Curso de Ingeniería de Software

Unidad 7
Construcción de software

Guadalupe Ibargüengoitia
Hanna Oktaba

Aprendizaje analítico y crítico

- **¿Porqué ?** – cuál es el problema, cuál es su causa, su contexto
- **¿Qué?** – solución al problema
- **¿Para qué?** – en qué medida se resuelve el problema

Objetivos

- Construir componentes individuales de software, basándose en el diseño y probarlos para eliminar los defectos.

- Realizar las revisiones entre colegas del código para incrementar la eliminación de defectos y fomentar la comprensión del código entre equipo.

Entradas de esta unidad

- **Condiciones**
  - Diseño del software completo
- **Productos de trabajo**
  - *Especificación de requerimientos de software*
  - *Diseño General de Arquitectura de Microservicios y de la BD de la Iteración* y del *Diseño detallado del Microservicio* (por equipo)

DEFINICIONES DE CONCEPTOS

Construcción de software

- Es la actividad central en el desarrollo de software.
- Es un proceso creativo e iterativo, que parte del diseño detallado de los componentes, los cuales se refinan introduciendo elementos del lenguaje de programación.
- Este refinamiento se repite hasta que se pueda derivar el código.
- La creación de código puede significar la construcción de programas nuevos en lenguajes de programación de alto o bajo nivel o la adaptación de código ya existente.

Objetivos de la construcción

- La generación del código de cada componente definido en el diseño.

- Probar cada componente individualmente para asegurar que el código es confiable antes de integrarlo con los demás componentes.

Fundamentos para la construcción de software (SWEBOK, 2014) (1/6)

- **Minimizar la complejidad**. Construir código que sea simple y lo puedan leer con facilidad las personas y las máquinas.
- Para lograrlo se hace uso de estándares, diseño modular y se aplican técnicas de construcción que fomenten la calidad.

Fundamentos para la construcción de software (SWEBOK, 2014) (2/6)

- **Anticiparse al cambio**. El software es muy propenso a tener que ser modificado frecuentemente debido a los cambios en el medio ambiente de su uso.
- **Anticiparse al cambio.** Es construir software que pueda extenderse y mejorarse sin causar efectos graves en la estructura general.

Fundamentos para la construcción de software (SWEBOK, 2014) (3/6)

- **Construir para la verificación**. Significa que el código escrito pueda ser revisado fácilmente por los desarrolladores para que detecten los defectos y puedan corregirlos.

- Algunas técnicas útiles para facilitar la verificación son:
  - El uso de estándares de codificación
  - Hacer revisiones al código antes de compilarlo
  - Organizarlo para facilitar la prueba automática de código
  - No incluir estructuras complejas o difíciles de entender.

Fundamentos para la construcción de software (SWEBOK, 2014) (4/6)

- **Reuso**. Antes de generar código nuevo, es muy recomendable revisar si se tiene código o componentes ya existentes y confiables.

- Componentes confiables se encuentran en las bibliotecas de los ambientes de programación y en código realizado por el equipo y que haya sido certificado por el mismo equipo.

Fundamentos para la construcción de software (SWEBOK, 2014) (5/6)

- **Uso de estándares en la construcción**. Aplicar estándares de proceso en la construcción ayuda a obtener los objetivos de eficiencia, calidad y costo del proyecto.
- El uso de estándares para la codificación ayuda a la comprensión del código, facilita su corrección y ayuda a disminuir la deuda técnica.

Fundamentos para la construcción de software (SWEBOK, 2014) (6/6)

- **Manejo de versiones.** Durante la construcción de software se generan varias versiones de cada componente. Se debe llevar un buen control del manejo de estos cambios para no incluir versiones erróneas en los componentes finales.
- Cuando un equipo está construyendo software en conjunto, se debe asegurar que no interfiera un desarrollador con otro, esto es que sus cambios estén coordinados.
- Usar herramientas de control de versiones.

¿Qué hemos aprendido?

- ¿Porqué hay que minimizar la complejidad del código?
- ¿Porqué hay que hacer el código fácil de cambiar?
- ¿Qué significa la verificación del código y porqué se hace?
- ¿Porqué hay que usar estándares en la construcción del código?
- ¿Porqué hay que controlar las versiones del código?

Actividades para la construcción de software

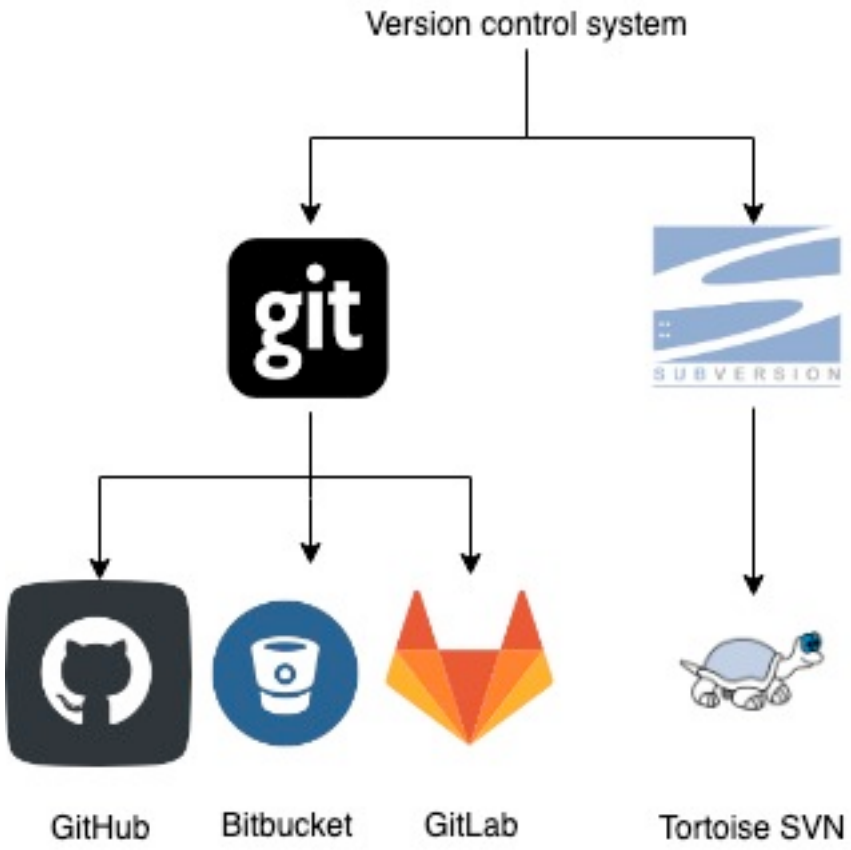
- Preparar la construcción del software
- Construir el código de cada componente
- Probar cada componente individualmente

Preparar la construcción del software

- Hay que asegurar que todos los miembros del equipo están capacitados y tienen instaladas las herramientas para el trabajo individual y el colaborativo, definidas en el diseño.
- El equipo conoce los estándares de construcción que debe de seguir.
- El equipo acuerda y define los elementos comunes (estilo de la interfaz de usuario, la base de datos, herramientas) para que todos trabajen con lo mismo.

Control de versiones del código

- Contexto- durante la construcción y mantenimiento del software los cambios en el código son inevitables.
- Problema – es muy fácil perder el control sobre los cambios, sobre todo en el trabajo en equipo.
- Solución – herramientas que ayudan a identificar los cambios y manejar las versiones de manera automatizada.
- Beneficios – identificar cada versión, poder regresar a las versiones anteriores en caso de que nueva versión presente problemas.



CVS Server

Concurrent Versions System
1986

Construir el código de cada componente

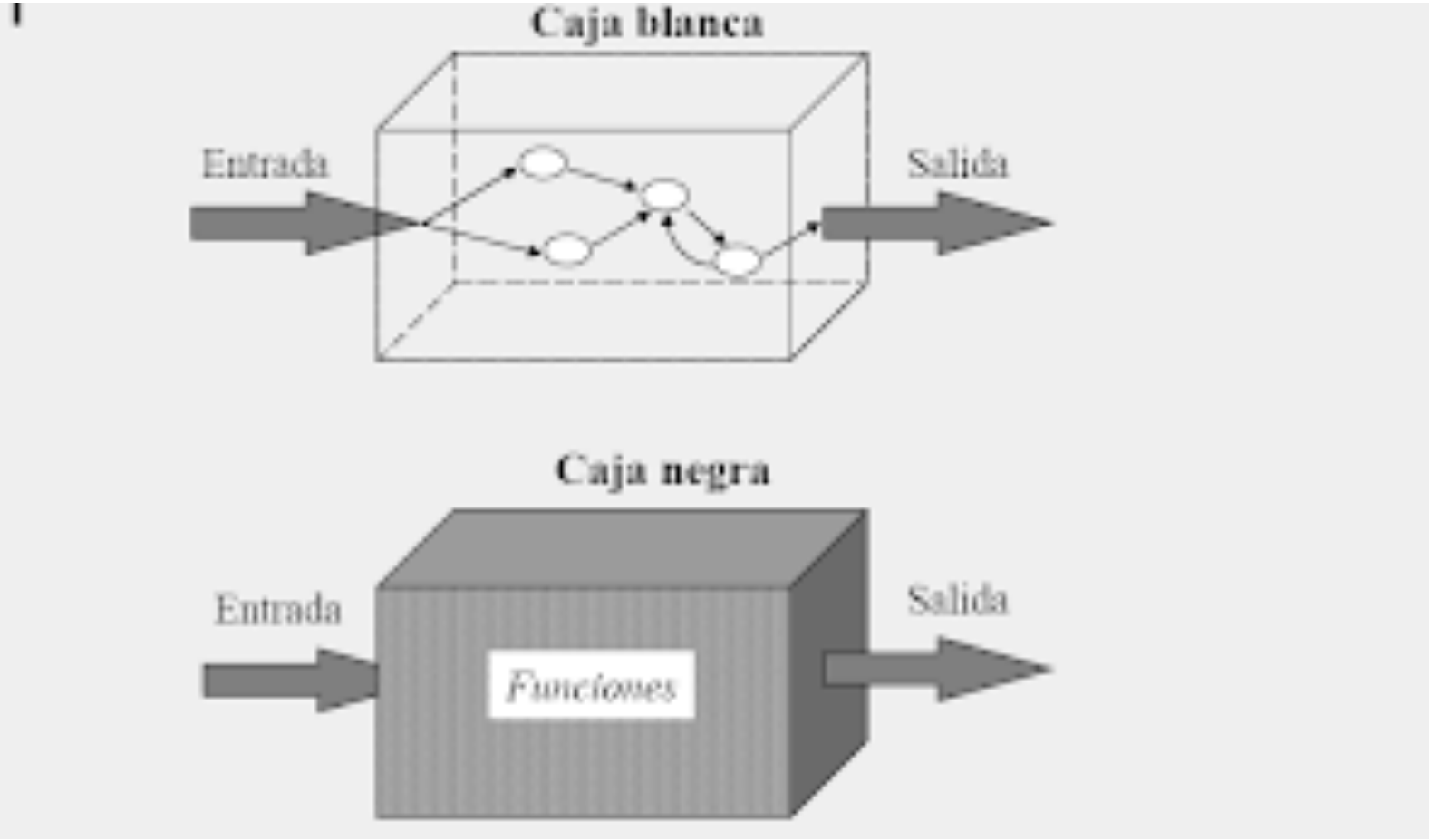
- Cada desarrollador crea una rama para su código en la herramienta de desarrollo colaborativo.
- Revisa el diseño de los componentes, que ha elegido o le fueron asignados, y construye el código en su rama local, siguiendo los estándares acordados.
- Revisa el código, lo compila y corrige los defectos hasta que pase la compilación con éxito.

Probar cada componente individualmente

- Las pruebas que se aplican a cada *componente* de forma individual se llaman *pruebas unitarias (unit test)*.
- El objetivo de pruebas unitarias es asegurar que el componente probado cumpla con el diseño y tenga el comportamiento esperado.
- Probar un componente para todos los datos posibles no es factible.
- Por lo tanto se define un criterio de salida que guía a los desarrolladores en la decisión de dejar de hacer pruebas o cuando las pruebas son suficientes para un determinado tipo de componente.

Tipos de prueba

- **Pruebas de caja negra.** Se proporcionan los datos de entrada y se observa si los resultados son esperados según la funcionalidad, sin considerar la estructura del código de un componente.
- **Pruebas de caja blanca/transparente.** Se considera la estructura del código para preparar los datos de entrada que permiten probar la ejecución de todas las instrucciones del código.


Caja blanca
Entrada
Salida
Caja negra
Entrada
Salida
Funciones

Pruebas unitarias

- **Se enfocan más en pruebas de caja blanca/ transparente** porqué su objetivo principal es validar la ejecución de todas sus instrucciones.
- Por lo tanto, cuando el código del componente tiene una o mas sentencias de decisión (tipo *if-then-else* o ciclos), se definen tantos casos de prueba, con diferentes datos de entrada, que causen la ejecución de cada uno de los flujos del código.

Complejidad Ciclomática (McCabe)

- Para saber cuántos casos de prueba se necesitan para probar todos los flujos de un código, existe una medida llamada ***Complejidad Ciclomática*** (**McCabe**).
- Esta medida se define como el número de sentencias de decisión en el código más uno.
- Este es el número de casos de prueba a definir, con datos de entrada diferentes, para probar todos los flujos de un código.

Sustitutos de componentes para las pruebas

- En muchas ocasiones, al querer probar un componente, se requiere de otros componentes que no están disponibles.
  - *Stub/mock/fake* – sustituto que simula la invocación del componente no existente y devuelve el resultado esperado.
  - *Driver* - sustituto que simula la inicialización de variables no locales y los parámetros para activar al componente que se está probando.

Sustitutos de componentes para las pruebas en lenguajes OO (Wikipedia)

- **Objetos simulados** (pseudoobjetos, *mock object*, objetos de pega) a los objetos que imitan el comportamiento de objetos reales de una forma controlada.
- En las pruebas unitarias, los objetos simulados se usan para simular el comportamiento de objetos complejos cuando es imposible o impráctico usar al objeto real en la prueba.
- Los objetos simulados son muy simples de construir y devuelven un resultado determinado sin implementar el algoritmo que lo produzca.

Características de prueba unitaria

- **Completa** - Debe cubrir la totalidad del código de negocio y debe cubrir la mayor cantidad de escenarios posibles.
- **Automática** - No requiere de intervención humana para su ejecución.
- **Autónoma** - No requiere de recursos adicionales, como una BD, por ejemplo, para su ejecución.
- **Independiente** - La ejecución de esta prueba no debe afectar a la ejecución de otra prueba.
- **Aislada** - La ejecución de otra prueba no debe afectar a la ejecución de esta prueba.
- **Repetible** - El procedimiento a realizar para su primer ejecución debe ser el mismo que hay que efectuar para cualquier ejecución futura y debe poder ser llevado a cabo en cualquier momento.
- **Consistente** - Cada vez que se ejecute, el resultado debe ser el mismo.

Aplicación de pruebas unitarias

- Cada vez que se incorpora una nueva prueba unitaria, su ejecución se debe llevar a cabo ejecutando también todas las pruebas unitarias que se han creado antes, y esta actividad será exitosa SÓLO si ninguna prueba falló.

Corrección de defectos en pruebas unitarias

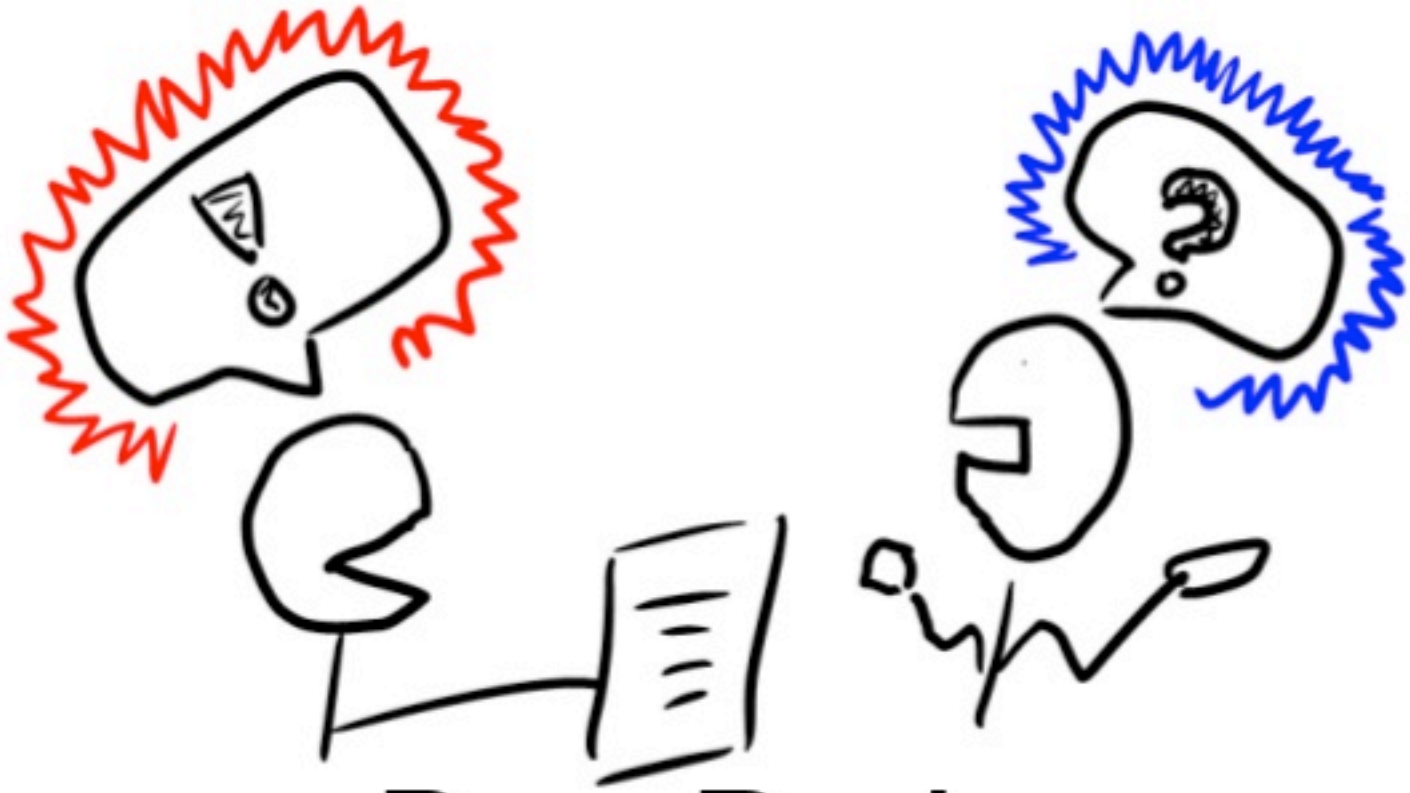
- Cada defecto identificado durante las pruebas unitarias debe ser corregido.
- Se busca la causa del defecto en el código, se modifica el código para eliminarlo, se vuelve a compilar y a probar con los mismos casos de pruebas unitarias.
- Es importante tener resguardado el código de casos de prueba para automatizar parte de este proceso.

Defectos fugados/escapados

- Un defecto no encontrado en la fase de pruebas (o validaciones), que aparece en una fase posterior se conoce como defecto fugado o escapado.
- Se estima que el costo de corrección de un defecto fugado en la siguiente fase es 10 veces mayor que en la fase previa. Por eso es muy importante detectar los defectos de manera más temprana posible.

Revisión por pares (Peer Review) o Revisión entre colegas

- Es la evaluación del producto de trabajo realizada por una o más personas con competencias similares (pares) a las de los productores del producto de trabajo.
- En desarrollo de software la revisión por pares es una verificación de la calidad de los resultados de trabajo y puede aplicarse a la especificación de requerimientos, del diseño y al **código**.
- Es una forma complementaria de encontrar defectos con "cabezas humanas".



Peer Reviews

(and how to survive them)

Alexander Serebrenik
Eindhoven University of Technology, The Netherlands

¿Qué hemos aprendido?

- ¿Qué es importante en la preparación de la construcción del software?
- ¿Qué es lo que hay que cuidar al construir el código de cada componente?
- ¿Qué son y para qué sirven las pruebas unitarias?
- ¿Qué son los defectos fugados y porqué hay que minimizarlos?

PRACTICA PD3 CONSTRUCCIÓN DE SOFTWARE

PD3 Construcción de software

Objetivos

Construir y probar el código para cada componente del diseño de software.

PD3 Construcción de software

Entradas

Condiciones

- Diseño de software completo

Productos de trabajo

- *Especificación de requerimientos de software*
- *Diseño General de Arquitectura de Microservicios y de la BD de la Iteración* y del *Diseño detallado del Microservicio* (por equipo)
- Herramienta(s) colaborativa(s)

PD3 Construcción de software

Actividades

1. Preparar la construcción del software.
2. Construir los componentes individualmente.
3. Realizar pruebas unitarias de cada componente y corregir defectos.
4. Realizar revisión del código entre colegas y corregir defectos.

TÉCNICAS PARA REALIZAR LAS ACTIVIDADES DE PD3

Actividad 1: Preparar la construcción del software

1. Instalar las herramientas del desarrollo

Responsable técnico se asegura que todos los miembros del equipo están capacitados en las herramientas que se van a usar, que tengan instaladas las mismas versiones en su equipo y que tengan acceso a las herramientas colaborativas.

Actividad 1: Preparar la construcción del software

2. Definir el plan de la construcción

- En una reunión coordinada por el responsable técnico se definen y distribuyen equitativamente las tareas de construcción del código con base en el diseño.
- Por cada tarea se colocan en el tablero las tarjetas en la columna Por Hacer (ToDo) con el nombre del responsable y la fecha de compromiso.

Actividad 1: Preparar la construcción del software

3. Revisar los estándares para la codificación

- **Para la base de datos**
- **Código Java**
- **HTTP 1.1**
- **REST API**

Actividad 1: Preparar la construcción del software

4. Homologar el estilo gráfico de la interfaz de usuario

- Acordar las hojas de estilo para la interfaz gráfica.
- Bajar el conjunto de elementos que definen el estilo gráfico de la interfaz de usuario a su equipo.
- Para que todos puedan construir sus pantallas de manera uniforme.

Actividad 1: Preparar la construcción del software

5. Construir la base de datos compartida

Instalar la base de datos en su equipo, usando el script de restauración que usa el *Esquema de entidades de persistencia* basado en el diseño de la base de datos para el proyecto.

Actividad 2. Construir los componentes individualmente.

1. El desarrollador responsable por una tarea de construcción mueve la tarea desde la columna "ToDo" a la columna de "DOING".
2. El desarrollador crea una sub-rama de la rama de “Develop" con el nombre de su actividad en GitLab.
3. El desarrollador descarga la sub-rama a su equipo local.

Actividad 2. Construir los componentes individualmente.

4. El desarrollador trabaja sobre su equipo local y:
   4.1. Codifica , basándose en el diseño, y realiza Commits locales de unidades de trabajo completas.
   4.2. Valida que su código esté apegado a los estándares de codificación.
   4.3. Incorpora documentación en línea "javadoc" (qué hace clase o método) a su código.
   4.4. Incorpora documentación en línea "double //" (cómo hace- algoritmo) a su código.
   4.5. Incorpora log (info, warning, error y debug) a su código.

Actividad 3. Realizar pruebas unitarias de cada componente y corregir defectos.

4.6. Genera las pruebas unitarias para sus clase de servicio en la sección correspondiente de la sub-rama local.

4.7 Ejecuta las pruebas unitarias de sus clases de servicios, corrige los defectos y vuelve a probar hasta eliminarlos.

5. El desarrollador realiza un "Push" de su sub-rama a GitLab.

Actividad 4. Realizar revisión del código entre colegas y corregir defectos.

6. El desarrollador crea un "Merge Request" y asigna al menos dos revisores de su equipo para que verifiquen su código.

7. Una notificación electrónica informa a los revisores que tienen una solicitud de revisión de código.

8. Cada revisor accede al código a revisar y realiza observaciones al código en GitLab.

Actividad 4. Realizar revisión del código entre colegas y corregir defectos.

9. El autor original del código recibe una notificación electrónica de cada observación y:

9.1. Analiza la observación

9.2. Si la observación es procedente:

9.2.1. Corrige su código y prueba localmente

9.2.2. Genera un Commit de la corrección

9.2.3. Realiza un "Push" de su código a GitLab

9.2.4. El GitLab informa a los revisores de que la corrección fue realizada

9.3. Si la observación no es procedente:

9.3.1. El GitLab informa al revisor el porqué la observación no procede

10. Repite el ciclo del punto 9 hasta que no haya observaciones

Actividad 4. Realizar revisión del código entre colegas y corregir defectos.

11. Al terminar todas las correcciones de las observaciones mueve la tarjeta de la tarea desde la columna ” DOING" a la columna de "DONE”.

PD3 Construcción de software

Resultados

Condiciones

- Construcción y pruebas unitarias de componentes del software terminados

Productos de trabajo

- Código documentado, probado, revisado y resguardado en las sub-ramas de cada desarrollador del repositorio compartido
- *Tablero* de la iteración actualizado

Resultados de esta unidad

- **Condiciones**
- Construcción y pruebas unitarias de componentes del software terminados

- **Productos de trabajo**
- Código documentado, probado, revisado y resguardado en las sub-ramas de cada desarrollador del repositorio compartido
- *Tablero* de la iteración actualizado

Bibliografía

- McCabe T.J. (Dic 1976). A software Complexity Measure. *IEEE Trans. Software Engineering, vol 2, no. 6*, 308-320. http://www.literateprogramming.com/mccabe.pdf
- Sommerville I. (2011). *Software Engineering.* Pearson.
- SWEBOK v 3.0. (2014).
- https://en.wikipedia.org/wiki/Mock_object