

Sesión S12: Planificación de pruebas

Planificación del proyecto

Efectividad y alcance de las pruebas

Proceso de desarrollo y niveles de pruebas

Planificación de pruebas

Las pruebas en diferentes modelos de proceso

Pruebas y diseño: TDD vs BDD

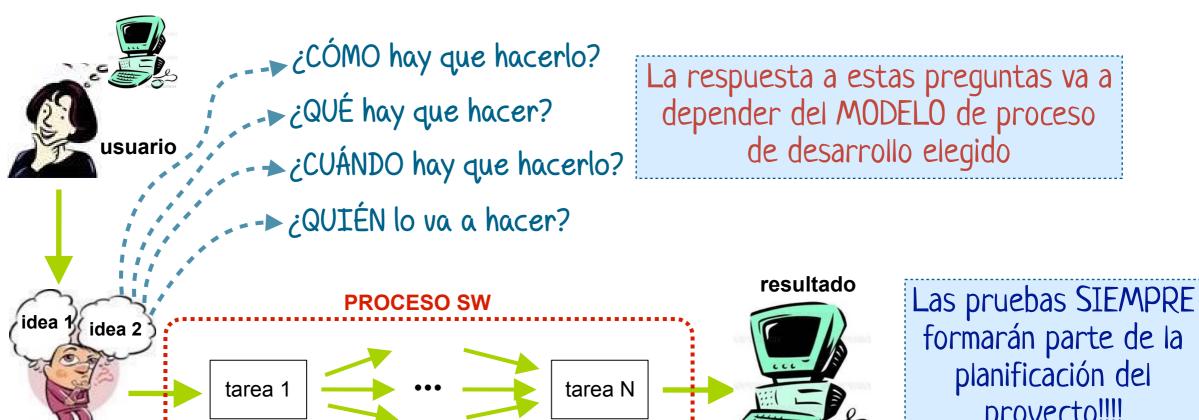
Otras prácticas de pruebas: Integraciones continuas (CI)



LA IMPORTANCIA DE LA FLANIFICACION

Cuando se planifica un proyecto, aunque el propósito siempre es el mismo...





formarán parte de la proyecto!!!!

- ... se hace de forma diferente dependiendo del **modelo de proceso**:
 - Planificación predictiva vs. adaptativa

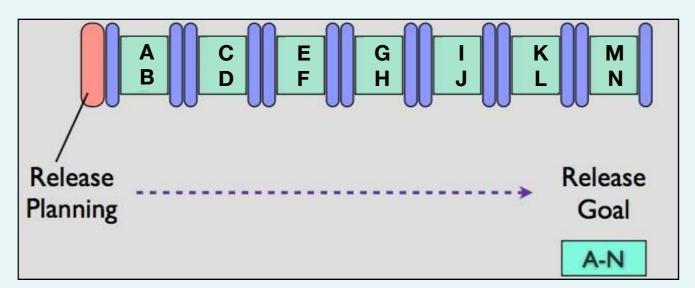
ingeniero sw

- Los modelos iterativos y ágiles realizan una planificación adaptativa
- Se establecen diferentes **niveles de planificación**: cada modelo de proceso considera determinados horizontes:
 - Los modelos ágiles planifican como mínimo a nivel de día, iteración y release

PLANIFICACIÓN PREDICTIVA VS. ADAPTATIVA

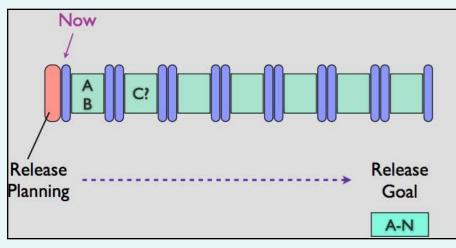
Imágenes adaptadas de: http://www.odd-e.com/material/2012/07_mda_nanyang/short_intro_agile.pdf

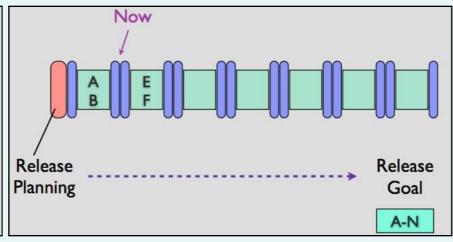


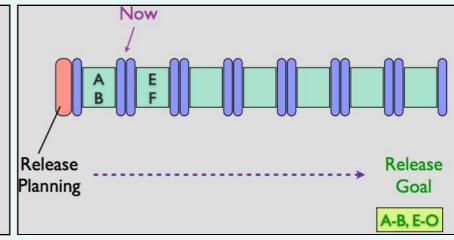


Soporta muy mal los cambios!!!

Adaptativa







No es posible tener claras todas las "variables" que intervienen en el desarrollo de un proyecto al comienzo del mismo (cono de incertidumbre), como por ejemplo requerimientos específicos, los detalles de la solución, cuestiones sobre el personal del proyecto, etc.



Un plan ADAPTATIVO intenta encontrar un equilibrio entre el esfuerzo invertido en realizar el plan, frente a la información (conocimiento) disponible en cada momento. Proporciona "agilidad": resulta "sencillo" incluir cambios!!!

MÚLTIPLES NIVELES DE PLANIFICACIÓN (I)

Un plan con muy poca exactitud es incontrolable

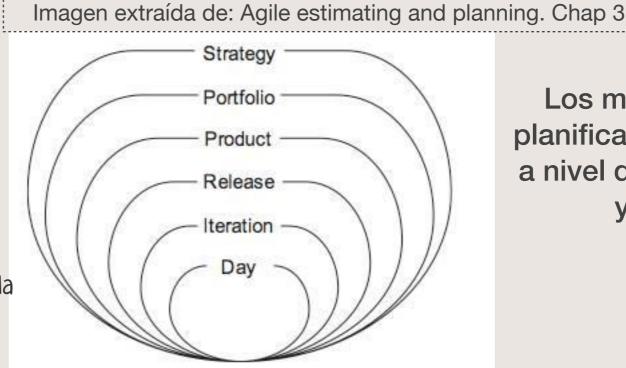


- Cuando nos marcamos objetivos, es importante recordar que no podemos "ver más allá del horizonte" y que la **exactitud** en nuestro plan decrecerá rápidamente tanto más cuanto más sobrepasemos dicho horizonte
 - Un proyecto "está en riesgo" si su planificación se extiende más allá del horizonte del planificador y no incluye tiempo para que éste "levante la cabeza", vuelva a mirar al nuevo horizonte, y realice los ajustes necesarios.

Release: considera las historias de usuario que serán desarrolladas en la SIGUIENTE entrega del desarrollo al cliente

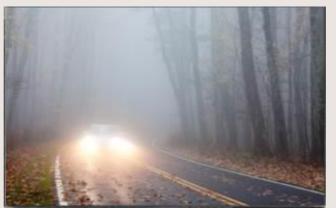
Product: considera la evolución de posteriores "releases"

Portfolio: considera la selección de desarrollos dentro de la estrategia de la empresa (desarrolladora)



Los modelos ágiles planifican como mínimo a nivel de *día*, *iteración* y *release*

Cada modelo de proceso considera determinados horizontes. Cada horizonte proporciona una "visibilidad" adecuada para el correcto progreso del desarrollo del proyecto



Con una mala "visibilidad"no es posible planificar correctamente!!!

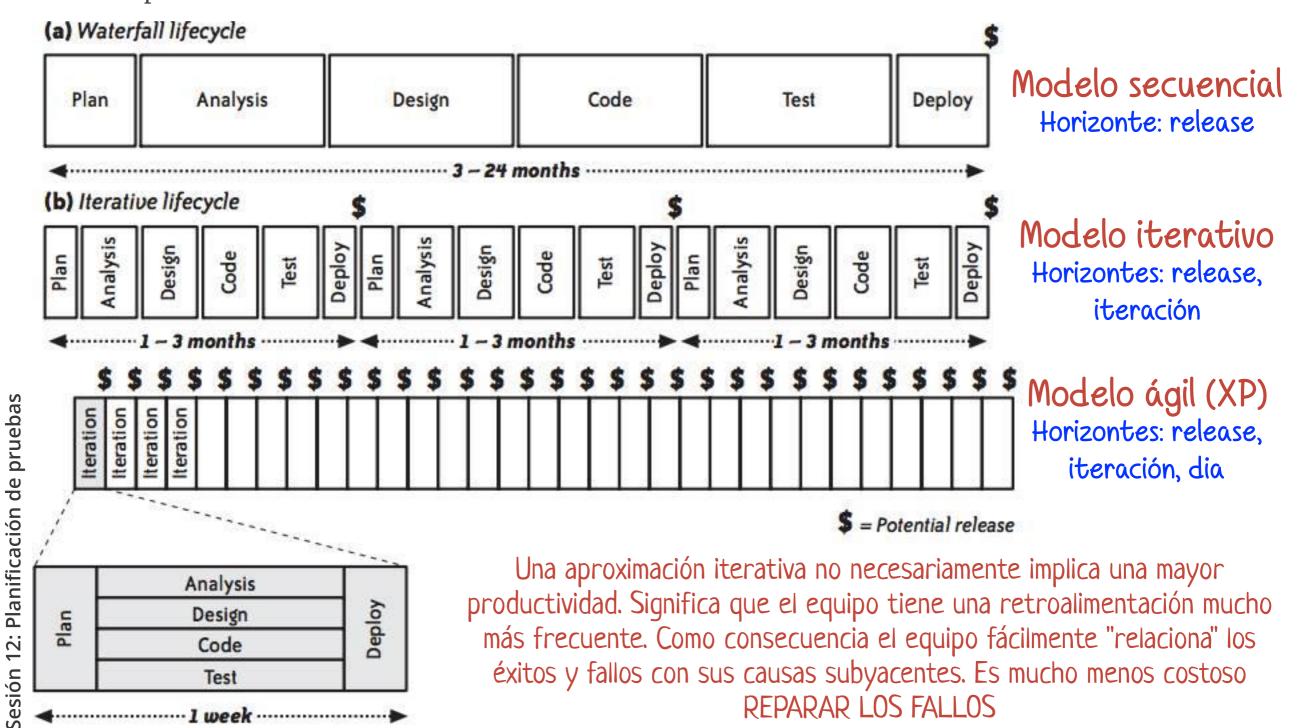


MÚLTIPLES NIVELES DE L'ANIFICACION (I) Cada modelo de proceso requiere planes con diferentes horizontes

Imágenes extraídas de:The art of agile development. James Shore. 2008. Cap. 3



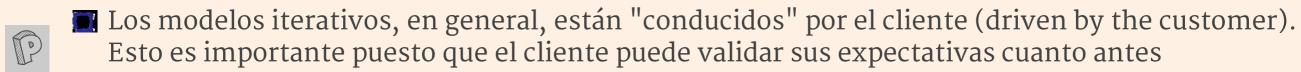
Cuando nos marcamos objetivos, es importante recordar que no podemos "ver más allá del horizonte" y que la exactitud en nuestro plan decrecerá rápidamente tanto más cuanto más sobrepasemos dicho horizonte



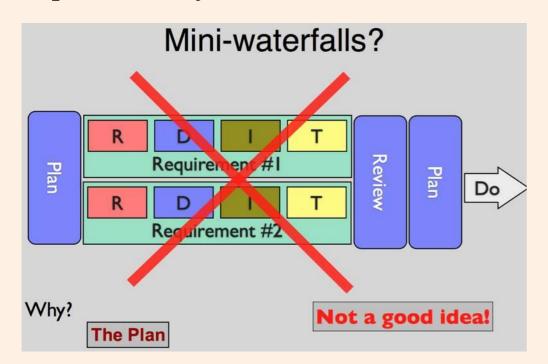
ITERACIONES Y TIME-BOXING

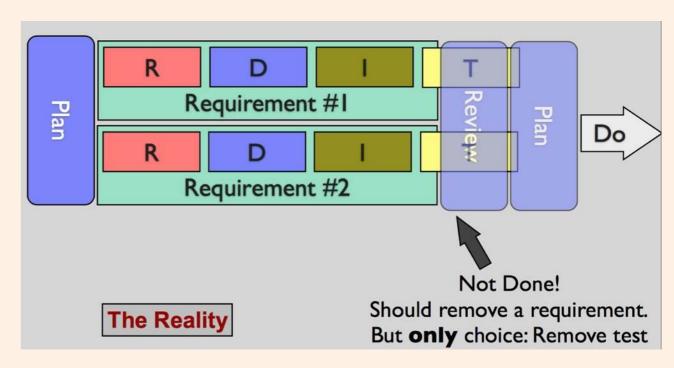
En un modelo iterativo es FUNDAMENTAL que las iteraciones sean del mismo "tamaño" y que sean time-boxed

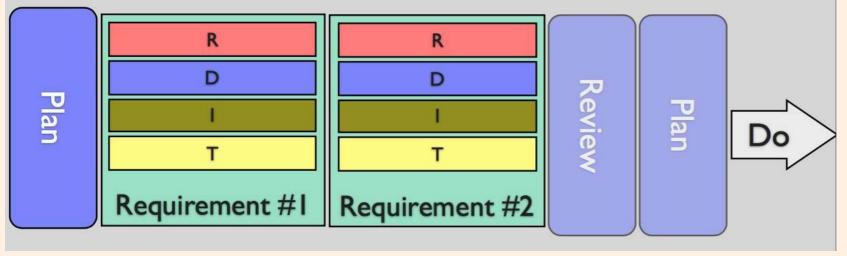
¿Quién decide lo que hay que hacer en la siguiente iteración?



Las iteraciones deben ser SIEMPRE time-boxed: nunca se retrasa el tiempo de entrega, es preferible y es mucho más sencillo cambiar el "scope"







Es un ERROR el planificar una iteración como una minicascada. Si lo hacemos así, no será posible cambiar el "scope" sin repercutir negativamente en el éxito del proyecto

ALCANCE Y EFECTIVIUAD DE LAS PKUEBAS

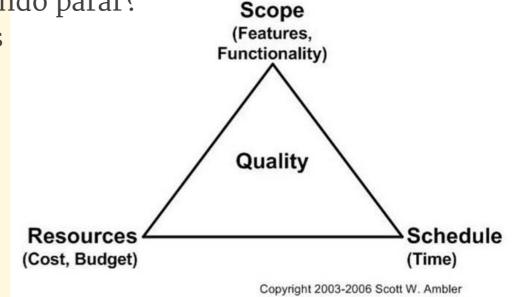
Para planificar las pruebas es importante tener en cuenta las siguientes cuestiones:



¿Cuántas pruebas son suficientes, y cómo decidir cuándo parar?

■ Factores para tomar la decisión: triángulo de recursos

El desarrollo del software debe equilibrar los tres vértices del triángulo: tiempo, dinero y funcionalidad. Estos tres recursos INFLUENCIAN la calidad (propiedades emergentes) que se incluyen (o no) en el software entregado



- Para conseguir un resultado aceptable hay que:
 - **Priorizar**: decidir qué tests son más importantes
 - Fijar criterios para conseguir unos objetivos de pruebas de forma que sepamos cuándo parar. P.ej. qué áreas van a ser más probadas y con qué cobertura, qué nivel de defectos se van a tolerar en un producto entregado... (completion criteria)
- El objetivo final es asegurar que las pruebas son **EFECTIVAS**
 - Un buen test es aquél que encuentra un defecto. Si encontramos un defecto estamos creando una oportunidad de mejorar la calidad del producto
- El proceso de pruebas debe ser **EFICIENTE**:
 - Encontrar el mayor número de defectos con el menor número de pruebas posibles
 - Para ello se deben utilizar buenas técnicas de DISEÑO de casos de prueba

EL PROCESO DE PRUEBAS

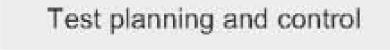




OBJETIVOS DEL PROCESO DE PRUEBAS

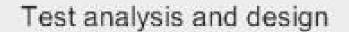
DEMOSTRAR que el software satisface los requerimientos

Descubrir situaciones en las que el comportamiento del software es incorrecto, indeseable, o no conforme a las especificaciones



→ Se determina ¿QUÉ? se va a probar, ¿CÓMO?, ¿QUIÉN? y ¿CUÁNDO? (planning)

Se determina QUÉ se va a hacer si los planes no se ajustan a la realidad (control)



- ⇒Se determina qué CASOS DE PRUEBA se van a utilizar
- →Un caso de prueba es un DATO CONCRETO + RESULTADO ESPERADO

Test implementation and execution →Se implementan y ejecutan los tests. Es la parte más visible, pero no es posible que los test sean efectivos sin realizar los pasos anteriores

Evaluating exit criteria and reporting

Se verifica que se alcanzan los *completion criteria* (p.ej 85% de cobertura) y se genera un informe

Test closure activities

➡En este punto las pruebas ya han finalizado. Se trata de asegurar que los informes están disponibles, ...





NIVELES DE PRUEBAS

Java, Maven, JUnit,

EasyMock





		VERIFICACIÓN. Objetivo: encontrar defectos Realizada por los desarrolladores			VALIDACIÓN. Objetivo: ver en qué grado se satisfacen las expectativas del cliente. Requieren al usuario
		UNIDAD	INTEGRACIÓN	SISTEMA	ACEPTACIÓN
oruebas	OBJETIVO	Encontrar defectos en las unidades. Deben de probarse de forma AISLADA	Encontrar defectos en la interacción de las unidades. Debe establecerse un ORDEN de integración	Encontrar defectos derivados del comportamiento del sistema como un todo.	Valorar en qué GRADO se satisfacen las expectativas del cliente. Basadas en criterios de ACEPTACIÓN (prop. emergentes) cuantificables
	DISEÑO	Camino básico (CB) Particiones equivalentes (CN)	Guías (consejos) en función de los tipos de interfaces (CN)	Transición de estados (CN)	Basado en requerimientos (CN) Basado en escenarios (CN) Pruebas de rendimiento (CN) Pruebas α y β (CN)

IMPLEMEN-TACIÓN Java, Maven, JUnit, DbUnit
Java, Maven, Junit,
DbUnit
Java, Maven, Junit,
Java, Maven,
JMeter

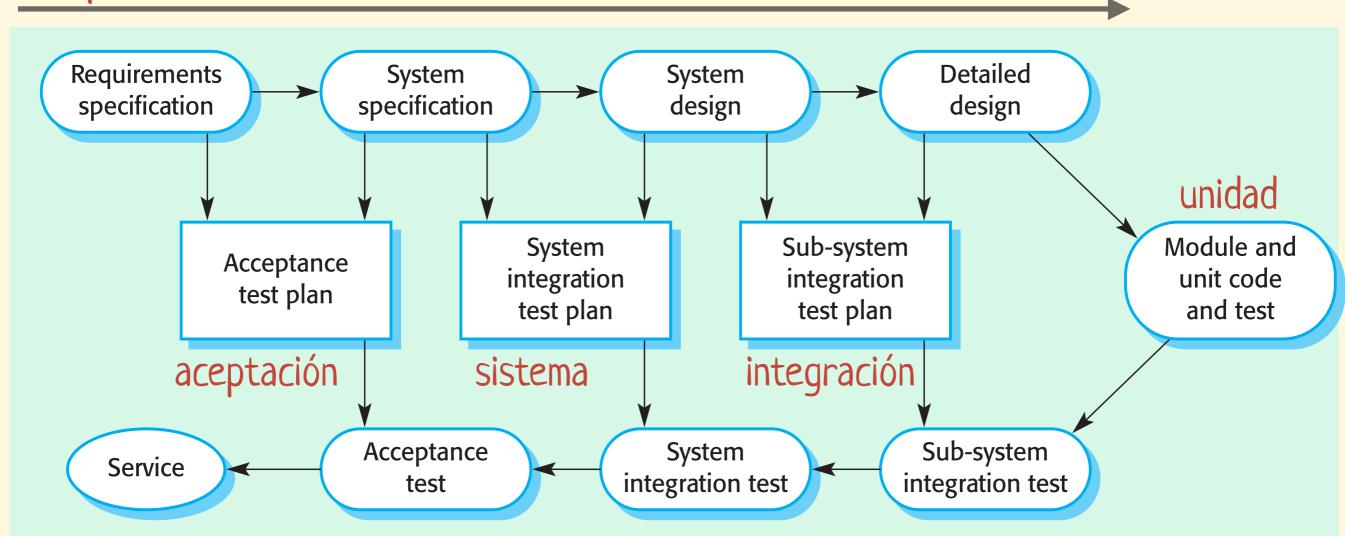
Selenium IDE Java, Maven, Junit, Webdriver. JMeter Usuario (α y β)

PLANIFICACIÓN TEMPORAL DE LOS NIVELES DE PRUEBAS



- Es importante conocer:
 - **CUÁNDO** se debe realizar cada **TIPO** de prueba
 - Qué artefactos son necesarios para diseñar y ejecutar cada tipo de pruebas

Tiempo



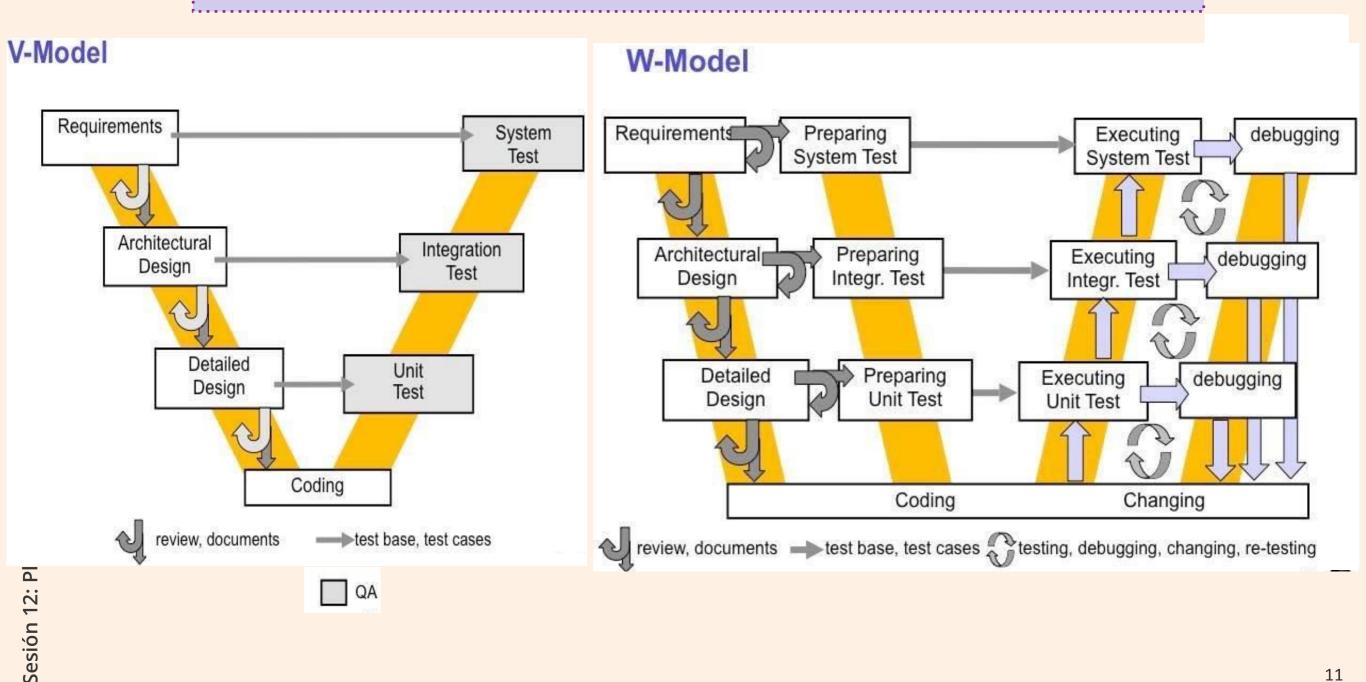
PRUEBAS EN MODELOS SECUENCIALES





Se trata de incluir en el plan de desarrollo las actividades de pruebas. Se debe contemplar un equilibrio entre las pruebas estáticas y las dinámicas

PLANIFICACIÓN DE PRUEBAS EN UN MODELO DE DESARROLLO SECUENCIAL



PRUEBAS EN MODELOS TIERATIVOS Y ÁGILES (I)

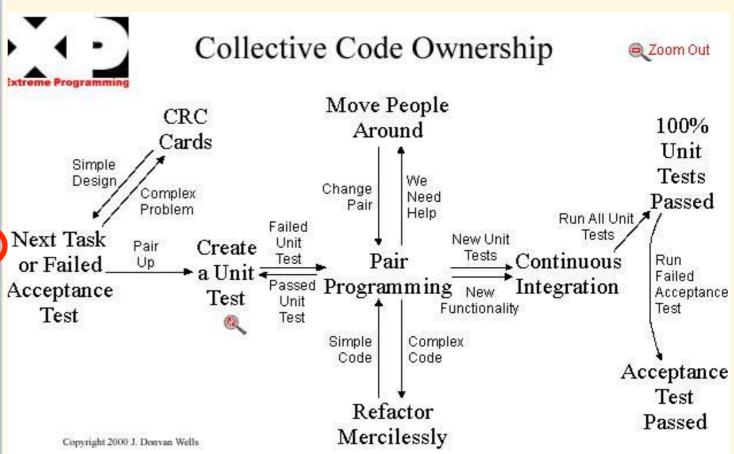
En modelos **iterativos**, las pruebas se planifican a nivel de ITERACIÓN y RELEASE (una release está formada por un conjunto de iteraciones)

PRUEBAS EN MODELOS ITERATIVOS

Modelo UP

Phases Disciplines Inception Elaboration Construction Transition **Business Modeling** Requirements Analysis & Design Implementation Test Deployment Configuration & Change Mgmt Project Management Environment Const Tran Const Const Elab #2 Elab #1 Initial #2 Iterations

Modelo XP



En modelos ágiles, las pruebas se planifican a nivel se de DÍA, ITERACIÓN y RELEASE

Cada RELEASE se divide en 4 fases. Cada fase de DÍA, I contiene una o más ITERACIONES

Sesión 12: Planificación de pruebas

PRUEBAS EN MODELOS TIERATIVOS Y ÁGILES (II)





El **product owner** es

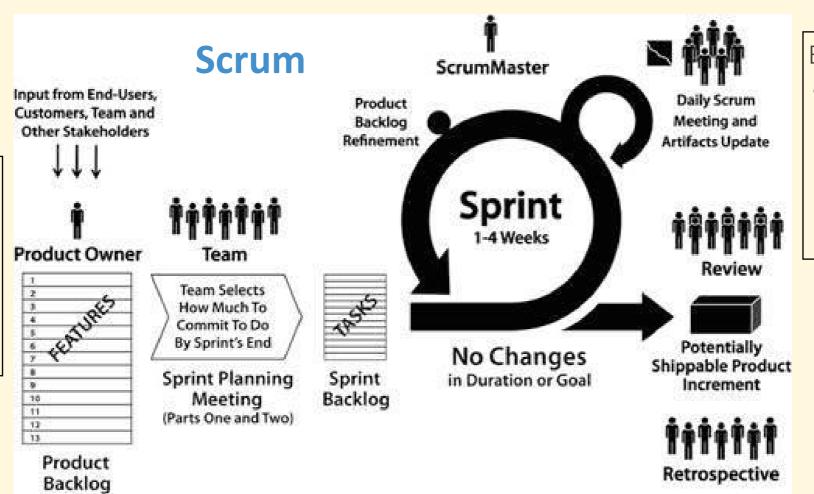
responsable de

conseguir el

máximo valor de

negocio del

producto

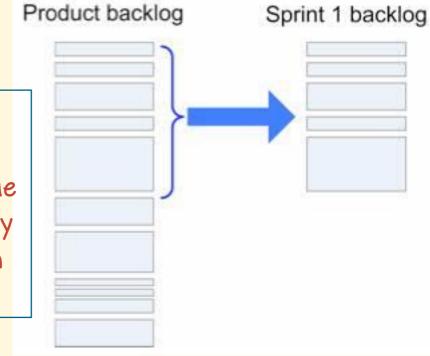


El **scrumMaster** es el que actúa como guía del grupo durante el proceso, "protege" al grupo del exterior, y sirve como ayuda al mismo. NO es un gestor de proyectos

Product backlog: Lista de requerimientos priorizados por su valor de negocio (los valores más altos al principio de la lista).

En un proceso Scrum NO hay una fase de testing "separada" del resto de actividades del desarrollo.

Cuando un desarrollador termina una historia de usuario, los tests tienen que estar preparados para su ejecución. Si el test pasa, la historia es aceptada y se pasa a la siguiente. Una vez que se han probado todas las historias y han pasado los tests, se da por concluido el sprint y se pasa al siguiente



¿QUÉ ASPECTO TIENE UN PLAN EN XP? (I) se planifica en varios niveles



Planificación de pruebas

Sesión 12:

del producto.

- **Story cards**: contienen historias de usuario. Representan características requeridas por el usuario (no casos de uso o escenarios). Se centran en el "beneficio" o resultado (value) que se pretende obtener. Deben describir un objetivo que permita a los testers evaluar una implementación exitosa de la misma. El formato suele ser:
 - As an [actor] I want [action] so that [achievement]. For example: As a Facebook member, I want to set different privacy levels on my page so that I can control who sees my page.

Task list: lista de tareas para las story cards de una iteración

ITERACIÓN (= I semana) no story Story card prioridad Task board Tests As a Story To Do Ready In Process To Verify Hours I would like esfuerzo Code the... As a user, I Code the... Code the... 33 SC LC can... So that 5 Code the... Code the... DC A **Historias** Criterios de aceptación de usuario Test the... Acceptance Criteria: - Criteria one **Tareas** As a user, I Code the... 13 - Criteria two can... 2 - Criteria three Code the... Las historias de usuario se **priorizan** Code the... Code the... As a user, I según su valor de negocio. El conjunto de 13 MC can... historias de usuario proporcionan la visión

3

Code the...

Sesión 12: Planificación de pruebas

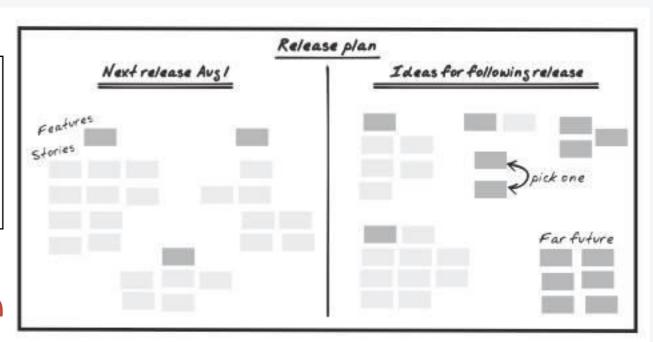
¿QUÉ ASPECTO TIENE UN PLAN EN XP? (II)



👅 Release plan board

Primero se crea el plan de entregas, consistente en una la lista de **historias de usuario** priorizadas por su valor de negocio (los valores más altos al principio de la lista). Se obtiene como resultado del proceso denominado *planning game*

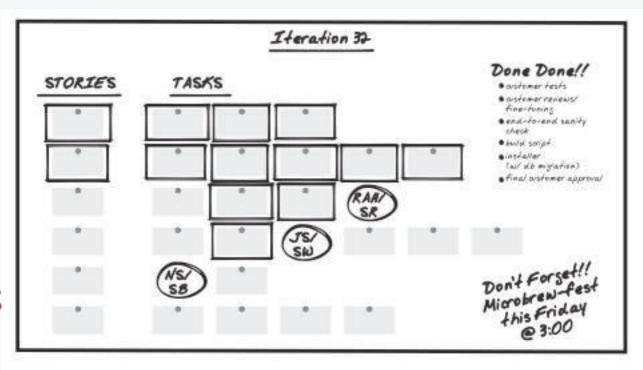
No se asignan recursos (¿quién?), ni tiempo (¿cúando?) hasta que comience cada iteración



Iteration plan board

Las historias de usuario se dividen en **tareas** concretas y son desarrolladas por los programadores. Cuando se empieza a trabajar en una tarea, el programador "pega" la tarjeta del tablero en su ordenador, dejando sus iniciales. Cuando la tarea se termina, se vuelve a colocar en el tablero y se marca en verde

Cada TASK tiene asociado un conjunto de TESTS (en el reverso de la tarjeta)



Sesión 12: Planificación de pruebas

¿ QUÉ ASPECTO TIENE UN PLAN EN SCRUM?



SPRINT (3-4 semanas)

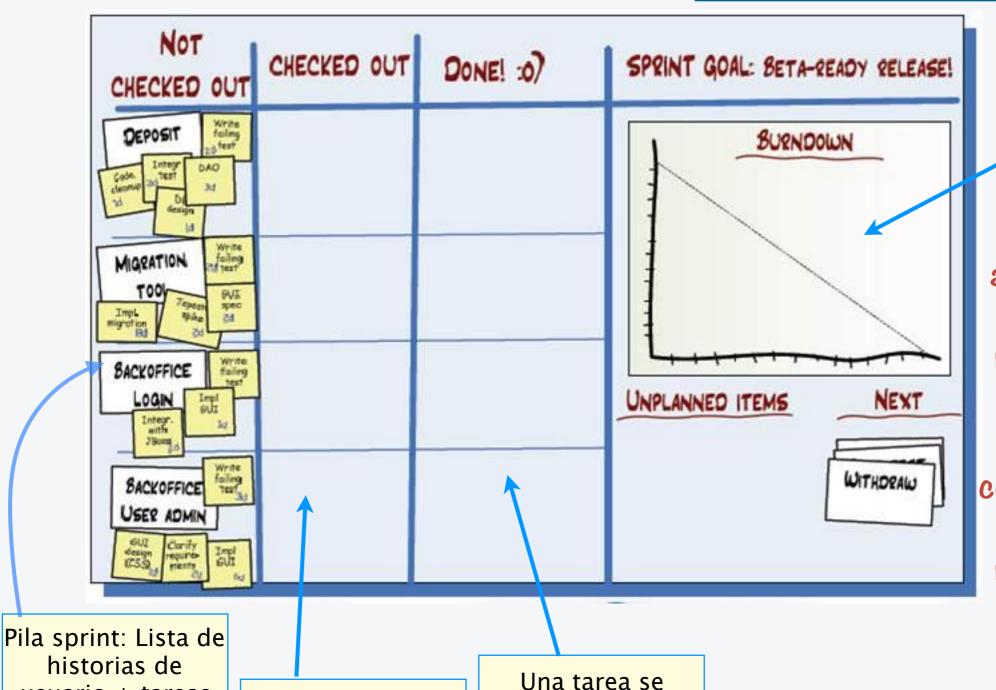


Gráfico de seguimiento

Cada TASK tiene asociado un conjunto de TESTS (en el reverso de la tarjeta)

Dichos tests
constituyen los criterios
de aceptación o
requerimientos de alto
nivel necesarios para
que la tarea pase al
estado DONE

usuario + tareas

Tareas en proceso

marca como DONE cuando pasa los tests



PRUEBAS Y DISEÑO: TUV





TDD (Test Development Driven) es una práctica de pruebas utilizada en

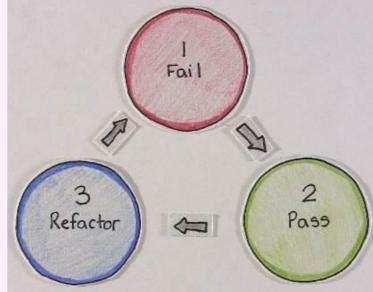
desarrollos ágiles, como por ejemplo XP



Paso 1. Escribir una prueba para el nuevo código y ver cómo falla

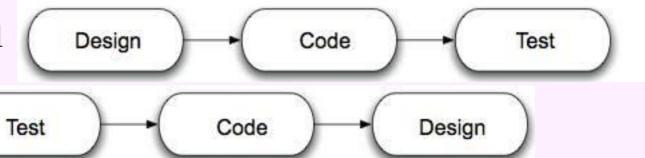
Paso 2. Implementar el nuevo código, haciendo "la solución más simple que pueda funcionar"

Paso 3. Comprobar que la prueba es exitosa y refactorizar el código

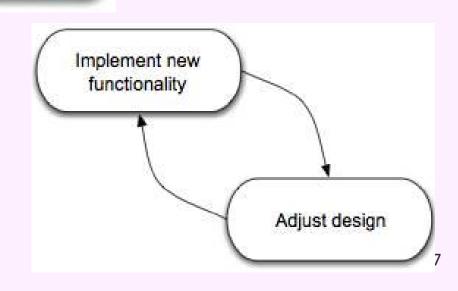


Con TDD el diseño de la aplicación "evoluciona" a medida que vamos desarrollando la aplicación

Aproximación tradicional



- Aproximación TDD
- El diseño de la aplicación se realiza de forma incremental ajustando la estructura del código en pequeños incrementos a medida que se añade más comportamiento.
 - En cualquier momento del desarrollo, el código exhibe el mejor diseño que los desarrolladores pueden concebir para soportar la funcionalidad actual



Ver http://xunitpatterns.com/ Philosophy%20Of%20Test %20Automation.html

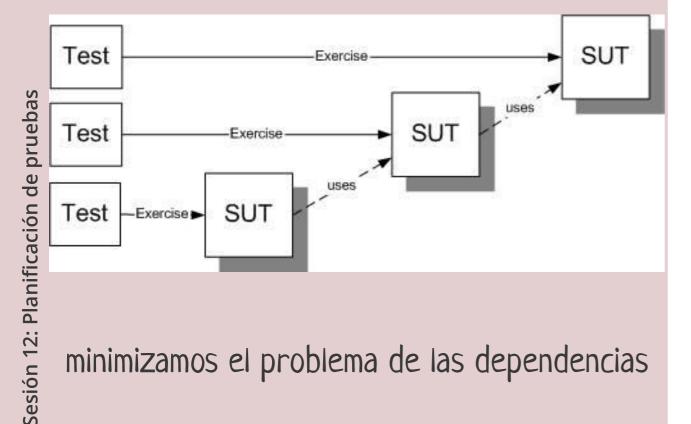
CLASSICAL VS MOCKIST TDD

Ver http://martinfowler.com/ articles/mocksArentStubs.html

La aproximación "clásica" de TDD consiste en utilizar objetos reales siempre que sea posible, y utilizar stubs (u otro "doble") si es "complicado" utilizar el objeto real.

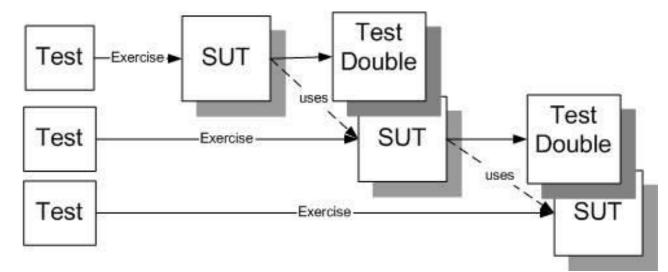
Por ejemplo, cuando estamos probando un método que como parte de su cometido debe enviar un correo

Esta aproximación soporta un estilo de diseño **INSIDE-OUT**: comenzamos por los componentes de bajo nivel. No necesitamos stubs



minimizamos el problema de las dependencias

- La aproximación "mockist" utilizará siempre un mock para cualquier objeto con un comportamiento interesante
 - Como consecuencia de esta aproximación surge lo que se denomina BDD (Behaviour Driven Development). Fué originalmente desarrollado por Dan North como una técnica para ayudar al aprendizaje de TDD, centrando su atención en cómo TDD contribuye a generar el diseño: "El diseño del sistema evoluciona a través de iteraciones a medida que se escriben los tests"
 - La aproximación "mockist" soporta un estilo de diseño OUTSIDE-IN: comenzamos por los componentes de alto nivel. Utilizaremos mocks para sustituir los componentes de capas inferiores



El diseño OUTSIDE-IN utiliza verificación basada en el comportamiento (no sólo es necesario especificar el estado inicial y final del objeto a probar, sino también la interacción con sus dependencias

INTEGRACIONES CONTINUAS (CI)







- El término "integración continua" nace con el proceso de desarrollo XP, como una de sus doce prácticas fundamentales
- Las integraciones continuas (**CI**: Continuous Integration) consisten en **INTEGRAR** el código del proyecto de forma ininterrumpida (en ciclos cortos) en una máquina aparte de la de cada desarrollador, la cual debe estar funcionando 24/7
- Si bien la práctica de CI no requiere de una herramienta específica, es habitual utilizar un Servidor de Integraciones Continuas para automatizar todo el proceso
- Las integraciones continuas realizan **CONSTRUCCIONES PLANIFICADAS** del sistema
 - Ejecutan el proceso de construcción (a partir de comandos Maven, por ejemplo) tantas veces como queramos y con la frecuencia que deseemos, sin mover ni un dedo.
 - Es importante que se ejecuten a **intervalos regulares lo más cortos posible**. Por ejemplo, supongamos que integramos el proyecto cada hora. Cada 60 minutos sabremos si nuestra construcción funciona o no. Esto hace que la búsqueda de errores sea más fácil (sólo hemos de mirar en los cambios que han ocurrido durante dicho intervalo). Además, estos problemas serán fáciles de resolver, porque en una hora no hemos tenido oportunidad de realizar grandes cambios que se habrían convertido en grandes problemas.

COMPONENTES Y FUNCIONAMIENTO DE UN SISTEMA DE CI



Imagen extraída de: "Continuous integration". Paul M. Duval. Addison Wesley (Capítulo 1)

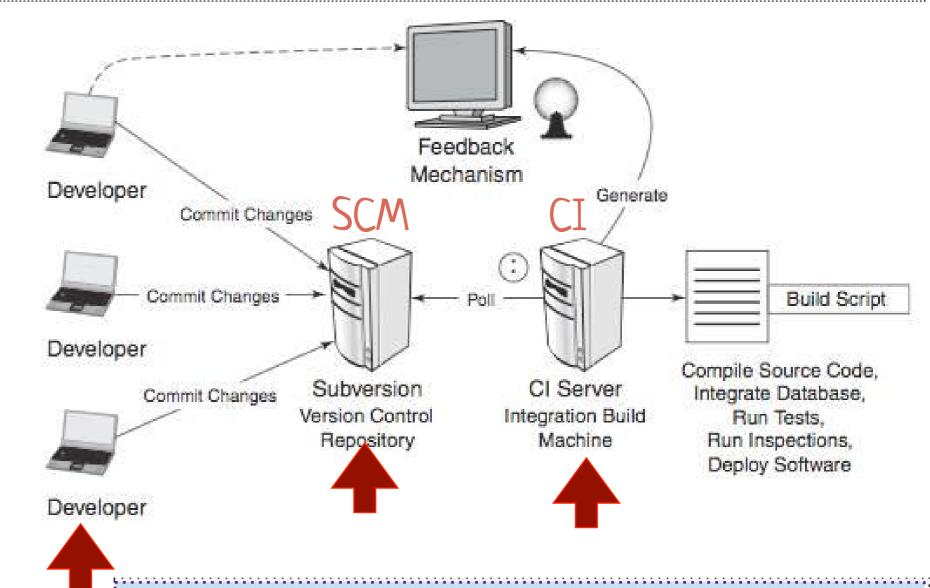
Funcionamiento.



 Un desarrollador "sube" el código en el que ha estado trabajando al repositorio SCM (después de hacer un build local), y sólo si las pruebas unitarias han "pasado".

Mientras tanto el servidor CI consulta el SCM para detectar cambios

- 2. El CI recupera la última versión del SCM y ejecuta un script de construcción del proyecto (construcción planificada)
- 3. El CI genera un feedback. En el momento en el que se "rompe" el sistema, hay que reparar el error



La integración continua no evita los fallos, pero reduce drásticamente el esfuerzo de encontrar los errores y repararlos.

Se agiliza el proceso de desarrollo mediante la automatización de las construcciones planificadas del sistema. La tarea de construir el sistema cada poco tiempo interfiere en el trabajo de los programadores (la construcción puede llevar desde unos pocos minutos a unas pocas horas). Un servidor de CI no tiene otra cosa mejor que hacer que construir el sistema, probarlo e informar de los resultados

SCM: System Configuration Manager

BDD: BEHAVIOR DRIVEN DEVELUPMENT

Hunting out value!!!!

Es muy importante entender por qué estamos llevando a cabo un proyecto (tener claros los objetivos del mismo)

Cualquier sistema software proporciona un VALOR de NEGOCIO.

Las expectativas del cliente dependen de dicho valor

BDD nos permite centrarnos en el valor de negocio de nuestra aplicación













BDD: HUNTING OUT VALUE





- Si preguntamos a los usuarios "qué quieren", generalmente obtendremos por respuesta un conjunto de requerimientos detallados sobre cómo imaginan la solución,
 - es decir, los usuarios no nos dicen lo que necesitan, diseñan una solución por nosotros
- Debemos centrarnos en las "features" (funcionalidades) que proporcionan un valor de negocio para el usuario,
 - es decir, funcionalidades que nos ayudan a conseguir los OBJETIVOS del proyecto
- Ejemplos de objetivos (goals):
 - Incrementar en un 10% el número de clientes proporcionando una forma sencilla de que ellos mismos gestionen sus cuentas
 - Incrementar las ventas de de libros en un 5% animando a los usuarios a que compren en nuestra tienda

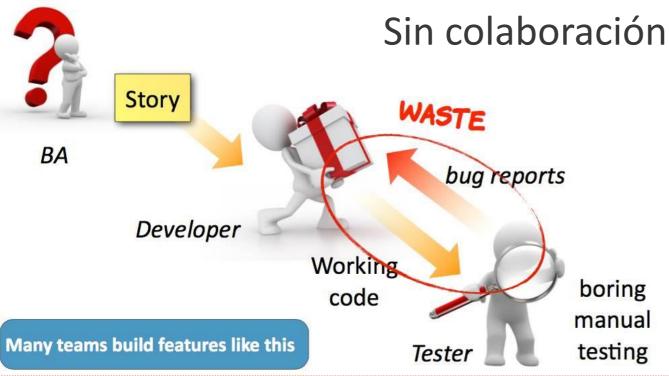
Behavior Driven Development, or BDD, is a set of software engineering practices designed to help teams build and deliver more valuable, higher quality software faster.

BDD Y LOS BENEFICIOS DE LA COLABORACIÓN

"a shared understanding"

We call this "The Three Amigos"

Comenzaremos aclarando exactamente cómo esperamos que sistema proporcione un valor de negocio, y focalizaremos el desarrollo en aquellas características que permitan maximizar la "rentabilidad" de nuestra aplicación



La colaboración permite identificar y compartir claramente los objetivos que "importan" (dan valor a nuestra aplicación) reduciendo el esfuerzo de desarrollo

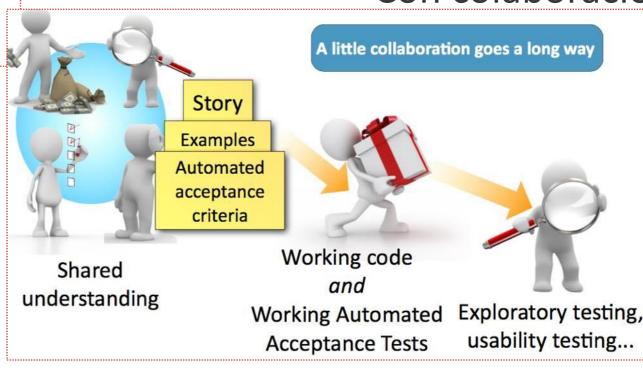
Shared understanding

Spanial Earning Standard Dollats from an Economy flight of Spanial Earning Standard points from an Economy fli

Con colaboración

Acceptance

Criteria

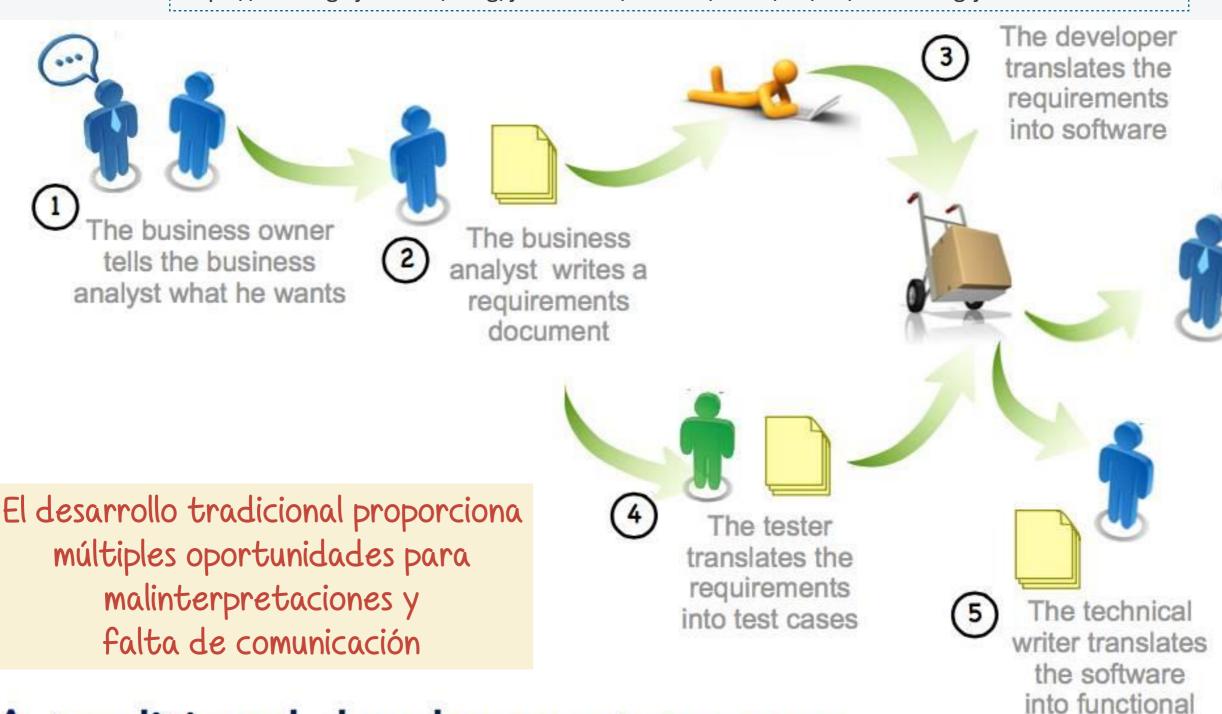


PROCESO DE DESARROLEO TRADICIONAL

A traditional development process

imagen extraída de "It's testing, Jim, but not as we know it". John Ferguson Smart. 2014.

https://weblogs.java.net/blog/johnsmart/archive/2014/06/25/its-testing-jim-not-we-know-it



24

and technical

documentation

PROCESO DE DESARROLLO CON BDD

imagen extraída de "It's testing, Jim, but not as we know it". John Ferguson Smart. 2014.

https://weblogs.java.net/blog/johnsmart/archive/2014/06/25/its-testing-jim-not-we-know-it

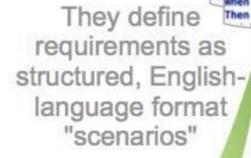
The business analyst, the developer and the tester elaborate the requirements together.





Sesión 12: Planificación de pruebas

The business owner and the business analyst have a conversation about what he needs.





The tester uses these scenarios as the basis for her tests

BDD utiliza conversaciones basadas en ejemplos, y expresadas de forma que sean fácilmente automatizables, reduciendo así la pérdida de



The automated tests provide feedback on progress and help document the application

información y los malentendidos

A BDD development process

BDD NO IMPLEMENTA TESTS UNITARIOS, IMPLEMENTA ESPECIFICACIONES DE BAJO NIVEL



Simplemente cambiando la forma de escribir los tests, centrándonos en lo que "debería" hacer el sistema, podemos escribir tests más "focalizados" en los objetivos concretos de dichos tests, convirtiéndolos así en "especificaciones". Por ejemplo, si pensamos en una aplicación bancaria:

```
public class BankAccountTest {
  @Test
  public void testTransfer() {...}
  @Test
  public void testDeposit() {...}
}
```

Test unitario "tradicional" (sin BDD)

VS.

Especificación de bajo nivel

```
Ejemplo de herramienta
que soporta BDD: Serenity
se consiente en
```

```
public class WhenTransferringInternationalFunds {
   @Test
   public void should_transfer_funds_to_destination_account() {...}
   @Test
   public void should_deduct_fees_as_a_separateTransation() {...}
   ...
}
```

Escenario

Given a customer has a current account **When** the customer transfers funds from this account to an overseas account **Then** the funds **should** be deposited in the overseas account **And** the transaction fee **should** be deducted from the current account

REFERENCIAS BIBLIOGRÁFICAS







- Software Testing: An ISTQB-ISEB Foundation Guide. Brian Hambling. British Computer Society; 2nd New edition. 2010
 - Capítulo 1
- Agile estimating and planning. Mike Cohn. Prentice Hall. 2006
 - Capítulo 3
- The art of agile development. James Shore. O'Reilly. 2008
 - Capítulo 3
- The Scrum primer. An introduction to project manager with scrum. Pete Deemer. 2007
 - http://www.scrumprimer.com/
- Continuous integration. Martin Fowler. 2006
 - http://www.martinfowler.com/articles/continuousIntegration.html
- BDD in action. Behavior-Driven Development for the whole software lifecycle. John Ferguson Smart. Manning. 2015
 - Capítulo 1