UNIVERSIDAD NACIONAL DE MOQUEGUA

CARRERA PROFESIONAL DE INGENIERÍA DE SISTEMAS E INFORMÁTICA



"EFECTO DE LA PROGRAMACIÓN ORIENTADA A ASPECTOS Y OBJETOS EN EL DISEÑO DE UN SOFTWARE–CASO: BIBLIOTECA DE LA UNIVERSIDAD NACIONAL DE MOQUEGUA SEDE – ILO"

TESIS

Autor: Bach. Renán Darío Gonzales Apaza
Asesor: Ing. Carlos Alberto Silva Delgado

PARA OPTAR EL TÍTULO PROFESIONAL DE
INGENIERO DE SISTEMAS E INFORMÁTICA

UNAM - Universidad Nacional de Moquegua
ILO – PERÚ - 2014

UNIVERSIDAD NACIONAL DE MOQUEGUA
CARRERA PROFESIONAL DE INGENIERÍA DE SISTEMAS E INFORMÁTICA

ACTA DE SUSTENTACIÓN

El Jurado suscribiente ha calificado el trabajo de Tesis:

TITULADO Efecto de la Programación Orientada a Aspectos y Objetos en el diseño de un Software - Caso: Biblioteca de la Universidad Nacional de Moquegua - Sede ILO

PRESIDENTE Ing. Hugo Euler Tito Chura

MIEMBROS 1er Miembro: Ing. Anibal Fernando Flores García

2do Miembro: Ing. Alex Peter Zuñiga Incalla

ASESOR Ing. Carlos Alberto Silva Delgado

PRESENTADO POR EL BACHILLER Renán Dario Gonzales Apaza

DE LA PROMOCIÓN: 2012

CUYOS RESULTADOS HAN SIDO LOS SIGUIENTES:

CALIFICATIVO DEL TRABAJO Muy Bueno (Diecisiete)

CALIFICATIVO DE LA SUSTENTACIÓN Muy Bueno (Diecisiete)

CALIFICATIVO FINAL Muy Bueno (Diecisiete) - APROBADO

POR LO EXPUESTO, EL BACHILLER Renán Dario Gonzales Apaza

HA SIDO DECLARADO EXPEDITO PARA QUE SE LE CONFIERA EL TÍTULO PROFESIONAL DE:

INGENIERO DE SISTEMAS E INFORMÁTICA

EN FE DE ELLO QUEDA ASENTADA LA PRESENTE ACTA

ILO, A LOS Once DÍAS DE Noviembre DE 2014

Presidente Euler Tito Chura

Miembro
Anibal Flores García

Miembro
M. Alex Zuñiga I.

Dedicatoria:

A mi madre, porque creyó en mí y porque me sacó adelante, dándome ejemplo digno de superación y entrega, porque en gran parte gracias a ella, hoy puedo ver alcanzado parte de mi meta, ya que siempre estuvo impulsándome en los momentos más difíciles de mi carrera, y porque el orgullo que siente por mí, fue lo que me hizo ir hasta el final. Va por ustedes, por lo que valen, porque admiro su fortaleza y por lo que han hecho de mí.

A mi padre por haber fomentado en mí el deseo de superación y el anhelo de triunfo en la vida.

Agradecimientos

Le agradezco en primer lugar a Dios, que me ha protegido y guiado durante toda mi vida. He cumplido parte de todas las metas que me he trazado y esta la considero como una de las más importantes en mi vida.

A mis padres Antonia y Renán que han estado siempre a mi lado aconsejándome y apoyándome en las decisiones que he tomado, por el amor, la confianza, la paciencia durante toda mi formación universitaria y por haberme guiado sabiamente hasta alcanzar este objetivo.

Agradezco a la universidad, mi alma matter, por haberme cobijado y brindado la formación que ahora me permitirá ayudar a construir una mejor sociedad.

Agradezco de forma muy especial a mi orientador Ing. Carlos Alberto Silva Delgado por haberme guiado en todo el proceso de desarrollo del presente trabajo por brindarme su tiempo y dedicación.

Deseo agradecer de forma especial a mis docentes: Ing. Anibal Fernando Flores Garcia, Ing. Alex Peter Zuñiga Incalla, Ing, Vaneza Flores Gutiérrez, Ing. Hugo Euler Tito Chura, Ing. Luis Amaro Villanueva Tapia. Porque fueron ejemplos que deseo seguir en mi vida profesional.

Al Ing. Alex Wilson Yucra Mamani de la Oficina de Informática y Sistemas de la UNAM, y al CP. Richard Baltazar Dorado Baca de la unidad de biblioteca de la UNAM Sede Ilo, por haberme brindado todas las facilidades para la elaboración del presente trabajo de investigación.

Y a todas las personas que contribuyeron directa o indirectamente aportando en este trabajo.

Índice general

Índice de imágenes VII
Índice de tablas X
Resumen 1
Introducción 2
1. Planteamiento de la investigación 3
1.1. Descripción del problema 3
1.2. Formulación del problema 4
1.2.1. Problema general 4
1.2.2. Problema específico 4
1.3. Justificación 5
1.4. Limitaciones 5
1.5. Objetivos 6
1.5.1. Objetivo general 6
1.5.2. Objetivos específicos 6
1.6. Hipótesis 6
1.6.1. Hipótesis general 6
1.6.2. Hipótesis específicas 6
1.7. Variables 6
1.7.1. Identificación de variables 6
1.7.2. Definición de variables 7
1.7.3. Operacionalización de variables 7
1.8. Tipo y diseño de la investigación 8
1.8.1. Tipo de investigación 8
1.8.2. Diseño de la investigación 8
1.8.3. Población 9
1.8.4. Técnicas e instrumentos para la recolección de datos 9
2. Marco teórico 10
2.1. Antecedentes de la investigación 10
2.2. Base teórica 11
2.2.1. Programación orientada a objetos 11

2.2.2. Programación orientada a aspectos 29
2.2.3. Diseño del software 48
2.3. Marco conceptual 65
2.3.1. Aspecto 65
2.3.2. Competencia 65
2.3.3. Complejidad 65
2.3.4. Diseño 65
2.3.5. Diseño de software 65
2.3.6. Efectividad 65
2.3.7. Efecto 65
2.3.8. Objeto 66
2.3.9. Requerimiento funcional 66
2.3.10. Requerimiento no funcional 66
2.3.11. Separación de competencias 66
3. Desarrollo 67
3.1. Descripción del caso de estudio: 67
3.1.1. Requerimientos del software (lenguaje natural) 67
3.2. Caso 1: Diseño de un software orientado a objetos 69
3.2.1. Requerimientos 69
3.2.2. Análisis 83
3.2.3. Diseño 95
3.2.4. Implementación 104
3.3. Caso 2: Diseño de un software orientado a aspectos 108
3.3.1. Requerimientos 108
3.3.2. Análisis 119
3.3.3. Diseño 126
3.3.4. Implementación 134
4. Pruebas y resultados 137
4.1. Resultados obtenidos de la prueba del tamaño (LDC) 137
4.2. Resultados obtenidos de la prueba complejidad ciclomática 140
4.3. Resultados obtenidos de la eficiencia 142
4.4. Resultados obtenidos de la mantenibilidad 146

5. Conclusiones y recomendaciones 149
5.1. Conclusión general 149
5.2. Conclusiones específicas 149
5.3. Recomendaciones 150
Bibliografía 151
Anexos 155
Anexo I : Tabla de autorizaciones 155
Anexo II: Matriz de consistencia 160
Anexo III: Diagrama de la base de datos del sistema 161
Anexo IV: Descripción textual del sistema 162

Índice de imágenes

Ilustración 1: Estado, comportamiento e identidad....18
Ilustración 2: Clasificación de lenguajes orientados a objetos según Wegner (1987)....21
Ilustración 3: Ciclo de Vida Metodología OMT....24
Ilustración 4: Pasos Micro proceso....26
Ilustración 5: Rational Unified Process....27
Ilustración 6: Incumbencias o intereses en un sistema....30
Ilustración 7: Implementación de la lógica del negocio con incumbencias transversales en la forma tradicional....31
Ilustración 8: Implementación de incumbencias transversales sin aspectos y con aspectos....32
Ilustración 9: Función del tejedor de aspectos....34
Ilustración 10: Proceso de tejido....34
Ilustración 11: Proceso de diseño orientado a aspectos....37
Ilustración 12: Connector design in the Fraunhofer Architecture Design process....39
Ilustración 13: Las cinco etapas del proceso de desarrollo de XP muestran que las iteraciones frecuentes son esenciales para desarrollar con éxito un sistema....40
Ilustración 14: Etapas del desarrollo de software orientado a aspectos....41
Ilustración 15: Protocolo entre un objeto y su coordinador en COOL....44
Ilustración 16: Protocolo de un portal RIDL....45
Ilustración 17: Spring AOP....47
Ilustración 18: Arquitectura de compilación con AspectC++....48
Ilustración 19: Transformación del modelo de análisis al modelo de diseño....49
Ilustración 20: Arquitectura centrada en datos....55
Ilustración 21: Arquitectura de flujo de datos....56
Ilustración 22: Arquitectura de llamada y retorno....57
Ilustración 23: Arquitectura orientada a objetos....57
Ilustración 24: Arquitectura estratificada....58
Ilustración 25: Arquitectura cliente – servidor....58
Ilustración 26: Arquitectura basada en componentes....59
Ilustración 27: Arquitectura en capas (N-Layer)....60
Ilustración 28: Arquitectura de modelo de vista del controlador....61
Ilustración 29: Arquitectura en capas (N-Tier)....61
Ilustración 30: Caso de uso préstamo de ejemplar....70
Ilustración 31: Reservar ejemplar....71
Ilustración 32: Devolución de ejemplares....73
Ilustración 33: Importar ejemplares....74
Ilustración 34: Exportar copia de seguridad....75
Ilustración 35: Importar copia de seguridad....76
Ilustración 36: Administración de usuarios....77
Ilustración 37: Gestión de utilidades....79
Ilustración 38: Administración de ejemplares....80
Ilustración 39: Gestión de administradores....82
Ilustración 40: Modelo de análisis préstamo de ejemplar....84

Ilustración 41: Modelo de análisis de reserva de ejemplar....84
Ilustración 42: Modelo de análisis de devolución de ejemplar....85
Ilustración 43: Modelos de análisis exportar copia de seguridad....85
Ilustración 44: Modelo de análisis de importación de copia de seguridad....85
Ilustración 45: Modelo de análisis de administración de usuarios....86
Ilustración 46: Modelo de análisis de gestión de utilidades....86
Ilustración 47: Modelo de análisis de administración de ejemplares....87
Ilustración 48: Modelo de análisis gestión de administradores....87
Ilustración 49: Diagrama de secuencia préstamo de ejemplar....88
Ilustración 50: Diagrama de secuencia de reserva de ejemplar....89
Ilustración 51: Diagrama de secuencia de administración de usuario: sanción y cuentas....89
Ilustración 52: Diagrama de secuencia de Administración de usuarios edición y cambio de escuela....90
Ilustración 53: Diagrama de secuencia importar copia de seguridad....90
Ilustración 54: Diagrama de secuencia de una devolución....91
Ilustración 55: Diagrama de secuencia de importación de ejemplar....91
Ilustración 56: Diagrama de secuencia de exportar copia de seguridad....92
Ilustración 57: Diagrama de secuencia de gestión de utilidades....93
Ilustración 58: Diagrama de secuencia de administración de ejemplares....94
Ilustración 59: Diagrama de secuencia de administración de ejemplares....95
Ilustración 60: Identificación de subsistemas....97
Ilustración 61: Subsistemas de administración de usuarios....98
Ilustración 62: Subsistema de préstamos....98
Ilustración 63: Subsistemas de catálogo....99
Ilustración 64: Subsistema de configuración....99
Ilustración 65: Subsistema de administración....100
Ilustración 66: Subsistema de utilidades....100
Ilustración 67: Diagrama de componentes Modelo Vista Controlador....101
Ilustración 68: Diagrama de despliegue del sistema....102
Ilustración 69: Diagrama de clases....103
Ilustración 70: Caso de uso de infraestructura y requerimientos de administración de usuarios....110
Ilustración 71: Caso de uso de infraestructura y requerimientos de administración de ejemplares....111
Ilustración 72: Caso de uso de infraestructura y requerimientos de administración ubicaciones y estados físicos....112
Ilustración 73: Caso de uso de infraestructura y requerimientos de importación de ejemplares....113
Ilustración 74: Caso de uso de infraestructura y requerimientos de gestión de administradores....114
Ilustración 75: Caso de uso de infraestructura y requerimientos de reserva de ejemplar....115
Ilustración 76: Caso de uso de infraestructura y requerimientos de préstamos....116
Ilustración 77: Caso de uso de infraestructura y requerimientos de devoluciones....116
Ilustración 78: Caso de uso de infraestructura y requerimientos de exportar Backup del sistema....117
Ilustración 79: Caso de uso de infraestructura y requerimientos de importación de Backup del sistema....118
Ilustración 80: Caso de uso de infraestructura y requerimientos gestión de utilidades....118
Ilustración 81: Administración de usuarios....119
Ilustración 82: Administración de ejemplares....120
Ilustración 83: Modelo de análisis de gestión de administradores....120

Ilustración 84: Reservas.....120
Ilustración 85: Préstamos.....121
Ilustración 86: Devoluciones.....121
Ilustración 87: Configuraciones del sistema.....122
Ilustración 88: Gestión de utilitarios.....122
Ilustración 89: Diagrama de secuencia de préstamo de ejemplar.....123
Ilustración 90: Ubicaciones de ejemplar.....124
Ilustración 91: Estados de ejemplar.....124
Ilustración 92: Cuentas de usuario.....125
Ilustración 93: Sanciones de usuario.....125
Ilustración 94: Reserva de ejemplares.....126
Ilustración 95: Gestión de administradores.....127
Ilustración 96: Administración de usuarios.....127
Ilustración 97: Catálogo.....128
Ilustración 98: Préstamos y devoluciones.....129
Ilustración 99: Sistema.....130
Ilustración 100: Utilidades.....131
Ilustración 101: Ilustración de la ejecución de los puntos de corte.....132
Ilustración 102: Diagrama de componentes MVC con la inclusión de aspectos.....133
Ilustración 103: Diagrama de despliegue.....133
Ilustración 104: Tejido de aspectos con las clases bases.....134
Ilustración 105: Diagrama comparativo en líneas de código.....139
Ilustración 106: Gráfico comparativo entre la programación orientada a aspectos y objetos.....142
Ilustración 107: Prueba de tiempo de ejecución en la administración de ejemplares en el lado del administrador.....144
Ilustración 108: Tiempos de ejecución en administración de ejemplares en el lado del administrador146
Ilustración 109: Gráfico comparativo de mantenibilidad entre la programación orientada a aspectos y objetos.....148

Índice de tablas

Tabla 1: Préstamo de ejemplar 71
Tabla 2: Reserva de ejemplar 72
Tabla 3: Devolución de ejemplares 74
Tabla 4: Importar ejemplares 75
Tabla 5: Exportar copia de seguridad 76
Tabla 6: Importar copia de seguridad 77
Tabla 7: Administración de usuarios 78
Tabla 8: Gestión de utilidades 80
Tabla 9: Administración de ejemplares 81
Tabla 10: Gestión de administradores 82
Tabla 11: Cuadro comparativo de cantidad de líneas de código. 138
Tabla 12: Tamaño de líneas de código de aspectos. 138
Tabla 13: Cuadro resumen comparativo entre la programación orientada a aspectos y objetos. 138
Tabla 14: Cuadro comparativo de la complejidad ciclomática entre la programación orientada a aspectos y objetos 141
Tabla 15: Tabla de escalas para la complejidad ciclomática. 141
Tabla 16: Prueba de eficiencia de administración de ejemplares en el panel del administrador. 143
Tabla 17: Prueba de eficiencia de búsqueda de ejemplares en el lado del usuario 145
Tabla 18: Tabla comparativa del índice de mantenibilidad. 147

Resumen

El presente trabajo tiene por objetivo establecer una comparación del efecto de la programación orientada a aspectos y la programación orientada a objetos en el diseño de un software tomando como caso la realización de un sistema de biblioteca para la Biblioteca de la Universidad Nacional de Moquegua Sede – Ilo. En su realización se incluyeron las etapas de desarrollo del software para así de esta forma determinar el efecto de cada tipo de programación en el producto. Se realizaron dos grupos de estudio uno orientado a aspectos y otro grupo orientado a objetos, en los que se desarrollaron actividades de especificación de requerimientos, análisis para de esta manera llegar al diseño y posteriormente pasar a la codificación, y en última instancia comparar ambos como productos finales. Los resultados muestran que el uso de la programación orientada a aspectos mejora el diseño de un software en comparación con la orientación a objetos.

Palabras clave: Programación orientada a aspectos, Programación orientada a objetos, diseño de un software.

Introducción

La evolución del diseño de software desde comienzos del desarrollo de software ha ido cambiando según la evolución de la introducción de nuevos retos de organizar la estructura de un sistema, los trabajos sobre diseño se centran en criterios de modularidad, de separación de tareas, de acoplamiento que permita un buen grado de mantenibilidad y evolución para llevar a una buena arquitectura del software de una manera decente.

El motivo que llevó a desarrollar el presente trabajo de investigación fue buscar una mejor forma de acercarse al concepto de diseño de software en la separación de unidades de software, aumento de la realización de tareas específicas en cada subsistema siguiendo el principio de divide y vencerás, brindado de una buena mantenibilidad y baja complejidad, ya que al realizar esto el producto final ofrecerá mejores características en diferentes ámbitos de calidad. Se piensa obtener resultados que generen un punto de partida para futuras investigaciones y reflexión sobre la forma tradicional que se ha venido pensando en la creación de un software, como también ofrecer una alternativa para el desarrollo de software. Se pretende demostrar que haciendo uso de la programación orientada a aspectos se lograra un mejor diseño comparado con la programación orientada a objetos.

Finalmente se presenta la estructura del trabajo de investigación realizado: En el capítulo uno se describe el problema que enfrenta la etapa del diseño del software, viéndolo como un punto de equilibrio de calidad, así como la búsqueda de la modularización de un sistema en diferentes niveles de competencias, haciendo una descripción en la incidencia de usar ambos enfoques en el diseño del software. En el capítulo dos se realiza un estudio teórico y caracterización de los enfoques de programación, así como la descripción del diseño y las características que debe de tener un buen diseño. En el capítulo tres se realiza el desarrollo de dos casos, uno orientado a aspectos y el otro orientado a objetos. En cada uno se siguen los mismos pasos, difiriendo en la forma de cómo se produce un software desde los requerimientos, análisis, pasando por el diseño hasta el producto final. En el capítulo cuatro se realizan pruebas a cada versión del sistema para finalmente realizar un análisis de los datos obtenidos y su respectiva interpretación, identificando los beneficios que puedan traer cada enfoque de desarrollo aplicado. Y por último en el capítulo final se enuncian las conclusiones obtenidas, así como futuros trabajos que podrían partir sobre este, o dar pasos para futuras discusiones.

1. Planteamiento de la investigación

1.1. Descripción del problema

El desarrollo de software en todos sus niveles continuamente enfrenta el incremento de la complejidad y tamaño de los sistemas que se desarrollan, consecuentemente los métodos que se aplican en el diseño del software deben ser escalables en tamaño y complejidad con el fin de entregar productos de calidad en el tiempo requerido. Es decir los métodos de programación que se aplican deberán proporcionar apoyo práctico de manera eficiente y eficaz para sus requerimientos y componentes. (Keuler, 2011).

El diseño del software cumple una función elemental en el desarrollo del software ya que es un punto de quiebre en donde se pone en peso a la calidad (Sommerville, 2011). El diseño además de tener la labor de implementar el análisis en relación a la especificación de requerimientos, sienta las bases para la implementación. Entonces en este punto es donde la implementación hereda la estructura y el modo en que ha de implementarse cada componente de software. Sin embargo una buena especificación de requerimientos y un buen análisis sin una buena separación de competencias al nivel de requerimientos, hacen el efecto que, en el diseño del software exista una pobre identificación o una sobrecarga en la tarea de identificar atributos del sistema y su división en subsistemas por ende la arquitectura a nivel del software se verá afectada y posteriormente en la implementación se dará el efecto de atributos del sistema implementados en otros componentes y a su vez diseminados por el resto del sistema, a este efecto se le denomina la tiranía de la descomposición dominante (Tarr P. Ossher H., y otros, 1999). Como consecuencia se recae en una poca mantenibilidad, y la complejidad de sistema aumenta, y a su vez existiendo una diferencia entre diseño e implementación, lo cual en la práctica no es deseable como se indica en (Bosch, 2000), un indicador de un buen sistema es aquel que guarda relación entre las etapas de su desarrollo.

Por otro lado, según como menciona en (Sommerville, 2011) al aumentar el tamaño del sistema las relaciones entre requerimientos y componentes son complejas, un requerimiento puede implementarse en un componente, y cada uno incluir partes, elementos de otros requerimientos. Es decir de una u otra manera si se extiende un cambio de requerimiento o lógica se afectará a varios componentes, lo que compromete seriamente al diseño del software y la mantenibilidad del sistema.

Uno de los principios del diseño del software es que debe de ser modular, y a su vez dividirse en subsistemas con características funcionales independientes, tenga integrados componentes que exhiban buenas características de diseño, y puedan implementarse de manera evolutiva para conducir a una baja complejidad.

Con el enfoque de subrutinas, procedimientos, y de objetos no se puede aplicar realmente este principio o si se aplica se hace creando componentes extra que realicen tal tarea pero a su vez incrementamos la complejidad, esto porque simplemente no hay soporte para ello.

La programación orientada a aspectos ofrece soporte a este problema, sin embargo en muchos casos, su aplicación se da en la etapa de implementación vale decir la codificación, esto quiere decir que en la práctica, en la etapa de diseño se hace mención a atributos del sistema pero no muestra la manera de cómo interactuarán con los requerimientos funcionales, o bien se realizan tablas de correspondencia entre atributos del sistema y requerimientos funcionales, con esto se piensa que el efecto en el diseño del software aplicando aspectos se hace de manera tardía provocando una no correspondencia entre fases de desarrollo, por lo tanto se debería aplicar desde los inicios del proceso de desarrollo del software. Pero conlleva a retos como su identificación y estructuración a través de los requerimientos, siguiendo el análisis hasta el diseño y llegando a la implementación, una vez que se realice se podrá comprender un efecto más cercano en el diseño del software.

Finalmente ante esta situación, se evidencia que diseño de software forma una de las principales partes del desarrollo de software. Así mismo se ve evidente comparar el efecto en el diseño de un software de estos enfoques en todo el proceso de desarrollo, por ello es que se realiza el presente trabajo de investigación.

1.2. Formulación del problema

1.2.1. Problema general

¿Cuál es el efecto de la programación orientada a aspectos y objetos en el diseño de un software?

1.2.2. Problema específico

- ¿Cuál es el efecto de la programación orientada a aspectos en el diseño de un software?
- ¿Cuál es el efecto de la programación orientada a objetos en el diseño de un software?

1.3. Justificación

La necesidad de mejorar la tarea del diseño de un software, elevar la abstracción, eficiencia y desarrollo, ha llevado al surgimiento de nuevos paradigmas de programación. Y de la manera de cómo abordamos el diseño de un software desde sus inicios hasta la entrega del producto final.

Esta investigación generará reflexión y discusión tanto sobre el conocimiento existente en la programación orientada a aspectos, como dentro de la programación orientada a objetos, ya que de alguna manera u otra, se confrontaran teorías sobre el efecto en nuestro caso sobre el diseño de un software, lo cual necesariamente conlleva hacer epistemología del conocimiento existente.

En este estudio, por otra parte, comparará el efecto de los tipos de programación aplicados en el diseño del software. Por tal razón investigaciones futuras encontrarán en este trabajo la profundización del conocimiento acerca de los enfoques de programación presentados cuya relevancia en el diseño del software, en particular, y de los programadores, analistas, en general, es indiscutible.

1.4. Limitaciones

La primera limitación de este estudio es la falta de un servidor propio en donde se ejecutaran los sistemas, se menciona porque a través del estudio y en la implementación existen *frameworks*, librerías que para el uso de funciones especiales, requieren la instalación de dependencias en el lado del servidor, lo cual limita la implementación.

Y la segunda limitante es que el proceso de desarrollo del software en algunos casos una actividad que incluye varias disciplinas a nivel de etapas, para el presente trabajo no se realizará con múltiples equipos de desarrollo sino se realizará de manera individual.

1.5. Objetivos

1.5.1. Objetivo general

Comparar el efecto de la programación orientada a aspectos y objetos en el diseño de un software

1.5.2. Objetivos específicos

- Determinar el efecto de la programación orientada a aspectos en el diseño de un software.
- Determinar el efecto de la programación orientada a objetos en el diseño de un software.

1.6. Hipótesis

1.6.1. Hipótesis general

En el diseño de un software con la programación orientada a aspectos la efectividad es mejor que la orientada a objetos.

1.6.2. Hipótesis específicas

- En el diseño de un software con la programación orientada a aspectos el efecto es favorable.
- En el diseño de un software en la programación orientada a objetos el efecto es desfavorable.

1.7. Variables

1.7.1. Identificación de variables

a) Variables independientes:
   - Programación orientada a aspectos
   - Programación orientada a objetos

b) Variable dependiente:
   - Diseño de un software

1.7.2. Definición de variables

a) Programación orientada a aspectos
   - Desde el punto de vista de las relaciones que tiene.
     Variable Independiente.
   - Desde el punto de vista factual.
     Cuantitativa.
   - Desde el punto de vista lógico formal.
     Individual.
   - Desde el punto de vista del número de valores.
     Dicotómica.
   - Desde el punto de vista de la función.
     Activa.

b) Programación orientada a objetos
   - Desde el punto de vista de las relaciones que tiene
     Variable Independiente.
   - Desde el punto de vista factual
     Cuantitativa.
   - Desde el punto de vista lógico formal
     Individual.
   - Desde el punto de vista del número de valores
     Dicotómica.
   - Desde el punto de vista de la función
     Activa

c) Diseño de un software
   - Desde el punto de vista de las relaciones que tiene
     Variable Dependiente
   - Desde el punto de vista factual
     Cuantitativa.
   - Desde el punto de vista lógico formal
     Individual.
   - Desde el punto de vista del número de valores
     Politómica.
   - Desde el punto de vista de la función
     Atributiva.

1.7.3. Operacionalización de variables

a) Programación orientada a aspectos
   - Definición: La Programación Orientada a Aspectos es un paradigma de programación cuya intención es permitir una adecuada modularización de las aplicaciones y posibilitar una mejor separación de incumbencias. Se pueden encapsular los diferentes conceptos que componen una

aplicación en entidades bien definidas, eliminando las dependencias entre cada uno de los módulos.

- Indicadores: Tamaño (LDC)

b) Programación orientada a objetos
   - Definición: La programación orientada a objetos es un paradigma de programación que usa los objetos en sus interacciones, para diseñar aplicaciones y programas informáticos. Está basado en varias técnicas, incluyendo herencia, cohesión, abstracción, poliforfismo, acoplamiento y encapsulamiento. (DavidA.Watt, 2010)

   - Indicadores: Tamaño (LDC)

c) Diseño de un software
   - Definición: Un diseño de software se entiende como una descripción de la estructura del software que se va a implementar, los modelos y las estructuras de datos utilizados por el sistema, las interfaces entre componentes del sistema y, en ocasiones, los algoritmos usados (Sommerville, 2011).

   - Indicadores: Complejidad Ciclomática, Eficiencia, Mantenibilidad.

1.8. Tipo y diseño de la investigación

1.8.1. Tipo de investigación

- Tipo: Experimental
- Nivel: Pre-experimental

1.8.2. Diseño de la investigación

El diseño es con grupos experimentales y pos prueba únicamente, el diseño de la investigación se desarrollará de la siguiente manera para el diseño de un sistema: un grupo con la programación orientada a aspectos, mientras que el otro grupo con la programación orientada a objetos.

$$\begin{array}{lll} G_1 & X_1 & O_1 \\ G_2 & X_2 & O_2 \end{array}$$

1.8.3. Población

- Población:
  Bibliotecas de universidades públicas.
- Muestra:
  Biblioteca de la universidad nacional de Moquegua sede – Ilo.

1.8.4. Técnicas e instrumentos para la recolección de datos

- La observación
  - Ficha de registro
- Análisis documental
  - Ficha bibliográfica
- Entrevistas
  - Guion de entrevista

2. Marco teórico

2.1. Antecedentes de la investigación

En el año 2013, Boticki y otros, en su artículo "Exploring the Educational Benefits of Introducing Aspect-Oriented Programming Into a Programming Course", realizó un experimento enseñando en forma separada a dos grupos de estudiantes, uno con la POA[1] y POO[2] con el objeto de evaluar el efecto educacional de cada tipo de programación. Llegando a la conclusión que la programación orientada a aspecto logra mejores resultados académicos, Los estudiantes que utilizan AOP escribieron código más corto y conciso, y sus programas se caracterizaron por una mejor modularidad. (Boticki, y otros, 2013)

En el año 2009, Netinant, P., en su artículo "Separation of concerns for Multithreads Object-Oriented Programming", Nos explica acerca del inconveniente de cuestiones no relacionadas con el dominio del problema con la programación orientada a objetos en la concurrencia, así mismo señala la importancia de la separación de incumbencias en el diseño e implementación de multitareas con la programación orientada a objetos para proporcionar una serie de beneficios tales como comprensión, reutilización, extensibilidad y adaptabilidad en el diseño y la implementación. (Netinant, 2009)

En el año de 2012, Lester Rodríguez, en su libro Caracterización de la Programación Orientada a Aspectos (POA), implementó una aplicación utilizando el paradigma orientado a objetos y orientado a aspectos con el objetivo de establecer una comparación entre ellos identificando las ventajas y desventajas de este último. (Acosta, 2012)

En el año de 2008, Tabares y otros en su artículo denominado El desarrollo de software orientado a aspectos: un caso práctico para un sistema de ayuda en línea, Explica la teoría de la POA, además de ello desarrolla un caso práctico en la cual se enfrenta a un sistema desarrollado orientado a objetos, re-diseñando la lógica del sistema comparando y obteniendo características positivas en cuanto a un código modularidad y acoplamiento. (Tabares, y otros, 2008)

En el año de 1998, William Harrison y Harold Ossher en su artículo titulado Subject-Oriented Programming (A Critique of Pure Objects), explica en forma teórica que los objetos están ligados con el concepto de identidad (separación de estado y comportamiento), así mismo muchos puntos de vista con abstracciones subjetivas del objeto distorsionan el

[1] Programación Orientada a Aspectos

[2] Programación Orientada a Objetos

concepto de programación orientada a objetos, por lo tanto se debe poner más atención en la identidad para unirlos. Define en conclusión que una clase es como el chip del software pero es pequeño para un paquete grande de funcionalidades. (William Harrison, 1998)

2.2. Base teórica

2.2.1. Programación orientada a objetos

2.2.1.1. Definición

La programación orientada a objetos es un paradigma de programación que usa los objetos en sus interacciones, para diseñar aplicaciones y programas informáticos. Está basado en varias técnicas, incluyendo herencia, cohesión, abstracción, polimorfismo, acoplamiento y encapsulamiento. (DavidA.Watt, 2010)

Es un método de implementación en el que los programas se organizan como colecciones cooperativas de objetos, cada uno de los cuales representa una instancia de alguna clase, y cuyas clases son, todas ellas miembros de una jerarquía de clases unidas mediante relaciones de herencia. (Booch, 2009)

La programación orientada a objetos se puede definir como una técnica o estilo de programación que utiliza objetos como bloques esenciales de construcción. Los elementos básicos de la POO son: objetos, mensajes, métodos y clases. (Barraza, y otros, 2006).

2.2.1.2. Características de la programación orientada a objetos

En (Barraza, y otros, 2006) Señalan que la potencia real de los objetos reside en sus propiedades: abstracción, herencia y polimorfismo.

a) Abstracción

Una abstracción denota las características esenciales de un objeto que lo distinguen de todos los demás tipos de objeto y proporciona así fronteras conceptuales nítidamente definidas respecto a la perspectiva del observador. (Booch, 2009)

Es la capacidad de crear tipos de datos definidos por el usuario. Es una definición de tipo que incluye datos y funciones (métodos) en una sola estructura definida llamada clase. La abstracción de datos es un concepto fundamental en el diseño de un programa, pues permite definir el dominio y la estructura de los datos (atributos), junto con una colección o conjunto de operaciones que tienen acceso a los mismos (métodos). La abstracción permite no

preocuparse de los detalles accesorios, y generalizar y centrarse en los aspectos que permiten tener una visión global del problema. (Barraza, y otros, 2006).

La abstracción es un nuevo tipo de dato definido por el programador; está compuesto por atributos (identificadores o datos) y métodos o acciones que se realizan sobre los atributos, definidos en una sola estructura llamada clase.

$$\text{CLASE} = \text{atributos} + \text{métodos}$$

Definir una abstracción significa describir una entidad del mundo real, con características, propiedades y métodos seleccionados con la posibilidad de emular una interacción entre ellas para lograr un fin.

b) Herencia

La herencia es la jerarquía de clases más importante (elemento esencial en los sistemas orientados a objetos). Básicamente la herencia define una relación entre clases, en la que una clase comparte la estructura de comportamiento definida en una o más clases (lo denominado herencia simple o múltiple). (Booch, 2009)

La herencia "Es la capacidad para crear nuevas clases de objetos que se construyen basados en clases existentes". Además, a estos nuevos objetos creados es posible asignarles nuevos atributos y métodos. (Aguilar, 1996).

Es la relación existente entre dos o más clases, donde una es la principal (madre) y otras son secundarias y dependen (heredan) de ellas (clases "hijas"), donde a la vez, los objetos heredan las características de los objetos de los cuales heredan. (Bahit, 2011).

La herencia permite definir una nueva clase extendiendo una clase ya existente. Esta nueva clase, clase derivada, posee todas las características de la clase más general y, además, la extiende en algún sentido. (Echevarría, y otros, 2004)

La herencia en términos más simples es una propiedad que permite obtener características adicionales de una clase "base" de las que ya tienen.

c) Polimorfismo

(Barraza, y otros, 2006 pág. 12) Nos dicen que polimorfismo proviene de dos raíces griegas: poli, múltiples y morfismo, formas. Esta característica es la capacidad que objetos similares tienen para responder de diferentes formas al mismo mensaje, y permite al programador implementar múltiples formas de un mismo método, dependiendo cada una de ellas de la clase sobre la que se realice la implementación. Esto permite acceder a varios

métodos distintos utilizando el mismo medio de acceso (el mismo nombre).El polimorfismo está muy relacionado con la herencia.

(Aguilar, 1996 pág. 19), con respecto al polimorfismo nos dice que "es la propiedad que indica, literalmente, la posibilidad de que una entidad tome muchas formas. En términos prácticos, el polimorfismo permite referirse a objetos de clases diferentes mediante el mismo elemento de programa y realizar la misma operación de diferentes formas, según sea el objeto que se referencia en ese momento".

Podemos concluir diciendo que es la capacidad de distintos objetos contar con métodos, propiedades y atributos del mismo nombre sin que se interfieran.

Adicionalmente tenemos:

d) Encapsulamiento

La encapsulación o encapsulamiento es la propiedad que permite asegurar que el contenido de la información de un objeto está oculta al mundo exterior: el objeto A no conoce lo que hace el objeto B, y viceversa. La encapsulación (también se conoce como la ocultación de la información), en esencia, es el proceso de ocultar todos los secretos de un objeto que no contribuyen a sus características esenciales (Aguilar, 1996).

El acceso a los datos de una clase debe realizarse de forma controlada, protegiéndolos de accesos no deseados. Cuando se desarrolla una aplicación, a veces es necesario ocultar los tipos de datos usados para que el usuario permanezca independiente de los detalles de los mismos. De esta manera, el usuario no es sensible a los cambios que se puedan producir en los tipos de datos elegidos dentro de una clase. (Echevarría, y otros, 2004)

e) Jerarquía

La jerarquía es una propiedad que permite la ordenación de las abstracciones. Las dos jerarquías más importantes de un sistema complejo son: estructura de clases (jerarquía «es-un» (is-a): generalización/especialización) y estructura de objetos (jerarquía «parte-de» (part-of): agregación) (Aguilar, 1996)

f) Modularidad

La modularidad es la propiedad que permite subdividir una aplicación en partes más pequeñas (llamadas módulos), cada una las cuales debe ser tan independiente como sea posible de la aplicación en sí y de las restantes partes. (Aguilar, 1996)

Característica que permite dividir una aplicación en varias partes más pequeñas (denominadas módulos), independientes unas de otras. (Bahit, 2011)

La modularidad, nos permite poder modificar las características de la clase que definen a un objeto, de forma independiente de las demás clases en la aplicación. En otras palabras, si nuestra aplicación puede dividirse en módulos separados, normalmente clases, y estos

módulos pueden compilarse y modificarse sin afectar a los demás, entonces dicha aplicación ha sido implementada en un lenguaje de programación que soporta la modularidad.

g) Tipos

Un tipo es una caracterización precisa de las propiedades estructurales o de comportamiento que comparten una serie de entidades. (Las clases implementan a los tipos) No son exactamente lo mismo. Los tipos son la puesta en vigor de la clase de los objetos, de modo que los objetos de tipos distintos no pueden intercambiarse o, como mucho, pueden intercambiarse sólo de formas muy restringidas. (Booch, 2009)

h) Concurrencia

La concurrencia es la propiedad que distingue un objeto activo de uno que no está activo. Una vez que se introduce la concurrencia a un sistema, hay que tener en cuenta como los objetos activos sincronizan sus actividades con otros, así como con objetos puramente secuenciales.

Mientras que la programación orientada a objetos se centra en la abstracción de datos, encapsulamiento y herencia, la concurrencia se centra en la abstracción de procesos y la sincronización. El objeto es un concepto que unifica estos dos puntos de vista distintos: cada objeto (extraído de una abstracción del mundo real) puede representar un hilo separado de control (una abstracción de un proceso). Tales objetos se llaman activos.

En un sistema orientado a objetos se puede conceptualizar el mundo como un conjunto de objetos cooperativos, algunos de los cuales son activos y sirven así como centros de la actividad independiente. (Booch, 2009)

i) Persistencia

Un objeto de software ocupa una cierta cantidad de espacio, y existe durante una cierta cantidad de tiempo.

Hay un espacio de existencia que va desde los objetos transitorios que surgen en la evaluación de una expresión hasta los objetos de una base de datos. Este espectro de persistencia abarca lo siguiente:

1. Resultados transitorios en la evaluación de expresiones.
2. Variables locales en la activación de procedimientos.
3. Variables propias, globales y elementos del montículo cuya duración difiere de su ámbito.
4. Datos que existen entre ejecuciones de un programa.
5. Datos que existen entre varias versiones de un programa.
6. Datos que sobreviven al programa.

La persistencia abarca algo más que la mera duración de los datos. En las bases de datos orientadas a objetos, no sólo persiste el estado del objeto, sino que su clase debe

trascender también a cualquier programa individual, de forma que todos los programas interpreten de la misma manera el estado.

La persistencia es la propiedad de un objeto por la que su existencia trasciende el tiempo es decir, el objeto continúa existiendo después de que su creador deja de existir) y / o el espacio (es decir, la posición del objeto varía con respecto al espacio de direcciones en el que fue creado). (Booch, 2009)

2.2.1.3. Fundamentos de la programación orientada a objetos

Cuando se desarrolla un programa orientado a objetos se está creando un modelo de alguna parte del mundo real, las partes con que construye el modelo provienen de los objetos que aparecen en el dominio del problema. Los objetos pueden ser organizados en categorías y una clase describe, en forma abstracta, todos los objetos de un tipo en particular.

Por lo tanto podemos decir lo siguiente:

a) Objeto

Un objeto es una entidad cuyo comportamiento se caracteriza por las acciones que realiza. Con más precisión, un objeto se define como una entidad caracterizada por un estado; su comportamiento se define por las operaciones que puede realizar; es una instancia de una clase; se identifica por un nombre; tiene una visibilidad limitada para otros objetos; se define el objeto mediante su especificación y su implementación. (Aguilar, 1996)

Un objeto tiene estado, comportamiento e identidad; la estructura y comportamiento de objetos similares están definidos en su clase común; los términos instancia y objeto son intercambiables. (Booch, 2009)

b) La naturaleza de los objetos

Un objeto tiene estado, comportamiento e identidad; la estructura y comportamiento de objetos similares están definidos en su clase común; los términos instancia y objeto son intercambiables (Booch, 2009).

1) Estado.

El estado de un objeto abarca todas las propiedades (normalmente estáticas) del mismo más los valores actuales (normalmente dinámicos) de cada una de esas propiedades.

Una propiedad es una característica distintiva o inherente, un rasgo o cualidad que contribuye a hacer que un objeto sea ese objeto y no otro. Todas las propiedades tienen

algún valor. Este valor puede ser una mera cantidad, o puede denotar a otro objeto. Esas cantidades son intemporales, inmutables y no instanciadas; mientras que los objetos existen en el tiempo, son modificables, tienen estado, son instanciados y pueden crearse, destruirse y compartirse.

2) Comportamiento

El comportamiento es cómo actúa y reacciona un objeto, en términos de sus cambios de estado y paso de mensajes. Representa su actividad visible y comprobable exteriormente.

Una operación denota un servicio que una clase ofrece a sus clientes. Las operaciones sobre un objeto son:

1. Modificador: Una operación que altera el estado de un objeto.
2. Selector: Una operación que accede al estado de un objeto, pero no altera ese estado.
3. Iterador: Una operación que permite acceder a todas las partes de un objeto en algún orden perfectamente establecido.
4. Constructor: Una operación que crea un objeto y / o inicializa su estado.
5. Destructor: Una operación que libera el estado de un objeto y / o destruye el propio objeto.

3) Roles y responsabilidades.

Todos los métodos y subprogramas libres asociados a un objeto concreto forman su protocolo. El protocolo define así la envoltura del comportamiento admisible en un objeto, englobando la visión estática y dinámica completa del mismo.

Para la mayoría de las abstracciones es útil dividir este protocolo en grupos lógicos de comportamiento. Estas colecciones, que constituyen una partición del espacio de comportamiento de un objeto, denotan papeles que un objeto puede desempeñar. Un papel es una máscara que se pone un objeto y por tanto define un contrato entre una abstracción y sus clientes.

Las responsabilidades de un objeto «incluyen dos elementos clave»: el conocimiento que un objeto mantiene y las acciones que puede llevar a cabo. Las responsabilidades están encaminadas a transmitir un sentido del propósito de un objeto y de su lugar en el sistema. Las responsabilidades de un objeto son todos los servicios que proporciona para todos los contratos que soporta. (Booch, 2009)

4) *Los objetos como máquinas.*

La existencia de un estado significa que el orden en el que se invocan las operaciones es importante. Esto da lugar a la idea de que cada objeto es como una pequeña máquina independiente. Se pueden entonces clasificar los objetos como activos o pasivos. Un objeto activo es aquel que comprende su propio hilo de control, mientras que un objeto pasivo no.

Los objetos activos suelen ser autónomos, pueden exhibir algún comportamiento sin que ningún otro objeto opere sobre ellos. Los pasivos, sólo pueden padecer un cambio de estado cuando se actúa explícitamente sobre ellos.

5) *Identidad*

La identidad es aquella propiedad de un objeto que lo distingue de todos los demás objetos.

i. Semántica:

La mayoría de los lenguajes de programación y bases de datos utilizan nombres de variable para distinguir objetos temporales, mezclando la posibilidad de acceder a ellos con su identidad.

ii. Copia, asignación e igualdad:

La compartición estructural se da cuando la identidad de un objeto recibe un alias a través de un segundo nombre. Es una operación de copia. La asignación es del mismo modo en general una operación de copia. El problema de la igualdad está en relación muy estrecha con el de la asignación. Aunque se presenta como un concepto simple la igualdad puede presentar una de dos cosas.

Primero, puede significar que dos nombres designan el mismo objeto y segundo, puede significar que dos nombres designan objetos distintos cuyos estados son iguales.

iii. Espacio de vida de un objeto:

El tiempo de vida de un objeto se extiende desde el momento en que se crea por primera vez (y consume así espacio por primera vez) hasta que ese espacio se recupera. Para crear explícitamente un objeto hay que declararlo o bien asignarle memoria dinámicamente.

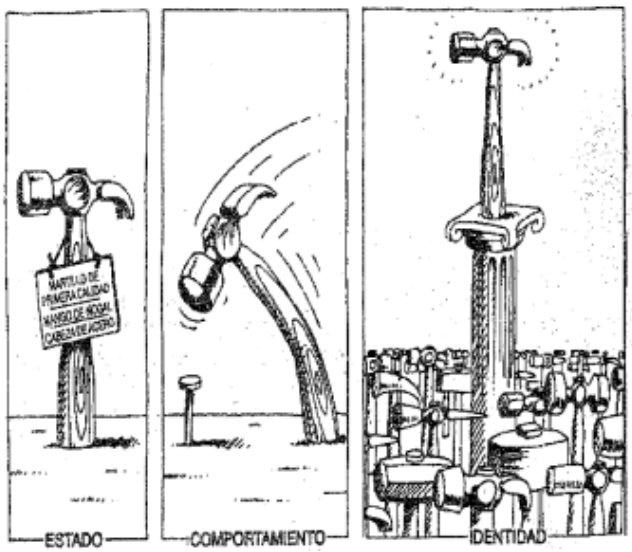


Ilustración 1: Estado, comportamiento e identidad

Fuente: (Booch, 2009)

6) *Tipos de relaciones*

La relación entre dos objetos cualesquiera abarca las suposiciones que cada uno realiza acerca del otro, incluyendo qué operaciones pueden realizarse y qué comportamiento se obtiene. Hay dos tipos de jerarquías de objetos:

i. Enlaces:

Una conexión física o conceptual entre objetos. Un objeto colabora con otros objetos a través de sus enlaces con éstos. Un enlace denota la asociación específica por la cual un objeto (el cliente) utiliza los servicios de otro objeto (el suministrador o servidor), o a través de la cual un objeto puede comunicarse con otro. Un objeto puede desempeñar uno de tres papeles:

1. Actor Un objeto que puede operar sobre otros objetos pero nunca se opera sobre él por parte de otros objetos; en algunos contextos, los términos objeto activo y actor son equivalentes.
2. Servidor Un objeto que nunca opera sobre otros objetos; sólo otros objetos operan sobre él.
3. Agente Un objeto que puede operar sobre otros objetos y además otros objetos pueden operar sobre él; un agente se crea normalmente para realizar algún trabajo en nombre de un actor u otro agente.

ii. Agregación:

Mientras que los enlaces denotan relaciones de igual-a-igual o cliente / servidor, la agregación denota una jerarquía todo / parte, con la capacidad de ir desde el todo (también llamado el agregado) hasta sus partes (conocido también como atributos).

La agregación puede o no denotar contención física. Existen claros pros y contras entre los enlaces y la agregación. La agregación es a veces mejor porque encapsula partes y secretos del todo. A veces son mejores los enlaces porque permiten acoplamientos más débiles entre los objetos.

c) Clase:

Una clase, puede definirse como un tipo de datos cuyas variables son objetos o instancias de una clase. Puede decirse que el concepto de clase es estático y el concepto de objeto es dinámico; ya que, sólo existe en tiempo de ejecución. (Echevarría, y otros, 2004).

Una clase es un conjunto de objetos que comparten una estructura común y un comportamiento común. Un solo objeto no es más que una instancia de una clase. Un objeto no es una clase, aunque una clase sí puede ser un objeto. La clase es un vehículo necesario, pero no suficiente, para la descomposición. (laddad, 2009).

Dicho de otra forma es la descripción abstracta de un grupo de objetos con propiedades similares (atributos), comportamiento común (métodos), relaciones comunes con otros objetos y semántica común.

En (Booch, 2009) se caracteriza a una clase de la siguiente manera:

1) Interfaz e implementación.

El interfaz de una clase proporciona su visión externa y por tanto enfatiza la abstracción a la vez que oculta su estructura y los secretos de comportamiento. Se compone principalmente de todas las operaciones aplicables a instancias de esta clase, pero también puede incluir la declaración de otras clases, constantes, variables, expresiones, según se necesiten para completar la abstracción.

Por contraste, la implementación es su visión interna que engloba los secretos de su comportamiento. La implementación de una clase se compone principalmente de la implementación de todas las operaciones definidas en el interfaz de la misma. El interfaz puede dividirse en:

1. Público (Public) Una declaración accesible a todos los clientes.
2. Protegida (Protected) Una declaración accesible sólo a la propia clase, sus subclases.
3. Privada (Private) Una declaración accesible sólo a la propia clase.

2) Relaciones entre clases

i. Asociación:

Una asociación sólo denota una dependencia semántica y no establece la dirección de esta dependencia (a menos que se diga lo contrario, una asociación implica relación

bidireccional) ni establece la forma exacta en que una clase se relaciona con otra (sólo puede denotarse esta semántica nombrando el papel que desempeña cada clase en relación con la otra).

ii. Uso, Dependencia o Instanciación.

Una clase usa los servicios de otra. Mientras que una asociación denota una conexión semántica bidireccional, una relación «de uso» es un posible refinamiento de una asociación, por el que se establece qué abstracción es el cliente y qué abstracción es el servidor que proporciona ciertos servicios.

iii. Herencia

La herencia es una relación entre clases en la que una clase comparte la estructura y / o comportamiento definidos en una (herencia simple) o más clases (herencia múltiple). La clase de las que otras heredan se llama superclase. La clase que hereda de otra o más clases se denomina subclase. La herencia define por tanto una jerarquía «de tipos» entre clases, en las que una subclase hereda de una o más superclases.

iv. Agregación

Las relaciones de agregación entre clases tienen un paralelismo directo con las relaciones de agregación entre los objetos correspondientes a esas clases.

- Contención por valor: un tipo de contención física que significa que el objeto no existe independientemente de la instancia que lo encierra. El tiempo de vida de ambos objetos está en íntima conexión. (composición)
- Contención por referencia: tipo de agregación menos directo. En este caso, la clase sigue denotando al todo, y una de sus partes sigue siendo una instancia de la clase, aunque ahora hay que acceder a esa parte indirectamente. Sus tiempos de vida no están tan estrechamente emparejados (agregación)

v. Metaclases

Es una clase cuyas instancias son, ellas mismas, clases. Se justifica su necesidad al hacer notar que en un sistema bajo desarrollo, una clase proporciona al programador una interfaz para comunicarse con la definición de objetos. Para este uso de las clases, es extremadamente útil para ellas ser objetos, de forma que puedan ser manipulados de la misma forma que todas las demás descripciones.

3) Atributo

Un atributo es una propiedad o característica de una clase y describe un rango de valores que la propiedad podrá contener en los objetos (instancias) de la clase. Una clase podrá contener varios o ningún atributo. (Schmuller, 2003)

Las clases tienen atributos que representan alguna propiedad de la clase que comparten todos los objetos de esa clase.

4) Operación y método

Una operación es una función o transformación que se puede aplicar o que puede ser aplicada por los objetos de una clase. Todos los objetos de una clase comparten las mismas operaciones, y una misma operación puede aplicarse a muchas clases distintas. Un método es la implementación de una operación para una clase.

Es un programa asociado a un objeto (o a una clase de objetos), cuya ejecución se desencadena mediante un mensaje. En analogía con un lenguaje basado en procedimiento se le llamaría función. (Tovilla, 2009)

2.2.1.4. Lenguajes de programación orientados a objetos

Existen varias clasificaciones de lenguajes de programación orientados a objetos, una ampliamente aceptada es la dada por Wegner.

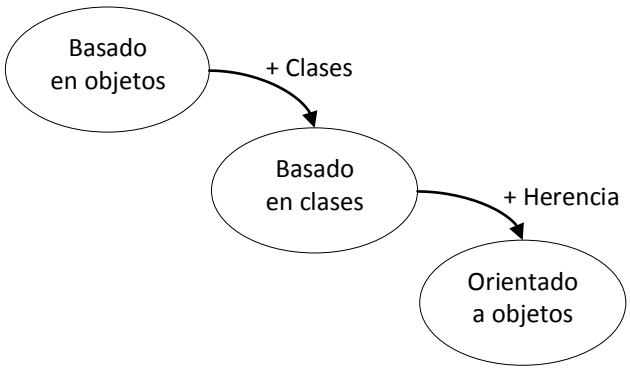


Ilustración 2: Clasificación de lenguajes orientados a objetos según Wegner (1987).
Fuente: (Aguilar, 1996).

La clasificación de Wegner divide los lenguajes en tres categorías (Aguilar, 1996):

- Lenguajes basados en objetos que soportan objetos Es decir, disponen de componentes caracterizados por un conjunto de operaciones (comportamiento) y un estado.
- Lenguajes basados en clases que implican objetos y clases Es decir, disponen de componentes tipo clase con operaciones y estado común. Una clase de un objeto se

construye con un «interfaz» que especifica las operaciones posibles y un «cuerpo» que implementa dichas operaciones.

- Lenguajes orientados a objetos que además de objetos y clases ofrecen mecanismos de herencia entre clases. Esto es, la posibilidad de derivar operaciones y atributos de una clase (superclase) a sus subclases. Esta última desglosada en dos:

  - Lenguajes orientados a objetos **puros**. Soportan en su totalidad el paradigma de orientación a objetos: Smalltalk, Eiffel, Simula.
  - Lenguajes orientados a objetos **híbridos** Soportan en su totalidad el paradigma de orientación a objetos sobre un núcleo de lenguaje híbrido:
    - C++ (extensión de C: Borland C++, Microsoft C++, Turbo C++, Visual C++, Symantec, Watcom)
    - Objective-C (extensión de C)
    - Object COBOL. (extensión de COBOL).
    - Object Pascal (extensión de Pascal: Turbo/Borland Pascal)
    - Visual Object (extensión de Clipper).
    - Delphi (extensión de Turbo Pascal 70)
    - Java (extensión de C++ y Ada-95)
    - PHP.

2.2.1.5. Ventajas de la programación orientada a objetos

Las principales ventajas de la programación orientada a objetos son (Barraza, y otros, 2006):

- Las herramientas POO nos ayudan a mejorar la complejidad.
- Mejora la productividad debido a la reutilización de código (herencia).
- Mejora la seguridad y calidad.
- Permite generar programas modulares mejor estructurados.

2.2.1.6. Inconvenientes de la programación orientada a objetos

Se cita algunos inconvenientes respecto a la programación orientada a objetos. (Martinez, 2014):

- Son más grandes que otros programas, consume más memoria y espacio en disco.
- Requiere mucho trabajo para crear, incluyendo la creación de diagramas en la fase de planificación y la codificación en la fase de ejecución.

Finalmente podemos concluir:

- La programación orientada a objetos no es la solución definitiva, pero permite construir programas complejos a partir de entidades de software más simples llamadas objetos, que son instancias reales o muestras de clases, lo cual permite heredar datos y códigos de aplicaciones existentes. Esto mejora la productividad del programador y

facilita la extensión y la reutilización de clases en otras aplicaciones con mínima modificación al código original. (Barraza, y otros, 2006)

- Uno de los principales problemas de la programación orientada a objetos, es la incapacidad de abstraer eficientemente la funcionalidad que se repite en distintos módulos que no estén relacionados en la jerarquía de clases. Uno de los ejemplos más claros de este inconveniente son las instrucciones de bitácora y de traza de ejecución. Si en un sistema se quisiera mostrar un mensaje cada vez que comienza la ejecución de un método, se tendría que agregar dicha instrucción en todos los métodos del sistema. Supongamos, que se incorpora una instrucción en cada uno de los métodos del sistema que muestre el nombre del método al cual se está invocando. Si en el futuro, se decide que es necesario ver además, los valores de los argumentos con los cuales fue llamado cada método, se necesitaría hacer una modificación en cada uno de los lugares donde fue aplicada la instrucción originalmente. Dicha modificación implica un esfuerzo lineal en comparación con el tamaño del sistema. Todos estos problemas surgen de la falta de mecanismos del paradigma para la abstracción de comportamiento transversal. (Erdody, 2006)

2.2.1.7. Metodologías orientadas a objetos

a) Metodología OMT

(Marcano, 2011) OMT[3] pone énfasis en especificaciones declarativas de la información, para capturar los requerimientos, especificaciones imperativas para poder descender prematuramente en el diseño, declaraciones que permiten optimizar los estados.

Los puntos más importantes para esta metodología son los siguientes:

- Poner énfasis en el análisis y no en el desarrollo.
- Poner énfasis en los datos más que en las funciones, lo que proporciona estabilidad al proceso de desarrollo.
- Utilizar una notación común en todas las fases a través de tres modelos que capturan los aspectos estáticos, dinámicos y funcionales que combinados proveen una descripción completa del software.

La Metodología OMT divide el proceso de desarrollo en tres partes aisladas: análisis, diseño e implantación.

1) Análisis:

El analista construye un modelo del dominio del problema, mostrando sus propiedades más importantes. El modelo de análisis es una abstracción resumida y precisa de lo que debe de hacer el sistema deseado y no de la forma en que se hará. Los elementos del modelo deben ser conceptos del dominio de aplicación y no conceptos informáticos tales como estructuras de datos. Un buen modelo debe poder ser entendido y criticado por expertos en el dominio del problema que no tengan conocimientos informáticos.

[3] Técnica de modelado de objetos

2) *Diseño del sistema:*

El diseñador del sistema toma decisiones de alto nivel sobre la arquitectura del mismo. Durante esta fase el sistema se organiza en subsistemas basándose tanto en la estructura del análisis como en la propuesta. Se selecciona una estrategia para afrontar el problema.

3) *Diseño de objetos:*

El diseñador de objetos construye un modelo de diseño basándose en el modelo de análisis, pero incorporando detalles de implementación. El diseño de objetos se centra en las estructuras de datos y algoritmos que son necesarios para implementar cada clase. OMT describe la forma en que el diseño puede ser implementado en distintos lenguajes (orientados y no orientados a objetos, base de datos, etc.).

4) *Implementación:*

Las clases de objetos y relaciones desarrolladas durante el análisis de objetos se traducen finalmente a una implementación concreta. Durante la fase de implementación es importante tener en cuenta los principios de la ingeniería del software de forma que la correspondencia con el diseño sea directa y el sistema implementado sea flexible y extensible. No tiene sentido que utilicemos análisis orientado a objetos y diseño orientado a objetos de forma que potenciemos la reutilización de código y la correspondencia entre el dominio del problema y el sistema informático, si luego perdemos todas estas ventajas con una implementación de mala calidad.

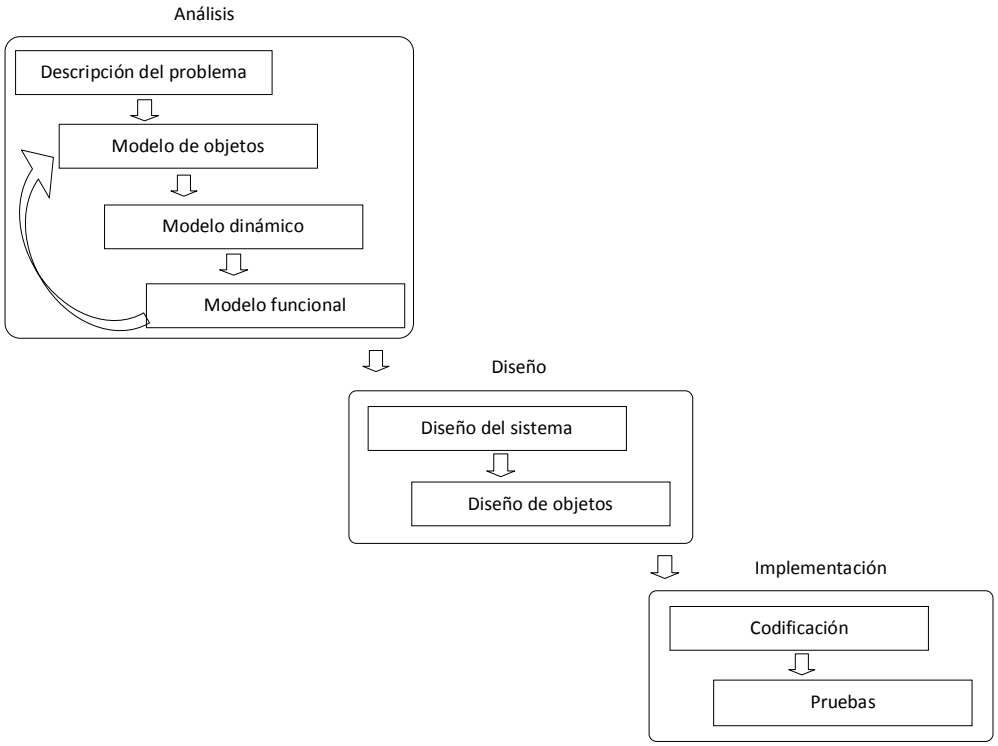


Ilustración 3: Ciclo de Vida Metodología OMT

Fuente: (Marcano, 2011).

b) Metodología BOOCH

Object Oriented Design - Grady Booch, "El diseño orientado a objetos es el método que lleva a una descomposición Orientado a Objetos. Aplicando DOO[4], se crea software resistente al cambio y escrito con economía de expresión. Se logra un mayor nivel de confianza en la corrección del software a través de la división inteligente de su espacio de estados. En última instancia, se reducen los riesgos inherentes al desarrollo de sistemas" (Booch, 1994).

El método original de Booch comienza por un análisis de flujo de datos, que se utiliza entonces como ayuda para identificar objetos, buscando tanto objetos concretos como objetos abstractos en el espacio del problema, que se encontraran a partir de las burbujas y almacenes de datos en el diagrama de flujo de datos (DFD[5]). (Romero Guillen, 2005)

Abarca un "micro proceso de desarrollo" y un "macro proceso de desarrollo".

1) *Macro Proceso*

Engloba una actividad de planificación arquitectónica, que agrupa capas de objetos por nivel de abstracción. Además identifica situaciones relevantes.

Crea un prototipo de diseño y valida el prototipo aplicándolo a situaciones de uso. Es un proceso de alto nivel.

Consta de 5 pasos:

- *Conceptualización,* se establecen las necesidades básicas
- *Análisis,* modelar un comportamiento deseado
- *Diseño,* se crea una arquitectura
- *Evolución,* Evolución de la aplicación(a través de refinamientos sucesivos.
- *Mantenimiento,* gestionar la entrega luego de la evolución

2) *Micro Proceso*

En esta parte se desarrolla lo siguiente (Guzman, y otros, 2007):

- Define un conjunto de "reglas" que regulan el uso de operaciones y atributos, de reglas, y políticas.
- Desarrolla situaciones que describen la semántica de las reglas y política.
- Crea un prototipo para cada política.
- Instrumenta y refina el prototipo.

[4] Desarrollo orientado a objetos

[5] Diagrama de flujo de datos

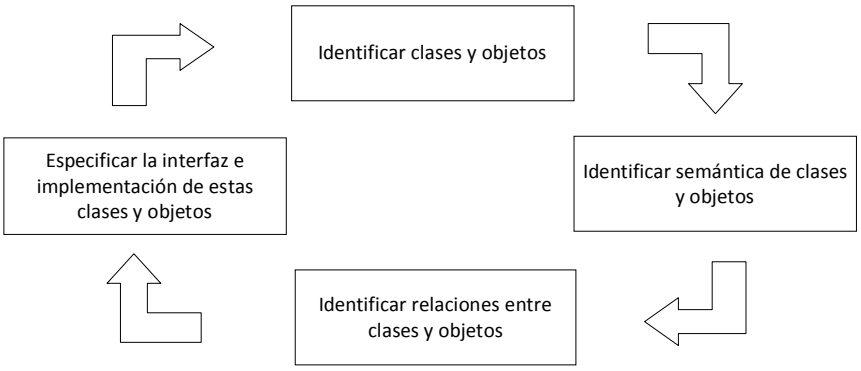


Ilustración 4: Pasos Micro proceso

Fuente: (Guzman, y otros, 2007).

c) Metodología Rational Unified Process (RUP)

El Rational Unified Process es un Proceso de Ingeniería de Software. Proporciona un enfoque disciplinado para la asignación de tareas y responsabilidades dentro de una organización de desarrollo. Su objetivo es asegurar la producción de software de alta calidad que satisfaga las necesidades de sus usuarios finales, dentro de un horario predecible y propuesto. (Rational Software, 2003).

El proceso unificado conocido como RUP, es un modelo de software que permite el desarrollo de software a gran escala, mediante un proceso continuo de pruebas y retroalimentación, garantizando el cumplimiento de ciertos estándares de calidad. Aunque con el inconveniente de generar mayor complejidad en los controles de administración del mismo. Sin embargo, los beneficios obtenidos recompensan el esfuerzo invertido en este aspecto.

Estructura del ciclo de vida del proceso de desarrollo unificado:

1) Fase de concepción

Esta fase tiene como propósito definir y acordar el alcance del proyecto con los patrocinadores, identificar los riesgos potenciales asociados al proyecto, proponer una visión muy general de la arquitectura de software y producir el plan de las fases y el de iteraciones.

2) Fase de elaboración

En la fase de elaboración se seleccionan los casos de uso que permiten definir la arquitectura base del sistema y se desarrollaran en esta fase, se realiza la especificación de los casos de uso seleccionados y el primer análisis del dominio del problema, se diseña la solución preliminar.

3) Fase de construcción

El propósito de esta fase es completar la funcionalidad del sistema, para ello se deben clarificar los requerimientos pendientes, administrar los cambios de acuerdo a las evaluaciones realizados por los usuarios y se realizan las mejoras para el proyecto.

4) Fase de transición

El propósito de esta fase es asegurar que el software esté disponible para los usuarios finales, ajustar los errores y defectos encontrados en las pruebas de aceptación, capacitar a los usuarios y proveer el soporte técnico necesario. Se debe verificar que el producto cumpla con las especificaciones entregadas por las personas involucradas en el proyecto.

El proceso del desarrollo del software según RUP puede ser descripto en dos dimensiones (Murillo, 2012):

- El eje horizontal representa el tiempo y muestra los aspectos dinámicos del proceso siendo expresados en términos de ciclos, fases, iteraciones e hitos
- El eje vertical representa los aspectos estáticos del proceso, cómo se describe en términos de actividades, artefactos, empleados y flujos de trabajo

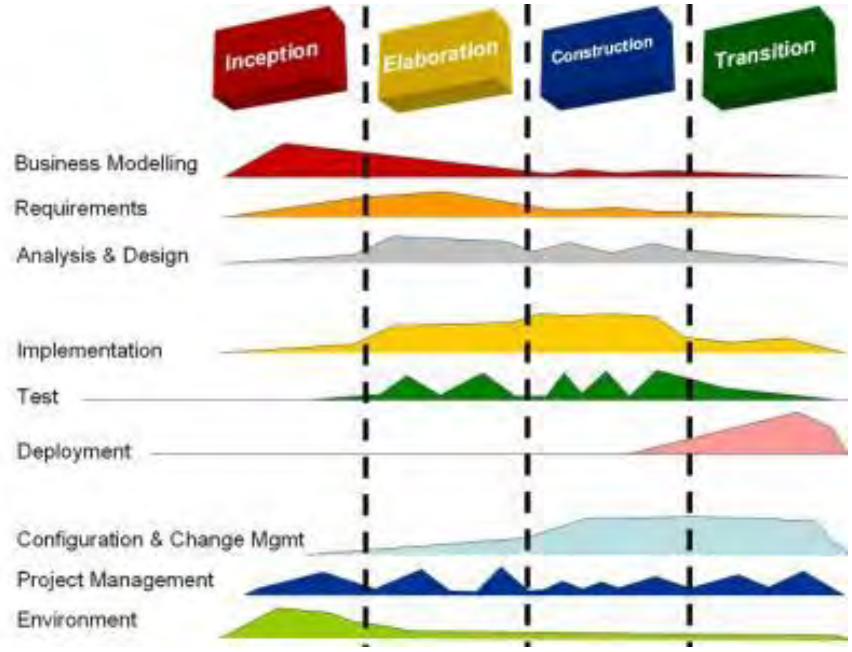


Ilustración 5: Rational Unified Process

Fuente: (RUP - Rational IBM, 2002).

d) OOram (Object-Oriented Role Analysis and Modeling)

Análisis y modelado de rol orientado a objetos, El Método OORAM se remonta a principios de los años 70 e introduce el concepto de modelo de roles como principal mecanismo de abstracción.

El modelo de roles describe los objetos que participan en una actividad y las interacciones entre ellos. El método OOram se define como "un marco de referencia para una familia de metodologías orientadas-a-objetos". El modelo de roles es una abstracción del modelo de objetos donde se reconoce un patrón de objetos y se describe como un correspondiente patrón de roles y es que la clase, se apoya en las capacidades (funcionalidad absoluta) de los objetos, mientras que el rol se apoya en la posición y responsabilidades de un objeto respecto de una estructura de otros tantos, que es precisamente lo que percibimos cuando examinamos un conjunto de entidades/objetos en colaboración. El modelo de roles no representa la descripción de la estructura entera de objetos observada, sino más bien de porciones de la misma, segmentadas temporalmente, denominadas "áreas de interés", y que muestran fenómenos colaborativos de interés. Resulta, así, que un modelo de roles es un modelo orientado-a-objetos de una estructura de objetos (González, y otros, 2008):

OOram plantea tres distintas perspectivas: contextual, externa e interna, y provee diferentes "vistas" aplicables según mejor convenga, siguiendo la acertada idea de que no hay que usar todos los recursos para describir un modelo, y que determinadas vistas, reunidas en un subconjunto concreto, tornan el modelo más comprensible que otras.

OOram se plantea un modelado orientado a objetos que retrasa la identificación de las clases hasta la fase de implementación, usando los conceptos de rol y tipo, respectivamente. Los roles permiten describir los servicios y características de los objetos sin necesidad de conocer, comprometerse o preocuparse por las clases que los implementarán. Un modelo de roles describe los objetos que participan en una actividad y las interacciones entre ellos; contiene un conjunto de roles, de modo que todos los objetos que ocupan una misma posición en la estructura son representados por un rol. Un rol describe el comportamiento de un objeto en el contexto de una actividad. (Ibáñez, y otros, 2000)

e) Objectory

ObjectOry, una metodología basada en el paradigma de orientación a objetos, fue creada por la compañía Objectory Systems, fundada en 1987 por Jacobson. Es un proceso organizado para la construcción industrial de software. Este proceso de diseño está guiado por casos de uso, una técnica que centra el entendimiento de un sistema en la forma en la cual es usado (González, y otros, 2008).

Etapas y definición de entregas:

1) *Modelo de requerimientos*
   - Modelo de Casos de Uso con interfaces de usuario del sistema.
   - Modelo de objetos del dominio.

2) *Modelo de análisis*
   - Modelo de objetos con objetos Entidades, de Interfaz y de Control

3) *Modelo de diseño*
   - Modelo de paquetes. Definición de módulos en la implementación y detalle de las clases de diseño en cada uno de ellos.

4) *Implementación*
   - Código de las clases, organizado por paquetes.

5) *Pruebas*
   - Definición de pruebas de unidad
   - Definición de pruebas de protocolo de clases.

2.2.1.8. Otras metodologías orientadas a objetos

a) Catalysis:

Un método orientado a objetos que fusiona mucho del trabajo reciente en métodos orientados a objetos, y además ofrece técnicas específicas para modelar componentes distribuidos.

b) Shlaer/Mellor:

El método para diseñar sistemas de tiempo real, puesto en marcha por Sally Shlaer y Steven Mellor en dos libros de 1991, Ciclos de vida de Objetos, modelando el Mundo en Estados y Ciclos de vida de Objetos, Modelando el mundo en Datos (Prentice Hall).

c) Fusion:

Desarrollado en Hewlett Packard a mediados de los noventa como primer intento de un método de diseño orientado a objetos estándar. Combina OMT y Booch con tarjetas CRC[6] y métodos formales.

2.2.2. Programación orientada a aspectos

2.2.2.1. Definición

La programación orientada a aspectos es un paradigma de programación cuyo principal objetivo es lograr una adecuada modularización de los conceptos involucrados en una aplicación, esto se traduce en lograr la separación entre los requerimientos funcionales de los no funcionales para obtener un mejor entendimiento de los conceptos, eliminando la dispersión del código y haciendo que las implementaciones resulten más comprensibles, adaptables y reutilizables. (Arismendi, y otros, 2008)

[6] Tarjetas clase-responsabilidad-comunicación, son anotaciones detalladas de las características para un sistema.

2.2.2.2. Comprendiendo la separación de intereses

En la Ingeniería del Software se ha utilizado durante años el principio de la separación de incumbencias (Separation of Concerns) para gestionar la complejidad del desarrollo de software mediante la separación de las funcionalidades principales de la aplicación de otras partes con un propósito específico, ortogonales a la funcionalidad principal (autenticación, administración, rendimiento, logging, etc.).

Es posible ver un sistema como la combinación de múltiples intereses. Se define interés como cualquier aspecto que le concierne a un programa, ya sea relacionado a la infraestructura, al negocio, a los requerimientos o a las estructuras de diseño.

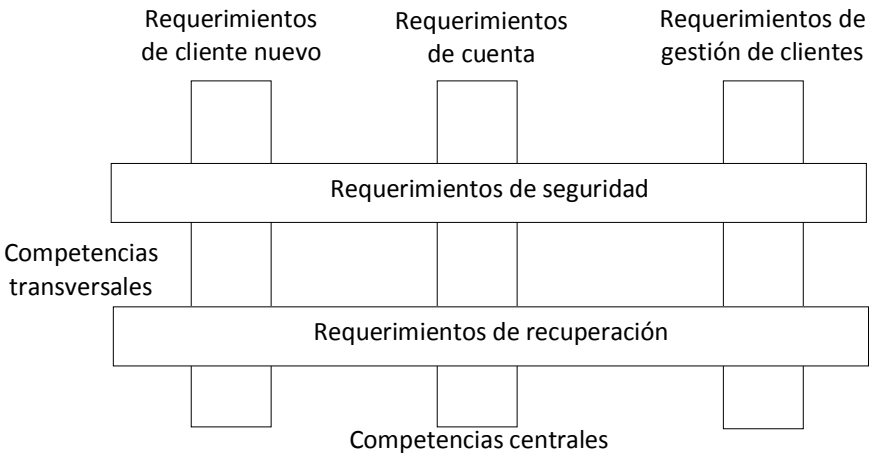


Ilustración 6: Incumbencias o intereses en un sistema

Fuente: (Sommerville, 2011).

El principio de separación de las competencias o intereses promueve el buen diseño e implementaciones reutilizables. Es un factor determinante en la evolución continua de los paradigmas de desarrollo porque los ingenieros de software necesitan mecanismos de desarrollo adecuadas para lograrlo. Aunque el paradigma orientado a objetos ya apoya este principio bastante bien, "aspecto" implica ofrecerles enfoques y técnicas mejoradas que permiten una mejor separación de las competencias en todo el ciclo de vida de desarrollo. (Brichau, y otros, 2005).

Existen varios tipos de competencias o intereses para (Sommerville, 2011):

- Competencias funcionales, que se relacionan con la funcionalidad específica a incluir en un sistema. Por ejemplo, en un sistema de control ferroviario, una competencia funcional específica consiste en el frenado del tren.
- Competencias de calidad del servicio, las cuales se relacionan con el comportamiento no funcional de un sistema. Éstas incluyen características como rendimiento, fiabilidad y disponibilidad.

- Competencias de política, que se relacionan con políticas generales que rigen el uso de un sistema. Las limitaciones políticas incluyen competencias de seguridad y protección, y las competencias relacionadas con las reglas de negocio.
- Competencias de sistema, las cuales se relacionan con atributos del sistema como un todo, tales como su mantenibilidad o configurabilidad.
- Competencias organizacionales, que se relacionan con las metas y prioridades de la organización. Entre ellas se incluyen producir un sistema dentro de presupuesto, usar los activos de software existentes y mantener la imagen de la organización.

Las competencias centrales de un sistema son aquellas competencias funcionales relacionadas con su propósito primario. Así, para un sistema de información de pacientes en un hospital, las competencias funcionales centrales son la creación, edición, recuperación y gestión de registros de pacientes. Además de las competencias centrales, los grandes sistemas tienen también competencias funcionales secundarias. Éstas pueden incluir funcionalidad que comparte información con las competencias centrales, o que se requiere para que el sistema pueda cumplir con sus requerimientos no funcionales.

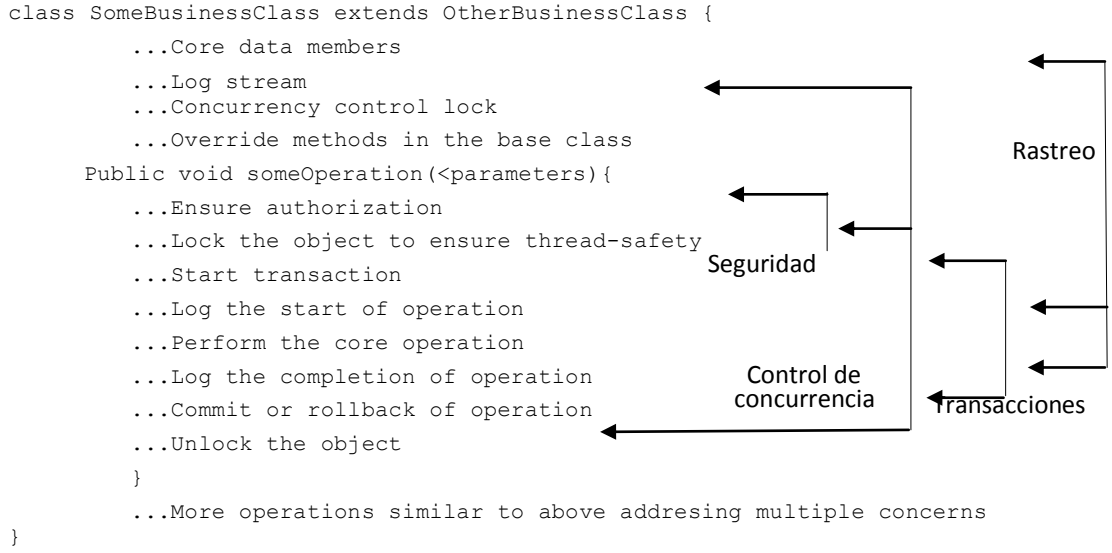


Ilustración 7: Implementación de la lógica del negocio con incumbencias transversales en la forma tradicional.

Fuente: (laddad, 2009).

En un sistema orientado a aspectos, es posible representar estas competencias transversales como aspectos separados. Un aspecto incluye una especificación de dónde debe tejerse en el programa la competencia transversal, y el código para implementar dicha competencia.

2.2.2.3. Aspecto

Gregor Kickzales (Kiczales), creador del paradigma de Programación Orientada a Aspectos (POA), define a un Aspecto como: "una unidad modular que se disemina por la

estructura de otras unidades funcionales. Los Aspectos existen tanto en la etapa de diseño como en la de implementación. Un Aspecto de diseño es una unidad modular del diseño que se entremezcla en la estructura de otras partes del diseño. Un Aspecto de implementación es una unidad modular del programa que aparece en otras unidades modulares del programa".

Lo que propone es agrupar los lenguajes orientados a objetos, los procedurales y funcionales como lenguajes de procedimiento generalizado LPG[7], ya que sus mecanismos claves de abstracción y composición pueden verse como agrupados bajo una misma raíz. Los métodos de diseño que han surgido para los LPG tienden a dividir el sistema en unidades de comportamiento o funciones.

En general, decimos que dos propiedades se entrecruzan si al implementarse deben componerse de manera diferente, y aún deban ser coordinadas. Esto se debe a que tienen un comportamiento en común. Al proveer los LPG solamente un único medio de composición, es el programador quien debe realizar la tarea extra descomponer manualmente las propiedades que se entrecruzan, lo cual lleva a un oscurecimiento y a un aumento de la complejidad del código.

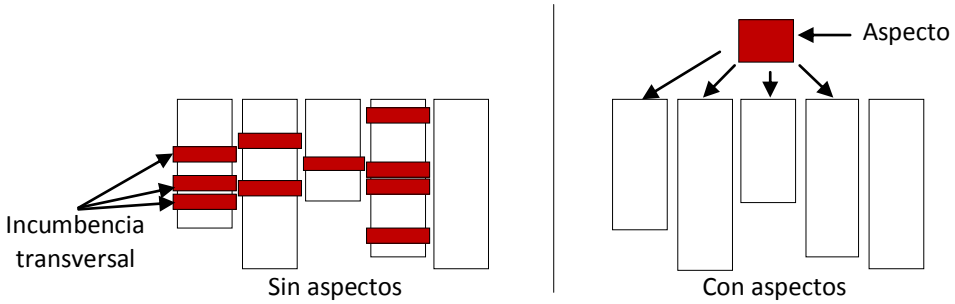


Ilustración 8: Implementación de incumbencias transversales sin aspectos y con aspectos
Fuente: (aspectc.org, Using AspectC++ for Qt Application Development, 2011).

Gracias a todo esto podemos diferenciar los aspectos de los demás componentes del sistema. Al momento de implementar una propiedad, tenemos que la misma tomará una de las dos siguientes formas:

Un componente: Si puede claramente encapsularse dentro de un procedimiento. Un elemento es claramente encapsulado si está bien localizado, es fácilmente accesible y resulta sencillo componerlo

Un aspecto: Si no puede claramente encapsularse en un procedimiento. (Ejemplo: autenticación, Accesibilidad, sincronización, seguridad entre otros.)

[7] Lenguajes de procedimiento generalizado

Teniendo estas definiciones es posible definir un aspecto como una propiedad de un sistema que no puede ser encapsulada claramente y envuelve diversos componentes funcionales.

Las características más relevantes de los aspectos para ser considerados como tales son (Fernando Asteasuain, 2002):

- Un aspecto no debe interferir con otro.
- Cada aspecto debe ser claramente identificable.
- Los aspectos deben ser fácilmente intercambiables

2.2.2.4. Fundamentos de la programación orientada a aspectos

La idea de la solución de la orientación aspecto tiene por objeto de eliminar la transversalidad de incumbencias en un único módulo bien definido (llamado aspecto) que puede estar compuesto de forma automática con el resto del programa (la denominada "Núcleo" o "base"). Cada técnica orientada a aspectos se define por un modelo Punto de corte. De acorde a esto se caracteriza por tres elementos: Join Points, Join point selection mechanism y un medio para especificar la semántica en puntos de unión (Keuler, 2011):

- **Join Points (Punto de unión):** Un punto de unión es un punto en un programa donde el comportamiento puede ser alterado, aumentado o eliminado. En otras palabras, los puntos de unión están considerado el tipo de interfaz entre los aspectos y elementos básicos. Ejemplo: unir puntos en el nivel de programación como son llamadas a métodos, acceso a campos, o llamadas de constructor.

  Dicho de otra manera un evento en un programa en ejecución en que puede ejecutarse el consejo asociado con un aspecto.

- **Join point selection mechanism:** Modelo de punto de unión (Punto de corte): Un punto de corte puede ser definida como una consulta en el propio programa. La manera más simple y trivial para capturar puntos de unión es enumerarlas ya sea utilizando sus nombres o Ids o cualquier propiedad pueden ser utilizados para su identificación exacta. Otra técnica que puede ser visto como una extensión de la enumeración de es capturar puntos de unión deseadas, por un patrón (en su mayoría un nombre o patrón de la firma). Dicho de otra forma el conjunto de eventos que se pueden referenciar en un punto de corte.

- **Adaptation (advice):** En términos orientados a aspectos seria “avisos”, se consideran como acciones que se ejecutan en cada punto de enlace dentro de un mismo punto de corte, estas acciones se traducen en rutinas o fragmentos de código.

En general, hay tres maneras de cómo el aviso puede ser introducido como son: antes, después, y alrededor.

- **Weaving (Tejido)**: El proceso de transformación que pone elementos de base y aspectos juntos se llama "tejer", la incorporación de código del consejo (advice) en los puntos de enlace (join point) especificados por un tejedor (weaving) de aspectos.

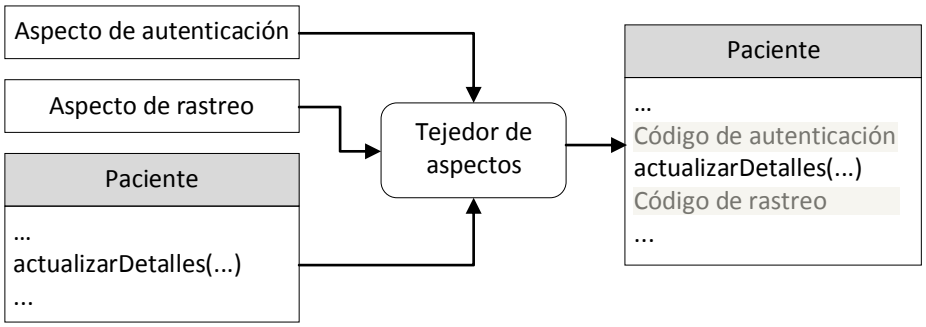


Ilustración 9: Función del tejedor de aspectos

Fuente: (Sommerville, 2011).

Los elementos básicos de la programación orientada a aspectos, la funcionalidad básica (requerimientos funcionales, lenguaje base) es capturada por el tejedor de aspectos, y luego cada aspecto (incumbencia transversal) es tejido con la funcionalidad básica siendo los puntos de corte quienes indican en que sección de código del lenguaje base deben escribirse.

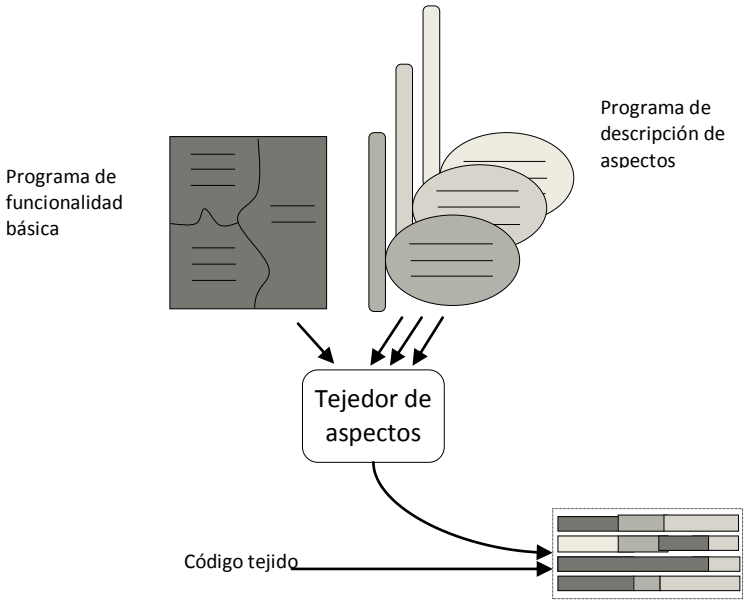


Ilustración 10: Proceso de tejido

Fuente: Elaboración propia

Existen tres enfoques diferentes al tejido de aspectos con el código base:

1. Pre procesamiento de código fuente, en el que un tejedor toma entrada de código fuente y genera nuevo código fuente en un lenguaje como Java o C++, que luego pueden compilarse usando el compilador de lenguaje estándar. Este enfoque se adoptó para el lenguaje AspectX con su asociado XWeaver (Birrer et al., 2005).
2. Tejido de tiempo de vinculación, en el que el compilador se modifica para incluir un tejedor de aspectos. Un lenguaje orientado a aspectos, como AspectJ, se procesa y genera bytecode Java estándar. Entonces éste puede ejecutarse directamente mediante un intérprete Java o procesarse aún más para generar código de máquina nativo.
3. Tejido dinámico en tiempo de ejecución. En este caso, se monitorizan los puntos de enlace y, cuando ocurre un evento referenciado en un punto de corte, se integra el consejo correspondiente con el programa en ejecución.

Respecto a esto (Fernando Asteasuain, 2002) señala que el tejido estático (tejido de tiempo de vinculación) evita que el nivel de abstracción introducido por la POA derive en un impacto negativo en la eficiencia de la aplicación, ya que todo el trabajo se realiza en tiempo de compilación, y no existe sobrecarga en ejecución. Si bien esto es deseable el costo es una menor flexibilidad: los aspectos quedan fijos, no pueden ser modificados en tiempo de ejecución, ni existe la posibilidad de agregar o remover nuevos aspectos.

Y sobre el tejido dinámico señala que el tejido dinámico implica que el proceso de composición se realiza en tiempo de ejecución, decrementando la eficiencia de la aplicación. El postergar la composición brinda mayor flexibilidad y libertad al programador, ya que cuenta con la posibilidad de modificar un aspecto según información generada en ejecución, como también introducir o remover dinámicamente aspectos. La característica dinámica de los aspectos pone en riesgo la seguridad de la aplicación.

En (Sommerville, 2011), se afirma que el enfoque usado más comúnmente para el tejido de aspectos es el tejido de tiempo de vinculación, pues esto permite la implementación eficiente de los aspectos sin una gran carga de tiempo de ejecución. El tejido dinámico es el enfoque más flexible, pero puede incurrir en significativas penalizaciones de rendimiento durante la ejecución del programa. Actualmente el pre procesamiento de código fuente se usa poco.

2.2.2.5. Desarrollo orientado a aspectos

Existen diversos autores acerca de cómo realizar los pasos para el desarrollo orientado a aspectos:

a) Proceso de diseño orientado a aspectos

(Sommerville, 2011), sugiere un proceso de diseño orientado a aspectos que incluyan las siguientes actividades:

1) Diseño del sistema central:

En esta etapa se diseña la arquitectura del sistema para soportar la funcionalidad central del sistema. La arquitectura debe considerar también los requerimientos de calidad de servicio, como los requerimientos de rendimiento y confiabilidad.

2) Identificación y diseño de aspectos:

A partir de las extensiones identificadas en los requerimientos del sistema, habrá que efectuar un análisis para ver si son aspectos en sí mismos o si deben descomponerse en varios aspectos. Una vez identificados los aspectos, pueden diseñarse por separado tomando en cuenta el diseño de las características del sistema central.

3) Diseño de composición:

En esta etapa se analiza el sistema central y los diseños de aspectos para descubrir dónde deben combinarse los aspectos con el sistema central. En esencia, se identifican los puntos de enlace en un programa donde se tejerán los aspectos.

4) Análisis y resolución de conflictos:

Un problema con los aspectos es que pueden interferir entre sí cuando se combinan con el sistema central. Los conflictos ocurren al haber un choque de puntos de corte con diferentes aspectos que especifican que deben combinarse en el mismo punto del programa.

5) Diseño de nombre:

Ésta es una importante actividad de diseño que define estándares para nombrar las entidades del programa. Esto es importante para evitar el problema de puntos de corte accidentales. Éstos ocurren cuando, en algún punto de enlace del programa, el nombre coincide accidentalmente con el de un patrón de punto de corte.

En consecuencia, el consejo se aplica de manera no intencional a dicho punto. Desde luego, esto es indeseable y puede conducir a un comportamiento inesperado del programa. Por lo tanto, habrá que diseñar un esquema de nomenclatura que minimice la probabilidad de que esto ocurra.

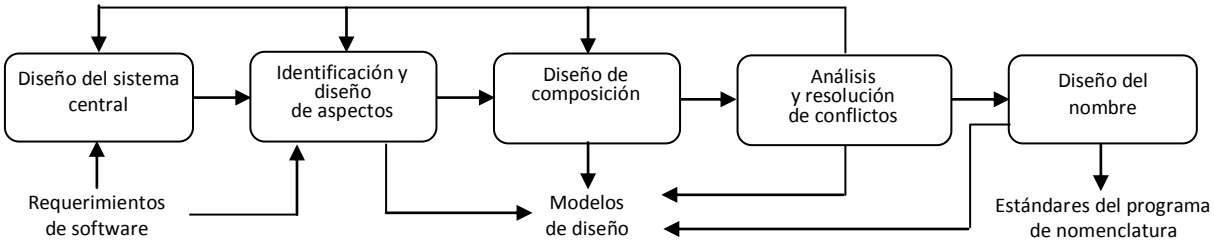


Ilustración 11: Proceso de diseño orientado a aspectos

Fuente: (Sommerville, 2011).

b) Desarrollo de software orientado a aspectos con casos de uso

(Jacobson, y otros, 2004) Proponen enfocarse en crear una mejor abstracción modular del sistema desde los inicios hasta el final del desarrollo del software usando aspectos en cada fase. Incluye las siguientes fases:

1) Captura de requisitos:

En esta fase se identifica tanto los requerimientos de aplicación y los requerimientos de infraestructura haciendo uso del modelado con casos de uso. De esta manera se logra identificar aspectos tempranos que conllevan a un tratamiento más temprano de incumbencias transversales. Como resultado se obtiene la lista de características del sistema que contiene requerimientos funcionales y requerimientos no funcionales

2) Análisis:

En esta fase se representan estructuras de elementos de análisis en forma de paquetes y clases (interfaz, controlador, entidad). Se conforman paquetes de denominación *use-case slice* y *non-uc-specific slice,* un *use-case slice* representa la realización del caso de uso en términos de requerimientos de aplicación y *non-uc-specific slice* contienen clases que se usan en muchos casos de uso.

3) Diseño:

En esta fase se incluyen actividades que refinan el análisis y lo realiza en términos de paquetes incluyendo en ellos las dos estructuras identificadas en las fases de requisitos y análisis. En esta etapa el diseño se enfoca hacia una plataforma de implementación, así mismo se identifican la arquitectura del sistema incluyendo aspectos y extensiones de clases. Por otro lado se selecciona el tipo de tejido a adoptar y se estructura una jerarquía de aspectos y/o mecanismos para evitar conflictos entre estos.

4) Implementación:

Se realiza la implementación de los requerimientos del sistema y requerimientos de dominio para finalmente realizar el tejido seleccionado en el diseño.

5) Pruebas

Las pruebas se realizan desde los requisitos hasta la codificación. Identificando si fuere el caso comportamientos no deseados en el sistema por la inclusión de aspectos.

c) Método de diseño orientado a aspectos:

(Keuler, 2011) Propone su *Connector design in the Fraunhofer Architecture Design process*:

1) Base connector modeling:

En la primera fase, un modelo *Base connector modeling* que se diseña de forma tal que refleje el comportamiento de núcleo interacción funcional del sistema. A esa fase podemos llamarle identificación de requerimientos funcionales.

2) Tactic selection:

Las tácticas usadas en el proceso de *connector design* son tácticas de ejecución. Las tácticas mismas podrían ser definidas en otros proyectos arquitectónicos, así, ya que no tienen que ver con un modelo arquitectónico particular, en absoluto. De manera similar a los patrones arquitectónicos, las tácticas pueden simplemente seleccionar en caso de que existan; de lo contrario, pueden ser adaptados o creados desde cero.

Se deben establecer **tácticas de detección** durante el tiempo de ejecución, dicho de otra manera la estructura de los componentes del sistema usualmente es impactado por la introducción de nuevos componentes o cambio de intercambio de información entre estas, estos deben ser capaces de identificar los incidentes en términos de diseño con el fin de reaccionar frente a ellos. Por otro lado se deben implementar **tácticas de recuperación** con el objetivo de reducir al mínimo las consecuencias negativas de un incidente que ya ha ocurrido. Además establecer **tácticas de prevención** que tienen por objeto reducir al mínimo la probabilidad de que un incidente en particular se producirá.

3) Táctica de tejido:

Refiere a que el tejido en sí requiere de un conjunto de medidas de pre-procesamiento que necesitan ser ejecutados de forma manual (join points). Los pasos de pre-procesamiento producen una llamada configuración de tejido que específica donde los componentes se unirán.

4) Análisis de impactos:

En otras palabras, tenemos que comprobar la idoneidad de las soluciones para los requisitos indicados y su compatibilidad con las decisiones de diseño realizados en las iteraciones anteriores. Por esta razón, el modelo resultante necesita ser comprobado para certificar sus propiedades. Las características del modelo que deben comprobarse dependen de la calidad que necesitamos alcanzar.

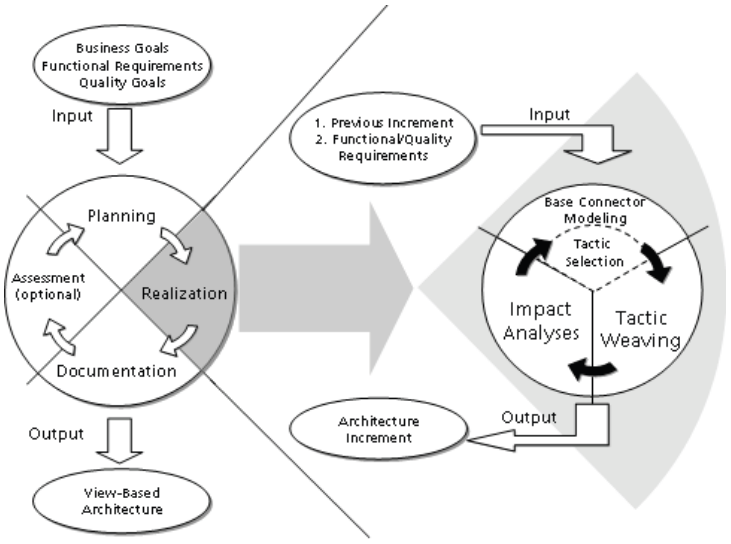


Ilustración 12: Connector design in the Fraunhofer Architecture Design process
Fuente: (Keuler, 2011).

d) Desarrollo de una aplicación basado en XP y POA

La Programación Extrema es una metodología ágil que promueve el desarrollo basado en principios simplicidad, comunicación, realimentación y coraje. Está diseñada para el uso con equipos pequeños que necesitan desarrollar software rápidamente en un ambiente de continuos cambios de requerimientos. Las cinco etapas de la XP[8] son (Kendall, 2005):

- Exploración
- Planeación
- Iteraciones a la primera versión
- Puesta en producción
- Mantenimiento

(Elhaibe, 2006) Establece que mientras XP es una metodología que afecta por completo el diseño y el desarrollo, POA es una técnica que apunta a la separación e implementación de funcionalidad entrecruzada.

Una de las principales ideas en el Corazón de XP es el Refactoring. Refactoring ofrece numerosos beneficios al desarrollo de software y por lo tanto es ampliamente soportado por XP. Sin embargo, refactoring es a menudo un proceso tedioso que implica muchos pasos repetitivos de cambio y testeo de código.

La programación orientada a aspectos, por otra parte, permite ampliar la arquitectura sin requerir demasiado refactoring. Por lo tanto, la adición del paradigma de la programación orientada a aspectos a la metodología XP puede proporcionar varias ventajas a los desarrolladores.

[8] Programación extrema

Con esto la simplicidad se mejora con respecto a la estructura del código de la aplicación. La funcionalidad entrecruzada esta ordenada dentro de Aspectos, los cuales hacen al código más simple de entender. Así mismo la POA puede mejorar la comunicación y el entendimiento del código entre los desarrolladores. En cuanto al efecto de la POA se considera: Diseño Simple, Propiedad Colectiva, Refactoring, Testing, Integración Continua y Estándares de Programación.

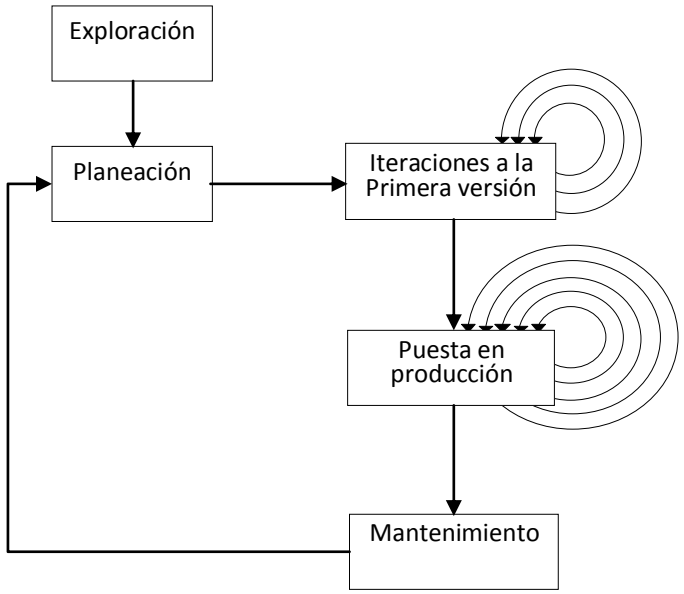


Ilustración 13: Las cinco etapas del proceso de desarrollo de XP muestran que las iteraciones frecuentes son esenciales para desarrollar con éxito un sistema.

Fuente: (Kendall, 2005)

e) Desarrollo basado en aspectos

(Laddad, 2002) Nos dice que el desarrollo de una aplicación basada en aspectos requiere de tres pasos:

1) Descomposición:

Descomponer los requisitos para identificar las incumbencias transversales y componentes. Se separan las preocupaciones a nivel de módulo de cortes transversales preocupaciones a nivel de sistema. Descomponer los requerimientos para distinguir aquellos que son componentes de los que son aspectos.

2) Implementación de incumbencias:

Implementar cada incumbencia por separado (aspectos y componentes)

3) Recomposición aspectual:

El proceso de recomposición, también conocido como el tejido o la integración, utiliza ésta información para componer el sistema final. Definir las reglas que permitan combinar los aspectos con los componentes.

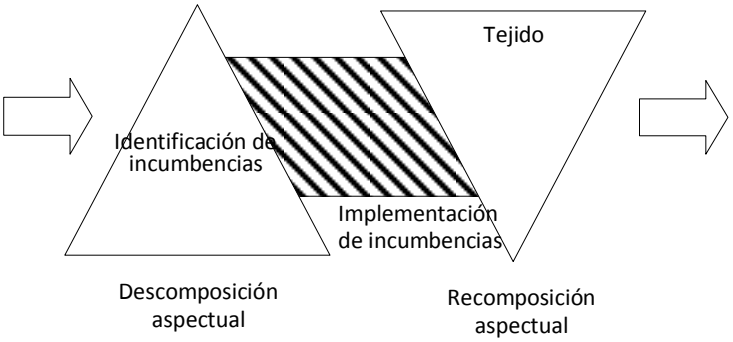


Ilustración 14: Etapas del desarrollo de software orientado a aspectos.
Fuente: (Laddad, 2002)

2.2.2.6. Beneficios e inconvenientes de la programación orientada a aspectos

La ingeniería de software como cualquier disciplina de la ingeniería está sujeta a costos y beneficios, la programación orientada a aspectos no está libre, los críticos de la POA a menudo hablan de lo difícil que es entender, y de hecho, la POA necesita tiempo, paciencia y práctica, POA exige que se piense en el diseño e implementación del sistema de una manera nueva.

a) Beneficios:

Algunas de las principales ventajas del paradigma de Programación Orientado a Aspectos son (Laddad, 2002):

- Ayuda a superar los problemas de codificación como son código Mezclado y código diseminado
- Implementación modularizada, logra separar cada concepto con mínimo acoplamiento
- Mayor posibilidad de evolución, La separación de conceptos permite agregar nuevos aspectos, modificar y/o remover aspectos existentes fácilmente.
- Mayor reusabilidad: Al ser implementados separadamente, tienen mayores probabilidades de ser reusados en otros sistemas con requerimientos similares
- Permite N-dimensiones
- Principio de mínimo acoplamiento y máxima cohesión

La implementación de cada interés en forma independiente, facilita el mantenimiento del sistema y la introducción de modificaciones o nuevas funcionalidades.

b) Inconvenientes:

- Para los sistemas orientados a aspectos, los procesos de las pruebas de validación no son diferentes a los de cualquier otro sistema. El programa final ejecutable se trata

como una **caja negra**[9] y se diseñan pruebas para demostrar si el sistema cumple o no con los requerimientos. Sin embargo, el uso de aspectos causa problemas reales con las inspecciones de programa y las pruebas de **caja blanca**[10], mientras que el código fuente del programa se usa para identificar potenciales pruebas de defecto. (Sommerville, 2011)

- En un sistema orientado a aspectos, es imposible la lectura de código secuencial. El lector debe examinar cada aspecto, comprender sus puntos de corte (que pueden ser patrones) y el modelo de punto de enlace del lenguaje orientado a aspectos.

(Cugola, y otros, 1999) Refieren:

- Posibles conflictos entre código funcional (código base) y otros aspectos. En general tales enfrentamientos resultan de la necesidad de romper la encapsulación de funcional unidades para implementar un aspecto diferente.
- Posibles enfrentamientos entre Aspectos. Los dos aspectos funcionan perfectamente cuando se ejecutan individualmente, pero al aplicarlos en conjunto resulta en un comportamiento fallido.
- Posibles choques entre el código de aspecto y mecanismos del lenguaje de especificaciones. Uno de los ejemplos más conocidos de problemas que cae dentro de esta categoría es la **Anomalía de herencia**[11].

(laddad, 2009) Refiere en el ámbito del proyecto y se detalla algunos inconvenientes:

- Hacer una inversión en el aprendizaje de POA
- La contratación de programadores expertos
- Modificación de la construcción y otros procesos de desarrollo
- Tratar con la disponibilidad de herramientas de ayuda

Acerca de la abstracción:

- La programación orientada a aspectos abstrae el flujo del programa aún más.
- Se requiere técnicas de descomposición de incumbencias transversales.

[9] Se dispone del Ejecutable, el DLL, el JAR, el Lib, el Bin, o cualquier archivo ejecutable (que no sea código fuente) al cual se le hacen las validaciones y verificaciones.

[10] Se dispone del código fuente para hacer verificaciones y validaciones conforme a los requerimientos.

[11] Forzar a la redefinición de los métodos heredados para mantener la integridad del sistema concurrente

2.2.2.7. Lenguajes orientados a aspectos

Un lenguaje de aspectos puede tener dos enfoques diferentes: Lenguajes de propósito específico y lenguaje de propósito general, los cuales brindan mecanismos para capturar y declarar aspectos (Fernando Asteasuain, 2002), (Joskowicz, 2008):

.

Los lenguajes de aspectos de dominio específico son capaces de manejar uno o más aspectos, pero no pueden manejar otros aspectos más allá de los aspectos para los cuales fueron diseñados. Tienen un mayor nivel de abstracción que el lenguaje base, y expresan los conceptos de dominio específico en un nivel de representación más alto.

Los lenguajes de aspectos de propósito general son diseñados para describir cualquier clase de aspecto, no solo específicos. El nivel de abstracción del lenguaje base y del lenguaje de aspectos de propósito general es el mismo. Proveen un ambiente más familiar para el desenvolvimiento, una vez que utiliza, generalmente los mismos ambientes de desarrollo también son usados por el lenguaje base.

Para (Quintero, 2000), si contrastamos estos dos enfoques, propósito general versus propósito específico, se tiene que los lenguajes de aspectos de propósito general no pueden cubrir completamente las necesidades. Tienen un severo inconveniente: Permiten la separación del código, pero no garantizan la separación de funcionalidades, es decir, que la unidad de aspecto solamente se utiliza para programar el aspecto. Sin embargo, esta es la idea central de la programación orientada a aspectos. En comparación con los lenguajes de aspectos de propósito general, los lenguajes de aspectos de dominio específico fuerzan la separación de funcionalidades.

Con los lenguajes específicos, el pasar de una aplicación a otra, significa comenzar de cero nuevamente, otros lenguajes, otras restricciones, que quizás sean similares, pero quizás sean completamente distintas. Si es el caso que se desee utilizar aspectos para una única aplicación, por una única vez, entonces un lenguaje de aspectos de dominio específico hará un mejor papel, con una separación de conceptos más clara.

2.2.2.8. Tecnología de soporte para la programación orientada a aspectos

a) COOL:

COOL es un lenguaje de dominio específico cuya finalidad es la sincronización de hilos concurrentes. El lenguaje base que utiliza es una versión restringida de Java, ya que se han de eliminar los métodos *wait*, *notify* y *notifyAll*, y la palabra clave *synchronized* para evitar que se produzcan situaciones de duplicidad al intentar sincronizar los hilos en el aspecto y en la clase.

Se identifica dos componentes principales *el coordinador* y *el objeto*, Los coordinadores no se pueden instanciar directamente, sino que se asocian con las instancias de las clases a las que coordinan en tiempo de instanciación, así mismo pueden los coordinadores tienen acceso total para visualizar todas las variables, métodos sean privados, protegidos o públicos de la clase.

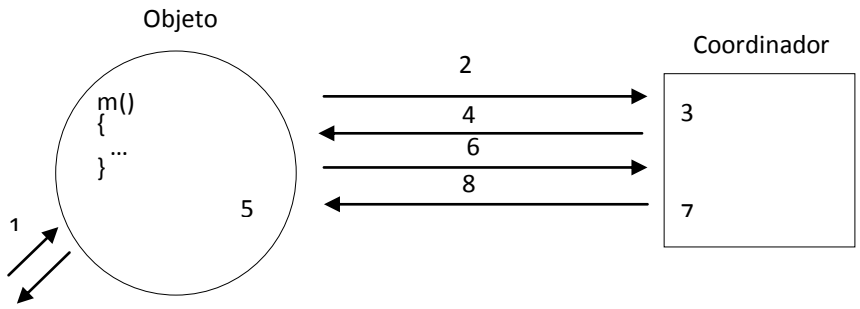


Ilustración 15: Protocolo entre un objeto y su coordinador en COOL

Fuente: (Steinmacher, 2003).

1. En el hilo H se invoca al método m del objeto (obj.m())
2. La petición se envía al coordinador del objeto.
3. El coordinador comprueba las restricciones de exclusión y las precondiciones para el método m. Si se da alguna restricción o no se cumple una precondición, se suspende H. Cuando todo se cumpla, H puede ejecutar el método m. Justo antes de esto, el coordinador ejecuta el bloque on_entry asociado con el método m.
4. Se envía la petición al objeto.
5. El hilo H ejecuta el método m en el objeto.
6. Se le envía el retorno del método al coordinador.
7. El coordinador ejecuta su bloque on_exit para el método m.
8. Finalmente se retorna de la invocación de m.

b) RIDL (Remote Interaction and Data transfers aspect Language)

Un programa RIDL consiste de un conjunto de módulos de "portales". Un portal es el encargado de manejar la interacción remota y la transferencia de datos de la clase asociada a él, y puede asociarse como máximo a una clase (Joskowicz, 2008).

Es un lenguaje de dominio específico que se basa en mecanismos para la manipulación de datos entre diferentes espacios de ejecución separadamente de los componentes funcionales de la aplicación.

Cuenta con un conjunto de módulos llamados portal. Un portal es encargado de manejar la interacción remota y transferencia de datos entre la clase asociada y ella. Las declaraciones

contenidas en un portal identifican clases que pueden ser referenciadas en llamadas remotas, además de identificar los métodos de una clase que serán exportados. Un portal es automáticamente asociado a una instancia de clase cuando la referencia a este objeto fuera exportada fuera del espacio de ejecución en que el objeto fue creado.

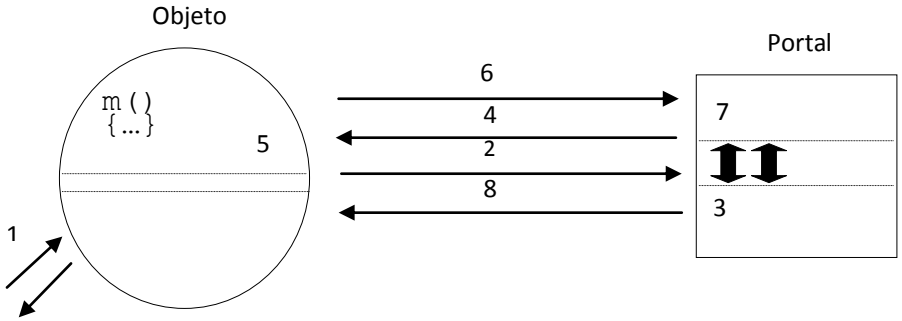


Ilustración 16: Protocolo de un portal RIDL

Fuente: (Steinmacher, 2003).

1. De algún espacio remoto ocurre una invocación al método m del objeto
2. La solicitud es enviada para el portal del objeto
3. El método es procesado de acuerdo su declaración en el portal, donde los parámetros son pasados del espacio remoto para el espacio local de acuerdo con la especificación del portal.
4. La solicitud es enviada al objeto
5. El método m es ejecutado
6. Cuando la invocación retorna, el retorno es presentado al portal del objeto.
7. El resultado es procesado de acuerdo con su declaración de operación remota.
8. La invocación del método retorna y el resultado es pasado al espacio remoto.

c) AspectJ

(laddad, 2009), AspectJ es una extensión a Java orientada a aspectos y de propósito general. AspectJ extiende Java con una nueva clase de módulos llamado aspecto. Los aspectos cortan las clases, las interfaces y a otros aspectos. Los aspectos mejoran la separación de competencias haciendo posible localizar de forma limpia los conceptos de diseño del corte.

El tejedor (Weaver) de aspectos es implementado como un pre-procesador que recibe el programa de componentes y el programa de aspectos como entrada, generando archivos java que pueden ser compilados normalmente.

En (laddad, 2009) se explica algunas aplicaciones:

- Aplicaciones Enterprise

Las aplicaciones empresariales necesitan hacer frente a muchas funcionalidades transversales: la gestión de transacción, seguridad, auditoría, acuerdo de nivel de servicio, el seguimiento, el control de concurrencia, la mejora de la disponibilidad de aplicaciones, gestión de errores, y así sucesivamente.

- Web y Servidor de aplicaciones

Apoya el desarrollo de aplicaciones empresariales basada en OSGi[12]. Se utilizan los aspectos para implementar varias funcionalidades transversales, como First Failure Data Capture (**FFDC**[13]), el seguimiento y la aplicación de políticas.

- Marcos de aplicación

Marcos de aplicaciones pueden utilizar POA efectivamente tratar funcionalidades transversales específicos manteniendo su estructura modularizada. Como se señaló anteriormente, Spring ofrece transversales soluciones como la gestión de transacciones y la seguridad a través de aspectos.

d) Spring AOP

Spring es un marco de trabajo ligero y más popular para aplicaciones empresariales. Para cumplir las necesidades de las aplicaciones de la empresa, incluye un sistema de POA sobre la base de interceptores y el patrón proxy. Implementaciones anteriores ofrecen un modelo de programación un tanto compleja. El nuevo modelo de programación, está basado en AspectJ, por lo tanto ofrece una experiencia mejor de programación y permite escribir aspectos personalizados sin dificultad. (laddad, 2009)

- DAO: Accesos JDBC con manejo de transacciones (desde el módulo AOP).
- ORM: Integración con Hibernate, JDO.
- JEE: Acceso e interacción con servicios Enterprise.
- Web: Provee un contexto apropiado para el desarrollo de aplicaciones web e integración con otros frameworks.
- POJO: (Plain Old Java Object) Revalora la simplicidad de las clases Java aportando manejo de transacciones de forma no intrusiva.
- XML: Configuración basada en archivos XML.
- Seguridad: Como un requerimiento no funcional implementado como un aspecto
- Remoting: RMI simplificado, acceso y publicación de web services.

---

[12] OSGI son las siglas de Open Services Gateway Initiative, Su objetivo es definir las especificaciones abiertas de software que permita diseñar plataformas compatibles que puedan proporcionar múltiples servicios.

[13] Es una tecnología que se utiliza para capturar el diagnostico de información capturada cuando se producen errores. Con la finalidad de reducir la necesidad de reproducir errores.

- Testing: Provee un paquete de pruebas específicas para componentes del framework e integrado con JUnit.
- Programación Orientada a Aspectos (AOP): Interceptando métodos con aspectos
- Hibernate: Gestión integrada de transacciones y recursos.
- Web Services: Configuración y consumo de un web service.
- Seguridad: Control y gestión de seguridad con Acegi.

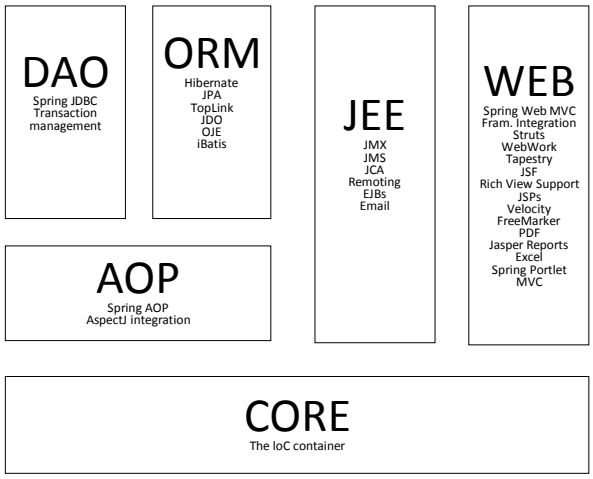


Ilustración 17: Spring AOP

Fuente: J2eebrain, http://www.j2eebrain.com/java-J2ee-spring-framework.html

e) AspectC

AspectC es un lenguaje de propósito general que extiende C. Es similar a AspectJ, pero sin soporte para la programación orientada a objetos.

El código de aspectos interactúa con la funcionalidad básica en los límites de una llamada a una función, y puede ejecutarse antes, después, o durante dicha llamada.

Como el lenguaje C es estático, el tejedor de AspectC es también estático. Dado que C es utilizado en un gran número de aplicaciones, es muy interesante poder disponer de un lenguaje orientado a aspectos que lo tenga como lenguaje base. Su tejido es estático. AspectC es ideal para el desenvolvimiento de sistemas donde la eficiencia es primordial, como en La implementación de sistemas operativos.

f) AspectC++

AspectC++ (Xerox, 2014) es un lenguaje orientado a aspectos de propósito general que extiende el lenguaje C++ para soportar el manejo de aspectos. Sintácticamente, un aspecto en este lenguaje es muy similar a una clase en C++. Sin embargo, además de funciones, un aspecto puede definir “avisos” (“advice”). Luego de la palabra clave “advice”, una expresión de corte (“pointcut expresion”) define el punto donde el aspecto modificará al programa (es decir, los “puntos de enlace” o “join points”).

Pueden utilizarse expresiones de corte para identificar un conjunto de puntos de enlaces. Se componen a partir de expresiones de corte y un conjunto de operadores algebraicos. La declaración de los avisos es utilizada para especificar código que debe ejecutarse en los puntos de enlace determinados por la expresión de corte. Los avisos pueden insertarse antes, después o durante la ejecución de los métodos donde se insertan.

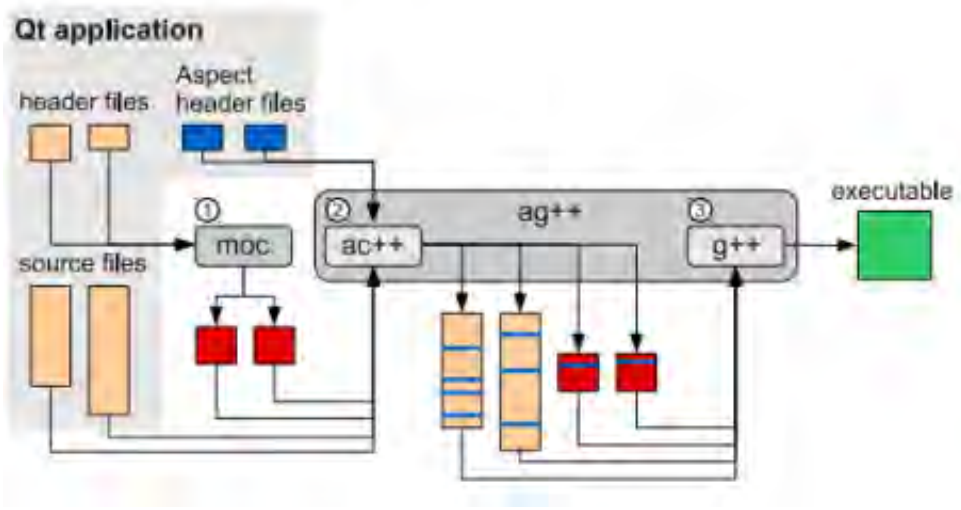


Ilustración 18: Arquitectura de compilación con AspectC++

Fuente: (Spinczyk, y otros, 2011)

1. El meta objeto **moc** lee los archivos de cabecera de la aplicación Qt[14] y genera un archivo de en código de C++ que implementan los métodos meta-objeto para todas las clases Qt.
2. El compilador ac++ lee las cabeceras de aspectos, preprocesa el código fuente C++ y determina donde los aspectos tendrán efecto. Seguidamente los aspectos se tejen en el código, generando un nuevo archivo.
3. El compilador C++ compila y enlaza los archivos tejidos y crea un archivo ejecutable.

2.2.3. Diseño del software

2.2.3.1. Definición

El diseño crea una representación o modelo del software, pero a diferencia del análisis (que se enfoca en la descripción de los datos, las funciones y el comportamiento requerido), el modelo de diseño proporciona detalles acerca de las estructuras de datos, las arquitecturas, las interfaces, y los componentes del software que son necesarios para implementar el sistema. El diseño permite al ingeniero de software modelar el sistema o producto en relación con su

[14] Qt es una biblioteca multiplataforma para desarrollar aplicaciones con interfaz gráfica o sin interfaz gráfica como herramientas para la línea de comandos y consolas para servidores.

calidad y mejorarse antes de generar código, de realizar pruebas y de que los usuarios finales se vean involucrados a gran escala. El diseño es el sitio donde se establece la calidad del software. (Pressman, 2009).

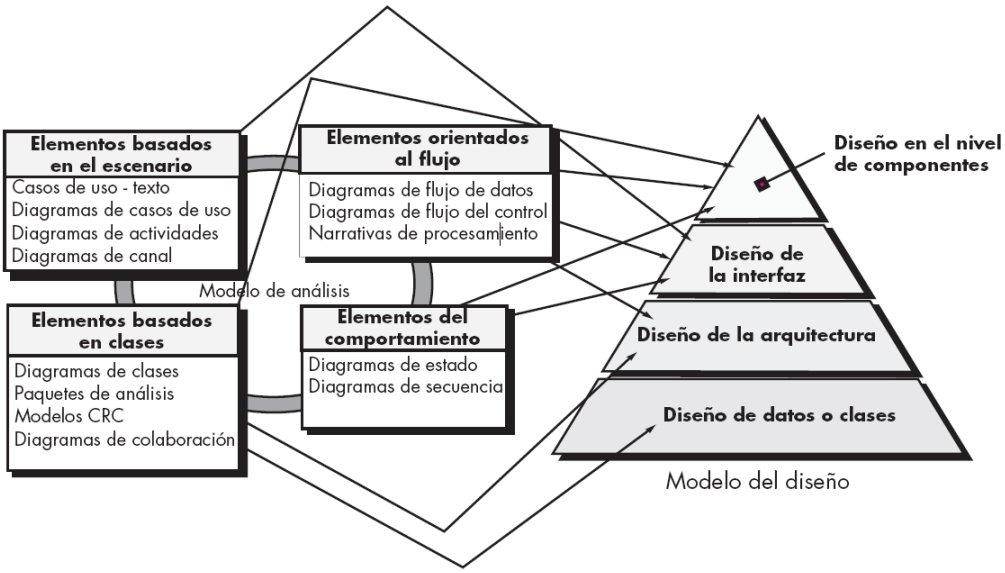


Ilustración 19: Transformación del modelo de análisis al modelo de diseño

Fuente: (Pressman, 2009).

El **diseño arquitectónico** define la relación entre los elementos estructurales más importantes del software, los estilos arquitectónicos y patrones de diseño que pueden usarse para satisfacer los requisitos definidos por el sistema, y las restricciones que afectan la manera en que se pueden implementar los patrones arquitectónicos. La representación del diseño arquitectónico puede derivarse de la especificación del sistema del modelo de análisis y de la interacción de los subsistemas definidos dentro del modelo del análisis. (Pressman, 2009)

El diseño de la interfaz describe la forma en que el software se comunica con los sistemas que interactúan con él y con los humanos que los utilizan. Una interfaz implica un flujo de información (por ejemplo, datos o control) y un tipo de comportamiento específico. Por lo tanto, los escenarios de uso y los modelos de comportamiento proporcionan mucha de la información que se requiere en el diseño de la interfaz.

2.2.3.2. Importancia del diseño

La importancia del diseño del software puede describirse con una sola palabra: Calidad. El diseño es la etapa en la que se fomentará la calidad en la ingeniería del software. El diseño proporciona las representaciones del software susceptibles de evaluar de la calidad. El diseño es la única forma en que, de manera exacta, un requisito del cliente se puede convertir en un sistema o producto de software terminado. El diseño del software sirve como fundamento para todas las actividades subsecuentes de la ingeniería del software y del soporte de éste. Sin

diseño se corre el riesgo de construir un sistema inestable, el cual fallará cuando se realicen cambios pequeños que será difícil de probar cuya finalidad no podrá evaluarse sino hasta etapas tardías del proceso del software, cuando queda poco tiempo y ya se ha gastado mucho tiempo en él. (Pressman, 2009). El Diseño es la única manera de materializar con precisión los requerimientos del cliente.

2.2.3.3. Proceso y calidad del diseño

(Pressman, 2009) Cita las tres características que sirven como guía en la evaluación de un buen diseño:

- El diseño debe implementar todos los requisitos explícitos contenidos en el modelo de análisis, y debe ajustarse a todos los requisitos implícitos que desea el cliente
- El diseño debe de ser una guía legible y comprensible para quienes generan código y quienes realizan pruebas y, en consecuencia dan soporte al software.
- El diseño debe proporcionar una imagen completa del software, dando dirección a los dominios de datos, funcionales y de comportamiento, desde una perspectiva de implementación.

Criterios técnicos para un buen diseño:

- Un diseño debe de ser modular, y a su vez dividirse en subsistemas con características funcionales independientes.
- Debe presentar una estructura arquitectónica que se haya creado mediante patrones de diseño reconocibles, la integren componentes que exhiban buenas características de diseño, y puedan implementarse de manera evolutiva.
- Un diseño debe conducir a interfaces que reduzcan la complejidad de las conexiones éntrelos componentes y el ambiente externo.

2.2.3.4. Atributos del diseño de software

Los siguientes términos definen un marco de trabajo relevante para realizar el diseño de software de manera correcta, En (Pressman, 2009) se indican los siguientes:

a) Abstracción

Refiere a una solución modular donde a cualquier problema se pueden exponer muchos grados de abstracción. Unas variables de grado de abstracción son las abstracciones procedimentales y de datos.

1) Las abstracciones procedimentales

Se refiere a una secuencia de instrucciones que tiene una función específica y limitada. El nombre de la abstracción procedimental implica estas funciones pero se omiten detalles específicos.

2) La abstracción de datos

Es una colección nombrada de datos que describe un objeto de datos. Se puede decir que la abstracción procedimental emplea información de la abstracción de datos.

b) Arquitectura

La arquitectura del software alude a "la estructura general del software y las formas en que la estructura proporciona una integridad conceptual para un sistema". En su forma más simple, la arquitectura es la estructura y organización de los componentes del programa, la manera en que estos componentes interactúan y la estructura de datos que utilizan los componentes.

c) Patrones

Los patrones son una solución general frente a problemas en un contexto específico.

d) Modularidad

La modularidad tiene que ver con que el software se divide en componentes con nombres independientes y que es posible abordar en forma individual. Está ligado a la complejidad, como refiere (Bruegge, y otros, 2002) para reducir la complejidad del dominio de aplicación identificamos partes más pequeñas, llamadas clases, y las organizamos en paquetes. En forma similar, para reducir la complejidad del dominio de solución descomponemos un sistema en partes más simples, llamadas subsistemas, que están compuestas de varias clases del dominio de solución.

e) Ocultación de información

Los módulos deben especificarse y diseñarse de manera que la información que está dentro del módulo sea inaccesible para otros módulos que no necesitan esa información. Con esto se proporciona mayores beneficios cuando se requieren modificaciones en las pruebas y beneficios en el mantenimiento.

f) Independencia funcional

El concepto de independencia funcional es la suma directa de la modularidad, de los conceptos de abstracción y ocultación de información. Se consigue cuando se realice una subfunción específica, está relacionado a la cohesión y acoplamiento.

g) Refinamiento

El refinamiento y la abstracción son conceptos complementarios, mientras que la abstracción define procedimientos y datos sin considerar detalles, el refinamiento ayuda a revelar los detalles de implementación.

h) Re fabricación

Una técnica de reorganización que simplifica el diseño (o código) de un componente sin cambiar su comportamiento.

2.2.3.5. La arquitectura de software

En (Sommerville, 2011) la arquitectura de software es importante porque afecta el desempeño y la potencia, así como la capacidad de distribución y mantenimiento de un sistema (Bosch, 2000). Como afirma Bosch, **los componentes individuales implementan los requerimientos funcionales del sistema.** Los requerimientos no funcionales dependen de la arquitectura del sistema, es decir, la forma en que dichos componentes se organizan y comunican. En muchos sistemas, los requerimientos no funcionales están también influidos por componentes individuales, pero no hay duda de que la arquitectura del sistema es la influencia dominante. Las arquitecturas del software según Somerville, se dividen en Arquitectura en pequeño y arquitectura en grande.

Las arquitecturas de software se diseñan en dos niveles de abstracción, arquitectura en pequeño y arquitectura en grande:

1. La arquitectura en pequeño se interesa por la arquitectura de programas individuales. En este nivel, uno se preocupa por la forma en que el programa individual se separa en componentes.

2. La arquitectura en grande se interesa por la arquitectura de sistemas empresariales complejos que incluyen otros sistemas, programas y componentes de programa. Tales sistemas empresariales se distribuyen a través de diferentes computadoras, que diferentes compañías administran y poseen.

2.2.3.6. Diseño arquitectónico

(Sommerville, 2011) Menciona que el diseño arquitectónico se interesa por entender cómo debe organizarse un sistema y cómo tiene que diseñarse la estructura global de ese sistema. Es el enlace crucial entre el diseño y la ingeniería de requerimientos, ya que identifica los principales componentes estructurales en un sistema y la relación entre ellos. La salida del proceso de diseño arquitectónico consiste en un modelo arquitectónico que describe la forma en que se organiza el sistema como un conjunto de componentes en comunicación.

La elección de arquitectura de distribución es una decisión clave que afecta el rendimiento y la fiabilidad del sistema. Debido a la estrecha relación entre los requerimientos no funcionales y la arquitectura de software, el estilo y las estructuras arquitectónicas

particulares que se elijan para un sistema dependerán de los requerimientos de sistema no funcionales

Sommerville cita cinco factores sobre las decisiones en el diseño arquitectónico:

1. Rendimiento: Si el rendimiento es un requerimiento crítico, la arquitectura debe diseñarse para localizar operaciones críticas dentro de un pequeño número de componentes, con todos estos componentes desplegados en la misma computadora en vez de distribuirlos por la red.
2. Seguridad: Si la seguridad es un requerimiento crítico, será necesario usar una estructura en capas para la arquitectura, con los activos más críticos protegidos en las capas más internas, y con un alto nivel de validación de seguridad aplicado a dichas capas.
3. Protección: Si la protección es un requerimiento crítico, la arquitectura debe diseñarse de modo que las operaciones relacionadas con la protección se ubiquen en algún componente individual o en un pequeño número de componentes. Esto reduce los costos y problemas de validación de la protección, y hace posible ofrecer sistemas de protección relacionados que, en caso de falla, desactiven con seguridad el sistema.
4. Disponibilidad: Si la disponibilidad es un requerimiento crítico, la arquitectura tiene que diseñarse para incluir componentes redundantes de manera que sea posible sustituir y actualizar componentes sin detener el sistema.
5. Mantenibilidad: Si la mantenibilidad es un requerimiento crítico, la arquitectura del sistema debe diseñarse usando componentes auto contenidos de grano fino que puedan cambiarse con facilidad.

   Los productores de datos tienen que separarse de los consumidores y hay que evitar compartir las estructuras de datos.

2.2.3.7. Vistas arquitectónicas

(Kruchten, 1995) habla sobre los conceptos de vista y puntos de vista, pues una vista no es más que una representación de todo el sistema software desde una determinada perspectiva, y un punto de vista se define como un conjunto de reglas (o normas) para realizar y entender las vistas.

El autor propone 4 vistas bien diferenciadas y se han de relacionar entre sí con una vista más, que es la denominada vista "+1". Estas 4 vista las denominó Kruchten como:

- Vista Lógica: En esta vista se representa la funcionalidad que el sistema proporcionara a los usuarios finales. Es decir, se ha de representar lo que el sistema debe hacer, y las funciones y servicios que ofrece y que indique las abstracciones clave en el sistema

como objetos o clases de objeto. Para completar la documentación de esta vista se pueden incluir los diagramas de clases, de comunicación o de secuencia de UML.

- Vista de Despliegue: En esta vista se muestra el sistema desde la perspectiva de un programador y se ocupa de la gestión del software; o en otras palabras, se va a mostrar cómo está dividido el sistema software en componentes y las dependencias que hay entre esos componentes. Para completar la documentación de esta vista se pueden incluir los diagramas de componentes y de paquetes de UML.
- Vista de Procesos: Que muestre cómo, en el tiempo de operación, el sistema está compuesto de procesos en interacción. Esta vista es útil para hacer juicios acerca de las características no funcionales del sistema, como el rendimiento y la disponibilidad. Para completar la documentación de esta vista se puede incluir el diagrama de actividad de UML.
- Vista Física: En esta vista se muestra desde la perspectiva de un ingeniero de sistemas todos los componentes físicos del sistema así como las conexiones físicas entre esos componentes que conforman la solución (incluyendo los servicios). Para completar la documentación de esta vista se puede incluir el diagrama de despliegue de UML.
- "+1" Vista de Escenarios: Esta vista va a ser representada por los casos de uso software y va a tener la función de unir y relacionar las otras 4 vistas, esto quiere decir que desde un caso de uso podemos ver cómo se van ligando las otras 4 vistas, con lo que tendremos una trazabilidad de componentes, clases, equipos, paquetes, etc., para realizar cada caso de uso. Para completar la documentación de esta vista se pueden incluir los diagramas de casos de uso de UML.

2.2.3.8. Estilos y patrones arquitectónicos

En (Pressman, 2009), un estilo arquitectónico es una transformación impuesta al diseño de todo un sistema. El objetivo es establecer una estructura para todos los componentes del sistema. En caso de que una arquitectura existente se vaya a someter a reingeniería, la imposición de un estilo arquitectónico desembocará en cambios fundamentales en la estructura del software, incluida una reasignación de la funcionalidad de los componentes.

Un patrón arquitectónico, al igual que un estilo, impone una transformación en el diseño de una arquitectura. Sin embargo, un patrón difiere de un estilo en varios elementos fundamentales:

1. El alcance de un patrón es menor ya que se concentra en un aspecto en lugar de hacerlo en toda la arquitectura.

2. Un patrón impone una regla sobre la arquitectura, pues describe la manera en que el software manejará algún aspecto de su funcionalidad al nivel de la infraestructura (ejemplo: concurrencia).
3. Los patrones arquitectónicos tienden a abarcar aspectos específicos del comportamiento dentro del contexto de la arquitectura (por ejemplo, como maneja una aplicación en tiempo real la sincronización o las interrupciones). Los patrones se usan junto con un estilo arquitectónico para determinar la forma de la estructura general de un sistema.

En términos más simples el estilo arquitectónico describe la estructura general del sistema y define los componentes del sistema, su relación e interactividad.

A diferencia del patrón arquitectónico que define la estructura básica del sistema, representa una plantilla de construcción que provee un conjunto de subsistemas.

2.2.3.9. Estilos arquitectónicos

En (Pressman, 2009) se clasifican estilos arquitectónicos como sigue:

a) Arquitectura centrada en datos:

Un almacén de datos se encuentra en el centro de esta arquitectura; otros componentes tienen acceso a él y cuenta con la opción de actualizar, agregar, eliminar o modificar los datos de ese almacén. Con esto se promueve la capacidad de integración.

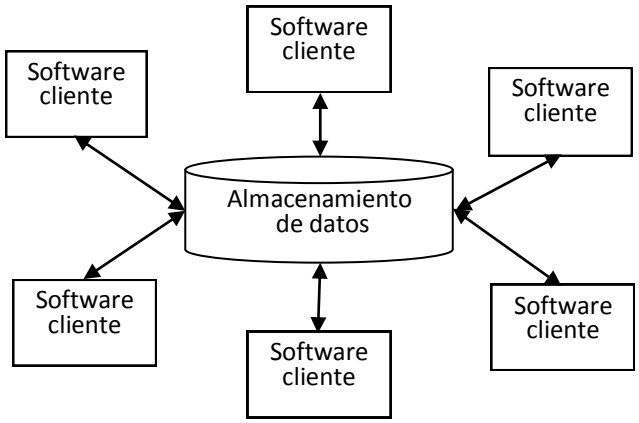


Ilustración 20: Arquitectura centrada en datos

Fuente: (Pressman, 2009).

Para (Zhu, 1995), Tienen la ventaja de que los clientes son independientes entre sí, y el almacén de datos es independiente de los clientes. Así, el sistema es escalable, ya que nuevos clientes pueden agregarse fácilmente. También puede modificar con respecto a los cambios en la funcionalidad de cualquier cliente, ya otros clientes no se verán afectados. Sin embargo, la modificabilidad de la representación de los datos almacenados en el componente de datos compartido podría ser problemático ya que puede afectar a todos los clientes en el sistema.

b) Arquitectura de flujo de datos

Esta arquitectura se aplica cuando los datos de entrada se habrán de transformar en datos de salida mediante una serie de componentes para el cálculo o la manipulación. Una estructura de tuberías y filtros, tiene un conjunto de componentes denominados filtros, conectados por tuberías que transmiten datos de un componente al siguiente. Cada filtro funciona sin tomar en cuenta al resto de los demás componentes.

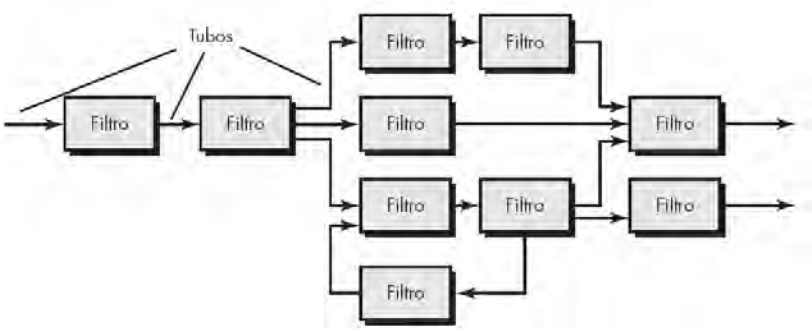


Ilustración 21: Arquitectura de flujo de datos

Fuente: (Pressman, 2009).

Para (Sommerville, 2011), se utiliza en aplicaciones de procesamiento de datos (tanto basadas en lotes [batch] como en transacciones), donde las entradas se procesan en etapas separadas para generar salidas relacionadas. Tiene como ventaja que es fácil de entender y soporta reutilización de transformación. El estilo del flujo de trabajo coincide con la estructura de muchos procesos empresariales. La evolución al agregar transformaciones es directa, puede implementarse como un sistema secuencial o como uno concurrente. La desventaja es que el formato para la transferencia de datos debe acordarse entre las transformaciones que se comunican. Cada transformación debe analizar sus entradas y sintetizar sus salidas al formato acordado. Esto aumenta la carga del sistema, y puede significar que sea imposible reutilizar transformaciones funcionales que usen estructuras de datos incompatibles.

c) Arquitectura de llamada y retorno

Permite que se obtenga una estructura de programa que resulta relativamente fácil de modificar y cambiar de tamaño. Se basan en la bien conocida abstracción de métodos. Existen dos subestilos:

- Arquitectura de programa principal/subprograma, se invoca desde un programa principal a varios componentes del programa que a su vez pueden invocar a otros componentes.

- Arquitectura de llamada a un procedimiento remoto: Los programas principal/subprograma se distribuyen en varias computadoras en red.

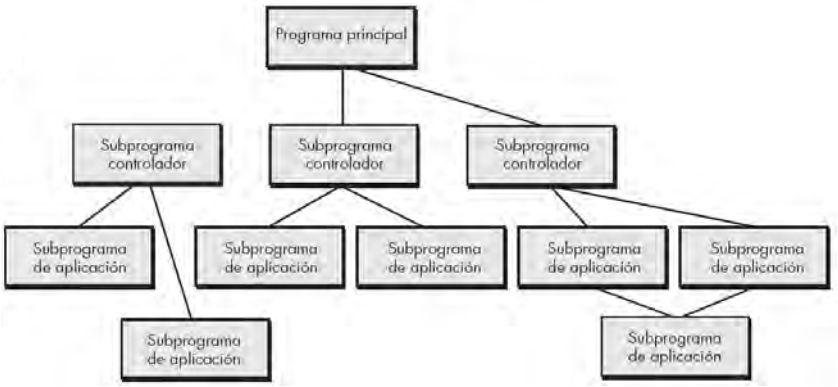


Ilustración 22: Arquitectura de llamada y retorno.

Fuente: (Pressman, 2009).

Para (Zhu, 1995), la descomposición en módulos disminuye la complejidad. Se persiguen escalabilidad y modificabilidad. Como desventaja está la dependencia y acoplamiento entre módulos, la reutilización y el mantenimiento son difíciles

d) Arquitectura orientada a objetos

Los componentes de un sistema se encapsulan los datos y las operaciones deben aplicarse para manipular los datos. La comunicación y coordinación entre componentes se consigue mediante el paso de mensajes

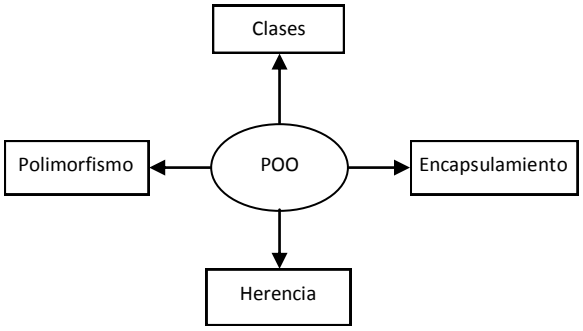


Ilustración 23: Arquitectura orientada a objetos

Fuente: Elaboración propia

La representación se oculta a otros objetos, como un objeto oculta su representación a sus los demás objetos, es posible cambiar su implementación sin modificar a los demás.

e) Arquitectura estratificada

En este estilo arquitectónico se divide en varias capas definidas; cada una de ellas realiza operaciones que acercan progresivamente al conjunto de instrucciones de la máquina.

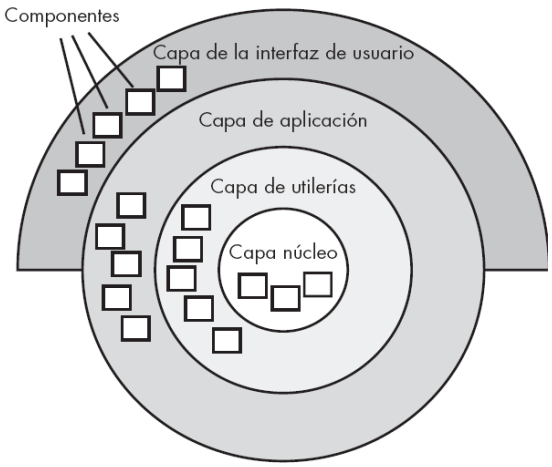


Ilustración 24: Arquitectura estratificada

Fuente: (Pressman, 2009).

Para (Sommerville, 2011), este patrón se usa cuando se tiene un sistema donde los grandes volúmenes de información generados deban almacenarse durante mucho tiempo. También puede usarse en sistemas dirigidos por datos, en los que la inclusión de datos en el repositorio active una acción o herramienta. Tiene la ventaja de que los componentes pueden ser independientes, no necesitan conocer la existencia de otros componentes. Los cambios hechos por un componente se pueden propagar hacia todos los componentes. La totalidad de datos se puede gestionar de manera consistente (por ejemplo, respaldos realizados al mismo tiempo), pues todos están en un lugar. Como desventaja es que el repositorio es un punto de falla único, de modo que los problemas en el repositorio afectan a todo el sistema. Es posible que haya ineficiencias al organizar toda la comunicación a través del repositorio. Quizá sea difícil distribuir el repositorio por medio de varias computadoras.

f) Otros estilos arquitectónicos

1) Cliente/Servidor:

Define una relación entre dos aplicaciones en las cuales una de ellas (cliente) envía peticiones a la otra (servidor y fuente de datos).

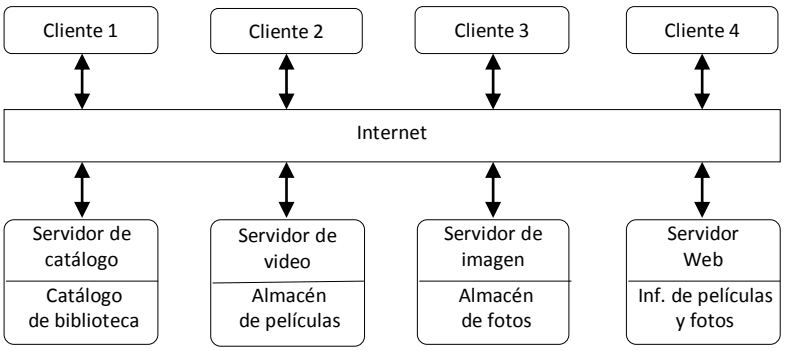


Ilustración 25: Arquitectura cliente – servidor.

Fuente: (Sommerville, 2011).

Para (Sommerville, 2011), se usa cuando, desde varias ubicaciones, se tiene que ingresar a los datos en una base de datos compartida. Como los servidores se pueden replicar, también se usan cuando la carga de un sistema es variable. La principal ventaja de este modelo es que los servidores se pueden distribuir a través de una red. La funcionalidad general (por ejemplo, un servicio de impresión) estaría disponible a todos los clientes, así que no necesita implementarse en todos los servicios. La desventaja es que cada servicio es un solo punto de falla, de modo que es susceptible a ataques de rechazo de servicio o a fallas del servidor. El rendimiento resultará impredecible porque depende de la red, así como del sistema. Quizás haya problemas administrativos cuando los servidores sean propiedad de diferentes organizaciones.

2) *Basado en componentes:*

Conjunto de componentes que exponen interfaces bien definidas y que colaboran entre sí para resolver el problema. Diseñado de forma que puedan ser reutilizados en distintos escenarios en distintas aplicaciones aunque algunos componentes son diseñados para tareas específicas.

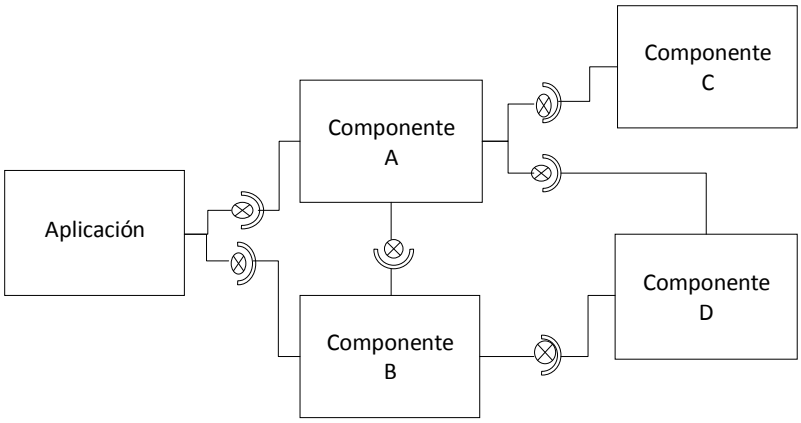


Ilustración 26: Arquitectura basada en componentes.

Fuente: (Blanca T., 2010)

3) *Arquitectura en capas (N-Layer):*

Distribución jerárquica de los roles y las responsabilidades para proporcionar una división efectiva de los problemas a resolver. Los roles indican el tipo y forma de interacción con otras capas y las responsabilidades la funcionalidad que implementan.

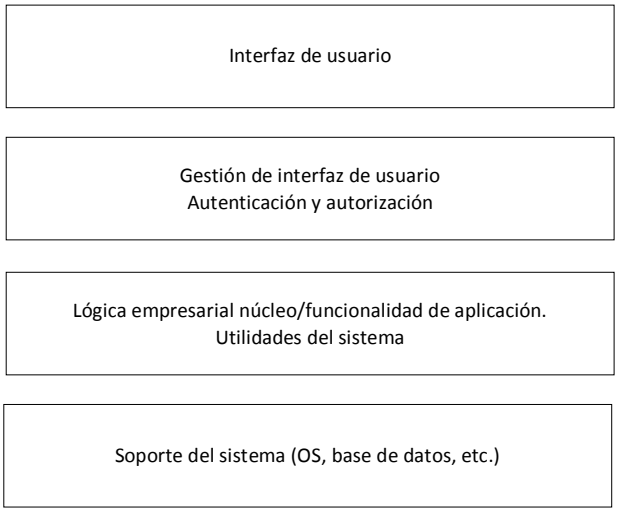


Ilustración 27: Arquitectura en capas (N-Layer).
Fuente: (Sommerville, 2011).

En (Sommerville, 2011) se menciona que se usa al construirse nuevas facilidades encima de los sistemas existentes; cuando el desarrollo se dispersa a través de varios equipos de trabajo, y cada uno es responsable de una capa de funcionalidad; cuando exista un requerimiento para seguridad multinivel. Permite la sustitución de capas completas en tanto se conserve la interfaz. Para aumentar la confiabilidad del sistema, en cada capa pueden incluirse facilidades redundantes (por ejemplo, autenticación). La desventaja en la práctica, suele ser difícil ofrecer una separación limpia entre capas, y es posible que una capa de nivel superior deba interactuar directamente con capas de nivel inferior, en vez de que sea a través de la capa inmediatamente abajo de ella. El rendimiento suele ser un problema, debido a múltiples niveles de interpretación de una solicitud de servicio mientras se procesa en cada capa.

4) Arquitectura de modelo de vista del controlador

Separa presentación e interacción de los datos del sistema. El sistema se estructura en tres componentes lógicos que interactúan entre sí. El componente Modelo maneja los datos del sistema y las operaciones asociadas a esos datos. El componente Vista define y gestiona cómo se presentan los datos al usuario. El componente Controlador dirige la interacción del usuario (por ejemplo, teclas oprimidas, clics del mouse, etcétera) y pasa estas interacciones a Vista y Modelo. (Sommerville, 2011)

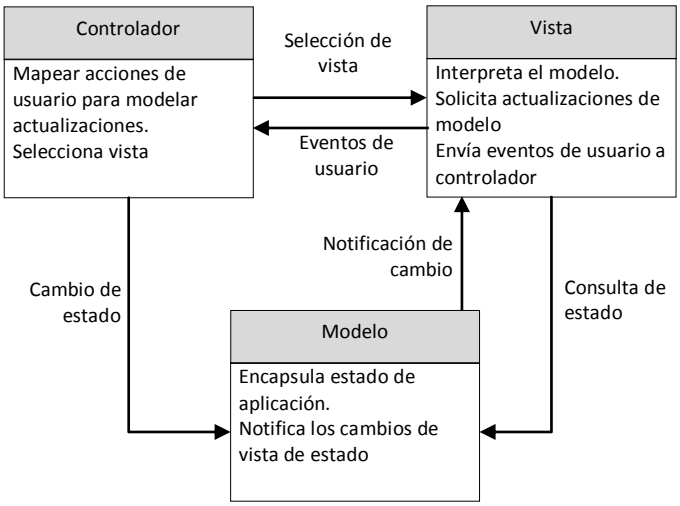


Ilustración 28: Arquitectura de modelo de vista del controlador.

Fuente: (Sommerville, 2011).

Para (Sommerville, 2011), sugiere que se usa cuando existen múltiples formas de ver e interactuar con los datos. También se utiliza al desconocerse los requerimientos futuros para la interacción y presentación. Permite que los datos cambien de manera independiente de su representación y viceversa. Soporta en diferentes formas la presentación de los mismos datos, y los cambios en una representación se muestran en todos ellos. Puede implicar código adicional y complejidad de código cuando el modelo de datos y las interacciones son simples.

5) Arquitectura en capas (N-Tier):

Conceptualmente igual que la arquitectura en capas (n-layer), aunque se define la separación de la funcionalidad en segmentos físicos separados (Tier). Normalmente la separación física se realiza en servidores diferenciados por razones de escalabilidad, seguridad, o simplemente necesidad.

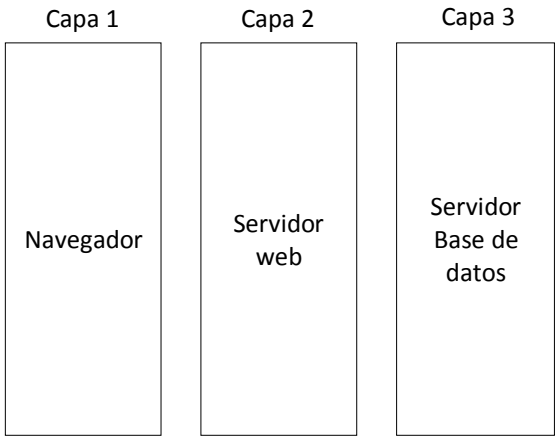


Ilustración 29: Arquitectura en capas (N-Tier)

Fuente: (Rojas, 2008).

2.2.3.10. Clases de diseño

Las clases de diseño presentan un mayor detalle técnico, pues son la guía para la implementación, Arlow y Neustad citados por Pressman sugieren características para que una clase esté bien formada:

a) Completa y suficiente:

Una clase de diseño debe ser la encapsulación correcta de todos los atributos y métodos que se pueden esperar, en forma razonable con base en una interpretación reconocible del nombre de la clase.

b) Primitivismo:

Los métodos asociados con una clase de diseño deben enfocarse en el cumplimiento de un servicio para la clase. Una vez que el servicio ha sido implementado con un método, la clase no debe proporcionar otra forma de complementar la misma cosa.

c) Cohesión alta:

Una clase de diseño cohesiva tiene un conjunto de responsabilidades pequeño y enfocado, y aplica atributos y métodos de manera sencilla para implementar dichas responsabilidades. Mientras cada método se enfoque solo en atributos asociados se mantendrá la cohesión

d) Acoplamiento bajo:

Dentro del modelo de diseño es necesario que las clases de diseño colaboren con alguna otra, sin embargo la colaboración debe de mantenerse en un mínimo aceptable. Si un modelo de diseño tiene un acoplamiento alto el sistema será difícil de implementar.

2.2.3.11. Implicancia de la cohesión en el diseño de un software

La cohesión implica que un componente o una clase solo encapsula atributos y operaciones relacionadas estrechamente entre sí y con la clase del propio componente (Pressman, 2009).

La cohesión tiene que ver con la forma en la que agrupamos unidades de software en una unidad mayor. Por ejemplo, la forma en la que agrupamos funciones en una librería, o la forma en la que agrupamos métodos en una clase.

Pressman cita la definición de varios tipos diferentes de cohesión:

a) Cohesión funcional:

Principalmente para operaciones, este grado de cohesión se presenta cuando un módulo realiza un solo cálculo y luego devuelve un resultado.

Se da cuando la función realiza una única operación, conllevando a esto una alta cohesión.

b) Cohesión de capa:

Exhibido para paquetes, componentes y clases, este tipo de cohesión ocurre cuando una capa superior tiene acceso a los servicios de una inferior, pero esta no tiene acceso a aquella.

c) Cohesión de comunicación:

Todas las operaciones con acceso a los mismos datos se definen dentro de una clase. En general, esa clase solo se concentra en los datos en cuestión, accesándolos y almacenándolos.

La cohesión tiene un alcance a nivel del grado en que una rutina cumple solo una función se debe alcanzar un alto grado de cohesión, sin embargo existen niveles inferiores de cohesión que se describen a continuación:

d) Cohesión secuencial:

Los componentes o las operaciones están agrupados de manera que el primero permita la entrada al siguiente, siguiendo esa secuencia en orden específico.

e) Cohesión procedimental:

Los componentes o las operaciones están agrupados de manea que permiten la invocación de uno inmediatamente después de que se invoque el anterior aunque no se hayan pasado datos entre ellos.

Se diferencia de la cohesión secuencial en que no se refieren a un dato en todo su proceso sino que se desenvuelven de manera independiente.

f) Cohesión temporal:

Las operaciones que se realizan reflejan un comportamiento o estado específico, como una operación que se realiza al principio o todas las operaciones realizadas cuando se detecta un error.

Significa que agrupamos unidades simplemente porque tienen que ejecutarse dentro del mismo periodo de tiempo, pero que no exista ninguna relación entre ellas, ni de forma secuencial, ni de comunicación, ni estén dentro de la misma categoría lógica.

g) Cohesión utilitaria:

Se han agrupado componentes, clases u operaciones que existen dentro de la misma categoría, pero que no tienen otra relación.

h) Cohesión lógica:

Esta cohesión se da cuando se efectúa distintas operaciones en una misma función bajo un parámetro de control

2.2.3.12. Implicancia del acoplamiento en el diseño de software

En (Sommerville, 2011), se refiere a que la comunicación y colaboración son elementos esenciales de cualquier sistema orientado a objetos. Sin embargo, hay un lado oscuro en esa importante y necesaria característica. A medida que aumenta la cantidad de comunicación y colaboración (es decir, a medida que crece el grado de conectividad entre clases), también aumenta la complejidad del sistema. Y a medida que ésta aumenta, la dificulta de implementar, probar y mantener el software.

En comparación con la cohesión que describe el principio que encapsula atributos y operaciones estrechamente relacionadas entre sí y con la clase del propio componente, el grado de acoplamiento describe la manera en que la clase se conectan con otras clases del sistema

El acoplamiento se manifiesta de varias formas según Lethbrigde y Laganiére citado por Pressman:

a) Acoplamiento de control:

Se presenta cuando una operación invoca a otra operación y le pasa una marca de control que condiciona a redirigir su flujo lógico interno de la segunda operación.

b) Acoplamiento de estampa:

Se da cuando una clase se declara como tipo para un argumento de otra clase. Por lo tanto la primera clase es parte de la definición de la segunda clase.

c) Acoplamiento de datos:

Ocurre cuando las operaciones pasan cadenas largas de argumentos de datos. El "ancho de banda" de la comunicación entre clases y componentes crece y la complejidad aumenta.

d) Acoplamiento de llamada a rutina:

Ocurre cuando una operación invoca a otra. Este grado de acoplamiento es común y, a menudo, muy necesario. Sin embargo, aumenta la conectividad de un sistema.

e) Acoplamiento de uso de tipo:

Ocurre cuando un componente A usa un tipo de datos definidos en el componente B.

f) Acoplamiento por inclusión o importación:

Ocurre cuando un componente A importa o incluye un paquete o el contenido del componente B.

g) Acoplamiento externo:

Ocurre cuando un componente se comunica o colabora con componentes de infraestructura.

Se describen a continuación los acoplamientos no deseados

h) **Acoplamiento del contenido:**
Ocurre cuando un componente modifica subrepticiamente datos internos de otro.

Es decir, en general, cuando programamos una primera unidad que está acoplada a una segunda, de esta sólo necesitamos conocer sus puntos de entrada y los resultados que nos va a devolver. No debe ser necesario conocer ningún detalle de la implementación de la segunda.

i) **Acoplamiento común:**
Ocurre cuando varios componentes usan una variable global.

Decimos que dos unidades están globalmente acopladas cuando se pasan datos entre sí a través de una estructura global.

Finalmente es cierto que existe la necesidad que las unidades del software se comunique entre ellas, de lo contrario se realizaría un sistema con componentes aislados sin ningún propósito fundamental al nivel global del sistema, lo que se debe hacer es reducir el acoplamiento en la medida que sea posible y que se busquen alternativas para poder evitarse.

2.3. Marco conceptual

2.3.1. Aspecto
Una unidad modular que se disemina por la estructura de otras unidades funcionales

2.3.2. Competencia
Es un requerimiento del sistema en un nivel determinado

2.3.3. Complejidad
Indica las interacciones y el número de estados posibles que se producen a través de éstos

2.3.4. Diseño
Es un proceso o labor a proyectar, coordinar, seleccionar y organizar un conjunto de elementos

2.3.5. Diseño de software
El diseño de software es la actividad que tiene por fin definir la arquitectura del sistema en términos de software y el definir el soporte para ello, también la creación de subsistemas, interfaces, y los componentes del software que son necesarios para implementar el sistema

2.3.6. Efectividad
Capacidad de lograr el efecto que se desea o se espera.

2.3.7. Efecto
Aquello que sigue por virtud de una causa, resultado de una causa

2.3.8. Objeto

Una entidad caracterizada por un estado; su comportamiento se define por las operaciones que puede realizar

2.3.9. Requerimiento funcional

Son declaraciones de los servicios que debe proporcionar el sistema, de la manera en que éste debe reaccionar a entradas particulares y de cómo se debe comportar en situaciones particulares.

2.3.10. Requerimiento no funcional

Son restricciones de los servicios o funciones ofrecidos por el sistema. Especifica criterios que pueden usarse para juzgar la operación de un sistema en lugar de sus comportamientos específicos, ya que éstos corresponden a los requisitos funcionales.

2.3.11. Separación de competencias

Es un intento de descomposición del software en módulos, donde cada uno de los cuales representa y encapsula un problema particular de interés.

3. Desarrollo

3.1. Descripción del caso de estudio:

Como implementación del caso de estudio se presenta el diseño de un software para la "Biblioteca de la Universidad Nacional de Moquegua Sede – Ilo" en el primer caso con la programación orientada a objetos y el segundo caso con la programación orientada a aspectos.

La Universidad Nacional de Moquegua cuenta con tres sedes: Sede Ilo, Sede Moquegua y Sede Extensión Ichuña. La biblioteca de la UNAM sede ILO funciona en las instalaciones del campus universitario Sede Ilo, y es el lugar donde se realiza todos los procesos propios de sus funciones, como es el caso de lo siguiente:

Préstamo de material bibliográfico: los usuarios sean estudiantes, egresados, personal administrativo, docentes y público en general para hacer uso del servicio de la biblioteca solicitan primeramente un libro de registro de ejemplares (libros) cada uno categorizado por especialidad o haciendo uso de un software simple de búsqueda de libros, luego el usuario dicta el código del libro y el bibliotecario realiza la búsqueda en los estantes, seguidamente si lo encuentra se le entrega una ficha al usuario para que rellene sus datos personales, carrera, datos del libro y una vez llenado entrega la ficha con su carnet de identificación. Y por último se hace la entrega del libro dándole un plazo de 2 días hábiles según reglamente para su entrega.

Devolución de material bibliográfico: El usuario realiza la devolución física del libro y el bibliotecario revisa la integridad del material, además verifica la fecha de entrega de la ficha de préstamo si está dentro de los días hábiles de préstamo establecido, si está dentro de los días establecidos se le entrega su carnet, quedando la ficha como material estadístico. Si está fuera del rango de días de préstamo establecido se le carga una multa de dos soles por días retrasados según reglamento y se suspende el servicio al usuario por una cantidad de días determinada según su categoría.

Realización de reportes de préstamos: El reporte se realiza de manera mensual, incluyendo en él un cruce de coincidencias entre categoría del libro y carrera del solicitante.

Administración de material bibliográfico: Consta de la catalogación, control de ubicaciones y estado de material. En el anexo Nro. IV se realiza un mayor detalle.

3.1.1. Requerimientos del software (lenguaje natural)

Se requiere un sistema que cubra los siguientes requerimientos:

- Registro, actualización, cambio de ubicación, cambio de estado, cambio de categoría, dar de baja y revisión de disponibilidad de ejemplares.
- Registro de préstamo y devolución de ejemplares.

- Registro de préstamo con fecha de entrega automática
- Autorización al uso de secciones del sistema
- Control de cuentas y sanciones de cada usuario
- Reporte de ejemplares, prestamos, retrasos, usuarios por intervalo de fechas
- Validación de ingreso de datos
- Autentificación antes de entrar a cualquier módulo.
- Control de disponibilidad de ejemplares
- Búsqueda de material bibliográfico vía web.
- Reserva de material bibliográfico y cancelación de material bibliográfico vía web
- Control de préstamos.
- Control de ciclos académicos.
- Importación de lista de ejemplares hacia el sistema
- Importación de lista de usuarios hacia el sistema
- Administración de Backup del sistema
- Restricción del uso de préstamos y reservas solo en un ciclo existente.
- Registro de actividad en el sistema
- Registro de la institución.
- Ingresar libros con su clasificación (incluyendo copias).
- Dar de baja a libros.
- Modificación de propiedades de los libros.
- Creación de cuentas de usuarios de la biblioteca
- Modificación de cuentas de usuarios.
- Cambio de escuela y perfiles de usuario
- Crear cuentas de bibliotecario.
- Modificación de cuenta de bibliotecario.
- Dar de baja cuentas de bibliotecario.
- Sanciones por atraso en las devoluciones.
- Permitir préstamos a usuarios esporádicos que se acerquen a la biblioteca.
- Permitir a los usuarios del sistema actualizar sus datos personales.
- Permitir ingreso al sistema mediante usuario y contraseña.

En la descripción del caso de estudio se ha realizado una especificación de requerimientos en un lenguaje natural expresado por el usuario para describir lo que espera que el sistema realice, así mismo sirve como base para tener una idea inicial del sistema, pero presenta problemas como explica (sommerville, 2008): La falta de claridad, confusión de requerimientos y conjunción de requerimientos. En los casos siguientes se realizará el proceso del diseño del software cada uno pensado según la forma de como concebir un software según el tipo de programación que se va a aplicar y ubicar los requerimientos funcionales y no funcionales en el diseño del software.

3.2. Caso 1: Diseño de un software orientado a objetos

Para el desarrollo del siguiente caso, se optó por seguir la Metodología *Rational Unified Process* más conocido como RUP. Porque permite seleccionar el conjunto de componentes de proceso para que se ajusten a las necesidades específicas del proyecto, además ofrece una guía en cada etapa del desarrollo del software, lo que permite una específica tarea en cada etapa, y que toda secuencia de pasos es objeto de modificaciones a lo largo del proceso de desarrollo, con lo cual, sólo al término del proceso podríamos tener una versión definitiva y completa.

3.2.1. Requerimientos

Los requerimientos para un sistema son descripciones de lo que el sistema debe hacer: el servicio que ofrece y las restricciones en su operación. Tales requerimientos reflejan las necesidades de los clientes por un sistema que atienda cierto propósito, como sería controlar un dispositivo, colocar un pedido o buscar información. (Sommerville, 2011).

Roger S. Pressman, conocido ingeniero de software y autor de varios libros de Ingeniería del Software, expresa que, para que un esfuerzo de desarrollo de software tenga éxito, es esencial comprender perfectamente los requisitos del software. Independientemente de lo bien diseñado o codificado que esté un programa, si se ha analizado y especificado pobremente, decepcionará al usuario y desprestigiará al que lo ha desarrollado.

Partiendo de estas definiciones queda claro que el objetivo del requerimiento es describir lo que debería hacer el sistema y permitir a los desarrolladores y al cliente ponerse de acuerdo con esa descripción.

Los casos de uso nos ayudan a capturar los requerimientos funcionales del sistema desde la perspectiva del usuario de esta forma da una vision temprana del sistema. Aunque existen diversas aproximaciones para realizar los requerimientos del sistema, los diagramas de casos de uso brindan una perspectiva grafica entendible entre usuario y analista para el modelado inicial del sistema con esto minimizamos incongruencias, incompatibilidades, redundancias entre los requerimientos en lenguahe natural del usuario.

Un modelo de caso de uso describe lo que hace un sistema sin describir como lo hace, es decir es un modelo lógico del sistema (Kendall, 2005). Kendall sugiere algunos lineamientos para la identificacion:

1. Revise las especificaciones del negocio e identifique los actores en el dominio del problema.

2. Identifique los eventos de alto nivel y desarrolle los casos de uso principales que describen dichos eventos y cómo los inician los actores. Examine cuidadosamente los papeles jugados por los actores para identificar todos los posibles casos de uso principales iniciados por cada actor. No se necesita mostrar los casos de uso con poca o ninguna interacción del usuario.
3. Revise cada caso de uso principal para determinar las posibles variaciones de flujo a través del caso de uso. Con este análisis, establezca las rutas alternativas. Debido a que el flujo de eventos normalmente es diferente en cada caso, busque actividades que podrían tener éxito o fallar. También busque cualesquier ramas en la lógica de caso de uso en que son posibles resultados diferentes.

Habiendo visto los detalles de la etapa de requerimientos se procedió a capturar e identificar los escenarios principales, entre ellos identificar a los actores, y su interacción en relaciones con el sistema.

3.2.1.1. Casos de uso

a) Préstamo de ejemplar

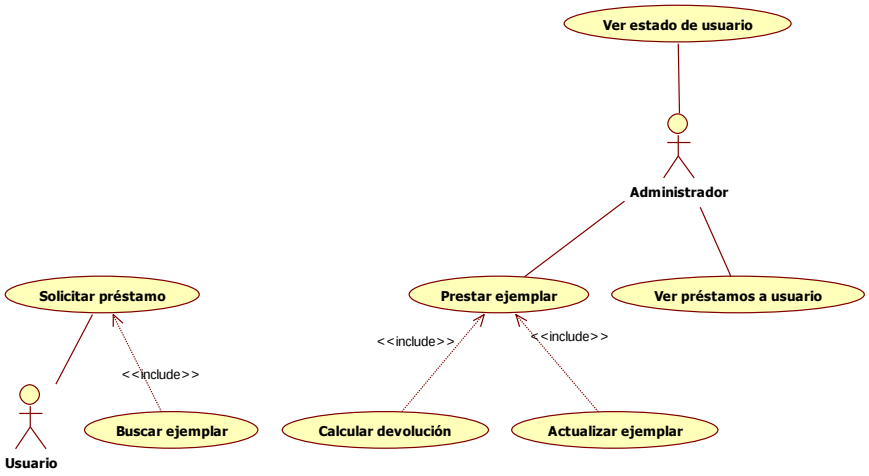


Ilustración 30: Caso de uso préstamo de ejemplar

Fuente: Elaboración propia.

Caso de Uso	Préstamo de ejemplar
Fuentes	Bibliotecario
Actor	Primario: Usuario, Secundario: Bibliotecario
Descripción	Préstamo de ejemplar
Flujo básico	El usuario busca un ejemplar Solicita el préstamo del ejemplar El bibliotecario verifica la disponibilidad del ejemplar en el sistema El bibliotecario solicita al usuario su código El bibliotecario ingresa el código al sistema

	Se registra el préstamo El sistema muestra la fecha de devolución. Fin del caso de uso
Flujos alternos	Paso 3: Si el ejemplar no está disponible. El sistema envía un aviso indicando que no está disponible. Se cancela el caso de uso. Paso 5: si no existe el usuario Se registra al usuario en el sistema. Se continúa con el caso de uso. Paso 6: No se registra el préstamo: El ejemplar fue reservado durante la atención. El usuario está suspendido. El usuario tiene cuentas por pagar pendientes. El usuario ha excedido el número de préstamos permitidos. El sistema muestra un aviso indicando los motivos. Se cancela el caso de uso
Pre-condiciones	El bibliotecario debe haber iniciado sesión El bibliotecario debe estar activo El bibliotecario debe tener los permisos de registrar un préstamo. Los datos ingresados se deben validar con el sistema Debe existir un ciclo en curso
Post-condiciones	El sistema actualiza el estado del ejemplar prestado a 'Prestado'. El sistema debe mostrar la lista de préstamos pendientes a devolución. El sistema registra el nombre de usuario del administrador que realizo el préstamo.
Notas	El número de máximo de préstamos permitidos y la duración de los préstamos depende de la política de la biblioteca y puede cambiar en el futuro.

Tabla 1: Préstamo de ejemplar

Fuente: Elaboración propia.

b) Reservar ejemplar

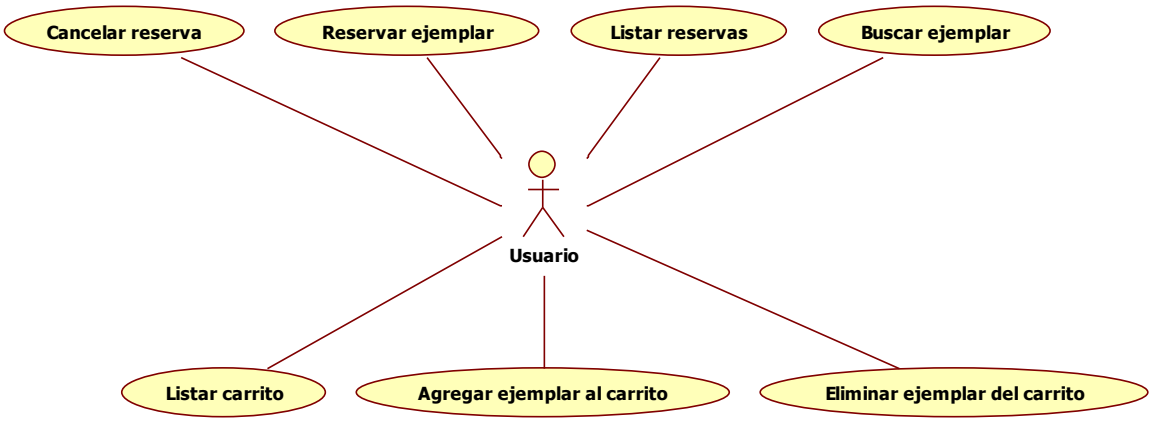


Ilustración 31: Reservar ejemplar

Fuente: Elaboración propia.

Caso de Uso	Reserva de ejemplar
Fuentes	Bibliotecario
Actor	Primario: Usuario
Descripción	Reserva de ejemplar
Flujo básico	1. El usuario realiza una búsqueda 2. El usuario agregar un ejemplar al carrito 3. El usuario lista el carrito de sus pre-reservas 4. El usuario reserva sus ejemplares 5. Fin del caso de uso
Flujos alternos	Paso 2: No se selecciona el ejemplar si 1. Está reservado 2. Esta prestado 3. Se regresa al paso 1 Paso 3: El usuario puede eliminar un ítem 1. El usuario elimina un ejemplar del carrito 2. El sistema lista el carrito de pre-reservas 3. Continúa el caso de uso. Paso 4: No se realiza la reserva si: 1. Si el ejemplar ya fue prestado durante el proceso de pre-reservación 2. El usuario está suspendido. 3. El usuario tiene cuentas por pagar pendientes. 4. El usuario ha excedido el número de préstamos permitidos. 5. El sistema muestra un aviso indicando los motivos. 6. Fin del caso de uso
Pre-condiciones	1. Para buscar ejemplares no es necesario estar registrado 2. Para solicitar una reserva es necesario iniciar sesión 3. Los datos ingresados se deben validar con el sistema 4. Debe existir un ciclo en curso
Post-condiciones	1. El sistema lista las reservas pendientes 2. El sistema actualiza el estado del ejemplar a 'Reservado'.
Notas	1. El número de máximo de reservas permitidas es la diferencia entre el número máximo de reservas permitidas para el tipo de usuario menos los ejemplares prestados.

Tabla 2: Reserva de ejemplar

Fuente: Elaboración propia.

c) Devolución de ejemplares

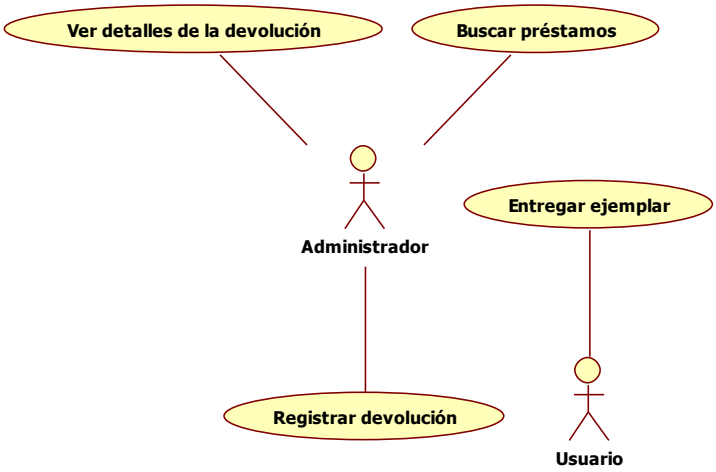


Ilustración 32: Devolución de ejemplares

Fuente: Elaboración propia.

Caso de Uso	Devolución de ejemplares
Fuentes	Bibliotecario
Actor	Primario: Usuario, Bibliotecario
Descripción	Devolución de ejemplares
Flujo básico	1. El usuario entrega el ejemplar 2. El bibliotecario busca los prestamos 3. El bibliotecario selecciona la devolución 4. El sistema muestra detalles del préstamo 5. El bibliotecario realiza el registro de la devolución 6. El sistema registra la devolución. 7. El sistema muestra detalles de la devolución 8. Fin del caso de uso
Flujos alternos	Paso 6: El sistema puede realizar lo siguiente en la devolución: 1. Sancionar al usuario si excedió los días máximos de préstamos permitidos. 2. Suspender al usuario si es sancionado. 3. Continúa el caso de uso.
Pre-condiciones	1. El bibliotecario debe haber iniciado sesión 2. El bibliotecario debe estar activo 3. El bibliotecario debe tener los permisos de registrar una devolución. 4. Debe existir el préstamo. 5. Los datos ingresados se deben validar con el sistema
Post-condiciones	1. El sistema lista la lista de préstamos. 2. El sistema actualiza el estado del ejemplar a 'Disponible'. 3. El sistema registra el nombre de usuario del administrador que realizo la devolución.

Notas	1. Para la devolución del ejemplar en préstamo no es necesario que exista un ciclo en curso. 2. Los días de sanciones de dan de acuerdo a la reincidencia especificados y depende de la política de la biblioteca y puede cambiar en el futuro modificando los perfiles de usuarios.

Tabla 3: Devolución de ejemplares

Fuente: Elaboración propia.

d) Importar ejemplares

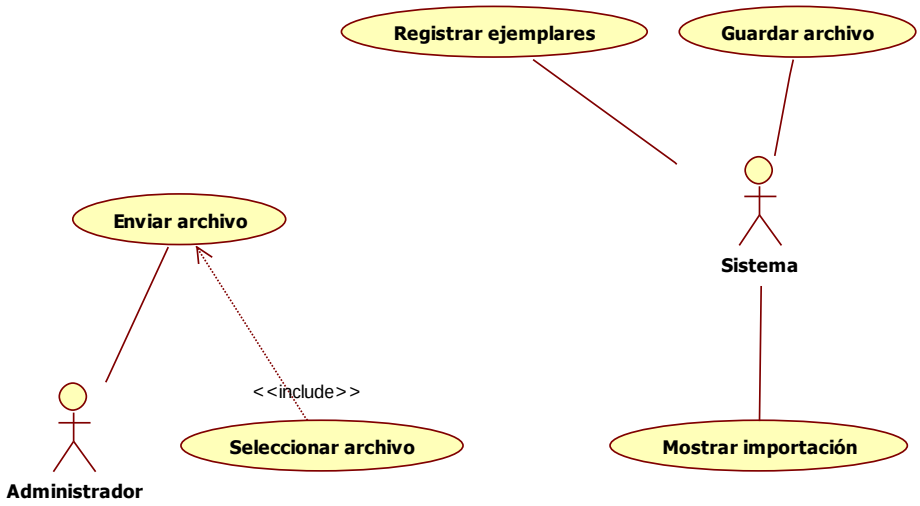


Ilustración 33: Importar ejemplares

Fuente: Elaboración propia.

Caso de Uso	Importar ejemplares
Fuentes	Bibliotecario
Actor	Primario: Bibliotecario
Descripción	Importar ejemplares
Flujo básico	1. El bibliotecario selecciona el archivo a importar 2. El bibliotecario envía el archivo al sistema 3. El sistema guarda el archivo importado 4. EL sistema registra la lista de ejemplares 5. Fin del caso de uso
Flujos alternos	Paso 3: el sistema no guarda el archivo importado 1. Muestra un mensaje indicando el error 2. Continua el caso de uso Paso 4: El sistema no realiza un registro 1. Muestra detalles del error 2. Continua el caso de uso
Pre-condiciones	1. El bibliotecario debe haber iniciado sesión

	2. El bibliotecario debe estar activo 3. El bibliotecario debe tener los permisos de realizar la importación 4. Se debe adjuntar un formato válido. 5. Los datos ingresados se deben validar con el sistema
Post-condiciones	1. El sistema muestra los detalles de la importación sean con éxito, duplicados, o inválidos 2. El sistema registra el nombre de usuario del administrador que realizo la operación.
Notas	1. El sistema podrá proporcionar un formato de archivo para el llenado de la lista de ejemplares.

Tabla 4: Importar ejemplares

Fuente: Elaboración propia.

e) Exportar copia de seguridad

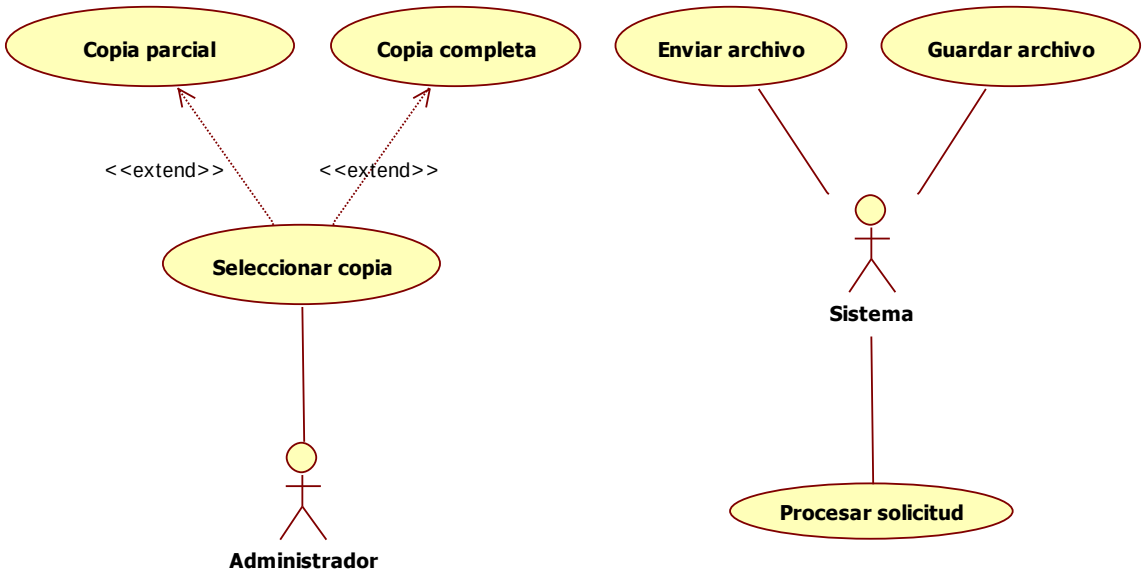


Ilustración 34: Exportar copia de seguridad

Fuente: Elaboración propia.

Caso de Uso	Exportar copia de seguridad
Fuentes	Bibliotecario
Actor	Primario: Bibliotecario
Descripción	Exportar copia de seguridad
Flujo básico	1. El bibliotecario selecciona el tipo de copia de seguridad 2. El sistema procesa la solicitud 3. El sistema envía un archivo de la copia de seguridad 4. El sistema guarda el archivo en el servidor 5. Fin del caso de uso

Flujos alternos	Paso 1: El bibliotecario puede seleccionar una de estas opciones: 1. Seleccionar una copia parcial 2. Seleccionar una copia completa 3. Continua el caso de uso Paso 4: El sistema no guarda el archivo 1. Muestra detalles del error 2. Continua el caso de uso
Pre-condiciones	1. El bibliotecario debe haber iniciado sesión 2. El bibliotecario debe estar activo 3. El bibliotecario debe tener los permisos para exportar una copia de seguridad
Post-condiciones	1. El sistema registra el nombre de usuario del administrador que realizo la operación.
Notas	1. El sistema podrá proporcionar una información acerca del tipo de copia de seguridad que piensa realizar el bibliotecario.

Tabla 5: Exportar copia de seguridad

Fuente: Elaboración propia.

f) Importar copia de seguridad

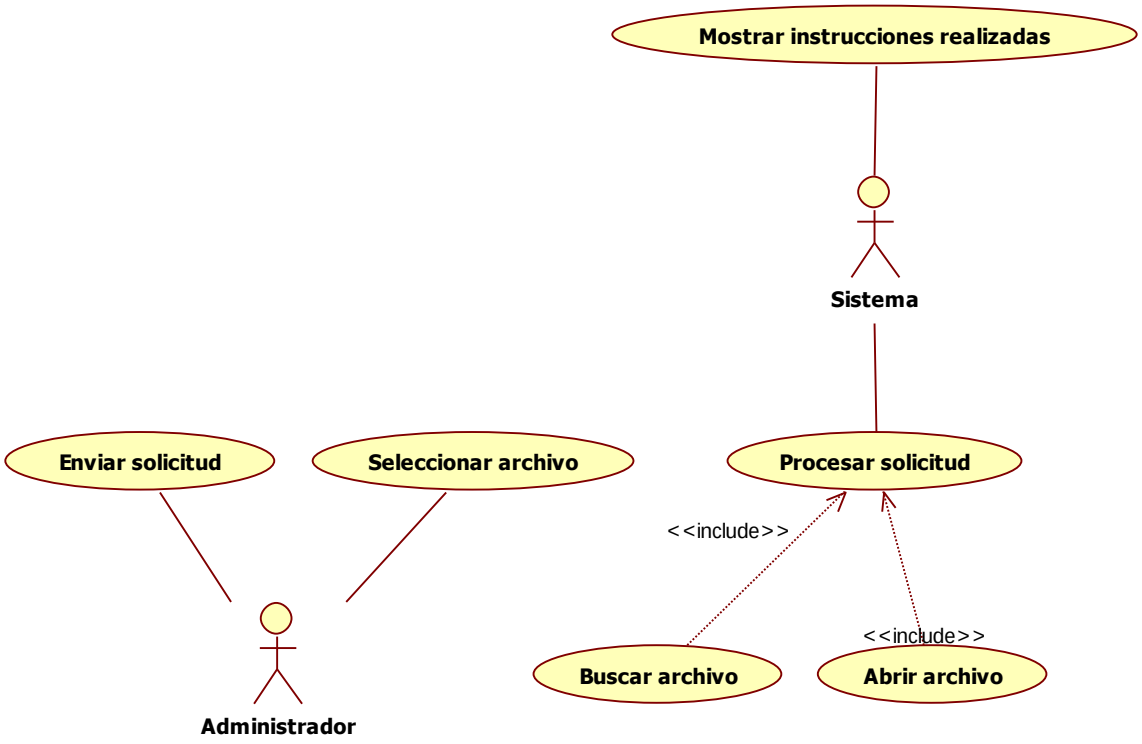


Ilustración 35: Importar copia de seguridad

Fuente: Elaboración propia.

Caso de Uso	Importar copia de seguridad
Fuentes	Bibliotecario
Actor	Primario: Bibliotecario
Descripción	Importar copia de seguridad
Flujo básico	1. El bibliotecario selecciona el archivo de copia de seguridad 2. El bibliotecario envía la solicitud 3. El sistema procesa la solicitud 4. El sistema muestra las instrucciones SQL realizadas 5. Fin del caso de uso
Flujos alternos	Paso 3: El sistema no procesa si: 1. No encuentra el archivo 2. No abre el archivo
Pre-condiciones	1. El bibliotecario debe haber iniciado sesión 2. El bibliotecario debe estar activo 3. El bibliotecario debe tener los permisos para importar una copia de seguridad
Post-condiciones	1. El sistema registra el nombre de usuario del administrador que realizo la operación.
Notas	1. El sistema podrá proporcionar una información acerca del punto de restauración realizada

Tabla 6: Importar copia de seguridad

Fuente: Elaboración propia.

g) Administración de usuarios

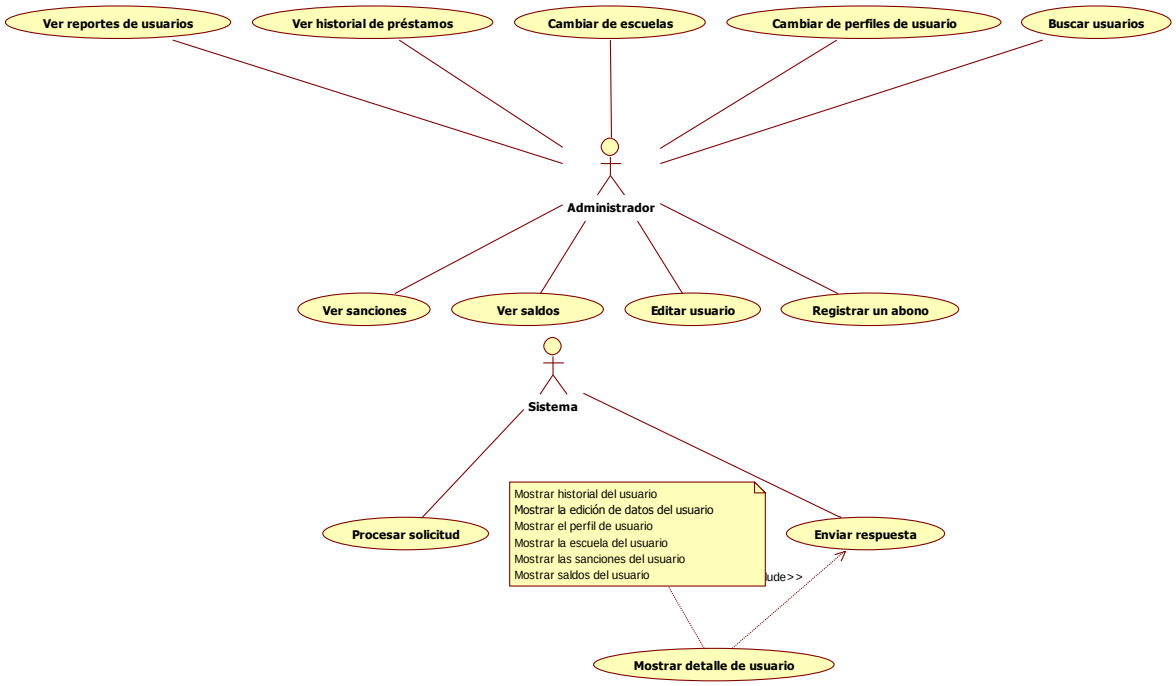


Ilustración 36: Administración de usuarios

Fuente: Elaboración propia.

Caso de Uso	Administración de usuarios
Fuentes	Bibliotecario
Actor	Primario: Bibliotecario
Descripción	Administración de usuarios
Flujo básico	1. El bibliotecario envía una opción 2. El sistema procesa la solicitud 3. El sistema envía la respuesta 4. El bibliotecario realiza acciones 5. El sistema procesa la solicitud 6. El sistema envía al respuesta 7. Fin del caso de uso
Flujos alternos	Paso 3: El sistema puede enviar una de las siguientes respuestas: 1. Mostrar historial del usuario 2. Mostrar la edición de datos del usuario 3. Mostrar el perfil de usuario 4. Mostrar la escuela del usuario 5. Mostrar las sanciones del usuario 6. Mostrar saldos del usuario Paso 4: El bibliotecario puede realizar una de las siguientes acciones: 1. Salir del historial del usuario 2. Editar datos del usuario 3. Cambiar perfil del usuario 4. Cambiar la escuela del usuario 5. Registrar el abono de un saldo pendiente
Pre-condiciones	1. El bibliotecario debe haber iniciado sesión 2. El bibliotecario debe estar activo 3. El bibliotecario debe tener los permisos para administrar todo o parte de la sección de usuarios.
Post-condiciones	1. El sistema registra el nombre de usuario del administrador que realizo la operación. 2. El sistema muestra un aviso de cambios realizados 3. Para todo caso el sistema regresa al paso 3
Notas	1. El perfil del usuario corresponde al tipo de usuario clasificado según la política de la biblioteca y puede cambiar en el futuro, así como también llevar un historial de cambios de la misma. 2. Luego de mostrar cualquier respuesta por parte del sistema, debe existir una opción de salida para regresar al punto inicial.

Tabla 7: Administración de usuarios

Fuente: Elaboración propia.

h) Gestión de utilidades

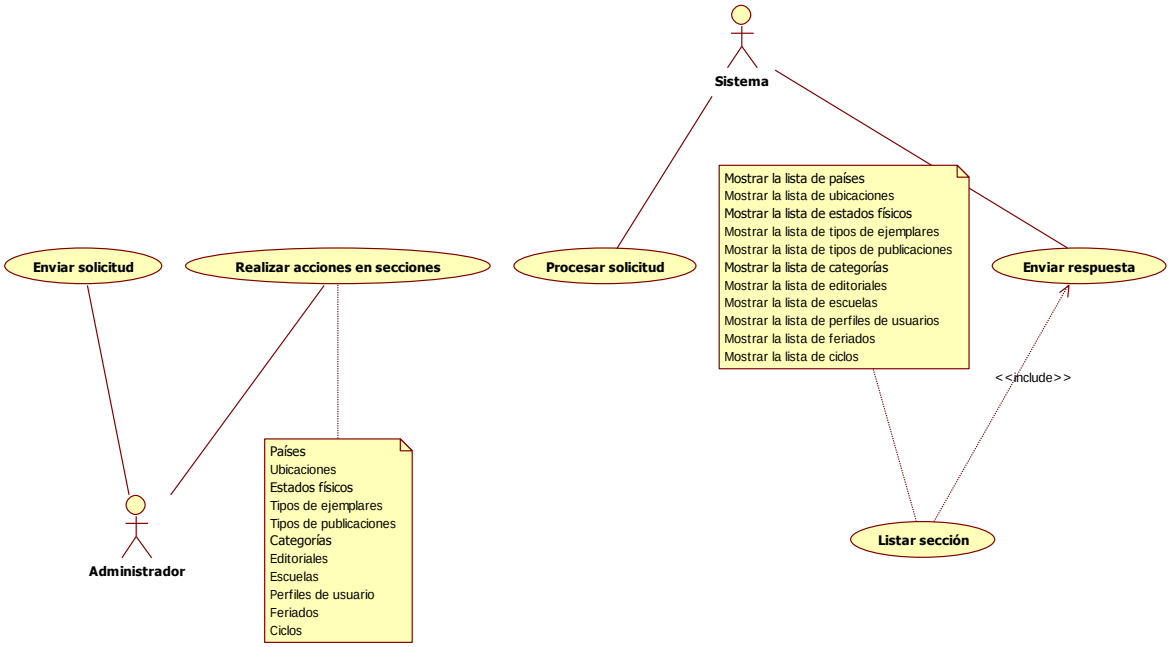


Ilustración 37: Gestión de utilidades

Fuente: Elaboración propia.

Caso de Uso	Gestión de utilidades
Fuentes	Bibliotecario
Actor	Primario: Bibliotecario
Descripción	En este caso de uso se administra secciones generales como son específicamente aquellas entidades independientes pero que en el momento de administrar publicaciones, usuarios, se hacen referencia a ellas para realizar operaciones como parte de estas mismas, además de ello tienen en común operaciones similares en su ciclo de vida.
Flujo básico	1. El bibliotecario selecciona una solicitud 2. El sistema procesa la solicitud 3. El sistema envía la respuesta 4. El bibliotecario realiza acciones 5. El sistema procesa la solicitud 6. El sistema envía al respuesta 7. Fin del caso de uso
Flujos alternos	Paso 3: El sistema puede enviar una de las siguientes respuestas: 1. Mostrar la lista de países 2. Mostrar la lista de ubicaciones 3. Mostrar la lista de estados físicos 4. Mostrar la lista de tipos de ejemplares 5. Mostrar la lista de tipos de publicaciones 6. Mostrar la lista de categorías 7. Mostrar la lista de editoriales 8. Mostrar la lista de escuelas 9. Mostrar la lista de perfiles de usuarios 10. Mostrar la lista de feriados 11. Mostrar la lista de ciclos

	Paso 4: El bibliotecario puede realizar las operaciones básicas en las secciones de: 1. Países 2. Ubicaciones 3. Estados físicos 4. Tipos de ejemplares 5. Tipos de publicaciones 6. Categorías 7. Editoriales 8. Escuelas 9. Perfiles de usuario 10. Feriados 11. Ciclos 12. Configuración
Pre-condiciones	1. El bibliotecario debe haber iniciado sesión 2. El bibliotecario debe estar activo 3. El bibliotecario debe tener los permisos para administrar cada una de las secciones. 4. Debe estar en la sección que desea administrar
Post-condiciones	1. El sistema registra el nombre de usuario del administrador que realizo la operación. 2. El sistema muestra un aviso de cambios realizados 3. Para toda acción realizada el sistema regresa a la sección donde se originaron las decisiones.
Notas	1. Luego de mostrar cualquier respuesta por parte del sistema, debe existir una opción de salida para regresar al paso 1. 2. Se entiende por operaciones básicas el registro, búsqueda, eliminación y actualización.

Tabla 8: Gestión de utilidades

Fuente: Elaboración propia.

i) Administrar ejemplares

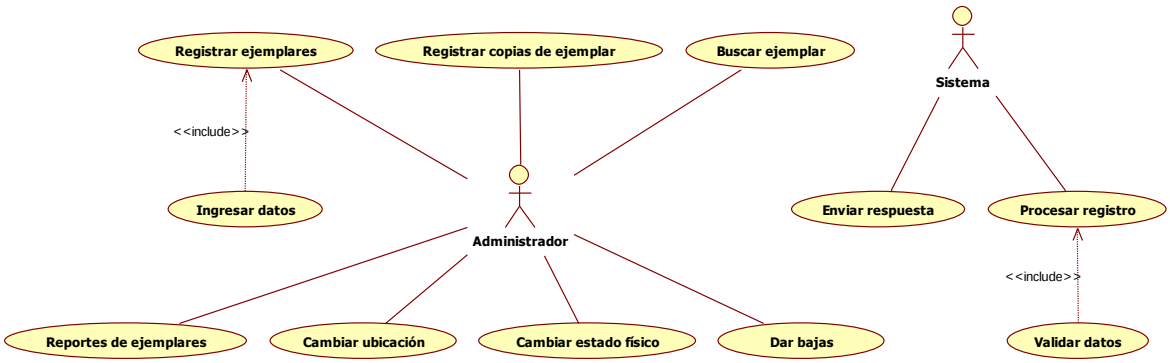


Ilustración 38: Administración de ejemplares

Fuente: Elaboración propia.

Caso de Uso	Administrar ejemplares
Fuentes	Bibliotecario
Actor	Primario: Bibliotecario (Administrador)
Descripción	Administrar ejemplares
Flujo básico	1. El bibliotecario selecciona una acción 2. El sistema procesa la solicitud 3. El sistema envía la respuesta 4. El bibliotecario realiza acciones 5. El sistema procesa la solicitud 6. El sistema envía la respuesta 7. Fin del caso de uso
Flujos alternos	Paso 1: El bibliotecario puede elegir una de las siguientes acciones: 1. Registrar ejemplares 2. Reportes 3. Cambios de ubicación 4. Cambios de estado físico 5. Búsqueda de ejemplares 6. Dar bajas 7. Realizar copias de ejemplar Paso 4: El bibliotecario puede realizar de cambios de estado para un ejemplar en las secciones de: 1. Ubicación 2. Estado físico
Pre-condiciones	1. El bibliotecario debe haber iniciado sesión 2. El bibliotecario debe estar activo 3. El bibliotecario debe tener los permisos para administrar cada una de las secciones.
Post-condiciones	1. El sistema registra el nombre de usuario del administrador que realizo una operación de cambio, registro, copia y baja. 2. El sistema muestra un aviso de cambios realizados 3. Para toda acción realizada el sistema regresa a la sección donde se originaron las decisiones.
Notas	1. Al registrar una baja no se borra el ejemplar solo se cambia de estado a 'Baja' inhabilitándolo de cualquier resultado de búsqueda, préstamos y reservas. 2. Para el registro de una copia de un ejemplar se deberá seleccionarla y continuar el proceso de registro normal de ejemplares con los campos de publicación pre-establecidos

Tabla 9: Administración de ejemplares

Fuente: Elaboración propia.

j) Gestión de administradores

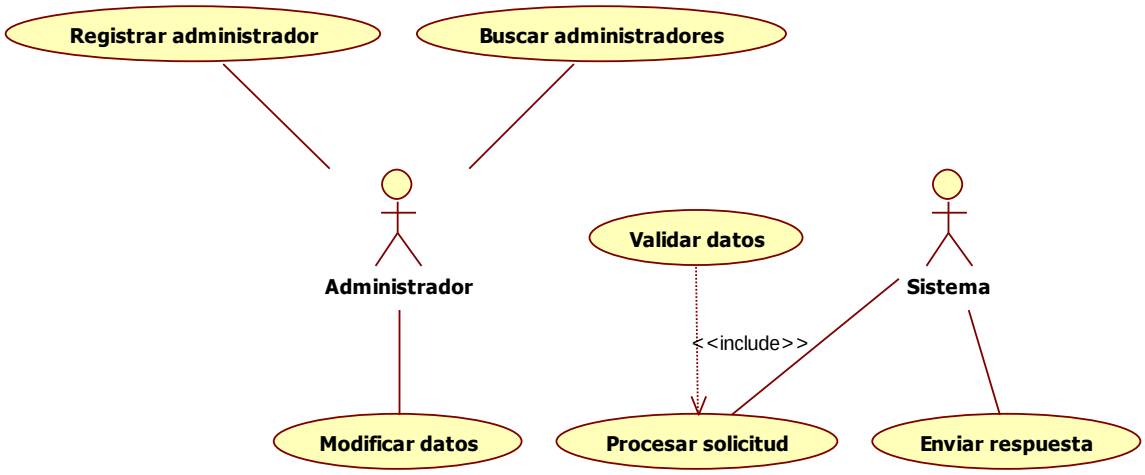


Ilustración 39: Gestión de administradores

Fuente: Elaboración propia.

Caso de Uso	Gestión de administradores
Fuentes	Bibliotecario
Actor	Primario: Bibliotecario (Administrador)
Descripción	Gestión de administradores
Flujo básico	1. El bibliotecario selecciona una acción 2. El sistema procesa la solicitud 3. El sistema envía la respuesta 4. Fin del caso de uso
Flujos alternos	Paso 1: El bibliotecario puede elegir una de las siguientes acciones: 1. Registrar administradores 2. Buscar administradores 3. Modificar datos
Pre-condiciones	1. El bibliotecario debe haber iniciado sesión 2. El bibliotecario debe estar activo 3. El bibliotecario debe tener los permisos para administrar cada una de las operaciones.
Post-condiciones	1. El sistema registra el nombre de usuario del administrador que realizo una operación de cambio, registro. 2. El sistema muestra un aviso ante cambios realizados 3. Para toda acción realizada el sistema regresa a la sección de administrador
Notas	1. No se puede eliminar un administrador 2. Se deben realizar validaciones ante el ingreso de datos

Tabla 10: Gestión de administradores

Fuente: Elaboración propia.

Hasta esta etapa se han identificado los principales escenarios del sistema, ademas se identificó a dos principales actores del sistema que son usuario y administrador quienes desarrollan la dinámica del sistema. La mayoría de requisitos funcionales estan comprendidos en algunos de los casos de uso, y otros requisitos que se identifican pero que no se asocian con ningun caso de uso sino se extiende una identificacion del requisito a traves de cada casos de uso, estos corresponden a las pre y post condiciones. Finalmente mismo quedan por identificar otras entidades intervinientes en cada caso de uso, lo que nos lleva a la siguiente etapa: el análisis.

3.2.2. Análisis

El análisis se enfoca en la producción de un modelo del sistema, llamado el modelo de análisis, que es correcto, completo, consistente y verificable. El análisis se diferencia de la obtención de requerimientos en que los desarrolladores se enfocan en la estructuración y formalización de los requerimientos obtenidos de los usuarios. (Bruegge, y otros, 2002).

El análisis es un refinamiento y estructuración de los requisitos capturados en la actividad anterior (requerimientos). El objetivo del análisis es conseguir una comprensión más precisa de los requisitos y una descripción de los mismos que sea fácil de entender (Jacobson00).

El análisis orientado a objetos es un método de análisis que examina los requisitos desde la perspectiva de las clases y objetos que se encuentran en el vocabulario del problema. (Booch, 2009)

En el modelo de análisis, se utilizan los estereotipos Entity-Control-Boundary (Simplificación del patrón Modelo-Vista-Controlador (university, 2014)) que son las clases: "boundary class", "control class", y "entity class" que al español serían: clase de interfaz, clase de control y clase entidad.

Los objetos de entidad representan la información persistente rastreada por el sistema. **Los objetos de frontera** representan la interacción entre los actores y el sistema. **Los objetos de control** representan las tareas realizadas por el usuario y soportadas por el sistema (Bruegge, y otros, 2002). Tiene ventajas como proporcionar a los desarrolladores una heurística simple para distinguir conceptos diferentes pero relacionados, el enfoque de tres tipos de objetos da como resultado objetos más pequeños y más especializados, conduce a modelos que son más adaptables al cambio.

Entonces en el modelo de análisis se procederá a describir las clases del análisis que actúan para la realización de los casos de uso en donde identificaremos en este contexto las clases que juegan un rol en el escenario seleccionado y lo iremos añadiendo al modelo del análisis final. Las clases del análisis luego evolucionan hacia otras clases más detalladas en el Modelo del Diseño. A continuación se procede a la realización de los casos de uso hacia el modelo de análisis en un contexto estático.

3.2.2.1. Modelo de análisis de préstamo de ejemplar

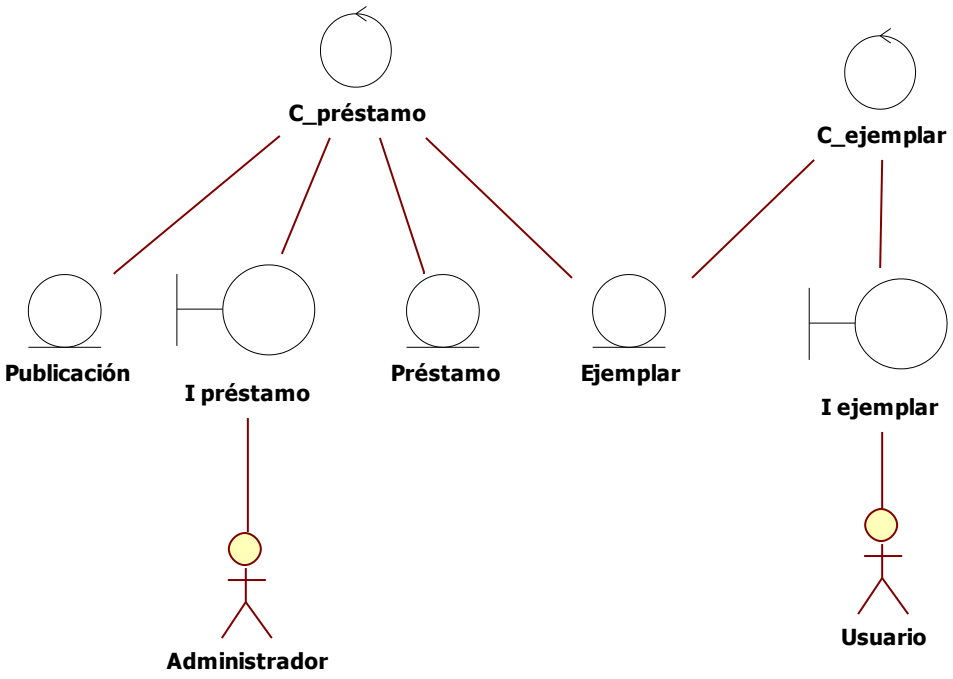


Ilustración 40: Modelo de análisis préstamo de ejemplar

Fuente: Elaboración propia.

3.2.2.2. Modelo de análisis de reserva de ejemplar

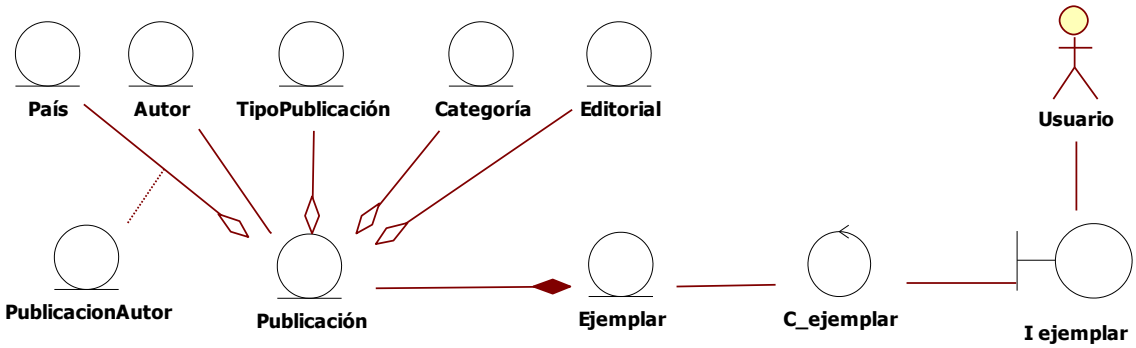


Ilustración 41: Modelo de análisis de reserva de ejemplar

Fuente: Elaboración propia.

3.2.2.3. Modelo de análisis de devolución de ejemplar

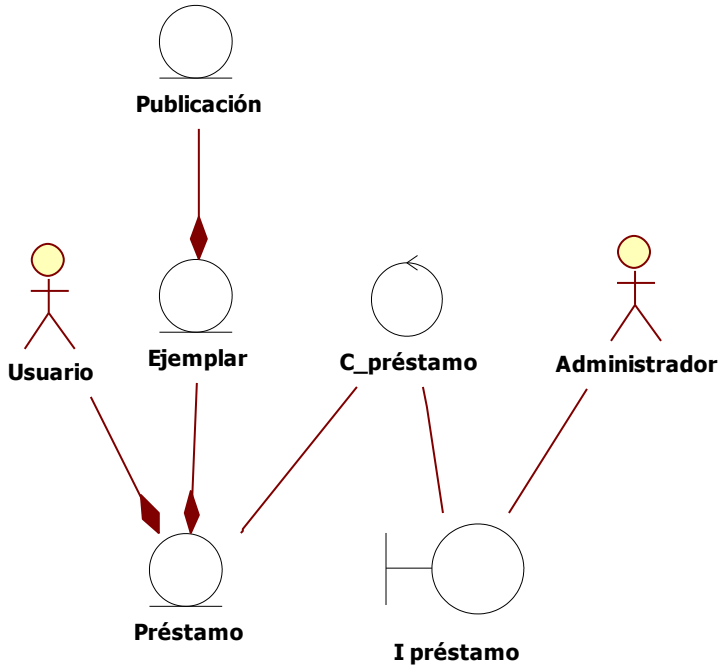


Ilustración 42: Modelo de análisis de devolución de ejemplar

Fuente: Elaboración propia.

3.2.2.4. Modelo de análisis de exportación de copia de seguridad

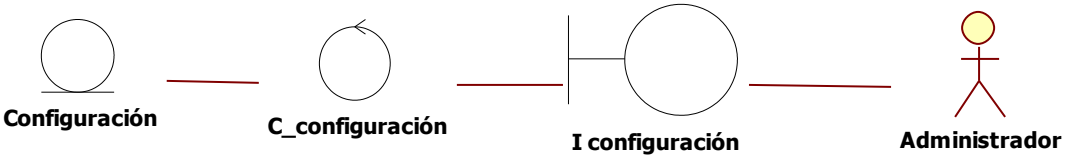


Ilustración 43: Modelos de análisis exportar copia de seguridad

Fuente: Elaboración propia.

3.2.2.5. Modelo de análisis de importación de copia de seguridad

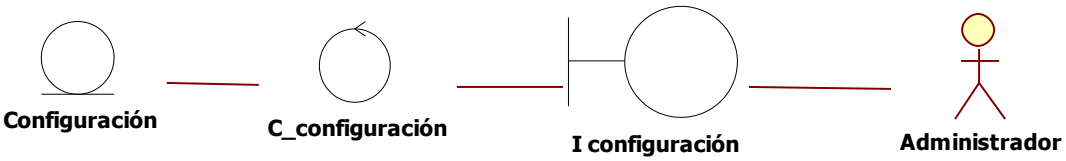


Ilustración 44: Modelo de análisis de importación de copia de seguridad

Fuente: Elaboración propia.

3.2.2.6. Modelo de análisis de administración de usuarios

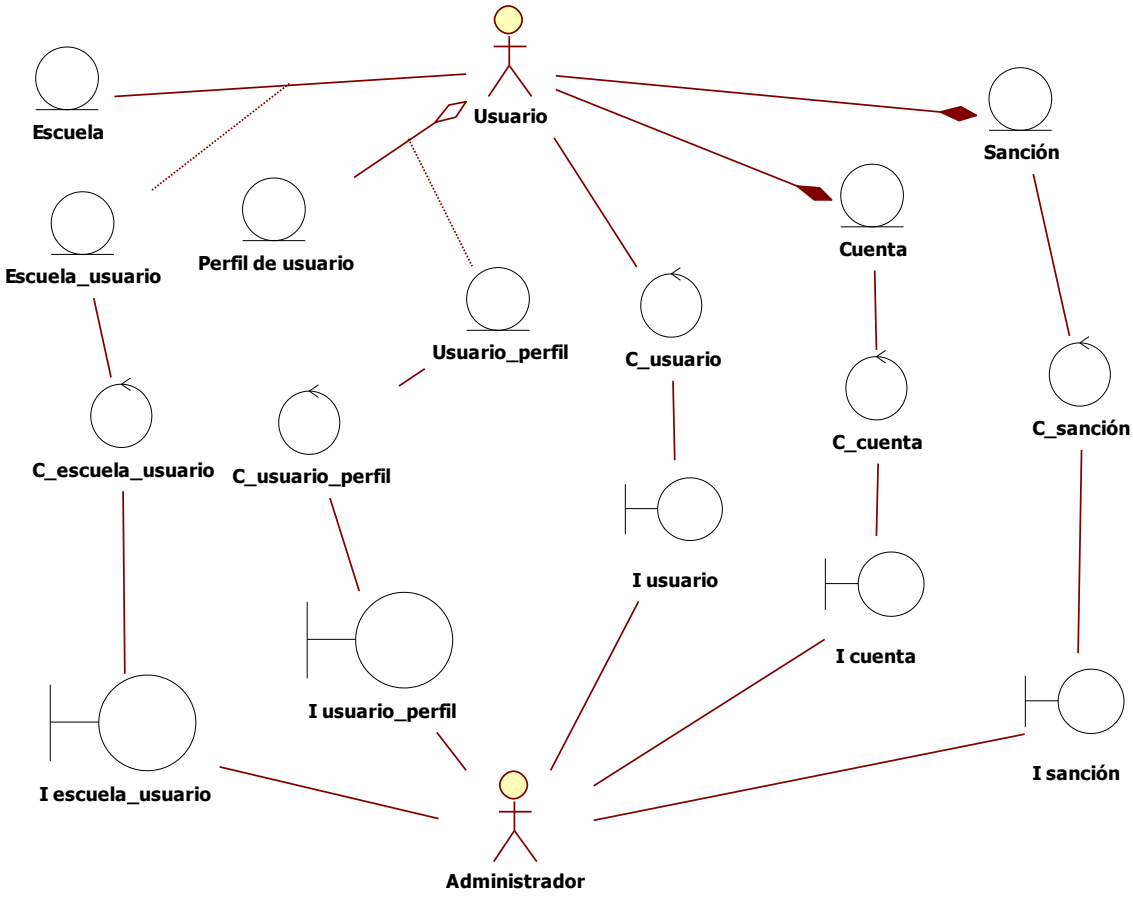


Ilustración 45: Modelo de análisis de administración de usuarios

Fuente: Elaboración propia.

3.2.2.7. Modelo de análisis de gestión de utilidades

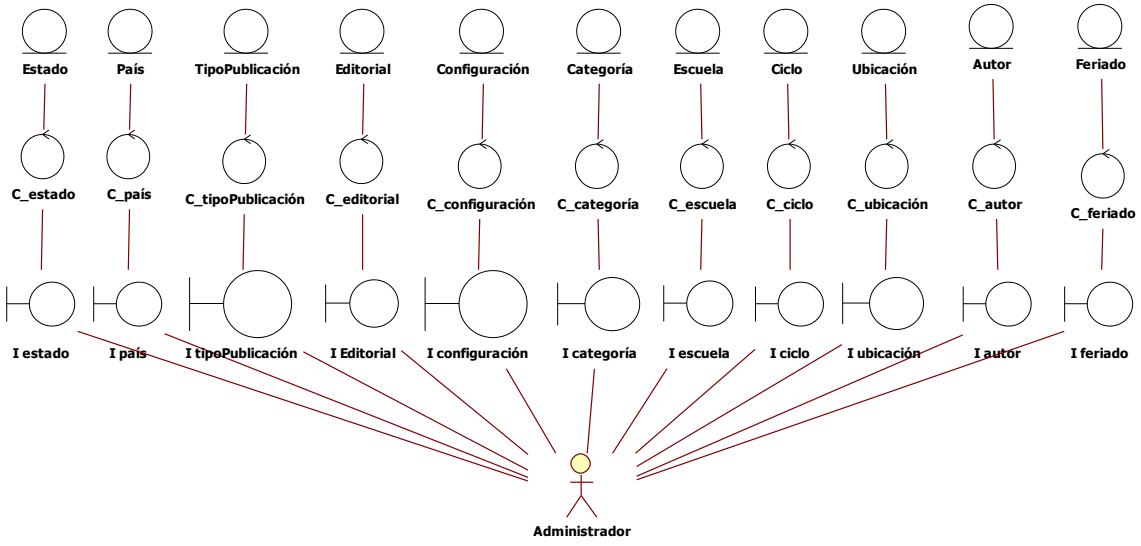


Ilustración 46: Modelo de análisis de gestión de utilidades

Fuente: Elaboración propia.

3.2.2.8. Modelo de análisis de administración de ejemplares

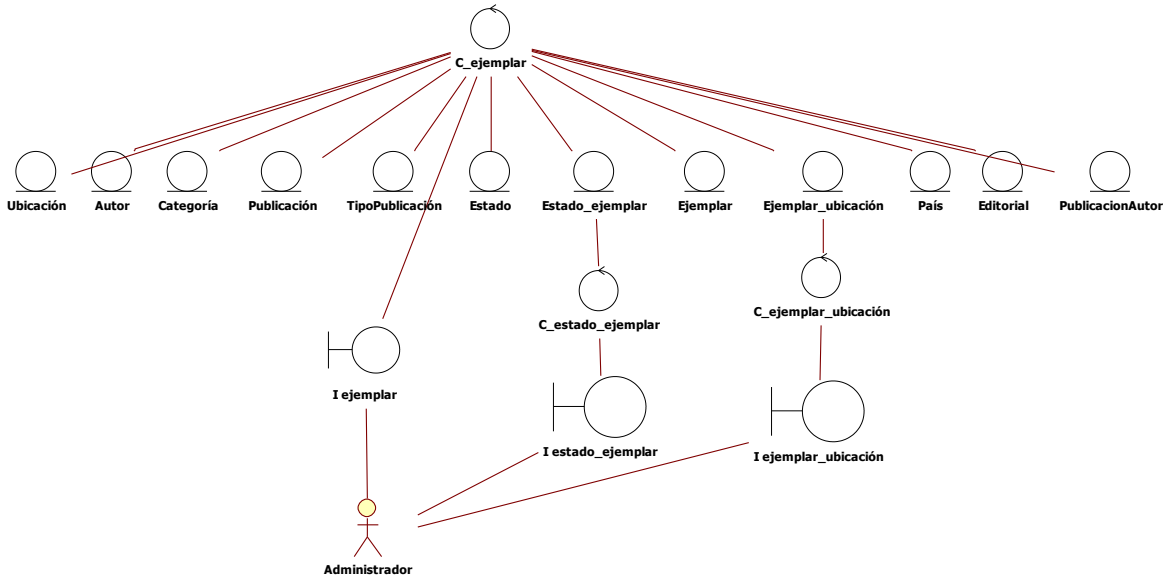


Ilustración 47: Modelo de análisis de administración de ejemplares

Fuente: Elaboración propia.

3.2.2.9. Modelo de análisis gestión de administradores

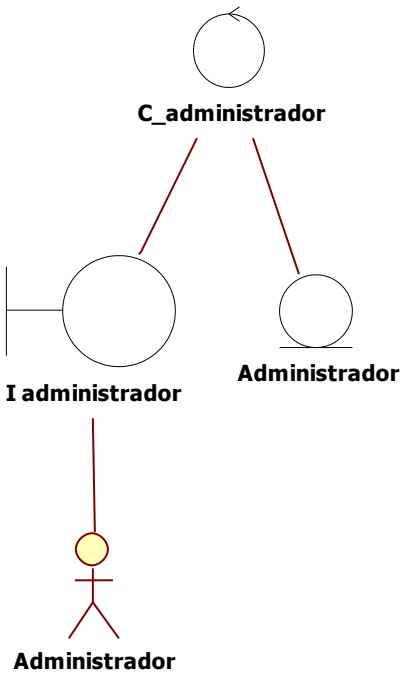


Ilustración 48: Modelo de análisis gestión de administradores

Fuente: Elaboración propia.

Hasta el momento se ha identificado la interacción de las clases del análisis que deben tener en los casos de uso en un contexto estático, pero no se ha identificado la interacción que deben tener dentro de los casos de uso, como se lleva o se ejecuta, por ejemplo sabemos el usuario realiza una búsqueda y también sabemos que el administrador realiza un registro de un usuario conocemos sus responsabilidades, pero, no sabemos cómo interactúan (los objetos) para llevar la realización del caso de uso, lo que nos lleva a desenvolver un modelo dinámico del sistema con las clases identificadas con diagramas de secuencias.

Un **diagrama de secuencia** une los casos de uso con los objetos. Muestra cómo se distribuye el comportamiento de un caso de uso (o escenario) entre sus objetos participantes. Los diagramas de secuencia no son, por lo general, un buen medio de comunicación con el usuario. Sin embargo, representan otro cambio de perspectiva y permiten que los desarrolladores encuentren objetos faltantes o áreas grises en la especificación del sistema (Bruegge, y otros, 2002)

Entonces vamos a apoyarnos en los modelos del análisis y en los casos de uso para describir la interacción de los objetos en un diagrama de secuencias.

3.2.2.10. Diagrama de secuencia préstamo de ejemplar

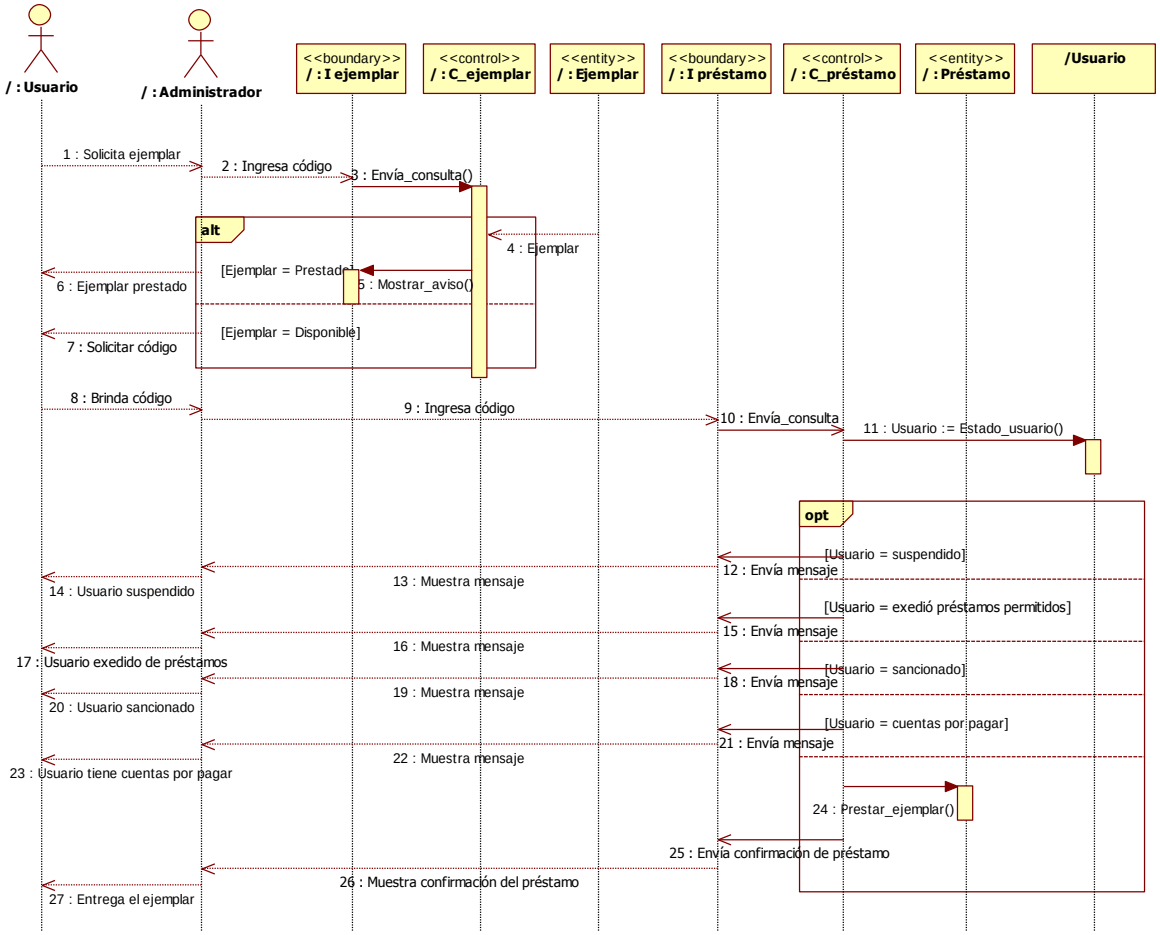


Ilustración 49: Diagrama de secuencia préstamo de ejemplar

Fuente: Elaboración propia.

3.2.2.11. Diagrama de secuencia de reserva de ejemplar

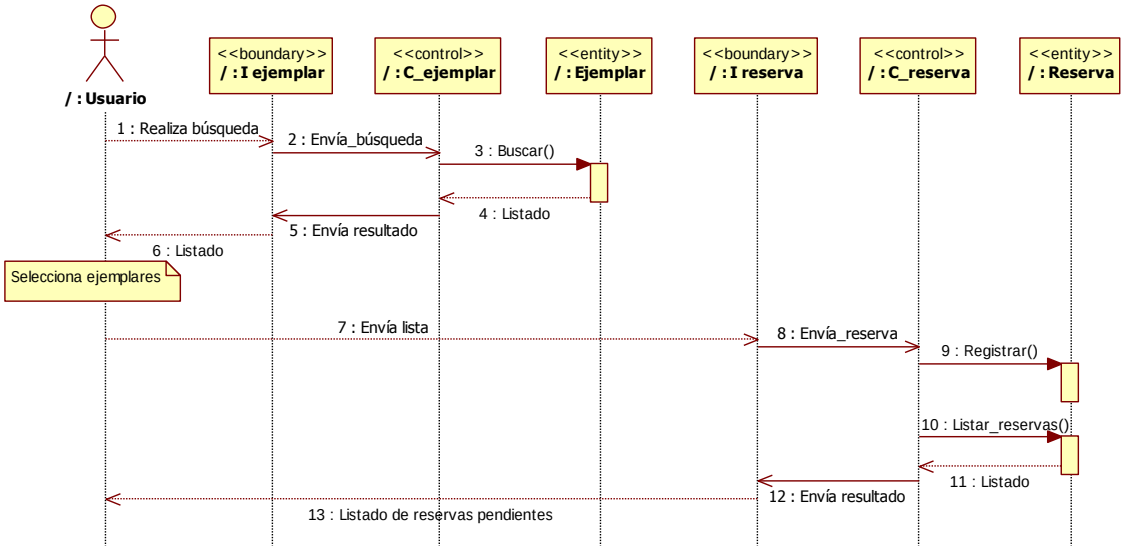


Ilustración 50: Diagrama de secuencia de reserva de ejemplar

Fuente: Elaboración propia.

3.2.2.12. Diagrama de secuencia de administración de usuario

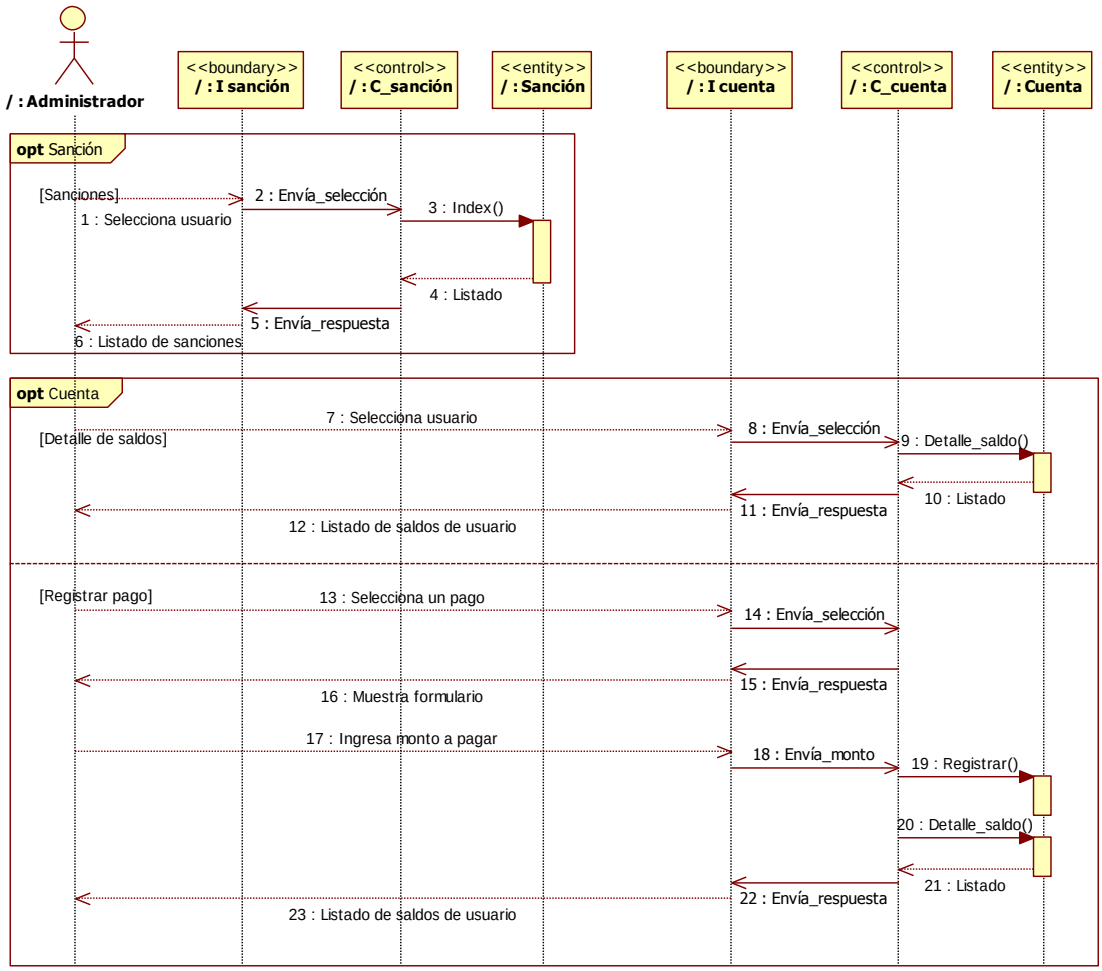


Ilustración 51: Diagrama de secuencia de administración de usuario: sanción y cuentas

Fuente: Elaboración propia.

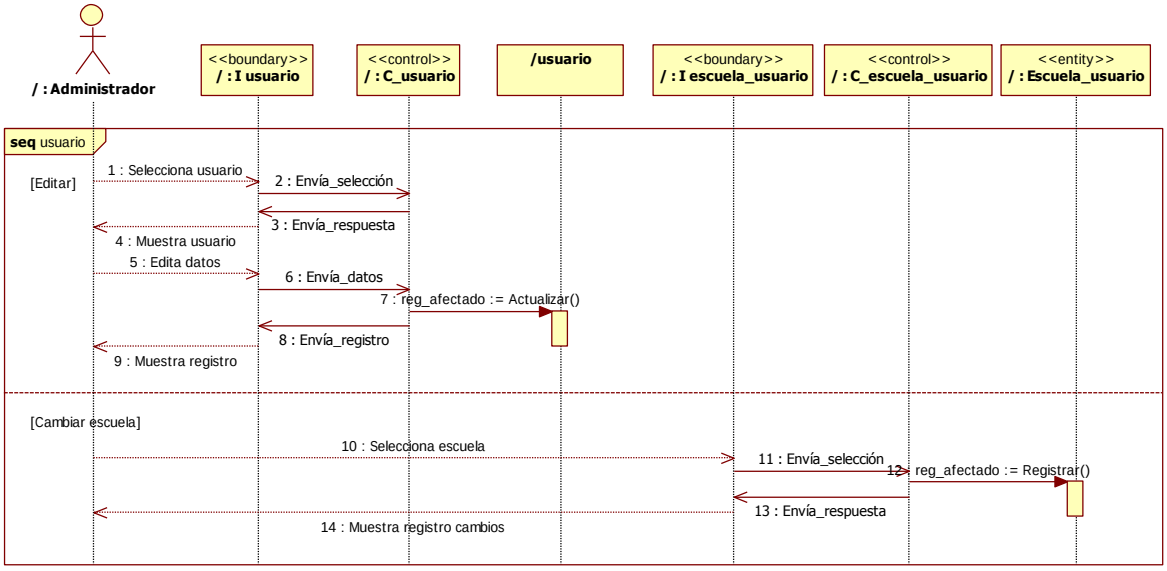


Ilustración 52: Diagrama de secuencia de Administración de usuarios edición y cambio de escuela

Fuente: Elaboración propia.

3.2.2.13. Diagrama de secuencia importar copia de seguridad

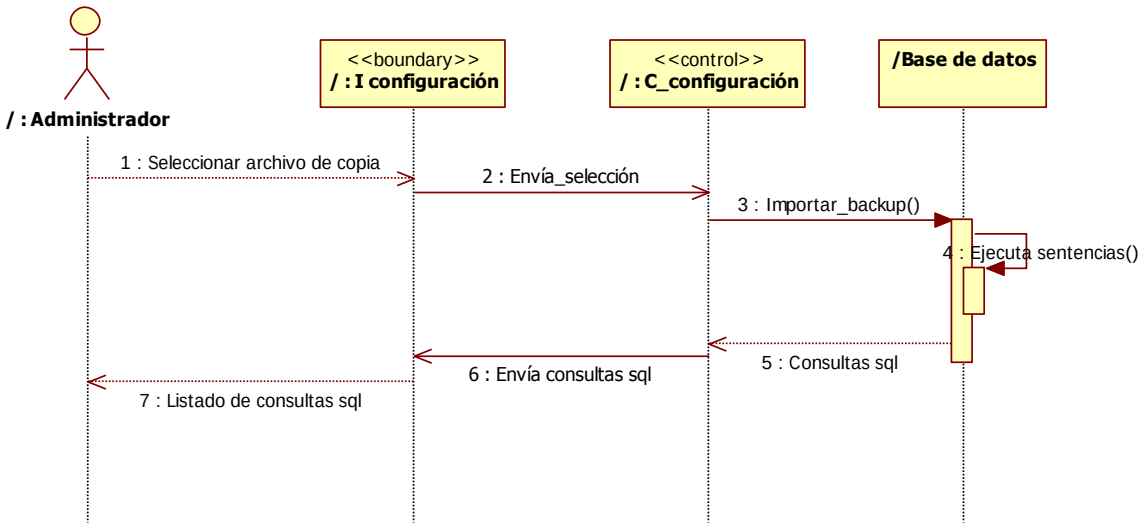


Ilustración 53: Diagrama de secuencia importar copia de seguridad

Fuente: Elaboración propia.

3.2.2.14. Diagrama de secuencia de una devolución

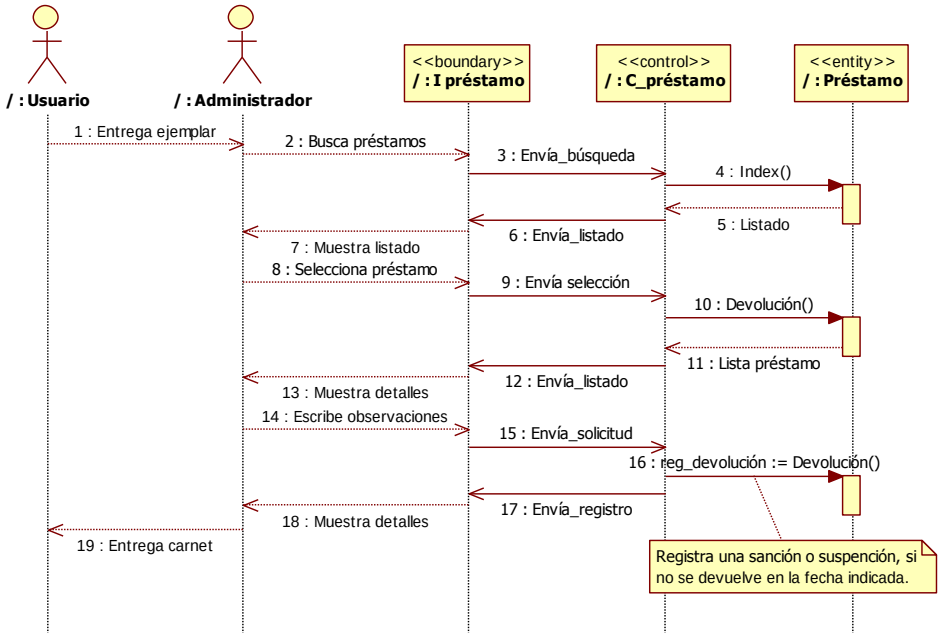


Ilustración 54: Diagrama de secuencia de una devolución

Fuente: Elaboración propia.

3.2.2.15. Diagrama de secuencia de importación de ejemplar

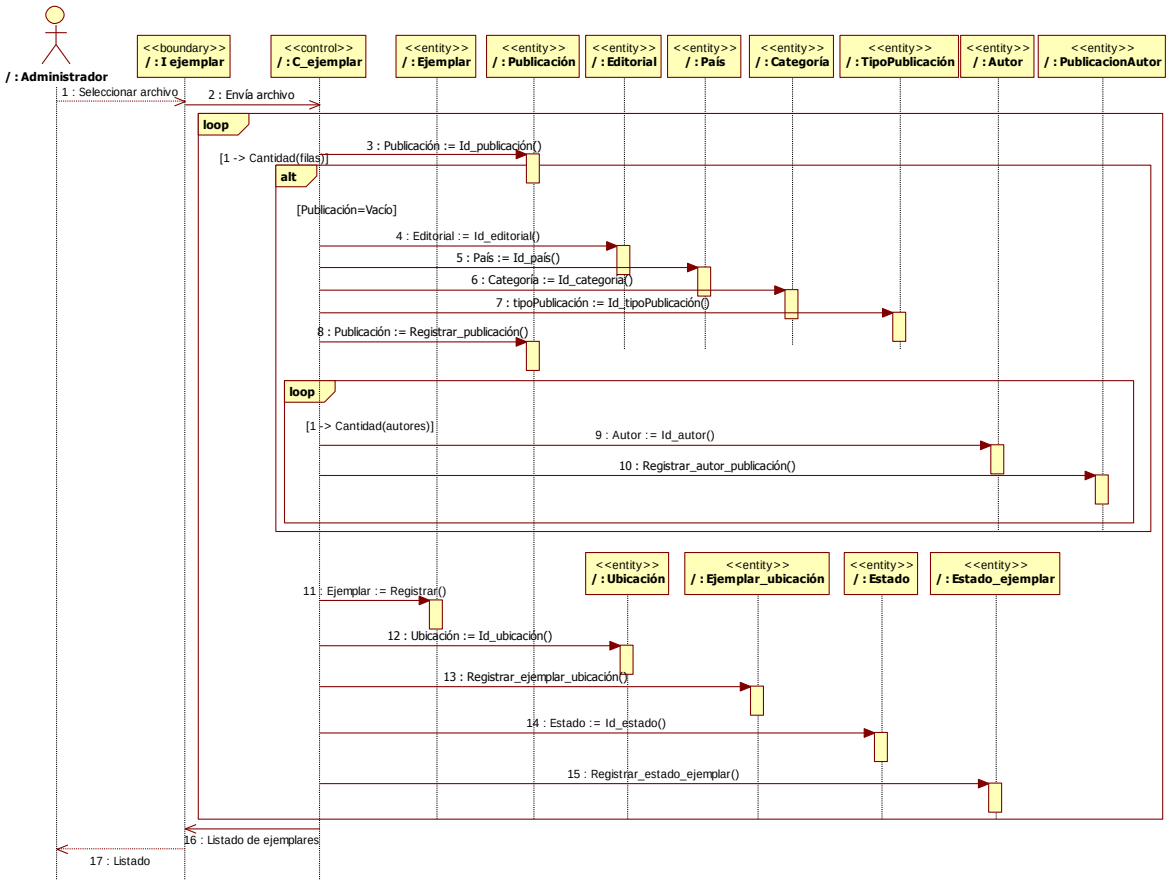


Ilustración 55: Diagrama de secuencia de importación de ejemplar

Fuente: Elaboración propia.

3.2.2.16. Diagrama de secuencia exportar copia de seguridad

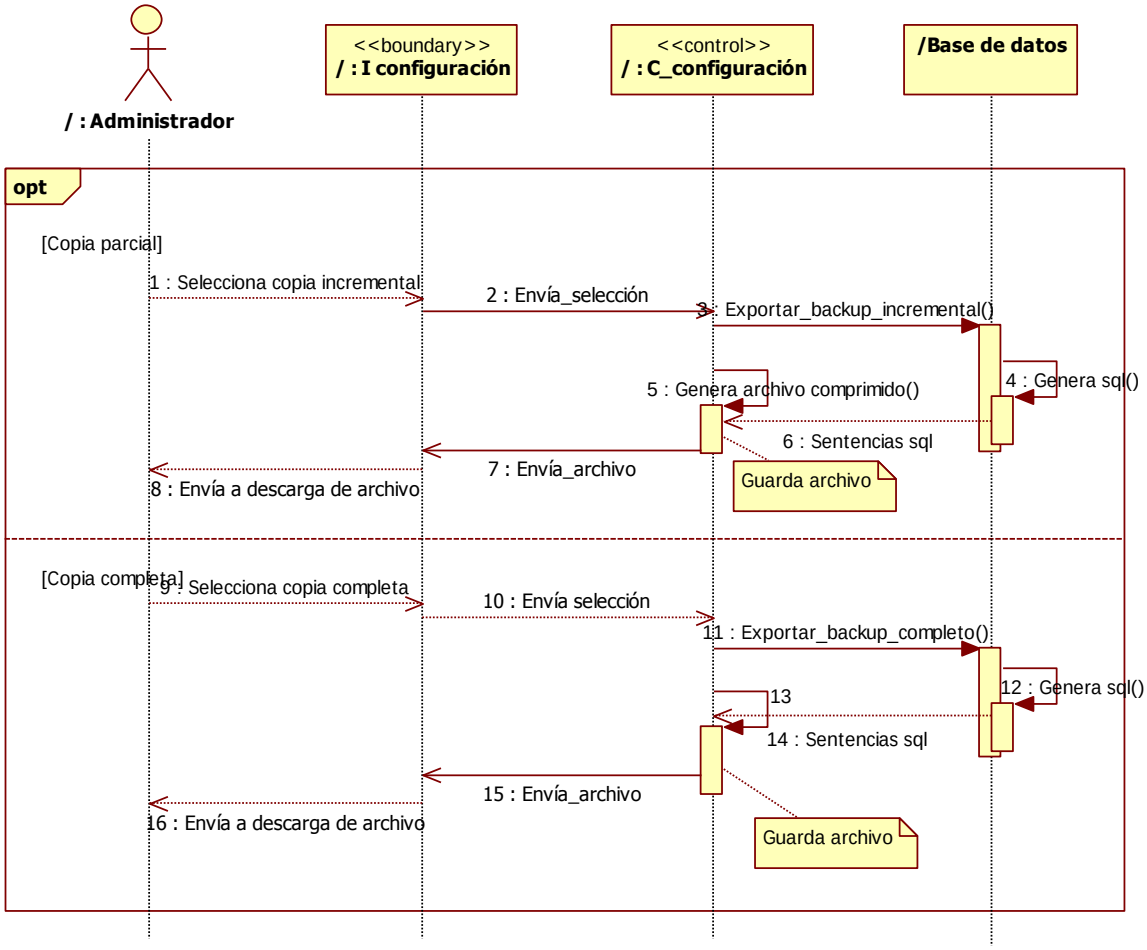


Ilustración 56: Diagrama de secuencia de exportar copia de seguridad

Fuente: Elaboración propia.

3.2.2.17. Diagrama de secuencia de gestión de utilidades

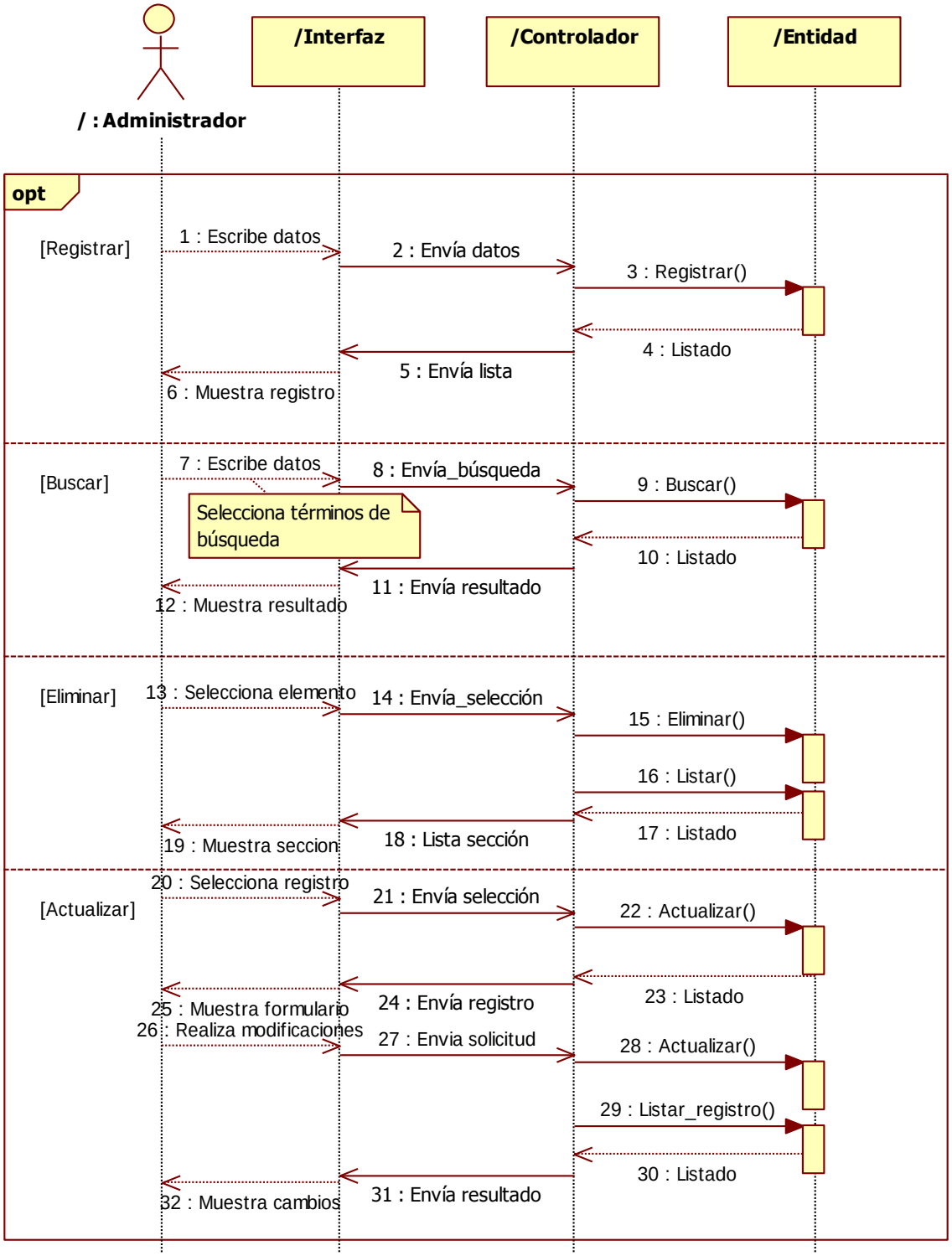


Ilustración 57: Diagrama de secuencia de gestión de utilidades

Fuente: Elaboración propia.

3.2.2.18. Diagrama de secuencia de administración de ejemplares

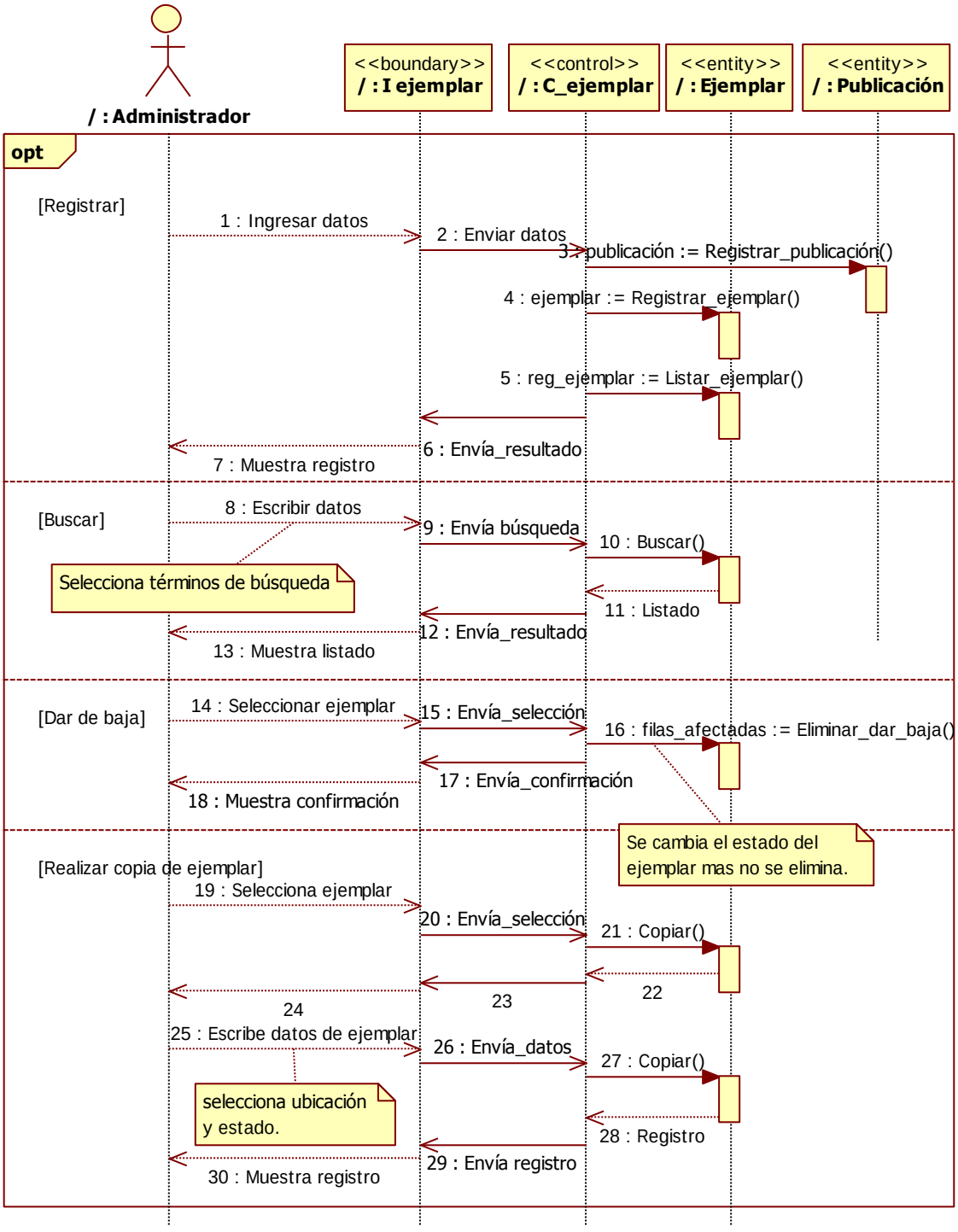


Ilustración 58: Diagrama de secuencia de administración de ejemplares

Fuente: Elaboración propia.

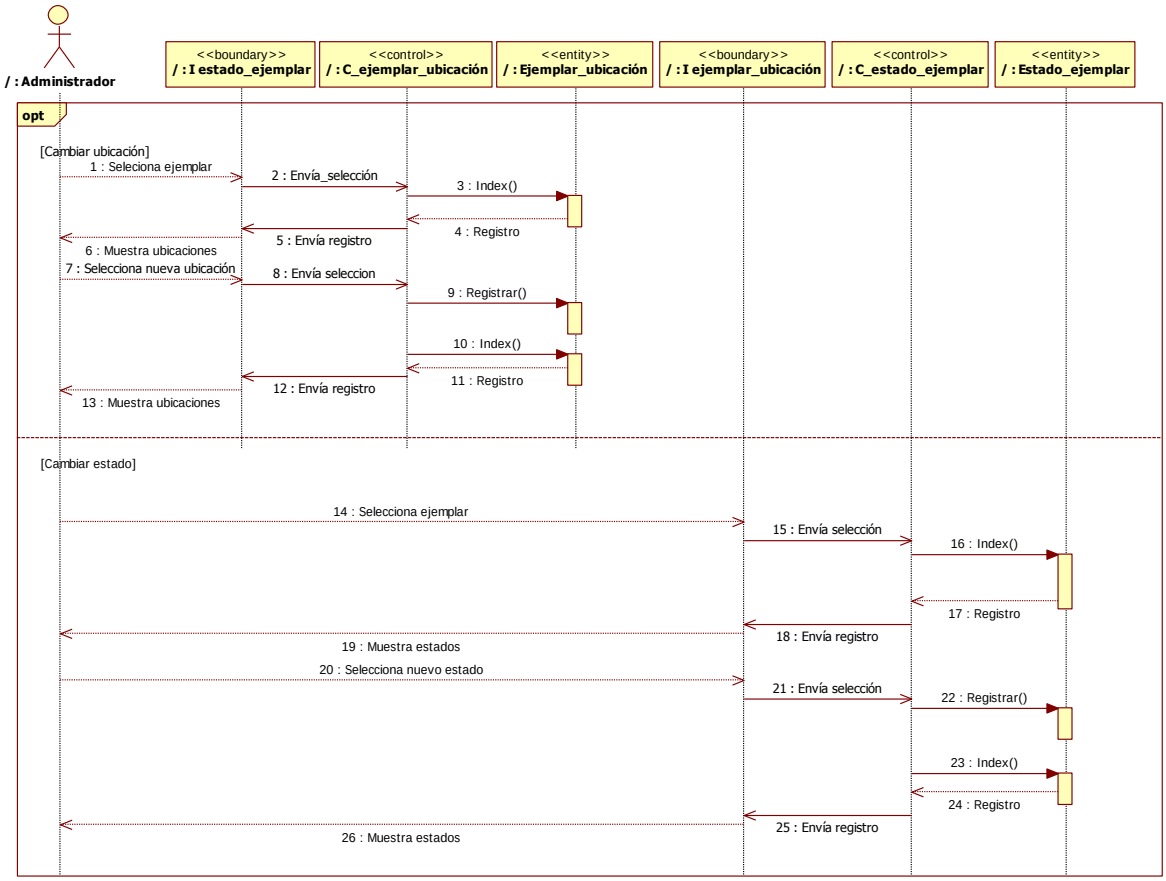


Ilustración 59: Diagrama de secuencia de administración de ejemplares

Fuente: Elaboración propia.

3.2.3. Diseño

El diseño de sistemas es la transformación del modelo de análisis en un modelo de diseño del sistema. Durante el diseño del sistema los desarrolladores definen los objetivos de diseño del proyecto y descomponen el sistema en subsistemas más pequeños que pueden ser realizados por equipos individuales. Los desarrolladores también seleccionan estrategias para la construcción del sistema como la plataforma de hardware y software en la que se ejecutará el sistema, la estrategia de almacenamiento de datos persistentes, el flujo de control global, la política de control de acceso y el manejo de condiciones de frontera (Bruegge, y otros, 2002).

El diseño de sistemas consiste en la transformación del modelo de análisis en el modelo de diseño, que toma en cuenta los requerimientos no funcionales y las restricciones descritas en el enunciado del problema y el documento de análisis de requerimientos.

La obtención de requerimientos con los casos de uso y el modelo de análisis se enfocó en la funcionalidad del sistema, ahora en el diseño nos enfocaremos en las estructuras de datos, componentes de software y hardware necesarios para implementarlo. Así mismo llenaremos el vacío entre la especificación, el análisis y el sistema que se entregará a los usuarios.

El diseño del sistema es el primer paso en esta dirección. El diseño del sistema da como resultado los siguientes productos:

- Una lista de objetivos de diseño que describe las cualidades del sistema que deben optimizar los desarrolladores

- Una arquitectura de software que describe la descomposición en subsistemas desde el punto de vista de responsabilidades del subsistema, dependencias entre subsistemas, correspondencia de los subsistemas con el hardware y decisiones de política principales, como el flujo de control, control de acceso y almacenamiento de datos.

3.2.3.1. Identificación de los objetivos de diseño

En esta sección nos centraremos en las cualidades que debe tener el sistema, y haremos uso de las especificaciones de pre y post condiciones de los casos de uso, así como las notas donde se señala algunas restricciones. Muchos de los objetivos de diseño pueden inferirse a partir de los requerimientos no funcionales, o inferirse de los requerimientos funcionales como forma de respaldar a las funciones realizadas.

Podemos identificar a la seguridad y esta a su vez se extiende a autenticación, autorización, manejo de excepciones. Pero existen otras emergentes como es la validación y registro de eventos.

- Autenticación
- Autorización
- Manejo de excepciones
- Validaciones
- Registro de eventos

Políticas de biblioteca

- Solo se podrán realizar préstamos dentro de ciclos académicos
- Las devoluciones pueden ser realizadas en cualquier momento.

Mantenibilidad

- Permitir al sistema realizar cambios de requerimientos en el menor tiempo posible.

3.2.3.2. Identificación de subsistemas

En la figura se muestra como el sistema está dividido en agrupaciones denominadas paquetes, estos son dependientes unos de otros, de manera tal que se pueden trabajar por separado al ser totalmente independientes de los demás paquetes.

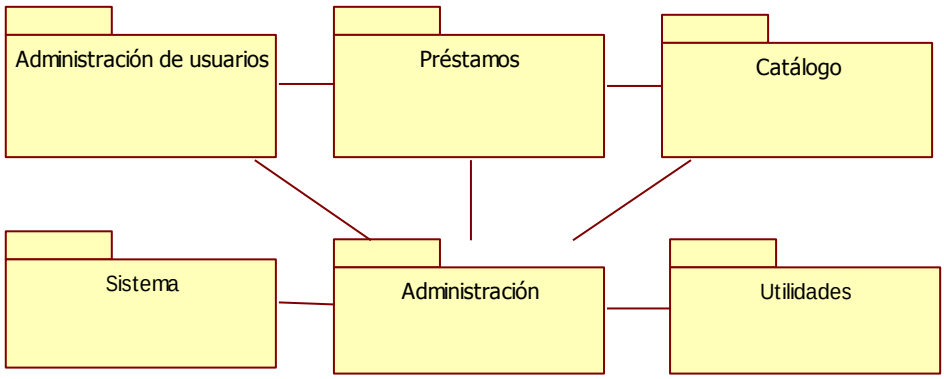


Ilustración 60: Identificación de subsistemas

Fuente: Elaboración propia.

- Paquete administración de usuarios: Es el paquete encargado de controlar la información que pertenece a los usuarios de la biblioteca, así como el ingreso, actualización, cambio de perfiles, cambio de escuelas, sanciones y cuentas.
- Paquete préstamos: Es el encargado de registrar los préstamos que realizan los usuarios en la biblioteca, registrar las devoluciones, generar las sanciones y suspensiones. Además contiene a la sección de reservas, este paquete es dependiente de los usuarios y catálogos.
- Paquete catálogo: Encargado de registrar ejemplares, copias, cambios de estados físicos, ubicaciones, y es totalmente independiente de los demás paquetes.
- Paquete de Sistemas: está encargado de llevar un registro de actividades, la configuración del sistema, feriados y control de ciclos. Ejerce influencia en los procesos de préstamos y es independiente en algunas secciones.
- Paquete de administración: Está encargado de administrar los accesos de los administradores de la biblioteca, es independiente.
- Paquete de utilidades: Encargado de realizar La gestión de diversos subsistemas con características similares en cuanto a realización de tareas.

1) Subsistemas de administración de usuarios

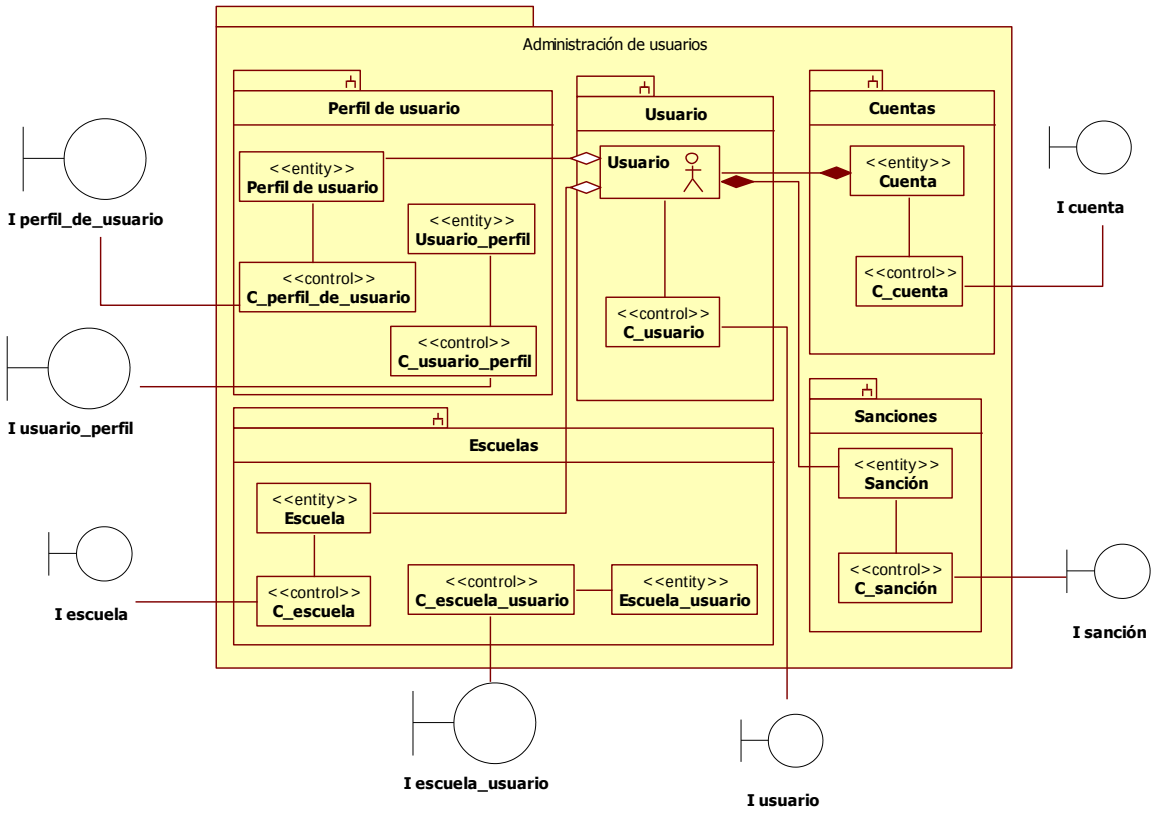


Ilustración 61: Subsistemas de administración de usuarios

Fuente: Elaboración propia.

En la ilustración se muestran los subsistemas del paquete de administración de usuarios. En el diagrama se puede observar la separación de subsistemas y sus relaciones con el subsistema usuario que es el principal subsistema.

2) Subsistema de préstamos

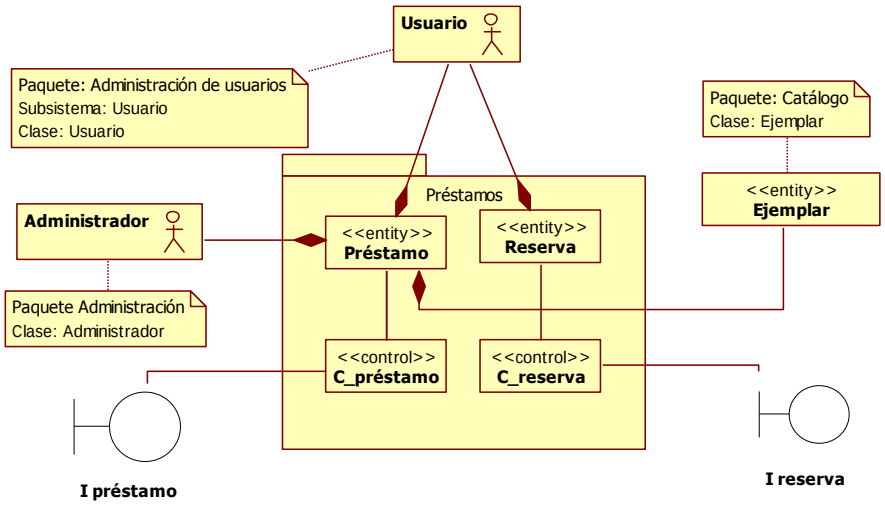


Ilustración 62: Subsistema de préstamos

Fuente: Elaboración propia.

En la ilustración número 61 se muestra la identificación de las entidades en el paquete de préstamos, así mismo se comprueba su dependencia de otras clases de entidad como son administrador, usuario y ejemplar.

3) Subsistemas de catálogo

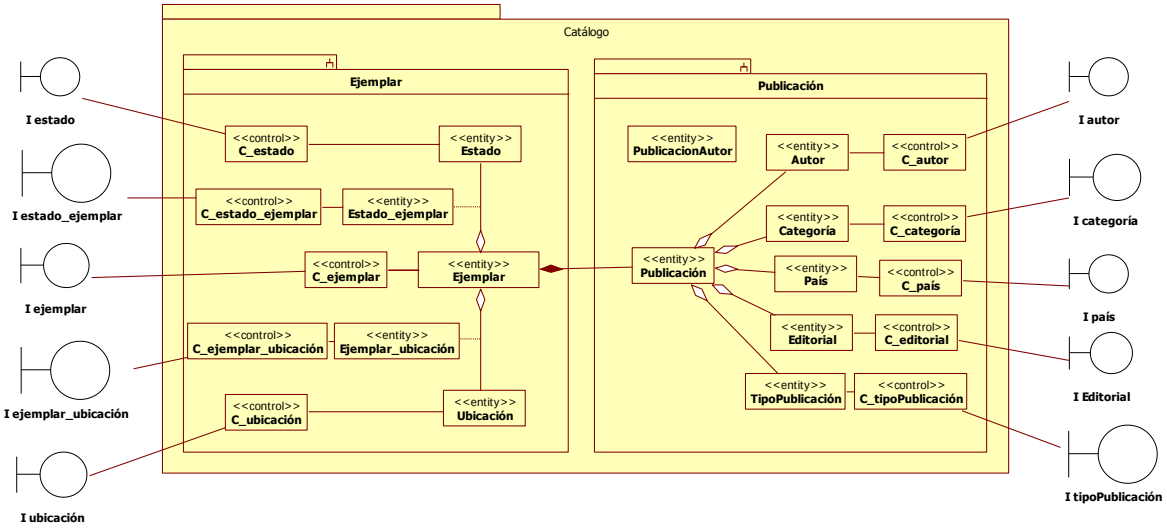


Ilustración 63: Subsistemas de catálogo

Fuente: Elaboración propia.

En el subsistema de catálogo se pueden identificar dos subsistemas debido a que existe una composición entre publicación a ejemplar, por ejemplo un ejemplar no podría existir si no se hace referencia a que publicación pertenece ya que el ejemplar solo contiene datos propios pero por si solo no daría mucha información sin hacer referencia a una publicación.

4) Subsistema de configuración

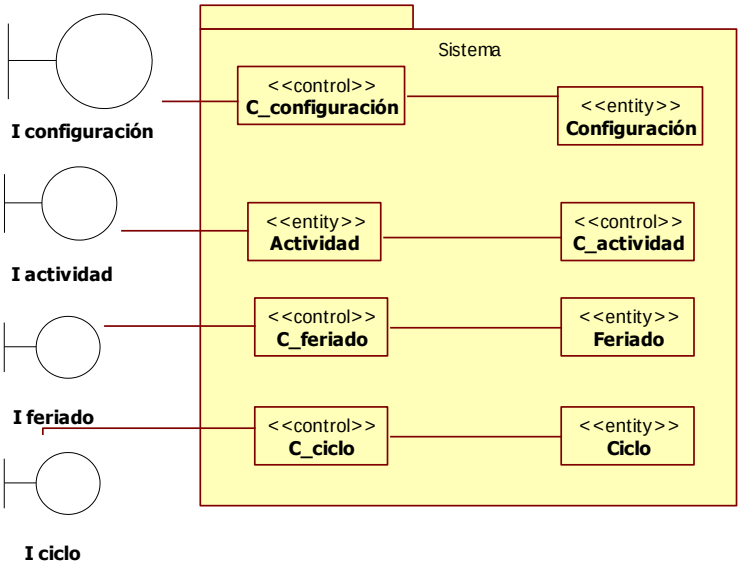


Ilustración 64: Subsistema de configuración

Fuente: Elaboración propia.

El paquete de configuración maneja entidades que tienen que ver con las variables del sistema, y con las restricciones de los requerimientos.

5) Subsistema de administración

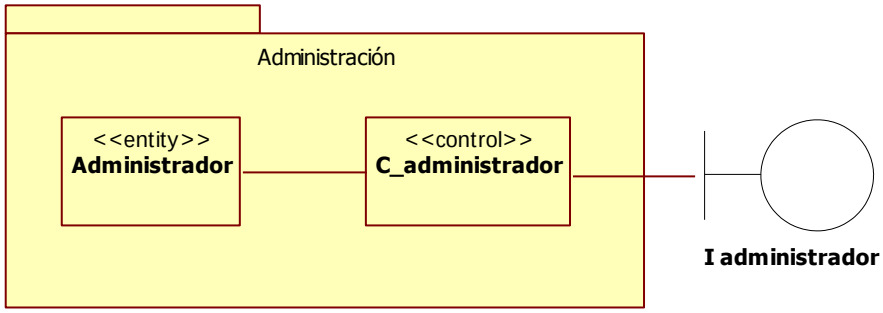


Ilustración 65: Subsistema de administración

Fuente: Elaboración propia.

En el diagrama se muestra la relación de la entidad, control e interfaz del paquete de administración que gestionara los administradores del sistema.

6) Subsistema de utilidades

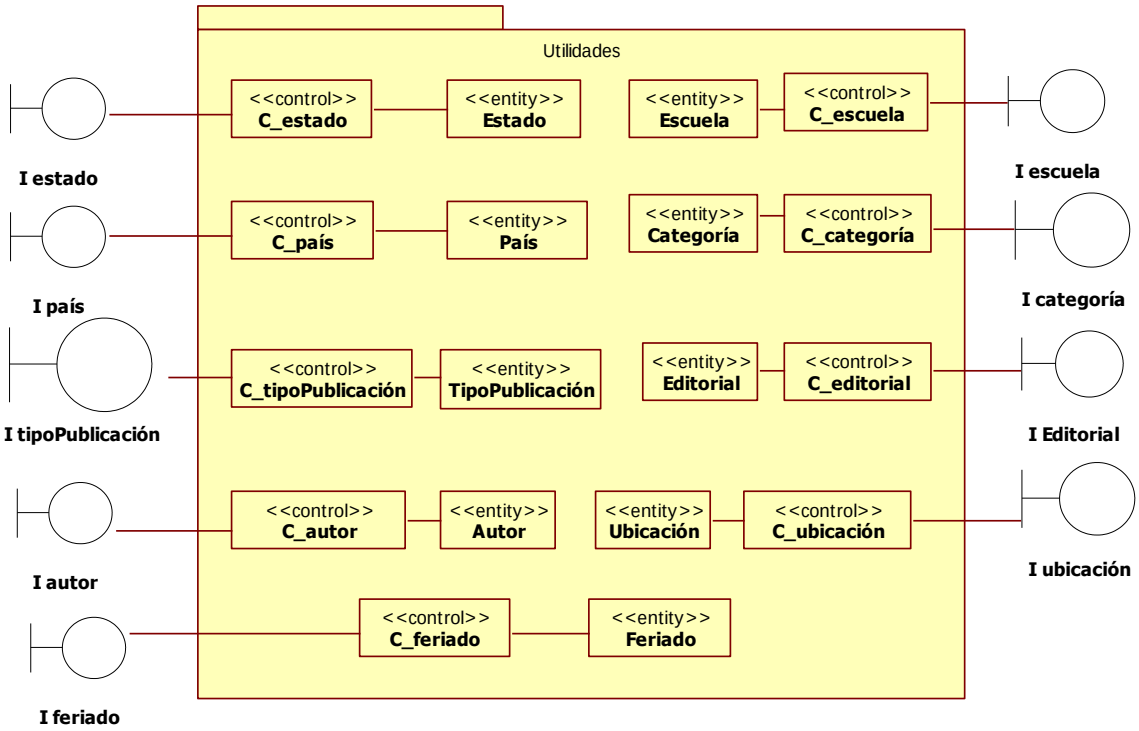


Ilustración 66: Subsistema de utilidades

Fuente: Elaboración propia.

El subsistema de utilidades ofrece accesos directos a la gestión de entidades con tareas de ejecución similares.

Hasta el momento se ha establecido el diseño de subsistemas con el objetivo de crear la máxima independencia entre subsistemas, pero no hemos definido la estructura del sistema visto desde un nivel más externo del sistema. En otras palabras la arquitectura.

3.2.3.3. Arquitectura de software

La arquitectura de software es importante porque afecta el desempeño y la potencia, así como la capacidad de distribución y mantenimiento de un sistema (Bosch, 2000). Como afirma Bosch, los componentes individuales implementan los requerimientos funcionales del sistema. Los requerimientos no funcionales dependen de la arquitectura del sistema, es decir, la forma en que dichos componentes se organizan y comunican. En muchos sistemas, los requerimientos no funcionales están también influidos por componentes individuales, pero no hay duda de que la arquitectura del sistema es la influencia dominante (Sommerville, 2011).

Entonces se ve necesario establecer la arquitectura del software para contar con una estructura con la cual se podrá desarrollar la implementación. Se elige la arquitectura del modelo vista controlador (citado en el marco teórico) debido a la facilidad de ver e interactuar con múltiples datos, además al no conocerse requerimientos futuros permite que se implementen fácilmente para la interacción y presentación. Nos brinda la ventaja de separar la lógica de la aplicación de la presentación con esto mínimo código ya que las vistas están separadas del código.

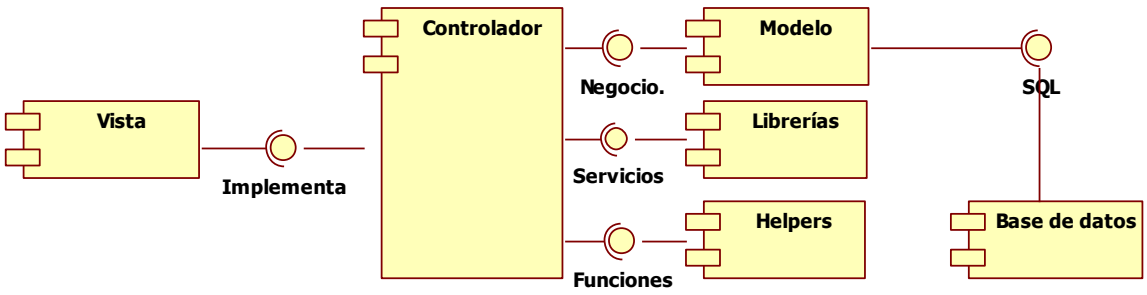


Ilustración 67: Diagrama de componentes Modelo Vista Controlador

Fuente: Elaboración propia.

En el diagrama de componentes se puede observar la interacción de los componentes modelo vista controlador, además de la colaboración de paquetes de librerías y helpers con el objeto de facilitar la elaboración del sistema.

Habiendo detallado la arquitectura del software en un entorno conceptual, nos queda por definir el soporte para el diseño del software, concentrarnos en el hardware, un diseño físico que de soporte a la arquitectura del diseño del software.

El diagrama de despliegue ilustra la implementación física de sistema, incluyendo el hardware, las relaciones entre hardware y el sistema en que se despliega (Kendall, 2005).

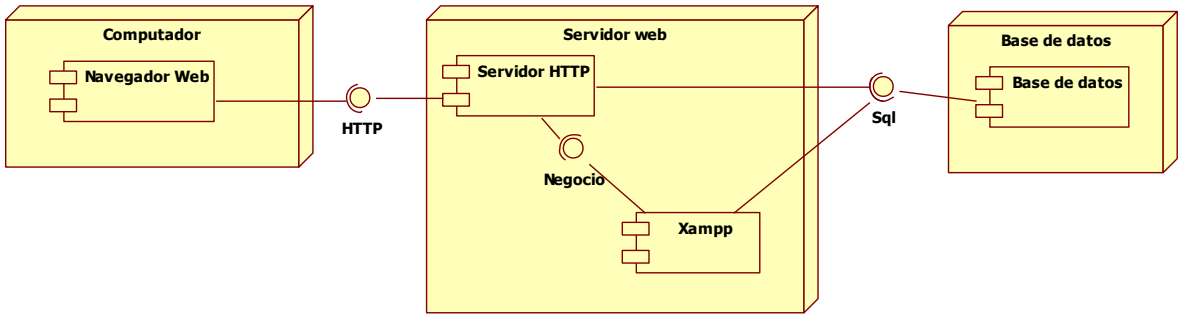


Ilustración 68: Diagrama de despliegue del sistema

Fuente: Elaboración propia.

El modelo de despliegue representa en otras palabras la correspondencia entre la arquitectura del software elegido y la arquitectura del sistema.

A continuación se presenta un diagrama de clases en el que se muestran tan sólo las clases de entidad realizadas en el análisis del sistema ya que representan la persistencia en la aplicación así como sus asociaciones, y de un modelo de datos que sería la transformación de clases al modelo relacional de base de datos donde se muestran las tablas que participan en las relaciones definidas en el diseño del software orientado a objetos.

3.2.3.4. Diagrama de clases

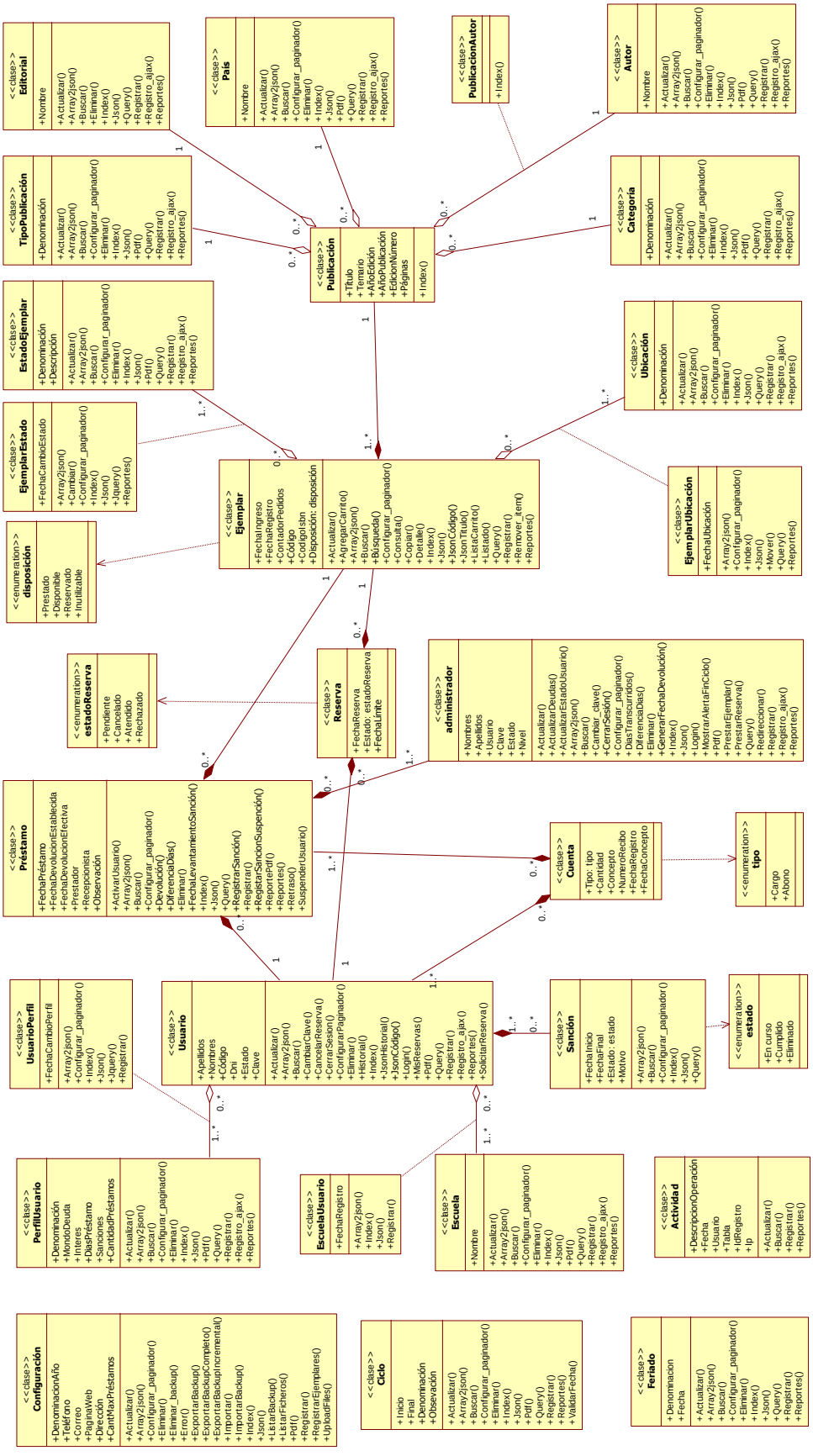


Ilustración 69: Diagrama de clases

Fuente: Elaboración propia.

3.2.3.5. Modelo de datos

Como definición una base de datos es un conjunto de datos relacionados entre sí. Por datos entendemos hechos conocidos que pueden registrarse u que tienen un significado implícito. (Elmasri, y otros, 1992).

Se han identificado las clases identidad que representan la persistencia en el modelo anterior y se hace su realización al modelo de entidad-relación quedando detallado en el anexo III.

3.2.3.6. Definición del control de acceso

En los casos de uso se observan ciertos roles definidos por lo tanto es necesario definir qué actores tienen acceso a partes del sistema y a que datos acceder. En el anexo Nro. I se establece un rol de accesos analizados según los requerimientos. Se usara este mismo anexo para el diseño orientado a aspectos.

3.2.4. Implementación

3.2.4.1. Elección del lenguaje de programación

PHP es un lenguaje de programación usado generalmente para la creación de contenido para sitios web. PHP es un acrónimo recurrente que significa "PHP Hypertext Pre-processor" (inicialmente PHP Tools, o, Personal Home Page Tools), y se trata de un lenguaje interpretado usado para la creación de aplicaciones para servidores, o creación de contenido dinámico para sitios web.

Su interpretación y ejecución se da en el servidor, en el cual se encuentra almacenado el script, y el cliente sólo recibe el resultado de la ejecución. Se elige este lenguaje por tratarse de un lenguaje gratuito por lo que se presenta como una alternativa de fácil acceso para todos., además que es un lenguaje multiplataforma, brinda soporte para diferentes manejadores de base de datos entre ellos Mysql, posee una amplia documentación, y permite técnicas de programación orientada a objetos.

3.2.4.2. Elección de un Framework para la arquitectura MVC

En la actualidad existen múltiples frameworks para Php que dan soporte a la arquitectura Modelo, Vista, Controlador. Pero dentro de ellos se eligió CodeIgniter debido a que es de fácil implementación, una curva de aprendizaje rápida y la codificación prácticamente se realiza

desde cero, esto lo puede volver muy básico, pero lo que se busca en esta investigación es desarrollar prácticamente desde cero.

3.2.4.3. Implementación de la capa de persistencia

En este caso, la capa de persistencia viene implementada como una base de datos en MySQL. Se ha elegido MySQL por tratarse de una base de datos libre, así como por su buena integración de su uso con PHP. En general, sistemas en los que la velocidad y el número de accesos concurrentes sea algo primordial. Para el diseño de la base de datos se ha utilizado MySQL Workbench Visual Database Designer.

3.2.4.4. Implementación de subsistemas

En esta etapa se desarrolló primeramente la implementación de subsistemas independientes dejando para después los subsistemas dependientes para una etapa posterior. Además de ello se implementó por una cuestión de organización solo los requerimientos funcionales dejando para el final los requerimientos no funcionales.

- Paquete de utilidades
- Paquete catálogo
- Paquete de Sistemas
- Paquete de administración

1) Paquete de utilidades

Para el desarrollo de este paquete se optó primero por codificar el modelo con operaciones básicas, luego el controlador y por último las vistas en este paquete la secuencia de operaciones se realizan de manera similar lo que no supondría inconvenientes puesto que existen una independencia entre ellos y cada uno se muestra por separado.

2) Paquete catálogo

En este paquete se hace uso de las interfaces del paquete de utilidades para tener acceso a operaciones de selección y registro, mas no de edición. Esto porque existen secciones como son: ubicaciones, países, estados físicos, tipos de ejemplar, autores, tipos de publicaciones, categorías, editoriales, escuelas, perfiles de usuarios. En el paquete de catálogo se identificaron dos subsistemas en el modelo de diseño el subsistema de ejemplares y subsistema de publicaciones. Primero se implementó el subsistema de publicaciones luego el subsistema de ejemplares.

3) Paquete de Sistemas

En este paquete se desarrolló las secciones identificadas en el modelo de diseño, se harán uso de estas secciones en los próximos paquetes.

4) Paquete de administración

En este paquete se desarrolló el único modelo de diseño identificado, se hará uso de este paquete en casi todas las demás secciones según mostradas en la ilustración de Identificación de subsistemas.

Siguiendo el desarrollo de los paquetes se desarrolló la implementación de paquetes dependientes:

5) Paquete administración de usuarios

Para la implementación de este paquete se hizo uso del paquete de utilidades en los accesos a escuela y perfiles de usuario para el momento de registrar. Se identificaron los subsistemas de perfil de usuarios, cuentas, escuelas, sanción y usuarios cada una realizaciones independientes y con relación de acceso de datos.

6) Paquete préstamos

En este paquete se desarrolló dos subsistemas: préstamos y reservas en ambos requieren de accesos a otras interfaces. En el primer caso del ejemplar (paquete catálogo), administrador (paquete de administración) y usuario (paquete de administración de usuarios). En el segundo caso del usuario (paquete de administración de usuarios).

3.2.4.5. Implementación de requerimientos no funcionales.

Según las reglas de acceso determinadas en el anexo Nro. I se implementaron estos requerimientos a lo largo de todas las clases requeridas así mismo también se implementó la autentificación, autorización, registro de eventos, sincronización, manejo de errores, transacción y validaciones.

Se implementó la autentificación en todas las clases de tipo controlador del sistema, agregándose estas propiedades antes del código funcional. Se implementó la autorización en todas las clases según el anexo Nro. III. El registro de eventos se realizó después de la ejecución del código funcional en las acciones de naturaleza de registro, actualización, inicios de sesión fallidos o con éxito. El manejo de excepciones en aquellas decisiones de entrada de datos a las funciones de los controladores. Y las validaciones en las operaciones de registro o actualización de datos.

Por ejemplo para registrar un nuevo ciclo se tiene que primeramente ver las restricciones de la política de la biblioteca de no registrar un ciclo nuevo si ya existe uno nuevo, también que toda la función deba estar dentro de un try-catch que corresponde al manejo de excepciones, luego una condición de autenticación caso contrario lo redirige a que inicie sesión, seguidamente una restricción de acceso a la función verificando el tipo de usuario y su estado, si todo marcha bien se ingresará a la condicional de mostrar la vista, dentro de ella se observan las validaciones de datos a ingresar puesto que es registro debe cumplir con los tipos de datos que se insertaran en la base de datos, en caso contrario solo se mostrará el formulario para ingresar datos, si se cumplen todas las condiciones se llegará a la sección del registro del ciclo académico. Finalmente una posterior condición es que cada operación que tenga que ver con registro o modificación tiene que ser registrada en la base de datos.

3.3. Caso 2: Diseño de un software orientado a aspectos

Para el desarrollo del siguiente caso, se optó por seguir el Desarrollo de software orientado a aspectos con casos de uso. Porque brinda una amplia documentación acerca de todo el proceso de desarrollo de software orientado a aspectos con un enfoque práctico, promueve la separacion de competencias desde tempranos inicios del software dando explicaciones sobre la forma de abordarlos y ademas específica la tarea a realizar en cada etapa. (Jacobson, y otros, 2004) Expresan que se necesita un enfoque integral para el desarrollo de sistemas de software con los aspectos de los requisitos, el análisis y el diseño, hasta la implementación y prueba.

3.3.1. Requerimientos

Según explican (Jacobson, y otros, 2004): Es bien sabido que la orientación a aspectos ayuda a modularizar incumbencias transversales durante la implementación, pero hay una necesidad de modularizar incumbencias transversales mucho antes, incluso durante los requisitos. Los casos de uso son una excelente técnica para este propósito.

Los enfoques de modularidad convencionales como las clases y componentes que no pueden mantener las incumbencias transversales separadas. Para resolver este problema, se necesitan dos cosas: una técnica de separación para mantener las incumbencias transversales independientes y un mecanismo de composición para componer estas incumbencias como parte del sistema deseado.

Son muchos los trabajos acerca de cómo identificar incumbencias transversales desde la identificación textual: (Rago, y otros, 2008), (Rashid 2003) hasta herramientas de ayuda para su identificación: (Lorena, y otros, 2013), (Weston, y otros, 2013).

(Jacobson, y otros, 2004) En casos de uso para aspectos, establecen reglas para el modelamiento de escenarios con aspectos, así como el tipo de relación entre un aspecto y un caso de uso a través de las relaciones *extends*, *include* y generalizaciones.

Según este autor en esta fase se identifican dos categorías de casos de uso: uno de aplicación y otro de infraestructura. Los casos de uso de aplicación representan los requerimientos funcionales del sistema y los casos de uso de infraestructura representan las cualidades del sistema que dan soporte para cada caso de uso de aplicación.

Los términos aspectuales normalmente describen los atributos de calidad de un sistema como el de seguridad y rendimiento, descripciones de impacto de la influencia de uno de los requisitos sobre otros requisitos, y dispersas o descripción de los requisitos que aparecen

repetidos en varios lugares en todo el documento de requerimientos a menudo se consideran indicadores de las preocupaciones transversales en los requisitos.

Esto implica que la programación orientada a aspectos toma en cuenta tanto a requerimientos funcionales y no funcionales desde tempranas etapas del ciclo de vida de desarrollo de software, manteniéndolos separados durante las demás etapas.

3.3.1.1. Administración de usuarios

Para la administración de usuarios será necesario que el administrador que tenga una sesión iniciada en el sistema además que esté habilitado, el administrador podrá buscar usuarios, cambiar sus perfiles, hacer cambios de escuelas, registrar sanciones, registrar un abono. Para todas estas actividades debe de estar autorizado. En el registro de usuarios se debe realizar una validación de los datos que se ingresa cuidando que no se ingresen datos repetidos Para cada acción de registro y actualización se llevará un registro de actividad en la cual indique la acción realizada o los cambios realizados y el usuario que realizo el cambio.

Identificación de la infraestructura del sistema:

- El administrador debe tener una sesión iniciada
- El administrador debe estar Autorizado
- Registro de actividad en registro y actualización
- validación de los datos en el registro de usuarios

Identificación de requerimientos de la aplicación:

- Buscar usuarios
- Cambiar sus perfiles
- Hacer cambios de escuelas
- Registrar pagos
- Registrar sanciones

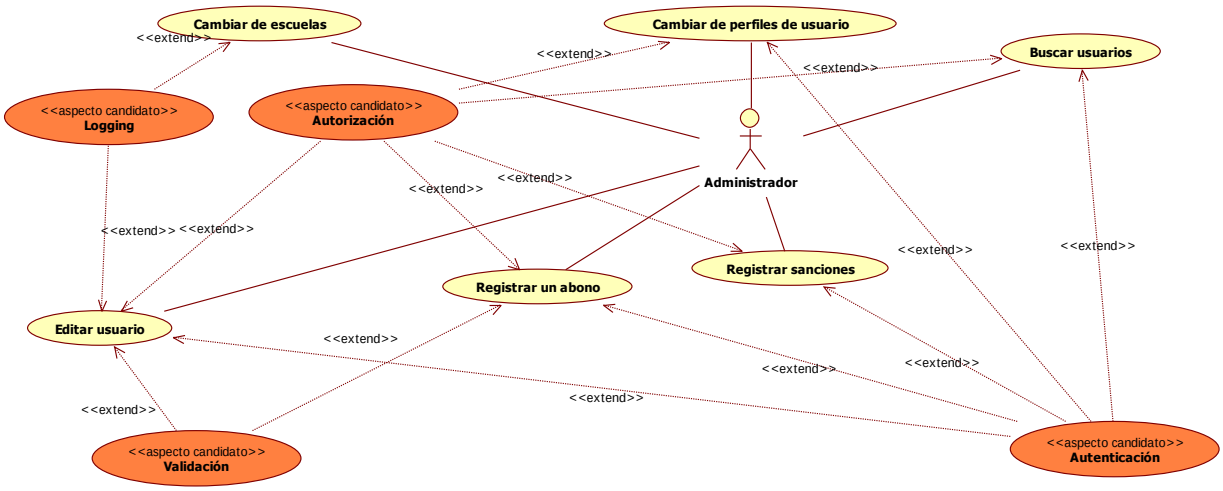


Ilustración 70: Caso de uso de infraestructura y requerimientos de administración de usuarios

Fuente: Elaboración propia.

3.3.1.2. Administración de ejemplares

Para este caso de uso el administrador debe haber iniciado una sesión y además estar autorizado para todo lo siguiente: realizar funciones de registrar ingresando los datos del ejemplar para ello es necesario verificar los datos antes de registrarlo, también pueden realizar búsquedas por cada uno de los campos del ejemplar, registrar unas copias de ejemplares cuando ya existe uno en el sistema y solo es necesario extender un código distinto, con su ubicación y estado, en otras veces se podrá dar de baja a un ejemplar y no se podrá volver a usar ni aparecerá en los resultados de búsqueda de los usuarios, cuando se realiza un registro, bajas o actualización el sistema realizará un registro de la acción así como de quien lo realizó.

Identificación de la infraestructura del sistema:

- Verificar los datos antes de registrarlo
- Ante registros, bajas o actualización el sistema realizará un registro de la acción así como de quien lo realizó.
- No se podrá mostrar en los resultados cuando se de baja a un ejemplar.
- Haber iniciado una sesión y además estar autorizado

Identificación de requerimientos de la aplicación:

- Registrar ejemplares
- Dar de bajas
- Reportes de ejemplares
- Buscar ejemplares
- Realizar copias de ejemplares

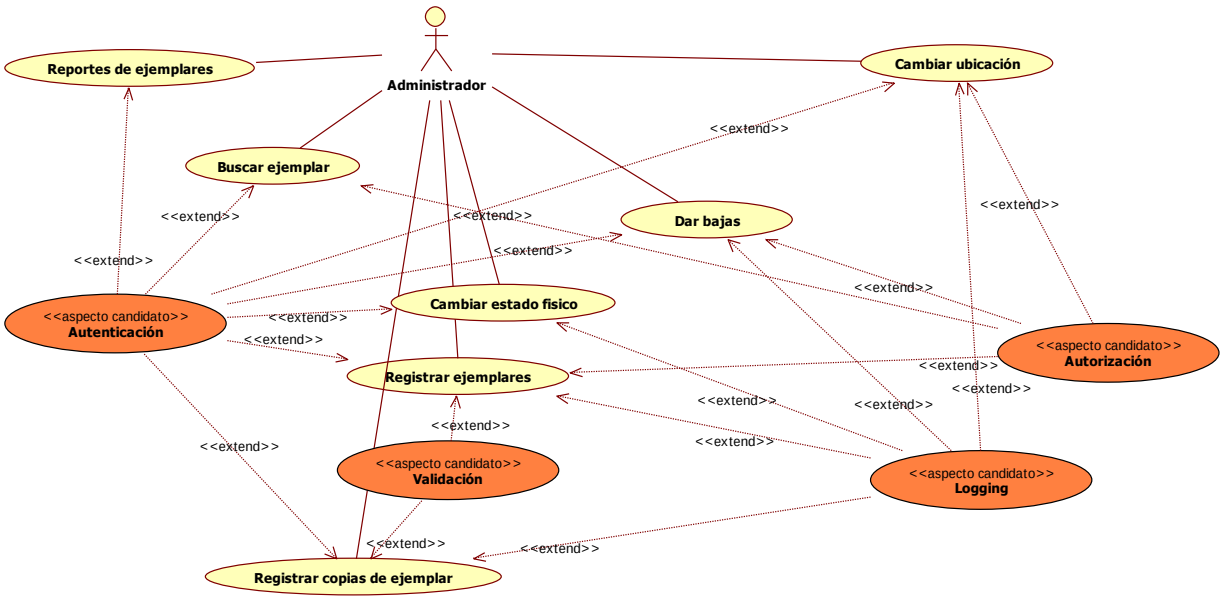


Ilustración 71: Caso de uso de infraestructura y requerimientos de administración de ejemplares
Fuente: Elaboración propia.

El administrador podrá también realizar cambios de ubicación física del ejemplar dentro del ámbito de las instalaciones de la biblioteca, así mismo también podrá realizar un cambio de estado físico donde se incluirá la fecha del cambio y la denominación del estado. Para realizar estas acciones es necesario llevar un registro de la actividad de cambio realizado por el usuario. Para los casos de cambios es necesario que el administrador haya iniciado sesión en el sistema y esté autorizado para realizar dichas acciones.

Identificación de la infraestructura del sistema:

- Llevar un registro de la actividad.
- Es necesario que se haya iniciado sesión en el sistema y se esté autorizado para realizar dichas acciones.

Identificación de requerimientos de la aplicación:

- Cambiar estados físicos
- Cambiar ubicaciones del ejemplar
- Ver registro de ubicaciones
- Ver registro de cambios de estados físicos

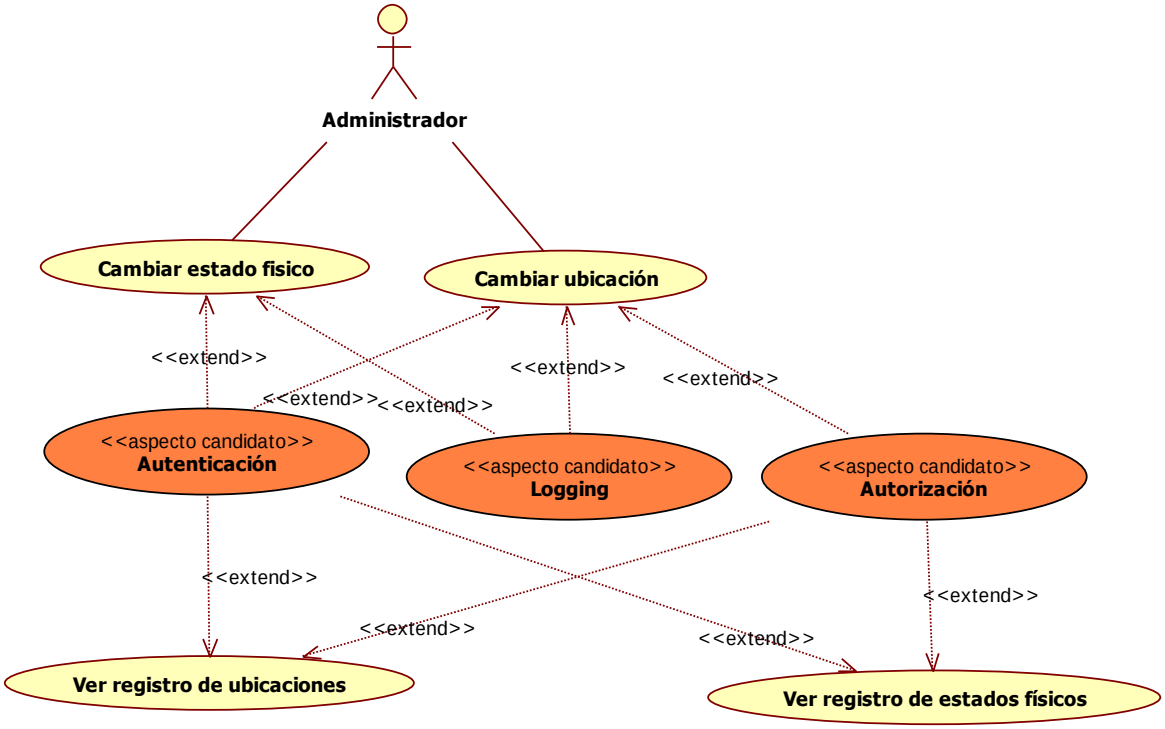


Ilustración 72: Caso de uso de infraestructura y requerimientos de administración ubicaciones y estados físicos.

Fuente: Elaboración propia.

El administrador podrá importar una lista de ejemplares de un formato en Excel, para ello tendrá que estar autorizado, activo y haber realizado una sesión en el sistema, el sistema validará el archivo y los datos de la lista de ejemplares en el archivo. Realizada la importación mostrará los resultados y hará un registro de la actividad realizada y el administrador que lo hizo.

Identificación de la infraestructura del sistema:

- El administrador tendrá que haber iniciado sesión
- El administrador tendrá que estar activo y autorizado
- Se registrara la actividad
- Se validaran los datos del archivo

Identificación de requerimientos de la aplicación:

- Seleccionar el archivo
- Enviar el archivo

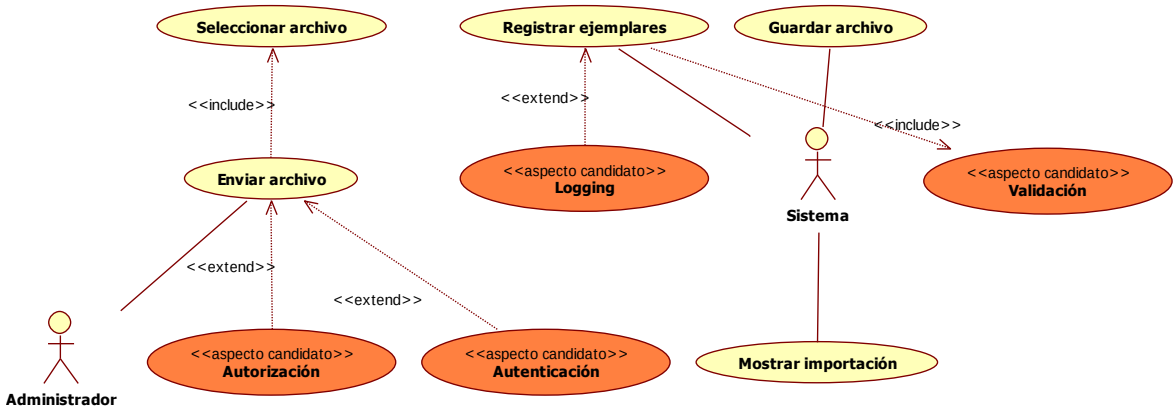


Ilustración 73: Caso de uso de infraestructura y requerimientos de importación de ejemplares

Fuente: Elaboración propia.

3.3.1.3. Gestión de administrador

El administrador podrá realizar un registro de administradores siempre y cuando este activo, haya iniciado sesión y tenga los permisos para poder realizar la operación, buscar en el sistema. El sistema tendrá que validar los datos ingresados y no exista registros duplicados, también podrá modificar datos como cambios de nombres, apellidos, DNI y demás campos. Para las acciones de registro y modificación se tendrá que llevar un registro de la actividad realizada indicando la operación, el registro afectado, la dirección IP y el nombre del usuario que realizo la acción.

Identificación de la infraestructura del sistema:

- El administrador debe de estar activo
- Validación de datos ingresados
- Registro de la actividad
- El administrador tiene que tener una sesión iniciada
- Tener permisos para realizar la operación

Identificación de requerimientos de la aplicación:

- Registro de administradores
- Buscar en el sistema
- Modificar datos de administradores

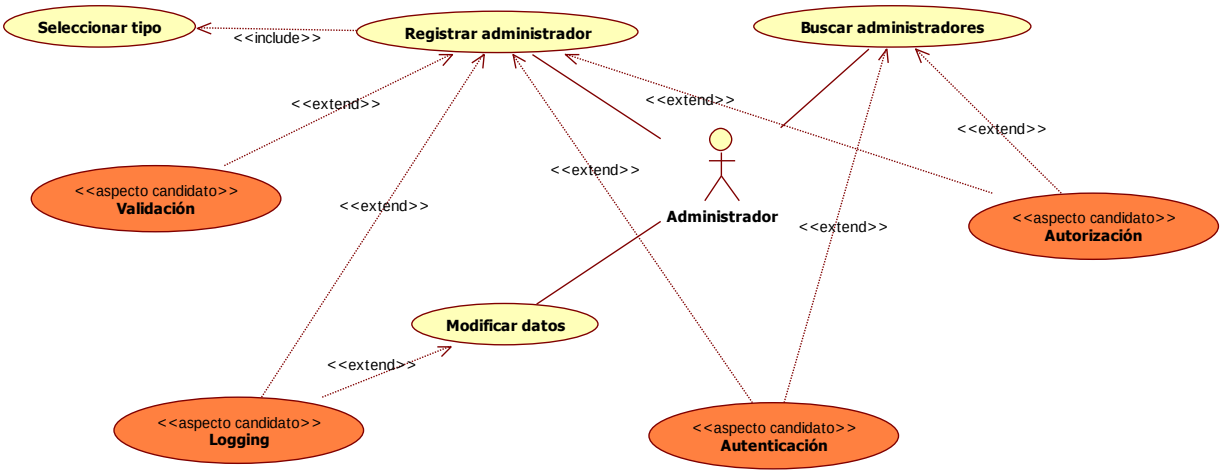


Ilustración 74: Caso de uso de infraestructura y requerimientos de gestión de administradores

Fuente: Elaboración propia.

3.3.1.4. Reservas de ejemplar

La reserva de un ejemplar se da en un ambiente web donde el usuario busca los ejemplares ya sean por títulos, autores, temario, editorial, código. Para realizar la reserva deberá seleccionar los ítems deseados según le perfil del usuario, seguidamente enviar su lista de reservas y el sistema le enviará una confirmación del éxito de la reserva, para realizar la reserva es necesario que el usuario haya iniciado sesión y esté activo para poder realizar una reserva. Los usuarios que estén sancionados no podrán realizar una reserva. El sistema también debe tener una opción para cancelar una reserva.

Identificación de la infraestructura del sistema:

- El usuario tiene que tener una sesión iniciada
- No se podrá reservar un ítem prestado
- El usuario no debe estar sancionado

Identificación de requerimientos de la aplicación:

- Buscar ejemplares
- Reservar ejemplares
- Cancelar reservas

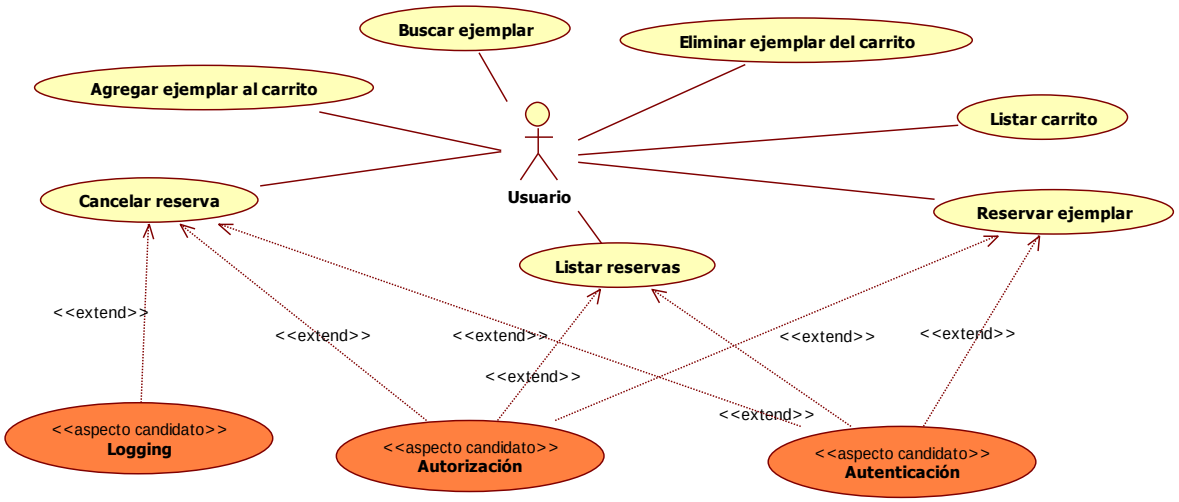


Ilustración 75: Caso de uso de infraestructura y requerimientos de reserva de ejemplar

Fuente: Elaboración propia.

3.3.1.5. Préstamos

El préstamo puede ser un préstamo directo en la biblioteca o un préstamo de una reserva realizada por un usuario, para realizar un préstamo el administrador debe haber iniciado sesión y además estar autorizado a realizar un préstamo. El préstamo solo se puede realizar dentro de un ciclo académico. El usuario solicita el ejemplar a través del código y el administrador ingresa el código y el sistema verifica la disponibilidad, si está disponible el administrador ingresa el código del usuario y después realiza el préstamo. Luego de haberse realizado el préstamo la disponibilidad del ejemplar cambia a **En préstamo**. Y por último se registra la actividad del préstamo indicando quien realizo el préstamo y el número de préstamo. La restricción es que solo se preste un ejemplar a un usuario que esté activo, no tenga deudas, no esté sancionado y que no haya sobrepasado el número de préstamos permitidos por tipo de usuario.

Identificación de la infraestructura del sistema:

- El administrador debe de haber iniciado sesión
- Tener la autorización para realizar el préstamo
- El préstamo debe de darse dentro de un ciclo académico
- Verificar la disponibilidad del ejemplar
- El usuario que esté activo
- El usuario no tenga deudas
- El usuario no esté sancionado
- El usuario no haya sobrepasado el número de préstamos permitidos por tipo de usuario

Identificación de requerimientos de la aplicación:

- Prestar ejemplar
- Prestar reserva

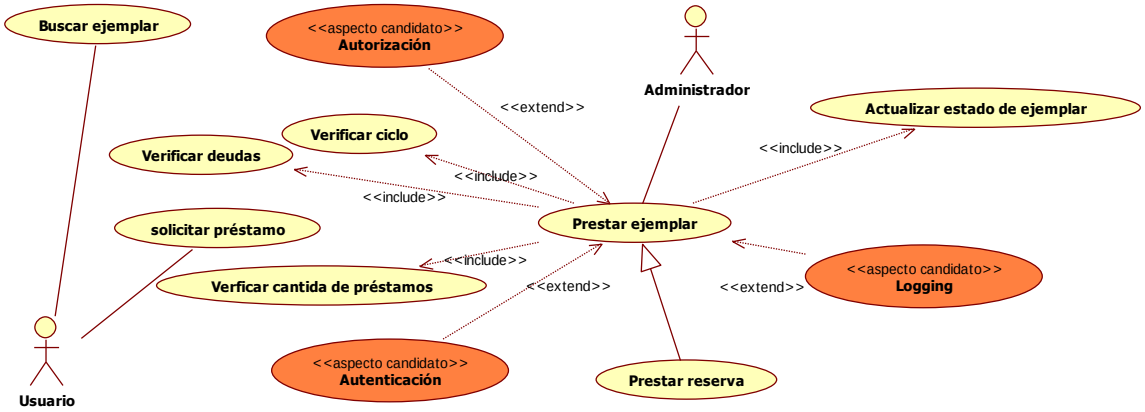


Ilustración 76: Caso de uso de infraestructura y requerimientos de préstamos.

Fuente: Elaboración propia.

3.3.1.6. Devoluciones

El usuario realiza la devolución física del libro y el administrador además de haber iniciado sesión, estar autorizado, revisa la integridad del material y verifica el registro de préstamos, ubica el préstamo realizado al usuario y selecciona el préstamo para realizar la devolución y adicionando una observación acerca del estado físico o la condición del ejemplar devuelto si fuese el cargo, Si la devolución es a destiempo se sanciona por la cantidad de días configurada para el usuario según su perfil. Así mismo cuando se realiza el registro de devolución la disponibilidad del ejemplar cambia a *Disponible*.

Identificación de la infraestructura del sistema:

- Haber iniciado sesión.
- Estar autorizado.
- Cambiar a Disponible un ejemplar cuando se realiza el registro de devolución.

Identificación de requerimientos de la aplicación:

- Buscar préstamos.
- Registrar devolución.

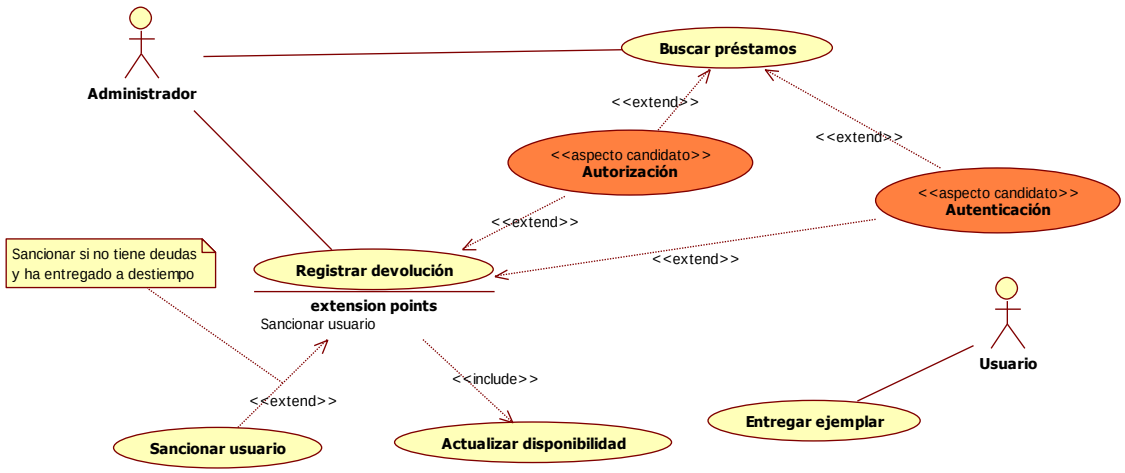


Ilustración 77: Caso de uso de infraestructura y requerimientos de devoluciones.

Fuente: Elaboración propia.

3.3.1.7. Configuraciones del sistema

El administrador podrá exportar un Backup del sistema podrá seleccionar una copia completa o incremental, para realizarlo tendrá que estar autorizado, haber iniciado sesión y además estar Activo.

Identificación de la infraestructura del sistema:

- El administrador tendrá que estar autorizado
- Haber iniciado sesión
- Estar activo

Identificación de requerimientos de la aplicación:

- Seleccionar una copia completa
- Seleccionar una copia incremental
- Recibir un archivo de la copia

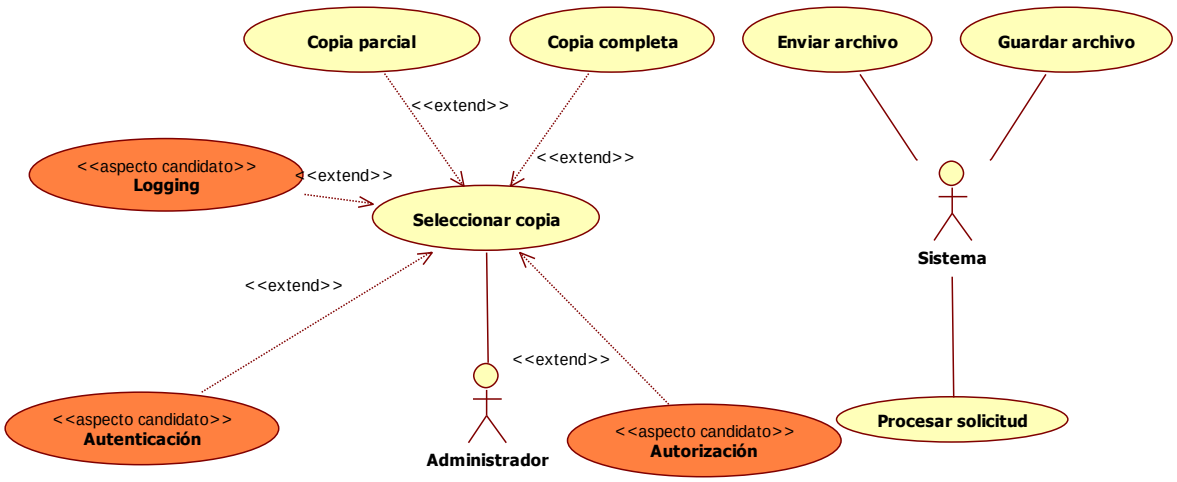


Ilustración 78: Caso de uso de infraestructura y requerimientos de exportar Backup del sistema.

Fuente: Elaboración propia.

El administrador podrá importar un *Backup* generado por el sistema, seleccionando un archivo para restablecer el sistema a una fecha deseada, para poder realizar esta acción el administrador tendrá que estar autorizado, haber iniciado sesión y además estar Activo, el sistema muestra las consultas realizadas y a su vez se registra el evento indicando la fecha y el usuario que realizó la acción.

Identificación de la infraestructura del sistema:

- El administrador tendrá que estar autorizado.
- El administrador tendrá que haber iniciado sesión.
- Registra el evento indicando la fecha y el usuario que realizó la acción.

Identificación de requerimientos de la aplicación:

- Enviar archivo.
- Seleccionar archivo.

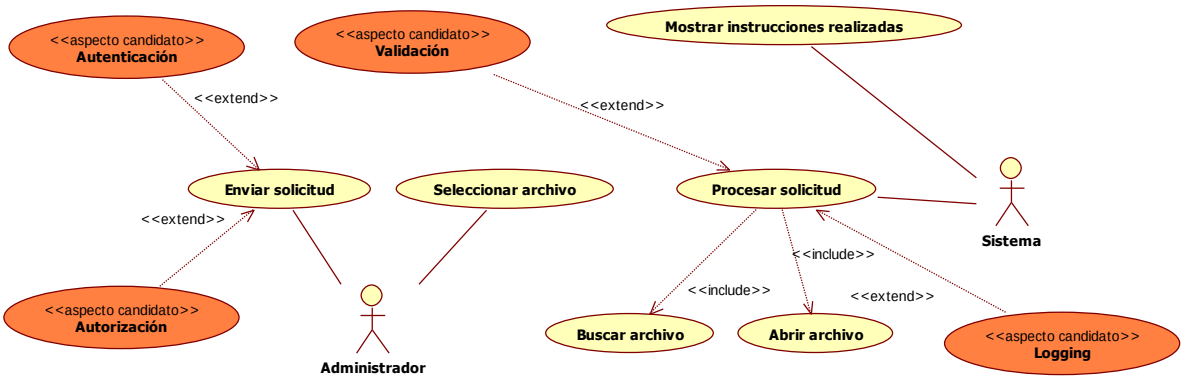


Ilustración 79: Caso de uso de infraestructura y requerimientos de importación de Backup del sistema.

Fuente: Elaboración propia.

3.3.1.8. Gestión de utilidades

El administrador podrá realizar registros, actualizaciones, busquedas y eliminaciones, asi mismo para estas acciones debe de estar autenticado en el sistema y estar autorizado para realizar dichas acciones, para registros y actualizaciones el sistema deberá llevar un registro de las acciones realizadas en ingreso de datos y actualizaciones.

Identificación de la infraestructura del sistema:

- El administrador tendrá que estar autorizado.
- El administrador tendrá que estar autenticado.
- Registro de las acciones realizadas en ingreso de datos y actualizaciones.

Identificación de requerimientos de la aplicación:

- Actualizaciones
- Búsquedas
- Registros

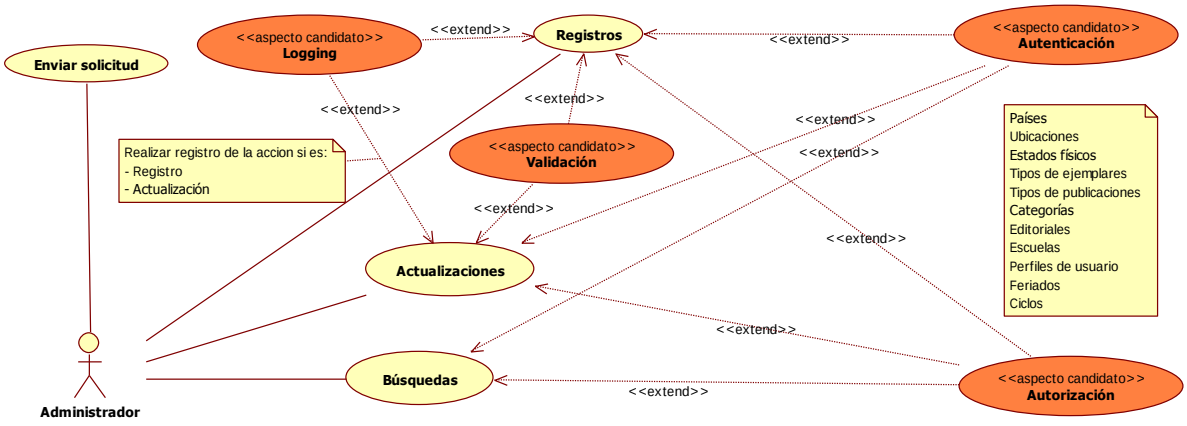


Ilustración 80: Caso de uso de infraestructura y requerimientos gestión de utilidades

Fuente: Elaboración propia.

3.3.2. Análisis

Las especificaciones de casos de uso describen el comportamiento deseado del sistema desde el punto de vista externo utilizando el lenguaje de los usuarios finales, pero no describe como se relacionan las entidades que participan y además el papel que juegan la nueva entidad llamada aspecto. Por ello se necesita una representación más precisa, esto es posible con el modelo de análisis.

El análisis es un refinamiento de los requisitos con el objetivo de crear especificación precisa de requisitos. Su propósito también es encontrar abstracciones en forma de clases que hacen los requisitos más claro en comparación a las especificaciones de casos de uso.

Según el modelo de análisis de (Jacobson, y otros, 2004) se identifican dos elementos:

- **Capa de aplicación**

La capa de aplicación contiene elementos que dan cuenta de los flujos de trabajo en los casos de uso de apoyo a los actores principales del sistema. Los elementos de esta capa normalmente utilizan los elementos de la capa de dominio para darse cuenta de casos de uso.

- **Capa de dominio**

La capa de dominio contiene elementos que representan conceptos de dominio significativos. Capturan información que se ha mantenido, seguido, o manipulados por el sistema y los comportamientos asociados para hacerlo. Estos elementos son normalmente compartidos a través de la realización de los casos de uso.

Se identifican los siguientes modelos de análisis incluyendo la capa de dominio y la capa de aplicación como la realización de los casos de uso en un contexto estático

3.3.2.1. Administración de usuarios

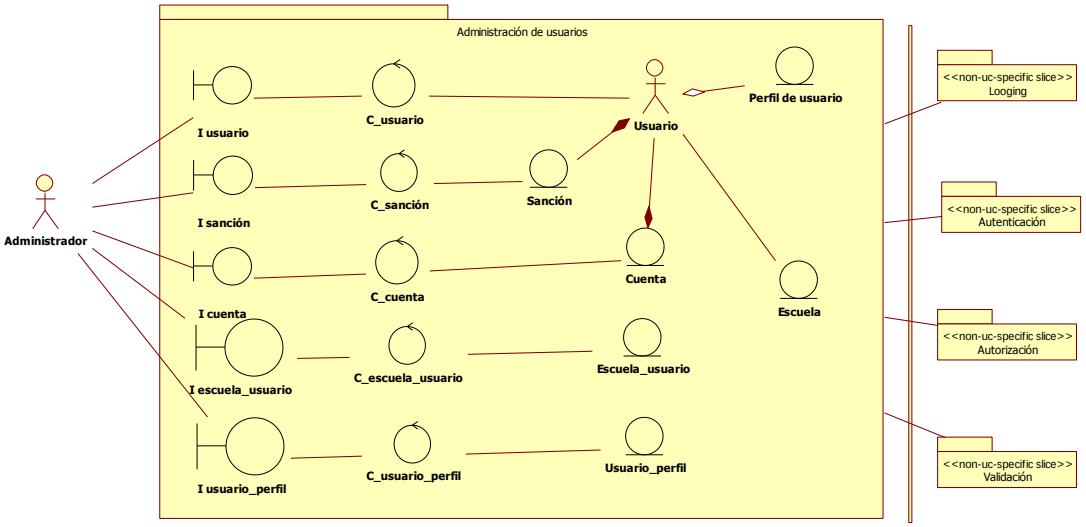


Ilustración 81: Administración de usuarios.

Fuente: Elaboración propia.

3.3.2.2. Administración de ejemplares

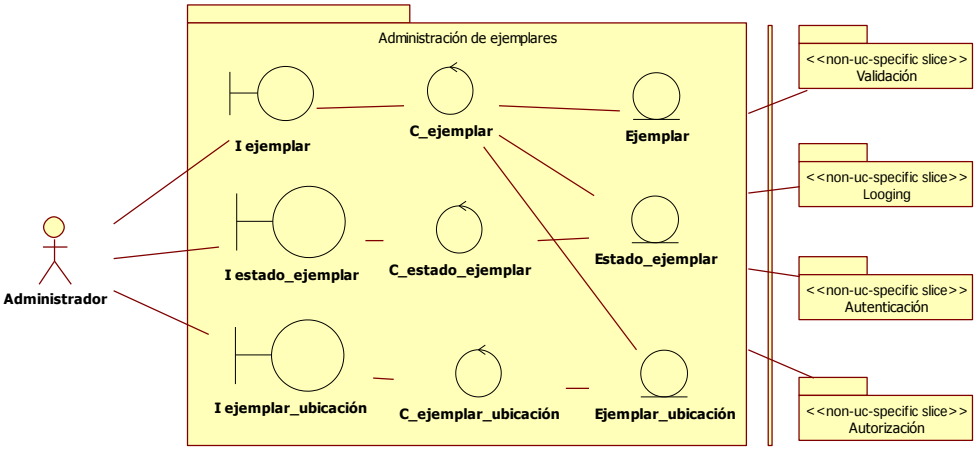


Ilustración 82: Administración de ejemplares.

Fuente: Elaboración propia.

3.3.2.3. Gestión de administrador

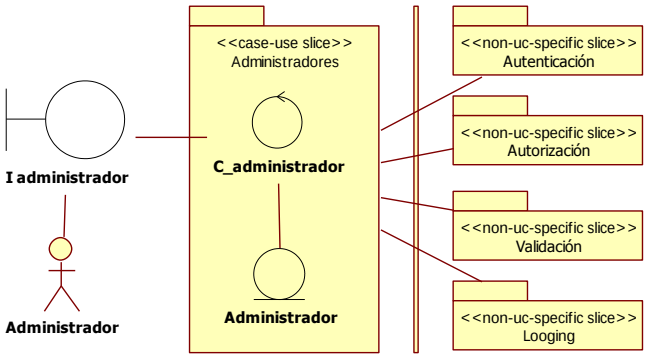


Ilustración 83: Modelo de análisis de gestión de administradores

Fuente: Elaboración propia.

3.3.2.4. Reservas

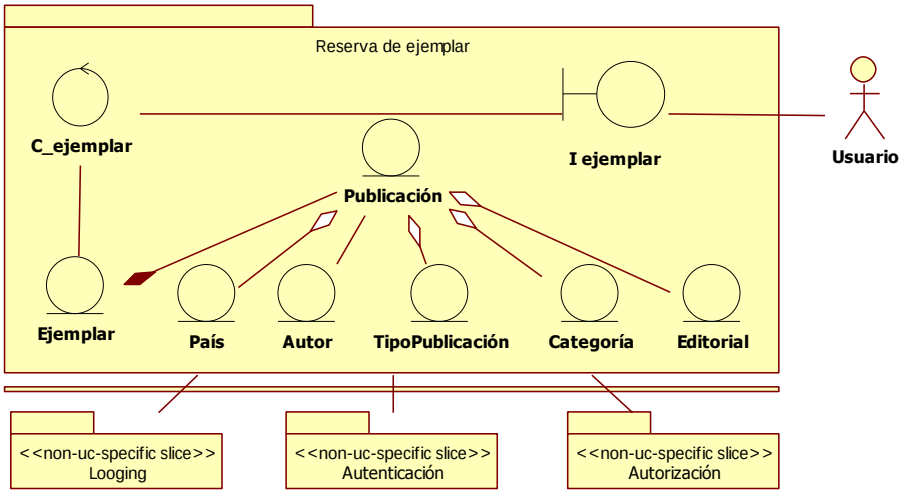


Ilustración 84: Reservas

Fuente: Elaboración propia.

3.3.2.5. Préstamos

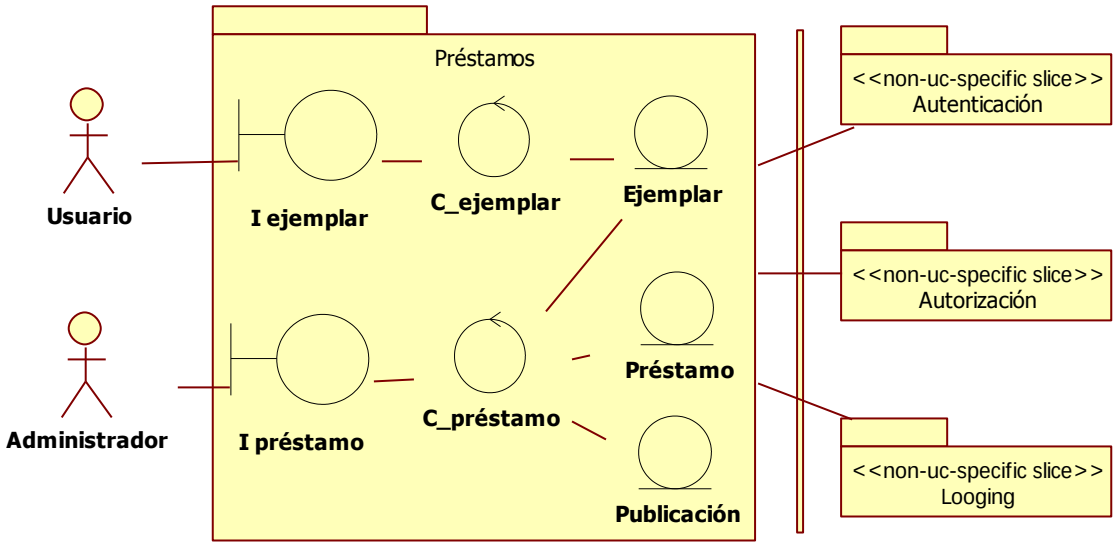


Ilustración 85: Préstamos

Fuente: Elaboración propia.

3.3.2.6. Devoluciones

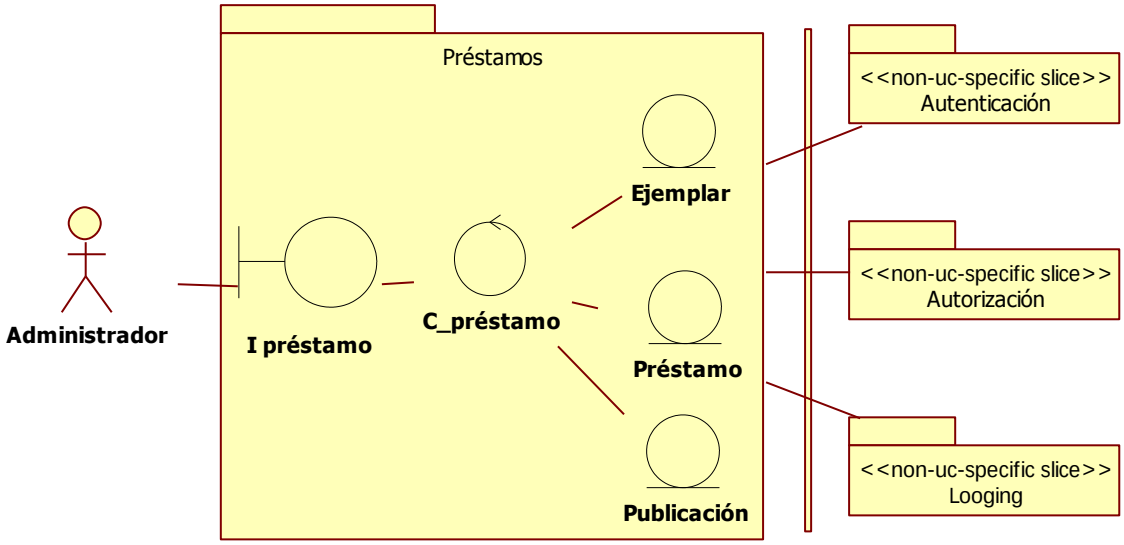


Ilustración 86: Devoluciones

Fuente: Elaboración propia.

3.3.2.7. Configuraciones del sistema

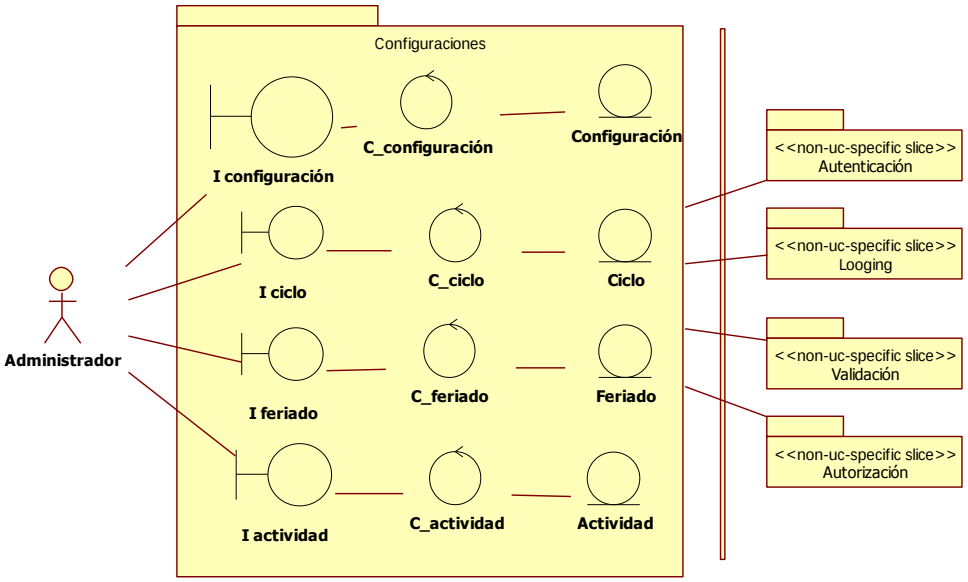


Ilustración 87: Configuraciones del sistema.

Fuente: Elaboración propia.

3.3.2.8. Gestión de utilidades

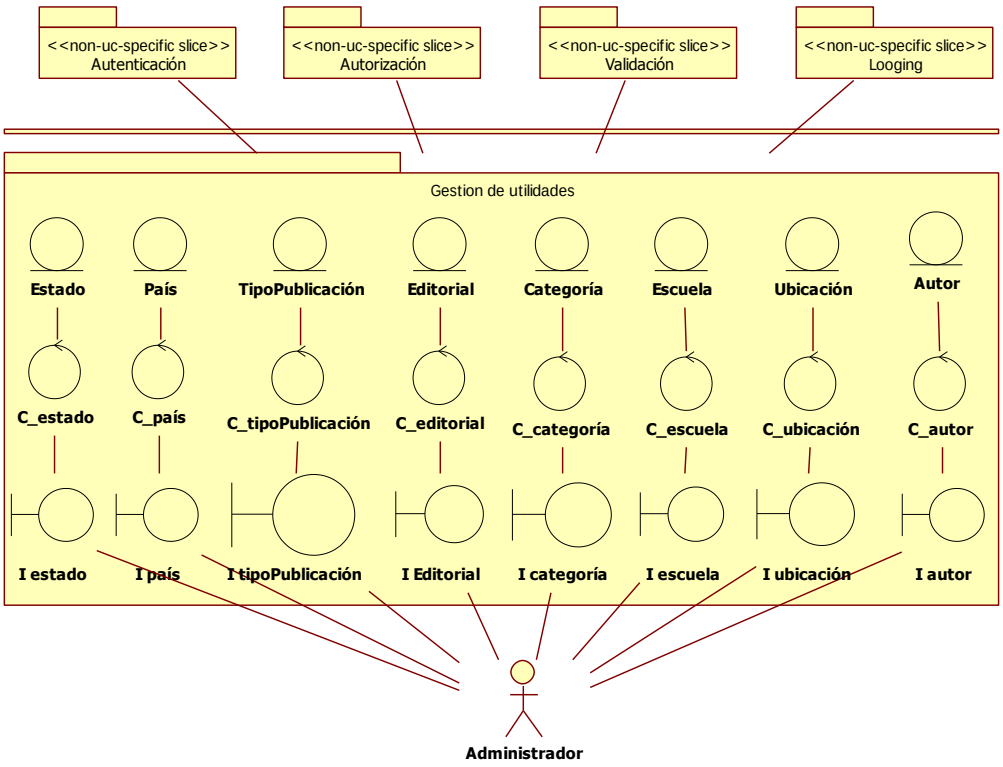


Ilustración 88: Gestión de utilitarios.

Fuente: Elaboración propia.

Se ha descrito el comportamiento estático del sistema incluyendo clases de entidad, frontera, controlador y aspectos, pero aún no se ha definido la interacción entre estos. Un modelo dinámico nos ayuda a comprender este comportamiento, trabajos como (Vidal, y otros, 2012), (Sánchez, y otros, 2013), (Zhang, y otros, 2009) describen la adaptación del diagrama de secuencias del UML para el modelado orientado a aspectos, con el objeto de lograr implementar el modelo dinámico en el cual participa un aspecto. En esta sección se presentará el diseño de los principales eventos que pertenecen a los modelos de análisis como lo son:

3.3.2.9. Diagrama de secuencia de préstamo de ejemplar

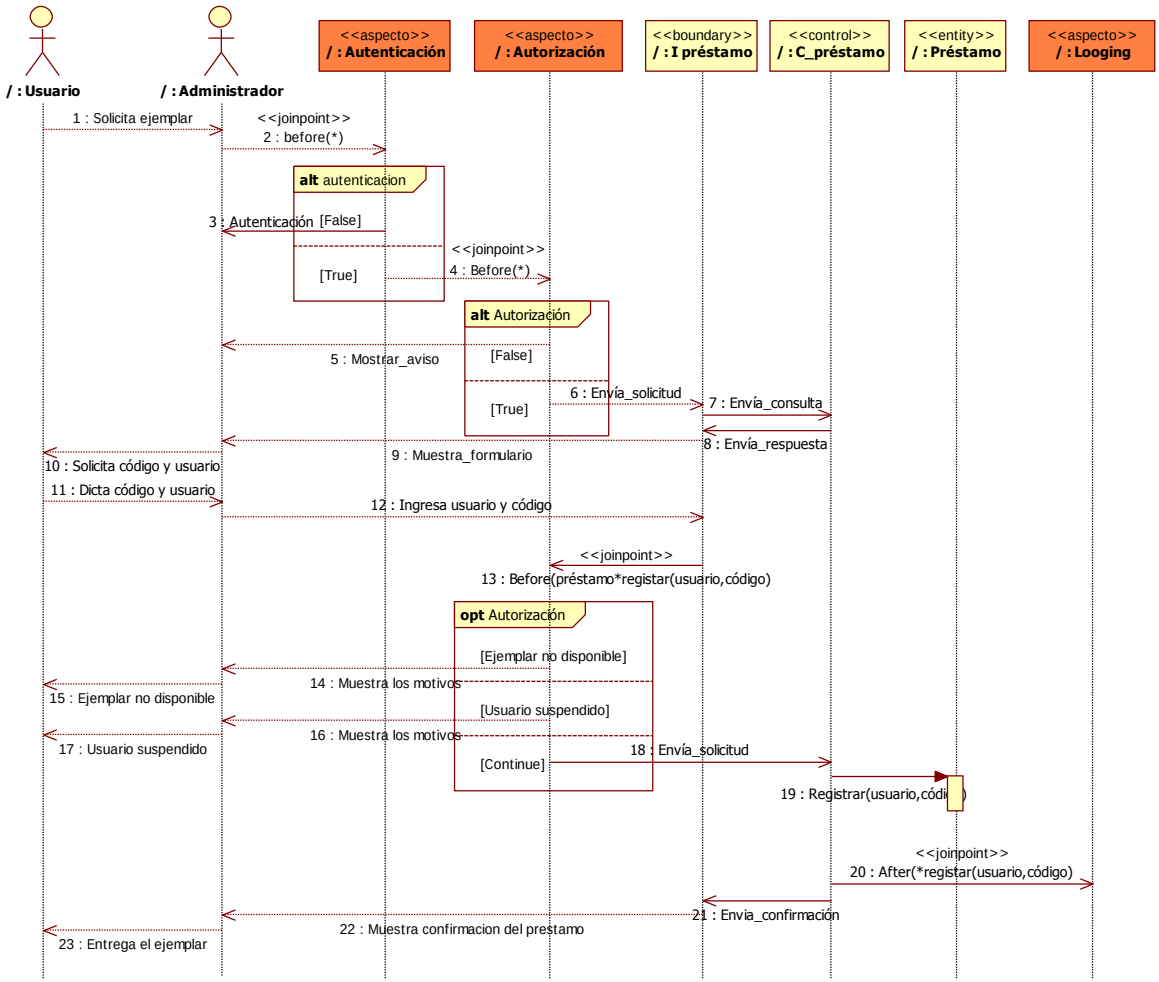


Ilustración 89: Diagrama de secuencia de préstamo de ejemplar.

Fuente: Elaboración propia.

3.3.2.10. Ejemplares

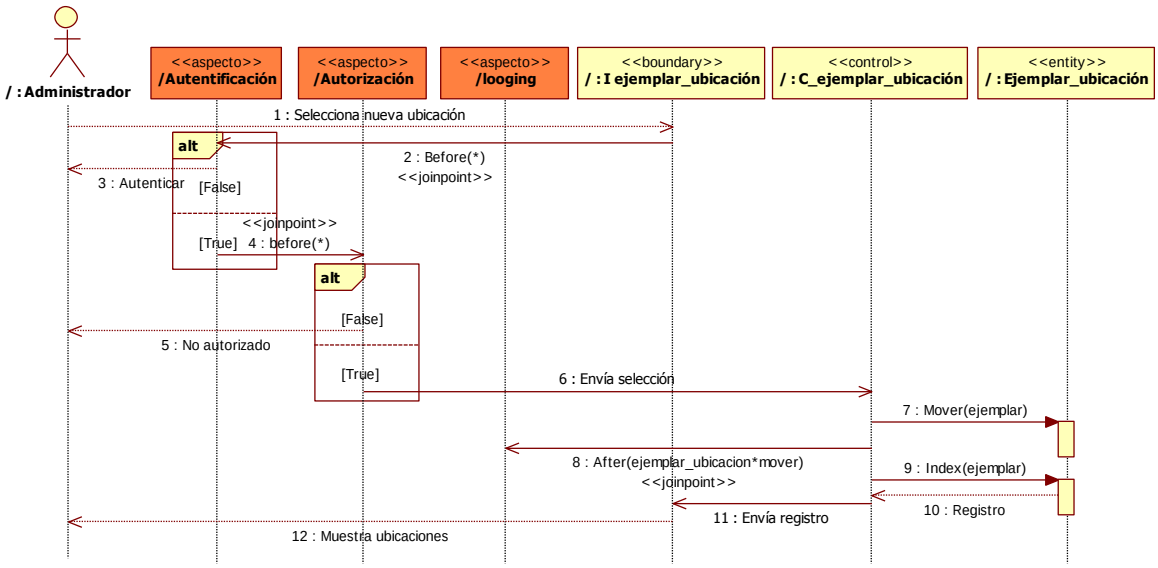


Ilustración 90: Ubicaciones de ejemplar.

Fuente: Elaboración propia.

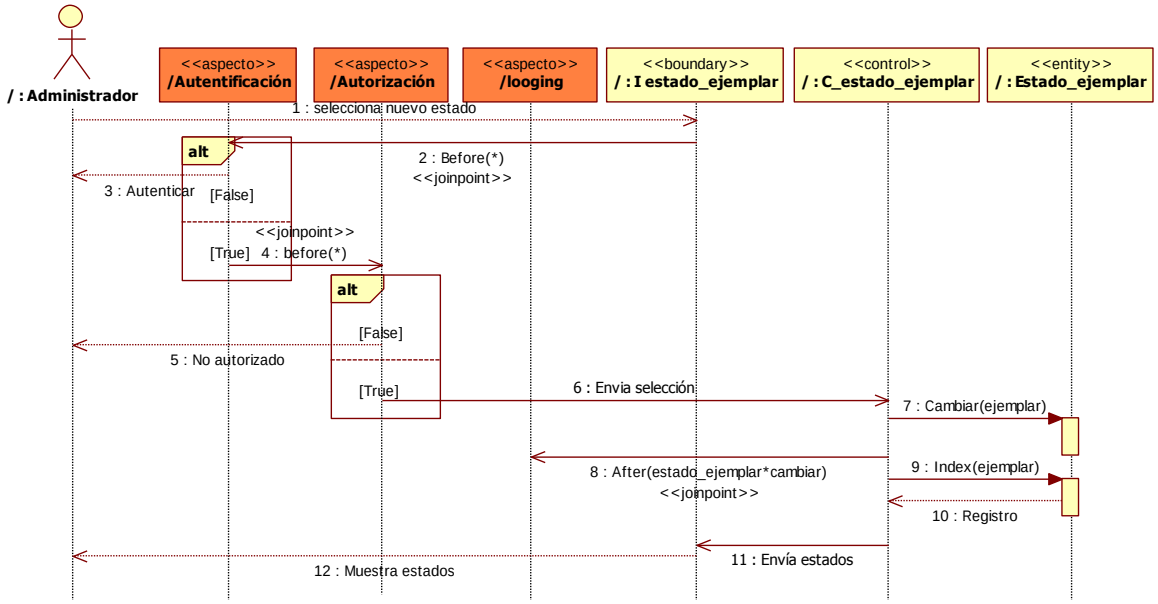


Ilustración 91: Estados de ejemplar.

Fuente: Elaboración propia.

3.3.2.11. Usuario

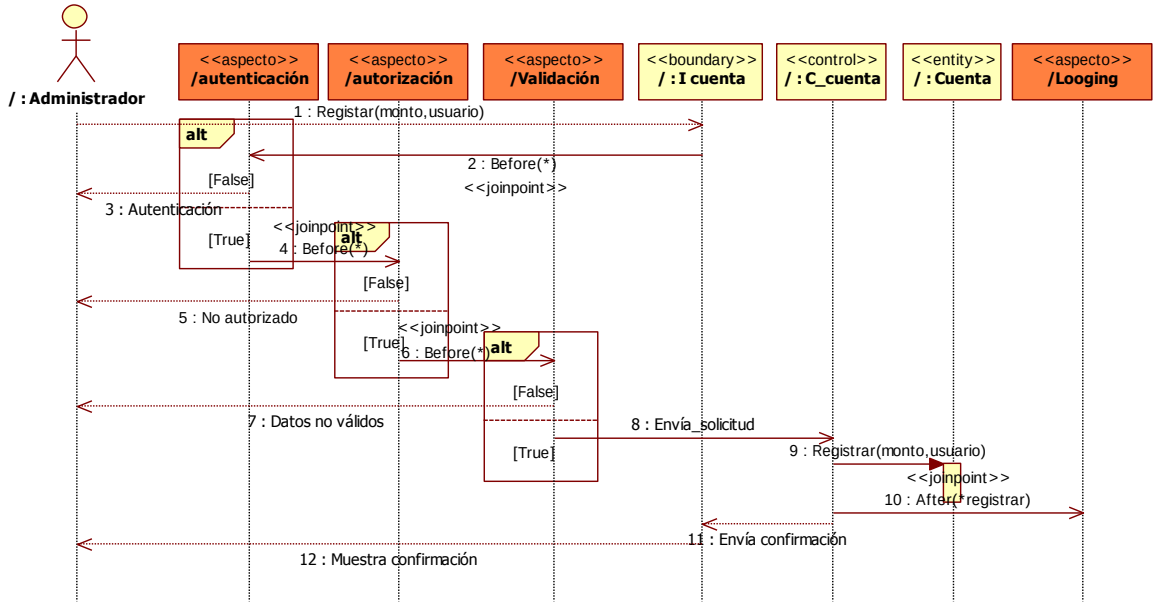


Ilustración 92: Cuentas de usuario.

Fuente: Elaboración propia.

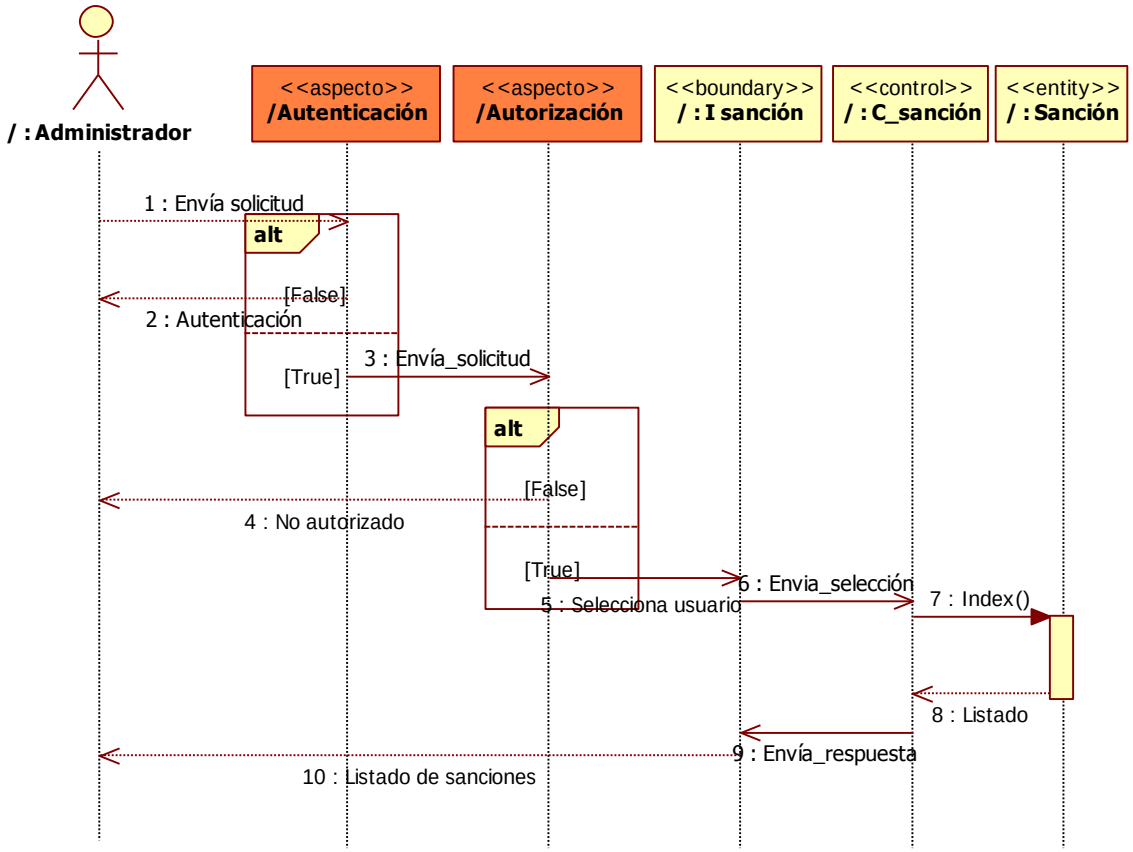


Ilustración 93: Sanciones de usuario.

Fuente: Elaboración propia.

3.3.2.12. Reservar ejemplares

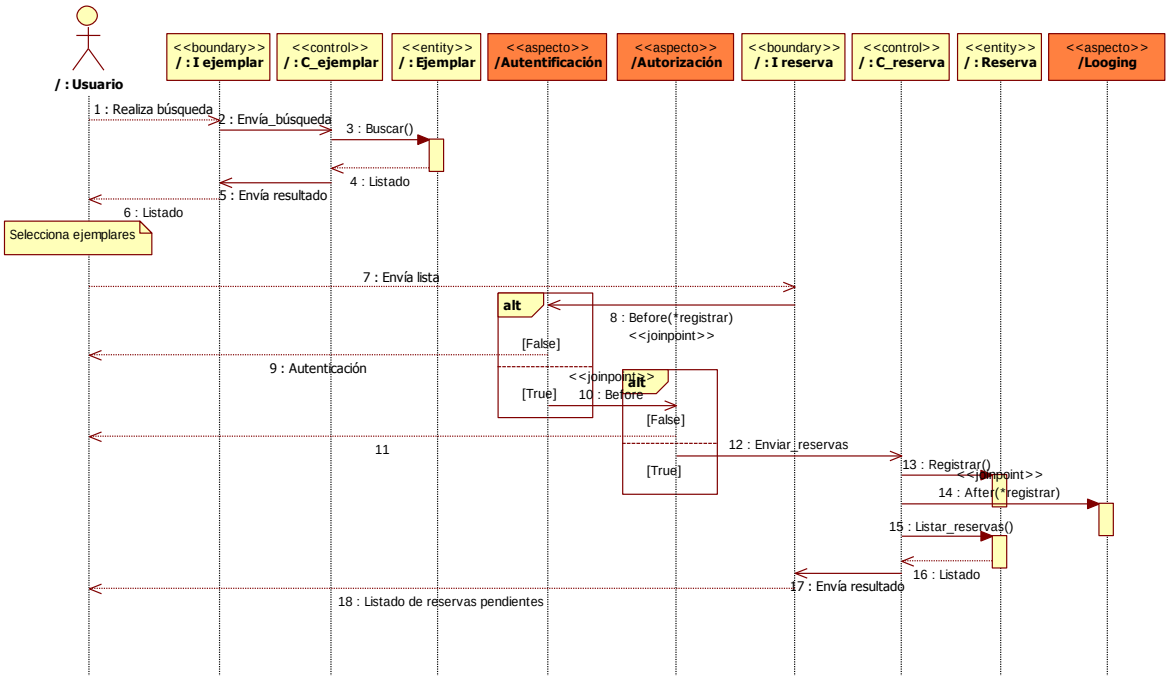


Ilustración 94: Reserva de ejemplares

Fuente: Elaboración propia.

3.3.3. Diseño

El diseño orientado a aspectos es el proceso de diseñar un sistema que utilice los aspectos para implementar las competencias transversales y extensiones que se identificaron durante el proceso de ingeniería de requerimientos. En esta etapa es necesario traducir las competencias que se relacionan con el problema que hay que resolver a los aspectos correspondientes en el programa que implementará la solución. También es indispensable comprender cómo se combinarán estos aspectos con otros componentes de sistema y garantizar que no surjan ambigüedades de composición. (sommerville, 2008)

El modelo de diseño es un refinamiento del análisis realizado con los aspectos, se ha definido una estructura de alto nivel, ahora se definirá la arquitectura que dará soporte al sistema.

3.3.3.1. Estructura de los elementos de diseño.

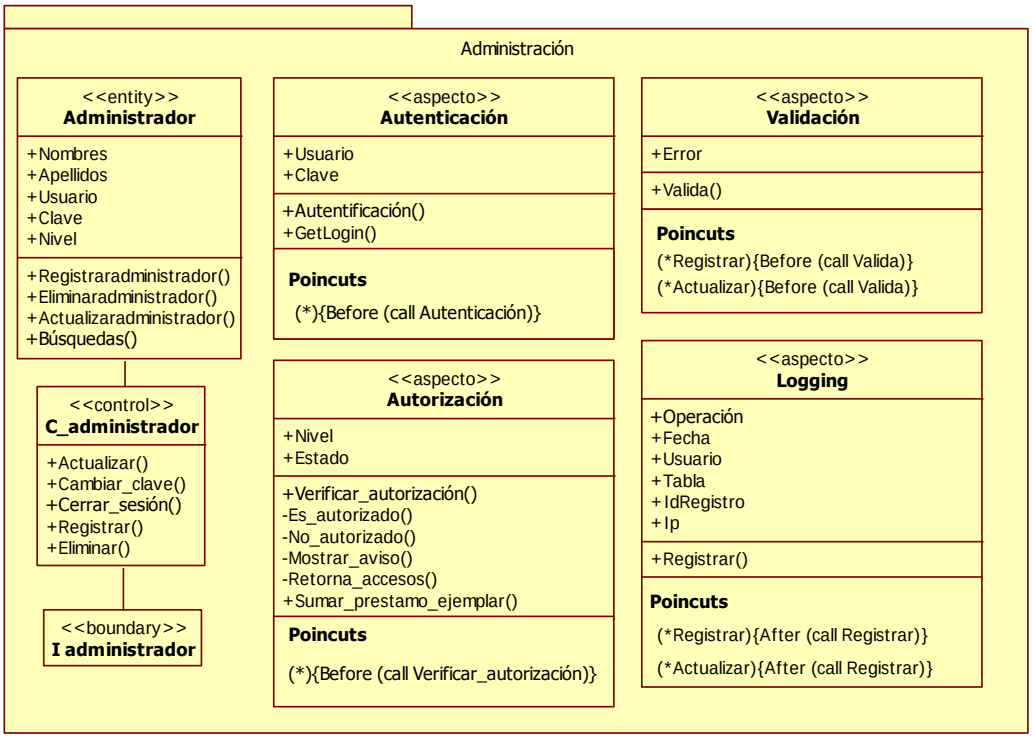


Ilustración 95: Gestión de administradores

Fuente: Elaboración propia.

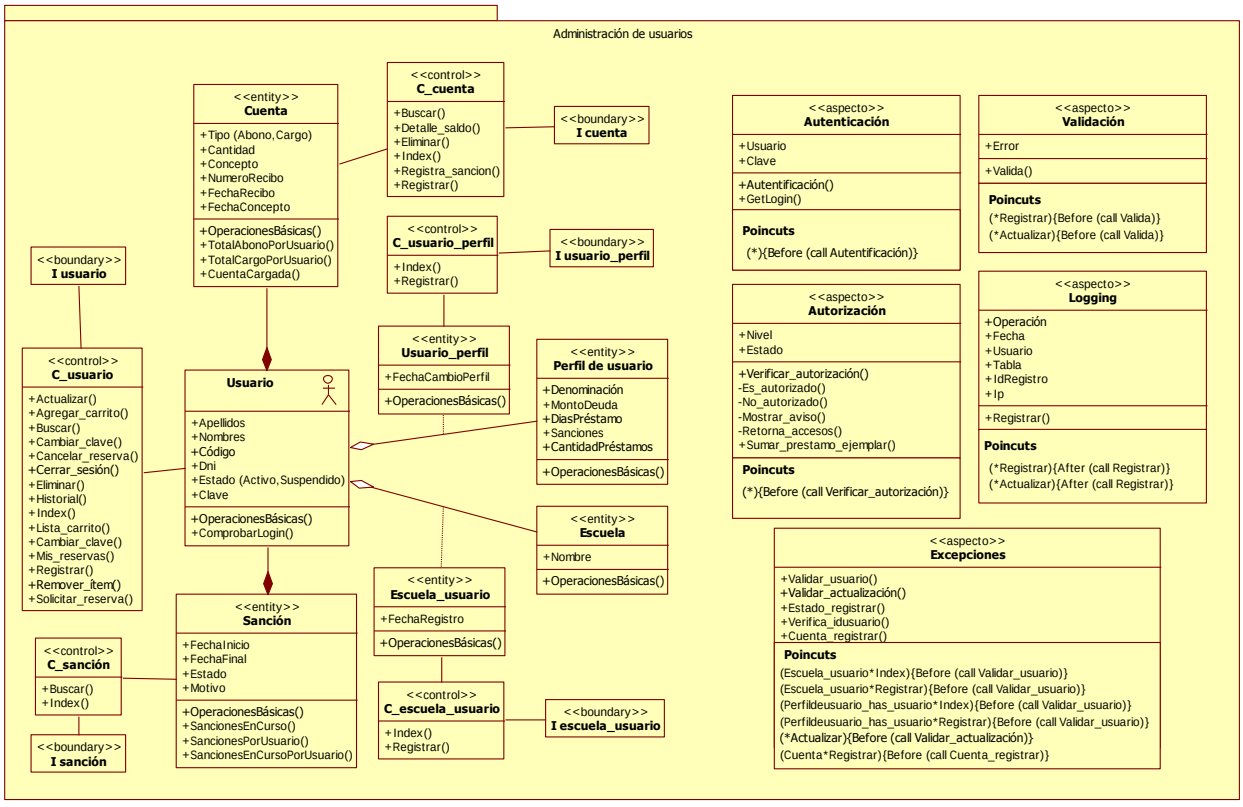


Ilustración 96: Administración de usuarios

Fuente: Elaboración propia.

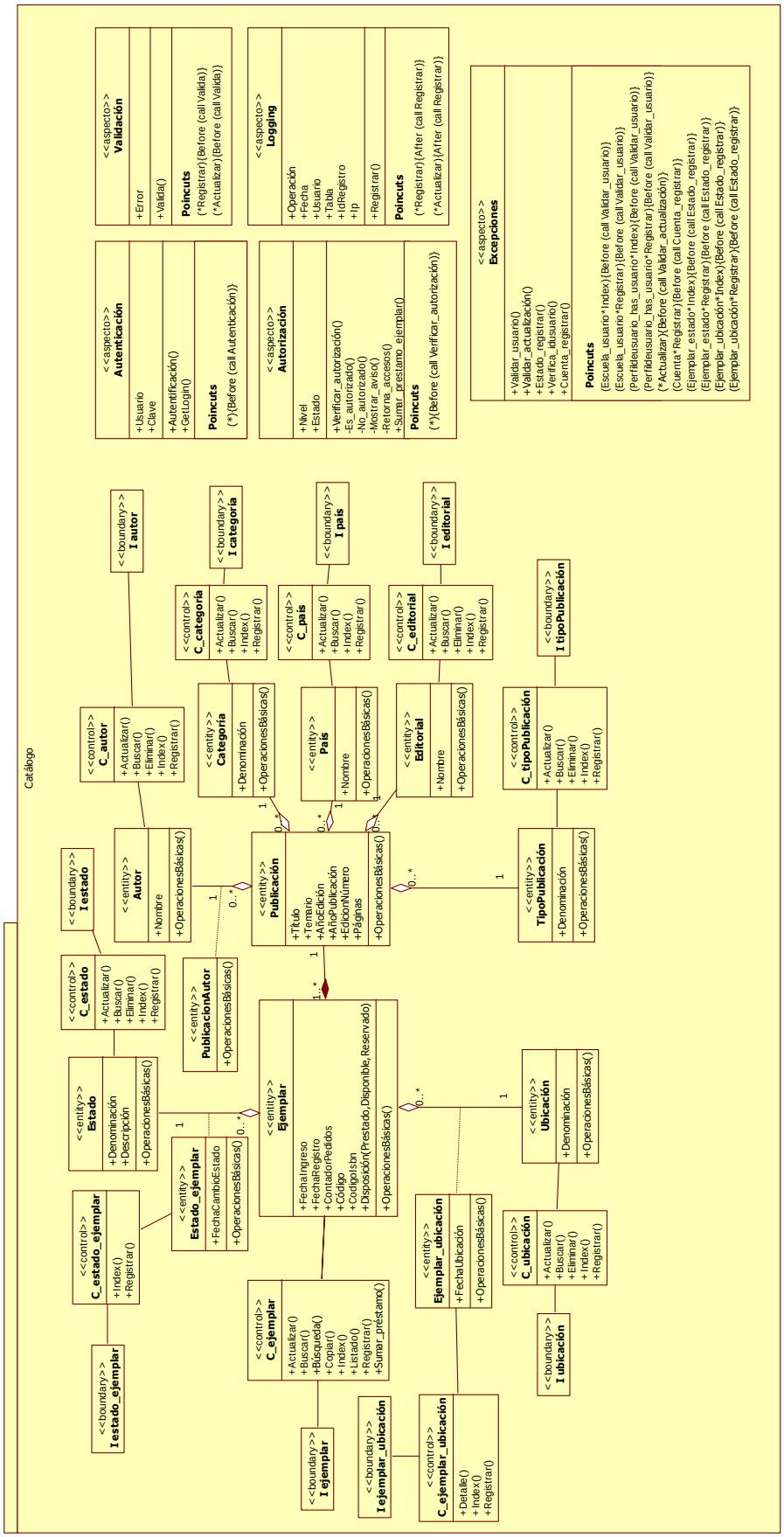


Ilustración 97: Catálogo

Fuente: Elaboración propia.

Ilustración 98: Préstamos y devoluciones

Fuente: Elaboración propia.

Ilustración 99: Sistema

Fuente: Elaboración propia.

Ilustración 100: Utilidades.

Fuente: Elaboración propia.

3.3.3.2. Análisis de conflictos entre aspectos

Una etapa adicional que se da en el diseño orientado a aspectos es la identificación de temprana de conflictos entre aspectos para evitar que no generen un comportamiento impredecible en el sistema una vez implementado. Trabajos como (Sardinha, y otros, 2013), (Casas, y otros, 2006), (Casas, y otros, 2005) (Hormazábal, 2010), (Brito, y otros, 2013) proveen conocimiento acerca del tratamiento que se da para estructurar los aspectos de forma que solo se ejecuten como se han diseñado. En términos simples "*Un conflicto ocurre cuando dos o más aspectos compiten por su activación*".

En esta etapa se tratarán los conflictos de una manera simple, a través de la asignación de prioridades a los aspectos utilizando instrucciones dominantes. De esta manera, si se desea dar precedencia a un aspecto de otro, se le asigna una prioridad más alta al primero (regla de precedencia). Como sigue:

1. Autenticación
2. Autorización
3. Manejo de excepciones
4. Validaciones
5. Looging

3.3.3.3. Diseño de nombres

Para permitir un manejo global con la intervención de los puntos de corte se establecen nombres comunes en todas las clases como son registrar, actualizar, eliminar y buscar. Y demás nombres que se identificarán en el archivo *aspectos.xml*. Por ejemplo para los nombres de *registrar* se ejecutarán los aspectos de autentificación, autorización, validación y looging.

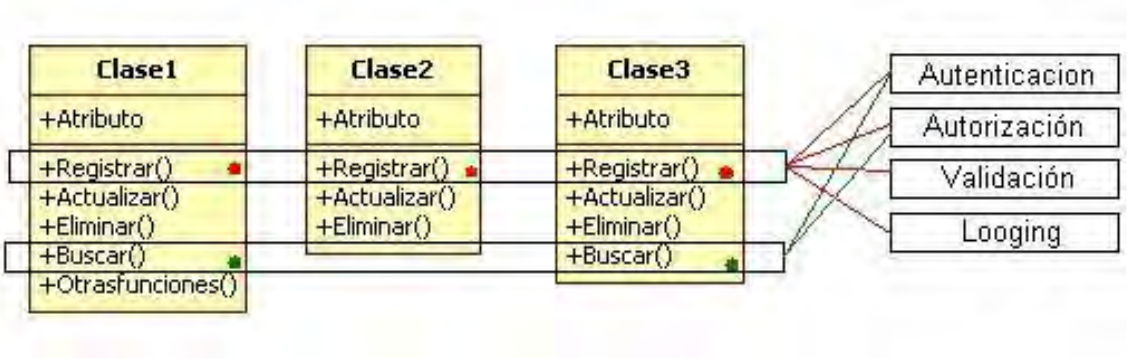


Ilustración 101: Ilustración de la ejecución de los puntos de corte.

Fuente: Elaboración propia.

3.3.3.4. Arquitectura de software

Una buena arquitectura mantiene incumbencias separadas, lo que significa que los cambios en una parte del sistema no se propagan a otras partes del sistema. Incluso si lo hacen, se puede identificar claramente qué cambios se deben hacer. Si existe la necesidad de extender la arquitectura, el impacto debería ser mínimo. Todo lo que ya trabaja debería seguir trabajando. Para un sistema que aplica la orientación aspecto, las diferentes incumbencia del sistema pueden mantenerse separadas de manera efectiva (Jacobson, y otros, 2004).

Se selecciona la arquitectura de modelo-vista-controlador con la inclusión de aspectos.

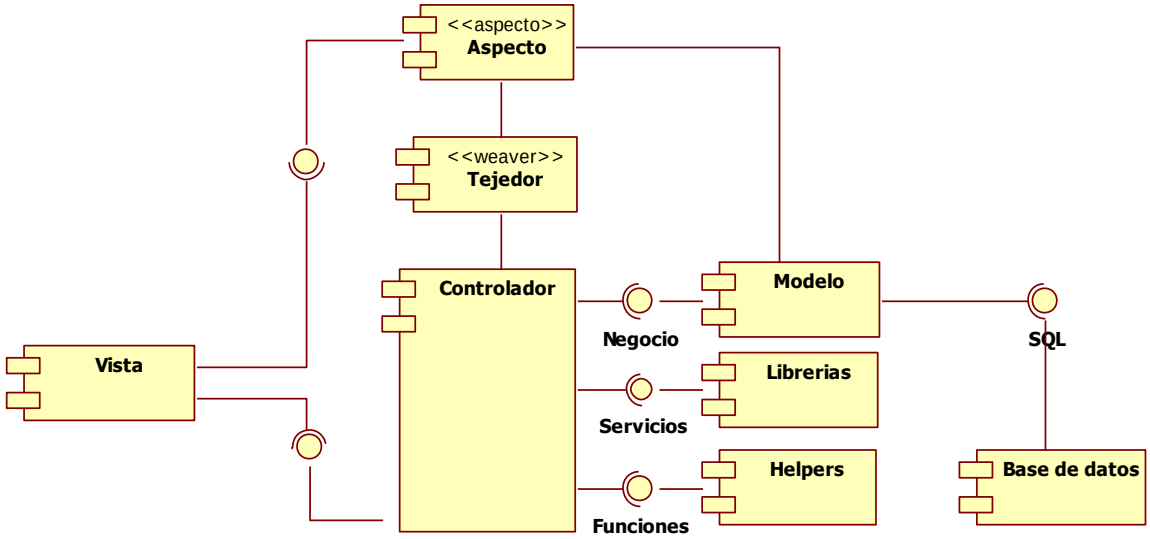


Ilustración 102: Diagrama de componentes MVC con la inclusión de aspectos

Fuente: Elaboración propia.

3.3.3.5. Estructura de implementación.

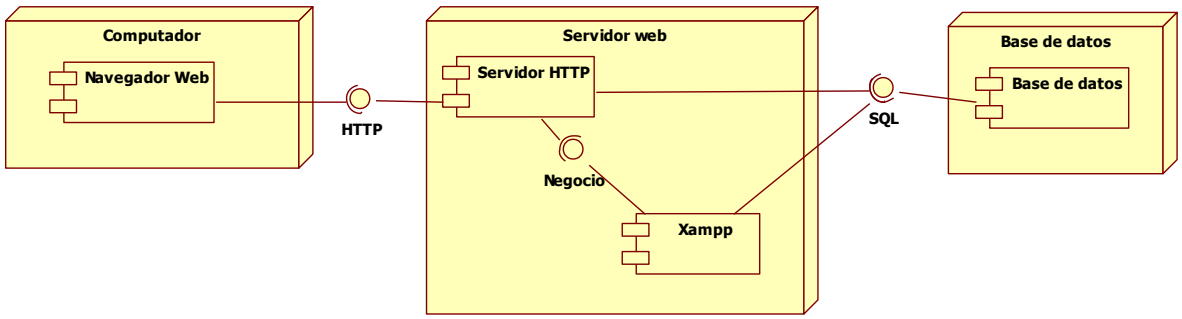


Ilustración 103: Diagrama de despliegue

Fuente: Elaboración propia.

3.3.4. Implementación

3.3.4.1. Elección del lenguaje de programación

La elección del lenguaje de programación para la implementación del software con la programación orientada a aspectos se debe a realizar los objetivos de la investigación, así mismo para que se realicen las pruebas bajos las mismas condiciones se elige Php.

3.3.4.2. Elección de un framework para la arquitectura MVC

Existen frameworks orientados a aspectos en Php como son:

- PHPAspect
- Aspect-Oriented PHP
- AOP-Php
- GO! Aspect-Oriented
- MFAOPHP

En algunos casos como PHPAspect, Aspect-Oriented PHP, MFAOPHP no proveen soporte para la arquitectura MVC, en otro como GO Aop exige la instalación de dependencias lo que realizarlo en el lado servidor escapa del alcance del proyecto, otro inconveniente está ligado a que no se realicen las pruebas bajos las mismas condiciones. Por esos motivos se elige Codeigniter. Se realizará el sistema orientado a aspectos incluyendo JoinPoints, PointCuts, Advices y Aspectos.

3.3.4.3. Elección del tipo de tejido de los aspectos

Para lograr la mayor flexibilidad y por ende la mantenibilidad del sistema se opta por un tejido dinámico, ya que cuenta con la posibilidad de modificar un aspecto según la información generada en ejecución, como también introducir o remover dinámicamente aspectos. Además PHP es un lenguaje de programación interpretado no se vería el impacto de la programación orientada a aspectos con un tejido estático.

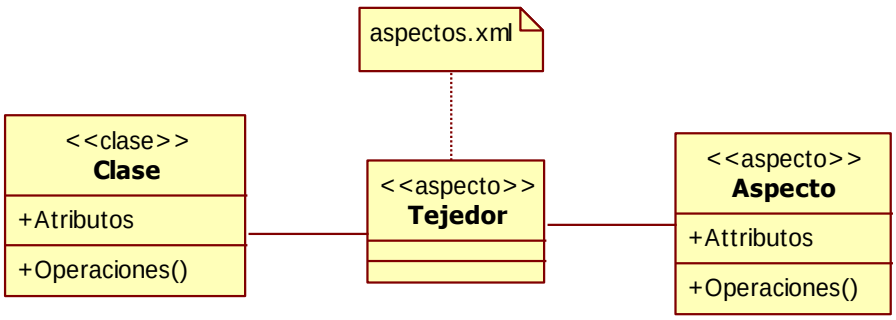


Ilustración 104: Tejido de aspectos con las clases bases.

Fuente: Elaboración propia.

En el archivo *aspectos.xml* se estructuran los puntos de unión donde se ejecutaran los avisos (advices) pudiéndose ejecutar antes, después o durante, puntos de corte donde se establece las condiciones para que entren en acción a los puntos de unión.

Durante el tiempo de ejecución el tejedor captura el nombre de la clase y el nombre de la función y los compara con los puntos de corte y ejecuta los puntos de unión los cuales indican que aspectos deben realizar su trabajo.

3.3.4.4. Implementación de la capa de aplicación

En esta etapa se desarrolló como primera etapa la capa de aplicación que incluye solo requerimientos funcionales y son los siguientes:

- Administración de utilidades: Para la codificación de esta sección se realizaron solo las operaciones básicas de la aplicación no se incluyeron las propiedades del dominio lo que significó un trabajo más simple.
- Administración de ejemplares: Se codificó primero la clase de publicaciones y luego la clase de ejemplares, así mismo la implementación de cambios de estado, cambios de ubicación, actualización, registro y búsqueda de ejemplares
- Configuración del sistema: Se codificaron las clases de configuración, feriados y ciclos, no se implementó las restricciones para realizar un registro de un ciclo.
- Gestión de administrador: Se codificó el registro de administradores incluyendo operaciones básicas, cambios de clave, cerrado de sesión. Sin incluir validaciones, ni controles de acceso.
- Administración de usuarios: En esta sección se codificaron el registro de usuarios con la inclusión de pertenencia a escuelas y perfiles de usuario, así mismo se codificó el registro de cambio de escuelas y perfiles de usuarios. Para el lado del usuario se incluyó las funciones de realizar reservas, ver su historial de préstamos, agregar a la lista de pre-reservas y notificaciones que incluyen las cuentas por pagar si los hubiera. En toda esta etapa se realizó solo los requerimientos de la aplicación sin incluir las restricciones de acceso.
- Administración de préstamos: Se codificó primero las reservas con operaciones como registro y listado de reservas pendientes, luego los prestamos como registros, devoluciones, registro de sanciones, registro de suspensiones y búsqueda de préstamos.

3.3.4.5. Implementación de la capa de dominio

En esta etapa se centró en realizar la unión entre los aspectos y las clases mediante el tejedor con el apoyo del archivo *aspectos.xml* que contiene los puntos de corte que ejecutaran los puntos de unión.

- Aspecto de autenticación: Se codificó para verificar si un usuario está identificado o no en el sistema, así mismo si se identifica será re-direccionado al mismo lugar que trató de ingresar o a su sitio de inicio. También permite no aplicar la autentificación a secciones que se consideran de libre acceso para cualquier visitante como la sección de búsquedas, agregación de ítems, listado de ítems en pre-reservas. Se estableció una regla en la cual este aspecto realiza su comprobación en todas las clases *controlador* y funciones del sistema.

- Aspecto de autorización: Se codificó para proteger las funcionalidades del sistema permitiendo que solo sean usados por aquellos usuarios o administradores que tengan concedida la autorización. En este software se configuró para que un punto de corte ejecute los puntos de unión para todas las clases y funciones del sistema. En esta sección además se estableció los perfiles de accesos según el tipo de usuario sea administrador o usuario.

- Manejo de excepciones: Se codificó para controlar las situaciones inusuales en la ejecución del software, pero que se conocen y están contempladas en el diseño del software, estas excepciones se implementaron en aquellas funciones que requerían la entrada de un dato para generar un reporte, o para precondiciones para realizar una operación. Por ejemplo para registrar un préstamo tiene que existir un ejemplar además que esté disponible, por otro lado el usuario tiene que existir, estar activo, no deber saldos, no estar sancionado, y no exceder su número de préstamos permitidos. Así mismo para las actualizaciones de un registro tiene que ingresarse un id identificador de registro para poder realizar la consulta, y para poder realizar la actualización tiene que existir el registro. En una situación en el manejo de excepciones el sistema permitirá regresar a un punto inicial deteniendo la ejecución de la función.

- Validaciones: Las validación permitió establecer las reglas de validación para verificar, controlar o filtrar cada una de las entradas de datos que provienen desde el exterior del sistema. Se implementó este aspecto en las operaciones de registro y actualización.

- Looging: Permitirá llevar un registro cronológico de las operaciones que guarden relación con el registro de datos, las actualizaciones, intentos fallidos de inicios de sesión o sesiones iniciadas, así también las operaciones de accesos bloqueados.

4. Pruebas y resultados

4.1. Resultados obtenidos de la prueba del tamaño (LDC)

Ésta es una medida del tamaño de un programa. Por lo general, cuanto más grande sea el tamaño del código de un componente, más probable será que el componente sea complejo y proclive a errores. La longitud del código podría indicar la proclividad al error en los componentes, un recuento muy alto podría indicar que un tipo o método intenta hacer demasiado trabajo y debe dividirse. También puede indicar que el tipo o método podría ser difícil de mantener.

Líneas de código		
Tipo de programación	POA	POO
TOTAL	6293	10543
actividad	123	125
administrador	479	998
autor	184	355
categoría	180	362
ciclo	210	375
configuración	631	901
cuenta	328	532
editorial	197	393
ejemplar	731	946
ejemplar_estado	130	242
ejemplar_ubicación	138	261
escuela	180	384
escuela_usuario	81	203
estadoejemplar	187	357
feriado	166	339
país	175	345
perfildeusuario	233	411
perfildeusuario_has_usuario	110	224
préstamo	593	751
publicación	10	10
publicacion_has_autor	18	18
reserva	185	201

sanción	131	179
tipopublicación	185	379
ubicación	192	346
usuario	516	906

Tabla 11: Cuadro comparativo de cantidad de líneas de código.

Fuente: Elaboración propia.

Programación orientada a aspectos	Líneas de código
Total	1296
Autentificación	152
Autorización	191
Excepciones	469
Looging	99
Validaciones	227
weaver	158

Tabla 12: Tamaño de líneas de código de aspectos.

Fuente: Elaboración propia.

Comparación		
Denominación	POA	POO
CLASES	6293	10543
ASPECTOS	1296	-
TOTAL	7589	10543

Tabla 13: Cuadro resumen comparativo entre la programación orientada a aspectos y objetos.

Fuente: Elaboración propia.

En la tabla del cuadro comparativo de cantidad de líneas de código se observa que el total de líneas de código con la programación orientada a objetos es de 10543 líneas de código y con la programación orientada a aspectos es de 6293 líneas de código, esto corresponde al desarrollo de las clases de tipo controlador ya que es donde se realiza la lógica de operaciones del sistema. Queda por indicar que la cantidad de líneas de código orientadas a objetos corresponden al desarrollo de codificación tanto con la especificación funcional y no funcional, por el contrario en el caso de la programación orientada a aspectos corresponde solo a especificación funcional, lo cual en un primer análisis indica que se está logrando una separación de competencias siguiendo el principio de diseño del software. No obstante los aspectos también ocupan espacios de líneas de código, y haciendo referencia al cuadro de tamaño de líneas de código de aspectos se puede observar un total de 1296 líneas de código en la cual se encapsulan la especificación no funcional o requerimientos del dominio de la aplicación. Finalmente el cuadro resumen comparativo entre la programación orientada a aspectos y objetos, en su resultado total nos da 10543 líneas de código realizados con la

programación orientada a objetos y 7589 líneas de código con la programación orientada a aspectos, lo cual representa un 28% menos, indicando un impacto positivo aplicando aspectos.

El siguiente cuadro nos muestra gráficamente la diferencia entre el uso de objetos y aspectos en la cual se observa una clara tendencia de disminución de líneas de código a favor de la orientación a aspectos.

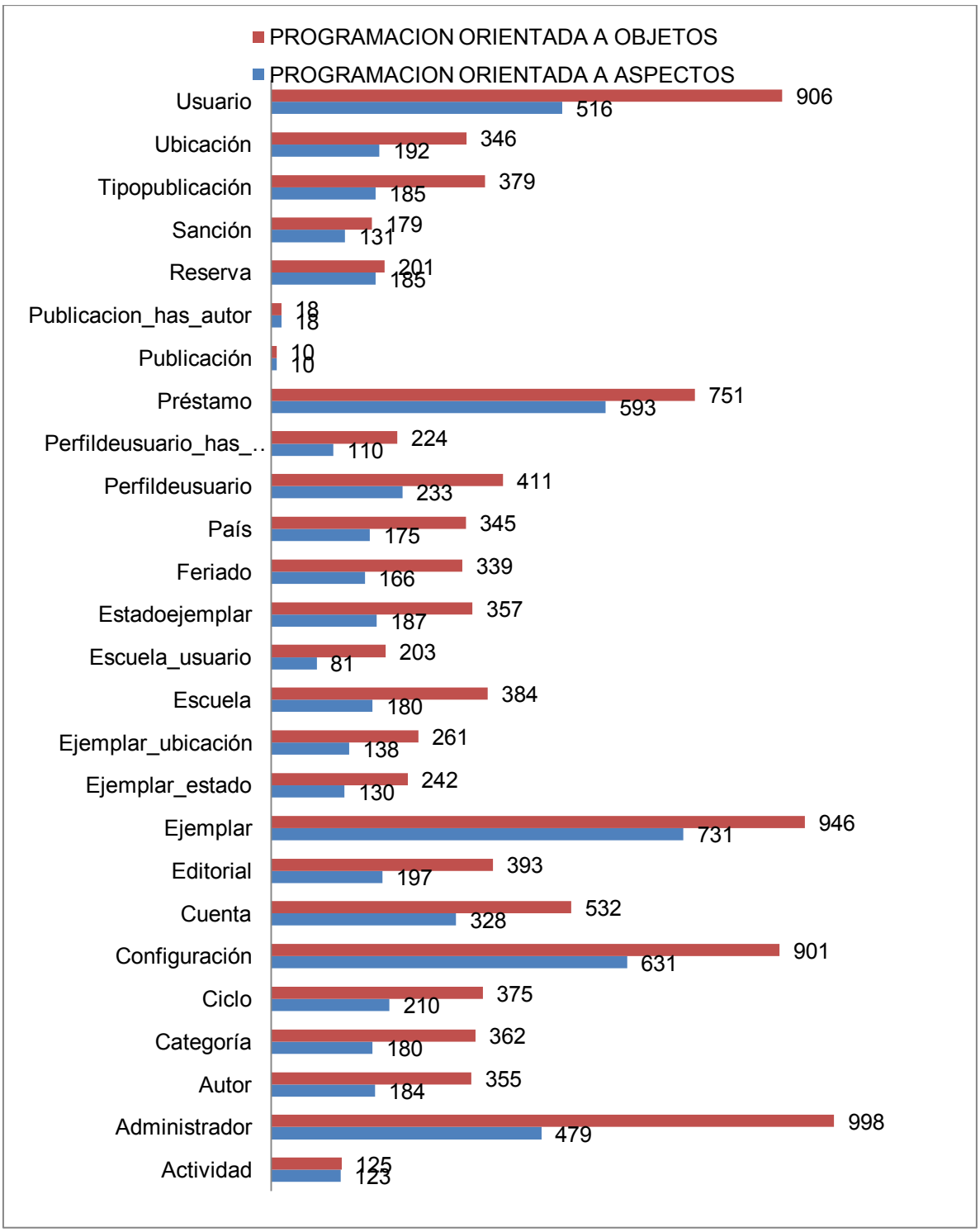


Ilustración 105: Diagrama comparativo en líneas de código.

Fuente: Elaboración propia.

4.2. Resultados obtenidos de la prueba complejidad ciclomática

La complejidad ciclomática calcula el número de puntos en donde se toman decisiones. Se basa en el recuento del número de caminos lógicos individuales contenidos en un programa. Para calcular la complejidad, Thomas McCabe utilizo la teoría y flujo de grafos. Para hallar la complejidad ciclomática, el programa se representa como un grafo y cada instrucción que contiene, un nodo del grafo. Las posibles vías de ejecución a partir de una instrucción (nodo) se representa en el grafo como aristas. Por lo tanto proporciona una medición cuantitativa de la complejidad lógica de un programa.

$$V(G) = e - n + 2p$$

Donde:

V (G) = Complejidad ciclomática.

e = Número de aristas del grafo. Una arista conecta dos vértices si una sentencia puede ser ejecutada inmediatamente después de la primera.

n = Número de nodos del grafo correspondientes a sentencias del programa.

p = Número de componentes conexos, nodos de salida.

Obteniéndose los siguientes resultados que corresponden a un promedio general entre lo realizado con la programación orientada a aspectos y objetos:

Denominación	Programación orientada a aspectos	Programación orientada a objetos
actividad	1.50	1.50
administrador	2.87	7.89
autor	0.88	7.13
categoría	0.88	7.13
ciclo	1.86	8.00
configuración	1.19	4.59
cuenta	1.91	8.60
editorial	0.88	8.38
ejemplar	4.64	10.40
ejemplar_estado	0.20	6.80
ejemplar_ubicación	0.17	5.83
escuela	0.88	8.38

escuela_usuario	0.33	9.33
estadoejemplar	0.78	6.33
feriado	1.00	8.14
país	0.86	8.00
perfildeusuario	1.57	8.71
perfildeusuario_has_usuario	0.25	5.20
préstamo	2.93	8.07
publicación	1.00	1.00
publicacion_has_autor	1.00	1.00
reserva	1.29	7.00
sanción	1.00	4.25
tipopublicación	0.88	8.38
ubicación	0.88	7.13
usuario	1.76	7.76

Tabla 14: Cuadro comparativo de la complejidad ciclomática entre la programación orientada a aspectos y objetos.

Fuente: Elaboración propia.

En el cuadro anterior se puede observar la diferencia de la complejidad ciclomática entre lo realizado con la programación orientada a aspectos y lo realizado con la programación orientada a objetos, en el cual se ve un efecto que la complejidad del código disminuye entre cada clase comparada y en algunos casos una caída sustancial en la complejidad. Se ve que esto se debe a que las decisiones lógicas del sistema en la orientación a aspectos están separadas de la funcionalidad principal, por lo tanto aquellas decisiones que tienen que ver con la lógica del sistema se realiza en los aspectos, lo cual apoya el principio de la cohesión en el diseño a nivel de las clases.

Al realizar una comparación general se obtiene 1.28 para la orientación a aspectos y 6.73 para la orientada a objetos en complejidad ciclomática y comparándolos con la tabla de escalas se evidencia que para lo realizado con la orientación a objetos tiene una complejidad muy alta y con lo realizado con la orientación a aspectos se obtiene una poca complejidad.

Escala	Descripción
1 - 2	Poca Complejidad
3 - 4	Complejidad Moderada
5 - 6	Complejidad Alta
6+	Complejidad muy alta

Tabla 15: Tabla de escalas para la complejidad ciclomática.

Fuente: (Pratt, 2013)

La siguiente ilustración refleja la escala de complejidad de cada clase, se demuestra el efecto que tiene la orientación a aspectos en la complejidad del sistema donde hace que baje notoriamente comparada con la orientada a objetos. Se puede decir que entre más complejidad tenga una rutina o un software, más se dificulta su mantenibilidad, la probabilidad de tener errores será mucho más alta e incluso añadir nuevas funcionalidades se volverá más complicado.

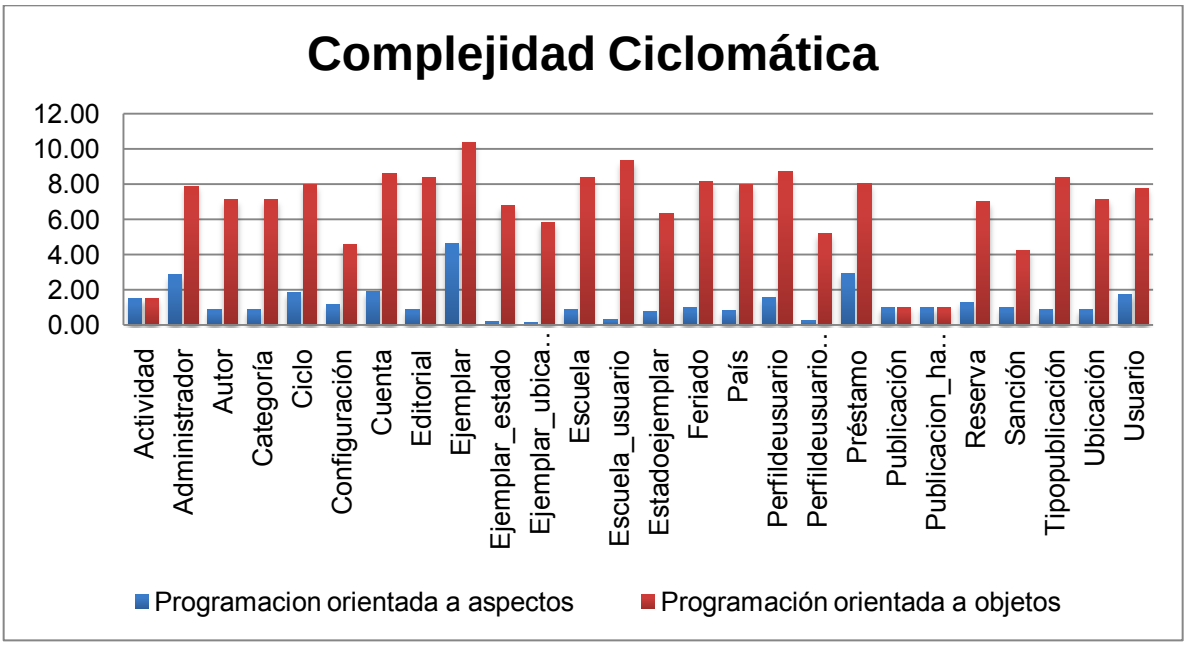


Ilustración 106: Gráfico comparativo entre la programación orientada a aspectos y objetos.

Fuente: Elaboración propia.

La conclusión es que un software resulta más eficiente cuanto menor sea su complejidad, con la orientación a aspectos se logra que a través del diseño usando aspectos la complejidad del software disminuya en comparación con la orientación a objetos.

4.3. Resultados obtenidos de la eficiencia

Para este caso se considera la cantidad de tiempo de ejecución y el consumo de memoria que se logra con cada caso de estudio. Se obtuvieron los siguientes resultados:

Nro.	Programación orientada a aspectos		Programación orientada a objetos	
	Segundos	MB	Segundos	MB
1	0.1406	2.22MB	0.1719	2.18MB
2	0.1406	2.22MB	0.1406	2.18MB
3	0.1406	2.22MB	0.1406	2.18MB
4	0.1563	2.22MB	0.1563	2.18MB

5	0.1406	2.22MB	0.1406	2.18MB
6	0.1563	2.22MB	0.1563	2.18MB
7	0.1406	2.22MB	0.1563	2.18MB
8	0.1406	2.22MB	0.1406	2.18MB
9	0.1406	2.22MB	0.0781	2.18MB
10	0.1406	2.22MB	0.1406	2.18MB
11	0.1406	2.22MB	0.125	2.18MB
12	0.1406	2.22MB	0.1406	2.18MB
13	0.1563	2.22MB	0.1563	2.18MB
14	0.1563	2.22MB	0.1563	2.18MB
15	0.1406	2.22MB	0.1719	2.18MB
16	0.0781	2.22MB	0.1563	2.18MB
17	0.1563	2.22MB	0.1563	2.18MB
18	0.1406	2.22MB	0.1406	2.18MB
19	0.1094	2.22MB	0.1563	2.18MB
20	0.1563	2.22MB	0.0938	2.18MB
21	0.1406	2.22MB	0.1563	2.18MB
22	0.1406	2.22MB	0.1563	2.18MB
23	0.0938	2.22MB	0.1406	2.18MB
24	0.1406	2.22MB	0.1563	2.18MB
25	0.1563	2.22MB	0.1406	2.18MB
26	0.1406	2.22MB	0.1563	2.18MB
27	0.1406	2.22MB	0.1563	2.18MB
28	0.1406	2.22MB	0.1406	2.18MB
29	0.1563	2.22MB	0.1406	2.18MB
30	0.1406	2.22MB	0.1563	2.18MB
31	0.1406	2.22MB	0.1563	2.18MB
32	0.1406	2.22MB	0.1563	2.18MB
	0.14013438	2.22MB	0.1465031	2.18MB

Tabla 16: Prueba de eficiencia de administración de ejemplares en el panel del administrador.

Fuente: Elaboración propia.

Nuestro cuadro anterior en el promedio de tiempo de ejecución en la sección del panel de administración nos muestra un total de 0.14013438 segundos para lo desarrollado con la

programación orientada a aspectos, y un total de 0.1465031 con la programación orientada a objetos, esto indica que si se diseña un software orientado a aspectos la ejecución de esta no incidirá de manera negativa, aunque existe una ligera diferencia. Se puede apreciar gráficamente la comparación de los tiempos de ejecución en la ilustración siguiente:

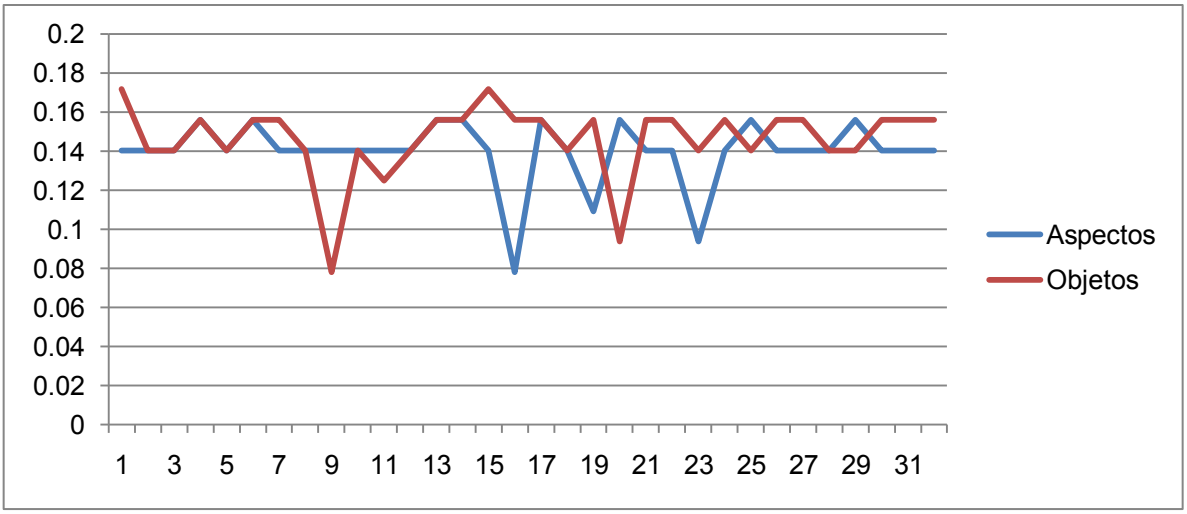


Ilustración 107: Prueba de tiempo de ejecución en la administración de ejemplares en el lado del administrador.

Fuente: Elaboración propia.

En los promedio de consumo de memoria nos da un consumo de 2.22MB en el software orientado a aspectos y 2.18MB para el software orientado a objetos, en este caso el consumo de memoria del software orientado a aspectos es mayor en 0.04MB, este consumo de memoria extra se debe al tejido dinámico que es el tejido que se adoptó para este caso de estudio.

El siguiente cuadro nos muestra los tiempos de ejecución en el lado del cliente donde se realizan las búsquedas de material bibliográfico, se obtiene un promedio de 0.12082 segundos para el software orientado a aspectos y 0.1260267 segundos para el software orientado a objetos.

Nro.	Programación orientada a aspectos		Programación orientada a objetos	
	Segundos	MB	Segundos	MB
1	0.1406	2.3MB	0.1406	2.16MB
2	0.1406	2.3MB	0.1406	2.16MB
3	0.1406	2.3MB	0.1406	2.16MB
4	0.1406	2.3MB	0.1406	2.16MB
5	0.125	2.3MB	0.1406	2.16MB

6	0.1406	2.3MB	0.1406	2.16MB
7	0.125	2.3MB	0.0625	2.16MB
8	0.125	2.3MB	0.125	2.16MB
9	0.1406	2.3MB	0.1406	2.16MB
10	0.125	2.3MB	0.1406	2.16MB
11	0.1406	2.3MB	0.1406	2.16MB
12	0.1406	2.3MB	0.0625	2.16MB
13	0.1406	2.3MB	0.1406	2.16MB
14	0.1406	2.3MB	0.1406	2.16MB
15	0.1406	2.3MB	0.1406	2.16MB
16	0.1406	2.3MB	0.1094	2.16MB
17	0.1406	2.3MB	0.0625	2.16MB
18	0.125	2.3MB	0.1406	2.16MB
19	0.125	2.3MB	0.1406	2.16MB
20	0.0625	2.3MB	0.1406	2.16MB
21	0.1406	2.3MB	0.1406	2.16MB
22	0.1406	2.3MB	0.125	2.16MB
23	0.0625	2.3MB	0.1406	2.16MB
24	0.0625	2.3MB	0.1406	2.16MB
25	0.0781	2.3MB	0.1094	2.16MB
26	0.125	2.3MB	0.125	2.16MB
27	0.125	2.3MB	0.1406	2.16MB
28	0.0625	2.3MB	0.125	2.16MB
29	0.0625	2.3MB	0.0625	2.16MB
30	0.125	2.3MB	0.1406	2.16MB
	0.12082	2.3MB	0.1260267	2.16MB

Tabla 17: Prueba de eficiencia de búsqueda de ejemplares en el lado del usuario

Fuente: Elaboración propia.

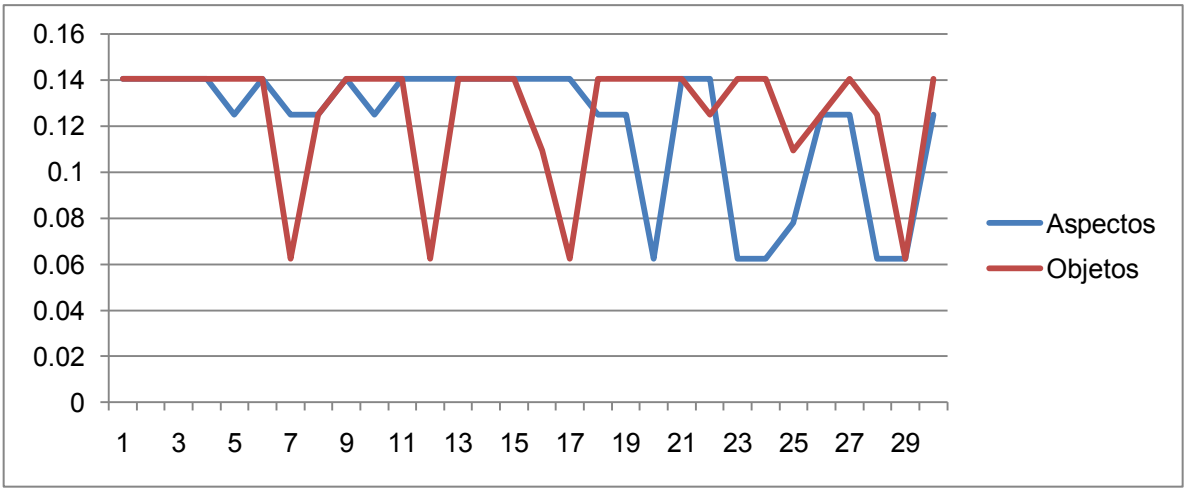


Ilustración 108: Tiempos de ejecución en administración de ejemplares en el lado del administrador

Fuente: Elaboración propia.

Se observa también un consumo de memoria de 2.3MB en el software orientado a aspectos y un consumo de memoria de 2.16MB para el software orientado a objetos, y una diferencia de 0.14MB de consumo extra por parte de la orientación a aspectos.

Finalmente se mantiene que los consumos de memoria son diferentes, por lo tanto el software orientado a aspecto consume más memoria que un software orientado a objetos, así mismo los tiempos de ejecución varían en función de la complejidad de la tarea, y ambos mantienen similares tiempos de ejecución.

4.4. Resultados obtenidos de la mantenibilidad

El resultado de la mantenibilidad es el logro de cortos tiempos de reparación para mantener una alta disponibilidad, de tal manera que sean minimizadas las paradas de los equipos productivos para el control de costos, cuando la disponibilidad es crítica. Se aplicará la técnica del índice de mantenibilidad para cuantificar el grado en que influyen en el mantenimiento, para ello se usará la fórmula de (Welker, y otros, 1995):

IM = 171 - 5.2 * ln(aveV) - 0.23 * aveV(g') - 16.2 * ln (aveLOC) + 50 * sin (sqrt(2.4 * perCM))

Donde:

aveV = Promedio del volumen Halstead por módulo.

aveV(g') = Promedio de complejidad ciclomática.

aveLOC = Promedio de las líneas de código.

perCM = Promedio de porcentaje de líneas de comentarios por módulo.

DENOMINACIÓN	ASPECTOS				OBJETOS			
	LOC	VOLUMEN	IMANT	CC	LOC	VOLUMEN	IMANT	CC
actividad	123	4607.11	78.50	1.50	125	4884	77.94	1.50
administrador	479	17466.43	77.52	2.87	998	33932.6	64.92	7.89
autor	184	6436.5	85.21	0.88	355	10547.31	70.56	7.13
categoría	180	6469.03	85.54	0.88	362	11210.24	69.93	7.13
ciclo	210	7364.81	79.29	1.86	375	12295.19	65.82	8.00
configuración	631	21234.84	85.00	1.19	901	28230.85	76.97	4.59
cuenta	328	11501.05	79.41	1.91	532	17288.34	65.88	8.60
editorial	197	6799.52	83.82	0.88	393	11910.28	68.00	8.38
ejemplar	731	30164.58	60.78	4.64	946	36496.78	52.25	10.40
ejemplar_estado	130	4418.08	82.90	0.20	242	7450.54	68.59	6.80
ejemplar_ubicación	138	4833.47	85.37	0.17	261	8277.89	70.95	5.83
escuela	180	6137.08	85.82	0.88	384	11069.55	68.75	8.38
escuela_usuario	81	2566.92	82.42	0.33	203	5526.59	61.48	9.33
estadoejemplar	187	6631.15	87.34	0.78	357	10872.86	73.02	6.33
feriado	166	5995.99	84.36	1.00	339	10601.79	68.19	8.14
país	175	6478.35	83.14	0.86	345	10707.15	67.89	8.00
perfildeusuario	233	9101.34	76.57	1.57	411	13782.5	63.57	8.71
perfildeusuario_has_usuario	110	3707.87	81.73	0.25	224	6549.8	70.88	5.20
préstamo	593	22946.76	72.62	2.93	751	26313.25	65.43	8.07
publicación	10	39.86	114.30	1.00	10	39.86	114.30	1.00
publicacion_has_autor	18	141.78	98.18	1.00	18	141.78	98.18	1.00
reserva	185	6041.71	82.50	1.29	201	5379.91	73.25	7.00
sanción	131	4440.12	77.79	1.00	179	5463.62	70.90	4.25
tipopublicación	185	6668.45	84.94	0.88	379	11454.01	68.79	8.38
ubicación	192	6582.44	84.41	0.88	346	10195.95	71.15	7.13
usuario	516	20278.51	78.47	1.76	906	30170.3	65.90	7.76

Tabla 18: Tabla comparativa del índice de mantenibilidad. LOC: Líneas de código, IMANT: Índice de mantenibilidad, CC: Complejidad ciclomática. Fuente: Elaboración propia.

En el cuadro anterior se realiza una comparación del efecto de cada tipo de programación en la mantenibilidad en el diseño realizado de un software, que refleja el grado en se puede realizar cambios al sistema, y esto implica el esfuerzo en realizarlo de hacer cambios respectivos, un bajo índice indicaría un sistema poco mantenible. En la ilustración siguiente se observa que en la orientación a aspectos para la mayoría las clases su índice de

mantenibilidad aumenta. Si lo comparamos con el siguiente cuadro de escalas podemos ubicar un promedio

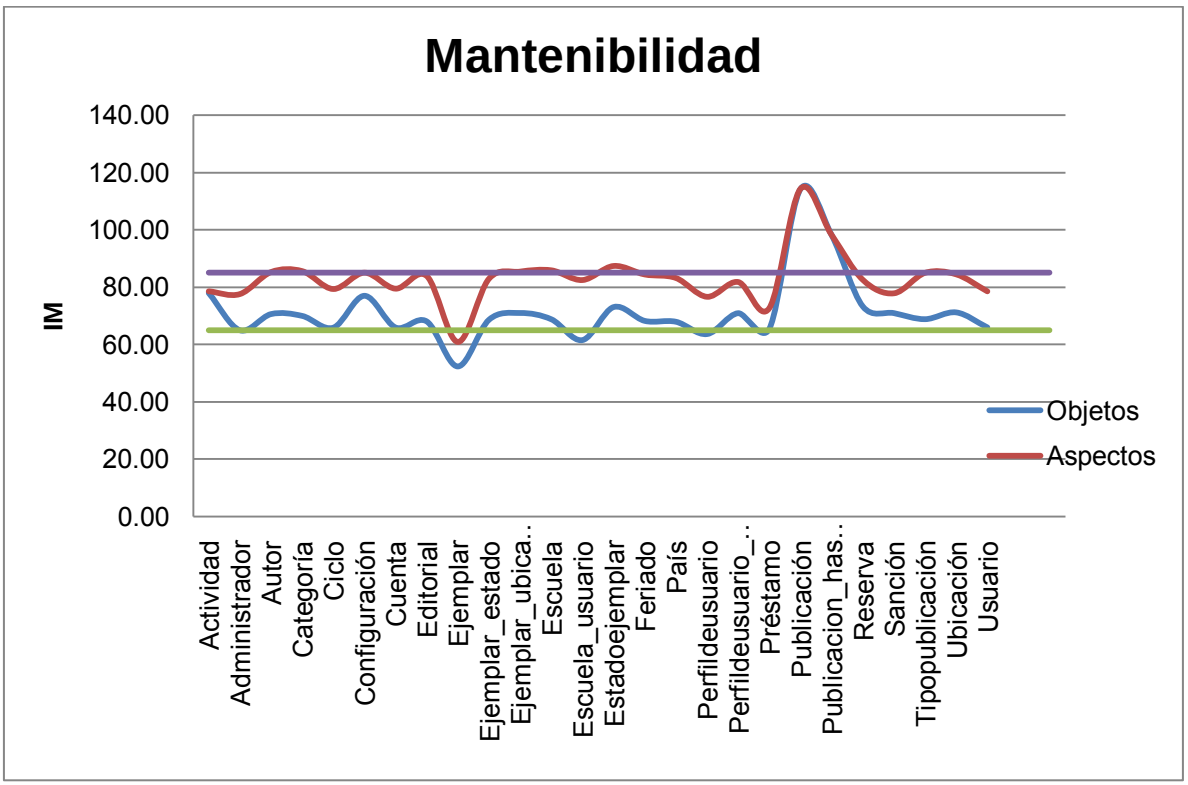


Ilustración 109: Gráfico comparativo de mantenibilidad entre la programación orientada a aspectos y objetos.

Fuente: Elaboración propia.

Cuanto mayor sea el IM, mayor mantenibilidad tendrá el sistema. Finalmente se evalúa que ambos enfoques tienen alto grado de mantenibilidad y están en un promedio aceptable, sin embargo se logra un mayor grado de mantenimiento utilizado la programación orientada a aspectos en el diseño de un software.

5. Conclusiones y recomendaciones

5.1. Conclusión general

- La programación orientada a aspectos influye positivamente en el diseño de un software porque al aplicar la separación de competencias en base a los requisitos del sistema se promueve una clara modularidad, una menor complejidad del sistema, una menor cantidad de líneas de código y un alto grado de mantenibilidad, en comparación con la programación orientada a objetos en el diseño de un software. Con esto en el diseño de un software aplicando la programación orientada a aspectos la efectividad es mejor que la programación orientada a objetos.

5.2. Conclusiones específicas

- Usando la programación orientada a objetos en el diseño de un software se favorece la modularidad, pero al no poder brindar soporte para funcionalidades comunes del sistema hace que la complejidad del software se incremente, por lo tanto no se logra una completa modularidad.

- La programación orientada a aspectos como la programación orientada a objetos buscan crear implementaciones modularizadas. La diferencia se halla en que la primera concibe el diseño del software como una separación multidimensional y la segunda se implementa en un espacio de requerimientos de una sola dimensión.

- La identificación temprana de competencias del sistema de forma separada permite desde los inicios del proceso de desarrollo del software, tener un panorama claro del diseño y una arquitectura adecuada a los requerimientos del software.

5.3. Recomendaciones

- Se plantea la elaboración de métricas orientadas a aspectos, como son la incidencia de los puntos de unión y corte sobre el código funcional, manejando una tabla de escalas que ayuden a verificar cuando se está incurriendo en un uso excesivo a nivel de codificación.

- Se plantea realizar estudios acerca del efecto de aplicar el enfoque de la programación orientada a aspectos en múltiples equipos de desarrollo de software, así como su influencia en las actividades colaborativas entre los equipos de trabajo.

- Realizar estudios de técnicas para identificar aspectos tempranos, o la elaboración de sistemas automatizados que ayuden en esta tarea, para de esta forma ayudar a un buen diseño desde el principio.

Bibliografía

1. **Acosta Lester Rodríguez** Caracterización de la Programación Orientada a Aspectos (POA) [Libro]. - [s.l.] : Editorial Academica Espanola, 2012. - pág. 76.
2. **Aguilar Luis Joyanes** Programación orientada a objetos [Libro]. - Madrid : [s.n.], 1996.
3. **Arismendi Rodrigo Almonacid y Venegas Susan Hernández** Programación Orientada a Aspectos [Informe]. - Concepción : [s.n.], 2008.
4. **Bahit Eugenia** POO y MVC en PHP [Libro]. - 2011.
5. **Barraza Olinda Velarde de [y otros]** Introducción a la programación orientada a objetos [Libro]. - Mexico : Pearson Educación, 2006.
6. **Blanca T. Abraham Zoghbi** Arquitectura Inteligente de Desarrollo de Software [Informe]. - Mérida : [s.n.], 2010.
7. **Booch Grady** Análisis y diseño de orientado a Objetos [Libro]. - 2009.
8. **Booch Grady** Analisis y Diseño Orientado Objetos y Aplicaciones [Libro]. - [s.l.] : Benjamin Cummings, 1994.
9. **Boticki I. Katic, M. y Martin S.** Exploring the Educational Benefits of Introducing Aspect-Oriented Programming Into a Programming Course [Publicación periódica]. - 2013.
10. **Brichau Johan y D'Hondt Theo** Aspect-Oriented Software Development [Publicación periódica]. - 2005.
11. **Brito Isabel Sofia, Moreira Ana y Rita A. Ribeiro João Araújo** Handling conflicts in aspect-oriented requeriments engineering [Publicación periódica]. - Portugal : [s.n.], 2013.
12. **Bruegge Bernd y Dutoit Allen H.** Ingenieria de software orientado a objetos [Libro]. - Munich : Prentice hall, 2002.
13. **Casas Sandra [y otros]** Clasificación y Resolución de Conflictos entre Aspectos [Publicación periódica]. - Argentina : [s.n.], 2005.
14. **Casas Sandra [y otros]** Conflictos entre Aspectos en Etapas del Desarrollo de Software [Publicación periódica]. - 2006.
15. **Cugola Gianpaolo, Ghezzi Carlo y Monga Mattia** Coding Different Design Paradigms for Distributed Applications with Aspect-Oriented Programming [Publicación periódica]. - Milano - Italy : [s.n.], 1999.
16. **DavidA.Watt William Findlay** Programming Language Design Concepts [Libro]. - England : [s.n.], 2010.
17. **Diaz Sergio Carrasco** Metodología de la investigación cientifica [Libro]. - [s.l.] : Editorial San Marcos, 2007.

18. **Echevarría Roberto Rodríguez, Sanchez Encarna Sosa y Ramos Álvaro Prieto** Programación orientada a objetos [Libro]. - 2004.
19. **Elhaibe Rodrigo H** Metodologías Ágiles y Programación orientada a Aspectos Estudio de una posible integración [Publicación periódica]. - Buenos aires : [s.n.], 2006.
20. **Elmasri Ramez y Navathe Shamkant B.** Sistemas de bases de datos [Libro]. - México : Addison-Wesley Iberoamericana, 1992. - Segunda edición.
21. **Erdody Diego M.S.** Un framework para la visualización de patrones de diseño distribuidos y concurrentes implementados con programación orientada a aspectos [Informe]. - 2006.
22. **Fernando Asteasuain Bernardo Ezequiel Contreras** Programación orientada a aspectos - Análisis del paradigma [Informe]. - Buenos aires : [s.n.], 2002.
23. **González Yunia Reyes, Bravo Lisbet M. González y Constanten Yadira Ruiz** Estudio comparativo sobre las principales metodologías pesadas y orientadas a objeto en el desarrollo de software [Publicación periódica]. - 2008.
24. **Guzman Martin [y otros]** Metodologias para analisis y diseño orientado a objetos y MDA [Publicación periódica] // Scribd. - 2007. - págs. 3-13.
25. **Hormazábal Carlos Sebastián Cabrera** Control de reentrancia de aspectos en aspectj [Informe]. - Santiago de Chile : [s.n.], 2010.
26. **Ibáñez María José Ortín y Molina Jesús García** Modelado basado en roles con UML [Publicación periódica]. - 2000.
27. **Jacobson Ivar y Ng Pan-Wei** Aspect-Oriented Software Development with Use Cases [Libro]. - [s.l.] : Addison Wesley Professional, 2004.
28. **Joskowicz José** Programación orientada a aspectos [Informe]. - 2008.
29. **Kendall Kendall &** Análisis y Diseño de Sistemas [Libro]. - México. : Pearson Education, 2005.
30. **Keuler Thorsten** An Aspect-oriented approach for improving architecture design efficiency [Libro]. - Kaiserslautern : [s.n.], 2011. - Vol. 33.
31. **Kiczales Gregor** [En línea]. - 15 de 05 de 2014. - http://www.cs.ubc.ca/~gregor/.
32. **Kruchten Philippe** Architectual Blueprints – the “4+1” View Model of Software Architecture [Publicación periódica]. - 1995.
33. **Laddad Ramnivas** [En línea] // Separate software concerns with aspect-oriented programming. - 18 de 01 de 2002. - 11 de 05 de 2014. - http://www.javaworld.com/article/2073918/core-java/i-want-my-aop---part-1.html?page=2.
34. **laddad Ramnivas** AspectJ in Action [Libro]. - 2009.
35. **Lorena Aramayo Adriana y Ignacio Rossi Joaquín** Una herramienta para identificar crosscutting concerns a través de LEL [Informe]. - 2013.

36. **Marcano Carlos Guillermo** Sistemas de informacion [En línea]. - 08 de 05 de 2011. - 13 de 05 de 2014. - http://sistemasdeinformacion2.wikispaces.com/OMT.
37. **Martinez Abdel** Java.net [En línea] // The Source for Java Technology Collaboration. - 20 de 01 de 2014. - 09 de 05 de 2014. - https://weblogs.java.net/blog/potty/archive/2014/01/20/introduction-object-oriented-programming-oop-part-i.
38. **Murillo Karen Rocío Jimenez** Propuesta de metodología y estándares para la administración de proyectos en las pequeñas y medianas empresas de software con base en los estandares del pmi [Informe]. - San José, Costa Rica : [s.n.], 2012.
39. **Netinant** Separation of concerns for Multithreads Object-Oriented Programming [Publicación periódica]. - [s.l.] : IEEE, 2009. - Vol. II.
40. **Pratt Michael** Pratt [En línea] // La Complejidad Ciclomática en PHP. - 12 de 07 de 2013. - 05 de 05 de 2014. - http://www.michael-pratt.com/blog/15/La-Complejidad-Ciclomatica-en-PHP/.
41. **Pressman Roger S.** Ingeniería del software - Un enfoque Practico [Libro]. - 2009. - 6.
42. **Quintero Antonia Maria Reina** Vision general de la programación orientada a aspectos [Informe]. - Sevilla : [s.n.], 2000.
43. **Quispe Griseldo Platon Palomino** Investigacion cualitativa y cuantitativa en educación [Libro]. - Puno : FCEDUC, 2008. - Tercerca edición.
44. **Rago Alejandro Miguel y Marcos Claudia** Análisis semántico para la identificación de aspectos [Conferencia]. - Tandil : [s.n.], 2008.
45. **Rational Software** Rational Unified Process [Informe]. - 2003.
46. **Rojas Diego** IComparable [En línea]. - 21 de 10 de 2008. - 2014. - http://icomparable.blogspot.com/2008/10/arquitectura-n-tier-o-arquitectura-n.html.
47. **Romero Guillen Paola** Metodos y Modelos [Sección del libro] // Analisis y Diseño Orientado a Objetos. - Laguna : Instituto Tecnologico de la Laguna, 2005.
48. **Sampieri Roberto Hernández, Fernández-Collado Carlos y Lucio Pilar Baptista** Metodología de la investigación [Libro]. - Itzapalapa - México : [s.n.], 2006.
49. **Sánchez Pablo [y otros]** Mapping Aspects from Requirements to Architecture [Publicación periódica]. - 2013.
50. **Sardinha Alberto [y otros]** Conflict identification with EA-Analizer [Publicación periódica]. - Lisboa : [s.n.], 2013.
51. **Schmuller Joseph** Aprendiendo UML en 24 horas [Libro]. - [s.l.] : Pearson, 2003.
52. **Sommerville Ian** Ingenieria de software [Libro]. - [s.l.] : Pearson, 2008.
53. **Sommerville Ian** Software Engineering 9th [Libro]. - 2011.
54. **Spinczyk Ute y Spinczyk Olaf** Using AspectC++ for Qt Application Development [Publicación periódica]. - 2011.

55. **Steinmacher Igor Fábio** Estudo de Princípios para Modelagem Orientada a Aspectos [Publicación periódica]. - Paraná : [s.n.], 2003.
56. **Tabares Marta Silvia, Alférez Salinas Germán y Alferez Salinas Edward** El desarrollo de software orientado a aspectos: un caso practico para un sistema de ayuda en linea [Publicación periódica]. - 2008.
57. **Tarr P. Ossher H. Harrison, W. y S.M. Sutton** N Degrees of separation: Multi-Dimensional Separation of Concerns [Publicación periódica]. - 1999.
58. **Tovilla Juan Carlos Sarmiento** Un paradigma proactivo orientado a objetos [Informe]. - México : [s.n.], 2009.
59. **University San José state** [En línea]. - 2014. - 04 de 06 de 2014. - http://www.cs.sjsu.edu/~pearce/modules/lectures/ooa/analysis/ecb.htm.
60. **Vidal Cristian L. [y otros]** Extensión del Diagrama de Secuencias UML (Lenguaje de Modelado Unificado) para el Modelado Orientado a Aspectos [Publicación periódica]. - Chile : [s.n.], 2012.
61. **Welker K.D y Oman P.W.** Software Maintainability Metrics Models in Practice [Publicación periódica]. - 1995.
62. **Weston Nathan [y otros]** Aspect Identification in Textual Requirements with EA-Miner [Libro]. - 2013.
63. **William Harrison Harold Ossher** Subject-Oriented Programming (A Critique of Pure Objects) [Publicación periódica]. - 1998.
64. **Xerox** The Home of AspectC++ [En línea]. - 08 de 04 de 2014. - 12 de 05 de 2014. - http://www.aspectc.org/.
65. **Zhang Jingjun [y otros]** Using Sequence Diagram to Support Aspect-Oriented Programming in MDA [Publicación periódica]. - 2009.
66. **Zhu Hong** Software Design Methodology [Libro]. - 1995.

Anexos

Anexo I : Tabla de autorizaciones

En esta tabla se incluyen las clases y la correspondencia de accesos a determinadas funciones en cada clase de cada tipo de usuario que interactúa con el sistema.

USUARIO			
USUARIO	BIBLIOTECARIO	ATENCIÓN	ADMINISTRADOR
agregar_carrito			
remover_item			
lista_carrito			
cambiar_clave			
historial			
solicitar_reserva			
mis_reservas			
cancelar_reserva			
	index	index	
	registrar	registrar	
	registro_ajax	registro_ajax	
	actualizar		
	eliminar		
	buscar	buscar	
	query	query	
ubicación			
	index		
	registrar		
	registro_ajax		
	actualizar		
	eliminar		
	buscar		
	query	query	
tipopublicación			
	index		
	registrar		
	registro_ajax		
	actualizar		
	eliminar		
	buscar		
	query		

sanción			
	index	index	
	buscar	buscar	
	query	query	
reserva			
	index	index	
	reservas_recientes	reservas_recientes	
	eliminar		
	json	json	
publicacion_has_autor			
	index		
publicación			
	index		
préstamo			
	index	index	
	registrar	registrar	
	retraso	retraso	
	devolución	devolucion	
	buscar	buscar	
	query	query	
	reporte_pdf		
	reportes		
perfildeusuario_has_usuario			
	index		
	registrar		
	json	json	
perfildeusuario			
	index		
	registrar		
	registro_ajax		
	actualizar		
	buscar		
	query		
país			
	index		
	registrar		
	registro_ajax		
	actualizar		
	buscar		
	query		
feriado			
	index		
	registrar		

	actualizar		
	eliminar		
	buscar		
	query		
estadoejemplar			
	index		
	registrar		
	registro_ajax		
	actualizar		
	eliminar		
	query		
	buscar		
	json		
escuela_usuario			
	index		
	registrar		
	json	json	
escuela			
	index		
	registrar		
	registro_ajax		
	actualizar		
	eliminar		
	buscar		
	query		
ejemplar_ubicación			
	index		
	mover		
	query		
	detalle		
	json		
ejemplar_estado			
	index		
	cambiar		
	query		
	json		
ejemplar			
	index		
	registrar		
	copiar		
	actualizar		
	buscar		
búsqueda			

listado	listado		
consulta	consulta		
	query		
editorial			
	index		
	registrar		
	registro_ajax		
	actualizar		
	eliminar		
	buscar		
	query		
cuenta			
	index		
saldo			
	detalle_saldo	detalle_saldo	
	registrar	registrar	
	eliminar		
	buscar		
	query		
configuración			
	query		
	index		
error	error	error	error
	exportar_backup_completo		exportar_backup_completo
	exportar_backup_incremental		exportar_backup_incremental
	importar		importar
	importar_backup		importar_backup
	exportar_backup		exportar_backup
	eliminar_backup		eliminar_backup
	listar_backup		listar_backup
	registrar		registrar
	actualizar		actualizar
	eliminar		eliminar
faq	faq	faq	faq
ciclo			
	index		
	registrar		
	registro_ajax		
	actualizar		
	buscar		
	query		
categoría			

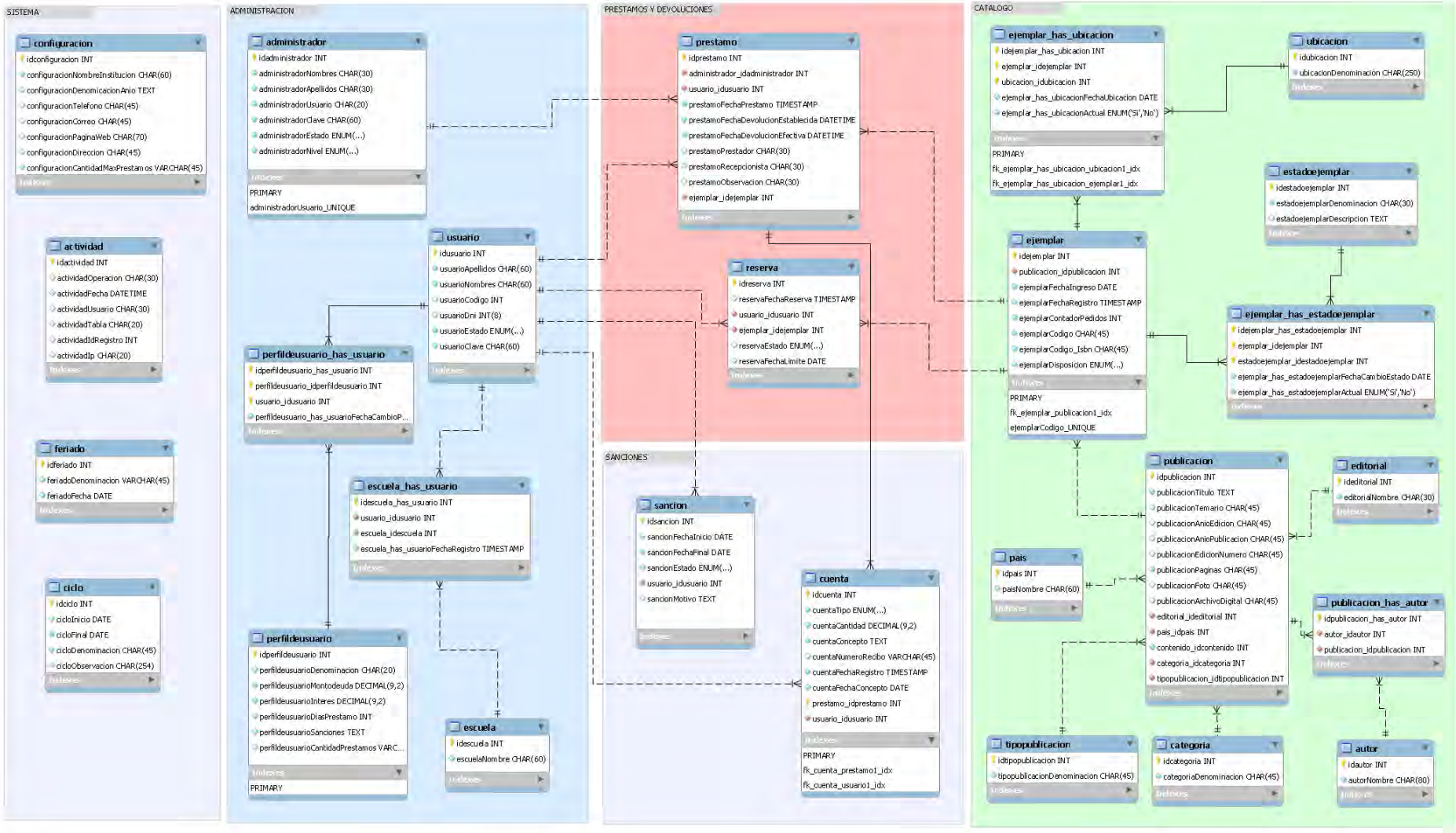
	index		
	registrar		
	registro_ajax		
	actualizar		
	eliminar		
	buscar		
	query		
autor			
	index		
	registrar		
	registro_ajax		
	actualizar		
	eliminar		
	buscar		
	query		
administrador			
	cambiar_clave	cambiar_clave	cambiar_clave
	redireccionar	redireccionar	redireccionar
	cerrar	cerrar	cerrar
	prestar_reserva	prestar_reserva	
	prestar_ejemplar	prestar_ejemplar	
			index
			registrar
			actualizar
			eliminar
			buscar

Anexo II: Matriz de consistencia

PROBLEMA	OBJETIVOS	HIPÓTESIS	VARIABLES E INDICADORES
PROBLEMA GENERAL: ¿Cuál es el efecto de la programación orientada a aspectos y objetos en el diseño de un software? PROBLEMAS ESPECÍFICOS: ¿Cuál es el efecto de la programación orientada a aspectos en el diseño de un software? ¿Cuál es el efecto de la programación orientado a objetos en el diseño de un software?	OBJETIVO GENERAL: Comparar el efecto de la programación orientada a aspectos y objetos en el diseño de un software OBJETIVOS ESPECÍFICOS: Determinar el efecto de la programación orientada a aspectos en el diseño de un software. Determinar el efecto de la programación orientada a objetos en el diseño de un software.	HIPÓTESIS GENERAL: En el diseño de un software en la programación orientada a aspectos la efectividad es mayor que la orientada a objetos. HIPÓTESIS ESPECÍFICAS: En el diseño de un software en la programación orientada a aspectos el efecto es favorable. En el diseño de un software en la programación orientada a objetos el efecto es desfavorable.	VARIABLE INDEPENDIENTE Programación orientada a aspectos Programación orientada a objetos Indicadores Tamaño (LDC) VARIABLE DEPENDIENTE Diseño de un software Indicadores Complejidad Ciclomática Eficiencia Mantenibilidad

MÉTODO Y DISEÑO	POBLACIÓN Y MUESTRA	TÉCNICAS E INSTRUMENTOS	
MÉTODO: Experimental DISEÑO Con grupos experimentales y pos prueba únicamente.	POBLACIÓN: Bibliotecas de universidades públicas MUESTRA: Biblioteca de la Universidad Nacional de Moquegua sede-Ilo GRUPOS: - Programación orientada a aspectos. - Programación orientada a objetos.	TÉCNICAS Observación Entrevistas Medición	INSTRUMENTOS Ficha de registro Guion de entrevista Ficha de medición

Anexo III: Diagrama de la base de datos del sistema



Anexo IV: Descripción textual del sistema

Título: Sistema de biblioteca
Descripción: El sistema pretende realizar el control de actividades de préstamos, devoluciones, sanciones, cuentas, administración de ejemplares y usuarios.
El préstamo se realiza a estudiantes, docentes y personal administrativo que tienen datos como son nombres, apellidos, D.N.I., código. Se pretende que ellos puedan realizar reservas en línea a través de un sistema de búsqueda de libros existentes en la biblioteca, y que puedan identificarse en el sistema por este medio con una clave y contraseña. El estudiante pertenece a una escuela o una categoría en la cual solo va la denominación, también puede cambiarse de escuela y llevar un historial, también llevar un perfil de usuario e ir cambiándose según sea el caso y siempre llevando un historial de cambios. El préstamo es el acto de realizar la entrega de un libro bajo la condición de una entrega según el reglamento de dos días para estudiantes, se debe establecer una fecha de préstamo, una fecha de devolución, la persona quien realiza el préstamo, la persona quien recibe la devolución y una observación en caso que exista en la devolución. Un préstamo se podría dar de dos formas: realizar el préstamo a través de una reserva realizada por un usuario, o un préstamo de manera más personal, cuando se realiza el préstamo el libro no debe quedar disponible para nadie más, de esta manera se garantiza que el sistema no haga préstamos de libros prestados. Una reserva se realiza por un usuario que se identifica en el sistema, este tiene que estar registrado y además estar activo, quiere decir que no tenga deuda o sanción en curso, un paso habitual seria que el usuario seleccione los ítems deseados después de una búsqueda y realice las reservas que le fueren permitidas de acuerdo a un tipo de usuario o perfil que incluya un tipo, una deuda, los días de préstamos permitidos, las sanciones enumeradas en días, y la cantidad de préstamos permitidos para ese perfil. Los días de préstamo se cuentan en días hábiles de estudio, por lo tanto los días feriados y días especiales no se consideran como día de préstamo, por ejemplo si se presta un jueves tendría que devolverse un lunes si el préstamo es para un estudiante al que se le puede prestar dos días. Sobre las sanciones según el reglamento para estudiantes se realizar por días, y en cada falta se va aumentando el número de días sancionado hasta ser sancionado durante todo el ciclo académico. Terminado el ciclo académico el usuario queda habilitado y podrá usar el servicio de la biblioteca Un ciclo académico tiene una denominación, su fecha de inicio y su fecha final, y alguna observación si fuera necesaria. Sobre la administración de los libros llámese ejemplares que tiene datos como: fecha de ingreso (a la institución), fecha de registro (en el sistema), contador de pedidos, código (código interno), código ISBN, el título, el temario, el año de edición, el año de publicación, el número de edición, el número de páginas, y la disposición que sea prestado, disponible. Su administración los realiza el bibliotecario, este puede tener distintos niveles de acceso para cada tarea específica, puede haber un bibliotecario que tenga acceso a todas las secciones y uno exclusivo a los préstamos y devoluciones

sin acceso a ninguna otra funcionalidad. El ejemplar tiene una ubicación que contenga la información de una denominación, y además que se pueda llevar un cambio de ubicación a través de un historial de cambios que registre la fecha del cambio, también tener un estado del ejemplar que incluya su denominación y descripción, que puede variar según sea por conveniente en su categorización y a su vez llevar un historial de cambios en el cual se establezca la fecha de cambio. Un ejemplar puede tener autores, la pertenencia a un país, su tipo de publicación, la categoría a la que pertenece, y a la editorial.

Para que se pueda ingresar al sistema se debe registrar administradores que tengan datos como nombres, apellidos, un usuario (que no se repita), una clave, un estado y un nivel (administrador, bibliotecario, atención).

Para cualquier operación de administración de las partes del sistema el administrador debe haberse identificado en el sistema de biblioteca.

Para cualquier cambio que se realice debe tener un registro que diga quien realizo el cambio y que se vea en un registro que tenga datos como la operación que se realizó, la fecha, el usuario quien realizó dicha operación, la sección afectada y el registro al cual se hace referencia.

Los pagos o deudas de un usuario pueden tener los siguientes datos un tipo si es cargo o abono la cantidad, el concepto del pago, el número de recibo (este no se repite), la fecha de registro y la fecha de concepto (cuando se hace el pago). Se registran por cada día incumplido en la entrega de un ítem, y solo son efectivos esos cargos en días hábiles de estudio. Una vez cancelados se puede sancionar o habilitar al usuario según sea su perfil de usuario.

Las sanciones que pueden tener datos como una fecha de inicio, una fecha final, un estado cumplido o en curso, se dan cuando un usuario incumple la entrega de un ejemplar en la fecha indicada, y se hace efectiva cuando se realiza el pago por los días incumplidos de acuerdo a su perfil de usuario y la devolución efectiva de todos los ítems que tenga en su dominio.

Se desea también que se registre el nombre de la institución, el nombre del año, el teléfono, un correo institucional, una dirección web y una dirección física.