



# Metodologías de Desarrollo Java EE

## Sesión 4: Prácticas Ágiles TDD & CI



# Puntos a tratar

- Automatización
  - Ventajas
  - Cuando y Cómo
  - Tipos y Dispositivos
- Desarrollo Dirigido por las Pruebas (TDD)
  - Beneficios vs Costes
  - Automatización de Pruebas
  - Cómo Escribir las Pruebas
  - ¿Qué Tengo que Probar?
- Integraciones Continuas (CI)
  - Prácticas
  - Construcciones Planificadas
  - CruiseControl
    - Configuración y Marcha



# Automatización

- Durante el desarrollo de un proyecto, son innumerables las ocasiones en que tenemos la **necesidad de construir el proyecto**
  - para comprobar si los cambios realizados son correctos
  - si las pruebas implementadas son efectivas
  - si el sistema sigue funcionando como lo hacia hasta ahora...
- ¿Por qué automatizar?
  - Por que *el tiempo es oro*. Los procesos automáticos son más rápidos que los manuales.





# Ventajas de la Automatización



- La automatización ofrece precisión, consistencia y repetición.
- Cuando un ser humano realiza una tarea repetitiva acaba por aburrirse, pero el ordenador realiza estas tareas una vez tras otra sin quejarse ;)
- La automatización reduce la documentación
  - no es necesario explicar al resto de componentes (o futuros compañeros) del equipo de trabajo los pasos a seguir para realizar un determinado proceso, sino que se le enseña donde esta el script y como se ejecuta (y si está interesado, el script estará documentado explicando todas las tareas que realiza).
- La automatización cambia el modo de trabajar
  - no solo hace el trabajo mas fácil
  - permite ejecutar los procesos críticos del proyecto tantas veces como deseemos.

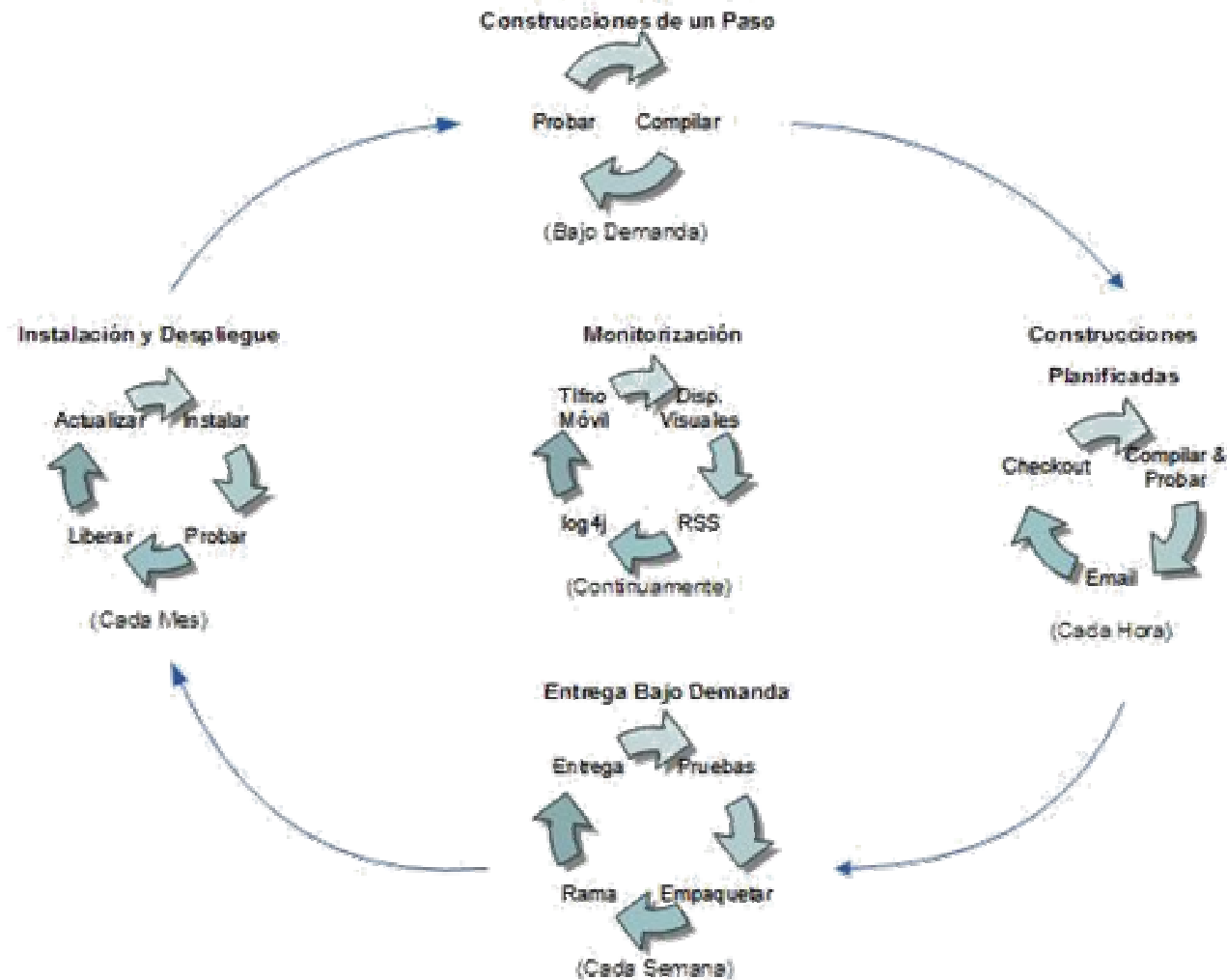


## ¿Cuando Automatizar?

- Cuando ejecutes el mismo proceso por segunda vez, éste debería ser automático. Raro sería que no hubiera una tercera vez.
- Los errores normalmente aparecen debido al aburrimiento, por lo tanto, si un proceso debe ser fiable y consistente, debe ser automatizado.
- *Nunca emplees más tiempo desarrollando una solución automática que el tiempo que te ahorrarías.*



# ¿Cuándo Ejecutamos Procesos Automáticos?





# Tipos de Automatización

- **Comandos**

- Cuando ejecutamos un comando y el ordenador realiza una serie de tareas de un modo consistente y repetible.

- **Planificadas**

- Una vez un comando automatiza un proceso, podemos incluirlo dentro de un sistema planificador, de modo que nadie tenga que ejecutar el comando de forma manual.

- **Basada en Eventos**

- Los comandos también se pueden ejecutar automáticamente cuando ocurre algún evento importante, por ejemplo, al subir un archivo al CVS.



# ¿Qué Necesitamos Para Automatizar?

- **Control de Versiones**

- Repositorio central con el código y documentación del proyecto, de modo que tenemos un único punto desde el cual construir el proyecto

*CVS, Subversion*

- **Pruebas Automáticas**

- Las pruebas que comprueban sus propios resultados incrementan la confianza del equipo en la aplicación.

*JUnit, Cactus*

- **Scripts**

- Para enseñarle al ordenador cómo automatizar los procesos.

*Shell scripts, Ant, Maven*

- **Dispositivos de Comunicación**

- Para ofrecer retroalimentación sobre los resultados de la construcción y las pruebas, pudiéndose realizar en múltiples y diversos dispositivos.

*Email, Wiki, Teléfono móvil*





# Puntos a tratar

- Automatización
  - Ventajas
  - Cuando y Cómo
  - Tipos y Dispositivos
- Desarrollo Dirigido por las Pruebas (TDD)
  - Beneficios vs Costes
  - Automatización de Pruebas
  - Cómo Escribir las Pruebas
  - ¿Qué Tengo que Probar?
- Integraciones Continuas
  - Prácticas
  - Construcciones Planificadas
  - CruiseControl
    - Configuración y Marcha



# Antecedentes de las Pruebas

- Mientras que todos los desarrolladores afirman la importancia que tienen las pruebas dentro de la producción de software de calidad...
- ...a casi nadie le hace gracia tener que probar su propio código, e incluso menos si lo tiene que hacer otra persona.
- XP ha cambiado la percepción que la comunidad tiene del proceso de pruebas, dando un nuevo aire al *"arte de las pruebas"*.
  - Se plantea que la escritura y ejecución de las pruebas deben ser el elemento central de los esfuerzos de desarrollo.
- Para la gran mayoría de desarrolladores, la realización de pruebas antes de XP era un proceso muy laborioso y pesado.
  - Se debía escribir un caso de prueba, preparar los datos a probar, documentar los resultados esperados antes de que cualquier prueba pudiera ejecutarse.
- XP aprovechó la automatización de los scripts para todas las pruebas relacionadas con el desarrollo, mediante la escritura de código para probar código
  - A los desarrolladores lo que nos gusta es escribir código y no docs.

# Test Driven Development

- El Desarrollo Dirigido por las Pruebas (*TDD*), es una de las técnicas ágiles más utilizadas
  - protege al proyecto de los cambios mediante un escudo de pruebas unitarias.
- El enfoque XP para las pruebas difiere de los procesos tradicionales en que **las pruebas se escriben antes de producir el código**.
- Una vez escrita la prueba, se debe escribir el menor y más simple trozo de código para pasar la prueba (ni más, ni menos).
- El escribir código que pasa la prueba hace que la prueba dirija el proceso de desarrollo, de ahí el término.
- La regla es **2x1** "probar dos veces, codificar una", en 3 pasos:
  1. Escribir una prueba para el nuevo código y comprobar como falla
  2. Implementar el nuevo código, haciendo *"la solución más simple que pueda funcionar"*
  3. Comprobar que la prueba es exitosa, y refactorizar el código.

**2x1**



## TDD y XP

- TDD traslada el proceso de pruebas a la primera plana de la atención del ingeniero
  - las pruebas que validan la implementación cumplen los requisitos
  - y el código escrito, por consiguiente, pasa las pruebas.
- TDD existía antes de XP.
  - No obstante, XP popularizó la práctica y ayudó a su extendida aceptación.



# Beneficios

- Los requisitos de las pruebas se consideran al inicio.
- No se omiten las pruebas, ya que se escriben primero.
- Escribir las pruebas aclara los requisitos.
- Escribir código de pruebas tiende a producir software mejor organizado.
- Se mejora la usabilidad de los interfaces
  - los desarrolladores se ven obligados a trabajar con el interfaz a probar.
- Validación de los cambios de forma inmediata
  - formando parte del proceso de construcción.
- Ofrece soporte a la refactorización.
- Se entrega un producto de mayor calidad al equipo de control de calidad,
  - las pruebas unitarias han eliminado gran parte de los fallos.



# Coste

- TDD incrementa de forma significativa la calidad del software entregado.
- La teoría de TDD es muy atractiva, pero la práctica tiene sus inconvenientes.
  - La escritura del código, tanto para añadir funcionalidad como para escribir pruebas, consume recursos muy valiosos para el proyecto → tiempo de un desarrollador.
  - Además, la implementación de un conjunto efectivo de pruebas automáticas no es una tarea trivial.
- TDD se puede asumir de forma efectiva mediante:
  - la aplicación de una correcta estrategia de inclusión de pruebas:
  - el uso de un framework adecuado para las mismas (*JUnit*, *Cactus*, *Mock Objects*, etc...)



# Coste → Factores a Considerar I

- Complejidad
  - Cada prueba, que forma parte de un conjunto de pruebas mayor, debe operar de forma aislada sin *efectos laterales*.
  - Este nivel de aislamiento es uno de los retos técnicos.
  - El uso de recursos comunes (BBDD) incrementa el acoplamiento entre las pruebas.
- Cobertura de las Pruebas
  - Debe existir una *estrategia de pruebas* para definir el ámbito y la distribución de éstas.
  - Un proyecto con una estructura pobre tiende a generar un *conjunto de pruebas "hinchado"*
    - los desarrolladores implementan un volumen excesivo de pruebas unitarias cuya cobertura se solapa.
  - Debemos definir una estrategia de pruebas en los inicios del proyecto y difundirla a todos los integrantes del equipo.



## Coste → Factores a Considerar II

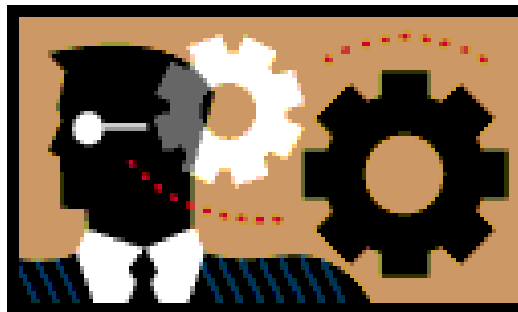
- Mantenimiento
  - Conforme crece el código de la aplicación, también lo hace el número de pruebas unitarias.
    - Los cambios en los requisitos y diseño de la aplicación implican, a su vez, actualizar un conjunto numeroso de casos de pruebas.
  - Aunque los beneficios pueden justificar esta sobrecarga de mantenimiento, el coste y el tiempo adicional necesarios deben tomarse en cuenta a la hora de planificar el proyecto.
- Proceso de Construcción
  - Un proceso automático y regular de construcción debe ejecutar todas las pruebas y reportar los errores.
  - El esfuerzo requerido para establecer y mantener las pruebas unitarias como parte del proceso de construcción también se debe tener en cuenta en el calendario del proyecto.





# Automatización de las Pruebas

Para que una prueba sea efectiva y repetible  
debe ser automática



- Al adoptar técnicas automáticas, no solo se reducen los tiempos del proyecto, sino también se aplican las mejores prácticas a la hora de las pruebas.



# Automatización → Ventajas I

- Precisión y Repetición
  - Los scripts de pruebas se ejecutan más de una vez a lo largo de un proyecto.
  - Incluso el mejor programador se equivoca, de modo que siempre se producen errores en el sistema.
    - Aunque el objetivo de la prueba sea ejecutarla una única vez para comprobar que todo funciona bien, normalmente se requieren varios ciclos de prueba-codificación.
  - Un sistema preciso se basa en la repetición de las pruebas entre los diferentes ciclos.
  - Las pruebas manuales requieren de mucho tiempo, y están sujetas a los errores humanos.
  - Una estrategia de prueba basada en procedimientos manuales conlleva el riesgo de introducir nuevos errores entre los diferentes ciclos de pruebas, los cuales no serían detectados.
    - una prueba automática que es 100% repetible evita este peligro.



## Automatización → Ventajas II

- Reducción de Tiempo de Entrega
  - El tiempo disponible para probar un sistema empresarial complejo siempre tiende a 0.
  - El tiempo disponible puede ser tan pequeño que la única solución factible, para los tiempos de entrega del cliente, sea la automatización de las pruebas de forma progresiva.
- Mejora de la Efectividad de las Pruebas
  - Ciertos tipos de pruebas, como las pruebas de carga y stress, son muy difíciles de conseguir sin herramientas automáticas.
- Desaparición de Tareas Rutinarias
  - Los encargados de las pruebas, como los desarrolladores, se aburren si realizan la misma tarea de forma repetida.
  - Con la ayuda de los generadores de código y los asistentes apropiados, se libera a los especialistas de realizar estas tareas y centrarse en otros aspectos del sistema.




# Automatización con peros...

- La adopción de pruebas automáticas no elimina completamente la necesidad de pruebas manuales.
  - Los expertos en pruebas pueden, y deben, continuar realizando pruebas manuales invasivas, “intentando romper el sistema”.
- Una estrategia efectiva de pruebas es aquella que combina las pruebas automáticas con las manuales
- ¿El tiempo necesario para escribir las pruebas automáticas justifica el esfuerzo que ahorraremos mediante la ejecución automática de éstas?
  - La automatización de las pruebas consume esfuerzos del proyecto → incurre en los costes económicos del mismo.
  - Estos costes se suelen recuperar en 2 o 3 ciclos de pruebas
    - Especialmente cuando incluimos los beneficios de la precisión y el consecuente incremento de la calidad.
    - A largo plazo el mismo script de prueba se utiliza en las fases de mantenimiento del proyecto.



# El Reto de las Pruebas JavaEE

- Una aplicación JavaEE distribuida presenta varios retos para el probador → **proceso complejo**.
- Las características distribuidas de una aplicación JavaEE
  - posiblemente en diferentes capas y niveles
  - con sus problemas relativos a la red
    - como pueden ser firewalls o políticas de seguridad de la empresa
  - puede interactuar con arquitecturas que usen mensajes síncronos
  - transacciones de negocio de larga duración que se ejecutan en sistemas heterogéneos
  - funcionalidades críticas para el negocio que se ofrecen a sistemas externos a través de una arquitectura orientada a Servicios Web
  - sin olvidar los atributos operacionales
    - seguridad, rendimiento, escalabilidad, robustez, ....
-  La tarea de probar una aplicación JavaEE es tan complejo para el probador como lo es implementarla para el programador.



# Herramientas de Automatización I

- Objetivo: simplificar el proceso de pruebas e incrementar la calidad de las pruebas realizadas.

## 1. Cobertura de Código

- Analizan el código base e informan de la profundidad y el alcance de las pruebas.

## 2. Medida de la Calidad

- Se ejecutan tanto de forma estática



# Herramientas de Automatización I

## 3. Generadores de Datos de Prueba

- Uno de los aspectos más pesados y difíciles es la generación de un conjunto de datos de prueba apropiado.
- Es especialmente difícil cuando se implementa una aplicación desde cero y no existe ningún sistema externo (tipo *legacy*) para probar el sistema.
- Estas herramientas generan los datos a partir del modelo de diseño, el esquema de la base de datos, esquemas XML y/o el código fuente de la aplicación.

## 4. Herramientas de Automatización de Pruebas

- Esta categoría general engloba las herramientas que ejecutan los script de prueba de una forma automática.
- Existen productos para la automatización de las pruebas, tanto unitarias como de integración, funcionales, de carga, stress, etc...



# Algunas Herramientas Gratuitas

Nombre	Descripción	Tipo	Referencia
Cactus	Framework de pruebas Unitarias y de Integración de la fundación Apache, el cual ofrece un modo de realizar pruebas de componentes JavaEE dentro del contenedor.	Unitarias / Integración	<a href="http://jakarta.apache.org/cactus">jakarta.apache.org/cactus</a>
HttpUnit	Ofrece una API Java para desarrollar un conjunto de pruebas funcionales para aplicaciones Web.	Unitarias / Funcionales	<a href="http://sourceforge.net/projects/httpunit">sourceforge.net/projects/httpunit</a>
JMeter	Aplicación cliente diseñada para comprobar la carga y el comportamiento de las pruebas.	Carga / Stress	<a href="http://jakarta.apache.org/jmeter">jakarta.apache.org/jmeter</a>
Grinder	Dirige las actividades de un script de pruebas en múltiples procesos y en diversas máquinas, mediante una consola gráfica.	Carga / Stress	<a href="http://grinder.sourceforge.net">grinder.sourceforge.net</a>
Emma	Librería de cobertura de código basado en consola/Ant que inspecciona el código y genera informes sobre el código probado y pendiente de probar.	Cobertura	<a href="http://emma.sourceforge.net">emma.sourceforge.net</a>





## ¿Dentro o Fuera del Contenedor?

- ¿Es necesario un framework como *Cactus* para probar código Java Enterprise?
- JavaEE mantiene una fuerte relación (*Servlets*, *JSPs*, *EJBs...*) con el contenedor.
- Las pruebas unitarias se centran en unidades de código de programación.
  - El código no existe aislado. Incluso el programa más sencillo depende de otras unidades de código (del mismo modo que cualquier programa Java depende de la JVM).
- Uno de los mayores retos dentro de las pruebas unitarias consiste en cómo "engañar" al código para que su comportamiento se evalúe de manera independiente al contexto.



# Mocks

- Es un objeto falso que simula el comportamiento de un objeto verdadero.
- Las pruebas mocks definen "objetos mock" que los casos de prueba pueden "pasar" como parámetros al código de negocio a probar.
  - Estos objetos suplantán a los objetos de negocio (implementan el mismo interfaz) y tienen un comportamiento simulado que los casos de prueba pueden configurar en tiempo de ejecución.
  - También se conocen como "maniquís de objetos" (*dummy objects*).
- Permiten refinar la práctica de las pruebas unitarias, ya que aseguramos total independencia del código respecto al contenedor.
  - En realidad, la creación de estados de aplicación independientes de la aplicación, en algunos casos es casi imposible, o demasiado costosa.
- Existen varios proyectos dedicados a la creación de objetos mock
  - EasyMock (<http://easymock.org/>)
  - DynaMock (<http://www.mockobjects.com>)
  - MockMaker (<http://mockmaker.sourceforge.net>).



# Pruebas Dentro del Contenedor

- Las pruebas integradoras, o dentro del contenedor J2EE, eliminan el problema de aislar las pruebas del contenedor y lo que hace es apoyarse en él.
  - Las pruebas integradoras van a probar el código del dominio desde dentro del contexto que ofrece el contenedor.
- Vamos a poder probar
  - los accesos a la Base de Datos utilizando el pool de conexiones que nos ofrece el servidor de aplicaciones.
  - el código de negocio de un bean de sesión a partir del contenedor EJB, pudiendo gestionar las transacciones.
  - los parámetros de entrada y salida que recibe un Servlet, así como los atributos que se almacenan tanto en la petición como en la respuesta.
  - etc...



# Supuesto de TDD

- Para aplicar TDD a un ejemplo real, vamos a seguir el supuesto del proyecto de integración, y vamos a desarrollar la persistencia de una reserva de un libro.
  - Para ello, las entradas serán los identificadores de usuario y de libro, así como las fechas de inicio y fin de la reserva.
  - Como salida, deberemos obtener un identificador de la reserva, así como comprobar que el estado del usuario ha cambiado a reserva.
- A continuación, se expone un posible enfoque de cómo escribir las pruebas mediante TDD en una serie de pasos.



# Cómo Escribir las Pruebas I

1. Piensa en lo que debería realizar el código a probar, y de momento, ignora como hacerlo.
  - Esto puede ser difícil para los programadores, porque se tiene la tendencia en pensar siempre en el "cómo".
  - Sin embargo, si se hace un esfuerzo, y se piensa primero en el qué y luego en el cómo, conseguiremos abstraer el problema.
2. Ahora implementa un prueba que utilice las clases y métodos que ¡todavía no se han implementado!
  - Esto también parece raro, pero funciona.
  - ¿Cómo escribir una prueba utilizando código que todavía no existe
  - Una vez determinadas las clases involucradas, la firma de los métodos y los valores de los parámetros, podemos escribir la prueba.



# Cómo Escribir las Pruebas II

```
public void testRealizaReserva() throws Exception {

    // Preparamos los datos de entrada
    String idUsuario = "profesor";
    String password = "profesor";
    String idLibro = "0131401572";

    Calendar cal = GregorianCalendar.getInstance();
    Date ahora = cal.getTime();
    Date ffin = DateUtils.addDays(ahora, 5);

    // Realizamos la operacion
    IOperacionDAO opDao = FactoriaDAOs.getInstance().getOperacionDAO();
    int numOp = opDao.realizaReserva(idUsuario, idLibro, ahora, ffin);

    // Comprobamos las salidas
    Assert.assertFalse("El número de la operación debe ser != 0",
        numOp == 0);

    IUserdaoDAO usuarioDao = FactoriaDAOs.getInstance().getUsuarioDAO();
    UsuarioTO unUsuario = usuarioDao.selectUsuario(idUsuario, password);
    Assert.assertEquals("El estado del usuario debería ser reserva",
        EstadoUsuario.reserva, unUsuario.getEstado());
}
```

## Cómo Escribir las Pruebas III

- La prueba no compilará, pero lo hará en breve.
  - Mediante herramientas como Eclipse, este proceso es sencillo.
- Primero generaremos la firma del método deseada en la clase o interfaz declarada.

The screenshot shows a code editor in Eclipse with the following code:

```
// Realizamos la operacion
IOperacionDAO opDao = FactoriaDAOs.getInstance().getOperacionDAO();
int numOp = opDao.realizaReserva(idUsuario, idLibro, ahora, ffin);

// Comprobamos las
Assert.assertFalse
    numOp == 0

IUsuarioDAO usuari
```

A context menu is open over the `realizaReserva` method call, showing the following options:

- Change method 'realizaReserva(UsuarioTO, LibroTO, Date, Date)' to 'realizaReserva(%)'
- Create method 'realizaReserva(String, String, Date, Date)' in type 'IOperacionDAO'
- Rename in file (Ctrl+2, R direct access)

- Podemos observar cómo ha generado la firma del método en el interfaz del objeto, ya que `opDao` es un interfaz.

The screenshot shows the definition of the `IOperacionDAO` interface:

```
public interface IOperacionDAO {

    public int realizaReserva(String idUsuario, String idLibro, Date ahora, Date ffin);

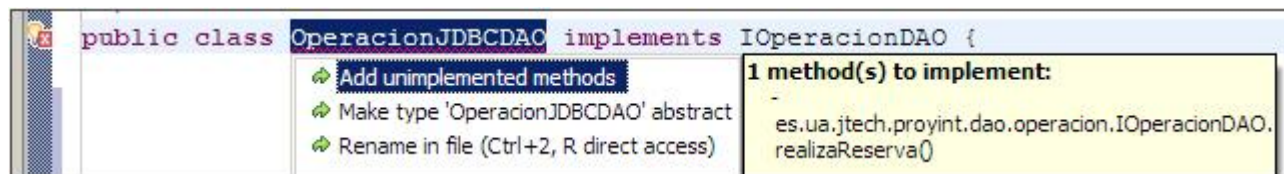
}
```



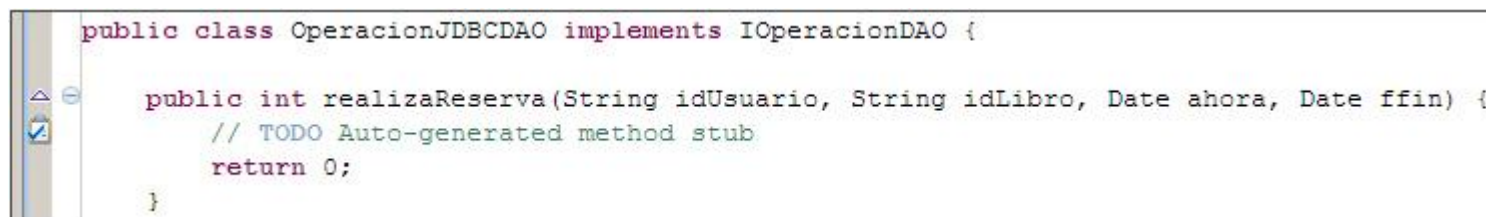


## Cómo Escribir las Pruebas IV

- Ahora queremos que la clase que implementa dicho interfaz también propague la firma del método
  - Mediante la implementación de los métodos que existan en la interfaz (IOperacionDAO) y no en la implementación (OperacionJBCDAO).
  - En el caso de tener varias clases que implementasen el interfaz, podemos utilizar la Jerarquía de Tipos (F4).



- Finalmente, obtenemos el esqueleto del método listo para implementar.

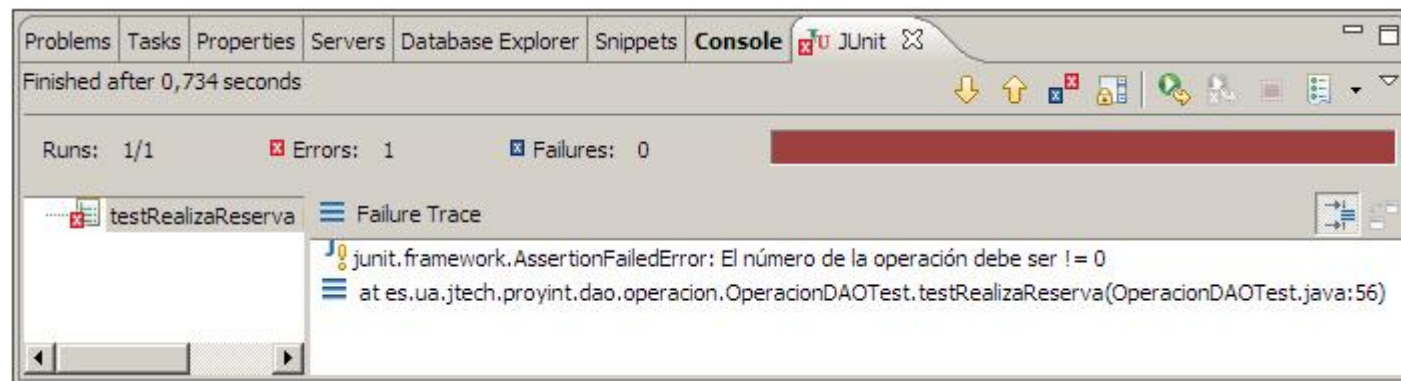






## Cómo Escribir las Pruebas V

3. Ahora la prueba ya compila, y la podemos ejecutar.
- Por supuesto, el resultado debe ser una prueba fallida, ya que todavía no hemos implementado nada.
  - Este es un paso importante, ya que aunque parezca redundante, realiza 2 funciones, primero valida que la prueba se ejecuta, y segundo que la prueba falla.
  - Si la prueba no hubiese fallado, entonces nuestro caso de prueba estaría mal y tendríamos que replanteárnoslo.
  - Nuestro objetivo final es que la prueba sea **verde**.





# Cómo Escribir las Pruebas VI

4. Cómo ya tenemos la prueba preparada, llega el momento de escribir el código del método a probar.
  - Partiendo del esqueleto generado, sabemos cuales son las entradas y las salidas.
  - Nuestro objetivo actual es escribir la menor cantidad de código que pase la prueba.
    - No es necesario que sea la solución definitiva, ya que una vez pasada la prueba, podremos refactorizar.
  - Resumiendo, hemos de buscar la solución más simple, sin pensar en posibles necesidades futuras.
- Conforme implementamos la solución, nos aparecerán problemas no planteados previamente, ya que ahora estamos pensando en el **cómo**
  - Por ejemplo, nos damos cuenta que no hemos lanzado la excepción DAOException en la firma del método, ya que nuestra prueba no estaba capturándola. Así pues, añadimos la excepción tanto en el interfaz como en la implementación.



# Cómo Escribir las Pruebas VII

```
public int realizaReserva(String idUsuario, String idLibro, Date ahora, Date ffin) throws DAOException {
    int result = 0;
    Connection conn = null;
    PreparedStatement st = null, stUsu = null;

    String sqlInsertOper = "insert into operacion(login, isbn, tipoOperacion, finicio, ffin) values (?, ?, ?, ?, ?)";
    String sqlUpdateUsuario = "update usuario set estadoUsuario=? where login=? ";

    try {
        conn = FactoriaFuenteDatos.getInstance().createConnection();
        conn.setAutoCommit(false);

        st = conn.prepareStatement(sqlInsertOper);
        st.setString(1, idUsuario); st.setString(2, idLibro); st.setString(3, TipoOperacion.reserva.toString());
        st.setDate(4, new java.sql.Date(ahora.getTime())); st.setDate(5, new java.sql.Date(ffin.getTime()));
        st.executeUpdate();

        ResultSet rs = st.getGeneratedKeys();
        if (rs.next()) {
            result = rs.getInt(1); // Obtenemos el id de la operación
        }
        rs.close();

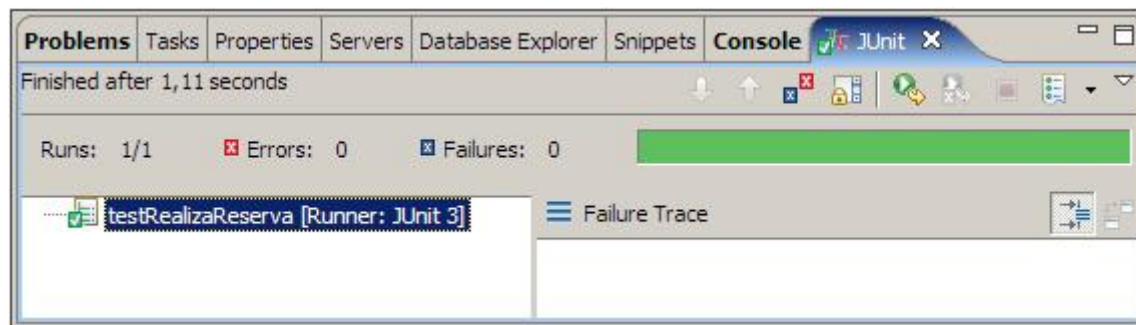
        stUsu = conn.prepareStatement(sqlUpdateUsuario);
        stUsu.setString(1, EstadoUsuario.reserva.toString()); stUsu.setString(2, idUsuario);
        stUsu.executeUpdate();

        conn.commit();
    } catch (SQLException sqle) {
        //...
        throw new DAOException("Error en el update de operacion", sqle);
    } finally {
        //...
    }
    return result;
}
```



## Cómo Escribir las Pruebas VIII

5. A continuación, volvemos a ejecutar la prueba, con un resultado exitoso.
  - En el caso de que la prueba no fuera verde, debemos revisar nuestra implementación.





## Cómo Escribir las Pruebas IV

6. Para comprobar que todo nuestro sistema sigue funcionando, ahora es el momento de ejecutar toda la suite de pruebas de la aplicación, incluyendo la prueba recién implementada.
  - De este modo, comprobamos que todo el sistema sigue funcionando.
  - En el caso de encontrar alguna prueba fallida, deberemos volver al paso 4.

En ocasiones, una prueba exitosa puede sacar a relucir una prueba mal diseñada o incompleta, de modo que el sistema falle en otra parte, pero no porque la prueba exitosa sea mala, sino porque existían pruebas que no comprobaban todos los posibles valores de entrada y/o salida.



# Cómo Escribir las Pruebas X

7. Finalmente, llega el momento de refactorizar.
    - Después de refactorizar, volveremos al paso 5 para ejecutar de nuevo las pruebas del método en cuestión y luego toda la suite.
- 
- Puede que parezca pesado tener que ejecutar las pruebas una y otra vez, pero hemos de **ser constantes** y no dejar este hábito, ya que la filosofía de "**hacer un cambio y ejecutar una prueba**" nos conducirá a un producto estable.



# ¿Qué Tengo que Probar?

1. Escribe pruebas por cada tarea a implementar, ya que "***el código es culpable hasta que se prueba su inocencia***".
2. Escribe pruebas por cada clase o combinaciones de clases que no sean triviales y que pueden provocar problemas. Es decir, **prueba todo lo que pueda romperse**.
3. Escribe **pruebas para el código que se ha roto** o que no funciona como se espera de él.
4. Evita las pruebas de métodos que sólo llaman a otro método (métodos *delegadores*) si el destino ya tiene su prueba.
5. Asume que **muchas pruebas es mejor que pocas**, y que nadie se quejará de que existan demasiadas pruebas. En cambio, si faltan pruebas seguro que alguien se enfada.
6. Escribe pruebas que inculquen **confianza** en el sistema. Aquellas áreas de código que sean más utilizadas, deberán ser las que tengan más pruebas.
7. Añade pruebas a las áreas **recién modificadas**. Tras refactorizar, comprueba que existen pruebas que comprueban que todo lo rediseñado sigue funcionando.
8. Vigila que las **suites de pruebas abarquen todos los subsistemas** que dependan de ellas.



# Conclusión

- No se espera hacer todas las pruebas de una sola vez, y que conforme crezca el código de aplicación, crecerá el código de pruebas.
- Una vez identificada la necesidad de una prueba, hay que codificarla y ejecutarla.
- Si la prueba falla, bien la prueba es incorrecta, bien el código a probar tiene un error.
  - En ambos casos, debemos corregir el problema y volver a probar.
- Ahora ya sólo nos queda cambiar el chip, y aplicar TDD en todos nuestros desarrollos.
- Como le dirían a *Luke TDD Skywalker*,  
**"que las pruebas te acompañen".**







# Puntos a tratar

- Automatización
  - Ventajas
  - Cuando y Cómo
  - Tipos y Dispositivos
- Desarrollo Dirigido por las Pruebas (TDD)
  - Beneficios vs Costes
  - Automatización de Pruebas
  - Cómo Escribir las Pruebas
  - ¿Qué Tengo que Probar?
- Integraciones Continuas (CI)
  - Prácticas
  - Construcciones Planificadas
  - CruiseControl
    - Configuración y Marcha



# Integraciones Continuas

- El término de Integraciones Continuas (*Continuous Integration*) proviene de una de las prácticas de XP (*Programación eXtrema*)
  - lo que no quiere decir que no existiera antes, ya que se utiliza desde mucho antes (incluso Microsoft mediante el trabajo de McConnell lo utiliza en sus desarrollos).
  - Pese a provenir de XP podemos utilizarlo sin hacer lo mismo con otras técnicas que promueve XP, ya que se trata de una técnica autónoma y esencial dentro de la actividad de cualquier equipo de desarrollo competente.
- Las Integraciones Continuas consisten en integrar el código del proyecto de forma ininterrumpida (en ciclo de 15 a 30 minutos)
  - En una máquina aparte a la de cada desarrollador, normalmente llamada entorno de desarrollo/integración, la cual debe estar funcionando 24/7.
  - Esta máquina descargará el código del proyecto del repositorio de control de versiones, construirá y probará el proyecto, para finalmente mostrar los datos obtenidos (fallos de construcción, pruebas fallidas) vía web o email a los integrantes del proyecto.



# Prácticas I

- Mantener un único repositorio centralizado
  - Incluso en los proyectos más sencillos, **es imprescindible el uso de un sistema de gestor de versiones**, donde almacenar las diferentes revisiones de toda información relevante para el proyecto (archivos de código fuente, imágenes, documentos, etc...)
  - Lo único que no se "suele" guardar en el repositorio es todo aquello que puede ser generado a partir de la información del repositorio
    - Es decir, ejecutables, war/ear del proyecto, etc...
  - Siendo CVS la herramienta más utilizada, la mejor elección actual es Subversion.
- Automatizar la construcción
  - Independientemente de utilizar un IDE, el cual construya el proyecto de forma automática y permita la generación de un desplegable, siempre hemos de tener un script (en nuestro caso Ant) independiente de la herramienta el cual se pueda ejecutar en otra máquina (por ejemplo, en el servidor), sin necesidad de tener instalado ningún IDE.



## Prácticas II

- Hacer que el proceso de construcción se auto-pruebe.
  - El propio script de construcción, además de compilar y crear un desplegable, debe ejecutar las pruebas, tanto unitarias, como a poder ser, de aceptación.
- Todo el mundo realiza *commits* a diario.
  - Ya que las integraciones comunican los cambios de los integrantes, la frecuente integración implica una buena comunicación.
  - A mayor de frecuencia de *commit*, menor es el margen de error
    - ya que todo el equipo comparte las últimas modificaciones del código.
  - Se fomenta que los desarrolladores implementen el código mediante pequeñas funcionalidades.
  - Por ultimo, respecto a la interacción repositorio<->IDE, siempre recordar el ciclo de *update* → *build* → *commit*.



## Prácticas III

- Cada *commit* debe construir el proyecto completo en una máquina de integración.
  - Mediante los *commits* diarios, aseguramos que las construcciones están continuamente probadas e integradas, pero en las máquinas locales.
  - No todos los entornos son iguales, y las diferencias entre la máquina de desarrollo y el de integración/producción, en numerosas ocasiones provoca errores no descubiertos previamente.
    - Para poder probar la aplicación en una máquina aparte de integración:
      - creamos un script manual de modo que cada cierto tiempo (idealmente cada vez que subamos un cambio al repositorio) descarguemos todo el código del repositorio y realicemos la integración, construcción y pruebas en un entorno completamente igual (tanto a nivel de hardware como de sistema operativo) al de producción
      - o instalamos un servidor de Integraciones Continuas.
  - No confundir la integración continua con ejecutar un script nocturno que realice las tareas anteriores.
    - La diferencia estriba en que con CI al producirse el error, inmediatamente se procede a su resolución.



# Prácticas IV

- Mantener el proceso construcción rápido.
  - Dentro de las CI está implícito que el proceso de construcción sea veloz.
  - Lo normal son procesos de construcción del orden de minutos (5-10), donde las pruebas absorben la mayoría del tiempo.
  - Cuando la duración del proceso sea del orden de horas, lo conveniente es dividir el proceso en 2 partes
    1. Cada desarrollador en local realiza una construcción básica con las pruebas rápidas, y las que están relacionadas con la parte actual de desarrollo
    2. Segundo proceso de construcción completo que pruebe toda la aplicación.
  - En ocasiones, las pruebas no tienen porque ejecutarse todas de modo secuencial, de modo que podemos crear procesos de construcción paralelos que disminuyan los tiempos de construcción.
- Despliegue automatizado.
  - Es muy importante automatizar el despliegue, tanto la construcción como las pruebas.
  - Últimamente se le esta dando importancia a la posibilidad de hacer un *rollback automático* de un despliegue
    - Si ponemos una aplicación en producción, y comienza a fallar, el tiempo necesario para dejar la aplicación en un estado anterior estable sea mínimo.



# Prácticas V

- Probar en un clon del entorno de producción.
  - Ya lo hemos comentado antes, mismo hardware y sistema operativo, pero incluso misma ip, mismo puerto, etc... Actualmente, es uso de la virtualización (por ejemplo, vía VmWare, VirtualPC) está facilitando la clonación de los entornos de producción.
- Facilitar la obtención de la última versión desplegable.
  - Todo el mundo relacionado con el equipo debe ser capaz de poner en funcionamiento al última versión de la aplicación. Para ello, debe haber un lugar conocido donde residirá la última versión del desplegable.



# Prácticas VI

- Todo el mundo puede ver lo que esta pasando.
  - En todo momento, la información del proyecto es pública, tanto los resultados de la construcción, como las pruebas.
  - A parte de la publicación en la web/wiki del proyecto, ya existen en el mercado diferentes aparatos para visualizar los resultados (semáforos, lámparas de lava, etc...) de forma explicita.







# Construcciones Planificadas

- Toman el fichero de construcción de la aplicación (el fichero *Ant*) y lo ejecutan por nosotros tantas veces queramos y con la frecuencia que deseamos, sin necesidad de mover ni un dedo.
  - Además, si queremos, podemos ejecutarlas de modo manual.
- Encuentran los problemas de integración (*tiempo de compilación*) y los fallos de pruebas (*tiempo de ejecución*) de forma rápida, ya que se están ejecutando a intervalos regulares.
  - Por ejemplo, si el planificador dice que tiene que ejecutar una construcción al inicio de cada hora, entonces nosotros sabremos cada 60 minutos si nuestra construcción funciona o no.
- La búsqueda de errores es más fácil, ya que solo 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 la oportunidad de realizar grandes cambios que se habrían convertido en grandes problemas.



# Necesidad de Construcciones Planificadas

- ¿Qué diferencia hay entre una construcción planificada y, digamos, todos los programadores ejecutando el fichero de construcción cada pocos minutos?
  - La verdad es que no existen muchos programadores que quisieran hacerlo, normalmente tienen mejores cosas que hacer.
  - El ciclo de construcción puede llevar desde unos pocos minutos a unas pocas horas, y la ejecución de la construcción interfiere con su trabajo.
  - Incluso si alguien del equipo de desarrollo pudiera ejecutar la construcción de un modo ágil y rápido, en algún momento dejaría de hacerlo
    - siempre hay una fecha de entrega próxima y el hecho de incluir un cambio en su sistema puede provocarles conflictos que cause retrasos en la entrega.
  - Los programadores normalmente sólo construyen las partes del sistema en las que están trabajando y no el sistema entero e integrado.



# Coste y Herramientas

- Una construcción planificada no tiene otra cosa mejor que hacer que construir el sistema y probarlo.
- Una vez tenemos un proceso de construcción de la aplicación, al poner este proceso en un planificador para que un ordenador lo ejecute de forma automático, todo van a ser ventajas.
- El **coste** de estas planificaciones solo se centran en el ordenador integrador y la preparación/configuración/arranque del sistema en los primeros días del proyecto (iteración 1).
  - Acabará costando muchísimo más al final si no empezamos a planificar las construcciones al principio.
- Para facilitarnos el trabajo, existen multitud de **herramientas** que nos ayudan en el proceso de integraciones continuas.
  - *CruiseControl*
  - *Continuum* (<http://maven.apache.org/continuum/>)

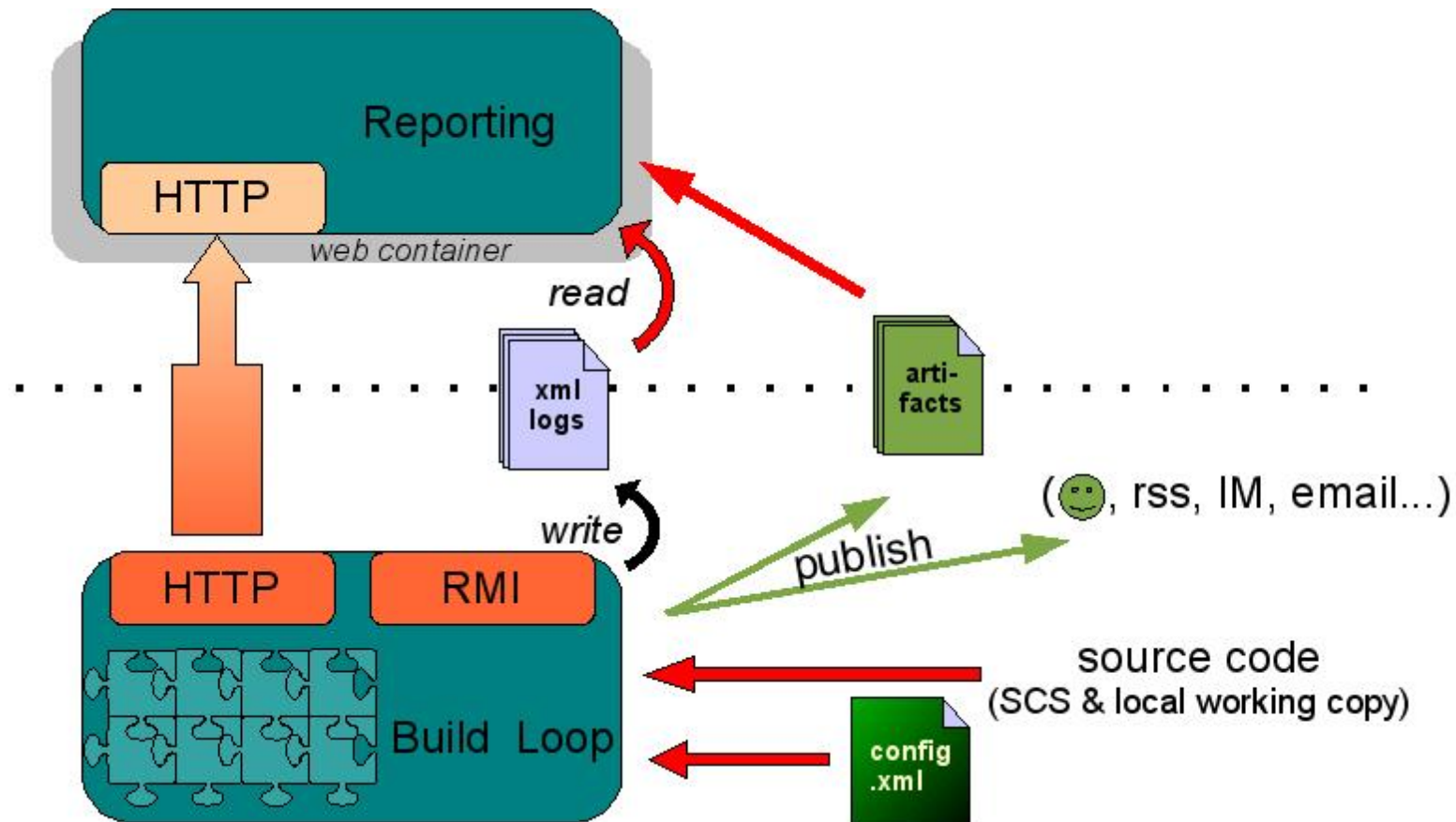


# CruiseControl



- *CruiseControl* (CC) (<http://cruisecontrol.sourceforge.net>) asiste al desarrollador en las Integraciones Continuas.
- Herramienta Java Open Source de construcción automática que mediante los scripts *Ant* y el sistema de Control de Versiones, asegura que el proyecto está en continua integración.
- *CruiseControl* se basa en un concepto muy sencillo.
  1. Una vez arrancada una instancia de CC, ésta comprueba el repositorio de control de versiones, y detecta cualquier cambio en el mismo.
  2. Al detectar un cambio, CC actualiza la copia local del proyecto, e invoca al script de construcción del mismo.
  3. Tras completar la construcción, CC publica una serie de artefactos (incluyendo el log de construcción) e informa a los miembros del proyecto del éxito o fallo de la construcción.

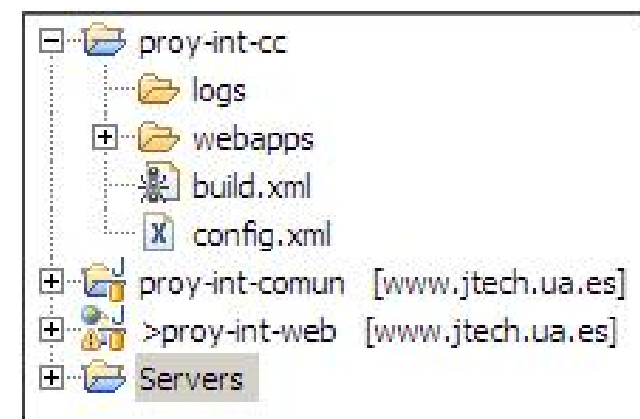
# Arquitectura CC





# Configuración - Eclipse

- Antes de montar CC, hemos de pensar donde montarlo.
- CruiseControl no necesita una máquina muy potente (ni mucho menos), solo se va a encargar de compilar y ejecutar las pruebas.
  - Cualquier ordenador de sobremesa nos vale como entorno de desarrollo/integración.
- Una forma cómoda de montar este entorno es crear un proyecto Eclipse con la configuración de CC de modo que orqueste la creación y ejecución de las pruebas del resto de proyecto.





## Configuración – logs y build.xml

- Crearemos una carpeta `logs` para almacenar los logs que posteriormente mostrará CC
- Y un fichero de construcción (`build.xml`) Ant encargado de la preparación de la copia local, descargar el contenido existente en el repositorio y posteriormente realizar una llamada a la tarea de construcción del proyecto.

```
<project name="proy-int-cc" default="buildCC" basedir=".">
<target name="buildCC">
    < cvs cvsroot=":extssh:USUARIO@www.jtech.ua.es:\usr\local\cvs-
jtech\USUARIO" quiet="true" command="co proy-int-comun" />
    < cvs cvsroot=":extssh:USUARIO@www.jtech.ua.es:\usr\local\cvs-
jtech\USUARIO" quiet="true" command="co proy-int-web" />
    <ant antfile="proy-int-web/build.xml" inheritall="false" target="test" />
</target>
</project>
```



# Configuración – config.xml

```
<cruisecontrol>
  <project name="proy-int-cc" buildafterfailed="false">
    <!-- Comprueba los cambios en el CVS
      modificationset: donde mirar para ver si ha habido cambios -->
    <modificationset quietperiod="30">
      <cvss localWorkingCopy="../proy-int-comun" />
      <cvss localWorkingCopy="../proy-int-web" />
    </modificationset>
    <!-- Frecuencia de búsqueda de cambios -->
    <schedule interval="30"> <!-- <pause startTime="2100" endTime="0300" /> -->
      <ant buildfile="build.xml" target="buildCC" />
    </schedule>
    <!-- Obtenemos la ultima versión
      bootstrappers: cosas que hacer antes del ciclo de construcción -->
    <bootstrappers>
      <currentbuildstatusbootstrapper file="logs/currenbuild.txt" />
    </bootstrappers>
    <!-- Guardamos los ficheros de log -->
    <log dir="logs"><merge dir="../proy-int-web/build/test-results" /></log>
    <publishers>
      <currentbuildstatuspublisher file="logs/currentbuildstatus.txt" />
    </publishers>
  </project>
</cruisecontrol>
```

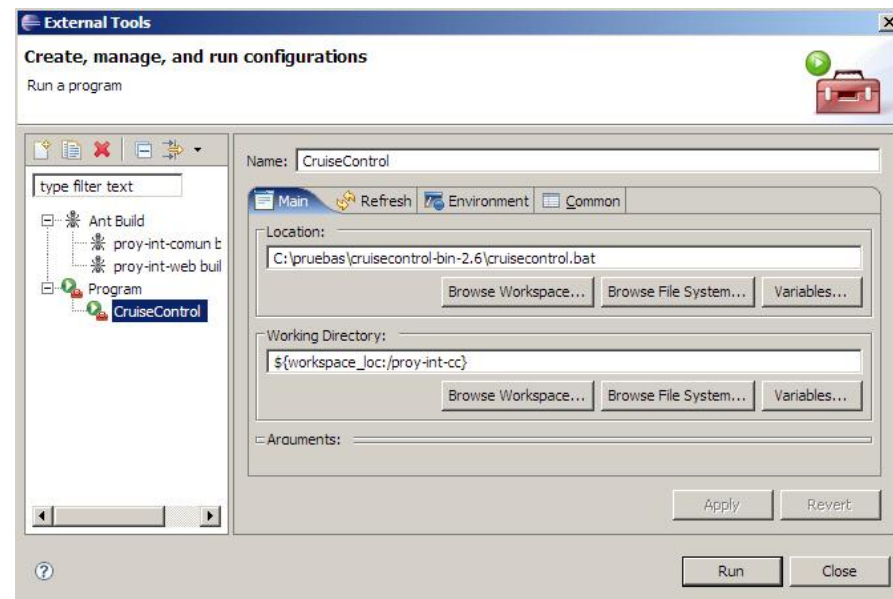




## En Marcha

- Una vez configurado el proyecto, llega el momento de arrancar *CruiseControl* y que se generen las integraciones continuas .
- Ejecutaremos el comando de CC desde nuestra carpeta de *CruiseControl*
  - en nuestro caso, desde dentro del proyecto `proy-int-cc`
- En Eclipse, la mejor forma de realizar esto es configurar la ejecución de una herramienta externa.

Para ellos, la crearemos desde Run --> External Tools, referenciando a la ruta donde tenemos instalada la herramienta.





# Resultados

- Si queremos visualizar los resultados, podemos acceder a <http://localhost:8080/cruisecontrol/>

CruiseControl Status Page				
<a href="#">Turn autorefresh off</a>				
Project	Last build result	Last build time	Last successful build time	Last label
<a href="#">bibliotecacc</a>	failed	15/05/2006 19:49:24	15/05/2006 19:21:41	build.1
Total	1			
Failed	1	100%		
listing generated at 19:53				

CruiseControl Status Page				
<a href="#">Turn autorefresh off</a>				
Project	Last build result	Last build time	Last successful build time	Last label
<a href="#">bibliotecacc</a>	passed	15/05/2006 19:53:19	15/05/2006 19:53:19	build.2
Total	1			
Passed	1	100%		
listing generated at 19:54				



# Detalle de la Construcción

**cruisecontrol**  
continuous integration toolkit

Project  
bibliotecacc

Latest Build  
16/05/2006 22:14:47 (build.11)  
16/05/2006 20:59:14 (build.10)  
16/05/2006 20:01:47  
16/05/2006 20:01:41  
16/05/2006 18:48:53 (build.9)  
16/05/2006 18:48:39 (build.10)  
16/05/2006 18:45:34 (build.9)  
16/05/2006 18:43:32  
15/05/2006 22:13:58 (build.8)  
15/05/2006 22:05:56 (build.7)  
15/05/2006 22:03:11 (build.6)

Build Results | Test Results | XML Log File | Metrics | Control Panel

**BUILD COMPLETE - build.11**  
Date of build: 05/16/2006 22:14:47  
Time to build: 41 seconds  
Last changed: 05/16/2006 22:14:08  
Last log entry: pintando listado de libros existentes  
[Build Artifacts](#)

**Errors/Warnings: (2)**  
BibliotecaSRC: C:\pruebas\bibliotecaCC\BibliotecaSRC\build.xml  
BibliotecaWeb: C:\pruebas\bibliotecaCC\BibliotecaWeb\build.xml

**Unit Tests: (7)**  
All Tests Passed

**Modifications since last successful build: (1)**

modified	aitormedrano	WebContent/existentes.jsp	pintando listado de libros existentes
----------	--------------	---------------------------	---------------------------------------

**Deployments by this build: (1)**  
Building jar: C:\pruebas\bibliotecaCC\BibliotecaWeb\dist\BibliotecaWeb.war



# Resultados de los Test

Build Results   Test Results   XML Log File   Metrics   Control Panel			
Name		Status	Time(s)
.es.ua.jtech.j2ee.dp.AllSuite			
testSelectLibros		Success	0.953
testSelectLibro		Success	0.047
testSelectUsuario		Success	0.047
testSelectNoUsuario		Success	0.031
testLogin		Success	3.766
testLoginKO		Success	0.188
testListarLibrosExistentes		Success	0.234
<a href="#">Properties »</a>			



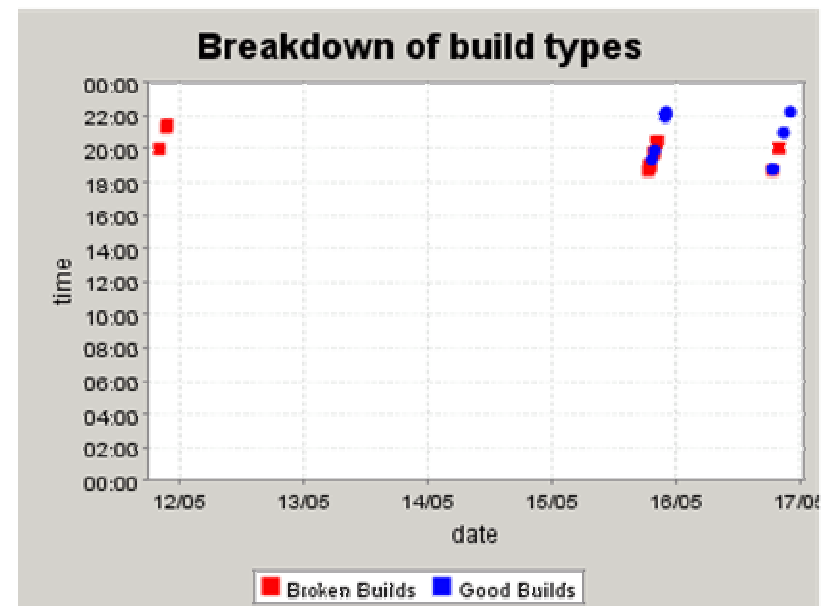
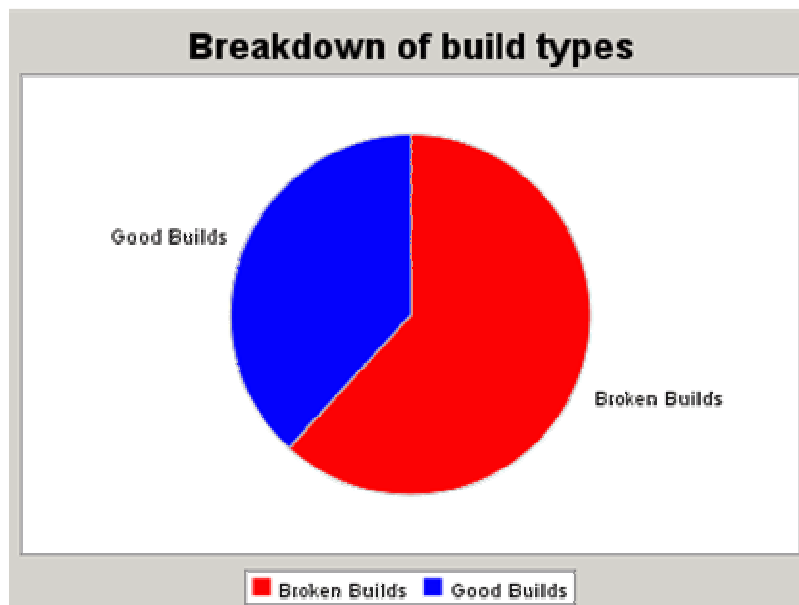
# Métricas

Build Results	Test Results	XML Log File	Metries	Control Panel
---------------	--------------	--------------	---------	---------------

Number of Build Attempts 34

Number of Broken Builds 21

Number of Successful Builds 13





# Componentes I

- Sistemas de Control de Versiones
  - CC utiliza estos sistemas para detectar los cambios en los proyectos. Estos componentes se fijan en el elemento `<modificacionsset>` del fichero de configuración CC.
  - Por ejemplo: *BuildStatus*, *ClearCase*, *CVS*, *FileSystem*, *SVN*, ...
- *Bootstrappers*
  - Dentro de la informática, el termino "*bootstrapper*" se utiliza para describir un proceso que se realiza para permitir la ejecución de otro proceso.
  - Así pues, estos componentes actualizan todo o parte del proyecto antes de que comience de forma oficial la construcción del mismo.
  - Se crearon para resolver un problema muy concreto:  
Si se utiliza el script de construcción para obtener la última versión del proyecto, ¿qué pasa si modificamos ese script de construcción ?
  - Por lo tanto, utilizaremos este componente para actualizar, al menos, el script de construcción. *Bootstrappers* también ofrecen un modo eficaz de realizar cualquier otra actividad de "**pre-construcción**".
  - Ejemplos de *bootstrappers* serán: *ClearCaseBootStrapper*, *CVSBootStrapper*, *SVNBootStrapper*, ...



# Componentes II

- Builders
  - Estos son los componentes que utiliza CC para construir realmente el proyecto.
  - Además de construir el sistema en intervalos regulares, podemos ejecutar los *builders* en tiempos y fechas concretas, mediante el uso de los atributos *multiple*, *time* y *day*.

```
<schedule interval="30" >
  <ant antscript="build.xml" target="cruise-build" multiple="1" />
  <ant antscript="build.xml" target="full-cruise-build" multiple="5" />
  <ant antscript="build.xml" target="nightly-cruise-build" time="0830" />
  <ant antscript="build.xml" target="weekly-cb" time="0305" day="Saturday" />
  <pause startTime="2100" endTime="0300" />
</schedule>
```

- En este ejemplo, cada 5 construcciones se ejecutará la tarea full-cruise-build.
- Las tareas de *nightly* y *weekly* ofrecen todavía más flexibilidad.
- Estas tareas son adecuadas para procesos que no se desea ejecutar en cada construcción
  - probablemente porque tardan mucho en completarse



# Componentes II

- Publishers
  - Los publicadores se ejecutan una vez finalizada la construcción, y tras haber escrito en el log de resultados. La intención de estos componentes es publicar los resultados de la construcción.
  - Ejemplos de publishers son: ArtifactsPublishers, Email, Execute, HTMLEmail, SCP, XSLTLogPublisher, FTPPublisher, ...

Ya sólo queda animarte a incluir una herramienta de Integraciones Continuas en tus proyectos, ya que se trata de un elemento fundamental para evitar problemas en el futuro.

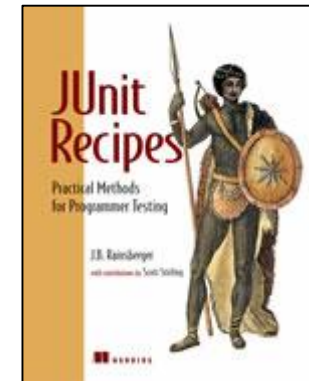
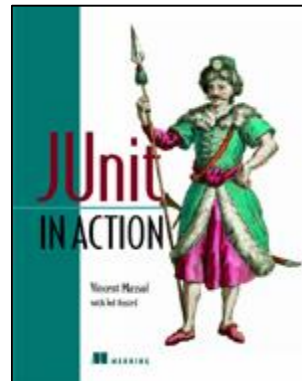
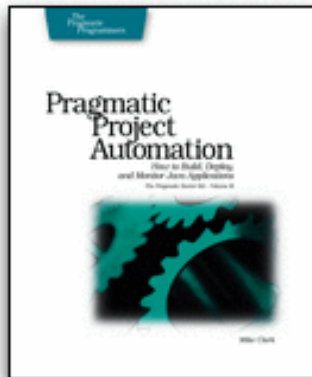




# Para Saber Más

- **Bibliografía**

- ***Pragmatic Project Automation*** de Mike Clark
- ***JUnit in Action*** de Vincent Massol
- ***JUnit Recipes*** de J. B. Rainsberger



- **Enlaces**

- Integraciones Continuas:  
[www.martinfowler.com/articles/continuousIntegration.html](http://www.martinfowler.com/articles/continuousIntegration.html)
- TDD y Modelado Ágil: [www.agiledata.org/essays/tdd.html](http://www.agiledata.org/essays/tdd.html)



¿Preguntas...?