/ Planeación

“Life is what happens to you while you're busy making other plans”

- John Lennon

Planeación

La planeación de un proyecto es una actividad centrada en completar un resultado dentro de un tiempo y costo determinado satisfaciendo las expectativas del cliente.

Esta guía presupone que los proyectos realizados por sus lectores presentarán características particulares como:

- Serán realizados por un individuo o un equipo pequeño de individuos.
- La dimensión del proyecto es relativamente pequeña y la primera fase de desarrollo es corta.
- El resultado del proyecto se presentará en un mercado competitivo que requiere velocidad y flexibilidad.
- Probablemente se efectuará sin horarios ni espacios bien definidos, utilizando el tiempo libre y localizándose en casa de uno de los colaboradores, la biblioteca de la escuela o el Starbucks más cercano.
- No es un trabajo solicitado por un cliente, sino un esfuerzo particular por colocar un producto en el mercado.
- Sus integrantes cuentan con poca experiencia en la producción comercial de software.
- Se ejecutan con un presupuesto muy limitado.
- Todos los miembros del equipo comparten responsabilidades a un nivel similar.

Distintos proyectos y contextos requieren distintos tipos de planeación: una página web hecha por una persona, no requiere el mismo tipo de planeación que el software tolerante a errores utilizado en aviones y hecho por un equipo de cientos de personas con un presupuesto de millones de dólares. Por lo general, entre mayor sea la complejidad y mayor el número de colaboradores de un proyecto, se requerirá una mayor estructura en su planeación. Por esto, aunque la mayoría de las metodologías comparten algunos principios, no hay "metodología panacéica" aplicable a todos los tipos de proyectos y aún la utilización de "mejores prácticas" no es garantía de éxito[26].

Las consideraciones anteriores apuntan a utilizar nuevas metodologías (como el enfoque ágil) en lugar de las metodologías tradicionales, las cuales están íntimamente vinculadas con los procesos y estructuras del negocio. Estas últimas, fueron diseñadas para equipos de trabajo mucho más grandes o proyectos de mayor duración y complejidad, haciendo muchos de sus procesos y documentación un esfuerzo innecesario para proyectos más pequeños.

Las metodologías ágiles surgieron como respuesta a metodologías que en teoría son efectivas, pero que en la práctica no lo son tanto. En el enfoque ágil, se persigue la mejora continua, encontrar la calidad desde la primera vez, producir únicamente lo que es necesario y enfocarse en el desarrollo del software en lugar de en la extensiva documentación, midiendo el progreso del proyecto en unidades de software funcional.

Scrum es uno de los métodos de desarrollo más populares para el desarrollo ágil y uno muy adecuado para las características de proyectos de este tipo, razones por las cuales será analizado con detalle y sugerido para planear la ejecución de un proyecto.

---

[26] McGuire, Eugene. Software process improvement: concepts and practices. Londres : Idea Group Inc. 1999. Página 19.

Scrum

Scrum extiende el método de desarrollo incremental a algo llamado proceso de control empírico, en donde los ciclos de retroalimentación se convierten en el elemento nuclear. Está inspirado en varias áreas de conocimiento como teoría de complejidad, sistemas dinámicos y la teoría de creación del conocimiento de Nonaka y Takeuchi, adaptándolas al desarrollo de software.

A grandes rasgos, las fases de desarrollo se dividen en pequeñas iteraciones llamadas “sprints”, las cuales duran típicamente entre una y cuatro semanas.

Primero, se identifican y capturan las tareas requeridas en una bitácora que es actualizada, estimada y priorizada al principio de cada una de estas iteraciones. El equipo mantiene breves juntas durante cada día de desarrollo para discutir el progreso, plan y potenciales problemas del proyecto y se trabaja en las tareas definidas en la bitácora. Finalmente al término del sprint, se demuestran y recapitulan los avances obtenidos y se inicia uno nuevo para repetir el ciclo hasta haber terminado la totalidad del proyecto.

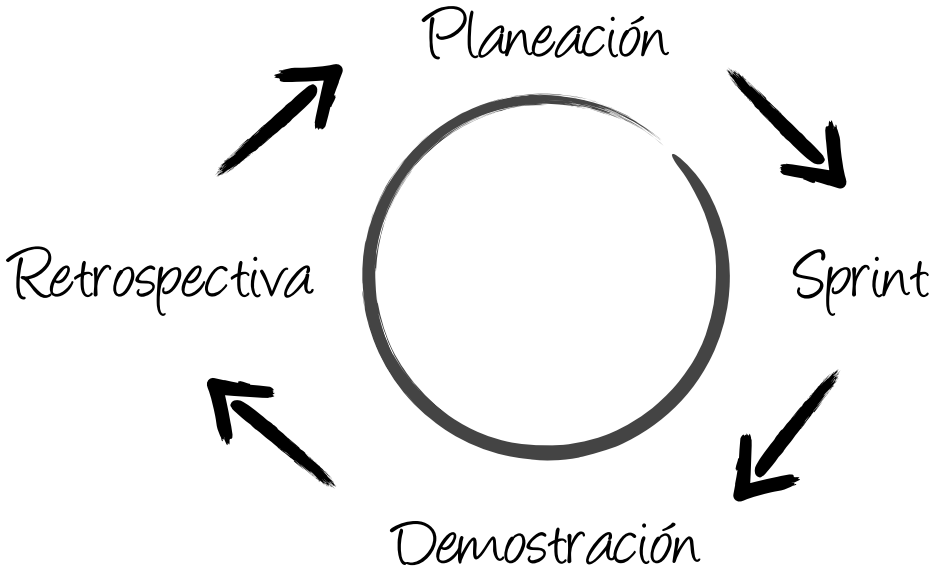


Figura 2: *El ciclo scrum*

Antes de explicar a detalle cada una de estas fases, hay que conocer los roles principales que existen en ellas:

Roles Importantes dentro de Scrum

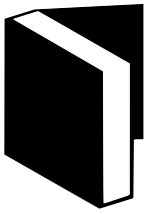
- **Equipo Scrum:**
  Trabajan efectivamente en los problemas del software. Se sugiere como máximo 9 personas. Los miembros del equipo deben decidir cómo se distribuye el trabajo y cómo se define.

- **Product Owner**
  Representa la voz del cliente y administra la bitácora del producto, una lista de tareas priorizada por su rentabilidad hacia el negocio. Por lo general se trata de un cliente, pero también puede ser alguien que sea parte de la organización. En un enfoque completo, su labor requiere conocimiento de procesos de ingeniería, marketing y negocios.

- **Scrum Master**
  Scrum Master es un nombre más divertido para quien en la normalidad sería conocido como “Project Manager”. Su tarea es la de poner al equipo de trabajo en las mejores circunstancias posibles y, después de cada iteración, hacer una retrospectiva para revisar las conclusiones y experiencias con el fin de motivar y mejorar el conocimiento del equipo para atacar la siguiente iteración.

El Ciclo Scrum

1. Planeación y Bitácora del Producto



En el primer sprint del proyecto, el "Product Owner" compila todos los requerimientos y especificaciones de un nuevo producto para plasmarlas en algo llamado "historias", las cuales son una descripción de funcionalidades de alto nivel escritas de forma breve. En el caso de planear la actualización de un producto se escribirían nuevas funciones y parches. Después de esto y al principio de los siguientes sprints, se seguirán los siguientes pasos:

A. Cada "historia" que no ha sido implementada, es puesta en la bitácora del producto y el rol de "Product Owner" les asigna prioridades.

B. El equipo asigna una valoración de complejidad para cada historia (Ej: 1, 2, 3, 5, 8), se estiman cuántas historias pueden ser realizadas durante una iteración y el equipo se compromete a hacer su mejor esfuerzo para terminar todo el trabajo propuesto durante el siguiente sprint.

2. Iteraciones (Sprints)

A. Todas las historias son implementadas progresivamente en orden de importancia, porque son las que representan más valor desde la perspectiva del negocio. Cada equipo (o individuo) es responsable de cumplir las historias que ha escogido.

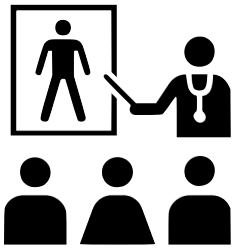
3. Juntas Diarias (¡pfff!)

Durante cada día de desarrollo, el Scrum Master se reúne con el equipo de trabajo para responder, en una junta breve, tres preguntas:

- ¿Qué avances se han hecho desde la última junta?
- ¿Qué va a hacerse desde ese punto hasta la siguiente junta?
- ¿Hay algún conflicto o situación que impida llevar a cabo el plan?

De esta forma se logra que todo el equipo esté informado acerca del progreso del proyecto y que los problemas que se han encontrado, sean resueltos.

4. Demostraciones

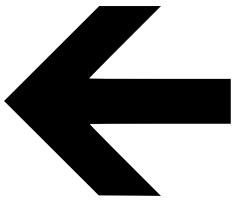


Al final de cada iteración se hace una demostración de todo el software funcional que se haya realizado: todas las historias son demostradas ante el rol de “Product Owner”, quien las aceptará o rechazará.

Como esta metodología está basada en la iteración de componentes de software funcional, en este punto las historias deberán operar en su totalidad y estar libres de errores. El último apartado de este capítulo, menciona la tarea de verificación y validación.

Buena Idea: Aún mejor es presentar las historias terminadas apenas se hayan completado, evitando una historia rechazada que se traslapa a la siguiente iteración.

5. Retrospectiva



Tras la demostración, los miembros del equipo y su juicioso Scrum Master responden las siguientes preguntas:

- ¿Qué se llevó a cabo bien?
- ¿Qué puede ser mejorado?
- ¿Qué acciones pueden tomarse para mantener una alta calidad en el futuro trabajo?

Realimentando la siguiente iteración con los resultados aplicables.

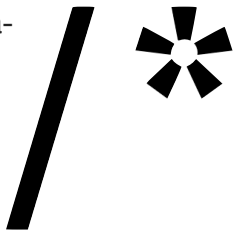
Algunas Buenas Prácticas para Cada Iteración

Claridad en el Código

Algunos programadores escriben código conciso, complicado e indescifrable para denotar su basto conocimiento acerca de la sintaxis del lenguaje y sus trucos más recónditos. Estos intentos desesperados por llamar la atención deben ser sustituidos por la escritura de un código claro, simple y entendible en el que distintas personas con distinta experiencia pueden trabajar fácilmente.

Un código entendible es más fácil de mantener; y durante su manutención, se reduce el riesgo de introducir nuevos errores porque su lógica no sea clara.

Comentar el Código



Los comentarios[27] son notas dentro del mismo código del programa que señalan pasos y características claves del mismo y que están escritas de tal forma que la computadora pueda ignorarlas para la ejecución del programa.

Esta práctica genera un tipo de documentación interna que, implementada correctamente, permite entender más fácilmente el código en cuestión y esto a su vez facilita el mantenimiento y actualización del proyecto, así como la re-utilización de código mediante la legendaria práctica del *copy-paste.*

Tampoco hay necesidad de caer en una sobre-explicación innecesaria: los comentarios son empleados cuando el código no puede ser más claro mediante otros medios.

---

[27] Morley, Deborah. Understanding Computers: Today and Tomorrow, Comprehensive. Estados Unidos: Cengage Learning. 2009. Páginas 561-562.

Buena Idea: Escribir los comentarios del código al mismo tiempo que se codifica. Escribirlos tiempo después de codificar es más complicado ya que probablemente requerirá re-entender el código.

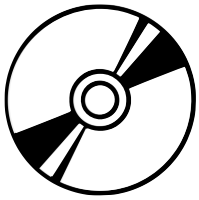
Wikis

Un wiki[28] es un software en línea que permite a todos sus visitantes cambiar su contenido mediante la edición de las páginas dentro de un explorador; convirtiéndola en una plataforma simple y fácil de usar para el trabajo colectivo de textos e hipertextos.

Un wiki privado para los participantes del proyecto puede utilizarse para escribir toda la documentación referente al proyecto: bitácoras, resultados de pruebas de verificación y validación, tareas a completar, etc.

En la red existen varias alternativas disponibles y la mayoría de ellas son gratuitas[29].

Respaldos de Seguridad



Discos duros dañados, archivos corruptos, virus, empleados vengativos borrando archivos sensibles, laptops hurtadas a manos de bandidos sin escrúpulos: parecería que la posibilidad de perder todo el trabajo invertido en una aplicación es ineludible. Aunque ciertamente no lo es, sí vale la pena tener un plan en que, ante un infortunio como los antes mencionados, permitiera reanudar el trabajo sobre el proyecto de manera rápida, sencilla y económica (en términos de tiempo y dinero) y sin tener que repetir ningún esfuerzo.

---

[28] Ebersbach, Anja. Wiki: Web Collaboration. Berlin: Springer. 2008. Pagina 12.

[29] :)

Por fortuna, un proyecto para una plataforma móvil tenderá a ser un conjunto de archivos y recursos de tamaño relativamente pequeño y que fácilmente podría respaldarse sobre un CD, DVD, Disco Duro, memoria USB o un servidor remoto. Además de los recursos y archivos del proyecto mismo, es útil hacer copias de seguridad de la documentación del proyecto.

Se exponen algunos principios básicos:

- **Ordenar la información**

  backup.tar.gz o *backup2.tar.gz* no son títulos muy descriptivos. Un buen orden y una buena denominación de los archivos de respaldo, hacen una recuperación de datos mucho más simple.

- **Distintas Locaciones**

  Lejos de detallar ejemplos como huracanes u otras catástrofes remotas , existe el caso mucho más cotidiano de que un ladrón entrara a una casa u oficina. Además de llevarse todas las computadoras del lugar, es probable que decidiera equiparse también con accesorios como discos duros y memorias, convirtiera cualquier práctica del respaldo de información en un esfuerzo absolutamente fútil.

Control de Versiones

Los sistemas de control de versiones (CVS) pueden considerarse como parte de las estrategias de backups y respaldos de seguridad, aunque incluye mayores alcances y beneficios. El mercado oferta un sinnúmero de posibilidades, varias de ellas gratuitas, cada una con una comunidad de adeptos y fanáticos que defienden religiosamente las virtudes de cada sistema y se ocupan en señalar las faltas y deficiencias de aquellos que no utilizan[30]. Aunque existen diferencias funcionales y filosóficas entre cada sistema, todos los CVS deberían poder hacer lo siguiente[31]:

- Mantener y permitir el desarrollo de un repositorio de contenido.
- Llevar un registro de todos los cambios hechos sobre cada recurso del repositorio.
- Proveer acceso al historial de ediciones de cada uno de estos recursos.

---

[30] Un vistazo fugaz a este recurso ejemplifica claramente el punto: http://atomized.org/2005/09/subversion-sucks/

[31] Loeliger, Jon. Version Control with Git. Estados Unidos: O'Reilly Media, Inc. 2009. Página 1.

Tomando en cuenta las consideraciones enunciadas al principio del capítulo, la elección del Sistema de Control de Versiones queda prácticamente reducida a la comodidad que pudiera significar un sistema sobre otro: como si el equipo tiene experiencia con algún sistema particular o si el CVS esta integrado al Entorno de Desarrollo (IDE) y supondría un uso mucho más simple del mismo.

Tips de Verificación y Validación para cada iteración

Como se mencionó anteriormente los componentes completados al término de cada iteración son componentes funcionales y libres de *bugs*. Lo que implica que deben haber sido probados y se ha verificado que su comportamiento es exactamente el esperado.

En este tipo de aplicaciones la verificación y validación del programa tiene un objetivo muy concreto: buscar la satisfacción del cliente por la compra que ha hecho. Los compradores de software son cada vez más intolerantes a productos con errores y ahora tienen altas expectativas que de no ser cumplidas, resultan en comentarios negativos en la tienda, o peor aún, en medios como blogs o redes sociales que podrían impactar negativamente en las ventas y percepción del producto.

Estas demostraciones justificadas de frustración son causadas principalmente por dos causas: un desempeño deficiente y falta de *usabilidad*. El equipo de desarrollo debe invertir todo esfuerzo necesario para eliminar o reducir de forma drástica cualquiera de estas deficiencias.

Debe considerarse que un producto final distribuido en una tienda, será puesto a prueba por un gran número de usuarios en un amplio conjunto de posibilidades impredecibles acerca su uso. Entre mayor sea su uso, mayores las probabilidades de que bugs “ocultos” o difíciles de encontrar sean, de hecho, encontrados.

Existe un sinnúmero de técnicas y procedimientos detallados para probar código, pero este trabajo se acotará a señalar algunas estrategias que den una perspectiva general y práctica de la calidad del producto y sus componentes:

Probar la aplicación en dispositivos físicos

Probar la aplicación permite verificar dos aspectos importantes de la aplicación:

- Primero, que la interfaz sea funcional (tamaño y disposición de los botones y texto) en un entorno real.

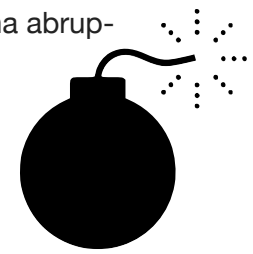
- Segundo, los simuladores no representan fidedignamente el desempeño de los dispositivos, uso de memoria, conectividad, sistema operativo y otros recursos de hardware. Puede haber diferencias importantes en la velocidad del programa o incluso no funcionar del todo. Acelerómetros o servicios de posicionamiento tampoco están disponibles en los simuladores.

Usuarios ßeta

El equipo de desarrollo (o desarrollador) están demasiado involucrados y conocen demasiado el producto para detectar todos los *bugs* por sí mismos. Solicitar algunos usuarios beta a través de redes sociales o en el círculo de amigos de uno mismo, permite encontrar bugs difíciles de encontrar y hacer mejoras en su usabilidad. Como una ventaja adicional se tiene acceso a nuevas perspectivas para descubrir nuevos segmentos de mercado y nuevas funcionalidades.

Errores Fatales

No hay frustración más grande para el usuario que una aplicación que termina abrupta e inexplicablemente.



Si este tipo de errores son detectados, la aplicación está todavía lejos de poder llegar al mercado.

Buena Idea: Errores fatales que se presentan de maneras misteriosas, ocurren comúnmente debido a problemas con la disponibilidad de memoria.

Verificar la interfaz de usuario

La interfaz del usuario debe ser obvia y consistente. En el capítulo “Interfaces de Usuario” se sugiere una metodología simple, pero valiosa para la verificación de la interfaz de usuario. Ampliamente recomendable. ;)

Verificar problemas de conectividad

Si la aplicación depende de conectividad con la red para funcionar, deben probarse casos en donde ésta sea nula o deficiente. La aplicación debe ser tolerante a este tipo de “eventualidades” no tan eventuales en los móviles.

Buena Idea: Poner el dispositivo dentro de una caja de zapatos forrada de papel aluminio[32] permite probar un enlace degradado de Wi-Fi o de datos; además de dar al desarrollador un romántico aire científico-experimental.

[32] Scholz, Fritz. Electroanalytical Methods: Guide to Experiments and Applications. Berlin: Springer. 2010. Página 335.