TEMAS SELECTOS DE DESARROLLO DE SOFTWARE

Roque Hernández, Ramón Ventura
Temas selectos de desarrollo de software / Ramón Ventura Roque Hernández, .—Ciudad de México : Colofón; Universidad Autónoma de Tamaulipas, 2019.
112 págs. ; 17 x 23 cm.
1. Software de computadora – Desarrollo
LC: QA76.76.A54 R66 DEWEY: 005.2762 R66

Centro Universitario Victoria
Centro de Gestión del Conocimiento. Tercer Piso
Cd. Victoria, Tamaulipas, México. C.P. 87149
consejopublicacionesuat@outlook.com

Matamoros SN, Zona Centro Ciudad Victoria, Tamaulipas C.P. 87000
Consejo de Publicaciones UAT
Tel. (52) 834 3181-800 • extensión: 2948 • *www.uat.edu.mx*

Fomento Editorial Una edición del Departamento de Fomento Editorial de la Universidad Autónoma de Tamaulipas

Edificio Administrativo, planta baja, CU Victoria
Ciudad Victoria, Tamaulipas, México
Libro aprobado por el Consejo de Publicaciones UAT
ISBN UAT: 978-607-8626-65-6

Colofón
Franz Hals núm. 130, Alfonso XIII
Delegación Álvaro Obregón C.P. 01460, Ciudad de México
www.paraleer.com/colofonedicionesacademicas@gmail.com
ISBN: 978-607-635-000-3

Impreso en México •*Printed in Mexico* El tiraje consta de 300 ejemplares

Este libro fue dictaminado y aprobado por el Consejo de Publicaciones UAT mediante un especialista en la materia. Asimismo fue recibido por el Comité Interno de Selección de Obras de Colofón Ediciones Académicas para su valoración en la sesión del primer semestre 2018, se sometió al sistema de dictaminación a "doble ciego" por especialistas en la materia, el resultado de ambos dictámenes fue positivo.

TEMAS SELECTOS DE DESARROLLO DE SOFTWARE

Ramón Ventura Roque Hernández

UAT
VERDAD, BELLEZA, PROBIDAD
Consejo de Publicaciones
Fomento Editorial
COLOFÓN

Índice

Introducción

Este libro muestra diferentes aristas de la ingeniería del *software* desde perspectivas contemporáneas y emergentes. El contenido es mayormente técnico, sin embargo, también contribuye al entendimiento de enfoques emergentes que han venido privilegiando a los seres humanos sobre procesos y herramientas tecnológicas.

En el primer capítulo se explica el proceso de desarrollo de *software* y su evolución partiendo de metodologías rígidas y tradicionales hasta la aparición de los enfoques ágiles. El segundo capítulo muestra a la Programación Extrema como un enfoque ágil que permite el desarrollo rápido de sistemas de *software* con la atención puesta en prácticas específicas orientadas a la creación de código sin soslayar la importancia de las personas involucradas en el proceso.

El tercer capítulo expone dos experiencias prácticas de investigación relacionadas con el desarrollo ágil utilizando Programación Extrema y *Scrum,* en donde el autor participó. En el cuarto capítulo se presentan algunos de los paradigmas más utilizados para la concepción, diseño e implementación de programas de cómputo y al mismo tiempo se introducen los fundamentos de dos de los paradigmas más recientes en la ingeniería de software: el orientado a objetos y el orientado a aspectos; primero, ubicando a ambos como resultado de la evolución de los paradigmas estudiados y luego, mostrando sus generalidades desde una perspectiva de implementación.

En el capítulo cinco se caracteriza con mayor detalle la orientación a objetos. Se destacan los elementos principales y secundarios de su modelo y se proporciona un panorama de la anatomía de sus programas. El capítulo seis contiene los fundamentos de la orientación a aspectos, un paradigma aún más reciente que introduce nuevos elementos conceptuales con la finalidad de lograr mayores niveles de organización, legibilidad y reutilización en el código de los programas desarrollados. En el capítulo siete se realiza una comparación entre la orientación a objetos y la orientación a aspectos y se muestran los beneficios, problemas y el estado actual de ambos paradigmas.

El capítulo ocho aborda la reutilización, un objetivo fundamental y deseable en los sistemas. Se explican tres de las tendencias más populares para lograr el uso repetido de elementos de *software*: la ingeniería de *software* basada en componentes, las líneas de producto y el desarrollo con componentes reutilizables.

El capítulo nueve reúne un conjunto de reflexiones finales sobre los temas tratados en este libro.

I.

INTRODUCCIÓN AL DESARROLLO DE *SOFTWARE* CONTEMPORÁNEO

1.1 El proceso de desarrollo de *software*

El proceso de desarrollo de *software* se puede conceptualizar como un conjunto de actividades que incluyen análisis, diseño, programación, pruebas, conversión, producción y mantenimiento (Laudon & Laudon, 2013). Existe una gran diversidad de enfoques metodológicos y cada uno organiza estas actividades de forma distinta. No existe una fórmula universal y única para desarrollar *software* que pueda garantizar siempre el éxito total de un proyecto en cualquier situación. Incluso, la aplicación de una misma metodología en escenarios diferentes podría producir variación en los resultados. Sin embargo, existen actividades básicas que siempre están presentes en los enfoques metodológicos sin importar las configuraciones concretas que puedan llegar a tener. A continuación, se describirán las actividades básicas de desarrollo de *software.*

La actividad inicial es el análisis de sistemas, cuyo objetivo es entender la problemática que da origen al desarrollo de *software*; en esta tarea se deben comprender y definir las necesidades de los usuarios, denominadas "requerimientos", las cuales guiarán el proceso completo de creación del *software*. La siguiente actividad es el diseño, en donde se realizan especificaciones de todos los detalles del programa que se está elaborando. Se deben describir las características más importantes como las entradas, las salidas, los procesos y las interfaces. La programación, construcción o implementación es otra actividad que consiste en escribir el código fuente que corresponde a los algoritmos del *software*; para esta tarea se utilizan las palabras reservadas de un lenguaje. En la programación es en donde se crea un sistema funcional con el que los usuarios puedan trabajar. La etapa de pruebas consiste en utilizar el programa de manera experimental y preliminar con el objetivo de localizar errores. Algunas pruebas que se deben realizar son: las pruebas unitarias, que revisan cada uno de los componentes por separado, las pruebas de integración, que examinan el funcionamiento del sistema completo y las pruebas de aceptación, que se centran en evaluaciones de los usuarios. Posteriormente se llevan a cabo las tareas de conversión, que permiten a los usuarios migrar del sistema antiguo al nuevo. Cuando los usuarios utilizan el nuevo sistema cotidianamente en su entorno de trabajo se dice que el *software* se encuentra en la etapa de "producción". En esta fase se realizan actividades de mantenimiento, las cuales se encargan de realizar modificaciones al programa para solucionar problemas o hacerle mejoras.

La filosofía y la configuración de las actividades que es propia de cada metodología puede diferenciar significativamente un enfoque metodológico de otro. En los siguientes apartados se explicarán brevemente las metodologías tradicionales, los problemas que se han identificado con ellas en el desarrollo de *software* y cómo éstos han dado paso a los recientes enfoques metodológicos ágiles.

1.2 Metodologías tradicionales

Una metodología de desarrollo es un conjunto de procedimientos, técnicas, principios y herramientas cuyo objetivo principal es crear programas de cómputo (Baird, 2002). Algunas de las metodologías tradicionales en la ingeniería del *software* han sido: El ciclo de desarrollo en cascada, el Proceso Unificado (PU) -*Unified Process (UP)*-, y el uso de prototipos evolutivos.

En el ciclo de desarrollo en cascada se sigue siempre una secuencia ordenada de fases (Análisis, Diseño, Codificación, Prueba, Mantenimiento). Este modelo de trabajo proviene de otros campos de la ingeniería que están orientados a la construcción de un producto que una vez desarrollado, no varía con el tiempo; por eso, aplicado al *software*, que es de naturaleza cambiante, el proceso se torna difícil e incómodo cuando es necesario realizar modificaciones después de las primeras fases. Entre más tarde se detectan los cambios, más caros resultan éstos. Esta metodología produce demasiada documentación, y requiere que una etapa termine antes de iniciar con la siguiente. El desarrollo en cascada es útil cuando se trata de un *software* pequeño, o los requerimientos son fijos y pueden definirse claramente desde el principio. Sin embargo, hoy en día se ha reconocido que el desarrollo en cascada resulta inflexible y poco adaptable para muchos proyectos modernos.

El Proceso Unificado (PU) es un enfoque de trabajo iterativo y disciplinado con un conjunto estándar de herramientas, plantillas y entregables que propone la reducción de riesgos en el proyecto. El PU se apoya en el Lenguaje Unificado de Modelado (UML) para comunicar requerimientos, arquitectura y diseño. A pesar de ser una buena propuesta para estandarizar la comunicación, puede generar mucha documentación y puede ser confuso en proyectos que no son de gran tamaño.

El uso de prototipos evolutivos es una metodología RAD *(Rapid Application Development)* que se basa en el uso de prototipos visuales, mediante los cuales el cliente perfila lo que necesita y lo va refinando a través de la comunicación continua con los desarrolladores. Esta metodología es flexible a los requerimientos cambiantes, pero la planeación puede resultar confusa pues se desconoce la duración del proyecto y es fácil terminar en un ciclo infinito de solicitudes y cambios.

1.3 Problemas en el desarrollo de *software*

El desarrollo de *software* es una actividad compleja que fácilmente puede resultar caótica, especialmente en proyectos de gran tamaño. En muchos sistemas las actividades típicas han sido codificar y corregir, esto es, trabajar a prueba y error; la mayoría de las veces sin la planeación suficiente y sin un diseño sencillo y funcional. De hecho, la construcción de los primeros sistemas antes de la existencia de la ingeniería de *software* fue realizada de esta manera. Esto podría funcionar hoy en

día en algunos escenarios, por ejemplo, si el sistema es muy pequeño; sin embargo, seguramente se producirían demasiados errores pues no se contaría con un control de calidad y se desperdiciaría gran cantidad de recursos pues se generaría trabajo adicional. En resumen, trabajar "a prueba y error" no es recomendable en la creación de *software* de buena calidad.

Con el advenimiento de las nuevas tecnologías, redes e internet, los sistemas tendieron a necesitar mayor cantidad de componentes que resultan difíciles de agregar, implementar, probar y corregir. Por esta razón, surgieron las metodologías de desarrollo de *software* con propuestas para hacer el proceso más ordenado, predecible y eficiente.

Las primeras metodologías que se usaron para el desarrollo de *software,* como el modelo en cascada, eran secuenciales y surgieron de la búsqueda de experiencia de gestión en otras ingenierías. Sin embargo, los productos y la naturaleza de los procesos en el *software* son claramente distintos. Los modelos secuenciales potencializan los efectos negativos de los errores y los cambios provocando que su detección sea tardía; contribuyendo además a incrementar los costos de corrección y codificación. Otra desventaja de estas metodologías es que presentan procesos burocráticos que retrasan el ritmo del proyecto, y consumen tiempo y recursos en su extensa documentación.

En un proyecto de *software*, el riesgo aparece en forma de errores y de cambios, los cuales se deben a la evolución del entorno del usuario. Casi siempre esa evolución sucede tan rápidamente que los cambios se producen durante el tiempo de desarrollo del proyecto, provocando que los requerimientos de los usuarios cambien o se incrementen. Poco a poco, las metodologías incorporaron esta valoración de riesgos ya que los modelos secuenciales no proporcionan un soporte adecuado para estos entornos evolutivos.

Como una alternativa relativamente reciente a todos estos modelos de trabajo, nacieron las metodologías ágiles que buscan el equilibrio entre no contar con proceso alguno y contar con un proceso sobrecargado (Fowler, 2004). Las metodologías ágiles están menos orientadas a la documentación, y más centradas en la programación; son orientadas a la gente y no a los procesos; y son adaptables en lugar de predictivas.

1.4 Enfoques metodológicos ágiles

Las metodologías ágiles de desarrollo de *software* (Larman, 2004) permiten la construcción rápida de *software* funcional mediante la adopción de modelos iterativos e incrementales en donde se intercalan actividades de análisis, diseño y construcción tomando como base los principios comunes de agilidad

en el proceso (Kendall & Kendall, 2013). Al ser iterativas, las metodologías ágiles funcionan con ciclos repetitivos de trabajo de corta duración; al ser incrementales, los ciclos terminan siempre con un *software* de mayor valor que el ciclo anterior para acercarse cada vez más a una versión estable y funcional con la que el usuario pueda interactuar.

Una metodología ágil está enfocada al cliente, es adaptable al cambio y no exige muchos requerimientos de documentación. Las metodologías ágiles comenzaron a emerger en la década de los noventa, como propuestas para desarrollar sistemas rápidamente en ambientes cambiantes, manteniendo la calidad y el control de costos. Estos enfoques pretenden aportar medios para lograr que los proyectos de *software* tengan éxito, pues se sabe que todavía en los últimos años un gran número de proyectos fracasa debido a problemas de gestión del proceso o de comunicación entre el equipo de trabajo. Aunque las metodologías ágiles no solucionan todas las problemáticas en el desarrollo de *software,* sí proporcionan diversos procesos para trabajar más rápidamente y de manera más adaptable (Maquen, Chayan & Reyes, 2017).

Existen muchas metodologías ágiles, y aunque cada una es diferente, todas poseen principios comunes, los cuales pueden ser consultados en el "Manifiesto para desarrollo ágil de software" (Agile Alliance, 2018). Los principios ágiles se abordarán en el siguiente sub-apartado de este trabajo.

Algunas metodologías ágiles son: FDD *(Feature Driven Development)*, *Scrum*, *Crystal*, ASD *(Adaptive Software Development)*, DSDM *(Dynamic Systems Development Methodology)*, y XP *(Programación Extrema o Extreme Programming)*.

FDD fue desarrollado por Jeff De Luca y Peter Coad. En esta metodología las iteraciones duran dos semanas. El FDD tiene cinco procesos. Tres de ellos se realizan al inicio del proyecto (Desarrollar un modelo global, Construir una lista de los rasgos, Planear por rasgo) y los dos restantes se realizan en cada iteración (Diseñar por rasgo, Construir por rasgo). Cada proceso debe dividirse en varias tareas y debe contar con un indicador de comprobación.

Scrum (Sims & Johnson, SCRUM: *Abreathtakinly Brief and Agile Introduction*, 2012) posee iteraciones o *sprints* de 30 días; antes de cada una se debe definir la funcionalidad que el equipo de desarrollo debe entregar. Durante la iteración se trabaja para estabilizar los requisitos. Todos los días el equipo se reúne 15 minutos en una junta llamada *Scrum* para discutir el trabajo del día siguiente.

Alistair Cockburn es el creador de la familia *Crystal*, donde se apoya el supuesto de que se deben aplicar diferentes metodologías a los distintos tipos de proyectos que existen. En este enfoque se analizan dos variables: la cantidad de personas que participan en el proyecto y el impacto que tendrán los errores.

Estas metodologías no son rígidas, por lo que su adopción resulta atractiva.

El ASD destaca la importancia del desarrollo adaptable. Posee 3 fases solapadas: especulación, colaboración y aprendizaje. Propone que las desviaciones en un ambiente adaptable guían hacia una solución correcta y enfatiza la manera de promover una actitud colaborativa y de aprendizaje en proyecto.

DSDM nació en Gran Bretaña en 1994 y creció considerablemente. Existe una organización dedicada a vender los manuales, cursos, certificaciones relacionadas con esta metodología. DSDM posee: un estudio de viabilidad y negocio para establecer la factibilidad, arquitecturas del sistema y un plan del proyecto. También existe un ciclo para la documentación de análisis, otro para el diseño operacional del sistema y un tercero para la implantación.

La Programación Extrema o XP (Beck & Andres, 2004) es una metodología ágil que ha tenido mucha aceptación entre algunos desarrolladores; y ha sido cuestionada por otros tantos. XP reúne elementos valiosos ya conocidos en ingeniería de *software*, administración y relaciones humanas y los interrelaciona para desarrollar sistemas con éxito en un entorno cambiante.

1.4.1 Principios ágiles de desarrollo de *software*

La agilidad es una forma de trabajo que en los últimos años ha ganado popularidad dentro de la ingeniería del *software* gracias a los buenos resultados que ha conseguido en la rápida creación de *software*. Cualquier proceso que se denomine ágil debe adoptar los principios definidos en el documento "Alianza ágil" (Agile Alliance, 2018), los cuales se explican a continuación:

1. **La más alta prioridad es satisfacer al cliente a través de la entrega rápida y continua de *software* con valor.** El cliente es la más alta prioridad, por lo que el *software* debe entregársele periódicamente, de manera oportuna y pertinente.
2. **Se aceptan los requerimientos cambiantes, incluso en etapas tardías del desarrollo.** En la agilidad se sabe que el software y sus requerimientos poseen una naturaleza cambiante, por lo tanto, se acepta este hecho en todas las etapas del proceso de desarrollo.
3. **El *software* funcional se entrega frecuentemente.** La forma de trabajo es iterativa, lo que significa que el desarrollo completo del *software* se divide en periodos cortos y repetitivos que permiten entregar el producto final en pocas semanas.
4. **Los responsables de negocio y los desarrolladores trabajan juntos durante todo el proyecto**. Los enfoques ágiles promueven la comunicación de clientes y desarrolladores. Esto provoca que la retroalimentación sea continua y que exista un entendimiento mutuo entre ambos equipos.

5. **Los proyectos se desarrollan en torno a individuos motivados.** Se reconoce que los equipos de trabajo se auto-organizan y están formados de seres humanos que requieren apoyo y confianza en la realización de sus tareas.
6. **El método más eficiente y efectivo de "comunicar" es la conversación cara a cara**. La agilidad privilegia la comunicación cara a cara, incluso cuando el trabajo es remoto. Por esta razón, una vídeo conferencia será preferida siempre sobre cualquier conversación de texto.
7. **El *software* funcionando es la medida principal de progreso**. El progreso del trabajo que realiza el equipo se realizará siempre tomando como indicador el *software* funcional que se obtenga y no con base en la documentación u otros productos generados.
8. **Los procesos ágiles promueven el desarrollo sostenible**. Se trabaja evitando las jornadas laborales extensas y los pesados ritmos de trabajo, pues generan tensión y disminuyen la productividad.
9. **La atención continua a la excelencia técnica y al buen diseño mejora la agilidad**. Durante todas las actividades del equipo de desarrollo se trabaja continuamente con buenas prácticas que permiten lograr un producto final de alta calidad.
10. **La simplicidad es esencial**. Se trabaja para lo que es necesario hacer en ese momento. Se dejan de lado todas las demás funcionalidades.
11. **Las mejores arquitecturas, requisitos y diseños emergen de equipos auto-organizados**. Los participantes del equipo de trabajo se organizan entre ellos para realizar sus tareas; este proceso se lleva a cabo en un ambiente de apoyo, aprendizaje e interacción.
12. **A intervalos regulares el equipo reflexiona sobre cómo ser más efectivo para a continuación ajustar y perfeccionar su comportamiento en consecuencia**. La reflexión sobre las formas de trabajo es una actividad fundamental que el equipo realiza para hacer ajustes y mejorar continuamente.

1.4.2 *Scrum*

Scrum es un marco de trabajo iterativo e incremental que puede utilizarse para desarrollar *software* mediante ciclos de trabajo llamados *Sprints*, los cuales tienen una longitud fija. Al principio de estos ciclos, el equipo selecciona los requisitos del cliente de una lista de prioridades; durante el *Sprint*, los requisitos seleccionados no cambian. Al final del *Sprint*, el equipo se compromete a completar su implementación.

En cuanto a la dinámica de trabajo, el equipo se reúne diariamente en sesiones cortas para revisar su progreso y ajustar lo necesario para el resto del proyecto. Al final de cada *Sprint*, el equipo se reúne con todos los interesados, y

demuestra lo que ha logrado. Durante estas reuniones se obtiene retroalimentación, que es contemplada para el siguiente ciclo de trabajo. Uno de los objetivos de *Scrum* es que el producto esté realmente terminado al final del proceso. Esto significa que el código debe estar integrado, probado y debe ser potencialmente entregable (Sutherland, J., 2017). La adaptación de los objetivos y prácticas resulta importante ya que se reconoce que en el desarrollo hay aprendizaje, riesgo y situaciones imprevistas que el equipo debe enfrentar. Los equipos pueden hacer frente a los problemas complejos de adaptación, al mismo tiempo que son capaces de entregar productos funcionales para el usuario final. Este enfoque de trabajo ha producido buenos resultados en entornos industriales y académicos y ha promovido la buena planificación de los proyectos (Castillo, 2018).

Un equipo *Scrum* está compuesto por un propietario del producto *(Product Owner)*, el Equipo de Desarrollo y el Maestro *Scrum (Scrum Master)*. El propietario del producto es quien representa a todos los participantes y se encarga de sacar el máximo provecho del trabajo que el equipo realiza; el equipo de desarrollo está integrado por personas que tienen los conocimientos técnicos necesarios para crear *software* a partir de un conjunto de necesidades. Finalmente, el *Scrum Master* es quien verifica que todos comprenden el proceso y sus actividades (Sutherland & Schwaber, 2017).

Los artefactos de *Scrum* son elementos que apoyan al equipo de desarrollo en sus labores. Sus nombres son los siguientes: *Product Backlog, Burndown* de entrega, *Sprint Backlog, y Sprint Burndown* y se explican a continuación:

a) ***Product Backlog.*** Es un listado que identifica y prioriza las características y funciones totales que se implementarán en el *software.*

b) ***Burndown de entrega.*** Este gráfico muestra, cada día, una nueva estimación de cuánto trabajo queda en total (medido en horas-persona) hasta que se terminen las tareas del equipo. Idealmente, se trata de un gráfico con pendiente descendente.

c) ***Sprint Backlog.*** Este documento expone las tareas en las que trabajará el equipo de desarrollo durante una iteración o sprint.

d) ***Sprint Burndown.*** Es una gráfica que muestra el trabajo que queda por hacer durante una iteración o sprint.

1.4.3. Programación Extrema (XP)

La Programación Extrema, *eXtreme Programming* o XP (Wells, 2018) es una de las metodologías ágiles más maduras y más referenciadas en la literatura, que promueve la colaboración y la comunicación con el cliente en el mismo lugar donde se lleva a cabo el desarrollo (Roque, Durán, López, y Mota, 2013). Este enfoque metodológico adopta una serie de prácticas orientadas directamente al desarrollo de *software* para priorizar a las personas que son parte del equipo de trabajo.

Una de esas prácticas que resulta fundamental es la programación por pares, la cual consiste en que dos personas trabajan en un mismo equipo de cómputo; cada una utiliza el teclado en diferentes momentos. Durante el proceso de desarrollo, ambos participantes se comunican, toman decisiones y complementan su trabajo. En las parejas no existe el rol de jefe o de subordinado y ambos deben estar de acuerdo en trabajar juntos.

Según mencionan Beck y Andres (2004) los programadores por pares realizan labores conjuntamente, clarifican ideas y avanzan cuando su pareja no puede hacerlo. Sin embargo, en este enfoque también se cuenta con momentos para trabajar individualmente; de esta manera, una persona puede desarrollar ideas por sí misma pero debe regresar para comentarlas con su pareja. Esto no aplica al código, el cual debe escribirse en pares todo el tiempo. Por su parte, los programadores pueden tomar varios descansos durante el día y las parejas pueden cambiar sus integrantes. En el siguiente capítulo se presenta la Programación extrema con mayor profundidad.

II.

PROGRAMACIÓN EXTREMA

2.1 Antecedentes

La Programación Extrema es un proceso ágil de desarrollo de *software* cuyas raíces se remontan a finales de los ochenta cuando Kent Beck y Ward Cunningham refinaron sus prácticas e ideas en numerosos proyectos. Sin embargo, fue hasta 1996 cuando la Programación Extrema dejó de ser una práctica informal para convertirse en metodología; esto ocurrió mientras Kent Beck revisaba el proyecto de nómina C3 para la compañía Chrysler (Fowler, 2004).

La Programación Extrema es ideal para equipos pequeños o medianos que desarrollan *software* cuyos requerimientos son cambiantes. De acuerdo con la definición de Kent Beck, es una disciplina eficiente, de bajo riesgo, flexible, predecible, científica y divertida para desarrollar *software* (Beck & Andres, 2004).

La mayoría de las ideas que componen la XP son antiguas prácticas de programación, administración y gerencia que han sido probadas por muchos años. La XP solo las reúne y se asegura de que se lleven a cabo correctamente ya que cada práctica compensa sus debilidades con las fortalezas de otras prácticas.

Con las metodologías de desarrollo tradicionales y rígidas, el riesgo y el cambio son importantes factores de fracaso en los proyectos. Con XP el riesgo se asume y a mayor riesgo, más adaptable resulta el proceso.

Existen cuatro variables de control en un proyecto de *software*: Tiempo, costo, calidad y alcance. XP hace énfasis en la importancia del alcance. Sugiere que el cliente fije las primeras 3 variables (tiempo, costo, calidad) y que el equipo de desarrollo maneje la cuarta (alcance). Permitiendo que cambie el alcance y utilizando las prácticas de XP se superan los problemas comunes de desarrollo. Permitir que el alcance cambie no significa que el proyecto tenga una longitud ilimitada; significa que si el cliente agrega una nueva característica al sistema, se debe eliminar alguna característica equivalente en tamaño, lo que permite el equilibrio en el proyecto. XP sugiere que la idea clásica del costo exponencial del cambio en un proyecto deja de ser válida si se utilizan las prácticas propuestas por esta metodología.

2.2 Los valores de la XP

Los valores sobre los que la Programación Extrema tiene sus cimientos son: Comunicación, sencillez, retroalimentación, valentía y respeto (Beck & Andres, 2004).

- **Comunicación**. XP adopta la comunicación como valor fundamental. Hace énfasis en la comunicación oral continua en lugar de hacer uso de documentos, reportes y planes elaborados. La comunicación fomenta el entendimiento para remover los obstáculos entre los involucrados en el proceso de desarrollo.
- **Sencillez**. "Hacer siempre lo más simple que pueda funcionar", "Implementar solo lo que el cliente necesita en el presente" y "Resolver los problemas de la manera

más sencilla posible" son algunas de las máximas de sencillez de XP. Tener en mente siempre los principios de sencillez durante el proceso permite implementar lo que el cliente necesita y no lo que él cree que puede necesitar.

- **Retroalimentación**. Obtener continuamente información concreta proveniente del cliente para asegurar la exactitud y calidad en el desarrollo. Se logra trabajando con el cliente, y produciendo *software* rápidamente para mostrárselo y obtener observaciones.
- **Valentía**. Implica tener confianza en el trabajo rápido y poder decidir en cualquier momento iniciar de cero el desarrollo si es necesario. La valentía se basa en la confianza de utilizar las pruebas y las herramientas automatizadas como base del desarrollo. La valentía debe estar soportada por los otros tres valores.
- **Respeto**. El respeto es un valor esencial que está relacionado con los otros cuatro mencionados previamente. El respeto está relacionado con la importancia que las personas le dan al proyecto y a los otros miembros del equipo. Todos tienen la misma importancia y se debe reconocer este valor en cada actividad que se realice.

2.3 Principios de la XP

Los principios de la XP ayudan a decidir entre las posibles alternativas que se presenten durante el desarrollo de *software*. Se prefiere una alternativa que cumpla con los principios, en lugar de otra que no lo haga.

2.3.1 Principios básicos

Los principios básicos engloban los valores de la XP, pero de una manera más concreta. Los principios básicos de la XP son: retroalimentación rápida, adoptar la sencillez, cambio incremental, aceptar el cambio, y trabajar con calidad.

- **Retroalimentación rápida**. El uso de iteraciones cortas permite evaluar si el producto cumple con las necesidades del sistema.
- **Adoptar la sencillez**. La simplicidad en las soluciones se logra al aplicar las máximas de sencillez a todos los problemas a resolver, centrándose en una iteración a la vez.
- **Cambio incremental.** En la planeación, desarrollo y diseño se prefiere una serie de pequeños cambios graduales en lugar de un solo cambio abrupto.
- **Aceptar el cambio**. La XP soluciona los problemas, al mismo tiempo que mantiene el desarrollo siempre abierto a nuevas posibilidades.
- **Trabajar con calidad**. Sobre cualquier situación se opta siempre por realizar trabajo de calidad, el cual se fundamenta en la realización de pruebas.

2.3.2 Principios secundarios

Los principios secundarios ayudan a tomar decisiones en situaciones más específicas y son: enseñar a aprender, iniciar con una pequeña inversión, jugar a ganar, experimentos concretos, comunicación

abierta y sincera, trabajar con los instintos de las personas, no contra ellos, responsabilidad aceptada, adaptación local, viajar con poco equipaje y métricas sinceras.

- **Enseñar a aprender**. Es más importante enseñar estrategias para aprender que dictar mandamientos inflexibles para seguir.
- **Iniciar con una pequeña inversión**. Se prefiere iniciar con pocos recursos e incrementarlos gradualmente. Demasiados recursos desde el principio pueden ser desastrosos para el proyecto.
- **Jugar a ganar**. Se refiere a adoptar la actitud de "jugar para ganar" en lugar de "jugar para no perder".
- **Experimentos concretos**. Cada decisión que se tome debe ser probada, para asegurarse que no es equivocada y que no incorpora riesgos adicionales al proyecto. El resultado de una sesión de diseño debe ser una serie de experimentos, no un diseño terminado.
- **Comunicación abierta y sincera**. Los desarrolladores deben ser libres para expresar sus sentimientos, señalar problemas y pedir ayuda sin ser castigados.
- **Trabajar con los instintos de las personas, no contra ellos**. Se basa en adoptar prácticas con las que las personas se sientan integradas para aprender e interactuar. Las prácticas adoptadas deben resolver problemas rápido y colaborar con los intereses de las personas a corto plazo.
- **Responsabilidad aceptada.** Es preferible que las personas acepten su propia responsabilidad en lugar de que ésta les sea impuesta. Esta aceptación debe darse en conjunto con el hecho de que los integrantes son parte de un equipo que toma decisiones para el bien del proyecto.
- **Adaptación local**. La XP se puede adaptar a las situaciones particulares de cada equipo de desarrollo. Puede verse como una responsabilidad aceptada para el proceso a adoptar.
- **Viajar con poco equipaje**. La manera más eficiente de avanzar rápido es viajar con poco equipaje. El equipaje debe ser ligero, simple y valioso. Esta analogía proporciona una reflexión importante para el viaje del desarrollo, para el cual se debe estar preparado en todo momento ya sea para migrar a un nuevo diseño, para adaptarse a nuevos integrantes del equipo, o a una nueva tecnología.
- **Métricas sinceras**. Se prefiere utilizar medidas que estén al alcance y entendimiento de todos, en lugar de métricas complejas que pueden resultar ineficientes e inflexibles.

2.4 Prácticas de la XP

Las prácticas de la XP son sencillas pero poderosas, especialmente cuando se usan en conjunto y de manera correcta. Existe una sinergia entre ellas, que permite que las fortalezas de unas apoyen las debilidades de otras.

2.4.1 Juego de planeación *(Planning game)*

El juego de planeación puede verse como una reunión, o un punto importante de interacción entre clientes y desarrolladores en un ambiente de confianza mutua. Para conducir esta práctica se recomienda utilizar tarjetas de archivo en blanco; en ellas se escriben las historias de usuario, que son acciones que el cliente desea que el sistema sea capaz de hacer. Aunque las tarjetas de archivo pudieran parecer rudimentarias, han demostrado ser elementos económicos, eficaces y comprobados para acercar a los clientes y desarrolladores.

La meta del juego de planeación es maximizar el valor del *software* que produce el equipo, al mismo tiempo que se obtienen beneficios considerables sin realizar grandes inversiones de recursos. En este juego se pueden distinguir 3 fases importantes: Exploración, compromiso y dirección.

En la fase de Exploración el cliente escribe sus historias de usuario; los desarrolladores estiman el tiempo ideal de implementación y si una historia es muy grande, se divide en varias más pequeñas para poder realizar la estimación.

En la fase de compromiso las historias son ordenadas por el cliente de acuerdo con su valor de negocios; y por los desarrolladores según su claridad para estimarlas. El equipo de desarrollo debe comunicar al cliente su velocidad de desarrollo en tiempo ideal por mes; para que el cliente elija el conjunto de tarjetas ya sea determinando una fecha o escogiendo las historias deseadas.

La fase de dirección permite actualizar la planeación conforme se va realizando el desarrollo. Al principio de cada iteración el cliente selecciona las historias que se deben implementar; si el equipo de desarrollo considera que la velocidad no ha sido estimada correctamente, puede solicitar que el cliente realice ajustes con base en los nuevos parámetros fijados. El cliente puede escribir una historia nueva, la cual será estimada por el equipo de desarrollo; el cliente puede entonces remover una o varias historias con tiempo estimado equivalente para poder agregar su nueva historia en esa iteración. Si el equipo de desarrollo lo cree pertinente, puede reconsiderar la velocidad de desarrollo de todas las historias restantes y ajustar de nuevo el plan.

El juego de planeación también se lleva a cabo dentro de las iteraciones; en este caso son los programadores quienes planean sus actividades con reglas muy similares y con las mismas 3 fases. Las tarjetas de archivo también se utilizan, solo que en esta ocasión para escribir tareas en lugar de historias.

En la fase de exploración las historias de la iteración se dividen en varias tareas, y una tarea a su vez puede dividirse en otras más pequeñas.

En la fase de compromiso, cada programador se hace responsable de algunas tareas, y es hasta después cuando establece el tiempo ideal estimado para completarlas. Cada programador debe establecer además su propia capacidad máxima de desarrollo

en días ideales de trabajo para decidir si las tareas aceptadas pueden ser completadas en el tiempo estimado, o debe delegar algunas de estas tareas a otros programadores.

En la fase de dirección cada programador toma una tarjeta con una tarea, y junto con un compañero escribe las pruebas para la tarea, las hace funcionar, integra y entrega su código al repositorio común y se asegura que todas las pruebas hayan sido superadas satisfactoriamente. También se lleva un registro del progreso individual de cada programador, y si alguien se encuentra con demasiadas tareas puede reducir el alcance de ellas, eliminar tareas innecesarias, o bien, hablar con el cliente para acortar el alcance de algunas historias o para posponerlas para la siguiente iteración.

2.4.2 Versiones pequeñas *(Small releases)*

Implica implementar la menor cantidad posible de historias de usuario a la vez. A pesar de su corto tamaño, es importante que cada versión tenga un sentido completo y propio, y que entregue un valor de negocios importante para el cliente. Durante la planeación de estas entregas, el cliente y el equipo de desarrollo definen las historias de usuario y el orden de su implementación. El equipo de desarrollo puede utilizar cada entrega como un punto de verificación en el que se puede medir la precisión de lo realizado comparando lo planeado con lo realizado; de esta manera se adquiere experiencia para la siguiente iteración.

2.4.3 Metáfora *(Metaphor)*

Permite establecer un conjunto común de términos para la comunicación en el proyecto. Una buena metáfora es una herramienta poderosa para unificar el lenguaje técnico y de negocios, y ayuda a entender los elementos básicos del proyecto y las relaciones entre ellos. Una metáfora hace que la tecnología pueda ser mejor comprendida por el cliente, por consecuencia, se sienta más cómodo y tienda a involucrarse de manera activa en el proceso.

2.4.4 Diseño simple *(Simple design)*

La estrategia que se utilice debe ser simple por sí misma y conducir al diseño más simple posible que sea consistente con las metas globales del proyecto y sea fácil de comunicar y entender. El diseño siempre está sujeto a cambios, y contiene solo lo necesario y nada más. Se evita en todo momento la complejidad innecesaria. De acuerdo con Kent Beck, creador de XP, un programa que posee un diseño simple es aquel que pasa todas las pruebas, no contiene código duplicado, refleja claramente lo que el programador planeó hacer y contiene la menor cantidad de elementos tales como clases y métodos.

En XP el diseño es un proceso continuo que ocurre en el nivel conceptual y físico. Dos premisas que describen el diseño simple son: “Diseñar y construir tan

simple como sea posible para que funcione y pase las pruebas" y "Hacer solo lo que se necesita en este momento".

XP no excluye la posibilidad de realizar el diseño a través de métodos gráficos; sin embargo estas ayudas visuales se deben convertir en código funcional lo más pronto posible y se recomienda no guardarlas ni invertirles demasiado tiempo.

2.4.5 Pruebas *(Testing)*

En XP los programadores escriben las pruebas antes de escribir el código a probar. Las pruebas brindan confianza al equipo de desarrollo y proporcionan la base para la refactorización. Se debe escribir una prueba y conservarla para todas aquellas situaciones que puedan desencadenar un error en producción. De esta manera, una vez escritas las pruebas ya es posible realizar el código necesario y suficiente para que sean superadas. El trabajo continúa hasta que se pasan todas las pruebas. Después, es necesario integrar el código en el repositorio global del equipo de desarrollo y ejecutar ahí las pruebas nuevamente.

Las pruebas deben ser independientes, es decir, no deben interactuar con otras. Deben realizarse con un entorno automatizado especial que permita ejecutarlas de manera rápida después de que la integración se ha llevado a cabo.

También el cliente escribe pruebas para cada historia de usuario. En ellas se especifican los escenarios que deben verificarse para asegurar que la historia se implementó correctamente. La mayoría de las veces los usuarios necesitan ayuda para escribir estas pruebas; para lo cual puede existir en el equipo una persona dedicada a brindar este soporte.

2.4.6 Refactorización *(Refactoring)*

Es la técnica de mejorar el código sin cambiar su funcionalidad para eliminar duplicidad, hacerlo más entendible, simplificarlo o hacerlo más flexible.

La refactorización puede verse como un proceso continuo de simplificación que se aplica al código, al diseño, a las pruebas y a la misma XP. De esta manera los programadores realizan refactorización durante todo el proceso de desarrollo.

2.4.7 Programación por pares *(Pair programming)*

Consiste en que dos desarrolladores comparten una computadora para realizar su trabajo, con lo que se logra que todas las decisiones de diseño sean tomadas por dos personas y que por lo menos dos personas estén siempre involucradas con esa parte del sistema. El intercambio de integrantes además fomenta la expansión del conocimiento con otras parejas en el equipo. Con la programación por pares existen menos probabilidades de que algunas pruebas sean pasadas por alto, pues

todo el código se revisa todo el tiempo. Mientras un programador trabaja a nivel de detalle, el otro trabaja a nivel conceptual.

La productividad en este esquema de trabajo es más alta que en el hipotético escenario donde el mismo trabajo se dividiera para que los dos programadores hicieran su parte individualmente y luego lo integraran. El código resultante, además, posee una mayor calidad.

Debe tenerse cuidado de no caer en la práctica de que una persona programe y otro solo observe. La programación por pares debe pensarse como una comunicación a muchos niveles entre dos personas. Si entre dos programadores existen rencillas, ambos pueden trabajar con parejas distintas. Si algún programador no desea trabajar en parejas, debe aprender a hacerlo o abandonar el equipo de desarrollo, ya que la programación por pares es una de las prácticas más importantes en XP, pues muchas otras dependen de ella para funcionar.

2.4.8 Propiedad colectiva de código *(Collective ownership)*

Cualquier persona puede cambiar el código en cualquier momento. Se basa en el pensamiento de que si una persona puede cambiar todo el código, entonces también es responsable de todo el código. Para que esta práctica no resulte en un desorden colectivo, es necesario que sea utilizada en conjunto con otras prácticas como la programación por pares, pruebas y estándares de codificación.La propiedad colectiva de código promueve el conocimiento del sistema con todos los miembros del equipo, y permite que el código complejo sea simplificado en corto tiempo por otros programadores.

2.4.9 Integración continua *(Continuous integration)*

Se trata de integrar los componentes de *software* tan frecuentemente como sea posible y por lo menos una vez al día en una computadora especial para tal propósito. Al momento de realizar la integración, se deben ejecutar las pruebas otra vez y asegurarse que todas fueron superadas. Para esto, se pueden utilizar herramientas automatizadas que faciliten la agilidad de este proceso. La integración continua permite que los conflictos entre el código de los diferentes programadores sean visibles en corto tiempo.

2.4.10 Semana de trabajo de 40 horas *(40-Hour week)*

No se puede mantener un nivel de calidad aceptable cuando se mantienen cargas de trabajo extremas. Además, la sola presencia de esas pesadas cargas de trabajo es un indicador de serios problemas en el proyecto. Al trabajar horas extras diariamente durante largos periodos de tiempo, el número de defectos aumenta, la comunicación se entorpece y la calidad disminuye.

La medida de "40 horas a la semana" no es restrictiva; el equipo de desarrollo puede acordar trabajar más o menos horas, siempre y cuando sea capaz de realizar un trabajo de calidad durante todo ese tiempo. Sin embargo, como regla general, no se debe trabajar tiempo extra por más de una semana.

2.4.11 Cliente en sitio de desarrollo *(On-Site customer)*

Es importante que el cliente se encuentre en el mismo sitio donde el desarrollo se lleva a cabo y permanezca ahí durante todo el ciclo de vida del proyecto; esto permite contar con su participación y su retroalimentación en todas las actividades. El cliente puede seguir realizando sus tareas de negocio cotidianas, siempre y cuando se encuentre separado de su lugar de trabajo habitual, se mantenga cercano a los desarrolladores y tenga la disponibilidad para involucrarse en las diversas tareas del proyecto.

2.4.12 Estándares de codificación *(Coding standards)*

Se refiere al uso de reglas, convenciones, recomendaciones y buenas prácticas de programación que deben ser adoptadas por el equipo de desarrollo. El uso de estándares de codificación garantiza la producción de software con un estilo uniforme, sin importar el programador que lo codificó. Existen muchos estándares, pero los programadores decidirán cuáles adquirir y se comprometerán a aplicarlos en todo momento.

2.5 Ciclo de un proyecto con XP

Un proyecto desarrollado con XP atraviesa una fase inicial corta de desarrollo, seguido de años de producción, soporte y refinamiento simultáneo; cuando el proyecto deja de tener sentido, se procede con la fase de terminación.

La premisa fundamental durante el ciclo de vida de XP es que el cliente y el desarrollador deben trabajar juntos para producir *software* valioso. Las seis fases de un proyecto en XP son: exploración, planeación, iteraciones, producción, mantenimiento y muerte del proyecto (Anwer, Aftab, Muhammad Shh & Waheed, 2017). A continuación se explica cada una de ellas.

2.5.1 Exploración

En esta fase la principal actividad es descubrir los requerimientos del sistema. El cliente comunica de manera general lo que el sistema debe hacer; para lo cual practica el uso de las historias de usuario, que son similares a los casos de uso y están escritos en un lenguaje no técnico; la longitud máxima de las historias de usuario es de 3 a 4 enunciados.

Durante esta fase se realiza la metáfora, que es la forma en la que el equipo conceptualiza el sistema. Está redactada con lenguaje del dominio del problema.

Los desarrolladores realizan estimaciones aproximadas acerca del tiempo para implementar las historias de usuario, y de las tareas de programación que realizan. Estas estimaciones se usan como práctica; pueden variar en las siguientes iteraciones, y se irán refinando con el avance del proyecto.

En esta fase los programadores utilizan y aprenden toda la tecnología que se usará cuando el sistema esté en producción, y realizan experimentos con ella; también exploran diferentes opciones para la arquitectura del sistema, y pueden realizar simulaciones para probar los escenarios menos favorables.

En ocasiones los desarrolladores necesitan invertir tiempo para entender aspectos específicos del planteamiento del problema; durante todo este tiempo investigan teniendo siempre metas específicas acerca de lo que desean conseguir. Es común que si se ha generado código en esta etapa, se descarte pues carece de la calidad necesaria para reutilizarlo.

El resultado de la fase de exploración es una metáfora del sistema y una primera lista de las historias del usuario. Al término de esta etapa, el cliente debe estar seguro que existe suficiente información en las historias de usuario para realizar una primera entrega; los programadores deben estar seguros que sus estimaciones son las mejores que pueden realizar con la información que tienen.

2.5.2 Planeación

Es una fase muy corta, de un día o dos, en la que los clientes y los desarrolladores se ponen de acuerdo en ciertas características para la entrega del sistema en una fecha determinada. Las características están contenidas en las historias del usuario, las cuales se irán implementando de manera coherente con aspectos técnicos y de negocios. La etapa de planeación involucra mucha negociación por parte de clientes y desarrolladores. Por ejemplo, algunas responsabilidades del cliente en la fase de planeación son: definir las historias de usuario, decidir el valor de negocios de cada una y decidir cuáles de estas historias serán incluidas en la entrega del sistema. El desarrollador debe estimar el tiempo de implementación de cada historia de usuario, prevenir al cliente sobre posibles riesgos técnicos y medir el progreso del equipo.

El juego de planeación es la herramienta principal para esta etapa. Implica que clientes y desarrolladores trabajen en conjunto con las historias del usuario para estimar y dar prioridad al trabajo. El resultado de esta fase es una fecha de entrega, no es una lista de tareas estructuradas.

2.5.3 Iteraciones

Esta fase consiste en dividir el alcance total del sistema en iteraciones. Cada iteración es un ciclo o repetición de actividades que tiene una longitud entre una y cuatro semanas. La primera iteración se centra en la arquitectura de los componentes requerida para construir un sistema básico. A partir de las siguientes iteraciones, el sistema debe contener algún valor de negocios que continuará creciendo con el desarrollo del proyecto. Cada iteración es corta con la finalidad de reducir los riesgos, y corregir los aspectos necesarios a la mayor brevedad. La división en iteraciones se realiza en una junta a la que asisten desarrolladores y clientes. Los clientes seleccionan las historias de uso del plan general y les dan prioridad según el valor de negocios de cada una. También se seleccionan las pruebas de aceptación que no hayan sido superadas en iteraciones anteriores.

El siguiente paso es dividir la iteración en tareas a realizar para implementar cada historia de usuario. La duración típica para una tarea va de uno a tres días ideales de programación; si resultan tareas que requieran más de ese tiempo, se deben dividir en varias tareas más pequeñas. Los desarrolladores se anotan para realizar las tareas y estiman el tiempo requerido para completarlas. Es conveniente que el mismo programador que hizo la estimación sea el que realice la tarea.

Durante el desarrollo, los programadores inician el trabajo de una nueva tarea escribiendo primero las pruebas y agregándolas al entorno de pruebas automatizado que poseen. Dos programadores comparten una misma computadora. Conforme se van completando las clases y componentes, éstas se van integrando en una computadora dedicada sólo a unificar el trabajo de todos. Durante la iteración, los programadores utilizan integración continua, ayudados por marcos de integración automatizados. Al final de la iteración, los clientes realizan las pruebas de aceptación que escribieron y cualquier historia de uso que no las supere, será ajustada en la siguiente iteración. Se recomienda realizar un pequeño festejo al término de cada iteración en donde el equipo convive y celebra el trabajo completado hasta ese momento.

2.5.4 Producción

En esta fase, el producto se verifica y se certifica para ser utilizado en el ambiente de producción real del cliente. Esta verificación se puede llevar a cabo también en un ambiente de producción simulado que debe reflejar el ambiente real del cliente. En este punto, la meta no es continuar con la evolución funcional, sino solo estabilizar el sistema realizando los ajustes necesarios.

Los usuarios realizan ciertas pruebas de aceptación en el producto; si se detectan situaciones erróneas o anómalas, se deben corregir; y hasta entonces la entrega se considerará completa. En teoría, cualquier sistema desarrollado

utilizando programación extrema puede ser llevado a producción desde la primera iteración. Cualquier documentación que el equipo de desarrollo realiza, se debe ajustar siempre a la filosofía de XP: "Debe ser suficiente pero no más". Haber llegado a esta fase también es motivo de celebración para el equipo de desarrollo y se recomienda realizar una fiesta o convivio.

2.5.5 Mantenimiento

Mantenimiento es el estado normal de un proyecto en XP. Esta es una fase progresiva, en donde se debe tener un plan para realizar las actualizaciones y cambios al sistema. Se deben tener consideraciones especiales con los cambios que se realizan al código, los datos que se modifican, la nueva tecnología que se incorpora al proyecto, así como el tiempo y el personal dedicado al mantenimiento. Debe existir un ambiente de confianza para realizar de manera segura y sencilla las modificaciones. Esta confianza se deriva de herramientas automatizadas de programación que se han utilizado a lo largo del desarrollo.

2.5.6 Muerte del proyecto

Kent Beck, autor de la XP menciona en su libro: "Morir bien es tan importante como vivir bien. Esto se aplica tanto a la XP como a las personas". La muerte del proyecto puede darse debido a que el usuario ya no tiene más historias para implementar, en cuyo caso se debe elaborar un reporte de 5 a 10 páginas para hacer más viables las modificaciones en un futuro, si existieran. Otras razones, no tan buenas, para que el sistema muera son: porque el sistema ya no es factible, no hay presupuesto para continuar, o tal vez porque los defectos son tantos que no conviene seguir adelante. La fase de muerte también sirve al equipo para aprender acerca del proceso de desarrollo y se considera también motivo suficiente para celebrar el fin de un ciclo.

2.6 Programación por pares como tema de investigación

La programación por pares, al ser una de las principales prácticas de la XP, ha sido abordada como tema de investigación en varios trabajos. A continuación se mencionan algunos de ellos.

Zhong, Wang, Chen y Li (2017) estudiaron el tiempo de cambio de roles en la programación por pares en estudiantes jóvenes. Éste es un aspecto poco abordado en la literatura. Sus resultados indican que los participantes disfrutaron más esta forma de programar cuando el cambio de roles se realizaba de forma libre y no en tiempos fijos.

Yang, Lee y Chang (2016) encontraron que la programación por pares eleva tanto la motivación de aprender como el nivel de retención en los alumnos que la utilizan en cursos de estructuras de datos.

En el trabajo de Plonka, Sharp, van der Linden y Dittrich (2015) se destaca la programación por pares como una actividad relevante y se estudia la transferencia de conocimiento en este tipo de programación. También se analizan casos concretos y se identifican seis tipos de transferencia de conocimiento entre los participantes.

La investigación de Prabu y Duraisami (2015) reconoce que la programación por pares ha sido el tema central de muchas investigaciones, sin embargo, poco ha sido abordada desde el entorno académico. En este trabajo se encontró que aunque esta técnica de programación tiene algunas desventajas, puede ser muy benéfica ya que promueve el deseo de trabajar de manera colaborativa y aumenta la percepción sobre la buena organización y el ahorro de tiempo.

El estudio de da Silva y Prikladnicki (2015) aborda la programación por pares en equipos de trabajo con integrantes en ubicaciones geográficas distintas a través de una revisión bibliográfica. Se encontró que aunque existen varias plataformas para lograr esta interacción, poco han sido evaluadas en la práctica.

Coman, Robillard, Silliti y Succi (2014) mencionan la falta de consenso en los resultados de investigaciones relacionadas con la programación por pares. El objetivo de este trabajo fue encontrar los mejores usos de esta práctica. Sus principales aportaciones son la diferenciación de interacciones entre los participantes de los equipos y un análisis de los escenarios en los que esta técnica podría ser benéfica o no recomendada.

Lazzarini y su equipo (Lazzarini Lemos, Cutigi Ferrari, Fagundes Silveira & Garcia, 2012) estudiaron dos prácticas ágiles desde escenarios académicos e industriales: la programación por pares y la denominada "probar-primero", ambas propuestas por el enfoque ágil de la Programación Extrema. Al aplicarlas, se encontró que aunque la calidad del *software* desarrollado aumentó, el tiempo de desarrollo se incrementó también. Los resultados fueron los mismos para los entornos académico e industrial.

En el trabajo de Swamidurai y Umpress (2012) se estudian las ventajas y desventajas de la programación por pares y se propone una alternativa basada en esta técnica, la cual se denomina *Collaborative-Adversarial Pair Programming* e intenta aprovechar sus beneficios y minimizar sus problemáticas.

Zacharis (2011) presenta una investigación que implementó el uso de la Programación por pares virtual en un curso de programación utilizando herramientas en línea. En el trabajo se realizaron mediciones acerca del desempeño y la satisfacción de los participantes. La programación por pares produjo resultados positivos para recomendarla como alternativa a la programación en solitario.

La investigación de Balijepally, Mahapatra, Nerur y Kenneth (2009) compara la programación por pares con la programación en solitario en aspectos relacionados con el desempeño y las respuestas afectivas de los participantes. Para los experimentos, se manipuló la complejidad de las tareas realizadas por ambos

grupos y se encontró que el trabajo por pares muestra un buen rendimiento, pero no supera el rendimiento del mejor de sus integrantes en solitario.

Dawande, Johar, Kumar y Mookerje (2008) realizó una comparación de la programación por pares con la programación en solitario buscando minimizar el esfuerzo y tiempo a través de un enfoque analítico. También hizo recomendaciones para aplicar una u otra forma de trabajo según los resultados que se obtuvieron.

La investigación de Mohd Zin, Idris y Kuman Subramaniam (2006) expone experiencias y resultados de la implementación de la programación extrema en entornos virtuales dentro de un ambiente de aprendizaje a distancia. Los resultados muestran que esta combinación produjo resultados positivos en cuanto a la eficiencia, motivación y confianza de los estudiantes.

2.7 Reflexiones finales sobre XP

La Programación Extrema es una metodología ágil de desarrollo de *software* que aporta una manera dinámica e interactiva de producir *software* de alta calidad en ambientes donde los requerimientos cambian frecuentemente. Es importante subrayar que XP no es exclusiva para desarrollar algún tipo de proyecto en particular, sino que puede ser aplicada a proyectos de cualquier naturaleza como por ejemplo, de escritorio, móvil o web.

La filosofía de XP encuadra valores que se promueven mediante prácticas con acción sinérgica, las cuales han demostrado ser eficaces en otras metodologías de desarrollo y administración. Cada práctica posee fortalezas y debilidades; cada una compensa las debilidades de otras; por eso es importante su uso conjunto para evitar el caos.

Los elementos fundamentales que forman el eje de la XP son el cliente y los programadores que trabajan en conjunto en todo momento y desarrollan una comunicación abierta en beneficio del producto final. XP promueve además la simplicidad en todos los aspectos de desarrollo e incrementa la confianza que el equipo tiene en el trabajo realizado.

Al tratarse de una metodología ágil, XP acelera la producción de *software*, haciendo poco énfasis en la documentación, centrándose en las personas y promoviendo el intercambio efectivo de ideas. También fomenta la convivencia entre los integrantes del equipo; y los hace parte del festejo de los logros alcanzados.

En XP se escriben las pruebas antes de realizar el código, y el trabajo se divide en iteraciones cortas que permiten obtener retroalimentación valiosa para dirigir el proyecto. Dos programadores comparten un solo equipo de cómputo, en donde realizan su trabajo y luego lo integran en otra computadora donde se encuentra el repositorio universal. Cada equipo de programadores es responsable

de que sus programas pasen la totalidad de las pruebas en su equipo (y también en la computadora donde se encuentra el repositorio). En XP el cliente se encuentra en el mismo sitio donde se lleva a cabo el desarrollo del sistema, proporcionando retroalimentación de primera mano. Es el uso de todas estas prácticas lo que hace de la Programación Extrema una metodología controversial.

Las opiniones acerca de XP se dividen. Los desarrolladores a favor ponderan el hecho de que el esfuerzo se encamina a realizar rápidamente código de alta calidad y se pronuncian a favor de la comunicación y participación de todos los involucrados. Además consideran que el enfoque humano es factor primordial para el éxito de cualquier desarrollo de sistemas. Por el contrario, otros desarrolladores cuestionan la XP argumentando que su filosofía se basa en suposiciones y aseveraciones empíricas; por lo que aún hacen falta estudios sólidos que demuestren su eficiencia y respalden sus fundamentos en escenarios específicos. También critican la orientación hacia la generación de código y el alejamiento de los convencionalismos en cuanto a la documentación.

Las personas son muy relevantes en XP; sin embargo, las relaciones humanas aún resultan difíciles de comprender y manejar. Mientras existen programadores que trabajan en equipo de manera natural, hay otros que parecieran haber nacido para trabajar solitariamente. Las inconveniencias como el demasiado tiempo invertido para la coordinación de tareas, la incompatibilidad entre los miembros del equipo y la obstaculización de actividades pueden presentarse cuando programadores con distintas personalidades de ambos grupos deben trabajar juntos en un ambiente de colaboración como el propuesto por XP. Estas inconveniencias han sido superadas en varios equipos de desarrollo ya sea al tomar solo lo mejor de XP, o bien, al adaptar las prácticas a las necesidades particulares de los participantes. En cada equipo donde se desee adoptar la XP, se deben considerar las condiciones y recursos locales para decidir si esta metodología es la más apropiada para trabajar.

III.

EXPERIENCIAS DE INVESTIGACIÓN CON DESARROLLO ÁGIL UTILIZANDO PROGRAMACIÓN EXTREMA Y *SCRUM*

3.1 Primer caso de estudio: Desarrollo de aplicaciones móviles con Programación Extrema y *Scrum*

3.1.1. Introducción

De acuerdo con la literatura, el desarrollo de aplicaciones móviles puede verse favorecido con las metodologías ágiles, pues el producto final de este tipo de proyectos debe estar disponible en el mercado en plazos cortos y además, sus requerimientos suelen ser muy cambiantes, característica que la agilidad promete manejar con buenos resultados. Existen varias metodologías ágiles que podrían utilizarse; dos de las más maduras y referenciadas en la literatura son XP y *Scrum*: La primera, enfocada en prácticas orientadas a la programación y la segunda, orientada a la coordinación del trabajo en equipo. ¿Pueden estos dos enfoques ágiles favorecer el desarrollo de aplicaciones móviles en proyectos de extensión pequeña a mediana con un corto tiempo de entrega y con programadores cuyo rango de experiencia es principiante? ¿Cuál de las dos metodologías resultaría más apropiada para los desarrollos de aplicaciones con esas características utilizando en concreto el lenguaje *Java* para el sistema operativo *Android*? Como una primera aproximación a la respuesta a estas preguntas, se diseñó un caso de estudio que permitiera realizar una comparación entre las dos metodologías. El principal objetivo fue recabar evidencias prácticas en torno a la aplicación de XP y *Scrum* al desarrollo de aplicaciones móviles. En este trabajo se presentan las experiencias y lecciones aprendidas de este caso desde un enfoque cualitativo. Las perspectivas cuantitativas y otros resultados relevantes se presentan en el trabajo de Roque, Herrera, López y Salinas (2017). Para esta investigación se pidió a los participantes que identificaran las principales ventajas y problemas que tuvieron en su trabajo con las metodologías ágiles. También se pidió al equipo de investigación que externara sus opiniones sobre el desempeño de ambos grupos. Un usuario externo ajeno a este caso de estudio, realizó una prueba de usabilidad al *software* construido por cada equipo y opinó acerca de cada uno.

3.1.2. Antecedentes: Caracterización breve de XP y *Scrum* para el estudio realizado

XP y *Scrum* son metodologías ágiles que privilegian la obtención de *software* funcional sobre la generación de documentación y entregables intermedios. *Scrum* resalta valores y prácticas basados en la administración de proyectos y está enfocado en la auto organización de los equipos, así como en sus mediciones diarias; XP enfatiza la colaboración y la comunicación con el cliente y entre el equipo mismo.

Entre las diferencias más notables se puede destacar que *Scrum* cuenta con iteraciones de entrega denominadas *Sprint* que constan generalmente de cuatro

semanas para su conclusión, mientras que en XP, es recomendable que las iteraciones de entrega sean más rápidas y comprendan de una a dos semanas como máximo. En estas metodologías se definen marcos de trabajo que determinan la dinámica del desarrollo. En *Scrum*, el equipo adopta sus propias prácticas de programación de manera autónoma y no se sigue un proceso prescriptivo sino que se otorga libertad al equipo de desarrollo para alcanzar los objetivos que se han establecido.

En XP, existen algunas prácticas que se recomienda adoptar para producir de inmediato *software* de buena calidad; por ejemplo, la programación por pares, en donde dos personas comparten un mismo equipo de cómputo para codificar. También, todas las actividades se deben realizar dentro de la jornada establecida para no llevar trabajo a casa.

Al final de cada iteración, el equipo participa en una celebración en donde todos conviven. El cliente debe estar presente todo el tiempo en el lugar del desarrollo y retroalimentar a los desarrolladores.

En *Scrum*, se enfatizan más los roles y las reuniones que las prácticas específicas. Existe un propietario del producto, quien se asegura que las necesidades del cliente son comprendidas por el equipo, y un maestro *Scrum*, quien actúa como facilitador para las situaciones problemáticas y para la metodología misma. Por su parte, los integrantes del equipo son los que realizan las actividades concretas del desarrollo. En cuanto a las reuniones, se cuenta con el denominado *Scrum* diario, en donde los participantes comparten lo que han hecho, los obstáculos que han tenido y lo que esperan realizar; también existe la Reunión de planeación del *Sprint*, la cual marca el inicio de una iteración en donde el equipo realiza compromisos de metas y actividades específicas. Otra reunión es la Revisión del *Sprint* en donde se muestra la funcionalidad completada y la que faltó por terminar; aquí se evalúa si se deben hacer cambios a los planes originales. La reunión Retrospectiva se realiza al final de cada iteración y en ella se comparten los aprendizajes obtenidos para hacer mejoras futuras.

3.1.3. Descripción del caso

Se convocó a quince estudiantes programadores universitarios para participar en un desarrollo experimental; deberían conocer por lo menos un lenguaje de programación y tener habilidades lógicas para la resolución de problemas. No se ofreció remuneración económica. Los interesados fueron estudiantes próximos a egresar de las carreras de licenciatura en informática y de ingeniería en sistemas computacionales. Se les aplicó un examen de selección para asegurar su nivel de conocimientos y habilidades. Al iniciar el trabajo se les capacitó a todos sobre el lenguaje *Java* para *Android* con el entorno *Eclipse*. Posteriormente, se dividió al grupo en dos equipos; uno recibió capacitación sobre la metodología XP y otro sobre

Scrum. Después, inició otra etapa de trabajo durante la cual se desarrolló el *software* solicitado. Cada equipo siguió la metodología particular que se le asignó y trabajó durante dos iteraciones para la entrega final. El equipo de XP estaba conformado por siete personas y el de *Scrum* por ocho.

El desarrollo consistió en una aplicación móvil en *Java* para *Android* que ayudara a ejercitar el razonamiento matemático en las personas; debería tener tres niveles de dificultad. El usuario podría seleccionar entre realizar sumas, restas o multiplicaciones. Cada vez que el usuario diera un resultado correcto al problema planteado por el *software*, éste debería informarle que contestó correctamente mostrando una imagen y reproduciendo un sonido característico; de la misma manera debería suceder si el usuario cometía un error. El programa debería validar las entradas vacías y debería tener un contador para el número de intentos de solución a cada problema. El usuario podría rendirse en cualquier momento y acto seguido, se le mostraría el resultado correcto; también se podría comenzar un nuevo juego en cualquier momento. Por último la aplicación debería proporcionar al usuario un mecanismo de ayuda que le orientara a utilizar el *software*, y debería mostrar los nombres de los desarrolladores y un medio de contacto.

3.1.4. Experiencias

Para recabar las experiencias de los desarrolladores y del equipo de investigación, se aplicaron cuestionarios escritos y se condujeron entrevistas no estructuradas. En los cuestionarios se solicitó mencionar las principales ventajas y problemáticas de la forma de trabajar que aportaba la metodología que se utilizó. En las entrevistas se interactuó personalmente con cada participante para conocer cómo fue su experiencia en el proyecto.

En los siguientes subapartados se presentan estos resultados desde las perspectivas de desarrolladores, investigadores y de un usuario externo que evaluó las aplicaciones finales entregadas por cada equipo.

3.1.4.1. Perspectiva de los desarrolladores participantes

Los participantes del equipo de XP percibieron las siguientes ventajas de la manera en la que trabajaron: la metodología permite que las diferentes visiones de cada persona se conjunten para producir un mejor producto final. La dinámica que se genera en el equipo favorece el aporte de ideas de todos los integrantes, el trabajo colaborativo bien distribuido y la reducción del tiempo de entrega. Los participantes expresaron que el énfasis en la comunicación en el equipo y la retroalimentación continua del cliente son elementos que ellos consideraron importantes para lograr la rapidez y la buena calidad con la que se desarrolló el *software*.

Los principales problemas o desventajas que los participantes del equipo de XP encontraron en este desarrollo de *software* en particular fueron: la falta de organización y coordinación del equipo, en especial durante la primera iteración, las diferentes formas de pensar, analizar, diseñar y programar de cada participante, así como la integración del código, que tomó más tiempo del previsto y, aunque no provocó un retraso en la entrega final, estuvo a punto de hacerlo.

También se realizaron entrevistas no estructuradas a los participantes que trabajaron con XP para conocer su experiencia general en este desarrollo experimental. Los desarrolladores reportaron haber trabajado de forma amena y haber estado muy motivados a seguir aprendiendo sobre la Programación Extrema. Las celebraciones simbólicas con galletas y refresco realizadas por el equipo al final de cada iteración les parecieron muy importantes para relajarse y sentir que su trabajo era reconocido. La práctica de no llevarse trabajo a la casa les pareció muy atractiva, pues contradecía la realidad laboral de muchas empresas de desarrollo de *software* que ellos conocían. Por otra parte, la programación por pares los hizo sentirse en confianza durante todo el proceso. La experiencia fue calificada como excelente por todos los entrevistados.

El equipo de *Scrum* identificó las siguientes ventajas de la metodología: se fomenta la participación conjunta de todos los integrantes para cumplir un objetivo; de esta manera, el equipo se integra con una forma ágil de trabajar en donde predomina la ayuda mutua. La participación activa y eficiente en el área de fortaleza de cada persona está siempre presente, lo cual contribuye a terminar el proyecto con mayor rapidez. Los participantes expresaron que el modelo de la división de la carga de trabajo les pareció bueno y que la relación con el resto del equipo fue cordial y permitió el desarrollo simultáneo de varias actividades en común acuerdo.

Los principales problemas o desventajas que encontraron los desarrolladores que siguieron *Scrum* en este proyecto en particular fueron: los diferentes niveles de habilidades y conocimientos de cada integrante, la poca paciencia que mostraron algunos miembros del equipo ante los problemas que surgían y la escasa manera de socializar entre el equipo. Por otra parte, el equipo tuvo dificultades en la toma de decisiones; también tuvo inconvenientes con el entorno de desarrollo y con algunos aspectos avanzados de la sintaxis del lenguaje utilizado.

3.1.4.2. Perspectiva de los investigadores

El equipo de investigación expresó que la aplicación de XP en el proyecto le pareció positiva en todos los aspectos ya que se logró producir a tiempo un software que cumplió con los requerimientos especificados y prevaleció un ambiente de trabajo agradable y de respeto hacia las reglas y los plazos de tiempo establecidos. Se

observó que cada miembro del equipo se retroalimentó de sus compañeros, sobre todo en la programación por pares, en donde dos integrantes utilizaron una sola computadora. En todo momento se percibió un ambiente de compañerismo, cooperación y buena convivencia que se vio fortalecido con las celebraciones simbólicas que se hicieron al final de cada iteración.

Los investigadores al observar al equipo que siguió la metodología de XP destacaron los siguientes aspectos positivos: el buen ambiente de trabajo que se percibió, la agilidad con la que se trabajó y el énfasis en la obtención de un producto funcional. Las principales áreas de oportunidad que encontraron se refieren a la falta de coordinación del equipo al principio del trabajo y a la lentitud con la que se integró el código.

Con *Scrum* también se logró producir un *software* que cumplió con los requerimientos establecidos. La aplicación de esta metodología también fue muy positiva según la percepción de los investigadores. Sin embargo, aunque el equipo cumplió con el límite de tiempo establecido para la entrega final, demoró un poco más en terminar que el equipo de XP; el ambiente de trabajo fue bueno pero no se observó una interacción tan fuerte entre los integrantes como en el equipo de XP; algunos, incluso se mostraron pasivos en algunos momentos. También la relación personal de los participantes se percibió un poco más neutral que en el otro equipo. La organización que adoptó el equipo de *Scrum* centralizó en unas cuantas personas las tareas medulares de programación. Aunque todo el equipo participó activamente en las etapas de análisis, diseño y pruebas, y todos los integrantes presenciaron con atención las labores de programación, no todos interactuaron igual con el código, como ocurrió en el equipo de XP.

Los investigadores destacaron los siguientes aspectos positivos en el equipo *Scrum*: la integración del equipo en las diversas tareas que había que realizar, la colaboración grupal en los momentos difíciles y la sinergia de habilidades diversas que logró el equipo con la estrategia que siguió. Por otra parte, las principales desventajas que encontraron se refieren a la poca coordinación inicial que tuvo el equipo de desarrollo, al variado nivel de conocimientos y habilidades de los participantes y a su mejorable nivel de iniciativa y convivencia.

3.1.4.3 Perspectiva de un Usuario final externo

La persona externa que condujo una prueba de usabilidad al *software* final entregado por el equipo de XP reportó no haber tenido problemas en el manejo de la aplicación desarrollada y constató además su completa funcionalidad; refirió que el *software* le parecía intuitivo y muy sencillo de utilizar.

Al probar la aplicación que desarrolló el equipo de *Scrum*, este usuario tampoco tuvo problemas en su manejo; sin embargo, indicó que le pareció que la interfaz gráfica no era tan sencilla de utilizar y que hubiera preferido que fuera más intuitiva.

Cuando se le planteó el siguiente escenario: si tuviera que elegir una de ambas aplicaciones para instalarla y usarla en su teléfono celular, ¿con cuál se quedaría?, sin dudarlo eligió el *software* del equipo de XP, pues dijo que le parecía más completo y sencillo.

3.1.5. Conclusiones y reflexiones

En este estudio se presentó una serie de experiencias derivadas de un caso de estudio en el que cada uno de los dos equipos participantes realizó una implementación distinta en *Java* para *Android* para un mismo conjunto de requerimientos. Uno de los equipos siguió la metodología XP y el otro utilizó *Scrum*. Quedó en evidencia que ambas metodologías son capaces de producir en corto tiempo aplicaciones móviles de tamaño pequeño, funcionales y concordantes con un conjunto de requerimientos cambiantes y con programadores cuya experiencia sea principiante a intermedia.

Aunque ambos equipos reportaron que su experiencia de desarrollo en este proyecto particular fue buena, el equipo de XP tuvo una mayor interacción entre sus integrantes y la convivencia y relación personal se percibió mejor que el equipo de *Scrum*. Se observó que las prácticas de XP fomentaron la retroalimentación continua y el aprendizaje grupal.

Ambos equipos tuvieron problemas iniciales para organizarse, seguramente debido a que sus integrantes no tenían una referencia previa de las habilidades y capacidad de trabajo de sus compañeros. También ambos equipos se encontraron con algunas dificultades técnicas relacionadas con el lenguaje y el entorno de desarrollo. Ante estas situaciones, el equipo de *Scrum* decidió centralizar algunas de las tareas de programación más críticas en unas cuantas personas, quienes contaban con la retroalimentación inmediata de sus compañeros. Esto pudo contribuir a que algunos integrantes adoptaran una actitud pasiva ante el trabajo por realizar. Por el contrario, el equipo de XP trabajó siempre en pequeños equipos que interactuaron casi de igual manera con el código del programa que se estaba desarrollando.

Las dificultades técnicas que aparecieron son comprensibles, ya que el tiempo de capacitación fue corto y los desarrolladores no tenían experiencia previa en el manejo de las herramientas utilizadas.

Las experiencias obtenidas en este caso de estudio exponen al menos dos dimensiones de complejidad a las que se debería prestar especial atención en el desarrollo de aplicaciones móviles con metodologías ágiles: una, técnica y la otra, humana.

Resolver problemas lógicos, diseñar aplicaciones, conocer el lenguaje de programación, tener nociones de arquitectura del *software* y manejar con destreza las herramientas necesarias son algunos aspectos técnicos relevantes para todo desarrollo de *software*. La buena actitud para trabajar en equipo, la disposición de colaborar activamente, la sencillez para aprender cosas nuevas, la paciencia para aceptar las situaciones cambiantes son aspectos humanos relevantes para que el desarrollo de *software* en cuestión tenga un buen desenlace.

Al ser una actividad técnica que involucra interacción humana, el desarrollo de aplicaciones móviles con metodologías ágiles resulta complejo por las relaciones interpersonales y la dinámica de los equipos que intervienen. Por otra parte, aunque cada metodología ágil define un marco de actividades estandarizado, la organización propia que adoptan los equipos resulta crucial para el desempeño de sus integrantes, así como para la obtención de un *software* final de calidad.

3.2 Segundo caso de estudio: programación individual y programación por pares en cursos universitarios

3.2.1 Introducción

La universidad no es un sitio inerte en donde se reúnen alumnos y maestros. Es un lugar en donde también se puede realizar investigación; es un semillero en donde germinan ideas, prototipos y proyectos utilizables y funcionales en el mundo real y es aquí en donde se prepara hoy a los profesionales del mañana. En este apartado se presenta un caso de estudio en el que se hizo un experimento con el objetivo de comparar la programación por pares y la programación individual en los cursos universitarios de desarrollo de *software.* El escenario fue una sesión regular de clase, en donde se pidió a los alumnos que desarrollaran un mismo programa bajo una de dos modalidades: individual o en parejas. Los resultados se presentan desde la perspectiva de percepción de los estudiantes. Los detalles completos de esta investigación se encuentran en el trabajo de Roque, Herrera, López y Bravo (2017).

3.2.2 Antecedentes: Caracterización breve de la Programación por pares

La programación por pares es una práctica ágil que surgió con la Programación Extrema. La programación por pares consiste en que dos personas desarrollen y prueben programas de computadora utilizando un solo equipo de cómputo. Se prefiere que una persona tenga más experiencia que la otra y que ambos estén de acuerdo en trabajar juntos. No hay un jefe y un subordinado. Quien entienda mejor cada proceso tomará la iniciativa de realizar el trabajo en el equipo de cómputo. La teoría dice

que cuando dos personas programan en parejas, la calidad del programa realizado es mejor, la forma de pensar puede ser más rápida y clara, se ahorra tiempo, se estimula la creatividad y el proceso en general resulta más divertido (Kendall & Kendall, 2013).

Si bien es cierto que la programación por pares es una forma de trabajar conocida y abordada en la literatura, la ingeniería de *software* reconoce que no todos los hechos que se consideran verdaderos están sólidamente fundamentados en evidencias empíricas (Juristo & M. Moreno, 2001). De esta manera, una misma técnica o práctica podría ser muy adecuada para un escenario particular y podría no ser pertinente en otras circunstancias. La aplicación de la programación por pares en escenarios concretos puede respaldar los supuestos sobre ella.

3.2.3 Descripción del caso

Se realizó un experimento con 94 alumnos de la carrera de licenciado en informática de una universidad mexicana. De manera aleatoria, 54 fueron asignados para programar por pares y 40 individualmente con el objetivo de resolver un mismo problema de programación en un laboratorio de la universidad. Los lenguajes que utilizaron fueron *Visual Basic .Net* y C# a través del entorno de desarrollo *Visual Studio.*

Durante el experimento se cuidó que todas las condiciones permanecieran constantes para todos los participantes, los cuales no supieron con anticipación los detalles de este estudio. Quienes trabajaron solos tuvieron una computadora para trabajar. Quienes trabajaron por pares tuvieron una sola computadora para ambos alumnos y se intercambiaban el teclado cada cinco minutos. Todos dispusieron de 70 minutos de una sesión regular de clase para resolver el problema presentado y 10 minutos adicionales para que cada participante contestara un cuestionario que se les proporcionó.

En el laboratorio de cómputo en donde se realizó el experimento había alumnos programando en pares y en solitario sentados alternadamente. A ninguno se le prohibió el acceso a sus notas, a búsquedas en internet o a trabajos realizados con antelación en clase. Sin embargo, no estaba permitido interactuar con integrantes de otros equipos.

3.2.4 Experiencias

De acuerdo con análisis estadístico que se realizó sobre las respuestas de los cuestionarios, los alumnos que programaron por pares reportaron puntuaciones más altas que los alumnos que programaron solos en características como la facilidad de trabajar, la satisfacción con el programa desarrollado y la detección rápida de errores. Un análisis centrado solo en las respuestas de quienes programaron en parejas dejó al descubierto que los alumnos de los últimos semestres detectaron errores más

rápido que los alumnos de los primeros semestres de la carrera y que obtuvieron mayores niveles de satisfacción en la experiencia de trabajo en pareja así como en el programa desarrollado. Los alumnos de los últimos semestres también reportaron haber tenido menores niveles de estrés durante su trabajo. También se observó que los alumnos que habían cursado materias de programación antes de ingresar a la universidad expresaron que se sentían más cómodos trabajando individualmente.

3.2.5 Conclusiones y reflexiones

Al comparar la percepción de los alumnos que trabajaron por pares y los que trabajaron solos se encontró que la programación por pares podría ser una buena alternativa para que los alumnos de cursos universitarios realizaran sus prácticas durante las sesiones de clase. Las experiencias recabadas hacen pensar que la programación por pares podría ser más apropiada para los alumnos de los últimos semestres y menos indicada para los primeros semestres de la carrera profesional. Se detectó que un posible inconveniente para la implementación de una estrategia de trabajo en parejas podría ser el antecedente de haber cursado materias de programación antes de ingresar a la universidad. Al contar con conocimientos previos de programación, un alumno podría sentirse más capacitado para trabajar solo y rehusarse a interactuar con una pareja durante las prácticas realizadas. De esta manera, los alumnos con estos antecedentes académicos deberían ser motivados y monitoreados para que se integraran sin problemas a la dinámica de la programación por pares.

IV.

PARADIGMAS Y LENGUAJES DE PROGRAMACIÓN

4.1 Introducción

Thomas Kuhn define un paradigma como un logro científico reconocido y aceptado por todos, que proporciona problemas y soluciones para una comunidad durante algún tiempo. Peter Wegner agrega que los paradigmas de programación se pueden definir comprensiblemente por sus propiedades o extensionalmente a través de ejemplos.

En la definición de Wegner se identifican los conceptos de comprensión y extensión; ambos son propios de la teoría de conjuntos. "Comprensiblemente" se refiere a proporcionar una descripción de los miembros del conjunto, mientras que "extensionalmente" hace referencia a la enumeración de lenguajes específicos que soportan ese paradigma. De esta manera es posible estudiar un paradigma en particular al explorar las características de uno o más lenguajes representativos (Appleby & Vandekopple, 1998).

Un paradigma de programación representa una forma de pensamiento para la solución de problemas y define la manera en que los algoritmos son implementados en un lenguaje de programación. Se dice que un lenguaje de programación soporta un paradigma, o incluso varios, si permite desarrollar programas apegados a él, o a ellos. Es bien sabido que un programador que conoce distintos paradigmas puede razonar sobre un mismo problema desde distintos enfoques; sin embargo, es común que los programadores estén familiarizados solo con un paradigma y con uno o varios lenguajes que lo soportan.

No hay un paradigma de programación que sea considerado como el mejor para todas las aplicaciones; aunque el orientado a objetos es adecuado en la mayoría de los proyectos de negocios actuales por su versatilidad y apego al mundo real.

4.2 Paradigmas de programación

Es importante subrayar que la clasificación de los paradigmas de programación no contiene categorías fijas o cerradas. Por ejemplo, algunas personas consideran confusa la separación entre el paradigma imperativo y el orientado a objetos pues consideran que la orientación a objetos debería ubicarse dentro del paradigma imperativo. La mayoría de los lenguajes funcionales y lógicos incluyen también algunas características imperativas.

4.2.1 Paradigma imperativo

El paradigma imperativo se basa en el modelo computacional *von Newman*, en el cual las operaciones son secuenciales y se utilizan variables en localidades de memoria para almacenar los datos a través de asignaciones. A los lenguajes basados en este modelo también se les denomina de "Cómputo por efectos laterales" ya que las regiones de memoria que contienen los valores de las variables se modifican de

acuerdo con el orden de ejecución de las instrucciones. De esta manera, el estado del sistema continuamente cambia hasta llegar a la solución del problema. En un programa realizado con un lenguaje imperativo, el nivel de detalle de especificación de los algoritmos es muy alto y además se requiere indicar el orden de ejecución de las instrucciones del programa. Ejemplos de lenguajes basados en este paradigma son: C, PASCAL, FORTRAN.

4.2.1.1 Paradigma estructurado

El paradigma estructurado en bloques permite dividir una solución completa en bloques con subrutinas y datos comunes. Los bloques tienen un identificador, contienen sus propias variables locales y pueden estar anidados. Un programa escrito en un lenguaje estructurado contiene procedimientos que se ejecutan mediante llamadas explícitas. Las llamadas deben coincidir con la declaración de los nombres de los procedimientos, y deben suministrar los valores para los parámetros requeridos, si es el caso.

Los procedimientos agrupan instrucciones que responden a una necesidad lógica particular. Las instrucciones pueden incluir asignaciones, bifurcaciones, iteraciones, y llamadas a otros procedimientos. En los lenguajes estructurados en bloques, el procedimiento es el principal bloque de construcción para los programas. *Ada, Pascal* y *C* pertenecen a esta categoría.

4.2.2 Paradigma declarativo

El paradigma declarativo hace énfasis en lo que la computadora debe realizar mediante la ejecución del programa. Los lenguajes declarativos pueden percibirse como de más alto nivel que los imperativos, pues el programador trabaja alejado de las operaciones del CPU sin tener que hacer asignaciones directas en las variables del programa. Dentro del paradigma declarativo se distinguen tres estilos de programación provenientes de dominios matemáticos: el funcional (teoría de funciones), el lógico (lógica matemática) y el de lenguaje de base de datos (cálculo relacional).

4.2.2.1 Paradigma funcional

El paradigma funcional está inspirado en el "Cálculo Lambda", un modelo computacional formal desarrollado por Alonzo Church en la década de los treinta. Se basa en la representación de funciones mediante expresiones con el objetivo de obtener resultados o salidas a partir de argumentos o entradas. De esta manera, se concibe una función como una regla de asociación entre los elementos de un conjunto origen y de un destino que regresa un valor unitario.

Las expresiones pueden contener términos independientes, o bien, otras expresiones. Al concluir la evaluación sencilla o recursiva de las expresiones se alcanza la solución del problema. El proceso de evaluación se entiende como el resultado de reescribir unas expresiones en otras, apegándose siempre a las definiciones de función

Existen lenguajes funcionales *puros* e *híbridos*. Un lenguaje funcional *puro* acepta solo funciones. Son pocos los lenguajes apegados totalmente a este paradigma, pues en los programas es deseable incluir también operaciones básicas de entrada y salida alejadas del paradigma original; los lenguajes que integran lo mejor de ambos modelos, se les conoce como *híbridos*. Algunos ejemplos de lenguajes de programación que implementan el paradigma funcional son LISP, ML y HASKELL.

4.2.2.2 Paradigma lógico

Los lenguajes de programación que soportan el paradigma lógico están basados en reglas, cálculo de predicados y cláusulas de *Horn*. El cálculo de predicados permite que la computadora considere un conjunto de hechos lógicos o reglas y sea capaz de derivar soluciones inteligentes a partir de ellos. Las cláusulas de *Horn* hacen posible que solo un nuevo hecho sea deducido por cada instrucción simple.

En un programa lógico se especifican las reglas sin seguir un orden en particular; el sistema de implementación del lenguaje es el encargado de seleccionar una secuencia de ejecución que produzca el resultado adecuado. El cómputo en este paradigma intenta encontrar valores que satisfagan ciertas relaciones específicas usando búsquedas orientadas a los objetivos finales a través de las reglas lógicas. El lenguaje lógico más conocido es el PROLOG. Sin embargo, los aspectos programáticos incluidos en algunas hojas de cálculo como EXCEL también pueden considerarse dentro de este paradigma.

4.2.2.3 Paradigma del lenguaje de base de datos

Los lenguajes de bases de datos permiten crear y modificar la estructura de una base de datos mediante un lenguaje de definición de datos o DDL -*Data Definition Language*-. También permiten interactuar con los datos almacenados en ella mediante el lenguaje de manipulación de datos o DML -*Data Manipulation Language*-.

Los lenguajes de base de datos pueden ser soportados por otros lenguajes de programación, proporcionando así un esquema sencillo y flexible para acceso a datos desde las aplicaciones. Por ejemplo, un popular lenguaje es SQL *(Structured Query Language)*. SQL posee un conjunto de sentencias que permite al programador concentrarse en el resultado deseado de una interacción con bases de datos; la tarea de obtener y presentar los datos queda a cargo del traductor. SQL es soportado por

gran parte de los productos relacionales del mercado actual; además, el modelo de datos utilizado por muchos lenguajes de programación modernos acepta la inclusión directa de sentencias SQL dentro de los programas realizados.

4.2.3 Paradigma orientado a objetos

El paradigma orientado a objetos propone la descomposición de un programa en varios objetos que interactúan entre sí para lograr la meta global del programa. En este contexto, cualquier elemento que interviene en un problema o en su solución puede modelarse como un objeto que puede tener funciones y valores encapsulados dentro de él. Las funciones son acciones que el objeto es capaz de realizar, mientras que los valores son características que describen al objeto.

El modelo de objetos establece las directrices para la representación de los objetos; sus elementos fundamentales son: abstracción, encapsulamiento, modularidad y jerarquía. Estos elementos como el paradigma mismo se abordarán con más detalle en el siguiente capítulo.

Algunos lenguajes combinan la orientación a objetos con otros paradigmas. Sin embargo, esta combinación exhibe un fuerte comportamiento imperativo. Como ejemplo, puede mencionarse CLOS, un lenguaje funcional *(extensión de Common Lisp)* con capacidades de orientación a objetos. Las raíces de la orientación a objetos se encuentran en el lenguaje de programación Simula 67 que implementó por primera vez varios de los conceptos más importantes de este paradigma. Ejemplos de lenguajes orientados a objetos son: SMALLTALK, C++, Java y los lenguajes de la plataforma .NET tales como VB.NET y C#.

4.2.4 Paradigma orientado a aspectos

En un sistema de *software* desarrollado con el paradigma orientado a objetos, se mezclan la funcionalidad básica y la secundaria o transversal. Se entiende por funcionalidad básica, aquella que está relacionada con la lógica principal del *software* que se está desarrollando; y por funcionalidad secundaria o transversal, aquella lógica que atraviesa varias partes del sistema, como por ejemplo la seguridad y el acceso a datos. Esta mezcla de funcionalidades es inherente a la orientación a objetos y promueve la aparición de características que empobrecen la calidad del código final.

El paradigma orientado a aspectos plantea como solución a este problema la separación de la lógica transversal en una nueva estructura denominada aspecto, aislándola de la funcionalidad básica, que puede estar expresada utilizando clases o módulos. Un esquema orientado a aspectos requiere de un mecanismo que relacione ambas lógicas y especifique la forma en que los aspectos se incorporan a la funcionalidad principal. Este enfoque tiene como objetivo permitir una manera

de trabajo más limpia y eficiente, así como obtener código de mayor calidad. Ejemplos de Lenguajes orientados a aspectos son: *AspectJ*, *AspectC* y *Java Aspect Component*. Los elementos del paradigma orientado a aspectos se abordan en el capítulo 6 de este trabajo.

4.3 Generaciones de lenguajes de programación

Grady Booch (Booch, 2000) cita a Wegner, quien propone una clasificación de lenguajes de programación de alto nivel en varias generaciones. De acuerdo con esta taxonomía, los lenguajes de la primera generación fueron utilizados principalmente para aplicaciones científicas y de ingeniería; hacían énfasis en la representación de expresiones matemáticas. Estos lenguajes fueron el primer acercamiento al dominio del problema, alejándose de la máquina como tal.

Los lenguajes de la segunda generación mostraron énfasis en abstracciones algorítmicas, teniendo como entorno máquinas cada vez más potentes y económicas que hacían posible el desarrollo de aplicaciones comerciales.

Para finales de los sesenta, el costo del *hardware* se había reducido mucho, y la capacidad de procesamiento había crecido casi exponencialmente. Los problemas que podían resolverse eran más complejos y requerían más tipos de datos; es así como nacieron lenguajes como ALGOL 60 y *Pascal*, que soportaban abstracción de datos, permitiendo al programador definir sus propios tipos.

En los setenta se produjeron casi dos mil diferentes lenguajes de programación producto de la intensa investigación y producción en esta área, así como de la necesidad de realizar programas cada vez más complejos. Aunque muy pocos de estos lenguajes pasaron a la posteridad, muchos de los conceptos introducidos por ellos encontraron su madurez en versiones subsecuentes de otros lenguajes.

En los lenguajes de la primera y principios de la segunda generación el bloque básico de construcción era el subprograma. Los programas tenían una estructura plana con datos globales; y aunque contenían subprogramas, las grandes modificaciones aún eran difíciles pues introducían errores fácilmente.

A mediados de los setenta, con los lenguajes de fines de la segunda y principios de la tercera generación se reconoció a los subprogramas como un mecanismo de abstracción denominado "abstracción procedimental" y no solo como medios para ahorrar trabajo. De esta manera los nuevos lenguajes de programación comenzaron a soportar diversas formas de paso de parámetros, y se establecieron las bases de la programación estructurada. También surgieron los métodos de diseño estructurado.

En los lenguajes de finales de la tercera generación, el módulo compilado de manera separada permitió desarrollar de manera aislada diferentes partes de un mismo programa. Sin embargo, los lenguajes de programación no contaban con

suficientes reglas de consistencia semántica entre las interfaces de los módulos, lo que provocaba con frecuencia errores en tiempo de ejecución.

Durante la década de los ochenta y noventa, con el advenimiento de los sistemas operativos gráficos, los lenguajes de programación comenzaron a ofrecer herramientas que permitían al programador realizar interfaces de usuario más atractivas con menor esfuerzo, implementar accesos a datos rápidamente, automatizar tareas y utilizar asistentes durante estos procesos.

Para el inicio de los noventa, la mayoría de los lenguajes comerciales proporcionaban soporte para la creación de programas visuales basados en objetos u orientados a ellos. En los lenguajes basados en objetos y orientados a objetos, el bloque de construcción físico es el módulo, que representa una colección de clases y objetos, en lugar de sub-programas. Los bloques de construcción lógicos son clases y objetos, en lugar de algoritmos. En estos lenguajes existen pocos o ningún dato global.

Cuando la década de los noventa llegaba a su fin, la popularidad de los lenguajes orientados a aspectos comenzó a crecer. El nacimiento de *AspectJ* y su posterior aceptación impulsaron el uso de la Programación Orientada a Aspectos (POA) entre las comunidades de programadores y académicos. Surgieron lenguajes orientados a aspectos de propósito general para implementar todo tipo de intereses; y de propósito específico, con características para manejar de manera especializada solo cierto tipo de aspectos en los programas.

Los lenguajes orientados a aspectos ofrecen mecanismos para mantener la separación completa de intereses en un programa; y para este propósito, extienden a algún lenguaje existente.

Algunos lenguajes de programación ofrecen la posibilidad de utilizar sofisticados entornos para unificar y coordinar las actividades del ciclo completo de desarrollo de software. Además, incorporan características que facilitan el desarrollo de aplicaciones con tecnologías de comunicación, seguridad, persistencia e interoperabilidad.

V.

ORIENTACIÓN A OBJETOS

5.1 Introducción

La orientación a objetos resulta útil para ayudar a combatir la complejidad propia de la mayoría de los sistemas; debido a esto, no solo se ha utilizado en el área de programación sino también en diseño de sistemas, *hardware* y sistemas operativos. De esta manera la orientación a objetos se ha convertido en un enfoque transversal dentro de la informática desde principios de los setenta. La POO plantea soluciones a los problemas de una manera apegada al dominio del problema, modelando objetos mediante la abstracción de sus características y comportamientos más relevantes. De manera formal, Booch (2000) define a la programación orientada a objetos como una manera de crear programas de cómputo utilizando conjuntos de objetos que cooperan entre sí, en donde cada objeto es un ejemplar creado a partir de una clase y en donde las clases son parte relaciones jerárquicas de herencia.

5.2 Historia

Las primeras concepciones de clases y objetos en lenguajes de programación se remontan al lenguaje Simula 67, que nació en el Centro de Cómputo Noruego. Sus creadores, Kristen Nygaard y Ole-Johan Dahl tenían el objetivo de describir sistemas y realizar simulaciones. Ellos buscaban una manera de expresar procesos permanentes y activos que pudieran ser creados y destruidos cuando fuera requerido; también incluir los procesos como una extensión de algún lenguaje existente, ejecutarlos de forma concurrente y agruparlos en clases.

Los lenguajes procedimentales existentes hasta entonces, solo permitían manejar datos pasivos y procedimientos desconectados de ellos. Bajo la visión de Simula, los procesos, que tiempo después se llamaron objetos, contenían procedimientos que estaban relacionados con sus datos; además, las clases estaban definidas en jerarquías.

Los años posteriores fueron decisivos para la adopción generalizada de estos conceptos: durante la década de los setenta y ochenta, la POO alcanzó su nivel de madurez en gran parte gracias a la evolución del lenguaje *Smalltalk*; y, a partir de los noventa se extendió como el estado del arte en cuanto a la programación de aplicaciones de negocios se refiere. Como puede observarse, la POO no es el resultado de una sola idea abrupta, sino la consecuencia de la evolución natural de los paradigmas de programación.

5.3 El modelo de objetos

Los fundamentos de ingeniería en los que se basan las tecnologías orientadas a objetos se denominan "Modelo de objetos" y representan un marco conceptual que sirve de referencia para estas tecnologías. En este modelo existen cuatro elementos

fundamentales: abstracción, encapsulamiento, modularidad y jerarquía; y tres elementos secundarios: tipificación, concurrencia y persistencia (Booch, 2000). Los elementos fundamentales son imprescindibles, y determinan en conjunto si un modelo es orientado a objetos o no. Los elementos secundarios resultan útiles para el modelo; por lo que su presencia es deseable pero no obligatoria.

5.3.1 Elementos fundamentales del modelo de objetos

5.3.1.1 Abstracción

La abstracción es una descripción simplificada de un elemento de un sistema, la cual contiene solo las características relevantes suficientes para trabajar con él en un contexto determinado, desde el punto de vista del observador.

Mediante la abstracción es posible enfocarse en el estudio de las peculiaridades interesantes de los elementos a representar en un contexto, dejando a un lado aquellas que pudieran ser irrelevantes y distraer la atención del observador. Es así como dos personas distintas pueden tener abstracciones diferentes acerca de los mismos elementos, o bien, una misma persona puede concebir abstracciones distintas para un mismo objeto en dos contextos diferentes. Por ejemplo, en el desarrollo de un juego de vídeo en donde una mascota es el personaje principal, la abstracción de la mascota podría incluir acciones como caminar, correr, ladrar, comer, jugar, o perseguir un objeto, mientras que en el desarrollo de un *software* gestor de expedientes veterinarios, estas acciones estarían fuera de lugar si se hiciera la abstracción de una mascota.

5.3.1.2 Encapsulamiento

Mientras que la abstracción modela el comportamiento exhibido por un objeto, el encapsulamiento se ocupa de la implementación de características que dan lugar a ese comportamiento y que no tienen razón de ser expuestas. En la vida cotidiana, hay muchas situaciones en las que se puede identificar el encapsulamiento. Por ejemplo, un televisor tiene botones de control con los cuales es posible aumentar o disminuir el volumen, o cambiar el canal que se muestra en la pantalla. Una persona solo puede interactuar con el televisor a través de esos botones, pues son de acceso público. Los circuitos, tarjetas y cables existen y están ocultos en el interior pero no están accesibles para las personas por razones de seguridad. En un cajero automático, las personas pueden interactuar con el sistema bancario a través de los botones o pantalla táctil del cajero porque es su interfaz pública. Una persona puede conocer el saldo en su tarjeta de débito y disponer de él siempre que tenga su tarjeta y conozca su contraseña, pero no sabrá cuál es el total del dinero efectivo que se encuentra en ese momento oculto dentro del cajero. ¿La razón? Es una información

que no le incumbe. Por eso el total del dinero efectivo no estará disponible para que las personas lo conozcan, aunque sería calculable con facilidad por el sistema bancario en cualquier momento.

En la Figura 5.1 se observa una clase con métodos públicos, que son visibles desde el exterior, y métodos privados que están ocultos al exterior, pero accesibles siempre desde el interior. Nótese que dentro del ámbito de esa misma clase, los métodos públicos pueden realizar llamadas a los privados y viceversa. De esta manera pueden identificarse dos conceptos importantes: La interfaz y la implementación. La primera se refiere a la vista externa, y la segunda provee detalles y mecanismos que consiguen el comportamiento deseado. La implementación queda oculta o encapsulada y se convierte en un secreto para cualquier observador externo.

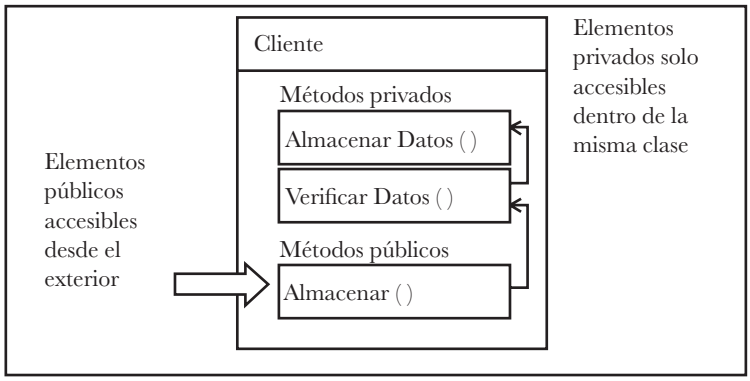

Figura 5.1 Ocultando elementos mediante el Encapsulamiento.

5.3.1.3 Modularidad

Un programa de cómputo puede ser dividido en varios componentes individuales llamados módulos. Esta división ayuda a reducir la complejidad global de las tareas y a establecer límites dentro del programa que permiten comprenderlo mejor. La capacidad para utilizar módulos puede ser provista de varias maneras por los diferentes lenguajes orientados a objetos: en forma de unidades, archivos, paquetes, etcétera, pero en todos los casos, los módulos actúan como contenedores físicos de clases y objetos.

El objetivo de la modularidad en el enfoque estructurado es agrupar subprogramas de manera significativa; por otra parte, en el paradigma orientado a objetos la meta consiste en decidir el lugar adecuado para empaquetar las clases y objetos a partir del diseño. La planeación estratégica de los módulos es la base para la reutilización de código en las aplicaciones.

5.3.1.4 Jerarquía

La jerarquía especifica una clasificación entre las abstracciones, y es un punto clave en el proceso de entendimiento de un problema. A pesar de que se sabe que una adecuada clasificación de las entidades produce una jerarquía coherente para la solución del problema, no existe un procedimiento único para lograr este objetivo en todos los casos. Cada persona organiza la información según su percepción, entendimiento, lógica, y otros factores que deba considerar en ese momento. Sin embargo, nunca debe perderse de vista que se busca conseguir una jerarquía sencilla y que facilite la reutilización.

La herencia es la jerarquía de clases más importante en un sistema orientado a objetos. Mediante este mecanismo, una clase puede ser definida en función de otras ya que una clase hija o subclase hereda de una clase padre o superclase sus características y comportamiento. Es así como una subclase puede especializar, aumentar o redefinir los elementos que ha heredado de sus ancestros. La herencia modela una relación del tipo "es un". Es decir, una subclase "es un" tipo de su superclase. De esta manera, un "Cliente con crédito" "es un" tipo especial de "Cliente" y un "Alumno de maestría" "es un" tipo particular de "Alumno".

Como puede verse en la Figura 5.2, las superclases contienen elementos comunes y las subclases elementos específicos; por esta razón, se dice que la herencia es una jerarquía de generalización-especialización.

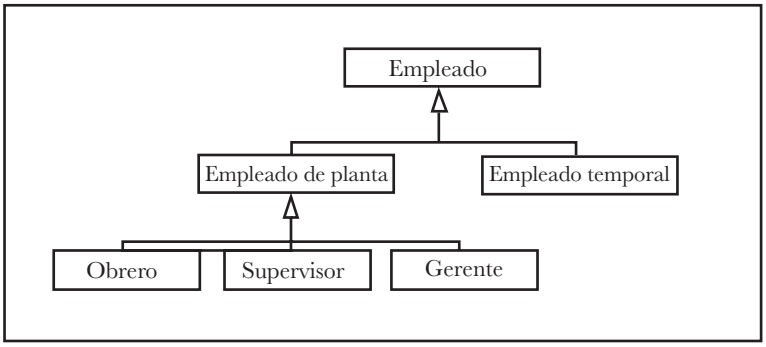

Figura 5.2 Jerarquía de clases.

5.3.2 Elementos secundarios del modelo de objetos

5.3.2.1 Tipificación

El término tipificación se refiere al concepto de *tipos de datos*. La idea central considera la congruencia entre los tipos de datos utilizados. Por ejemplo, al sumar dos números enteros se espera obtener otro número entero y no una cadena de texto.

Un lenguaje orientado a objetos puede tener comprobación estricta de tipos, débil o incluso no tener tipos. La comprobación estricta impone una consistencia

rigurosa entre los tipos; es decir, una operación de un objeto no puede ser llamada a menos que el prototipo exacto de esa operación se encuentre definido en su clase de manera propia o por herencia. De esta manera cualquier violación a la correspondencia de tipos puede ser detectada en tiempo de compilación. En lenguajes sin tipos la comprobación se realiza en tiempo de ejecución, y las violaciones a la congruencia de tipos producen errores también en tiempo de ejecución. Algunos lenguajes de programación pueden no encajar en estas categorías por su particular e híbrido manejo de tipos; en estos casos, aunque poseen mecanismos para pasar por alto las reglas sobre tipos, existe cierta tendencia a la comprobación estricta.

El tiempo de asignación o ligadura se refiere al momento en el que los nombres utilizados en un programa se ligan con sus tipos; y puede realizarse de manera estática o dinámica. La asignación temprana de tipos se llama ligadura estática y se lleva a cabo en tiempo de compilación, y la asignación tardía o ligadura dinámica se hace en tiempo de ejecución.

Cuando la herencia y el enlace dinámico interactúan, dan lugar al polimorfismo: una característica que la orientación a objetos en la que un solo nombre puede hacer referencia a objetos de muchas clases diferentes relacionadas con una superclase común. Este tema se aborda en la sección 5.5.4.

5.3.2.2 Concurrencia

La concurrencia es la característica que permite que varios objetos estén activos al mismo tiempo como consecuencia del uso de hilos de control en los programas, los cuales responden a requerimientos especiales de manejo de eventos sincronizados. Los hilos de control hacen posible la ejecución paralela de procesos; y por medio del monitoreo de la sincronización de los objetos activos se pueden mantener y acceder los objetos, sus métodos y su significado sin problemas entre los múltiples hilos.

En un programa, la implementación de concurrencia con la Programación Orientada a Objetos se puede hacer mediante el uso de clases y objetos que proveen esta funcionalidad. Por su parte, la Programación Orientada a Aspectos ha brindado en los últimos años alternativas aún más modulares para tratar la concurrencia como un interés transversal aislado. Gracias a esta relativa facilidad de programarse y al soporte nativo brindado por los sistemas operativos modernos, los sistemas concurrentes son cada vez más utilizados.

5.3.2.3 Persistencia

La persistencia es la característica que permite a un objeto continuar existiendo aún después de que el programa que lo creó ha terminado su ejecución. Se puede hablar de persistencia en el tiempo y en el espacio. La primera ocurre cuando la existencia

de un objeto trasciende la vida del programa e implica el uso de tecnologías para preservar su estado y su clase. La segunda se refiere a la representación completa del objeto en función de las localidades de memoria que ocupa. Aunque algunos lenguajes de programación ofrecen soporte nativo para la persistencia, esta característica en el modelo de objetos puede conducir a las bases de datos orientadas a objetos. Algunos productos disponibles en el mercado ofrecen una interfaz virtual orientada a objetos que oculta un modelo real de implementación relacional; esta tecnología contrastante suele ser de ayuda al desarrollar ciertas aplicaciones que operan sobre una capa de datos que ya existe.

5.4 Los objetos: del mundo real al *software*

Para la creación de un programa de cómputo orientado a objetos que modele objetos del mundo real, es necesario realizar las actividades del ciclo de vida del *software* con una visión orientada a objetos. El paradigma de objetos ha impactado todas las etapas del ciclo de vida; hoy en día existen herramientas, notaciones y metodologías estándar bastante utilizadas que consideran a los objetos como parte central de su filosofía. Como todas las demás etapas de un ciclo de vida orientado a objetos, las tres primeras (análisis, diseño, programación) adoptan como elemento central a las clases y los objetos. Al respecto, Booch (2000) explica que el análisis orientado a objetos examina los requerimientos de los usuarios tomando en consideración los términos y conceptos que se encuentran en el dominio del problema. Por otra parte, el diseño orientado a objetos necesita una notación con la cual sea posible caracterizar los modelos lógico y físico así como conceptualizar vistas estáticas y dinámicas del sistema.

La Figura 5.3 ejemplifica estas definiciones bajo el siguiente escenario: Durante el análisis se identifica un objeto libro correspondiente al dominio de la aplicación; en el diseño se especifican con detalle sus características y comportamiento dentro de una abstracción denominada “Libro”. En la programación, se implementa esa clase, y a partir de ella es posible crear varios objetos idénticos para usarlos en la aplicación.

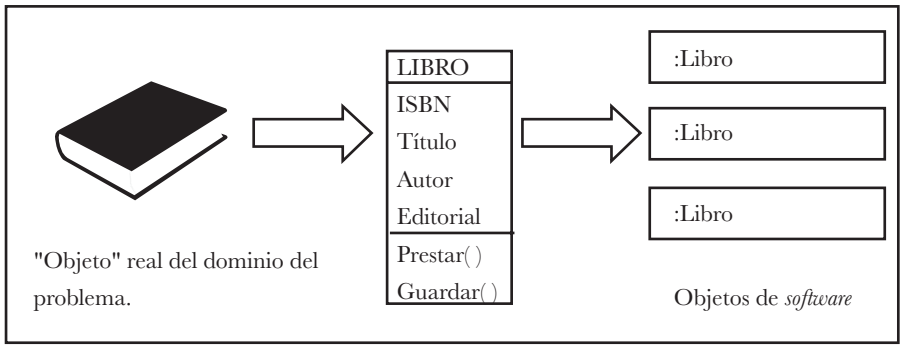

Figura 5.3 Proceso de creación de objetos de *software* Clase o plantilla para crear objetos.

5.5 Los programas en lenguajes orientados a objetos

Los conceptos clave que por su importancia deben abordarse desde la perspectiva de los programas orientados a objetos son: las clases y objetos, su forma de comunicación a través de mensajes, la herencia y el polimorfismo.

5.5.1 Clases y objetos

En la mayoría de los lenguajes comerciales orientados a objetos como C++, *Java*, C#, la clase es la base para la creación de objetos; sin embargo existen otros tantos lenguajes, en su mayoría de uso académico como *Self, Kevo* y *Omega* que permiten crear objetos basados en prototipos, de manera gráfica a partir de nada, o por herencia tomando otro objeto como base, lo que produce un código más claro (Quiroga, 2004).

Una clase es una plantilla o molde que se utiliza como contenedor de todas las características similares de un tipo de objeto. Como un objeto es creado a partir de una clase, también contiene todas las características definidas en ella. Un objeto posee atributos y acciones dentro de él. Los atributos determinan las características que lo describen, por ejemplo, identificadores e indicadores de estado; las acciones denotan el comportamiento que es capaz de realizar, por ejemplo: manipular datos, realizar cálculos, procesos o comprobaciones.

Dentro de un programa, un objeto puede representar diferentes conceptos como: cosas tangibles, roles o papeles, organizaciones, incidentes, interacciones, especificaciones o lugares (Joyanes Aguilar, 2012). Al proceso de creación de un objeto se le denomina instanciación y con él comienza su tiempo de vida. Se dice que un objeto es una instancia de una clase, lo que significa que un objeto es un ejemplar creado a partir de una clase. Pueden derivarse muchos objetos de la misma clase, pero cada uno es independiente del resto pues tiene su propio estado e identidad. El estado de un objeto es el valor de sus atributos en un momento determinado y puede cambiar a lo

largo del tiempo, dependiendo de las operaciones realizadas y de las interacciones con otros objetos. La identidad de un objeto está determinada por un identificador único que lo distingue del resto de los objetos aun cuando todos pertenezcan a la misma clase.

Por ejemplo, un plano con dibujos y especificaciones físicas de una casa puede considerarse como una clase. Tomando el plano como referencia se pueden construir muchas casas iguales que ocupen un lugar en el terreno y tengan una ubicación definida en una ciudad. Cada casa sería considerada como un objeto diferente creado a partir de la definición de una clase. Aunque todas las casas sean idénticas, cada una conservaría su propio estado. Por ejemplo, las coordenadas de ubicación de cada una serían distintas, mientras que la fachada de una casa podría pintarse de color azul y el resto de las fachadas podría ser de color verde. Cada casa tendría una dirección distinta en la ciudad aun si fueran adyacentes en la misma calle, por lo que se dice que cada una tendría identidad sería única.

Generalmente, una clase se crea con el objetivo de crear instancias a partir de ella. Sin embargo, existen clases especiales que contienen un conjunto de elementos que solo son útiles como base para otras clases y no tendrían sentido propio si fueran instanciadas; estas clases reciben el adjetivo de "abstractas" pues corresponden a conceptos comunes que requieren una especialización obligatoria para ser instanciadas. Es decir, una clase abstracta solo puede servir como súper-clase en una jerarquía, y no es posible utilizarla para crear objetos directamente. Por ejemplo, "Comida" podría ser una clase abstracta porque el término no hace referencia a un platillo en particular como "Pollo en mole" o "Arroz blanco", las cuales sí serían clases específicas derivadas de "Comida".

Un objeto puede contener otros objetos, y estos a su vez a otros más. Cuando un objeto está contenido dentro de otro puede formar parte de una relación de agregación o de composición. La agregación modela una relación del tipo "parte de" que sería inadecuado representar con herencia. De esta manera un empleado "es parte de" una compañía, y un alumno "es parte de" una escuela. Nótese que un empleado no "es una" compañía y un alumno no "es una" escuela. En la agregación, los elementos tienen tiempos de vida independientes, por lo que sus existencias también son independientes. Si un empleado deja una compañía, la compañía sigue trabajando con el resto de los empleados y el empleado también sigue sus labores en otra parte. Incluso, podría ser que un empleado trabajara en dos o más compañías distintas al mismo tiempo. Con la agregación se denota que un objeto contiene referencias a otros objetos. Por el contrario, una relación de composición existe si hay una relación de pertenencia fuerte, es decir, cuando un objeto "tiene" otro objeto. La composición implica que el tiempo de vida de los objetos está ligado, pues todos ellos se crean y se destruyen con el objeto contenedor. Además, un objeto no puede ser al

mismo tiempo parte de dos objetos como sucede en la agregación. Por ejemplo, un libro "tiene" muchas páginas. Si se destruye el libro se destruyen las páginas también y una página no puede ser parte de otro libro simultáneamente.

Sin importar si se trata de otros objetos, métodos o valores, dentro de una clase el programador selecciona la visibilidad de estos elementos: puede ocultarlos, hacerlos públicos o restringirlos mediante modificadores de acceso. Dependiendo del lenguaje de programación que se esté utilizando, se dispone de elementos privados, que son accesibles solo dentro de la clase a la que pertenecen, públicos, que son accesibles desde el exterior de la clase, o protegidos, que son accesibles desde las clases derivadas. Estos modificadores representan mecanismos necesarios para la implementación del encapsulamiento.

5.5.2 Comunicación entre objetos

Los objetos se comunican en un programa a través de mensajes (ver Figura 5.4), los cuales son órdenes que se envían a un objeto para que éste realice acciones. Cuando un objeto se encuentra inactivo y recibe un mensaje, su estado cambia a activo.

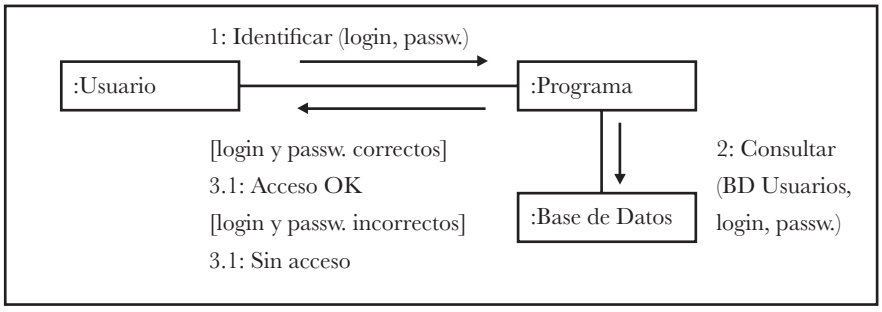

Figura 5.4 Representación de la comunicación de los objetos en un programa.

En el nivel de implementación, un mensaje equivale a la llamada a un método definido dentro del objeto y debe incluir: el nombre del objeto, la acción que se desea ejecutar y los parámetros correctos indicados por la firma del método. La firma del método se refiere al conjunto conformado por el nombre del método, el número y tipo de datos de sus parámetros.

Cuando se crean métodos con el mismo nombre que difieren en su lista de parámetros, ocurre la sobrecarga, con la cual es posible implementar varias versiones de la funcionalidad proporcionada por un método. La versión correcta del método se ejecutará dependiendo de la firma con la que sea llamado.

Un tipo especial de método es el constructor, que se ejecuta de manera automática cuando se crea un objeto. Un constructor es el encargado de inicializar

el estado del objeto y de ejecutar las operaciones requeridas para su funcionamiento. Existe otro método denominado destructor, que se ejecuta antes de que el objeto sea destruido. El destructor es el encargado de liberar el estado y los recursos de un objeto que fue creado, pero que ya no se utilizará.

5.5.3 Herencia

La herencia permite escribir clases nuevas partiendo de otras existentes. Para implementar este mecanismo se requiere una clase base que herede sus características a otras subclases, las cuales brindarán una funcionalidad más refinada. Los tipos de herencia pueden clasificarse de varias maneras. Si se toma en cuenta el número de clases bases que tiene una subclase, se pueden establecer dos posibilidades: herencia simple o múltiple. Una subclase que hereda de una sola clase base forma parte de un mecanismo de herencia simple, mientras que una subclase que hereda de más de una clase base al mismo tiempo es parte de un esquema de herencia múltiple. Según sea el lenguaje de programación en el que se trabaje, se dispone de herencia simple, múltiple, o ambas. Por ejemplo *Java*, C# y *Smalltalk* admiten solo herencia simple mientras que C++ soporta ambas.

5.5.4 Polimorfismo

El polimorfismo, cuyo nombre refiere a una multiplicidad de formas, hace posible que diferentes objetos con una clase base común respondan de modo diferente al mismo mensaje; de acuerdo con el momento en el que se determina el tipo de los objetos involucrados, puede clasificarse como estático o dinámico. El estático es aquel que se implementa en tiempo de compilación. Por ejemplo, la sobrecarga de métodos en algunos lenguajes de programación es polimorfismo estático. Por el contrario, en el dinámico, el tipo de los objetos se determina en tiempo de ejecución. Algunos autores reconocen solo la existencia del "polimorfismo dinámico" y se refieren a él como "polimorfismo".

La manera como algunos lenguajes de programación como C# y C++ soportan el polimorfismo dinámico es mediante elementos virtuales. Al definir un elemento virtual en una clase base, se especifica que una clase derivada será capaz de proporcionar una implementación diferente para ese elemento, aunque no es obligatorio que lo haga. Si no lo hace, la clase derivada utilizará intacto el elemento virtual heredado de la clase base. Cuando la clase derivada proporciona una nueva implementación, se dice que ha "sobre-escrito" o "anulado" al correspondiente elemento de la clase base. En este contexto, se define "sobre-escritura" o "anulación" como el reemplazo, que hace una clase derivada, de un elemento definido en una clase base.

5.6 Reutilización orientada a objetos

La reutilización es una característica que permite al *software* ser incorporado a distintos programas, por diferentes programadores en múltiples ocasiones, lo cual hace posible el ahorro de tiempo y esfuerzo de desarrollo. Las clases realizadas con el paradigma orientado a objetos pueden ser reutilizadas con facilidad mediante los mecanismos de encapsulamiento y herencia. Por ejemplo, una clase puede diseñarse con estrategia, colocando los modificadores de acceso apropiados a los elementos para que puedan ser privados, públicos o heredables pensando en que otro programador tenga la posibilidad de tomarla como clase base y personalizarla con elementos propios sin mayor problema.

Adicionalmente, la mayoría de los lenguajes modernos orientados a objetos incluyen una biblioteca de clases listas para utilizarse, ya sea como clases base o de forma directa en los programas. Esto proporciona también un eficaz método para reutilizar el *software*.

5.7 Aplicación del paradigma orientado a objetos

Aun cuando el paradigma orientado a objetos por sí mismo posee grandes virtudes que facilitan el proceso de producción de *software*, los desarrolladores todavía tienen a su cargo la tarea de aplicar bien la filosofía de objetos. Un buen entendimiento del paradigma es el primer paso para conseguir un diseño adecuado que conlleve a una solución eficiente.

La formación profesional del equipo de desarrollo juega un papel fundamental en la creación de un sistema orientado a objetos y debe respaldar siempre el trabajo que se lleva a cabo. Por ejemplo, aún hoy en día, existen muchos desarrolladores cuya formación profesional fue con la programación estructurada, por lo que el paradigma de objetos les es ajeno en diferentes niveles. Algunos lo desconocen completamente, otros conocen los conceptos del modelo pero no los han llevado a la práctica; y otros más han evolucionado, adoptando en sus desarrollos la orientación a objetos por completo. En cualquiera de los casos anteriores, el proceso de evolución del paradigma estructurado al orientado a objetos es una labor de aprendizaje que no debe tomarse a la ligera.

Aunque las herramientas, metodologías y lenguajes son solo auxiliares durante el proceso, si son seleccionados con cautela y utilizados con inteligencia, suelen ser poderosos aliados en las tareas del equipo de desarrollo. Además, en todo momento es muy importante que la abstracción se mantenga siempre en un nivel adecuado y homogéneo; y que en la fase de programación los objetos sean administrados con mesura y conciencia para asegurar buenos resultados.

VI.

ORIENTACIÓN A ASPECTOS

6.1 El problema de la separación de intereses en el *software*

Para comprender la naturaleza de la orientación a aspectos, primero, es necesario entender qué es la separación de intereses y por qué representa un reto para el desarrollo de *software*.

Un interés, también llamado preocupación, o *concern*, en idioma inglés, es una característica o conjunto de características que un sistema debe tener. Un sistema de *software* implementa intereses de dos tipos: básicos y secundarios. Los intereses básicos son aquellos directamente relacionados con la funcionalidad principal del programa. Los intereses secundarios son aquellos que deben implementarse, pero no forman parte de la lógica principal; están presentes en varias partes del mismo sistema, por eso se dice que son intereses "transversales" o *Crosscutting concerns*. Por ejemplo, en un sistema empresarial, la generación de nóminas, facturación e inventario son intereses básicos, mientras que el manejo de errores, los mecanismos de acceso al sistema y la persistencia de datos son intereses secundarios. La Figura 6.1 muestra una representación gráfica de ambos tipos de intereses.

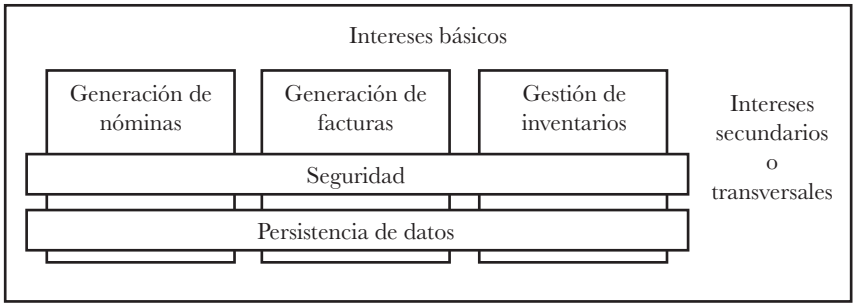

Figura 6.1 Representación de intereses básicos y transversales

La funcionalidad de los intereses básicos puede implementarse con facilidad con la ayuda de la Programación Orientada a Objetos o de la programación estructurada. Sin embargo, con estos enfoques, los intereses secundarios se extienden por varios módulos y no pueden ser encapsulados en un solo lugar. Algunos ejemplos típicos de estos intereses son: seguridad, manejo de errores, sincronización, persistencia de datos, manejo de memoria, bitácoras, restricciones de tiempo, uso transparente de transacciones, conexiones a bases de datos y administración de comunicaciones.

La descomposición de intereses de un programa puede pensarse como un espacio n-dimensional, en donde cada tipo de interés en el sistema ocupa su propia dimensión. La POO es un paradigma que si bien es práctico y eficiente, no puede aislar bien cada dimensión pues trabaja con un único espacio dimensional. Esto es una limitante del paradigma orientado a objetos que origina una mezcla indeseable

de lógica principal con intereses transversales. De esta manera, en la POO no se logra una separación de intereses "limpia" por completo.

Un sistema con varias dimensiones de intereses que es implementado en un espacio unidimensional, tiene código disperso y código enredado causado por los intereses transversales. El código disperso o *scattered code* se presenta cuando el código para un interés transversal se encuentra esparcido por diferentes partes del sistema. El código enredado o *tangled code* ocurre cuando una clase o módulo contiene funcionalidad principal y además incorpora funcionalidades secundarias. La presencia de estos dos tipos de código en los programas origina problemáticas tales como la dificultad de reutilización, mantenimiento y evolución, bajos niveles de calidad y la disminución de la productividad.

La separación de intereses en el *software* ha sido abordada desde dos visiones: una multidimensional y otra bidimensional. Ambas pretenden superar los problemas de la orientación a objetos a través de una separación más limpia de intereses. Sin embargo, la bidimensional ha ganado mucha popularidad quizás gracias a que es la manera de trabajar del popular lenguaje *AspectJ*. Este modelo realiza la separación entre intereses básicos y transversales. Los transversales se convierten en "aspectos" y deben relacionarse con los básicos. Por otra parte, cuando se adopta un modelo multidimensional, todos los intereses tienen la misma importancia y no existe la clasificación entre intereses básicos y secundarios. Esta visión aporta mucha flexibilidad en el proceso de composición, sin embargo, puede llegar a ser compleja y ocasionar problemas en las interacciones de los intereses. En este texto se adoptará la perspectiva bidimensional para describir a la Programación Orientada a Aspectos (POA), o AOP, por sus siglas en idioma inglés correspondientes a *Aspect Oriented Programming.*

La POA es un paradigma de programación que propone una solución al problema de la separación completa de intereses que existe en otros paradigmas como el orientado a objetos. La POA permite encapsular los intereses transversales en entidades bien delimitadas que se denominan aspectos. De esta manera, los aspectos diseminados por el sistema se separan de la funcionalidad principal y cada aspecto se separa de otros. Entonces, cada aspecto es una entidad aislada con la cual se trabaja con eficiencia y se logra más organización y limpieza en el *software*.

Un aspecto se define como un interés con presencia transversal en un sistema; además, no puede encapsularse bien y es una unidad conceptual que se puede identificar y aislar (Algorry, 2005). Kiczales, uno de los creadores de la POA, define un aspecto como una entidad modular que se encuentra diseminada a través de otras unidades funcionales y reconoce que estas entidades no solo existen en el código de programación sino también en etapas tempranas como el diseño.

La POA no sustituye al paradigma con el que se desarrolla la funcionalidad principal de los programas; por el contrario, incorpora un nivel superior de abstracción, por lo que puede aplicarse a la POO o a la programación estructurada.

6.2 Historia de la separación de intereses

Dijkstra, en la década de los setenta introdujo el término "Separación de intereses" para referirse a la identificación, encapsulamiento y manipulación de partes del *software* que son relevantes para una funcionalidad particular. También en los setenta, David Parnas reconoció que la mejor manera de crear sistemas era a través de la identificación y separación de los intereses del sistema utilizando la modularización. Las primeras ideas orientadas a aspectos surgieron a principios de los noventa del grupo Demeter que trabajaba con la Programación Adaptativa, que es considerada como la versión inicial de la POA. En 1995, Cristina Lopes y Walter Huersch presentaron avances sobre la separación de intereses con técnicas como Filtros de Composición y Programación Adaptativa para los intereses transversales y pusieron en relieve el tema de la separación de intereses como un problema importante que la ingeniería de *software* debía resolver. En 1995 también se publicó la primera definición de aspecto por el grupo Demeter.

En 1996, Gregor Kickzales, de los laboratorios Xerox PARC -*Palo Alto Research Center*- creó el término de "Programación Orientada a Aspectos", delimitó su marco de trabajo y agregó precisión a la definición de un aspecto. El grupo de trabajo dirigido por Kickzales fue también creador de *AspectJ*, un lenguaje popular orientado a aspectos de propósito general basado en *Java*.

La POA es el resultado de la evolución de líneas de investigación previas, las cuales en algunos casos se han adoptado como técnicas de implementación de POA. Por ejemplo la meta-programación, métodos de reflexión, algunos patrones de diseño y técnicas de intercepción de mensajes.

6.3 La metodología orientada a aspectos

Laddad (2003) explica que el desarrollo de sistemas orientados a aspectos incluye tres fases: descomposición aspectual, implementación de los intereses, y recomposición aspectual.

> La descomposición aspectual es una etapa inicial en donde se extraen los requerimientos, se identifican y se ubican como intereses principales o intereses transversales. La fase de implementación es en donde cada uno de los intereses de la fase anterior se programa de manera independiente. En la etapa de recomposición aspectual se especifican reglas que relacionan los intereses principales y los transversales. Estas reglas permiten realizar la integración del sistema completo y generar un programa ejecutable para los usuarios finales, para quienes el uso de aspectos es imperceptible.

Para ilustrar estas fases, puede utilizarse la analogía de un rayo de luz que representa los requerimientos y que pasa por un prisma para descomponer los aspectos, implementarlos y después pasa por otro prisma para integrarlos de nuevo en un sistema final completo.

6.4 Fundamentos de la POA

Para realizar una implementación orientada a aspectos se requiere de un lenguaje base para programar la funcionalidad básica, de uno o varios lenguajes de aspectos para describir el comportamiento de los aspectos y de un tejedor que combine todos los lenguajes utilizados. El lenguaje base implementa el paradigma que mejor se adapte a las necesidades particulares para esa aplicación. Los lenguajes orientados a aspectos (LOA) se utilizan para encapsular los intereses transversales en aspectos. Los LOA están relacionados con el lenguaje base y deben ser compatibles con él. Es frecuente que los LOA sean extensiones del lenguaje base que incorporan soporte para aspectos.

La interacción entre el código de la funcionalidad base y el código de los aspectos se realiza mediante los puntos de enlace, los cuales son lugares específicos en el código base donde se agrega el comportamiento adicional descrito en los aspectos.

El tejedor es el encargado de combinar adecuadamente los módulos de la funcionalidad básica con los aspectos para producir el programa ejecutable; así como de llevar a cabo las tareas necesarias para la compilación. Para realizar este proceso, conocido como entretejido, el tejedor se guía por los puntos de enlace.

6.5 Entretejido estático y dinámico

El entretejido estático se realiza en tiempo de compilación, por lo que la ejecución del programa no se ve sobrecargada con procesos adicionales que disminuyen su eficiencia; también las comprobaciones se hacen en tiempo de compilación, lo que anticipa la aparición de posibles errores.

En el entretejido estático se genera un nuevo código fuente donde se insertan los aspectos en los puntos de enlace del código base. Para este propósito, el tejedor puede necesitar la conversión del código de aspectos al lenguaje base antes de integrarlo al nuevo código fuente, para luego compilarlo y obtener un ejecutable. Otra posibilidad es que el tejedor genere el ejecutable a partir del código base y del código de aspectos, en cuyo caso el entretejido se realiza a nivel *byte-code*.

Este tipo de entretejido es poco flexible ya que los aspectos están siempre fijos y no se acepta su modificación en tiempo de ejecución; tampoco es posible agregar nuevos aspectos o remover algunos ya existentes de manera dinámica. Sin embargo, resulta fácil de implementar y requiere menos recursos que uno dinámico.

En el entretejido dinámico los aspectos y las reglas deben estar disponibles tanto en tiempo de compilación como en tiempo de ejecución para que el LOA

pueda ejecutar el código de los aspectos dinámicamente. De esta manera, los aspectos pueden ser modificados, eliminados o agregados en tiempo de ejecución. Como consecuencia de este dinamismo, el rendimiento del programa disminuye, y también existe incertidumbre ante la aparición de nuevos errores.

6.6. Lenguajes orientados a aspectos

Los lenguajes orientados a aspectos (LOA) pueden ser de dominio específico o de propósito general. Los que son de dominio específico están diseñados para tratar con uno o más aspectos que pertenezcan a un determinado tipo, y no pueden manejar aquellos aspectos para los que no fueron considerados. Estos lenguajes tienen un nivel de abstracción superior al del lenguaje base y restringen su uso para asegurarse de que los aspectos sean escritos en el lenguaje aspectual.

Algunos ejemplos de LOA de dominio específico son: COOL -*Coordination Language*-, para el manejo de hilos concurrentes en los programas, RIDL -*Remote Interaction and Data-transfers Language*- para distribución, transferencia de datos e invocación remota, AML. Para cálculo de matrices dispersas, y RG para procesamiento de imágenes. Este tipo de lenguajes proporcionan una separación clara de intereses y obligan a que los aspectos sean utilizados adecuadamente, pero solo trabajan con aspectos pertenecientes a un dominio particular, por lo que el programador debe invertir mayor cantidad de tiempo para aprender varios lenguajes si desea implementar diferentes tipos de aspectos en un mismo sistema de *software*.

Por el contrario, un LOA de propósito general está diseñado para manipular cualquier tipo de aspecto, y adopta el mismo conjunto de instrucciones del lenguaje base sin aplicar restricciones a su uso. Es decir, extiende las instrucciones del lenguaje base para añadir el soporte aspectual. Algunos LOA de propósito general son: para *Java*, *AspectJ*, *Java Aspect Component* y *AspectWerkz*; para C, *AspectC*; para C++, AspectC++; para *Smalltalk*, *AspectS* y *Apostle*; para *Phyton*, *Phythius*; para *Ruby*, *AspectR*. Un programador puede aprender a utilizar un LOA en menos tiempo si conoce bien el lenguaje base. Los LOA proporcionan un entorno adecuado para programar cualquier tipo de aspectos; sin embargo, pueden permitir que los aspectos implementen también funcionalidad básica, lo que es una desventaja.

6.6.1 Características principales de un LOA

Algorry (2005) identifica las siguientes características que debería proporcionar una herramienta para la POA. Están basadas en los trabajos de Robert Filman *(NASA Ames Research Center)*, Daniel Friedman *(Indiana University)*, Awas Rashid y Katharina Mehner (Universidad de Lancaster).

- **Soporte para intereses transversales.** La herramienta debe integrar nuevos componentes y/o semántica especial para componentes existentes que permitan definir intereses transversales.
- **Entretejido.** La herramienta debe proveer entretejido estático, dinámico o ambos; tal vez utilizando un pre-procesador, compilador, cargador, un compilador JIT *-Just in Time-* o una máquina virtual.
- **Prescindencia.** El programador debe ser capaz de escribir su código base sin que deba conocer detalles adicionales del comportamiento de los aspectos.
- **Mecanismos de cuantificación.** La cuantificación es la posibilidad de indicar los puntos de unión en donde se aplicará un aspecto sin necesidad de enumerarlos uno por uno (Kicillof, 2008). La cuantificación se refiere al tipo de condiciones permitidas por la herramienta; y puede clasificarse en estática y dinámica.

La cuantificación estática es la que surge de la estructura del código y se determina cuando se cumple la condición descrita en él. Por ejemplo: "Ejecute la acción A cuando se produzca una llamada a un método cuyo nombre comience con *set*" -Caja negra-, o bien: "Ejecute la acción A cada vez que se produzca una asignación a una variable del tipo *integer* dentro de un ciclo *while*" -Caja blanca-. En la cuantificación dinámica las condiciones se expresan en términos de sucesos o historia de sucesos que ocurren al momento de la ejecución del código. Por ejemplo: "Ejecute la acción cuando se produzca una excepción" o "Ejecute la acción cuando se produzcan 3 intentos de introducción de contraseña".

- Mecanismos de interacción y resolución de conflictos.- La herramienta debe anticipar situaciones especiales y proporcionar alternativas para manejarlas. Ejemplos de algunas situaciones especiales son: cuando varios aspectos interactúan simultáneamente, o cuando hay aspectos que actúan sobre otros aspectos. Una posible alternativa de solución sería determinar un orden de aplicación de los aspectos para garantizar un comportamiento deseable y estable.

6.7 Los programas en lenguajes orientados a aspectos

En un programa orientado a aspectos, las sentencias de código deben ser del tipo: "En programas P cuando se presente la condición C, ejecute la acción A" (Filman & Friedman, 2000). Es importante notar que:

- La acción A representa la funcionalidad de un aspecto.
- La POA no forma parte de los programas P, los cuales pueden ser escritos a través de la orientación a objetos.
- La POA modifica externamente el comportamiento de los programas P.
- La acción A se incorpora a los programas P en el proceso de entretejido.

- La acción A debe proveer una interfaz para interactuar con los programas P.
- La condición C debe estar respaldada por un mecanismo de cuantificación.
- La acción A debe estar escrita en un algún lenguaje, que suele ser el mismo usado para escribir los programas P.

Para lograr esta estructura en las sentencias, los programas orientados a aspectos utilizan los siguientes conceptos adicionales a los requeridos por el paradigma con el que se implementó la funcionalidad base: Puntos de enlace o *Join points*, puntos de corte o *Pointcuts*, consejos o *Advices*, introducciones y aspectos. A continuación se explican cada uno de ellos.

- **Puntos de enlace.** Son ubicaciones definidas en la ejecución del código principal donde se puede realizar la integración de un aspecto. Un punto de enlace puede ser de diferentes tipos dependiendo del lenguaje. Por ejemplo en *AspectJ* existen puntos de llamada a métodos y constructores; puntos de retorno posteriores a la ejecución de métodos y constructores; puntos *get* y *set*; puntos de ejecución de manejadores de excepciones y puntos de inicialización estática y dinámica.
- **Puntos de corte.** Especifican las condiciones que deben cumplirse para que la integración con el programa principal se lleve a cabo. Su principal meta es indicar al LOA el momento para realizar la composición con un punto de enlace. Puede ser que el punto de corte tenga un nombre que lo identifique, o que sea anónimo.
- **Consejo.** Es el código que se va a ejecutar en el momento en que se alcanza un punto de corte. Se puede especificar la ejecución del consejo antes, después o en lugar del punto de corte.
- **Introducciones.** Permiten agregar miembros nuevos como por ejemplo métodos, constructores, o atributos en las clases y cambiar las relaciones de herencia.
- **Aspectos.** Son unidades de encapsulamiento, equivalentes a las clases en POO, que contienen varios puntos de corte, introducciones y consejos.

6.8 Aplicaciones de la POA

Con POA se puede agregar código a uno o más métodos al mismo tiempo, modificar el comportamiento de un método, agregar nuevos elementos a una clase, y modificar las jerarquías dentro del programa. Esto resulta útil en la implementación de patrones de diseño de manera más transparente en los programas y en la introducción de advertencias de compilación adicionales para el programador. También es posible impedir la ejecución de un método, o reintentar su ejecución bajo ciertas condiciones; esto resulta relevante al implementar intereses relacionados con la seguridad.

Existen situaciones relacionadas con el aseguramiento de la calidad en donde la POA también resulta útil. Por ejemplo: análisis de escenarios no invasivos del tipo "qué pasaría si", simulación de escenarios complejos agregando fallas al sistema para probar su respuesta, y soporte de traza en los reportes de errores, es decir, bitácoras no invasivas para comprender el comportamiento del sistema (Laddad, *Aspect-Oriented Programming will improve quality,* 2003). Las técnicas de POA permitirían eliminar con facilidad cualquier elemento que se haya agregado para estos objetivos. Esto puede hacerse aún si el sistema completo no fuera orientado a aspectos.

VII.

CONSIDERACIONES SOBRE OBJETOS Y ASPECTOS

7.1 Introducción

La Programación Orientada a Aspectos (POA) puede extender a muchos otros paradigmas existentes, sin embargo, es habitual que se utilice en conjunto con la Programación Orientada a Objetos (POO); de esta manera, un esquema de trabajo cotidiano involucra la POA para encapsular la funcionalidad secundaria dispersa y la POO para realizar la funcionalidad base del sistema.

Aunque la POO es eficaz al clasificar conceptos comunes y organizarlos para la construcción de programas, no maneja bien los conceptos entrecruzados que involucran módulos sin relación. Por el contrario, la POA separa los intereses dispersos de la funcionalidad base, proponiendo superar los problemas de la POO. Es así como la combinación POO-POA sugiere una combinación efectiva para el desarrollo de *software*.

La POA trabaja a un nivel superior de abstracción de la POO y permite manejar los intereses dispersos de una mejor manera; sin embargo, puede introducir nuevos problemas al escenario de desarrollo. También es importante notar que la separación completa de intereses propuesta por la POA promueve la facilidad de mantenimiento y la evolución general del sistema.

En los siguientes apartados se analizarán beneficios, problemas y estado actual de ambos paradigmas de programación.

7.2 Beneficios y problemas de la POO

La POO proporciona una manera de programar más natural y cómoda que la de otros paradigmas como la programación estructurada; este nuevo nivel de abstracción es uno de los mayores beneficios que brinda la POO al programador. La versatilidad y aplicabilidad del paradigma son otros puntos a su favor. Por su naturaleza, es posible abordar un amplio rango de problemas en distintas áreas con diversos grados de complejidad. La reutilización del código obtenida como consecuencia de las jerarquías y el encapsulamiento es otra importante ventaja de la POO que minimiza los costos asociados al tiempo de desarrollo. Por otra parte, realizar un buen programa orientado a objetos no es tarea fácil o intuitiva. Por ejemplo, el uso de patrones da lugar a una arquitectura sólida pero también compleja y confusa en los programas.

A pesar de que la POO ofrece una buena manera de organizar los programas y proporciona un buen nivel de claridad si se lleva a cabo de forma correcta, muchas veces resulta difícil alcanzar esta organización limpiamente. Por ejemplo, como se mencionó en el capítulo anterior, manejar los intereses dispersos con POO da lugar al código disperso y al código enredado, que agregan confusión al código del sistema resultante. El verdadero encapsulamiento en POO no puede ser implementado

por completo, y las metas de alta cohesión y bajo acoplamiento no se alcanzan del todo fácilmente. De esta manera, para agregar o modificar funcionalidad secundaria en un programa orientado a objetos se requiere el acceso total al código de la funcionalidad básica. Esto dificulta el entendimiento del código así como su correcta modificación y reutilización. Diversos enfoques como la programación generativa, la metaprogramación y la POA son soluciones que buscan mejorar estas inconveniencias propias de la POO.

7.3 Estado actual de la POO

Hoy en día la POO es un tema bastante estudiado y abordado; también existen muchos lenguajes que soportan su implementación; de hecho, resulta difícil encontrar un lenguaje comercial moderno que no soporte la orientación a objetos.

La POO se aborda en el salón de clases como una asignatura típica dentro de la retícula de programación de las carreras informáticas, en lugar de, o adicionalmente a la programación estructurada. Los contenidos académicos están bien documentados y discutidos en diversas fuentes de información. Por otra parte, hoy en día, la orientación a objetos se ha convertido en la opción por defecto para el desarrollo cotidiano de aplicaciones de negocios gracias a la madurez plena de la que goza el paradigma.

Desde hace algunos años se han adoptado estándares para el desarrollo orientado a objetos, y en este proceso han sido de importancia crucial los organismos que los promueven, como el OMG -*Object Management Group*-. Como ejemplo se puede mencionar al UML: el lenguaje gráfico de modelado estándar aceptado para el análisis y diseño orientado a objetos, cuya especificación se encuentra siempre en revisión y actualización coordinadas por OMG.

Así mismo se han desarrollado metodologías estandarizadas que hacen realidad un ciclo de vida completo con actividades orientadas a objetos compatibles entre sí, con lo cual el proceso completo se ha vuelto consistente. También se han identificado maneras de abordar situaciones comunes en los programas orientados a objetos y así han nacido los patrones, que bien utilizados en los programas, contribuyen a crear una arquitectura robusta.

Las herramientas CASE se han apegado a la orientación a objetos y ofrecen diversas opciones de ayuda al equipo de desarrollo. Por ejemplo, existen herramientas de diseño que realizan modelos completos basados en UML y ofrecen la posibilidad de codificar las clases en diversos lenguajes de programación a partir de los diagramas de entrada.

7.4 Beneficios y problemas de la POA

Los beneficios del uso de la POA son derivados de la forma en la que se modularizan los conceptos y se traducen en una mejoría global de la calidad del *software*. La

separación de intereses puede aumentar la productividad del equipo de trabajo y puede contribuir a utilizar los recursos más eficientemente; como consecuencia, los costos de desarrollo y mantenimiento pueden reducirse también.

La orientación a aspectos en los programas facilita la abstracción y reduce las dependencias entre los módulos; también aporta sencillez a la construcción, pruebas y evolución de los sistemas sin alterar su desempeño. Gracias a la POA es posible eliminar el código duplicado y solucionar las inconveniencias del código disperso y del código enredado, por lo que promueve *software* más ordenado y que resulta más fácil de reutilizar y refactorizar.

El proceso de conocer la OA, así como el diseño y la selección correcta de aspectos requieren de práctica, lo cual implica una curva de aprendizaje inicial que podría ser inconveniente para algunos proyectos y para algunos equipos. Sin embargo, una vez adquiridos los conocimientos de POA, se pueden extender sin problemas hacia otras aplicaciones.

Experiencias previas indican que con POA se pueden introducir efectos secundarios negativos al realizar modificaciones en el sistema. Por ejemplo, existen los denominados choques de código, que representan conflictos de nombres o de funcionalidades que producen resultados inesperados y son causados por varios aspectos que actúan sobre un mismo bloque de código. También con la POA resulta posible la introducción de código malicioso en los sistemas. Esto se lograría a través de aspectos con acciones malintencionadas, en especial cuando la seguridad en el equipo de trabajo no es confiable.

La depuración de los programas orientados a aspectos podría parecer problemática pues el flujo del programa puede tener varios brincos debido a la presencia de los aspectos; sin embargo, las herramientas actuales proveen medios que hacen que la tarea de depuración sea similar a la que se realiza en la orientación a objetos.

7.5 Estado actual de la POA

La comunidad de usuarios de la POA ha crecido en los últimos años. Las herramientas disponibles para este paradigma y los lenguajes de programación también se incrementan y mejoran día a día. *AspectJ* para *Java* se ha convertido en un lenguaje popular de referencia para realizar programas orientados a aspectos. La POA se considera el paso natural que sigue a la POO, pues aprovecha su potencial, al mismo tiempo que maneja bien la separación de intereses dispersos. Sin embargo, aún no se adopta en el entorno empresarial con suficiente madurez; tal vez por lo reciente del paradigma, o por la falta de técnicas pertinentes para todas las actividades del ciclo de vida del *software*.

La POA ha sido el centro de atención de las investigaciones académicas y es un tema actual entre algunos desarrolladores de *software*. Las instituciones educativas han comenzado a ofrecer capacitación sobre POA en forma de asignaturas opcionales, conferencias o talleres en los planes de estudio de las universidades en México.

El número de libros publicados de POA sigue siendo muy escaso en comparación con los libros que abordan la POO. Sin embargo, las editoriales ya ofrecen diferentes títulos sobre la orientación a aspectos desde la perspectiva de *AspectJ*. Algunos textos de ingeniería del *software* han incorporado la OA en sus contenidos y se han convertido en una fuente de referencia inicial a este paradigma. Hoy en día, dentro de las actividades del ciclo de vida se están incluyendo los aspectos de diversas maneras en las fases de desarrollo de *software*.

También han surgido guías de diseño orientado a aspectos que describen usos incorrectos de la POA, patrones, y soluciones orientadas a aspectos para problemas específicos.

7.6. Incorporación de objetos y aspectos en las primeras etapas del ciclo de vida del *software*

La incorporación de objetos y aspectos en las primeras etapas de desarrollo representa un importante campo de estudio para la homogenización inicial de conceptos en las actividades de desarrollo. Además, si se utilizan objetos y aspectos durante el desarrollo completo, se mantiene una consistencia generalizada en el proceso; de esta manera se conserva la misma perspectiva en el problema y en la solución facilitando la trazabilidad y por ende, el trabajo al desarrollador. En este contexto, las etapas tempranas, entendidas como la especificación de requerimientos y el diseño, requieren una atención especial, pues es en ellas donde se sientan las bases completas para la construcción de un nuevo sistema.

Un requerimiento es la descripción de una característica que un sistema es capaz de realizar para satisfacer un propósito global. La especificación de un conjunto de requerimientos define el marco de trabajo de un problema para resolver. El primer paso para lograr esta especificación es realizar un examen detallado de las funcionalidades deseables del sistema, para posteriormente documentar y revisar los requerimientos capturados.

El diseño es el proceso creativo de transformación del problema en una solución (Lawrence, 2013). Es así como se pueden elaborar múltiples diseños, todos correctos, para satisfacer una misma especificación de requerimientos. El resultado de la etapa de diseño involucra documentación de tipo conceptual y técnico; cada una pensada para una audiencia distinta: el diseño conceptual está dirigido a los clientes y el técnico hacia los desarrolladores de sistemas.

7.6.1 Incorporando objetos

La orientación a objetos no debe verse como un ciclo de vida por sí mismo, sino como una filosofía de representación que puede ser aplicada a cualquier modelo de ciclo de vida de *software* que sea elegido para el desarrollo. Es el modelo elegido el que determina la secuencia de las actividades en las que se usará la orientación a objetos.

7.6.1.1 Objetos y UML

UML, *Unified Modeling Language* -Lenguaje Unificado de Modelado por sus siglas en inglés- es una notación orientada a objetos, aceptada y estandarizada por OMG desde 1997; es el resultado de la fusión de varias de las notaciones más importantes existentes para orientación a objetos entre la década de los ochenta y noventa. Algunos de los autores líderes que proponían sus propios lenguajes gráficos orientados a objetos en ese tiempo y que se consideran pioneros de UML son: Grady Booch, Ivar Jacobson, y Jim Rumbaugh.

El Lenguaje Unificado de Modelado es una notación que se utiliza para describir *software* desde diferentes perspectivas; puede representar diferentes tipos de sistemas y se ha utilizado incluso para modelar procesos de negocio o flujos de trabajo. El UML no es una metodología, por lo tanto, no representa una manera para desarrollar *software* sino para especificarlo. UML es independiente de cualquier metodología de desarrollo de *software*, aunque es compatible con la gran mayoría de ellas debido a su flexibilidad y facilidad de uso. UML es un lenguaje que permite expresar distintos aspectos relacionados con el *software* y de esta manera, promueve la comprensión y documentación de los sistemas, así como la comunicación acerca de ellos.

En UML existen bloques básicos para representar elementos, relaciones y diagramas (Jacobson, Booch, & Rumbaugh, 2005). Los elementos son entidades relevantes para una situación, las relaciones son enlaces entre ellos y los diagramas son agrupaciones de elementos y relaciones. Los diagramas pueden verse como modelos acerca del *software* que permiten visualizar un sistema desde diferentes aristas. Existen trece diferentes tipos de diagramas en UML: clases, objetos, componentes, estructura compuesta, casos de uso, secuencia, comunicación, estados, actividades, despliegue, paquetes, tiempos, visión global de interacciones. Cada uno puede utilizarse de forma independiente del resto; es decir, no es obligatorio aplicar los trece tipos en todos los escenarios.

Los siguientes diagramas son los que más se usan en el proceso de creación de *software*: casos de uso, secuencias, actividades y clases.

1. **Diagramas de casos de uso.** Representan escenarios en que se puede utilizar un sistema. Incluyen la relación del usuario y actores con funcionalidades que ofrece el *software*. Sirven para documentar los requerimientos de los usuarios y el comportamiento del sistema.

2. **Diagramas de clases.** Estos diagramas sirven para plasmar las clases, que son los elementos fundamentales para la construcción de programas orientados a objetos. Un diagrama de clases muestra características y acciones que están asociadas a un tipo de objetos y también las relaciones que existen entre varias clases de un mismo sistema.
3. **Diagramas de secuencias.** Los diagramas de secuencias permiten mostrar cómo se comunican los objetos derivados de las clases con un enfoque en el orden temporal en el que esta comunicación se lleva a cabo. Estos diagramas también son útiles para apreciar la información que se envía y se recibe en los objetos participantes.
4. **Diagrama de actividades.** Los diagramas de actividades muestran el flujo de un proceso y detallan cada una de las actividades que se llevan a cabo. Poseen elementos para representar procesos, condiciones y repeticiones. Estos diagramas se consideran una versión evolucionada de los tradicionales diagramas de flujo que se popularizaron con la programación estructurada hace ya algunas décadas.

Puede utilizarse UML durante el proceso completo de desarrollo, ya sea para visualizar, especificar, comunicar o documentar, según se requiera. UML, mediante sus diagramas, permite representar las vistas estáticas y dinámicas del sistema, así como restricciones y formalización. La vista estática se logra con los siguientes diagramas que muestran la estructura del sistema: de clases, objetos, paquetes, despliegue, componentes y de estructura compuesta. La vista dinámica se representa mediante los siguientes diagramas que muestran el comportamiento del sistema: de casos de uso, actividades, estados e interacciones (secuencia, comunicación, tiempos y vista de interacción). Las restricciones y formalización se expresan con OCL -*Object Constraint Language*- Lenguaje de Restricción de Objetos, por sus siglas en inglés, que es un lenguaje estandarizado de especificación formal.

7.6.1.2 Los objetos en la especificación de requerimientos

Al inicio de la etapa de requerimientos se realiza un análisis escrito de las necesidades que el sistema debe cubrir, utilizando el lenguaje del usuario. De esta manera se incluyen conceptos y escenarios que familiarizan a los desarrolladores con el contexto del sistema al mismo tiempo que describen los requerimientos de una manera entendible por el usuario.

Posteriormente, los requerimientos de un sistema se especifican con los diagramas de casos de uso de UML, cada uno de los cuales muestra un escenario del sistema mediante una funcionalidad específica representada gráficamente. Es importante subrayar que el conjunto de todos los casos de uso identificados representa la funcionalidad completa del sistema.

Los elementos de un diagrama de caso de uso son: actores, que son entidades relacionadas con el sistema; casos, que representan funcionalidad visible para el actor; extensiones, que son especializaciones de un caso de uso, y las inclusiones, que son usos de casos de uso ya definidos.

Los casos de uso son un medio eficaz de comunicación con clientes y miembros del equipo, y son la pauta a seguir en todo el proceso de desarrollo. Los casos de uso también muestran distintas perspectivas de los requerimientos del sistema, y de esta manera, ayudan a encontrar defectos en ellos. De ahí la importancia de la adecuada representación de la información en los casos de uso.

Aunque una correcta descripción de requerimientos en lenguaje natural es traducida a casos de uso con facilidad, debe actuarse siempre con cautela, ya que se pueden esconder detalles importantes y dejar otros tantos en la vaguedad. Por eso, una vez especificados los casos, es necesario revisarlos de nuevo para encontrar ambigüedades, identificar dependencias, asegurar completitud y prever posibles problemas potenciales.

Para respaldar cada uno de los diagramas de casos de uso se crea un documento que contiene un flujo de eventos escrito desde el punto de vista del actor. El documento, redactado con lenguaje del dominio del problema, describe de manera sencilla la interacción entre el actor y el sistema, sin mencionar los detalles de implementación. El documento contiene acerca del caso de uso: su objetivo, la manera como comienza y termina, los momentos de interacción entre los actores y él, y el flujo de eventos normal, alternativo y excepcional.

El flujo de eventos normal describe el proceso llevado a cabo en el escenario más típico. El flujo de eventos alternativo incluye las variantes del flujo normal y casos especiales. El flujo excepcional describe el manejo de situaciones de error. Estos flujos se describen utilizando texto simple informal, texto formal estructurado con precondiciones y post-condiciones, o bien, pseudo-código.

El flujo de eventos se puede documentar también usando tablas con columnas: en una columna se escribe el flujo normal; en otra los flujos alternativos y en otra el flujo que considera las excepciones ocurridas. Otra opción es utilizar diagramas de estados, secuencias o actividades que hagan referencia al caso de uso a documentar.

De forma conceptual, la especificación de requerimientos debería realizarse de una manera orientada a objetos para dar inicio a la etapa de diseño. Sin embargo, la especificación de requerimientos puede traslaparse con los primeros pasos del diseño, ya que en ambos casos, se deben identificar objetos y las relaciones entre ellos. Por esta razón, en el desarrollo orientado a objetos a menudo se omite escribir una especificación de requerimientos diferente del documento de definición de requerimientos (Lawrence, 2013).

7.6.1.3 Los objetos en el diseño

El diseño orientado a objetos se inicia explorando los requerimientos para localizar candidatos para convertirse en clases. Los principales blancos de atención son aquellos sustantivos que conlleven a: estructuras, sistemas externos, dispositivos, roles, procedimientos, lugares, organizaciones, y en general, los elementos que manipula el sistema a construir. De esta manera se identifican clases y atributos, los cuales luego deben refinarse al examinar otros requerimientos. Suelen aparecer nuevos elementos durante el proceso mientras que otros desaparecen. También se deben identificar los comportamientos relacionados, para lo cual se centra la atención en los verbos: acciones, eventos, procedimientos o servicios.

A lo largo del proceso es necesario representar las clases y las relaciones existentes entre ellas con varios niveles de detalle, para lo cual se utilizan diagramas de clases de UML. Adicionalmente, la colaboración entre los objetos se representa con los correspondientes diagramas UML que muestran la vista dinámica del sistema.

El refinamiento sucesivo en combinación con los diagramas de UML produce diferentes versiones del diseño del sistema. Cada versión se encuentra mejorada en comparación con su predecesora y realiza aportaciones más apegadas al diseño definitivo. Los resultados del diseño del sistema se utilizan para realizar el diseño de programas, que se encuentra en un nivel más detallado pues incluye características de implementación que no siempre son visibles al usuario; por ejemplo, las estructuras de datos que los programadores deben tomar en cuenta, las interfaces de los objetos, y las relaciones de herencia o composición.

En el diseño de programas se deben incorporar también la interfaz de usuario, la gestión de datos y la gestión de tareas; y es común que se adopte el uso de patrones o buenas prácticas de diseño orientado a objetos para enfrentar situaciones recurrentes.

Para el diseño de la interfaz de usuario se realizan esbozos en papel o electrónicos, los cuales deben contener los elementos visuales que el usuario va a utilizar para interactuar con el sistema. Estos prototipos deben ser revisados y aceptados por el cliente antes de implementarse usando las clases correspondientes. En relación con la gestión de los datos en el programa, se debe especificar la manera y los métodos de acceso que serán utilizados sin importar si se trata de archivos, bases de datos relacionales o bases de datos orientadas a objetos. Así mismo, durante el diseño de la gestión de tareas se especifica la manera de coordinar las actividades que el sistema va a realizar; puede tratarse de tareas dirigidas por eventos, o por tiempos, de manera recurrente. En ambos casos es importante que cada tarea sea expresada con el nivel de detalle apropiado para evitar confusiones en la implementación.

7.6.2 Incorporando aspectos

El modelado de aspectos se puede realizar utilizando técnicas generales de modelado de intereses, o técnicas particulares para orientación a aspectos. El modelado general de intereses puede utilizarse para todo tipo de programas ya sean orientados a aspectos o no, partiendo de la idea que un programa realiza varios intereses distintos. Por el contrario, las técnicas particulares de orientación a aspectos buscan una consistencia en la manera de manejar los intereses transversales desde el modelado hasta la implementación; y están influenciadas por uno o varios modelos de implementación orientados a aspectos. Las técnicas de desarrollo orientadas a aspectos tienen por objetivo proporcionar soporte para la identificación, modularización y composición de intereses transversales en todas las actividades del desarrollo de *software* (Blair, y otros, 2004).

Las representaciones de aspectos en las primeras etapas de desarrollo de *software* se han denominado "aspectos tempranos" y han adoptado los principales enfoques de los modelos de programación existentes en la POA y han dado lugar a diversas formas de modelado. A pesar de que la orientación a aspectos es muy utilizada en la fase de programación, aún el modelado de los intereses no se realiza de manera estandarizada en todo el ciclo de vida. Existen propuestas unificadoras recientes que persiguen el soporte y modelado de aspectos a través de todas las fases del ciclo de vida; un ejemplo es el trabajo de Pinciroli y Zeligueta (2017) que utiliza un marco de desarrollo de *software* orientado a aspectos denominado AOP4ST.

El desarrollo de software orientado a aspectos persigue que los intereses se puedan identificar y modelar en sus propios formalismos de manera separada en cada fase del proceso (Sutton & Rouvellou, 2004). Debe tenerse en cuenta que el uso de aspectos en el desarrollo de *software* no es obligatorio; es decir, si es posible encapsular con suficiente limpieza la funcionalidad de los aspectos dentro del programa base, se deberían utilizar solo las técnicas tradicionales.

7.6.2.1 Los aspectos en la especificación de requerimientos

En la etapa de identificación de requerimientos, primero se buscan candidatos para formar aspectos; aquellos intereses con amplio alcance, así como las restricciones del sistema son buenos prospectos. Después, se requieren técnicas para proporcionar mecanismos de abstracción y composición para modularizar y componer esos intereses. Algunas de ellas logran la separación de intereses mediante la división de requerimientos en funcionales y no funcionales. Siendo entonces, los no funcionales, candidatos para ser aspectos.

Existen diferentes enfoques que soportan la separación de intereses en el análisis orientado a aspectos. Algunos, de manera genérica como *Kaos*, y *PREVIew* y otros con soporte directo para aspectos como AOCRE, AORE y *Theme*.

7.6.2.2 Los aspectos en el diseño

En la literatura se ha reportado que la orientación a aspectos debe tomarse en cuenta en todas las fases del ciclo de vida del *software*, pero de manera especial debería considerarse en el diseño (Clemente, Hernández, Herrero, Murillo & Sánchez, 2004).

Los programadores convencionales encuentran difícil la creación de programas orientados a aspectos hasta que desarrollan experiencia y se adaptan a plantear soluciones en este paradigma. Esta situación se agrava cuando el diseño del sistema se realiza sin considerar aspectos, los cuales, son incorporados hasta la fase de programación. En este escenario, el programador está obligado a rediseñar módulos del sistema para poder introducir los aspectos usando algún lenguaje de programación. Esto puede causar más conflictos cuando en algunas organizaciones no se permite que el programador realice actividades de diseño. La incoherencia de un diseño sin aspectos para un programa orientado a aspectos produce ambigüedad entre las tareas y responsabilidades de los diseñadores y los programadores; esto afecta la facilidad de depuración y mantenimiento del sistema pues a primera vista no queda claro si las adecuaciones de los diseñadores afectarán el trabajo de los programadores y viceversa.

Algunos autores sugieren características deseables en las técnicas de diseño orientadas a aspectos. Por ejemplo, las técnicas deberían contar con notaciones que permitan encapsular requerimientos transversales, proporcionar facilidad de entendimiento para sistemas complejos y promover la reutilización y la evolución de los modelos (Blair, y otros, 2004). Las propiedades deseables para un esquema de modelado de intereses incluyen capacidad de capturar diversos tipos de intereses, independencia para minimizar el impacto de otras herramientas y técnicas aplicadas, capacidad de expresar sin dificultad la información necesaria acerca de cualquier interés, completitud que permita expresar todos los elementos necesarios del modelo y facilidad de uso para facilitar la adopción y uso del esquema de modelado (Sutton & Rouvellou, 2004).

VIII.

REUTILIZACIÓN DE *SOFTWARE*

8.1 Introducción al concepto de reutilización

Reutilizar *software* significa utilizar repetidamente, en contextos diferentes, algún elemento necesario para un sistema de *software*. En otras palabras, es la capacidad en la que un producto puede reciclarse en otros desarrollos de aplicaciones (Castro, Rivera, Fernández & Acevedo, 2017). La reutilización de algoritmos y código fuente es muy común; sin embargo, también se pueden reutilizar requerimientos, especificaciones, diseños y pruebas, así como artefactos y esquemas de bases de datos. Es posible distinguir dos perspectivas diferentes de esta definición: el desarrollo con elementos reutilizables y el desarrollo de elementos reutilizables. Cuando se desarrolla un sistema con elementos reutilizables, la actividad central consiste en incorporar al nuevo proyecto elementos de *software* previamente elaborados en lugar de construirlos desde el principio. Por el contrario, el desarrollo de elementos reutilizables se encarga de crear aquellos elementos que después serán reutilizados en otros sistemas.

8.1.1 Historia de la reutilización de *software*

Fue McIlroy, en 1968, quien propuso una biblioteca de componentes reutilizables para cálculos numéricos e introdujo así por primera vez el concepto de reutilización formal de *software*. En la propuesta de McIlroy era posible construir grandes sistemas a partir de pequeños componentes que se podían acceder desde un repositorio. Esta idea proporcionó los fundamentos para el concepto de Fábricas de *software*, el cual establecía un conjunto de procedimientos repetibles en donde se reutilizaba la mayor cantidad de elementos posibles, incluyendo la experiencia. A mediados de la década de los setenta, Robert Lanergan inició el primer proyecto de reutilización formal en una organización. Para finales de esa misma década, la reutilización había ocupado un lugar importante como tema de investigación y fue así como se llevó a cabo el primer taller internacional de reutilización.

En la década de los ochenta creció todavía más el interés por la reutilización. Surgieron abundantes programas de reutilización promovidos por los gobiernos de Estados Unidos y Europa, así como de grandes organizaciones en Japón y en Estados Unidos. En 1988 la definición de reutilización fue delimitada por Basili, quien incluyó cualquier elemento relacionado con un proyecto de *software*, incluso conocimiento. Para finales de los ochenta hubo contribuciones relacionadas con el manejo de librerías, las técnicas de clasificación, la creación y distribución de componentes.

En la década de los noventa, Prieto-Díaz (1993) realizó taxonomías importantes acerca de las diferentes formas de reutilización que aún hoy en día son aceptadas. A partir de esta década, también se resaltó la importancia de los factores económicos, administrativos, culturales y legales relacionados con la reutilización. En las décadas posteriores, la reutilización se ha reconocido como un proceso institucional y bien

definido. Hasta la fecha se siguen realizando propuestas para mejorar y aplicar las tecnologías de reutilización y se conducen estudios para conocer sus tendencias de implantación y adopción en las organizaciones de desarrollo de *software*.

8.1.2 Objetivos de la reutilización

Desde sus inicios, la reutilización ha sido concebida como una manera de enfrentar la "crisis del *software*", identificada por primera vez a finales de los sesenta como la causante de proyectos fallidos, que resultan de mala calidad, defectuosos, retrasados o fuera de presupuesto. Al reutilizar elementos de *software* con calidad y funcionalidad comprobada, se puede disminuir el tiempo, el costo y el esfuerzo de desarrollo, con lo que es posible encausar los recursos del equipo de trabajo hacia la satisfacción de los requerimientos concretos de los usuarios del nuevo sistema de *software*.

8.1.3 Reutilizar para ganar e invertir para reutilizar

La reutilización de *software* genera beneficios económicos a mediano y largo plazo. Sin embargo, para implementar un programa de reutilización se debe contar con una inversión inicial elevada, relacionada con la obtención del *software* reutilizable, el uso del *software* obtenido, y la implantación del proceso en la organización. Esta paradoja de la administración de la reutilización de *software* fue identificada a través de un proyecto realizado en *Nippon Telegraph and Telephone Corporation* (NTT), que exhibe la dificultad de obtener ganancias de la reutilización de *software* sin una inversión considerable de tiempo y esfuerzo; y por otra parte, muestra que es muy difícil justificar considerables cantidades de tiempo y esfuerzo sin ganancias tangibles a corto plazo (Isoda, 1995).

La puesta en marcha de un programa formal de reutilización también debe considerar factores que alteran la estructura de la organización. Por ejemplo, se deben adaptar las funciones del personal y de los equipos de trabajo; también los procedimientos deben agregar actividades propias del proceso de reutilización.

8.1.4 La reutilización desde diversas aristas

En el trabajo de Prieto-Díaz (1993) se clasifican las formas de reutilización a través de varias perspectivas: por esencia, por dominio, por modalidad, por técnica, por intención y por producto. Cada categoría se describe a continuación.

- **Por esencia.** Se refiere a las ideas y conceptos que representan soluciones, procedimientos y habilidades que pueden ser aplicadas nuevamente. También incluye aquellos elementos y componentes concebidos como parte de un todo.
- **Por dominio.** En esta perspectiva se puede hablar de reutilización horizontal y vertical. La vertical se produce entre sistemas que pertenecen al mismo dominio, en cuyo caso

puede existir un alto grado de reutilización. La horizontal se produce entre sistemas de diferentes dominios, por lo que la reutilización entre ellos es menos probable.

- **Por modalidad.** La reutilización puede ser planificada u oportunista. Es planificada cuando ésta es sistemática y con procedimientos definidos y es oportunista cuando ésta es sólo una práctica informal.
- **Por técnicas.** Existe la reutilización por composición y por generación. La primera incorpora en los nuevos sistemas, elementos existentes que se encuentran encapsulados. En la reutilización por composición, no hay componentes auto-contenidos, sino que el *software* es el resultado de un proceso que produce ciertas codificaciones pre-establecidas en función de condiciones de entrada.
- **Por intención.** En esta perspectiva se pueden identificar elementos de caja negra, caja blanca o caja transparente. En los elementos de caja negra, se conoce su interface y funcionalidad pero no su contenido, por lo que las modificaciones no son posibles. Si un elemento es de caja blanca, sí se puede modificar su funcionalidad. Las cajas transparentes permiten conocer su contenido, pero no permiten modificaciones.
- **Por productos.** Ejemplos de los productos del ciclo de vida de desarrollo de software que se pueden reutilizar son: requisitos, especificaciones, arquitecturas, documentación, código y pruebas. Estos productos pueden reutilizarse en varios contextos si su compatibilidad lo permite.

8.2 Tendencias en la reutilización

Existe un gran número de enfoques de reutilización. Tres de ellos representan las tendencias más importantes en esta área: la ingeniería de *software* basada en componentes (CBSE-*Component Based Software Engineering)*, la ingeniería de líneas de productos (PLE -*Product Line Engineering)* y el desarrollo basado en componentes listos para utilizarse (COTS-*Component Off The Shelf Based Development)*. A continuación se explica cada uno de ellos.

8.2.1 Ingeniería de *software* basada en componentes -*Component Based Software Engineering* (CBSE)-

Un componente se puede definir como una entidad coherente que ha sido encapsulada y que tiene interfaces y dependencias explícitas, además, que puede desarrollarse y distribuirse de manera aislada para después ser integrada en otros contextos. La ingeniería del *software* basada en componentes (CBSE) es una forma de reutilizar *software* que incorpora estos elementos encapsulados en el desarrollo de nuevos sistemas mediante el enfoque de reutilización de caja negra.

La CBSE es un proceso centrado en las interfaces y la composición. La interface de un componente permite conocer las maneras en las que es posible comunicarse

con él para utilizarlo. La composición es el proceso de integrar el componente a un nuevo sistema de *software* que utiliza su funcionalidad. Los componentes deben tener una interface estándar y bien definida que debe ser documentada y conocida por las aplicaciones que van a utilizarlos. Un componente puede caracterizarse en términos de sus interfaces. La anatomía de un componente consta de elementos internos, interfaces de aplicación e interfaces con la plataforma, los cuales se explican a continuación.

- **Elementos internos.** Forman la estructura privada y oculta de un componente. Se encargan de implementar la funcionalidad que ofrece el componente.
- **Interfaces de aplicación.** Las interfaces de aplicación son públicas e interactúan con otros componentes o aplicaciones y describen la estructura de los mensajes intercambiados entre ellos.
- **Interfaces con la plataforma.** Son las que definen la interacción del componente con la plataforma sobre la que se ejecuta. Son una descripción de los mensajes que el componente intercambia con el *hardware*, procesador, memoria, sistema operativo, entorno de ejecución, acceso a los periféricos y sistemas de comunicación.

Un componente debe ser bien documentado, cohesivo, acoplado y debe estar certificado. También debe estar apegado a un modelo de componentes; de esta manera, se asegura que puede ser parte de otro sistema que sea compatible con el mismo modelo. Además, el componente debe ser seguro, es decir, debe controlar los accesos hacia sus elementos internos y debe ser diseñado para interactuar de manera segura con otros sistemas de *software*.

8.2.1.1 Ciclo de vida CBSE

En el desarrollo basado en componentes se pueden identificar las siguientes actividades:

- **Encontrar.** Se llevan a cabo búsquedas de componentes entre los que están disponibles.
- **Seleccionar.** Se seleccionan componentes específicos para utilizarlos en el desarrollo de *software* basado en componentes.
- **Adaptar.** Se refiere a la adecuación de los componentes elegidos para incorporarlos al contexto en que se van a utilizar.
- **Crear.** Cuando la adaptación de componentes no fue suficiente para satisfacer los requerimientos de los clientes, es necesario crear componentes nuevos a partir de cero.
- **Componer.** Se refiere a las actividades relacionadas con la inclusión de los componentes al nuevo sistema.
- **Reemplazar.** Se refiere a eliminar una versión anterior de un componente para incorporar una nueva versión de éste.

8.2.2 Ingeniería de líneas de productos *-Product Line Engineering* (PLE)-

Busca reutilizar los elementos comunes de distintos proyectos en una misma área con el objetivo de lograr alta productividad, buena calidad, así como reducidos costos y tiempos de entrega. Con esta forma de reutilización se seleccionan, refinan y establecen prácticas para identificar y aprovechar elementos similares entre productos de un área de aplicación concreta (Mili, Mili, Yacoub & Addy, 2002). Es así como cada nuevo proyecto de *software* no es un esfuerzo independiente de los productos y experiencias de otros proyectos, sino una oportunidad de reutilizar las características comunes de un grupo de sistemas construidos dentro de un mismo contexto. Los productos desarrollados comparten una arquitectura de dominio común y evolucionan en conjunto con la línea completa. Cada elemento reutilizable mantiene una visión de reutilización futura en otras aplicaciones; sin embargo, la filosofía PLE considera que los elementos que son genéricos en su totalidad son difíciles de implementar en otras aplicaciones.

En una línea de productos se requieren tres elementos imprescindibles:

- **Análisis de dominio.** Su objetivo es obtener un modelo de dominio que incluya las características comunes y variaciones entre los miembros de la familia de la línea de productos.
- **Arquitectura de *software*.** Se refiere a una infraestructura estándar que es adoptada como arquitectura propia por cada producto de la línea.
- **Proceso de desarrollo.** Es el conjunto de actividades que se realizan para crear el *software*. Se pueden identificar actividades de la ingeniería de dominio y de la ingeniería de la aplicación.

8.2.2.1 Ciclo de vida PLE

El ciclo de vida PLE contiene actividades de ingeniería de dominio y de ingeniería de aplicación. La ingeniería de dominio se encarga de las actividades y productos que engloban la arquitectura común de los productos de la línea, mientras que la ingeniería de aplicación se ocupa del desarrollo individual de cada producto.

La elaboración de la arquitectura es una actividad muy importante para la implementación de una línea de productos. Los ingenieros de dominio analizan los modelos de dominio y establecen una arquitectura común, la cual luego se instancia en cada aplicación para convertirse en la arquitectura individual de cada producto desarrollado, también toma en cuenta sus especificaciones propias. Las características que la arquitectura adopte afectarán la calidad de la familia completa de productos. Si la arquitectura presenta problemas, los sistemas individuales también los tendrán. Por eso es importante destacar las características deseables de la arquitectura: robustez, flexibilidad y alto nivel de configuración.

Cuando se desarrolla un proyecto a partir de una línea de productos, la arquitectura base se instancia para ese proyecto particular. Concretamente, la instanciación en este contexto se refiere a:

- Tomar los elementos comunes que pertenecen a la arquitectura del dominio.
- Identificar los elementos que son específicos del producto que está desarrollándose.
- Validar que la arquitectura de aplicación del producto específico corresponda con la arquitectura de dominio.

Al instanciar elementos se deben cuidar dos situaciones: la correspondencia entre las arquitecturas base e instanciada, y la sincronización de las versiones de ambas arquitecturas. La primera sirve para determinar en la instancia el grado de satisfacción de las restricciones impuestas por la arquitectura de dominio. La segunda ayuda a revisar que los cambios realizados en elementos de la arquitectura base no afecten a los productos individuales.

8.2.3 Desarrollo basado en componentes comerciales -*Component Off the Shelf Based Development* (COTS)-

Es un enfoque de reutilización que busca incorporar a las aplicaciones los componentes que se encuentran disponibles comercialmente. Un componente COTS es un *software* que se vende o se permite usar bajo algún tipo de licenciamiento; el acceso a su código fuente está restringido para quien lo adquiere, pues el vendedor es quien lo desarrolló y también se encarga de su actualización.

El desarrollo COTS posee varias ventajas. Por ejemplo, ofrece ganancias económicas, pues se desarrolla una sola vez y se utiliza varias veces en diversos proyectos. Los componentes COTS suelen ser de buena calidad, ya que han sido sometidos a una gran variedad de pruebas; tienen buena funcionalidad, pues sus desarrolladores son expertos en el dominio del componente; promueven un desarrollo veloz, se pueden utilizar de inmediato; y liberan al cliente de tareas de mantenimiento, el vendedor es el encargado de solucionar los problemas con el componente.

Por otra parte, el desarrollo COTS también posee varias desventajas. Por ejemplo, un componente no podría utilizarse en sistemas en donde la seguridad y la confiabilidad sean críticas. Como el cliente no tiene acceso al código fuente, desconoce los algoritmos y técnicas que se utilizaron en su desarrollo. Por estas razones se dice que el desarrollo con componentes COTS privilegia la composición y la funcionalidad sobre el buen diseño.

8.2.3.1 Ciclo de vida COTS

En el ciclo de vida de una aplicación desarrollada mediante el enfoque COTS se pueden identificar las siguientes etapas: selección, integración, verificación-

validación y mantenimiento. Se considera que los requerimientos particulares de la aplicación ya han sido analizados, capturados y especificados antes de iniciar con la etapa de selección.

- **Selección.** Significa elegir los componentes más adecuados de las varias opciones disponibles. En este proceso se deben considerar atributos de calidad como eficiencia, confiabilidad, facilidad de integración y actualización. También se debe evaluar la pertinencia de la funcionalidad del componente para esa aplicación y se debe asegurar que no existan interacciones indeseadas con otros componentes del sistema.
- **Integración.** En esta fase, el componente seleccionado se agrega a la aplicación que lo utilizará. Se debe revisar que exista compatibilidad en las interfaces de los componentes, que exista concordancia en sus funcionalidades y que el desempeño del sistema integrado sea aceptable.
- **Verificación y validación.** En esta etapa se debe asegurar que el componente cumple con los requerimientos específicos de la aplicación. Por la naturaleza y filosofía COTS, se realizan pruebas de caja negra para evaluar la funcionalidad del sistema.
- **Mantenimiento.** Implica la obtención de versiones actualizadas de un componente. En esta fase se depende del vendedor y de su equipo de desarrollo, pues son ellos quienes conocen y modifican el código fuente para corregir errores, agregar nueva funcionalidad u optimizar alguna tarea. Las fechas de liberación de la nueva versión del componente, las características que se incluyen en ella y el costo son factores que están fuera del control de los ingenieros de aplicación. Cada vez que el vendedor entrega una nueva versión del componente, se deben realizar pruebas con él para asegurar que no se están agregando nuevos problemas al sistema de *software*.

8.3 Reflexiones finales sobre la reutilización de *software*

COTS y CBSD son los enfoques más parecidos en la práctica, ambos utilizan la composición como medio de reutilización de los componentes. Sin embargo, en COTS los componentes son desarrollados por terceros mientras que en CBSD son desarrollados en la misma organización. Es así como el código fuente del componente está disponible en CBSD pero no en COTS; como consecuencia, los tipos de pruebas que se realizan en CBSD pueden ser de caja blanca y caja negra, mientras que en COTS solo pueden ser de caja negra. Con un enfoque COTS, el autor, propietario y responsable de la actualización del componente es el desarrollador que lo comercializa; en CBSD es la organización en la que se desarrolla. Las actividades más representativas de cada uno de los enfoques también suele ser distinta: en CBSE es realizar adaptaciones a los componentes, en PLE es realizar instanciaciones de la arquitectura base y en COTS es adquirir los componentes.

IX.

REFLEXIONES FINALES

En este libro se presentó una visión global del desarrollo de *software* a través de temas relevantes en esta área. Se abordaron las metodologías tradicionales y emergentes, se profundizó sobre la programación extrema. Se presentaron dos casos de estudio sobre desarrollo ágil con programación extrema y *Scrum*. Se hizo una reseña de los principales paradigmas de programación existentes. Se hizo énfasis en las características de la orientación a objetos y la orientación a aspectos desde el punto de vista de implementación y de las primeras etapas de desarrollo del *software*. Se realizó una comparación entre la orientación a objetos y la orientación a aspectos, en donde se mostraron las ventajas, desventajas y estado actual de ambos paradigmas. También se estudiaron los distintos enfoques de la reutilización de *software*.

Las principales lecciones aprendidas son:

- El desarrollo de *software* consiste en una serie de actividades encaminadas a la creación de un programa de cómputo que los usuarios puedan utilizar. Existen diversas metodologías que organizan estas actividades de acuerdo con modelos y filosofías propias. En la actualidad son muy utilizados los enfoques ágiles, los cuales priorizan a las personas y encaminan sus esfuerzos hacia la obtención de un producto final funcional.
- La Programación Extrema es un enfoque metodológico ágil que recomienda un conjunto de prácticas para llevar a cabo la programación de aplicaciones informáticas. La programación por pares es una de sus prácticas más importantes; los programadores realizan su trabajo en parejas y ambos comparten una misma computadora.
- Los paradigmas de programación representan marcos de trabajo que facilitan diferentes formas de pensamiento sobre un mismo problema y sus posibles soluciones. Su evolución ha ofrecido nuevas formas de organizar los programas; cada una de ellas realiza aportaciones intentando superar o especializar características de sus antecesores.
- Todos los paradigmas de programación tienen ventajas y desventajas, y resultan apropiados en mayor o menor grado según sea el dominio, la magnitud y la complejidad del problema a resolver.
- Como resultado natural de la evolución de estos paradigmas, surgió la Orientación a Objetos (OO), que durante los últimos años se ha convertido en un medio estable, unificado, efectivo y versátil para manejar la complejidad de los sistemas, pues organiza los programas como un conjunto de abstracciones del mundo real comunicándose entre sí para lograr objetivos comunes.
- Actualmente, la OO tiene bases bien cimentadas y se ha convertido en la elección de la mayoría de empresas para desarrollar sus nuevos proyectos de software. El mayor beneficio de la OO se obtiene cuando se utiliza para organizar clases con elementos comunes mediante jerarquías, lo que contribuye a la reutilización del código. A pesar de que han surgido algunas metodologías de desarrollo pensadas para el paradigma orientado a objetos, las etapas de cualquier ciclo de vida elegido pueden adaptarse a la filosofía de objetos.

- La OO presenta inconvenientes; por ejemplo, se ha identificado la problemática de la implementación de intereses transversales en los programas, lo que resulta en código enredado y código disperso, producto de mezclar funcionalidad principal y secundaria para lograr las metas del sistema.
- Como una propuesta para enfrentar los problemas de la OO, la Orientación a Aspectos (OA) propone la separación completa de intereses: los intereses de la funcionalidad principal se implementan con un paradigma como el orientado a objetos y los intereses transversales se implementan con la ayuda de los aspectos.
- La OA hace posible un código más claro, y facilita el mantenimiento y la evolución de los sistemas; por el contrario, puede introducir errores y conductas no deseadas si el paradigma no se aplica correctamente.
- La adecuada combinación de objetos y aspectos incorporan al sistema lo mejor de ambos paradigmas, lo cual contribuye al éxito del sistema final. Algunos programadores adeptos a los paradigmas tradicionales, encuentran difícil ingresar al mundo de los aspectos, pero la experiencia adquirida en sus primeros proyectos disminuye la posterior curva de aprendizaje.
- La orientación a aspectos es un paradigma joven aún en comparación con la orientación a objetos, pero en corto tiempo ha sido blanco de propuestas académicas para buscar alternativas y mantener la separación de intereses durante todas las etapas del ciclo de vida. La fase de diseño ha capturado especial interés para la representación de aspectos, pues se ha identificado que en muchas ocasiones el programador se ve obligado a rediseñar el sistema para incorporar los aspectos, con lo que se rompe la consistencia del sistema global y se añade confusión a las responsabilidades del diseñador y el analista.
- A pesar de la diversidad de propuestas existentes para mantener la separación de intereses en todas las etapas de desarrollo, aún falta un consenso definitivo que proclame la aceptación universal de metodologías, principios y técnicas orientadas a aspectos. Varios autores proponen ya criterios que ayudan a identificar las metodologías más apropiadas. Algunas de las características buscadas son: facilidad, generalidad, independencia y completitud.
- La reutilización es una práctica que permite disminuir el esfuerzo en el proceso de desarrollo de *software*. Se logra al utilizar un mismo elemento en diferentes contextos. Paradigmas como la OO y la OA hacen posible la reutilización de elementos a través de clases, herencia y aspectos. La reutilización formal o planificada implica un programa administrativo apoyado por políticas bien definidas.
- Algunas tendencias en las prácticas de reutilización son: la ingeniería de *software* basada en componentes (CBSE), la ingeniería de línea de productos (PLE), y el desarrollo basado en componentes comerciales (COTS).

REFERENCIAS

Agile Alliance. (31 de enero de 2018). Manifiesto por el Desarrollo Ágil de Software. Obtenido de http://agilemanifesto.org/iso/es/

Algorry, M. (2005). Programación Orientada a Aspectos. USERS Code, 1(3), 60-64.

Anwer, F., Aftab, S., Muhammad Shh, S., & Waheed, U. (2017). Comparative Analysis of Two Popular Agile Process Models: Extreme Programming and Scrum. International Journal of Computer Science and Telecommunications, 8(2), 1-7.

Appleby, D., & Vandekopple, J. J. (1998). Lenguajes de programación: Paradigma y práctica. Ciudad de México: McGrawHill.

Baird, S. (2002). Teach yourself Extreme Programming in 24 hours. Estados Unidos: Sams Publishing.

Balijepally, V., Mahapatra, R., Nerur, S., & Kenneth, P. H. (2009). Are Two Heads Better than One for Software Development? The Productivity Paradox of Pair Programming. MIS Quarterly, 33(1), 91-118.

Beck, K., & Andres, C. (2004). eXtreme Programming explained. Embrace Change. Estados Unidos: Addison Wesley.

Blair, G., Blair, L., Rashid, A., Moreira, A., Araujo, J., & Chitchyan, R. (2004). Engineering Aspect Oriented Systems. En R. Filman, T. Elrad, S. Clarke, & M. Aksit, Aspect-Oriented Software Development (1 ed., págs. 379-406). Adisson-Wesley Professional.

Booch, G. (2000). Análisis y diseño orientado a objetos con aplicaciones. Ciudad de México: Pearson/Addison Wesley Longman.

Castillo, L. (2018). Resultados preliminares más significativos tras cuatro años de aplicación de la metodología SCRUM en las prácticas de laboratorio. ReVisión. Revista de Investigación en Docencia Universitaria de la Informática, 11(1), 53-64.

Castro, Y., Rivera, J., Fernández, J., & Acevedo, E. (2017). Construcción de un repositorio de activos de software para el desarrollo ágil de aplicaciones aplicando un método para el reuso. Lámpsakos, 1(17), 69-76.

Clemente, P., Hernández, J., Herrero, J., Murillo, J., & Sánchez, F. (2004). Aspect-Orientation in the Software Lifecycle: Fact and Fiction. En R. Filman, T. Elrad, S. Clarke, & M. Aksit, Aspect-Oriented Software Development (págs. 425-458). Addison-Wesley.

Coman, I. D., Robillard, P. N., Silliti, A., & Succi, G. (2014). Cooperation, collaboration and pair-programming: Field studies on backup behavior. The Journal of Systems and Software, 124-134.

da Silva Estácio, B. J., & Prikladnicki, R. (2015). Distributed Pair Programming: A Systematic Literature Review. Information and Software Technology, 1-10.

Dawande, M., Johar, M., Kumar, S., & Mookerjee, V. S. (2008). A Comparison of Pair Versus Solo Programming Under Different Objectives: An Analytical Approach. Information Systems Research, 19(1), 71-92.

Fairley, R. (1994). Ingeniería de Software. McGrawHill.

Filman, R., & Friedman, D. (2000). Aspect-Oriented Programming is Quantification and Obliviousness. Estados Unidos: RIACS.

Fowler, M. (2004). UML distilled. Estados Unidos: Addison Wesley.

Isoda, S. (1995). Experiences of a software reuse project. Journal of Systems and Software.

Jacobson, I., Booch, G., & Rumbaugh, J. (2005). El Proceso Unificado de Desarrollo de Software. Pearson Addison Wesley.

Joyanes Aguilar, L. (2012). Fundamentos de programación, algoritmos, estructuras de datos y objetos. McGrawHill.

Juristo, N., & M. Moreno, A. (2001). Basics of Software Engineering Experimentation. Springer.

Kendall, K., & Kendall, J. (2013). Systems Analysis and Design. Estados Unidos: Prentice Hall.

Kicillof, N. (2008). Obtenido de Programación Orientada a Aspectos: http://www.willydev.net/descargas/prev/AOP.PDF

Laddad, R. (2003). AspectJ in action Practical Aspect-Oriented Programming. Estados Unidos: Manning Publications Co.

Laddad, R. (Nov.-Dic. de 2003). Aspect-Oriented Programming will improve quality. IEEE Software, 20(6), 90-91.

Larman, C. (2004). Agile & Iterative Development - A Manager's Guide. Addison Wesley.

Laudon, J., & Laudon, K. (2013). Sistemas de información gerencial. Ciudad de México: Pearson.

Lawrence, P. (2013). Software Engineering Theory and Practice. Pearson.

Lazzarini Lemos, O. A., Cutigi Ferrari, F., Fagundes Silveira, F., & Garcia, A. (2012). Development of Auxiliary Functions: Should You Be Agile? - An Empirical Assessment of Pair Programming and Test-First Programming. Proceedings of the 34th International Conference on Software Engineering (pp. 529-539). IEEE Press.

Maquen, G., Chayan, A., & Reyes, L. (2017). Técnicas para el uso de la metodología Extreme Programming en el desarrollo de software. Hacedor Revista Científica, 1(1), 1-11.

Mili, H., Mili, A., Yacoub, S., & Addy, E. (2002). Reuse Based Software Engineering. Wiley Inter-Science.

Mohd Zin, A., Idris, S., & Kuman Subramaniam, N. (2006). Implementing Virtual Pair Programming in E-Learning Environment. Journal of Information Systems Education, 17(2), 113-117.

Pinciroli, F., & Zeligueta, L. (2017). Modelado de negocios orientado a aspectos con AOP4ST. XIX Workshop de Investigadores en Ciencias de la Computación (págs. 591-595). Buenos Aires, Argentina: RedUNCI.

Plonka, L., Sharp, H., van der Linden, J., & Dittrich, Y. (2015). Knowledge transfer in pair programming: An in-depth analysis. Int. J. Human-Computer Studies, 66-78.

Prabu, P., & Duraisami, S. (2015). Impact of Pair Programming for Effective Software Development Process. International Journal of Applied Engineering Research, 10(8), 18969-18986.

Pressman, R., & Maxim, B. R. (2014). Software Engineering: A Practitioner's approach (6 ed.). Estados Unidos: McGrawHill.

Prieto-Díaz, R. (Mayo de 1993). Status Report: Software Reusability. IEEE Software, 10(3).

Prywes, N. S., Pnueli, A., & Shastry, S. (1979). Use of a Nonprocedural Specification Language and Associated Program Generator in Software Development. ACM Transactions on Programming Languages and Systems.

Quiroga, D. (2004). Orientación a Objetos. Revista Users.Code, 1(1), págs. 58-62.

Roque Hernández, R. V. (2011). Tesis doctoral: "Separación multidimensional de aspectos para la especificación y reutilización de requisitos en Lenguaje Z". Vigo, Pontevedra, España.

Roque Hernández, R. V., Herrera Izaguirre, J. A., López, M. A., & Bravo Herrera, D. (2017). Comparación de la programación individual y por pares en los cursos universitarios. Temas de Ciencia y Tecnología, 11-22.

Roque Hernández, R. V., Herrera Izaguirre, J., López Mendoza, A., & Salinas Escandón, J. M. (2017). A Practical Approach to the Agile Development of Mobile Apps in the Classroom. Innovación Educativa, 97-114.

Scott, M. L. (2015). Programming Language Pragmatics. Estados Unidos: Morgan Kaufmann Pub.

Schwartz, J. T., Dewar, R. B., Schonberg, E., & Dubinsky, E. (1986). Programming with sets; an introduction to SETL. Springer-Verlag.

Sebesta, R. (2012). Concepts of programming languages. Estados Unidos: Pearson.

Sims, C., & Johnson, H. (2011). Elements of Scrum. Foster City, California: Dymaxicom.

Sims, C., & Johnson, H. L. (2012). SCRUM: Abreathtakinly Brief and Agile Introduction. Lexington, KY, EU: Dymaxicon.

Sommerville, I. (2015). Software Engineering (10 ed.). Estados Unidos: Pearson.
Sutherland, J., & Schwaber, K. (Julio de 2017). Scrum Guide. Obtenido de scruminc.com: https://www.scrum.org/resources/scrum-guide
Sutherland, Jeff. (2 de Abril de 2017). The Scrum Papers: Nut, Bolts, and Origins of an Agile Framework. Obtenido de scruminc.com: https://www.scruminc.com/scrumpapers.pdf
Sutton, S. J., & Rouvellou, I. (2004). Concern Modeling for Aspect-Oriented Software Development. En R. Filman, T. Elrad, S. Clarke, & M. Aksit, Aspect-Oriented Software Development (págs. 479-506). Addison Wesley.
Swamidurai, R., & Umpress, D. A. (2012). Collaborative-Adversarial Pair Programming. International Scholarly Research Network Software Engineering.
Tarr, P., & Sutton, J. S. (1999). N Degrees of Separation: Multi-Dimensional Separation of Concerns. The 21st. Conference on Software Engineering. Los Angeles, California.
Wells, D. (28 de enero de 2018). Extreme Programming: A gentle introduction. Recuperado el 28 de noviembre de 2018, de http://www.extremeprogramming.org/
Yang, Y.-F., Lee, C.-I., & Chang, C.-K. (2016). Learning motivation and retention effects of pair programming in data structures courses. Education for Information, 1(32), 249-267.
Zacharis, N. (2011). Measuring the Effects of Virtual Pair Programming in an Introductory Programming Java Course. IEEE Transactions on Education, 168-170.
Zave, P. (1991). An insider's evaluation of PAISLey. IEEE Transactions on Software Engineering.
Zhong, B., Wang, Q., Chen, J., & Li, Y. (2017). Investigating the Period of Switching Roles in Pair Programming in a Primary School. Educational Technology & Society, 20(3), 220-233.

Temas selectos de desarrollo de software, de Ramón Ventura Roque Hernández, publicado por la Universidad Autónoma de Tamaulipas y Colofón, se terminó de imprimir en marzo de 2020 en los talleres de Ultradigital Press S.A. de C.V. Centeno 195, Col. Valle del Sur, C.P. 09819, Ciudad de México. El tiraje consta de 300 ejemplares impresos de forma digital en papel Cultural de 75 gramos. El cuidado editorial estuvo a cargo del Consejo de Publicaciones UAT.