Deuda Técnica en el contexto de GeneXus

Introducción **2**

Deuda técnica **2**

Definición 3

Tipos de deuda técnica 4

Causas de la deuda técnica 5

Gestión y planificación de deuda técnica 6

Contexto del desarrollo con GeneXus **8**

Percepción de la deuda técnica en GeneXus **9**

Recomendaciones para la gestión de la deuda técnica en el desarrollo con GeneXus **11**

Identificación 11

Comunicación 14

Medición y monitoreo 14

Prevención 15

Pago 15

Referencias **16**

Anexo 1 **17**

Herramientas para la gestión de la deuda técnica con GeneXus **17**

KBDoctor 17

Soporte para DT de código 17

Soporte para DT de arquitectura 18

LSIExtensions 18

GXtest - cobertura de tests 18

Introducción

El término deuda técnica (DT) en el contexto de desarrollo de software fue acuñado por primera vez por Ward Cunningham en 1992 en su artículo [1] en el cual realiza una analogía de la deuda técnica con obtener dinero prestado y su generación de intereses, en la cual según cuenta está bien endeudarse un poco (generar deuda técnica) si esto permite un rápido desarrollo, siempre y cuando la misma sea pagada (solucionada) rápidamente y genere poco interés, porque caso contrario es donde se presenta el peligro.

La deuda técnica no tiene una definición específica, sino que se refiere a un concepto, idea o metáfora, tampoco implica ciertas acciones en concreto ni un momento del tiempo dado y generalmente no puede aislarse del contexto donde surge.

En base a esto, en lo que sigue, se intenta bajar a tierra dicho concepto, explorar en el contexto de GeneXus como plataforma o entorno low code y determinar cómo sus particularidades afectan la generación, percepción y gestión de la deuda técnica.

Deuda técnica

Se puede decir que, para definir o detectar la deuda técnica (DT) en el desarrollo de software, es importante entender el punto de referencia temporal donde se está parado, ésta detección se realiza cuando las decisiones, acciones o cambios de contexto que pudieran darse en el proceso de desarrollo ya han ocurrido y nos encontramos en una situación en la que estos elementos han generado cierto perjuicio o costo a nivel de esfuerzo, tiempo, etc. que están dificultando o imposibilitando la evolución del software. Este transcurso de tiempo entre la acción, la detección y la posible corrección determinan que en el normal flujo de los procesos o iteraciones del desarrollo de software se vayan acumulando o arrastrando las consecuencias de las DT, incurriendo en una especie de “interés” o sobre costo a la hora de corregir la situación que en ciertos casos se podría haber evitado si no se hubiera generado la DT o si hubiera sido solucionada inmediatamente fuese detectada.

Estas decisiones tomadas en el pasado que hoy implican deuda técnica pudieron ser intencionales ya a sabiendas de la posible generación de DT buscando un beneficio a corto plazo o involuntarias con total seguridad de que se están haciendo las cosas perfectamente pero debido a cambios en el contexto, por ejemplo o el simple paso del tiempo generen DT.

Tiene sentido hablar de deuda técnica en el entendido de que al hablar de desarrollo de software este tiene ciclos de evolución o iteración, por tanto un software el cual no estuviese sujeto a cambios a lo largo del tiempo carece de relevancia en cuanto a deuda técnica se refiere, pues difiere del concepto de perjuicio o costo mencionado anteriormente y no habría deuda o interés a pagar ya que no hay cambios en el software.

En todos los casos la cuestión termina recayendo en la importancia de la visibilización y ser conscientes de la existencia de la deuda técnica, de esta forma permitir el análisis del impacto y costos en el proyecto para una adecuada gestión a futuro y de esta forma asegurarse que sea “pagada” lo antes posible, porque al pasar el tiempo y avanzar en las iteraciones del proceso de software y su evolución, se van acumulando “intereses” en la forma de aumento de los costos o consecuencias de no aplicar las soluciones o correcciones en tiempo y forma.

Es importante mencionar que lo anterior es independiente a que el software funcione de forma óptima y no contenga errores o defectos. La deuda técnica no se trata de errores o bugs y tampoco se trata solo de calidad interna, la DT se encuentra en cualquier aspecto que afecte la adecuada evolución del software.

La deuda técnica es un factor invisible para el usuario externo o consumidor del producto de software y solo es visible para aquellos que trabajen en el desarrollo y mantenimiento del software.

Definición

En el metamodelo presentado en la Figura 1, extendido por Jerónimo Junior en [2], se pretende identificar todos aquellos elementos que de cierta forma intervienen directa o indirectamente en lo que refiere a deuda técnica dentro del proceso de desarrollo de software.

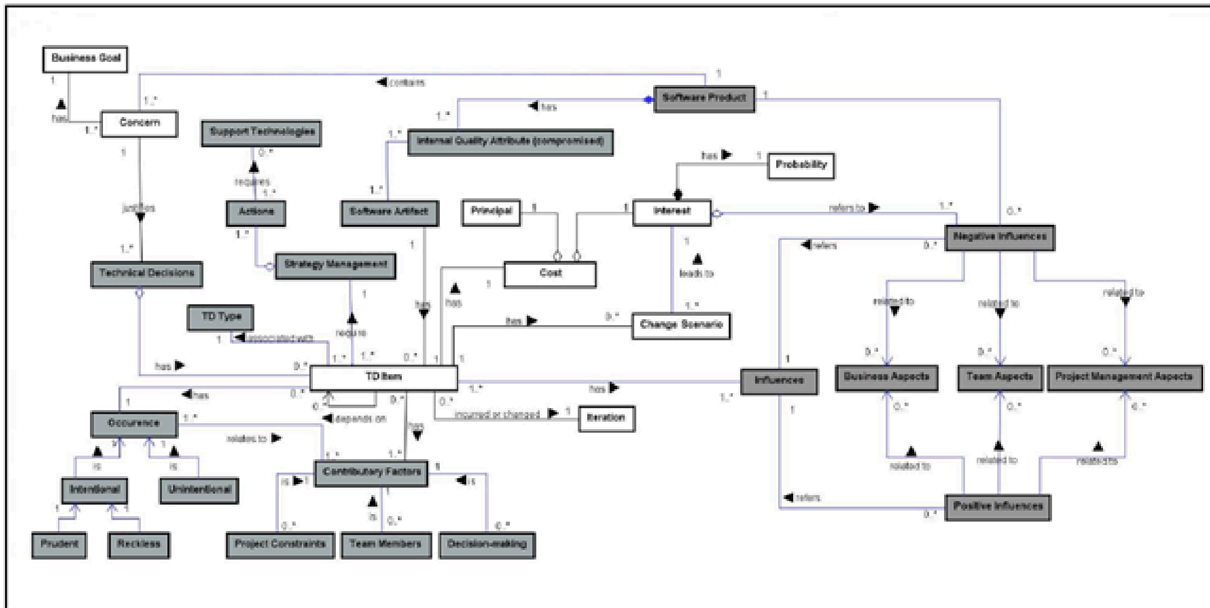

Figura 1 - Metamodelo de la deuda técnica

El elemento principal de este metamodelo es el llamado *TD Item*, el cual se puede definir como un elemento de deuda técnica concreto ya detectado, en donde pueden apreciarse las

relaciones que éste tiene con otros elementos como el Costo, Tipo, Influencia, Ocurrencia y Factores contributivos.

Tipos de deuda técnica

La deuda técnica es un concepto que se categoriza desde diferentes perspectivas, una de ellas es la que se presenta a continuación, tomando como referencia el tipo de artefacto en el cual es detectada:

- Arquitectónica

Refiere al tipo de deuda técnica que puede ser encontrada a nivel de la arquitectura de un producto de software, aquí se pueden encontrar cuestiones como por ejemplo, dependencias con comportamiento complejo o violaciones de las buenas prácticas arquitectónicas.

- Código

Esta categoría, refiere a la deuda técnica a nivel de codificación, teniendo como ejemplo el código espagueti, violaciones a las buenas prácticas de codificación o los llamados *code smells*.

- Build

Por la categoría build, se encuentran tipos de deuda técnica referidas al proceso de build o de compilación/implantación de software, como podría ser incluir archivos innecesarios a la hora de hacer build, o archivos versionados que complejizan la estructura.

- Documentación

En esta categoría se engloba todo lo que respecta a documentación del proceso de software, ya sea por la falta, la claridad o la no utilización de maquetación para la misma.

- Requerimientos

Esta categoría de deuda técnica hace referencia al proceso de relevar, definir, acordar y acotar los requerimientos del producto de software, generando deuda técnica por ejemplo aquellos requerimientos relevados de forma poco clara, o de forma ambigua o no especificados correctamente.

- Servicio

En esta categoría se engloban los tipos de deuda técnica que refieren al uso y manipulación de servicios, tanto sea su consumo como su disponibilización.

- Test

Por categoría de test, se entiende la DT generada por un insuficiente o inadecuado testeo, cobertura o metodología de pruebas.

Otras categorías que no son tan comunes, pueden ser de usabilidad, de personas, de infraestructura, de procesos o incluso de defectos; este último en lo que respecta a decisiones de postergación de solución a los mismos.

Causas de la deuda técnica

Las causas de la generación de deuda técnica son variadas y en algunos casos se corresponden con los tipos de deuda técnica mencionados anteriormente.
Hay ciertas causas que son más notorias o recurrentes y que podrían clasificarse dentro de las siguientes categorías o grupos:

- Carencia de conocimiento

En todos los pasos o tareas que componen la construcción de software, la carencia de conocimiento referido al cómo se realiza una acción o como realizarla de forma correcta, ya sea a nivel de codificación, diseño, documentación, etc. tiende a ser una causa importante de generación de deuda técnica.

- Planificación o gestión

Cuando los proyectos tienen falencias de planificación o de gestión, se deriva en falta de entendimiento por parte de los equipos de trabajo, desorganización a nivel general y presiones de tiempo, elementos que afectan en gran medida a la generación de deuda técnica.
Además la forma en que se organizan los equipos y el trabajo así como las metodologías aplicadas en los procesos también juegan un papel relevante contribuyendo mediante su carencia o aplicaciones cuando no corresponden. Como por ejemplo definir el rol desarrollador del backend a una persona que no tiene experiencia en dicha área o utilizar una metodología de segmentación dentro de un proceso que implique que ciertas personas estén ociosas mientras otras tienen una presión de tiempo para derivar el trabajo a su contraparte.

- Tecnologia e infraestructura

La utilización de tecnologías inadecuadas por ser insuficientes o excesivamente complejas en relación al proyecto, por ser obsoletas, deprecadas, o simplemente no contribuir en el producto a desarrollar son factores que aumentan el riesgo de generación de deuda técnica, así también el uso inadecuado, configuración deficiente de los entornos de trabajo o de implementación del software.

- Personas

Causas propias de las personas que intervienen en el proceso de desarrollo, a nivel individual como grupal, del estilo de falta de compromiso, interacción o comunicación, falta de trabajo en equipo, rispideces, etc. también son factores que contribuyen en la generación de deuda técnica.

- Desarrollo o codificación

Los factores relacionados a la codificación o programación son un grupo importante de causas de generación de deuda técnica, estos están directamente involucrados con la calidad interna y pueden darse por complejidad en el código, workarounds, code smells violaciones a las buenas prácticas, etc. Es importante recordar que no se refiere a errores de código o falta de funcionalidades.

- Factores externos

Aquí, los cambios de políticas o de gestión a nivel superior, los cuales escapan de las personas que integran el proceso de desarrollo, cambios en el contexto, o el simple paso del tiempo, pueden poner al descubierto o generar tipos de deuda técnica que no han sido o no pudieron ser contemplados hasta ahora y su causa escapa de las categorías anteriores y del proceso en sí mismo pero dado que se habla siempre de productos de software sujetos a iteración de procesos o evolución, se puede encontrar deuda técnica por estos motivos.

Gestión y planificación de deuda técnica

Una de las tantas formas de organizar las actividades de gestión de la deuda técnica podría ser agrupando las mismas en 3 etapas, las cuales buscan en conjunto definir un proceso para mitigar los efectos generados por la DT en todas las etapas del desarrollo de software y así mismo aprovechar las ventajas que pueden generarse al incurrir en aquellos tipos de deuda técnica que de forma consciente y planificada tienen como objetivo agilizar procesos, o un rápido prototipado por ejemplo.

La primera etapa o medida consiste en visibilizar y difundir la DT a los demás integrantes del proceso de desarrollo, toda aquella DT generada o detectada debe ser sociabilizada, de esta forma haciendo partícipe al resto y permitiendo que se analicen y discutan los impactos, medidas y prioridades de solución, además de discutir y difundir medidas de prevención. Se pueden identificar aquí por ejemplo las siguientes actividades:

- Identificación: Se identifica la DT determinando sus causas por ejemplo que hayan sido por decisiones técnicas y si las mismas fueron intencionales o no intencionales.
- Comunicación: Visibilizar la DT identificada a las personas involucradas en los procesos de desarrollo.
- Prevención: Establecer prácticas de prevención para evitar incurrir en DT nuevamente.

La segunda etapa parte de la base de que una vez que la DT fue visibilizada, esta debe ser registrada, de forma de darle un seguimiento, tener una comprensión de la misma y cuantificar de cierta medida el alcance o costo tanto de mantenerla como de solucionarla. Se pueden identificar aquí por ejemplo las siguientes actividades:

- Medición: Se evalúa la relación costo/beneficio de la DT detectada y/o se estima en relación al proyecto en general.

- Priorización: Se adoptan reglas predefinidas para realizar un ranking de prioridades que ayuden a la hora de decidir pagarla.
- Monitoreo: Observar y monitorear a lo largo del tiempo la evolución de la DT detectada.
- Representación/Documentación: Documentar la DT representándola o codificándola en algún estándar predefinido para un mejor entendimiento de las partes involucradas.

Por último, partiendo del análisis anterior se procede a la reducción de dichos elementos de deuda técnica en la medida que sea posible y en base a las prioridades fijadas, siempre con la estrategia clara de reducir la DT lo antes posible, de esta forma evitar lo mencionado anteriormente como “pago de intereses” que las misma conlleva. Se puede identificar aquí la siguiente actividad:

- Pago/solución: consiste en eliminar o reducir el impacto de la DT.

Estas etapas y actividades [3] no son un proceso independiente del desarrollo de software, sino que deben estar incluidas en cada iteración del mismo.

Otro modo organizacional, en vez de la agrupación por etapas podría ser directamente la encadenación de las actividades mencionadas anteriormente en un ciclo continuo teniendo como actividad permanente la Prevención, tal como se muestra en la Figura 1.

Figura 1 - Actividades de gestión de la deuda técnica

Contexto del desarrollo con GeneXus

GeneXus es una plataforma propietaria de desarrollo autodefinida como Low-Code, la cual está enfocada en simplificar y automatizar la creación de aplicaciones de software ya sea web, de escritorio o móvil [4].

Se basa en un proceso ágil de 3 pasos, modelado, generación y validación, Principalmente es una herramienta de desarrollo basado en modelos, pero aplicada a su propia forma de modelar, utilizando un lenguaje descriptivo para hacerlo donde los modelos creados son en base al conocimiento que tienen los usuarios de su negocio.

GeneXus consta de un IDE o interfaz de desarrollo la cual permite la creación de artefactos de software propios de la plataforma tanto para el manejo de procedimientos, lógica de negocio o persistencia de datos. Por otro lado, cada artefacto permite definir reglas o agregar código en su propio lenguaje siendo este último siempre reducido a su mínima expresión, por eso entra en la categoría Low-Code.

Estos artefactos componen lo que se llama una Base de Conocimiento en GeneXus (KB por sus siglas en inglés) que es la que almacena toda la información que el usuario define, para luego generar la correspondiente aplicación en el lenguaje objetivo que se desee (Java, .Net, etc.). Así mismo, se encarga de comunicarse con el motor de base de datos que corresponda y crear el esquema relacional en cuestión, en base a los artefactos definidos en la KB.

Lo anterior engloba las particularidades propias de esta tecnología o plataforma, estas diferencias tendrán un rol importante, ya que diferencian al desarrollo con GeneXus de otros procesos de desarrollo de software con otras tecnologías en cuanto a deuda técnica se refiere. Por tanto, los características importantes de GeneXus a tener en cuenta a la hora de hablar de DT:

- Baja cantidad de codificación o escritura de código
- Basado en Modelos pero no estándares
- Artefactos y conceptos de software exclusivos de la plataforma.
- Autogeneración de código en lenguaje objetivo.
- Autogeneración de esquema relacional en motor de base de datos.
- Plataforma integral con su propio y único entorno de desarrollo.
- Software propietario.
- Enfoque de metodología en 3 pasos.
- Enfocado al conocimiento del usuario del negocio y no al conocimiento en desarrollo de software.
- Ambiente colaborativo propio y no intercambiable.

Percepción de la deuda técnica en GeneXus

En este punto se desarrollarán ciertas particularidades de la deuda técnica al llevar adelante un proyecto con GeneXus. Para esto se tendrán en cuenta las clasificaciones de las causas de la deuda técnica comentadas en la sección *Causas de la deuda técnica*.

Es importante destacar que si bien un desarrollador podría percibir mayoritariamente la deuda técnica en artefactos como el código o la arquitectura, ya que estos son los más relacionados a su tarea principal, no implica que estos sean los únicos artefactos donde un desarrollador puede percibir la deuda técnica. Cuando se trabaja en un proyecto y dentro de un equipo también se puede detectar la deuda técnica perteneciente a las otros tipos.

A continuación se hará un análisis de la deuda técnica agrupando según las causas que pueden generarlas, pero teniendo en cuenta particularidades en GeneXus. Esto implica que si no se desarrolla o explicitan algunos ítems en particular no se pretende indicar que no ocurran en el desarrollo con GeneXus, sino que no se encontraron particularidades al compararlo con el desarrollo en otras tecnologías.

<u>Carencia de conocimiento</u>

- Es posible que en algunas empresas se contraten desarrolladores que solo tienen como capacitación el curso de "Analista GeneXus" ofrecido por la propia empresa que desarrolla y da mantenimiento a la plataforma. Más allá de los conocimientos técnicos que uno puede obtener de dicho curso, estos desarrolladores pueden no manejar otras áreas como la de gestión de proyectos, redes de computadoras o bases de datos. Por lo tanto esto se podría ver como una carencia de ciertas bases del conocimiento de desarrollo de software de algunos desarrolladores GeneXus.

<u>Planificación o gestión</u>

- Una característica interesante de GeneXus es la posibilidad de independencia de los desarrolladores. El hecho de que sea una plataforma low code puede permitir que un desarrollador lleve adelante más tareas en comparación a otros lenguajes: análisis de los requisitos, diseño de la arquitectura, desarrollo tanto de backend como frontend, e inclusive testing. Debido a esto se podría llevar adelante un proyecto con menos recursos humanos y en menos tiempo. Esto implica que, sin buena comunicación o gestión del proyecto, un desarrollador podría cometer errores que comprometan la calidad interna del software, generando de forma no intencional código de mala calidad o degradando la estructura de la arquitectura.

<u>Tecnologia e infraestructura</u>

- *Versiones de la plataforma*
  En la actualidad GeneXus tiene disponible y en mantenimiento varias versiones, donde se requiere de licencias de uso para cada una de estas. El hecho de que sea una plataforma paga es una diferencia considerable con muchas otras herramientas de programación. Esto tiene como ventaja que cuando uno paga por una licencia, además

del uso de la misma se está accediendo a un servicio de garantía y mantenimiento ofrecido por el proveedor. Por otra parte, un inconveniente que puede traer el requerir de licencias, es que cada una y cada versión implica un gasto (principalmente económico, dependiendo del plan contratado) que se debe realizar. Esto puede ser un factor a tener en cuenta al momento de decidir si migrar de versión o no. Otro factor que puede influir es la cantidad de modificaciones que pueden sufrir los objetos de la base de conocimiento al momento de migrar entre dos versiones, produciendo posibles errores en el funcionamiento o visualización; lo que implicaría tiempo de trabajo corrigiendo los cambios.

- *Actividad en el marketplace*
  GeneXus ofrece su propio marketplace [5], una plataforma en donde se pueden conseguir patrones, extensiones u objetos externos para incorporarlos al propio sistema. Algunos se encuentran gratuitos y otros no. Puede ocurrir que algunas extensiones se deprequen, o dejen de funcionar de forma correcta con el paso del tiempo o al pasar a una versión más reciente.
- *Limitantes de la tecnología*
  Debido a al diseño de la plataforma, GeneXus aún no ha podido resolver de forma correcta u óptima algunas de sus funcionalidades. Es por esto que en ocasiones los desarrolladores se ven obligados a buscar algún *workaround*, o inyectar código nativo dentro del código Genexus.
- *Falta de un repositorio*
  Genexus ofrece la plataforma GXServer como herramienta de versionado de proyectos. Su objetivo principal es el de lograr compartir de forma remota un proyecto entre varios desarrolladores. Esta herramienta se encuentra disponible desde la versión Genexus Evolution 3, lo que implica que versiones anteriores a esta no pueden hacer uso de la plataforma. Además se debe tener en cuenta que no todas las funcionalidades que se tienen disponibles actualmente en GXServer existen desde que fue lanzado, sino que se fueron agregando con el paso del tiempo y de las versiones de Genexus; por lo tanto solo las versiones más recientes de GeneXus (aproximadamente Gx 15) pueden hacer uso de todas las funcionalidades de GXServer. Al comparar GXServer con las herramientas de repositorios y control de versionado más populares (GitHub y GitLab), se puede notar que la correspondiente a Genexus aún no se encuentra a la altura de las otras. Las principales diferencias que se encontraron son la falta de Pull Requests y, a nuestro parecer, la poca facilidad para gestionar las ramas dentro de las versiones.

Personas

- No se encontraron particularidades en este aspecto en relación a Genexus.

Desarrollo o codificación

- *Manejo de objetos*
  Lógica de negocios en objetos de frontend: Por razones de complejidad o simplicidad al momento de desarrollar y luego generar nuevas versiones de una aplicación se recomienda implementar la lógica de negocio del lado del servidor. Debido a que GeneXus permite implementar tanto la capa de presentación (frontend) como la capa de

acceso a datos (parte del backend) en un solo lugar, puede ser más fácil agregar la lógica de negocio en las pantallas del usuario (transacciones, webpanels, etc), ya sea por simplicidad al momento de resolver un problema, o quizás no se tenga bien definido estándares al momento de implementar.

Manejo de reglas
Debido al paradigma de programación en el que se basa GeneXus, la sección de las reglas es una parte principal al momento de desarrollar y además un diferenciador a muchos otros lenguajes. A veces dependiendo del requerimiento, puede ocurrir que algunas reglas se tornen complejas o de gran tamaño. Causando dificultad al comprender el código al momento de una revisión.

- *Manejo de la arquitectura*
  Debido a la "cercanía" que se encuentran los desarrolladores a la arquitectura (en algunas versiones se debe pasar de prototipo a diseño, en otras solo se debe modificar una transacción) uno podría generar cambios en ella con mayor facilidad que en otras tecnologías, al estar resolviendo un problema en particular como por ejemplo el largo de un campo de tipo string, sin considerar el sistema en su totalidad, lo que a futuro podría afectar el funcionamiento de otros procedimientos o reportes.

Factores externos

- *Poca actividad de la comunidad*
  Los foros públicos (espacio donde la comunidad consulta y opina) no ha demostrado mucha actividad en comparación a la participación que tienen aquellos que trabajan en otros lenguajes. En particular, Stack Overflow, una de las plataforma más conocidas para el intercambio de conocimiento entre desarrolladores no demuestra mucha información ni actividad cuando se habla de Genexus.
- *Documentación*
  Si bien GeneXus maneja una plataforma Wiki [6] donde la propia empresa que ofrece el entorno de programación define y documenta funcionalidades y también permite los aporte de usuarios externos, se considera que la plataforma no tiene tanta especificación, desarrollo o ejemplos en los artículos existentes.

Recomendaciones para la gestión de la deuda técnica en el desarrollo con GeneXus

En esta sección se presentan distintos componentes de software que pueden ayudar en las distintas actividades de gestión de la deuda técnica, atacando diferentes tipos de DT y sus causas. A modo de resumen, hay herramientas en forma de extensión del IDE de GeneXus, instalables por fuera del IDE y objetos que permiten evitar y/o gestionar de mejor forma la DT. Estos componentes del ecosistema GeneXus se resumen en la Tabla 1. Vale mencionar que algunas de ellas no fueron probadas en la práctica y los datos que se mencionan son en base a su descripción y especificaciones.

Herramienta/componente/artefacto	Tipo	Disponibilidad	Adquisición	Tipo de DT que aplica	Actividad de gestión que apoya	Causa que mitiga
KB Doctor	Extensión	Desde XEvo1 hasta 16	OpenSource	Código, Arquitectura	Identificación, Medición, Pago	Codificación
LsiExtensions	Extensión	Desde XEvo2 hasta 16	OpenSource	Arquitectura	Identificacion	Codificación
GXServer	WebApp	Desde X Evo3	Paga, Standalone	Infraestructura	Prevención Comunicación	Infraestructura
Deployment Unit	Objeto	Desde v15u10	Integrado	Build	Prevención	Infraestructura
GXtest	Extensión	Desde v15u7	Paga, Integrado	Testing	Prevención	Codificación

Tabla 1 - Componentes para gestión de DT

Identificación de la deuda técnica

El primer paso para gestionar la deuda técnica es poder identificarla. Actualmente hay algunas herramientas que permiten hacer esto en las bases de conocimiento. Las herramientas que se presentan aquí se recabaron mediante una búsqueda ad-hoc en internet. En su mayoría se instalan en forma de extensión en GeneXus agregando funcionalidades y artefactos al trabajar con los objetos de la base de conocimiento.

<u>KBDoctor</u>

Quizás una de las más conocidas sea KBDoctor [7], la cual permite detectar ciertas malas prácticas en los objetos de forma automática. Esta herramienta también puede ejecutarse de forma automática en un pipeline que corra luego de cada commit de los desarrolladores de forma desatendida.

LSIExtensions

LSI.Extensions es un conjunto de utilidades en forma de extensiones disponible para detectar y ayudar en la detección y corrección de malas prácticas o simplemente hacer refactors de forma más rápida [8].

Ambas herramientas son de código abierto y están disponibles para varias versiones de GeneXus. El tipo de deuda técnica de los vistos anteriormente sobre los cuales se enfoca es la de código y arquitectura, ésta última en menor medida.

GXtest

Respecto a la deuda técnica de testing, la herramienta GXtest [9] es la distribuida automáticamente dentro del IDE de GeneXus desde la versión 16. A diferencia de las herramientas anteriores, esta herramienta es propietaria y funciona mediante licenciamiento en GXServer. Permite la creación de pruebas unitarias, de integración e interfaz para ser integradas en un servidor de integración continua.

GeneXus Server

Es una aplicación web comercializada por GeneXus para la gestión de repositorio de código centralizado. Se adquiere mediante el pago de una suscripción mensual de acuerdo a distintos planes disponibles. Como gestor de código permite la integración de cambios de distintos desarrolladores de forma centralizada, manteniendo ramas de desarrollo e historial de cambios.

Objeto Deployment Unit

Este objeto permite agrupar objetos de la base de conocimiento para ser publicados juntos [10]. Contar con el uso de este objeto permite prevenir de mejor manera la generación de deuda técnica categorizada como de build, ya que se agrupan y seleccionan específicamente cuales son los artefactos a deployar.

Pestaña documentación de objetos

Los objetos de las bases de conocimiento tienen una pestaña *Documentation* [11] que permite agregar cualquier tipo de información al estilo wiki sobre el mismo objeto. Esto es útil para que se anote información interna valiosa sobre la funcionalidad que implementa el objeto u otras consideraciones importantes para el mantenimiento y uso del mismo.

Security Scanner

Para buscar posibles vulnerabilidades de seguridad se encuentra la extensión Security Scanner [12], que analiza la base de conocimiento tomando como referencia el top 10 de los riesgos de seguridad de la OWASP, una organización internacional que trabaja para mejorar la seguridad del software en general.

Otro aspecto importante al identificar deuda técnica de codificación es verificar que el código existente siga buenas prácticas de programación. En este sentido se puede validar a modo de lista de verificación si un objeto cumple con los requerimientos acordados en cuanto a la

calidad del código. En la sección de Prevención se enumeran algunas reglas y sugerencias que han sido recomendadas en la comunidad GeneXus.

Comunicación

Más allá de las actividades típicas de realizar reuniones en donde se trabaje en torno a la DT, es importante que haya herramientas y mecanismos integrados en el proceso de desarrollo que apoyen dichas actividades. En este sentido, a partir de la versión 17 de GeneXus se cuenta con la extensión Event Dispatcher [13], que permite integrarse con herramientas utilizadas popularmente como Jira para el manejo de incidentes y Slack para comunicación entre equipos, por ejemplo.
Event Dispatcher es una extensión de GXServer desarrollada por GeneXus, por lo que requiere contar con una licencia de GXServer.

Medición y monitoreo

Para esta etapa, las herramientas KBDoctor, LSIExtensions y GXtest permiten obtener distintas métricas como por ejemplo detectar objetos no usados, casos no controlados por el código y cobertura de las pruebas ejecutadas, entre otras. Según la arquitectura del proyecto puede variar qué métricas aportan valor - hay varias que directamente no aplican en algunos proyectos - pero hay algunas que van a agregar valor desde el aspecto calidad de código en todos los proyectos.
En el anexo de este documento se presentan algunos de los indicadores que estas herramientas permiten obtener.

Prevención

Una técnica de prevención es la revisión entre pares. Consiste en que un miembro del equipo revise los cambios introducidos por otro miembro con el objetivo de evaluar la calidad del código de este último. De esta forma se pueden detectar code smells o cuestionar la forma de resolver algo en particular. Si bien con GeneXus Server no es posible realizar *pull requests* o *merge requests* como en otros gestores basados en git, se pueden realizar mecanismos similares mediante el versionado de la base de conocimiento o exportando los cambios pendientes y enviando a otro desarrollador para que haga la revisión.

Otro aspecto fundamental para la prevención es tener un código legible, mantenible y extensible, reduciendo así la acumulación de deuda técnica por no entender un código ya existente. En este sentido herramientas como KBDoctor permiten reflejar acuerdos de codificación que se tomen en un equipo de trabajo o empresa en su conjunto. En el Anexo se presentan varias métricas de ésta y otras herramientas.

Buenas prácticas

Si bien hay reglas de programación que pueden depender del equipo y son más bien acuerdos entre los miembros del mismo, hay otras que es recomendable seguirlas ya que de lo contrario dificultan el correcto mantenimiento y evolución de un sistema.

A continuación se enumeran algunas de estas reglas y sugerencias.

- Las KBs deben estar bien modularizadas. Los módulos deben ser especializados, por ejemplo un módulo para funciones, backend, frontend, lógica de negocio, etc.
- Los nombres de objetos y variables deben indicar claramente su propósito, de manera que facilite el entendimiento del código implementado.
- Los atributos deben tener una descripción y help que den información más precisa sobre lo que representan.
- Las Tablas deben tener nombres que representen la realidad y no el nombre heredado por la transacción que las crea.
- Las variables que hagan referencia a un atributo deben ser basadas en el mismo y tener el mismo nombre del atributo, agregando uno o más sufijos para calificarla en el caso que sea necesario.
- Agrupar las reglas de los objetos por atributo o comportamiento
- Usar indentación apropiada
- Usar dominios enumerados en vez de constantes
- Los atributos deben estar basados en dominios para responder a cambios de largo fácilmente
- Usar sentencia *do-case* en vez de *if else if* anidados.
- Utilizar subrutinas cuando corresponda
- Utilizar SDTs en lugar de multiples parametros
- Ser cuidadosos en el uso de strings, usar *format* para traducir y si no se quiere traducir usar !”texto”
- No meter lógica innecesariamente en objetos de interfaz, ya que de esta forma no pueden ser sometidos a pruebas unitarias
- Usar los objetos de forma adecuada

Pago

Los reportes que generan las herramientas son un insumo para esta etapa, ya que en base a las métricas recogidas permiten analizar de forma más objetiva en cuáles objetos hay problemas identificados de forma de poder priorizarlos para ser atacados por el equipo de desarrollo. En general, lo que se hace es reescribir el código, eliminar o modificar dependencias o rediseñar parcial o completamente un objeto, entre otras acciones.

El objetivo de todas las etapas de gestión es trabajar en pos de reducir la cantidad de deuda técnica que se tiene acumulada, llevándola a la metáfora que se utiliza, se corresponde al pago de la deuda. Es importante notar que por más que se trabaje en la prevención de la misma, es imposible evitarla totalmente, dado que el mismo paso del tiempo puede ocasionar la aparición de ítems asociados a deuda técnica.

Hay varias técnicas para el pago de ítems de deuda técnica. En general consisten en hacer explícito el pago mediante tarjetas en el backlog del equipo y dedicarle un porcentaje del total del tiempo disponible para nuevas funcionalidades y corrección de errores al pago de DT.

Hay que tener en cuenta que no es posible librarse completamente de la deuda técnica pero hacer una gestión apropiada de la misma no permite que esta se convierta en un impedimento para mantener y evolucionar un sistema a través del tiempo, tecnologías y personas.

Referencias

[1] The WyCash Portfolio Management System: https://c2.com/doc/oopsla92.html
[2] Propuesta de doctorado en desarrollo:
https://figshare.com/articles/thesis/DOCTORAL_PROPOSAL_A_FRAMEWORK_BASED_ON_SOFTWARE_PROJECT_CONTEXT_FOR_SUPPORTING_THE_FORECASTING_AND_MANAGEMENT_OF_TECHNICAL_DEBTS/11982249
[3] Artículo *Technologies to Support The Technical Debt Management In Software Projects: A Qualitative Research*:
https://www.cos.ufrj.br/index.php/pt-BR/publicacoes-pesquisa/details/15/2857
[4] GeneXus, preguntas frecuentes: https://www.genexus.com/es/productos/genexus/faq
[5] GeneXus Marketplace: https://marketplace.genexus.com/home.aspx
[6] GeneXus wiki: https://wiki.genexus.com
[7] Repositorio de KBDoctor: https://github.com/enriquealmeida/KBDoctor
[8] Documentación de LSI Extensions: http://lsigxextensions.sourceforge.net/
[9] Página del producto GXtest: https://www.genexus.com/es/productos/gxtest
[10] Objeto Deployment Unit:
https://wiki.genexus.com/commwiki/servlet/wiki?38886,Category%3ADeployment+Unit+object
[11] Pestaña Documentación:
https://wiki.genexus.com/commwiki/servlet/wiki?6685,Object+Documentation
[12] Security Scanner: https://marketplace.genexus.com/product.aspx?securityscanner,en
[13] Event Dispatcher:
https://wiki.genexus.com/commwiki/servlet/wiki?46160,Event+Dispatcher+extension
[14] Extensión para chequeo de objetos no referenciados:
http://lsigxextensions.sourceforge.net/noreferenciados.shtml
[15] Extensión de verificación de objetos:
http://lsigxextensions.sourceforge.net/verificacion.shtml
[16] Métrica de Cobertura de código de las pruebas:
https://wiki.genexus.com/commwiki/servlet/wiki?44954,Test+Coverage

Anexo 1

Herramientas para la gestión de la deuda técnica con GeneXus

En este anexo se presentan a modo de resumen algunas de las métricas de las herramientas presentadas para la gestión de la deuda técnica en GeneXus. Esta lista no pretende ser extensiva en cuanto a métricas ni tampoco en cuanto a herramientas.
Las herramientas de las cuales se presentan métricas completas son KBDoctor, LSI Extensions y GXtest.

KBDoctor

Se pueden configurar distintos valores para algunas métricas a recoger durante un análisis. Se destacan algunas de ellas, agrupando por el tipo de DT - de código o de arquitectura - al cual da soporte.

Soporte para DT de código

CleanUnusedVariables: true | false. Indica si se deben eliminar las variables no utilizadas en los objetos analizados.

ParamINOUT = true | false. Valida si todos los parámetros tienen definido explícitamente si son de entrada, de salida o entrada y salida.

CodeCommented = true | false. El código que se encuentre comentado es marcado como error o no.

VariablesBasedAttOrDomain, AttributeBasedOnDomain, SDTBasedAttOrDomain = true | false
Indica si se valida que variables. Atributos y SDTs respectivamente están basados en atributos o dominios.

EmptyConditionalBlocks, ForEachsWithoutWhenNone, NewsWithoutWhenDuplicate = true | false
Útiles para chequear si posibles casos no implementados que pueden resultar en problemas de situaciones no controladas en la lógica del sistema.

Para finalizar, hay otras validaciones numéricas que pueden hacerse respecto a la calidad del código, como máximo anidamiento permitido, máximo nivel de complejidad permitido, máximo tamaño de bloque y máxima cantidad permitida de parámetros en los objetos. Las propiedades respectivas son **MaxNestLevel, MaxComplexity, MaxBlockSize y MaxParameterCount** que apuntan a tener un código más legible, mantenible y extensible, reduciendo así la acumulación de deuda técnica.

Soporte para DT de arquitectura

CheckModule = true | false. Todos los objetos deben estar dentro de algún módulo.

List Tables es otra funcionalidad que permite analizar la estructura de las tablas, obteniendo métricas de la cantidad de claves ya que tener muchas claves puede ser problemático, largo de las claves y si tienen largo variable por ejemplo que no es una buena.

Además de estos hay otros reportes que permiten obtener información que pueden ser problemáticos a nivel de arquitectura, como tener objetos que referencian a tablas definidas en otro módulo.

LSIExtensions

Se identificaron dos extensiones en particular que trabajan sobre calidad de código.

Buscar objetos no referenciados [14]

- Verificar objetos no utilizados
- Permite buscar objetos y atributos no utilizados.

Verificación de objetos [15]

- Validaciones varias
- Permite hacer una búsqueda de errores lógicos en un objeto:
    - variables que no se utilizan
    - que sólo se leen o escriben
    - parámetros no utilizados

También incluye una función para arreglar algunos de estos errores.

GXtest - cobertura de tests

Además de poder contar con tests, que dan mayor tranquilidad al desarrollador a la hora de hacer cambios en objetos, se puede obtener la métrica **Test Coverage** [16] que indica el grado de cubrimiento que tiene un test sobre los objetos a los que llama directamente.
Una métrica que se debería agregar y se cuenta en otras herramientas es la **cobertura total de los tests** sobre los objetos del proyecto.