



UTPL
La Universidad Católica de Loja

Modalidad Abierta y a Distancia

Metodologías de Desarrollo

Guía didáctica



Índice

Primer
bimestre

Segundo
bimestre

Solucionario

Referencias
bibliográficas

Facultad de Ingenierías y Arquitectura

Departamento de Ciencias de la Computación y Electrónica

Metodologías de Desarrollo

Guía didáctica

Carrera	PAO Nivel
▪ <i>Tecnologías de la Información</i>	VI

Autores:

Soto Guerrero Fernanda Maricela
Correa Tenesaca Roddy Andrés



D S O F _ 3 0 4 4

Asesoría virtual
www.utpl.edu.ec

Índice

Primer bimestre

Segundo bimestre

Solucionario

Referencias bibliográficas

Universidad Técnica Particular de Loja

Metodologías de Desarrollo

Guía didáctica

Soto Guerrero Fernanda Maricela

Correa Tenesaca Roddy Andrés

Diagramación y diseño digital:

Ediloja Cía. Ltda.

Telefax: 593-7-2611418.

San Cayetano Alto s/n.

www.ediloja.com.ec

edilojacialtda@ediloja.com.ec

Loja-Ecuador

ISBN digital - 978-9942-25-934-9



Reconocimiento-NoComercial-CompartirIgual
4.0 Internacional (CC BY-NC-SA 4.0)

Usted acepta y acuerda estar obligado por los términos y condiciones de esta Licencia, por lo que, si existe el incumplimiento de algunas de estas condiciones, no se autoriza el uso de ningún contenido.

Los contenidos de este trabajo están sujetos a una licencia internacional Creative Commons **Reconocimiento-NoComercial-CompartirIgual 4.0 (CC BY-NC-SA 4.0)**. Usted es libre de **Compartir** – copiar y redistribuir el material en cualquier medio o formato. **Adaptar** – remezclar, transformar y construir a partir del material citando la fuente, bajo los siguientes términos: **Reconocimiento**- debe dar crédito de manera adecuada, brindar un enlace a la licencia, e indicar si se han realizado cambios. Puede hacerlo en cualquier forma razonable, pero no de forma tal que sugiera que usted o su uso tienen el apoyo de la licenciatario. **No Comercial**-no puede hacer uso del material con propósitos comerciales. **Compartir igual**-Si remezcla, transforma o crea a partir del material, debe distribuir su contribución bajo la misma licencia del original. No puede aplicar términos legales ni medidas tecnológicas que restrinjan legalmente a otras a hacer cualquier uso permitido por la licencia. <https://creativecommons.org/licenses/by-nc-sa/4.0/>

Índice

Índice

1. Datos de información.....	8
1.1. Presentación de la asignatura	8
1.1. Competencias genéricas de la UTPL	8
1.2. Competencias específicas de la carrera.....	8
1.3. Problemática que aborda la asignatura.....	9
2. Metodología de aprendizaje.....	10
3. Orientaciones didácticas por resultados de aprendizaje.....	12
Primer bimestre	12
Resultado de aprendizaje 1	12
Contenidos, recursos y actividades de aprendizaje	12
 Semana 1	13
 Unidad 1. Naturaleza del software y su relación con las organizaciones	13
1.1. La naturaleza del software.....	14
Actividades de aprendizaje recomendadas	19
 Semana 2	19
1.2. Ingeniería del software.....	20
1.3. Relación del software con la industria.....	29
Actividades de aprendizaje recomendadas	31
Autoevaluación 1	33

Primer
bimestre

Segundo
bimestre

Solucionario

Referencias
bibliográficas

Resultado de aprendizaje 2	36
Contenidos, recursos y actividades de aprendizaje	36
Semana 3	36
Unidad 2. Procesos de ingeniería de software	37
2.1. Modelos de procesos de software.....	39
Actividades de aprendizaje recomendadas	44
Semana 4	44
Actividades de aprendizaje recomendadas	52
Semana 5	53
2.2. Metodologías tradicionales	54
Actividades de aprendizaje recomendadas	61
Semana 6	62
2.3. Metodologías ágiles	62
Actividades de aprendizaje recomendadas	66
Semana 7	66
2.4. Metodologías tradicionales vs. metodologías ágiles	71
Actividades de aprendizaje recomendadas	75
Autoevaluación 2	76
Actividades finales del bimestre.....	79
Semana 8	79
Actividades de aprendizaje recomendadas	79

Segundo bimestre	80
Resultado de aprendizaje 2	80
Contenidos, recursos y actividades de aprendizaje	80
 Semana 9	81
Unidad 3. Metodologías ágiles de desarrollo de software	81
3.1. Programación extrema.....	82
Actividades de aprendizaje recomendadas	91
Autoevaluación 3	92
 Semana 10	95
3.2. Scrum	95
Actividades de aprendizaje recomendadas	107
 Semana 11	108
Actividades de aprendizaje recomendadas	125
Autoevaluación 4	126
Resultado de aprendizaje 3	129
Contenidos, recursos y actividades de aprendizaje	129
 Semana 12	129
Unidad 4. Automatización del proceso de desarrollo de software.	130
4.1. DevOps	130
Actividades de aprendizaje recomendadas	136
 Semana 13	137
Actividades de aprendizaje recomendadas	151

Semana 14	151
Actividades de aprendizaje recomendadas	157
Autoevaluación 5	158
Semana 15	161
 Unidad 5. Nuevos paradigmas de desarrollo	161
5.1. Aplicaciones nativas en la nube	161
Actividades de aprendizaje recomendadas	165
Autoevaluación 6	166
Actividades finales del bimestre.....	169
Semana 16	169
Actividades de aprendizaje recomendadas	169
4. Solucionario	170
5. Referencias bibliográficas	180

Índice

Primer
bimestre

Segundo
bimestre

Solucionario

Referencias
bibliográficas

Índice

Primer bimestre

Segundo bimestre

Solucionario

Referencias bibliográficas



1. Datos de información

1.1. Presentación de la asignatura



1.1. Competencias genéricas de la UTPL

- Organización y planificación del tiempo.

1.2. Competencias específicas de la carrera

- Gestionar la implementación de soluciones de negocio mediante la ejecución de proyectos de TI que cumplan adecuadamente los requisitos especificados por la organización.

1.3. Problemática que aborda la asignatura

- Conocimiento sobre la naturaleza del *software* y su relación con la industria.
- Aplicabilidad de modelos tradicionales y ágiles en el contexto de proyectos de *software* y su gestión.
- Estudio de las principales metodologías de desarrollo ágil, su importancia, factores de éxito y aplicabilidad en proyectos de *software*.
- Conocimiento sobre nuevas, importancia y aplicabilidad de prácticas de desarrollo como DevOps, escenarios de automatización y aplicabilidad en contextos *cloud* y OnPremise.
- Conocimiento sobre nuevos paradigmas de desarrollo, características y estilos arquitectónicos que impulsa el desarrollo de aplicaciones nativas en *cloud*.



2. Metodología de aprendizaje

Antes de iniciar con el estudio de las unidades, es importante que analice y siga las siguientes recomendaciones:

- Dedique como mínimo una hora diaria al estudio, revise el plan docente semanalmente y cumpla con cada una de las actividades planteadas, tales como síncronas, asíncronas, tareas, cuestionarios, actividades recomendadas y las autoevaluaciones que se encuentran al final de cada unidad de estudio.
- Agende las fechas de las actividades en línea y entrega de tareas, esto le permitirá tener un mayor control y seguimiento de las actividades que tiene que realizar.
- Ingrese con periodicidad (semanalmente) al EVA, para que revise los anuncios académicos publicados por su tutor, estos anuncios contienen lineamientos u orientaciones que debe seguir en cada semana de estudio.
- Se recomienda que mientras aprende los temas propuestos, avance en el desarrollo de las tareas, que deberán ser enviadas mediante el EVA para su calificación de acuerdo con las fechas establecidas (ver plan docente). Es necesario aclarar que solamente las tareas enviadas por este medio serán calificadas.

- Para empezar el estudio de esta asignatura, se recomienda primero leer las páginas de la 1 a la 9 del texto-guía, en el que se encuentra las indicaciones de cómo desarrollará el trabajo en esta asignatura. Seguidamente, inicie con el estudio de los temas de la semana 1 planificados en el plan docente.
- Al final de cada unidad de estudio, se plantean autoevaluaciones. Sea honesto en su desarrollo, esto le permitirá tener un criterio para determinar su nivel de aprendizaje, y si usted cree que necesita revisar algún tema una y otra vez, hágalo; identifique y mejore su metodología de estudio, y, por último y no menos importante, es necesario que planifique su tiempo.
- Las dudas e inquietudes que tenga en relación con la asignatura, realícelas a su tutor mediante los diferentes canales que le brinda la UTPL, tales como mensajes en el EVA, correo electrónico, vía telefónica o por chat de consultas.

Antes de iniciar su proceso de aprendizaje, se lo invita a revisar con detenimiento el plan docente que se encuentra en el EVA, en este se detallan las competencias y los resultados de aprendizaje que va a adquirir en la asignatura, los contenidos, las actividades de aprendizaje y el tiempo estimado de estudio; con toda la información proporcionada podrá organizarse para que su aprendizaje sea el idóneo.



3. Orientaciones didácticas por resultados de aprendizaje



Primer bimestre

Resultado de aprendizaje 1

Aplica ciclos y procesos de desarrollo de acuerdo a las características de un proyecto de software.

Contenidos, recursos y actividades de aprendizaje

Estimado estudiante: inicie con el desarrollo de la primera unidad de esta asignatura, es necesario que identifique desde la raíz del software para conocer cómo surgió y para qué, cuáles han sido sus beneficios y los inconvenientes hasta los días actuales; de esa manera, será capaz de reconocer la falta de organización o técnicas que permitan identificar, de manera exacta, cuáles son las necesidades reales para las que es requerido.

Índice

Primer bimestre

Segundo bimestre

Solucionario

Referencias bibliográficas

Con esa idea en su mente responda a los siguientes interrogantes:

¿Cómo cree usted que el desarrollo de software ha impactado en nuestra sociedad? ¿Considera que el software por sí solo puede funcionar, o qué se requiere?; a lo largo de su vida ¿Qué tipo software ha utilizado y reconoce? Algunas de estas preguntas serán respondidas y lo orientarán a que las analice con mayor detenimiento a lo largo de la unidad 1.



Semana 1



Unidad 1. Naturaleza del software y su relación con las organizaciones

Creo que este es el mejor consejo: piensa constantemente cómo podrías hacer mejor las cosas.

Elon Musk, cofundador de PayPal, Space X y Tesla

1.1. La naturaleza del software

La humanidad ha necesitado siempre establecer mecanismos para todo tipo de comunicación, desde los tambores parlantes, lenguajes codificados —por ejemplo, el código Morse— y la transmisión física del sonido. Para la comunicación se necesita siempre de dos elementos: el emisor y el receptor; pero qué pasa si los dos no hablan el mismo lenguaje, o utilizan mecanismos diferentes al que conocen; la comunicación no se va a poder establecer, porque no hay comprensión ni entendimiento del mensaje transmitido; para este caso, existe un tercer elemento necesario, un “interpretador” que traduzca ese mensaje al receptor.

Siguiendo el contexto de la comunicación, es importante analizar uno de los puntos de inicio para la computación. Se parte de la máquina de Turing, dado este nombre por su creador Alan Turing, quién decidió **expresar** las formulaciones de teoremas matemáticos en algoritmos que sean **interpretados** por una máquina. Con su mecanismo fundamentó las bases teóricas para la construcción de las computadoras, formalizó el concepto del algoritmo y así los lenguajes de programación (Díaz, 2013).

Antes de continuar es importante que tenga claro qué es un **algoritmo**: es una serie ordenada de instrucciones bien definidas, que llevan a la solución de un determinado problema.

Ahora bien, continúe con la idea central. La analogía que se quiere transmitirle es que el software permite también establecer una comunicación entre un emisor y un receptor, en este caso entre un usuario y una máquina, con el fin de obtener los beneficios que ya todos conocen hoy en día, facilitar el desarrollo de ciertas tareas mediante su automatización.

[Índice](#)[Primer bimestre](#)[Segundo bimestre](#)[Solucionario](#)[Referencias bibliográficas](#)

1.1.1. Definición de software

Es muy importante partir de las definiciones básicas o formales para consolidar los conocimientos, por ello lea, analice y comprenda las definiciones de la Tabla 1, estás han sido recopiladas de autores y estándares referidos a *software*.

Tabla 1. Definiciones de *software*

Autor	Definición
IEEE Std 610.12-1990	Programas informáticos, procedimientos y posiblemente documentación y datos asociados relacionados con el funcionamiento de un sistema informático.
Piattini, 2004	Conjunto de programas, procedimientos y documentación asociada a la operación de un sistema informático.
Pressman, 2010	<p>El <i>software</i> de computadora es el producto que construyen los programadores profesionales y al que después se le da mantenimiento durante un largo tiempo. Incluye programas que se ejecutan en una computadora de cualquier tamaño y arquitectura.</p> <p>El autor también indica que el <i>software</i> es:</p> <ol style="list-style-type: none">1. Instrucciones (programas de cómputo) que cuando se ejecutan proporcionan las características, función y desempeño buscados.2. Estructuras de datos que permiten que los programas manipulen en forma adecuada la información3. Información descriptiva, tanto en papel como en formas virtuales, que describe la operación y uso de los programas.
Diccionario de la Real Academia Española	Conjunto de programas, instrucciones y reglas informáticas para ejecutar ciertas tareas en una computadora.
Sommerville, 2011	Programas de cómputo y documentación asociada. Los productos de <i>software</i> se desarrollan para un cliente en particular o para un mercado en general.

Índice

Primer bimestre

Segundo bimestre

Solucionario

Referencias bibliográficas

Cómo puede observar, las definiciones analizadas llevan a tener claro que el *software* es un conjunto de procedimientos y programas que se ejecutan en una computadora para cumplir con una tarea o una acción específica.

1.1.2. Dominios de aplicación

El *software* se aplica en diferentes ámbitos y se lo ha logrado clasificar en categorías, de acuerdo con las funciones o tareas por realizar. En la Tabla 2 se incluyen las siete categorías en las que clasifica (Pressman, 2010):

Tabla 2. Dominios de aplicación del software

Categorías/Dominios de aplicación	Descripción
Software de sistemas	Es un conjunto de programas escritos para dar servicio a otros programas, se caracterizan por la gran interacción con el <i>hardware</i> de la computadora, uso intensivo por parte de usuarios múltiples, operación concurrente y recursos compartidos. Se lo conoce también como <i>software de base</i> . Ejemplos: sistemas operativos, controladores de dispositivos, compiladores, interfaz gráfica de usuarios y línea de comandos.

[Índice](#)[Primer bimestre](#)[Segundo bimestre](#)[Solucionario](#)[Referencias bibliográficas](#)

Categorías/Dominios de aplicación	Descripción
Software de aplicación	<p>Son programas aislados que resuelven una necesidad específica de negocios, se los conoce también como <i>software especializados</i>. Se usa para controlar funciones de negocios en tiempo real, por ejemplo: procesamiento de transacciones en un punto de venta, control de procesos de manufactura en tiempo real.</p> <p>En la literatura se encuentra que a este tipo de <i>software</i> se lo subclasifica en aplicaciones de negocio, aplicaciones de utilería, aplicaciones personales, aplicaciones de entretenimiento, entre otros, tal como se menciona en Olarte (2017).</p> <p>Ejemplos: Microsoft Office (Word, Excel, Power Point, entre otros), navegadores (Google Chrome, Mozilla Firefox), tratamiento de imágenes e ilustraciones (Paint, paquete de Adobe), entre otros.</p>
Software de ingeniería y ciencias	<p>Son caracterizados por algoritmos “devoradores de números”. Las aplicaciones van de la astronomía a la vulcanología, del análisis de tensiones en automóviles a la dinámica orbital del transbordador espacial, y de la biología molecular a la manufactura automatizada.</p> <p>Ejemplos: AutoCAD, Matlab, Google Earth.</p>
Software incrustado	<p>Reside dentro de un producto o sistema. Se usa para implementar y controlar características y funciones para el usuario final y para el sistema en sí. Ejecuta funciones limitadas y particulares –por ejemplo, control del tablero de un horno de microondas–, o provee una capacidad significativa de funcionamiento y control –funciones digitales en un automóvil, tales como el control del combustible, del tablero de control y de los sistemas de frenado–. Se lo conoce también como <i>software embebido</i> o <i>empotrado</i>.</p>

Índice

Primer bimestre

Segundo bimestre

Solucionario

Referencias bibliográficas

Categorías/Dominios de aplicación	Descripción
Software de línea de productos	Es diseñado para proporcionar una capacidad específica para uso de muchos consumidores diferentes. El software de línea de productos se centra en algún mercado limitado y particular –por ejemplo, control del inventario de productos–, o se dirige a mercados masivos de consumidores (procesamiento de textos, hojas de cálculo, gráficas por computadora, multimedios, entretenimiento, administración de base de datos y aplicaciones para finanzas personales o de negocios).
Aplicaciones web	Esta categoría de software centrado en redes agrupa una amplia gama de aplicaciones y actúa como una interfaz entre el usuario e internet. Son un conjunto de archivos de hipertexto vinculados que presentan información con uso de texto, audio o video y gráficas limitadas. Sin embargo, desde que surgió la web 2.0, las aplicaciones web están evolucionando hacia ambientes de cómputo sofisticados, integradas con bases de datos corporativas y aplicaciones de negocios. Ejemplos: Aplicaciones web estáticas, dinámicas, animadas, gestores de contenido (Wordpress, Joomla, Drupal), portales, E-commerce.
Software de inteligencia artificial	Hace uso de algoritmos no numéricos para resolver problemas complejos que no son fáciles de tratar computacionalmente o con el análisis directo. Las aplicaciones en esta área incluyen robótica, sistemas expertos, reconocimiento de patrones (imagen y voz), redes neurales artificiales, demostración de teoremas y juegos.

1.1.3. Software heredado

En el estudio de Medina et al. (2019), se indica que el software heredado es aquel cuya tecnología es considerada obsoleta, su estructura está deteriorada, contiene reglas del negocio que no aplican a toda la organización y no se puede reemplazar fácilmente.

Índice

Primer bimestre

Segundo bimestre

Solucionario

Referencias bibliográficas



Actividades de aprendizaje recomendadas

Analice con mayor detenimiento los dominios de aplicación del software; identifique sistemas de su entorno, o sobre los que usted conoce, y clasifíquelos de acuerdo con el dominio, y finalmente, establezca la diferencias entre ellos. Puede referirse al libro de Pressman (2010, p. 6).



Semana 2

Una vez que ha conocido los inicios del software y sus dominios de aplicación, es momento de estudiarlo con mayor profundidad. Hay una frase muy conocida entorno a estos temas “la crisis del software”, que desde sus inicios ha tenido contratiempos y, como resultado, la insatisfacción de los usuarios finales. Para superar todos los contratiempos, se creó una disciplina que se conoce como la **ingeniería del software**, y es en esta semana que la analizará con detalle, y finalizará la unidad 1.

1.2. Ingeniería del software

La ingeniería del software es considerada como una disciplina dedicada al estudio del desarrollo, operación y mantenimiento del software, es decir, que se preocupa desde las primeras etapas de la especificación del sistema hasta su mantenimiento después de ponerlo en operación. Además de que es considerada esencial para el funcionamiento de las sociedades, por la inclusión de técnicas que apoyan la especificación, el diseño y la evolución de un sistema.

La ingeniería del software es importante ya que, como lo indica Pressman (2010), permite construir sistemas complejos en un tiempo razonable y con alta calidad.

Es muy importante partir de una base conceptual para consolidar sus conocimiento, por ello, analice con mayor detenimiento el siguiente apartado con los conceptos de la ingeniería de software.

1.2.1. Conceptos

El concepto de ingeniería de software nace en 1968, en la segunda etapa cronológica del software, cuando surge la denominada crisis de esta industria, que fue marcada por los excesos de costos, la escasa fiabilidad, la insatisfacción de los usuarios y el tiempo de creación de software que no finalizaba en el plazo establecido.

Revise algunos de los conceptos propuestos sobre la ingeniería de software en la Tabla 3.

Tabla 3. Conceptos de ingeniería del software

Autores	Conceptos
Fritz (1968)	Ingeniería del software es el establecimiento y uso de principios robustos de ingeniería, orientados a obtener software que sea fiable y funcione de manera eficiente sobre máquinas reales.
Blum (1992)	La ingeniería del software es la aplicación de herramientas, métodos y disciplinas para la producción y mantenimiento de soluciones automáticas a problemas del mundo real.
Swebok (2004)	Señala la definición dada por la IEEE: la ingeniería del software es la aplicación de un enfoque sistemático, disciplinado y cuantificable al desarrollo, operación y mantenimiento del software, es decir, la aplicación de la ingeniería al software.
Pressman (2010)	La ingeniería del software es una disciplina que integra métodos, herramientas y procedimientos para el desarrollo de software de computadora.
Sommerville (2011)	La ingeniería del software es una disciplina de la ingeniería que se interesa por todos los aspectos de la producción de software. Sus actividades fundamentales son la especificación, el desarrollo, la validación y la evolución del software.
Pantaleo (2015)	La ingeniería del software administra los procesos de gestión y desarrollo de los proyectos de software. Su importancia radica en que los cambios realizados a algunos de los aspectos mencionados va a impactar en el otro.
Palomo (2020)	La ingeniería del software amplía la visión del desarrollo de software como una actividad esencialmente de programación, contemplando, además, otras actividades de análisis y diseño previos, y de integración y verificación posteriores.

Tal como puede observar, todos los autores analizados convergen hacia una misma orientación sobre lo que definen como ingeniería del software. El fin al que conlleva es a la disciplina que orienta y organiza para desarrollar software, pero con la característica de que sea fiable.

¿Cuál de las definiciones ha quedado en su mente? ¿Cuál lo orienta a tener una idea más clara de la ingeniería del software?

Es recomendable que destaque los términos principales para que se familiarice con la disciplina, por ejemplo, principios, métodos, herramientas, procesos.

1.2.2. Principios

De acuerdo con lo analizado en el ítem anterior, la ingeniería de software se identifica por apoyar el proceso de su desarrollo, y, para lograrlo, algunos autores han definido principios de gran importancia con el objetivo de que el proceso se lleve de forma exitosa.

Los principios que han logrado definirse son aplicados durante el proceso de la ingeniería del software, es decir, durante el análisis, construcción y gestión del software, pero hay que considerar que es necesario también contar con métodos apropiados y técnicas específicas. Es importante tener en cuenta que los principios son la base de todos los métodos, técnicas, metodologías y herramientas que se estudiarán en la unidad 2.

Para Pressman (2010), los principios ayudan a establecer un conjunto de herramientas mentales para una práctica sólida de la ingeniería de software.

Ghezzi et al. (1991) proponen los siguientes principios:

- a. **Rigor y formalidad:** mediante un enfoque riguroso podrán producirse productos más confiables, controlando sus costos e incrementando el grado de confianza en estos. La formalidad es un requerimiento más fuerte que el rigor: requiere que el proceso de software sea guiado y evaluado por leyes matemáticas. En todos los campos de la ingeniería, el proceso de diseño sigue una secuencia de pasos bien

Índice

Primer bimestre

Segundo bimestre

Solucionario

Referencias bibliográficas

definidos, establecidos en forma precisa y posiblemente probados, siguiendo en cada paso algún método o aplicando alguna técnica. Esto aplica también en el caso de la ingeniería de software, por ejemplo, en el caso de la especificación del software, la cual puede establecerse de forma rigurosa utilizando lenguaje natural, o también puede darse formalmente mediante una descripción formal en un lenguaje de sentencias lógicas. Tradicionalmente, es en la fase de codificación en la que se utiliza un enfoque formal, ya que los programas son objetos formales: son escritos en un lenguaje cuya sintaxis y semántica están completamente definidas. La aplicación del principio de rigor y formalidad tiene influencia beneficiosa en la obtención de cualidades del software, tales como la confiabilidad, verificabilidad, mantenibilidad, reusabilidad, portabilidad, comprensibilidad e interoperabilidad.

- b. **Separación de intereses:** este principio permite enfrentarse a los distintos aspectos individuales de un problema de forma que se pueda concentrar en cada uno por separado. La única forma de enfrentar la complejidad de un proyecto es separar los distintos intereses. Las formas en que el autor menciona que pueden separarse los distintos intereses pueden ser según el tiempo, cualidades, visiones del software, tamaño, e incluso las responsabilidades. La complejidad global puede resolverse mucho mejor concentrándose en los distintos aspectos por separado. Este principio es la base para dividir el trabajo de un problema complejo en asignaciones de trabajo específicas, posiblemente a personas distintas con habilidades diferentes.
- c. **Modularidad:** un sistema complejo puede dividirse en piezas más simples llamadas módulos; un sistema compuesto de módulos es llamado modular. El principal beneficio del modularidad es que permite la aplicación del principio de separación de intereses en dos fases: al enfrentar los detalles de cada módulo por separado ignorando detalles de los

otros módulos, y al enfrentar las características globales de todos los módulos y sus relaciones para integrarlos en un único sistema coherente. Si estas fases son ejecutadas en ese orden, se dice que el sistema es diseñado de abajo hacia arriba (*bottom up*), en el orden inverso, se dice que el sistema es diseñado de arriba hacia abajo (*top down*). El principio de modularidad tiene tres objetivos principales: capacidad de descomponer un sistema complejo, capacidad de componerlo a partir de módulos existentes y comprensión del sistema en piezas. Este procedimiento refleja el bien conocido principio de “divide y vencerás”; la capacidad de comprender cada parte de un sistema en forma separada ayuda a la modificabilidad del sistema. Debido a la naturaleza evolutiva del *software*, muchas veces se debe volver hacia atrás, al trabajo previo, y modificarlo.

- d. **Abstracción:** es un proceso mediante el cual se identifican aspectos relevantes de un problema ignorando los detalles; es un caso especial del principio de separación de intereses, en el cual se separan los aspectos importantes de los detalles de menor importancia.

El principio de abstracción es un principio importante que se aplica tanto a los productos de *software* como a los procesos.

- e. **Anticipación al cambio:** el *software* sufre cambios constantemente, como se vio al tratar la mantenibilidad del *software*. Estos cambios pueden surgir por la necesidad de eliminar errores que no fueron detectados antes de liberar la aplicación, o por la necesidad de apoyar la evolución de la aplicación, debido a nuevos requerimientos o cambios en los requerimientos existentes. Por lo tanto, este principio puede ser utilizado para lograr la evolucionabilidad del *software* y también la reusabilidad de componentes, viendo

la reusabilidad como evolucionabilidad de granularidad más fina, a nivel de componentes. La aplicación de este principio requiere que se disponga de herramientas apropiadas para gestionar las varias versiones y revisiones del *software* en forma controlada.

- f. **Generalidad:** establece que, al tener que resolver un problema, se debe buscar un problema más general que posiblemente esté oculto tras el problema original, posiblemente la solución al problema general tenga potencial de reúso, o se diseñe un módulo que puede ser invocado por más de un punto en la aplicación, en lugar de tener varias soluciones especializadas. Una solución general posiblemente sea más costosa, en términos de rapidez de ejecución, requerimientos de memoria o tiempo de desarrollo, que una solución especializada para el problema original, por lo que debe evaluarse la generalidad respecto al costo y a la eficiencia al momento de decidir. La generalidad es un principio fundamental si se tiene como objetivo el desarrollo de herramientas generales o paquetes para el mercado, ya que para ser exitosas, deberán cubrir las necesidades de distintas personas, por ejemplo, los procesadores de texto representan una tendencia general en el *software*. Para cada área específica de aplicación, existen paquetes generales que proveen soluciones estándares a problemas comunes.
- g. **Incrementalidad:** es un proceso que se desarrolla en forma de pasos, en incrementos, alcanzando el objetivo deseado mediante aproximaciones sucesivas a este, en el que cada aproximación es alcanzada mediante un incremento de la previa. Consiste en identificar subconjuntos tempranos de una aplicación que sean útiles, de forma que se obtenga retroalimentación (*feedback*) temprana del cliente. Esto permite que la aplicación evolucione en forma controlada

en los casos en que los requerimientos iniciales no están estables o completamente entendidos. La motivación de este principio es que muchas veces no es posible obtener todos los requerimientos antes de comenzar el desarrollo de una aplicación, sino que estos van emergiendo a partir de la experimentación con la aplicación o partes de esta. Por lo tanto, lo antes que se pueda contar con el *feedback* del usuario sobre la utilidad de la aplicación, más fácil será incorporar los cambios requeridos en el producto. Este principio está ligado al principio de anticipación al cambio, y es otro de los principios en los que se basa la evolucionabilidad.

Hooker (1996) propone los siguientes principios:

- a. **La razón de que exista todo:** un sistema de software existe para dar valor a sus usuarios y todas las decisiones deben tomarse en consideración a esto. Por ello, es importante que antes de especificar un requerimiento, una funcionalidad o las plataformas del *hardware*, es necesario plantearse *¿Esto agrega valor real al sistema?*, si la respuesta es “no”, entonces no lo haga. Todos los demás principios apoyan en este.
- b. **Mantenerlo sencillo:** todo diseño debe ser tan simple como sea posible, esto facilita conseguir un sistema que sea comprendido más fácilmente y que sea susceptible de recibir mantenimiento. Simple tampoco significa “rápido y sucio”. Con frecuencia, se requiere mucha reflexión y trabajo con iteraciones múltiples para poder simplificar. Lo importante es que este principio nos permita obtener un software más fácil de mantener y menos propenso al error.

Índice

Primer bimestre

Segundo bimestre

Solucionario

Referencias bibliográficas

- c. **Mantener la visión:** una visión clara es esencial para el éxito de un proyecto de *software*. Comprometer la visión de la arquitectura de un sistema de *software* debilita y, finalmente, hará que colapsen incluso los sistemas bien diseñados. Tener un arquitecto que pueda mantener la visión y que obligue a su cumplimiento garantiza un proyecto de *software* muy exitoso.
- d. **Otros consumirán lo que usted produce:** en algún momento, alguien más usará, mantendrá, documentará o, de alguna forma, dependerá de su capacidad para entender un sistema que fue desarrollado; así que siempre se debe establecer especificaciones, diseñar e implementar con la seguridad de que alguien más tendrá que entender lo que usted haga. Es necesario pensar en aquellos que deben dar mantenimiento y ampliar el sistema. Mantener en la mente que hacer su trabajo más fácil agrega valor al sistema.
- e. **Ábrase al futuro:** un sistema con larga vida útil tiene más valor. En los ambientes de cómputo actuales –donde las especificaciones cambian de un momento a otro, y las plataformas de *hardware* quedan obsoletas con solo unos meses de edad– es común que la vida útil del *software* se mida en meses y no en años. Sin embargo, los sistemas de *software* con verdadera **fortaleza industrial** deben durar mucho más tiempo. Para tener éxito en esto, los sistemas deben ser fáciles de adaptar a esos y otros cambios. Los sistemas que lo logran son los que se diseñaron para ello desde el principio.
- f. **Planee por anticipado la reutilización:** la reutilización ahorra tiempo y esfuerzo. La reutilización del código y de los diseños se ha reconocido como uno de los mayores beneficios de usar tecnologías orientadas a los objetos. Para reforzar las posibilidades de la reutilización, se requiere reflexión

Índice

Primer bimestre

Segundo bimestre

Solucionario

Referencias bibliográficas

y planeación. La planeación anticipada en busca de la reutilización disminuye el costo e incrementa el valor tanto de los componentes reutilizables como de los sistemas en los que se incorpora.

- g. *¡Piense!*: pensar en todo con claridad antes de emprender la acción casi siempre produce mejores resultados. Cuando se piensa en algo, es más probable que se haga bien; si usted piensa en algo y aun así lo hace mal, eso se convierte en una experiencia valiosa. La aplicación de los primeros seis principios requiere pensar con intensidad, por lo que las recompensas potenciales son enormes.

Mall (2018) considera que los principios desplegados por la ingeniería de software sirven para superar las limitaciones cognitivas humanas y, en su estudio, se queda con dos de los propuestos por Ghezzi et al. (1991) y por Hooker (1996), que son la **abstracción** y la **descomposición**.

Una reflexión interesante manifiesta Pressman (2010): “Si todo ingeniero y equipo de software tan sólo siguiera los siete principios de Hooker, se eliminarían muchas de las dificultades que se experimentan al construir sistemas complejos basados en computadora”.

¿Qué tal le parecen estos principios? ¿Considera usted que son realmente aplicados en la industria del desarrollo de software o no?
Se lo invita a investigar sobre la **crisis del software** –alrededor del año 1968– y, de esa manera, identifique que tan alejados de esa realidad nos encontramos.

1.3. Relación del software con la industria

La industria del software ha crecido de manera exponencial y es uno de los factores dominantes en la economía del mundo industrializado, por lo que resultaría difícil operarlo en la modernidad que nos encontramos (Pressman, 2010).

El *software* es parte de los elementos de un sistema de información que se encuentra conformado, además, por el *hardware*, los datos y las redes de comunicaciones, en el que cada uno de ellos es esencial para poder operar día a día en cada uno de los sectores de la sociedad. Nos encontramos cada vez más modernizados y con acciones automatizadas; por esta razón, el *software* también ha tenido que someterse a este ritmo acelerado de las Tecnologías de la Información y la Comunicación (TIC) , que se han expandido en todas las regiones del mundo.

Uno de los elementos innatos de las TIC, que ha permitido globalizarlas, es sin duda el internet, por la comunicación, el acceso a la información, las transacciones, en fin, por todas las bondades que ofrece. Además de todo lo bueno que han logrado, hay que estar consciente de que también hay aspectos negativos, como los fraudes, y de que hoy en día se habla de la ciberseguridad y de los muchos desafíos que se van generando.

El *software* apoya a cada uno de los sectores económicos que brindan diferentes servicios, tanto públicos como privados, tales como salud, educación, banca, transporte, comunicaciones, entretenimiento o comercio, para que puedan operar y competir de una manera óptima. Es decir, el *software* apoya a estos sectores en la dinamización que requieren por el efecto globalizado, además de la innovación y los cambios tecnológicos.

Índice

Primer bimestre

Segundo bimestre

Solucionario

Referencias bibliográficas

1.3.1. Importancia

La importancia del *software* ha ido en aumento desde su aparición. Es muy evidente que hoy en día todas las organizaciones dependen, en mayor o menor medida del *software* para operar; el *software* está presente en la vida pública y privada de todos y se vuelve cada vez más importante. El *software* es fundamental, porque afecta a casi todos los aspectos de la vida y ha invadido el comercio, la cultura y las actividades cotidianas; en ese entonces, Pressman (2010) ya destaca cómo nos hemos seguido beneficiando una década después.

La fabricación y la distribución industrial están completamente computarizadas; el sistema financiero, el entretenimiento, incluida la industria musical, los juegos por computadora, el cine y la televisión usan *software* de manera intensiva (Sommerville, 2011).

El *software* está en todos lados, en las computadoras, en los dispositivos móviles, en los automóviles, en los electrodomésticos, en los robots o dispositivos mecanizados, entre otros; es lo que permite cumplir con las diferentes acciones para las que requiere el usuario final.

¿Conoce a detalle a algunos de los sectores económicos que han sido mencionados? Si su respuesta es afirmativa, revise cómo ha apoyado el *software* para su evolución.

1.3.2. Consideraciones finales

Para una organización o para cualquier sector económico, es imprescindible que se apoye en la tecnología para lograr sus objetivos y generar ventajas competitivas, de tal manera que su modelo de negocio, procesos y sistemas de información sean la fortaleza para ese cometido.

El *software* es parte de la tecnología, por lo que no puede faltar dentro de la organización; este lo puede obtener mediante el desarrollo propio, o mediante su compra o adquisición. Para decidirlo debe considerar algunos puntos relevantes como los que se mencionan a continuación:

- Tener muy claras las funciones que requiere, considerando los procesos, subprocessos, actividades y personas, es decir, identificar claramente cuál es el valor agregado que desea obtener.
- Identificar el factor técnico; responder si la organización cuenta con el personal técnico requerido para el desarrollo, además del *hardware*, redes.
- Identificar el factor económico para verificar qué resulta más conveniente, desarrollarlo o comprarlo; verificar pagos por mantenimiento, licencias y de implementación.

Estos son solo algunos criterios que deben considerarse, además de constatar que su equipo, o la empresa de desarrollo por contratar, asegure la calidad en el resultado que se pretende obtener siguiendo una efectiva ingeniería de *software* mediante sus procesos que lo garantizan.



Actividades de aprendizaje recomendadas

Revise el capítulo 1 del libro de Pressman (2010) para reforzar los contenidos analizados. Diríjase a la sección **Problemas y puntos por evaluar** (pp. 21-22).

Índice

Primer
bimestre

Segundo
bimestre

Solucionario

Referencias
bibliográficas

Haga una lectura comprensiva del [Tema 1. Introducción a la Ingeniería de Software](#) del [OCW-Ingeniería de Software I](#) de la [Universidad de Cantabria](#) (Blanco et al., 2011) y refuerce los conocimientos adquiridos hasta el momento.

Es hora de medir su conocimiento desarrollando la siguiente autoevaluación:



Autoevaluación 1

Estimado estudiante: mediante este cuestionario usted pondrá a prueba lo aprendido hasta el momento. En las siguientes preguntas, revise cada ítem y seleccione la respuesta correcta. Recuerde que sus resultados constituyen un reflejo del autoaprendizaje de la asignatura.

1. ¿Qué es el software?
 - a. Conjunto de procedimientos y programas que se ejecutan en una computadora para cumplir con una tarea o una acción específica.
 - b. Conjunto de funciones y códigos que se ejecuta en un computador.
 - c. Conjunto de instrucciones en código binario que se encarga de ejecutar una acción del sistema operativo.
2. Es un conjunto de conjunto de programas, instrucciones y reglas informáticas para ejecutar ciertas tareas en una computadora:
 - a. Algoritmo.
 - b. Ingeniería del software.
 - c. Software.
3. El paquete de Matlab, ¿a qué dominio de aplicación del software corresponde?
 - a. Software de ingeniería y ciencias.
 - b. Software de aplicación.
 - c. Software de línea de productos.

Índice

Primer bimestre

Segundo bimestre

Solucionario

Referencias bibliográficas

4. Seleccione cuál de las opciones planteadas son características de un software heredado:
 - a. Resultan costosos de mantener y tienen alto riesgo en su evolución.
 - b. Es conveniente renovarlos siempre, sin importar su tamaño y complejidad.
 - c. Evolucionan a ambientes de cómputo sofisticados y resuelven necesidades específicas de negocios.
5. La ingeniería de software es:
 - a. Una disciplina que se encarga del desarrollo y mantenimiento del *software*.
 - b. Una disciplina que se encarga del desarrollo, operación y mantenimiento del *software*.
 - c. Una disciplina que se encarga de la operación y mantenimiento del *software*.
6. ¿En qué etapa cronológica surge el concepto de la ingeniería de software?
 - a. Segunda etapa cronológica.
 - b. Tercera etapa cronológica.
 - c. Cuarta etapa cronológica.
7. Los programas que se construyen con carácter de especializados corresponden al dominio:
 - a. *Software* de ingeniería y ciencias.
 - b. *Software* de aplicación.
 - c. *Software* de línea de productos.

Índice

Primer bimestre

Segundo bimestre

Solucionario

Referencias bibliográficas

8. Ayudan a establecer un conjunto de herramientas para una práctica sólida de la ingeniería de software:
 - a. Principios.
 - b. Técnicas.
 - c. Métodos.
9. Un sistema complejo puede dividirse en piezas más simples llamadas módulos, esto corresponde al principio de:
 - a. Abstracción.
 - b. Generalidad.
 - c. Modularidad.
10. Es un proceso mediante el cual se identifican los aspectos relevantes de un problema ignorando los detalles, esto corresponde al principio de:
 - a. Abstracción.
 - b. Generalidad.
 - c. Modularidad.

[Ir al solucionario](#)

¿Cómo le fue en la autoevaluación?, espero que muy bien.

Puede verificar sus respuestas que se encuentran al final de este texto-guía. Si no consiguió un buen resultado, es necesario que revise nuevamente los puntos que aún no estén claros, o persistan dudas. Recuerde interactuar con su tutor y de hacer uso de los medios de comunicación que le brinda la UTPL; esto es muy importante que lo tenga siempre presente.

Hemos finalizado con el estudio de la unidad 1.

¡Felicitaciones y avance!

Índice

Primer bimestre

Segundo bimestre

Solucionario

Referencias bibliográficas

Resultado de aprendizaje 2

Conoce diferentes enfoques para abordar procesos de desarrollo o implantación de aplicaciones en diferentes contextos.

Contenidos, recursos y actividades de aprendizaje

Estimado estudiante: iniciará el estudio de la segunda unidad de esta asignatura, al tener claro lo qué es el *software* y cómo influye en una organización o en una actividad en particular, y sobre todo que conocemos de la existencia de la disciplina de la ingeniería del software, que es la que permite desarrollarlo, operarlo y mantenerlo. No obstante, ahora la pregunta que viene bien hacerse es ¿Cómo?

Esta pregunta será respondida a lo largo de esta segunda unidad mediante la identificación de lo que es un proceso de desarrollo; los modelos propuestos y de las metodologías que nos permiten abordar de manera adecuada un desarrollo de un sistema o de un *software*.

Se lo invita a descubrir este apasionante tema.

¡Inicie!



Semana 3



Unidad 2. Procesos de ingeniería de software

Como ya se analizó en la unidad 1, la ingeniería de software define un conjunto de elementos para lograr que el desarrollo de software sea fiable. Por lo tanto, recapitule lo resaltado por Pressman (2010), la ingeniería de software incluye un proceso, métodos y herramientas, fundamentada en el compromiso con la calidad. Esto se representa en la Figura 1.

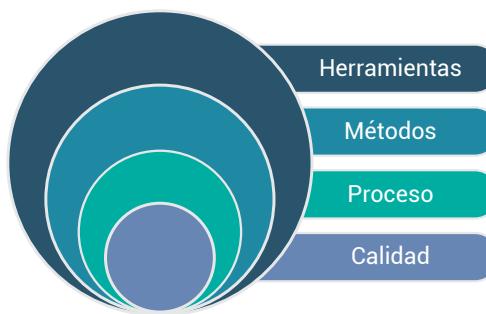


Figura 1. Elementos de la ingeniería del software

Nota: Adaptada de Pressman (2010)

Continuando con Pressman (2010), un **proceso** es un **conjunto de actividades, acciones y tareas** que se ejecutan cuando va a crearse algún producto, en el que:

- i. Una **actividad** busca lograr un objetivo amplio del trabajo, por ejemplo, la comunicación con los participantes.
- ii. Una **acción** es un conjunto de tareas que produce un producto importante del trabajo, por ejemplo, un modelo del diseño de la arquitectura.

[Índice](#)[Primer bimestre](#)[Segundo bimestre](#)[Solucionario](#)[Referencias bibliográficas](#)

- iii. Una **tarea** se centra en un objetivo pequeño pero bien definido que produce un resultado tangible, por ejemplo, realizar una prueba unitaria.

Para Sommerville (2011), un **proceso de software** es una **serie de actividades relacionadas** que conducen a la elaboración de un producto de software.

De acuerdo con los autores de esta rama, existen diferentes procesos de software, en los que las principales diferencias que surgen son en el número de actividades y en los nombres para ellas. En la Tabla 4 se detallan las actividades propuestas por dos autores.

Tabla 4. Actividades del proceso de software

Pressman (2010)	Sommerville (2011)
<ol style="list-style-type: none">1. Comunicación: antes del inicio del trabajo, es importante comunicarse y colaborar con el cliente. Se deben reunir los requerimientos que ayuden a definir las características y funciones del software.2. Planeación: define el trabajo de ingeniería de software, describe las tareas técnicas por realizar, los riesgos probables, los recursos que se requieren, los productos del trabajo que se obtendrán y la programación de las actividades.3. Modelado: un ingeniero de software crea modelos a fin de entender mejor los requerimientos del software y el diseño que los satisfará.4. Construcción: esta actividad combina la generación de código y las pruebas que se requieren para descubrir errores.5. Despliegue: el software se entrega al consumidor, quien lo evalúa y que le da retroalimentación, que se basa en esa evaluación.	<ol style="list-style-type: none">1. Especificación del software: definición de la funcionalidad del software y de las restricciones de su operación.2. Diseño e implementación del software: desarrollo del software para cumplir con las especificaciones.3. Validación del software: validar el software para asegurarse de que cumple con lo que el cliente quiere.4. Evolución del software: el software tiene que evolucionar para satisfacer las necesidades cambiantes del cliente.

Índice

Primer bimestre

Segundo bimestre

Solucionario

Referencias bibliográficas

Hay que considerar que los procesos de software pueden ser muy complejos y esto dependerá de la naturaleza de los proyectos; muchos equipos de desarrollo pueden definir su propio proceso, sin embargo, lo verdaderamente importante es que sí lo tengan definido y lo sigan correctamente. Se menciona que los procesos de software se pueden clasificar en dos categorías: 1) **dirigidos por un plan**, en el que todas las actividades se definen anticipadamente, y 2) **procesos ágiles**, en los que la planificación es incremental y puede ser cambiante de acuerdo con las necesidades del cliente.

Debe considerar que todo en el mundo es muy cambiante o volátil, todo cambio es muy rápido, por lo que es necesario que los procesos de software también se alineen a este ritmo, y se verifiquen de no incluir técnicas obsoletas, además de monitorear las mejores prácticas en la industria de la ingeniería de software. Otro punto importante es tener en cuenta que los procesos de software pueden mejorarse con la estandarización de los procesos, tal como lo indica Sommerville (2011), además, esto apoya a introducir nuevos métodos y técnicas.

Hay que tener en cuenta que en la actualidad la exigencia de un software para una empresa se ha convertido en una necesidad imperativa, lo que ha llevado a que ese requerimiento demande ser mejor, más barato y que se lo entregue en plazos cada vez más cortos. Esto puede ser logrado orientándose siempre hacia la mejora de procesos de software con el fin de aumentar la calidad de este producto, reducir sus costos y acelerar el proceso.

2.1. Modelos de procesos de software

Un modelo de proceso de software es la representación abstracta de un proceso y puede desarrollarse desde varias perspectivas; muestra las actividades implicadas en un proceso, los artefactos usados en este, las restricciones que se aplican al proceso y los roles de las personas que lo ejecutan (Sommerville, 2011).

También es posible decir que un modelo de proceso define un conjunto prescrito de elementos del proceso y un flujo predecible para el trabajo del proceso (Pressman, 2010).

Estos autores lo que quieren decir es que un modelo de proceso de software orienta a seguir las fases o etapas del desarrollo de software (ciclo de vida del desarrollo de software), definir las actividades y desarrollar las tareas necesarias para obtener un producto *software*, de acuerdo con el rol que cada miembro del equipo de desarrollo deba cumplir. Todo esto de acuerdo con el tipo de modelo de proceso por seguir, que, de forma general, son:

- Modelo en cascada.
- Modelo en espiral.
- Modelo de proceso incremental.
- Modelo de proceso evolutivo.
- Modelo de proceso iterativo.

En la 2 encontrará un mapa mental con las ideas más relevantes de los procesos y modelos de procesos de *software*.

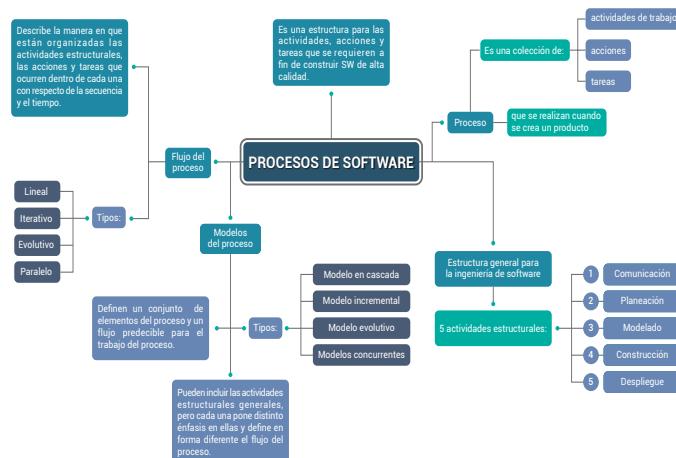


Figura 2. Mapa mental de procesos y modelos de procesos de *software*

2.1.1. Modelo cascada (*waterfall*)

El modelo en cascada fue el primero en ser propuesto para el desarrollo de software y se lo conoce como el ciclo de vida clásico; en este, las actividades fundamentales del proceso de desarrollo de software se llevan a cabo como fases separadas y consecutivas (Ojeda y Fuentes, 2012).

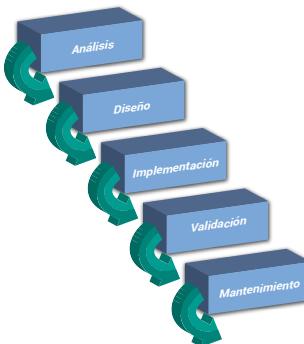


Figura 3. Actividades del modelo en cascada.

Nota: Adaptada de Pressman (2010).

En la Figura 3 se detallan las cinco actividades del modelo en cascada, en ella se refleja el enfoque sistemático y secuencial o lineal que define a este modelo. Esto último quiere decir que se cambia de actividad solo cuando se han conseguido los objetivos de la anterior.

A continuación, se indica brevemente lo que significa y qué se hace en cada una de las actividades:

- i. **Análisis:** consiste en definir cuáles son los servicios y objetivos del sistema por desarrollar. Se trabaja con los clientes y usuarios finales del sistema, es necesario también establecer las restricciones de este. El resultado de esta actividad es obtener las especificaciones del sistema; a este documento se lo conoce como la Especificación de Requerimientos del Sistema (ERS).

- ii. **Diseño:** en esta actividad se realiza el modelado de cómo funcionará, de manera general, el sistema; se distinguen cuáles son los requerimientos de *software* y cuáles los de *hardware* y red. Aquí es necesario establecer la arquitectura¹ completa del sistema.
- iii. **Implementación:** consiste en trasladar el diseño en código.
- iv. **Validación:** esta actividad consiste en verificar y comprobar que el sistema o *software* realiza correctamente las tareas para las que fue solicitado y cumplen con su especificación. Estas pruebas deben probarse por separado (cada módulo) y de forma integral (el sistema como un todo). Cuando todas estas pruebas sean exitosas, puede entregarse el sistema al cliente.
- v. **Mantenimiento:** en esta última actividad del modelo en cascada, una vez que el sistema ha sido puesto en funcionamiento, se corrigen los errores que no fueron descubiertos antes y se pueden mejorar unidades o módulos del sistema.

2.1.2. Modelo de proceso incremental

En el modelo de proceso incremental se realiza y se entrega el sistema en partes pequeñas, pero con una característica principal, son utilizables. A cada una de estas partes se conoce como incremento. En otras palabras, se puede decir que el sistema se divide en un conjunto de particiones que se van desarrollando en diferentes momentos o ritmos y que se van integrando de forma incremental, conforme se van completando.

¹ La arquitectura del software debe modelar la estructura de un sistema y la manera en la que los datos y componentes del procedimiento colaboran entre sí (Pressman, 2010).

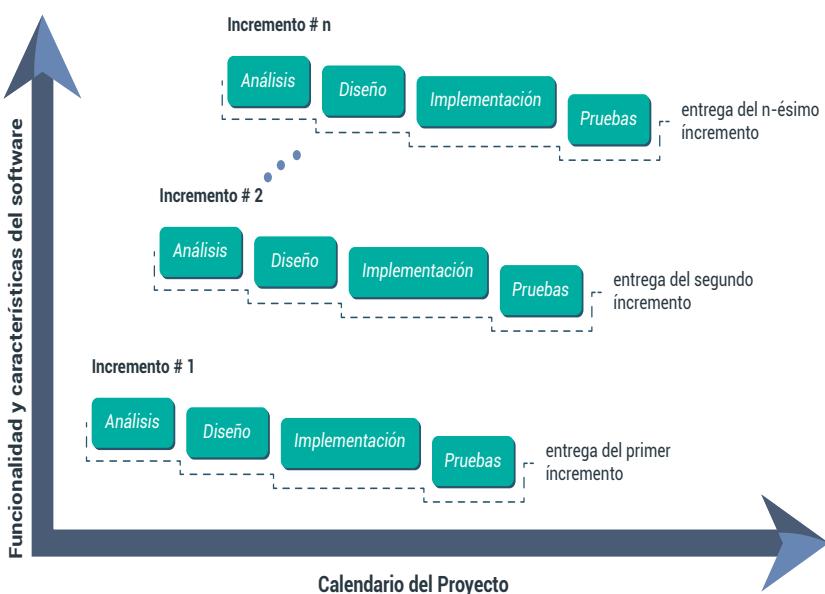


Figura 4. Modelo de proceso incremental

Nota: Adaptada de (Pressman, 2010)

En la Figura 4 se observa el modelo de proceso incremental. Las actividades que se incluyen muchas veces son las del modelo en cascada —**análisis, diseño, implementación y pruebas**—, al igual se realizan en secuencia lineal, pero cada una de ellas producen los incrementos.



Actividades de aprendizaje recomendadas

Para afianzar lo estudiado, revise y analice el siguiente video. En este se habla de los modelos de procesos de software que profundizará en la unidad 2; por ello, para esta actividad puede ver hasta el minuto 4:37, que son los temas que ha abordado durante esta tercera semana. El título del video es [Ingeniería del Software II-Tema 1. Revisión modelos de proceso-Fernando Pereñiguez](#).

Una vez que ha visto el video, responda a lo siguiente:

¿Qué tal le ha parecido? ¿Le ha sido útil esta explicación complementaria? ¿Las actividades propuestas en el modelo lineal (cascada) son las mismas que ha estudiado aquí en el texto-guía?; si no lo son, ¿por qué ocurre esto?, y finalmente, al finalizar las iteraciones en el modelo incremental, ¿se puede ir enseñando el incremento al cliente para obtener información y así mejorar el proceso? ¿Usted considera que esto puede ser efectivo?



Semana 4

¡Qué bien!, ya ha conocido lo que es un proceso de software y los modelos que le apoyan para seguirlo. Estudió al modelo más básico y del cual todos los demás lo que tratan de hacer es mejorarlo; abordó el modelo en cascada o lineal –su flujo es secuencial–, luego estudió el incremental, pero no son los únicos que existen, por ello, en esta nueva semana se adentra en los siguientes. ¡Ánimos!

2.1.3. Modelo de proceso evolutivo

El modelo de proceso evolutivo es iterativo y se caracteriza por la manera en la que permite desarrollar versiones cada vez más completas del software, genera en cada iteración una versión final que se acerca al producto total (Pressman, 2010). Este mismo autor indica que es necesario comprender bien el conjunto de requerimientos o el producto básico, pero los detalles del producto o extensiones del sistema aún están por definirse. Hay que tener claro que un software evoluciona en el tiempo, ya que en una empresa sus requerimientos de negocio cambian, por lo que se necesita de un modelo de proceso diseñado explícitamente para adaptarse a un producto que evoluciona con el tiempo.

Este modelo de proceso tiene dos variantes: los prototipos y el modelo espiral.

Prototipos

Ayudan a entender de mejor manera los requisitos, o lo que hay que hacer. Se realiza un diseño rápido y un prototipo de esos requerimientos identificados, se lo presenta al cliente y, en ese momento, se llega a otra característica importante de este modelo: la retroalimentación.

Para Sommerville (2011), los prototipos de un sistema permiten a los usuarios ver qué tan bien el sistema los apoya, puesto que este modelo ayuda con la selección y validación de sus requerimientos hasta obtener nuevas ideas para ellos y descubrir áreas de fortalezas y debilidades en el software. En la Figura 5 se presenta el esquema y las actividades del modelo de prototipos.

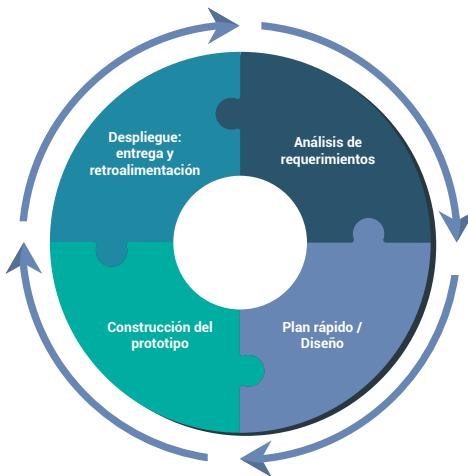


Figura 5. Modelo de proceso evolutivo-prototipos

Nota: Adaptada de (Pressman, 2010)

Modelo espiral

Este modelo evolutivo comprende las mejores características del modelo de proceso en cascada y del de prototipos. Además, se incluye actividades del análisis de alternativas, identificación y reducción de riesgos (Tinoco, 2010).

Fue propuesto, en 1988, por Barry Boehm y representa una secuencia de actividades con retroalimentación de una actividad a otra que forma así una espiral, tal como se muestra en la Figura 6. Ahí se puede observar que cada ciclo en la espiral representa una fase (actividad) del proceso de desarrollo de software.

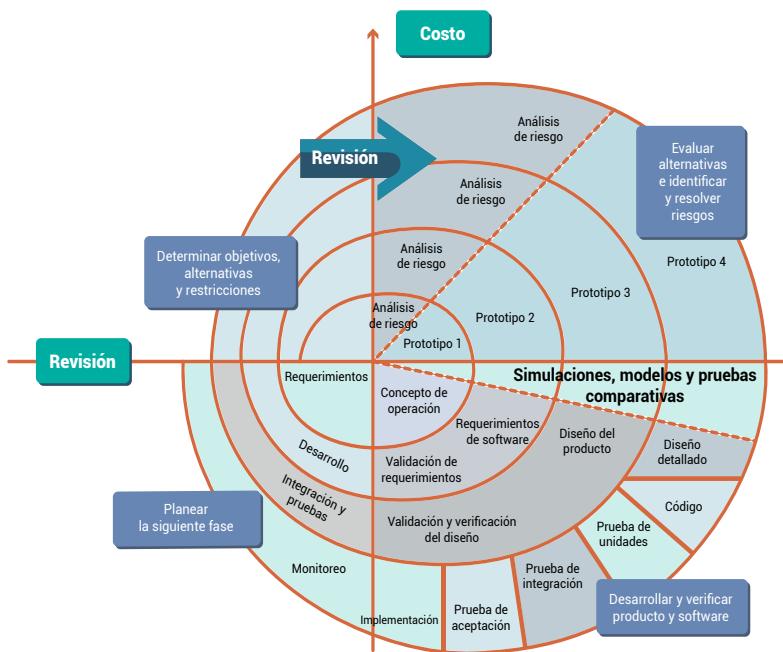


Figura 6. Modelo de proceso evolutivo en espiral

Nota: Tomada de programas.cuaed.unam.mx/repositorio/moodle/pluginfile.php/1150/mod_resource/content/1/contenido/index.html

El modelo en espiral se divide en cuatro sectores, tal como se presenta en la Figura 6. A continuación se describen:

- **Determinar objetivos, alternativas y restricciones:** es el símil del análisis del modelo en cascada. Se definen los objetivos del software y, además, se identifican los riesgos para identificar alternativas y restricciones del proyecto.
- **Evaluar alternativas e identificar y resolver riesgos:** es necesario analizar los riesgos identificados y evaluar las alternativas para reducir esos riesgos.
- **Desarrollar y verificar producto y software:** en este sector se realiza el diseño, la programación e implementación, además de realizar las pruebas necesarias para validar lo desarrollado.

- **Planear la siguiente fase:** se evalúa si es necesario una nueva iteración, si es así, entonces, se la planifica.

En Pressman (2010) se menciona que el modelo de desarrollo espiral es un generador de modelo de proceso impulsado por el riesgo que tiene dos características principales: 1) tiene un enfoque cíclico para el crecimiento incremental del grado de definición de un sistema y su implementación, mientras que disminuye su grado de riesgo, y 2) tiene un conjunto de puntos de referencia de anclaje puntual para asegurar el compromiso del participante con soluciones factibles y mutuamente satisfactorias.

El mismo autor considera que con el empleo del modelo en espiral, el software se desarrolla en una serie de entregas evolutivas, en el que en las primeras iteraciones se tiene un prototipo, y en las iteraciones posteriores se obtienen versiones cada vez más completas del sistema. Cada iteración finaliza con una evaluación de riesgos y un prototipo.

Gutiérrez (2011) menciona que el modelo en espiral es considerado el mejor modelo para el desarrollo de sistemas grandes, pero que no es aconsejable para sistemas pequeños por su alta complejidad. Otros puntos por considerar es que se requiere de experiencia para la identificación de riesgos y es costoso.

2.1.4. Modelo de proceso iterativo

Para Tinoco et al. (2010), el modelo de proceso iterativo fracciona el proyecto en iteraciones de diferente longitud, cada una de ellas produce un producto completo y entregable. En Gutiérrez (2011) se indica que cada iteración refina lo realizado en la anterior, con el fin de mejorar los productos (entregables). Esto quiere decir que cada iteración mejora más la calidad del producto. En este modelo las principales características son la revisión y la mejora, y es justamente esta la diferencia con el modelo incremental, ya que en él se agrega funcionalidad al cumplirse el ciclo, mientras que en el iterativo se mejora la funcionalidad.

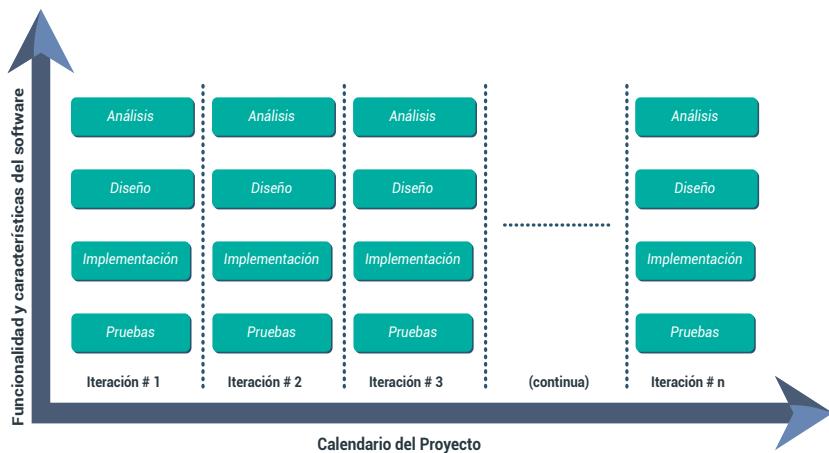


Figura 7. Modelo de proceso iterativo

Nota: Adaptada de <http://aurumsol.com/espanol/articulos/art1/art1-4.html>

2.1.5. Proceso unificado

En el libro de Pressman (2010) se hace un análisis detallado de este modelo, en el que se destaca lo siguiente:

El proceso unificado intenta obtener los mejores rasgos y características de los modelos tradicionales del proceso del software. Este reconoce la importancia de la comunicación con el cliente y los métodos directos para describir su punto de vista respecto de un sistema. Mediante los casos de uso, hace énfasis en la importancia de la arquitectura del software y que el arquitecto se centre en las metas correctas —comprensible, flexibilidad para cambios futuros y reutilización—.

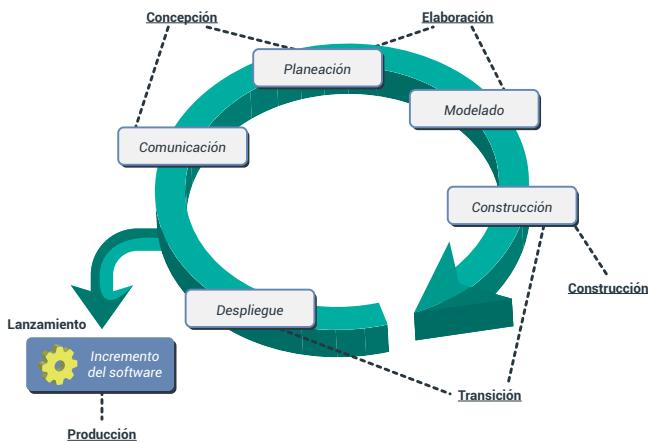


Figura 8. Fases del proceso unificado

Nota: Adaptada de Pressman (2010)

Las fases del proceso unificado tienen un objetivo similar al de las actividades de los procesos de desarrollo de software mencionadas por Pressman (2010) en la Tabla 4 de la semana 3. A continuación se describen las cinco fases:

- Concepción:** agrupa las actividades de comunicación con el cliente y la planeación. Se identifican los requerimientos —descritos mediante casos de uso—; se define la arquitectura, y se identifican los recursos y riesgos, además de planificar las fases.

Índice

Primer bimestre

Segundo bimestre

Solucionario

Referencias bibliográficas

- ii. **Elaboración:** se incluyen las actividades de planeación y modelado. En esta etapa se mejoran y amplían los casos de uso preliminares diseñados en la fase anterior y se aumenta la representación de la arquitectura (los modelos del caso de uso, de requerimientos, del diseño, de la implementación y del despliegue). Al finalizar esta fase, es necesario revisar con mayor detalle el plan a fin de asegurar que el alcance, los riesgos y las fechas de entrega siguen siendo razonables, como resultado de esto se puede modificar el plan.
- iii. **Construcción:** aquí se desarrollan los componentes del software que harán que cada caso de uso sea operativo para los usuarios finales. Se completan los modelos de requerimientos y diseño, con el fin de que reflejen la versión final del incremento de software. Se implementan en código fuente todas las características y funciones necesarias para el incremento de software. Es importante que a medida que se implementan los componentes, se diseñen y efectúen pruebas unitarias para cada uno, además de realizar las actividades de integración (ensamble de componentes y pruebas de integración). Se emplean casos de uso para obtener un grupo de pruebas de aceptación que se ejecutan antes de comenzar la siguiente fase del proceso unificado.
- iv. **Transición:** incluye las últimas etapas de la construcción y la primera parte de la actividad de despliegue general (entrega y retroalimentación). Se entrega el software a los usuarios finales para las pruebas y se reportan los defectos y los cambios necesarios. Además, el equipo de software genera los manuales de usuario, guías de solución de problemas, procedimientos de instalación, para el lanzamiento. Al finalizar esta fase, el software se convierte en un producto utilizable que se lanza.

- v. **Producción:** esta última coincide con la actividad de despliegue del proceso general (Tabla 4). Se vigila el uso que se da al software; se brinda apoyo para el ambiente de operación, y se reportan defectos y solicitudes de cambio para su evaluación.

Algo muy importante en este modelo es que al mismo tiempo que se llevan a cabo las fases de construcción, transición y producción, comience el trabajo sobre el siguiente incremento del software; es decir, las cinco fases del proceso unificado no ocurren en secuencia, sino que concurren en forma escalonada.



Actividades de aprendizaje recomendadas

- Ha culminado con la cuarta semana de estudio y para complementar con las temáticas analizadas, se lo invita a revisar y analizar el siguiente video. En este se habla de los modelos de procesos de software que profundizará en la unidad 2; por ello, para esta actividad puede ver desde el minuto 4:38, que corresponde a los temas abordados. El título del video es [Ingeniería del Sofware II-Tema 1. Revisión modelos de proceso-Fernando Pereñiguez](#).

Una vez que ha visto el video, responda a lo siguiente:

¿Le ha sido útil esta explicación complementaria? Un prototipo sirve para realizar una prueba de validación, ¿Qué hacemos luego? ¿Cuál modelo es generado luego con el fin de mejorar el de prototipado?

- Haga una lectura comprensiva del [Tema 1: El proceso de desarrollo de software](#) del [OCW Diseño Avanzado de Software](#). Analice la importancia del proceso de desarrollo de software y describa cada uno de los modelos de proceso.

- Adicionalmente, puede observar el siguiente video que presenta, de una manera dinámica, un resumen sobre el proceso unificado. El título del video es [Proceso Unificado](#).



Semana 5

Estimado estudiante: se encuentra en la mitad del primer bimestre, ¿**Cómo le están pareciendo los temas?** Se espera que le resulten de interés y de mucho provecho. Una vez finalizados los modelos de procesos de desarrollo de software, es importante profundizar un poco más; por ello, en esta semana abordará las famosas metodologías de desarrollo de software, inicialmente con las conocidas metodologías tradicionales —que se basan en los modelos tradicionales que revisamos hasta la semana 4—.

Ahora que ya conoce sobre los procesos y los modelos del desarrollo de software, es momento de introducir el término de metodologías del desarrollo de software.

Inicie con una definición. El *Diccionario de la lengua española* indica que una “metodología es un conjunto de métodos que se siguen en una investigación científica o en una exposición doctrinal” (2020). Para el desarrollo de software, una metodología es un conjunto de técnicas que permiten desarrollar software con calidad. Las metodologías del desarrollo de software son **sistemáticas, predecibles y repetibles**, cuyo objetivo es **mejorar** la productividad en el desarrollo y la calidad del producto software.

A partir de la definición de los modelos de procesos de software, se denominan diferentes metodologías que siguen a esos modelos y permiten definir procesos específicos para planificar, ejecutar y controlar el desarrollo del software.

Índice

Primer bimestre

Segundo bimestre

Solucionario

Referencias bibliográficas

Se han dividido en dos tipos: las metodologías tradicionales, que analizará a continuación y, más adelante, a las metodologías ágiles.

2.2. Metodologías tradicionales

En las metodologías tradicionales se concibe al proyecto como uno solo y con una estructura bien definida. El proceso es de manera secuencial, en una sola dirección y sin marcha atrás; el proceso es rígido y no cambia. Los requerimientos son acordados de una vez y para todo el proyecto, lo que demanda grandes plazos de planeación previa y poca comunicación con el cliente (Montero et al., 2018).

Las metodologías tradicionales son reconocidas por su orientación a la planeación.

2.2.1. MSF

Microsoft Solutions Framework (MSF), como su nombre indica, fue propuesta por Microsoft en 1994, y ha sido el resultante del conjunto de las mejores prácticas de sus proyectos de desarrollo de software. Se basa específicamente en un conjunto de principios, modelos, disciplinas, conceptos, directrices y prácticas. Esta metodología ha retomado algunas de las características propias de las metodologías tradicionales. MSF es una guía de desarrollo de software flexible que permite aplicar, de manera individual e independiente, cada uno de sus componentes; es escalable y asegura resultados con menor riesgo y de mayor calidad, y se centra en el proceso y las personas (Pérez, 2011).

Características: algunas de las características más representativas de MSF se han rescatado del sitio oficial de Microsoft² y del estudio de Pérez (2011):

² [https://docs.microsoft.com/es-es/previous-versions/jj161047\(v=vs.120\)?redirectedfrom=MSDN](https://docs.microsoft.com/es-es/previous-versions/jj161047(v=vs.120)?redirectedfrom=MSDN)

- Alinea los objetivos de negocio y de tecnología.
- Establece de manera clara los objetivos, los roles y las responsabilidades;
- Sigue los modelos de procesos en cascada y espiral.
- Implementa un proceso iterativo controlado por hitos o puntos de control.
- Controla los riesgos.
- Responde con eficacia ante los cambios.
- Es adaptable, flexible y escalable, e independiente de tecnologías.

Principios fundamentales de MSF

Los principios y actitudes de MSF sirven de guía al equipo de proyecto. MSF se basa en nueve principios fundamentales:

1. Fomentar una comunicación abierta: compartir niveles de información apropiados entre todos los miembros del equipo y en toda la empresa.
2. Intentar lograr una visión compartida: esto permite empoderar a los miembros del equipo y les permite actuar con agilidad para poder tomar decisiones rápidas.
3. Empoderar a los miembros del equipo.
4. Establecer responsabilidades claras y compartidas.
5. Ofrecer valor incremental: 1) lo que se entrega tiene un valor óptimo para las partes interesadas, y 2) determinar las **frecuencias de entrega** óptimas.
6. Mantenerse ágil, esperar cambios y adaptarse a ellos: Esto significa que una organización está preparada para los cambios y puede adaptarse y ajustarse sin contratiempos.
7. Invertir en la calidad: se debe incorporar de manera proactiva al ciclo de vida de entrega de la solución.

8. Aprender de todas las experiencias: 1) a nivel de proyecto; 2) a nivel individual, y 3) a nivel de la organización.
9. Colaborar con clientes internos y externos.

Proceso: el proceso o el ciclo de vida de MSF se presenta en la Figura 9; en ella se identifican las cinco fases que lo componen: visión, planificación, desarrollo, estabilización e implantación.

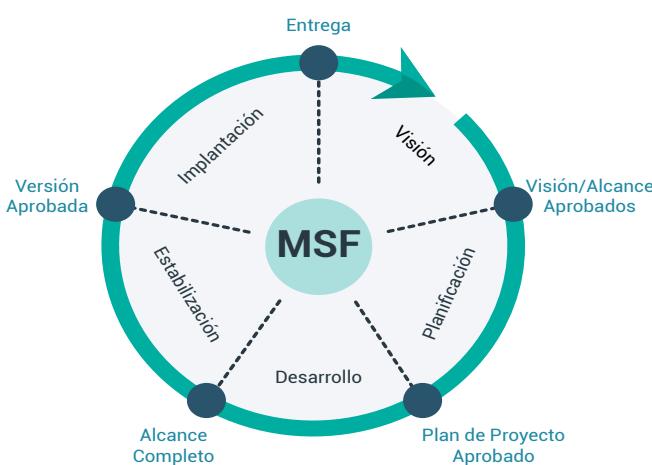


Figura 9. Proceso de MSF

Nota: Tomada de darjelingsilva.files.wordpress.com/2018/05/3-metd-msf.pdf

Visión y alcance: en esta primera fase el objetivo es tener una visión común y clara de lo que se quiere lograr. Se definen los líderes y responsables del proyecto, los requerimientos funcionales, sus alcances y restricciones. Los resultados de esta fase son el documento de visión y alcance y la identificación de los riesgos del proyecto.

Planificación: el equipo prepara los planes de trabajo, estimaciones de costos y cronograma del proyecto, además de diseñar la solución y especificación funcional. Algunos de los entregables son el documento de especificaciones y cronograma del proyecto.

Desarrollo: en esta fase se realiza la construcción de los componentes (documentación y **código**), **las pruebas y la infraestructura**. Los entregables de esta fase son **código fuente y ejecutables**, actas de aceptación de entregas parciales, plan de pruebas y arquitectura.

Estabilización: en esta fase las pruebas hacen énfasis en el uso y operación a partir de condiciones reales. El equipo se enfoca en priorizar, resolver errores y preparar la solución para el lanzamiento. La fase se culmina con la aceptación de las pruebas.

Implantación: en esta fase el equipo libera el *software*; implanta la tecnología base y los componentes relacionados; estabiliza la instalación; traspasa el proyecto al personal soporte y operaciones, y obtiene la aprobación final del cliente.

2.2.2. RUP

El Proceso Unificado Racional o Rational Unified Process (RUP por sus siglas en inglés) es un proceso formal y constituye una de las metodologías más utilizadas para el análisis, implementación y documentación de sistemas orientados a los objetos. RUP es un proceso para el desarrollo de un proyecto de *software* que define claramente **quién, cómo, cuándo y qué** debe hacerse en el proyecto (Fernández y Cadelli, 2014).

Características: algunas de las características de la metodología RUP se mencionan a continuación:

- **Es guiada y manejada por casos de uso:** los casos de uso se implementan para asegurar que toda la funcionalidad se realiza en el sistema, además de que contienen las descripciones de las funciones y afectan a todas las fases y vistas.

- **Está centrada en la arquitectura:** la arquitectura se describe mediante diferentes vistas del sistema. Es importante establecer una arquitectura básica pronto, realizar prototipos, evaluarla y, finalmente, refinarla durante el curso del proyecto.
- **Sigue los modelos de procesos de desarrollo de software, iterativo e incremental:** es práctico dividir los grandes proyectos en proyectos más pequeños, en el que cada uno se desarrolla en una iteración que tiene como resultado un incremento.
- **Su desarrollo está basado en componentes:** como se indicó en la característica anterior, además de dividir en proyectos más pequeños, es necesario dividir a todo el sistema en partes que se denominarán componentes. Esto permite que el sistema se vaya creando a medida que se obtienen sus componentes.
- **Utiliza lenguaje unificado de modelado (UML):** para el desarrollo de todos los modelos necesarios para el sistema.
- **Es un proceso integrado:** es necesario que cada uno de los elementos que integran el proceso, tales como los ciclos, fases, flujos de trabajo, mitigación de riesgos, control de calidad, gestión del proyecto y control de configuración se integren.

Proceso: el proceso o el ciclo de vida de RUP se presenta en la Figura 10; en ella se identifican los elementos principales que lo componen: fases, iteraciones y disciplinas.

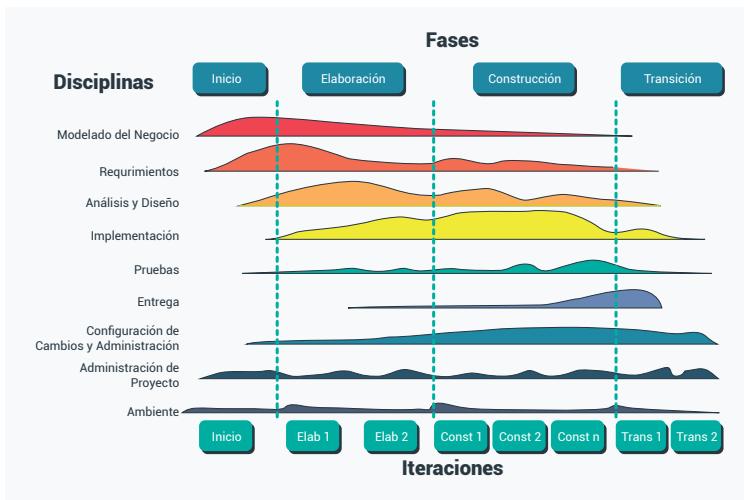


Figura 10. Proceso RUP

Nota: Tomada de Martínez (2014)

A continuación, se describen cada una de las cuatro fases que incluye RUP:

- **Inicio:** en esta fase se explora el problema o las necesidades y se identifica el objetivo del proyecto, su factibilidad, estimación de costos. El objetivo principal es tener claro el alcance del proyecto e identificar los requerimientos. Algunos de los **productos** principales de esta fase son visión del negocio, especificación de requisitos, modelos de casos de uso y planificación de las fases.
- **Elaboración:** es considerada como una de las fases más delicadas y difíciles técnicamente, tal como mencionan Kroll y Krutchen (2003), por las actividades que se llevan a cabo y los productos resultantes. Los objetivos que persigue es definir, validar y obtener una arquitectura base, completar la visión, planificar la fase de construcción.

Índice

Primer bimestre

Segundo bimestre

Solucionario

Referencias bibliográficas

- **Construcción:** aquí se lleva a cabo la construcción del producto y se procede a su implantación y respectivas pruebas. En esta fase se realizan tantas iteraciones hasta que se termine, y el producto está listo para ser entregado al cliente. Los productos de esta fase son modelos completos de los casos de uso, análisis, diseño, despliegue e implementación, arquitectura, riesgos mitigados y manual de usuario.
- **Transición:** en esta fase se garantiza que el producto entregado es el correcto y es puesto en las manos del usuario final. Los objetivos son conseguir que el usuario se valga por sí mismo y que el producto final cumpla con los requisitos esperados, que funcione y satisfaga al usuario.

Recuerde que se inicia este tema indicando que RUP indica el qué, cómo, cuándo y quién lo hace. En la Tabla 5 y Figura 11 se indica de qué forma se relacionan.

Tabla 5. Elementos de RUP

Interrogante	Elementos RUP
¿Quién?	Roles: definen el comportamiento y las responsabilidades de los miembros del equipo.
¿Cómo?	Actividades: corresponde a crear o actualizar algún producto.
¿Qué?	Productos (artefactos): son los resultados tangibles del proyecto, lo que se va creando y usando hasta obtener el producto final.
¿Cuándo?	Flujos de trabajo: corresponden a las disciplinas y se clasifican en flujos del proceso (modelado del negocio, requerimientos, análisis y diseño, implementación, pruebas, entrega), y flujos de apoyo (configuración de cambios, administración de proyecto, ambiente) y es en las iteraciones donde se ejecutan con mayor o menor intensidad, tal como se reflejan en la Figura 10.



Figura 11. Roles, actividades y productos

Nota: Adaptada de [The Rational Unified Process Made Easy. A Practitioner's Guide to the RUP: A Practitioner's Guide to the RUP](#). Kroll (2003)

De acuerdo con Kroll et al (2003), la Figura 11 representa la relación que existe entre los roles, actividades y productos en RUP. El rol indica quién está haciendo el trabajo; una actividad describe cómo se hace el trabajo y el producto captura lo que se hace.



Actividades de aprendizaje recomendadas

Ha finalizado la quinta semana de estudio. Se lo invita a realizar las siguientes actividades para complementar los temas:

Índice

Primer bimestre

Segundo bimestre

Solucionario

Referencias bibliográficas



Semana 6

Estimado estudiante: como puede identificar, el proceso de desarrollo de software es complejo pero muy fascinante. Imagínese estar dentro de un equipo de desarrollo de software en el que aplique cada una de las actividades y flujos de trabajo que propone las metodologías que ha estudiado hasta este momento. ¿Considera que se las puede mejorar? Una vez que finalice esta sexta semana de estudio, usted podrá dar su respuesta.

¡Ánimos que va por buen camino!

2.3. Metodologías ágiles

En vista de que se siguen manteniendo los mismos problemas del pasado entorno al desarrollo de software, un grupo de 17 expertos de esta industria se reunieron del 11 al 13 febrero del 2001, en Utah, Estados Unidos. Algunos de ellos eran representantes de las metodologías que son reconocidas como **ágiles**, tales como Extreme

Programming (XP), SCRUM, DSDM, Adaptive Software Development, Crystal, Feature-Driven Development (FDD), Pragmatic Programming. El resultado de esta reunión fue la declaración del **Manifiesto ágil**³.

Las metodologías de desarrollo ágil están orientadas a las personas y a la entrega de valor y flexibilidad en un tiempo corto; tratan de adaptarse a las necesidades que muchas de las veces son dinámicas y cambiantes. Para lograr estas características, se orientan hacia los modelos de procesos iterativos e incrementales.

2.3.1. Manifiesto ágil



Estamos poniendo al descubierto mejores métodos para desarrollar software...

Figura 12. Reunión del Manifiesto ágil

Nota: Tomada de agilemanifesto.org

Como se indicó en el punto 2.3, resultado de la reunión del 2001 fue el Manifiesto para el desarrollo ágil de software, cuando declararon que se encontraban descubriendo formas mejores de desarrollar software, mediante la formulación de valores y principios que definen y rigen a este movimiento. En la Figura 12 se observa la imagen icónica de la reunión.

³ agilemanifesto.org/iso/es/manifesto.html

Es importante tener claro que en el **Manifiesto ágil** se definen preferencias, no alternativas, y se fomenta la atención en ciertas áreas sin excluir a otras.

2.3.2. Principios y paradigmas

Antes de hacer hincapié en los 12 principios que se declaran para estas metodologías, es importante partir de lo que estos expertos han aprendido a valorar mediante su experiencia y a los que se conocen como los valores ágiles. Estos valores son cuatro y se los presenta en la Figura 13.



Figura 13. Valores ágiles

En la declaratoria del **Manifiesto ágil** se incluyen los 12 principios que se enlistan a continuación en la Figura 14:

- 1** La mayor prioridad es **satisfacer al cliente** mediante la entrega temprana y continua de software con valor.
- 2** Aceptar que **los requisitos cambien**, incluso en etapas tardías del desarrollo. Los procesos Ágiles aprovechan el cambio **para proporcionar ventaja competitiva al cliente**.
- 3** Entregar **software funcional frecuentemente**, entre dos semanas y dos meses, con preferencia al **periodo de tiempo más corto posible**.
- 4** Los **responsables de negocio y los desarrolladores trabajan juntos de forma cotidiana** durante todo el proyecto.
- 5** Los proyectos se desarrollan en torno a **individuos motivados**. Hay que **darles el entorno y el apoyo que necesitan, y confiarles la ejecución del trabajo**.
- 6** El **método más eficiente y efectivo de comunicar información** al equipo de desarrollo y entre sus miembros es la **conversación cara a cara**.
- 7** El **software funcionando** es la medida principal de progreso.
- 8** Los **procesos Ágiles promueven el desarrollo sostenible**. Los promotores, desarrolladores y usuarios deben ser capaces de mantener un ritmo constante de forma indefinida.
- 9** La **atención continua** a la excelencia técnica y al buen diseño mejora la Agilidad.
- 10** La **simplicidad**, o el arte de maximizar la cantidad de trabajo no realizado, es esencial.
- 11** Las mejores arquitecturas, requisitos y diseños emergen de **equipos auto-organizados**.
- 12** A intervalos regulares el **equipo reflexiona sobre cómo ser más efectivo** para ajustar y perfeccionar su comportamiento en consecuencia.

Figura 14. Principios ágiles

Con base en los valores y principios que los definen, es posible identificar características para este movimiento ágil:

- Es altamente colaborativo.
- Está centrado en la calidad
- Sus equipos son autoorganizados, generalistas y especialistas.
- Se basa en la práctica y no en la teoría.



Actividades de aprendizaje recomendadas

Para afianzar lo estudiado, se le propone que realice un cuadro en el que pueda agrupar los principios ágiles de acuerdo con los valores, con el fin de identificar claramente cómo se sustentan esos principios en los valores.

Adicionalmente puede ingresar al [sitio web del Manifiesto ágil](#) para que lo analice a profundidad.



Semana 7

Estimado estudiante: se encuentra ya en la última semana de estudio de la unidad 2. Ya conoce acerca del segundo enfoque metodológico para el desarrollo de software, el ágil, identifica sus inicios y las premisas que sigue. Ahora es momento de conocer algo más, su ciclo de desarrollo. Y, para finalizar, es importante que realice un análisis comparativo entre las metodologías ágiles y tradicionales.

2.3.3. Ciclo de desarrollo

Las metodologías ágiles siguen los modelos de procesos iterativos e incrementales, tal como vio en los modelos de procesos **tradicionales**, en el punto 2.1. La mayoría se basa en el modelo en cascada y luego se han incluido variantes para apoyar en la mejora del proceso como tal. El ciclo de desarrollo de las metodologías ágiles no es la excepción; tal como se observa en la Figura 15, se sigue viendo las actividades del ciclo en cascada, pero cada uno de ellas se realiza en un tiempo definido (iteración) en el que, como

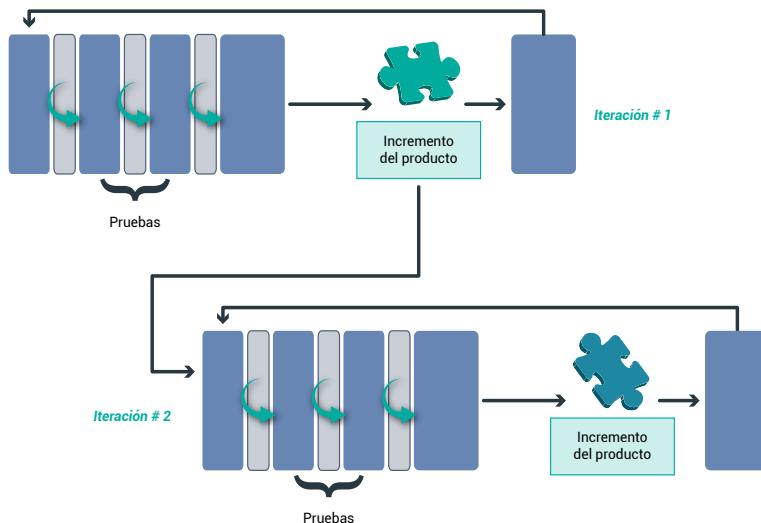


Figura 15. Ciclo de vida de las metodologías ágiles

Cada una de las metodologías que sigue la filosofía ágil define su propio proceso o ciclo de vida, esto se profundizará en la unidad 3 a partir del segundo bimestre.

Índice

Primer bimestre

Segundo bimestre

Solucionario

Referencias bibliográficas

Algo importante, y que representa uno de los cambios en este enfoque ágil, corresponde a los requisitos del sistema y que son recogidos en la actividad del análisis, tal como en los modelos tradicionales. Los requerimientos o requisitos se transforman a historias de usuario (HU). Las HU son un enfoque de requerimientos ágil que se focaliza en establecer conversaciones acerca de las necesidades de los clientes (Izaurrealde, 2013).

Una historia de usuario, definida como *User Story* (US por sus siglas en inglés), representa una forma de obtener requisitos de usuario. Se indica mediante frases simples lo que tiene que hacer el sistema, desde el punto de vista del usuario, es decir, se describen mediante lenguaje natural las características de una funcionalidad que compone a un sistema; esto se logra con frases cortas y sencillas, que reflejan la necesidad de un usuario.

Las HU tienen ciertas características que se mencionan a continuación:

- Están centradas en describir **quién** la propone, **qué** se tiene que hacer, **por qué** se propone, pero **NO** tienen que indicar cómo se hace.
- Expresan necesidades funcionales que aportan valor.
- Tienen formato libre, no hay un formato estricto que haya sido establecido.
- Son consideradas como una versión simplificada de los casos de uso.
- Son descritas por los propios usuarios.
- Describen las pruebas de aceptación, es decir, los criterios con los que el usuario las dará por finalizadas.

Índice

Primer bimestre

Segundo bimestre

Solucionario

Referencias bibliográficas

Pese a que no existe un formato definido, como se indica en las características, sí existe algo muy básico que muchas veces se escriben en post-its (notas), y se pegan en los tableros para que estén visibles a todo el equipo de desarrollo del proyecto. En ellos se describe lo siguiente:

**"Yo como <rol o tipo de usuario>,
quiero/deseo/necesito <funcionalidad u objetivo>,
para <valor o beneficio a lograr>"**

Vea un ejemplo para clarificar lo mencionado:

Requerimiento: *Identificar las farmacias de turno cerca de mi ubicación.*

La HU que describa esa funcionalidad o necesidad se describiría de la siguiente manera:

Como paciente,
deseo conocer la lista de farmacias de turno cercanas a mi ubicación
para compra de medicina en horarios nocturnos o fines de semana.

Además de la descripción en las HU, es posible incluir conversaciones con el usuario que permitan desagregar más conocimiento de la funcionalidad requerida y las pruebas o criterios de aceptación que permitirán indicar si una historia está finalizada.

Cuando una HU describe varias funcionalidades a la vez y no permite completarla en la iteración planificada, porque se requiere de mayor tiempo, entonces se tiene una **historia épica**. Una historia épica es una HU demasiado grande, o puede ser ambigua. Cuando se tiene este, caso es importante dividirla en historias más pequeñas para cumplir con uno de los atributos que deben incluir las HU.

Existen buenas prácticas para la escritura de HU que en la literatura se encuentran como atributos para estas. El primero que vamos a mencionar es el acrónimo INVEST⁴ (por sus siglas en inglés): *Independent* (independiente), *Negotiable* (negociable), *Valuable* (valiosa), *Estimable* (estimable), *Small* (pequeña), *Testable* (validable). A continuación, se detalla cada atributo:

- **Independiente:** evitar la dependencia entre HU, esto facilita la priorización y la planificación.
- **Negociable:** una HU no se convierte en una obligación contractual, sino en un compromiso de establecer conversaciones con los clientes o usuarios para convenir los detalles.
- **Valiosa:** las HU deben brindar valor al cliente y a los usuarios. Con el fin de lograr este atributo, existe la recomendación de que sean ellos mismos quienes las escriban o participen de su escritura.
- **Estimable:** es importante estimar el tamaño de una HU o el tiempo necesario para su implementación. Existen problemas que se pueden dar y que complican la estimación, estos son 1) falta de conocimiento del dominio; –es necesaria una mayor conversación–, y 2) la historia es muy grande.
- **Pequeña:** una HU debe ser implementada en una iteración, de lo contrario complica su estimación.
- **Validable:** una HU debe ser probada para comprobar si esta ha sido terminada. Se suele aplicar el enfoque del Desarrollo Guiado por Pruebas conocido como Test Driven Development (TDD por sus siglas en inglés).

⁴ xp123.com/articles/invest-in-good-stories-and-smart-tasks/

[Índice](#)[Primer bimestre](#)[Segundo bimestre](#)[Solucionario](#)[Referencias bibliográficas](#)

2.4. Metodologías tradicionales vs. metodologías ágiles

En esta instancia, una vez que ha conocido acerca de las metodologías iniciales, conocidas en la literatura como tradicionales, rígidas o pesadas por su orientación a la planificación y las metodologías ágiles, es un buen momento para analizarlas en conjunto.

En la Tabla 6 se presentan algunas de las características de las metodologías tradicionales versus las metodologías ágiles según el punto de vista de diferentes autores:

Tabla 6. Comparación entre las metodologías tradicionales vs. las metodologías ágiles

	Modelos tradicionales	Modelos ágiles
Hirsch (2005)	<ul style="list-style-type: none">▪ Plan de proyecto detallado de principio a fin. Definido al inicio del proyecto▪ Fase de especificación dedicada▪ Se firman los requisitos iniciales después se establece un riguroso régimen de solicitud de cambio▪ Documentos de requisitos exhaustivos▪ Especificaciones completas de arquitectura y diseño▪ La arquitectura y el diseño tratan de adaptarse a futuras ampliaciones;▪ El trabajo de programación se concentra en la fase de implementación;▪ Fase de pruebas al final del proyecto;	<ul style="list-style-type: none">▪ Plan general para el proyecto global, planes detallados por iteración▪ Las necesidades evolucionan en el curso del proyecto▪ No hay un régimen de solicitud de cambio; o es muy relajado▪ Fácil acceso al cliente, no se depende de una documentación exhaustiva de los requisitos.▪ Mínimo trabajo de arquitectura y diseño por adelantado;▪ La arquitectura es validada por iteración;▪ El trabajo de programación se extiende a todo el proyecto;▪ Las pruebas se extienden a todo el proyecto;

	Modelos tradicionales	Modelos ágiles
	<ul style="list-style-type: none"> ▪ Las pruebas son diseñadas y ejecutadas por especialistas en pruebas; ▪ Papel formal de control de calidad, y ▪ El rol de QA es responsable de las revisiones formales e informales, el proceso de desarrollo, la gestión de la configuración, las pruebas, las inspecciones del código. 	<ul style="list-style-type: none"> ▪ Las pruebas funcionales son especificadas y ejecutadas por los usuarios finales; ▪ No hay un papel explícito de control de calidad, y ▪ La calidad es el resultado de cómo se trabaja.
Canós et al. (2012)	<ul style="list-style-type: none"> ▪ Basadas en normas provenientes de estándares seguidos por el entorno de desarrollo; ▪ Cierta resistencia a los cambios; ▪ Impuestas externamente; ▪ Proceso muy controlado, con numerosas políticas/normas, y ▪ Existe un contrato prefijado; ▪ El cliente interactúa con el equipo de desarrollo mediante reuniones; ▪ Grupos grandes y posiblemente distribuidos; ▪ Más artefactos; ▪ Más roles, y ▪ La arquitectura del software es esencial y se expresa mediante modelos. 	<ul style="list-style-type: none"> ▪ Basadas en heurísticas provenientes de prácticas de producción de código ▪ Especialmente preparados para cambios durante el proyecto ▪ Impuestas internamente (por el equipo) ▪ Proceso menos controlado, con pocos principios ▪ No existe contrato tradicional o al menos es bastante flexible; ▪ El cliente es parte del equipo de desarrollo; ▪ Grupos pequeños (10 integrantes) y trabajando en el mismo sitio; ▪ Pocos artefactos; ▪ Pocos roles, y ▪ Menor énfasis en la arquitectura del software

	Modelos tradicionales	Modelos ágiles
Cadavid et al. (2013)	<ul style="list-style-type: none"> ▪ Predictivos; ▪ Orientados a procesos; ▪ Proceso rígido; ▪ Se concibe como un proyecto; ▪ Poca comunicación con el cliente; ▪ Entrega de <i>software</i> al finalizar el desarrollo, y ▪ Documentación extensa. 	<ul style="list-style-type: none"> ▪ Adaptativos; ▪ Orientados a personas; ▪ Proceso flexible; ▪ Un proyecto es subdividido en varios proyectos más pequeños; ▪ Comunicación constante con el cliente; ▪ Entregas constantes de <i>software</i>, y ▪ Poca documentación.

Existe una institución que se encarga de evaluar los proyectos de desarrollo de *software* a nivel mundial, se trata de Standish Group⁵, que nació en 1985, en Estados Unidos. Su misión es cambiar el mundo en la forma en que se gestione el desarrollo de *software*, y para ello estudian cuáles son los fallos o las causas de los fracasos en esta industria. Dentro de los parámetros que se evalúan es el tamaño de los proyectos, las metodologías que siguen, entre otros.

Su estudio lo reflejan en lo que denominan el Chaos Report, que analiza aproximadamente 50 000 proyectos de desarrollo de *software* a nivel mundial. En la Figura 16 se presentan los resultados de acuerdo con el tamaño del proyecto y la metodología que utilizan (ágil o cascada) del estudio realizado en 2018. Claramente se puede identificar que el tamaño de un proyecto afecta a la tasa de éxito de este, ya que apenas un 18% de los proyectos grandes llega a culminar con respecto al 59% de los pequeños. Otro punto interesante que se

⁵ www.standishgroup.com/

destaca es que el éxito de ellos ha llegado de la mano con lo ágil, por lo que en el estudio se destaca que estos dos factores, el tamaño y el enfoque, tienen más impacto en las tasas de éxito. Por lo tanto, los proyectos pequeños ágiles tienen mayor probabilidad de éxito que los demás.

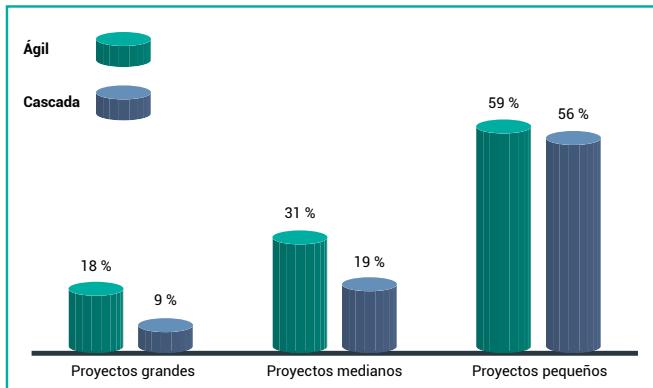


Figura 16. Tasas de éxito de un proyecto según el tamaño y enfoque

Nota: Tomada de Standish Group-Chaos Report 2018

Ahora se analiza en la Figura 17 el resultado del estudio del Chaos Report 2015, según el enfoque metodológico, en el que clasifican a los proyectos según el éxito o fracaso, y en un punto intermedio los cuestionados, que durante esos años resultaron el grupo mayoritario tanto para los que siguieron el enfoque ágil como para el cascada; sin embargo, se destaca más los resultados de los proyectos exitosos y fracasados, donde el resultado es evidente. Los proyectos ágiles exitosos triplican a los que siguen el enfoque en cascada, lo que implica que posiblemente conlleven a un mayor costo.

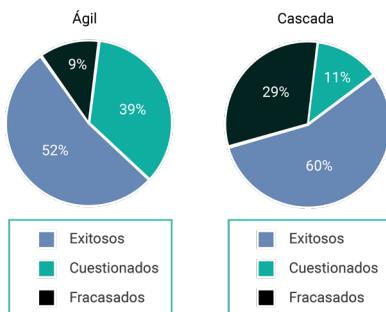


Figura 17. Tasas de éxito de un proyecto según el enfoque metodológico
Nota: Tomada de Standish Group-Chaos Report 2015

Se ha considerado importante este análisis para reflexionar nuevamente sobre la importancia de mantener una organización y seguir un proceso adecuado en los proyectos de desarrollo de software. La metodología debe ser acorde a las necesidades del proyecto, no es posible decir cuál es mejor; o que existe una general, todo depende de diversos factores como la naturaleza, tamaño y los recursos con los que se cuenta. Lo que sí es importante tomar en cuenta son los resultados del estudio representado en las dos figuras anteriores. Si la práctica induce a que los proyectos grandes están más orientados al fracaso, pues, entonces, es preferible dividirlo en fragmentos pequeños.



Actividades de aprendizaje recomendadas

Para culminar con la temática y afianzar lo presentado, se le recomienda que indague sobre otros estudios referentes al análisis entre los resultados de las metodologías tradicionales versus las ágiles, así como del Grupo Standish. ¿Qué encontró? ¿Los resultados son similares o difieren?

Es hora de medir su conocimiento desarrollando la siguiente autoevaluación:



Autoevaluación 2

Estimado estudiante: mediante este cuestionario, usted pondrá a prueba lo aprendido hasta el momento. En las siguientes preguntas, revise cada ítem y seleccione la respuesta correcta. Recuerde que sus resultados constituyen un reflejo del autoaprendizaje de la asignatura.

1. Es una estructura para las actividades, acciones y tareas para la construcción de software:
 - a. Modelo de proceso.
 - b. Flujo de proceso.
 - c. Proceso de software.

2. Las fases o etapas del modelo en cascada son:
 - a. Análisis, diseño, implementación validación y mantenimiento.
 - b. Análisis, arquitectura, desarrollo, validación y mantenimiento.
 - c. Análisis, desarrollo, pruebas y mantenimiento.

3. Es el primer modelo en incluir actividades de identificación y mitigación de riesgos:
 - a. Modelo espiral.
 - b. Prototipos.
 - c. RUP.

4. Las principales características de este modelo son la revisión y la mejora:
 - a. Modelo incremental.
 - b. Modelo evolutivo.
 - c. Modelo iterativo.
5. Esta metodología permite aplicar de manera individual e independiente cada uno de sus componentes:
 - a. MSF.
 - b. RUP.
 - c. Agile.
6. En RUP las actividades responden al:
 - a. ¿Qué?
 - b. ¿Cuándo?
 - c. ¿Cómo?
7. Es considerada una metodología ágil:
 - a. RUP.
 - b. DSDM.
 - c. Waterfall.
8. Es una característica del movimiento ágil:
 - a. Se basa en la práctica y no en la teoría.
 - b. Sus equipos no son autoorganizados.
 - c. Se centra en la arquitectura.
9. Las metodologías ágiles siguen los modelos:
 - a. Evolutivo-prototipado.
 - b. Iterativo-incremental.
 - c. Evolutivo-iterativo.

Índice

Primer bimestre

Segundo bimestre

Solucionario

Referencias bibliográficas

10. Los resultados del Chaos Report inducen a que:

- a. Los proyectos más grandes estén orientados al fracaso.
- b. Los proyectos más grandes estén orientados al éxito.
- c. Los proyectos más pequeños estén orientados al fracaso.

[Ir al solucionario](#)

¿Cómo le fue en la autoevaluación?, espero que muy bien.

Puede verificar sus respuestas con las que se encuentran al final de este texto-guía. Si no consiguió un buen resultado, es necesario que revise nuevamente los puntos que aún no estén claros, o persistan dudas. Recuerde interactuar con su tutor y de hacer uso de los medios de comunicación que le brinda la UTPL. Esto es muy importante que lo tenga siempre presente.

Ha finalizado con el estudio de la unidad 2 y del primer bimestre.

¡Felicitaciones!

Índice

Primer bimestre

Segundo bimestre

Solucionario

Referencias bibliográficas



Actividades finales del bimestre



Semana 8



Actividades de aprendizaje recomendadas

- Apreciado estudiante: ha llegado a la etapa final del primer bimestre, y seguro que usted cumplió con todas las actividades de aprendizaje. Como preparación para el examen presencial se sugiere que:
- Revise las unidades y subtemas estudiados en cada una de las semanas del bimestre, y refuerce los que considera necesarios;
- Analice los recursos facilitados en este texto-guía;
- Revise las autoevaluaciones de las unidades del bimestre;
- Prepárese para la evaluación presencial del primer bimestre y comuníquese con su tutor en caso de que se presenten dudas, y
- Consulte el horario y lugar para rendir la evaluación presencial de la asignatura.



Segundo bimestre

Resultado de aprendizaje 2

Conoce diferentes enfoques para abordar procesos de desarrollo o implantación de aplicaciones en diferentes contextos.

Contenidos, recursos y actividades de aprendizaje

En el primer bimestre estudió la importancia de las metodologías de desarrollo, los modelos de proceso de desarrollo de software, contextos tradicionales de desarrollo de software, y estudió una introducción y comparación de las metodologías ágiles y tradicionales. En este segundo bimestre revisará las principales metodologías ágiles, escenarios de automatización de software y nuevos paradigmas de desarrollo que están siendo utilizados actualmente en la industria de desarrollo. Tenga presente que la transformación digital de las organizaciones ha ampliado la demanda digital de los consumidores para utilizar canales digitales, mediante aplicaciones de software, que les permita acceder a los diferentes servicios que ofrecen las organizaciones, lo que ha conducido a los equipos de desarrollo a utilizar nuevas estrategias, metodologías y escenarios de automatización para lograr entregar nuevas características de software en el menor tiempo posible.

A continuación, iniciará la unidad 3, que aborda las metodologías ágiles XP y Scrum.



Semana 9



Unidad 3. Metodologías ágiles de desarrollo de software

Las metodologías ágiles han revolucionado el mundo del desarrollo de software; han permitido romper paradigmas tradicionales, rígidos y poco flexibles para abordar nuevas formas de desarrollar productos digitales de una forma más flexible, ágil, sencilla y que, sobre todo, garantice la calidad del software. Como ya se estudió en el primer bimestre, las metodologías ágiles basan su contexto de trabajo en valores y principios dictados por el **Manifiesto ágil**, producto de lo cual han surgido a lo largo de los años diferentes metodologías de trabajo que han ayudado a los ingenieros de software a afrontar nuevos desafíos y proyectos de software de una manera distinta y esquematizada. Las metodologías más destacadas son programación extrema o más conocida como XP (por sus siglas en inglés que corresponden a Extreme Programming), Scrum, Kanban, Agile Inception, entre otras. A continuación, estudiará las dos principales metodologías que son más utilizadas en la industria del software.

3.1. Programación extrema

Programación extrema es una de las principales metodologías de desarrollo de software ágil, creada por Kent Beck y difundida mediante el libro *Extreme Programming Explained: Embrace Change* (Kent y W., 1999). Aunque su origen data de años previos desde el lanzamiento del **Manifiesto ágil**, en 2001, esta metodología ágil se enfoca en la adaptabilidad y la sencillez en sus actividades de desarrollo. XP utiliza un esquema de retroalimentación rápida y gran poder de comunicación para maximizar el valor entregado a un cliente con especial énfasis en la priorización y validación de necesidades desde etapas tempranas de desarrollo de software. XP, sustenta sus prácticas y estrategias en cinco valores primordiales, que todo proyecto de desarrollo de software debe cumplir: 1) comunicación, 2) simplicidad, 3) retroalimentación, 4) coraje o valentía, y 5) el respeto. A continuación, en la Tabla 7, revisará cada uno de estos valores:

Tabla 7. Valores de XP

Valor	Descripción
Comunicación	<p>XP tiene como objetivo mantener comunicaciones efectivas y fluidas entre los miembros del proyecto y clientes. Para los programadores, la comunicación se enfoca en utilizar prácticas simples que implique un mínimo esfuerzo para que el equipo entienda que hace cada uno de los componentes, funciones o clases que se escriben en el código. Para esto, la forma más sencilla de comunicar es utilizando comentarios a medida que el código se vaya desarrollando. Otro método son las pruebas unitarias que describen la forma en cómo las clases y métodos interactúan para definir una funcionalidad.</p> <p>Con respecto a la comunicación que debe existir con los clientes, se considera al cliente como pieza fundamental del equipo de desarrollo, de manera que siempre esté disponible cuando el equipo de desarrollo lo necesite para explicar, fundamentar, aprobar o determinar una funcionalidad.</p>

[Índice](#)[Primer bimestre](#)[Segundo bimestre](#)[Solucionario](#)[Referencias bibliográficas](#)

Valor	Descripción
Simplicidad o sencillez	XP promueve mantener la simplicidad con respecto a la definición de actividades, toma de decisiones, diseño y construcción de componentes de software, para que estos elementos permitan a desarrolladores, diseñadores y otros roles del equipo mantener el enfoque en la construcción de elementos que sean fáciles de desarrollar y mantener. De la misma manera, la sencillez aplica para la especificación y la documentación del proyecto, no se requiere un amplio esfuerzo para realizar actividades que sean de soporte ya que la metodología, como su nombre lo sugiere, se enfoca en hacer un fuerte énfasis en la programación y no tanto en las actividades o entregables de soporte.
Retroalimentación	XP se enfoca en obtener la apreciación y opinión sobre las funcionalidades del proyecto que se van desarrollando en el menor tiempo posible, de forma que estos resultados posibiliten y minimicen el esfuerzo que deben realizar los programadores para refactorizar, o rehacer aquellas requerimientos o funcionalidades que no cumplen con lo especificado por el equipo y el cliente. De esta manera, el equipo, centra todos sus esfuerzos en construir y refactorizar el producto según sea necesario, siempre manteniendo el enfoque y esfuerzo sobre las actividades que agregan valor al proyecto.
Coraje o valentía	Este valor se enfoca en utilizar prácticas que impliquen valentía para asumir los resultados de la retroalimentación y coraje para aceptar que se debe reconstruir o descartar alguna porción de código que se ha construido. Otro enfoque de este valor es permitir a los desarrolladores sincerarse sobre sus actividades y tener la posibilidad de comentar al equipo abiertamente cuando se ha producido un error, o cuando no se ha podido avanzar en alguna funcionalidad por alguna dificultad técnica o de habilidad. Esto permite que el equipo tome las acciones que sean necesarias para reorganizar el trabajo y para mantener el impulso sobre las entregas planificadas.

Valor	Descripción
Respeto	XP promueve abiertamente el respeto en el equipo, para que se pueda consolidar una forma de trabajo armónica que saque el mejor rendimiento de todos los miembros del equipo. El respeto se manifiesta fundamentalmente entre los miembros del equipo, para que no se afecten negativamente los cambios en el código que impliquen esfuerzos adicionales, y que permitan cumplir con las fechas establecidas para que no demore el trabajo entre compañeros, especialmente cuando existe dependencia o paralelismo entre actividades, de forma que el equipo pueda mantener un ritmo de trabajo sostenible que le permita cumplir las metas del proyecto.

3.1.1. Roles de los miembros del equipo en XP

El equipo de XP se conforma por diferentes roles, que son complementarios entre sí. Los roles principales en XP son el programador, cliente, tester, entrenador o coach, encargado de seguimiento (*tracker*), consultor y gestor o *big boss*. Un miembro del equipo puede asumir más de un rol, por ejemplo, un programador puede compartir el rol de rastreador o *tester*, según sea necesario. Sin embargo, no es aconsejable que se compartan los roles de entrenador o gerente del proyecto; es importante que estos roles sean asumidos por personas que no formen parte del equipo de desarrollo para que sus intereses no afecten a las actividades del equipo.

Es importante destacar que XP es flexible, por lo que, para conformar un equipo de trabajo, es necesario conocer el contexto del proyecto y determinar los roles necesarios según sea necesario. A continuación, en la Tabla 8, revisará cada uno de estos roles:

Tabla 8. Roles de XP

Rol	Descripción
Programador	Es el responsable de implementar las HU del cliente, estimar el esfuerzo y se encarga de codificar las HU planificadas para cada una de las iteraciones.
Cliente	Parte interesada del proyecto de software de desarrollo. Como parte del equipo es quien determina la funcionalidad, valida y retroalimenta al equipo de desarrollo los resultados de la revisión funcional del producto en cada una de las iteraciones.
Tester	Encargado de ejecutar y difundir los resultados de las pruebas sobre cada una de las entregas realizadas, además, se encarga de gestionar las herramientas de soporte necesarias para validar todos los tipos de prueba que sean necesarios.
Entrenador o Coach	Responsable de evangelizar el proceso de desarrollo, se encarga de capacitar y guiar a los miembros del equipo sobre la metodología de desarrollo XP.
Encargado de seguimiento o traker	Responsable de dar seguimiento a la evolución de las tareas realizadas por los programadores, compararlas con las estimaciones iniciales y brindar información estadística sobre la evolución del proyecto.
Consultor	Miembro externo al equipo de proyecto. Es quien posee el conocimiento específico sobre algún tema necesario para el proyecto, también actúa como un rol de apoyo para resolver problemas específicos del proyecto.
Gestor (<i>big boss</i>)	Vínculo entre el cliente y los programadores. Este rol tiene altas capacidades de gestión y seguimiento, su labor es principalmente de coordinación entre las partes interesadas del proyecto.

3.1.2. Fases de la programación extrema

El proceso de desarrollo de software, basado en la metodología XP, consta de cuatro fases principales que se van desarrollando en ciclos iterativos, como se observa en la Figura 18. Parte por la fase de planificación, diseño, codificación y finaliza el ciclo con la ejecución de pruebas de funcionamiento antes de hacer un incremento formal de una o un conjunto de características planificadas.

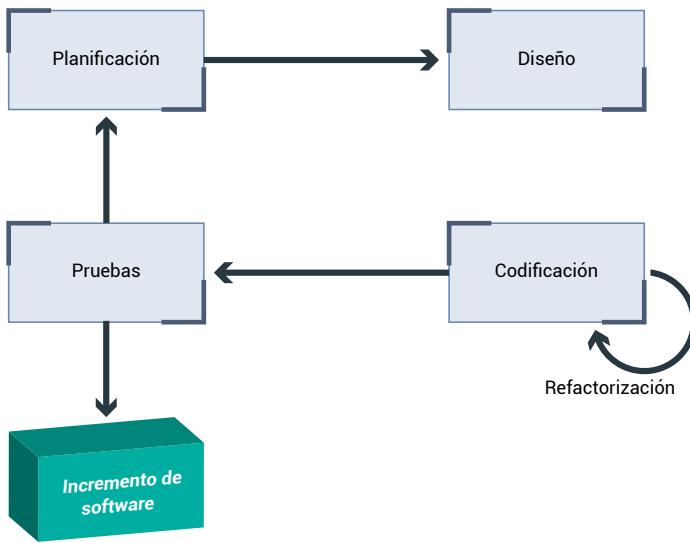


Figura 18. Fases de la programación externa

A continuación, estudiará cada una de estas fases:

Planificación: en la fase de planificación, la metodología plantea un acercamiento continuo entre las partes del proyecto. La planificación inicia desde la identificación de las HU, en estas se describen los requerimientos del cliente, criterios de aceptación, y se planifican las iteraciones necesarias para completar el producto. Las HU son evaluadas por los programadores para determinar el esfuerzo en tiempo de desarrollo. Este insumo permite que el equipo planifique y distribuya el trabajo entre los miembros del equipo para completar las iteraciones.

Algunas prácticas importantes para considerar en esta fase se resumen en la Tabla 9.

Tabla 9. Prácticas de la fase de planificación de XP

Práctica	Descripción
Historias de usuario	Representa una descripción de las necesidades del cliente en lenguaje natural. Generalmente este esquema es muy utilizado en la gran mayoría de metodologías ágiles para la especificación de requisitos que se maneja en un contexto tradicional de desarrollo. Mínimamente contienen una descripción de las necesidades del cliente y el criterio de aceptación con el que se evaluará la funcionalidad a la que representa.
Plan de entregas o <i>release plan</i>	Establece el conjunto de lanzamientos que se planificarán para el proyecto y se compone de las HU que conformarán un lanzamiento. Este plan es determinado entre los interesados del proyecto, incluye hitos y fechas de cada lanzamiento.
Plan de iteraciones	Determina la cantidad de iteraciones que se realizarán mientras dure el proyecto. Contiene un conjunto de las HU comprometidas para que los programadores construyan las funcionalidades. Es importante mencionar que las primeras iteraciones generalmente contienen las HU de mayor prioridad, lo que permite dar al cliente el mayor valor de su proyecto en el menor tiempo posible.
Reuniones diarias	Son reuniones que se realizan diariamente entre los miembros del equipo para conocer el avance y restricciones encontradas sobre las actividades planificadas.

Diseño: en esta fase se enfocan los esfuerzos en crear los diseños arquitectónicos, de interfaces o de elementos que permitan a los programadores comprender el problema y determinar la forma en cómo se deben construir las funcionalidades. Es importante que los diseños sean simples y claros, y que representen claramente lo que se pretende explicar.

Algunas prácticas importantes para considerar se resumen en la Tabla 10.

Tabla 10. Prácticas de la fase de diseño de XP

Práctica	Descripción
Simplicidad	Se enfoca en uno de los valores más importantes de la metodología: realizar diseños simples y claros que se implementen rápidamente es más importante que realizar especificaciones complejas y que tomen demasiado tiempo de realizar. Los diseños se deben enfocar en un funcionamiento básico y minimalista.
Soluciones	Más conocidas como <i>spike</i> , son pequeños elementos de prueba que se realizar fuera del contexto del proyecto para dar solución a un problema específico que afecte a la construcción de una funcionalidad de software.
Refactorización	Actividad que consiste en escribir nuevamente el código de algún componente de software sin la necesidad de cambiar la funcionalidad, sino enfocada en escalar el código de la aplicación hasta hacerlo lo más simple y funcional posible.
Metáforas	Consiste en crear una descripción sencilla del contexto del proyecto, esto permite que todo el equipo de proyecto tenga una comprensión común de lo que se pretende alcanzar. Según Tomayko y Herbsleb (2003), una metáfora tiene dos propósitos: una es la comunicación que maneje un lenguaje sencillo y entendible sobre el proyecto, y, la segunda razón, la metáfora debe contribuir a la construcción de la arquitectura de software.

Codificación: esta fase se enfoca específicamente en las actividades de codificación que son ejecutadas por los programadores, en esta fase, se traducen las HU a funcionalidades de software.

Las prácticas para considerar en esta fase se resumen en la Tabla 11.

Tabla 11. Prácticas de la fase de codificación de XP

Práctica	Descripción
Disponibilidad del cliente	El cliente es una pieza fundamental dentro del proyecto. Este participa activamente en las actividades de desarrollo brindando todos los detalles necesarios a los programadores durante la etapa de desarrollo y pruebas del sistema, de forma que esta interacción permite que el equipo tenga la retroalimentación necesaria a tiempo para alcanzar los hitos esperados del proyecto.
Uso de estándares	XP promueve el uso de estándares como un fuerte impulso en las actividades de desarrollo y simplicidad para la construcción de los componentes de software, esto facilita la codificación entre los miembros del equipo. Es importante que previo a codificar, los miembros del equipo definan los estándares que se utilizarán mientras dure el proyecto, y así evitar confusiones y complicaciones sobre el entendimiento del código.
Programación dirigida por pruebas	Más conocida TDD, es una práctica de desarrollo de software que prioriza la codificación de las pruebas unitarias antes de escribir el código que conformará una funcionalidad; esto hace que cuando se creen los artefactos, primero se ejecuten las pruebas unitarias y luego se compile el código. Esta práctica permite identificar defectos en etapas tempranas para que sean corregidos a tiempo.
Programación en pares	XP promueve la programación en pares, esto significa que un par de programadores escriben la misma HU, lo que brinda la posibilidad de minimizar los errores evaluando cada una de sus soluciones. Esta práctica se enfoca específicamente en garantizar la calidad y minimizar los problemas en etapas o fases posteriores a la entrega de productos.
Integraciones permanentes	Esta práctica promueve la necesidad de los programadores de trabajar siempre sobre la última versión disponible del código fuente. Esto se logra principalmente estableciendo reglas de desarrollo que permitan que, al finalizar el día, todos los desarrolladores integren el código realizado para que esté disponible para el equipo.
Propiedad colectiva del código	En XP, ningún programador es dueño de las funcionalidades que se construyan; todos los miembros del equipo están dispuestos a revisar y contribuir con ideas y prácticas que contribuyan a la calidad del proyecto.

[Índice](#)[Primer bimestre](#)[Segundo bimestre](#)[Solucionario](#)[Referencias bibliográficas](#)

Práctica	Descripción
Ritmo sostenido	Todo el equipo de desarrollo debe mantener un ritmo constante sobre el desarrollo del proyecto. Este ritmo se debe planificar y organizar en función de las características del proyecto, de forma que se aproveche al personal únicamente el tiempo que sean productivos y no se generen sobrecargas.
40 horas semanales	XP promueve la productividad del equipo, esto implica que la organización del trabajo no debe superar las 8 horas diarias y 40 semanales. XP considera que se debe garantizar la productividad y motivación de los miembros del equipo y evitar cualquier tipo de sobrecarga.

Pruebas: en esta fase se ejecutan las pruebas, se da seguimiento y se corrigen las funcionalidades construidas que necesiten algún ajuste. Para ello se pueden utilizar diferentes métodos y herramientas de evaluación, según las necesidades del proyecto, estas pueden incluir pruebas de tipo estáticas y dinámicas, según el área de cobertura que se pretenda evaluar.

Algunas prácticas importantes para considerar se resumen en la Tabla 12

Tabla 12. Prácticas de la fase de pruebas de XP

Práctica	Descripción
Pruebas unitarias	Las pruebas unitarias son una forma de comprobar el correcto funcionamiento de los diferentes fragmentos de código, generalmente se aplican en aquellas funciones que realiza algún tipo de lógica concreta y que retorna un resultado. En el contexto de XP, estas pruebas son ejecutadas haciendo uso de la práctica TDD, revisada anteriormente, en la que primero se evalúa que los fragmentos de código cumplan con el objetivo para el que fueron construidas y, si pasan las pruebas, se ejecuta el código.
Detección y corrección de errores	Esta es una actividad transversal sobre las pruebas que se realizan al producto. XP promueve que cada vez que se detecta un error, este se debe corregir de forma inmediata y mantener el ciclo hasta que el error se solucione.

Práctica	Descripción
Pruebas de aceptación	Las pruebas de aceptación son creadas en función de las HU; estas determinan la forma y el objetivo que debe cumplir cada una de las funcionalidades descritas en las HU, para esto se utiliza el criterio de aceptación descrito para determinar si una HU debe darse por terminada, o debe ser corregida.

Una vez finalizadas estas fases y aprobadas las HU comprometidas, se hace el lanzamiento y se produce un incremento de software. Es importante mencionar que estas fases utilizan un ciclo iterativo, por lo cual, al finalizar una iteración, enseguida inicia otra hasta culminar con el desarrollo del producto de software.



Actividades de aprendizaje recomendadas

- Vea el siguiente video y refuerce su conocimiento sobre los principios y prácticas recomendadas de la metodología XP. El título del video es [XP's Values, Principles, and Practices-Georgia Tech-Software Development Process](#). ¿Por qué cree que la comunicación es el valor principal de la metodología?
- Una de las prácticas más comunes de XP es la programación en pares. ¿No quedó claro el concepto? Se lo invita a reforzar la lectura revisando el siguiente video titulado [Agile in Practice: Pair Programming](#) –desde el minuto 0:15 hasta el 2:50– ¿Cómo cree que la programación por pares ayuda al proyecto a entregar funcionalidades de software de gran calidad?

Es hora de medir su conocimiento desarrollando la siguiente autoevaluación:



Autoevaluación 3

Estimado estudiante: mediante este cuestionario, usted pondrá a prueba lo aprendido hasta el momento. En las siguientes preguntas, revise cada ítem y seleccione la respuesta correcta. Recuerde que sus resultados constituyen un reflejo del autoaprendizaje de la asignatura.

1. La programación extrema se enfoca en:
 - a. La adaptabilidad y la sencillez.
 - b. La adaptabilidad y la documentación.
 - c. La sencillez y la programación.

2. La programación extrema se centra en cuatro valores, estos son:
 - a. Comunicación, sencillez, retroalimentación, coraje o valentía, y respeto.
 - b. Simplicidad, sencillez, retroalimentación y respeto.
 - c. La comunicación, simplicidad, retroalimentación, coraje o valentía, y respeto.

3. El valor que se centra en obtener la apreciación y opinión de las funcionalidades del proyecto es:
 - a. Comunicación.
 - b. Retroalimentación.
 - c. Respeto.

Índice

Primer bimestre

Segundo bimestre

Solucionario

Referencias bibliográficas

4. Permitir que los desarrolladores comuniquen alguna dificultad sobre las tareas asignadas corresponde al valor:
 - a. Coraje o valentía.
 - b. Comunicación.
 - c. Retroalimentación.
5. El miembro del equipo externo que apoya para resolver problemas específicos del proyecto es:
 - a. Cliente.
 - b. Consultor.
 - c. Gestor.
6. La refactorización es una actividad que:
 - a. Consiste en reescribir nuevamente el código de algún componente de software.
 - b. Consiste en arreglar los bugs identificados en el código.
 - c. Consiste en describir el problema identificado en el código.
7. El rol que es la parte interesada del proyecto, quien valida y retroalimenta las funcionalidades construidas es:
 - a. Cliente.
 - b. Consultor.
 - c. Gestor
8. Las metáforas son:
 - a. Una descripción sencilla del contexto del proyecto.
 - b. Una descripción amplia del contexto del proyecto.
 - c. Un resumen de la visión del proyecto.

Índice

Primer bimestre

Segundo bimestre

Solucionario

Referencias bibliográficas

9. TDD es:
- a. Práctica de desarrollo de software que consiste en escribir el código de las pruebas funcionales.
 - b. Práctica que consiste en automatizar las pruebas funcionales.
 - c. Práctica de desarrollo de software que consiste en escribir las pruebas unitarias antes que el código.
10. La práctica que promueve la productividad del equipo:
- a. Determina que el equipo no debe superar las 8 horas diarias y 40 semanales de trabajo.
 - b. Determina que el equipo debe superar las 8 horas diarias y 40 semanales de trabajo.
 - c. Implica que el equipo debe estar automotivado para promover la productividad.

[Ir al solucionario](#)



Semana 10

Ha finalizado el estudio de la metodología XP. A continuación, estudiará otra metodología ágil: Scrum. En la industria de software, en la actualidad, es la más utilizada para afrontar proyectos de software por su adaptabilidad, flexibilidad y contextos bien marcados, adaptables a tamaño o complejidad de cualquier proyecto de software. El estudio de esta metodología está planificado para las semanas 10 y 11. En esta primera semana conocerá el marco conceptual de la metodología, y estudiará sus principios, prácticas, roles y los principales artefactos de Scrum. Se lo invita a seguir ampliando sus conocimientos sobre metodologías ágiles.

¡Adelante!

3.2. Scrum

Scrum es una metodología ágil que utiliza un conjunto de buenas prácticas que permite trabajar colaborativamente en equipo para obtener los mejores resultados posibles, mediante la entrega progresiva de valor, para construir un producto o un servicio.

Esta metodología puede ser utilizada en cualquier contexto de la ingeniería; sin embargo, es ampliamente adoptada para gestionar proyectos de desarrollo de software, en los cuales los equipos de desarrollo se enfrentan a un escenario volátil y cambiante en las especificaciones de desarrollo de cualquier proyecto de software.

Primer bimestre

Segundo bimestre

Solucionario

Referencias bibliográficas

A nivel industrial, *Scrum* es la metodología ágil más utilizada a nivel mundial, según el *Décimo cuarto informe anual del estado de Agile*, año 2020⁶. Como se puede observar en la Figura 19, el 58% del total de todos los proyectos de desarrollo de software se abordaron con la metodología *Scrum*; el porcentaje restante se divide en el uso de metodologías como *XP*, *Kanban* y, en mayor proporción, se utilizan metodologías híbridas que se organizan mediante la combinación de diferentes metodologías. Esto es posible gracias a la gran adaptabilidad de las metodologías ágiles.

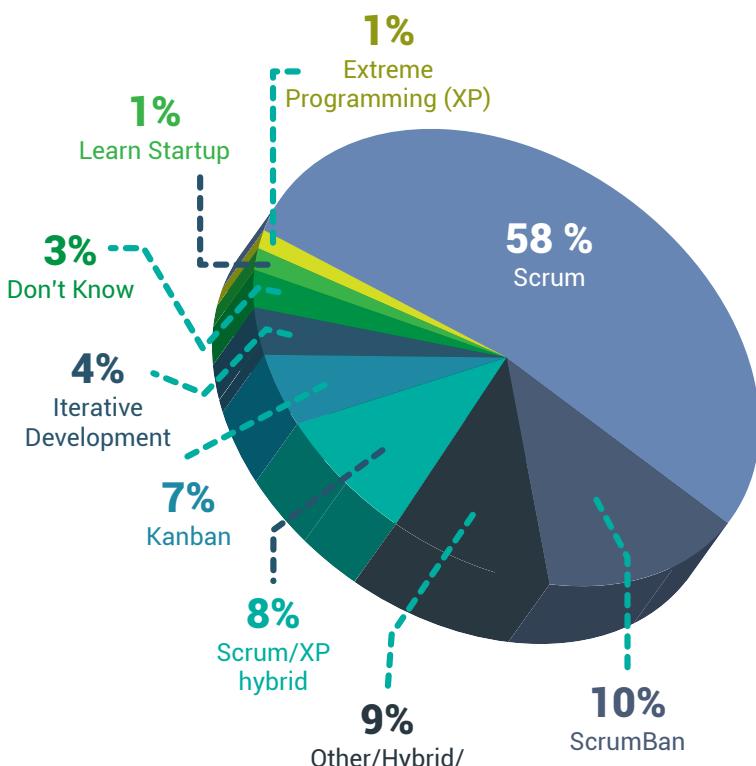


Figura 19. Metodologías ágiles más utilizadas hasta 2020

Nota: Tomada de 14th Annual State of Agile Report (2020)

⁶ stateofagile.com/#ufh-i-615706098-14th-annual-state-of-agile-report/7027494

Índice

Primer bimestre

Segundo bimestre

Solucionario

Referencias bibliográficas

Scrum basa su modelo de trabajo en la combinación de dos modelos estudiados en el primer bimestre, el modelo iterativo y el modelo incremental (estudiados en la sección 2.1), en el que toda la carga de trabajo se descompone en la HU y, posteriormente, en actividades que son priorizadas en función del valor de negocio que se aporta al proyecto. En palabras simples, el proyecto se descompone en diferentes partes que se construyen en función del grado de importancia y aporte al negocio, de esta forma se obtienen resultados rápidos y concretos desde las etapas iniciales del proyecto.

La guía para del cuerpo de conocimiento de *Scrum* (SCRUMstudy™, 2016) conceptualiza a *Scrum* como una metodología de adaptación, iterativa, rápida, flexible y eficaz; diseñada para ofrecer un valor significativo de forma rápida en todo el proyecto (SCRUMstudy™, 2016). *Scrum* se enfoca en centrar los esfuerzos iniciales en actividades fundamentales que agregan valor al cliente en cada de una sus iteraciones, con lo cual los clientes obtienen el más alto valor de sus proyectos, mediante la entrega progresiva de valor.

La entrega progresiva de valor significa que, mediante métodos de priorización, se clasifican las HU en función de su grado de importancia y aporte para el valor del proyecto. La construcción de las HU, mediante *Scrum*, permite que los equipos sean más eficaces, flexibles y tengan la capacidad de escalar a medida que avanza el desarrollo del proyecto.

La metodología se compone de principios, aspectos y procesos. A continuación, revisará cada uno de estos componentes de la metodología.

3.2.1. Principios

Los principios de Scrum son el fundamento sobre el que se basa su marco. Estos principios pueden aplicarse a cualquier tipo de proyecto u organización, y deben respetarse a fin de garantizar la aplicación adecuada de Scrum (SCRUMstudy™, 2016). Los principios no pueden modificarse ni ponerse en discusión, y tienen que ser aplicados y considerados en cada una de las fases de la metodología.

En la Tabla 13, usted podrá revisar, de manera resumida, los principios de la metodología.

Tabla 13. Principios de Scrum

Principio	Descripción
Control empírico	Se basa en que el conocimiento proviene de la experiencia, por tanto, en la metodología se toman decisiones en función de los conocimientos que van transcurriendo en el proyecto. Este principio se basa en tres ideas fundamentales: transparencia, inspección y adaptación
Autoorganización	Se enfoca en el equipo. En Scrum existe apertura para la autoorganización, esto significa que se promueve un gran sentido de compromiso permitiendo que los equipos se organicen entre sí y creando un ambiente innovador y propicio para el crecimiento y la productividad.
Colaboración	Este principio se centra en las tres dimensiones básicas relacionadas con el trabajo colaborativo: el conocimiento, la articulación y la apropiación. Fomenta la gestión de proyectos como un proceso de creación de valor compartido con equipos que trabajan e interactúan conjuntamente para ofrecer el mayor valor posible.
Priorización basada en valor	Se enfoca en que, mediante cada uno de los <i>sprint</i> , el equipo ofrezca el máximo valor de negocio desde el principio hasta la finalización de un proyecto.
Asignación de un bloque de tiempo o <i>time-box</i>	Describe un espacio de tiempo asignado para manejar cada una de las actividades de un proyecto. Scrum Determina un tiempo específico para llevar a cabo <i>sprint</i> y ceremonias o eventos.

Índice

Primer bimestre

Segundo bimestre

Solucionario

Referencias bibliográficas

Principio	Descripción
Desarrollo iterativo	Define el modelo de desarrollo. El modelo iterativo enfatiza en administrar mejor las entregas y los tiempos según las necesidades de los clientes.

Nota: Tomada de Cuerpo de conocimiento de Scrum-SBOK (SCRUMstudy™, 2016)

¿Los principios le parecieron interesantes y fáciles de aplicar?, lo invito a revisar la sección “2. Principios de la Guía SBOK” (2016), y a estudiar cada uno de estos principios a profundidad.

3.2.2. Aspectos

Los aspectos son aquellos elementos que se deben abordar y gestionar mientras dura un proyecto. Según el cuerpo de conocimiento de Scrum, se deben considerar cinco aspectos: 1) organización; 2) justificación del negocio; 3) calidad; 4) cambio, y 5) riesgo. A continuación, estudiará cada uno de estos aspectos:

1. **Organización:** este aspecto se enfoca en describir los roles y responsabilidades definidos para alcanzar una implementación exitosa de la metodología Scrum. Los roles se clasifican en roles centrales, que son los roles fundamentales que se requieren para construir un producto o servicio, están comprometidos con el proyecto y de ellos depende el éxito o fracaso del proyecto; y los roles no centrales, que son roles no fundamentales, en el sentido de que no tienen ninguna obligación formal dentro del equipo de proyecto para ejecutar algún o un grupo de actividades que sirvan para construir un producto o servicio; sin embargo, son importantes en la interacción con los roles centrales, para proporcionar información respecto del proyecto, o para gestionar algún tipo de interés de este.

Índice

Primer bimestre

Segundo bimestre

Solucionario

Referencias bibliográficas

A continuación, se describen cada uno de los roles:

- **Roles centrales:** en Scrum existen tres roles centrales que componen al equipo que se encargará de conducir al éxito cualquier proyecto, estos son el propietario del producto o *product owner*, *Scrum máster* y el equipo de *Scrum*. Estos roles están asociados directamente con la creación del proyecto, están comprometidos y son los responsables de producir los incrementos al final de cada *sprint*. En la Tabla 14 se describen los roles centrales y se destacan sus principales responsabilidades:

Tabla 14. Roles centrales de Scrum

Rol	Descripción	Responsabilidades
<i>Product owner/ Propietario del producto</i>	Es la cara del equipo Scrum frente al cliente, es responsable de gestionar todas las funcionalidades; maximizar el valor entregado y garantizar el éxito del proyecto	<ul style="list-style-type: none">▪ Determinar las funcionalidades del producto en forma de HU;▪ Administrar el <i>backlog</i> del Producto;▪ Administrar y priorizar las HU, y▪ Aprobar o rechazar los incrementos al final de cada <i>sprint</i>;▪ Maximizar el retorno de inversión del proyecto;▪ Justificar la viabilidad del proyecto, y▪ Asegurar los recursos financieros del proyecto.
<i>Scrum máster</i>	Es el facilitador y eliminador de dependencias, asegura que todo el equipo de Scrum tenga un ambiente propicio para trabajar hasta finalizar el proyecto; además guía, facilita y enseña todas las prácticas de la metodología a todos los involucrados del proyecto	<ul style="list-style-type: none">▪ Determinar las funcionalidades del producto en forma de HU.▪ Administrar el <i>backlog</i> del Producto▪ Administrar y priorizar las HU;▪ Aprobar o rechazar los incrementos al final de cada <i>sprint</i>;▪ Maximizar el retorno de inversión del proyecto;▪ Justificar la viabilidad del proyecto, y▪ Asegurar los recursos financieros del proyecto.

Rol	Descripción	Responsabilidades
Equipo de Scrum	Conjunto de profesionales que se encarga de materializar el producto o servicio de software, generalmente se compone de un equipo multidisciplinario con diferentes habilidades y responsabilidades que se encargan de ejecutar las tareas programadas para cada sprint.	<ul style="list-style-type: none"> ▪ Garantizar el conocimiento de múltiples habilidades; ▪ Capacidad de entender las HU y actividades asignadas; ▪ Construir los entregables y producir incrementos funcionales al finalizar cada sprint; ▪ Capacidad para autoorganizarse y motivarse, y ▪ Mantenerse como un equipo unido y ubicado en el mismo lugar.

Nota: Tomada de Cuerpo de conocimiento de Scrum-SBOK (SCRUMstudy™, 2016)

- **Roles no centrales:** los roles no centrales son aquellos que no tienen ninguna responsabilidad formal sobre la creación de los incrementos, estos roles se describen en la Tabla 15.

Tabla 15. Roles no-centrales de Scrum

Rol	Descripción
Socios	Rol colectivo que incluye a clientes, usuarios finales y patrocinadores que interactúan directamente con el equipo principal de Scrum y es el único rol que tiene algún grado de influencia sobre el proyecto.
Cuerpo de asesoramiento de Scrum	Rol opcional que consiste en un conjunto de expertos que generalmente se involucran en la definición de objetivos relacionados con el proyecto
Vendedores	Este rol no tiene influencia directa con el proyecto; representa intereses externos que no forman parte del proyecto.
Jefe de propietarios del producto	Responsable de coordinar el trabajo de múltiples varios propietarios del producto. Aunque no tiene influencia sobre el proyecto, permite dar seguimiento en proyectos más grandes, portafolios o programas.

Rol	Descripción
Jefe de Scrum máster	Responsable de coordinar a todos los Scrum máster. Es conocido como el <i>Scrum de scrums</i> , participa cuando existen varios equipos que trabajan paralelamente en un proyecto.

Nota: Tomada de Cuerpo de conocimiento de Scrum-SBOK (SCRUMstudy™, 2016)

Evidentemente, los roles centrales son fundamentales, ya que son quienes se encargan de influir con el conocimiento técnico para construir un producto o servicio de software; sin embargo, la participación del cliente es fundamental para garantizar una retroalimentación efectiva sobre las necesidades reales del proyecto. Para comprender cómo se produce esta interacción, lo invito revisar la Figura 20.

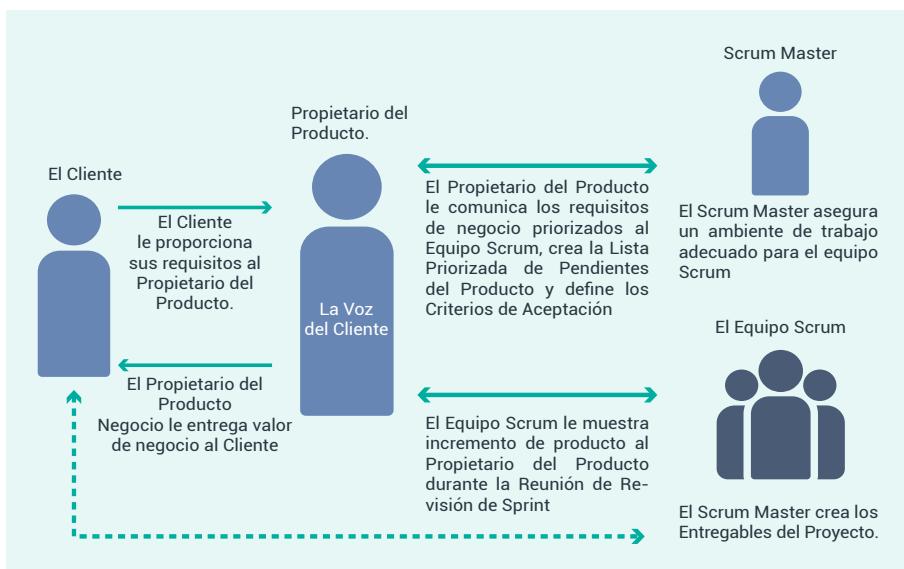


Figura 20. Interacción entre roles de Scrum

Nota: Tomada de Cuerpo de conocimiento de Scrum-SBOK (SCRUMstudy™, 2016)

La figura es muy sencilla de comprender, en ella se identifican de forma muy clara cómo los roles interactúan entre sí para abordar un proyecto. ¿Cree usted que sería muy sencillo para una organización adoptar la organización metodológica de Scrum? ¿Cómo lograría hacerlo? Lo invito a navegar en internet e investigar cómo empresas como Spotify, Facebook, Netflix, o cualquier otra gran empresa, han logrado adoptar esta metodología y cómo se han visto beneficiada su dinámica de trabajo.

2. **Justificación del negocio:** en un proyecto es importante realizar una evaluación apropiada del negocio antes de empezar un proyecto, esto permite que los roles de toma de decisión puedan determinar la viabilidad o factibilidad del proyecto. La justificación del negocio se basa en el principio de entrega impulsada por valor, lo que significa que, independientemente del tamaño o la complejidad del proyecto, mediante estrategias de priorización la metodología, permite entregar resultados que agreguen valor al cliente en las primeras entregas y en el menor tiempo posible; esto posibilita demostrar el valor del proyecto a los socios mediante la verificación de los incrementos funcionales.
3. **Calidad:** asegurar la calidad de los productos y cumplir las expectativas de los clientes es uno de los principales motivos para utilizar una metodología ágil. La retroalimentación continua que ofrecen los clientes permite que el equipo de proyecto, mediante el principio de adaptabilidad y mejora continua, adapte las veces que el cliente crea necesario el producto hasta alcanzar los niveles de calidad acordados. Esto se logra mediante las entregas graduales de incrementos de producto, por lo tanto, los entregables funcionales pueden ser evaluados y ajustados las veces que el cliente y el equipo de Scrum acuerden hasta cumplir con las expectativas del cliente.

Índice

Primer bimestre

Segundo bimestre

Solucionario

Referencias bibliográficas

4. **Cambio:** los proyectos de desarrollo de *software*, por naturaleza, son cambiantes y, en un principio, definen requerimientos volátiles, con poco detalle. Esto sucede porque en un principio es difícil poder definir todos los requisitos del proyecto con gran exactitud, y este es el principal motivo por el cual los proyectos de *software* fallan. En este sentido, Scrum se adapta perfectamente a los proyectos de *software*, ya que al establecer escenarios de trabajo cortos, que resultan en incrementos funcionales de *software*, el cliente puede evaluar los requerimientos descritos en forma de HU, cambiar y agregar si es necesario nuevas características. Este proceso de cambio generalmente pasa mediante el propietario del producto que es quien evalúa la urgencia y la factibilidad del cambio para determinar si los cambios se deben agregar sobre un *sprint* en desarrollo, o se deben programar para *sprint* posteriores.
5. **Riesgo:** el riesgo es un evento o conjunto de eventos que pueden afectar positiva⁷ o negativamente⁸ a un proyecto en caso de materializarse. Scrum, promueve una gestión eficiente del riesgo, esto se logra mediante la identificación de riesgos al inicio de cada uno de los *sprint*, lo que implica que al tener escenarios de trabajo cortos y concretos se puede hacer un seguimiento más efectivo a la materialización de riesgos, de forma que la gestión del riesgo se realice de forma preventiva y proactiva a lo largo de los *sprint*.

Ahora que ha estudiado cómo Scrum aborda estos cinco aspectos fundamentales, ¿Piensa usted que estos aspectos permiten cubrir de una manera más eficiente la gestión de un proyecto de *software* a diferencia de como se lo gestiona en un aspecto tradicional? Lo invito a revisar la Guía SBOK (SCRUMstudy™, 2016), sección 6. Cambio, y a reflexionar **¿Por qué la gestión del cambio** en Scrum es más eficiente y ágil que en un contexto tradicional?

⁷ Riesgo positivo: representa una oportunidad en un proyecto.

⁸ Riesgo negativo: representa una amenaza y podría afectar negativamente a un proyecto.

Índice

Primer bimestre

Segundo bimestre

Solucionario

Referencias bibliográficas

¡Siga adelante con el estudio de los procesos de Scrum!

3.2.3. Procesos

Scrum hace un abordaje metodológico basado en 5 fases y 19 procesos que se deben considerar para gestionar un proyecto. A continuación, en la Tabla 16, se presentan cada uno, y su relación entre las fases y procesos.

Tabla 16. Procesos de Scrum

Fase	Procesos
Inicio	1. Creación de la visión del proyecto. 2. Identificación del <i>Scrum master</i> y los interesados. 3. Formación de los equipos de Scrum. 4. Desarrollo de épicas. 5. Creación de la lista priorizada de pendientes del producto. 6. Realizar la planificación del lanzamiento.
Planificación y estimación	7. Creación de HU. 8. Aprobación, estimación y asignación de las HU. 9. Creación de tareas. 10. Estimación de tareas. 11. Creación de la lista de pendientes del <i>sprint</i> .
Implementación	12. Creación de entregables. 13. Llevar a cabo la reunión diaria. 14. Mantenimiento de la lista priorizada de pendientes del producto.
Revisión y retrospectiva	15. Convocar el <i>Scrum de scrums</i> . 16. Demostración y validación del <i>sprint</i> . 17. Retrospectiva del <i>sprint</i> .
Lanzamiento	18. Envío de entregables. 19. Retrospectiva del proyecto.

Nota: Tomada de Cuerpo de conocimiento de Scrum-SBOK (SCRUMstudy™, 2016)

Índice

Primer bimestre

Segundo bimestre

Solucionario

Referencias bibliográficas

Con respecto al contexto tradicional, estos procesos permiten abordar los proyectos de una forma más ágil y sencilla. Se lo invita a revisar en la Guía SBOK (SCRUMstudy™, 2016) –secciones: 8. Inicio; 9. Planificación y estimación; 10. Implementación; 11. Revisión y retrospectiva, y 12. Lanzamiento— cada una de estas fases y estudiar cómo se aplican cada uno de los procesos revisados. Luego de estudiarlos ¿Cree usted que se deben aplicar todos los procesos de forma estricta? ¿Se puede prescindir de algún proceso? Si su respuesta es positiva, indague por qué y cómo lo podría hacer.

3.2.4. Artefactos

Los artefactos son elementos físicos que se producen como resultado de la metodología. Los tres principales artefactos de Scrum son 1) *product backlog*, 2) *sprint backlog*, y 3) incremento. Revise cada uno de estos artefactos a continuación:

1. ***Product backlog*:** o pila del producto, es el artefacto principal de Scrum, representa un inventario de las HU que componen un producto o servicio. Es la principal fuente de información sobre un producto, permite dar seguimiento a las HU que se tienen que construir y las que están en desarrollo. Este artefacto es responsabilidad directa del propietario del producto; este gestiona, prioriza y organiza las HU en función del valor que representa para el negocio.

Al estar conformado principalmente de HU, permite que los interesados del proyecto tengan una visión general del estado de construcción de un producto, esto debido a que las HU se describen en un lenguaje entendible, natural que el cliente pueda interpretar. Adicionalmente, el *product backlog* puede contener errores encontrados que deben ser corregidos, actividades técnicas como esquemas de configuración para desarrollo de software o herramientas, trabajos de investigación necesarios para el proyecto entre otros.

2. **Sprint backlog:** el *sprint backlog* o pila del *sprint* representa el objetivo que se pretende alcanzar al producir un incremento en la finalización de un *sprint*. Se constituye de una porción del *product backlog*, conformado por las HU comprometidas para construir durante un *sprint*. Los elementos que conforman el *sprint backlog* generalmente se descomponen en tareas más pequeñas, que permiten asignar tareas o actividades, y distribuir el trabajo entre los miembros del equipo de desarrollo; además permite dar seguimiento al avance y medir la evolución diaria durante el *sprint* del incremento.
3. **Incremento:** el incremento es el resultado de un *sprint*, esto se traduce en un componente o parte funcional de un sistema de software que es entregado para ser manipulado por el cliente para validar y verificar el funcionamiento y cumplimiento de sus expectativas respecto del proyecto.



Actividades de aprendizaje recomendadas

- Para reforzar lo estudiado, a continuación, se lo invita a revisar el siguiente video: [Introduction to Scrum-7 Minutes](#), desde el minuto 0:44 hasta el minuto 3:00, en el cual usted podrá comprender la diferencia entre Scrum y el proceso de desarrollo tradicional basado en el modelo en cascada, y responda ¿Cómo definiría usted esta diferencia entre modelos de desarrollo? ¿Cómo se producen los incrementos en Scrum y cascada? Desde el minuto 3:00 a 7:00 usted podrá reforzar otros conceptos estudiados en esta semana.

Índice

Primer bimestre

Segundo bimestre

Solucionario

Referencias bibliográficas

- Los artefactos en *Scrum* son imprescindibles, ya que estos contienen las necesidades en forma de HU de las funcionalidades que deben completar los proyectos. Se lo invita a que revise este video [Artefactos Scrum](#), para que complemente su conocimiento, e investigue ¿Cómo hacer una representación gráfica de avance y seguimiento sobre el trabajo finalizado y pendiente?
- Revise el recurso [Agile Project Management](#), del [OCW Creating Video Games](#). Haga una revisión exploratoria, desde la página 1 a la 10, y conozca una breve historia sobre la gestión de proyectos tradicional y ágil con *Scrum*. Destaque la importancia del **Manifiesto ágil** en el cambio de paradigma de tradicional a ágil.



Semana 11

Después de haber estudiado los conceptos fundamentales de *Scrum*, principios, aspectos, procesos y artefactos, está listo para entrar en el núcleo de la metodología. A continuación, complementará el estudio de la metodología con el estudio del *sprint*, las ceremonias o eventos que se realizan antes, durante y después del *sprint*, y realizará una descripción detallada de los aspectos que suceden en un proyecto de software.

¡Continúe con el estudio de *Scrum*!

3.2.5. Sprint

Partiendo del modelo de proceso que es utilizado en *Scrum* para construir un proyecto de desarrollo de software, el proceso iterativo permite que el proyecto se vaya construyendo en partes durante cada iteración, y el proceso incremental define que esas partes constituyan incrementos graduales hasta completar un producto. Esta interacción se conoce en *Scrum* como un *sprint* y constituye una versión funcional del producto por implementar.

Un *sprint* debe ser programado para períodos cortos de tiempo que abarcan entre 1 y 4 semanas de desarrollo. Determinar la duración de un *sprint*, es un aspecto fundamental para mantener la agilidad de un proyecto de software, por lo tanto, se debe hacer una planificación adecuada antes de comenzar un *sprint* y determinar correctamente la cantidad de puntos de historia⁹ que un equipo es capaz de desarrollar en un determinado tiempo de *sprint*.

Un proyecto puede estar compuesto por varios *sprint*, como se observa en la Figura 21. No existe un número determinado de *sprint* para un proyecto, por lo que se puede adaptar la metodología al contexto del proyecto para el que se esté trabajando. El único aspecto obligatorio es planificar cada *sprint*, con una fecha de inicio y otra de fin, y que esta no supere el rango comprendido entre 1 a 4 semanas.

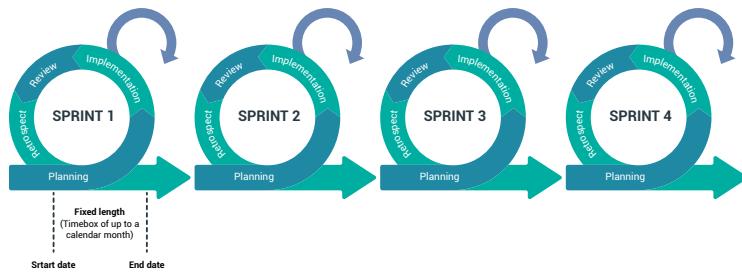


Figura 21. El *sprint*

Nota: Tomada de VisualParadigm (2020)

⁹ Punto de historia o storypoint es la complejidad de una HU.

Índice

Primer bimestre

Segundo bimestre

Solucionario

Referencias bibliográficas

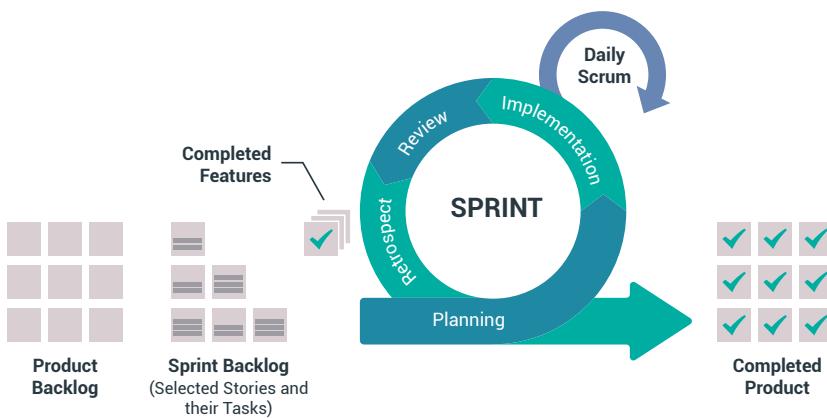


Figura 22. Relación entre el *sprint* y los artefactos de Scrum

Nota: Tomada de VisualParadigm (2020)

Como se explicó en la sección 3.2.4, al iniciar un *sprint* se selecciona las HU que se van a comprometer y estas se descomponen en tareas más pequeñas, como se aprecia en la Figura 23, estas son distribuidas equitativamente entre los miembros del equipo de desarrollo para construir las funcionalidades comprometidas durante un *sprint*.

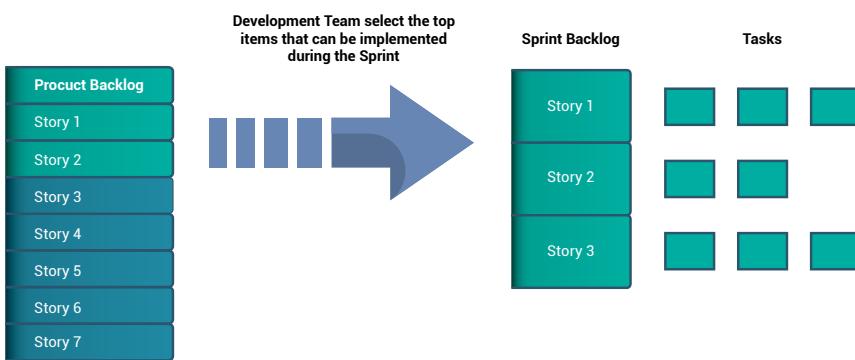


Figura 23. Descomposición del product backlog y sprint backlog

Nota: Tomada de VisualParadigm (2020)

3.2.6. Ceremonias/reuniones

Las ceremonias o reuniones que se llevan a cabo en Scrum son un elemento clave en la metodología, mediante estas se realiza el seguimiento, validación, planificación e identificación de puntos de mejora continua para el equipo. A continuación, en la Tabla 17, revisará cada una de estas ceremonias.

Tabla 17. Ceremonias de Scrum

Ceremonia/reunión	Descripción
<i>Sprint planning</i>	El <i>sprint planning</i> o planificación del <i>sprint</i> , como su nombre lo indica, es la reunión inicial que se realiza al comienzo de cada <i>sprint</i> ; en esta el equipo de Scrum define el objetivo y meta por alcanzar durante el desarrollo del <i>sprint</i> . El propietario del producto presenta el conjunto de HU priorizadas en el <i>backlog</i> del producto y junto al equipo de desarrollo se determina las HU que se van a implementar durante un <i>sprint</i> . La metodología sugiere que esta ceremonia no supere las 8 horas de trabajo continuo.

Ceremonia/reunión	Descripción
Daily Scrum	<i>Daily Scrum</i> , o más conocida como reunión diaria, es una ceremonia que se lleva a cabo diariamente mientras dure el <i>sprint</i> . En esta ceremonia, todos los miembros del equipo se encuentran de pie durante un tiempo no mayor a 15 minutos y cada miembro del equipo responde a tres preguntas concretas ¿Qué hice ayer?, ¿Qué impedimentos tuve al realizar el trabajo?, ¿Qué voy a hacer hoy? Las preguntas deben ser contestadas de forma sencilla y permiten identificar elementos que estén afectando a la productividad del equipo y sincronizar el equipo de acuerdo con la planificación del <i>sprint</i> .
Sprint review	Esta ceremonia se lleva a cabo al finalizar un <i>sprint</i> , en un espacio de tiempo no mayor a cuatro horas. En esta reunión participan los miembros del equipo y los interesados del proyecto. El equipo Scrum presenta los resultados del <i>sprint</i> al propietario del producto e interesados y se explican las HU que se han podido completar y cuáles no; de esta forma se verifica el incremento del producto de software producido durante un <i>sprint</i> y actualizan la lista priorizada de requerimientos según sea necesario.
Sprint retrospective	Esta reunión se desarrolla también al finalizar un <i>sprint</i> , en un tiempo no mayor a cuatro horas. No es obligatoria como la revisión del <i>sprint</i> , solamente tiene que desarrollarse cuando se estime necesario. En esta reunión participa el Scrum máster y los miembros del equipo de Scrum, sirve para intercambiar opiniones respecto del <i>sprint</i> que acaba de finalizar. El objetivo de esta reunión es tratar específicamente los aspectos positivos y negativos del <i>sprint</i> , con el objetivo de identificar buenas prácticas que han ayudado al desarrollo del <i>sprint</i> , o los problemas que han surgido durante el <i>sprint</i> para establecer un proceso de mejora del equipo para el próximo <i>sprint</i> .

Ahora que conoce que es un *sprint*, su relación con los artefactos y las ceremonias o reuniones que se llevan a cabo, mientras se aborda un proyecto de software con Scrum, se lo invita a hacer una pequeña descripción y explicar en qué fase de Scrum se desarrolla cada una de las ceremonias estudiadas en la Tabla 17.

Para comprender y reforzar sus conocimientos, revise el siguiente recurso de video: [Ceremonias de Scrum](#).

Es hora de conocer el flujo de *Scrum*. ¡Avance!

3.2.7. Flujo de *Scrum*

El flujo de *Scrum* integra todos los elementos estudiados hasta el momento; considera y organiza la metodología en función de los procesos definidos en la sección 3.2.3. Parte desde la concepción del proyecto, planificación, ejecución, verificación hasta producir los incrementos de software, como se observa en la Figura 24.

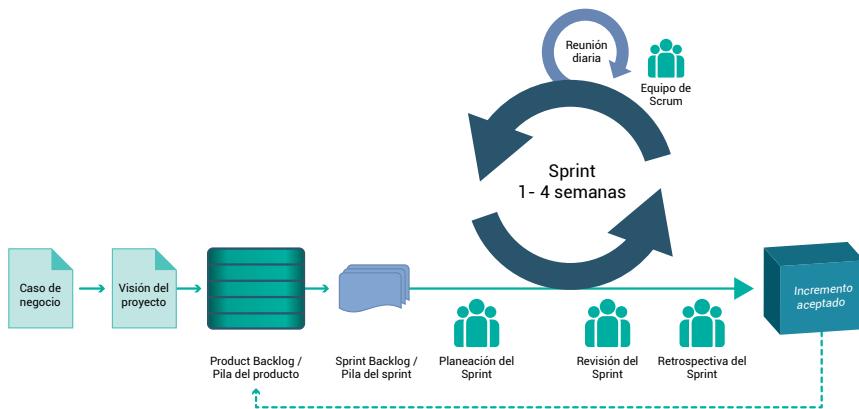


Figura 24. Flujo de *Scrum*

A continuación, se hará una descomposición del flujo general y se describirá qué va pasando en cada una de las fases mientras se va ejecutando el flujo para un proyecto de software.

Fase de inicio: esta es la fase inicial del flujo; parte de especificaciones iniciales entre el propietario del producto y el cliente o interesados del proyecto. En esta fase se exponen las necesidades o motivos por el cual se desea desarrollar el proyecto, para esto se crean las especificaciones formales y de alto nivel. La necesidad

Índice

Primer bimestre

Segundo bimestre

Solucionario

Referencias bibliográficas

se traduce en un caso de negocio; este es un documento formal que expresa la razón por la cual se debe iniciar el proyecto, los antecedentes, objetivos de negocio, resultados de negocio, riesgos y estimaciones generales en tiempo esfuerzo y costo de lo que se pretende desarrollar.

El caso de negocio es el insumo que se requiere para describir la visión formal del proyecto. La visión del proyecto es un documento formal de inicio del proyecto, es responsabilidad del propietario del producto, en esta se explican las necesidades empresariales, objetivos y declaraciones de alto nivel especificadas en el caso de negocio. Es importante mencionar que en durante la creación de la visión del proyecto, no es necesario incluir detalles demasiado específicos sobre el proyecto, se debe dejar espacio para la flexibilidad de que el proyecto se adapte a los cambios o necesidades que surjan durante el proyecto. Es decir, la visión debe centrarse en exponer el problema y no la solución. Por ejemplo, la visión podría describir una descripción corta del problema que se pretende abordar, como la descrita a continuación:

Desarrollar una aplicación móvil que permita a los usuarios buscar y seleccionar productos desde un catálogo digital, comprarlos, realizar el pago en línea y emitir una factura electrónica.

Como se observa, una descripción corta del problema permite describir el problema que se pretende abordar con el proyecto y deja abierta la posibilidad de ajustar los requerimientos o funcionalidades de la aplicación móvil al contexto u objetivos que se plantee el cliente. Una vez finalizada la declaración de la visión, se crea el acta de constitución del proyecto, que autoriza el inicio del proyecto, de manera formal, entre las partes involucradas: el cliente y el propietario del producto.

Índice

Primer bimestre

Segundo bimestre

Solucionario

Referencias bibliográficas

El siguiente paso es determinar el presupuesto referencial del proyecto, este está a cargo del propietario del producto y debe incluir costos de personal involucrado en el proyecto; costos por materiales o recursos, por ejemplo: costos por adquisición de productos de *hardware*, *software*, infraestructura, servicios *cloud*, licencias, entre otros; todo lo que sea necesario para el proyecto, y finalmente, no se puede descartar los costos varios, en estos se consideran aspectos como gastos por oficina, conectividad, viajes, viáticos, servicios básicos, entre otros. Planificar de esta manera brinda las pautas para poder seguir con la identificación de roles clave del proyecto.

Para identificar los roles involucrados en el proyecto, se consideran los roles centrales y no centrales –estudiados en la sección 3.2.2. Aspecto de organización–. En este se formaliza el rol del propietario del producto, Scrum máster y los interesados del proyecto; estos se especifican en una matriz de interesados, en la que se describen los datos generales de los roles del proyecto. Realizado esto, se procede a identificar los recursos humanos y no-humanos del proyecto.

Los recursos humanos se enfocan en la conformación del equipo de Scrum, es decir, los roles que se van a encargar de implementar el proyecto. Este equipo puede estar conformado por analistas de negocio, diseñadores UX/UI, arquitectos de *software*, desarrolladores, *testers*, DevOps, arquitecto *cloud*, o cualquier rol técnico que aporte desde su ámbito al desarrollo del proyecto.

Los recursos no humanos son aquellos elementos o insumos que se utilizan para desarrollar el proyecto; es decir, espacios físicos, de reunión, recursos computacionales, herramientas de *software* y otros elementos que se crean necesarios.

En este punto el propietario del producto, juntamente con el cliente, proceden a desarrollar las épicas¹⁰ del proyecto. Esta actividad parte de los temas generales expuestos en la visión del proyecto que se descomponen en épicas del proyecto y, posteriormente, en HU **más pequeñas**; estos son los elementos que conforman el *backlog* del producto, artefacto principal de Scrum, que es utilizado para realizar más adelante la planificación del *sprint*, como se observa en la primera parte del flujo de Scrum, Figura 25.

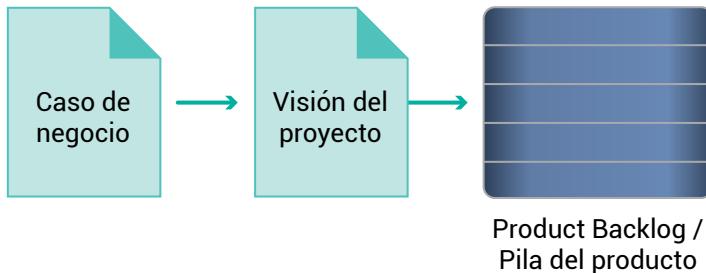


Figura 25. Fase de inicio del flujo de Scrum

Fase de planificación y estimación: en este punto se identifican las HU candidatas descritas en la pila del producto, como se observa en la Figura 26, se discuten con los miembros del equipo de Scrum. Las HU se aprueban, estiman y se asignan a los responsables de construir las funcionalidades. Es importante mencionar que una de las actividades más importantes de esta fase es la planificación y estimación, ya que estimar correctamente influye directamente en el cumplimiento de las metas propuestas durante el *sprint planning*.

¹⁰ Épica: es una HU de alto nivel que describe una funcionalidad de software.

Índice

Primer bimestre

Segundo bimestre

Solucionario

Referencias bibliográficas

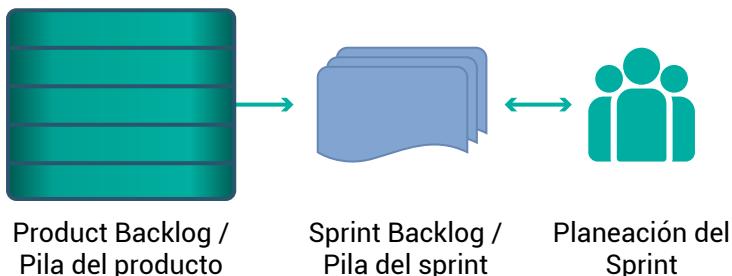


Figura 26. Fase de planeación del flujo de Scrum

Existen algunos métodos de estimación que pueden ser utilizados, estos se describen en la Tabla 18.

Tabla 18. Métodos de estimación de Scrum

Método	Descripción
Wideband Delphi	Consiste en alcanzar un consenso sobre el esfuerzo que demanda construir cada HU candidata para el <i>sprint</i> , para esto se utiliza el juicio de expertos para determinar el esfuerzo de cada HU.
Puño de cinco	Consiste en conformar una votación entre los miembros del equipo, en la que se ponen a consideración las HU, y se realiza una votación en la que los miembros alzan la mano y clasifican el esfuerzo de la historia, del 1 al 5, en función de los dedos de la mano.
Estimación por afinidad	Consiste en estimar definiendo tamaños de esfuerzo de las HU, por ejemplo, grande, pequeño, mediano, entre otros.
Planning poker	Quizás el método más utilizado, basado en Wideband Delphi, consiste en utilizar cartas o herramientas de software que son utilizadas por los miembros del equipo para estimar el esfuerzo o el tamaño de una HU.

Resultado de estas estimaciones, se identifican los puntos de historia; estos permiten identificar el esfuerzo y complejidad de cada HU para poder utilizar este indicador como métrica de HU por comprometer para un *sprint*. Es importante mencionar que el análisis de las HU candidatas se considera la prioridad asignada, en consideración del valor que va a aportar al negocio al finalizar el *sprint*.

Índice

Primer bimestre

Segundo bimestre

Solucionario

Referencias bibliográficas

Por ejemplo, siguiendo la descripción de la visión utilizada como ejemplo en el punto anterior, es más importante comprometer la HU de venta de productos que la HU de registro de productos, ya que, al desarrollar primero la HU de venta de productos, en el primer incremento ya el cliente puede utilizar la aplicación independientemente de cómo se registren los productos, ya que el objetivo del negocio es vender sus productos mediante la plataforma.

De esta forma, se determina las HU que se van a comprometer para cada uno de los *sprint*; posteriormente, estas HU se descomponen en tareas más pequeñas que son asignadas a los miembros del equipo de Scrum. Por ejemplo, para la HU, “**Como** dueño del negocio **deseo** vender productos mediante una aplicación móvil **para** aumentar las ventas y ofrecer un mejor servicio a los clientes”. Las tareas serían las que se presentan en la Tabla 19.

Tabla 19. Ejemplo de descomposición de una historia de usuario

Tareas
Mostrar los productos disponibles.
Seleccionar los productos.
Almacenar los productos seleccionados en un carrito de compras.
Registrar la compra.

La descomposición de la HU permite identificar las tareas que serán distribuidas entre los miembros del equipo de desarrollo. El proceso de descomposición se realiza por cada historia de usuario comprometida y se crea la lista de pendientes priorizada del *sprint* (*backlog* del *sprint*) y se procede a realizar el mismo ejercicio de estimación por cada actividad. En esta ocasión se recomienda que las tareas se estimen en horas de desarrollo, lo que depende del nivel de complejidad que estás tengan. Continuando con el ejemplo anterior, la HU: “**Como** dueño del negocio **deseo** vender productos mediante una aplicación móvil **para** aumentar las ventas y ofrecer un mejor servicio a los clientes”, la estimación debería quedar como se describe en la Tabla 20.

Tabla 20. Ejemplo de estimación en horas de desarrollo de tareas

Tareas	Esfuerzo en horas de desarrollo
Mostrar los productos disponibles	2 horas
Seleccionar los productos	2 horas
Almacenar los productos seleccionados en un carrito de compras	7 horas
Registrar la compra	2 horas

La estimación de ejemplo determina que el esfuerzo en horas para desarrollar la HU, descrita en la Tabla 20, es de 13 horas de desarrollo. Cabe mencionar que esta estimación depende del nivel y habilidades del equipo de desarrollo, puede darse el caso que el equipo esté conformado por desarrolladores *junior*, lo cual implicaría ampliar las horas de esfuerzo para poder cumplir con las tareas.

La estimación y priorización de HU es importante en un proyecto con *Scrum* es un factor muy importante. Gran parte del éxito de los proyectos dependen de que tan bien se estiman las HU. Se lo invita a repasar la visión de ejemplo descrita anteriormente; describir las HU; construir el *product backlog*; identificar, y estimar las HU que usted consideraría para el primer *sprint*. Tome como referencia el siguiente recurso de video: [Scrum. Ejemplo práctico de la planeación del sprint](#). Busque herramientas en internet que le permitan hacer esta gestión y estimación basada en *planning poker*.

Revise el siguiente recurso de video: [Scrum poker. Estimación y planeación del sprint en Scrum](#), desde el minuto 00:40 a 07:30, y comprenda cómo realizar la estimación por el método de *planning poker*.

Fase de implementación: en esta fase se implementan las HU comprometidas. Siguiendo el flujo de *Scrum*, se encontrará en el *sprint* (ver Figura 27). El *sprint* representa el núcleo de la metodología;

Índice

Primer bimestre

Segundo bimestre

Solucionario

Referencias bibliográficas

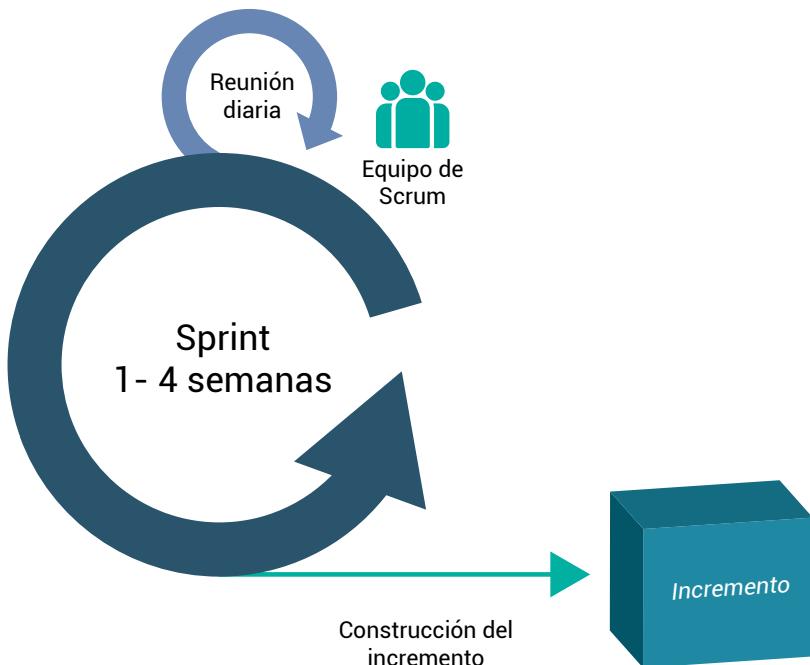


Figura 27. Fase de implementación del flujo de Scrum

Mientras dura esta fase, que puede estar comprometida entre 1 a 4 semanas de la duración del *sprint*, se crean los entregables; es decir, se desarrollan las actividades planificadas hasta construir las funcionalidades del producto de software. En este proceso se van celebrando las ceremonias de Scrum, que permiten dar un seguimiento y retroalimentación continua de las actividades planificadas. El flujo sigue de la siguiente forma:

- El *sprint* inicia con el resultado de la planificación del *sprint*, se especifican en el *sprint backlog*, se organizan y construyen cada una de las HU comprometidas.
- Mientras dura el *sprint*, se realiza el *daily Scrum*, o reunión diaria, mediante esta se da seguimiento a las actividades diarias del equipo y se identifica el avance de trabajo del equipo. Es importante mencionar que en caso de identificarse impedimentos que afecten al normal desarrollo de las tareas asignadas, estas son registradas diariamente en un registro de impedimentos para que el *Scrum* máster se encargue de solucionar los problemas o impedimentos que surjan. Un ejemplo de impedimento puede ser una débil descripción de la tarea asignada, lo cual requerirá que el *Scrum* máster gestione la aclaración de la tarea con el propietario del producto.
- Para dar seguimiento al avance del *sprint*, el *Scrum* máster promueve el uso del *Scrum board*, que es un espacio donde se colocan las tareas en función del estado; lo más común es utilizar un tablero *kanban*, con las tareas pendientes, las que están en proceso y las que están terminadas. El *Scrum board* puede ser gestionado con un tablero físico haciendo uso de alguna herramienta de software como Trello, Jira, entre otros.
- Durante el *sprint* se pueden realizar otras actividades, tales como la identificación de riesgos, actualización del plan de riesgos, actualización de dependencias entre tareas y gestionar solicitudes de cambio, depende de la necesidad de cada una de estas.
- Se actualiza el *sprint backlog* y se hace el refinamiento de este en caso de ser necesario. Es importante destacar que el *backlog* debe permanecer actualizado, sin impedimentos y organizado para que pueda ser utilizado efectivamente por los miembros del equipo.

Índice

Primer bimestre

Segundo bimestre

Solucionario

Referencias bibliográficas

Es importante mencionar que al finalizar un *sprint*, el incremento puede ser aceptado o rechazado en función de la validación que el equipo de Scrum realice con el propietario del producto y los interesados. En caso de existir un incremento, entregable o incluso una HU rechazada, esta se debe mantener en el *backlog* del producto hasta que sea aceptada y sea considerada en los próximos *sprint*.

Con respecto a las solicitudes de cambio durante el *sprint*, el responsable de aprobar o rechazar las solicitudes es el propietario del producto. El cambio únicamente es aprobado y se incorpora al *backlog* del *sprint*, si es que tiene una prioridad muy alta o crea dependencia sobre las HU que ya han sido comprometidas, caso contrario, el cambio se agrega al *backlog* del producto para ser considerado en posteriores *sprint*.

Otras actividades importantes en esta fase corresponden a labores de control de calidad, pruebas de *software*, gestión de defectos, entre otras. Es importante considerar este tipo de actividades de forma que el incremento al finalizar el *sprint* cumpla no solo funcionalmente, sino cuente con todas las certificaciones de calidad y de defectos para ser verificado en la fase de revisión y retrospectiva.

Evidentemente una diferencia radical entre un proyecto ágil y tradicional es la eficiencia y flexibilidad para abordar los constantes cambios que se producen en un proyecto de *software*. Revise la Guía de Scrum (2016) –sección 6. Cambio– y profundice sobre cómo se gestionan eficientemente los cambios en Scrum. Revise el proceso y los factores de decisión para esta actividad.

Fase de revisión y retrospectiva: esta fase corresponde, específicamente con dos ceremonias importantes de Scrum, la revisión del *sprint* y la retrospectiva del *sprint*.

Cuando el incremento está listo y se está por finalizar el *sprint*, se realiza la ceremonia de revisión del *sprint*. En esta se realiza la validación y demostración del *sprint* al propietario del producto y los interesados, estos son los que, en función de los criterios de aceptación descritos en las HU, determinan si los entregables son aceptados o rechazados. Esta reunión se celebra según el siguiente flujo:

- El equipo de desarrollo, junto con el *Scrum máster*, explican el objetivo de la reunión, demuestran el trabajo realizado, solicitan comentarios a los interesados,
- Se discute el trabajo no se ha podido realizar para identificar impedimentos o falencias en las HU y se presente el próximo *sprint*. Con ello se determina si el incremento es aprobado o rechazado.
- Al finalizar la reunión se actualizan los riesgos del proyecto, el cronograma de trabajo y plan de lanzamiento, dependencias y el *Scrum board* para prepararse para el siguiente *sprint*.

Con respecto a la retrospectiva, este proceso se lo lleva a cabo únicamente cuando el equipo de *Scrum* lo considera necesario, en este aspecto, se lleva a cabo la ceremonia de retrospectiva del *sprint*. En esta el equipo hace una revisión de las situaciones que tuvieron que pasar para crear el incremento. En este punto el equipo, liderado por el *Scrum máster*, determina las acciones que funcionaron o buenas prácticas utilizadas, las que no funcionaron, o que el equipo debe mejorar respecto del proceso actual, los impedimentos o problemas y las oportunidades que ha dejado el *sprint*. Esto sirve para mejorar de cada a los siguientes *sprint*.

Índice

Primer bimestre

Segundo bimestre

Solucionario

Referencias bibliográficas

Los resultados de la retrospectiva finalmente permiten que entre los miembros del equipo reafirme los compromisos para crear acuerdos de mejora y lecciones aprendidas respecto de lo que sucedió durante el *sprint*. Una vez finalizadas estas acciones, y si el incremento es aprobado se da por finalizado el *sprint*, y se cierra la iteración para comenzar una nueva hasta finalizar el proyecto.

Para comprender con un ejemplo práctico cómo llevar a cabo la revisión, revise el siguiente recurso de video: [Scrum. Ejemplo práctico revisión del sprint](#).

Para comprender con un ejemplo práctico cómo realizar la retrospectiva, revise el siguiente recurso de video: [Scrum. Ejemplo práctico retrospectiva](#).

Fase de lanzamiento: esta es la fase final de Scrum. En esta se planifica la forma en cómo se va a realizar el envío de entregables. En este punto se consideran todos los acuerdos iniciales alcanzados con el cliente que pueden incluir el producto o servicio de software, documentación, transición de conocimientos, entre otros. Cuando el producto está terminado a satisfacción de las partes, se genera el acuerdo de entregables funcionales; este es un documento cierre formal del negocio y la aprobación formal por parte del cliente o del patrocinador. Según el SBOK (SCRUMstudyTM, 2016), obtener la aceptación formal del cliente es fundamental para el reconocimiento de los ingresos. La responsabilidad de obtener esa aceptación se define en las políticas de la empresa y no es necesariamente la responsabilidad del propietario del producto.



Actividades de aprendizaje recomendadas

- Ha finalizado el estudio de la metodología Scrum. A continuación, lo invito a reforzar sus conocimientos sobre el flujo de Scrum revisando el siguiente video: [Scrum Proceso](#).
- Revise el recurso [Agile Project Management](#), del [OCW Creating Video Games](#). Complemente lo aprendido hasta el momento y refuerce los contenidos abordados sobre los artefactos y la relación entre *el product backlog*, *sprint backlog*, y las tareas del *sprint*. Destaque la importancia de utilizar el *Scrum board*, como instrumento de transparencia en un *sprint*.
- Realice un cuadro comparativo entre las metodologías XP y Scrum; y, según sus conocimientos, reflexione y analice por qué Scrum es la metodología más utilizada en la industria de software. Destaque sus ventajas por sobre otras metodologías e indique si es posible combinar metodologías, es decir, Scrum con XP, Scrum con Kanban, entre otros.

Es hora de medir nuestro conocimiento desarrollando la siguiente autoevaluación:



Autoevaluación 4

Estimado estudiante: mediante este cuestionario usted pondrá a prueba lo aprendido hasta el momento. En las siguientes preguntas, revise cada ítem y seleccione la respuesta correcta. Recuerde que sus resultados constituyen un reflejo del autoaprendizaje de la asignatura.

1. Scrum es una metodología ágil que:
 - a. Utiliza un conjunto de buenas prácticas que permiten trabajar colaborativamente en equipo para obtener los mejores resultados posibles mediante la entrega progresiva de valor.
 - b. Utiliza un conjunto de fases y procesos que se deben seguir para obtener los mejores resultados posibles mediante la entrega progresiva de valor.
 - c. Permite construir productos o servicios de software, de forma ágil y dinámica, que utilizan un conjunto de prácticas para obtener el mayor valor posible.
2. Administrar el backlog del producto es responsabilidad de:
 - a. Scrum máster.
 - b. Equipo Scrum.
 - c. Product owner.

3. Una historia de usuario es:
 - a. Una descripción en lenguaje natural que representan un requisito de software.
 - b. Una descripción en lenguaje técnico que representa un requisito de software.
 - c. Una historia narrada por el usuario que describe una característica de software.
4. Un sprint es:
 - a. Un espacio de tiempo no mayor a cuatro semanas en el que el equipo de Scrum se encarga de implementar las HU comprometidas.
 - b. Un espacio de tiempo mayor a cuatro semanas en el que el equipo de Scrum se encarga de implementar las HU comprometidas.
 - c. Un espacio de tiempo dedicado específicamente a planificar el desarrollo de las HU comprometidas.
5. La asignación de un bloque de tiempo o time-box es:
 - a. Un espacio de tiempo que tiene inicio y no tiene fin para una actividad de Scrum.
 - b. Un espacio de tiempo que tiene un inicio y final para una actividad de Scrum.
 - c. Un espacio de tiempo que se toma para realizar una reunión en Scrum.
6. Los roles centrales de Scrum son:
 - a. Product owner, Scrum máster, equipo de Scrum.
 - b. Cliente, Scrum máster, equipo de Scrum.
 - c. Product owner, cliente, Scrum máster.

Índice

Primer bimestre

Segundo bimestre

Solucionario

Referencias bibliográficas

7. Construir los entregables y producir los incrementos funcionales es responsabilidad del:
 - a. Scrum máster.
 - b. Product owner.
 - c. Equipo de Scrum.
8. El proceso de aprobación, estimación y asignación de HU es un proceso de la fase de:
 - a. Inicio.
 - b. Planificación y estimación.
 - c. Implementación.
9. El incremento es resultado de:
 - a. Sprint.
 - b. Ceremonia.
 - c. Retrospectiva.
10. La ceremonia que no debe durar más allá de 15 minutos es:
 - a. *Daily Scrum*.
 - b. *Sprint review*.
 - c. *Sprint retrospective*.

Ir al solucionario

Índice

Primer bimestre

Segundo bimestre

Solucionario

Referencias bibliográficas

Resultado de aprendizaje 3

Personaliza metodologías para aplicarlas a proyectos específicos.

Contenidos, recursos y actividades de aprendizaje

Hasta el momento ha finalizado con el estudio de las dos metodologías principales del agilismo XP y Scrum. Ser ágil no solo implica gestionar de forma eficiente el proyecto y producir entregables cada vez más rápidos, sino implica apoyar el proceso de desarrollo en escenarios de automatización que mejoren la forma en cómo se integra y entrega el software actualmente. En un mundo cada vez más digital, se requieren de mecanismos que permitan entregar nuevas características de software en ambientes productivos cada vez más transparente que alcance los niveles de calidad adecuados y permitidos para ser usados por usuarios cada vez más exigentes. En esta unidad estudiará la práctica de desarrollo de software DevOps, y revisará la importancia y origen de esta nueva práctica y los principales conceptos y estrategias de esta práctica.



Semana 12



Unidad 4. Automatización del proceso de desarrollo de software

4.1. DevOps

El término **DevOps** fue introducido en 2007-2009 por Patrick Debois, Gene Kim y John Willis, y representa la combinación entre Desarrollo (Dev) y Operaciones (Ops). Para muchos autores, DevOps es la continuación lógica del Agile que comenzó en 2001 (G. Kim et al. , 2016). DevOps es una cultura o un conjunto de prácticas que reducen las barreras entre los desarrolladores que desean innovar y entregar más rápido, y el personal de operaciones, que desean garantizar la estabilidad de los sistemas de producción y la calidad de los sistemas (Krief, 2019).

La comunicación y este vínculo entre Dev y Ops permiten dar un mejor seguimiento de las implementaciones de producción de extremo a extremo y las implementaciones más frecuentes permiten entregar *software* con una mejor calidad, lo que ahorra dinero para la empresa y crea equipos altamente eficientes.

DevOps se basa en tres ejes principales:

- **La cultura de la colaboración:** es la esencia de DevOps. El hecho de que los equipos ya no están separados por la especialización de silos –un equipo de desarrolladores Dev, un equipo de Ops, un equipo de calidad, entre otros–, sino todo lo contrario; estas personas se unen formando equipos multidisciplinarios que tienen el mismo objetivo: entregar valor agregado al producto lo más rápido posible.

- **Procesos:** para lograr un despliegue rápido, estos equipos deben seguir los procesos de desarrollo tomando como base el uso de las metodologías ágiles con fases iterativas que permiten una mejor calidad de funcionalidad y retroalimentación rápida. Es necesario integrar el proceso de desarrollo de ágil, con los flujos de integración entrega y despliegue continuo, para lo cual el proceso de DevOps incluye las siguientes fases.
 - La planificación y priorización de funcionalidades.
 - El desarrollo de *software*.
 - Integración continua y entrega.
 - Despliegue continuo.
 - Operación continua.
 - Monitoreo continuo.

Estas fases están en constante interacción, se desarrollan de forma incremental e iterativa a lo largo de la vida del proyecto.

- **Herramientas:** la elección de herramientas y productos utilizados por los equipos es muy importante, esto permite respaldar los procesos de DevOps. Es importante que se considere el uso de plataformas altamente flexibles y adaptables a las necesidades de cada proyecto.

La Figura 28 ilustra cómo estos tres ejes principales de la cultura DevOps permiten combinar y establecer mecanismos de entrega de valor continuo mediante escenarios automatizados.

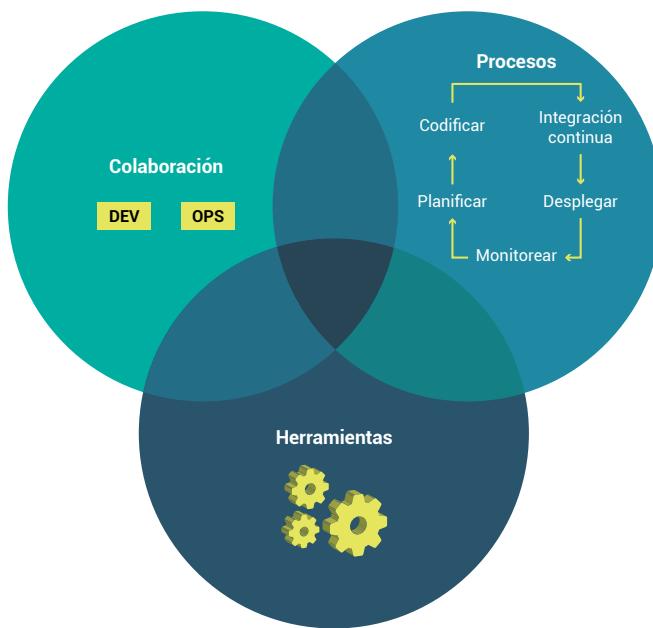


Figura 28. Ejes fundamentales de DevOps

Nota: Adaptado de Krief (2019)

De esta forma, puede observar cómo estos tres ejes permiten integrar a personas, procesos y herramientas que, combinados entre sí, permiten establecer un flujo de entrega de valor continua a los usuarios finales¹¹.

Los beneficios de establecer una cultura DevOps dentro de una empresa son los siguientes:

- Mejor colaboración y comunicación en equipos, lo que tiene un impacto humano y social dentro de la empresa.
- Tiempos de entrega más cortos para la producción, lo que resulta en un mejor rendimiento y satisfacción del usuario final.

¹¹ www.donovanbrown.com/post/what-is-devops

Para Krief (2019), facilitar esta colaboración y mejorar la comunicación entre Dev y Ops considera la implementación de algunos elementos clave como los siguientes:

- Implementación de aplicaciones más frecuentes con integración y entrega continua (CI/CD).
- La implementación y automatización de pruebas unitarias y de integración, con un proceso centrado en el diseño basado en comportamiento (BDD) o diseño basado en prueba (TDD).
- La implementación de un medio de recopilación de comentarios de los usuarios.
- Monitoreo de aplicaciones e infraestructura.

4.1.1. El proceso de DevOps

Como se describió en el punto anterior, DevOps se basa en un ciclo iterativo compuesto por diferentes fases que componen el proceso de DevOps. Como se observa en la Figura 29, DevOps es la integración del equipo de Desarrollo Dev y Operaciones Ops, partiendo de este concepto, el equipo de desarrollo inicia con la planificación, desarrollo o codificación, construcción y pruebas del componente construido. Es importante mencionar que la integración del código desarrollado, construcción y pruebas conforman la integración continua (CI). Una vez cumplido este escenario de desarrollo, se habilita el componente de operaciones, que se encarga de lanzar el producto, desplegar, operar y monitorear; estas actividades componen la entrega y despliegue continuo (CD) del producto.

Índice

Primer bimestre

Segundo bimestre

Solucionario

Referencias bibliográficas

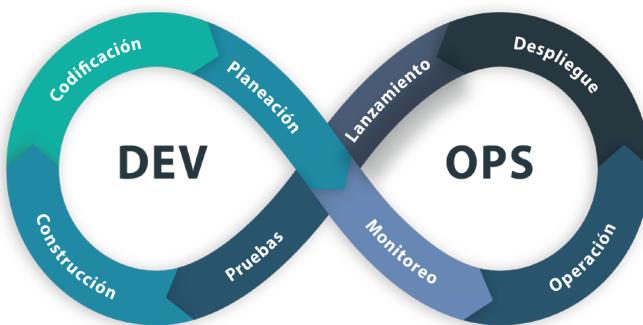


Figura 29. El proceso de DevOps

Nota: Tomada de sesitdigital.com/arquitectura-de-software-y-cultura-devops/

El ciclo DevOps es infinito y solo termina cuando el equipo lo considera. Cabe destacar que el ciclo no solo puede ser utilizado para procesos de desarrollo que recién empiezan, sino que puede ser utilizado por una organización para mantener, gestionar y automatizar la entrega de nuevas características de software mientras el ciclo de vida del producto esté activo.

De acuerdo con Kelly (2019), las fases clave involucradas en DevOps se resumen en la Tabla 21.

Tabla 21. Fases del proceso de DevOps

Fase	Descripción
Planificación	<p>Etapa en la que los equipos de desarrollo y operaciones de TI, junto con otras partes interesadas, determinan el conjunto de características deseadas, acompañado de un valor de iteración y criterios para cada fase del proyecto. Los desarrolladores, administradores de sistemas, gestores de productos, el personal de <i>marketing</i> y los escritores técnicos necesitan un asiento en la mesa para colaborar en el plan de desarrollo.</p> <p>Los planes del proyecto generalmente se describen en repositorios centrales, tales como Atlassian Jira o Confluence, y cada miembro del equipo debe tener acceso a los planes del proyecto en cualquier momento, desde cualquier lugar.</p>

[Índice](#)[Primer bimestre](#)[Segundo bimestre](#)[Solucionario](#)[Referencias bibliográficas](#)

Fase	Descripción
Codificar y construir	<p>Algunos equipos tratan el código y la compilación como fases separadas. Los equipos generalmente definen si integrar o tratar estas actividades como fases independientes, esto depende mucho del equipo y naturaleza de los proyectos.</p> <p>En la fase de código, los desarrolladores realizan su trabajo y cuando completan sus tareas e integran su trabajo en un repositorio de código fuente, tales como GitLab o GitHub.</p> <p>La fase de construcción implica la recuperación de código de software del repositorio centralizado mediante una herramienta automatizada, como Chef o Puppet o un servidor de integración como Jenkins o Travis CI. Estas herramientas de automatización compilan el código del software generan un artefacto binario que puede ser desplegado o utilizado para ejecutar pruebas funcionales.</p>
Integración continua	La integración continua es una práctica mediante la cual los equipos de desarrollo fusionan el código en un repositorio central. En esta fase los equipos automatizan tareas como revisiones de código, validación y pruebas.
Pruebas	<p>Las pruebas de DevOps generalmente se automatizan mediante el uso de herramientas especializadas, esto no quiere decir que se prescinde del personal de calidad. La automatización permite a los desarrolladores realizar pruebas continuas mediante herramientas como Selenium o JUnit para probar múltiples bases de código en paralelo.</p> <p>Las pruebas automatizadas también producen informes detallados sobre la base de código. Las partes interesadas de una organización pueden usar estos informes para obtener información sobre el ciclo de desarrollo y la madurez del producto.</p>
Despliegue continuo	Con la implementación continua, cada cambio de código pasa por toda la secuencia y entra en ambientes de prueba o productivos de forma automática. Una organización puede programar tantas implementaciones por día como necesite, en función de los requisitos y la velocidad de su equipo.



Índice

Primer bimestre

Segundo bimestre

Solucionario

Referencias bibliográficas

Fase	Descripción
Operaciones	En esta fase del proceso de DevOps, los administradores de TI administran el software en ambientes productivos. Herramientas como Ansible, Puppet, PowerShell, Chef, Salt y Otter proporcionan las capacidades necesarias de gestión y recopilación de datos, así como vistas operativas de las aplicaciones de producción. Cabe mencionar que incluso previo a la operación, se pueden automatizar la creación de las infraestructuras necesarias para desplegar las aplicaciones, mediante prácticas de infraestructura como código IaC.
Monitoreo continuo	Los equipos de desarrollo y operaciones deben monitorear continuamente sus aplicaciones en ambientes productivos. Las herramientas de monitoreo populares para esta fase incluyen New Relic, Datadog, Grafana, Wireshark, Splunk y Nagios.

Nota: Adaptado de Kelly (2019)

Hasta el momento conoce el componente teórico de DevOps. Empezará a ver cómo los tres ejes fundamentales empiezan a interactuar entre sí. Ha visto como los proceso de DevOps se fusionan y cómo se empiezan a relacionar los equipos de Dev y Ops mediante el uso de herramientas que soportan las actividades de automatización de estos escenarios. Para reforzar el conocimiento adquirido hasta el momento, realice las actividades recomendadas.



Actividades de aprendizaje recomendadas

Revise el siguiente video: [What is DevOps? | Introduction To DevOps | Devops For Beginners | DevOps Tutorial | Simplilearn](#). Reflexione sobre la importancia de DevOps en las grandes industrias y responda a las siguientes preguntas ¿Por qué DevOps es tan importante para las grandes empresas tecnológicas? ¿Con qué frecuencia las grandes

empresas como Facebook, Google y AWS están realizando nuevos despliegues? ¿Cómo DevOps apoya a los procesos de transformación digital? ¿Se podría aplicar la cultura DevOps en otros procesos de ingeniería?



Semana 13

Hasta el momento ha visto la parte conceptual de la cultura DevOps. Seguramente, resultado de las preguntas planteadas en la actividad recomendada de la semana 12, le causó grandes impresiones y ha visto como DevOps realmente ha impulsado la cantidad de entregas frecuentes de software que realizan las grandes compañías tecnológicas. A continuación, revisará cada una de las prácticas continuas de DevOps, y estudiará cómo estas complementan la entrega de valor al usuario final de una forma ágil y rápida. Además, conocerá cómo los equipos de DevOps mejoran considerablemente sus prácticas mediante el uso de herramientas y mecanismos propios de DevOps.

4.1.2. Integración continua (*Continuous Integration [CI]*)

Martin Fowler define a la integración continua de la siguiente manera:

La integración continua es una práctica de desarrollo de software en la que los miembros de un equipo integran su trabajo con frecuencia [...] Cada integración se verifica mediante una compilación automatizada (incluida una prueba) para detectar errores de integración lo más rápido posible.
(2006, s.p.)

Índice

Primer bimestre

Segundo bimestre

Solucionario

Referencias bibliográficas

Si revisa esa definición, puede identificar tres elementos clave: miembros de un equipo, integrarse y lo más rápido posible; es decir, integración continua es un proceso automático que permite al equipo de desarrollo contar con un mecanismo que se encarga de integrar el código que generan los desarrolladores; realizar verificaciones o pruebas automáticas, y compilar el código, siempre y cuando este pase las pruebas. Todo este proceso es automático cada vez que un desarrollador integra una nueva característica, de esta manera se permite que las entregas sean rápidas y así se integren nuevas funcionalidades.

Como puede observar, este proceso requiere de un espíritu de equipo de colaboración y comunicación, porque la ejecución de CI impacta a todos los miembros en términos de metodología de trabajo y, por lo tanto, de colaboración; además, CI requiere la implementación de procesos (ramificación, confirmación, solicitud de extracción, revisión de código, entre otros) con automatización que se realiza con herramientas adaptadas a todo el equipo (Git, Jenkins, Azure, DevOps, entre otros). Y, por último, CI debe ejecutarse rápidamente para recopilar comentarios sobre la integración del código lo antes posible y, por lo tanto, poder ofrecer nuevas funciones más rápidamente a los usuarios.

Para configurar CI, por lo tanto, es necesario tener un administrador de código fuente (SCM, por sus siglas en inglés) que permita la centralización del código de todos los miembros del equipo. Este administrador de código puede ser de cualquier tipo, por ejemplo, Git¹², SVN¹³, Team Foundation Source Control (TFVC)¹⁴, AWS CodeCommit¹⁵ o cualquier otra herramienta de gestión de código fuente. También es importante contar con un gestor de integración

¹² git-scm.com/

¹³ subversion.apache.org/

¹⁴ docs.microsoft.com/en-us/azure/devops/repos/tfvc/?view=azure-devops

¹⁵ aws.amazon.com/es/codecommit/

Índice

Primer bimestre

Segundo bimestre

Solucionario

Referencias bibliográficas

continua, comúnmente esto se logra implementando un servidor de CI que gestione todo el proceso de integración continua como Jenkins, GitLab CI, TeamCity, Azure Pipelines, GitHub Actions, Travis CI, Circle CI, entre otros.

En este contexto, cada miembro del equipo trabajará en el código de la aplicación diariamente, de forma iterativa e incremental –como en los métodos ágiles y *Scrum*–. Cada tarea o característica se divide y gestiona mediante el uso de ramas, por medio de los flujos de trabajo definidos por cada repositorio de código fuente, como por ejemplo, el flujo de trabajo de BitBucket¹⁶. Regularmente, incluso varias veces al día, los miembros archivan o confirman su código y preferiblemente con pequeñas confirmaciones que pueden repararse fácilmente en caso de error. Esto, por lo tanto, se integrará en el resto del código de la aplicación con todas las otras asignaciones de los miembros del equipo de desarrollo.

Esta integración de todos los *commits* o confirmaciones es el punto de partida del proceso de CI. Este proceso es ejecutado por un servidor CI que automatiza y se activa en cada confirmación. Este mecanismo parte por recuperar el código del repositorio de fuentes; se empaqueta; se ejecutan las pruebas unitarias; se compila, y se despliega la aplicación. Es importante mencionar que este es un mecanismo básico, ya que el servidor de integración permite configurar todos los pasos que el equipo crea necesario antes, durante o después del proceso de compilación y despliegue de aplicaciones.

Cabe destacar que mientras se van ejecutando estos procesos automáticos. El proceso de integración siempre entra en un proceso de mejora continua; es decir, se va optimizando y adaptando conforme el proyecto lo requiera, de forma que el proceso de integración permita que los desarrolladores se vean retroalimentados

¹⁶ www.atlassian.com/es/git/tutorials/comparing-workflows/gitflow-workflow

mediante comentarios, notificaciones o mecanismos que posean las herramientas utilizadas para conocer el estado de su integración, como por ejemplo, puede suceder el caso que el código no se compile por alguna falla que el código presente, o que no pase alguna prueba unitaria; en este caso, el mismo proceso de integración alertará a los desarrolladores sobre estos problemas.

Con un proceso de CI optimizado y completo, el desarrollador puede solucionar rápidamente su problema y mejorar su código, o discutirlo con el resto del equipo, y confirmar su código para una nueva integración (ver Figura 30).

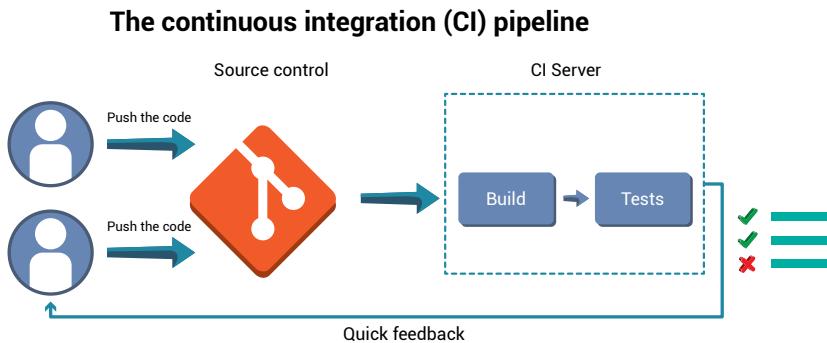


Figura 30. El proceso de integración continua

Nota: Tomada de Krief (2019)

La figura muestra los pasos cíclicos de la integración continua. Los miembros del equipo introducen el código en el SCM y la ejecución de la compilación y prueba la ejecuta el servidor de CI. Y el propósito de este proceso rápido es proporcionar comentarios rápidos a los miembros.

Por ejemplo, un escenario de integración continua mínimo debería contar con un repositorio de código y un servidor de integración para generar un artefacto, como se observa en la Figura 31.



Figura 31. Ejemplo de integración continua

Siguiendo este esquema de ejemplo, es importante mencionar que el integrador debe ser configurado y contar con todos los detalles necesarios para que se ejecuten las pruebas unitarias; se notifiquen los resultados de las pruebas, y se cree el artefacto. Para configurar el escenario de integración continua expuesto en la Figura 31, a continuación se muestra el *script de pipeline* que se debe configurar en una herramienta como Jenkins para integrar, de forma automática, una aplicación escrita en un contexto de desarrollo basado en Java y Maven. Para probar este *script*, siga la [Guía de Jenkins](#).

```
pipeline {
    agent any
    tools {
        maven 'Maven 3.3.9'
        jdk   'jdk8'
    }
    stages {
        stage( ' Clonar el código ' ) {
            steps {
                sh 'git clone URL_REPO'
            }
        }
        stage (' Pruebas unitarias ') {
            steps {
                sh 'mvn test'
            }
        }
        stage ('Construcción') {
            steps {
                sh 'mvn clean package'
            }
        }
    }
}
```

Índice

Primer bimestre

Segundo bimestre

Solucionario

Referencias bibliográficas

Generalmente se utilizan *scripts*, o se configuran pasos mediante los entornos gráficos que proveen las herramientas¹⁷. A través de *scripts* se definen los pasos que el servidor debe ejecutar para integrar y construir los artefactos. Como se observa en el *script* anterior, se automatiza la ejecución de tres pasos: la obtención del código desde el repositorio; la ejecución de las pruebas unitarias, y el empaquetamiento y construcción del artefacto. En caso de no pasar cada una de las fases propuestas, el proceso se detiene y se retroalimenta, en forma automática, a los miembros del equipo para que estos realicen las refactorizaciones necesarias.

4.1.3. Entrega continua (*Continuous Delivery [CD]*)

La entrega continua es una práctica de desarrollo de *software* mediante el cual se prepara los cambios en el código y se entregan en un ambiente de producción. Esta práctica permite ampliar las posibilidades del proceso de CI, y se activa luego de que el proceso de compilación ha sido satisfactorio; esto generará un artefacto listo para su implementación o pruebas, según sea el objetivo del proceso de entrega continua.

El CD a menudo comienza con un paquete de aplicación preparado por CI, que se instalará de acuerdo con una lista de tareas automatizadas. Estas tareas pueden ser de cualquier tipo: descomprimir, detener y reiniciar el servicio, copiar archivos, reemplazar la configuración, entre otros. La ejecución de pruebas funcionales y de aceptación también se puede realizar durante el proceso de CD.

A diferencia de CI, CD tiene como objetivo probar toda la aplicación con todas sus dependencias. Esto es muy visible en aplicaciones de microservicios compuestas de varios servicios y API. CI solo probará el microservicio en desarrollo mientras que, una vez implementado en un entorno provisional, será posible probar y validar toda la aplicación, así como las API y microservicios de los que

¹⁷ www.ionos.es/digitalguide/paginas-web/desarrollo-web/jenkins-tutorial/

Índice

Primer bimestre

Segundo bimestre

Solucionario

Referencias bibliográficas

está compuesta. En la práctica, hoy en día, es muy común vincular CI con CD en un entorno de integración; es decir, CI se implementa al mismo tiempo en un entorno. De hecho, es necesario para que los desarrolladores puedan tener en cada confirmación no solo la ejecución de pruebas unitarias, sino también una verificación de la aplicación en su conjunto (UI y funcional) con la integración de los desarrollos de los otros miembros del equipo.

Es muy importante que el paquete generado durante CI y que se implementará durante el CD sea el mismo que se instalará en todos los entornos, y este debería ser el caso hasta la producción. Sin embargo, puede haber transformaciones del archivo de configuración que difieren según el entorno, pero el código de la aplicación (binarios, DLL y JAR) debe permanecer sin cambios.

Este carácter inmutable del código es la única garantía de que la aplicación verificada en un entorno tendrá la misma calidad que la versión implementada en el entorno anterior y la misma que se implementará en el entorno siguiente. Si se van a realizar cambios en el código después de la verificación en uno de los entornos, una vez hechos, la modificación tendrá que pasar por el ciclo de CI y CD nuevamente.

Las herramientas configuradas para CI/CD a menudo se completan con otras soluciones que son las siguientes:

- **Un administrador de paquetes:** constituye el espacio de almacenamiento de los paquetes generados por CI y recuperados por CD. Estos administradores deben admitir feeds, versiones y diferentes tipos de paquetes. Hay varios en el mercado, tales como Nexus, ProGet, Artifactory y Azure Artifacts.
- **Un administrador de configuración:** administra los cambios de configuración durante el CD. La mayoría de las herramientas de CD incluyen un mecanismo de configuración con un sistema de variables.

- En CD, la implementación de la aplicación en cada entorno de ensayo se activa de la siguiente manera:
 - Se puede activar automáticamente después de una ejecución exitosa en un entorno anterior. Por ejemplo, puede imaginar un caso en el que la implementación en el entorno de preproducción se active automáticamente cuando las pruebas de integración se hayan realizado con éxito en un entorno dedicado.
 - Se puede activar manualmente, para entornos sensibles como el entorno de producción, después de una aprobación manual por parte de una persona responsable de validar el correcto funcionamiento de la aplicación en un entorno.

Lo importante en un proceso de CD es que la implementación en el entorno de producción, es decir, para el usuario final, sea habilitada de forma manual; es decir, cuando se han hecho todas las validaciones y verificaciones, recién allí se pueda aprobar un paso a producción, como se observa en la Figura 32.

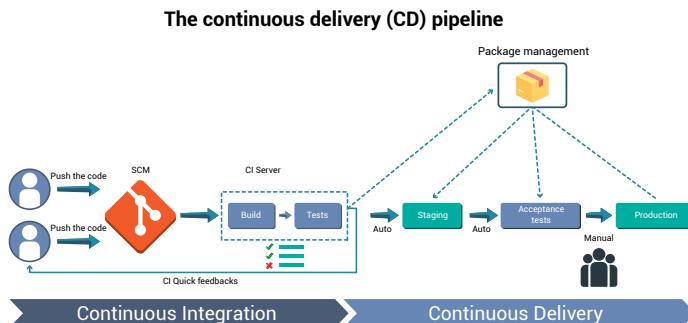


Figura 32. Proceso de entrega continua
Nota: Tomada de Krief (2019)

Esta figura muestra claramente que el proceso de CD es una continuación del proceso de CI. Representa la cadena de pasos de CD, que son automáticos para entornos de preparación, pero manuales para implementaciones de producción. También muestra

Índice

Primer bimestre

Segundo bimestre

Solucionario

Referencias bibliográficas

que el paquete es generado por CI y se almacena en un administrador de paquetes y que es el mismo paquete que se implementa en diferentes entornos.

Retomando el ejemplo descrito en la sección de CI, el artefacto generado si es compilado satisfactoriamente debe ser desplegado en un ambiente de producción que puede ser en un escenario compuesto de infraestructura física, *cloud* o en un contexto de contenedores, depende de la naturaleza del proyecto. Se lo invita a completar el *script* de ejemplo descrito en el punto de CI. ¿Qué etapa agregaría al *script* para ejecutar el artefacto JAR generado? A continuación, se describe el nuevo paso por implementar para desplegar el artefacto generado. Recuerde seguir los pasos de la [documentación oficial de Jenkins](#).

```
pipeline {
    agent any
    tools {
        maven 'Maven 3.3.9'
        jdk 'jdk8'
    }
    stages {
        stage( 'Clonar el código' ) {
            steps {
                sh 'git clone URL_REPOITORIO'
            }
        }
        stage ('Pruebas unitarias') {
            steps {
                sh 'mvn test'
            }
        }
        stage ('Construcción') {
            steps {
                sh 'mvn clean package'
            }
        }
        stage ('Despliegue') {
            steps {
                sh 'java -jar ARTEFACTO'
            }
        }
    }
}
```

Índice

Primer bimestre

Segundo bimestre

Solucionario

Referencias bibliográficas

Como puede observar, en un ejemplo sencillo se crea otra etapa en el proceso de CI/CD y se complementa el *script*, con el comando adecuado para realizar un despliegue automático según sea necesario. Cabe recalcar que las configuraciones dependen inicialmente del contexto tecnológico que se esté empleando para construir o mantener un proyecto con estrategias DevOps.

4.1.4. Despliegue continuo

Como lo vio en el punto anterior, la implementación continua es una extensión de CD, pero esta vez con un proceso que automatiza toda la canalización de CI/CD, desde el momento en que el desarrollador compromete su código para la implementación en producción mediante todos los pasos de verificación.

Esta práctica rara vez se implementa en las empresas, porque requiere una amplia cobertura de pruebas (unidad, funcional, integración, rendimiento, entre otros). Para la aplicación y la ejecución exitosa de estas pruebas es suficiente para validar el correcto funcionamiento de la aplicación con todas estas dependencias, pero también es posible la implementación automatizada en un entorno de producción sin ninguna acción de aprobación.

El proceso de despliegue continuo también debe tener en cuenta todos los pasos para restaurar la aplicación en caso de un problema de producción. El despliegue continuo se puede implementar con el uso y la implementación de técnicas de alternancia de características (o indicadores de características), lo que implica encapsular las funcionalidades de la aplicación en características y activar sus características bajo demanda, directamente en producción, sin tener que volver a implementar el código de la aplicación.

Otra técnica, como se observa en la Figura 33, es utilizar dos ambientes, siguiendo la figura, uno azul y otro verde. Primero desplegamos en el entorno azul, luego en el verde; esto asegurará que no se requiera tiempo de inactividad y que se tenga certeza sobre el estado y control sobre los eventos que generan las aplicaciones.

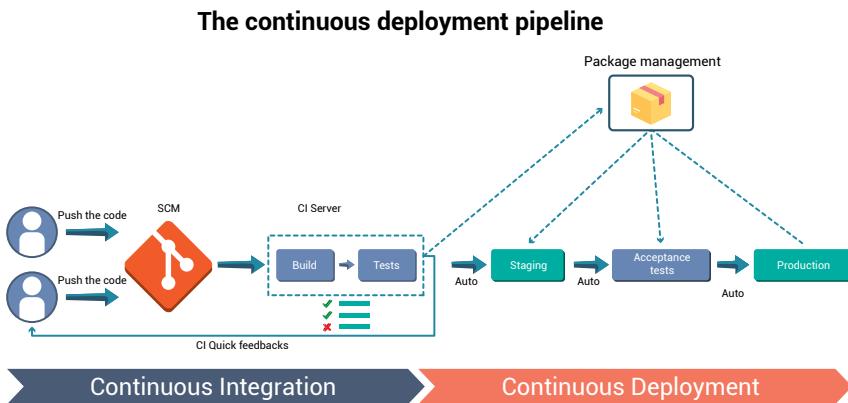


Figura 33. Nota: Tomada de Krief (2019)

Como se puede apreciar, el esquema es similar al de CD, pero con la diferencia de que representa la implementación automatizada de extremo a extremo. Los procesos de CI/CD son, por lo tanto, una parte esencial de la cultura de DevOps, ya que CI permite a los equipos integrar y probar la coherencia de su código, y obtener retroalimentación rápida muy regularmente. El CD se implementa automáticamente en uno o más entornos de preparación y, por lo tanto, ofrece la posibilidad de probar toda la aplicación hasta que se implemente en producción. Finalmente, la implementación continua automatiza la implementación de la aplicación desde el compromiso con el entorno de producción.

¿Ha comprendido los procesos de integración, entrega y despliegue?
Realice un cuadro comparativo e identifique las diferencias entre estos elementos de DevOps.

¿El aprendizaje se está poniendo cada vez más interesante? No se podría complementar el equipo de operaciones con el de desarrollo sin considerar a la infraestructura como un elemento importante para hacer el proceso de integración y entrega de forma automática mediante herramientas especializadas. Para conseguir eso, requiere estudiar el paradigma de infraestructura como código y revisar

cómo puede automatizar el aprovisionamiento de infraestructura de forma automática. A continuación, estudiará cómo aprovisionar infraestructura como código y su relación e importancia en un contexto de DevOps.

4.1.5. Infraestructura como código (IaC)

IaC es una práctica que consiste en escribir el código de los recursos que conforman una infraestructura. Esta práctica comenzó a surtir efecto con el auge de la cultura DevOps y la modernización de la infraestructura de la nube. En un contexto OnPremise, los equipos de Ops que implementan infraestructuras manualmente y esto toma más tiempo para entregar cambios en la infraestructura debido a factores como adquisiciones, configuración, manejo inconsistente o el riesgo de errores y defectos. Además, con la modernización de la nube y su escalabilidad, la forma en que se construye una infraestructura requiere una revisión de las prácticas de aprovisionamiento y cambio mediante la adaptación de un método más automatizado.

IaC es el proceso de escribir en código los recursos que se requieren para aprovisionamiento y configuración de componentes de infraestructura para automatizar su implementación de manera repetible y consistente. A continuación, se destacan los beneficios de IaC:

- La estandarización de la configuración de la infraestructura reduce el riesgo de error. El código que describe la infraestructura se versiona y controla en un administrador de código fuente.
- El código está integrado en las canalizaciones de CI/CD.
- Las implementaciones que realizan cambios en la infraestructura son más rápidas y eficientes.

- Hay una mejor administración, control y una reducción en los costos de infraestructura.

IaC también brinda beneficios a un equipo de DevOps al permitir que Ops sea más eficiente en tareas de mejora de infraestructura, en lugar de perder tiempo en la configuración manual y al darle a Dev la posibilidad de actualizar sus infraestructuras y realizar cambios sin tener que pedir más recursos de Ops. También permite la creación de entornos efímeros y de autoservicio que brindarán a los desarrolladores y evaluadores más flexibilidad para probar nuevas características, de forma aislada e independiente, de otros ambientes.

Las herramientas más utilizadas para IaC son Puppet, Ansible, Chef, Terraform o Plataformas y, como servicio en la nube, AWS CloudFormation, Deployment Manager de Google Console, entre otras. El proceso inicia con escribir plantillas de código, o más conocidas como *templates*, en formatos estructurados y legibles –en formato JSON, YAML, YML, entre otros– y ejecutarlos mediante las herramientas antes mencionadas para que se aprovisione la infraestructura de forma automática. ¿Cómo es posible esto? A continuación, en un ejemplo, se describe cómo conseguir esto. Un caso común es aprovisionar un servidor o instancia en nube para desplegar las aplicaciones, a continuación, se ejemplifica una plantilla para aprovisionar infraestructura con AWS CloudFormation.

Índice

Primer bimestre

Segundo bimestre

Solucionario

Referencias bibliográficas

```
AWS::TemplateFormatVersion: "2010-09-09"
Description : Template para aprovisionar una Instancia EC2

Resources:
  MyServidorEC2:
    Type: "AWS ::EC2 :: Instance"
    Properties :
      ImageId: "ami - 09d95fab7fff3776c"
      InstanceType : t2. micro
      KeyName: jenkins - server
      SecurityGroupIds :
        -!Ref myFirewallDemo
    Tags:
      - Key: Name
        Value: Servidor para aplicaciones

  myFirewall:
    Type: AWS :: EC2 :: SecurityGroup
    Properties :
      GroupDescription : Permite conexion ssh desde internet
      SecurityGroupIngress:
        - IpProtocol : tcp
          FromPort: 22
          ToPort: 22
          CidrIp: 0.0.0.0/0
    Tags:
      - Key: Name
        Value: sshDemo
  VpcId: vpc - 22c1b946
```

De esta manera se puede aprovisionar infraestructuras haciendo uso de prácticas IaC.

¿Cree usted que lo podría lograr utilizando herramientas como Terraform, Ansible o cualquiera de su preferencia? Revise la documentación oficial:

- Terraform:
learn.hashicorp.com/collections/terraform/aws-get-started
- Ansible:
docs.ansible.com/ansible/latest/scenario_guides/guide_aws.html



Actividades de aprendizaje recomendadas

- Revise el siguiente artículo "[Automated CI/CD with Jenkins](#)", y vea el siguiente video [Building Docker Images using Jenkins step by step | Devops Integration Live Demo | JavaTechie](#), en su totalidad. Analice cómo implementar una canalización CI/CD haciendo uso de Jenkins, como servidor de integración, y despliegue una aplicación sencilla de forma automática. Siga los pasos que se mencionan en el artículo.
- Navegue en internet y busque la tabla periódica de DevOps; revise las múltiples herramientas que podría utilizada para cada fase de DevOps, y destaque las más importantes y utilizadas.



Semana 14

Ha llegado a la última semana de estudio de DevOps; ha abordado muchos temas que son tendencia en la actualidad en términos de la ingeniería de software actual. DevOps sin duda está revolucionando la forma en cómo se construyen, automatizan y se hacen entregas cada vez más frecuentes y rápidas asegurando términos de calidad muy altos. Si usted ha seguido el estudio de esta unidad a conciencia, complementando sus estudios con las actividades recomendadas seguramente estará muy emocionado con los contenidos abordados hasta el momento. Para finalizar esta unidad, se lo invita ha revisar cómo se configuran las canalizaciones y combinaciones CI/CD; revisará un ejemplo básico de implementación en la nube de AWS.

¿Está muy emocionado verdad?

¡Continúe, mantenga el impulso!

4.1.6. Esquemas de implementación y *pipelines*

Una canalización o *pipeline* es un flujo que cubre el ciclo de vida completo de una aplicación. La mayoría de las canalizaciones y procesos de CI/CD se construyen alrededor de una sola aplicación. Están diseñados a medida para hacer todo lo que necesita la aplicación y pueden incluir las siguientes actividades:

- Desarrollo de aplicaciones.
- Pruebas automatizadas.
- Ejecución de análisis de seguridad o código estático.
- Infraestructura de aprovisionamiento/configuración mediante IaC.
- Publicación de artefactos.
- Configuraciones varias.
- Implementación o despliegues de aplicaciones.

Estas canalizaciones hacen todo lo posible para llevar la aplicación a donde necesita ir y el estado en el que debe estar. Incluso puede reutilizar algunas secciones de la canalización en otros lugares, tales como escaneos de seguridad o códigos estáticos. Sin embargo, en la mayoría de los casos, la tubería está muy acoplada a las necesidades específicas o naturaleza de cada aplicación que se está implementando.

Según Atkinson y Edwards (2018), las canalizaciones de CI/CD juegan un papel muy importante en el ciclo de vida de la entrega de software. No solo es responsable de entregar su aplicación a un entorno de forma automática y predecible, sino que también juega un papel en la calidad de ese software. Las canalizaciones de calidad incluyen varias etapas para verificar no solo la calidad del código, sino también si tiene algún problema de seguridad o rendimiento. Las etapas típicas se describen en la Tabla 22:

Tabla 22. Etapas de *pipelines*

Etapa	Descripción
Desarrollo	En esta etapa se construye la aplicación a partir del código fuente. Los binarios creados aquí son los que se implementan más adelante.
Pruebas	En esta etapa se ejecutarán las pruebas unitarias contra el código. Este es el primer control de calidad y garantiza que los cambios realizados por el equipo de desarrollo no rompan la funcionalidad esperada.
Código estático/ Análisis de seguridad	Este es un análisis de los archivos fuente de la aplicación. Este escaneo buscará vulnerabilidades, dudas técnicas y puede verificar la cobertura del código. Esta etapa podría dividirse en múltiples pasos, dada su elección de herramientas para realizar el escaneo.
Almacenamiento de artefactos	Una buena especificación toma los artefactos binarios creados en la etapa de construcción y los guarda para implementaciones posteriores. Esto garantiza que el código que se probó y se implementó con éxito se puede promover más tarde con la confianza de que pasó todas sus comprobaciones.
Implementación	Este proceso crea una implementación simple de los archivos binarios en el entorno especificado, es decir, desarrollo, calidad (QA), producción, entre otros.
Pruebas de extremo a extremo/ rendimiento	Una vez que se implementa el código, puede ejecutar pruebas automatizadas para asegurarse de que funcione correctamente. Esto puede ser mediante un navegador, o golpeando los puntos finales de la API. Esta etapa también se puede dividir en varias etapas, según sus necesidades.

Nota: Tomada de Atkinson y Edwards (2018)

Como puede ver en este escenario, una canalización de CI/CD puede hacer mucho trabajo para garantizar que el código que se entrega es un producto de calidad. Las canalizaciones de CI/CD son el complemento perfecto para cualquier equipo que utilice metodologías ágiles. En una situación ideal, los equipos de desarrollo están construyendo características que están comprometidas con el control de origen a menudo. Cada confirmación inicia una compilación mediante la canalización y se ejecuta todo el conjunto

Índice

Primer bimestre

Segundo bimestre

Solucionario

Referencias bibliográficas

de pruebas. El código se implementa en un entorno inferior para las pruebas de aceptación del usuario, y se crean artefactos para su promoción posterior. Una vez que se completa la función, o tal vez el *sprint*, la nueva característica se implementa en producción.

Como se ha estudiado hasta el momento, la cultura DevOps tiene una estrecha relación con las metodologías ágiles, incluso pudo ver que algunos autores lo consideran como una evolución del agilismo, pero ¿Cómo logra colocar las aplicaciones en entornos productivos de forma automática? Seguramente, requiere de entornos flexibles, adaptables y escalables para poder colocar las aplicaciones en producción de forma automática. Esta transformación cultural de DevOps también implica el aprovechamiento de nuevos elementos tecnológicos, y uno de los más importantes para poner en práctica DevOps es el uso del *cloud computing*, mecanismo para poner en funcionamiento estas prácticas que ha venido estudiando. El uso del *cloud computing*, para complementar estas prácticas, no implica que no se puedan aplicar estos escenarios de implementación en escenarios OnPremise; sin embargo, el modelo de la nube es mucho más flexible y ágil que un escenario OnPremise.

La nube ya cuenta con algunos esquemas y servicios ya implementados que permiten configurar los elementos necesarios para poner en funcionamiento los mecanismos de CI/CD. A continuación, revisará uno de los esquemas de implementación de *pipelines* en la nube más conocidos para conseguir un CI/CD de extremo a extremo y lograr automatizaciones en tiempos récord.

Esquema de implementación como servicio de AWS

AWS es una de las nubes más utilizadas a nivel mundial, junto con Microsoft Azure y Google Cloud, dominan el mercado. Como se mencionó anteriormente, estas nubes ofrecen capacidades como servicios; es decir, las organizaciones no requieren de elementos físicos o esquemas OnPremise para desplegar sus aplicaciones en

Índice

Primer bimestre

Segundo bimestre

Solucionario

Referencias bibliográficas

infraestructuras de TI, sino que utilizan servicios disponibles, flexibles y accesibles mediante internet, que únicamente se utilizan para configurar las canalizaciones DevOps.

Uno de los escenarios más comunes para configurar este tipo de prácticas es el descrito en la Figura 34.

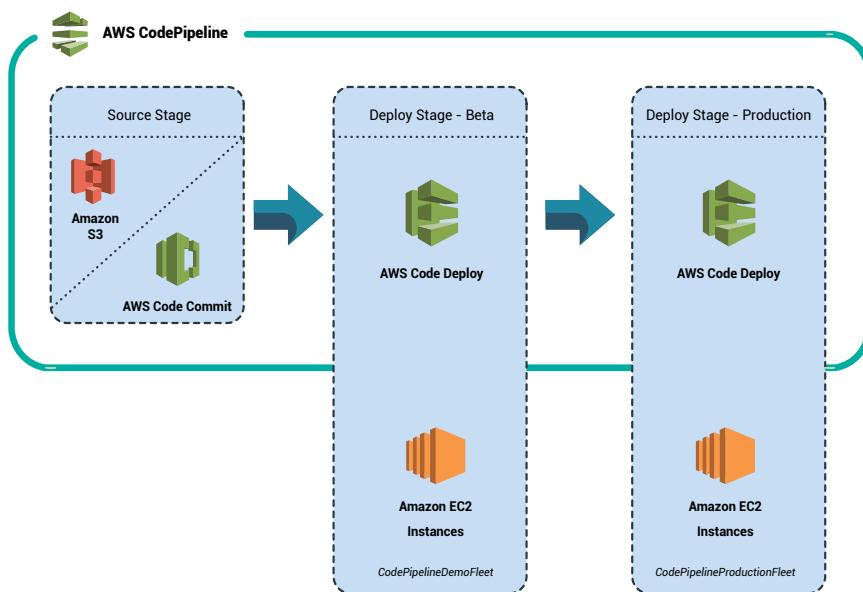


Figura 34. Escenario de canalización DevOps en AWS

Nota: Tomada de AWS (2020)

Como se puede observar, AWS provee una serie de servicios propios para establecer canalizaciones automáticas o *pipelines* compuestos principalmente por el servicio de AWS CodePipeline, que gestiona cada una de las etapas para desplegar aplicaciones en servidores virtuales haciendo uso de instancias EC2 de AWS. Si analiza paso a paso, verá que el esquema de canalización garantiza CI/CD. Revise cómo lo hace.

- AWS CodePipeline: gestiona la canalización CI/CD e integra todos los servicios necesarios para administrar la automatización DevOps de extremo a extremo.
- Source Stage: es la etapa principal en la que se almacena el código fuente de las aplicaciones. AWS provee de dos posibilidades: almacenar el código en el servicio de almacenamiento S3, o utilizar el servicio de AWS CodeCommit para gestionar el repositorio de código y que todos los desarrolladores puedan integrar el código que van generando.
- DeployStage: esta etapa, como se observa en la figura, ofrece dos posibilidades: desplegar la aplicación en un ambiente **beta** o de prueba, y otro de **producción**; dependiendo de las necesidades del proyecto, esto lo logra haciendo uso del servicio AWS CodeDeploy, que cumple con las acciones necesarias para completar un CI/CD; es decir, recupera el código de la etapa Source Stage, lo integra, evalúa y despliega la aplicación en una instancia o servidor de AWS para levantar las aplicaciones. El mismo flujo sirve tanto para el ambiente beta, como de producción.

De esta manera, puede observar cómo con un ejemplo sencillo se puede automatizar una canalización DevOps con CI/CD para los proyectos.

¿Se atreve a intentarlo? Siga la siguiente guía de implementación de AWS.

Además, es importante mencionar que los escenarios pueden incluir otros elementos tecnológicos servicios y formas de desplegar las aplicaciones, incluso se definen escenarios tan flexibles que se puede agregar o integrar otras herramientas que permitan optimizar cada una de las canalizaciones. A continuación, se lo invita a revisar otros escenarios en nubes de [Microsoft Azure](#) y [Google Cloud](#).



Actividades de aprendizaje recomendadas

- Revise el siguiente recurso de video: [AWS CodePipeline tutorial | Build a CI/CD Pipeline on AWS](#). Analice el caso de uso propuesto en el video y vea lo fácil y sencillo que resulta crear una canalización CI/CD en un entorno de *cloud computing*. En este caso, el caso de uso tiene mucha similitud al esquema en AWS estudiado en esta semana. Lo invito a intentarlo.
- Navegue en internet y configure en un entorno local un servidor de integración e implemente una canalización CI/CD que permita que una aplicación sencilla se pueda desplegar de forma automática. Utilice el entorno tecnológico de su preferencia.

Es hora de medir nuestro conocimiento desarrollando la siguiente autoevaluación:



Autoevaluación 5

Estimado estudiante: mediante este cuestionario pondrá a prueba lo aprendido hasta el momento. En las siguientes preguntas, revise cada ítem y seleccione la respuesta correcta. Recuerde que sus resultados constituyen un reflejo del autoaprendizaje de la asignatura.

1. DevOps es el vínculo entre:
 - a. Desarrolladores y operaciones.
 - b. Desarrolladores y analistas.
 - c. Desarrolladores y clientes.

2. Los tres ejes principales de DevOps son:
 - a. La cultura de la colaboración, la integración y las herramientas.
 - b. La cultura de la colaboración, los procesos y las herramientas.
 - c. Los procesos, la automatización y las herramientas.

3. La integración continua es:
 - a. Una práctica de desarrollo de software en la que todos los miembros del equipo integran el código con frecuencia y este es verificado, de forma automática, mediante un servidor de integración.
 - b. Una práctica de desarrollo de software en la que todos los miembros del equipo de desarrollo integran las nuevas características de software mediante un repositorio de código.
 - c. Un proceso automático que integra el código de los desarrolladores para desplegar las aplicaciones.

Índice

Primer bimestre

Segundo bimestre

Solucionario

Referencias bibliográficas

4. La entrega continua es:
 - a. La extensión de la integración continua, su objetivo es recuperar el código fuente y evaluarlo para encontrar defectos.
 - b. Se complementa con la integración continua, su objetivo es compilar el código fuente y generar un artefacto binario.
 - c. La extensión de la integración continua, su objetivo es recuperar el código fuente, compilarlo y desplegarlo para verificar el funcionamiento de la aplicación.
5. La entrega continua se puede activar automáticamente siempre y cuando:
 - a. El proceso de integración sea satisfactorio y sea desplegado automáticamente sin defectos.
 - b. El proceso de integración sea satisfactorio y sea desplegado en un entorno productivo sin evaluar la aplicación.
 - c. Se definan los entornos productivos en los que se desplegará la aplicación
6. El despliegue continuo:
 - a. Hace un control de automatización de extremo a extremo de todo el ciclo de vida de la aplicación, lo logra mediante canalizaciones bien definidas.
 - b. Hace un control de automatización para integrar y desplegar durante todo el ciclo de vida de la aplicación, lo logra mediante pruebas automatizadas.
 - c. Automatiza todo el ciclo de vida de la aplicación, sin embargo, es necesario establecer un control manual para controlar los despliegues automáticos.

Índice

Primer bimestre

Segundo bimestre

Solucionario

Referencias bibliográficas

7. Infraestructura como código es:

- a. Una práctica que consiste en especificar los recursos de infraestructura que se requieren adquirir
- b. Una práctica que consiste en definir y aprovisionar los recursos de infraestructura mediante código.
- c. Una práctica que consiste en crear infraestructuras como código para desplegar aplicaciones de software.

8. La etapa de desarrollo de un pipeline constituye:

- a. Una etapa en la que se escribe el código fuente de una aplicación.
- b. Una etapa en la que se lee el código fuente de una aplicación.
- c. Una etapa en la que se prueba el código fuente de una aplicación.

9. Almacenar artefactos significa:

- a. Almacenar los artefactos generados automáticamente para desplegar y verificar las distintas versiones que se generan.
- b. Almacenar los artefactos para desplegarlos en entornos productivos.
- c. Almacenar los artefactos generados automáticamente y mantener un historial de implementaciones para futuras comprobaciones.

10. Una canalización CI/CD garantiza:

- a. La entrega de software rápida y de calidad.
- b. La entrega de software flexible y eficiente.
- c. La entrega de software para implementaciones futuras.

[Ir al solucionario](#)

Índice

Primer bimestre

Segundo bimestre

Solucionario

Referencias bibliográficas

Ha llegado a la penúltima semana de esta interesante asignatura. Esta semana complementará lo estudiado revisando el paradigma de aplicaciones nativas en la nube o más conocidas como *cloud native apps* (en inglés). Esta semana hará una revisión breve de este tipo de aplicaciones, verá cómo considerar a una aplicación nativa en la nube y qué características deben cumplir estas. Lo invito a abordar esta última y tan emocionante temática de fin de ciclo. No olvide validar sus conocimientos mediante las actividades de aprendizaje recomendadas y las autoevaluaciones.

Continúe con este nuevo paradigma de desarrollo.



Semana 15



Unidad 5. Nuevos paradigmas de desarrollo

5.1. Aplicaciones nativas en la nube

La evolución de los métodos de desarrollo se ha visto marcada por una evidente evolución y nuevas formas de construir software, al abordar esta asignatura ha realizado un recorrido muy importante, pasando por conceptos base, tradicionales, ágiles, automatización, y ha visto cómo estas múltiples formas de construir software han

Índice

Primer bimestre

Segundo bimestre

Solucionario

Referencias bibliográficas

conducido a los equipos a ser cada vez más eficiente; además ha visto cómo estos métodos han causado un impacto considerable y positivo. En este sentido, otras revoluciones como la nube, DevOps, ahora la contenerización y los nuevos estilos arquitectónicos han creado nuevos paradigmas de desarrollo y han conducido a un nuevo mundo del *software*: el mundo de las aplicaciones nativas en la nube.

Según Arundel y Domingus (2019), el término nativo de la nube se ha convertido en una forma abreviada cada vez más popular de hablar de aplicaciones y servicios modernos que aprovechan la nube, los contenedores y la orquestación, a menudo basados en *software* de código abierto que están diseñados para aprovechar los marcos del *cloud computing*. Pero ¿Cualquier aplicación que se implementa en la nube será considerada como una aplicación nativa de nube? Evidentemente solo desplegar una aplicación en la nube no la convierte en nativa en la nube. El concepto va más allá de poder migrar de un entorno OnPremise a Cloud, para que una aplicación sea nativa de nube debe cumplir las características descritas en la Tabla 23.

Tabla 23. Característica de una aplicación nativa en la nube

Característica	Descripción
Automatizable	Si las aplicaciones deben ser implementadas y administradas por máquinas, en lugar de humanos, deben cumplir con estándares, formatos e interfaces comunes. Una herramienta que ayuda a esto es Kubernetes, que proporciona estas interfaces estándar de una manera que significa que los desarrolladores de aplicaciones ni siquiera necesitan preocuparse por ellas.
Ubicuo y flexible	Debido a que están desacoplados de los recursos físicos como los discos, o cualquier conocimiento específico sobre el nodo de cómputo en el que se están ejecutando; las aplicaciones nativas en nube se pueden mover fácilmente entre nodos, esto se logra gracias a la contenerización de aplicaciones.
Resistente y escalable	Las aplicaciones tradicionales tienden a tener puntos únicos de falla la aplicación deja de funcionar si su proceso principal falla; o si la máquina subyacente tiene una falla de hardware, o si un recurso de red se congestiona. Las aplicaciones nativas de la nube, debido a que están distribuidas de forma inherente, pueden estar altamente disponibles mediante redundancia y alta disponibilidad.
Dinámica	Las aplicaciones nativas en nube requieren de un orquestador de contenedores como Kubernetes para programar contenedores que aprovechen al máximo las capacidades de cómputo que proporciona la nube. Se puede ejecutar muchas copias de ellos para lograr una alta disponibilidad y realizar actualizaciones continuas para actualizar sin problemas los servicios y sin perder el tráfico.
Observable	Las aplicaciones nativas de la nube, por su naturaleza, son más difíciles de inspeccionar y depurar. Por lo tanto, un requisito clave de los sistemas distribuidos es la observabilidad –el monitoreo, el registro, el seguimiento y las métricas ayudan a los ingenieros a comprender qué están haciendo sus sistemas y qué están haciendo mal–.

Índice

Primer bimestre

Segundo bimestre

Solucionario

Referencias bibliográficas

Característica	Descripción
Distribuida	Cloud native es un enfoque para crear y ejecutar aplicaciones que aprovecha la naturaleza distribuida y descentralizada de la nube. Se trata de cómo funciona su aplicación, no dónde se ejecuta. En lugar de implementar su código como una entidad única —conocida como monolito—, las aplicaciones nativas de la nube tienden a estar compuestas por microservicios distribuidos, múltiples y cooperativos.

Nota: Tomada de Arundel y Domingus (2019)

Las aplicaciones nativas en la nube son un conjunto de servicios pequeños, independientes y de bajo acoplamiento (RedHat, 2020); un enfoque para diseñar actualizar y escalar aplicaciones con gran rapidez y calidad. A partir de este enfoque, las aplicaciones basadas en microservicios o las aplicaciones que emplean arquitecturas sin servidor —o más conocidas como serverless— toman gran fuerza, ya que estos estilos arquitectónicos presentan grandes bondades y garantizan las características necesarias para que las aplicaciones sean nativas en nube. Otras tecnologías o elementos clave para la construcción de aplicaciones nativas en la nube son DevOps, contenedores, orquestadores, las API, entre otros.

Ha finalizado los contenidos del segundo bimestre, seguramente está muy emocionado por lo que viene. Se lo invita a mantener su ritmo de estudio en la carrera hasta titularse; seguro los contenidos abordados durante el ciclo académico han sido de gran ayuda en su formación. Para culminar esta semana, no olvide reforzar lo aprendido mediante las actividades de aprendizaje recomendadas.



Actividades de aprendizaje recomendadas

- Lea el siguiente artículo: reflexione y responda ¿Cuál es el objetivo de construir o implementar aplicaciones nativas en la nube? ¿Cuál es el papel de la CNCF? ¿Qué grandes empresas lideran la CNCF? ¿Cuál es la importancia de los microservicios en entornos nativos en Cloud?
- Investigue: ¿Cuáles son los cuatro elementos clave de las aplicaciones nativas en la nube?, ¿Cómo estos elementos confluyen entre sí? ¿Qué estrategia utilizaría usted o cómo implementaría el acoplamiento en las aplicaciones que utilizan este paradigma?

Es hora de medir nuestro conocimiento desarrollando la siguiente autoevaluación:



Autoevaluación 6

Estimado estudiante: mediante este cuestionario, usted pondrá a prueba lo aprendido hasta el momento. En las siguientes preguntas, revise cada ítem y seleccione la respuesta correcta. Recuerde que sus resultados constituyen un reflejo del autoaprendizaje de la asignatura.

1. Aplicaciones nativas en cloud son:

- a. Servicios modernos que aprovechan la nube, los contenedores y la orquestación, a menudo basados en software de código abierto.
- b. Servicios modernos que aprovechan la nube, los contenedores y la orquestación, a menudo basados en software de código licenciado.
- c. Servicios que aprovechan la capacidad de cómputo de la nube para ser expuestos mediante internet.

2. Las características de las aplicaciones nativas en nube son:

- a. Automatizable, flexible, escalable, dinámica, observable y distribuida
- b. Escalable, dinámica automatizable, eficaz, observable y distribuida
- c. Automatizable, ubicuo y flexible, resistente y escalable, dinámica, observable, distribuida

Índice

Primer bimestre

Segundo bimestre

Solucionario

Referencias bibliográficas

3. Automatizable significa:

- a. Que definan aplicaciones y funcionalidades de forma automática.
- b. Que las aplicaciones hagan uso de herramientas que utilicen estándares e interfaces que gestionen las implementaciones de forma automática.
- c. Que las aplicaciones hagan uso del factor humano para automatizar tareas como procesos.

4. Ubícuo y flexible significa que:

- a. Las aplicaciones se puedan mover fácilmente entre nodos; es decir, que no tengan dependencias de recursos o servicios para poder funcionar.
- b. Las aplicaciones se puedan migrar fácilmente entre nodos; es decir, que no tengan dependencias de recursos o servicios para poder funcionar.
- c. Las aplicaciones sean lo suficientemente flexibles como para poder automatizar funcionalidades propias de la nube.

5. Resistente y escalable significa que:

- a. Las aplicaciones sean altamente disponibles mediante redundancia y alta disponibilidad.
- b. Las aplicaciones sean redundantes y escalen automáticamente.
- c. Las aplicaciones sean altamente disponibles y resilientes.

Índice

Primer bimestre

Segundo bimestre

Solucionario

Referencias bibliográficas

6. Dinámicas significa que:

- a. Las aplicaciones puedan ejecutar múltiples instancias de aplicación sin perder capacidades de cómputo del servicio.
- b. Las aplicaciones puedan ejecutar múltiples instancias de aplicación sin perder disponibilidad del servicio.
- c. Las aplicaciones puedan escalar automáticamente según demande la concurrencia al servicio.

7. Observables significa que:

- a. Las aplicaciones sean fáciles de utilizar y recuperar.
- b. Las aplicaciones sean fáciles de monitorear e inspeccionar.
- c. Las aplicaciones sean fáciles de observar y corregir.

8. Distribuidas significa que:

- a. Las aplicaciones estén distribuidas y descentralizadas.
- b. Las aplicaciones estén distribuidas y centralizadas.
- c. Las aplicaciones estén disponibles y distribuidas.

9. Las arquitecturas más utilizadas para construir aplicaciones nativas en nube son:

- a. Servicios y serveless.
- b. Microservicios y SOA.
- c. Microservicios y serveless.

10. Los elementos clave de las aplicaciones nativas en nube son:

- a. DevOps, microservicios, contenedores, CI/CD.
- b. DevOps, serveless, contenedores, CI/CD.
- c. DevOps, microservicios orquestadores, CI/CD.

Ir al solucionario

Índice

Primer bimestre

Segundo bimestre

Solucionario

Referencias bibliográficas



Actividades finales del bimestre



Semana 16



Actividades de aprendizaje recomendadas

Apreciado estudiante: ha llegado a la etapa final del primer bimestre, y seguro que usted cumplió con todas las actividades de aprendizaje. Como preparación para el examen presencial se sugiere que:

- Revise las unidades y subtemas estudiados en cada una de las semanas del bimestre, y refuerce los que considera necesarios.
- Analice los recursos facilitados en este texto-guía.
- Revise las autoevaluaciones de las unidades del bimestre.
- Prepárese para la evaluación presencial del primer bimestre y comuníquese con su tutor en caso de que se presenten dudas.
- Consulte el horario y lugar para rendir la evaluación presencial de la asignatura.



4. Solucionario

Autoevaluación 1		
Pregunta	Respuesta	Retroalimentación
1	a	El software es un conjunto de procedimientos y programas que se ejecutan en una computadora para cumplir con una tarea, funcionalidad, proceso o acción específica.
2	c	El Diccionario de la lengua española define software como el conjunto de programas, instrucciones y reglas informáticas para ejecutar ciertas tareas en una computadora.
3	a	Matlab es un software de ingeniería y ciencias, ya que es un sistema para realizar cálculos numéricos y apoya a las ciencias de las matemáticas. Recuerde que este tipo de software es caracterizado por ser algoritmos "devoradores de números".
4	a	Pressman indica que el software heredado tiene ciertas características: puede resultar costoso de mantener y tener mayor riesgo en su evolución, además de que se caracteriza por su longevidad e importancia crítica para el negocio.
5	b	La ingeniería del software es considerada como una disciplina dedicada al estudio del desarrollo, operación y mantenimiento del software. Hay que tener en cuenta que se preocupa por todo el ciclo de vida del software, desde las primeras etapas de la especificación del sistema hasta el mantenimiento del sistema después de ponerlo en operación.
6	a	El concepto de ingeniería de software surge como apoyo a la denominada crisis del software en el año de 1968, durante la segunda etapa cronológica.

Índice

Primer bimestre

Segundo bimestre

Solucionario

Referencias bibliográficas

Autoevaluación 1		
Pregunta	Respuesta	Retroalimentación
7	b	El software de aplicación está enfocado en resolver un problema específico del mundo actual, este tipo de software se construye con base en una necesidad específica que necesite apoyar a las actividades operativas de las personas, por ejemplo, los programas de ofimática.
8	a	Pressman indica que los principios del software ayudan a establecer un conjunto de herramientas mentales para una práctica sólida de la ingeniería de software.
9	c	El principio de modularidad define que un sistema complejo puede dividirse en piezas más simples llamadas módulos, y que un sistema compuesto de módulos es llamado modular.
10	a	El principio de abstracción es un proceso mediante el cual se identifican los aspectos relevantes de un problema ignorando los detalles; es un caso especial del principio de separación de intereses en el cual se separan los aspectos importantes de los detalles de menor importancia.

Ir a la
autoevaluación

Autoevaluación 2		
Pregunta	Respuesta	Retroalimentación
1	c	Algunos autores definen lo que es un proceso de software, específicamente, para Pressman, es un conjunto de actividades (busca lograr un objetivo amplio del trabajo), acciones (conjunto de tareas que producen un producto importante del trabajo) y tareas (se centra en un objetivo pequeño, pero bien definido que produce un resultado tangible) que se ejecutan cuando va a crearse algún producto.
2	a	Para el autor en mención, las actividades del proceso de software que define en su libro son comunicación, planeación, modelado, construcción y despliegue.
3	a	El modelo espiral, que es parte de los modelos evolutivos, además de comprender las mejores características del modelo de proceso en cascada y del de prototipos, incluye actividades del análisis de alternativas, identificación y reducción de riesgos.
4	c	El modelo de proceso iterativo fracciona el proyecto en iteraciones de diferente longitud, cada una de ellas produce un producto completo y entregable, en el que en cada iteración se refina lo realizado en la anterior, con el fin de mejorar los productos (entregables). En cada iteración se mejora más la calidad del producto, por ello las principales características son la revisión y la mejora.
5	a	MSF es una guía de desarrollo de software flexible que permite aplicar, de manera individual e independiente, cada uno de sus componentes. Es escalable y asegura resultados con menor riesgo y de mayor calidad; se centra en el proceso y las personas.
6	c	En RUP, las actividades responden al ¿cómo?; los productos, al ¿qué?, y los flujos de trabajo, al ¿cuándo?
7	b	RUP y Waterfall son metodologías tradicionales y DSDM es parte del grupo de las metodologías ágiles.
8	a	RUP tiene como una de sus características estar centrada en la arquitectura y son características del movimiento ágil estar basados en la práctica y no en la teoría, y que sus equipos son autoorganizados.

Índice

Primer bimestre

Segundo bimestre

Solucionario

Referencias bibliográficas

Autoevaluación 2		
Pregunta	Respuesta	Retroalimentación
9	b	Las metodologías ágiles siguen los modelos de procesos iterativos e incrementales, esto puede verificarlo gráficamente en la Figura 15.
10	a	Los resultados del estudio del Chaos Report, de 2018, indican que el tamaño de un proyecto afecta a la tasa de éxito de este, y son los proyectos grandes los que apenas en un 18% logran culminar, por lo que indican que están más orientados al fracaso.

Ir a la
autoevaluación

Autoevaluación 3		
Pregunta	Respuesta	Retroalimentación
1	a	La programación extrema se enfoca en hacer las prácticas de desarrollo de software lo más sencillas posibles, para que el equipo pueda construir funcionalidades que hagan lo que se requiere y que se puedan adaptar a los diferentes cambios que surjan del proyecto.
2	c	La comunicación, simplicidad, retroalimentación, coraje o valentía, y el respeto son los valores fundamentales de XP, estos deben garantizarse en cualquier proyecto. La no aplicación de alguno de estos podría afectar el éxito del proyecto.
3	b	La retroalimentación es un valor fundamental en cualquier metodología ágil, mediante esta el equipo puede conocer de la parte interesada si las funcionalidades construidas cumplen las expectativas de los clientes.
4	a	El valor de coraje o valentía es, sin duda, un valor importante. Los equipos transparentes y que tienen la valentía de asumir un problema o un defecto permiten que se puedan tomar acciones inmediatas para no retrasar el proyecto.
5	b	El consultor es un rol más bien de apoyo, se lo considera cuando el equipo tiene un impedimento que no puede solucionar. Este rol ayuda en la resolución de problemas específicos.
6	a	La refactorización es una actividad que permite reconstruir, modificar o mejorar alguna funcionalidad de software mediante la recodificación del código fuente. Un equipo de software debe garantizar esta actividad para adaptarse a los cambios que requiera el proyecto, o corregir algún problema.
7	a	El cliente es, sin duda, un rol fundamental. Este determina, valida y retroalimenta sobre las funcionalidades que debe tener el software.
8	a	Las metáforas son una descripción sencilla del contexto del proyecto; mediante esta descripción, todos los miembros del equipo pueden tener un entendimiento común de los diversos aspectos que se abordan en un proyecto.

Índice

Primer bimestre

Segundo bimestre

Solucionario

Referencias bibliográficas

Autoevaluación 3		
Pregunta	Respuesta	Retroalimentación
9	c	TDD es una práctica de desarrollo de software que permite automatizar las pruebas unitarias, primero escribe la prueba y luego el código capaz de completar una funcionalidad.
10	a	Garantizar la productividad del equipo significa no superar las 40 horas semanales, con esto, los miembros del equipo siempre están frescos y dispuestos a cumplir con sus actividades de forma eficaz.

Ir a la
autoevaluación

Autoevaluación 4		
Pregunta	Respuesta	Retroalimentación
1	a	Scrum es una metodología ágil y flexible constituida por un conjunto de buenas prácticas que permiten a los equipos trabajar colaborativamente en entregables pequeños y priorizados para brindar el máximo valor posible a los clientes desde las primeras entregas.
2	c	El product owner o propietario del producto es el responsable de administrar el backlog del producto. Este se encarga de agregar, incluir, actualizar y seleccionar las HU que se deben comprometer para cada sprint específico.
3	a	Una historia de usuario es una descripción en lenguaje natural que expresa, desde el punto de vista del usuario, qué debe hacer una funcionalidad y su criterio de aceptación.
4	a	Un sprint es un espacio de tiempo en el que el equipo de Scrum se dedica a completar los entregables comprometidos para cada sprint.
5	b	Toda actividad y ceremonia en Scrum tiene asignado un tiempo específico para ser abordado, por tal motivo es importante que estas actividades tengan un inicio y fin muy marcado.
6	a	Los roles centrales de Scrum son product owner, Scrum máster, Equipo de Scrum, estos son los encargados de gestionar y construir los entregables funcionales.
7	c	Construir los entregables es responsabilidad del equipo de Scrum o más conocido como equipo de desarrollo.
8	b	Aprobar, estimar y asignar HU es una actividad que se realiza en la fase de planificación y estimación.
9	a	Al finalizar cada sprint se produce un incremento como resultado de estas actividades. Este incremento debe ser totalmente funcional; es decir, funcionalidades de software que son entregadas para que el usuario las manipule.

Índice

Primer bimestre

Segundo bimestre

Solucionario

Referencias bibliográficas

Autoevaluación 4

Pregunta	Respuesta	Retroalimentación
10	a	La ceremonia que no debe durar más de 15 minutos es el Daily Scrum o reunión diaria. Esta se celebra en el equipo de Scrum y sirve para determinar las actividades que se van realizando día a día con el objetivo de remover impedimentos a tiempo.

Ir a la
autoevaluación

Autoevaluación 5		
Pregunta	Respuesta	Retroalimentación
1	a	DevOps es el vínculo e integración de los equipos de desarrollo y operaciones para abordar un proyecto de software.
2	b	Los tres ejes fundamentales de DevOps son la cultura de colaboración entre miembros de los equipos Dev Y Ops, los procesos y el uso de las herramientas para soportar las actividades de automatización.
3	a	La integración continua es una práctica de desarrollo de software en la que todos los miembros del equipo integran el código con frecuencia, y este es verificado de forma automática mediante un servidor de integración.
4	c	La entrega continua es la extensión de la integración continua, su objetivo es recuperar el código fuente, compilarlo y desplegarlo para verificar el funcionamiento de la aplicación.
5	a	La entrega continua se activa automáticamente después de terminado el proceso de integración continua.
6	a	El despliegue continuo hace un control de automatización de extremo a extremo, esto se logra mediante canalizaciones o pipelines bien definidos.
7	b	Infraestructura como código es una práctica que consiste en definir y aprovisionar los recursos de infraestructura mediante código.
8	a	Una etapa de un pipeline constituye una etapa en la que se escribe el código fuente de una aplicación.
9	c	Almacenar artefactos es una actividad que permite mantener un historial versiones de artefactos que puede ser utilizado para futuras comprobaciones.
10	a	Una canalización garantiza la entrega de software rápido y de calidad.

Ir a la
autoevaluación

Autoevaluación 6		
Pregunta	Respuesta	Retroalimentación
1	a	Las aplicaciones nativas en cloud son servicios modernos que aprovechan la nube, los contenedores y la orquestación, a menudo basados en software de código abierto.
2	c	Las características de las aplicaciones nativas en nube son automatizable, ubicuo y flexible, resistente y escalable, dinámica, observable y distribuida.
3	b	La característica automatizable significa que las aplicaciones hagan uso de herramientas que utilicen estándares e interfaces que gestionen las implementaciones de forma automática.
4	a	La característica ubicuo y flexible significa que las aplicaciones se pueden mover fácilmente entre nodos; es decir, que no tengan dependencias de recursos o servicios para poder funcionar.
5	a	La característica resistente y escalable significa que las aplicaciones son altamente disponibles mediante redundancia y alta disponibilidad.
6	b	La característica dinámica significa que las aplicaciones pueden ejecutar múltiples instancias de aplicación sin perder disponibilidad del servicio.
7	b	La característica observable significa que las aplicaciones sean fáciles de monitorear e inspeccionar.
8	a	La característica distribuidas significa que las aplicaciones están distribuidas y descentralizadas.
9	c	Las arquitecturas más utilizadas para construir aplicaciones nativas en nube son los microservicios y serveless.
10	a	Los elementos clave de las aplicaciones nativas en cloud son DevOps, microservicios, contenedores y CI/CD.

Ir a la
autoevaluación



5. Referencias bibliográficas

Arundel, J. y Domingus, J. (2019). *Cloud Native DevOps with Kubernetes: Building, Deploying, and Scaling Modern Applications in the Cloud.*

Atkinson, B. y Edwards, D. (2018). *Generic Pipelines Using Docker: The DevOps Guide to Building Reusable, Platform Agnostic CI/CD Frameworks.* doi.org/10.1007/978-1-4842-3655-0

AWS. (2020). *Cómo configurar una canalización de entrega e integración continua (CI/CD).* aws.amazon.com/es/getting-started/projects/set-up-ci-cd-pipeline/

Blanco, C., Hernández, J., López, P. y Ruiz, F. (2011). *Curso: Ingeniería del Software I (2011)*, Tema: *Ingeniería del Software I (2011)*. ocw.unican.es/course/view.php?id=169§ion=1

Blum, B. I. (1992). *Software Engineering: A Holistic View*. Oxford University Press.

Cadavid, A. N., Martínez, J. D. F. y Vélez, J. M. (2013). Revisión de metodologías ágiles para el desarrollo de software. *Prospectiva*, 11(2), 30-39.

Canós, J. H., y Letelier, M. C. P. P. (2012). Metodologías ágiles en el desarrollo de software.

Díaz, E. (2013). *La naturaleza del software* (Spanish edition). Edición de Kindle.

Índice

Primer bimestre

Segundo bimestre

Solucionario

Referencias bibliográficas

Fernández, J. M., y Cadelli, S. (2014). *Convivencia de metodologías: Scrum y Rup en un proyecto de gran escala* (tesis doctoral, Universidad Nacional de La Plata).

Fowler, M. (2006). *Continuous Integration*. martinfowler.com/articles/continuousIntegration.html

Fritz, B. (1968). Software Engineering: A Report on a Conference Sponsored by NATO Science Committee (pp. 13-45). NATO.

Kim, J., Humble, P. y Debois, J. W. (2016). *The DevOps Handbook: How to Create World-Class Agility, Reliability*. itevolution.com/book/the-devops-handbook/

Ghezzi, C. Jazayeri, M., y Mandrioli, D. (1991). Fundamentals of Software Engineering.

Gutiérrez, D. (2011). *Métodos de desarrollo de software*. Universidad de los Andes.

Hirsch, M. (2005, May). Moving from a Plan Driven Culture to Agile Development. International Conference on Software Engineering (Vol. 27, p. 38).

Hooker, D. (1996). Seven Principles of Software Development.

IEEE. (1990). IEEE Standard Glossary of Software Engineering Terminology (IEEE Std 610.12-1990). [doi: 10.1109/IEEESTD.1990.101064](https://doi.org/10.1109/IEEESTD.1990.101064).

Izaurrealde, M. P. (2013). Caracterización de especificación de requerimientos en entornos ágiles: historias de usuario. *Trabajo de especialidad, Febrero*.

Índice

Primer bimestre

Segundo bimestre

Solucionario

Referencias bibliográficas

Kelly, W. (2019). *Cómo desmitificar el proceso DevOps, paso a paso.* searchdatacenter.techtarget.com/es/consejo/Como-desmitificar-el-proceso-DevOps-paso-a-paso

Kent, B., R. J., y W., C. (1999). *Extreme Programming: Embrace Change.*

Krief, M. (2019). *Learning DevOps: The Complete Guide to Accelerate Collaboration with Jenkins.* books.google.com.ec/books?id=vDa6DwAAQBAJ&printsec=frontcover&dq=learning+devops&hl=es&sa=X&ved=2ahUKEwi5yu6D-4frAhXEg-AKHQbqDpEQ6AEwAHoECAYQAg#v=onepage&q=learning devops&f=false

Kroll, P. y Kruchten, P. (2003). *The Rational Unified Process Made Easy: A Practitioner's Guide to the RUP.* Addison-Wesley Professional.

Mall, R. (2018). *Fundamentals of software engineering.* PHI Learning Pvt. Ltd.

Martínez, A. y Martínez, R. (2014). Guía a Rational Unified Process. Escuela Politécnica Superior de Albacete-Universidad de Castilla la Mancha.

Medina, J.; Pineda, E. y Téllez, F. (2019). Requerimientos de software: prototipado, software heredado y análisis de documentos. *Ingeniería y Desarrollo*, 37(2).

Montero, B. M. Cevallos, H. V. y Cuesta, J. D. (2018). Metodologías ágiles frente a las tradicionales en el proceso de desarrollo de software. *Espirales: Revista Multidisciplinaria de Investigación*, 2(17).

Olarte, L. (2017). *Clasificación de software de sistemas y aplicaciones.* conogasi.org/articulos/clasificacion-de-software-de-sistemas-y-aplicaciones/

Índice

Primer bimestre

Segundo bimestre

Solucionario

Referencias bibliográficas

Ojeda, J. C. y Fuentes, M. D. C. G. (2012). Taxonomía de los modelos y metodologías de desarrollo de software más utilizados. *Universidades*, (52), 37-47.

Pantaleo, G. y Rinaudo, L. (2015). *Ingeniería de software*. Alfaomega Grupo Editor.

Pérez, O. A. (2011). Cuatro enfoques metodológicos para el desarrollo de Software RUP–MSF–XP–SCRUM. *Inventum*, 6(10), 64-78.

Pressman, R. (2010). *Ingeniería de software* (Novena edición ed.). Pearson Educación SA.

Real Academia Española. (2020). *Diccionario de la lengua española*, 23.^a ed. dle.rae.es

RedHat. (2020). ¿Qué son las app cloud nativas? www.redhat.com/es/topics/cloud-native-apps

SCRUMstudyTM. (2016). *Guía SBOKTM*.

Sommerville, I. (2011). *Ingeniería del Software: un enfoque práctico*. (Séptima edición ed.). McGraw-Hill.

Tinoco, O.; Rosales, P., y Salas, J. (2010). Criterios de selección de metodologías de desarrollo de software. *Industrial data*, 13(2), 70-74.

Tomayko, J., y Herbsleb, J. (2003). How Useful Is the Metaphor Component of Agile Methods? : A Preliminary Study. *Computer Science Department*. <http://repository.cmu.edu/compsci/2250>

VisualParadigm. (2020). *Why Fixed Length Sprints in Scrum?* www.visual-paradigm.com/Scrum/why-fixed-length-of-sprints-in-Scrum/