CLASIFICACIÓN DE LOS PATRONES DE DISEÑO IDÓNEOS EN PROGRAMACIÓN ANDROID

Osuna Tirado Jorge Luis[1], Ibarra Astorga José Alejandro[1], Lepe Mendoza José Carlos[1], Reyes Ramírez Ricardo Ulises[1], Álvaro Peraza Garzón[1]

[1]Facultad de Informática Mazatlán, Universidad Autónoma de Sinaloa, México.

Resumen

La eficiencia en programación es muy importante. Un programa con código eficiente dura menos tiempo en compilarse y realiza acciones con mayor rapidez. En la programación enfocada al sistema Android, existe una gran cantidad de patrones de diseño para evitar tener código innecesario (código espagueti) que dificulte la comprensión al usuario o al mismo desarrollador. Junior java desarrolladores son personas que están iniciando en la programación que no conocen patrones de diseño para facilitar la forma en la que escriben código. Se analizarán diversos patrones para comprobar su eficiencia y se clasificaron de acuerdo al enfoque que tendrá el programa.

Palabras clave: Patrones de diseño, código espagueti.

INTRODUCCIÓN

Actualmente en el entorno del desarrollo de software se ha diversificado en múltiples disciplinas como lo es la programación de aplicaciones para plataformas móviles como lo es Android que presenta un reto a desarrollador al momento de diseñar e implementar ciertas tecnologías y reglas como los patrones de diseño.

Los patrones de diseño son simples soluciones para los problemas comunes que se presentan al hacer o al desarrollar un proyecto de software, éstos nos ayudan guiándonos con consejos de cómo aplicarlos y permitiendo re-utilizarlos.

En 1979 el arquitecto Christopher Alexander aportó al mundo de la arquitectura el libro The Timeless Way of Building; en él proponía el aprendizaje y uso de una serie de patrones para la construcción de edificios de una mayor calidad, en la que esa mayor calidad se refería a la arquitectura antigua y la menor calidad correspondía a la arquitectura moderna, que el romper con la arquitectura antigua había perdido esa conexión con lo que las personas consideraban que era calidad [1].

En palabras de este autor, Christopher Alexander en su libro A Pattern Language nos dice, "cada patrón describe un problema que ocurre infinidad de veces en nuestro entorno, así como la solución al mismo, de tal modo que podemos utilizar esta solución un millón de veces más adelante sin tener que volver a pensarla otra vez." [2].

En 1987, Ward Cunningham y Kent Beck, hacen la observación que los nuevos programadores que comenzaban en el paradigma de programación orientada a objetos, se cuestionaban cómo se podría tener segmentos de código reutilizables y luego de alguna manera, poderlos implementar a las ideas de herencia y polimorfismo. Leyendo a Alexander se dieron cuenta del paralelo que existía entre la arquitectura y el desarrollo de software, de modo que usaron varias ideas de Alexander para desarrollar

cinco patrones de interacción hombre-ordenador y publicaron un artículo titulado Using Pattern Languages for OO Programs [3].

No obstante, en el año de 1994 fue cuando los patrones de diseño tuvieron un gran éxito en el mundo de la informática a partir de la publicación del libro Design Patterns: Elements of Reusable Object Oriented Software escrito por el grupo Gang of Four (GoF) compuesto por Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides, en el que se recogían 23 patrones de diseño comunes. Si bien ellos no son los que lo inventaron, no fue hasta luego de la publicación de ese libro que empezó a difundirse la idea de patrones de diseño en el desarrollo de software.

Erich Gamma at al. definen que “un patrón de diseño es una abstracción de una solución en un nivel alto. Los patrones solucionan problemas que existen en muchos niveles de abstracción” y los clasificaron en 3 grandes categorías basadas en su propósito: creacionales, estructurales y de comportamiento [4].

Se ha detectado que los desarrolladores de software recién egresado de las carreras afines a estas, carecen de buenas prácticas en el diseño e implementación o bien de un estándar del mismo, Carlos José Villagrá Arnedo at al. hace referencia al problema como “cuando se les exige crear sistemas completos desde cero, su código muestra problemas de base como código espagueti: software con mala estructuración, repeticiones innecesarias, deficiente paso de parámetros, escasa comprensión del paradigma orientado a objetos, etc” [5]. El desarrollo de software no estandarizado se considera de mala calidad dado a que puede representar riesgos. Unas de las causas que determina y afecta la calidad del software son la implementación de malas prácticas que aparecen desde el inicio del desarrollo.

Sin embargo, estas pueden ser predichas y controladas. El no implementar un estándar con factores de calidad como legibilidad, desempeño y la escalabilidad puede derivar en muchos problemas.

Un problema principal del software mal diseñado son los costos que se derivan después de su implementación. Muchas veces estos costos no son tomados en cuenta.

Las principales causas que conllevan a una mala calidad de software y que serán recurrentes año tras año, son:

1. Falta de dominio del lenguaje. En la mayoría de los proyectos los desarrolladores en un principio no son expertos en los conceptos y temas propios del lenguaje, para el cual se está desarrollando el software.

   Con el tiempo ellos logran conocer mucho sobre el lenguaje y se llegan a convertir en unos verdaderos expertos. Sin embargo, mucho de este desconocimiento al inicio se traduce en un buen número de defectos introducidos al sistema por reglas y requerimientos funcionales malentendidos.

2. No implementar ingeniería de Software. La inexperiencia que poseen los recién egresados de las carreras de sistemas o carreras afines al desarrollo de software, se suele arrastrar malas prácticas de las cuales es difícil salir, si esto lo llevamos a la programación móvil es un problema mayor dado que al tener unos recursos limitado de hardware se debe de considerar la estabilidad de todo el sistema.

   Ahora bien, en el momento de agregar nuevas utilidades a la aplicación si el código está ilegible dificulta el mantenimiento del mismo, teniendo en consideración que se está perdiendo tiempo en tratar de entender qué es lo que hace el código. Esto llega a ser la fuente principal de los defectos no funcionales que causan interrupciones dañinas, corrupción de datos y fallas de seguridad durante la operación.

3. Calendarios pocos realistas. Cuando los desarrolladores se ven obligados a cumplir con una fecha de entrega del producto en este caso el software lo primero que se sacrifica son las buenas prácticas de desarrollo de software aumentando la presión haciendo la prioridad la entrega que la calidad.

Antonio Leiva nos describe la necesidad de usar los patrones de diseño como “buscar siempre una nueva solución a los mismos problemas reduce tu eficacia como desarrollador, porque estás perdiendo mucho tiempo en el proceso. No hay que olvidar que el desarrollo de software también es una ingeniería, y que por tanto en muchas ocasiones habrá reglas comunes para solucionar problemas comunes.” [6].

Los costos operativos que implica el diseñar, analizar y desarrollo de sistemas de software suelen ser de un costo alto, por esa razón se necesita investigar el desarrollo de aplicaciones de software, haciendo uso de técnicas o reglas que posibiliten la reutilización de estructuras de software y con ello buscar el rendimiento eficiente de las aplicaciones.

Los patrones de diseño siguen una serie de reglas que ya marca un estándar que se ha implementado a lo largo de los años por parte de los programadores, evitando realizar código innecesario en el desarrollo de aplicaciones. Los junior java desarrolladores son personas que están iniciando en la programación Android y no conocen los patrones de diseño, causando que sus programas puedan carecer de estructura o sea software de mala calidad.

Cuando se toman en cuenta los patrones de diseño de software, se intenta identificar los patrones repetitivos o bien ese proceso donde se puede simplificar. Idealmente, podría usarse un conjunto de patrones de diseño de diseño para “generar” una aplicación.

La importancia de realizar esta investigación radica en el desarrollo de programas para el sistema Android por parte de los junior java desarrolladores, dado a su inexperiencia en el campo, causando que las aplicaciones que realicen tengan código redundante, poco o nada leíble e ineficiente en el uso de recursos donde se ejecutarán siendo estos escasos en su mayoría. Tener un catálogo de patrones de diseño basados en la eficiencia es una herramienta necesaria para los nuevos desarrolladores al momento de realizar un programa en Android.

Teniendo esta información en cuenta se plantea como objetivo clasificar los distintos patrones de diseño idóneos para la reducción de código espagueti en el desarrollo de aplicaciones móviles en la plataforma Android.

METODOLOGÍA

Las técnicas de investigación utilizadas durante el desarrollo de este trabajo han sido de tipo documental, debido a que se ha investigado y complementado por medio de la revisión bibliográfica de documentos como lo son: tesis, tesinas, artículos y libros acerca del tema.

Para poder hacer esto primero se identificó dónde y en qué caso las malas prácticas son implementadas, para después hacer recolección de los patrones que resuelven estos tipos de problemas, ya teniendo el concentrado de patrones de diseño se realizaron pruebas de rendimiento a los fragmentos de código donde se utilizan patrones de diseño y así poder descartar los no óptimos, haciendo esto se pudo realizar un catálogo de aquellos patrones que resultan idóneos para la programación de plataforma móvil Android.

En cuanto la forma de medir la eficiencia cabe mencionar que se eligió el lenguaje java, que es unos de los lenguajes utilizados para hacer código nativo de la plataforma, la cual nos brinda una potente

herramienta en sus librerías llamado JUnit que ayuda a medir los tiempos de ejecución y peticiones del programa.

RESULTADOS

Los patrones de diseño se clasifican de acuerdo al tipo de enfoque necesario. En la tabla 1 se presentan cuales se analizaron de tipo creacional y estructural. Cabe señalar que también existen patrones de comportamiento, pero no fue posible analizarlos.

Tabla 1. Patrones clasificados de acuerdo a su tipo de enfoque.

	Patrones creacionales	Patrones estructurales
Patrones analizados	Factory Method	Adapter
	Abstract Factory	Bridge
	Builder	Proxy
	Singleton	

Los patrones idóneos para la creación de aplicaciones móviles según su clasificación son los siguientes:

Patrones creacionales

Son los que facilitan la tarea de creación de nuevos objetos, de tal forma que el proceso de creación pueda ser desacoplado de la implementación del resto del sistema [6]. En este grupo están los patrones que sirven de guía para construir objetos. Además, independizan el ¿Qué?, ¿Quién?, ¿Cómo? y ¿Cuándo? en la creación del objeto.

Factory Method (patrón de diseño): forma una clase constructora común en la que se crearán todos los objetos de un determinado tipo.

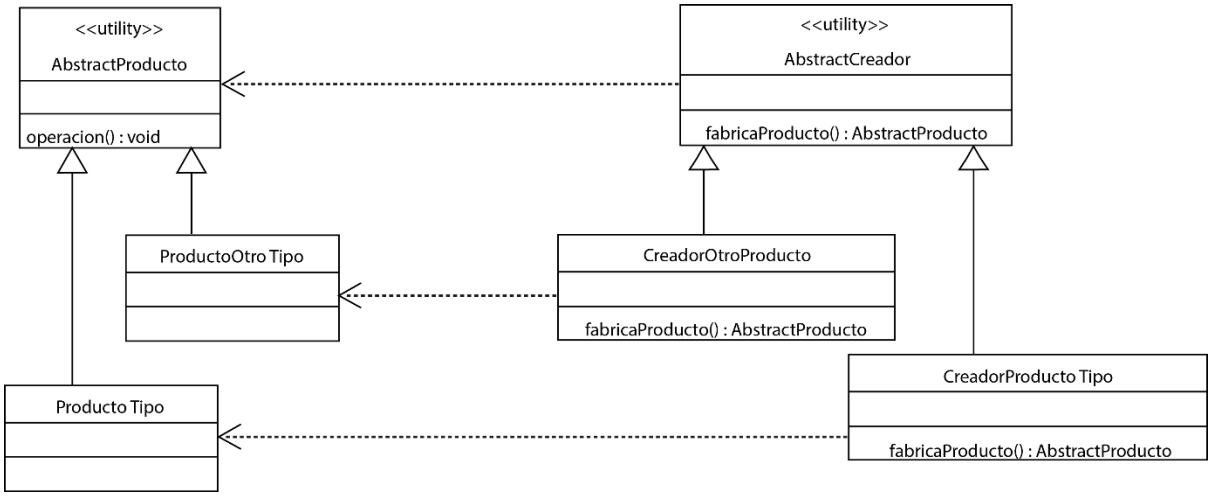

Figura 1. Esquema hecho en UML donde se describe el comportamiento de patrón [4].

En un "marco de referencia" para aplicaciones que pueden presentar múltiples documentos al usuario, no se sabe, a priori, los tipos de documentos con los que va a trabajar cada aplicación concreta. El "marco de referencia" debe, por tanto, instanciar clases, pero sólo tiene conocimiento de las clases

abstractas, las cuales no pueden ser instanciadas. La solución está en hacer que CrearDocumento() sea un Método de Fábrica que se encargue de la "fabricación" del objeto oportuno en cada situación (es una operación abstracta en la clase abstracta, pero concreta y distinta en las subclases correspondientes a cada tipo de aplicación.

Patrón de Abstract Factory (Fábrica Abstracta): hace abstracto el tipo de familia definido con la que se va a trabajar utilizando objetos de varias familias sin que éstas se mezclen.

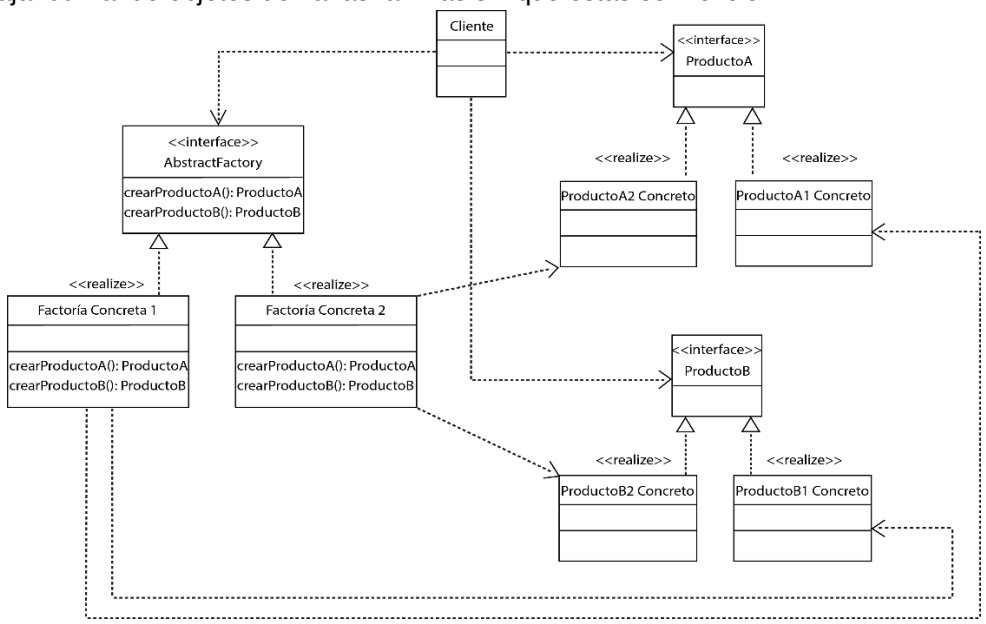

Figura 2. Esquema hecho en UML donde se describe el comportamiento de patrón [4].

Es cierto que las Factorías se encargan de generar una jerarquía de clases, pero su función fundamental es encapsular una jerarquía de objetos y reducir el conjunto de conceptos con los que trabajamos.

Facilita el intercambio de familias de objetos. Al crear una familia completa de objetos con una factoría abstracta, es fácil cambiar toda la familia de una vez simplemente cambiando la factoría concreta.

Promueve la consistencia entre productos. El uso de la factoría abstracta permite forzar a utilizar un conjunto de objetos de una misma familia.

Builder (Constructor): Es usado para permitir la creación de una variedad de objetos complejos desde un objeto fuente.

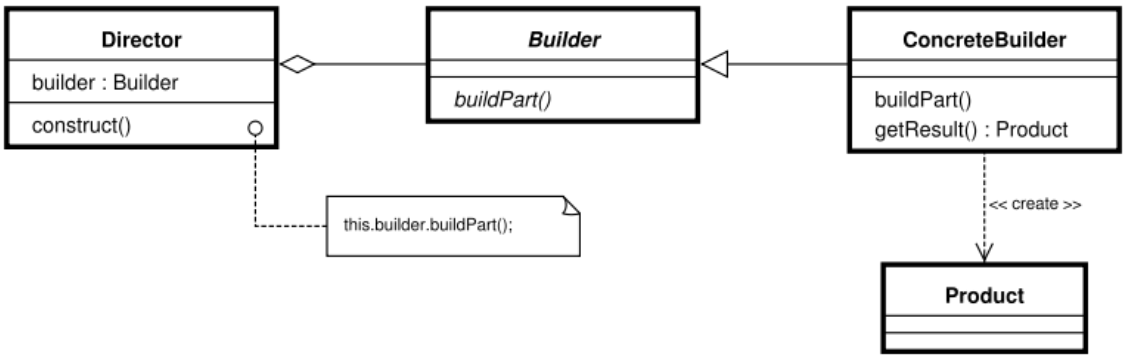

Figura 3. Esquema hecho en UML donde se describe el comportamiento de patrón [4].

Este patrón Builder se utiliza en situaciones en las que debe construirse un objeto repetidas veces o cuando este objeto tiene gran cantidad de atributos y objetos asociados, y en donde usar constructores para crear el objeto no es una solución cómoda.

Se trata de un patrón de diseño bastante útil también en la ejecución de test (unit test por ejemplo) en donde debemos crear el objeto con atributos válidos o por defecto. Normalmente resuelve el problema sobre decidir qué constructor utilizar. A menudo las clases tienen muchos constructores y es muy difícil mantenerlos. Es común ver constructores múltiples con distintas combinaciones de parámetros.

Singleton (instancia única): Es un patrón de diseño que permite restringir la creación de objetos pertenecientes a una clase o el valor de un tipo a un único objeto.

Su intención consiste en garantizar que una clase sólo tenga una instancia y proporcionar un punto de acceso global a ella por ejemplo al consumir una API con una instancia única de un objeto que contenga la misma se hacen menos llamadas al servidor donde se aloja por lo que esto ayuda a ahorrar datos de transferencia y el lag (tiempo que tarda el servidor en responder a la petición) que esto produce.

El patrón singleton se implementa creando en nuestra clase un método que crea una instancia del objeto solo si todavía no existe alguna. Para asegurar que la clase no puede ser instanciada nuevamente se regula el alcance del constructor (con modificadores de acceso como protegido o privado).

La instrumentación del patrón puede ser delicada en programas con múltiples hilos de ejecución. Si dos hilos de ejecución intentan crear la instancia al mismo tiempo y esta no existe todavía, sólo uno de ellos debe lograr crear el objeto. La solución clásica para este problema es utilizar exclusión mutua en el método de creación de la clase que implementa el patrón.

Patrones estructurales

Según Francisco José García Peñalvo “un patrón estructural de clases utiliza la herencia para componer interfaces o implementaciones, por ejemplo, el patrón Adapter. Un patrón estructural de objetos describe la forma en que se componen objetos para obtener nueva funcionalidad, además se añade la flexibilidad de cambiar la composición en tiempo de ejecución” [7]. En este grupo encontramos los patrones que organizan las clases y objetos de características similares y así formar estructuras más grandes.

Adapter: permite que una clase haga uso de una interfaz a la cual no tiene acceso, adaptando la primera especialmente para que sea convertida en otra parecida.

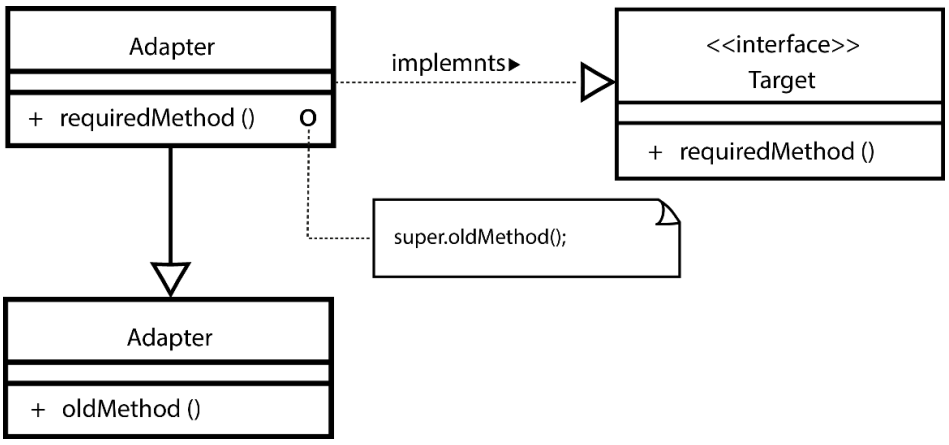

Figura 4. Esquema hecho en UML donde se describe el comportamiento de patrón [4].

Se busca determinar dinámicamente qué métodos de otros objetos llama un objeto.

Bridge: desacopla una abstracción de su implementación permitiendo modificarla a gusto sin que la primera cambie su forma original.

Tenemos la necesidad de que la implementación de una abstracción sea modificada en tiempo de ejecución o nuestro sistema requiere que la funcionalidad (parcial o total) de nuestra abstracción esté desacoplada de la implementación para poder modificar tanto una como otra sin que ello obligue a la cambiar las demás clases.

Se aplica cuando queremos evitar enlaces permanentes entre una abstracción y una implementación, tanto las abstracciones como las implementaciones deben ser extensibles por medio de subclases, queremos que los cambios en la implementación de una abstracción no afecten al cliente, necesitamos que la implementación de una característica sea compartida entre múltiples objetos, etc.

Proxy: El patrón proxy trata de proporcionar un objeto intermediario que represente o sustituya al objeto original con motivo de controlar el acceso y otras características del mismo.

Para explicar la motivación del uso de este patrón veamos un escenario donde su aplicación sería la solución más adecuada al problema planteado. Consideremos un editor que puede incluir objetos gráficos dentro de un documento. Se requiere que la apertura de un documento sea rápida, mientras que la creación de algunos objetos (imágenes de gran tamaño) es costosa. En este caso no es necesario crear todos los objetos con imágenes nada más abrir el documento porque no todos los objetos son visibles. Interesa por tanto retrasar el coste de crear e inicializar un objeto hasta que es realmente necesario (por ejemplo, no abrir las imágenes de un documento hasta que no son visibles). La solución que se plantea para ello es la de cargar las imágenes bajo demanda. Pero, ¿cómo cargar las imágenes bajo demanda sin complicar el resto del editor? La respuesta es utilizar un objeto proxy. Dicho objeto se comporta como una imagen normal y es el responsable de cargar la imagen bajo demanda.

CONCLUSIONES

En esta investigación faltó profundizar más en la clasificación de los patrones de comportamiento, pero en conclusión lo que concierne a los creacionales y estructurales se observa que la combinación de los patrones de diseño estos dan mayor legibilidad en la lectura y agilizan el proceso de compilado de las aplicaciones en Android por lo que el objetivo de hacer eficiente el código se cumple.

Otras de las observaciones es que al volverse el código más legible se reducen considerablemente las malas prácticas y el código espagueti.

Como se puede observar en la tabla 1 faltó analizar y catalogar a distintos patrones de diseño, esto se debió por falta de tiempo y organización, en un futuro se planea examinar los patrones faltantes del tipo creacional y estructural, además de analizar en su totalidad a los patrones de diseño de comportamiento.

REFERENCIAS

[1] Christopher Alexander. The Timeless Way of Building, New York: Oxford University Press, 1979.

[2] Christopher Alexander, Sara Ishikawa, Murray Silverstein. A Pattern Language con Max Jacobson, Ingrid Fiksdahl-King, Shlomo Angel. New York: Oxford University Press, 1977.

[3] Ward Cunningham, Kent Beck. Using Pattern Languages for Object-Oriented Programs: Portland, Oregon, OOPSLA-87(Object-Oriented Programming, Systems, Languages & Applications), 1987

[4] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software, Boston: Addison-Wesley Professional, 1994.

[5] Patricia Compañ Rosique, Carlos José Villagrá Arnedo, Francisco J. Gallego-Durán, Rosana Satorre Cuerda. Explicando el bajo nivel de programación de los estudiantes: ReVisión, Vol. 11, Nº. 1, 2018 (Ejemplar dedicado a: Investigación en Docencia Universitaria de la Informática)

[6] Antonio Leiva, Patrones de diseño de software, devExperto: https://devexperto.com/patrones-de-diseno-software.

[7] Francisco José García Peñalvo, Patrones De Alexander a la Tecnología de Objeto, Universidad de Salamanca.