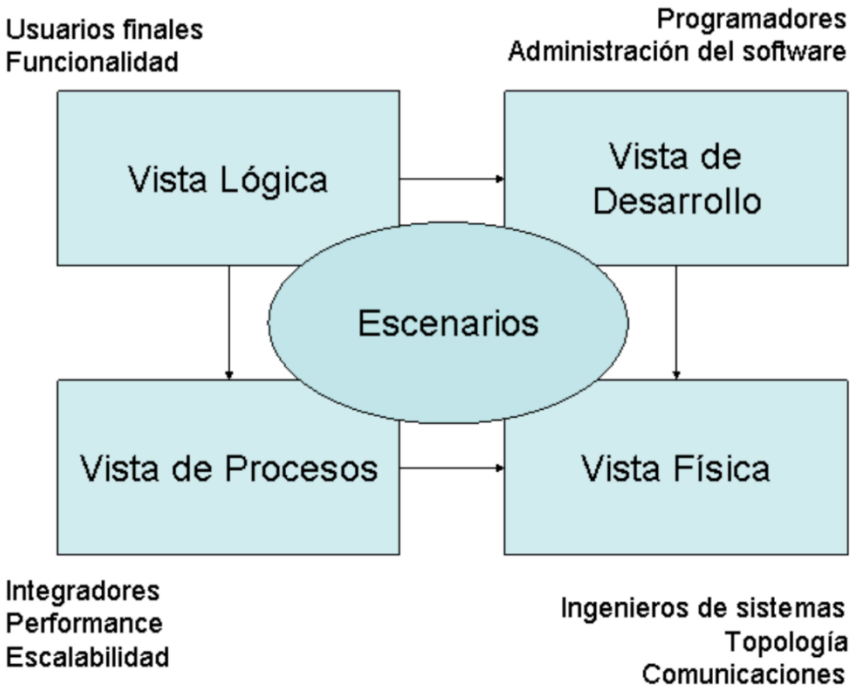


Figure 7 Modelo 4+1 Vista

En el SGA XAUCE AKADEMOS para el MINED ya se tiene dicho modelo puesto que en esta investigación se modelan los diagramas correspondientes para la Arquitectura Limpia por capas.

2.7 Diagrama de paquetes

Un paquete de diseño es una colección de clases, relaciones, diagramas y otros paquetes que están de alguna forma relacionados. Es utilizado para estructurar el modelo de diseño dividiéndolo en partes más pequeñas.

Los paquetes de diseño deben utilizarse fundamentalmente como herramienta organizacional del modelo para agrupar elementos relacionados. Pueden ser usados en cualquier nivel, desde el nivel más alto, donde son usados para dividir el sistema en dominios, hasta el nivel más bajo, donde son

usados para agrupar casos de usos individuales, clases, o componentes. A continuación, se muestra el diagrama de paquetes del sistema.

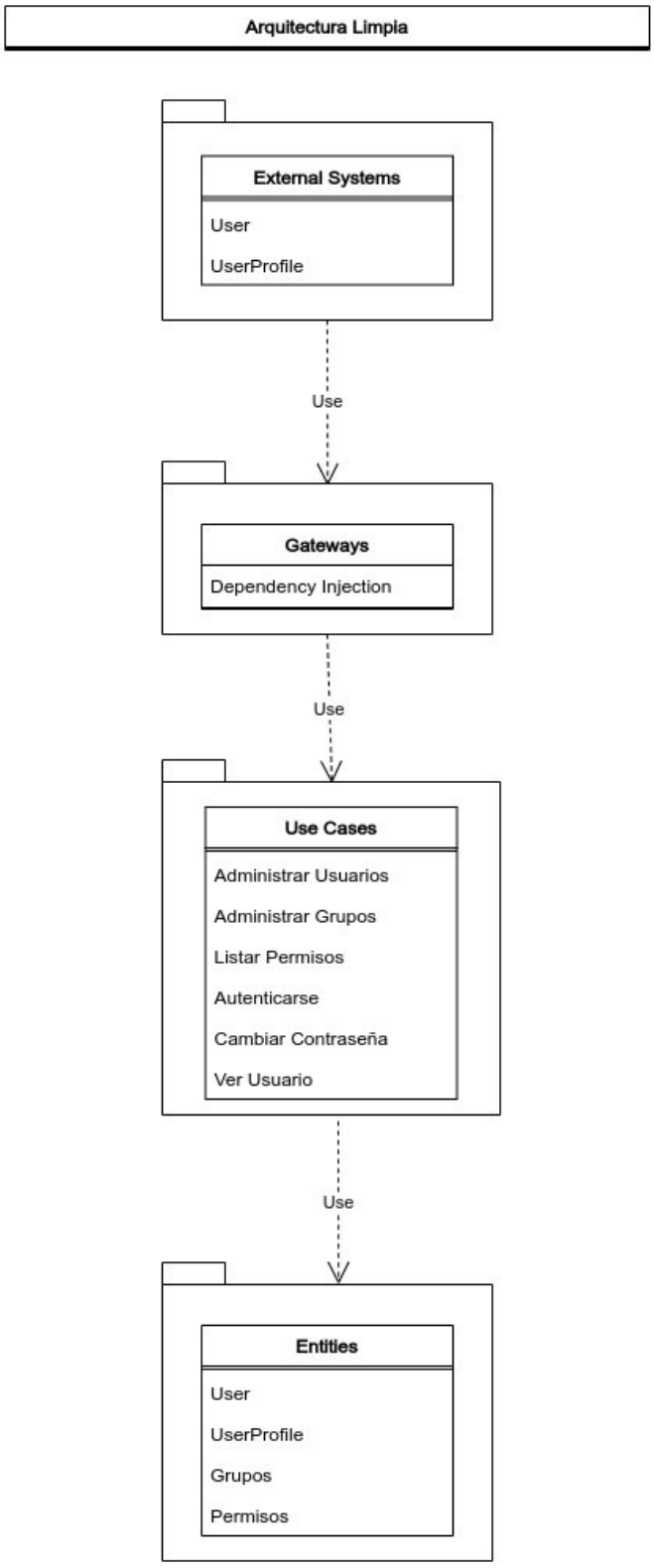


Figure 8 Diagrama de Paquetes. Fuente: Elaboración propia.

2.8 Diagrama de Clases

Un tipo de diagrama de vital importancia para la definición del sistema es el diagrama de clases. En este diagrama se representan las entidades de forma estática en forma de clases. Una clase puede contener atributos, propiedades y métodos [CITATION Roi16 \l 3082]. A continuación, se muestra una parte del diagrama de clases del módulo Usuario.

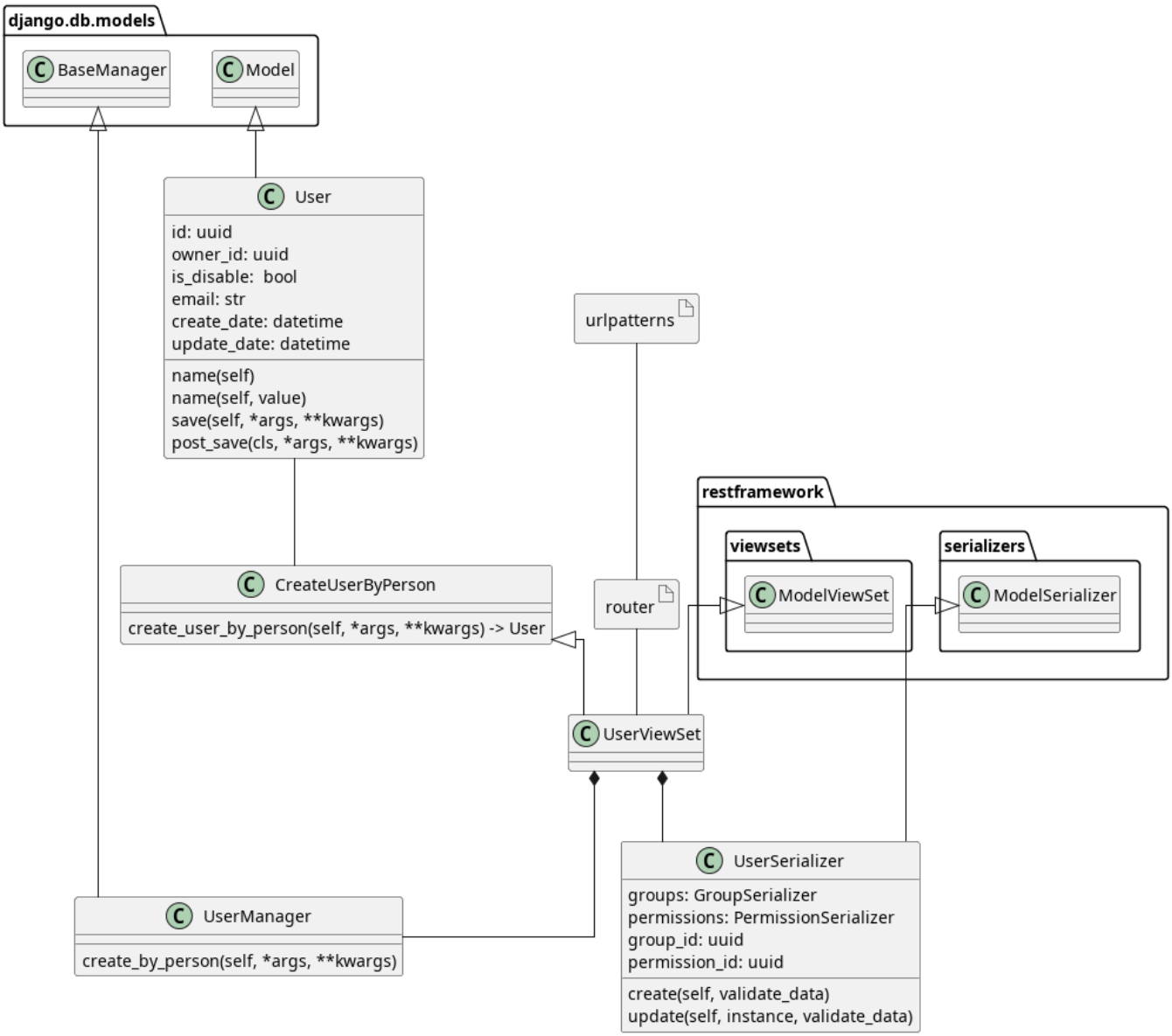


Figure 9: Diagrama de Clases del módulo Usuario. Fuente: Elaboración propia.

2.9 Conclusiones parciales

Como parte del desarrollo del presente capítulo se determinan las siguientes conclusiones:

- ✓ Se describió la Arquitectura Limpia por Capas en la parte del *Backend* para el SGA XAUCE AKADEMOS para el MINED.
- ✓ El modelado del negocio se realizó haciendo uso de la descripción de procesos del negocio y el modelo conceptual, mientras que el modelado del sistema se desarrolló mediante la descripción de requisitos por procesos, de acuerdo al escenario seleccionado de la metodología AUP-UCI.
- ✓ Se definió como modelo de diseño “4+1” Vistas y como patrones de diseño utilizados, experto, bajo acoplamiento, alta cohesión e inyección de dependencia dentro de los GRASP.

Capítulo 3: Validación de la propuesta de solución

Durante todo proceso de desarrollo de software es necesario comprobar la calidad del mismo. Primeramente, se realiza la implementación para luego pasar a realizar un conjunto de pruebas para encontrar todos los errores que aún puedan existir en el software desarrollado. En el siguiente capítulo se define el estándar de codificación usado en la implementación y los resultados de varias pruebas que se realizaron para asegurar el correcto funcionamiento del producto.

3.1 Patrones de diseño del sistema

Para comenzar la implementación de un software no basta con definir la arquitectura, es necesario además establecer las directrices que permiten lograr un sistema bien estructurado, para así construir una solución eficaz. Estas directrices son llamadas Patrones de Diseño de Software. Para satisfacer las necesidades de la sociedad cada día se hace más necesario desarrollar un software de gran alcance y complejidad, en lo que son de gran utilidad los patrones de diseño empleados como mecanismos de reutilización.

Para la realización del sistema se emplearán cuatro patrones de diseño, estas son Experto, Bajo acoplamiento, Alta Cohesión e Inyección de Dependencias. A continuación, se muestra la selección de los mismos:

- Experto: Sigue el principio de asignar las responsabilidades a la clase que mayor información contenga. Su empleo trae como beneficios mantener el encapsulamiento de la información (los objetos utilizan su propia información para llevar a cabo las tareas), se distribuye el comportamiento entre las clases que contienen la información requerida (estimula las definiciones de clases cohesivas que son más fáciles de entender y mantener), bajo acoplamiento y alta cohesión[CITATION Cra03 \l 3082].
- Bajo Acoplamiento: Sigue el principio de asignar una responsabilidad de manera que el acoplamiento permanezca bajo. El acoplamiento es una medida de la fuerza con que un elemento depende de otro, por tanto, un elemento con bajo acoplamiento depende de la menor cantidad de elementos posibles, estos elementos pueden ser clases, subsistemas y sistemas, por ejemplo. Emplear el bajo acoplamiento posibilita que el cambio de un componente afecte en menor medida a los restantes y favorece la reutilización de los elementos[CITATION Cra03 \l 3082].
- Alta Cohesión: Sigue el principio de asignar una responsabilidad de manera que la cohesión permanezca alta. La cohesión es una medida de la fuerza con la que se relacionan los elementos y del grado de focalización de las responsabilidades de estos. Un elemento con

responsabilidades altamente relacionadas, y que no realiza diversas funciones, tiene alta cohesión. Como regla empírica, una clase con alta cohesión tiene un número relativamente pequeño de métodos, con funcionalidad altamente relacionada, y no realiza demasiadas funciones. Una mala cohesión causa, normalmente, un mal acoplamiento, y viceversa. El empleo de alta cohesión incrementa la claridad y facilita la comprensión del diseño, simplifica el mantenimiento y las mejoras, facilita el soporte para emplear bajo acoplamiento e incrementa la reutilización[CITATION Cra03 \l 3082].

- Inyección de Dependencia: Basado en las teorías de Inversión de Control se abstraen y se eliminan todas las dependencias que los módulos poseen entre sí y se integran desde una de las capas (dígase capa en el sentido coloquial) superiores de la arquitectura[CITATION Cra03 \l 3082].

```
class StructureUserObjectPermissionViewSet(
    usecases.CreateStructureUserObjPermission, BaseAkViewSet
):
    """
    Permission CRUD for users, only for admin users
    """

    queryset = managers.StructureUserObjectPermissionManager().order_by("-create_date")
    serializer_class = StructureUserObjectPermissionSerializer
    filter_backends = (
        filters.SearchFilter,
        filters.OrderingFilter,
        DatatablesFilterBackend,
    )
    search_fields = ("content_object__name", "user__username", "permission__name")
    ordering_fields = [
        "content_object__name",
        "user__username",
        "permission__name",
        "content_object__type__name",
    ]
```

Figure 10 Ejemplo de inyección de dependencia. Fuente: Elaboración propia.

3.2 Estándares de codificación

Los estándares de codificación usados en la base de código se dividen en 4 atendiendo al lugar en el código donde se aplicaron.

- Nombre de clases: Los nombres de clases utilizan Pascal Case: se refiere al estilo de encadenar palabras con letras capital en una sola palabra.

- Nombre de las funciones y variables: Los nombres de las funciones y variables utilizan Snake-Case: se refiere al estilo de sustituir los espacios por un guion bajo "_".
- Nombre de las constantes: Los nombres de las constantes utilizan Screaming Snake Case: se refiere a poner una cadena de palabras en mayúscula y sustituir los espacios por guiones bajos "_".
- Para el resto de código se usa las directrices de *Pep8* que no afecten la legibilidad y elegancia del código escrito. Pep8 define las claves de cómo debería escribirse el código *Pythónico*, define conceptos como identación, longitudes de líneas, guías de importación de paquetes, espaciado entre operadores, como escribir cadenas de texto, convenciones de nombres y anotaciones de variables. En algunos casos el uso excesivo de aplicar pep8 conduce a código poco elegante o ilegible, queda a criterio del programador donde aplicar su uso.

```
# region Clean architecture: migrations and fixtures
def get_model_permissions(model):
    """
        List of permsissions for a model
    :param model:
    :return: Permission QuerySet
    Ex: get_model_permissions(Structure)
    """
    content_type = ContentType.objects.get_for_model(model)
    return Permission.objects.filter(content_type_id=content_type.id)
```

Figure 11 Ejemplo de los estándares de codificación. Fuente: Elaboración propia.

Figure 12 Ejemplos de los estándares de codificación. Fuente: Elaboración propia.

3.3 Pruebas de software

Las pruebas de software son procesos que permiten verificar y revelar la calidad de un producto. Son utilizadas para identificar posibles fallos de implementación, calidad o usabilidad de un programa [CITATION Rog10 \l 3082]. El objetivo de esta fase es detectar y solucionar los errores que presenta el componente desarrollado, y perfeccionar la solución implementada.

3.3.1 Métodos de Prueba

Los métodos de pruebas definen estrategias para descubrir fallos en el sistema. Como métodos de prueba, Pressman en su 7ma edición [CITATION Rog10 \l 3082], propone pruebas de caja blanca y pruebas de caja negra, pero en la investigación se llevó a cabo la siguiente:

Pruebas de caja blanca

Mediante los métodos de prueba de caja blanca, el ingeniero de software puede obtener casos de prueba que: garanticen que se ejercite por lo menos una vez los caminos independientes de cada módulo; ejercitan todas las decisiones lógicas en sus vertientes verdadera y falsa; ejecuten todos los bucles en sus límites y con sus límites operacionales, y que se ejerciten las estructuras internas de datos para asegurar su validez [CITATION Dan19 \l 3082]. Estas pruebas se realizan al código fuente para asegurar que la operación interna se ajuste a las especificaciones.

3.3.2 Estrategia de prueba

Una estrategia de prueba del software integra las técnicas de diseño de casos de prueba en una serie de pasos bien planificados que dan como resultado una correcta construcción del software. La estrategia proporciona un mapa que describe los pasos que hay que llevar a cabo como parte de la prueba, cuándo se deben planificar y realizar estos pasos, y cuánto esfuerzo, tiempo y recursos se van a requerir. Por tanto, cualquier estrategia de prueba debe incorporar la planificación de la prueba, el diseño de casos de prueba, la ejecución de las pruebas y la agrupación y evaluación de los datos resultantes[CITATION Edg11 \l 3082]

Roger Pressman en la 7ma edición del libro “Ingeniería de software. Un enfoque práctico” propone cuatro niveles de prueba [CITATION Rog10 \l 3082]:

- Pruebas unitarias: son pruebas de caja blanca que se utilizan con el objetivo de detectar errores de implementación en el componente desarrollado. Además, se identifican errores de entrada o salida de datos.
- Pruebas de integración: verifican que cada componente desarrollado no presente errores cuando se integre con los demás.
- Pruebas de validación: son pruebas de caja negra que se enfocan en la satisfacción de las necesidades del cliente, verificando las acciones que el usuario realiza el sistema y la correcta entrada y salida de datos.

- Pruebas del sistema: son pruebas que confirman el correcto funcionamiento de las funciones desarrolladas.

Para la validación de la propuesta de solución de la presente investigación, se definió una estrategia de prueba que incluye pruebas de unidad, en correspondencia a uno de los cuatro niveles antes mencionados. Se incorporó solo esta prueba unitaria puesto que se hizo una reingeniería al código y ya después se da paso a una sola verificación. Además, esta estrategia se diseñó teniendo en cuenta las disciplinas de pruebas internas de la metodología AUP-UCI, seleccionada para guiar el desarrollo del producto. Para las pruebas internas se empleó el método de caja blanca a través de la técnica de clases de pruebas de Django donde te automatiza el proceso y te da un reporte de pruebas.

3.3.3 Pruebas unitarias

Las pruebas unitarias son realizadas a pequeñas porciones de código, por separados, para verificar su correcta funcionalidad, las mismas se pueden ir efectuando desde el comienzo de la implementación, no necesariamente se tiene que esperar a finalizar el software. A continuación, un reporte de pruebas unitarias.

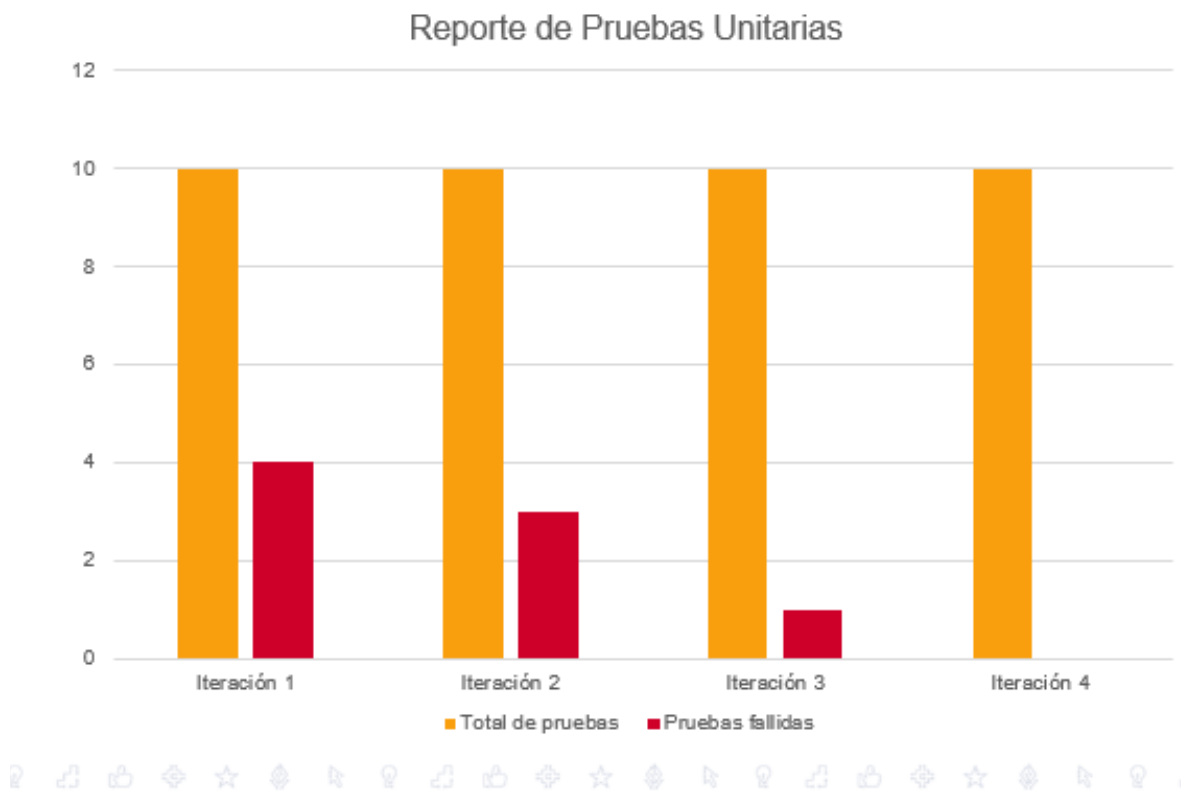


Figure 13 Reporte de Pruebas Unitarias. Fuente: Elaboración propia.

Se tuvieron un total de diez pruebas en cuatro iteraciones en cuatro meses. En la primera iteración se obtuvieron cuatro no conformidades de el total de pruebas, en la segunda iteración tres pruebas fallidas del total de pruebas, en la tercera iteración una no conformidad y en la cuarta iteración ya el código estaba robusto y funcional.

```
def test_create_with_admin_user(self):
    """
    check that a admin user create user
    """
    count = User.objects.all().count()
    response = self.client_admin.post(self.url, self.data, format="json")
    self.assertEqual(response.status_code, status.HTTP_201_CREATED)
    self.assertEquals(count + 1, User.objects.all().count())
```

Figure 14 Ejemplo de prueba: crear usuario desde el rol de Administrador. Fuente: Elaboración propia.

3.4 Resultados de la Investigación

Figure 15 Arquitectura anterior en el módulo Seguridad. Fuente: Elaboración propia.

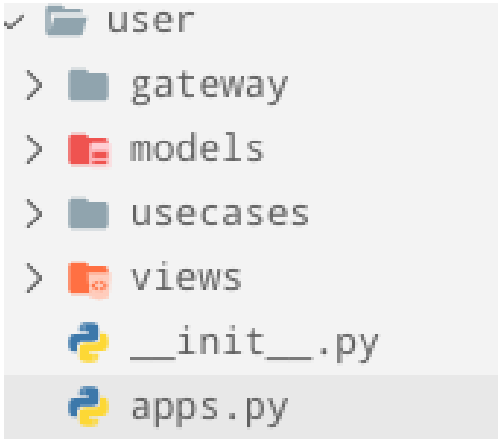


Figure 16 Arquitectura limpia del módulo Seguridad. Fuente: Elaboración propia.

En las anteriores imágenes se evidencia claramente el cambio que pudo dar la arquitectura del módulo Seguridad. La primera imagen tiene muchas más carpetas y archivos que da como resultado

deficiencias en el sistema de gestión. Sin embargo, la imagen de la Arquitectura Limpia muestra las cuatro capas, por lo que se evidencia un gran avance y mejora de los estándares de mantenibilidad, fácil de entender y manejar.

3.4 Conclusiones parciales

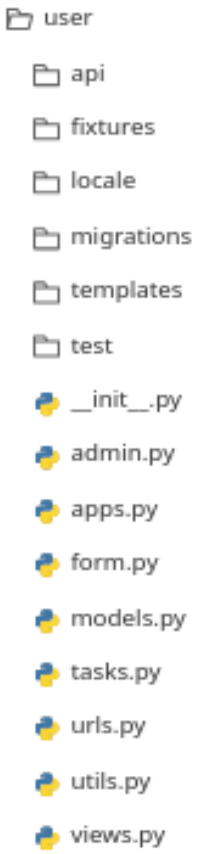
Como parte del desarrollo del presente capítulo se determinan las siguientes conclusiones:

- ✓ Se describieron elementos referentes a la implementación y validación de la propuesta de solución.
- ✓ Se evidenciaron los estándares de codificación empleados en el desarrollo en la base del código.
- ✓ Se evidenció la estrategia de prueba seguida para la validación del software, documentando los resultados obtenidos a partir del empleo de la técnica: Clases de pruebas de Django.

Conclusiones

Al finalizar la presente investigación y una vez implementada la propuesta de solución, se concluye que:

- La definición de los elementos teóricos y metodológicos asociados a la Arquitectura Limpia, permitió profundizar en los aspectos esenciales para el desarrollo de la presente investigación.
- El estudio de las aplicaciones existentes a nivel nacional que permiten la visualización de la información mediante diferentes aspectos, demostró que ninguna constituye buena mantenibilidad, reusabilidad y facilidad de prueba en el contexto de la arquitectura de software, reflejado en la literatura especializada.
- El diagrama de clases y diagrama de paquetes propició el éxito en el desarrollo de los componentes de la propuesta de solución al problema abordado en esta investigación.
- El uso de técnicas de desarrollo dirigido por pruebas y comportamiento permitió la obtención de una arquitectura robusta, resistente a errores y con capacidad de recuperación.
- Las pruebas de software permitieron comprobar la calidad del resultado obtenido y la validez de su funcionamiento en el contexto de la gestión y análisis de la arquitectura.



Recomendaciones

Luego de haber cumplido los objetivos de la presente investigación y una vez implementada la propuesta de solución, se recomienda que:

- La aplicación permita crear un módulo para poder aislar la lógica del negocio del marco de trabajo (Django) porque sí se logró hacer Arquitectura Limpia , el código está más reusable pero los objetivos que persigue dicha arquitectura no son esos nada más, se debe extender al desacoplamiento del negocio de cualquier marco de trabajo, con el objetivo de que si se va a cambiar el marco de trabajo se pueda separar la lógica de negocio para otra parte.

Referencias bibliográficas

Bibliography

1. *Evolución de las Metodologías y Modelos utilizados en el Desarrrollo de Software.* **Johanna Patricia Zumba Gamboa, Cecibel Alexandra León Arreaga.** 10, Guayaquil : INNOVA Research Journal, 2018, Vol. III. ISSN.

2. **Gutiérrez, Cristhian Fabián Cabezas.** *Estudio Cualitativo del impacto de Aplicación de Buenas Prácticas para la Administración de empresas con mención en Gerencia de la Calidad y Productividad.* Quito : s.n., 2016.

3. **Andrade, Jorge Vladimir Chávez.** *Estandarización de los Procesos de Desarrollo de Software utilizando Buenas Prácticas de Programación y SCRUM como marco de trabajo ágil en Departamentos de TI.* Ambato : s.n., 2019.

4. **Pressman, Roger S.** *Ingeniería del Software, un enfoque práctico.* D. F : The McGraw-Hill, 2010. ISBN.

5. **Difabio, Federico Ezequiel.** *Diseño e implementación de una Arquitectura de Software guiada por dominio.* s.l. : UNRN Sede Atlántica, 2021.

6. *Utilización de Arquitecturas Limpias para Trabajo con Buenas Prácticas.* **Julio César Martínez Z, César henao et al.** 2, Medellin : Innovación Digital y Desarrollo Sostenible, 2020, Vol. I. ISSN.

7. **Orestes de la Fé Concepción, Javier Vélez Pérez.** *Propuesta de una Arquitectura en PHP para el sistema ERP cubano.* La Habana : s.n., 2008.

8. **M, Solanes.** *Implementant Clean Architecture.* USA : s.n., 2018.

9. **F., Rodríguez.** *A Clean Approach to Flutter Development through the Flutter Clean Architecture Package.* USA : s.n., 2019.

10. **M., Knill.** *Refactoring to clean architecture.* 2019.

11. **Ramírez, Miguel Medina.** *Definición de Arquitectura de software.* San Antonio : s.n., 2021.

12. **Serrano, Ernesto Mastrapa.** *Sistema de apoyo de la Gestión de la Información Académica.* La Habana : s.n., 2014.

13. **Marín, Yircy Diem Collazo.** *Sistema de Gestión Académica: Módulo de Gestión de profesores.* La Habana : s.n., 2005.

14. **Iván F. Palomo, Carlos G. Veloso et al.** *Sistema de Gestión de la Investigación en la Universidad de Talca, Chile.* Chile : Información Tecnológica, 2007.

15. **Marcelo, Yaguachi Barahona Paúl.** *Análisis, Diseño y desarrollo de un Sistema de Gestión Académica vía web para institutos de investigación y posgrado implementado en la Facultad de Ingeniería.* Quito : s.n., 2015.

16. *El Campus Virtual de la Universidad de barcelona. Modelos de enseñanza y aprendizaje emergentes.* **Joan Anton Sánchez i Valero, Max Muntadas Pekkola, et al.** 2, Barcelona : RELATEC, 2008, Vol. VII. ISSN.

17. *Mantenibilidad del Software. Consideraciones para su especificación y validación.* **Jenny Adones Farfán, Vianca Vega Zepeda.** 4, Antofagasta : Revista chilena de ingeniería, 2020, Vol. XXVIII.

18. **Francisco Ruiz, Macario Polo.** *Mantenimiento del Software.* Real : ULM-ESI, 2001.

19. **Gómez, Elena Albertos.** *Arquitecturas Software para Microservicios: Una Revisión Sistemática de la Literatura.* Madrid : s.n., 2018.

20. *Arquitectura de software para la construcción de un sistema de cuadro de mando integral como herramienta de inteligencia de negocios.* **Cabrera, Gustavo Adolfo Hernández.** 2, Bogotá : Tecnología, Investigación y Academia, 2017, Vol. V. ISSN.

21. **Martin, Robert C.** *Arquitectura Limpia.* s.l. : AANAYA MULTIMEDIA, 2018. ISBN.

22. **Giordani, Leonardo.** *Clean Architectures in Python.* s.l. : Lean Publishing, 2020.

23. **Orestes de la Fé Concepción, Javier Vélez Pérez.** *Propuesta de una arquitectura en PHP para el sistema ERP cubano.* La Habana : s.n., 2008.

24. **Macías, Julio.** *La plataforma EVEA: Experiencias en su uso.* La Habana : Full-text, 2021.

25. *Experiencias en el uso de la Tecnología Educativa en el período Covid-19 en la Universidad de Oriente.* **Gustavo Cervantes Montero, Oscar García Fernández, Alina Díaz Frog.** 2, universidad de oriente : Maestro y Sociedad, 2021, Vol. XVIII. ISSN.

26. **Benítez, Drymon Alfonso.** *Herramienta para generar productos de trabajos de la metodología variación AUP-UCI.* la Habana : s.n., 2017.

27. **Sánchez, Tamara Rodríguez.** *Metodología de desarrollo para la Actividad productiva de la UCI.* La Habana : s.n., 2014.

28. *El lenguaje de programación Python.* **Ivet Challenger Pérez, Yanet Díaz Ricardo y Roberto Antonio Becerra García.** 2, Holguín : ciget.holguin.inf.cu, 2014, Vol. XX. ISSN.

29. **Sparks, Geoffrey.** *Una Introducción al UML.* Chile : sparxsystems.com.ar, 2000.

30. *Configuración de usuarios en git para grupos de desarrollo en entornos universitarios.* **josé Arístides valencia Ruiz, José Miguel Loor Intriago, Emilio Antonio Cedeño Palma y Anaisa Hernández González.** 9, s.l. : San Gregorio, 2015, Vol. I. ISSN.

31. **G., Ángel R. Ávila.** *Impacto del uso del sistema de control de versiones GITLAB como heramienta de monitoreo y evaluación académica de trabajos colaborativos en la facultad como Informática, Electrónica y Comunicación.* Panamá : s.n., 2019.

32. **López, Roisbel Portales.** *Sistema para la selección de modelos de negocio en la comercialización de los productos del Centro de Informática Industrial.* La Habana : s.n., 2016.

33. **Boeras, Mairelys.** *Especificación de requisitos de software.* La Habana : s.n., 2021.

34. —. *Modelo Conceptual.* La Habana : s.n., 2021.

35. **Larman, Craig.** *UML y Patrones: Una introducción al análisis y diseñoorientado a objetos y al proceso unificado.* s.l. : Prentice-Hall, 2003.

36. *Generador de valores interesantes para casos de pruebas unitarias.* **Dania Mailen Rojas Robert, Zeyla Pérez Morales, martha Dunia Delgado Dapena.** 2, La Habana : scielo, 2019, Vol. XL. ISSN.

37. *Desafíos y estrategias prácticas de los estudios empíricos sobre las técnicas de prueba del software.* **Edgar Serna, Fernando Arango.** 1, Cali : Ingeniería y Competitividad, 2011, Vol. XIII. ISSN.

38. **C., Fredy Méndez.** *Sistema de Gestión Académica para la Unidad Educativa ´´Manuel Guerrero´´.* Cuenca : s.n., 2012.

39. **Kaplan-Moss, Adrian Holovaty y Jacob.** *The Definitive Guide to Django.* New York : Apress, 2009. ISBN.

40. *A study of the effectiveness of usage examples in REST API documentation.* **Sohan S. M, frank Maurer, Craig Anslow y Martin P. Robillard. et al.** s.l. : IEEE Symposium on Visual languages and human-Centric Computing (VL/HCC), 2017.