Competencias del Ingeniero en Informática

Formar profesionales capaces de diseñar, desarrollar, proyectar, dirigir, construir, operar y mantener sistemas informáticos, incluyendo las diversas técnicas y actividades relacionadas con el tratamiento de la información como soporte de conocimientos, de la comunicación humana, y entre máquinas. También deben ser capaces de interpretar los nuevos desarrollos tecnológicos en el área de la Informática para la administración de recursos escasos, que sobre bases económicas orienten al ingeniero en la necesidad de lograr óptimos resultados en los plazos de ejecución prefijados y con sentido de responsabilidad social. Deben asimismo poder entender en asuntos de Ingeniería legal, económica y financiera, realizar arbitrajes y pericias, tasaciones y valuaciones referidas a lo específico de la especialidad, en los recursos humanos involucrados y en la enseñanza de los conocimientos tecnológicos y científicos correspondientes

- **Participar en la toma de decisiones estratégicas de una organización y asesorar, en concordancia con las mismas acerca de las políticas de desarrollo de sistemas de información.**
- **Evaluar, clasificar y seleccionar proyectos de sistemas de información y evaluar y seleccionar alternativas de asistencia interna.**
- **Planificar, efectuar y evaluar los estudios de factibilidad inherentes a todo proyecto de diseño de sistemas de información y de modificación o reemplazo de los mismos, así como los sistemas de computación asociados.**
- **Planificar, dirigir, ejecutar y controlar el relevamiento, análisis, diseño, desarrollo, implementación y prueba de sistemas de información.**
- **Evaluar y seleccionar los sistemas de programación disponibles con miras a su utilización en sistemas de información.**
- **Evaluar y seleccionar, desde el punto de vista de los sistemas de información, los equipos de procesamiento y comunicación y los sistemas de base.**
- **Organizar y dirigir el área de sistemas; determinar el perfil de los recursos humanos necesarios y contribuir a su selección y formación.**
- **Participar en la elaboración de programas de capacitación para la utilización de sistemas de información.**
- **Determinar y controlar el cumplimiento de las pautas técnicas que rigen el funcionamiento y la utilización de recursos informáticos en cada organización.**
- Elaborar métodos y normas a seguir en cuestiones de seguridad y privacidad de la información procesada y/o generada por los sistemas de información; participar en la determinación de las acciones a seguir en esta materia y evaluar su aplicación.
- Elaborar métodos y normas a seguir en cuestión de salvaguardia y control, de los recursos físicos y lógicos, de un sistema de computación; participar en la determinación de las acciones a seguir en esta materia y evaluar su aplicación.
- Desarrollar modelos de simulación, sistemas expertos y otros sistemas informáticos destinados a la resolución de problemas y asesorar en su aplicación.
- Realizar auditorías en áreas de sistemas y centros de cómputos así como en los sistemas de información utilizados.
- Realizar arbitrajes, pericias y tasaciones referidas a los sistemas de información y a los medios de procesamiento de datos.
- Realizar estudios e investigaciones conducentes a la creación y mejoramiento de técnicas de desarrollo de sistemas de información y nuevas aplicaciones de la tecnología informática existente.

Software Development Life Cycle

Planificación del concepto

Esta es la primera fase de toda vida de un desarrollo de sistemas. En ella, las personas que promueven el desarrollo del proyecto, junto a los interesados en su conclusión, **definen los sistemas a diseñar** y **determinan el alcance** de todo el proceso, permitiendo que se definan los límites para aspectos como los recursos materiales y humanos, el presupuesto y el tiempo para cada tarea.

Definir los requisitos

Una vez que los interesados en el diseño definen el alcance del trabajo a realizar, los expertos en Tecnologías de la Información empiezan a relacionarse con los usuarios finales del sistema, a fin de **definir los requisitos a cumplir** por el proyecto finalizado. Una vez que se recaban todos los requisitos, los expertos de TI vuelven a reunirse con los usuarios para repasarlos en una **fase de verificación**. Esta fase termina cuando los usuarios finales validan los requisitos que se han definido.

El diseño

Los trabajadores de TI empiezan a convertir los requisitos definidos en una realidad técnica. Es el momento de crear un **diseño técnico** con el que previsualizar el trato que se les darán a los requisitos definidos en el desarrollo del nuevo sistema. Después, se crea un diseño técnico más detallado en el que se da respuesta a todas las funciones tecnológicas que necesita el sistema para cumplir sus objetivos.

Fase del desarrollo y pruebas

En esta fase los especialistas en TI empiezan a crear el sistema diseñado. Crean el **software y la arquitectura física** necesaria para albergar la base de datos del sistema. Una vez terminada la

construcción de todos los componentes del sistema, empiezan a realizarse las **pruebas**, durante las cuales los responsables de la calidad se aseguran de que los requisitos de negocio se cumplen, usando un esquema detallado de testeo.

La puesta en marcha

Los especialistas en TI ponen en manos de los usuarios finales el sistema completado, a fin de que puedan **empezar a utilizarlo**, suministrándoles, además, toda la documentación necesaria para aprender a utilizarlo correctamente. También suelen dedicarse algunas horas a formar en el uso del sistema a los usuarios.

Operaciones y mantenimientos

Durante una fase de operación total, los expertos desarrolladores controlan el sistema para **asegurar que cumple los requisitos** de negocio solicitados antes del diseño. Se ofrece un servicio de **mantenimiento y de soporte a los usuarios** para garantizar que el sistema sigue funcionando correctamente.

Disposición

Esta fase comprende el fin del ciclo de vida del sistema y su retiro del funcionamiento. Se deben seguir unos **pasos sistemáticos** para finalizar el sistema en un entorno de seguridad, que permita **conservar toda información útil o sensible** de cara a continuar con los negocios en un sistema nuevo.

Conociendo las fases del ciclo SDLC, comprenderemos mejor la importancia de un desarrollo estructurado.

Proceso Básico del Ciclo de Vida de un Sistema

Si bien, es cierto que **existen diversas metodologías y formas de desarrollar software, la realidad es que hay modelos tan antiguos que ya son como básicos al momento del ciclo de vida de un software**. Un ejemplo de esto, es el modelo en cascada para el proceso de desarrollo de un sistema, con el cuál veremos a ciencia cierta el proceso básico, del cual muchos modelos más se empezarán a desarrollar.

• **Planificación**. El primer punto importante en el ciclo de vida de software, es analizar brevemente los requerimientos que el cliente pide para la elaboración del sistema que necesita. Esta etapa requiere de cierto conocimiento para poder entender la idea que el cliente propone, además de que regularmente debes tomar nota con cada uno de los puntos importantes que se te solicitan, de este modo puedes hacer una planificación al momento y llegar incluso a determinar los tiempos de desarrollo que te llevará, antes de proceder a entregar el producto final. Un punto importante por el cual la planificación siempre debe estar en los ciclos de vida del software. Es porque el cliente se imagina su producto final de una forma tan abstracta, que necesitas hacer que ponga los pies en el suelo para obtener resultados que se acerquen más a la realidad.

• **Implementación**. Una vez que hemos platicado con el cliente y tenemos lo que es un análisis de requerimientos, necesidades y funcionalidades por parte de una aceptación en ambas partes, entonces procedemos con lo que es el ciclo de vida de desarrollo de software. Para este punto, existen una infinidad de metodologías de desarrollo de software, que nos ofrecen la posibilidad de trabajar de distintas formas. Más adelante hablaremos más específicamente de ellas, sin embargo la implementación, es básicamente la parte donde los programadores empiezan a codificar o desarrollar el sistema que se necesita, básicamente se trata del ciclo de vida del desarrollo de sistemas, sin importar el lenguaje de programación mediante el cual se vayan a elaborar.

• **Pruebas**. Una vez que el sistema se va desarrollando, es importante para el ciclo de vida del desarrollo del software, que se realicen ciertas pruebas conforme se vaya avanzando. La idea es que no se termine el desarrollo para poder hacer pruebas, si no que mucho antes, durante el proceso de creación, estas ya se puedan ir ejecutando. Las pruebas nos van a permitir ver si el sistema que se está desarrollando es funcional, si tiene algunos errores, si le faltan ciertas cosas para funcionar correctamente, pues básicamente para avanzar al siguiente punto del ciclo de desarrollo de software, será necesario haber pasado las pruebas correctamente.

• **Documentación**. Muchas metodologías de lo que es el ciclo de vida software, van creando documentación, conforme se va avanzando en el desarrollo del sistema. Sin embargo algunas otras prefieren no hacer la documentación hasta el final. Ahora si que sea cual sea la metodología que se elija, la documentación siempre será importante, pues considera que no siempre vas a estar vos y tu equipo disponibles y cuando otro equipo llegue a programar lo que ustedes hicieron, será indispensable que haya una documentación de la cual se puedan basar, para poder empezar a desarrollar nuevamente el sistema incompleto.

• **Despliegue**. Ya casi llegando a lo que son las últimas etapas del desarrollo de software, nos encontramos con el Despliegue. Este no es otra cosa, más que el momento en que el sistema ya está terminado y ha sido aprobado para que se elabore el producto final. ahora será el momento de distribuirlo y celebrar, pues gracias al equipo de trabajo es como se habrá llegado a esta fase. Lamentablemente, de las etapas de desarrollo de software, esta es a la cual muchos nunca llegan. Pues una gran cantidad de software incompleto se queda en el camino

debido a distintos puntos o motivos. Puede ser que el equipo no se unió, el cliente declinó, el proyecto no fue funcional, etc. Así que de haber llegado a esta fase de desarrollo de software, tu como tu equipo deberán sentirse orgullosos y es momento de volver a desarrollar un proyecto más.

• **Mantenimiento**. La última de las fases del desarrollo de software, es el mantenimiento. Que creías, que nunca más verías al software que hicieron, terminaron y distribuyeron. Pues claro que si lo volverías a ver, pues es momento de darle mantenimiento. Acá además se pueden agregar lo que son las actualizaciones, dependiendo del tipo de desarrollo. Si el equipo siguió trabajando con el software desarrollado y encontraron formas de hacerle mejoras, entonces parte del mantenimiento será actualizarlo a la versión final en todo momento.

Paradigmas de los Modelos del Ciclo de Vida del Software

Una de las cosas principales, que se deben elegir al momento de empezar un proyecto de desarrollo de software, son precisamente las etapas del desarrollo de software. Si bien, nos queda claro que no todos tenemos las mismas ideas y no todos pensamos de la misma manera, afortunadamente ya existen modelos preestablecidos bajo los cuales podemos elaborar nuestro proyecto.

Es por eso que a continuación se muestra cuales son algunos de los paradigmas de los Modelos del ciclo de vida de desarrollo de sistemas. Bajo los cuales se pueden encontrar una gran cantidad de modelos distintos para desarrollar software, veamos.

• **Paradigma Tradicional**. Existen algunas metodologías del ciclo de vida de desarrollo de sistemas, que se manejan a la antigua, a estas también se le conocen como paradigmas tradicionales. Si bien, es verdad que las metodologías actuales están basadas con fundamentos de lo que fueron los paradigmas tradicionales, hoy en día ya hemos evolucionado, sin embargo los paradigmas tradicionales ahí se mantienen. Estos paradigmas, se caracterizan principalmente por ser lineales sin vuelta atrás, es decir, se trataba de completar cada proceso de principio a fin, hasta que quedara listo para avanzar a la segunda fase del ciclo del software. Esto generaba grandes dificultades y pérdidas de tiempo si se encontraba algún error en una fase avanzada, pues el proceso a realizarse era, volver atrás y volver a pasar nuevamente por las fases que ya se habían hecho y reestructurar de acuerdo a las modificaciones, pero todo con un proceso lineal, lento y tardado.

• **Paradigma Orientado a Objetos**. Una de las genialidades más exquisitas, es el desarrollo de software mediante programación orientada a objetos. Con esta forma del ciclo de vida de los sistemas, lo que se pretende es que el código fuente sea reutilizable para otros proyectos o mini proyectos alternos relacionados con el programa base, pues se utilizan Clases. Básicamente la etapas de desarrollo de software en el paradigma orientado a objetos, se conforma principalmente lo que es la creación de clases, seguido del análisis de requisitos, un paso fundamental para determinar no solamente la duración del desarrollo, si no también los costos al final del proyecto. Y por supuesto el diseño, pues ya con el paradigma orientado a objetos, el diseño es mucho mejor que con un paradigma tradicional.

• **Paradigma de Desarrollo Ágil**. Los modelos de ciclo de vida ágiles, son de los más utilizados hoy en día. El objetivo de este paradigma, es el desarrollo de proyectos en poco tiempo. Para lo cual, se hace una eliminación de procesos tediosos, se agilizan las fases de desarrollo, las iteraciones se hacen en un corto periodo de tiempo, los riesgos se desechan y se evitan para no tener que lidiar con ellos y siempre se da solución a los problemas de forma rápida. Si algo

demora mucho en dar solución, lo mejor es dejarlo de lado y seguir avanzando. Una de las principales diferencias del paradigma de desarrollo ágil con los paradigmas anteriores, es que el cliente se ve involucrado en el proyecto durante el desarrollo de este. A diferencia del paradigma tradicional donde el cliente solo está al principio, de igual forma en el paradigma orientado a objetos sucede lo mismo. Acá el cliente interfiere, da mejoras, propone ideas y se mantiene al tanto del desarrollo del producto. Lo que ayuda aún mas, pues el producto final se realiza de forma satisfactoria en un menor lapso de tiempo.

Ciclo de Vida del Software en las distintas Metodologías

El ciclo de vida de un proyecto se software, empieza cuando se da la recolección de requerimientos para el programa a desarrollar y termina cuando el producto ha quedado completado y es entregado al cliente que lo pidió. Sin embargo en el intermedio, hay una gran cantidad de fases por las cuales se tiene que pasar y cada metodología tiene fases distintas en su ciclo de desarrollo de programas, es por eso que a continuación, veremos como están compuestos cada uno de los modelos de ciclo de vida del software, sin entrar a escenarios profundos, pues son tantas metodologías por mencionar que nos podríamos llevar todo el día.

Modelo en Cascada

Una de las metodologías más antiguas en lo que es el ciclo de vida de un modelo informático, es el modelo de cascada. Esta metodología es lineal y consta de algunas fases que hay que seguir y completar para poder avanzar a la fase siguiente. No es precisamente la mejor metodología, pero si se utiliza de forma correcta los resultados pueden ser muy buenos. Está compuesta por las siguientes fases:

1. Requerimientos
2. Diseño
3. Implementación y Desarrollo
4. Integración
5. Pruebas o Validación
6. Despliegue o Instalación
7. Mantenimiento

Como se puede ver, **el ciclo de vida de un programa realizado bajo la metodología en cascada, es extenso pero muy bien estructurado**. El detalle aquí es que no se puede saltarte fases ni volver a repetirlas, por ejemplo.

Si se realiza un análisis de requerimientos, avanzamos a diseñar el programa y ya estamos en el desarrollo y de momento el cliente nos dice que desea modificar los requerimientos, digamos que por tratarse del modelo en cascada, no es posible volver atrás. Por lo tanto se tendría que reiniciar el proyecto o bien concluirlo y ver como queda el software al final.

Como se mencionaba, **es una metodología lineal en cascada y si no se completa cada una de las fases al 100%, no es posible avanzar a la fase que sigue**, así es como funciona y se debe seguir al pie de la letra, por muy exagerado que esta parezca.

Modelo en el Espiral

El modelo espiral en ingeniería del software tiene un enfoque muy distinto al modelo de cascada, principalmente porque su enfoque va dirigido hacia el análisis de riesgos. El modelo de ciclo de vida en espiral, consiste en realizar diversas iteraciones, pasando por cada una de sus fases una y otra vez. A diferencia del modelo cascada que no tiene vuelta atrás, en el modelo en espiral se pueden hacer las iteraciones que se consideren necesarias y estas son sus fases principales:

1. Determinación de Objetivos
2. Análisis de riesgos
3. Desarrollo y Pruebas
4. Planificación

Entre las principales ventajas de desarrollar un proyecto con el modelo espiral, es que los riesgos se van disminuyendo conforme avanzan los ciclos o iteraciones, de hecho no se puede avanzar a un ciclo nuevo, si no se ha dado solución a todos los riesgos latentes. Lamentablemente el modelo es realmente costoso y para que se pueda tener un alto nivel de eficacia en la evaluación final del proyecto con este ciclo de vida, se necesita que del equipo tenga un gran nivel de conocimientos y si es posible buena experiencia para superar cualquier riesgo al cual se puedan enfrentar.

Modelo Iterativo o por Prototipos

Es un modelo que se maneja a base de prototipos, es decir. Es uno de los primeros ciclos de vida que permitían que el código fuente fuera reutilizable, sin embargo con el modelo iterativo no solo es utilizable, si no que para muchos, estos prototipos pueden llegar a ser el producto final que siempre quisieron, lo cual lo hace realmente relevante y destacable, por encima del resto de los modelos anteriores que se pueda encontrar.

Básicamente, las fases del ciclo de vida del sistema, son las siguientes:

1. Inicialización
2. Iteración
3. Lista de Control

Una de las principales ventajas del modelo iterativo, es que la retroalimentación a los usuarios se proporciona desde muy temprano, haciendo que adentrarse en el proyecto sea demasiado sencillo. Por supuesto que el hecho de contar con iteraciones nos da ciertas ventajas, pues con cada iteración realizada, se van separando las partes complejas de el, permitiendo más el acceso al software. Y por supuesto, un sistema creado mediante el ciclo de vida iterativo, tiende a no fallar casi, lo cual es garantía de satisfacción para el cliente en este caso o para la empresa que está implementando esta metodología.

Modelos del Ciclo de Vida del Desarrollo Ágiles

Las tendencias, con el paso del tiempo suelen cambiar para bien y en el caso de las metodologías del ciclo de vida desarrollo de software no es la excepción. Y un claro ejemplo de esto, son los modelos de desarrollo ágil. **Estos procesos se caracterizan por estar basados en las etapas del ciclo de vida del software tradicionales, pero combinándolas con algunas técnicas y siendo aún mas solapadoras** en cuando al orden que se deben ejecutar.

¿Qué es y cuáles son los valores y principios del Manifiesto Ágil?

En marzo de 2001, 17 críticos de los modelos de producción basados en procesos, convocados por Kent Beck, se reunieron en Snowbird, Utah, para discutir sobre el desarrollo de software. En la reunión se acuñó el término "Métodos Ágiles" para definir a aquellos que estaban surgiendo como alternativa a las metodologías formales, a las que consideraban excesivamente "pesadas" y rígidas por su carácter normativo y fuerte dependencia de planificaciones detalladas, previas al desarrollo en el momento de mayor incertidumbre.

Los "Métodos Ágiles" no son una metodología, son una mentalidad y comportamiento guiados por valores y principios.

Los valores ágiles deben de estar fuertemente interiorizados en quienes nos consideremos ágiles, creemos en ellos y deben de formar parte de nuestro ADN, ya que afectan directamente a como las personas nos vamos a comportar.

Valores del Manifiesto Ágil:

Los integrantes de la reunión resumieron en cuatro valores lo que ha quedado denominado como "Manifiesto Ágil", que son los valores sobre los que se asientan los métodos ágiles:

Estamos poniendo al descubierto mejores métodos para desarrollar software, haciéndolo y ayudando a otros a que lo hagan. Con este trabajo hemos llegado a valorar:

- **A los individuos y su interacción**, por encima de los procesos y las herramientas
- **El software que funciona**, por encima de la documentación exhaustiva
- **La colaboración con el cliente**, por encima de la negociación contractual
- **La respuesta al cambio**, por encima del seguimiento de un plan

Aunque hay valor en los elementos de la derecha, valoramos más los de la izquierda.

Principios del Manifiesto Ágil:

Tras los cuatro valores descritos, los firmantes redactaron los siguientes, como los principios que de ellos se derivan:

- Nuestra principal prioridad es satisfacer al cliente a través de la entrega temprana y continua de software de valor

- Son bienvenidos los requisitos cambiantes, incluso si llegan tarde al desarrollo. Los procesos ágiles se doblegan al cambio como ventaja competitiva para el cliente
- Entregar con frecuencia software que funcione, en periodos de un par de semanas hasta un par de meses, con preferencia en los períodos breves
- Las personas del negocio y los desarrolladores deben trabajar juntos de forma cotidiana a través del proyecto
- Construcción de proyectos en torno a individuos motivados, dándoles la oportunidad y el respaldo que necesitan y procurándoles confianza para que realicen la tarea
- El método más eficiente y efectivo de comunicar información al equipo de desarrollo y entre sus miembros es la conversación cara a cara
- El software que funciona es la principal medida del progreso
- Los procesos ágiles promueven el desarrollo sostenido. Los patrocinadores, desarrolladores y usuarios deben mantener un ritmo constante de forma indefinida
- La atención continua a la excelencia técnica enaltece la agilidad
- La simplicidad como arte de maximizar la cantidad de trabajo que no se hace, es esencial
- Las mejores arquitecturas, requisitos y diseños emergen de equipos que se auto-organizan
- En intervalos regulares, el equipo reflexiona sobre la forma de ser más efectivo y ajusta su conducta en consecuencia

Kent Beck	James Grenning	Robert C. Martin
Mike Beedle	Jim Highsmith	Steve Mellor
Arie van Bennekum	Andrew Hunt	Ken Schwaber
Alistair Cockburn	Ron Jeffries	Jeff Sutherland
Ward Cunningham	Jon Kern	Dave Thomas
Martin Fowler	Brian Marick	

Modelo Scrum

El ciclo de vida del sistema, puede agilizarse si se utiliza la metodología Scrum, uno de los modelos del ciclo de vida del desarrollo del software más populares y mas recientes.

El modelo Scrum, se encuentra basado en lo que es el desarrollo incremental, es decir, conforme pasen las fases y las iteraciones, mayor va a ser el tamaño del proyecto que se esté desarrollando, es por eso que uno de los principales requisitos para llevarlo a cabo, es que el equipo de desarrollo sea de calidad. Teniendo una alta calidad en el equipo, tendremos garantizado un excelente funcionamiento.

Como mencionaba al principio, **el modelo Scrum, deja de seguir metodologías lineales, podemos despedirnos del modelo cascada y secuencial, pues ahora procedemos a solapar las fases y no importará en que momento se tenga que volver atrás**, siempre habrá un equipo de trabajo de buena calidad, que tenga ese soporte para aguantar los cambios que son ciertamente normales dentro de la metodología Scrum.

Por último, como ingrediente vital tenemos la comunicación, y es que en este modelo, cada integrante del equipo de trabajo tendrá la posibilidad de desempeñar cada uno de los roles. Con el modelo scrum se debe estar comunicado con el equipo de trabajo en todo momento, para estar al tanto de los sucesos.

Ahora veremos brevemente, cuales son los procesos que el modelo Scrum utiliza:

1. Product Backlog
2. Sprint Backlog
3. Sprint Planning Meeting
4. Daily Scrum o Stand-up Meeting
5. Sprint Review
6. Sprint Retrospective

Estas son las fases del ciclo de vida del software en esta metodología, el cuál básicamente **consiste en realizar un análisis de los requerimientos del sistema (Product Backlog**), señalar cuales serán los objetivos a corto o mediano plazo dentro de un **sprint**, osea, la fase de desarrollo. Posteriormente los desarrolladores harán lo suyo, se realizan algunas pruebas y se retroalimenta de acuerdo a lo conseguido al terminar la última fase. Recuerda que aquí, se pueden añadir nuevas cosas en todo momento, pues el modelo Scrum no se bloquea en ninguna de sus fases.

Roles en Scrum

Product Owner

Participa en la definición del producto. Representa al negocio y prioriza historias de usuario. Nexo de unión entre los implicados. Debe maximizar el valor del producto.

Scrum Master

Encargado de que se cumplan las reglas de Scrum. Resuelve posibles conflictos. Motiva y protege al equipo. Su tarea es facilitar el trabajo al equipo.

Development Team

Equipo multidisciplinar autoorganizado (desarrolladores, QA, diseño, UX, arquitectos...) Encargado del desarrollo del producto.

Ceremonias de Scrum

Daily Scrum

Reunión diaria donde sólo los involucrados pueden hablar. Se responden 3 preguntas:

- ¿Qué hiciste ayer?
- ¿Qué vas a hacer hoy?
- ¿Tienes algún bloqueo?

Sprint Review

Al final del sprint. Se revisa el trabajo que se ha completado y el que no se ha terminado. Se hace una demostración y se obtiene feedback.

Sprint Planning

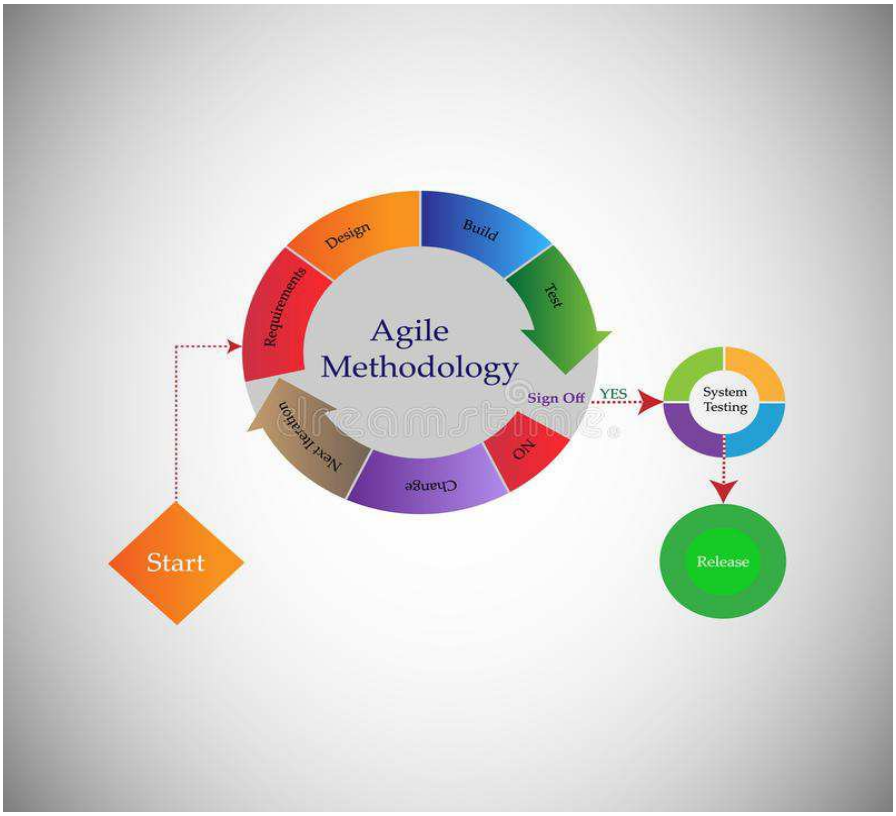
Al inicio de cada sprint. Se selecciona el trabajo que se va a hacer en este sprint y se estima.

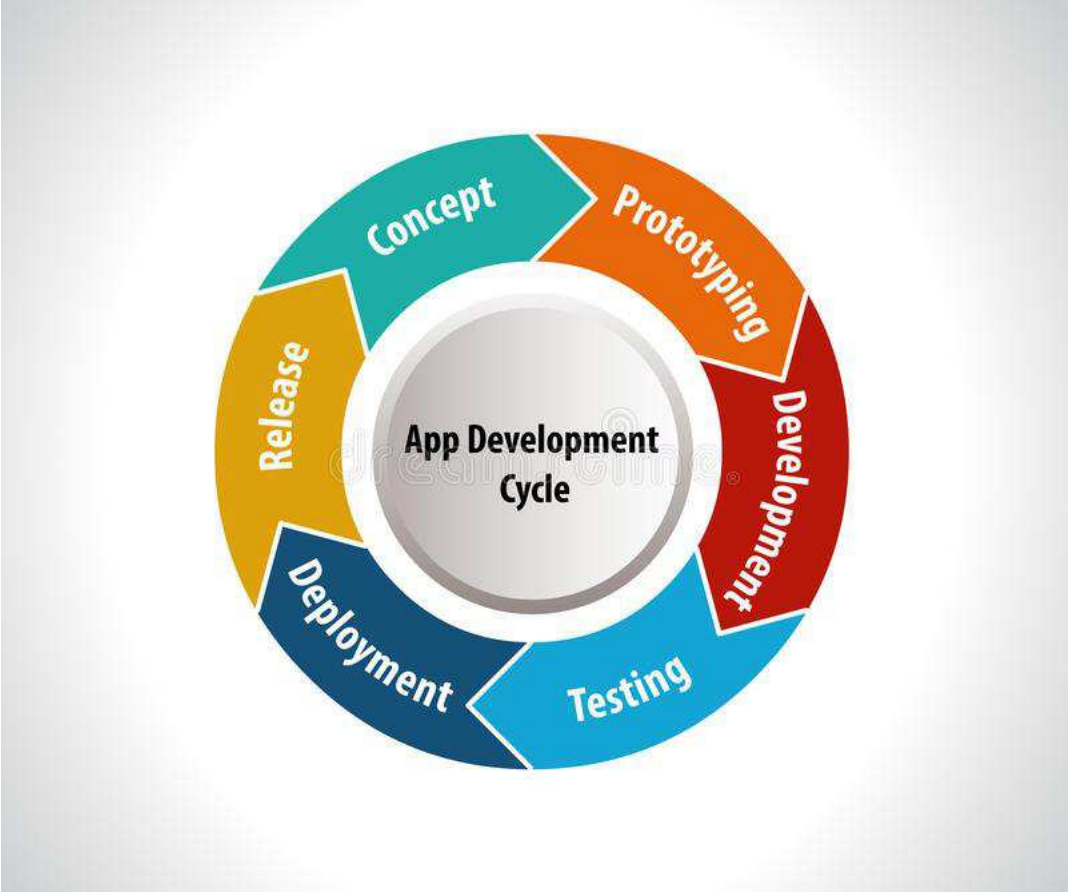
Sprint Retrospective

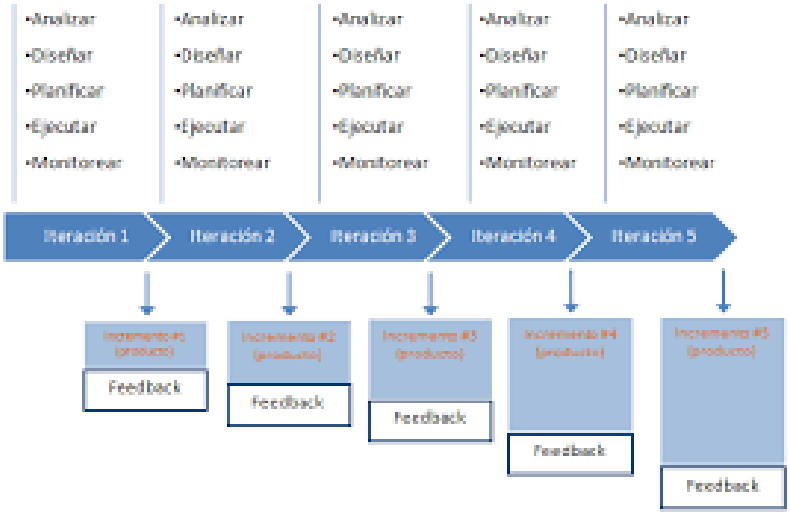
Al final del sprint. Se reunen todos los implicados para analizar qué se ha hecho bien y se debe seguir haciendo y qué se ha hecho mal y se debe cambiar.

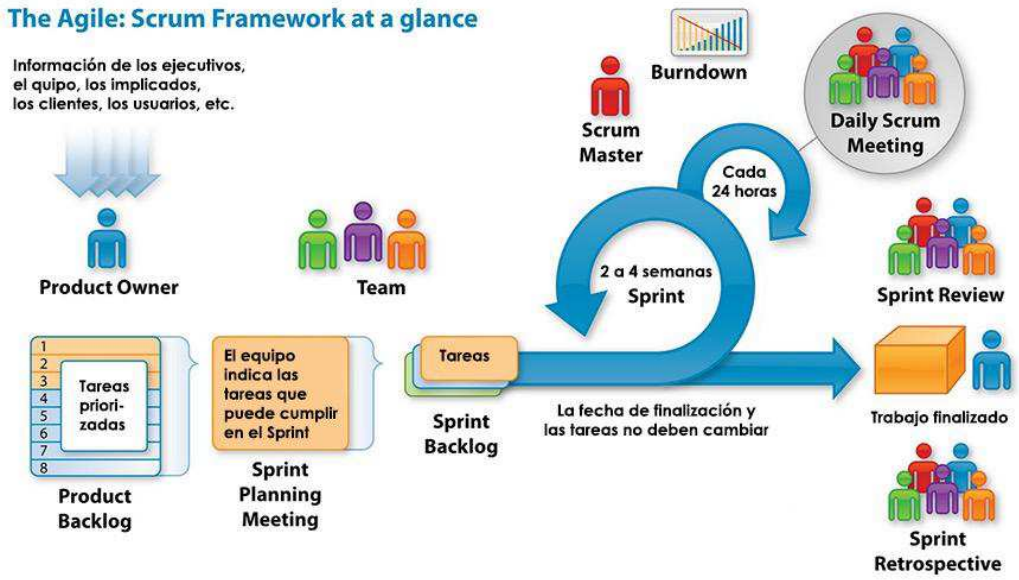
La importancia de Priorizar

- Es una responsabilidad del Product Owner
- Se debe priorizar por el valor que aporta cada historia
- No se debe priorizar por la complejidad para desarrollarlas
- Existen muchas técnicas, como por ejemplo:
  - Modelo Kano:
    - Requisitos obligatorios (Básicos)
    - Requisitos deseados (Esperados)
    - Requisitos no esperados (Inesperados)
    - Indiferentes (No aportan valor)
  - MoSCoW: (Must, Should, Could y Won't)


Agile
Methodology
Requirements
Design
Build
Test
Sign Off
YES
NO
Change
Next Iteration
Start
System
Testing
Release


Concept
Prototyping
Development
Testing
Deployment
Release
App Development Cycle


•Analizar
•Diseñar
•Planificar
•Ejecutar
•Monitorear
Iteración 1
Iteración 2
Iteración 3
Iteración 4
Iteración 5
Feedback
Feedback
Feedback
Feedback
Feedback


The Agile: Scrum Framework at a glance
Información de los ejecutivos, el quipo, los implicados, los clientes, los usuarios, etc.
Burndown
Scrum Master
Daily Scrum Meeting
Cada 24 horas
Product Owner
Team
2 a 4 semanas Sprint
Sprint Review
1
2
3
4
5
6
7
8
Tareas priori-zadas
El equipo indica las tareas que puede cumplir en el Sprint
Tareas
La fecha de finalización y las tareas no deben cambiar
Trabajo finalizado
Product Backlog
Sprint Planning Meeting
Sprint Backlog
Sprint Retrospective

	Scrum	Gestión tradicional de proyectos
El énfasis está en	Las personas	Los procesos
Documentación	Sólo mínima; según se requiera	Integral
Estilo de procesos	Iterativo	Lineal
Planificación por adelantado	Baja	Alta
Priorización de requerimientos	Según el valor del negocio y regularmente actualizada	Fijo en el plan de proyecto
Garantía de calidad	Centrada en el cliente	Centrada en el proceso
Organización	Auto-organizada	Gestionada
Estilo de gestión	Descentralizado	Centralizado
Cambio	Actualizaciones al Backlog Priorizado del Producto	Sistema formal de gestión del cambio
Liderazgo	Liderazgo colaborativo y servicial	Mando y control
Medición del rendimiento	El valor del negocio	Conformidad con el plan
Retorno sobre la inversión (RSI)	Al comienzo y a lo largo del proyecto	Al final del proyecto
Participación del cliente	Alta durante todo el proyecto	Varía dependiendo del ciclo de vida del proyecto

Modelo Kanban

El modelo Kanban, es uno de los modelos más visuales de las metodologías ágiles. **Consiste en la creación de un tablero con etiquetas, donde se seccionan cada una de las fases de su desarrollo**, además se clasifica de acuerdo a los equipos de trabajo y se les asignan objetivos a corto, mediano y largo plazo.

Entre las ventajas de este modelo del ciclo de vida del software, **destaca el hecho de no tener un orden tal cual, de hecho todas las fases comienzan a trabajar a la par, no hay tiempos de espera y básicamente su objetivo es que los desarrolladores y programadores esten trabajando todo el tiempo**. Si concluyes con las fases del proyecto que te corresponde, seguramente tendrás que avanzar en fases del nuevo proyecto que está por venir.

Por supuesto, la metodología Kanban, también requiere de un equipo totalmente capacitado, pues solamente de esta forma se podrán lograr los objetivos. Así que aquí les muestro las fases del proceso del ciclo de vida de un sistema, mediante la metodología japonesa Kanban:

1. Definir el Flujo de Trabajo
2. Fases del Ciclo de Producción
3. Stop Starting, start finishing
4. Tener un Control

Modelo XP o Programación extrema

Posiblemente la más destacada de las metodologías ágiles para los ciclos de vida de un software, es la metodología XP o modelo de programación extrema. A diferencia del resto de las metodología del mundo, habidas y por haber, esta es adaptable de acuerdo a las necesidades y requerimientos que se tengan que implementar, con la ventaja de que podemos hacer uso de cualquier modelo anterior para el desarrollo y de inmediato salirnos y programar otras cosas, es muy solapador y permite mucha más libertad en el equipo de trabajo que el resto de los modelos.

Además, si querías una diferencia aún mayor, en la metodología de programación extrema, el cliente se encuentra involucrado en el proceso de desarrollo, lo que hace que al final el producto pueda estar terminado en un menor tiempo, pues evitamos muchas perdidas de tiempo, elaborando cosas que no son y que en la revisión al cliente no le agradarán, acá el cliente va viendo lo que se va desarrollando y tiene la libertad de proponer cambios, ideas, requerimientos o actualizaciones sin ningún problema.

Los valores que componen a al modelo de programación extrema, son los siguientes:

1. Comunicación
2. Simplicidad
3. Retroalimentación
4. Valentía
5. Respeto

Esta serie de valores, son de suma importancia para que se pueda llevar a cabo un proyecto de alta calidad. Cada uno de ellos, tiene su razón de ser y existir, por ejemplo, la comunicación, la cual debe estar incluso con el cliente y ni hablar del resto de los equipos de trabajo. La simplicidad corresponde al hecho de no hacer cosas que quiten mucho tiempo, la idea es

terminar rápido y las cosas que sean muy tardadas es mejor dejarlas de lado. **La retroalimentación es vital, más cuando los equipos de trabajo deben ser de dos personas, siempre es bueno aprender cosas nuevas de nuestro compañero de trabajo** y esto seguramente todos lo hemos vivido alguna vez.

La valentía es un valor integrado como programador, pues deber ser valiente para afrontar los cambios que se vengan, tomar decisiones radicales y en todo momento mantener esa fuerza que tanto a ti como a tu equipo de trabajo los debe mantener a tope. Y por supuesto el respeto, esto es en todo el equipo de trabajo, hasta el cliente debe tener un margen de respeto por el equipo de desarrollo. Con estos valores la metodología tendrá una buena formación, pero vamos a ver cuales son las características principales de la programación extrema:

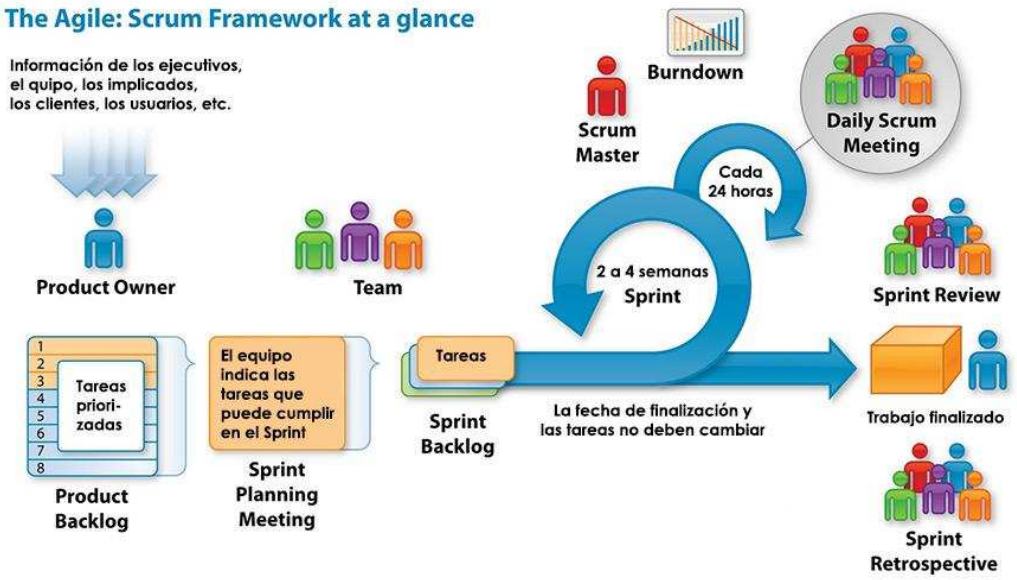
1. Tipo de Desarrollo Iterativo e incremental
2. Pruebas Unitarias
3. Trabajo en Equipo
4. Trabajo junto al cliente
5. Corrección de Errores
6. Reetructuración del Código
7. El Código es de todos
8. Código simple es la clave

Como te puedes dar cuenta, pasos como hacer el código simple o las pruebas unitarias para prevenir errores y tener que darle muchas vueltas al código, además de que las fases se segmentan en pequeñas porciones, para que si hay errores, estos puedan ser modificados fácilmente.

Sin embargo seguramente notaste que no hay documentación por ningún lado, esto es porque con tanta comunicación, en realidad no es necesaria, sin embargo dentro del código se van dejando comentarios con las cosas que otro programador no pueda llegar a entender y se utilizan variables o clases entendibles para que todo el mundo tenga la facilidad de comprenderlo. Ya solamente en caso muy necesario, se procede a hacer documentación breve para algunas partes, pero regularmente no se utiliza de forma tradicional.

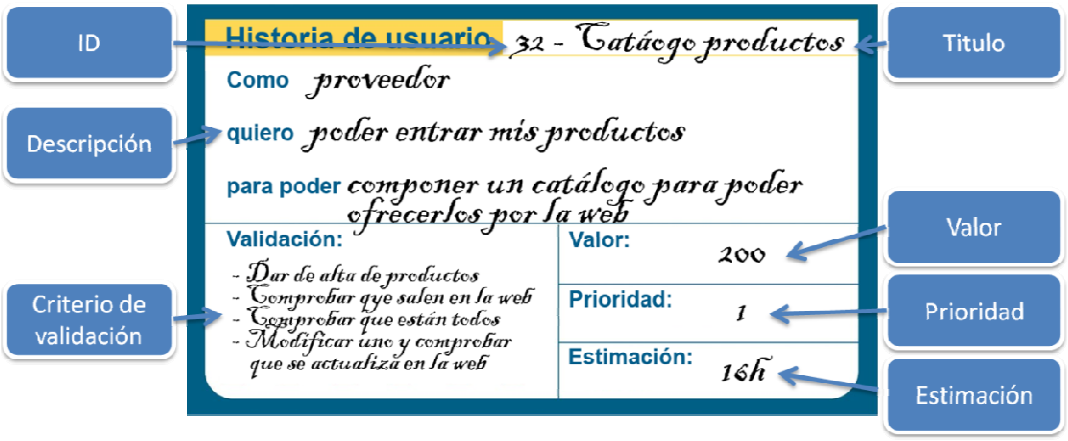
Conclusiones

Como se pudo ver, son muy diversos los ciclos de vida de un software. Sin embargo al final, una de las cosas con las cuales deberás contar, es con el sentido de adaptación, pues regularmente sabemos que al desarrollar un software, difícilmente nuestros jefes tendrán un enfoque o método determinado para trabajar. **Es por eso que manejar un modelo de programación extrema y acostumbrarse es una excelente alternativa**, de esta forma se estará dominando cada una de las metodologías del ciclo de software de antaño y se podrá afrontar cualquier situación complicada en la cuál se pueda encontrar.



Historias de Usuarios

- Los criterios de Mike Cohn para escribir buenas historias de usuario (Como [rol del usuario], quiero [objetivo], para poder [beneficio])
- El modelo INVEST, para evaluar la calidad de una historia de usuario viendo si cumple cada una de las características:
    a. Independent (Independiente)
    b. Negotiable (Negociable)
    c. Valuable (Valiosa para el usuario)
    d. Estimable (Estimable)
    e. Small (Pequeña)
    f. Testeable (Comprobable)



Historia de Usuario	
Número: 1	**Usuario:** Cliente
Nombre historia: Cambiar dirección de envío	
Prioridad en negocio: Alta	**Riesgo en desarrollo:** Bajo
Puntos estimados: 2	**Iteración asignada:** 1
Programador responsable: José Pérez	
Descripción: Como cliente quiero cambiar la dirección de envío de un pedido para que me pueda llegar a casa o a la oficina	
Validación: El cliente puede cambiar la dirección de entrega de cualquiera de los pedidos que tiene pendiente de envío	

Planning Poker

La técnica de Planning Poker, es una práctica ágil ideada por James Grenning, cuyo propósito es estimar el esfuerzo y duración de las tareas en cada Sprint.

En un primer momento, el modelo inicial consta de 8 cartas donde cada participante consta de un juego de cartas. De forma individual cada participante estima el esfuerzo que puede demorar una tarea, luego todos muestran su carta boca arriba y se suma el esfuerzo estimado.

Cuando el esfuerzo es considerado demasiado extenso, se levanta la carta "∞". Estas tareas que exceden el tamaño máximo deben descomponerse en subtareas de menor tamaño.

Cada equipo puede utilizar un juego de cartas con las numeraciones adecuadas a la unidad de esfuerzo con la que trabajan, y el tamaño máximo de tarea o historia que se va a estimar.

Variante: Sucesión De Fibonacci

Esta variante basada en la sucesión de Fibonacci, comienza con los números 0 y 1, y a partir de estos, cada término es la suma de los dos anteriores.

Esta técnica es utilizada, basada en el hecho de que al aumentar el tamaño de las tareas, aumenta también la incertidumbre y el margen de error. Importante conocer que la estimación no se realiza levantando varias cartas para componer la cifra exacta, sino poniendo boca arriba la carta con la cifra más aproximada a la estimación.

Los números se representan como la siguiente figura:

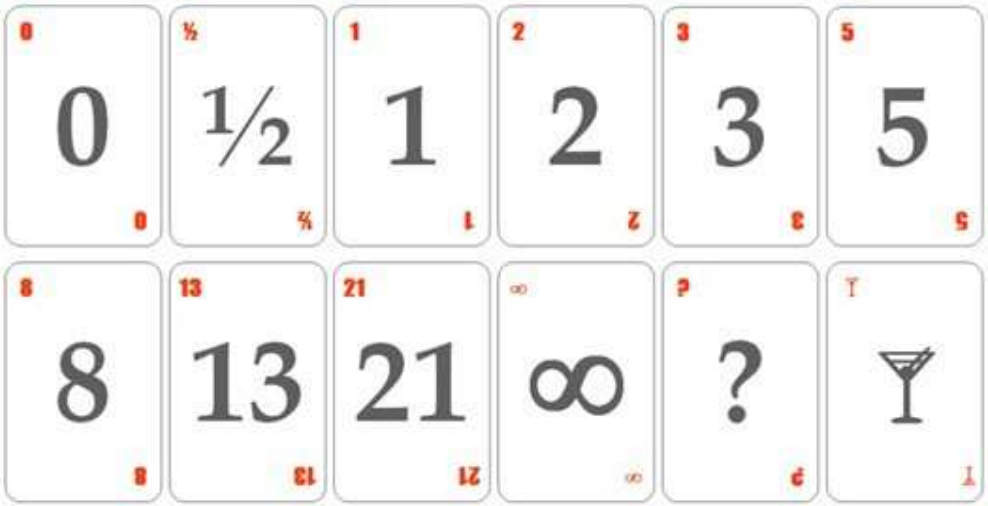


Figura. Cartas del planning poker (Fibonacci). Tomado de Scrum Manager

Es frecuente que se emplee una carta con un símbolo de duda o interrogación para indicar que por alguna razón no se puede precisar una estimación. Se puede incluir también una carta con alguna imagen que indique que se necesita un descanso

Modo de actuación:

Para cada tarea (historia de usuario o funcionalidad) el Product Owner expone la descripción empleando un tiempo máximo.

1. Se establece otro tiempo para que el Product Owner clare las posibles dudas del equipo.
2. Cada participante selecciona la carta, o cartas que representan su estimación, y las separa del resto, boca abajo.
3. Cuando todos han hecho su selección, se muestran boca arriba.
4. Si la estimación resulta "infinito", por sobrepasar el límite máximo establecido, la tarea debe dividirse en subtareas de menor tamaño.
5. Si las estimaciones resultan muy dispares, se puede optar por las siguientes interrogantes:

    - ¿Por qué es necesario tanto tiempo? y ¿por qué es necesario tan poco tiempo? Tras escuchar las razones, se repite la estimación.
    - Dejar a un lado la estimación de esa tarea y retomar al final o en otro momento aquellas que hayan quedado pendientes.
    - Se puede solicitar al Product Owner un desglose la funcionalidad y valorar cada una de las funcionalidades resultantes.

Este protocolo, evita en la reunión atascos de análisis circulares entre diversas opciones de implementación, hace participar a todos los asistentes, reduce el cuarto de hora o la media hora de tiempo de estimación de una funcionalidad, consigue alcanzar consensos sin discusiones, resulta divertido y dinamiza la reunión de planificación.

Lubaris Info 4 Media S.L. (2014). Gestión de Proyectos Scrum Manager. Obtenido de http://www.scrummanager.net

Daily Scrum

Para garantizar que su equipo no se sienta frustrado por sus scrums diarios, aquí hay seis consejos breves para hacer que los scrums diarios sean más útiles con menos divagaciones

Limite la discusión a lo que se logró y se logrará. En lugar de pedirles a los participantes que describan lo que hicieron ayer y lo que harán hoy, pregunte qué lograron y lograrán.

Solo hable sobre el trabajo del sprint actual y en la preparación del próximo sprint. La discusión debe limitarse a los miembros del equipo de trabajo realizados en pos de la meta de sprint del equipo.

Más allá de eso, permito un poco de discusión sobre el trabajo de preparación para el próximo sprint. Por ejemplo, "Divido las historias para el próximo sprint y creo que todas son lo suficientemente pequeñas ahora. Además, programé tiempo con la señorita Importante para que podamos reunirnos con su próximo sprint ".

Déle a la gente algo que decir sobre su trabajo no dirigido hacia la meta del sprint. A veces, los miembros del equipo han pasado una parte significativa del día anterior en un trabajo que no está relacionado con el objetivo del sprint. Quizás un gerente los necesitaba en un proyecto especial.

Puede ser comprensiblemente tentador describir este trabajo. Pero las pautas que he presentado no lo permiten, ya que el trabajo no estaba relacionado con el objetivo del sprint.

Proporcione a los miembros del equipo una frase para usar cuando trabajen fuera del trabajo planificado del sprint. Me gusta la simple frase "otras cosas". Permitir esto le permite a la persona declarar que hizo más de lo que su trabajo relacionado con los objetivos puede indicar. Esto puede ayudarlos a sentirse mejor. Esencialmente no lleva tiempo y puede decirle a un Scrum Master si alguien está trabajando con demasiada frecuencia en trabajos que no son de sprint.

Haz que sea doloroso (literalmente) divagar demasiado. Tomé este consejo de Kayleigh, una gran Scrum Master para su equipo. Ella compró una pequeña pelota medicinal. Dos o tres kilogramos (4.5 a 6.5 libras) funcionan bien. Mientras realiza una actualización, cada persona sostiene la pelota directamente frente a ellos. Para la mayoría de nosotros, ese peso es suficiente para mantenernos conscientes cuando continuamos demasiado tiempo.

Brinde a los miembros del equipo una forma de indicar cuándo alguien está divagando. Para los casos en que los oradores no se dan cuenta de que duran demasiado, den a los miembros del equipo una forma de indicar que la persona ha hablado demasiado. He visto a equipos usar zumbadores, sostener ratas de goma (para indicar que alguien está "bajando por un agujero de rata"), o incluso usar una muñeca Elmo, que indica "Suficiente, sigamos adelante".

Mantenga a las personas adivinando quién hablará a continuación. Cuando sabemos que nos tocará hablar a continuación en una reunión, muchos de nosotros desconectamos al orador actual mientras preparamos mentalmente lo que diremos.

Para evitar que esto suceda en sus scrums diarios, haga que el orador actual llame al próximo orador. Tenga una regla que todos no puedan simplemente llamar a la persona a su lado. Agregue un poco de diversión haciendo que una persona "pierda un punto" si llama a alguien que ya ha dado su actualización.

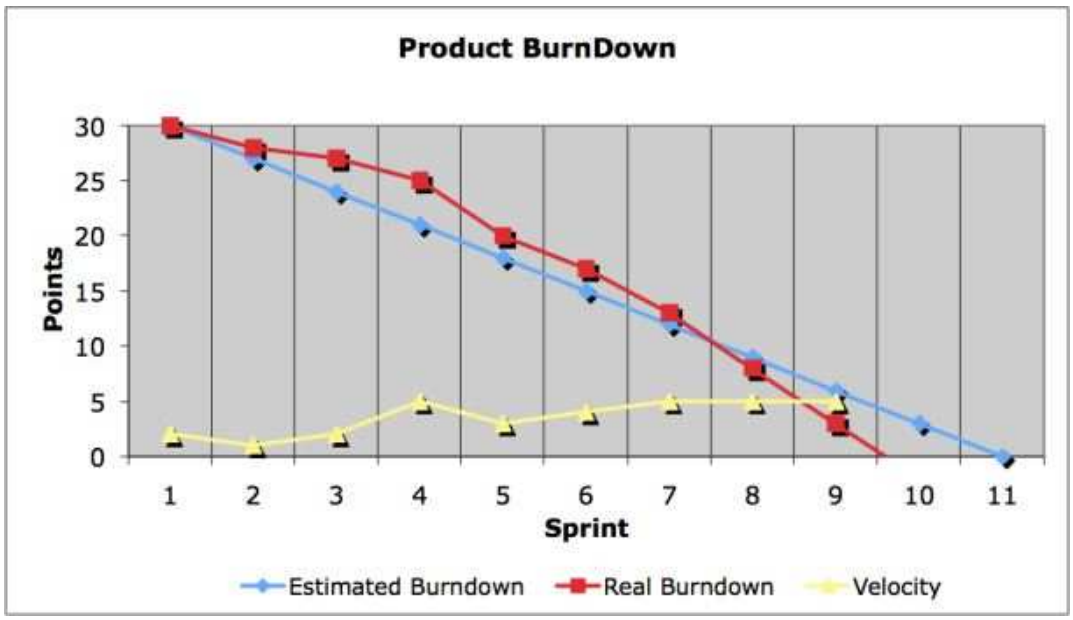
Agregue cosas como si una persona pierde tres puntos, tiene que contar una broma o presentar la retrospectiva. Las posibilidades son infinitas.

Mantener sus reuniones scrum diarias útiles es una forma segura de tener éxito con ágil.

Scrum Burndown Chart (Gráficos de trabajo pendiente)

Un gráfico de trabajo pendiente a lo largo del tiempo muestra la velocidad a la que se está completando los objetivos/requisitos. Permite extrapolar si el Equipo podrá completar el trabajo en el tiempo estimado.

El Scrum Burndown Chart es una herramienta de medición visual que muestra el trabajo completado por día frente a la tasa de finalización proyectada para el lanzamiento actual del proyecto. Su propósito es permitir que el proyecto esté en camino de entregar la solución esperada dentro del cronograma deseado.

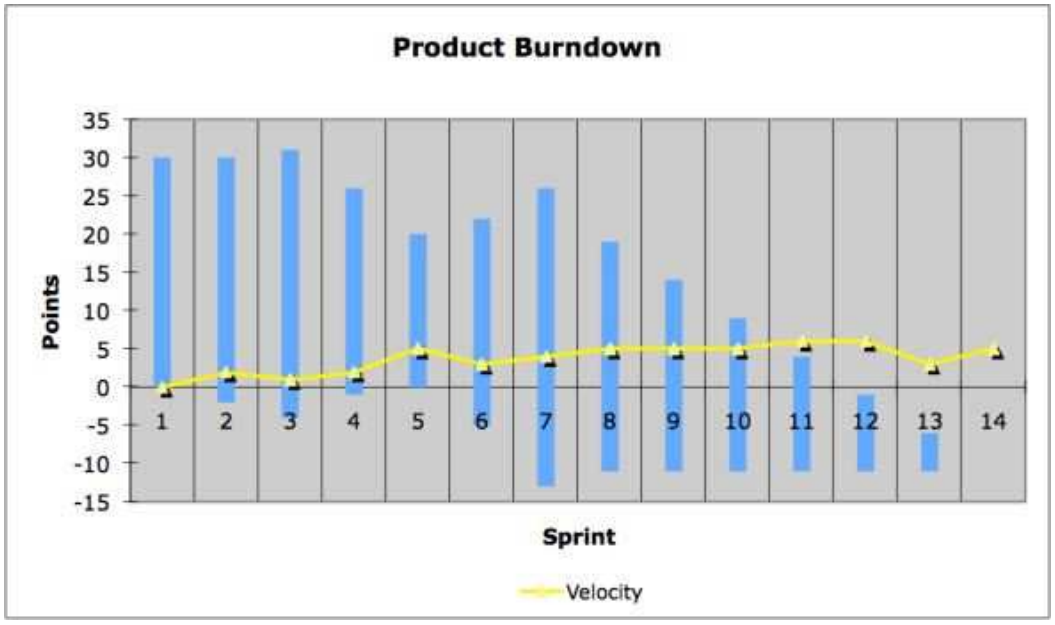


Simple Burndown Chart

La tasa de progreso de un equipo Scrum se llama "velocidad". Expresa la cantidad de puntos de historia completados por iteración. Una regla de importación para calcular la velocidad es que solo se cuentan las historias que se completan al final de la iteración. Se prohíbe estrictamente contar el trabajo parcialmente terminado (p. Ej., Solo codificación; falta la prueba).

Después de algunos Sprints, la velcidad de un Equipo Scrum probablemente será predecible y permitiría una estimación bastante precisa sobre el tiempo necesario hasta que se completen todas las entradas en el Backlog del Producto Scrum. Si la velocidad de un equipo Scrum es 30 puntos de historia y la cantidad total de trabajo restante es 155, podemos predecir que necesitamos alrededor de 6 Sprint para completar todas las historias en el Backlog.

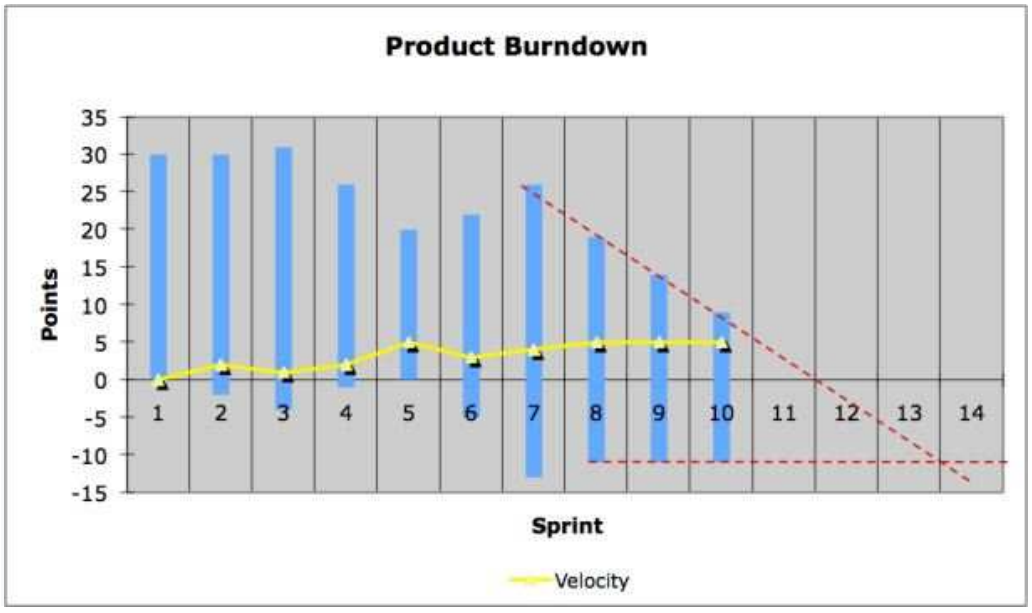
Sin embargo, en realidad, las entradas en el Backlog del producto Scrum cambiarán durante la duración del proyecto. Se agregan nuevas historias y otras historias se cambian o incluso se eliminan. En la tabla de Burndown simple, la velocidad del equipo Scrum y el cambio en el alcance no se pueden distinguir. Para reflejar esto, se puede usar otra forma de diagrama.



Separación de cambios de velocidad y alcance

Aquí se usa un gráfico de barras en lugar de un diagrama lineal. El tamaño de cada barra representa la cantidad total de trabajo restante al comienzo de cada sprint. La velocidad del equipo se resta de la parte superior, mientras que los cambios en el alcance cambian la parte inferior de la barra.

Para ser aún más precisos, también podemos tener en cuenta la tasa de cambios en el trabajo total. Sin embargo, debemos tener cuidado al usar este modelo, ya que la tasa de cambio será alta al comienzo del proyecto, pero caerá al final.

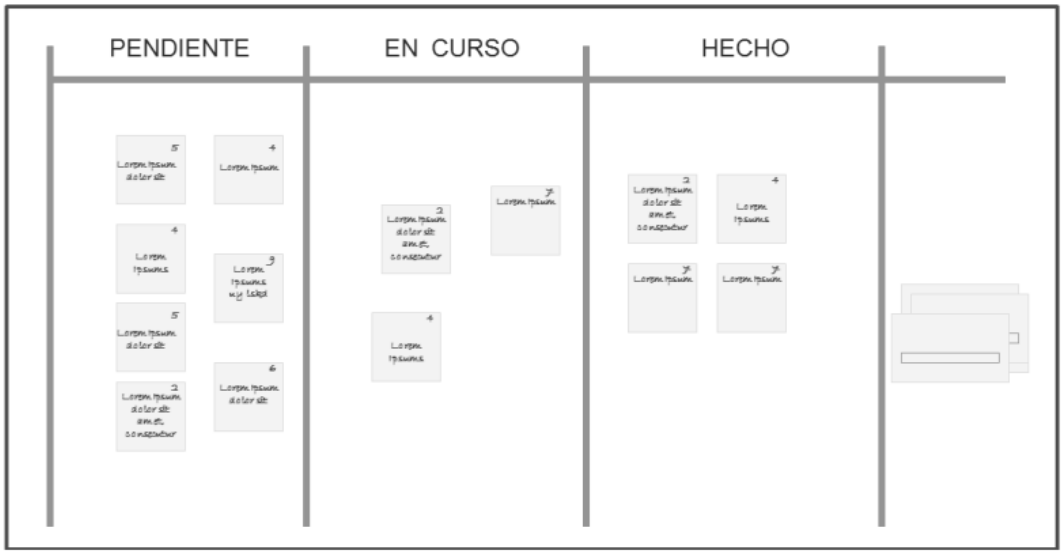


Extended Burndown Chart with Prediction

Qué es un tablero Kanban

Un tablero Kanban es un tablero en el cual vamos a tener una serie de tarjetas, cada una de ellas corresponderá a una tarea dentro de un proyecto software, y vamos a tener una serie de posiciones por las que las tarjetas se van a ir moviendo. Esto nos va a permitir, con una gestión muy visual, ver cómo evoluciona el proyecto.

Este sistema fue inventado por el señor Taiichi Ohno, de Toyota.



Scrum vs Kanban

En muchos casos se intenta diferenciar Kanban de lo que es la metodología Scrum en sí, por lo que vamos a ver algunas de las diferencias que existen entre ambas:

1. En Scrum existen los roles de Scrum Master, de Product Owner y del equipo, mientras que Kanban no existen roles.
2. En Scrum se trabaja con iteraciones de tiempo fijo, con unos ciclos fijos que se denominan Sprints, mientras que en Kanban tenemos un trabajo continuo y no tenemos esas interacciones o esos ciclos durante el desarrollo.
3. Scrum limita el WIP (work in process o número de tareas que se pueden tener en paralelo en una de las posiciones del tablero) por iteración, mientras que Kanban limitar ese WIP por el estado del flujo de trabajo.
4. Mientras que Scrum exige equipos multidisciplinares, para que todos los miembros del equipo puedan realizar varias tareas y sea todo más ágil, en Kanban se permiten los

equipos formados por especialistas. En este caso podemos tener algún problema a la hora de gestionar esos equipos, pero existen una serie de normas o de prácticas para llevar a cabo para solucionarlo.

5. En Scrum no se permiten cambiar las tareas del Sprint, es decir, una vez que la tarea asignada al mismo no puede ser movida, en todo caso lo que se permite es modificar la fecha de entrega del Sprint, pero no esa tarea. En Kanban, por el contrario, se puede modificar la tarea hasta que entra en flujo, hasta ese momento podemos modificar la tarea.
6. En Scrum la pila del producto, es decir, el conjunto de tareas que tenemos que realizar durante el Sprint, tiene que tener al menos el tamaño de un Sprint, ya que, lógicamente, no podemos tener menos de un Sprint. En Kanban, al tener un ritmo de trabajo continuo, lo que se hace es ir arrastrando las nuevas tareas por el panel hasta que lleguen a su estado final y finalicen. Cualquier nuevo requisito del cliente será una nueva tarjeta o post-it que se añadirá al flujo de entrada y que seguirá su flujo normal hasta la salida.
7. En Scrum se mide todo lo que sea necesario, se miden historias, es decir, cuánto nos va a llevar realizar cada historia de usuario, se mide cuánto tiempo o esfuerzo nos va a llevar realizar cada una de las tareas y se mide también la velocidad del equipo, que es la cantidad de trabajo que hemos realizado dividido por la cantidad de tiempo. En Kanban, como ya tenemos una cierta habilidad de la metodología, no se miden ni tareas ni velocidad.
8. En Scrum se necesita una pila del producto priorizada, porque como el desarrollo completo lo vamos a dividir en distintos Sprints, la necesitamos para que los primeros Sprints se encarguen de realizar las tareas de mayor prioridad. Con lo que vamos a conseguir llevar valor al cliente de una manera mucho más rápida y las tareas con menor prioridad serán realizando en los Sprints posteriores. Esta priorización la hará el Product Owner. En Kanban la historia o tarea es arrastrada directamente desde el cliente, por lo cual no necesita esa priorización.
9. En Scrum se tienen una serie de reuniones y se utilizan una serie de gráficos, como el burn down, en el que podemos ver el avance del proyecto, y el burn up, que sirve para medir la velocidad del equipo. En Kanban no se considera ni ese tipo de reuniones ni de gráficos.
10. En Scrum los tableros se van a resetear al final de cada Sprint, es decir, conforme vamos finalizando el mismo, el tablero queda vacío y comenzamos de nuevo añadir nueva nuevas historias de usuario, las siguientes en prioridad. En Kanban, como vamos a tener un flujo de entrada-salida, conforme las tarjetas van pasando por cada uno de los estados hasta llegar al estado final, cuando llegan a ese estado salen del tablero y se archivan, vamos a tener un flujo continuo.