De la Idea al Algoritmo: Un Análisis Exhaustivo de las Fases Preliminares en el Desarrollo de Software

Introducción: Más Allá del "Enunciado" - Formalizando el Proceso de Creación de Software

La percepción de que el desarrollo de software comienza con un "enunciado", seguido de un "diagrama de flujo" y finalmente la "codificación", refleja una comprensión intuitiva y fundamentalmente correcta de que la creación de un sistema informático es un proceso estructurado y no un acto espontáneo de escritura de código. Esta intuición es valiosa, ya que apunta a la existencia de un marco metodológico que la industria y la academia han refinado durante décadas. El objetivo de este informe es formalizar esa comprensión, proporcionando la terminología precisa y un análisis detallado de las etapas que un programador, o más bien un equipo de desarrollo, debe ejecutar antes de que se escriba una sola línea de código funcional.

El término informal "enunciado" corresponde, en el léxico de la ingeniería de software, a una de las fases más críticas y determinantes para el éxito de un proyecto: el **Análisis de Requisitos**. Este proceso, junto con otros, se enmarca dentro de lo que se conoce como el **Ciclo de Vida del Desarrollo de Software (SDLC, por sus siglas en inglés)**. El SDLC es un marco conceptual que define las tareas a realizar en cada etapa del desarrollo de un producto de software, desde su concepción inicial hasta su retirada.[1]

El viaje desde una idea de negocio abstracta hasta un producto de software tangible y funcional se puede concebir como un descenso a través de un gradiente de abstracción. Las fases iniciales operan en un nivel muy alto, enfocándose en el "qué" y el "porqué" del proyecto: ¿qué problema se está resolviendo? ¿Por qué es importante para el negocio? ¿Qué funcionalidades debe ofrecer el sistema? A medida que el proceso avanza hacia la codificación, las fases se vuelven progresivamente más técnicas, detalladas y concretas, centrándose en el "cómo": ¿cómo se estructurará el sistema? ¿Qué tecnologías se utilizarán? ¿Cómo se implementará cada algoritmo?

Este informe está estructurado para guiar al lector a través de este gradiente de abstracción. Se comenzará por desglosar las fases fundamentales pre-codificación, proporcionando una descripción exhaustiva de sus objetivos, actividades y entregables. Posteriormente, se contextualizarán estas fases dentro de los diferentes modelos o

metodologías del SDLC, como el modelo en Cascada y los enfoques Ágiles. Finalmente, se profundizará en los artefactos y técnicas clave, como los diagramas de flujo y su evolución moderna, para ofrecer una visión completa y profesional del proceso de creación de software.

Sección 1: La Arquitectura del Proceso: Las Fases Fundamentales Pre-Codificación

Antes de que un desarrollador pueda empezar a programar, es imperativo construir un cimiento sólido para el proyecto. Este cimiento se erige a través de una serie de fases secuenciales y lógicas que transforman una necesidad abstracta en un plan de construcción técnico y detallado. Omitir o ejecutar de manera deficiente cualquiera de estas etapas es una de las principales causas de sobrecostos, retrasos y fracaso en los proyectos de software.

1.1. Fase 0: Planificación y Viabilidad - La Génesis del Proyecto

La verdadera primera fase del desarrollo de software no se pregunta qué hará el sistema, sino si el proyecto debe existir en primer lugar. Esta etapa, a menudo denominada el "fuzzy front-end" por su naturaleza exploratoria y no sujeta a plazos estrictos [2], establece las bases estratégicas y económicas de toda la iniciativa. Su objetivo es alinear la propuesta tecnológica con los objetivos del negocio y asegurar que el esfuerzo es justificable y alcanzable.

Las actividades clave durante esta fase son de naturaleza estratégica y analítica. Comienzan con la **identificación clara del problema u oportunidad de negocio** que el software pretende abordar.[4] Esto implica comprender el "dolor" del cliente o la ventaja competitiva que se busca obtener. A continuación, se realiza un **estudio de viabilidad** exhaustivo, que evalúa el proyecto desde múltiples perspectivas: técnica (¿tenemos la tecnología y la experiencia para construirlo?), económica (¿los beneficios esperados superan los costos de desarrollo y mantenimiento?) y operativa (¿se integrará el nuevo sistema en los flujos de trabajo existentes?).[6]

Una parte crucial de la planificación es la **definición del alcance (scope)** inicial del proyecto. Esto implica establecer límites claros sobre lo que el sistema hará y, lo que es igualmente importante, lo que no hará.[6] Esta delimitación es un mecanismo de defensa

fundamental contra el "scope creep" o la ampliación descontrolada del alcance, un fenómeno que puede hacer descarrilar proyectos al agotar recursos y extender indefinidamente los plazos.[8] Finalmente, se realiza una **estimación preliminar de recursos** (humanos, tecnológicos y financieros) y se establece un **cronograma de alto nivel** para las fases posteriores.[1]

El entregable principal de esta fase no es un documento técnico, sino un artefacto de negocio: un **plan de proyecto** o un **estudio de viabilidad aprobado**. Este documento sirve como la luz verde para proceder, asegurando que todas las partes interesadas clave comparten una visión común sobre los objetivos, el alcance y las restricciones del proyecto antes de invertir recursos significativos.

1.2. Fase 1: Análisis y Especificación de Requisitos - El Corazón del "Qué"

Esta es la fase que responde directamente a la noción de un "enunciado" y es, sin duda, la más crítica para el éxito del software.[10] Su propósito es traducir las necesidades y expectativas, a menudo vagas y abstractas de las partes interesadas (stakeholders), en un conjunto de especificaciones precisas, completas y verificables de lo que el sistema debe hacer.[11] Un error en esta etapa tiene un efecto de amplificación, siendo su corrección en fases posteriores entre 10 y 100 veces más costosa.[8]

El proceso comienza con la **obtención de requisitos (elicitation)**, que implica un trabajo proactivo para descubrir y recopilar las necesidades de todas las fuentes relevantes: clientes, usuarios finales, expertos en el dominio, analistas de negocio, etc. Las técnicas para esta recopilación son variadas e incluyen entrevistas, talleres colaborativos, cuestionarios, observación de procesos existentes y revisión de documentación.[13]

Una vez recopilados, los requisitos brutos deben ser sometidos a un riguroso **análisis y negociación**. Es común que los requisitos de diferentes stakeholders sean contradictorios o que las solicitudes iniciales no sean técnicamente viables o excedan el presupuesto. El rol del analista de sistemas es identificar estas ambigüedades y conflictos, facilitar la comunicación entre las partes para llegar a un consenso y ayudar a priorizar las funcionalidades.[8]

Los requisitos analizados se clasifican en dos categorías principales:

- **Requisitos Funcionales:** Describen las funcionalidades o servicios que el sistema debe proveer. Definen el "qué" hace el sistema. Por ejemplo: "El sistema debe

permitir al usuario registrarse con un correo electrónico y una contraseña" o "El sistema debe generar un informe de ventas mensual en formato PDF".

- **Requisitos No Funcionales:** Describen las cualidades o restricciones del sistema. Definen el "cómo" debe operar el sistema y establecen criterios de calidad. Ejemplos incluyen el rendimiento ("la página de inicio debe cargar en menos de 2 segundos"), la seguridad ("todas las contraseñas de usuario deben estar encriptadas en la base de datos"), la usabilidad, la fiabilidad y la compatibilidad.[16]

El resultado culminante de esta fase es el **Documento de Especificación de Requisitos de Software (SRS, por sus siglas en inglés)**. Este documento es el artefacto más importante de las fases iniciales. Actúa como un contrato formal entre el equipo de desarrollo y el cliente, detallando todo lo que el software hará y cómo se comportará.[6] Un buen SRS es inequívoco, completo, consistente, verificable (cada requisito puede ser probado) y rastreable (se puede seguir el origen de cada requisito y su implementación a lo largo del ciclo de vida).[5] Es la evolución formal y profesional del "enunciado" inicial.

1.3. Fase 2: Diseño del Sistema - La Construcción del "Cómo"

Con el "qué" claramente definido en el SRS, la fase de diseño se encarga de trazar el "cómo". Esta etapa funciona como un puente crucial entre los requisitos del dominio del problema y la solución en el dominio técnico. Aquí, los ingenieros y arquitectos de software toman las especificaciones funcionales y no funcionales y las utilizan para crear los planos detallados del sistema de software que se va a construir.[1]

El proceso de diseño no es monolítico; se aborda en dos niveles de abstracción distintos, pasando de lo general a lo específico:

- **Diseño de Alto Nivel (o Diseño Arquitectónico):** Esta primera etapa se centra en la estructura global del sistema. Se toman decisiones fundamentales que tendrán un impacto duradero en el proyecto. Las actividades incluyen la selección de la arquitectura de software (ej. cliente-servidor, microservicios, N-capas), la elección de las tecnologías clave (lenguajes de programación, frameworks, sistemas de gestión de bases de datos), la descomposición del sistema en sus principales módulos o componentes y la definición de las interfaces y las interacciones entre ellos.[12] El objetivo es crear un esqueleto robusto y escalable para la aplicación.
- **Diseño de Bajo Nivel (o Diseño Detallado):** Una vez establecida la arquitectura, el enfoque se desplaza hacia el interior de cada módulo definido en el diseño de alto

nivel. Para cada componente, se especifica su lógica interna de manera detallada. Esto implica definir las estructuras de datos específicas que se utilizarán, diseñar los algoritmos para cada una de las funciones que el módulo debe realizar y especificar las interfaces de programación de aplicaciones (APIs) con una precisión que permita su posterior implementación.[18] Es en esta etapa donde herramientas como los diagramas de flujo o el pseudocódigo se vuelven indispensables para describir la lógica de procesamiento.

El entregable principal de esta fase es el **Documento de Diseño de Software (SDD, por sus siglas en inglés)**. Este documento contiene los "planos" del software. Incluye diagramas de arquitectura, modelos de entidad-relación para la base de datos, diseños de la interfaz de usuario (UI), especificaciones detalladas de cada módulo y sus APIs, y cualquier otra información necesaria para guiar al equipo de programadores durante la fase de implementación.[12]

La progresión desde un concepto de negocio hasta un plan técnico detallado no es solo una secuencia de tareas, sino un mecanismo deliberado para la reducción de riesgos. La fase de Planificación aborda la pregunta abstracta "¿Deberíamos hacer esto?". El Análisis de Requisitos traduce esa idea en un "¿Exactamente, qué debería hacer?". El Diseño lo convierte en un "Técnicamente, ¿cómo lo haremos?". Esta progresión sistemática asegura que las decisiones más costosas y difíciles de revertir (las decisiones técnicas de diseño e implementación) se basen en un entendimiento profundo y validado del problema de negocio. Cada fase valida la anterior, minimizando el riesgo de construir un sistema técnicamente impecable pero que no resuelve la necesidad del usuario o del negocio.

1.4. El Lenguaje Visual de la Lógica: Diagramas de Flujo y Modelado Moderno

La mención del "diagrama de flujo" como un paso intermedio entre el "enunciado" y la "codificación" es particularmente perspicaz, ya que identifica una herramienta clásica y fundamental del **Diseño de Bajo Nivel**. El propósito de estas representaciones visuales es doble: son tanto una herramienta para el pensamiento estructurado como un documento para la comunicación técnica.

El rol principal de un **diagrama de flujo** es ofrecer una representación gráfica de un algoritmo o proceso.[20] Antes de sumergirse en la sintaxis específica de un lenguaje de programación, el desarrollador puede usar un diagrama de flujo para visualizar la secuencia de pasos, los puntos de decisión y los bucles que componen la lógica de una función o procedimiento. Esto permite identificar errores lógicos, simplificar procesos

complejos y organizar el pensamiento de una manera independiente del lenguaje de programación.[21] Los símbolos estandarizados, como los óvalos para el inicio y el fin, los rectángulos para los procesos, los rombos para las decisiones y los paralelogramos para la entrada/salida de datos, proporcionan un vocabulario visual común que facilita la comprensión.[21]

Sin embargo, aunque los diagramas de flujo siguen siendo útiles para ilustrar algoritmos específicos, la industria del software ha desarrollado herramientas de modelado más sofisticadas y completas. La más prominente es el **Lenguaje Unificado de Modelado (UML)**. UML no es un solo tipo de diagrama, sino una familia de diagramas estandarizados diseñados para visualizar, especificar, construir y documentar diferentes aspectos de un sistema de software. Su poder reside en su capacidad para modelar el sistema desde múltiples perspectivas:

- **Diagramas de Casos de Uso:** Se utilizan durante el análisis de requisitos para capturar las interacciones entre los "actores" (usuarios o sistemas externos) y el sistema. Ayudan a definir el alcance funcional del software de una manera que es fácilmente comprensible para las partes interesadas no técnicas.[1]
- **Diagramas de Clases:** Son fundamentales en el diseño arquitectónico de sistemas orientados a objetos. Modelan la estructura estática del sistema, mostrando las clases, sus atributos, métodos y las relaciones entre ellas (como la herencia o la asociación).[10]
- **Diagramas de Secuencia y Diagramas de Actividad:** Estos diagramas modelan el comportamiento dinámico del sistema. Los diagramas de secuencia muestran cómo los objetos interactúan entre sí a lo largo del tiempo para llevar a cabo un caso de uso. Los diagramas de actividad, considerados la evolución moderna de los diagramas de flujo, son excelentes para modelar flujos de trabajo complejos, procesos de negocio y la lógica de operaciones con múltiples ramificaciones y actividades paralelas.[4]

Estas herramientas de modelado visual cumplen una función dual crítica. Internamente, para el equipo de desarrollo, son herramientas de pensamiento que obligan a la claridad y al rigor en el diseño. Externamente, son documentos de comunicación que permiten validar la funcionalidad con los clientes (usando diagramas de casos de uso), discutir la arquitectura con otros ingenieros (usando diagramas de clases) y explicar algoritmos complejos (usando diagramas de actividad). Sirven como un puente indispensable entre la comprensión del negocio y la implementación técnica.

Sección 2: El Paradigma como Contexto: Modelos del Ciclo de Vida

Las fases de planificación, análisis y diseño no existen en un vacío. Son componentes de un marco de trabajo más amplio que dicta su secuencia, interrelación y el grado de formalidad con el que se ejecutan. Este marco es el modelo del ciclo de vida del desarrollo de software (SDLC). La elección del modelo adecuado depende de la naturaleza del proyecto, la estabilidad de los requisitos, el tamaño del equipo y la cultura de la organización. Comprender estos modelos es esencial para contextualizar cómo se llevan a la práctica las fases pre-codificación.

2.1. El Modelo Secuencial Clásico: La Metodología en Cascada (Waterfall)

El modelo en Cascada es el paradigma más antiguo y tradicional del SDLC, y es el que mejor se alinea con la percepción lineal del usuario ("paso 1, paso 2, paso 3"). Su característica definitoria es un flujo de desarrollo estrictamente secuencial. Cada fase del ciclo de vida debe completarse en su totalidad y su resultado (generalmente un documento formal) debe ser aprobado antes de que la siguiente fase pueda comenzar.[2] El progreso fluye de una fase a la siguiente en una única dirección, como el agua cayendo por una cascada, sin posibilidad de volver atrás fácilmente.[12]

Las fases canónicas de un modelo en Cascada puro son:

1. **Análisis de Requisitos:** Se recopilan y documentan todos los requisitos del sistema en el SRS.
2. **Diseño del Sistema:** Se crea la arquitectura y el diseño detallado del software, plasmado en el SDD.
3. **Implementación (Codificación):** Los desarrolladores escriben el código basándose en los documentos de diseño.
4. **Pruebas (Verificación):** El equipo de calidad prueba el sistema completo para encontrar y reportar errores.
5. **Despliegue (Instalación):** El software se instala en el entorno del cliente.
6. **Mantenimiento:** Se corrigen errores descubiertos tras el despliegue y se realizan mejoras.[11]

La principal ventaja del modelo en Cascada es su simplicidad y su facilidad de gestión. La división en fases discretas con entregables claros hace que la planificación y el seguimiento del proyecto sean sencillos.[12] Es un modelo adecuado para proyectos pequeños, con requisitos muy bien definidos, estables y comprendidos desde el principio.[3]

Sin embargo, su gran desventaja, y la razón por la que ha sido suplantado en gran medida para proyectos complejos, es su rigidez e inflexibilidad ante el cambio.[3] El modelo asume que todos los requisitos pueden ser definidos y congelados al inicio del proyecto, una suposición que rara vez se cumple en la realidad. Si durante la fase de pruebas se descubre un error en los requisitos, o si el cliente solicita un cambio, el costo de volver a una fase anterior es prohibitivamente alto, ya que puede requerir rehacer todo el trabajo posterior. Esta rigidez genera un alto riesgo de que el producto final, entregado muchos meses o años después, ya no satisfaga las necesidades cambiantes del mercado o del cliente.

A pesar de sus limitaciones prácticas en el desarrollo de software moderno, el modelo en Cascada mantiene un valor pedagógico incalculable. Su estructura lógica y secuencial descompone el complejo proceso de creación de software en pasos comprensibles y discretos. Esto lo convierte en el andamiaje conceptual perfecto para que un estudiante o un programador junior construya una comprensión fundamental del flujo de trabajo, desde la idea hasta el mantenimiento. La consulta que motiva este informe es, en sí misma, una prueba de que este modelo lineal es la forma más intuitiva de conceptualizar el proceso por primera vez.

2.2. Flexibilidad y Evolución: Perspectivas Iterativas y Ágiles

La industria del software reconoció las limitaciones de la rigidez del modelo en Cascada y desarrolló nuevos paradigmas para gestionar la incertidumbre y el cambio, que son inherentes a la mayoría de los proyectos de software. Esta evolución no fue una cuestión de moda, sino una respuesta directa a la necesidad de mitigar los riesgos asociados con requisitos volátiles y proyectos de larga duración.

El **Modelo Iterativo e Incremental** fue uno de los primeros pasos hacia una mayor flexibilidad. En lugar de construir todo el sistema de una sola vez, el desarrollo se divide en ciclos más pequeños llamados "iteraciones".[19] Cada iteración pasa por su propio mini-ciclo de vida (análisis, diseño, codificación, pruebas) y produce una versión funcional, aunque incompleta, del software. Este "incremento" del producto se entrega al cliente, quien puede proporcionar retroalimentación. Las iteraciones posteriores refinan las

funcionalidades existentes y añaden nuevas, construyendo el sistema de forma incremental.[2] Este enfoque permite detectar problemas de diseño de manera temprana, adaptar el producto a los cambios en los requisitos y entregar valor al cliente mucho antes en el ciclo de vida del proyecto.

Los **Modelos Ágiles**, como Scrum o la Programación Extrema (XP), llevan esta filosofía de flexibilidad y retroalimentación al siguiente nivel. El desarrollo Ágil es más una mentalidad que un proceso rígido, centrada en la colaboración continua con el cliente, la entrega rápida de software funcional, la respuesta al cambio por encima del seguimiento de un plan y la valoración de los individuos y sus interacciones.[11] En un marco como Scrum, el trabajo se organiza en "sprints", que son iteraciones de corta duración (típicamente de 1 a 4 semanas). Al comienzo de cada sprint, el equipo selecciona un pequeño conjunto de requisitos de una lista priorizada (el Product Backlog) y se compromete a desarrollarlos, probarlos y entregar un incremento de producto potencialmente desplegable al final del sprint.[5]

En los enfoques Ágiles, las fases de "análisis de requisitos" y "diseño" no son eventos únicos que ocurren al principio del proyecto. Son actividades continuas y emergentes. Los requisitos se refinan "justo a tiempo", antes de ser implementados en un sprint, y el diseño evoluciona a medida que el equipo aprende más sobre el sistema y las necesidades del usuario. Esto contrasta fuertemente con el enfoque de "gran diseño por adelantado" del modelo en Cascada, permitiendo una adaptabilidad mucho mayor a la incertidumbre y al cambio. La proliferación de estos modelos adaptativos, como el Modelo en Espiral que introduce explícitamente el análisis de riesgos en cada ciclo [2], demuestra que la historia de las metodologías de software es, en esencia, la historia de la búsqueda de mejores formas de gestionar el riesgo.

Tabla 1: Comparativa de Fases del SDLC en Metodologías Clave

Para visualizar cómo los principios fundamentales del desarrollo de software persisten a través de diferentes metodologías, aunque su implementación varíe drásticamente, la siguiente tabla compara la ejecución de actividades clave en los modelos Cascada, V-Model (una variante de Cascada que enfatiza la relación entre desarrollo y pruebas) y Ágil (Scrum).

Actividad Fundamental	Implementación en Cascada	Implementación en V-Model	Implementación en Ágil (Scrum)
Definición de Requisitos	Fase única y exhaustiva al inicio del proyecto. Produce un SRS completo y "congelado". [12]	Similar a Cascada, pero la fase de "Análisis de Requisitos" se empareja explícitamente con la planificación de las "Pruebas de Aceptación". [29]	Actividad continua. Los requisitos (Historias de Usuario) se gestionan en un Product Backlog. Se detallan justo a tiempo antes de cada Sprint. [29]
Diseño Arquitectónico	Fase única después de los requisitos. Se diseña la arquitectura completa del sistema por adelantado ("Big Design Up Front"). [18]	La fase de "Diseño de Alto Nivel" se empareja con la planificación de las "Pruebas del Sistema". [5, 29]	El diseño es emergente. Se establece una arquitectura inicial simple, que evoluciona y se refina a lo largo de los sprints a medida que se añaden funcionalidades.
Diseño Detallado	Fase única después del diseño arquitectónico. Se detallan todos los módulos antes de la codificación. [12]	La fase de "Diseño de Bajo Nivel" se empareja con la planificación de las "Pruebas de Integración". La "Implementación" se empareja con las "Pruebas Unitarias". [29]	El diseño detallado de una funcionalidad específica ocurre dentro del sprint en el que se implementa, a menudo de forma colaborativa por

			los desarrolladores.
Retroalimentación del Cliente	Ocurre principalmente al final del proyecto, durante las pruebas de aceptación del usuario, cuando los cambios son muy costosos.	Similar a Cascada, la retroalimentación principal es tardía.	Continua y frecuente. Ocurre al final de cada sprint (Sprint Review), permitiendo una rápida corrección del rumbo. [29]

Esta tabla ilustra un punto clave: mientras que la Cascada trata el análisis y el diseño como fases monolíticas y secuenciales, los modelos más modernos los integran como actividades continuas y recurrentes a lo largo de todo el ciclo de vida, priorizando la adaptabilidad y la retroalimentación temprana sobre la planificación exhaustiva inicial.

Sección 3: Profundización Técnica y Estratégica

Comprender la secuencia de las fases es el primer paso. El siguiente es apreciar la profundidad y la disciplina requeridas para ejecutar cada una de ellas con éxito. Esta sección profundiza en los principios y prácticas que sustentan un análisis de requisitos y un diseño de software robustos, elementos que en última instancia determinan la calidad y la longevidad del producto final.

3.1. La Ingeniería de Requisitos: Un Pilar Contra el Fracaso del Proyecto

El análisis de requisitos es mucho más que simplemente tomar notas de lo que el cliente dice que quiere. Es una disciplina de ingeniería por derecho propio, conocida como **Ingeniería de Requisitos**, que abarca un conjunto de procesos para descubrir, analizar, documentar, validar y gestionar los requisitos de un sistema.[13]

Uno de los mayores desafíos es el **problema de la ambigüedad**. Los clientes y usuarios finales suelen tener una idea abstracta o incompleta de sus necesidades, o les resulta difícil articularlas en un lenguaje preciso y sin contradicciones.[11] Un requisito como "el sistema debe ser fácil de usar" es subjetivo e inverificable. El trabajo del ingeniero de requisitos es sondear más profundamente para traducir esa necesidad en criterios objetivos y medibles, como "un nuevo usuario debe ser capaz de completar el registro en menos de 3 minutos sin necesidad de consultar la ayuda". Este proceso de transformar la visión del cliente en especificaciones técnicas inequívocas es fundamental para evitar malentendidos que conducen a la construcción del producto equivocado.[8]

Otro aspecto crítico es la **gestión del cambio**. En casi todos los proyectos, los requisitos evolucionan con el tiempo debido a cambios en el mercado, nuevas regulaciones o una mejor comprensión del problema por parte del cliente. Sin un proceso formal para gestionar estos cambios, el proyecto puede caer víctima del "scope creep", donde se añaden nuevas funcionalidades de forma descontrolada, agotando el presupuesto y retrasando la entrega.[8] Una buena práctica de ingeniería de requisitos incluye establecer una línea base de requisitos aprobada y un proceso de control de cambios que evalúe el impacto de cada nueva solicitud en el cronograma, el costo y la arquitectura del sistema.

3.2. Principios de un Diseño Robusto y Escalable

La fase de diseño es donde se toman las decisiones técnicas que tendrán el impacto más profundo y duradero en la calidad del software. Un buen diseño no solo satisface los requisitos actuales, sino que también anticipa el cambio futuro, dando como resultado un sistema que es fácil de mantener, extender y escalar.

Dos principios fundamentales guían el diseño de software modular: **cohesión** y **acoplamiento**.

- **Alta Cohesión:** Se refiere al grado en que los elementos dentro de un mismo módulo (por ejemplo, las funciones de una clase) están relacionados entre sí y enfocados en una única y bien definida tarea. Un módulo altamente cohesivo es más fácil de entender y mantener.
- **Bajo Acoplamiento:** Se refiere al grado de interdependencia entre diferentes módulos. Un bajo acoplamiento significa que los módulos son independientes entre sí y se comunican a través de interfaces estables y bien definidas. Esto es

deseable porque un cambio en un módulo no debería requerir cambios en cascada en muchos otros módulos.

Para evitar "reinventar la rueda" y beneficiarse de la experiencia colectiva de la industria, los diseñadores a menudo utilizan **patrones de diseño (design patterns)**. Estos son soluciones generales, probadas y reutilizables para problemas de diseño de software que ocurren comúnmente dentro de un contexto dado.[1] Utilizar patrones de diseño no solo acelera la fase de diseño, sino que también conduce a un código más robusto, flexible y mantenible, ya que se basan en principios de diseño sólidos.

Finalmente, una herramienta poderosa para validar y refinar el diseño antes de la codificación es el **prototipado**. Un prototipo es una versión preliminar y a menudo parcial del software que se crea para explorar y validar aspectos del diseño, especialmente la interfaz de usuario (UI) y la experiencia de usuario (UX).[5] Los prototipos pueden variar desde simples bocetos en papel (maquetas o mockups) hasta modelos interactivos de alta fidelidad. Permiten a los usuarios y stakeholders "tocar y sentir" el sistema propuesto, proporcionando una retroalimentación invaluable que es mucho más efectiva que la revisión de documentos de diseño abstractos.[1] Esta retroalimentación temprana ayuda a asegurar que el producto final no solo sea técnicamente sólido, sino también intuitivo y satisfactorio de usar.

Tabla 2: Artefactos Clave de las Fases Pre-Codificación

La siguiente tabla resume el flujo de trabajo de las fases preparatorias, conectando cada etapa con su objetivo principal, actividades clave y el resultado tangible que produce. Sirve como una guía de referencia rápida para entender el proceso desde la concepción hasta los planos finales.

Fase del SDLC	Objetivo Principal	Actividades Clave	Artefacto/Entregable Principal
Planificación y Viabilidad	Determinar si el proyecto es justificable y factible desde	Identificación del problema, análisis de costo-beneficio, estudio	Plan de Proyecto o Estudio de Viabilidad Aprobado.

	una perspectiva de negocio, técnica y económica.	de viabilidad, definición del alcance inicial. [1, 6]	
Análisis de Requisitos	Definir de manera precisa, completa e inequívoca qué debe hacer el sistema para satisfacer las necesidades de los stakeholders.	Obtención de requisitos (entrevistas, talleres), análisis y negociación, clasificación (funcionales/no funcionales), validación. [11, 13, 14]	Documento de Especificación de Requisitos de Software (SRS). [6, 13]
Diseño de Alto Nivel (Arquitectónico)	Definir la estructura general del sistema, sus componentes principales, sus interrelaciones y las tecnologías subyacentes.	Selección de la arquitectura, descomposición en módulos, selección de tecnología, diseño de la base de datos. [12]	Sección de Arquitectura del Documento de Diseño de Software (SDD).
Diseño de Bajo Nivel (Detallado)	Detallar la lógica interna y la implementación de cada componente o módulo definido en el diseño de alto nivel.	Definición de estructuras de datos, diseño de algoritmos, creación de diagramas de flujo/actividad, especificación de interfaces (APIs). [18, 19, 21]	Especificaciones de Módulos, Diagramas de Flujo/UML, y el resto del Documento de Diseño de Software (SDD).

Conclusión: Construyendo Cimientos Sólidos para el Software

El análisis de las fases que preceden a la codificación revela una verdad fundamental en la ingeniería de software: la calidad de un sistema informático no se forja en el teclado del programador, sino en la rigurosidad y la disciplina aplicadas en las etapas preparatorias. La intuición inicial de un proceso estructurado ("enunciado", "diagrama de flujo", "codificación") es la semilla de una comprensión profesional que, una vez formalizada, se convierte en una poderosa herramienta para construir software de manera efectiva y predecible.

Este informe ha desglosado y formalizado este proceso. Se ha establecido que la secuencia profesional y correcta de las fases pre-codificación es: **Planificación y Viabilidad**, seguida por el **Análisis de Requisitos**, que a su vez conduce al **Diseño del Sistema**, para finalmente dar paso a la **Implementación** o codificación.

- El "enunciado" del que se partía se ha identificado y formalizado como la fase de **Análisis de Requisitos**. Esta no es una simple declaración, sino un proceso de ingeniería disciplinado cuyo resultado es el **Documento de Especificación de Requisitos de Software (SRS)**, el pilar sobre el que se construye todo el proyecto.
- El "diagrama de flujo" se ha ubicado correctamente como una herramienta valiosa dentro de la fase de **Diseño**, específicamente en el diseño de bajo nivel, para visualizar y refinar la lógica de los algoritmos antes de su implementación. También se ha contextualizado junto a herramientas de modelado más modernas y completas como UML.

Más allá de la terminología, la conclusión más importante es la comprensión del valor de la metodología. Invertir tiempo y esfuerzo en la planificación, el análisis y el diseño no es un retraso en el inicio de la "verdadera" tarea de programar; es la actividad que más contribuye a evitar sobrecostos, incumplimiento de plazos y, en última instancia, el fracaso del proyecto. Un proceso bien ejecutado, que transforma sistemáticamente una idea abstracta en un conjunto de planos técnicos detallados, es el mejor predictor del éxito.

Para continuar el desarrollo profesional, es esencial seguir explorando los diferentes modelos del ciclo de vida del software. Comprender cuándo aplicar un enfoque secuencial como la Cascada para proyectos simples y predecibles, y cuándo adoptar la flexibilidad y adaptabilidad de las metodologías Ágiles para entornos complejos y

cambiantes, es lo que distingue a un programador competente de un ingeniero de software completo. La construcción de software es, en esencia, la construcción de cimientos sólidos, y estos cimientos se establecen mucho antes de que se escriba la primera línea de código.

Works cited

1. Ciclo de vida de desarrollo de software | Microsoft Power Automate, accessed October 30, 2025, https://www.microsoft.com/es-co/power-platform/topics/phases-of-the-software-development-lifecycle
2. Ciclo de vida del software: todo lo que necesitas saber - Intelequia, accessed October 30, 2025, https://intelequia.com/es/blog/post/ciclo-de-vida-del-software-todo-lo-que-necesitas-saber
3. ¿Qué es el ciclo de vida del desarrollo de software (SDLC)? - Amazon AWS, accessed October 30, 2025, https://aws.amazon.com/es/what-is/sdlc/
4. Aprende qué es Desarrollo de Software y sus etapas (Clase fácil) - YouTube, accessed October 30, 2025, https://www.youtube.com/watch?v=s5ABwHaN7as
5. 5 pasos de cualquier desarrollo de software - Startechup, accessed October 30, 2025, https://www.startechup.com/es/blog/5-steps-of-software-development/
6. ¿Qué es el ciclo de vida del desarrollo de software (SDLC)? - IBM, accessed October 30, 2025, https://www.ibm.com/es-es/think/topics/sdlc
7. ¿Cómo se hace un análisis de requisitos de software? - TIC Portal, accessed October 30, 2025, https://www.ticportal.es/glosario-tic/analisis-requisitos-software
8. La importancia del análisis de requerimientos en el desarrollo de software - Proefex, accessed October 30, 2025, https://proefexperu.com/blog/la-importancia-del-analisis-de-requerimientos-en-el-desarrollo-de-software
9. 7 Etapas del ciclo de desarrollo de software - Bambu Mobile, accessed October 30, 2025, https://bambu-mobile.com/ciclo-de-desarrollo-de-software/
10. Diseño de software: fases y modelos - W&B Asset Studio, accessed October 30, 2025, https://www.wbassetstudio.com/blog/diseno-de-software-fases-y-modelos/
11. Proceso para el desarrollo de software - Wikipedia, la enciclopedia libre, accessed October 30, 2025, https://es.wikipedia.org/wiki/Proceso_para_el_desarrollo_de_software

12. Desarrollo en cascada - Wikipedia, la enciclopedia libre, accessed October 30, 2025, https://es.wikipedia.org/wiki/Desarrollo_en_cascada
13. gc.scalahed.com, accessed October 30, 2025, https://gc.scalahed.com/recursos/files/r161r/w25870w/Analisis_de_requisitos_del_software.pdf
14. Análisis de requisitos: Pasos clave y técnicas para hacerlo bien - ClickUp, accessed October 30, 2025, https://clickup.com/es-ES/blog/138879/analisis-de-requisitos
15. Tema N° 10 Análisis de los Requisitos | PDF - Slideshare, accessed October 30, 2025, https://es.slideshare.net/slideshow/tema-n-10-anlisis-de-los-requisitos/249771777
16. Ciclo de Vida del Software: Fases y Estrategias para un Desarrollo Eficiente | Lisit, accessed October 30, 2025, https://lisit.cl/novedades/ciclo-de-vida-del-software-fases-y-estrategias-para-un-desarrollo-eficiente/
17. www.ibm.com, accessed October 30, 2025, https://www.ibm.com/es-es/think/topics/sdlc#:~:text=La%20fase%20de%20dise%C3%B1o%20implica,la%20base%20de%20datos%2C%20etc.
18. Qué es la metodología waterfall y cuándo utilizarla [2025] - Asana, accessed October 30, 2025, https://asana.com/es/resources/waterfall-project-management-methodology
19. Ciclo de Vida del Software o SDLC - Fases y Modelos - Isvisoft, accessed October 30, 2025, https://isvisoft.com/ciclo-vida-software-sdlc/
20. www.uv.mx, accessed October 30, 2025, https://www.uv.mx/personal/aherrera/files/2020/05/DIAGRAMAS-DE-FLUJO.pdf
21. Diagramas de Flujo - Algoritmos y Programación, accessed October 30, 2025, https://lab.anahuac.mx/~hselley/ayp/diagramasDeFlujo.html
22. Diagrama de flujo - Wikipedia, la enciclopedia libre, accessed October 30, 2025, https://es.wikipedia.org/wiki/Diagrama_de_flujo
23. ¿Qué es un diagrama de flujo y cómo hacerlo? - Lucidchart, accessed October 30, 2025, https://www.lucidchart.com/pages/es/que-es-un-diagrama-de-flujo
24. Diagramas de flujo - Programación - Picuino, accessed October 30, 2025, https://www.picuino.com/es/prog-flowchart.html

25. www.pipedrive.com, accessed October 30, 2025, https://www.pipedrive.com/es/blog/metodologia-de-cascada#:~:text=La%20metodolog%C3%ADa%20de%20cascada%20se%20desarrolla%20en%20cinco%20fases%20diferentes,%2C%20implementaci%C3%B3n%2C%20verificaci%C3%B3n%20y%20mantenimiento.
26. Modelo de cascada en la gestión de proyectos | Blog Lucidchart, accessed October 30, 2025, https://www.lucidchart.com/blog/es/metodologia-gestion-proyectos-cascada
27. Modelos de Desarrollo de Software - Cómo citar el artículo Número completo Más información del artículo Página de la revista en redalyc.org Sistema de Inform, accessed October 30, 2025, https://www.redalyc.org/journal/3783/378366538003/378366538003.pdf
28. Metodología Cascada | PDF | Ingeniería de software - Scribd, accessed October 30, 2025, https://es.scribd.com/document/501968111/Metodologia-Cascada
29. El Ciclo de Vida del Software | Proceso Básico en Metodologías - ok hosting, accessed October 30, 2025, https://okhosting.com/blog/el-ciclo-de-vida-del-software/