**PLAN DE CALIDAD**

**Grupo: B**

**Empresa / proyecto:**

**Curso: 2° 2° A**

**Integrantes:**

**Ramiro F. de la Peña,**

**Santiago Frías,**

**Martín Gentile,**

**Alberto Malek**

**Objetivos**

Es necesario realizar determinadas tareas de manera sistemática, como así también llevar a cabo la planificación desde el comienzo del proyecto para lograr alcanzar los objetivos planteados y así lograr un producto de software de calidad adecuada.

Los objetivos planteados son:

1. Funcionalidad
2. Alcance
3. Documentación
4. Comunicación
5. Tiempo
6. Riegos

**Funcionalidad**

Con respecto a los entregables luego de finalizar cada sprint nos proponemos cumplir con el 80% de las funcionalidades requeridas como mínimo objetivo. Teniendo en cuenta que se busca un código limpio, fácil de leer y mantener, como así también asegurarnos de alcanzar la meta de cero errores en cada entrega.

**Alcance**

Se busca definir los estándares de calidad mínimos requeridos para cumplimentar de manera satisfactoria tanto los entregables como el producto final, contando con el respaldo de herramientas profesionales del desarrollo de software.

**Documentación**

Se considera que a pesar de utilizar una metodología de trabajo ágil, es necesario documentar cada paso. Por eso uno de los objetivos es que siempre exista documentación que respalde cada uno de los sprint, que la misma esté revisada y aprobada por el cliente.

**Comunicación**

En cuanto a la comunicación es primordial asegurarse de cumplir con todos y cada uno de los puntos establecidos en el *Plan de Comunicación*.

**Tiempo**

Es de suma importancia cumplir con los objetivos de tiempo establecidos en el planeamiento del proyecto, por este motivo se fija evitar desvíos según lo establecido en el *Plan de Tiempo.*

**Riesgos**

Se busca lograr cumplir en su totalidad el plan de identificación y mitigación de los riesgos definido en el documento *Plan de Riesgo.*

**Proceso de Software**

Por definición es un conjunto de actividades, métodos, prácticas y transformaciones que las personas usan para desarrollar y mantener software y sus productos asociados. En nuestro caso el tipo de proceso a implementar será ágil, ya que utilizaremos como metodología de trabajo SCRUM y nos centraremos en la comunicación constante entre los integrantes del grupo para lograr desarrollar nuestras habilidades en excelencia.

**Actividades**

Para modelar los procesos proponemos utilizar un formato de tabla con cierta descripción de las actividades que se irán desarrollando a lo largo del proyecto.

| **Código Tarea** | **Actividad** | **Producto Resultado** | **Producto/s utilizados** | **Responsable** | **Rol Responsable** | **Doc. De Referencia** |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |

**Adaptación de procesos ágiles**

Para lograr un compromiso como equipo de trabajo, definimos nuestros valores y principios.

Valores como equipo de trabajo:

* Simplicidad
* Comunicación
* Entusiasmo
* Excelencia

Principios como equipo de trabajo:

1. Feedback constante.
2. Entendimiento compartido.
3. Entregas pequeñas y proceso continuo.

**Medidas de Calidad**

| **Práctica** | **Desarrollo** | **Objetivo** |
| --- | --- | --- |
| Planificación | Iteraciones de 2 semanas | Obtener feedback del usuario de forma rápida. Aplicar un proceso de desarrollo adaptable a las necesidades del cliente. |
| Simplicidad | Código simple y mantenible (se seguirán las prácticas que se detallan más adelante en el plan) | Se busca que el código sea simple y comentado para que podamos entender todos los integrantes del equipo que se está haciendo sin tener que perder el tiempo. |
| Pruebas | Testear el código por cada funcionalidad desarrollada | Se busca realizar pruebas constantemente durante el desarrollo para encontrar errores lo antes posible y evitar retrabajo. |
| Documentación y Comunicación | Comunicación constante con el cliente | Se busca que el cliente esté informado del progreso, y en caso de ser necesario, comunique los cambios requeridos |

**Planificación**

En el inicio de cada sprint se definirá un entregable (o varios) a ser completados. Los mismos tendrán alguna funcionalidad requerida por el cliente. Por este motivo es que al final de cada sprint y a modo de validación del mismo se solicitará al cliente que informe si el entregable está completo y si cumple con la funcionalidad requerida.

En el apartado de solución de conflictos se detalla las acciones a realizar en caso de no cumplir en su totalidad con el entregable

**Documentación y Comunicación**

En cada sprint se validará que se haya cumplido con las reuniones establecidas. A su vez, se asegurará que cada reunión tenga su documentación respaldatoria o cualquier documento de relevancia para lo tratado en la misma. Se controlará que sea entregada en tiempo y forma, que su contenido esté actualizado, que se encuentre con el formato definido al iniciar el proyecto y que sea concisa y simple.

Es considerado como una falta grave el no registro alguno de los entregables al final de cada sprint. En caso de que no exista dicha documentación revisar apartado solución de conflictos.

**Simplicidad**

Se estará generando código constantemente y el objetivo es que el mismo sea de calidad como asi también simple. Para ello nos basaremos en los principios del código limpio:

* Definiciones: Ya sea para nombrar variables, funciones, parámetros o clases se busca cumplir con dos aspectos básicos:
  + Deben ser precisos y entregar la idea central. Es decir, deben ir directos al punto.
  + El nombre de una función, parámetro, etc debe demostrar lo que realmente representa o lo que hace aunque el mismo sea extenso.
* Edición: Al ser un equipo, constantemente estaremos encontrando código que escribió otro. Esta regla define que si nos topamos con código poco claro, dejarlo más limpio de lo que estaba antes de editarlo.
* Funcionalidad: Para estructurar un código limpio, es necesario crear funciones simples, claras y pequeñas. Cada función debe tener una sola responsabilidad y delegar el resto de las necesarias en sub-funciones.
* DRY (Don´t Repeat Yourself): Este principio define que no pueden existir dos o más partes de la aplicación que realicen la misma función.
* Comentarios: Parte primordial del entendimiento del código son los comentarios acerca de las funcionalidades. Estos deben ser claros y estar siempre actualizados hasta las últimas modificaciones realizadas.
* Errores: Manejo avanzado de excepciones para el continuo funcionamiento de la aplicación y evitar errores inesperados.
* Pruebas: Deben ser rápidas, independientes entre sí, de respuesta concreta, repetibles y con tiempo definido.

**Pruebas**

Con respecto a las pruebas, como se mencionaba en uno de los principios del código limpio, deben ser:

* Rápidas: El test debe ser rápido, permitiendo que sea realizado muchísimas veces y en cualquier momento.
* Independiente: Debe ser independiente, con el fin de evitar que cause efecto cascada cuando ocurra alguna falla, lo que dificulta el análisis de problemas.
* Repetible: Debe permitir la repetición del test, muchísimas veces y en diferentes ambientes.
* Concretas: Los tests bien escritos deben retornar respuestas del tipo “true” o “false” para que el error no sea subjetivo.
* Tiempo definido: Los tests deben seguir estrictamente el criterio de puntualidad. Por eso los mismos son escritos antes del propio código, así evitamos que sea muy complejo para realizar el test posteriormente.

Para cada prueba se utilizará el siguiente formato:

| **Caso de Prueba** | |
| --- | --- |
| Objetivo | Breve descripción de lo que se intenta realizar |
| Tipo | ej: Unitaria, funcional, end to end |
| Identificador | ej: PU\_Login\_001 |
| Nombre del Caso | ej: Test de Log In |
| Tiempo de prueba | Demora y periodicidad |
| Responsable | Analista designado |
| Pasos a seguir | Resultado esperado por paso |

**Buenas Prácticas de Desarrollo**

* Clean Code / Código Limpio:

1. Usa nombres con significado

El código tiene que poder leerse como si fuera un libro. Para ello, los nombres de variables, métodos, clases tienen que seleccionarse con cuidado para que den significado a lo que estamos intentando "contar" en nuestro programa.

1. Haz unidades de código pequeñas

Las clases y métodos cortos simplifican la comprensión del código y lo hacen más mantenible

1. KISS Keep it simple, s… - Mantenlo simple. Las unidades de código deben hacer una única cosa

Muy relacionado con el punto anterior, si nuestras estructuras de código son demasiado grandes, es porque probablemente estén haciendo más de una cosa. Es lo que se conoce como el Principio de Responsabilidad Única, uno de los 5 principios SOLID, muy presentes en las ideas del código limpio.

1. Las funciones deben tener un número limitado de argumentos

Preferir las funciones sin parámetros sobre el resto. Un máximo de 3 parámetros es un buen límite.

1. Sigue el principio DRY: Don't Repeat Yourself

El problema de esto es que a veces la duplicidad de código no es tan evidente como podría parecer, así que tendremos que estar atentos y refactorizar cuando detectemos estos casos.

1. Evita utilizar comentarios siempre que sea posible.

Siempre que te veas en la necesidad de escribir un comentario, intenta reescribir el código o nombrar las cosas de otra forma, de tal forma que el comentario se vuelva irrelevante.

1. Utiliza un formato único en tu código

Cualquier formato, por extraño que sea, es mejor que ninguno en absoluto. Dará coherencia y estructura al código, y unas pautas a seguir por todos los desarrolladores. Algunos de ellos ya los hemos mencionado, como el tamaño de las funcione y las clases. Algunas otras cosas que puedes acordar:

1. Abstrae tus datos: no uses getters y setters indiscriminadamente

Es importante identificar si nuestra clase es una simple estructura de datos o es un objeto con comportamiento. Los objetos esconderán sus datos mediante abstracciones y expondrán funciones para operar con ellos. Por el contrario las estructuras de datos exponen su contenido pero sus funciones no realizan ninguna operación relevante.

1. ¿Conoces la Ley de Demeter?

Ley de Demeter: nuestro objeto no debería conocer las entrañas de otros objetos con los que interactúa

1. Lanza excepciones en lugar de devolver códigos de retorno

Lo idóneo es crearse excepciones propias que den un significado más semántico al error de un solo vistazo, y proveerlas de mensajes de error claros y descriptivos.

1. Establece fronteras

Las fronteras nos permiten establecer límites entre nuestro código y ese código que no controlamos

1. Escribe tests

Los tests ayudan a definir y validar la funcionalidad del software que estés implementando.

* SOLID:
  + S – Single Responsibility Principle (SRP) - Principio de Responsabilidad Única

***“A class should have one, and only one, reason to change.”***

La S del acrónimo del que hablamos hoy se refiere a **Single Responsibility Principle (SRP)**. Según este principio “una clase debería tener **una, y solo una, razón para cambiar**”. Es esto, precisamente, “razón para cambiar”, lo que Robert C. Martin identifica como “responsabilidad”.

El principio de Responsabilidad Única es **el más importante y fundamental de SOLID**, muy sencillo de explicar, pero el más difícil de seguir en la práctica.

El propio Bob resume cómo hacerlo: *“Gather together the things that change for the same reasons. Separate those things that change for different reasons”*, es decir: “Reúne las cosas que cambian por las mismas razones. Separa aquellas que cambian por razones diferentes”.

* + O – Open/Closed Principle (OCP)- Principio de Abierto/Cerrado

***“You should be able to extend a classes behavior, without modifying it.”***

El segundo principio de SOLID lo formuló Bertrand Meyer en 1988 en su libro “[Object Oriented Software Construction](https://sophia.javeriana.edu.co/~cbustaca/docencia/POO-2016-01/documentos/Object%20Oriented%20Software%20Construction-Meyer.pdf)” y dice: “Deberías ser capaz de extender el comportamiento de una clase, sin modificarla”. En otras palabras: las clases que usas deberían estar **abiertas para poder extenderse y cerradas para modificarse**.

En su blog Robert C. Martin defendió este principio que [a priori puede parecer una paradoja](https://blog.cleancoder.com/uncle-bob/2014/05/12/TheOpenClosedPrinciple.html). Es importante tener en cuenta el **Open/Closed Principle (OCP)** a la hora de desarrollar **clases,** [**librerías**](https://profile.es/blog/librerias-javascript/) **o frameworks.**

* + L – Liskov Substitution Principle (LSP) - Principio de Sustitución de Liskov

***“Derived classes must be substitutable for their base classes.”***

La L de SOLID alude al apellido de quien lo creó, [Barbara Liskov](https://www.eldiario.es/sociedad/Barbara-Liskov-medidas-mujeres-ingenierias_0_780422139.html), y dice que **“las clases derivadas deben poder sustituirse por sus clases base”.**

Esto significa que los objetos deben poder ser reemplazados por instancias de sus subtipos sin alterar el correcto funcionamiento del sistema o lo que es lo mismo: si en un programa utilizamos cierta clase, **deberíamos poder usar cualquiera de sus subclases** sin interferir en la funcionalidad del programa.

Según Robert C. Martin incumplir el **Liskov Substitution Principle (LSP)** implica violar también el principio de Abierto/Cerrado.

* + I – Interface Segregation Principle (ISP) - Principio de Segregación de la Interfaz

***“Make fine grained interfaces that are client specific.”***

En el cuarto principio de SOLID, *el tío Bob* sugiere: “Haz interfaces que sean específicas para un tipo de cliente”, es decir, para **una finalidad concreta**.

En este sentido, según el **Interface Segregation Principle (ISP),** es preferible contar con muchas interfaces que definen pocos métodos que tener una interfaz forzada a implementar muchos métodos a los que no dará uso.

* + D – Dependency Inversion Principle (DIP) - Principio de Inversión de Dependencias

***“Depend on abstractions, not on concretions.”***

Llegamos al último principio: “**Depende de abstracciones**, no de clases concretas”.

Así, Robert C. Martin recomienda:

1. Los módulos de alto nivel **no deberían depender de módulos de bajo nivel**. Ambos deberían depender de abstracciones.
2. **Las abstracciones no deberían depender de los detalles**. Los detalles deberían depender de las abstracciones.

El objetivo del **Dependency Inversion Principle (DIP)** consiste en reducir las dependencias entre los módulos del código, es decir, alcanzar un bajo acoplamiento de las clases.