FACULTAD DE INGENIERÍA
UNIVERSIDAD DE LA REPÚBLICA
URUGUAY

INFORME PROYECTO DE GRADO

Juan Pablo Goyení
Marcos Olivera
Nicolás Carro

Tutora de Proyecto
Mónica Wodzislawski

Usuarios Responsables
Matías Reina
Federico Toledo

Resumen

En la actualidad la verificación de software se ha convertido en una de las etapas más críticas del ciclo de vida del desarrollo de software y es parte esencial de cualquier proceso de desarrollo, debiendo realizarse con la mayor eficacia y eficiencia, para detectar el mayor número de problemas y disminuir su costo. Esto se debe a la gran importancia que han adquirido los Sistemas Informáticos y a la vertiginosa expansión hacia todos los horizontes, lo que hace a la calidad un requerimiento indispensable e imprescindible.

Un gran porcentaje del software creado en Uruguay es desarrollado en GeneXus, una herramienta de especificación de sistemas de información basada en conocimiento, que modela la realidad a través de un conjunto de instancias de objetos-tipo y las almacena en una base de conocimiento. En el área de pruebas, GeneXus cuenta con GXtest, una herramienta para automatización de pruebas funcionales, y con GXUnit, una herramienta que da soporte a las pruebas unitarias, la cual fue desarrollada en el presente proyecto aunque ya contaba con antecedentes.

En este documento se analiza el estado del arte referente a pruebas de software, GeneXus, otros lenguajes de similares características, y metodologías de pruebas con el fin de definir los requerimientos necesarios para la construcción de una nueva versión de GXUnit. También se realizaron presentaciones del estado del arte del proyecto a distintos actores interesados en las cuales se obtuvieron nuevos requerimientos y una priorización de los mismos. Los grupos GeneXus del curso Proyecto de Ingeniería de Software del Instituto de Computación (InCo), Facultad de Ingeniería (Fing), Universidad de la República (UdelaR) utilizaron la herramienta para la construcción y ejecución de pruebas unitarias, compartieron sus experiencias y aportaron nuevos requerimientos para futuras versiones de GXUnit.

Se presenta la arquitectura, modelo de diseño e implementación de la nueva versión de GXUnit con ejemplos prácticos de utilización de la herramienta. También un caso de estudio referente a la utilización de GXUnit en el curso Proyecto de Ingeniería de Software.

Palabras Claves: Automatización, GeneXus, GXUnit, Prueba unitaria, Pruebas de software, Unidad de software

Agradecimientos

A Mónica Wodzislawski por sus valiosos aportes en todas las etapas de desarrollo del presente proyecto.

A Matías Reina y Federico Toledo por el apoyo y constante compromiso.

Al Centro de Ensayo de Software y a Abstracta por prestarnos sus instalaciones para realizar las reuniones de seguimiento del proyecto y presentaciones del mismo.

A Luciano Silveira y Federico Azzato de Artech que nos aportaron mucho con su conocimiento en Extensiones de GeneXus.

A Martín Olivieri de Artech que nos ayudó con la publicación de GXUnit en GeneXus MarketPlace.

A Gustavo Carriquiry, Ursula Bartram, Alejandro Araujo y Enrique Almeida que nos aportaron con la definición y priorización de requerimientos.

Integrantes de los 2 grupos GXUnit del Proyecto de Ingeniería de Software 2007 dado que el proyecto realizado por ellos nos sirvió como punto de partida.

Integrantes de los grupos 02 y 03 del Proyecto de Ingeniería de Software 2011 que utilizaron la herramienta y compartieron sus experiencias con GXUnit.

A la familia y amigos por el apoyo incondicional.

Contenido

1. Introducción

Hoy en día el software ha adquirido un rol relevante en todos los ámbitos, está presente no solo en las computadoras sino que en una amplia gama de dispositivos. La verificación de software es parte esencial de cualquier proceso de desarrollo, debido a la importancia que han adquirido los sistemas informáticos y la calidad que requieren.

La introducción de pruebas en etapas tempranas aporta varios beneficios facilitando la construcción de software de mayor calidad. Detecta defectos en los métodos durante la etapa de desarrollo, reduciendo los costos del proyecto ya que si son detectados en etapas finales demandan un mayor esfuerzo de análisis y corrección. Además facilita el mantenimiento correctivo ya que la ejecución del conjunto de pruebas unitarias disminuye la probabilidad de que un cambio en una o más partes del código afecte a otras funciones.

Una de las tareas clave es que los desarrolladores construyan y ejecuten pruebas unitarias, que si bien son muy útiles en todas las metodologías de desarrollo constituyen uno de los pilares fundamentales de las metodologías ágiles y de integración continua como *Testing Driven Development* (TDD) que se basa en construir primero las pruebas y luego el objeto a probar.

Con el objetivo de poder explotar estas ventajas desde GeneXus [1], se han lanzado varias propuestas a la comunidad de usuarios, ya que la herramienta no presenta facilidades nativas para llevar a cabo las pruebas en unidades de software y actualmente no existen productos en la industria para crear, mantener y ejecutar pruebas unitarias en un ambiente integrado al de GeneXus. A nuestro juicio, la respuesta más trascendente es el proyecto "GXUnit"[2-4], en el cual se basa este trabajo.

1.1. *Antecedentes de GXUnit*

La propuesta GXUnit tuvo su origen en el año 2003[5] cuando se anuncia el objetivo de crear una herramienta de automatización de pruebas unitarias en GeneXus. En el año 2004 se formalizó la propuesta[6] en la línea de las herramientas xUnit. Los objetivos básicos planteados fueron los siguientes: crear un marco de pruebas asociado a GeneXus, poder escribir las pruebas en GeneXus, ejecutar las pruebas y registrar los resultados.

En el año 2006 se retoma bajo la forma de "Proyecto Colaborativo GeneXus"[2, 7] y se propone el proyecto para su desarrollo en Julio de 2007, en el ámbito del curso "Proyecto de Ingeniería de Software 2007" del InCo, Facultad de Ingeniería, Universidad de la República del Uruguay (UdelaR). Entre agosto y noviembre de dicho año dos grupos de estudiantes desarrollaron en forma independiente y en paralelo, dos versiones primarias de la herramienta GXUnit [8]. Los productos de software resultantes de dichos desarrollos han sido liberados al dominio público y pueden ser descargados desde Internet[3, 4].

Luego de esta experiencia, se publica un documento[9], donde se explica en detalle el trabajo realizado en estas dos versiones, se obtienen conclusiones del producto y se sugiere trabajo a futuro sobre el tema. Además, se confecciona una Tesis de Maestría[10] en Ingeniería en Computación donde se propone una herramienta para automatizar la creación, mantenimiento y ejecución de pruebas unitarias asociado a GeneXus basado en funcionalidades adaptadas desde FIT[11].

1.2. *Motivación*

En nuestra experiencia como desarrolladores, el sufrir presiones por entregar el producto en tiempo conlleva generalmente a no entregarlo en forma, lo cual, nos hace entrar en un ciclo de más presión, menos pruebas y más errores. Estos errores terminan siendo identificados en etapas tardías del proyecto, lo cual aumenta exponencialmente el esfuerzo de corrección y por ende el costo, el cual según Tassey[12] puede elevarse hasta un 75% del total del proyecto. Por lo tanto, es fundamental la identificación temprana de errores mediante la ejecución de pruebas unitarias. Sin embargo, la construcción de un ambiente controlado para la generación y ejecución de pruebas aún no había sido cristalizada en GeneXus. Ejecutar pruebas manualmente (sin herramientas auxiliares) demanda tiempo y esfuerzo porque es necesario crear y mantener los programas que ejecutan las pruebas, salvar los resultados de las ejecuciones y mantener ambientes aislados para no experimentar comportamientos inesperados. Como consecuencia, usualmente se subestima la etapa de pruebas unitarias, perdiendo los beneficios mencionados**.**

En este sentido es oportuno recordar la frase atribuida a Boris Beizer: "*las fallas más notorias en la historia del desarrollo del software fueron todas debidas a defectos en las unidades, defectos que podrían haber sido encontrados con apropiadas pruebas unitarias*"[13].

Dado nuestra vinculación actual con GeneXus y su gran difusión en el país resultó de sumo interés la propuesta de abordar el área de automatización de pruebas unitarias y construir una herramienta integrada a su ambiente de desarrollo.

1.3. *Objetivos*

En nuestra experiencia como programadores GeneXus, hemos comprobado que las pruebas unitarias requieren mucho tiempo y esfuerzo por parte del desarrollador. Por lo tanto, pensamos que sería de mucha utilidad que GeneXus de soporte a la verificación de los objetos de una forma automatizada y sin recurrir a herramientas externas al ambiente de desarrollo.

El principal objetivo del proyecto es desarrollar una herramienta que de soporte a pruebas unitarias en GeneXus, basándonos en una investigación sobre cómo se realizan pruebas unitarias en distintos lenguajes de programación. Es deseable que la herramienta contemple:

- Creación y mantenimiento automatizado de pruebas unitarias.
- Integración con el ambiente de desarrollo GeneXus.
- Ejecución de los casos de pruebas.
- Registro de resultados.
- Simulación de objetos GeneXus (*Mocks*).
- Simulación de consultas a la base de datos.

Además, se elaborarán los documentos de análisis y diseño necesarios para poder continuar su desarrollo.

Los cuatro primeros puntos son considerados requerimientos básicos (primarios) para la definición del alcance del proyecto. Los puntos siguientes son considerados requerimientos secundarios y son analizados en el estado del arte para poder incluirlos en futuras versiones de GXUnit.

1.4. *Resultados*

En el presente proyecto se definen los conceptos generales de pruebas de software, pruebas de sistema, pruebas de integración y pruebas unitarias. Se realizó un estudio de las distintas herramientas que dan soporte a las pruebas unitarias para varios lenguajes de programación actuales haciendo foco en lenguajes de cuarta generación y específicamente en GeneXus.

Se analizó en profundidad GeneXus y otros lenguajes de cuarta generación con el objetivo de conocer el contexto de trabajo e identificar requerimientos para la construcción de GXUnit, una herramienta que brinda soporte a pruebas unitarias en GeneXus.

Se definió una arquitectura en capas para la construcción de GXUnit, de esta forma se logró desacoplar la lógica de la herramienta con la comunicación e interacción con las funcionalidades que brinda el Software Development Kit(SDK)[14] de GeneXus minimizando el esfuerzo de mantenimiento tras modificaciones del SDK.

Tras el análisis de distintas bases de conocimiento y el aporte de distintos actores en la definición de los requerimientos se eligieron tres tipos de objetos GeneXus para realizar pruebas unitarias, los objetos *Procedure*, *Transaction* y *Data Provider*.

Para la construcción de las pruebas se definió un nuevo tipo de objeto, el objeto *Test Case* basado en un procedimiento. Se eligió el objeto procedimiento para definir las pruebas debido a la flexibilidad que presentan, el programador puede utilizar el lenguaje de los procedimientos para construir pruebas más ricas y completas. El hecho de que sea un procedimiento brinda la posibilidad de guardar los casos de prueba en la base de conocimiento y poder importar y exportar las pruebas entre distintas bases de conocimiento.

Se definió un nuevo tipo de objetos para la visualización de los resultados, el objeto de tipo *Result*. El objeto de tipo *Result* es un árbol con toda la información de la ejecución de los casos de prueba, qué casos se ejecutaron, qué tiempo tardaron cada uno y los resultados de los *asserts*. Los objetos de tipo *Result* se pueden importar y exportar entre distintas bases de conocimiento, junto con el objeto a probar y sus casos de prueba.

Se introdujo el concepto de suite de pruebas dando la posibilidad de agrupar los casos de prueba según un criterio predefinido para poder ejecutarlos como un conjunto. Esta funcionalidad es esencial para las pruebas de regresión, ante cualquier cambio en un objeto el programador puede ejecutar todos los casos de prueba asociados al objeto y verificar que no haya introducido un nuevo error.

Se logró integrar la herramienta al entorno de desarrollo de GeneXus, un requerimiento imprescindible, desarrollando la misma como una extensión en GeneXus, lo cual no era posible antes de la versión X de GeneXus.

Con la integración de la herramienta al entorno de desarrollo de GeneXus resulta factible la utilización de TDD como metodología de desarrollo, ya que el programador puede crear las pruebas antes de los objetos a probar.

Se publicó la herramienta en el MarketPlace de GeneXus[15], del cual se descargó más de 200 veces. Fue utilizada por los estudiantes de los grupos que desarrollan en GeneXus del Proyecto de Ingeniería de Software 2011 de la Facultad de Ingeniería de la UdelaR. A propuesta de docentes, clientes y a raíz de la utilización creciente de la herramienta, fue presentada en el Evento Internacional GeneXus 2011 con una charla enfocada en los beneficios que brinda. Como consecuencia, se incrementó significativamente la utilización de GXUnit y el soporte a los usuarios.

Se presenta un caso de estudio de utilización de la herramienta en un proceso de desarrollo de un sistema real, en el ámbito del curso Proyecto de Ingeniería de Software de la Facultad de Ingeniería de la UdelaR.

2. Conceptos generales

A continuación se presentan los conceptos más importantes para el contexto del proyecto, extraídos esencialmente desde el SWEBOK[16], aunque se complementan con otras definiciones.

2.1. *Pruebas de software*

"*Las pruebas de software se realizan para evaluar la calidad de un producto, y para mejorarlo, al encontrar defectos y problemas. Consiste en la verificación dinámica del comportamiento de un programa con un conjunto finito de casos de prueba contra el comportamiento esperado*" (SWEBOK). Esta definición se complementa con la definición de Kaner: "*Las pruebas son una investigación técnica orientada a proporcionar información sobre la calidad de un producto de software para un actor o usuario"[17]*

Niveles de las pruebas

Las pruebas son realizadas a distintos niveles en el proceso de desarrollo y de mantenimiento. Por lo tanto, el blanco de éstas puede variar, puede ser un único objeto, un grupo de objetos (relacionados por propósito, uso, comportamiento, o estructura), o un sistema completo. Se pueden distinguir entre tres grandes etapas llamadas pruebas unitarias, pruebas de integración y pruebas de sistema[16].

Pruebas Unitarias

Las pruebas unitarias verifican la funcionalidad, de forma aislada, de partes o unidades de software que pueden probarse por sí solas. Dependiendo del contexto, éstas pueden ser los subprogramas individuales o una componente mayor hecha de unidades estrechamente relacionadas.

Pueden ser ejecutadas sin tener desarrollados todos los objetos involucrados. Para ellos se recurre al uso de Drivers para simular las llamadas al objeto bajo prueba, y de *Stubs* y *Mocks*, para simular los objetos llamados por el objeto bajo prueba.

Generalmente, las pruebas unitarias se realizan con conocimiento del código de la unidad de software que se desea probar, con el soporte de herramientas para depurar e involucra a los programadores que escribieron el código.

Pruebas de Integración

Las pruebas de integración son el proceso de verificar el funcionamiento en conjunto entre componentes de software. Las estrategias de las pruebas de integración clásicas son bottom-up, top-down y big bang. La primera es una estrategia donde los componentes de más bajo nivel se prueban primero y luego son usados para facilitar las pruebas de los de mayor nivel. Este proceso es repetido hasta que los componentes de nivel más alto sean probados. Con la estrategia top-down se procede a la inversa, se prueban primero los componentes de mayor nivel y se van integrando los módulos invocados por éstos, hasta llegar al nivel inferior. En el enfoque big bang, todos o la mayoría de los módulos desarrollados son acoplados entre sí para formar un sistema completo o una parte importante del sistema y luego se procede a realizar las pruebas.

La estrategia se elige dependiendo de la metodología de desarrollo ya que por ejemplo, no

puede practicarse una estrategia top-down, si no se tienen los métodos de más alto nivel para integrarlos.

Pruebas de Sistema

El objetivo de estas pruebas es verificar el comportamiento de todo un sistema. Son apropiadas para las pruebas de requerimientos no funcionales, tales como, seguridad, velocidad, exactitud y fiabilidad; también para las pruebas de requerimientos funcionales que prueben ciclos funcionales completos. También son evaluadas a este nivel las interfaces a otras aplicaciones, así como el ambiente operativo y ciertos dispositivos de hardware.

2.2. *Lenguajes de cuarta generación*

En esta sección se brinda una visión acerca de los Lenguajes de Cuarta Generación (4GL), más específicamente los "Generadores de Aplicaciones", dando una definición y ejemplificando con dos productos concretos, GeneXus y Power Builder[18]. Además de GeneXus se elige Power Builder porque fue premiado por Software Gurú en el año 2010 como el segundo mejor producto de Generación de Aplicaciones (detrás de GeneXus)[19].

2.2.1. ¿Qué es un Lenguaje de Cuarta Generación?

Los lenguajes de cuarta generación son herramientas para optimizar el desarrollo de software automatizando su generación. Se han utilizado principalmente en la generación de código para GUI y en la implementación de programas que facilitan las tareas de los desarrolladores y clientes. El concepto base de los lenguajes 4GL es que el programador especifica los resultados que quiere obtener y la herramienta genera el código para obtenerlos. Existen diferentes tipos de lenguajes de cuarta generación, cada uno con una función en particular, entre ellos están los generadores de reportes, los generadores de formularios de interfaz gráfica, administradores de datos y los generadores de aplicaciones, dentro de este último conjunto está GeneXus [20].

2.2.2. GeneXus

GeneXus es una herramienta de especificación de sistemas de información basada en conocimiento, modela la realidad a través de un conjunto de instancias de objetos-tipo y las almacena en una base de conocimiento. Está orientada principalmente a aplicaciones empresariales, captura la visión de los desarrolladores utilizando un lenguaje mayormente declarativo a partir del cual se genera código para múltiples lenguajes y plataformas utilizando una misma especificación. GeneXus tiene un módulo de normalización, que permite generar y mantener las bases de datos en tercera forma normal. Genera aplicaciones de software basadas en las especificaciones de los usuarios pudiendo mantenerlas, según los constructores de la herramienta, en forma relativamente sencilla ante cambios en los requerimientos, o ante un cambio de la plataforma utilizada.

Objetos GeneXus

GeneXus establece un conjunto de objetos-tipo predefinidos a través de los cuales representa la realidad. Esta sección se basa en los documentos GeneXus: Filosofía[21] y Desarrollo basado en conocimiento[22]. A continuación se describen los objetos-tipos más importantes:

ATRIBUTO

Es el elemento semántico fundamental, tiene un nombre, un conjunto de características y un significado. El "significado" es el segundo elemento básico fundamental y en el modelo de GeneXus se define nombre (atributo) = significado (atributo). Los nombres de los atributos son esenciales, un atributo tiene el mismo nombre en todos los lugares que aparece y no hay dos atributos diferentes (con significado diferente) con el mismo nombre. Todo Atributo forma parte de un objeto Transacción, y éstas, como veremos más adelante, no son más que vistas de los datos que son definidas por el programador GeneXus según sus necesidades.

Los atributos pueden además ser de tipo fórmula, lo que permite definir atributos complejos en una transacción. Por ejemplo, supongamos que tenemos en una transacción dos atributos, PersonaNombre y PersonaApellido, supongamos luego que queremos manejar el nombre completo de la persona, en ese caso sólo sería necesario definir en la misma transacción un

nuevo atributo fórmula cuyo valor fuera el resultado de la concatenación de los atributos PersonaNombre y PersonaApellido. En los atributos de tipo fórmula podrían además ser llamados objetos de tipo Procedimiento, que veremos más adelante.

Otro elemento fundamental son los Subtipos, es la redefinición de atributos preexistentes, de tal forma que adquieran el mismo significado. El atributo, con las consideraciones del significado y los subtipos, son el elemento fundamental de la teoría de GeneXus. Los atributos constituyen el marco de referencia sobre el cual GeneXus edifica sus modelos.

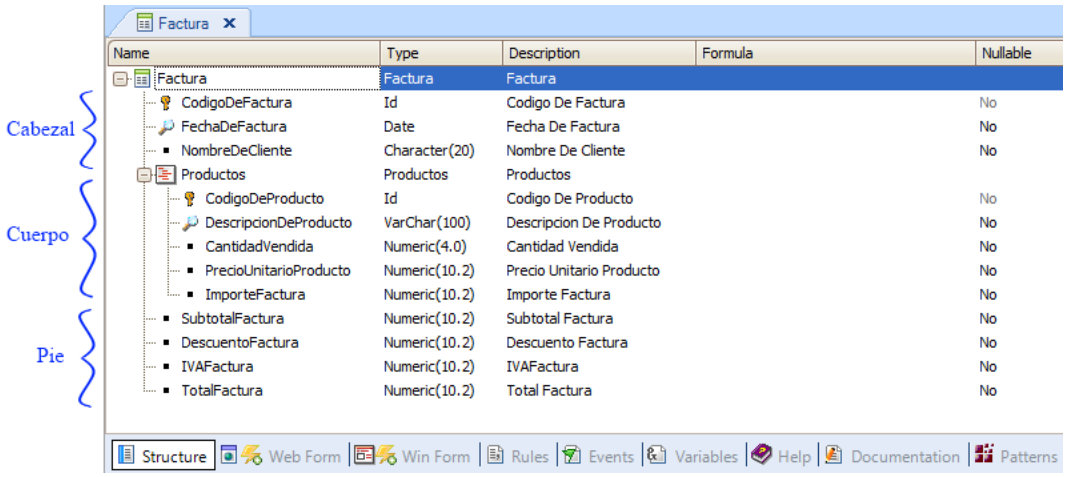
TRANSACCIONES

Cada programador tiene una o múltiples visiones de los datos que utiliza cotidianamente.

Entre estas visiones, podemos pensar en un primer tipo: el que agrupa aquellas que se utilizan para manipular los datos (introducirlos, modificarlos, eliminarlos y visualizarlos en forma limitada), estas visiones de usuario son llamadas **Transacciones**.

Cada transacción tiene un conjunto de partes predefinidas, las más importantes son:

Estructura de los datos de la Transacción: los datos se presentan de acuerdo a una estructura. A continuación se presenta la estructura de una transacción simple como ejemplo *(fig 2.2.1)*.



(fig 2.2.1 –transacción Factura)

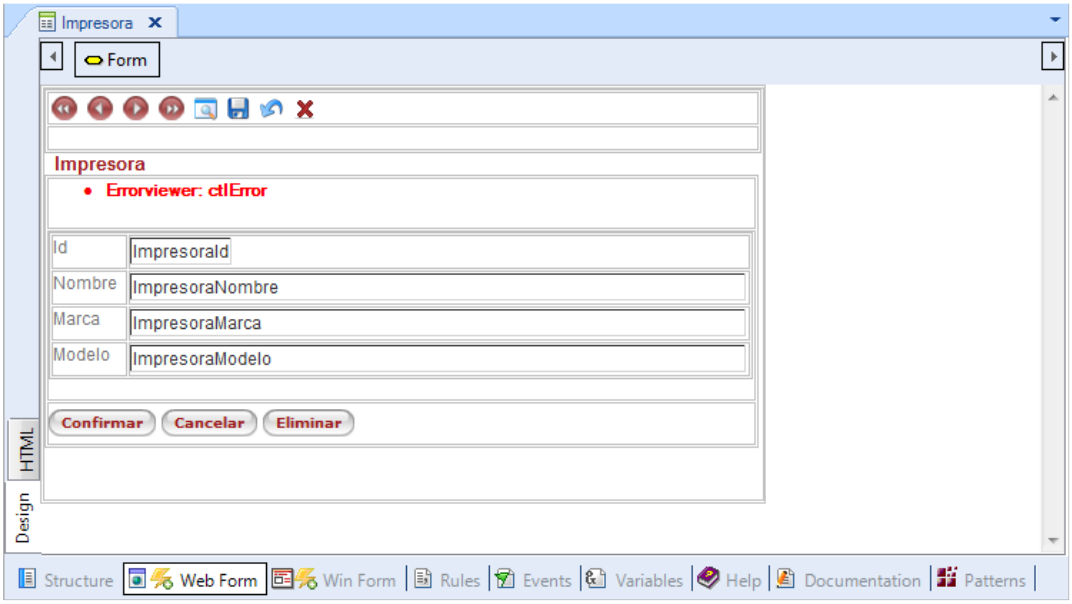
En esta representación existen tres grupos de atributos: un "prólogo o cabezal", que ocurre una sola vez y que aparece al principio, un "cuerpo" que ocurre un cierto número de veces y un "epílogo" o "pie" que ocurre una sola vez. Dentro del cabezal, aparece al lado del atributo ***CodigoDeFactura*** una llave, lo que significa que, para cada Factura (cada ocurrencia del Cabezal y del Pie) existe un único ***CodigoDeFactura***, luego existe un conjunto de atributos: lo cual significa que este conjunto de atributos puede ocurrir múltiples veces dentro de cada ***Factura*** y es llamado cuerpo. El cuerpo *tiene* **debe tener** un identificador, que identifica bien cada una de sus líneas dentro de la ***Factura***. En este caso lo es ***CodigoDeProducto*** y esa condición se señala colocándole un a llave a la derecha. Luego del cierre del grupo repetitivo se encuentra un epílogo o pie.

Reglas: se pueden definir un conjunto de reglas que afectan a la transacción. El propósito es definir su comportamiento, se pueden llegar a crear reglas tan complejas como sea necesario porque su sintaxis permite la invocación de cualquier tipo de procedimientos.

A modo de ejemplo se incluyen las siguientes:
No existirán dos Facturas con el mismo ***CódigoDeFactura***.

El ***CódigoDeFactura*** se atribuirá de manera correlativa.
La ***FechaDeFactura*** tomará como opción por defecto la fecha del día de su emisión.
La ***FechaDeFactura*** no podrá ser menor a la fecha de emisión de dicha Factura.
No se admitirá ninguna ***Factura*** que deje negativo el ***StockDelProducto*** para alguno de sus productos.
No se admitirá ninguna ***Factura*** que haga que el ***SaldoDeudorDelCliente*** involucrado supere su límite de crédito.
Cuando la aceptación de una Factura determine que el ***StockDelProducto < PuntoDePedidoProducto***, para alguno de sus productos, se deberá activar la rutina de ***Aprovisionamiento (Producto)***.

Interfaz: cada transacción proveerá una interfaz gráfica por defecto, que permitirá el mantenimiento de los datos afectados por la Transacción. Esta interfaz puede ser modificada por el usuario puesto que esta parte del objeto tiene las mismas características que los objetos WebPanel o WorkPanel *(fig. 2.2.2 y 2.2.3)* .

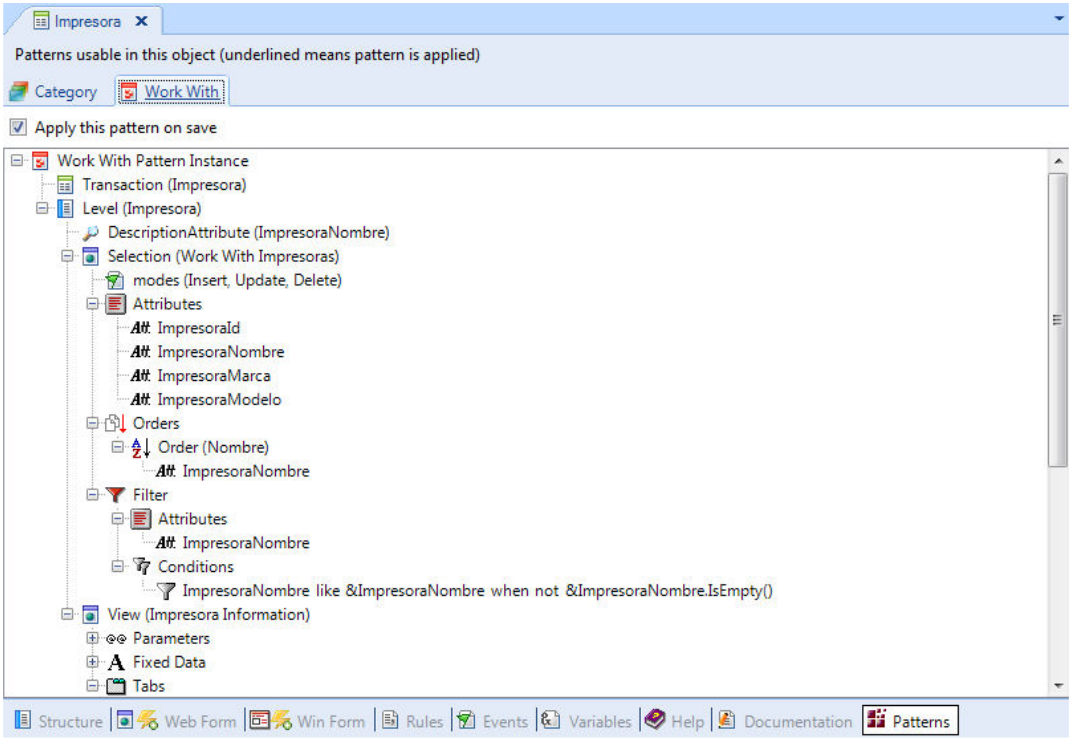


(fig 2.2.2 – diseño de la interfaz)

(fig 2.2.3 – vista del formulario web)

Patrones: para aumentar la productividad y consistencia de los sistemas, GeneXus permite utilizar patrones *(fig 2.2.4)* que pueden adaptarse a cada objeto Transacción. En esta parte

pueden definirse qué patrones se aplicarán a la transacción y pueden definirse aspectos particulares de cada patrón para cada transacción específica. Estos patrones permiten definir el comportamiento de la transacción de forma amigable y sencilla para luego generar el código GeneXus que implemente dicho comportamiento.



(fig2.2. 4 – patrón Work With aplicado a la transacción Impresora)

Fórmulas: se pueden definir un conjunto de fórmulas que afectan a la transacción a través de los atributos, como, por ejemplo, las siguientes.

Importe Factura = ***CantidadVendida **** ***PrecioUnitarioProductoSubTotalFactura*** = sumatoria dentro de la ***Factura (CódigoDeFactura)*** de ***ImporteFactura***
DescuentoFactura = función de cálculo del descuento (***CódigoDeFactura***)
IVAFactura = función de cálculo del IVA (***SubTotalFactura – DescuentoFactura***)
TotalFactura = ***SubTotalFactura – DescuentoFactura*** + ***IVAFactura***
SaldoDeudorDelCliente = sumatoria de facturas impagas (***CódigoDeCliente***)

Bussines Component: es una propiedad de las Transacciones que permite que las mismas sean utilizadas desde cualquier objeto GeneXus, sin necesidad de utilizar la interfaz gráfica. Se define un nuevo tipo de datos con el nombre de la Transacción permitiendo reutilizar la lógica encapsulada en la misma.

PROCEDIMIENTOS
Las Transacciones, según lo visto anteriormente, permiten de manera declarativa describir las visiones de datos de los usuarios y, como consecuencia, atesorar el conocimiento necesario para proyectar, generar y mantener de manera automática, la base de datos y los programas que los usuarios necesitan para manipular sus propias visiones de datos. Pero esto no es suficiente: existe la necesidad de procesos "*batch*" o similares, ya que estos procesos no se pueden inferir a partir del conocimiento aportado por las visiones de datos.

Cuando fue liberada la primera versión de GeneXus, las transacciones resolvían aproximadamente el 70% del problema, generando los programas correspondientes. Para lograr un 100% de cobertura a las necesidades de los clientes, se recurrió a un objeto que permitiera descripciones procedurales: un lenguaje procedural de alto nivel.

El lenguaje utilizado en los Procedimientos es un lenguaje con instrucciones para manipular los datos, de tal forma que no pierden validez ante los cambios estructurales de la base de datos. Además tienen las instrucciones lógicas y aritméticas normales de un lenguaje procedural de alto nivel.

Los procedimientos son comúnmente usados para tres tipos de procesos:

- Procesos *batch* de actualización. Por ejemplo: eliminar todas las facturas de fecha anterior a una fecha dada y que ya fueron pagadas.
- Subrutinas de uso general. Por ejemplo: rutina de monto escrito en donde, dado un importe se devuelve un literal con el importe en letras (1010 => 'Mil diez')
- Procesos a ejecutar en un servidor de aplicaciones o servidor de base de datos.

De esta manera se resuelven todos los problemas casuísticos encontrados, respetando siempre las líneas maestras de consistencia y ortogonalidad de GeneXus.
Un punto alto del objeto es su capacidad procedural que complementa las capacidades declarativas de las Transacciones y uno bajo es el hecho de escribir descripciones procedurales que permiten una productividad menor que la de hacer descripciones declarativas, como es el caso de las Transacciones.

Los "objetos-tipo" más importantes son "**Transacción**" y "**Procedimiento**". Con ellos se puede hacer una descripción de la realidad aunque, con el tiempo, se ha introducido un conjunto importante de nuevos objetos, algunos para lograr una descripción más completa (**GXflow** que permite describir los flujos de trabajo de las operaciones del negocio y generar el código necesario para utilizar el servidor especializado en administrarlos), otros para permitir diálogos más amigables con los usuarios (**Work Panels**, **Web Panels**, **GXportal**), otros para facilitar la reutilización del conocimiento permitiendo un aumento de productividad muy importante (**Business Components**, **Data Providers**, **Mini-Procs**, **Patterns**), otros para llevar al terreno de los usuarios finales la formulación de las consultas a las Bases de Datos (**GXquery**) o a las DataWarehouses (**GXplorer**). Estos objetos pueden combinarse para resolver las necesidades más complejas.
A continuación se describen algunos de ellos.

REPORTES (En versión 9.0)
Un reporte es un proceso que permite visualizar los datos de la base de datos combinados con información adicional definida por el sistema utilizando filtros seleccionados por el usuario. La salida del listado puede ser enviada a pantalla o a la impresora. Con este objeto se pueden definir desde listados simples (por ejemplo, listar los clientes) hasta muy sofisticados, en los cuales existan varios cortes de control, múltiples lecturas a la base de datos y parametrización. Un reporte no puede actualizar la base de datos y su sintaxis es declarativa para definir el diseño del reporte y procedural para la carga de los datos.
Se dispone además de una herramienta GXquery[23] para reportes dinámicos sobre la base de datos. En la última versión de GeneXus este objeto está fusionado con el objeto Procedimiento, que se encarga de proveer todas las funcionalidades que antes proveía el objeto Reporte.

WORK PANELS
Un Work Panel es una pantalla que permite al usuario realizar consultas interactivas a la base de datos. Cuanto más los usuarios utilizan el computador para su trabajo, se torna más necesaria la utilización de diálogos sofisticados, que le permitan sentarse a pensar frente al mismo. Los Work Panels permiten diseñar este tipo de diálogos del usuario.

Por ejemplo: Un Work Panel que muestra la lista de clientes y que permite (a elección del

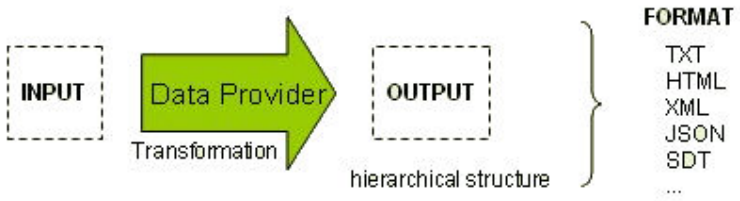
usuario) ver cuáles son sus facturas o su deuda.

WEB PANELS
Son similares al grupo de Work Panels pero requieren un navegador de aplicaciones (Browser) para ser ejecutados en ambientes Internet/Intranet/Extranet.

DATA VIEWS
Permiten considerar correspondencias entre tablas de bases de datos preexistentes y tablas GeneXus y tratarlas con la misma inteligencia que si fueran objetos GeneXus.

DATA PROVIDERS
El tipo de objetos Data Providers, está orientado a facilitar el intercambio de información tanto dentro de la misma aplicación como con aplicaciones externas.

La idea principal del tipo de objeto es simple, dados parámetros de entrada, obtendremos un determinado output como resultado *(fig 2.2.5)*:



(fig2.2.5 – esquema de Data Providers)

Aunque todo Data Provider puede implementarse utilizando el objeto Procedure, existe una gran diferencia entre ellos, mientras el segundo se centra en la transformación de los datos y en la salida de los mismos, el primero se centra simplemente en la salida, haciendo que el objetivo sea más claro. A continuación podemos ver un ejemplo comparando el código de un objeto Procedure y un Data Provider *(fig 2.2.6)*:

Procedure

```
&XmlWriter.Open(...)
&XmlWriter.WriteStartElement('Clients')
For Each
   &XmlWriter.WriteStartElement('Client')
      &XmlWriter.WriteElement('Code', CustomerId.ToString())
      &XmlWriter.WriteElement('Name', CustomerName)
   &XmlWrite.EndElement()
Endfor
&XmlWriter.EndElement()
&XmlWriter.Close()
```

Data Provider

```
Clients
{
     Client
     {
          Code = CustomerId
          Name = CustomerName
     }
}
```

(fig 2.2.6 – Comparación código Procedimiento y Data Provider)

Podemos observar que el código del DP (Data Provider) es más sencillo de interpretar y nos indica inmediatamente cuál es su objetivo, no tenemos en el medio todo el código de transformación que posee el objeto Procedure.

Otro aspecto importante de los Data Providers es que nos permiten exponer los datos de forma sencilla en varios formatos, por ejemplo xml, Json, Txt, Html, SDT, etc., además de permitir ser expuestos como Web Services, lo cual permite utilizarlos para varios propósitos, como por ejemplo RSS Feeds.

Debido a la gran versatilidad que proveen estos objetos, están siendo cada vez más utilizados.

Partiendo de los objetos descritos, el modelo de datos físico es diseñado en base a la Teoría de Bases de Datos Relacionales y asegura una base de datos en tercera forma normal (sin redundancia). Esta normalización es efectuada automáticamente por GeneXus.
El repositorio de GeneXus mantiene las especificaciones de diseño en forma abstracta, o sea que no depende del ambiente objeto, lo que permite que, a partir del mismo repositorio, se puedan generar aplicaciones funcionalmente equivalentes, para ser ejecutadas en diferentes plataformas.

Como consecuencia de lo anterior sería posible, por ejemplo, que un usuario de una aplicación IBM AS/400 centralizada desarrollada 100% con GeneXus, quizás hace 15 años, pueda funcionar total o parcialmente en un ambiente JAVA o .NET sin tener que modificar los objetosoriginales. Desarrollar aplicaciones con GeneXus da la posibilidad de dividir una aplicación de manera tal que cada parte puede ser ejecutada en una plataforma diferente, utilizándose el lenguaje más apropiado para generar los programas en cada una de estas plataformas. Su última versión (Evolution 1) posee una arquitectura extensible, por lo cual pueden agregarse al producto paquetes de software, conocidos como extensiones, capaces de interactuar con la base de conocimiento, pudiendo ser desarrolladas por terceros. Las extensiones GeneXus son comentadas en el capítulo "Desarrollo de GXUnit".

2.2.3. Power Builder

Power Builder es un herramienta de desarrollo o lenguaje 4GL, de clase empresarial, orientada a objetos, que permite el desarrollo de diferentes tipos de aplicaciones y componentes para arquitecturas cliente/servidor, distribuidas y Web. Es uno de los productos del grupo Sybase. PowerBuilder incluye, dentro de su ambiente integrado de desarrollo, herramientas para crear la interfaz de usuario, generar reportes y tener acceso al contenido de una base de datos.

Las aplicaciones son dirigidas por eventos, o sea el programador define las acciones a realizar cuando se dispare el evento correspondiente. Los componentes de la aplicación que se definen con Power Builder tienen eventos predefinidos, por ejemplo, un botón tiene el evento "*click*".

Power Builder también incluye un lenguaje de programación llamado **Powerscript**, el cual es usado para especificar el comportamiento de la aplicación en respuesta a eventos del sistema o del usuario, tal como cerrar una ventana o presionar un botón. Powerscript provee una gran cantidad de funciones predefinidas, por ejemplo, hay una función para actualizar la base de datos, para abrir ventanas, también permite definir nuevas funciones.

Cada componente creado en Power Builder es un módulo auto-contenido o sea un objeto. Cada objeto contiene características particulares y comportamientos (propiedades, eventos y funciones), al ser orientado a objetos se puede tomar ventaja de distintas técnicas, tales como polimorfismo, encapsulamiento y herencia.

Power Builder brinda la posibilidad de trabajar con varios ambientes en un área de trabajo determinado, abrir y editar objetos en distintos ambientes a la vez. Un ambiente de Power Builder puede ser de distintos tipos:

Ambiente de Aplicación

Una aplicación ejecutable cliente/servidor o multinivel.

Ambiente .NET

Brinda la posibilidad de desplegar aplicaciones como ventanas .NET o aplicaciones ASP.NET, también Web Services.

Ambiente de Servidor de Aplicaciones o EAServer

Un componente que puede ser desplegado para un EAServer u otro servidor de J2EE

Objetos Power Builder

La estructura básica de un ambiente Powerscript son los objetos. A continuación se describen los más importantes:

APPLICATION

El objeto Application es el punto de entrada a la aplicación. Es el objeto que define el comportamiento de la aplicación, por ejemplo, qué fuente se utiliza para el texto, qué se debe hacer cuando finalice/comience la aplicación.

WINDOW

El objeto Window es la interfaz primaria entre los usuarios y una aplicación Power Builder, mediante este objeto se puede desplegar información o requerirla al usuario y responder a acciones del teclado y mouse. En el objeto Window se pueden alojar varios controles, entre ellos botones, textos, tablas.

DATA WINDOW

Es un objeto que se utiliza para recuperar y manipular datos de una base relacional o de otro origen como una planilla Excel. Este objeto brinda la posibilidad de elegir la forma de mostrar la información.

MENU

Son listas de ítems que el usuario puede seleccionar, con el mouse o con el teclado. Cada ítem del menú es definido como un objeto Menu.

GLOBAL FUNCTION

En Power Builder se pueden definir dos tipos de funciones: a nivel de objeto y globales. Las funciones a nivel de objeto son definidas para un tipo de objeto en particular, como una ventana o un menú y son encapsuladas con el objeto que fueron definidas. Las globales no son encapsuladas, son guardadas como un objeto aparte. Tiene propósito general y no actúan en una instancia particular de un objeto.

QUERIES

Son sentencias SQL que se guardan con un nombre y de esta forma pueden ser utilizadas como origen de datos para un DataWindow. Las queries son definidas una vez y después pueden ser reutilizadas.

STRUCTURE

Un Structure es una colección de variables relacionadas, de igual o distinto tipo agrupadas bajo un mismo nombre. Este objeto permite referirse a entidades relacionadas como si fueran una unidad. Existen dos tipos de Structure: globales y a nivel de objeto. Las estructuras a nivel de objeto están asociadas a un tipo de objeto en particular, como una ventana o menú y siempre pueden ser utilizadas en scripts referentes al objeto. Los objetos de tipo Structures globales no están asociados a ningún tipo de objeto, pueden declararse instancias y ser utilizadas en cualquier script de la aplicación.

USER OBJECT

Las aplicaciones generalmente tienen funcionalidades que se usan repetidamente en diferentes partes. Con los objetos User Object se puede definir una vez la funcionalidad y utilizarla cuantas veces sea necesario. Estos objetos se dividen en visuales y no visuales. Los Visual User Objects son controles reutilizables o un conjunto de controles que tienen un comportamiento consistente. Por ejemplo pueden consistir en un conjunto de botones que funcionan como una unidad. Los non Visual User Object son módulos de procesamiento reutilizables, no tienen componentes visuales. Son utilizados típicamente para definir las reglas de negocio y otros procesos que actúan como una unidad y pueden ser ejecutadas en un servidor.

LIBRARIE

Los objetos son guardados en librerías. Cuando se ejecuta una aplicación los objetos son obtenidos de las librerías. Las aplicaciones pueden utilizar cualquier cantidad de librerías y cuando son creadas se especifican las librerías que se quieren utilizar.

3. Software para construcción y ejecución de pruebas

En el mercado existen múltiples herramientas para probar unidades de software, diseñadas para distintos tipos de lenguajes y con distintos objetivos. A continuación veremos las que fueron consideradas más relevantes por su cantidad de usuarios y similitud con este proyecto. La sección se divide entre las herramientas denominadas *xUnit* y las que no entran en esta categoría.

3.1. *Frameworks*[1] *xUnit*

xUnit se le denomina a un conjunto de frameworks de pruebas unitarias en distintos lenguajes de programación y plataformas de desarrollo. Estos frameworks permiten probar diferentes elementos o unidades de software como funciones u objetos. La filosofía de la propuesta consiste en crear y ejecutar pruebas embebidas en el framework, escribiéndolas y controlando los resultados en el mismo lenguaje en que está escrito el programa a probar[10]. La ventaja principal de los frameworks *xUnit* es que, al proveer una solución automatizada, no se necesita escribir las mismas pruebas una y otra vez ni recordar cuál es el resultado que deberían dar ni codificar el programa que las invoque. Por lo tanto, es un medio de controlar las pruebas de regresión, necesarias cuando una parte del código ha sido modificada y se desea ver que el nuevo programa cumple con los requerimientos anteriores y que su funcionalidad no se ha alterado.

La mayoría de los frameworks funcionan de la siguiente manera: dependiendo del contexto de la prueba (algún valor de entrada y/o del estado de la base de datos) se evalúa el valor de retorno esperado; si la unidad cumple con la especificación, entonces el framework devolverá que el método del objeto pasó exitosamente la prueba; en caso de que el valor esperado sea diferente al obtenido durante la ejecución, el framework devolverá un fallo en el método correspondiente. A su vez, permite crear tantas pruebas como sea necesario.

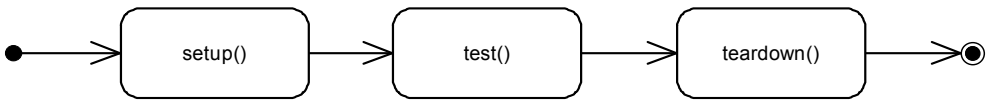
Para garantizar su resultado, las pruebas deben aislarse y su ambiente debe ser creado y restaurado antes y después de su ejecución respectivamente, postulando los siguientes patrones básicos:

- **Caso de prueba.** Método o procedimiento, generado usualmente en el lenguaje del framework, que verifica cierta funcionalidad o funcionalidades.

- **Contexto de pruebas.** Un contexto de prueba (*test fixture*) es el conjunto de precondiciones o estados requeridos para correr una prueba. El desarrollador debe armar el estado antes de las pruebas y luego volver al estado original.

- **Suite de pruebas.** Es un conjunto de pruebas que comparten el mismo contexto. Puede haber orden de precedencia en la ejecución de las pruebas.

- **Ejecución de las pruebas.** La ejecución de una prueba unitaria individual es la siguiente *(fig 3.1.1)*:



(fig 3.1.1 – secuencia de ejecución de las pruebas)

1- Por definición ver glosario.

En la figura, el método test es el caso de prueba, mientras que los métodos *setup* y *teardown* inicializan y limpian el contexto de pruebas respectivamente.

La mayoría de los frameworks *xUnit* han sido implementados en lenguajes orientados a objetos y proveen un conjunto básico de funcionalidades que siguen los patrones mencionados anteriormente[24]. Estas funcionalidades incluyen:

- La especificación de los resultados esperados con invocaciones a métodos de comprobación (*Assertion Methods*). Estos métodos implementan el oráculo que queda incluido en el código de la prueba. Las decisiones deben ser booleanas y verificables por una computadora. También es posible comprobar excepciones esperadas, en cuyo caso la falla es notificada sólo si la excepción no se produce[25].
- La ejecución de un conjunto de pruebas (*suites*) como una única operación.
- La exposición de los resultados de las pruebas. La ejecución de las pruebas pierde su sentido si no se conocen los resultados. Éstos deben ser comprendidos fácilmente y percibidos de forma no ambigua. La prueba se detiene ante la falla, aunque generalmente no impide seguir ejecutando el resto de las pruebas. Se utiliza una semántica de colores definida por: verde = pasa, rojo = falla.

Cabe destacar que *xUnit* ha sido extendido a más de 30 lenguajes[25, 26] y su versión para Java, denominada JUnit [27], se ha transformado en el estándar para desarrollar pruebas unitarias en Java[28]. Actualmente, los frameworks se incorporan a los Entornos Interactivos de Desarrollo(*IDEs*, su sigla en inglés) y existe gran cantidad de extensiones para éstos[29].

A continuación se listan las herramientas con metodología xUnit que se consideraron más relevantes para el proyecto.

JUnit

Vale la pena destacar a JUnit [27] por ser el framework *xUnit* más conocido y utilizado, creado para probar aplicaciones Java.

Es un conjunto de clases que permite realizar la ejecución de clases Java de manera controlada, para poder evaluar si el funcionamiento de cada uno de los métodos de la clase se comporta como se espera.

El framework incluye formas de ver los resultados en modo texto o gráfico. En la actualidad las herramientas de desarrollo como Eclipse[30] y NetBeans[31] cuentan con plug-ins que permiten ejecutar las pruebas JUnit y ver los resultados integrados en su *IDE*. Luego de cada ejecución de un conjunto de pruebas, la herramienta permite ver los resultados satisfactorios en verde y en rojo los no satisfactorios, mostrando a su vez en qué lugar de la prueba (en cuál *assertion*) ocurrió el fallo.

Utilización de JUnit

A continuación se ofrecerá un ejemplo de cómo codificar, ejecutar y observar el resultado de una prueba utilizando JUnit. Se mostrarán fragmentos de código para ejemplificar la sintaxis e identificar los patrones de la arquitectura *xUnit* utilizados. El ejemplo corresponde a la versión de JUnit 4.6.

La primera figura muestra el código de una clase simple, llamada Persona *(fig 3.1.2)*, y la segunda, su clase de prueba asociada *(fig 3.1.3)*:

.

```
1
2  public class Persona {
3
4      private String nombre;
5      private int edad;
6
7      public Persona(String nom, int edad) {
8          this.nombre = nom;
9          this.edad = edad;
10     }
11
12     public void setEdad(int edad) {
13         if (edad < 0)
14             throw new IllegalArgumentException();
15         else
16             this.edad = edad;
17     }
18
19     public String getNombre() {
20         return this.nombre;
21     }
22
23     public int getEdad() {
24         return this.edad;
25     }
26
27 }
```

(fig 3.1.2 – código Java de la clase Persona)

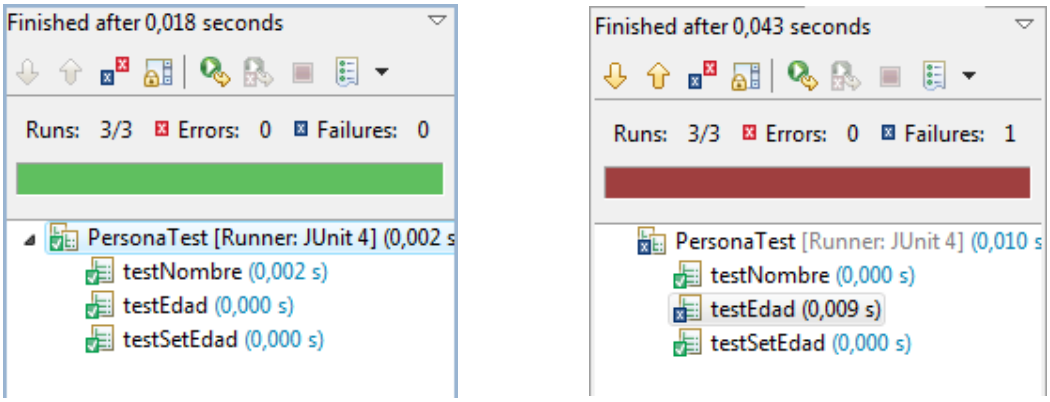

```
1
2  import static org.junit.Assert.*;
8
9  public class PersonaTest {
10
11     private Persona persona;
12
13     public static junit.framework.Test suite() {
14         return new JUnit4TestAdapter(PersonaTest.class);
15     }
16
17     @Before
18     public void setUp() {
19         persona = new Persona("Jorge Lopez",25);
20     }
21
22     @Test
23     public void testNombre() {
24         assertEquals("La persona debería llamarse Jorge Lopez", "Jorge Lopez", persona.getNombre());
25     }
26
27     @Test
28     public void testEdad() {
29         assertEquals("La persona debería tener 25", 25, persona.getEdad());
30     }
31
32     @Test (expected=IllegalArgumentException.class)
33     public void testSetEdad() {
34         persona.setEdad(-1);
35     }
36
37     @After
38     public void tearDown() {
39         persona = null;
40     }
41
42 }
```

(fig 3.1.3 – código de la clase de prueba asociada a la clase Persona)

El método *suite,* se utiliza para poder correr la prueba dentro de alguna suite de prueba (más adelante se ejemplificará). El método *setUp* contiene la anotación *Before* por lo que será ejecutado antes que las pruebas, mientras que *tearDown* al ser precedido de la anotación *After*, se ejecutará luego de finalizadas las mismas. Estos métodos son los que inicializan y limpian el contexto de pruebas respectivamente. Los demás procedimientos llevan la anotación *Test*, lo que indica que son casos de prueba. Dentro de *testNombre* y *testEdad* puede verse cómo se utilizan *métodos assert* para validar el resultado de la prueba. Los métodos *assert* implementan el oráculo al comparar el resultado de la ejecución del caso de prueba con el resultado esperado. En este caso se utilizó el método *assertEquals*, pero existen numerosos métodos *assert* para comparar distintos tipos de resultados[32].

Ejecución de pruebas y visualización de resultados

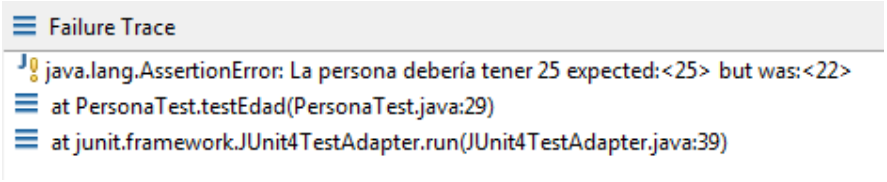
Las nuevas *IDEs* de desarrollo para Java traen apoyo para la ejecución y visualización de los resultados de las pruebas como parte del paquete básico de instalación o plug-ins. A continuación se muestran ejemplos de cómo se visualiza el resultado de correr pruebas utilizando el *IDE* Eclipse[30]. En la primera imagen se muestra una ejecución satisfactoria, mientras que en la segunda se muestra una falla en uno de los casos de prueba *(fig 3.1.4)*.



(fig 3.1.4 – Visualización de los resultados de los casos de prueba)

En la imagen se pude ver el informe de ejecución que incluye: la cantidad de pruebas ejecutadas, cantidad de errores, fallas y tiempo de ejecución de cada prueba.

En la siguiente imagen se muestra el mensaje de error desplegado *(fig 3.1.5)*:



(fig 3.1.5 – Mensaje de error)

En este mensaje pueden observarse el resultado esperado y el resultado obtenido del caso de prueba que falló.

Como fue mencionado anteriormente, además de la visualización gráfica de los resultados mediante las *IDEs* de desarrollo, también pueden ejecutarse y visualizarse los resultados de las pruebas utilizando la consola. La utilización de la consola independiza del uso de una *IDE* de desarrollo, además de permitir la ejecución de pruebas *batch*, programadas para ejecutarse por ejemplo, cada noche. A continuación se muestra el resultado de una ejecución por consola *(fig 3.1.6)*:

```
.............................F.........
.....................................
Time: 0,026
There was 1 failure:
1) testAssertStringEquals(junit.tests.framework.AssertTest)junit.framework.ComparisonFailure: null expected:<[a]> but was:<[b]>
        at junit.tests.framework.AssertTest.testAssertStringEquals(AssertTest.java:45)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)
        at junit.tests.framework.AllTests.main(AllTests.java:13)

FAILURES!!!
Tests run: 80,  Failures: 1,  Errors: 0
```

(fig 3.1.6 – Ejecución de las pruebas utilizando la consola)

En la figura 3.1.6 se muestra el resultado de correr la suite *AllTests* de las pruebas realizadas al framework JUnit, cambiando el código para que ocurriera una falla.

Construcción de una *suite* de pruebas

Se puede también ejecutar un conjunto de casos de prueba de una sola vez utilizando el concepto de "suite de pruebas". Utilizando el ejemplo anterior suponemos que existe un conjunto de clases asociado de alguna manera a la clase Persona. Para probar todos los casos

mediante una sola ejecución creamos la clase *PersonaSuite (fig 3.1.7)*.

```
1  import org.junit.runner.RunWith;
2  import org.junit.runners.Suite;
3  import org.junit.runners.Suite.SuiteClasses;
4
5  import junit.framework.JUnit4TestAdapter;
6  import junit.framework.Test;
7
8  @RunWith(Suite.class)
9  @SuiteClasses({
10     PersonaTest.class
11     //Todas las clases de prueba de personas
12 })
13
14 public class PersonaSuite {
15     public static Test suite() {
16         return new JUnit4TestAdapter(PersonaSuite.class);
17     }
18 }
```

Se agregan los casos o suites de prueba que serán incluidos en la ejecución

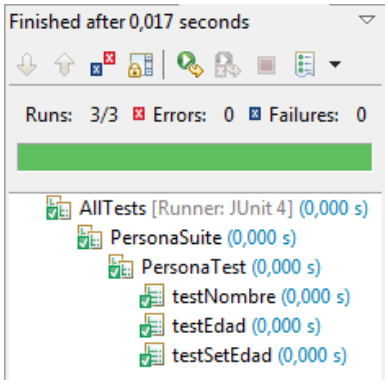
(fig 3.1.7 – Clase PersonaSuite)

Aquí se agregan todos los casos de prueba que se deseen ejecutar con la suite. A su vez pueden agregarse al conjunto suites completos obteniéndose así cualquier cantidad de niveles de suites de pruebas. Por ejemplo, la siguiente imagen muestra una suite (llamada *AllTests*) que contiene a la suite *PersonaSuite (fig 3.1.8)*.

```
1
2  import org.junit.runner.RunWith;
3  import org.junit.runners.Suite;
4  import org.junit.runners.Suite.SuiteClasses;
5
6  import junit.framework.JUnit4TestAdapter;
7  import junit.framework.Test;
8
9  @RunWith(Suite.class)
10 @SuiteClasses({
11     PersonaSuite.class
12     // Todos los suites
13 })
14
15 public class AllTests {
16     public static Test suite() {
17         return new JUnit4TestAdapter(AllTests.class);
18     }
19 }
```

(fig 3.1.8 – Suite AllTests)

La ejecución de esta *suite* muestra un resultado de la siguiente manera *(fig 3.1.9)*:



(fig 3.1.9 – Resultado de ejecución de una suite)

Gracias a la utilización de suites pueden efectuarse múltiples pruebas en la misma ejecución, por lo tanto, tiene sentido administrar el contexto de un caso de prueba en particular. Esto puede realizarse mediante las anotaciones *BeforeClass* y *AfterClass* para inicializar y limpiar el contexto de la prueba dentro de una clase.

PBUnit

PBUnit[33] es un Framework para realizar Pruebas Unitarias en Power Builder, está basado en JUnit y fue adaptado desde Java por John Urberg. A continuación se verán ejemplos de utilización de esta herramienta con el fin de conocer cómo se hacen pruebas unitarias en un lenguaje 4GL concreto[34, 35].

El primer paso para comenzar a escribir las pruebas unitarias con PBUnit es agregar la librería "PBUnit.pbl" al proyecto a ser probado. Para escribir el código que va a probar un objeto de la aplicación se crea una sub-clase de TestCase, y se crea un nuevo evento que comienza con la cadena "test". La clase TestCase se encuentra en la librería "PBUnit.pbl".

```
/*      test_calculateTax event, sub-clase de testCase    */
n_tax   ln_tax
dec     ld_ret

ln_tax = create n_tax
ld_ret = ln_tax.calculateTax(100.00)

// se hace un assert tal que el resultado esperado de calculateTax es 25.00
this.assert(ld_ret = 25.00)
```

Si el valor pasado por parámetro a la función *assert* es verdadero (por ejemplo el valor de retorno de la función *calculateTax* es 25 cuando se la llama con el valor 100) entonces la prueba pasa satisfactoriamente, si no la prueba falla. También cuenta con dos funciones *assertNotNull* y *assertIsValid*, ambas, junto con la función *assert* dan la posibilidad de definir el mensaje a ser desplegado en caso de error.

```
/*        test_calculateTax event, la clase es sub-clase de testCase                */

this.assertNotNull("No se retornan valores para 100.00", ld_ret)
this.assert("No se obtiene el resultado esperado para 100.00 (" + string(ld_ret) + ")", ld_ret =
25.00)
```

PBUnit brinda la posibilidad de definir una porción de código que se va a ejecutar al inicio de todas las pruebas definiéndola sólo una vez en el evento *setup*. Si se quiere ejecutar un mismo código al finalizar cada prueba se lo debe agregar al evento *teardown*. PBUnit llama los eventos *setuUp*() y *tearDown*() para cada prueba, de esta forma no hay impactos negativos entre las pruebas, quedan aisladas.

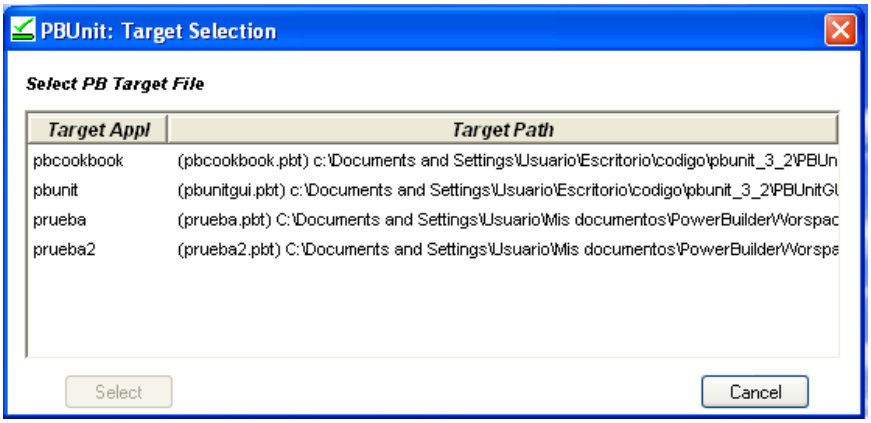
Los casos de prueba se pueden agrupar y gestionar juntos, creando una sub-clase de *testSuite* y agregando cada caso de prueba, inicializándolos en el evento constructor.

```
/*        constructor() event para una sub-clase de testSuite        */

initialize("tst_n_tax")    // el texto entre comillas es el nombre de un caso de prueba
```

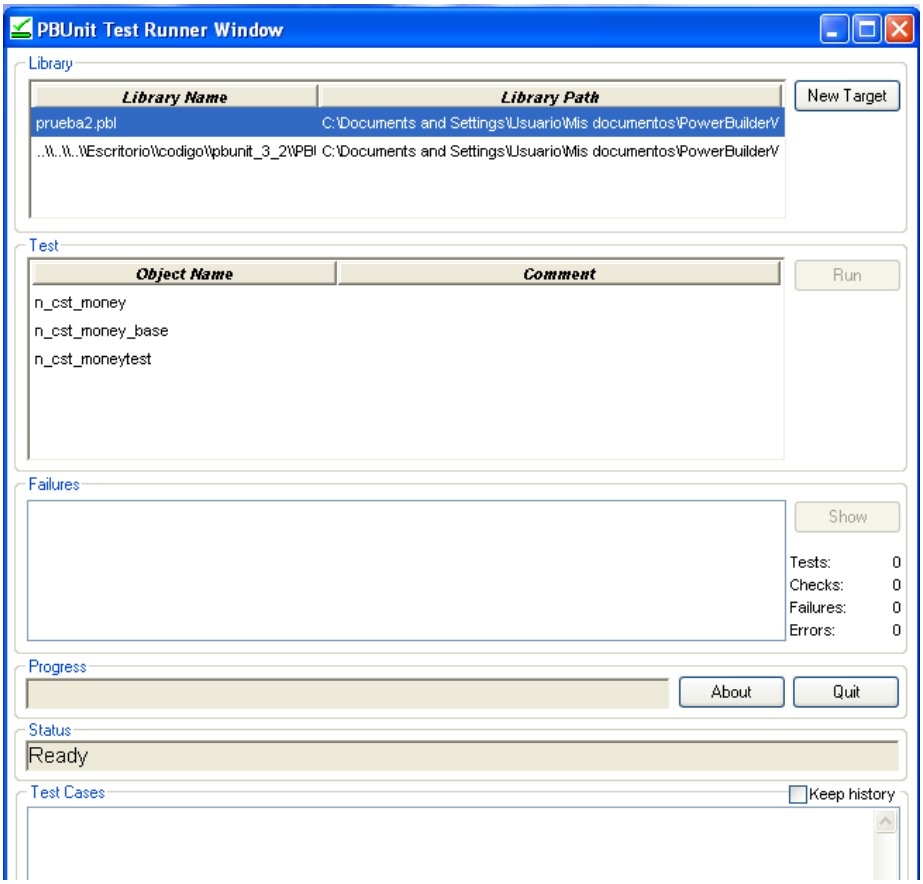
Hay dos formas de ejecutar las pruebas en PBUnit: método Estático y Dinámico. Para ejecutar el método estático se sobrescribe el método *runTest*() en el objeto sub-clase de *TestObject* agregando los llamados a las pruebas que se quiera ejecutar. Con el método estático hay que recordar de modificar el evento *runTest*() cuando se quiere agregar llamados a otras pruebas. Utilizando el método dinámico no se sobrescribe el método *runTest*(), se definen las pruebas como eventos sin argumentos y se les asignan un nombre que comience con *test*, de esta forma todos los eventos que comienzan con test son ejecutados automáticamente por PBUnit.

PBUnit tiene una interfaz gráfica que permite ejecutar las pruebas. Al iniciar la herramienta busca los proyectos de Power Builder que tienen pruebas definidas con PBUnit, con el objetivo de seleccionar qué proyecto probar *(fig 3.1.10)*.



(fig 3.1.10 – Selección de Proyecto a Probar)

Una vez inicializado y seleccionado el proyecto, se puede elegir qué prueba ejecutar *(fig 3.1.11)*.



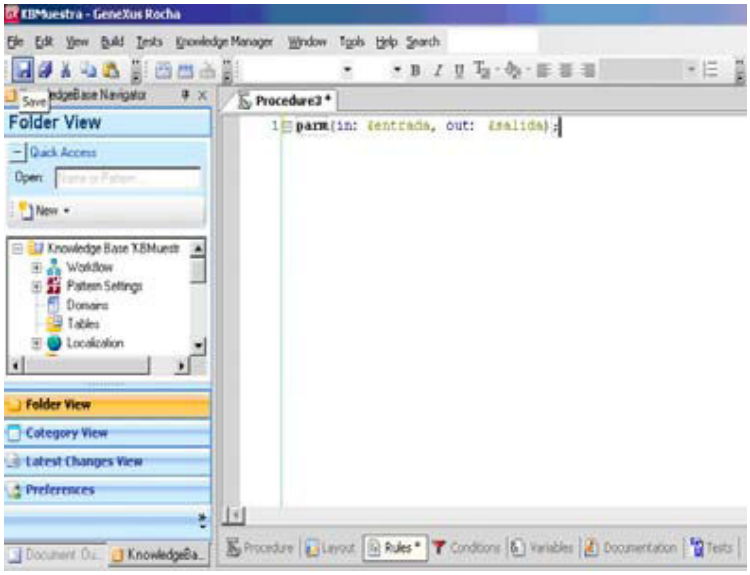
(fig 3.1.11 – Selección de Prueba a Ejecutar)

3.1.1. GXUnit (versión 2007)

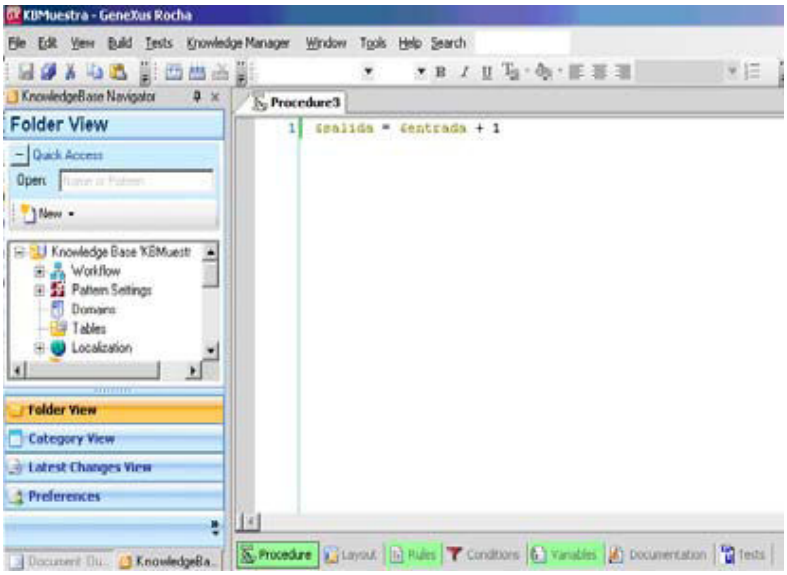
Como ya se mencionó, en el año 2007 en el curso Proyecto de Ingeniería de Software de la Facultad de Ingeniería de la UdelaR[8] se propuso un proyecto para el desarrollo de una herramienta para pruebas unitarias en GeneXus. La herramienta permite definir y especificar instancias de objetos para prueba unitaria ("*fixtures*") de otras instancias de "objetos tipos" GeneXus del sistema. Los objetos para prueba recogen la especificación de la prueba a implementar y a partir de ésta se genera el código necesario para ejecutarla en diferentes plataformas.

A continuación presentamos un ejemplo de utilización de la herramienta GXUnit, con el objetivo de comprender su funcionamiento y mostrar las funcionalidades que brinda.

En este ejemplo se desea crear un objeto para probar un procedimiento GeneXus. En las figuras *(fig 3.1.12)* y *(fig 3.1.13)* se muestran sus parámetros (regla *Parm*) y su código, respectivamente.

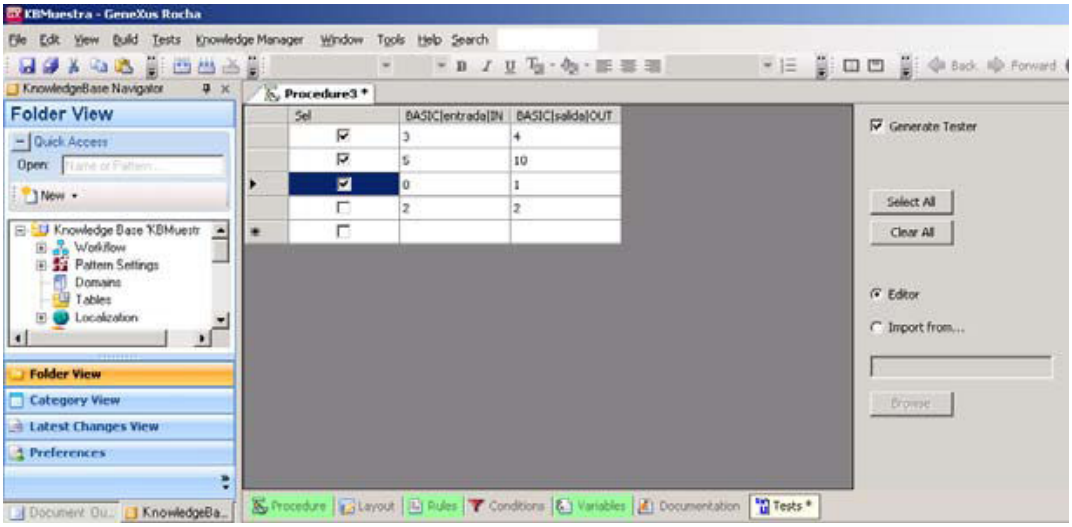


(fig 3.1.12 – Regla Parm del objeto a probar)



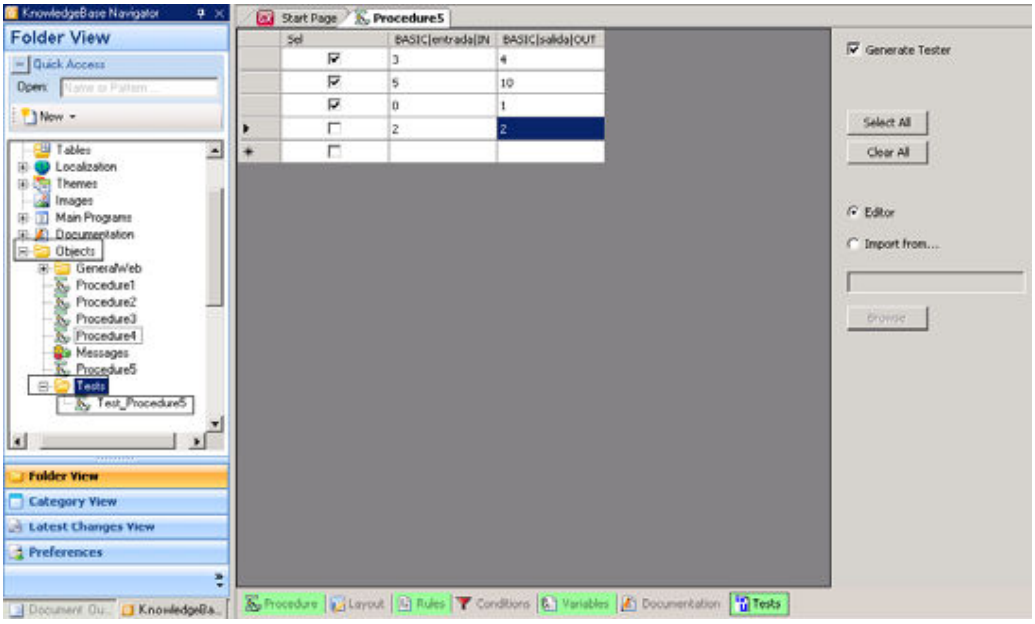
(fig 3.1.13 – Código del objeto Procedimiento a probar)

Los datos de prueba se ingresan a la pestaña "*Tests*" (parte agregada al objeto). El editor implementado muestra la grilla, donde las columnas se corresponden a los parámetros del procedimiento a probar y las filas a los casos de prueba. Es posible seleccionar qué casos de prueba se van a correr en la próxima ejecución. También es posible cargar la grilla a partir de un archivo XML (marcando la opción "*Import from…*") *(fig 3.1.14)*.



(fig 3.1.14 – Editor de pruebas)

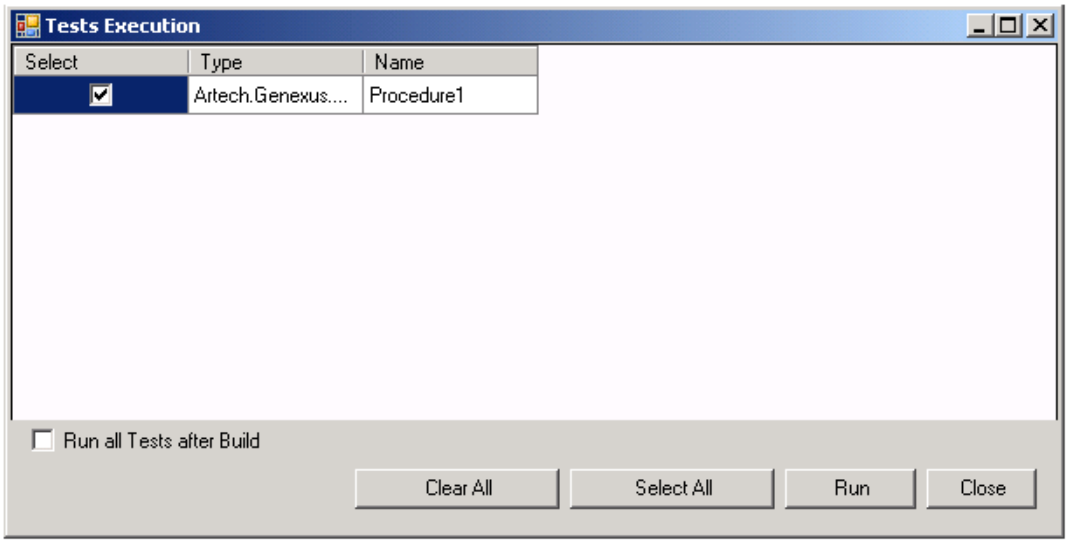
Al guardar el procedimiento a probar se genera el procedimiento verificador y el archivo XML con los datos de los casos de prueba *(fig 3.1.15)*.



(fig 3.1.15 – Creación del Procedimiento Verificador)

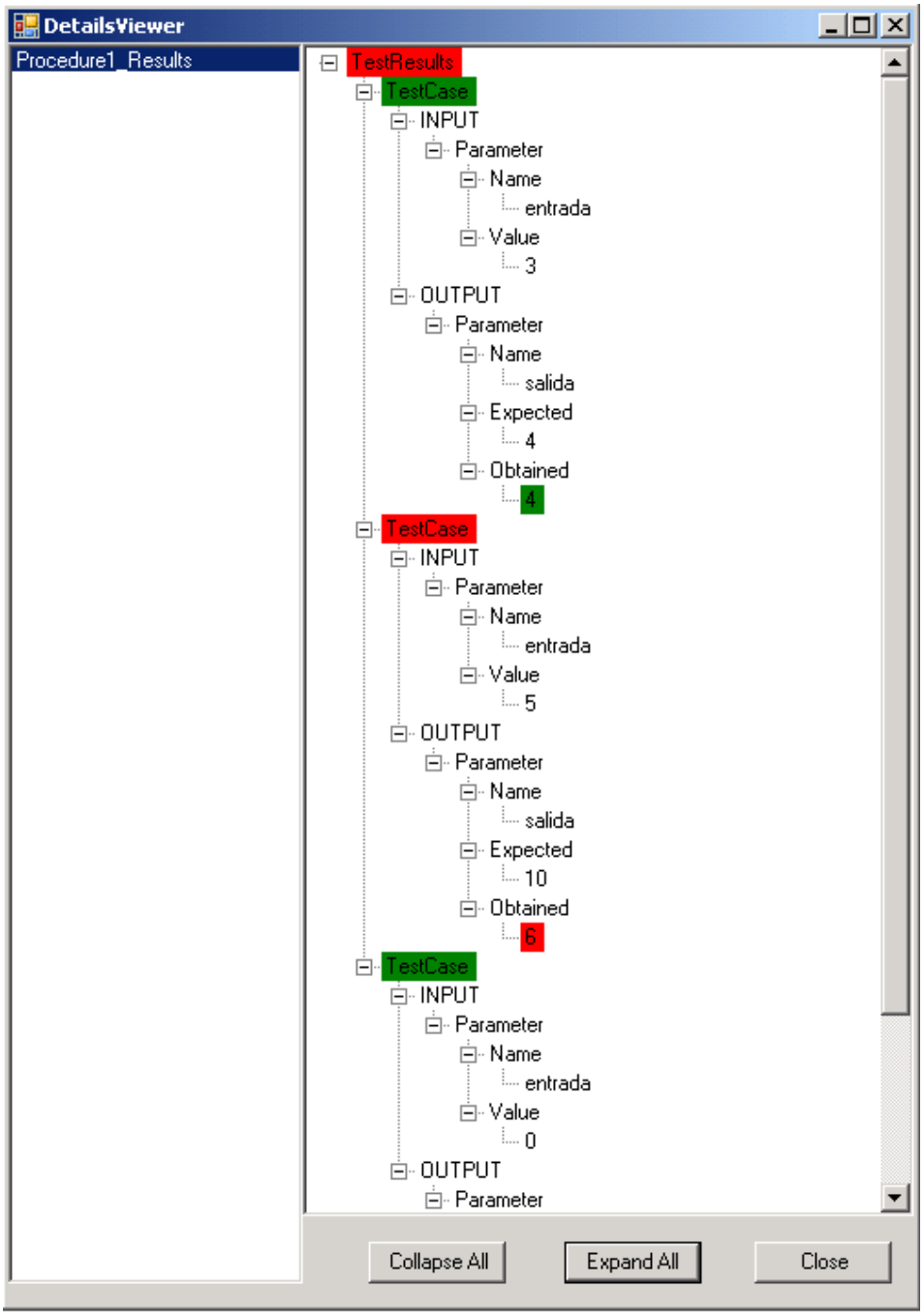
Se ejecutan con la opción "*Run Tests*" del menú "*Tests*" *(fig 3.1.16), (fig 3.1.17)*.

(fig 3.1.16 – Menú de Pruebas a Ejecutar)



(fig 3.1.17 – Selección de pruebas a Ejecutar)

El resultado de la ejecución aparecerá en un archivo XML, visible desde el *IDE (fig 3.1.18)*.



(fig 3.1.18 – Resultado de la ejecución de los casos de prueba)

3.2. *Otras herramientas de pruebas de software*

Mockrunner

Mockrunner [36] es un framework de pruebas unitarias de aplicaciones Java que extiende JUnit. Soporta verificación de aplicaciones web y simulación de acceso a la base de datos. Por esta razón se decidió agregar Mockrunner a la lista de frameworks de pruebas unitarias que fueron estudiados, ya que GeneXus tiene una muy estrecha relación con las bases de datos.

Utilizando Mockrunner para simular el acceso a la base de datos

Mockrunner permite simular conexiones con la base de datos para así evitar que se tengan que ajustar los datos de la base y que éstos sean modificados o corrompidos por las pruebas. Utilizando Mockrunner puede construirse una prueba de un método que necesita acceso a la base, sin ejecutar ninguna sentencia SQL en la base real. Se puede especificar cuál será el resultado de una consulta armando manualmente el resultado que devolverá o cargándolo desde un archivo de texto. Para demostrar su funcionamiento, siguiendo con el ejemplo planteado para JUnit, se construyó la clase *PersonaManager* que obtiene de la base un conjunto de personas *(fig 3.2.1)*:

```
1
2  import java.sql.Connection;
3  import java.sql.ResultSet;
4  import java.sql.SQLException;
5  import java.util.ArrayList;
6  import java.util.List;
7
8  public class PersonaManager {
9      private Connection connection;
10
11     public Connection getConnection() {
12         return connection;
13     }
14
15     public void setConnection(Connection connection) {
16         this.connection = connection;
17     }
18
19     public List<Persona> encuentraPersonasMenores() {
20         List<Persona> result = new ArrayList<Persona>();
21         try {
22             String sql = "SELECT nombre,edad FROM Personas WHERE edad < 18";
23             ResultSet rs = connection.createStatement().executeQuery(sql);
24             while (rs.next()) {
25                 Persona persona = new Persona(rs.getString("nombre"),rs.getInt("edad"));
26                 result.add(persona);
27             }
28         } catch (SQLException exception) {
29             exception.printStackTrace();
30         }
31         return result;
32     }
33
34 }
```

(fig 3.2.1 – Clase PersonaManager)

Se utiliza el método *setConnection* para poder establecer la conexión simulada o la real desde donde la clase es llamada.

La clase *PersonaManagerTest (fig 3.2.2)* es la clase de prueba de *PersonaManager*.

```
1
2⊕import java.sql.Connection;
12
13 public class PersonaManagerTest extends BasicJDBCTestCaseAdapter {
14     private Connection jdbcConnection;
15
16     @Before
17     protected void setUp() {
18         MockConnection connection = getJDBCMockObjectFactory().getMockConnection();
19         StatementResultSetHandler resultSetHandler = connection.getStatementResultSetHandler();
20         jdbcConnection = connection;
21
22         MockResultSet resultSet = resultSetHandler.createResultSet();
23         resultSet.addColumn("nombre");
24         resultSet.addColumn("edad");
25         resultSet.addRow(new Object[] { "Juan Perez", new Integer(16) });
26         resultSet.addRow(new Object[] { "Jorge Gonzalez", new Integer(7) });
27
28         resultSetHandler.prepareResultSet("SELECT nombre,edad FROM Personas WHERE edad < 18", resultSet);
29     }
30
31     @Test
32     public void testEncuentraPersonasMenores() {
33         PersonaManager instance = new PersonaManager();
34         instance.setConnection(jdbcConnection);
35         List<Persona> result = instance.encuentraPersonasMenores();
36         assertEquals(2, result.size());
37         Persona persona = result.get(0);
38         assertEquals("Juan Perez", persona.getNombre());
39         assertEquals(16, persona.getEdad());
40     }
41 }
```

(fig 3.2.2 – Clase PersonaManagerTest)

En el método *setUp* se crea la conexión para la simulación. En este caso se carga el resultado que devolverá la consulta manualmente y se asocia el resultado a la consulta que se realizará.

Luego, como el framework es una extensión de JUnit pueden utilizarse *asserts* para verificar los resultados obtenidos y a su vez, se provee de ciertos métodos *verifyX* específicos para verificaciones de conexiones a la base de datos. Estos métodos son análogos a los *asserts*, son decisiones booleanas que controlan si ciertas condiciones o sentencias fueron ejecutadas. Las más relevantes son:

- *verifySQLStatementExcecuted* y *verifySQLStatementNotExcecuted* verifican si se ejecutó o no respectivamente una sentencia SQL.
- *verifySQLStatementParameter* verifica si cierto parámetro de la consulta es el esperado.
- *verifyCommitted*, *verifyNotCommitted* verifican que se haya realizado un *commit* o no respectivamente.
- *verifyRolledBack*, *verifyNotRolledBack* verifican que se haya realizado un *rollback* o no respectivamente.
- *verifyConnectionClosed* verifica que la conexión se haya cerrado correctamente.

También es posible cargar el resultado de una consulta a partir de un archivo de texto, ya que cargarlo manualmente puede ser tedioso y demasiado estático. En el siguiente ejemplo puede verse como construir el resultado de una consulta tomando los datos desde un archivo de texto *(fig 3.2.3)*:

```
FileResultSetFactory factory = new FileResultSetFactory("src/com/mockrunner/example/jdbc/bookstore.txt");
factory.setFirstLineContainsColumnNames(true);
MockResultSet result = getStatementResultSetHandler().createResultSet("bookresult", factory);
getStatementResultSetHandler().prepareResultSet("select.*isbn,.*quantity.*", result);
```

(fig 3.2.3 – Ejemplo de carga de los datos de prueba utilizando un archivo de texto)

La ejecución de las pruebas se realiza de la misma forma que JUnit ya que el Mockrunner es

una extensión del mismo.

Test Manager y Visual Studio 2010

Visual Studio es un ambiente de desarrollo muy utilizado para desarrollar aplicaciones .NET por lo que sus herramienta de pruebas unitarias merecen ser estudiadas.

Visual Studio 2010[37] provee un nuevo manejador de pruebas que permite al usuario crear pruebas unitarias automatizadas mediante una interfaz de usuario para una aplicación. Las pruebas creadas pueden ser ejecutadas desde Visual Studio 2010, línea de comandos, o ser asociados a un caso de prueba y ser ejecutado desde un plan de pruebas usando Microsoft Test Manager. En esta herramienta al igual que con JUnit luego de ejecutar las pruebas puede verse cuáles casos pasaron las pruebas y cuáles no. Además se pueden generar reportes donde pueden verse los porcentajes de pruebas que pasaron, estadísticas de las pruebas ejecutadas, etc.

ABAP Unit

ABAP Unit[38] es una herramienta integrada en el entorno de ejecución de ABAP[39], el entorno de desarrollo de SAP[40]. Puede ser utilizada para ejecutar pruebas unitarias. Esta herramienta fue mencionada por ser para aplicaciones ABAP y este último ser un ambiente de desarrollo dirigido por modelos.

PSUnit

PSUnit[41] es una herramienta de pruebas unitarias basada en xUnit para PeopleSoft[42]. Es una colección de objetos PeopleTools administrados (páginas, componentes, PeopleCode, etc.) que facilita la generación y ejecución de pruebas. PeopleSoft también posee un ambiente de desarrollo dirigido por modelos, por lo que PSUnit también es un buen ejemplo de herramienta de pruebas unitarias relevante para el proyecto.

ProUnit

ProUnit[43] es un framework para la creación de pruebas unitarias para la plataforma Progress 4GL[44]. Fomenta la creación de componentes durante el proceso de desarrollo y permite probarlas de forma continua por lo que cualquier error puede ser detectado de forma rápida y profunda. ProUnit es un ejemplo de una herramienta sobre un lenguaje procedural similar a GeneXus.

EasyMock y JMock

EasyMock[45] y JMock[46] son librerías Java que ofrecen la creación de objetos *mock* para poder utilizarse en las pruebas unitarias. Su intención es soportar "test-driven development" [25] mediante el uso de objetos *mock* para ayudar a diseñar y probar las interacciones entre objetos del sistema. Pueden verse como una extensión de JUnit.

DbUnit

Otra herramienta oportuna de mencionar es DbUnit[47] ya que está orientada al manejo de bases de datos y el trabajo con GeneXus implica un gran uso de base de datos, por lo que estudiar esta herramienta puede proporcionar buenas ideas a la hora de diseñar un framework para pruebas en GeneXus.

Esta herramienta es una extensión de JUnit (también usable con Ant) dirigida a proyectos fuertemente asociados a bases de datos que, entre otras cosas, deja la base en un estado conocido entre diversas ejecuciones de pruebas. Es una excelente manera de evitar el vasto conjunto de problemas que pueden ocurrir cuando una prueba corrompe la base de datos y causa que las siguientes pruebas fallen o incrementen el problema.

Robot framework

Robot framework[48] es un framework genérico de Pruebas dirigidas por palabras clave (Keyword-Driven Testing[2]) para pruebas de aceptación y Aceptance Test-Driven Development (ATDD). Tiene una sintaxis basada en tablas, para crear casos de prueba de forma sencilla. Además, su funcionalidad puede ser extendida por librerías desarrolladas en Python o en Java. Los usuarios también pueden crear nuevas *keywords* desde las existentes usando la misma sintaxis simple que se utiliza para crear casos de prueba.

2- Por definición ver glosario.

3.3. *Características de las herramientas*

Herramienta	JUnit	Mockrunner	VS 2010	ABAPUnit
Licenciamiento	Gratuito	Gratuito	Pago	Pago
Lenguaje	Java	Java	.NET	ABAP
Assertions	Si	Si	Si	Si
Reportes generados	Muestran resultados: *Pass* o *Fail*. Indica que *assertion* o excepción falló	Muestran resultados: *Pass* o *Fail*. Indica que *assertion* o excepción falló	Igual que JUnit con reportes extra con porcentaje de pruebas que pasaron, fallaron, etc.	Igual que JUnit con manejo de tipo de error con distintos grados de información
Herramientas complementarias	Mockrunner, EasyMock, JMock, DbUnit	-	PEX, Moles	-
Grado de uso en la industria (*)	Muy usado	Usado	Muy usado	Usado

Herramienta	PSUnit	ProUnit	PBUnit	GXUnit 2007
Licenciamiento	Pago	Pago	Gratuito	Gratuito
Lenguaje	PeopleSoft	Progress 4GL	PowerBuilder	GeneXus
Assertions	Si	Si	Si	No
Reportes generados	Muestran resultados: *Pass* o *Fail*. Indica que *assertion* o excepción falló	Muestran resultados: *Pass* o *Fail* y mensaje de si se llamó de forma específica al método *fail*.	Muestran resultados: *Pass* o *Fail*. Indica que *assertion* o excepción falló	Muestran resultados: *Pass* o *Fail*
Herramientas complementarias	--	--	--	--
Grado de uso en la industria (*)	Poco usado	Poco usado	Poco usado	No se usa

(*) Como indicador del grado de uso en la industria fue utilizada la herramienta Google Trends[49] y la cantidad de resultados utilizando el buscador de Google. Se especificó una escala de la siguiente manera:

Muy usado: más de 100.000 resultados
Usado: entre 1.000 y 99.999 resultados
Poco usado: entre 1 y 999 resultados

4. Unidad de prueba

Al hablar de pruebas unitarias lo primero que se debe analizar es qué considerar como una unidad de prueba. En esta sección se analizan las componentes a considerar como unidad en dos de los lenguajes que se estudiaron y finalmente se eligen cuáles unidades se incluirán dentro del alcance del proyecto.

4.1. *Unidad en Power Builder*

En esta sección se plantea la discusión de lo que se puede considera una unidad de prueba en Power Builder según nuestra investigación de la herramienta.

Power Builder es un lenguaje Orientado a Objetos, cada componente creado es un objeto con propiedades, eventos y funciones. Según nuestra experiencia con la herramienta de pruebas unitarias PBUnit y viendo ejemplos concretos de pruebas observamos que las pruebas están enfocadas en las funciones de los objetos, en sus eventos y no en el objeto como un todo. Si bien lo más elemental de estos objetos son las propiedades no aporta probar una propiedad ya que no se generan acciones a partir de ella. Por ejemplo:

```
//definimos la propiedad "Importe"

Private Decimal Importe;

/*        of_getImporte (): decimal        */

/* definimos la función que retorna el valor de la propiedad*/

return   Importe;
```

No tiene sentido probar una propiedad, pero si las funciones y eventos que la utilizan. Luego de la ejecución de cada función se compara el estado del objeto o el valor de retorno de la función con el resultado esperado. Podemos decir que los eventos son funciones, pero con un momento de ejecución predefinido, por ejemplo un "*click*". Para probar los eventos se deben ejecutar explícitamente y no esperar que suceda la acción que lo dispara, luego hay que poder comparar el estado del objeto con el valor esperado.

Por lo tanto podemos ver que en Power Builder, la unidad de prueba es una función o evento, que utiliza los valores de las propiedades para comparar y retornar un resultado.

Utilizando este concepto de unidad, PBUnit permite probar todos los objetos generados, brindando gran versatilidad puesto que pueden diseñarse pruebas sobre un objeto tan detalladamente y exhaustivamente como se desee.

4.2. *Unidad en GeneXus*

A efectos de la implementación de GXUnit surge la siguiente incógnita: ¿Qué considerar unidad en GeneXus? Podríamos considerarla como un objeto, una subrutina, un evento, una regla, una propiedad. En esta sección analizaremos el concepto de "unidad de prueba" en GeneXus, comparándolo con otro lenguajes de desarrollo y validando el concepto de unidad elegido estudiando los distintos objetos GeneXus.

En el paradigma de orientación a objetos, la unidad de trabajo se refiere a una clase, un método

de una clase o un conjunto pequeño de clases (class-clusters)[50]. En el caso de GeneXus, definimos la "unidad" GeneXus como un objeto GeneXus completo, las subrutinas, propiedades, eventos son probadas utilizando las pruebas unitarias del objeto.

¿Por qué no considerar unidad por ejemplo una subrutina? Nuestra experiencia como desarrolladores GeneXus nos enseña que lo más relevante a la hora de realizar pruebas en GeneXus, no es el hecho de probar una subrutina o una parte del objeto, sino que lo más importante es probar el objeto en su totalidad. Generalmente, las funcionalidades implementadas mediante la utilización de subrutinas, reglas, propiedades, etc., suelen no ser críticas. En el caso de las subrutinas y reglas complejas, pueden ser implementadas como objetos Procedimiento. Además, tanto las reglas como las propiedades pueden ser probadas aunque seleccionemos el objeto como unidad.

Aunque se defina de esta manera la "unidad", no todos los objetos GeneXus son elegidos para probarlos de forma unitaria, sino que solo se eligen aquellos cuyas pruebas tengan sentido (ej: Procedure, Transaction, Web Panel). A continuación se presentan los criterios utilizados para la elección.

Funcionalidad:

Como primer criterio, se puede considerar que para que tenga sentido probar un objeto, el mismo debe aportar alguna funcionalidad a las aplicaciones desarrolladas por GeneXus. Por ejemplo, un objeto de tipo Theme, no aporta funcionalidad alguna, sino que simplemente sirve para encapsular ciertas propiedades de diseño de la aplicación. Sin embargo, un objeto de tipo Procedure puede agregar una variedad de funcionalidades a la aplicación, por ejemplo, insertar, borrar o modificar datos, registros de la base de datos, realizar cálculos de distintos tipos, etc.

Otros tipos de objetos GeneXus que no aportan funcionalidad, son los tipos de objetos que se utilizan para definir estructuras de datos (Table, Data View, Domain, Structured Data Type, Subtype Group), estos objetos son utilizados por GeneXus para determinar la estructura o relación de los datos, pero no poseen funcionalidad o comportamiento que necesite ser probado.

Las instancias de Patrones no ameritan pruebas unitarias, puesto que las mismas en realidad, tienen como única finalidad encapsular lógica para aplicar a los objetos de tipo Transacción determinados patrones de diseño, y generar otros objetos que implementan dicha lógica. Este grupo de objetos podrían ser de interés para probar unitariamente, pero esto no es compatible con el concepto de pruebas unitarias. Por lo tanto, lo que realmente sería interesante es probar los objetos generados por el patrón.

Independencia:

Con los objetos de tipo Attribute sucede algo interesante, porque si bien puede ser importante permitir probar la unidad a los que son de tipo fórmula, la prueba carece de sentido si estos atributos no están ligados a un objeto de tipo Transacción. Por lo tanto, consideramos que no tiene sentido probar los atributos de forma aislada, sino que probarlos en contextos de interés, por ejemplo, en las reglas de una Transacción, en un Procedimiento que utilice el atributo, etc.

External Objects

Debido al estudio de las KBs que realizamos (donde sus ocurrencias son mínimas), las reuniones con expertos en el tema (en donde se reconoció que no es de los objetos más relevantes), y el hecho de que no estén desarrollados en GeneXus, concluimos que los External Objects no iban a ser incluidos en el primer alcance de GXUnit. Aunque si es de interés que la herramienta los soporte en versiones futuras, puesto que sería útil por ejemplo, para mantener documentación dentro de la misma KB sobre los External Objects.

Conjunto de Objetos:

Puede considerarse un pequeño conjunto de objetos GeneXus como una unidad, por ejemplo, en el caso en que un objeto llame a otro tan simple que no tenga sentido usar un *mock* para simularlo, sino que sea viable utilizar el objeto real. Por lo tanto, una unidad en GeneXus puede ser un único objeto o un pequeño conjunto de objetos que juntos implementen cierta componente unitaria.

En la primera etapa del desarrollo de GXUnit, no fueron implementados generadores de *Mocks*, por lo que en caso de probar un objeto y este referencie a otros objetos, se utilizaan los objetos referenciados para ejecutar las pruebas.

Considerando las características estudiadas anteriormente, los objetos que consideraremos unidad y que pueden probarse de forma unitaria son:

- Data Provider
- Data Selector
- Procedure
- Query
- Transaction
- Web Panel
- Work Panel

4.3. *Objetos a ser probados de forma unitaria*

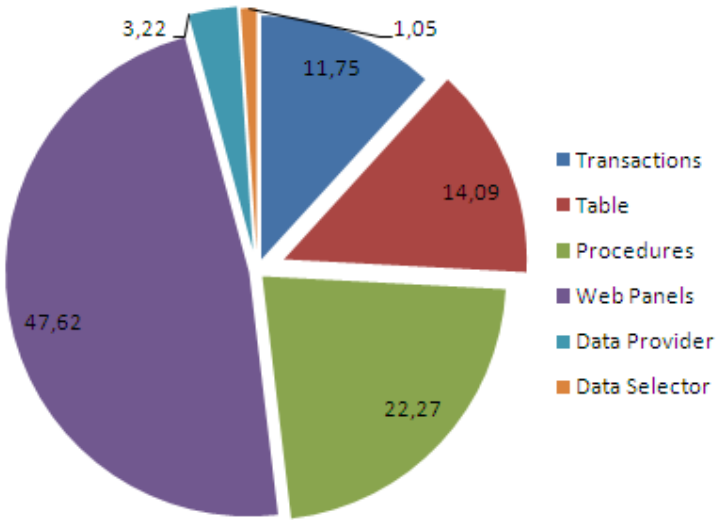
En GeneXus existen múltiples objetos que permiten al programador desarrollar los programas que desea. En esta sección, se analizan los objetos y su impacto en el desarrollo de aplicaciones GeneXus. De este análisis se obtienen los objetos GeneXus que se incluyen dentro de GXUnit como objetos a ser probados de forma unitaria, para poder definir el alcance del proyecto.

Para el siguiente análisis se utilizaron 19 bases de conocimiento (KB) GeneXus, de las cuales, 4 son sistemas en producción, o sea, que se están utilizando actualmente y cuya composición es realmente compleja. Las restantes 15 son sistemas de mediano/pequeño porte, en su mayoría aplicaciones que muestran ejemplos acerca de las funcionalidades GeneXus. Al final de la sección se encuentran tablas comparativas con los números de cada tipo de objeto de las KBs analizadas.

Sobre el análisis de los datos al final de la sección, cabe mencionar que se basó en los objetos GeneXus que se consideraron relevantes para el desarrollo de GXUnit, y que a su vez resultan interesantes por sí mismos, por ejemplo, no se consideraron los objetos Attribute, porque estos objetos sólo son instanciados a través de las Transactions, Tables y DataViews, y por lo tanto no fueron considerados.

Cabe destacar también que el total de las KBs estudiadas fueron desarrolladas para entornos web, por lo que ni siquiera se consideraron los objetos que se utilizan para Windows. Esto se debe a que la tendencia es ,construir aplicaciones web. Para contrarrestar esta carencia, se decidió considerar a los Web Panels como Work Panels, porque se entiende que ambos cumplen una función similar, aunque no poseen las mismas restricciones ni estilos de programación. Los objetos Table, fueron considerados únicamente con el objetivo de mostrar su relevancia dentro de una KB.

En el gráfico siguiente, están los porcentajes promedio de cada objeto considerado relevante *(fig 4.3.1)*:



(fig 4.3.1 – Porcentaje de los objetos GeneXus)

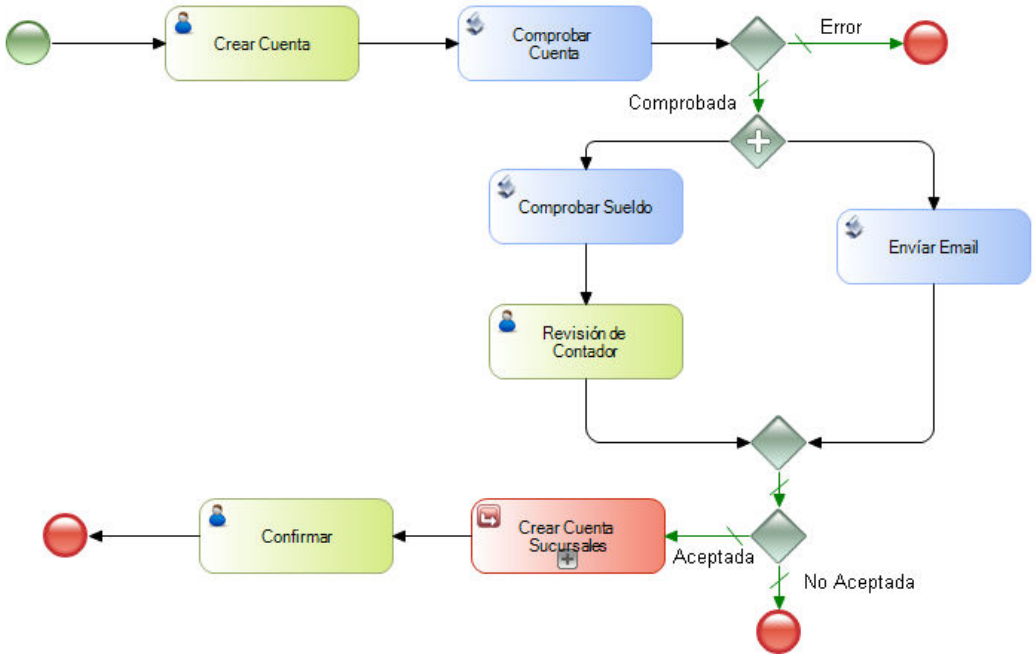
Como se puede ver, el objeto que parece tener más peso dentro de las KBs es el Web Panel, con casi la mitad de objetos (en promedio) dentro de las KBs El objeto que lo sigue es el Procedure, con un poco menos de la mitad de apariciones que el objeto Web Panel. En un segundo escalón vienen los objetos Transaction y por último están los objetos Data Provider y Data Selectors.

Sin embargo, aunque el objeto Web Panel es el que tiene (en promedio) más ocurrencias, se

considera que el objeto Procedure es el más relevante, o al menos tiene la misma relevancia, en cuanto al desarrollo de la herramienta GXUnit. Esta conclusión surge de la convicción de que la lógica de un Procedure es más completa y rica para analizar con el objetivo de desarrollar GXUnit, si bien es muy necesario considerar para el desarrollo de GXUnit el manejo de eventos en los Web Panels. Por ejemplo, un Web Panel no puede modificar la base de datos, mientras que un Procedure puede tanto seleccionar registros de la base de datos como modificarlos, en este sentido es más compleja la lógica de un Procedure, puesto que por ejemplo, hay que buscar formas de simular la base de datos, ver cómo se comportan las unidades de trabajo lógicas (UTLs), etc.

Hay que hablar además de los objetos "Business Process Diagram" (BPD), que pueden llegar a ser muy interesantes para probar como unidad. Los BPD son diagramas que modelan procesos de workflow, en los mismos, pueden aparecer tanto Procedures como Web Panels, y a su vez, maneja objetos internos como condicionales, etc. Por lo tanto, resulta de interés probar ciertos caminos críticos del proceso ante eventuales entradas. En los datos recabados no se encontraron diagramas de proceso, o se encontraron muy pocos (de uno a cuatro) en tres de las KBs

Ejemplo de un diagrama de proceso GeneXus *(fig 4.3.2)*:



(fig 4.3.2 – Ejemplo de Diagrama de Proceso de GeneXus)

Por ejemplo, en este caso podría ser de interés ver que si se comienza el proceso y se ingresa un cliente que ya posee una cuenta en el sistema, el proceso termina dando un error acorde a lo que se pretendía.

Este es el tipo de pruebas unitarias a las que nos referimos cuando mencionamos los BPD. Sería necesario permitir al usuario definir los datos de entrada del sistema así como también las acciones en las tareas (Task) de usuario (son las de color verde, ej.: Crear Cuenta), y por último, cuál sería el flujo que debería seguir el proceso, para verificar si se cumple o no.

Estas pruebas podrían a su vez, apoyarse en casos de pruebas unitarios definidos para los objetos que componen el BPD, pero este aspecto ya escapa al alcance de las pruebas unitarias y

por lo tanto al alcance del Proyecto de Grado.

A futuro, podría extenderse GXUnit para este tipo de objetos, puesto que está siendo cada día más utilizado por los desarrolladores y como ya dijimos pueden encontrarse varios casos interesantes de probar utilizando flujos, otro caso interesante por ejemplo, puede ser simular flujos sobre objetos BPD a efectos de probar un solo objeto unidad perteneciente al mismo.

Datos relevantes según estudio para el desarrollo de GXUnit:

Objetos	Importancia	Puntos Interesantes	Pruebas en la Actualidad
Attribute	Muy Baja	Fórmulas	Sólo se evalúan algunos casos a través de Web Panels por ejemplo
Business Process Diagram	Baja	- Flujos del Proceso - Actualizaciones en la base de datos	Suele probarse a través de código agregado por el desarrollador utilizando parámetros. En general es muy difícil de mantener y no es muy utilizado para grandes diagramas
Category	-		
Data Provider	Media	Resultado generado	Se prueba en ejecución junto con otros objetos, por ejemplo, se lo llama utilizando un Web Panel
Data Selector	Media	Tablas Navegadas	Igual que el Data Provider
Data View	-	Puede ser de interés verificar el rendimiento al utilizar este tipo de objetos	
Diagram	-		
Document	-		
Domain	-		
External Object	-	Son una interfaz para que GeneXus utilice programas externos desarrollados en lenguajes soportados por GeneXus.	Son llamados desde algún objeto GeneXus
File	-		
Folder	-		
Image	-		
Language	-		

Objetos	Importancia	Puntos Interesantes	Pruebas en la Actualidad
Master Page	Media	Por lo general no muestran datos, se utiliza para dar seguridad y formato a un grupo de formularios	Se ejecutan algunos casos interesantes
Menu	Baja	Se utiliza para dar opciones al usuario o redireccionar a formularios	No hay datos
Menubar	Baja	Ídem que Menu	No hay datos
Patterns	-	WorkWith, Auditing	
Procedure	Muy Alta	- Actualiza la base de datos - Genera reportes - Agrega contenido en respuestas HTTP - Interactúa con el sistema operativo - Carga SDTs - Lee la base de datos - Implementan Web Services	Se ejecuta enviando mensajes a algún log de la aplicación Se ejecuta utilizando el modo Debug de GeneXus
Query	-		
Structured Data Type	-		
Subtype Group	-		
Styles	-		
Table	-		
Theme	-		

Objetos	Importancia	Puntos Interesantes	Pruebas en la Actualidad
Transaction	Muy Alta	- Resulta interesante comprobar que las reglas se ejecuten en el evento que se desea porque cuando un objeto Transaction tiene muchas reglas, el código se complica, y si se cambia algo, se puede deshacer algo que ya existía - Tienen eventos al igual que los Web Panel, porque toda Transaction viene con un formulario asociado - Hay que tener en cuenta el evento "After Trn", es un evento pre definido que existe en las Transactions	Se realizan pruebas por parte del desarrollador ejecutando la aplicación y se observa si cumple con los requerimientos Muchas veces, cuando se desarrolla se corrigen errores pero se generan nuevos, porque el desarrollador no prueba todos los casos problema
Web Component	Alta	- Se pueden agregar a los Web Panels - Tienen un comportamiento casi idéntico a los Web Panels	Se prueban igual que los Web Panel
Web Panel	Muy Alta	- Programación orientada a eventos - Sesiones web - Carga registros de la base de datos - Puede estar contenido en una Master Page - Pueden contener User Controls	Son ejecutadas por el desarrollador para casos que resulten de interés Pueden enviar mensajes a un log o ejecutar el objeto en el modo Debug de GeneXus
Work Panel	Alta	Es parecido a los Web Panels, con la diferencia que es para Windows	

	Objetos						
Knowledge Base	Transaction	Table	Procedures	Web Panels	Data Provider	Data Selector	Total
Billing System	10	12	21	52	2	1	98
Community Life	6	7	12	33	1	0	59
CursoPemex	11	12	21	42	3	2	91
CursoPractico	10	12	18	62	14	1	117
Ejemplo Universidad Aperez	11	13	15	64	11	0	114
Fenix	13	13	20	76	2	0	124
GXWiki	12	13	140	73	30	27	295
KBLibreria	3	5	0	8	0	0	16
PeopleAnd Organizations	24	27	13	107	0	0	171
Reserva Salones	4	5	0	6	0	0	15
ServicioDeSalud	11	8	13	23	1	1	57
TG KB Politica Comercial	14	41	8	141	1	0	205
TG Portal Medico LSZ	8	7	11	18	1	1	46
Travel Agency	7	10	12	53	3	1	86
Universidades	9	11	13	58	10	1	102
Kb Produccion 1	131	135	1203	433	33	7	1942
Kb Produccion 2	99	100	751	587	7	4	1548
Kb Produccion 3	43	52	99	130	2	0	326
Kb Produccion 4	62	65	212	184	1	0	524

	Transactions	Table	Procedures	Web Panels	Data Provider	Data Selector
Porcentajes de Objetos dentro de cada KB	10,2	12,24	21,43	53,06	2,04	1,02
	10,17	11,86	20,34	55,93	1,69	0
	12,09	13,19	23,08	46,15	3,3	2,2
	8,55	10,26	15,38	52,99	11,97	0,85
	9,65	11,4	13,16	56,14	9,65	0
	10,48	10,48	16,13	61,29	1,61	0
	4,07	4,41	47,46	24,75	10,17	9,15
	18,75	31,25	0	50	0	0
	14,04	15,79	7,6	62,57	0	0
	26,67	33,33	0	40	0	0
	19,3	14,04	22,81	40,35	1,75	1,75
	6,83	20	3,9	68,78	0,49	0
	17,39	15,22	23,91	39,13	2,17	2,17
	8,14	11,63	13,95	61,63	3,49	1,16
	8,82	10,78	12,75	56,86	9,8	0,98
	6,75	6,95	61,95	22,3	1,7	0,36
	6,4	6,46	48,51	37,92	0,45	0,26
	13,19	15,95	30,37	39,88	0,61	0
	11,83	12,4	40,46	35,11	0,19	0
Totales	**11,75**	**14,09**	**22,27**	**47,62**	**3,22**	**1,05**

Otra razón que aumenta la relevancia de los objetos Procedure, es que en las KBs de producción, que son además las que tienen más objetos, predominan este tipo de objeto.

Por lo general, toda la lógica compleja, o lo que se desea sea reutilizable se implementa en un objeto Procedure, ya que los mismos puede ser llamados desde otros objetos del mismo tipo, Web Panels, Work Panels, Data Providers, Attribute (desde las fórmulas), reglas en las Transactions, etc.

Conclusiones

En base a este análisis se comenzó la implementación de GXUnit para los objetos de tipo Procedure, ya que todos los desafíos que se encontraron en su desarrollo generaron conocimiento que fue utilizado para implementar el resto de los objetos (Transacciones, Data Providers) en GXUnit.

Decidimos no incluir los WebPanels ya que difiere de los demás objetos incluidos en GXUnit por su necesidad de interactuar con el usuario. Además, los WebPanels pueden probarse utilizando la herramienta GXtest. La programación "procedural" de los objetos Procedure puede facilitar la implementación de GXUnit porque resulta más simple e intuitiva la programación para generar código que ayude a probar el objeto. Hay que destacar que nuestra experiencia en el desarrollo de aplicaciones GeneXus nos indica que generalmente los objetos con lógica más compleja dentro de las KBs son los de tipo Procedure.

Es importante resaltar la importancia de los objetos de tipo Data Provider, puesto que cada vez se usan más y para implementar aspectos claves del sistema. Su programación declarativa los hacen más atractivos para utilizar que un procedimiento y más fáciles de mantener.

La clave de la importancia de los Data Providers es que, como ya dijimos, generalmente se utilizan en lugares claves. Por ejemplo, puede ser utilizado para cargar un objeto de tipo SDT (tipo de datos estructurados, que permite manipular colecciones, etc.), cuyo objetivo sea determinar las páginas que está habilitado a navegar determinado usuario según la configuración de seguridad, si algún programador introduce un error en el objeto Data Provider, éste repercutiría sobre toda la aplicación y dejaría vulnerable al sistema. Por lo tanto, en este caso sería de mucha utilidad que el programador contara con sendas pruebas unitarias sobre el objeto para minimizar la introducción de errores al sistema.

En base a nuestro estudio concluimos que era mejor comenzar implementando GXUnit para generar pruebas unitarias sobre objetos de tipo Procedimiento, para luego agregar los objetos de tipo Transacción y Data Provider. Una vez que GXUnit permitió generar pruebas unitarias para estos tres objetos tipo, se continuó mejorando la herramienta para brindar mayores facilidades de uso.

5. Requerimientos de la herramienta

El 26 de octubre de 2010 se realizó una reunión en el Centro de Ensayo de Software donde se presentó el proyecto ante los principales involucrados en la herramienta desde su comienzo y usuarios de la comunidad interesados en su desarrollo.

En base a las conclusiones alcanzadas en la reunión, la investigación realizada sobre las herramientas de pruebas unitarias y en especial sobre una herramienta para el entorno de desarrollo GeneXus llegamos a los siguientes requerimientos a cumplir:

1- Embebida en la *IDE* GeneXus.

   GXUnit se encuentra completamente embebida dentro del *IDE* de GeneXus permitiendo al analista la utilización de la herramienta sin necesidad de ejecutar ninguna otra aplicación ni salir del ambiente que está utilizando.

2- Definición de Casos de Prueba asociados a objetos GeneXus.

   La herramienta permite crear casos de prueba asociados a un objeto GeneXus para probar de forma unitaria dicho objeto. Las pruebas se escriben en el propio lenguaje procedural de GeneXus permitiendo así inicializar y limpiar el contexto de la prueba, incluyendo llamadas a cualquier objeto GeneXus.

   La funcionalidad de los casos de prueba es similar a la de los objetos Procedimiento, lo cual permite no sólo tener toda la potencia de estos últimos, sino también independizar del lenguaje las pruebas, ya que el código para la ejecución de las pruebas será generado por GeneXus.

   En la reunión se concluyó que realizar pruebas para los objetos GeneXus Procedimiento, Transacción (Business Component) y Data Providers sería la prioridad debido a las tendencias actuales de desarrollo.

3- Métodos de tipo *Assert* para controlar los resultados de la llamada al objeto asociado.

   Se dispone de métodos del tipo *Assert* utilizados para comparar los resultados de una ejecución con los valores esperados para dicha ejecución. Estas llamadas permiten indicar qué objetos fallaron y cuáles no, al ejecutar pruebas unitarias con GXUnit.

4- Suites de prueba para encapsular conjuntos de casos de prueba.

   Los casos de prueba pueden ser agrupados mediante el uso de Suites. Un Suite puede a su vez contener un conjunto de Suites y uno o más casos de prueba facilitando la búsqueda y ejecución de las pruebas.

5- Ejecución de los Suites o casos de prueba.

   De manera sencilla pueden seleccionarse Suites enteros o casos de prueba y mediante un botón ejecutarlos, dando al desarrollador la posibilidad de probar los objetos que construye de manera simple y práctica.

6- Visualización de los resultados de los casos de prueba.

   Luego de una ejecución se permite visualizar el resultado obtenido informándose de los casos de prueba ejecutados, los que fallaron (identificando en cuál *Assert*) y los que cumplieron con lo esperado satisfactoriamente.

7- Visualización del histórico de pruebas que se han realizado.

   Se permite ver el histórico de pruebas realizadas, por fecha y hora de ejecución y al

seleccionar una de las ejecuciones puede verse su resultado tal como se mostró al finalizar la ejecución.

8- Exportar e Importar objetos de prueba.

Todos los objetos de GXUnit se pueden exportar e importar entre las distintas Bases de Conocimiento que tengan instalada la herramienta como cualquier otro objeto GeneXus.

Culminado el proyecto se tiene una herramienta de mucha utilidad para la comunidad debido a la importancia de las pruebas unitarias en el desarrollo de un producto de software en todas las etapas del mismo, especialmente en la de implementación que es donde se introducen más errores. Además, la herramienta es útil para un desarrollo basado en Test-Driven Development. Creamos un caso de prueba para validar si se cumple con un requerimiento, implementamos el objeto para que pase el caso de prueba y así sucesivamente hasta cumplir con lo deseado.

6. Desarrollo de GXUnit

El objetivo de esta sección es presentar los aspectos técnicos de GeneXus, que permitieron el desarrollo de GXUnit, así como también, comentar los puntos fuertes y débiles que posee la estrategia de desarrollo elegida.

Cuando se habla acerca de una herramienta para la verificación unitaria de software, se valora de forma positiva que la misma esté integrada al entorno de desarrollo utilizado para implementarlo, lo cual conlleva beneficios varios, que por lo general se traducen en un mejor y mayor uso de la herramienta de verificación.

Como ventajas más importantes de que la herramienta esté integrada podemos nombrar la practicidad y facilidad de uso de la misma. Cuando hablamos de practicidad, nos referimos a que el programador de GeneXus, no necesita salir del entorno de desarrollo para realizar las pruebas, hecho que por lo general molesta bastante y el programador termina por elegir no ejecutar las pruebas, o ejecutarlas con menor frecuencia. La facilidad de uso de la herramienta radica en que al estar integrada al entorno de desarrollo, minimiza las tareas de configuración por parte del desarrollador, para ejecutar las pruebas. La herramienta aprovecha las propiedades ya configuradas de la KB que se está utilizando, como por ejemplo, el lenguaje a generar y el manejador de base de datos.
Estas dos características ayudarán a que el usuario no necesite invertir mucho tiempo y esfuerzo para ejecutar las pruebas unitarias y pueda ejecutarlas con más frecuencia.

Como otras cualidades interesantes de las herramientas de verificación unitaria, se destaca la velocidad de ejecución de las pruebas unitarias, la posibilidad de guardar al menos los últimos resultados y obviamente, que las pruebas sean almacenables para poder ejecutarse en cualquier momento, sin necesidad de diseñarlas cada vez que se desea verificar un objeto.

Teniendo en cuenta estas características deseables para una herramienta de verificación unitaria, es que presentaremos a continuación cómo se desarrollará GXUnit para intentar maximizarlas.

6.1. *Cómo integrar GXUnit a GeneXus*

Desde la versión 10 de GeneXus, el entorno de desarrollo[51] brinda la facilidad de extenderlo. Este diseño permite agregar libremente módulos al programa para implementar las más diversas funcionalidades, los cuales pueden tener cualquier complejidad, desde módulos que incorporen una opción al menú, hasta módulos que generen código. En teoría, no existen límites en cuanto a las funcionalidades que podrían llegar a implementarse para extender GeneXus ya que en principio, no habría diferencia entre los módulos desarrollados por Artech y los desarrollados para implementar las nuevas funcionalidades.

Estos módulos o paquetes utilizados para extender GeneXus, deben estar programados en C#. Para permitir el desarrollo de extensiones por parte de los usuarios, Artech provee el GeneXus Platform SDK, que aporta las librerías necesarias para implementar las mencionadas extensiones. Acompañando las librerías, también vienen ejemplos y documentación sobre cómo extender GeneXus. El SDK también provee una herramienta para Visual Studio 2005 o superior, que permite comenzar proyectos de extensiones GeneXus con configuraciones predeterminadas, facilitando la implementación de las extensiones.

Además del material mencionado anteriormente, aportado por Artech, existen foros y wikis especializados sobre el tema extensiones. Sin lugar a dudas, un material importante fue aportado por las experiencias de los dos grupos del curso Proyecto de Ingeniería de Software, GXUnit1 y GXUnit2, que lograron desarrollar dos prototipos de GXUnit, que fueron utilizados como

ejemplos para el desarrollo de este nuevo GXUnit. Se estudió parte del código C# desarrollado para implementar determinadas características de GXUnit.

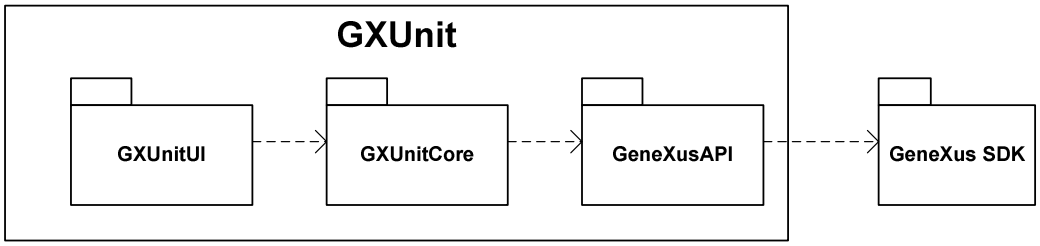
En resumen, utilizando el diseño de GeneXus que permite agregar funcionalidades al entorno de desarrollo a través de las extensiones, se desarrolló el módulo GXUnit, para que sea al mismo tiempo, práctico y amigable para el programador GeneXus, que desee mantener un conjunto de pruebas unitarias para verificar sus objetos GeneXus.

7. Arquitectura y diseño de la solución

GXUnit desde su comienzo fue pensado para seguir creciendo más allá del alcance de este proyecto, por lo que se decidió optar por un diseño que soporte este requerimiento.

La arquitectura comprende tres grandes componentes *(fig 7.1)*:

- GeneXus API
- GXUnit Core
- GXUnit UI



(fig 7.1 – Componentes GXUnit)

7.1. *GeneXus API*

El Subsistema GeneXus API recibe solicitudes de la capa superior "GXUnitCore" y las procesa utilizando las funcionalidades del SDK de GeneXus.

A continuación puede verse un diagrama de clases donde se muestra la integración de las principales clases de la componente *(fig 7.1.1)*.

Expone sus funcionalidades mediante un controlador façade que es consultado por la componente GXUnitCore.

Manejadores

Se definen Manejadores para cada objeto GeneXus utilizado. Estos manejadores utilizan los servicios que ofrece el SDK de GeneXus para cada objeto. Administran los objetos que manejan ofreciendo servicios de creación, modificación y borrado de los mismos. Los manejadores son: ManejadorResultado, ManejadorProcedimiento, ManejadorTransaccion, ManejadorFolder, ManejadorSDT, ManejadorDataProvider y ManejadorTestSuite.

Resultado

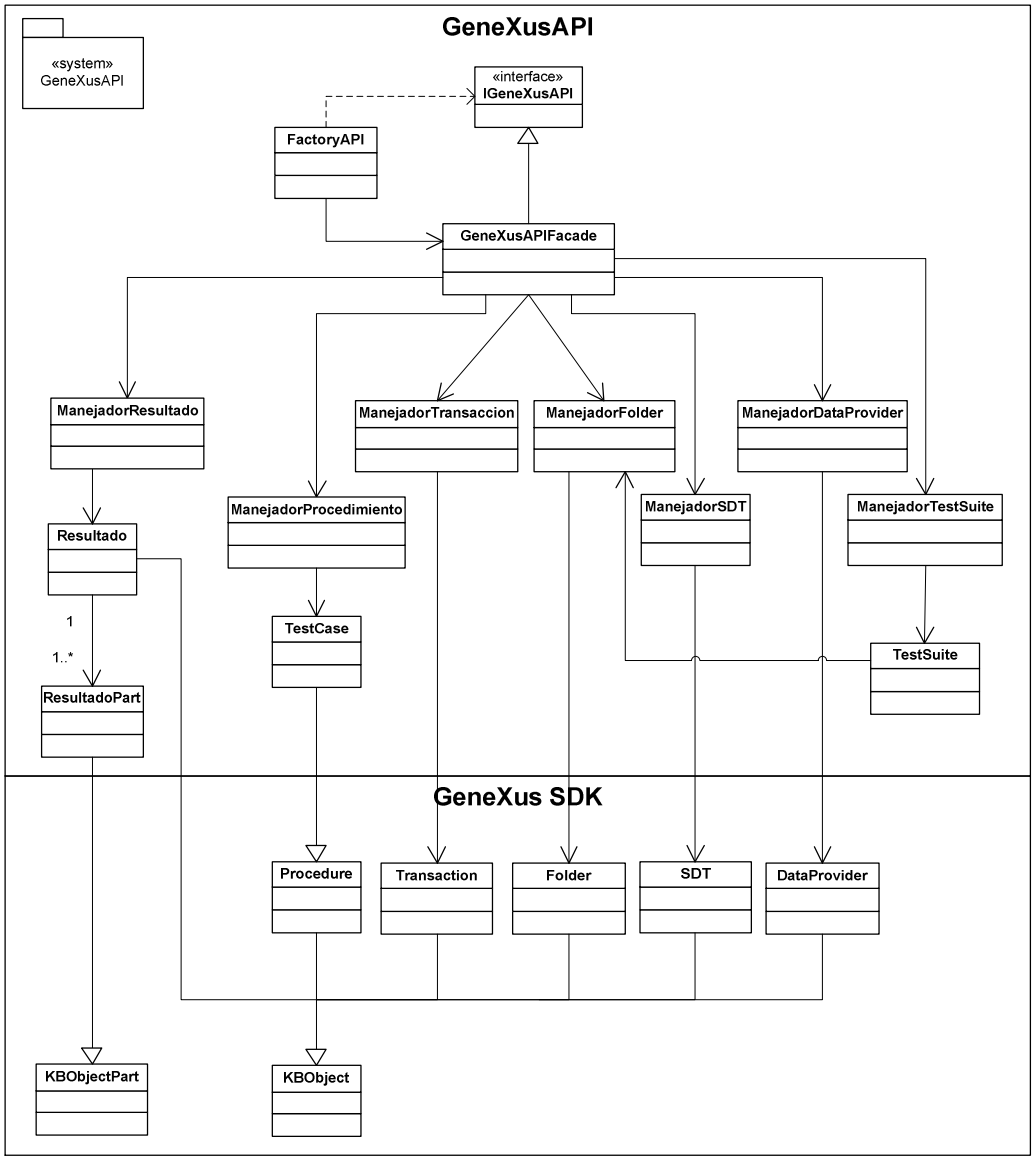
Se representa el resultado de una ejecución de un conjunto de pruebas mediante un nuevo tipo de objeto GeneXus llamado "Result". Este objeto guarda un XML que contiene la estructura de la ejecución de los distintos casos de prueba para luego poder representarlo gráficamente mediante una ventana de visualización. La clase Resultado hereda de KBObject y tiene asociada la parte ResultadoPart que hereda de KBObjectPart. En la clase ResultadoPart se guarda el XML del resultado de la prueba.

TestCase

El concepto de caso de prueba fue implementado definiendo un nuevo tipo de objeto GeneXus llamado "TestCase". Este objeto hereda de Procedure por lo que tiene muchas de sus propiedades, por ejemplo, puede utilizar tanto su definición de variables, como su editor de texto y el lenguaje estructurado que éste maneja. Esta elección permite abstraerse del lenguaje en que se está generando para la ejecución de las pruebas ya que se maneja como cualquier objeto GeneXus utilizando la interfaz que éste provee. Agrega dos propiedades a la lista que ya provee el procedimiento, la propiedad TestCase que siempre tiene el valor true y la propiedad ObjectToTest que contiene el nombre del objeto GeneXus que está probando el caso de prueba.

TestSuite

Para manejar el concepto de suites de prueba se utilizó el objeto Folder de GeneXus. Al crear una suite utilizando la herramienta se crea un Folder con el nombre seleccionado bajo la Folder/Suite elegida, o un Folder especial si se trata del primer nivel de suites. Se utiliza así la estructura de árbol del objeto Folder para la administración de las suites. Luego los objetos Test Case se crean dentro de estas Folders quedando agrupados dentro de la suite que representa. La clase tiene un atributo que indica cuál es la Folder GeneXus por la cual está representada.



(fig 7.1.1 – Clases de GeneXusAPI)

7.2. *GXUnitCore*

El Subsistema GXUnitCore *(fig 7.2.1)* es el núcleo de GXUnit, que es totalmente independiente del SDK de GeneXus. Define la lógica de la herramienta. Por ejemplo, crea la definición de un procedimiento de "*Assertion*" y solicita su creación al subsistema GeneXusAPI. Recibe solicitudes de la capa superior "GXUnitUI", las procesa y delega las funciones de comunicación e interacción con GeneXus al subsistema GeneXusAPI.

ManejadorRunner

Es la clase responsable de orquestar la definición y creación del Procedimiento Runner, cuyo objetivo es la ejecución de los casos de prueba. Su método principal crea el Procedimiento Runner a partir de los casos de prueba seleccionados para la ejecución, el siguiente paso es delegar la creación del objeto GeneXus Procedure correspondiente a la capa inferior GeneXusAPI. Otro método importante es el de ejecución del procedimiento Runner, luego de recibir la solicitud de la capa superior delega la orden a la capa inferior GeneXusAPI que ejecuta los procedimientos.

GXUnitInicializador

Es la clase que inicializa la herramienta, define los procedimientos *assert* y los SDTs necesarios para el funcionamiento. Los procedimientos *assert* se definen creando objetos de la clase Procedimiento que se explica más abajo. Para la definición de los objetos SDTs se crea un objeto del tipo SDTipo, que se explica también en esta sección. Luego delega la creación de los objetos GeneXus correspondientes a la capa inferior denominada GeneXusAPI.

DTDataProvider

Es la clase que modela un objeto Data Provider de GeneXus en el subsistema GXUnitCore.

DTTransaction

Esta clase representa el objeto Transaction de GeneXus en el subsistema GXUnitCore. Tiene las siguientes propiedades:

- Nombre: Es un String que representa el nombre del Objeto.
- Atributos: Es una colección del tipo DTAtributo utilizado para definir los atributos de la transacción.
- Propiedades: Es una colección del Tipo DTPropiedad la cual representa las Propiedades de la transacción, por ejemplo BUSSINES_COMPONENT.

DTAtributo

Representa un atributo de una transacción. Tiene las siguientes propiedades:

- Nombre: Define el nombre del atributo
- Tipo: Define el tipo del atributo.
- EsClave: Define si el atributo forma parte de la clave.
- EsSoloLectura: Define si el atributo es sólo de lectura.

DTTestCase

Representa el nuevo tipo de Objeto TestCase de GXUnit. Además de tener definidas las propiedades de la clase Procedimiento se le agregaron las siguientes:

- Objecttotest: Representa el nombre del objeto a ser probado por el caso de prueba.

Procedimiento

Su objetivo es modelar un Procedimiento GeneXus en el subsistema GXUnitCore, tiene las siguientes propiedades:

- Nombre - Es un String que define el nombre del Procedimiento.
- Source - Es un String que define el código fuente del Procedimiento.
- Rules - Es un String que define las reglas del Procedimeinto.
- Folder - Es un String que define la carpeta dentro de la Base de Conocimiento en que se almacena el Procedimiento.
- Variables - Es una colección del Tipo DTVariable la cual representa las variables del Procedimiento.
- Propiedades - Es una colección del Tipo DTPropiedad la cual representa las Propiedades del Procedimiento, por ejemplo CALL_PROTOCOL.

DTFolder

Es la clase que modela un objeto Folder de GeneXus en el subsistema GXUnitCore.

DTTestSuite

Es la clase que modela las Suites de Prueba en GXUnit.

SDTipo

Esta clase modela un Objeto SDT de GeneXus en el subsistema GXUnitCore. El SDT puede tener varios niveles y se representa con la clase **SDTipoNivel**, además tiene varios ítems que se representan con la clase **SDTipoNivelItem**.

La clase SDTipo tiene las siguientes propiedades:

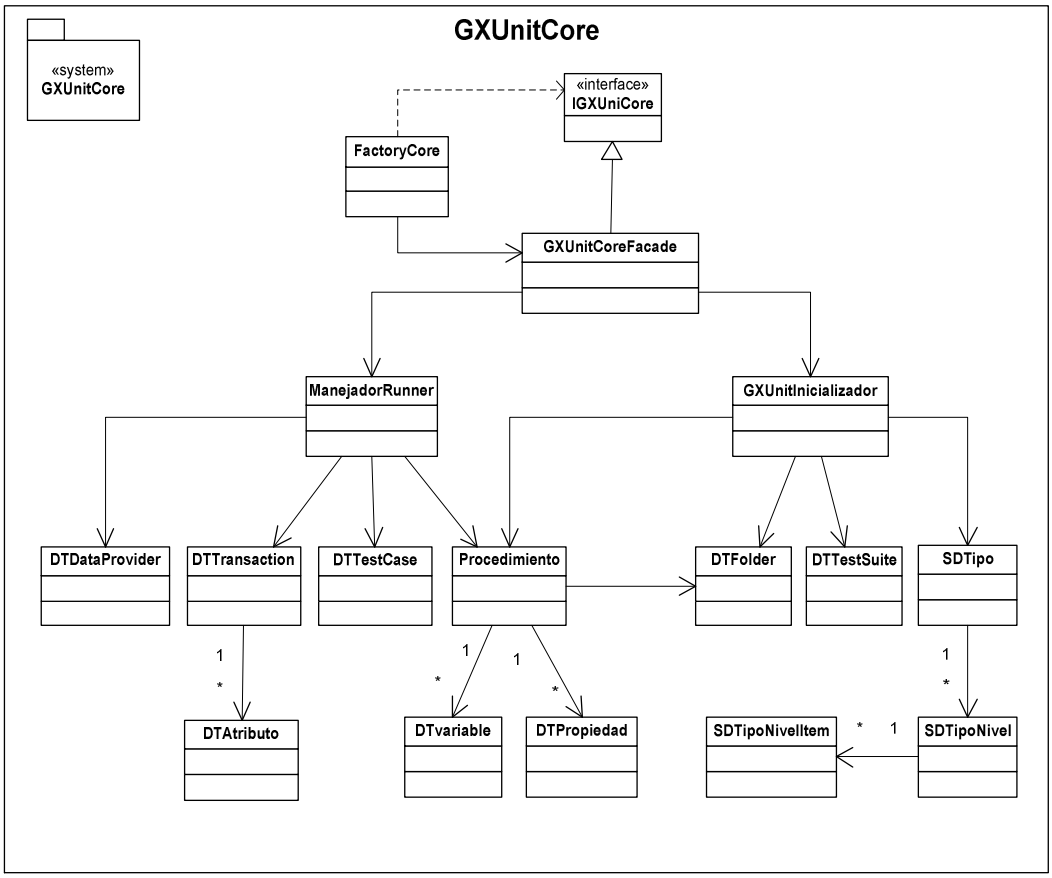
- Nombre - Es un String que define el nombre del SDT.
- Niveles - Representa el conjunto de niveles del SDT.
- Propiedades - Representa las propiedades del SDT.

La clase SDTipoNivel tiene las siguientes propiedades:

- Nombre - Es un String que define el nombre del Nivel.
- Items - Representa el conjunto de ítems del Nivel
- EsColeccion - Booleano que indica si el nivel es una colección.
- Niveles - Representa los niveles del Nivel.

La clase SDTipoNivelItem tiene las siguientes propiedades:

- Nombre - Es un String que define el nombre del Item.
- Tipo - Representa el tipo de dato del Item.



(fig 7.2.1 – Clases GXUnitCore)

7.3. *GXUnitUI*

El objetivo de esta componente es mantener la comunicación entre el usuario y la herramienta GXUnit. Recibe distintos comandos como Crear Test Case, Crear Test Suite, Ejecutar Test Cases, Generar Test Cases, etc. Muestra resultados de las ejecuciones de los casos de prueba al usuario. El objetivo de este subsistema es independizar las solicitudes del usuario del resto de la extensión, derivando las mismas a la capa siguiente, "GXUnitCore".

Para definir la arquitectura del sistema se tuvieron en cuenta los siguientes puntos:

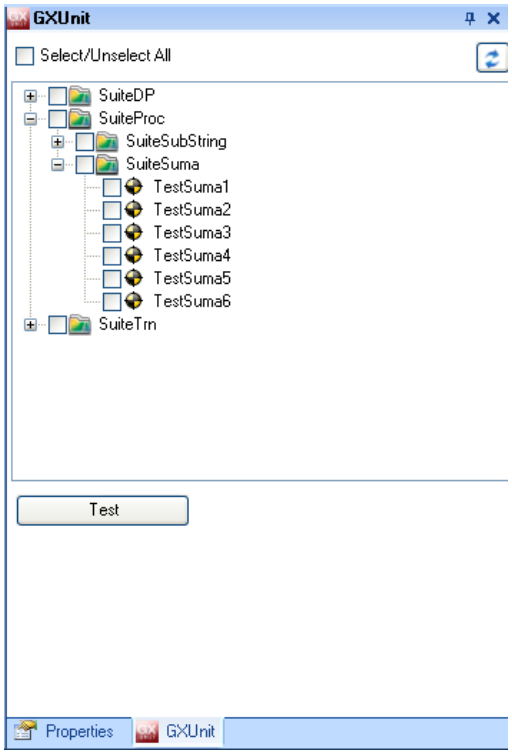
- Encapsular la comunicación con el SDK de GeneXus, definiendo una capa especializada en dicha comunicación con el objetivo de minimizar los impactos en el sistema tras futuras modificaciones del SDK. Esta decisión fue tomada tras recoger la experiencia de los grupos de Proyecto de Ingeniería de Software que trabajaron en la construcción de la primera versión de GXUnit. La lógica de sus implementaciones estaba acoplada al SDK de GeneXus, dificultando su mantenimiento al ser modificado.

- Facilitar la comprensión del funcionamiento de GXUnit y aislar la lógica de la herramienta de la comunicación con GeneXus. Maximizar la utilización de código GeneXus para definir y ejecutar los Casos de Prueba, lo cual asegura la compatibilidad hacia adelante con los distintos lenguajes que se puedan llegar a generar.

8. GXUnit en funcionamiento

GXUnit facilita la administración de Casos de prueba y Suites de prueba, su ejecución y la visualización de los resultados obtenidos.

Como se indicó en el capítulo "Desarrollo de GXUnit" para el desarrollo de GXUnit se utilizó el concepto de extensiones de GeneXus. Para la administración de Suites y Casos de prueba se creó una "Tool Window" que permite realizar diversas acciones *(fig 8.1)*.



(fig 8.1 – Tool Window GXUnit)

A continuación se describen las funcionalidades que ofrece la herramienta y cómo se utilizan.

8.1. *Métodos Assert*

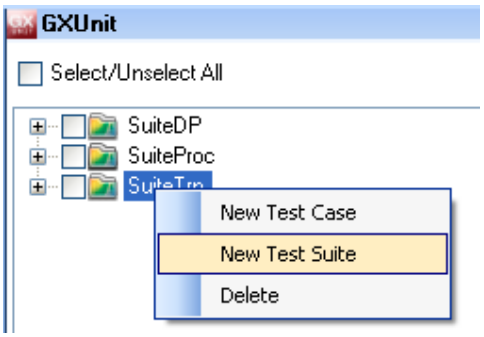
Los métodos de tipo *Assert* son utilizados para comparar los resultados de la ejecución de las unidades bajo prueba con los resultados esperados. En GXUnit, los métodos *Assert* fueron implementados como Procedimientos GeneXus que proveen por defecto dos métodos básicos, *AssertNumericEquals* y *AssertStringEquals*, para comparar variables numéricas y cadenas de caracteres respectivamente. Estos procedimientos son creados al inicializarse GXUnit y forman parte de la KB. Este conjunto de métodos combinado con programación básica en GeneXus, permite la comparación exacta de cualquier tipo de datos, por ejemplo, si queremos comparar dos variables de tipo SDT, nos alcanza con recorrer los elementos de cada variable y compararlos entre ellos.

Si bien GXUnit provee por defecto dos métodos básicos de *assert*, permite al usuario implementar fácilmente nuevos métodos. Los métodos provistos por GXUnit están desarrollados utilizando código GeneXus, por lo que cualquier usuario podría utilizar dicho código para crear sus propios métodos *assert*, cambiando únicamente el código donde se comparan los resultados.

Al ejecutarse uno de estos métodos se irán guardando los resultados para luego poder visualizar todos los resultados de la ejecución de los casos de prueba.

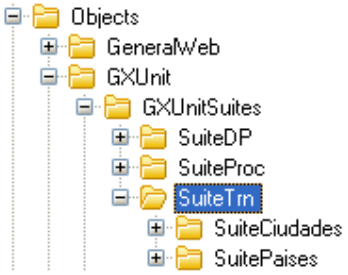
8.2. *Administración de Suites de prueba*

Como ya se explicó, una suite de pruebas tiene la funcionalidad de agrupar cierto conjunto de casos de prueba según el criterio del usuario. Para crear una nueva Suite de Pruebas se debe hacer *click* derecho sobre una suite ya creada *(fig 8.2.1)* o sobre cualquier lugar de la Tool Window GXUnit para crear una Suite del primer nivel.



(fig 8.2.1 – Creación de Suite desde la Tool Window GXUnit)

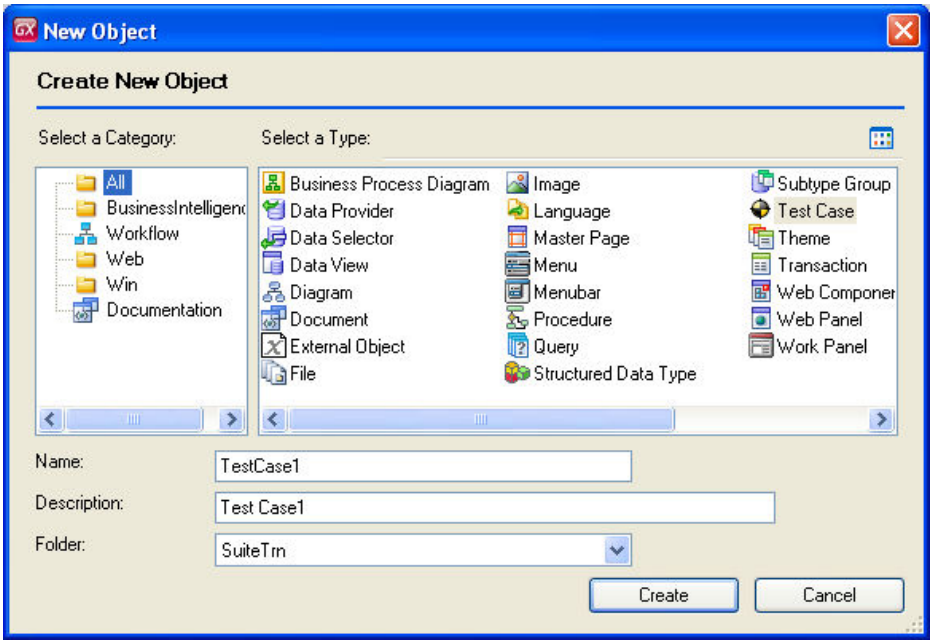
Luego de seleccionar Nueva Suite se pide el nombre de la misma y al ingresarlo se crea una nueva Folder GeneXus bajo la estructura Objects -> GXUnitSuites *(fig 8.2.2)*.



(fig 8.2.2 – Carpeta GeneXus creada por el objeto Suite)

8.3. *Administración de Casos de prueba*

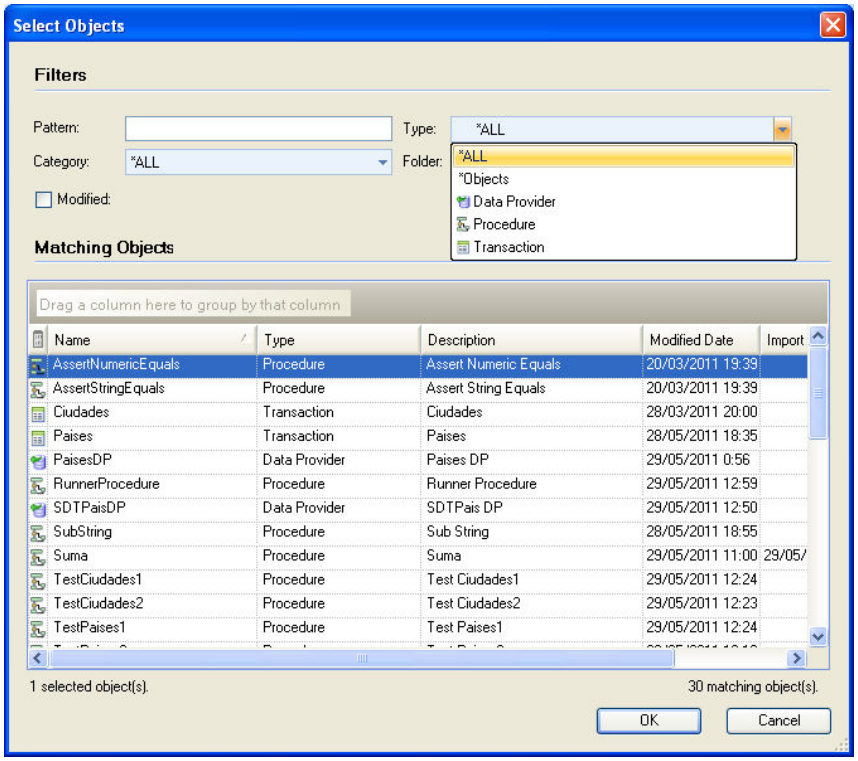
Para crear un nuevo Caso de Prueba, basta hacer *Click* derecho en una Suite y seleccionar "Nuevo TestCase". En el cuadro de dialogo de nuevo objeto GeneXus se debe seleccionar el objeto Test Case y como Folder queda seleccionada la Suite elegida *(fig 8.3.1)*.



(fig 8.3.1 – Selección de objeto al crear un nuevo Test Case)

Cabe aclarar que como mencionamos en el diseño, Test Case es un objeto GeneXus, por lo que podemos utilizar la interfaz de GeneXus para crear cualquier objeto. Por tanto, un TestCase puede crearse también desde "File" – "New" – "Object" (o Ctrl+N), pero debe elegirse manualmente la Suite (Folder GeneXus correspondiente).

Luego de seleccionar el nombre y suite del TestCase se selecciona el objeto a probar. Se permiten seleccionar objetos Procedure, DataProvider o Transaction *(fig 8.3.2)*.



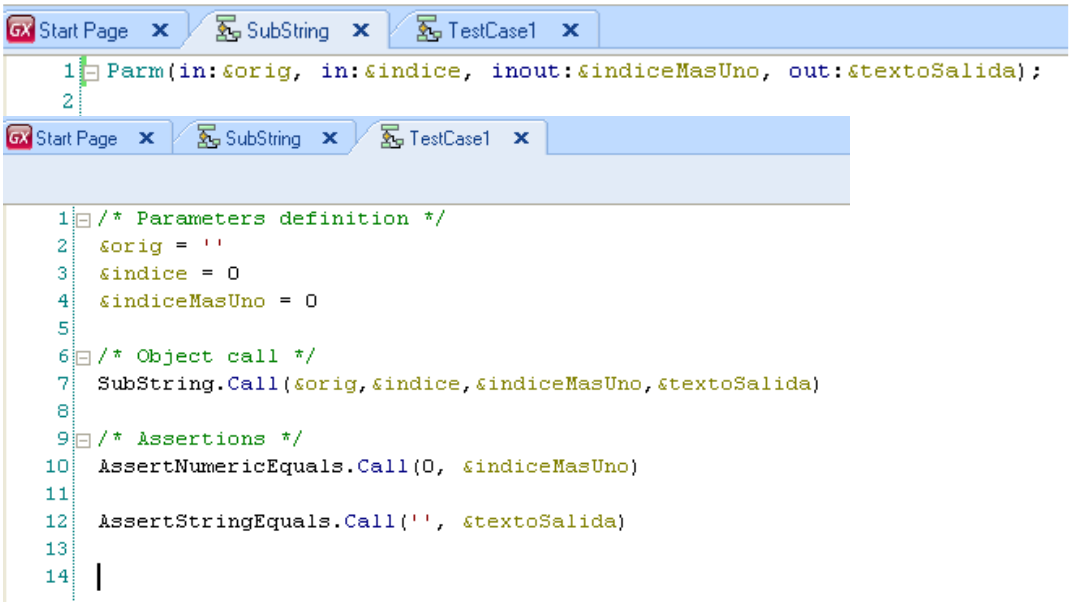
(fig 8.3.2 – Selección del objeto a probar)

El objeto Test Case tiene un editor que utiliza la sintaxis del objeto Procedimiento para ejecutar otro objeto GeneXus e implementar las comparaciones invocando a los objetos de tipo *Assert*.

Al crearse un caso de prueba se genera automáticamente código GeneXus por defecto, en el cual se llama al objeto a probar y se incluyen *asserts* para evaluar los resultados de la invocación. El usuario modifica este código para adaptarlo a sus necesidades.

Casos de prueba de procedimientos

En el caso de que el objeto asociado al caso de prueba sea un Procedimiento se crea una variable por cada parámetro que se encuentre en la regla "Parm" del procedimiento a probar y se procede según su tipo: si es de entrada o entrada/salida se inicializa con un valor por defecto y si es de salida o entrada/salida se evalúa con una llamada al *Assert* correspondiente *(fig 8.3.1)*.



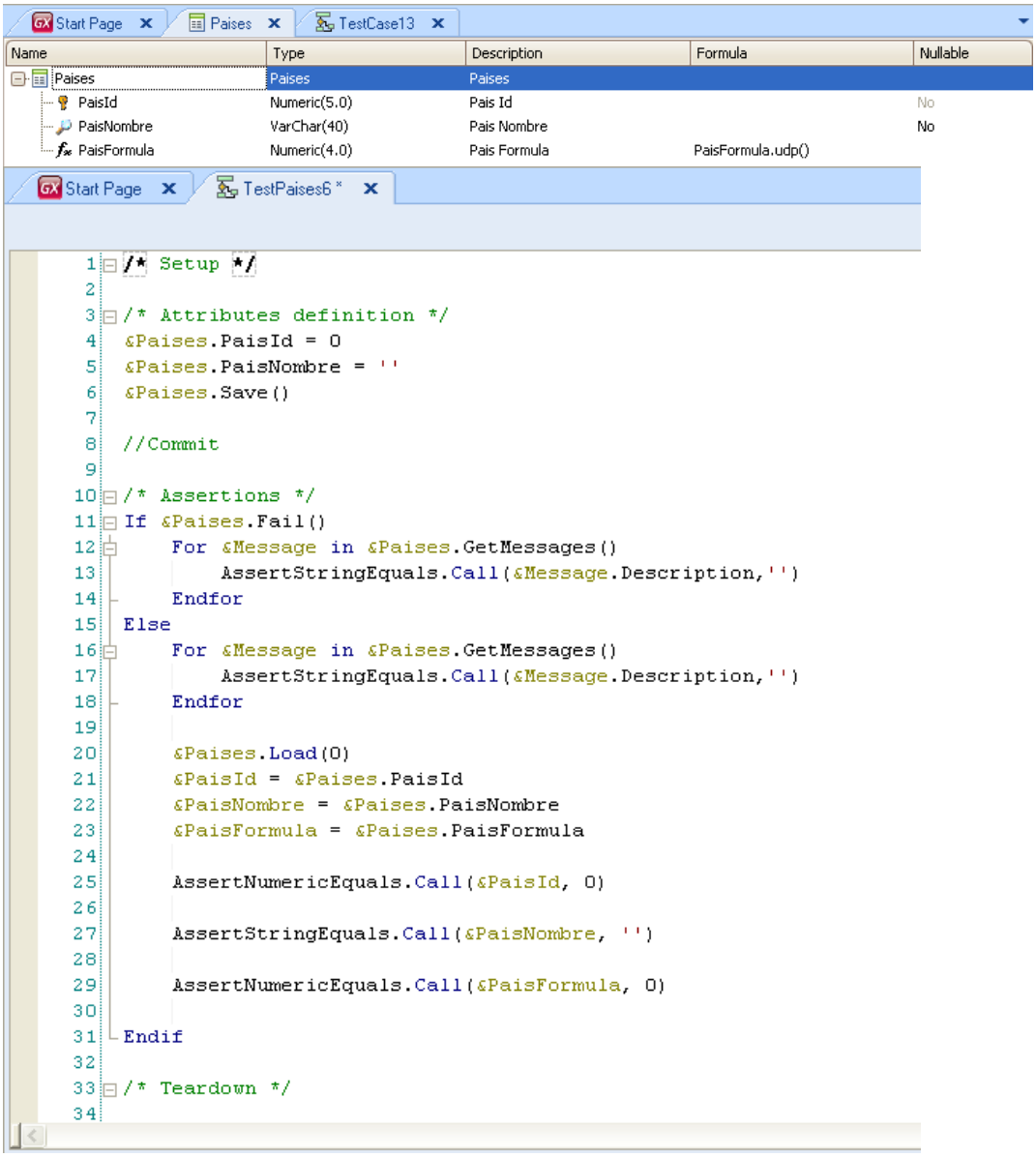
(fig 8.3.1 – Código por defecto al probar un objeto de tipo Procedure)

Casos de prueba de transacciones

Al crear un nuevo caso de prueba asociado a una transacción con la propiedad Business Component (BC) se genera código GeneXus por defecto de la siguiente manera *(fig 8.3.2)*:

- Se define una variable del tipo de la transacción BC
- Se inicializa con un valor por defecto todos los atributos (excepto los de tipo read only)
- Se guarda (*save* del BC)
- Se carga (*load* del BC)

- Se incluye un *assert* para cada mensaje de la Transacción
- Se incluye un *assert* por cada atributo del Business Component



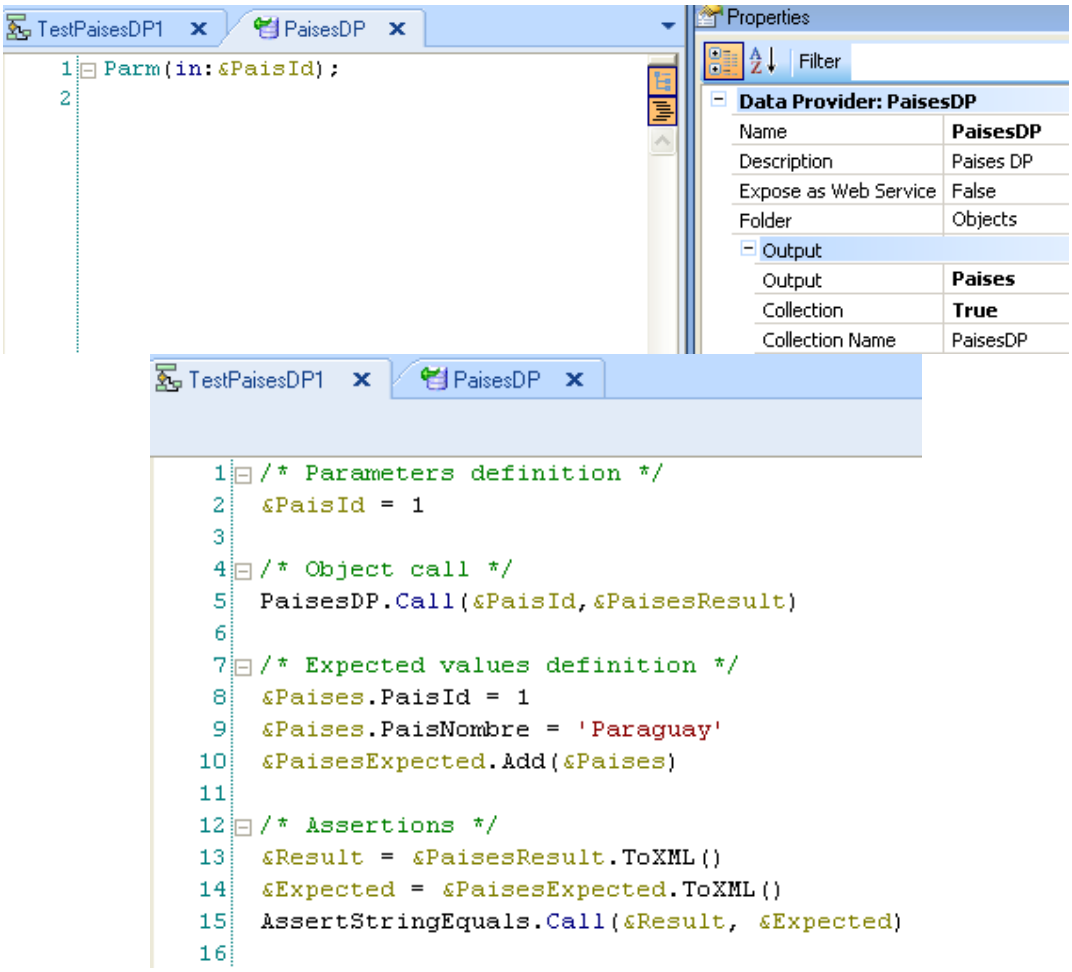
(fig 8.3.2 – Código por defecto de la prueba de un objeto de tipo Transaction)

Casos de prueba de Data Providers

Al crear un nuevo caso de prueba asociado a un Data Provider se genera código GeneXus por defecto de la siguiente manera *(fig 8.3.3)*:

- Se inicializan los parámetros de entrada del DP
- Se llama al DP
- Se inicializa el resultado esperado (BC o SDT)
- Se incluye un *AssertStringEquals* para comparar el Output esperado con el obtenido

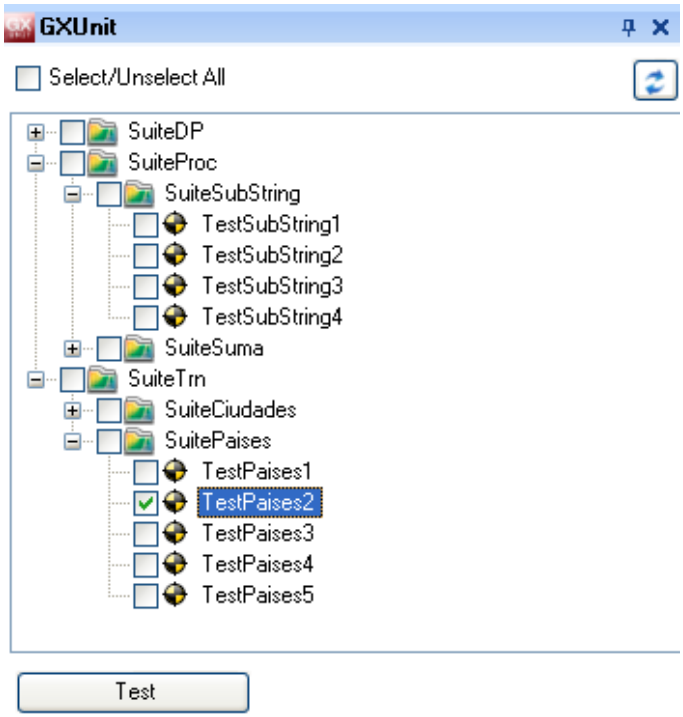
El output de un Data Provider puede ser un SDT o un BC, GXUnit automáticamente genera código por defecto adaptándose a esta variante. En cualquiera de los casos genera una variable GeneXus del tipo que corresponda e inicializa el resultado esperado también de la manera que corresponda.



(fig 8.3.3 – Código por defecto de la prueba de un objeto de tipo Data Provider)

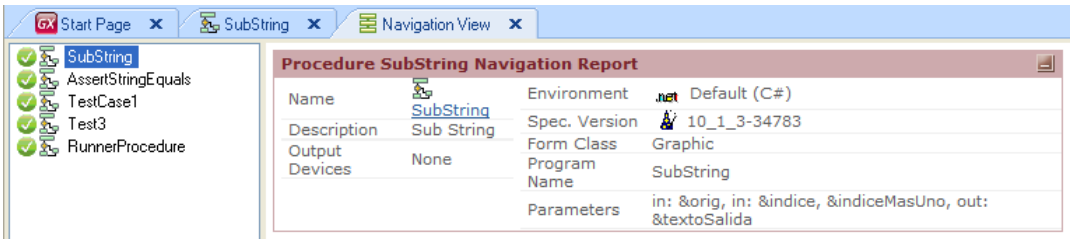
8.4. *Ejecución de las pruebas*

La ejecución de los casos de prueba se realiza desde el botón Test de la Tool Window, que contiene los conjuntos de Casos de Prueba agrupados en sus respectivas Suites *(fig 8.4.1)*.



(fig 8.4.1 – Selección de casos de prueba a ejecutar)

Seleccionando los casos de prueba a ejecutar (mediante su respectivo check box) al presionar el botón "Test", se crea un nuevo procedimiento (de nombre RunnerProcedure) que tiene las llamadas a todas las pruebas que se quieren hacer, se generan todos los objetos necesarios y luego se ejecutan *(fig 8.4.2)*.

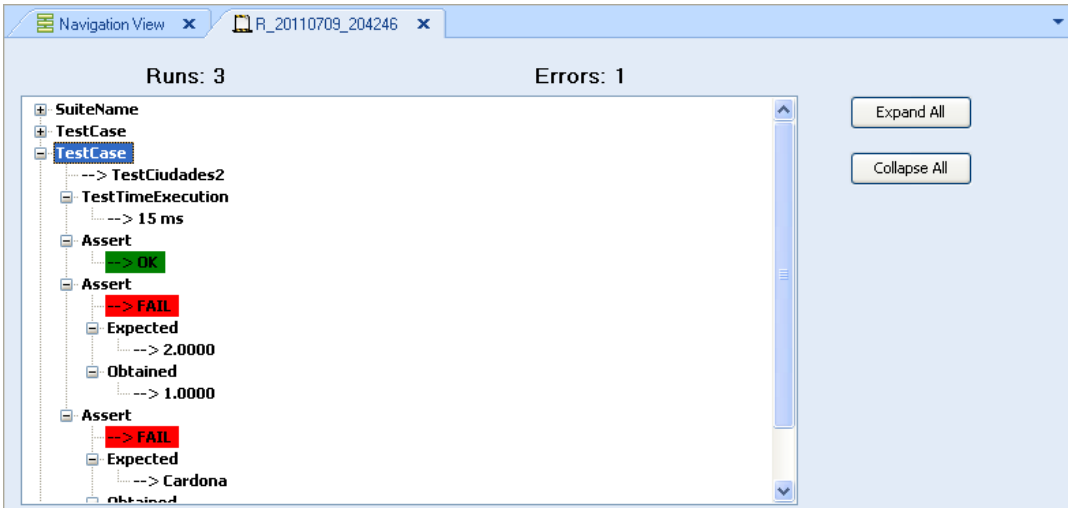


(fig 8.4.2 – Generación del objeto RunnerProcedure y todas sus dependencias)

El RunnerProcedure se ejecuta mediante un comando que provee el SDK de GeneXus para la ejecución de objetos.

Visualización de resultados

Luego de ejecutados los casos de prueba se crea un objeto "Result" donde quedará registro del resultado de las prueba. Este objeto tendrá como nombre la fecha y hora de ejecución. En el cabezal se muestra los Test Cases ejecutados y la cantidad que fallaron. Los *asserts* que fallen se mostrarán en rojo con el detalle de la falla y los que se completen satisfactoriamente se mostrarán en verde *(fig 8.4.3)*.



(fig 8.4.3 – Resultado de la ejecución de las pruebas seleccionadas)

9. Pruebas de GXUnit

Para probar la herramienta se definió un plan de pruebas que fue incrementándose a medida que avanzaba el proyecto.

Se definieron distintos escenarios de prueba para los cuales se especificó a alto nivel el alcance de la prueba y el comportamiento esperado de la misma. Luego, para cada escenario se definieron distintos casos de prueba, en los cuales se especificó detalladamente cómo ejecutar el caso y su resultado esperado. Además, como soporte para poder ejecutar las pruebas, se definió un conjunto de objetos GeneXus de prueba que incluye Procedimientos, Transacciones, Data Providers y SDTs para poder probar cada tipo de objeto GeneXus incluido en el alcance. Los SDTs son necesarios para considerar DataProviders que tengan como salida SDTs.

Para cada una de las versiones liberadas se ejecutó el conjunto de pruebas diseñadas y se fue enriqueciendo según las nuevas funcionalidades incluidas.

9.1. *Escenarios de prueba de GXUnit*

Nombre	Descripción	Comportamiento Esperado
Crear Nueva KB	Crear una nueva KB	Se crea una carpeta GXUnit en el árbol de objetos de la KB y se crean los objetos que necesita la herramienta
Abrir KB por primera vez	Abrir una KB que nunca haya sido abierta con una versión de GeneXus con la extensión GXUnit instalada	Se crea una carpeta GXUnit en el árbol de objetos de la KB y se crean los objetos que necesita la herramienta
Abrir KB ya utilizada	Abrir una KB que contenga los objetos que agrega GXUnit	La KB abre y se mantienen los objetos creados por GXUnit
Inicializar GXUnit con objetos	Ejecutar la opción del menú para iniciar los objetos GXUnit en una KB que ya contenga los objetos	Los objetos son recreados
Inicializar GXUnit sin objetos	Eliminar los objetos creados por GXUnit y ejecutar la funcionalidad mencionada anteriormente	Se crea la carpeta GXUnit en el árbol de objetos con los objetos GXUnit
Eliminación GXUnit	Ejecutar la funcionalidad “Eliminar Objetos GXUnit”	Los objetos se eliminan de la KB, tanto los objetos dentro de la carpeta GXUnit como los casos de prueba que se crearon
Re eliminar GXUnit	Una vez eliminados los objetos, volver a ejecutar la funcionalidad “Eliminar Objetos GXUnit”	No se elimina ningún otro objeto ni muestra ningún error.
Crear Suite sin objetos GXUnit	Crear un Suite luego de eliminar los objetos GXUnit	Mostrar error por no existir los objetos necesarios y permitir crearlos

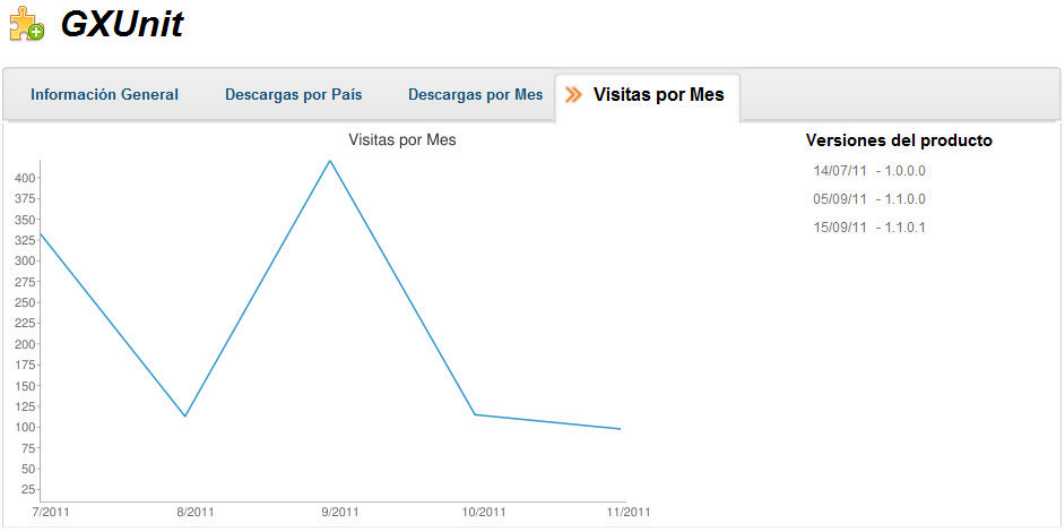
Crear Test Case para Procedimiento	Crear un nuevo objeto de tipo Test Case y asociarlo a un Procedimiento	Se crea un nuevo Test Case con la propiedad TestCase = "true", objectToTest con el objeto asociado y se genera el código automatizado que llama al Procedimiento
Crear Test Case para Transacción BC	Crear un nuevo objeto de tipo Test Case y asociarlo a una Transacción	Se crea un nuevo Test Case con la propiedad TestCase = "true", objectToTest la transacción y se genera el código automatizado que llama al BC
Crear Test Case para Data Provider	Crear un nuevo objeto de tipo Test Case y asociarlo a un DataProvider	Se crea un nuevo Test Case con la propiedad TestCase = "true", objectToTest el data provider y se genera el código automatizado que llama al DataProvider
Guardar un nuevo Test Case	Para cada uno de los Test Case creados, guardar el objeto creado	Se agrega el objeto Test Case a la lista de Test Cases en la Tool Window de GXUnit
Modificar Test Case	Modificar el código generado de forma automática al crear un Test Case. Modificar una variable, guardar y cerrarlo	El código existente al abrir el Test Case es igual al modificado por el usuario
Eliminar Test Case	Eliminar un objeto Test Case	El objeto debe eliminarse y debe desaparecer de la lista de Test Case de la Tool Window
Exportación de Test Case	Exportar un caso de prueba de una KB, borrar el Test Case e importarlo	El objeto Test Case es idéntico al primero y se agrega a los Test Case de la lista
Exportar objeto Test Case a KB sin el objeto asociado	Exportar un objeto Test Case de una KB, importarlo en otra KB que no contenga el Objeto Asociado	La importación muestra un error que indica que no existe el objeto asociado
Ejecutar Test Case Uno a uno	Seleccionar un Test Case de la lista en la Tool Window de GXUnit y presionar el botón Test	Se realiza Build del Test Case seleccionado y de los objetos a los que llama (si no han sido especificados y/o generados). Se crea el Procedimiento ProcedureRunner que llama al Test Case seleccionado y se le hace Build. Se muestra en Pantalla el resultado del Test

Ejecutar todos los Test Case	Seleccionar todos los Test Case de la lista en la Tool Window de GXUnit y presionar el botón Test	Se realiza Build de todos los TC y de los objetos a los que llama (si no han sido especificados y/o generados). Se crea el Procedimiento ProcedureRunner que llama al Test Case seleccionado y se le hace Build. Se muestra en Pantalla el resultado del Test
Crear Suite de Prueba	Crea una suite en el primer nivel de la Tool Window GXUnit	Queda la suite creada
Crear Suite de Prueba en segundo nivel	Crea una suite debajo de una suite existente en la Tool Window GXUnit	Queda la suite creada debajo de la seleccionada como padre
Crear Test Case debajo de una Suite	Crea un Test Case debajo de una suite existente en la Tool Window GXUnit	Queda el Test Case creado debajo de la seleccionada como padre
Eliminar Suite vacía	Eliminar Suite que no contenga ningún Test Case ni Suite	Ya no existe la suite eliminada
Eliminar Suite con hijos	Eliminar Suite que contenga Test Cases y Suites	Ya no existe la suite eliminada ni ninguno de los hijos que contenía
Ejecutar test en KB .NET	Ejecutar un Test Case en una KB generando en .NET	Se ejecuta y muestra el resultado obtenido
Ejecutar test en KB Java	Ejecutar un Test Case en una KB generando en Java	Se ejecuta y muestra el resultado obtenido
Ejecutar test en KB Ruby	Ejecutar un Test Case en una KB generando en Ruby	Se ejecuta y muestra el resultado obtenido
Exportar resultado	Exportar un objeto Result	Genera el xpz
Importar resultado	Importar un objeto Result	El objeto Result queda creado en la KB

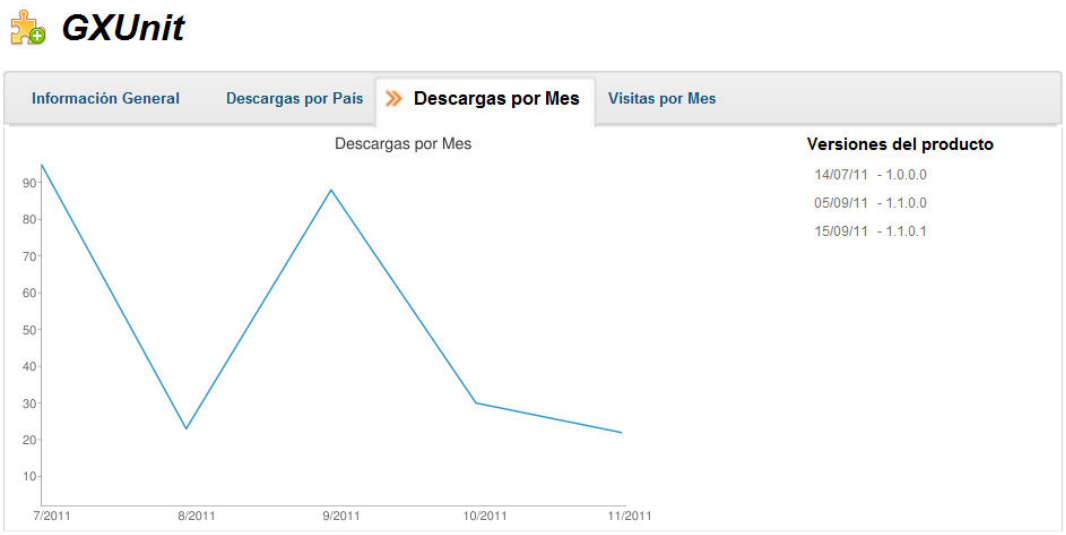
10. Utilización de GXUnit e importancia en la comunidad GeneXus

Como ya se mencionó, GXUnit es una herramienta solicitada dentro de la comunidad desde hace años. Por este motivo el proyecto tuvo gran apoyo desde Artech, aportando requerimientos, dando soporte sobre extensiones e inclusive incluyéndonos dentro del Evento Internacional GeneXus 2011[52] para dar una charla sobre GXUnit[53]. Además la herramienta se encuentra disponible en el GeneXus MarketPlace [15] desde que se tuvo una versión estable y se ha ido actualizando con el avance del proyecto. En la imagen *(fig 10.1)* puede verse información general, en las imágenes *(fig 10.2)* y *(fig 10.3)* pueden verse las descargas y visitas por mes respectivamente, y en la imagen *(fig 10.4)* pueden verse los porcentajes de descargas por países. Todas las imágenes fueron extraídas de las estadísticas del MarketPlace.

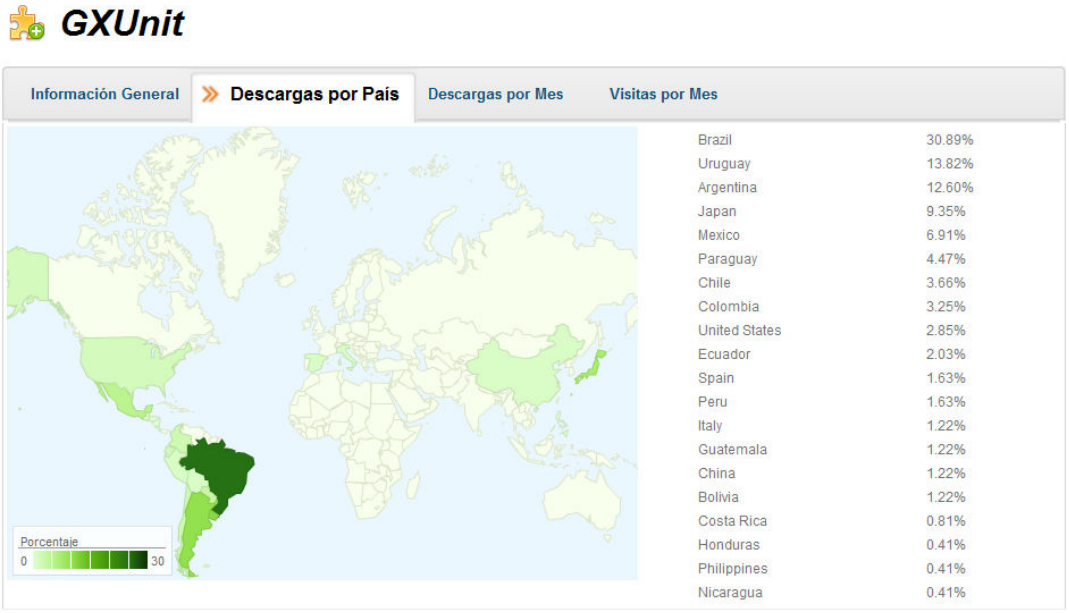
(fig 10.1 – Información General)



(fig 10.2 – Visitas por mes)



(fig 10.3 – Descargas por Mes)



(fig 10.4 – Descargas por país)

Luego de la presentación de GXUnit en el encuentro GeneXus, la cantidad de usuarios aumentó considerablemente como puede verse en la Figura *(fig 10.4)*. Este crecimiento en el número de usuarios requirió darles soporte a través del MarketPlace, a través del cual se obtuvieron sugerencias que se utilizaron para implementar mejoras a la herramienta. De los usuarios de GXUnit se destacan dos grupos en particular y se utiliza la experiencia de los mismos como caso de estudio para analizar.

10.1. *Caso de Estudio: Grupo 02 y 03 - PIS 2011*

En la edición 2011 del curso Proyecto de Ingeniería de Software de la Facultad de Ingeniería de la UdelaR, dos de los grupos utilizaron GeneXus como herramienta de desarrollo. Se ofreció a estos grupos utilizar la versión de GXUnit construida en este proyecto y en esta sección se describen las experiencias de los grupos con la herramienta. La sección expone por qué decidieron utilizar GXUnit, en dónde y cómo lo utilizaron, y finalmente las conclusiones que sacaron de la herramienta. Este material fue tomado de documentos que ambos grupos elaboraron sobre GXUnit[54, 55].

Grupo 02

Uso de GXUnit

Decidieron utilizar esta herramienta por la usabilidad que presenta, la facilidad de aprendizaje y las funcionalidades que provee, como diseñar pruebas unitarias y ejecutar pruebas de regresión. Además es la única herramienta de esta naturaleza incorporada a GeneXus y la misma fue liberada hace poco, por lo tanto era de interés para el proyecto conocer las experiencias con la misma.

Se utilizó GXUnit para testear el módulo de Seguridad, dado que el mismo es uno de los módulos más críticos del sistema. Los procedimientos que se probaron son más que nada procedimientos que crean instancias de distintos objetos y las relacionan. También se prueban procedimientos de tipo "*get*" pero los mismos son testeados principalmente para ver que los datos guardados en los procedimientos de creación asociados sean correctos.

Conclusiones de los grupos sobre GXUnit

"…La herramienta utilizada resultó ser de fácil aprendizaje y uso. La misma es intuitiva, permite rápidamente generar tests y ejecutar pruebas de regresión.

Dado el avance de nuestra aplicación logramos detectar pocos errores pero seguramente si hubiéramos utilizado antes la herramienta le podríamos haber sacado un provecho mucho mayor. De todas formas cabe destacar que de no haberse encontrado los errores a tiempo en la primera ejecución del test se hubiera pasado un tiempo posiblemente mayor para encontrar los mismos ya que estos hubieran sido detectados recién en pruebas funcionales del sistema. Además, gracias a haber construido este test y a esta herramienta pudimos destinar a realizar las pruebas de regresión pocos minutos, cuando probar todos los procedimientos sin la misma hubiera llevado horas.

Como única crítica a la herramienta que podemos encontrar es que sus métodos de *Assert* solo permiten comparar enteros y strings lo cual hizo que no pudiéramos testear varios procedimientos o al menos no de forma sencilla ya que es muy común devolver SDTs en GeneXus los cuales no se pueden comparar con GXUnit. Entendemos que este es un requerimiento muy exquisito pero con él se podría explotar a un nivel mucho mayor el testing con la herramienta…"[54]

Grupo 03

La razón por la que el grupo usó GXUnit fue porque permite una rápida definición de casos de prueba para un módulo dado. Al desarrollar un producto de forma iterativa e incremental se torna muy engorroso probar las mismas funcionalidades manualmente a través de las sucesivas

liberaciones, por lo que lo más interesante a destacar luego de haber usado la herramienta, es la posibilidad de ejecutar pruebas de regresión. La misma brinda la posibilidad de guardar los sets de pruebas y ejecutarlos muchas veces sin esfuerzo logrando la automatización de las pruebas unitarias. Se usó GXUnit en forma experimental, para verificar los procedimientos más significativos de un caso de uso en Test Cases independientes, para lo que el uso de la herramienta se mostró apropiado.[55]

Encuesta a desarrolladores que utilizaron la herramienta

Luego de finalizado el proyecto realizamos una encuesta a los distintos desarrolladores de los grupos para obtener su opinión con respecto a GXUnit.

Desarrollador 1, Marcel Burdiat - Grupo 02

Pregunta 1: ¿Le resultó útil la guía de usuario para instalar y utilizar GXUnit?
Respuesta: Si, utilizamos la misma para instalar la herramienta.

Pregunta 2: ¿Utilizaron otra herramienta de pruebas además de GXUnit?
Respuesta: No, GXUnit fue la única.

Pregunta 3: ¿Qué objetos GeneXus probaron con GXUnit?
Respuesta: Utilizamos GXUnit solamente con procedimientos.

Pregunta 4: ¿Que otro objeto GeneXus le gustaría probar con GXUnit? Si hay alguno, ¿cuál le parece que debería ser el siguiente a incluir?
Respuesta: -.

Pregunta 5: ¿Le pareció útil la generación de código por defecto generado por GXUnit?
Respuesta: Nos pareció útil, sin embargo no la usamos ya que probamos muchos objetos en el mismo test.

Pregunta 6: ¿Le agregaría algo al código por defecto?
Respuesta: Podría ser bueno que se agreguen todas las variables del tipo correcto ya que se crean variables con las que se llaman a procedimientos pero no del tipo correcto.

Pregunta 7: ¿Le resulto útil la agrupación de los test mediante las suites?
Respuesta: Nos pareció útil, sin embargo no la usamos ya que realizamos un solo test.

Pregunta 8: ¿Qué información le agregaría al objeto resultado?
Respuesta: -.

Pregunta 9: ¿Tuvieron algún tipo de interferencia entre los objetos GXUnit y los de la propia aplicación?
Respuesta: En alguna ocasión fue necesario bajar el plugin de GXUnit para poder realizar un *update* de GxServer, no sé bien el motivo.

Pregunta 10: ¿Les resulto útil GXUnit en general?
Respuesta: Si.

Desarrollador 2, Tatiana García - Grupo 03

Pregunta 1: ¿Le resultó útil la guía de usuario para instalar y utilizar GXUnit?
Respuesta: Si

Pregunta 2: ¿Utilizaron otra herramienta de pruebas además de GXUnit?
Respuesta: No

Pregunta 3: ¿Qué objetos GeneXus probaron con GXUnit?
Respuesta: Ninguno

Pregunta 4: ¿Que otro objeto GeneXus le gustaría probar con GXUnit? Si hay alguno, ¿cuál le parece que debería ser el siguiente a incluir?
Respuesta: -.

Pregunta 5: ¿Le pareció útil la generación de código por defecto generado por GXUnit?
Respuesta: Más o menos.

Pregunta 6: ¿Le agregaría algo al código por defecto?
Respuesta: No sé.

Pregunta 7: ¿Le resulto útil la agrupación de los test mediante las suites?
Respuesta: Si queda más ordenado.

Pregunta 8: ¿Qué información le agregaría al objeto resultado?
Respuesta: No sé.

Pregunta 9: ¿Tuvieron algún tipo de interferencia entre los objetos GXUnit y los de la propia aplicación?
Respuesta: Si, la verdad es que estábamos usando GXServer y me dio algunos problemas para hacer *update* y *commit* luego que intenté utilizar GXUnit.

Pregunta 10: ¿Les resulto útil GXUnit en general?
Respuesta: Particularmente no, porque no la utilicé debido al problema que me generó con GXServer.

Desarrollador 3, Juan Pablo Revello - Grupo 03

Pregunta 1: ¿Le resultó útil la guía de usuario para instalar y utilizar GXUnit?
Respuesta: Muy útil y detallada.

Pregunta 2: ¿Utilizaron otra herramienta de pruebas además de GXUnit?
Respuesta: No.

Pregunta 3: ¿Qué objetos GeneXus probaron con GXUnit?
Respuesta: Transacciones y Procedimientos.

Pregunta 4: ¿Que otro objeto GeneXus le gustaría probar con GXUnit? Si hay alguno, ¿cuál le parece que debería ser el siguiente a incluir?
Respuesta: Ninguno.

Pregunta 5: ¿Le pareció útil la generación de código por defecto generado por GXUnit?
Respuesta: Es muy útil en primera instancia cuando uno aún no sabe bien como debería funcionar pero obviamente no alcanza con eso.

Pregunta 6: ¿Le agregaría algo al código por defecto?
Respuesta: Algunos casos bordes dependiendo del tipo en los cuales esta definidas las variables que se ingresan como parámetros.

Pregunta 7: ¿Le resulto útil la agrupación de los test mediante las suites?
Respuesta: Si, ayuda a mantener ordenada la kb.

Pregunta 8: ¿Qué información le agregaría al objeto resultado?
Respuesta: Creo que ninguna.

Pregunta 9: ¿Tuvieron algún tipo de interferencia entre los objetos GXUnit y los de la propia aplicación?
Respuesta: Tuvimos problemas al utilizar GXServer junto con los objetos GXUnit.

Pregunta 10: ¿Les resulto útil GXUnit en general?
Respuesta: Si.

Desarrollador 4, Andrés Fernández - Grupo 03

Pregunta 1: ¿Le resultó útil la guía de usuario para instalar y utilizar GXUnit?
Respuesta: Sí, se pudo instalar GXUnit sin complicaciones. La guía es clara y permitió comenzar a utilizar GXUnit rápidamente.

Pregunta 2: ¿Utilizaron otra herramienta de pruebas además de GXUnit?
Respuesta: No

Pregunta 3: ¿Qué objetos GeneXus probaron con GXUnit?
Respuesta: Únicamente procedimientos.

Pregunta 4: ¿Que otro objeto GeneXus le gustaría probar con GXUnit? Si hay alguno, ¿cuál le parece que debería ser el siguiente a incluir?
Respuesta: Web Panels y Web Components

Pregunta 5: ¿Le pareció útil la generación de código por defecto generado por GXUnit?
Respuesta: Sí como guía inicial de como programar los procedimientos de testeo, aunque luego no respetamos el orden de los distintos bloques.

Pregunta 6: ¿Le agregaría algo al código por defecto?
Respuesta: No

Pregunta 7: ¿Le resulto útil la agrupación de los test mediante las suites?
Respuesta: Si

Pregunta 8: ¿Qué información le agregaría al objeto resultado?
Respuesta: La línea en el procedimiento donde se declara el *assert* junto al resultado de ese *assert* (falle o no).

Pregunta 9: ¿Tuvieron algún tipo de interferencia entre los objetos GXUnit y los de la propia aplicación?
Respuesta: Con la aplicación propiamente no, pero tuvimos algunos inconvenientes con GXServer y los objetos generados por GXUnit.

Pregunta 10: ¿Les resulto útil GXUnit en general?
Respuesta: Si, fue muy útil para automatizar nuestras pruebas de regresión.

11. Conclusiones

Al comenzar la etapa de implementación se encontraron dificultades debido a la poca información disponible acerca del desarrollo de extensiones para GeneXus, teniendo que apoyarse fundamentalmente en el foro de extensiones, en extensiones de código abierto disponibles en internet y en los proyectos realizados por los grupos de Ingeniería de Software del año 2007.

La etapa de desarrollo resultó compleja debido a la decisión de definir una arquitectura en capas dado que se debieron implementar los servicios de las capas inferiores para ser consumidos por las capas superiores. Por esta razón un objeto perteneciente a una capa superior no puede comunicarse directamente con el SDK de GeneXus. Sin embargo la decisión de definir una arquitectura en capas resultó muy beneficiosa al momento de probar la herramienta y extenderla. El hecho de que cada capa centraliza y encapsula sus funcionalidades resultó más fácil para realizar un desarrollo incremental, identificar los errores y agregar nuevas funcionalidades.

Sobre el uso de la herramienta pudimos concluir que aumenta la productividad del proceso de verificación al facilitar la construcción de casos de prueba y automatizar la ejecución de los mismos. Ejecutar todas las pruebas con la facilidad de un botón, permite realizar pruebas de regresión de una manera sencilla y práctica lo que puede inclusive ayudar a incrementar el número de veces que el desarrollador ejecuta las pruebas. Los dos grupos estudiados que utilizaron la herramienta coinciden en que GXUnit es muy útil para el mantenimiento de casos de prueba y ejecución de pruebas de regresión.

Además, gracias al mantenimiento de casos de prueba es posible definir los casos antes de implementar la lógica del objeto a probar, por lo tanto puede utilizarse sobre GeneXus la metodología de desarrollo “Test-Driven Development”.

Finalmente, creemos que la mayoría de los objetivos propuestos para este trabajo fueron logrados con éxito, siendo el principal la consolidación de una herramienta estable para la explotación en la industria. También se logró generar una recopilación de conocimientos muy valiosa para continuar con el desarrollo de la herramienta.

12. Trabajo futuro

Algo que se manejó al comienzo del proyecto pero no se llegó a incluir en el alcance fue la simulación de objetos GeneXus (*mocks* de objetos) y la simulación de acceso a la base de datos, éstos son requerimientos deseables para seguir desarrollando la herramienta.

Sería de mucha utilidad poder ejecutar casos de prueba de forma *batch* (por ejemplo con tareas *MSBuild*[3]), pudiendo ser ejecutados fuera del *IDE* de GeneXus, así como obtener los datos de entrada de archivos. También generar datos de prueba que nos garanticen el cubrimiento del código según algún criterio preestablecido.

La integración con GXtest [56] es un requerimiento deseable para la herramienta, integración en el sentido que GXtest pueda utilizar el banco de pruebas de GXUnit, y que GXUnit a su vez, pueda utilizar las funcionalidades de GXtest para dar soporte a las pruebas unitarias en WebPanels.

El incremento en la utilización de Smart Devices hace importante dar un soporte especial a pruebas unitarias sobre las aplicaciones para los dispositivos.

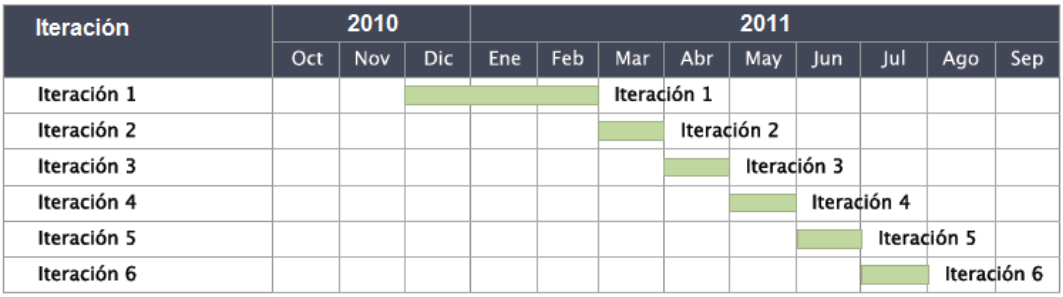
Mejorar la visualización de los resultados y generación de reportes con toda la información y estadísticas de ejecución de las pruebas.

También es deseable que las suites de prueba pasen a ser un nuevo objeto GeneXus para permitir que un Test Case pueda estar en más de una suite, ya que sólo puede estar en un Folder.

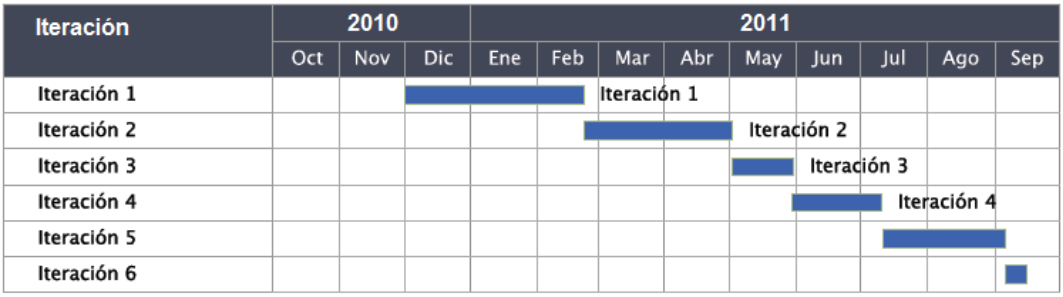
3- Por definición ver glosario.

13. Anexo A - Cronograma

Al comenzar la implementación de GXUnit, se definió un cronograma de liberaciones de las distintas funcionalidades de la herramienta. En esta sección se especifica cuál fue ese cronograma indicando el contenido de cada liberación, la fecha esperada de la liberación y la fecha en que realmente se liberó.



(*fig 13.1 – Gantt de iteraciones planificado*)



(*fig 13.2 – Gantt de iteraciones reales*)

En las figuras *13.1* y *13.2* se muestran la planificación de las iteraciones y la ejecución real de las mismas.

Iteración 1

Fecha planeada de liberación: Febrero 2011

Fecha de liberación: 21/02/2011

Inicialización de GXUnit

Al abrir una KB se crean un conjunto de objetos necesarios para poder utilizar la herramienta. Si estos objetos ya existen no se hace nada, sino son creados y colocados en una carpeta llamada GXUnit en el árbol de objetos.

Esta funcionalidad también puede ejecutarse mediante una opción del menú en cualquier momento.

Eliminación de objetos GXUnit

Existe una funcionalidad en el menú que elimina todos los objetos relacionados con GXUnit, esto incluye todos los objetos TestCase que hayan sido creados.

Caso de prueba

Es posible crear un objeto GeneXus de un nuevo tipo llamado Test Case. Al crear un nuevo objeto de este tipo se pregunta a qué objeto está asociado. Éste tiene un editor donde se utiliza la sintaxis del objeto Procedimiento para llamar a cualquier otro objeto GeneXus y realizar las comparaciones llamando a los objetos de tipo *Assert* que se describen más adelante.

Al crearse un caso de prueba asociado a un procedimiento se genera automáticamente código GeneXus donde se llama al objeto a probar y se realizan *asserts* para evaluar los resultados de dicha llamada. Se crea una variable por cada una que se encuentre en la regla Parm del procedimiento a probar y según su tipo, si es de entrada o entrada/salida se inicializa con un valor por defecto o si es de salida o entrada/salida se evalúa con una llamada a un *Assert* (según el tipo).

Luego de que el caso de prueba está creado puede re generarse su código por defecto mediante una opción del menú (teniendo en cuenta cualquier cambio en la regla Parm del objeto a probar).

Exportación e Importación de Casos de Prueba

Dado que el nuevo objeto "Test case" está basado en un procedimiento, se puede exportar e importar entre las distintas Bases de Conocimiento que tengan instalada la herramienta.

Asserts

Los métodos de tipo *Assert* son utilizados para comparar los resultados de la ejecución de objetos bajo prueba, con los resultados que se esperan de dichos objetos. En esta iteración fueron implementados como Procedimientos GeneXus. Se proveen dos de estos métodos, *AssertNumericEquals* y *AssertStringEquals*, para comparar variables numéricas y cadenas de caracteres respectivamente. Estos procedimientos son creados al inicializarse GXUnit y forman parte de la KB.

Al ejecutarse uno de estos métodos se guardan los resultados (en esta iteración en un XML) para luego poder visualizar todos los resultados de la ejecución de los casos de prueba.

Generación y ejecución de Casos de Prueba

Todos los Casos de Prueba

Se pueden generar y ejecutar todos los Casos de Prueba existentes en la base de Conocimiento mediante el comando "Generar" y "Ejecutar" en el menú GXUnit.

Conjunto de Casos de Prueba

La herramienta tiene una Tool Window que contiene, en esta iteración, la totalidad de los casos de prueba.

Seleccionando los casos de prueba que quieren ser ejecutados (mediante su respectivo check box) al presionar botón Build Test Cases se crea un nuevo procedimiento (TestRunner) que tiene las llamadas a todos los objetos de tipo Test Case a ejecutar.

Ejecución de Casos de Prueba y visualización de resultados

Luego, al presionar Ejecutar en el menú de GXUnit o Test en la Tool Window se ejecuta el TestRunner y se muestra el resultado de los casos de prueba (el XML) en la Start Page.

Iteración 2

Fecha planeada de liberación: Marzo 2011

Fecha de liberación: 01/05/2011

Casos de prueba de Procedimientos

En la primera iteración se realizó un prototipo para la generación de código de casos de prueba de Procedimientos. Con el conocimiento adquirido en la construcción del prototipo se redefine el generador de código adaptándolo a la arquitectura del sistema.

Casos de prueba de Transacciones

Al crear un nuevo caso de prueba asociado a una transacción con la propiedad Business Component (BC) se genera código genexus por defecto de la siguiente manera:

- Se define una variable del tipo de la transacción
- Inicializa con un valor por defecto todos los atributos
- Guarda (*save* del BC)
- Se carga (*load* del BC)
- Se realiza un *assert* por cada atributo del Business Component

Ambiente de pruebas

En la primera iteración la ejecución de las pruebas está limitada exclusivamente para ambiente C#. En la iteración 2 la ejecución de los casos de prueba es independiente del ambiente que se esté generando.

Muestra de Resultados

Los resultados se guardan como un nuevo tipo de objeto GeneXus “TestResult”, permitiendo de esta forma guardar los resultados dentro de la KB, exportarlos, importarlos y demás utilidades que pueden realizarse a un objeto. Se puede visualizar los resultados (el XML) en una pestaña del nuevo objeto.

Iteración 3

Fecha planeada de liberación: Abril 2011

Fecha de liberación: 29/05/2011

Suites

Se crea un nuevo tipo de objeto GeneXus “Suite” para agrupar Casos de Prueba. En la pantalla de definición del objeto se listan todos los casos de prueba permitiendo al usuario seleccionar un conjunto de ellos. En la Tool Window se visualizan las suites creados con sus respectivos casos de prueba.

Generación y ejecución

Se agrega un botón en el Tool Window para poder generar y ejecutar los casos de prueba

seleccionados con un solo "*click*".

Muestra de Resultados

Se agrega una nueva pestaña al objeto "TestResult" con el fin de facilitar la lectura de los resultados obtenidos.

Casos de Prueba de Data Providers

Dado que los Data Providers retornan los datos mediante un SDT, se comparan los SDTs haciéndoles un ToXml() y utilizando el *AssertStringEquals*.

Iteración 4

Fecha planeada de liberación: Mayo 2011

Fecha de liberación: 10/07/2011

Suites

La estructura de los objetos Suite es arborescente, al crear un nuevo Suite o Test case se indica cuál es su Suite padre. En la Tool Window se visualiza la estructura de los Suites y permite su ejecución (todos los Test cases y Suites hijos).

Muestra de Resultados

Se mejora la pestaña de visualización del objeto "TestResult". Además del árbol con los resultados se agrega un resumen del total de los casos de prueba ejecutados, la cantidad de errores y la fecha y hora de ejecución.

Casos de Prueba de Data Providers

Al crear un objeto Caso de prueba asociado a un Data Provider se inicializa su código fuente de la siguiente manera:

- Crea variables SDT del tipo devuelto por el Data Provider
- Inicializa una variable con valores por defecto, para luego ser modificada por el valor esperado.
- Realiza la llamada al Data Provider
- Compara con el *AssertStringEquals* el resultado esperado con el valor obtenido aplicando la función ToXml a ambos SDTs.

Assert

Los métodos *Assert* tienen un nuevo parámetro descriptivo para poder diferenciarlos dentro de una prueba. Esta descripción se muestra en el árbol de resultados.

Iteración 5

Fecha planeada de liberación: Junio 2011

Fecha de liberación: 05/09/2011

Suites

La modificación de los suites se puede hacer directamente desde la tool window, arrastrando lo test cases y suites a otros niveles del árbol.

Muestra de Resultados

Se mejora la estética de la pestaña de visualización del objeto "TestResult". Identificamos los errores expandiendo los nodos y resaltando con color rojo, y los satisfactorios colapsándolos y resaltando con verde. Además se muestra el tiempo de ejecución de cada Prueba.

Iteración 6

Fecha planeada de liberación: Julio 2011

Fecha de liberación: 15/09/2011

Corrección de errores

Se corrigen los errores encontrados.

Revisión General de GXUnit

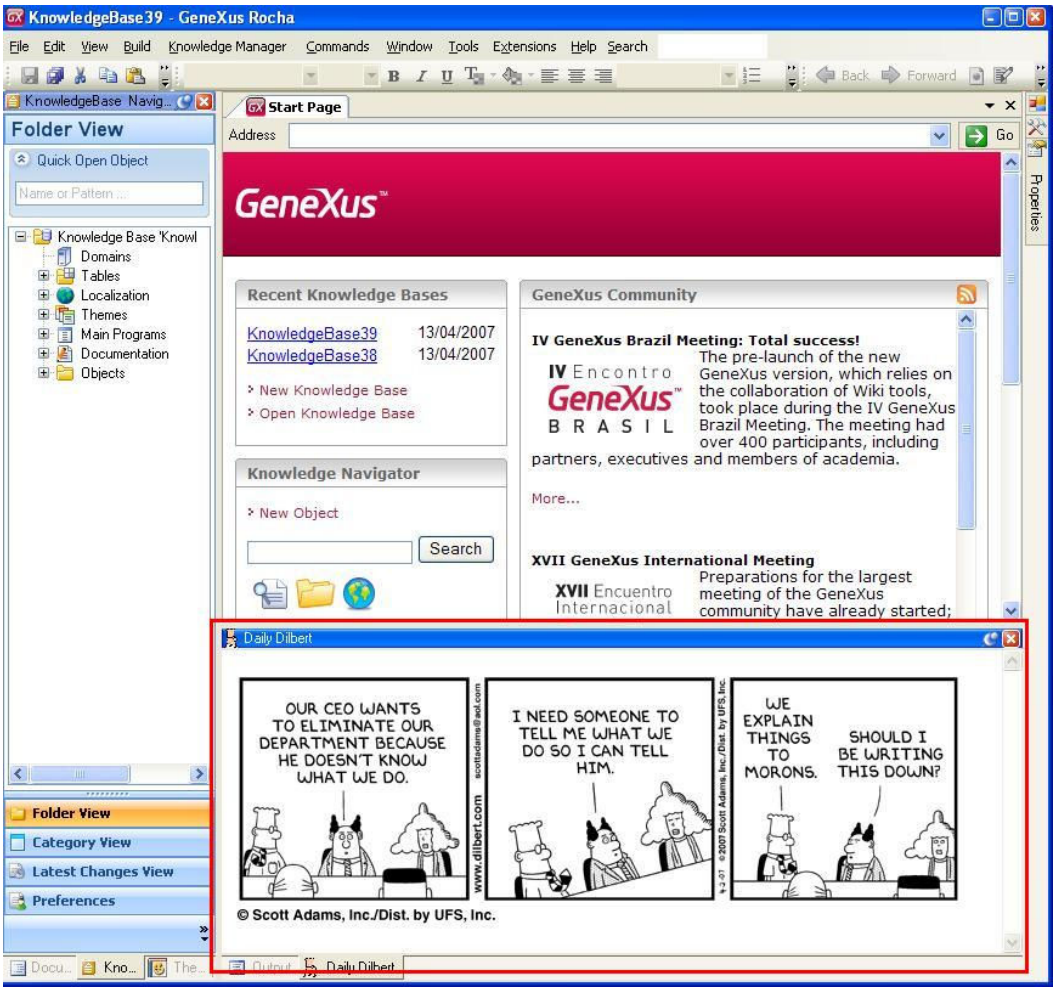
Revisión general de las funcionalidades de la herramienta.

Ejemplos

Además definimos ejemplos de errores que se dan en la realidad, los cuales serían detectados con GXUnit.

14. Anexo B - Ejemplo de extensión

Una de las extensiones de ejemplo que acompañan el SDK de GeneXus es Daily Dilbert *(fig 14.1)*, esta extensión consiste básicamente en una ventana de herramientas GeneXus que muestra HTML creado dinámicamente para mostrar fuentes RSS utilizando transformaciones XSL.



(fig 14.1 – Extensión Daily Dilbert)

El formulario de la ventana (*DilbertWindow*.cs) se deriva de la clase *AbstractToolWindow*, y contiene un System.Windows.Forms.WebBrowser con la propiedad Dock para que cubra todo el formulario de la ventana.

El SDK también provee un archivo Transform.xslt, que contiene las instrucciones para crear contenido HTML a partir de un XML obtenido a través de una fuente RSS.

Para ver el ejemplo con mayor profundidad se puede encontrar más información en[57].

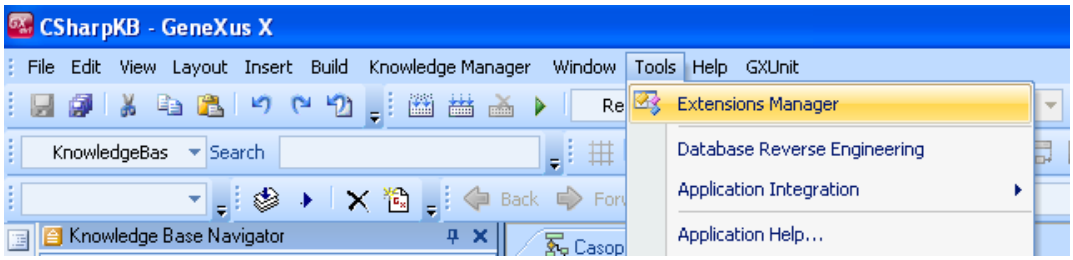
Este es un ejemplo de extensión GeneXus, en donde se agrega una nueva ventana de herramienta al entorno de GeneXus para que muestre información procedente de una fuente de RSS, de la misma forma, se puede crear una extensión que agregue un nuevo tipo de objeto o alguna funcionalidad nueva, por ejemplo, existe otra extensión que borra los archivos generados por GeneXus asociados a un determinado objeto, cuando se borra el objeto GeneXus de la KB.

15. Anexo C - Guía de usuario

Este anexo tiene el objetivo de mostrar las funcionalidades ofrecidas por la herramienta de pruebas unitarias GXUnit. La misma ha sido desarrollada como una extensión para GeneXus X Ev 1 U6.

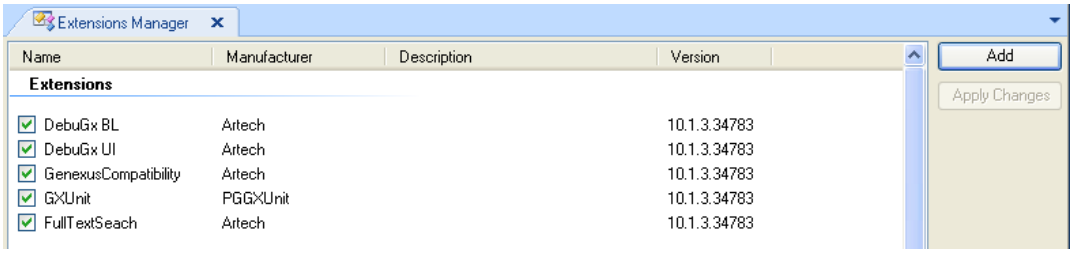
Instalación

Como cualquier extensión de GeneXus X, GXUnit se instala a partir de un archivo Dynamic-link library (DLL) generado para este propósito. En el *IDE* de GeneXus se accede al "Extension Manager" en el menú "Tools" *(fig 15.1)*.



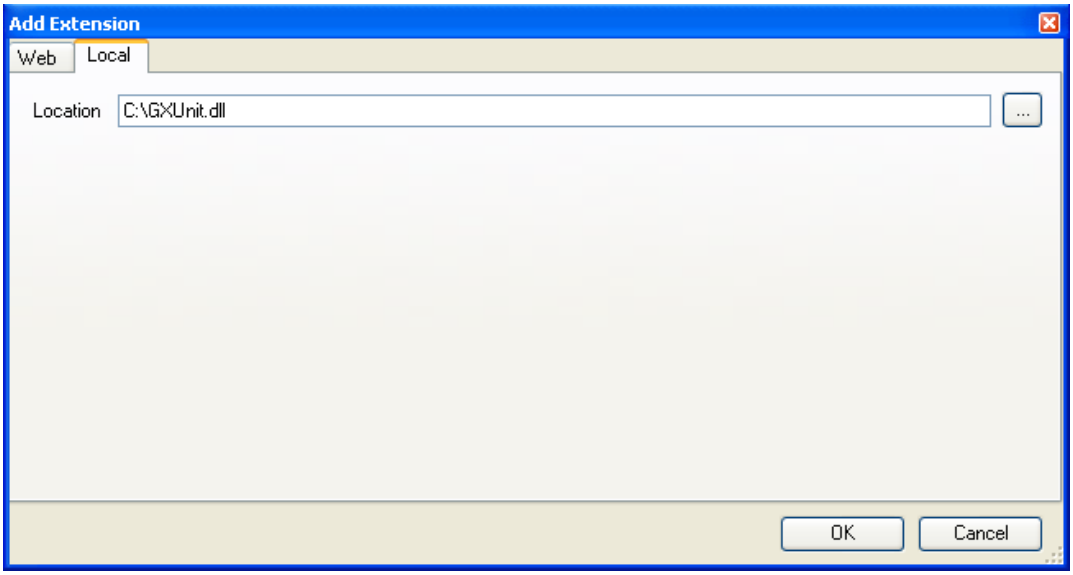
(fig 15.1 – Extension Manager)

Una vez en el "Extension Manager" se puede agregar el archivo DLL mediante el botón "Add" *(fig 15.2)*.



(fig 15.2 – Botón "Add" del Extension Manager)

Aquí se debe seleccionar la pestaña "Local" y se busca el archivo DLL donde se encuentre guardada en el disco local *(fig 15.3)*.



(fig 15.3 – Selección de la ubicación de la dll)

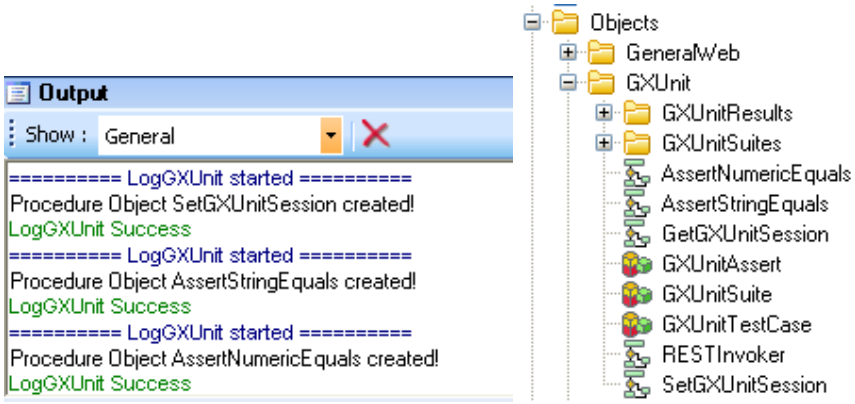
Al presionar "OK" pregunta si se desea instalar y al acceder se copia el archivo DLL a la carpeta Packages que se encuentra dentro de la instalación de GeneXus.

Una vez alcanzado este paso puede verse GXUnit en la lista de extensiones que se muestra en la fig 15.2. Luego se debe activar el check box de GXUnit y presionar "Apply Changes". GeneXus exige reiniciar e instala la herramienta.

Además se debe instalar el *template* para el código de los Test Cases "TestCaseSourceCall.dkt". Dicho archivo debe copiarse en la carpeta Templates que se encuentra en el directorio de instalación de GeneXus.

Inicialización de GXUnit

Al abrir una KB (Knowledge Base) se crean un conjunto de objetos necesarios para poder utilizar la herramienta. Estos objetos son colocados en una carpeta llamada GXUnit en el árbol de objetos *(fig 15.4)*.



(fig 15.4 – Objetos GXUnit Inicializados)

Esta funcionalidad también puede ejecutarse mediante una opción del menú en cualquier momento *(fig 15.5)*.

(fig 15.5 – Inicialización manual de los objetos GXUnit)

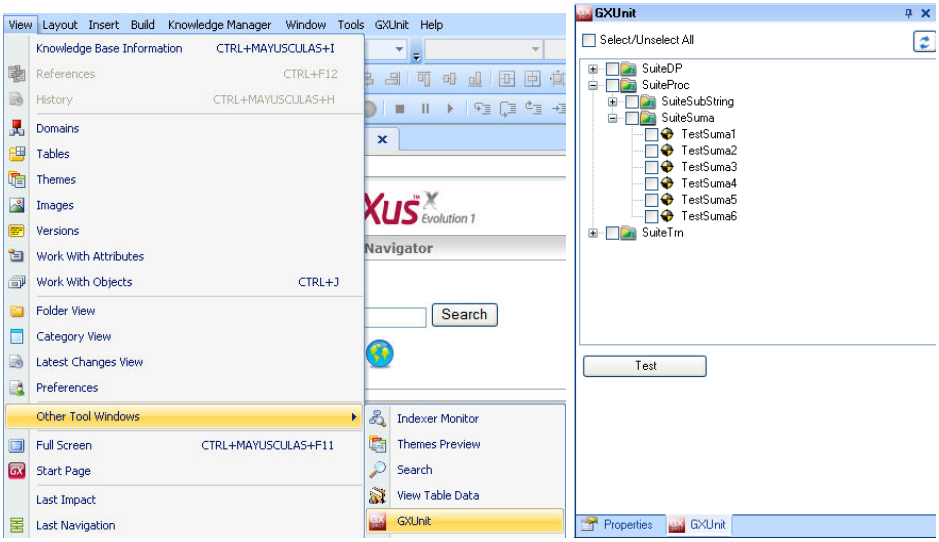
Eliminación de objetos GXUnit

Existe una funcionalidad en el menú que elimina todos los objetos relacionados con GXUnit, esto incluye todos los objetos Test Case y todas las Suites que hayan sido creados *(fig 15.6)*.

(fig 15.6 – Borrar los objetos relacionados a GXUnit)

Tool Window GXUnit

Para poder trabajar con la herramienta se debe habilitar la Tool Window GXUnit *(fig 15.7)*.

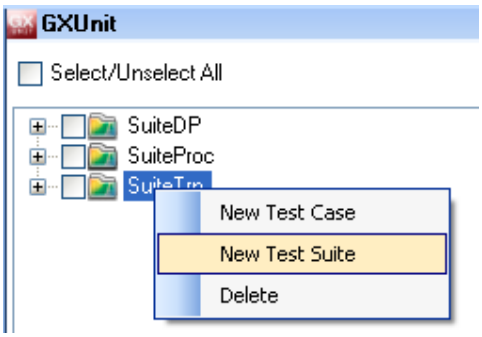


(fig 15.7 – Visualizar la Tool Window de GXUnit)

Suite de prueba

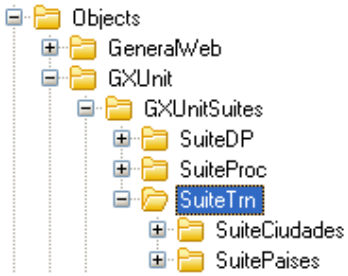
Una suite de pruebas tiene la funcionalidad de agrupar cierto conjunto de Casos de prueba según un criterio definido por el usuario.

Para crear una nueva Suite de Pruebas se debe hacer *click* derecho sobre una suite ya creada o sobre cualquier lugar de la Tool Window GXUnit para crear una Suite del primer Nivel *(fig 15.8)*.



(fig 15.8 – Crear una nueva Suite desde la Tool Window)

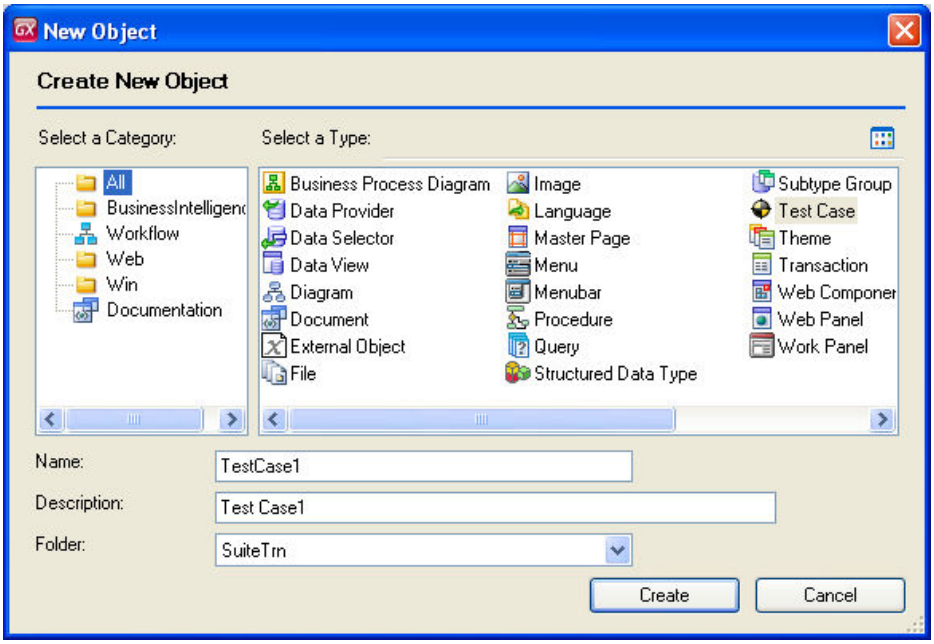
Luego de seleccionar Nueva Suite se pide el nombre de la misma y al ingresarlo se crea una nueva Folder GeneXus bajo la estructura Objects/GXUnitSuites (*fig 15.9*).



(fig 15.9 – Creación de los objetos Folder a partir de las suites)

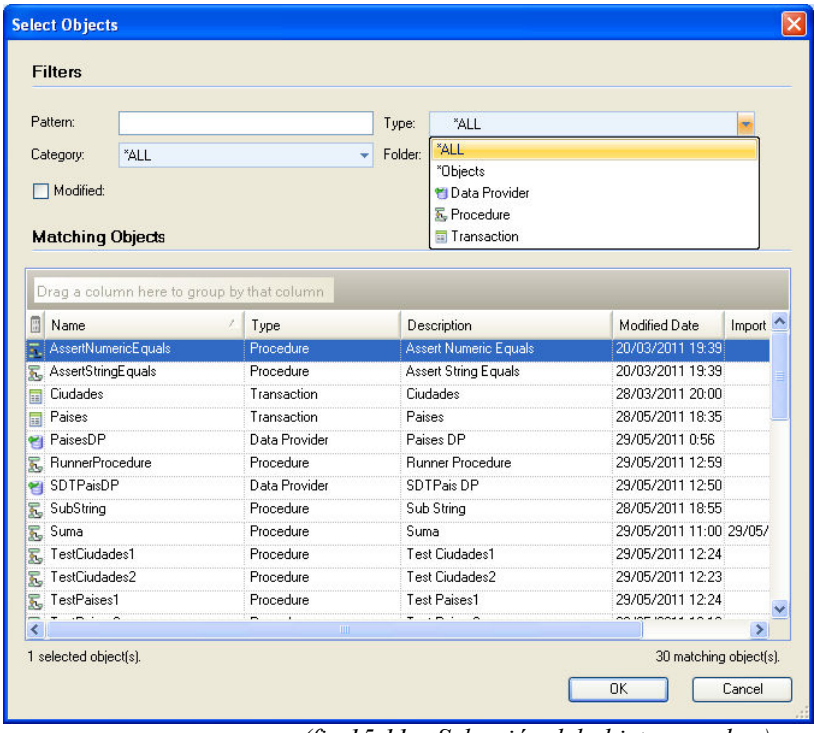
Caso de prueba

Para crear un nuevo Caso de Prueba se presiona *click* derecho en una Suite y se selecciona Nuevo Test Case. En el cuadro de dialogo de nuevo objeto GeneXus se debe seleccionar el objeto TestCase y como Folder queda seleccionada la Suite elegida *(fig 15.10)*.



(fig 15.10 – Creación de un nuevo Test Case)

Al crear un nuevo objeto de este tipo se pregunta a qué objeto está asociado. En esta versión se permiten seleccionar objetos Procedure, DataProvider o Transaction *(fig 15.11)*.



(fig 15.11 – Selección del objeto a probar)

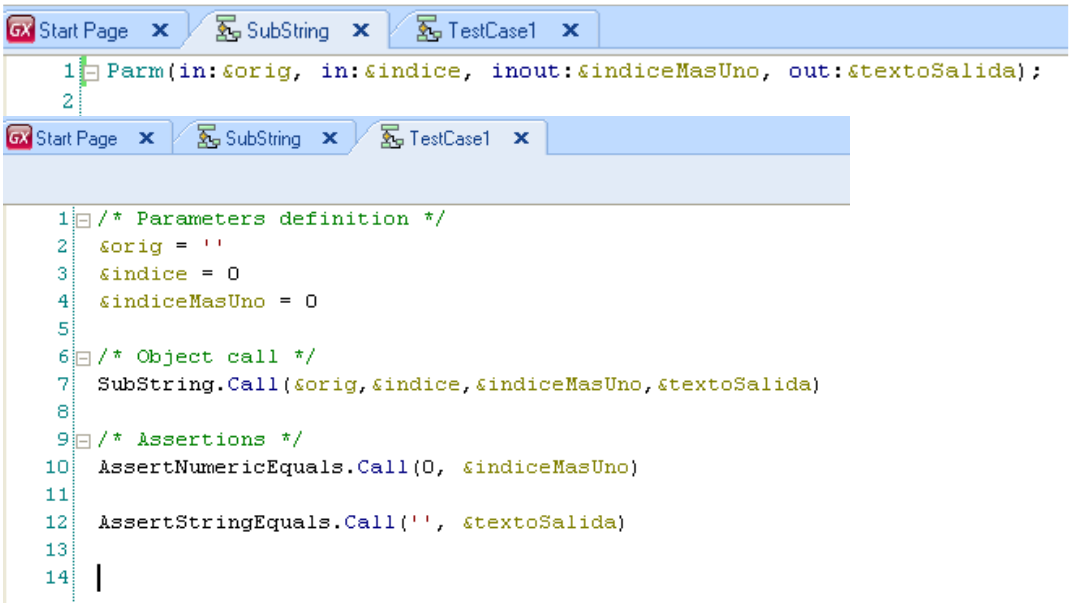
El objeto TestCase tiene un editor donde se puede utilizar la sintaxis del objeto Procedimiento para llamar a cualquier otro objeto GX y realizar las comparaciones llamando a los objetos de tipo *Assert* que se describen más adelante.

Al crearse un caso de prueba se genera automáticamente código GeneXus por defecto donde se

llama al objeto a probar y se realizan *asserts* para evaluar los resultados de dicha llamada. El objetivo es modificar este código para que se adapte a las necesidades del usuario.

Caso de prueba de Procedimiento

Para el caso de que el objeto asociado al caso de prueba sea un Procedimiento se crea una variable por cada una que se encuentre en la regla "Parm" del procedimiento a probar y según su tipo, si es de entrada o entrada/salida se inicializa con un valor por defecto o si es de salida o entrada/salida se evalúa con una llamada a un *Assert* (según el tipo) *(fig 15.12)*.

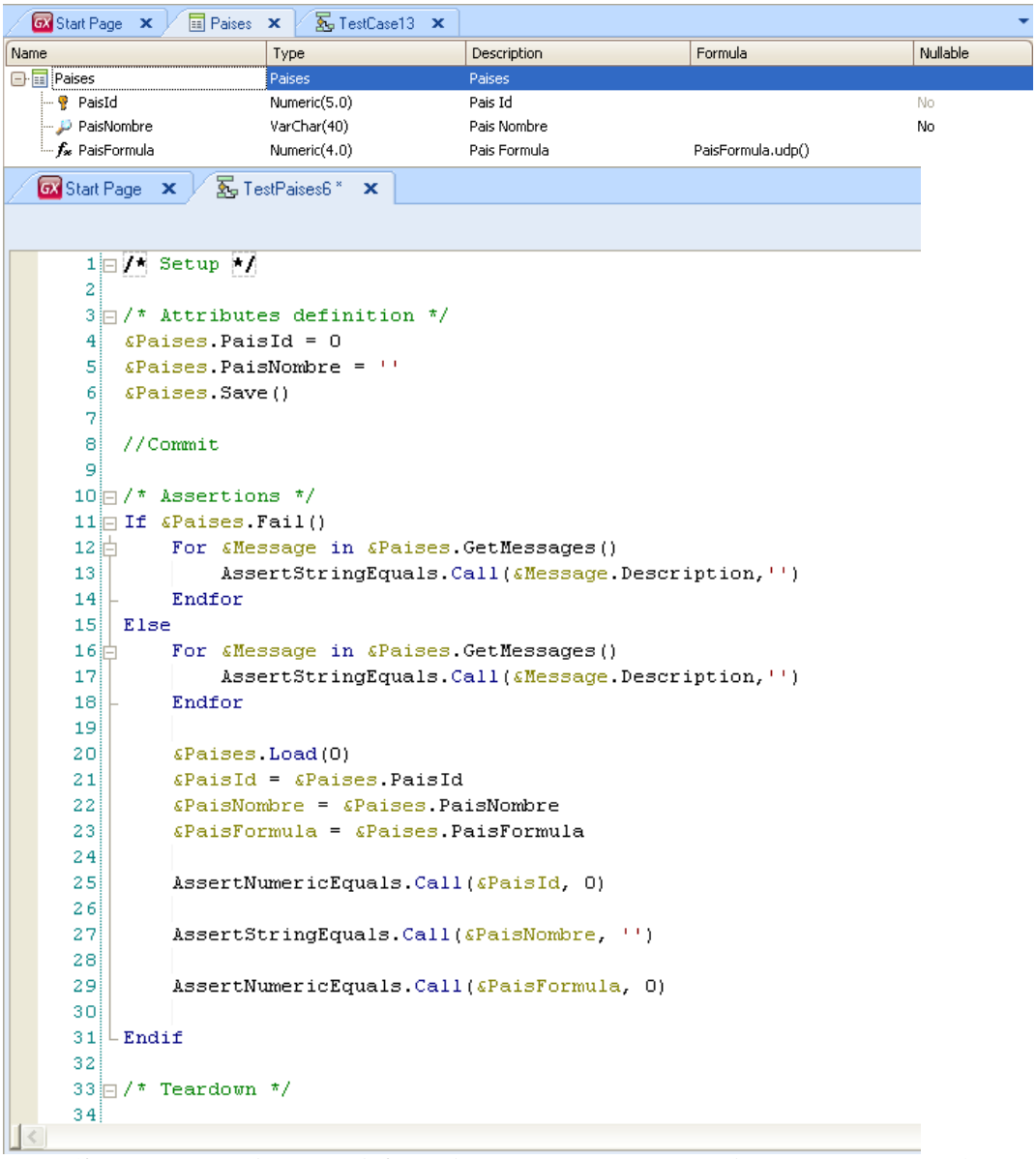


(fig 15.12 – Código por defecto de un TestCase que prueba a un Procedimiento)

Casos de prueba de Transacciones

Al crear un nuevo caso de prueba asociado a una transacción con la propiedad Business Component (BC) se genera código GeneXus por defecto de la siguiente manera *(fig 15.13)*:

- Se define una variable del tipo de la transacción BC
- Inicializa con un valor por defecto todos los atributos
- Guarda (*save* del BC)
- Se carga (*load* del BC)
- Se realiza un *assert* para cada mensaje de la Transacción
- Se realiza un *assert* por cada atributo del Business Component

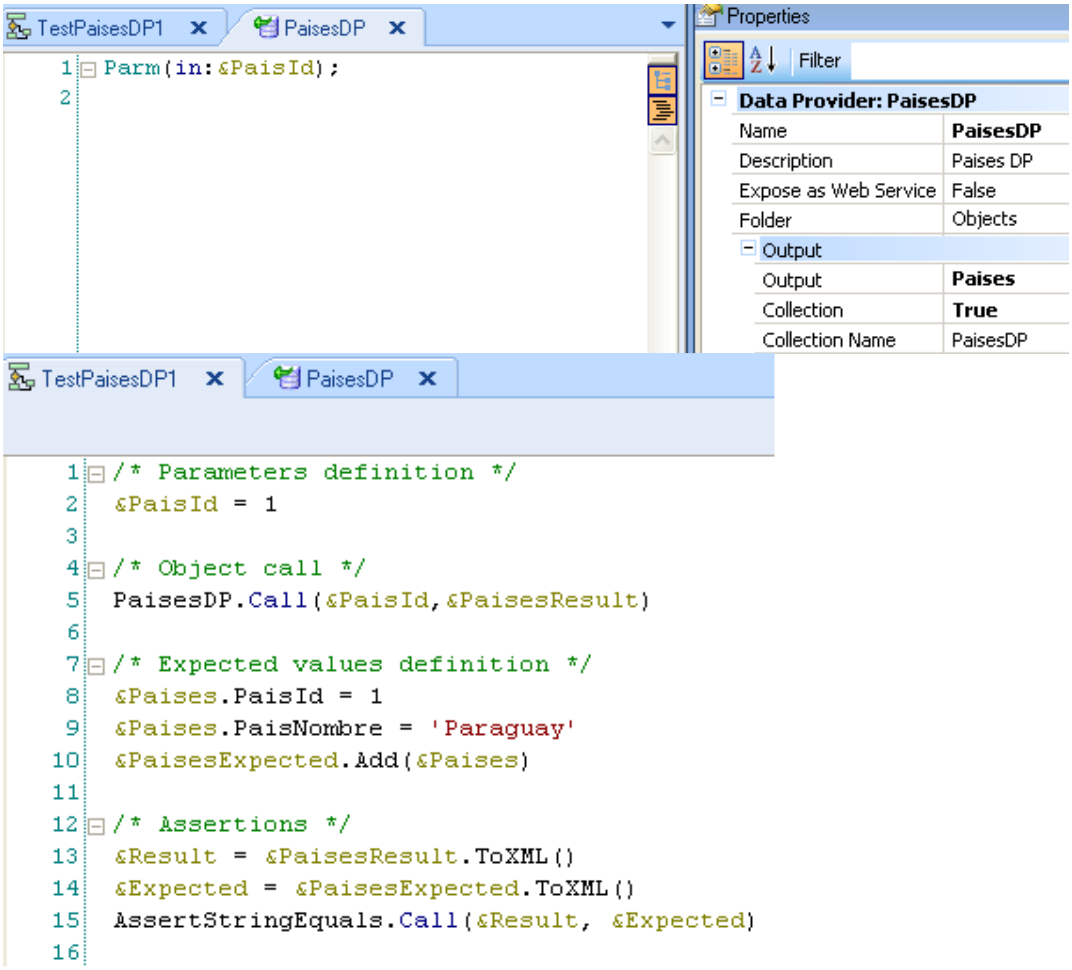


(fig 15.13 – Código por defecto de un TestCase que prueba una Transaccion)

Casos de prueba de Data Providers

Al crear un nuevo caso de prueba asociado a un Data Provider (DP) se genera código GeneXus por defecto de la siguiente manera *(fig 15.14)*:

- Se inicializan los parámetros de entrada del DP
- Se llama al DP
- Se inicializa el resultado esperado (BC o SDT)
- Se realiza un *AssertStringEquals* para comparar el Output esperado con el obtenido



(fig 15.14 – Código por defecto de un TestCase que prueba un Data Provider)

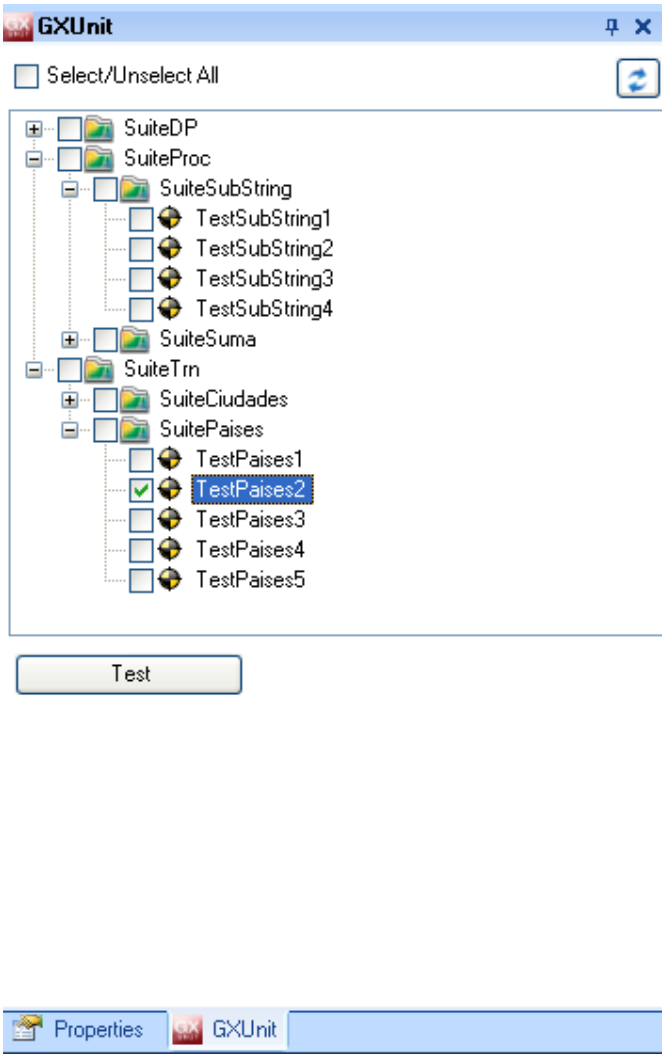
Asserts

Los métodos de tipo *Assert* son utilizados para comparar los resultados de la ejecución de objetos bajo prueba con los resultados que se esperan de dichos objetos. Fueron implementados como Procedimientos GeneXus. Se proveen dos de estos métodos, *AssertNumericEquals* y *AssertStringEquals*, para comparar variables numéricas y cadenas de caracteres respectivamente. Estos procedimientos son creados al inicializarse GXUnit y forman parte de la KB.

Al ejecutarse uno de estos métodos se guardan los resultados para luego poder visualizar todos los resultados de la ejecución de los casos de prueba.

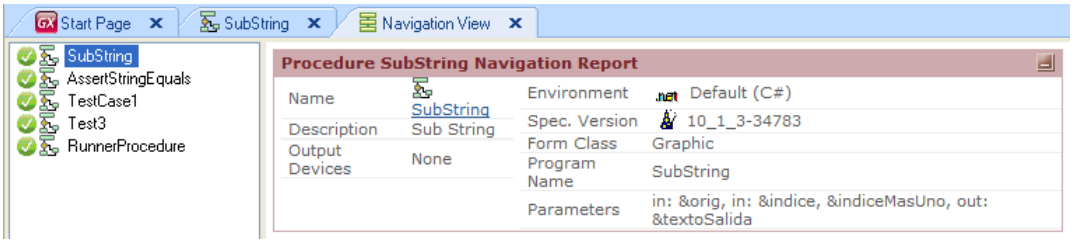
Ejecución de Casos de Prueba

La Tool Window contiene los conjuntos de Casos de Prueba agrupados en sus respectivas Suites*(fig 15.15)*.



(fig 15.15 – Suites y casos de prueba existentes)

Seleccionando los casos de prueba que quieren ser ejecutados (mediante su check box) al presionar el botón "Test" se crea un nuevo procedimiento (RunnerProcedure) que tiene las llamadas a todos los objetos de tipo TestCase a ejecutar, se generan todos los objetos necesarios y se ejecuta *(fig 15.16)*.

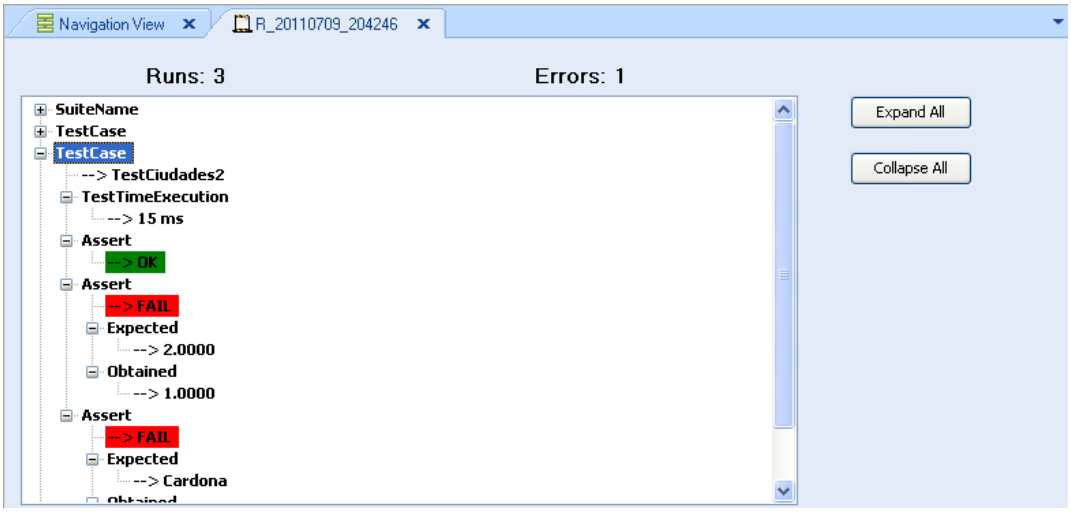


(fig 15.16 – Objetos generados)

Visualización de resultados

Luego de ejecutados los casos de prueba se crea un objeto Resultado donde queda registro del

resultado de la prueba. Este objeto tiene como nombre la fecha y hora de ejecución. En el cabezal se muestra los objetos de tipo TestCase ejecutados y la cantidad de ellos que fallaron. Los *Asserts* que fallen se muestra en rojo con el detalle de la falla y los que se completen satisfactoriamente se muestra en verde *(fig 15.17)*.



(fig 15.17 – Resultado de la ejecución de las pruebas)

Exportación e Importación de Casos de Prueba y Resultados

Tanto los Test Case, Suites como objetos Resultado se pueden exportar e importar entre las distintas Bases de Conocimiento que tengan instalada la herramienta como cualquier otro objeto GeneXus.

16. Anexo D - Glosario

Desarrollo dirigido por pruebas (TDD: Test-Driven Development)

Es una práctica de programación que involucra otras dos prácticas: Escribir las pruebas primero (Test First Development) y Refactorización (Refactoring). Para escribir las pruebas generalmente se utilizan las pruebas unitarias (Unit Test en inglés). En primer lugar, se escribe una prueba y se verifica que las pruebas fallen, luego se implementa el código que haga que la prueba pase satisfactoriamente y seguidamente se refactoriza el código escrito. El propósito del desarrollo guiado por pruebas es lograr un código limpio que funcione (Del inglés: Clean code that works). La idea es que los requerimientos sean traducidos a pruebas, de este modo, cuando las pruebas pasen se garantizará que los requerimientos se hayan implementado correctamente.

Desarrollo dirigido por pruebas de aceptación (ATDD: Acceptance Test Driven Development)

Los equipos que practican un desarrollo dirigido por pruebas de aceptación, definen pruebas de aceptación en colaboración mientras discuten cada una de las historias de cada uno. Esta práctica ayuda a dejar al descubierto ciertos supuestos que puedan generarse y confirmar que todos tienen un entendimiento en común de cuando está "terminado". Durante la implementación, el equipo técnico automatiza las pruebas de aceptación en lenguaje natural escribiendo código que los una al software emergente. De esta manera, las pruebas ATDD se convierten en requerimientos ejecutables[58].

Pruebas dirigidas por datos

Pruebas dirigidas por datos es un término utilizado en las pruebas de software para la creación de lógica de pruebas re-usable para reducir el costo del mantenimiento y mejorar el cubrimiento de las pruebas. Es una metodología utilizada en la automatización de pruebas donde scripts de pruebas son ejecutados y verificados basados en la información guardada en las bases de datos. Todo lo que pueda cambiar (ambiente, "end points", datos de pruebas, "locations", etc.) es separado del resto de la lógica de las pruebas (scripts) y colocado en una componente externa. Esta puede ser una configuración o un "dataset" de pruebas. Luego se pueden agregar más datos o se puede cambiar la configuración para re-usar la misma lógica de las pruebas y ejecutar múltiples escenarios.

Programación Extrema (XP: Extreme Programming)

Es una metodología de desarrollo de software que intenta mejorar la calidad del software y la rápida respuesta a requerimientos cambiantes por parte del cliente. Promueve frecuentes "releases" en cortos ciclos de desarrollo que intentan mejorar la productividad e introducen puntos donde se pueden adoptar nuevos requerimientos por parte del cliente.

Desarrollo dirigido por modelos

El desarrollo dirigido por modelos es una metodología de desarrollo de software que se enfoca en crear modelos o abstracciones, más cercanas a conceptos de un dominio particular que a algoritmos o conceptos de computación.

Pruebas dirigidas por palabras clave

Pruebas dirigidas por palabras clave (Keyword-driven testing o también conocido como table-driven testing o action-word testing) es una metodología de pruebas de software para la automatización de las pruebas que separa el proceso de creación de pruebas en dos etapas: la etapa de planificación y la etapa de implementación.

En la etapa de planificación se crean las diferentes palabras clave. Como ejemplo se puede ver una tabla que contiene una combinación de pasos para la prueba de un sistema de inicio de sesión.

Objeto	Acción	Datos
Campo de texto (dominio)	Escribir texto	< dominio >
Campo de texto (nombre de usuario)	Escribir texto	< nombre de usuario >
Campo de texto (contraseña)	Escribir texto	< contraseña >
Botón de inicio de sesión	*Click*	Un *click* izquierdo

Luego en la etapa de implementación en la mayoría de los casos (ya que depende del framework o herramienta utilizada) los implementadores crean una interfaz para las palabras clave como "controlar" o "escribir texto". Los diseñadores (que no tienen por qué saber programar) crean los casos basados en las palabras clave definidas en la etapa de planificación que ya han sido implementadas. La prueba es ejecutada mediante un *driver* que lee la palabra clave y ejecuta el código correspondiente.

Aunque la metodología puede ser utilizada para realizar pruebas de forma manual, es una técnica particularmente útil para realizar automatización de pruebas. Las ventajas que posee para realizar pruebas automatizadas son la reusabilidad y por lo tanto la facilidad de mantenimiento de las pruebas que son creadas en un nivel alto de abstracción.

Mocks

Los *mocks* en programación orientada a objetos son objetos que imitan el comportamiento de objetos reales de manera controlada. La forma en que se utiliza es la siguiente:

Supongamos un objeto A que llama a otro objeto B. Se quiere probar de forma unitaria el objeto A, por lo tanto no se quiere utilizar B para que su comportamiento no influya en la prueba. Lo que se hace entonces es crear un objeto *mock* B' que simula el comportamiento de B pero de una manera controlada para la prueba que se está utilizando. Entonces para la ejecución de la prueba en lugar de directamente utilizar B se utiliza su *mock* B'.

Robot Framework

Robot Framework es un framework genérico de automatización de pruebas de Keyword-driven testing para Acceptance level testing y Acceptance Test-driven development (ATDD). Posee una sintaxis tabular para crear casos de prueba y sus capacidades pueden ser extendidas por

librerías de pruebas implementadas tanto en Python como en Java. Los usuarios pueden crear nuevas keywords (utilizando las que ya existen), usando la misma sintaxis simple que se usa para crear casos de prueba[48].

Error

1 - La diferencia entre un valor o condición computada, observada, o medida y el valor o condición verdadera, especificada, o teóricamente correcta.

2 - Un paso, proceso o definición de datos incorrecta.

3 - Un resultado incorrecto.

4 - Una acción humana que produce un resultado incorrecto.

[IEEE Std 610.12-1990]

Falta

1 - Un defecto en un dispositivo o componente de hardware.

2 - Un paso, proceso o definición de datos incorrecta en un programa de computación.

[IEEE Std 610.12-1990]

Falla

La incapacidad de un sistema de realizar sus funciones requeridas con su performance especificada. [IEEE 610.12-1990]

Integración continua

La integración continua implementa procesos continuos donde se compila y ejecutan las pruebas de todo un proyecto frecuentemente (generalmente una vez al día). El modelo ideal de integración continua permite que la construcción y ejecución de pruebas sea realizada cada vez que el código cambia.

Framework

Un framework es una estructura de soporte, normalmente con módulos de software concretos, con base a la cual otro proyecto de software puede ser más fácilmente organizado y desarrollado. Típicamente, puede incluir soporte de programas, bibliotecas y un lenguaje interpretado entre otras herramientas, para así ayudar a desarrollar y unir los diferentes componentes de un proyecto.

FIT: Framework for Integrated Test

FIT es una herramienta open-source para pruebas automatizadas. Integra el trabajo de clientes, analistas, testers y desarrolladores. Los clientes proveen ejemplos de cómo el software debería comportarse. Los ejemplos son programados y automáticamente son controlados para verificar su correctitud. A su vez son formateados en tablas y guardados como html usando aplicaciones de oficina comunes como planillas de cálculo. Cuando FIT verifica el documento, crea una

copia y colorea la tabla con verde, rojo o amarillo dependiendo de si el software se comportó como se esperaba.

Criterio de selección

Un criterio de selección de pruebas es un medio para decidir cuál sería un conjunto adecuado de casos de prueba. Un criterio puede ser utilizado para seleccionar los casos de prueba o para evaluar si un conjunto de casos de prueba es apropiado.

Oráculo

Un oráculo es cualquier agente (humano o mecánico) que se utiliza para decidir si un programa se comportó como se esperaba en una prueba dada y de acuerdo a ello produce un veredicto de "Pass" o "Fail". Existen muchos tipos de oráculos y la automatización de los oráculos puede ser muy costosa.

Testeabilidad

El término "testeabilidad del software" tiene dos significados relacionados pero diferentes. Por un lado, se refiere al grado en que el software cumple un criterio de cubrimiento de cierta prueba. Por otro lado, es definido como la probabilidad, posiblemente medida estadísticamente, de que, si contiene fallas, éstas se expondrán durante la ejecución de determinadas pruebas. Ambas definiciones son importantes.

OpenEdge Advanced Business Language o Progress 4GL

Es un lenguaje de desarrollo de aplicaciones de negocio clasificado como un lenguaje de programación de cuarta generación. Usa una sintaxis similar al idioma inglés para simplificar el desarrollo de software. Hasta su versión 10.0 era llamado PROGRESS o Progress 4GL, pero con esta versión su nombre fue cambiado a OpenEdge Advanced Business Language (OpenEdge ABL) para superar la percepción negativa de la industria hacia los lenguajes 4GL.

MSBuild

MSBuild es una plataforma gratuita de Microsoft que permite construir proyectos de Visual Studio o ejecutar soluciones sin la necesidad de utilizar o siquiera tener instalado el entorno de desarrollo de Visual Studio.

17. Referencias

1. Artech. GeneXus. 1988; Available from: http://www.genexus.com/.
(Último acceso abril 2012)

2. Enrique Almeida Alejandro Araújo, Uruguay Larre Borges. Proyecto colaborativo GXUnit2006. Available from:
http://wiki.gxtechnical.com/commwiki/servlet/hwiki?GxUnit
(Último acceso abril 2012)

3. Facultad de Ingeniería UdelaR, Uruguay. GXUnit, Grupo I. 2007; Available from:
http://www.gxopen.com/gxopenrocha/servlet/hproject?721
(Último acceso abril 2012)

4. Facultad de Ingeniería UdelaR, Uruguay. GXUnit, Grupo II. 2007; Available from:
http://www.gxopen.com/gxopenrocha/servlet/hproject?720
(Último acceso abril 2012)

5. Almeida Enrique. En el proceso de crear un framework de testing. 2003.

6. Almeida Enrique. Software Testing: Tres enfoques para un mismo problema. Encuentro GeneXus2004.

7. Enrique Almeida Alejandro Araújo, Uruguay Larre Borges. Collaborative Projects. 2006; Available from:
http://www.concepto.com.uy/archivosvinculados/EncuentroGX2006CollaborativeProjects.ppt.
(Último acceso abril 2012)

8. Facultad de Ingeniería UdelaR, Uruguay. Proyecto de Ingeniería de Software. 2007.

9. Enrique Almeida Alejandro Araújo, Uruguay Larre Borges. Proyecto GXUnit. 2008; Available from: http://trac2.assembla.com/gxextensions/browser/trunk/GxUnit
(Último acceso abril 2012)

10. Araújo Alejandro. Especificación de un marco de pruebas asociado a GeneXus con adaptación de funcionalidades FIT: Universidad de la República, Uruguay; 2008.

11. Cunningham Ward. FIT - Framework for Integrated Tests. 2007; Available from:
http://fit.c2.com/
(Último acceso abril 2012)

12. G. Tassey. The Economic Impacts of Inadequate Infrastructure of Software Testing2002. Available from: http://www.nist.gov/director/planning/upload/report02-3.pdf
(Último acceso abril 2012)

13. Beizer Boris. Carta enviada a Swtest-discuss. Swtest-discuss1997.

14. Artech. GeneXus Platform Software Development Kit. Available from: http://wiki.gxtechnical.com/commwiki/servlet/hwiki?GeneXus+X%2FGeneXus+Platform+SDK,
(Último acceso abril 2012)

15. Artech. GeneXus MarketPlace. Available from: http://marketplace.genexus.com
(Último acceso abril 2012)

16. Society IEEE Computer. SWEBOK - Guide to the Software Engineering Body of Knowledge2004. Available from: http://www.computer.org/portal/web/swebok
(Último acceso abril 2012)

17. Definición de Pruebas de Software. Cem Kaner.

18. Sybase. Power Builder. Available from: http://www.powerbuilder.org/
(Último acceso abril 2012)

19. Guru Sofware. Premios SG Guía 20102010. Available from: http://sg.com.mx/content/view/1013
(Último acceso abril 2011)

20. Alcides Solano Gustavo Yong, Andrés Camacho. Introducción a los Lenguajes de Cuarta Generación (4GL). Available from: www.di-mare.com/adolfo/cursos/2007-1/pp-Intro4GL.pdf
(Último acceso abril 2012)

21. Breogán Gonda Nicolás Jodal. GeneXus: Filosofía2010. Available from: www.genexus.com/files/wp-filosofia-genexus?es
(Último acceso abril 2012)

22. Breogán Gonda Nicolás Jodal. Desarrollo basado en el conocimiento2007. Available from: http://www.genexus.com/portal/agxppdwn.aspx?2,59,1080,O,S,0,22885%3bS%3b1%3b2315,
(Último acceso abril 2012)

23. Artech. GXQuery. Available from: www.genexus.com/gxquery
(Último acceso abril 2012)

24. G. Meszaros. xUnit Test Patterns.Refactoring Test Code2007.

25. K. Beck. Test Driven Development by Example2002.

26. Wikipedia. Lista de frameworks de testeo unitario2011. Available from: http://en.wikipedia.org/wiki/List_of_unit_testing_frameworks
(Último acceso abril 2012)

27. Erich Gamma Kent Beck. JUnit. Available from: http://www.junit.org/
(Último acceso abril 2012)

28. Corporation Oracle. Java. Available from: http://www.java.com
(Último acceso abril 2012)

29. Aberdour Mark. Open Source Software Testing Tools, news and discussion. Available from: http://www.opensourcetesting.org
(Último acceso abril 2012)

30. Foundation Eclipse. Eclipse Project. Available from: http://www.eclipse.org/
(Último acceso abril 2012)

31. Corporation Oracle. NetBeans. Available from: http://netbeans.org/
(Último acceso abril 2012)

32. Erich Gamma Kent Beck. JUnit Assert methods. Available from: http://www.junit.org/apidocs/org/junit/Assert.html
(Último acceso abril 2012)

33. John Urberg Phil Yandel. PBUnit. Available from: http://sourceforge.net/projects/pbunit/
(Último acceso abril 2012)

34. Rehberg C. In A Nutshell.

35. P.C.Yandel. PBUnit Cooking Up Some Sample Tests.

36. project Mockrunner. Mockrunner. Available from: http://mockrunner.sourceforge.net/
(Último acceso abril 2012)

37. Microsoft. Visual Studio 2010. 2010; Available from: http://www.microsoft.com/spain/visualstudio/
(Último acceso abril 2012)

38. AG SAP. ABAP Unit. Available from: http://help.sap.com/saphelp_nw2004s/helpdata/en/a2/8a1b602e858645b8aac1559b638ea4/frameset.htm
(Último acceso abril 2012)

39. AG SAP. ABAP. Available from: http://www.sdn.sap.com/irj/sdn/abap
(Último acceso abril 2012)

40. AG SAP. SAP. Available from: http://www.sap.com/index.epx
(Último acceso abril 2012)

41. PeopleSoft Inc. PSUnit (Peoplesoft). Available from: https://blogs.oracle.com/peopletools/entry/psunit_unit_test_framework_for
(Último acceso abril 2012)

42. PeopleSoft Inc. Peoplesoft. Available from: http://blogs.oracle.com/peopletools/
(Último acceso abril 2012)

43. Corporation Progress Software. ProUnit. Available from: http://prounit.sourceforge.net/
(Último acceso abril 2012)

44. Corporation Progress Software. OpenEdge Advanced Business Language o Progress 4GL. Available from: http://web.progress.com/es-es/openedge/index.html

(Último acceso abril 2012)

45. Tremblay Henri. Easy mock. Available from: http://easymock.org/
(Último acceso abril 2012)

46. Pryce Nat. JMock. Available from: http://www.jmock.org/
(Último acceso abril 2012)

47. DbUnit. Available from: http://www.dbunit.org/
(Último acceso abril 2012)

48. Robot Framework. Available from: http://code.google.com/p/robotframework/
(Último acceso abril 2012)

49. Google. Google Trends. Available from: http://www.google.com/trends
(Último acceso abril 2012)

50. Binder Robert V. Testing Object-Oriented Systems: Models, Patterns, and Tools1999.

51. Artech. GXTechnical Wiki: GeneXus Extensions.

52. Artech. Encuentro Internacional GeneXus. Available from:
http://www.genexus.com/encuentro
(Último acceso abril 2012)

53. Nicolás Carro Juan Pablo Goyeni, Marcos Olivera. Probar GeneXus code con GeneXus code. 2011; Available from: http://www.genexus.com/encuentro2011/conferencia-materiales?es,0,,2417
(Último acceso abril 2012)

54. PIS Grupo 02 -. Informe de Verificación Unitaria GxUnit. 2011.

55. PIS Grupo 03 -. Verificación unitaria utilizando la herramienta GXUnit. 2011.

56. Abstracta. GXtest. Available from: http://abstracta.com.uy/
(Último acceso abril 2012)

57. Artech. GXTechnical Wiki: Daily Dilbert Extension. Available from:
http://wiki.gxtechnical.com/commwiki/servlet/hwiki?Daily+Dilbert,
(Último acceso abril 2012)

58. Pekka Klärck Elisabeth Hendrickson. Acceptance Test Driven Development (ATDD) in Practice. Available from: http://agile2009.agilealliance.org/node/641
(Último acceso abril 2012)