Módulo 3: Desarrollo ágil

Diseño y Desarrollo de Software (1er. Cuat. 2019)

Profesora titular de la cátedra:
Marcela Capobianco

Profesores interinos:
Sebastián Gottifredi
Gerardo I. Simari

Licenciatura en Ciencias de la Computación – UNS

Licencia

Desarrollo Ágil

Muchas vertientes diferentes de ideas parecidas

- *Extreme Programming* (XP) nace en los ‘90.
- La metodología *Scrum* desarrollada por Ken Schwaber y Jeff Sutherland.
- En el año 2001 se reunió un grupo de gente preponderante en el área y escribieron el Manifiesto del Desarrollo Ágil.

Base del Desarrollo Ágil

- Usar reglas suaves pero suficientes.
- Tener en cuenta el factor humano.
- Usar reglas para manejar la comunicación (fundamental).
- La agilidad implica capacidad de *maniobrar* a tiempo, respondiendo rápidamente ante el cambio.

Juego cooperativo

- Se piensa en el desarrollo de software como un “juego cooperativo”.
- Las metodologías pueden en general ser vistas como *convenciones sobre coordinación*.
- Otro posible acercamiento (expresado en *Adaptive Software Development*) es pensar al SW como un sistema adaptativo complejo.

Problema: Comunicación

- La comunicación perfecta es imposible.
- *Objetivo*: aprender a manejar la “incompletitud de la comunicación”.
- Cuanto más diferente es la persona, menor es el paso que puede dar para entendernos.
- Hay que explicar desde lo más básico.

Problema: *Comunicación*

- Cuando escribimos los requerimientos, lo hacemos como si pudiéramos hacerlo en forma *completa* y *correcta*.
- Tenemos que determinar el nivel de experiencia del lector a quien va dirigido (a mayor experiencia, se puede escribir menos).
- Las diferencias *culturales* (general y de disciplina) pueden causar grandes problemas.

El Manifiesto Ágil

Objetivo: ver qué hay en común entre varias "*metodologías lightweight*":

- Adaptive SW
- XP
- Scrum
- Crystal
- Pragmatic programming...

El Manifiesto Ágil

- Nadie estaba de acuerdo con mezclar las metodologías para crear una sola.
- Se identificó la necesidad de *responder al cambio*.
- Se identificaron 4 valores clave.
- No se consiguió consenso con respecto a las *prácticas* detalladas.

El Manifiesto Ágil

We are unconvering better ways of developing SW by doing it and helping others do it... We have come to value:

- *Individuals and interactions over processes and tools*
- *Working software over comprehensive documentation*
- *Customer collaboration over contract negotiation*
- *Responding to change over following a plan*

Reflexiones

- No incluye la necesidad de trabajar en forma distinta en situaciones distintas.
- No es lo mismo un proyecto de 10 personas que un proyecto de 100.

Principios del manifiesto (1)

- La prioridad es satisfacer al cliente con "*early and continuous delivery of valuable software*".
- Dar la bienvenida a los cambios, aún en estado avanzado del desarrollo. Permiten la ventaja competitiva.
- Entregas sucesivas, desde un par de semanas a un par de meses, con preferencia hacia períodos cortos.

Principios del manifiesto (2)

- Los clientes y los desarrolladores deben trabajar juntos a diario.
- Darles a las personas el ambiente, la confianza y el apoyo que necesitan para realizar el trabajo (*motivación*).
- Favorecer la conversación cara a cara en el equipo para tener efectividad y eficiencia.

Principios del manifiesto (3)

- El SW en funcionamiento es la medida primaria del progreso.
- Los procesos ágiles promueven el desarrollo "sustentable":

  Los sponsors, desarrolladores y usuarios deben ser capaces de mantener un paso constante por tiempo indefinido.

Principios del manifiesto (4)

- Atención continua a la excelencia técnica y buen diseño.
- La simplicidad (*maximizar el trabajo no realizado*) es esencial.
- Las mejores arquitecturas, requerimientos y diseños surgen de equipos *auto-organizados*.
- En intervalos regulares, el equipo reflexiona sobre cómo volverse más efectivo y realiza *ajustes*.

Video sobre metodologías ágiles

What is Agile?

`https://www.youtube.com/watch?v=Z9QbYZh1YXY`

Scrum

- Es un modelo de desarrollo ágil, una metodología de *gestión del trabajo*.
- Los proyectos progresan a través de una serie de iteraciones llamadas *sprints.*
- Se basa en el modelo espiral.
- Se van evaluando permanentemente los *riesgos* (por ejemplo, la cancelación del proyecto por superar tiempos o recursos asignados).

Scrum: Orígenes de la idea

"El enfoque de 'carrera de relevos' en el desarrollo de productos ... puede entrar en conflicto con los objetivos de máxima velocidad y flexibilidad. En su lugar, un enfoque holístico o estilo 'rugby' – donde un equipo intenta ir a la distancia como una unidad, pasando la pelota hacia adelante y hacia atrás – pueden servir mejor a los actuales requisitos competitivos."

Hirotaka Takeuchi, Ikujiero Nonaka: "The New Product Development Game", Harvard Business Review, January 1986.

Orígenes de Scrum

- Jeff Sutherland
  - Primeros Scrums en Easel Corporation en el año 1993
  - En IDX Systems Corporation: "*Scrum of Scrums*" con más de 500 personas
- Ken Schwaber
  - ADM: *Advanced Development Methods*
  - Scrum es presentado en OOPSLA '96 junto con Sutherland
- Mike Beedle: *Scrum Patterns* en PLOP
- Ken Schwaber y Mike Cohn fundan la *Scrum Alliance* en 2002.

Algunas organizaciones que han usado Scrum

- IBM
- Yahoo
- Google
- Electronic Arts
- Lockheed Martin
- Philips
- Siemens
- Nokia
- Microsoft
- BBC
- John Deere
- Time Warner
- ...

Scrum

- Para evaluar el avance y los riesgos se usan *prototipos.*
- Se definen *objetivos* para cada una de las iteraciones.
- Permite hacer una buena gestión de los cambios.

¿Cuándo usarlo?

Es una buena metodología cuando:

- No se conoce el dominio de aplicación del problema.
- Desarrolladores y usuarios tienen poca experiencia en el tema.
- Hay falta de precisión sobre los problemas a resolver.
- Los requerimientos son inestables o hay condicionamientos estrictos en cuanto a tiempos y costos.

¿Para qué sirve?

- Situaciones inciertas
- Tecnología desconocida o muy novedosa
- Requerimientos *complejos*
- Requerimientos *cambiantes*

Roles: *Comprometidos* vs. *Interesados*

A pig and a chicken are walking down a road. The chicken looks at the pig and says, "*Hey, why don't we open a restaurant*?" The pig looks back at the chicken and says, "*Good idea, what do you want to call it*?" The chicken thinks about it and says, "*Why don't we call it 'Ham and Eggs'*?" "I don't think so," says the pig, "*I'd be committed but you'd only be involved*.".

Roles

Comprometidos (“pigs”):

Tienen autoridad para hacer lo necesario para que salga adelante el proyecto:

- Product Owner
- Scrum Master
- Equipo de trabajo.

Roles

Interesados (“hens”):

Son espectadores.

No deben interferir innecesariamente.

No tienen autoridad directa sobre el progreso y la ejecución del mismo:

- Resto de los dirigentes de la empresa
- Otros equipos de desarrollo.

Product Owner

- Es alguien que desempeña un rol en Marketing o un “key user” en desarrollos internos.
- Se encarga de priorizar el *Product Backlog*.
- Scrum le da la posibilidad al dueño del producto de ir modificando las especificaciones de trabajo, en respuesta a las condiciones cambiantes del negocio.
- Representa los intereses del cliente.
- Acepta o rechaza decisiones/entregables.

Scrum Master

- Es el responsable de asegurar que el *Scrum Team* se conduzca bajo los valores y prácticas de Scrum.
- Protege al equipo asegurando que no se *sobrecomprometan* en lo que puedan llevar a cabo durante un sprint.
- Facilita el *Daily Scrum* y se encarga de resolver los problemas que son planteados por el equipo durante la reunión.

Scrum Team

- No incluye ninguno de los *roles tradicionales* de la ingeniería del software (como analistas, programadores, diseñadores, etc.).
- Cada integrante trabaja en equipo para completar las tareas que fueron comprometidas en conjunto a ser desarrolladas durante el sprint.
- Se apunta a que el equipo desarrolle un sentimiento de camaradería (“*estamos todos juntos en esto*”).
- El número típico de integrantes va desde 6 a 10.
- También se pueden hacer “*Scrums de Scrums*”.

Scrum Team

- Los miembros deben ser *full time*.
- Puede haber excepciones, como un administrador del bases de datos o un especialista en requerimientos.
- El equipo se auto-organiza.
- Los miembros sólo deben cambiar entre sprints.
- El equipo es *cross-funcional*.

Reunión de planificación

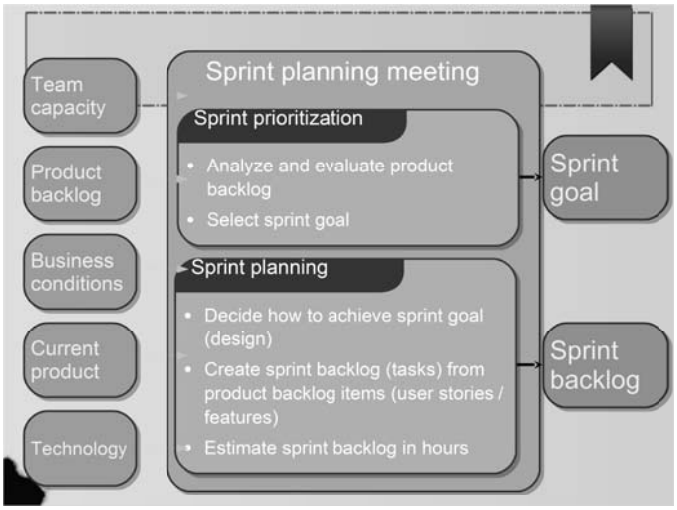
- El proyecto (y cada iteración) empieza con una *reunión de planeamiento.*
- Es llevada a cabo por el Dueño del Producto, el Scrum Master y el equipo.
- *Objetivo*: elegir qué se va a hacer en la siguiente iteración.
- Se divide en dos partes de 4 horas cada una.

Etapas

- En la primera, el Dueño del producto, el Scrum Master y el Equipo eligen los requerimientos entre los existentes en la Especificación del Producto.
- Esto se hace de manera tal que puedan implementarse durante la iteración.
- En la segunda parte, el objetivo es que el equipo planee cómo va a llevar a cabo la iteración actual y arme la *especificación del trabajo.*
- En esta parte es opcional la participación del dueño del proyecto.

Etapas

- La especificación del trabajo puede no estar completa, se irá completando durante la iteración.
- Esta segunda parte ya es considerada parte de la iteración, o sea parte del tiempo especificado para el "sprint".
- Durante cada una de las iteraciones, todos los días, debe realizarse una *Reunión diaria de Scrum* (SDM, en inglés).



Reunión diaria

- La duración debe ser de aproximadamente 15 minutos.
- *Objetivos*:
  - Sincronizar el trabajo diario de todos.
  - Dar visibilidad a cada miembro del equipo.
  - Fomentar la comunicación entre los mismos.

Reunión diaria

El Scrum master va hablando con cada miembro del equipo y éste debe responder:

- *¿Qué hice desde la última reunión diaria de Scrum*?
- *¿Qué es lo que voy a hacer entre esta y la próxima*?
- *¿Qué es lo que está obstaculizando mi trabajo*?

Reunión diaria

- Se hace de pie (es también llamada *Stand Up Daily Meeting*, coincidiendo con la abreviatura SDM).
- Si alguien falta, debe enviar (por ejemplo, mediante un correo electrónico o un representante) la respuesta a las tres preguntas anteriormente mencionadas.

Reunión diaria

- El Scrum Master ordena el debate ante los obstáculos.
- Es el mediador, pero no da órdenes; el equipo se *auto-organiza* para cumplir los objetivos.
- Sólo debe hablar una persona por vez.

Reunión diaria a distancia

- Si el equipo está separado físicamente, se puede implementar la reunión por correo electrónico.
- La idea es escribir las preguntas ya planteadas (o una variación) en cada mail.
- Responder estas preguntas de manera que entienda alguien que no está siguiendo el día a día.
- Importante: dejar esta información en algún repositorio (por ejemplo, un foro) del proyecto.
- Hoy en día se utiliza mucho la videoconferencia.

Reunión diaria

- Puede ser útil agregar las siguientes secciones al trabajar por mail o en foros:
- Dentro de ¿*Qué hice ayer*? (o como sección aparte) marcar de alguna forma los *Logros alcanzados.*
- Agregar un ítem de *Lecciones aprendidas*.
- Agregar un ítem de ¿*Qué tengo pendiente*? para tener una perspectiva del trabajo que está por venir.

Reunión de revisión de la iteración

- Una vez finalizada la iteración, se debe realizar una *Reunión de Revisión de la Iteración*, de aproximadamente 4 horas de duración.
- Es la única reunión en la que no sólo los involucrados sino también los interesados pueden participar y opinar.
- El objetivo principal de esta reunión es que el equipo tenga la oportunidad de mostrarle al dueño del producto y otros interesados la funcionalidad “terminada”.

Reunión de revisión de la iteración

- Se comienza presentando:
  - los *objetivos* de la iteración
  - la especificación del producto con la que se *comprometieron*
  - la especificación del producto *completado*.
- Diferentes miembros del equipo pueden discutir sobre lo que anduvo bien o no durante la iteración (*no deben ser detalles técnicos*).
- Es informal (¡*sin transparencias*!)

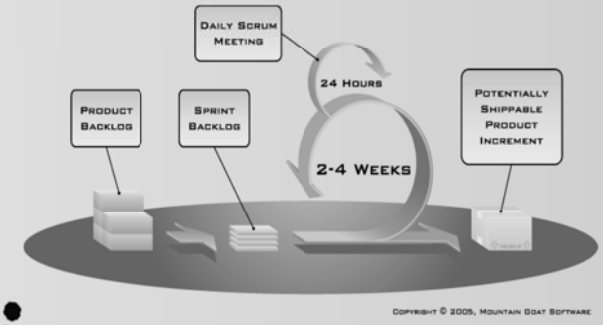
Reunión de revisión de la iteración

- Al final, los interesados del lado del cliente pueden dar cada uno su impresión.
- Entre todos se comienzan a discutir futuros cambios para la especificación del producto basados en las respuestas recibidas.

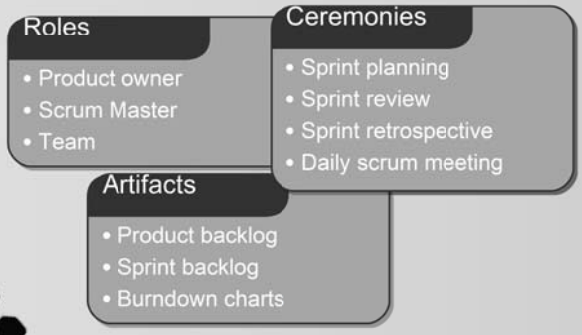
Reunión retrospectiva del Sprint

- En esta reunión, el Scrum Master hace que el equipo haga una revisión de lo hecho tanto a nivel técnico como no.
- *Objetivo*: Mejorar en la siguiente iteración.
- Todo el equipo contesta dos preguntas:
  - *¿Qué anduvo bien durante la iteración*?
  - *¿Qué se puede mejorar para la siguiente iteración*?
- El Scrum Master toma nota de las respuestas.

Resumen



Resumen



Especificación del Producto (*Product Backlog*)

- Es una lista de requerimientos que se encuentra bajo responsabilidad del dueño del producto.
- Este mismo debe ir ingresando los requerimientos deseados priorizándolos según el valor de retorno de la inversión que le den a sus representados.
- Espíritu: La especificación del producto "*nunca está completa*".

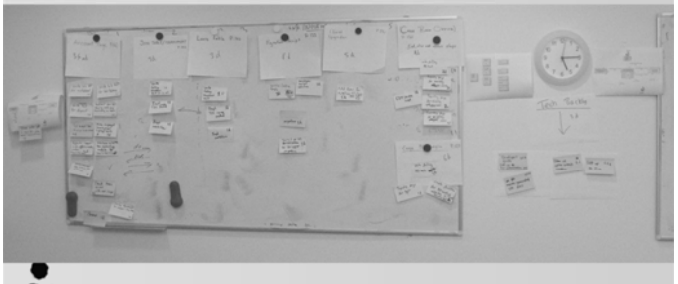
Especificación del Producto (*Product Backlog*)

- La especificación es dinámica y existe mientras haya un producto.
- Si un requerimiento no se termina en una iteración, pasa a la siguiente (o a otra), con mayor o menor prioridad dependiendo de la decisión del dueño del producto.
- A partir del *product backlog* se planifica qué "historias de usuario" se realizarán en el sprint determinado.

Product Backlog: Ejemplo

	Item #	Description	Est	By
Very High				
	1	**Finish database versioning**	16	KH
	2	**Get rid of unneeded shared Java in database**	8	KH
	-	**Add licensing**	-	-
	3	Concurrent user licensing	16	TG
	4	Demo / Eval licensing	16	TG
		Analysis Manager		
	5	File formats we support are out of date	160	TG
	6	Round-trip Analyses	250	MC
High				
	-	**Enforce unique names**	-	-
	7	In main application	24	KH
	8	In import	24	AM
	-	**Admin Program**	-	-
	9	Delete users	4	JM
	-	**Analysis Manager**	-	-
	10	When items are removed from an analysis, they should show up again in the pick list in lower 1/2 of the analysis tab	8	TG
	-	**Query**	-	-
	11	Support for wildcards when searching	16	T&A
	12	Sorting of number attributes to handle negative numbers	16	T&A
	13	Horizontal scrolling	12	T&A
	-	**Population Genetics**	-	-
	14	Frequency Manager	400	T&M
	15	Query Tool	400	T&M
	16	Additional Editors (which ones)	240	T&M
	17	Study Variable Manager	240	T&M
	18	Haplotypes	320	T&M
	19	**Add icons for v1.1 or 2.0**	-	-
	-	**Pedigree Manager**	-	-
	20	Validate Derived kindred	4	KH
Medium				
	-	**Explorer**	-	-
	21	Launch tab synchronization (only show queries/analyses for logged in users)	8	T&A
	22	Delete settings (?)	4	T&A

Historias de usuario



Formato de *user stories*

As who I want
what so that
why

User stories

Videos sobre las historias de usuario y su formato:

Agile user stories:

`https://www.youtube.com/watch?v=apOvF9NVguA`

User stories

Videos sobre las historias de usuario y su formato:

Splitting user stories – Agile practices:

`https://www.youtube.com/watch?v=EDT0HMtDwYI`

Sprint Goal

La *meta del sprint* es una frase corta que indica en qué se focalizará el trabajo durante el sprint.

Por ejemplo:

Database Application

Make the application run on SQL Server in addition to Oracle.

Especificación del Trabajo (*Sprint Backlog* o *Commited Backlog*)

- Aquí se definen los requerimientos que el equipo seleccionó para convertir en un *incremento de funcionalidad*.
- Las tareas deben demandar entre 4 y 16 horas cada una (si no, analizar cómo dividirlas en sub-tareas).
- La especificación del trabajo es una "foto" de lo que el equipo planea tener hecho en esa iteración.
- Es creada en la reunión de planeamiento, y es actualizada constantemente durante la iteración.

A tener en cuenta

- Scrum *no dice nada* acerca de hacer o no hacer diseño, hacer o no hacer test unitarios, hacer o no hacer documentación, etc.
- Scrum únicamente nos indica cómo conseguir:
  - que todos trabajen con el mismo objetivo,
  - a corto plazo, y de manera de
  - dejar visible cómo avanza el proyecto día a día.
- Lo ideal es complementarlo con otra metodología, como la *Programación Extrema* (XP), que veremos a continuación.

Comparación entre *Scrum* y otras metodologías

	Waterfall	Spiral	Iterative	SCRUM
Defined processes	Required	Required	Required	Planning & Closure only
Final product	Determined during planning	Determined during planning	Set during project	Set during project
Project cost	Determined during planning	Partially variable	Set during project	Set during project
Completion date	Determined during planning	Partially variable	Set during project	Set during project
Responsiveness to environment	Planning only	Planning primarily	At end of each iteration	**Throughout**
Team flexibility, creativity	Limited - cookbook approach	Limited - cookbook approach	Limited - cookbook approach	**Unlimited during iterations**
Knowledge transfer	Training prior to project	Training prior to project	Training prior to project	**Teamwork during project**
Probability of success	Low	Medium low	Medium	**High**

Fuente: K. Schwaber: "*Scrum Development Process*", 1997.

Resumen de *Scrum*

En el siguiente video se da un resumen rápido que sirve a modo de repaso del tema:

`https://www.youtube.com/watch?v=XU0llRltyFM`

Extreme Programming

- Extreme Programming (XP) es una metodología liviana (*lightweight*) de desarrollo de software.
- Estas metodologías surgen como contrapartida de las llamadas *heavyweight* o pesadas.

Metodologías *Heavyweight*

We created software to help us create software. But this quickly got out of control and dreadnought CASE tools were born. These tools, originally created to help us follow the rules, are too hard to use themselves. Computer programmers find it necessary to cut corners and skip important practices to stay on schedule. No one is actually following the heavy methodologies we have handcuffed ourselves with…

Don Wells: "Extreme Programming – A Gentle Introduction", 1999.

XP

Extreme Programming (XP) is one of several new lightweight methodologies. XP has a few rules and a modest number of practices, all of which are easy to follow. XP is a clean and concise environment developed by observing what makes software development go faster and what makes it move slower. It is an environment in which programmers feel free to be creative and productive but remain organized and focused.

Don Wells: "Extreme Programming – A Gentle Introduction", 1999.

¿En qué dirección vamos?

- No queremos volver a los años ‘60.
- Queremos prácticas *concisas* y *efectivas*.
- *Extreme Programming* (XP) es una de las metodologías lightweight, entre varias que existen y pueden combinarse.
- Como toda metodología, es una colección de reglas y prácticas.

Extreme Programming

- *Extreme Programming* (XP) es un acercamiento “extremo” al desarrollo iterativo de software.
- Pueden producirse nuevas versiones muy frecuentemente (hasta *varias veces por día*).
- Los incrementos se entregan a los clientes aproximadamente cada dos semanas.
- Todos los tests se deben superarse para *cada versión*: una versión sólo es aceptada si pasa a todos ellos.

Prácticas de XP (1)

- En forma similar a Scrum, las *historias de usuario* son el corazón de la planificación en XP.
- Se crean en tarjetas que se pueden imprimir o hacer a mano (son la version XP de los casos de uso).
- Para diseñar la arquitectura del sistema se usan prototipos conocidos como *metáforas del sistema*.
- *CRC (Class-Responsibility-Collaboration) Cards*, una técnica de diseño “groupware”, alienta a todos los miembros a entender el diseño y contribuir al mismo.

Prácticas de XP (2)

- *Planificación incremental*: los requerimientos se guardan en las story cards, y para elegir cuáles se incluirán en una release se usan prioridades.
- Los desarrolladores dividen estas historias en tareas (muy similar a Scrum).

Prácticas de XP (3)

- *Entregas pequeñas*: se empieza por el mínimo conjunto de funcionalidad que entrega valor de negocio.
- *Diseño simple*: se usa sólo el diseño necesario para satisfacer los requerimientos.
- *Desarrollo basado en testing*: se usan frameworks de testing unitario automático y se apunta a escribir los tests *antes* de implementar la funcionalidad.

Prácticas de XP (4)

- *Refactoring* es una técnica de programación esencial a XP que apunta a descubrir un diseño arquitectónico más efectivo.
- La *calidad* del código es muy importante para un proyecto XP. Para lograrla se usa:
  - *Pair programming*
  - *Refactoring*
  - Crear tests antes de que este escrito el código.

Prácticas de XP (5)

El testing ocupa un lugar "de honor":

- Se busca un buen testeo unitario y una buena cobertura del test de aceptación.
- El lema es que los desarrolladores son responsables de *demostrarle* a sus clientes que el código funciona.
- Es decir, *no son los clientes* los encargados de probar el código a fondo por si hay problemas.

Prácticas de XP (6)

- *Collective Code Ownership*: los desarrolladores trabajan en todas las áreas del sistema para evitar "islas de conocimiento".

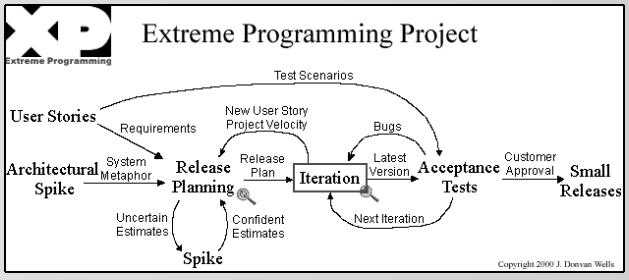
  Cualquiera puede cambiar cualquier cosa.
- *Integración continua*: tan pronto como se termina una tarea, se integra al resto del proyecto.

  Luego se deben pasar todos los unit tests.

Prácticas de XP (7)

- *Ritmo mantenible*: el uso de horas extras debe ser realmente una excepción.
- *On-site customer*: uno de los clientes debe estar disponible todo el tiempo para el equipo XP.

Proyecto XP: *Vista general*



Historias de usuario: *Repaso*

- Se usan para crear *estimaciones de tiempo* para la reunión de planificación de entrega.
- Son también usadas como reemplazo de un gran documento de requerimientos.
- Son escritas por los clientes como cosas que el sistema tiene que hacer por ellos en *tres oraciones de texto no técnico*.
- Guían la creación de los *tests de aceptación*.

Diferencia con requerimientos tradicionales

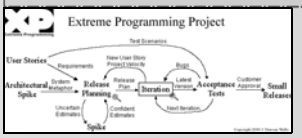
- Las historias de usuario sólo proveen suficiente detalle para hacer una *estimación razonable de bajo riesgo* acerca de cuánto tomará implementarla.
- A la hora de implementar la historia, los desarrolladores se comunican con el cliente para recibir una descripción más detallada de los requerimientos cara a cara.
- Cada historia puede tomar entre 1 y 3 semanas en tiempo ideal.
- Si se estima que llevará más tiempo, hay que dividirla.

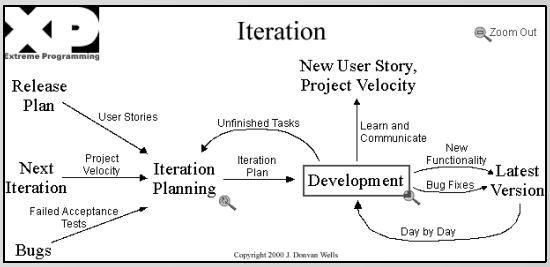
Diferencia con requerimientos tradicionales

- Las historias se enfocan en las necesidades del usuario.
- Se debe evitar dar detalles específicos: tecnología, diseño de bases de datos, algoritmos, etc.
- Se debe tratar de mantener las historias focalizadas en las necesidades y beneficios de los usuarios.

User stories

Proyecto XP: *Iteración*





Release Planning

- Se hace una reunión de planificación de entrega (*release planning meeting*) para crear un plan de entrega que resume el proyecto total.
- Este plan se usa luego para crear planes de las iteraciones individuales (*iteration plans*).
- Lo principal es que el equipo estime cada historia de usuario en términos de “semanas ideales de programación”.

Release Planning

- Las historias se imprimen en tarjetas.
- Los desarrolladores y clientes mueven las tarjetas en una mesa para crear un conjunto de historias que serán implementadas para la proxima entrega.
- Se busca crear un sistema:
  - Usable
  - Testeable
  - Con buen sentido de negocio
  - Que pueda ser entregado pronto

Release Planning

- Las iteraciones individuales se planifican en detalle justo antes de que comience esa iteración, y *nunca con antelación.*
- ¿Qué pasa si el release plan final no satisface a la gerencia?
  - ¡*Negociar*! Tener en cuenta las 4 variables: alcance, recursos, tiempo y calidad.
  - Nunca cambiar las estimaciones reales para "dibujar" un plan más agradable (esto indefectiblemente se paga al final).

Release Planning

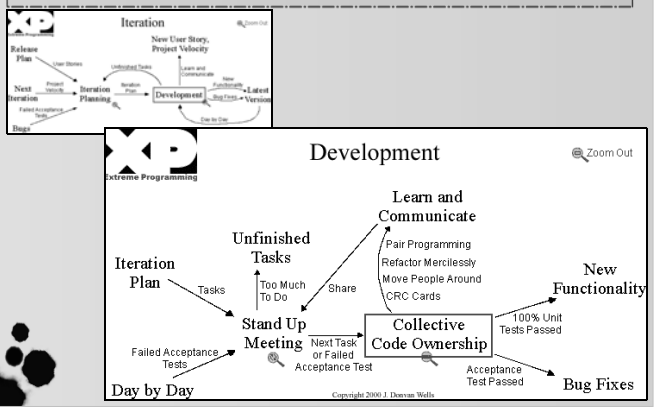
- Cuando la velocidad del proyecto cambia dramáticamente en una o dos iteraciones (o después de varias) hay que continuar, pero...
- Hacer un nuevo release planning con los clientes para crear un nuevo plan.

Entregas pequeñas y frecuentes

- El equipo entrega versiones iterativas a los clientes frecuentemente.
- La reunión de planficación de entrega se usa para descubrir pequeñas unidades de funcionalidad que tienen sentido desde el punto de vista del negocio.
- Esto es crítico para obtener feedback valioso en el tiempo adecuado que tenga impacto en el desarrollo.

Mientras más tarde se introduce una funcionalidad, menor será el tiempo para arreglarla.

Proyecto XP: *Desarrollo*



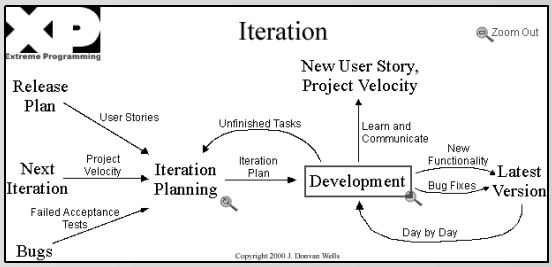
Desarrollo iterativo

- El desarrollo iterativo provee agilidad al proceso de desarrollo.
- *Regla general*: dividir el proceso de desarrollo en una docena de iteraciones de 1 a 3 semanas de longitud.
- Mantener la longitud de las iteraciones constante para que la planificación y las mediciones sean *simples*.
- Adelantarse va en contra de las reglas.

Desarrollo iterativo

- Tomarse los *deadlines* en serio (si no se llega, hay que re-planificar).
- Concentrar el esfuerzo en completar las tareas más importantes *según la elección del cliente*.
- ¡No tener varias tareas sin terminar elegidas por los desarrolladores!

Proyecto XP: *Iteración*



Planificación de iteraciones

- Se realiza al principio de cada iteración.
- Cada iteración dura de 1 a 3 semanas.
- Se eligen las historias de usuario de acuerdo al plan de entrega (*release plan*).
- Se eligen los test de aceptación fallados para ser arreglados.
- Las historias y los tests se dividen en tareas de programación que los respalden.

Planificación de iteraciones

- Las tareas se escriben en tarjetas, como las historias de usuario.
- Las tareas se escriben en el lenguaje de los desarrolladores.
- Las tareas duplicadas se *eliminan*. Las tarjetas restantes comprenden el plan detallado para la próxima iteración.
- Los desarrolladores se alistan para las tareas y estiman la duración de sus propias tareas.
- La estimación de cada tarea debe ser de 1 a 3 días.

Cambiar de lugar a las personas

- Evita cuellos de botella y pérdida seria de información.
- Importante: evitar las “islas de conocimiento”:
  - Cada desarrollador sabe un poquito de cada sección de código.
  - Favorecer que cada uno trabaje en una sección nueva en al menos parte de la iteración.
  - Combinar esto con *pair programming* hace que no haya pérdida en la productividad.

Pair programming

- Todo el código es creado por *dos personas trabajando juntas* en una sola computadora.
- Incrementa la calidad sin impactar en el tiempo de entrega.
- Aunque es contraintuitivo, se postula que:
  - Dos personas en una computadora agregan tanta funcionalidad como dos trabajando separadas.
  - La calidad resultante es mayor.

Pair programming

- Una persona tipea y piensa tácticamente sobre el método que está siendo creado mientras la otra piensa estratégicamente sobre cómo este método se integra en la clase.
- Puede tomar tiempo acostumbrarse a esta práctica.

¿No es un desperdicio?

- Dos personas harán el trabajo de una.
- Mayores costos.
- ¿Por qué querría poner a dos personas en un trabajo que puede hacer una?

Cómo ayuda pair programming

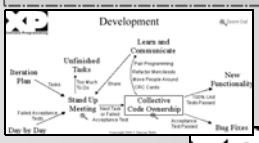
- Revisión continua
- Menos defectos / se detectan antes
- Calidad en el diseño
- El "*peer pressure*" ayuda a que se realicen las entregas a tiempo
- Menos distracciones
- Construye la comunicación en el grupo

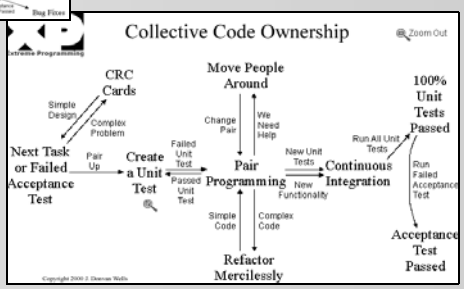
Cómo ayuda pair programming

Veamos un ejemplo de cómo funciona en el mundo real:

`https://www.youtube.com/watch?v=q-QWdFa4awI`

Proyecto XP: *Collective Code Ownership*





Collective Code Ownership

Collective Code Ownership alienta a todos a contribuir nuevas ideas en todos los segmentos del proyecto:

- Cualquier desarrollador puede cambiar líneas de código de cualquier módulo.
- Ninguna persona es un cuello de botella para los cambios.
- Cada desarrollador crea los tests unitarios para su propio código.

Collective Code Ownership

- Todo el código que es publicado al repositorio incluye tests unitarios.
- El código que se agrega, los bugs arreglados, etc. son cubiertos por el testeo automático.
- Se debe confiar en el *test suite* para cuidar el repositorio.
- Antes de entregar cualquier código, éste debe pasar el *test suite* en un 100%.

Reunión diaria de pie

- Objetivo: Evitar reuniones en las que la mayoría no contribuye.
- Se busca la comunicación entre todo el equipo.
- Evita que se generen muchas otras reuniones.

Elegir una buena metáfora

- Se busca ante todo un diseño simple.
- Elegir una metáfora para nombrar los métodos y clases en forma consistente.
- Es muy importante elegir los nombres correctos para mejorar la comprensión del sistema.
- *Ejemplo*: El sistema de sueldos de Crysler fue estructurado como una línea de producción.
- *Consejo*: no elegir una metáfora naive a menos que sea muy simple.

CRC Cards

- Usar las tarjetas CRC para diseñar el sistema en equipo:
  - Clases
  - Responsabilidades
  - Colaboraciones
- Permiten pensar al sistema desde una mentalidad “orientada a objetos”.
- Las tarjetas CRC individuales representan objetos.

CRC Cards

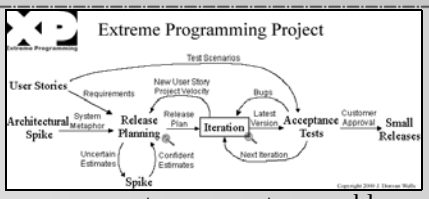
- La clase se puede escribir arriba, las responsabilidades se listan a la izquierda, y las clases que colaboran a la derecha.
- Puede ser que ninguna tarjeta esté completa.

Student	
Student number Name Address Phone number Enroll in a seminar Drop a seminar Request transcripts	Seminar

Diseñar...

- El diseño se alcanza en 3 maneras en un proyecto XP:
  - CRC cards (*nivel estratégico*)
  - Refactoring (*nivel táctico*)
  - Pair programming (*ambos niveles*)
- Se van colocando las tarjetas en la mesa, pensando cómo serán los módulos.
- Para probar el diseño, alguien "*simula la interacción del sistema*" enviando mensajes entre los objetos.

Crear una solución *spike*



- Se usan para encontrar respuestas a problemas técnicos difíciles, o a problemas de diseño.
- Es un programa simple para explorar soluciones potenciales – *luego se descarta*.
- Para reducir el riesgo se pueden encargar a dos desarrolladores que trabajen en forma separada.

Refactorizar

- Muchas veces reutilizamos código que no tiene la calidad adecuada sólo por pensar que si lo cambiamos dejará de funcionar.
- ¿Es esto efectivo en costo? *XP dice que no.*
- Se busca eliminar *redundancia*, eliminar funcionalidad que *no se usa* y rejuvenecer diseños *obsoletos*.
- Esto es *refactoring*.
- *Máxima*: "Refactoring a través del ciclo de vida del proyecto ahorra tiempo e incrementa la calidad."

Otras prácticas recomendadas por XP

- Usar estándares de codificación.
- Interactuar siempre con el usuario:
  - Test de funcionalidad (provee los datos)
  - Programación de cada tarea
- Recordar que se ahorró tiempo al comienzo.
- Crear tests unitarios *antes del código*.
- Ayuda a considerar qué es necesario hacer, y a producir código simple y sencillo sin redundancias.

Integración secuencial

- Pueden surgir problemas por la integración en paralelo de subsistemas.
- La solución de XP es usar un esquema de *integración secuencial* combinado con *collective code ownership*.
- Todo el código fuente es publicado en el repositorio en turnos. Sólo un par de programadores integra, testea y publica cambios en un momento dado.

Se puede usar un token físico

Tests Unitarios

- Son elementos clave de XP.
- Primero se debe crear o bajar un *unit test framework* para poder crear *unit test suites* automatizados.
- No se puede publicar código sin los tests.
- *Los tests se deben crear desde el principio.*

Tests Unitarios

- No es necesario que haya *code ownership* si todas las clases están protegidas por los tests unitarios.
- Con *tests unitarios automáticos* se puede mezclar un changeset con la última versión publicada en poco tiempo.

Unit Testing Frameworks

- Son herramientas de desarrollo, como lo son los editores y compiladores.
- No deben permanecer en reserva hasta el último mes del proyecto.
- Son fáciles de crear (esto ayuda a personalizar y entender mejor).
- También se pueden obtener para la mayoría de los lenguajes (por ejemplo, JUnit para Java).

Tests de aceptación

- Estos tests se *crean a partir de las historias de usuario*.
- Durante una iteración, las historias de usuario seleccionadas en la reunión de planificación se *traducirán* a tests de aceptación.
- El cliente especifica los escenarios a testear cuando una historia de usuario se implementa.
- Son tests de *caja negra*.
- Una historia *no se considera completa* hasta que haya pasado sus tests de aceptación.

Problemas con XP

- Compromiso por parte del cliente.
- Diseño arquitectónico:
  - El diseño incremental puede provocar que se tomen malas decisiones al principio.
  - Hacer *refactoring* de la arquitectura es muy caro.
- *Test complacency.*

Repaso de XP

XP – Software Development Process:

`https://www.youtube.com/watch?v=hbFOwqYIOcU`

Repaso de XP

XP's Values, Principles, and Practices:

`https://www.youtube.com/watch?v=C0WiUiznoUQ`

Repaso de XP

Repasos de prácticas XP:

- Incremental planning: https://www.youtube.com/watch?v=ttjyP9ncXlk
- Small releases: https://www.youtube.com/watch?v=1r4NW2HgUBg
- Simple design: https://www.youtube.com/watch?v=FerJJT-Q1xs
- Test first: https://www.youtube.com/watch?v=cmoDqkh-ss4
- Refactoring: https://www.youtube.com/watch?v=LsLniadcRTw
- Pair programming: https://www.youtube.com/watch?v=a8WaP3Fwqa0
- Continuous integration: https://www.youtube.com/watch?v=4bbvbyOHfRY
- On-site customer: https://www.youtube.com/watch?v=wEmVqz-C-yc

Bibliografía

- Manifiesto Ágil (http://agilemanifesto.org/)
- Scrum and XP from the Trenches, 2nd Ed. Henrik Kniberg. Free ebook.
- *What is Scrum*? http://www.mountaingoatsoftware.com/scrum
- *What is Extreme Programming*?

  https://ronjeffries.com/xprog/what-is-extreme-programming/

Otra bibliografía

K. Schwaber: "*Scrum development process*". Business object design and implementation. Springer, London, 1997. 117-134.