eXtreme Programming

Ingeniería del Software II

1. Metodologías actuales

2. Metodologías ágiles
   - eXtreme Programming
     - Prácticas
   - Test Driven Development
   - Crítica
   - Conclusiones

Status actual de los proyectos de software

- Falta de entendimiento del negocio.
- Cambios en el negocio.
- Altos costos de mantenimiento.
- Gran cantidad de defectos.
- Inestabilidad de las personas.
- Retrasos en las entregas.
- Proyectos cancelados.

Status actual de los proyectos de software

- Falta de entendimiento del negocio.
- Cambios en el negocio.
- Altos costos de mantenimiento.
- Gran cantidad de defectos.
- Inestabilidad de las personas.
- Retrasos en las entregas.
- Proyectos cancelados.

Problemas de las metodologías tradicionales

Pretenden

- imponer un modelo predecible sobre una entidad compleja.
- minimizar o restringir el cambio en todas las fases.
- congelar los requerimientos en forma temprana.
- congelar la arquitectura y diseño en las primeras fases del ciclo de vida.

Problemas de las metodologías tradicionales (cont.)

- Son orientadas a la *documentación*.
- Priorizan a los *procesos* por sobre las *personas*.
- Plantean la construcción en términos de todo o nada.
- No incorporan *feedback* sobre el proceso.

Metodologías ágiles

- No evitar el cambio sino *incorporarlo* al ciclo de vida.
- Controlar el *costo* del cambio.
- Decidir en el momento adecuado (en particular, no *antes*).
- ¡OJO! No aplica a todos los cambios.

1. Metodologías actuales

2. Metodologías ágiles
   - eXtreme Programming
     - Prácticas
   - Test Driven Development
   - Crítica
   - Conclusiones

Metodologías ágiles

- *Personas* e *interacciones* sobre procesos y herramientas.
- Software funcionando por sobre documentación clara.
- Cliente involucrado en vez de negociación contractual.
- Responder al cambio en vez de seguir un plan.

Metodologías ágiles

- *Personas* e *interacciones* sobre procesos y herramientas.
- Software funcionando por sobre documentación clara.
- Cliente involucrado en vez de negociación contractual.
- Responder al cambio en vez de seguir un plan.

"En tanto se valoricen los ítems de la izquierda, valoraremos más los ítems de la derecha."

Metodologías ágiles - ¿Cómo?

- Producir el primer *delivery* en semanas para tener rápido feedback.
- Soluciones simples, es decir menos cambio para hacer cambios.
- Mejorar la calidad constantemente.
- Test continuo, para detectar errores antes.

Metodologías ágiles - ¿Cómo? (cont.)

Control del proceso

- *Feedback* frecuente para conocer la situación actual en intervalos regulares.
  - El tiempo de iteración debe ser el más corto posible (entre 2 semanas y 1 mes).
- El cliente tiene mayor *control* sobre el proyecto.
  - El cliente y el equipo de desarrollo evalúan permanentemente el progreso.
  - El cliente puede cambiar las funcionalidades del software de acuerdo a los cambios en el negocio.

Metodologías ágiles - ¿Cómo? (cont.)

Control del proceso

- *Feedback* frecuente para conocer la situación actual en intervalos regulares.
  - El tiempo de iteración debe ser el más corto posible (entre 2 semanas y 1 mes).
- El cliente tiene mayor *control* sobre el proyecto.
  - El cliente y el equipo de desarrollo evalúan permanentemente el progreso.
  - El cliente puede cambiar las funcionalidades del software de acuerdo a los cambios en el negocio.

Metodologías ágiles - ¿Cómo? (cont.)

Control del proceso

- *Feedback* frecuente para conocer la situación actual en intervalos regulares.
  - El tiempo de iteración debe ser el más corto posible (entre 2 semanas y 1 mes).
- El cliente tiene mayor *control* sobre el proyecto.
  - El cliente y el equipo de desarrollo evalúan permanentemente el progreso.
  - El cliente puede cambiar las funcionalidades del software de acuerdo a los cambios en el negocio.

Metodologías ágiles - ¿Cómo? (cont.)

Control del proceso

- *Feedback* frecuente para conocer la situación actual en intervalos regulares.
  - El tiempo de iteración debe ser el más corto posible (entre 2 semanas y 1 mes).
- El cliente tiene mayor *control* sobre el proyecto.
  - El cliente y el equipo de desarrollo evalúan permanentemente el progreso.
  - El cliente puede cambiar las funcionalidades del software de acuerdo a los cambios en el negocio.

eXtreme Programming

Un poco de historia...

Surge en la comunidad Smalltalk (1980s - Kent Beck, Ward Cunningham).
1996 — Beck hace auditoría del proyecto de liquidación de sueldos de Chrysler. Debido a la baja calidad del código entregado, Beck sugirió desechar todo el código y empezar de cero.
Pone en marcha una serie de prácticas aprendidas anteriormente, y se van refinando hasta convertirse en lo que serían las bases de XP.

eXtreme Programming

Un poco de historia - los valores

- Communication
- Simplicity
- Feedback
- Courage
- Respect

eXtreme Programming - Roles

Los roles reemplazan la visión clásica de responsabilidades. Todo miembro del equipo *podría* tomar cualquiera de los roles en un momento dado.

- Desarrollador – escribe código, tests; comunica.
- Cliente – escribe *stories*, *acceptance tests*, prioriza.
- Tester – complemento del desarrollador si es necesario.
- Tracker – seguimiento de estimaciones, mejoras al proceso.
- Consultor – miembro externo, aporta conocimiento técnico.
- Coach/Instructor – responsable del proceso (à la ScrumMaster).
- Manager – tomador de decisiones.

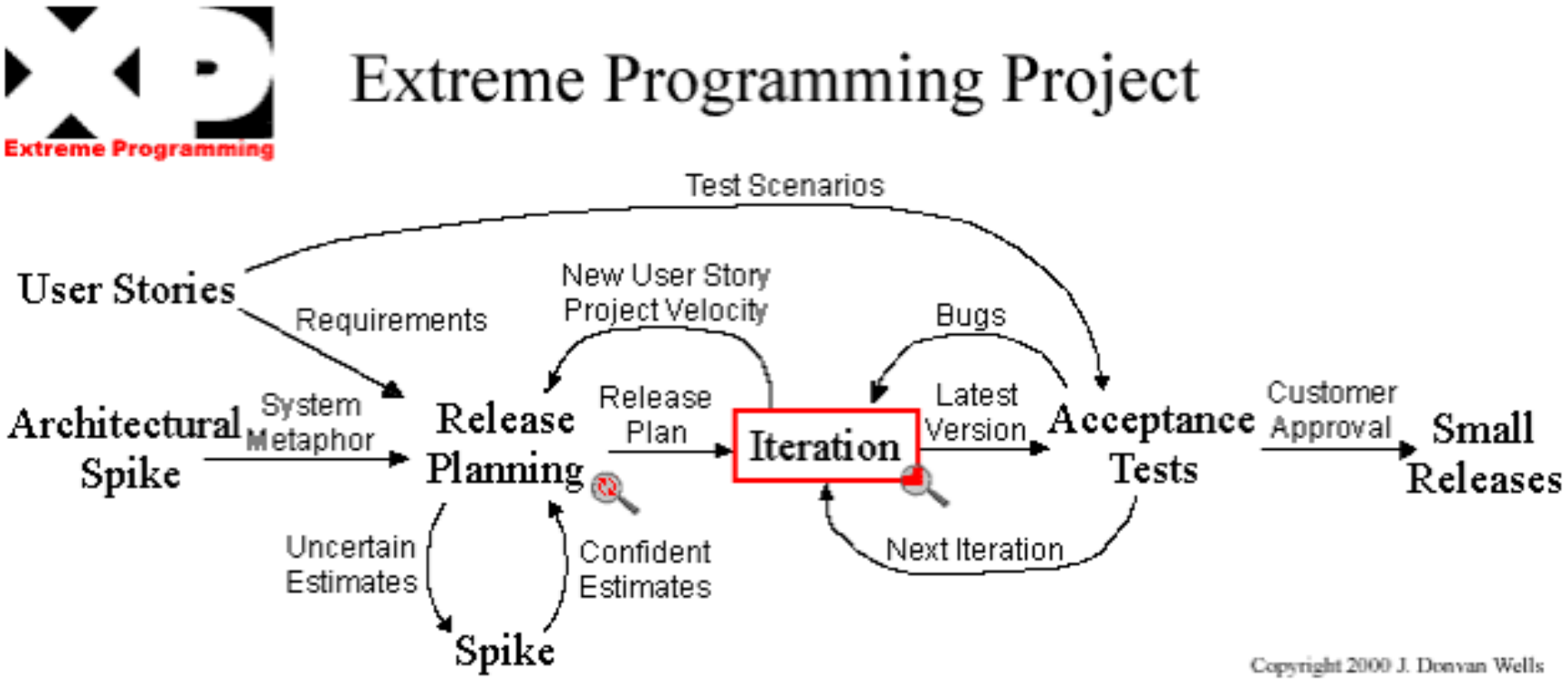
eXtreme Programming - Prácticas

- Planning Process (Planning Game).
- Releases chicos e iteraciones cortas.
- Metáforas (acuerdo en nombres/descripciones).
- Diseño simple.
- Testing continuo.
- Refactoring.
- TDD (*Test Driven Development*).
- Pair programming.
- *Ownership* colectivo.
- Integración continua.
- Semana de 40 horas.
- Cliente *on-site*.
- Estándar de código.

eXtreme Programming - Prácticas

- Planning Process (Planning Game).
- Releases chicos e iteraciones cortas.
- Metáforas (acuerdo en nombres/descripciones).
- Diseño simple.
- Testing continuo.
- Refactoring.
- TDD (*Test Driven Development*).
- Pair programming.
- *Ownership* colectivo.
- Integración continua.
- Semana de 40 horas.
- Cliente *on-site*.
- Estándar de código.

eXtreme Programming - Prácticas



eXtreme Programming - Planning game

- Release planning
  - Exploration - *user stories*, prioridades del cliente.
  - Commitment - alcance del release.
  - Steering - ajustes del plan; agregar/quitar requerimientos.
- Iteration planning
  - Exploration - *user stories* → *task cards*.
  - Commitment - asignación de tareas a miembros del equipo, estimación.
  - Steering - iteración de trabajo y contraste con *user stories*.

eXtreme Programming - Planning game

- Release planning
  - Exploration - *user stories*, prioridades del cliente.
  - Commitment - alcance del release.
  - Steering - ajustes del plan; agregar/quitar requerimientos.
- Iteration planning
  - Exploration - *user stories* $\longrightarrow$ *task cards*.
  - Commitment - asignación de tareas a miembros del equipo, estimación.
  - Steering - iteración de trabajo y contraste con *user stories*.

eXtreme Programming - Simple Design

No pensar ni agregar nada antes de tiempo. Mantener todo tan simple como sea posible, satisfaciendo el requerimiento y *aceptar* el cambio en próximas iteraciones. No "pensar mejor" que el cliente.

- YAGNI - You aren't gonna need it.
- KISS - Keep it simple, stupid!

eXtreme Programming - Simple Design

No pensar ni agregar nada antes de tiempo. Mantener todo tan simple como sea posible, satisfaciendo el requerimiento y *aceptar* el cambio en próximas iteraciones. No "pensar mejor" que el cliente.

Acrónimos graciosos...

- YAGNI - You aren't gonna need it.
- KISS - Keep it simple, stupid!

eXtreme Programming - Simple Design

No pensar ni agregar nada antes de tiempo. Mantener todo tan simple como sea posible, satisfaciendo el requerimiento y *aceptar* el cambio en próximas iteraciones. No "pensar mejor" que el cliente.

Acrónimos graciosos...

- YAGNI - You aren't gonna need it.
- KISS - Keep it simple, stupid!

eXtreme Programming - Testing continuo

El testing debe verse como una parte integral del proceso de desarrollo, y el mismo desarrollador debe estar íntimamente involucrado en él.

Hoy en día sobran herramientas para llevarlo a cabo (JUnit, NUnit, etc.), pero voluntad... El ideal es mantener un ritmo sostenible de implementación de funcionalidad $\longrightarrow$ test. Más de esto en TDD.

eXtreme Programming - Pair programming

Dos miembros del equipo trabajando en una misma máquina:

- Uno se ocupa de los detalles del código en sí, tiene el control de la máquina.
- El otro mantiene la mirada en el contexto, revisa el código, sugiere cambios, etc.

eXtreme Programming - Pair programming

Dos miembros del equipo trabajando en una misma máquina:

- Uno se ocupa de los detalles del código en sí, tiene el control de la máquina.
- El otro mantiene la mirada en el contexto, revisa el código, sugiere cambios, etc.

Los roles deben ir cambiando sucesivamente en una misma iteración. Además, los pares deben trabajar en distintos problemas dentro de la misma iteración, incluso armando y desarmando pares $\longrightarrow$ Ownership colectivo.

eXtreme Programming - Integración continua

Una herramienta y un buen mantenimiento de *source control* son vitales – la comunicación de las personas es vital.
Todos los integrantes del equipo deben trabajar constantemente sobre la *más reciente* versión de todo lo implementado.
No sirve de nada si uno se pasa todo el día con una solución maravillosa a un problema si después la integración es muy difícil o imposible...

TDD - Test-Driven Development

Es un hecho – los desarrolladores no suelen testear.

"Solución": forzar el test *antes* de la implementación.

- Enunciar claramente un comportamiento lo más simple y pequeño posible del *story* que se está implementando.
- Escribir un test que lo valide.
- Correr *todos* los tests que tenemos al momento: el nuevo falla.
- Escribir el código más simple que *haga pasar el test*.
- Correr los tests nuevamente: todos pasan.
- Refactoreo + retesteo.
- Repeat!

TDD - Test-Driven Development

Es un hecho – los desarrolladores no suelen testear.

"Solución": forzar el test *antes* de la implementación.

- Enunciar claramente un comportamiento lo más simple y pequeño posible del *story* que se está implementando.
- Escribir un test que lo valide.
- Correr *todos* los tests que tenemos al momento: el nuevo falla.
- Escribir el código más simple que *haga pasar el test*.
- Correr los tests nuevamente: todos pasan.
- Refactoreo + retesteo.
- Repeat!

TDD - Test-Driven Development

Es un hecho – los desarrolladores no suelen testear.

"Solución": forzar el test *antes* de la implementación.

- Enunciar claramente un comportamiento lo más simple y pequeño posible del *story* que se está implementando.
- Escribir un test que lo valide.
- Correr *todos* los tests que tenemos al momento: el nuevo falla.
- Escribir el código más simple que *haga pasar el test*.
- Correr los tests nuevamente: todos pasan.
- Refactoreo + retesteo.
- Repeat!

TDD - Test-Driven Development

Es un hecho – los desarrolladores no suelen testear.

"Solución": forzar el test *antes* de la implementación.

- Enunciar claramente un comportamiento lo más simple y pequeño posible del *story* que se está implementando.
- Escribir un test que lo valide.
- Correr *todos* los tests que tenemos al momento: el nuevo falla.
- Escribir el código más simple que *haga pasar el test*.
- Correr los tests nuevamente: todos pasan.
- Refactoreo + retesteo.
- Repeat!

TDD - Test-Driven Development

Es un hecho – los desarrolladores no suelen testear.

"Solución": forzar el test *antes* de la implementación.

- Enunciar claramente un comportamiento lo más simple y pequeño posible del *story* que se está implementando.
- Escribir un test que lo valide.
- Correr *todos* los tests que tenemos al momento: el nuevo falla.
- Escribir el código más simple que *haga pasar el test*.
- Correr los tests nuevamente: todos pasan.
- Refactoreo + retesteo.
- Repeat!

TDD - Test-Driven Development

Es un hecho – los desarrolladores no suelen testear.

"Solución": forzar el test *antes* de la implementación.

- Enunciar claramente un comportamiento lo más simple y pequeño posible del *story* que se está implementando.
- Escribir un test que lo valide.
- Correr *todos* los tests que tenemos al momento: el nuevo falla.
- Escribir el código más simple que *haga pasar el test*.
- Correr los tests nuevamente: todos pasan.
- Refactoreo + retesteo.
- Repeat!

TDD - Test-Driven Development

Es un hecho – los desarrolladores no suelen testear.

"Solución": forzar el test *antes* de la implementación.

- Enunciar claramente un comportamiento lo más simple y pequeño posible del *story* que se está implementando.
- Escribir un test que lo valide.
- Correr *todos* los tests que tenemos al momento: el nuevo falla.
- Escribir el código más simple que *haga pasar el test*.
- Correr los tests nuevamente: todos pasan.
- Refactoreo + retesteo.
- Repeat!

¿Somos suficientemente ágiles?

¿Podemos eliminar prácticas?

Saquemos...

- el refactoring...
- el test de unidad...
- el *pair programming*...
- el cliente *on-site*...

¿Somos suficientemente ágiles?

¿Podemos eliminar prácticas?

Saquemos...

- el refactoring...
- el test de unidad...
- el *pair programming*...
- el cliente *on-site*...

¿Somos suficientemente ágiles?

¿Podemos eliminar prácticas?

Saquemos...

- el refactoring...
- el test de unidad...
- el *pair programming*...
- el cliente *on-site*...

¿Somos suficientemente ágiles?

¿Podemos eliminar prácticas?

Saquemos...

- el refactoring...
- el test de unidad...
- el *pair programming*...
- el cliente *on-site*...

¿Somos suficientemente ágiles?

¿Podemos eliminar prácticas?

Saquemos...

- el refactoring...
- el test de unidad...
- el *pair programming*...
- el cliente *on-site*...

¿Somos suficientemente ágiles?

¿Podemos eliminar prácticas?

Saquemos...

- el refactoring...
- el test de unidad...
- el *pair programming*...
- el cliente *on-site*...

La mayoría de quienes dicen aplicar XP ¡NO aplican bien todas las prácticas!


Req.
Cambiantes
Diseño
emergente
Refactoring
Testing
Pair
Programming
Collective
Ownership
On-Site Client

Dónde SÍ sirve

XP sirve entonces donde

- Los requerimientos son muy difusos.
- El cliente mismo no tiene muy en claro lo que quiere.
- Equipos de desarrollo chicos.
- Integrantes no novatos.
- Alto nivel de comunicación.

Donde NO sirve

XP no sirve entonces donde

- Equipos de desarrollo grandes...
- o con poca comunicación...
- o con gran rango de *expertise*.
- Sistemas críticos.