Leonardo Alfredo Velazquez Luna

1. EVOLUCIÓN JAVA

1.1. ¿Qué es Java?

Java es un lenguaje de programación de propósito general que fue creado por Sun Microsystems en 1995 (ahora propiedad de Oracle Corporation). Java se destaca por ser un lenguaje orientado a objetos y por su capacidad de ejecutarse en múltiples plataformas, lo que significa que el código Java puede ejecutarse en diversos sistemas operativos así como computadoras personales, dispositivos móviles, servidores y sistemas embebidos, todo esto sin necesidad de modificaciones significativas. Esto se logra a través del uso de la máquina virtual Java (JVM), que interpreta el código Java en tiempo de ejecución.

Asimismo, Java es un lenguaje compilado, lo que significa que el código fuente es convertido a código máquina antes de ser ejecutado. Es utilizado por desarrolladores para crear aplicaciones web, aplicaciones móviles, juegos, software empresarial, etc.

1.2. Plataforma Java

La plataforma Java es un conjunto de componentes que permiten el desarrollo y ejecución de aplicaciones Java. Los componentes principales de la plataforma Java son:

- La JVM (Java Virtual Machine), que es un software que interpreta el código Java y lo convierte en código máquina para la plataforma específica en la que se está ejecutando.
- El entorno de ejecución Java (JRE, Java Runtime Environment)
- El JDK (Java Development Kit), que es un conjunto de herramientas que permiten desarrollar aplicaciones Java.
- La API (Application Programming Interface), que es un conjunto de clases y métodos que proporcionan funcionalidades para el desarrollo de aplicaciones Java.

1.1. Ediciones y versiones de Java

Java se divide en diferentes ediciones (Editions) y versiones (Versions), las más conocidas son:

- Java Standard Edition (Java SE): Destinada a aplicaciones de escritorio y desarrollo en general.
- Java Enterprise Edition (Java EE): Orientada a aplicaciones empresariales y web.
- Java Micro Edition (Java ME): Diseñada para aplicaciones en dispositivos móviles y embebidos.

1.2. Plataforma Java

La plataforma Java es el entorno para desarrollar y gestionar applets y aplicaciones Java. Consta de tres componentes principales: el lenguaje Java, los paquetes Java y la máquina virtual Java.

El lenguaje Java y los paquetes son similares a C++ y sus bibliotecas de clases. Los paquetes Java contienen clases, que están disponibles en cualquier implementación Java compatible. La interfaz de programación de aplicaciones (API) debe ser la misma en cualquier sistema que soporte Java.

Java difiere de un lenguaje tradicional, como C++, en la forma en que compila y ejecuta. En un entorno de programación tradicional, usted escribe y compila el código fuente de un programa en código objeto para un sistema operativo y un hardware específicos. El código objeto se enlaza con otros módulos de código objeto para crear un programa en ejecución. El código es específico con respecto a un conjunto determinado de hardware de sistema y no funciona en otros sistemas si no se realizan cambios. Esta figura muestra el entorno de despliegue de un lenguaje tradicional.

- Aplicaciones y applets Java

Un applet es un programa Java diseñado para incluirse en un documento web HTML. Puede escribir su applet Java e incluirlo en una página HTML, de la misma forma que se incluye una imagen. Cuando utiliza un navegador habilitado para Java para ver una página HTML que contiene un applet, el código del applet se transfiere al sistema y lo ejecuta la máquina virtual Java del navegador.

- Máquina virtual Java

La máquina virtual Java es un entorno de ejecución que puede añadir a un navegador web o a cualquier sistema operativo, como por ejemplo IBM i. La máquina virtual Java ejecuta las instrucciones que genera un compilador Java. Consta de un intérprete de código de bytes y un tiempo de ejecución que permiten que los archivos de clase Java se ejecuten en cualquier plataforma, independientemente de la plataforma en la que se desarrollaron originalmente.

- Archivos JAR y de clase Java

Un archivo JAR (Java ARchive) es un formato de archivo que combina muchos archivos en uno. El entorno Java difiere de otros entornos de programación en que el compilador Java no genera código de máquina para un conjunto de instrucciones específico de hardware. En su lugar, el compilador Java convierte el código fuente Java en instrucciones de máquina virtual Java, que almacenan los archivos de clase Java. Puede utilizar archivos JAR para almacenar los archivos de clase. El archivo de clase no tiene como destino una plataforma de hardware específica, sino que tiene como destino la arquitectura de máquina virtual Java.

- Hebras Java

Una hebra es una única secuencia independiente que se ejecuta dentro de un programa. Java es un lenguaje de programación multihebra, por lo que puede haber más de una hebra en ejecución en la máquina virtual Java a la vez. Las hebras Java proporcionan una forma para que un programa Java realice varias tareas al mismo tiempo. Una hebra es esencialmente un flujo de control de un programa.

- Java Development Kit

Java Development Kit (JDK) es software para desarrolladores Java. Incluye el intérprete Java, las clases Java y las herramientas de desarrollo Java: compilador, depurador, desensamblador, appletviewer, generador de archivos de resguardo y generador de documentación.

1.3. Historia de Java

Java fue creado por James Gosling y su equipo en Sun Microsystems a principios de la década de 1990. La primera versión pública de Java, conocida como "Java 1.0", se lanzó en 1996. El objetivo de Java era crear un lenguaje de programación que fuera portátil, seguro y robusto. Desde entonces, Java ha evolucionado con nuevas versiones y características. Fue adquirido por Oracle Corporation en 2010.

Los Inicios de Java

El lenguaje Java fue desarrollado en sus inicios por James Gosling, en el año 1991. Inicialmente Java era conocido como Oak o Green.

La primera versión del lenguaje Java es publicada por Sun Microsystems en 1995. Y es en la versión del lenguaje JDK 1.0.2, cuando pasa a llamarse Java, corría el año 1996.

En las primeras versiones de Java 1.1, 1.2 y 1.3 es en la que el lenguaje va tomando forma, con la inclusión de tecnologías como JavaBeans, JDBC para el acceso a base de datos, RMI para las invocaciones en remoto, Collections para la gestión de múltiples estructuras de datos o AWT para el desarrollo gráfico, entre otros.

Java Community Process (JCP)

La versión Java 1.4 pasa a ser la primera versión gestionada por la comunidad mediante el Java Community Process (JCP).

Se trabaja con Java Specification Requests (JSRs) que son las nuevas funcionalidades que se busca que tenga el lenguaje.

Java 1.4 se liberaba como JSR 59, corría el año 2002. ALgunas de las características que contenía eran: librería NIO para IO no bloqueante, JAXP para el procesado de XML y XSLT o el API para preferencias.

Java 5

En 2004 se estaba trabajando con la versión Java 1.5, pero con vistas a reflejar el nivel de madurez de la plataforma Java se renombra a Java 5.

A partir de este momento se identifica el JDK con la versión 1.x, mientras que la plataforma Java sigue con la nueva política de versionado.

Así JDK 1.5 corresponde con Java 5 , JDK 1.6 corresponde con Java 6 ,… y así sucesivamente.

Dentro de Java 5 podemos encontrar el uso de genéricos, el autoboxing/unboxing entre tipos de datos primitivos y sus clases, el uso de enumerados y la aparición del bucle for-each.

Java 6

En el año 2006 aparece la versión Java 6 en la que podíamos encontrar cosas como el soporte de lenguajes de script, facilidades para la exposición y consumo de webservices mediante JAX-WS, nuevos tipos de drivers con JDBC 4 y la versión 2 de JAXB.

Java como Open Source

Una de las cosas que sucede en noviembre 2006 es que Sun Microsystems lo convierte en Open Source mediante una licencia GNU General Public License (GPL).

Dando lugar en mayo 2008 a lo que se conoce como OpenJDK, con OpenJDK 6.

Java 7

Llegado julio de 2011 ve la luz Java 7, la cual trae como novedades el soporte de lenguajes dinámicos, dotando a la JVM de un soporte de mútiples lenguajes y una nueva librería I/O para el manejo de ficheros.

También aparecen cosas menores, pero muy útiles como el manejo de String dentro de la validación en una estructura switch o la capacidad de poner subrayados en los números para que se puedan leer mejor.

Java 8

Versión aparecida en marzo de 2014.

Entre las características de Java 8 tenemos el soporte expresiones Lambda y uso de Streams, que permiten un estilo más funcional para los programas Java. Dentro de este enfoque más funcional también aparecen las transformaciones MapReduce.

Ve la luz el Proyecto Nashorn para disponer de un engine Javascript y así poder incluir este lenguaje dentro de las aplicaciones Java.

Otras cosas son un nuevo API Date y Time y la inclusión de JavaFX 8 dentro de la JDK de Java.

1.4. JDK

(Java Development Kit) es un conjunto de herramientas que los desarrolladores pueden utilizar para crear aplicaciones Java. El JDK incluye las siguientes herramientas:

- javac, que es un compilador que convierte el código fuente Java en bytecode.
- java, que es un intérprete que ejecuta el bytecode Java.
- jar, que es una herramienta que empaqueta los archivos Java en un archivo JAR.
- javadoc, que es una herramienta que genera documentación para las clases Java.

1.5. JVM

Es responsable de ejecutar el código compilado en bytecode y proporcionar una capa intermedia entre el software y el hardware.

La JVM (Java Virtual Machine) es un software que interpreta el código Java y lo convierte en código máquina para la plataforma específica en la que se está ejecutando. La JVM es responsable de la ejecución de las aplicaciones Java.

La JVM tiene las siguientes características:

- Portabilidad: La JVM permite que el mismo código Java se ejecute en diferentes plataformas.

- Seguridad: La JVM proporciona un entorno de ejecución aislado en el que se ejecuta el código Java.

Existen varias implementaciones de la JVM, incluyendo la JVM de Oracle, OpenJDK y otras desarrolladas por terceros.

1.6. Documentación API

Se define tres plataformas en un intento por cubrir distintos entornos de aplicación. Así, ha distribuido muchas de sus API (Application Program Interface) de forma que pertenezcan a cada una de las plataformas:

- Java ME (Java Platform, Micro Edition) o J2ME — orientada a entornos de limitados recursos, como teléfonos móviles, PDAs (Personal Digital Assistant), etc.
- Java SE (Java Platform, Standard Edition) o J2SE — para entornos de gama media y estaciones de trabajo. Aquí se sitúa al usuario medio en un PC de escritorio.
- Java EE (Java Platform, Enterprise Edition) o J2EE — orientada a entornos distribuidos empresariales o de Internet.

Las clases en las API de Java se organizan en grupos disjuntos llamados paquetes. Cada paquete contiene un conjunto de interfaces, clases y excepciones relacionadas. La información sobre los paquetes que ofrece cada plataforma puede encontrarse en la documentación de ésta.

El conjunto de las API es controlado por Sun Microsystems junto con otras entidades o personas a través del programa JCP (Java Community Process). Las compañías o individuos participantes del JCP pueden influir de forma activa en el diseño y desarrollo de las API, algo que ha sido motivo de controversia.

2. GIT & GITHUB

2.1. Introducción a Git

Git es un sistema de control de versiones distribuido, lo que significa que un clon local del proyecto es un repositorio de control de versiones completo. Estos repositorios locales plenamente funcionales permiten trabajar sin conexión o de forma remota con facilidad. Los desarrolladores confirman su trabajo localmente y, a continuación, sincronizan la copia del repositorio con la del servidor.

La flexibilidad y popularidad de Git lo convierten en una excelente opción para cualquier equipo. Muchos desarrolladores y graduados universitarios ya saben cómo usar Git. La comunidad de usuarios de Git ha creado recursos para entrenar a los desarrolladores y la popularidad de Git facilita recibir ayuda cuando se necesita. Casi todos los entornos de desarrollo tienen compatibilidad con Git y las herramientas de línea de comandos de Git implementadas en todos los sistemas operativos principales.

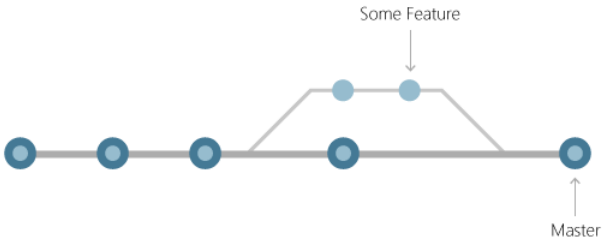
Aspectos básicos de Git

Cada vez que se guarda el trabajo, Git crea una confirmación. Una confirmación es una instantánea de

todos los archivos en un momento dado. Si un archivo no ha cambiado de una confirmación a la siguiente, Git usa el archivo almacenado anteriormente. Este diseño difiere de otros sistemas que almacenan una versión inicial de un archivo y mantienen un registro de las diferencias a lo largo del tiempo.

Las confirmaciones crean vínculos a otras confirmaciones, formando un gráfico del historial de desarrollo. Es posible revertir el código a una confirmación anterior, inspeccionar cómo cambian los archivos de una confirmación a la siguiente y revisar información como dónde y cuándo se realizaron los cambios. Las confirmaciones se identifican en Git mediante un hash criptográfico único del contenido de la confirmación. Dado que todo tiene hash, es imposible realizar cambios, perder la información o dañar los archivos sin que Git lo detecte.

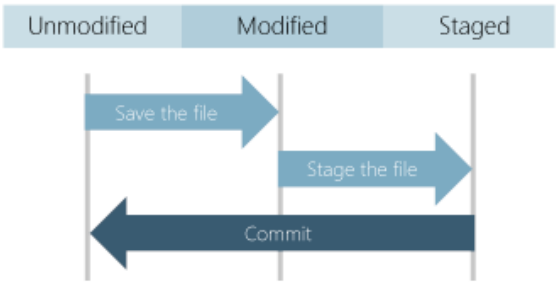
Ramas



Cada desarrollador guarda los cambios en su propio repositorio de código local. Como resultado, puede haber muchos cambios diferentes basados en la misma confirmación. Git proporciona herramientas para aislar los cambios y volver a combinarlos posteriormente. Las ramas, que son punteros ligeros para el trabajo en curso, administran esta separación. Una vez finalizado el trabajo creado en una rama, se puede combinar de nuevo en la rama principal (o troncal) del equipo.

Archivos y confirmaciones

Los archivos de Git se encuentran en uno de estos tres estados: modificados, almacenados provisionalmente o confirmados. Cuando se modifica un archivo por primera vez, los cambios solo existen en el directorio de trabajo. Todavía no forman parte de una confirmación ni del historial de desarrollo. El desarrollador debe almacenar provisionalmente los archivos modificados que se incluirán en la confirmación. El área de almacenamiento provisional contiene todos los cambios que se incluirán en la siguiente confirmación. Una vez que el desarrollador esté satisfecho con los archivos almacenados provisionalmente, los archivos se empaquetan como una confirmación con un mensaje que describe lo que ha cambiado. Esta confirmación pasa a formar parte del historial de desarrollo.



El almacenamiento provisional permite a los desarrolladores elegir qué cambios de archivo se guardarán en una confirmación para desglosar los cambios grandes en una serie de confirmaciones más pequeñas. Al reducir el ámbito de las confirmaciones, es más fácil revisar el historial de confirmaciones para buscar cambios de archivo específicos.

Ventajas de Git

Desarrollo simultáneo

Todos los usuarios tienen su propia copia local de código y pueden trabajar simultáneamente en sus propias ramas. Git funciona sin conexión, ya que casi todas las operaciones son locales.

Versiones de lanzamiento más rápidas

Las ramas permiten un desarrollo flexible y simultáneo. La rama principal contiene código estable y de alta calidad desde el que publica. Las ramas de características contienen trabajo en curso y se combinan con la rama principal tras la finalización. Al separar la rama de versión del desarrollo en curso, es más fácil administrar código estable y enviar actualizaciones más rápidamente.

Integración incorporada

Debido a su popularidad, Git se integra en la mayoría de las herramientas y productos. Todos los IDE principales tienen compatibilidad integrada con Git y muchas herramientas admiten la integración y la implementación continuas, las pruebas automatizadas, el seguimiento de los elementos de trabajo, las métricas y la integración de características de informes con Git. Esta integración simplifica el flujo de trabajo diario.

Sólido soporte técnico de la comunidad

Git es de código abierto y se ha convertido en el estándar de facto para el control de versiones. No hay escasez de herramientas y recursos disponibles para que los equipos aprovechen. El volumen de soporte técnico de la comunidad para Git en comparación con otros sistemas de control de versiones facilita recibir ayuda cuando se necesita

Git funciona con cualquier equipo

Utilizar Git con una herramienta de administración de código fuente aumenta la productividad de un equipo al fomentar la colaboración, aplicar directivas, automatizar procesos y mejorar la visibilidad y la rastreabilidad del trabajo. El equipo puede decidirse por herramientas individuales para el control de versiones, el seguimiento de los elementos de trabajo y la integración e implementación continuas. O bien, pueden elegir una solución como GitHub o Azure DevOps que admita todas estas tareas en un solo lugar.

2.2. Comandos básicos de Git

clear—Limpia pantalla de la terminal git bash.
git status—Muestra estado de la rama actual.
git log --oneline–Muestra commits en formateado a una línea por commit.
git add . or specific files—Añade todo o archivos especificos al staging area.
git commit -m "message"—Crear commit con mensaje.
git branch—Muestra todas las ramas.
git branch branch_name—Crea una nueva rama.
git checkout branch_name—Cambiar a otra rama.
git checkout -b branch_name—Crea y cambia a una rama nueva.
git switch branch_name —Cambiar a otra rama.
git commit -am "message"—añade todos los archivos y crea un nuevo commit con mensaje.
git merge branch_name—Une una rama con la rama en la que te encuentras.
git switch -c branch_name— Crea y cambia a una rama nueva.
git ls-files–Ver archivos que están en el staging area.
git rm file_name.ext– Eliminar archivo.
git checkout file_to_undo_changes.ext–regresa a la última modificación de un archivo.
git restore file_name.ext or .–Restaura archivo o archivos especificos.
git stash –Guardar cambios sin haber hecho commit
git stash list–muestra stashes.
git stash push -m "message"–Poner mensaje a Stash.
git stash drop n_stash–Elmina n_stash de lista de stashes.
git stash clear–Elimina todos los stashes.

2.3. Repositorios en GitHub (Ramas, Uniones de ramas, Conflictos)

git remote add origin URL .—Conecta repositorio remoto a uno local.
git push origin branch_name –Actualiza repositorio remoto, desde una rama local.
git remote—Muestra las direcciones remotas conectadas al repositorio local.
git branch -a—Muestra todas las ramas incluyendo las remotas
git branch -r—Muestra todas las ramas remotas
git remote show origin—Muestra detalles de un origen remoto.
git fetch origin—Trae los cambios remotos al staging area del repositorio local.
git pull origin branch_name—Actualiza rama local, git fetch y git merge.
git clone URL—clonar repositorio remoto en un repositorio local.

3. IMPLEMENTACIÓN DE SCRUM: CÓMO EMPEZAR

1. **Elige un responsable del producto (Product Owner, PO).**

Este individuo es quien posee la visión de lo que vas a hacer, producir o lograr. Toma en cuenta los riesgos y recompensas, qué es posible, qué puede hacerse y qué es lo que le apasiona.

2. **Selecciona un equipo.**

¿Quiénes serán las personas que harán efectivamente el trabajo? Este equipo debe contar con todas las habilidades necesarias para tomar la visión de los responsables del producto y hacerla realidad. Los equipos deben ser pequeños, de tres a nueve personas por regla general.

3. **Elige un Scrum Master.**

Ésta es la persona que capacitará al resto del equipo en el enfoque Scrum y que ayudará al equipo a eliminar todo lo que lo atrasa.

4. **Crea y prioriza una bitácora del producto.**

Se trata de una lista de alto nivel de todo lo que debe hacerse para volver realidad la visión. Esta bitácora existe y evoluciona durante el periodo de vida del producto; es la guía de caminos hacia éste. En un momento dado, la bitácora del producto es la visión definitiva de "todo lo que el equipo podría hacer, en orden de prioridad". Hay sólo una bitácora del producto; esto significa que el responsable del producto debe tomar decisiones de priorización en todo el espectro. El responsable del producto debe consultar tanto a todos los interesados como al equipo para cerciorarse de que representa lo que la gente necesita y lo que se puede hacer.

5. **Afina y estima la bitácora del producto.**

Es crucial que la gente que realmente se hará cargo de los elementos de la bitácora del producto estime cuánto esfuerzo implicarán. El equipo debe examinar cada elemento de la bitácora y ver si, en efecto, es viable. ¿Hay información suficiente para llevar a cabo el elemento? ¿Éste es lo bastante pequeño para estimarse? ¿Existe una definición de "terminado"; es decir, todos están de acuerdo en los criterios que deben cumplirse para poder decir que algo está "terminado"? ¿Esto crea valor visible? Cada elemento debe poder mostrarse, demostrarse y (es de esperar) entregarse. No calcules la bitácora en horas, porque la gente es pésima para esto. Calcula por tamaño relativo: pequeño, mediano o grande. O, mejor todavía, usa la serie de Fibonacci y estima el valor puntual de cada elemento: 1, 2, 3, 5, 8, 13, 21, etcétera.

6. **Planeación del sprint.**

Ésta es la primera de las reuniones de Scrum. El equipo, el Scrum Master y el responsable del producto se sientan a planear el sprint. Los sprints son siempre de extensión fija, inferior a un mes. La mayoría de la gente ejecuta en la actualidad uno o dos sprints semanales. El equipo examina el inicio de la bitácora y pronostica cuánto puede llevar a cabo en este

sprint. Si el equipo ha pasado por varios sprints, debe considerar el número de puntos que acumuló en el más reciente. Este número se conoce como velocidad del equipo. El Scrum Master y el equipo deben tratar de aumentar ese número en cada sprint. Ésta es otra oportunidad para que el equipo y el responsable del producto confirmen que todos comprenden a la perfección cómo esos elementos cumplirán la visión. Durante esta reunión todos deben acordar asimismo una meta de sprint, que todos han de cumplir en este sprint.

Uno de los pilares de Scrum es que una vez que el equipo se compromete con lo que cree que puede terminar en un sprint, eso se queda ahí. No puede cambiar ni crecer. El equipo debe ser capaz de trabajar en forma autónoma a lo largo del sprint para terminar lo que pronosticó que podía hacer.

7. **Vuelve visible el trabajo.**

La forma más común de hacerlo en Scrum es crear una tabla de Scrum con tres columnas: Pendiente, En proceso y Terminado. Notas adhesivas representan los elementos por llevar a cabo y el equipo avanza por la tabla conforme los va

concluyendo, uno por uno. Otra manera de volver visible el trabajo es crear un diagrama de finalización. En un eje aparece el número de puntos que el equipo introdujo en el sprint y en el otro el número de días. Cada día, el Scrum Master suma el número de puntos completados y los grafica en el diagrama de finalización. Idealmente, habrá una pendiente descendente que conduzca a cero puntos para el último día del sprint.

8. **Parada diaria o Scrum diario.**

Éste es el pulso de Scrum. Cada día, a la misma hora, durante no más de quince minutos, el equipo y el Scrum Master se reúnen y contestan tres preguntas:

- ¿Qué hiciste ayer para ayudar al equipo a terminar el sprint?
- ¿Qué harás hoy para ayudar al equipo a terminar el sprint?
- ¿Algún obstáculo te impide o impide al equipo cumplir la meta del sprint?

Eso es todo, en eso consiste la reunión. Si se prolonga más de quince minutos lo estás haciendo mal. Lo que esto hace es ayudar al equipo a saber exactamente dónde se encuentra todo en el curso de un sprint. ¿Todas las tareas serán terminadas a tiempo? ¿Hay oportunidades de ayudar a otros miembros del equipo a vencer obstáculos? Las tareas no se asignan desde arriba; el equipo es autónomo: él lo hace. Tampoco se rinden informes detallados a la dirección. El Scrum Master se encarga de eliminar los obstáculos, o impedimentos, contra el progreso del equipo.

9. **Revisión del sprint o demostración del sprint.**

Ésta es la reunión en la que el equipo muestra lo que hizo durante el sprint. Todos pueden asistir, no sólo el responsable del producto, el Scrum Master y el equipo, sino también los demás interesados, la dirección, clientes, quien sea. Ésta es una reunión abierta en la que el equipo hace una demostración de lo que pudo llevar a Terminado durante el sprint.

El equipo debe mostrar únicamente lo que satisface la definición de Terminado, lo total y completamente concluido y que puede entregarse sin trabajo adicional. Esto puede no ser un producto terminado, pero sí una función concluida de uno de ellos.

10. Retrospectiva del sprint.

Una vez que el equipo ha mostrado lo que logró en el sprint más reciente –la cosa "terminada" y en posibilidad de enviarse a los clientes en busca de realimentación–, piensa en qué marchó bien, qué pudo haber marchado mejor y qué puede mejorar en el siguiente sprint. ¿Cuál es la mejora en el proceso que como equipo pueden implementar de inmediato?

Para ser eficaz, esta reunión requiere cierto grado de madurez emocional y una atmósfera de confianza. La clave es que no se trata de buscar a quién culpar; lo que se juzga es el proceso. ¿Por qué tal cosa ocurrió de tal manera? ¿Por qué pasamos por alto tal otra? ¿Qué podríamos hacer más rápido? Es crucial que la gente, como equipo, asuma la responsabilidad de su proceso y de sus resultados y busque soluciones también como equipo. Al mismo tiempo, debe tener fortaleza para tocar los temas que le incomodan de un modo orientado a la solución, no acusatorio. Y el resto del equipo ha de tener la madurez de oír la realimentación, aceptarla y buscar una solución, no ponerse a la defensiva.

Al final de la reunión, el equipo y el Scrum Master deben acordar una mejora al proceso que implementarán en el siguiente sprint. Esa mejora al proceso, también llamada kaizen, debe incorporarse en la bitácora del sprint siguiente, con pruebas de aceptación. De esta manera, el equipo podrá ver fácilmente si en verdad implementó la mejora y qué efecto tuvo ésta en la velocidad.

11. Comenzar el siguiente sprint.

Comienza de inmediato el ciclo del siguiente *sprint* tomando en cuenta la experiencia del equipo con los impedimentos y mejoras del proceso.