Guía metodológica para la identificación e intervención de deuda técnica en un proyecto existente

David Santiago Castro Vargas.

I. INTRODUCCIÓN

El reconocimiento de procesos deteriorados llevo a indagar en la totalidad del ciclo de vida de un proyecto Informático, todas las fases del proyecto contenían síntomas de un deterioro en sus procesos, los diagnósticos dados se agruparon en el concepto de deuda técnica. Para la creación de esta guía, se realizó el ejercicio experimental de seleccionar un proyecto público para hacer un análisis fase por fase documentando, solucionando e interpretados los síntomas apoyándose en la bibliografía encontrada sobre deuda técnica.

En este ejercicio se usaron las herramientas y metodologías más apropiadas para enfrentar el problema, se documentó estrategias de identificación, prevención y capturadoras del problema. La razón de ser un proyecto desconocido es validar sus requerimientos e identificar si la deuda técnica fue consciente o inconsciente. Esta guía contiene enlaces de referencia y documentación de los pasos y procesos realizados. Toda la guía del proyecto se evidencia en el siguiente enlace: https://github.com/DavidCastro4444/Tennis-Refactoring-Kata.git

II. PASOS INICIALES

Para dar inicio al ejercicio se partió de un análisis del proyecto seleccionado, este análisis tuvo como cobertura la identificación de objetivos del proyecto, componentes, pruebas y smell code. Para esta parte inicial, se establecieron parámetros iniciales de un chequeo del proyecto como se evidencia en la tabla 1.1.

Versión de compilación	El proyecto corre bajo la versión java 8 mínimo para su compilación
Ejecución de pruebas	Por consola comando mvn test
Diagnóstico Clase TennisGame1	Code smells Decompose Conditional en las líneas 26 a 71 Replace Conditional with Polymorphism Remove Assignments to Parameters
Diagnóstico Clase TennisGame2	Code smells Consolidate Duplicate Conditional Fragments en las líneas 18 a 99 Decompose Conditional en las líneas 26 a 71 Replace Conditional with Polymorphism Remove Assignments to Parameters
Diagnóstico Clase TennisGame3	*Code smells* Extract Variable
Diagnóstico Clase TennisGame4	*Code smells* Extract Variable
Lógica del diseño de pruebas	La prueba hace uso de arrayList con la finalidad de generar diferentes posibles escenarios, una vez dimensionado los puntajes, se hace uso del polimorfismo ppor medio de una interfaz que generar patrones de comportamiento para las clases: TennisGame1, TennisGame2, TennisGame3 y TennisGame4

Tabla 1

III. CLEAN CODE Y XP PRACTICES

La deuda técnica no solo corresponde a un deterioro de proceso técnico, también hace referencia a los deterioros que se generan a partir de dificultades de lectura de código, agilidad de mantenibilidad y factores que se centran en la relación código desarrollador. Para superar estos obstáculos se plantearon estándares de buenas prácticas como Clean Code y XP Practices, que permiten un modelo a seguir al momento de escribir nuestro código.

Las características que se analizaron a lo largo del ejercicio se evidencian en la tabla 2

Código enfocado	El código malo hace demasiadas cosas, el código limpio es enfocado
Entendible	Nuestro código debe cumplir con los principios KIS y YAGNI
Escalable	El código se escribe para los desarrolladores y no para el computador. (SOLID, POO, GoF, etc.)

Duplicidad	No deben existir funciones con implementaciones duplicadas
Testeable	Debe tener pruebas unitarias y de aceptación
Principio menor asombro	Implementaciones en función de su nombre

Tabla 2

IV. Testing Debt

Para establecer una afirmación objetiva de código de calidad en el ejercicio, es fundamental entender el concepto de deuda técnica por Testing, estableciendo como base, más no única y ni aseverando que se haga de forma completa, las pruebas unitarias. Siendo tan importante la claridad y calidad del código, existen diferentes patrones para la creación y seguimiento del código, algunos de ellos relacionados con metodologías ágiles

.

Para este proyecto se identificó una clase de Test, que permite realizar y validar el comportamiento esperado de las diferentes clases utilizadas y diseñadas en el proyecto. En la tabla 3 se muestra las características tomadas en cuenta.

FAST	Su ciclo de vida debe ser corto, tiempo esperado no superior a 5 segundos.
ISOLATED	Su ciclo de vida no debe verse afectado por el funcionamiento o comportamiento de ninguna prueba externa o recurso.
REPEATABLE	Debe repetirse de manera indefinida y no verse afectado su comportamiento por eso.
SELF VALIDATION	Debe ser auto evaluable, esto quiere decir que la prueba misma representa el resultado y no genera ambigüedad para entenderlo, haciendo que sea false o true.
TIMELY	Va relacionado con el ciclo del proyecto, esta característica va relacionada con lo oportuno de la prueba, haciendo hincapié en la creación de pruebas antes de salir a producción como valor mínimo esperado.

Tabla 3

V. Better Code Hub

Better Code Hub es un servicio de análisis de código fuente basado en la web que verifica el cumplimiento de una base de código con las diez pautas presentadas en creación de software mantenible. Para estes ejercicio se realizó un proceso de análisis de código para obtener una visión técnica del proyecto evaluado. Los resultados arrojados se evidencian en la tabla 4

Write short units of code.	Las unidades son los grupos de código más pequeños que se pueden mantener y ejecutar de forma independiente. En Java, las unidades son métodos o constructores. Una unidad siempre se ejecuta como un todo. No es posible invocar solo unas pocas líneas de una unidad. Por lo tanto, la pieza de código más pequeña que se puede reutilizar y probar es una unidad
Write simple units of code.	La complejidad es una característica de calidad a menudo cuestionada. El código que parece complejo para un desarrollador externo o novato puede parecer sencillo para un desarrollador que está íntimamente familiarizado con él. En cierta medida, lo "complejo" está en el ojo del espectador. Sin embargo, hay un punto en el que el código se vuelve tan complejo que modificarlo se convierte en una tarea extremadamente arriesgada y que consume mucho tiempo, y mucho menos probar las modificaciones después. Para mantener el código mantenible, debemos poner un límite a la complejidad. Otra razón para medir la complejidad es conocer el número mínimo de pruebas que necesitamos para estar suficientemente seguros de que el sistema actúa de forma predecible. Antes de que podamos definir un límite de complejidad de código de este tipo, debemos poder medir la complejidad. Una forma común de evaluar objetivamente la complejidad es contar el número de rutas posibles a través de un fragmento de código. La idea es que cuantos más caminos se puedan distinguir, más complejo será un fragmento de código. Podemos determinar el número de caminos sin ambigüedades contando el número de puntos de bifurcación
Write code once.	Una filosofía adoptada por un compilador y sus bibliotecas de software asociadas o por una biblioteca de software/ marco de software que se refiere a la capacidad de escribir un programa de computadora que se puede compilar en todas las plataformas sin necesidad de modificar su código fuente
Keep unit interfaces small.	Hay muchas situaciones en la vida diaria de un programador donde las largas listas

	de parámetros parecen inevitables. En el apuro de hacer las cosas, puede agregar algunos parámetros más a ese método para que funcione en casos excepcionales. Sin embargo, a largo plazo, esta forma de trabajar conducirá a métodos difíciles de mantener y de reutilizar. Para mantener su código mantenible, es esencial evitar largas listas de parámetros o interfaces de unidades, limitando la cantidad de parámetros que tienen
Separate concerns in modules.	Es un principio de diseño para separar un programa de computadora en distintas secciones. Cada sección aborda una preocupación separada, un conjunto de información que afecta el código de un programa de computadora. Una preocupación puede ser tan general como "los detalles del hardware de una aplicación" o tan específica como "el nombre de la clase que se creará"
Couple architecture components loosely.	Tener una visión clara de la arquitectura del software es esencial cuando crea y mantiene software. Una buena arquitectura de software le brinda una idea de lo que hace el sistema, cómo lo hace y cómo se organiza la funcionalidad (es decir, en agrupaciones de componentes). Le muestra la estructura de alto nivel, el "esqueleto", por así decirlo, del sistema. Tener una buena arquitectura hace que sea más fácil encontrar el código fuente que está buscando y comprender cómo interactúan los componentes (de alto nivel) con otros componentes.
Keep your codebase small.	Una base de código es una colección de código fuente que se almacena en un repositorio, se puede compilar e implementar de forma independiente y es mantenida por un equipo. Un sistema tiene al menos una base de código. Los sistemas más grandes a veces tienen más de una base de código. Un ejemplo típico es el software empaquetado. Puede haber una base de código para la funcionalidad estándar, y hay diferentes bases de código mantenidas de forma independiente para complementos específicos del cliente o del mercado.
Automate tests.	En las pruebas de software, la automatización de pruebas es el uso de software separado del software que se está probando para controlar la ejecución de las pruebas y la comparación de los resultados reales con los resultados previstos. [1] La automatización de pruebas puede automatizar algunas tareas repetitivas pero necesarias en un proceso de prueba formalizado que ya existe, o realizar pruebas adicionales que serían difíciles de realizar manualmente. La automatización de pruebas es fundamental para la entrega continua y las pruebas continuas
Write clean code.	Se considera que el código es limpio (en inglés, clean code) cuando es fácil de leer y entender. Si resuelve los problemas sin agregar complejidad innecesaria, permitiendo que el mantenimiento o las adaptaciones, por algún cambio de requerimiento, sean tareas más sencillas, entonces estamos hablando de “clean code”. Para crear código limpio hay que conocer y poner en práctica un conjunto de principios o técnicas de desarrollo que nos ayudarán a evitar los code smells, es decir, esos síntomas de un programa que te dan el indicio de que existe un problema más profundo.

Tabla 4.

VI. Continuos Integration

La integración continua (CI) es una práctica de software que requiere enviar código con frecuencia a un repositorio compartido. Confirmar código con mayor frecuencia detecta errores antes y reduce la cantidad de código que un desarrollador necesita depurar para encontrar el origen de un error. Las actualizaciones frecuentes de código también facilitan la combinación de cambios de diferentes miembros de un equipo de desarrollo de software. Esto es excelente para los desarrolladores, que pueden pasar más tiempo escribiendo código y menos tiempo depurando errores o resolviendo conflictos de fusión.

Cuando confirma código en su repositorio, puede compilar y probar continuamente el código para asegurarse de que la confirmación no introduzca errores. Sus pruebas pueden incluir filtros de código (que verifican el formato de estilo), controles de seguridad, cobertura de código, pruebas funcionales y otros controles personalizados.

Construir y probar su código requiere un servidor. Puede compilar y probar actualizaciones localmente antes de enviar el código a un repositorio, o puede usar un servidor de CI que verifica las nuevas confirmaciones de código en un repositorio

VII. CI USANDO GITHUB ACTIONS

CI usando GitHub Actions ofrece flujos de trabajo que pueden construir el código en su repositorio y ejecutar sus pruebas. Los flujos de trabajo pueden ejecutarse en máquinas virtuales alojadas en GitHub o en máquinas que usted mismo aloja.

Puede configurar su flujo de trabajo de CI para que se ejecute cuando se produzca un evento de GitHub (por ejemplo, cuando se inserte código nuevo en su repositorio), en un horario establecido o cuando se produzca un evento externo mediante el webhook de distribución del repositorio

GitHub ejecuta sus pruebas de CI y proporciona los resultados de cada prueba en la solicitud de extracción, para que pueda ver si el cambio en su rama introduce un error. Cuando todas las pruebas de CI en un flujo de trabajo pasan, los cambios que impulsó están listos para ser revisados por un miembro del equipo o combinados. Cuando falla una prueba, uno de sus cambios puede haber causado la falla. En la tabla 5 se evidencian los componentes para la realizar el proceso.

File YAML	Los archivos de flujo de trabajo utilizan la sintaxis YAML y deben tener una extensión de archivo .ymlo .yaml Debe almacenar archivos de flujo de trabajo en el .github/workflowsdirectorio de su repositorio.
Name	El nombre de su flujo de trabajo. GitHub muestra los nombres de sus flujos de trabajo en la página de acciones de su repositorio. Si omite name, GitHub lo establece en la ruta del archivo de flujo de trabajo en relación con la raíz del repositorio. La forma para correr la prueba unitaria se aconseja que sea por consola
On	Para desencadenar automáticamente un flujo de trabajo, use on para definir qué eventos pueden hacer que se ejecute el flujo de trabajo. Puede definir eventos únicos o múltiples que pueden desencadenar un flujo de trabajo o establecer un cronograma. También puede restringir la ejecución de un flujo de trabajo para que solo se produzca en archivos, etiquetas o cambios de rama específicos.
Filtros	Algunos eventos tienen filtros que le brindan más control sobre cuándo debe ejecutarse su flujo de trabajo. Por ejemplo, el push evento tiene un branches filtro que hace que su flujo de trabajo se ejecute solo cuando se produce una inserción en una rama que coincide con el branches filtro, en lugar de cuando se produce cualquier inserción.
on.<pull_request|pull_request_target>.<branches|branches-ignore>	Al usar los eventos pull_requesty pull_request_target, puede configurar un workflow para que se ejecute solo para solicitudes de extracción dirigidas a ramas específicas. Utilice el branchesfiltro cuando desee incluir patrones de nombres de sucursales o cuando desee incluir y excluir patrones de nombres de sucursales. Utilice el branches-ignorefiltro cuando solo desee excluir patrones de nombre de rama. No puede usar los filtros branchesy branches-ignorepara el mismo evento en un flujo de trabajo.
Jobs	Un job es un conjunto de pasos en un workflow que se ejecutan en el mismo ejecutor. Cada paso es un script de shell que se ejecutará o una acción que se ejecutará. Los pasos se ejecutan en orden y dependen unos de otros. Dado que cada paso se ejecuta en el mismo corredor, puede compartir datos de un paso a otro. Por ejemplo, puede tener un paso que cree su aplicación seguido de un paso que pruebe la aplicación que se creó.
Build	Al construir nuestro proyecto, se estipulan pasos como versión, máquina virtuál, dependencias o cualquiér catacterística que sea necesaria para correr el proyecto. Por esta razón, dentro del jobs en el scope de build, se cinfiguran estas características
Análisis de código	Una característica importante, es la capacidad de automatizar el análisis de código a nuestro proyecto, este facilita la calidad de nuestro repositorio y verificar inconsistencias a tendencias establecidas
Pruebas	Nuestro proyecto debe contar con un escenarios de pruebas, para esta funcionalidad se realiza una actividad programable dentro del workflow, sistematizando las pruebas y ejecutandolas de manera automática.

Tabla 5.

VIII. CICLOS GUIADOS POR ARQUITECTURA

Es un ciclo en donde los objetivos de negocios y los atributos de calidad del producto conducen el diseño de la arquitectura, y ésta es la base para la definición del resto del ciclo de producción y evolución a partir de

- Definir la estructura del proyecto (ciclos de vida, estimaciones, conformación de equipos, plan de comunicación, estrategia de configuration management, plan de pruebas, estrategia de integración e implantación del producto).

-
- Definir la estrategia de integración entre proveedores.
-
- Definir los mecanismos de coordinación entre grupos ubicados en diferentes locaciones.
-
- Definir la estrategia de transferencia de conocimiento a grupos de mantenimiento.

El Software Engineering Institute (SEI) viene trabajando desde hace más de diez años en la definición de métodos para soportar los ciclos de vida guiados por la arquitectura. Son métodos prácticos que se sustentan en el uso de escenarios. Un escenario es una instancia concreta del uso del sistema y se compone de estímulo, elemento estimulado, resultado esperado en base a mediciones y el ambiente en donde el escenario se produce.

IX. QAW

Es un taller (workshop) en donde se integran los diferentes involucrados para identificar los atributos de calidad que serán drivers del diseño de arquitectura del producto. QAW facilita la resolución temprana de conflictos, obtiene consensos entre los stakeholders y ayuda a mejorar los requerimientos a todos los niveles.

X. ATAM

Es un método altamente estructurado para evaluar un diseño de arquitectura. ATAM permite detectar, de manera temprana, riesgos técnicos, conflictos entre atributos, puntos sensitivos del diseño y soluciones. Basado en el proyecto guía, se realizaron las siguientes fichas de escenarios.

Scenario Refinement for Scenario N	
Scenario(s):	El aplicativo se encuentra bajo una interacción constante de peticiones, y en diferentes procesos de juego por solicitud de marcadores
Business Goals:	Dar los resultados del juego, de manera correcta y en tiempos cortos para no obtaculisar la fluidez del juego
Relevant Quality Attributes:	Performance, usability
Stimulus:	Cambios en el score del juego
Stimulus Source:	Proceso externo monitoriado por el aplicativo
Environment:	Partida en tiempo real
Artifact (if known):	Servicios de cambio de marcador
Response:	Actualización del marcado de manera correcta
Response Measure:	95% de estabilidad
Questions:	¿Qué pasa si una respuesta genera un cuello de botella? ¿Plan de contingencia si da un resultado erroneo?
Issues:	Corrección de marcadores

Figura 1.

Scenario Refinement for Scenario N	
Scenario(s):	El aplicativo se encuentra bajo una interacción constante de peticiones, y en diferentes procesos de juego por solicitud de marcadores
Business Goals:	Dar los resultados del juego, de manera correcta y en tiempos cortos para no obtaculisar la fluidez del juego
Relevant Quality Attributes:	Performance, usability
Stimulus:	Cambios en el score del juego
Stimulus Source:	Proceso externo monitoriado por el aplicativo
Environment:	Partida en tiempo real
Artifact (if known):	Servicios de cambio de marcador
Response:	Actualización del marcado de manera correcta
Response Measure:	95% de estabilidad
Questions:	¿Qué pasa si una respuesta genera un cuello de botella? ¿Plan de contingencia si da un resultado erroneo?
Issues:	Corrección de marcadores

Figura 2.

XI. CONCLUSIONES

La deuda técnica es un concepto que se usa en desarrollo para definir el coste de mantener y arreglar un software mal construido, a menudo por hacerlo rápido o por no haber llevado a cabo un buen control de calidad antes de lanzarlo. Al realizar un ejercicio de selección de un proyecto público se evidencian todas las aristas y perspectivas que no se tuvieron en cuenta para la realización de este, mostrando que la deuda técnica es un valor poco trabajado en los proyectos y se obtuvieron las siguientes conclusiones.

- Es importante tener una estrategia que administre la deuda con prudencia y minimice la deuda imprudente.
-
- Uso de herramientas actualizadas para medir y analizar los errores.
-
- Hacer un buen uso de patrones fuertes y detectables mediante división en clases, métodos comprensibles.
-
- Realizar procesos estandarizados para la refactorización y la gestión de calidad.

Una de las maneras de controlar la deuda técnica es crear conciencia sobre esta, tanto en el la parte comercial como en los equipos de desarrollo de software. La deuda tecnológica debe gestionarse adecuadamente para minimizar el riesgo asociado.

XII. REFERENCIAS

- N. Zazworka, C. Seaman, and F. Shull, “Prioritizing design debt investment opportunities,” in Proceeding

of the 2nd working on Managing technical debt - MTD ’11, 2011, p. 39.

- Santos, J. A. M., Rocha-Junior, J. B., Prates, L. C. L., do Nascimento, R. S., Freitas, M. F., & de Mendonça, M. G. (2018). A systematic review on the code smell effect. Journal of Systems and Software, 144, 450-477.
- Di Nucci, D., Palomba, F., Tamburri, D. A., Serebrenik, A., & De Lucia, A. (2018, March). Detecting code smells using machine learning techniques: are we there yet?. In 2018 ieee 25th international conference on software analysis, evolution and reengineering (saner) (pp. 612-621). IEEE.
- Saleh, S. M., Huq, S. M., & Rahman, M. A. (2019, February). Comparative study within Scrum, Kanban, XP focused on their practices. In 2019 International Conference on Electrical, Computer and Communication Engineering (ECCE) (pp. 1-6). IEEE.
- Vallon, R., da Silva Estácio, B. J., Prikladnicki, R., & Grechenig, T. (2018). Systematic literature review on agile practices in global software development. Information and Software Technology, 96, 161-180.