# DESARROLLO DE UNA MEJORA EN LA METODOLOGÍA V EN LA FASE DE PRUEBAS DE SOFTWARE IMPLEMENTADO EN EL CAMPO DE LA AERONÁUTICA.

# TESINA

PARA OBTENER EL GRADO DE

**MAESTRÍA EN**
**SISTEMAS INTELIGENTES MULTIMEDIA**

PRESENTA
**I.S.C MARIANA AURELIA ÁLVAREZ BÁEZ**

SANTIAGO DE QUERÉTARO, QUERÉTARO, JUNIO DE 2017.

*A mi familia.*

24 de agosto de 2016

Dr. Miguel González Valadez
Director de posgrado

La abajo firmante, miembro del Comité Tutorial del alumno I.S.C Mariana Aurelia Álvarez Báez, una vez revisada la Tesis o tesina titulada: "Desarrollo de una mejora en la Metodología V en la fase de pruebas de software implementado en el campo de la Aeronáutica", autoriza que el citado trabajo sea presentado por el alumno para la revisión del mismo con el fin de alcanzar el grado de Maestra de Sistemas Inteligentes Multimedia durante el Examen de Titulación correspondiente.

Y para que así conste se firma la presente a los 24 días del mes de agosto del año 2016.

M. en C. Itzel Fabiola Mata Ávila
Asesor en Planta

11 de mayo de 2017

Respetables miembros del Jurado,

Me ha tocado el honor de haber sido designado Revisor del trabajo titulado "DESARROLLO DE UNA MEJORA EN LA METODOLOGÍA V EN LA FASE DE PRUEBAS DE SOFTWARE IMPLEMENTADO EN EL CAMPO DE LA AERONÁUTICA" del Ing. Mariana Aurelia Álvarez Báez.

Después de haber leído detalladamente el trabajo que me fue entregado, he tenido la oportunidad de intercambiar información con el sustentante y como resultado de estas acciones he concluido que:

El trabajo tiene los siguientes aspectos positivos:

1.- Trabajo enfocado a resolver un problema real de la industria.
2.- Estructura consistente.
3.- Integración de metodologías modernas para el desarrollo de proyectos a gran escala.

El trabajo tiene las siguientes oportunidades de mejora:

1.- Desarrollo de un plan experimental exhaustivo para la validación de hipótesis.
2.- Mejora de la justificación basada en el contexto del campo de la aeronáutica.
3. Identificar las principales contribuciones del trabajo.

Haciendo un análisis crítico del trabajo y balanceando lo positivo y las oportunidades de mejora, considero RECOMENDAR al Jurado que le otorgue el Grado de Maestro en Sistemas Inteligentes Multimedia, al Ing. Mariana Aurelia Álvarez Báez por lo que acepto se imprima el trabajo de tesis.

No obstante lo anterior, le solicitaría al sustentante me responda las siguientes preguntas:

1.- ¿Cuál es la principal contribución de tu trabajo?
2.- ¿Cuál es la diferencia entre tu trabajo y esfuerzos similares?

3.- ¿Consideras un resultado diferente al descrito en este trabajo, dada la naturaleza de la evaluación presentada? De ser así, ¿en qué casos no se cumplirían las hipótesis planteadas?

Le agradecería al Honorable Jurado tenga en consideración la propuesta de otorgar el Grado que pongo a su consideración.

Atentamente,

Ismael Solís Moreno
Doctor en Ciencias de la Computación

# Resumen

Actualmente, los equipos de desarrollo de software aeronáutico necesitan acelerar la manera en que crean nuevo software ya que la industria está avanzando cada vez más rápido, innovando e intentando estar a la vanguardia tecnológica. Sin embargo, sigue habiendo software con fallas, lo que sigue permitiendo pérdidas considerables en tiempo y dinero.

El acelerar las cosas no debe comprometer la calidad de los productos, sin embargo, por querer entregar en tiempo, la calidad de los productos se ve comprometida, ocasionando que el software falle incluso en pleno vuelo, por lo que lo ahorrado en la creación del software, se ve perdido a la hora de arreglar sus defectos.

Es por esto que es necesario implementar mejoras en las metodologías que se utilizan actualmente para poder garantizar la entrega a tiempo y que aun así el software se ejecute sin fallas. Las metodologías actuales dan resultados bastante satisfactorios y siguen siendo eficientes, sin embargo, se necesita implementar mejoras que las actualicen al mundo que vivimos el día de hoy.

Como respuesta a esta necesidad, se buscó implementar la metodología V o de cuatro niveles a una ejecución ágil de manera que se llevara a cabo en conjunto con los elementos de una metodología ágil, reduciendo así tiempo y garantizando la calidad de los productos. Esta metodología es la más utilizada actualmente en el Centro de Ingeniería Avanzada en Turbo-máquinas (GEIQ) para el desarrollo de proyectos de software aeronáutico, por lo que invertir en ella podría producir beneficios para el Centro.

Para lograr esta mejora, se tomaron los procesos que actualmente corren en los equipos de GEIQ *Aviation Systems and Digital Services* y se fue introduciendo poco a poco herramientas y conceptos ágiles de *Scrum*, los artefactos y las juntas en las que se basa, así como los principios en los que se fundamentaron los métodos ágiles. El propósito de esta mejora es detectar el punto dónde se perdía más tiempo, presupuesto del proyecto y donde surgían más fallas. Para su sustentación, se tomaron los tiempos promedios que los proyectos suelen consumir.

Como resultado de esta investigación, se introdujo la validación y verificación del software al mismo momento que el desarrollo para que la solución de las fallas fuera en el momento y no hasta el final, ahorrando así tiempo valioso de ejecución del proyecto.

Para lograr la introducción exitosa del V&V en la etapa de desarrollo, es importante valerse de los elementos de las metodologías ágiles y *pair programming*, para que el ciclo de vida de software fluya sin contratiempos.

El resultado fue una reducción de hasta un 19% en tiempo de ejecución de desarrollo y V&V.

# Índice de contenido

Resumen ..... 1
Glosario ..... 7
Definiciones ..... 7
Siglas ..... 9
Capítulo 1.- Introducción ..... 10
1.1. Antecedentes ..... 10
1.2. Problema ..... 11
1.3. Justificación ..... 11
Objetivo General ..... 13
1.4. Hipótesis ..... 13
Capítulo 2.- Marco Teórico ..... 14
2.1. Ciclo de vida del software ..... 14
2.1.1. Las pruebas en el ciclo de vida de software ..... 16
2.2. Metodologías de desarrollo de software ..... 16
2.2.1. Modelo Cascada ..... 17
2.2.2. Modelo V o de Cuatro Niveles ..... 19
2.3. Metodologías actuales para la creación de pruebas de software ..... 21
2.4. Software aviónico ..... 24
2.5. Estándares DO-178 ..... 27
2.6. Criticidad de Software ..... 27
2.7. Verificación de software ..... 28
2.8. Validación de software ..... 29
2.9. Técnicas de Pruebas de Software ..... 30
2.9.1. Tipos de prueba ..... 31
2.9.2. De unidad ..... 33
2.9.3. De integración ..... 34
2.9.4. De regresión ..... 38
2.9.5. De aceptación ..... 39
2.10. Metodologías Ágiles ..... 40
2.10.1. Test Driven Development ..... 42

2.10.2. Scrum ..... 43
2.10.3. Kanban ..... 50
2.10.4. Lean ..... 52
2.10.5. Extreme Programming ..... 54
2.10.6. Pair Programming ..... 56
2.11. Relación Marco Teórico - Investigación ..... 57
Capítulo 3.- Solución propuesta ..... 58
3.1. Investigación previa ..... 58
3.1.1. Resultados de la investigación previa ..... 58
3.2. Problemática con Agile ..... 62
3.3. Metodología V ágil ..... 63
3.4. Estableciendo el proyecto ..... 66
3.4.1. Requerimientos del cliente ..... 67
3.4.2. Arquitectura de alto nivel ..... 68
3.5. Planeación ..... 69
3.5.1. Preparación del *backlog* ..... 69
3.5.2. Definition of Done ..... 74
3.5.3. Definición del tamaño de *sprint* ..... 75
3.6. Implementación ..... 75
3.6.1. Proceso de desarrollo de sprint ..... 75
3.6.1.1. Validación y verificación informal ..... 76
3.6.1.2. Validación y verificación formal ..... 77
3.6.2. Creación de los equipos de trabajo ..... 79
3.6.2.1. Equipo de liderazgo y administración de proyecto ..... 79
3.6.2.2. Equipo técnico ..... 79
3.6.3. Ejecución de *sprints* ..... 80
3.6.4. Simulación ..... 82
Resultados ..... 90
Conclusiones ..... 93

Recomendaciones ..... 96
Referencias bibliográficas ..... 97

# Índice de figuras y tablas

Figura 1 - Modelo de desarrollo de software 'Cascada'. ..... 19
Figura 2 - Modelo de desarrollo de software 'V o de Cuatro Niveles'. ..... 20
Figura 3 - Verificación en el ciclo de vida de software. ..... 29
Figura 4 - Validación en el ciclo de vida de software. ..... 30
Figura 5 - Pruebas de integración. ..... 35
Figura 6 - Descripción gráfica de 'Top Down'. ..... 36
Figura 7 - Descripción gráfica de 'Bottom Up'. ..... 36
Figura 8 - Descripción gráfica de pruebas de integración 'Big-Bang'. ..... 37
Figura 9 - Ciclo de Test-Driven Development (TDD) ..... 43
Figura 10 - Modelo de desarrollo Scrum ..... 44
Figura 11 - Scrum Meetings ..... 47
Figura 12 - Tablero Kanban con estados. ..... 51
Figura 13 - Pasos Lean para cualquier proceso. ..... 54
Figura 14 - Diagrama de arquitectura a alto nivel ..... 69
Figura 15 - Diagrama de flujo de etapa de V&V informal ..... 77
Figura 16 - Diagrama de flujo de etapa de V&V formal ..... 78
Figura 17 - 1ra opción de organigrama de equipo técnico. ..... 80
Figura 18 - 2da opción de organigrama de equipo técnico ..... 80
Figura 19 - Calendario del sprint 1 ..... 81
Figura 20 - Calendario del sprint 2 ..... 81
Figura 21 - Calendario del sprint 3 al n ..... 82
Figura 22 - Gráfica comparativa de costos Cascada y V-ágil ..... 91

Tabla 1 – Cotización inicial del ciclo informal. ..... 59
Tabla 2 - Cotización inicial ciclo formal. ..... 60
Tabla 3 - Costo de desarrollo en Cascada. ..... 61
Tabla 4 - Costo de V&V en Cascada. ..... 62
Tabla 5 - Tabla de requerimientos del cliente ..... 68
Tabla 6 - Organización de requerimientos ..... 71

Tabla 7 - Organización de documentación obligatoria ..... 71
Tabla 8 - Llenado de parámetros para cada ítem..... 74
Tabla 9 - Datos promedio históricos. ..... 83
Tabla 10 - Calendarización del proyecto bajo metodología Cascada. ..... 84
Tabla 11 - Calendarización del proyecto bajo metodología V ágil..... 89
Tabla 12 - Costo de desarrollo en metodología V ágil..... 91
Tabla 13 - Costo final de V&V en metodología V ágil..... 91

# Glosario

## Definiciones

- ***Display***:
  Pantalla que muestra el estado de todos los componentes aviónicos en el tablero de vuelo de los aviones.
- ***Drivers***:
  Simulación de llamada de una función a otra, para poder ejecutar un *test case*.
- ***Long dispatch***:
  Mensaje de alerta emitido por el motor cuando falla un componente, pero permite que el avión vuele no más de 250 horas.
- ***No-dispatch***:
  Mensaje de alerta emitido por el motor cuando falla un componente vital para su operación. Este mensaje indica que el avión ya no puede volar de manera segura.
- ***Problem Reports***:
  Documento que especifica defectos y fallas encontrados durante la validación y verificación del software.
- **Retorno de inversión**:
  Indicador financiero que mide la rentabilidad de una inversión, es decir, la relación que existe entre la utilidad neta o la ganancia obtenida, y la inversión.
- **Requerimiento Derivado**:
  Tipo de requerimiento que no especifica una funcionalidad solicitada por el cliente, pero que es necesaria para alcanzar los objetivos establecidos por este.
- **Requerimiento de Seguridad**:
  Tipo de requerimiento que especifica las necesidades y funcionalidades críticas del sistema de software.
- **Sistema de administración de vuelo**:
  Componente aviónico que automatiza varias tareas a realizar durante un vuelo, tales como la gestión de planes de vuelo, reduciendo el trabajo de la tripulación al mando.

- ***Short dispatch***:

  Mensaje de alerta emitido por el motor cuando falla un componente, pero permite que el avión vuele no más de 125 horas.

- ***Software Development Plan***:

  Plan de Desarrollo de Software, en español. Es el documento que describe el plan a seguir por los desarrolladores para la creación de nuevo software, así como los procesos a seguir y las herramientas a utilizar para ese fin.

- ***Software Verification Plan***:

  Plan de Verificación de Software, en español. Es el documento que describe el plan a seguir por los desarrolladores para la verificación de un software, así como los procesos a de revisión y validación a seguir, el tipo de pruebas a aplicar y las herramientas a utilizar para ese fin.

- ***Test Specification Procedure***:

  Es el documento que detalla cómo el *tester* llevará a cabo las pruebas: los pasos a seguir, las entradas a utilizar, las salidas esperadas, la herramienta a utilizar y el resultado de la prueba.

- ***Stubs***:

  Simulación de una función cuando al desarrollar un *test case*, se requiere entradas de esta para poder probar una función.

## Siglas

- **FADEC** – *Full Authority Digital Engine Control*.
- **FMS** – *Flight Management Display*, siglas en inglés de Sistema de Administración de vuelo.
- **GEIQ** – *General Electric Infrastructure Querétaro*. Siglas con las que se conoce al Centro de Ingeniería Avanzada en Turbo-máquinas de General Electric
- **IEEE** - *Institute of Electrical and Electronics Engineers*.
- **ISTQB** – *International Software Testing Qualifications Board*.
- **MC/DC** – *Modified Condition / Decision Coverag*, siglas para metodología de prueba que busca ejecutar todas las decisions y condiciones implementadas.

# Capítulo 1.- Introducción

## 1.1. Antecedentes

Un accidente de aviación es un incidente que ocurre con cualquier aeronave, ya sea en tierra o en vuelo. De acuerdo a la Organización de Aviación Civil Internacional (ICAO, por sus siglas en inglés) está definido como (1):
"Una ocurrencia asociada con la operación de una aeronave que tiene lugar durante el tiempo que cualquier persona aborda la aeronave con la intención de volar hasta que todas las personas de la misma hayan desembarcado en la que un tripulante recibe lesiones severas o letales o la aeronave sufre daños estructurales.

Estos incidentes suelen ocurrir en medida por fallos en el software, que es cada vez más complejo. Según un reporte estadístico lanzado por la Red de Seguridad Aérea (2), tan sólo en el 2013 se registraron, por lo menos, 29 accidentes con fatalidades humanas y 265 víctimas mortales. El número de víctimas es sin embargo significativamente menor a la media de los últimos diez años, que se situaban en 720 víctimas/año (3), dado a la evolución tecnológica de los últimos años; sin embargo, los niveles de complejidad que ha alcanzado el desarrollo de software aviónico, ha dificultado la tarea de probarlo de tal manera que garantice un número menor de accidentes y, por lo tanto, de víctimas mortales, así como pérdidas monetarias muy grandes.

Actualmente existen diversas metodologías de pruebas, tanto como estáticas como dinámicas, que permiten probar el software y detectar posibles fallos. Estas metodologías procuran cubrir todos los posibles escenarios en los que el software será sometido y son regidas por estándares de certificación que, según las funciones que tiene que cumplir el software, puede poner en peligro la seguridad del avión en mayor o menor medida. De acuerdo al DO-178 (11), según el nivel que reciba el software, se establecen los métodos permitidos para el desarrollo del software dando lugar a diferentes obligaciones de documentación y revisión, por ejemplo, un software nivel A, como el utilizado en Aeronáutica, debió haber pasado por un escrutinio que asegure

que se cubren todas las decisiones, así como pruebas donde se ejecuten cada posible resultado (MC/DC[1]).

## 1.2. Problema

Por más avances realizados en la última década en la tecnología, tales como la introducción de herramientas de automatización de código y *test cases* en el desarrollo de software aplicado especialmente en el área de la Aviación, los accidentes aéreos por fallas en el software siguen ocurriendo con frecuencia alrededor del mundo. Las investigaciones realizadas por las organizaciones de Aviación mundiales no han podido descifrar si las pruebas en el software, cada vez más complejo, empleadas en los distintos productos de la aeronave son suficientes, óptimas y completas como para garantizar que el software no falle durante su ejecución en vuelo.

## 1.3. Justificación

En la actualidad el índice de fallas o demoras en proyectos de software es muy alto. Según Luis López Goytia, en su libro "*Programación Orientada a objetos C++ y Java*" (44), en el año 2001 se registró que sólo el 28% de los proyectos de software realizados en el mundo fueron exitosos, 23% fueron cancelas y 49%, excedidos en tiempo.

Las compañías hacen algún tipo de pruebas de software, aunque no todas le dan el mismo grado de seriedad a este proceso. Usualmente esto tiene que ver con el tamaño y la madurez de las compañías, la inversión en pruebas de software fluctúa entre el 10% en compañías pequeñas y el 70% en compañías transnacionales. Uno de los problemas más comunes que causan que los proyectos fallen o que los costos se eleven mucho, ésta relacionado con la selección de un método de pruebas de software adecuado.

[1] Ver Glosario para su definición.

En la industria aeroespacial, el software tiene una gran relevancia e impacto dado que usualmente éste es el que habilita y controla las características o capacidades que puede ofrecer un hardware o componente electrónico, resultando en sistemas de aviónica totalmente dependientes al software, donde el desempeño durante la ejecución no tolera fallas.

Las aplicaciones de software utilizadas en instrumentos de navegación e interfaces humano-máquina son a final de cuentas la herramienta principal utilizada por los pilotos durante el vuelo (desde el despegue hasta el aterrizaje). Las fallas en estas aplicaciones pueden derivar en costos para las aerolíneas, costos para el proveedor del equipo, daños materiales e inclusive lamentables pérdidas humanas.

Es por todo esto que este tipo de software es categorizado como software de misión crítica, y es necesario contar con la certificación de organismos reguladores como la FAA o EASA, los cuales garantizan la confiabilidad y seguridad de este tipo de productos y dan su autorización para uso comercial y/o militar.

Existen casos documentados como el accidente de Malasia en el 2005, donde la FAA diagnosticó que la causa fue un problema de software y ordenó a todas las compañías que vuelan aviones Boeing 777 a hacer una actualización de software.

A medida que el software se vuelve más complejo, las pruebas y certificaciones que aseguran la viabilidad del mismo se vuelven igual de complejos. Día a día surgen nuevas tecnologías y metodologías para el desarrollo de pruebas de software que aparentan facilitar cada vez más las tareas, pero por otro lado añaden mayor complejidad en las certificaciones. También es importante mencionar que existen diversas herramientas que asisten en los procesos de verificación y validación de software de misión crítica, pero, no existen métodos específicos que asocien una metodología con el uso óptimo de estas herramientas.

## Objetivo General

Desarrollar una mejora en la metodología de pruebas de software implementado en el campo de la Aeronáutica.

## 1.4. Hipótesis

- La correcta elección de metodologías de pruebas y la estandarización de las mismas disminuirá en un 20% el número de ocurrencia de fallas en el software durante su ejecución en vuelo.
- La utilización de múltiples metodologías de pruebas garantiza pruebas robustas y completas de los requerimientos del software del producto aviónico.
- A medida que el software empleado sea complejo, nuevas metodologías de pruebas tendrán que ser desarrolladas para probarlo óptima y completamente.

# Capítulo 2.- Marco Teórico

## 2.1. Ciclo de vida del software

De acuerdo a Francisco Ruiz y su conferencia "Procesos de Ingeniería de Software" (4) , la Ingeniería de software es la aplicación y enfoque sistemático de herramientas y métodos que permite la producción de sistemas de software o como es descrita por la IEEE (1074) es: "Una aproximación lógica a la adquisición, el suministro, el desarrollo, la explotación y el mantenimiento del software".

Para lograrlo, la Ingeniería de software se basa en el ciclo de vida del software, que no es más que la división en fases con objetivos específicos del desarrollo total del software que surge de la necesidad de dar una solución a un problema y termina cuando el programa creado es puesto en marcha y se le da el apropiado mantenimiento. Es una manera de organizar el desarrollo de software.

De acuerdo a la Universidad Nacional de Colombia (5) (6), en general, las fases del ciclo de vida se componen de la siguiente manera:

- **Análisis**:

En esta etapa se debe comprender detalladamente cuál es el problema a resolver, así como los factores que lo provocan, esto con el fin de entender el entorno donde se desarrollará la solución. Se enfoca en dar una perspectiva del sistema desde los puntos funcionales, así como el establecimiento de los requerimientos del cliente.

El resultado de esta fase es una descripción concisa y real del problema que dé lugar a enlistar las tareas que el programa de software desempeñará, así como sus causas y efectos para poder establecer a detalle sus funcionalidades y relaciones que crearán su comportamiento final.

- **Diseño**:

En esta etapa se desarrolla el cómo se va a solucionar el problema a tratar y el cómo se desarrollarán los requerimientos dados por el cliente. Se decide cuál es la mejor

manera a tratar la situación a resolver estableciendo la estructura del programa siempre considerando los aspectos de software y hardware donde el programa será implementado.

Se parte de lo general a lo particular, diseñando primero la estructura a grandes rasgos del programa, pasando a dividirlo en tareas más específicas y manejables para facilitar su implementación y corrección. Las relaciones lógicas entre sus módulos, sus objetos y entidades son especificadas durante esta etapa al igual que la manera en que se implementará.

- **Implementación**:

También llamada codificación, es aquí donde el diseño es llevado a la acción, es decir, se desarrolla mediante las herramientas, procesos y lenguajes de programación convenientes para dar lugar a la solución del problema. Los requerimientos del cliente toman forma y se vuelven un producto, codificando algoritmos u estructuras de datos previamente definidos.

- **Pruebas**:

El diseño y la implementación conllevan muchos errores por lo que es necesaria la creación de pruebas que garanticen que el problema ha sido resuelto correctamente y que los requerimientos del cliente son logrados con éxito. Garantizan el correcto funcionamiento del programa de software que se dio como resultado de las etapas anteriores.

- **Mantenimiento**:

Una vez que el programa es puesto a funcionar y provee la solución para la que fue creado, puede ir deteriorándose, es por ello que es importante darle actualizaciones para que nunca deje de ser útil y se mantenga funcionando correctamente. Se determinan las deficiencias del programa y se realizan las acciones necesarias para revocarlas mientras se preserva su integridad.

- **Documentación**:

Aunque esta no es una etapa como tal en el ciclo de vida de software, juega un papel muy importante. Es imperativo poner por escrito todos los procedimientos, diagramas, tablas e instrucciones del programa que se creó. Esto con el fin de tener una guía para comprender o usar el programa y facilitar su mantenimiento y futuras modificaciones. Lo recomendable es llevar esta comunicación escrita durante todas las etapas.

### 2.1.1. Las pruebas en el ciclo de vida de software

El desarrollo de software es un proceso que involucra ambigüedad, suposiciones y fallas en las comunicaciones del equipo. Cada cambio realizado a un componente de software, la adición de una nueva funcionalidad, el intento de reparar un defecto, conlleva a una posibilidad de generar un fallo. Con cada fallo, el riesgo de que el software no cumpla con los requisitos del cliente, se incrementa. Las pruebas de software tienen como objetivo reducir o minimizar el impacto de estos fallos en la funcionalidad y en los resultados esperados de la aplicación. (7) Son, como lo define la IEEE, "los procesos de evaluación de un sistema o componente para verificar que satisface los requisitos especificados, o identificar diferencias entre lo esperado y los resultados obtenidos".

Las pruebas de software son consideradas como el proceso de ejecutar una aplicación en condiciones específicas, monitoreando los resultados, evaluando los aspectos del componente, proceso o sistema. Involucrarlas en el ciclo de vida de software permitirá que los errores se detecten lo antes posible y por lo tanto, permite a los desarrolladores concentrarse en la calidad del software, en los plazos de implementación y en los costos asociados.

## 2.2. Metodologías de desarrollo de software

A lo largo del tiempo, el desarrollo de software a nivel proyecto, se ha vuelto cada vez más complejo, entre las peticiones de los clientes, las fechas de entrega y la capacidad y velocidad de desarrollo han hecho que proyectos de software nacidos de grandes ideas hayan fracasado.

Para solucionar esto, se fueron creando métodos y modelos desde 1960 que fueran capaces de desarrollar de manera estructurada y metódica sistemas de software funcionales y de gran escala partiendo de la especificación precisa de requisitos y diseño a la planeación y control del proceso de desarrollo.

Actualmente, se enfocan muchos estudios a la creación de metodologías para tener un desarrollo de software más eficaz, que se haga en menor tiempo logrando e incluso dé mejores resultados que siguiendo un método tradicional.

Un **modelo** de desarrollo de software es una representación simplificada del proceso de desarrollo vista desde una perspectiva específica, mientras que una **metodología** es un enfoque más estructurado que incluye modelos y reglas que involucra el desarrollo de software.

### 2.2.1. Modelo Cascada

El método Cascada es considerado el enfoque clásico del ciclo de vida de desarrollo de software por excelencia por su desempeño lineal, es decir, que cada fase empieza cuando la fase anterior termina, idealmente consiguiendo todos los objetivos por fase. Cuando se considera que la etapa aún no está lista para pasar a la siguiente etapa, se queda en la actual.

Este método fue originalmente propuesto por Winston W. Royce en 1970 (8), en su publicación "Administrando el desarrollo de grandes sistemas de software", en el que exploró cómo el modelo inicial se podría desarrollar en un modelo iterativo, con la regeneración a partir de cada fase influenciando las fases subsecuentes.

Royce demostró que mientras el desarrollo de grandes sistemas de software necesitara un enfoque más a fondo, no habría riesgo en un solo paso del enfoque secuencial y propuso un enfoque iterativo e incremental y abogó para que los proyectos tengan que pasar por esto al menos 2 veces, y aunque esto último no llegó a cimentarse como

parte del modelo, hoy por hoy el modelo cascada es muy popular y cuenta con 5 pasos esenciales (Véase *Figura 1*):

- **Análisis**:
  Esta etapa consiste en reunir las necesidades del producto y su resultado es el establecimiento de los requerimientos base del programa de software a desarrollar; se definen la planificación y alcance del proyecto.

- **Diseño**:
  Esta etapa describe la estructura interna del programa y suele representarse con diagramas.

- **Implementación**:
  Esta etapa consiste en el desarrollo de la codificación del software en cualquier nivel.
  Suele ser la etapa de más larga duración y en la que suelen surgir más cambios, ya sea por cambios de alcance, corregir fallas y errores o introducir mejoras.

- **Pruebas**:
  Las partes del software deben funcionar correctamente, tanto por separado como integradas ya en un solo programa, el software debe cumplir con los requisitos previamente especificados, así como su funcionamiento sin fallos.
  En esta etapa se buscan todos los posibles fallos para posteriormente eliminarlos.

- **Mantenimiento**:
  El software seguirá evolucionando aún después de haberlo entregado al cliente, es decir, puede que requiera actualizaciones y adaptaciones para seguir funcionando en el entorno para el que fue desarrollado, por lo que en esta etapa se busca tener el software listo para las necesidades del día a día, aunque puede que requiera pasar por etapas de Implementación y Pruebas para cumplir con esos objetivos.

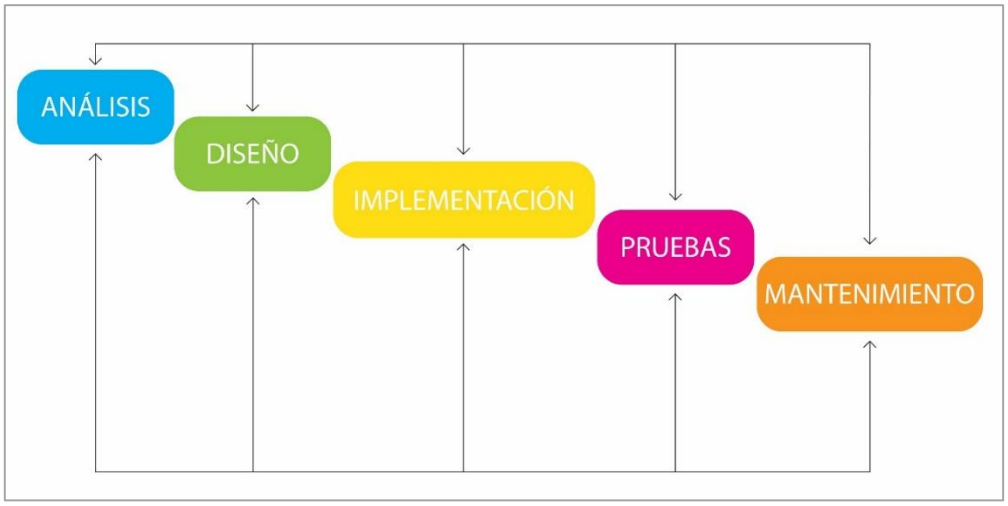


*Figura 1 - Modelo de desarrollo de software 'Cascada'.*

Actualmente, es el modelo más usado cuando el proyecto es rígido, ya que ayuda a prevenir que se sobrepasen las fechas de entrega y los costos esperados, y permite que al final de cada fase se tenga la oportunidad de revisar el progreso del proyecto, aunque no precisamente refleja el proceso del desarrollo de software, ya que se tarda mucho tiempo en pasar por todo el ciclo.

### 2.2.2. Modelo V o de Cuatro Niveles

Para la Unidad de Desarrollo Tecnológico en Inteligencia Artificial, el modelo V o de Cuatro Niveles (43) es un método robusto utilizado para proyectos pequeños que involucran hasta cinco personas y su objetivo es regular el proceso, de manera que se minimicen los riesgos y se garantice la calidad del proyecto.

Busca ser más efectivo que el Modelo Cascada, por lo que se desarrolla bajo el esquema en el que, para cada fase de desarrollo, existe una fase paralela de verificación y validación, obedeciendo al principio de que para cada fase debe existir un resultado verificable.

Este método se divide en cuatro niveles lógicos (43) en el que se desarrollan las fases del ciclo de vida de software:

- **Nivel #1:**

  Este nivel está orientado al cliente del sistema a desarrollar. Es el inicio y fin del proyecto, ya que constituye los dos extremos del ciclo.

  En este nivel se encuentra la fase de **Análisis de requerimientos** y su entregable verificable sería la aceptación del usuario a todo lo acordado para el sistema.

- **Nivel #2:**

  El enfoque de este nivel es meramente funcional, es decir, establece las características funcionales del sistema. En este nivel se encuentra la fase de **Diseño** y su entregable verificable es la compilación de los resultados de las pruebas de integración.

- **Nivel #3:**

  Durante este nivel, los componentes tanto de hardware y software que mejor le convengan al sistema son definidos. La fase correspondiente a este nivel es la de **Arquitectura del sistema** y su entregable verificable son los resultados de las pruebas de integración.

- **Nivel #4:**

  Durante este nivel, se desarrollan los elementos y módulos del sistema. La correspondiente fase a este nivel es el de la **Implementación** y su entregable verificable son las pruebas unitarias.

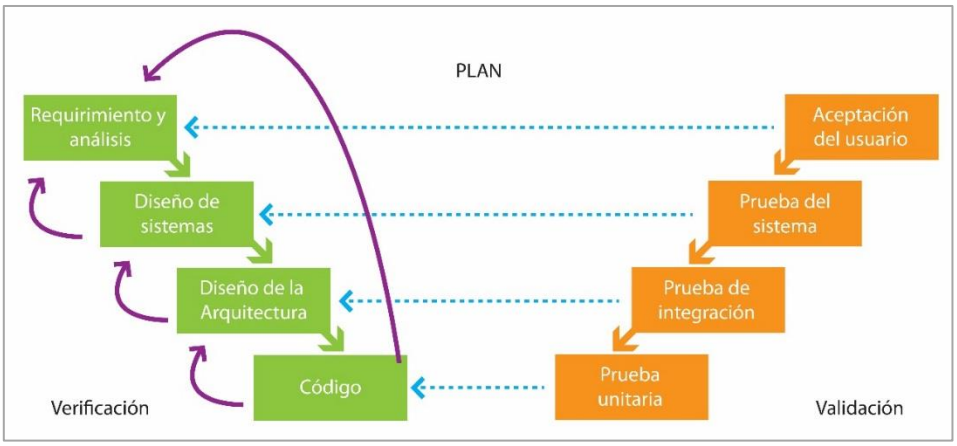


*Figura 2 - Modelo de desarrollo de software 'V o de Cuatro Niveles'.*

La proximidad entre una fase de desarrollo con su fase de verificación y validación va decreciendo conforme se avanza de nivel, por eso su descripción gráfica se traduce en una 'V' (Véase Figura 2). La longitud de esta separación intenta ser proporcional a la distancia en el tiempo entre una fase y su homóloga de verificación.

Dado que cada fase es validad y verificada, las iteraciones entre niveles pueden darse continuamente y en numerosas veces, por lo que se puede volver un método de desarrollo largo y tedioso a pesar de su robustez.

## 2.3. Metodologías actuales para la creación de pruebas de software

De acuerdo con Javier Sánchez Zapata y su artículo "Pruebas de software" (9), las pruebas de software son sumamente importantes dentro de cualquier metodología de Ingeniería de software, ya que buscan garantizar la integridad y buena calidad del software creado mediante la generación de casos de pruebas efectivos que revelan la presencia de fallas y errores.

Una **falla** es toda discrepancia visible que se produce al ejecutar un programa con un defecto produciendo que el software deje de funcionar correctamente y un error es aquella equivocación cometida por el desarrollador del software. Al implementar una metodología de pruebas, lo ideal sería poder determinar un conjunto efectivo de casos de prueba tal que su ejecución exitosa implique encontrar todas las fallas y errores y permita su posterior eliminación y corrección.

El proceso de pruebas formal debe contemplar por lo menos los siguientes 5 puntos importantes:

- **Planeación**:
  Esta es la etapa inicial de cualquier proceso de pruebas, ya que se determinan las metas, estrategias generales y actividades en un plan de pruebas tomando en cuenta los siguientes aspectos:

- **Alcance**:

  Se determinan las funcionalidades del software que serán probadas en el tiempo determinado para tal actividad, es decir, definir qué queremos probar, por qué queremos probar esas funcionalidades, cuánto tiempo le dedicaremos a esta fase y cómo llevaremos a cabo estas pruebas.

  Estas funcionalidades se determinan bajo un análisis de riesgos realizado previamente, tomando en cuenta la magnitud de impacto de cada posible falla.

- **Tipos**:

  Dependiendo del producto de software, es el tipo de pruebas que se le debe aplicar (Véase Sección. 2.9).

- **Estrategia**:

  Es necesario determinar a través de un análisis de riesgos sobre cuáles funcionalidades se deben aplicar las pruebas, así como el nivel de profundidad de estas a partir de sus criterios de entrada y salida.

- **Criterios de Salida**:

  Estos criterios se deben definir para cada nivel de pruebas a ejecutar y pueden ser, entre algunos ejemplos, número de defectos críticos o porcentaje de funcionalidades de alto riesgo probadas.

- **Otros aspectos** que pueden incluir la estimación de tiempos, un punto que suele darse mal en la actualidad; la asignación de recursos y roles dentro del equipo que desarrollará las pruebas, entre otros.

- **Diseño de pruebas:**

  Una vez aprobado el plan de pruebas, empieza el diseño de las mismas, es decir, la creación de casos de pruebas (*test cases*, en inglés), que ayudarán al encargado de crear las pruebas (*tester*, en inglés) a desarrollarlas. Estos casos de prueba deben cubrir toda la lógica de las partes del software que se haya decidido probar, así como todas las posibles combinaciones para todos los posibles resultados, esto con el fin de atrapar todos los defectos que puedan existir.

- **Implementación y ejecución de pruebas:**

  Esta es la etapa donde se corren los casos de prueba bajo un entorno ya sea real o simulado donde se pueda observar el comportamiento de los casos de prueba sobre la lógica del programa. Los resultados de la ejecución de los casos de prueba deben ser documentados, con el fin de llevar un registro de dónde fue encontrada la falla o qué parte del código funciona correctamente. Es importante recalcar que no es tarea del *tester* implementar los cambios correspondientes para eliminar las fallas; se regresa el reporte de fallas a los codificadores para que sean ellos quienes eliminan los errores encontrados. Esto implica un nuevo proceso de pruebas para determinar si las fallas fueron corregidas con éxito.

- **Evaluación de criterios de salida:**

  Los criterios de salida son necesarios para determinar si es posible dar por terminado el proceso de pruebas. Cada proceso en cada proyecto tiene requisitos del mismo cliente y requisitos de calidad que se deben cumplir, por lo que se hace un análisis de los resultados de las pruebas para saber si estos requisitos fueron cumplidos satisfactoriamente, además de haber cumplido en su totalidad con el plan de pruebas (*test procedures*, en inglés) previamente establecido.

- **Cierre del proceso:**

  Esta es la etapa final donde se verifica si los entregables finales están listos según el plan de pruebas para pasar a la siguiente etapa en el ciclo de vida del software.

  Es importante mencionar que el proceso de pruebas puede ser suspendido antes de llegar a su cierre formal por causas tales como recorte de presupuesto, cambio de alcance, baja de personal, entre otros.

El software se divide en varios niveles para poder brindarle el proceso de pruebas adecuado.

Los niveles en que suele ser divido son:

- **Pruebas de sistema:**

  El software ya validado se integra con el resto del sistema para validar puntos como:

  - **Rendimiento:**

    Conocer los tiempos de repuestas del sistema en funcionamiento, así como su espacio total en disco, el flujo de datos a través de los canales de comunicaciones, entre otros.

  - **Resistencia**:

    Busca determinar qué tanto puede soportar el programa en ejecución y con los recursos disponibles para él.

- **Pruebas de validación:**

  Se prueba un software completamente integrado para evaluar el cumplimiento con los requisitos especificados.

- **Pruebas de integración**:

  Se comprueban la compatibilidad y funcionalidad de todas las interfaces que componen el software. Estas interfaces pueden ser módulos funcionales, aplicaciones individuales o aplicaciones de red.

## 2.4. Software aviónico

De acuerdo al Capítulo 1 del manual *Fundamentals of Avionics* (10), los instrumentos aviónicos son componentes electrónicos utilizados por las aeronaves para el desempeño de sus funciones, de ahí que su nombre se derivara de '*aviation electronics*' (instrumentos electrónicos de aviación, en español) pasando a ser '*avionics´* (aviónicos, en español).

Los *avionics* incluyen diversos tipos de dispositivos que van desde la navegación y desempeño hasta el control total de la aeronave. La evolución de estos instrumentos ha dependido de la creciente necesidad de dar al piloto y copiloto más información para llevar la aeronave a su destino sana y salva.

La instrumentación fue muy limitada desde la invención de la primera aeronave en 1903, por los hermanos Wright, hasta la Primera Guerra Mundial. La demanda de mejorar las aeronaves y sus instrumentos aceleraron su evolución y para 1918, la cabina ya contaba con indicadores de velocidad y gasolina, altímetros, brújulas, entre otros.

Esta evolución siguió un ritmo acelerado cuando empezó la carrera espacial, pero no fue sino hasta la década de 1970, que la aviación comercial dio un gran salto. La nueva tecnología fue introducida a la par que los aviones Boeing 757, 767 y Airbus 310 y desde entonces, la aviación militar, espacial y comercial ha invertido muchos recursos en su investigación e innovación.

Conforme la industria aeronáutica se ha vuelto cada vez más compleja, los instrumentos también, requiriendo sistemas de software más capaces y veloces, altamente compatibles con el hardware implementado. Los sistemas embebidos, resultado de esta evolución instrumental, cada vez requieren de una planeación más estricta dado que su enfoque está dirigido a la seguridad y confiabilidad de su funcionamiento.

El software aviónico es regulado por organismos que buscan su eficiencia al máximo para evitar pérdidas humanas. Uno de estos organismos es la *Federal Aviation Administration* (FAA), la cual es responsable de la seguridad de la aviación civil.

La FAA fue creada bajo el Acta de Aviación Federal de 1958 en Estados Unidos y sus roles más importantes son (11):

- Regular la aviación civil para promover su seguridad.
- Promover el desarrollo de nuevas tecnologías aeronáuticas.
- Desarrollar y operar un sistema de control aéreo y navegación para tanto la aviación civil como militar.
- Continuar con las investigaciones que fortalecen el *National Airspace System*[2] y la aeronáutica civil.

[2] El *National Airspace System* (NAS) está compuesto por el espacio aéreo, la infraestructura de navegación y los aeropuertos de los Estados Unidos.

- Desarrollar programas para contrarrestar el ruido en el espacio aéreo y otros efectos ambientales ocasionados por la aviación civil.
- Regular el transporte en el espacio comercial de los Estados Unidos.

Para cumplir con estos roles, la FAA enfoca todos sus esfuerzos a 7 principales actividades:

- **Regulación de Seguridad**:

  Vigila y actualiza continuamente las regulaciones y estándares que cubren la manufactura, operación y mantenimiento de aeronaves bajo certificaciones que garantizan su correcto y seguro funcionamiento.

- **Administración de Espacio y Tráfico Aéreo**:

  Vigila el uso seguro y eficiente de los espacios aéreos designados para volar, así como desarrolla reglas para el tráfico aéreo y asigna los espacios para un mejor control.

  Opera una red de torres aeroportuarias de control y servicio.

- **Infraestructura de Navegación Aérea**:

  Construyen e instalan ayudas visuales y electrónicas para hacer de la navegación aérea algo más sencillo. Las mantienen, operan y aseguran su calidad de uso.

- **Aviación Civil Extranjera**:

  Promueven la aviación segura a lo largo del mundo. Intercambian información importante con otras organizaciones importantes del ramo, así como provee certificaciones de seguridad a operadores extranjeros.

- **Transporte en Espacio Comercial:**

  Regulan y promueven el espacio aéreo de los Estados Unidos para su uso comercial en transacciones nacionales e internacionales.

- **Investigación, Ingeniería y Desarrollo**:

  Desarrollan y promueven la investigación de sistemas y procesos que mejoren la calidad y la seguridad de los sistemas de navegación y control aéreos, así como

la manufacturación de mejores aeronaves, motores e instrumentos aeronáuticos.

- **Otros programas**:
  Los intereses de la FAA intentan cubrir todo lo posible con la industria de la aviación, y puede ir desde programas de seguridad hasta investigaciones sobre el funcionamiento interno de los aeropuertos, entre otros.

## 2.5. Estándares DO-178

De acuerdo con Bernard Dion y su webinar, "*DO-178C Airbone Software Regulation Standard*" (12), las organizaciones regulatorias como la FAA manejan estándares para garantizar la calidad de los sistemas, así como su adecuado funcionamiento. Cumplir con estos estándares implica lograr una certificación de calidad que garantiza que las aeronaves cuentan con sistemas en óptimo estado.

Existe un grupo de estándares denominados como DO-178. Estos estándares son formalmente denominados como "*Software Considerations in Airborne Systems and Equipment Certification*", según la FFA (13), y son la guía para asegurar la seguridad del funcionamiento del software dependiendo de su nivel de criticidad. Actualmente están en uso el DO-178B y el DO-178C.

El B ha estado en uso por más de 20 años y toma en cuenta las entradas, las restricciones y los requerimientos de los clientes. Fue escrito para usarse como guía a la hora de determinar si el software creado puede desempeñarse seguramente.

El C está reemplazando poco a poco al B desde finales del 2011, donde puntos ambiguos fueron mejor descritos e incluye las nuevas técnicas de desarrollo de software que han ido apareciendo desde que el B fuera lanzado en 1992.

## 2.6. Criticidad de Software

La criticidad del software es determinada por el Proceso de Aseguramiento de Seguridad (ARP461) (14) mediante la examinación de los efectos del comportamiento de las condiciones del sistema. Las fallas ocasionadas por la mala implementación o ejecución del sistema y sus efectos sobre la aeronave, sus tripulantes y pasajeros determinan su criticidad.

La criticidad puede ser categorizada de la siguiente manera:

- **Catastrófico (A)**:
  Una falla de este tipo es mortal. Puede causar desde la perdida material total de la aeronave hasta vidas humanas.

- **Peligroso (B)**:
  Este tipo de fallas tiene un alto impacto negativo en el desempeño total de la aeronave, así como introduce obstáculos difíciles a la hora de tripularla. Puede causar daños graves materiales y serios daños físicos a los pasajeros.

- **Mayor (C)**:
  Las fallas mayores reducen significativamente el desempeño de los tripulantes dentro de la aeronave y puede provocar el descontento de los pasajeros.

- **Menor (D)**:
  Las fallas de este tipo difícilmente ponen el peligro la aeronave y pasajeros, a lo mucho podrían implicar un cambio de ruta o un retardo a la hora de partir.

- **Sin efecto (E)**:
  Tal y como lo dice su nombre, estas fallas no tienen ningún impacto en la seguridad de la aeronave y sus pasajeros.

## 2.7. Verificación de software

El organismo de certificación de *testers*, ISTQB (15), define la verificación de software como el proceso de bajo nivel donde se asegura que el sistema creado fue diseñado

cumpliendo con los requerimientos especificados por el cliente desde la etapa de Análisis.

Esta actividad empieza desde la etapa de Desarrollo y la mejor forma de saber si se está cumpliendo con sus objetivos, es responder la pregunta: **¿Estoy desarrollando el producto correctamente?**, es decir, los estándares se están cumpliendo y los requerimientos se están satisfaciendo correctamente.

La verificación ayuda a entender el sistema desarrollado desde las etapas iniciales y a bajar el efecto de los defectos en las últimas etapas de desarrollo, reduciendo las posibilidades de fallas.

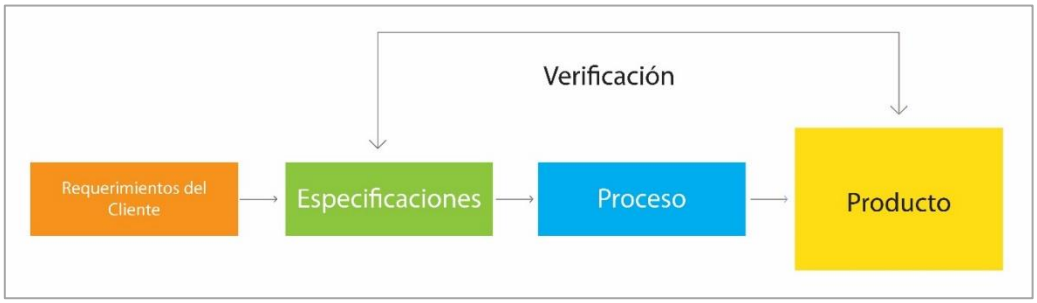


*Figura 3 - Verificación en el ciclo de vida de software.*

## 2.8. Validación de software

ISTQB (16) define la validación de software como el proceso de alto nivel que determina si el sistema cumple con los requerimientos establecidos, cumple con las funciones establecidas y con los objetivos de la organización, así como con las necesidades de los clientes. El estándar *IEEE-610-1990 - IEEE Standard Computer Dictionary* lo define como "el proceso de evaluación de software que se ejecuta durante o al final del proceso de Desarrollo de un sistema de software para determinar si este cumple o no con los requerimientos especificados".

Este proceso ayuda determinar si el sistema de software que fue solicitado por el cliente satisface por completo sus necesidades y se lleva a cabo al final de la etapa de Desarrollo, justo después que los procesos de verificación fueron completados. Busca responder la pregunta: **¿Estoy desarrollado el producto correcto?**

En esta etapa se realizan pruebas una vez que el sistema fue terminado, las pruebas se realizan por completo en el ambiente final donde el sistema desempeñara sus funciones.

Un sistema puede pasar su proceso de verificación con éxito, pero no necesariamente pasar el de validación, esto sucede debido a que es construido bajo las especificaciones establecidas, pero estas especificaciones no determinan con exactitud los requerimientos del cliente. También puede pasar que algún requerimiento o especificación fue malentendido a la hora del Desarrollo provocando que su ejecución no sea la esperada.

Durante la validación del sistema, si un resultado no esperado es encontrado en los resultados obtenidos de las pruebas, el resultado debe ser reportado como un defecto. No todos los defectos provocarán un malfuncionamiento, pero si lograrán el objetivo de validación, que es asegurar que el sistema hace lo que debe de hacer.

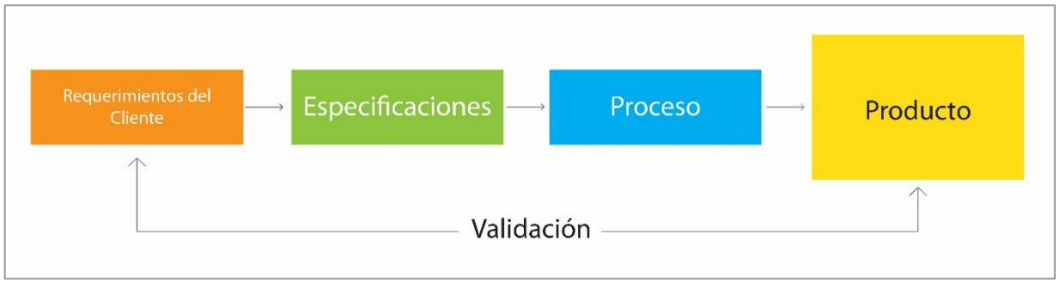


*Figura 4 - Validación en el ciclo de vida de software.*

## 2.9. Técnicas de Pruebas de Software

Eduardo A. Rodríguez Tello define en su conferencia "Estrategias y técnicas de prueba del software" (17) que las pruebas de software son divididas en diferentes técnicas para su correcta utilización y para proveer la información que los desarrolladores necesitan del sistema para mejorarlo.

Existen dos técnicas básicas para diseñar *test cases*:

- **Caja Negra o Estructurales** ('*Black Box*', en inglés):

Se centran en comparar las entradas con las salidas, es decir, sin detenerse a analizar cómo se procesan, esperan ver las salidas esperadas con las entradas actuales que fueron introducidas.

Los *testers* no tienen acceso al código ni a su lógica interna. Tienen por objetivo probar que el sistema desarrollado cumpla con las funciones para las cuáles fue creado, para esto deben seleccionar los datos de entrada que usarán para poder ver determinadas salidas.

Las pruebas de Caja Negra permiten crear *test cases* que:

- Encuentren errores de funciones mal implementadas o faltantes.
- Encuentren errores de interfaz o de bases de datos.
- Encuentren errores de estructuras de datos, comportamiento e inicialización.

- **Caja Blanca o Funcionales** ('*White Box*', en inglés):
  Se centran en analizar la correcta implementación de las funciones y clases en el código. Analizan la lógica interna del sistema y hacen énfasis en la reducción de errores internos.

  Para poder elegir las pruebas de Caja Blanca a aplicar, los *testers* deben decidir qué elementos estructurales del sistema requieren probar y qué información desean obtener de ellas.

  Las pruebas de Caja Blanca permiten crear *test cases* que:

  - Garanticen que todas las rutas posibles dentro de la lógica sean probadas al menos una vez.
  - Las decisiones lógicas ejecuten sus lados falso y verdadero.
  - Se ejecuten los ciclos en sus límites operacionales.

### 2.9.1. Tipos de prueba

Dependiendo de lo que se quiere probar y el fin con el que se va a probar, las pruebas son divididas en diferentes tipos. Cada tipo se enfoca en un objetivo en particular de cada funcionalidad o componente del sistema. Dependiendo de estos objetivos, los *test procedures* son creados.

Según ISTQB (18) (19), los tipos de pruebas que existen son:

- **Pruebas funcionales**:

  Este tipo de pruebas verifican acciones o funciones específicas del código para saber si su funcionamiento cumple con los requerimientos, sin embargo, no es posible saber a través de ellas si el cliente final puede hacer uso de esos componentes o funcionalidades sin problemas.

  Estas pruebas se hacen bajo dos posibles perspectivas:

  - **Con base en requerimientos**:

    Los requerimientos van tomando prioridad dependiendo de los riesgos que presenta su posible malfuncionamiento al sistema en total. Esta categorización asegura que los requerimientos más importantes y más críticos sean incluidos en los *test procedures*.

  - **Escenarios de negocios**:

    Los escenarios más utilizados en el entorno de negocios donde el sistema creado se desenvolverá día a día. Hace uso de los procesos del negocio para crear los *test procedures* y busca encontrar deficiencias en su funcionamiento directo con los clientes finales. (20)

- **Pruebas no funcionales**:

  ISTQB (21) define a las características de calidad del sistema como los objetivos de este tipo de pruebas. No van referenciadas a aspectos de funcionalidades o acciones, sino a atributos propios del sistema como desempeño y compatibilidad, de seguridad, escalabilidad, cumplimiento de regulaciones, etc.

Estas pruebas pueden ser ejecutadas al mismo nivel que las funcionales y ayudan a medir y comparar resultados de pruebas contra aspectos de alto nivel del sistema.

- **Pruebas estructurales**:

  ISTQB (22) considera que este tipo de pruebas va dirigido a la estructura del sistema y son 100% pruebas de *White box*, ya que están interesadas en saber qué es lo que está pasando dentro del sistema, una visión completa de la lógica implementada.

  Al querer conocer el funcionamiento de la lógica tal cual, los *test cases* creados deben ejecutar la lógica cuantas veces sean necesarias sin importar si el objetivo real de esa funcionalidad.

  Las pruebas estructurales son usadas a cualquier nivel de pruebas, incluso en pruebas de sistema y aceptación, aun cuando las estructuras no sean tan directas a la arquitectura del sistema.

### 2.9.2. De unidad

Se le conoce como **unidad** al pedazo de código más pequeño en el que se puedan realizar pruebas; estos pueden ser clases, métodos o procedimientos e interfaces. Las pruebas de unidad o '*unit testing*' (en inglés), de acuerdo con el ISTQB (23), es el tipo de pruebas donde cada unidad de código es probada individualmente con el fin de probar su funcionamiento independiente de las demás unidades.

Este tipo de prueba es realizada, usualmente, por los mismos desarrolladores de software (aunque esto no es necesario, pueden ser desarrollado por *testers* que no estuvieron involucrados en el desarrollo del sistema) para asegurarse de que el código desarrollado está comportándose como está especificado en los requerimientos y el diseño.

Al ser pruebas por unidad de código, las entradas utilizadas y los resultados esperados difieren de unidad a unidad, ya que dependen de la lógica de cada una. Las pruebas de unidad son pruebas de Caja Blanca y son realizadas antes de las pruebas de Integración.

Este tipo de pruebas dan la certeza de que el diseño fue bien implementado o el código bien escrito, y ayudan a notar si hará la manutención del mismo mucho más sencilla. La mejor manera para empezar a confiar casos de prueba de unidad es escribir casos que fallen a la primera, pero que vayan a pasar al momento en que al código se le hagan los cambios pertinentes. Si el caso de prueba sigue fallando a pesar de los cambios en código, entonces el caso está mal. Es por esto mismo que es importante ver ambos resultados, cuando falla y cuando pasa.

Los casos de pruebas de unidad deben ser fáciles de leer, pero esto no implica que deban repetir la lógica del código tal cual, sino que emulen la funcionalidad del código de manera que se puedan encontrar defectos. Una buena forma de hacer esto, sería usar constantes y valores finales, de esta forma es más fácil detectar si algo no está funcionando correctamente. Un dato importante a considerar es el de siempre usar valores dentro del rango aceptable de la variable a probar y después, a manera opcional, valores aleatorios y de estrés. Hay que cubrir el código con los casos de pruebas necesarios que realmente lo prueben y no solo lo ejecuten.

Es importante recordar que no debemos eliminar casos de prueba que ya existan solo por querer aumentar su complejidad, la única razón válida para hacer esto es si el diseño o el código sufrieron un cambio radical que conlleve la actualización o eliminación de las pruebas.

Las pruebas de unidad ayudan a encontrar defectos a una etapa temprana en el desarrollo de software y en consecuencia, en mantener bajo los costos que implicarían resolver esos defectos si fueran encontrados en etapas más adelantadas del desarrollo.

### 2.9.3. De integración

Para el ISTQB (24), el enfoque de las **Pruebas de integración** es ver el comportamiento de varios módulos, componentes o interfaces del sistema interactuando en conjunto, como uno solo, intercambiando información y ejecutando algoritmos al mismo tiempo. También puede darse entre la unión de sistemas y una vez que dos o más partes se unen en uno, se hacen pruebas de integración para corroborar su ensamblaje.

Una vez que dos partes diferentes se han unificado sin defectos, la integración ha sido completada.

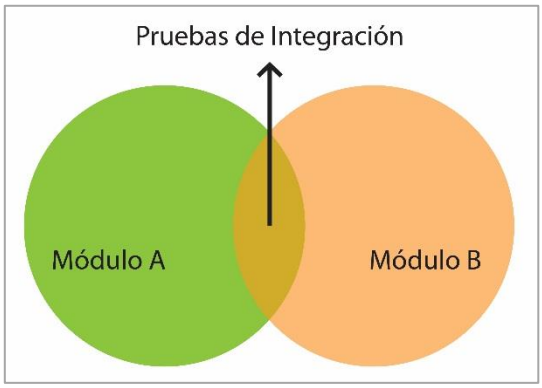


*Figura 5 - Pruebas de integración.*

ISTQB (24) considera que este tipo de pruebas son realizadas por un equipo especializado en integración, y suelen seguir dos posibles estrategias de procesamiento:

- **De arriba-abajo** (*Top Down*, en inglés)*:*

  Esta estrategia comienza por la prueba por el componente principal, sustituyendo por *stubs* aquellos componentes o funciones que llama. Los *stubs* son llamadas ficticias a los componentes que llaman los componentes integrados que se están probando, con el fin de simular que es todo el sistema el que se está probando.

  Conforme avanzan las pruebas, los *stubs* son sustituidos por los componentes reales, siguiendo la jerarquía de control de la arquitectura del sistema, yendo de arriba hacia abajo, es decir, avanzando de los componentes de mayor nivel hacia los de bajo nivel.

Los resultados que arrojan las pruebas que siguen esta estrategia son más consistentes porque la integración es hecha en un entorno simulado muy parecido al real.

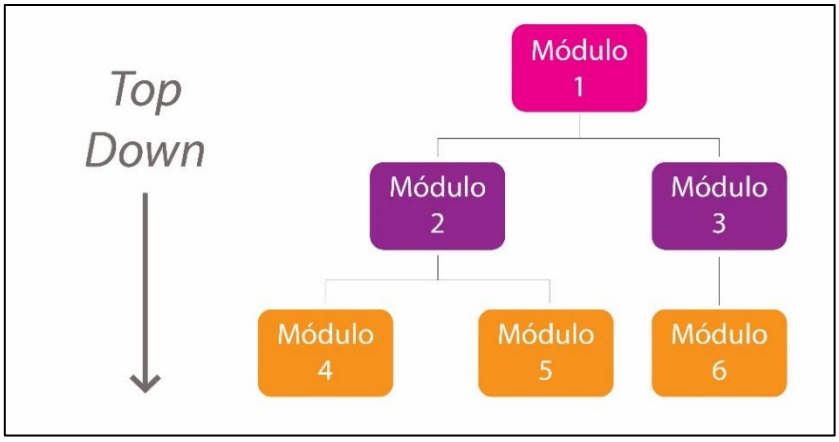


*Figura 6 - Descripción gráfica de 'Top Down'.*

- **De abajo-arriba** (*Bottom Up*, en inglés)*:*
  La integración comienza por los componentes que no necesitan llamar a ningún otro, finalizando en los principales, es decir, avanzan de los componentes de bajo nivel hacia los de mayor nivel.

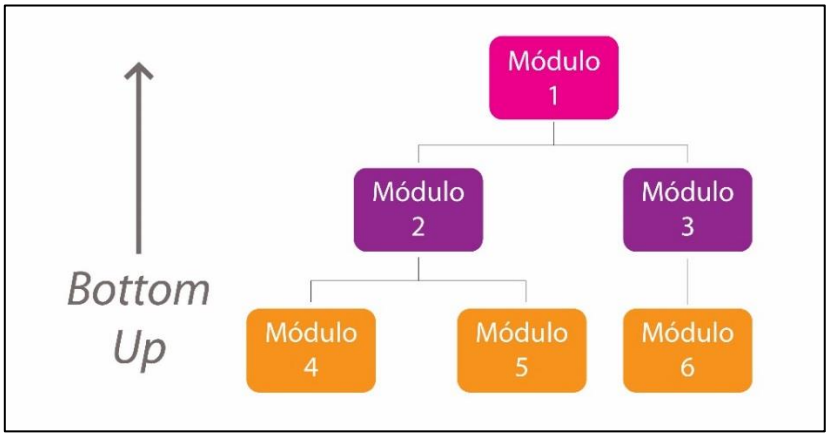


*Figura 7 - Descripción gráfica de 'Bottom Up'.*

Esta estrategia hace uso de *drivers* para sustituir a los componentes que llaman al sistema bajo prueba. Esto ayuda a que se alcance la eficiencia al mismo tiempo que se cumplan todos los requerimientos.

Las pruebas pueden ir a la par al desarrollo, pero esto implica que muchos defectos sean encontrados hasta el fin del desarrollo, ya que los componentes se van creando conforme se va avanzando y pueden no conectarse al resto del sistema tan fácilmente como se creyó al comienzo, lo que implica un aumento en costos y retraso de fechas de entrega.

- **Integración Big-Bang**:

  Para hacer las pruebas de integración *Big-Bang*, según ISTQB (24) (25), todos los módulos o componentes deben ser integrados simultáneamente en el sistema que conformarán. Para poder hacer está integración, todos los módulos no debieron haber sido probados independientemente unos de otros, así que no hay manera de saber si estos funcionan correctamente o no.

  Este tipo de pruebas es muy complicado, ya que no dice con certeza dónde ni cómo surgieron los errores, así que, para encontrarlo, los desarrolladores deberán ir a través de los módulos que pudieron haber ocasionado el error.

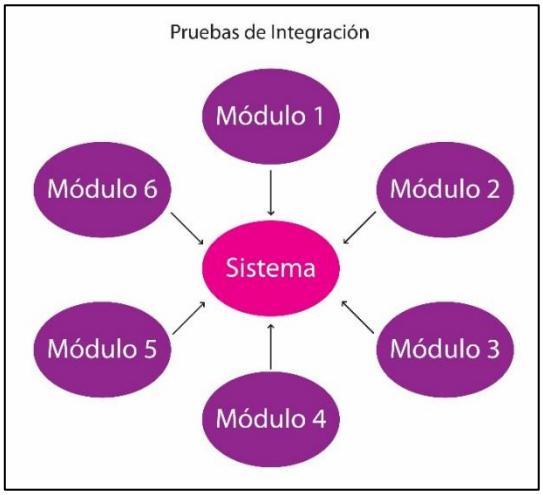


*Figura 8 - Descripción gráfica de pruebas de integración 'Big-Bang'.*

- **Pruebas Incrementales**:

  Según el artículo "*What is Incremental Integration Testing*?", liberado por la compañía Exforsys Inc. (26), la integración y las pruebas toman lugar basándose

en funcionalidades que van integrándose una a una mientras los *tests procedures* son creados función por función y al fusionarse.

Los *testers* integran los módulos uno a uno usando *stubs* y *drivers* para cubrir los defectos que se van encontrando por módulo, los cuales deben ser arreglados antes de que la integración continúe para así evitar la propagación de defectos a lo largo de todo el sistema. Es por esta razón que este tipo de integración va de la mano con pruebas de unidad.

### 2.9.4. De regresión

Tal y como explica Margaret Rouse en su artículo "*Regression Testing*" (27) , puede suceder que las soluciones a los defectos encontrados hayan enmascarado otros posibles defectos o incluso haber introducido al código nuevos defectos. Para detectarlos, existen las **Pruebas de regresión**, las cuales buscan verificar que las modificaciones dentro del código o en el entorno donde se está ejecutando el sistema no causaron efectos no deseados que comprometan su correcto funcionamiento.

Estas pruebas son funcionales y por lo general son realizadas de manera automática con el fin de que su ejecución sea más rápida ya que deben ser ejecutadas varias veces para garantizar que los posibles defectos no existen, al igual que los defectos ya corregidos.

Los casos de pruebas de regresión que se crearon para una versión específica de software, se usan para probar la versión más nueva, con tal de asegurar que las funcionalidades que no fueron modificadas en esta nueva versión sigan funcionando correctamente.

Las pruebas de regresión son incrementales, es decir, cuando hay un nuevo set de pruebas, se le agregan las pruebas del set anterior para garantizar que siguen funcionando como deben ser. Si es necesario, se agregan *test cases* para que al final se tengan todos los casos necesarios para probar la funcionalidad.

### 2.9.5. De aceptación

Para el ISTQB (28), una vez completadas todas las etapas del ciclo de vida de software y los defectos encontrados gracias a los otros tipos de prueba fueron reparados, es momento de entregar el sistema creado al cliente para su aceptación. Se busca establecer confianza entre el sistema creado y el cliente, asegurando que el sistema funciona cómo fue requerido, mediante una serie de pasos que verifican que el sistema cumple con los objetivos para los cuales fue creado.

Puede ser considerado un tipo de validación, por lo que su planificación debe realizarse en etapas tempranas del Desarrollo del sistema, con el objetivo de utilizar los resultados como indicador de validez. Sus pruebas son funcionales de caja negra y se dividen de la siguiente manera, de acuerdo al artículo "*5 types of user acceptance tests - the perfect UAT framework*" de Thomas Peham (29):

- **Pruebas Alpha**:
  Este tipo de prueba normalmente toma lugar en el mismo entorno donde fue desarrollado el sistema y son llevadas a cabo por los mismos desarrolladores o grupos selectos de posibles usuarios, quienes buscan defectos potenciales que comprometan la confiabilidad del sistema.

- **Pruebas Beta**:
  También conocidas como **Pruebas de campo**, este tipo de pruebas son desarrolladas en un entorno controlado por el mismo cliente e involucra la participación de grupos más grandes de posibles usuarios, quienes otorgan una retroalimentación para mejorar el sistema.

  Tanto las pruebas **Alpha** como las **Beta**, se realizan mucho antes de que el sistema llegue al cliente final.

- **Prueba de aceptación de usuario** (*User acceptance test*, en inglés):
  Esta prueba de aceptación se enfoca principalmente en la funcionalidad del sistema, validando su uso en los escenarios elegidos por el cliente. Este tipo de

prueba es realizada entre el cliente y los líderes técnicos que llevaron el proyecto en la fase de Desarrollo.

- **Prueba de aceptación operacional** (*Operational acceptance test*, en inglés):
  Estas pruebas determinan si el sistema está listo o no para ser ya distribuido para uso del cliente final; por esto mismo, también reciben el nombre de **Prueba de aceptación de producción** (*Production acceptance test*, en inglés).

  Estas pruebas pueden involucrar pruebas de restauración y diversos riesgos a los que el sistema podría estar expuesto. También involucran las pruebas de las tareas de mantenimiento y vulnerabilidades en la seguridad del sistema.

- **Prueba de aceptación contractual** (*Contract acceptance test*, en inglés):
  Son pruebas realizadas con base en los criterios especificados en el contrato que una vez se estableció entre el cliente y el equipo de desarrollo del sistema.

- **Prueba de conformidad** (*Compliance acceptance test*, en inglés):
  Estas pruebas son realizadas contra todas las regulaciones legales, gubernamentales y de calidad a las que el sistema desarrollado está enfrentado. Estas regulaciones se establecen considerando el entorno de ejecución y uso del sistema, así como su criticidad y tipo de clientes finales.

Las pruebas de aceptación son útiles porque capturan los requerimientos de una manera directa a su verificación, es decir, tal cual deben ser implementados para funcionar correctamente y ayudan a identificar defectos que pudieron ser pasados por alto en las pruebas de unidad o integración.

## 2.10. Metodologías Ágiles

En febrero de 2001, un grupo de representantes de *Extreme Programming*, Scrum, *Pragmatic Programming* y otras metodologías se reunieron para buscar una solución a la necesidad de aligerar las ya existentes metodologías de desarrollo de software.

De esta reunión, surgió "*The Agile Alliance*", y juntos crearon el "*Manifiesto for Agile Software Development*" (30) (31), en el cuál se establecen principios y valores para desarrollar software de la mejor manera colaborativa.

Estos principios son:

1. La máxima prioridad es satisfacer al cliente con entregas continuas de software totalmente funcional.
2. Agile es capaz de aceptar cambios de alcance y requerimientos en cualquier punto del desarrollo, esto con el fin de ayudar al cliente a mantenerse con ventaja competitiva.
3. Entregar software funcional frecuentemente. Estas entregas puede ser cada semana o cada mes, siempre prefiriendo entregar lo más pronto posible.
4. Equipos administrativos y de desarrollo deben trabajar en conjunto todos los días a lo largo del proyecto.
5. Hacer crecer los proyectos alrededor de gente motivada por su trabajo. Es importante crear un entorno de confianza, apoyo y comodidad para los involucrados en el proyecto.
6. El método de comunicación más eficiente y efectiva es cara-a-cara.
7. El avance del proyecto se mide por cada pedazo de software funcional.
8. Los procesos ágiles promueven un entorno de desarrollo sostenible. Todos los involucrados deben mantener paso constante.
9. La atención constante a la excelencia técnica y el buen diseño mejora la agilidad de desarrollo.
10. La simplicidad es esencial.
11. Las mejores arquitecturas, requerimientos y diseños surgen de equipos bien organizados.
12. A intervalos regulares, el equipo debate maneras de volverse más efectivo.

Estos principios ayudaron a los creadores a darse cuenta que hay situaciones que se deben siempre cuidar y por eso establecieron los siguientes valores:

- Se debe valorar a los individuos y sus interacciones, por encima de los procesos y las herramientas.
- El software funcional debe tener más peso que la documentación exhaustiva.

- Es más importante la colaboración con el cliente que la negociación contractual.
- La respuesta al cambio es más valiosa que el seguimiento de un plan.

Agile se ha ido desarrollando a través de varias metodologías y busca el mejor balance entre recursos. En metodologías de desarrollo más tradicionales, un proyecto puede tomar mucho tiempo en ser completado y el cliente podría no llegar a ver nada del sistema requerido hasta el final del desarrollo. Con Agile, las iteraciones de desarrollo suceden en menor tiempo y al final de cada iteración, el cliente puede ver una parte del sistema ya funcionando.

### 2.10.1. Test Driven Development

Agile hace uso de métodos de pruebas de software que incluye hacer las pruebas antes de escribir el código. Uno de estos métodos es el *Test Driven Development* (TDD, por sus siglas en inglés).

Este método avanzado de pruebas, definido por el ISTQB (32) y complementado por Roy Osherove en su plática "*Understanding Test Driven Development*" (33), consiste en hacer pruebas de unidad automatizadas que lleven a la creación del código funcional. El resultado de usar este tipo de pruebas es el poder ejecutarlas a cualquier momento y obtener resultados que ayuden a los desarrolladores a crear más y más funcionalidades del sistema en proceso.

TDD se basa en la consigna "Rojo, Verde y Refactoriza" en la que cada uno explica la técnica de este método:

- **Rojo**: Crear un *test case* y hacerlo fallar.
- **Verde**: Crear un *test case* y hacerlo pasar de todas las maneras posibles. Estresarlo si es necesario para que eso ocurra.
- **Refactoriza**: Mejorar el código de manera que los *test cases* pasen con naturalidad.

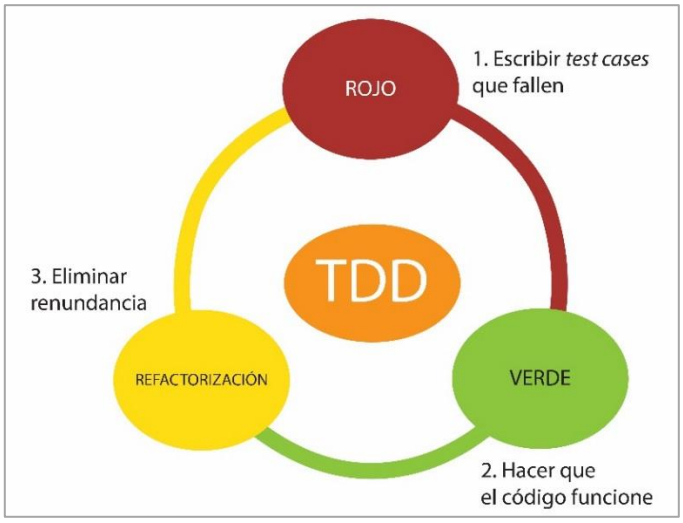


*Figura 9 - Ciclo de Test-Driven Development (TDD)*

Con esta consigna, TDD pretende que haya retroalimentación constante sobre cada componente y garantizar que su funcionalidad sigue siendo la adecuada, además de forzar que haya un análisis y diseño exhaustivo que lo garantice desde el principio.

Las pruebas de aceptación bajo este método son creadas al mismo tiempo que los requerimientos, es decir, antes de ser codificados ya fueron probados. Se usan los *test cases* fallados para dar retroalimentación más rápida sobre los requerimientos que no funcionan como deberían, haciendo que las pruebas y los requerimientos estén interrelacionados.

### 2.10.2. Scrum

Basados en la "*Scrum Reference Card*" de Michael James en conjunto con las definiciones provistas por el ISTQB (34) (35), se puede decir que **Scrum** es un marco de referencia Ágil utilizado para el desarrollo incremental de software entre uno o más equipos multifuncionales, organizándolos desde siete personas, cada uno provisto de roles estructurados, juntas específicas, reglas y artefactos. Los equipos involucrados son responsables de crear y adaptar sus procesos de desarrollo e ingeniería a este marco.

Usando Scrum, el software es creado mediante iteraciones ('*sprint*', en inglés) de misma longitud de tiempo, que puede ser de entre 1 a 4 semanas, con el fin de dar entregables con mayor regularidad. Cada iteración da como resultado

funcionalidades que se van integrando al sistema, que se va completando con el paso de las iteraciones, de ahí que se diga que es incremental. Cada funcionalidad integrada debió haber sido probada y debe funcionar correctamente.

Scrum tomó base del modelo de desarrollo Cascada, del que tomó sus 5 fases (Véase Cap. 2.1) y las adaptó para que se ejecutarán todas en menor tiempo (lo que duré la iteración) y así poder dar retroalimentación en corto plazo que esperar a la entrega del sistema completo. La duración de cada iteración es establecida por las capacidades del equipo de desarrollo y una vez acordada su fecha de fin, es inamovible.

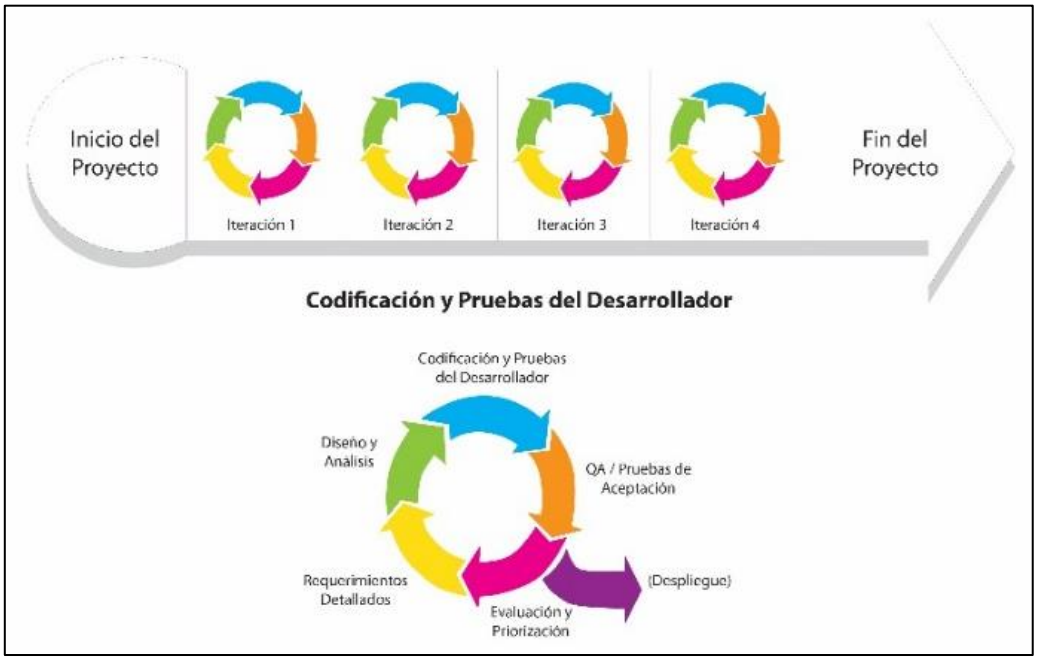


*Figura 10 - Modelo de desarrollo Scrum*

Lo recomendable es que los *sprints* sean de duración corta, reforzando así la importancia de una buena estimación en cuando a cantidad de trabajo y el tiempo requerido para hacerlo.

Para la estructuración correcta de cada sprint, existen roles y juntas específicas que van estableciendo la carga de trabajo a los recursos disponibles.
Siguiendo con la "*Scrum Reference Card*" de Michael James en conjunto con el ISTQB (34) (35), los roles se dividen en:

- ***'Product owner'***:

  El Product owner es la única persona responsable de maximizar el retorno de inversión[3] (ROI, por sus siglas en inglés) del esfuerzo de desarrollo; controla la visión completa del sistema a entregar y del proyecto en el que se está desarrollando y re-prioriza la carga de trabajo ('*backlog*', en inglés) para ajustarlo a las expectativas a largo plazo del proyecto decidiendo si continuar o no con el desarrollo.

  A pesar de que es un rol con mucho liderazgo, puede contribuir como un miembro más del equipo.

- **Equipo de Desarrollo de Scrum** ('*Scrum Development Team*', en inglés):

  Los miembros del equipo deben contar con habilidades de *testing*, codificación y análisis. En conjunto, son auto-organizados, sin ser esclavos de roles asignados externamente. El éxito del equipo tiene mayor probabilidad si el involucramiento es a largo plazo y de tiempo completo, por lo que se evita el movimiento de personas a otros equipos fuera del proyecto en el que están trabajando.

  Tal y como lo explica Lyssa Adkins en su vídeo "*The High Performance Tree*" (36), *Scrum* se basa en valores como el compromiso, enfoque, coraje, respeto y adaptabilidad que busca pasar a los miembros de cada equipo, creando el "Árbol de Alto Rendimiento", donde el crecimiento es tanto individual como grupal y poco a poco transforma al equipo en un grupo colaborativo capaz de tomar decisiones importantes, motivados por la confianza.

- ***'Scrum Master'*:**

  El *Scrum Master* facilita el proceso eliminando obstáculos y creando un ambiente propicio para la auto-organización manteniendo visibles los artefactos de *Scrum*. También protege al equipo de interferencias externas y distracciones para mantener el ritmo de entrega.

[3] ROI, '*Return of investment*', en inglés. Ver Glosario para su definición.

Sin embargo, no tiene autoridad en la gestión del equipo ni puede tomar ninguna decisión, a pesar de su papel de liderazgo facilitando las reuniones que *Scrum* establece.

Pero estos roles no tendrían sentido, si no hubiera artefactos que facilitarán el trabajo para todos los involucrados. El propósito de estos artefactos es cumplir con todos los objetivos para los que *Scrum* fue implementado.

Los artefactos de *Scrum* son:

- ***Backlog* del Producto** ('*Product Backlog*', en inglés)**:**
  Es la lista ordenada de funcionalidad deseada y es visible para todos los interesados. Es constantemente priorizada y los ítems superiores son más granulares que los inferiores.

- **Ítems del *Backlog* del Producto** ('*Product Backlog Item*', en inglés)**:**
  Los PBIs (por sus siglas en inglés) especifican el qué de una funcionalidad especificada por el cliente y son divididos en *User Stories.*

- ***Backlog* del *Sprint*** ('*Sprint Backlog*', en inglés)**:**
  Son los ítems elegidos del *backlog* del producto para ser trabajados en el sprint. Este alcance es fijo y son establecidos por el equipo de desarrollo.

- **Tareas del *Sprint*** ('*Sprint Task*', en inglés)**:**
  Son las actividades que especifican el "cómo" alcanzar el "qué" del PBI.

- ***'Story Points'***:
  Son las unidades relativas utilizadas por el equipo para estimar sus esfuerzos. Esta unidad es dada preferentemente en relación a las horas de trabajo, pero asignadas a un número de la serie de Fibonacci.

- **Historias** ('*User Stories*', en inglés):
  Son los enunciados concisos que capturan los requerimientos desde el punto de vista de usuario explicando su "qué", "quién" y "cómo".

- ***'Definition of Done'*:**
  Es una lista con todos los criterios que deben ser considerados para dar por completadas las tareas.

- **Gráfica de avance del *Sprint*** ('*Sprint Burndown Chart*', en inglés)**:**
  Esta gráfica indica el total de horas faltantes de las tareas por hacer en el *sprint*. Se re-estima diariamente permitiendo la auto-organización eficaz del equipo. Algunas variaciones, como agregar personas o modificaciones de alcance, tienden a reducir su efectividad.

- **Gráfica de producto liberado** ('*Product/Release Burndown Chart*', en inglés)**:**
  Esta gráfica muestra el seguimiento del esfuerzo restante del *Product Backlog* de un sprint a otro; para esto, utiliza tendencias históricas.

Las reuniones que establece *Scrum* son:

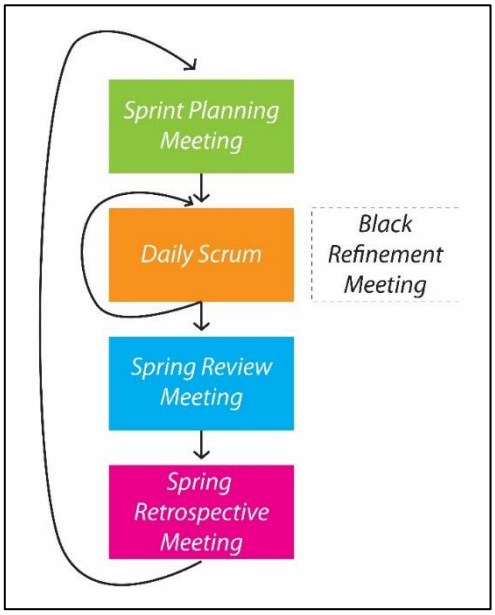


*Figura 11 - Scrum Meetings*

- **Junta de Planeación de Sprint** ('*Sprint Planning Meeting*', en inglés):
  Al inicio de cada sprint, el *Product owner* y el equipo se reúnen para negociar los ítems del *backlog* que trabajarán durante ese sprint. El *Product owner* selecciona los ítems con mayor prioridad y el equipo selecciona la cantidad de trabajo que considera será capaz de terminar en el tiempo estipulado.

  Seleccionando los ítems, el equipo debe hacer la mejor estimación posible. Planificar su capacidad por hora y comparar sus estimaciones con datos reales hace que conforme se avanza en *sprints* se precisen mejor las estimaciones y la velocidad de desarrollo aumente.

  Cada ítem es estimado y asignado y así, arranca el *sprint*.

- **Junta Diaria de Scrum y Ejecución de Sprint** ('*Daily Scrum and Sprint Execution Meeting*', en inglés):
  Cada día, durante la misma hora y el mismo lugar, el equipo de desarrollo comparte por 15 minutos los avances del día anterior, a que se dedicará durante el día y si tiene obstáculos que le impidan seguir con sus tareas asignadas.

  La idea de esta junta es que sea breve, por lo que se recomienda que sea de pie y temas que requieran una discusión a mayor profundidad, se ven fuera de esta.

- **Junta de Revisión de Sprint** ('*Sprint Review Meeting*', en inglés):
  Al finalizar el *sprint*, el equipo vuelve a reunirse, ahora con el fin de da a conocer el estado de cada una de las tareas y mostrar el incremento del sistema a todos los interesados.

  Se define si los ítems fueron completados en su totalidad o no y, los que no, son regresados al *backlog* total del proyecto. El *Product Owner* determina si estos ítems siguen siendo prioritarios o no, ya que entre *sprints* puede cambiar el alcance del proyecto.

- **Junta de Retrospectiva de Sprint** ('*Sprint Retrospective Meeting*', en inglés):
  Cada *sprint* finaliza con una retrospectiva, es decir, con una reflexión sobre su propio avance. Inspeccionan su comportamiento a lo largo del sprint y adoptan lo que funcionó para el futuro.
  Las retrospectivas suelen exponer los obstáculos que se presentaron durante el *sprint* y si fueron o no resueltos y cómo. Esto es aprendido por el *Scrum Master* para poder tomar acciones y seguir facilitando el flujo de trabajo en los *sprints* futuros.

- **Junta de Refinamiento de *Backlog*** ('*Backlog Refinement Meeting*', en inglés):
  En este tipo de junta se estima la cantidad de esfuerzo que se debe invertir para completar los ítems del *backlog* trabajado durante ese sprint.

  El *Scrum Master* puede ayudar al equipo a identificar los ítems de trabajo que tienen valor comercial, mientras promueve una definición rigurosa de "terminado" que incluye las pruebas adecuadas y refactorización.

### 2.10.3. Kanban

Del artículo de Dan Radigan, "*A brief introduction to Kanban*" (37), la palabra '**Kanban**' viene de las palabras japonesas 'Kan', que significa 'Visual' y 'Ban' que significa 'Tarjeta' o 'Tablero', que al unirlas da, en un sentido muy literal, 'Tablero Visual', que es una definición muy cercana a lo qué es Kanban, ya su propósito principal es el de dar un seguimiento muy visual al backlog, de manera que al tenerla así de presente, se pueda ir optimizando y reduciendo los tiempos de entrega de las tareas.

Kanban fue desarrollado por la compañía automotriz japonesa Toyota en los años 1940s, mientras optimizaba sus procesos de ingeniería modelándolos como si fueran estantes de supermercado, en la idea de que solo la cantidad demandada de productos era los presentados, haciendo óptimo el flujo de mercancía. Cuando los niveles de inventario concuerdan con los niveles de consumo, el supermercado podía decir que su eficiencia administrativa se había mantenido o aumentado.

Cuando Toyota introdujo esta propuesta, pidió a sus equipos de trabajo que se pasaran tarjetas entre sí cuando hubieran excedido sus capacidades de trabajo y estuvieran terminadas sus tareas.

Para un equipo de desarrollo de software, esto se traduce como la comparación del trabajo en proceso con la capacidad de los recursos, humanos y administrativos, para cumplir con el actual *backlog*. El equipo solo puede enfocarse en *backlog* activo, y cuando este es terminado, uno nuevo es activado.

Para llevar el control de esto, Kanban cuenta con los siguientes artefactos clave:

- **Tablero de tareas** ('*Kanban board*', en inglés):
  ISTQB menciona que el tablero despliega varios estados que especifican cómo va cada tarea del *backlog* actual. Se usa para priorizar las tareas y evitar cuellos de botella entre estados durante el desarrollo.

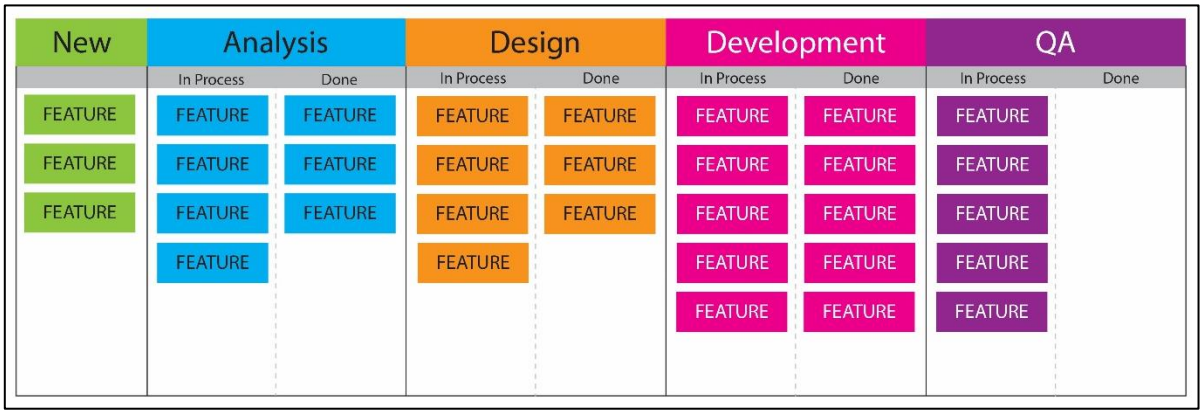


*Figura 12 - Tablero Kanban con estados.*

- **Capacidad de trabajo actual** ('*Work-in-progress Limit*', en inglés):
  Es el número de tareas que pueden estar en cada estado al mismo tiempo sin salirse de control el proceso. Entre menor sea *backlog* que se tenga en progreso, mayor es la atención que se le proporciona.

- **Tiempo de espera** ('*Lead time*', en inglés):
  Es el tiempo que trascurre desde que se abrió la necesidad de tener una tarea necesaria para el sistema, hasta que su desarrollo se completó. Kanban busca que este tiempo sea optimizado efectivamente conforme avanza el desarrollo del *backlog* actual, de manera que cada vez el tiempo sea espera sea menor hasta completar todo.

- **Tiempo de ciclo** ('*Cycle time*', en inglés):
  Es el tiempo que toma una tarea en pasar por todo el proceso hasta ser totalmente cerrada, es decir, que no repetirá ningún estado. No es lo mismo al *Lead time*, ya que calcula cuantas veces esta tarea fue trabajada por el equipo. Es una métrica importante para Kanban, ya que ayuda a pronosticar los tiempos de entrega con mejor precisión.

Usando Kanban, es responsabilidad de todo el equipo de desarrollo asegurar que el trabajo se mueva sin obstáculos a través de los procesos de ingeniería implementados y aun cuando las tecnologías han evolucionado, Kanban sigue siendo el núcleo de los sistemas administrativos que controlan sus tiempos de entrega estrictamente y busca proveer a los equipos de opciones de planificación más flexibles, la generación de resultados de producción más rápida y transparencia en el ciclo de desarrollo.

### 2.10.4. Lean

El Lean Enterprise Institute (38) nos introduce a la metodología Lean, la cual se basa en la idea de maximizar la utilidad mientras se minimizan los desperdicios, es decir, crear más funcionalidades para los clientes con lo menos recursos posibles.

Para lograr esto, Lean piensa que es importante cambiar el enfoque de optimizar tecnologías, recursos y departamentos de manera vertical y por separado a hacer más óptimo el flujo de productos y servicios a través del flujo horizontal entre tecnologías, recursos y departamentos y los clientes.

Eliminar desperdicios a través de este flujo optimizado crea procesos más eficientes, que ocupan menos recursos que en aquellos procesos que están bajo metodologías más tradicionales. Además, los hace más adaptables ante cualquier cambio. Todo se vuelve más simple.

Lean empezó con Henry Ford, el creador de la compañía automovilística Ford, quién se vio estancado en un único modelo de automóvil cuando la sociedad empezó a pedir otro. Esta necesidad social se vio satisfecha por otros competidores y Ford entró en crisis, puesto que no tenía la capacidad de producir otros modelos. Para afrontar esto, fue introduciendo grandes máquinas que avanzaban a grandes velocidades sacando la producción en costos cada vez menores, sin embargo, los tiempos de producción no eran constantes entre los pasos del proceso, lo cual era difícil de mantener inventarios y fechas de entrega.

Justo después de la Segunda Guerra Mundial, otra compañía automotriz, Toyota, pone atención a este proceso y lo perfecciona, creando el "Sistema de Producción Toyota". Este sistema cambió el enfoque de la velocidad de las máquinas, al proceso en sí de producción. Calculaban el volumen que realmente necesitaban producir a la velocidad real de las máquinas, de esta manera sabían cuánto y cuándo.

El sistema de Toyota fue pulido y explicado en el libro "*The machine that changed the world*" de James P. Womack, Daniel Roos y Daniel T. Jones y fue reducido a una guía de 5 pasos para su implementación a cualquier tipo de proceso.

Estos pasos son descritos en los "*Lean Thinking*" de James P. Womack y Daniel T. Jones (39) son:

1. Especificar el valor desde el punto de vista del cliente final por cada familia de productos.
2. Identificar todos los pasos en el flujo de valores para cada familia de productos, eliminando todos esos posibles pasos que no le den valor agregado.
3. Hacer que los pasos de cada proceso implementado fluyan en una secuencia ajustada, para que el producto fluya sin contratiempos hacia el cliente.

4. Los clientes tienen la posibilidad de asignarle valor a cada estado del flujo del proceso.
5. Un proceso perfecto será alcanzado cada vez que se ejecute y conforme cada iteración todos los valores vayan siendo especificados y los flujos vayan siendo implementados sin desperdicios.

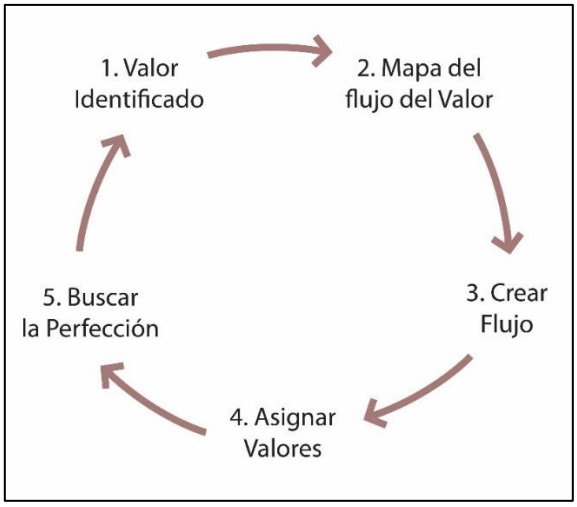


*Figura 13 - Pasos Lean para cualquier proceso.*

### 2.10.5. Extreme Programming

Es también conocido como XP, este tipo de proceso ágil empezó a utilizarse en 1996, creado por Kent Beck y ha ido ganando popularidad porque se enfoca en estresar al máximo la satisfacción del cliente; en vez de entregar ciertas funcionalidades en una fecha establecida, busca entregar todo lo que el cliente quiere y de la manera en que lo necesita tan rápido como salga de producción.

Bajo este método, los desarrolladores son entrenados para aceptar cambios en el proyecto de manera natural y adaptarlos rápidamente a lo ya creado. Para lograr esto, todos los miembros del equipo y los clientes colaboran activamente en la creación del sistema, juntándose y organizándose para llegar a la solución de los problemas tan rápido como les es posible.

Según Don Wells (40), experto en la materia, el *Extreme Programming* consta de las siguientes reglas:

- **Planeación**:
    - Los casos de usos son escritos.
    - La planeación final, libera el calendario de actividades.
    - Se hacen entregas pequeñas.
    - El proyecto es dividido en iteraciones.
    - La planeación de cada iteración empieza al momento en que inicia la iteración.

- **Administración**:
    - Le da al equipo de desarrollo un espacio libre y abierto para dar liberación creativa.
    - Se hacen juntas *stand-up* al empezar el día.
    - El equipo es dinámico.

- **Diseño**:
    - El diseño se enfoca en ser simple.
    - Se hace uso de tarjetas CRC para diseñar el sistema en equipo. **CRC** corresponde a **C**lases, **R**esponsabilidades y **C**olaboración (41) y su objetivo es que entre mayor sea el número de personas involucradas en el diseño del sistema, mayor es el número de ideas incorporadas a este.
- **Codificación**:
    - El cliente siempre está disponible para aportar cualquier idea o respuesta que el equipo de desarrollo necesite.
    - El código debe ser creado para cumplir con los estándares de codificación.
    - Se codifican las pruebas unitarias primero.
    - La creación de código es *Pair Programming* (Véase Cap. 2.10.6).
    - Se integra par en par, por lo que se están integrando unidades de código con frecuencia.

- **Prueba**:
    - Todas las unidades de código deben tener pruebas de unidad.
    - Todas las pruebas de unidad deben haber sido aprobadas antes de ser liberado el código.
    - Pruebas de aceptación son llevadas a cabo a menudo y cada una es documentada.

Y busca mejorar la planeación del desarrollo en las siguientes 5 maneras:

- **Comunicación**: Existe comunicación eficiente entre clientes y programadores.
- **Simplicidad**: El diseño se mantiene simple y limpio para un mejor entendimiento.
- **Retroalimentación**: Existe retroalimentación a lo creado y por crear a cada momento del ciclo de desarrollo del software.
- **Respeto**: El éxito depende en el respeto mutuo entre los miembros del equipo ya que cada miembro aporta contribuciones únicas al equipo.
- **Coraje**: Los desarrolladores tienen la capacidad de responder eficazmente a los cambios en los requerimientos y tecnologías. (35) (40) (41)

### 2.10.6. Pair Programming

Don Wells (42) define el ***Pair Programming*** como un tipo de *Extreme Programming* donde todas las tareas enviadas a desarrollo son creadas por dos perdonas al mismo tiempo, compartiendo recursos. Busca incrementar la calidad del software desarrollado sin impactar los tiempos de entrega, así como hacer ahorros en los costos de producción.

Dos personas compartiendo recursos suelen trabajar más rápido, añadiendo más innovaciones a las funcionalidades y con mayor calidad que trabajando por separado. Estas dos personas deben tener el mismo nivel de experiencia para poder trabajar en conjunto para poder avanzar sin problemas. No busca ser una relación de uno-aprendiendo-del-otro.

## 2.11. Relación Marco Teórico - Investigación

Los conceptos contemplados en este marco teórico fueron pensados para dar mejor explicación a la mejora aplicada a la metodología V y que se busca explicar como resultado de esta propuesta.

Como se puede observar, a lo largo de la historia, se han desarrollado varias metodologías, todas buscando ser eficientes, proporcionando herramientas a los desarrolladores para que la creación de software sea mucho más fácil. Sin embargo, la tecnología ha ido avanzando más rápido de lo que las metodologías evolucionan, por lo que es necesario tomar lo mejor de ellas y adaptarlas al mundo de hoy.

Las metodologías de desarrollo de software están conformadas por muchas etapas y detalles que son necesarios entender para poder aplicarlas satisfactoriamente, por esto se hace énfasis en cada concepto que pueda ayudar a aplicar la mejora propuesta con éxito.

Las metodologías de cascada o la de V suelen consumir tiempo valioso a la hora de implementarlas, mientras que las ágiles buscan que su implementación a la hora de crear software sea de manera más práctica, sencilla, dinámica y rápida. Se puede concluir que existen diferentes tipos de metodologías y formas de implementarlas, pero que dependerá de la naturaleza de cada proyecto cuáles elegir.

Concretándonos únicamente en el software empleado en el campo aeronáutico, debido a los estándares a los que está sometido, así como los órganos internacionales que lo certifican, muchos de los conceptos presentados en este marco teórico, son aplicados en su totalidad a cada proyecto, sin embargo, aun cuando se deben contemplar cada aspecto, la urgencia de entregar a tiempo y sin fallas es cada vez más imperativa, es por esto que surgen cuestiones que se deben resolver, tales como qué metodología es más eficaz o qué tipo de prueba puede cubrir las necesidades del DO-178B.

# Capítulo 3.- Solución propuesta

## 3.1. Investigación previa

Con el fin de poder aplicar la teoría explicada en esta tesina, se realizó una investigación de campo con diversos equipos de desarrollo de software aeronáutico del Centro de Ingeniería Avanzada en Turbo-máquinas de General Electric en la ciudad de Santiago de Querétaro (GEIQ). Los equipos a investigar fueron seleccionados mediante la condición de que usarán (o pretendieran usar) la metodología ágil de *Scrum*.

El objetivo de esta investigación se estableció pensando en la hipótesis de que la correcta elección de metodologías de desarrollo de software tendría un impacto al llegar a la fase de pruebas y la estandarización de las mismas disminuirá en un 20% el número de ocurrencia de fallas en el software durante su ejecución en vuelo. Buscando la manera de conocer cómo los equipos realizan sus esfuerzos de verificación y validación se encontró que ya se busca la estandarización de metodologías de desarrollo en general (y no sólo de pruebas) mediante las metodologías ágiles, por lo que se generó el nuevo enfoque de conocer qué tanto de estas metodologías utilizan, cómo las implementan y qué buscan lograr al utilizarlas, así como que tan efectivas les resultaban.

### 3.1.1. Resultados de la investigación previa

Se encontró que todos los equipos entrevistados comprenden a la perfección los conceptos básico ágiles de *Scrum*, sin embargo, cada uno los aplica diferente y no en su totalidad; lo utilizan únicamente para la organización de *sprints*, pero se sigue trabajando en cascada, lo cual es incorrecto, ya que no produce beneficios en el cliente final, y sus beneficios se quedan localmente.

Como parte de esta investigación, también se encontró que GEIQ trabaja en proyectos que constaron en promedio con 1 000 requerimientos de alto nivel y 1 100

requerimientos de bajo nivel, que a su vez derivaron en 1 100 funciones de código. Para el desarrollo de estos proyectos se estimó 12 meses y para su fase de V&V, se estimó entre 6 y 12 meses.

Siguiendo con un promedio, cada ingeniero de GEIQ trabaja 170 horas de ingeniería al mes, y cada función de código es implementada en 18 horas por promedio, por lo que las 1 100 funciones son desarrolladas en 19 800 horas.

Para la validación y verificación informal, es decir, para una versión preliminar de los requerimientos de alto nivel, se trabajan 60 horas de ingeniería por requerimiento; dando así un total de 60 000 horas invertidas.

Para la validación y verificación informal de los requerimientos de bajo nivel, se trabajan 12 horas por requerimiento; dando un total de 13 200 horas invertidas.

Todo este trabajo se cotiza en 93 000 horas, tal y como se aprecia en la siguiente tabla:

| | # Ítems | Tiempo de ciclo Informal | Costo informal |
|---|---|---|---|
| **Funciones de Código** | 1 100 | 18 horas | 19 800 horas |
| **Requerimientos de alto nivel** | 1 000 | 60 horas | 60 000 horas |
| **Requerimientos de bajo nivel** | 1 100 | 12 horas | 13 200 horas |
| **Total** | | | 93 000 horas |

*Tabla 1 – Cotización inicial del ciclo informal.*

Una vez realizado el V&V informal, se pasa a la fase de formalización, el cual tiene la siguiente cotización:

| | # Ítems | Tiempo de formalización | Costo formalización |
|---|---|---|---|
| **Funciones de Código** | 1 100 | 8 horas | 8 800 horas |
| **Requerimientos de alto nivel** | 1 000 | 40 horas | 40 000 horas |
| **Requerimientos de bajo nivel** | 1 100 | 12 horas | 13 200 horas |

| | | | |
|---|---|---|---|
| **Total** | | | 62 000 horas |

*Tabla 2 - Cotización inicial ciclo formal.*

Por cada prueba de alto nivel, se invierten 40 horas, dando como resultado 40 000 horas; por las de bajo nivel, 12 horas, dando en total 13 200 horas y por la formalización del código se toma 8 horas por función, completando el esfuerzo en 8 800 horas.

Esta formalización conlleva 62 000 horas de ingeniería. El costo total de estos esfuerzos alcanza las 155 000 horas, algo difícil de llevar a cabo en la actualidad.

Si se desarrollara este tipo de proyectos bajo la metodología de cascada, se tendría un costo de desarrollo inicial de 19 800 horas (ver Tabla 1). Considerando que el esfuerzo tiene como tiempo de desarrollo los próximos 12 meses y la disponibilidad de 170 horas por ingeniero, se necesitarían 10 ingenieros.

Una vez pasados estos 12 meses, estos 10 ingenieros entregan a la fase de V&V informal, dónde el costo para llevarlo a cabo es de 73 200 horas (considerando las horas que consumirían el V&V de tanto requerimientos de alto nivel, como de bajo nivel).

Como resultado de esta primera vuelta informal de V&V, la reparación de los defectos y errores encontrados se consumirán 9 900 horas del costo total.

Esta primera vuelta provocará que exista una segunda vuelta, para poder probar el código con los defectos y errores corregidos. En promedio, estas segundas vueltas consumen 36 600 horas en pruebas y derivan en 4 950 para la corrección de nuevos defectos y errores. Se necesitan 18 300 horas para validar y verificar esas nuevas correcciones.

La formalización de la validación y verificación no ha sido considerada aún. Esta formalización suele consumir 53 200 horas y se requieren 5 940 horas para corregir los defectos y errores que surjan. Se necesitan 21 960 horas para validar y verificar esas nuevas correcciones.

En total, la fase de V&V requeriría 203 260 horas para ser completada y 29 590 horas para corregir defectos y errores más formalizar.

Para poder entregar la fase de V&V completa en 6 meses, la empresa necesitaría crear un equipo de 200 ingenieros. Si se tienen 12 meses para trabajar esta fase, el equipo tendría que ser de 100 ingenieros. Ambos escenarios son difíciles de llevar a cabos en la empresa actualmente, donde los equipos más grandes oscilan entre los 50 y 60 ingenieros, todos ellos invirtiendo sus 170 horas en proyectos diferentes.

Cabe mencionar que los 10 ingenieros involucrados en el desarrollo del software no podrían participar en la fase de V&V y podrían no estar disponibles para la etapa de correcciones.

Como se puede apreciar, la empresa cotizó un proyecto en 155 000 horas, pero lo trabajaría en 252 650 horas; es decir, un 63% por encima de lo inicialmente presentado al cliente (Ver Tablas 3 y 4).

| | Costo |
|---|---|
| **Desarrollo informal** | 19 800 horas |
| **Corrección de errores** | 9 900 horas |
| **Corrección de errores (2da. vuelta)** | 4 950 horas |
| **corrección de errores (3ra. vuelta)** | 8 800 horas |
| **Formalización** | 5 940 horas |
| **Total** | 29 590 horas |

*Tabla 3 - Costo de desarrollo en Cascada.*

| | Costo |
|---|---|
| **V&V informal** | 73 200 horas |
| **V&V de la corrección de errores (2da. vuelta)** | 36 600 horas |
| **V&V de la corrección de errores (3ra. vuelta)** | 18 300 horas |

| | |
|---|---|
| **Formalización de V&V** | 53 200 horas |
| **V&V errores en la formalización de V&V** | 21 960 horas |
| **Total** | 203 260 horas |

*Tabla 4 - Costo de V&V en Cascada.*

El objetivo de esta investigación es hacer desarrollo de software usando la metodología "V o de cuatro niveles" de manera ágil, es decir, desarrollar los cuatro niveles del ciclo de vida usando *Scrum*. El objetivo será implementar la metodología con éxito, teniendo como punto crítico un inicio de proyecto sin complicaciones y que se generen ahorros en costos a la hora de llegar a la parte de V&V.

## 3.2. Problemática con Agile

Como resultado de esta investigación de campo, se encontraron los siguientes puntos en común entre los equipos:

- Se utiliza *Scrum* solamente para la organización de los equipos, es decir, los miembros toman roles propios de la metodología ágil como *Scrum Master y Product Owner* y se llevan a cabo las juntas de planeación, retrospectiva y *stand-up*.

- Utilizan *Scrum* para la planeación de su trabajo, es decir, se organizan en *sprints* y mantienen vivo un *backlog* de *ítems*, elegidos por el *Product Owner* y monitoreados por los *Scrum Masters*, a través de historias y *story points*. Estos ítems son trabajados después de que un *Definition of Done* ha sido establecido.

- *Scrum* no es considerado a la hora de desarrollar el software ni para su entrega al cliente. Aunque hay equipos que, si buscan entregar partes del software funcionando, el desarrollo se lleva a través de *sprints* establecidos por etapa, es decir, un *sprint* no contempla todas las etapas del ciclo de vida de software, sino que hay *sprints* para cada una de las etapas sin considerar las demás.

  Como resultado de esto, el software suele entregarse con errores, dado que no fue verificado ni validado al momento de su creación, por lo que el cliente no

ve el valor real de la ejecución de una metodología ágil. Además, se invierte más dinero y tiempo en la corrección y eliminación de errores cuando los *sprints* de verificación y validación se alcanzan.

- Existe una mentalidad poco flexible y renuente de los desarrolladores a cambiar su forma de trabajo. Al pensar en el desarrollo del software que se les asignó, siguen contemplándolo como una metodología en Cascada, sin siquiera considerar la aplicación de la V y muchos menos pasarlo a una metodología ágil. Les es complicado pensar en la ejecución de todas las etapas de desarrollo en poco tiempo o incluso al mismo tiempo.

  En consecuencia, los equipos siguen segregados por funcionalidad y la interacción se da por etapas, dejando a equipos bloqueados por falta de trabajo cuando su etapa no ha sido alcanzada ya que no se puede pasar a la siguiente. Otro aspecto que se ve afectado es que la detección de errores, esta se da muy tarde una vez que el código ya está casi terminado, cuando lo ideal sería identificarlos desde una etapa temprana para corregirlos y aprender de ellos.

## 3.3. Metodología V ágil

Considerando este nuevo enfoque, se propone llevar el ciclo de vida completo con la metodología V de manera ágil, empezando por el análisis de la arquitectura para la definición de *epics* (en caso de ser necesarios), historias y *sprints* hasta el desarrollo y verificación del software, entregando al cliente funcionalidad real libre de errores al final de cada *sprint*, buscando así garantizar su satisfacción.

Se proponen los siguientes pasos para dar arranque al proyecto e irlo llevando bajo una metodología V ágil:

1. **Requerimientos del cliente**:

Solicitar los requerimientos del cliente antes de arrancar la ejecución del proyecto y someterlos a un análisis para asegurar que estos son correctos y completos.
Realizar tantas revisiones como sean necesarias hasta llegar a acuerdos entre las dos partes. Como en toda metodología, el objetivo siempre será satisfacer las necesidades del cliente.

Nota importante: En este punto aún no ha sido creado el equipo ni definido el *backlog*.

2. **Definir arquitectura de alto nivel**:
   Una vez que se tienen los requerimientos del cliente, el arquitecto debe definir la arquitectura en base a ellos, definiéndose sus módulos, bases de datos, protocolos de comunicación, entre otros.

3. **Partir el proyecto en *epics***:
   Una vez definida la arquitectura de alto nivel, el arquitecto y el (los) *product owner(s)* del proyecto deben dividir el desarrollo en *epics*, dependiendo de la complejidad del proyecto.
   Esta división debe ser congruente con las prioridades funcionales establecidas previamente con el cliente. Es importante que al llegar a este punto, la arquitectura esté lo suficientemente estable y sólida, para evitar cambios en ella que afecten etapas ya en ejecución.

   Nota importante: Una de las razones del éxito de las metodologías ágiles es que son fácilmente adaptables a los cambios, por lo cual se debe estar preparado para seguir recibiendo cambios posteriores.

4. **Definir el *backlog***:
   Dividir los *epics* en grupos de funcionalidades priorizados creando *stories*.
   Considerar todos los aspectos de la arquitectura del producto a desarrollar que puedan presentar obstáculos en el momento de su desarrollo. En el *backlog* se

debe incluir todo lo que el proyecto debe entregar al cliente como código, análisis o documentación.

Nota importante: La definición del *backlog* es la parte crucial del desarrollo ágil. Para el desarrollo de software aviónico es importante incluir a CEO[4], *Certification*, DQA[5] y CTHs[6] para considerar todos los aspectos que serán requeridos por la FAA.

5. **Asignar *Story Points*:**
   Después de crear el *backlog* se debe asignar *story points* a cada ítem. Esto solo será un aproximado de los *story points* finales ya que poco a poco se irán refinando.

6. **Partir el *backlog* en *sprints*:**
   Una vez que el *backlog* fue definido se debe de partir en el número de *sprints* que se tienen disponibles para entregar el proyecto. Por ejemplo, si se tiene un año para la entrega y los *sprints* duran un mes se debe entonces partir en 12 *sprints*.

   Partiendo el *backlog* en *sprints* ya se tiene una idea de los *story points* por entregar y se puede definir el tamaño del equipo que se requiere.
   Este grupo de funcionalidades debe ser desarrollado de manera inteligente, por lo que es importante determinar la secuencia más lógica para ir entregando de acuerdo a lo establecido previamente con el cliente.

7. **Definir el *scrum team*:**
   Del equipo de desarrolladores y *testers* que participarán en el proyecto, se definen los grupos interdisciplinarios con los que se trabajará a lo largo de todo el proyecto. Es importante destacar que cada *scrum team* debe ser formado por pocas personas que estén dispuestas a colaborar ampliamente y a ser

[4] CEO, por sus siglas en inglés, *Chief Engineer Officer*.
[5] DQA, por sus siglas en inglés, *Department of Quality Assurance*.
[6] CTH, por sus siglas en inglés, *Controlled Title Holder*.

multifuncionales, es decir, que puedan participar en una o más etapas del ciclo de desarrollo.

8. **'*Definition of Done*' (para cada uno de los niveles)**:
   Establecer los criterios que se considerarán para dar por terminada cada tarea de cada nivel.

9. **Correr los *sprints***:
   El desarrollo empieza. Cada *sprint* conlleva la ejecución de cada nivel de la metodología V, es decir, se empieza desde la documentación de requerimientos hasta la validación y verificación del código recién creado.

   El objetivo de cada *sprint* es tener código funcional y sin errores que pueda ser entregado al cliente.

   Para ver los detalles de cómo correr un sprint con el ciclo completo de vida V ver la sección 3.6.

10. **Aceptación del cliente**:
    El cliente evalúa si todas sus necesidades por funcionalidad mostrada fueron cubiertas. Cambios a nivel arquitectura del sistema o a nivel de administración de proyecto pueden surgir bajo este punto.

    Estas discusiones de aceptación se deberán llevar a cabo al finalizar cada *sprint*.

## 3.4. Estableciendo el proyecto

La forma de establecer los requerimientos del cliente no varía entre metodologías y este caso no será la excepción. Establecer una comunicación constante, directa y abierta con el cliente es sin duda la mejor manera de empezar, mantener y cerrar un proyecto.

Es altamente recomendable que el cliente y el equipo de desarrollo estén en constante comunicación y es válido aceptar cambios de bajo impacto durante la ejecución de los *sprints*.

Las siguientes secciones muestran en forma práctica como aplicar la metodología ágil en un proyecto aviónico con metodología V.

### 3.4.1. Requerimientos del cliente

Para la explicación efectiva de esta mejora, se tendrá como ejemplo[7] de cliente a una manufacturera de aviones que busca modernizar la Pantalla Principal de Vuelo de los *displays* de su flota comercial actualizando únicamente los siguientes aspectos:

| Pantalla Principal de Vuelo | |
|---|---|
| **1** | La Pantalla Principal de Vuelo ('*Primary Flight Display*') debe actualizar los indicadores de velocidad vertical e indicador de altura de manera visual para la fácil interpretación del piloto y copiloto. |
| **1.1** | Indicador de velocidad. |
| **1.1.1** | La barra del indicador de velocidad debe marcar la velocidad actual en verde. |
| **1.1.2** | La barra del indicador de velocidad debe marcar la velocidad máxima permitida en rojo. |
| **1.1.3** | La barra del indicador de velocidad vertical debe mostrar en un *display* numérico la velocidad vertical a la que se está en ese momento. |
| **1.2** | Indicador de altura. |
| **1.2.1** | La barra del indicador de altura debe mostrar la altura seleccionada en verde. |
| **1.2.2** | La barra del indicador de altura debe parpadear y emitir una alarma cuando la |

[7] Este ejemplo fue creado para propósitos meramente académicos y no reflejan requerimientos reales.

| | |
|---|---|
| | altura seleccionada se salga del rango permitido. |
| **2** | Los mensajes serán emitidos de acuerdo a su tipo. |
| **2.1** | Mensajes de mantenimiento. |
| **2.1.1** | Un mensaje de mantenimiento será mostrado cuando un sensor mandé un valor fuera de su rango funcional. |
| **2.1.2** | Un mensaje de mantenimiento será mostrado cuando las horas de operación del motor sean alcanzadas. |
| **2.2** | Mensajes de alerta. |
| **2.2.1** | Un mensaje de alerta será desplegado en la pantalla de *display* cuando el rango de altura se salga del valor permitido. |
| **2.2.2** | Un mensaje de alerta será desplegado en la pantalla del *display* cuando el tanque de gasolina alcance la capacidad mínima. |
| **3** | El motor mandará mensajes al display cuando falle un componente y se requiera tomar medidas de seguridad oportunas. |
| **3.1** | Alerta de *No-dispatch*[8]. |
| **3.1.1** | Un mensaje de *No-dispatch* emitirá una alarma sonora cuando un componente vital falle. |
| **3.2** | Alerta de *Long dispatch*[9]. |
| **3.2.1** | Un mensaje de *Long dispatch* emitirá una alarma sonora de 30 segundos. |
| **3.2.2** | Un mensaje de *Long dispatch* mostrará un contador con las horas restantes permitidas de vuelo. |
| **3.3** | Alerta de *Short dispatch*[10]. |
| **3.3.1** | Un mensaje de *Short dispatch* emitirá una alarma sonora de 15 segundos. |
| **3.3.2** | Un mensaje de Short dispatch mostrará un contador con las horas restantes permitidas de vuelo. |

*Tabla 5 - Tabla de requerimientos del cliente*

### 3.4.2. Arquitectura de alto nivel

Es ampliamente recomendable contar con un diagrama de la arquitectura a alto nivel para tener una visión amplia del producto que se estará trabajando.

[8] *No-dispatch* es un tipo de mensaje de alerta. Ver Glosario para su definición.
[9] *Long dispatch* es un tipo de mensaje de alerta. Ver Glosario para su definición.
[10] *Short dispatch* es un tipo de mensaje de alerta. Ver Glosario para su definición.

Esta visión ayudará a determinar mejor las prioridades de cada requerimiento, así como la creación de los mismos; incluso puede ayudar a determinar qué conocimientos técnicos requieren los miembros del equipo.

Para el ejemplo planteado, se creó el siguiente diagrama:

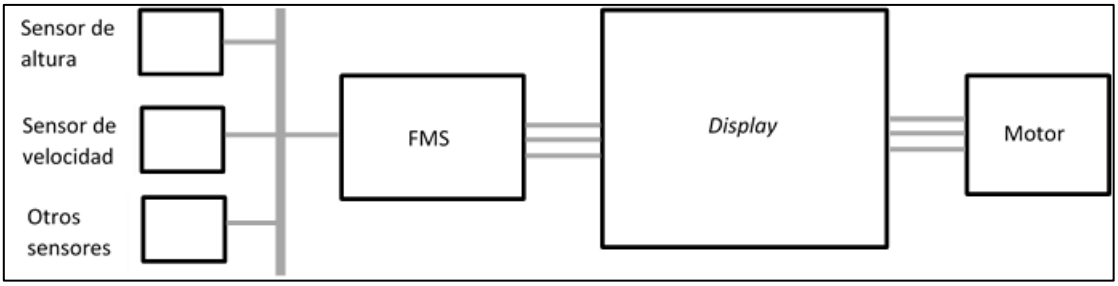


*Figura 14 - Diagrama de arquitectura a alto nivel*

Como se aprecia en el diagrama, existen buses de comunicación involucrados. Esto se traduce a que los miembros de los equipos de sistema y software deben considerar su funcionamiento a la hora de crear diseños, requerimientos y código, por lo que es importante que estén familiarizados con el tipo de estándares de comunicación que el producto requiera.

También se aprecia que las señales de los sensores son procesadas por el Sistema de Administración de Vuelo (FMS[11], por sus siglas en inglés) antes de llegar al *display*, por lo que la lectura y escritura, así como conversión de los datos debe ser considerada.

## 3.5. Planeación

### 3.5.1. Preparación del *backlog*

Una vez que se tienen los requerimientos del cliente y fechas de entrega, se procede a analizarlos y preparar el *backlog* en cual será partido en *sprints* con aceptación del cliente.

[11] FMS, siglas en inglés de *Flight Management System*. Ver Glosario para definición.

El *backlog* es un ente vivo durante toda la ejecución del proyecto, pero es sumamente importante tener la planeación y calendarización del *backlog* de por lo menos los 2 próximos *sprints* a partir del que se está llevando a cabo. Esto porque no se sabe el resultado del *sprint* anterior o si los criterios del cliente van cambiar.

A continuación, se muestran los pasos ideales para la preparación del *backlog*:

**Paso 1. Organización de requerimientos.**

Pasar los requerimientos a una forma organizada (se sugiere una tabla como la que se muestra en Tabla 1, donde se puedan clasificar de acuerdo a su criticidad, complejidad, prioridad o cualquier otro criterio establecido por el cliente.

| *Story*-ID | Grupo | Ítem / *Story* | Tipo | Complejidad | *SP* | Prioridad | Dependencias |
|---|---|---|---|---|---|---|---|
| **Ítem 1** | Indicador de velocidad | Req-111 | | | | | |
| **Ítem 2** | Indicador de velocidad | Req-112 | | | | | |
| **Ítem 3** | Indicador de velocidad | Req-113 | | | | | |
| **Ítem 4** | Indicador de altura | Req-121 | | | | | |
| **Ítem 5** | Indicador de altura | Req-122 | | | | | |
| **Ítem 6** | Mensajes de mantenimiento | Req-211 | | | | | |
| **Ítem 7** | Mensajes de mantenimiento | Req-212 | | | | | |
| **Ítem 8** | Mensajes de alerta | Req-221 | | | | | |
| **Ítem 9** | Mensajes de alerta | Req-222 | | | | | |
| **Ítem 10** | Alarma de *No-dispatch* | Req-311 | | | | | |
| **Ítem 11** | Alarma de *Long dispatch* | Req-321 | | | | | |
| **Ítem 12** | Alarma de *Long* | Req-322 | | | | | |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | *dispatch* | | | | | | |
| **Ítem 13** | Alarma de *Short dispatch* | Req-331 | | | | | |
| **Ítem 14** | Alarma de *Short dispatch* | Req-332 | | | | | |

*Tabla 6 - Organización de requerimientos*

**Paso 2. Documentación.**

Agregar a la planeación, la documentación obligatoria que requiere todo software aplicado en aviación. Ya que durante cada sprint se estarán desarrollando funcionalidades (ya sean nuevas o actualizaciones), surge nueva documentación o la ya existente sufre modificaciones y es por ello que es importante considerarla en cada *backlog*.

| *Story*-ID | Grupo | Ítem / *Story* | Tipo | Complejidad | *SP* | Prioridad | Dependencias |
|---|---|---|---|---|---|---|---|
| **Ítem 15** | Documentación | SDP[12] | | | | | |
| **Ítem 16** | Documentación | SVP[13] | | | | | |
| **Ítem 17** | Documentación | TSP[14] plantilla | | | | | |
| **Ítem 18** | Documentación | Plantilla de diseño | | | | | |
| **Ítem 19** | Documentación | Equipo y herra-mientas | | | | | |

*Tabla 7 - Organización de documentación obligatoria*

**Paso 3. Asignación de prioridades.**

Se asignan las prioridades de cada requerimiento, es decir, se establece cuáles son más importantes a desarrollar primero. Para determinar esto se toma en cuentas las peticiones del cliente, así como el sistema en sí, ya que

[12] Siglas para *Software Development Plan*. Ver el Glosario para definición.
[13] Siglas para *Software Verification Plan*. Ver el Glosario para definición.
[14] Siglas para *Test Specification Procedure*. Ver el Glosario para definición.

puede haber funcionalidades que no puedan existir hasta que otras ya estén conclusas.

La prioridad 0 está lejos de significar poca relevancia, al contrario, se consideran ítems no funcionales, pero necesarios para la creación y documentación de los ítems que si son funcionales.

**Paso 4. Definir las dependencias.**

Las dependencias van ligadas a las prioridades. Como se menciona anteriormente, hay funcionalidades que necesitan de otras para existir o, también, completar tareas.

La importancia de definirlas es para evitar posibles obstáculos durante la ejecución. Podría suceder que se empiece un *story* que deba ser bloqueado a mitad de *sprint* por falta de funcionalidades aún inexistentes, creando así un riesgo para toda la ejecución del proyecto.

**Paso 5. Definir la complejidad.**

Determinar qué tan complejo puede ser el desarrollo de las funcionalidades es algo complicado en sí, ya que puede uno caer en subjetividad. La mejor manera de determinar esto, es pensando en las habilidades y conocimiento técnico del equipo, así como en el peso funcional de cada tarea.

Se puede clasificar la complejidad cómo mejor acomode al equipo. Para este ejemplo, se utilizó una denominación sencilla: Fácil, Medio y Difícil.

**Paso 6. Asignación preliminar de *story points*.**

| *Story*-ID | Grupo | Ítem / *Story* | Tipo | Complejidad | *SP* | Prioridad | Dependencias |
|---|---|---|---|---|---|---|---|
| **Ítem 1** | Indicador de velocidad | Req-111 | S[15] | Fácil | 2 | 2 | N/A |
| **Ítem 2** | Indicador de velocidad | Req-112 | S | Fácil | 2 | 2 | N/A |
| **Ítem 3** | Indicador de velocidad | Req-113 | S | Medio | 8 | 2 | N/A |
| **Ítem 4** | Indicador de altura | Req-121 | S | Fácil | 2 | 1 | N/A |
| **Ítem 5** | Indicador de altura | Req-122 | S | Medio | 13 | 1 | N/A |
| **Ítem 6** | Mensajes de mantenimiento | Req-211 | D[16] | Medio | 5 | 7 | N/A |
| **Ítem 7** | Mensajes de mantenimiento | Req-212 | D | Medio | 5 | 7 | N/A |
| **Ítem 8** | Mensajes de alerta | Req-221 | S | Medio | 8 | 3 | N/A |
| **Ítem 9** | Mensajes de alerta | Req-222 | S | Medio | 8 | 3 | N/A |
| **Ítem 10** | Alerta de *No-dispatch* | Req-311 | S | Difícil | 21 | 4 | Ítems 8 y 9 |
| **Ítem 11** | Alerta de *Long dispatch* | Req-321 | S | Difícil | 13 | 5 | Ítems 8 y 9 |
| **Ítem 12** | Alerta de *Long dispatch* | Req-322 | S | Difícil | 13 | 5 | Ítems 8 y 9 |
| **Ítem 13** | Alerta de *Short dispatch* | Req-331 | S | Difícil | 13 | 6 | Ítems 8 y 9 |
| **Ítem 14** | Alerta de *Short dispatch* | Req-332 | S | Difícil | 13 | 6 | Ítems 8 y 9 |
| **Ítem 15** | Documentación | SDP | Doc[17] | Difícil | 21 | 0 | N/A |
| **Ítem 16** | Documentación | SVP | Doc | Difícil | 21 | 0 | N/A |
| **Ítem 17** | Documentación | TSP plantilla | Doc | Medio | 8 | 0 | N/A |
| **Ítem 18** | Documentación | Plantilla de | Doc | Medio | 8 | 0 | N/A |

[15] *S* corresponde a que es un requerimiento de Seguridad. Ver el Glosario para definición.
[16] *D* corrsponder a que es un requerimiento Derivado. Ver el Glosario para definición.
[17] *Doc* corresponde a que es documentación.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | diseño | | | | | |
| **Ítem 19** | Documentación | Equipo y herramientas | Doc | Difícil | 13 | 0 | N/A |

*Tabla 8 - Llenado de parámetros para cada ítem*

Al llegar a este punto, se puede decir que ya se tiene definido en forma clara el *backlog* del proyecto, este *backlog* inicial debe ser lo más parecido al resultado final que se desea alcanzar.

Es importante no olvidarse que el *backlog* es una lista viva de peticiones del cliente, las cuales pueden ser modificadas, eliminadas o agregadas durante el transcurso del proyecto, y el equipo de desarrollo y planeación debe ser capaz de adaptarse a los cambios de manera eficaz y efectiva.

Algo importante a mencionar es que, dado que se está hablando de software crítico y certificable, las organizaciones certificadoras solicitan todo tipo de documentación, pero por lo general, este primer paso no es necesario, ya que otro tipo de software comercial no requiere el mismo tipo de documentación y es preferible pasar directo a la ejecución, tal y como lo establece el manifiesto de *Agile*.

### 3.5.2. *Definition of Done*

Los criterios para establecer si las tareas se están concluyendo con éxito, son establecidos considerando los requerimientos del cliente y las normas de calidad a las que todo producto debe atenerse para lograr su certificación al garantizar su aeronavegabilidad.

Aunque este conjunto de criterios es para manejo interno del equipo, se procuran para alcanzar la aceptación del cliente satisfactoriamente.

Se recomiendan los siguientes puntos para establecer un buen DoD:

- Que establezca lo que el cliente quiere ver. Si el cliente busca ciertas características en cada funcionalidad, estas deben estar plasmadas en el DoD.

- Que se garantice la calidad del producto, es decir, que todos los artefactos cumplan con las especificaciones reglamentarias necesarias de las normas que se estén aplicando.
- Que los objetivos de cada *task* hayan sido alcanzados y que todos los *tasks* estén cerrados satisfactoriamente para poder cerrar *stories* con éxito.
- No requiere ser muy extenso, es mejor si es conciso, claro y fácil de seguir.

Para el ejemplo que se está llevando a cabo, se creó el siguiente DoD:

- Congruencia entre el diseño y código.
- El código compila sin errores.
- El código está correctamente documentado.
- Las fallas encontradas durante las pruebas y escrutinios fueron documentadas.
- Las fallas encontradas fueron atendidas.

### 3.5.3. Definición del tamaño de *sprint*

Para la exitosa implementación de esta mejora, se sugieren *sprints* de **4 semanas**. Esta cantidad de semanas se considera ideal para correr todo el proceso completo de desarrollo (ver sección 3.6)

## 3.6. Implementación

### 3.6.1. Proceso de desarrollo de sprint

En el desarrollo se presentan dos etapas de validación y verificación, la informal y la formal. Ambas cubren todos los aspectos de V&V de los artefactos creados, sin embargo, los resultados de la etapa formal ya llegan a manos del cliente, donde si cree necesario, los cambios son direccionados mediante artefactos llamados *Problem Reports*[18] (PR, por sus siglas en inglés). Es importante recalcar que la atención dada a cada PR levantado genera mayores costos.

[18] Ver Glosario para su definición.

A lo largo de este capítulo, se mostrará el proceso que se propone para llevar ambas etapas en cada *sprint* y lograr la implementación de la metodología V de manera ágil, a la vez que se reducen errores y el desarrollo se vuelve más eficiente.

#### 3.6.1.1. Validación y verificación informal

Durante esta etapa, el desarrollo al software comienza, al igual que las etapas de V&V informales, es decir, todas las fallas y errores encontrados durante esta etapa no genera PRs, convirtiéndola en la etapa donde se generarían los ahorros.

Como se puede apreciar en la Figura 15, cada artefacto creado tiene su inmediata tarea de V&V y se iterará en ese momento las veces que sean necesarios.

Nota importante: Lo ideal sería que la iteración entre desarrollo – V&V fuera no más de 1 vez, ya que se sugiera que vayan tan a la par, que se lleven a cabo con *pair programming* entre el desarrollador y el *tester*.

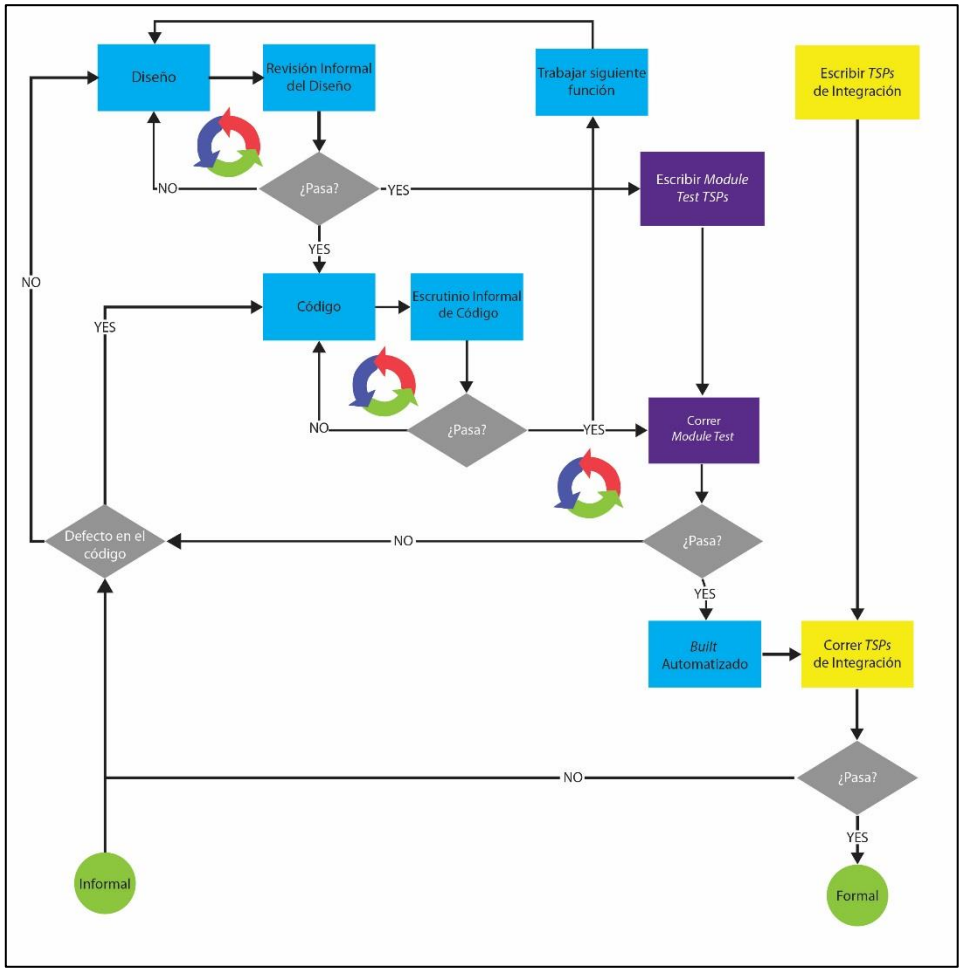


*Figura 15 - Diagrama de flujo de etapa de V&V informal*

### 3.6.1.2. Validación y verificación formal

En la etapa de V&V formal empiezan a surgir los PRs, pero se espera una cantidad baja, dado el trabajo realizado durante la etapa informal (Ver Figura 16).

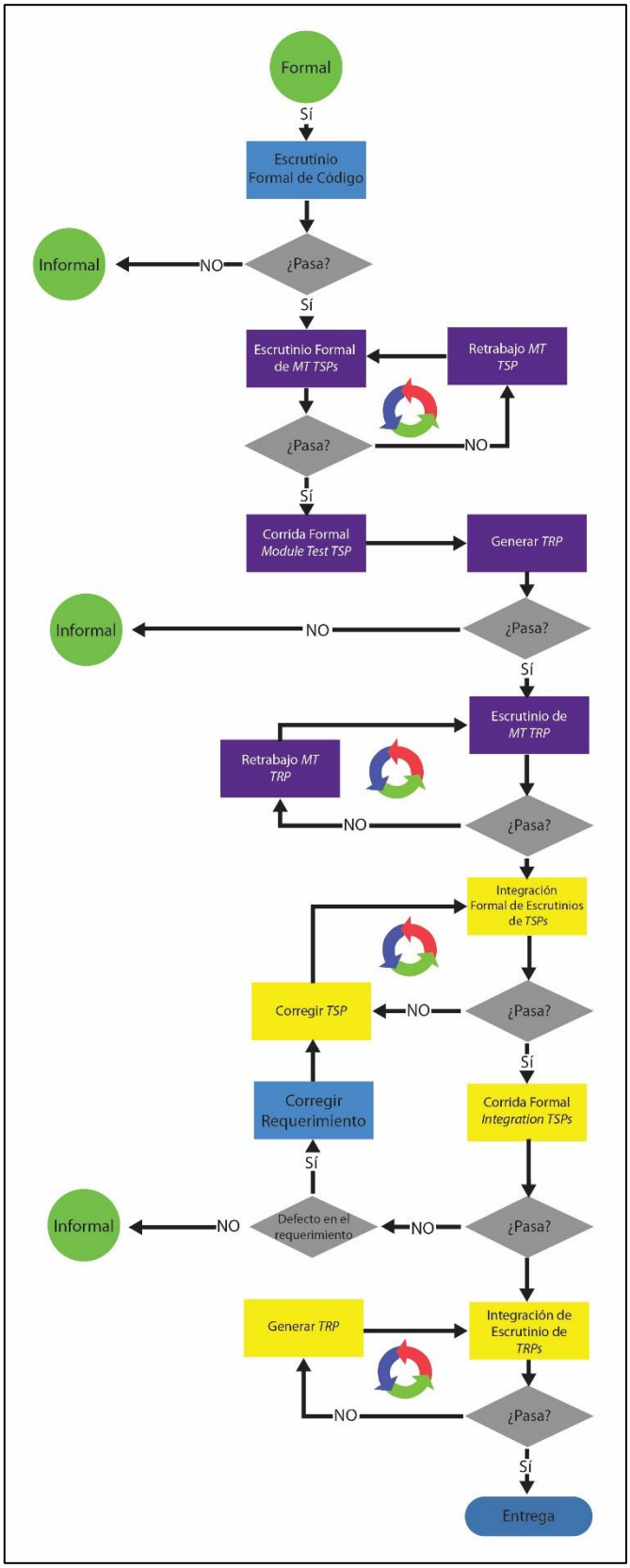


*Figura 16 - Diagrama de flujo de etapa de V&V formal*

### 3.6.2. Creación de los equipos de trabajo

#### 3.6.2.1. Equipo de liderazgo y administración de proyecto

Desde el inicio del proyecto, se recomienda que el equipo de liderazgo y administración esté conformado de la siguiente manera:

- 1 líder de equipo.
- 1 administrador de proyectos, especializado en proyectos de ingeniería de software.
- De 2 a 3 ingenieros de Sistemas con amplia experiencia.
- El arquitecto líder del software.

Empezar el proyecto con gente altamente experimentada, garantiza un comienzo efectivo, rápido y exitoso, ya que se valdrán de sus conocimientos y experiencias para tomar mejores decisiones y no se perderá tiempo en el "cómo se hará".

#### 3.6.2.2. Equipo técnico

Para la implementación de esta mejora, se recomienda que el equipo técnico esté conformado de la siguiente manera:

- Un equipo de Software y uno de V&V. El equipo de Sistemas provee las entradas para el equipo de desarrollo, pero no es parte de él.
- Un *Scrum Master* por cada equipo o grupo de máximo 12 personas.
- Si el equipo de desarrollo y V&V juntos no sobrepasa 12 personas entonces solo conformar un solo equipo con un solo *Scrum Master*.
- Los equipos no deben sobrepasar las 12 personas, para que sean equipos manejables y la comunicación aun sea efectiva entre ellos. Tener más personas involucradas no es sinónimo de que se terminará más pronto de lo planeado.

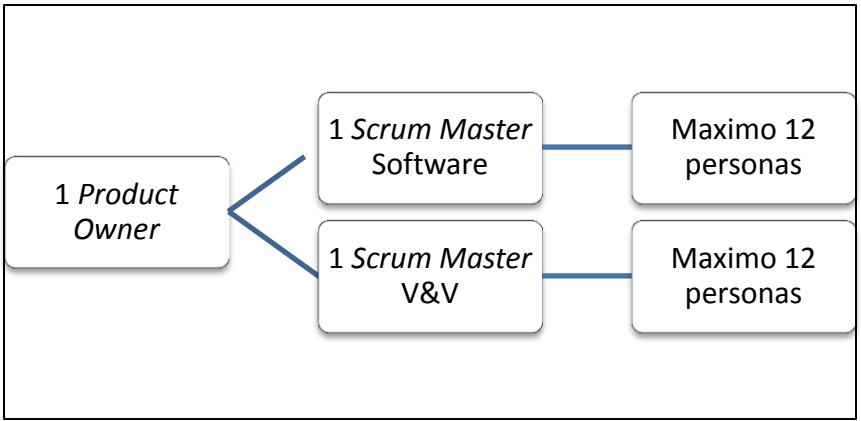

Figura 17 - 1ra opción de organigrama de equipo técnico.

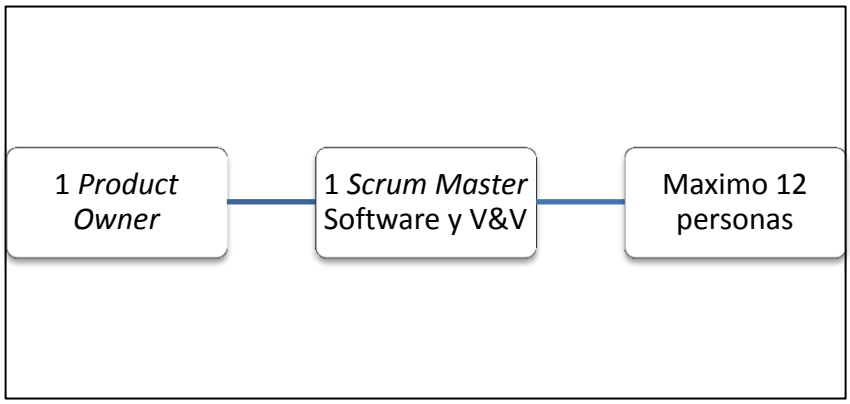


*Figura 18 - 2da opción de organigrama de equipo técnico*

### 3.6.3. Ejecución de *sprints*

Para la ejecución exitosa del proceso, se recomienda llevar a cabo los *sprints* de la siguiente manera:

- ***Kick-off***

  Antes de arrancar los *sprints*, es importante que los que conformar al equipo de liderazgo y administración de proyecto haya establecido los planes de presupuesto, el alcance, recursos, garantía de calidad y de contingencia, así como un plan de comunicación.

  Con el cliente se establecen los objetivos, fechas de entrega y los medios para presentar las funcionalidades.

- ***Sprint* 1**

  El propósito del *sprint* 1 es generar los primeros requerimientos de sistema, por lo que arranca con todo el equipo de sistemas ya involucrado.

En el área de software, se involucran sólo dos miembros experimentados del equipo software, para la realización de la documentación pertinente al proyecto. El objetivo es usar la experiencia de estos para que la creación de documentos sea más ágil y eficiente.

Mientras tanto, el equipo de V&V busca, analiza y compra las mejores herramientas para realizar las pruebas, garantizando la calidad de estas.

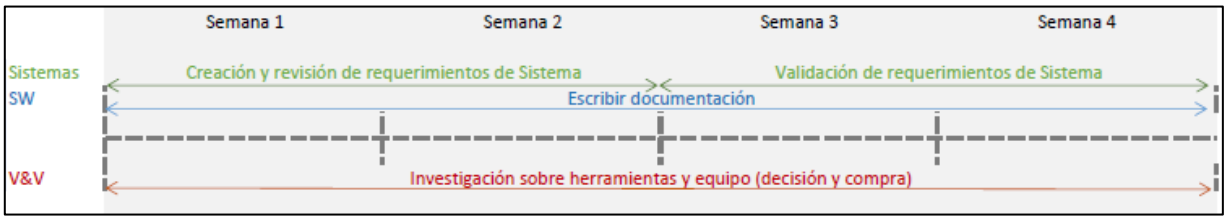


*Figura 19 - Calendario del sprint 1*

- ***Sprint* 2**

  Durante este *sprint*, el equipo de sistemas ya ha generado requerimientos y sigue en esta actividad; el equipo de software comienza la revisión y liberación de la documentación y el equipo de V&V configura las herramientas que ha adquirido.

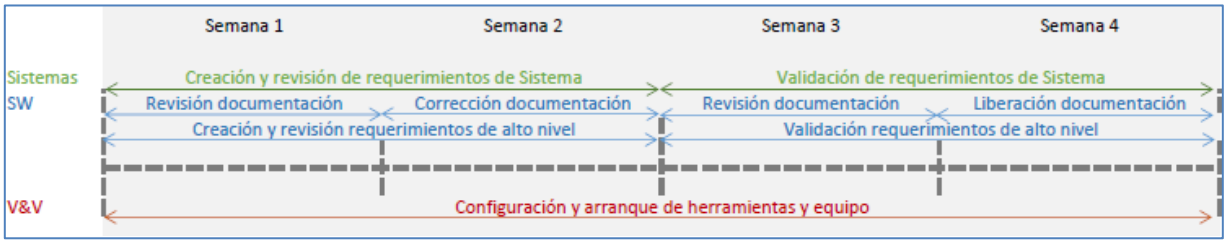


*Figura 20 - Calendario del sprint 2*

- ***Sprint* 3**

  Es durante este sprint, que el desarrollo de todos los artefactos empieza realmente y se va replicando hasta el último sprint planeado del proyecto.

  El equipo de software comienza con los requerimientos de sistemas que fueron creados hace ya dos *sprints*; el equipo de V&V debe ir trabajando en paralelo con el equipo de software para garantizar la entrega a tiempo.

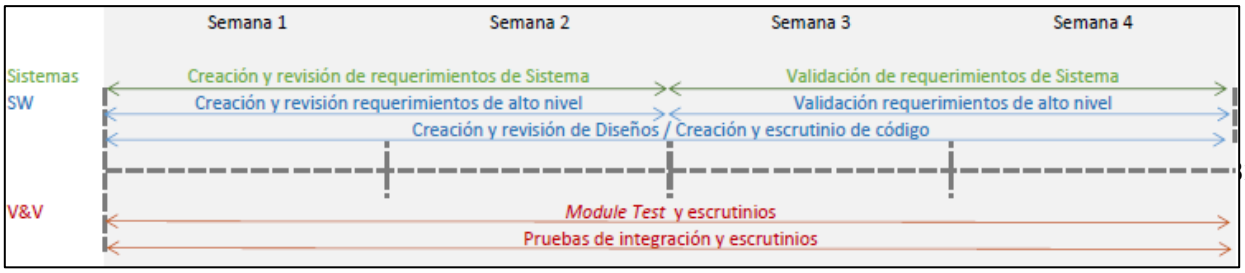

*Figura 21 - Calendario del sprint 3 al n*

### 3.6.4. Simulación

Para demostrar que trabajar con esta propuesta es mejor que trabajar con la metodología Cascada, se realizó la siguiente simulación, usando como base los datos promedio históricos[19], las fórmulas y macros que se utilizaron cuando el proyecto fue trabajado:

| | Etapas | Actividades | Ítems | Horas por ítem | Total hrs /actividad |
|---|---|---|---|---|---|
| **Informal** | Diseño | Diseño | 1,100 | 18 | 19,800 |
| | | Revisión informal de diseño | 1,100 | 4.5 | 4,950 |
| | Código | Desarrollo | 1,100 | 27 | 29,700 |
| | | Escrutinio de Código | 1,100 | 2.25 | 2,475 |
| | V&V de Requerimientos de bajo nivel | Escribir *MT TSPs* | 1,100 | 18 | 19,800 |
| | | Correr *MT TSPs* | 1,100 | 2.25 | 2,475 |
| | V&V de Requerimientos de alto nivel | Escribir *TSPs* | 200 | 45 | 9,000 |
| | | Correr *TSPs* | 200 | 4.5 | 900 |
| **Formal** | Código | Escrutinio formal de código | 1,100 | 2.25 | 2,475 |
| | V&V formal de requerimientos de bajo nivel | Escrutinio formal de *MT* | 1,100 | 9 | 9,900 |
| | | Correcciones *MT TSPs* | 1,100 | 2 | 2,200 |
| | | Corrida formal de *MT TSPs* | 1,100 | 2.25 | 2,475 |
| | | Generar *TRPs* | 1,100 | 2.25 | 2,475 |

[19] La simulación fue preparada usando algunos datos históricos como referencia del proyecto de Aviation Digital Power – APMS del 2014 a 2016,

| | | | | | |
|---|---|---|---|---|---|
| | | Escrutinio formal de*TRPs* | 1,100 | 2.25 | 2,475 |
| | | Correcciones *TRPs* | 1,100 | 0.5 | 550 |
| | V&V formal de requerimientos de alto nivel | Escrutinio Formal *TSPs* de Integración | 200 | 18 | 3,600 |
| | | Correcciones *TSPs* de Integración | 200 | 4 | 800 |
| | | Correr *TSPs* de Integración | 200 | 4.5 | 900 |
| | | Generar *TRPs* de Integración | 200 | 2.25 | 450 |
| | | Escrutinio *TRPs* de Integración | 200 | 2.25 | 450 |
| | | Correcciones *TRPs* de Integración | 200 | 0.5 | 100 |
| **TOTAL** | | | | | 117,950 |

*Tabla 9 - Datos promedio históricos.*

Si se suma el total de horas de cada actividad, se obtiene un total de 117,950 horas, es decir, que ese es el costo ideal del proyecto. Considerando esto, se fijó un plazo de 18 meses para completar el Proyecto bajo este número ideal de horas, dando la libertad de mover los recursos a conveniencia.

Como se explicó en capítulos anteriores, la metodología de Cascada se desarrolla por etapas, por lo que una etapa comienza cuando la anterior ha finalizado. En este caso, con los 18 meses fijos, se establecieron períodos para cada fase para así poder determinar si era posible completar el proyecto en este tiempo, tal y como se puede apreciar en la Tabla 10.

| | | DISEÑO | | | CODIGO | | | | | | TESTING | | | | | | FORMAL | | | PRs | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Actividad** | Items por mes | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 |
| **Diseño** | 367 | 6600 | 6600 | 6600 | | | | 1980 | 1980 | 1980 | | | | 990 | 990 | 990 | | | | 330 | 330 | 330 |
| **Revision informal de diseño** | 367 | 1650 | 1650 | 1650 | | | | | | | | | | | | | | | | | | |
| **Desarrollo** | 183 | | | | 4950 | 4950 | 4950 | 4950 | 4950 | 4950 | | | | 2970 | 2970 | 2970 | | | | 1485 | 1485 | 1485 |
| **Escrutinio de Código** | 367 | | | | | | | 825 | 825 | 825 | | | | | | | | | | | | |
| **Escribir *MT TSPs*** | 183 | | | | | | | | | | 3300 | 3300 | 3300 | 3300 | 3300 | 3300 | | | | | | |
| **Correr *MT TSPs*** | 183 | | | | | | | | | | 413 | 413 | 413 | 413 | 413 | 413 | | | | | | |
| **Escribir *TSPs*** | 33 | | | | | | | | | | 1500 | 1500 | 1500 | 1500 | 1500 | 1500 | | | | | | |
| **Correr *TSPs*** | 33 | | | | | | | | | | 150 | 150 | 150 | 150 | 150 | 150 | | | | | | |
| **Escrutinio formal de código** | 367 | | | | | | | | | | | | | | | | 825 | 825 | 825 | | | |
| **Escrutinio formal de *MT*** | 367 | | | | | | | | | | | | | | | | 3300 | 3300 | 3300 | | | |
| **Correcciones *MT TSPs*** | 367 | | | | | | | | | | | | | | | | 733 | 733 | 733 | | | |
| **Corrida formal de *MT TSPs*** | 367 | | | | | | | | | | | | | | | | 825 | 825 | 825 | | | |
| **Generar *TRPs*** | 367 | | | | | | | | | | | | | | | | 825 | 825 | 825 | | | |
| **Escrutinio *TRPs*** | 367 | | | | | | | | | | | | | | | | 825 | 825 | 825 | | | |
| **Correcciones *TRPs*** | 367 | | | | | | | | | | | | | | | | 183 | 183 | 183 | | | |
| **Escrutinio Formal *TSPs* de Integración** | 67 | | | | | | | | | | | | | | | | 1200 | 1200 | 1200 | | | |
| **Correcciones *TSPs* de Integración** | 67 | | | | | | | | | | | | | | | | 267 | 267 | 267 | | | |
| **Correr *TSPs* de Integración** | 67 | | | | | | | | | | | | | | | | 300 | 300 | 300 | | | |
| **Generar *TRPs* de Integración** | 67 | | | | | | | | | | | | | | | | 150 | 150 | 150 | | | |
| **Escrutinio *TRPs* de Integración** | 67 | | | | | | | | | | | | | | | | 33 | 33 | 33 | | | |
| **Correcciones *TRPs* de Integración** | 67 | | | | | | | | | | | | | | | | 150 | 150 | 150 | | | |

*Tabla 10 - Calendarización del proyecto bajo metodología Cascada.*

Para la etapa de Diseño se tienen contempladas 19,800 horas en el desarrollo de diseños (18 horas por ítem) y 4,950 horas en su revisión informal (4.5 horas por ítem). Considerando que ya existe la documentación pertinente para la ejecución del proyecto, se planearon 3 meses para llevar esta etapa a cabo, empezando en el Mes 1. Dado el número de ítems por desarrollar y el tiempo, durante cada mes se deben desarrollar 367 diseños y revisar el mismo número, 367. Para poder lograr este número, es necesario contar con un equipo de 49 personas considerando que una persona puede producir 170 horas a la semana.

Si se logra terminar la etapa de Diseño en los 3 primeros meses del proyecto, podemos comenzar con la etapa de desarrollo de código en el Mes 4. El desarrollo de código puede llegar a ser una de las etapas más largas, por lo que se contempló 6 meses para su ejecución, e invirtiendo 29,700 horas de desarrollo de código (27 horas por función). Esta etapa también contempla 3 meses para escrutinios o revisiones informales de código, invirtiendo 2,475 horas en escrutinios o revisiones (2.25 horas por función), empezando en el Mes 7.

Para los Meses 4, 5 y 6 es necesario contar con un equipo de 29 personas, sin embargo, para los Meses 7, 8 y 9, el número de personas aumenta a 46 personas. Esto se debe a que durante estos meses no solo comienzan los escrutinios de código, sino que los errores encontrados durante la revisión informal de los diseños deben ser trabajados.

Para efectos de esta simulación, las correcciones de Diseños, en su primera vuelta, consumen un 30% más, por lo que el costo de esa actividad ejecutada en esos 3 meses se eleva a 5,940 horas más. Si después de esta vuelta, hay necesidad de otra, es probable que sea de menos errores. Una segunda vuelta involucraría hasta un 15%, elevando el costo 2 970 horas. Juntando estas horas de correcciones, el costo se eleva 8,910 horas más, es decir que, en vez de que estas actividades se realicen en 19,800 horas (como estaba planeado), se realizarán en 28,710 horas.

Una vez pasadas las etapas de Desarrollo de Diseño y Código, empieza la etapa de Validación y Verificación (V&V). Esta etapa puede llegar a ser la etapa más larga de todo el ciclo, ya que este V&V es aplicado tanto a los requerimientos de alto nivel como a los de bajo nivel y podría conllevar varios ciclos de correcciones.

En nuestro proyecto simulado, planeamos empezar el V&V en el Mes 10 con intención de trabajarla por 6 meses, es decir, finalizando en el Mes 15, considerando que para la escritura de *test procedures* para los requerimientos de alto nivel se consumirían 9,000 horas, mientras que su ejecución, 900 horas (esto porque se pueden correr varios *test procedures* al mismo tiempo). En cuanto a la escritura de *test procedures* para los requerimientos de bajo nivel se prevén 19,800 horas y 2,475 horas para su ejecución. Todo esto con un equipo de 32 personas.

La diferencia entre las horas consideradas entre los *test procedures* de alto nivel contra los de bajo nivel en la cantidad de funciones de código probadas al mismo tiempo. Las funciones de código son validadas y verificadas por módulos de requerimientos de bajo nivel, es decir, se aplicaría una técnica de pruebas de unidad (ver Cap. 2.9.2); mientras que, a los requerimientos de alto nivel, se aplicarían la técnica de pruebas de integración (ver Cap. 2.9.3).

Las correcciones de Código se estiman de manera igual que las de Diseño; considerando el número de ítems totales y la probabilidad de 30% de errores a trabajar, el costo se eleva 8 910 horas, es decir, que en vez de que sea una actividad de 29 700 horas, sería de 38 610 horas.

Dado que estamos trabajando con software aviónico, la fase de V&V debe ser formalizada, es decir, los *test procedures* deben ser ejecutados en la versión final del software. Esto debe contemplado en la planeación del proyecto y para fines de esta simulación, se consideraron 3 meses, es decir del Mes 16 al 18, con un equipo de 57 personas. Este V&V formal también es llevado tanto para alto como para bajo nivel.

Para la formalización del V&V de alto nivel se contemplan 6 300 horas, donde vienen incluidos escrutinios de *test procedures* más correcciones y su ejecución. Para el V&V de bajo nivel se contemplan, 20,075 horas, incluidas las mismas actividades. Hasta este punto, hemos trabajado los 18 meses considerados para el proyecto y a simple vista, podría considerarse que se terminó a tiempo, sin embargo, tomando en cuenta que la etapa de Diseño se llevó a cabo con un costo mayor y que si durante la formalización de V&V, fueron encontrados errores en los diseños o en el código, estos deben ser arreglados bajo un proceso de reporte de problemas (PR, por sus

siglas en inglés), por lo que se regresaría a esa etapa forzado por el PR correspondiente y corriendo un ciclo para el artefacto afectado. En esta simulación se prevén 3 meses más para la solución de PR, esto aumenta un 33% la inversión (990 horas) si son PR de Diseño y un 15% (4,455 horas) si son de Implementación.

En contraste, ejecutar el proyecto con la metodología propuesta de Vertical Ágil produce ahorros o, por lo menos, mantenerse dentro del costo original planteado.

Tal y como se explica en el Cap. 3.6, los primeros 3 *sprints* son planeados para trabajar en la documentación operacional y funcional del proyecto, sin embargo, se puede empezar la etapa de desarrollo de Diseño en el Sprint 1, dando lugar a que esta etapa se desarrolle por los 18 *sprints* considerados para el proyecto. Esto permite que se trabajen menos ítems por sprint, es decir, que, repartiendo los 1,100 diseños, repartidos entre los 18 *sprint*, se tendrían que trabajar 65 ítems, invirtiendo 1,164.706 horas. Este número será constante a lo largo del proyecto.

La actividad de V&V de Diseños no se puede empezar el V&V durante el mismo *sprint*, porque no habría suficientes entradas para llevarla a cabo, sin embargo, puede ser comenzado justo al siguiente, en el *Sprint* 2, dándonos lugar a que se trabajen 69 ítems por cada uno de los 17 *sprints*, equivalente a 309.375 horas por *sprint*.

Para el *sprint* 3 es posible arrancar todas las demás actividades y trabajarlas hasta el último *sprint*. A diferencia del V&V de Diseño, el V&V de código puede empezar al mismo tiempo que la implementación, ya que el número de ítems que se requieren por *sprint* por cada actividad de V&V es menor a la cantidad de ítems que se desarrollan.

La implementación de código implica el desarrollo de 73 ítems por *sprint*, es decir, el uso de 1,980 horas, mientras que las actividades de V&V son parecidas en números. Como estas actividades están siendo ejecutadas al mismo tiempo, el mismo número de *test procedures* iniciales como los de formalización son, por ende, igual; 69 *test procedures* de bajo nivel y 13, de alto nivel.

Este enfoque permite que se requieran menos personas, manteniendo el número de recursos constante durante casi todo el proyecto. Como en el *sprint* 1 se ejecuta solo

una actividad, no es necesario involucrar a todo el equipo contemplado para todo el proyecto. Para esta simulación se calcularon 7 personas, es decir que las 1,165 horas a trabajar en el *Sprint* 1 entre las 170 horas disponibles en el equipo nos da ese número. Este número se ve aumentado a 9 personas en el *Sprint* 2 por la adición de una actividad. A partir del *Sprint* 3, el equipo se mantiene en 43 personas. Como se estima que en el último *sprint* las actividades de Desarrollo de Diseños y Código hayan terminado y solo sigan las de V&V, el equipo se reduciría a 22 personas.

| Actividad | Items por *sprint* | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Diseño** | 65 | 1165 | 1165 | 1165 | 1165 | 1165 | 1165 | 1165 | 1165 | 1165 | 1165 | 1165 | 1165 | 1165 | 1165 | 1165 | 1165 | 1165 | |
| **Revision informal de diseño** | 69 | | 309 | 309 | 309 | 309 | 309 | 309 | 309 | 309 | 309 | 309 | 309 | 309 | 309 | 309 | 309 | 309 | |
| **Desarrollo** | 73 | | | 1980 | 1980 | 1980 | 1980 | 1980 | 1980 | 1980 | 1980 | 1980 | 1980 | 1980 | 1980 | 1980 | 1980 | 1980 | |
| **Escrutinio de Código** | 73 | | | 165 | 165 | 165 | 165 | 165 | 165 | 165 | 165 | 165 | 165 | 165 | 165 | 165 | 165 | 165 | |
| **Escribir *MT TSPs*** | | | | 349 | 349 | 349 | 349 | 349 | 349 | 349 | 349 | 349 | 349 | 349 | 349 | 349 | 349 | 349 | |
| **Correr *MT TSPs*** | | | | 594 | 594 | 594 | 594 | 594 | 594 | 594 | 594 | 594 | 594 | 594 | 594 | 594 | 594 | 594 | |
| **Escribir *TSPs*** | 69 | | | 1238 | 1238 | 1238 | 1238 | 1238 | 1238 | 1238 | 1238 | 1238 | 1238 | 1238 | 1238 | 1238 | 1238 | 1238 | 1238 |
| **Correr *TSPs*** | 69 | | | 155 | 155 | 155 | 155 | 155 | 155 | 155 | 155 | 155 | 155 | 155 | 155 | 155 | 155 | 155 | 155 |
| **Escrutinio formal de código** | 13 | | | 563 | 563 | 563 | 563 | 563 | 563 | 563 | 563 | 563 | 563 | 563 | 563 | 563 | 563 | 563 | 563 |
| **Escrutinio formal de *MT*** | 13 | | | 56 | 56 | 56 | 56 | 56 | 56 | 56 | 56 | 56 | 56 | 56 | 56 | 56 | 56 | 56 | 56 |
| **Correcciones *MT TSPs*** | 69 | | | 155 | 155 | 155 | 155 | 155 | 155 | 155 | 155 | 155 | 155 | 155 | 155 | 155 | 155 | 155 | 155 |
| **Corrida formal de *MT TSPs*** | 69 | | | 619 | 619 | 619 | 619 | 619 | 619 | 619 | 619 | 619 | 619 | 619 | 619 | 619 | 619 | 619 | 619 |
| **Generar *TRPs*** | 69 | | | 138 | 138 | 138 | 138 | 138 | 138 | 138 | 138 | 138 | 138 | 138 | 138 | 138 | 138 | 138 | 138 |
| **Escrutinio *TRPs*** | 69 | | | 155 | 155 | 155 | 155 | 155 | 155 | 155 | 155 | 155 | 155 | 155 | 155 | 155 | 155 | 155 | 155 |
| **Correcciones *TRPs*** | 69 | | | 155 | 155 | 155 | 155 | 155 | 155 | 155 | 155 | 155 | 155 | 155 | 155 | 155 | 155 | 155 | 155 |
| **Escrutinio Formal *TSPs* de Integración** | 69 | | | 155 | 155 | 155 | 155 | 155 | 155 | 155 | 155 | 155 | 155 | 155 | 155 | 155 | 155 | 155 | 155 |
| **Correcciones *TSPs* de Integración** | 69 | | | 34 | 34 | 34 | 34 | 34 | 34 | 34 | 34 | 34 | 34 | 34 | 34 | 34 | 34 | 34 | 34 |
| **Correr *TSPs* de Integración** | 13 | | | 225 | 225 | 225 | 225 | 225 | 225 | 225 | 225 | 225 | 225 | 225 | 225 | 225 | 225 | 225 | 225 |
| **Generar *TRPs* de Integración** | 13 | | | 50 | 50 | 50 | 50 | 50 | 50 | 50 | 50 | 50 | 50 | 50 | 50 | 50 | 50 | 50 | 50 |
| **Escrutinio *TRPs* de Integración** | 13 | | | 56 | 56 | 56 | 56 | 56 | 56 | 56 | 56 | 56 | 56 | 56 | 56 | 56 | 56 | 56 | 56 |
| **Correcciones *TRPs* de Integración** | 13 | | | 28 | 28 | 28 | 28 | 28 | 28 | 28 | 28 | 28 | 28 | 28 | 28 | 28 | 28 | 28 | 28 |
| **Diseño** | 13 | | | 28 | 28 | 28 | 28 | 28 | 28 | 28 | 28 | 28 | 28 | 28 | 28 | 28 | 28 | 28 | 28 |
| **Revision informal de diseño** | 13 | | | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 |

*Tabla 11 - Calendarización del proyecto bajo metodología V ágil.*

# Resultados

El mismo ejemplo mencionado en el Cap.3.1.1 bajo la propuesta planteada en esta tesina, da las siguientes estimaciones:
Partiendo de que el desarrollo inicial tendría un costo de 19,800 horas y la etapa de V&V de 73,200 horas, pero con la diferencia que se trabajarían los esfuerzos al mismo tiempo (Ver capítulo 3.3), es decir, el equipo de V&V estará probando software al mismo tiempo que el equipo de desarrollo está implementando.

Como consecuencia de esto, las correcciones de defectos y errores son realizadas por el mismo equipo que desarrolló el software. Se estima que estas correcciones se realicen en 9 900 horas, mientras que su validación y verificación en 36,600 horas (de acuerdo a la cotización inicial). El objetivo es que se encuentren tantos defectos y errores sea posible, para evitar así una fase de V&V larga que conlleve muchos recursos.

La formalización del código y de las pruebas se llevaría de la misma manera, por lo que se estima que se realice en 8,800 horas y su fase de validación y verificación, en 53,200 horas. Los defectos y errores serán corregidos en 1,980 horas y su V&V, en 7,320 horas.

Esto tiene un primer impacto en la cantidad de ingenieros que se establece para realización de las actividades del proyecto. Si se consideraba que fueran 12 meses de desarrollo y 6 meses de V&V, se puede tomar los 18 meses para la implementación de ambas fases en el cual se necesitarían 14 ingenieros para el desarrollo y 56 para validar y verificar.
Si se contara con 24 meses, se necesitarían 10 ingenieros para el desarrollo y 42 ingenieros para el V&V.

En total, el proyecto tendría un costo total de 210,800 horas (40,480 horas del costo de desarrollo más 170,320 horas del costo de V&V), un 36% por encima de la

cotización inicial (ver Tablas 12 y 13), es decir, un **19%** de ahorro a comparación de la aplicación de la metodología en Cascada.

| | Costo |
|---|---|
| **Desarrollo informal** | 19 800 horas |
| **Corrección de errores** | 9 900 horas |
| **Formalización** | 8 800 horas |
| **Corrección de errores en la formalización** | 1 980 horas |
| **Total** | 40 480 horas |

*Tabla 12 - Costo de desarrollo en metodología V ágil.*

| | Costo |
|---|---|
| **V&V informal** | 73 200 horas |
| **V&V de la corrección de errores** | 36 600 horas |
| **Formalización de V&V** | 53 200 horas |
| **Corrección de errores en la formalización de V&V** | 7 320 horas |
| **Total** | 170 320 horas |

*Tabla 13 - Costo final de V&V en metodología V ágil.*

Se puede apreciar en la Figura 22, la diferencia de costos entre ambas metodologías y con esto se puede concluir que la metodología propuesta (V ágil) es conveniente tanto en costo como en tiempos.

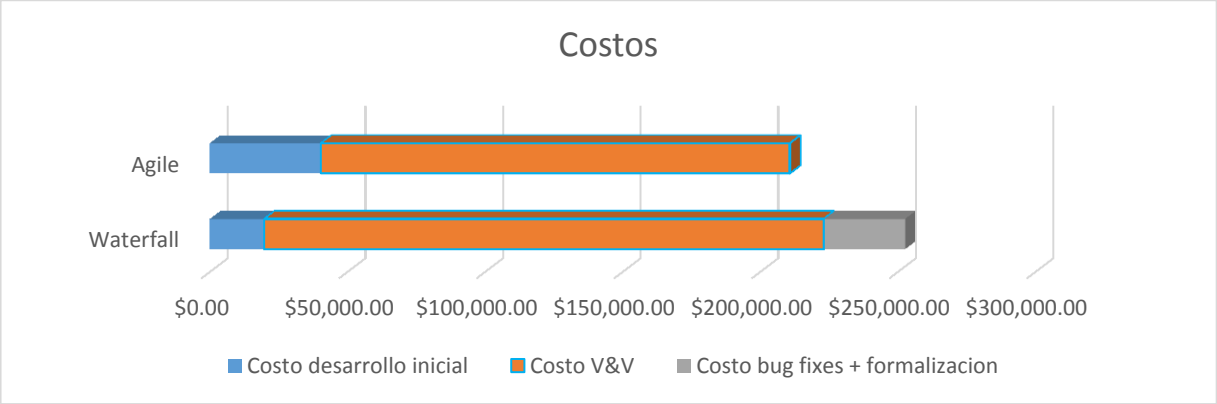


*Figura 22 - Gráfica comparativa de costos Cascada y V-ágil.*

Esta investigación demuestra que el número de ocurrencias de fallas no depende únicamente de la correcta elección de metodologías de pruebas y la estandarización de las mismas, sino que involucra desde el principio la manera en que se elige la metodología de desarrollo de software. El V&V es solo una fase afectada por todas las demás fases. Trabajar en sinergia dentro del mismo ciclo de vida es lo que provocará la disminución de fallas.

Esto mismo lleva a decir que a medida que el software empleado sea complejo, nuevas metodologías de pruebas tendrán que ser desarrolladas para probarlo óptima y completamente, ya que los tiempos de entrega se han vuelto cada vez más demandantes, tal y cómo se puede apreciar, de entre 18 a 24 meses, es un tiempo relativamente corto para entregar un producto de software tan complejo que su malfuncionamiento puede derivar en pérdidas humanas.

En cuanto a que la utilización de múltiples metodologías de pruebas garantice que estas sean más robustas y completas de los requerimientos del software del producto aviónico, dependerá de las decisiones que se tomen con el cliente.

# Conclusiones

- Esta mejora permite introducir al equipo de V&V con el equipo de software mediante *Peer programming*, incrementando la comunicación y por ende la velocidad de ejecución del proyecto.

- Ambas metodologías terminan el desarrollo y el V&V en 18 meses, pero la diferencia radica en que la metodología V ágil lo cierra sin errores, mientras que en la metodología Cascada falta una fase de correcciones.

- El costo del proyecto desarrollado en cascada es de 141,215 horas, mientras que desarrollado en vertical es de 117,950 horas. Eso implica un 19.72% más de horas al desarrollarlo en Cascada.

- Al cliente se le puede dar software funcional, sin errores y de manera formal al final de cada *sprint*.

- El cliente queda satisfecho al poder ver de manera tangible y formal el desarrollo de su producto.

- Con esta propuesta, se busca que todos los errores sean atrapados en la parte informal, permitiendo la transición a la parte formal sea libre de errores o con muy pocos y de bajo impacto.

- En la fase formal se espera encontrar muy pocos errores y de bajo impacto; incluso permite la formalización desde el principio, lo que ahorraría más tiempo y dinero.

- Durante la fase de escrutinio de Verificación en la metodología V ágil, cada error identificado es atendido de forma inmediata, permitiendo la oportunidad de aprender rápidamente y eliminando replicar el mismo error en repetidas ocasiones. En cambio, usando la metodología Cascada, si se

introduce un error durante el desarrollo, este no es atrapado hasta que se llega a la parte de V&V, la cual se da hasta un año después que se empezó el desarrollo. Esto trae las siguientes consecuencias:

  - Que un error se haya propagado o duplicado impactando varios módulos de código.
  - Que el equipo de desarrollo ya no esté conformado por las mismas personas que empezaron.

- Trabajando con la metodología Cascada, la eliminación de errores requiere ciclos de correcciones que pueden consumir hasta 3 meses cada uno y no garantizan que el código esté libre de errores.

- En cambio, trabajando con V ágil, no es necesario incluir una fase de correcciones ya que está va incluida durante la ejecución de las demás fases. Esto garantiza un código más limpio al final del desarrollo.

- La metodología V ágil permite crear equipos de alto rendimiento y ampliamente colaborativos, ya que interactúan de manera cercana, beneficiando la buena calidad y velocidad del proyecto.

- Trabajando con la metodología V ágil, los cambios en requerimientos de sistema no tienen un gran impacto en el desarrollo del producto y pueden ser fácilmente incorporados por el equipo del desarrollo, ya que este equipo está enfocado en otro set de requerimientos.

- Se reduce los bloqueos, ya que tanto los equipos de Software y V&V están concentrados en tratar las mismas funcionalidades.

- Las cantidades de recursos es muy variable a lo largo de la ejecución de la metodología en Cascada, mientras que, en la V ágil, los equipos se mantienen con el mismo número de personas durante 15 de 18 *sprints*.

- La cantidad de ítems que debe ser trabajados al mes en metodología Cascada y la cantidad de recursos requeridos son un número difícil de alcanzar. Es poco probable que se puede crear un equipo así de grande que genere esa cantidad de entregables al mes. Dada esta improbabilidad, la cantidad de meses para trabajar en cascada no sería de 21, sino que sería mucho mayor.

- Usando metodología Cascada, la formalización puede llevarse el mismo tiempo que la fase de *testing* (6 meses), por lo que terminar un proyecto de 1,100 diseños en 18 meses es poco probable.

# Recomendaciones

Se hacen las siguientes recomendaciones con el fin de que la implementación de esta mejora sea eficaz y eficiente:

- Tener un buen *Product Owner* y *Scrum Master(s)*. Un buen equipo de trabajo, experimentado y comunicativo es clave para el desarrollo eficiente de cada *sprint*. Un equipo con buenos elementos mantiene una amplia visión sobre el proyecto.
- Se debe establecer una buena comunicación con el arquitecto; cambios inesperados pueden venir desde la arquitectura de producto, conocerlos de antemano ayuda a tomar mejores decisiones.
- Incorporar los equipos de SW y V&V como se sugiere para avanzar rápidamente; es importante eliminar la idea de que el equipo de V&V debe trabajar al final de etapa de desarrollo.
- Los ingenieros con más experiencia deben ser incorporados desde el arranque del proyecto y los de menos experiencia deben ser incorporados a partir del 3er *sprint*. Esto permite que el proyecto empiece con velocidad y se mantenga así conforme se van integrando más elementos
- Fomentar la multifuncionalidad de los equipos, evitando que los equipos sean aislados por actividad, ya que esto corta la comunicación y evita el flujo dinámico entre las personas.
- Es altamente recomendable que los ingenieros involucrados en la ejecución del proyecto comprendan la metodología. Esto incentiva la eficiencia.
- Aunque la metodología V ágil presenta muchas ventajas por sobre la metodología Cascada, siempre será recomendable evaluar qué es lo que nuestro cliente quiere y que es lo mejor para la ejecución de nuestros proyectos.

# Referencias bibliográficas


1. **International Civil Aviation Organization.** Archived Assembly Meetings. *Documento.* [En línea] 2013. http://www.icao.int/Meetings/AMC/MA/Accidet Investigation and Prevention (AIG) Divisional Meeting/AIGdiv08_WP02_sp.pdf.

2. **Flight Safety Foundation.** "Number of accidents and fatalities per year.". *Reporte estadístico.* [En línea] 2013. https://aviation-safety.net/statistics/period/stats.php?cat=A1.

3. **HEGAN - Basque Aeronautical Cluster.** "2013, un año muy seguro para la aviación comercial.". *Artículo.* [En línea] 01 de 02 de 2014. http://hegan.siccluster.com/sicc/verFicha.do?ididioma=1&idficha=25712&l=-1.

4. **Ruiz, Francisco.** "Procesos de Ingeniería del Software". *Presentación de conferencia.* [En línea] 2010. [Citado el: 17 de noviembre de 2015.] http://www.ctr.unican.es/asignaturas/is1/is1-t02-trans.pdf.

5. **Colombia., Universidad Nacional de.** "Ciclo de vida del Software". *Clase virtual.* [En línea] 2015. [Citado el: 22 de noviembre de 2015.] http://www.virtual.unal.edu.co/cursos/sedes/manizales/4060024/Lecciones/Capitulo%20I/problemas.htm.

6. **Anónimo.** "Etapas del ciclo de vida". *Artículo.* [En línea] 2015. [Citado el: 22 de noviembre de 2015.] https://ciclodevidasoftware.wikispaces.com/ETAPAS+DEL+CICLO+DE+VIDA.

7. **Soules de Caseros, Lourdes.** "La importancia de realizar pruebas de software". *Artículo.* [En línea] 2013. [Citado el: 22 de noviembre de 2015.] http://negociosymanagement.com.ar/?p=2807.

8. **Miranda, Abdiel.** "Metodología en cascada". *Artículo.* [En línea] 2013. [Citado el: 23 de noviembre de 2015.] https://prezi.com/xtvq8pyvmhzn/modelo-en-cascada-royce-1970/.

9. **Zapata Sánchez, Javier.** "Pruebas de software". *Artículo.* [En línea] 2013. [Citado el: 06 de diciembre de 2015.] https://pruebasdelsoftware.wordpress.com/tag/metodologia-de-pruebas/ .

10. **University of Michigan.** Chapter 1. *Fundamentals of Avionics.* 2015.

11. **Federal Aviation Administration.** "Acta de Aviación Federal". *Documento.* [En línea] 2015. [Citado el: 3 de enero de 2016.] www.faa.gov.
12. **Dion, Bernard.** "Do-178C Airborne Software Rulation Standard.". *Webinar.* [En línea] 2013. [Citado el: 03 de enero de 2016.] http://www.esterel-technologies.com/wp-content/uploads/2013/09/Webinar-How-to-Comply-with-DO-178C.pdf .
13. **Federal Aviation Admnistration.** "Airborne Software Assurance.". *Advisory Circular.* [En línea] 2013. [Citado el: 03 de enero de 2016.] http://www.faa.gov/documentlibrary/media/advisory_circular/ac_20-115c.pdf.
14. **SAE International.** "Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment.". *Documento.* [En línea] 2015. [Citado el: 03 de enero de 2016.] http://standards.sae.org/arp4761/ .
15. **ISTQB Exam Certification.** "What is Verification in software testing? Or what is software verification?". *Artículo.* [En línea] 2015. [Citado el: 21 de diciembre de 2015.] http://istqbexamcertification.com/what-is-verification-in-software-testing-or-what-is-software-verification/.
16. —. "What is Validation in software testing? Or what is software validation?". *Artículo.* [En línea] 2015. [Citado el: 24 de diciembre de 2015.] http://istqbexamcertification.com/what-is-validation-in-software-testing-or-what-is-software-validation/.
17. **Rodríguez Tello, Eduardo A.** "Estrategias y técnicas de prueba del software". *Presentación de conferencia.* [En línea] 2012. [Citado el: 03 de enero de 2016.] http://www.tamps.cinvestav.mx/~ertello/swe/sesion15.pdf .
18. **ISTQB Exam Certification.** "What are Software Test Types?". *Artículo.* [En línea] 2015. [Citado el: 28 de diciembre de 2015.] http://istqbexamcertification.com/what-are-software-test-types/ .
19. —. "What is Functional Testing (Testing of functions) in Software?". *Artículo.* [En línea] 2015. [Citado el: 28 de diciembre de 2015.] http://istqbexamcertification.com/what-is-functional-testing-testing-of-functions-in-software/.
20. **Eriksson, Ulf.** "Functional vs Non-Functional Testing.". *Artículo.* [En línea] 2012. [Citado el: 28 de diciembre de 2015.] http://reqtest.com/testing-blog/functional-vs-non-functional-testing/.

21. **ISTQB Exam Certification.** "What is Non-functional testing (Testing of software product characteristics)?". *Artículo.* [En línea] 2015. [Citado el: 28 de diciembre de 2015.] http://istqbexamcertification.com/what-is-non-functional-testing-testing-of-software-product-characteristics/.
22. —. "What is Structural testing (Testing of software structure/arquitecture)?". *Artículo.* [En línea] 2015. [Citado el: 28 de diciembre de 2015.] http://istqbexamcertification.com/what-is-structural-testing-testing-of-software-structurearchitecture/ .
23. —. "What is Unit testing?". *Artículo.* [En línea] 2015. [Citado el: 24 de diciembre de 2015.] http://istqbexamcertification.com/what-is-unit-testing/.
24. —. "What is Integration Testing?". *Artículo.* [En línea] [Citado el: 24 de diciembre de 2015.] http://istqbexamcertification.com/what-is-integration-testing/.
25. —. "What is Big Bang Integration Testing?". *Artículo.* [En línea] 2015. [Citado el: 27 de diciembre de 2015.] http://istqbexamcertification.com/what-is-big-bang-integration-testing/ .
26. **Exforsys Inc.** "What is Incremental Integration Testing?". *Artículo.* [En línea] 2015. [Citado el: 27 de diciembre de 2015.] http://www.exforsys.com/tutorials/testing-types/incremental-integration-testing.html.
27. **Rouse, Margaret.** "Regression Testing.". *Artículo.* [En línea] 2015. [Citado el: 26 de diciembre de 2015.] http://searchsoftwarequality.techtarget.com/definition/regression-testing.
28. **ISTQB Exam Certification.** "What is Acceptance Testing?". *Artículo.* [En línea] 2015. [Citado el: 26 de diciembre de 2015.] http://istqbexamcertification.com/what-is-acceptance-testing/.
29. **Peham, Thomas.** "5 types of user acceptance tests - the perfect UAT framework.". *Artículo.* [En línea] 2015. [Citado el: 26 de diciembre de 2015.] http://usersnap.com/blog/types-user-acceptance-tests-frameworks/.
30. **Beck, Kent.** "Manifesto for Agile Software Development.". *Artículo.* [En línea] 2011. [Citado el: 02 de enero de 2016.] http://agilemanifesto.org/principles.html .
31. —. "Principles behind the Agile Manifesto.". *Artículo.* [En línea] 2011. [Citado el: 02 de enero de 2016.] http://agilemanifesto.org/principles.html.
32. **ISTQB Exam Certification.** "What is Test-Driven Development?". *Artículo.* [En línea] 2015. [Citado el: 03 de enero de 2016.] http://istqbexamcertification.com/what-are-

the-different-agile-testing-methodology-test-driven-development-behavior-driven-development/#test_driven_development.
33. **Osherove, Roy.** "Understanding Test Driven Development.". *Plática virtual.* [En línea] 2012. [Citado el: 24 de diciembre de 2015.] www.artofunittesting.com.
34. **James, Michael.** "Scrum Reference Card.". *Artículo.* [En línea] 2015. [Citado el: 02 de enero de 2016.] http://scrumreferencecard.com/reference-card-de-scrum/.
35. **ISTQB Exam Certification.** "What are Agile Software Development approaches? Scrum, Kanban, XP explained.". *Artículo.* [En línea] 2015. [Citado el: 02 de enero de 2016.] http://istqbexamcertification.com/what-are-agile-software-development-approaches-scrum-kanban-xp-explained/.
36. **Adkins, Lyssa.** "The High Performance Tree.". *Plática en video.* [En línea] 2011. [Citado el: 30 de diciembre de 2015.] https://www.youtube.com/watch?v=t3kKechcwYM.
37. **Radigan, Dan.** "A brief introduction to Kanban.". *Artículo.* [En línea] 2015. [Citado el: 29 de diciembre de 2015.] https://www.atlassian.com/agile/kanban/.
38. **Lean Enterprise Institute.** "A brief history of Lean.". *Artículo.* [En línea] 2015. [Citado el: 02 de enero de 2016.] http://www.lean.org/WhatsLean/History.cfm.
39. —. "Principles of Lean.". *Artículo.* [En línea] 2015. [Citado el: 02 de enero de 2016.] http://www.lean.org/WhatsLean/Principles.cfm.
40. **Wells, Don.** "the Rules of Extreme Programming.". *Artículo.* [En línea] 2013. [Citado el: 29 de diciembre de 2015.] http://www.extremeprogramming.org/index.html.
41. —. "Pair Programming.". *Artículo.* [En línea] 2013. [Citado el: 29 de diciembre de 2015.] http://www.extremeprogramming.org/rules/pair.html .
42. —. "CRC Cards.". *Artículo.* [En línea] 2013. [Citado el: 29 de diciembre de 2015.] http://www.extremeprogramming.org/rules/crccards.html.
43. **Unidad de Desarrollo Tecnológico en Inteligencia Artificial.** "Metodología de desarrollo de sotware. El Modelo en V o de Cuatro Niveles.". [En línea] 2008. [Citado el: 13 de 12 de 2015.] http://www.iiia.csic.es/udt/es/blog/jrodriguez/2008/metodologia-desarrollo-sotware-modelo-en-v-o-cuatro-niveles.
44. **Goytia, Luis López.** *Programación Orientada a Objetos C++ y Java.* s.l. : Grupo Editorial Patria, 2014.