



Universidad Popular Autónoma del Estado de Puebla
Centro Interdisciplinario de Posgrados
Investigación y Consultoría
Departamento de Ingeniería y Tecnologías de Información
Doctorado en Ingeniería de Software

“MoACPOO, Modelo para Asegurar la mejora de la Calidad en
el desarrollo de programas en los cursos de Programación Orientada a
Objetos”

Tesis que para obtener el Grado de Doctora
en Ingeniería de Software

Presenta

Niño Membrillo Yedid Erandini

Directora de Tesis

Dra. María del Rocío Morales Salgado



UPAEP – Secretaría General

Dirección General de Apoyos Académicos

Dirección del Centro de Recursos para el Aprendizaje y la Investigación.

Biblioteca Central - **Karol Wojtyla**

Tesis Digitales Restricciones de uso:

DERECHOS RESERVADOS ©

PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis está protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de textos, imágenes, gráficas, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente de donde la obtuvo mencionando el autor o autores involucrados en el documento.

Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

AGRADECIMIENTOS

A mi esposo Tito y a mis hijos Jonas y Abner, por su apoyo incondicional, por su paciencia y comprensión durante mi formación doctoral, por acompañarme en mis largas jornadas dedicadas a este trabajo y darme ánimos durante el desarrollo de este trabajo doctoral, pero sobre todo por haber creído en mí y ser el motor de mi vida. A mi mamá y papá porque gracias a la formación que me dieron y a su ejemplo, aprendí que hay que ser perseverantes para lograr lo que uno se propone. Y a mi sobrina Larissa, porque a pesar de su corta edad, mostró comprensión y paciencia cuando me acompañaba en casa.

A la Doctora María del Rocío Morales Salgado quien además de ser una excelente directora de tesis es un gran ser humano. Gracias por acompañarme en este arduo camino, por su compromiso y por compartir sus conocimientos. Nunca olvidaré, el rigor, la paciencia y el empeño con la que condujo este trabajo doctoral, gracias por tener fe en mí.

Al Doctor Sodel Vázquez Reyes y a la Doctora Bárbara Emma Sánchez Rinza, mis revisores de tesis, quienes siempre me apoyaron incondicionalmente y tuvieron disposición para darme recomendaciones, sus aportaciones fueron muy valiosas.

Finalmente quiero agradecer los docentes que me dieron clases, sus enseñanzas fueron relevantes no solo en mi formación doctoral sino también de manera profesional. Y no puedo olvidar a mis compañeros de aula, con quienes trabaje en diversos proyectos y compartimos experiencias. Con orgullo puedo decir que actualmente algunos de ellos son grandes amigos.

Gracias a todos.

ÍNDICE

Resumen.....	1
Abstract.....	1
Introducción	2
Capítulo 1. Propósito y contexto	5
1.1 Problemática.....	5
1.2 Propósito de la investigación	8
1.3 Justificación de la investigación	8
1.4 Preguntas de investigación	8
1.5 Objetivo general	9
1.6 Objetivos específicos.....	9
1.7 Propuesta.....	9
1.8 Hipótesis y supuestos	9
Capítulo 2. Marco teórico.....	11
Introducción	11
2.1 Programación	12
2.2 Prácticas de ingeniería de software.....	16
2.3 Modelos de calidad del software	25
Conclusión.	45
Capítulo 3. Contexto de la enseñanza de la programación en las universidades.	46
Introducción	46
3.1 Estatus de la programación en el mundo	46
3.2 Enseñanza de la programación orientada a objetos.....	47
3.3 Calidad en la programación orientada a objetos	49
Conclusión.	50
Capítulo 4. Metodología.....	52
Introducción.	52
4.1 Metodología de la investigación	53
4.1.1 Revisión de literatura.....	53
4.1.2 Enfoque y Alcance	54

4.1.3	Selección de la muestra	54
4.1.4	Instrumento de recolección de datos.....	55
4.1.5	Recolección de datos	56
4.1.6	Análisis y obtención de resultados.....	57
4.2	Metodología del modelo MoACPOO	58
	Conclusión.	58
Capítulo 5. Resultados de las mejores prácticas de ingeniería de software y su incorporación en el ciclo de vida para los cursos de programación en la formación profesional en México.		59
	Introducción.	59
5.1	Enfoques y herramientas utilizados en los cursos de programación.	60
5.2	Ciclo de vida como apoyo en los cursos de programación en el ámbito universitario.	62
5.2.1	Requerimientos.....	63
5.2.2	Análisis.....	64
5.2.3	Diseño	66
5.2.4	Codificación	67
5.2.5	Compilación	68
5.2.6	Pruebas	69
5.2.7	Mantenimiento.....	71
5.3	Selección de las mejores prácticas de ingeniería de software	72
	Conclusiones.	80
Capítulo 6. Resultados de los criterios de calidad para los cursos de programación.....		81
	Introducción.	81
6.1	Especificaciones que los docentes solicitan y utilizan en los cursos de programación.	82
6.2	Selección de las especificaciones con el método de escalamiento de Likert.	88
6.3	Definición de los criterios de calidad en las fases del ciclo de vida.....	92
	Conclusiones.	94
Capítulo 7. Modelo para el aseguramiento de la calidad con prácticas de Ingeniería de Software como apoyo en la programación orientada a objetos.		96
	Introducción.	96
7.1	Modelo MoACPOO	96
7.1.1	Planteamiento del problema	98
7.1.2	Análisis.....	99

7.1.3	Diseño	102
7.1.4	Codificación	104
7.1.5	Compilación	106
7.1.6	Pruebas	108
	Conclusiones	109
Capítulo 8. Resultados de la aplicación del modelo		111
	Introducción.	111
8.1	Contexto de la investigación para la aplicación del modelo	112
8.2	Planificación para la aplicación del modelo.....	113
8.3	Resultados de la aplicación del modelo	114
8.4	Calificaciones de estudiantes	120
8.5	Resultados de la posprueba.....	123
	Conclusiones	127
Capítulo 9. Conclusiones.....		129
	Referencias.....	135
ANEXO 1 Cuestionario		142
ANEXO 2 Ejemplos pilotos		146
	Ejemplo piloto 1.	146
	Ejemplo piloto 2.	148
ANEXO 3 Posprueba		149

ÍNDICE DE TABLAS

Tabla 2.1. Ejemplos de buenas prácticas por área de conocimiento	18
Tabla 2.2. Clasificación de diagramas de UML por vista	20
Tabla 2.3. Pruebas por fase en el desarrollo del software	24
Tabla 2.4. Características de calidad de ISO/IEC 9126	29
Tabla 2.5. Subcaracterísticas de calidad de ISO /IEC 9126.....	29
Tabla 2.6. Características de calidad en uso en ISO/IEC 9126	31
Tabla 2.7 Características/Subcaracterísticas Modelo calidad en uso ISO/IEC25010.....	33
Tabla 2.8. Modelo de calidad del producto en ISO/IEC 25010	34
Tabla 2.9. Características/Subcaracterísticas en ISO/IEC 25010.....	34
Tabla 2.10. Factores de calidad de McCall.....	37
Tabla 2.11. Criterios de calidad de McCall	38
Tabla 2.12. Factores y criterios de calidad con base en las perspectivas de McCall.....	40
Tabla 2.13. Características y subcaracterísticas en Bohem.....	41
Tabla 2.14. Relación de los niveles en Bohem	42
Tabla 2.15. Propiedades y característica del modelo de Dromey	44
Tabla 2.16. Propiedades y su asociación con las características en el modelo de Dromey	45
Tabla 4.1 Universidades e IES en donde se respondieron los cuestionarios	56
Tabla 5.1. Prácticas utilizadas por los docentes en la fase de requerimientos en los cursos de programación en México 2015 – 2016.	64
Tabla 5.2. Prácticas de ingeniería de software utilizadas por los docentes en la fase de análisis en los cursos de programación en México 2015 – 2016.	65
Tabla 5.3. Prácticas de ingeniería de software utilizadas por los docentes en la fase de diseño en los cursos de programación en México 2015 – 2016.	66
Tabla 5.4. Prácticas de ingeniería de software utilizadas por los docentes en la fase de codificación en los cursos de programación en México 2015 – 2016.	68
Tabla 5.5. Prácticas de ingeniería de software utilizadas por los docentes en la fase de compilación en los cursos de programación en México 2015 – 2016.	69
Tabla 5.6. Prácticas de ingeniería de software utilizadas por los docentes en la fase de pruebas en los cursos de programación en México 2015 – 2016.....	70
Tabla 5.7. Prácticas de ingeniería de software utilizadas por los docentes en la fase de mantenimiento en los cursos de programación en México 2015 – 2016.....	71
Tabla 5.8. Prácticas de ingeniería de software más frecuentes usadas por los docentes en los cursos de programación en México 2015 – 2016.	72
Tabla 5.9. Mejores prácticas de ingeniería de software y su inserción en el CV	79
Tabla 6.1. Respuesta de las especificaciones solicitadas por los docentes en el desarrollo de programas en las universidades públicas en México 2015 – 2016.....	89
Tabla 6.2. Puntuación de las especificaciones utilizando el método de escalamiento de Likert	90
Tabla 6.3. Especificaciones por fases del ciclo de vida.	91
Tabla 6.4. Criterios de calidad que se pueden adaptar en los cursos de programación	92
Tabla 6.5. Criterios de calidad para los cursos de programación.	93

Tabla 6.6. Inclusión de los criterios de calidad en el ciclo de vida para los cursos de programación.	94
Tabla 7.1. Métricas para el criterio de calidad comprensión	99
Tabla 7.2. Métricas para el criterio de calidad completitud	102
Tabla 7.3. Métricas para el criterio de calidad consistente	104
Tabla 7.4. Métricas para el criterio de calidad entendible	105
Tabla 7.5. Métricas para el criterio de calidad corrección	107
Tabla 7.6. Métricas para el criterio de calidad exactitud	109
Tabla 8.1 Grupos experimental y de control	113
Tabla 8.2. Escala de satisfacción para los proyectos	114
Tabla 8.3. Resultados de las métricas del proyecto, por estudiante	115
Tabla 8.4. Calificaciones obtenidas del proyecto del grupo B	117
Tabla 8.5. Calificaciones obtenidas del proyecto del grupo A	118
Tabla 8.6. Calificaciones de los estudiantes en el curso de PI	121
Tabla 8.7. Puntuaciones por sección y total de la posprueba	124
Tabla 9.1. Nivel de satisfacción del proyecto de cada grupo	131
Tabla 9.2. Medias y porcentaje de estudiantes que respondieron la posprueba	131

ÍNDICE DE IMÁGENES

Imagen 2.1. Temáticas del trabajo doctoral.....	12
Imagen 2.2. Áreas de conocimiento de SWEBOK para los cursos de programación	18
Imagen 2.3. Modelo de calidad para la calidad interna y externa	28
Imagen 2.4. Modelo de calidad en uso en ISO/IEC 25010	32
Imagen 4.1. Fases de la metodología de la investigación	53
Imagen 5.1. Estadísticos descriptivos de 31 variables	74
Imagen 5.2. Medida de adecuación muestral KMO.....	75
Imagen 5.3. Fragmento de la matriz de correlaciones	75
Imagen 5.4. Porcentaje de varianza explicada	76
Imagen 5.5. Matriz de componentes extraídos	77
Imagen 5.6. Matriz de componentes rotados.	78
Imagen 7.1 Modelo MoACPOO.....	97
Imagen 7.2. Fase de análisis	100
Imagen 7.3. Fase de diseño	103
Imagen 7.4. Fase de codificación	105
Imagen 7.5. Fase de compilación	107
Imagen 7.6. Fase de pruebas	108
Imagen 8.1. Prueba t student para las calificaciones.....	123
Imagen 8.2. Prueba t student de la posprueba.....	126

ÍNDICE DE GRÁFICAS

Gráfica 5.1. Cursos de programación que imparten los docentes	60
Gráfica 5.2. Enfoques utilizados por los docentes en los cursos de programación	61
Gráfica 5.3. Herramientas de apoyo en los cursos de programación	61
Gráfica 5.4. Importancia de dar a conocer un CV en los cursos de programación.....	62
Gráfica 5.5 Docentes que consideran importante la fase de requerimientos	63
Gráfica 6.1. Especificaciones para verificar el aprendizaje	83
Gráfica 6.2. Especificaciones solicitadas para desarrollar un programa	84
Gráfica 6.3. Especificaciones solicitadas para analizar un problema.....	85
Gráfica 6.4. Especificaciones para diseñar la solución de un problema.....	86
Gráfica 6.5. Especificaciones solicitadas durante la codificación.....	87
Gráfica 6.6. Especificaciones que utilizan los docentes para pruebas	88
Gráfica 8.1. Puntajes obtenidos en el proyecto, grupo experimental	118
Gráfica 8.2. Puntajes obtenidos en el proyecto, grupo de control.....	119
Gráfica 8.3. Puntajes obtenidos en el proyecto, grupos A y B.....	120
Gráfica 8.4. Comparación de calificaciones de estudiantes, grupo A y B	122
Gráfica 8.5. Relación de las puntuaciones del proyecto y posprueba, grupos A y B	125

Resumen

Los cursos de programación son relevantes en la formación profesional de los estudiantes que estudian carreras relacionadas con las ciencias computacionales. Sin embargo, representan un reto para ellos al presentar dificultades para comprender los conceptos, abstraer el problema, plantear la solución y codificarla. Teniendo como consecuencia la creación de programas no correctos y con defectos, fenómeno que se identificó al hacer una revisión de literatura y en entrevistas realizadas a 55 docentes de 29 Universidades de México. Por ello, se propone un modelo que asegure el uso de buenas prácticas de ingeniería de software y criterios de calidad en el desarrollo de programas de los estudiantes en los cursos de programación orientada a objetos que coadyuve en la mejora de su proceso de desarrollo al crear programas con un análisis completo, con un diseño adecuado, con un código limpio y con los resultados correctos, es decir con calidad. El modelo está sustentado en la investigación realizada sobre las prácticas de ingeniería de software y los criterios de calidad utilizados en los cursos de programación en el ámbito universitario, así como de la revisión de literatura de los principales modelos de calidad del software. El modelo fue probado en un curso de programación orientada a objetos para evaluar su pertinencia.

Abstract

Programming courses are important in the professional training of students that study computer science related careers. However, these courses are a challenge for students since they have difficulties in understanding concepts, abstracting a problem, proposing a solution and encoding it. This results in the creation of incorrect programs with defects, a phenomenon that was identified when doing a literature review and interviews conducted to 55 teachers of 29 universities of Mexico. Thus, a model is proposed, such that it ensures the use of good practices of software engineering and quality criteria in the program development of the students in the object oriented programming courses that help improve their development process by creating programs with a complete analysis, with a suitable design, with clean code and with the correct results. The model is supported by research about software engineering practices and quality criteria used in programming courses in the university environment, as well as, in the literature review of the main quality software models. The model was tested in an object-oriented programming course to evaluate if it's relevant.

Introducción

Los cursos de programación son de gran relevancia en la formación profesional de los estudiantes que estudian carreras relacionadas con las ciencias computacionales, debido a que al insertarse en el campo laboral contribuyen en dar solución a proyectos de software. Con base en la Encuesta Nacional de Ocupación y Empleo (ENOE) de 2018 a nivel nacional existen 1.4 millones de personas que estudiaron alguna carrera técnica o profesional relacionadas con las Tecnologías de Información y Comunicación (TIC) (43.9%), con las Ciencias de la Computación (39.6%) y, con Electrónica y Automatización (16.5%). Del total de encuestados sólo el 72.3% trabajan (1,056,933), el 29.8% (314,967) ejercen como profesionistas y de estos el 59.8% (188,351) se desempeñan como desarrolladores y analistas de software y multimedia (INEGI, 2019). Asimismo, la Universidad Autónoma Metropolitana (UAM) de la unidad Azcapotzalco de la división de Ciencias Básicas e Ingenierías (CBI), presentó un reporte con los resultados de la encuesta aplicada a 185 egresados de un total de 595 de la carrera de Ingeniería en Computación, el 31.6% (59 egresados) indicaron que se dedican a la programación como principal actividad que desempeñan en su trabajo; este reporte se fundamenta con la información que se procesa de manera permanente y consistente desde 2005 (Universidad Autónoma Metropolitana, 2017). En consecuencia, los estudiantes deben desarrollar competencias y habilidades de pensamiento crítico y solución de problemas que es una de las competencias del siglo XXI. Sin embargo, los estudiantes en los cursos de programación, principalmente en los introductorios, se enfrentan a retos importantes al seguir un proceso para comprender un problema y estructurar su solución mediante una serie de acciones que implican el conocimiento conceptual y la aplicación de estos para crear un programa por medio de la computadora. Desafortunadamente, el estatus de los cursos de programación en la educación superior en México no es alentador, ya que los estudiantes presentan dificultades y deficiencias para comprender los conceptos, abstraer el problema, plantear la solución y codificarla, teniendo como resultados programas no correctos y con defectos, que es un fenómeno que se identificó al hacer una revisión de literatura y con la realización de una entrevista no estructurada realizada a 55 docentes de 29 Universidades de México.

Es por lo anterior, que en el presente trabajo doctoral se propone el modelo MoACPOO (Modelo para Asegurar la mejora de la Calidad en el desarrollo de programas en los cursos

de Programación Orientada a Objetos), un modelo para la mejora de los programas en los cursos de programación orientada a objetos que incluye un conjunto de buenas prácticas de ingeniería de software y criterios de calidad para que los estudiantes puedan desarrollar un programa con calidad. Es decir, la creación de programas con calidad implica la comprensión y análisis del problema para identificar los requisitos predefinidos que serán la base para el diseño, el cual debe estar soportado por los principios SOLID por sus siglas en inglés (Single responsibility principle; Open closed principle; Liskov substitution principle; Interface segregation principle; Dependency inversion principle) lo que permite tener un código mejor estructurado y que aunado a las buenas prácticas de código limpio se tendrá un código entendible, mantenible y funcional. Para resumir, el análisis debe ser completo, el diseño adecuado, el código limpio y en todo momento se debe cumplir con los requisitos predefinidos para obtener los resultados correctos.

En este sentido, el conjunto de buenas prácticas de ingeniería de software y los criterios de calidad se identificaron y se definieron, respectivamente, aplicando un cuestionario a 84 docentes de 38 universidades públicas incorporadas a la Asociación Nacional de Universidades e Instituciones de Educación Superior (ANUIES), además de la consulta de referentes teóricos.

Por otra parte, el modelo MoACPOO se aplicó en un ambiente real en un curso de programación orientada a objetos con un enfoque cuantitativo, un diseño cuasiexperimental y un muestreo no probabilístico para evaluarlo, con dos grupos uno experimental y uno de control.

El trabajo doctoral está estructurado por nueve capítulos. En el primero se presenta la problemática y el contexto de la investigación. En el segundo se encuentra el marco teórico con base en las temáticas de programación, prácticas de ingeniería de software y modelos de calidad. En el capítulo tres se expone el contexto de la enseñanza de la programación en las universidades con base en el estado del arte. En el capítulo cuatro se presenta la metodología con la que se llevó a cabo la investigación. En el capítulo cinco se presentan los resultados obtenidos para identificar de las mejores prácticas de ingeniería de software. En el capítulo seis se presentan los resultados obtenidos para definir los criterios de calidad. En el capítulo siete se expone el modelo propuesto MoACPOO. En el capítulo ocho se presentan los

resultados al aplicar MoACPOO en un ambiente real. Y en el capítulo nueve se exponen las conclusiones y recomendaciones para trabajos futuros.

Capítulo 1. Propósito y contexto

1.1 Problemática

Uno de los desafíos a los que se enfrenta la industria del software es el desarrollar software exitoso, desafortunadamente el 70% de los proyectos se retrasan, exceden el presupuesto, no cumplen con las especificaciones del cliente o fracasan (Lynch, 2018), siendo los estudiantes y egresados de carreras afines a las ciencias computacionales quienes contribuyen en dar solución a proyectos de software cuando ingresan al mercado laboral, y aunque no es determinante, coadyuvan con estos resultados.

CONACYT señala que el número de egresados de carreras relacionadas con las TICs son principalmente de las áreas de ingeniería y tecnología. En la actualidad, existe una fuerte preocupación en este sector porque México no cuenta con los recursos humanos adecuados para cubrir la demanda de las empresas, lo que implica que éstas deban invertir tiempo y dinero en capacitar a los egresados y nuevos empleados (Mejía, 2017, pág. 13). La OECD (2019) indica que en México casi la mitad de los empleadores alertan de una falta de competencias en su sector y consideran que la educación y formación de los solicitantes de empleo no es adecuada para sus necesidades. En consecuencia, es importante que las universidades donde se imparten carreras afines a las ciencias computacionales actualicen o reestructuren sus planes de estudio y realicen un vínculo con la industria, por ello la Asociación de Maquinaria Computacional (ACM) y el Instituto de Ingeniería Eléctrica y Electrónica (IEEE) han generado pautas para los planes de estudio relacionados con las ciencias computacionales como Ingeniería de software (Joint Task Force on Computing Curricula, 2015) y Tecnologías de la Información (Task Group on Information Technology Curricula, 2017). Asimismo, en la mesa de trabajo de diseño curricular de la ANIEI (Asociación Nacional de Instituciones de Educación en Tecnologías de Información) se plantearon estrategias para que las Instituciones de Educación Superior (IES) con programas educativos de Tecnologías de la Información en México diseñen un currículo flexible orientado a las demandas de las competencias profesionales (Aguas García, Balderas Pérez, & Díaz Mendoza, 2018).

En este sentido, los cursos de programación que son ofertados en las carreras relacionadas con ciencias computacionales son relevantes para el desarrollo de software y deben permitir

al estudiante dar una correcta solución a problemas por medio de la computadora con base en un proceso de desarrollo adecuado que potencie su ingenio y su razonamiento lógico. Además, el estudiante debe estar consiente que, la programación no solo se trata de escribir código sino también de la capacidad de analizar un problema, identificar sus componentes clave, modelar los datos y procesos, y crear o refinar un programa a través de un enfoque ágil de diseño y pensamiento, en donde los estudiantes podrían usar la programación como una herramienta de creación de conocimiento y modelado en actividades creativas de resolución de problemas (Romero, Lepage, & Lille, 2017). Sin embargo, en la práctica los cursos de programación, como es la programación orientada a objetos, representan un desafío para los estudiantes debido a que enfrentan diversas dificultades y deficiencias al no comprender los conceptos, al escaso entendimiento de los problemas planteados, a la dificultad de identificar las estructuras a utilizar, a la escasa comprensión de la semántica y al desconocimiento de la sintaxis del lenguaje de programación a utilizar, siendo evidente que hay poca planificación del proceso en la resolución de problemas (Özmen & Altun, 2014; Olier, Gómez, & Caro, 2017; Bosse & Gerosa, 2016).

Las deficiencias a las que se enfrentan los estudiantes en los cursos de programación repercuten en el desarrollo de programas no correctos y con defectos, por lo que el desarrollo de software con calidad de código en un ambiente universitario es un desafío (Krusche, Berisha, & Bruegge, 2016). La baja calidad de programas desarrollados por estudiantes se debe a la dificultad de abstracción del problema; falta de revisión de diseño y código; se centran más en la codificación; carecen de habilidades de diseño y resolución de problemas y no identifican defectos (Nel, Nel, & Cronje, 2015), es por ello que para apoyar a los estudiantes en la creación de programas con calidad se debe incluir un proceso de desarrollo con buenas prácticas (Rong, Zhang, Qi, & Shao, 2016).

Por otra parte, el estatus de la programación en la educación superior en México no es alentador por la dificultad que presentan estudiantes para entender el problema; al desinterés de seguir una directriz que guíe su desarrollo; a la falta del uso de buenas prácticas y al número de estudiantes en el aula, que origina el desarrollo de programas no funcionales; y repercute en los índices de reprobación (UAM, 2013; Olivares Rojas, Jiménez Murillo, Ortiz Ortiz, & Rodríguez Maya, 2015; Macías Brambila, Zamora Ramos, Osorio Angel, & Jiménez Rodríguez, 2016). El fenómeno anterior se identificó considerando una entrevista no

estructurada realizada a 55 docentes de 29 Universidades de México: El Instituto Tecnológico de Aguascalientes, la Universidad Autónoma de Aguascalientes, la Universidad Autónoma de Baja California (UABC), el Instituto Politécnico Nacional (IPN), la Universidad Autónoma Metropolitana (UAM), la Universidad Nacional Autónoma de México (UNAM), la Universidad Politécnica de Durango, el Instituto Tecnológico de León, la Universidad Autónoma del Estado de Hidalgo, la Universidad de Guadalajara (UDG), la Universidad Autónoma del Estado de México (UAEM), la Universidad Politécnica de Texcoco (UPTex), el Tecnológico de Estudios Superiores de Chimalhuacán (TESCHI), la Universidad Tecnológica de Nezahualcóyotl (UTN), la Universidad Michoacana de San Nicolás Hidalgo, la Universidad Tecnológica de la Mixteca, la Benemérita Universidad Autónoma de Puebla (BUAP), el Instituto Tecnológico de Puebla, la Universidad Tecnológica de Izúcar de Matamoros (UTIM), la Universidad Tecnológica de Huejotzingo, la Universidad Tecnológica de Tecamachalco, la Universidad Politécnica de San Luis Potosí, la Universidad Autónoma de San Luis Potosí (UASLP), la Universidad Politécnica de Altamira, el Instituto Tecnológico de Matamoros, el Instituto Superior de Coatzacoalcos, la Universidad Veracruzana, la Universidad Autónoma de Yucatán, y la Universidad Autónoma de Zacatecas.

Ante la situación planteada, los estudiantes al tener deficiencias en programación enfrentan problemas en cursos subsecuentes que demandan proyectos que deben dar solución mediante programación, no cumpliendo con el objetivo de los proyectos. Por ello esta tesis está orientada en incorporar el uso de buenas prácticas de ingeniería de software y criterios de calidad en los cursos de programación orientada a objetos con la finalidad de que los estudiantes y egresados de las universidades mejoren el desarrollo de sus programas y puedan afrontar mejor el ingreso al campo laboral para contribuir con dar solución a necesidades informáticas y aporten valor a las empresas. De esta manera podrán hacer la transición de un entorno académico a una carrera dentro de una corporación, organización, institución académica o incluso un entorno empresarial (Task Group on Information Technology Curricula, 2017).

1.2 Propósito de la investigación

El propósito de la investigación es ofrecer un modelo, que incluye buenas prácticas de ingeniería de software utilizadas por docentes de universidades e instituciones de educación superior públicas en México y criterios de calidad sustentado en la revisión de literatura de modelos de calidad, como apoyo en los cursos de programación orientada a objetos para contribuir en la mejora de la calidad de los programas desarrollados por los estudiantes, creándoles hábitos que los ayude en su vida profesional.

1.3 Justificación de la investigación

La importancia de llevar a cabo la investigación se debe a que actualmente y desde hace años a los estudiantes se les dificultan los cursos de programación teniendo carencias en el desarrollo de programas y por ende no se cumplen con los resultados esperados. Desde finales de los 60 se ha observado empíricamente que estudiantes de programación aprenden en forma muy limitada, porque se le da poca importancia al desarrollo de la lógica y se enfocan más al lenguaje, por lo que aprenden a prueba y error, teniendo como consecuencia, la formación de programadores buenos para codificar usando lenguajes, pero sin bases lógicas sólidas (López Román, 2011).

Por lo anterior, y considerando el problema planteado con antelación, se tiene la intención de coadyuvar en mitigar las dificultades que presentan los estudiantes en los cursos de programación orientada a objetos contribuyendo así en el desarrollo de programas con calidad. Por ello, se propone un modelo para la mejora de programas en los cursos de programación orientada a objetos, que brinda un conjunto de buenas prácticas de ingeniería de software y criterios de calidad para que los estudiantes puedan desarrollar un programa con calidad. Si bien es cierto que las buenas prácticas de ingeniería de software incluidas en el modelo ya existen, el modelo en cada una de sus fases permea buenas prácticas de ingeniería de software, definición de artefactos y criterios de calidad, aunado a la inclusión de los principios SOLID y de código limpio para el diseño y la codificación respectivamente.

1.4 Preguntas de investigación

1. ¿Cuáles son las mejores prácticas de ingeniería de software utilizadas en los cursos de programación por los docentes en las universidades públicas en México?

2. ¿Cuáles son los criterios de calidad para el desarrollo de programas en los cursos de programación?
3. ¿Cómo influye un modelo con buenas prácticas de ingeniería de software y criterios de calidad en los cursos de programación orientada a objetos?

1.5 Objetivo general

Diseñar un modelo que asegure el uso de buenas prácticas de ingeniería de software y criterios de calidad en el desarrollo de programas de los estudiantes universitarios en los cursos de programación orientada a objetos que coadyuve en la mejora de su proceso de desarrollo y les permita desarrollar programas con calidad.

1.6 Objetivos específicos

1. Identificar las mejores prácticas de ingeniería de software utilizadas por los docentes en los cursos de programación en las universidades públicas en México.
2. Definir los criterios de calidad que puedan aplicarse en el desarrollo de programas en los cursos de programación con base en los principales modelos de calidad del software.
3. Evaluar el modelo en un curso de programación orientada a objetos determinando como influye en los estudiantes.

1.7 Propuesta

En este trabajo de tesis se llevó a cabo una investigación de las prácticas utilizadas en los cursos de programación con docentes de universidades públicas en México, así como la revisión de literatura de los modelos de calidad del software, a fin de diseñar un modelo para la mejora de los programas en los cursos de programación orientada a objetos que incluye un conjunto de buenas prácticas y criterios de calidad, para fomentar en los estudiantes su uso, madurar su proceso de desarrollo e involucrarlos con buenas prácticas que les permita mejorar sus programas.

1.8 Hipótesis y supuestos

Las hipótesis de la investigación son:

H0. Si se utiliza un modelo que incluya un conjunto de buenas prácticas y criterios de calidad no se mejorará el desarrollo de programas en los cursos de programación orientada a objetos.

H1. Si se utiliza un modelo que incluya un conjunto de buenas prácticas y criterios de calidad se mejorará el desarrollo de programas en los cursos de programación orientada a objetos.

Además, se tiene los siguientes supuestos:

- Los docentes de las universidades públicas en México utilizan pocas prácticas de ingeniería de software en los cursos de programación.
- Si se analizan los modelos de calidad del software y las especificaciones dadas por los docentes, se podrán determinar criterios de calidad para el desarrollo de programas en los cursos de programación que podrán alinearse con las mejores prácticas de ingeniería de software en las diferentes fases del ciclo de vida.

Capítulo 2. Marco teórico.

Introducción

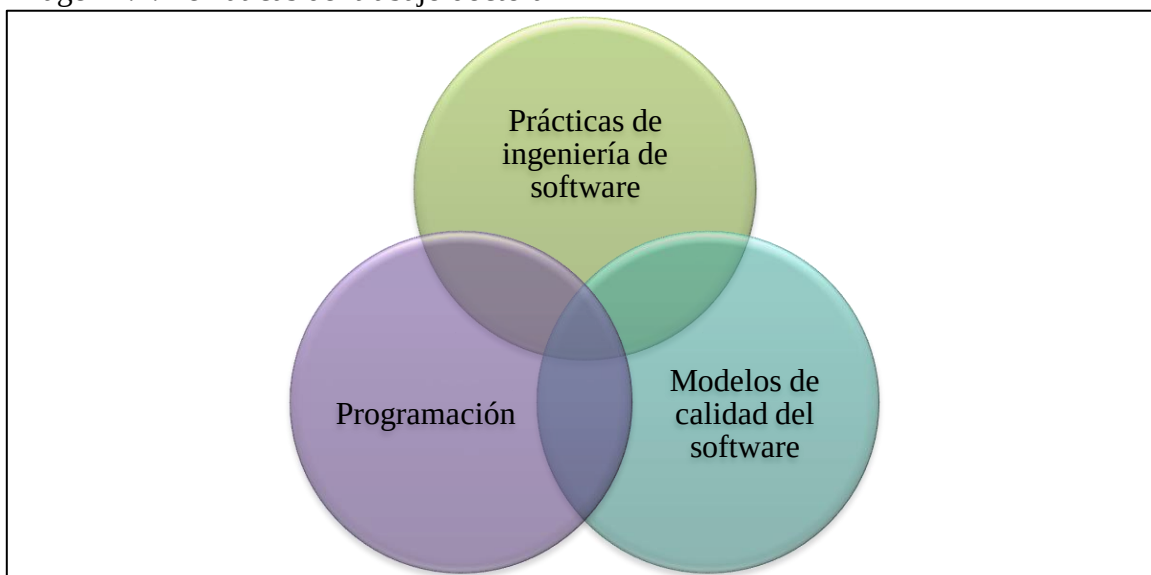
Tomando en cuenta que la ingeniería de software es una disciplina para desarrollar, operar y mantener software con calidad mediante la aplicación de herramientas, técnicas y procedimientos, es importante que estudiantes y egresados de las carreras relacionadas con ciencias computacionales potencien la competencia de pensamiento crítico y solución de problemas en los cursos de programación en las universidades, debido a que la industria del software busca talento para el desarrollo de software con calidad que proporcione valor a los usuarios.

Por consiguiente, los estudiantes deben hacer uso de buenas prácticas de ingeniería de software y evaluar la calidad durante la creación de sus programas en los cursos de programación, lo que implica la comprensión y análisis del problema, el diseño y la codificación de la solución, la funcionalidad correcta del programa y la verificación de los resultados, que de manera integral contribuyen en la resolución de problemas en diferentes contextos.

Debido a la importancia que presentan la programación, las buenas prácticas de ingeniería de software y la calidad en la creación de programas en los cursos de programación, en este capítulo se presentan las temáticas programación, prácticas de ingeniería de software y modelos de calidad del software que son el fundamento teórico del Modelo para Asegurar la mejora de la Calidad en el desarrollo de programas en los cursos de Programación Orientada a Objetos, MoACPOO, propuesto en este trabajo doctoral.

Las temáticas que se encuentran presentes en este trabajo doctoral son tres: 1) Prácticas de ingeniería de software, 2) Programación y 3) Modelos de calidad del software, ver Imagen 2.1, que se encuentran dentro del cuerpo de conocimiento de ingeniería de software de la Sociedad de Computadores del Instituto de Ingeniería Eléctrica y Electrónica (IEEE) en las áreas de conocimiento fundamentos de computación y calidad del software, a nivel mundial.

Imagen 2.1. Temáticas del trabajo doctoral



Fuente: Elaboración propia

Asimismo, las temáticas que guían el trabajo doctoral se encuentran en el área de conocimiento de programación e ingeniería de software que es una de las ocho áreas de conocimiento de los perfiles profesionales de la Asociación Nacional de Instituciones de Educación en Tecnologías de Información (ANIEI) que aplican para los planes de estudio de las carreras afines con las ciencias computacionales de las universidades en México.

Considerando que la ingeniería de software es la disciplina que versa sobre las temáticas ya mencionadas, es importante conocer su significado: La ingeniería de software se define como la aplicación de un enfoque sistemático, disciplinado y cuantificable al desarrollo, operación y mantenimiento de software, es decir la aplicación de la ingeniería de software (IEEE Computer Society, 2014).

2.1 Programación

La programación es el proceso que se sigue para dar solución a un problema por medio de la computadora. El proceso consta de diversas fases que inicia con la comprensión del problema hasta la implementación correcta de su solución, a lo que se le denomina ciclo de vida del software. Existen dos niveles en la construcción de programas, el primero se refiere al desarrollo de programas pequeños que se realizan de manera individual mientras que el segundo se relaciona con el desarrollo de programas de mayor magnitud que integran un

proyecto de software que requiere un equipo de trabajo (Joyanes Aguilar, 2008). En el contexto del trabajo doctoral la programación se orienta al primer nivel de construcción focalizándose en el entendimiento de los conceptos y en el proceso a seguir, siendo la comprensión del problema de gran relevancia para crear una solución adecuada.

Como se indicó anteriormente, para dar solución a un problema mediante una computadora requiere un proceso, siendo el ciclo de vida en cascada considerado por diversos autores (Fuentes-Rosado & Moo-Medina, 2017; Rong, Zhang, Qi, & Shao, 2016; Humprey, 2001; Adnan & Ritzhaupt, 2018) con las siguientes fases (Pressman, 2002).

1. Análisis de los requisitos del software. En esta fase se definen los requisitos del sistema con base en la comprensión del dominio de la información, se determina la función requerida, el comportamiento, el rendimiento y la interconexión. Para programas individuales, el problema se analiza con base en los requerimientos predefinidos del problema a resolver, indicando exactamente lo que debe hacer el programa y el resultado esperado.
2. Diseño. Los requisitos del software se traducen en un diseño que represente la solución del software, se define su arquitectura, las representaciones de interfaz y el detalle procedimental. Así, se diseña una solución que resuelva el problema mediante el uso de alguna técnica de diseño que es independiente a los lenguajes de programación para su posterior implementación.
3. Generación de código. El diseño se traduce a código con un lenguaje de programación, al que se le denomina código fuente, el código debe ser sencillo y limpio, codificando las funciones y las clases, en el paradigma orientada a objetos.
4. Pruebas. Una vez que se ha generado el código, se aplican pruebas al programa con diversos datos de entrada para comprobar que se cumple los requisitos, que funciona adecuadamente y que se obtienen los resultados correctos.
5. Mantenimiento. El programa se actualiza y modifica después de ser entregado al cliente a fin de que se cumplan con las necesidades de los usuarios. Se actualiza porque los requisitos deben adaptarse a un entorno externo o por mejoras funcionales. Y se modifica porque se han encontrado errores o por la falta de funcionalidad.

Es importante mencionar que, después de codificar la solución el programa se ejecuta para traducir el código fuente a lenguaje máquina lo que tiene lugar al compilar el programa,

detectando errores de compilación que se producen por el uso incorrecto de las reglas del lenguaje de programación, conocidos como errores de sintaxis. Posteriormente se obtiene el código objeto el cual es enlazado con las bibliotecas del compilador detectando errores en tiempo de ejecución, que tienen lugar por instrucciones que la computadora puede comprender, pero no ejecutar. Cuando ya no existen errores durante la ejecución se produce el programa ejecutable. Sin embargo, la ejecución del programa no garantiza que los resultados sean correctos debido a los errores lógicos que se producen, porque la lógica del programa no fue adecuada, estos errores son los más difíciles de detectar ya que el programa puede funcionar y no producir errores de compilación ni de ejecución, y el error se identifica por la obtención de resultados incorrectos (Joyanes Aguilar, 2008).

Para la obtención de resultados correctos y para crear programas claros, legibles, comprensibles y modificables no solo por las personas que lo construyeron sino también para las demás (Joyanes Aguilar, 2008), se debe hacer uso de buenas prácticas, ver sección 2.3, Pressman (2010) indica que existen principios que guían la codificación: entender el problema a resolver, comprender los principios básicos de diseño, elegir un lenguaje de programación para codificar la solución, seleccionar un ambiente de programación que facilite el trabajo, crear pruebas unitarias para los componentes a codificar. En la codificación, las reglas de estilo permiten crear programas con calidad a fin de que el programa sea comprensible y modificable.

Por otra parte, en los cursos de programación se utilizan paradigmas de programación siendo el estructurado y el orientado a objetos los principales. En el paradigma estructurado se especifica un conjunto de instrucciones secuenciales y de control que actualizan los valores en las variables con base en la ejecución paso a paso del programa, estas instrucciones pueden estar incluidas en funciones que tienen un propósito específico y resuelve una tarea concreta, haciendo énfasis en el algoritmo del problema a resolver. “En el paradigma orientado a objetos, el programa se organiza como un conjunto finito de objetos que contiene datos y operaciones (funciones miembro o métodos) que llaman a esos datos y que se comunican entre sí mediante mensajes” (Joyanes Aguilar, 2008, pág. 57). El paradigma estructurado se encuentra fuera del alcance del trabajo doctoral, debido a que el modelo es para apoyar el desarrollo de programas en el paradigma orientada a objetos.

En el paradigma orientado a objetos se debe realizar análisis, diseño y codificación, para abstraer los requisitos, las ideas arquitectónicas del diseño e implementar la solución con un lenguaje de programación. El objetivo del análisis es realizar un modelo conceptual para comprender el problema que describe la solución para desarrollar un modelo de objetos, en el dominio del problema, con un conjunto de características, objetos y relaciones. El diseño debe ser limpio, comprensible y fácil de mantener, en este se determinan las clases del sistema, sus interfaces públicas y las relaciones con otras clases, creando un diagrama de clases, con clases atómicas que tengan un solo propósito. Además, los atributos y métodos definidos en las clases deben ser accesibles a cualquier objeto que se instancia sólo a través de su interfaz ocultando los detalles de sus operaciones y sin que se afecte el comportamiento de la clase por realizar alguna modificación en otra clase, teniendo así una alta cohesión y un bajo acoplamiento. (Dooley, 2017). En la codificación se lleva a cabo la implementación del diseño haciendo uso de un lenguaje de programación orientado a objetos.

La programación orientada a objetos (POO) encapsula un conjunto de propiedades y comportamientos en una unidad denominada objeto para dar solución a problemas. Un objeto representa una entidad del mundo real, por consiguiente, es necesario realizar una abstracción del objeto para determinar tres características importantes (Dooley, 2017): 1) identidad, nombre que se le asigna al objeto para distinguirlo de otros. 2) estado, valores asociados a los atributos que describen los datos almacenados en el objeto en un momento dado y 3) comportamiento, conjunto definido de comportamientos, métodos, que al ser invocados cambian el estado del objeto.

Los objetos con características similares se describen por una clase que representa su abstracción con propiedades llamadas atributos y comportamientos denominados métodos que estarán disponibles para los objetos de la clase. Los atributos son datos que están asociados a un tipo de dato y denotan la estructura del objeto y los métodos definen las operaciones sobre los datos del objeto o instancia de una clase. Dooley (2017) indica que las clases se pueden organizar en jerarquías en donde una clase llamada clase base o superclase hereda sus atributos y sus métodos a otras clases llamadas subclases que pueden agregar atributos y métodos propios, haciendo una extensión de la clase, que es una propiedad de la POO; además al hacer la extensión de la clase se puede realizar la sobreescritura de las clases y aplicar el polimorfismo.

Las propiedades de la POO son (Joyanes Aguilar, 2008):

- 1 Abstracción. se centran en el qué es y qué es el objeto lo que permite determinar los atributos y métodos del objeto.
- 2 Encapsulamiento y ocultación de datos. Los objetos que poseen las mismas características y comportamiento se agrupan en clases que encapsulan los atributos y métodos, ocultando los datos y sus detalles de implementación para que puedan ser accedidos solo mediante la interfaz definida.
- 3 Polimorfismo. El comportamiento de un método que tiene el mismo nombre para diferentes clases se ejecuta de manera distinta en función del objeto que lo invoca. Un método con el mismo nombre en una misma clase puede tener sobrecarga al tener diferentes parámetros, de esta manera los diferentes objetos diferentes pueden responder de modo diferente al mismo mensaje.
- 4 Herencia. Permite definir nuevas clases llamadas derivadas que se especializan al presentar las mismas características y comportamiento de otras clases ya existentes, así como otras adicionales. Las clases ya existentes se denominan clases base que implementan la propiedad de generalización al compartir información entre dos entidades evitando la redundancia.
- 5 Reutilización de código. Una vez que se ha creado, escrito y depurado una clase, se puede poner a disposición de otros programadores para ser utilizada en otros proyectos reduciendo los costes de inversión y mejorando la productividad. El concepto de herencia proporciona una ampliación o extensión importante a la idea de reusabilidad.

Puesto que en la programación se sigue un proceso para dar solución correcta a un problema y a que existen diversas actividades que coadyuvan en ello, es importante adoptar prácticas de ingeniería de software en el desarrollo de la solución, en el contexto del trabajo doctoral en el desarrollo de programas en los cursos de programación.

2.2 Prácticas de ingeniería de software

Las prácticas de ingeniería de software son un conjunto de enfoques que han sido probados de manera empírica, se apoyan de herramientas y metodologías que se aplican en el desarrollo de software para mejorar su calidad, ejemplo de prácticas de ingeniería de software son el

control de versiones y la revisión del código (Gerrit Singer, 2013), por ello se le denominan buenas prácticas.

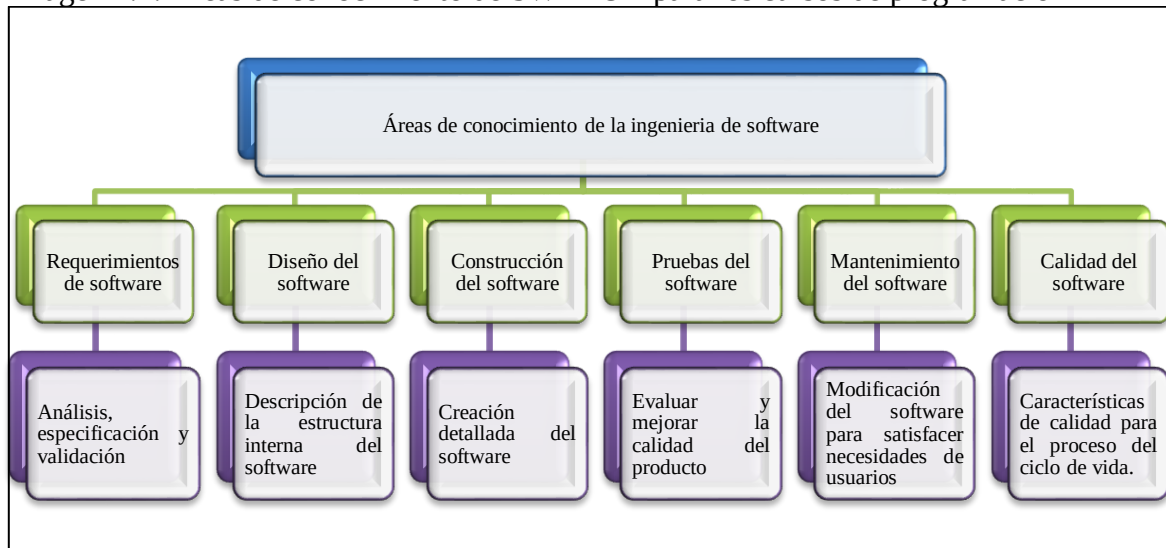
Una buena práctica es el resultado de aplicar una habilidad, herramienta y técnica para poder aumentar el éxito en un proyecto de software, sin embargo, no siempre se aplican de manera uniforme, ya que depende del proyecto (IEEE Computer Society, 2014). Así, las prácticas de ingeniería de software se utilizan principalmente en el desarrollo de software en la industria, no obstante, pueden ser utilizadas para el desarrollo de software en el aula, siendo los cursos de programación una base fundamental en ello, contribuyendo en que los estudiantes puedan desarrollar y potenciar el pensamiento crítico y la resolución de problemas, que es una de las competencias del siglo XXI.

Se debe agregar que, Watts Humphrey desarrolló la metodología Proceso de Software Personal (PSP) es un marco de trabajo que incluye un conjunto de buenas prácticas para planificar, medir y administrar el trabajo de un programador mediante un enfoque disciplinado y estructurado, con 8 fases, para la mejora del proceso personal: planeación, diseño, revisión de diseño, código, revisión de código, compilación, pruebas y posmortem. Las prácticas que se incluyen en este proceso son: declaración de requisitos, diseño conceptual, estimación de tamaño y de tiempo, registro de defectos inyectados y removidos, propuesta de mejora, diseño estático y dinámico, revisión de diseño y de código (Humphrey, 2000).

La IEEE Computer Society (2014) define en su guía SWEBOK V3.0 15 áreas de conocimiento que sirven de línea base para el cuerpo de conocimiento de la ingeniería de software. Seis de estas áreas pueden aplicarse en los cursos de programación, ver Imagen 2.2, requerimientos del software, diseño del software, construcción del software, pruebas del software, mantenimiento del software y calidad del software. Cada área de conocimiento recomienda el uso de buenas prácticas, en la Tabla 2.1 se encuentran ejemplos de éstas, las cuales se pueden utilizar en el desarrollo de software.

La guía tiene como propósito dar a conocer un conjunto de conocimientos de la ingeniería de software que pueden ser aplicados a los proyectos de desarrollo de software en diversos ámbitos.

Imagen 2.2. Áreas de conocimiento de SWEBOK para los cursos de programación



Fuente: Elaboración propia. Basado en (IEEE Computer Society, 2014)

Tabla 2.1. Ejemplos de buenas prácticas por área de conocimiento

Área de conocimiento	Buenas prácticas
Requerimientos de software	• Aplicar técnicas de recolección de datos • Usar técnicas para analizar requisitos • Generar documento de especificación de requisitos • Validar requisitos
Diseño del software	• Realizar Abstracción procesal, de datos y de control • Descomponer y modularizar • Utilizar principios y patrones de diseño • Hacer revisión de diseño • Modelar la descripción estructural y de comportamiento • Diseñar casos de prueba
Construcción del software	• Crear código simple y legible • Usar estándares de construcción • Organizar el código • Documentar el código • Reutilizar el código • Depurar código
Pruebas del software	• Aplicar pruebas de unidad, de integración, del sistema y de aceptación, entre otras. • Hacer uso de técnicas para pruebas caja blanca, caja negra y ad hoc • Aplicar casos de prueba

Área de conocimiento	Buenas prácticas
Mantenimiento del software	• Corregir de defectos • Estimar el coste de mantenimiento • Planificar el mantenimiento • Refactorizar código
Calidad del software	• Identificar y categorizar los defectos • Registrar y eliminar los defectos • Revisar código • Aplicar técnicas analíticas • Utilizar métricas en el proceso de desarrollo

Fuente: Elaboración propia. Basado en (IEEE Computer Society, 2014)

En el ámbito del desarrollo de software, los requerimientos son de suma importancia ya que son esenciales para todo el proceso de desarrollo, “un requerimiento de software se define como una propiedad que debe exhibir el software para resolver algún problema del mundo real” (IEEE Computer Society, 2014, pág. 1.1), para identificarlos se aplican técnicas de recolección de datos a los *stakeholders*, se hace un análisis de requisitos y se describen en un documento de especificación de requerimientos. Sin embargo, los requerimientos de problemas en los cursos de programación se encuentran predefinidos por lo que los estudiantes sólo deben analizar el problema para identificarlos y utilizarlos en el diseño.

El diseño se define como “la actividad del ciclo de vida de ingeniería de software en la que se analizan los requisitos de software para producir una descripción de la estructura interna del software que servirá de base para su construcción.” (IEEE Computer Society, 2014, pág. 2.1). Para el modelado visual, se puede utilizar el estándar Lenguaje Unificado de Modelado (UML) propuesto por James Rumbaugh, Ivar Jacobson y Grady Booch, orientado al desarrollo orientado a objetos y “se usa para especificar, visualizar construir y documentar artefactos de un sistema de software” (Rumbaugh, Jacobson, & Booch, 2000, pág. 3). Con UML se pueden realizar 8 diagramas, ver Tabla 2.2, clasificados en vista estática y vista dinámica. La estática, modela los conceptos del dominio para su implementación y la dinámica describe el comportamiento de los objetos y su comunicación.

Tabla 2.2. Clasificación de diagramas de UML por vista

Vista	Diagrama	Descripción
Estática	Clases	Modela los objetos y sus relaciones mediante clases describiendo la estructura del sistema.
	Casos de uso	Modela la funcionalidad del sistema desde las perspectivas de los usuarios.
	Componentes	Modela la organización de los componentes de un sistema y sus relaciones.
	Despliegue	Modela la arquitectura del sistema mediante distribución física de los componentes.
Dinámica	Estados	Modelan el comportamiento del sistema describiendo los estados posibles de los objetos.
	Actividad	Modela el proceso como un flujo de trabajo interno que describe el comportamiento de una operación.
	Secuencia	Modela el comportamiento del caso de uso mediante un conjunto de mensajes.
	Colaboración	Modela las interacciones entre grupos de objetos para representar un comportamiento.

Fuente: Elaboración propia. Adaptado de (Rumbaugh, Jacobson, & Booch, 2000)

A fin de mejorar la estructura y legibilidad del software, se deben utilizar los principios de diseño SOLID. Los principios SOLID son guías que se aplican en el diseño orientado a objetos con la finalidad de elaborar un código limpio, fácil de mantener y de reutilizar, es decir software de calidad; y son: (Martín, 2003; Martín & Martín, 2006):

1. Responsabilidad única. Todas las clases deben tener una sola responsabilidad y sus métodos deben estar alineados a ésta para que exista una alta cohesión. En caso de que la clase tenga más de una responsabilidad será un motivo de cambio y se deberán separar las responsabilidades en tantas clases como responsabilidades existan.
2. Abierto/Cerrado. Es un principio que está asociado a la herencia en donde una clase está abierta para ser extendida añadiendo nuevas clases lo que conlleva añadir nuevos métodos o extender su comportamiento; y cerrado a la modificación de código ya que al aplicar la herencia no se afectan otras clases, sino se reutiliza el código ya existente. Con este principio se tiene clases mantenibles y reutilizables.
3. Substitución Liskov. Este principio está asociado con un diseño de herencia correcto ya que indica que toda clase derivada se debe utilizar como su clase base y los métodos que

sobrescriben deben tener un comportamiento válido, es decir no pueden lanzar excepciones ni estar vacíos. Además, un cliente de una clase base debe funcionar correctamente con cualquier clase derivada.

4. Segregación de interfaces. Toda interfaz debe determinar el comportamiento de una clase y debe ser específica para cada clase con la finalidad de que no tengan métodos que no necesita. De existir métodos que una clase no requiere, se estaría violando el principio de sustitución Liskov y de responsabilidad única. Es por ello, que se deberán tener tantas interfaces que definan los comportamientos que las clases requieran.

5. Inversión de dependencias. Las clases base no deben depender de las clases derivadas, ambas deben depender de interfaces o clases abstractas las cuales deberán estar en los niveles superiores con la finalidad de que una clase interactúe con otras sin conocer los detalles de su implementación.

Después del modelamiento se debe realizar la codificación haciendo uso de buenas prácticas de código limpio que es un código fácil de escribir, de leer y mantener y al que se le pueda aplicar la refactorización haciéndolo más simple y eliminando redundancias. Martin (2009) recomienda 16 buenas prácticas para crear código limpio y profesional: nombres con sentido, funciones, comentarios, formato, objetos y estructuras de datos, manejo de errores, límites, pruebas de unidad, clases, sistemas, emergencias, concurrencia, refinamiento sucesivo, internos de Junit, refactorización SerialDate y olores y heurística. Empero, solo 6 se utilizarán para el trabajo doctoral. A continuación, se indican las buenas prácticas que se deben aplicar del código limpio (Martin R. C., 2009):

1. Nombres con sentido. Se deben elegir nombres descriptivos que especifiquen el cometido de los conceptos abstractos para clases, métodos, variables, constantes y cualquier elemento que requiera de un identificador, para facilitar su comprensión. Los nombres deben ser claros, fácil de localizar y se debe elegir una palabra por cada concepto abstracto, por lo que se debe hacer uso de una convención de nombres que permita palabras compuestas. Y se deberá considerar que:

- Las clases deben ser sustantivos
- Los métodos deben ser verbos
- Las propiedades deben hacer uso del prefijo get y set
- El uso de una literal será utilizado como variable de control en un bucle

2. Métodos. Todos los métodos deben ser de tamaño reducido, deben cumplir con el principio de responsabilidad única y de abierto/cerrado, no debe existir la duplicación de código y debe tener cero argumentos o a lo más dos. Como el código se lee de izquierda a derecha y de arriba a abajo, los métodos deben aparecer descendentemente con base en su nivel de abstracción. Cuando un método incluya bloques en instrucciones if, else, while y similares deben ser preferentemente para invocar a un método.

3. Comentarios. Las anotaciones en el código deben ser claros, precisos y se deben utilizar solo para expresar algo que por sí sola una línea de código no lo describa, de lo contrario será una redundancia. Cuando se utilice un comentario se debe actualizar al realizar alguna modificación al código y no dejar líneas comentadas de código que ya no se utilizan. Los comentarios legales deben ser incluidos.

4. Formato. El código fuente es un conjunto de líneas de texto que representan sentencias que comunica su intención al ser leído, por ello es necesario tener un estilo de formato en el código que permita su legibilidad facilitando su comprensión y su mantenimiento. Para que un código tenga formato

- El nombre del archivo del código debe ser sencillo y claro.
- Se debe dejar una línea en blanco para indicar la separación entre conceptos independientes como métodos, importaciones y paquetes.
- Las variables deben declararse de la forma más aproximada a su uso.
- Las variables locales deben aparecer en la parte superior de cada método.
- Las variables de control del bucle se deben declarar dentro del bucle.
- Las variables de instancia se declaran en la parte superior de la clase.
- Si un método invoca a otro deben estar verticalmente próximas, es decir el método invocado debe aparecer debajo del que realice la invocación.
- Usar espacio en blanco horizontal entre operadores para acentuar la separación.
- Los argumentos de los métodos se separan ya que son independientes.
- En las ecuaciones los términos se separan mediante espacios en blanco
- Los métodos de la clase, se indentan un nivel a la derecha de la clase.
- Las implementaciones de los métodos se indentan un nivel a su declaración.
- Las implementaciones de bloque se indentan un nivel a la derecha de su bloque.

5. Procesamiento errores. El procesamiento de excepciones es relevante porque permiten controlar errores en tiempo de ejecución, un ejemplo son la validación de datos de entrada. Para un manejo adecuado de las excepciones se deben clasificar con base en su tipo y evitar duplicados, para ello es necesario tener excepciones comunes generando una clase de excepciones.

6. Clases. La estructura de una clase inicia con constantes estáticas públicas, si existen, posteriormente se declaran las variables estáticas privadas seguidas de las variables de instancia privadas y después se incluyen los métodos. No obstante, las clases deben ser de tamaño reducido aplicando el principio de responsabilidad única. Los métodos de la clase deben manipular el mayor número de variables de instancia para que exista cohesión y en caso de no existir, se deberán dividir en más clases las variables y métodos.

Con respecto a las pruebas de software, deben aplicarse en el proceso de desarrollo de software con la finalidad de comparar si las salidas obtenidas son las esperadas tras la ejecución de un programa con base en ciertos datos de entrada. “Las pruebas del software consisten en verificar el comportamiento de un programa dinámicamente a través de un grupo finito de casos de prueba, seleccionados adecuadamente en el dominio de ejecución.” (IEEE Computer Society, 2014). Los casos de prueba deben estar asociados a un conjunto de datos de entrada y condiciones de ejecución para verificar los resultados esperados. Los casos de prueba deben contener la siguiente información (Serna, 2013):

1. Un conjunto de datos de entrada, recibidos por el código bajo prueba desde una fuente externa que puede ser hardware, software o personas.
2. Condiciones de ejecución requeridas para efectuar la prueba.
3. Salidas esperadas, que son los resultados específicos esperados al ejecutar el código bajo prueba.

Los casos de prueba se utilizan con las diferentes pruebas, cada prueba está asociada a una fase del proceso de desarrollo de software como se muestra en la Tabla 2.3, se inicia probando cada componente del programa de manera individual para asegurarse que se cumple con la funcionalidad requerida, posteriormente cada componente se va integrando a fin de identificar conflictos de comunicación entre componentes. A continuación, se prueba que el software completo cumpla con todos los requerimientos especificados y finalmente se realizan pruebas para comprobar que el funcionamiento completo del sistema sea adecuado.

Tabla 2.3. Pruebas por fase en el desarrollo del software

Fase	Prueba	Descripción
Codificación	Unidad	Verifican que la información desde y hacia un componente o módulo del software fluya adecuadamente y que su funcionamiento sea correcto, considerando la lógica de procesamiento interno y estructuras de datos. De esta manera se comprueba que todos los caminos de control importantes se ejecutan de acuerdo con lo especificado dentro de la frontera del módulo.
Diseño	Integración	Consisten en comprobar que cada componente o módulo, previamente probados que constituyen el software completo, se integren de manera incremental verificando que funcionen correctamente y que no exista pérdida de datos entre interfaces.
Análisis de requisitos	Validación	Comprueban que el software desarrollado cumple con todos los requerimientos del cliente, de no ser así se crea una lista de las deficiencias encontradas para corregirlas para que se obtenga un producto que realiza lo que el cliente desea.
Análisis del sistema	Sistema	Son un conjunto de pruebas diversas con propósitos diferentes que comprueban que todo el sistema funcione correctamente verificando que los componentes ya integrados realicen las funciones asignadas, así se detectan errores en el producto ya integrado.

Fuente: Elaboración propia. Basado en (Pressman, Ingeniería del Software un enfoque práctico, 2010)

Es importante destacar que en el paradigma orientado a objetos las pruebas de unidad prueban clases individuales y métodos, debido a que cada clase y cada instancia encapsula sus atributos y sus operaciones, así la prueba está dirigida por las operaciones encapsuladas por la clase y su comportamiento, debido a que se conoce cómo deben verse los datos, cuáles son las firmas de los métodos y cuáles deberían ser los valores y tipos de retorno, ya que se conocen todos los detalles del código que se está probando. Las pruebas de integración se realizan con un conjunto de clases que interactúan entre sí, es decir se prueban las clases dependientes que interactúan con las clases independientes a fin de corroborar su comportamiento y que las interfaces e interacciones entre clases seas adecuadas. Con la prueba del sistema se prueba todo el producto, sin conocer los detalles de su diseño e

implementación, excepto por las entradas y las salidas que se producen. (Pressman, 2010; Dooley, 2017).

Finalmente, en el contexto del trabajo doctoral, las prácticas de ingeniería de software se enfocan al proceso de desarrollo de programas en los cursos de programación, así las prácticas de ingeniería de software deben aplicarse, de manera integral, en el proceso de desarrollo de programas para obtener resultados aceptables, lo cual se logra con un desarrollo con calidad.

2.3 Modelos de calidad del software

Los productos software y los sistemas informáticos se utilizan en la realización de diversas funciones empresariales, comerciales y personales, en donde la satisfacción personal, el éxito empresarial y la seguridad humana y ambiental que proporcionen valor para las partes son los objetivos por cumplir en sistemas de alta calidad (BS ISO/IEC, 2011).

La calidad es un aspecto de gran relevancia que se debe considerar en el desarrollo de software para obtener un producto software de calidad, debido a que los usuarios finales esperan que al ser implementado el producto de software se tengan considerados todos los requerimientos funcionales y no funcionales especificados, que funcione correctamente y que no tenga defectos. Sin embargo, en muchas ocasiones no es así, debido a una a las tres premisas del software que es la calidad.

Existen diversas definiciones de calidad, la ISO/IEC JTC (2000) la define como “La totalidad de las características de una entidad que influyen en su capacidad para satisfacer las necesidades establecidas e implícitas” (pág.20). Chemuturi (2011) la define como un “un atributo de un producto o servicios proporcionado a los consumidores que se ajusta en su totalidad o supera las mejores especificaciones disponibles para ese producto o servicio” (pág.2). En (BS ISO/IEC, 2011) “La calidad de un sistema es el grado en que el sistema satisface las necesidades declaradas e implícitas de sus diversas partes interesadas, y por lo tanto proporciona valor” (pág.8). Otra definición es “grado de desempeño de las principales características con las que debe cumplir un sistema computacional durante su ciclo de vida, dichas características de cierta manera garantizan que el cliente cuente con un sistema confiable, lo cual aumenta su satisfacción frente a la funcionalidad y eficiencia del sistema construido.” (Álvarez C, Alarcón A, & Callejas C, 2016, pág. 237)

Las definiciones anteriores son diferentes, pero en todos los casos la calidad se encuentra en el proceso de desarrollo del producto para satisfacer las necesidades de los usuarios finales. Así, la construcción del software con calidad debe hacer uso de métodos, herramientas y buenas prácticas de ingeniería de software, ver sección 2.3, para un software con calidad. También los conceptos de factores y criterios de calidad se encuentran presentes de manera implícita, los factores son “una condición o características que contribuye activamente a la calidad del software.” (McCall, Richards, & Walters, 1977, pág. 2.1) y los criterios son “atributos del software o proceso de producción del software por el cual los factores pueden ser juzgados y definidos.” (McCall et al., 1977, pág. 2.2).

La calidad tiene cuatro dimensiones (Chemuturi, 2011):

1. Calidad de la especificación. En los proyectos de software se recopilan los requisitos del usuario para definir las especificaciones para el producto a desarrollar, si las especificaciones son débiles, el diseño también lo será teniendo como resultado que el desarrollo y por ende el software sean incorrectos. Por ello es necesario involucrar analistas de negocios o analistas de sistemas para llevar a cabo la ingeniería de requisitos que redacte adecuadamente las especificaciones e incluir estándares internos o de algún organismo para incluir aspectos como funcionalidad, capacidad, uso previsto, confiabilidad y seguridad.
2. Diseño de calidad. El software debe estar bien diseñado y debe cumplir con las especificaciones definidas. Para lograr un buen diseño se deben adoptar estándares, al igual que en la especificación. Para el diseño se puede realizar un diseño conceptual seleccionando un enfoque para una solución y utilizar este enfoque en la ingeniería para elaborar los detalles en la solución. El diseño del software normalmente considera diseño de: funcionalidad, de arquitectura de software, de bases de datos, de interfaz de usuario, del informe y aspectos de seguridad, tolerancia a fallos, mantenibilidad, acoplamiento y cohesión, entre otras.
3. Desarrollo de calidad. Para la construcción del software con calidad es necesario seguir el diseño y adherirse a las pautas de codificación previamente definidas del lenguaje de programación. Las pautas de codificación ayudan a los desarrolladores a escribir código confiable y sin defectos, y contienen convenciones de nomenclatura, de formato, de eficiencia y prevención de defectos. Al igual que el diseño se deben

considerar aspectos de seguridad, eficiencia y tolerancia a fallas, por mencionar algunos.

4. Calidad de conformidad. Se refiere a la calidad del producto terminado que no tiene defectos cuando el software está operando e incluye de manera implícita las tres dimensiones antes mencionadas. Por ello es importante determinar la eficacia de las actividades mediante un conjunto de métricas de calidad que incluyen la eficiencia de eliminación de defectos y la densidad de defectos.

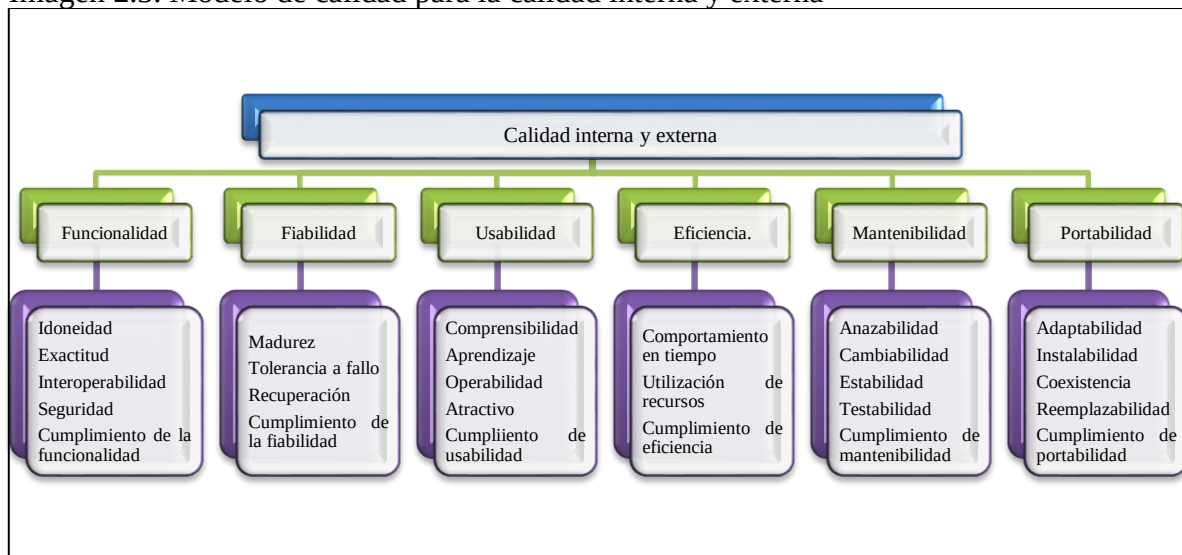
En las cuatro dimensiones se consideran aspectos como funcionalidad, eficiencia y mantenibilidad que son factores de calidad y deben estar acorde a los estándares de la industria del software promulgados por los organismos que establecen y publican normas aplicables al software y que se encuentran establecidos en modelos de calidad del software. Los modelos de calidad de software evalúan el proceso, el producto o la calidad en uso y pueden tener diversos factores de calidad compuestos por criterios que son evaluados por métricas, con el propósito de abordar la evaluación desde lo general a lo particular, y permitir la reducción de la subjetividad en la asignación de un valor, ya sea cuantitativo o cualitativo (Álvarez C et.al., 2016). Algunos modelos de calidad son promulgados por organismos internacionales como la ISO (la Organización Internacional de Normalización) y la IEC (Comisión Electrotécnica Internacional) que forman el sistema especializado para la estandarización mundial.

El estándar ISO/IEC 9126 proporciona un modelo de calidad, un conjunto de métricas de calidad internas, externas y de calidad en uso para medir los atributos de calidad definidos en ISO/IEC 9126-1, que pueden ser utilizadas por desarrolladores, evaluadores, gerentes de calidad y adquirentes para evaluar el producto software (ISO/IEC FDIS, 2000; ISO/IEC JTC1, 2001) y se ha utilizado en el sector educativo y en empresas de desarrollo de software (Callejas-Cuervo, Alarcón-Aldana, & Álvarez-Carreño, 2017). Para garantizar la calidad del producto software, básicamente se pueden seguir dos enfoques, uno es la garantía del proceso mediante el cual se desarrolla el producto y el otro es la evaluación de la calidad del producto final. La calidad del producto debe evaluarse utilizando un modelo de calidad definido que establezca objetivos de calidad para productos intermedios y producto software (ISO/IEC JTC1, 2001). Así, este modelo se constituye de cuatro partes (ISO/IEC FDIS, 2000):

1. ISO/IEC 9126-1, Modelo de calidad. Define los términos para las características y subcaracterísticas de calidad del producto software.
2. ISO / IEC 9126-2, Métricas Externas. Miden el comportamiento de la calidad del producto software durante la etapa de prueba del proceso del ciclo de vida y durante cualquier etapa operativa.
3. IEC 9126-3, Métricas internas. Miden la calidad de los productos intermedios del producto software durante sus etapas de desarrollo.
4. ISO / IEC 9126-4, Métricas de calidad en uso. Miden la efectividad, productividad, seguridad y satisfacción con la que los usuarios al hacer del uso del producto software cumplen con sus objetivos.

En este sentido, ISO / IEC 9126-1 describe un modelo con atributos de calidad que se utilizan como criterios para validar un producto de software mediante métricas de calidad en uso, internas y externas (ISO/IEC JTC1, 2000; Álvarez C et al., 2016). Los atributos se clasifican en una estructura de árbol jerárquica con características (características de calidad) y subcaracterísticas (atributos de calidad), como se muestra en la Imagen 2.3.

Imagen 2.3. Modelo de calidad para la calidad interna y externa



Fuente: Elaboración propia. Basado en (ISO/IEC FDIS, 2000)

ISO/IEC 9126 consta de dos partes (ISO/IEC FDIS, 2000):

1. La primera parte se refiere a la calidad interna y externa que especifica seis características, ver Tabla 2.4, (funcionalidad, confiabilidad, usabilidad, eficiencia, mantenibilidad y portabilidad) que se subdividen en subcaracterísticas, ver Tabla 2.5, para medir las propiedades intrínsecas durante el diseño y la codificación, así como el comportamiento del producto software durante la prueba u operación.
2. La segunda parte se refiere a la calidad en uso, que mide el grado en que un producto software satisface las necesidades de los usuarios de manera efectiva y con seguridad mediante cuatro características (efectividad, productividad, seguridad y satisfacción), ver Tabla 2.6, que son aplicables a todo tipo de software.

Tabla 2.4. Características de calidad de ISO/IEC 9126

Característica	Descripción
Funcionalidad	Capacidad del producto software para proporcionar funciones que satisfagan las necesidades establecidas e implícitas cuando el software se utiliza en condiciones específicas.
Fiabilidad	Capacidad del producto software para mantener un nivel específico de rendimiento cuando se usa en condiciones específicas.
Usabilidad	Capacidad del producto software para ser entendido, aprendido, usado y atractivo para el usuario, cuando se usa bajo condiciones específicas.
Eficiencia	Capacidad del producto software para proporcionar un rendimiento adecuado, en relación con la cantidad de recursos utilizados, bajo las condiciones establecidas.
Mantenibilidad	Capacidad del producto software para ser modificado.
Portabilidad	Capacidad del producto software para transferirse de un entorno a otro.

Fuente: Elaboración propia. Basado en (ISO/IEC FDIS, 2000)

Tabla 2.5. Subcaracterísticas de calidad de ISO /IEC 9126

Subcaracterística	Descripción
Idoneidad	Capacidad del producto software para proporcionar un conjunto apropiado de funciones para tareas específicas y objetivos del usuario.
Exactitud	Capacidad del producto software para proporcionar los resultados correctos con el grado de precisión necesario.
Interoperabilidad	Capacidad del producto software para interactuar con uno o más sistemas.
Seguridad	Capacidad del producto software para proteger la información y los datos de a personas o sistemas no autorizados.

Subcaracterística	Descripción
Cumplimiento de la funcionalidad	Capacidad del producto software para adherirse a estándares, convenciones o regulaciones en leyes y prescripciones similares relacionadas con la funcionalidad.
Madurez	Capacidad del producto software para mantener un nivel específico de rendimiento cuando se usa en condiciones específicas.
Tolerancia a fallos	Capacidad del producto software para mantener un nivel específico de rendimiento en casos de fallas de software.
Recuperación	Capacidad del producto software para restablecer un nivel específico de rendimiento y recuperar los datos directamente afectados en caso de falla.
Cumplimiento de la fiabilidad	Capacidad del producto software para cumplir con los estándares, convenciones o regulaciones relacionadas con la confiabilidad.
Comprensibilidad	Capacidad del producto software para permitir al usuario comprender si el software es adecuado y cómo se puede usar para tareas particulares y condiciones de uso.
Aprendizaje	Capacidad del producto software para permitir al usuario aprender su aplicación.
Operabilidad	Capacidad del producto software para permitir al usuario operarlo y controlarlo.
Atractivo	Capacidad del producto software de ser atractivo para el usuario.
Cumplimiento de la usabilidad	Capacidad del producto software para adherirse a estándares, convenciones, guías de estilo o regulaciones relacionadas con la usabilidad.
Comportamiento en tiempo	Capacidad del producto software para proporcionar tiempos adecuados de respuesta, procesamiento y tasas de rendimiento al realizar su función, en las condiciones establecidas.
Utilización de recursos	Capacidad del producto software para utilizar cantidades y tipos de recursos adecuados cuando el software realiza su función en condiciones establecidas.
Cumplimiento de eficiencia	Capacidad del producto software para adherirse a estándares o convenciones relacionadas con la eficiencia.
Anazabilidad	Capacidad del producto software para ser diagnosticado por deficiencias o causas de fallas del software.
Cambiabilidad	Capacidad del producto software para permitir la implementación de una modificación específica.
Estabilidad	Capacidad del producto software para evitar efectos inesperados por modificaciones del software.
Testabilidad	Capacidad del producto software para permitir la validación del software modificado.

Subcaracterística	Descripción
Cumplimiento de la mantenibilidad	Capacidad del producto software para adherirse a estándares o convenciones relacionadas con la mantenibilidad.
Adaptabilidad	Capacidad del producto software para adaptarse a diferentes entornos específicos sin aplicar acciones o medios distintos de los previstos.
Instalabilidad	Capacidad del producto software para instalarse en un entorno específico.
Coexistencia	Capacidad del producto software para coexistir con otro software independiente en un entorno común que comparte recursos comunes.
Reemplazabilidad	Capacidad del producto software para ser utilizado en lugar de otro producto software especificado para el mismo propósito en el mismo entorno.
Cumplimiento de la portabilidad	La capacidad del producto de software para cumplir con los estándares o convenciones relacionadas con la portabilidad.

Fuente: Elaboración propia. Basado en (ISO/IEC FDIS, 2000)

Tabla 2.6. Características de calidad en uso en ISO/IEC 9126

Característica	Descripción
Efectividad	Capacidad del producto software para permitir que los usuarios logren con precisión e integridad sus objetivos al realizar sus tareas.
Productividad	Capacidad del producto software para permitir que los usuarios consuman recursos en relación con la efectividad lograda.
Seguridad	Capacidad del producto software para permitir niveles aceptables de riesgo durante el funcionamiento normal y en contingencia sin dañar a personas, recursos, negocios y medio ambiente.
Satisfacción	Capacidad del producto software para satisfacer a los usuarios en sus actividades realizadas al hacer uso de este.

Fuente: Elaboración propia. Basado en (ISO/IEC FDIS, 2000; ISO/IEC JTC1, 2001)

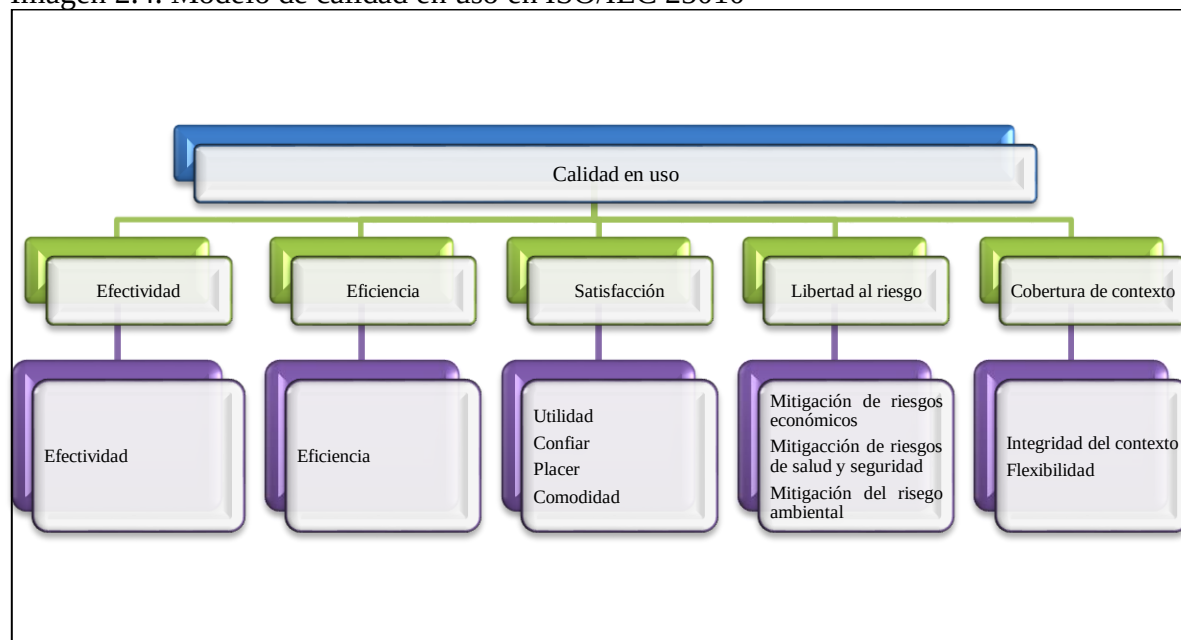
El estándar internacional ISO/IEC 25010 de la serie Square actualmente define tres modelos de calidad que en conjunto sirven de marco para desarrolladores de software, integradores de sistemas, adquirentes, propietarios, mantenedores, contratistas, profesionales de aseguramiento y control de calidad y usuarios para identificar características de calidad de productos software, establecer requisitos, sus criterios de satisfacción y las medidas correspondientes. Y, el estándar ISO/IEC 2502n proporciona medidas de calidad internas y externas para garantizar la calidad y mejora de productos software durante el proceso de desarrollo del ciclo de vida. Las medidas internas miden el grado en que un conjunto de atributos estáticos del producto software satisface las necesidades establecidas e implícitas

al ser usado en condiciones específicas. Las medidas externas, miden el grado en que el producto software permite que el comportamiento satisfaga las necesidades establecidas e implícitas en condiciones específicas (ISO/IEC, 2016).

Los modelos clasifican la calidad del producto software en un conjunto de características de calidad y en algunos casos de subcaracterísticas que pueden medirse directa o indirectamente, y están organizadas de manera jerárquica con una terminología consistente para especificar, medir y evaluar el grado en que el producto software proporciona valor y satisface las necesidades declaradas e implícitas de los diversos interesados. Los modelos de calidad son (BS ISO/IEC, 2011):

1. Modelo de calidad en uso. Se caracteriza por el resultado de la interacción del producto software con las partes interesadas al satisfacer sus necesidades y se compone por cinco características, algunas de las cuales se subdividen en subcaracterísticas, como se muestra en la Imagen 2.4: efectividad, eficiencia, satisfacción, libertad al riesgo y cobertura de contexto. El significado de cada una de las características y subcaracterísticas se encuentran en la Tabla 2.7.

Imagen 2.4. Modelo de calidad en uso en ISO/IEC 25010



Fuente: Elaboración propia. Basado en (BS ISO/IEC, 2011)

Tabla 2.7 Características/Subcaracterísticas Modelo calidad en uso ISO/IEC25010

Característica/Subcaracterística	Descripción
Efectividad	Precisión e integridad con la que los usuarios logran objetivos específicos.
Eficiencia	Recursos de eficiencia gastados en relación con la precisión e integridad con la que los usuarios logran objetivos.
Satisfacción	Grado en que se satisfacen las necesidades del usuario cuando se utiliza un producto o sistema en un contexto de uso específico.
Utilidad	Grado en que un usuario está satisfecho con el logro de los objetivos, incluidos los resultados del uso y sus consecuencias.
Confiar	Grado en que un usuario u otra parte interesada tiene confianza en que un producto o sistema se comportará como se esperaba.
Placer	Grado en que un usuario obtiene placer al satisfacer sus necesidades personales
Comodidad	Grado en que un producto mitiga el riesgo potencial para el estado económico, la vida humana, la salud o el medio ambiente

Fuente: Elaboración propia. Basado en (BS ISO/IEC, 2011)

2. Modelo de calidad del producto se relacionan con las propiedades estáticas del software y las propiedades dinámicas del producto software y clasifica las propiedades de calidad del producto en ocho características: idoneidad funcional, confiabilidad, eficiencia de rendimiento, usabilidad, seguridad, compatibilidad, mantenibilidad y portabilidad. Cada característica se compone de un conjunto de subcaracterísticas, como se muestra en la Tabla 2.8. El significado de las características y subcaracterísticas se encuentran en la Tabla 2.9.
3. Modelo de calidad de datos, es complementario al modelo de calidad del producto que se encuentra en el estándar ISO/IEC 25012 que pertenece a la serie Square, en donde las medidas de calidad de datos dependientes del sistema se centran en la contribución que los datos hacen a la calidad del producto software.

Tabla 2.8. Modelo de calidad del producto en ISO/IEC 25010

Característica	Subcaracterística
Idoneidad funcional	Integridad funcional, corrección funcional, adecuación funcional.
Eficiencia en el desempeño	Comportamiento del tiempo, utilización de recursos, capacidad.
Compatibilidad	Coexistencia, interoperabilidad.
Usabilidad	Idoneidad reconocida, capacidad de aprendizaje, operabilidad, protección contra errores del usuario, interfaz de usuario estética, accesibilidad.
Fiabilidad	Madurez, disponibilidad, tolerancia a fallos, recuperabilidad.
Seguridad	Confidencialidad, integridad, no repudio, responsabilidad, autenticidad.
Mantenibilidad	Modularidad, reutilización, analizabilidad, modificabilidad, comprobabilidad,
Portabilidad	Adaptabilidad, instalabilidad, reemplazabilidad

Fuente: Elaboración propia. Basado en (BS ISO/IEC, 2011)

Tabla 2.9. Características/Subcaracterísticas en ISO/IEC 25010

Característica/Subcaracterística	Descripción
Idoneidad funcional	Grado en que un producto o sistema proporciona funciones que satisfacen las necesidades establecidas e implícitas al ser usado bajo condiciones específicas.
Integridad funcional	Grado en que el conjunto de funciones cubre todas las tareas especificadas y los objetivos del usuario.
Corrección funcional	Grado en que un producto o sistema proporciona los resultados correctos con el grado de precisión necesario.
Adecuación funcional	Grado en que las funciones facilitan el cumplimiento de tareas y objetivos específicos.
Eficiencia en el desempeño	Rendimiento relativo a la cantidad de recursos utilizados en las condiciones establecidas.
Comportamiento del tiempo	Grado en que los tiempos de respuesta, procesamiento y tasas de rendimiento de un producto o sistema al realizar sus funciones, cumplen con los requisitos.
Utilización de recursos	Grado en que las cantidades y tipos de recursos utilizados por un producto o sistema, al realizar sus funciones, cumplen con los requisitos.
Capacidad	Grado en que los límites máximos de un producto o parámetro del sistema cumplen los requisitos.

Característica/Subcaracterística	Descripción
Compatibilidad	Grado en que un producto, sistema o componente puede intercambiar información con otros productos, sistemas o componentes.
Coexistencia	Grado en que un producto puede realizar sus funciones requeridas de manera eficiente compartiendo un entorno y recursos comunes con otros productos.
Interoperabilidad	Grado en que dos o más sistemas, productos o componentes pueden intercambiar y utilizar información.
Usabilidad	Grado en que los usuarios pueden usar un producto o sistema para lograr sus objetivos con efectividad, eficiencia y satisfacción en un contexto de uso específico.
idoneidad reconocida	Grado en que los usuarios pueden reconocer si un producto o sistema es apropiado para sus necesidades.
Capacidad de aprendizaje	Grado en que usuarios específicos pueden usar un producto o sistema para lograr objetivos específicos de aprender a usar el producto o sistema con efectividad, eficiencia, libertad de riesgos y satisfacción en un contexto de uso específico.
Operabilidad	Grado en que un producto o sistema tiene atributos que facilitan su operación y control.
Protección contra errores del usuario	Grado en que un producto o sistema tiene protección contra errores del usuario.
Interfaz de usuario estética	Grado en que una interfaz de usuario permite una interacción agradable y satisfactoria para el usuario.
Accesibilidad	Grado en que un producto o sistema puede ser utilizado por personas con la más amplia gama de características y capacidades para lograr un objetivo específico en un contexto.
Fiabilidad	Grado en que un sistema, producto o componente realiza funciones específicas en condiciones específicas durante un período de tiempo específico.
Madurez	grado en que un sistema, producto o componente cumple con las necesidades de confiabilidad bajo operación normal.
Disponibilidad	grado en que un sistema, producto o componente está operativo y accesible cuando se requiere para su uso.
Tolerancia a fallos	Grado en que un sistema, producto o componente funciona según lo previsto a pesar de la presencia de fallas de hardware o software.

Característica/Subcaracterística	Descripción
Recuperabilidad	Grado en que, en caso de interrupción o falla, un producto o sistema puede recuperar los datos directamente afectados y restablecer el estado deseado del sistema.
Seguridad	Grado en que un producto o sistema protege la información y los datos para que las personas, productos o sistemas tengan el grado de acceso a los datos adecuado a sus tipos y niveles de autorización.
Confidencialidad	Grado en que un producto o sistema garantiza que los datos sean accesibles solo para aquellos autorizados a tener acceso.
Integridad	Grado en que un sistema, producto o componente impide el acceso no autorizado o la modificación de programas de computadora o datos.
No repudio	Grado en que se puede demostrar que las acciones o eventos han tenido lugar, para que los eventos o acciones no puedan ser repudiados más tarde.
Responsabilidad	Grado en que las acciones de una entidad se pueden rastrear de forma exclusiva a la entidad.
Autenticidad	Grado en que se puede demostrar que la identidad de un sujeto o recurso es la reclamada.
Mantenibilidad	Grado de efectividad y eficiencia con el cual un producto o sistema puede ser modificado por los mantenedores previstos.
Modularidad	Grado en que un sistema o programa de computadora se compone de componentes discretos de tal manera que un cambio en un componente tiene un impacto mínimo en otros componentes.
Reutilización	Grado en que un activo puede usarse en más de un sistema, o en la construcción de otros activos.
Analizabilidad	Grado de efectividad y eficiencia con el cual es posible evaluar el impacto en un producto o sistema de un cambio previsto o diagnosticarlo por deficiencias.
Modificabilidad	Grado en que un producto o sistema puede modificarse de manera efectiva y eficiente sin introducir defectos o degradar la calidad del producto existente.
Comprobabilidad	grado de efectividad y eficiencia con el que se pueden establecer criterios de prueba para un sistema, producto o componente y se pueden realizar pruebas para determinar si se han cumplido esos criterios.

Característica/Subcaracterística	Descripción
Portabilidad	Grado de efectividad y eficiencia con el cual un sistema, producto o componente puede transferirse de un hardware, software u otro entorno operativo o de uso a otro.
Adaptabilidad	Grado en que un producto o sistema se puede adaptar de manera efectiva y eficiente para hardware, software u otros entornos operativos o de uso diferentes o en evolución.
Instalabilidad	Grado de efectividad y eficiencia con el que un producto o sistema se puede instalar y / o desinstalar con éxito en un entorno específico.
Reemplazabilidad	Grado en que un producto puede reemplazar a otro producto de software especificado para el mismo propósito en el mismo entorno.

Fuente: Elaboración propia. Basado en (BS ISO/IEC, 2011)

El modelo McCall fue el primer modelo de calidad que proporciona un mecanismo para especificar y medir la calidad de un producto software mediante once factores de calidad, veintitrés criterios de calidad y métricas. Los factores son características del software que contribuye activamente en su calidad, los criterios se definieron como los atributos del software que proporcionan o determinan una característica del producto de software final, ver Tablas 2.10 y 2.11, y las métricas son medidas de los criterios o subcriterios relacionados con los factores de calidad (McCall et al., 1977; Álvarez C et al., 2016).

Tabla 2.10. Factores de calidad de McCall

Factores	Descripción
Exactitud	Grado en que un programa satisface sus especificaciones y cumple los objetivos del usuario.
Fiabilidad	Medida en que se puede esperar que un programa realice su función prevista con la precisión requerida
Eficiencia	La cantidad de recursos informáticos y código requeridos por un programa para realizar una función.
Integridad	Medida en que se puede controlar el acceso a software o datos por parte de personas no autorizadas.
Usabilidad	Esfuerzo requerido para aprender, operar, preparar entradas e interpretar salidas de un programa.

Factores	Descripción
Mantenimiento	Esfuerzo requerido para localizar y corregir un error en un programa operativo.
Testabilidad	Esfuerzo requerido para probar un programa para asegurar que realiza su función prevista.
Flexibilidad	Se requiere esfuerzo para modificar un programa operativo.
Portabilidad	Esfuerzo requerido para transferir un programa de una configuración de hardware y / o entorno de sistema de software a otro.
Reusabilidad	Grado en el que un programa puede usarse en otras aplicaciones.
Interoperabilidad	Esfuerzo requerido para acoplar un sistema con otro.

Fuente: Elaboración propia. Basado en (McCall et al., 1977)

Tabla 2.11. Criterios de calidad de McCall

Criterio	Definición
Trazabilidad	Esos atributos del software que proporcionan un hilo desde los requisitos hasta la implementación con respecto al desarrollo específico y el entorno operativo.
Compleitud	Esos atributos del software que proporcionan la implementación completa de las funciones requeridas.
Consistencia	Los atributos del software que proporcionan un diseño uniforme y técnicas de implementación y notación
Exactitud	Esos atributos del software que proporcionan la precisión requerida en los cálculos y resultados.
Tolerancia de error	Esos atributos del software que proporcionan continuidad de operación en condiciones no nominales.
Sencillez	Esos atributos del software que proporcionan la implementación de funciones de la manera más comprensible.
Modularidad	Esos atributos del software que proporcionan una estructura de módulos altamente independientes
Generalidad	Esos atributos del software que proporcionan amplitud a las funciones realizadas.
Expansibilidad	Esos atributos del software que proporcionan la expansión de los requisitos de almacenamiento de datos o funciones computacionales.
Instrumentación	Esos atributos del software que proporcionan la medición del uso o la identificación de errores.
Autodescriptividad	Esos atributos del software que proporcionan una explicación de la implementación de una función
Eficiencia de ejecución	Esos atributos del software que proporcionan un tiempo de procesamiento mínimo.

Criterio	Definición
Eficiencia de almacenamiento	Esos atributos del software que proporcionan requisitos mínimos de almacenamiento durante la operación.
Control de acceso	Esos atributos del software que proporcionan el control del acceso de software y datos
Auditoria de acceso	Esos atributos del software que proporcionan una auditoría del acceso de software y datos.
Operabilidad	Los atributos del software que determinan la operación y los procedimientos relacionados con la operación del software.
Formación	Esos atributos del software que proporcionan la transición desde la operación actual o la familiarización inicial
Comunicatividad	Aquellos atributos del software que proporcionan entradas y salidas útiles que pueden asimilarse.
Independencia del sistema de software	Esos atributos del software que determinan su dependencia del entorno del software (sistemas operativos, utilidades, rutinas de entrada / salida, etc.
Independencia de máquina	Esos atributos del software que determinan su dependencia del sistema de hardware.
Comunidad de comunicaciones	Esos atributos del software que la interoperabilidad proporciona el uso de protocolos estándar y rutinas de interfaz.
Comunidad de datos	Esos atributos del software que proporcionan el uso de representaciones de datos estándar.
Concisión	Esos atributos del software que proporcionan la implementación de una función con una cantidad mínima de código

Fuente: Elaboración propia. Basado en (McCall et al., 1977)

El conjunto de métricas establecidas proporciona una cobertura integral de las características de un producto software, que sirven de guía para los evaluadores, el personal de garantía de calidad y los esfuerzos independientes de verificación y validación. Este modelo ha sido implementado en diferentes contextos como el sector agrícola, desarrollo ágil de software y en empresas del sector del software (Callejas-Cuervo et al., 2017).

Los factores de calidad se agrupan en tres perspectivas, ver Tabla 2.13, (McCall et al., 1977):

1. Operación del producto (exactitud, fiabilidad, eficiencia, integridad y usabilidad)
2. Revisión del producto (Mantenibilidad, flexibilidad, testabilidad)
3. Transición del producto (Portabilidad, reutilización e interoperabilidad)

Tabla 2.12. Factores y criterios de calidad con base en las perspectivas de McCall

Perspectiva	Factores de calidad	Criterios de calidad
Operación del producto	Exactitud	Trazabilidad, completitud y consistencia
	Fiabilidad	Consistencia, exactitud, tolerancia de error, sencillez,
	Eficiencia	Eficiencia de ejecución, eficiencia de almacenamiento
	Integridad	Control de acceso, auditoria de acceso
	Usabilidad	Operabilidad, formación, comunicatividad
Revisión del producto	Mantenibilidad	Consistencia, sencillez, modularidad, autodescriptividad, concisión
	Flexibilidad	Modularidad, generalidad, expansibilidad, autodescriptividad
	Testabilidad	Sencillez, modularidad, instrumentación, autodescriptividad
Transición del producto	Portabilidad	Modularidad, autodescriptividad, independencia del sistema de software
	Reutilización	Modularidad, generalidad, autodescriptividad, Control de acceso
	Interoperabilidad	Modularidad, independencia de máquina, comunidad de comunicaciones y comunidad de datos

Fuente: Elaboración propia. Basado en (McCall et al., 1977)

Otro modelo de calidad es el de Bohem que proporciona un marco claro y bien definido para evaluar los problemas asociados con la calidad del software, a través de conjuntos de definiciones, distinciones, pautas y experiencias consistentes y de apoyo mutuo, que sirve como una base para futuros refinamientos y extensiones (Bohem, Brown, & Lipow, 1976). Y ha sido utilizado en la evaluación de costos, en la definición de un lenguaje de programación y en empresas del sector salud, educativo y riesgos (Callejas-Cuervo et al., 2017).

El modelo de Bohem tiene una estructura jerárquica que define un conjunto de características y subcaracterísticas de la calidad del software bien definidas y diferenciadas, ver Tabla 2.13 y Tabla 2.14. En su nivel superior se reflejan los usos reales de la calidad del software (utilidad, mantenibilidad, portabilidad) y en nivel inferior proporciona un conjunto de características primitivas que se combinan en conjuntos de condiciones necesarias para las características de nivel intermedio (portabilidad, confiabilidad, eficiencia, usabilidad,

ingeniería humana, comprensibilidad y flexibilidad) y se encuentran relacionadas con las métricas para las evaluaciones que se pueden realizar (Bohem et al., 1976).

Tabla 2.13. Características y subcaracterísticas en Bohem

Característica/Subcaracterística	Descripción
Accesibilidad	El código posee accesibilidad en la medida en que facilita el uso selectivo de sus partes.
Responsabilidad	El código posee responsabilidad en la medida en que se puede medir su uso.
Exactitud	El código posee precisión en la medida en que sus resultados son lo suficientemente precisos para satisfacer su uso previsto.
Aumentabilidad	El código posee la capacidad de aumento en la medida en que puede acomodar la expansión en funciones computacionales de componentes o requisitos de almacenamiento de datos.
Comunicatividad	El código posee comunicatividad en la medida en que facilita la especificación de entradas y proporciona salidas fáciles de asimilar y útiles.
Integridad	El código posee la integridad en la medida en que todas sus partes están presentes y cada parte está completamente desarrollada.
Concisión	El código posee concisión en la medida en que no hay información excesiva.
Consistencia	El código posee consistencia interna en la medida en que contiene notación, terminología y simbología uniformes dentro de sí mismo, y consistencia externa en la medida en que el contenido sea trazable a los requisitos.
Independencia del dispositivo	El código posee la independencia del dispositivo en la medida en que puede ejecutarse en configuraciones de hardware de computadora que no sean las actuales.
Eficiencia	El código posee eficiencia en la medida en que cumple su propósito sin desperdicio de recursos.
Ingeniería humana	El código posee ingeniería humana en la medida en que cumple su propósito sin perder el tiempo y la energía de los usuarios, ni degradar su moral.
Legibilidad	El código posee legibilidad en la medida en que su función se distingue fácilmente al leer el código.

Característica/Subcaracterística	Descripción
Mantenimiento	El código posee mantenimiento en la medida en que facilita la actualización para satisfacer nuevos requisitos o corregir deficiencias.
Modificabilidad	El código posee la capacidad de modificación característica en la medida en que facilita la incorporación de cambios, una vez que se ha determinado la naturaleza del cambio deseado
Portabilidad	El código posee portabilidad en la medida en que puede ser operado fácilmente en configuraciones de computadora diferentes a la actual.
Fiabilidad	El código posee confiabilidad en la medida en que realice sus funciones previstas satisfactoriamente.
Robustez	El código posee robustez en la medida en que puede continuar funcionando a pesar de alguna violación de los supuestos en su especificación.
Autocontención	El código posee autocontención en la medida en que realiza todas sus funciones explícitas e implícitas dentro de sí mismo.
Autodescriptividad	El código es autodescriptivo en la medida en que contiene suficiente información para que un lector determine o verifique sus objetivos, suposiciones, restricciones, entradas, salidas, componentes y estado de revisión.
Estructuración	El código está estructurado en la medida en que posee un patrón definido de organización de sus partes interdependientes.
Testabilidad	El código tiene capacidad de prueba en la medida en que facilita el establecimiento de criterios de verificación y respalda la evaluación de su desempeño.
Comprensibilidad	El código posee comprensión en la medida en que su propósito sea claro para el inspector.
Usabilidad	El código es usable en la medida en que es confiable, eficiente y diseñado por humanos.

Fuente: Elaboración propia. Basado en (Bohem et al., 1976).

Tabla 2.14. Relación de los niveles en Bohem

Nivel superior (Utilidad general)	Nivel intermedio (Factores)	Nivel inferior (criterios)
Utilidad general	Portabilidad	Independencia del dispositivo
		Autocontención
Utilidad	Fiabilidad	Autocontención
		Exactitud
		Compleitud

		Robustez / Integridad
		Consistencia
	Eficiencia	Responsabilidad
		Eficiencia del dispositivo
		Accesibilidad
	Ingeniería humana	Robustez / Integridad
		Accesibilidad
		Comunicatividad
Mantenibilidad	Comprobabilidad	Responsabilidad
		Accesibilidad
		Comunicatividad
		Autodescripción
		Estructuración
	Comprensibilidad	Consistencia
		Autodescripción
		Estructuración
		Concisión
		Legibilidad
	Modificabilidad	Estructuración
		Aumentable

Fuente: Elaboración propia. Basado en (Bohem et al., 1976).

El modelo de calidad de Dromey es un modelo genérico que ofrece una base constructiva sólida que proporciona orientación directa para comprender e incorporar la calidad en el software tanto de arriba hacia abajo, durante el diseño, como de abajo hacia arriba, durante implementación, con base en propiedades y características, ver Tabla 2.15. En la calidad de arriba hacia abajo se identifican qué propiedades deben satisfacerse para cada forma estructural para construir un atributo de calidad de alto nivel; esta perspectiva es más útil para los diseñadores que tienen la responsabilidad de especificar y factorizar atributos de calidad de alto nivel en el diseño de software. En la calidad de abajo hacia arriba se identifican las propiedades de calidad importantes que vinculan entidades con atributos de calidad de alto nivel para cada componente que cumplan con las propiedades particulares del producto; esta perspectiva puede proporcionar consejos concretos que son de vital importancia para implementar el proceso de creación de calidad en el software (Dromey, 1995).

Tabla 2.15. Propiedades y característica del modelo de Dromey

Propiedades/Características	Descripción
Corrección	Un programa debe tener los requisitos genéricos mínimos para su corrección con base en la computabilidad, la integridad y la consistencia.
Estructuración	Se centran en la forma en que las declaraciones individuales y los componentes de la declaración se implementan y la forma en que las declaraciones y los bloques de declaración se componen, se relacionan entre sí y se utilizan.
Modularidad	Abordan en gran medida los problemas de diseño de alto nivel asociados con los módulos y cómo interactúan con el resto de un sistema.
Descriptivas	Describen el software agregando comentarios explícitos a un programa para documentar cómo la implementación realiza su funcionalidad deseada mediante la manipulación de variables con propiedades prescritas.
Funcionalidad	Un programa debe funcionar adecuadamente y debe ser preciso con base en su especificación.
Fiabilidad	Un programa puede ser correcto con respecto a su especificación, satisfacer sus requisitos funcionales y las entradas deben satisfacer la condición previa esperada.
Usabilidad	La calidad de la interfaz de usuario, su diseño y características de rendimiento deben ser definidas adecuadamente en los programas.
Eficiencia	La redundancia computacional y lógica son factores importantes que afectan la eficiencia de un programa, por lo que debe existir una coincidencia entre la estructura de control del programa y la estructura de datos.
Mantenibilidad	El software debe ser fácil de mantener por lo que también debe ser fácil de entender.
Portabilidad	Un programa o sistema es portátil si requiere poco o ningún cambio para compilarlo y ejecutarlo en otros sistemas.
Reusabilidad	Software que tiene propiedades que permiten su uso en otros contextos de aplicación sin cambios

Fuente: Elaboración propia. Basado en (Dromey, 1995).

Además, el modelo permite asegurar la calidad del software y la clasificación sistemática de defectos de calidad al asociar un conjunto de propiedades de calidad, atributos, con cada una de las formas estructurales que se utilizan para definir los componentes y sus declaraciones en un lenguaje de programación utilizado para definir requisitos individuales. Las propiedades en orden de precedencia son (Dromey, 1995):

1. Propiedades de corrección (requisitos genéricos mínimos para la corrección).
2. Propiedades de estructuración (bajo nivel, problemas de diseño intramódulo).
3. Propiedades de modularidad (problemas de diseño de alto nivel entre módulos)
4. Propiedades descriptivas (varias formas de documentación específica)

Estas propiedades de calidad están a su vez vinculadas a los atributos de calidad de alto nivel de la norma ISO-9126. Así, este modelo proporciona un proceso explícito para construir propiedades de calidad en el software al establecer vínculos entre las características tangibles del producto y los atributos de calidad menos tangibles (Dromey, 1995), ver Tabla 2.16.

Tabla 2.16. Propiedades y su asociación con las características en el modelo de Dromey

Propiedades	Características
Corrección	Funcionalidad, fiabilidad, usabilidad, mantenibilidad, portabilidad, reusabilidad
Estructuración	Funcionalidad, fiabilidad, usabilidad, eficiencia, mantenibilidad, portabilidad, reusabilidad.
Modularidad	Fiabilidad, mantenibilidad, portabilidad, reusabilidad.
Descriptivas	Funcionalidad, fiabilidad, usabilidad, mantenibilidad, portabilidad, reusabilidad.

Fuente: Elaboración propia. Basado en (Dromey, 1995).

Conclusión.

En este capítulo se mostraron las fases generales que se deben seguir en la programación, las buenas prácticas de ingeniería de software que pueden ser utilizadas al desarrollar un programa y los principales modelos de calidad para evaluar el software, que son las temáticas bajo las cuales se desarrolla el presente trabajo doctoral. Así, en los cursos de programación se deben considerar las buenas prácticas de ingeniería de software y los criterios de calidad de los principales modelos de calidad del software con la finalidad de elegir aquellas necesarias que contribuyan en la comprensión de los conceptos y su aplicación al desarrollar o crear programas con calidad, que son dificultades que presentan los estudiantes en los cursos de programación y que se solicitan en el campo laboral. Es por ello por lo que las universidades e instituciones de educación superior deben preocuparse por fomentar su uso en los cursos de programación.

Capítulo 3. Contexto de la enseñanza de la programación en las universidades.

Introducción

Como consecuencia de las deficiencias a las que se enfrentan los estudiantes en los cursos de programación, ver sección 1.1, en este capítulo se hizo la revisión de literatura actualizada para conocer el estatus que guardan los cursos de programación en las universidades, centrándose en los cursos de programación orientados a objetos.

Además, como en los cursos de programación orientada a objetos, se requiere comprender y aplicar un conjunto de conceptos en la resolución de problemas, que son requisitos para el desarrollo de aplicaciones de mayor magnitud, se indagaron diversos esfuerzos que contribuyan en mejorar la comprensión de conceptos y su aplicación. Así, como aquellos que hagan uso de buenas prácticas y modelos de calidad en los cursos de programación orientada a objetos, debido a que el desarrollo de programas debe estar orientado a la calidad. Lo anterior, es importante porque el fenómeno de las dificultades que se presentan en los cursos de programación no ha sido resuelto.

3.1 Estatus de la programación en el mundo

La programación se encuentra presente de manera implícita en diversas actividades personales y empresariales al hacer uso de software de aplicación y de manera explícita en la industria del software y en los planes de estudios de universidades que ofrecen carreras relacionadas con ciencias de la computación en donde se imparten cursos de programación. Por consiguiente, la enseñanza de la programación requiere de estrategias que permitan a los estudiantes adquirir el conocimiento teórico para comprender lo que es la programación y seguir un proceso de elaboración adecuado o ciclo de vida que sirva de directriz a los estudiantes para que apliquen los conocimientos que les permita crear un programa con calidad. Sin embargo, los estudiantes se enfrentan a diversas dificultades, ver problemática del capítulo 1, haciendo que los cursos de programación sean difíciles (Mechan, Phalp, & Grimm, 2016; Krusche, Berisha, & Bruegge, 2016; Thuné & Eckerdal, 2019), frecuentemente fallen (Kofi Sarpong, John Kingsley, & Prince Yaw, 2013; Özmen & Altun, 2014; Bosse & Gerosa, 2016) y tengan altas tasas de fracaso (Insuasti, 2016; Vihavainen, Airaksinen, & Watson, 2014; Ayalew, Tshukudu, & Lefoane, 2018; Alammery, 2019;

Kanaparan, Cullen, & Mason, 2019) a nivel mundial, pero existen pocas investigaciones que indican la tasa de fracaso.

En este sentido, la media mundial de fracaso en los cursos introductorios de programación, CS1, fue del 32.3% con fundamento en la publicación de artículos entre 1960 y junio de 2013 (Watson & Li, 2014), en 2006 la tasa de fracaso a nivel mundial fue del 33% considerando 63 universidades de 15 países (Bennedsen & Caspersen, 2007) y en 2017 fue del 28% con base en 161 universidades (Bennedsen & Caspersen, 2019). En la universidad de los Andes en Venezuela tienen un índice de reprobación que oscila entre el 68.6% y 86.2% de los estudiantes matriculados entre 2015 y 2016 (Pérez & Azuaje, 2019), en la universidad de Regis en Denver se tiene una tasa de fracaso del 50% hasta el 2018 (Lindoo, 2018) y en la Universidad de Oriente en Cuba el índice de aprobados no supera el 74% (Almarales Sarasola & García Fernández, 2019) .

Se debe agregar que, en México, en la Universidad Autónoma Metropolitana Unidad Cuajimalpa, las materias que presentan altos índices de reprobación son programación, programación estructurada y programación orientada a objetos (UAM, 2013). En el Instituto Tecnológico de Chihuahua, Instituto Tecnológico de Cd. Juárez e Instituto Tecnológico de Mexicali, las materias con mayor porcentaje de reprobados son programación, fundamentos de programación y estructura de datos (Olivares Rojas et al., 2015). En el Instituto Tecnológico de Ciudad Guzmán, se ha detectado que año con año la matrícula en las carreras afines con computación ha disminuido aumentando el índice de reprobación y deserción (Nava & Pinto, C.A, 2013). Y en el área de conocimiento “formación tecnológica” de la Universidad Tecnológica de Jalisco, las materias relacionadas con tecnología tienen un índice de reprobación del 60%, y dentro de esta, las materias de programación y desarrollo de software tienen el 70% de reprobados (Macías Brambila et al., 2016).

Como se observa, las dificultades para aprender a programar son un fenómeno universal y no son específicas del curso, escuela o país (Figueiredo, Gomes, & García-Peñalvo, 2016).

3.2 Enseñanza de la programación orientada a objetos

Uno de los cursos más populares en las universidades de todo el mundo es el enfoque orientado a objetos, ver sección 2.1, empero como todo curso de programación son un desafío para los estudiantes especialmente en un nivel introductorio de programación. Las

dificultades que presentan los estudiantes además de las mencionadas en la problemática, es la comprensión y aplicación de conceptos para la resolución de problemas (Sánchez-García, Urías-Ruiz, & Gutiérrez-Herrera, 2015; Kunkle & Allen, 2016; Mechan et al., 2016; Olie et al., 2017; Qian & Lehman, 2017; Shefer, Nosova, & Lebedeva, 2018; Wong, Maizatul, & Tan, 2018; Almarales Sarasola & García Fernández, 2019; Boudia, Bengueddach, & Haffaf, 2019; Sun, Wu, & Liu, 2020), el entorno de desarrollo y el lenguaje de modelado (Sánchez-García et al., 2015; Almarales Sarasola & García Fernández, 2019), la abstracción, la descripción y las soluciones inadecuadas del problema, la codificación sin estándares definidos y el mal uso de los términos (Machín, 2018).

De igual manera, los métodos tradicionales de enseñanza y aprendizaje en el aula no son efectivos y son insuficientes (Wong et al., 2018) para resolver las dificultades, por ello y con la finalidad de contribuir en la mejora de los cursos de programación orientada a objetos, facilitando una experiencia de aprendizaje que ayude a los estudiantes en la comprensión de conceptos, adquisición de habilidades y planteen soluciones a problemas (Boudia et al., 2019) se han desarrollado propuestas con diferentes perspectivas derivadas de investigaciones internacionales.

Las investigaciones internacionales citadas para mejorar la comprensión y aplicación de conceptos de la programación orientada a objetos son ejemplos de los últimos cuatro años. Algunos autores han adoptado el uso de aplicaciones interactivas como los tutores inteligentes (Machín, 2018) y la gamificación (Norero Ochoa, 2017; Thongmak, 2018). Otros autores diseñan, desarrollan e implementan herramientas de aprendizaje como juegos (Wong et al., 2018), herramientas de animación interactiva (Sakeenah & Mabroukah, 2019), tutoriales interactivos (Almarales Sarasola & García Fernández, 2019) y herramientas de aprendizaje (Olier et al., 2017). Otros enfoques son el diseño e implementación de estrategias de aprendizaje (Boudia et al., 2019; Jusuf, Ibrahim, & Suparman, 2019), aplicación de pruebas automatizadas, gestión de configuración y entorno de desarrollo integrado en cursos masivos (Torchiano & Bruno, 2018) y el desarrollo de herramienta para la evaluación de la comprensión de conceptos (Kunkle & Allen, 2016).

La investigación que incluye buenas prácticas en el proceso de desarrollo en estudiantes universitarios es la de Shefer, et al (2018) que propone una metodología para la enseñanza de la programación orientada a objetos basada en el análisis de los requisitos de los estándares

educativos y profesionales. Por lo tanto, para que los estudiantes de pregrado dominen el programa educativo básico (OOP) y formen las calificaciones correspondientes propuestas por el estándar profesional de las funciones y competencias laborales establecidas por el estándar educativo, es necesario construir una cierta trayectoria de capacitación, que debe ser reflejado en el plan de estudios en forma de una secuencia de disciplinas y en el programa de la disciplina en forma de una secuencia de secciones sobre el estudio de lenguajes de programación. Al finalizar el curso, los estudiantes de pregrado desarrollan competencias en el campo de la programación incluidos los métodos para diseñar, analizar y crear complejos sistemas de software. Sin embargo, sólo se presentó el enfoque para organizar los cursos de programación en la universidad estatal de los Urales del Sur en Rusia.

3.3 Calidad en la programación orientada a objetos

Un aspecto importante de la programación orientada a objetos es la calidad, existen investigaciones para la calidad en el desarrollo de software en este paradigma (González Reyes, Hernández González, & André Ampuero, 2016; Janicijevic, Krsmanovic, Zivkovic, & Lazarevic, 2016; Wojszczyk & Wójcik , 2016; Stevenson & Wood, 2018; Ifeanyi, Chukwuemeka, Uzoma, Chioma, & Okemiri , 2019; Kumar, Ranjan, & Gupta, 2020) y métodos que garantizan la calidad en cada etapa del ciclo de vida para su desarrollo (Wojszczyk & Wójcik , 2016).

En los cursos de programación orientado a objetos también es importante la calidad con la que se desarrollan los programas, una manera de comprender el rendimiento de aprendizaje de los estudiantes son las pruebas de software para descubrir fallas, y pueden usarse como una forma efectiva de validar objetivamente la calidad del software (Sun et al., 2020). Sin embargo, en las universidades los cursos de programación se centran en aplicar los conceptos teóricos y en la corrección funcional, descuidando el tema del código limpio, ver sección 2.3, y mantenible (Dietz, Manner, Harrer, & Lenhard, 2018). El código limpio incluye una gama amplia de consejos centrados en el diseño y el código destinados a hacer que los programas y sus diseños sean legibles, más fáciles de entender y cambiar (Stevenson & Wood, 2018; Ghanem , HamaAli, Talal, & Layth, 2018). En este sentido, Dietz et al., (2018) presentaron el concepto de código limpio haciendo énfasis en la escritura y revisión de código como método didáctico en la enseñanza de la programación y Ghanem et al, (2018) proponen una

herramienta para la ingeniería de software con un modelo de calidad basado en métricas estándar de complejidad de la programación orientada a objetos incluyendo un atributo de código limpio. Para el diseño existen los principios SOLID que cubren el diseño de paquetes y clases, minimizando sus dependencias, dando calidad en el diseño (Stevenson & Wood, 2018), pero la cantidad de defectos de sintaxis, semántica y pragmática son alarmantes en los modelos orientados a objetos en los modeladores novatos, por lo que es necesario desarrollar técnicas educativas para la prevención de defectos y mecanismos de eliminación lo cual es posible si se conocen los defectos y sus causas (Ma, 2017). En relación con ello, en el Proceso de Software Personal (PSP) se lleva a cabo la administración de defectos durante el proceso de desarrollo de un programa, además demuestra los potenciales para lograr el desarrollo de alta calidad para ingenieros de software individuales, quienes pueden mejorar la calidad del código fuente. PSP se ha utilizado con fines educativos para proporcionar a los estudiantes un proceso de software de superación personal, pudiendo dejar una gran parte de los defectos a la prueba de la unidad (Rong et al., 2016).

Conclusión.

En este capítulo se ha contextualizado la situación de la programación en las universidades, detectando como fenómeno común que existen deficiencias en la programación aunado al desconocimiento de buenas prácticas que permiten desarrollar programas regidos bajo un proceso. Por consiguiente, los cursos introductorios de programación presentan índices de reprobación relativamente altos, así como en los cursos de programación orientada a objetos, lo que causa estrés y desánimo para los estudiantes.

En este sentido, se han realizado propuestas para mejorar los cursos de programación orientada a objetos, como son tutores inteligentes, gamificación y desarrollo de herramientas de aprendizaje e interactivas. Sin embargo, solo dos investigaciones son las que se relacionan con el modelo propuesto en este trabajo doctoral, la primera es la metodología para la enseñanza de la programación orientada a objetos que integra un conjunto de buenas prácticas, pero solo se presenta un enfoque de organización integral para los cursos de programación de una universidad en Rusia (Shefer et al., 2018). La segunda es el método didáctico que incorpora el concepto de código limpio que hace énfasis en la escritura y revisión de código, el cual ha tenido éxito por seis años (Dietz et al., 2018). Sin embargo, de

las investigaciones citadas en este capítulo no existe una que integre buenas prácticas de software como los principios SOLID y el código limpio, así como criterios para evaluar la calidad durante todo el desarrollo del programa.

Capítulo 4. Metodología.

Introducción.

En este capítulo se expone la metodología que se siguió en este trabajo doctoral, la cual se divide en dos partes. En la primera parte se indica la metodología de la investigación para la obtención de resultados sobre las mejores prácticas de ingeniería de software utilizadas por los docentes en los cursos de programación en las universidades públicas en México, así como para obtener los resultados para definir los criterios de calidad que puedan aplicarse en el desarrollo de programas en los cursos de programación. En la segunda parte, se detalla la metodología para evaluar el modelo MoACPOO propuesto.

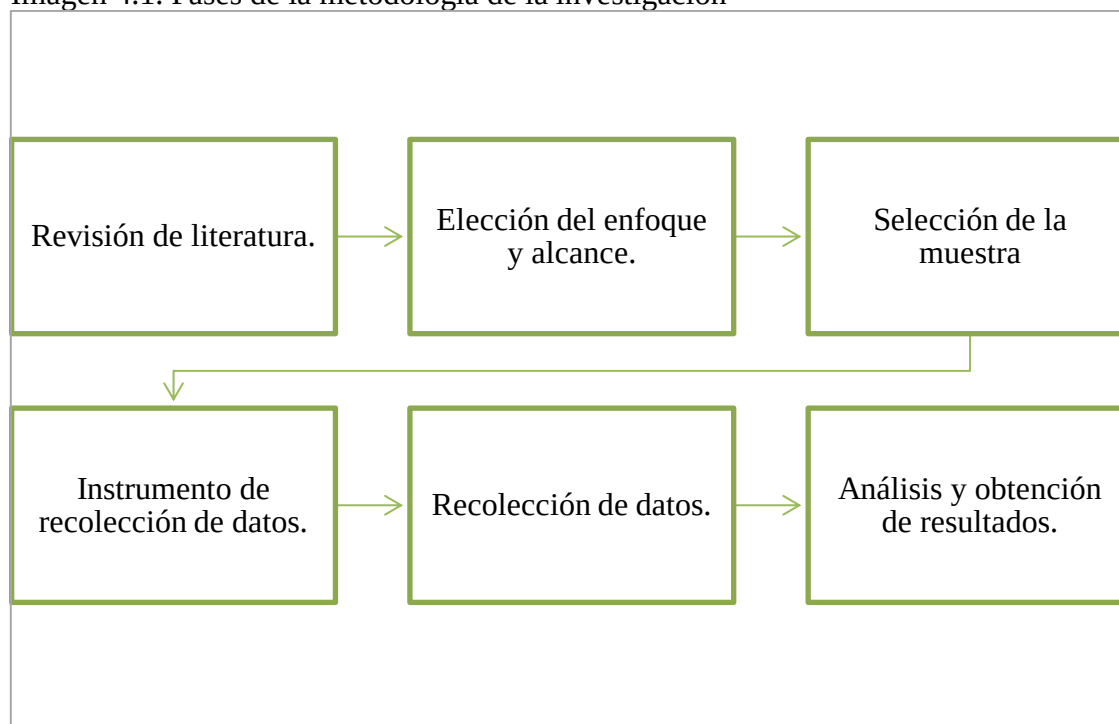
La metodología de la investigación de este trabajo doctoral se realizó con un enfoque cuantitativo y con un alcance descriptivo, utilizando como instrumento de recolección de datos un cuestionario que fue validado por el grado de concordancia entre jueces y con el coeficiente de Alfa de Cronbach. El cuestionario se aplicó a una muestra de 87 docentes de 38 universidades públicas incorporadas a ANUIES, que ofertan carreras que incluyen en sus planes de estudio materias afines con programación. Sin embargo, el análisis de los datos para identificar las mejores prácticas de ingeniería de software utilizadas por los docentes en los cursos de programación en las universidades públicas en México y para definir los criterios de calidad que puedan aplicarse en el desarrollo de programas en los cursos de programación con base en los principales modelos de calidad del software, se encuentran en los capítulos 5 y 6, respectivamente.

Con respecto a la metodología para evaluar el modelo MoACPOO se siguió un enfoque cuantitativo y cuasiexperimental, con un muestreo no probabilístico. El modelo se utilizó en un entorno real en un curso de programación orientada a objetos en 2019 considerando un grupo de control y un grupo experimental. Para el análisis de datos se utilizó estadística descriptiva y la prueba de t student para muestras independientes, este último para comprobar si el modelo propuesto es viable. Los resultados de la evaluación del modelo se encuentran en el capítulo 8.

4.1 Metodología de la investigación

La metodología de la investigación se conformó de seis fases, ver Imagen 4.1, Con esta metodología se obtuvieron los resultados de las mejores prácticas de ingeniería de software y los criterios de calidad que se permearon en un ciclo de vida que es el sustento para el modelo propuesto llamado MoACPOO.

Imagen 4.1. Fases de la metodología de la investigación



Fuente: Elaboración propia.

4.1.1 Revisión de literatura

Para poder obtener los referentes teóricos relacionados con la investigación, se seleccionaron los lugares de acceso a la información, físicos y con búsquedas de acceso mediante Internet, consultándose y seleccionando los documentos para la elaboración del marco teórico y contextual. Los lugares de acceso físicos fueron biblioteca del Centro Universitario UAEM Texcoco y biblioteca del Instituto Tecnológico de Monterrey campus Ciudad de México. Las búsquedas de acceso mediante internet fueron consultadas en ARPN Journal of Systems and Software, Association for Computing Machinery (ACM), base de datos arXiv.org, colección Scielo México, directory of Open Access Journals, DSapce@MIT, Educational Research

International, ELSIEVER, Engineering Journals, IEEE Xplore, International Journal of Advanced Research in Computer Science, International Journals of Computer and Communication Engineering, International Journals of Computer Applications & Information Technology, Journals in Informatics on Education, librería en línea de la Universidad Popular Autónoma de Puebla (UPAEP) y de la Universidad Autónoma del Estado de México (UAEM). Es importante comentar, que los referentes teóricos para el marco contextual fueron actualizados.

4.1.2 Enfoque y Alcance

La investigación de este trabajo doctoral se realizó con un enfoque cuantitativo y con un alcance descriptivo, porque se realizó la recolección de datos sobre las prácticas de ingeniería de software utilizadas por los docentes en la enseñanza de la programación, haciendo un análisis de los datos con estadística descriptiva y componentes principales para obtener las mejores prácticas de ingeniería de software y con el método de escalamiento de Likert (Hernández Sampieri, Fernández-Collado, & Baptista Lucio, 2006) se determinaron los criterios de calidad, aunado a la revisión de literatura. En ambos casos se permearon en un proceso de desarrollo de programas para los cursos de programación.

4.1.3 Selección de la muestra

La población objetivo está conformada por 87 Instituciones de Educación Superior (IES) públicas en México incorporadas a la Asociación Nacional de Universidades e IES de la República Mexicana A.C. (ANUIES) que ofertan carreras relacionadas con ciencias computacionales y ofertan cursos de programación (ANUIES, 2015). ANUIES es un organismo que ha participado en la formulación de planes, programas y políticas nacionales para la educación superior para mejorar la calidad educativa y tiene registradas 180 IES, actualmente 194, en diversos estados del País de las cuales 139 son públicas.

Para la selección de la muestra se utilizó la fórmula para poblaciones finitas con un grado de confiabilidad de 90 (1.645) en lugar de 95 (1.96) y un margen de error de 1 en vez de .05 teniendo como resultado una muestra de 38.31 que corresponde a 38 universidades a las que se deben acceder para poder aplicar el instrumento de recolección de datos. El ajuste en el grado de confiabilidad y en el margen de error se debe a la poca respuesta que desde abril del

2015 se tuvo para acceder a 66 IES, a pesar de contactar por correo electrónico a los responsables de las licenciaturas en diversas universidades, de acudir a congresos solicitando el apoyo para responder el instrumento, de contactar directamente a ANUIES para solicitar una cita y distribuir el instrumento, de enviar el instrumento mediante Google drive a coordinadores y directivos de universidades donde se ofertan carreras relacionadas a las ciencias computacionales y solicitar el apoyo con conocidos para aplicar el instrumento.

4.1.4 Instrumento de recolección de datos

La técnica de recolección de datos fue el cuestionario, elaborándose una primera versión para definir el constructo con base en dos perspectivas. En la primera, se consultó literatura clásica, básica y actualizada relacionada con el tema de programación y modelos de calidad. En la segunda, se realizaron entrevistas no formales a 55 docentes de 29 universidades en México, ver Problemática, a fin de conocer las prácticas de ingeniería de software que utilizan en los cursos de programación. El instrumento se aplicó a una muestra por conveniencia, en donde participaron 9 docentes expertos en el área quienes de manera voluntaria apoyaron para realizar una crítica al instrumento orientado hacia la comprensión, vocabulario y ortografía, obteniendo así el cuestionario a validar.

La validación del instrumento se llevó a cabo en dos fases. En la primera, la validez del instrumento se realizó mediante el grado de concordancia entre 7 jueces, profesores – investigadores expertos en el área, y con base en una prueba binominal se eliminaron dos ítems y se modificaron dos por no tener el grado de concordancia significativo, resultado mayor a .05. Posteriormente, se obtuvo el promedio de las respuestas con un resultado de .20 por lo que el instrumento tiene validez. En la segunda fase, se aplicó el instrumento a una muestra por conveniencia de 17 docentes con características similares a la población objetivo, como prueba piloto. La validación del instrumento se realizó con el coeficiente de Alfa de Combrach, consistencia interna que mide la precisión del constructo, obteniendo un resultado de .92. Es importante mencionar que del instrumento sólo se consideraron las respuestas de la pregunta 10, que se encuentra en escala de Likert, por lo que esta sección del instrumento tiene validez.

Como resultado de la validación, se obtuvo el instrumento final conformado con interrogantes de tipo dicotómicas, abierta, nominales, ordinales y en escala de Likert con sus respectivos ítems, ver Anexo I.

4.1.5 Recolección de datos

La recolección de datos se llevó a cabo con un cuestionario que fue respondido por 84 docentes de 38 universidades públicas en México, ver Tabla 4.1, incorporadas a la ANUIES. La recolección de datos se realizó de manera directa en los congresos: Congreso Internacional de Informática y Computación de la ANIEI (CNCIIC) 2015 llevado a cabo en Puerto Vallarta, Conferencia Mexicana en Tecnología Humanitaria (MHTC) 2015 IEEE sección Puebla,, Congreso Internacional de Investigación e Innovación en Ingeniería de Software (CONISOFT) 2015 y 2016 celebrados en la ciudad de San Luis Potosí y Puebla respectivamente, Congreso Internacional de las Tecnologías de la Información y Comunicación (CITIQ) 2016 efectuado en la ciudad de Querétaro. Asimismo, el instrumento fue enviado de manera electrónica mediante Google drive a coordinadores y directivos de universidades donde se ofertan carreras relacionadas a las ciencias computacionales para que a su vez pudieran enviar el instrumento a sus docentes y se aplicó de manera indirecta mediante el apoyo de conocidos para que aplicarían el instrumento en las universidades donde laboran.

Tabla 4.1 Universidades e IES en donde se respondieron los cuestionarios

Estado	Universidad/Institución de Educación Superior
Aguascalientes	Instituto Tecnológico de Aguascalientes
	Instituto Tecnológico de Mexicali
Baja California	Universidad Autónoma de Baja California
	Universidad Autónoma de Baja California Sur
Colima	Universidad de Colima
Ciudad de México	Instituto Politécnico Nacional
	Universidad Autónoma Metropolitana
	Universidad Nacional Autónoma de México
Durango	Instituto Tecnológico de Durango
	Universidad de Guanajuato
Guanajuato	Instituto Tecnológico de León

	Instituto Tecnológico de Pachuca
Hidalgo	Universidad Autónoma del Estado de Hidalgo
Jalisco	Universidad de Guadalajara
Estado de México	Instituto Tecnológico de Tlanepantla
	Tecnológico de Estudios Superiores de Chalco
	Tecnológico de Estudios Superiores de Cuautitlán Izcalli
	Tecnológico de Estudios Superiores de Ecatepec
	Universidad Autónoma del Estado de México
Michoacán	Universidad Michoacana de San Nicolás Hidalgo
	Instituto Tecnológico de Oaxaca
Nuevo León	Universidad Autónoma de Nuevo León
Oaxaca	Universidad Autónoma Benito Juárez de Oaxaca
Puebla	Benemérita Universidad Autónoma de Puebla
	Instituto Tecnológico de Puebla
	Instituto Tecnológico Superior de la Sierra Norte de Puebla
Querétaro	Universidad Autónoma de Querétaro
Quintana Roo	Instituto Tecnológico de Chetumal
San Luis Potosí	Instituto Tecnológico de San Luis Potosí
	Universidad Autónoma de San Luis Potosí
Tabasco	Instituto Tecnológico de Villa Hermosa
Tamaulipas	Instituto Tecnológico de Matamoros
	Universidad Autónoma de Tamaulipas
Veracruz	Instituto Tecnológico de Orizaba
	Universidad Veracruzana
Yucatán	Universidad Autónoma de Yucatán
Zacatecas	Instituto Tecnológico de Zacatecas
	Universidad Autónoma de Zacatecas

Fuente: Elaboración propia.

4.1.6 Análisis y obtención de resultados

El análisis de los datos obtenidos se realizó en dos partes, la primera mediante estadística descriptiva y componentes principales para obtener las mejores prácticas que los docentes utilizan en los cursos de programación y en la segunda con estadística descriptiva y escalamiento de Likert para determinar los criterios de calidad en el proceso de desarrollo de programas. Ambos resultados se integraron en las fases del ciclo de vida para el desarrollo

de programas. El análisis y la obtención de los resultados de las mejores prácticas de ingeniería de software y de los criterios de calidad, se explican en los capítulos 5 y 6 respectivamente.

4.2 Metodología del modelo MoACPOO

La metodología para validar el modelo MoACPOO tuvo un enfoque cuantitativo y cuasiexperimental. El modelo se aplicó en un ambiente real en un curso de programación orientada a objetos con dos grupos uno de control y otro experimental, el muestreo fue no probabilístico en virtud de que la muestra fue por conveniencia. Para el análisis de datos se utilizó estadística descriptiva y la prueba de t student para muestras independientes, este último para comprobar si el modelo propuesto es viable. Sin embargo, los detalles de la metodología se encuentran en el capítulo 8, debido a que también se presentan los resultados y es relevante se tenga el contexto de la metodología de la investigación.

Conclusión.

La metodología de este trabajo doctoral se dividió en dos fases. La primera fase fue para la investigación realizada a nivel nacional que se llevó a cabo las universidades públicas en México para obtener las mejores prácticas y criterios de calidad que son el fundamento para proponer el modelo MoACPPPO. La segunda fase fue para la evaluación del modelo MoACPOO mediante la aplicación del modelo en un entorno real. En ambas fases se utilizó un enfoque cuantitativo y para la evaluación del modelo se utilizó un diseño cuasiexperimental.

Los resultados después de aplicar la metodología de la investigación se encuentran en los capítulos 5 y 6, mientras que los de la metodología del modelo MoACPOO se encuentran en el capítulo 8. La razón por la cual los resultados se encuentran en capítulos diferentes y no en este capítulo se debe por un lado en organizar los resultados con base en su naturaleza y por el otro debido a que los análisis estadísticos presentados son diferentes.

Capítulo 5. Resultados de las mejores prácticas de ingeniería de software y su incorporación en el ciclo de vida para los cursos de programación en la formación profesional en México.

Introducción.

La importancia de llevar a cabo la investigación sobre las prácticas de ingeniería de software utilizadas por los docentes en los cursos de programación se debe en que actualmente y desde hace años a los estudiantes se les dificultan los cursos de programación en muchas universidades, ver sección 1.1. Por ello se considera importante que los estudiantes en su formación profesional cuenten con una directriz, ciclo de vida o proceso de desarrollo, basadas en buenas prácticas de ingeniería de software, ver sección 2.2, prácticas de ingeniería de software, en aras de crear programas que funcionen correctamente.

Considerando lo anterior, este capítulo está regido por la pregunta de investigación **¿Cuáles son las mejores prácticas de ingeniería de software utilizadas en los cursos de programación por los docentes en las universidades públicas en México?**

Para dar respuesta a la pregunta de investigación, se aplicó un cuestionario a 84 docentes de 38 universidades públicas cumpliendo con el objetivo **Identificar las mejores prácticas de ingeniería de software utilizadas por los docentes en los cursos de programación en las universidades públicas en México.**

Con base en la problemática planteada, ver sección 1.1, y a las reflexiones de Watson y Li (2014) que indican que no se han producido mejoras significativas en los índices de aprobación a través del tiempo, ya que el porcentaje promedio de estudiantes que no acreditan los cursos básicos programación se ha mantenido constante en los años 2000, teniendo tasas similares a las décadas de los 80 y 90; y, a que una de las razones de fracaso en los estudiantes universitarios en los cursos de programación se debe a problemas en el proceso de programación (Özmen & Altun, 2014), el supuesto de este capítulo es: **Los docentes de las universidades públicas en México utilizan pocas prácticas de ingeniería de software en los cursos de programación.**

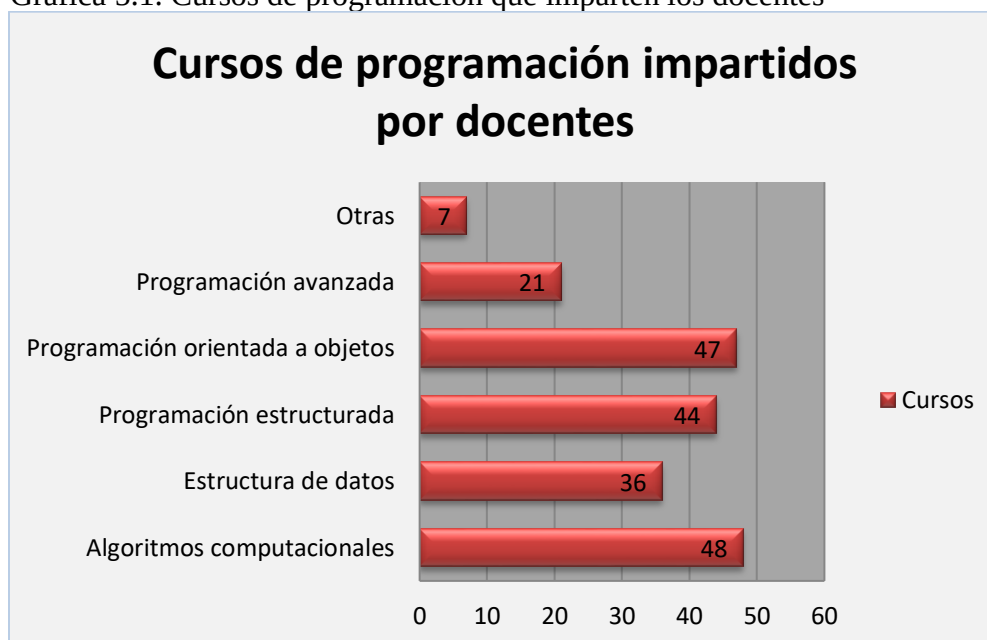
En este sentido, con el levantamiento de datos de las prácticas de ingeniería de software que los docentes utilizan en los cursos de programación y haciendo un análisis de frecuencias de las prácticas utilizadas, se seleccionaron las mejores aplicando estadística descriptiva y

haciendo un análisis de componentes principales; las mejores mismas que fueron adheridas a las fases, adaptadas, del ciclo de vida en cascada.

5.1 Enfoques y herramientas utilizados en los cursos de programación.

Las universidades e Instituciones de Educación Superior (IES) incluyen en sus programas de estudio cursos relacionados con programación, siendo los más instruidos algoritmos computacionales con un 57.14%, programación orientada a objetos con el 55.95% y programación estructurada con el 52%, considerando que 63 docentes de 84 imparten más de un curso relacionado con programación, ver Grafica 5.1.

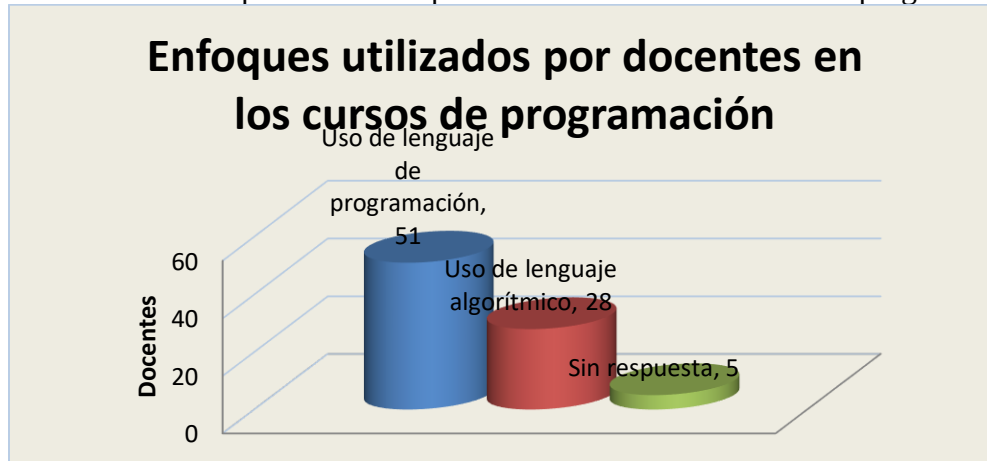
Gráfica 5.1. Cursos de programación que imparten los docentes



Fuente: Elaboración propia.

En los cursos mencionados anteriormente, el 60.71% de los docentes enseñan un lenguaje de programación para el desarrollo de programas, sin embargo, la programación parte de un análisis del problema antes de proponer una solución, la cual se debe estructurar previo a la codificación, que es la finalidad del enfoque algorítmico, Desafortunadamente, solo el 33.33% de los docentes utilizan este enfoque, ver Grafica 5.2.

Gráfica 5.2. Enfoques utilizados por los docentes en los cursos de programación



Fuente: Elaboración propia.

Dado que los cursos de programación son difíciles para los estudiantes, 58 de 84 docentes utilizan como apoyo al menos una herramienta visual mediante entornos interactivos para comprender los conceptos de programación. En la Gráfica 5.3 se puede observar que el 28.57% no utilizan ninguna herramienta, y el 26.37% utilizan otras herramientas como raptor, blueJ, Pseint y codeBlocks, herramientas de diseño y lenguaje de programación. No obstante, y en atención a lo expuesto, (Watson & Li, 2014, pág. 42) “el aumento en la cantidad de herramientas disponibles para apoyar a los estudiantes en los cursos de programación es interesante ver que no se han producido mejoras significativas en los índices de aprobación”.

Gráfica 5.3. Herramientas de apoyo en los cursos de programación



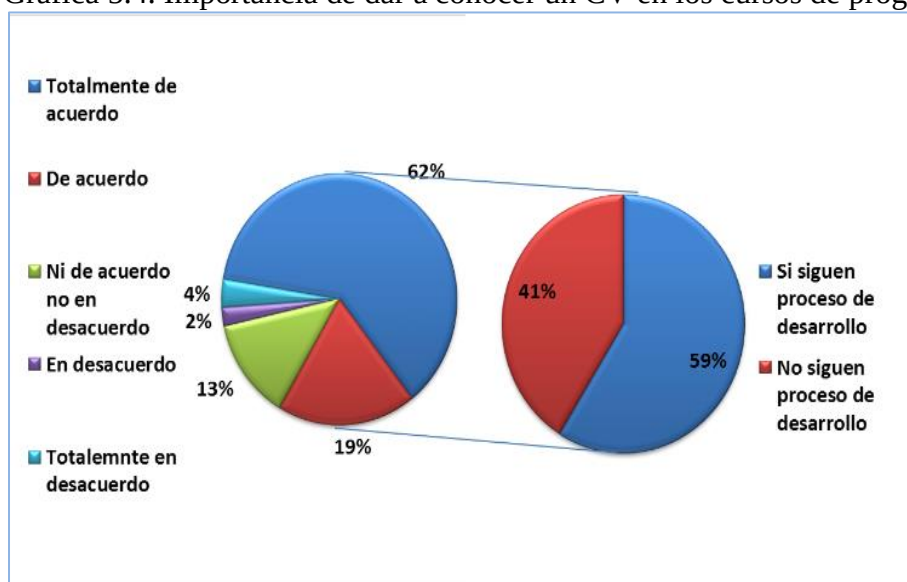
Fuente: Elaboración propia.

5.2 Ciclo de vida como apoyo en los cursos de programación en el ámbito universitario.

Ciclos de vida (CV) para el desarrollo de software se encuentran presentes en la industria del software siendo los estudiantes y egresados de las universidades las personas que contribuyen en dar solución a problemas mediante el uso de tecnologías de información y comunicación (TIC), que incluyen programas de software, al insertarse en el mercado laboral. Por ello, las universidades se deben ocupar que en los cursos de programación se adopte un CV que permita a los estudiantes guiar, mostrar y formar el hábito de seguir un proceso en la creación de programas para que no desarrollen programas a prueba y error. Los estudiantes al seguir un proceso para el desarrollo de programas puedan entender el problema, hacer un diseño e implementar la solución (López Román, 2011).

En este sentido, el 81% de docentes indican que es importante que los estudiantes conozcan un CV, considerando las respuestas totalmente de acuerdo y de acuerdo. No obstante, sólo el 59% de ellos siguen un proceso para el desarrollo de programas, ver Grafica 5.4.

Gráfica 5.4. Importancia de dar a conocer un CV en los cursos de programación.



Fuente: Elaboración propia.

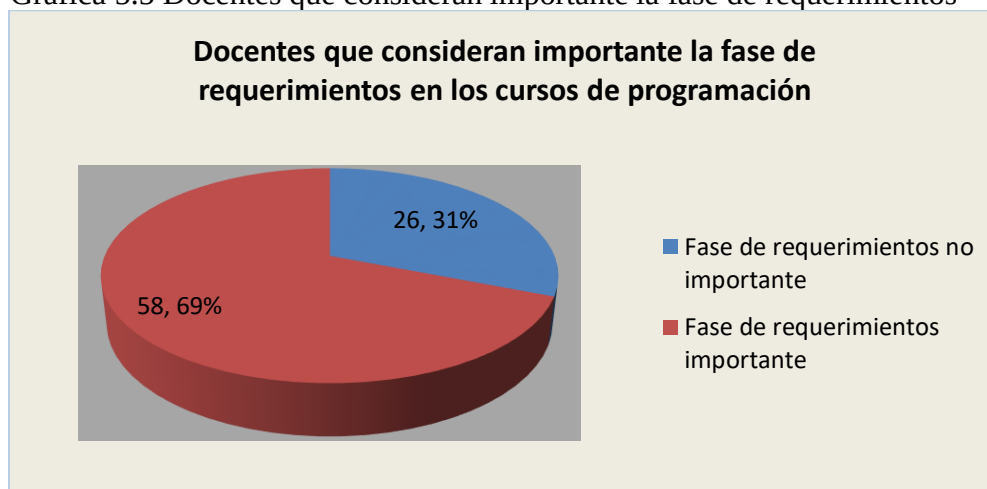
Con la finalidad de conocer las prácticas de ingeniería de software que los docentes utilizan en cada una de las fases del CV en los cursos de programación, se incluyeron en el cuestionario 7 preguntas abiertas, una por cada fase. A continuación, se presentan los resultados.

5.2.1 Requerimientos

En los cursos de programación los requisitos están predefinidos en el planteamiento del problema y deben ser analizados por los estudiantes para comprender lo que se les está solicitando y así determinar los elementos que intervendrán en la solución del problema. La importancia de utilizar buenas prácticas de ingeniería de software en los cursos de programación en esta fase es que los estudiantes conozcan en etapas tempranas de su formación profesional que los requerimientos indican las necesidades y restricciones de la solución.

En la Gráfica 5.5, se visualiza que 58 de 84 docentes consideran importante los requerimientos para que el estudiante pueda entender la problemática a resolver permitiendo estructurar mejor sus ideas e identificar los elementos que intervendrán en la obtención de un buen resultado.

Gráfica 5.5 Docentes que consideran importante la fase de requerimientos



Fuente: Elaboración propia.

Sin duda, es importante incluir una etapa explícita de requerimientos en los cursos de programación porque con esta se gana el entendimiento del problema, lo que resulta importante para el desarrollo del programa. Además, se forma el hábito en los estudiantes de no omitir dicha fase y de hacer conciencia social de considerar los requerimientos en los cursos de programación como se hace en las empresas que desarrollan software.

Dentro de este marco y con base en los resultados del cuestionario, se identificaron las prácticas de ingeniería de software utilizadas por los docentes en los cursos de programación

en la fase de requerimientos, ver Tabla 5.1, en donde el 19.04% consideran las entrevistas, encuestas y observación, y utilizan los requerimientos funcionales y no funcionales de la IEEE 830 para identificar los requisitos. Sin embargo, en los cursos de programación no se da solución a un proyecto de software con *stakeholders* con los que se puede realizar esta actividad.

Tabla 5.1. Prácticas utilizadas por los docentes en la fase de requerimientos en los cursos de programación en México 2015 – 2016.

Práctica utilizada por el docente	Frecuencia	Porcentaje %
Sin contestar.	30	35.71
Definición y comprensión del problema.	11	13.10
Entrevistas, encuestas u observación.	10	11.90
Utilizar requerimientos funcionales y no funcionales IEEE830.	6	7.14
Planteamiento de problemas cotidianos.	4	4.76
Predefinidos en planteamiento y descripción del problema.	3	3.57
Catálogo de elementos.	3	3.57
Levantamiento de datos (identificación de variables, datos E/S)	3	3.57
Definición de requerimientos de información y de datos.	2	2.38
Interfaces de usuario.	2	2.38
Identificar requerimientos.	2	2.38
Clasificación de requerimientos participando activamente con el usuario.	1	1.19
Maquetado de entidades.	1	1.19
Ejercicios.	1	1.19
Redacción de algoritmo y lectura de comprensión.	1	1.19
Ver ejemplos de su uso correcto.	1	1.19
Hoja de requerimientos.	1	1.19
Especificación y modelado de caso de uso.	1	1.19
Análisis de caja negra.	1	1.19
Total, de docentes	84	100.00

Nota: El porcentaje más alto fueron de docentes que no utilizan ninguna práctica para definir los requerimientos. Las prácticas sombreadas con gris son las que presentan mayor frecuencia, no obstante entrevistas, encuestas y observación, y utilizar los requerimientos funcionales y no funcionales de la IEEE 830 no serán consideradas.

Fuente: Elaboración propia.

5.2.2 Análisis

Después de haber entendido y comprendido la problemática a resolver, deberán analizar los requisitos predefinidos, para poder identificar los elementos que conformarán el programa tales como: variables, constantes, tipos de datos, fórmulas, funciones/métodos, parámetros,

datos de entrada, datos de salida y los resultados esperados por mencionar algunos. Las respuestas dadas por los docentes de las prácticas de ingeniería de software que utilizan en los cursos de programación se muestran en la Tabla 5.2.

Tabla 5.2. Prácticas de ingeniería de software utilizadas por los docentes en la fase de análisis en los cursos de programación en México 2015 – 2016.

Práctica utilizada por el docente	Frecuencia	Porcentaje %
Sin contestar.	35	41.67
Identificar atributos, datos, métodos, funciones, interacciones, operaciones*.	14	16.67
Casos de uso.	5	5.95
Uso de UML.	4	4.76
Planteamiento de soluciones.	3	3.57
Diagramas de flujo.	2	2.38
Algoritmos.	1	1.19
BPMN.	2	2.38
Descripción del problema.	1	1.19
Determinar alcance del proyecto.	1	1.19
Determinar compromisos de entrega y factibilidad.	1	1.19
Diagramas preliminares.	1	1.19
Diseño centrado en el usuario.	1	1.19
Documentación.	1	1.19
Ejercicios.	1	1.19
Equipos de 4 personas.	1	1.19
Herramientas gráficas.	1	1.19
Lúdicamente.	1	1.19
Matriz de trazabilidad.	1	1.19
Modelado de negocio y dominio.	1	1.19
Necesidades de usuario.	1	1.19
Pruebas de escritorio para el algoritmo.	1	1.19
Razonamiento.	1	1.19
Técnica divide y vencerás.	1	1.19
Uso de Project.	1	1.19
Uso de cajas negras.	1	1.19
Total, de docentes	84	100.00

Nota: *Discriminando a los docentes que no contestaron, sólo el 31.81%, indican que la identificación de atributos, datos, métodos, funciones, interacciones, operaciones.

Fuente: Elaboración propia.

Como se puede observar, el 5.95% de los docentes son los casos de uso, que pueden utilizarse para que los estudiantes describan el flujo normal y alternativo. Además, algunas prácticas están orientadas al desarrollo del software mientras que otras están enfocadas más a la fase de diseño. Es evidente que un gran porcentaje no utiliza ninguna práctica.

5.2.3 Diseño

La fase de diseño consiste en traducir los requisitos predefinidos que conformarán el programa en una representación que indique como debe comportarse utilizando alguna técnica de diseño dependiendo del paradigma de programación. Cabe comentar que algunas respuestas no corresponden propiamente a técnicas de diseño, ya que este trabajo busca conocer las prácticas que los docentes utilizan en los cursos de programación, por lo que no se especificó que deberían responder con alguna técnica de diseño. Como se puede apreciar en la Tabla 5.3, existen tres técnicas de diseño principalmente que 32 docentes de los 84 utilizan en el modelado UML, diagramas de flujo y algoritmos.

Tabla 5.3. Prácticas de ingeniería de software utilizadas por los docentes en la fase de diseño en los cursos de programación en México 2015 – 2016.

Práctica utilizada por el docente	Frecuencia	Porcentaje %
Sin contestar	23	27.38
Modelado UML.	11	13.10
Diagrama de flujo.	12	14.29
Algoritmo.	9	10.71
Lúdicamente.	3	3.57
Pseudocódigo.	3	3.57
Patrones de diseño.	2	2.38
Mockups.	2	2.38
Casos de uso.	2	2.38
Modelo conceptual.	1	1.19
Identificar módulos del sistema.	1	1.19
Se diseña una práctica.	1	1.19
Mediante programación estructurada.	1	1.19
Estructurar base del objetivo.	1	1.19
Elaboración de interfaces amigables.	1	1.19
Uso de IEEE 29110.	1	1.19
Diseño de interfaces centradas en el usuario.	1	1.19
Parámetros en métodos.	1	1.19
Estructuración general.	1	1.19

Ejercicios.	1	1.19
Documentación.	1	1.19
Diseño colaborativo.	1	1.19
Diagrama de clase.	1	1.19
Brainstoming.	1	1.19
Powerpoint.	1	1.19
Diseño de la arquitectura del sistema.	1	1.19
Total, de docentes	84	100.00

Nota: Las prácticas más utilizadas por los docentes son: modelado UML, diagrama de flujo y algoritmos; que es congruente con los cursos de programación que imparten los docentes, siendo éstas: programación orientada a objetos, algoritmos y programación estructurada.

Fuente: Elaboración propia.

Los estudiantes en esta fase deberán realizar el diseño de los programas considerando los elementos identificados en la fase de requerimientos organizándolos, clasificándolos y estructurándolos, para que se otorgue semántica al diseño, teniendo como finalidad poder visualizar los pasos que el programa debe seguir hasta obtener los resultados esperados.

5.2.4 Codificación

La fase de codificación tiene la finalidad traducir el diseño a código fuente utilizando como herramienta un lenguaje de programación con base al paradigma utilizado. Si bien es cierto que en la codificación se debe cuidar la sintaxis y la semántica del lenguaje de programación, también lo es el funcionamiento del programa para obtener los resultados correctos, lo cual no necesariamente depende de la sintaxis y semántica de las líneas de código ya que existen errores lógicos y errores de ejecución que hacen que un programa no funcione de manera correcta.

En la Tabla N se incluyen las prácticas de ingeniería de software que los docentes utilizan en los cursos de programación, el 67.85% que al ser transferidas a los estudiantes contribuyen por un lado en hacerlos conscientes de que la codificación va más allá que escribir líneas de código haciendo uso de la sintaxis y cuidando su semántica, no incurriendo en la realización de un programa a prueba y error; y por otro en mejorar la creación del código fuente reduciendo los errores e incrementando la productividad. También 12 docentes utilizan prácticas de código limpio para que el código fuente sea legible y fácil de comprender, como es la indentación de las líneas de código y el uso de comentarios.

Tabla 5.4. Prácticas de ingeniería de software utilizadas por los docentes en la fase de codificación en los cursos de programación en México 2015 – 2016.

Práctica utilizada por el docente	Frecuencia	Porcentaje %
Sin contestar.	27	30.00
Sólo lenguaje de programación (sintaxis y reglas).	19	21.11
Documentación de código (comentarios), indentación, identificadores.	12	13.33
Convertir diagrama de flujo (diseño) a código.	8	8.89
Estándar de codificación y lista de cotejo.	5	5.56
Ejercicios.	2	2.22
Control de versiones, definición de roles, flujos de trabajo en equipos.	2	2.22
Uso de herramientas.	1	1.11
Uso de PSP.	1	1.11
Seudocódigo.	1	1.11
Transcripción de código.	2	2.22
Modularidad por UML.	1	1.11
Desarrollo de rutinas con estructuras de control.	1	1.11
Diagrama y codificación de procesos.	1	1.11
Trabajar con programas parecidos y adaptarlos.	1	1.11
Lúdicamente.	1	1.11
Uso adecuado de variables.	1	1.11
Previa generación de algoritmos.	1	1.11
Validación de procesos modelados en diseño y manejo de excepciones.	1	1.11
Uso de framework.	1	1.11
Reutilización de código.	1	1.11
Total, de respuestas*	90	100.00
Total, de docentes	84	100.00

Notas: La mayoría de los docentes, considerando sólo los que respondieron a esta interrogante, utilizan la sintaxis y semántica como una buena práctica, pero como se explicó en el tercer párrafo, la codificación va más allá.

* Hubo docentes que utilizan más de una práctica en la codificación por lo que se tienen 90 respuestas y no 84.

Fuente: Elaboración propia.

5.2.5 Compilación

La compilación es la fase que consiste en traducir el código fuente a lenguaje máquina detectando los errores de sintaxis y advertencias. A diferencia de otras fases, la no corrección de los errores detectados por el compilador impactará en que el programa no se ejecute y no se podrá continuar con la fase de pruebas. En la Tabla 5.5, se muestra que el 13.92% de docentes coinciden que se deben entender los errores y corregirlos, lo que es adecuado ya

que si no se comprende el significado del error no se podrán realizar las correcciones respectivas.

Tabla 5.5. Prácticas de ingeniería de software utilizadas por los docentes en la fase de compilación en los cursos de programación en México 2015 – 2016.

Práctica utilizada por el docente	Frecuencia	Porcentaje %
No contestó.	39	46.43
Entender defectos y corregir.	14	16.67
Conteo de defectos de sintaxis.	5	5.95
Documentar resultados de compilación (bitácora de defectos).	5	5.95
Rastrear errores (uso de debug).	5	5.95
Revisión de código antes de compilar.	3	3.57
validación de errores.	2	2.38
Depende de la herramienta.	1	1.19
No usar como revisión de codificación.	1	1.19
Casos de prueba.	1	1.19
Prueba general.	1	1.19
Pruebas unitarias.	1	1.19
Uso de un IDE para asistir la detección y depuración	1	1.19
Concientizar a los alumnos que el IDE oculta los comandos de compilación.	1	1.19
Uso de IDE.	1	1.19
Control de versiones.	1	1.19
Transcribir lenguaje coloquial a máquina.	1	1.19
Verificar.	1	1.19
Total, de docentes	84	100.00

Nota: Las prácticas que se encuentran sombreadas son las que con más frecuencia utilizan los docentes

Fuente: Elaboración propia.

Dos aportaciones importantes que hacen 5 docentes de 84 son llevar un control de los errores y el uso del depurador para rastrear los mismos. La importancia radica en hacer conciencia en los estudiantes de la cantidad de errores que cometen para tomar acciones que les permitan erradicar o mitigar los mismos, así como detectar errores lógicos al conocer los valores de las variables durante la ejecución del programa.

5.2.6 Pruebas

La fase de prueba tiene lugar cuando la compilación ya no presenta ningún defecto; sin embargo, no se puede decir que se ha concluido satisfactoriamente con el proceso de programación, dado que al programa se le deben aplicar pruebas para corroborar que se

emitan los resultados esperados. Las prácticas que los docentes utilizan en esta fase se muestran en la Tabla 5.6.

Tabla 5.6. Prácticas de ingeniería de software utilizadas por los docentes en la fase de pruebas en los cursos de programación en México 2015 – 2016.

Práctica utilizada por el docente	Frecuencia	Porcentaje %
No contestó.	27	32.14
Casos de prueba.	24	28.57
Probar con diferentes datos (identificar BUGS).	7	8.33
Pruebas de integración y del sistema.	4	4.76
Aplicar pruebas de escritorio.	3	3.57
Pruebas unitarias.	2	2.38
Uso de caja blanca y negra.	2	2.38
Aprender a depurar.	2	2.38
Ejemplos con resultados.	1	1.19
Secuencia de ajuste y visualización de objetivos.	1	1.19
Uso de métricas.	1	1.19
Práctica.	1	1.19
OmegaUP.	1	1.19
Pruebas de estrés al programa.	1	1.19
Framework de testing.	1	1.19
Bitácoras.	1	1.19
Validación de resultados con diseño.	1	1.19
TDD.	1	1.19
Crear diferentes rutinas de ejecución.	1	1.19
Pruebas mínimas.	1	1.19
Corregir errores encontrados y realizar nuevamente pruebas.	1	1.19
Total, de docentes	84	100.00

Nota: Las prácticas sombreadas son las que presentan mayor frecuencia.

Fuente: Elaboración propia.

Como se puede apreciar que el 28.57% de los docentes utilizan casos de prueba para probar un programa para verificar que los resultados sean correctos. Otra práctica que utilizan los docentes es la detección de errores y la depuración de código, para corregir los errores, resultado de las pruebas.

5.2.7 Mantenimiento

El mantenimiento es la última fase del ciclo de vida, en la cual se podrán corregir los errores encontrados en el programa, realizando así un mantenimiento de tipo correctivo. No obstante, en cualquier fase al ser detectado un error se deberá corregir. El no aplicar esta fase se considera viable y no, dependiendo de la perspectiva. No es viable debido a que los programas que se realizan en los cursos de programación no conllevan un mantenimiento perfectivo ni adaptativo; caso contrario es viable ya que el mantenimiento correctivo tiene lugar sobre todo después de las fases de compilación y pruebas, tal como se aprecia en la Tabla 5.7, donde el 17.86% realiza este tipo de mantenimiento.

Tabla 5.7. Prácticas de ingeniería de software utilizadas por los docentes en la fase de mantenimiento en los cursos de programación en México 2015 – 2016.

Práctica utilizada por el docente	Frecuencia	Porcentaje %
No contestó.	51	60.71
Mantenimiento correctivo.	15	17.86
Reutilizar código.	4	4.76
Modificar requerimientos.	2	2.38
No parchar.	1	1.19
Se realiza si se genera una nueva versión.	1	1.19
Relación estrecha del cliente con sus problemáticas.	1	1.19
Buenas prácticas de programación.	1	1.19
Realimentación de pruebas y métodos variantes versionados.	1	1.19
Verificar tiempo, costo e impacto del mantenimiento.	1	1.19
Seguimiento del programa.	1	1.19
Integración continua.	1	1.19
UML.	1	1.19
leer código ya desarrollado.	1	1.19
Documentación.	1	1.19
Agregar módulos.	1	1.19
Total, de docentes	84	100.00

Nota: La fase de mantenimiento es la que presenta el índice más alto de ausencia de respuesta proporcionada con las prácticas, No obstante, es importante que sea considerada porque al corregir los defectos se está realizando un mantenimiento correctivo, como lo realiza el 17.86% de los docentes. Las prácticas sombreadas son las que presentan mayor frecuencia.

Fuente: Elaboración propia.

5.3 Selección de las mejores prácticas de ingeniería de software

Las respuestas obtenidas en cada una de las fases del CV se codificaron utilizando variables dicotómicas para poder representarlos en una tabla de frecuencias, los valores perdidos no fueron considerados (requerimientos 30, análisis 35, diseño 23, codificación 27, compilación 39, pruebas 27 y mantenimiento 51). Posteriormente se seleccionaron las prácticas que tuvieron una ocurrencia de tres o más discriminándose las prácticas que por su naturaleza están enfocadas al desarrollo de software o bien que no pertenecen a la fase. En la Tabla 5.8 se muestran las prácticas que los docentes utilizan en los cursos de programación y es preocupante que sólo el 27% de los docentes utilicen al menos una práctica.

Tabla 5.8. Prácticas de ingeniería de software más frecuentes usadas por los docentes en los cursos de programación en México 2015 – 2016.

Fase	Práctica utilizada por el docente	Frecuencia
Requerimientos	Definición y comprensión del problema.	11
	Predefinidos en planteamiento y descripción del problema.	3
	Catálogo de elementos.	3
Análisis	Identificar atributos, datos, métodos, funciones, interacciones, operaciones.	14
	Casos de uso.	5
	Uso de UML.	4
	Planteamiento de soluciones.	3
Diseño	Modelado UML.	11
	Diagrama de flujo.	12
	Algoritmo.	9
	Lúdicamente.	3
	Pseudocódigo.	3
Codificación	Sólo lenguaje de programación (sintaxis y reglas).	19
	Documentación de código (comentarios), indentación, identificadores.	12
	Convertir diagrama de flujo (diseño) a código.	8
	Estándar de codificación y lista de cotejo.	5
Compilación	Entender defectos y corregir.	14
	Conteo de defectos de sintaxis.	5
	Documentar resultados de compilación (bitácora de defectos).	5
	Rastrear errores (uso de debug).	5
	Revisión de código antes de compilar.	3
Pruebas	Casos de prueba.	23
	Probar con diferentes datos (identificar BUGS).	7
Mantenimiento	Mantenimiento correctivo.	15

Fuente: Elaboración propia.

Como se puede observar el 13.10% de los docentes dan importancia a la definición y comprensión del problema y el 16.67% a la identificación de los elementos que conformarán el programa tales como: atributos, datos, métodos, funciones, interacciones y operaciones. Estas dos prácticas están relacionadas ya que si se logra entender el problema se podrán identificar las variables, tipos de datos, operaciones y pasos a seguir, mismos que deberán utilizarse en el diseño para la solución del problema.

Las tres técnicas de diseño principalmente utilizadas son: modelado UML, diagramas de flujo y algoritmos; indicando los docentes que son importantes debido a que su uso requiere organizar, clasificar y estructurar los elementos para que se otorgue semántica al diseño, determinando la lógica del programa.

Con respecto a la codificación, el 22.70% de los docentes manifiestan que es importante que los estudiantes conozcan la sintaxis y las reglas de un lenguaje de programación ya que de ello depende que el conjunto de instrucciones del programa sea acorde al diseño. Además, el 14.29% están a favor de que el código sea fácil de entender por lo que se deben incluir comentarios, indentación y uso de identificadores, que son reglas las cuales se pueden definir en un estándar de codificación. Después de la codificación tiene lugar la compilación de un programa pudiendo surgir errores que ocasionan que el programa no funcione de manera adecuada y para poder corregirlos es necesario entender el error. Ante ello el 16.67% de los docentes aplican esta práctica, no obstante, sólo el 6% hacen uso de la revisión de código, llevar una bitácora de defectos, y uso de debug.

El 27.38% docentes indican que los casos de prueba para probar un programa son relevantes para verificar que los resultados sean correctos con base en los requisitos predefinidos del planteamiento del problema.

El mantenimiento tiene lugar cuando se corrigen los errores detectados en las fases previas a lo que el 17.86% de los docentes opinan que se trata del mantenimiento correctivo que está implícita durante el proceso de desarrollo.

Por otro lado, para seleccionar las mejores prácticas de ingeniería de software se analizaron las respuestas de un total de 31 variables en escala de Likert (5 siempre, 4 Casi siempre, 3 Mayoría de las veces, 2 Pocas veces, 1 Nunca), ver pregunta 12 del cuestionario Anexo I eliminándose cuatro por ser redundantes, sustentado en el análisis descriptivo y por componentes. Por cada una de las variables, se obtuvo la distribución de frecuencias y la

tabla de descriptivos en donde se incluyen estadísticas fundamentales de cada una de las variables. En la Imagen 5.1 se puede observar que el 81% de las prácticas, presentan una media arriba de 3 por lo que son aceptables. Los resultados anteriores se consideran propicios ya que indican que 25 prácticas de 31 son las más utilizadas por los docentes.

Imagen 5.1. Estadísticos descriptivos de 31 variables

Estadísticos descriptivos					
	N	Mínimo	Máximo	Media	Desv. típ.
Identificar datos de entrada	84	1	5	4.63	.833
identificar datos de salida	84	1	5	4.56	.961
Verificar resultados correctos	84	1	5	4.54	.924
Ejecución correcta del programa	84	1	5	4.45	1.046
Identificar funciones/métodos	84	1	5	4.42	1.008
Identificar variables a utilizar	84	1	5	4.40	.920
Conocer la sintaxis del lenguaje	84	1	5	4.30	1.073
identificar formulas	84	1	5	4.29	1.071
Uso de algoritmo	84	1	5	4.25	1.051
Verificar que el programa sea claro y legible	84	1	5	4.15	1.135
Prueba de validación de datos	84	1	5	4.15	1.177
Uso de comentarios en el código	84	1	5	4.12	1.216
Validación de datos de entrada	84	1	5	4.11	1.064
Operar programas	84	1	5	3.99	1.177
Codificar de manera modular	84	1	5	3.96	1.246
Uso de diagrama de flujo	84	1	5	3.86	1.272
Depurar código	84	1	5	3.77	1.090
Uso de pseudocódigo	84	1	5	3.63	1.324
Hacer seguimiento de variables	84	1	5	3.52	1.187
Codificar de manera secuencial	84	1	5	3.51	1.401
Uso de diagrama de clases	84	1	5	3.40	1.457
Modularizar programa lineal	84	1	5	3.37	1.269
Uso de pruebas de escritorio	84	1	5	3.37	1.479
Revisión de código (defectos de sintaxis)	84	1	5	3.35	1.331
Transcribir código	84	1	5	3.24	1.350
Uso de diagrama de actividades	84	1	5	2.88	1.492
Registrar tiempo de desarrollo	84	1	5	2.67	1.484
Registrar defectos lógicos	84	1	5	2.64	1.403
Registra defectos en ejecución	84	1	5	2.60	1.309
Registrar defectos de sintaxis	84	1	5	2.44	1.365
Uso de diagrama Nassi-Scheiderman	84	1	5	2.11	1.280
N válido (según lista)	84				

Fuente: Resultados con SPSS.

Se aplicó la medida de adecuación muestral de Kaiser-Meyer-Olkin (KMO), para saber la factibilidad de realizar el análisis por componentes principales, el cual fue favorable, ver Imagen 5.2. La finalidad del análisis de componentes principales es agrupar las prácticas de ingeniería de software para obtener un número menor de variables sin que exista pérdida de información.

Imagen 5.2. Medida de adecuación muestral KMO

KMO y prueba de Bartlett		
Medida de adecuación muestral de Kaiser-Meyer-Olkin.		.664
Prueba de esfericidad de Bartlett	Chi-cuadrado aproximado	1303.933
	gl	465
	Sig.	.000

Fuente: Resultados con SPSS.

El análisis de los datos se inició con la matriz de correlaciones de las 31 variables con un resultado del valor determinante cercano a cero y con la matriz anti-imagen con resultados de coeficientes bajos. Los valores de correlación superiores a .7 fueron registro de defectos de sintaxis con los defectos lógicos y de ejecución; identificación de los datos de entrada con la identificación de datos de salida; e identificación de fórmulas con identificar funciones/métodos, por lo que se consideran practicas altamente correlacionadas. Así, los docentes que usan una de estas prácticas también utilizan las prácticas correlacionadas. Debido a la gran cantidad de datos la Imagen 5.3 muestra una fracción de la matriz de correlaciones en donde se observa la relación lineal entre las variables.

Imagen 5.3. Fragmento de la matriz de correlaciones

		Verificar resultados correctos	Validación de datos de entrada	Ejecución correcta del programa	Revisión de código (defectos de sintaxis)	Depurar código	Transcribir código	Uso de pruebas de escritorio	Modularizar programa lineal
Correlación	Verificar resultados correctos	1.000	.443	.394	-.044	.229	.099	.100	.014
	Validación de datos de entrada	.443	1.000	.172	.246	.198	.200	.128	.033
	Ejecución correcta del programa	.394	.172	1.000	.008	.101	.111	.015	-.145
	Revisión de código (defectos de sintaxis)	-.044	.246	.008	1.000	.054	.162	-.041	.131
	Depurar código	.229	.198	.101	.054	1.000	.274	.232	.261
	Transcribir código	.099	.200	.111	.162	.274	1.000	.293	.356
	Uso de pruebas de escritorio	.100	.128	.015	-.041	.232	.293	1.000	.415
	Modularizar programa lineal	.014	.033	-.145	.131	.261	.356	.415	1.000
Sig. (Unilateral)	Verificar resultados correctos		.000	.000	.344	.018	.184	.182	.449
	Validación de datos de entrada			.058	.012	.036	.034	.124	.383
	Ejecución correcta del programa				.473	.179	.158	.445	.093
	Revisión de código (defectos de sintaxis)					.311	.071	.355	.118
	Depurar código						.006	.017	.008
	Transcribir código							.003	.000
	Uso de pruebas de escritorio								.000
	Modularizar programa lineal								

Fuente: Resultados con SPSS.

La extracción de componentes principales se realizó con el criterio de Kaiser-Gutman, autovalor mayor que 1, obteniendo las comunales de las 31 variables, siendo las mejor explicadas: registrar defectos lógicos, identificar datos de entrada, identificar datos de salida, uso de diagrama Nassi-Sheiderman, registrar defectos de sintaxis y validación de datos de entrada; las comunales de estas variables oscilan entre .804 y .894. Considerando estos resultados, las variables antes mencionadas son las que más estarán representadas en los componentes principales, lo cual es coherente con las correlaciones de las variables, aunado a que al dar solución a un problema se requiere de un análisis para identificar los datos de entrada y salida, de un diseño, de la codificación en donde es deseable llevar un registro de defectos con base en la corrección de defectos y de la ejecución que permite hacer pruebas validando los datos de entrada. Así, se generaron 9 valores mayores a uno que son los componentes comunes extraídos para resumir las variables originales en un 69.34%, ver Imagen 5.4.

Imagen 5.4. Porcentaje de varianza explicada

Componente	Autovalores iniciales			Sumas de las saturaciones al			Suma de las saturaciones al		
	Total	% de la varianza	% acumulado	Total	% de la varianza	% acumulado	Total	% de la varianza	% acumulado
1	6.965	22.466	22.466	6.965	22.466	22.466	4.082	13.167	13.167
2	3.378	10.897	33.363	3.378	10.897	33.363	3.782	12.201	25.368
3	2.240	7.226	40.590	2.240	7.226	40.590	2.213	7.140	32.508
4	1.785	5.758	46.348	1.785	5.758	46.348	2.156	6.955	39.463
5	1.683	5.430	51.778	1.683	5.430	51.778	2.028	6.542	46.005
6	1.580	5.098	56.876	1.580	5.098	56.876	1.946	6.278	52.284
7	1.476	4.763	61.638	1.476	4.763	61.638	1.862	6.006	58.290
8	1.250	4.031	65.669	1.250	4.031	65.669	1.793	5.785	64.074
9	1.139	3.673	69.342	1.139	3.673	69.342	1.633	5.268	69.342
10	.982	3.168	72.510						
11	.920	2.969	75.479						
12	.803	2.589	78.068						
13	.716	2.308	80.377						
14	.664	2.142	82.518						
15	.636	2.051	84.570						
16	.596	1.922	86.492						
17	.534	1.721	88.213						
18	.462	1.490	89.703						
19	.417	1.345	91.048						
20	.395	1.276	92.324						
21	.378	1.221	93.545						
22	.361	1.165	94.710						
23	.335	1.080	95.790						
24	.295	.951	96.742						
25	.261	.842	97.583						
26	.220	.709	98.292						
27	.184	.594	98.886						
28	.134	.434	99.320						
29	.113	.364	99.683						
30	.065	.208	99.892						
31	.034	.108	100.000						

Fuente: Resultados con SPSS.

Con la correlación entre las variables y los componentes extraídos, ver Imagen 5.5, se determinó el nombre a cada uno de los componentes. Los componentes resultantes con base en la agrupación de las variables son: componente 1 “Análisis de requisitos”, componente 2” Bitácoras y prueba”, componente 3 “Diseño en programación estructurada”, componente 4 “Seguimiento del código”, componente 5 “Verificación”, componente 6 “Comprobación del programa”, componente 7 “Diseño en programación orientada a objetos”, componente 8 “Codificación” y componente 9 “Revisión de código”.

Imagen 5.5. Matriz de componentes extraídos

	Componente								
	1	2	3	4	5	6	7	8	9
Identificar funciones/métodos	.711	-.200	-.224	-.206	-.046	-.087	-.258	.253	.074
identificar formulas	.665	-.302	.001	-.310	-.027	-.003	-.022	.331	.135
Prueba de validación de datos	.656	.035	-.287	.051	.142	.218	.148	-.102	.115
Verificar que el programa sea claro y legible	.613	.015	-.233	.216	.063	-.127	-.099	-.214	-.266
identificar datos de salida	.606	-.596	-.141	-.201	.127	-.151	.036	.040	-.086
Depurar código	.594	.117	-.065	.131	.099	-.305	.042	-.134	.234
Uso de comentarios en el código	.573	.049	-.172	.059	-.350	-.044	-.171	-.142	-.080
Identificar variables a utilizar	.543	-.267	-.016	-.172	-.325	-.029	.204	.107	-.040
Hacer seguimiento de variables	.534	.085	.181	.213	-.196	-.224	.003	-.085	-.446
Operar programas	.499	-.136	.071	.351	-.212	.260	.088	-.128	-.468
Conocer la sintaxis del lenguaje	.481	-.199	-.052	-.357	-.327	.187	-.064	-.212	.159
Transcribir código	.474	.091	.113	.450	-.247	.023	.030	.029	.081
Codificar de manera modular	.466	-.217	-.024	.079	-.187	.216	-.440	-.422	.112
Registrar defectos de sintaxis	.442	.692	-.173	-.223	.059	-.140	.191	-.011	-.052
Registrar defectos lógicos	.544	.645	-.148	-.293	.170	-.115	.162	-.033	-.072
Identificar datos de entrada	.603	-.619	-.113	-.195	.109	-.174	.122	.056	.053
Registra defectos en ejecución	.568	.591	-.130	-.213	.185	.040	.076	-.024	-.113
Registrar tiempo de desarrollo	.434	.534	-.109	.002	.223	-.151	-.068	-.013	.025
Ejecución correcta del programa	.335	-.533	.040	.061	.403	.069	-.046	-.172	-.061
Uso de algoritmo	.252	-.240	.654	-.013	.049	-.280	.040	.170	-.044
Uso de diagrama de flujo	.193	.053	.632	-.243	-.110	.300	.234	.159	-.185
Codificar de manera secuencial	.302	.074	.592	-.411	.001	.115	-.043	-.285	.190
Uso de pseudocódigo	.326	.005	.474	-.008	.332	-.156	-.238	.210	-.313
Uso de diagrama de actividades	.388	.249	.176	.413	.190	.132	-.339	.270	.111
Modularizar programa lineal	.314	.173	.174	.276	-.516	-.298	.175	.031	.276
Verificar resultados correctos	.255	-.335	.173	.292	.414	.021	.290	-.296	.212
Uso de diagrama Nassi-Scheiderman	.357	.427	.402	-.111	-.027	.539	-.150	-.122	.130
Validación de datos de entrada	.359	-.061	.008	.293	.279	.321	.628	.012	.105
Uso de diagrama de clases	.362	.014	-.174	.173	.260	.342	-.424	.333	.089
Revisión de código (defectos de sintaxis)	.252	-.036	-.278	.111	-.278	.401	.257	.448	-.006
Uso de pruebas de escritorio	.347	.168	.296	.280	-.059	-.337	-.032	.138	.363

Fuente: Resultados con SPSS.

En la Imagen anterior, se puede observar que el componente 1, análisis de requisitos, en un programa se describe por la identificación de funciones/métodos, identificación de las fórmulas, prueba de validación de datos, programa claro y legible, identificación de datos de entrada y de salida. Es decir, entre más se realice la identificación de las variables anteriores

mejor será el análisis de requisitos. Sin embargo, entre más se verifique que los resultados sean correctos es porque no se realizó un buen análisis de requisitos. No obstante, para mejorar la interpretación de los componentes, se obtiene la matriz de componentes rotados con el método Varimax.

La matriz de componentes rotados convergió en 11 iteraciones mejorando la saturación de las variables, Para la agrupación de las variables se consideraron las saturaciones $> .400$ reduciendo las 31 variables en 9 componentes. Cabe destacar, que la variable “uso de pseudocódigo” se discrimina toda vez que no cumple con la saturación mínima, tal como se muestra en la Imagen 5.6. Con base en lo anterior, en los cursos de programación se deben considerar los nueve componentes resultantes, aplicando las variables en cada uno de ellos. No obstante, el componente 3 y el 7 se deberá aplicar considerando el paradigma de programación que se esté utilizando.

Imagen 5.6. Matriz de componentes rotados.

	Componente								
	1	2	3	4	5	6	7	8	9
Identificar datos de entrada	.869	-.030	-.029	.097	.030	.295	-.046	.050	-.030
identificar datos de salida	.851	-.008	-.043	.198	-.073	.234	-.001	.054	-.066
identificar formulas	.787	.133	.209	-.019	.121	.007	.184	.080	.149
Identificar funciones/métodos	.746	.233	-.058	.098	.114	-.119	.302	.192	.064
Identificar variables a utilizar	.574	.058	.161	.233	.184	-.002	-.165	.122	.261
Registrar defectos lógicos	.098	.931	.104	.061	.047	-.019	-.004	.002	.020
Registrar defectos de sintaxis	-.003	.883	.047	.056	.120	-.085	-.056	-.020	.084
Registra defectos en ejecución	.075	.849	.134	.131	-.019	.015	.132	.068	.067
Registrar tiempo de desarrollo	-.006	.683	-.048	.069	.159	.019	.229	.011	-.107
Prueba de validación de datos	.316	.423	-.080	.141	.031	.386	.168	.284	.255
Uso de diagrama de flujo	.045	.011	.814	.127	-.002	.009	-.041	-.091	.152
Codificar de manera secuencial	.112	.158	.688	-.118	.098	.085	-.078	.347	-.285
Uso de diagrama Nassi-Scheiderman	-.207	.321	.613	.012	.043	.045	.321	.435	.107
Uso de algoritmo	.301	-.174	.494	.139	.320	.090	.039	-.268	-.308
Uso de pseudocódigo	.206	.103	.393	.275	.005	.001	.379	-.295	-.389
Operar programas	.106	-.019	.140	.770	.010	.196	.085	.157	.238
Hacer seguimiento de variables	.160	.215	.133	.709	.248	-.033	-.001	-.026	-.099
Verificar que el programa sea claro y legible	.264	.339	-.226	.545	.060	.167	.142	.174	-.093
Uso de comentarios en el código	.278	.238	-.089	.411	.229	-.133	.051	.391	.084
Modularizar programa lineal	.024	.076	.044	.155	.780	-.070	-.151	.089	.145
Uso de pruebas de escritorio	.075	.130	.086	-.014	.696	.076	.201	-.053	-.138
Transcribir código	.030	.072	.040	.366	.503	.143	.208	.146	.194
Depurar código	.281	.411	-.127	.106	.423	.255	.082	.132	-.148
Verificar resultados correctos	.126	-.090	.029	.036	.112	.785	.015	.033	-.161
Validación de datos de entrada	.057	.164	.122	.089	.071	.746	.023	-.113	.420
Ejecución correcta del programa	.419	-.161	.003	.163	-.205	.497	.170	.087	-.225
Uso de diagrama de clases	.186	.100	-.077	.007	-.088	.035	.759	.123	.157
Uso de diagrama de actividades	-.057	.159	.090	.144	.293	.080	.720	.004	-.001
Codificar de manera modular	.217	-.069	-.011	.269	.064	.077	.214	.734	-.134
Conocer la sintaxis del lenguaje	.453	.087	.187	.037	.032	-.043	-.129	.576	.129
Revisión de código (defectos de sintaxis)	.191	.018	-.008	.096	.035	-.016	.146	-.026	.768

Fuente: Resultados con SPSS.

Finalmente se contrastaron las prácticas obtenidas con las de los docentes, concordando que en el análisis se identifican los elementos; en el diseño se usan diagramas de flujo, algoritmo y modelado UML; en la codificación se usan comentarios, conocimiento de la sintaxis y revisión de código; en la compilación se incluyen bitácoras de defectos y en las pruebas se verifican resultados. Así, las mejores prácticas de ingeniería de software se permearon en las diferentes fases del CV, ver Tabla 5.9, para proporcionar un conjunto de buenas prácticas para fomentar el uso de estas en los cursos de programación en el ámbito universitario.

Tabla 5.9. Mejores prácticas de ingeniería de software y su inserción en el CV

Fase	Prácticas
Requerimientos	Comprender el problema.
Análisis	Identificar las variables a utilizar. Identificar los datos de entrada. Identificar los datos de salida. Identificar fórmulas. Identificar funciones/métodos.
Practica complementaria	Registrar tiempo de desarrollo (análisis).
Diseño	Diagrama de flujo. Diagrama de Nassi-Scheiderman. Algoritmo. Diagrama de clases. Diagrama de actividades.
Prácticas complementarias	Diseñar prueba de validación. Pruebas de escritorio. Registrar tiempo de desarrollo (diseño).
Codificación	Poner comentarios. Conocer la sintaxis del lenguaje. Verificar que el programa sea claro y legible. Codificar de manera secuencial. Codificar de manera modular. Revisión de código.
Prácticas complementarias	Operar programas. Seguimiento de variables. Depurar código. Transcribir código. Modularizar un programa lineal. Registrar tiempo de desarrollo (codificación).
Compilación	Registra defectos lógicos. Registrar defectos de sintaxis. Registrar defectos en ejecución. Registrar tiempo de desarrollo (compilación).
Pruebas	Verificar resultados correctos. Validación de datos de entrada. Ejecución correcta del programa.
Práctica complementaria	Registrar tiempo de desarrollo (pruebas).

Fuente: Elaboración propia.

Conclusiones.

En este capítulo se presentaron los resultados derivados de la investigación realizada a nivel nacional en las universidades públicas, que permitió identificar las mejores prácticas de ingeniería de software que los docentes utilizan en los cursos de programación, en las diferentes fases del CV, tomando como base el CV en cascada.

Si bien se pudo observar que 70 de los 84 docentes, que corresponde al 80.95%, indican la importancia de dar a conocer a los alumnos el CV en la enseñanza de la programación; sólo el 47.61% siguen un proceso para el desarrollo de programas.

Por otro lado, con el análisis realizado se presume que la mayoría de los docentes tienen más dominio del uso de prácticas en las fases de diseño, codificación y pruebas, cuando se desarrolla un programa.

Las prácticas obtenidas con la investigación se clasificaron con base en su naturaleza, descartándose aquellas prácticas que están dirigidas al desarrollo del software, y se incluyeron en el CV; el cual se propone sea considerado para el desarrollo de programas en el ámbito educativo.

Con base en lo anterior, el supuesto se fundamenta con los resultados obtenidos en la investigación, ya que el 52% de los docentes en la fase de requerimientos, el 69.04% de los docentes en la fase de análisis, el 54.75% de los docentes en la fase de diseño, el 51.09 de los docentes en la fase de codificación, el 61.90% de los docentes en la fase de compilación, el 54.75% de los docentes en la fase de pruebas y el 77.37% de los docentes en la fase de mantenimiento no utilizan las mejores prácticas identificadas en el ciclo de vida de desarrollo de programas en las Universidades públicas en México. Además, en promedio un 33.69% de docentes no respondieron por fase, se infiere que es porque no se utilizan prácticas de ingeniería de software o bien no se apegan a un proceso para el desarrollo de programa en los cursos de programación.

Los resultados obtenidos servirán de guía para el siguiente capítulo debido a que se tienen que establecer condiciones básicas a cumplir para las diferentes actividades del CV, a estas condiciones se les denominan criterios de calidad básicos, mismos que serán expuestos en el siguiente capítulo.

Capítulo 6. Resultados de los criterios de calidad para los cursos de programación.

Introducción.

En los cursos de programación además de seguir un proceso de desarrollo adecuado, CV, que fomente el uso de buenas prácticas de ingeniería de software en etapas tempranas en la formación profesional de los estudiantes, se deben establecer criterios de calidad (CC) con la finalidad de desarrollar programas con la calidad básica necesaria, es decir que el análisis sea completo, el diseño sea adecuado, el código sea limpio y se cumpla en todo momento con los requisitos predefinidos para obtener los resultados correctos. Si bien es cierto que los CC del software se encuentran inmersos en estándares y modelos de calidad, ver sección 2.3, orientados al desarrollo de software en el ámbito empresarial, también es cierto que deben estar presentes en los cursos de programación, con la finalidad de que los futuros desarrolladores maduren su proceso de desarrollo, se involucren con prácticas de calidad y mejoren sus programas.

En este sentido, una de las dificultades que se presenta en los cursos de programación es que los estudiantes invierten tiempo buscando soluciones para desarrollar programas que funcionen de manera correcta, lo cual no logran, siendo una causa de fracaso para los estudiantes (Özmen & Altun, 2014). Por ello, y por la problemática planteada en este trabajo doctoral se considera importante que en los cursos de programación se consideren CC ya que es un reto para los docentes lograr que los estudiantes desarrollen código con calidad (Krusche et al., 2016).

Considerando lo anterior, la pregunta de investigación que guía este capítulo es: **¿Cuáles son los criterios de calidad para desarrollo de programas en los cursos de programación?**

Para responder a esta pregunta se realizó una revisión de literatura sobre modelos de calidad, ver sección 2.4, y se aplicó un cuestionario a 84 docentes de 38 universidades públicas de México para cumplir con el objetivo: **Definir los criterios de calidad que puedan aplicarse en el desarrollo de programas en los cursos de programación con base en los principales modelos de calidad del software.**

Con las respuestas obtenidas de los 84 docentes se hizo un análisis con el método de escalamiento de Likert seleccionando las favorables y se determinaron los CC considerando su definición y su clasificación en los modelos de calidad, tropicalizándolos para los cursos

de programación. Además, con los resultados obtenidos se pudo comprobar el supuesto de este capítulo: **Si se analizan los modelos de calidad del software y las especificaciones dadas por los docentes, se podrán determinar criterios de calidad para el desarrollo de programas en los cursos de programación, que podrán alinearse con las mejores prácticas de ingeniería de software en las diferentes fases del CV.** No obstante, la determinación de los criterios de calidad fue básica y se estableció uno por cada fase del CV. Es importante mencionar que una especificación para el contexto de este capítulo son las actividades que realiza el docente para verificar que el programa sea correcto y las actividades que se solicitan a los estudiantes realicen durante el desarrollo del programa; en ambos casos orientadas a la calidad.

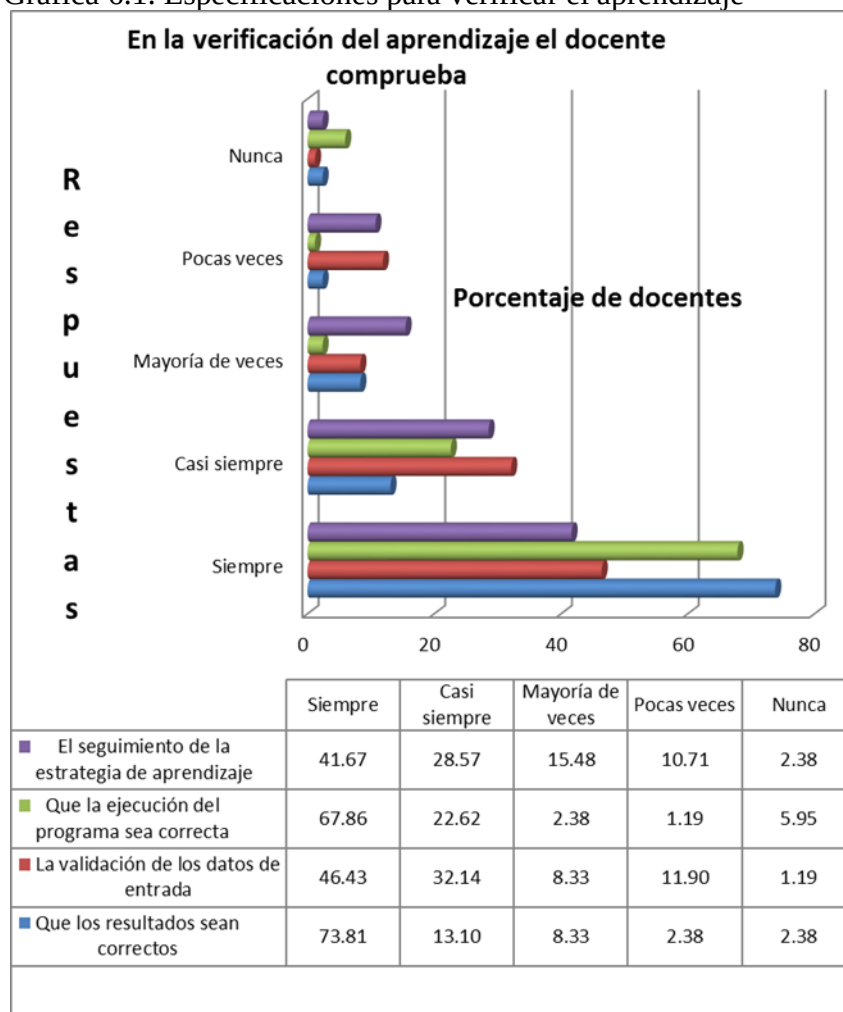
6.1 Especificaciones que los docentes solicitan y utilizan en los cursos de programación.

En los cursos de programación los docentes tienen un rol esencial en la formación de los estudiantes, por lo que es importante que indiquen las especificaciones a considerar durante el desarrollo de programas, de forma tal que se conozcan con antelación los criterios de evaluación para que un programa se considere correcto, además de que al cumplir con las especificaciones establecidas se puede obtener una solución correcta.

Para conocer las especificaciones que los docentes solicitan a los estudiantes en los cursos de programación, el cuestionario cuenta con 27 ítems en escala de Likert repartidas en siete aspectos, ver cuestionario, Anexo I. Sin embargo, el último aspecto que trata de las actividades que utilizan los docentes en la didáctica, no es considerado debido a que se utilizaría en conjunto con las habilidades, lo cual está fuera del alcance de este trabajo doctoral, ya que su delimitación no lo considera.

Por lo que se refiere al primer aspecto, se enfoca en la verificación del aprendizaje, en promedio el 70.83% de docentes indican que verifican que la ejecución del programa y los resultados sean correctos, ver Gráfica 6.1, pero sólo el 46.43% que corresponde a 39 docentes verifican que los datos de entrada sean los solicitados en el problema. Con fundamento en la entrevista no formal realizada a 55 docentes de 29 universidades, ver sección 1.1, indicaron que la razón por la que no se verifica la validación de los datos de entrada en los programas, es por la cantidad de estudiantes que existen en los grupos a los que se imparte clase.

Gráfica 6.1. Especificaciones para verificar el aprendizaje



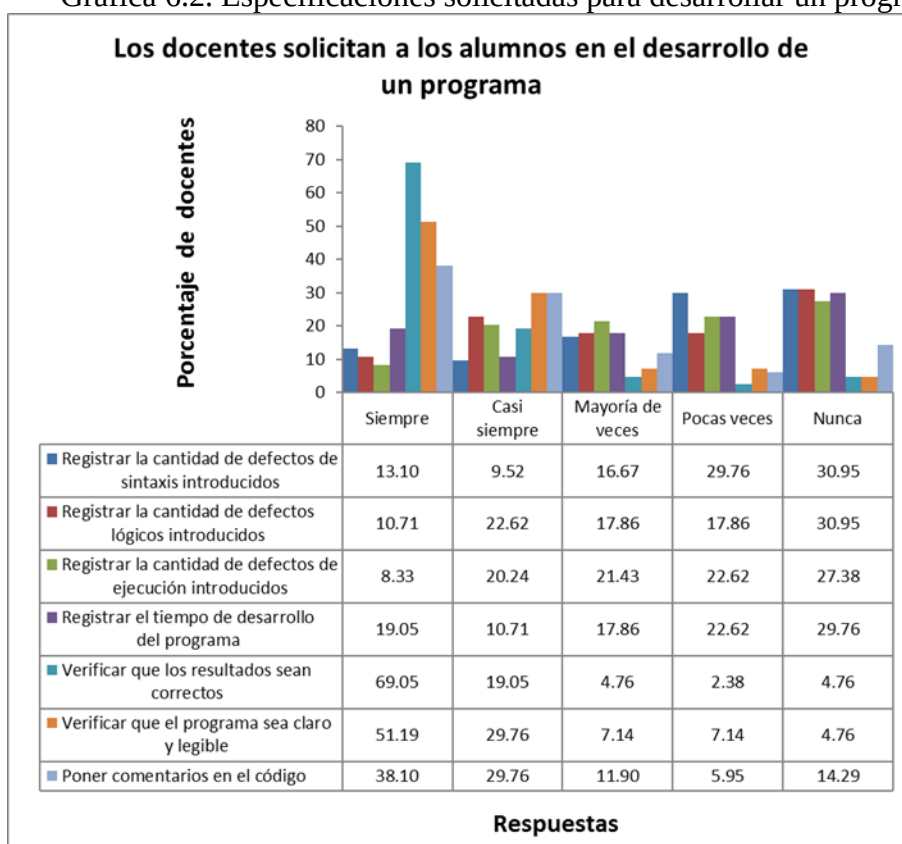
Fuente: Elaboración propia.

El segundo aspecto está enfocado a las especificaciones que los docentes solicitan al desarrollar un programa que se da desde dos perspectivas, con bitácoras para el registro de defectos y de tiempos, así como en la verificación del funcionamiento del programa y de la legibilidad del código.

Como se puede observar en la Gráfica 6.2, para los docentes no es importante que los estudiantes lleven una bitácora del registro de defectos que se han inyectado en el código por los errores cometidos por el estudiante, así como el registro de los tiempos de desarrollo de un programa, ya que sólo entre el 8% y el 19% de docentes lo solicitan. Lo anterior tiene lugar debido a que los estudiantes antes de llevar este tipo de registros deben conocer y formarse el hábito de seguir un CV que permita generar soluciones correctas.

Por otro lado, más del 50% de los docentes ponen mayor atención en verificar que los resultados sean correctos, es decir que la salida del programa es la solicitada, y que la codificación del programa sea clara y legible; lo cual es consistente con lo solicitado por los docentes en el análisis de un programa.

Gráfica 6.2. Especificaciones solicitadas para desarrollar un programa



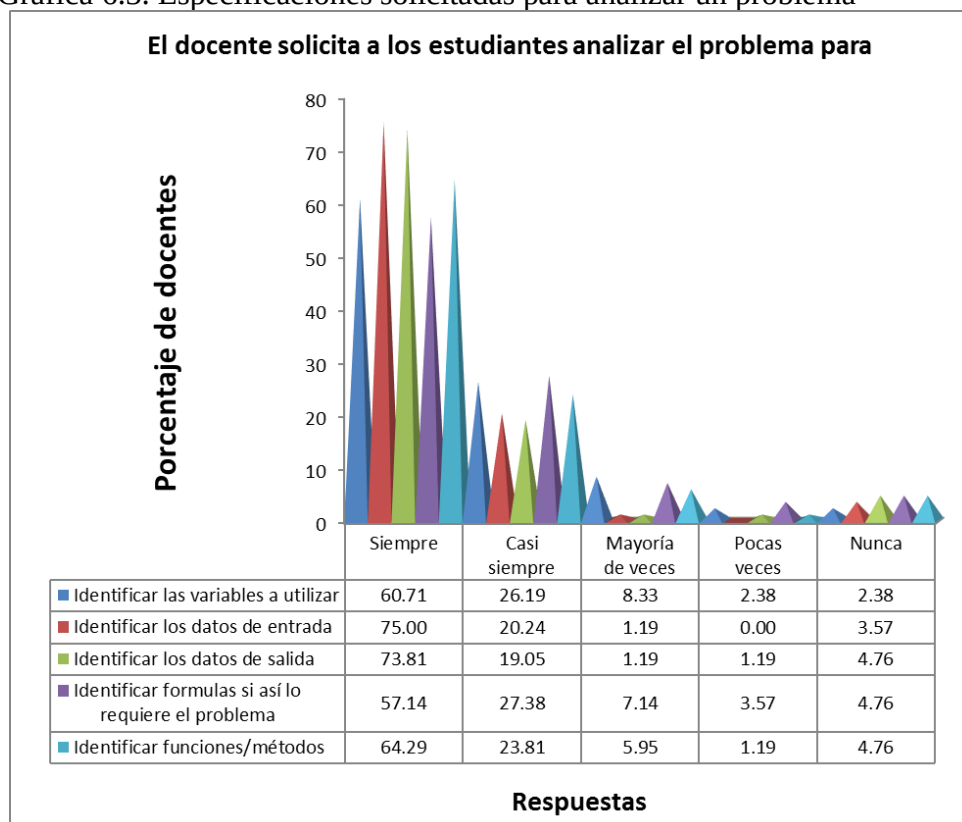
Fuente: Elaboración propia.

El tercer aspecto está orientado al análisis del problema en donde el estudiante después de comprender el problema debe abstraer los elementos que utilizará para dar solución a este. Cabe mencionar, que los elementos se obtienen con base en los requisitos predefinidos del problema a resolver.

En promedio el 66.19% de docentes solicitan se identifiquen los datos de entrada y de salida, las variables, funciones y métodos, y las fórmulas, en caso de existir, para realizar cálculos, ver Gráfica 6.3.

La identificación de los elementos mencionados en el párrafo anterior, son importantes debido a que se analiza el problema desde una perspectiva sistémica del mismo, de esta manera se fomenta en los alumnos que antes de realizar un diseño se debe comprender el problema a resolver y considerar los elementos con los que darán solución utilizando alguna técnica de diseño.

Gráfica 6.3. Especificaciones solicitadas para analizar un problema



Fuente: Elaboración propia.

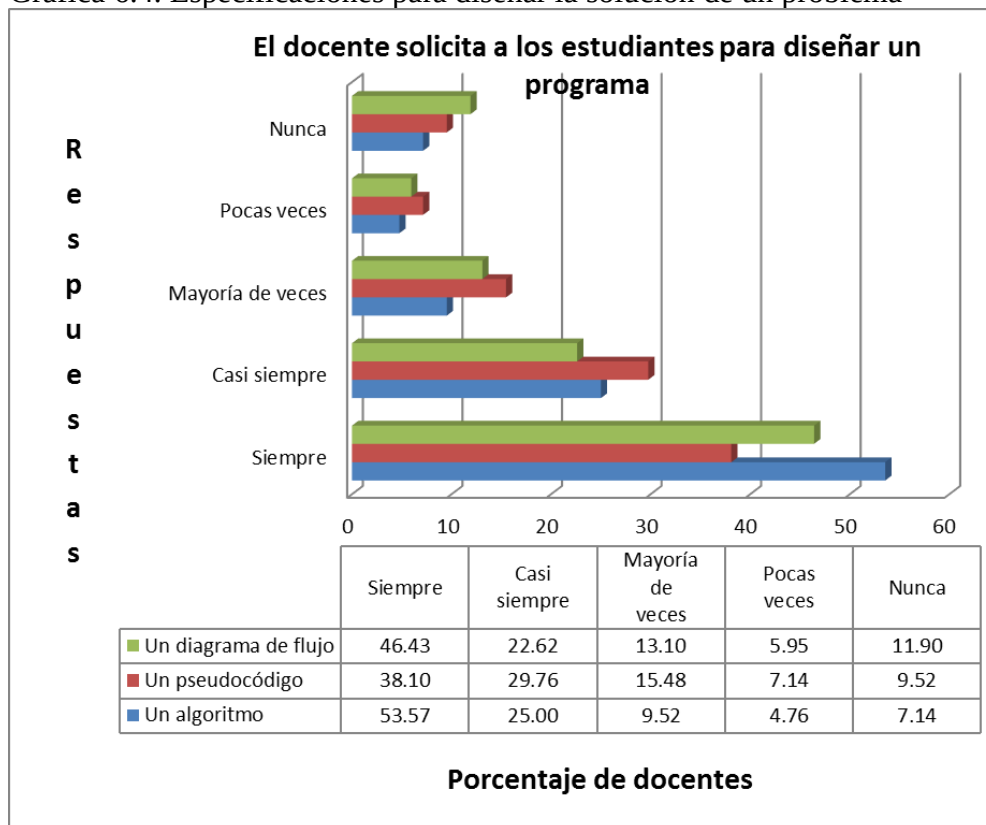
El cuarto y quinto aspecto se enfocan en la codificación, el primero está orientada en conocer qué se les solicita a los estudiantes previo a la codificación de un programa, diseño, ver Gráfica 6.4, y el segundo se dirige en el cómo se les solicita se codifique, ver Gráfica 6.5.

En la Gráfica 6.4, se puede observar que en promedio 9.527% de docentes no solicitan se realice un diseño, pero no es determinante debido a que podrían especificar otra técnica de diseño que no se encuentra en el instrumento.

Considerando lo anterior, la mayoría de los docentes solicitan se realice el diseño utilizando las técnicas de diseño: algoritmo, diagrama de flujo y pseudocódigo. Sin embargo, sin

importar la técnica de diseño utilizada, en todos los casos el diseño debe ser claro y legible, con base en las respuestas del 51.19% de docente; esto permite que el diseño pueda ser entendido por cualquier persona, ver Grafica 6.2.

Gráfica 6.4. Especificaciones para diseñar la solución de un problema

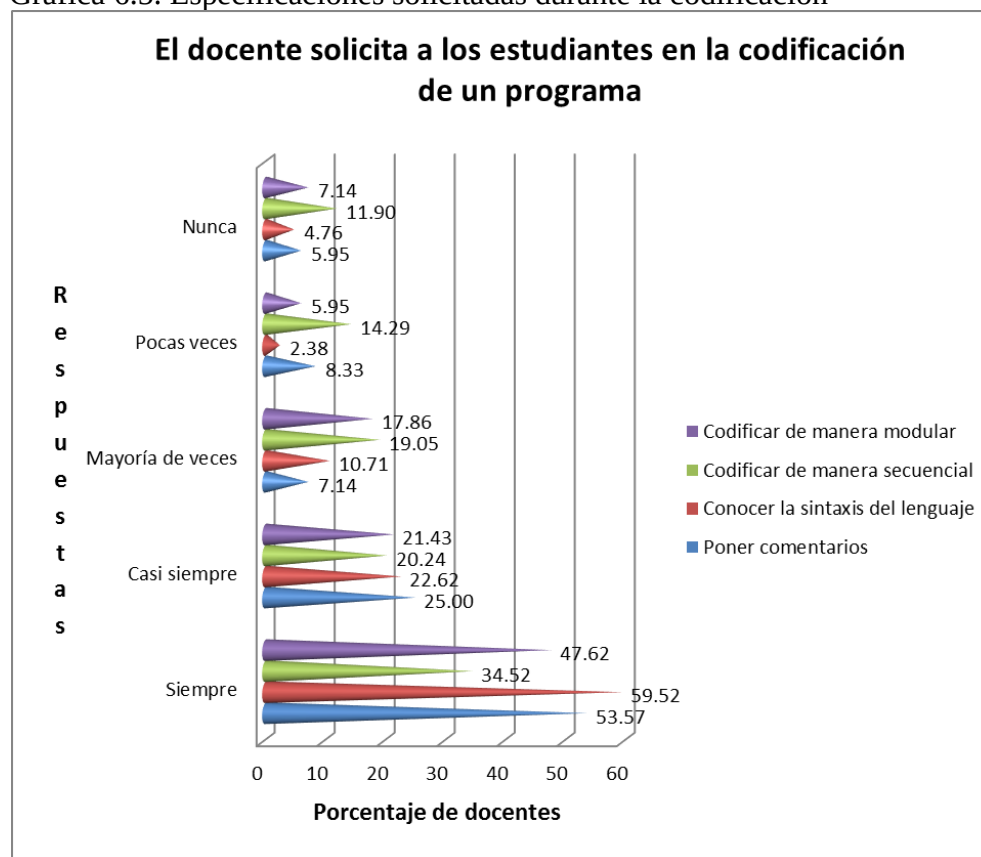


Fuente: Elaboración propia.

En la Gráfica 6.5 se puede observar que existe mayor tendencia en solicitar que la codificación sea modular y que es importante conocer la sintaxis del lenguaje. La codificación modular hace que el código sea más legible y entendible, además permite al estudiante hacer uso de la técnica divide y vencerás dividiendo el problema en subproblemas más manejables. Y, aunque el conocer la sintaxis del lenguaje es importante, no es impedimento que de no ser así se pueda codificar la solución apoyándose de algún material del lenguaje de programación en cuestión, siempre y cuando la solución diseñada sea correcta.

Con respecto a las respuestas dadas por los docentes para la especificación de comentarios, existe una inconsistencia debido a que existe una diferencia de 15.47 puntos porcentuales con respecto a la Gráfica 6.2, en donde se realizó la misma pregunta.

Gráfica 6.5. Especificaciones solicitadas durante la codificación



Fuente: Elaboración propia.

El sexto aspecto está dirigido a las pruebas que se realizan con los programas para validar los datos de entrada y corroborar que los resultados son correctos, ver Gráfica 6.6, lo cual resulta redundante con el primer aspecto. La razón de la redundancia fue corroborar que las respuestas proporcionadas fueran congruentes, en consecuencia, existe una diferencia de 7.14 puntos porcentuales para la validación de los datos de entrada y una diferencia de 2.38 puntos porcentuales para verificar que los resultados sean correctos. Teniendo en cuenta esta diferencia, es posible que se deba a la interpretación de la pregunta, ya que es más precisa la pregunta en el sexto aspecto.

Se infiere que el 46.43% de los docentes no validan los datos de entrada debido a que al corroborar que los resultados sean correctos sólo se prueban con los datos de entrada con base en los solicitados en los requisitos predefinidos del problema.

Gráfica 6.6. Especificaciones que utilizan los docentes para pruebas



Fuente: Elaboración propia.

6.2 Selección de las especificaciones con el método de escalamiento de Likert.

Con la finalidad de seleccionar las especificaciones, ítems, para alinearlos con el CV para el desarrollo de programas en los cursos de programación, se utilizó el método de escalamiento de Likert (Sampieri et al, 2006).

Es importante mencionar que, para efectos de poder obtener las puntuaciones de los ítems, no se evaluó cada instrumento de manera particular ya que con ello sólo se podía obtener la preferencia de cada docente. Así, la evaluación de los instrumentos se realizó en su conjunto, es decir considerando las 84 respuestas. Para la selección de los ítems, se consideró el valor de 2.5 como la puntuación mínima para indicar que el ítem presentó una calificación

favorable, ya que se utilizaron 5 categorías. En la Tabla 6.1, se presentan las respuestas de los ítems de 84 docentes para su valoración en un rango de 5 a 1, en la escala de Likert.

Tabla 6.1. Respuesta de las especificaciones solicitadas por los docentes en el desarrollo de programas en las universidades públicas en México 2015 – 2016

Criterio de calidad utilizada por el docente	5	4	3	2	1	Total
Cuando realiza la verificación del aprendizaje comprueba:						
Que los resultados sean correctos.	62	11	7	2	2	84
La validación de los datos de entrada.	39	27	7	10	1	84
Que la ejecución del programa sea correcta.	57	19	2	1	5	84
El seguimiento de la estrategia de aprendizaje.	35	24	13	9	2	83 ¹
Cuando los alumnos desarrollan un programa usted solicita:						
Registrar la cantidad de defectos de sintaxis introducidos.	11	8	14	25	26	84
Registrar la cantidad de defectos lógicos introducidos.	9	19	15	15	26	84
Registrar la cantidad de defectos de ejecución introducidos.	7	17	18	19	23	84
Registrar el tiempo de desarrollo del programa.	16	9	15	19	25	84
Verificar que los resultados sean correctos.	58	16	4	2	4	84
Verificar que el programa sea claro y legible.	43	25	6	6	4	84
Poner comentarios en el código.	32	25	10	5	12	84
Los alumnos con base a un problema deben analizarlo para:						
Identificar las variables a utilizar.	51	22	7	2	2	84
Identificar los datos de entrada.	63	17	1	0	3	84
Identificar los datos de salida.	62	16	1	1	4	84
Identificar formulas si así lo requiere el problema.	48	23	6	3	4	84
Identificar funciones/métodos.	54	20	5	1	4	84
Los alumnos, previo a la codificación deben diseñar, usando:						
Un algoritmo.	45	21	8	4	6	84
Un pseudocódigo.	32	25	13	6	8	84
Un diagrama de flujo.	39	19	11	5	10	84
En la codificación los alumnos deben:						
Poner comentarios.	45	21	6	7	5	84
Conocer la sintaxis del lenguaje.	50	19	9	2	4	84
Codificar de manera secuencial.	29	17	16	12	10	84
Codificar de manera modular.	40	18	15	5	6	84
Las pruebas que se realizan a un programa son para:						
Corroborar que los resultados sean correctos.	64	15	2	0	3	84
Corroborar que los datos de entrada han sido validados.	45	22	7	5	5	84

Nota: Los valores asociados a la escala de Likert de las 5 categorías son; 5 Siempre, 4 Casi siempre, 3 Mayoría de las veces, 2 Pocas veces, 1 Nunca. No existe la fase de compilación debido a que fue eliminada con base en el juicio de expertos.

¹Un docente no dio respuesta.

Fuente: Elaboración propia.

Tomando como base las respuestas proporcionadas por los docentes en cada una de las 27 especificaciones, ítems asociados a cinco categorías, se procedió a obtener la puntuación en cada una de las categorías. Para ello se multiplicó cada ocurrencia de respuesta por el valor de cada categoría del 1 al 5 según corresponda. Así, por ejemplo, para obtener la puntuación del ítem “Que los resultados sean correctos” se multiplicó el valor de 62, ver Tabla 6.1, que corresponde al número de respuestas de la categoría 5, “Siempre”, teniendo como resultado 310, ver Tabla 6.2. Para obtener la puntuación total para cada especificación, se dividió el puntaje de las respuestas en la categoría 5 “Siempre” entre 84, a excepción del seguimiento de la estrategia de aprendizaje que se dividió entre 83, ya que un docente no contestó la pregunta.

En la selección de las especificaciones, se consideraron las puntuaciones que obtuvieron un valor igual o mayor a 2.50, ya que el 5 tiene un significado de muy favorable y 1 de muy desfavorable, con base en que 5 corresponde a “Siempre” y 1 corresponde a “Nunca”. En la Tabla 6.2, se pueden observar las puntuaciones obtenidas para cada una de las especificaciones, ítems, seleccionando solo 14, sombreadas en color gris, que obtuvieron una puntuación favorable.

Tabla 6.2. Puntuación de las especificaciones utilizando el método de escalamiento de Likert

Especificación solicitada por el docente	5	4	3	2	1	Puntuación
Cuando realiza la verificación del aprendizaje comprueba:						
Que los resultados sean correctos.	310	44	21	4	2	3.69
La validación de los datos de entrada.	195	108	21	20	1	2.32
Que la ejecución del programa sea correcta.	285	76	6	2	5	3.39
El seguimiento de la estrategia de aprendizaje.	175	96	39	18	2	2.11 ¹
Cuando los alumnos desarrollan un programa usted solicita:						
Registrar la cantidad de defectos de sintaxis introducidos.	55	32	42	50	26	0.65
Registrar la cantidad de defectos lógicos introducidos.	45	76	45	30	26	0.54
Registrar la cantidad de defectos de ejecución introducidos.	35	68	54	38	23	0.42
Registrar el tiempo de desarrollo del programa.	80	36	45	38	25	0.95
Verificar que los resultados sean correctos.	290	64	12	4	4	3.45
Verificar que el programa sea claro y legible.	215	100	18	12	4	2.56
Poner comentarios en el código.	160	100	30	10	12	1.90
Los alumnos con base en un problema deben analizarlo para:						
Identificar las variables a utilizar.	255	88	21	4	2	3.04
Identificar los datos de entrada.	315	68	3	0	3	3.75

Identificar los datos de salida.	310	64	3	2	4	3.69
Identificar formulas si así lo requiere el problema.	240	92	18	6	4	2.86
Identificar funciones/métodos.	270	80	15	2	4	3.21
Los alumnos previos a la codificación deben diseñar, usando:						
Un algoritmo.	225	84	24	8	6	2.68
Un pseudocódigo.	160	100	39	12	8	1.90
Un diagrama de flujo.	195	76	33	10	10	2.32
En la codificación los alumnos deben:						
Poner comentarios.	225	84	18	14	5	2.68
Conocer la sintaxis del lenguaje.	250	76	27	4	4	2.98
Codificar de manera secuencial.	145	68	48	24	10	1.73
Codificar de manera modular.	200	72	45	10	6	2.38
Las pruebas que se realizan a un programa es para:						
Corroborar que los resultados sean correctos.	320	60	6	0	3	3.81
Corroborar que los datos de entrada han sido validados.	225	88	21	10	5	2.68

Nota: Los valores asociados a la escala de Likert de las 5 categorías son; 5 Siempre, 4 Casi siempre, 3 Mayoría de las veces, 2 Pocas veces, 1 Nunca.

¹Para obtener el resultado se realizó la división entre 83, ya que se un valor cuantitativo que es la cantidad de respuestas de esta especificación.

Fuente: Elaboración propia.

Considerando que las especificaciones sombreadas de la Tabla 6.2 son las seleccionadas, se incluyeron en cada una de las fases del CV, discriminando las especificaciones redundantes presentes en diferentes interrogantes, ver Tabla 6.3.

Tabla 6.3. Especificaciones por fases del ciclo de vida.

Fase	Especificación.
Requerimientos	Sin especificación. ¹
Análisis	Identificar las variables a utilizar. Identificar los datos de entrada. Identificar los datos de salida. Identificar fórmulas. Identificar funciones/métodos.
Diseño	Utilizar como técnica de diseño un algoritmo.
Codificación	Poner comentarios. Conocer la sintaxis del lenguaje. Verificar que el programa sea claro y legible.
Compilación	Sin especificación. ²
Pruebas	Corroborar que los resultados sean correctos. Corroborar que los datos de entrada han sido validados.
Mantenimiento	Sin criterios.

Nota: Las especificaciones fueron seleccionadas con base en las respuestas proporcionadas por los docentes a nivel nacional en el periodo 2015 – 2016.

¹ no existen especificaciones porque los requisitos ya están predefinidos.

² no existen especificaciones ya que el ítem fue eliminado con base al juicio de expertos.

Fuente: Elaboración propia.

6.3 Definición de los criterios de calidad en las fases del ciclo de vida.

Para definir los CC se consideró la revisión de literatura de los modelos de calidad para la evaluación de los productos de software, ver sección 2.3, se seleccionaron los CC que mejor se adaptaron a las especificaciones en los cursos de programación, ver Tabla 6.4. Como resultado los criterios se clasificaron en cuatro colores: el color verde corresponde al CC entendible, el color rojo al CC exactitud, el color amarillo al CC completitud y el color azul al CC consistencia. Para la clasificación de los CC se consideró el objetivo de cada uno con base en la definición de cada subcaracterística, de su clasificación en el modelo y de su adaptación en el proceso de desarrollo.

Tabla 6.4. Criterios de calidad que se pueden adaptar en los cursos de programación

Modelo de calidad	Criterio	Descripción
ISO 9126	Comprensibilidad	Capacidad del producto software para permitir al usuario comprender si el software es adecuado y cómo se puede usar para tareas particulares y condiciones de uso.
	Exactitud	Capacidad del producto software para proporcionar los resultados correctos con el grado de precisión necesario.
	Satisfacción	Capacidad del producto software para satisfacer a los usuarios en sus actividades realizadas al hacer uso de este.
ISO 25010	Idoneidad reconocida	Grado en que los usuarios pueden reconocer si un producto o sistema es apropiado para sus necesidades.
	Satisfacción	Grado en que se satisfacen las necesidades del usuario cuando se utiliza un producto o sistema en un contexto de uso específico.
McCall	Completitud	Esos atributos del software que proporcionan la implementación completa de las funciones requeridas.
	Consistencia	Los atributos del software que proporcionan un diseño uniforme y técnicas de implementación y notación
	Exactitud	Esos atributos del software que proporcionan la precisión requerida en los cálculos y resultados.
Bohem	Comprensibilidad	El código posee comprensión en la medida en que su propósito sea claro para el inspector.
	Consistencia	El código posee consistencia interna en la medida en que contiene notación, terminología y simbología uniformes dentro de sí mismo, y consistencia externa en la medida en que el contenido sea trazable a los requisitos.
	Exactitud	El código posee precisión en la medida en que sus resultados son lo suficientemente precisos para satisfacer su uso previsto.
	Legibilidad	El código posee legibilidad en la medida en que su función se distingue fácilmente al leer el código.
	Testabilidad	El código tiene capacidad de prueba en la medida en que facilita el establecimiento de criterios de verificación y respalda la evaluación de su desempeño.
Dromey	Descriptivas	Describen el software agregando comentarios explícitos a un programa para documentar cómo la implementación

		realiza su funcionalidad deseada mediante la manipulación de variables con propiedades prescritas.
	Estructuración	Se centran en la forma de en qué las declaraciones individuales y los componentes de la declaración se implementan y la forma en que las declaraciones y los bloques de declaración se componen, se relacionan entre sí y se utilizan.
	Funcionalidad	Un programa debe funcionar adecuadamente y debe ser preciso con base en especificación.

Fuente: Elaboración propia. Con base en (BS ISO/IEC, 2011; Bohem et al., 1976; Dromey, 1995; ISO/IEC FDIS, 2000; ISO/IEC JTC1, 2001; ISO/IEC, 2016; McCall et al., 1977; Álvarez et al., 2016; Callejas-Cuervo et al., 2017)

En la Tabla 6.5 se encuentra la descripción de cada uno de los CC. Se debe agregar que se incluyeron otro dos CC, el primero es comprensibilidad el cual no pertenece a ninguna fase, pero es importante porque si un problema no se comprende no se puede crear una solución correcta y el segundo es corrección, que se relaciona con el mantenimiento correctivo para corregir los errores en la fase de compilación y pruebas.

Tabla 6.5. Criterios de calidad para los cursos de programación.

Criterio de calidad	Descripción
Comprensión	Grado en que se toma conciencia del problema para dar solución por medio de la programación.
Complejidad	Grado en que la especificación del programa cumple con todos los requisitos predefinidos y la funcionalidad requerida.
Consistencia	Grado en que se usa un diseño uniforme que tenga trazabilidad con la codificación.
Entendible	Grado en que un código es fácil de entender, de modificar y de reutilizar.
Corrección	Grado en que un programa se ejecuta de manera correcta y está listo para ser probado
Exactitud	Capacidad que tiene el programa para proporcionar los resultados correctos.

Nota: La descripción de cada criterio de calidad fue adaptado al ciclo de vida para los cursos de programación.

Fuente: Elaboración propia.

Considerando los CC de la Tabla 6.5 y las especificaciones de la Tabla 6.3, se contrastaron con las mejores prácticas, existiendo una convergencia entre los resultados de las buenas prácticas en las diferentes fases del CV. Finalmente, con los resultados obtenidos del escalamiento tipo Likert, de los CC y de las mejores prácticas de Ingeniería de software, se incluyeron se incluyeron los CC en las fases del CV, ver Tabla 6.6.

Tabla 6.6. Inclusión de los criterios de calidad en el ciclo de vida para los cursos de programación.

Fase	Prácticas	Criterio de calidad
Requerimientos	Comprender el problema.	Sin criterio de calidad
Análisis	Identificar las variables a utilizar. Identificar los datos de entrada. Identificar los datos de salida. Identificar fórmulas. Identificar funciones/métodos.	Complejidad
Práctica complementaria	Registrar tiempo de desarrollo (análisis).	
Diseño	Diagrama de flujo. Diagrama de Nassi-Scheiderman. Algoritmo. Diagrama de clases. Diagrama de actividades.	Consistencia
Prácticas complementarias	Diseñar prueba de validación. Pruebas de escritorio. Registrar tiempo de desarrollo.	
Codificación	Poner comentarios. Conocer la sintaxis del lenguaje. Verificar que el programa sea claro y legible. Codificar de manera secuencial. Codificar de manera modular. Revisión de código.	Entendible
Prácticas complementarias	Operar programas. Seguimiento de variables. Depurar código. Transcribir código. Modularizar un programa lineal. Registrar tiempo de desarrollo.	
Compilación	Registrar defectos lógicos. Registrar defectos de sintaxis. Registrar defectos en ejecución. Registrar tiempo de desarrollo	Corrección
Pruebas	Verificar resultados correctos. Validación de datos de entrada. Ejecución correcta del programa.	Exactitud
Práctica complementaria	Registrar tiempo de desarrollo.	

Fuente: Elaboración propia.

Conclusiones.

En este capítulo se dieron a conocer los resultados obtenidos de las especificaciones que los docentes utilizan en los cursos de programación en las universidades públicas en México, y fueron consideradas como base aunado a los CC de los modelos de calidad revisados, ver sección 2.3, para definir los CC para el desarrollo de programas en los cursos de programación.

Considerando los resultados de las especificaciones se pudo observar que las especificaciones que se encontraban en diferentes aspectos fueron inconsistentes, un ejemplo es el uso de comentarios en el código con una diferencia de 15.47 puntos porcentuales entre el aspecto 2, especificaciones al desarrollar un programa, y el aspecto 5, especificaciones solicitadas en la codificación. Esta diferencia se atribuye al contexto de la pregunta que pudo haber sido interpretada de diferentes maneras. No obstante, al hacer la selección de las especificaciones, las mismas fueron identificadas como favorables.

Se debe agregar que el 56% de las especificaciones tuvieron una puntuación favorable y después de ser contratadas con los resultados de las mejores prácticas se adhirieron a las fases del CV, empero, fueron convergentes por lo que no se adicionó ninguna.

Por lo que se refiere a los CC, se definieron 4 después de hacer una revisión de los CC de modelos de calidad del software, ver sección 2.4. Los criterios que se definieron son: comprensión, completitud, consistencia, entendible, corrección y exactitud; estos criterios no son determinantes para desarrollar software de calidad a nivel empresarial, pero sirven de apoyo en los cursos de programación para que los estudiantes desarrollen programas, haciendo conciencia de que la calidad es primordial para crear programas correctos y funcionales.

Teniendo en cuenta los CC, se adhirieron en las fases del CV que se considera importante debido a que los estudiantes se limitan en dar solución a un problema, sin considerar un análisis exhaustivo, un diseño coherente con el análisis que sirva de base en la codificación, un código entendible, la validación de entrada de datos y la verificación de resultados correctos. En consecuencia, el supuesto se comprueba; sin embargo, sólo se determinó un CC por fase, que sirve como punto de partida para determinar otros e incluso proponer un modelo de calidad para los cursos de programación.

En conclusión, el contar con un CV que considere CC, aunque de manera básica, para permitirá potenciar habilidades duras o específicas de los estudiantes, contribuyendo así a una formación más integral.

Capítulo 7. Modelo para el aseguramiento de la calidad con prácticas de Ingeniería de Software como apoyo en la programación orientada a objetos.

Introducción.

En este capítulo se describe la propuesta del modelo MoACPOO para Asegurar la mejora de la Calidad en el desarrollo de programas en los cursos de Programación Orientada a Objetos que está sustentado en los resultados obtenidos de las mejores prácticas de ingeniería de software, ver capítulo 5, y de los criterios de calidad, ver capítulo 6, al aplicar la metodología de la investigación, ver capítulo 4. Hay que mencionar, además que el modelo MoACPOO incluye un ciclo de vida que se compone de cinco fases: análisis, diseño, codificación, compilación y pruebas, que tienen asociadas buenas prácticas de ingeniería de software, un artefacto y un criterio de calidad que se acompaña de métricas. También, considera como requisito la comprensión del problema para poder llevar a cabo el desarrollo de un programa. El propósito del modelo es contribuir en la mejora de los programas en los cursos de programación orientada a objetos, al madurar el proceso de desarrollo de los estudiantes que son los futuros desarrolladores.

7.1 Modelo MoACPOO

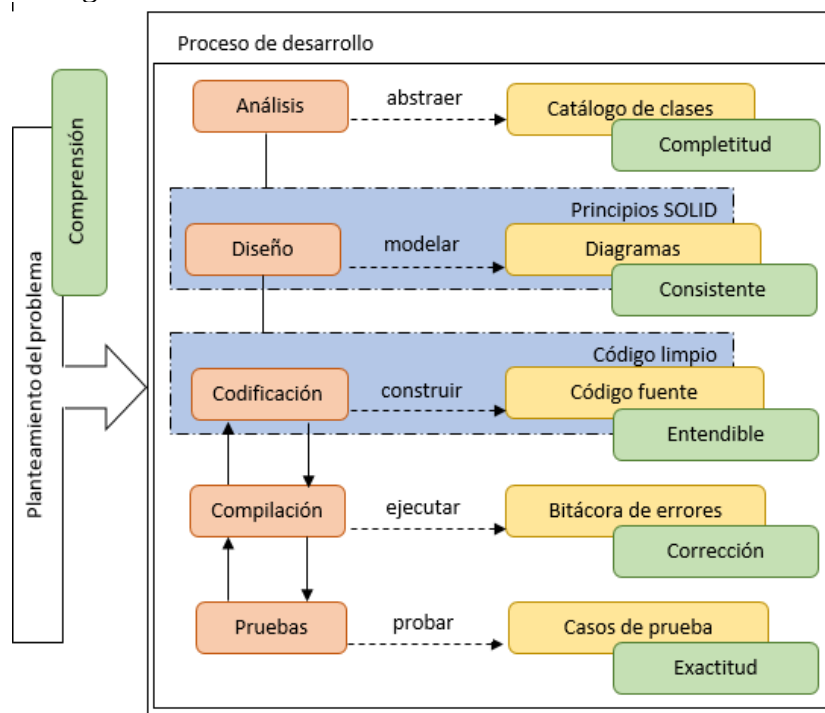
El modelo propuesto se construyó con base en los resultados de la investigación realizada, en este trabajo doctoral, de las prácticas de ingeniería de software que los docentes en México utilizan en los cursos de programación, en la educación profesional, obteniendo las mejores. Los resultados se obtuvieron al aplicar una encuesta a 84 docentes de 38 universidades públicas incorporadas a la Asociación Nacional de Universidades e Instituciones de Educación Superior (ANUIES) y se sustentaron con el análisis de datos mediante estadística descriptiva y de componentes principales, ver capítulo 5. Así mismo, se determinaron los criterios de calidad para el proceso de desarrollo en la programación, ver capítulo 6, considerando las buenas prácticas de ingeniería de software obtenidas de la investigación realizada y de la revisión previa de la literatura de los modelos de calidad del software ISO/IEC 9126-1 (ISO/IEC FDIS, 2000), (ISO/IEC JTC1, 2001)ISO/IEC 25000 (Suárez Lorenzo & Gárzas Parra, 2014), ISO/IEC 25010:2011 (BS ISO/IEC, 2011) , ISO/IEC 25023:2016 (ISO/IEC, 2016), McCall (McCall et al., 1977), Bohem (Bohem et al. 1976),

Dromey (Dromey, 1995) y de la comparación de estos modelos (Álvarez C et al., 2016; Callejas-Cuervo et al., 2017). Así, los resultados de la investigación se apoyan de las teorías del positivismo, del empirismo, y de los datos.

En consecuencia, la contribución de este trabajo doctoral son las prácticas de ingeniería de software y los criterios de calidad que se permearon en un ciclo de vida tomando como base el ciclo de vida en cascada (Pressman, 2010) que sustentan el modelo MoACPOO, que tiene como propósito coadyuvar en la maduración del proceso de desarrollo en la programación orientado a objetos e involucrar a los futuros desarrolladores con buenas prácticas que les permita mejorar sus programas.

En la Imagen 7.1 se muestra la representación gráfica general del modelo propuesto, en donde se indican las fases del proceso de desarrollo con color rojo, las acciones que se distinguen por ser verbos y se encuentran sobre una línea punteada dirigida, los artefactos por cada una de las fases se encuentran marcados en color melón, los criterios de calidad por cada uno de los artefactos se denotan en color verde, y los principios de diseño y de codificación se señalan en color azul.

Imagen 7.1 Modelo MoACPOO



Fuente: Elaboración propia.

El modelo está conformado por un planteamiento del problema que representa la entrada que da pauta al inicio del proceso de desarrollo. El proceso de desarrollo está constituido por cinco fases secuenciales (análisis, diseño, codificación, compilación y pruebas) que integran prácticas de ingeniería de software, obtenidas de la investigación realizada, para dar solución a un problema. En cada una de las fases se definen las actividades a seguir mediante acciones que se deben realizar para obtener los artefactos del proceso de desarrollo. Y, con el fin de apoyar que el proceso de desarrollo se realice con calidad y con base en la investigación previa de los modelos de calidad, se determinaron los criterios de calidad (comprensión, completitud, consistente, entendible, corrección y exactitud) por cada artefacto y de la entrada, que se consideran relevantes debido a que los criterios de calidad establecen los atributos internos desde la perspectiva del desarrollador. Además, las fases de diseño y codificación tienen como pilar buenas prácticas de ingeniería de software para el desarrollo de software claro, legible, con alta cohesión y bajo acoplamiento. Para entender el modelo general, se explican a continuación las fases del proceso de desarrollo con su gráfico correspondiente.

7.1.1 Planteamiento del problema

El modelo inicia con la exposición de la problemática a resolver, en donde los requisitos se encuentran previamente definidos. Es importante que antes de iniciar con el proceso de desarrollo para dar solución al problema, éste sea entendido completamente, debido a que permitirá identificar los requisitos.

El criterio de calidad que se debe cumplir es la comprensión, que consiste en que el desarrollador ha leído el problema planteado describiendo en qué consiste con base en el dominio del problema. Este criterio es importante debido a que si el problema a resolver no ha sido entendido no se podrá dar solución al mismo o bien la solución será incorrecta por no ser la que corresponda al problema.

En la Tabla 7.1 se muestran las métricas e interpretación del valor medido, así como dos interrogantes binarias a responder después de haber realizado preguntas en el contexto del problema. La primera métrica es la comprensión de conceptos y problemática la cual es importante porque permite conocer si el problema fue entendido; sin embargo, para cerciorarse se determinó una segunda métrica denominada conceptos comprendidos en el

problema. El resultado de ambas métricas se utiliza para obtener el resultado de la métrica porcentaje de comprensión del problema que si es igual a 100% entonces se ha comprendido el problema completamente.

Tabla 7.1. Métricas para el criterio de calidad comprensión

Comprensión	Capacidad de entendimiento del problema que tiene el desarrollador para dar solución al mismo.	
Interpretación del valor medido	$0 \leq CCP \leq 1$	CCP = Comprensión de Conceptos y Problemática
	$0 \leq CC \leq 1$	CC = Conceptos Comprendidos del problema
	$0\% \leq PC \leq 100\%$	PC = Porcentaje de comprensión del problema
	CC y CP entre más cercano a 1 es mejor	
	PC entre más cercano a 100% es mejor	
$CCP = \frac{(CDP+CPP)}{2}$ $CC = \frac{NCC}{TCC}$ $PC = 100\% * (CC * CCP)$		CDP = ¿Comprende los conceptos del dominio del problema? [1,0] CPP = ¿Describe en qué consiste la problemática planteada? [1,0] NCC = Número de conceptos comprendidos TCC = Total de conceptos que se deben comprender

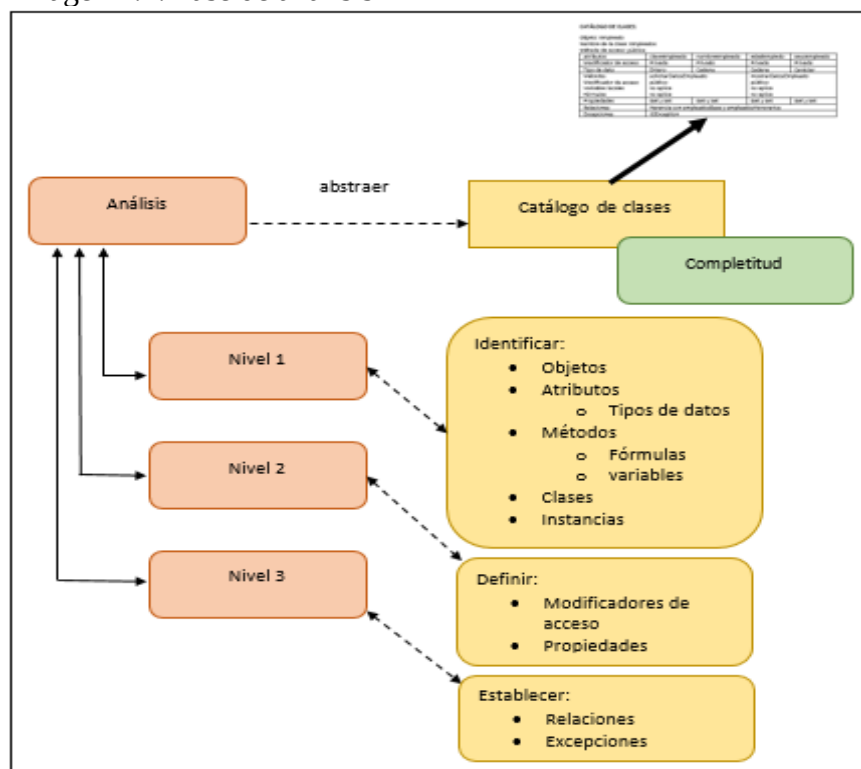
Fuente: Elaboración propia.

7.1.2 Análisis

La primera fase del proceso de desarrollo es el análisis que consiste en describir que debe hacer el programa y los resultados a los que debe llegar, a fin de representar el problema en objetos relacionados con sus respectivos atributos y operaciones que se obtienen mediante la abstracción de datos y funcional de los requisitos predefinidos en el dominio del problema.

La fase de análisis, en color rojo, se divide en tres niveles, ver Imagen 7.2, que se complementan para abstraer la especificación de datos, comportamiento y relaciones de las clases, teniendo como resultado el artefacto catálogo de clases, marcado en color melón. El criterio de calidad asociado a esta fase mediante el resultado de la especificación es completitud indicado en color verde, que es el grado en que la especificación del programa cumple con todos los requisitos predefinidos y la funcionalidad requerida, es decir las clases definidas con sus atributos, métodos y modificadores de acceso, las relaciones entre clases y las excepciones.

Imagen 7.2. Fase de análisis



Fuente: Elaboración propia.

En el nivel 1, después de analizar el problema, se identifican los conceptos que se representan en objetos, se identifican las características de los objetos que son traducidos en atributos o variables de instancias, y como toda variable deben tener asociado un tipo de dato para indicar el dominio de valores permitidos. Posteriormente, se identifican los métodos, que son las operaciones que puede realizar un objeto y modifican su estado haciendo uso de sus atributos, a los cuales se les pueden asociar fórmulas matemáticas para manipular atributos y obtener un resultado. Después, se identifican los objetos comunes para representarlos en una clase que incluirá los atributos y métodos que son compartidos por los objetos; asimismo se identifican las variables locales. Finalmente, se identifican las instancias o creación de objetos a realizar.

En el nivel 2, se define la visibilidad de la clase, de sus atributos y métodos mediante los modificadores de acceso para determinar desde donde se pueden acceder a éstos y así ocultar la información mediante el encapsulamiento, que es un principio de la programación orientada a objetos. Enseguida, se definen las propiedades de los atributos, cuando aplique,

con modificador de acceso privado para realizar procesos relacionados con la obtención y modificación de valores, ocultando su código de implementación, por lo que se le denominan métodos de lectura y de escritura. Por último, se define al menos un método constructor para inicializar los objetos, proporcionando un valor inicial a todos y cada uno de los atributos de la clase. Si es más de un constructor se debe realizar la sobrecarga de este método, *overloading*.

En el nivel 3, se establecen las relaciones entre las clases, las cuales pueden ser dependencia, asociación, composición, agregación y herencia, esta última simple o múltiple, y en caso de ser necesario interfaces. Con la herencia, se pueden redefinir los métodos, *overriding*, es decir proporcionar una nueva implementación del método. Así, con la herencia y la sobrecarga tiene lugar el polimorfismo.

Finalmente, se identifican las excepciones con el propósito de continuar con el flujo normal del programa ante la existencia de sucesos inesperados, visualizando el mensaje correspondiente.

Para el criterio de calidad completitud, se deben considerar los requisitos predefinidos y la identificación de los elementos de los niveles del análisis. En la Tabla 7.2, se muestran las métricas y la interpretación del valor medido de este criterio.

La métrica asociada a este criterio es el porcentaje de completitud que se obtiene de la sumatoria de los resultados de 6 características: 1) Total de clases, son las clases con su método de acceso identificadas adecuadamente. 2) Total de atributos, son los atributos por clase identificados idóneamente y deben acompañarse de su tipo de dato y modificador de acceso. 3) Total de métodos, son los métodos por clase con su modificador de acceso, tipo de dato de retorno y argumentos, identificados correctamente. 4) Total de variables locales, son el número de variables por método con su tipo de dato, identificadas apropiadamente. 5) Total de relaciones, es la identificación satisfactoria de las relaciones entre clases. 6) Total de excepciones, son las excepciones que se identificaron de manera correcta. Un resultado del porcentaje de completitud del 100% significa que la abstracción de datos y funcional fue correcta.

Tabla 7.2. Métricas para el criterio de calidad completitud

Completitud	La especificación del programa considera la abstracción de datos y funcional para dar solución al problema con base en la traducción de los requisitos predefinidos, los cuales deben estar indicados de manera explícita en la fase de análisis.	
Interpretación del valor medido	$0 \leq C_1 \dots C_6 \leq 1$ $0\% \leq PCO \leq 100\%$	C_1 = Total de clases; C_2 = Total de atributos; C_3 = Total de métodos; C_4 = Total de variables locales C_5 = Relaciones entre clases; C_6 = Total de excepciones PCO = Porcentaje de completitud
	C_1 a C_6 entre más cercano a 1 es mejor PCO entre más cercano a 100% es mejor	
$C_1 = \frac{NCi}{NCt}$ $C_2 = \frac{NAi}{NAt}$ $C_3 = \frac{NMi}{NMt}$ $C_4 = \frac{NVi}{NVt}$ $C_5 = \frac{NRi}{NRt}$ $C_6 = \frac{NEi}{NEt}$ $PCO = ((\sum_{i=1}^6 C_i)/6) \times 100\%$	NCi = Número de clases identificadas correctamente NCt = Número de clases totales NAi = Número de atributos identificados correctamente NAt = Número de atributos totales NMi = Número de métodos identificados correctamente NMt = Número de métodos totales NVi = Número de variables identificadas correctamente NVt = Número de variables locales totales NRi = Número de relaciones identificadas correctamente NRt = Número de relaciones totales NEi = Número de excepciones identificadas correctamente NEt = Número de excepciones totales	

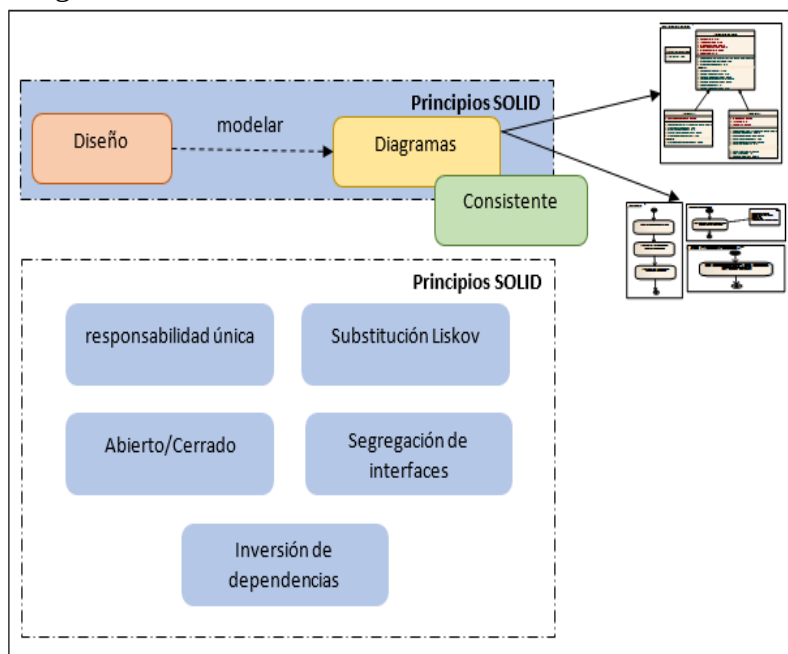
Fuente: Elaboración propia.

7.1.3 Diseño

En la Imagen 7.3 se muestra la fase de diseño que está indicada con color rojo, esta fase es de gran relevancia debido a que en ella se representan de manera gráfica los requisitos predefinidos en el dominio del problema, mismos que fueron abstraídos en la fase de análisis, por lo que el artefacto de esta fase son los diagramas marcados en color melón. Para tener un buen diseño, con alta cohesión y bajo acoplamiento, se consideraron como pilar para el diseño los cinco principios SOLID (Martín, 2003), indicados en color azul. La inclusión de estos principios en el modelo se debe a que al ser independientes del lenguaje de programación se utilizan como una guía para el desarrollo orientado a objetos, contribuyendo en tener un código limpio y correcto, facilitando la fase de codificación al crear un código

legible y fácil de mantener, aunado a tener mejores egresados para el ámbito profesional contribuyendo en hacer mejores desarrolladores de software. El criterio de calidad para esta fase es consistente, indicado en color verde, que es el grado en que un diseño es uniforme, completo y cohesivo. Así, los diagramas de clases y de actividades deberán apegarse a la notación grafica de UML integrando los requisitos predefinidos y deberán apoyarse con los principios SOLID.

Imagen 7.3. Fase de diseño



Fuente: Elaboración propia.

En esta fase, de diseño, se deben modelar los diagramas con el lenguaje de modelado unificado UML que es “un lenguaje de modelado visual que se usa para especificar, visualizar, construir y documentar artefactos de un sistema de software” (Rumbaugh, Jacobson, & Booch, 2000, pág. 3). UML se utiliza desde dos perspectivas: la primera es representar el conjunto de clases y sus relaciones visualizando las mismas en diagrama de clases y la segunda es representar el comportamiento mediante el flujo de actividades de los métodos de las clases mediante el diagrama de actividades. No obstante, se deberán aplicar los principios SOLID.

El criterio de calidad para esta fase es consistente, en la Tabla 7.3 se muestran las métricas e indicadores de este criterio. El diseño se realizará visualizando las dependencias entre clases

para tener un bajo acoplamiento y que el diseño pueda ser traducido de manera sencilla a un lenguaje de programación teniendo una solución funcional del problema planteado, la cual podrá ser mantenible, extensible y reutilizable.

Tabla 7.3. Métricas para el criterio de calidad consistente

Consistente	El programa presenta un diseño uniforme y cohesivo al hacer uso de principios SOLID y guías de estilo en donde se consideran todos los requisitos predefinidos de la fase de análisis, facilitando su traducción a código.	
Interpretación del valor medido	LCOM = 0 TC, valores bajos PCA > 0	LCOM = Falta de cohesión de los métodos ¹ TC = Tamaño de clase PCA = Total de clases abstractas
$LCOM^* = \frac{PROMEDIO(\forall_i M(A_i)) - m }{ 1 - m }$ $TC = NTA + NTM$ $PCA = \left(\frac{NCA}{NTC} \right) \times 100\%$		$M(A_i)$ = Número de métodos que acceden al atributo i m = Número de atributos de la clase NTA = Número total de atributos (incluye herencia) NTM = Número total de métodos (incluye la herencia) NTC = Número total de clases NCA = Número de clases abstractas

Notas: ¹Métrica de Henderson-Sellers (citado en (Piattini Velthuis, García Rubio, García Rodríguez de Guzman, & Pino, 2015))

Fuente: Elaboración propia

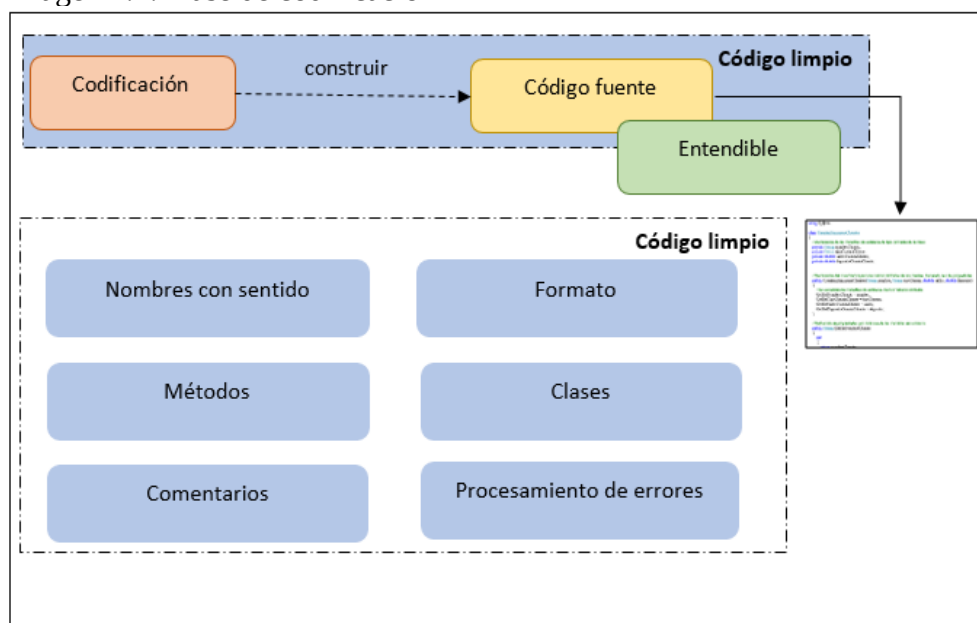
7.1.4 Codificación

La codificación consiste en traducir los diagramas, que consideran los requisitos predefinidos, realizados en la fase de diseño en un lenguaje de programación. El resultado obtenido en esta fase será un código fuente, señalado en color melón, listo para ser compilado y probado, por lo que es necesario conocer la sintaxis del lenguaje de programación a utilizar. Como el código fuente es un artefacto relevante, es importante su calidad, lo cual se logra aplicando código limpio.

El código limpio es un conjunto de principios y buenas prácticas tales como reglas de: nombrado, funciones, formato, indentación y uso de comentarios, entre otros, que se deben aplicar disciplinadamente para obtener un código correcto. Como se mencionó en la sección 2.2, se recomiendan 16 buenas prácticas para crear código limpio y profesional: nombres con sentido, funciones, comentarios, formato, objetos y estructuras de datos, manejo de errores, limites, pruebas de unidad, clases, sistemas, emergencias, concurrencia, refinamiento

sucesivo, internos de Junit, refactorización SerialDate y olores y heurística (Martin R. C., 2009). Para el modelo propuesto en este trabajo doctoral, se consideraron seis indicados en color azul, ver Imagen 7.4, debido a que son con las que se debe iniciar para fomentar el código limpio al aprender programación orientada a objetos.

Imagen 7.4. Fase de codificación



Fuente: Elaboración propia.

Aplicar código limpio es independiente del paradigma y lenguaje de programación, y se tiene como resultado un código legible; es decir, fácil de entender, modificar y reutilizar. Así, el criterio de calidad para esta fase es Entendible, que es la capacidad en que un código puede ser entendido en su estructura y funcionalidad. En la Tabla 7.4 se muestran las métricas y su interpretación de valor medido.

Tabla 7.4. Métricas para el criterio de calidad entendible

Entendible	El programa presenta una estructura, organización y formato homogéneo haciendo legible el código, facilitando su entendimiento y permitiendo conocer la funcionalidad de este.	
Interpretación del valor medido	$0\% \leq PCE \leq 100\%$ $0\% \leq PLOCcom \leq 100\%$ $0 \leq TIND \leq 1$	PLOCcom = Porcentaje de Líneas de código comentadas PCE = Porcentaje de código entendible TIND = Indentación total en el código

	PCE entre más cercano a 100% es mejor PCLO entre más cercano a 0% es mejor TIND entre más cercano a 1 es mejor
$PCE = \left[\left(\frac{NID + NCS + NMV + NBI}{NTI + NTC + NTV + NTB} \right) \right] \times 100\%$ $PLOCcom = \left[\left(\frac{LOCcom}{LOC - Lbco} \right) \right] \times 100\%$ $TIND = \frac{NBI}{NTB}$	NID = Núm. de identificadores descriptivos NTI= Núm. total de identificadores en el código NCS = Núm. de clases nombrados con sustantivo NTC = Núm. total de clases NMV = Núm. de métodos nombrados con verbo NTM = Núm. total de métodos NBI = Núm. de bloques indentados NTB = Núm. total de bloques LOC = Líneas de código LOCcom = líneas de código comentadas Lbco = líneas en blanco

Fuente: Elaboración propia.

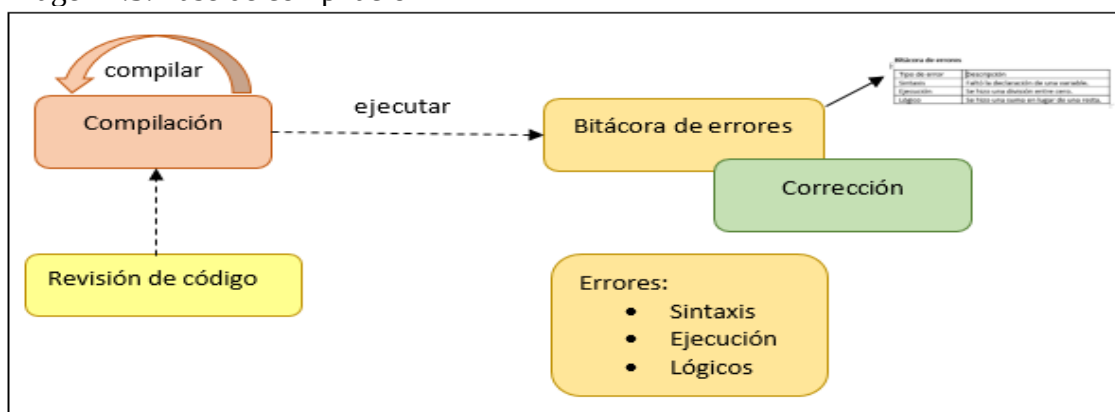
7.1.5 Compilación

El código fuente debe ser compilado para que pueda ser traducido a un lenguaje entendido por la computadora. Durante la fase de compilación se realiza un análisis del código con la finalidad de reconocer que su estructura y escritura sintáctica y semántica sean correctas, y en caso de que no sea así, se detectan los errores cometidos en la fase de codificación que inyectan defectos en el código haciendo que el programa no se pueda ejecutar. De esta manera el compilador es quien se encarga de enviar los mensajes de errores de compilación, que son los errores de sintaxis, como por ejemplo alguna palabra reservada mal escrita. Sin embargo, cuando un programa está libre de estos errores, no se garantiza el funcionamiento correcto del programa ya que existen dos tipos de errores que no son detectados por el compilador los de ejecución y los lógicos.

Los errores de ejecución hacen que la ejecución del programa se detenga debido a que existen operaciones que no pueden realizarse, como por ejemplo cuando se utiliza un índice fuera del rango de un vector, por ello es importante el manejo de excepciones, mientras que los errores lógicos tienen lugar cuando los resultados no son los esperados. Ambos errores se detectan cuando el programa está libre de errores de sintaxis y los lógicos se visualizan en la fase de pruebas.

En la Imagen 7.5 se muestra con color rojo la fase de compilación que tiene implícito el compilador para detectar los errores de sintaxis. Para mitigar los errores de sintaxis se debe hacer una revisión al código, indicado en color amarillo, antes de compilar para corregir en la medida de lo posible este tipo de errores. Una vez compilado el código y generada la lista de errores, se tendrán que corregir los mismos, haciendo la revisión de código para erradicar los errores y volver a compilar; esto se realiza hasta que el código esté libre de errores de sintaxis. Posteriormente se procede con la ejecución del código, en donde podrá tener lugar los errores de ejecución y los lógicos en caso de ejecutarse bien el programa. De esta manera, se deberá llevar un control de los tres tipos de errores, aunque el lógico se detecte en la fase de pruebas, en un artefacto denominado bitácora de errores que se señala en color melón. Por lo anteriormente mencionado, el criterio de calidad que está asociado a esta fase es corrección, señalado en color verde, que es el grado en que un programa se ejecuta de manera correcta y está listo para ser probado, es decir sin errores de sintaxis ni de ejecución. En la Tabla 7.5 se muestran las métricas asociadas a este criterio de calidad.

Imagen 7.5. Fase de compilación



Fuente: Elaboración propia.

Tabla 7.5. Métricas para el criterio de calidad corrección

Corrección	Grado en que un programa se ejecuta de manera correcta y está listo para ser probado, es decir sin defectos de sintaxis y de ejecución.	
Interpretación del valor medido	$0 \leq DES \leq \infty$ $0 \leq TEsC \leq \infty$ $0 \leq DEE \leq \infty$ $0 \leq TEeC \leq \infty$	DES = Densidad de errores de sintaxis TEsC = Total de errores de sintaxis corregidos DEE = Densidad de errores de ejecución TEeC = Total de errores de ejecución corregidos

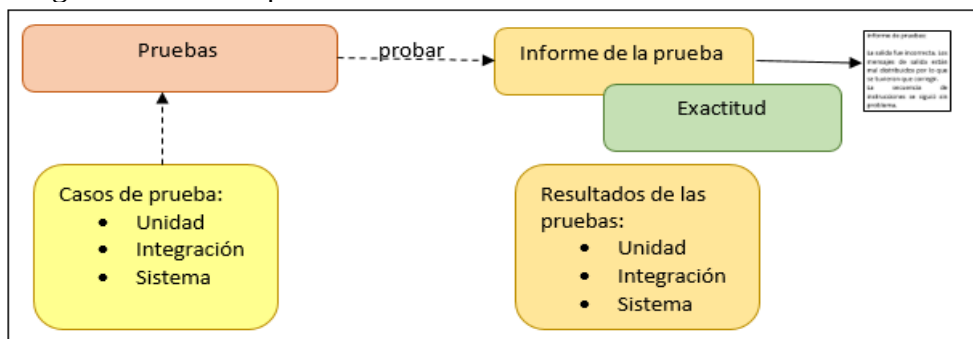
DES, DEE, TEeC y TEeC entre más cercano a 0 es mejor	
$DES = 1000 \left(\frac{TES}{KLOC} \right)$ $TEsC = TES - CEsC$ $DEE = 1000 \left(\frac{TEE}{KLOC} \right)$ $TEeC = TEE - CEeC$	TES = Total de errores de sintaxis CEsC = Total de errores de sintaxis corregidos TEE = Total de errores de ejecución CEeC = Total de errores de ejecución corregidos

Fuente: Elaboración propia.

7.1.6 Pruebas

En la Imagen 7.6 se muestra la última fase pruebas, coloreada en rojo, que tiene lugar al ejecutar el programa y se espera que los resultados sean correctos. Sin embargo, para corroborar que así sea, se debe realizar la verificación de los resultados mediante tres pruebas: de unidad, de integración y del sistema, que si bien es cierto se utilizan para probar software robusto en empresas, también pueden ser utilizadas y adaptadas en el ámbito académico. Los resultados de las pruebas se deberán documentar en el artefacto informe de pruebas, señaladas en color melón. El criterio de calidad de esta fase es exactitud que es grado en que un programa después de ser ejecutado proporciona los resultados correctos

Imagen 7.6. Fase de pruebas



Fuente: Elaboración propia.

Para la realización de las pruebas se utiliza los casos de prueba que se planifican con antelación, y debe contener como mínimo la siguiente información (Serna, 2013): un conjunto de datos de entrada, condiciones de ejecución de la prueba y la salida esperada. Para el caso del modelo, los casos de prueba estarán definidos desde el inicio al igual que los requerimientos, por lo que estarán incluidos con el planteamiento del problema. Los casos de

prueba incluirán los datos con los que se probará el programa, las acciones a realizar y los resultados a los que se deben llegar, a fin de probar los programas y evaluar si los resultados reales son los que ya están predefinidos. En caso de que los resultados no sean los esperados, se debe llevar a cabo la corrección del código hasta obtener lo que se solicitó.

El incluir los casos de pruebas en el modelo tiene la finalidad de garantizar que el programa esta correcto al verificar las salidas que emite el programa a fin de que el programador se sienta satisfecho por el desarrollo realizado, además de concientizar que la programación conlleva un proceso bien definido al que se le aplican buenas prácticas de ingeniería de software hasta en la última fase del proceso.

El criterio de calidad exactitud tiene la finalidad de conocer si los resultados reales coinciden con los resultados esperados, verificados por la especificación de requerimientos. Las métricas que corresponden al criterio de calidad exactitud se encuentran en la Tabla 7.6.

Tabla 7.6. Métricas para el criterio de calidad exactitud

Exactitud	Grado en que el programa realiza los cálculos y el procedimiento correcto de forma tal que en la fase de pruebas los resultados reales corresponden con la salida esperada.	
Interpretación del valor medido	$0\% \leq PCPex \leq 100\%$	PCPex = Porcentaje de casos de prueba exitosos
	$0 \leq DDL \leq \infty$	DDL = Densidad de errores lógicos
	PCPex entre más cercano a 100% es mejor DDL entre más cercano a 0 es mejor	
$PCPex = \frac{CPE}{TCP}$ $DDL = 1000 \times \frac{TDL}{KLOC}$		CPE = Casos de prueba exitosos TCP = Total de casos de prueba TDL = Total de errores lógicos

Fuente: Elaboración propia.

Conclusiones

El modelo MoACPOO es la contribución principal del presente trabajo doctoral, el cual se propone para la mejora de los programas en los cursos de programación orientada a objetos. Este modelo fue el resultado de la investigación realizada sobre las mejores prácticas de ingeniería de software y criterios de calidad considerando la respuesta de 84 docentes de 38 universidades públicas en México, así como de la revisión de la literatura.

El modelo se caracteriza por incluir un ciclo de vida que consta de cinco fases que integran un conjunto de buenas prácticas de ingeniería de software considerando la naturaleza de la fase, muestra de ello son los principios SOLID (Martín, 2003) en la fase de diseño y los principios de código limpio (Martin R. C., 2009) que tienen lugar en la codificación.

El modelo no solo integra un conjunto de buenas prácticas sino también incluye la acción que se debe realizar en cada una de las fases con palabras en forma de verbo que son sencillas de entender, éstas están dirigidas a las actividades que se deben realizar por cada una de las fases y que permite obtener como resultado los artefactos incluidos en las fases. Para ilustrar mejor, la fase de análisis se asocia con la palabra abstraer, debido que se debe realizar la abstracción funcional y de datos para identificar los objetos, las clases y las relaciones por mencionar algunas, las cuales se deben incluir en un catálogo de elementos. Así, cada una de las fases tiene sus particularidades.

Se debe agregar que cada fase incorpora un criterio de calidad que permite medir que las actividades que se realizaron en el proceso fueron correctas. Para ello se generaron las métricas de calidad por cada uno de los criterios.

En conclusión, el modelo propuesto es un modelo completo que permitirá seguir un proceso de desarrollo para la creación de programas en los cursos de programación orientada a objetos, cuidando la calidad y contribuyendo en la aplicación de buenas prácticas de ingeniería de software, que si bien es cierto no se pueden comparar a las utilizadas en la industria del software, si se hace uso de las necesarias, básicas y comunes que todo futuro desarrollador debe conocer. En cuanto a los criterios de calidad, sirven de base para corroborar la calidad del programa realizado, que, si bien puede resultar un poco tedioso su aplicación, se debe crear conciencia de la necesidad de hacer uso de estos para tener como resultado programas con calidad.

Capítulo 8. Resultados de la aplicación del modelo

Introducción.

Los cursos de programación orientada a objetos (POO) como todo curso de programación requieren de la comprensión de los conceptos y de la aplicación práctica de los mismos, lo cual se les dificulta a los estudiantes. Por ello, se propuso el modelo MoACPPPO, presentado en el capítulo 4, que incorpora prácticas de ingeniería de software y criterios de calidad (CC) que tienen asociadas métricas de calidad resultados de la investigación realizada en esta tesis, ver Capítulo 3, y que sustentan el modelo propuesto.

La contribución del modelo es mejorar el proceso de desarrollo de programas de los estudiantes en los cursos de programación orientada a objetos (POO). Sin embargo, es importante aplicar el modelo para obtener resultados que permitan responde a la pregunta de investigación de este capítulo **¿Cómo influye un modelo con buenas prácticas de ingeniería de software y criterios de calidad en los cursos de programación orientada a objetos?**

La respuesta a esta pregunta se sustenta en la aplicación del modelo MoACPOO en un curso de POO en el periodo 2019 con dos grupos de la Licenciatura en Informática Administrativa (LIA) del Centro Universitario UAEM Texcoco (CUUT) de la Universidad Autónoma del Estado de México (UAEMex), uno de los grupos fue el grupo de control y el otro fue el grupo experimental y de esta manera cumplir con el objetivo de este capítulo **evaluar el modelo en un curso de programación orientada a objetos determinando como influye en los estudiantes.**

Para evaluar el modelo MoACPOO se realizó un análisis con estadística descriptiva de las puntuaciones alcanzadas en los proyectos desarrollados por los estudiantes, de las calificaciones obtenidas por los estudiantes para acreditar el curso y de las puntuaciones obtenidas por los estudiantes en la posprueba. Sin embargo, sólo los proyectos y la posprueba se pueden considerar como contundentes para realizar una comparación dado que en las calificaciones obtenidas en el curso intervienen otros elementos tales como tareas, actividades, y prácticas entre otros.

Para aceptar o rechazar la hipótesis nula, **Si se utiliza un modelo que incluya un conjunto de buenas prácticas y criterios de calidad no se mejorará el desarrollo de programas en**

los cursos de programación orientada a objetos, se aplicó la prueba de t student para la comparación de medias entre muestras independientes con un nivel de confianza de 0.05%, con la finalidad de conocer si existen diferencias significativas entre las puntuaciones obtenidas en la posprueba por los estudiantes de los grupos de control y experimental. A continuación, se presenta el contexto de la investigación y los resultados.

8.1 Contexto de la investigación para la aplicación del modelo

Como resultado de la investigación realizada en este trabajo doctoral, ver capítulos 5 y 6, se propone el modelo MoACPOO, ver capítulo 7, que incorpora prácticas de ingeniería de software y criterios de calidad (CC) distribuidas en 5 fases.

La contribución del modelo es mejorar el proceso de desarrollo de programas de los estudiantes en los cursos de programación orientada a objetos (POO) considerando la comprensión de los conceptos y que da lugar de la hipótesis planteada. Así, para validar el modelo y por ende la hipótesis, el modelo MoACPOO se aplicó en un ambiente real con un enfoque cuantitativo y un diseño cuasiexperimental debido a que “manipula deliberadamente, al menos, una variable independiente para observar su efecto y relación con una o más variables dependientes. Los sujetos no se asignan al azar a los grupos ni se emparejan, sino que dichos grupos ya están formados antes del experimento” (Hernández Sampieri, et al., 2006, pág. 203).

Partiendo de lo anterior, se utilizó un muestreo no probabilístico en virtud de que la muestra fue por conveniencia debido a que en el periodo 2019 en la Licenciatura en Informática Administrativa (LIA) del Centro Universitario UAEM Texcoco (CUUT) de la Universidad Autónoma del Estado de México (UAEMex) se imparte el curso de POO, actualmente denominada Programación Imperativa (PI) por el cambio en el plan de estudios, aunado al acceso que se tiene en los grupos por encontrarse en la universidad donde labora quien escribe este trabajo.

La población de estudiantes en el curso de PI fue de 55 distribuidos en dos grupos, matutino y vespertino, con base en la realización de inscripción de los estudiantes en dicho curso.

En la Tabla 8.1 se pueden observar las características de los grupos A y B que son los grupos que cursaron PI y que se consideraron para realizar la validación experimental del modelo

propuesto con base en el análisis de los resultados obtenidos mediante técnicas estadísticas. El grupo A es el grupo de control y el grupo B es el grupo experimental.

Tabla 8.1 Grupos experimental y de control

Grupo	Turno	Estudiantes	Mujeres	Hombres	Grupo
A	Matutino	29	12	17	Control
B	Vespertino	26	8	18	Experimental

Fuente: Elaboración propia.

8.2 Planificación para la aplicación del modelo

El curso de PI se compone de 4 unidades que se imparten con una duración de 6 horas a la semana durante 16 semanas, no obstante, los temas de POO se abordan en la última unidad a la que se les destinaron 5 semanas con un total de 30 horas. Sin embargo, con la finalidad de aplicar el modelo se realizó una departamentalización del curso PI por los docentes de los grupos A y B en cuanto a la planeación de los contenidos del curso, tiempo destinado en los temas, exámenes y proyecto. Por cada una las unidades se proporcionaron los conceptos de manera teórica-conceptual a los estudiantes para posteriormente aplicarlos mediante prácticas de laboratorio en donde se dosificaron para las tres primeras unidades el uso de buenas prácticas más no así la aplicación del modelo debido a éste se aplicó en la última unidad y sólo para el grupo B.

En el grupo B se utilizaron dos ejemplos pilotos. El primer ejemplo, ver Anexo 2, fue utilizado para guiar a los estudiantes en la aplicación del modelo MoACPOO durante los temas de la unidad con la finalidad de concientizar a los estudiantes que el uso de buenas prácticas y los CC aplicados en el proceso de desarrollo de programas mejora la calidad al crear sus programas. En el segundo ejemplo, ver Anexo 2, se les dejó como proyecto a los estudiantes en el que aplicaron el modelo en la solución al problema planteado.

Es importante mencionar, que el cálculo de las métricas fue realizado por el docente debido a que era necesario asegurar que los resultados no fueran manipulados.

Los resultados presentados a continuación se dan en tres sentidos. El primero es el puntaje obtenido por los estudiantes en el proyecto que formó parte de la calificación y se ponderó a 2.0 utilizando una escala de satisfacción para medir la puntuación obtenida, ver Tabla 8.2, el grupo B aplicó el modelo MoACPOO. El segundo son las calificaciones obtenidas de los estudiantes en el curso de PI, en una escala del 0.0 al 10.0. El tercero son los resultados

obtenidos por los estudiantes de los grupos A y B en la posprueba, que no formaron parte de su calificación.

Tabla 8.2. Escala de satisfacción para los proyectos

Puntaje	Nivel de satisfacción
1.6 – 2.0	Excelente
1.2 – 1.5	Satisfactorio
0.8 – 1.1	Suficiente
0.4 – 0.7	Insuficiente
0.0 - .0.3	Deficiente

Fuente: Elaboración propia.

8.3 Resultados de la aplicación del modelo

Como se mencionó anteriormente, el primer ejemplo sirvió para que los estudiantes del grupo B aplicarían el modelo MoACPOO guiados por el docente. Las fases de análisis, codificación y pruebas no se les dificultaron a los estudiantes, dado que durante el curso se estuvieron dosificando las buenas prácticas del modelo MoACPOO durante las unidades previas a la última unidad, aunque fue guiado. Sin embargo, la fase de diseño se les dificulta porque además de aprender la nomenclatura para los diagramas tuvieron que comprender los principios SOLID, ver sección 2.2.

Con respecto a la compilación, a los estudiantes se les dificulta entender los errores que detecta el compilador dado que se encuentran en inglés aunado a que el llevar el registro de los errores lo encuentran tedioso y les causa distracción, por lo que el registro de los errores en las bitácoras ya no fue considerado.

Posteriormente, al primer ejemplo piloto, los estudiantes aplicaron el modelo MoACPOO en la realización del proyecto, en equipo, en el curso de PI considerando el ejemplo piloto 2, ver Anexo 2. Con la finalidad de conocer si los estudiantes habían comprendido el problema y conceptos se tuvo una sesión de dudas, posteriormente cada uno realizó su proyecto. Sin embargo, para corroborarlo se solicitó a los estudiantes escribieran en una hoja cuál era el problema al que darían solución, así como los conceptos que se encontraban en el planteamiento del problema, la cual se utilizó una vez que los estudiantes entregaron el proyecto para obtener los resultados de las métricas por cada una de las fases del CV del modelo MoACPOO. En la Tabla 8.3, se encuentran los resultados de las métricas.

Tabla 8.3. Resultados de las métricas del proyecto, por estudiante

Alumno	Problema Comprensión	Análisis Compleitud	Diseño Consistente		Codificación Entendible			Pruebas Exactitud	
	PC	PCO	LCOM	PCA	PLOCom	PCE	TIND	PCPex	DDL
1	83	83	4	0	0	40	1	0	NT
2	83	83	4	0	0	40	1	0	NT
3	1	83	1	0	50	100	1	100	0
4	50	33	4	0	0	50	1	0	166.66
5	1	100	0	0	40	100	1	100	0
6	1	83	1	0	50	100	1	100	0
7	1	83	1	0	0	41	1	0	NT
8	1	79	2	0	50	100	1	100	0
9	1	79	2	0	50	100	1	100	0
10	83	79	2	0	0	100	1	100	0
11	50	33	4	0	0	50	1	0	166.66
12	1	83	1	0	0	41	1	0	NT
13	50	33	4	0	0	50	1	0	166.66
14	1	83	1	0	0	41	1	0	NT
15	1	79	2	0	50	100	1	100	0
16	1	100	0	0	40	100	1	100	0
17	1	100	0	0	10	100	1	100	0
18	50	33	4	0	0	50	1	0	166.66
19	83	83	4	0	0	40	1	0	NT
20	1	83	1	0	50	100	1	100	0
21	1	100	0	0	10	100	1	100	0
22	1	100	0	0	10	100	1	100	0
23	1	100	0	0	40	100	1	100	0
24	1	100	0	0	10	100	1	100	0
25	83	83	5	0	0	100	1	100	0
26	83	83	5	0	0	100	1	100	0

Notas: PC (Porcentaje de comprensión del problema =1), PCO (Porcentaje de completitud = 100), LCOM (Falta de cohesión de métodos = 0), PCA (Porcentaje de clases abstractas > 0), PLOCom (Porcentaje de Líneas de código comentadas 0% <= PLOCom <=100%), PCE (Porcentaje de código entendible, 0% <= PCE <=100%), TIND (Indentación total en el código = 1), PCPex (Porcentaje de casos de prueba exitosos, 0% <= PCPex <=100%), DDL(Densidad de errores lógicos, 0 <= DDL <=∞), NT (no terminaron).

Fuente: Elaboración propia.

Como se puede observar en la Tabla 8.3, 10 estudiantes que corresponde al 38.46% tuvieron dificultad en la comprensión del problema, había confusión en cómo se calcula el sueldo del empleado (profesores, administrativas o ambos) y al concepto de guardias.

Con respecto a la fase de análisis, sólo 7 estudiantes que corresponde al 26.92% obtuvieron 100 en el porcentaje de completitud que indica que realizaron la abstracción de datos y funcional de manera correcta. Sin embargo, el 73.08% de los estudiantes no consideraron las

excepciones que fue un factor común y otros además identificaron un menor número de clases y de métodos, así como duplicaron atributos haciendo uso de sinónimos.

En relación con la fase de diseño, considerando los resultados en LCOM, 19 estudiantes que representa el 73.08% presentaron falta de cohesión en las clases ya que utilizaron atributos no comunes en diferentes métodos, lo que indica que tres de los principios SOLID (responsabilidad única, abierto/cerrado y substitución Liskov) ver sección 2.3 son los que presentan mayor dificultad. El valor deseable de LCOM es cero, 7 estudiantes que representa el 26.92% lo obtuvieron y 6 estudiantes que representa el 23.08% obtuvieron un resultado de 1. Asimismo, en el porcentaje de clases abstracta el 100% de los estudiantes obtuvieron 25 lo que es adecuado debido a que sólo existe una clase abstracta.

En cuanto a la fase de codificación, 13 estudiantes que corresponde al 50% presentaron cero comentarios en su código, se infiere que se les olvidó incluirlos. El 100% de los estudiantes indentaron su código y 16 estudiantes que representan el 61.54 entregaron un código entendible en su totalidad.

Por lo que concierne a la fase de pruebas, solo se aplicó la prueba del sistema para verificar el funcionamiento del programa, se asume que los estudiantes aplicaron las pruebas de unidad y de integración, sin embargo, no se solicitaron las evidencias de estas. Así, los resultados que se presentan son considerando un caso de prueba para verificar los resultados. En este sentido, la ejecución de los programas de 16 estudiantes que representan el 69.57% fue exitosa, es decir presentaron los resultados esperados y no tuvieron errores lógicos. Pero 6 estudiantes que representa el 23.08% no terminaron el proyecto y el 15.38% que corresponde a 4 estudiantes no tuvieron los resultados correctos y tuvieron errores de lógica.

Por otro lado, el proyecto tuvo una ponderación del 20% de la calificación de cada estudiante, el puntaje obtenido en el proyecto por cada estudiante se aprecia en la Tabla 8.4, en donde se muestra que 15 estudiantes, que representa el 57.69%, lograron un puntaje satisfactorio y excelente en el proyecto, 5 estudiantes que representan el 19.23% alcanzaron un nivel de satisfacción suficiente y, por el contrario 6 estudiantes que representan el 23.08%, obtuvieron un puntaje por debajo de .6, que es considerado como insuficiente y deficiente.

Conviene subrayar que en la realización del proyecto tres estudiantes lo consideraron como complejo, teniendo dificultad en la realización de este, en aplicar los principios SOLID y en codificar la solución, por lo que su desempeño no fue el esperado. Cabe señalar que de las

buenas prácticas que incorpora el modelo MoACPOO, los principios SOLID son los que se les dificultan a los estudiantes.

Tabla 8.4. Calificaciones obtenidas del proyecto del grupo B

Alumno	Puntaje Proyecto	Alumno	Puntaje Proyecto
1	0.3	15	1.3
2	0.3	16	1.8
3	1.7	17	1.8
4	1.0	18	1.0
5	1.8	19	0.3
6	1.7	20	1.7
7	0.5	21	1.8
8	1.3	22	1.8
9	1.3	23	1.8
10	1.1	24	1.8
11	1.0	25	1.5
12	0.5	26	1.5
13	1.0	Promedio	1.235
14	0.5		

Puntuación	Frecuencia	Puntuación	Frecuencia
0.3	3	1.3	3
0.5	3	1.5	2
1.0	4	1.7	3
1.1	1	1.8	7

Nota: La calificación máxima a obtener es de 2.0

Fuente: Elaboración propia.

La Gráfica 8.1, ilustra mejor los puntajes obtenidos por los estudiantes del grupo B, grupo experimental, en donde se reitera que 6 estudiantes que representan el 23.08% tienen una puntuación por debajo de .8 que son los estudiantes que tienen un nivel de satisfacción insuficiente y deficiente. Ante esta situación se tuvo una entrevista no formal con estos estudiantes para conocer la razón del puntaje obtenido indicando tres de ellos que habían dado prioridad a otro proyecto de otro curso que debían entregar, pero no se cuenta con evidencia de ello. Tres estudiantes, indicaron que el diseño y la codificación se les dificulta. Sin embargo, 10 estudiantes que representa el 38.46% obtuvieron un nivel de satisfacción excelente, 5 estudiantes que representa el 19.23% lograron un puntaje satisfactorio y 5 más consiguieron un nivel de satisfacción suficiente. Por consiguiente, el 76.92% de los estudiantes realizaron proyectos con un nivel de satisfacción entre suficiente y excelente.

Gráfica 8.1. Puntajes obtenidos en el proyecto, grupo experimental



Fuente: Elaboración propia.

Los estudiantes del grupo A, grupo de control, realizaron su proyecto sin hacer uso del modelo MoACPOO. Al igual que en el grupo B, el proyecto tuvo una ponderación del 20% de la calificación, el puntaje obtenido por cada estudiante se aprecia en la Tabla 8.5.

Tabla 8.5. Calificaciones obtenidas del proyecto del grupo A

Alumno	Puntaje Proyecto	Alumno	Puntaje Proyecto
1	1.6	16	1.0
2	1.0	17	1.5
3	1.0	18	1.0
4	1.0	19	1.1
5	1.3	20	1.4
6	1.0	21	1.3
7	1.1	22	1.3
8	1.3	23	1.0
9	1.1	24	1.3
10	1.4	25	1.5
11	1.0	26	1.0
12	1.6	27	0.0
13	1.0	28	1.1
14	1.2	29	1.0
15	1.3	Promedio	1.152

Puntuación	Frecuencia	Puntuación	Frecuencia
0.0	1	1.3	6
1.0	11	1.4	2
1.1	4	1.5	2
1.2	1	1.6	2

Nota: La calificación máxima a obtener es de 2.0

Fuente: Elaboración propia.

De acuerdo con la Tabla 8.5, 28 estudiantes que representan el 96.55% obtuvieron un puntaje entre 1.0 y 1.6 en sus proyectos que se encuentra en un nivel de satisfacción entre suficiente y excelente y sólo un estudiante obtuvo deficiente. Sin embargo, los resultados muestran que sólo 2 estudiantes que representan el 6.90% lograron un nivel de satisfacción excelente. En la Gráfica 8.2, se observa que 11 estudiantes que representan el 37.93% obtuvieron en su proyecto un nivel de satisfactorio, 15 estudiantes que representan el 51.72% tienen un nivel de suficiente y 1 que representa el 3.45 presenta un nivel de deficiente.

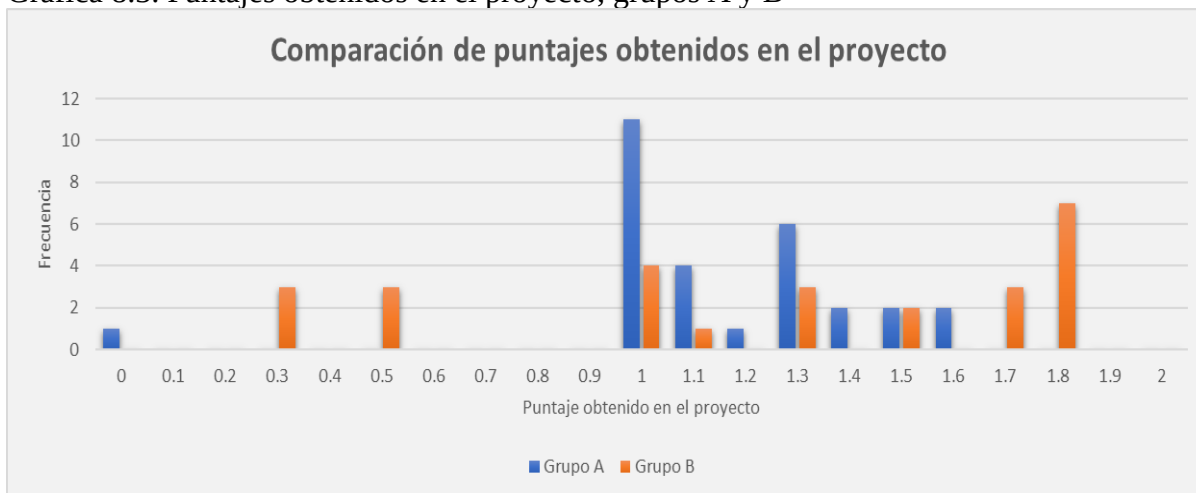
Gráfica 8.2. Puntajes obtenidos en el proyecto, grupo de control



Fuente: Elaboración propia.

Por lo que se refiere a la media obtenida con base en los resultados de las puntuaciones de los proyectos de los estudiantes de los grupos A y B en el curso de PI, ver Tablas 8.4 y 8.5, se concluye que no existe diferencia significativa entre la realización del proyecto aplicando el modelo MoACPOO y sin utilizarlo. No obstante, es necesario realizar la comparación de los puntajes obtenidos en los proyectos en cada uno de los grupos, debido a que es evidente que al aplicar el modelo MoACPOO los estudiantes del grupo B lograron obtener un mejor nivel de satisfacción que los estudiantes del grupo A, ver Gráfica 8.3.

Gráfica 8.3. Puntajes obtenidos en el proyecto, grupos A y B



Fuente: Elaboración propia.

Como se puede apreciar en la Gráfica 8.3, a pesar de que 6 estudiantes del grupo B tuvieron un nivel de satisfacción insuficiente y deficiente, el 38.46% de los estudiantes del grupo B obtuvieron un nivel de satisfacción excelente contra el 6.90% que lograron los estudiantes del grupo A. Asimismo, el 19.23% de los estudiantes del grupo B consiguieron un nivel satisfactorio en contraste con un 37.93% de los estudiantes del grupo A. Cabe señalar que el 19.23% de estudiantes del grupo B lograron un puntaje satisfactorio con una diferencia de 32.49 puntos porcentuales con respecto a los estudiantes del grupo A donde el 51.72% de los estudiantes alcanzó un nivel de satisfacción suficiente en sus proyectos.

Con respecto a lo anterior, se deduce que la aplicación del modelo MoACPOO contribuye en la mejora de la calidad en el desarrollo de los programas en el curso de PI, anteriormente denominado programación orientada a objetos.

8.4 Calificaciones de estudiantes

Un parámetro importante que considerar en el curso de PI del cual fue objeto de estudio el modelo MoACPOO aplicado por los estudiantes del grupo B, son las calificaciones obtenidas en ambos grupos. Las calificaciones que se muestran en la Tabla 8.6 están dadas en una escala de 0.0 a 10.0, que son las obtenidas por cada estudiante, siendo la mínima aprobatoria 6.0. Como se puede observar, el promedio de calificaciones obtenido en ambos grupos es similar, por lo que no existen diferencias entre los grupos A y B, incluso en ambos grupos dos estudiantes no acreditaron el curso. Empero, el promedio de calificaciones en ambos

grupos es muy bajo de 6.7 y 6.8, por lo que es conveniente representar la frecuencia de las calificaciones en una gráfica para compararlas y observar las diferencias de estas en ambos grupos.

Tabla 8.6. Calificaciones de los estudiantes en el curso de PI

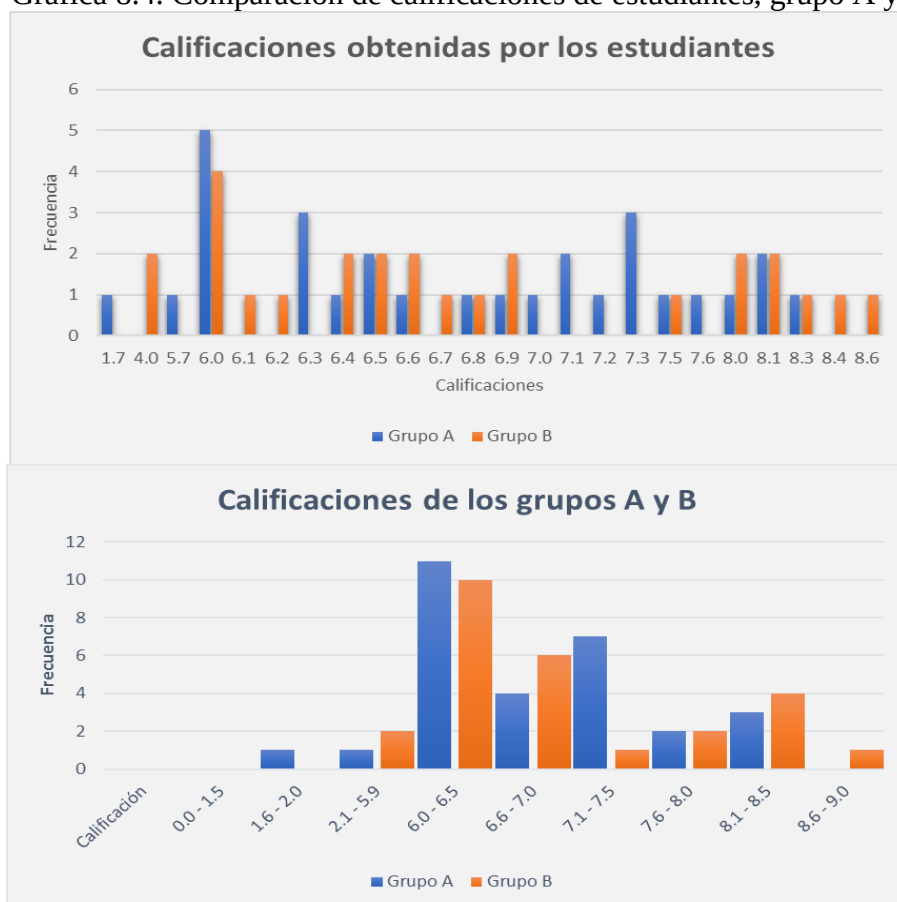
Estudiante	Calificaciones	
	Grupo A (control)	Grupo B (experimental)
1	8.3	6.4
2	6.4	6.7
3	6.0	7.5
4	6.0	4.0
5	7.0	6.6
6	5.7	6.5
7	6.8	6.9
8	7.1	8.0
9	6.6	6.0
10	7.3	6.0
11	6.3	6.5
12	8.1	6.6
13	6.5	4.0
14	6.9	6.8
15	7.5	6.1
16	6.3	8.1
17	8.1	8.1
18	6.3	8.4
19	6.5	6.0
20	7.6	6.9
21	7.3	6.4
22	7.3	6.2
23	6.0	8.0
24	7.2	8.3
25	8.0	8.6
26	6.0	6.0
27	1.7	
28	7.1	
29	6.0	
Promedio	6.686	6.754

Fuente: Elaboración propia.

En la Gráfica 8.4, se observa que en ambos grupos el mayor porcentaje de estudiantes obtienen calificaciones que oscilan 6.5 y 7.5, el porcentaje de estudiantes en el grupo A es del 75.86% y en el grupo B es del 58.62% y dos estudiantes en cada uno de los grupos no acreditaron el curso. Además, no existen diferencias significativas entre las medias de las calificaciones obtenidas en el curso de PI en ambos grupos, ver lo que resulta del análisis realizado con la prueba de T student para muestras independientes, ver Imagen 8.1, lo que aparentemente indica que la comprensión de los conceptos, los conocimientos que se tienen

de POO y su aplicación práctica son similares. Sin embargo, se debe considerar que la calificación obtenida incluye además del proyecto, la evaluación continua de diversas actividades que cada docente incluyó en su curso. Es posible que las calificaciones obtenidas por los estudiantes del grupo B se deba a que en las primeras actividades que se realizaron se les dificultó el uso de buenas prácticas que se incluyen en el modelo MoACPOO. Por consiguiente, se aplicó un posttest a los estudiantes de ambos grupos para conocer la comprensión de los conceptos y la aplicación de estos en un pequeño caso práctico.

Gráfica 8.4. Comparación de calificaciones de estudiantes, grupo A y B



Fuente: Elaboración propia.

Por otro lado, y para comparar las medias de los dos grupos se llevó a cabo la prueba de t student para muestras independientes con un nivel de confianza del 0.05%, ver Imagen 8.1, teniendo como resultado 6.686 en la media para el grupo A y 6.754 para el grupo B y como el nivel de significación bilateral fue de .834 para ambos grupos que es mayor a .05 para la

prueba de Levene, se concluye que no existen diferencias entre las medias de los grupos. Además, de que no se puede asegurar que exista diferencia de las medias de $-.0676$, debido a que entre los intervalos de confianza existe 0, lo que podría indicar que puede o no haber diferencias.

Imagen 8.1. Prueba t student para las calificaciones

Estadísticos de grupo				
Grupo	N	Media	Desviación tip.	Error tip. de la media
Calificación A	29	6.686	1.2029	.2234
B	26	6.754	1.1738	.2302

Prueba de muestras independientes									
		Prueba de Levene para la igualdad de varianzas		Prueba T para la igualdad de medias					
		F	Sig.	t	gl	Sig. (bilateral)	Diferencia de medias	Error tip. de la diferencia	95% Intervalo de confianza para la diferencia
Calificación	Se han asumido varianzas iguales	.156	.694	-.211	53	.834	-.0676	.3212	-.7119 .5766
	No se han asumido varianzas iguales			-.211	52.603	.834	-.0676	.3208	-.7111 .5758

Fuente: Elaboración propia.

8.5 Resultados de la posprueba

Al finalizar el curso de PI se llevó a cabo una posprueba en los grupos de control y experimental, A y B respectivamente, con la finalidad de corroborar que no existan diferencias significativas considerando las puntuaciones de los proyectos y las calificaciones obtenidas por los estudiantes en el curso de PI. La posprueba tuvo una duración de dos horas y fue respondida de manera individual.

La posprueba está dividida en tres secciones, ver Anexo 3. La primera sección está orientada a la comprensión de los conceptos de programación orientada a objetos a partir de preguntas abiertas con un valor de 4.8. La segunda sección se enfoca a la comprensión de conceptos a partir de incisos de relación con un valor de 1. La tercera sección incluye a la aplicación práctica de los conceptos con un valor de 4.2. Hay que destacar, que la puntuación obtenida por cada uno de los estudiantes no fue considerada como parte de su calificación para acreditar el curso, solo se utilizó para comprobar la comprensión de los conceptos y su aplicación. A continuación, se muestran las puntuaciones obtenidas por los estudiantes en cada una de las secciones de la posprueba, Tabla 8.7.

Tabla 8.7. Puntuaciones por sección y total de la posprueba

Estudiante	Resultados de la posprueba Grupo de control (A)				Resultados de la posprueba Grupo experimental (B)			
	Sección 1	Sección 2	Sección 3	Total	Sección 1	Sección 2	Sección 3	Total
1	0.8	0.7	0.0	1.5	4.4	0.5	1.0	5.9
2	2.0	0.7	0.0	2.7	4.4	0.5	2.0	6.9
3	2.2	0.4	0.0	2.6	3.4	0.8	3.7	7.9
4	2.2	0.0	0.0	2.2	4.4	0.5	0.0	4.9
5	0.8	0.4	0.0	1.2	3.7	0.8	1.5	6.0
6	0.0	0.0	0.0	0.0	4.0	0.7	1.3	6.0
7	2.0	0.5	0.0	2.5	3.9	0.6	3.0	7.5
8	2.9	0.4	0.0	3.3	4.1	0.8	3.8	8.7
9	3.0	0.3	1.7	5.0	3.1	0.8	2.0	5.9
10	1.2	0.4	0.0	1.6	2.7	0.7	2.6	6.0
11	2.2	0.8	0.5	3.5	3.6	0.3	3.1	7.0
12	0.8	0.7	0.0	1.5	3.6	0.5	3.2	7.3
13	2.2	0.7	1.0	3.9	3.0	0.5	0.0	3.5
14	1.4	0.4	0.0	1.8	3.0	1.0	3.7	7.7
15	1.2	0.4	0.0	1.6	4.0	0.5	1.2	5.7
16	2.4	0.4	0.0	2.8	4.1	0.9	3.7	8.7
17	2.2	0.5	0.8	3.5	3.7	0.8	3.4	7.9
18	1.6	0.5	0.0	2.1	4.1	0.9	4.0	9.0
19	1.0	0.2	0.0	1.2	4.0	1.0	2.0	7.0
20	2.7	0.2	0.0	2.9	3.2	0.6	2.5	6.3
21	1.2	0.1	0.0	1.3	3.9	0.7	1.5	6.1
22	2.0	0.5	0.0	2.5	3.6	0.6	2.2	6.4
23	1.0	0.5	0.0	1.5	4.1	0.7	2.9	7.7
24	2.1	0.6	0.0	2.7	4.0	0.8	3.2	8.0
25	2.2	0.5	0.0	2.7	4.5	0.9	3.7	9.1
26	1.7	0.0	0.0	1.7	2.8	0.7	2.0	5.5
27	1.0	0.2	0.0	1.2				
28	2.0	0.3	0.0	2.3				
29	0.7	0.3	0.0	1.0				
Promedio	1.7	0.4	0.1	2.2	3.7	0.7	2.4	6.9

Fuente: Elaboración propia.

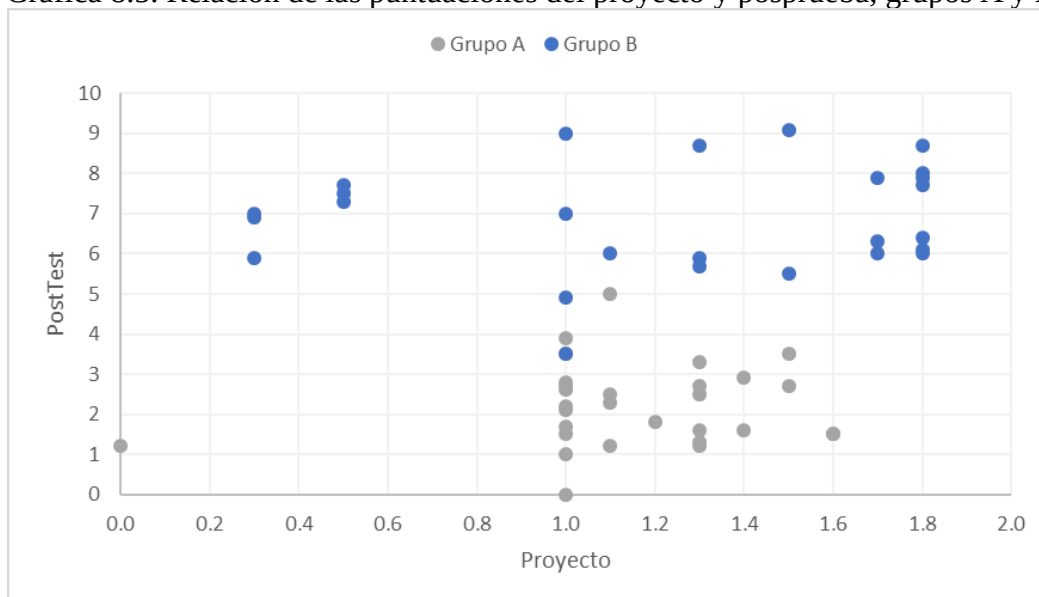
Como se aprecia en la Tabla anterior, 8.7, los estudiantes del grupo A tuvieron dificultad para contestar la sección 3 de la posprueba, que es la aplicación práctica de los conceptos, con respecto del grupo B. Precisamente, solo el 13.79% del grupo A respondió la sección 3 con una puntuación que oscila entre .8 y 1.7 de un valor total de 4.2 de la sección. El 100% de los estudiantes del grupo A respondieron la sección 1, sin embargo, se infiere que no han comprendido los conceptos debido a que se tiene una media de 1.7 de un valor de 4.8 de la sección. Hay que mencionar, además, que el 89.66% de los estudiantes del grupo A respondieron la sección 2 obteniendo una media del .4, el valor total de esta sección es de 1.

Con respecto a los estudiantes del grupo B, el 100% respondieron la sección 1 y 2, y el 92.31% respondió la sección 3. El valor de la media para la sección 1 es de 3.7, para la sección 2 es de .7 y de la sección 3 es de 2.4.

Definitivamente en la posprueba existe diferencia entre el puntaje obtenido por los estudiantes del grupo A y del grupo B. Pero también hay que reconocer que los estudiantes del grupo B en la sección 3 que debían aplicar los conceptos de la POO no salieron altos considerando la media del grupo mas no así el puntaje individual de cada estudiante.

Por otro lado, se consideró importante realizar una gráfica de dispersión para mostrar los valores de los resultados obtenidos en el proyecto y en la posprueba por cada estudiante, debido a que son las dos evidencias contundentes que se tienen de los puntajes. En la gráfica 8.5 se puede observar cómo los estudiantes del grupo A que obtuvieron un puntaje entre 1.0 y 1.6 de un valor total de 2.0, en el posttest tuvieron dificultades para las secciones siendo la puntuación más alta de 5.0. Caso contrario, los estudiantes del grupo B incluyendo aquellos que en su proyecto tuvieron puntajes debajo de 1.0 respondieron la posprueba obteniendo puntuaciones entre 6.0 y 9.1, a excepción de 6 estudiantes que obtuvieron puntuaciones entre 3.5 y 5.9.

Gráfica 8.5. Relación de las puntuaciones del proyecto y posprueba, grupos A y B



Fuente: Elaboración propia.

Como resultado de este análisis, se deduce que el grupo B, grupo experimental, con el que se utilizó el modelo MoACPOO comprendió y aplicó mejor los conceptos que el grupo A, mejorando su proceso de desarrollo y por ende sus programas, prueba de ello son los resultados en el proyecto y en la posprueba.

En este mismo sentido y para comprobar si existe hallazgo de que existen diferencias significativas entre las medias de las puntuaciones obtenidas en la posprueba por los estudiantes que aplicaron el modelo MoACPOO, grupos B, y las puntuaciones obtenidas por los estudiantes que no aplicaron el modelo, grupo A, se realizó la prueba de t student para muestras independientes con un nivel de confianza del 0.05%, para aceptar o rechazar la hipótesis nula (H_0) de las medias “No existen diferencias entre las medias” y por ende aceptar o rechazar la hipótesis nula de este trabajo doctoral “Si se utiliza un modelo que incluya un conjunto de buenas prácticas y criterios de calidad no se mejorará el desarrollo de programas en los cursos de programación orientada a objetos”, ver Imagen 8.2.

Imagen 8.2. Prueba t student de la posprueba

Estadísticos de grupo				
Grupo	N	Media	Desviación tip.	Error tip. de la media
Puntuación A	29	2.217	1.0344	.1921
B	26	6.869	1.3472	.2642

Prueba de muestras independientes									
		Prueba de Levene para la igualdad de varianzas		Prueba T para la igualdad de medias					
		F	Sig.	t	gl	Sig. (bilateral)	Diferencia de medias	Error tip. de la diferencia	95% Intervalo de confianza para la diferencia
Puntuación	Se han asumido varianzas iguales	2.247	.140	-14.447	53	.000	-4.6520	.3220	-5.2978 -4.0061
	No se han asumido varianzas iguales			-14.241	46.751	.000	-4.6520	.3267	-5.3092 -3.9947

Fuente: Elaboración propia.

Como se puede observar en la Imagen anterior, 8.2, los resultados de la prueba de muestras independientes de la prueba T student para la igualdad de medias indica que existe una diferencia de -4.6520 entre las medias de la posprueba de los grupos A y B. Además, al no existir el cero entre los límites del intervalo de confianza, inferior y superior, se rechaza la hipótesis nula de las medias H_0 , así como la del trabajo doctoral, por lo que se acepta la hipótesis alternativa de este trabajo doctoral H_1 “Si se utiliza un modelo que incluya un conjunto de buenas prácticas y criterios de calidad se mejorará el desarrollo de programas en los cursos de programación orientada a objetos”.

Considerando el análisis de las puntuaciones de la posprueba, se indica que el modelo MoACPOO contribuyó en la mejora de los programas de los estudiantes al hacer uso de buenas prácticas de ingeniería de software y CC en el proceso de desarrollo, que conlleva la comprensión de conceptos de la POO en un curso de PI, lo que se apoya por un lado del puntaje alcanzado en el proyecto y por el otro del puntaje obtenido en la posprueba. Conviene subrayar que, aunque los estudiantes no obtuvieron calificaciones altas para acreditar el curso, han comprendido los conceptos de POO y el 50% de los estudiantes los han aplicado estos de manera aceptable considerando el puntaje de la posprueba arriba de 7.0.

Por otra parte, el modelo MoACPOO al ser probado sólo en un grupo, no se puede generalizar que sea un modelo que mejore la calidad de los programas en todos los cursos de POO, ya que para poder afirmarlo es necesario que sea aplicado en otros cursos y comparar los resultados.

Conclusiones

En este capítulo se expusieron los resultados de la evaluación del modelo MoACPOO que está sustentado en el uso de buenas prácticas y CC que se encuentran incluidas en el CV que incorpora el modelo, cumpliendo así con el objetivo de este capítulo.

Los estudiantes del grupo B en general se enfrentaron al uso de buenas prácticas que desconocían, los principios SOLID que se les ha dificultado su aplicación, así como llevar el registro de los errores en una bitácora por lo que esta última no fue aplicada. Debido al tiempo para darle prioridad a la comprensión de los conceptos de la POO, de su aplicación y de que los estudiantes siguieran el modelo MoACPOO, los CC se dieron a conocer a los estudiantes y se les explicó su finalidad e importancia, pero las métricas no fueron calculadas por ellos sino por el docente aunado a que se requería de la confiabilidad de los resultados.

Para la evaluación del modelo se consideraron dos elementos importantes, el primero fue el proyecto realizado por los estudiantes de los grupos A y B, grupo de control y experimental respectivamente, en donde no existieron diferencias significativas de los puntajes obtenidos por los estudiantes de ambos grupos, que podría pensarse que el modelo MoACPOO no contribuye en la mejora del desarrollo de programas en los cursos de programación orientada a objetos. Sin embargo, existe una diferencia de 31.56 puntos porcentuales entre el nivel de satisfacción excelente obtenido por los estudiantes del grupo B, 38.46%, con respecto del

grupo A, 6.90%. Otro rasgo importante, es que el 19.23% de estudiantes del grupo B lograron un puntaje satisfactorio con una diferencia de 32.49 puntos porcentuales con respecto a los estudiantes del grupo A donde el 51.72% de los estudiantes alcanzó un nivel de satisfacción suficiente en sus proyectos. Así, se deduce que el modelo MoACPOO contribuye en la mejora del desarrollo de programas.

Como la deducción no se apoya de un análisis estadístico, se aplicó una posprueba sobre la comprensión de conceptos de la POO y de la aplicación de estos en pequeño caso práctico, a los estudiantes de los grupos A y B. Con las puntuaciones obtenidas de la posprueba se realizó un análisis estadístico con la prueba t student para muestras independientes, teniendo como resultado que existen diferencias entre la media de las muestras, lo cual es evidente al contrastar las puntuaciones obtenidas por los estudiantes del grupo A, donde el puntaje más alto fue de 5.0 en una escala del 0.0 al 10.0. De esta manera se rechazó la hipótesis nula y se acepta que, si se utiliza un modelo que incluya un conjunto de buenas prácticas y criterios de calidad se mejorará el desarrollo de programas en los cursos de programación orientada a objetos. Pero, no hay que perder de vista que el modelo MoACPOO sólo se probó con un grupo, por lo que es necesario y recomendable sea aplicado en otros cursos de POO.

Por lo que se refiere a los estudiantes que acreditaron el curso, en ambos grupos sólo dos estudiantes no acreditaron y las medias de ambos grupos no presentan diferencias significativas. Pero, considerando la posprueba de los estudiantes del grupo A, se deduce que tendrán dificultades en cursos posteriores de programación al no haber comprendido los conceptos ni aplicarlos.

Capítulo 9. Conclusiones

Con base en los resultados de la investigación se concluye que en los cursos de programación orientada a objetos, y en los demás cursos de programación, en las universidades es necesario el uso de un ciclo de vida o proceso de desarrollo para los programas que incluya buenas prácticas de ingeniería de software aunado a medir la calidad en el proceso de desarrollo de los estudiantes lo cual se realiza con métricas de calidad del software, de forma que impacte favorablemente en la formación profesional de los estudiantes y puedan enfrentar mejor el campo laboral. Como resultado se diseñó un modelo para la mejora de programas en los cursos de programación orientada a objetos denominado MoACPOO, ver Capítulo 7.

El diseño del modelo MoACPOO está fundamentado desde tres perspectivas: los datos, el empirismo y el positivismo. Así, el modelo se basa en la revisión de literatura de las buenas prácticas de ingeniería de software y de los criterios de calidad de los principales modelos de calidad del software, ver sección 2.3, en la metodología para obtener los resultados de las mejores prácticas de ingeniería de ingeniería de software que 84 docentes de 38 universidades públicas utilizan en los cursos de programación, ver capítulos 4, y del análisis estadístico de los datos resultados de la investigación para obtener las mejores prácticas de ingeniería de software y criterios de calidad., ver capítulos 5 y 6. En consecuencia, se cumplieron con los objetivos: identificar las mejores prácticas de ingeniería de software utilizadas por los docentes en los cursos de programación en las universidades públicas en México y definir los criterios de calidad que puedan aplicarse en el desarrollo de programas en los cursos de programación con base en los principales modelos de calidad del software. Asimismo, se realizó el análisis estadístico de los datos al probar el modelo en un ambiente real con un grupo experimental, ver capítulo 8, dando lugar al objetivo evaluar el modelo en un curso de programación orientada a objetos determinando como influye en los estudiantes y se comparó con el grupo de control.

La evaluación del modelo MoACPOO se probó en un curso de programación orientada a objetos, actualmente denominada programación imperativa, en el Centro Universitario UAEM Texcoco en el programa educativo en Informática Administrativa en el periodo 2019, considerando dos ejemplos pilotos, los cuales se utilizaron para el grupo experimental. El primero se utilizó para guiar a los estudiantes en la aplicación del modelo y el segundo fue dejado a los estudiantes como proyecto en donde aplicaron el modelo MoACPOO. Además,

se aplicó una posprueba al final del curso para conocer la comprensión de los conceptos y su aplicación.

Con respecto al proyecto realizado por los estudiantes del grupo experimental y después de obtener las métricas correspondientes a los CC por fase se identificó que:

- Al 73.08% de los estudiantes al realizar la abstracción de datos y funcional, en la fase de análisis, no consideran las excepciones que tiene como consecuencia que la ejecución del programa termine al ser probado con datos diferentes al dominio de datos que se deben ingresar. Sin embargo, al introducir los datos esperados el programa se ejecuta correctamente.
- En la fase de diseño al 73.08% principalmente se les dificulta aplicar los principios de responsabilidad única, abierto/cerrado y sustitución Liskov, ver sección 2.2, reflejado en la falta de cohesión en el diagrama de clases y a pesar de que identificaron bien la herencia, existe en algunos casos redundancia de atributos.
- En relación con la codificación, se deduce que al 50% de estudiantes se les olvida documentar el código ya que la mitad del grupo no los incluyó, que si bien es cierto el código que es descriptivo debe carecer de comentarios, es importante que en los cursos de programación al no ser totalmente descriptivos sean incluidos en bloques y no línea por línea.
- A los estudiantes se les complica entender los errores que detecta el compilador dado que se encuentran en inglés aunado a que el llevar el registro de los errores lo encuentran tedioso, encontrándolo como distractor en la corrección de su código. Por lo cual de manera consensuada se decidió que la bitácora de registro no se llevara a cabo.
- Se asume que las pruebas de unidad y de integración las realizó cada uno de los estudiantes en el desarrollo del proyecto, por lo que solo se aplicó un caso de prueba para verificar los resultados siendo exitosos en el 69.57% de los estudiantes.
- 6 estudiantes no terminaron el proyecto, 3 de ellos indicaron que fue complejo para ellos y 3 dieron prioridad a proyectos de otras materias.

Se debe agregar que para la evaluación del modelo MoACPOO y la comparación entre los grupos, se consideraron las puntuaciones obtenidas por los estudiantes en sus proyectos con base en una escala de satisfacción y en la posprueba. En la Tabla 9.1 se puede observar que

a pesar de que el 23.08% de estudiantes del grupo experimental no alcanzaron un nivel de satisfacción suficiente o más, el 76.92% obtuvo un nivel de satisfacción mejor con respecto al 96.55% del grupo de control. Y, aunque a los resultados obtenidos de la prueba t student para muestras independientes indica que no existen diferencias significativas, se deduce que la aplicación del modelo MoACPOO contribuye en la mejora de la calidad en el desarrollo de los programas de los estudiantes.

Tabla 9.1. Nivel de satisfacción del proyecto de cada grupo

Puntaje	Nivel de satisfacción	Grupo de control % de estudiantes	Grupo experimental % de estudiantes
1.6 – 2.0	Excelente	6.90% (2 estudiantes)	38.46% (10 estudiantes)
1.2 – 1.5	Satisfactorio	37.93% (11 estudiantes)	19.23% (5 estudiantes)
0.8 – 1.1	Suficiente	51.72% (15 estudiantes)	19.23% (5 estudiantes)
0.4 – 0.7	Insuficiente	0% (0 estudiantes)	11.54% (3 estudiantes)
0.0 - .0.3	Deficiente	3.45% (1 estudiante)	11.54% (3 estudiantes)

Notas: El proyecto tuvo una ponderación de 2.0.

Grupo de control, 29 estudiantes y grupo experimental, 26 estudiantes

Fuente: Elaboración propia.

En este mismo sentido, los resultados de la posprueba se muestran en la Tabla 9.2, y a pesar de que la media obtenida por sección de los estudiantes del grupo experimental no es alta que es lo que se esperaba, resulta evidente que existe una mejor comprensión de los conceptos y aplicación de estos comparados con el grupo de control, quienes tuvieron un nivel de satisfacción aceptable en los proyectos. Lo anterior se corrobora, con los resultados obtenidos del análisis estadístico con la prueba t student para muestras independientes que indica que existe una diferencia de -4.6520 entre los grupos de control y experimental. Por consiguiente, se acepta la hipótesis alternativa de este trabajo doctoral H1 “Si se utiliza un modelo que incluya un conjunto de buenas prácticas y criterios de calidad se mejorará el desarrollo de programas en los cursos de programación orientada a objetos”.

Tabla 9.2. Medias y porcentaje de estudiantes que respondieron la posprueba

Sección	Media Grupo de control	Media Grupo experimental	Grupo de control % de estudiantes que respondieron	Grupo experimental % de estudiantes que respondieron
1	1.7	3.7	100%	100%
2	0.4	0.7	89.66%	100%
3	0.1	2.4	13.79%	92.31%

Notas: Sección 1 (valor 4.8) y 2 (valor 1), comprensión de datos

Sección 3 (5.2), aplicación de conceptos

Fuente: Elaboración propia.

Se debe dejar claro que la validación del modelo MoACPOO se realizó sólo en un grupo debido a que el curso de programación orientada a objetos en la carrera de LIA se apertura cada año para dos grupos, por lo que es necesario que sea aplicado en otros cursos de programación orientada a objetos dentro de la misma institución en donde se llevó a cabo el experimento, así como en otras universidades para obtener resultados que evidencien que el modelo mejora la calidad de los programas en todos los cursos de programación orientada a objetos.

Es recomendable desarrollar una aplicación para la medición de las métricas del modelo MoACPOO debido a que el cálculo de estas representa un esfuerzo e inversión de tiempo al calcularlas por cada estudiante, sobre todo en grupos grandes. Una propuesta, es que la aplicación deberá ser sencilla de usar y debe presentar una interfaz gráfica con un menú de opciones: 1) Estudiantes, la cual permita capturar, eliminar o modificar los datos personales básicos de los estudiantes, como es su matrícula y nombre, para poder identificarlos y obtener los resultados de cada uno. 2) CC, que permita seleccionar el CC que se desea medir, el cual deberá incorporar todas las métricas asociadas a este, así como la interpretación del valor medido. Por cada métrica se solicitará se ingresen los datos esperados y los reales, de cada estudiante, a fin de que al realizar el cálculo se obtenga el resultado de cada métrica y a su vez se obtenga el resultado de cada CC. 3) Reportes de resultados de métricas, se generará un reporte tabular en donde se visualicen los resultados de cada una de las métricas por fase, por CC y por estudiante y 4) Generación de estadísticas, se emitirán graficas de los resultados obtenidos de las métricas por fase y CC, ya sea por estudiante o por grupo. Además, al seleccionar cada métrica y CC se dará una descripción de ellos.

Actualmente no existe un modelo que incluya buenas prácticas de ingeniería de software y criterios de calidad que permita medir la calidad en el proceso de desarrollo de programas en los cursos de programación de manera integral, por lo que el modelo propuesto MoACPOO es una contribución para apoyar en dichos cursos, y de ser probado en otros cursos de programación en diversas universidades del país con resultados positivos, el modelo MoACPOO podría convertirse en un estándar de facto. Sin embargo, para que el modelo pueda ser utilizado por docentes en los cursos de programación primero deben conocerlo, lo cual se puede lograr haciendo una guía del modelo aunado a un plan de trabajo que guíe al docente y por ende al estudiante en la aplicación del modelo con base al ejemplo piloto 1,

ver Anexo 2, a fin de introducirlos con las buenas prácticas de ingeniería de software y CC. Asimismo, se explicarían como se deben obtener cada una de las métricas, con base en ejemplos. Como parte de la guía, se propondrá hacer uso del ejemplo piloto 2, ver Anexo 2, para que los estudiantes den solución al mismo y los docentes puedan evaluar la pertinencia del modelo al obtener los resultados de los estudiantes, mismos que deberán ser dados a conocer. Por supuesto, que sería un trabajo a futuro como continuación de este trabajo doctoral, el cual se puede seguir como parte de un proyecto académico, con o sin recursos, para poder acceder a las Universidades ya sea por medio de redes de colaboración o por gestiones entre académicos.

En este mismo sentido, el modelo MoACPOO podría tomarse como base para ser adaptado a los cursos de programación estructurada, debido a que las fases del proceso de desarrollo y las acciones no cambian. El nombre de cuatro de los cinco artefactos son los mismos a excepción del artefacto de la fase de análisis sustituyéndose el catálogo de clases por un catálogo de elementos. En la fase de diseño el modelado de la solución será con diagramas de flujo, algoritmos o pseudocódigo, y como los principios SOLID son aplicados en la POO, es necesario que, al realizar el diseño, en programación estructurada, se aplique el principio de responsabilidad único al hacer uso de funciones. Las prácticas de código limpio pueden utilizarse en la codificación de las soluciones, así como hacer uso de las bitácoras de errores y casos de prueba. En relación con los CC, las métricas correspondientes a los CC completitud y consistente de las fases de análisis y diseño respectivamente, no podrían aplicarse. Sin embargo, en las métricas del CC completitud se pueden sustituir por funciones en lugar de métodos, variables globales en vez de atributos y las excepciones y relaciones deben eliminarse. Con respecto a las métricas del CC consistente, se cambiarían las métricas a variables globales, variables locales y métodos y en el CC entendible correspondiente a la fase de codificación, se eliminaría las métricas de clase y las métricas de métodos se cambiarían por funciones. Las métricas de los CC corrección y exactitud de las fases de compilación y pruebas, no tendrían cambios.

Por otro lado, es importante se realicen investigaciones a nivel nacional sobre la situación de los cursos de programación que incluyan sino a todas las universidades si una muestra mayor, haciendo uso de una muestra aleatoria estratificada constante a fin de considerar la misma cantidad de docentes por cada una de las universidades. La importancia es disponer de datos

estadísticos de las dificultades en los cursos de programación en el país, identificando por estados las áreas de oportunidad para alinearlas a sus características laborales y sociales, para generar estrategias para ofrecer a los estudiantes una formación integral que les permita potenciar sus habilidades y competencias para dar solución a proyectos de software.

Finalmente, se recomienda hacer un vínculo entre las academias y las empresas a fin de obtener las competencias y habilidades que requieren tengan los estudiantes, y de esta manera adaptar el modelo MoACPOO bien que se incluya en un marco de trabajo que se proponga para alinear y organizar los cursos relacionados con la programación en las universidades, como la metodología para la enseñanza de la programación orientada a objetos propuesta en Shefer, et al (2018) con la diferencia de que sea para cualquier universidad. Para hacer la vinculación entre las academias y las empresas, se partiría de un proyecto entre cuerpo académicos del Centro Universitario UAEM Texcoco, (CUUT), de la Universidad Popular Autónoma del Estado de Puebla (UPAEP), de la Universidad Autónoma de Zacatecas (UAZ) y de la Benemérita Universidad Autónoma de Puebla (BUAP) en el contexto de competencias y habilidades que requieren las empresas que desarrollan software. En consecuencia, se haría un estudio para identificar a las empresas con las cuáles se pudiera realizar la vinculación, se seleccionarían y se contactarían con la finalidad de poder acceder a estas con el objetivo de hacer un convenio de colaboración después de presentarles el proyecto, subrayando la importancia de hacer sinergia dado que los egresados serán los futuros desarrolladores de software y deben estar formados para cubrir profesionalmente los requerimientos de la empresa para dar respuesta a las demandas y necesidades laborales. Así, se podrían obtener las competencias y habilidades que requieren diversas empresas lo cual se utilizaría, por un lado, en la adecuación de los planes de estudio y, por otro, para generar una segunda versión del modelo MoACPOO.

Referencias

- Adnan, N. H., & Ritzhaupt, A. D. (2018). Software engineering design principles applied to instructional design: What can we learn from our sister discipline? *Association for Educational Communications & Technology*, 77-94.
- Aguas García, N., Balderas Pérez, K., & Díaz Mendoza, J. (2018). Mesa de Trabajo: Diseño Curricular. In *El curriculum flexible y las competencias transformables* (pp. 8-15). México: Alfa-Omega.
- Alammary, A. (2019). Blended learning models for introductory programming courses: A systematic review. *PLoS ONE*, 14(9). doi:<https://doi.org/10.1371/journal.pone.0221765>
- Almarales Sarasola, M., & García Fernández, O. (2019). Tutorial linteractivo para la Enseñanza de la Programación Orientada a Objetos. *Revista de Investigación, Formación y Desarrollo*, 65-72. doi:<https://doi.org/10.34070/rif.v7i1>
- Álvarez C, A., Alarcón A, A., & Callejas C, M. (2016). Comparación de modelos y estándares de evaluación de calidad para una plataforma de aprendizaje virtual. *Actas de Ingeniería*, 2, pp. 254-262. Colombia.
- ANUIES. (2015). *Anuario estadístico 2015-2016*. Asociación Nacional de Universidades e Instituciones de Educación Superior. Retrieved from <http://www.anui.es.mx/iinformacion-y-servicios/informacion-estadistica-de-educacion-superior/anuario-estadistico-de-educacion-superior>
- Ayalew, Y., Tshukudu, E., & Lefoane, M. (2018). Factors Affecting Programming Performance of First Year Students at a University in Botswana. *African Journal of Research in Mathematics, Science and Technology Education*, 22(3), 363-373. doi:<https://doi.org/10.1080/18117295.2018.1540169>
- Bennedsen, J., & Caspersen, M. (2007). Failure Rates in Introductory Programming. *inroads – The SIGCSE Bulletin*, 39(2), 32-36.
- Bennedsen, J., & Caspersen, M. (2019). In Introductory Programming — 12 Years Later. *ACM Inroads*, 10(2), 30-35. doi:<https://doi.org/10.1145/3324888>
- Bohem, B. W., Brown, J. R., & Lipow, M. (1976). Quantitative evaluation of software quality. *ICSE '76 Proceedings of the 2nd international conference on Software engineering*, 592-605.
- Bosse, Y., & Gerosa, M. A. (2016). Why is programming so difficult to learn? Patterns of Difficulties Related to Programming Learning. *ACM SIGSOFT Software Engineering Notes*, 41(6).
- Boudia, C., Bengueddach, A., & Haffaf, H. (2019). Collaborative Strategy for Teaching and Learning Object-Oriented Programming Course: A Case Study at Mostafa Stambouli

- MascaraUniversity, Algeria. *Informatica*, 43, 129-144.
doi:<https://doi.org/10.31449/inf.v43i1.2335>
- BS ISO/IEC. (2011). *Systems and software engineering -Systems and Software Quality Requirements and Evaluation (SQUARE)- System and software quality models*. England: BSI Standards Publication. Retrieved from
<https://pdfs.semanticscholar.org/57a5/b99e244337c9f4678b5b23d25.pdf>
- Callejas-Cuervo, M., Alarcón-Aldana, A. C., & Álvarez-Carreño, A. M. (2017). Modelos de calidad del software, un estado del arte. *Entramado*, 13(1), 236-250.
- Chemuturi, M. (2011). *Mastering Software Quality Assurance: Best Practices, Tools and Techniques for Software Developers*. J. Ross Publishing. Books 24x7.
- Dietz, L. W., Manner, J., Harrer, S., & Lenhard, J. (2018). Teaching Clean Code., (pp. 24-27).
- Dooley, J. F. (2017). *Software development, design and coding: With patterns, debugging, unit testing, and refactoring*. Apress. Books24x7.
- Dromey, R. G. (1995). A Model for Software Product Quality. *IEEE Transactions on Software Engineering*, 21(2), 146-162. doi:10.1109/32.345830
- Figueiredo, J., Gomes, N., & García-Peñalvo, F. J. (2016). ne-Course for Learning Programming. *Fourth International Conference on Technological Ecosystems for Enhancing Multiculturality – TEEM'16*, (pp. 549-553).
doi:<http://dx.doi.org/10.1145/3012430.3012572>
- Fuentes-Rosado, J. I., & Moo-Medina, M. (2017, julio). Dificultades de aprender a programar. *Educación en ingeniería*, 12(24), 76-82.
- Gerrit Singer, L. (2013). *Improving the adoption of software engineering practices trough persuasive interventions*. Lulu.com.
- Ghanem , M., HamaAli, K., Talal, M., & Layth, F. (2018). Three Levels Quality Analysis Tool for Object Oriented Programming. *International Journal of Advanced Computer Science and Applications*, 9(11), 522-536.
- González Reyes, A., Hernández González, A., & André Ampuero, M. (2016). Modelo básico inicial de calidad externa para productos de software. *Revista Cubana de Ciencias Informáticas*, 10, 94-111. Retrieved from <http://www.redalyc.org/articulo.oa?id=378346333008>
- Hernández Sampieri, R., Fernández-Collado, C., & Baptista Lucio, P. (2006). *Metodología de la Investigación*. México: Mc Graw Hill.
- Humphrey, W. S. (2000). *The Personal Software Process, Technical Report*. Pittsburgh: Carnegie Mellon University.

- Humphrey, W. S. (2001). *Introducción al Proceso de Software Personal*. (J. Zapata Martínez, Trans.) Madrid: Addison Wesley.
- IEEE Computer Society. (2014). *SWEBOK V3.0 Guide to the Software Engineering Body of Knowledge*. IEEE.
- Ifeanyi, A., Chukwuemeka, A., Uzoma, A., Chioma, A., & Okemiri, H. (2019). Technological Advancement in Object Oriented Programming Paradigm for Software Development. *International Journal of Applied Engineering Research*, 14(8), 1835-1841.
- INEGI. (2019). Estadísticas a propósito de las ocupaciones relacionadas de las Tecnologías de la Información y Comunicación datos nacionales. *Comunicado de Prensa Núm. 301/19*. Retrieved from https://www.inegi.org.mx/contenidos/saladeprensa/aproposito/2019/OcupaTIC2019_Nal.pdf
- Insuasti, J. (2016). Problemas de enseñanza y aprendizaje de los fundamentos de programación. *Revista educación y desarrollo social*, 10(2), 234-246.
- ISO/IEC. (2016). *Systems and software engineering -Systems and Software Quality Requirements and Evaluation (SQUARE)- Measurement of system and software product quality*. Vernier, Geneva, Switzerland: ISO/IEC. Retrieved from <https://www.sis.se/api/document/preview/920620/>
- ISO/IEC FDIS. (2000). *Information technology -Software product quality- Part 1: Quality model*. Geneva: ISO/IEC. Retrieved from <https://www.cse.unsw.edu.au/~cs3710/PMmaterials/Resources/9126-1%20Standard.pdf>
- ISO/IEC JTC1. (2001). *ISO/IEC 9126-4:Software Engineering - Software product quality - Part 4: Quality in use metrics*. ISO/IEC.
- Janicijevic, I., Krsmanovic, M., Zivkovic, N., & Lazarevic, S. (2016). Software quality improvement: a model based on managing factors impacting software quality. *Software Quality Journal*, 24, 247-270. doi:10.1007/s11219-014-9257-z
- Joint Task Force on Computing Curricula. (2015). *Software Engineering 2014: Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering*. Association for Computing Machinery. Retrieved from <https://www.acm.org/binaries/content/assets/education/se2014.pdf>
- Joyanes Aguilar, L. (2008). *Fundamentos de programación. Algoritmos, estructura de datos y objetos*. (Cuarta ed.). España: McGraw-Hill.
- Jusuf, H., Ibrahim, N., & Suparman, A. (2019). Developing a Hybrid Learning Strategy for Students' Engagement in Object-Oriented Programming Course. *Universal Journal of Educational Research*, 7(9A), 78-87. doi:10.13189/ujer.2019.071610

- Kanaparan, G., Cullen, R., & Mason, D. (2019). Effect of Self-efficacy and Emotional Engagement on Introductory Programming Students. *Australasian Journal of Information Systems*, 23. Retrieved from <https://journal.acs.org.au/index.php/ajis/article/view/1825/883>
- Kofi Sarpong, A.-M., John Kingsley, A., & Prince Yaw, O. (2013). Causes of Failure of Students in Computer Programming Courses: The Teacher – Learner Perspective. *International Journal of Computer Application*, 77(12), 27-32.
- Krusche, S., Berisha, M., & Bruegge, B. (2016). Teaching code review management using branch based workflows. *IEEE/ACM 38th IEEE International Conference on Software Engineering Companion*, 384-393. doi:10.1145/2889160.2889191
- Kumar, M., Ranjan, S., & Gupta, A. (2020). Object-Oriented Metrics for Quality Improvement of Object-Oriented Software. *International Journal of Innovative Technology and Exploring Engineering (IJITEE)*, 9(3), 26-29. doi:10.35940/ijitee.B7394.019320
- Kunkle, W. M., & Allen, R. B. (2016). The Impact of Different Teaching Approaches and Languages on Student Learning of Introductory Programming Concepts. *ACM Trans. Comput. Educ.*, 16(1), 26. doi:<http://dx.doi.org/10.1145/2785807>
- Lindoo, E. (2018). BACK to the Basics in an Effort to Improve Student Retention in intro to Programming Classes. 34(2), 72-79.
- López Román, L. (2011). Metodologías para la enseñanza aprendizaje de la programación estructurada y orientada a objetos. *Sistemas, cibernética e informática*, 8(1), 52-60.
- Lynch, J. (2018). *Project Resolution Benchmark Report*. The Standish Group International, InC. Retrieved from https://www.standishgroup.com/sample_research_files/DemoPRBR.pdf
- Ma, Z. (2017). An approach to improve the quality of object-oriented models from novice modelers through project practice. *Front. Comput. Sci*, 11(3), 485-498. doi:10.1007/s11704-016-5164-8
- Machín, I. T. (2018). Intelligent Tutor for Programming System Using Multiple Intelligences. *IEEE Latin America Transactions*, 16(2), 634-638.
- Macías Brambila, H. R., Zamora Ramos, V. M., Osorio Angel, S., & Jiménez Rodríguez, M. (2016). The development of skills in software desing using a virtual learning enviroment. *Tecnología educativa revista CONAIC*, 3(1), 22-28.
- Martín, R. C. (2003). *Agile software development: principles, patterns, and practices*. Nueva Jersey: Prentice Hall.
- Martin, R. C. (2009). *Clean Code A handbook of Agile Software Craftsmanship*. United States of America: Prentice Hall.

- Martin, R. C., & Martín, M. (2006). *Agile Principles, Patterns, and Practices in C#*. Nueva Jersey: Prentice Hall PTR.
- McCall, J. A., Richards, P. K., & Walters, G. F. (1977). *Factors in Software Quality*. RADC TR-77-369. Rome: Rome Air Development Center. Retrieved from <https://pdfs.semanticscholar.org/82a9/18fd83f1c0addb890ef313ff892807a10a11.pdf>
- Mechan, S., Phalp, K., & Grimm, F. (2016). Towards a Better Object-Oriented Software Development Education Using the DCI Software Architecture. *Software Engineering Education Going Agile*, 113-121.
- Mejía, A. O. (2017). Personal calificado en la indutria de las TICs en México: El caso de las Mipymes. *XVII congreso Latino-Americano de Gestión Tecnológica*, (pp. 1-20). México. Retrieved from http://www.uam.mx/altec2017/pdfs/ALTEC_2017_paper_523.pdf
- Nava, R. M., & Pinto, C.A, C. A. (2013). Mejora del aprendizaje de Programación Orientada a Objetos. *Revista iberoamericana para la investigación y el desarrollo educativo*, 10.
- Nel, G., Nel, L., & Cronje, J. (2015). Attributes contributing to students use of quality software development practices. *The african journal of information and communication*, 15, 38-50.
- Norero Ochoa, G. (2017). Aprendizaje Basado en Juegos para la Enseñanza de la Programación Orientada a Objetos y el Diseño Multimedia. *Diseño, Arte y Arquitectura*, 21-32.
- OECD. (2019). Higher Education in Mexico: Labour Market Relevance and Outcomes, Higher. París: OECD Publishing. Retrieved from <https://doi.org/10.1787/9789264309432-en>
- Olier, A. J., Gómez, A. A., & Caro, M. F. (2017). Design and Implementation of a Teaching Tool for Introduction to Object Oriented Programming. *IEEE LATIN AMERICA TRANSACTIONS*, 15(1), 97-102.
- Olivares Rojas, J. C., Jiménez Murillo, J. A., Ortiz Ortiz, O., & Rodriguez Maya, N. (2015). Desarrollo de una aplicación para fortalecer el aprendizaje de los fundamentos de programación. *Revista de ciencia e ingeniería del instituto tecnológico superior de coatzacoalcos*, 2(2), 1-4.
- Özmen, B., & Altun, A. (2014). Undergraduate Students' Experiences in Programming: Difficulties and Obstacles. *Turkish Online Journal of Qualitative Inquiry*, 5(3), 9-27.
- Pérez, J., & Azuaje, M. (2019). LE1: una estrategia amistosa para un curso introductorio de programación. *Educación en Ingeniería*, 14(28), 45-53. doi: <http://doi.org/10.26507/rei.v14n28.998>
- Piattini Velthuis, M. G., García Rubio, F. O., García Rodríguez de Guzman, I., & Pino, F. (2015). *Calidad de sistemas de Información*. España: RA-MA.

- Pressman, R. S. (2002). *Ingeniería de software un enfoque práctico*. España: Mac Graw Hill.
- Pressman, R. S. (2010). *Ingeniería del Software un enfoque práctico* (Séptima ed.). México: McGraw Hill.
- Qian, Y., & Lehman, J. (2017). Students' Misconceptions and Other Difficulties in Introductory Programming: A Literature Review. *ACM Trans. Comput. Educ*, 18(1), 24. doi:<https://doi.org/10.1145/3077618>
- Romero, M., Lepage, A., & Lille, B. (2017). Computational thinking development through creative programming in higher education. *International Journal of Educational Technology in Higher Education*, 14(1), 1-15.
- Rong, G., Zhang, H., Qi, S., & Shao, D. (2016). Can software engineering students program defect-free? an educational approach. *IEEE/ACM 38th IEEE International Conference on Software Engineering Companion*, 364-373. doi:10.1145/2889160.2889189
- Rumbaugh, J., Jacobson, I., & Booch, G. (2000). *El lenguaje unificado de modelado. Manual de referencia*. Madrid: Addison Wesley.
- Sakeenah, A., & Mabroukah, A. (2019). An Interactive Animation Tool for Java Object Oriented Programming Understanding. *International Journal of Programming Languages and Applications (IJPLA)*, 9(1/2/3). doi:10.5121/ijpla.2019.9301
- Sánchez-García, J. E., Urías-Ruiz, M., & Gutiérrez-Herrera, B. E. (2015). Análisis de los Problemas de Aprendizaje de la Programación Orientada a Objetos. *Ra Ximhai*, 11(4), 289-304.
- Serna, E. M. (2013). *Prueba Funcional el Software, Un proceso de verificación constante*. Colombia, Medellín: Fondo Editorial ITM. Retrieved from <https://fondoeditorial.itm.edu.co/libros-electronicos/prueba-funcional-software/mobile/index.html>
- Shefer, O. P., Nosova, L. S., & Lebedeva, T. N. (2018). A Modern Methodology for Teaching Programming at a University. *Scientific and Technical Information Processing*, 45(2), 81-86. doi:10.3103/S0147688218020077
- Stevenson, J., & Wood, M. (2018). Recognising object-oriented software design quality: a practitioner-based questionnaire survey. *Software Qual Journal*(26), 321-365. doi:10.1007/s11219-017-9364-8
- Suárez Lorenzo, F., & Gárzas Parra, J. (2014). *I Jornada sobre Calidad del Producto Software e ISO 25000*. Santiago de Compostela: Xunta de Galicia.
- Sun, Q., Wu, J., & Liu, K. (2020). Toward Understanding Students' Learning Performance in an Object-Oriented Programming Course: The Perspective of Program Quality. *IEEE Access*, 8, 37505-37517. doi:10.1109/ACCESS.2020.2973470

- Task Group on Information Technology Curricula. (2017). *Information Technology Curricula 2017: Curriculum Guidelines for Baccalaureate Degree Programs in Information Technology*. New York, USA: Association for Computing Machinery. doi:10.1145/3173161
- Thongmak, M. (2018). Creating gameful experience in the object-oriented programming classroom: A case study. *Online Journal of Applied Knowledge Management*, 6(1), 30-53.
- Thuné, M., & Eckerdal, A. (2019). Analysis of Students' learning of computer programming in a computer laboratory context. *European Journal of Engineering Education*, 44(5), 769-786. doi: 10.1080/03043797.2018.1544609
- Torchiano, M., & Bruno, G. (2018). Integrating Software Engineering Key Practices into an OOP Massive In-Classroom Course: an Experience Report., (pp. 64-71). doi:Publication:SEEM '18: Proceedings of the 2nd International Workshop on Software Engineering Education for Millennials
- UAM. (2013). Informe de actividades. 1-334. Retrieved from http://www.cua.uam.mx/pdf/Informe_anual_2013_Unidad_Cuajimalpa_Rector.pdf
- Universidad Autónoma Metropolitana. (2017). Resultados de la encuesta aplicada a egresados de la Licenciatura en Ingeniería en Computación, división de Ciencias Básicas e Ingeniería Unidad Azcapotzalco. (d. d. egresados, Ed.) Retrieved from http://www.uam.mx/egresados/estudios/acreditacionlic/Unidad_Azcapotzalco/1_2017/P122_Ingenieria_en_Computacion.pdf
- Vihavainen, A., Airaksinen, J., & Watson, C. (2014). A Systematic Review of Approaches for Teaching Introductory Programming and Their Influence on Success. (pp. 19-26). New York: Association for Computing Machinery. doi:http://dx.doi.org/10.1145/2632320.2632349
- Watson, C., & Li, F. W. (2014). Failure Rates in Introductory Programming Revisited. *ITiCSE '14: Proceedings of the 2014 conference on Innovation & technology in computer science education*, (pp. 39-44). doi:https://doi.org/10.1145/2591708.2591749
- Wojszczyk, R., & Wójcik, R. (2016). The Model of Quality Assessment of Implementation of Design Patterns. *Computing and Artificial Intelligence, 13th International Conference. Advances in Intelligent Systems and Computing*, 474. doi:10.1007/978-3-319-40162-1_55
- Wong, Y., Maizatun, H., & Tan, W. (2018). Learning Object-Oriented Programming Paradigm Via Game-Based Learning Game – Pilot Study. *The International Journal of Multimedia & Its Applications (IJMA)*, 10(6), 181-197. doi:10.5121/ijma.2018.10615

ANEXO 1 Cuestionario

INSTRUCCIONES. Estimado profesor, por favor, lea cuidadosamente cada una de las preguntas que se presentan en el instrumento y se le solicita su valiosa colaboración para responder todas y cada una de las preguntas con veracidad, agradeciendo de antemano su franqueza. La información obtenida, será utilizada para fines de investigación. Gracias por su apoyo.

1. Entidad federativa de la universidad de procedencia: _____
2. Escuela o universidad de procedencia: _____
3. ¿Cuáles de las siguientes materias ha impartido en licenciaturas donde se imparten materias relacionadas con la programación?

Algoritmos computacionales	☐	Programación avanzada	☐
Programación estructurada	☐	Programación orientada a objetos	☐
Estructura de datos	☐	Otra ☐	Especifique _____

4. Cuando imparte las materias relacionadas con programación ¿Sigue algún proceso para el desarrollo de los programas [S/N]? ____ Si su respuesta es “S” Explique en qué consiste: _____

5. Elija el enfoque con el que usted trabaja cuando imparte las materias relacionadas con la programación:

Uso de lenguaje de programación ☐ Uso de lenguaje algorítmico ☐

6. ¿Con cuál(es) de las siguientes herramientas se apoya para la enseñanza de la programación?

Scratch ☐ Alice ☐ Logo ☐ Ninguna ☐

Small Basic ☐ Kodu ☐ KPL ☐ Otra ☐ Especifique_____

7. Considera que los alumnos deben conocer lo que es un ciclo de vida del software para las materias de programación:

Totalmente de acuerdo ☐ De acuerdo ☐ Ni de acuerdo ni en desacuerdo ☐
En desacuerdo ☐ Totalmente en desacuerdo ☐

8. Considera que la fase de requerimientos es importante en la enseñanza de la programación [S/N] _____ ¿Por qué? _____

9. Por cada una de las fases presentadas indique cuáles son las mejores prácticas que utiliza con sus alumnos en la enseñanza de la programación.

Fase	Mejores prácticas que usted utiliza
Requerimientos	
Análisis	
Diseño	
Codificación	
Compilación	
Pruebas	
Mantenimiento	

10. Enumere del 1 al 9 las habilidades específicas que considera deben tener los alumnos para la programación (1 es la de mayor prioridad):

Razonamiento lógico-matemático	☐	Pensamiento algorítmico	☐
Uso de herramientas de diagramación	☐	Codificar una solución en LP ¹	☐
Uso de herramientas de los LP ¹	☐	Detección de defectos en el programa	☐
Reglas sintácticas y semánticas del LP ¹	☐	Uso de técnicas de diseño	☐
Uso de la técnica divide y vencerás	☐		☐

Si hay otra(s) que considere mencionela(s):

¹Lenguaje de Programación

11. Enumere del 1 al 9 las habilidades genéricas que considera deben tener los alumnos para la programación (1 es la de mayor prioridad):

Trabajo en equipo		Analizar información	
Interpretar información		Extraer conclusiones	
Organización de información		Clasificar información	
Capacidad de comunicación		Adaptación a cambios	
Toma de decisiones			

Si hay otra(s) que considere mencionela(s):

12. Para cada una de las siguientes marque con una **X**, según corresponda [1 “Siempre”, 2 “Casi siempre”, 3 “La mayoría de las veces”, 4 “Pocas veces”, 5 “Nunca”]

Cuando realiza la verificación del aprendizaje usted comprueba:	1	2	3	4	5
Que los resultados sean correctos					
La validación de los datos de entrada					
La ejecución del programa sea correcta					
La estrategia de aprendizaje					
Cuando los alumnos desarrollan un programa usted solicita:	1	2	3	4	5
Registrar defectos de sintaxis					
Registrar defectos lógicos					
Registrar defectos en ejecución					
Registrar tiempo de desarrollo					
Verificar resultados correctos					
Verificar programa sea claro y legible					
Poner comentarios en el código					
Los alumnos con base a un problema deben analizarlo para:	1	2	3	4	5

Identificar las variables a utilizar					
Identificar los datos de entrada					
Identificar los datos de salida					
Identificar formulas si existen					
Identificar funciones o métodos					
Los alumnos, previo a la codificación deben diseñar, usando:	1	2	3	4	5
Un algoritmo					
Un pseudocódigo					
Un diagrama de flujo					
En la codificación los alumnos deben:	1	2	3	4	5
Poner comentarios					
Conocer la sintaxis del lenguaje					
Codificar de manera secuencial					
Codificar de manera modular					
Las pruebas que se realizan a un programa son para:					
Corroborar resultados correctos					
Validación de los datos de entrada					
Las actividades que incluye en su didáctica son:	1	2	3	4	5
Revisión de código					
Depurar código					
Transcribir código					
Realizar pruebas de escritorio					
Modularizar un programa lineal					
Operar los programas					
Hacer seguimiento de variables					
Uso de diagramas de Nassi-Scheidernan					
Uso de diagramas de clases					
Uso de diagramas de actividades					

Gracias por su colaboración.

ANEXO 2 Ejemplos pilotos

Ejemplo piloto 1.

Planteamiento del problema

La sucursal bancaria “Coyoacán”, tiene dos tipos de cuentas: de ahorro y de crédito. Todas las cuentas bancarias se encuentran asociadas a un cliente, de esta manera las cuentas deben tener: nombre del cliente, número de cuenta, tipo de cuenta, saldo, cantidad depositada, fecha de depósito, retiro y el nuevo saldo. Las cuentas de crédito tienen una fecha de corte (depende de la cuenta), una fecha límite de pago (que son veinte días después de la fecha de corte), intereses por pago tardío del 16% sobre su saldo y comisión por retiro de efectivo del 10% sobre la cantidad retirada. Las cuentas de ahorro se les calcula un interés del 2% cada mes (los días 12) sobre su saldo, lo que incrementa su ahorro. Es importante comentar que el nombre del cuentahabiente no puede estar vacío lo cual se manejará mediante una excepción.

El programa deberá validar que exista saldo suficiente si se desea realizar un retiro, en caso de que no sea así se envía mensaje “no se tiene saldo disponible”; para el caso del depósito no puede ser una cantidad negativa desplegando el mensaje “El depósito no puede ser una cantidad negativa”. Asimismo, se debe validar que la fecha de pago sea entre la fecha de corte y la fecha límite de pago para no cobrar intereses.

Por otro lado, para la obtención del nuevo saldo, se debe considerar el depósito o pago realizado por el cliente, así como los intereses. Es importante destacar que las cuentas de ahorro tienen un monto de aportación inicial y una chequera; mientras que las de crédito tienen un límite de crédito de \$42,000.00. Ejemplos de salidas del programa:

El nombre del cuentahabiente es: Armando

El número de cuenta es: A1722543789

El tipo de cuenta es: Ahorro

Monto inicial aportado: \$2000.00

Chequera número: A2732-1

El saldo de la cuenta antes del depósito es: \$5827.80

Retiro realizado: \$100.00
Depósito realizado: \$0.00
Fecha de depósito: 4 abril 2017
Saldo de la cuenta después de movimientos:\$5727.80
Saldo de la cuenta con intereses: \$5727.80
El nombre del cuentahabiente es: Roberto
El número de cuenta es: A1847956239
El tipo de cuenta es: Ahorro
Monto inicial aportado: \$5000.00
Chequera número: A5233-5
El saldo de la cuenta antes del depósito es: \$8799.50
Retiro realizado: \$100.00
Depósito realizado: \$2630.00
Fecha de depósito: 12 mayo 2017
Saldo de la cuenta después de movimientos:\$11329.50
Saldo de la cuenta con intereses: \$11329.50
El nombre del cuentahabiente es: Rodolfo
El número de cuenta es: C5792854729
El tipo de cuenta es: Crédito
Saldo disponible: \$29174.50
Límite de crédito: \$42,000.00
Fecha de corte: 10 de abril de 2017
Fecha de límite de pago: 30 de abril de 2017
Saldo de la cuenta antes de movimientos: \$12825.50
Retiro realizado: \$110.00
Depósito realizado: \$5000.00
Fecha de depósito: 1 de julio de 2017
Saldo de la cuenta después de movimientos:\$7935.5
Intereses cargados: \$1269.68
Saldo de la cuenta después de intereses: \$9205.18

Ejemplo piloto 2.

Planteamiento del problema

Considere una escuela que tiene empleados quienes tienen un número de empleado, nombre, apellidos, edad, teléfono. Existen dos tipos de empleados profesores y administrativos. Los profesores a diferencia de los administrativos tienen asignadas horas clase, un pago por hora y un sueldo; mientras que los administrativos tienen un sueldo base, pago de horas extras y pago por guardias. En ambos casos el sueldo de los empleados se obtiene considerando horas clase y pago por hora o bien sueldo base, horas extras y pago por guardias. Una vez comprendido el problema desarrolle los siguientes incisos:

- 1) Describa la problemática planteada indicando los conceptos de POO que debe utilizar
- 2) Realice el análisis del problema realizando una abstracción de datos y funcional
- 3) Identifique los modificadores de acceso, las propiedades, las relaciones y excepciones
- 4) Realice el diagrama de clases haciendo uso de las relaciones del inciso 3.
- 5) Realice los diagramas de actividades de una de las clases.
- 6) Desarrolle el código en NetBeans, apóyese del diagrama de clases que incluye la abstracción de datos y funcional. Recuerde incluir constructores, métodos y más.
- 7) Revise su código antes de compilar y corrija los errores detectados.
- 8) Ejecute el programa y si no compila registre los errores de sintaxis en una bitácora que indique el error de sintaxis presentado.
- 9) Una vez que su código esté libre de errores de sintaxis y se haya ejecutado, revise que los resultados sean correctos, de lo contrario registre en la bitácora los errores lógicos, y en su caso de ejecución.
- 10) Finalmente realice las capturas de pantalla de la ejecución del programa, mismas que se deberán anexar en el documento en la sección de desarrollo, como son imágenes deberá indicar los títulos de estas.
- 11) Una vez desarrollado el documento deberá entregarlo considerando los puntos de los criterios de evaluación que se le entregaron al inicio del semestre.
- 12) Deberá incluir también la carpeta de su proyecto en *NetBeans*, a fin de que el docente pueda ejecutar el mismo.

ANEXO 3 Posprueba

**PROGRAMACIÓN IMPERATIVA
(PROGRAMACIÓN ORIENTADA A OBJETOS)
POSPRUEBA
PERIODO 2019-B**

Nombre del alumno: _____

Grupo: _____

I Responda cada una de las siguientes preguntas proporcionando los conceptos según corresponda (valor .2 cada pregunta):

1. ¿Qué es programación orientada objetos?

2. ¿Qué es el encapsulamiento?

3. ¿Qué es el polimorfismo?

4. ¿Qué es la reutilización?

5. ¿Qué es un objeto?

6. ¿Qué es una clase?

7. ¿Qué es una clase abstracta?

8. ¿Qué es la abstracción de datos y funcional?

9. ¿Qué es un método?

10. ¿Qué es un mensaje?

11. ¿Qué es un constructor?

12. ¿Cuáles son los métodos de acceso y para que se utiliza?

13. ¿Qué son las propiedades y qué métodos se asocian a éstas?

14. ¿Cuáles relaciones entre clases conoce?

15. ¿Qué es la herencia?

16. ¿Qué tipos de herencia existen?

17. ¿Qué tipo de herencia no es soportada por Java y como se puede emular?

18. ¿Qué es el *override* (sobreescritura de métodos)?

19. ¿Qué es el *overload* (sobrecarga de métodos)?

20. ¿Qué es una excepción?

21. ¿Qué es el principio de responsabilidad única?

22. ¿Qué es refactorización?

23. ¿Qué es un diagrama de clases?

24. ¿Qué es un diagrama de actividades?

II Indique en el paréntesis el número que corresponde (valor .1 cada una):

- | | |
|---|---|
| 1 Una entidad que tiene un estado, un comportamiento y una identidad. | 7 Definen el comportamiento de un tipo de objetos. |
| 2 Define un conjunto de objetos con características similares, en la que se almacenan datos y aquellos métodos (operaciones) que manipulan los datos. | 8 Conocidas como variables de instancia que definen el estado de un tipo de objetos al asociarlas con un valor. |
| 3 Es una invocación que se envía a un objeto, el cual se ejecutará al recibirlo; incluye el identificador que contiene la acción a realizar por el objeto junto con los datos que éste necesita para efectuar su trabajo. | 9 Se utiliza cuando un objeto utiliza a otro. |
| 4 Crea el objeto y devuelve una referencia al objeto creado. | 10 Es como una caja negra en donde existe el ocultamiento de los datos, en donde se pueden acceder a los atributos privados por medio de métodos público. |
| 5 Se utilizan con las variables de instancia privadas para poder modificar su valor o bien recuperarlo. | () Métodos |
| 6 Es un método que se utiliza para inicializar las variables de instancia: | () use |
| | () Atributos |
| | () Encapsulamiento |
| | () Constructor |
| | () Propiedades |
| | () Objeto |
| | () Mensaje |
| | () Clase |
| | () new |

III Desarrolle los incisos con base en el siguiente planteamiento:

Una veterinaria tiene empleados veterinarios y auxiliares, y como tal comparten los mismos datos: número de empleado, nombre, apellido materno, apellido paterno, edad y teléfono. Los veterinarios tienen una cédula profesional mientras que los auxiliares veterinarios aún se encuentran estudiando. En ambos casos el sueldo neto se calcula con base en un sueldo base más sus prestaciones, sin embargo, las prestaciones de un veterinario son prima por antigüedad y fondo de ahorro y para el auxiliar es apoyo de transporte y apoyo de comida. Recuerde hacer una clase para leer datos denominada `LecturaDatos`.

- 1) Realice la abstracción de datos y funcional (valor 1)
- 2) Desarrolle el diagrama de clases, debe tener al menos un método de *override* (sobreescripción) y *overload* (sobrecarga) (valor 1.2)
- 3) Desarrolle un diagrama de actividades de un método (valor .5)
- 4) Elija una clase de su diagrama y codifíquela. (valor 1)
- 5) Indique en su código con comentarios: el constructor, propiedades, instanciación de la clase `LecturaDatos` (valor .25)
- 6) Indique el código del método en donde incluyó la sobreescripción y la sobrecarga (valor .25)