ESCUELA POLITÉCNICA NACIONAL

FACULTAD DE INGENIERÍA DE SISTEMAS

PROPUESTA DE UN MÉTODO PARA LA IMPLEMENTACIÓN DE SISTEMAS SOFTWARE

PROYECTO PREVIO A LA OBTENCIÓN DEL TÍTULO DE INGENIERO EN SISTEMAS INFORMÁTICOS Y DE COMPUTACIÓN

CEVALLOS RAZA ALEJANDRO JAVIER
alexcevallos197@hotmail.com

DIRECTOR: MSC. ING. CÓRDOVA BAYAS MARCOS RAÚL
raul.cordova@epn.edu.ec

Quito, Mayo 2014

DECLARACIÓN

Yo Alejandro Javier Cevallos Raza, declaro bajo juramento que el trabajo aquí descrito es de mi autoría; que no ha sido previamente presentada para ningún grado o calificación profesional; y, que he consultado las referencias bibliográficas que se incluyen en este documento.

A través de la presente declaración cedo mis derechos de propiedad intelectual correspondientes a este trabajo, a la Escuela Politécnica Nacional, según lo establecido por la Ley de Propiedad Intelectual, por su Reglamento y por la normatividad institucional vigente.

Alejandro Javier
Cevallos Raza

CERTIFICACIÓN

Certifico que el presente trabajo fue desarrollado por Alejandro Javier Cevallos Raza, bajo mi supervisión.

_____

Msc. Ing. Raúl Córdova
DIRECTOR DEL PROYECTO

AGRADECIMIENTO

Agradezco al Señor y Dios de mi vida por haberme dado la victoria después de transcurridos 2 años en este proyecto de titulación y no lo hubiera logrado en verdad sin Sus ideas, Sus revelaciones para este proyecto, su Fuerza infundida en mi vida, Su gracia en medio de no saber de este tema que es su habilidad en mi para después abrirme paso en mi propósito en la vida para ayudar a muchos otros.

Agradezco a mis padres debido a que después de mucho tiempo me animaron a seguir adelante a pesar de desanimarme por otros que lo lograron antes que yo y siguieron creyendo para que llegue el ansiado día en que ellos vean esta victoria en mi vida.

Agradezco a José María Arévalo por su guía en la Palabra ya que mediante esta pude saber el camino adonde tenía que ir en este proyecto por lo que su guía fue fundamental en este logro.

Agradezco a mi mejor amigo Edison Quisigüiña por su amistad brindada a lo largo de este tiempo, por escuchar y por animarme a seguir adelante para lograr esta victoria que ahora se ve en estos días.

Agradezco a mis amigos cercanos a mi vida y los que estuvieron en su tiempo cerca ya que no hubiera logrado esto si no fuera por sus oraciones, sus palabras de ánimo que me infundieron la fuerza necesaria para lograr esto.

Agradezco al Ing. Raúl Córdova por su tiempo prestado en revisar este proyecto, por su guía y por su paciencia a mi persona, la verdad agradezco que me haya esperado lo suficiente para que esto sea posible, por eso, estoy agradecido.

Agradezco al Ing. Samaniego por siempre exhortarme a terminar lo que ya había empezado y animarme a pesar de las dificultades de terminar este proyecto.

Por último, agradezco a la gente que participó en la encuesta de desarrollo ya que su participación fue importante en la realización del capítulo 1, gracias a ustedes por su tiempo.

DEDICATORIA

Dedico a mi Señor y mi Dios este Proyecto de Titulación, que esto sirva conforme a Su propósito, que pueda ser útil en Sus manos, que sea un tributo de ofrenda para Él y que pueda ser de mucha utilidad a todos los que lo leen.

Dedico este proyecto a toda la gente que piensa que puede mejorar su estilo de implementación cuando se realiza este proceso mediante la programación en el lenguaje que se use, tampoco importando si se implementa sobre una computadora de escritorio, un servidor web o sobre dispositivos móviles, la verdad muchos piensan que el diseño solo debe quedar en diagramas y plantillas y que solo importa la tecnología en la que se implementa, pero en este proyecto se dedica a también a esta gente para que se vea que el diseño puede y debe formar parte de la implementación del sistema de tal forma que el diseño corrobore lo que se ha implementado o lo corrija, de todas formas, no sirve de nada la tecnología sino se construye (en este caso se implementa) sobre una solida base de diseño que participe en la implementación por lo que se debe involucrar esto y debe involucrar que el desarrollador o desarrolladores prueben su código, por eso se hace indispensable desarrollar por pruebas para lograr robustez en el código implementado ya que con ello se desarrollará conforme a lo esperado por el diseño y se podrá ver toda la gama del comportamiento de cada rutina a lo largo del proceso de implementación.

Se concluiría que este proyecto está dedicado a aquellos que quieren mejorar su estilo de desarrollo en sus implementaciones futuras marcando una mejor tendencia de desarrollo que a largo plazo puede ser usado como una buena práctica si se la realiza.

CONTENIDO

INDICE DE FIGURAS

INDICE DE TABLAS

RESUMEN

El presente proyecto plantea una propuesta para la elaboración de un método para la implementación de sistemas software dentro del proceso de ciclo de vida del software. La propuesta abarca la investigación de la forma de implementación del software, la elaboración del método junto con sus respectivas plantillas y de ahí la aplicación de este método a un caso de estudio particular.

Este proyecto se divide en cuatro capítulos en los cuales se describe detalladamente el proceso de implementación partiendo del diseño hasta obtener la implementación de tal diseño.

En el Capítulo I se describen la situación actual del proceso de implementación, sus fundamentos, su gestión y consideraciones prácticas antes de implementar. Se abarca la selección de la población objetivo, la encuesta, los resultados de la encuesta y sus conclusiones. Así mismo se toma en cuenta aspectos como la minimización de la complejidad, implementar para verificar, medición de la implementación, codificación y pruebas de implementación, entre otros aspectos.

En el capítulo II se comienza a elaborar el método de implementación y se detalla en lo que se ha basado. Este capítulo abarca el método desde el modelo de implementación hasta las pruebas de integración. Se muestra también las metodologías, estándares y herramientas que fueron utilizadas en el método.

En el capítulo III se plantea un caso de estudio en el que se ponga a prueba el método que se ha propuesto en el capítulo II. El caso de estudio es de desarrollo de la implementación del ‘Sistema de Préstamos de los Laboratorios del DICC’, el mismo que se ha realizado con un enfoque de implementación basado en el Desarrollo Guiado por Pruebas y el Proceso de Programación por Pseudocódigo.

Finalmente, en el Capítulo IV se presentan las conclusiones y recomendaciones de lo referido al Proyecto de Titulación.

CAPÍTULO 1. IMPLEMENTACIÓN DE SISTEMAS SOFTWARE.

1.1 SITUACIÓN ACTUAL DEL PROCESO DE IMPLEMENTACIÓN DE SOFTWARE.

1.1.1. INTRODUCCION. [7]

Existe una tendencia a identificar el proceso de creación de un sistema informático con la programación, que es cierta cuando se trata de programas pequeños para uso personal, y que dista de la realidad cuando se trata de grandes proyectos.

El proceso de creación de software, desde el punto de vista de la ingeniería, incluye los siguientes pasos:

1) Reconocer la necesidad de un sistema para solucionar un problema o identificar la posibilidad de automatización de una tarea.
2) Recoger los requisitos del sistema. Debe quedar claro qué es lo que debe hacer el sistema y para qué se necesita.
3) Realizar el análisis de los requisitos del sistema. Debe quedar claro cómo debe realizar el sistema las cosas que debe hacer. Las pruebas que comprueben la validez del sistema se pueden especificar en esta fase.
4) Diseñar la arquitectura del sistema. Se debe descomponer el sistema en partes de complejidad abordable.
5) Implementar el sistema. Consiste en realizar un diseño detallado, especificando completamente todo el funcionamiento del sistema, tras lo cual la codificación (programación propiamente dicha) debería resultar inmediata.
6) Implantar (instalar) el sistema. Consiste en poner el sistema en funcionamiento junto con los componentes que pueda necesitar (bases de datos, redes de comunicaciones, etc.)

La ingeniería del software se centra en los pasos de planificación y diseño del sistema, mientras que antiguamente (programación artesanal) la realización de un

sistema consistía casi únicamente en escribir el código, bajo sólo el conocimiento de los requisitos y con una modesta fase de análisis y diseño.

1.1.2. DEFINICIÓN DE LA POBLACION OBJETIVO. [1]

Es la recolección de elementos u objetos que poseen la información buscada por el investigador y acerca de la cual se realizan las deducciones.

La población objetivo debe definirse en términos de elementos, unidades de muestreo, extensión y tiempo. Un elemento es el objeto acerca del cual se desea la información. En una investigación de encuesta, el elemento por lo general es el encuestado. Una unidad de muestreo es un elemento o unidad que contiene el elemento disponible para selección en alguna etapa del proceso de muestreo.

La población objetivo para la selección de personas a encuestar se definió como:
Elementos: Desarrolladores de software (de preferencia pasantes de desarrollo).
Unidades: Personas.
Extensión: Ciudad de Quito.
Fecha: 14/02/2012 – 13/03/2012.

1.1.3. DETERMINACION DEL MARCO DE MUESTREO. [1]

1.1.3.1. Técnicas de Muestreo.

Es importante que la muestra elegida sea representativa para la población en la que está trabajando ya que está extensa. Existen algunas técnicas de muestreo y se clasifican como probabilísticas y no probabilísticas, como se ve en la Figura 1.1.

1.1.3.1.1. Probabilísticas.

Es el procedimiento de muestra en el que cada elemento de la población tiene una oportunidad probabilística fija de ser seleccionada para la muestra.

Las técnicas de muestreo probabilístico se clasifican en:

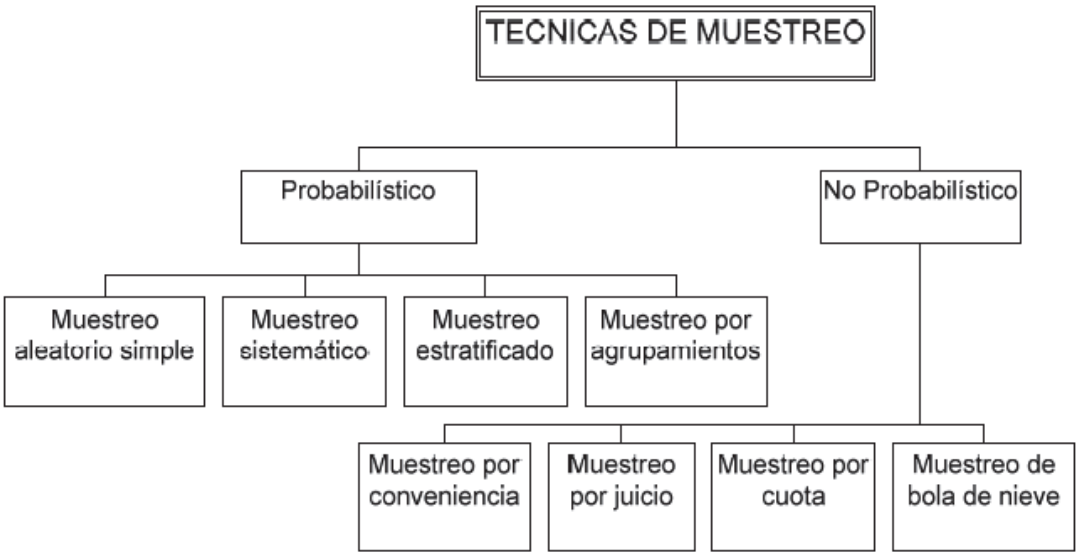


Figura 1.1. Clasificación de las técnicas de muestreo. [2].

Muestreo Aleatorio Simple.

Es la técnica de muestreo probabilístico en el que cada elemento de la población tiene una probabilidad de selección conocida y equitativa. Cada elemento se selecciona en forma independiente a otro elemento y la muestra se toma por un procedimiento aleatorio de un marco de muestreo.

Muestreo Sistemático.

Es la técnica de muestreo probabilístico en la que se elige la muestra al seleccionar un punto de inicio aleatorio y luego se elige cada 'n' elemento en la sucesión del marco de muestreo.

Muestreo por Agrupamientos.

Es la técnica de muestreo probabilístico donde la población objetivo primero se divide en subpoblaciones mutuamente excluyentes y colectivamente exhaustivas llamadas agrupamientos. Luego, una muestra aleatoria de agrupamientos se selecciona con base a una técnica de muestreo probabilístico como el muestreo aleatorio simple. Para cada agrupamiento seleccionado se incluyen ya sea todos los elementos en la muestra o se toma una muestra de elementos en forma probabilística. [2]

La mayoría de los sondeos tiene uno de los tres objetivos: estimar el total poblacional, o estimar la medida de una población, o estimar la proporción poblacional.

1.1.3.1.2. No Probabilísticas.

Son las técnicas de muestreo que no utilizan procedimientos de selección por casualidad. En su lugar, se basa en el juicio personal del investigador.

1.1.4. SELECCIÓN DE TECNICAS DE MUESTREO. [1]

Se refiere al número de elementos que se incluirán en el estudio. Determinar el tamaño de la muestra incluye consideraciones cuantitativas y cualitativas, esto quiere decir que se puede tener un universo finito o un universo infinito.

Por el momento, basándose en el juicio personal del investigador, se ha propuesto extraer una muestra de 20 desarrolladores, de preferencia pasantes de la Escuela Politécnica Nacional.

1.1.5. ELABORACIÓN DE LA ENCUESTA. [1]

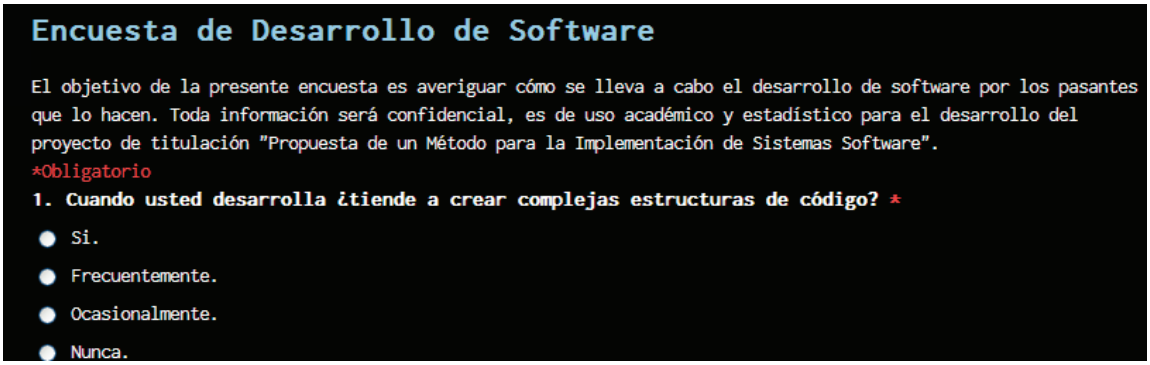


Figura 1.2. Vista previa de la encuesta.

La encuesta fue realizada vía web utilizando los formularios Google provistos de Google Docs, que recolecta la información en la nube y cada registro es una fila en una hoja de cálculo, que solo registra la fecha y hora de la persona que terminó de responder la encuesta, asegurando la confidencialidad para los encuestados. La dirección web de la encuesta es la siguiente: http://tinyurl.com/encuesta-desarrollo-software [15]

1.1.6. ANALISIS DE RESULTADOS DE LA ENCUESTA. [1]

Los resultados que se presentan a continuación son los que se obtuvieron luego de la tabulación de las encuestas realizadas.

La tabla con los resultados se encuentra en el anexo A.

Pregunta 1.

Tipo de Pregunta. Opción múltiple de única respuesta.

Enunciado.

Cuando usted desarrolla ¿tiende a crear complejas estructuras de código?

Resultado.

El 45% desarrolla código de alta complejidad ocasionalmente, el 25% lo hace de forma frecuente, el 15% lo hace y solamente el 15% no lo hace. Así se puede concluir que los desarrolladores están realizando código de baja complejidad.

Si	Frecuentemente	Ocasionalmente	Nunca
3	5	9	3

Tabla 1.1. Resultado de tendencia de implementación de código complejo.

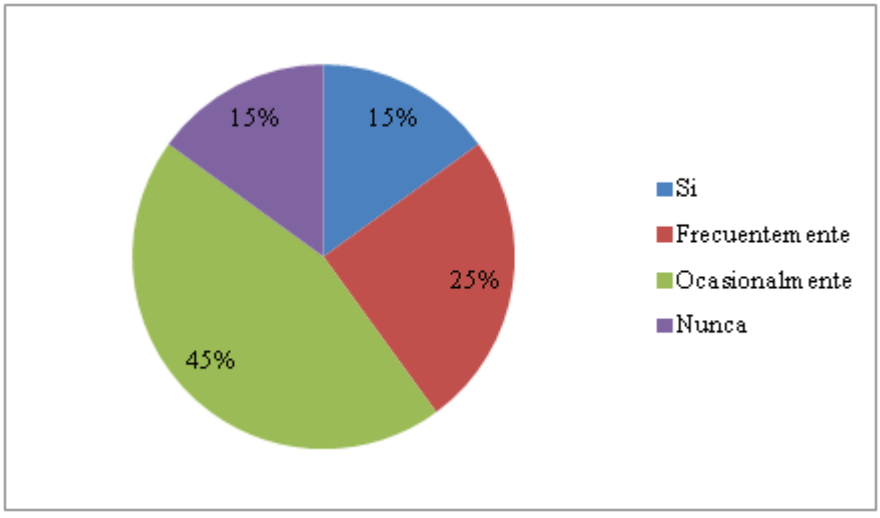


Figura 1.3. Resultado de tendencia de implementación de código complejo.

Pregunta 2.

Tipo de Pregunta. Opción múltiple de única respuesta.

Enunciado.

Cuando usted desarrolla ¿utiliza estándares y técnicas conocidos o se basa en su experiencia personal?

Resultado.

El 35% de desarrolladores utiliza estándares, técnicas y experiencia personal, el 35% solo utiliza estándares y experiencia personal. Solo el 20% utiliza técnicas y experiencia personal mientras que el 10% su experiencia personal. Así se puede concluir que los desarrolladores utilizan estándares, técnicas conocidas y su experiencia personal.

Utilizo estándares, técnicas y mi experiencia personal	7
Utilizo estándares y mi experiencia personal	7
Utilizo técnicas y mi experiencia personal	4
Solo utilizo mi experiencia personal	2

Tabla 1.2. Resultado de tendencia al utilizar estándares, técnicas y experiencia personal.

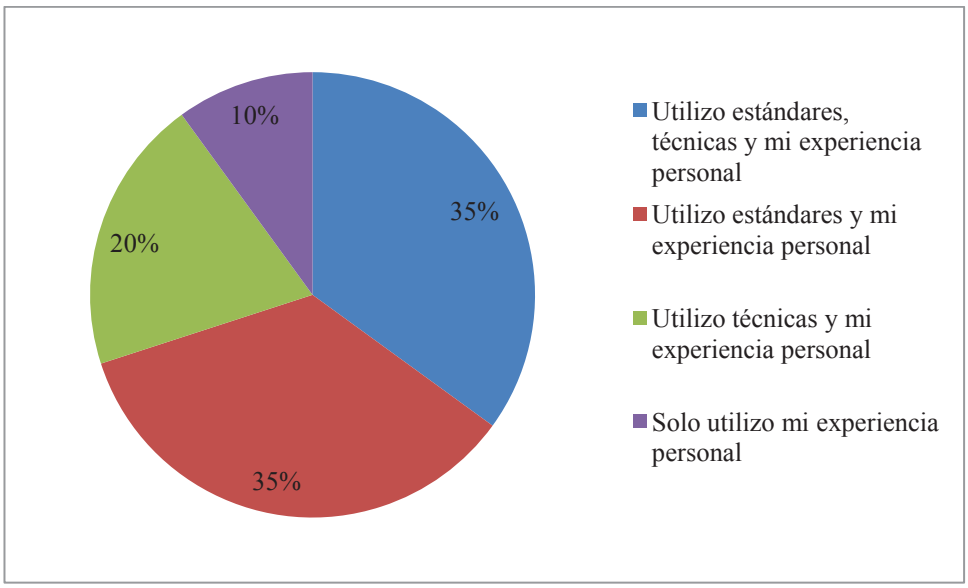


Figura 1.4. Resultado de tendencia al utilizar estándares, técnicas y experiencia personal.

Pregunta 3.

Tipo de Pregunta. Opción múltiple de única respuesta.

Enunciado.

Cuando usted desarrolla ¿contempla la posibilidad de que su código sea alterado?

Resultado.

El 30% de desarrolladores siempre contempla la posibilidad de que su código sea alterado, el 40% de forma frecuente, el 20% de forma ocasional y solo un 10% nunca. Así se puede concluir que los desarrolladores si contemplan la posibilidad de que su código sea alterado.

Siempre	Frecuentemente	Ocasionalmente	Nunca
6	8	4	2

Tabla 1.3. Resultado de posibilidad de alteración al código.

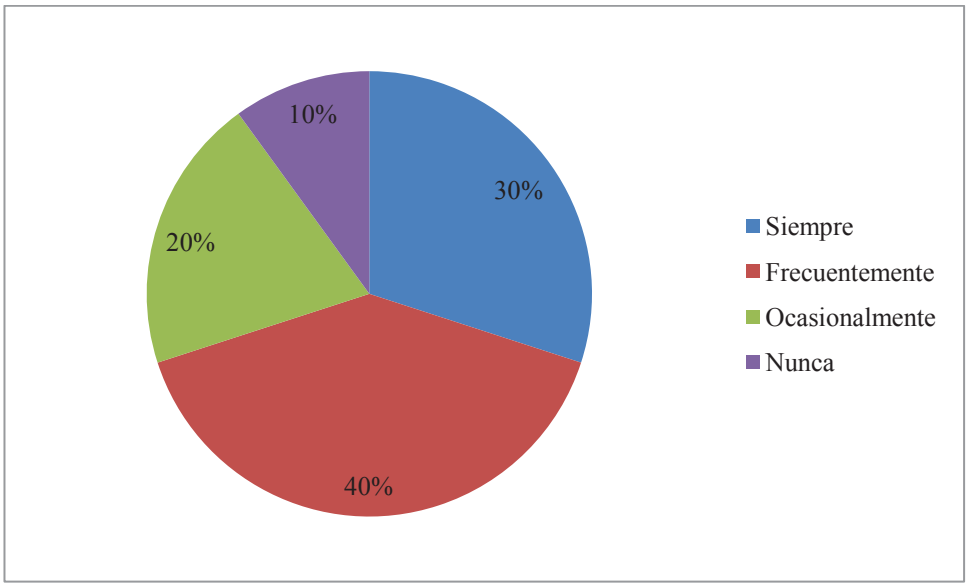


Figura 1.5. Resultado de posibilidad de alteración al código.

Pregunta 4.

Tipo de Pregunta. Opción múltiple de única respuesta.

Enunciado.

En el equipo de desarrollo, ¿se utilizan estándares para los documentos y sus contenidos?

Resultado.

El 15% de desarrolladores siempre utilizan estándares para los documentos y sus contenidos, el 30% de forma frecuente, el 50% de forma ocasional y solo un 5% nunca. Así se puede concluir que ocasionalmente se utilizan estándares para los documentos y sus contenidos.

Siempre	Frecuentemente	Ocasionalmente	Nunca
3	6	10	1

Tabla 1.4. Resultado de utilización de estándares en la documentación.

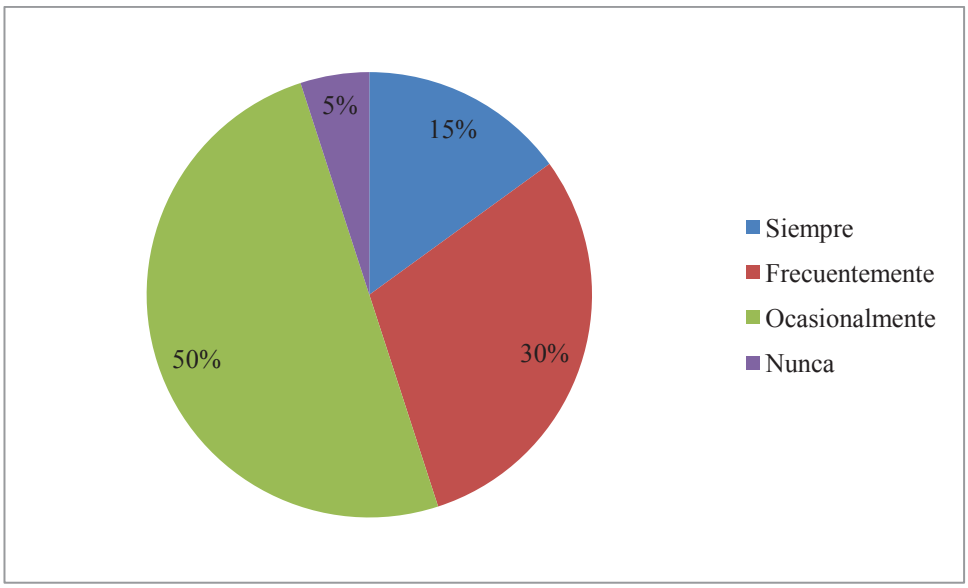


Figura 1.6. Resultado de utilización de estándares en la documentación.

Pregunta 5.

Tipo de Pregunta. Opción múltiple de única respuesta.

Enunciado.

Cuando programa, ¿utilizan estándares de programación?

Resultado.

Solo el 5% de desarrolladores siempre utilizan estándares de programación, el 65% de forma frecuente, el 30% de forma ocasional y 0% nunca. Así se puede concluir que de forma frecuente se utilizan estándares de programación (ver Figura 1.7 y Tabla 1.5).

Pregunta 6.

Tipo de Pregunta. Opción múltiple de única respuesta.

Enunciado.

¿Utiliza estándares para modelar las aplicaciones (como UML por ejemplo)?

Siempre	Frecuentemente	Ocasionalmente	Nunca
1	13	6	0

Tabla 1.5. Resultado de utilización de estándares de programación.

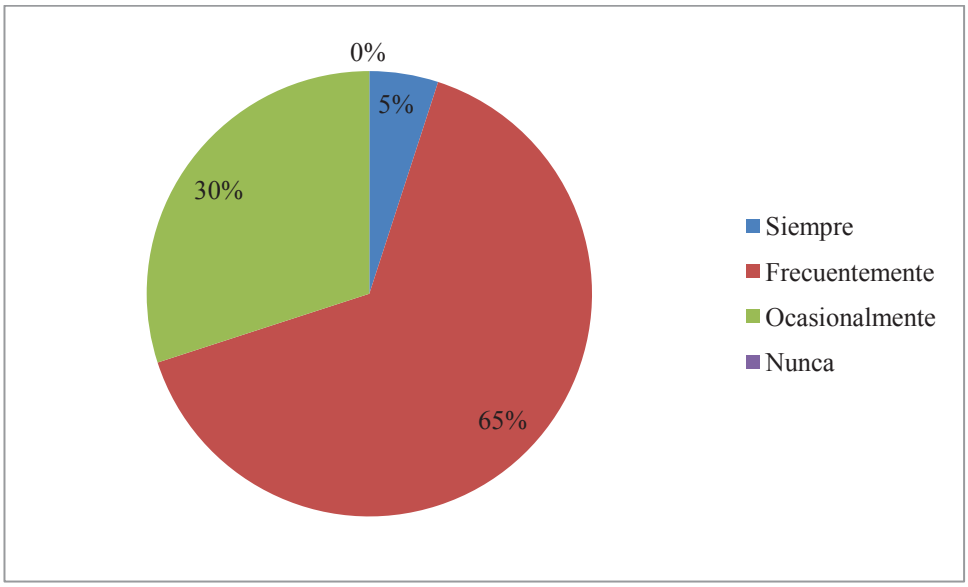


Figura 1.7. Resultado de utilización de estándares de programación.

Resultado.

El 25% de desarrolladores siempre utilizan estándares para modelar las aplicaciones, el 35% de forma frecuente, el 35% de forma ocasional y solo el 5% nunca. Así se puede concluir que los desarrolladores si utilizan estándares para modelar las aplicaciones (ver Figura 1.8).

Siempre	Frecuentemente	Ocasionalmente	Nunca
5	7	7	1

Tabla 1.6. Resultado de utilización de estándares en la modelación de aplicaciones.

Pregunta 7.

Tipo de Pregunta. Opción múltiple de única respuesta.

Enunciado.

Usted (o su equipo de desarrollo) ¿realizan revisiones del código?

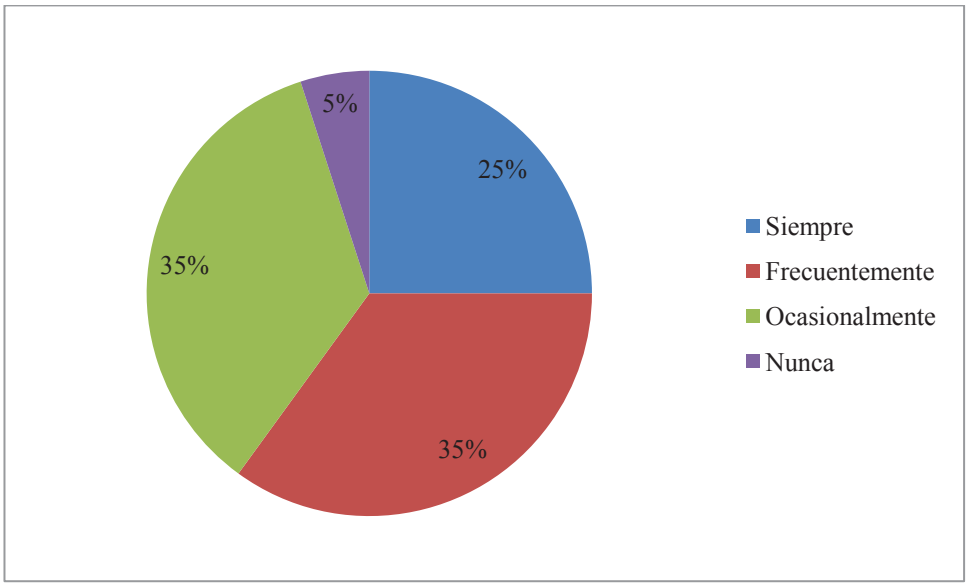


Figura 1.8. Resultado de utilización de estándares en la modelación de aplicaciones.

Resultado.

El 20% de desarrolladores siempre realizan revisiones de código, el 40% de forma frecuente, el 40% de forma ocasional y el 0% nunca. Así se puede concluir que los desarrolladores si realizan revisiones de código.

Siempre	Frecuentemente	Ocasionalmente	Nunca
4	8	8	0

Tabla 1.7. Resultado de frecuencia en la revisión del código.

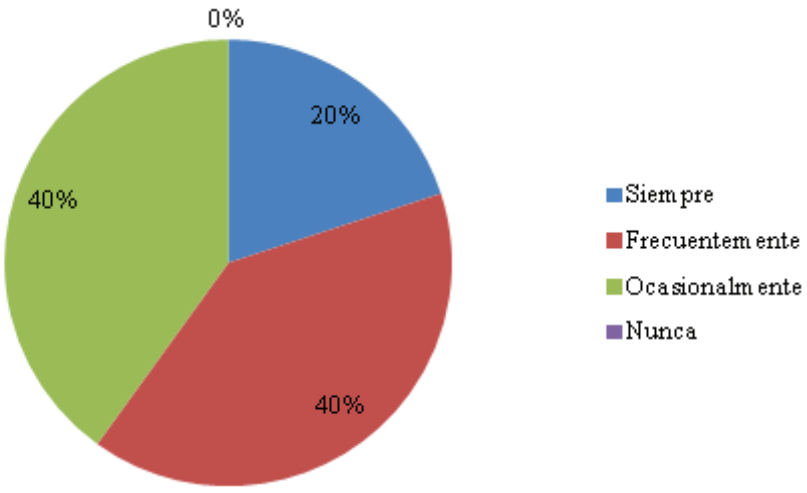


Figura 1.9. Resultado de frecuencia en la revisión del código.

Pregunta 8.

Tipo de Pregunta. Opción múltiple de única respuesta.

Enunciado.

Usted (o su equipo de desarrollo) ¿realizan pruebas de unidad?

Resultado.

El 20% de desarrolladores siempre realizan revisiones de código, el 30% de forma frecuente, el 50% de forma ocasional y el 0% nunca. Así se puede concluir que los desarrolladores ocasionalmente realizan pruebas de unidad.

Siempre	Frecuentemente	Ocasionalmente	Nunca
4	6	10	0

Tabla 1.8. Resultado de realización de las pruebas de unidad.

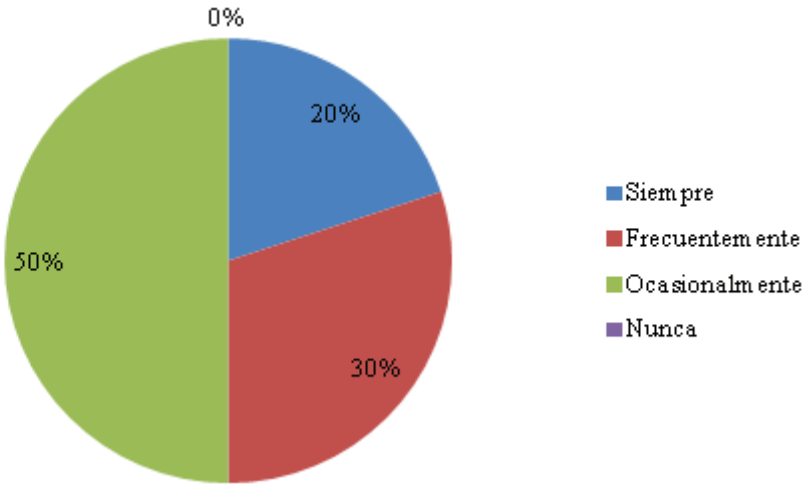


Figura 1.10. Resultado de realización de las pruebas de unidad.

Pregunta 9.

Tipo de Pregunta. Opción múltiple de única respuesta.

Enunciado.

Usted (o su equipo de desarrollo) ¿realizan pruebas de integración?

Resultado.

El 20% de desarrolladores siempre realizan revisiones de código, el 30% de forma frecuente, el 50% de forma ocasional y el 0% nunca. Así se puede concluir que los desarrolladores ocasionalmente realizan pruebas de integración.

Siempre	Frecuentemente	Ocasionalmente	Nunca
4	6	10	0

Tabla 1.9. Resultado de realización de las pruebas de integración.

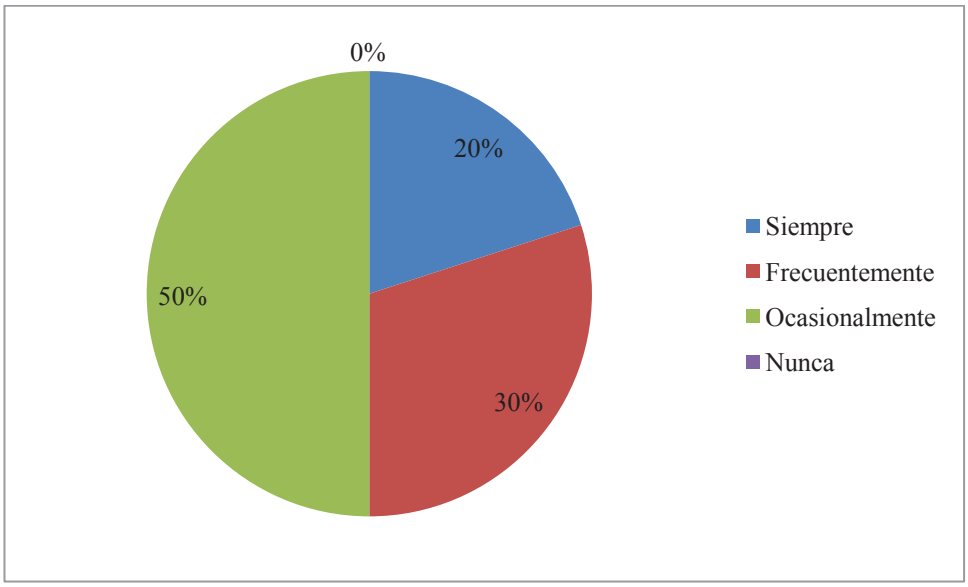


Figura 1.11. Resultado de realización de las pruebas de integración.

Pregunta 10.

Tipo de Pregunta. Opción múltiple de única respuesta.

Enunciado.

Usted (o su equipo de desarrollo) ¿desarrollan código organizado para dar soporte a las pruebas automatizadas?

Resultado.

Solo el 10% de desarrolladores siempre desarrollan código para dar paso a las pruebas automatizadas, el 30% de forma frecuente, el 35% de forma ocasional y el 25% nunca. Así se puede concluir que se desarrolla ocasionalmente código organizado para dar soporte a las pruebas automatizadas (ver Figura 1.12).

Siempre	Frecuentemente	Ocasionalmente	Nunca
2	6	7	5

Tabla 1.10. Resultado de realización de código para pruebas.

Pregunta 11.

Tipo de Pregunta. Opción múltiple de única respuesta.

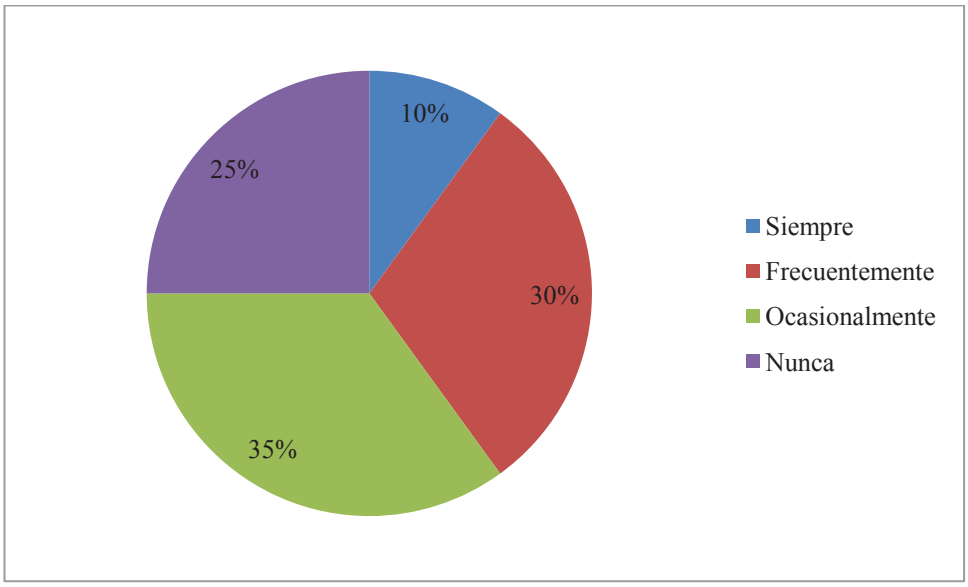


Figura 1.12. Resultado de realización de código para pruebas.

Enunciado.

Usted (o su equipo de desarrollo) ¿qué modelo de ciclo de vida para el desarrollo de software utilizan en el entorno donde desarrollan?

Resultado.

Se ha visto que el 20% utilizan prototipado evolutivo, el 35% programación extrema, el 10% Scrum, el 10% Iconix, el 10% RUP, solo el 5% UP, un modelo de ciclo de vida no escuchado antes, y solo el 10% ninguno. Así se puede concluir que la mayoría de los desarrolladores utilizan programación extrema para desarrollar los sistemas software (ver Figura 1.13).

Prototipado evolutivo	4
Programación extrema	7
Scrum	2
Iconix	2
RUP	2
UP	1
Ninguno	2

Tabla 1.11. Resultado de la modelo de ciclo de vida más utilizado.

Pregunta 12.

Tipo de Pregunta. Opción múltiple.

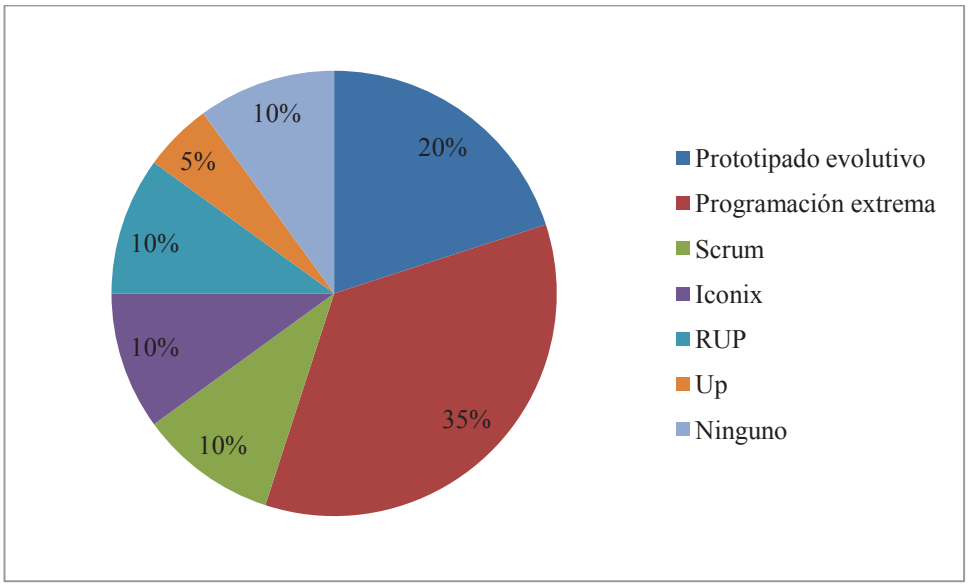


Figura 1.13. Resultado de la modelo de ciclo de vida más utilizado.

Enunciado.

Usted (o su equipo de desarrollo) ¿qué métricas utilizan para medir el código que se ha desarrollado?

Resultado.

Se ha visto que 12 utilizan el código desarrollado, 6 el código modificado, 8 el código reusado, 3 el código eliminado, 8 la complejidad del código, solo 1 las estadísticas de inspección del código, solo 1 las tasas para hallar y reparar los fallos, 9 esfuerzo y 9 planificación. Así se puede concluir que se mide el código normalmente por código desarrollado, esfuerzo y planificación.

Código desarrollado	12
Código modificado	6
Código reusado	8
Código eliminado	3
Complejidad del código	8
Estadísticas de inspección del código	1
Tasas para reparar y encontrar fallos	1
Esfuerzo	9
Planificación	9

Tabla 1.12. Resultado de métricas para cuantificar el código desarrollado.

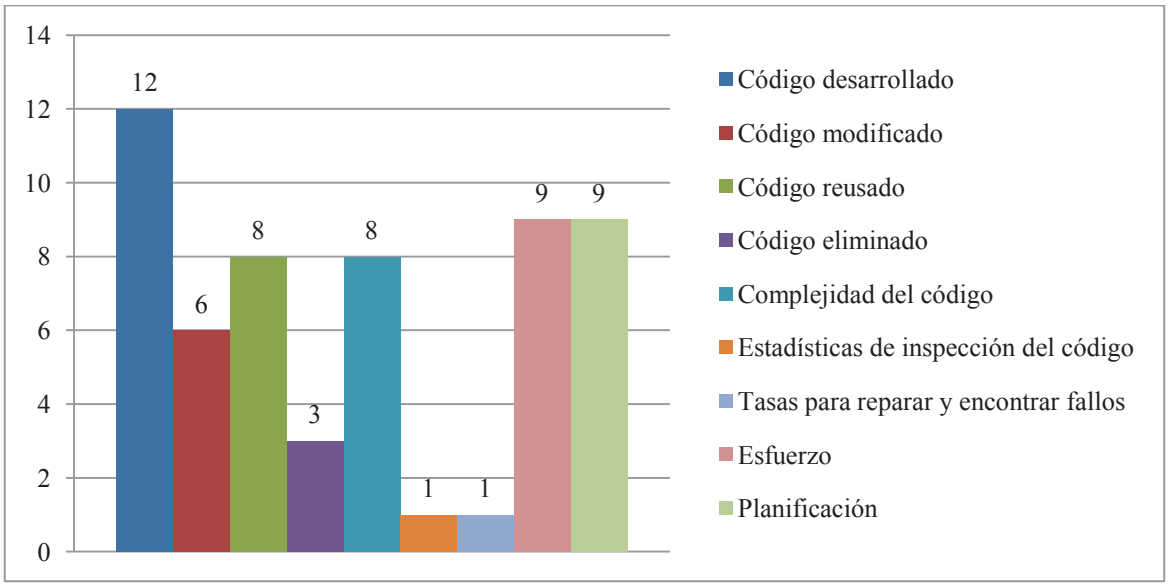


Figura 1.14. Resultado de métricas para cuantificar el código desarrollado.

Pregunta 13.

Tipo de Pregunta. Opción múltiple de única respuesta.

Enunciado.

Usted (o su equipo de desarrollo) ¿realizan cambios pequeños o grandes que anticipen las posibles fallas del sistema a desarrollar?

Resultado.

El 40% realizan cambios que anticipen a las posibles fallas del sistema de forma frecuente mientras que el 60% lo hace de forma ocasional. Así se puede concluir que se realizan ocasionalmente cambios pequeños o grandes que anticipan posibles fallas del sistema a desarrollar (ver Figura 1.15).

Siempre	Frecuentemente	Ocasionalmente	Nunca
0	8	12	0

Tabla 1.13. Resultado de anticipación a fallas.

Pregunta 14.

Tipo de Pregunta. Opción múltiple de única respuesta.

Enunciado.

Usted (o su equipo de desarrollo) ¿utilizan los diseños realizados para implementar el software a desarrollar?

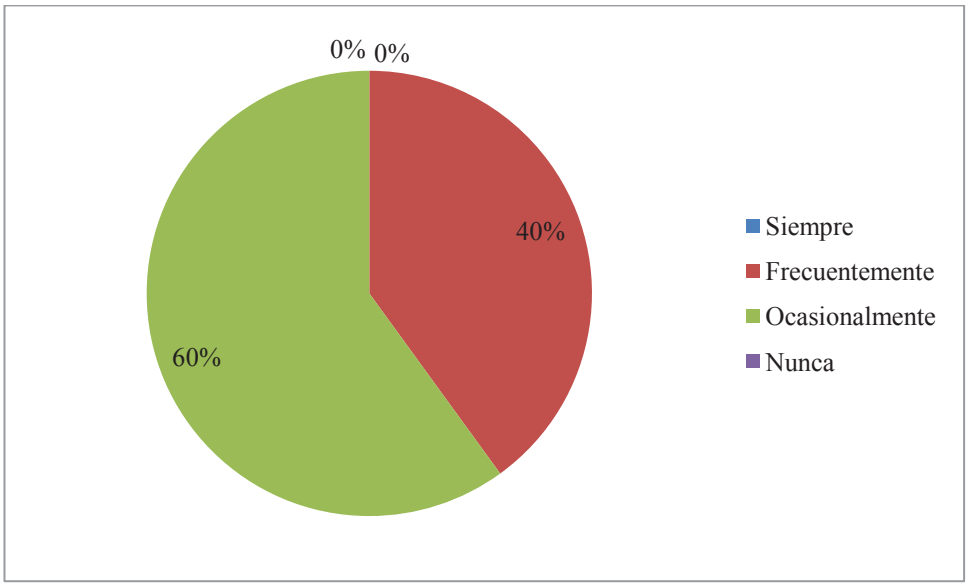


Figura 1.15. Resultado de anticipación a fallas.

Resultado.

El 15% de desarrolladores siempre utilizan los diseños realizados, el 55% de forma frecuente, el 30% de forma ocasional y el 0% nunca. Así se puede concluir que los desarrolladores utilizan los diseños realizados para implementar el software a desarrollar.

Siempre	Frecuentemente	Ocasionalmente	Nunca
3	11	6	0

Tabla 1.14. Resultado de frecuencia de implementación de diseños.

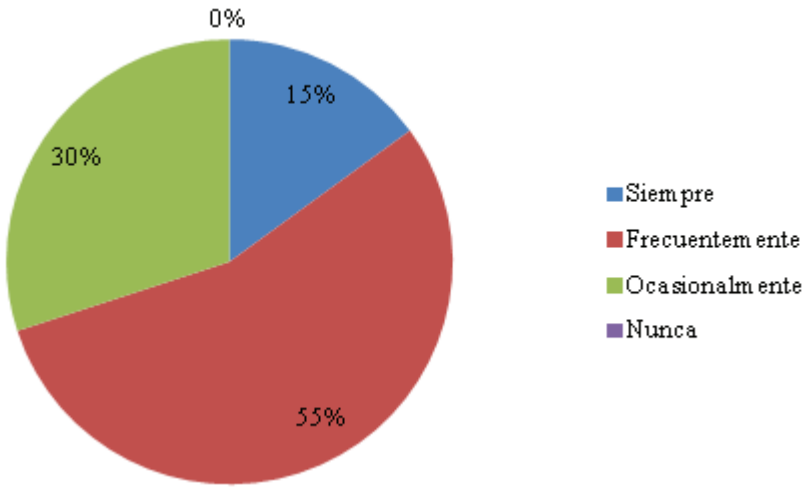


Figura 1.16. Resultado de frecuencia de implementación de diseños.

Pregunta 15.

Tipo de Pregunta. Opción múltiple.

Enunciado.

¿Qué lenguajes de implementación usted utiliza (puede escoger más de uno)?

Resultado.

Se ha visto que 6 utilizan lenguajes de configuración, 10 lenguajes de kit y herramientas y 17 los lenguajes de programación. Así se puede concluir que se desarrolla los sistemas software por medio de los lenguajes de programación y además utilizan los lenguajes de kit de herramientas.

Lenguajes de configuración	6
Lenguajes de kit de herramientas	10
Lenguajes de programación	17

Tabla 1.15. Resultado de los lenguajes de implementación más utilizados.

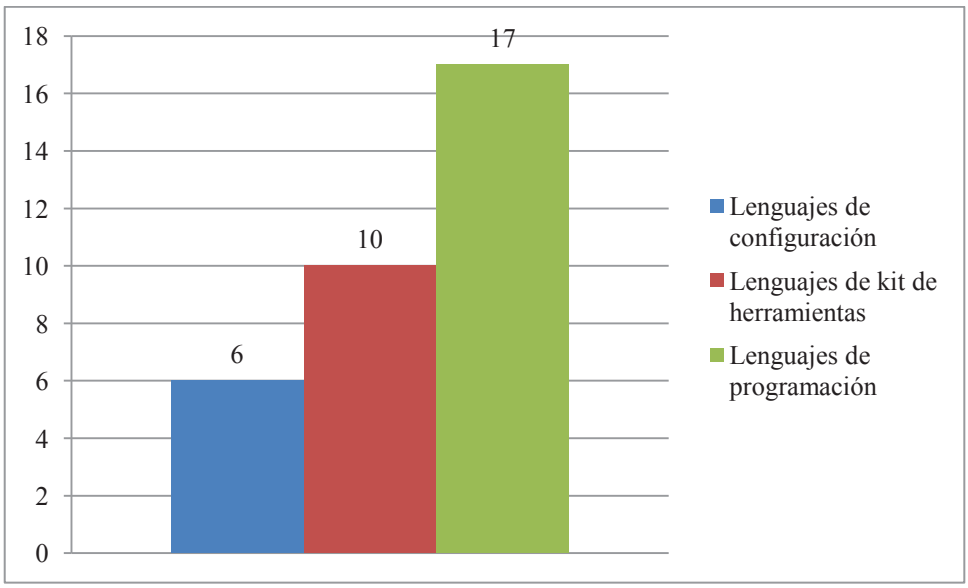


Figura 1.17. Resultado de los lenguajes de implementación más utilizados.

Pregunta 16.

Tipo de Pregunta. Opción múltiple.

Enunciado.

Usted (o su equipo de desarrollo) ¿Conoce las notaciones que tiene el lenguaje de programación que usted utiliza? (SI/NO). (En caso de que sea SI) ¿Qué notaciones tiene el lenguaje de programación que usted utiliza?

Resultado.

Se ha visto que de los 20 desarrolladores encuestados el 85% (17) sabía las notaciones que tenía el lenguaje de programación mientras que el 15% (3) no. Y de los 17 que si sabían, 10 respondieron que su lenguaje era lingüístico, 8 respondieron formal y 8 visual. Así se puede concluir que la notación del lenguaje de programación que predomina hoy día es lingüística (ver Figura 1.19).

Si	No
17	3

Tabla 1.16. Resultados de desarrolladores que sabían de las notaciones de su lenguaje de programación.

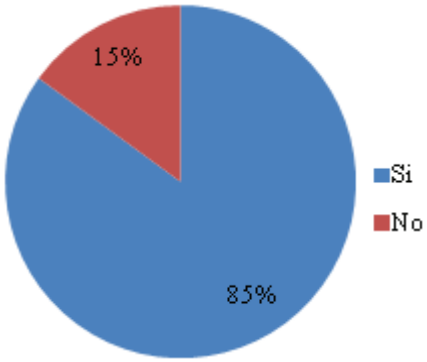


Figura 1.18. Resultados de desarrolladores que sabían de las notaciones de su lenguaje de programación.

Lingüística	Formal	Visual
10	8	8

Tabla 1.17. Resultados de la tendencia de notación del lenguaje de programación.

Pregunta 17.

Tipo de Pregunta. Opción múltiple.

Enunciado.

¿Qué consideraciones usted (o el equipo de desarrollo) utilizan para codificar las aplicaciones?

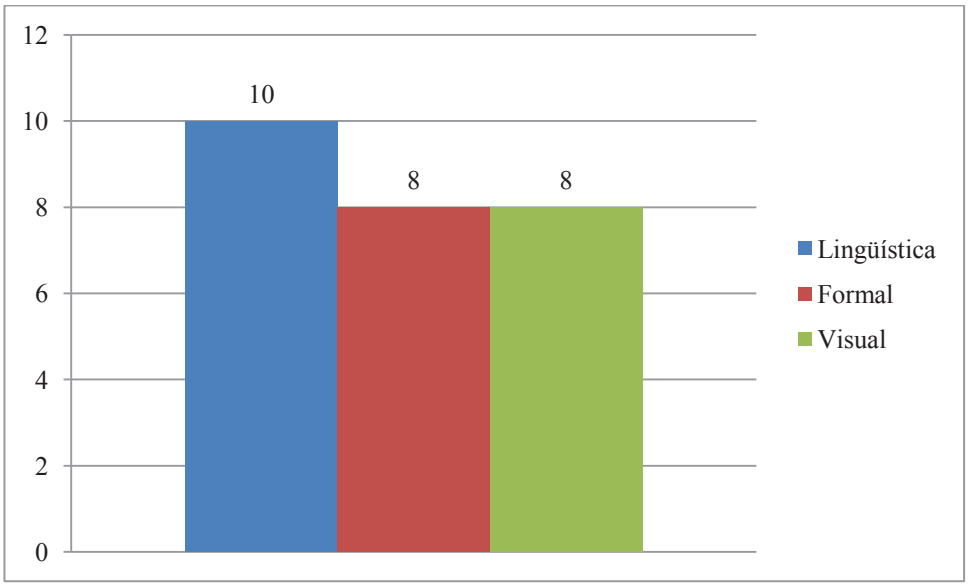


Figura 1.19. Resultados de la tendencia de notación del lenguaje de programación.

Resultado.

Se ha visto que 11 utilizan técnicas para crear código entendible, 14 utilizan clases, tipos enumerados, variables, constantes y otras entidades similares, 10 el uso de estructura del control, 11 manejo de las condiciones de error, 7 prevención de las violaciones de seguridad a nivel de código, 5 el uso de recursos, 9 organización del código fuente, 11 documentación del código y 8 ajuste del código. Así se puede concluir que las consideraciones que se utilizan para codificar las aplicaciones son el uso de clases, tipos enumerados, variables, constantes nombradas y otras entidades similares.

Técnicas para crear código entendible	11
El uso de clases, tipos enumerados, variables, constantes nombradas y otras entidades similares.	14
Uso de estructuras de control.	10
Manejo de las condiciones de error.	11
Prevención de las violaciones de seguridad a nivel de código.	7
Uso de recursos mediante el uso de los mecanismos de exclusión y de la disciplina en el acceso a los recursos reutilizables en serie.	5
Organización del código fuente	9
Documentación del código	11
Ajuste (tuning) del código	8

Tabla 1.18. Resultados de las consideraciones tomadas al desarrollar.

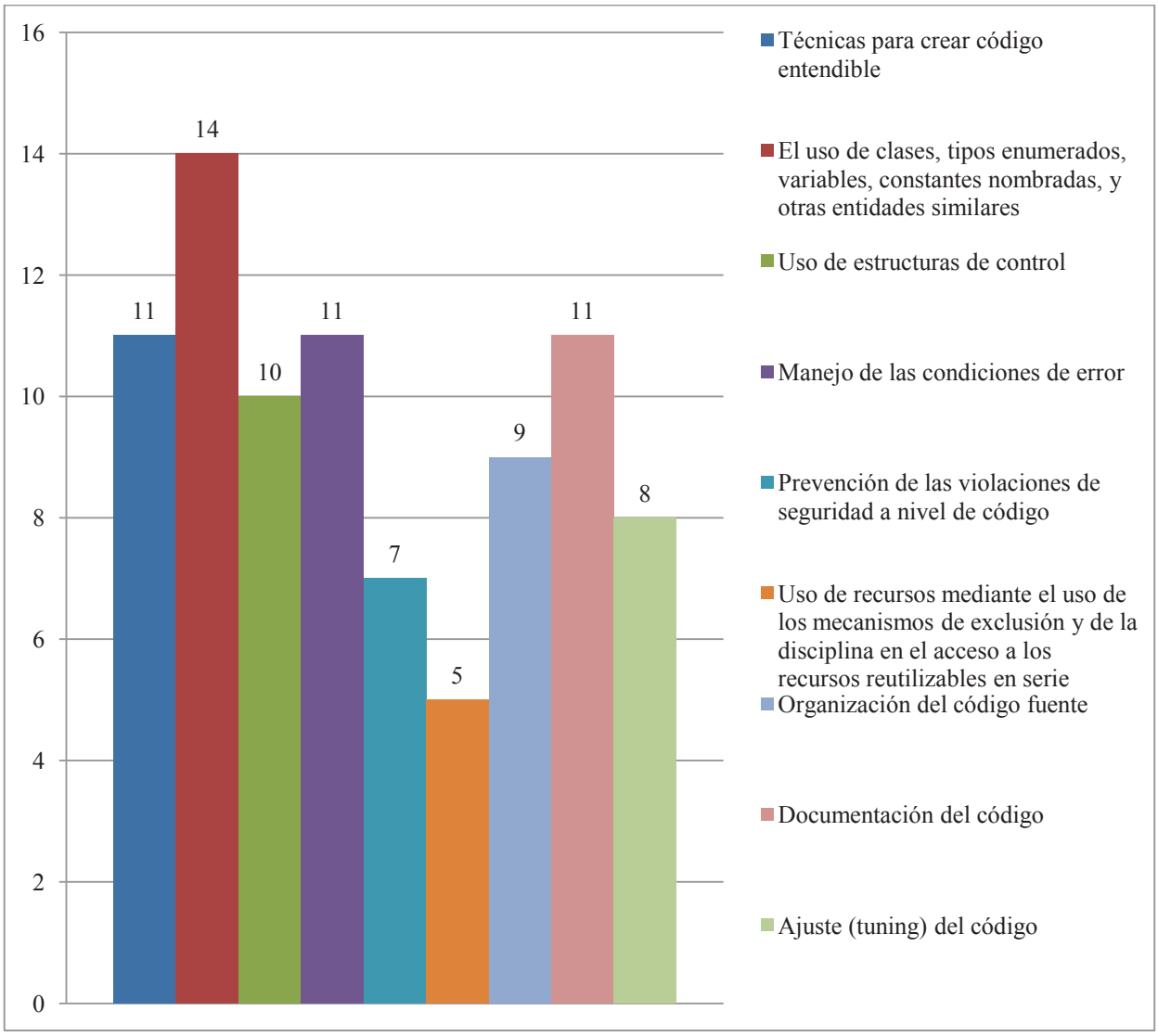


Figura 1.20. Resultados de las consideraciones tomadas al desarrollar.

Pregunta 18.

Tipo de Pregunta. Opción múltiple.

Enunciado.

¿Qué tareas usted (o el equipo de desarrollo) realizan para el reuso de código en el desarrollo de un sistema?

Resultado.

Se ha visto que 14 seleccionan las unidades reutilizables, bases de datos, procedimientos y datos de prueba; 11 evalúan el código o reutilizan la prueba y 5 realizan la presentación de la información sobre la reutilización de código nuevo, procedimientos de prueba y datos de prueba. Así se puede concluir que los

desarrolladores se enfocan más en la selección de unidades reutilizables, bases de datos, procedimientos de prueba, o datos de prueba.

La selección de unidades reutilizables, bases de datos, procedimientos de prueba, o datos de prueba.	14
La evaluación de código o reutilización de prueba.	11
La presentación de la información sobre la reutilización de código nuevo, los procedimientos de prueba, o datos de prueba.	5

Tabla 1.19. Resultados de las consideraciones tomadas al reutilizar información de desarrollo.

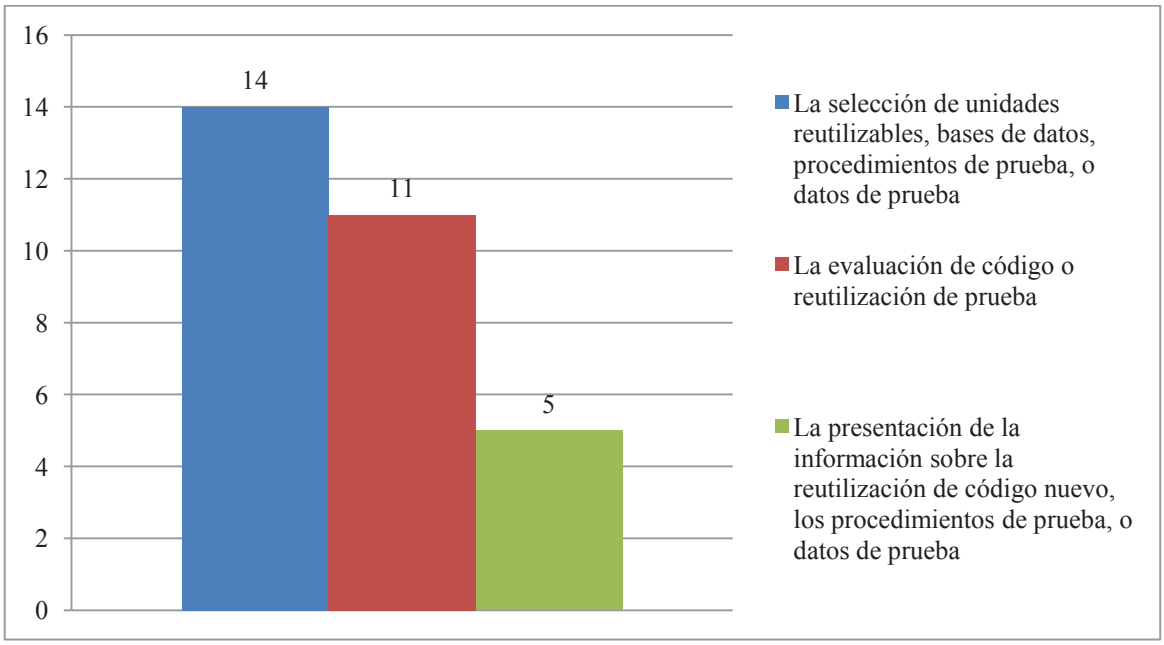


Figura 1.21. Resultados de las consideraciones tomadas al reutilizar información de desarrollo.

Pregunta 19.

Tipo de Pregunta. Opción múltiple.

Enunciado.

¿Qué técnicas usted (o el equipo de desarrollo) realizan para la calidad de la implementación en el desarrollo de un sistema?

Resultado.

Se ha visto que 13 realizan pruebas de unidad e integración; 10 realizan de primer desarrollo, 5 código de paso a paso, 4 el uso de afirmaciones, 10 depuración, 7

revisiones técnicas y 3 análisis estático. Así se puede concluir que los desarrolladores se enfocan más en las pruebas de unidad y de integración, en las pruebas de primer desarrollo y la depuración.

Pruebas de unidad y de integración	13
Prueba de primer desarrollo	10
Código de paso a paso	5
Uso de afirmaciones	4
Depuración	10
Revisiones técnicas	7
Análisis estático	3

Tabla 1.20. Resultados de las consideraciones tomadas al medir la calidad de implementación.

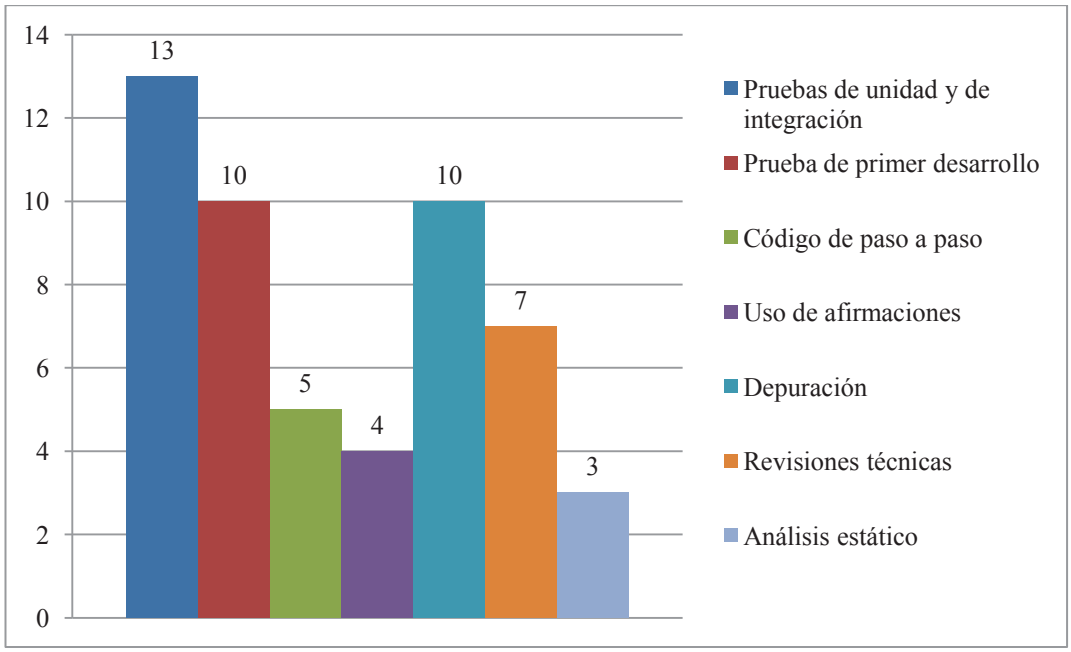


Figura 1.22. Resultados de las consideraciones tomadas al medir la calidad de implementación.

Pregunta 20.

Tipo de Pregunta. Opción múltiple.

Enunciado.

¿Qué aspectos usted (o el equipo de desarrollo) usan para la integración de componentes en la implementación durante el desarrollo de un sistema?

Resultado.

Se ha visto que 15 planean la secuencia en que los componentes se integrarán; 6 crean las estructuras para apoyar las versiones liberadas, 5 determinan el grado de las pruebas y la calidad del trabajo antes de integrar las partes y 7 determinan en qué punto ya prueban las versiones no liberadas. Así se puede concluir que los desarrolladores solo se preocupan en saber en qué secuencia se integrarán las partes del sistema.

Planificación de la secuencia en la que los componentes se integrarán.	15
Creación de estructuras para apoyar las versiones no liberadas de la aplicación.	6
Determinar el grado de las pruebas y la calidad del trabajo realizado en los componentes antes de su integración.	5
Determinación de puntos en el proyecto en el que versiones provisionales del software se ponen a prueba.	7

Tabla 1.21. Resultados de las consideraciones tomadas al medir al momento de integrar las partes desarrolladas.

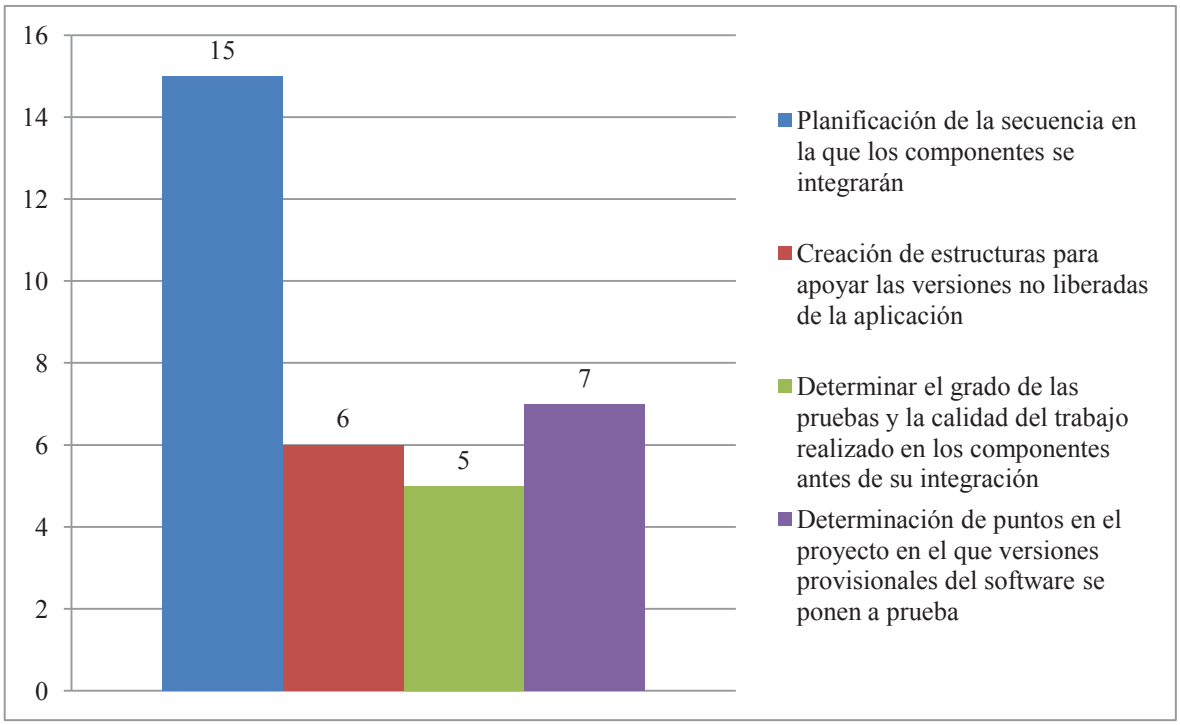


Figura 1.23. Resultados de las consideraciones tomadas al medir al momento de integrar las partes desarrolladas.

1.1.7. CONCLUSIONES ACERCA DE LA SITUACIÓN ACTUAL DEL PROCESO DE IMPLEMENTACIÓN DE SOFTWARE.

- Por lo general, los desarrolladores no realizan código complejo para lo que hacen.
- Se están utilizando estándares al momento de programar, algunos usan técnicas mientras otros no las usan.
- Debido a que los requisitos varían conforme al tiempo los desarrolladores, están conscientes de que su código puede ser alterado.
- No se ha puesto atención a la documentación del código, por lo que se realiza de forma ocasional.
- Los desarrolladores le dan la importancia suficiente a los estándares de programación para el desarrollo de software.
- La modelación es fundamental al momento de implementar, ya sea que se lo realice de forma ocasional o la mayor parte del tiempo.
- Se revisa el código implementado la mayor parte del tiempo o al menos de forma ocasional.
- Los desarrolladores por implementar olvidan la importancia de realizar pruebas de unidad e integración, lo que lo realizan de forma ocasional.
- En forma ocasional se automatizan las pruebas para el código implementado.
- El modelo de ciclo de vida de programación extrema es el más utilizado por los desarrolladores hoy en día.
- Se mide el código desarrollado mediante el número de líneas desarrolladas, por esfuerzo, planificación, reutilización del código y su complejidad.
- Los desarrolladores no están conscientes de que al momento de desarrollar su código puede fallar.
- Se utilizan los diseños de los sistemas software para realizar la implementación de manera frecuente.
- Además de que se utilizan los lenguajes de programación, también se utilizan los lenguajes de kit de herramientas como herramientas de implementación.
- La tendencia del lenguaje de programación utilizado por los desarrolladores encuestados es lingüística.

- Los desarrolladores basan su trabajo en el uso de clases, tipos enumerados, variables, constantes y otras entidades similares junto con las técnicas para crear código entendible, manejo de las condiciones de error y la documentación de código.
- Los desarrolladores no se preocupan en probar el código a reutilizar y presentar su información aunque si llevan una secuencia de reutilización.
- La calidad de implementación se basa en las pruebas de unidad y de integración, en las pruebas de primer desarrollo y la depuración.
- Los desarrolladores no ponen la suficiente atención en realizar las pruebas de las versiones no liberadas en la integración de las partes aunque por lo menos si se sigue una secuencia de integración.

1.1.8. PUNTOS A CUBRIR DEL METODO DE IMPLEMENTACION DE SISTEMAS SOFTWARE.

- Debe generar código de baja complejidad.
- Debe ser prioritario los estándares de programación.
- Establecer recomendaciones que hagan que el código sea fácilmente alterado.
- Enfocar la documentación del código en el diseño sin generar excesiva volumen de esta.
- Reunir estándares, técnicas reconocidas de las mejores prácticas en conjunto con la experiencia personal.
- Modelar a bajo nivel las partes a implementar para que la implementación se facilite.
- Establecer la revisión del código con el fin de mejorarlo.
- Realizar pruebas de unidad e integración a baja escala del código que se implementa.
- Establecer la automatización de las pruebas al código mediante herramientas o código que realice pruebas al código implementado.
- Utilizar los modelos de ciclo de vida más usados hoy en día, es decir, utilizar las metodologías ágiles en el método.

- Medir el código desarrollado en función de las métricas más utilizadas y por herramientas.
- Establecer código que pueda prevenir las fallas futuras del sistema.
- Interpretar los diseños de tal forma que se los pueda plasmar en un lenguaje de programación.
- Dar énfasis a los lenguajes de programación como lenguajes de implementación aunque también debe ser abierto a otros lenguajes de implementación.
- Debe ser adaptable con cualquier notación que el lenguaje de programación que pueda tener.
- Basar el método en las mejores prácticas de implementación de código.
- Incluir las pruebas de código a reutilizar.
- Implementar sistemas software con código de calidad mediante buenas prácticas de calidad de implementación.
- Establecer una secuencia de integración y probar las partes antes de su integración con el sistema en general.

1.2 FUNDAMENTOS DE IMPLEMENTACIÓN DE SOFTWARE. [3]

Los fundamentos de la implementación de software incluyen:

- Minimizar la complejidad.
- Anticipar el cambio.
- Implementar para verificación.
- Estándares para implementación.

Los tres primeros conceptos se aplican al diseño así como en la implementación. Las siguientes secciones se definen estos conceptos y describir cómo se aplican a la implementación.

1.2.1. MINIMIZAR LA COMPLEJIDAD.

Un gran factor en cómo la gente comunica la intención de los computadores es la capacidad muy limitada de las personas para mantener estructuras complejas e información en sus memorias de trabajo, especialmente durante largos periodos

de tiempo. Esto nos lleva a uno de los principales impulsores en la implementación de software: minimizar la complejidad. La necesidad de reducirla se aplica esencialmente en todos los aspectos de la implementación de software, y es particularmente crítica para el proceso de verificación y ensayo de las implementaciones de software.

En implementación de software, la complejidad reducida es lograda a través de enfatizar la creación de código que es simple y fácil de leer en vez de inteligente.

Minimizar la complejidad se realiza a través de hacer uso de los estándares (mencionados en el tópico 1.2.4, Estándares para Implementación) y a través de numerosas técnicas específicas (mencionadas en el tópico 1.4.3, Codificación). También es apoyado por las técnicas de calidad (mencionadas en el tópico 1.4.6, Calidad de Implementación) centradas en la actividad antes mencionada.

1.2.2. ANTICIPAR EL CAMBIO.

La mayoría del software va a cambiar con el tiempo, y la anticipación del cambio lleva a muchos aspectos de la implementación de software. El software es inevitablemente parte de los cambios del entorno externo, y los cambios en los entornos externos de software afectan en diversas maneras.

Anticiparse al cambio es soportado por varias técnicas específicas (resumidas en el tópico 1.4.3, Codificación).

1.2.3. IMPLEMENTAR PARA VERIFICACION.

Implementar para verificación significa la creación de software de tal manera que las fallas puedan ser corregidas fácilmente por los ingenieros de software escribiendo el software, así como durante las pruebas independientes y las actividades operacionales. Las técnicas que dan soporte a implementar para verificación incluyen el seguimiento de estándares de codificación para dar soporte a revisiones de código, pruebas unitarias, organización del código que soporte a pruebas automatizadas, y uso restringido de estructuras de lenguaje complejas o difíciles de entender, entre otras.

1.2.4. ESTÁNDARES EN IMPLEMENTACIÓN.

Estándares que afectan directamente a las cuestiones de implementación incluyen:

- Los métodos de comunicación (por ejemplo, los estándares para formatos de documentos y contenidos).
- Los lenguajes de programación (por ejemplo, los estándares de lenguaje para lenguajes como Java y C + +).
- Las plataformas (por ejemplo, los estándares de interfaz de programador de llamadas del sistema operativo).
- Herramientas (por ejemplo, los estándares esquemáticos de notaciones como UML <<en español, Lenguaje de Modelado Unificado>>).

Uso de estándares externos. La implementación depende de la utilización de las normas externas para los lenguajes de implementación, herramientas de implementación, interfaces técnicas y las interacciones entre la implementación de software y otras áreas de conocimiento. Los estándares provienen de numerosas fuentes, incluyendo las especificaciones de interfaz de hardware y software, como el Object Management Group (OMG) y de organizaciones internacionales como el IEEE o ISO.

Uso de estándares internos. Los estándares también pueden ser creados sobre una base de organización a nivel corporativo o para su uso en proyectos específicos. Estas normas apoyan a la coordinación de las actividades del grupo, lo que minimiza la complejidad, anticipa el cambio e implementa para verificación.

1.3 GESTIÓN DE IMPLEMENTACIÓN. [3]

1.3.1. MODELOS DE IMPLEMENTACION.

Numerosos modelos han sido creados para desarrollar software algunos de ellos enfatizan la implementación más que otros.

Algunos modelos son más lineales desde el punto de vista de implementación, como los modelos de ciclo de vida de cascada y entrega basada en escenarios.

Estos modelos tratan a la implementación como una actividad que ocurre solo después de que el trabajo prerrequisito importante haya sido completado - incluyendo el trabajo detallado de requisitos, el trabajo de diseño amplio y una planificación detallada. Los enfoques más lineales tienden a enfatizar las actividades que preceden a la implementación (requisitos y el diseño), y tienden a crear más separaciones claras entre las distintas actividades. En estos modelos, el énfasis principal de la implementación puede ser de codificación.

Otros modelos son más repetitivos, como el prototipado evolutivo, Programación Extrema y Scrum. Estos enfoques tienden a tratar a la implementación como una actividad que se produce al mismo tiempo con otras actividades de desarrollo de software, incluidos los requisitos, el diseño y planificación, o se los superpone. Estos enfoques tienden a mezclar las actividades de diseño, codificación, y prueba, y que a menudo tratan la combinación de actividades como implementación.

En consecuencia, lo que es considerado como "implementación" depende en cierta medida en el modelo de ciclo de vida utilizado.

1.3.2. PLANIFICACION DE LA IMPLEMENTACION.

La elección del método a implementar es un aspecto clave de la actividad de planificación de la implementación. La elección del método de implementación afecta el grado en que los requisitos previos de implementación se lleven a cabo, el orden en que se lleven a cabo, y el grado en que se espera que se complete antes de iniciarse la implementación.

El enfoque de la implementación afecta a la capacidad del proyecto para reducir la complejidad, a anticipar el cambio, e implementar para verificación. Cada uno de estos objetivos también se puede tratar en los niveles de proceso, requisitos y diseño - sino que también estarán influenciados por la elección del método de implementación.

La planificación de la implementación también define el orden en que los componentes son creados e integrados, los procesos de gestión de calidad de software, la colocación de asignaciones de tareas a los ingenieros de software específicos, y las otras tareas, de acuerdo con el método elegido.

1.3.3. MEDICION DE LA IMPLEMENTACION.

Numerosas actividades de implementación y artefactos pueden ser medidos, incluyendo el código desarrollado, código modificado, código reutilizado, código destruido, complejidad del código, estadísticas de inspección de código, tasas de fallas encontradas y arregladas, esfuerzo y planificación. Estas mediciones pueden ser útiles para los propósitos de gestión de implementación, asegurando la calidad durante la implementación, mejorando el proceso de implementación, así como por otras razones.

1.4 CONSIDERACIONES PRÁCTICAS ANTES DE IMPLEMENTAR SISTEMAS SOFTWARE. [3]

La implementación es una actividad en la que el software tiene que llegar a un acuerdo con arbitrarias y caóticas restricciones del mundo real, y hacerlo exactamente. Debido a su proximidad con las restricciones del mundo real, la implementación está más impulsada por consideraciones prácticas que algunas áreas de conocimiento, y la ingeniería de software es quizás más como artesanal en el área de implementación.

1.4.1. DISEÑO DE LA IMPLEMENTACIÓN.

Algunos proyectos se asignan más a la actividad de diseño que a la implementación, otros a una fase explícitamente centrada en el diseño. Independientemente de la distribución exacta, algo del trabajo de diseño detallado se producirá en el nivel de implementación, y que el trabajo de diseño tiende a ser dictado por restricciones inamovibles impuestas por el problema del mundo real que está siendo dirigida por el software.

Así como los trabajadores de la construcción construyen una estructura física debe hacer modificaciones a pequeña escala para tener en cuenta las brechas

inanticipadas en los planes del constructor, los trabajadores de implementación de software deben hacer modificaciones a pequeña y grande escala, los trabajadores de implementación de software deben hacer modificaciones en una escala más pequeña o más grande para dar cuerpo a los detalles del diseño del software durante la implementación.

Los detalles de la actividad de diseño en el nivel de implementación son esencialmente los mismos como se describen en el Área de Conocimiento de Diseño de Software, pero son aplicados a una menor escala.

1.4.2. LENGUAJES DE IMPLEMENTACIÓN.

Los lenguajes de comunicación incluyen todas las formas de comunicación por el cual un ser humano puede especificar una solución del problema ejecutable en un ordenador.

El tipo más simple de lenguaje de implementación es un lenguaje de configuración, en la que los ingenieros de software eligen entre un conjunto limitado de opciones predefinidas para crear nuevas o personalizadas instalaciones de software. Los archivos de configuración basados en texto utilizados tanto en Windows y sistemas operativos Unix son ejemplos de esto, y la lista de selección de estilo de menú de algunos generadores de programas constituye otra.

Los lenguajes de kit de herramientas se utilizan para implementar aplicaciones de herramientas (conjuntos integrados de partes reutilizables de aplicaciones específicas), y son más complejos que los lenguajes de configuración.

Los lenguajes de programación son el tipo más flexible de lenguajes de implementación. También contienen la menor cantidad de información acerca de áreas específicas de aplicación y procesos de desarrollo, y por lo tanto requieren mayor capacitación y habilidad para utilizar con efectividad.

Hay tres tipos generales de notación que se utiliza para lenguajes de programación, a saber:

- Lingüística.
- Formal.
- Visual.

Las notaciones lingüísticas se distinguen en particular por el uso de las cadenas de texto como palabras para representar las implementaciones complejas de software, y la combinación de tales como cadenas de textos en los patrones que tienen una sintaxis de frase similar. Utilizadas correctamente, cada cadena, debe tener una fuerte connotación semántica proporcionando una comprensión inmediata e intuitiva de lo que sucederá cuando la construcción de software subyacente se ejecuta.

Las notaciones formales dependen menos de los significados intuitivos y cotidianos de las palabras y las cadenas de texto y mucho más en definiciones respaldadas por precisas, inambiguas, y formales (o matemáticas) definiciones. Las notaciones formales de implementación y los métodos formales están en el corazón de la mayoría de las formas de la programación de sistemas, donde la precisión, el comportamiento del tiempo, y la capacidad de prueba son más importantes que la facilidad de mapeo en lenguaje natural. Las implementaciones formales también utilizan formas definidas con precisión de la combinación de símbolos que eviten la ambigüedad de muchas implementaciones de lenguaje natural.

Las notaciones visuales dependen mucho menos de las notaciones orientadas al texto de la implementación, tanto lingüística como formalmente, y en lugar de ello dependen de la interpretación visual directa y la colocación de las entidades visuales que representan el software subyacente. La implementación visual tiende a ser algo limitada por la dificultad de hacer declaraciones "complejas" usando solamente el movimiento de las entidades visuales en una pantalla. Sin embargo, también puede ser una poderosa herramienta en los casos en que la tarea de programación principal es simplemente para construir y "ajustar" una interfaz

visual para un programa, comportamiento detallado del cual se ha definido anteriormente.

1.4.3. CODIFICACIÓN.

Las siguientes consideraciones se aplican a la actividad de implementación de software:

- Técnicas para la creación de código fuente comprensible, incluyendo nombres y el diseño del código fuente.
- Uso de las clases, tipos enumerados, variables, constantes con nombre, y otras entidades similares.
- Uso de estructuras de control.
- El manejo de las condiciones – tanto de errores y excepciones previstas (entrada de datos erróneos, por ejemplo).
- Prevención de las violaciones de seguridad a nivel de código (desbordamientos de búfer o se desborda un índice de matriz, por ejemplo).
- Uso de recursos mediante el uso de los mecanismos de exclusión y disciplina en el acceso a los recursos reutilizables en serie (incluyendo hilos o bloqueos de base de datos).
- La organización del código fuente (en las declaraciones, rutinas, clases, paquetes, u otras estructuras).
- Documentación del código.
- Ajuste del código.

1.4.4. PRUEBAS DE IMPLEMENTACIÓN.

La implementación consta de dos formas de evaluación, que a menudo son realizadas por el ingeniero de software que escribió el código:

- Pruebas de unidad.
- Pruebas de integración.

El propósito de las pruebas de implementación es el de reducir la brecha entre el momento en que los fallos se insertan en el código y el tiempo en que estos se detectan. En algunos casos, las pruebas de implementación se realizan después

de que el código haya sido escrito. En otros casos, los casos de prueba pueden ser creados antes de que el código sea escrito.

Las pruebas de implementación implican típicamente un subconjunto de tipos de pruebas, que se describen en el Área de Conocimiento de Pruebas de software. Por el momento, las pruebas de implementación no suelen incluir las pruebas de sistema, pruebas alfa, pruebas beta, pruebas de estrés, pruebas de configuración, las pruebas de usabilidad, u otros tipos más especializados de pruebas.

Dos estándares se han publicado sobre el tema: el estándar IEEE 829-1998, para la documentación de pruebas de software y el estándar IEEE 1008-1987, para las pruebas unitarias de software.

1.4.5. REUSO.

Como se indicó en la introducción de (IEEE 1517-1599):
"La reutilización de la implementación de software implica más que la creación y uso de las bibliotecas de activos. Se requiere la formalización de la práctica de la reutilización mediante la integración de los procesos de reutilización y actividades en el ciclo de vida del software." Sin embargo, la reutilización es lo suficientemente importante en la implementación de software que se incluye aquí como un tema.

Las tareas relacionadas con la reutilización en la implementación de software durante la codificación y las pruebas son las siguientes:

- La selección de las unidades reutilizables, bases de datos, procedimientos de prueba, o los datos de prueba.
- La evaluación de código o reutilización de prueba.
- La presentación de información de reutilización sobre el código nuevo, los procedimientos de prueba, o datos de prueba.

1.4.6. CALIDAD DE IMPLEMENTACIÓN.

Existen numerosas técnicas para garantizar la calidad del código, como es implementado. Las técnicas principales que se usan para la implementación son:

- Pruebas de unidad y de integración.
- Prueba de primer desarrollo.
- Código de paso a paso.
- Uso de afirmaciones.
- Depuración.
- Revisiones técnicas.
- Análisis estático.

La técnica específica o técnicas seleccionadas dependerán de la naturaleza del software a implementar, así como en las habilidades establecidas de los ingenieros de software que realizan la implementación.

Las actividades de calidad de implementación se diferencian de otras actividades de calidad por su enfoque. Las actividades de calidad de implementación se centran en el código y en los artefactos que están estrechamente relacionados con el código: los diseños de pequeña escala a diferencia de otros artefactos que no están tan directamente relacionados con el código, como los requisitos, diseños de alto nivel, y planes.

1.4.7. INTEGRACIÓN.

Una actividad fundamental durante la implementación es la integración de las rutinas construidas por separado, clases, componentes y subsistemas. Además, un sistema software en particular puede que tenga que ser integrado con otros sistemas software o hardware.

Las preocupaciones relacionadas con la integración de la implementación incluyen la planificación de la secuencia en la que los componentes se integrarán, la creación de estructuras para apoyar a las versiones no liberadas del programa, determinar el grado de las pruebas y la calidad del trabajo realizado en los componentes antes de su integración, y la determinación de puntos en el proyecto en el que las versiones no liberadas del software se ponen a prueba.

CAPÍTULO 2. MÉTODO PARA LA IMPLEMENTACIÓN DE SOFTWARE.

2.1 ELABORACIÓN DEL MÉTODO DE IMPLEMENTACIÓN.

PROCEDIMIENTO:

Este proceso se basará en los siguientes pasos:

Paso 1) Seleccionar modelo de implementación.
Paso 2) Planificar la implementación.
Paso 3) Seleccionar métricas y sus técnicas.
Paso 4) Implementación y pruebas unitarias.
Paso 5) Verificación
Paso 6) Integración.
Paso 7) Pruebas de Integración.

2.1.1. SELECCIONAR MODELO DE IMPLEMENTACION.

PROCEDIMIENTO:

Se elegirá un modelo tradicional o se elegirá cualquier proceso de desarrollo iterativo para utilizar lo referente a implementación.

SALIDA:

La siguiente descripción breve de la selección del modelo de implementación.

Selección del Modelo de Implementación.
Constará el modelo de implementación elegido y su justificación.

2.1.2. PLANIFICAR LA IMPLEMENTACION.

PROCEDIMIENTO:

Este proceso se basará en los siguientes pasos:

Paso 1) Preparar la planificación.
Paso 2) Planificar la minimización de la complejidad.
Paso 3) Planificar la anticipación de cambios.

Paso 4) Seleccionar los estándares de implementación.

Paso 5) Realizar el diseño de la implementación.

Paso 6) Seleccionar los lenguajes de implementación.

Paso 7) Seleccionar las consideraciones para codificar.

Paso 8) Preparar las pruebas unitarias.

Paso 9) Preparar las pruebas de integración.

Paso 10) Considerar el reuso.

2.1.2.1. Preparar la Planificación.

PROCEDIMIENTO:

Este proceso se basará en los siguientes pasos:

Paso 1) Determinar el costo de su desarrollo.

Paso 2) Determinar el riesgo incurrido durante su desarrollo.

Paso 3) Establecer la funcionalidad principal del sistema.

Paso 4) Enumerar los documentos del diseño del sistema que servirán en la planificación.

Paso 5) Determinar la gente que intervendrá del equipo de desarrollo y de la parte del negocio.

Paso 6) Determinar lo primero a implementarse.

Paso 7) Determinar lo siguiente a implementar y la el tiempo en que se lo realizaría.

Paso 8) Cuando se termine de implementar la funcionalidad actual, se repite el Paso 7 con la siguiente funcionalidad, hasta terminar de implementar el sistema.

SALIDA:

La siguiente planificación de la primera funcionalidad que se implementará del sistema.

Costo de Desarrollo:
<Se coloca el tipo de sistema que va a implementarse y la justificación de su costo>.

Riesgo Incurrido:
<Se coloca el nivel de confiablidad del sistema y la gravedad de un posible error al implementar el sistema>.

Funcionalidad Principal del Sistema:
<La funcionalidad que se basa en el proceso principal que el Negocio realiza>.

Documentos de Diseño:
<Una lista de los documentos que servirán para la implementación>.

Participantes:
<Se enumeran a las personas del equipo de desarrollo y a las personas del negocio>.

Característica a Implementar:
<Se coloca lo primero que se empezará a implementar>.

Próxima Característica:
<Se coloca la que irá cuando se haya implementado la característica a implementar>.

Tiempo de Espera a Implementar la Próxima Característica:
<Se colocarán los meses que se esperará a implementar la característica y de ser necesario, el número de semanas y de días también (se omiten los meses si el tiempo es menor a un mes)>.

De la segunda funcionalidad en adelante, se hará de la siguiente manera la planificación.

Característica a Implementar:
<Se coloca la característica a implementar>.

Próxima Característica:
<Se coloca la que irá cuando se haya implementado la característica a implementar>.

Tiempo de Espera a Implementar la Próxima Característica:
<Se colocarán los meses que se esperará a implementar la característica y de ser necesario, el número de semanas y de días también (se omiten los meses si el tiempo es menor a un mes)>.

2.1.2.2. Planificar la Minimización de la Complejidad.

Se procede a identificar los términos (u objetos) del dominio del problema junto con sus atributos y esto se basa en el diagrama de clases y las restricciones añadidas en el script de la base de datos (derivado su diseño del diseño del diagrama de clases). Para realizar esta tarea se debe usar lo descrito en la Tabla 2.1.

Objeto (*)	**Atributos (*)**	**Restricciones**
Objeto 1	Atributo 1	Restricción 1
	Atributo 2	Restricción 2
	…	…
	Atributo ‘l’	Restricción ‘l’
Objeto 2	Atributo 1	Restricción 1
	Atributo 2	Restricción 2
	…	…
	Atributo ‘m’	Restricción ‘m’
. . .		
Objeto ‘n’	Atributo 1	Restricción 1
	Atributo 2	Restricción 2
	…	…
	Atributo ‘o’	Restricción ‘o’

Siendo ‘l’, ’m’, ’n’ y ’o’ un entero positivo.

(*) Los datos son obligatorios.

Tabla 2.1. Plantilla de descripción de objetos.

2.1.2.3. Planificar la Anticipación de los Cambios.

Se procede a aislar las zonas inestables (susceptibles de cambio) de modo que el efecto de un cambio pueda ser controlado en un solo lugar. Estos son los pasos para el control de tales cambios.

Paso 1) Identificar elementos que parecen susceptibles de cambiar.
Paso 2) Identificar elementos que sean susceptibles de cambiar.
Paso 3) Aislar elementos que parecen susceptibles de cambiar.

1) Identificar elementos que parecen susceptibles de cambiar.

Se incluye una lista de cambios potenciales y la probabilidad de cada cambio.

2) Identificar elementos que sean susceptibles de cambiar.

Se encierra cada componente volátil en su propia clase o se encierran varios componentes que cambien al mismo tiempo por su comportamiento común.

3) Aislar elementos que parecen susceptibles de cambiar. [5]

Se deben diseñar las interfaces entre clases para ser insensibles a los cambios potenciales. Se deben diseñar las interfaces de manera que los cambios se limiten al interior de la clase y el exterior no se vea afectado. Cualquier otra clase usando la clase cambiada no debería ser consciente de que el cambio se ha producido.

Los pasos 2 y 3 se los usa en caso de que los requisitos sean susceptibles de cambio, no se lo usa con diseños predeterminados a no ser que se requiera modificación en los diseños.

Áreas que Son Susceptibles de Cambio. [5]

- Lógica de negocio.
- Dependencias de HW.
- Entrada y salida.
- Características de lenguajes no estándar.
- Diseño difícil y áreas de implementación.

- Variables de estado.
- Restricciones de tamaño de datos.

SALIDA:

La siguiente tabla en la que constan las áreas susceptibles de cambio junto con su probabilidad usadas en el paso 1.

Elemento	**Probabilidad**
Elemento 1	Porcentaje 1.
Elemento 2	Porcentaje 2.
. . .	
Elemento ‘n’	Porcentaje ‘n’

Siendo ‘n’ un entero positivo.

Tabla 2.2. Tabla de Áreas Susceptibles de Cambio.

2.1.2.4. Seleccionar los Estándares de Implementación.

Se procede a seleccionar las prácticas necesarias para realizar la implementación del sistema, mediante la siguiente lista de control [5].

Codificación.

- ¿Se han definido convenciones de codificación para los nombres, comentarios y formato?
- ¿Se han definido las prácticas de codificación específicas que están implicadas en la arquitectura, por ejemplo, cómo las condiciones de error que se manejarán, cómo la seguridad se abordará, y así sucesivamente?
- ¿Ha identificado su ubicación en la ola de la tecnología (época tecnológica) y ajustó su enfoque para que coincida? Si es necesario, ¿se ha identificado cómo se va a programar en el lenguaje en lugar de limitarse a programar en él?

Trabajo en Equipo.

- ¿Se ha definido un procedimiento de integración, es decir, ha definido los

pasos específicos que un programador debe atravesar antes de comprobar el código en las fuentes principales?

- ¿Los programadores programarán en parejas o individualmente, o alguna combinación de los dos?

Aseguramiento de la Calidad.

- ¿Los programadores escribirán casos de prueba para su código antes de escribir el código en sí?
- ¿Los programadores escribirán pruebas unitarias para el código independientemente si ellos escriben primeros o últimos?
- ¿Los programadores darán un paso a través de su código en el depurador antes de que lo registren?
- ¿Los programadores probarán la integración de su código antes de que la registren?
- ¿Los programadores revisarán o inspeccionarán el código de los demás?

Herramientas de Implementación y Gestión de Versiones.

- ¿Se ha seleccionado una herramienta de control de versiones?
- ¿Se ha seleccionado un lenguaje y versión del lenguaje o versión del compilador?
- ¿Se ha decidido si se debe permitir el uso de las características de lenguaje no estándar?
- ¿Se ha identificado y adquirido otras herramientas que se van a utilizar (editor, herramienta de refactoreo, un depurador, un marco de prueba, comprobador de sintaxis, etc.)?

2.1.2.5. Realizar el Diseño de la Implementación.

Se realizan 2 procesos para el diseño respectivo de la implementación.

Paso 1) Buscar objetos del mundo real.

Paso 2) Definir el diseño respectivo del la implementación.

BUSCAR OBJETOS DEL MUNDO REAL [5].

Se resume en forma general todos los aspectos que suceden con los objetos del dominio del problema y la interacción que pueden tener entre sí. Esto se extrae del diagrama de clases. Los pasos son los siguientes.

Paso 1) Determinar lo que puede ser hecho a cada objeto (cosas que se le podrían hacer).

Paso 2) Determinar lo que cada objeto puede hacer a otros objetos (su relación hacia otros objetos).

Paso 3) Determinar las partes de cada objeto que serán visibles a otros objetos (qué partes serán públicas y que partes serán privadas).

Paso 4) Definir la interfaz pública de cada objeto, incluyendo los servicios ofrecidos por cada objeto y si la clase del que depende el objeto tiene herencia con respecto a otras clases de sus objetos dependientes.

SALIDA:

Se tendrá la siguiente descripción de cada objeto, detallando lo que le sucede, lo que puede hacer a otros objetos, cómo son vistos por otros objetos, incluyendo sus interfaces.

Actividades:

- **Objeto 1.** Actividad 1, actividad 2,…, actividad ‘l’.
- **Objeto 2.** Actividad 1, actividad 2,…, actividad ‘m’.

.

.

.

- **Objeto ‘n’.** Actividad 1, Actividad 2,…, Actividad ‘o’.

Interacciones entre Objetos:

- Interacción (efecto que un objeto hace sobre otros) 1.
- Interacción 2.

.

.

.

- Interacción ‘p’.

Partes Visibles:

- Para cualquier Objeto 1 serán visibles el atributo 1, atributo 2,…, atributo ‘q’ de cualquier otro objeto.
- Para cualquier Objeto 2 (o 1) serán visibles el atributo 1, atributo 2,…, atributo ‘r’ de cualquier otro objeto.

.

.

.

- Para cualquier Objeto n (n-1,…, 2 o 1) serán visibles el atributo 1, atributo 2,…, atributo ‘s’ de cualquier otro objeto.

Interfaces y otras Relaciones:

- La Clase 1 es implementada/heredada por la interfaz/clase ‘x’.
- La Clase 2 es implementada/heredada por la interfaz/clase ‘y’ (o ‘x’).

.

.

.

- La Clase ‘n’ es implementada/heredada por la interfaz/clase que sea cualquiera excepto ‘n’.

Siendo ‘l’, ’m’, ’n’, ’o’, ‘p’, ‘q’, ‘r’, ‘s’, ‘x’, ‘y’ un entero positivo.

Esta es una reseña de cómo se organizará la información de los objetos del dominio del problema. Los detalles específicos son detallados en el diagrama de clases.

Casos Específicos.

Se tendrá que especificar las partes visibles para elementos específicos.

- Para cualquier Objeto ‘x’ (que sea Objeto ‘y’) serán visibles el atributo 1, atributo 2,…, atributo ‘n’ del Objeto ‘y’.

Siendo ‘x’, ’y’, ‘n’ número entero.

DEFINIR EL DISEÑO DEL SISTEMA [5].

Se procede al describir el diseño del sistema que va a implementarse por lo que se siguen los siguientes pasos:

Paso 1) Describir la arquitectura del sistema.

Paso 2) Describir sus subsistemas.

Paso 3) Mostrar el diagrama que describa el diseño estructural de los objetos del problema.

1) Describir la arquitectura del sistema.

Se describe la naturaleza del sistema a implementar (nativo o web), si se implementa en una computadora, servidor o en un dispositivo móvil. Además, se muestra el diagrama que describa su arquitectura.

2) Describir sus subsistemas.

Describir todos los subsistemas en los que se divide y el diagrama que los representa. Estos son los subsistemas en los que normalmente cualquier sistema se divide:

- Lógica de negocio.
- Interfaz de usuario.
- Acceso de base de datos.
- Dependencias del sistema.

3) Mostrar el diagrama de los objetos del problema.

Se describe el diseño estructural de los objetos del problema, normalmente se utiliza el diagrama de clases, las tarjetas CRC o cualquiera que describa a los objetos del problema.

SALIDA:

Se tendrá el diseño general del sistema, desde el nivel arquitectónico hasta el nivel de datos.

Nivel 1. Sistema Software.

<Descripción del sistema y diseño correspondiente>

Nivel 2. División en Subsistemas o Paquetes.

- **Lógica de Negocio.**

<Descripción y diseño correspondiente>

- **Interfaz de Usuario.**

<Descripción y diseño correspondiente>

- **Acceso a Base de Datos.**

<Descripción y diseño correspondiente>

- **Dependencias del Sistema.**

<Descripción>

Nivel 3. División en Clases.

<Diagrama que represente a las clases>.

2.1.2.6. Seleccionar los Lenguajes de Implementación.

Se procede a seleccionar el lenguaje de programación de acuerdo al uso que se le quiere dar al sistema, mediante la tabla 2.3 [5].

Tipo de Programa	Mejores Lenguajes	Peores Lenguajes
Procesamiento de Línea de Comandos	Cobol, Fortran, SQL	-
Desarrollo Multiplataforma	Java, Perl, Python.	Assembler, C#, Visual Basic.
Manipulación de Base de Datos	SQL, Visual Basic	Assembler, C
Manipulación Directa de la Memoria	Assembler, C, C++	C#, Java, Visual Basic
Sistemas Distribuidos	C#, Java	-
Uso de la Memoria Dinámica	C, C++, Java	-

Programa fácil de mantener	C++, Java, Visual Basic	Assembler, Perl
Ejecución Rápida	Assembler, C, C++, Visual Basic	JavaScript, Perl, Python
Para entornos con memoria limitada	Assembler, C	C#, Java, Visual Basic
Cálculo matemático	Fortran	Assembler
Proyectos de fácil implementación	Perl, PHP, Python, Visual Basic	Assembler
Programa en Tiempo Real	C, C++, Assembler	C#, Java, Python, Perl, Visual Basic.
Redacción de Reportes	Cobol, Perl, Visual Basic	Assembler, Java
Programa seguro	C#, Java	C, C++
Manipulación de Cadenas	Perl, Python	C
Desarrollo Web	C#, Java, JavaScript, PHP, Visual Basic	Assembler, C

Tabla 2.3. Los mejores y peores lenguajes para determinados tipos de programas.

2.1.2.7. Seleccionar las Consideraciones para Codificar.

Se realizan 2 procesos para las consideraciones respectivas en la implementación.

Paso 1) Identificación de patrones de diseño en el sistema.

Paso 2) Selección de herramientas de desarrollo.

IDENTIFICACIÓN DE PATRONES DE DISEÑO EN EL SISTEMA.

Se procede a identificar los patrones de diseño que se han establecido en el sistema. Se muestran estos patrones en la tabla 2. [5].

Patrón	**Descripción**
Fábrica Abstracta	Soporta la creación de conjuntos de objetos relacionados especificando el tipo de conjunto, pero no los tipos de cada objeto específico.

Adaptador	Convierte la interfaz de una clase a una interfaz diferente.
Puente	Construye una interfaz y una implementación de tal manera que cualquiera puede variar sin otra variación.
Decorador	Se fija responsabilidades a un objeto de forma dinámica, sin necesidad de crear subclases específicas para cada posible configuración de las responsabilidades.
Fachada	Proporciona una interfaz coherente al código que de otro modo no ofrecería una interfaz consistente.
Método de Fábrica	Crea instancias de clases derivadas de una clase base específica sin necesidad de realizar un seguimiento de las clases derivadas individuales en cualquier parte, pero si este método.
Iterador	Un objeto servidor que proporciona acceso a cada elemento de un conjunto secuencial.
Observador	Mantiene varios objetos en sincronía con cada otro al hacer un tercer objeto responsable de notificar al conjunto de objetos acerca de cambios a los miembros del conjunto.
Instancia Única	Proporciona acceso global a una clase que tiene una y sólo una instancia.
Estrategia	Define un conjunto de algoritmos o conductas que son dinámicamente intercambiables entre sí.
Método Plantilla	Define la estructura de un algoritmo, pero deja un poco de la implementación detallada a las subclases.

Tabla 2.4. Patrones de Diseño Populares.

SELECCIÓN DE HERRAMIENTAS DE DESARROLLO.

Se procede a seleccionar las herramientas que asistirán a la implementación del sistema. Se muestran tales herramientas mediante la siguiente lista de control [5].

- ¿Se tiene un ambiente de desarrollo integrado efectivo?
- ¿El IDE soporta lo siguiente?
  - Vista esquemática del programa.
  - Salto a definiciones de clases, rutinas y variables.
  - Formato del código fuente.
  - Juego de llaves (de comienzo y fin).

    - Búsqueda y reemplazo de cadenas de archivos múltiple.
    - Compilación conveniente.
    - Depuración integrada.
- ¿Se tienen herramientas que automaticen los refactoreos comunes?
- ¿Se está usando control de versiones para gestionar lo siguiente?
    - Código fuente.
    - Contenido.
    - Requerimientos.
    - Diseños.
    - Planes de Proyecto.
    - Otros artefactos del proyecto (especificar).
- ¿Si se está trabajando en un proyecto muy grande, se está usando un diccionario de datos o algún otro repositorio central que contenga descripciones acreditadas de cada clase usada en el sistema?
- ¿Se han considerado bibliotecas de código como alternativas para escribir código personalizado si está disponible?
- ¿Se está haciendo uso de un depurador interactivo?
- ¿Se utiliza 'make' u otro software de control de dependencias para desarrollar programas eficientemente y confiablemente?
- ¿El ambiente de pruebas incluye las siguientes cosas?
    - Marco de trabajo automatizado de pruebas.
    - Generadores de pruebas automatizados.
    - Monitores de cobertura.
    - Perturbadores de sistema.
    - Herramientas diferenciales.
    - Software de detección de defectos.
- ¿Se ha creado alguna herramienta personalizada que podría ayudar a dar soporte a las necesidades específicas del proyecto, especialmente herramientas que automaticen tareas repetitivas?
- ¿En general, el entorno se beneficia del soporte de la herramienta adecuada?

2.1.2.8. Preparar las Pruebas Unitarias.

Se realizan 4 procesos para el diseño respectivo de la implementación.

Paso 1) Identificar el diseño general de la clase.

Paso 2) Diseñar el pseudocódigo de cada rutina.

Paso 3) Diseñar los casos de prueba de cada rutina.

Paso 4) Seleccionar el marco de trabajo de pruebas.

IDENTIFICAR EL DISEÑO GENERAL DE LA CLASE [5].

Se procede a identificar el diseño general de cada clase que tiene un papel específico en el sistema. Sus pasos son los siguientes.

Paso 1) Identificar sus responsabilidades específicas.

Paso 2) Identificar que "secretos" la clase esconde.

Paso 3) Determinar exactamente que abstracción la interfaz de la clase captura.

Paso 4) Determinar si la clase será derivada de otra clase, y si las otras clases les será permitido derivarse de ella.

Paso 5) Identificar sus métodos públicos clave.

SALIDA:

Se tendrá la descripción del diseño general de la clase a implementar.

Clase: <Nombre de la clase>.

Responsabilidades Específicas.

- <Responsabilidad 1>.
- <Responsabilidad 2>.

.

.

.

- <Responsabilidad 'l'>.

Secretos.

- <Secreto 1>.

- <Secreto 2>.

.

.

.

- <Secreto ‘m’>.

Abstracción Capturada por la Clase.

<Atributo 1>, <atributo 2>,…, <atributo ‘n’>.

Clase Principal:

<Clase que la hereda> o Ninguna.

Clases Herederas:

<Clase 1>, <clase 2>,…, <clase ‘o’> o Ninguna.

Métodos Públicos Clave:

- <Método 1>.
- <Método 2>.

.

.

.

- <Método ‘p’>.

Siendo ‘l’, ’m’, ’n’, ’o’, ’p’ un entero positivo.

DISEÑAR EL PSEUDOCÓDIGO DE CADA RUTINA [5].

Se procede a transformar en pseudocódigo cualquier diagrama que represente la rutina. Los pasos son los siguientes.

Paso 1) Mostrar su diseño.

Paso 2) Verificar los prerrequisitos.

Paso 3) Definir el problema que resolverá.

Paso 4) Determinar el manejo de errores.

Paso 5) Determinar la posibilidad de eficiencia.

Paso 6) Investigar funcionalidad disponible en las bibliotecas estándares.

Paso 7) Investigar los algoritmos.

Paso 8) Escribir el pseudocódigo.

Paso 9) Intentar algunas ideas en pseudocódigo y repetir de ser necesario.

1) Mostrar su diseño.

Se muestra el diseño en el que basa la rutina. Normalmente es el diagrama de actividades en el caso de UML o es el diagrama de flujo, entre otros.

2) Verificar los prerrequisitos.

Se verifica que el trabajo de la rutina sea bien definido y que encaje en el diseño general.

3) Definir el problema que resolverá.

Se describe acerca de lo que hace la rutina para resolver el problema que describe. Además, se describen los datos generales de la rutina que son los siguientes.

- La información que la rutina esconderá.
- Entradas.
- Salidas.
- Precondiciones.
- Poscondiciones.

4) Determinar el manejo de errores.

Se debe determinar la posibilidad de que algo pueda salir mal al momento de ejecutar la rutina, y manejar los errores causados por las fallas en la rutina o usarlas para que el sistema responda a tales fallas cuando sucedan.

5) Determinar la posibilidad de eficiencia.

Se determina la posibilidad de mejorar el diseño de la rutina con el fin de que mejore el rendimiento general del sistema.

6) Investigar funcionalidad disponible en las bibliotecas estándares.

Se investiga si en el mismo lenguaje de programación se pueden usar funciones que provienen de bibliotecas estándares verificando sino se ha desarrollado más de lo necesario.

7) Investigar los algoritmos.

Se investiga información de algoritmos con el fin de verificar lo que ya esté disponible.

8) Escribir el pseudocódigo.

Se escribe el pseudocódigo en lenguaje de alto nivel, es decir, se escribe las cosas que va a hacer la rutina sin relación al lenguaje de programación. Además, se escribe lo que hará la rutina, debido a que ello irá en el encabezado de la rutina.

9) Determinar los datos a manejarse.

Se determinan los tipos de datos que participarán en la rutina.

10) Intentar algunas ideas en pseudocódigo y repetir de ser necesario.

Se refina en pseudocódigo de la rutina en caso de que haya como mejorar el rendimiento de esta.

SALIDA:

Se tendrá la descripción del pseudocódigo de la rutina donde se enfoca su propósito y sus datos específicos.

Nombre de la Rutina: <nombre>.
Clase: <clase a la que pertenece>.
Diseño:
<Diagrama que lo representa>.
Prerrequisitos:

- <Prerrequisito 1>.
- <Prerrequisito 2>.

.

.

.

- <Prerrequisito ‘l’>.

Problema a Resolver:

a) Detalles.

- <Detalle 1>.
- <Detalle 2>.

.

.

.

- <Detalle ‘m’>.

b) Datos.

- <Información que la rutina esconderá>.
- <Entradas>.
- <Salidas>.
- <Precondiciones>.
- <Pos condiciones>.

Manejo de Errores.

<La descripción de los posibles errores y la manera en la que se los manejarán>.

Posibilidad de Eficiencia.

<La descripción de las maneras en las que se puede mejorar la rutina>, N/A (No Aplica) o N/D (No Disponible).

Funcionalidad Disponible en Bibliotecas Estándares.

<Descripción de la bibliotecas y lo que va a usarse de ellas> o N/D (No Disponible).

Algoritmos Disponibles:

<Nombre y descripción del (o los) algoritmos>, N/A (No Aplica) o N/D (No Disponible).

Pseudocódigo:

```
<Descripción de la rutina>
Inicio <Nombre de rutina> (parámetro 1, parámetro 2, …, parámetro 'n')
   <Cuerpo>
Fin
```

```
<Otro pseudocódigo del mismo nombre>
```

Datos a Manejarse:

- <Tipo de dato, estructura o colección 1>.
- <Tipo de dato, estructura o colección 2>.

.

.

.

- <Tipo de dato, estructura o colección 'o'>.

Siendo 'l', 'm', 'n', 'o' un entero positivo.

DISEÑAR LOS CASOS DE PRUEBA DE CADA RUTINA. [5]

Para diseñar las pruebas se requiere el diseño de la rutina para diseñar toda alternativa de prueba para esta. Se realizan los siguientes pasos.

Paso 1) Empezar con una prueba por el camino recto a través de la rutina.

Paso 2) Añadir una prueba por cada una de las siguientes palabras clave o sus equivalentes: 'if', 'while', 'repeat', 'for', 'and' y 'or'.

Paso 3) Añadir una prueba por cada caso en una sentencia 'case'. Si la sentencia 'case' no tiene un caso por defecto, añadir una prueba más.

Elaboración de la Prueba [9].

Paso 1) Ingresar el número del caso.

Paso 2) Describir el propósito y detalles de la prueba.

Paso 3) Ingresar los datos de la pruebas.

Paso 4) Ingresar el resultado esperado.

SALIDA:

Por cada rutina se tendrá la siguiente tabla de todas las pruebas posibles en ella.

Rutina: <Nombre de la rutina>.

# de Caso	Descripción	Datos de Entrada	Resultado Esperado
1	<Propósito y detalles>.	<Dato 1>. <Dato 2>. … <Dato 'l'>.	<Valor resultante de la rutina o un comportamiento esperado>.
2	<Propósito y detalles>.	<Dato 1>. <Dato 2>. … <Dato 'm'>.	<Valor resultante de la rutina o un comportamiento esperado>.
. . .			
N	<Propósito y detalles>.	<Dato 1>. <Dato 2>. … <Dato 'o'>.	<Valor resultante de la rutina o un comportamiento esperado>.

Siendo 'l', 'm', 'n', 'o' un entero positivo.

Tabla 2.5. Prueba de la Rutina.

SELECCIONAR EL MARCO DE TRABAJO DE PRUEBAS [8].

Se procede a seleccionar el marco de trabajo de pruebas disponible actualmente en función del lenguaje de programación con el que se trabaja.

2.1.2.9. Preparar las Pruebas de Integración.

Se procede a realizar un plan de integración en el que trazará la estrategia para integrar las partes al sistema general estos son los pasos [9]:

Paso 1) Determinar cómo se llevarán a cabo las pruebas.

Paso 2) Determinar las cosas a ser probadas.

Paso 3) Determinar roles y responsabilidades.

Paso 4) Describir los prerrequisitos para comenzar las pruebas.

Paso 5) Describir el ambiente de pruebas.

Paso 6) Describir los supuestos de las pruebas de integración.

Paso 7) Determinar lo que hay que hacer cuando las pruebas tengan éxito.

Paso 8) Determinar lo que hay que hacer cuando las pruebas fallan.

Paso 9) Determinar si debe constar un glosario en la implementación del sistema.

SALIDA:

Se tendrá un plan de integración donde constará la forma de cómo se llevará la integración.

Ejecución de las Pruebas:

<La forma en la que estas pruebas van a llevarse a cabo>.

Lista de Cosas a Ser Probadas:

<Característica 1>.

<Característica 2>.

…

<Característica 'l'>.

Prerrequisitos:

- <Prerrequisito 1>.
- <Prerrequisito 2>.

…

- <Prerrequisito 'm'>.

Ambiente de Pruebas:

<Descripción del lugar en donde se desenvolverán las pruebas>.

Supuestos:

<La situación esperada al momento que se ejecuten las pruebas>.

Actividades en Caso de Éxito de la Prueba:

- <Actividad 1>.
- <Actividad 2>.

…

- <Actividad ‘n’>.

Actividades en Caso de Fallo de la Prueba:

- <Actividad 1>.
- <Actividad 2>.

…

- <Actividad ‘o’>.

Glosario:

N/A (No aplica) porqué <justificación> o.

- <Término 1>.
- <Término 2>.

…

- <Término ‘p’>.

Siendo ‘l’, ’m’, ’n’, ’o’, ‘p’ un entero positivo.

Se tomarán las pruebas unitarias cómo referencia para las pruebas de integración, las cuales se implementarán antes de que se integre el código al sistema general.

NOTA:

Solo las pruebas de integración tienen sentido si el sistema se desarrolla en varios módulos.

2.1.2.10. Considerar el Reuso.

En caso de que se requiera el reuso en la implementación, se debe proceder a elegir si es que se lo va a considerar y su justificación por lo que se determinará si es que el reuso se lo aplicará en la implementación.

Consideración del Reuso.

<Si / No>.

Justificación.

<Razones por las que se realizaría o no>.

A continuación, cuando se decida realizar el reuso en la implementación, se seguirán estos pasos para el reuso:

Paso 1) Selección de unidades reutilizables.

Paso 2) Verificación de la posibilidad de reuso.

Paso 3) Comunicar información de las partes en reutilización.

2.1.2.10.1. Selección de Unidades Reutilizables.

Para la selección de las unidades reutilizables se proceden a seguir los siguientes pasos:

Paso 1) Seleccionar el nivel de reuso.

Paso 2) Seleccionar las técnicas de reuso.

1) Seleccionar el nivel de reuso.

Se selecciona el nivel de reuso en el que se va a operar para la implementación del sistema, estos son los niveles [4]:

Reutilización de Sistemas de Aplicaciones	La totalidad de un sistema de aplicaciones puede ser reutilizada incorporándolo sin ningún cambio en otros sistemas, configurando la aplicación para diferentes clientes o desarrollando familias de aplicaciones que tienen una arquitectura común pero que son adaptadas a clientes particulares.
Reutilización de Componentes	La reutilización de componentes de una aplicación varía en tamaño desde subsistemas hasta objetos simples.
Reutilización de Objetos y Funciones	Pueden reutilizarse componentes software que implementan una única función.

Tabla 2.6. Niveles de Reuso.

2) Seleccionar las técnicas de reuso.

Se selecciona las técnicas que van a usarse en el reuso, estas son:

Técnica	**Descripción**
Desarrollo basado en componentes	Los sistemas se desarrollan integrando

	componentes (colecciones de objetos) que cumplen los estándares de modelado de componentes.
Marcos de aplicaciones	Las colecciones de clases completas y abstractas pueden adaptarse y extenderse para crear sistemas de aplicaciones.
Envoltura de sistemas heredados	Sistemas heredados que pueden ser 'envueltos' definiendo un conjunto de interfaces y proporcionando acceso a estos sistemas heredados a través de estas interfaces.
Sistemas orientados a servicios	Los sistemas se desarrollan enlazando servicios compartidos, que pueden ser proporcionados de forma externa.
Líneas de productos de aplicaciones	Un tipo de aplicación se generaliza alrededor de una arquitectura común para que pueda ser adaptada para diferentes clientes.
Integración COTS	Los sistemas se desarrollan integrando sistemas de aplicaciones existentes.
Aplicaciones verticales configurables	Un sistema genérico se diseña para que pueda configurarse para las necesidades de clientes de sistemas particulares.
Librerías de programas	Están disponibles para reutilización las librerías de funciones y de clases que implementan abstracciones comúnmente usadas.
Generadores de programas	Un sistema generador incluye conocimiento de un tipo de aplicación particular y puede generar sistemas o fragmentos de un sistema en ese dominio.
Desarrollo de software orientado a aspectos	Componentes compartidos entrelazados en una aplicación en diferentes lugares cuando se compila el programa.

Tabla 2.7. Técnicas de Reuso [4].

SALIDA:

Se tendrá la descripción de las partes reutilizables en donde constará el nivel y las técnicas de reuso.

Nivel de Reuso:

<Nivel de reuso seleccionado>.

Técnicas de Reuso:

- <Técnica 1>.
- <Técnica 2>.

.

.

.

- <Técnica ‘n’>.

Siendo ’n’ un entero positivo.

2.1.2.10.2. Verificación de la Posibilidad de Reuso.

Se procede verificar los aspectos para buscar la posibilidad de reuso, estos son los pasos [4]:

Paso 1) Verificar la agenda de desarrollo de software.

Paso 2) Verificar la vida esperada del software.

Paso 3) Verificar los conocimientos, habilidades y experiencia del grupo de desarrollo.

Paso 4) Revisar la criticidad del software y sus requerimientos no funcionales.

Paso 5) Verificar el dominio del sistema.

Paso 6) Verificar la plataforma sobre la cual se ejecuta el sistema.

Paso 7) Decidir sobre la posibilidad de reuso y su justificación.

1) Verificar la agenda de desarrollo de software.

Se verifica si se utiliza componentes individuales en caso de que haya tiempo para implementación y de no haber se reusan sistemas para optimizar el tiempo.

2) Verificar la vida esperada del software.

Se verifica si es que el sistema será uno que se utilizará a largo plazo por lo que se debe explicar las implicaciones que traerá el reuso a largo plazo, si es a corto plazo se debe explicar sus implicaciones inmediatas.

3) Verificar los conocimientos, habilidades y experiencia del grupo de desarrollo.

Se revisa si es que el equipo de desarrollo tiene experiencia en lo que va a reusar explicando los antecedentes de experiencia previa.

4) Revisar la criticidad del software.

Se revisa la categoría a la que pertenece el sistema a desarrollar y de ser crítico se debe contar con los respectivos certificados de las partes reutilizadas.

5) Verificar el dominio del sistema.

Si se trabaja con dominios de aplicaciones, es decir, sistemas genéricos, se debe describir sus características y la forma en la que se adaptaría.

6) Verificar la plataforma sobre la cual se ejecuta el sistema.

Se debe explicar la plataforma sobre la que se desarrolla el sistema y verificar si las partes que van a reusarse son desarrolladas en un lenguaje que pertenece a la plataforma del sistema a implementar, si el lenguaje pertenece a otra plataforma y si el lenguaje no depende de la plataforma para ser reusados.

7) Decidir sobre la posibilidad de reuso y su justificación.

Se decide basándose en todo los aspectos de la posibilidad de reuso proceder a realizar el reuso o no, y también la justificación de la decisión final.

SALIDA:

Se tendrá la descripción de la posibilidad de reuso donde se la analiza para concluir con su realización.

Agenda de Desarrollo de Software:

<Agenda corta / Agenda disponible>.

Vida esperada del software:

Sistema a <corto / largo> plazo, <sus implicaciones>.

Conocimientos, habilidades y experiencia del grupo de desarrollo:

<Experiencia y antecedentes de las partes reutilizadas con respecto al grupo de desarrollo>.

Criticidad del software.

<Categoría del software y de ser necesario, imágenes de certificación de las partes reutilizadas>.

Dominio del sistema.
N/A (No Aplica) o <Características y forma de adaptar el sistema genérico>.

Plataforma sobre la cual se ejecuta el sistema.
<Plataforma sobre la que se desarrolla el sistema> y <lenguaje y plataforma (si pertenece a ella) en la que corre el sistema>.

Conclusión.
<La decisión final de la reutilización y su justificación>.

2.1.2.10.3. Comunicar Información de las Partes en Reutilización.

Se procede a seleccionar el mecanismo adecuado para comunicar la información de las partes de reutilización. Estos son [6]:

Mecanismo de Retroalimentación.
Comunica el uso y el impacto de los productos software y recursos en el proyecto.

Mecanismo de Comunicación.
Resuelve problemas, responde preguntas y hace recomendaciones respecto a los productos software y recursos que el proyecto encuentra.

SALIDA:
Se tendrá la selección del mecanismo, su justificación y su forma de implementación (se recomienda usar ambos si se trata de un equipo de desarrollo).

Mecanismo:
<Retroalimentación, Comunicación o Ambos>.

Justificación:

<Razones por las que va a usarse el o los mecanismos>.

Implementación:
<Formas de cómo de implementará el o los mecanismos>.

2.1.3. SELECCIONAR METRICAS Y SUS TECNICAS.

Para realizar el análisis de resultados, se requiere utilizar métricas para la implementación así como elegir las técnicas para aplicarlas. Estos son los pasos a seguir:

Paso 1) Seleccionar métricas de implementación.
Paso 2) Seleccionar técnicas de medición.

2.1.3.1. Seleccionar Métricas de Implementación.

Se procede a elegir las métricas (una como mínimo) para medir la implementación, estas son las métricas divididas en estas categorías [5].

MÉTRICAS CON RESPECTO AL TAMAÑO [5].

- Total de líneas de código escritas.
- Total de líneas de comentario.
- Total de número de clases y rutinas.
- Total de declaraciones de datos.
- Total de líneas en blanco.

MÉTRICAS CON RESPECTO A LA PRODUCTIVIDAD [5].

- Horas de trabajo dedicadas al proyecto.
- Horas de trabajo dedicadas en cada clase o rutina.
- Número de veces en que cada clase o rutina ha cambiado.
- Dólares invertidos en cada proyecto.
- Dólares invertidos por cada línea de código.
- Dólares invertidos por proyecto.

MÉTRICAS CON RESPECTO AL SEGUIMIENTO DE DEFECTOS [5].

- Severidad de cada defecto.
- Localización de cada defecto (clase o rutina).
- Origen de cada defecto (requerimientos, diseño, implementación, pruebas).
- Manera en que cada defecto es corregido.
- Persona responsable por cada defecto.
- Número de líneas afectadas por cada corrección de defecto.
- Horas de trabajo dedicadas a corregir cada defecto.
- Tiempo promedio requerido en encontrar un defecto.
- Tiempo promedio requerido en arreglar un defecto.
- Número de intentos hechos para corregir cada defecto.
- Número de nuevos errores resultantes de la corrección de defectos.

MÉTRICAS CON RESPECTO A LA CALIDAD GLOBAL [5].

- Número total de defectos.
- Número de defectos en cada clase o rutina.
- Promedio de fallos por miles de líneas de código.
- Tiempo medio entre fallos.
- Errores detectados por compilador.

MANTENIBILIDAD [5].

- Número de rutinas públicas en cada clase.
- Número de parámetros pasados a cada rutina.
- Número de rutinas privadas y/o variables en cada clase.
- Número de variables locales usadas por cada rutina.
- Número de rutinas llamadas por cada clase o rutina.
- Número de puntos de decisión en cada rutina.
- Control de flujo de complejidad en cada rutina.
- Líneas de código en cada clase o rutina.
- Líneas de comentario en cada clase o rutina.
- Número de declaraciones de datos en cada clase o rutina.
- Número de líneas en blanco en cada clase o rutina.
- Número de ‘gotos’ en cada clase o rutina.
- Número de sentencias de entrada y salida en cada clase o rutina.

2.1.3.2. Seleccionar Técnicas de Medición.

Se estima el tamaño de la implementación y el esfuerzo necesario para completarla en estas siguientes formas [5]:

- Utilizar software de planificación.
- Utilizar un enfoque algorítmico.
- Tener expertos externos para estimar proyecto.
- Estimar las piezas del proyecto, y luego añadirlas.
- Que la gente estime sus propias piezas, y luego añadirlas.
- Estimar el tiempo que se necesita para todo el proyecto, y luego dividir el tiempo entre todas las piezas.
- Referirse a la experiencia en proyectos anteriores.
- Mantener las estimaciones anteriores y ver lo exactos que eran. Utilizarlos para ajustar nuevas estimaciones.

SALIDA:

Se tendrá la selección de las métricas y así mismo de sus técnicas.

Métricas de Implementación:

- <Métrica 1>.
- <Métrica 2>.

.

.

.

- <Métrica 'm'>.

Técnicas de Medición:

- <Técnica 1>.
- <Técnica 2>.

.

.

.

- <Técnica ‘n’>.

Siendo ‘m’, ’n’ un entero positivo.

2.1.4. IMPLEMENTACION Y PRUEBAS UNITARIAS.

Se procede a implementar todas las clases del sistema como tal por lo que se seguirán estos pasos:

Paso 1) Revisar el diseño general para la clase y armarla.
Paso 2) Construir las rutinas dentro de la clase.
Paso 3) Revisar y probar la clase en conjunto.

1) Revisar el diseño general para la clase y armarla.

Se procede a revisar los diseños que han sido identificados de las clases que se ha realizado en el paso 2.1.2.8 “Preparar las Pruebas Unitarias” para implementar la estructura de la clase. Si se regresa a este paso, es posible que se tenga que volver a identificar el diseño la clase.

2) Construir las rutinas dentro de la clase.

Mediante el pseudocódigo de cada rutina y también las pruebas de unidad se proceden a implementar cada rutina que contenga la clase. Se procede a seguir estos pasos para implementar cada rutina.

Paso 1) Escribir la declaración de la rutina.
Paso 2) Convertir el pseudocódigo en comentarios de alto nivel.
Paso 3) Realizar el Proceso de DGP con PPP con todas las pruebas unitarias anteriormente diseñadas.

Proceso de DGP con PPP:

Paso 1) Añadir una prueba unitaria en función del caso de prueba que se realice.
Paso 2) Llenar con código cada comentario correspondiente a la prueba.
Paso 3) Ejecutar todas las pruebas (incluyendo la actual).
Paso 4) Si la prueba no tuvo éxito, arreglar el código y volver a ejecutar las pruebas hasta que todas las pruebas tengan éxito.

Paso 5) Refactorar el código.

3) Revisar y probar la clase en conjunto.

Después de que la clase ha sido implementada se la prueba ejecutando todas las pruebas de unidad de todas las rutinas de la clase. Al final se revisa como ha quedado la clase a implementar.

PROCESO DE PROGRAMACION POR PSEUDOCODIGO [5].

En caso de que tenga que realizarse una implementación en la que las pruebas unitarias no puedan tener acceso a las rutinas, entonces se realiza este proceso para obtener el código documentado para mantenimiento. Estos son los pasos.

Paso 1) Verificar el diseño previo de las rutinas y de la clase.
Paso 2) Convertir el diseño de las rutinas en pseudocódigo.
Paso 3) Escribir la declaración de las rutinas.
Paso 4) Convertir el pseudocódigo en comentarios de alto nivel.
Paso 5) Llenar con código cada comentario de la rutina.
Paso 6) Verificar y refactorar el código.

2.1.5. VERIFICACION.

Después de que se ha implementado el código, lo siguiente es verificar lo implementado, por lo que se siguen los siguientes pasos [5]:

Paso 1) Verificar mentalmente las rutinas por errores.
Paso 2) Compilar las rutinas.
Paso 3) Recorrer el código en el depurador.
Paso 4) Si existen errores (o advertencias), removerlos de las rutinas.

1) Verificar mentalmente las rutinas por errores.

Se procede a verificar el estado de la rutina para ver si contiene algún error por lo se puede practicar las “pruebas de escritorio”, inspecciones, etc.

2) Compilar las rutinas.

Cuando se compile las rutinas se debe asegurar que el compilador opere en el nivel más exhaustivo posible y eliminar toda causa de error o advertencia del compilador.

3) Recorrer el código en el depurador.

Cuando se lo haga, se lo tiene que hacer con cada línea de código para asegurarse que cada línea haga lo que tenga que hacer. En el proceso se encuentran los errores que hay que corregir.

4) Si existen errores, removerlos de la rutina.

Una vez que los errores han sido detectados se los remueve, sin embargo, si se encuentra que la rutina es defectuosa se recomienda empezar de nuevo y volverla a codificar.

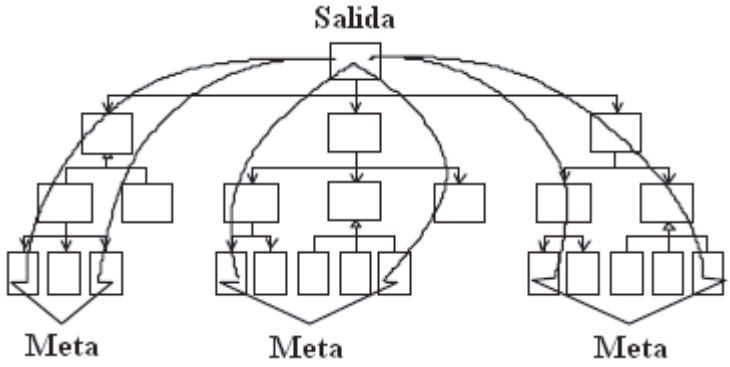
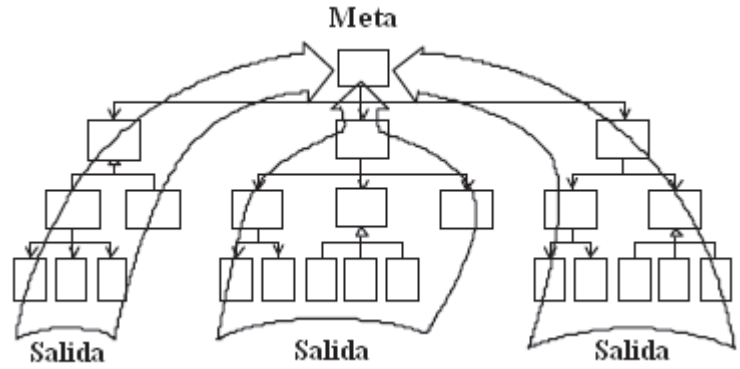
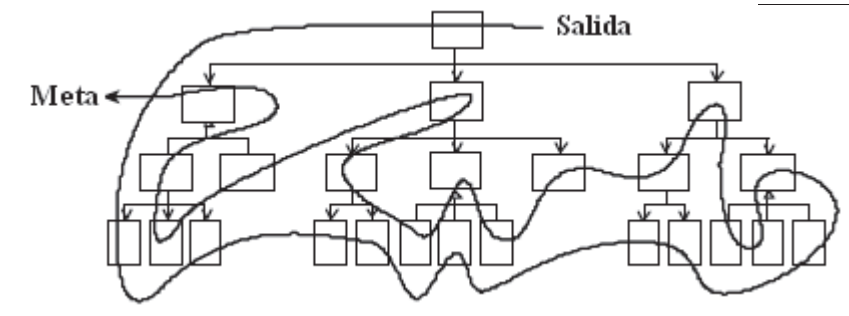
2.1.6. INTEGRACION.

En el caso de que varias personas estén implementando un sistema, se procede a elegir la estrategia de implementación para el caso correspondiente, estas son las estrategias.

Integración Descendente	En la integración descendente, la clase en el tope de la jerarquía es escrita e integrada primero. El tope es la ventana principal, el bucle de control de aplicaciones, el objeto que contiene la rutina de ejecución principal. Los fragmentos de código tienen que ser escritos para ejercer la clase superior. De ahí, como clases son integradas ascendentemente, los fragmentos de código de clases son remplazados por clases reales.
Integración Ascendente	Se escriben y se integran las clases en el fondo de la jerarquía primero. Añadir las clases de bajo nivel una a la vez en lugar de todas a la vez es lo que hace de la integración ascendente una estrategia incremental de integración. Se escriben controladores de pruebas para ejercitar las clases de bajo nivel iniciales y se añaden clases a la estructuración de controladores de prueba como estos sean desarrollados. A medida que se añaden clases de alto nivel, se reemplazan las clases controladoras por clases reales.
Integración de Emparedado	Se integra primero las clases de objetos de negocio de alto nivel en el tope de la jerarquía. Después se integra las clases de interfaces de dispositivo y las clases de utilidad ampliamente utilizadas en el fondo. Estas clases de alto y bajo nivel son el pan del emparedado. Se deja las clases de nivel medio hasta después.
Integración Orientada a	En la integración orientada a riesgos, se identifica el nivel de riesgo asociado con cada clase. Se decide cuales serán las partes más

Riesgos	desafiantes para implementar, y se las implementa primero.
Integración Orientada a Características	Se utilizan estrategias de integración incremental recursivamente integrando pequeñas piezas para formar características y después integrar incrementalmente las características para formar un sistema.
Integración en 'T'	En este enfoque, una sección vertical específica es seleccionada por desarrollo temprano e integración. Estas secciones podrían ejercitar el sistema de extremo a extremo, y podrían ser capaces de eliminar cualquier problema principal en las suposiciones del diseño del sistema. Una vez que la sección vertical ha sido implementada (y cualquier problema asociado haya sido corregido), después la amplitud total del sistema puede ser desarrollada (como el menú del sistema en una aplicación de escritorio).

Tabla 2.8. Estrategias de Integración [5].

Integración Descendente	
Integración Ascendente	
Integración de Emparedado	

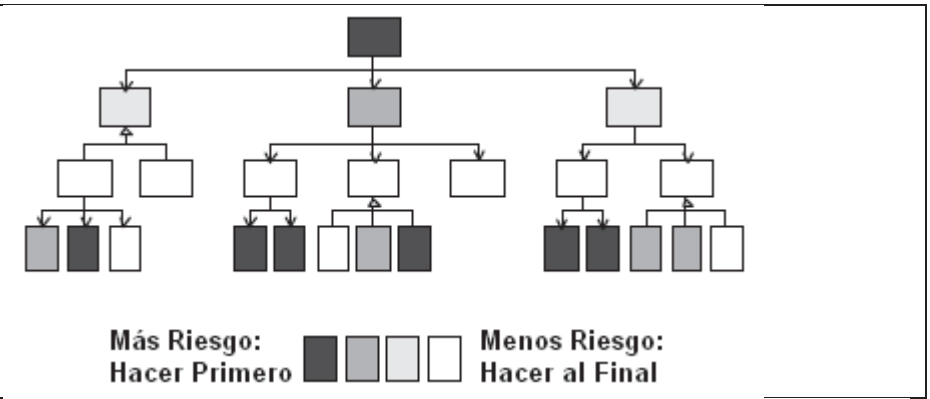
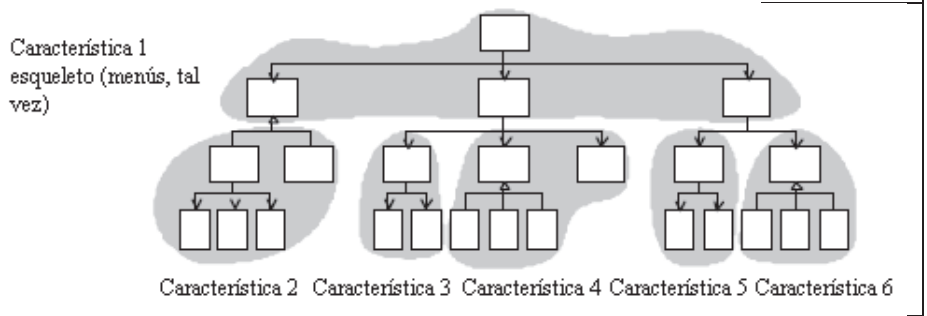
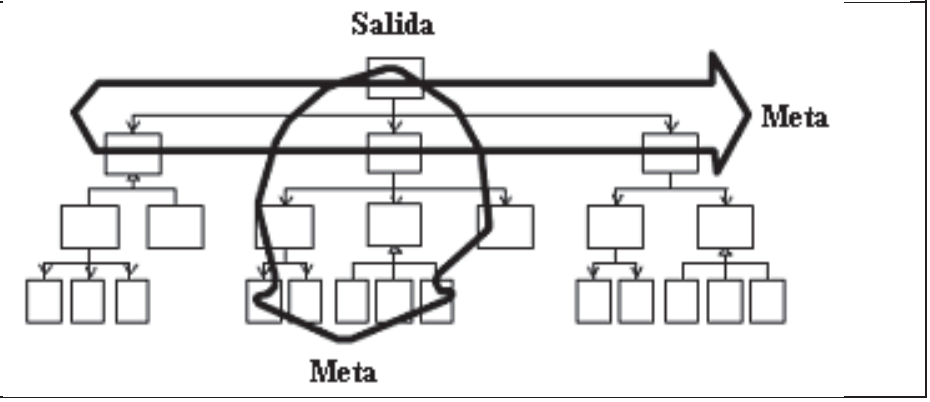
Integración Orientada a Riesgos	
Integración Orientada a Características	
Integración en 'T'	

Tabla 2.9. Formas de Implementar las Estrategias de Integración [5].

2.1.7. PRUEBAS DE INTEGRACION.

Mediante la integración incremental, se escribe y se prueba la aplicación en piezas pequeñas y se combinan las piezas una a la vez. Se siguen los siguientes pasos [5] [9] [10].

1) Desarrollar una parte pequeña y funcional del sistema. Puede ser la parte funcional más pequeña, la parte más difícil, una parte clave o alguna combinación. Se debe probarla y depurarla minuciosamente. Servirá de esqueleto en el que se cuelguen los músculos, nervios y la piel que conforman las partes restantes del sistema.

2) Diseñar, codificar, probar y depurar una clase (para eso se siguen los siguientes pasos).

- **a.** Crear el caso de prueba (similar a la creación de las pruebas de unidad).
- **b.** Ejecutar todas las pruebas y ver si la nueva prueba falla.
- **c.** Integrar la parte.
- **d.** Ejecutar todos los casos de prueba (incluyendo la nueva) viendo que todas tengan éxito.
- **e.** Si existe algún defecto en el software hallado por las pruebas, arreglarlo y volver a probar.
- **f.** Refactorar el código.
- **g.** Repetir el proceso con otra parte a integrar.

2.2 METODOLOGÍAS Y ESTÁNDARES UTILIZADOS.

2.2.1. DESARROLLO GUIADO POR PRUEBAS (DGP) [10].

Es un proceso de desarrollo de software que depende en la repetición de un ciclo de desarrollo muy corto: primero el desarrollador escribe un (inicialmente fallido) caso de prueba automatizado que define una mejora deseada o una función, de ahí produce la mínima cantidad de código para aprobar la prueba, y finalmente refactora el nuevo código a los estándares aceptables.

2.2.2. PROCESO DE PROGRAMACION POR PSEUDOCODIGO (PPP) [5].

Define un enfoque específico para usar el pseudocódigo para coordinar la creación de código entre las ruinas en el que este se lo escribe primero en forma de comentarios mediante lenguaje sencillo, de ahí se rellena la rutina de código alrededor del pseudocódigo haciendo que la rutina sea entendible.

2.2.3. ESTÁNDAR IEEE PARA TECNOLOGÍAS DE LA INFORMACIÓN – PROCESOS DEL CICLO DE VIDA DEL SOFTWARE – PROCESOS DE REUSO [6].

Este estándar especifica los procesos, actividades y tareas a ser aplicadas durante cada fase del ciclo de vida del software para permitir un producto software a ser construido desde recursos reutilizables. También especifica los procesos,

actividades y tareas para permitir la identificación, construcción, mantenimiento y gestión de los recursos suplidos.

2.3 HERRAMIENTAS UTILIZADAS.

2.3.1. AMBIENTE DE DESARROLLO INTEGRADO (ADI) [11].

Es una aplicación software que provee facilidades comprensivas para programadores para desarrollo de software. Un ADI normalmente consiste en un editor de código, herramientas de automatización de implementación y un depurador.

Los ADI son diseñados para maximizar la productividad del programador proveyendo componentes muy unidos con interfaces de usuario similares. Los ADI presentan un programa simple en el cuál todo el desarrollo está hecho. Esta aplicación típicamente provee algunas características para software de autoría, modificación, compilación, despliegue y depuración. Esto contrasta con software de desarrollo usando herramientas no relacionadas.

Un objetivo de los ADI es de reducir la configuración necesaria para unir múltiples utilidades de desarrollo para en lugar ello proveer el mismo conjunto de prestaciones como una unidad cohesiva. Reducir el tiempo de instalación que puede incrementar la productividad del desarrollador, en casos donde aprender a usar el ADI es más rápido que integrar manualmente todas las herramientas individuales. La integración más fuerte de todas las tareas de desarrollo tiene el potencial de mejorar la productividad global más allá de solo ayudar con las tareas de instalación.

2.3.2. MARCOS DE PRUEBAS UNITARIAS [12].

Es un conjunto de suposiciones, conceptos y herramientas que proveen soporte para las pruebas automatizadas de software. La principal ventaja de tal marco de trabajo es el costo bajo de mantenimiento. Si hay cambio a algún caso de prueba entonces solo el archivo de caso de prueba necesita ser actualizado de ahí el resto de sus componentes se mantienen igual.

Escoger el marco de trabajo correcto ayuda en mantener bajos costos. Los costos asociados con la elaboración de pruebas son debido a los esfuerzos de desarrollo y mantenimiento.

2.3.3. HERRAMIENTAS 'CASE' [13].

Son una clase de software que automatizan algunas de las actividades involucradas en varias fases del ciclo de vida. Por ejemplo, cuando se establecen los requerimientos funcionales de una aplicación propuesta, las herramientas de prototipado pueden ser utilizadas para desarrollar modelos gráficos de pantallas de aplicación para asistir a los usuarios finales para visualizar como una aplicación lucirá después del desarrollo. Subsecuentemente, los diseñadores del sistema pueden utilizar herramientas de diseño automatizadas para transformar los requerimientos funcionales prototipados en documentos de diseño detallados. Los programadores pueden entonces utilizar generadores automatizados de código para convertir los documentos de diseño en código.

2.3.4. LENGUAJE DE MODELADO UNIFICADO (UNIFIED MODELING LANGUAGE 'UML') [14].

Es un lenguaje de modelado de propósito general en el campo de la ingeniería de software. Este incluye un conjunto de técnicas de notación gráficas para crear modelos visuales de sistemas software orientados a objetos intensivos. Combina técnicas de modelado de datos, modelado de negocio y modelado de componentes. Puede ser usado con todos los procesos, a lo largo del ciclo de vida de desarrollo de software, y a través de diferentes tecnologías de implementación.

CAPÍTULO 3. APLICACIÓN DEL MÉTODO A UN CASO DE ESTUDIO.

3.1 DESCRIPCIÓN DEL CASO DE ESTUDIO.

NOMBRE DEL CASO DE ESTUDIO.

Sistema de Préstamos de los Laboratorios del DICC.

DESCRIPCION.

El sistema ayudará a gestionar los préstamos entrantes y salientes de todas las cosas que se prestan de parte de los laboratorios con el fin de controlar los préstamos y llevar un registro de las personas que han pedido el préstamo de uno o más objetos del laboratorio para fines académicos.

3.2 APLICACIÓN DEL MÉTODO EN EL DESARROLLO DE UN PROTOTIPO.

3.2.1. SELECCIÓN DEL MODELO DE IMPLEMENTACION.

Selección del Modelo de Implementación.

Se elegirá RUP, debido a que es un modelo iterativo e incremental y se lo utilizará para iterar varias veces el proceso de implementación del sistema software.

3.2.2. PLANIFICAR LA IMPLEMENTACION.

3.2.2.1. Preparar la Planificación.

PRIMERA PLANIFICACION DEL SISTEMA.

Costo de Desarrollo:

Este sistema, debido a que lo usarán los auxiliares de laboratorio, entonces será un sistema de escritorio que no requerirá un costo alto de desarrollo debido a su simplicidad en la arquitectura del sistema a implementar.

Riesgo Incurrido:

Su nivel de confiabilidad es Nominal y su riesgo incurre en que su fallo incurriría en pérdida de información para los usuarios.

Funcionalidad Principal del Sistema:

El sistema registrará los préstamos así cómo registrará que se hayan devuelto.

Documentos de Diseño:

- Diagramas de casos de uso.
- Diagramas de clases.
- Diagramas de actividades.
- Diagrama de Despliegue.
- Diseño relacional de la base de datos.
- Interfaces del sistema.

Participantes.

- **Equipo de Desarrollo.** 1 Desarrollador.
- **Negocio.** 1 Auxiliar.

Característica a Implementar:

Implementación de las interfaces.

Próxima Característica:

Conexión entre Interfaces.

Tiempo de Espera a Implementar la Próxima Característica:

3 semanas.

3.2.2.2. Planificar la Minimización de la Complejidad.

Objeto (*)	Atributos (*)	Restricciones
Persona Sistemas	Identificador	Clave principal. Auto incrementable.
	Cédula	Longitud de 10 caracteres. Debe ser única.
	Nombre	
	Apellido	
	Sexo	'Masculino' o 'Femenino'.
	Correo	
	Dirección	

	Teléfono Convencional	Longitud de 7 o 9 caracteres.
	Teléfono Celular	Longitud de 10 caracteres.
Auxiliar	Usuario	Debe ser único.
	Contraseña	
	Estado	
Implemento	Nombre	Debe ser único.
	Cantidad	Mayor que 0
Préstamo	Fecha	
	Hora	
	Tiempo	Mayor que 0
	Cantidad Implemento	Mayor que 0
Devolución	Cantidad del Préstamo	Mayor que 0
	Fecha	
	Hora	
	Estado	'OK' o 'Dañado'
	Observación	

Tabla 3.1. Plantilla de descripción de objetos.

3.2.2.3. Planificar la Anticipación de los Cambios.

1) Identificar elementos que parecen susceptibles de cambiar.

Elemento	Probabilidad
Lógica de negocios de los préstamos del laboratorio.	60%
Características propias del lenguaje de programación.	50%
Diseño difícil y áreas de implementación.	70%
Variables de estado.	60%
Restricciones del tamaño.	80%

Tabla 3.2. Tabla de Elementos Susceptibles de Cambio.

Se omitirán los pasos 2 y 3 debido a que se supondrá que eso será controlado al momento de implementar.

3.2.2.4. Seleccionar los Estándares de Implementación.

Codificación.

- **¿Se han definido convenciones de codificación para los nombres, comentarios y formato?**

  Se propone nombrar las variables usando el estándar Java, los comentarios serán para describir lo que hace el procedimiento y se usarán tabuladores para el facilitar la lectura del código.

- **¿Se han definido las prácticas de codificación específicas que están implicadas en la arquitectura, por ejemplo, cómo las condiciones de**

error que se manejarán, cómo la seguridad se abordará, y así sucesivamente?

Se utilizará el Proceso de Programación por Pseudocódigo [5] y se utilizará el Desarrollo Guiado por Pruebas [10].

- **¿Ha identificado su ubicación en la ola de la tecnología (época tecnológica) y ajustó su enfoque para que coincida? Si es necesario, ¿se ha identificado cómo se va a programar en el lenguaje en lugar de limitarse al programar en él?**

  Como práctica se utilizará un lenguaje de programación que genere sistemas de escritorio debido a que su lenguaje es nativo y se puede expresar lo que se intenta desarrollar.

Trabajo en Equipo.

- **¿Se ha definido un procedimiento de integración, es decir, ha definido los pasos específicos que un programador debe atravesar antes de comprobar el código en las fuentes principales?**

  No se ha definido un procedimiento de integración debido a que el caso de estudio es implementado por una sola persona.

- **¿Los programadores programarán en parejas o individualmente, o alguna combinación de los dos?**

  Se programará individualmente debido a que solo existe una persona desarrollando la aplicación.

Aseguramiento de la Calidad.

- **¿Los programadores escribirán casos de prueba para su código antes de escribir el código en sí?**

  Se escribirán casos de prueba para establecer lo que se espera del código del sistema.

- **¿Los programadores escribirán pruebas unitarias para el código independientemente si ellos escriben primeros o últimos?**

  Se empezará por escribir pruebas de unidad de la capa del negocio del sistema.

- **¿Los programadores darán un paso a través de su código en el depurador antes de que lo registren?**
  Se debe usar el depurador para ver si el código responde a lo previsto.
- **¿Los programadores probarán la integración de su código antes de que la registren?**
  La integración no será utilizada por lo que no habrán pruebas de integración.
- **¿Los programadores revisarán o inspeccionarán el código de los demás?**
  Es deber del desarrollador inspeccionar su código ya que no hay más desarrolladores que lo hagan.

Herramientas de Implementación y Gestión de Versiones.

- **¿Se ha seleccionado una herramienta de control de versiones?**
  Se selecciona el servicio de almacenamiento de archivos Dropbox, ya que también puede controlar hasta las 15 últimas versiones.
- **¿Se ha seleccionado un lenguaje y versión del lenguaje o versión del compilador?**
  Se pretende seleccionar el lenguaje en el proceso mencionado en este capítulo.
- **¿Se ha identificado y adquirido otras herramientas que se van a utilizar (editor, herramienta de refactoreo, un depurador, un marco de prueba, comprobador de sintaxis, etc.)?**
  Debido a que es un caso de estudio es posible que se apliquen algunas de estas herramientas.

3.2.2.5. Realizar el Diseño de la Implementación.

BUSCAR OBJETOS DEL MUNDO REAL.

Se procede a describir todos los aspectos que suceden con los objetos del dominio del problema del caso de estudio antes descrito.

Actividades:

- **Persona Sistemas.** Registrar persona, consultar persona(s), asignar

teléfono convencional, asignar teléfono celular, consultar identificador, consultar nombre, consultar apellido, consultar teléfono convencional, consultar teléfono celular.

- **Auxiliar.** Registrar auxiliar, consultar auxiliar, consultar contraseña, asignar contraseña, consultar estado, asignar estado, verificar auxiliar, verificar contraseña.
- **Implemento.** Registrar implemento, consultar implemento, consultar cantidad, asignar cantidad.
- **Préstamo.** Registrar préstamo, consultar préstamo(s), verificar existencia, consultar identificador, consultar responsable, consultar implemento, consultar cantidad implemento, consultar tiempo.
- **Devolución.** Registrar devolución, consultar devolución(es), asignar observación.

Interacciones entre Objetos:

- Un implemento puede ser solicitado por varias personas de sistemas.
- Una persona de sistemas puede solicitar varios implementos.
- Al solicitar un préstamo se disminuye la cantidad del implemento dependiendo de la cantidad que se ha solicitado.
- Una persona de sistemas puede devolver varios implementos.
- Un implemento puede ser devuelto por varias personas de sistemas.
- Al eliminar un préstamo se registra la devolución.
- Al eliminar el préstamo se suma al inventario los implementos que se han devuelto si están en buen estado.

Partes Visibles:

- Para cualquier préstamo será visible el nombre, cantidad y el tiempo del implemento.
- Para cualquier préstamo será visible el nombre y apellido de la persona sistemas.
- Para cualquier devolución será visible el nombre y apellido de la persona sistemas.

- Para cualquier devolución será visible la cantidad, tiempo, fecha y hora del préstamo.
- Para cualquier devolución será visible el nombre del implemento.
- Para cualquier auxiliar será visible el nombre y apellido de la persona sistemas (correspondiente).
- Para cualquier persona sistemas (que sea auxiliar) será visible el usuario y estado del auxiliar.

Interfaces y otras Relaciones:

- La clase Auxiliar es heredada por la clase Persona Sistemas.

Nota:

Se tuvo especificar solo los auxiliares de las personas sistemas para que se visualicen su usuario y contraseña.

DEFINIR EL DISEÑO DEL SISTEMA.

Se describe el diseño basado en el caso de estudio y por el cual se basará su respectiva implementación.

Nivel 1. Sistema Software.

El sistema será nativo, será implementado en una computadora de escritorio. Su diseño se encuentra en el Anexo B, Sistema Software.

Nivel 2. División en Subsistemas o Paquetes.

- **Lógica de Negocio.**

Procedimientos para realizar los préstamos en los laboratorios del DICC. Su diseño se encuentra en el Anexo B, Lógica de Negocio.

- **Interfaz de Usuario.**

Se implementará un sistema nativo de escritorio que se ejecutará sobre el Sistema Operativo Windows XP. Su diseño se encuentra en el Anexo B, Interfaces de Usuario.

- **Acceso a Base de Datos.**

Se utilizará el motor de base de datos SQL Server, se ha basado el diseño de la base de datos en el diagrama de clases, algunos de los métodos serán utilizados como procedimientos almacenados y su conectividad será por medio de comandos SQL y LINQ. Su diseño se encuentra en el Anexo B, Base de Datos.

- **Dependencias del Sistema.**

Funciones que proporciona el sistema operativo Windows XP.

Nivel 3. División en Clases.

Su diseño se encuentra en el Anexo B, Diagramas de Clases.

3.2.2.6. Seleccionar los Lenguajes de Implementación.

Dado a que se elegirá un programa seguro y se manipulará una base de datos se escogerá C# y SQL por manipular datos por medio de una operación.

3.2.2.7. Seleccionar las Consideraciones para Codificar.

IDENTIFICACIÓN DE PATRONES DE DISEÑO EN EL SISTEMA.

Se identifica mediante los diagramas de clases (ver Anexo B) que se ha diseñado el patrón Fachada debido a que se lo usa tanto en la clase Auxiliar como en la clase PersonaSistemas.

SELECCIÓN DE HERRAMIENTAS DE DESARROLLO.

- **¿Se tiene un ambiente de desarrollo integrado efectivo?**

Visual Studio 2008.

- **¿El IDE soporta lo siguiente?**
  - Vista esquemática del programa. *Si*
  - Salto a definiciones de clases, rutinas y variables. *Si*
  - Formato del código fuente. *Si*
  - Juego de llaves (de comienzo y fin). *Si*
  - Búsqueda y reemplazo de cadenas de archivos múltiple. *Si*
  - Compilación conveniente. *Si*
  - Depuración integrada. *Si*

- **¿Se tienen herramientas que automaticen los refactoreos comunes?**

Si, debido a que las funciones de refactoreo se encuentran dentro del IDE.

- **¿Se está usando control de versiones para gestionar lo siguiente?**
  - Código fuente. *Si*
  - Contenido. *Si*
  - Requerimientos. *Si*
  - Diseños. *Si*
  - Planes de Proyecto. *Si*
  - Otros artefactos del proyecto (especificar). *No*

- **¿Si se está trabajando en un proyecto muy grande, se está usando un diccionario de datos o algún otro repositorio central que contenga descripciones acreditadas de cada clase usada en el sistema?**

No se está trabajando en un proyecto grande, por lo que no se hace prescindible un diccionario de datos.

- **¿Se han considerado bibliotecas de código como alternativas para escribir código personalizado si está disponible?**

Se desarrollarán bibliotecas de código para reducir código repetido.

- **¿Se está haciendo uso de un depurador interactivo?**

Si, en el IDE está integrado ello.

- **¿Se utiliza 'make' u otro software de control de dependencias para desarrollar programas eficientemente y confiablemente?**

No.

- **¿El ambiente de pruebas incluye las siguientes cosas?**
  - Marco de trabajo automatizado de pruebas. *Si*
  - Generadores de pruebas automatizados. *No*
  - Monitores de cobertura. *No*
  - Perturbadores de sistema. *No*

- Herramientas diferenciales. *No*
- Software de detección de defectos. *Si*

- **¿Se ha creado alguna herramienta personalizada que podría ayudar a dar soporte a las necesidades específicas del proyecto, especialmente herramientas que automaticen tareas repetitivas?**

El IDE ayuda con generación de código con el fin de evitar implementar código innecesario para solo enfocarse en la lógica de negocio.

- **¿En general, el entorno se beneficia del soporte de la herramienta adecuada?**

Si.

3.2.2.8. Preparar las Pruebas Unitarias.

IDENTIFICAR EL DISEÑO GENERAL DE LA CLASE [5].

Se procede a identificar el diseño general de todas las clases del sistema. Como ejemplo solo se hará con 2 clases del caso de estudio.

1)

Clase: PersonaSistemas.

Responsabilidades Específicas.

- Esta clase se encarga de almacenar un registro de una persona de la facultad, sea quien fuere.
- Con esta clase se puede relacionar a cualquier persona de sistemas con el sistema mismo cómo usuario y cómo persona que participa del préstamo de unos de los implementos del laboratorio.

Secretos.

- Atributos de la clase.

Abstracción Capturada por la Clase.

Cédula, nombre, apellido, sexo, correo, dirección, telefonoConvencional y telefonoCelular.

Clase Principal:
Ninguna.

Clases Herederas:
Auxiliar.

Métodos Públicos Clave:

- PersonaSistemas.
- consultarId.
- consultarNombre.
- consultarApellido.
- registrarPersonaSistemas.

2)

Clase: Auxiliar.

Responsabilidades Específicas.

- Esta clase se encarga de almacenar a los auxiliares que usan el sistema.
- Consulta los auxiliares con el fin de autenticarlos o de modificar su contraseña.
- Cuando un auxiliar deja de ejercer sus funciones, esta clase inactiva el usuario que lo representa.

Secretos.

- Los atributos de la clase.
- La manera en la que la clase almacena los datos en el sistema.
- La manera en que el auxiliar relaciona con sus datos de su clase que la hereda.
- La manera en la que desactiva a los auxiliares.

- La manera en la que se cambia la contraseña de un auxiliar.

Abstracción Capturada por la Clase.

El usuario, la contraseña y el estado.

Clase Principal:

PersonaSistemas.

Clases Herederas:

Ninguna.

Métodos Públicos Clave:

- Auxiliar.
- verificarAuxiliar.
- verificarContraseña.
- registrarAuxiliar.

DISEÑAR EL PSEUDOCÓDIGO DE CADA RUTINA [5].

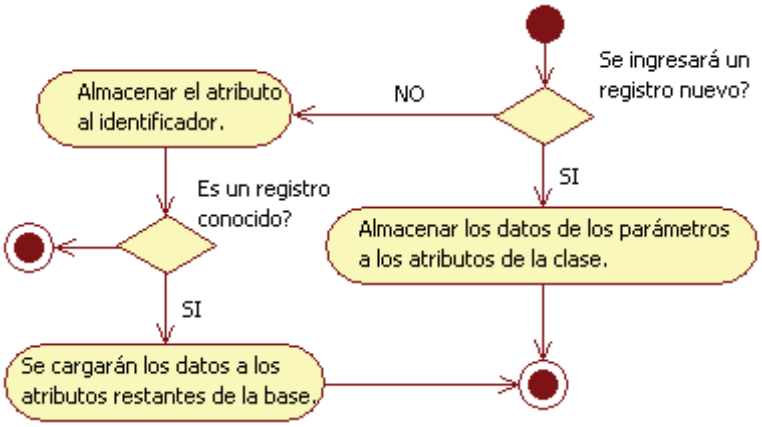
Se procede a diseñar el pseudocódigo de todas las rutinas de todas las clases del sistema. Como ejemplo solo se hará con 2 rutinas del caso de estudio.

1)

Nombre de la Rutina: PersonaSistemas.

Clase: PersonaSistemas.

Diseño:



Prerrequisitos:

- En esta rutina se recogen todos los detalles obligatorios de un registro de la persona de sistemas.

Problema a Resolver:

a) Detalles.

- Recoge los datos obligatorios de una persona sistemas para una inserción.
- Carga el registro a la clase desde una consulta a base de datos.

b) Datos.

- La rutina esconderá el procedimiento para recoger los datos de la persona de sistemas.
- Se tendrá como entradas: el identificador, la cédula, el nombre, el apellido, el sexo, el correo y la dirección.
- No tendrá valor de retorno como salida.
- Las precondiciones son: la cédula es correctamente ingresada y tiene 10 caracteres, el sexo solo es ingresado como ‘Masculino’ o ‘Femenino’ y ningún campo es nulo.
- Habrá como garantía que la clase almacene los datos principales de una persona de sistemas.

Manejo de Errores.

Debe asegurarse que se cumplan las precondiciones descritas para evitar errores al momento de almacenar los resultados al sistema.

Posibilidad de Eficiencia.

N/A.

Funcionalidad Disponible en Bibliotecas Estándares.

La funcionalidad de LINQ para cuando se realice la consulta del registro.

Algoritmos Disponibles:

Se ha investigado de rutinas constructoras en algunos ejemplos de instanciación de objetos.

Pseudocódigo:

```
En esta rutina se recogen todos los detalles obligatorios de un
registro de la persona de sistemas.
Inicio PersonaSistemas (cedula, nombre, apellido, sexo, correo,
dirección)
   Asignar todos los valores de los parámetros.
Fin
```

```
En esta rutina se recoge el identificador para manipular los datos la
persona de sistemas.
Inicio PersonaSistemas (identificador)
   Asignar el valor del parámetro al atributo de la clase.
   Si se ha asignado un identificador de un dato conocido, entonces
     Se cargarán los datos a los atributos restantes de la base.
   Fin Si.
Fin
```

Datos a Manejarse:

- 2 enteros.
- 2 arreglos de caracteres.
- 8 cadenas de texto.
- 2 enumeraciones.

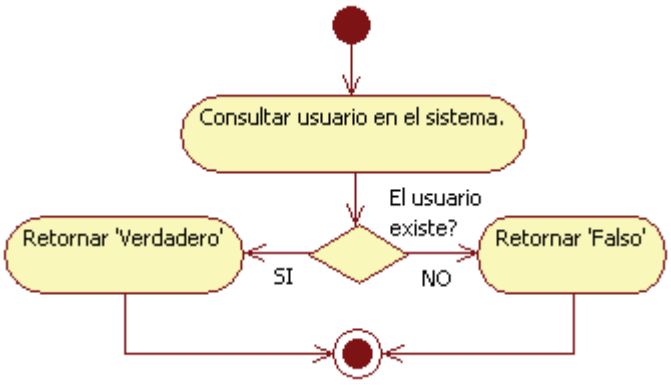
En este caso, se ha diseñado 2 rutinas con el mismo nombre (polimorfismo de la rutina) con el fin de controlar el uso de la clase cuando se ingresa un registro nuevo y cuando se manipula un registro conocido.

2)

Nombre de la Rutina: verificarAuxiliar.

Clase: Auxiliar.

Diseño:



Prerrequisitos:

- Se verifica que el usuario exista por lo que necesitará conexión con la base de datos para lograrlo.

Problema a Resolver:

a) Detalles.

- Verificará si el usuario existe.
- Emitirá una señal afirmativa si el usuario en verdad existe, de lo contrario emitirá una señal negativa.

b) Datos.

- La rutina esconderá el procedimiento para hallar su existencia en el sistema.
- No se tendrá parámetros de entrada.
- Se tendrá un valor de retorno booleano como salida.
- No existen precondiciones.
- El valor de retorno será garantía de la rutina que indicará el éxito o el fracaso de la rutina en hallar al usuario.

Manejo de Errores.

No existe probabilidad de error en esta rutina.

Posibilidad de Eficiencia.

Se realizará la búsqueda del nombre del usuario mediante una consulta a la base de datos, cosa que no se necesitará por el momento un algoritmo optimizado.

Funcionalidad Disponible en Bibliotecas Estándares.

Se utilizará una forma de conexión a base de datos llamada LINQ, con el fin de realizar la consulta a la base sin preocuparse por la conectividad a la base de datos.

Algoritmos Disponibles:

Cómo si es posible utilizar LINQ, entonces no será prescindible investigar de algoritmos.

Pseudocódigo:

```
Verificará si el usuario existe y emite una señal afirmativa si el
usuario en verdad existe, de lo contrario emite una señal negativa.
Inicio verificarAuxiliar ()
   Consultar en la base de datos el nombre del usuario.
   Si el usuario existe entonces,
     Retornar 'Verdadero'
   De lo contrario
     Retornar 'Falso'
   Fin Si
Fin
```

Datos a Manejarse:

- Un entero.
- Una cadena de texto.

DISEÑAR LOS CASOS DE PRUEBA DE CADA RUTINA.

Se procede a diseñar los casos de prueba de cada rutina, en este caso se realizará los casos de prueba de 2 rutinas del caso de estudio.

Rutina: verificarAuxiliar.

# de Caso	Descripción	Datos de Entrada	Resultado Esperado
1	Se verifica si el valor es positivo cuando se halla el usuario.	Una cadena de un usuario existente.	Verdadero.
2	Se verifica si el valor es negativo cuando no se halla usuario.	Una cadena de un usuario inexistente.	Falso.

Tabla 3.3. Tabla de Pruebas Unitarias.

Rutina: verificarContraseña.

# de Caso	Descripción	Datos de Entrada	Resultado Esperado
1	Se verifica si el valor es positivo cuando se ingresa la contraseña correcta.	Una cadena de un usuario existente. Una cadena de su contraseña correcta.	Verdadero.
2	Se verifica si el valor es negativo cuando se ingresa la contraseña incorrecta.	Una cadena de un usuario existente. Una cadena de su contraseña incorrecta.	Falso.
3	Se verifica que lance una excepción si se trata de un usuario inexistente.	Una cadena de un usuario inexistente.	Excepción.

Tabla 3.4. Tabla de Pruebas Unitarias.

SELECCIONAR EL MARCO DE TRABAJO DE PRUEBAS.

El marco de trabajo de pruebas disponible [8] es NUnit para el caso de estudio en el que se ha elegido C#.

3.2.2.9. Preparar las Pruebas de Integración.

La integración se usa si se desarrolla en varios módulos pero debido a que se hace uno solo, no se requiere la integración.

3.2.2.10. Considerar el Reuso.

Se procede a considerar el reuso en este caso de estudio por lo se responde de esta manera:

Consideración del Reuso.
Si.
Justificación.
Debido a que se ha desarrollado un sistema similar a este caso de estudio antes, se reusarán sus partes para adaptarlas a este caso de estudio.

3.2.2.10.1. Selección de Unidades Reutilizables.

Se procede a seleccionar el nivel de reuso y técnicas.

Nivel de Reuso:

Reutilización de Objetos y Funciones.

Técnicas de Reuso:

- Integración COTS.
- Librerías de programas.
- Generadores de programas.

3.2.2.10.2. Verificación de la Posibilidad de Reuso.

Se procede a analizar la posibilidad de reuso en la implementación del caso de estudio.

Agenda de Desarrollo de Software:

Agenda disponible.

Vida esperada del software:

Sistema a largo plazo, sus implicaciones no perjudicaran al sistema ya que las partes reusadas serán adaptadas al caso de estudio y serán documentadas.

Conocimientos, habilidades y experiencia del grupo de desarrollo:

Las partes reusables provienen de un sistema que ha sido realizado por la persona quien está aplicando el caso de estudio.

Criticidad del software.

Nominal.

Dominio del sistema.

N/A.

Plataforma sobre la cual se ejecuta el sistema.

Windows XP y C# que pertenece a la plataforma de Windows.

Conclusión.
Se procederá con el reuso debido a que ahorrará esfuerzos en la implementación del caso de estudio.

3.2.2.10.3. Comunicar Información de las Partes en Reutilización.

Ahora se elegirán los mecanismos pertinentes para informar el reuso.

Mecanismo:
Retroalimentación.

Justificación:
Solo existe una persona que implementará el sistema y solo documentará lo que suceda en la implementación del sistema.

Implementación:
Mediante las plantillas usadas en este caso de estudio.

3.2.3. SELECCIONAR METRICAS Y SUS TECNICAS.

Se procede a elegir las métricas y las técnicas que se aplicarán a este caso de estudio.

3.2.3.1. Seleccionar Métricas de Implementación.

Se elegirán 2 métricas para el proyecto:

- Total de líneas de código escritas.
- Total de número de clases y rutinas.

3.2.3.2. Seleccionar Técnicas de Medición.

Se elegirá la siguiente técnica:

- Estimar las piezas del proyecto, y luego añadirlas.

De tal forma que quedaría de esta forma la selección:

Métricas de Implementación:

- Total de líneas de código escritas.
- Control de flujo de complejidad en cada rutina.
- Líneas de código en cada clase o rutina.

Técnicas de Medición:

- Estimar las piezas del proyecto, y luego añadirlas.

3.2.4. IMPLEMENTACION Y PRUEBAS UNITARIAS.

Se procede a implementar todas las clases del sistema, es decir implementar toda su estructura incluyendo sus rutinas. Para ejemplo se implementará una clase, el resto se adjuntará como anexo el código fuente del caso de estudio.

Clase: Implemento.

1) Revisar el diseño general para la clase y armarla.

Después de la revisión de su diseño general y en el diagrama de clases, se la arma bajo el lenguaje elegido (C#), con sus atributos y los cuerpos de cada rutina.

```
using System;
using System.Collections.Generic;
using System.Text;

namespace Sistema_de_Préstamos.Datos
{
    class Implemento
    {
        private string nombre;
        private int cantidad;

        public Implemento(string nombre, int cantidad)
        {
            throw new System.NotImplementedException();
        }

        public Implemento(int id)
        {
            throw new System.NotImplementedException();
        }

        public int consultarCantidad()
        {
            throw new System.NotImplementedException();
        }

        public void asignarCantidad(int cantidad)
        {
            throw new System.NotImplementedException();
```

```
        }

        public void registrarImplemento()
        {
            throw new System.NotImplementedException();
        }
    }
}
```

Nota: En este caso no se ha tomado en cuenta las librerías usadas ni el espacio de nombres.

2) Construir las rutinas dentro de la clase.

Se procede a construir cada rutina dentro de la clase:

Rutina: Implemento.

a. Escribir la declaración de la rutina.

```
public Implemento(string nombre, int cantidad)
{
   throw new System.NotImplementedException();
}

public Implemento(int id)
{
        throw new System.NotImplementedException();
}
```

b. Convertir el pseudocódigo en comentarios de alto nivel.

Se usará el pseudocódigo proveniente del Anexo C.

```
//Recoge los datos obligatorios de un implemento para una inserción.
public Implemento(string nombre, int cantidad)
{
   //Verificar si se ingresó el implemento antes.

   //Si se ha ingresado antes, entonces
      //Lanzar una excepción.
      //Terminar la rutina.
   //De lo contrario
      //Almacenar los datos de los parámetros a los atributos.
   //Fin Si
}

// Carga el registro de un implemento a la clase desde una consulta a
//base de datos.
public Implemento(int id)
{
   //Cargar de la base los datos del implemento a los atributos.
}
```

c. Añadir una prueba unitaria en función del caso de prueba que se realice.

En este caso, se toma el primer caso, construyendo una clase que sostenga todas las pruebas unitarias de tal rutina y la prueba unitaria, que es una rutina que no retorna nada y que en ella se verifica el valor de verdad.

# de Caso	Descripción	Datos de Entrada	Resultado Esperado
1	Se verifica que lance una excepción si se ha ingresado antes el registro.	Nombre de un implemento existente. Cualquier cantidad.	Excepción.

Tabla 3.5. Primera Prueba Unitaria de la Rutina 'Implemento'.

```
[TestFixture]
public class PruebaImplemento
{
   Datos.Implemento implementoConocido;

    [Test]
    [ExpectedException(typeof(ApplicationException))]
   //Se verifica que lance una excepción si se ha ingresado antes el
   //registro.
   public void insertarImplementoConocido()
   {
      implementoConocido = new Datos.Implemento("Esfero Bic Verde",
         2);
   }
}
```

d. Rellenar cada comentario correspondiente con código correspondiente a la prueba.

Se rellena de código los comentarios de la rutina Implemento respecto a la primera prueba.

```
public Implemento(string nombre, int cantidad)
{
   //Verificar si se ingresó el implemento antes.
   int existenciaImplemento;
   Persistencia.LINQ_PréstamosDataContext
      consultaImplementoConocido = new
      Persistencia.LINQ_PréstamosDataContext();
   existenciaImplemento = (from tablaImplemento in
         consultaImplementoConocido.Implemento
      where tablaImplemento.nombre == nombre
      select tablaImplemento).Count<Persistencia.Implemento>();

   if (existenciaImplemento > 0) //Si se ha ingresado antes, entonces
   {
      //Lanzar una excepción.
      //Terminar la rutina.
      throw new ApplicationException();
```

```
    }
    else //De lo contrario
    {
        //Almacenar los datos de los parámetros a los atributos.
    }//Fin Si
}
```

e. Ejecutar todas las pruebas (incluyendo la actual).

Se ejecutará solamente la que ya se había creado.

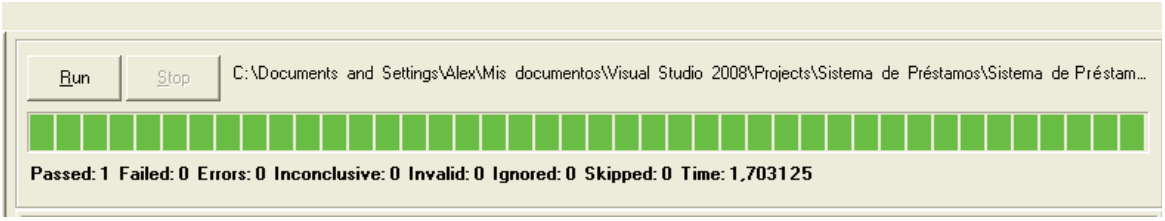


Figura 3.1. Primera Ejecución de la Primera Prueba Unitaria de la Rutina 'Implemento'.

f. Refactorar el código.

Debido a que el dato previamente si existía en la base de datos, entonces se mejorará la prueba para que lo inserte sin preocuparse por insertarlo manualmente.

```
[Test]
[ExpectedException(typeof(ApplicationException))]
//Se verifica que lance una excepción si se ha ingresado antes el
//registro.
public void insertarImplementoConocido()
{
    Persistencia.Implemento implementoPrueba;

    //Verificar si el implemento de prueba aún existe.
    Persistencia.LINQ_PréstamosDataContext insercionImplementoPrueba =
        new Persistencia.LINQ_PréstamosDataContext();
    existenciaImplemento = (from tablaImplemento
            in insercionImplementoPrueba.Implemento
        where tablaImplemento.nombre == "Esfero Bic Verde"
    select tablaImplemento).Count<Persistencia.Implemento>();

    if (existenciaImplemento == 0) //Si el implemento de prueba no
    //existe, entonces
    {
        //Insertar el implemento conocido.
        implementoPrueba = new Persistencia.Implemento();
        implementoPrueba.nombre = "Esfero Bic Verde";
        implementoPrueba.cantidad = 2;
        insercionImplementoPrueba.Implemento.
            InsertOnSubmit(implementoPrueba);
        insercionImplementoPrueba.SubmitChanges();
    }//Fin si.
```

```
    //Intentar ingresar el implemento.
    implementoConocido = new Datos.Implemento("Esfero Bic Verde", 2);
}
```

Esta prueba al igual que las demás pruebas ha sido hecha con PPP.

g. Añadir la segunda prueba unitaria.

# de Caso	Descripción	Datos de Entrada	Resultado Esperado
2	Se verifica que se ingrese los datos a la clase.	Nombre de un implemento inexistente. Cualquier cantidad.	No debe existir el registro en la base.

Tabla 3.6. Segunda Prueba Unitaria de la Rutina 'Implemento'.

```
[Test]
//Se verifica que se ingrese los datos a la clase.
public void insertarNuevoImplemento()
{
    //Verificar la existencia del implemento de prueba.
    Persistencia.LINQ_PréstamosDataContext consultaImplementoPrueba =
        new Persistencia.LINQ_PréstamosDataContext();
    int existenciaImplemento = (from tablaImplemento in
            consultaImplementoPrueba.Implemento
        where tablaImplemento.nombre == "Esfero Bic Azúl"
        select tablaImplemento).Count<Persistencia.Implemento>();

    if (existenciaImplemento > 0) //Si existe el implemento de prueba,
    //entonces
    {
        //Eliminarlo.
        Persistencia.Implemento implementoPrueba = (from tablaImplemento
                in consultaImplementoPrueba.Implemento
            where tablaImplemento.nombre == "Esfero Bic Azúl"
            select tablaImplemento).Single();
            consultaImplementoPrueba.Implemento.
                DeleteOnSubmit(implementoPrueba);
            consultaImplementoPrueba.SubmitChanges();
    }//Fin Si.

    //Ingresar el implemento de prueba.
    implementoNuevo = new Implemento("Esfero Bic Azúl", 4);

    //Consultarlo.
    Assert.AreEqual(4, implementoNuevo.consultarCantidad());
}
```

h. Rellenar cada comentario correspondiente con código correspondiente a la prueba.

```
public Implemento(string nombre, int cantidad)
{
    //Verificar si se ingresó el implemento antes.
    int existenciaImplemento;
```

```
    Persistencia.LINQ_PréstamosDataContext consultaImplementoConocido =
       new Persistencia.LINQ_PréstamosDataContext();
    existenciaImplemento = (from tablaImplemento
          in consultaImplementoConocido.Implemento
       where tablaImplemento.nombre == nombre
       select tablaImplemento).Count<Persistencia.Implemento>();

    if (existenciaImplemento > 0) //Si se ha ingresado antes, entonces
    {
       //Lanzar una excepción.
       //Terminar la rutina.
       throw new ApplicationException();
    }
    else //De lo contrario
    {
       //Almacenar los datos de los parámetros a los atributos.
       this.nombre = nombre;
       this.cantidad = cantidad;
    }//Fin Si
}
```

Lo único que se rellena del código es en la línea en la que se almacenan los datos de los parámetros a los atributos.

```
public int consultarCantidad()
{
    //Retornar la cantidad.
    return cantidad;
}
```

i. Ejecutar todas las pruebas (incluyendo la actual).

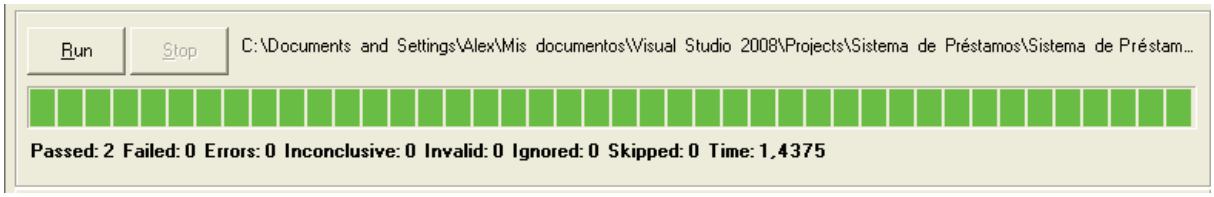


Figura 3.2. Segunda Ejecución de las Pruebas Unitarias de la Rutina 'Implemento'.

j. Refactorar el código.

Se realiza una ligera mejora en la clase que sostiene en las pruebas añadiendo un atributo en el que usa para contar los registros de la existencia de un préstamo (no se mostrará el código de la clase).

k. Añadir la tercera prueba unitaria.

# de Caso	Descripción	Datos de Entrada	Resultado Esperado
3	Se verifica que se	Identificador de un	El registro

	carguen los datos de un cierto registro a la clase.	registro existente.	almacenado en la clase debió haberse almacenado en la clase.

Tabla 3.7. Tercera Prueba Unitaria de la Rutina 'Implemento'.

```
[Test]
//Se verifica que se carguen los datos de un cierto registro a la clase.
public void cargarImplemento()
{
   int identificador, cantidadImplemento;
   Persistencia.Implemento implementoPrueba = new
      Persistencia.Implemento();

   //Verificar la existencia del implemento de prueba.
   Persistencia.LINQ_PréstamosDataContext consultaImplementoPrueba =
      new Persistencia.LINQ_PréstamosDataContext();
   existenciaImplemento = (from tablaImplemento
         in consultaImplementoPrueba.Implemento
      where tablaImplemento.nombre == "Esfero Bic Negro"
      select tablaImplemento).Count<Persistencia.Implemento>();

   if (existenciaImplemento == 0) //Sino existe el implemento de prueba,
   //entonces
   {
      //Crearlo.
      implementoPrueba.nombre = "Esfero Bic Negro";
      implementoPrueba.cantidad = 7;
      consultaImplementoPrueba.Implemento.
         InsertOnSubmit(implementoPrueba);
      consultaImplementoPrueba.SubmitChanges();
   } //Fin Si.

   //Recuperar el identificador y la cantidad del implemento de prueba.
   implementoPrueba = (from tablaImplemento
         in consultaImplementoPrueba.Implemento
      where tablaImplemento.nombre == "Esfero Bic Negro"
      select tablaImplemento).Single();
   identificador = implementoPrueba.identificador;
   cantidadImplemento = implementoPrueba.cantidad;

   //Cargar el implemento a la clase.
   implementoConocido = new Implemento(identificador);

   //Consultarlo.
   Assert.AreEqual(implementoConocido.consultarCantidad(),
      cantidadImplemento);
}
```

I. Rellenar cada comentario correspondiente con código correspondiente a la prueba.

```
//Carga el registro de un implemento a la clase desde una consulta a
//base de datos.
public Implemento(int id)
{
```

```
    //Cargar de la base los datos del implemento a los atributos.
    Persistencia.LINQ_PréstamosDataContext consultaImplemento = new
       Persistencia.LINQ_PréstamosDataContext();
    Persistencia.Implemento implemento = (from tablaImplemento
          in consultaImplemento.Implemento
       where tablaImplemento.identificador == id
       select tablaImplemento).Single();
    this.nombre = implemento.nombre;
    this.cantidad = implemento.cantidad;
}
```

m. Ejecutar todas las pruebas (incluyendo la actual).

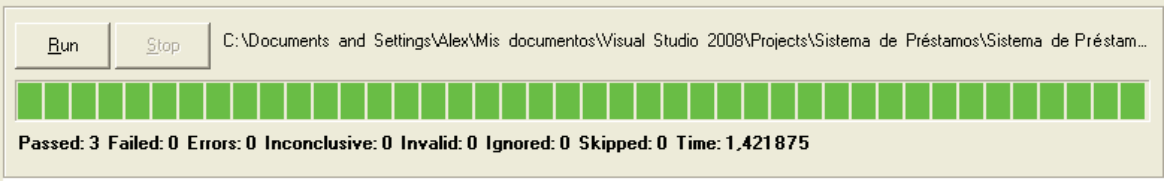


Figura 3.3. Tercera Ejecución de las Pruebas Unitarias de la Rutina 'Implemento'.

n. Refactorar el código.

Se añadirá a la prueba una instrucción para eliminar lo que ha consultado con el fin de limpiar la base de datos de los datos de prueba y se añadirán variables en lugar de usar un número y una cadena, y se reducirá código.

```
[Test]
//Se verifica que se carguen los datos de un cierto registro a la clase.
public void cargarImplemento()
{
   int identificador, cantidadImplemento = 7;
   string nombreImplemento = "Esfero Bic Negro";
   Persistencia.Implemento implementoPrueba = new
      Persistencia.Implemento();

   //Verificar la existencia del implemento de prueba.
   Persistencia.LINQ_PréstamosDataContext consultaImplementoPrueba = new
      Persistencia.LINQ_PréstamosDataContext();
   existenciaImplemento = (from tablaImplemento
         in consultaImplementoPrueba.Implemento
      where tablaImplemento.nombre == "Esfero Bic Negro"
      select tablaImplemento).Count<Persistencia.Implemento>();

   if (existenciaImplemento == 0) //Sino existe el implemento de prueba,
   //entonces
   {
      //Crearlo.
      implementoPrueba.nombre = nombreImplemento;
      implementoPrueba.cantidad = cantidadImplemento;
      consultaImplementoPrueba.Implemento.
         InsertOnSubmit(implementoPrueba);
      consultaImplementoPrueba.SubmitChanges();
   } //Fin Si.
```

```
    //Recuperar el identificador del implemento de prueba.
    implementoPrueba = (from tablaImplemento
          in consultaImplementoPrueba.Implemento
       where tablaImplemento.nombre == "Esfero Bic Negro"
       select tablaImplemento).Single();
    identificador = implementoPrueba.identificador;

    //Cargar el implemento a la clase.
    implementoConocido = new Implemento(identificador);

    //Consultarlo.
    Assert.AreEqual(implementoConocido.consultarCantidad(),
       cantidadImplemento);

    //Eliminar el implemento de prueba.
    consultaImplementoPrueba.Implemento.DeleteOnSubmit(implementoPrueba);
    consultaImplementoPrueba.SubmitChanges();
}
```

De esta forma se ha implementado una clase en la que sostiene todas las pruebas unitarias de la rutina y estas dieron forma a la rutina del sistema a implementar.

Rutina: asignarCantidad.

a. Escribir la declaración de la rutina.

```
public void asignarCantidad(int cantidad)
{
    throw new System.NotImplementedException();
}
```

b. Convertir el pseudocódigo en comentarios de alto nivel.

```
//Asigna la cantidad del implemento.
public void asignarCantidad(int cantidad)
{
    //Modificar la cantidad del implemento.
}
```

c. Añadir una prueba unitaria en función del caso de prueba que se realice.

# de Caso	Descripción	Datos de Entrada	Resultado Esperado
1	Se verifica que se haya ingresado la cantidad esperada.	Una cantidad cualquiera.	Cantidad incrementada en una unidad.

Tabla 3.8. Prueba Unitaria de la Rutina 'asignarCantidad'.

Se construirá una rutina que sostenga esta prueba pero que también generará el implemento de prueba en la base de datos con el fin de que se efectúe tal prueba (realizada con PPP) y también se añadieron rutinas auxiliares para sostener la prueba principal.

```
[TestFixture]
public class PruebaAsignarCantidad
{
    Implemento implemento;
    const int CANTIDAD = 2;
    const string IMPLEMENTO = "Esfero Bic Verde";
    Persistencia.LINQ_PréstamosDataContext consultaTransaccionImplemento
        = new Persistencia.LINQ_PréstamosDataContext();

    [SetUp]
    public void iniciar()
    {
        if (verificarImplemento()) //Si existe el implemento de prueba,
        //entonces
            implemento = new Implemento(cargarImplemento()); //Se lo
            //carga.
        else //De lo contrario
        {
            //Se lo inserta.
            insertarImplemento();
            //Se lo carga.
            implemento = new Implemento(cargarImplemento());
        }
        //Fin si
    }

    [Test]
    //Se verifica que se haya ingresado la cantidad esperada.
    public void modificarCantidad()
    {
        //Incrementar la cantidad del implemento en 1.
        int cantidadAnterior;
        cantidadAnterior = implemento.consultarCantidad();
        implemento.asignarCantidad(implemento.consultarCantidad() + 1);

        //Verificar el cambio.
        Assert.AreEqual(cantidadAnterior + 1,
            implemento.consultarCantidad());
    }

    bool verificarImplemento()
    {
        //Verificar la existencia del implemento
        int cantidadImplemento = (from tablaImplemento
                in consultaTransaccionImplemento.Implemento
            where tablaImplemento.nombre == IMPLEMENTO
            select tablaImplemento).Count<Persistencia.Implemento>();

        if (cantidadImplemento > 0) //Si el implemento existe, entonces
            return true; //Retornar Verdadero.
        else //De lo contrario
            return false; //Retornar Falso.
        //Fin Si.
    }


    void insertarImplemento()
    {
        //Insertar el implemento y su cantidad.
        Persistencia.Implemento implementoNuevo = new
```

```
            Persistencia.Implemento();
        implementoNuevo.nombre = IMPLEMENTO;
        implementoNuevo.cantidad = CANTIDAD;
        consultaTransaccionImplemento.Implemento.
            InsertOnSubmit(implementoNuevo);
        consultaTransaccionImplemento.SubmitChanges();
    }

    int cargarImplemento()
    {
        //Recuperar el identificador del implemento.
        int identificadorImplemento = (from tablaImplemento
                in consultaTransaccionImplemento.Implemento
            where tablaImplemento.nombre == IMPLEMENTO
            select tablaImplemento.identificador).Single();

        //Retornar el identificador del implemento.
        return identificadorImplemento;
    }
}
```

d. Rellenar cada comentario correspondiente con código correspondiente a la prueba.

```
//Asigna la cantidad del implemento.
public void asignarCantidad(int cantidad)
{
   //Modificar la cantidad del implemento.
   this.cantidad = cantidad;
}
```

e. Ejecutar de nuevo la prueba.

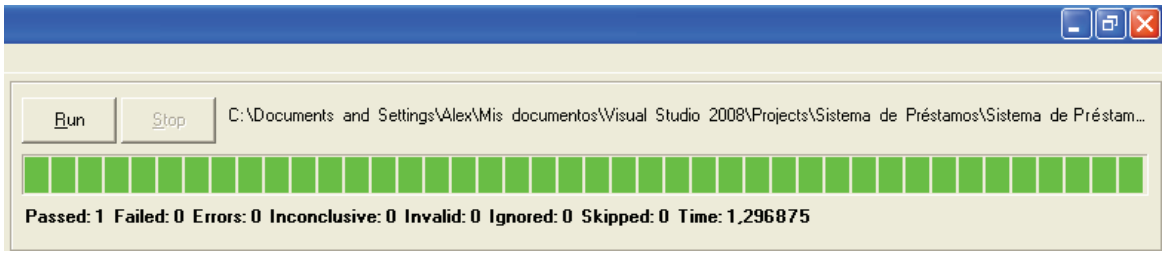


Figura 3.4. Primera Ejecución de la Prueba Unitaria de la Rutina 'asignarCantidad'.

f. Refactorar el código.

```
[Test]
//Se verifica que se haya ingresado la cantidad esperada.
public void modificarCantidad()
{
   //Incrementar la cantidad del implemento en 1.
   int cantidadAnterior;
   cantidadAnterior = implemento.consultarCantidad();
   implemento.asignarCantidad(implemento.consultarCantidad() + 1);
```

```
    //Verificar el cambio.
    Assert.AreEqual(cantidadAnterior + 1,
        implemento.consultarCantidad());

    //Eliminar el implemento de prueba.
    eliminarImplemento();
}
```

```
void eliminarImplemento()
{
    //Buscar el implemento.
    Persistencia.Implemento implemento = (from tablaImplemento
          in consultaTransaccionImplemento.Implemento
       where tablaImplemento.nombre == IMPLEMENTO
       select tablaImplemento).Single();

    //Eliminarlo.
    consultaTransaccionImplemento.Implemento.DeleteOnSubmit(implemento);
    consultaTransaccionImplemento.SubmitChanges();
}
```

En la prueba se ha agregado la eliminación del implemento para limpiar la base de datos de los registros de prueba.

Rutina: registrarImplemento.

a. Escribir la declaración de la rutina.

```
public void registrarImplemento()
{
    throw new System.NotImplementedException();
}
```

b. Convertir el pseudocódigo en comentarios de alto nivel.

```
//Se encarga de almacenar en la base de datos el registro del
Implemento.
public void registrarImplemento()
{
    //Si la cantidad es positiva y se ha ingresado el implemento,
       entonces
       //Registrar los datos a la base.
    //De lo contrario
       //Lanzar una excepción.
       //Terminar la rutina.
    //Fin Si
}
```

c. Añadir una prueba unitaria en función del caso de prueba que se realice.

Se tomará la primera prueba.

# de Caso	Descripción	Datos de Entrada	Resultado Esperado
1	Se verifica que se haya ingresado el	Implemento ingresado.	Registro insertado en la base.

	implemento.	Cantidad positiva.	

Tabla 3.9. Primera Prueba Unitaria de la Rutina 'registrarImplemento'.

Una vez más se creará una clase en la sostendrá todas las pruebas e incluso tendrá rutinas auxiliares que sostendrán a las pruebas unitarias.

```
[TestFixture]
public class PruebaRegistrarImplemento
{
    int cantidadInicialImplementos;
    Datos.Implemento implementoNuevo;
    Datos.Implemento implementoSinNombre;
    Datos.Implemento implementoCantidadNula;
    Persistencia.LINQ_PréstamosDataContext consultaTransaccionImplemento
        = new Persistencia.LINQ_PréstamosDataContext();

    [SetUp]
    public void iniciar()
    {
    //Limpiar la base de anteriores implementos de prueba.
    if (verificarImplemento("Esfero Bic Rojo"))
        eliminarRegistro("Esfero Bic Rojo"); ;
    if (verificarImplemento(""))
        eliminarRegistro("");
    if (verificarImplemento("Esfero Bic Azúl"))
        eliminarRegistro("Esfero Bic Azúl");

    //Instanciar los implementos de prueba.
    implementoNuevo = new Datos.Implemento("Esfero Bic Rojo", 4);
    implementoSinNombre = new Datos.Implemento("", 2);
    implementoCantidadNula = new Datos.Implemento("Esfero Bic Azúl", 0);

    //Contar la cantidad de implementos inicial.
    cantidadInicialImplementos = consultarCantidadImplemento();
    }

    [Test]
    //Se verifica que se haya ingresado el implemento.
    public void insercionImplemento()
    {
        //Registrar el implemento de prueba.
        implementoNuevo.registrarImplemento();

        //Verificar la inserción.
        Assert.AreEqual(cantidadInicialImplementos + 1,
            consultarCantidadImplemento());

        //Eliminar el registro de prueba otra vez.
        eliminarRegistro("Esfero Bic Rojo");
    }

    int consultarCantidadImplemento()
    {
        cantidadInicialImplementos = (from tablaImplemento
                in consultaTransaccionImplemento.Implemento
            select tablaImplemento).
            Count<Datos.Persistencia.Implemento>();
```

```
        return cantidadInicialImplementos;
    }

    void eliminarRegistro(string registro)
    {
        if (verificarImplemento(registro)) //Si existe el implemento de
        //prueba, entonces
        {
            //Eliminarlo.
            Datos.Persistencia.Implemento implemento =
            (from tablaImplemento
                    in consultaTransaccionImplemento.Implemento
                where tablaImplemento.nombre == registro
                select tablaImplemento).SingleOrDefault();

            consultaTransaccionImplemento.Implemento.
                DeleteOnSubmit(implemento);
            consultaTransaccionImplemento.SubmitChanges();
    }//Fin Si.
}

    bool verificarImplemento(string implemento)
    {
        //Verificar la existencia del implemento
        int cantidadImplemento = (from tablaImplemento
                in consultaTransaccionImplemento.Implemento
            where tablaImplemento.nombre == implemento
            select tablaImplemento).Count<Persistencia.Implemento>();

        if (cantidadImplemento > 0) //Si el implemento existe, entonces
            return true; //Retornar Verdadero.
        else //De lo contrario
            return false; //Retornar Falso.
        //Fin Si.
    }
}
```

También se ha creado otras instancias de la clase para adelantar lo que se hará con las siguientes pruebas unitarias y se ha utilizado rutinas auxiliares de las anteriores clases que sostenían las pruebas unitarias.

d. Rellenar cada comentario correspondiente con código correspondiente a la prueba.

```
public void registrarImplemento()
{
   if (nombre.Length > 0) //Si la cantidad es positiva
   //y se ha ingresado el implemento, entonces
   {
      //Registrar los datos a la base.
      Persistencia.LINQ_PréstamosDataContext insercionImplemento = new
         Persistencia.LINQ_PréstamosDataContext();
      Persistencia.Implemento nuevoImplemento = new
      Persistencia.Implemento();
      nuevoImplemento.nombre = nombre;
      nuevoImplemento.cantidad = cantidad;
```

```
        insercionImplemento.Implemento.InsertOnSubmit(nuevoImplemento);
        insercionImplemento.SubmitChanges();
    }
    //De lo contrario
        //Lanzar una excepción.
        //Terminar la rutina.
    //Fin Si
}
```

e. Ejecutar todas las pruebas (de nuevo).

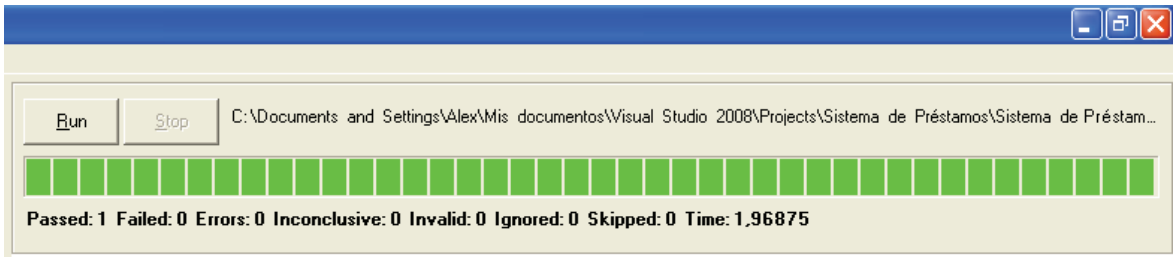


Figura 3.5. Primera Ejecución de la Primera Prueba Unitaria de la Rutina 'registrarImplemento'.

f. Refactorar el código.

Por el momento, no hay nada que mejorar, no se refactora por el momento.

g. Añadir la segunda prueba unitaria.

# de Caso	Descripción	Datos de Entrada	Resultado Esperado
2	Se verifica que lance una excepción cuando no se ha ingresado el implemento.	Implemento no ingresado (cadena de texto vacía). Cantidad cualquiera.	Excepción.

Tabla 3.10. Segunda Prueba Unitaria de la Rutina 'registrarImplemento'.

```
[Test]
[ExpectedException(typeof(ApplicationException))]
public void insercionImplementoSinNombre()
{
    implementoSinNombre.registrarImplemento();
}
```

h. Rellenar cada comentario correspondiente con código correspondiente a la prueba.

```
public void registrarImplemento()
{
    if (nombre.Length > 0) //Si la cantidad es positiva
    //y se ha ingresado el implemento, entonces
```

```
    {
        //Registrar los datos a la base.
        Persistencia.LINQ_PréstamosDataContext insercionImplemento = new
            Persistencia.LINQ_PréstamosDataContext();
        Persistencia.Implemento nuevoImplemento = new
            Persistencia.Implemento();
        nuevoImplemento.nombre = nombre;
        nuevoImplemento.cantidad = cantidad;
        insercionImplemento.Implemento.InsertOnSubmit(nuevoImplemento);
        insercionImplemento.SubmitChanges();
    }
    else //De lo contrario
        //Lanzar una excepción.
        //Terminar la rutina.
        throw new ApplicationException();
    //Fin Si
}
```

i. Ejecutar todas las pruebas (incluyendo la actual).

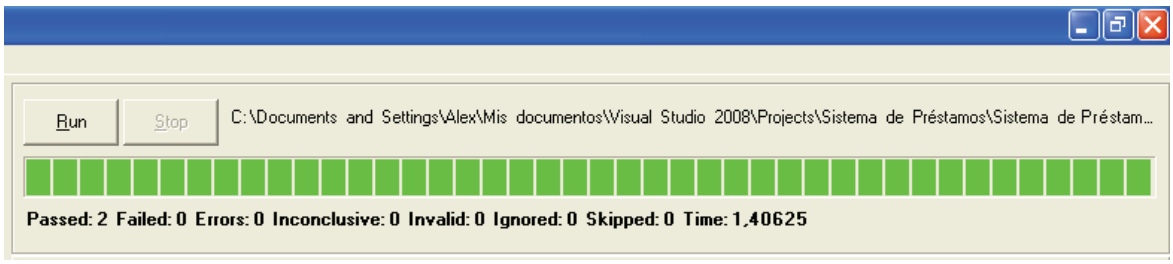


Figura 3.6. Segunda Ejecución de las Pruebas Unitarias de la Rutina 'registrarImplemento'.

j. Refactorar el código.

Por el momento, no hay nada que mejorar, no se refactora por el momento.

k. Añadir la tercera prueba unitaria.

# de Caso	Descripción	Datos de Entrada	Resultado Esperado
3	Se verifica que lance una excepción cuando la cantidad no es positiva.	Implemento ingresado. Cantidad no positiva.	Excepción.

Tabla 3.11. Segunda Prueba Unitaria de la Rutina 'registrarImplemento'.

```
[Test]
[ExpectedException(typeof(ApplicationException))]
public void insercionImplementoCantidadNula()
{
    implementoCantidadNula.registrarImplemento();
}
```

l. Rellenar cada comentario correspondiente con código correspondiente a la prueba.

```
public void registrarImplemento()
{
   if (nombre.Length > 0 && cantidad > 0) //Si la cantidad es positiva
   //y se ha ingresado el implemento, entonces
   {
      //Registrar los datos a la base.
      Persistencia.LINQ_PréstamosDataContext insercionImplemento = new
         Persistencia.LINQ_PréstamosDataContext();
      Persistencia.Implemento nuevoImplemento = new
         Persistencia.Implemento();
      nuevoImplemento.nombre = nombre;
      nuevoImplemento.cantidad = cantidad;
      insercionImplemento.Implemento.InsertOnSubmit(nuevoImplemento);
      insercionImplemento.SubmitChanges();
   }
   else //De lo contrario
      //Lanzar una excepción.
      //Terminar la rutina.
      throw new ApplicationException();
   //Fin Si
}
```

En realidad no se ha agregado código sino simplemente se ha modificado la estructura selectiva para que detecte la cantidad.

m. Ejecutar todas las pruebas (incluyendo la actual).

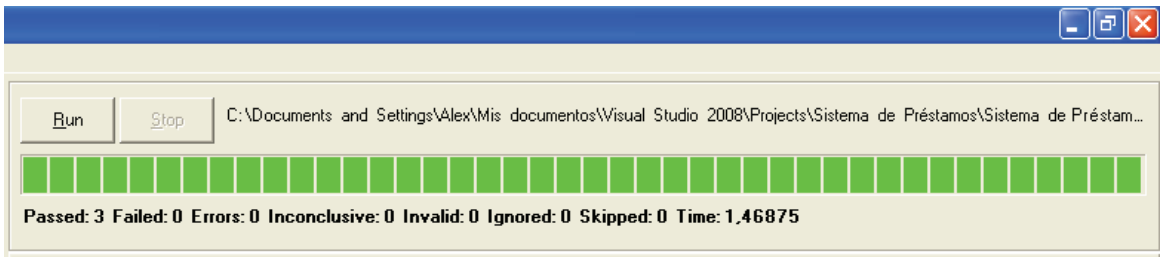


Figura 3.7. Tercera Ejecución de las Pruebas Unitarias de la Rutina 'registrarImplemento'.

n. Refactorar el código.

Por el momento, no hay nada que mejorar, no se refactora por el momento.

3) Revisar y probar la clase en conjunto.

Se prueban todas las clases que sostienen las pruebas unitarias de todas las rutinas que se implementaron, en este caso, se probará toda la clase Implemento.

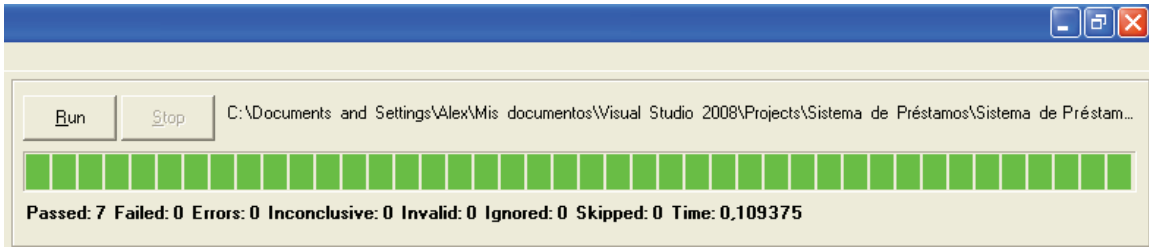


Figura 3.8. Ejecución de Todas las Pruebas Unitarias de la Clase 'Implemento'.

Esta es la forma cómo ha quedado la clase Implemento.

```
class Implemento
{
   private string nombre;
   private int cantidad;

   //Recoge los datos obligatorios de un implemento para una inserción.
   public Implemento(string nombre, int cantidad)
   {
      //Verificar si se ingresó el implemento antes.
      int existenciaImplemento;
      Persistencia.LINQ_PréstamosDataContext consultaImplementoConocido
         = new Persistencia.LINQ_PréstamosDataContext();
      existenciaImplemento = (from tablaImplemento
            in consultaImplementoConocido.Implemento
         where tablaImplemento.nombre == nombre
         select tablaImplemento).Count<Persistencia.Implemento>();

      if (existenciaImplemento > 0) //Si se ha ingresado antes, entonces
      {
         //Lanzar una excepción.
         //Terminar la rutina.
         throw new ApplicationException();
      }
      else //De lo contrario
      {
         //Almacenar los datos de los parámetros a los atributos.
         this.nombre = nombre;
         this.cantidad = cantidad;
      }//Fin Si
   }

   //Carga el registro de un implemento a la clase desde una consulta a
   //base de datos.
   public Implemento(int id)
   {
      //Cargar de la base los datos del implemento a los atributos.
      Persistencia.LINQ_PréstamosDataContext consultaImplemento = new
         Persistencia.LINQ_PréstamosDataContext();
      Persistencia.Implemento implemento = (from tablaImplemento
            in consultaImplemento.Implemento
         where tablaImplemento.identificador == id
         select tablaImplemento).Single();
      this.nombre = implemento.nombre;
      this.cantidad = implemento.cantidad;
   }
```

```
    //Recupera la cantidad del implemento.
    public int consultarCantidad()
    {
        //Retornar la cantidad.
        return cantidad;
    }

    //Asigna la cantidad del implemento.
    public void asignarCantidad(int cantidad)
    {
        //Modificar la cantidad del implemento.
        this.cantidad = cantidad;
    }

    //Se encarga de almacenar en la base de datos el registro del
    //Implemento.
    public void registrarImplemento()
    {
        if (nombre.Length > 0 && cantidad > 0) //Si la cantidad es
        //positiva y se ha ingresado el implemento, entonces
        {
            //Registrar los datos a la base.
            Persistencia.LINQ_PréstamosDataContext insercionImplemento
                = new Persistencia.LINQ_PréstamosDataContext();
            Persistencia.Implemento nuevoImplemento
                = new Persistencia.Implemento();
            nuevoImplemento.nombre = nombre;
            nuevoImplemento.cantidad = cantidad;
            insercionImplemento.Implemento.InsertOnSubmit(nuevoImplemento);
            insercionImplemento.SubmitChanges();
        }
        else //De lo contrario
            //Lanzar una excepción.
            //Terminar la rutina.
            throw new ApplicationException();
    //Fin Si
    }
}
```

'PPP' CON RUTINAS INACCESIBLES A LAS PRUEBAS UNITARIAS.

Con respecto a este caso de estudio se procederá a realizar la implementación de todas las interfaces de caso de estudio (las interfaces de este caso de estudio no pueden ser accedidas por las pruebas unitarias) por lo que se procederá a realizarlo de esta forma, en este caso se tomará la interfaz 'Registrar Implemento Nuevo' ya que el IDE diseña las interfaces como clases cuyos objetos en su diseño son atributos de esta clase.

1) Verificar el diseño previo de las rutinas y de la clase.

Clase: Registrar Implemento Nuevo.

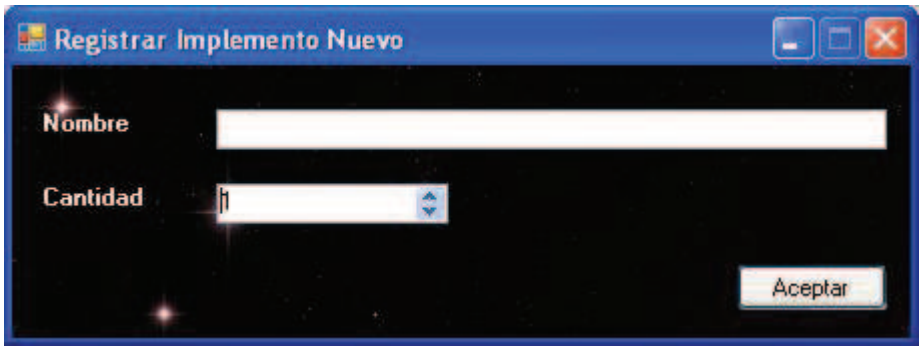


Figura 3.9. Interfaz 'Registrar Implemento Nuevo'.

Figura 3.10. Diseño de la Interfaz 'Registrar Implemento Nuevo' como Clase.

En este caso se han descartado las rutinas que fueron creadas por el IDE para solo trabajar por lo diseñado en el caso de estudio.

Diseños de las Rutinas:

- **texNombre_Leave:**

Esta rutina se activa cuando se deja la caja de texto 'texNombre', pero su diseño radica en 'registrarNombre'.

- **butAceptar_Click:**

Esta rutina se activa cuando se hace clic en el botón 'butAceptar', pero su diseño radica en 'registrarImplemento'.

- **registrarNombre:**

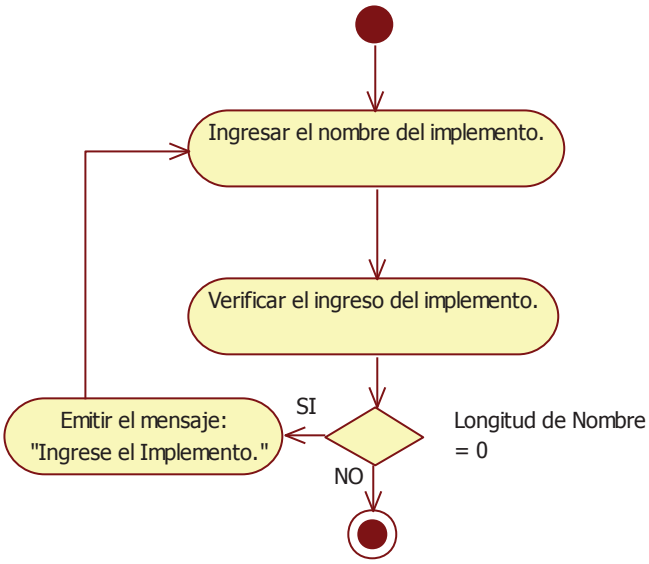


Figura 3.11. Diagrama de Actividades de la Rutina 'registrarNombre'.

- **registrarImplemento:**

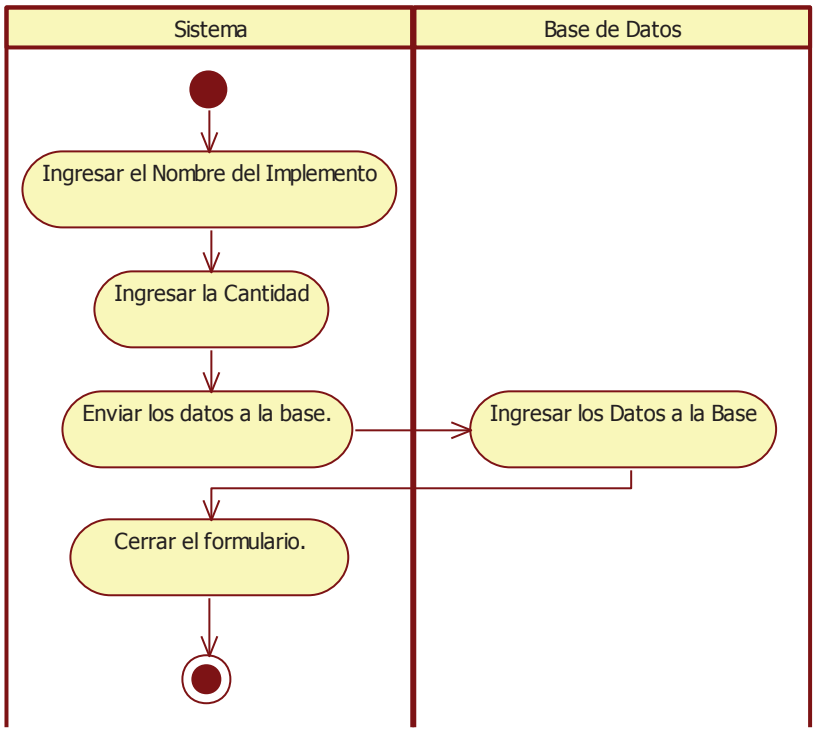


Figura 3.12. Diagrama de Actividades de la Rutina 'registrarImplemento'.

Este diseño está basado en el diagrama de actividades 'Registrar Implemento Nuevo' del caso de uso 'registrarImplementoNuevo' (anexado en los archivos de este caso de estudio).

2) Convertir el diseño de las rutinas en pseudocódigo.

```
Inicio texNombre_Leave (sender, e)
      Registrar nombre del implemento.
Fin

Inicio butAceptar_Click (sender, e)
      Verificar el ingreso del implemento.

      Si se ha ingresado el implemento, entonces
            Registrar implemento
      Fin Si
Fin

Inicio registrarNombre ()
      Sino se ha ingresado el implemento, entonces
            Emitir un mensaje de error y colocar el cursor para ingresar
            el implemento.
      Fin Si.
Fin

Inicio registrarImplemento ()
      Ingresar los datos a la base.
      Cerrar la ventana.
Fin
```

Entre la rutina del botón y la rutina 'registrarNombre' se reparte la funcionalidad del diseño de la rutina 'registrarNombre' con el fin de que se controle lo ingresado en la ventana.

3) Escribir la declaración de las rutinas.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace Sistema_de_Préstamos
{
    public partial class RegistrarImplementoNuevo : Form
    {
        public RegistrarImplementoNuevo()
        {
            InitializeComponent();
        }

        private void texNombre_Leave(object sender, EventArgs e)
        {

        }

        private void butAceptar_Click(object sender, EventArgs e)
        {

        }
```

```
        void registrarNombre()
        {

        }

        void registrarImplemento()
        {

        }
    }
}
```

No solo se ha escrito la declaración de las rutinas sino que se visualizan las librerías declaradas y el espacio de nombres que el mismo IDE genera para la declaración de las rutinas, incluyendo la rutina constructora de la interfaz con su implementación predeterminada.

4) Convertir el pseudocódigo en comentarios de alto nivel.

```
public partial class RegistrarImplementoNuevo : Form
{
    public RegistrarImplementoNuevo()
    {
        InitializeComponent();
    }

    private void texNombre_Leave(object sender, EventArgs e)
    {
        //Registrar nombre del implemento.
    }

    private void butAceptar_Click(object sender, EventArgs e)
    {
        //Verificar el ingreso del implemento.

        //Si se ha ingresado el implemento, entonces
            //Registrar implemento
        //Fin Si.
    }

    void registrarNombre()
    {
    //Sino se ha ingresado el implemento, entonces
        //Emitir un mensaje de error y colocar el cursor para
        //ingresar el implemento.
    //Fin Si.
    }

    void registrarImplemento()
    {
        //Ingresar los datos a la base.
        //Cerrar la ventana.
    }
}
```

5) Llenar con código cada comentario de la rutina.

```
public partial class RegistrarImplementoNuevo : Form
{
    public RegistrarImplementoNuevo()
    {
        InitializeComponent();
    }

    private void texNombre_Leave(object sender, EventArgs e)
    {
        //Registrar nombre del implemento.
        registrarNombre();
    }

    private void butAceptar_Click(object sender, EventArgs e)
    {
        //Verificar el ingreso del implemento.
        registrarNombre();

        if (texNombre.Text.Length > 0) //Si se ha ingresado el
        //implemento, entonces
            //Registrar implemento
            registrarImplemento();
        //Fin Si.
    }

    void registrarNombre()
    {
        if (texNombre.Text.Length == 0) //Sino se ha ingresado el
        //implemento, entonces
        {
            //Emitir un mensaje de error y colocar el cursor para
            //ingresar el implemento.
            MessageBox.Show("Ingrese el Implemento.", this.Text,
                MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
            texNombre.Focus();
        } //Fin Si.
    }

    void registrarImplemento()
    {
        //Ingresar los datos a la base.
        Datos.Implemento implemento = new
            Datos.Implemento(texNombre.Text, (int)numCantidad.Value);
        implemento.registrarImplemento();

        //Cerrar la ventana.
        this.Close();
    }
}
```

6) Verificar y refactorar el código.

```
Inicio registrarImplemento ()
      Sino se ingresado algún implemento antes, entonces
            Ingresar los datos a la base.
            Cerrar la ventana.
      De lo contrario
```

```
            Emitir el mensaje de error respecto al implemento repetido.
      Fin Si
Fin
```

```
void registrarImplemento()
{
    try //Sino se ingresado algún implemento antes, entonces
    {
        //Ingresar los datos a la base.
        Datos.Implemento implemento = new
            Datos.Implemento(texNombre.Text, (int)numCantidad.Value);
        implemento.registrarImplemento();

        //Cerrar la ventana.
        this.Close();
    }
    catch (Exception e) //De lo contrario.
    {
        //Emitir el mensaje de error respecto al implemento repetido.
        MessageBox.Show("Implemento Repetido",
            this.Text,MessageBoxButtons.OK, MessageBoxIcon.Error);
    } //Fin Si.
}
```

Se modifico primero el pseudocódigo para ver como se modificaría la rutina, después se volvió a convertir en comentarios de alto nivel para completar con código lo que faltaba y así logró controlarse para evitar ingresos repetidos. Se ha requerido realizar la implementación de la clase 'Implemento' para implementar esta interfaz.

3.2.5. VERIFICACION.

Para aplicar la verificación, se lo procederá a hacer con la interfaz 'Registrar Implemento Nuevo'.

1) Verificar mentalmente las rutinas por errores.

```
public partial class RegistrarImplementoNuevo : Form
{
    public RegistrarImplementoNuevo()
    {
        InitializeComponent();
    }

    private void texNombre_Leave(object sender, EventArgs e)
    {
        //Registrar nombre del implemento.
        registrarNombre();
    }

    private void butAceptar_Click(object sender, EventArgs e)
    {
        //Verificar el ingreso del implemento.
```

```
            registrarNombre();

            if (texNombre.Text.Length > 0) //Si se ha ingresado el
            //implemento, entonces
                //Registrar implemento
                registrarImplemento();
            //Fin Si.
        }

        void registrarNombre()
        {
            if (texNombre.Text.Length == 0) //Sino se ha ingresado el
            //implemento, entonces
            {
                //Emitir un mensaje de error y colocar el cursor para
                //ingresar el implemento.
                MessageBox.Show("Ingrese el Implemento.", this.Text,
                    MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
                texNombre.Focus();
            } //Fin Si.
        }

        void registrarImplemento()
        {
            try //Sino se ingresado algún implemento antes, entonces
            {
                //Ingresar los datos a la base.
                Datos.Implemento implemento = new
                    Datos.Implemento(texNombre.Text,
                     (int)numCantidad.Value);
                implemento.registrarImplemento();

                //Cerrar la ventana.
                this.Close();
            }
            catch (Exception e) //De lo contrario.
            {
                //Emitir el mensaje de error respecto al implemento
                //repetido.
                MessageBox.Show("Implemento Repetido",
                    this.Text,MessageBoxButtons.OK, MessageBoxIcon.Error);
            } //Fin Si.
        }
}
```

2) Compilar las rutinas.

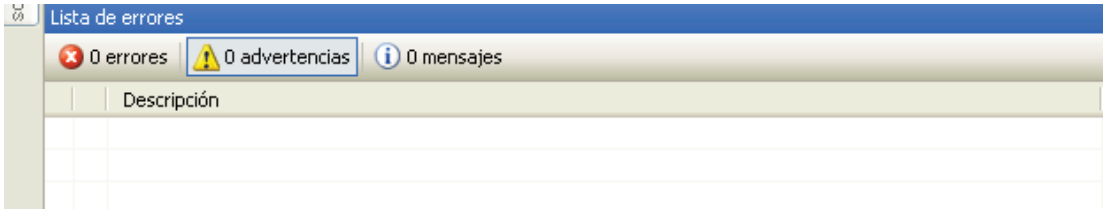


Figura 3.13. Compilación de las rutinas en Visual Studio 2008.

3) Recorrer el código en el depurador.

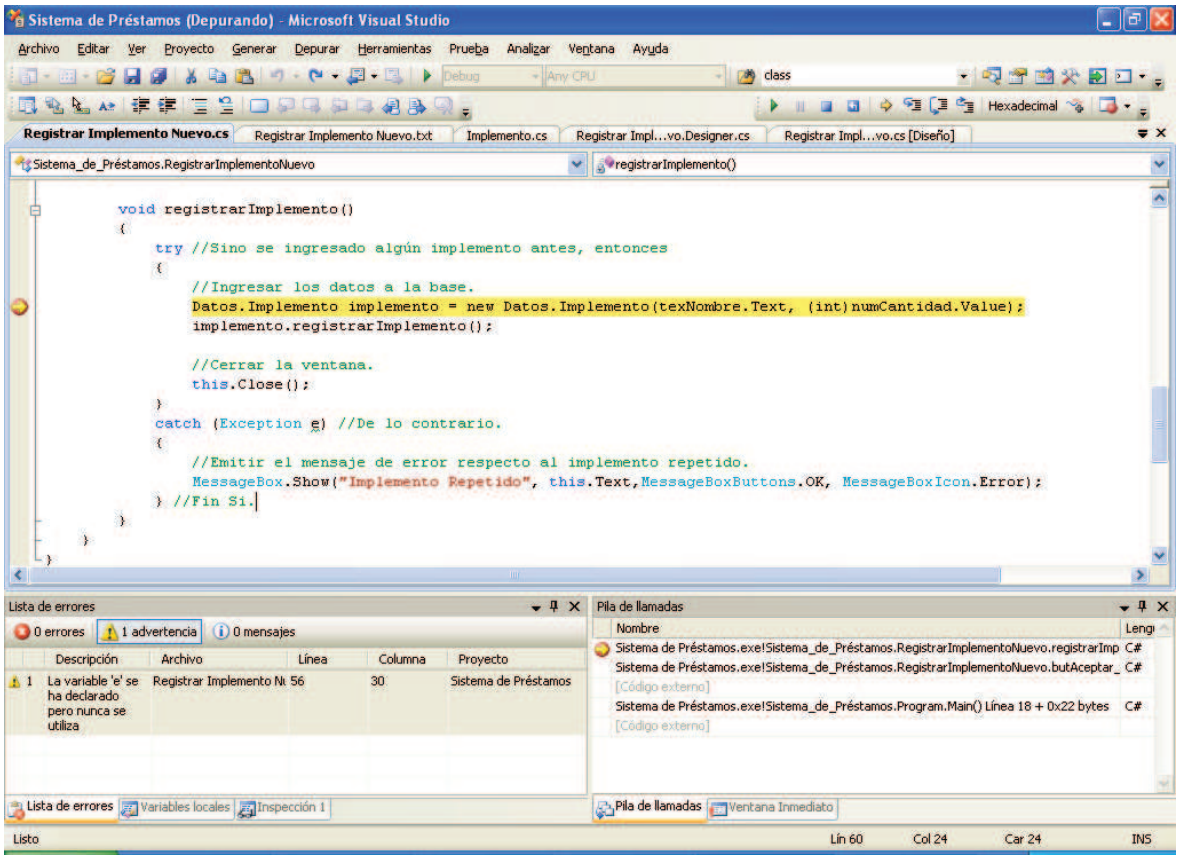


Figura 3.14. Depuración de la rutina 'registrarImplemento'.

Se logra observar que en el momento de la depuración se encuentra una variable que no se utiliza nunca y es la razón de la advertencia en el IDE por lo que se procederá a eliminarla en el siguiente paso.

4) Si existen errores (o advertencias), removerlos de las rutinas.

```
void registrarImplemento()
{
    try //Sino se ingresado algún implemento antes, entonces
    {
        //Ingresar los datos a la base.
        Datos.Implemento implemento = new
        Datos.Implemento(texNombre.Text, (int)numCantidad.Value);
        implemento.registrarImplemento();

        //Cerrar la ventana.
        this.Close();
    }
    catch //De lo contrario.
    {
        //Emitir el mensaje de error respecto al implemento repetido.
        MessageBox.Show("Implemento Repetido",
            this.Text,MessageBoxButtons.OK, MessageBoxIcon.Error);
    } //Fin Si.
```

```
}
```

Se elimina la razón de la advertencia, es decir, se elimina la variable ‘e’ por lo que la advertencia desaparece.

3.2.6. INTEGRACION.

La integración no se realizará por la razón expuesta en el punto ‘Preparar las Pruebas de Integración’.

3.2.7. PRUEBAS DE INTEGRACION.

Las pruebas de integración no se realizarán por la razón expuesta en el punto ‘Preparar las Pruebas de Integración’.

3.3 ANÁLISIS GENERAL DE RESULTADOS.

ESTADISTICAS DE CODIGO.

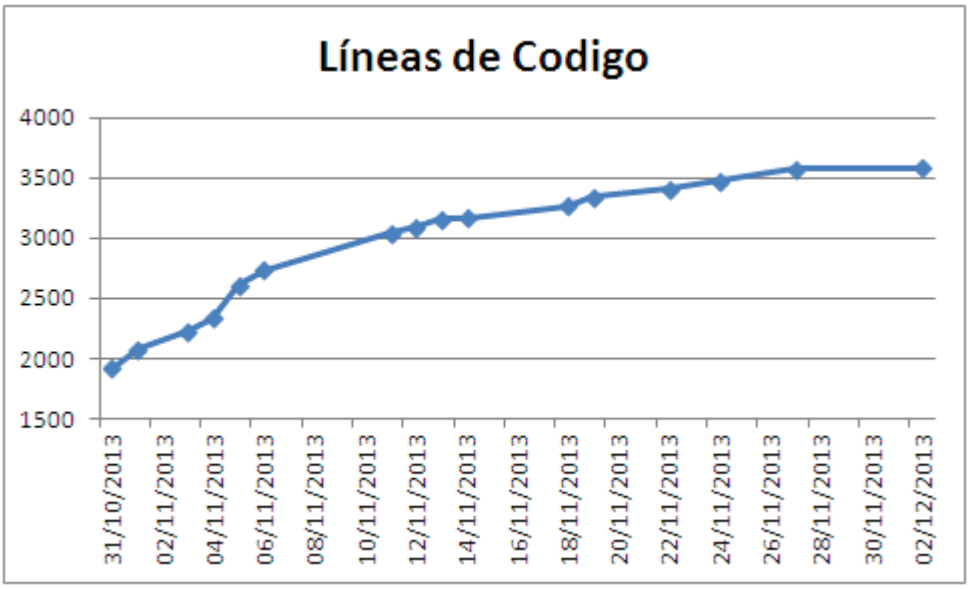


Figura 3.15. Comportamiento del Número de Líneas de Código.

El incremento es pronunciado en la fase de pruebas de unidad, en el 06 de Noviembre del 2013 (ver Anexo D), se vuelve menos pronunciada ya que se implementan menos rutinas hasta que termina de implementarse la funcionalidad básica.

ESTADISTICAS DE CLASES.

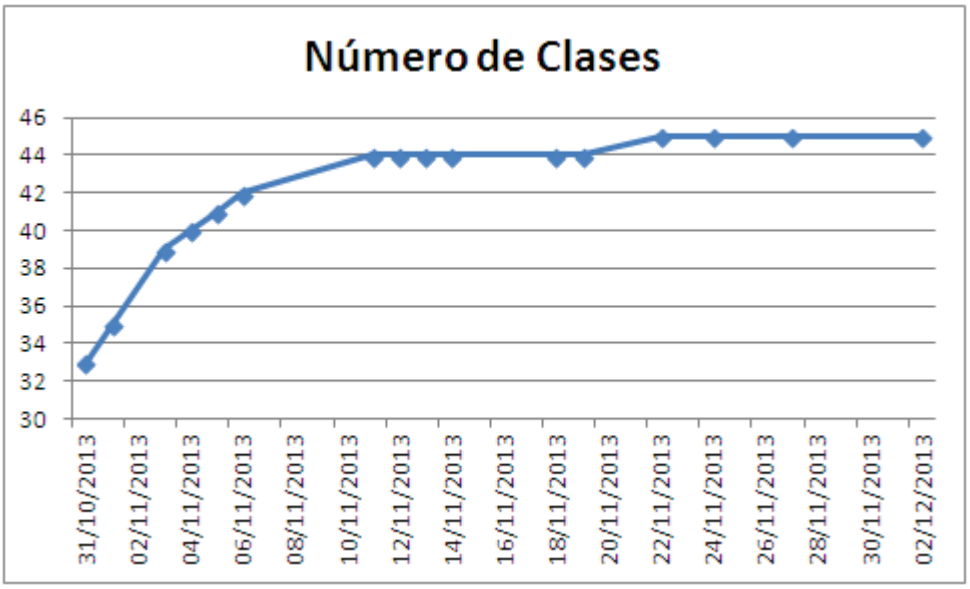


Figura 3.16. Comportamiento del Número de Clases.

El incremento de las clases se debe al periodo de pruebas de unidad debido a que cada clase creada fue para sostener todas las alternativas de las rutinas y cuando el número de clases se vuelve constante es porque se implementan las interfaces del sistema, el incremento repentino de clases el 22 de Noviembre del 2013 (ver Anexo D) fue por la creación de una enumeración y se la cuenta como una clase más.

ESTADISTICAS DE RUTINAS.

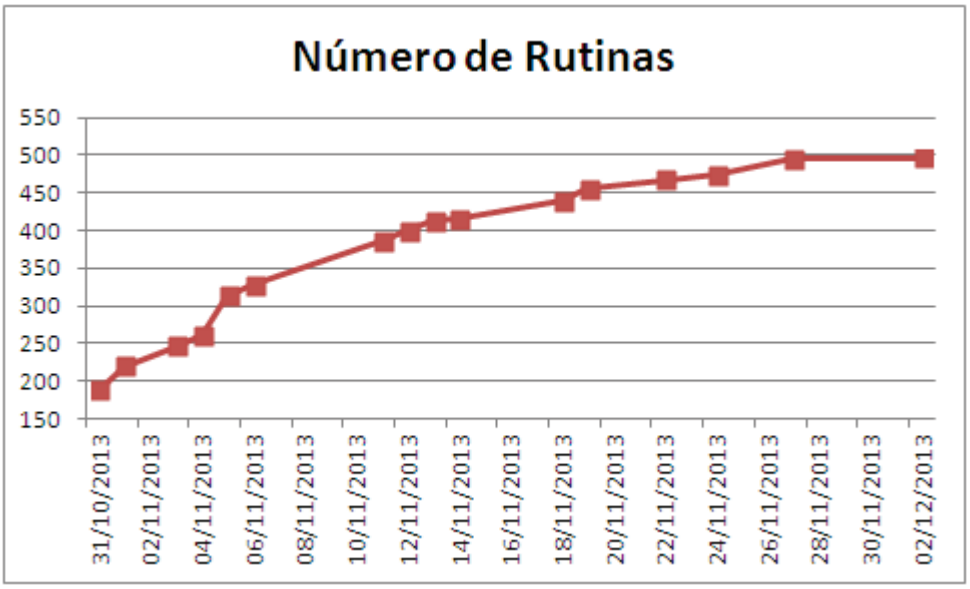


Figura 3.17. Comportamiento del Número de Rutinas.

El incremento de rutinas es pronunciado debido a que se emplean las mismas rutinas para crear el escenario de las pruebas de unidad y cuando es menos pronunciada la pendiente es por la implementación de las interfaces hasta que la pendiente tiende a ser horizontal debido a que se ha culminado la implementación de la funcionalidad básica del caso de estudio.

CAPÍTULO 4. CONCLUSIONES Y RECOMENDACIONES.

4.1 CONCLUSIONES.

- Se ha observado en Quito que la mayoría de los desarrolladores implementan código susceptible a fallas. Esto se debe a que no se había realizado el diseño previamente o a que no se basaban en el diseño realizado.
- En Quito, los programadores que trabajan en las empresas de desarrollo no siguen un método para la implementación del software, fundamentalmente porque carecen del conocimiento para realizar este proceso.
- El trabajo realizado exigió mucho esfuerzo y mucha investigación, ya que no se tenía previamente un conocimiento suficiente sobre el proceso de implementación de sistemas software.
- Los desarrolladores deben aceptar el cambio que puede venir de requisitos que no son estables y saber responder ante cualquier otro cambio que se deba realizar en la implementación del software.
- Al final de este trabajo se ha podido determinar que para realizar una implementación de software robusta y de calidad, este proceso debe estar basado en cuatro principios fundamentales que son: minimización de la complejidad, anticipación al cambio, implementación susceptible de ser verificada y aplicación estricta de estándares de implementación.
- También se ha podido determinar que para implementar correctamente software se debe seguir un proceso de gestión de la implementación que implica usar modelos de implementación, planificar y medir la implementación.
- Finalmente también es importante resaltar el hecho de que una adecuada implementación de software dependerá del diseño previo, de la adecuada selección del lenguaje de implementación, de una codificación de alta calidad, de la aplicación de pruebas de unidad y de integración y de un inteligente uso de código reutilizable.

4.2 RECOMENDACIONES.

- Se recomienda utilizar el Desarrollo Guiado por Pruebas para desarrollar sistemas de pequeña y mediana complejidad con el fin de asegurar un código de alta calidad.

- Es recomendable generar el código de un sistema tomando en cuenta el diseño realizado, que puede haber sido obtenido mediante modelación y pseudocódigo.
- Se recomienda profundizar en el estudio de los tipos de notación que se utilizan para los lenguajes de programación: lingüístico, formal y visual.
- Se recomienda realizar la implementación de software basado en los fundamentos para implementación propuestos en el SWEBOK.

BIBLIOGRAFÍA

LIBROS Y ESTÁNDARES

[1] GUANOCHANGA, Gabriela; VALLEJO, Oscar. Propuesta para la Realización del Proceso de Ingeniería de Requisitos para Empresas Desarrolladoras de Software en Quito. Tesis de Ingeniería en Sistemas. Escuela Politécnica Nacional. Quito 2010.

[2] MALHOTRA, Naresh K. Investigación de Mercados. Prentice Hall, México 2008.

[3] IEEE COMPUTER SOCIETY. Guide to the Software Engineering Body of Knowledge. Version 3. CS Press. US. 2004.

[4] SOMMERVILLE, Ian. Ingeniería de Software. Séptima Edición. Pearson Educación. Madrid. 2005.

[5] McCONNELL, Steve. Code Complete: A Practical Handbook of Software Construction. Second Edition. Microsoft Press. US. 2004.

[6] IEEE COMPUTER SOCIETY. IEEE Standard for Information Technology—Software Life Cycle Processes—Reuse Processes. Estándar 1517-1999. US. 1999.

INTERNET.

[7] WIKIPEDIA. Programación. http://es.wikipedia.org/wiki/Programaci%C3%B3n. Acceso, 05 de Abril de 2012.

[8] WIKIPEDIA. List of unit testing frameworks. http://en.wikipedia.org/wiki/List_of_unit_testing_frameworks. Acceso, 28 de Diciembre de 2012.

[9] EXFORSYS. Integration Testing: Why? What? & How? www.exforsys.com/tutorials/testing/integration-testing-whywhathow.html. Acceso, 04 de Enero de 2013.

[10] WIKIPEDIA. Test-driven development. http://en.wikipedia.org/wiki/Test-driven_development. Acceso, 13 de Enero de 2013.

[11] WIKIPEDIA. Integrated development environment. http://en.wikipedia.org/wiki/Integrated_development_environment. Acceso, 30 de Enero de 2013.

[12] WIKIPEDIA. Test automation framework. http://en.wikipedia.org/wiki/Test_automation_framework. Acceso, 30 de Enero de 2013.

[13] WIKIPEDIA. Computer-aided software engineering. http://en.wikipedia.org/wiki/Computer-aided_software_engineering. Acceso, 30 de Enero de 2013.

[14] WIKIPEDIA. Unified Modeling Language. http://en.wikipedia.org/wiki/Unified_Modeling_Language. Acceso, 26 de Noviembre de 2013.

[15] CEVALLOS, Alejandro. Encuesta de Desarrollo de Software. http://tinyurl.com/encuesta-desarrollo-software. Acceso, 27 de Enero de 2014.

GLOSARIO

IEEE: Institute of Electrical and Electronics Engineers (Instituto de Ingenieros Eléctricos y Electrónicos).

ISO: International Organization for Standardization (Organización Internacional para Estandarización).

XP: Extreme Programming (Programación Extrema).

RUP: Rational Unified Process (Proceso Unificado Rational).

DGP: Desarrollo Guiado por Pruebas.

PPP: Proceso de Programación por Pseudocódigo.

IDE: Integrated Development Environment (Ambiente de Desarrollo Integrado <<ADI>>).

UML: Unified Modeling Language (Lenguaje de Modelado Unificado).

SWEBOK: Software Engineering Body of Knowledge (Cuerpo de Conocimiento de Ingeniería de Software).

ANEXO A

Resultados de la Encuesta

Marca temporal	1. Cuando usted desarrolla ¿tiende a crear complejas estructuras de código?	2. Cuando usted desarrolla ¿utiliza estándares y técnicas conocidos o se basa en su experiencia personal?	3. Cuando usted desarrolla ¿contempla la posibilidad de que su código sea alterado?	4. En el equipo de desarrollo, ¿se utilizan estándares para los documentos y sus contenidos?	5. Cuando programa, ¿utilizan estándares de programación?	6. ¿Utiliza estándares para modelar las aplicaciones (como UML por ejemplo)?
2/14/2012 23:44:50	Si.	Utilizo estándares, técnicas y mi experiencia personal.	Siempre.	Ocasionalmente.	Frecuentemente.	Frecuentemente.
2/15/2012 0:12:46	Ocasionalmente.	Utilizo estándares y mi experiencia personal.	Ocasionalmente.	Frecuentemente.	Frecuentemente.	Ocasionalmente.
2/15/2012 4:26:41	Frecuentemente.	Utilizo estándares, técnicas y mi experiencia personal.	Siempre.	Siempre.	Frecuentemente.	Siempre.
2/15/2012 9:20:54	Nunca.	Solo utilizo mi experiencia personal.	Nunca.	Ocasionalmente.	Frecuentemente.	Frecuentemente.
2/15/2012 21:43:27	Ocasionalmente.	Utilizo estándares y mi experiencia personal.	Nunca.	Frecuentemente.	Frecuentemente.	Siempre.
2/16/2012 21:08:16	Ocasionalmente.	Utilizo estándares y mi experiencia personal.	Ocasionalmente.	Frecuentemente.	Frecuentemente.	Frecuentemente.

2/27/2012 19:57:10	Frecuentemente.	Solo utilizo mi experiencia personal.	Frecuentemente.	Ocasionalmente.	Ocasionalmente.	Ocasionalmente.
2/27/2012 20:04:27	Ocasionalmente.	Utilizo estándares, técnicas y mi experiencia personal.	Frecuentemente.	Frecuentemente.	Frecuentemente.	Ocasionalmente.
2/28/2012 8:32:35	Ocasionalmente.	Utilizo técnicas y mi experiencia personal.	Frecuentemente.	Nunca.	Ocasionalmente.	Nunca.
2/28/2012 9:19:17	Ocasionalmente.	Utilizo estándares y mi experiencia personal.	Frecuentemente.	Ocasionalmente.	Frecuentemente.	Ocasionalmente.
2/29/2012 15:06:40	Ocasionalmente.	Utilizo técnicas y mi experiencia personal.	Ocasionalmente.	Ocasionalmente.	Frecuentemente.	Siempre.
3/2/2012 17:36:20	Frecuentemente.	Utilizo técnicas y mi experiencia personal.	Frecuentemente.	Ocasionalmente.	Ocasionalmente.	Ocasionalmente.
3/5/2012 14:44:47	Si.	Utilizo estándares y mi experiencia personal.	Siempre.	Ocasionalmente.	Frecuentemente.	Frecuentemente.
3/6/2012 10:09:57	Ocasionalmente.	Utilizo estándares y mi experiencia personal.	Frecuentemente.	Frecuentemente.	Siempre.	Frecuentemente.
3/8/2012 13:55:47	Nunca.	Utilizo estándares, técnicas y mi experiencia personal.	Siempre.	Frecuentemente.	Frecuentemente.	Frecuentemente.
3/8/2012 16:40:15	Frecuentemente.	Utilizo estándares, técnicas y mi experiencia personal.	Frecuentemente.	Ocasionalmente.	Frecuentemente.	Siempre.

3/11/2012 15:51:05	Nunca.	Utilizo técnicas y mi experiencia personal.	Siempre.	Ocasionalmente.	Ocasionalmente.	Ocasionalmente.
3/13/2012 11:38:31	Ocasionalmente.	Utilizo estándares, técnicas y mi experiencia personal.	Ocasionalmente.	Siempre.	Ocasionalmente.	Siempre.
3/13/2012 11:46:13	Si.	Utilizo estándares y mi experiencia personal.	Siempre.	Ocasionalmente.	Ocasionalmente.	Frecuentemente.
3/13/2012 11:48:31	Frecuentemente.	Utilizo estándares, técnicas y mi experiencia personal.	Frecuentemente.	Siempre.	Frecuentemente.	Ocasionalmente.

Marca temporal	7. Usted (o su equipo de desarrollo) ¿realizan revisiones del código?	8. Usted (o su equipo de desarrollo) ¿realizan pruebas de unidad?	9. Usted (o su equipo de desarrollo) ¿realizan pruebas de integración?	10. Usted (o su equipo de desarrollo) ¿desarrollan código organizado para dar soporte a las pruebas automatizadas?	11. Usted (o su equipo de desarrollo) ¿qué modelo de ciclo de vida para el desarrollo de software utilizan en el entorno donde desarrollan?	12. Usted (o su equipo de desarrollo) ¿qué métricas utilizan para medir el código que se ha desarrollado (puede mencionar más de uno)?	13. Usted (o su equipo de desarrollo) ¿realizan cambios pequeños o grandes que anticipen las posibles fallas del sistema a desarrollar?
2/14/2012 23:44:50	Frecuentemente.	Siempre.	Ocasionalmente.	Nunca.	Programación extrema.	Complejidad del código., Esfuerzo., Planificación.	Frecuentemente.
2/15/2012 0:12:46	Frecuentemente.	Frecuentemente.	Ocasionalmente.	Ocasionalmente.	RUP	Código desarrollado., Esfuerzo.	Frecuentemente.

2/15/2012 4:26:41	Siempre.	Siempre.	Siempre.	Siempre.	Up	Código desarrollado., Código modificado.	Ocasionalmente.
2/15/2012 9:20:54	Ocasionalmente.	Ocasionalmente.	Frecuentemente.	Nunca.	Programación extrema.	Código reusado.	Ocasionalmente.
2/15/2012 21:43:27	Siempre.	Frecuentemente.	Frecuentemente.	Frecuentemente.	Prototipado evolutivo.	Código desarrollado., Código modificado.	Frecuentemente.
2/16/2012 21:08:16	Ocasionalmente.	Frecuentemente.	Frecuentemente.	Ocasionalmente.	Iconix.	Complejidad del código., Planificación.	Frecuentemente.
2/27/2012 19:57:10	Frecuentemente.	Ocasionalmente.	Ocasionalmente.	Ocasionalmente.	Ninguno.	Código desarrollado.	Ocasionalmente.
2/27/2012 20:04:27	Frecuentemente.	Ocasionalmente.	Frecuentemente.	Ocasionalmente.	Programación extrema.	Código modificado., Código reusado., Complejidad del código.	Ocasionalmente.
2/28/2012 8:32:35	Ocasionalmente.	Frecuentemente.	Frecuentemente.	Nunca.	Prototipado evolutivo.	Código desarrollado., Esfuerzo.	Ocasionalmente.
2/28/2012 9:19:17	Frecuentemente.	Frecuentemente.	Frecuentemente.	Ocasionalmente.	Scrum.	Código desarrollado., Esfuerzo., Planificación.	Ocasionalmente.
2/29/2012 15:06:40	Ocasionalmente.	Ocasionalmente.	Ocasionalmente.	Frecuentemente.	Programación extrema.	Código desarrollado., Código reusado., Complejidad del código., Planificación.	Ocasionalmente.

3/2/2012 17:36:20	Frecuentemente.	Frecuentemente.	Ocasionalmente.	Nunca.	Programación extrema.	Código desarrollado., Código reusado., Complejidad del código.	Frecuentemente.
3/5/2012 14:44:47	Siempre.	Siempre.	Siempre.	Siempre.	Programación extrema.	Código desarrollado., Código modificado., Código reusado., Código eliminado., Complejidad del código., Esfuerzo., Planificación.	Frecuentemente.
3/6/2012 10:09:57	Ocasionalmente.	Ocasionalmente.	Siempre.	Frecuentemente.	Prototipado evolutivo.	Código desarrollado., Esfuerzo., Planificación.	Frecuentemente.
3/8/2012 13:55:47	Frecuentemente.	Ocasionalmente.	Ocasionalmente.	Ocasionalmente.	Scrum.	Código desarrollado., Complejidad del código., Esfuerzo., Planificación.	Ocasionalmente.
3/8/2012 16:40:15	Siempre.	Ocasionalmente.	Siempre.	Frecuentemente.	Iconix.	Código desarrollado., Tasas para reparar y encontrar fallos., Esfuerzo., Planificación.	Ocasionalmente.
3/11/2012 15:51:05	Ocasionalmente.	Ocasionalmente.	Ocasionalmente.	Ocasionalmente.	RUP	Planificación.	Ocasionalmente.

3/13/2012 11:38:31	Ocasionalmente.	Ocasionalmente.	Ocasionalmente.	Nunca.	Ninguno.	Código reusado., Complejidad del código., Estadísticas de inspección del código., Esfuerzo.	Ocasionalmente.
3/13/2012 11:46:13	Ocasionalmente.	Ocasionalmente.	Ocasionalmente.	Frecuentemente.	Programación extrema.	Código modificado., Código reusado., Código eliminado.	Frecuentemente.
3/13/2012 11:48:31	Frecuentemente.	Siempre.	Ocasionalmente.	Frecuentemente.	Prototipado evolutivo.	Código modificado., Código reusado., Código eliminado.	Ocasionalmente.

Marca temporal	14. Usted (o su equipo de desarrollo) ¿utilizan los diseños realizados para implementar el software a desarrollar?	15. ¿Qué lenguajes de implementación usted utiliza (puede escoger más de uno)?	16. Usted (o su equipo de desarrollo) ¿Conoce las notaciones que tiene el lenguaje de programación que usted utiliza?	¿Qué notaciones tiene el lenguaje de programación que usted utiliza (puede escoger más de uno)?	17. ¿Qué consideraciones usted (o el equipo de desarrollo) utilizan para codificar las aplicaciones (puede escoger más de uno)?	18. ¿Qué tareas usted (o el equipo de desarrollo) realizan para el reuso de código en el desarrollo de un sistema (puede escoger más de uno)?	19. ¿Qué técnicas usted (o el equipo de desarrollo) realizan para la calidad de la implementación en el desarrollo de un sistema (puede escoger más de uno)?

2/14/2012 23:44:50	Frecuentemente.	Lenguajes de programación.	Si.	Lingüístico., Visual.	Técnicas para crear código entendible., El uso de clases, tipos enumerados, variables, constantes nombradas, y otras entidades similares., Manejo de las condiciones de error – tanto las excepciones y errores previstos (como el ingreso de datos incorrectos, por ejemplo)., Prevención de las violaciones de seguridad a nivel de código (saturaciones del búfer o los desbordamientos del índice de matriz, por ejemplo)., Organización del código fuente (en las declaraciones, las rutinas, clases, paquetes, u otras estructuras)., Documentación del código., Ajuste (tuning) del código.	La selección de unidades reutilizables, bases de datos, procedimientos de prueba, o datos de prueba., La evaluación de código o reutilización de prueba.	Pruebas de unidad y de integración., Prueba de primer desarrollo., Depuración., Revisiones técnicas.

2/15/2012 0:12:46	Siempre.	Lenguajes de configuración (como los archivos de configuración de un sistema operativo)., Lenguajes de kit de herramientas (que sirven para elaborar código a partir de un código reutilizable)., Lenguajes de programación.	Si.	Lingüístico., Formal.	El uso de clases, tipos enumerados, variables, constantes nombradas, y otras entidades similares., Uso de estructuras de control., Manejo de las condiciones de error – tanto las excepciones y errores previstos (como el ingreso de datos incorrectos, por ejemplo)., Ajuste (tuning) del código.	La evaluación de código o reutilización de prueba.	Prueba de primer desarrollo., Depuración.

2/15/2012 4:26:41	Siempre.	Lenguajes de configuración (como los archivos de configuración de un sistema operativo)., Lenguajes de programación.	Si.	Lingüístico., Formal.	Técnicas para crear código entendible., Manejo de las condiciones de error – tanto las excepciones y errores previstos (como el ingreso de datos incorrectos, por ejemplo)., Prevención de las violaciones de seguridad a nivel de código (saturaciones del búfer o los desbordamientos del índice de matriz, por ejemplo)., Uso de recursos mediante el uso de los mecanismos de exclusión y de la disciplina en el acceso a los recursos reutilizables en serie (incluyendo hilos y bloqueos de bases de datos)., Documentación del código.	La selección de unidades reutilizables, bases de datos, procedimientos de prueba, o datos de prueba., La presentación de la información sobre la reutilización de código nuevo, los procedimientos de prueba, o datos de prueba.	Pruebas de unidad y de integración., Prueba de primer desarrollo., Depuración.

2/15/2012 9:20:54	Ocasionalmente.	Lenguajes de programación.	No.		Uso de estructuras de control.	La selección de unidades reutilizables, bases de datos, procedimientos de prueba, o datos de prueba.	Pruebas de unidad y de integración.

2/15/2012 21:43:27	Frecuentemente.	Lenguajes de programación.	Si.	Lingüístico., Visual.	El uso de clases, tipos enumerados, variables, constantes nombradas, y otras entidades similares., Manejo de las condiciones de error – tanto las excepciones y errores previstos (como el ingreso de datos incorrectos, por ejemplo)., Uso de recursos mediante el uso de los mecanismos de exclusión y de la disciplina en el acceso a los recursos reutilizables en serie (incluyendo hilos y bloqueos de bases de datos)., Organización del código fuente (en las declaraciones, las rutinas, clases, paquetes, u otras estructuras)., Documentación del código.	La evaluación de código o reutilización de prueba.	Pruebas de unidad y de integración., Prueba de primer desarrollo., Depuración., Revisiones técnicas.

2/16/2012 21:08:16	Frecuentemente.	Lenguajes de programación.	No.	El uso de clases, tipos enumerados, variables, constantes nombradas, y otras entidades similares., Manejo de las condiciones de error – tanto las excepciones y errores previstos (como el ingreso de datos incorrectos, por ejemplo)., Uso de recursos mediante el uso de los mecanismos de exclusión y de la disciplina en el acceso a los recursos reutilizables en serie (incluyendo hilos y bloqueos de bases de datos).	La evaluación de código o reutilización de prueba.	Pruebas de unidad y de integración., Código de paso a paso.

2/27/2012 19:57:10	Frecuentemente.	Lenguajes de configuración (como los archivos de configuración de un sistema operativo)., Lenguajes de kit de herramientas (que sirven para elaborar código a partir de un código reutilizable)., Lenguajes de programación.	Si.	Formal., Visual.	Técnicas para crear código entendible., El uso de clases, tipos enumerados, variables, constantes nombradas, y otras entidades similares., Uso de estructuras de control., Manejo de las condiciones de error – tanto las excepciones y errores previstos (como el ingreso de datos incorrectos, por ejemplo)., Organización del código fuente (en las declaraciones, las rutinas, clases, paquetes, u otras estructuras)., Documentación del código., Ajuste (tuning) del código.	La selección de unidades reutilizables, bases de datos, procedimientos de prueba, o datos de prueba., La evaluación de código o reutilización de prueba., La presentación de la información sobre la reutilización de código nuevo, los procedimientos de prueba, o datos de prueba.	Prueba de primer desarrollo., Código de paso a paso., Depuración.
2/27/2012 20:04:27	Ocasionalmente.	Lenguajes de kit de herramientas (que sirven para elaborar código a partir de un código reutilizable).	Si.	Visual.	Técnicas para crear código entendible., Uso de estructuras de control., Documentación del código.	La selección de unidades reutilizables, bases de datos, procedimientos de prueba, o datos de prueba.	Pruebas de unidad y de integración., Depuración.

2/28/2012 8:32:35	Ocasionalmente.	Lenguajes de kit de herramientas (que sirven para elaborar código a partir de un código reutilizable)., Lenguajes de programación.	Si.	Lingüístico., Formal.	El uso de clases, tipos enumerados, variables, constantes nombradas, y otras entidades similares., Organización del código fuente (en las declaraciones, las rutinas, clases, paquetes, u otras estructuras)., Ajuste (tuning) del código.	La selección de unidades reutilizables, bases de datos, procedimientos de prueba, o datos de prueba.	Pruebas de unidad y de integración., Revisiones técnicas.
2/28/2012 9:19:17	Frecuentemente.	Lenguajes de programación.	Si.	Lingüístico.	El uso de clases, tipos enumerados, variables, constantes nombradas, y otras entidades similares., Uso de estructuras de control., Organización del código fuente (en las declaraciones, las rutinas, clases, paquetes, u otras estructuras).	La evaluación de código o reutilización de prueba., La presentación de la información sobre la reutilización de código nuevo, los procedimientos de prueba, o datos de prueba.	Pruebas de unidad y de integración.

2/29/2012 15:06:40	Frecuentemente.	Lenguajes de programación.	Si.	Formal.	Técnicas para crear código entendible., El uso de clases, tipos enumerados, variables, constantes nombradas, y otras entidades similares., Uso de estructuras de control., Documentación del código., Ajuste (tuning) del código.	La selección de unidades reutilizables, bases de datos, procedimientos de prueba, o datos de prueba.	Pruebas de unidad y de integración., Revisiones técnicas.
3/2/2012 17:36:20	Ocasionalmente.	Lenguajes de programación.	Si.	Formal.	El uso de clases, tipos enumerados, variables, constantes nombradas, y otras entidades similares., Uso de estructuras de control., Manejo de las condiciones de error – tanto las excepciones y errores previstos (como el ingreso de datos incorrectos, por ejemplo)., Organización del código fuente (en las declaraciones, las rutinas, clases, paquetes, u otras estructuras)., Documentación del código.	La selección de unidades reutilizables, bases de datos, procedimientos de prueba, o datos de prueba.	Prueba de primer desarrollo.

3/5/2012 14:44:47	Frecuentemente.	Lenguajes de configuración (como los archivos de configuración de un sistema operativo)., Lenguajes de kit de herramientas (que sirven para elaborar código a partir de un código reutilizable)., Lenguajes de programación.	Si.	Formal.	Técnicas para crear código entendible., El uso de clases, tipos enumerados, variables, constantes nombradas, y otras entidades similares., Manejo de las condiciones de error – tanto las excepciones y errores previstos (como el ingreso de datos incorrectos, por ejemplo)., Prevención de las violaciones de seguridad a nivel de código (saturaciones del búfer o los desbordamientos del índice de matriz, por ejemplo)., Organización del código fuente (en las declaraciones, las rutinas, clases, paquetes, u otras estructuras)., Ajuste (tuning) del código.	La selección de unidades reutilizables, bases de datos, procedimientos de prueba, o datos de prueba., La evaluación de código o reutilización de prueba., La presentación de la información sobre la reutilización de código nuevo, los procedimientos de prueba, o datos de prueba.	Pruebas de unidad y de integración., Prueba de primer desarrollo., Código de paso a paso., Depuración., Análisis estático.

3/6/2012 10:09:57	Frecuentemente.	Lenguajes de kit de herramientas (que sirven para elaborar código a partir de un código reutilizable)., Lenguajes de programación.	Si.	Formal., Visual.	Técnicas para crear código entendible., El uso de clases, tipos enumerados, variables, constantes nombradas, y otras entidades similares., Uso de estructuras de control., Manejo de las condiciones de error – tanto las excepciones y errores previstos (como el ingreso de datos incorrectos, por ejemplo)., Prevención de las violaciones de seguridad a nivel de código (saturaciones del búfer o los desbordamientos del índice de matriz, por ejemplo)., Documentación del código.	La selección de unidades reutilizables, bases de datos, procedimientos de prueba, o datos de prueba.	Pruebas de unidad y de integración., Depuración., Revisiones técnicas.
3/8/2012 13:55:47	Frecuentemente.	Lenguajes de programación.	Si.	Lingüístico.	Técnicas para crear código entendible., El uso de clases, tipos enumerados, variables, constantes nombradas, y otras entidades similares., Uso de estructuras de control., Manejo de las condiciones	La selección de unidades reutilizables, bases de datos, procedimientos de prueba, o datos de prueba.	Prueba de primer desarrollo., Uso de afirmaciones., Depuración., Revisiones técnicas., Análisis estático.

de error – tanto las excepciones y errores previstos (como el ingreso de datos incorrectos, por ejemplo)., Prevención de las violaciones de seguridad a nivel de código (saturaciones del búfer o los desbordamientos del índice de matriz, por ejemplo)., Uso de recursos mediante el uso de los mecanismos de exclusión y de la disciplina en el acceso a los recursos reutilizables en serie (incluyendo hilos y bloqueos de bases de datos)., Organización del código fuente (en las declaraciones, las rutinas, clases, paquetes, u otras estructuras)., Documentación del código.

3/8/2012 16:40:15	Frecuentemente.	Lenguajes de kit de herramientas (que sirven para elaborar código a partir de un código reutilizable)., Lenguajes de programación.	Si.	Lingüístico., Visual.	Técnicas para crear código entendible., El uso de clases, tipos enumerados, variables, constantes nombradas, y otras entidades similares., Organización del código fuente (en las declaraciones, las rutinas, clases, paquetes, u otras estructuras)., Documentación del código., Ajuste (tuning) del código.	La selección de unidades reutilizables, bases de datos, procedimientos de prueba, o datos de prueba., La evaluación de código o reutilización de prueba.	Código de paso a paso., Depuración., Revisiones técnicas., Análisis estático.
3/11/2012 15:51:05	Ocasionalmente.	Lenguajes de programación.	No.		Ajuste (tuning) del código.	La evaluación de código o reutilización de prueba.	Prueba de primer desarrollo.

3/13/2012 11:38:31	Frecuentemente.	Lenguajes de configuración (como los archivos de configuración de un sistema operativo)., Lenguajes de kit de herramientas (que sirven para elaborar código a partir de un código reutilizable).	Si.	Visual.	El uso de clases, tipos enumerados, variables, constantes nombradas, y otras entidades similares., Manejo de las condiciones de error – tanto las excepciones y errores previstos (como el ingreso de datos incorrectos, por ejemplo)., Prevención de las violaciones de seguridad a nivel de código (saturaciones del búfer o los desbordamientos del índice de matriz, por ejemplo)., Documentación del código.	La selección de unidades reutilizables, bases de datos, procedimientos de prueba, o datos de prueba., La evaluación de código o reutilización de prueba.	Pruebas de unidad y de integración., Uso de afirmaciones.

3/13/2012 11:46:13	Siempre.	Lenguajes de configuración (como los archivos de configuración de un sistema operativo)., Lenguajes de kit de herramientas (que sirven para elaborar código a partir de un código reutilizable).	Si.	Lingüístico., Visual.	Técnicas para crear código entendible., Uso de estructuras de control., Uso de recursos mediante el uso de los mecanismos de exclusión y de la disciplina en el acceso a los recursos reutilizables en serie (incluyendo hilos y bloqueos de bases de datos).	La evaluación de código o reutilización de prueba., La presentación de la información sobre la reutilización de código nuevo, los procedimientos de prueba, o datos de prueba.	Prueba de primer desarrollo., Código de paso a paso., Uso de afirmaciones.
3/13/2012 11:48:31	Ocasionalmente.	Lenguajes de kit de herramientas (que sirven para elaborar código a partir de un código reutilizable)., Lenguajes de programación.	Si.	Lingüístico.	Técnicas para crear código entendible., Prevención de las violaciones de seguridad a nivel de código (saturaciones del búfer o los desbordamientos del índice de matriz, por ejemplo).	La selección de unidades reutilizables, bases de datos, procedimientos de prueba, o datos de prueba.	Pruebas de unidad y de integración., Uso de afirmaciones.

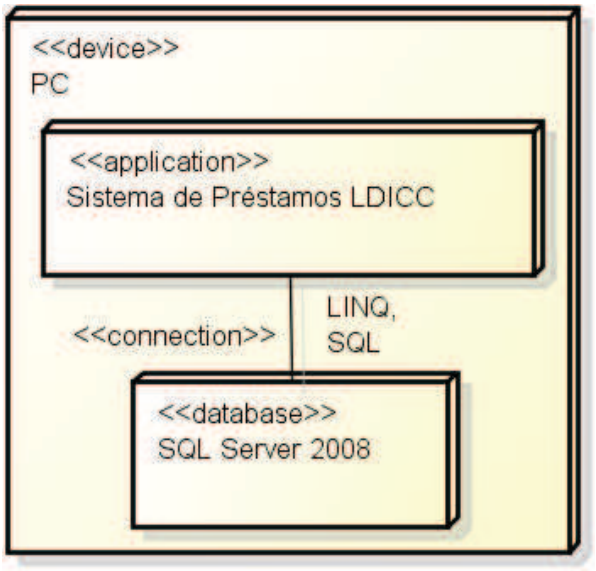
Marca temporal	20. ¿Qué aspectos usted (o el equipo de desarrollo) usan para la integración de componentes en la implementación durante el desarrollo de un sistema (puede escoger más de uno)?
2/14/2012 23:44:50	Creación de estructuras para apoyar las versiones no liberadas de la aplicación., Determinación de puntos en el proyecto en el que versiones provisionales del software se ponen a prueba.
2/15/2012 0:12:46	Planificación de la secuencia en la que los componentes se integrarán., Creación de estructuras para apoyar las versiones no liberadas de la aplicación.
2/15/2012 4:26:41	Planificación de la secuencia en la que los componentes se integrarán., Determinar el grado de las pruebas y la calidad del trabajo realizado en los componentes antes de su integración., Determinación de puntos en el proyecto en el que versiones provisionales del software se ponen a prueba.

2/15/2012 9:20:54	Determinación de puntos en el proyecto en el que versiones provisionales del software se ponen a prueba.
2/15/2012 21:43:27	Planificación de la secuencia en la que los componentes se integrarán.
2/16/2012 21:08:16	Creación de estructuras para apoyar las versiones no liberadas de la aplicación.
2/27/2012 19:57:10	Creación de estructuras para apoyar las versiones no liberadas de la aplicación., Determinar el grado de las pruebas y la calidad del trabajo realizado en los componentes antes de su integración.
2/27/2012 20:04:27	Planificación de la secuencia en la que los componentes se integrarán.
2/28/2012 8:32:35	Planificación de la secuencia en la que los componentes se integrarán.
2/28/2012 9:19:17	Planificación de la secuencia en la que los componentes se integrarán., Determinación de puntos en el proyecto en el que versiones provisionales del software se ponen a prueba.
2/29/2012 15:06:40	Planificación de la secuencia en la que los componentes se integrarán.
3/2/2012 17:36:20	Planificación de la secuencia en la que los componentes se integrarán.
3/5/2012 14:44:47	Planificación de la secuencia en la que los componentes se integrarán., Determinar el grado de las pruebas y la calidad del trabajo realizado en los componentes antes de su integración., Determinación de puntos en el proyecto en el que versiones provisionales del software se ponen a prueba.
3/6/2012 10:09:57	Planificación de la secuencia en la que los componentes se integrarán., Determinación de puntos en el proyecto en el que versiones provisionales del software se ponen a prueba.
3/8/2012 13:55:47	Determinar el grado de las pruebas y la calidad del trabajo realizado en los componentes antes de su integración., Determinación de puntos en el proyecto en el que versiones provisionales del software se ponen a prueba.
3/8/2012 16:40:15	Planificación de la secuencia en la que los componentes se integrarán.
3/11/2012 15:51:05	Planificación de la secuencia en la que los componentes se integrarán.
3/13/2012 11:38:31	Planificación de la secuencia en la que los componentes se integrarán., Determinar el grado de las pruebas y la calidad del trabajo realizado en los componentes antes de su integración.
3/13/2012 11:46:13	Planificación de la secuencia en la que los componentes se integrarán., Creación de estructuras para apoyar las versiones no liberadas de la aplicación.
3/13/2012 11:48:31	Planificación de la secuencia en la que los componentes se integrarán., Creación de estructuras para apoyar las versiones no liberadas de la aplicación.

ANEXO B

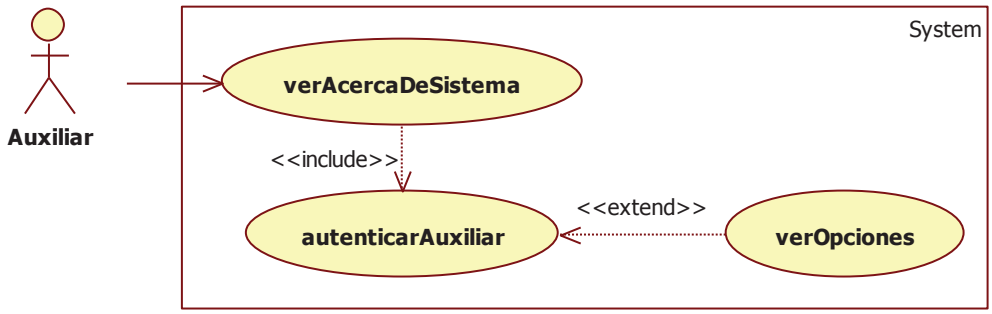
Diseño del Sistema

SISTEMA SOFTWARE.

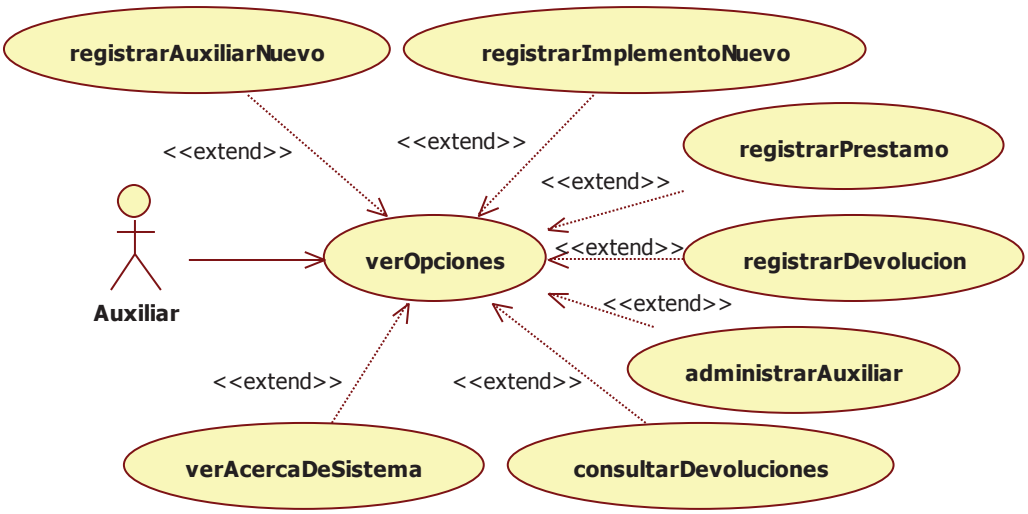


LOGICA DE NEGOCIO.

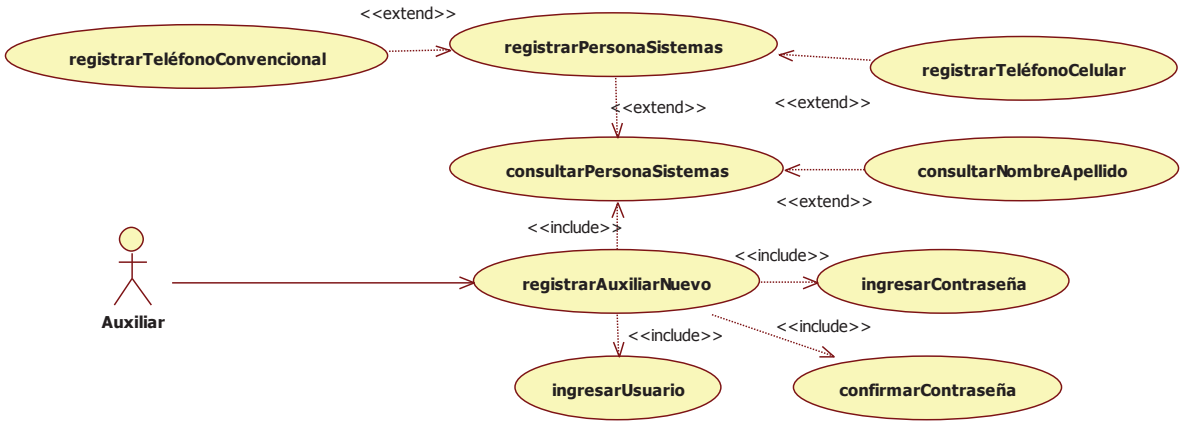
Ingreso al Sistema



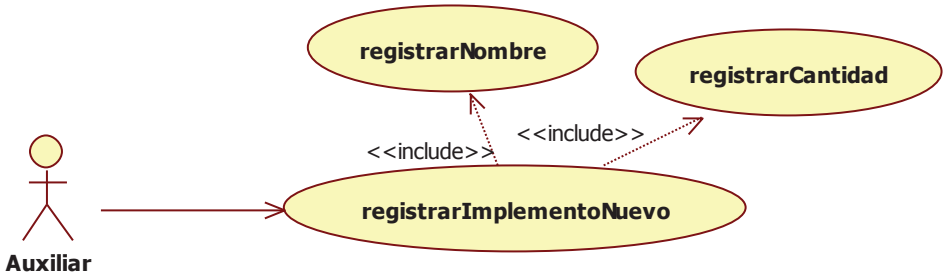
Menú Principal



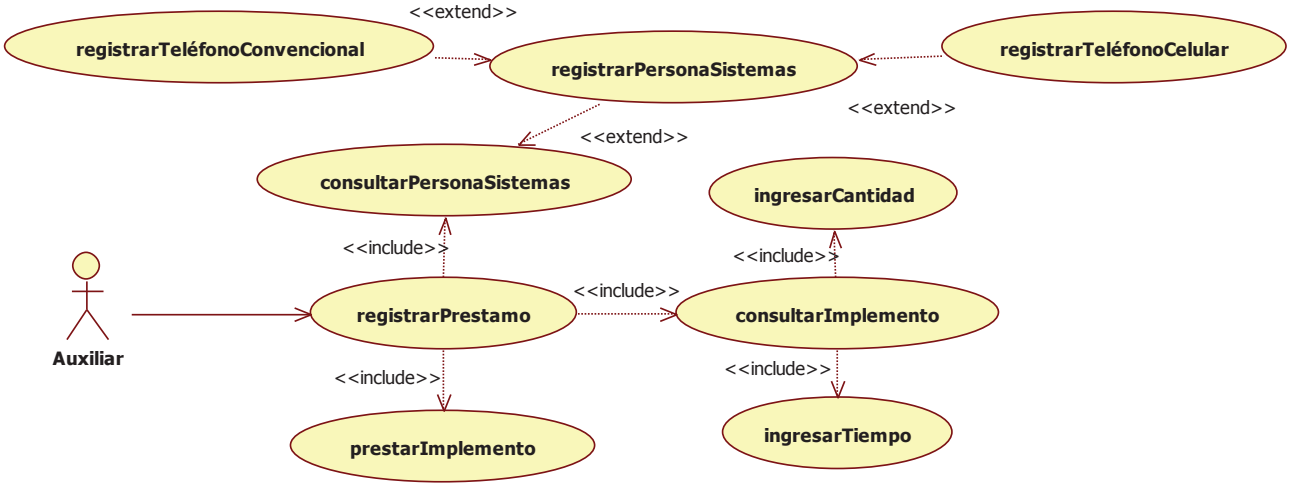
Registrar Auxiliar Nuevo.



Registrar Implemento Nuevo.

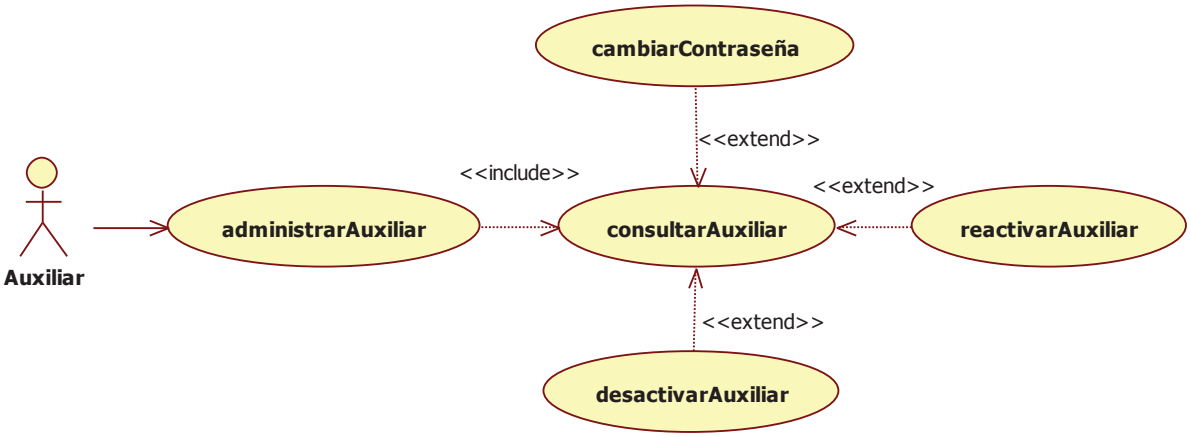


Registrar Préstamo.

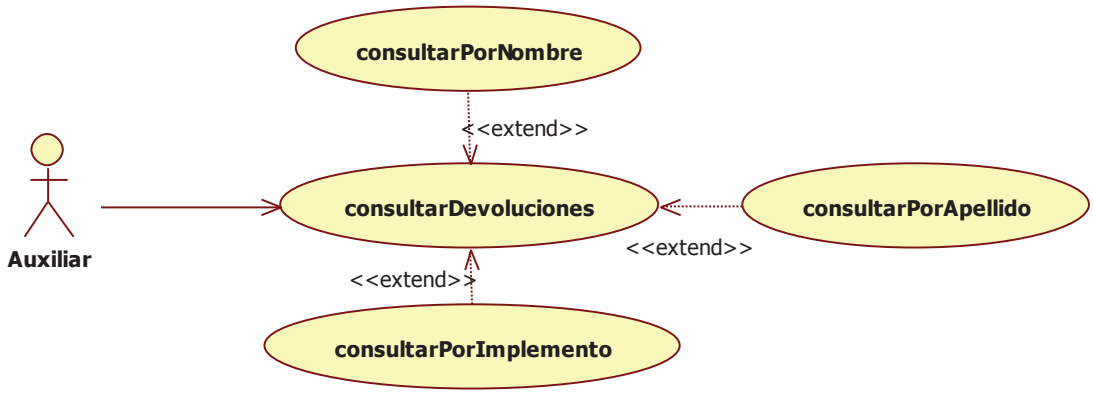


Registrar Devolución.

Administrar Auxiliar.



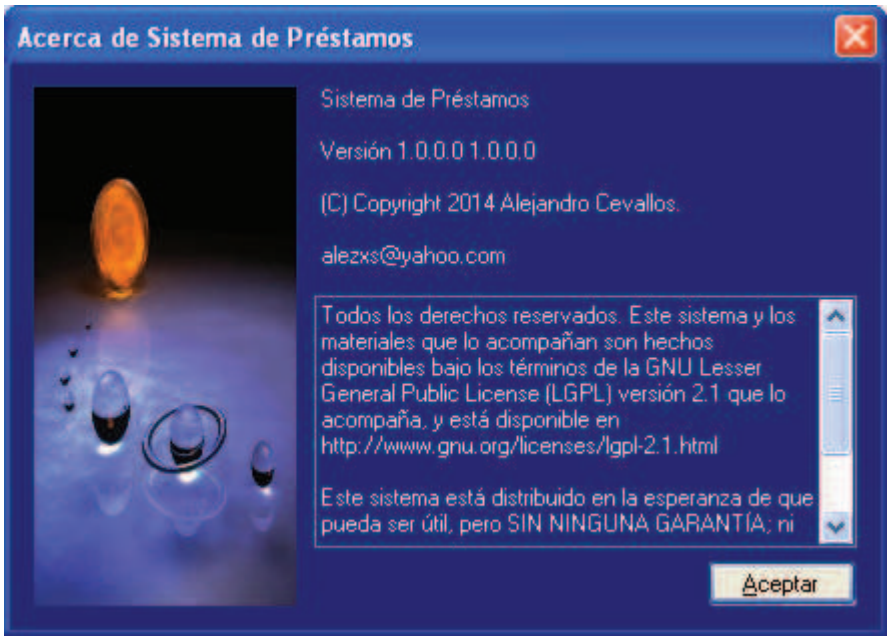
Consultar Devoluciones.



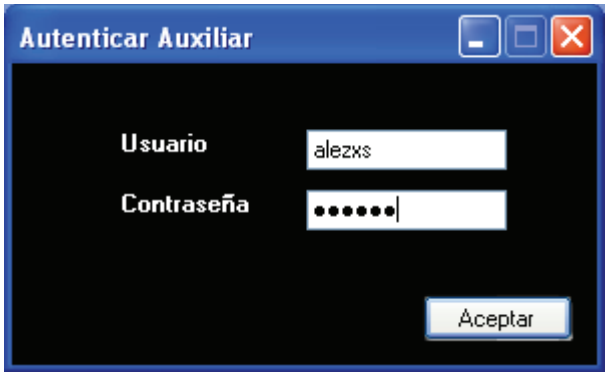
INTERFACES DE USUARIO.

Principales.

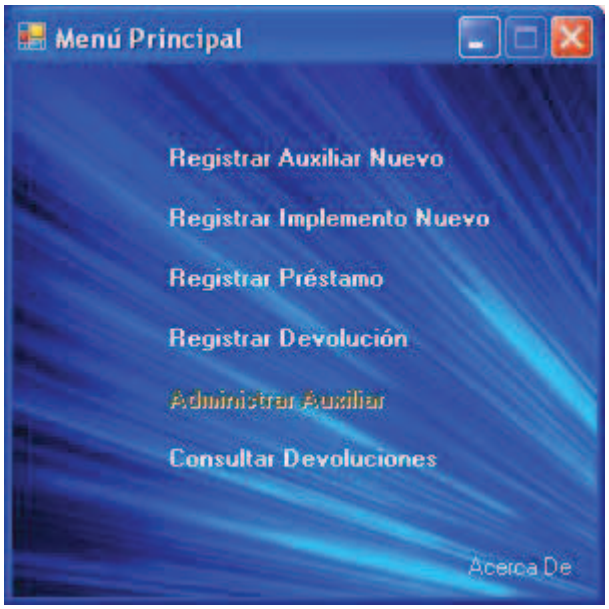
- **Acerca del Sistema de Préstamos.**



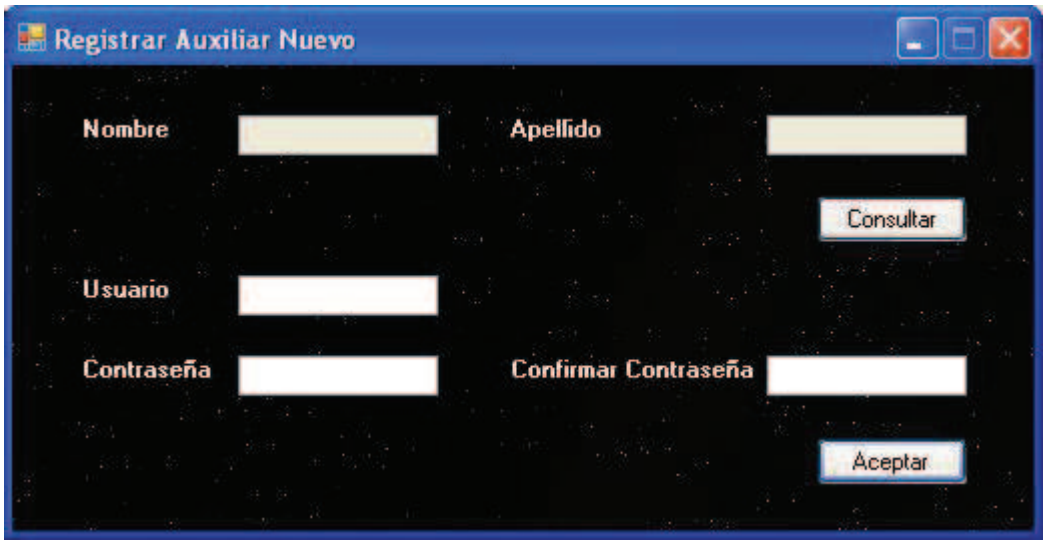
- **Autenticar Auxiliar.**



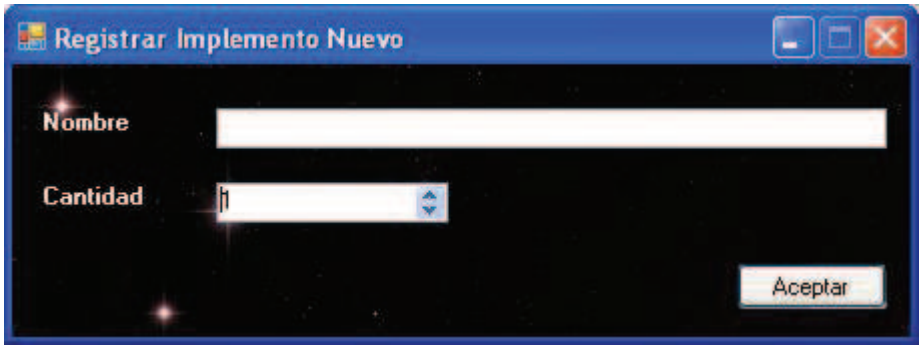
- **Menú Principal.**



- **Registrar Auxiliar Nuevo.**

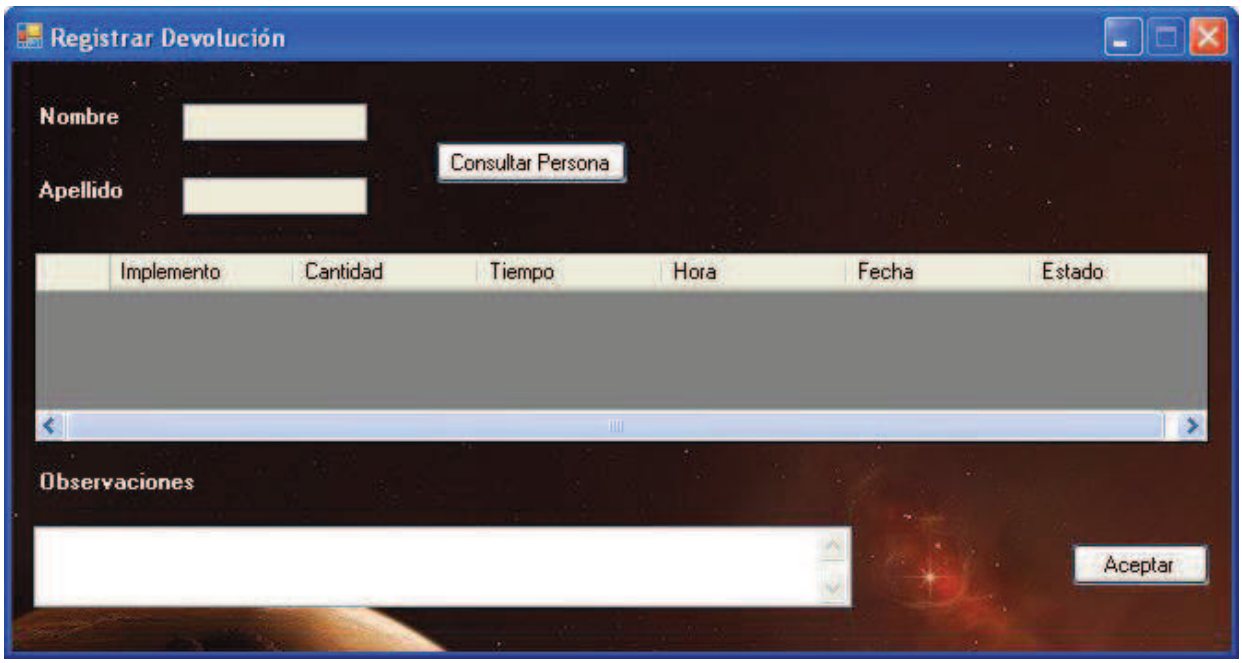


- **Registrar Implemento Nuevo.**



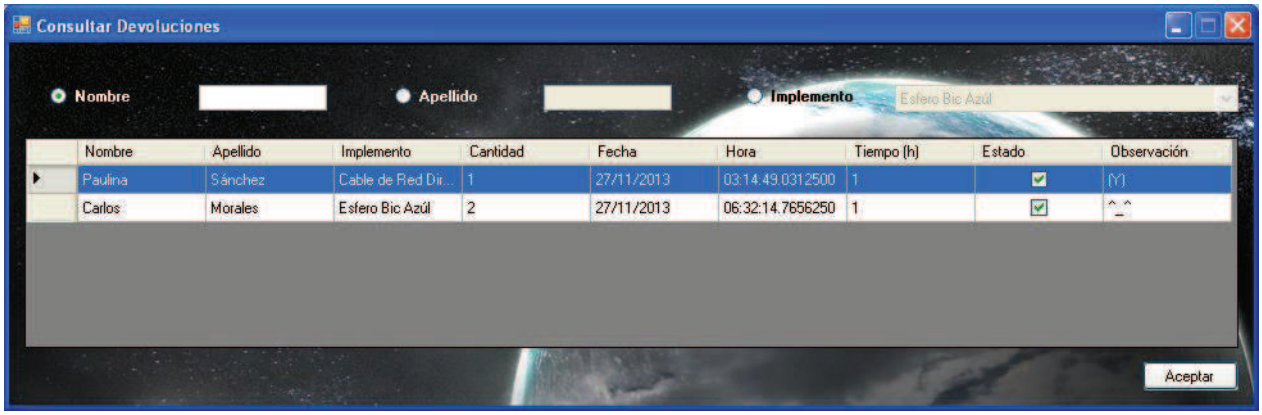
- **Registrar Préstamo.**

- **Registrar Devolución.**



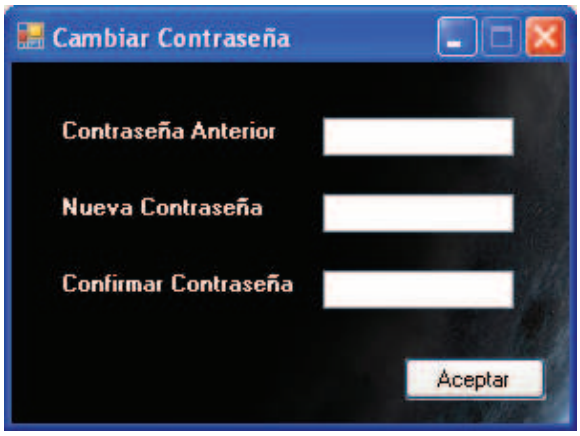
- **Administrar Auxiliar.**

- **Consultar Devoluciones.**

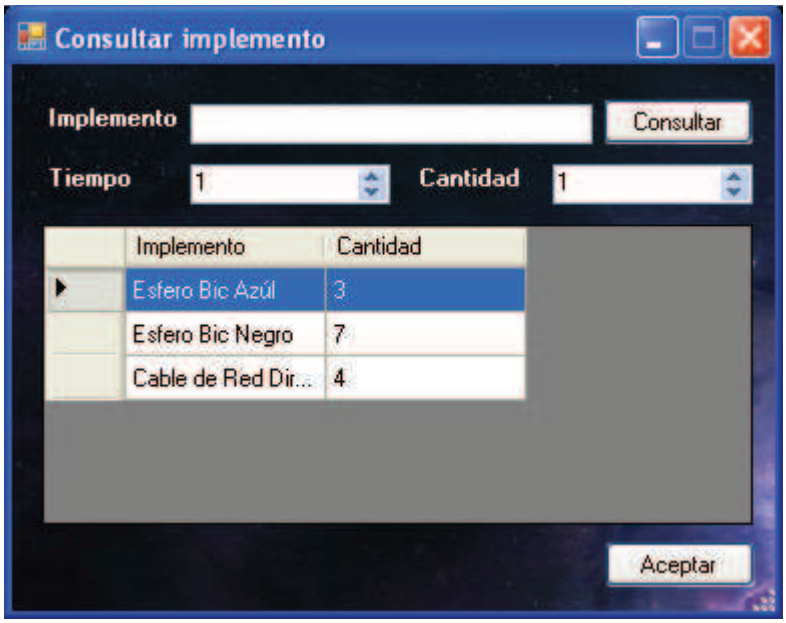


Secundarias.

- **Auxiliar.**

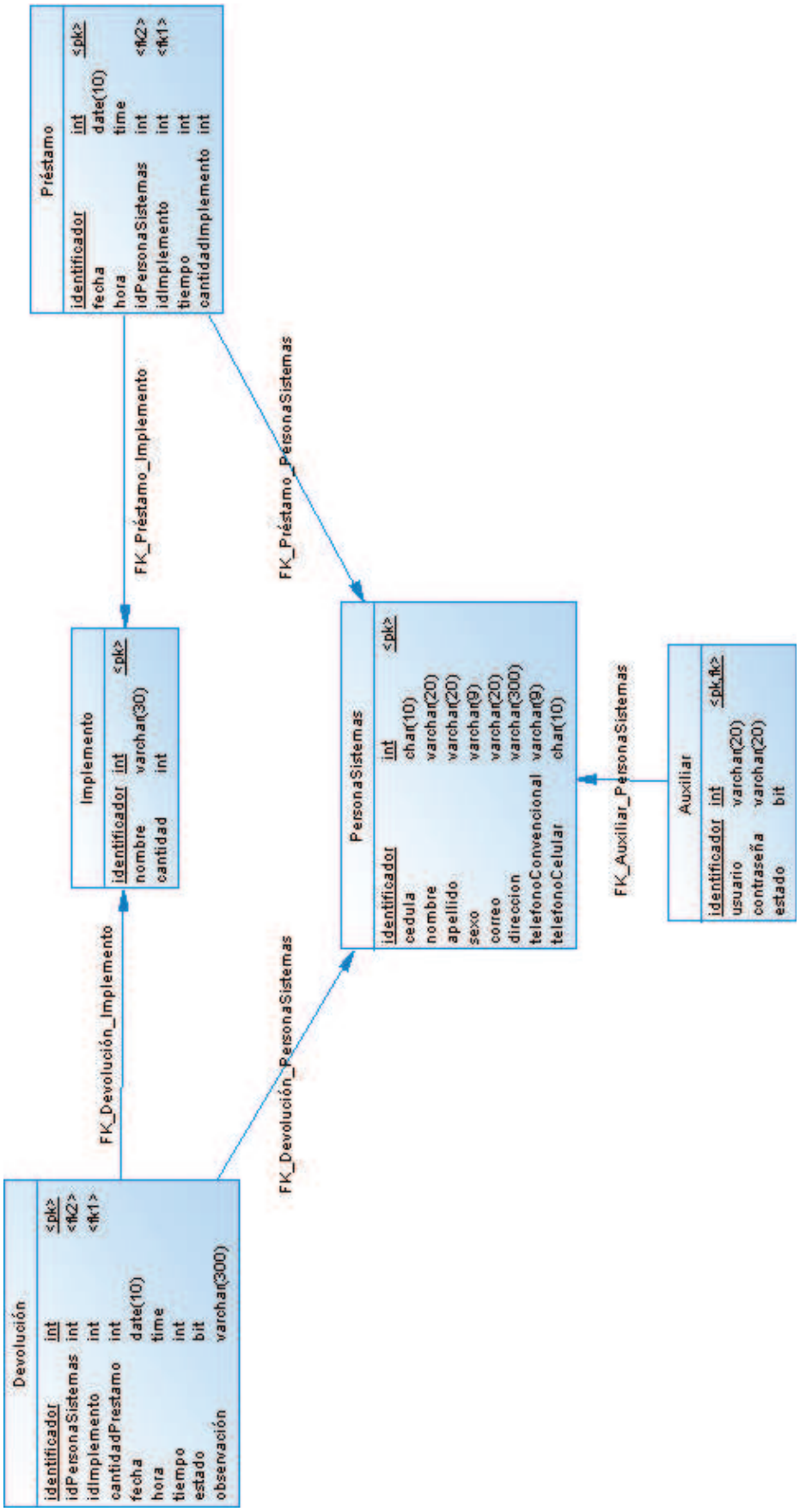


- **Implemento.**



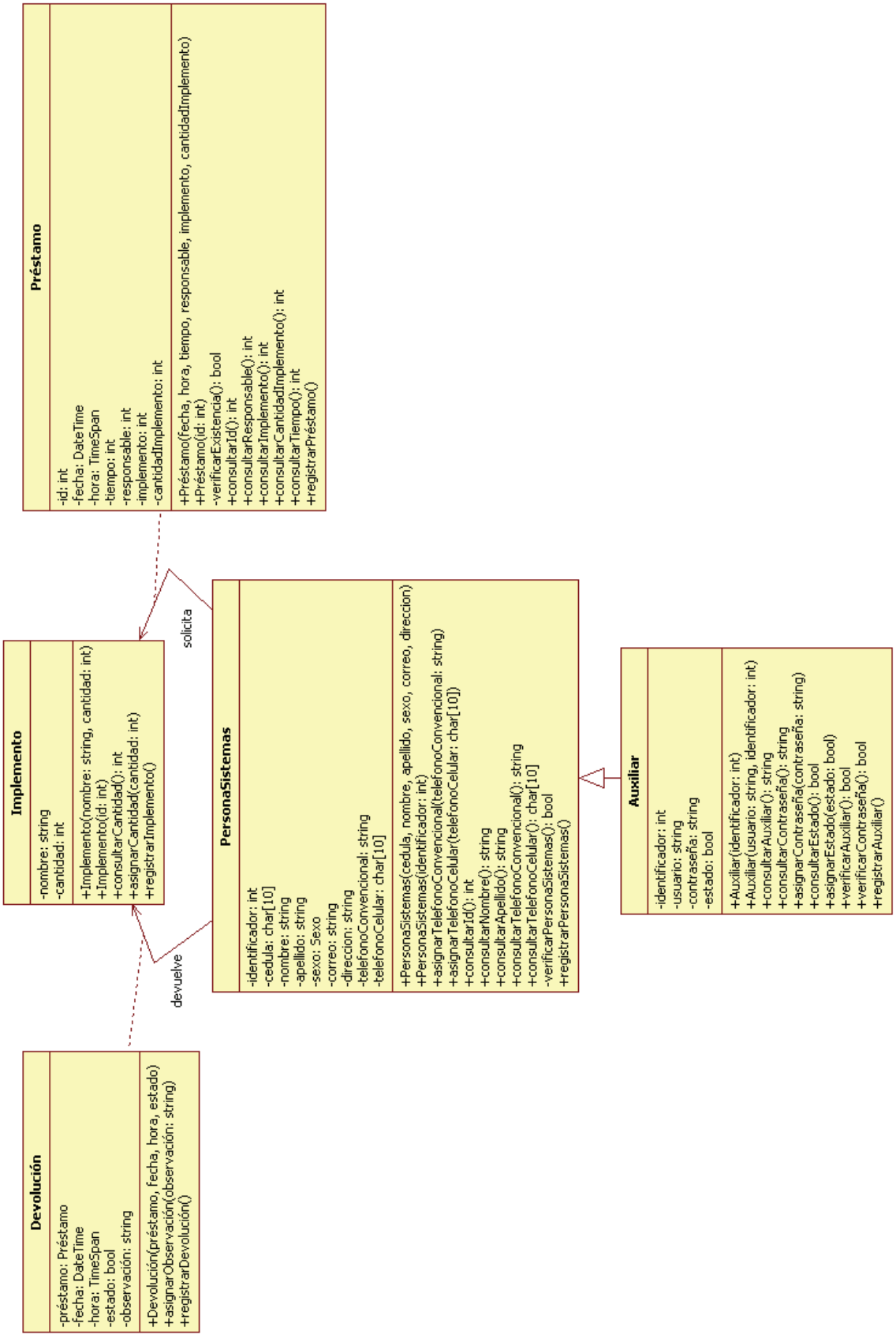
- **Persona Sistemas.**

BASE DE DATOS.

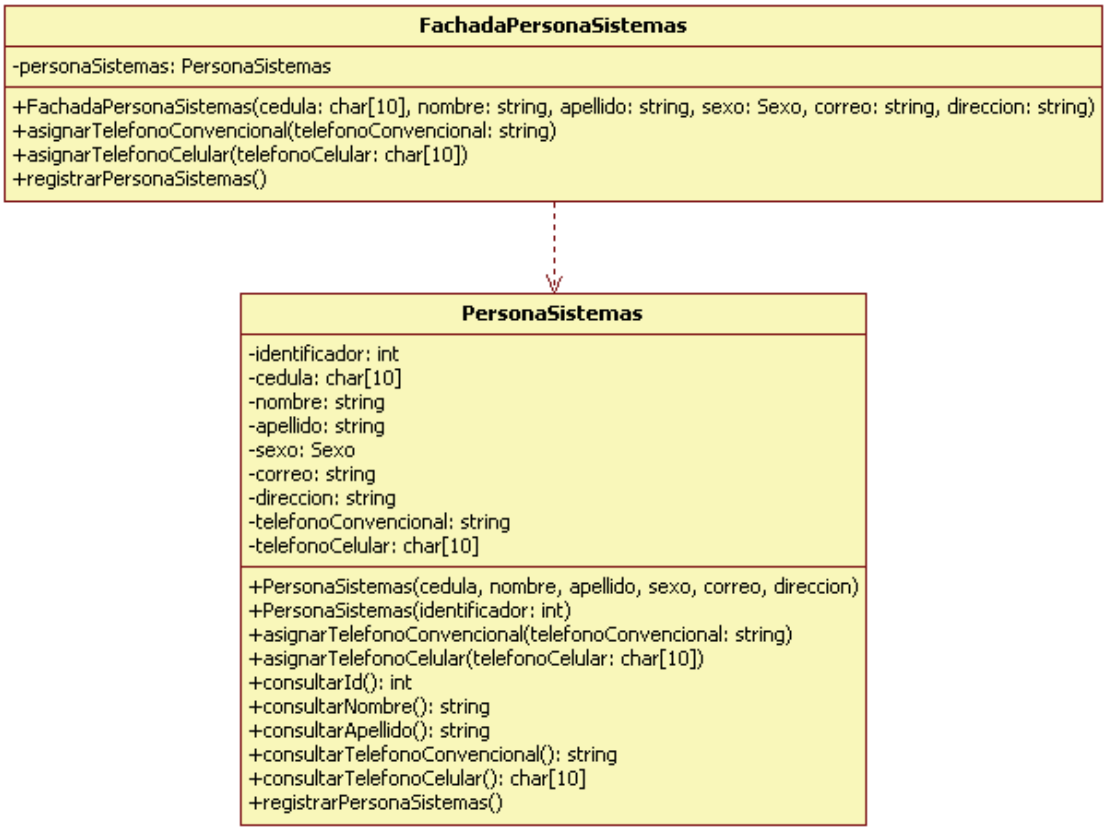


DIAGRAMAS DE CLASES.

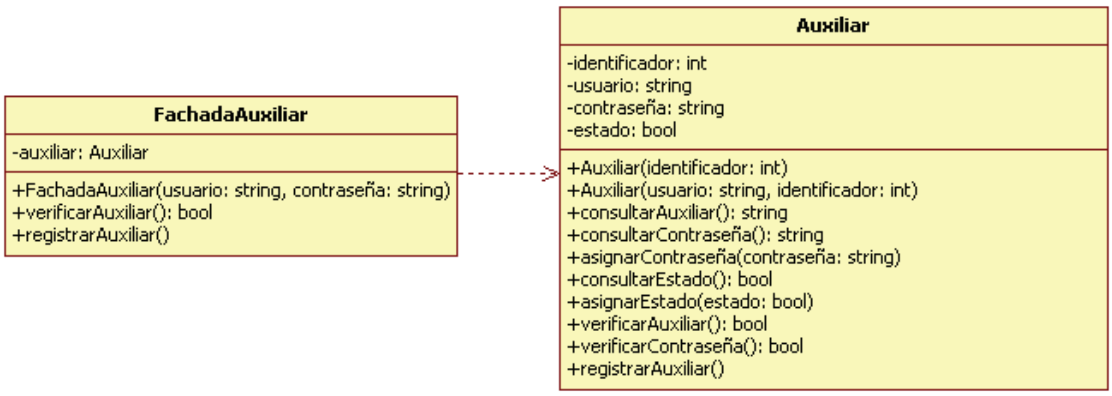
Principal.



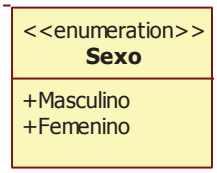
Patrón Fachada de Persona Sistemas.



Patrón Fachada de Auxiliar.



Enumeración Sexo.



ANEXO C

Diseño Detallado del Sistema

DISEÑO GENERAL DE LAS CLASES.

Clase: PersonaSistemas.

Responsabilidades Específicas.

- Esta clase se encarga de almacenar un registro de una persona de la facultad, sea quien fuere.
- Con esta clase se puede relacionar a cualquier persona de sistemas con el sistema mismo cómo usuario y cómo persona que participa del préstamo de unos de los implementos del laboratorio.

Secretos.

- Atributos de la clase.

Abstracción Capturada por la Clase.

Cédula, nombre, apellido, sexo, correo, dirección, telefonoConvencional y telefonoCelular.

Clase Principal:

Ninguna.

Clases Herederas:

Auxiliar.

Métodos Públicos Clave:

- PersonaSistemas.
- consultarId.
- consultarNombre.
- consultarApellido.
- registrarPersonaSistemas.

Clase: Auxiliar.

Responsabilidades Específicas.

- Esta clase se encarga de almacenar a los auxiliares que usan el sistema.
- Consulta los auxiliares con el fin de autenticarlos o de modificar su contraseña.
- Cuando un auxiliar deja de ejercer sus funciones, esta clase inactiva el usuario que lo representa.

Secretos.

- Los atributos de la clase.
- La manera en la que la clase almacena los datos en el sistema.
- La manera en que el auxiliar relaciona con sus datos de su clase que la hereda.
- La manera en la que desactiva a los auxiliares.
- La manera en la que se cambia la contraseña de un auxiliar.

Abstracción Capturada por la Clase.

El usuario, la contraseña y el estado.

Clase Principal:

PersonaSistemas.

Clases Herederas:

Ninguna.

Métodos Públicos Clave:

- Auxiliar.
- verificarAuxiliar.
- verificarContraseña.
- registrarAuxiliar.

Clase: Implemento.

Responsabilidades Específicas.

- Se encarga de almacenar el implemento a la base de datos.
- Se encargar de cargar los datos del implemento.

Secretos.

- Los atributos de la clase.
- La manera en la que la clase almacena los datos en el sistema.

Abstracción Capturada por la Clase.

Nombre, cantidad.

Clase Principal:

Ninguna.

Clases Herederas:

Ninguna.

Métodos Públicos Clave:

- Implemento.
- consultarCantidad.
- registrarImplemento.

Clase: Préstamo.

Responsabilidades Específicas.

- Se encarga de almacenar los datos del préstamo a la base.
- Carga los datos almacenados de un préstamo para fines de una devolución.

Secretos.

- Los atributos de una clase.
- La manera en que la clase almacena un registro en la base de datos.

Abstracción Capturada por la Clase.

Id, fecha, hora, tiempo, responsable, implemento, cantidadImplemento.

Clase Principal:

Ninguna.

Clases Herederas:

Ninguna.

Métodos Públicos Clave:

- Préstamo.
- registrarPréstamo.
- verificarExistencia.
- consultarId.
- consultarResponsable.
- consultarImplemento.
- consultarCantidadImplemento.
- consultarTiempo.

Clase: Devolución.

Responsabilidades Específicas.

- Se encarga de almacenar un registro de devolución en la base de datos.

Secretos.

- Atributos de la clase.
- La manera en que la clase almacena un registro en la base de datos.

Abstracción Capturada por la Clase.

Préstamo, fecha, estado, observación.

Clase Principal:

Ninguna.

Clases Herederas:

Ninguna.

Métodos Públicos Clave:

- Devolución.
- registrarDevolución.

Clase: FachadaPersonaSistemas.

Responsabilidades Específicas.

- Se encarga de disminuir la complejidad de la clase PersonaSistemas.
- Enfoca el uso de la clase en el registro de la PersonaSistemas a la base de datos.

Secretos.

- Complejidad de la clase PersonaSistemas.

Abstracción Capturada por la Clase.

Cédula, nombre, apellido, sexo, correo, dirección, telefonoConvencional y telefonoCelular.

Clase Principal:

Ninguna.

Clases Herederas:

Ninguna.

Métodos Públicos Clave:

- FachadaPersonaSistemas.
- registrarPersonaSistemas.

Clase: FachadaAuxiliar.

Responsabilidades Específicas.

- Se encarga de disminuir la complejidad de la clase Auxiliar.
- Enfoca el uso de la clase en el registro del Auxiliar a la base de datos.

Secretos.

- Complejidad de la clase Auxiliar.

Abstracción Capturada por la Clase.

Usuario, contraseña.

Clase Principal:

Ninguna.

Clases Herederas:

Ninguna.

Métodos Públicos Clave:

- FachadaAuxiliar.
- verificarAuxiliar.
- registrarAuxiliar.

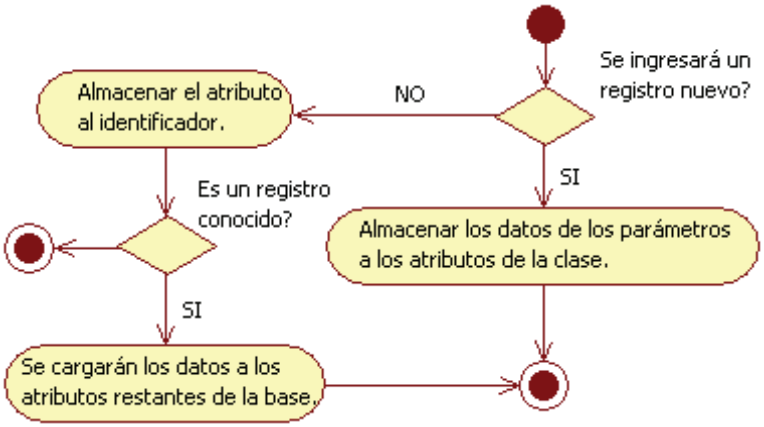
PSEUDOCÓDIGO DE CADA RUTINA.

PersonaSistemas.

Nombre de la Rutina: PersonaSistemas.

Clase: PersonaSistemas.

Diseño:



Prerrequisitos:

- En esta rutina se recogen todos los detalles obligatorios de un registro de la persona de sistemas.

Problema a Resolver:

a) Detalles.

- Recoge los datos obligatorios de una persona sistemas para una inserción.
- Carga el registro a la clase desde una consulta a base de datos.

b) Datos.

- La rutina esconderá el procedimiento para recoger los datos de la persona de sistemas.
- Se tendrá como entradas: el identificador, la cédula, el nombre, el apellido, el sexo, el correo y la dirección.
- No tendrá valor de retorno como salida.
- Las precondiciones son: la cédula es correctamente ingresada y tiene 10 caracteres, el sexo solo es ingresado como ‘Masculino’ o ‘Femenino’ y ningún campo es nulo.
- Habrá como garantía que la clase almacene los datos principales de una persona de sistemas.

Manejo de Errores.

Debe asegurarse que se cumplan las precondiciones descritas para evitar errores al momento de almacenar los resultados al sistema.

Posibilidad de Eficiencia.

N/A.

Funcionalidad Disponible en Bibliotecas Estándares.

La funcionalidad de LINQ para cuando se realice la consulta del registro.

Algoritmos Disponibles:

Se ha investigado de rutinas constructoras en algunos ejemplos de instanciación de objetos.

Pseudocódigo:

```
En esta rutina se recogen todos los detalles obligatorios de un
registro de la persona de sistemas.
Inicio PersonaSistemas (cedula, nombre, apellido, sexo, correo,
dirección)
   Asignar todos los valores de los parámetros.
Fin
```

```
En esta rutina se recoge el identificador para manipular los datos la
persona de sistemas.
Inicio PersonaSistemas (identificador)
   Asignar el valor del parámetro al atributo de la clase.
   Si se ha asignado un identificador de un dato conocido, entonces
      Se cargarán los datos a los atributos restantes de la base.
   Fin Si.
Fin
```

Datos a Manejarse:

- 2 enteros.
- 2 arreglos de caracteres.
- 8 cadenas de texto.
- 2 enumeraciones.

Nombre de la Rutina: asignarTelefonoConvencional.

Clase: PersonaSistemas.

Diseño:

Prerrequisitos:

- Se almacena el teléfono convencional después de instanciada la clase.

Problema a Resolver:

a) Detalles.

- Agrega un teléfono convencional si la persona lo posee.

b) Datos.

- La rutina esconderá la asignación del parámetro al atributo de la clase.
- Se tendrá como entrada el teléfono convencional (cadena de texto).
- No habrán salidas.
- La precondición es que el dato será correctamente ingresado.
- La poscondición será el dato ingresado al atributo de la clase.

Manejo de Errores.

Debido a que lo que se ingresa es una cadena se controlará desde el sistema del ingreso del teléfono.

Posibilidad de Eficiencia.

Se podría lanzar una excepción si el ingreso no fuese correcto.

Funcionalidad Disponible en Bibliotecas Estándares.

N/D.

Algoritmos Disponibles:

N/A.

Pseudocódigo:

```
Agrega un teléfono convencional si la persona lo posee.
Inicio asignarTelefonoConvencional (telefonoConvencional)
   Agregar teléfono convencional.
Fin
```

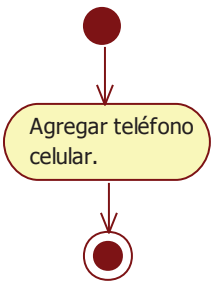
Datos a Manejarse:

- 2 cadenas de texto.

Nombre de la Rutina: asignarTelefonoCelular.

Clase: PersonaSistemas.

Diseño:



Prerrequisitos:

- Se almacena el teléfono celular después de instanciada la clase.

Problema a Resolver:

a) Detalles.

- Agrega un teléfono celular si la persona lo posee.

b) Datos.

- Se esconderá la asignación del parámetro al atributo de la clase.
- La entrada será el teléfono celular (arreglo de 10 caracteres).
- No tendrá salida.
- La precondición será el teléfono celular tendrá justo 10 caracteres.
- La poscondición será el teléfono almacenado en el atributo de la clase.

Manejo de Errores.

Es posible que no se ingrese un número y que se ingrese menos de 10 caracteres, tales cosas serán controladas desde la aplicación.

Posibilidad de Eficiencia.

Se podría lanzar una excepción si el ingreso no fuese correcto.

Funcionalidad Disponible en Bibliotecas Estándares.

N/D.

Algoritmos Disponibles:

N/A.

Pseudocódigo:

```
Agrega un teléfono celular si la persona lo posee.
Inicio asignarTelefonoCelular (telefonoCelular)
   Agregar teléfono celular.
Fin
```

Datos a Manejarse:

- 2 arreglos de caracteres.

Nombre de la Rutina: consultarId.

Clase: PersonaSistemas.

Diseño:

Prerrequisitos:

- Retorna el identificador, un atributo de la clase.

Problema a Resolver:

a) Detalles.

- Recupera el identificador de una Persona Sistemas específica.

b) Datos.

- La rutina esconderá el retorno del atributo.
- No tendrá ninguna entrada.
- Retornará el identificador.
- No hay precondiciones.
- No hay poscondiciones.

Manejo de Errores.

No hay posibilidad de error.

Posibilidad de Eficiencia.

N/A.

Funcionalidad Disponible en Bibliotecas Estándares.

N/D.

Algoritmos Disponibles:

N/A.

Pseudocódigo:

```
Recupera el identificador de una Persona Sistemas específica.
Inicio consultarId ()
   Retornar el identificador.
Fin
```

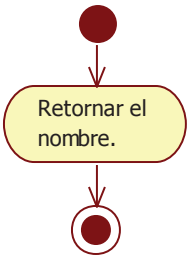
Datos a Manejarse:

- Un entero.

Nombre de la Rutina: consultarNombre.

Clase: PersonaSistemas.

Diseño:



Prerrequisitos:

- Retorna el nombre, un atributo de la clase.

Problema a Resolver:

a) Detalles.

- Recupera el nombre de una Persona Sistemas específica.

b) Datos.

- La rutina esconderá el retorno del atributo.
- No tendrá ninguna entrada.
- Retornará el nombre.
- No hay precondiciones.
- No hay poscondiciones.

Manejo de Errores.

No hay posibilidad de error.

Posibilidad de Eficiencia.

N/A.

Funcionalidad Disponible en Bibliotecas Estándares.

N/D.

Algoritmos Disponibles:

N/A.

Pseudocódigo:

```
Recupera el nombre de una Persona Sistemas específica.
```

```
Inicio consultarNombre ()
   Retornar el nombre.
Fin
```

Datos a Manejarse:

- Una cadena de texto.

Nombre de la Rutina: consultarApellido.

Clase: PersonaSistemas.

Diseño:

Prerrequisitos:

- Retornará el apellido, un atributo de la clase.

Problema a Resolver:

a) Detalles.

- Recupera el apellido de una Persona Sistemas específica.

b) Datos.

- La rutina esconderá el retorno del atributo.
- No tendrá ninguna entrada.
- Retornará el apellido.
- No hay precondiciones.
- No hay poscondiciones.

Manejo de Errores.

No hay posibilidad de error.

Posibilidad de Eficiencia.

N/A.

Funcionalidad Disponible en Bibliotecas Estándares.

N/D.

Algoritmos Disponibles:

N/A.

Pseudocódigo:

```
Recupera el apellido de una Persona Sistemas específica.
Inicio consultarApellido ()
   Retornar el apellido.
Fin
```

Datos a Manejarse:

- Una cadena de texto.

Nombre de la Rutina: consultarTelefonoConvencional.

Clase: PersonaSistemas.

Diseño:

Prerrequisitos:

- Retorna el teléfono convencional, un atributo de la clase.

Problema a Resolver:

a) Detalles.

- Recupera el teléfono convencional de una Persona Sistemas específica.

b) Datos.

- La rutina esconderá el retorno del atributo.

- No tendrá ninguna entrada.
- Retornará el teléfono convencional.
- No hay precondiciones.
- No hay poscondiciones.

Manejo de Errores.

No hay posibilidad de error.

Posibilidad de Eficiencia.

N/A.

Funcionalidad Disponible en Bibliotecas Estándares.

N/D.

Algoritmos Disponibles:

N/A.

Pseudocódigo:

```
Recupera el teléfono convencional de una Persona Sistemas específica.
Inicio consultarTelefonoConvencional ()
   Retornar el teléfono convencional.
Fin
```

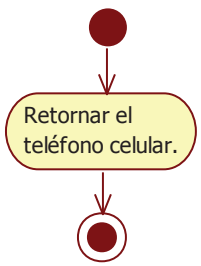
Datos a Manejarse:

- Una cadena de texto.

Nombre de la Rutina: consultarTelefonoCelular.

Clase: PersonaSistemas.

Diseño:



Prerrequisitos:

- Retorna el teléfono celular, un atributo de la clase.

Problema a Resolver:

a) Detalles.

- Recupera el teléfono celular de la clase instanciada.

b) Datos.

- La rutina esconderá el retorno del atributo.
- No tendrá ninguna entrada.
- Retornará el teléfono celular.
- No hay precondiciones.
- No hay poscondiciones.

Manejo de Errores.

No hay posibilidad de error.

Posibilidad de Eficiencia.

N/A.

Funcionalidad Disponible en Bibliotecas Estándares.

N/D.

Algoritmos Disponibles:

N/A.

Pseudocódigo:

```
Recupera el teléfono celular de la clase instanciada.
Inicio consultarTelefonoCelular ()
   Retornar el teléfono celular.
Fin
```

Datos a Manejarse:

- Un arreglo de caracteres.

Nombre de la Rutina: verificarPersonaSistemas.

Clase: PersonaSistemas.

Diseño:

Prerrequisitos:

- Verifica en la base de datos si se ha ingresado anteriormente a una Persona Sistemas.

Problema a Resolver:

a) Detalles.

- Verifica si la Persona Sistemas existe.
- Emite el resultado de su existencia dependiendo de la búsqueda.

b) Datos.

- La rutina esconderá el procedimiento para hallar su existencia en el sistema.
- No se tendrá parámetros de entrada.
- Se tendrá un valor de retorno booleano como salida.
- No existen precondiciones.
- El valor de retorno será garantía de la rutina que indicará el éxito o el fracaso de la rutina en hallar a la persona sistemas.

Manejo de Errores.

No existe probabilidad de error en esta rutina.

Posibilidad de Eficiencia.

Se realizará la búsqueda del nombre del usuario mediante una consulta a la base de datos, cosa que no se necesitará por el momento un algoritmo optimizado.

Funcionalidad Disponible en Bibliotecas Estándares.

Se utilizará una forma de conexión a base de datos llamada LINQ, con el fin de realizar la consulta a la base sin preocuparse por la conectividad a la base de datos.

Algoritmos Disponibles:

Cómo si es posible utilizar LINQ, entonces no será prescindible investigar de algoritmos.

Pseudocódigo:

```
Verifica si la Persona Sistemas existe y emite el resultado de su
existencia dependiendo de la búsqueda.
Inicio verificarPersonaSistemas ()
   Verificar la existencia de la persona ingresada.
   Retornar el resultado de la búsqueda.
Fin
```

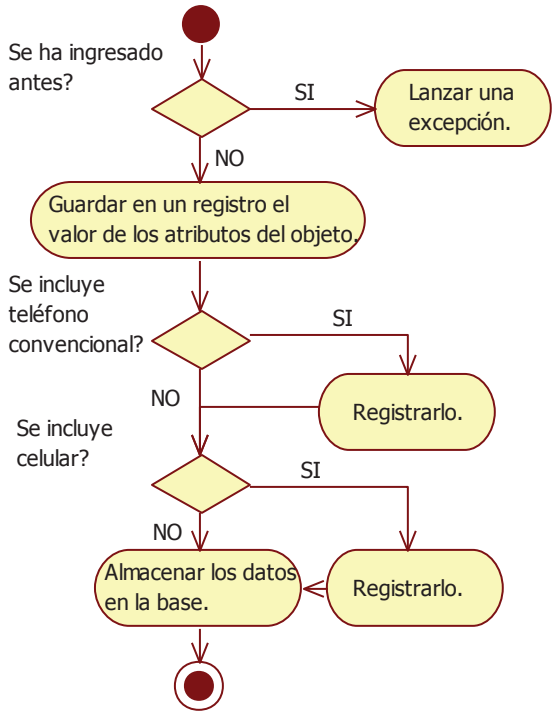
Datos a Manejarse:

- Un entero.
- Una cadena de texto.

Nombre de la Rutina: registrarPersonaSistemas.

Clase: PersonaSistemas.

Diseño:



Prerrequisitos:

- Verifica si existe un registro repetido (mediante otra rutina).
- Verifica si deben incluirse los teléfonos convencional y celular.
- Registra los datos en la base.

Problema a Resolver:

a) Detalles.

- Se encarga de almacenar en la base de datos el registro de la Persona Sistemas.

b) Datos.

- La rutina esconderá la forma como almacena los datos y los datos opcionales que recoge.
- No tiene ninguna entrada.
- No tiene ninguna salida.
- La precondición es que los datos son previamente ingresados.
- La poscondición es que los datos serán guardados en la base.

Manejo de Errores.

Existe la posibilidad de ingreso incorrecto, lo cuál puede ser controlado por medio de las rutinas de la clase que recogen los datos y de las restricciones de la base datos.

Posibilidad de Eficiencia.

N/A.

Funcionalidad Disponible en Bibliotecas Estándares.

Se usará las bibliotecas que contengan la conectividad a base de datos mediante LINQ para registrar los datos.

Algoritmos Disponibles:

N/A.

Pseudocódigo:

```
Se encarga de almacenar en la base de datos el registro de la Persona
Sistemas.
Inicio registrarPersonaSistemas ()
   Si se ha ingresado antes, entonces
     Lanzar una excepción y terminar la rutina.
   Fin Si.

   Guardar en un registro el valor de los atributos del objeto.

   Si se incluye un teléfono convencional, entonces
     Registrarlo.
   Fin Si.

   Si se incluye un teléfono celular, entonces
     Registrarlo.
   Fin Si.

   Almacenar los datos en la base.
Fin
```

Datos a Manejarse:

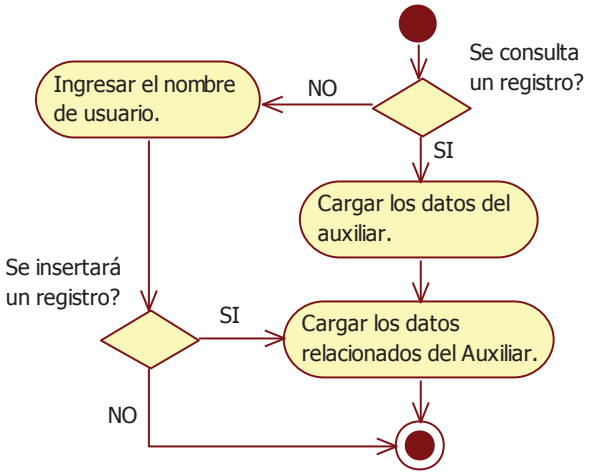
- 4 arreglos de caracteres.
- 11 cadenas de texto.
- 1 enumeración.

Auxiliar.

Nombre de la Rutina: Auxiliar.

Clase: Auxiliar.

Diseño:



Prerrequisitos:

- Carga los datos para una inserción o autenticación.
- Carga los datos para una consulta.

Problema a Resolver:

a) Detalles.

- Recoge los datos obligatorios de un auxiliar para una inserción.
- Carga el registro del auxiliar a la clase desde una consulta a base de datos.
- Recoge los datos obligatorios de un auxiliar para ser autenticado.

b) Datos.

- La rutina esconderá el procedimiento para recoger los datos del auxiliar.
- Se tendrá como entradas: el identificador y el usuario.
- No se tendrá salidas.
- El usuario es único.
- Los datos del auxiliar son cargados a la clase.

Manejo de Errores.

No existe posibilidad de errores.

Posibilidad de Eficiencia.

N/A.

Funcionalidad Disponible en Bibliotecas Estándares.

La funcionalidad de LINQ para cuando se realice la consulta del registro.

Algoritmos Disponibles:

Se ha investigado de rutinas constructoras en algunos ejemplos de instanciación de objetos.

Pseudocódigo:

```
Recoge los datos obligatorios de un auxiliar para una inserción o
autenticación.
Inicio Auxiliar (usuario, identificador)
   Ingresar el nombre de usuario.
   Si se ingresa un registro, entonces
      Cargar los datos relacionados al auxiliar.
   Fin Si.
Fin
```

```
Carga los datos para una consulta.
Inicio Auxiliar (identificador)
   Cargar los datos del auxiliar.
   Cargar los datos relacionados al auxiliar.
Fin
```

Datos a Manejarse:

- 2 enteros.
- 4 cadenas de texto.

Nombre de la Rutina: consultarAuxiliar.

Clase: Auxiliar.

Diseño:

Prerrequisitos:

- Retorna el nombre del auxiliar, un atributo de la clase.

Problema a Resolver:

a) Detalles.

- Recupera el usuario (nombre del auxiliar).

b) Datos.

- La rutina esconderá el retorno del atributo.
- No tendrá ninguna entrada.
- Retornará el usuario.
- No hay precondiciones.
- No hay poscondiciones.

Manejo de Errores.

No hay posibilidad de error.

Posibilidad de Eficiencia.

N/A.

Funcionalidad Disponible en Bibliotecas Estándares.

N/D.

Algoritmos Disponibles:

N/A.

Pseudocódigo:

```
Recupera el usuario (nombre del auxiliar).
Inicio consultarAuxiliar ()
   Retornar el auxiliar.
Fin
```

Datos a Manejarse:

- Una cadena de texto.

Nombre de la Rutina: consultarContraseña.

Clase: Auxiliar.

Diseño:

Prerrequisitos:

- Retorna la contraseña, un atributo de la clase.

Problema a Resolver:

a) Detalles.

- Recupera la contraseña del auxiliar.

b) Datos.

- La rutina esconderá el retorno del atributo.
- No tendrá ninguna entrada.
- Retornará la contraseña.
- No hay precondiciones.
- No hay poscondiciones.

Manejo de Errores.

No hay posibilidad de error.

Posibilidad de Eficiencia.

N/A.

Funcionalidad Disponible en Bibliotecas Estándares.

N/D.

Algoritmos Disponibles:

N/A.

Pseudocódigo:

```
Recupera la contraseña del auxiliar.
Inicio consultarContraseña ()
   Retornar la contraseña.
Fin
```

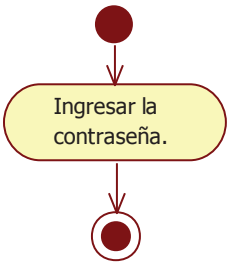
Datos a Manejarse:

- Una cadena de texto.

Nombre de la Rutina: asignarContraseña.

Clase: Auxiliar.

Diseño:



Prerrequisitos:

- Se almacena la contraseña después de instanciada la clase.

Problema a Resolver:

a) Detalles.

- Agrega la contraseña del auxiliar.

b) Datos.

- Se esconderá la asignación del parámetro al atributo de la clase.
- La entrada será la contraseña ingresada.
- No tendrá salida.
- La precondición será el usuario ya instanciado.
- La poscondición será la contraseña almacenada en el atributo de la clase.

Manejo de Errores.

No hay posibilidad de error.

Posibilidad de Eficiencia.

N/A.

Funcionalidad Disponible en Bibliotecas Estándares.

N/D.

Algoritmos Disponibles:

N/A.

Pseudocódigo:

```
Agrega la contraseña del auxiliar.
Inicio asignarContraseña (contraseña)
   Ingresar la contraseña.
Fin
```

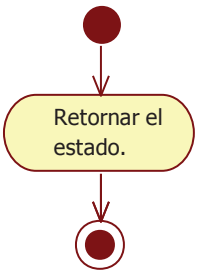
Datos a Manejarse:

- 2 cadenas de texto.

Nombre de la Rutina: consultarEstado.

Clase: Auxiliar.

Diseño:



Prerrequisitos:

- Retorna el estado, un atributo de la clase.

Problema a Resolver:

a) Detalles.

- Recupera el estado actual del usuario.

b) Datos.

- La rutina esconderá el retorno del atributo.
- No tendrá ninguna entrada.
- Retornará el estado.
- No hay precondiciones.
- No hay poscondiciones.

Manejo de Errores.

No hay posibilidad de error.

Posibilidad de Eficiencia.

N/A.

Funcionalidad Disponible en Bibliotecas Estándares.

N/D.

Algoritmos Disponibles:

N/A.

Pseudocódigo:

```
Recupera el estado actual del usuario.
Inicio consultarEstado ()
   Retornar el estado.
Fin
```

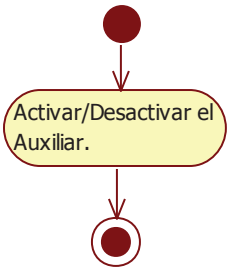
Datos a Manejarse:

- Un booleano.

Nombre de la Rutina: asignarEstado.

Clase: Auxiliar.

Diseño:



Prerrequisitos:

- Se almacena el estado después de instanciada la clase.

Problema a Resolver:

a) Detalles.

- Asigna el estado del auxiliar.

b) Datos.

- Se esconderá la asignación del parámetro al atributo de la clase.
- La entrada será el estado asignado.
- No tendrá salida.
- La precondición será el usuario ya instanciado.
- La poscondición será el estado almacenado en el atributo de la clase.

Manejo de Errores.

No hay posibilidad de error.

Posibilidad de Eficiencia.

N/A.

Funcionalidad Disponible en Bibliotecas Estándares.

N/D.

Algoritmos Disponibles:

N/A.

Pseudocódigo:

```
Asigna el estado del auxiliar.
Inicio asignarEstado (estado)
   Activar/Desactivar el Auxiliar.
Fin
```

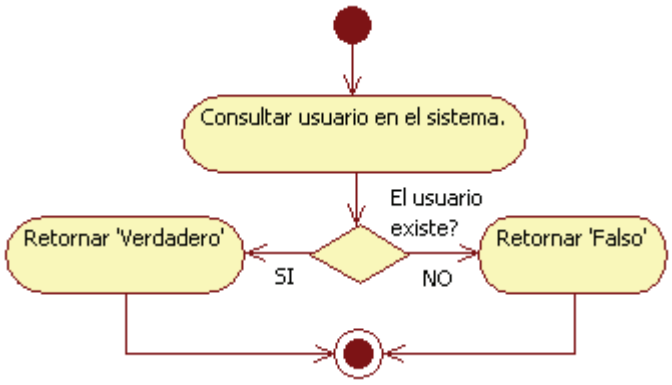
Datos a Manejarse:

- Un booleano.

Nombre de la Rutina: verificarAuxiliar.

Clase: Auxiliar.

Diseño:



Prerrequisitos:

- Se verifica que el usuario exista por lo que necesitará conexión con la base de datos para lograrlo.

Problema a Resolver:

a) Detalles.

- Verificará si el usuario existe.
- Emitirá una señal afirmativa si el usuario en verdad existe, de lo contrario emitirá una señal negativa.

b) Datos.

- La rutina esconderá el procedimiento para hallar su existencia en el sistema.
- No se tendrá parámetros de entrada.
- Se tendrá un valor de retorno booleano como salida.
- No existen precondiciones.
- El valor de retorno será garantía de la rutina que indicará el éxito o el fracaso

de la rutina en hallar al usuario.

Manejo de Errores.

No existe probabilidad de error en esta rutina.

Posibilidad de Eficiencia.

Se realizará la búsqueda del nombre del usuario mediante una consulta a la base de datos, cosa que no se necesitará por el momento un algoritmo optimizado.

Funcionalidad Disponible en Bibliotecas Estándares.

Se utilizará una forma de conexión a base de datos llamada LINQ, con el fin de realizar la consulta a la base sin preocuparse por la conectividad a la base de datos.

Algoritmos Disponibles:

Cómo si es posible utilizar LINQ, entonces no será prescindible investigar de algoritmos.

Pseudocódigo:

```
Verificará si el usuario existe y emitirá una señal afirmativa si el
usuario en verdad existe, de lo contrario emitirá una señal negativa.
Inicio verificarAuxiliar ()
   Consultar en la base de datos el nombre del usuario.
   Si el usuario existe entonces,
      Retornar 'Verdadero'.
   De lo contrario
      Retornar 'Falso'.
   Fin Si.
Fin
```

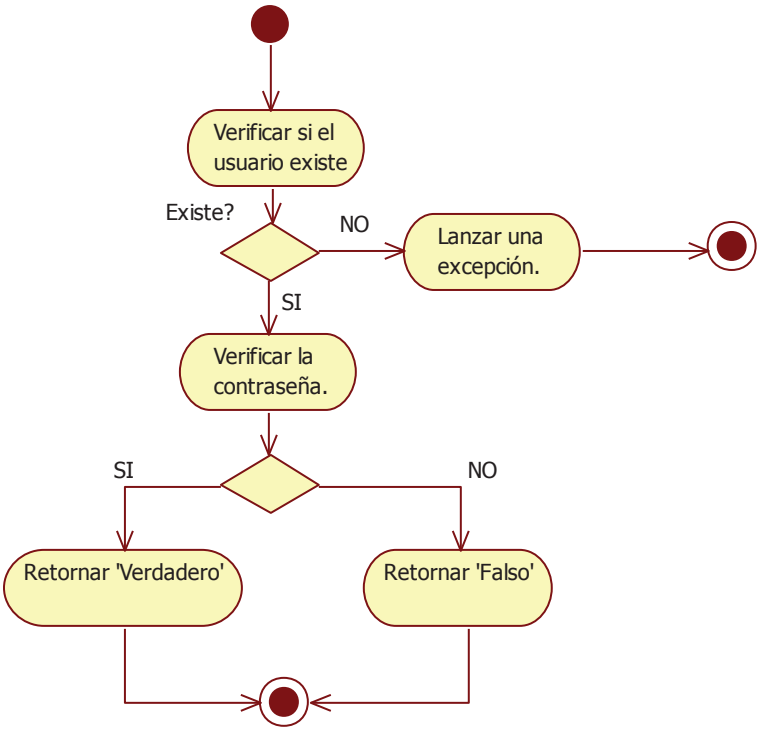
Datos a Manejarse:

- Un entero.
- Una cadena de texto.

Nombre de la Rutina: verificarContraseña.

Clase: Auxiliar.

Diseño:



Prerrequisitos:

- Lanzará una excepción si se busca una contraseña de un usuario inexistente.
- Realiza una consulta a la base de datos para dar el retorno del valor correspondiente.

Problema a Resolver:

a) Detalles.

- Verificará si es que la contraseña ingresada ha sido la correcta.
- Emitirá una señal afirmativa si la contraseña es correcta, de lo contrario emitirá una señal negativa.

b) Datos.

- La rutina esconderá el procedimiento para hallar la contraseña del auxiliar.
- La entrada será la contraseña a comprobar.
- Se tendrá un valor de retorno booleano como salida.
- No existen precondiciones.
- El valor de retorno será garantía de la rutina que indicará el éxito o el fracaso de la rutina en comprobar la contraseña.

Manejo de Errores.

Habrá un error provocado en la rutina que demuestre que el usuario no existe por lo que se recomienda ejecutar primero la rutina que demuestre la existencia del usuario.

Posibilidad de Eficiencia.

Se realizará la búsqueda de la contraseña mediante una consulta a la base de datos, cosa que no se necesitará por el momento un algoritmo optimizado.

Funcionalidad Disponible en Bibliotecas Estándares.

Se utilizará una forma de conexión a base de datos llamada LINQ, con el fin de realizar la consulta a la base sin preocuparse por la conectividad a la base de datos.

Algoritmos Disponibles:

N/A.

Pseudocódigo:

```
Verificará si es que la contraseña ingresada ha sido la correcta y
emitirá una señal afirmativa si la contraseña es correcta, de lo
contrario emite una señal negativa.
Inicio verificarContraseña ()
   Verificar si el usuario existe.

   Sino existe el usuario, entonces
     Lanzar una excepción.
     Interrumpir la rutina.
   Fin Si.

   Verificar la contraseña.

   Si la contraseña es correcta, entonces
     Retornar Verdadero.
   De lo contrario
     Retornar Falso.
   Fin Si.
Fin
```

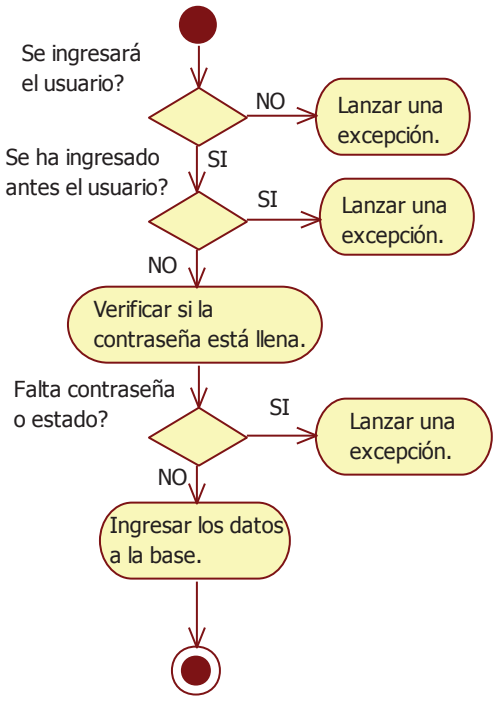
Datos a Manejarse:

- 2 enteros.
- 2 cadenas de texto.

Nombre de la Rutina: registrarAuxiliar.

Clase: Auxiliar.

Diseño:



Prerrequisitos:

- Mandará una excepción si falta algún dato.
- Registra los datos en la base.

Problema a Resolver:

a) Detalles.

- Se encarga de almacenar en la base de datos el registro del Auxiliar.

b) Datos.

- La rutina esconderá la forma como almacena los datos.
- No tiene ninguna entrada.
- No tiene ninguna salida.
- No tiene ninguna precondición.
- La poscondición es que los datos serán guardados en la base.

Manejo de Errores.

Es posible que no se ingresen la contraseña ni el estado por lo que a propósito será lanzada una excepción que advierta que se debe ingresar la contraseña y el estado.

Posibilidad de Eficiencia.

N/A.

Funcionalidad Disponible en Bibliotecas Estándares.

Se usará las bibliotecas que contengan la conectividad a base de datos mediante LINQ para registrar los datos.

Algoritmos Disponibles:

N/A.

Pseudocódigo:

```
Se encarga de almacenar en la base de datos el registro del Auxiliar.
Inicio registrarAuxiliar ()
   Si no se realizará el registro, entonces
     Lanzar una excepción.
     Terminar la rutina.
   Fin Si.

   Si se ingresó antes un usuario, entonces
     Lanzar una excepción.
     Terminar la rutina.
   Fin Si.

   Verificar si la contraseña está llena.

   Si falta contraseña o estado, entonces
     Lanzar una excepción.
     Terminar la rutina.
   De lo contrario
     Ingresar los datos a la base.
   Fin Si.
Fin
```

Datos a Manejarse:

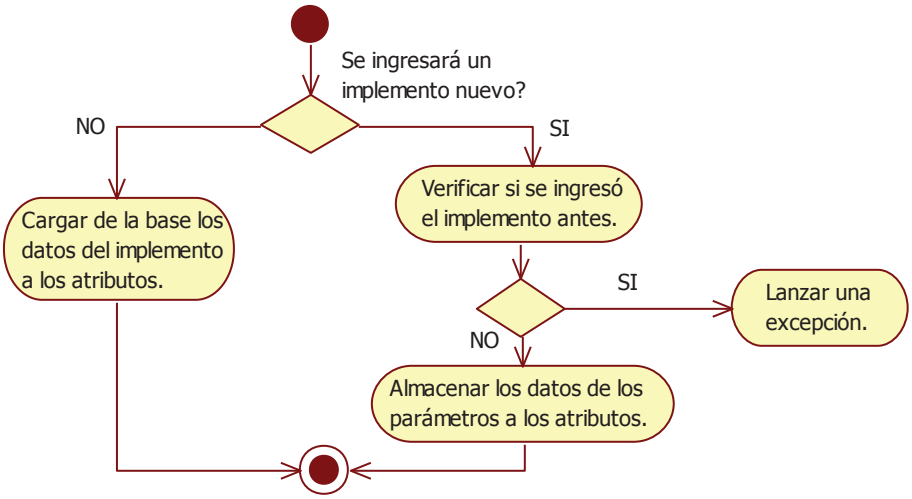
- 4 cadenas de texto.
- 2 booleanos.
- 2 enteros.

Implemento:

Nombre de la Rutina: Implemento.

Clase: Implemento.

Diseño:



Prerrequisitos:

- Carga los datos para una inserción.
- Carga los datos para una consulta.

Problema a Resolver:

a) Detalles.

- Recoge los datos obligatorios de un implemento para una inserción.
- Carga el registro de un implemento a la clase mediante una consulta a base de datos.

b) Datos.

- La rutina esconderá el procedimiento para recoger los datos del implemento.
- Se tendrá como entradas: el identificador, el implemento y la cantidad.
- No se tendrá salidas.
- La precondición es que el implemento es único.
- La poscondición es que los datos del implemento son cargados a la clase.

Manejo de Errores.

Es posible que se inserte de nuevo el implemento y eso causaría un error por lo que hay que evitar que eso suceda.

Posibilidad de Eficiencia.

N/A.

Funcionalidad Disponible en Bibliotecas Estándares.

La funcionalidad de LINQ para cuando se realice la consulta del registro.

Algoritmos Disponibles:

Se ha investigado de rutinas constructoras en algunos ejemplos de instanciación de objetos.

Pseudocódigo:

```
Recoge los datos obligatorios de un implemento para una inserción.
Inicio Implemento (nombre, cantidad)
   Verificar si se ingresó el implemento antes.

   Si se ha ingresado antes, entonces
      Lanzar una excepción.
      Terminar la rutina.
   De lo contrario
      Almacenar los datos de los parámetros a los atributos.
   Fin Si
Fin
```

```
Carga el registro de un implemento a la clase mediante una consulta a
base de datos.
Inicio Implemento (id)
   Cargar de la base los datos del implemento a los atributos.
Fin
```

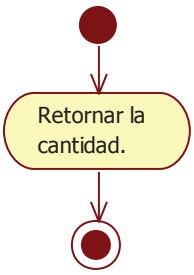
Datos a Manejarse:

- 3 enteros.
- Una cadena de texto.

Nombre de la Rutina: consultarCantidad.

Clase: Implemento.

Diseño:



Prerrequisitos:

- Retorna la cantidad, un atributo de la clase.

Problema a Resolver:

a) Detalles.

- Recupera la cantidad del implemento.

b) Datos.

- La rutina esconderá el retorno del atributo.
- No tendrá ninguna entrada.
- Retornará la cantidad.
- No hay precondiciones.
- No hay poscondiciones.

Manejo de Errores.

No hay posibilidad de error.

Posibilidad de Eficiencia.

N/A.

Funcionalidad Disponible en Bibliotecas Estándares.

N/D.

Algoritmos Disponibles:

N/A.

Pseudocódigo:

```
Recupera la cantidad del implemento.
Inicio consultarCantidad ()
   Retornar la cantidad.
Fin
```

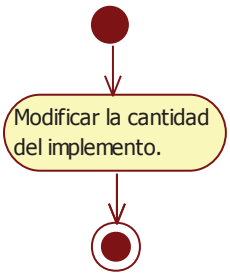
Datos a Manejarse:

- Un entero.

Nombre de la Rutina: asignarCantidad.

Clase: Implemento.

Diseño:



Prerrequisitos:

- Se almacena la cantidad ingresada después de instanciada la clase.

Problema a Resolver:

a) Detalles.

- Asigna la cantidad del implemento.

b) Datos.

- Se esconderá la asignación del parámetro al atributo de la clase.
- La entrada será la cantidad asignada.
- No tendrá salida.
- La precondición será el implemento instanciado.
- La poscondición será la cantidad almacenada en el atributo de la clase.

Manejo de Errores.

No hay posibilidad de error.

Posibilidad de Eficiencia.

N/A.

Funcionalidad Disponible en Bibliotecas Estándares.

N/D.

Algoritmos Disponibles:

N/A.

Pseudocódigo:

```
Asigna la cantidad del implemento.
Inicio asignarCantidad (cantidad)
   Modificar la cantidad del implemento.
Fin
```

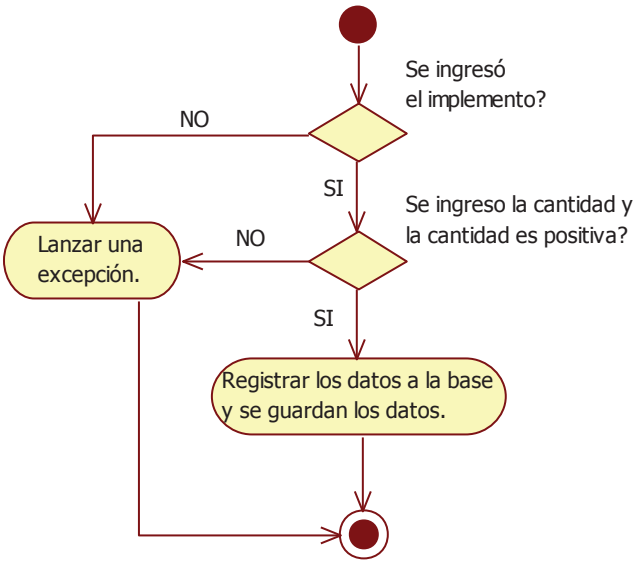
Datos a Manejarse:

- 2 enteros.

Nombre de la Rutina: registrarImplemento.

Clase: Implemento.

Diseño:



Prerrequisitos:

- Manda una excepción si la cantidad no fuera positiva.

- Registrará los datos en la base.

Problema a Resolver:

a) Detalles.

- Se encarga de almacenar en la base de datos el registro del Implemento.

b) Datos.

- La rutina esconderá la forma como almacena los datos.
- No tiene ninguna entrada.
- No tiene ninguna salida.
- La precondición es que no se ingresará otra vez un implemento.
- La poscondición es que los datos serán guardados en la base.

Manejo de Errores.

Es posible que se ingrese una cantidad no positiva por lo que la rutina lanzará una excepción a propósito con el fin de que no se ingrese un implemento inexistente.

Posibilidad de Eficiencia.

Se puede mejorar la rutina para reforzar la validación de esta rutina si es que ha instanciado la clase para consultas.

Funcionalidad Disponible en Bibliotecas Estándares.

Se usará las bibliotecas que contengan la conectividad a base de datos mediante LINQ para registrar los datos.

Algoritmos Disponibles:

N/A.

Pseudocódigo:

```
Se encarga de almacenar en la base de datos el registro del
Implemento.
Inicio registrarImplemento ()
   Si la cantidad es positiva y se ha ingresado el implemento, entonces
      Registrar los datos a la base.
```

```
   De lo contrario
     Lanzar una excepción.
     Terminar la rutina.
   Fin Si
Fin
```

Datos a Manejarse:

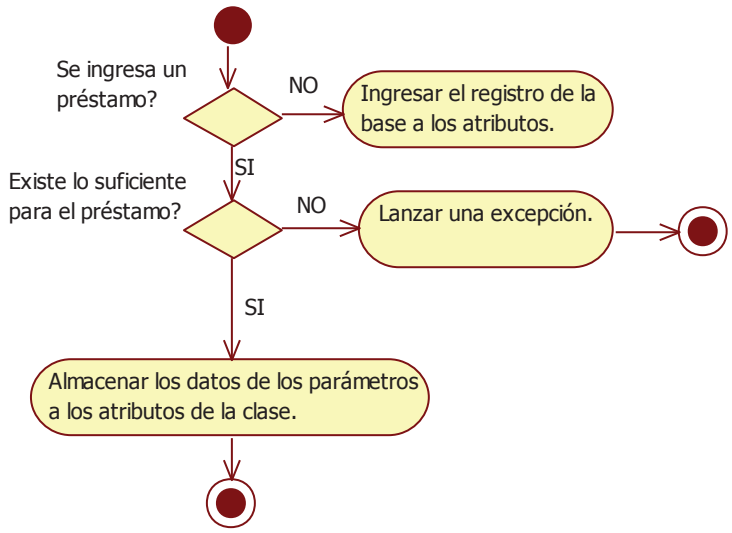
- 2 enteros.
- 2 cadenas de texto.

Préstamo:

Nombre de la Rutina: Préstamo.

Clase: Préstamo.

Diseño:



Prerrequisitos:

- Carga los datos para una inserción.
- Carga los datos para una consulta.

Problema a Resolver:

a) Detalles.

- Recoge los datos de un préstamo para una inserción.
- Carga el registro de un préstamo a la clase desde una consulta a base de datos.

b) Datos.

- La rutina esconderá el procedimiento para recoger los datos del préstamo.
- Se tendrá como entradas: el identificador, fecha, hora, tiempo, responsable, implemento, cantidad Implemento.
- No se tendrá salidas.
- La precondición es que se dispone de implementos y responsables del préstamo.
- La poscondición es que los datos del préstamo son cargados a la clase.

Manejo de Errores.

Existirá la probabilidad de error debido a que se podría realizar un préstamo más allá de lo disponible por lo que se la controlará para que no suceda.

Posibilidad de Eficiencia.

N/A.

Funcionalidad Disponible en Bibliotecas Estándares.

Se usará las bibliotecas que contengan la conectividad a base de datos mediante LINQ para registrar los datos.

Algoritmos Disponibles:

N/A.

Pseudocódigo:

```
Recoge los datos de un préstamo para una inserción.
Inicio Préstamo (fecha, hora, tiempo, responsable, implemento,
cantidadImplemento)
   Si existe lo suficiente para el préstamo, entonces
     Almacenar los datos de los parámetros a los atributos.
   De lo contrario
     Lanzar una excepción.
     Terminar la rutina.
   Fin Si.
Fin
```

```
Carga el registro de un préstamo a la clase desde una consulta a base
de datos.
Inicio Préstamo (id)
   Ingresar el registro de la base a los atributos.
Fin
```

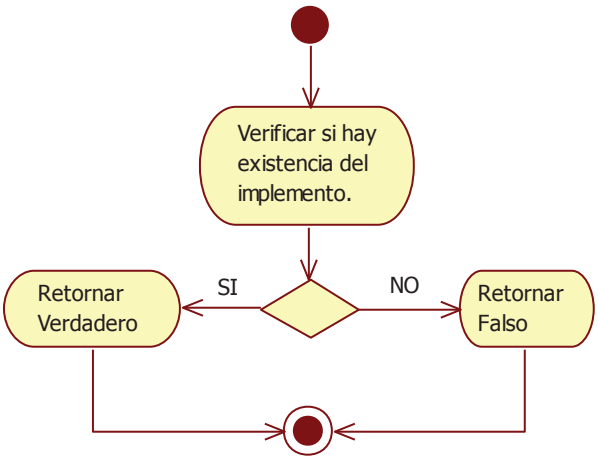
Datos a Manejarse:

- 2 datos de fecha.
- 2 datos de hora.
- 10 enteros.

Nombre de la Rutina: verificarExistencia.

Clase: Préstamo.

Diseño:



Prerrequisitos:

- Se verifica que la cantidad pedida no sea más allá de lo disponible por lo que necesitará conexión con la base de datos para lograrlo.

Problema a Resolver:

a) Detalles.

- Verificará si es que hay suficiente existencia del implemento para el préstamo realizado.

b) Datos.

- La rutina esconderá el procedimiento para verificar la existencia del implemento.
- No se tendrá parámetros de entrada.
- Se tendrá un valor de retorno booleano como salida.

- No existen precondiciones.
- El valor de retorno será garantía de la rutina que indicará el éxito o el fracaso de la rutina en verificar la existencia del implemento.

Manejo de Errores.

No existe probabilidad de error en esta rutina.

Posibilidad de Eficiencia.

Se realizará la verificación mediante una consulta a base de datos, cosa que no se necesitará por el momento un algoritmo optimizado.

Funcionalidad Disponible en Bibliotecas Estándares.

Se utilizará una forma de conexión a base de datos llamada LINQ, con el fin de realizar la consulta a la base sin preocuparse por la conectividad a la base de datos.

Algoritmos Disponibles:

Cómo si es posible utilizar LINQ, entonces no será prescindible investigar de algoritmos.

Pseudocódigo:

```
Verificará si es que hay suficiente existencia del implemento para el
préstamo realizado.
Inicio verificarExistencia ()
   Verificar si hay existencia del implemento.

   Si es que hay existencia del implemento, entonces
     Retornar 'Verdadero'
   De lo contrario
     Retornar 'Falso'
   Fin Si.
Fin
```

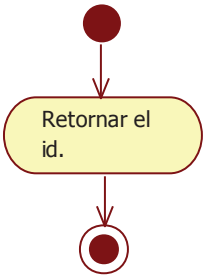
Datos a Manejarse:

- 2 enteros.

Nombre de la Rutina: consultarId.

Clase: Préstamo.

Diseño:



Prerrequisitos:

- Retorna el id, un atributo de la clase.

Problema a Resolver:

a) Detalles.

- Recupera el identificador del préstamo.

b) Datos.

- La rutina esconderá el retorno del atributo.
- No tendrá ninguna entrada.
- Retornará el identificador.
- No hay precondiciones.
- No hay poscondiciones.

Manejo de Errores.

No hay posibilidad de error.

Posibilidad de Eficiencia.

N/A.

Funcionalidad Disponible en Bibliotecas Estándares.

N/D.

Algoritmos Disponibles:

N/A.

Pseudocódigo:

```
Recupera el identificador del préstamo.
Inicio consultarId ()
   Retornar el id.
Fin
```

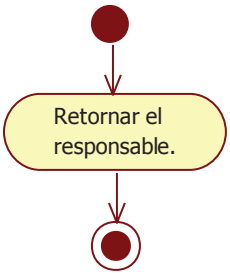
Datos a Manejarse:

- Un entero.

Nombre de la Rutina: consultarResponsable.

Clase: Préstamo.

Diseño:



Prerrequisitos:

- Retorna el responsable, un atributo de la clase.

Problema a Resolver:

a) Detalles.

- Recupera el identificador de la persona Sistemas que será responsable del préstamo.

b) Datos.

- La rutina esconderá el retorno del atributo.
- No tendrá ninguna entrada.
- Retornará el responsable.
- No hay precondiciones.
- No hay poscondiciones.

Manejo de Errores.

No hay posibilidad de error.

Posibilidad de Eficiencia.

N/A.

Funcionalidad Disponible en Bibliotecas Estándares.

N/D.

Algoritmos Disponibles:

N/A.

Pseudocódigo:

```
Recupera el identificador de la persona sistemas que será responsable
del préstamo.
Inicio consultarResponsable ()
   Retornar el responsable.
Fin
```

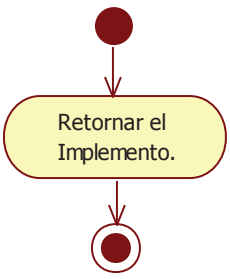
Datos a Manejarse:

- Un entero.

Nombre de la Rutina: consultarImplemento.

Clase: Préstamo.

Diseño:



Prerrequisitos:

- Retorna el implemento, un atributo de la clase.

Problema a Resolver:

a) Detalles.

- Recupera el identificador del implemento que se tomará en el préstamo.

b) Datos.

- La rutina esconderá el retorno del atributo.
- No tendrá ninguna entrada.
- Retornará el implemento.
- No hay precondiciones.
- No hay poscondiciones.

Manejo de Errores.

No hay posibilidad de error.

Posibilidad de Eficiencia.

N/A.

Funcionalidad Disponible en Bibliotecas Estándares.

N/D.

Algoritmos Disponibles:

N/A.

Pseudocódigo:

```
Recupera el identificador del implemento que se tomará en el préstamo.
Inicio consultarImplemento ()
   Retornar el implemento.
Fin
```

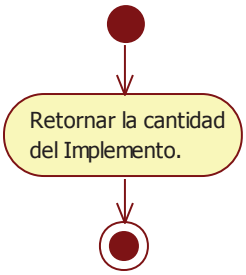
Datos a Manejarse:

- Un entero.

Nombre de la Rutina: consultarCantidadImplemento.

Clase: Préstamo.

Diseño:



Prerrequisitos:

- Retorna la cantidad del implemento, un atributo de la clase.

Problema a Resolver:

a) Detalles.

- Recupera la cantidad del implemento usado en el préstamo.

b) Datos.

- La rutina esconderá el retorno del atributo.
- No tendrá ninguna entrada.
- Retornará la cantidad del implemento.
- No hay precondiciones.
- No hay poscondiciones.

Manejo de Errores.

No hay posibilidad de error.

Posibilidad de Eficiencia.

N/A.

Funcionalidad Disponible en Bibliotecas Estándares.

N/D.

Algoritmos Disponibles:

N/A.

Pseudocódigo:

```
Recupera la cantidad del implemento usado en el préstamo.
Inicio consultarCantidadImplemento ()
   Retornará la cantidad del Implemento.
Fin
```

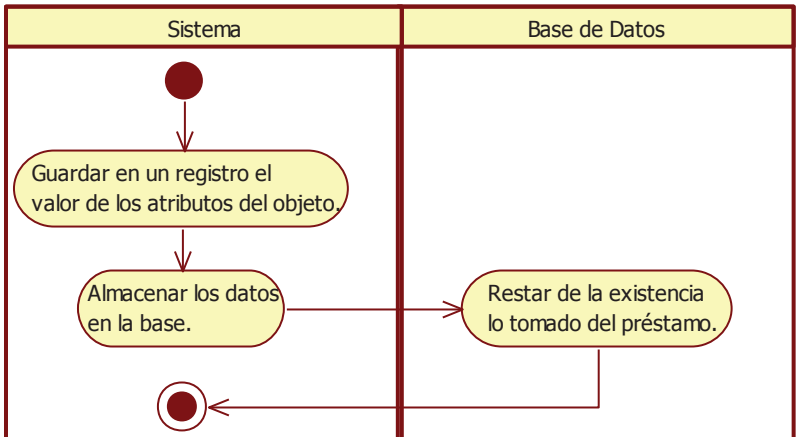
Datos a Manejarse:

- Un entero.

Nombre de la Rutina: registrarPréstamo.

Clase: Préstamo.

Diseño:



Prerrequisitos:

- Registrará los datos en la base.
- En la base de datos se actualizará la cantidad de implemento disponible.

Problema a Resolver:

a) Detalles.

- Se encarga de registrar en la base de datos el evento de préstamo.
- Actualiza la existencia del implemento en la base de datos.

b) Datos.

- La rutina esconderá la forma como registra los préstamos.
- No tiene ninguna entrada.
- No tiene ninguna salida.
- La precondición es que si hay existencia disponible para el préstamo.
- La poscondición es que los datos serán guardados en la base y se

actualizará la existencia del implemento.

Manejo de Errores.

Es posible que se ingrese una cantidad no positiva por lo que habrá una posibilidad de que lance la excepción por no ingresar la cantidad mínima requerida (1 implemento mínimo).

Posibilidad de Eficiencia.

Es posible mejorar la validación de la rutina si se la quiere hacer más eficiente.

Funcionalidad Disponible en Bibliotecas Estándares.

Se usará las bibliotecas que contengan la conectividad a base de datos mediante LINQ para registrar los datos.

Algoritmos Disponibles:

N/A.

Pseudocódigo:

```
Se encarga de registrar en la base de datos el evento de préstamo.
Inicio registrarPréstamo ()
   Guardar en un registro el valor de los atributos del implemento.

   Almacenar los datos en la base.
Fin
```

```
Actualiza la existencia del implemento en la base de datos.
Inicio actualizarExistenciaPréstamo (implemento, cantidad)
   Restar de la existencia lo tomado del préstamo.
Fin
```

Datos a Manejarse:

- 2 datos de fecha.
- 2 datos de hora.
- 8 enteros.

Nota: Se creó una rutina adicional debido a que esta será ejecutada por el motor de la base de datos por lo que no incluye en el diagrama original.

Devolución:

Nombre de la Rutina: Devolución.

Clase: Devolución.

Diseño:

Prerrequisitos:

- Carga los datos para una inserción.
- Consulta los datos de un préstamo ya realizado.

Problema a Resolver:

a) Detalles.

- Recoge los datos obligatorios de una devolución para una inserción.

b) Datos.

- La rutina esconderá el procedimiento para recoger los datos de la devolución.
- Se tendrá como entradas: el préstamo, fecha, hora y estado.
- No se tendrá salidas.
- La precondición es que existirá el préstamo para ser consultado.
- La poscondición es que los datos de la devolución son cargados a la clase.

Manejo de Errores.

No existe posibilidad de error.

Posibilidad de Eficiencia.

N/A.

Funcionalidad Disponible en Bibliotecas Estándares.

N/D.

Algoritmos Disponibles:

N/A.

Pseudocódigo:

```
Recoge los datos obligatorios de una devolución para una inserción.
Inicio Devolución (préstamo, fecha, hora, estado)
   Registrar la devolución en los atributos.
Fin
```

Datos a Manejarse:

- 2 objetos Préstamo
- 2 datos de fecha.
- 2 datos de hora.
- 2 booleanos.

Nombre de la Rutina: asignarObservación.

Clase: Devolución.

Diseño:

Prerrequisitos:

- Se guardará la observación después de ingresar los datos de la devolución.

Problema a Resolver:

a) Detalles.

- Agrega una observación respecto a lo que ha sucedido del préstamo realizado.

b) Datos.

- Se esconderá la asignación del parámetro al atributo de la clase.
- La entrada será la observación ingresada.
- No tendrá salida.
- La precondición será la devolución ingresada.
- La poscondición será la observación ingresada dentro de la devolución.

Manejo de Errores.

Es posible que exista error por desbordamiento de lo que se ingresa por lo que se debe controlar en el sistema ello o ajustar el tamaño del campo correspondiente en la base de datos.

Posibilidad de Eficiencia.

N/A.

Funcionalidad Disponible en Bibliotecas Estándares.

N/D.

Algoritmos Disponibles:

N/A.

Pseudocódigo:

```
Agrega una observación respecto a lo que ha sucedido del préstamo
realizado.
Inicio asignarObservación (observación)
   Agregar una observación a la devolución.
Fin
```

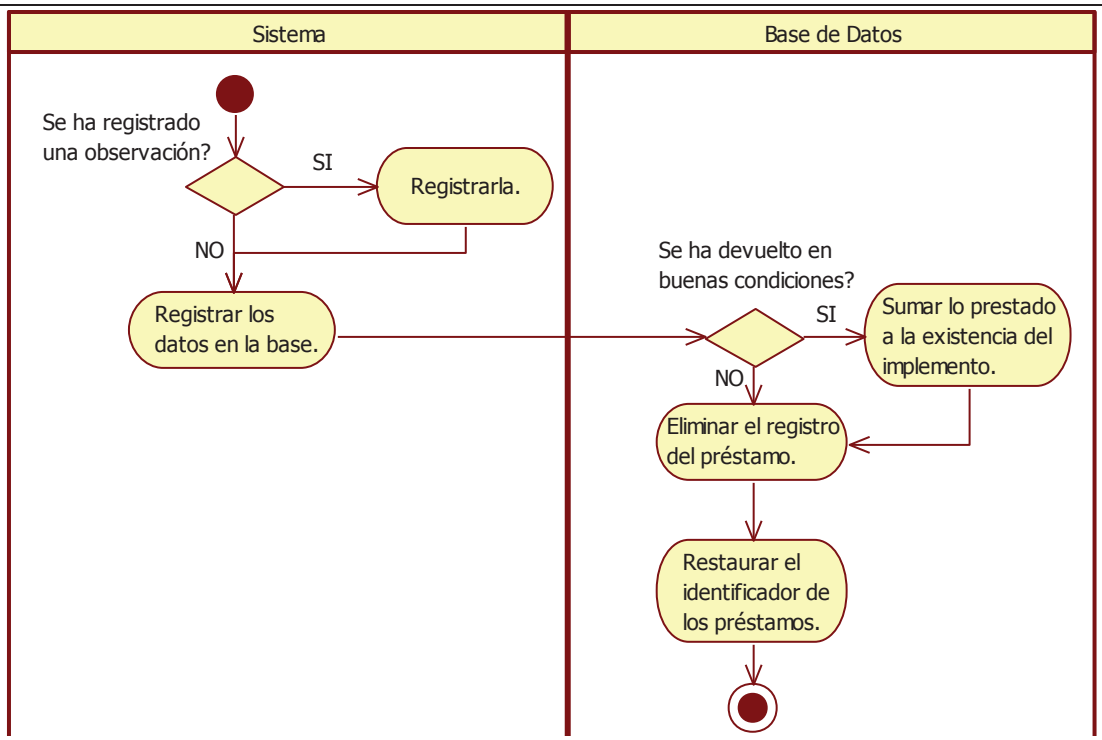
Datos a Manejarse:

- 2 cadenas de texto.

Nombre de la Rutina: registrarDevolución.

Clase: Devolución.

Diseño:



Prerrequisitos:

- Registra los datos en la base.
- En la base de datos se actualiza la cantidad de implemento disponible.

Problema a Resolver:

a) Detalles.

- Se encarga de registrar en la base de datos el evento de la devolución.
- Actualizará la existencia del implemento si fue devuelto en óptimas condiciones.

b) Datos.

- La rutina esconderá la forma como registra las devoluciones.
- No tiene ninguna entrada.
- No tiene ninguna salida.
- La precondición es que existirá un préstamo que va a consultarse.
- La poscondición es que los datos serán guardados en la base, se actualizará la existencia del implemento (si este está bien) y se eliminará un registro de préstamo.

Manejo de Errores.

No existe la probabilidad de errores.

Posibilidad de Eficiencia.

Existe la posibilidad de eficiencia si existe una forma de reforzar la rutina.

Funcionalidad Disponible en Bibliotecas Estándares.

Se usará las bibliotecas que contengan la conectividad a base de datos mediante LINQ para registrar los datos.

Algoritmos Disponibles:

N/A.

Pseudocódigo:

```
Se encarga de registrar en la base de datos el evento de la
devolución.
Inicio registrarDevolución ()
   Si se ha registrado una observación, entonces
     Registrarla.
   Fin Si.

   Registrar los datos en la base.
Fin
```

```
Actualizará la existencia del implemento si fue devuelto en óptimas
condiciones.
Inicio actualizarExistenciaDevolución (implemento, cantidad, estado,
responsable)
   Si se ha devuelto en buenas condiciones, entonces
     Sumar lo prestado a la existencia del implemento.
   Fin Si.
   Eliminar el registro del préstamo.
   Restaurar el identificador de los préstamos.
Fin
```

Datos a Manejarse:

- Un objeto Préstamo.
- 2 datos de fecha.
- 2 datos de hora.
- 2 booleanos.
- 2 cadenas de texto.
- 3 enteros.

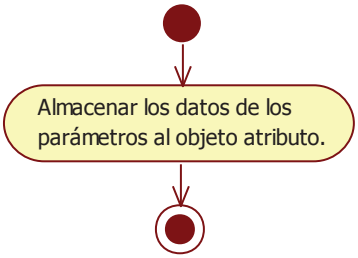
Nota: Se creó una rutina adicional debido a que esta será ejecutada por el motor de la base de datos por lo que no incluye en el diagrama original.

FachadaPersonaSistemas:

Nombre de la Rutina: FachadaPersonaSistemas.

Clase: FachadaPersonaSistemas.

Diseño:



Prerrequisitos:

- Los datos capturados son almacenados en la clase que la representa.

Problema a Resolver:

a) Detalles.

- Ayuda a simplificar la complejidad de la clase PersonaSistemas para usarla en ingreso de datos.

b) Datos.

- La rutina esconderá la forma cómo almacena los datos capturados en su clase origen.
- Las entradas son: la cédula, nombre, apellido, sexo, correo, dirección.
- No tiene salidas.
- La clase origen tiene las funciones necesarias para su funcionamiento.
- Los datos capturados son almacenados en la clase.

Manejo de Errores.

Los errores pueden suceder de la clase origen, por lo que tal clase debe ser controlada.

Posibilidad de Eficiencia.

N/A.

Funcionalidad Disponible en Bibliotecas Estándares.

N/D.

Algoritmos Disponibles:

N/A.

Pseudocódigo:

```
Ayuda a simplificar la complejidad de la clase PersonaSistemas para
usarla en ingreso de datos.
Inicio FachadaPersonaSistemas (cedula, nombre, apellido, sexo, correo,
direccion)
   Almacenar los datos de los atributos al objeto atributo.
Fin
```

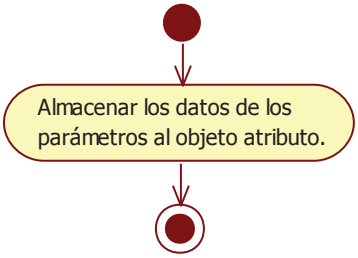
Datos a Manejarse:

- Un objeto PersonaSistemas.
- Un arreglo de caracteres.
- 4 cadenas de texto.
- Una enumeración.

Nombre de la Rutina: asignarTelefonoConvencional.

Clase: FachadaPersonaSistemas.

Diseño:



Prerrequisitos:

- Se almacena el teléfono convencional en la clase PersonaSistemas.

Problema a Resolver:

a) Detalles.

- Agrega un teléfono convencional si la persona lo posee.

b) Datos.

- La rutina esconderá la asignación del parámetro a la clase origen.
- Se tendrá como entrada el teléfono convencional (cadena de texto).
- No habrán salidas.
- La precondición es que el dato será correctamente ingresado.
- La poscondición será el dato ingresado a la clase origen.

Manejo de Errores.

Debido a que lo que se ingresa es una cadena se controlará desde el sistema del ingreso del teléfono.

Posibilidad de Eficiencia.

Se podría lanzar una excepción si el ingreso no fuese correcto.

Funcionalidad Disponible en Bibliotecas Estándares.

N/D.

Algoritmos Disponibles:

N/A.

Pseudocódigo:

```
Agrega un teléfono convencional si la persona lo posee.
Inicio asignarTelefonoConvencional (telefonoConvencional)
   Agregar un teléfono convencional.
Fin
```

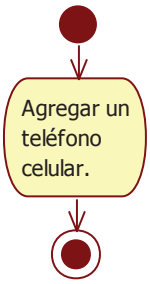
Datos a Manejarse:

- Un objeto PersonaSistemas.
- Una cadena de texto.

Nombre de la Rutina: asignarTelefonoCelular.

Clase: FachadaPersonaSistemas.

Diseño:



Prerrequisitos:

- Se almacena el teléfono celular en la clase PersonaSistemas.

Problema a Resolver:

a) Detalles.

- Agrega un teléfono celular si la persona lo posee.

b) Datos.

- La rutina esconderá la asignación del parámetro a la clase origen.
- Se tendrá como entrada el teléfono celular (arreglo de 10 caracteres).
- No habrán salidas.
- La precondición es que el dato tendrá justo 10 caracteres.
- La poscondición será el dato ingresado a la clase origen.

Manejo de Errores.

Es posible que no se ingrese un número y que se ingrese menos de 10 caracteres, tales cosas serán controladas desde la aplicación.

Posibilidad de Eficiencia.

Se podría lanzar una excepción si el ingreso no fuese correcto.

Funcionalidad Disponible en Bibliotecas Estándares.

N/D.

Algoritmos Disponibles:

N/A.

Pseudocódigo:

```
Agrega un teléfono celular si la persona lo posee.
Inicio asignarTelefonoCelular (telefonoCelular)
   Agregar un teléfono celular.
Fin
```

Datos a Manejarse:

- Un objeto PersonaSistemas.
- Un arreglo de caracteres.

Nombre de la Rutina: registrarPersonaSistemas.

Clase: FachadaPersonaSistemas.

Diseño:

Prerrequisitos:

- Ejecuta la rutina de registro de datos de la clase PersonaSistemas.

Problema a Resolver:

a) Detalles.

- Activa la ejecución de la inserción de datos de la clase PersonaSistemas.

b) Datos.

- La rutina esconderá lo que se ordena ejecutar.
- No tiene entradas.
- No tiene salidas.
- La precondición es que los datos son previamente ingresados.
- La poscondición es que el registro será ejecutado.

Manejo de Errores.

Existe la posibilidad de error por parte de la clase origen por lo que se deben controlar tales cosas en la clase origen.

Posibilidad de Eficiencia.

N/A.

Funcionalidad Disponible en Bibliotecas Estándares.

N/D.

Algoritmos Disponibles:

N/A.

Pseudocódigo:

```
Activa la ejecución de la inserción de datos de la clase
PersonaSistemas.
Inicio registrarPersonaSistemas ()
   Ejecutar el registro de PersonaSistemas.
Fin
```

Datos a Manejarse:

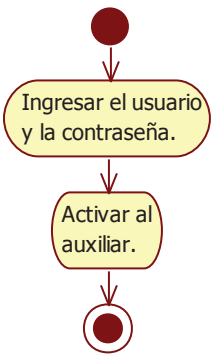
- Ninguno.

FachadaAuxiliar:

Nombre de la Rutina: FachadaAuxiliar.

Clase: FachadaAuxiliar.

Diseño:



Prerrequisitos:

- Los datos capturados son almacenados en la clase que los representa.

Problema a Resolver:

a) Detalles.

- Ayuda a simplificar la complejidad de la clase Auxiliar para usarla en ingreso de datos.

b) Datos.

- La rutina esconderá la forma cómo almacena los datos capturados en su clase origen.
- Las entradas son: el usuario, contraseña.
- No tiene salidas.
- La clase origen tiene las funciones necesarias para su funcionamiento.
- Los datos capturados son almacenados en la clase.

Manejo de Errores.

Los errores pueden suceder de la clase origen, por lo que tal clase debe ser controlada.

Posibilidad de Eficiencia.

N/A.

Funcionalidad Disponible en Bibliotecas Estándares.

N/D.

Algoritmos Disponibles:

N/A.

Pseudocódigo:

```
Ayuda a simplificar la complejidad de la clase Auxiliar para usarla en
ingreso de datos.
Inicio FachadaAuxiliar (usuario, contraseña)
   Ingresar el usuario y la contraseña.
   Activar al auxiliar.
Fin
```

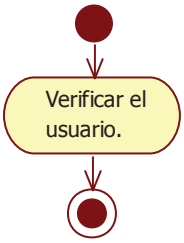
Datos a Manejarse:

- Un objeto Auxiliar.
- 2 cadenas de texto.

Nombre de la Rutina: verificarAuxiliar.

Clase: FachadaAuxiliar.

Diseño:



Prerrequisitos:

- Ejecuta la rutina de verificación del usuario de la clase Auxiliar.

Problema a Resolver:

a) Detalles.

- Activa la verificación de la existencia del usuario.

b) Datos.

- La rutina esconderá lo que se ordena ejecutar.
- No tiene entradas.
- Se tendrá un valor de retorno booleano como salida.
- La precondición es que el usuario es previamente ingresado.
- La poscondición es que retornará el valor de verificación.

Manejo de Errores.

Existe la posibilidad de error por parte de la clase origen por lo que se deben controlar tales cosas en la clase origen.

Posibilidad de Eficiencia.

N/A.

Funcionalidad Disponible en Bibliotecas Estándares.

N/D.

Algoritmos Disponibles:

N/A.

Pseudocódigo:

```
Activa la verificación de la existencia del usuario.
Inicio verificarAuxiliar ()
   Verificar el usuario.
Fin
```

Datos a Manejarse:

- Un booleano.

Nombre de la Rutina: registrarAuxiliar.

Clase: Auxiliar.

Diseño:

Prerrequisitos:

- Ejecuta la rutina de registro de datos de la clase Auxiliar.

Problema a Resolver:

a) Detalles.

- Activa la ejecución de la inserción de datos de la clase Auxiliar.

b) Datos.

- La rutina esconderá lo que se ordena ejecutar.

- No tiene entradas.
- No tiene salidas.
- La precondición es que los datos son previamente ingresados.
- La poscondición es que el registro será ejecutado.

Manejo de Errores.

Existe la posibilidad de error por parte de la clase origen por lo que se deben controlar tales cosas en la clase origen.

Posibilidad de Eficiencia.

N/A.

Funcionalidad Disponible en Bibliotecas Estándares.

N/D.

Algoritmos Disponibles:

N/A.

Pseudocódigo:

```
Activa la ejecución de la inserción de datos de la clase Auxiliar.
Inicio registrarAuxiliar ()
   Registrar el auxiliar.
Fin
```

Datos a Manejarse:

- Ninguno.

CASOS DE PRUEBA DE CADA RUTINA.

PersonaSistemas.

Rutina: PersonaSistemas.

# de Caso	Descripción	Datos de Entrada	Resultado Esperado
1	Se verifica si es que solo se almacenaron los datos básicos.	Cédula. Nombre. Apellido. Sexo. Correo. Dirección.	Teléfono convencional nulo.

2	Se verifica que se hayan cargado los datos de un registro conocido.	Un identificador de alguien existente.	Id. Nombre. Apellido.
3	Se verifica que el identificador sea nulo.	Un identificador nulo.	Un identificador nulo.

Rutina: asignarTelefonoConvencional.

# de Caso	Descripción	Datos de Entrada	Resultado Esperado
1	Se verifica si es que se ingresó el teléfono convencional.	Teléfono convencional (cadena de texto de 7 o 9 caracteres).	Teléfono convencional almacenado en la clase.

Rutina: asignarTelefonoCelular.

# de Caso	Descripción	Datos de Entrada	Resultado Esperado
1	Se verifica si es que se ingresó el teléfono celular.	Teléfono celular (arreglo de 10 caracteres).	Teléfono celular instanciado en el objeto.

Rutina: registrarPersonaSistemas.

# de Caso	Descripción	Datos de Entrada	Resultado Esperado
1	Se verifica si es que se ingresó el registro con sus teléfonos convencional y celular.	Datos requeridos de la clase. Teléfono convencional. Teléfono celular.	Registro ingresado con teléfonos incluidos.
2	Se verifica si es que se ingresó el registro con su teléfono convencional.	Datos requeridos de la clase. Teléfono convencional.	Registro ingresado con teléfono convencional solamente.
3	Se verifica si es que se ingresó el registro con su teléfono celular.	Datos requeridos de la clase. Teléfono celular.	Registro ingresado con teléfono celular solamente.
4	Se verifica si es que se ingresó el registro sin poseer teléfono alguno.	Datos requeridos de la clase.	Registro ingresado sin teléfono alguno.
5	Se verifica que lance una excepción si es que se ha ingresado un dato previo.	Datos de prueba con la cédula de un usuario previamente ingresado.	Una excepción como resultado de un intento de ingreso de un registro repetido.

Auxiliar.

Rutina: Auxiliar.

# de Caso	Descripción	Datos de Entrada	Resultado Esperado

1	Se verifica si es que la clase se usará para consulta de un Auxiliar.	Identificador positivo.	Hallazgo de su registro en base de datos.
2	Se verifica si es que la clase se usará para la inserción.	Nombre de usuario. Identificador positivo.	No debe existir su registro en la base de datos.
3	Se verifica si es que la clase se usará para una autenticación.	Nombre de usuario. Identificador neutro.	El resto de sus datos no deben ser cargados.

Rutina: asignarContraseña.

# de Caso	Descripción	Datos de Entrada	Resultado Esperado
1	Se verifica si es se almacena la contraseña.	Una contraseña (una cadena de texto).	La contraseña almacenada dentro de la clase.

Rutina: asignarEstado.

# de Caso	Descripción	Datos de Entrada	Resultado Esperado
1	Se verifica si es que se almacena el estado en la clase.	Un estado (dato booleano).	El estado almacenado en la clase.

Rutina: verificarAuxiliar.

# de Caso	Descripción	Datos de Entrada	Resultado Esperado
1	Se verifica si el valor es positivo cuando se halla el usuario.	Una cadena de un usuario existente.	Verdadero.
2	Se verifica si el valor es negativo cuando no se halla usuario.	Una cadena de un usuario inexistente.	Falso.

Rutina: verificarContraseña.

# de Caso	Descripción	Datos de Entrada	Resultado Esperado
1	Se verifica si el valor es positivo cuando se ingresa la contraseña correcta.	Una cadena de un usuario existente. Una cadena de su contraseña correcta.	Verdadero.
2	Se verifica si el valor es negativo cuando se ingresa la contraseña incorrecta.	Una cadena de un usuario existente. Una cadena de su contraseña incorrecta.	Falso.
3	Se verifica que lance una excepción si se trata de un usuario	Una cadena de un usuario inexistente.	Excepción.

	inexistente.		

Rutina: registrarAuxiliar.

# de Caso	Descripción	Datos de Entrada	Resultado Esperado
1	Se verifica que lance una excepción si se ha ingresado antes.	Un usuario previamente ingresado.	Excepción.
2	Se verifica que lance una excepción si falta la contraseña y el estado.	Contraseña vacía. Estado inactivo. Una persona sistemas existente.	Excepción.
3	Se verifica que lance una excepción si falta la contraseña.	Contraseña vacía. Estado activo. Una persona sistemas existente.	Excepción.
4	Se verifica que lance una excepción si falta el estado.	Contraseña llenada. Estado inactivo. Una persona sistemas existente.	Excepción.
5	Se verifica que se haya ingresado el auxiliar esperado.	Contraseña llenada. Estado activo. Una persona sistemas existente.	Registro insertado en la base de datos.
6	Se verifica que en realidad se tenga el identificador para la inserción.	Una persona sistemas inexistente (un identificador nulo).	Excepción.

Implemento.

Rutina: Implemento.

# de Caso	Descripción	Datos de Entrada	Resultado Esperado
1	Se verifica que lance una excepción si se ha ingresado antes el registro.	Nombre de un implemento existente. Cualquier cantidad.	Excepción.
2	Se verifica que se ingrese los datos a la clase.	Nombre de un implemento inexistente. Cualquier cantidad.	No debe existir el registro en la base.
3	Se verifica que se carguen los datos de un cierto registro a la clase.	Identificador de un registro existente.	El registro almacenado en la clase debió haberse almacenado en la clase.

Rutina: asignarCantidad.

# de Caso	Descripción	Datos de Entrada	Resultado Esperado
1	Se verifica que se haya ingresado la cantidad esperada.	Una cantidad cualquiera.	Cantidad incrementada en una unidad.

Rutina: registrarImplemento.

# de Caso	Descripción	Datos de Entrada	Resultado Esperado
1	Se verifica que se haya ingresado el implemento.	Implemento ingresado. Cantidad positiva.	Registro insertado en la base.
2	Se verifica que lance una excepción cuando no se ha ingresado el implemento.	Implemento no ingresado (cadena de texto vacía). Cantidad cualquiera.	Excepción.
3	Se verifica que lance una excepción cuando la cantidad no es positiva.	Implemento ingresado. Cantidad no positiva.	Excepción.

Préstamo.

Rutina: Préstamo.

# de Caso	Descripción	Datos de Entrada	Resultado Esperado
1	Se verifica que se haya cargado los datos a la clase para registrarlos en la base.	Fecha. Hora. Tiempo. Responsable Implemento. Cantidad de Implemento <= Cantidad disponible.	Datos cargados a la clase excepto el identificador.
2	Se verifica que se haya cargado los datos a la clase desde la base.	Id.	Datos cargados existan en la base de datos.
3	Se verifica que se haya lanzado una excepción sino se tiene lo suficiente para el préstamo.	Fecha. Hora. Tiempo. Responsable Implemento. Cantidad de Implemento > Cantidad disponible.	Excepción.

Rutina: registrarPréstamo.

# de Caso	Descripción	Datos de Entrada	Resultado Esperado

1	Se verifica que se ha ingresado el préstamo con normalidad.	Fecha. Hora. Tiempo. Responsable Implemento. Cantidad de Implemento <= Cantidad disponible.	Registro ingresado en la base de datos y que se restado de la cantidad disponible del implemento.

Devolución.

Rutina: registrarDevolución.

# de Caso	Descripción	Datos de Entrada	Resultado Esperado
1	Se verifica si la devolución se ha registrado con su respectiva observación.	Préstamo. Fecha. Hora. Estado (OK). Observación.	Devolución registrada con observación. Cantidad de implemento restaurada. Préstamo respectivo eliminado.
2	Se verifica si la devolución se ha registrado sin observación.	Préstamo. Fecha. Hora. Estado (OK).	Devolución registrada sin observación. Cantidad de implemento restaurada. Préstamo respectivo eliminado.
3	Se verifica que al registrar la devolución con observación, no se haya alterado la cantidad disponible debido a que no se ha retornado en buen estado.	Préstamo. Fecha. Hora. Estado (Dañado). Observación.	Devolución registrada con observación. Cantidad de implemento intacta. Préstamo respectivo eliminado.
4	Se verifica que al registrar la devolución sin observación, no se haya alterado la cantidad disponible debido a que no se ha retornado en buen estado.	Préstamo. Fecha. Hora. Estado (Dañado).	Devolución registrada sin observación. Cantidad de implemento intacta. Préstamo respectivo eliminado.

ANEXO D

Estadísticas del Caso de Estudio

NUMERO DE LINEAS DE CODIGO.

Fecha	31/10/2013	01/11/2013	03/11/2013	04/11/2013	05/11/2013	06/11/2013
Líneas de Código	1917	2074	2227	2339	2611	2735

Fecha	11/11/2013	12/11/2013	13/11/2013	14/11/2013	18/11/2013	19/11/2013
Líneas de Código	3042	3092	3154	3165	3264	3344

Fecha	22/11/2013	24/11/2013	27/11/2013	02/12/2013
Líneas de Código	3407	3470	3577	3579

NUMERO DE CLASES.

Fecha	31/10/2013	01/11/2013	03/11/2013	04/11/2013	05/11/2013	06/11/2013
Número de Clases	33	35	39	40	41	42

Fecha	11/11/2013	12/11/2013	13/11/2013	14/11/2013	18/11/2013	19/11/2013
Número de Clases	44	44	44	44	44	44

Fecha	22/11/2013	24/11/2013	27/11/2013	02/12/2013
Número de Clases	45	45	45	45

NUMERO DE RUTINAS.

Fecha	31/10/2013	01/11/2013	03/11/2013	04/11/2013	05/11/2013	06/11/2013
Número de Rutinas	188	220	247	261	313	328

Fecha	11/11/2013	12/11/2013	13/11/2013	14/11/2013	18/11/2013	19/11/2013
Número de Rutinas	386	400	412	415	439	456

Fecha	22/11/2013	24/11/2013	27/11/2013	02/12/2013
Número de Rutinas	467	474	495	496