Deuda técnica

¿Qué es?

La deuda técnica es una metáfora utilizada para identificar una forma de hacer algo de forma provisional en el corto plazo para resolver un problema pero que debe ser pagada con intereses en el largo plazo, asumiendo que eso provisional no fue realizado de manera óptima.

Quizás es algo de como lo estoy interpretando, pero por las dudas vale aclarar que no siempre las decisiones se toman como algo provisional. Por como yo lo veo, algo se considera provisional como al dejar un comentario del tipo // FIX ME o // ToDo, son acciones del tipo intencional. Es decir cuando uno es consciente de la acción o decisión que se está tomando. También se deben tener en cuenta las acciones inintencionales que pueden provocar deuda técnica (por ejemplo por desconocimiento).

En el ámbito del desarrollo de software está relacionado con la calidad **interna** de un sistema de software, ya que no se hace visible al usuario final. Para ser un problema considerado deuda técnica debe proporcionar un **beneficio en el corto plazo** (ahorro de tiempo/costo) pero al largo plazo **pone en riesgo la mantenibilidad y/o evolución** del sistema.

Están relacionados con la deuda técnica:

- Código de mala calidad, que viola las reglas de codificación
- Atajos tomados durante el diseño
- Defectos conocidos, cuya eliminación es pospuesta
- Problemas de arquitectura, como violación de la modularidad
- Code smells
- Aspectos de baja calidad interna generales, que afectan la mantenibilidad y evolución

NO están relacionados con deuda técnica

- Defectos
- Problemas triviales de calidad de código, que no violan las reglas de codificación
- Falta de procesos de soporte
- Funcionalidades no implementadas

  Qué tipo de funcionalidades no implementadas?
  funcionalidades que aún no fueron implementadas y están dentro del plan a futuro?
  funcionalidades que deberían de estarlo y a cambio se tiene un stub?

¿Dónde se encuentra?

Se encuentra en el código, los objetos y componentes de un sistema; artefactos en general relacionados a un sistema de software.

Según el metamodelo y manejando la deuda técnica como una metáfora, está en más lados además de los artefactos.

Como se dijo antes, **no es visible al usuario final** del sistema.

¿Cómo se manifiesta?

Problemas para mantener y evolucionar un producto de software.

Por ejemplo cuando se quiere agregar una funcionalidad y se torna una actividad cada vez más compleja porque requiere cambios grandes en la arquitectura del sistema que podrían haberse evitado habiendo hecho un mejor diseño y evolución del sistema.

Esto es lo mismo que "no realizar tareas de refactor en el código cada cierto tiempo" ? o es aparte?

También puede manifestarse cuando un desarrollador debe arreglar un defecto y quiere modificar el código realizado por otro desarrollador pero no logra comprender cómo funciona el código para poder hacer el arreglo o le lleva mucho más tiempo del que podría llevarle si el código fuera de buena calidad.

¿Cómo la perciben los desarrolladores?

Dificultad para entender código ya existente para agregar una nueva funcionalidad o modificar algo existente. Mala calidad de código, no sigue buenas prácticas.

Ejemplos de deuda técnica en GeneXus

Del artículo Buenas prácticas de programación en GX se puede decir que no seguir las reglas que se mencionan o estar del lado opuesto a las recomendaciones realizadas por la comunidad son ejemplos de deuda técnica en GeneXus.

Reglas

- Usar método GIK para nomenclatura de atributos
- Los atributos deben tener una descripción y help
- 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, si la lógica lo permite
- Las variables que hagan referencia a un atributo deben ser basadas en el mismo y tener el mismo nombre del Atributo, agregando uno más sufijos para calificarla en el caso que sea necesario

Sugerencias

- Se recomienda que al definir subtipos estos pertenezcan a un grupo específico y no sean definidos en el grupo "None". Cada grupos de subtipos debería tomar el nombre del Atributo Subtipo que identifica al Grupo (primario), o una concatenación de nombres si hay varios primarios
- Agrupar las reglas por atributo o comportamiento
- Usar indentacion
- En los foreach no usar and sino que agregar un where en cada linea (hay muchos comentarios de lo hago de esta forma o de otra, bastante opinable)

  [mi aporte] - creo que depende basatante del caso

  antes que nada considerar si se puede usar un índice o crearlo de ser posible

  También se debería revisar la mejor forma de que se haga la consulta en la BD y no del lado del cliente.
- Los atributos deben estar basados en dominios para responder a cambios de largo fácilmente
- Usar dominios enumerados en vez de constantes
- Tomarse un tiempo para ver objetos que ya no se utilicen para reducir tiempos de operaciones como build all, respaldo, update, etc.

- *[mi aporte] - las variables tienen que tener un nombre que la identifique, sin importar el largo - &a, &aux están mal*

En http://icgenexus.blogspot.com/2017/10/buenas-practicas-de-genexus.html se hace mucho énfasis en claridad de código, uso apropiado de espacios, nomenclatura, indentación, etc.

Usar sentencia do-case en vez de if else if anidados.

Se deben utilizar subrutinas cuando corresponda.

Utilizar SDT en lugar de múltiples parámetros.

Propone marcar cosas a arreglar (FIXME) o implementar (TODO) como comentario en el código.

Tener cuidado con los strings, usar format para las traducciones, si no se quiere traducir usar !"texto".

En relación a esto último, últimamente me ha pasado que el manejo de traducciones basado en el objeto Spanish no se detecta en todos los posibles textos. Por ejemplo en Captions de Forms o en campos de Data Types usados a traves del format. Como después de tiempo investigando no encontré una referencia que indique cómo solucionarlo o si está reportado como un bug de la plataforma, terminé "cambiando a mano" el texto. Esto implica que se tuvo que hacer un registro del cambio para tener en cuenta en posibles revisiones del codigo o futuros cambios.
Esto es deuda técnica?

En los otros artículos se menciona la revisión de código entre pares (Federico Toledo), y David habla de análisis de código estático como forma de evitar la creación de deuda técnica.
Tener cuidado con los términos utilizados; el análisis estático de código está más asociado a herramientas que lo realizan automatizado. Si se trata de que es una persona la que realiza el análisis se le denomina revisiones.

Mi aporte:

Tener pruebas unitarias para los objetos que sea posible y necesario.
totalmente de acuerdo - inclusive manejarse como regla general en el testing.

… Esto viene de la mano de lo que mencionaba Federico Toledo que se estaba trabajando en ese entonces para mejorar gxunit (hoy **GXtest** incluye pruebas unitarias con versionado dentro de la misma KB).
No tener pruebas automatizadas sobre el código es mencionado como un tipo particular de deuda técnica. En GeneXus no está automatizable aún el tener métricas para la cobertura de código de las pruebas como para definir criterios de calidad y condiciones de éxito / fallo de un pipeline por ejemplo.

En conclusión a lo anterior se podria decir que, en comparación a otros lenguajes, la falta o carencia de herramientas de testing no sea deuda técnica en sí, pero podrian verse como factores potenciadores de esta en el proceso de testeo.

Mala práctica aunque común: meter lógica de negocio en los webpanels. ¿Se puede evitar?
Mala práctica porque no pueden ser testeados con pruebas unitarias el código que se incluye ahí.

Podríamos decir que esto es un code-smell específico de Gx?
Creo que no, es de alguna forma alentado al tener la posibilidad de poner ese código ahí pero es algo que existe en muchos de los frameworks que conozco donde se puede ejecutar lógica a discreción del lado del cliente. Sí se podría mencionar como un ejemplo concreto.

Herramientas

KBDoctor

Es una herramienta para análisis estático de código, distribuida como plugin para GeneXus con posibilidad de ejecutar el análisis vía terminal usando una task de MsBuild. La documentacion esta aqui: https://wiki.genexus.com/commwiki/wiki?5069,KBDoctor,

Es open source y gratuita: https://github.com/enriquealmeida/KBDoctor

Para instalarla hay que conseguir la versión compatible con la versión de GeneXus que se esté usando. Es compatible con Evo 1, Evo 2, Evo 3, 15 y 16.

Métricas soportadas

Las métricas se pueden definir desde el IDE desde el menú Tools -> KBDoctor -> About KBDoctor -> Edit Review Object ini file.

Los controles que se pueden realizar son los siguientes (la gran mayoría son booleanos):

- CleanUnusedVariables: elimina las variables sin utilizar de los objetos
- FixVariables: Arregla las definiciones de variables, asignando un atributo o dominio
- ParamINOUT: chequea si todos los parámetros tienen las etiquetas IN, OUT o INOUT
- CheckCommitOnExit: Chequea si la propiedad Commit on exit = YES
- CheckModule: chequea que todos los objetos pertenezcan a un módulo
- CodeCommented: chequea si hay código comentado para marcarlo como error
- VariablesBasedAttOrDomain: chquea que las variables esten basadas en atributos o dominios
- AttributeBasedOnDomain: chequea que los atributos están basados en dominios
- SDTBasedAttOrDomain: chequea que los ítems de los SDT están basados en atributos o dominios
- AttributeWithoutTable: chequea si todos los atributos pertenecen a una tabla

- AssignTypes: chequea si tienen el tipo o dominio correcto asignado
- ParameterTypes: chequea si tienen el tipo o dominio correcto asignado
- EmptyConditionalBlocks: chequea si hay condiciones sin código dentro
- ConstantsInCode: chequea si hay constantes hardcodeadas
- ForEachsWithoutWhenNone: chequea si hay algun 'ForEach' sin clausula 'When None'
- NewsWithoutWhenDuplicate:
- ProceduresCalledAsFunction: chequea si hay procedures llamados como funciones
- DocumentsInWebPanels: chequea si hay variables de manejo de archivo y algunas otras en webpanels. Esto da un warning si se usan variables ExcelDocument, HttpRequest, WordDocument, etc para no incentivar su uso fuera de procedures
- MaxNestLevel = 7 => máximo nivel de anidamiento del código
- MaxComplexity = 30 => máximo nivel de complejidad del código
- MaxBlockSize = 300 => largo máximo de un bloque de código
- MaxParameterCount = 6 => máxima cantidad de parámetros permitidos en la regla *parm* de los objetos

Como se puede ver hay muchísimas opciones sobre qué controlar usando KBDoctor, el cómo sacarle valor a estos reportes es responsabilidad del lector, pero se sugiere definir las métricas lo antes posible (idealmente antes de que se escriba la primera línea de código).

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.

Estas son MaxNestLevel, MaxComplexity, MaxBlockSize y MaxParameterCount, dado que apuntan a tener un código más legible, mantenible y extensible, reduciendo así la acumulación de deuda técnica.

Ejecución del análisis desde la terminal

Dentro del archivo ``.msbuild`` contenido dentro del zip de **KBDoctor** se encuentran todas las *tasks* que se van a poder ejecutar desde la terminal. Las más relevantes son **ReviewObjects** y **ReviewCommits**.

ReviewObjects

ReviewObjects nos permite analizar los objetos dentro de una **KB**, modificados dentro de un rango de tiempo parametrizable.

Si invocamos la *task* parametrizando ``DateFrom``, **KBDoctor** va a realizar un análisis de todos los objetos modificados desde esa fecha. En cambio, si simplemente invocamos la *task* pero no parametrizamos ``DateFrom``, **KBDoctor** va a analizar únicamente los objetos modificados en los últimos dos días.

Ejemplo de ejecución de task **ReviewObjects** sin uso de fecha:

``` cmd
MSBuild.exe KBDoctor.msbuild /t:ReviewObjects

```

Ejemplo de ejecución de task **ReviewObjects** pasando fecha por parámetro (acepta formato "Día-Mes-Año"):

``` cmd
MSBuild.exe KBDoctor.msbuild /t:ReviewObjects /p:DateFrom="01-08-2019"

```

ReviewCommits

ReviewCommits depende del uso de GXServer en el proyecto, dado que toma información del mismo.

Al igual que a la task ReviewObjects se la puede parametrizar con *DateFrom* y *DateTo*, pero si no se pasan esos parámetros simplemente analiza los commits de los últimos dos días. Lo que sí hay que parametrizar son las credenciales usadas para conectarse con GXServer, con nombres *ServerUser* y *ServerPassword*

Ejemplo de ejecución de task ReviewCommits sin uso de fecha:

MSBuild.exe /t:ReviewCommits /p:ServerUser="usuario";ServerPassword="contraseña"

Con fecha:

MSBuild.exe /t:ReviewCommits /p:DateFrom="01-08-2019";DateTo="07-08-2019";ServerUser="usuario";ServerPassword="contraseña"

Template de archivo msbuild para ReviewObjects

Una vez configurados los análisis que van a ser tenidos en cuenta, hay que configurar el archivo de MsBuild.

Se puede tomar como ejemplo el siguiente:

```
<Project DefaultTargets="ReviewObjects"
 xmlns="http://schemas.microsoft.com/developer/msbuild/2003">

<Import Project="C:\GeneXus16\Genexus.Tasks.targets" />
<Import Project="C:\GeneXus16\Packages\KBDoctorCmd.Tasks.targets"/>

<Target     Name="ReviewObjects">
<OpenKnowledgeBase Directory="$(KBPath)" />
<ReviewObjectsCmd DateFrom="$(DateFrom)" />
<CloseKnowledgeBase />
</Target>

<Target     Name="ReviewCommits">
<OpenKnowledgeBase Directory="$(KBPath)" />
        <ReviewCommitCmd     DateFrom="$(DateFrom)" DateTo="$(DateTo)"
 ServerUser="$(ServerUser)" ServerPassword="$(ServerPassword)"/>
<CloseKnowledgeBase />
</Target>

</Project>
```

En la ejecución por msbuild por ejemplo se puede ver algo así:

```
warning : Commit on EXIT = YES (Procedure 'SendCrearCitaRequest')
warning : Object without module. (Procedure 'SendCrearCitaRequest')
warning : Commented code [/&RNAME =GETPARAMETERVALUE("AG...] (Procedure
'SendCrearCitaRequest', Source)
warning : &citaERR.ERR_3.CWE_2 = &ErrorMsg (&citaERR.ERR_3.CWE_2) Doesn't have domain but
(&ErrorMsg) is BasedOn ErrorMsg (Procedure 'SendCrearCitaRequest' Source, Line: 29)
warning : &citaERR.ERR_3.CWE_2 = &ErrorMsg (&citaERR.ERR_3.CWE_2) Doesn't have domain but
(&ErrorMsg) is BasedOn ErrorMsg (Procedure 'SendCrearCitaRequest' Source, Line: 41)
warning : logError -- Parameter (IN): (&Pgmname) Variables are based on different domains
Value<>(No Domain) (Procedure 'SendCrearCitaRequest' Source, Line: 33)
warning : logError -- Parameter (IN): (&Pgmname) Variables are based on different domains
Value<>(No Domain) (Procedure 'SendCrearCitaRequest' Source, Line: 45)
warning : String Constant in code (Procedure 'SendCrearCitaRequest' Source, Line: 1)
warning : String Constant in code (Procedure 'SendCrearCitaRequest' Source, Line: 1)
warning : String Constant in code (Procedure 'SendCrearCitaRequest' Source, Line: 3)
warning : String Constant in code (Procedure 'SendCrearCitaRequest' Source, Line: 3)
warning : String Constant in code (Procedure 'SendCrearCitaRequest' Source, Line: 4)
warning : String Constant in code (Procedure 'SendCrearCitaRequest' Source, Line: 4)
warning : String Constant in code (Procedure 'SendCrearCitaRequest' Source, Line: 5)
warning : String Constant in code (Procedure 'SendCrearCitaRequest' Source, Line: 5)
warning : getParameterValueProcedure called as a function (Procedure 'SendCrearCitaRequest'
Source, Line: 3)
warning : getParameterValueProcedure called as a function (Procedure 'SendCrearCitaRequest'
Source, Line: 4)
warning : getParameterValueProcedure called as a function (Procedure 'SendCrearCitaRequest'
```

```
Source, Line: 5)
warning : logErrorProcedure called as a function (Procedure 'SendCrearCitaRequest' Source,
Line: 33)
warning : logErrorProcedure called as a function (Procedure 'SendCrearCitaRequest' Source,
Line: 45)
```

Otra herramienta: https://marketplace.genexus.com/product.aspx?lsiextensiones,en no la he investigado aún.

Algo que puede ayudar es una configuración para que se traten los warnings como errores para que los devs los atiendan y que no se ignoren.

Me comentaste: "¿Acumulación de warnings podría ser un tipo de deuda técnica no clasificada por ejemplo?"
No me dio el tiempo para revisarlo en profundidad aun pero pareciera que no está clasificado dentro de ninguna categoría. Creo que se le puede encontrar una en la cual insertarlo y no crear una nueva quizás.
Tendría que revisarlo un poco más, pero me parece que podria clasificarse como DT en el "build". Existen algunos warnings que en el entorno de prototipo no generan error pero al momento de compilar saltan errores.

Para la gestión de la deuda técnica no veo nada que sea específico de GeneXus ya que son actividades generales. Sí hay diferencias en las herramientas para medir la deuda técnica técnica pero no en las de seguimiento por ejemplo.

En cuanto a los tipos de deuda técnica veo que todos aplican en lo conceptual pero nuevamente se manifiestan de forma distinta en los artefactos. Voy a comentar los que entiendo son más diferentes en GeneXus.

Arquitectura: no tengo experiencia en la definición de reglas complejas para transacciones pero creo que puede llegar a plantear dificultad a la hora de implementar restricciones complejas en la base de datos real generada.

Código: en general no difiere de otras herramientas ya que se codifica también de manera libre. Lo particular que creo es alentado por GeneXus es que permite meter lógica de negocio dentro de un objeto de interfaz como lo son los paneles.

Documentación: en GeneXus los objetos tienen una sección específica para documentación, que si no es utilizada correctamente puede considerarse deuda técnica. Es cierto que la documentación puede ser construida por fuera de la KB también. All the documentation sources inside the Knowledge Base make up a wiki-like documentation system. Application documentation is an important part of your Knowledge Base. Development documents range from application requirements to developer-to-developer notes, to-do lists, etc.

Servicios: no entendí bien a qué se refiere este punto, si es conexión a otros servicios o la definición de los mismos.

Testing: creo que en general en GeneXus no hay un uso muy aceptado de pruebas automatizadas, es algo que existe hace relativamente poco y para versiones más nuevas .

Versionado: entiendo hay situaciones que no es práctico el uso del versionador de código de GeneXus. Por ejemplo a la hora de code review Yonathan mencionaba que no es posible hacer un merge request para que alguien revise los cambios antes de ser integrados.

Como dije el resto que no nombré acá creo que aplican pero no hay ninguna consideración especial que se deba tener por trabajar en GeneXus.

En cuanto al proceso de desarrollo y la gestión de la deuda técnica macro creo son cosas que trascienden a la tecnología.

Del meta modelo no me dió para dedicarle mucho pero creo no hay nada especialmente destacable por el hecho de usarse GeneXus. Las actividades y cómo se gestionan los ítems de deuda técnica se hacen con otras herramientas como Trello, JIRA, etc y no dependen de la herramienta de desarrollo, si bien algunas pueden tener integraciones con estas herramientas para un mejor seguimiento de issues asociados a deuda técnica.

La modularización es importante. Las KB deben estar bien modularizadas.
Los módulos deben ser especializados (funciones, backend, frontend, lógica de negocio, etc)

Preguntas interesantes / pendientes:
Tiene muchas tablas u objetos públicos?
Objetos grandes?
Muchos warnings?
Mucho comentario?
Objetos no alcanzables?
¿Variables no referenciadas?
Codigo "viejo" (ej defined by, noread, etc)
Claves de tablas muy largas.
Falta de subtipos.