Licenciatura en Sistemas - Ingeniería de Software I

Resumen de Bibliografía.

Equipo docente: Lic. Romina Mansilla

Lic. Leandro Ríos

Proceso de Software y Ciclo de Vida

PROCESO DE SOFTWARE

1. Ventajas de definir un proceso de software

Un proyecto sin estructura es un proyecto inmanejable. No puede ser planificado, ni estimado, ni su progreso ser controlado, y mucho menos alcanzar un compromiso de costes o tiempos. La idea originaria de buscar ciclos de vida que describan los estados por los que pasa el producto, o procesos software que describan las actividades a realizar para transformar el producto, surge de la necesidad de tener un esquema que sirva como base para planificar, organizar, asignar personal, coordinar, presupuestar, y dirigir las actividades de la construcción de software. De hecho, muchos proyectos han terminado mal porque las fases de desarrollo se realizaron en un orden erróneo.

Por tanto, al comienzo de un proyecto software, se debe elegir el Ciclo de Vida que seguirá el producto a construir, en base a las consideraciones del apartado anterior. El Modelo de Ciclo de Vida elegido llevará a encadenar las tareas y actividades del Proceso Software de una determinada manera: algunas tareas no será necesario realizarlas, otras deberán realizarse más de una vez, etc. Una vez conseguido un Proceso Software concreto para el proyecto en cuestión, se está preparado para planificar los plazos del proyecto, asignar personas a las distintas tareas, presupuestar los costos del proyecto, etc.

Concretamente, los procesos software se usan como:

- Guía prescriptiva sobre la documentación a producir para enviar al cliente.
- Base para determinar qué herramientas, técnicas y metodologías de Ingeniería de Software serán más apropiadas para soportar las diferentes actividades.
- Marco para analizar o estimar patrones de asignación y consumo de recursos a lo largo de todo el ciclo de vida del producto software.
- Base para llevar a cabo estudios empíricos para determinar qué afecta a la productividad, el coste y la calidad del software.
- Descripción comparativa sobre cómo los sistemas software llegan a ser lo que son.

2. Standard IEEE sobre proceso de software

A continuación, se describen las fases o subprocesos que conforman el Proceso Base de Construcción del Software del estándar IEEE 1074 [IEEE 1074, 1989]. Dicho estándar determina el conjunto de actividades esenciales, no ordenadas en el tiempo, que deben ser incorporadas dentro de un Modelo de Ciclo de Vida del producto software. El Proceso Base de Construcción de Software consiste en analizar las necesidades de la organización en un dominio, desarrollar una solución que las satisfaga y posteriormente reinsertar la solución en el dominio, bajo un marco de gestión, seguimiento, control y gestión de la calidad.

El proceso software está compuesto, a su vez, de cuatro procesos principales cada uno de los cuales agrupa una serie de actividades que se encargan de la realización de sus requisitos asociados. Estos procesos son los siguientes:

- **Proceso de Selección de un Modelo de Ciclo de Vida del Producto:** identifica y selecciona un ciclo de vida para el software que se va a construir.
- **Procesos de Gestión del Proyecto:** crean la estructura del proyecto y aseguran el nivel apropiado de la gestión del mismo durante todo el ciclo de vida del software.
- **Procesos Orientados al Desarrollo del Software**: producen, instalan, operan y mantienen el software y lo retiran de su uso. Se clasifican en procesos de pre-desarrollo, desarrollo y post-desarrollo.
- **Procesos Integrales del Proyecto:** son necesarios para completar con éxito las actividades del proyecto software. Aseguran la terminación y calidad de las funciones del mismo. Son simultáneos a los procesos orientados al desarrollo del software e incluyen actividades de no desarrollo.

2.1 Proceso de Selección de un Modelo de Ciclo de Vida del Producto

Durante este proceso se selecciona un ciclo de vida que establece el orden de ejecución de las distintas actividades marcadas por el proceso e involucradas en el proyecto. Como veremos más adelante, en base

al tipo de producto software a desarrollar y a los requisitos del proyecto se identifican y analizan posibles modelos de ciclo de vida para dicho proyecto y se selecciona un único modelo que lo soporte adecuadamente. El estándar no dictamina ni define un ciclo de vida del software específico, ni una metodología de desarrollo, sólo requiere elegir y seguir un modelo de ciclo de vida.

2.2 Procesos de Gestión del Proyecto

La gestión del proyecto presupone establecer condiciones para el desarrollo del mismo. Involucra actividades de planificación, estimación de recursos, seguimiento y control, y evaluación del proyecto.

La planificación de proyectos se define como la predicción de la duración de las actividades y tareas a nivel individual, los recursos requeridos, la concurrencia y solapamiento de tareas para ser desarrollados en paralelo y el camino crítico a través de la red de actividades.

La estimación se define como la predicción de personal, esfuerzo y costos que se requerirá para terminar todas las actividades y productos conocidos asociados con el proyecto.

La determinación del tamaño del producto a desarrollar es unsa de las primeras tareas en la gestión del proyecto, ya que sin unos conocimientos razonables, es imposible planificar o estimar el esfuerzo involucrado. El tamaño se define como la cantidad de código fuente, especificaciones, casos de prueba, documentación del usuario y otros productos tangibles que son salida del proyecto. La determinación del tamaño se basa principalmente en la experiencia de proyectos anteriores.

El seguimiento de proyectos es la recolección de datos y su acumulación sobre recursos consumidos, costos generados, e hitos asociados con un proyecto. Medir en un proyecto se define como el registro de todos los productos generados en el mismo, de todos los recursos requeridos, planificación y solapamiento de todas las actividades y tareas y de todos los factores que impactan en el proyecto (conocimientos, métodos, herramientas, lenguajes, limitaciones, problemas y el entorno físico).

La medición en los proyectos de desarrollo de software es una actividad fundamental para la mejora de la productividad, el costo y la calidad del producto final. La medición, entrada a estudios empíricos, es el único modo para detectar fallos en el proceso de construcción de software.

Incluye tres subprocesos:

1. *Proceso de iniciación del proyecto* (asignar los recursos, definir el entorno)
2. *Proceso de seguimiento y control del proyecto* (analizar riesgos, realizar planificación de contingencias)
3. *Proceso de gestión de la calidad del software* (desarrollar métricas de calidad, identificar necesidades de mejora)

2.3 Procesos Orientados al Desarrollo del Software

Pre - Desarrollo

Son los procesos que se deben realizar antes de que comience el desarrollo propiamente dicho del software. El esfuerzo de desarrollo se inicia con la identificación de una necesidad de automatización. Esta necesidad, para ser satisfecha, puede requerir una nueva aplicación, o un cambio de todo o parte de una aplicación existente. El pre-desarrollo abarca desde el reconocimiento del problema hasta la determinación de los requisitos funcionales a nivel de sistema, pasando por el estudio de la viabilidad de su solución automatizada.

Incluye dos subprocesos:

1. *Proceso de exploración de conceptos* (identificar ideas o necesidades, formular soluciones potenciales, conducir estudios de viabilidad, planificar la transición del sistema si aplica)
2. *Proceso asignación del sistema* (se realiza cuando el sistema requiere tanto del desarrollo de hardware como de software, o cuando no se puede asegurar que sólo se necesita desarrollo de software)

Desarrollo

Son los procesos que se deben realizar para la construcción del producto software. Estos definirán qué información obtener y cómo estructurar los datos, qué algoritmos usar para procesar los datos y cómo implementarlos y qué interfaces desarrollar para operar con el software y cómo hacerlo.

Incluye tres subprocesos:

1. *Proceso de requisitos* (definir y desarrollar los requisitos de software y los de interfaz)
2. *Proceso de diseño* (realizar el diseño arquitectónico, analizar el flujo de datos, diseñar la base de datos, las interfaces, el diseño detallado)
3. *Proceso de implementación* (generar el código fuente, los datos de prueba, la documentación)

Post - Desarrollo

Son los procesos que se deben realizar para instalar, operar, soportar, mantener y retirar un producto software. Se realizan después de la construcción del software. Es decir, se aplican a las últimas fases del ciclo de vida del software.

Una vez terminada la prueba del software, éste está casi preparado para ser entregado a los usuarios finales. Sin embargo, antes de la entrega se llevan a cabo una serie de actividades de garantía de calidad para asegurar que se han generado y catalogado los registros y documentos internos adecuados, que se ha desarrollado una documentación de alta calidad para el usuario y que se han establecido los mecanismos apropiados de control de configuraciones. Entonces, el software ya puede ser distribuido a los usuarios finales.

Tan pronto como se entrega el software a los usuarios finales, el trabajo del ingeniero del software cambia. En ese momento, el enfoque pasa de la construcción al mantenimiento; corrección de errores, adaptación al entorno y mejora de la funcionalidad. En todos los casos, la modificación del software no sólo afecta al código, sino también a la configuración entera; es decir, todos los documentos, datos y programas desarrollados en la fase de planificación y desarrollo.

Incluye cuatro subprocesos:

1. *Proceso de instalación* (transporte y la instalación de un sistema software desde el entorno de desarrollo al entorno de destino)
2. *Proceso de operación y soporte* (operar el sistema. proveer de asistencia técnica y consultas, mantener el histórico de peticiones de soporte)
3. *Proceso de mantenimiento* (se interesa por los errores, defectos, fallos, mejoras y cambios del software)
4. *Proceso de retiro* (es la jubilación de un sistema existente de su soporte activo o de su uso mediante el cese de su operación o soporte, o mediante su reemplazo tanto por un nuevo sistema como por una versión actualizada)

2.4 Procesos Integrales del Proyecto

Son procesos simultáneos y complementarios a los procesos orientados al desarrollo. Incluyen actividades imprescindibles para que el sistema construido sea fiable (procesos de verificación y validación, gestión de la configuración) y sea utilizado al Son procesos simultáneos y complementarios a los procesos orientados al desarrollo.

Los Procesos Integrales comprenden dos tipos de actividades:

- aquellas que se realizan discretamente y se aplican dentro de un ciclo de vida del software, y
- aquellas que se realizan para completar otra actividad. Estas son actividades que se invocan (llamadas como a una subrutina) y no se aplican dentro del modelo de ciclo de vida del software para cada instancia.máximo de sus capacidades (procesos de formación, documentación).

Incluye cuatro subprocesos:

1. *Proceso de verificación y validación* (planificar, desarrollar y ejecutar las pruebas)
2. *Proceso de gestión de la configuración* (su objetivo es el control de los cambios en el sistema, mantener su coherencia y su "rastreabilidad" o "trazabilidad", y poder realizar auditorías de control sobre la evolución de las configuraciones)
3. *Proceso de desarrollo de documentación* (planificar, implementar y distribuirla)
4. *Proceso de formación* (de desarrolladores, personal de soporte técnico y clientes)

3. Tabla Resumen de IEEE

Adjunta al final del documento.

CICLOS DE VIDA

1. Introducción

No existe un único Modelo de Ciclo de Vida que defina los estados por los que pasa cualquier producto software. Dado que existe una gran variedad de aplicaciones para las que se construyen productos software y que dicha variedad supone situaciones totalmente distintas, es natural que existan diferentes Modelos de Ciclo de Vida. El ciclo de vida apropiado se elige en base a:

- la cultura de la corporación,
- el deseo de asumir riesgos,
- el área de aplicación,
- la volatilidad de los requisitos,
- y hasta qué punto se entienden bien dichos requisitos.

El ciclo de vida elegido ayuda a relacionar las tareas que forman el proceso software de cada proyecto. A continuación, se presentan los diferentes modelos de ciclo de vida más representativos, tanto entre las metodologías tradicionales como las ágiles. Las metodologías tradicionales (o predictivas), encaran las fases que componen el ciclo de vida del desarrollo de Software, de manera sucesiva. Es decir, que una fase sucede a otra, y cada fase, ocupa un espacio lineal en el. En cambio, las metodologías ágiles, solapan estas etapas, permitiendo ahorrar tiempo, evitando la dependencia (cada etapa es independiente de la otra) y haciendo del ciclo de vida, un proceso iterativo (se inicia con el relevamiento, se finaliza con la implementación y se vuelve a comenzar para abordar nuevas funcionalidades).

2. Metodologías tradicionales

2.1. Cascada

Este modelo fue presentado por primera vez por Royce en 1970. Se representa, frecuentemente, como un simple modelo con forma de cascada de las etapas del software, como muestra la Figura 2.1. En este modelo la evolución del producto software procede a través de una secuencia ordenada de transiciones de una fase a la siguiente según un orden lineal. Tales modelos semejan una máquina de estados finitos para la descripción de la evolución del producto software. El modelo en cascada ha sido útil para ayudar a estructurar y gestionar grandes proyectos de desarrollo de software dentro de las organizaciones.

Este modelo permite iteraciones durante el desarrollo, ya sea dentro de un mismo estado, ya sea de un estado hacia otro anterior, como muestran las flechas ascendentes de la Figura 2.1. La mayor iteración se produce cuando una vez terminado el desarrollo y cuando se ha visto el software producido, se decide comenzar de nuevo y redefinir los requisitos del usuario.

El uso del modelo en cascada:

- Obliga a especificar lo que el sistema debe hacer (o sea, definir los requisitos) antes de construir el sistema (esto es, diseñarlo).
- Obliga a definir cómo van a interactuar los componentes (o sea, diseñar) antes de construir tales componentes (o sea codificar).

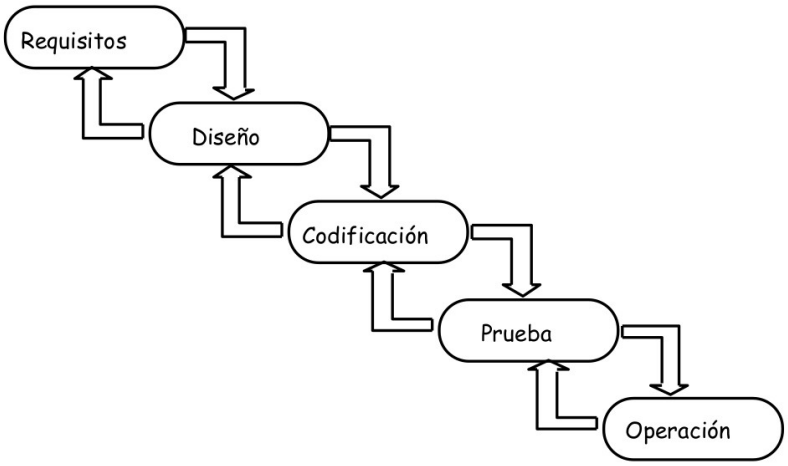


Figura 2.1. Ciclo de vida en cascada

- Permite al líder del proyecto seguir y controlar los progresos de un modo más exacto. Esto le permite detectar y resolver las desviaciones sobre la planificación inicial.
- Requiere que el proceso de desarrollo genere una serie de documentos que posteriormente pueden utilizarse para la validación y el mantenimiento del sistema.

A menudo, durante el desarrollo, se pueden tomar decisiones que den lugar a diferentes alternativas, el modelo en cascada no reconoce esta situación. Por ejemplo, dependiendo del análisis de requisitos se puede implementar el sistema desde cero, o adoptar uno ya existente, o comprar un paquete que proporcione las funcionalidades requeridas. Es decir, resulta demasiado estricto para la flexibilidad que necesitan algunos desarrollos.

El modelo en cascada asume que los requisitos de un sistema pueden ser congelados antes de comenzar el diseño.

La figura 2.2. muestra la constante evolución de las necesidades del usuario. Se han representado en un espacio de tiempo/funcionalidad: según pasa el tiempo, aumentan las expectativas de funcionalidades que el usuario espera que tenga el sistema. La evolución está simplificada, pues ni mucho menos es lineal ni continua, pero para hacerse una idea es suficiente.

El ciclo de vida en cascada tiene tres propiedades muy positivas:

- Las etapas están organizadas de un modo lógico. Es decir, si una etapa no puede llevarse a cabo hasta que se hayan tomado ciertas decisiones de más alto nivel, debe esperar hasta que esas decisiones estén tomadas. Así, el diseño espera a los requisitos, el código espera a que el diseño esté terminado, etc.

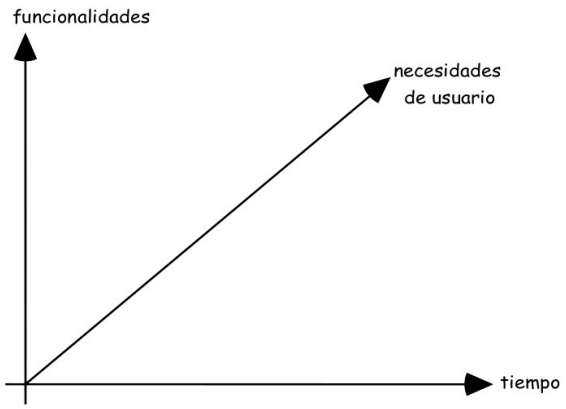


Figura 2.2. Evolución constante de las necesidades del usuario

- Cada etapa incluye cierto proceso de revisión, y se necesita una aceptación del producto antes de que la salida de la etapa pueda usarse. Este ciclo de vida está organizado de modo que se pase el menor número de errores de una etapa a la siguiente.
- El ciclo es iterativo. A pesar de que el flujo básico es de arriba hacia abajo, el ciclo de vida en cascada reconoce, como ya se ha comentado, que los problemas encontrados en etapas inferiores afectan a las decisiones de las etapas superiores.

Existe una visión alternativa del modelo de ciclo de vida en cascada, mostrada en la figura 2.3, que enfatiza en la validación de los productos, y de algún modo en el proceso de composición existente en la construcción de sistemas software.

El proceso de análisis o descomposición subyacente en la línea superior del modelo de la figura 2.3 consiste en: los requisitos del sistema global se dividen en requisitos del hardware y requisitos del software. Estos últimos llevan al diseño preliminar de múltiples funciones, cada una de las cuales se expande en el diseño detallado, que, a su vez, evoluciona aún en un número mayor de programas unitarios. Sin embargo, el ensamblaje del producto final fluye justo en sentido contrario, dentro de un proceso de síntesis o composición. Primero se aceptan los programas unitarios probados. Entonces, éstos se agrupan en módulos que, a su vez, deben ser aceptados una vez probados. Los módulos se agrupan para certificar que el grupo formado por todos ellos incluyen todas las funcionalidades deseadas. Finalmente, el software es integrado con el hardware hasta formar un único sistema informático que satisface los requisitos globales.

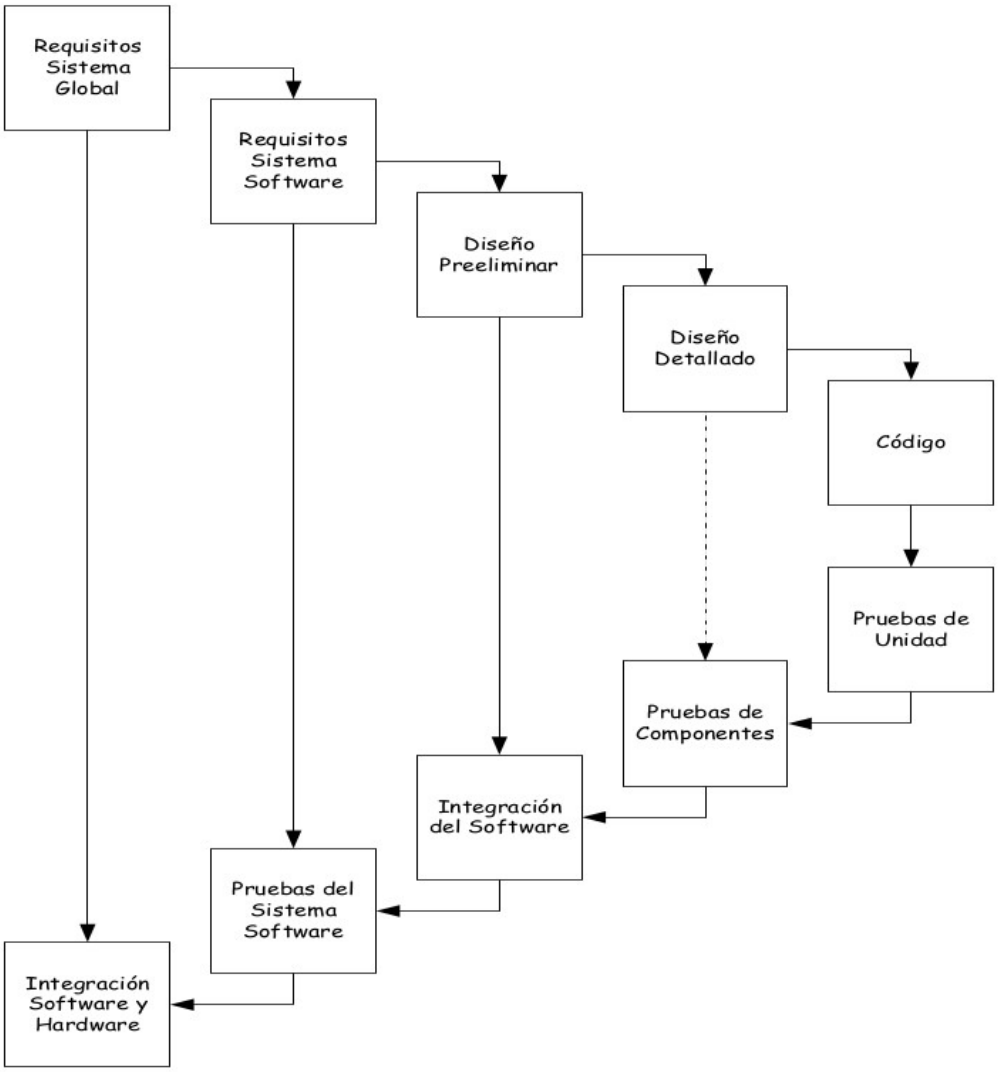


Figura 2.3. Visión alternativa del ciclo de vida en cascada

2.2. Prototipado

Suele ocurrir que, en los proyectos software, el cliente no tiene una idea muy detallada de lo que necesita, o que el ingeniero de software no está muy seguro de la viabilidad de la solución que tiene en mente. En estas condiciones, la mejor aproximación al problema es la realización de un prototipo.

El modelo de desarrollo basado en prototipos tiene como objetivo contrarestar el problema ya comentado del modelo en cascada: la congelación de requisitos mal comprendidos. La idea básica es que el prototipo ayude a comprender los requisitos del usuario. El prototipo debe incorporar un subconjunto de la función requerida al software, de manera que se puedan apreciar mejor las características y posibles problemas.

La construcción del prototipo sigue el ciclo de vida estándar, sólo que su tiempo de desarrollo será bastante más reducido, y no será muy rigurosa la aplicación de los estándares.

El problema del prototipo es la elección de las funciones que se desean incorporar, y cuáles son las que hay que dejar fuera, pues se corre el riesgo de incorporar características secundarias, y dejar de lado alguna característica importante. Una vez creado el prototipo, se le enseña al cliente, para que “juegue” con él durante un período de tiempo, y a partir de la experiencia aportar nuevas ideas, detectar fallos, etc.

Cuando se acaba la fase de análisis del prototipo, se refinan los requisitos del software, y a continuación se procede al comienzo del desarrollo a escala real. En realidad, el desarrollo principal se puede haber arrancado previamente, y avanzar en paralelo, esperando para un tirón definitivo a la revisión del prototipo.

El ciclo de vida clásico queda modificado de la siguiente manera por la introducción del uso de prototipos:

1. Análisis preliminar y especificación de requisitos
2. Diseño, desarrollo e implementación del prototipo
3. Prueba del prototipo
4. Refinamiento iterativo del prototipo
5. Refinamiento de las especificaciones de requisitos
6. Diseño e implementación del sistema final

Existen tres modelos derivados del uso de prototipos:

- *Maqueta*. Aporta al usuario ejemplo visual de entradas y salidas. La diferencia con el anterior es que en los prototipos desechables se utilizan datos reales, mientras que las maquetas son formatos encadenados de entrada y salida con datos simples estáticos.
- *Prototipo desechable*. Se usa para ayudar al cliente a identificar los requisitos de un nuevo sistema. En el prototipo se implantan sólo aquellos aspectos del sistema que se entienden mal o son desconocidos. El usuario, mediante el uso del prototipo, descubrirá esos aspectos o requisitos no captados. Todos los elementos del prototipo serán posteriormente desechados.
- *Prototipo evolutivo*. Es un modelo de trabajo del sistema propuesto, fácilmente modificable y ampliable, que aporta a los usuarios una representación física de las partes claves del sistema antes de la implantación. Una vez definidos todos los requisitos, el prototipo evolucionará hacia el sistema final. En los prototipos evolutivos, se implantan aquellos requisitos y necesidades que son claramente entendidos, utilizando diseño y análisis en detalle así como datos reales.

2.3. Espiral

El modelo en espiral para el desarrollo de software representa un enfoque dirigido por el riesgo para el análisis y estructuración del proceso software. Fue presentado por primera vez por Böehm en 1986. El enfoque incorpora métodos de proceso dirigidos por las especificaciones y por los prototipos. Esto se lleva a cabo representando ciclos de desarrollo iterativos en forma de espiral, denotando los ciclos internos del ciclo de vida análisis y prototipado precoz, y los externos, el modelo clásico. La dimensión radial indica los costes de desarrollo acumulativos y la angular el progreso hecho en cumplimentar cada desarrollo en espiral. El análisis de riesgos, que busca identificar situaciones que pueden causar el fracaso o sobrepasar el presupuesto o plazo, aparecen durante cada ciclo de la espiral. En cada ciclo, el análisis del riesgo representa groseramente la misma cantidad de desplazamiento angular, mientras que el volumen

desplazado barrido denota crecimiento de los niveles de esfuerzo requeridos para el análisis del riesgo como se ve en la figura 2.4.

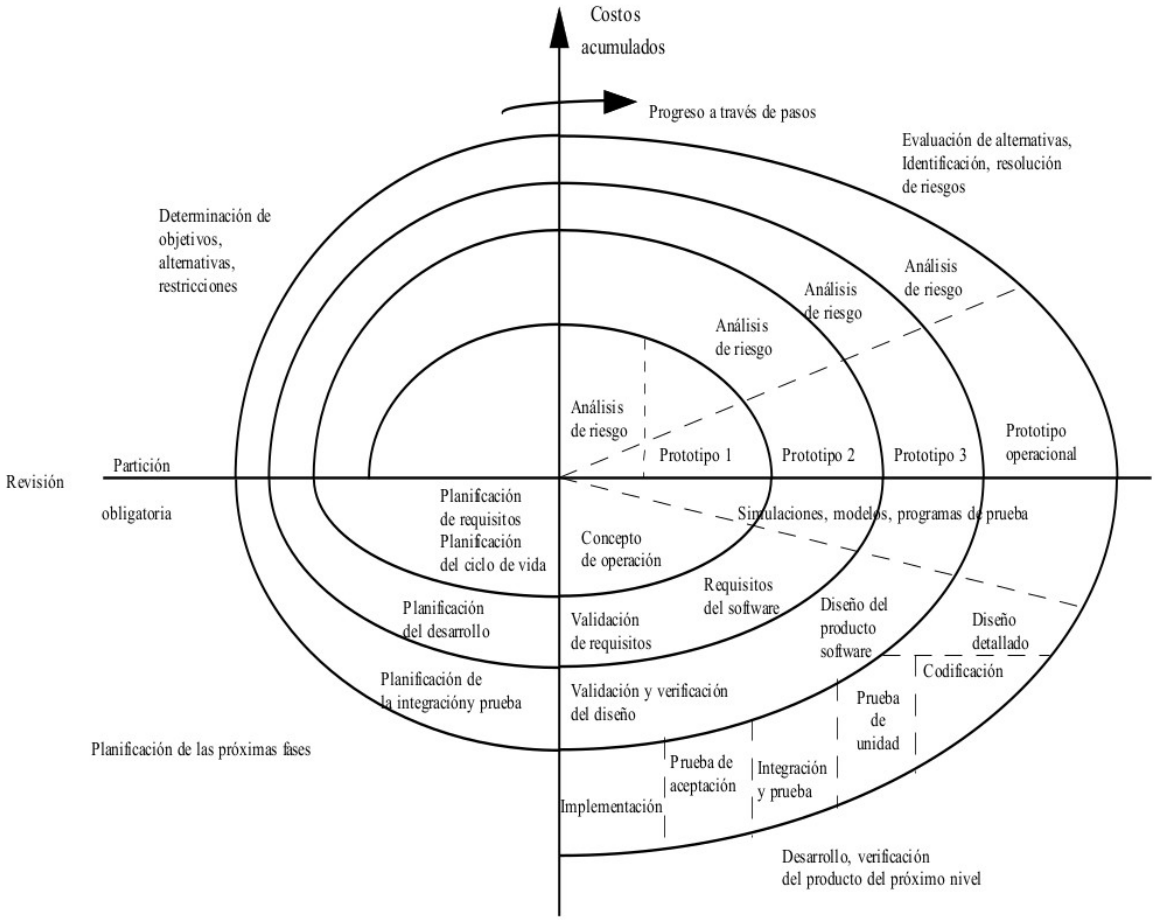


Figura 2.4. Modelo en Espiral

La primera ventaja del modelo en espiral es que su rango de opciones permiten utilizar los modelos de proceso de construcción de software tradicionales, mientras su orientación al riesgo evita muchas dificultades. De hecho, en situaciones apropiadas, el modelo en espiral proporciona una combinación de los modelos existentes para un proyecto dado. Otras ventajas son:

- Se presta atención a las opciones que permiten la reutilización de software existente.
- Se centra en la eliminación de errores y alternativas poco atractivas.
- No establece una diferenciación entre desarrollo de software y mantenimiento del sistema.
- Proporciona un marco estable para desarrollos integrados hardware- software.

En realidad el modelo en espiral no significa una visión radicalmente distinta de los modelos tradicionales, o prototipado. Cualquiera de los modelos pueden verse con una representación espiral. Este modelo de Böehm es más bien una formalización o representación de los modelos de ciclo de vida más acertada que la representación en forma de cascada, pues permite observar mejor todos los elementos del proceso (incluido riesgos, objetivos, etc.).

3. Metodologías ágiles

El desarrollo ágil de software, no es más que una metodología de gestión de proyectos adaptativa, que permite llevar a cabo, proyectos de desarrollo de software, adaptándose a los cambios y evolucionando en forma conjunta con el software.

3.1. Scrum

Esta metodología ayuda a las personas, equipos y organizaciones a generar valor a través de soluciones adaptables para problemas complejos.

> *"[...] es un marco de trabajo iterativo e incremental para el desarrollo de proyectos, productos y aplicaciones. Estructura el desarrollo en ciclos de trabajo llamados Sprints. Son iteraciones de 1 a 4 semanas, y se van sucediendo una detrás de otra. Los Sprints son de duración fija – terminan en una fecha específica aunque no se haya terminado el trabajo, y nunca se alargan. Se limitan en el tiempo. Al comienzo de cada Sprint, un equipo multi-funcional selecciona los elementos (requisitos del cliente) de una lista priorizada. Se comprometen a terminar los elementos al final del Sprint. Durante el Sprint no se pueden cambiar los elementos elegidos. [...] Todos los días el equipo se reúne brevemente para informar del progreso, y actualizan unas gráficas sencillas que les orientan sobre el trabajo restante. Al final del Sprint, el equipo revisa el Sprint con los interesados en el proyecto, y les enseña lo que han construido. La gente obtiene comentarios y observaciones que se pueden incorporar al siguiente Sprint. Scrum pone el énfasis en productos que funcionen al final del Sprint que realmente estén "hechos"; en el caso del software significa que el código esté integrado, completamente probado y potencialmente para entregar [...]" (The Scrum Primer, 2009, pág. 5)*

Scrum es simple. Es deliberadamente incompleto, solo define las partes necesarias para implementar la teoría de Scrum y se basa en la inteligencia colectiva de las personas que lo utilizan. En lugar de proporcionar a las personas instrucciones detalladas, las reglas de Scrum guían sus relaciones e interacciones.

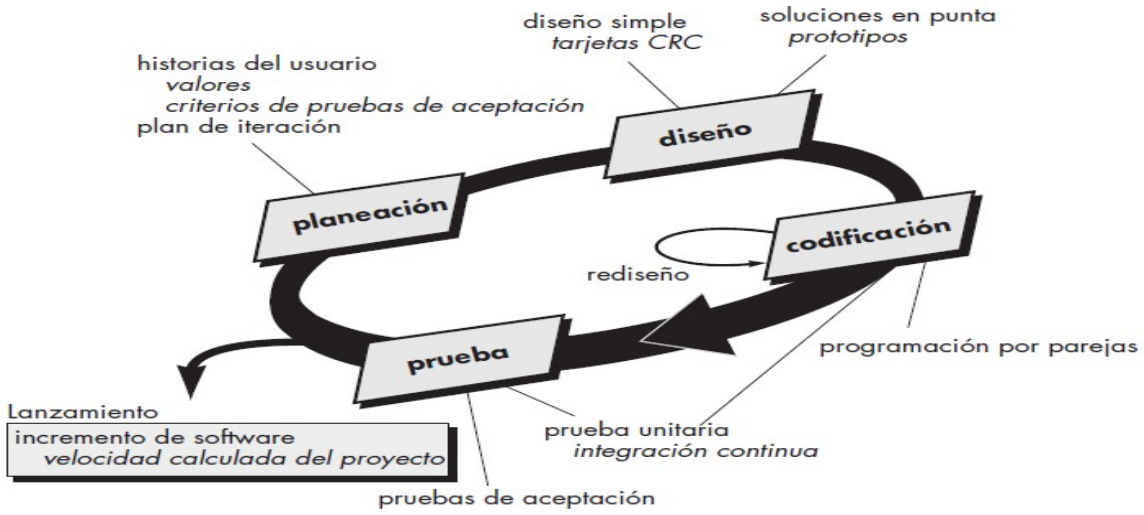
Scrum combina cuatro eventos formales (Planning, Daily, Review y Retrospective) para la inspección y adaptación dentro de un evento contenedor, el Sprint. Estos eventos funcionan porque implementan los pilares empíricos de Scrum:

- **Transparencia**: El proceso y el trabajo emergentes deben ser visibles para aquellos que realizan el trabajo, así como para los que reciben el trabajo. Con Scrum, las decisiones importantes se basan en el estado percibido de sus tres artefactos formales. Los artefactos que tienen poca transparencia pueden conducir a decisiones que disminuyen el valor y aumentan el riesgo. La transparencia permite la inspección. La inspección sin transparencia genera engaños y desperdicios.
- **Inspección**: Los artefactos de Scrum y el progreso hacia objetivos acordados deben ser inspeccionados con frecuencia y diligentemente para detectar varianzas o problemas potencialmente indeseables. Para ayudar con la inspección, Scrum proporciona cadencia en forma de cinco eventos. La inspección permite la adaptación. La inspección sin adaptación se considera inútil. Los eventos de Scrum están diseñados para provocar cambios.
- **Adaptación**: Si algún aspecto de un proceso se desvía fuera de los límites aceptables o si el producto resultante es inaceptable, el proceso que se está aplicando o los materiales que se producen deben ajustarse. El ajuste debe realizarse lo antes posible para minimizar la desviación adicional. La adaptación se vuelve más difícil cuando las personas involucradas no están empoderadas o no poseen capacidad para autogestionarse. Se espera que un equipo de Scrum se adapte en el momento en que aprenda algo nuevo por medio de la inspección.

3.2. eXtreme Programming

eXtreme Programming (programación extrema) también llamado XP, es una metodología que tiene su origen en 1996, de la mano de Kent Beck, Ward Cunningham y Ron Jeffries. A diferencia de Scrum, XP propone solo un conjunto de prácticas técnicas, que aplicadas de manera simultánea, pretenden enfatizar los efectos positivos de en un proyecto de desarrollo de Software.

La programación extrema usa un enfoque orientado a objetos como paradigma preferido de desarrollo, y engloba un conjunto de reglas y prácticas que ocurren en el contexto de cuatro actividades estructurales: planeación, diseño, codificación y pruebas.



Además, se apoya en cinco valores, los cuales enfatizan la esencia colaborativa del equipo.

Valores

1. ***Comunicación:*** En XP, todo es trabajado en equipo: desde el relevamiento y análisis hasta el código fuente desarrollado. Todo se conversa cara a cara, procurando hallar soluciones en conjunto a los problemas que puedan surgir.

2. ***Simplicidad:*** Se pretende desarrollar solo lo necesario y no perder tiempo en detalles que no sean requeridos en el momento. En este aspecto, se asemeja a otra metodología ágil, denominada Kanban, en la cual, un proceso "anterior" solo produce lo que el proceso posterior demanda.
3. ***Retroalimentación:*** El objetivo de eXtreme Programming es entregar lo necesario al cliente, en el menor tiempo posible. A cambio, demanda al cliente, un feedback continuo -retroalimentación-, a fin de conocer sus requerimientos e implementar los cambios tan pronto como sea posible.
4. **Respeto:** El equipo respeta la idoneidad del cliente como tal (sólo éste, es quien conoce el valor para el negocio) y el cliente, a la vez, respeta la idoneidad del equipo (confiando en ellos profesionalmente para definir y decidir el "cómo" se llevará a cabo el desarrollo de lo requerido).
5. ***Coraje:*** Se dice que en XP un equipo debe tener el valor para decir la verdad sobre el avance del proyecto y las estimaciones del mismo, planificando el éxito en vez de buscar excusas sobre los errores.

Prácticas Técnicas

1. ***CLIENTE IN-SITU (ON-SITE CUSTOMER):*** En XP se requiere que el cliente esté dispuesto a participar activamente del proyecto, contando con la disponibilidad suficiente y necesaria, para interactuar con el equipo en todas las fases del proyecto.
2. ***SEMANA DE 40 HORAS (40 HOUR WEEK):*** eXtreme Programming asegura la calidad del equipo, considerando que éste, no debe asumir responsabilidades que le demanden mayor esfuerzo del que humanamente se puede disponer. Esto marca una diferencia radical con otras metodologías, sobre todo, con aquellas que siguen una línea tradicional donde las estimaciones deficientes obligan al equipo de desarrollo, a sobre-esfuerzos significativos.
3. ***METÁFORA (METAPHOR):*** A fin de evitar los problemas de comunicación que suelen surgir en la práctica, entre técnicos y usuarios, eXtreme Programming propone el uso de metáforas, intentando hallar un punto de referencia que permita representar un concepto técnico con una situación en común con la vida cotideana y real.
4. ***DISEÑO SIMPLE (SIMPLE DESIGN):*** Esta práctica, deriva de un famoso principio técnico del desarrollo de software: KISS. Cuyas siglas derivan del inglés "Keep it simple, stupid!" -¡Manténlo simple, estúpido!-. Básicamente, el principio KISS, se trata de mantener un diseño sencillo, estandarizado, de fácil compresión y refactorización. Puede resumirse en "hacer lo mínimo indispensable, tan legible como sea posible".
5. ***REFACTORIZACIÓN (REFACTORING):*** La refactorización (o refactoring) es una técnica que consiste en modificar el código fuente de un software sin afectar a su comportamiento externo.
6. ***PROGRAMACIÓN EN PAREJAS (PAIR PROGRAMMING):*** Otra de las prácticas técnicas fundamentales de eXtreme Programming, es la programación de a pares. Se estila (aunque no de forma constante) programar en parejas de dos desarrolladores, los cuales podrán ir intercambiando su rol, en las sucesivas Pair Programming. ¿En qué consiste la programación de a pares? Sencillo. Dos programadores, sentados frente a una misma computadora, cada uno cumpliendo un rol diferente. Las combinaciones y características de este rol, no solo son variables, sino que además, son inmensas, permitiendo la libertad de "ser originalmente creativos".
7. ***ENTREGAS CORTAS (SHORT RELEASES):*** Se busca hacer entregas en breves lapsos de tiempo, incrementando pequeñas funcionalidades en cada iteración. Esto conlleva a que el cliente pueda tener una mejor experiencia con el Software, ya que lo que deberá probar como nuevo, será poco, y fácilmente asimilable, pudiendo sugerir mejoras con mayor facilidad de implementación.
8. ***TESTING***: En esta práctica, podemos encontrar tres tipos de test -pruebas- propuestos por eXtreme Programming:
    - Test Unitarios, consisten en testear el código de manera unitaria (individual) mientras se va programando. Técnica conocida como Unit Testing, la cual forma parte de la técnica TDD (Test-driven development -desarrollo guiado por pruebas-).
    - Test de aceptación, están más avocados a las pruebas de funcionalidad, es decir al comportamiento funcional del código. Estas pruebas, a diferencia de los Test Unitarios, son definidas por el cliente y llevadas a cabo mediante herramientas como Selenium, por ejemplo, y basadas en Casos de Uso reales que definirán si la funcionalidad desarrollada, cumple con los objetivos esperados (criterios de aceptación).
    - Test de integración, tienen por objeto, integrar todos los test que conforman la aplicación, a fin de validar el correcto funcionamiento de la misma, evitando que nuevos desarrollos, dañen a los anteriores.

9. ***ESTÁNDARES DE CÓDIGO (CODING STANDARDS)***: Los estándares de escritura del código fuente, son esenciales a la hora de programar. Permiten hacer más legible el código y más limpio a la vez de proveer a otros programadores, una rápida visualización y entendimiento del mismo.
10. ***PROPIEDAD COLECTIVA (COLLECTIVE OWNERSHIP):*** Para eXtreme Programming no existe un programador "dueño" de un determinado código o funcionalidad. La propiedad colectiva del código tiene por objetivo que todos los miembros del equipo conozcan "qué" y "cómo" se está desarrollando el sistema, evitando así, lo que sucede en muchas empresas, que existen "programadores dueños de un código" y cuando surge un problema, nadie más que él puede resolverlo, puesto que el resto del equipo, desconoce cómo fue hecho y cuáles han sido sus requerimientos a lo largo del desarrollo.
11. ***INTEGRACIÓN CONTINUA (CONTINUOUS INTEGRATION):*** La integración continua de XP propone que todo el código (desarrollado por los miembros del equipo) encuentren un punto de alojamiento común en el cuál deban enviarse los nuevos desarrollos, diariamente, previo correr los test de integración, a fin de verificar que lo nuevo, no "rompa" lo anterior.
12. ***JUEGO DE PLANIFICACIÓN (PLANNING GAME):*** Con diferencias pero grandes similitudes con respecto a la planificación del Sprint en Scrum, la dinámica de planificación de eXtreme Programming llevada a cabo al inicio de la iteración, suele ser la siguiente:
    - El cliente presenta la lista de las funcionalidades deseadas para el sistema, escrita con formato de "Historia de Usuario", en la cual se encuentra definido el comportamiento de la misma con sus respectivos criterios de aceptación.
    - El equipo de desarrollo estima el esfuerzo que demanda desarrollarlas, así como el esfuerzo disponible para el desarrollo (las iteraciones en XP, suelen durar 1 a 4 semanas) . Éstas estimaciones pueden hacerse mediante cualquier técnica de estimación.
    - El cliente decide qué Historias de Usuario desarrollar y en qué orden.

3.3. Kanban

Kanban es un término Japonés el cual puede traducirse como "insignia visual" (Kan: visual, ban: sello o insignia) y comenzó a implementarse, prácticamente una década después de otros enfoques adaptativos. Básicamente, Kanban se apoya en una producción a demanda: se produce solo lo necesario, de manera tal que el ritmo de la demanda sea quien controle al ritmo de la producción (se necesitan N partes, entonces solo se producen N partes), limitando el trabajo en curso a una cantidad predefinida.

> *[...] Un sistema Kanban es un sistema o proceso diseñado para disparar trabajo cuando hay capacidad para procesarlo [...] (2005, David Anderson, Microsoft)*

En el sistema Kanban, este disparador es representado por tarjetas, que se distribuyen en cantidades limitadas (esto es, lo que limitará el trabajo en curso). Cada ítem de trabajo, se acompaña de una de estas tarjetas, por lo cual, un nuevo ítem sólo podrá iniciarse si se dispone de una tarjeta Kanban libre. Cuando no hay más tarjetas libres, no se pueden iniciar nuevos trabajos. Y cuando un ítem es concluido, una nueva tarjeta se libera, permitiendo el comienzo de un nuevo ítem de trabajo.

Partiendo de las bases anteriores, en el desarrollo de Software, se "virtualiza" esta esencia, llevándola a la práctica mediante la implementación de tres reglas:

Mostrar el proceso

Esta regla busca hacer visibles los items de trabajo permitiendo conocer de manera explícita el proceso trabajo actual, así como los impedimentos que vayan surgiendo. Dicha visualización, se realiza a través de tableros físicos, al igual que en Scrum, solo que con diferentes columnas (que veremos más adelante).

En el caso de Kanban, la implementación de tableros físicos, es fundamental a la hora de comprender la capacidad de proceso del equipo de desarrollo. Para cumplir con la regla de mostrar el proceso, será necesario definir con precisión:

1. ***Punto de inicio y finalización de la visibilidad del proceso***: Limitar la visualización a las actividades del proceso en las cuales el equipo de desarrollo tiene mayor influencia para efectuar cambios. A partir de estos límites, se negocia la forma de interacción entre procesos anteriores y posteriores.

2. ***Tipos de ítems de trabajo:*** Esto hace referencia a los tipos de items solicitados en el proceso de trabajo (por ejemplo, bugs, refactoring, actualizaciones, etc.)
3. ***Diseño de las tarjetas que acompañarán a cada ítem:*** La información contenida en cada tarjeta, debe ser lo suficientemente precisa, como para permitir su evaluación y la toma de decisiones a cualquier parte involucrada en el proyecto. Generalmente, la información mínima que debe contener cada item es: titulo, ID, fecha de entrada, tipo de ítem, persona asignada, prioridad y deadline.

Limitar el trabajo en curso (WIP)

En Kanban, el límite de trabajo en curso, está dado por el WIP: Work in Progress. El límite WIP, consiste en definir la cantidad de ítems simultáneos que pueden ejecutarse en un mismo proceso. Los límites WIP, nos permitirán detectar los cuellos de botella rápidamente. Con solo observar el tablero prestando atención en las columnas bloqueadas y su procesos anterior y posterior inmediatos, podremos deducir donde se encuentran los puntos de conflicto, y trabajar para remediarlos.

Existen diversas formas de manejar los cuellos de botella, pero sin dudas, la que mejores resultados ha demostrado dar, es la de colocar una cola de entrada (buffer) previa al proceso donde el cuello de botella se genera.

Optimizar el flujo

El flujo de trabajo es la progresión visible de los ítems de los sucesivos procesos en un sistema Kanban. El objetivo de optimizar dicho flujo de trabajo, es alcanzar un proceso de desarrollo estable, previsible y acorde a las necesidades del proyecto, en el cual se distinga un ritmo de trabajo parejo.

Cuando un ítem de trabajo se encuentra estancado, el equipo deberá colaborar a fin de lograr recuperar un flujo parejo y tomar medidas para prevenir futuros estancamientos.