UNIVERSIDAD DE COSTA RICA
SISTEMA DE ESTUDIOS DE POSGRADO

INCORPORACIÓN DE ELEMENTOS DE LA PRÁCTICA PROFESIONAL EN LA INDUSTRIA DE DESARROLLO DE SOFTWARE A UN CURSO DE DISEÑO DE SOFTWARE

Trabajo final de investigación aplicada sometido a la consideración de la Comisión del Programa de Estudios de Posgrado en Computación e Informática para optar al grado y título de Maestría Profesional en Computación en Informática

MAURICIO ALONSO CASTRO VALERIO

Ciudad Universitaria Rodrigo Facio, Costa Rica
2021

Dedicatoria

A Johnny, mi papá y Heissel, mi mamá, que me dieron educación y formación.

A Sandra, mi esposa y Paula, mi hija, que me apoyaron y motivaron para cerrar este ciclo.

Agradecimientos

Al profesor Dr. Gustavo López por su guía y consejos para elaborar este trabajo de la mejor forma posible.

A los profesores M.Sc. Alan Calderón Castro y Mag. Andrea Chacón Páez por sus amables aportes y colaboración en la revisión de este trabajo.

"Este trabajo final de investigación aplicada fue aceptado por la Comisión del Programa de Estudios de Posgrado en Computación e Informática de la Universidad de Costa Rica, como requisito parcial para optar al grado y título de la Maestría Profesional en Computación e Informática"

Dr. Marcelo Jenkins Coronas

Representante de la Decana

Sistema de Estudios de Posgrado

Dr. Gustavo López Herrera

Profesor Guía

M.Sc. Alan Calderón Castro

Lector

Mag. Andrea Chacón Páez

Lectora

Dra. Gabriela Marín Raventós

Directora del Programa de Posgrado en Computación e Informática

---

Mauricio Alonso Castro Valerio

Sustentante

Tabla de contenidos

Dedicatoria ii
Agradecimientos ii
Hoja de Aprobación iii
Tabla de contenidos iv
RESUMEN vi
Lista de tablas vii
Lista de figuras vii
CAPÍTULO 1 1
Introducción 1
Justificación 2
Contexto de la propuesta 2
Objetivos 3
Estructura 4
CAPÍTULO 2 6
Marco conceptual 6
CAPÍTULO 3 11
Estado del arte 11
CAPÍTULO 4 14
Metodología 14
CAPÍTULO 5 17
Ejercicios de diseño orientado a objetos 17
Crucigrama de conceptos de análisis y diseño orientado a objetos 19
Rompecabezas de diseño 21
Sistema de parqueo 23
Diseñar Twitter 28
CAPÍTULO 5 33
Analogías de los patrones de diseño y su evaluación 33
Analogías de los patrones de diseño 33
1. Abstract Factory 34
2. Builder 35
3. Factory Method 36
4. Prototype 37

5. Singleton 38
6. Adapter 39
7. Bridge 40
8. Composite 41
9. Decorator 42
10. Façade 43
11. Flyweight 44
12. Proxy 45
13. Chain of responsibility 46
14. Command 47
15. Interpreter 48
16. Iterator 49
17. Mediator 50
18. Memento 51
19. Observer 52
20. State 53
21. Strategy 54
22. Template Method 55
23. Visitor 56
Evaluación de las analogías 57
CAPÍTULO 7 63
Percepción del aprendizaje sobre patrones de diseño 63
Autopercepción del conocimiento de los estudiantes 63
Calificaciones del curso 71
CAPÍTULO 8 73
Uso y relevancia de patrones de diseño en la industria 73
CAPÍTULO 9 96
Conclusiones 96
Referencias 101

RESUMEN

Diseñar e implementar software orientado a objetos no es tarea fácil y lo es aún menos para estudiantes de ingeniería de software y profesionales con poca experiencia. La academia sienta las bases de lo que requiere un futuro ingeniero de software, pero además de conocer la teoría detrás del análisis, diseño y la programación orientada a objetos, es necesario acumular experiencia y práctica que ayuden a tomar buenas decisiones en esos contextos. Con el pasar de los años, los profesionales se dan cuenta que hay problemas comunes, pero que hay también soluciones listas y probadas, que pueden ser adaptadas y reutilizadas para resolver muchos de esos problemas.

La experiencia y conocimiento que se genera en la práctica profesional puede ser de mucha utilidad si se incorpora, de alguna forma, en los procesos de formación académica de los estudiantes de ingeniería de software. Esta investigación explora algunas propuestas de autores e ingenieros de software, que pueden ayudar en los procesos de enseñanza y aprendizaje de patrones de diseño a nivel universitario.

Como parte de uno de los dos ejes principales de esta investigación, se trabajó con estudiantes del curso CI-0136 Diseño de Software, que es parte de la carrera de Bachillerato en Computación con varios énfasis de la Escuela de Ciencias de la Computación e Informática en la Universidad de Costa Rica. Durante la ejecución de dicho curso, se trabajó en el desarrollo y aplicación de ejercicios de diseño orientado a objetos, así como en una lista de metáforas de los 23 patrones de diseño clásicos del *Gang of Four*. Los estudiantes dieron retroalimentación acerca de la claridad y utilidad de estas metáforas o analogías de los patrones de diseño con elementos del mundo real. Además, participaron en una encuesta ejecutada a principios, mediados y finales del semestre, con el fin de medir su autopercepción del avance en sus conocimientos.

En lo que respecta al segundo eje de este trabajo, se obtuvo la colaboración de profesionales en ingeniería de software. Se buscó obtener su valoración con respecto a las necesidades de conocimiento y experiencia práctica en diseño y patrones de diseño de software orientado a objetos en la industria. También, se quiso conocer la relevancia que, para ellos, como profesionales, tiene este tema, así como el uso real que ellos y sus equipos de trabajo le dan en la práctica laboral. Esto con el fin de obtener más insumos que puedan apoyar algunos de los procesos de enseñanza y aprendizaje de los estudiantes en esta área.

Lista de tablas

Tabla 1. Patrones de diseño del “Gang of Four” categorizados. Tomado de (Gamma et al., 1995). 7
Tabla 2. Respuestas del crucigrama de conceptos de análisis y diseño orientado a objetos. 19
Tabla 3. Lenguajes orientados a objetos más utilizados. 82

Lista de figuras

Figura 1. Crucigrama de conceptos de análisis y diseño orientado a objetos. 20
Figura 2. Rompecabezas de diseño. Tomado de (Freeman et al., 2004, pp. 78–80). 22
Figura 3. Solución al rompecabezas de diseño. Tomado de (Freeman et al., 2004, p. 94). 23
Figura 4. Clases principales del sistema de parqueo. Tomado de (McDowell, 2015, p. 313). 25
Figura 5. Clases ParkingLot y Level. Tomado de (McDowell, 2015, p. 314). 26
Figura 6. Clase ParkingSpot. Tomado de (McDowell, 2015, pp. 314–315). 27
Figura 7. Diseño de alto nivel de sistema similar a Twitter. Tomado de (Educative Inc, 2021). 31
Figura 8. Diseño de base de datos de sistema similar a Twitter. Tomado de (Educative Inc, 2021). 32
Figura 9. Analogía de abstract factory con fábrica de automóviles. 34
Figura 10. Analogía de builder con una “Cajita Feliz”. 35
Figura 11. Analogía de abstract factory con la creación de diferentes estilos de pizza. 36
Figura 12. Analogía de prototype con la mitosis. 37
Figura 13. Analogía de singleton con una única impresora de acceso global. 38
Figura 14. Analogía de adapter con un adaptador de corriente. 39
Figura 15. Analogía de bridge con televisores y controles remotos. 40
Figura 16. Analogía de composite con una expresión aritmética. 41
Figura 17. Analogía de decorator con capas de ropa. 42
Figura 18. Analogía de facade con servicio 911. 43
Figura 19. Analogía de flyweight con aplicación de música. 44
Figura 20. Analogía de proxy con tarjetas de débito y crédito. 45
Figura 21. Analogía de chain of responsibility con servicio de atención bancaria. 46
Figura 22. Analogía de command con control remoto programable. 47
Figura 23. Analogía de interpreter con la música y las partituras. 48
Figura 24. Analogía de iterator con un control remoto recorriendo canales. 49
Figura 25. Analogía de mediator con controlador aéreo. 50
Figura 26. Analogía de memento con transacciones ACID. 51
Figura 27. Analogía de observer con suscripción a un periódico. 52
Figura 28. Analogía de state con máquina expendedora. 53
Figura 29. Analogía de strategy con medios de transporte. 54
Figura 30. Analogía de template method con la construcción de una casa. 55
Figura 31. Analogía de visitor con servicio de taxi. 56
Figura 32. Evaluación de la claridad y utilidad de las analogías. 58

Figura 33. Evaluación de la claridad y utilidad de las analogías por categoría de patrón. 61
Figura 34. Autocalificación de los estudiantes con respecto a conocimiento y experiencia en patrones de diseño. 64
Figura 35. ¿Cree que implementar software usando patrones de diseño es difícil? 66
Figura 36. ¿Cree que implementar software usando patrones de diseño toma más o menos tiempo que sin hacerlo? 67
Figura 37. ¿Qué tan probable es que intente utilizar patrones de diseño en el software que usted desarrolla? 68
Figura 38. Familiaridad con algunos conceptos de análisis y diseño orientado a objetos. 69
Figura 39. Familiaridad con cada uno de los 23 patrones de diseño del GoF. 70
Figura 40. Comparación de autocalificaciones y calificaciones finales. 72
Figura 41. Años de experiencia laboral. 74
Figura 42. Importancia del conocimiento y experiencia en patrones de diseño. 75
Figura 43. Nivel de conocimientos y experiencia en patrones de diseño de los entrevistados. 76
Figura 44. Conocimiento en patrones de diseño de estudiantes e ingenieros. 77
Figura 45. Experiencia en patrones de diseño de estudiantes e ingenieros. 78
Figura 46. Conocimiento de otros catálogos de patrones de diseño además del GoF. 79
Figura 47. Patrones de diseño usados con mayor frecuencia. 80
Figura 48. Estilos y patrones arquitecturales más utilizados. 81
Figura 49. Dificultad de aplicar patrones de diseño en los lenguajes más utilizados. 83
Figura 50. ¿Cree que implementar software usando patrones de diseño es difícil? 84
Figura 51. ¿Cree que implementar software usando patrones de diseño toma más tiempo que sin hacerlo? 84
Figura 52. ¿Qué tan de acuerdo estas con estas afirmaciones? 87
Figura 53. Frecuencia de utilización de patrones de cada categoría. 88
Figura 54. Frecuencia con que reconocen patrones de diseño en código existente. 89
Figura 55. Frecuencia con que encuentran patrones de diseño mal implementados. 90
Figura 56. Frecuencia con que encuentran la necesidad de modificar o adaptar un patrón para sus propias necesidades. 91
Figura 57. Momento en el que se introducen patrones de diseño. 92
Figura 58. ¿Es hoy más menos importante la programación orientada a objetos? 93

UNIVERSIDAD DE COSTA RICA

SEP Sistema de Estudios de Posgrado

Autorización para digitalización y comunicación pública de Trabajos Finales de Graduación del Sistema de Estudios de Posgrado en el Repositorio Institucional de la Universidad de Costa Rica.

Yo, Mauricio Alonso Castro Valerio, con cédula de identidad 1-1016-0714, en mi condición de autor del TFG titulado INCORPORACIÓN DE ELEMENTOS DE LA PRÁCTICA PROFESIONAL EN LA INDUSTRIA DE DESARROLLO DE SOFTWARE A UN CURSO DE DISEÑO DE SOFTWARE

Autorizo a la Universidad de Costa Rica para digitalizar y hacer divulgación pública de forma gratuita de dicho TFG a través del Repositorio Institucional u otro medio electrónico, para ser puesto a disposición del público según lo que establezca el Sistema de Estudios de Posgrado. **SI** [X] **NO *** [ ]

***En caso de la negativa favor indicar el tiempo de restricción: ______________ año (s).**

Este Trabajo Final de Graduación será publicado en formato PDF, o en el formato que en el momento se establezca, de tal forma que el acceso al mismo sea libre, con el fin de permitir la consulta e impresión, pero no su modificación.

Manifiesto que mi Trabajo Final de Graduación fue debidamente subido al sistema digital Kerwá y su contenido corresponde al documento original que sirvió para la obtención de mi título, y que su información no infringe ni violenta ningún derecho a terceros. El TFG además cuenta con el visto bueno de mi Director (a) de Tesis o Tutor (a) y cumplió con lo establecido en la revisión del Formato por parte del Sistema de Estudios de Posgrado.

INFORMACIÓN DEL ESTUDIANTE:

Nombre Completo: Mauricio Alonso Castro Valerio.

Número de Carné: 960759 Número de cédula: 1-1016-0714.

Correo Electrónico: mauricio.castrovalerio@ucr.ac.cr / mauricio.castro@gmail.com.

Fecha: 13 de octubre de 2021. Número de teléfono: 8980-7241 / 2260-5365.

Nombre del Director (a) de Tesis o Tutor (a): Gustavo López Herrera.

FIRMA ESTUDIANTE

Nota: El presente documento constituye una declaración jurada, cuyos alcances aseguran a la Universidad, que su contenido sea tomado como cierto. Su importancia radica en que permite abreviar procedimientos administrativos, y al mismo tiempo genera una responsabilidad legal para que quien declare contrario a la verdad de lo que manifiesta, puede como consecuencia, enfrentar un proceso penal por delito de perjurio, tipificado en el artículo 318 de nuestro Código Penal. Lo anterior implica que el estudiante se vea forzado a realizar su mayor esfuerzo para que no sólo incluya información veraz en la Licencia de Publicación, sino que también realice diligentemente la gestión de subir el documento correcto en la plataforma digital Kerwá.

CAPÍTULO 1

Introducción

La importancia de patrones en la creación de sistemas complejos ha sido, desde hace mucho tiempo, reconocida y valorada en varias disciplinas de la ingeniería (Lartigue & Chapman, 2018). Christopher Alexander fue el primero en proponer la idea de usar un lenguaje de patrones para definir la arquitectura de edificaciones y ciudades completas. Las ideas de Alexander y otros que han trabajado sobre su base, están, desde hace tiempo, también arraigadas en la comunidad de desarrolladores de software orientado a objetos (Gamma, Helm, Johnson, & Vlissides, 1995).

Patrones de diseño son, en este contexto, abstracciones de alto nivel de soluciones específicas a tareas y problemas comunes en la ingeniería de software. Son recetas genéricas que los desarrolladores pueden adaptar a sus necesidades específicas. Sin embargo, existe debate y poca evidencia empírica, de qué tan útiles son en realidad estos patrones para desarrolladores experimentados y más aún, para desarrolladores novatos y estudiantes de ingeniería de software (Lartigue & Chapman, 2018).

Dada esta realidad, la pregunta de investigación que se ha planteado para este trabajo es: ¿Cómo se podrían incorporar elementos de la práctica profesional en la industria de desarrollo de software en los procesos de enseñanza y aprendizaje de patrones de diseño de software a nivel universitario?

A continuación, se detalla la justificación de su utilidad, beneficios y aportes. Seguido a esto, se describe el contexto en el que se desarrolló este estudio, su objetivo general y los específicos, para finalizar con una breve descripción de la estructura general del documento.

Justificación

Muchas universidades tienen cursos dedicados en su totalidad a patrones de diseño, tanto a nivel de pregrado como de posgrado (Lartigue & Chapman, 2018). Sin embargo, un problema frecuente de estos cursos es el de enseñar patrones de diseño en una forma mecánica y memorística, cubriendo catálogos de patrones de forma aislada y sin un contexto que amalgame el conocimiento (Lartigue & Chapman, 2018).

Por otra parte, diferentes estudios y trabajos muestran que hay formas más efectivas para enseñar y aprender patrones de diseño, que se enfocan en ejercicios y experiencias, así como en analogías con elementos familiares para los estudiantes (Lartigue & Chapman, 2018). Un estilo de aprendizaje que incluye elementos visuales, actividades más informales y contenido emocional, provoca una mayor actividad cerebral que incrementa la probabilidad de que nuestro cerebro se mantenga atento y que recuerde más información (McLaughlin, Pollice, & West, 2006). La importancia de un aprendizaje efectivo de patrones de diseño radica en que, el futuro campo laboral de muchos de los estudiantes, la industria, avala y afirma la relevancia que estos tienen en la práctica profesional y la importancia de que los estudiantes sean expuestos a ellos (Jalil & Noah, 2007).

En la Universidad de Costa Rica, particularmente en el curso de Diseño de Software, se han aplicado desde hace varios años los principios del aprendizaje activo y basado en problemas. En este trabajo de investigación se busca fortalecer dicho curso al incorporar algunos elementos de la práctica profesional en la industria de desarrollo de software.

Contexto de la propuesta

Esta propuesta se circunscribe al curso CI-0136 Diseño de Software impartido a nivel de bachillerato en la carrera de Bachillerato en Computación con varios énfasis de la Escuela de Ciencias de la Computación e Informática en la Universidad de Costa Rica.

El curso cubre temas generales de análisis y diseño de software orientado a objetos, como principios SOLID, abstracción, encapsulamiento y delegación, por mencionar algunos ("Diseño de Software | Escuela de Ciencias de la Computación e Informática", 2020). Se adentra también en áreas relacionadas a patrones arquitectónicos, pero principalmente en los 23 patrones de diseño descritos por (Gamma et al., 1995), el GoF (*Gang of Four)*, en su libro clásico en la materia. Estos temas son de gran importancia para las actividades de diseño de software, pero dominarlos es difícil para programadores experimentados y más aún para estudiantes, lo que convierte su enseñanza en un reto (Weiss, 2005).

Por otra parte, al referirse a elementos de la práctica profesional en la industria de desarrollo de software, se hace en relación con las experiencias vividas en una empresa de desarrollo de software a la medida y de servicios de *outsourcing* basada en Costa Rica y con filiales en diversos países.

Se trabajó en obtener insumos de la industria en forma de ejercicios y retroalimentación, que proporcionaron ingenieros de software con base en su experiencia profesional. Estos ejercicios, unos tomados de la literatura en la materia, así como los provenientes de la industria, fueron compilados con el fin de reutilizarlos en versiones futuras del curso. Los objetivos y la metodología de este estudio presentan más detalles de cómo se consiguió todo esto.

Objetivos

El objetivo general de esta investigación es: Incorporar elementos de la práctica profesional en la industria de desarrollo de software a un curso de diseño de software a nivel universitario.

Los objetivos específicos, que se desprenden del anterior objetivo general son:

1. Desarrollar y validar ejercicios y analogías que permitan a los estudiantes aprender sobre patrones de diseño incorporando problemas de diseño comunes en la práctica profesional.

2. Comparar la autopercepción del nivel de conocimiento de los estudiantes en patrones de diseño con la evaluación obtenida por medio de las pruebas y ejercicios realizados.
3. Obtener retroalimentación de la industria con respecto al uso de patrones de diseño y su relevancia real en la práctica profesional.

Estructura

En el siguiente capítulo, el segundo de este documento, se describe el marco conceptual de esta investigación, que permite comprender mejor el contenido en las siguientes secciones. Se presenta la definición de patrón de diseño, sus características, categorías y principios de diseño orientado a objetos que los sustentan. En el capítulo 3, se presentan diferentes tendencias en lo que a enseñanza y aprendizaje de patrones de diseño se refiere. Se expone lo que algunos autores y académicos están haciendo para facilitar estos procesos y, además, algunas opiniones de profesionales de la industria acerca de los beneficios que aporta el uso de patrones de diseño. El capítulo 4 ahonda en la metodología seguida para cumplir con cada uno de los objetivos planteados, comenzando con los ejercicios y analogías de los patrones de diseño.

En el capítulo 5 se describen cuatro ejercicios relacionados con diseño orientado a objetos. Se empieza con un enfoque académico y se sigue con uno más centrado en la industria y algunas ideas de lo que se requiere para afrontar entrevistas laborales y ejercicios que en ellas se suele solicitar desarrollar y resolver. En el siguiente capítulo, el sexto, se presentan las analogías o metáforas desarrolladas para cada uno de los 23 patrones de diseño del GoF, así como la evaluación acerca de su claridad, efectuada por los 16 estudiantes del curso de diseño.

El capítulo sétimo presenta los hallazgos en torno a la percepción de los estudiantes del curso acerca de su aprendizaje de patrones de diseño. Los estudiantes fueron encuestados para evaluar su conocimiento en diseño y patrones de diseño orientados a objetos en tres puntos diferentes del curso. Los resultados de esta encuesta y el análisis de su progreso a través de

estos diferentes momentos se presentan y analizan, junto con un apartado que compara su autoevaluación con las calificaciones finales del curso.

En el capítulo 8 se exploran las opiniones de un grupo de 35 profesionales en ingeniería de software que trabajan para una compañía de desarrollo de software a la medida con sede en Costa Rica y oficinas en varios países de Latinoamérica. Se da a conocer que patrones de diseño, principios, técnicas y lenguajes de programación utilizan con mayor frecuencia en su día a día, así como la relevancia que ellos creen tienen los patrones de diseño en las labores que realizan. En el capítulo noveno y final, se exponen las conclusiones de esta investigación.

CAPÍTULO 2

Marco conceptual

Christopher Alexander dijo que, "cada patrón de diseño describe un problema que ocurre una y otra vez en nuestro entorno y que luego describe el núcleo de la solución a ese problema, de forma que se puede utilizar esa solución más de un millón de veces sin hacerlo de la misma forma dos veces" (Gamma et al., 1995, pp. 22–23). Sin embargo, aunque los patrones de diseño han sido usados y probados por muchos desarrolladores, no es posible simplemente introducirlos en un sistema sin un análisis previo (Freeman, Robson, Bates, & Sierra, 2004). Saber cuándo un patrón de diseño es aplicable requiere conocimiento y experiencia, no sólo acerca de los patrones mismos, sino también acerca de fundamentos de análisis y diseño orientado a objetos (Freeman et al., 2004).

Una de las características más importantes de los patrones de diseño es que proporcionan un vocabulario compartido. Cada vez que los desarrolladores se comunican en términos de patrones de diseño están hablando no solo de sus nombres, sino también de las cualidades, características y restricciones que representan (Freeman et al., 2004). Parte importante de este vocabulario compartido son los nombres de los patrones mismos y una categorización como la que hizo el GoF de los patrones de diseño clásicos en tres grandes grupos, como se muestra en la Tabla 1 (Gamma et al., 1995).

Los 23 patrones de diseño definidos por el GoF no son todos los patrones de diseño que existen, sino más bien el primer catálogo de ellos que se creó y sobre el que muchos otros autores e ingenieros han continuado trabajando como base (Freeman et al., 2004; Huang & Yang, 2008). Existen catálogos generales y otros que apuntan a dominios específicos, como el de arquitectura de aplicaciones empresariales (Fowler, Rice, Foemmel, Mee, & Stafford, 2002) y el de computación en la nube (Fehling, Leymann, Retter, Schupeck, & Arbitter, 2014), por mencionar algunos.

Tabla 1. Patrones de diseño del "Gang of Four" categorizados. Tomado de (Gamma et al., 1995).

Categoría	*Creacionales*	*Estructurales*	*Comportamiento*
Patrones	• *Abstract factory* • *Builder* • *Factory method* • *Prototype* • *Singleton*	• *Adapter* • *Bridge* • *Composite* • *Decorator* • *Façade* • *Flyweight* • *Proxy*	• *Chain of responsibility* • *Command* • *Interpreter* • *Iterator* • *Mediator* • *Memento* • *Observer* • *State* • *Strategy* • *Template method* • *Visitor*

Para poder siquiera pensar en patrones de diseño hay que empezar por conocer aspectos básicos, pero fundamentales, del diseño y programación orientada a objetos. En primer lugar está la abstracción, que tiene que ver con encontrar comportamientos comunes en dos o más partes, extraer ese comportamiento a un lugar común y reutilizarlo donde sea necesario (McLaughlin et al., 2006). También está la herencia, que se da cuando una clase hereda comportamiento de una clase padre y puede hacer ajustes si lo requiere. Otro de los conceptos claves es el de polimorfismo, altamente relacionado con herencia, que se da cuando una clase puede ser usada en lugar de su clase padre. Por último está la encapsulación, que se da cuando se separa o esconde una parte del código del resto (McLaughlin et al., 2006). Otros principios y técnicas de diseño orientado a objetos que ayudan a reutilizar código se fundamentan sobre estos conceptos.

Las dos técnicas más comunes para reutilizar funcionalidad en sistemas orientados a objectos son herencia y composición (Gamma et al., 1995). La primera, como ya se dijo, permite definir la implementación de una clase en términos de otra, por medio de clases padre e hijas. La segunda, composición, permite crear nueva funcionalidad al ensamblar o componer objetos para obtener comportamientos más complejos (Gamma et al., 1995). Ambas tienen sus ventajas y desventajas, pero al ser la composición más dinámica y respetuosa de la encapsulación, se prefiere su uso para un mejor diseño orientado a objetos (Gamma et al., 1995). La delegación es un tipo de composición, en la que un objecto delega sus operaciones a otro y le da a la composición muchas de las ventajas de la herencia (Gamma et al., 1995).

Una tercera técnica que permite reusar funcionalidad es la de tipos parametrizados, también conocida en inglés como *generics* (Gamma et al., 1995). Esta técnica permite definir un tipo sin definir todos los otros tipos que el primero utiliza. Estos tipos no especificados son suministrados como parámetros en el momento de su uso. Por ejemplo, una clase que representa una lista puede ser parametrizada con respecto al tipo de elementos que contiene, como números enteros o hileras de caracteres (Gamma et al., 1995). A pesar de que estas son construcciones básicas utilizadas en el diseño orientado a objetos, no es extraño que los programadores principiantes las utilicen de forma errónea (Joshi & Joshi, 2016).

Existen algunos principios de diseño orientado a objetos que pueden ayudar a asegurarse de escribir código de la manera correcta (Joshi & Joshi, 2016). Los principios SOLID (*Single Responsibility Principle, Open/Closed Principle, Liskov Substitution Principle, Interface Segregation Principle y Dependency Injection Principl*e, por sus siglas en inglés) son una guía sencilla para tener en mente al escribir código (Joshi & Joshi, 2016). Estos principios son fundamentales para escribir código robusto, extensible y mantenible (Joshi & Joshi, 2016). No obstante, no es suficiente contar con listas de principios y patrones de diseño, es necesario algo que haga que todas las piezas de un sistema encajen en una organización estructurada de diseño y eso es lo que se conoce como arquitectura (McLaughlin et al., 2006).

Arquitectura es un concepto sobre cuya definición final no existe un acuerdo definitivo, dicen (Fowler et al., 2002). Estos autores mencionan que la definición se puede describir en dos partes. La primera tiene que ver con la descomposición de más alto nivel de las partes de un sistema y la segunda con las decisiones que son difíciles de cambiar en ese sistema. Por otra parte (Bass, Clements, & Kazman, 2012, p. 4), definen arquitectura como “el conjunto de estructuras requeridas para razonar acerca de un sistema y que comprende elementos de software, las relaciones entre ellos y sus propiedades”. Estos elementos de la arquitectura se pueden mezclar de diferentes maneras para resolver problemas particulares y algunas de estas composiciones han resultado ser útiles a través del tiempo y en diferentes dominios. Estas estrategias han sido documentadas y diseminadas en forma de patrones arquitectónicos (Bass et al., 2012).

Sin embargo, a pesar de estar en un contexto de más alto nivel que el de patrones de diseño, la definición de patrones arquitectónicos no varía de lo que dice Christopher Alexander: son soluciones comunes y efectivas para problemas recurrentes (Fowler et al., 2002). No importa la perspectiva desde la que se quiera abordar el tema de patrones de diseño, su aprendizaje y enseñanza son complejos y un reto tanto para estudiantes como para profesores (Tao, Liu, Mottok, Hackenberg, & Hagel, 2015).

Existen principios y teoría generales de educación que pueden ser aplicados a la enseñanza de la ingeniería de software, como, por ejemplo, el aprendizaje activo. Este es un enfoque en el que se propicia un ambiente en el que los estudiantes pueden adquirir la profundidad de conocimiento necesaria en áreas como la de patrones de diseño (Warren, 2005). La filosofía básica de este paradigma consiste en involucrar a los estudiantes en su aprendizaje, haciéndolos participar en actividades, en lugar de actuar como agentes pasivos de recepción de conocimiento (Warren, 2005). El aprendizaje activo involucra a los estudiantes proveyendo los estímulos necesarios para que usen sus procesos cognitivos de alto nivel y fomentando un entendimiento profundo de la materia (Warren, 2005).

Este trabajo no pretende ahondar en alguna técnica de enseñanza o aprendizaje específica, sino más bien tomar ideas de diferentes enfoques que diversos autores sugieren como beneficiosas

para estudiantes e ingenieros menos maduros. Por ejemplo, el libro *Head First Design Patterns* (Freeman et al., 2004), intenta abordar el problema que representa trabajar con ejemplos muy complejos para estudiantes con poca experiencia, por medio de la presentación de los patrones de diseño en contextos que les puedan resultar familiares (Dukovich & Janzen, 2009).

CAPÍTULO 3

Estado del arte

Los patrones de diseño de software fueron popularizados por (Gamma et al., 1995), el *Gang of Four*, con la publicación de su libro *Patrones de Diseño*, que es considerado la "biblia" de los diseñadores de software orientado a objetos (Huang & Yang, 2008). Sin embargo, pese a su popularidad y toda la información que contiene, no es considerado apropiado como libro de texto único para cursos en esta área. Este libro es muy abstracto para estudiantes sin experiencia práctica en el diseño y desarrollo de software (Huang & Yang, 2008). Cursos que se basan solamente en libros de este tipo, que sirven mejor como material de referencia, combinados con clases magistrales clásicas, generan, por lo regular, un nivel de conocimiento superficial y frágil (Huang & Yang, 2008).

Diversos estudios señalan que los estudiantes están menos propensos a aprender al solamente escuchar que cuando están activamente involucrados en actividades. Por esto, las clases tradicionales de ciencias de la computación y otras disciplinas STEM (*Science, Technology, Engineering and Mathematics*) están siendo reconsideradas (Dehbozorgi, Macneil, Maher, & Dorodchi, 2019). Por ejemplo, la pedagogía del aprendizaje activo está siendo adoptada por muchas facultades de ciencias de la computación en tiempos recientes, en reemplazo de las clases magistrales clásicas (Dehbozorgi et al., 2019). No obstante, no hay un solo tipo de aprendizaje activo, sino que existen diferentes enfoques, como TBL (*Team Based Learning*), aprendizaje cooperativo, aprendizaje colaborativo o PBL (*Problem Based Learning*) (Dehbozorgi et al., 2019).

Existen esfuerzos como el de (Huang & Yang, 2008), que exponen cómo adoptaron una estrategia de aprendizaje basada en PBL para su curso de patrones de diseño, guida por el concepto de "aprender haciendo", pero no son los únicos. En tiempos más recientes, (Tao et al., 2015) trabajaron en un proyecto de investigación en las universidades de Qingdao y Regensburg para aplicar la enseñanza JiTT (*Just-in-Time Teaching*) en su curso de patrones de diseño. La

enseñanza *just-in-time* se basa en un enfoque constructivista, en el que trabajan en la resolución de problemas en contextos de la vida real, sesiones de preguntas, discusiones, tareas y evaluaciones.

Otros que dan gran importancia al enfoque de aprender utilizando un contexto de la vida real son (Dukovich & Janzen, 2009), que crearon módulos multimedia para introducir a los estudiantes al mundo de los patrones de diseño, usando ejemplos no relacionados con las ciencias de la computación. En su trabajo, mencionan como los estudiantes manifestaron ser capaces de recordar cómo funcionan algunos patrones de diseño, no por medio de su definición, sino del ejemplo de la vida real con que lo aprendieron por analogía. Por otra parte, más allá de la academia, la industria y la comunidad de desarrollo de software, confirman la necesidad de una buena formación en patrones de diseño (Jalil & Noah, 2007).

Ingenieros de software en la industria aseguran que los patrones de diseño no solo proveen un vocabulario común para comunicar sus diseños, sino que también promueven la reutilización y hacen el software más mantenible y flexible (Jalil & Noah, 2007). Además, los ingenieros han observado como los patrones de diseño ayudan a que los profesionales menos experimentados produzcan mejores diseños (Jalil & Noah, 2007).

De forma contemporánea a la publicación del libro del *Gang of Four* (Gamma et al., 1995), ya algunos como (Beck et al., 1995) se preguntaban: “los patrones suenan muy promisorios, pero, ¿cómo se usan realmente en la industria y qué beneficios, si es que hay alguno, tienen en la práctica?” Estos autores documentaron sus experiencias con patrones de diseño en empresas como AT&T, Motorola, Siemens e IBM, así como los beneficios que encontraron en ellos. Entre estos, confirman que los patrones mejoran la comunicación en el equipo, motivan la reutilización de buenas prácticas y dan la oportunidad a ingenieros novatos de aprender de la experiencia de los más maduros.

Este trabajo propone abordar la enseñanza de patrones de diseño poniendo en práctica algunas de las técnicas sugeridas por investigadores y educadores para asegurar un aprendizaje efectivo

de los mismos. Además, pretende incorporar elementos de la industria que puedan enriquecer el proceso de formación de los estudiantes en esta área, al aportar ejemplos, ejercicios y perspectivas de la práctica profesional a la que muchos de estos alumnos estarán expuestos al terminar sus estudios.

CAPÍTULO 4

Metodología

La metodología para llevar a cabo este estudio gira en torno al cumplimiento de los objetivos definidos y está guiada por la metáfora de la "cebolla de la investigación", planteada por (Saunders & Tosey, 2013). Esta metáfora menciona que, es común, al inicio de una investigación, enfocarse en los datos que serán necesarios para responder una pregunta o solucionar un problema y cómo se obtendrán dichos datos. Sin embargo, las técnicas para obtener estos datos y los procedimientos para analizarlos son solo la decisión final en el diseño de la investigación, el centro de la "cebolla" (Saunders & Tosey, 2013). La temporalidad, estrategias, metodología y filosofía de investigación son, de adentro hacia afuera, las capas externas de la "cebolla". Son estas las que proveen el contexto y límites dentro de los que se deben seleccionar las técnicas para la recolección de los datos y los procedimientos para su análisis (Saunders & Tosey, 2013).

Con base en esta guía y empezando desde la capa más exterior de la "cebolla", se determinó seguir la Ciencia del Diseño como filosofía de investigación. En lo que respecta a la elección metodológica, se procedió con una mixta, en la que se utilizaron estrategias como las encuestas y un caso de estudio, que es el grupo de estudiantes del curso de diseño de software. Además, se realizaron actividades de investigación, recolección y aprovechamiento de los elementos de la industria que se consideraron pertinentes para este trabajo, como se desarrolla a continuación para cada uno de los objetivos establecidos.

Con respecto al **Objetivo 1**, relativo al desarrollo y validación de ejercicios y analogías que permitan a los estudiantes aprender sobre patrones de diseño incorporando problemas de diseño comunes en la práctica profesional se procedió en dos partes.

Primero, se tomaron ideas de las obras de (Freeman et al., 2004) y (McLaughlin et al., 2006). Ambos libros cuentan con un importante número de ejercicios, como crucigramas, ejercicios de asociar y de completar, que pueden realizarse en clase, como parte de tareas, exámenes o

proyectos (Freeman et al., 2004). Estos ejercicios, tomados de la literatura, fueron complementados con otros que se estén practicando en procesos de reclutamiento y entrenamiento de ingenieros de software en la industria. Además, se diseñaron un proyecto y un examen que permitieron poner en práctica el conocimiento adquirido de forma integral, utilizando varios patrones de diseño en una misma solución o sistema. Como parte de este objetivo, se hizo una compilación de los ejercicios desarrollados y aplicados para que puedan ser reutilizados en versiones futuras del curso.

Segundo, se desarrollaron metáforas de los patrones de diseño de forma similar a lo propuesto por (Shvets, 2021) , que en su sitio web, *refactoring.guru*, así como en su libro electrónico *Dive Into Design Patterns* (Shvets, 2019), presenta un catálogo de 22 de los patrones de diseño clásicos. Para cada patrón se exponen las partes que lo describen, como intención, motivación, estructura, código, así como la sección más importante para este trabajo: analogías de los patrones de diseño con elementos del mundo real. Para este trabajo se desarrollaron analogías o metáforas de los patrones de diseño con elementos del mundo real para apelar a la parte emocional del estudiante de la misma forma en la que se hace en algunos libros de texto (Freeman et al., 2004). Estas metáforas se expusieron a los estudiantes y se obtuvo su evaluación, por medio de una encuesta, para determinar en qué medida las percibieron como claras, explicativas y fáciles de recordar.

También, se ejecutó una encuesta en varios puntos del curso, con el fin de medir la autopercepción que los estudiantes tienen de su conocimiento en temas de análisis y diseño orientado a objetos y patrones de diseño. Esta encuesta está basada en la realizada por (Lartigue & Chapman, 2018) para obtener información demográfica de los estudiantes y su experiencia autorreportada en esta materia. La encuesta se aplicó al inicio, a la mitad y al final del curso, para medir el avance que los estudiantes percibían tener en cada momento. Los resultados finales del curso fueron comparados con la información autorreportada por los estudiantes, con el propósito de identificar coincidencias y divergencias en las valoraciones y así cumplir con el **Objetivo 2** de este trabajo.

Otro paso en las mediciones que se hicieron como parte de este estudio y que permitió completar el **Objetivo 3**, fue consultar la opinión de profesionales en ingeniería de software enmarcados en el contexto de esta investigación, con el fin de obtener puntos de vista de la industria acerca de la importancia del conocimiento en patrones de diseño. Para esto, se elaboró una encuesta que permitió conocer diferentes aspectos relacionados con su experiencia, en la práctica profesional, con respecto al uso y relevancia de los patrones de diseño.

Todos los productos generados a partir de las tareas asociadas a cada objetivo han sido compilados como parte de este este trabajo. Se generó un pequeño compendio con los ejercicios aplicados, de forma que puedan ser reutilizados en el futuro. Las metáforas y su evaluación por parte de los estudiantes fueron también analizadas. Por otra parte, los resultados de la encuesta del conocimiento autorreportado por los estudiantes son presentados y examinados, con el fin de identificar y ayudar a fortalecer áreas débiles para versiones futuras del curso. La encuesta aplicada a ingenieros de software con respecto al uso y relevancia de patrones de diseño provee insumos que fueron documentados, con la intención de que puedan servir para alinear algunos temas académicos con respecto a lo que se hace en la industria.

Finalmente, los conocimientos adquiridos en el proceso de investigación son utilizados para proponer mejoras didácticas a un curso de diseño de software que incorpore ejercicios más alineados con la realidad de la práctica profesional y para priorizar las temáticas más relevantes de patrones de diseño en el curso.

A partir del siguiente capítulo, comienza la exposición de los resultados obtenidos en esta investigación. Se presentarán en el mismo orden en que se plantearon los objetivos, que es el mismo en el que se ha desarrollado la presente sección, que describe las actividades que se requieren para el cumplimiento de las metas propuestas. De esta forma, en concordancia con el primer objetivo, se presentan a continuación los ejercicios de diseño orientado a objetos considerados.

CAPÍTULO 5

Ejercicios de diseño orientado a objetos

Este capítulo se desarrolla en el contexto del curso CI-0136 Diseño de Software, impartido a nivel de bachillerato en la carrera de Bachillerato en Computación con varios énfasis de la Escuela de Ciencias de la Computación e Informática en la Universidad de Costa Rica. Durante la ejecución del curso, se aplicaron algunos ejercicios sencillos tomados o basados en los presentados por (McLaughlin et al., 2006) y (Freeman et al., 2004). Estas obras son mencionadas por varios autores como útiles para el aprendizaje y entendimiento de patrones diseño (Allison & Harrison, 2007; Dukovich & Janzen, 2009; Tao et al., 2015).

No obstante, es importante entender que, como los mismos (Freeman et al., 2004) mencionan, en la vida real los desarrolladores deben centrarse en aspectos generales de diseño y no en patrones de diseño específicos. Los patrones de diseño deben usarse cuando se encuentra una necesidad natural para ellos, pero no se deben usar y se debe optar por una solución más simple cuando esto sea posible (Freeman et al., 2004). Esta perspectiva más pragmática, donde la simplicidad debe privar cuando sea posible, es de gran importancia en la práctica laboral.

Cuando se desarrolla software en el mundo real puede caerse en la tentación de implementar algo que parece atractivo pero que puede tener graves consecuencias en el futuro (Sarcar, 2018). Por ejemplo, podría elegirse utilizar una determinada tecnología o enfoque para conseguir una solución rápida a un problema inmediato. No obstante, si se exploran limitaciones a futuro se podría descubrir que tal alternativa va a llegar a costar más tiempo y dinero que si se toma un camino más complicado, pero correcto, desde inicio (Sarcar, 2018). Algo similar sucede con los patrones de diseño y los llamados antipatrones.

Los antipatrones son “soluciones recurrentes que crean más problemas de los que soluciona” (MacDonald, 2019, p. 193). Los antipatrones llevan de un problema a una mala solución, por lo que se han documentado de forma similar a los patrones de diseño. Generalmente incluyen

detalles que ayudan a reconocerlos, como por qué parecen atractivos, por qué la solución es mala en el largo plazo y además sugieren patrones de diseño que sí pueden llevar a buenas soluciones (Freeman et al., 2004).

Existen muchos contextos y ejemplos de antipatrones, pero un ejemplo sencillo de recordar es el llamado *golden hammer* (martillo de oro). Este se da cuando se requiere desarrollar un nuevo componente o sistema y el equipo de desarrollo solo está (o solo quiere estar) familiarizado con una única tecnología. Puede suceder que este nuevo elemento de software sea más apto para ser desarrollado con otra tecnología, pero el equipo decide forzarlo a que sea implementado con la única que conoce.

Ni las tecnologías, como se vio en el ejemplo anterior, ni los patrones de diseño deben introducirse a la fuerza en un sistema sólo porque sí. Es necesario pensar en diseño y no en patrones de diseño, pero para esto se requiere aprendizaje y experiencia, de forma que sea posible reconocer en qué situaciones aplican de forma natural y evitar caer en la sobreingeniería (Freeman et al., 2004).

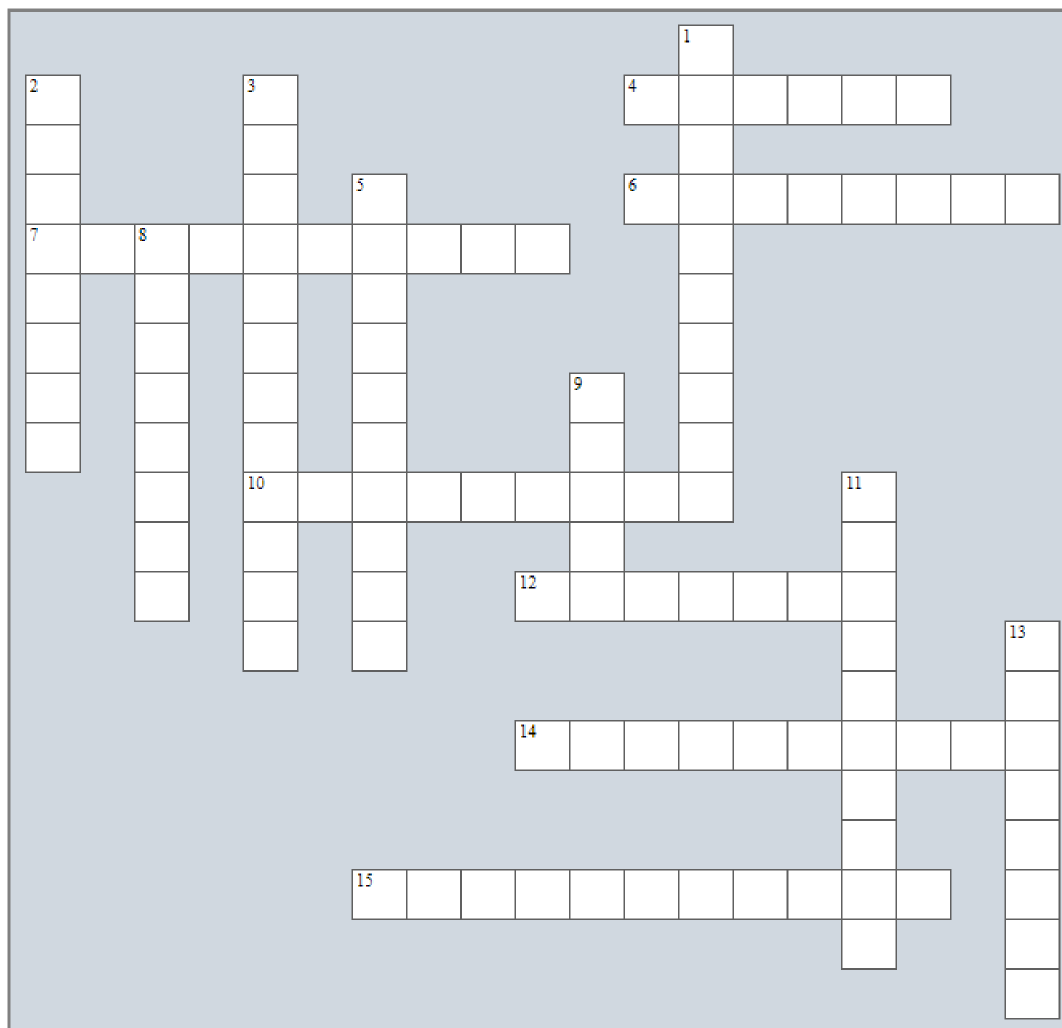
Con el fin de establecer este contraste entre lo académico y lo que se practica en la industria, se tomaron ejercicios de libros y sitios web que son regularmente usados en entrevistas laborales reales de grandes compañías de software. Estos ejercicios, por su naturaleza más práctica, se enfocan menos en conceptos, memorización y aplicación de patrones de diseño específicos, y más en diseño orientado a objetos en general. Se expondrán primero dos ejercicios académicos simples y luego otros dos de mayor complejidad que suelen citarse como ejemplos de preguntas hechas en entrevistas laborales.

Crucigrama de conceptos de análisis y diseño orientado a objetos

El primero de los ejercicios es un crucigrama de conceptos de análisis y diseño orientado a objetos, como se presenta en la Figura 1. Este crucigrama incluye conceptos seleccionados de acuerdo a los temas del curso de diseño en cuestión, pero la idea del juego como tal fue tomada de ejercicios como los que se presentan en (McLaughlin et al., 2006). Las respuestas a este crucigrama se pueden observar en la Tabla 2.

Tabla 2. Respuestas del crucigrama de conceptos de análisis y diseño orientado a objetos.

Dirección	*Across*	*Down*
Concepto	4. *repeat*	1. delegación
	6. herencia	2. depender
	7. encapsular	3. acoplamiento
	10. extensión	5. substituir
	12. cambiar	8. cohesión
	14. interfaces	9. única
	15. composición	11. agregación
		13. abstraer



Across

4. Don't ______ yourself
6. LSP revela problemas relacionados a ella
7. Se debe ______ lo que varía
10. Cerrado para modificación pero abierto para ______
12. Una clases bien diseñada debe tener una sola razón para ______
14. ______ específicas son mejores que una de propósito general
15. Cuando un objeto es dueño de comportamientos de otros objetos

Down

1. Transferir la responsabilidad de una tarea a otra clase o método
2. ______ de abstracciones y no de implementaciones
3. El código debe tener bajo ______
5. Subclases se deben poder ______ por su clase padre
8. El código debe tener alta ______
9. Toda clase debe tener una ______ responsabilidad
11. Variación de Composición
13. Se deben ______ las partes de código común

Figura 1. Crucigrama de conceptos de análisis y diseño orientado a objetos.

De forma similar a un crucigrama, un rompecabezas es otro tipo de ejercicio o juego que puede resultar familiar a los estudiantes y en general a cualquier persona.

Rompecabezas de diseño

El rompecabezas que se presenta a continuación, es un ejercicio tomado de (Freeman et al., 2004, pp. 79–80), que permite poner en práctica conceptos de diseño orientado a objetos, como herencia, composición e inclusive un patrón de diseño: *strategy.*

En la Figura 2 se presenta un conjunto de clases e interfaces relacionadas con un juego de aventura. Hay clases que representan personajes y otras que simbolizan comportamientos de armas que los personajes pueden usar. Cada personaje puede hacer uso de una sola arma a la vez, pero esta puede cambiarse en cualquier momento.

Tareas

1. Ordene las clases.
2. Identifique una clase abstracta, una interfaz y ocho clases.
3. Dibuje flechas entre las clases
    1. De este tipo para para herencia ("extends")
    2. De este tipo para para interfaz ("implements")
    3. De este tipo para para "HAS-A"
4. Ponga el método setWeapon() en la clase correcta

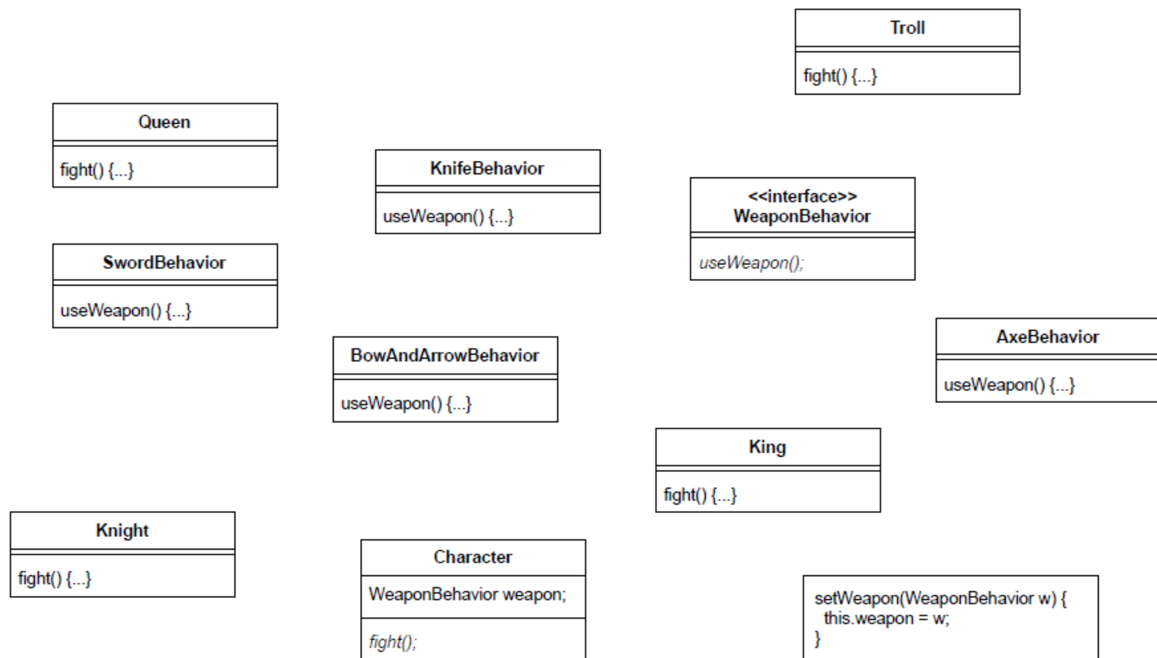


Figura 2. Rompecabezas de diseño. Tomado de (Freeman et al., 2004, pp. 78–80).

La solución a este rompecabezas se presenta en la Figura 3, en la que se detallan las relaciones que deben existir entre las clases abstractas, las concretas, interfaces y sus implementaciones, de forma que el ejercicio pueda ser resuelto de acuerdo con lo solicitado.

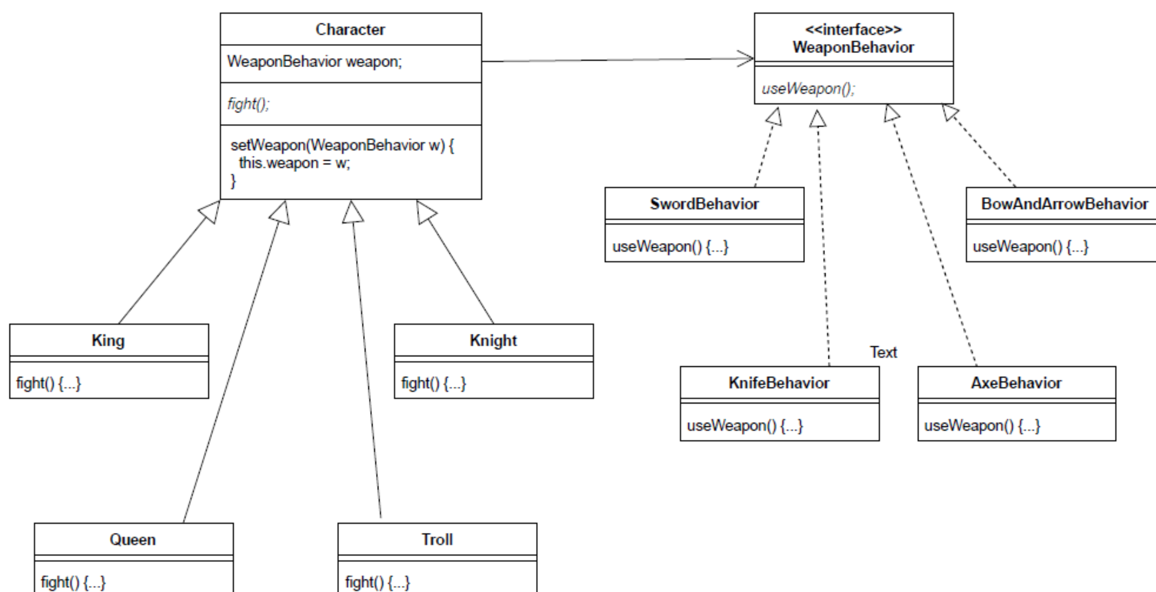


Figura 3. Solución al rompecabezas de diseño. Tomado de (Freeman et al., 2004, p. 94).

Los dos ejercicios presentados anteriormente son de origen académico y para ser ejecutados por estudiantes de ingeniería de software. Sin embargo, es importante conocer el tipo de preguntas y ejercicios que hacen compañías de software reales en sus procesos de reclutamiento.

Sistema de parqueo

En las entrevistas laborales de las más grandes compañías tecnológicas es usual enfrentarse a problemas relacionados con algoritmos y problemas de codificación (McDowell, 2015). En este contexto, (McDowell, 2015) presenta en su libro situaciones, preguntas y ejercicios a los que se ha enfrentado en entrevistas en compañías como Google, Microsoft, Apple y Amazon.

Se debe tomar en cuenta que, como dice (McDowell, 2015), las preguntas en entrevistas laborales reales acerca de diseño orientado a objetos, no se tratan de repetir conceptos teóricos

y listar patrones de diseño. Las preguntas suelen ser más acerca de demostrar entendimiento de cómo crear software orientado a objetos mantenible y elegante. La idea es probar la capacidad de un entrevistado para describir clases y métodos para implementar una solución utilizando objetos de la vida real.

Un ejemplo de un ejercicio de diseño orientado a objetos de este tipo es el de diseñar un sistema de parqueo. Las preguntas o solicitudes en una entrevista laboral normalmente serán vagas, como esta: "diseñe un parqueo utilizando principios de software orientado a objetos", pues es la intención del entrevistador que el entrevistado haga preguntas, converse y hasta formule supuestos (McDowell, 2015). Algunos de los supuestos para este ejercicio, que no son los únicos que se podrían hacer, son:

- El parqueo tiene varios pisos.
- Cada piso tiene varias filas de espacios.
- Es posible parquear motocicletas, autos y buses.
- El parqueo tiene lugares para motos, autos compactos y lugares grandes.
- Una motocicleta puede parquear en cualquier espacio.
- Un carro puede parquearse en un espacio para auto compacto o uno grande.
- Un bus puede parquearse en 5 lugares grandes consecutivos en la misma fila. Un bus, no se puede parquear en un espacio para auto compacto (McDowell, 2015).

El código completo para este problema puede ser encontrado en (CareerCup, 2015), pero a continuación se citan los detalles más relevantes de la solución. Para empezar, se crea una clase abstracta *Vehicle* de la que extienden *Car, Bus y Motorcycle*. Se crea también una clase *ParkingSpot* que tiene una propiedad que indica su tamaño, como se muestra en la Figura 4.

```
1   public enum VehicleSize { Motorcycle, Compact,    Large }
2
3   public abstract class Vehicle {
4     protected ArrayList<ParkingSpot> parkingSpots = new ArrayList<ParkingSpot>();
5     protected String licensePlate;
6     protected int spotsNeeded;
7     protected VehicleSize size;
8
9     public int getSpotsNeeded() { return spotsNeeded; }
10    public VehicleSize getSize() { return size; }
11
12    /* Park vehicle in this spot (among others, potentially) */
13    public void parkInSpot(ParkingSpot s) { parkingSpots.add(s); }
14
15    /* Remove car from spot, and notify spot that it's gone */
16    public void clearSpots() { ... }
17
18    /* Checks if the spot is big enough for the vehicle (and is available). This
19     * compares the SIZE only. It does not check if it has enough spots. */
20    public abstract boolean canFitInSpot(ParkingSpot spot);
21  }
22
23  public class Bus extends Vehicle {
24    public Bus() {
25      spotsNeeded = 5;
26      size = VehicleSize.Large;
27    }
28
29    /* Checks if the spot is a Large. Doesn't check num of spots */
30    public boolean canFitInSpot(ParkingSpot spot) { ... }
31  }
32
33  public class Car extends Vehicle {
34    public Car() {
35      spotsNeeded = 1;
36      size = VehicleSize.Compact;
37    }
38
39    /* Checks if the spot is a Compact or a Large. */
40    public boolean canFitInSpot(ParkingSpot spot) { ... }
41  }
42
43  public class Motorcycle extends Vehicle {
44    public Motorcycle() {
45      spotsNeeded = 1;
46      size = VehicleSize.Motorcycle;
47    }
48
49    public boolean canFitInSpot(ParkingSpot spot) { ... }
50  }
```

Figura 4. Clases principales del sistema de parqueo. Tomado de (McDowell, 2015, p. 313).

Además, se debe tener una clase *ParkingLot* que es un *wrapper* para un arreglo de niveles o pisos (*Level*). No es la única opción para atacar el problema, pero se hace así para separar la lógica que encuentra lugares y parquea los vehículos de las acciones más generales de *ParkingLot* (McDowell, 2015): Esto se muestra en la Figura 5.

```
1  public class ParkingLot {
2    private Level[] levels;
3    private final int NUM_LEVELS = 5;
4
5    public ParkingLot() { ... }
6
7    /* Park the vehicle in a spot (or multiple spots). Return false if failed. */
8    public boolean parkVehicle(Vehicle vehicle) { ... }
9  }
10
11 /* Represents a level in a parking garage */
12 public class Level {
13   private int floor;
14   private ParkingSpot[] spots;
15   private int availableSpots = 0; // number of free spots
16   private static final int SPOTS_PER_ROW = 10;
17
18   public Level(int flr, int numberSpots) { ... }
19
20   public int availableSpots() { return availableSpots; }
21
22   /* Find a place to park this vehicle. Return false if failed. */
23   public boolean parkVehicle(Vehicle vehicle) { ... }
24
25   /* Park a vehicle starting at the spot spotNumber, and continuing until
26    * vehicle.spotsNeeded. */
27   private boolean parkStartingAtSpot(int num, Vehicle v) { ... }
28
29   /* Find a spot to park this vehicle. Return index of spot, or -1 on failure. */
30   private int findAvailableSpots(Vehicle vehicle) { ... }
31
32   /* When a car was removed from the spot, increment availableSpots */
33   public void spotFreed() { availableSpots++; }
34 }
```

Figura 5. Clases ParkingLot y Level. Tomado de (McDowell, 2015, p. 314).

Finalmente, la clase ParkingSpot simplemente tiene una propiedad que representa el tamaño del espacio, como se puede ver en la Figura 6.

```
1   public class ParkingSpot {
2     private Vehicle vehicle;
3     private VehicleSize spotSize;
4     private int row;
5     private int spotNumber;
6     private Level level;
7
8     public ParkingSpot(Level lvl, int r, int n, VehicleSize s) {...}
9
10    public boolean isAvailable() { return vehicle == null; }
11
12    /* Check if the spot is big enough and is available */
13    public boolean canFitVehicle(Vehicle vehicle) { ... }
14
15    /* Park vehicle in this spot. */
16    public boolean park(Vehicle v) { ... }
17
18    public int getRow() { return row; }
19    public int getSpotNumber() { return spotNumber; }
20
21    /* Remove vehicle from spot, and notify level that a new spot is available */
22    public void removeVehicle() { ... }
23  }
```

Figura 6. Clase ParkingSpot. Tomado de (McDowell, 2015, pp. 314–315).

Este ejercicio, con algunas modificaciones, fue incluido en el proyecto final del curso de diseño. Este tipo de problemas de diseño son candidatos ideales para ser incluidos en proyectos similares en futuras versiones del curso de diseño. El proyecto incluyó otras tareas que tenían como meta automatizar servicios que una nueva torre de apartamentos ofrecería a sus inquilinos y clientes. Por motivos didácticos y de objetivos del curso, la solución debía incluir el uso de diferentes patrones de diseño, pero en este ejemplo se ha hecho referencia a un diseño e implementación más libre, como sucede en las entrevistas laborales.

Ejercicios más avanzados que los vistos anteriormente, pero más cercanos a los de entrevista laboral reales, son los de diseño de sistemas más complejos. Algunos de estos ejercicios pueden ser acerca de cómo diseñar sistemas populares, como Netflix, Instagram, Dropbox o Twitter (Educative Inc, 2021).

Diseñar Twitter

En la red social Twitter los usuarios pueden publicar y leer mensajes cortos de 140 caracteres llamados tuits (*tweets* en inglés). Los usuarios registrados pueden publicar y leer tuits, pero los que no están registrados solo pueden leerlos (Educative Inc, 2021). Un entrevistador podría presentar el enunciado de este ejercicio como una lista de requerimientos funcionales y no funcionales que deben ser satisfechos para crear una versión simplificada de Twitter. Entre los requerimientos funcionales podría solicitarse cumplir con lo siguiente:

1. Los usuarios deben poder publicar tuits.
2. Un usuario debe poder seguir a otros usuarios.
3. Un usuario debe poder marcar tuits como favoritos.
4. El sistema debe poder crear y desplegar el *timeline* (cronología) de un usuario que consiste en los eventos más recientes de los otros usuarios que sigue.
5. Los tuits pueden contener fotos y videos (Educative Inc, 2021).

Podrían existir otros requerimientos adicionales para un sistema más complejo y cercano a Twitter, como, por ejemplo, buscar tuits, etiquetar usuarios, tendencias y otros, pero esto sería demasiado para una pregunta práctica en una entrevista laboral.

En lo que respecta a requerimientos no funcionales, estos podrían requerir más conocimiento y experiencia de la que podría tener un estudiante universitario de nivel intermedio, pero se presentan para completar el escenario que se quiere describir. Algunos de estos requerimientos no funcionales podrían ser de la forma:

1. El servicio debe estar siempre altamente disponible.
2. La generación del *timeline* tiene una latencia máxima permitida de 200 milisegundos.
3. Se prefiere la disponibilidad sobre la consistencia. Esto es, algunos tuits podrían no ser visibles para algunos usuarios por algún tiempo corto (Educative Inc, 2021).

El primer paso para diseñar el sistema sería definir el API (*Application Programming Interface*) de tipo REST (*REpresentational State Transfer*), por ejemplo, para exponer la funcionalidad solicitada. Publicar un tuit podría hacerse por medio del siguiente servicio:

```
tweet(api_dev_key, tweet_data, tweet_location, user_location, media_ids)
```

donde los parámetros tienen el siguiente significado:

- api_dev_key (hilera de caracteres): el identificador del desarrollador del API. Necesario para el desarrollo y pruebas sobre la plataforma.
- tweet_data (hilera de caracteres): el texto del tuit, de 140 caracteres como máximo.
- tweet_location (hilera de caracteres): localización a la que se refiere el tuit en términos de latitud y longitud (opcional).
- user_location (hilera de caracteres): localización del usuario que publica el tuit en términos de latitud y longitud (opcional).
- media_ids (arreglo de números): lista de identificadores de fotos y videos asociados al tuit que deben ser cargados de forma separada (opcional).

El valor de retorno de este servicio sería una hilera de caracteres que contenga el URL (*Uniform Resource Locator*) para acceder al tuit o un error de HTTP (*Hypertext Transfer Protocol*) (Educative Inc, 2021). Habría que seguir un enfoque similar para describir el API para cada uno de los restantes requerimientos funcionales, pero nuevamente, lo que interesa aquí es mostrar un ejemplo manejable de una pregunta de diseño de este tipo.

Adicionalmente, sería necesario, como parte de la respuesta a este ejercicio, hacer preguntas específicas sobre detalles desconocidos o bien plantear supuestos y verificar que sean aceptables. Por ejemplo:

- ¿Cuántos usuarios activos por día se tendrían? Probablemente millones.

- ¿Cuántos tuits nuevos por día manejaría el sistema?, ¿cuántos tweets se marcarían como favoritos por día?, ¿cuántas veces se vería un mismo tuit por día? Seguramente las respuestas a todas estas preguntas estarían también en el orden de millones.
- ¿Cuáles serían las necesidades de almacenamiento? Se debería considerar el tamaño de un tuit (140 caracteres) y asumir que, por ejemplo, para un solo tuit se necesitarían 280 bytes para el texto más 30 bytes para la información de control. Por tanto, un solo tuit necesitaría de 310 bytes y para saber cuánto espacio es necesario para almacenar toda la información en un día se podría multiplicar 310 por el número de tuits nuevos por día.
- ¿Cuáles serían las necesidades de ancho de banda? Se podría seguir un enfoque similar al del cálculo del espacio de almacenamiento para estimar cuántos gigabytes por segundo sería necesario soportar (Educative Inc, 2021).

Con todo lo anterior en mente se podría proponer un diseño de alto nivel capaz de soportar esta carga. Sería necesario tener múltiples servidores de aplicación con balanceo de carga para distribuir el tráfico. También, se necesitaría de bases de datos eficientes para poder escribir y leer un número grande de tuits, así como almacenamiento para fotos y videos (Educative Inc, 2021). En una entrevista real, una pizarra sería útil para presentar un diseño de alto nivel del sistema como el que se muestra en la Figura 7.

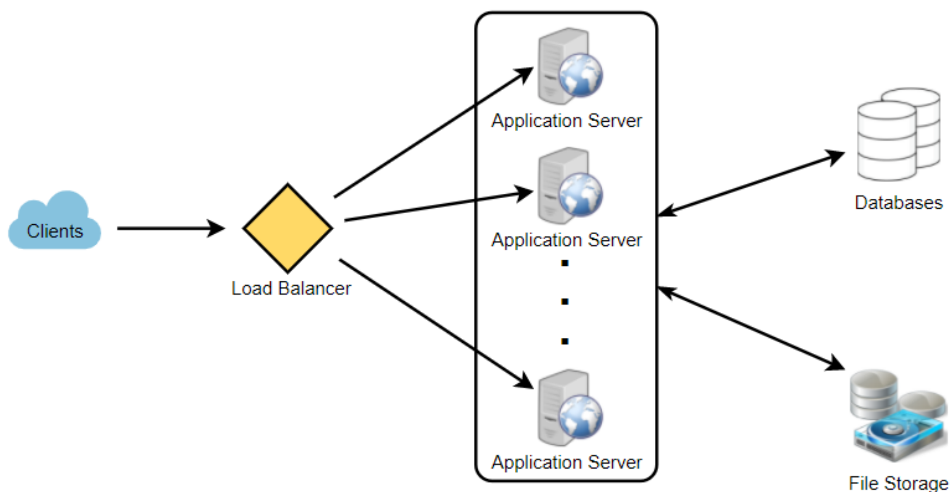


Figura 7. Diseño de alto nivel de sistema similar a Twitter. Tomado de (Educative Inc, 2021).

Finalmente, se debería también hablar de un diseño de bases de datos para soportar las necesidades de almacenamiento de información de usuarios, tuits, favoritos y los usuarios que son seguidos. Se podría hablar también de cómo segmentar la cantidad masiva de datos en la base datos (*sharding*) para optimizar rendimiento y tiempos de respuesta, o de la inclusión de un sistema de caché por motivos similares, pero sería ahondar mucho en los detalles de este ejemplo. Un diseño simple de una base de datos relacional para este sistema podría ser como el que se expone en la Figura 8.

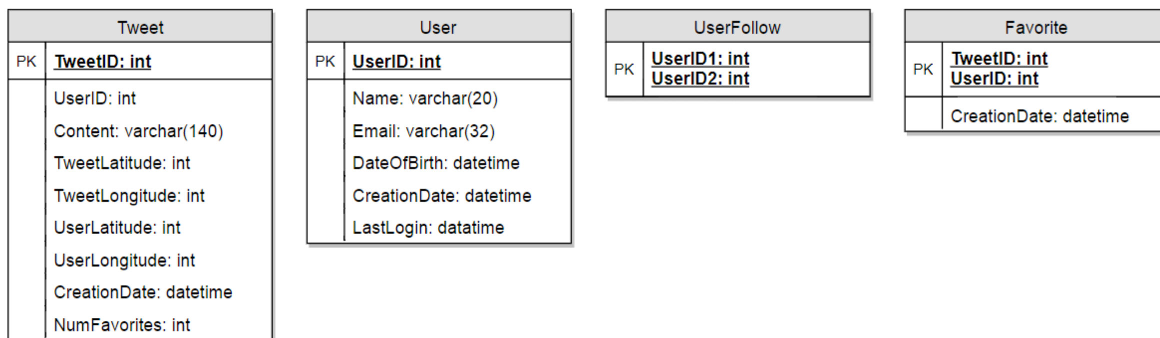


Figura 8. Diseño de base de datos de sistema similar a Twitter. Tomado de (Educative Inc, 2021).

De vuelta a los requerimientos más simples del curso de diseño en torno al que gira este trabajo, se debe recordar que el aprendizaje de patrones diseño y, en general, de análisis y diseño orientado a objetos, comienza por conocer conceptos y elementos más básicos. En la siguiente sección se presentan las analogía o metáforas de diseño que se elaboraron para ayudar a que los estudiantes pudieran relacionar los 23 patrones de diseño del GoF con elementos que les pudieran ser familiares en el mundo real. Se presenta también la valoración que hicieron ellos de la claridad de cada una de las propuestas que se hicieron para la metáfora de cada patrón de diseño.

CAPÍTULO 5

Analogías de los patrones de diseño y su evaluación

En esta sección se presentan las analogías o metáforas que se elaboraron para cada patrón de diseño, así como los resultados de la encuesta que se aplicó para evaluar la claridad de estas, según la percepción de los estudiantes del curso usado como caso de estudio en este trabajo.

Analogías de los patrones de diseño

A continuación, se expondrán las metáforas de todos los patrones de diseño empezando por los creacionales, siguiendo con los patrones estructurales, y terminando con los de comportamiento. La idea, en cada caso, es presentar un ejemplo o situación natural del mundo real que sirva para representar cada patrón de diseño de una forma tan clara y fácil de recordar como sea posible.

1. Abstract Factory

El patrón de diseño creacional *abstract factory* tiene que ver con la creación de familias de productos (SourceMaking.com, 2019). Este patrón se puede ejemplificar haciendo referencia a un fabricante de autos y la forma en que construyen sus diferentes modelos, usando piezas con funcionalidades similares, pero que tienen detalles específicos según el modelo.

Por ejemplo, Toyota es un fabricante de automóviles y tiene modelos como RAV4, Hilux y Yaris (ver Figura 9). Todos esos modelos tienen elementos en común. Si se desea construir un carro nuevo se necesitará de llantas, frenos, motor, carrocería y ventanas, entre otras partes. Este conjunto de piezas forma la definición de una familia de productos para un modelo en específico y se debe contar con una versión específica de cada parte para cada modelo. Al iniciar la fabricación de un carro nuevo debe conocerse el modelo a fabricar. Si este es, por ejemplo, un Hilux, es necesario poder obtener las llantas, los frenos, el motor, la carrocería y las ventanas de un vehículo Hilux y no los de otro modelo (SourceMaking.com, 2019).

Figura 9. Analogía de abstract factory con fábrica de automóviles.

2. Builder

El patrón *builder* es uno creacional que permite construir objectos complejos paso a paso y puede ser comparado con un producto del mundo real de una cadena de restaurantes de comida rápida (SourceMaking.com, 2019).

La Cajita Feliz es la marca del menú infantil de McDonald's, que contiene un elemento principal, por ejemplo, hamburguesa o *nuggets*; un elemento adicional, como papas fritas; una bebida como jugo o agua, por lo general y un juguete (ver Figura 10). Los contenidos de la Cajita pueden variar, pero su proceso de construcción es siempre el mismo. El cajero ingresa la información de que productos fueron solicitados para cada categoría, sus compañeros reciben la orden y preparan la Cajita Feliz con la comida, el juguete y por aparte se servirá la bebida. Este proceso de "construcción" será siempre el mismo sin importar los elementos específicos de la orden (SourceMaking.com, 2019).

Figura 10. Analogía de builder con una "Cajita Feliz".

3. Factory Method

Suponga que existe una pizzería en Estados Unidos que nació en Nueva York, pero ha tenido tanto éxito que se quiere abrir franquicias en otras ciudades como Chicago y San Francisco. Cada ciudad debe poder tener variaciones en el estilo y los tipos de ingredientes que se utilizarán para cada pizza, por motivos de producción local, precios y preferencias de sus habitantes. Por ejemplo, la pizza en Nueva York tendría una pasta delgada y en Chicago gruesa, pero se debe garantizar que todas las franquicias cumplan con los mismos estándares y procesos de la pizzería original (Freeman et al., 2004).

Estos requerimientos implican que la pizza debe poder tener ingredientes con variaciones locales, pero los procesos generales deben ser los mismos en todas las franquicias para garantizar una calidad uniforme del producto (ver Figura 11). Operaciones como la preparación general, el horneado, cortado y empaquetado de la pizza deben ser las mismas en todas las pizzerías, pero el proceso de “creación” de la pizza con sus ingredientes locales específicos puede variar en cada ciudad (Freeman et al., 2004). Este paso especifico de “creación” de la pizza es análogo con el de un *factory method,* que es un patrón de diseño creacional que encapsula la creación de un objeto al permitir a subclases decidir qué objeto instanciar (Freeman et al., 2004).

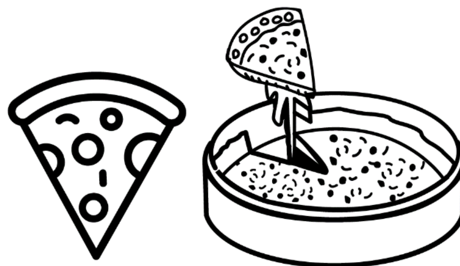


Figura 11. Analogía de abstract factory con la creación de diferentes estilos de pizza.

4. Prototype

Prototype es otro patrón de diseño creacional, que, en su caso, tiene como objetivo crear nuevos objetos al copiar o clonar uno existente (Shvets, 2019).

Una analogía interesante de este patrón con un proceso del mundo real, en este caso de la biología, es la de la división de las células por mitosis (ver Figura 12). Este proceso resulta en dos células genéticamente idénticas, por lo que se puede decir que la célula se clona a sí misma (Shvets, 2019).

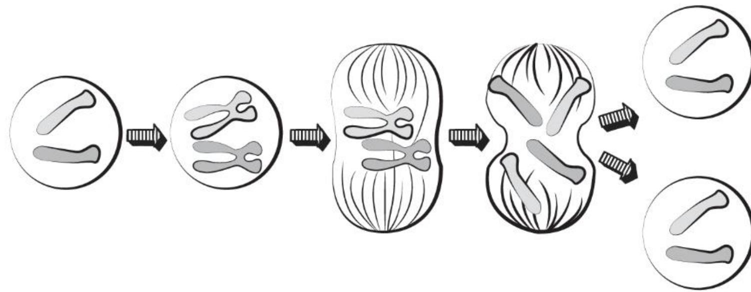


Figura 12. Analogía de prototype con la mitosis.

5. Singleton

Imagine que se tiene una oficina con docenas de colaboradores y sus computadoras: ¿será buena idea tener una impresora por cada colaborador y computadora? Sin duda sería posible, pero esto implicaría muchos gastos extra, alta ociosidad de estas impresoras, así como múltiples puntos de mantenimiento y fallo (Potts, 2018).

En contraposición al esquema recién mencionado, se podría contar con una única impresora, robusta y con una única cola de trabajo, a la que todas computadoras tuvieran acceso (ver Figura 13). La impresora podría ser un poco más complicada de configurar al inicio, pues tendría este único punto de entrada para servir a varios clientes, pero sería más fácil de modificar y escalar de acuerdo con las necesidades de impresión de la oficina (Potts, 2018). Aquí, como en el patrón creacional *singleton*, se tiene una sola instancia de "algo" y un punto global de acceso a ella (Shvets, 2019).

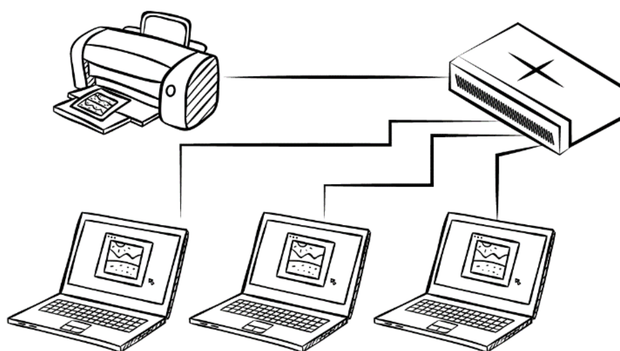


Figura 13. Analogía de singleton con una única impresora de acceso global.

6. Adapter

Adapter es un patrón de diseño estructural, que se puede representar por medio de una situación típica cuando se viaja entre diferentes continentes, o en ocasiones inclusive entre países en un mismo continente, pero con diferentes estándares eléctricos (Freeman et al., 2004).

Un enchufe de tipo americano estándar no puede ser usado en tomacorrientes europeos, pues tienen interfaces diferentes. Si se espera poder conectar un artefacto eléctrico que tiene un enchufe de dos clavijas a una toma que funciona para enchufes de tres clavijas, se tendrá un problema. La solución es un dispositivo que se pondrá en medio, donde se pueden insertar las dos clavijas del artefacto eléctrico y que a su vez cuenta con las tres clavijas que el tomacorriente está esperando. De esta forma, se hace posible que fluya la electricidad entre dos interfaces incompatibles, como se muestra en la Figura 14 (Freeman et al., 2004).

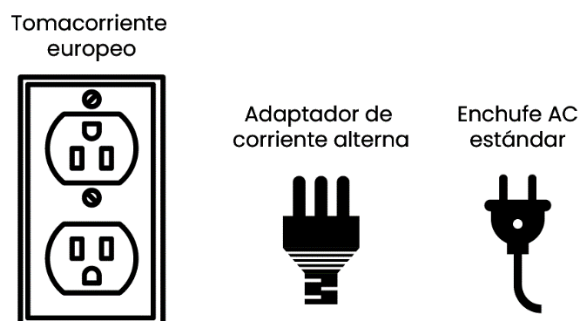


Figura 14. Analogía de adapter con un adaptador de corriente.

7. Bridge

El control remoto de un televisor o radio puede tener una operación de “arriba” para cambiar el canal o estación a la siguiente. Esta operación, en términos del dispositivo principal (televisor o radio), se descompone en una “llamada” para obtener el canal actual y otra para solicitar sintonizar el canal más uno (Shvets, 2019).

En el caso del ejemplo anterior, como en el del patrón de diseño *bridge*, el control remoto funciona como una abstracción del dispositivo que controla y puede tener acciones o botones que agrupan varias funciones pequeñas individuales que existen en el televisor o radio (ver Figura 15). Todos los controles remotos trabajan contra un dispositivo por medio de una interfaz general, permitiendo que el mismo control puede manejar diferentes tipos de dispositivos (Shvets, 2019).

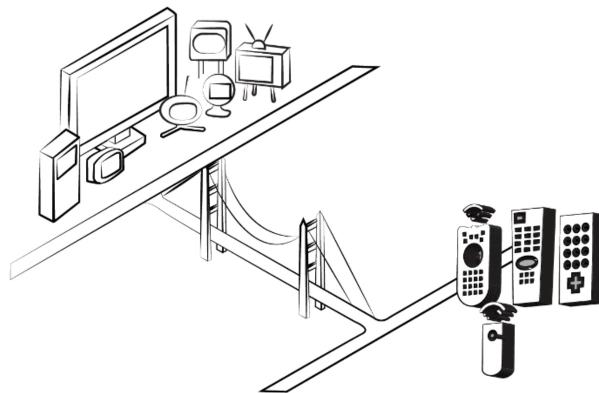


Figura 15. Analogía de bridge con televisores y controles remotos.

8. Composite

Una expresión aritmética consta de un operando, un operador (+, -, *, /) y otro operando. Cada operando puede ser un número u otra expresión aritmética. De esta forma, tanto *2+3* como *(2+3) + 4* y también *(2+3) + (4*6)* son todas expresiones aritméticas válidas (ver Figura 16).

Así, tal como en el patrón de diseño estructural *composite,* cada operando puede ser un número individual u otra operación en sí misma. Tanto los objetos individuales, como los complejos, son tratados de la misma forma (SourceMaking.com, 2019).

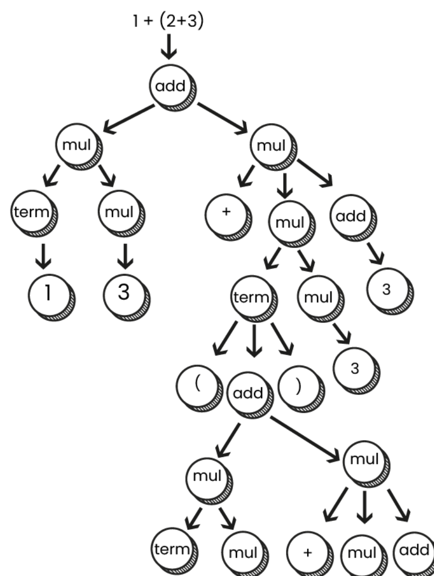


Figura 16. Analogía de composite con una expresión aritmética.

9. Decorator

El patrón de diseño *decorator* es uno estructural que permite añadir comportamientos o características a un objeto, al poner ese objecto dentro de otro que tiene esas nuevas características (Shvets, 2019).

Por ejemplo, si hace frío alguien puede vestir un suéter y si el frío es mayor podría ponerse una chaqueta sobre el suéter. Si aparte de frío intenso, hay lluvia, podría además usarse un impermeable (ver Figura 17). En cada caso, cada “capa” de ropa agrega una funcionalidad extra a la vestimenta. Por ejemplo: el suéter quita el frío, pero no tiene bolsillos, la chaqueta protege aún más del frío y aparte tiene bolsillos. El impermeable remedia, además, el problema de mojarse con la lluvia y agrega un gorro para no mojarse la cabeza (Shvets, 2019).

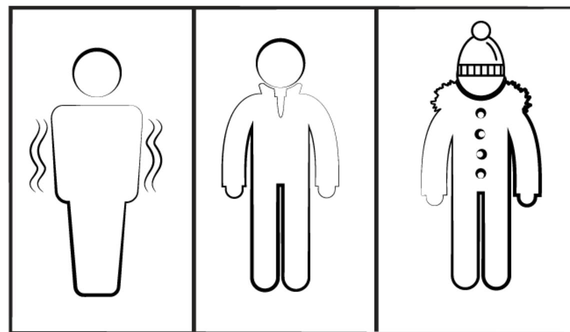


Figura 17. Analogía de decorator con capas de ropa.

10. Façade

Cuando alguien tiene una emergencia, esta puede ser relativamente simple o compleja. Normalmente, lo que hace una persona ante una emergencia es llamar al 911, identificarse, describir la emergencia, dar una dirección y los operadores se pondrán en contacto con todas las entidades pertinentes. Por ejemplo, en un caso en el que alguien se cayó y se golpeó, en el 911 podrían tener que contactar sólo a la Cruz Roja. Sin embargo, en otro más complejo, como el de un incendio en un local comercial donde hay mucha gente e incluso heridos, deberían hacer todos los contactos para que la emergencia sea atendida por Cruz Roja, bomberos y policía (ver Figura 18).

De esta forma, al igual que en el patrón de diseño estructural *facade*, se define una interfaz unificada de alto nivel que funciona como fachada para acceder a otros servicios (SourceMaking.com, 2019).

Figura 18. Analogía de facade con servicio 911.

11. Flyweight

El patrón de diseño estructural *flyweight* permite ahorrar memoria al usar una instancia de una clase para proveer muchas "instancias virtuales" (Freeman et al., 2004, p. 332). Esta es una definición muy técnica que hace difícil comparar este patrón con alguna situación del mundo real, pero se puede hacer una analogía con alguna aplicación de software que la mayoría de la gente conoce.

La administración de canciones y *playlists* en aplicaciones web y móviles proporciona un buen ejemplo de este patrón en el mundo virtual (ver Figura 19). En cualquiera de estas aplicaciones es posible tener diferentes *playlists* que pueden contener las mismas canciones, pero en cada *playlist* esa canción podría tener una posición diferente, así como otra información adicional en el contexto de esa lista, como su creador, por ejemplo. (Blancarte, 2016). De esta forma, cada canción es la misma en toda lista, pero tiene información extra relacionada de acuerdo con su contexto.

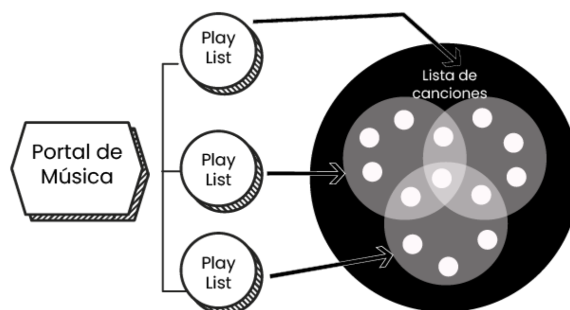


Figura 19. Analogía de flyweight con aplicación de música.

12. Proxy

Una tarjeta de débito o crédito es el "representante" de una cuenta bancaria, que a su vez es la "representante" del dinero real. Tanto la tarjeta, como el dinero real, pueden ser usadas para hacer un pago, por lo que se puede decir que implementan la misma "interfaz". En este caso, la tarjeta controla la forma en que se accede al dinero real en la cuenta bancaria (ver Figura 20) (Shvets, 2019).

El comportamiento y relación entre una tarjeta de débito o crédito y el dinero real, es similar al del patrón de diseño estructural *proxy*, que proporciona un objeto intermediario para acceder al objeto original (Shvets, 2019).

Figura 20. Analogía de proxy con tarjetas de débito y crédito.

13. Chain of responsibility

El patrón de diseño de comportamiento *chain of responsibility* permite pasar una solicitud a través de una cadena de receptores hasta que alguno la pueda procesar (SourceMaking.com, 2019).

Por ejemplo, si alguien requiere hacer un reclamo por una transacción monetaria no autorizada en una cuenta bancaria, normalmente esta persona llamará al servicio telefónico del banco en cuestión como paso inicial. Típicamente, la primera etapa constaría de una computadora dando opciones de qué número marcar dependiendo del servicio requerido. Se hace lo necesario para ser atendido por un operador, se le explica el caso y el operador podría determinar que no tiene el rol o privilegios necesarios para procesar un reclamo de este tipo y transferiría la llamada a un supervisor que sí los tenga. Este último eslabón se encargaría de procesar la solicitud y terminaría el proceso de atención (ver Figura 21).

Figura 21. Analogía de chain of responsibility con servicio de atención bancaria.

14. Command

Suponga que se tiene un control remoto para automatización de tareas en una casa, como por ejemplo encender y apagar luces, ventiladores, abrir y cerrar la puerta de la cochera, entre otras. El control remoto tiene una serie de botones sin configurar y para cada uno de ellos se puede seleccionar una acción diferente. El control remoto no sabe qué acción se ejecutará ni sobre qué objetivo (luz, ventilador, puerta), sólo sabe que si el botón es presionado se invocará una acción sobre su objetivo (Freeman et al., 2004). Cada botón puede ser configurado con cualquier acción y ejecutarse de forma transparente sin que el control remoto se vea afectado (ver Figura 22).

La situación anterior es análoga al objetivo del patrón de diseño de comportamiento *command*, que permitir encapsular la información de una solicitud de forma que pueda usarse para configurar o parametrizar diferentes clientes (Freeman et al., 2004).

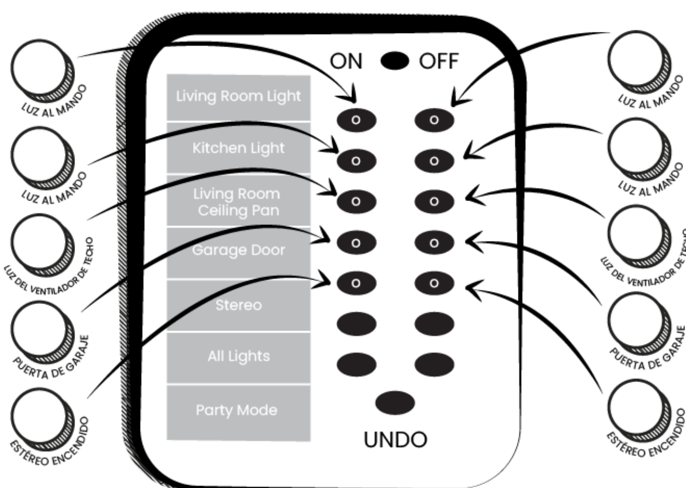


Figura 22. Analogía de command con control remoto programable.

15. Interpreter

El patrón de comportamiento *interpreter* permite, dado un lenguaje, definir una representación para su gramática junto con un intérprete que usa esa representación para interpretar sentencias en ese lenguaje (SourceMaking.com, 2019).

El mundo de la música proporciona un buen ejemplo de este patrón, pues el tono y duración de un sonido pueden ser representados en un lenguaje o notación musical sobre un pentagrama (ver Figura 23). Esta notación define un lenguaje para la música, de forma que los músicos puedan tocar a partir de una partitura y así reproducir los tonos y duraciones que están representados para cada sonido (SourceMaking.com, 2019).

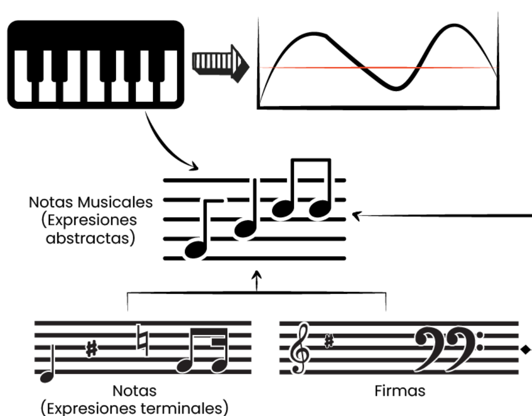


Figura 23. Analogía de interpreter con la música y las partituras.

16. Iterator

Suponga que está en un hotel en otro país y no conoce los canales de televisión, pero quiere buscar algún programa que le interese. Lo usual sería tomar el control remoto, encender el televisor y empezar a cambiar los canales en secuencia. El control remoto le dará la orden al televisor de sintonizar el próximo canal (o el anterior), pero él no sabe nada acerca de los canales, sólo “ordena” recorrerlos (ver Figura 24). Usted podrá elegir algún canal basado en algún criterio, como por ejemplo si ese canal está en su idioma o si está presentando música de su agrado, o bien puede continuar evaluando los siguientes canales (SourceMaking.com, 2019).

De esta forma, el control remoto, tal como el patrón de comportamiento *iterator*, provee una manera de recorrer una lista de elementos sin necesidad de conocer mayores detalles acerca de ellos (SourceMaking.com, 2019).

Figura 24. Analogía de iterator con un control remoto recorriendo canales.

17. Mediator

Los pilotos de aeronaves que se aproximan o despegan del área de control de un aeropuerto no se comunican directamente entre ellos. Los pilotos se comunican con un controlador aéreo en una torre cerca de la pista del aeropuerto. Sin este controlador aéreo los pilotos tendrían que conocer acerca de todas las otras aeronaves cerca del aeropuerto y discutir directamente con decenas de otros pilotos las prioridades de aterrizaje y despegue, causando un caos total (ver Figura 25) (Shvets, 2019).

Así, un controlador aéreo se comporta como el patrón de diseño *mediator*, que reduce dependencias entre diferentes entidades y restringe su comunicación a un único punto central (Shvets, 2019).

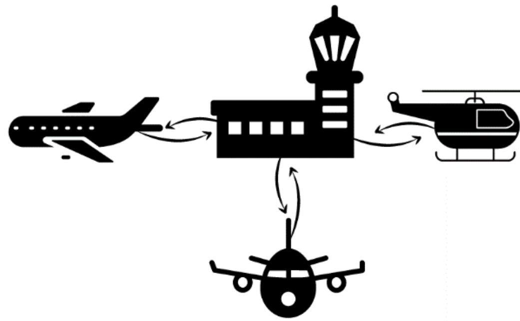


Figura 25. Analogía de mediator con controlador aéreo.

18. Memento

El patrón de diseño de comportamiento memento es otro complicado de comparar con elementos o situaciones del mundo físico tangible. Es un patrón que permite guardar y restaurar el estado previo de un objeto sin revelar detalles de su implementación (Shvets, 2019).

Memento podría compararse con algunas características de las transacciones en una base de datos con propiedades ACID (*Atomicity, Consistency, Isolation, Durability*). Por ejemplo, si todas las operaciones contenidas en una transacción son exitosas hay un *commit* y los cambios son guardados como finales, sumando este estado más reciente al historial de cambios de los datos. Caso contrario, si una transacción falla, todos los cambios efectuados por las operaciones son deshechos y la base de datos vuelve a su estado previo, como si nada hubiera sucedido, tal como se muestra en la Figura 26 (Starcevic, 2016).

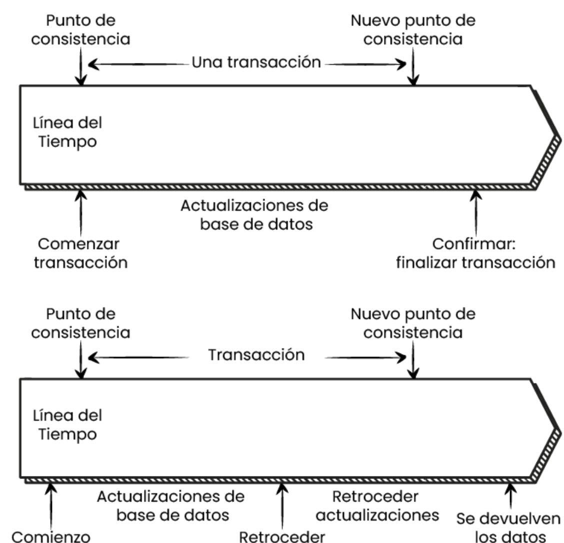


Figura 26. Analogía de memento con transacciones ACID.

19. Observer

Imagine que un nuevo periódico comienza a funcionar y a publicar sus ediciones diarias. Usted se suscribe y cada vez que hay una nueva edición, esta le es entregada. Mientras usted se mantenga como suscriptor recibirá todas las nuevas ediciones cada día. Usted se desinscribe cuando ya no quiere recibir más las ediciones diarias y estas ya no le serán entregadas (ver Figura 27). Mientras el periódico se mantenga en el negocio, todos los potenciales clientes (personas, comercios, hoteles, etc.) se pueden suscribir y desinscribir de él (Shvets, 2019).

Lo anterior, es análogo a lo que describe el patrón de diseño de comportamiento *observer*, que permite definir un mecanismo de suscripción a un objeto que notificará acerca de nuevos eventos a quienes se registren y se mantengan observando a ese objeto (Shvets, 2019).

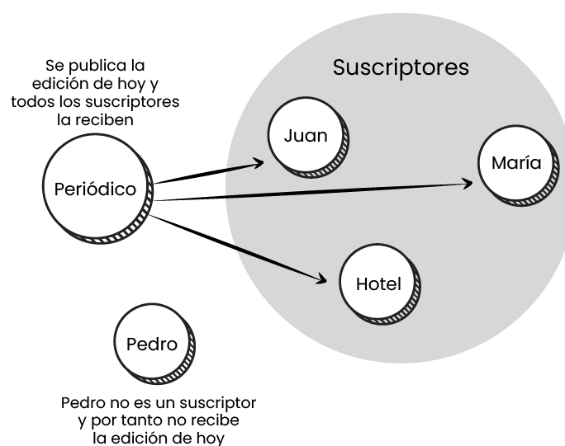


Figura 27. Analogía de observer con suscripción a un periódico.

20. State

El patrón de diseño *state* permite a un objeto modificar su comportamiento cuando su estado interno cambia (SourceMaking.com, 2019). Por ejemplo, el comportamiento de una máquina expendedora variará dependiendo de su estado actual con respecto al dinero recibido y a su inventario (ver Figura 28).

Si se intenta obtener una bebida o una golosina de esta máquina y no se ha insertado el dinero, se obtendrá un mensaje de error e información cómo proceder. Una vez se deposite el dinero la máquina revisará si el monto es el requerido, si no lo es, mostrará un mensaje de error y retornará el dinero. Si el monto es exactamente el requerido, dispensará el producto sin dar cambio, pero si el monto es mayor al requerido deberá dispensar el producto y además entregar el cambio correspondiente. Otro caso es en el que el monto es mayor o igual a lo requerido, pero no hay inventario del producto solicitado, en cuyo caso debe devolver el dinero e informar al usuario de la situación (SourceMaking.com, 2019).

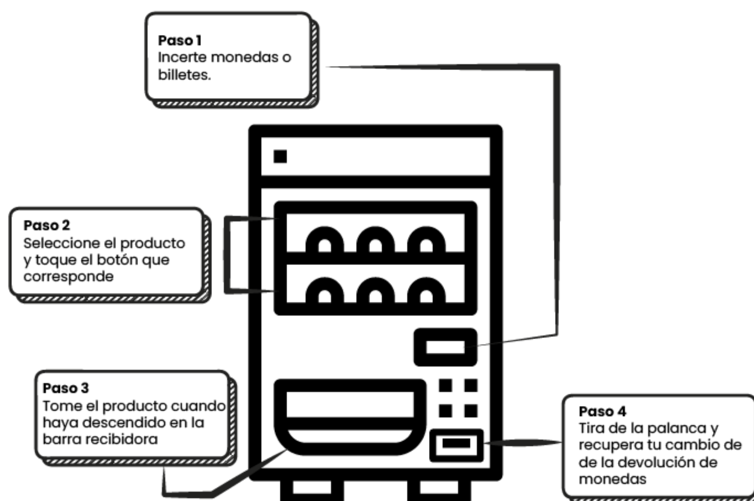


Figura 28. Analogía de state con máquina expendedora.

21. Strategy

Suponga que debe llegar al aeropuerto y cualquier medio de transporte lo ayudará a conseguir esa meta, pero de diferentes formas (ver Figura 29). Por ejemplo, si el día está soleado y el aeropuerto está cerca, podría decidir caminar o transportarse en bicicleta (si tiene donde dejarla en el aeropuerto). Otra opción podría ser ir en carro y estacionarlo en el parqueo e incurrir en un gasto importante de pagar por varios días de servicio de parqueo. Podría también decidir ir en bus o en tren, a un costo bajo y llegando de una forma expedita (Shvets, 2019).

Cada una de las opciones descritas sería una diferente estrategia, similar a lo que describe el patrón de diseño *strategy*. Este patrón de comportamiento permite definir una familia de algoritmos, encapsularlos y usarlos de forma independiente e intercambiable según la situación (Shvets, 2019).

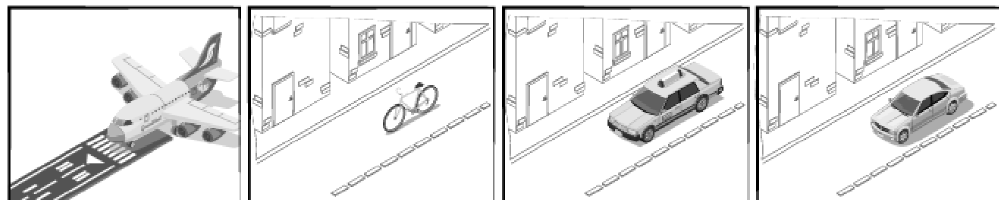


Figura 29. Analogía de strategy con medios de transporte.

22. Template Method

El patrón de comportamiento *template method* define el esqueleto de un algoritmo en una clase padre, pero permite a clases hijas sobrescribir partes de ese algoritmo sin cambiar su estructura general (Shvets, 2019). De la misma forma, puede haber diferentes formas y secuencias de actividades al construir una casa, pero su "algoritmo" principal tendrá pasos que deben llevar a fuerza una secuencia.

Esta secuencia podría implicar, primero hacer los movimientos de tierra, luego sentar las bases, más tarde construir la estructura principal, instalar paredes, plomería y electricidad, entre otras tareas. No es posible pensar en levantar paredes sin bases, o instalar ventanas sin paredes, pero las paredes podrían ser de concreto o de madera, así como las ventanas podrían ser de un estilo o tipo de vidrio en particular (ver Figura 30). Así, se tiene un "algoritmo" general para construir una casa, pero diversos detalles pueden ser ajustados de acuerdo con las necesidades (Shvets, 2019).

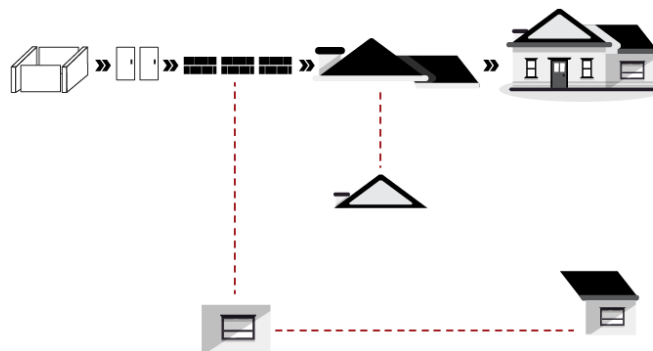


Figura 30. Analogía de template method con la construcción de una casa.

23. Visitor

El patrón de diseño de comportamiento *visitor* representa una operación que se puede ejecutar sobre elementos de una estructura, sin cambiar esos elementos sobre los que opera (SourceMaking.com, 2019).

Por ejemplo, cuando se solicita un servicio de transporte a una compañía de taxi, se está aceptando un visitante y entonces la compañía envía un transporte al cliente (ver Figura 31). Una vez que llega el vehículo y se aborda, ya el cliente no está en control de su movilización, sino que ahora lo está el transporte aceptado (SourceMaking.com, 2019).

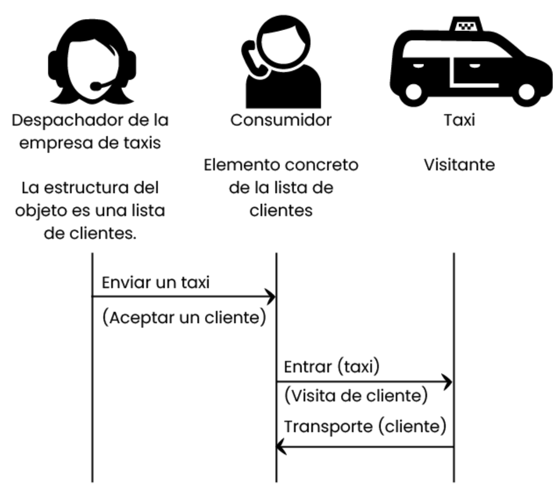


Figura 31. Analogía de visitor con servicio de taxi.

Evaluación de las analogías

Cada una de las analogías o metáforas tenía como propósito presentar a los estudiantes una comparación clara y útil de cada patrón de diseño con elementos del mundo real que les pudieran resultar familiares.

Para evaluar las analogías, estas fueron presentadas, una a una, al grupo de 16 estudiantes del curso de diseño, sin especificar cuál correspondía a cuál patrón de diseño. Se elaboró una encuesta con la lista de los 23 patrones de patrones de diseño en orden aleatorio y, al terminar de exponer cada una, se pidió a los estudiantes adivinar a cuál patrón de diseño correspondía.

Acto seguido, se les dio la respuesta correcta y se les solicitó calificar qué tan de acuerdo estaban con que la metáfora era clara y útil, usando una escala Likert de 5 puntos. Los resultados se presentan en la Figura 32.

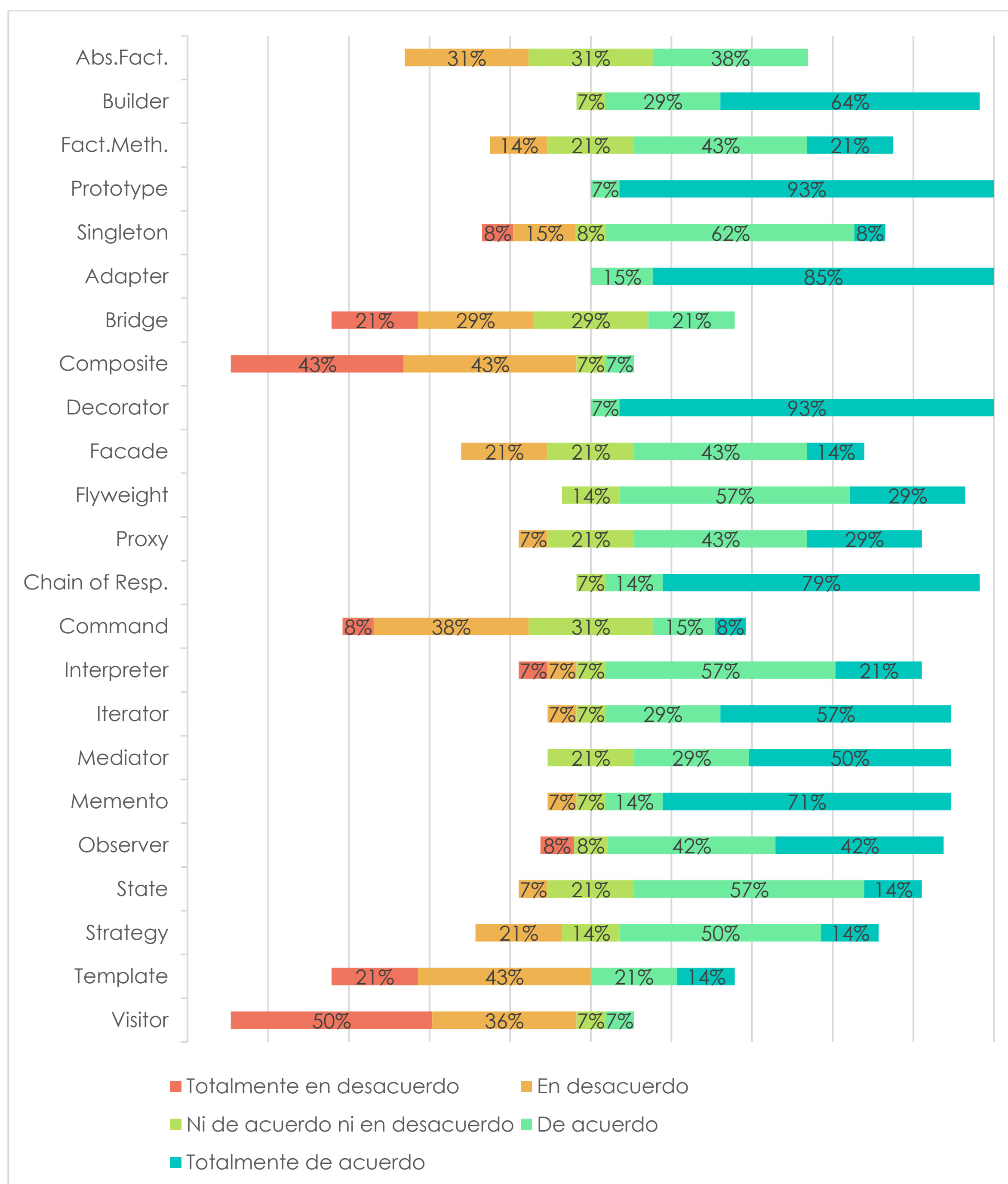


Figura 32. Evaluación de la claridad y utilidad de las analogías.

Las 5 metáforas con calificaciones más altas (de acuerdo o totalmente de acuerdo con su claridad), fueron 2 creacionales, 2 estructurales y 1 de comportamiento:

- Builder (93%)
- Prototype (100%)
- Adapter (100%)
- Decorator (100%)
- Chain of responsibility (93%)

Por otra parte, las 5 metáforas con calificaciones más bajas (totalmente en desacuerdo o en desacuerdo con su claridad) fueron ningún creacional, 2 estructurales y 3 de comportamiento:

- Bridge (50%)
- Composite (86%)
- Command (46%)
- Template method (64%)
- Visitor (86%)

Las metáforas mejor calificadas pueden ser más claras que otras por diferentes motivos. Uno de ellos puede ser la relativa simplicidad del patrón de diseño, que implica también una mayor facilidad en explicarlo o encontrar algún ejemplo sencillo del mundo real. Otro motivo puede ser que su sola descripción implica la mención de alguna palabra clave que identifica bien al patrón de diseño.

Por ejemplo, en el caso de *prototype* se menciona la clonación de células por medio de la mitosis, que hace referencia a un proceso de creación y, más en específico, de copia de algo, que concuerda muy bien con la definición de ese patrón de diseño. En el caso de *chain of responsibility* se habla de una serie de eslabones en una “cadena” para completar un proceso y para *adapter,* se menciona un adaptador de enchufes. *Builder* hace referencia a un proceso de

“construcción” y la metáfora de *decorator* habla de añadir capas y funcionalidad a algo, que puede servir para identificar fácilmente un patrón sencillo como este.

Las metáforas peor calificadas pueden ser poco claras también por diversas razones. Entre ellas que el patrón mismo sea poco usado o popular, porque es muy complejo o porque dicho patrón es, en sí mismo, complicado de ejemplificar por medio situaciones o elementos del mundo real. La metáfora peor calificada es la de *visitor*, que es un patron relativamente complejo, que podría no ser tan usado ni mencionado como otros más simples, es difícil de explicar y difícil de representar.

Template method, por otra parte, es un patrón relativamente sencillo, pero difícil de representar porque su propia definición habla de algoritmos, clases y subclases, que, en conjunto, hacen complicado trasladarlo a un ejemplo con elementos tangibles. *Composite* también es relativamente sencillo, e inclusive menciona algunas palabras clave al decir que se deben tratar objetos simples y complejos de la misma forma. Sin embargo, es posible que ejemplificarlo con operaciones aritméticas haya agregado alguna complejidad a la analogía.

De manera similar a *template method*, el patrón *bridge* resulta bastante difícil de representar, pues es otro que en su definición hace referencia directa a abstracciones e implementaciones complejas de ejemplificar. Por otro lado, la analogía de *command* obtuvo la calificación menos mala de entre las malas, pero no deja de resultar un poco sorpresivo que no fuera tan clara, pues la metáfora se tomó de un ejemplo que parecía bastante evidente y didáctico de (Freeman et al., 2004).

Otra forma de visualizar los resultados totales de la evaluación de las metáforas es agrupando los patrones de diseño por categoría, esto es, en creacionales, estructurales y de comportamiento, y promediando las calificaciones de los patrones en cada categoría, como se presenta en la Figura 33. La categoría con mayores opciones favorables en promedio (de acuerdo o totalmente de acuerdo con su claridad) es la de patrones de diseño creacionales con

un 73%, seguida por la de comportamiento con un 65% y finalmente la de estructurales con un 64%.

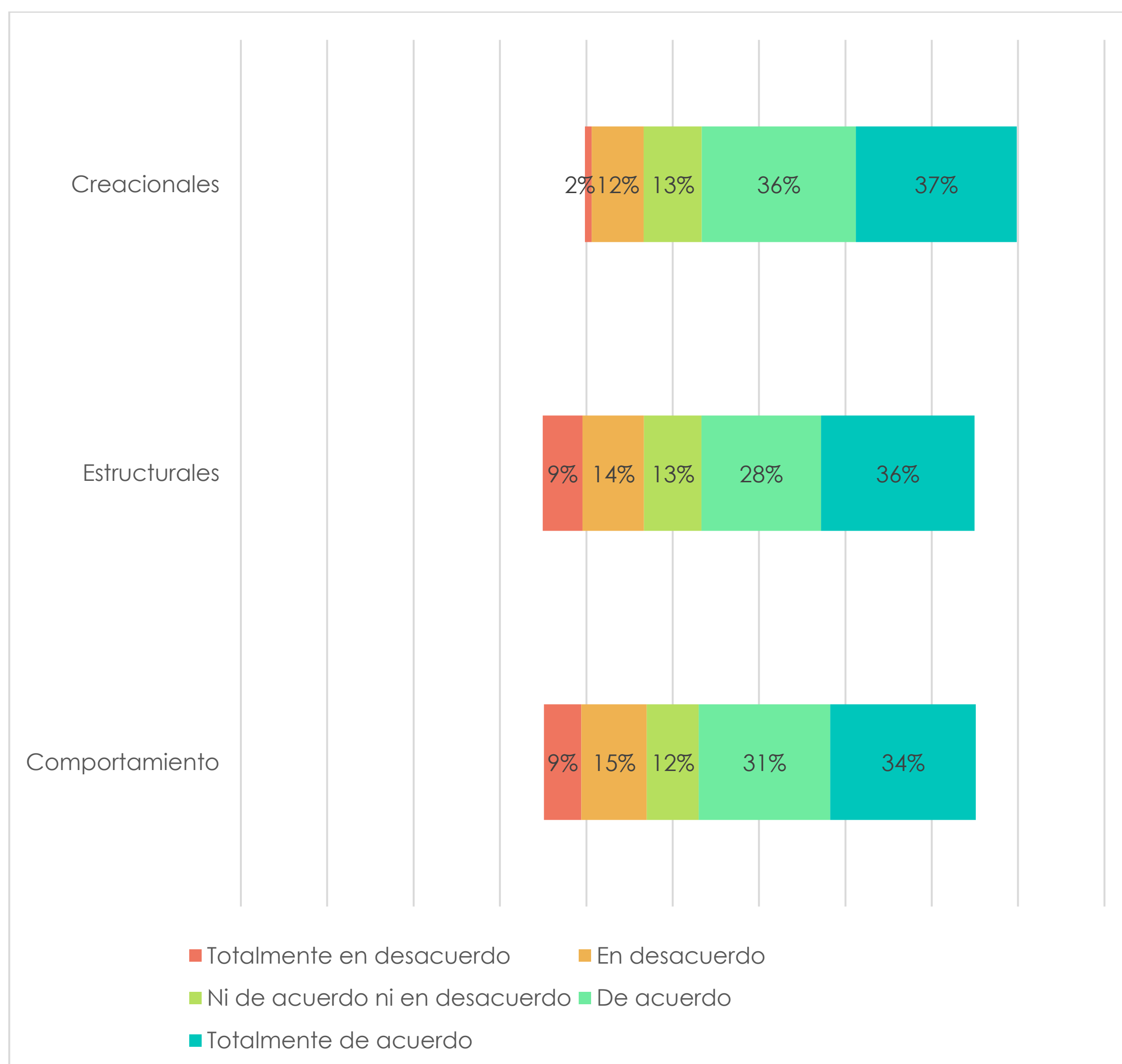


Figura 33. Evaluación de la claridad y utilidad de las analogías por categoría de patrón.

Se está muy cerca de un empate en las opiniones favorables de las metáforas en las categorías de comportamiento y estructurales, pero la de creacionales tiene un promedio mayor claro de opiniones favorables. Esto podría deberse a que, por alguna razón, y cómo se mencionó anteriormente, existe una mayor familiaridad de los estudiantes con los patrones de diseño creacionales o que es más fácil representar este tipo de patrones por medio de analogías con el mundo real. Además, los patrones creacionales suelen estar ligados en mayor grado a una palabra clave que describe su finalidad única, que es crear objetos. Otros patrones como los estructurales y los de comportamiento tienen, en su mayoría, diferentes objetivos, pero tienen también algunas similitudes con otros patrones, lo que puede llevar más fácilmente a confundir uno con el otro.

El conocimiento y experiencia de los estudiantes en análisis, diseño y patrones de diseño orientado a objetos puede haber influido en que algunas metáforas les resultaran más o menos claras. Con el fin de valorar este conocimiento y experiencia, se presentan, en la siguiente sección, los resultados y análisis de la encuesta que se ejecutó con los 16 estudiantes del curso de diseño en tres momentos diferentes de un semestre.

CAPÍTULO 7

Percepción del aprendizaje sobre patrones de diseño

Esta parte de la investigación se efectuó también con los estudiantes del curso CI-0136 Diseño de Software, que se incluye en la carrera de Bachillerato en Computación con varios énfasis de la Escuela de Ciencias de la Computación e Informática en la Universidad de Costa Rica. La primera parte de este capítulo analiza los resultados de una encuesta completada por los estudiantes del curso, que intenta valorar sus conocimientos en análisis y diseño orientado a objetos. La segunda parte, presenta información acerca de las calificaciones finales del curso y la analiza en contraste con la autoevaluación hecha por los estudiantes acerca de cómo autoperciben su nivel de conocimiento en la materia.

Autopercepción del conocimiento de los estudiantes

Se elaboró una encuesta, basada en la realizada por (Lartigue & Chapman, 2018), para obtener una valoración del conocimiento y experiencia de los estudiantes en análisis y diseño orientado a objetos y patrones de diseño, así como algunos datos demográficos. El grupo estaba conformado por 16 estudiantes y la encuesta se ejecutó en tres momentos diferentes del curso: al inicio, a la mitad y al final. Estas tres rondas de ejecución permitieron apreciar el progreso en la autopercepción de su conocimiento a medida que el curso avanzaba.

La primera parte de la encuesta incluía dos preguntas para recabar información demográfica. Se preguntó, en primera instancia, acerca de su experiencia laboral, pero solo un estudiante reportó tenerla y en una modalidad de medio tiempo. Por otra parte, promediando las respuestas de las tres ejecuciones de la encuesta, un 42% de los estudiantes expresó haber trabajado en software de un tamaño máximo de entre 500 y 1000 líneas y un 46% en software de entre 1000 y 2000 líneas. El restante 12% creé haberlo hecho en software de menos de 500 líneas. La segunda

parte del cuestionario estaba totalmente dedicada a valorar el conocimiento y experiencia de los estudiantes trabajando con patrones de diseño.

Resulta interesante contrastar los resultados a través de las tres rondas de aplicación de la encuesta en lo que al conocimiento y experiencia de los estudiantes se refiere. En cada ejecución se les pidió autocalificarse en estas dos áreas, conocimiento y experiencia, usando una escala de 1 a 10 y obteniendo los resultados que se presentan en la Figura 34. En ambas áreas se puede notar un crecimiento en la autopercepción de sus capacidades en paralelo al progreso en el curso.

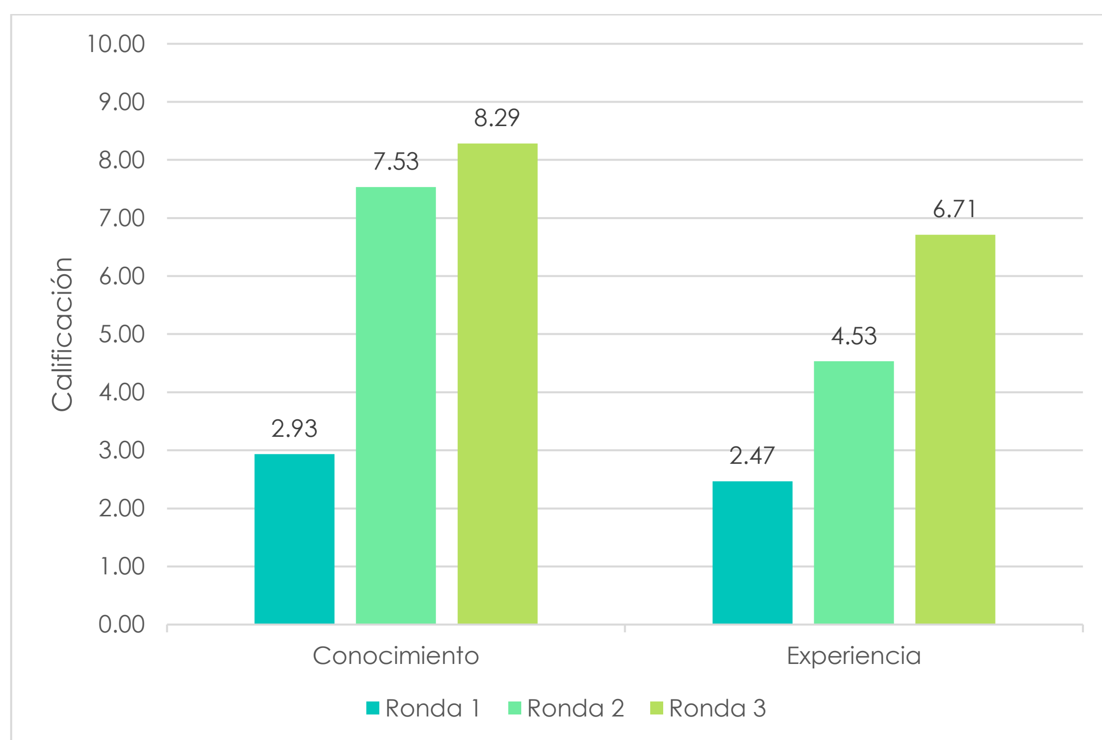


Figura 34. Autocalificación de los estudiantes con respecto a conocimiento y experiencia en patrones de diseño.

Luego, se les preguntó acerca de su opinión con respecto a la dificultad para implementar software utilizando patrones de diseño. Como se puede notar en la Figura 35, el desconocimiento en la materia era alto al inicio, pero se redujo con cada iteración.

El porcentaje de estudiantes que consideraron la dificultad moderadamente fácil subió en cada ronda, pero el de los que la calificaron como moderadamente difícil osciló hacia abajo y hacia arriba, para cerrar con un porcentaje menor en la tercera ronda que en la primera. Algo similar sucedió con la proporción de los que lo consideraron muy difícil, aunque el resultado de la tercera ronda bajo con respecto a la segunda, para cerrar en el mismo valor del 7% que el de quienes lo consideraron muy fácil en esa última ronda.

En las rondas 1 y 2 nadie consideró que fuera muy fácil implementar software utilizando patrones de diseño, lo que sí sucedió, con un 7%, en la número 3. Además, el porcentaje de estudiantes que calificaron la tarea evaluada como moderadamente difícil o muy difícil, descendió de un 67% en las rondas 1 y 2 a un 64% en la ronda 3. Todo esto, en conjunto, muestra un leve ascenso en cada ronda con respecto al conocimiento adquirido y también un leve descenso en la dificultad con la que ven la tarea de implementar software usando patrones de diseño. Conocer mejor los patrones de diseño no necesariamente hace que sea mucho más fácil su aplicación, pero ese conocimiento podría influir en otras áreas.

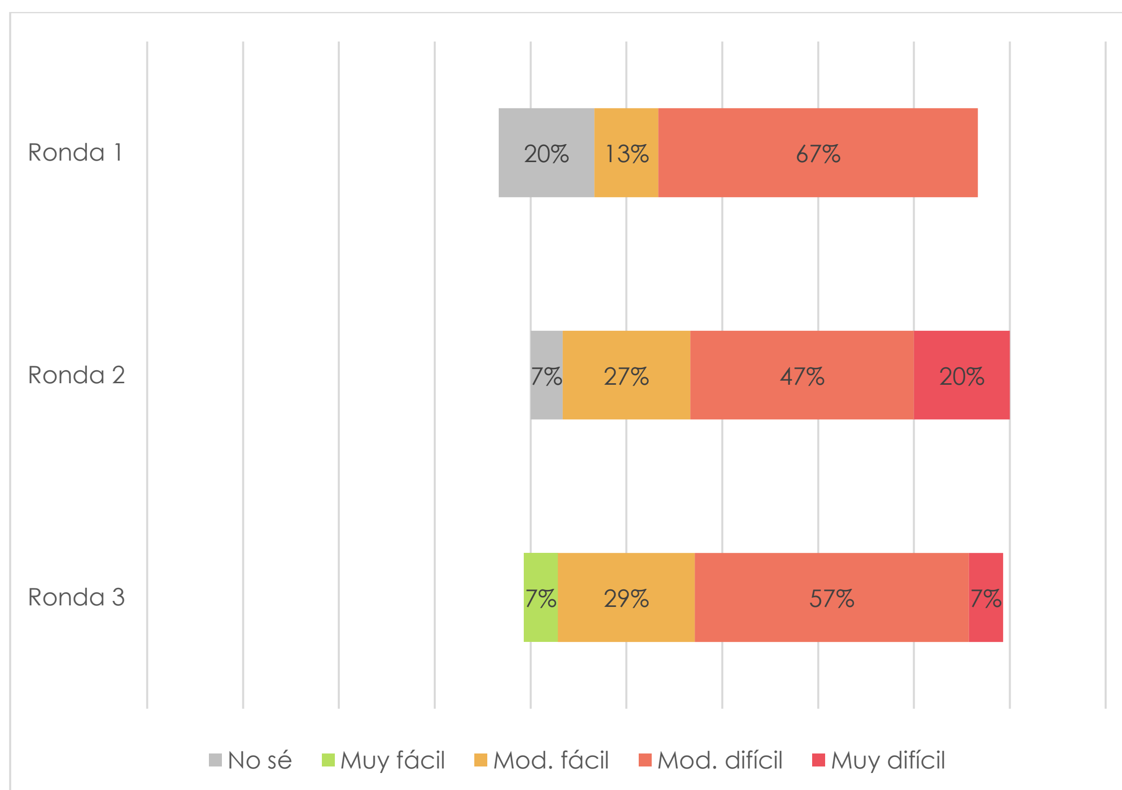


Figura 35. ¿Cree que implementar software usando patrones de diseño es difícil?

Se consultó también a los estudiantes sobre su opinión acerca de si implementar software utilizando patrones de diseño toma más o menos tiempo que al no utilizarlos. En las respuestas a esta pregunta se refleja también el progreso en el conocimiento de los patrones, como se puede apreciar en la Figura 36.

El porcentaje de quienes piensan que toma mucho menos tiempo, aumentó en cada ronda, desde 0%, pasando por 13% y finalizando con 21%, poco más de una quinta parte de los estudiantes. Por otro lado, la proporción de quienes piensan toma algo más de tiempo se mantuvo relativamente estable (alrededor de un 70%, en promedio), mientras que los que creen que toma mucho más tiempo se redujo en cada iteración hasta llegar a un 7% en la última iteración. Este

progreso en las apreciaciones positivas acerca del nivel de dificultad e inversión de tiempo requeridos para aplicar patrones de diseño podría influir en una mayor disposición de los estudiantes a utilizarlos.

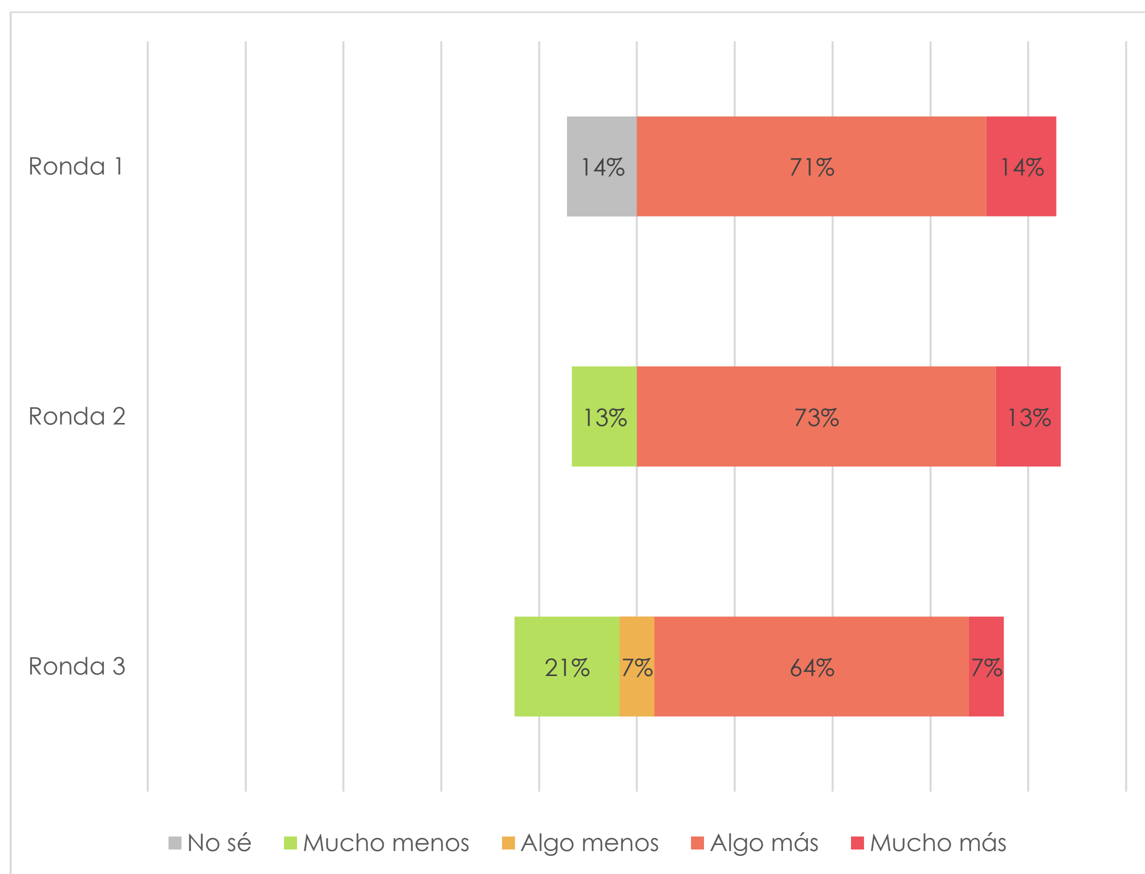


Figura 36. ¿Cree que implementar software usando patrones de diseño toma más o menos tiempo que sin hacerlo?

Considerando lo anterior, se planteó la pregunta: ¿qué tan probable es que intente utilizar patrones de diseño en el software que usted desarrolla? En la Figura 37 se puede notar un ascenso progresivo en la proporción de quienes lo consideran algo o muy probable. El porcentaje de los que lo creen muy improbable o algo improbable cayó a cero en la tercera ronda, mientras

los que consideran algo probable o muy probable conforman el 100% de los estudiantes en esa última iteración. Un 80% lo consideraron muy probable en la segunda ronda, pero ese número cayó 1 punto porcentual para la tercera ronda, aunque fue para engrosar el porcentaje de los que lo consideran algo probable, que es todavía algo positivo.

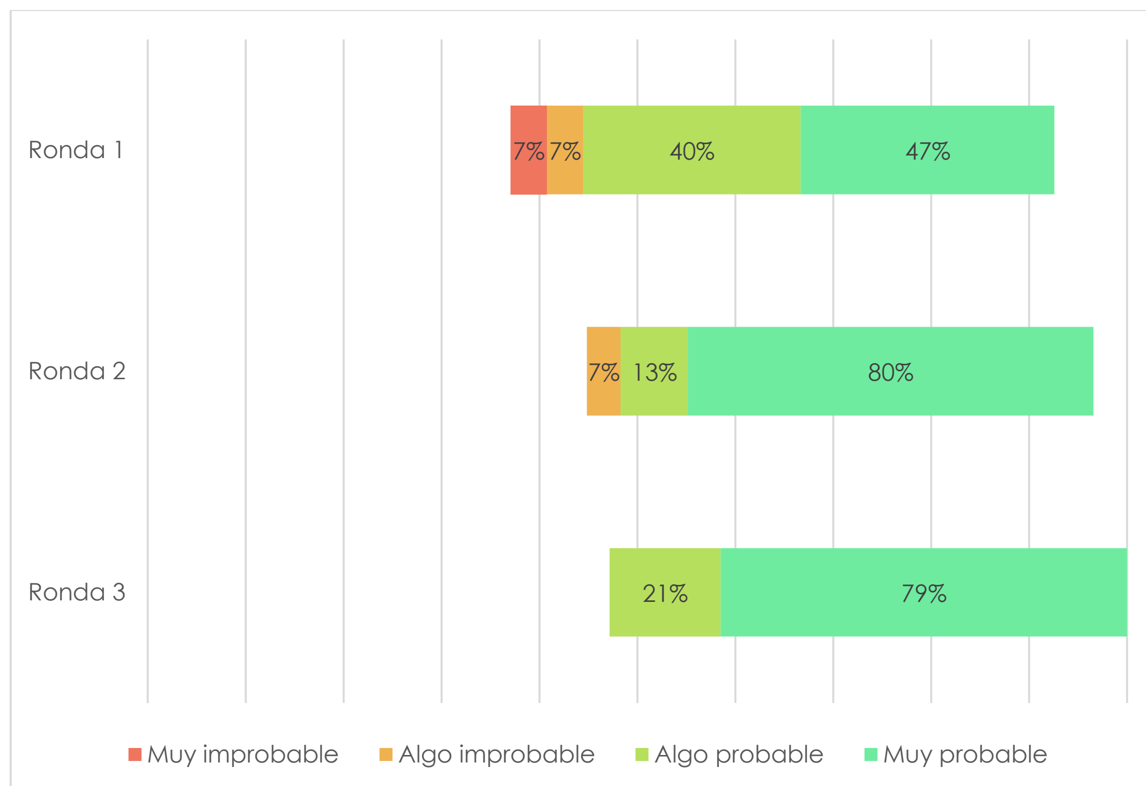


Figura 37. ¿Qué tan probable es que intente utilizar patrones de diseño en el software que usted desarrolla?

La siguiente sección de la encuesta tenía como propósito medir el conocimiento de los estudiantes con respecto a temas generales de análisis y diseño orientado a objetos, como se muestra en la Figura 38. Como se puede apreciar, hubo un crecimiento en el nivel promedio de conocimiento de todos los temas evaluados a través de las tres rondas de la encuesta, pero el de las tres categorías principales de patrones de diseño del GoF creció a partir de un punto casi

nulo. Los estudiantes ya habían estado expuesto a conceptos de análisis y diseño orientado a objetos, más no así a los patrones de diseño básicos y su categorización.

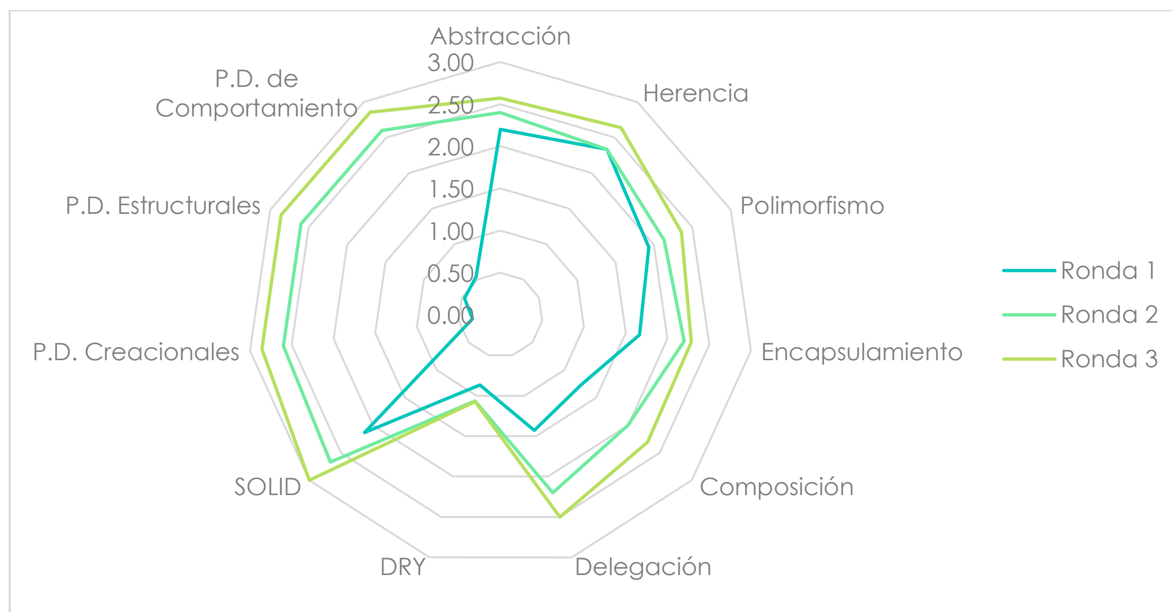


Figura 38. Familiaridad con algunos conceptos de análisis y diseño orientado a objetos.

Como paso final en esta encuesta, se quiso ahondar más en la percepción de su conocimiento en los 23 patrones de patrones del GoF, a través de las tres rondas de aplicación del instrumento. La Figura 39 muestra una progresión en este conocimiento de forma bastante uniforme. Para la primera aplicación había algunas nociones acerca de algunos patrones como *prototype, singleton, proxy, chain of responsibility, iterator y template method*, probablemente por tener nombres un poco más comunes. Para la segunda ronda se dio, en promedio, un avance uniforme en la medida en que los estudiantes creen saber acerca cada patrón de diseño. Para la tercera ronda se mantuvo un aumento en la autopercepción de su conocimiento, a excepción de los casos de *abstract factory* y *bridge*, que bajaron levemente con respecto a su evaluación para la ronda número 2.

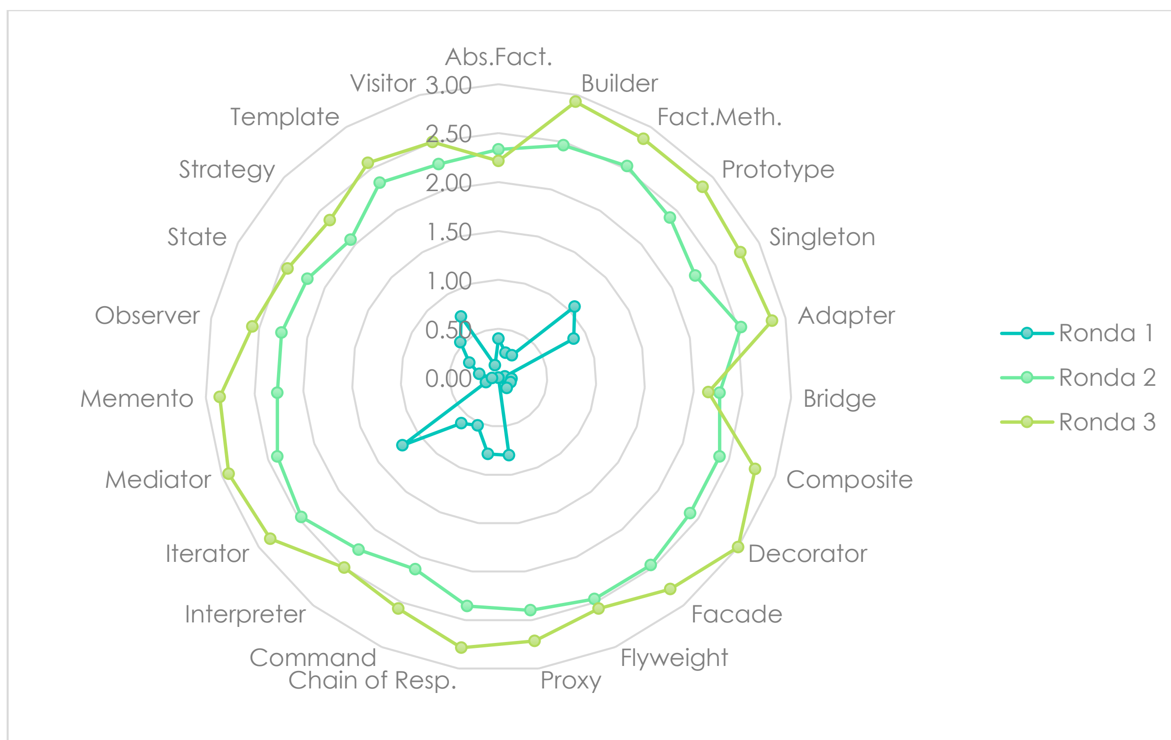


Figura 39. Familiaridad con cada uno de los 23 patrones de diseño del GoF.

Tal como se mencionó anteriormente, los estudiantes reportaron tener algún conocimiento, desde el principio de curso, acerca de 6 patrones de diseño *prototype, singleton, proxy, chain of responsibility, iterator y template method*. Los patrones *prototype* y *chain of* responsibility están entre las 5 metáforas mejor calificadas en la sección de Evaluación de las analogías, y este conocimiento previo podría haber ayudado a identificarlas mejor.

Por otra parte, *abstract factory* y *bridge* son los únicos patrones acerca de los que reportan conocer o entender menos en tercera ronda que en segunda. El patrón *bridge* se encuentra en la lista de patrones peor calificados también de la sección Evaluación de las analogías, lo que parece confirmar que es un patrón complicado de comprender. Mención aparte merece el patrón *template method*, pues los estudiantes parecían tener algunas referencias de él, pero también fue calificado como una analogía poco clara en la sección anterior. A pesar de que, como se

mencionó en esa sección, *bridge* es un patrón difícil de ejemplificar con elementos del mundo real, pues tiene estrecha relación con algoritmos, clases y subclases, podría haber margen de mejora en la definición de su metáfora.

Como parte final del análisis del conocimiento de los estudiantes en patrones de diseño es importante comparar lo que ellos mismo reportan, como se presentó en los párrafos anteriores, contra las calificaciones finales del curso. Esto será de ayuda para confirmar el avance que los estudiantes sienten haber hecho a través de las tres aplicaciones de esta encuesta.

Calificaciones del curso

En la sección anterior, específicamente en la Figura 34, se mostró el promedio de autocalificación de los estudiantes para cada una de las tres rondas de ejecución de la encuesta que se les realizó. En la ronda final, ellos autocalificaron, en promedio, su conocimiento en 8.29 puntos de 10 posibles. Esta calificación no estuvo lejos de la realidad, pues el promedio de calificaciones finales del curso se ubicó en 8.44, también de 10 puntos posibles.

Si se contrastan autocalificaciones y calificaciones reales finales más en detalle, como se muestra en el diagrama de caja en la Figura 40, es posible notar que hay una mayor dispersión en el rango de autocalificaciones que los estudiantes se asignaron que en el de las que obtuvieron. La autocalificación mínima fue de 7 y la máxima de 9, mientras que todas las calificaciones finales estuvieron entre 8 y 9. Por otra parte, ~75% de las autocalificaciones estuvieron entre 8 y 9, pero la totalidad de las calificaciones reales estuvieron dentro de eso mismo rango. Ningún estudiante obtuvo una calificación menor a 8, pero menos de los que se autocalificaron con 9 lograron en realidad ese puntaje.

Aquí se debe tomar en cuenta que se está comparando la calificación final de un curso de todo un semestre, en el que hubo tareas, proyecto y examen en diferentes puntos en el tiempo, contra la autocalificación de únicamente la tercera ronda de la encuesta de los estudiantes. El conocimiento era menor al principio del curso y fue progresando durante el semestre, como

muestra la Figura 34, pero la autocalificación de tercera ronda muestra una valoración solamente en ese momento, mientras que la calificación final del curso toma en cuenta todas las actividades del semestre y sus resultados.

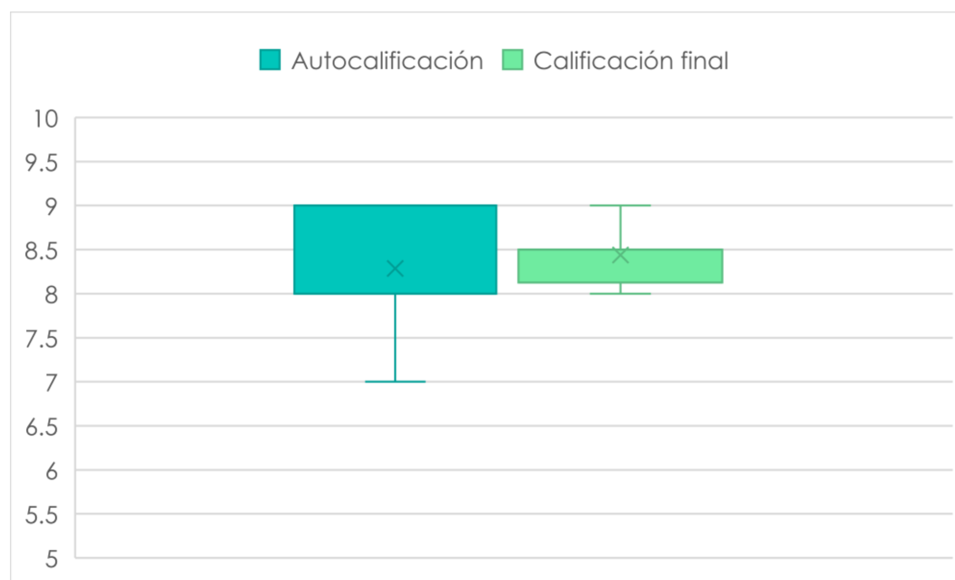


Figura 40. Comparación de autocalificaciones y calificaciones finales.

Ahora bien, con el fin de contrastar algunos de los resultados de la encuesta hecha a los estudiantes, así como lo expuesto en torno a los ejercicios de diseño orientado a objetos, con lo que sucede en la industria, se realizó una encuesta a profesionales en ingeniería de software. Esta encuesta aborda algunos aspectos académicos, que podrían no estar tan frescos para algunos ingenieros, pero se enfoca principalmente en intentar conocer qué consideran útil y relevante acerca de diseño y patrones de diseño orientados a objetos en su práctica profesional. El objetivo es, obtener insumos que puedan ayudar a alinear, en lo posible, lo que se enseña en la academia con lo que se practica en la industria.

CAPÍTULO 8

Uso y relevancia de patrones de diseño en la industria

Como paso siguiente y final en esta investigación, se distribuyó un cuestionario para ser completado por empleados de una empresa de desarrollo de software a la medida y de servicios de *outsourcing* basada en Costa Rica y con filiales en diversos países de América Latina. Fue contestada por un total de 35 ingenieros de software, de los cuales 23 residen en Costa Rica, 5 en Perú, 3 en Colombia, 2 en Bolivia, 1 en Argentina y 1 en México.

La experiencia laboral de los encuestados varía en un rango desde los 3 a los 40 años, con una mediada de 13 años, como se muestra en la Figura 41. Aproximadamente, un 25% de los entrevistados tiene entre 3 y 8 años de experiencia y un 50% tiene entre 8 y 20, mientras que el otro 25% restante tiene más de 20 años de experiencia laboral. Uno de los ingenieros reportó tener 40 años de experiencia laboral, pero esto es considerado como un valor atípico en el gráfico. Con base en toda esta información es posible considerar que cerca de un 75% de la muestra puede ser considerada como de amplia experiencia profesional.

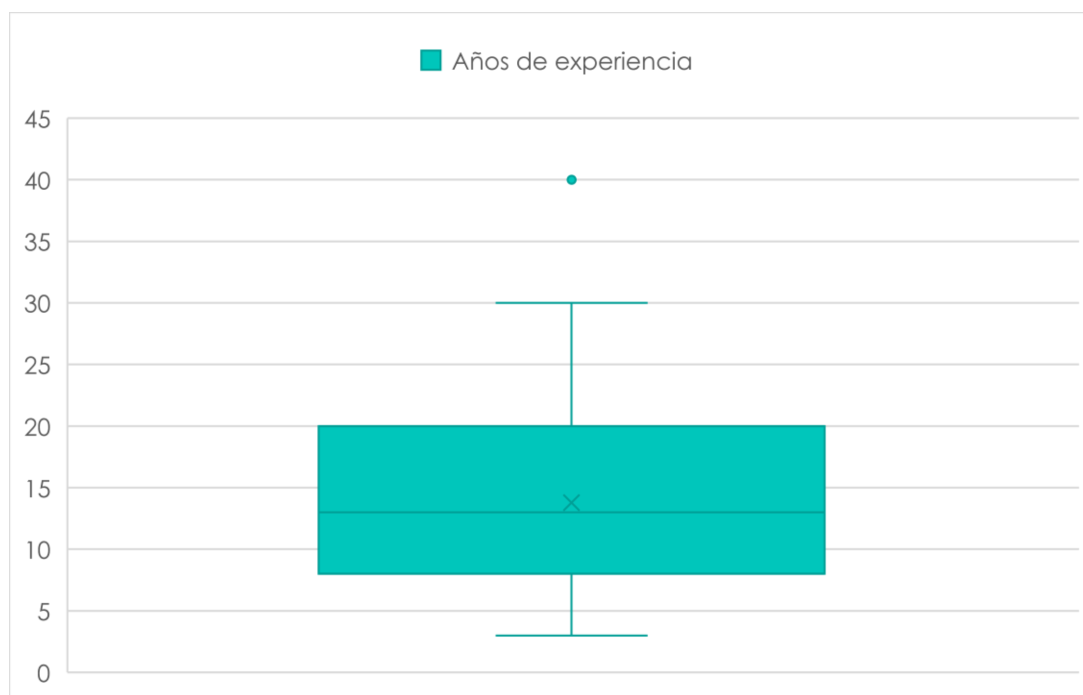


Figura 41. Años de experiencia laboral.

Dos de las primeras preguntas en el cuestionario estaban dirigidas a conocer, qué importancia tiene para los ingenieros, el conocimiento y experiencia en el uso de patrones de diseño orientados a objetos (ver Figura 42). Ninguno de los entrevistados considera que alguna de estas dos áreas carezca de importancia, sino todo lo contrario, una amplia mayoría considera estas capacidades como importantes o muy importantes. Parecen considerar el conocimiento como más importante que la experiencia, pero ambas van de la mano.

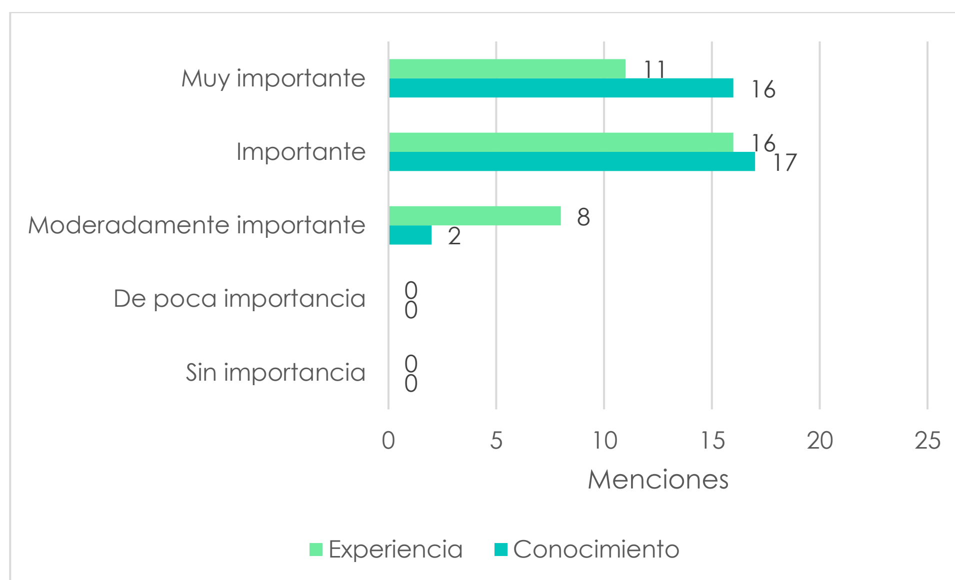


Figura 42. Importancia del conocimiento y experiencia en patrones de diseño.

De forma similar, se les pidió calificar sus propios conocimientos y experiencia en patrones de diseño utilizando una escala de 1 a 5, donde 1 representaba la calificación más baja y 5 la más alta. Los resultados muestran que los ingenieros creen tener, casi en su totalidad, el mismo nivel en cada una de estas dos áreas, como se muestra en la Figura 43. Ninguno cree tener un nivel de 1, el mínimo, y pocos se autocalifican con un 2, así como pocos también se califican con un 5, el máximo, pero la mayoría cree tener un conocimiento y experiencia avanzada en patrones de diseño.

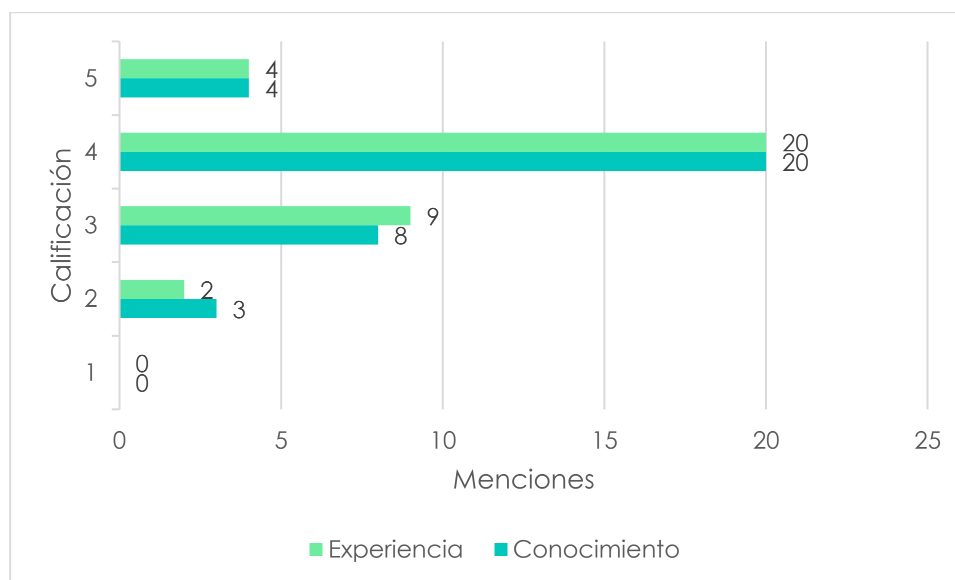


Figura 43. Nivel de conocimientos y experiencia en patrones de diseño de los entrevistados.

Los estudiantes del curso respondieron a preguntas similares con respecto a su conocimiento y experiencia en patrones de diseño en la sección Autopercepción del conocimiento de los estudiantes. En cuanto a conocimiento, al comparar respuestas de ambos grupos se puede observar una menor dispersión en el rango de respuestas de los estudiantes, que se autocalifican, en la tercera ronda de su encuesta, entre 7 y 9, como se observa en la Figura 44. Por otra parte, los ingenieros se califican de una forma más conservadora que los estudiantes, en un rango entre 4 y 10. Al menos un 75% los estudiantes creen estar entre 8 y 9 en lo que respecta a sus conocimientos en patrones de diseño, mientras que sólo cerca del 25% de los ingenieros cree superar la calificación de 8 y un 50% se sitúa entre 6 y 8.

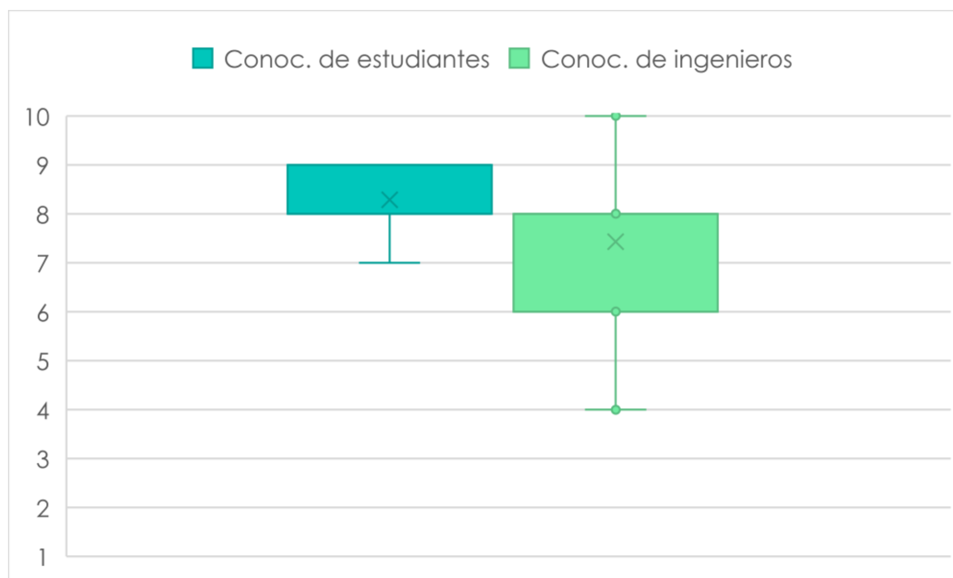


Figura 44. Conocimiento en patrones de diseño de estudiantes e ingenieros.

Con respecto a la experiencia trabajando con patrones de diseño que reportan estudiantes e ingenieros, se puede observar una comparación de los datos obtenidos en la Figura 45. Un 50% de los estudiantes, así como un 50% de los ingenieros, se ubica entre las calificaciones de 6 y 8. Ambos grupos reportan un puntaje mínimo de 4, pero el máximo que reportan los estudiantes es de 8, mientras que al menos un 25% de los ingenieros se califican entre 8 y 10. La experiencia reportada por algunos ingenieros tiene valores en rangos mayores que la de los estudiantes, lo cual tiene sentido. Además, es interesante notar como los ingenieros reportan su conocimiento y experiencia de forma casi equivalente y con una mediana de 8 en ambas mediciones.

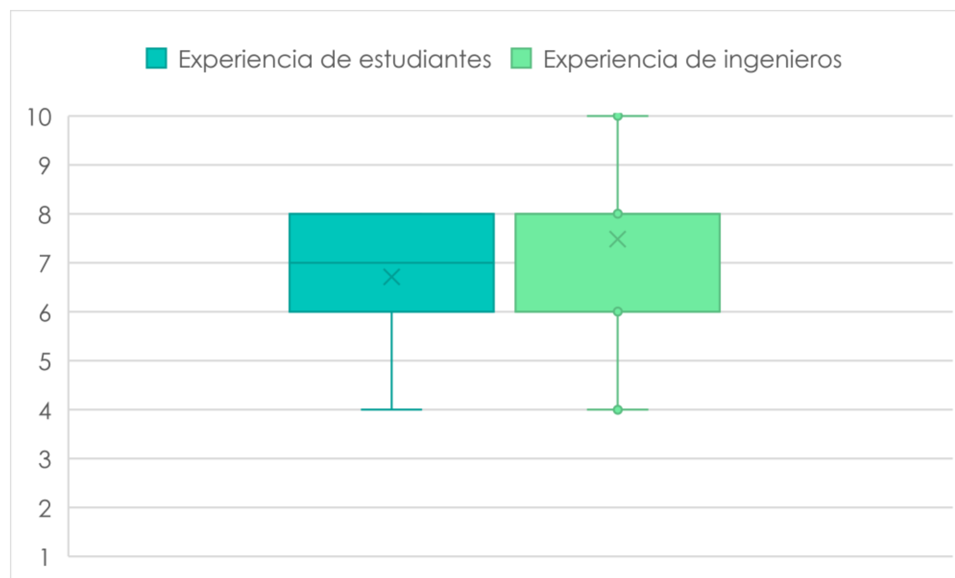


Figura 45. Experiencia en patrones de diseño de estudiantes e ingenieros.

Para confirmar estas autocalificaciones positivas de los profesionales, se les preguntó si conocen el catálogo de patrones de diseño del GoF, a lo que solo 2 de 35 personas respondieron de forma negativa. En contraste, se esperaría que, dado el alto nivel de conocimiento y experiencia que reportan, también conocieran otras listas o catálogos de patrones de diseño. No obstante, solo 12 dicen hacerlo, mientras que 23 reportan no conocer algún otro dominio de patrones de diseño. Además, se les pidió nombrar los otros catálogos de patrones de diseño que conocen, pero las respuestas no fueron tan congruentes con sus autocalificaciones en la materia.

Algunos de los entrevistados mencionan como catálogos de patrones de diseño algunos relativos a plataformas o herramientas específicas, como Kubernetes o React, pero son pocas las respuestas que se enfocan en listas o dominios más generales. En otros casos mencionan patrones específicos como MVC (*Model View Controller*), MVP (*Model-View-Presenter*), MVVM (*Model-View-ViewModel*). También, se citan principios de diseño como KISS (*Keep It Simple, Stupid*) o DRY (*Don't Repeat Yourself*), que no son ni catálogos ni patrones de diseño en sí

mismos. Las pocas menciones a algunos catálogos de diseño, aparte del del GoF, se pueden apreciar en la Figura 46.

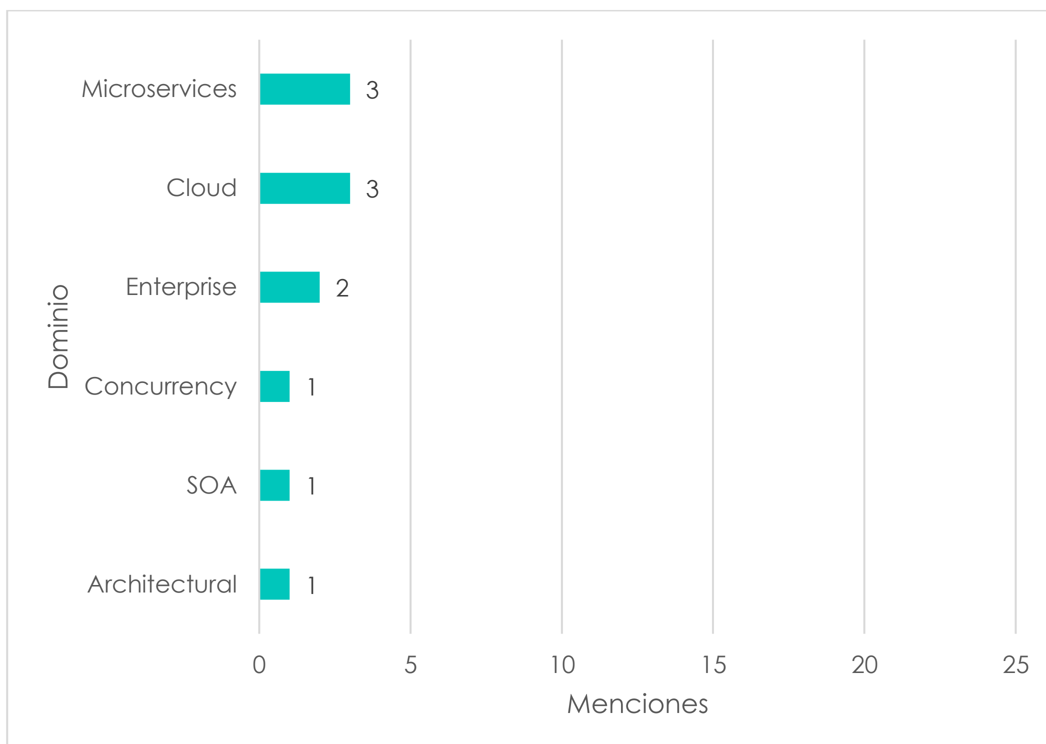


Figura 46. Conocimiento de otros catálogos de patrones de diseño además del GoF.

De vuelta al tema de patrones de diseño específicos, se preguntó cuáles son los que usan con mayor frecuencia. Los resultados muestran que la gran mayoría de los patrones que los ingenieros dicen usar son, en efecto los del GoF, y solo hay unas pocas menciones a otros patrones de otros catálogos como MVC o *Repository*, por ejemplo (Figura 47). Llama atención el gran número de menciones que hay para *singleton*, pero también el hecho que los tres primeros lugares son patrones de diseño creacionales. En la sección Evaluación de las analogías se expuso que los patrones de diseño creacionales podrían ser los más naturales o fáciles de

reconocer para los estudiantes, pero más allá de eso, aquí queda en evidencia que también son de los más usados en la práctica profesional.

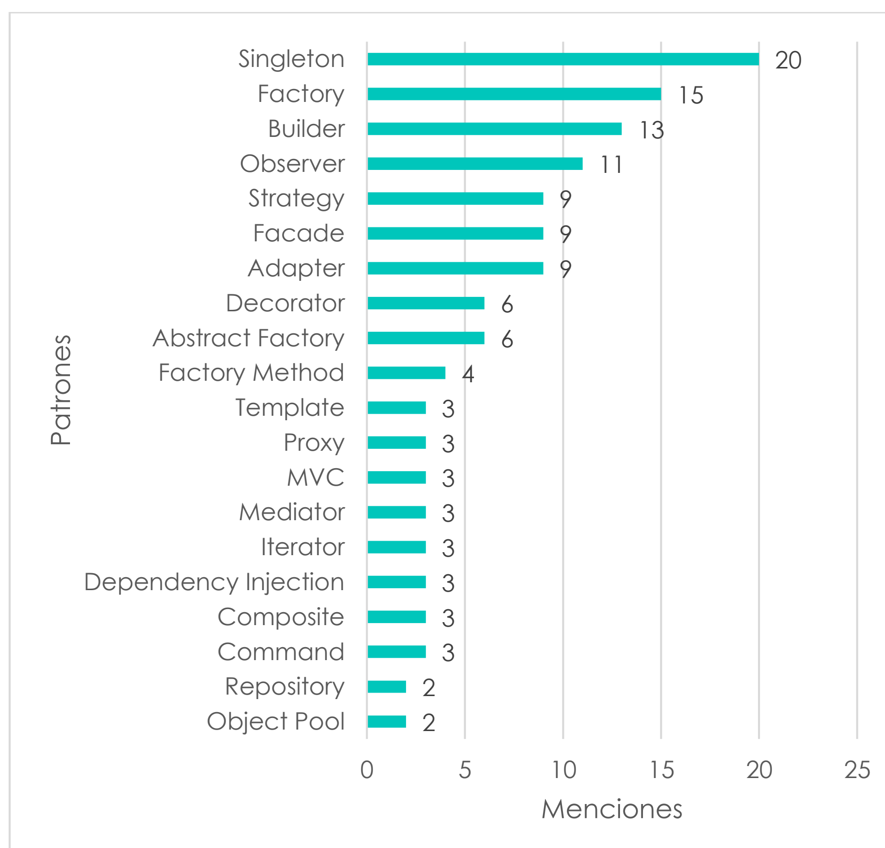


Figura 47. Patrones de diseño usados con mayor frecuencia.

Se preguntó también a los ingenieros, acerca de los patrones o estilos arquitecturales que más utilizan en los proyectos en los que trabajan. Las respuestas incluyen una amplia variedad de patrones y estilos que ya se habían mencionado en respuestas a preguntas anteriores de catálogos y patrones específicos utilizados, como se muestra en la Figura 48. Las menciones a MVC siguen siendo de las más altas y, además, en este caso, se citan también las arquitecturas *n-tier* y los microservicios como parte de las tres primeras posiciones.

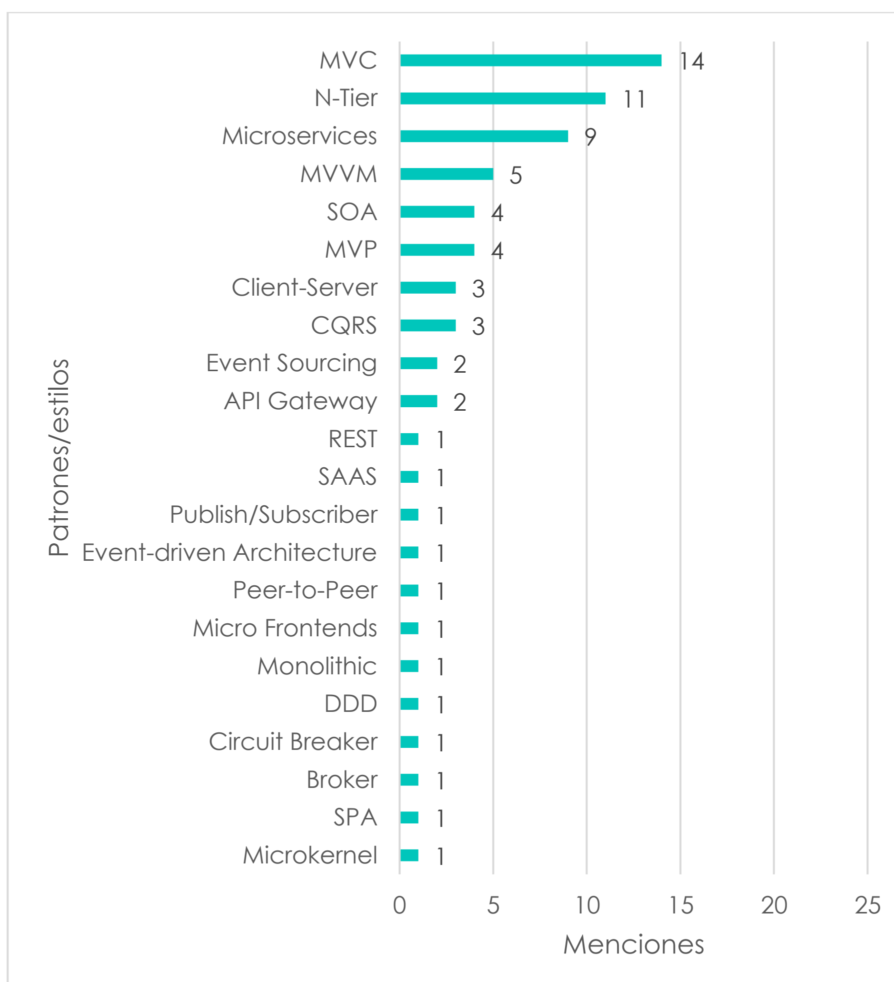


Figura 48. Estilos y patrones arquitecturales más utilizados.

Ahora bien, los patrones de diseño orientado a objetos pueden ser descritos en lenguaje natural, así como por medio de diagramas o ejemplos, como ya se ha visto en secciones anteriores, pero deben ser implementados en algún lenguaje que soporte este paradigma. Se quiso conocer cuáles son esos lenguajes orientados a objetos que más utilizan los entrevistados y para eso se les dio a escoger múltiples opciones de una lista. Los lenguajes en la lista se escogieron en base

a su popularidad según índices como (TIOBE Software BV, 2021) y (Zapponi, 2021). Los resultados se muestran en la Tabla 3.

Tabla 3. Lenguajes orientados a objetos más utilizados.

Lenguaje	*Menciones*
Java	30
JavaScript	29
C#	20
TypeScript	15
Python	14
C++	14
PHP	12
Go	7
Ruby	7

Además, se quiso conocer qué tan difícil creen los entrevistados que es aplicar patrones de diseño en los lenguajes con los que trabajan. Para este efecto, se muestran en la Figura 49 sus opiniones con respecto a los cuatro lenguajes que más utilizan. Se puede observar cómo Java y C# son los lenguajes que más votos favorables tienen en las categorías de muy fácil y fácil, además del hecho de que nadie considera que sea muy difícil aplicar patrones de diseño en ninguno de estos lenguajes.

Vale la pena resaltar que JavaScript es el lenguaje que más veces es mencionado como difícil para aplicar patrones de diseño. Esto puede deberse a que JavaScript es un lenguaje orientado a objetos basado en prototipos en lugar de clases, como lo son, por ejemplo, Java, C# y C++. Esta característica hace que la herencia y las jerarquías de objetos, parte esencial de muchos

de los patrones, funcionen de una forma un poco diferente en JavaScript y no tan clara para todos los desarrolladores (MDN, 2021).

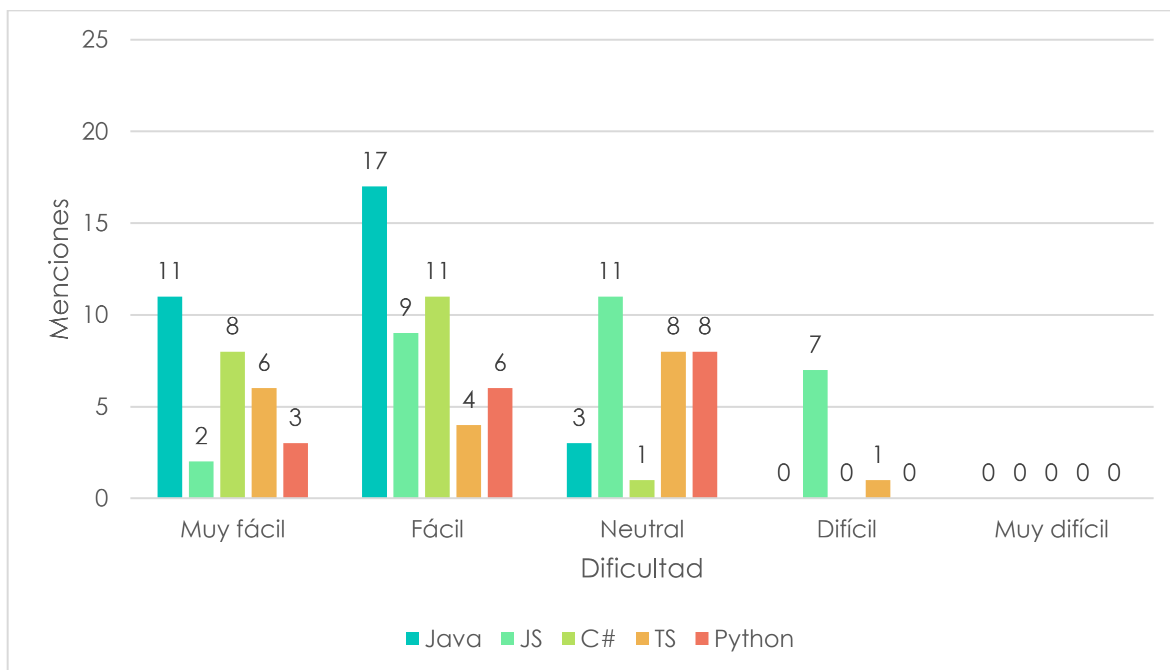


Figura 49. Dificultad de aplicar patrones de diseño en los lenguajes más utilizados.

Continuando con el tema de la dificultad que los ingenieros perciben que tiene aplicar patrones de diseño en el software que diseñan e implementan, se les plantearon dos preguntas similares a unas que se les hicieron a los estudiantes en su respectivo cuestionario y que se documentaron en la sección Autopercepción del conocimiento de los estudiantes. Estas preguntas son: ¿cree que implementar software usando patrones de diseño es difícil? y ¿cree que implementar software usando patrones de diseño toma más tiempo que sin hacerlo? Las respuestas se muestran en Figura 50 y Figura 51.

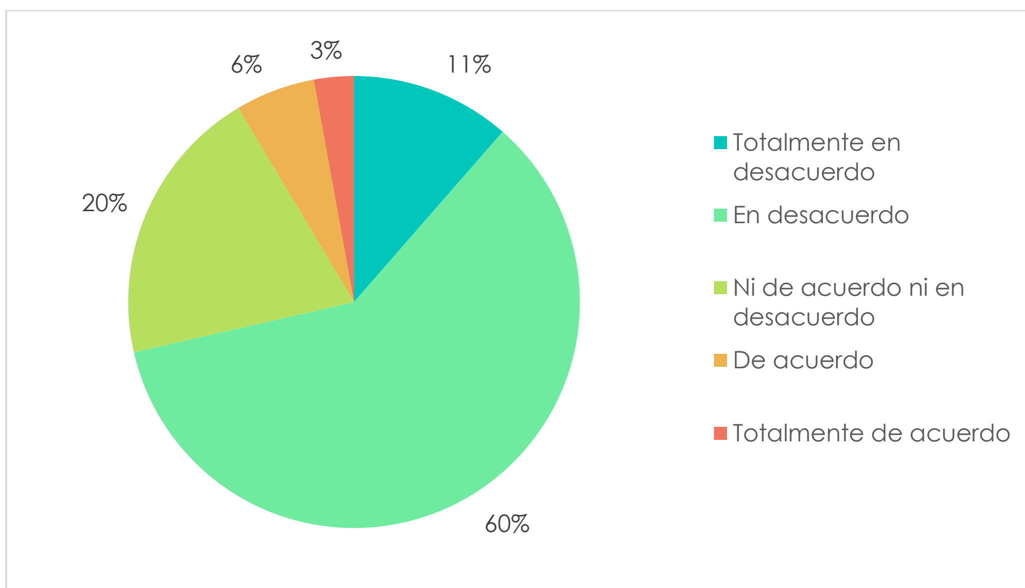


Figura 50. ¿Cree que implementar software usando patrones de diseño es difícil?

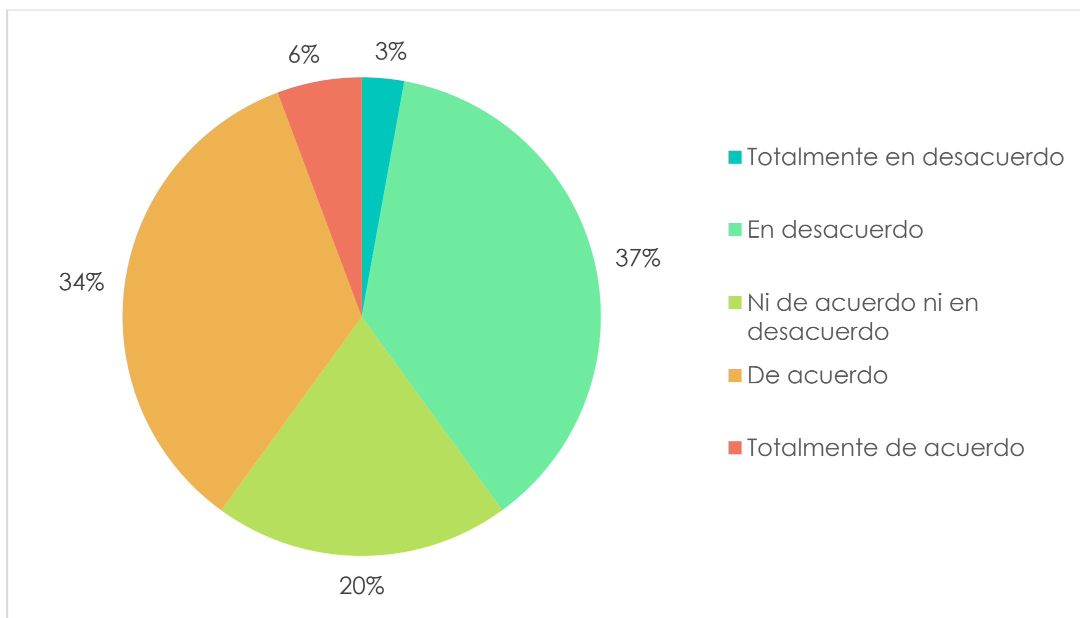


Figura 51. ¿Cree que implementar software usando patrones de diseño toma más tiempo que sin hacerlo?

Solo un 9% de los ingenieros está de acuerdo o totalmente de acuerdo con la afirmación de que utilizar patrones de diseño es difícil. Un 71% de ellos están en desacuerdo o total desacuerdo con que sea una tarea difícil. Por otra parte, en la sección Autopercepción del conocimiento de los estudiantes, estos respondieron a esta misma pregunta, donde un 64% afirmó, en tercera ronda, que implementar software utilizando patrones de diseño es moderadamente difícil o muy difícil. El contraste es claro entre desarrolladores novatos y experimentados en lo que a la percepción de la dificultad de esta práctica se refiere.

Por otra parte, un 40% de los profesionales está de acuerdo o totalmente con que utilizar patrones de diseño en el desarrollo de software toma más tiempo que no hacerlo. Exactamente la misma proporción de la muestra, otro 40%, está en desacuerdo o total desacuerdo con que este sea el caso. Volviendo nuevamente al caso de los estudiantes analizado en la sección Autopercepción del conocimiento de los estudiantes, en tercera ronda, un 29% de los estudiantes manifestó creer que utilizar patrones de diseño toma algo menos o mucho menos tiempo que no hacerlo, mientras que un 71% consideró que tomaba algo más o mucho más tiempo. La respuesta de los profesionales parece estar dividida en partes igual hacia cada extremo, mientras que la falta de experiencia de los estudiantes parece inclinar la balanza hacia la percepción de que sí consume más tiempo.

Es importante contrastar las respuestas a las dos preguntas anteriores. Utilizar patrones de diseño parece no ser tan difícil, al menos para los profesionales, pero no hay una respuesta clara acerca de si toma más tiempo hacerlo. Sin embargo, hay una percepción claramente positiva acerca de los beneficios que acarrea el uso de los patones de diseño (Beck et al., 1995; McLaughlin et al., 2006). El posible tiempo extra invertido parece valer bien la pena. Como sugiere la Figura 52, la percepción de que se está de acuerdo o totalmente de acuerdo con los posibles beneficios que fueron citados, supera el 89% en cuatro casos de seis.

Los dos que no superan ese umbral son el relativo a que ayudan a programadores novatos a producir mejores diseños de forma más ágil y el que tiene que ver con capturar partes esenciales de un diseño para fines como la documentación. El porcentaje de opiniones en desacuerdo o totalmente en desacuerdo con estas afirmaciones es de solamente un 15% y un 6% respectivamente. Por otra parte, un 60% y 74% respectivamente están de acuerdo o totalmente de acuerdo con ellas, pero las posiciones neutras son más de una quinta parte en cada caso. Los ingenieros consultados creen que el uso de patrones de diseño es favorable para desarrolladores novatos, pero perciben este beneficio como menor a los otros analizados.

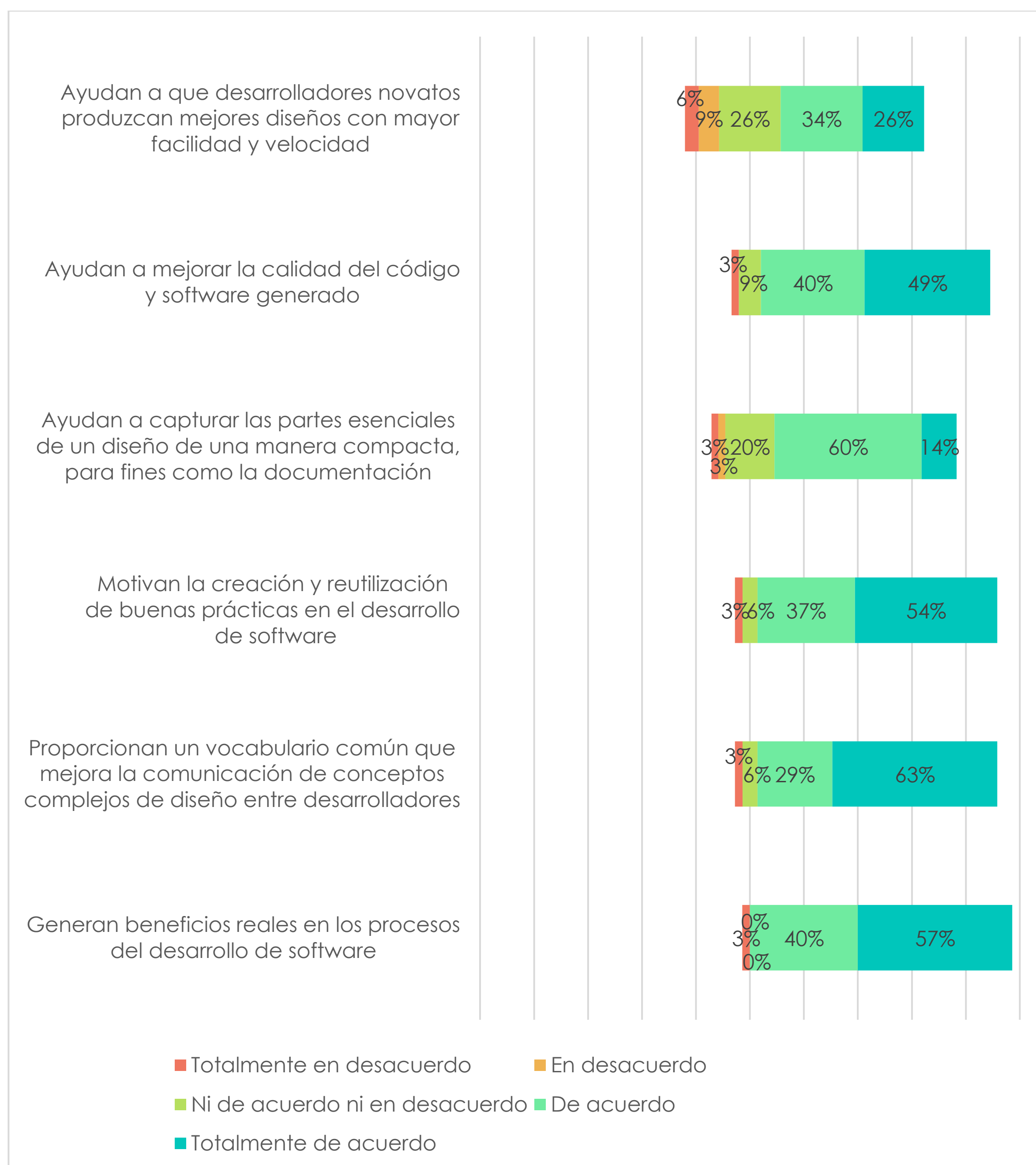


Figura 52. ¿Qué tan de acuerdo estas con estas afirmaciones?

Otro conjunto de preguntas que se formularon a los ingenieros de software tiene que ver con la frecuencia con la que realizan determinadas actividades relacionadas con patrones de diseño. Se les preguntó cuáles categorías de patrones de diseño utilizaban con mayor frecuencia: creacionales, estructurales o de comportamiento. Como se puede ver en la Figura 53, los patrones de diseño creacionales son nuevamente confirmados como los más populares y al contrario que los estructurales y de comportamiento, nadie dice que no los use nunca o casi nunca. Si se toma un segundo lugar con base en las frecuencias de uso más altas de casi siempre y siempre, los patrones estructurales ocupan ese segundo lugar y los de comportamiento el tercero.

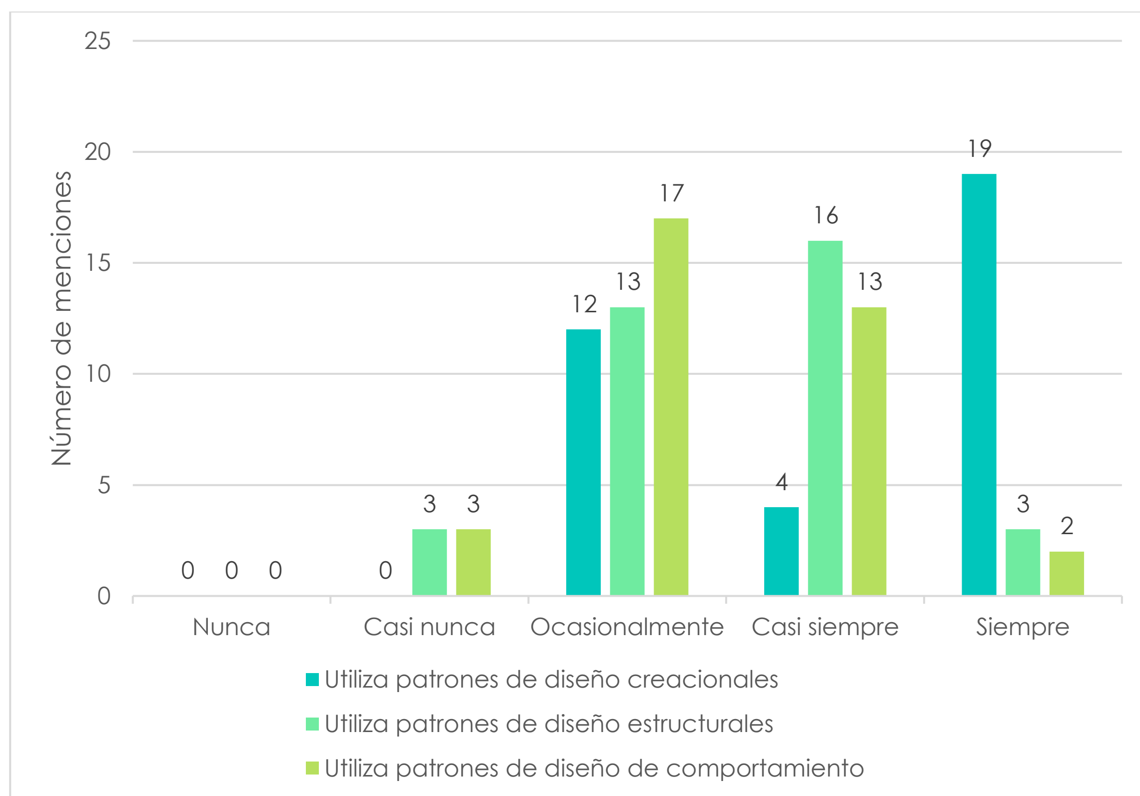


Figura 53. Frecuencia de utilización de patrones de cada categoría.

También se les consultó con qué frecuencia reconocían patrones de diseño en código existente sobre el que trabajaban. Ninguno dijo no hacerlo nunca, solo tres dijeron que casi nunca, mientras que 32 ingenieros los reconocen ocasionalmente, siempre o casi siempre, como se puede observar en la Figura 54. Estos números tan altos pueden tener relación con la experiencia de los miembros de la muestra, que, como ya se mencionó, está en un nivel alto.

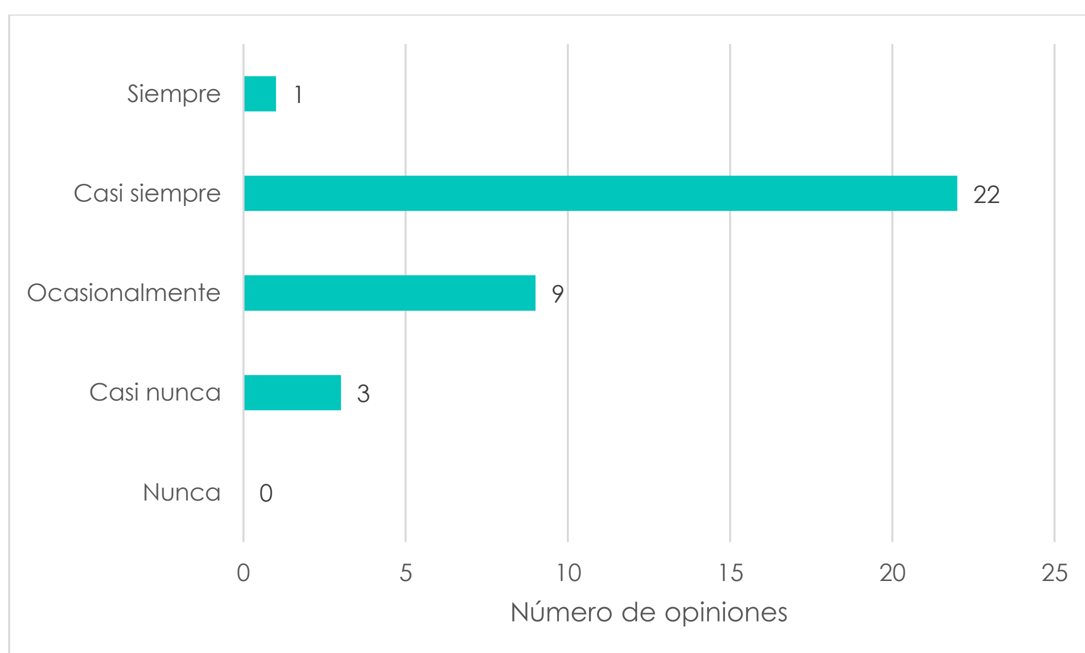


Figura 54. Frecuencia con que reconocen patrones de diseño en código existente.

Siendo este proceso de reconocimiento de patrones tan exitoso, resulta interesante conocer con qué frecuencia creen que los patrones que identifican están mal implementados. Los resultados en la Figura 55 muestran una distribución normal, siendo ocasionalmente la respuesta más popular

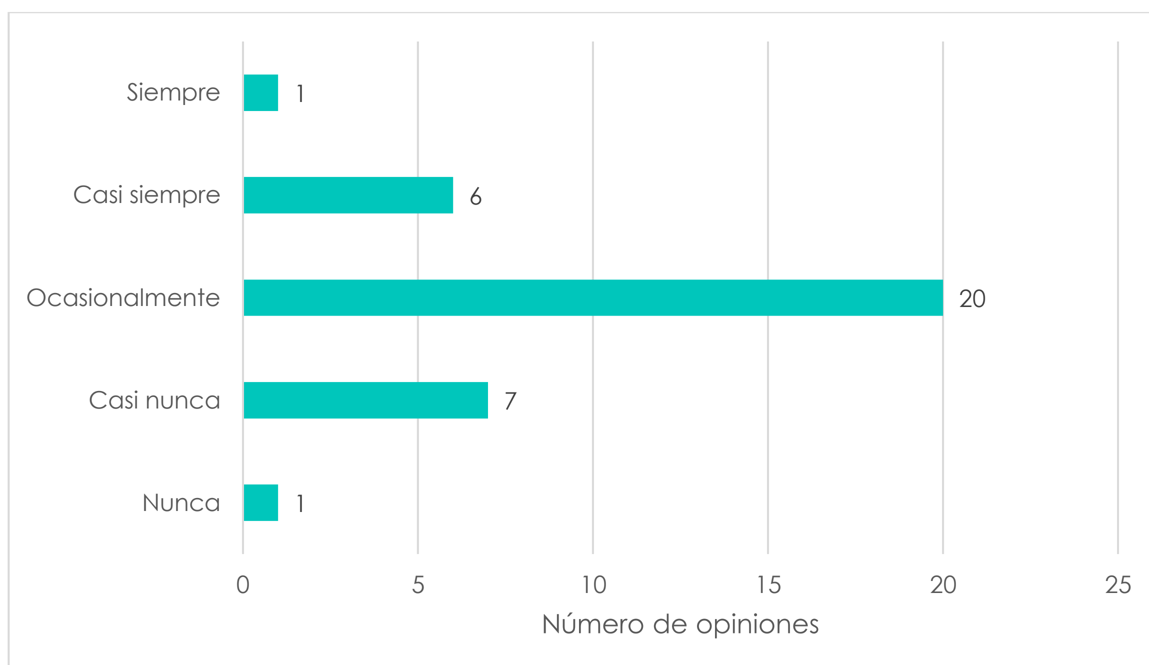


Figura 55. Frecuencia con que encuentran patrones de diseño mal implementados.

Una posible causa por la que un patrón podría parecer estar mal implementado, o podría efectivamente estarlo, es porque los patrones de diseño suelen requerir modificaciones para acomodarse a situaciones o necesidades específicas y estas adaptaciones podrían acarrear errores. En la Figura 56 se puede observar que 24 de los 35 desarrolladores manifestaron tener que hacer estas adaptaciones ocasionalmente o casi siempre.

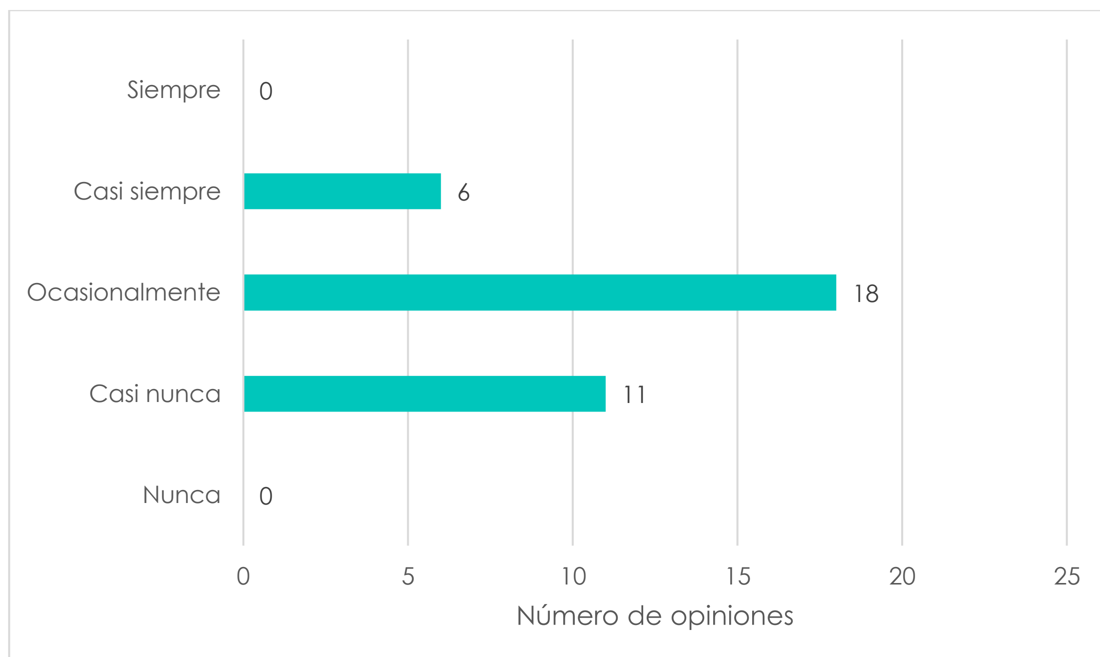


Figura 56. Frecuencia con que encuentran la necesidad de modificar o adaptar un patrón para sus propias necesidades.

Ahora bien, los patrones de diseño pueden ser introducidos al iniciar un proceso de diseño o programación o al hacer refactorización de código. Esto también puede ir de la mano del tipo de metodología usada para la gestión de un proyecto. En enfoques predictivos, como el de cascada, hay fases claras y grandes de análisis, diseño, programación y pruebas y cada una comienza cuando termina la anterior (Sliger & Consulting, 2008). Esto hace que sea necesaria una fase de diseño completa para un proyecto dado y de identificarse la necesidad de introducir patrones de diseño, probablemente se haría tempranamente desde esta fase.

Por otra parte, cuando se utilizan metodologías ágiles, aún existen las fases mencionadas, pero en intervalos de tiempo mucho más reducidos y en los que se desarrolla de forma iterativa (Sliger & Consulting, 2008). Dada la mayor prevalencia de enfoques ágiles en tiempos recientes y en específico en la empresa donde trabajan los ingenieros entrevistados, puede esperarse que los

patrones de diseño sean mayormente introducidos al hacer refactorización en medio de un proceso incremental e iterativo.

Además, como mencionan (Freeman et al., 2004), el tiempo de la refactorización es el tiempo de los patrones de diseño. No obstante, la Figura 57 muestra las respuestas a la pregunta del momento en que se introducen patrones de diseño: al inicio del trabajo para la resolución de un problema o al refactorizar el código propio o el de otros y la opción de refactorización obtiene apenas una ligera ventaja.

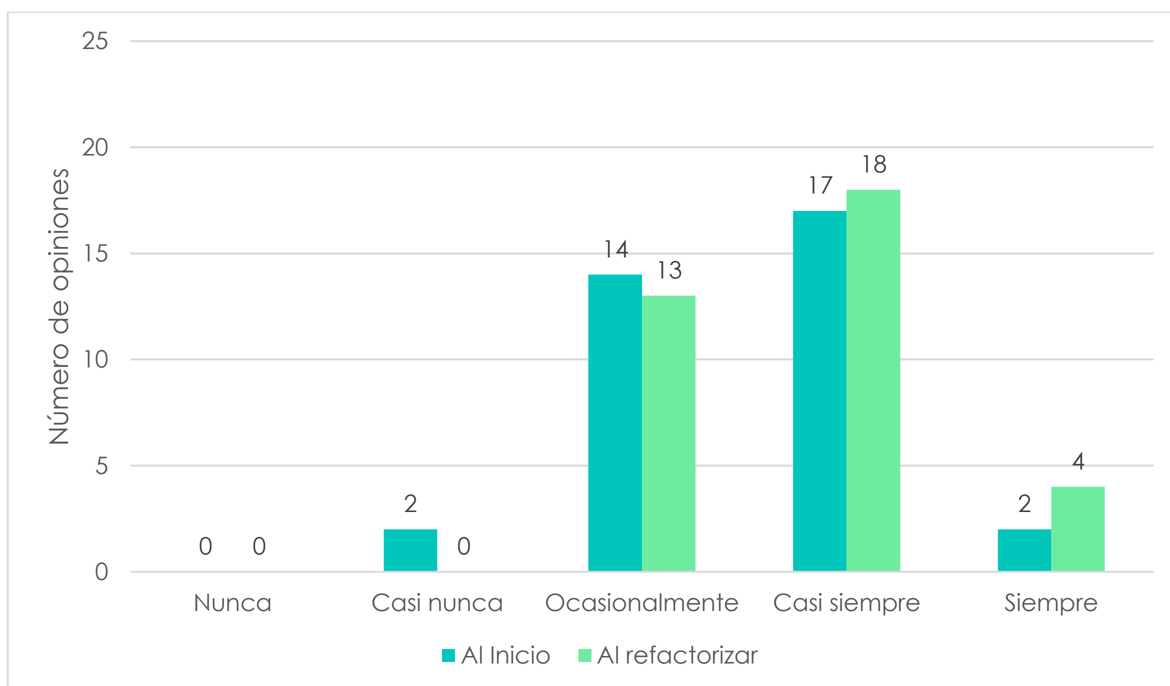


Figura 57. Momento en el que se introducen patrones de diseño.

La última pregunta del cuestionario contestado por los profesionales en ingeniería de software fue: ¿qué tan de acuerdo está con la siguiente afirmación: hoy en día es menos importante el conocimiento y experiencia en programación orientada a objetos debido al auge de otros

paradigmas como la programación funcional? Esta pregunta se formuló pensando en el auge que tiene la programación funcional en aplicaciones del "mundo real" en años recientes (Hinsen, 2009). Tanto es así, que lenguajes originalmente orientados a objetos, como Java y C# agregaron capacidades de programación funcional hace poco más de 10 años (Alic, Omanovic, & Giedrimas, 2016).

Sin embargo, los resultados en la Figura 58 muestran que la mayoría de los entrevistados están en desacuerdo o totalmente en desacuerdo con la aseveración de que el conocimiento y experiencia en programación orientada a objetos sea menos importante hoy en día frente a otros paradigmas como la programación funcional.

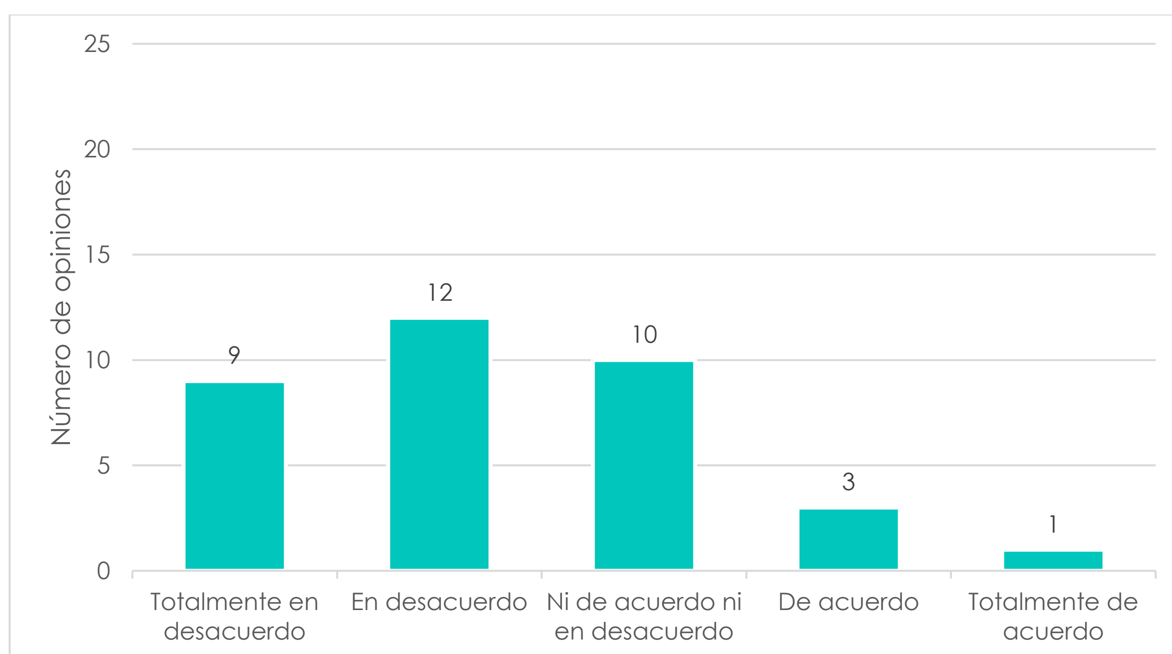


Figura 58. ¿Es hoy más menos importante la programación orientada a objetos?

Finalmente, como se mencionó al inicio de este capítulo, el rango de años de experiencia de los ingenieros encuestados varía desde 3 hasta 40, con una mediana de 13 años. Se analizó si existen divergencias entre las respuestas del 50% de los ingenieros menos experimentados y el 50% de aquellos que tienen mayor experiencia laboral. Para la mayoría de las preguntas existe congruencia entre las respuestas de cada uno de los dos grupos, pero hay unas pocas preguntas cuyas respuestas muestran algunas diferencias interesantes. A saber:

- **¿Qué tan importante considera el conocimiento en patrones de diseño para el ejercicio de su profesión?** Solamente 5 ingenieros del grupo menos experimentado de los encuestados consideran que es muy importante (valor máximo de la escala), pero 11 ingenieros del grupo más experimentado lo consideran también de esa forma. Hay una diferencia de poco más del doble.
- **¿Qué tan importante considera la experiencia trabajando con patrones de diseño para el ejercicio de su profesión?** Solamente 4 ingenieros del grupo menos experimentado de los encuestados consideran que es muy importante (valor máximo de la escala), pero 7 ingenieros del grupo más experimentado lo consideran también de esa forma. Hay una diferencia de casi el doble.
- **¿Conoce otras listas, catálogos o dominios de patrones de diseño, aparte del GoF**? Solo 2 ingenieros del grupo menos experimentado de los encuestados conocen otros catálogos, pero 10 ingenieros del grupo más experimentado afirman también conocer otros dominios de patrones de diseño. El primer grupo tiene solo una quinta parte de las respuestas afirmativas del segundo.
- **¿Qué tan de acuerdo esta con la siguiente afirmación?: Los patrones de diseño proporcionan un vocabulario común que mejora la comunicación de conceptos complejos de diseño.** En este caso, 8 ingenieros del grupo menos experimentado de los encuestados dicen estar totalmente de acuerdo con la afirmación (valor máximo de la escala), pero 14 ingenieros del grupo más experimentado lo están también. Hay una diferencia cercana al doble.

- **¿Con qué frecuencia utiliza patrones de diseño creacionales?** Aquí, 7 ingenieros del grupo menos experimentado de los encuestados dicen utilizarlos siempre o casi siempre, mientras que 16 ingenieros del grupo más experimentado dicen también hacerlo. Hay una diferencia de más del doble.
- **¿Con qué frecuencia utiliza patrones de diseño estructurales?** En este caso, 3 ingenieros del grupo menos experimentado de los encuestados dicen utilizarlos siempre o casi siempre, mientras que 16 ingenieros del grupo más experimentado dicen también hacerlo. El primer grupo tiene menos de una quinta parte de las respuestas positivas del segundo.
- **¿Con qué frecuencia utiliza patrones de diseño de comportamiento?** 5 ingenieros del grupo menos experimentado de los encuestados dicen utilizarlos siempre o casi siempre, mientras que 10 ingenieros del grupo más experimentado dicen también hacerlo. Hay una diferencia de un 100%.
- **¿Con qué frecuencia usted reconoce patrones de diseño en código existente?** Las respuestas a esta pregunta muestran que 8 ingenieros del grupo menos experimentado de los encuestados dicen reconocerlos siempre o casi siempre, mientras que 15 ingenieros del grupo más experimentado dicen también hacerlo. Hay una diferencia cercana al doble.

Lo anterior parece sugerir que una práctica laboral más prolongada puede influir en algunas opiniones. Los ingenieros más experimentados están más convencidos de la importancia del conocimiento y la experiencia en patrones de diseño, conocen más acerca de ellos y sus beneficios, además de que los reconocen y aplican con mucha mayor frecuencia. Esta y otras conclusiones de esta investigación se desarrollan en la siguiente sección.

CAPÍTULO 9

Conclusiones

En esta investigación se abordaron diferentes aspectos académicos, que, como lo mencionan varios autores, pueden ayudar a un mejor entendimiento y aprendizaje de diseño y patrones de diseño orientados a objetos (Allison & Harrison, 2007; Dukovich & Janzen, 2009; Freeman et al., 2004; Tao et al., 2015). También, se tomaron en cuenta insumos de la industria, en forma de ejercicios utilizados en entrevistas labores y por medio del análisis de las opiniones de ingenieros de software profesionales acerca de sus prácticas más comunes y relevantes en lo que a diseño orientado a objetos se refiere. Se intentó contrastar los resultados de cada una de las secciones de este trabajo, mezclando observaciones tomadas de ambos mundos, el académico y el de la industria.

Se trabajó con estudiantes del curso CI-0136 Diseño de Software, de la carrera de Bachillerato en Computación con varios énfasis de la Escuela de Ciencias de la Computación e Informática en la Universidad de Costa Rica. Con ellos se ejecutaron algunos ejercicios sencillos como los que (Freeman et al., 2004; McLaughlin et al., 2006) presentan en sus obras, pero como estos mismos autores mencionan, en la vida real los desarrolladores deben centrarse más en aspectos generales de diseño y no en introducir patrones de diseño a diestra y siniestra. Por esto, es importante que los estudiantes se enfrenten no solo a ejercicios sobre patrones de diseño específicos, sino también a situaciones y problemas de diseño generales más reales, como lo harían en entrevistas laborales de grandes compañías como Google, Microsoft, Apple y Amazon (McDowell, 2015).

No obstante, una entrevista laboral se puede enfocar en muchos temas y algunos de ellos pueden ser, sin duda, teóricos. Uno de los beneficios más grandes de los patrones de diseño es que proporcionan un lenguaje compartido, que ayuda a comunicar conceptos de forma simple y efectiva (Beck et al., 1995). De esta forma, alguien podría proponer utilizar un patrón como *abstract factory* o *chain of responsibility* como parte de la solución a un problema, pero entre

menos experiencia se tenga en diseño y programación orientada a objetos, más difícil puede ser recordar a que se refiere cada uno de estos patrones. Aquí, es donde las metáforas de los patrones de diseño proporcionan analogías con objetos y situaciones del mundo real, que pueden facilitar recordar su esencia, además del dónde, cuándo y cómo aplicarlos (Dukovich & Janzen, 2009; Freeman et al., 2004).

Con lo anterior en mente, se desarrollaron metáforas para cada uno de los 23 patrones de diseño del GoF y fueron sometidas a la evaluación de los estudiantes del curso de diseño. Este ejercicio dejó en evidencia que hay patrones más sencillos de explicar y entender que otros, como lo son los creacionales. Estos patrones suelen ser más sencillos que los estructurales y los de comportamiento, porque su único objetivo es crear objetos, mientras que los de las otras categorías tienen finalidades variadas y es fácil confundir algunos de ellos entre sí.

También, se encontró que algunos de estos patrones resultan complejos de comparar con elementos del mundo real. En algunos casos su definición misma hace referencia a intangibles como algoritmos, clases y subclases, que son difíciles de relacionar con objetos o situaciones del mundo real, tal es el caso de *template method*. Por otro lado, hay algunos que no se pueden explicar sin mencionar una palabra clave que los identifique, como es el caso de *prototype*, que sería difícil explicar sin mencionar el proceso de hacer una copia o un clon de algo. Sin duda, es posible decir también, que hay margen de mejora en la definición de algunas de las metáforas.

Ahora bien, el conocimiento de los estudiantes del curso no fue siempre el mismo, sino que progresó con el tiempo. Se elaboró un cuestionario que incluía una autoevaluación de sus conocimientos y experiencia en diseño orientado a objetos, que contestaron al principio, a la mitad y al final del semestre y que confirmó el avance en aspectos medulares del curso como lo son los patrones de diseño. Con el pasar del tiempo empezaron a ver los patrones como un poco menos difíciles de implementar, como algo que no necesariamente toma mucho más tiempo hacer y a mostrar mayor propensión a utilizarlos en el futuro.

Los estudiantes afirmaron tener conocimientos en aspectos de diseño y programación orientada a objetos que son fundamentales para entender los patrones de diseño mismos. No obstante, el conocimiento en estos últimos en específico y su categorización como creacionales, estructurales y de comportamiento, creció desde un punto casi nulo. Ellos reportaron conocer algo de unos pocos de estos patrones desde la primera ronda de aplicación de la entrevista, a lo mejor por tener nombres algo comunes que se podría relacionar con otros conceptos.

Por ejemplo, *proxy* es un término que podrían haber escuchado en otros contextos. *Iterator* puede ser fácilmente recordado por quienes han trabajado en Java, pues es una interfaz muy conocida y utilizada en ese lenguaje para recorrer colecciones de objetos. Por otra parte, reportaron que patrones como *bridge* y *abstract factory* les eran aún menos familiares en la tercera ronda de evaluación que en la segunda. La metáfora del patrón *bridge* es una de las que tuvo una calificación más baja en la evaluación de las analogías y parece confirmarse aquí como uno de los patrones más complicados de explicar y entender.

Aparte de esta autoevaluación de los estudiantes en momentos diferentes del curso, se hizo una evaluación final del desempeño de cada uno de ellos durante el semestre. Se consideraron tareas, actividades en clases, proyecto y examen, como parte de su calificación final. El nivel de conocimiento que los estudiantes percibían tener en la última aplicación de la encuesta en la que se incluía la autoevaluación, promedió una calificación de 8.29 puntos de 10 posibles. De forma similar, el promedio de las calificaciones finales reales del curso fue de 8.44 puntos, también de 10 posibles. Este promedio real final general es positivo y es posible ver una concordancia con sus autoevaluaciones.

Por otro lado, como contraparte a estas valoraciones hechas por los estudiantes, fue interesante realizar consultas sobre algunos de los mismos temas y otros más, a profesionales en ingeniería de software. La mayoría de los 35 profesionales entrevistados tenían más de 10 años de experiencia, lo que en algunos casos derivó en que algunos conceptos teóricos pudieran no estar tan frescos en sus memorias o que en definitiva ignoraran otros más recientes que no aprendieron en la universidad. La gran mayoría dice conocer los patrones del GoF y que son los

que más utilizan, pero cuando se les pregunta por otros catálogos de patrones de diseño más avanzados o específicos, se citan solo unos pocos de forma correcta y hay varias menciones a cosas que no lo son. Aún con estas pocas referencias vale la pena resaltar que patrones para microservicios y el *cloud* están en boga y son de creciente importancia en la industria, lo que hace una educación formal en ellos de mayor relevancia cada día.

También, con respecto a los patrones de diseño del GoF, los tres patrones que dicen utilizar con mayor frecuencia en su vida laboral son *singleton*, *factory* (que en un sentido estricto podrían hacer referencia a *abstract factory* o *factory method*) y *builder*, todos creacionales. Cuando se les preguntó acerca de cuál categoría de patrones utilizan más, la respuesta también fue creacionales. Esto parece confirmar, como se comentó en la evaluación de las metáforas, que los patrones creaciones son los más simples, fáciles de entender y de mayor uso, no solo por estudiantes, sino también por profesionales. En el peor de los casos, esto podría también indicar que su conocimiento en patrones estructurales y de comportamiento es menor y por eso no saben reconocer cuándo y dónde aplicarlos. Lo anterior podría sugerir que debe dárseles un énfasis mayor a estas categorías de patrones de diseño en los cursos donde se enseñen formalmente.

Por otra parte, no cabe duda que los ingenieros entrevistados concuerdan, en una gran mayoría, con que los beneficios de usar patrones de diseño que, autores como (Jalil & Noah, 2007), (Beck et al., 1995) y (Freeman et al., 2004), mencionan en sus obras, son reales. Los patrones de diseño, en efecto, ayudan a desarrolladores novatos a hacer mejores diseños, mejoran la calidad del código y del software generado, capturan las partes esenciales de un diseño, motivan la creación y reutilización de buenas prácticas, proporcionan un vocabulario común y propician, en general, beneficios reales en los procesos de desarrollo de software.

Además, se confirma que la experiencia laboral puede influir en la percepción de algunos temas en lo que concierne a patrones de diseño. El 50% menos experimentado del total de ingenieros encuestados tuvo, en una gran mayoría, opiniones similares a las del 50% más experimentado, pero hubo unas pocas excepciones. El grupo con mayor experiencia laboral ve como de suma importancia el conocer de patrones de diseño y poner ese conocimiento en práctica, además que

les es más fácil reconocer los patrones en código existente e introducirlos en el software que desarrollan.

Finalmente, aun cuando existen otros paradigmas de programación más allá del orientado a objetos, los ingenieros consultados están, en su mayoría, en desacuerdo con la idea que esto le reste importancia a la programación orientada a objetos. Por tanto, mientras haya uso y demanda de este paradigma y sus prácticas en el mundo laboral, es muy probable que este siga siendo un tema también de suma importancia en la academia y en la preparación universitaria de los estudiantes. De aquí que la mejora continua, la actualización y el avance en la cantidad de temas cubiertos en los cursos de diseño de software, sean de vital importancia para la inserción de los estudiantes, como futuros ingenieros, en el mundo laboral.

Referencias

Alic, D., Omanovic, S., & Giedrimas, V. (2016). Comparative analysis of functional and object-oriented programming. En *2016 39th International Convention on Information and Communication Technology, Electronics and Microelectronics, MIPRO 2016 - Proceedings* (pp. 667–672). Institute of Electrical and Electronics Engineers Inc. https://doi.org/10.1109/MIPRO.2016.7522224

Allison, C. D., & Harrison, N. B. (2007). *Teaching design patterns: a matter of principle*.

Bass, L., Clements, P., & Kazman, R. (2012). *Software Architecture in Practice, Third Edition. Software Architecture*. Boston: Addison-Wesley.

Beck, K., Coplien, J. O., Crocker, R., Dominick, L., Meszaros, G., Paulisch, F., & Vlissides, J. (1995). Industrial experience with design patterns. En *Proceedings - International Conference on Software Engineering* (pp. 103–114). IEEE. https://doi.org/10.1109/icse.1996.493406

Blancarte, O. (2016). Flyweight Structural pattern. Recuperado el 30 de mayo de 2021, de https://reactiveprogramming.io/blog/en/design-patterns/flyweight

CareerCup. (2015). CtCI-6th-Edition/Java/Ch 07. Object-Oriented Design/Q7_04_Parking_Lot/ Latest commit. Recuperado el 17 de junio de 2021, de https://github.com/careercup/CtCI-6th-Edition/tree/master/Java/Ch 07. Object-Oriented Design/Q7_04_Parking_Lot

Dehbozorgi, N., Macneil, S., Maher, M. Lou, & Dorodchi, M. (2019). A Comparison of Lecture-based and Active Learning Design Patterns in CS Education. En *Proceedings - Frontiers in Education Conference, FIE* (Vol. 2018-Octob). Institute of Electrical and Electronics Engineers Inc. https://doi.org/10.1109/FIE.2018.8659339

Diseño de Software | Escuela de Ciencias de la Computación e Informática. (2020). Recuperado el 29 de agosto de 2020, de https://www.ecci.ucr.ac.cr/cursos/ci-0136

Dukovich, A., & Janzen, D. S. (2009). Design patterns go to hollywood: Teaching patterns with multimedia. En *ITNG 2009 - 6th International Conference on Information Technology: New Generations* (pp. 684–689). https://doi.org/10.1109/ITNG.2009.199

Educative Inc. (2021). Designing Twitter - Grokking the System Design Interview. Recuperado el 29 de junio de 2021, de https://www.educative.io/courses/grokking-the-system-design-interview/m2G48X18NDO

Fehling, C., Leymann, F., Retter, R., Schupeck, W., & Arbitter, P. (2014). *Cloud Computing Patterns*. *Cloud Computing Patterns*. Springer Vienna. https://doi.org/10.1007/978-3-7091-1568-8

Fowler, M., Rice, D., Foemmel, M., Mee, R., & Stafford, R. (2002). *Patterns of Enterprise Application*. *Wesley, Addison*.

Freeman, E., Robson, E., Bates, B., & Sierra, K. (2004). *Head First Design Patterns*. O' Reilly & Associates, Inc.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Hinsen, K. (2009). The promises of functional programming. *Computing in Science and Engineering*, *11*(4), 86–90. https://doi.org/10.1109/MCSE.2009.129

Huang, H., & Yang, D. (2008). Teaching design patterns: A modified PBL approach. En *Proceedings of the 9th International Conference for Young Computer Scientists, ICYCS 2008* (pp. 2422–2426). https://doi.org/10.1109/ICYCS.2008.127

Jalil, M. A., & Noah, S. A. M. (2007). The difficulties of using design patterns among novices: An exploratory study. En *Proceedings - The 2007 International Conference on Computational Science and its Applications, ICCSA 2007* (pp. 97–103). Kuala Lampur, Malaysia.

https://doi.org/10.1109/ICCSA.2007.75

Joshi, B., & Joshi, B. (2016). Overview of SOLID Principles and Design Patterns. En *Beginning SOLID Principles and Design Patterns for ASP.NET Developers* (pp. 1–44). Apress. https://doi.org/10.1007/978-1-4842-1848-8_1

Lartigue, J. W., & Chapman, R. (2018). Comprehension and application of design patterns by novice software engineers. En *Proceedings of the ACMSE 2018 Conference on - ACMSE '18* (pp. 1–10). New York, New York, USA: ACM Press. https://doi.org/10.1145/3190645.3190686

MacDonald, D. (2019). Anti-patterns and dark patterns. *Practical UI Patterns for Design Systems*, 193–221. https://doi.org/10.1007/978-1-4842-4938-3_5

McDowell, G. L. (2015). *Cracking the Coding interview: 189 Programming Questions and Solutions 6th Edition* (6th ed.). Palo Alto, CA: CareerCup.

McLaughlin, B. D., Pollice, G., & West, D. (2006). *Head First Object-Oriented Analysis and Design: A Brain Friendly Guide to OOA&D (Head First)*. O'Reilly Media, Inc.

MDN. (2021). Detalles del modelo de objetos - JavaScript | MDN. Recuperado el 22 de junio de 2021, de https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Details_of_the_Object_Model

Potts, J. D. (2018). Design Patterns 4: The Singleton | John D Potts. Recuperado el 29 de mayo de 2021, de https://www.johndpotts.com/2018-09-03-design-patterns-4-singleton/

Sarcar, V. (2018). Anti-patterns. *Design Patterns in C#*, 391–395. https://doi.org/10.1007/978-1-4842-3640-6_28

Saunders, M., & Tosey, P. (2013). The Layers of Research Design. *Rapport*, (Winter).

Shvets, A. (2019). *Dive Into Design Patterns*. Refactoring.Guru.

Shvets, A. (2021). Design Patterns. Recuperado el 19 de septiembre de 2020, de https://refactoring.guru/design-patterns

Sliger, M., & Consulting, S. (2008). Agile Project Management and the PMBOK ® Guide. *PM Network*, 1–7. Recuperado de http://www.educause.edu/visuals/shared/pd/Agile_project_management.pdf

SourceMaking.com. (2019). Design Patterns. Recuperado el 30 de mayo de 2021, de https://sourcemaking.com/design_patterns

Starcevic, D. (2016). Design Patterns Stories - Memento. Recuperado el 5 de junio de 2021, de http://www.design-patterns-stories.com/patterns/Memento/

Tao, Y., Liu, G., Mottok, J., Hackenberg, R., & Hagel, G. (2015). Just-in-Time-Teaching experience in a Software Design Pattern course. En *IEEE Global Engineering Education Conference, EDUCON* (Vol. 2015-April, pp. 915–919). IEEE Computer Society. https://doi.org/10.1109/EDUCON.2015.7096082

TIOBE Software BV. (2021). TIOBE Index. Recuperado el 8 de julio de 2021, de https://www.tiobe.com/tiobe-index/

Warren, I. (2005). Teaching Patterns and Software Design. En *Proceedings of the 7th Australasian Conference on Computing Education - Volume 42* (pp. 39–49). AUS: Australian Computer Society, Inc. https://doi.org/10.5555/1082424.1082430

Weiss, S. (2005). Teaching design patterns by stealth. *ACM SIGCSE Bulletin*, *37*(1), 492. https://doi.org/10.1145/1047124.1047500

Zapponi, C. (2021). Github Language Stats - GitHut 2.0 A SMALL PLACE TO DISCOVER LANGUAGES IN GITHUB. Recuperado el 22 de junio de 2021, de https://madnight.github.io/githut/#/pull_requests/2021/1